Neural Networks from Scratch in Python

Harrison Kinsley & Daniel Kukieła

Acknowledgements

Harrison Kinsley:

My wife, Stephanie, for her unfailing support and faith in me throughout the years. You've never doubted me.
Each and every viewer and person who supported this book and project. Without my audience, none of this would have been possible.
The Python programming community in general for being awesome!
Daniel Kukieła for your unwavering effort with this massive project that Neural Networks from Scratch became. From learning C++ to make mods in GTA V, to Python for various projects, to the calculus behind neural networks, there doesn't seem to be any problem you cannot solve and it is a pleasure to do this for a living with you. I look forward to seeing what's next!

Daniel Kukieła:

My son, Oskar, for his patience and understanding during the busy days. My wife, Katarzyna, for the boundless love, faith and support in all the things I do, have ever done, and plan to do, the sunlight during most stormy days and the morning coffee every single day.
Harrison for challenging me to learn Python then pushing me towards learning neural networks. For showing me that things do not have to be perfectly done, all the support, and making me a part of so many interesting projects including "let's make a tutorial on neural networks from scratch," which turned into one the biggest challenges of my life — this book. I wouldn't be at where I am now if all of that didn't happen.
The Python community for making me a better programmer and for helping me to improve my language skills.

Table of Contents

Acknowledgements **2**

Copyright **8**

License for Code **9**

Readme **10**

Introducing Neural Networks **11**

A Brief History 12
What is a Neural Network? 13

Coding Our First Neurons **25**

A Single Neuron 26
A Layer of Neurons 30
Tensors, Arrays and Vectors 34
Dot Product and Vector Addition 38
A Single Neuron with NumPy 40
A Batch of Data 44
Matrix Product 47
Transposition for the Matrix Product 50
A Layer of Neurons & Batch of Data w/ NumPy 54

Adding Layers **59**

Training Data 62
Dense Layer Class 66
Full code up to this point: 70

Activation Functions ***72***
The Step Activation Function 73
The Linear Activation Function 74
The Sigmoid Activation Function 75
The Rectified Linear Activation Function 76
Why Use Activation Functions? 77
Linear Activation in the Hidden Layers 79
ReLU Activation in a Pair of Neurons. 81
ReLU Activation in the Hidden Layers 85
ReLU Activation Function Code 95
The Softmax Activation Function 98
Full code up to this point: 108

Calculating Network Error with Loss ***111***
Categorical Cross-Entropy Loss 112
The Categorical Cross-Entropy Loss Class 123
Combining everything up to this point: 125
Accuracy Calculation 129

Introducing Optimization ***131***
Full code up to this point: 136

Derivatives ***139***
The Impact of a Parameter on the Output 140
The Slope 142
The Numerical Derivative 146
The Analytical Derivative 154
Summary. 164

Gradients, Partial Derivatives, and the Chain Rule ***166***
The Partial Derivative 167
The Partial Derivative of a Sum 168
The Partial Derivative of Multiplication. 170
The Partial Derivative of *Max* *172*
The Gradient 173
The Chain Rule 174
Summary. 178

Backpropagation ***180***
Categorical Cross-Entropy loss derivative 215

Categorical Cross-Entropy loss derivative code implementation 218
Softmax activation derivative 220
Softmax activation derivative code implementation 226
Common Categorical Cross-Entropy loss and Softmax activation derivative 230
Common Categorical Cross-Entropy loss and Softmax activation derivative - code implementation. 234
Full code up to this point: 243

Optimizers ***249***
Stochastic Gradient Descent (SGD) 250
Learning Rate 257
Learning Rate Decay 274
Stochastic Gradient Descent with Momentum 283
AdaGrad 293
RMSProp 298
Adam. 304
Full code up to this point: 309

Testing with Out-of-Sample Data ***321***

Validation Data ***328***

Training Dataset ***332***

L1 and L2 Regularization ***335***
Forward Pass. 336
Backward pass 340

Dropout ***361***
Forward Pass. 362
Backward Pass 369
The Code. 370

Binary Logistic Regression ***388***
Sigmoid Activation Function 389
Sigmoid Function Derivative 391
Sigmoid Function Code 395
Binary Cross-Entropy Loss 396
Binary Cross-Entropy Loss Derivative 398
Binary Cross-Entropy Code 401
Implementing Binary Logistic Regression and Binary Cross-Entropy Loss .404

Full code up to this point: 407

Regression ***423***

Linear Activation 425
Mean Squared Error Loss 426
Mean Squared Error Loss Derivative 427
Mean Squared Error (MSE) Loss Code 428
Mean Absolute Error Loss 429
Mean Absolute Error Loss Derivative 430
Mean Absolute Error Loss Code. 431
Accuracy in Regression 432
Regression Model Training 433
Full code up to this point: 458

Model Object ***475***

Full code up to this point: 512

A Real Dataset ***532***

Data preparation 534
Data loading 536
Data preprocessing. 543
Data Shuffling 546
Batches. 549
Training 563
Full code up to now: 570

Model Evaluation ***594***

Saving and Loading Models and Their Parameters ***601***

Retrieving Parameters 601
Setting Parameters 605
Saving Parameters 609
Loading Parameters 610
Saving the Model 612
Loading the Model. 615

Prediction / Inference ***617***

Full code: 633

Closing ***661***

Copyright

Copyright © 2020 Harrison Kinsley
Cover Design copyright © 2020 Harrison Kinsley

No part of this book may be reproduced in any form or by any electronic or mechanical means, with the following exceptions:

1. Brief quotations from the book.
2. Python Code/software (strings interpreted as logic with Python), which is housed under the MIT license, described on the next page.

License for Code

The Python code/software in this book is contained under the following MIT License:

Copyright © 2020 Sentdex, Kinsley Enterprises Inc., https://nnfs.io

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

Readme

The objective of this book is to break down an extremely complex topic, neural networks, into small pieces, consumable by anyone wishing to embark on this journey. Beyond breaking down this topic, the hope is to dramatically demystify neural networks. As you will soon see, this subject, when explored from scratch, can be an educational and engaging experience. This book is for anyone willing to put in the time to sit down and work through it. In return, you will gain a far deeper understanding than most when it comes to neural networks and deep learning.

This book will be easier to understand if you already have an understanding of Python or another programming language. Python is one of the most clear and understandable programming languages; we have no real interest in padding page counts and exhausting an entire first chapter with a basics of Python tutorial. If you need one, we suggest you start here: *https://pythonprogramming.net/python-fundamental-tutorials/*

To cite this material:

Harrison Kinsley & Daniel Kukieła. Neural Networks from Scratch (NNFS). https://nnfs.io

Chapter 1

Introducing Neural Networks

We begin with a general idea of what **neural networks** are and why you might be interested in them. Neural networks, also called **Artificial Neural Networks** (though it seems, in recent years, we've dropped the "artificial" part), are a type of machine learning often conflated with deep learning. The defining characteristic of a *deep* neural network is having two or more **hidden layers** — a concept that will be explained shortly, but these hidden layers are ones that the neural network controls. It's reasonably safe to say that most neural networks in use are a form of deep learning.

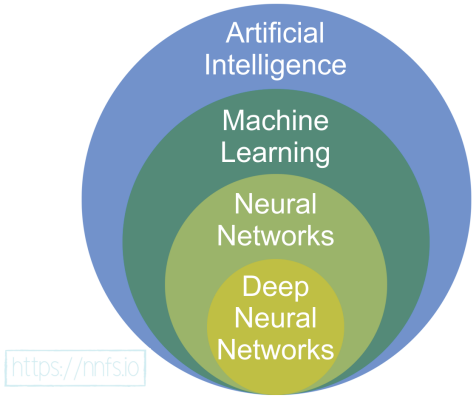


Fig 1.01: Depicting the various fields of artificial intelligence and where they fit in overall.

A Brief History

Since the advent of computers, scientists have been formulating ways to enable machines to take input and produce desired output for tasks like **classification** and **regression**. Additionally, in general, there's **supervised** and **unsupervised** machine learning. Supervised machine learning is used when you have pre-established and labeled data that can be used for training. Let's say you have sensor data for a server with metrics such as upload/download rates, temperature, and humidity, all organized by time for every 10 minutes. Normally, this server operates as intended and has no outages, but sometimes parts fail and cause an outage. We might collect data and then divide it into two classes: one class for times/observations when the server is operating normally, and another class for times/observations when the server is experiencing an outage. When the server is failing, we want to label that sensor data leading up to failure as data that preceded a failure. When the server is operating normally, we simply label that data as "normal."

What each sensor measures in this example is called a feature. A group of features makes up a feature set (represented as vectors/arrays), and the values of a feature set can be referred to as a sample. Samples are fed into neural network models to train them to fit desired outputs from these inputs or to predict based on them during the inference phase.

The "normal" and "failure" labels are **classifications** or **labels**. You may also see these referred to as **targets** or **ground-truths** while we fit a machine learning algorithm. These targets are the classifications that are the *goal* or *target*, known to be *true and correct*, for the algorithm to learn. For this example, the aim is to eventually train an algorithm to read sensor data and accurately predict when a failure is imminent. This is just one example of supervised learning in the form of classification. In addition to classification, there's also regression, which is used to predict numerical values, like stock prices. There's also unsupervised machine learning, where the machine finds structure in data without knowing the labels/classes ahead of time. There are additional concepts (e.g., reinforcement learning and semi-supervised machine learning) that fall under the umbrella of neural networks. For this book, we will focus on classification and regression with neural networks, but what we cover here leads to other use-cases.

Neural networks were conceived in the 1940s, but figuring out how to train them remained a mystery for 20 years. The concept of **backpropagation** (explained later) came in the 1960s, but neural networks still did not receive much attention until they started winning competitions in 2010. Since then, neural networks have been on a meteoric rise due to their sometimes seemingly

 nnfs_N3051 apineirom97@gmail.com

magical ability to solve problems previously deemed unsolvable, such as image captioning, language translation, audio and video synthesis, and more.

Currently, neural networks are the primary solution to most competitions and challenging technological problems like self-driving cars, calculating risk, detecting fraud, and early cancer detection, to name a few.

What is a Neural Network?

"Artificial" neural networks are inspired by the organic brain, translated to the computer. It's not a perfect comparison, but there are neurons, activations, and lots of interconnectivity, even if the underlying processes are quite different.

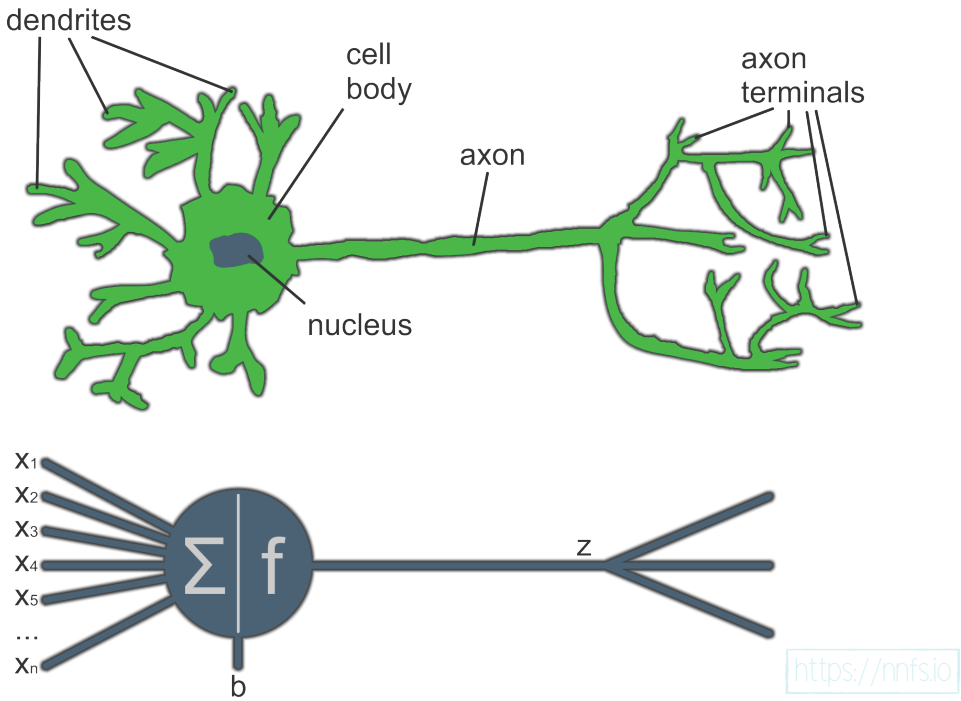


Fig 1.02: Comparing a biological neuron to an artificial neuron.

A single neuron by itself is relatively useless, but, when combined with hundreds or thousands (or many more) of other neurons, the interconnectivity produces relationships and results that frequently outperform any other machine learning methods.

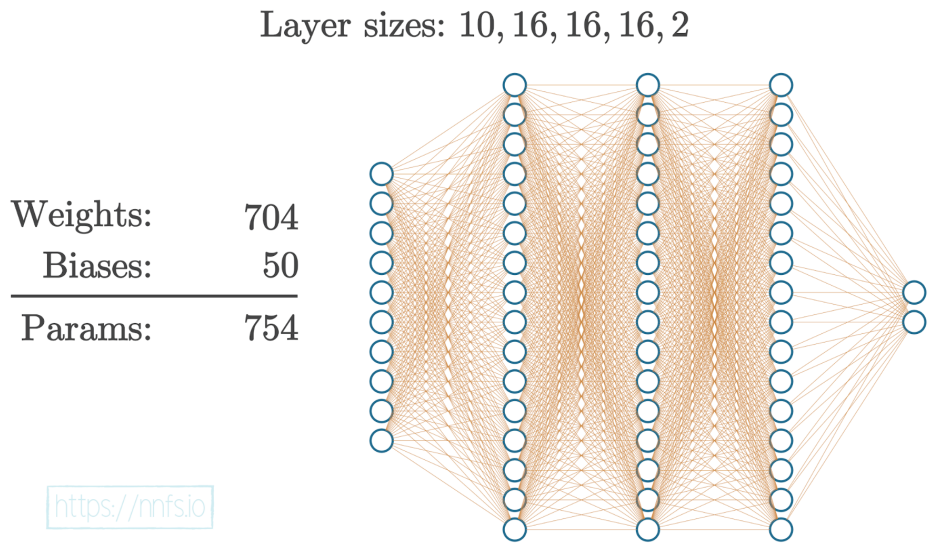


Fig 1.03: Example of a neural network with 3 hidden layers of 16 neurons each.

Anim 1.03: https://nnfs.io/ntr

The above animation shows the examples of the model structures and the numbers of parameters the model has to learn to adjust in order to produce the desired outputs. The details of what is seen here are the subjects of future chapters.

It might seem rather complicated when you look at it this way. Neural networks are considered to be "black boxes" in that we often have no idea *why* they reach the conclusions they do. We do understand *how* they do this, though.

Dense layers, the most common layers, consist of interconnected neurons. In a dense layer, each neuron of a given layer is connected to every neuron of the next layer, which means that its output value becomes an input for the next neurons. Each connection between neurons has a weight associated with it, which is a trainable factor of how much of this input to use, and this weight gets multiplied by the input value. Once all of the *inputs·weights* flow into our neuron, they are

summed, and a bias, another trainable parameter, is added. The purpose of the bias is to offset the output positively or negatively, which can further help us map more real-world types of dynamic data. In chapter 4, we will show some examples of how this works.

The concept of weights and biases can be thought of as "knobs" that we can tune to fit our model to data. In a neural network, we often have thousands or even millions of these parameters tuned by the optimizer during training. Some may ask, "why not just have biases or just weights?" Biases and weights are both tunable parameters, and both will impact the neurons' outputs, but they do so in different ways. Since weights are multiplied, they will only change the magnitude or even completely flip the sign from positive to negative, or vice versa. *Output* = *weight·input+bias* is not unlike the equation for a line $y = mx+b$. We can visualize this with:

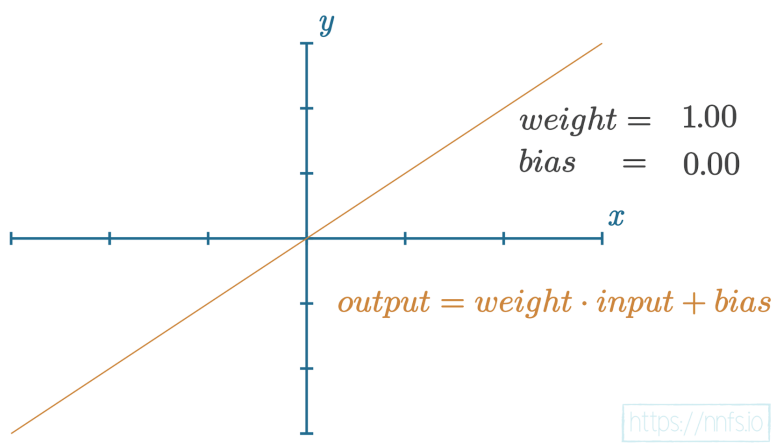


Fig 1.04: Graph of a single-input neuron's output with a weight of 1, bias of 0 and input *x*.

Adjusting the weight will impact the slope of the function:

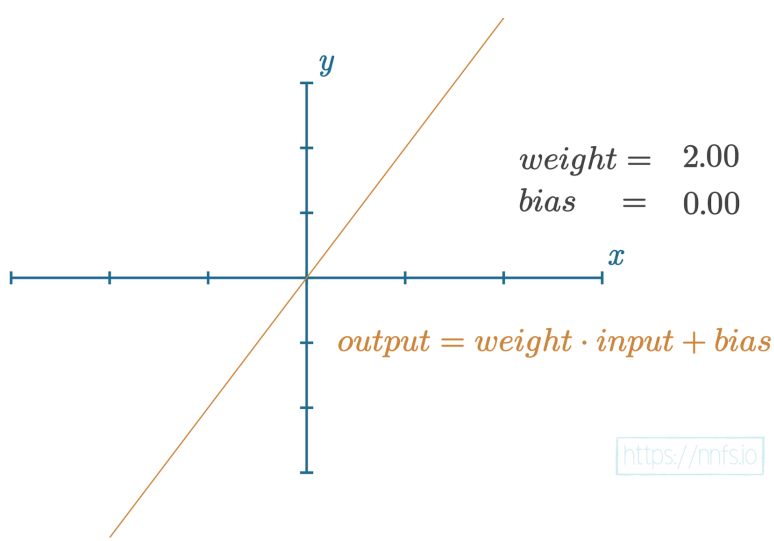


Fig 1.05: Graph of a single-input neuron's output with a weight of 2, bias of 0 and input *x*.

As we increase the value of the weight, the slope will get steeper. If we decrease the weight, the slope will decrease. If we negate the weight, the slope turns to a negative:

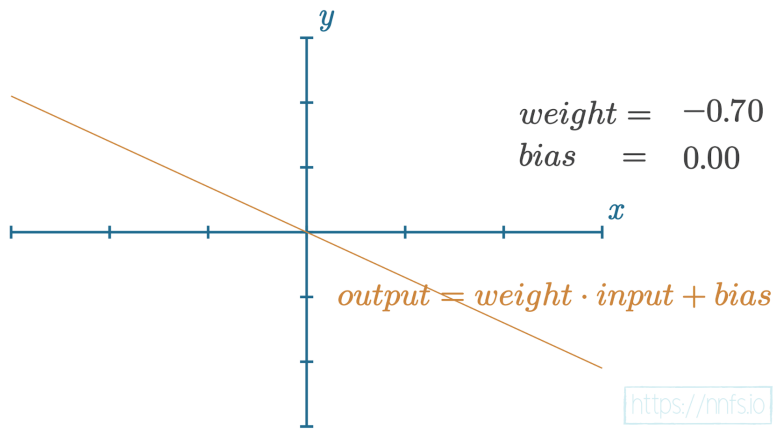


Fig 1.06: Graph of a single-input neuron's output with a weight of -0.70, bias of 0 and input *x*.

This should give you an idea of how the weight impacts the neuron's output value that we get from *inputs·weights+bias.* Now, how about the bias parameter? The bias offsets the overall function. For example, with a weight of 1.0 and a bias of 2.0:

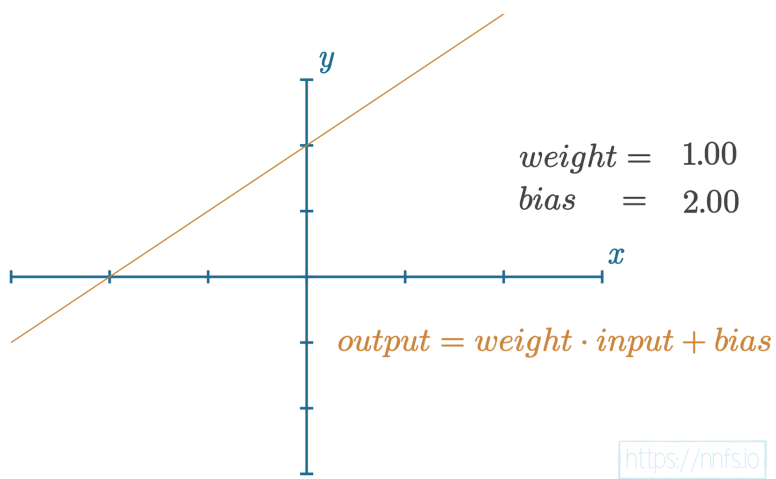


Fig 1.07: Graph of a single-input neuron's output with a weight of 1, bias of 2 and input *x*.

As we increase the bias, the function output overall shifts upward. If we decrease the bias, then the overall function output will move downward. For example, with a negative bias:

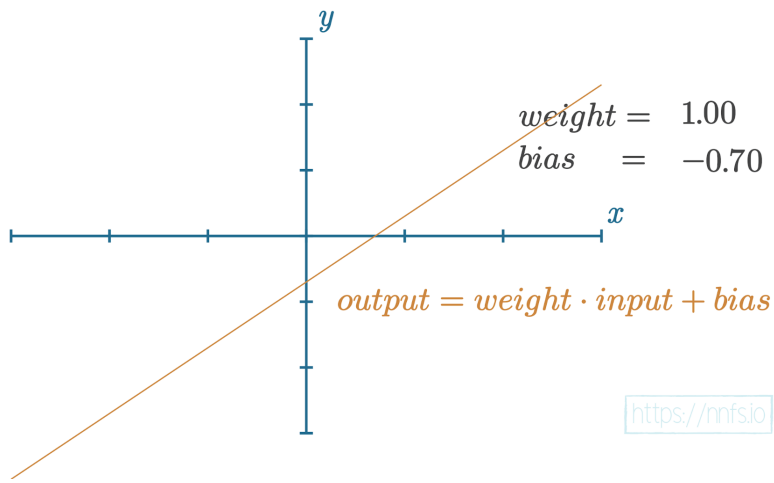


Fig 1.08: Graph of a single-input neuron's output with a weight of 1.0, bias of -0.70 and input *x*.

Anim 1.04-1.08: https://nnfs.io/bru

As you can see, weights and biases help to impact the outputs of neurons, but they do so in slightly different ways. This will make even more sense when we cover **activation functions** in chapter 4. Still, you can hopefully already see the differences between weights and biases and how they might individually help to influence output. Why this matters will be conveyed shortly.

As a very general overview, the step function meant to mimic a neuron in the brain, either "firing" or not — like an on-off switch. In programming, an on-off switch as a function would be called a **step function** because it looks like a step if we graph it.

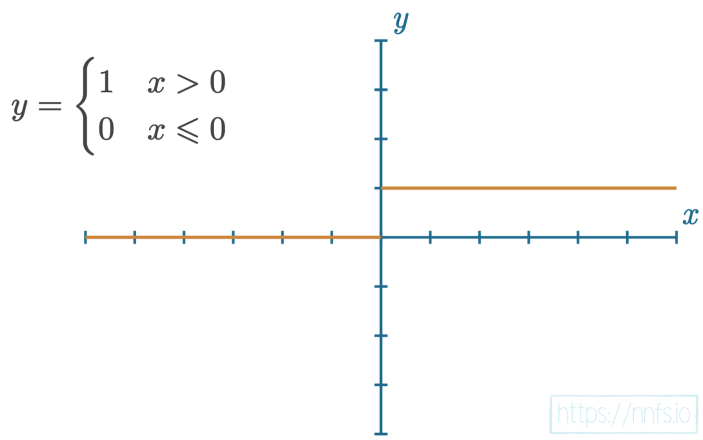


Fig 1.09: Graph of a step function.

For a step function, if the neuron's output value, which is calculated by *sum(inputs · weights) + bias*, is greater than 0, the neuron fires (so it would output a 1). Otherwise, it does not fire and would pass along a 0. The formula for a single neuron might look something like:

```
output = sum(inputs * weights) + bias
```

We then usually apply an activation function to this output, noted by *activation()*:

```
output = activation(output)
```

While you can use a step function for your activation function, we tend to use something slightly more advanced. Neural networks of today tend to use more informative activation functions (rather than a step function), such as the **Rectified Linear** (ReLU) activation function, which we will cover in-depth in Chapter 4. Each neuron's output could be a part of the ending output layer, as well as the input to another layer of neurons. While the full function of a neural network can get very large, let's start with a simple example with 2 hidden layers of 4 neurons each.

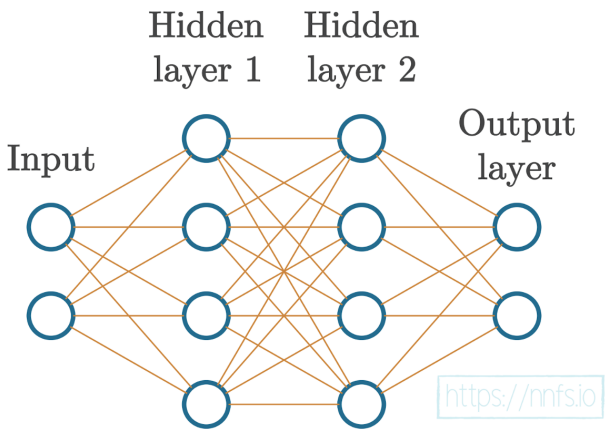


Fig 1.10: Example basic neural network.

Along with these 2 hidden layers, there are also two more layers here — the input and output layers. The input layer represents your actual input data, for example, pixel values from an image or data from a temperature sensor. While this data can be "raw" in the exact form it was collected, you will typically **preprocess** your data through functions like **normalization** and **scaling**, and your input needs to be in numeric form. Concepts like scaling and normalization will be covered later in this book. However, it is common to preprocess data while retaining its features and having the values in similar ranges between 0 and 1 or -1 and 1. To achieve this, you will use either or both scaling and normalization functions. The output layer is whatever the neural network returns. With classification, where we aim to predict the class of the input, the output layer often has as many neurons as the training dataset has classes, but can also have a single output neuron for binary (two classes) classification. We'll discuss this type of model later and, for now, focus on a classifier that uses a separate output neuron per each class. For example, if our goal is to classify a collection of pictures as a "dog" or "cat," then there are two classes in total. This means our output layer will consist of two neurons; one neuron associated with "dog" and the other with "cat." You could also have just a single output neuron that is "dog" or "not dog."

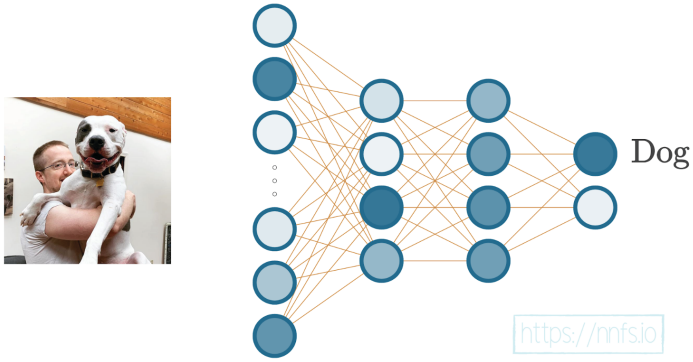


Fig 1.11: Visual depiction of passing image data through a neural network, getting a classification

For each image passed through this neural network, the final output will have a calculated value in the "cat" output neuron, and a calculated value in the "dog" output neuron. The output neuron that received the highest score becomes the class prediction for the image used as input.

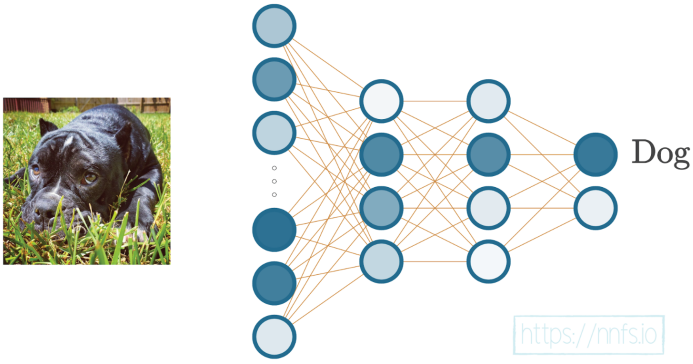


Fig 1.12: Visual depiction of passing image data through a neural network, getting a classification

Anim 1.11-1.12: https://nnfs.io/qtb

The thing that makes neural networks appear challenging is the math involved and how scary it can sometimes look. For example, let's imagine a neural network, and take a journey through what's going on during a simple forward pass of data, and the math behind it. Neural networks are really only a bunch of math equations that we, programmers, can turn into code. For this, do not worry about understanding everything. The idea here is to give you a high-level impression of what's going on overall. Then, this book's purpose is to break down each of these elements into painfully simple explanations, which will cover both forward and backward passes involved in training neural networks.

When represented as one giant function, an example of a neural network's forward pass would be computed with:

$$L = -\sum_{l=1}^{N} y_l log(\forall_{j=1}^{n_3} \frac{e^{\sum_{i=1}^{n_2}(\forall_{j=1}^{n_2} max(0,\sum_{i=1}^{n_1}(\forall_{j=1}^{n_1} max(0,\sum_{i=1}^{n_0} X_i w_{1,i,j}+b_{1,j}))_i w_{2,i,j}+b_{2,j}))_i w_{3,i,j}+b_{3,j}}{\sum_{k=1}^{n_3} e^{\sum_{i=1}^{n_2}(\forall_{j=1}^{n_2} max(0,\sum_{i=1}^{n_1}(\forall_{j=1}^{n_1} max(0,\sum_{i=1}^{n_0} X_i w_{1,i,j}+b_{1,k}))_i w_{2,i,j}+b_{2,k}))_i w_{3,i,k}+b_{3,k}}})$$

https://nnfs.io

Fig 1.13: Full formula for the forward pass of an example neural network model.

Anim 1.13: https://nnfs.io/vkt

Naturally, that looks extremely confusing, and the above is actually the easy part of neural networks. This turns people away, understandably. In this book, however, we're going to be coding everything from scratch, and, when doing this, you should find that there's no step along the way to producing the above function that is very challenging to understand. For example, the above function can also be represented in nested python functions like:

```
loss = -np.log(
    np.sum(
        y * np.exp(
            np.dot(
                np.maximum(
                    0,
                    np.dot(
                        np.maximum(
                            0,
                            np.dot(
                                X,
                                w1.T
                            ) + b1
                        ),
                        w2.T
                    ) + b2
                ),
                w3.T
            ) + b3
        ) /
        np.sum(
            np.exp(
                np.dot(
                    np.maximum(
                        0,
                        np.dot(
                            np.maximum(
                                0,
                                np.dot(
                                    X,
                                    w1.T
                                ) + b1
                            ),
                            w2.T
                        ) + b2
                    ),
                    w3.T
                ) + b3
            ),
            axis=1,
            keepdims=True
        )
    )
)
```

https://nnfs.io

Fig 1.14: Python code for the forward pass of an example neural network model.

There may be some functions there that you don't understand yet. For example, maybe you do not know what a log function is, but this is something simple that we'll cover. Then we have a sum operation, an exponentiating operation (again, you may not exactly know what this does, but it's nothing hard). Then we have a dot product, which is still just about understanding how it works, there's nothing there that is over your head if you know how multiplication works! Finally, we have some transposes, noted as .T, which, again, once you learn what that operation does, is not a challenging concept. Once we've separated each of these elements, learning what they do and how they work, suddenly, things will not appear to be as daunting or foreign. Nothing in this forward pass requires education beyond basic high school algebra! For an animation that depicts how all of this works in Python, you can check out the following animation, but it's certainly not expected that you'd immediately understand what's going on. The point is that this seemingly complex topic can be broken down into small, easy to understand parts, which is the purpose of the coming chapters!

Anim 1.14: https://nnfs.io/vkr

A typical neural network has thousands or even up to millions of adjustable **parameters** (weights and biases). In this way, neural networks act as enormous functions with vast numbers of **parameters**. The concept of a long function with millions of variables that could be used to solve a problem isn't all too difficult. With that many variables related to neurons, arranged as interconnected layers, we can imagine there exist some combinations of values for these variables that will yield desired outputs. Finding that combination of parameter (weight and bias) values is the challenging part.

The end goal for neural networks is to adjust their weights and biases (the parameters), so when applied to a yet-unseen example in the input, they produce the desired output. When supervised machine learning algorithms are trained, we show the algorithm examples of inputs and their associated desired outputs. One major issue with this concept is **overfitting** — when the algorithm only learns to fit the training data but doesn't actually "understand" anything about underlying input-output dependencies. The network basically just "memorizes" the training data.

Thus, we tend to use "in-sample" data to train a model and then use "out-of-sample" data to validate an algorithm (or a neural network model in our case). Certain percentages are set aside for both datasets to partition the data. For example, if there is a dataset of 100,000 samples of data and labels, you will immediately take 10,000 and set them aside to be your "out-of-sample" or "validation" data. You will then train your model with the other 90,000 in-sample or "training" data and finally validate your model with the 10,000 out-of-sample data that the model hasn't yet seen. The goal is for the model to not only accurately predict on the training data, but also to be similarly accurate while predicting on the withheld out-of-sample validation data.

This is called **generalization**, which means learning to fit the data instead of memorizing it. The idea is that we "train" (slowly adjusting weights and biases) a neural network on many examples of data. We then take out-of-sample data that the neural network has never been presented with and hope it can accurately predict on these data too.

You should now have a general understanding of what neural networks are, or at least what the objective is, and how we plan to meet this objective. To train these neural networks, we calculate

how “wrong” they are using algorithms to calculate the error (called **loss**), and attempt to slowly adjust their parameters (weights and biases) so that, over many iterations, the network gradually becomes less wrong. The goal of all neural networks is to generalize, meaning the network can see many examples of never-before-seen data, and accurately output the values we hope to achieve. Neural networks can be used for more than just classification. They can perform regression (predict a scalar, singular, value), clustering (assign unstructured data into groups), and many other tasks. Classification is just a common task for neural networks.

Supplementary Material: https://nnfs.io/ch1
Chapter code, further resources, and errata for this chapter.

Chapter 2

Coding Our First Neurons

While we assume that we're all beyond beginner programmers here, we will still try to start slowly and explain things the first time we see them. To begin, we will be using **Python 3.7** (although any version of Python 3+ will likely work). We will also be using **NumPy** after showing the pure-Python methods and Matplotlib for some visualizations. It should be the case that a huge variety of versions should work, but you may wish to match ours exactly to rule out any version issues. Specifically, we are using:

Python 3.7.5
NumPy 1.15.0
Matplotlib 3.1.1

Since this is a *Neural Networks from Scratch in Python* book, we will demonstrate how to do things without NumPy as well, but NumPy is Python's all-things-numbers package. Building from scratch is the point of this book though ignoring NumPy would be a disservice since it is among the most, if not the most, important and useful packages for data science in Python.

A Single Neuron

Let's say we have a single neuron, and there are three inputs to this neuron. As in most cases, when you initialize parameters in neural networks, our network will have weights initialized randomly, and biases set as zero to start. Why we do this will become apparent later on. The input will be either actual training data or the outputs of neurons from the previous layer in the neural network. We're just going to make up values to start with as input for now:

```
inputs = [1, 2, 3]
```

Each input also needs a weight associated with it. Inputs are the data that we pass into the model to get desired outputs, while the weights are the parameters that we'll tune later on to get these results. Weights are one of the types of values that change inside the model during the training phase, along with biases that also change during training. The values for weights and biases are what get "trained," and they are what make a model actually work (or not work). We'll start by making up weights for now. Let's say the first input, at index 0, which is a 1, has a weight of 0.2, the second input has a weight of 0.8, and the third input has a weight of -0.5. Our input and weights lists should now be:

```
inputs = [1, 2, 3]
weights = [0.2, 0.8, -0.5]
```

Next, we need the bias. At the moment, we're modeling a single neuron with three inputs. Since we're modeling a single neuron, we only have one bias, as there's just one bias value per neuron. The bias is an additional tunable value but is not associated with any input in contrast to the weights. We'll randomly select a value of 2 as the bias for this example:

```
inputs = [1, 2, 3]
weights = [0.2, 0.8, -0.5]
bias = 2
```

This neuron sums each input multiplied by that input's weight, then adds the bias. All the neuron does is take the fractions of inputs, where these fractions (weights) are the adjustable parameters, and adds another adjustable parameter — the bias — then outputs the result. Our output would be calculated up to this point like:

```
output = (inputs[0]*weights[0] +
          inputs[1]*weights[1] +
          inputs[2]*weights[2] + bias)

print(output)

>>>
2.3
```

The output here should be **2.3**. We will use >>> to denote output in this book.

```
inputs = [1.0, 2.0, 3.0]
weights = [0.2, 0.8, -0.5]
bias = 2.0

output = inputs[0]*weights[0] + inputs[1]*weights[1] + inputs[2]*weights[2] + bias
print(output)

>>> 2.3
```

Fig 2.01: Visualizing the code that makes up the math of a basic neuron.

Anim 2.01: https://nnfs.io/bkr

What might we need to change if we have 4 inputs, rather than the 3 we've just shown? Next to the additional input, we need to add an associated weight, which this new input will be multiplied with. We'll make up a value for this new weight as well. Code for this data could be:

```
inputs = [1.0, 2.0, 3.0, 2.5]
weights = [0.2, 0.8, -0.5, 1.0]
bias = 2.0
```

Which could be depicted visually as:

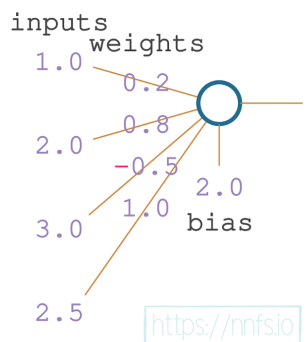


Fig 2.02: Visualizing how the inputs, weights, and biases from the code interact with the neuron.

Anim 2.02: https://nnfs.io/djp

All together in code, including the new input and weight, to produce output:

```
inputs = [1.0, 2.0, 3.0, 2.5]
weights = [0.2, 0.8, -0.5, 1.0]
bias = 2.0

output = (inputs[0]*weights[0] +
          inputs[1]*weights[1] +
          inputs[2]*weights[2] +
          inputs[3]*weights[3] + bias)
```

```
print(output)

>>>
4.8
```

Visually:

```
inputs = [1.0, 2.0, 3.0, 2.5]
weights = [0.2, 0.8, -0.5, 1.0]
bias = 2.0

output = inputs[0]*weights[0] + inputs[1]*weights[1] + inputs[2]*weights[2] + inputs[3]*weights[3] + bias
print(output)

>>> 4.8
```

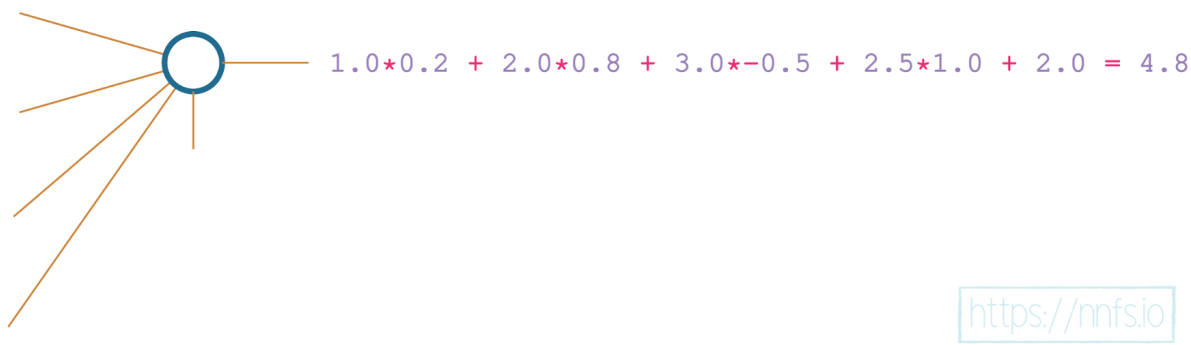


Fig 2.03: Visualizing the code that makes up a basic neuron, with 4 inputs this time.

Anim 2.03: https://nnfs.io/djp

A Layer of Neurons

Neural networks typically have layers that consist of more than one neuron. Layers are nothing more than groups of neurons. Each neuron in a layer takes exactly the same input — the input given to the layer (which can be either the training data or the output from the previous layer), but contains its own set of weights and its own bias, producing its own unique output. The layer's output is a set of each of these outputs — one per each neuron. Let's say we have a scenario with 3 neurons in a layer and 4 inputs:

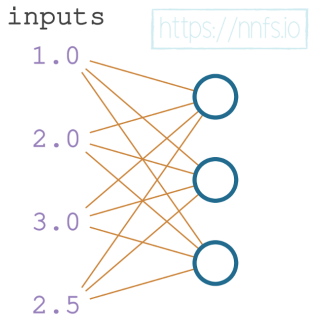


Fig 2.04: Visualizing a layer of neurons with common input.

Anim 2.04: https://nnfs.io/mxo

We'll keep the initial 4 inputs and set of weights for the first neuron the same as we've been using so far. We'll add 2 additional, made up, sets of weights and 2 additional biases to form 2 new neurons for a total of 3 in the layer. The layer's output is going to be a list of 3 values, not just a single value like for a single neuron.

```
inputs = [1, 2, 3, 2.5]

weights1 = [0.2, 0.8, -0.5, 1]
weights2 = [0.5, -0.91, 0.26, -0.5]
weights3 = [-0.26, -0.27, 0.17, 0.87]

bias1 = 2
bias2 = 3
bias3 = 0.5

outputs = [
        # Neuron 1:
        inputs[0]*weights1[0] +
        inputs[1]*weights1[1] +
        inputs[2]*weights1[2] +
        inputs[3]*weights1[3] + bias1,

        # Neuron 2:
        inputs[0]*weights2[0] +
        inputs[1]*weights2[1] +
        inputs[2]*weights2[2] +
        inputs[3]*weights2[3] + bias2,

        # Neuron 3:
        inputs[0]*weights3[0] +
        inputs[1]*weights3[1] +
        inputs[2]*weights3[2] +
        inputs[3]*weights3[3] + bias3]

print(outputs)

>>>
[4.8, 1.21, 2.385]
```

```
inputs = [1.0, 2.0, 3.0, 2.5]
weights1 = [0.2, 0.8, -0.5, 1.0]
weights2 = [0.5, -0.91, 0.26, -0.5]
weights3 = [-0.26, -0.27, 0.17, 0.87]
bias1 = 2.0
bias2 = 3.0
bias3 = 0.5

outputs = [
    inputs[0]*weights1[0] + inputs[1]*weights1[1] + inputs[2]*weights1[2] + inputs[3]*weights1[3] + bias1,
    inputs[0]*weights2[0] + inputs[1]*weights2[1] + inputs[2]*weights2[2] + inputs[3]*weights2[3] + bias2,
    inputs[0]*weights3[0] + inputs[1]*weights3[1] + inputs[2]*weights3[2] + inputs[3]*weights3[3] + bias3
]
print(outputs)

>>> [4.8, 1.21, 2.385]
```

https://nnfs.io

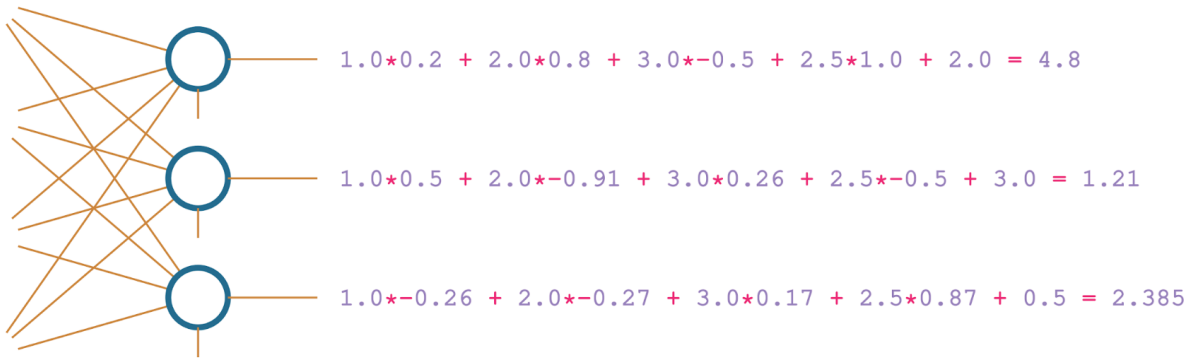


Fig 2.04.2: Code, math and visuals behind a layer of neurons.

Anim 2.04: https://nnfs.io/mxo

In this code, we have three sets of weights and three biases, which define three neurons. Each neuron is "connected" to the same inputs. The difference is in the separate weights and bias that each neuron applies to the input. This is called a **fully connected** neural network — every neuron in the current layer has connections to every neuron from the previous layer. This is a very common type of neural network, but it should be noted that there is no requirement to fully connect everything like this. At this point, we have only shown code for a single layer with very few neurons. Imagine coding many more layers and more neurons. This would get very challenging to code using our current methods. Instead, we could use a loop to scale and handle dynamically-sized inputs and layers. We've turned the separate weight variables into a list of weights so we can iterate over them, and we changed the code to use loops instead of the hardcoded operations.

```
inputs = [1, 2, 3, 2.5]
weights = [[0.2, 0.8, -0.5, 1],
           [0.5, -0.91, 0.26, -0.5],
           [-0.26, -0.27, 0.17, 0.87]]
biases = [2, 3, 0.5]

# Output of current layer
layer_outputs = []
# For each neuron
for neuron_weights, neuron_bias in zip(weights, biases):
    # Zeroed output of given neuron
    neuron_output = 0
    # For each input and weight to the neuron
    for n_input, weight in zip(inputs, neuron_weights):
        # Multiply this input by associated weight
        # and add to the neuron's output variable
        neuron_output += n_input*weight
    # Add bias
    neuron_output += neuron_bias
    # Put neuron's result to the layer's output list
    layer_outputs.append(neuron_output)

print(layer_outputs)
```

```
>>>
[4.8, 1.21, 2.385]
```

This does the same thing as before, just in a more dynamic and scalable way. If you find yourself confused at one of the steps, `print()` out the objects to see what they are and what's happening. The `zip()` function lets us iterate over multiple iterables (lists in this case) simultaneously. Again, all we're doing is, for each neuron (the outer loop in the code above, over neuron weights and biases), taking each input value multiplied by the associated weight for that input (the inner loop in the code above, over inputs and weights), adding all of these together, then adding a bias at the end. Finally, sending the neuron's output to the layer's output list.

That's it! How do we know we have three neurons? Why do we have three? We can tell we have three neurons because there are 3 sets of weights and 3 biases. When you make a neural network of your own, you also get to decide how many neurons you want for each of the layers. You can combine however many inputs you are given with however many neurons that you desire. As you progress through this book, you will gain some intuition of how many neurons to try using. We will start by using trivial numbers of neurons to aid in understanding how neural networks work at their core.

With our above code that uses loops, we could modify our number of inputs or neurons in our layer to be whatever we wanted, and our loop would handle it. As we said earlier, it would be a disservice not to show NumPy here since Python alone doesn't do matrix/tensor/array math very efficiently. But first, the reason the most popular deep learning library in Python is called "TensorFlow" is that it's all about doing operations on **tensors**.

Tensors, Arrays and Vectors

What are "tensors?"

Tensors are *closely-related to* arrays. If you interchange tensor/array/matrix when it comes to machine learning, people probably won't give you too hard of a time. But there are subtle differences, and they are primarily either the context or attributes of the tensor object. To understand a tensor, let's compare and describe some of the other data containers in Python (things that hold data). Let's start with a list. A Python list is defined by comma-separated objects contained in brackets. So far, we've been using lists.

This is an example of a simple list:

```
l = [1,5,6,2]
```

A list of lists:

```
lol = [[1,5,6,2],
       [3,2,1,3]]
```

A list of lists of lists!

```
lolol = [[[1,5,6,2],
          [3,2,1,3]],
         [[5,2,1,2],
          [6,4,8,4]],
         [[2,8,5,3],
          [1,1,9,4]]]
```

Everything shown so far could also be an array or an array representation of a tensor. A list is just a list, and it can do pretty much whatever it wants, including:

```
another_list_of_lists = [[4,2,3],
                         [5,1]]
```

The above list of lists cannot be an array because it is not **homologous**. A list of lists is homologous if each list along a dimension is identically long, and this must be true for each dimension. In the case of the list shown above, it's a 2-dimensional list. The first dimension's length is the number of sublists in the total list (2). The second dimension is the length of each of those sublists (3, then 2). In the above example, when reading across the "row" dimension (also called the second dimension), the first list is 3 elements long, and the second list is 2 elements long — this is not homologous and, therefore, cannot be an array. While failing to be consistent in one dimension is enough to show that this example is not homologous, we could also read down the "column" dimension (the first dimension); the first two columns are 2 elements long while the third column only contains 1 element. Note that every dimension does not necessarily need to be the same length; it is perfectly acceptable to have an array with 4 rows and 3 columns (i.e., 4x3).

A matrix is pretty simple. It's a rectangular array. It has columns and rows. It is two dimensional. So a matrix can be an array (a 2D array). Can all arrays be matrices? No. An array can be far more than just columns and rows, as it could have four dimensions, twenty dimensions, and so on.

```
list_matrix_array = [[4,2],
                     [5,1],
                     [8,2]]
```

The above list could also be a valid matrix (because of its columns and rows), which automatically means it could also be an array. The "shape" of this array would be 3x2, or more formally described as a shape of *(3, 2)* as it has 3 rows and 2 columns.

To denote a shape, we need to check every dimension. As we've already learned, a matrix is a 2-dimensional array. The first dimension is what's inside the most outer brackets, and if we look at the above matrix, we can see 3 lists there: `[4,2]`, `[5,1]`, and `[8,2]`; thus, the size in this dimension is 3 and each of those lists has to be the same shape to form an array (and matrix in this case). The next dimension's size is the number of elements inside this more inner pair of brackets, and we see that it's 2 as all of them contain 2 elements.

With 3-dimensional arrays, like in *lolol* below, we'll have a 3rd level of brackets:

```
lolol = [[[1,5,6,2],
          [3,2,1,3]],
         [[5,2,1,2],
          [6,4,8,4]],
         [[2,8,5,3],
          [1,1,9,4]]]
```

The first level of this array contains 3 matrices:

```
[[1,5,6,2],
  [3,2,1,3]]

[[5,2,1,2],
 [6,4,8,4]]
```

And

```
[[2,8,5,3],
 [1,1,9,4]]
```

That's what's inside the most outer brackets and the size of this dimension is then 3. If we look at the first matrix, we can see that it contains 2 lists — `[1,5,6,2]` and `[3,2,1,3]` so the size of this dimension is 2 — while each list of this inner matrix includes 4 elements. These 4 elements make up the 3rd and last dimension of this matrix since there are no more inner brackets. Therefore, the shape of this array is *(3, 2, 4)* and it's a 3-dimensional array, since the shape contains 3 dimensions.

Array :

```
lolol = [[[1,5,6,2],
          [3,2,1,3]],
         [[5,2,1,2],
          [6,4,8,4]],
         [[2,8,5,3],
          [1,1,9,4]]]
```

https://nnfs.io

Shape :

(3, 2, 4)

Type :

3D Array

Fig 2.05: Example of a 3-dimensional array.

Anim 2.05: https://nnfs.io/jps

Finally, what's a tensor? When it comes to the discussion of tensors versus arrays in the context of computer science, pages and pages of debate have ensued. This intense debate appears to be caused by the fact that people are arguing from entirely different places. There's no question that a tensor is not just an array, but the real question is: "What is a tensor, to a computer scientist, in the context of deep learning?" We believe that we can solve the debate in one line:

A tensor object is an object that can be represented as an array.

What this means is, as programmers, we can (and will) treat tensors as arrays in the context of deep learning, and that's really all the thought we have to put into it. Are all tensors *just* arrays? No, but they are represented as arrays in our code, so, to us, they're only arrays, and this is why there's so much argument and confusion.

Now, what is an array? In this book, we define an array as an ordered homologous container for numbers, and mostly use this term when working with the NumPy package since that's what the main data structure is called within it. A linear array, also called a 1-dimensional array, is the simplest example of an array, and in plain Python, this would be a list. Arrays can also consist of multi-dimensional data, and one of the best-known examples is what we call a matrix in mathematics, which we'll represent as a 2-dimensional array. Each element of the array can be accessed using a tuple of indices as a key, which means that we can retrieve any array element.

We need to learn one more notion — a vector. Put simply, a vector in math is what we call a list in Python or a 1-dimensional array in NumPy. Of course, lists and NumPy arrays do not have the same properties as a vector, but, just as we can write a matrix as a list of lists in Python, we can also write a vector as a list or an array! Additionally, we'll look at the vector algebraically (mathematically) as a set of numbers in brackets. This is in contrast to the physics perspective, where the vector's representation is usually seen as an arrow, characterized by a magnitude and a direction.

Dot Product and Vector Addition

Let's now address vector multiplication, as that's one of the most important operations we'll perform on vectors. We can achieve the same result as in our pure Python implementation of multiplying each element in our inputs and weights vectors element-wise by using a **dot product**, which we'll explain shortly. Traditionally, we use dot products for **vectors** (yet another name for a container), and we can certainly refer to what we're doing here as working with vectors just as we can call them "tensors." Nevertheless, this seems to add to the mysticism of neural networks — like they're these objects out in a complex multi-dimensional vector space that we'll never understand. Keep thinking of vectors as arrays — a 1-dimensional array is just a vector (or a list in Python).

Because of the sheer number of variables and interconnections made, we can model very complex and non-linear relationships with non-linear activation functions, and truly feel like wizards, but this might do more harm than good. Yes, we will be using the "dot product," but we're doing this because it results in a clean way to perform the necessary calculations. It's nothing more in-depth than that — as you've already seen, we can do this math with far more rudimentary-sounding words. When multiplying vectors, you either perform a dot product or a cross product. A cross product results in a vector while a dot product results in a scalar (a single value/number).

First, let's explain what a dot product of two vectors is. Mathematicians would say:

$$\vec{a} \cdot \vec{b} = \sum_{i=1}^{n} a_i b_i = a_1 b_1 + a_2 b_2 + \ldots + a_n b_n$$

A dot product of two vectors is a sum of products of consecutive vector elements. Both vectors must be of the same size (have an equal number of elements).

Let's write out how a dot product is calculated in Python. For it, you have two vectors, which we can represent as lists in Python. We then multiply their elements from the same index values and then add all of the resulting products. Say we have two lists acting as our vectors:

```
a = [1, 2, 3]
b = [2, 3, 4]
```

To obtain the dot product:

```
dot_product = a[0]*b[0] + a[1]*b[1] + a[2]*b[2]
print(dot_product)

>>>
20
```

```
a = [1, 2, 3]
b = [2, 3, 4]

dot_product = a[0]*b[0] + a[1]*b[1] + a[2]*b[2]
>>> 20
```

$$\vec{a} \cdot \vec{b} = [1, 2, 3] \cdot [2, 3, 4] = 1 \cdot 2 + 2 \cdot 3 + 3 \cdot 4 = 20$$

https://nnfs.io

Fig 2.06: Math behind the dot product example.

Anim 2.06: https://nnfs.io/xpo

Now, what if we called *a* "inputs" and *b* "weights?" Suddenly, this dot product looks like a succinct way to perform the operations we need and have already performed in plain Python. We need to multiply our weights and inputs of the same index values and add the resulting values together. The dot product performs this exact type of operation; thus, it makes lots of sense to use here. Returning to the neural network code, let's make use of this dot product. Plain Python does not contain methods or functions to perform such an operation, so we'll use the NumPy package, which is capable of this, and many more operations that we'll use in the future.

We'll also need to perform a vector addition operation in the not-too-distant future. Fortunately, NumPy lets us perform this in a natural way — using the plus sign with the variables containing vectors of the data. The addition of the two vectors is an operation performed element-wise, which means that both vectors have to be of the same size, and the result will become a vector of this

size as well. The result is a vector calculated as a sum of the consecutive vector elements:

$$\vec{a} + \vec{b} = [a_1 + b_1, a_2 + b_2, \ldots, a_n + b_n]$$

A Single Neuron with NumPy

Let's code the solution, for a single neuron to start, using the dot product and the addition of the vectors with NumPy. This makes the code much simpler to read and write (and faster to run):

```
import numpy as np

inputs = [1.0, 2.0, 3.0, 2.5]
weights = [0.2, 0.8, -0.5, 1.0]
bias = 2.0


outputs = np.dot(weights, inputs) + bias

print(outputs)

>>>
4.8
```

```
inputs = [1.0, 2.0, 3.0, 2.5]
weights = [0.2, 0.8, -0.5, 1.0]
bias = 2.0

outputs = np.dot(weights, inputs) + bias
```

https://nnfs.io

```
np.dot([0.2, 0.8, -0.5, 1.0], [1.0, 2.0, 3.0, 2.5]) =
= 0.2*1.0 + 0.8*2.0 + -0.5*3.0 + 1.0*2.5 = 2.8
```

Fig 2.07: Visualizing the math of the dot product of inputs and weights for a single neuron.

```
inputs = [1.0, 2.0, 3.0, 2.5]
weights = [0.2, 0.8, -0.5, 1.0]
bias = 2.0

outputs = np.dot(weights, inputs) + bias
>>> 4.8
```

https://nnfs.io

```
np.dot(weights, inputs) + bias = 2.8 + 2.0 = 4.8
```

Fig 2.08: Visualizing the math summing the dot product and bias.

Anim 2.07-2.08: https://nnfs.io/blq

A Layer of Neurons with NumPy

Now we're back to the point where we'd like to calculate the output of a layer of 3 neurons, which means the weights will be a matrix or list of weight vectors. In plain Python, we wrote this as a list of lists. With NumPy, this will be a 2-dimensional array, which we'll call a matrix. Previously with the 3-neuron example, we performed a multiplication of those weights with a list containing inputs, which resulted in a list of output values — one per each neuron.

We also described the dot product of two vectors, but the weights are now a matrix, and we need to perform a dot product of them and the input vector. NumPy makes this very easy for us — treating this matrix as a list of vectors and performing the dot product one by one with the vector of inputs, returning a list of dot products.

The dot product's result, in our case, is a vector (or a list) of sums of the weight and bias products for each of the neurons. From here, we still need to add corresponding biases to them. The biases can be easily added to the result of the dot product operation as they are a vector of the same size. We can also use the plain Python list directly here, as NumPy will convert it to an array internally.

Previously, we had calculated outputs of each neuron by performing a dot product and adding a bias, one by one. Now we have changed the order of those operations — we're performing dot product first as one operation on all neurons and inputs, and then we are adding a bias in the next operation. When we add two vectors using NumPy, each i-th element is added together, resulting in a new vector of the same size. This is both a simplification and an optimization, giving us simpler and faster code.

```
import numpy as np


inputs = [1.0, 2.0, 3.0, 2.5]
weights = [[0.2, 0.8, -0.5, 1],
           [0.5, -0.91, 0.26, -0.5],
           [-0.26, -0.27, 0.17, 0.87]]
biases = [2.0, 3.0, 0.5]


layer_outputs = np.dot(weights, inputs) + biases

print(layer_outputs)

>>>
array([4.8   1.21  2.385])
```

```
inputs = [1.0, 2.0, 3.0, 2.5]
weights = [[0.2, 0.8, -0.5, 1.0],
           [0.5, -0.91, 0.26, -0.5],
           [-0.26, -0.27, 0.17, 0.87]]
biases = [2.0, 3.0, 0.5]

outputs = np.dot(weights, inputs) + biases


np.dot(weights, inputs) = [np.dot(weights[0], inputs),
np.dot(weights[1], inputs), np.dot(weights[2], inputs)]
= [2.8, -1.79, 1.885]
```

https://nnfs.io

Fig 2.09: Code and visuals for the dot product applied to the layer of neurons.

```
inputs = [1.0, 2.0, 3.0, 2.5]
weights = [[0.2, 0.8, -0.5, 1.0],
           [0.5, -0.91, 0.26, -0.5],
           [-0.26, -0.27, 0.17, 0.87]]
biases = [2.0, 3.0, 0.5]

outputs = np.dot(weights, inputs) + biases
>>> array([4.8   1.21  2.385])
```

https://nnfs.io

Fig 2.10: Code and visuals for the sum of the dot product and bias with a layer of neurons.

Anim 2.09-2.10: https://nnfs.io/cyx

This syntax involving the dot product of weights and inputs followed by the vector addition of bias is the most commonly used way to represent this calculation of *inputs·weights+bias*. To explain the order of parameters we are passing into *np.dot()*, we should think of it as whatever comes first will decide the output shape. In our case, we are passing a list of neuron weights first and then the inputs, as our goal is to get a list of neuron outputs. As we mentioned, a dot product of a matrix and a vector results in a list of dot products. The *np.dot()* method treats the matrix as a list of vectors and performs a dot product of each of those vectors with the other vector. In this example, we used that property to pass a matrix, which was a list of neuron weight vectors and a vector of inputs and get a list of dot products — neuron outputs.

A Batch of Data

To train, neural networks tend to receive data in **batches**. So far, the example input data have been only one sample (or **observation**) of various features called a feature set:

```
inputs = [1, 2, 3, 2.5]
```

Here, the `[1, 2, 3, 2.5]` data are somehow meaningful and descriptive to the output we desire. Imagine each number as a value from a different sensor, from the example in chapter 1, all simultaneously. Each of these values is a feature observation datum, and together they form a **feature set instance**, also called an **observation**, or most commonly, a **sample**.

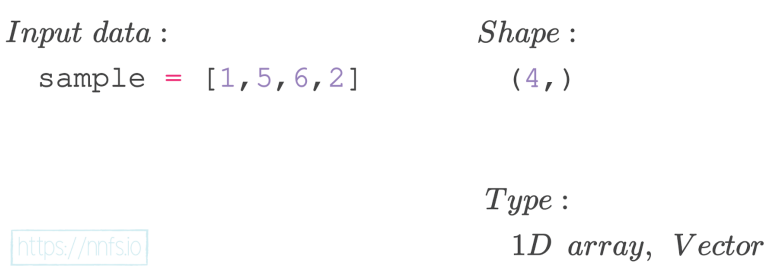


Fig 2.11: Visualizing a 1D array.

Anim 2.11: https://nnfs.io/lqw

Often, neural networks expect to take in many **samples** at a time for two reasons. One reason is that it's faster to train in batches in parallel processing, and the other reason is that batches

help with generalization during training. If you fit (perform a step of a training process) on one sample at a time, you're highly likely to keep fitting to that individual sample, rather than slowly producing general tweaks to weights and biases that fit the entire dataset. Fitting or training in batches gives you a higher chance of making more meaningful changes to weights and biases. For the concept of fitment in batches rather than one sample at a time, the following animation can help:

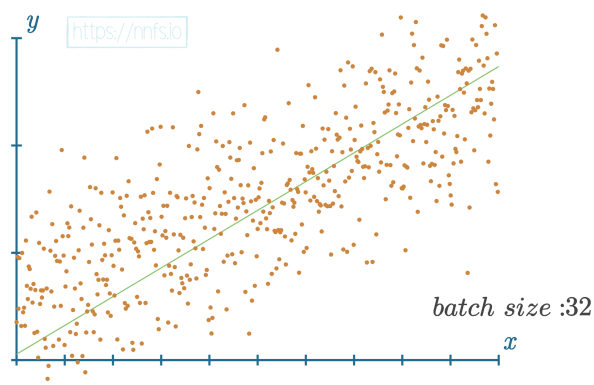


Fig 2.12: Example of a linear equation fitting batches of 32 chosen samples. See animation below for other sizes of samples at a time to see how much of a difference batch size can make.

Anim 2.12: https://nnfs.io/vyu

An example of a batch of data could look like:

Input data :

```
batch = [[1,5,6,2],
         [3,2,1,3],
         [5,2,1,2],
         [6,4,8,4],
         [2,8,5,3],
         [1,1,9,4],
         [6,6,0,4],
         [8,7,6,4]]
```

Shape :

```
(8, 4)
```

Type :

2D Array, Matrix

https://nnfs.io

Fig 2.13: Example of a batch, its shape, and type.

Anim 2.13: https://nnfs.io/lqw

Recall that in Python, and in our case, lists are useful containers for holding a sample as well as multiple samples that make up a batch of observations. Such an example of a batch of observations, each with its own sample, looks like:

```
inputs = [[1, 2, 3, 2.5], [2, 5, -1, 2], [-1.5, 2.7, 3.3, -0.8]]
```

This list of lists could be made into an array since it is homologous. Note that each "list" in this larger list is a sample representing a feature set. `[1, 2, 3, 2.5]`, `[2, 5, -1, 2]`, and `[-1.5, 2.7, 3.3, -0.8]` are all **samples**, and are also referred to as **feature set instances** or **observations**.

We have a matrix of inputs and a matrix of weights now, and we need to perform the dot product on them somehow, but how and what will the result be? Similarly, as we performed a dot product on a matrix and a vector, we treated the matrix as a list of vectors, resulting in a list of dot products. In this example, we need to manage both matrices as lists of vectors and perform dot products on all of them in all combinations, resulting in a list of lists of outputs, or a matrix; this operation is called the **matrix product**.

Matrix Product

The **matrix product** is an operation in which we have 2 matrices, and we are performing dot products of all combinations of rows from the first matrix and the columns of the 2nd matrix, resulting in a matrix of those atomic **dot products**:

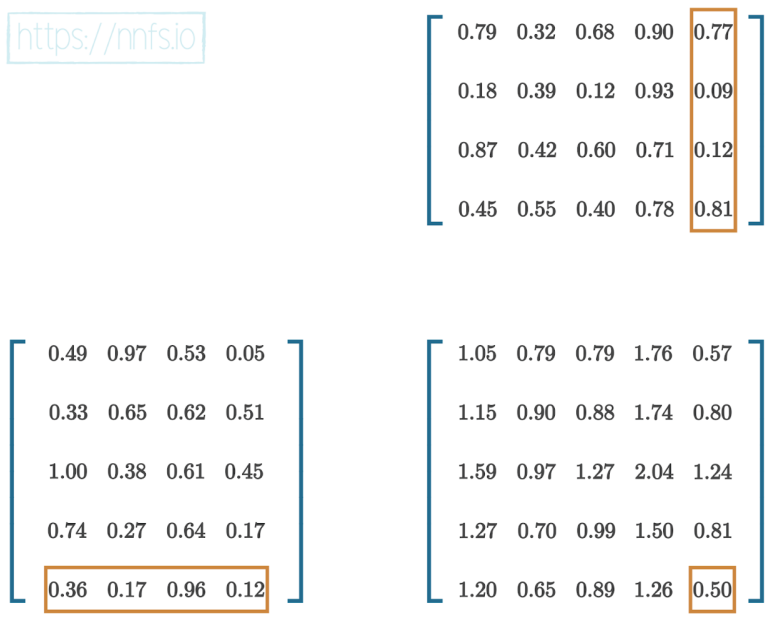


Fig 2.14: Visualizing how a single element in the resulting matrix from matrix product is calculated. See animation for the full calculation of each element.

Anim 2.14: https://nnfs.io/jei

To perform a matrix product, the size of the second dimension of the left matrix must match the size of the first dimension of the right matrix. For example, if the left matrix has a shape of *(5, 4)* then the right matrix must match this 4 within the first shape value *(4, 7)*. The shape of the resulting array is always the first dimension of the left array and the second dimension of the right array, *(5, 7)*. In the above example, the left matrix has a shape of *(5, 4)*, and the upper-right matrix has a shape of *(4, 5)*. The second dimension of the left array and the first dimension of the second array are both *4*, they match, and the resulting array has a shape of *(5, 5)*.

To elaborate, we can also show that we can perform the matrix product on vectors. In mathematics, we can have something called a column vector and row vector, which we'll explain better shortly. They're vectors, but represented as matrices with one of the dimensions having a size of 1:

$$a = \begin{bmatrix} 1 & 2 & 3 \end{bmatrix}$$

$$b = \begin{bmatrix} 2 \\ 3 \\ 4 \end{bmatrix}$$

a is a row vector. It looks very similar to a vector *a* (with an arrow above it) described earlier along with the vector product. The difference in notation between a row vector and vector are commas between values and the arrow above symbol *a* is missing on a row vector. It's called a row vector as it's a vector of a row of a matrix. *b*, on the other hand, is called a column vector because it's a column of a matrix. As row and column vectors are technically matrices, we do not denote them with vector arrows anymore.

When we perform the matrix product on them, the result becomes a matrix as well, like in the previous example, but containing just a single value, the same value as in the dot product example we have discussed previously:

$$ab = \begin{bmatrix} 1 & 2 & 3 \end{bmatrix} \begin{bmatrix} 2 \\ 3 \\ 4 \end{bmatrix} = \begin{bmatrix} 20 \end{bmatrix}$$

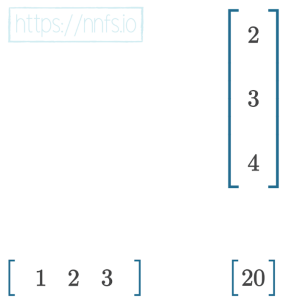


Fig 2.15: Product of row and column vectors.

Anim 2.15: https://nnfs.io/bkw

In other words, row and column vectors are matrices with one of their dimensions being of a size of 1; and, we perform the **matrix product** on them instead of the **dot product**, which results in a matrix containing a single value. In this case, we performed a matrix multiplication of matrices with shapes *(1, 3)* and *(3, 1)*, then the resulting array has the shape *(1, 1)* or a size of *1x1*.

Transposition for the Matrix Product

How did we suddenly go from 2 vectors to row and column vectors? We used the relation of the dot product and matrix product saying that a dot product of two vectors equals a matrix product of a row and column vector (the arrows above the letters signify that they are vectors):

$$\vec{a} \cdot \vec{b} = ab^T$$

We also have temporarily used some simplification, not showing that column vector *b* is actually a **transposed** vector *b*. The proper equation, matching the dot product of vectors *a* and *b* written as matrix product should look like:

$$ab^T = \begin{bmatrix} 1 & 2 & 3 \end{bmatrix} \begin{bmatrix} 2 \\ 3 \\ 4 \end{bmatrix} = \begin{bmatrix} 20 \end{bmatrix}$$

Here we introduced one more new operation — **transposition**. Transposition simply modifies a matrix in a way that its rows become columns and columns become rows:

$$\begin{bmatrix} 00 & 01 & 02 & 03 & 04 \\ 05 & 06 & 07 & 08 & 09 \\ 10 & 11 & 12 & 13 & 14 \\ 15 & 16 & 17 & 18 & 19 \end{bmatrix}^{\mathrm{T}} = \begin{bmatrix} 00 & 05 & 10 & 15 \\ 01 & 06 & 11 & 16 \\ 02 & 07 & 12 & 17 \\ 03 & 08 & 13 & 18 \\ 04 & 09 & 14 & 19 \end{bmatrix}$$

https://nnfs.io

Fig 2.16: Example of an array transposition.

Anim 2.16: https://nnfs.io/qut

$$\begin{bmatrix} 0.49 & 0.97 & 0.53 & 0.05 & 0.33 \\ 0.65 & 0.62 & 0.51 & 1.00 & 0.38 \\ 0.61 & 0.45 & 0.74 & 0.27 & 0.64 \\ 0.17 & 0.36 & 0.17 & 0.96 & 0.12 \\ 0.79 & 0.32 & 0.68 & 0.90 & 0.77 \end{bmatrix}^{\mathrm{T}} = \begin{bmatrix} 0.49 & 0.65 & 0.61 & 0.17 & 0.79 \\ 0.97 & 0.62 & 0.45 & 0.36 & 0.32 \\ 0.53 & 0.51 & 0.74 & 0.17 & 0.68 \\ 0.05 & 1.00 & 0.27 & 0.96 & 0.90 \\ 0.33 & 0.38 & 0.64 & 0.12 & 0.77 \end{bmatrix}$$

https://nnfs.io

Fig 2.17: Another example of an array transposition.

Anim 2.17: https://nnfs.io/pnq

Now we need to get back to row and column vector definitions and update them with what we have just learned.

A row vector is a matrix whose first dimension's size (the number of rows) equals 1 and the second dimension's size (the number of columns) equals *n* — the vector size. In other words, it's a 1×n array or array of shape (1, n):

$$a = \begin{bmatrix} a_1 & a_2 & a_3 & \dots & a_n \end{bmatrix}$$

With NumPy and with 3 values, we would define it as:

```
np.array([[1, 2, 3]])
```

Note the use of double brackets here. To transform a list into a matrix containing a single row (perform an equivalent operation of turning a vector into row vector), we can put it into a list and create numpy array:

```
a = [1, 2, 3]
print(np.array([a]))

>>>
array([[1, 2, 3]])
```

Again, note that we encase `a` in brackets before converting to an array in this case.
Or we can turn it into a 1D array and expand dimensions using one of the NumPy abilities:

```
a = [1, 2, 3]
print(np.expand_dims(np.array(a), axis=0))

>>>
array([[1, 2, 3]])
```

Where *np.expand_dims()* adds a new dimension at the index of the *axis*.
A column vector is a matrix where the second dimension's size equals 1, in other words, it's an array of shape (n, 1):

$$b = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ \dots \\ b_n \end{bmatrix}$$

With NumPy it can be created the same way as a row vector, but needs to be additionally transposed — transposition turns rows into columns and columns into rows:

$$\begin{bmatrix} b_1 & b_2 & b_3 & \dots & b_n \end{bmatrix}^T = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ \dots \\ b_n \end{bmatrix}$$

$$\begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ \dots \\ b_n \end{bmatrix}^T = \begin{bmatrix} b_1 & b_2 & b_3 & \dots & b_n \end{bmatrix}$$

To turn vector *b* into row vector *b*, we'll use the same method that we used to turn vector *a* into row vector *a*, then we can perform a transposition on it to make it a column vector *b*:

$$b = \begin{bmatrix} 2 & 3 & 4 \end{bmatrix}$$

$$b^T = \begin{bmatrix} 2 & 3 & 4 \end{bmatrix}^T = \begin{bmatrix} 2 \\ 3 \\ 4 \end{bmatrix}$$

With NumPy code:

```
import numpy as np

a = [1, 2, 3]
b = [2, 3, 4]

a = np.array([a])
b = np.array([b]).T


print(np.dot(a, b))

>>>
array([[20]])
```

We have achieved the same result as the dot product of two vectors, but performed on matrices and returning a matrix — exactly what we expected and wanted. It's worth mentioning that NumPy does not have a dedicated method for performing matrix product — the dot product and matrix product are both implemented in a single method: *np.dot()*.

As we can see, to perform a matrix product on two vectors, we took one as is, transforming it into a row vector, and the second one using transposition on it to turn it into a column vector. That allowed us to perform a matrix product that returned a matrix containing a single value. We also performed the matrix product on two example arrays to learn how a matrix product works — it creates a matrix of dot products of all combinations of row and column vectors.

A Layer of Neurons & Batch of Data w/ NumPy

Let's get back to our inputs and weights — when covering them, we mentioned that we need to perform dot products on all of the vectors that consist of both input and weight matrices. As we have just learned, that's the operation that the matrix product performs. We just need to perform transposition on its second argument, which is the weights matrix in our case, to turn the row vectors it currently consists of into column vectors.

Initially, we were able to perform the dot product on the inputs and the weights without a transposition because the weights were a matrix, but the inputs were just a vector. In this case, the dot product results in a vector of atomic dot products performed on each row from the matrix and this single vector. When inputs become a batch of inputs (a matrix), we need to perform the matrix product. It takes all of the combinations of rows from the left matrix and columns from the right matrix, performing the dot product on them and placing the results in an output array. Both arrays have the same shape, but, to perform the matrix product, the shape's value from the index 1 of the first matrix and the index 0 of the second matrix must match — they don't right now.

```
inputs = [[1.0, 2.0, 3.0, 2.5],
          [2.0, 5.0, -1.0, 2.0],
          [-1.5, 2.7, 3.3, -0.8]]
weights = [[0.2, 0.8, -0.5, 1.0],
           [0.5, -0.91, 0.26, -0.5],
           [-0.26, -0.27, 0.17, 0.87]]
```

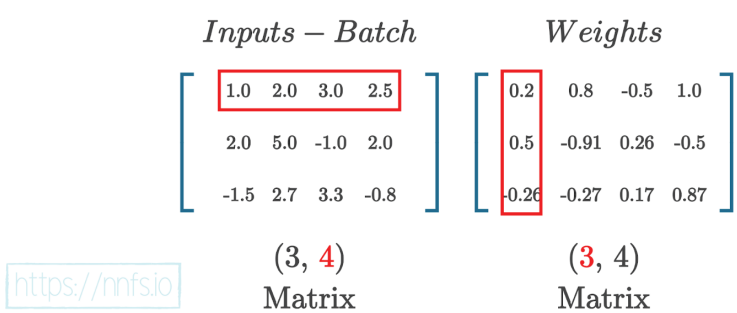


Fig 2.18: Depiction of why we need to transpose to perform the matrix product.

If we transpose the second array, values of its shape swap their positions.

```
inputs = [[1.0, 2.0, 3.0, 2.5],
          [2.0, 5.0, -1.0, 2.0],
          [-1.5, 2.7, 3.3, -0.8]]
weights = [[0.2, 0.8, -0.5, 1.0],
           [0.5, -0.91, 0.26, -0.5],
           [-0.26, -0.27, 0.17, 0.87]]
```

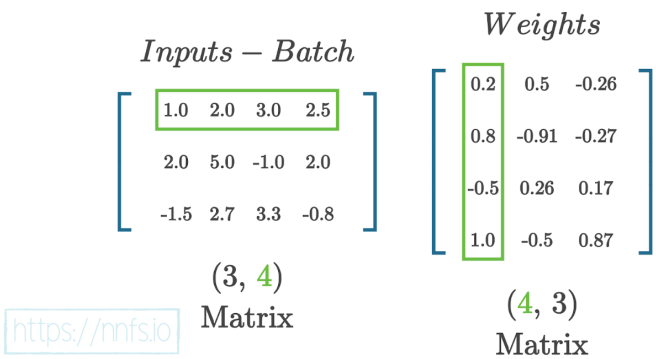


Fig 2.19: After transposition, we can perform the matrix product.

Anim 2.18-2.19: https://nnfs.io/crq

If we look at this from the perspective of the input and weights, we need to perform the dot product of each input and each weight set in all of their combinations. The dot product takes the row from the first array and the column from the second one, but currently the data in both arrays are row-aligned. Transposing the second array shapes the data to be column-aligned. The matrix product of inputs and transposed weights will result in a matrix containing all atomic dot products that we need to calculate. The resulting matrix consists of outputs of all neurons after operations performed on each input sample:

```
inputs = [[1.0, 2.0, 3.0, 2.5],
          [2.0, 5.0, -1.0, 2.0],
          [-1.5, 2.7, 3.3, -0.8]]
weights = [[0.2, 0.8, -0.5, 1.0],
           [0.5, -0.91, 0.26, -0.5],
           [-0.26, -0.27, 0.17, 0.87]]
biases = [2.0, 3.0, 0.5]

outputs = np.dot(inputs, np.array(weights).T) + biases


np.dot(inputs, np.array(weights).T)
```

https://nnfs.io

$$\begin{bmatrix} 1.0 & 2.0 & 3.0 & 2.5 \\ 2.0 & 5.0 & -1.0 & 2.0 \\ -1.5 & 2.7 & 3.3 & -0.8 \end{bmatrix} \begin{bmatrix} 0.2 & 0.5 & -0.26 \\ 0.8 & -0.91 & -0.27 \\ -0.5 & 0.26 & 0.17 \\ 1.0 & -0.5 & 0.87 \end{bmatrix} = \begin{bmatrix} 2.8 & -1.79 & 1.885 \\ 6.9 & -4.81 & -0.3 \\ -0.59 & -1.949 & -0.474 \end{bmatrix}$$

Fig 2.20: Code and visuals depicting the dot product of inputs and transposed weights.

Anim 2.20: https://nnfs.io/gjw

We mentioned that the second argument for *np.dot()* is going to be our transposed weights, so first will be inputs, but previously weights were the first parameter. We changed that here. Before, we were modeling neuron output using a single sample of data, a vector, but now we are a step forward when we model layer behavior on a batch of data. We could retain the current parameter order, but, as we'll soon learn, it's more useful to have a result consisting of a list of layer outputs per each sample than a list of neurons and their outputs sample-wise. We want the resulting array to be sample-related and not neuron-related as we'll pass those samples further through the network, and the next layer will expect a batch of inputs.

We can code this solution using NumPy now. We can perform *np.dot()* on a plain Python list of lists as NumPy will convert them to matrices internally. We are converting weights ourselves though to perform transposition operation first, *.T* in the code, as plain Python list of lists does not support it. Speaking of biases, we do not need to make it a NumPy array for the same reason — NumPy is going to do that internally.

Biases are a list, though, so they are a 1D array as a NumPy array. The addition of this bias vector to a matrix (of the dot products in this case) works similarly to the dot product of a matrix and vector that we described earlier; The bias vector will be added to each row vector of the matrix. Since each column of the matrix product result is an output of one neuron, and the vector is going to be added to each row vector, the first bias is going to be added to each first element of those vectors, second to second, etc. That's what we need — the bias of each neuron needs to be added to all of the results of this neuron performed on all input vectors (samples).

```
inputs = [[1.0, 2.0, 3.0, 2.5],
          [2.0, 5.0, -1.0, 2.0],
          [-1.5, 2.7, 3.3, -0.8]]
weights = [[0.2, 0.8, -0.5, 1.0],
           [0.5, -0.91, 0.26, -0.5],
           [-0.26, -0.27, 0.17, 0.87]]
biases = [2.0, 3.0, 0.5]

outputs = np.dot(inputs, np.array(weights).T) + biases
>>> array([[ 4.8    1.21   2.385],
           [ 8.9   -1.81   0.2  ],
           [ 1.41   1.051  0.026]])
```

https://nnfs.io

$$\begin{bmatrix} 2.8 & -1.79 & 1.885 \\ 6.9 & -4.81 & -0.3 \\ -0.59 & -1.949 & -0.474 \end{bmatrix} + \begin{bmatrix} 2.0 & 3.0 & 0.5 \end{bmatrix} = \begin{bmatrix} 4.8 & 1.21 & 2.385 \\ 8.9 & -1.81 & 0.2 \\ 1.41 & 1.051 & 0.026 \end{bmatrix}$$

Fig 2.21: Code and visuals for inputs mulitplied by the weights, plus the bias.

Anim 2.21: https://nnfs.io/qty

Now we can implement what we have learned into code:

```
import numpy as np


inputs = [[1.0, 2.0, 3.0, 2.5],
          [2.0, 5.0, -1.0, 2.0],
          [-1.5, 2.7, 3.3, -0.8]]
weights = [[0.2, 0.8, -0.5, 1.0],
           [0.5, -0.91, 0.26, -0.5],
           [-0.26, -0.27, 0.17, 0.87]]
biases = [2.0, 3.0, 0.5]

layer_outputs = np.dot(inputs, np.array(weights).T) + biases

print(layer_outputs)

>>>
array([[ 4.8    1.21   2.385],
       [ 8.9   -1.81   0.2  ],
       [ 1.41   1.051  0.026]])
```

As you can see, our neural network takes in a group of samples (inputs) and outputs a group of predictions. If you've used any of the deep learning libraries, this is why you pass in a list of inputs (even if it's just one feature set) and are returned a list of predictions, even if there's only one prediction.

Supplementary Material: https://nnfs.io/ch2
Chapter code, further resources, and errata for this chapter.

Chapter 3

Adding Layers

The neural network we've built is becoming more respectable, but at the moment, we have only one layer. Neural networks become "deep" when they have 2 or more **hidden layers**. At the moment, we have just one layer, which is effectively an output layer. Why we want two or more **hidden** layers will become apparent in a later chapter. Currently, we have no hidden layers. A hidden layer isn't an input or output layer; as the scientist, you see data as they are handed to the input layer and the resulting data from the output layer. Layers between these endpoints have values that we don't necessarily deal with, hence the name "hidden." Don't let this name convince you that you can't access these values, though. You will often use them to diagnose issues or improve your neural network. To explore this concept, let's add another layer to this neural network, and, for now, let's assume these two layers that we're going to have will be the hidden layers, and we just have not coded our output layer yet.

Before we add another layer, let's think about what will be coming. In the case of the first layer, we can see that we have an input with 4 features.

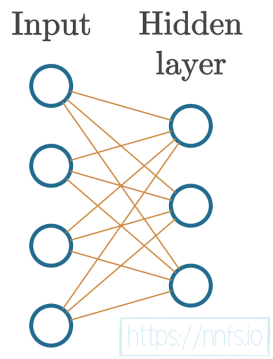


Fig 3.01: Input layer with 4 features into a hidden layer with 3 neurons.

Samples (feature set data) get fed through the input, which does not change it in any way, to our first hidden layer, which we can see has 3 sets of weights, with 4 values each.

Each of those 3 unique weight sets is associated with its distinct neuron. Thus, since we have 3 weight sets, we have 3 neurons in this first hidden layer. Each neuron has a unique set of weights, of which we have 4 (as there are 4 inputs to this layer), which is why our initial weights have a shape of *(3,4)*.

Now, we wish to add another layer. To do that, we must make sure that the expected input to that layer matches the previous layer's output. We have set the number of neurons in a layer by setting how many weight sets and biases we have. The previous layer's influence on weight sets for the current layer is that each weight set needs to have a separate weight per input. This means a distinct weight per neuron from the previous layer (or feature if we're talking the input). The previous layer has 3 weight sets and 3 biases, so we know it has 3 neurons. This then means, for the next layer, we can have as many weight sets as we want (because this is how many neurons this new layer will have), but each of those weight sets must have 3 discrete weights.

To create this new layer, we are going to copy and paste our `weights` and `biases` to `weights2` and `biases2`, and change their values to new made up sets. Here's an example:

```
inputs = [[1, 2, 3, 2.5],
          [2., 5., -1., 2],
          [-1.5, 2.7, 3.3, -0.8]]
weights = [[0.2, 0.8, -0.5, 1],
           [0.5, -0.91, 0.26, -0.5],
           [-0.26, -0.27, 0.17, 0.87]]
biases = [2, 3, 0.5]
```

```
weights2 = [[0.1, -0.14, 0.5],
            [-0.5, 0.12, -0.33],
            [-0.44, 0.73, -0.13]]
biases2 = [-1, 2, -0.5]
```

Next, we will now call *outputs* "*layer1_ouputs*":

```
layer1_outputs = np.dot(inputs, np.array(weights).T) + biases
```

As previously stated, inputs to layers are either inputs from the actual dataset you're training with or outputs from a previous layer. That's why we defined 2 versions of *weights* and *biases* but only 1 of *inputs* — because the inputs for layer 2 will be the outputs from the previous layer:

```
layer2_outputs = np.dot(layer1_outputs, np.array(weights2).T) + biases2
```

All together now:

```
import numpy as np

inputs = [[1, 2, 3, 2.5],
          [2., 5., -1., 2],
          [-1.5, 2.7, 3.3, -0.8]]
weights = [[0.2, 0.8, -0.5, 1],
           [0.5, -0.91, 0.26, -0.5],
           [-0.26, -0.27, 0.17, 0.87]]
biases = [2, 3, 0.5]
weights2 = [[0.1, -0.14, 0.5],
            [-0.5, 0.12, -0.33],
            [-0.44, 0.73, -0.13]]
biases2 = [-1, 2, -0.5]

layer1_outputs = np.dot(inputs, np.array(weights).T) + biases
layer2_outputs = np.dot(layer1_outputs, np.array(weights2).T) + biases2

print(layer2_outputs)
```

```
>>>
array([[ 0.5031  -1.04185 -2.03875],
       [ 0.2434  -2.7332  -5.7633 ],
       [-0.99314  1.41254 -0.35655]])
```

At this point, our neural network could be visually represented as:

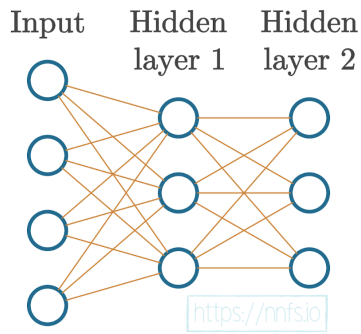


Fig 3.02: 4 features input into 2 hidden layers of 3 neurons each.

Training Data

Next, rather than hand-typing in random data, we'll use a function that can create non-linear data. What do we mean by non-linear? Linear data can be fit with or represented by a straight line.

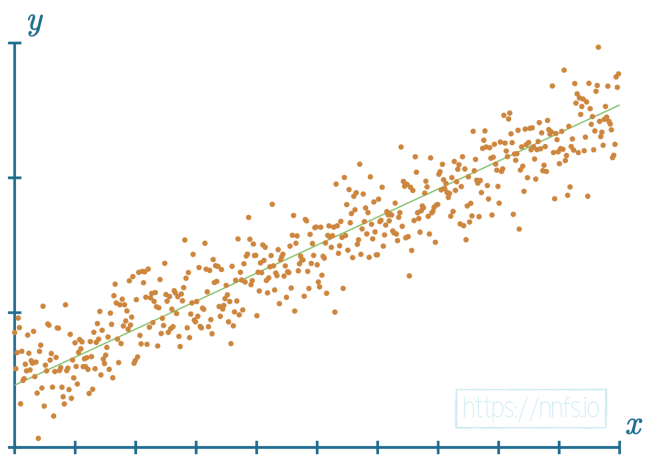


Fig 3.03: Example of data (orange dots) that can be represented (fit) by a straight line (green line).

Non-linear data cannot be represented well by a straight line.

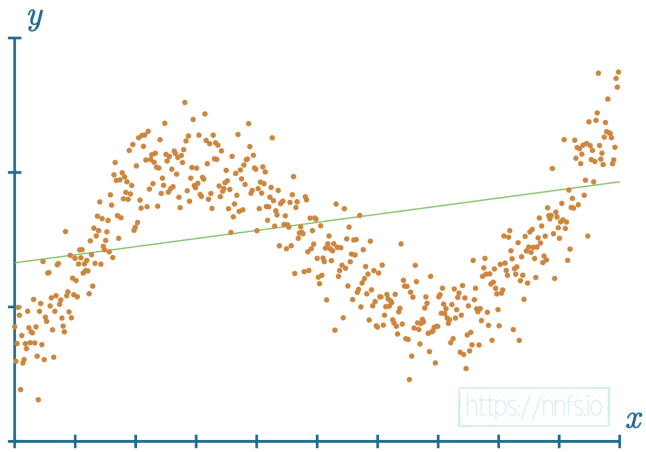


Fig 3.04: Example of data (orange dots) that is not well fit by a straight line.

If you were to graph data points of the form *(x, y)* where *y = f(x)*, and it looks to be a line with a clear trend or slope, then chances are, they're linear data! Linear data are very easily approximated by far simpler machine learning models than neural networks. What other machine learning algorithms cannot approximate so easily are non-linear datasets. To simplify this, we've created a Python package that you can install with pip, called *nnfs*:

```
pip install nnfs
```

The nnfs package contains functions that we can use to create data. For example:

```
from nnfs.datasets import spiral_data
```

The *spiral_data* function was slightly modified from https://cs231n.github.io/neural-networks-case-study/, which is a great supplementary resource for this topic.

You will typically not be generating training data from a function for your neural networks. You will have an actual dataset. Generating a dataset this way is purely for convenience at this stage. We will also use this package to ensure repeatability for everyone, using `nnfs.init()`, after importing NumPy:

```
import numpy as np
import nnfs

nnfs.init()
```

The `nnfs.init()` does three things: it sets the random seed to 0 (by the default), creates a *float32* dtype default, and overrides the original dot product from NumPy. All of these are meant to ensure repeatable results for following along.

The `spiral_data` function allows us to create a dataset with as many classes as we want. The function has parameters to choose the number of classes and the number of points/observations per class in the resulting non-linear dataset. For example:

```
import matplotlib.pyplot as plt

X, y = spiral_data(samples=100, classes=3)

plt.scatter(X[:, 0], X[:, 1])
plt.show()
```

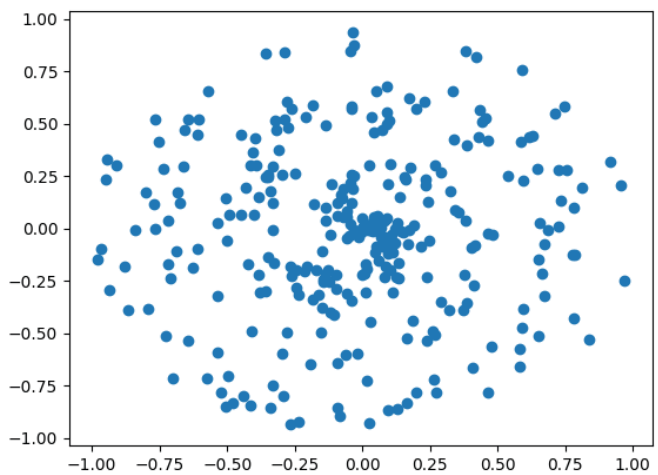


Fig 3.05: Uncolored spiral dataset.

If you trace from the center, you can determine all 3 classes separately, but this is a very challenging problem for a machine learning classifier to solve. Adding color to the chart makes this more clear:

```
plt.scatter(X[:, 0], X[:, 1], c=y, cmap='brg')
plt.show()
```

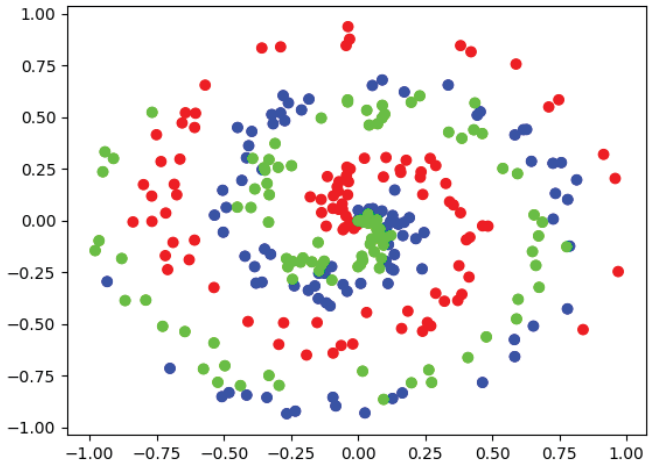


Fig 3.06: Spiral dataset colored by class.

Keep in mind that the neural network will not be aware of the color differences as the data have no class encodings. This is only made as an instruction for the reader. In the data above, each dot is the feature, and its coordinates are the samples that form the dataset. The "classification" for that dot has to do with which spiral it is a part of, depicted by blue, green, or red color in the previous image. These colors would then be assigned a class number for the model to fit to, like 0, 1, and 2.

Dense Layer Class

Now that we no longer need to hand-type our data, we should create something similar for our various types of neural network layers. So far, we've only used what's called a **dense** or **fully-connected** layer. These layers are commonly referred to as "dense" layers in papers, literature, and code, but you will occasionally see them called fully-connected or "fc" for short in code. Our dense layer class will begin with two methods.

```
# Dense layer
class Layer_Dense:

    # Layer initialization
    def __init__(self, n_inputs, n_neurons):
        # Initialize weights and biases
        pass  # using pass statement as a placeholder

    # Forward pass
    def forward(self, inputs):
        # Calculate output values from inputs, weights and biases
        pass  # using pass statement as a placeholder
```

As previously stated, weights are often initialized randomly for a model, but not always. If you wish to load a pre-trained model, you will initialize the parameters to whatever that pre-trained model finished with. It's also possible that, even for a new model, you have some other initialization rules besides random. For now, we'll stick with random initialization. Next, we have the *forward* method. When we pass data through a model from beginning to end, this is called a **forward pass**. Just like everything else, however, this is not the only way to do things. You can have the data loop back around and do other interesting things. We'll keep it common and perform a regular forward pass.

To continue the `Layer_Dense` class' code let's add the random initialization of weights and biases:

```
    # Layer initialization
    def __init__(self, n_inputs, n_neurons):
        self.weights = 0.01 * np.random.randn(n_inputs, n_neurons)
        self.biases = np.zeros((1, n_neurons))
```

Here, we're setting weights to be random and biases to be 0. Note that we're initializing weights to be *(inputs, neurons),* rather than (*neurons, inputs)*. We're doing this ahead instead of transposing every time we perform a forward pass, as explained in the previous chapter. Why zero biases? In specific scenarios, like with many samples containing values of 0, a bias can ensure that a neuron fires initially. It sometimes may be appropriate to initialize the biases to some non-zero number, but the most common initialization for biases is 0. However, in these scenarios, you may find success in doing things another way. This will vary depending on your use-case and is just one of many things you can tweak when trying to improve results. One situation where you might want to try something else is with what's called **dead neurons**. We haven't yet covered activation functions in practice, but imagine our step function again.

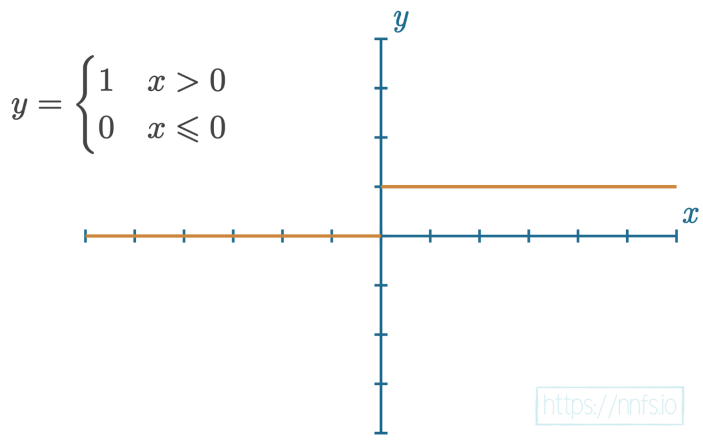


Fig 3.07: Graph of a step function.

It's possible for *weights · inputs + biases* not to meet the threshold of the step function, which means the neuron will output a 0. Alone, this is not a big issue, but it becomes a problem if this happens to this neuron for every one of the input samples (it'll become clear why once we cover backpropagation). So then this neuron's 0 output is the input to another neuron. Any weight multiplied by zero will be zero. With an increasing number of neurons outputting 0, more inputs to the next neurons will receive these 0s rendering the network essentially non-trainable, or "dead."

Next, let's explore `np.random.randn` and `np.zeros` in more detail. These methods are convenient ways to initialize arrays. `np.random.randn` produces a Gaussian distribution with a mean of 0 and a variance of 1, which means that it'll generate random numbers, positive and negative, centered at 0 and with the mean value close to 0. In general, neural networks work best with values between -1 and +1, which we'll discuss later in an upcoming chapter. So this `np.random.randn` generates values around those numbers. We're going to multiply this Gaussian distribution for the weights by *0.01* to generate numbers that are a couple of magnitudes smaller. Otherwise, the model will take more time to fit the data during the training process as starting values will be disproportionately large compared to the updates being made during

training. The idea here is to start a model with non-zero values small enough that they won't affect training. This way, we have a bunch of values to begin working with, but hopefully none too large or as zeros. You can experiment with values other than *0.01* if you like.

Finally, the `np.random.randn` function takes dimension sizes as parameters and creates the output array with this shape. The weights here will be the number of inputs for the first dimension and the number of neurons for the 2nd dimension. This is similar to our previous made up array of weights, just randomly generated. Whenever there's a function or block of code that you're not sure about, you can always print it out. For example:

```
import numpy as np
import nnfs

nnfs.init()

print(np.random.randn(2, 5))
```

```
>>>
[[ 1.7640524   0.4001572   0.978738    2.2408931   1.867558  ]
 [-0.9772779   0.95008844 -0.1513572  -0.10321885  0.41059852]]
```

The example function call has returned a 2x5 array (which we can also say is "*with a shape of (2,5)*") with data randomly sampled from a Gaussian distribution with a mean of 0.

Next, the `np.zeros` function takes a desired array shape as an argument and returns an array of that shape filled with zeros.

```
print(np.zeros((2, 5)))

>>>
[[0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0.]]
```

We'll initialize the biases with the shape of *(1, n_neurons)*, as a row vector, which will let us easily add it to the result of the dot product later, without additional operations like transposition.

To see an example of how our method initializes weights and biases:

```
import numpy as np
import nnfs

nnfs.init()

n_inputs = 2
n_neurons = 4

weights = 0.01 * np.random.randn(n_inputs, n_neurons)
biases = np.zeros((1, n_neurons))

print(weights)
print(biases)

>>>
[[ 0.01764052  0.00400157  0.00978738  0.02240893]
 [ 0.01867558 -0.00977278  0.00950088 -0.00151357]]
[[0. 0. 0. 0.]]
```

On to our forward method — we need to update it with the dot product+biases calculation:

```
def forward(self, inputs):
    self.output = np.dot(inputs, self.weights) + self.biases
```

Nothing new here, just turning the previous code into a method. Our full *Layer_Dense* class so far:

```
# Dense layer
class Layer_Dense:

    # Layer initialization
    def __init__(self, n_inputs, n_neurons):
        self.weights = 0.01 * np.random.randn(n_inputs, n_neurons)
        self.biases = np.zeros((1, n_neurons))

    # Forward pass
    def forward(self, inputs):
        self.output = np.dot(inputs, self.weights) + self.biases
```

We're ready to make use of this new class instead of hardcoded calculations, so let's generate some data using discussed dataset creation method and use our new layer to perform a forward pass:

```
# Create dataset
X, y = spiral_data(samples=100, classes=3)

# Create Dense layer with 2 input features and 3 output values
dense1 = Layer_Dense(2, 3)

# Perform a forward pass of our training data through this layer
dense1.forward(X)

# Let's see output of the first few samples:
print(dense1.output[:5])
```

Go ahead and run everything.

Full code up to this point:

```
import numpy as np
import nnfs
from nnfs.datasets import spiral_data

nnfs.init()


# Dense layer
class Layer_Dense:

    # Layer initialization
    def __init__(self, n_inputs, n_neurons):
        # Initialize weights and biases
        self.weights = 0.01 * np.random.randn(n_inputs, n_neurons)
        self.biases = np.zeros((1, n_neurons))
```

```
    # Forward pass
    def forward(self, inputs):
        # Calculate output values from inputs, weights and biases
        self.output = np.dot(inputs, self.weights) + self.biases


# Create dataset
X, y = spiral_data(samples=100, classes=3)

# Create Dense layer with 2 input features and 3 output values
dense1 = Layer_Dense(2, 3)

# Perform a forward pass of our training data through this layer
dense1.forward(X)

# Let's see output of the first few samples:
print(dense1.output[:5])
```

```
>>>
[[ 0.0000000e+00  0.0000000e+00  0.0000000e+00]
 [-1.0475188e-04  1.1395361e-04 -4.7983500e-05]
 [-2.7414842e-04  3.1729150e-04 -8.6921798e-05]
 [-4.2188365e-04  5.2666257e-04 -5.5912682e-05]
 [-5.7707680e-04  7.1401405e-04 -8.9430439e-05]]
```

In the output, you can see we have 5 rows of data that have 3 values each. Each of those 3 values is the value from the 3 neurons in the *dense1* layer after passing in each of the samples. Great! We have a network of neurons, so our neural network model is almost deserving of its name, but we're still missing the activation functions, so let's do those next!

Supplementary Material: https://nnfs.io/ch3
Chapter code, further resources, and errata for this chapter.

Chapter 4

Activation Functions

In this chapter, we will tackle a few of the activation functions and discuss their roles. We use different activation functions for different cases, and understanding how they work can help you properly pick which of them is best for your task. The activation function is applied to the output of a neuron (or layer of neurons), which modifies outputs. We use activation functions because if the activation function itself is nonlinear, it allows for neural networks with usually two or more hidden layers to map nonlinear functions. We'll be showing how this works in this chapter.

In general, your neural network will have two types of activation functions. The first will be the activation function used in hidden layers, and the second will be used in the output layer. Usually, the activation function used for hidden neurons will be the same for all of them, but it doesn't have to.

The Step Activation Function

Recall the purpose this activation function serves is to mimic a neuron "firing" or "not firing" based on input information. The simplest version of this is a step function. In a single neuron, if the *weights* · *inputs* + *bias* results in a value greater than 0, the neuron will fire and output a 1; otherwise, it will output a 0.

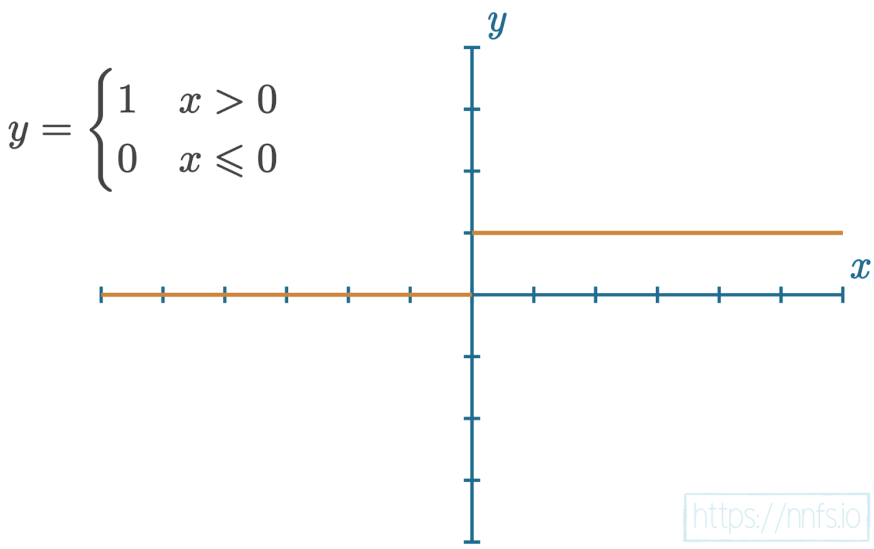


Fig 4.01: Step function graph.

This activation function has been used historically in hidden layers, but nowadays, it is rarely a choice.

The Linear Activation Function

A linear function is simply the equation of a line. It will appear as a straight line when graphed, where y=x and the output value equals the input.

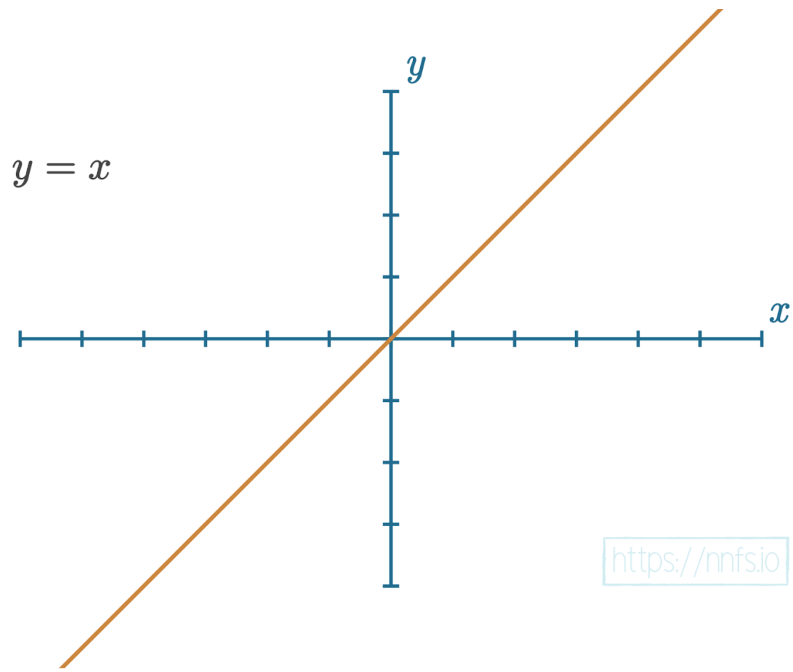


Fig 4.02: Linear function graph.

This activation function is usually applied to the last layer's output in the case of a regression model — a model that outputs a scalar value instead of a classification. We'll cover regression in chapter 17 and soon in an example in this chapter.

The Sigmoid Activation Function

The problem with the step function is it's not very informative. When we get to training and network optimizers, you will see that the way an optimizer works is by assessing individual impacts that weights and biases have on a network's output. The problem with a step function is that it's less clear to the optimizer what these impacts are because there's very little information gathered from this function. It's either on (1) or off (0). It's hard to tell how "close" this step function was to activating or deactivating. Maybe it was very close, or maybe it was very far. In terms of the final output value from the network, it doesn't matter if it was *close* to outputting something else. Thus, when it comes time to optimize weights and biases, it's easier for the optimizer if we have activation functions that are more granular and informative.

The original, more granular, activation function used for neural networks was the **Sigmoid** activation function, which looks like:

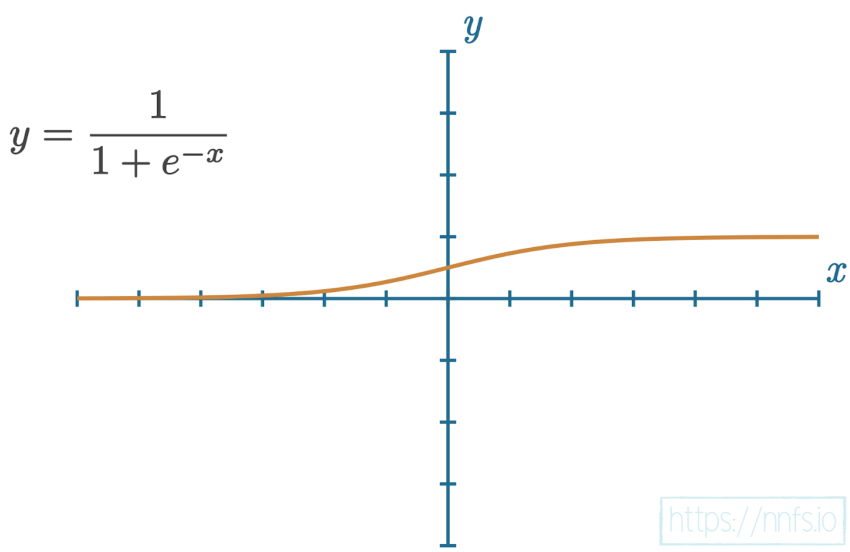


Fig 4.03: Sigmoid function graph.

This function returns a value in the range of 0 for negative infinity, through 0.5 for the input of 0, and to 1 for positive infinity. We'll talk about this function more in chapter 16.

As mentioned earlier, with "dead neurons," it's usually better to have a more granular approach for the hidden neuron activation functions. In this case, we're getting a value that can be reversed to its original value; the returned value contains all the information from the input, contrary to a function like the step function, where an input of 3 will output the same value as an input of 300,000. The output from the Sigmoid function, being in the range of 0 to 1, also works better with neural networks — especially compared to the range of the negative to the positive infinity — and adds nonlinearity. The importance of nonlinearity will become more clear shortly in this chapter. The Sigmoid function, historically used in hidden layers, was eventually replaced by the **Rectified Linear Units** activation function (or **ReLU**). That said, we will be using the Sigmoid function as the output layer's activation function in chapter 16.

The Rectified Linear Activation Function

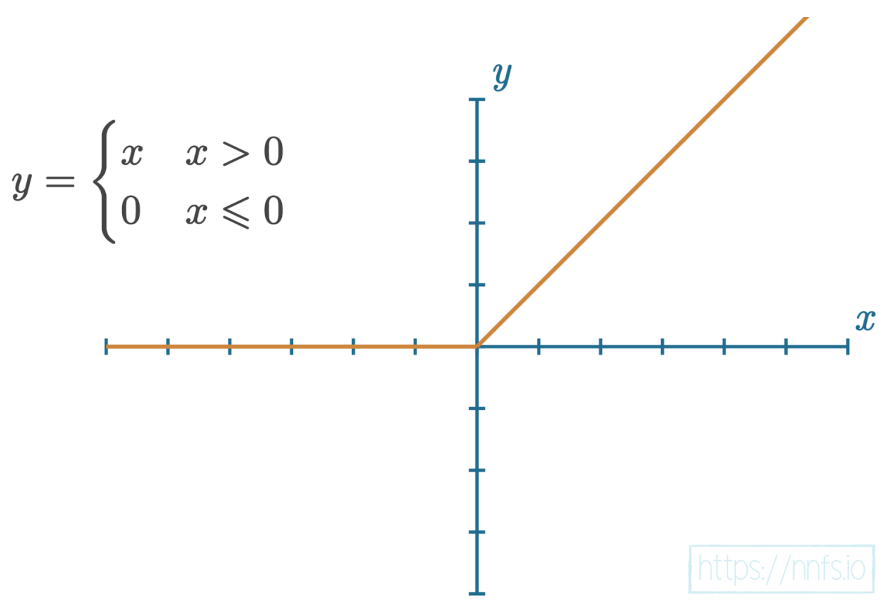


Fig 4.04: Graph of the ReLU activation function.

The rectified linear activation function is simpler than the sigmoid. It's quite literally $y=x$, clipped at 0 from the negative side. If x is less than or equal to 0, then y is 0 — otherwise, y is equal to x.

$$y = \begin{cases} x & x > 0 \\ 0 & x \leqslant 0 \end{cases}$$

This simple yet powerful activation function is the most widely used activation function at the time of writing for various reasons — mainly speed and efficiency. While the sigmoid activation function isn't the most complicated, it's still much more challenging to compute than the ReLU activation function. The ReLU activation function is extremely close to being a linear activation function while remaining nonlinear, due to that bend after 0. This simple property is, however, very effective.

Why Use Activation Functions?

Now that we understand what activation functions represent, how some of them look, and what they return, let's discuss *why* we use activation functions in the first place. In most cases, for a neural network to fit a nonlinear function, we need it to contain two or more hidden layers, and we need those hidden layers to use a nonlinear activation function.

First off, what's a nonlinear function? A nonlinear function cannot be represented well by a straight line, such as a sine function:

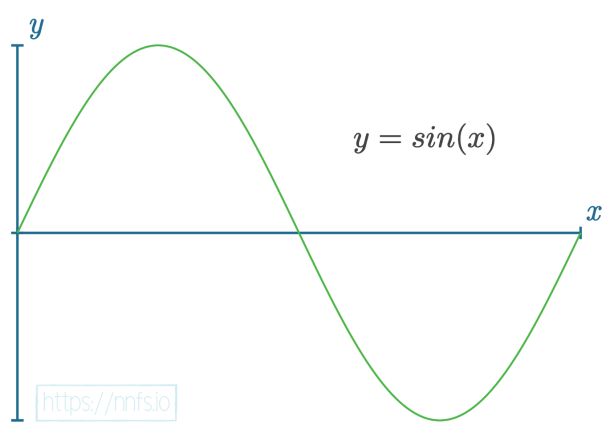


Fig 4.05: Graph of y=sin(x)

While there are certainly problems in life that are linear in nature, for example, trying to figure out the cost of some number of shirts, and we know the cost of an individual shirt, and that there are no bulk discounts, then the equation to calculate the price of any number of those products is a linear equation. Other problems in life are not so simple, like the price of a home. The number of factors that come into play, such as size, location, time of year attempting to sell, number of rooms, yard, neighborhood, and so on, makes the pricing of a home a nonlinear equation. Many of the more interesting and hard problems of our time are nonlinear. The main attraction for neural networks has to do with their ability to solve nonlinear problems. First, let's consider a situation where neurons have no activation function, which would be the same as having an activation function of $y=x$. With this linear activation function in a neural network with 2 hidden layers of 8 neurons each, the result of training this model will look like:

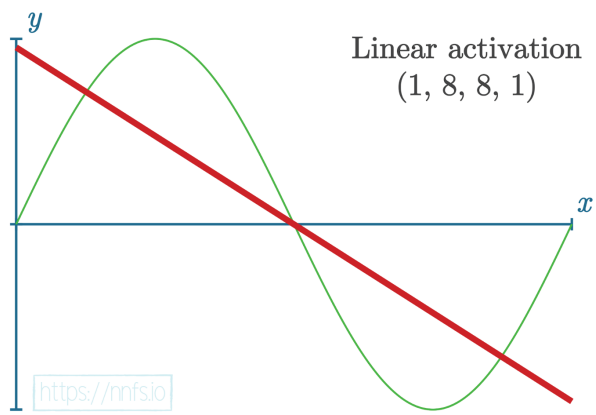


Fig 4.06: Neural network with linear activation functions in hidden layers attempting to fit y=sin(x)

When using the same 2 hidden layers of 8 neurons each with the rectified linear activation function, we see the following result after training:

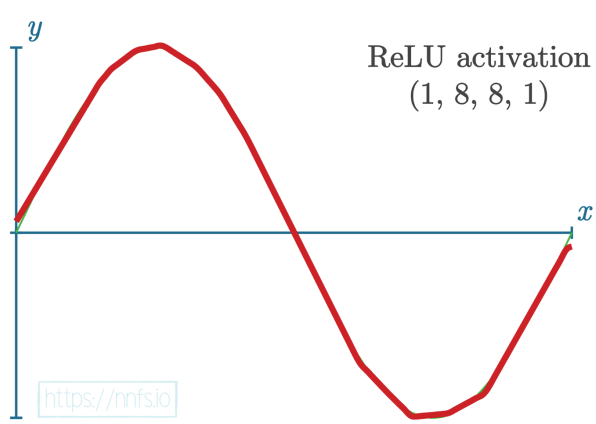


Fig 4.07: ReLU activation functions in hidden layers attempting to fit y=sin(x)

Linear Activation in the Hidden Layers

Now that you can see that this is the case, we still should consider *why* this is the case. To begin, let's revisit the linear activation function of $y=x$, and let's consider this on a singular neuron level. Given values for weights and biases, what will the output be for a neuron with a $y=x$ activation function? Let's look at some examples — first, let's try to update the first weight with a positive value:

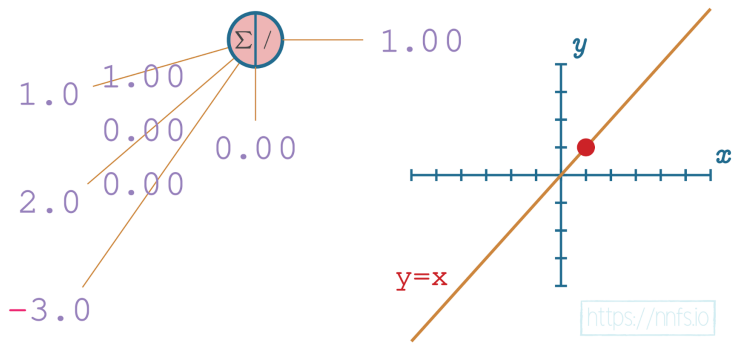


Fig 4.08: Example of output with a neuron using a linear activation function.

As we continue to tweak with weights, updating with a negative number this time:

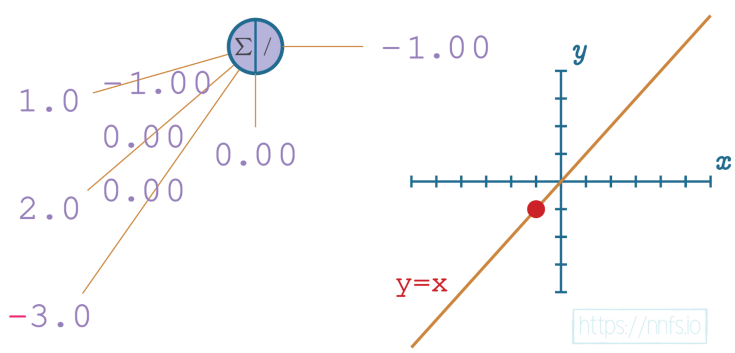


Fig 4.09: Example of output with a neuron using a linear activation function, updated weight.

And updating weights and additionally a bias:

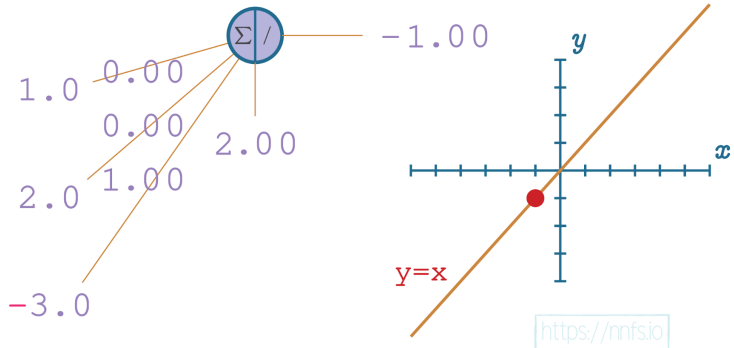


Fig 4.10: Example of output with a neuron using a linear activation function, updated another weight.

No matter what we do with this neuron's weights and biases, the output of this neuron will be perfectly linear to $y=x$ of the activation function. This linear nature will continue throughout the entire network:

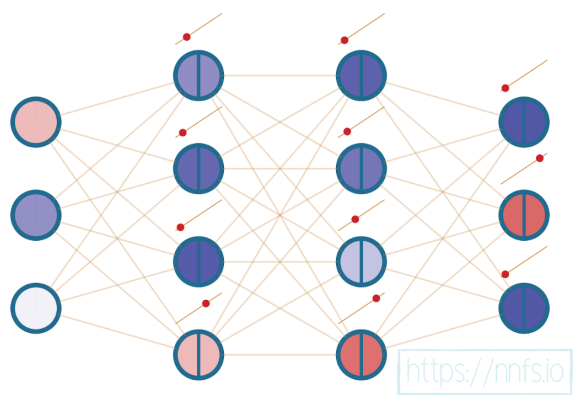


Fig 4.11: A neural network with all linear activation functions.

No matter what we do, however many layers we have, this network can only depict linear relationships if we use linear activation functions. It should be fairly obvious that this will be the case as each neuron in each layer acts linearly, so the entire network is a linear function as well.

ReLU Activation in a Pair of Neurons

We believe it is less obvious how, with a barely nonlinear activation function, like the rectified linear activation function, we can suddenly map nonlinear relationships and functions, so now let's cover that. Let's start again with a single neuron. We'll begin with both a weight of 0 and a bias of 0:

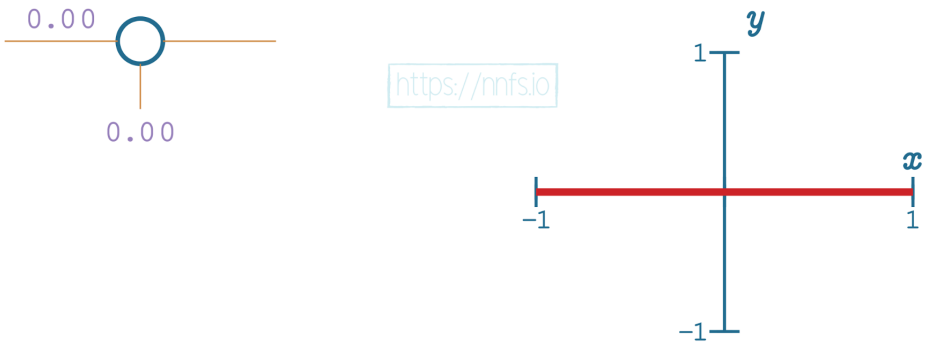


Fig 4.12: Single neuron with single input (zeroed weight) and ReLU activation function.

In this case, no matter what input we pass, the output of this neuron will always be a 0, because the weight is 0, and there's no bias. Let's set the weight to be 1:

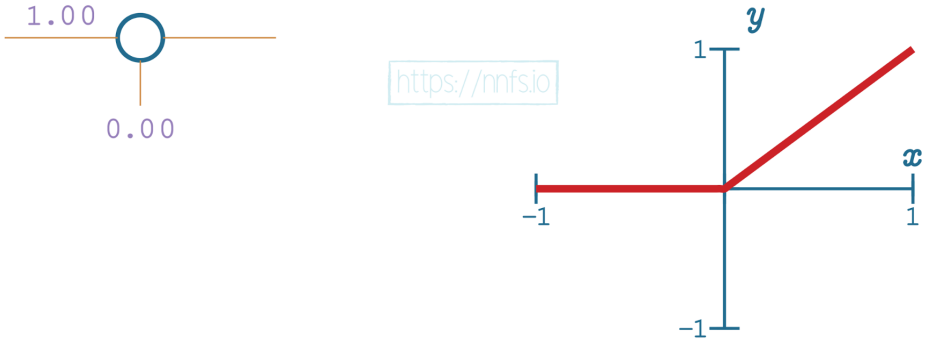


Fig 4.13: Single neuron with single input and ReLU activation function, weight set to 1.0.

Now it looks just like the basic rectified linear function, no surprises yet! Now let's set the bias to 0.50:

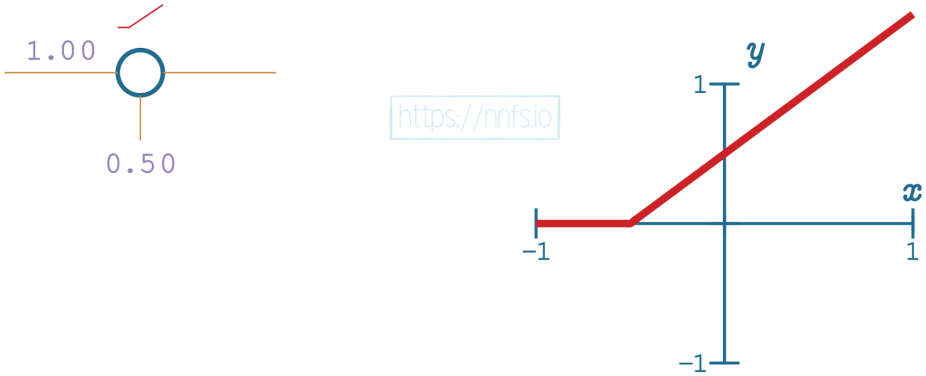


Fig 4.14: Single neuron with single input and ReLU activation function, bias applied.

We can see that, in this case, with a single neuron, the bias offsets the overall function's activation point *horizontally*. By increasing bias, we're making this neuron activate earlier. What happens when we negate the weight to -1.0?

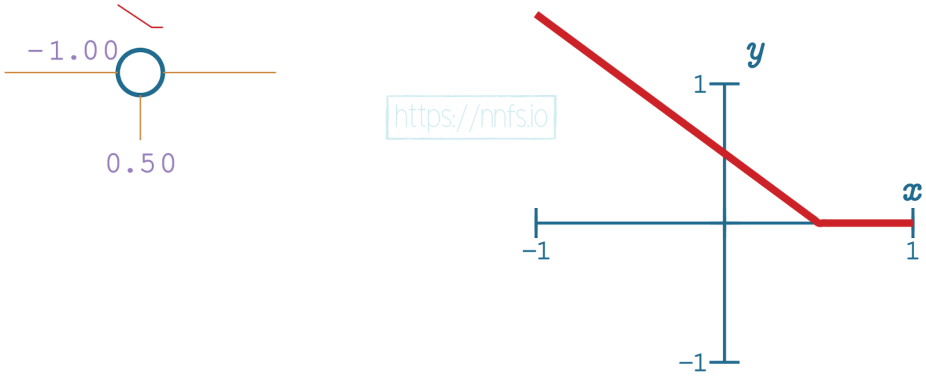


Fig 4.15: Single neuron with single input and ReLU activation function, negative weight.

With a negative weight and this single neuron, the function has become a question of when this neuron *deactivates*. Up to this point, you've seen how we can use the bias to offset the function horizontally, and the weight to influence the slope of the activation. Moreover, we're also able to control whether the function is one for determining where the neuron activates or deactivates.

What happens when we have, rather than just the one neuron, a pair of neurons? For example, let's pretend that we have 2 hidden layers of 1 neuron each. Thinking back to the $y=x$ activation function, we unsurprisingly discovered that a linear activation function produced linear results no matter what chain of neurons we made. Let's see what happens with the rectified linear function for the activation. We'll begin with the last values for the 1st neuron and a weight of 1, with a bias of 0, for the 2nd neuron:

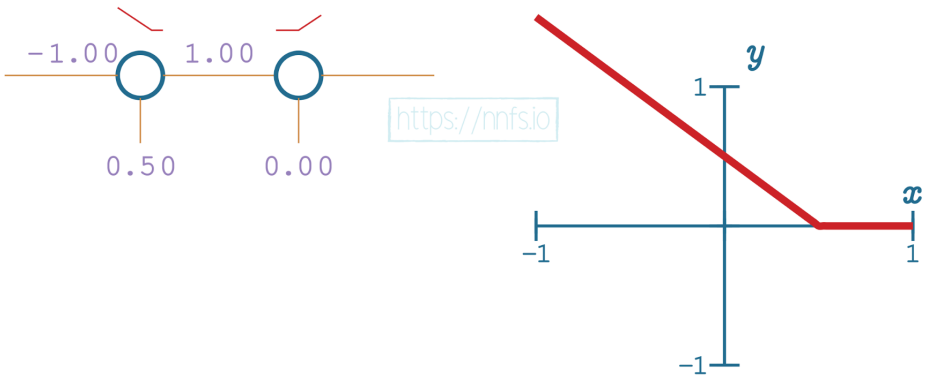


Fig 4.16: Pair of neurons with single inputs and ReLU activation functions.

As we can see so far, there's no change. This is because the 2nd neuron's bias is doing no offsetting, and the 2nd neuron's weight is just multiplying output by 1, so there's no change. Let's try to adjust the 2nd neuron's bias now:

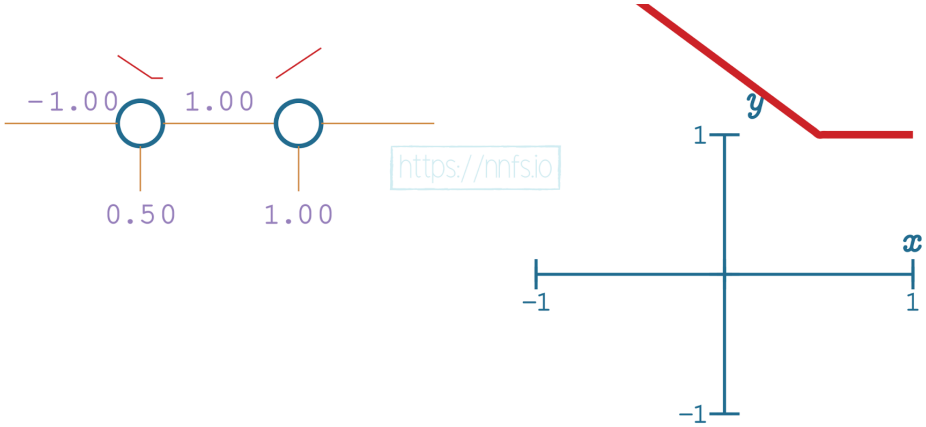


Fig 4.17: Pair of neurons with single inputs and ReLU activation functions, other bias applied.

Now we see some fairly interesting behavior. The bias of the second neuron indeed shifted the overall function, but, rather than shifting it *horizontally*, it shifted the function *vertically*. What then might happen if we make that 2nd neuron's weight -2 rather than 1?

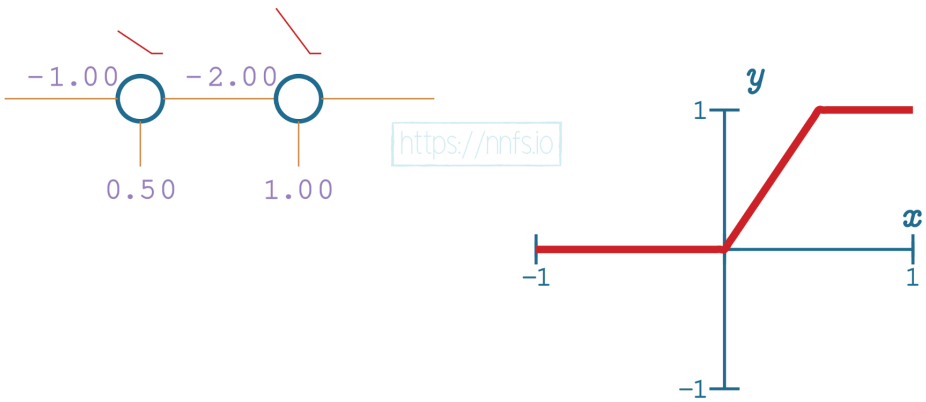


Fig 4.18: Pair of neurons with single inputs and ReLU activation functions, other negative weight.

Something exciting has occurred! What we have here is a neuron that has both an activation and a deactivation point. When *both* neurons are activated, when their "area of effect" comes into play, they produce values in the range of the granular, variable, and output. If any neuron in the pair is inactive, the pair will produce non-variable output:

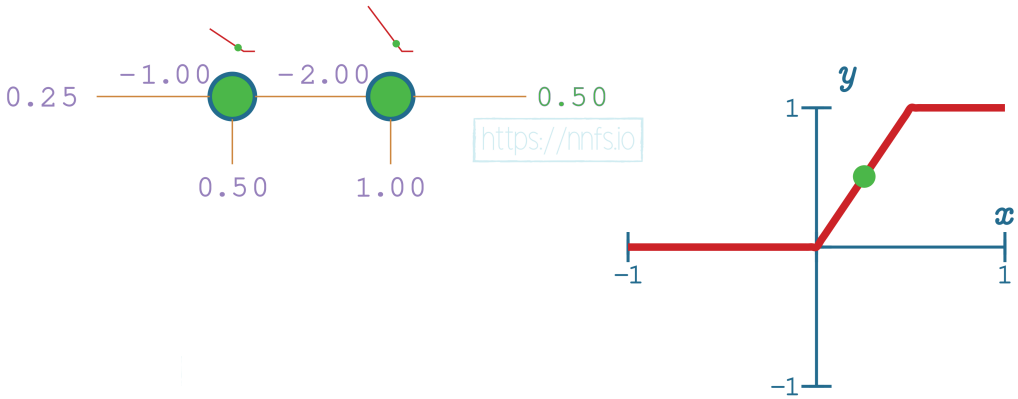


Fig 4.19: Pair of neurons with single inputs and ReLU activation functions, area of effect.

ReLU Activation in the Hidden Layers

Let's now take this concept and use it to fit to the sine wave function using 2 hidden layers of 8 neurons each, and we can hand-tune the values to fit the curve. We'll do this by working with 1 pair of neurons at a time, which means 1 neuron from each layer individually. For simplicity, we are also going to assume that the layers are not densely connected, and each neuron from the first hidden layer connects to only one neuron from the second hidden layer. That's usually not the case with the real models, but we want this simplification for the purpose of this demo. Additionally, this example model takes a single value as an input, the input to the sine function, and outputs a single value like the sine function. The output layer uses the Linear activation function, and the hidden layers will use the rectified linear activation function.

To start, we'll set all weights to 0 and work with the first pair of neurons:

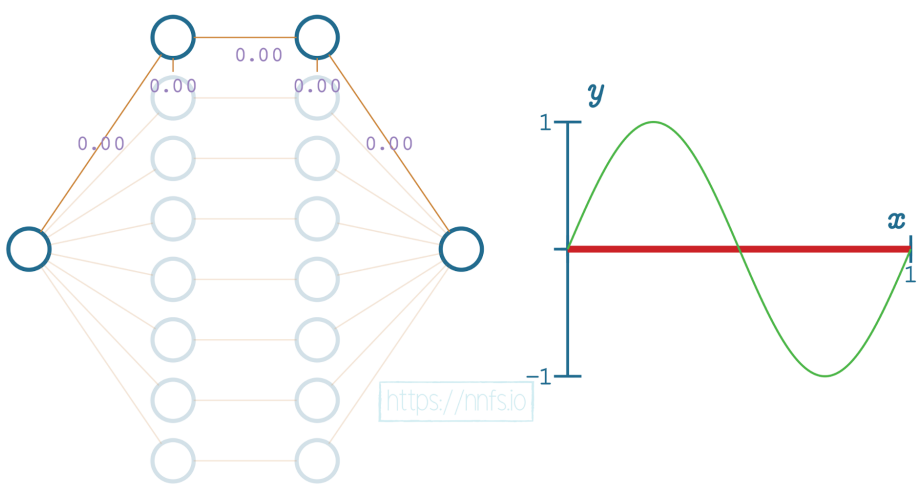


Fig 4.20: Hand-tuning a neural network starting with the first pair of neurons.

Next, we can set the weight for the hidden layer neurons and the output neuron to 1, and we can see how this impacts the output:

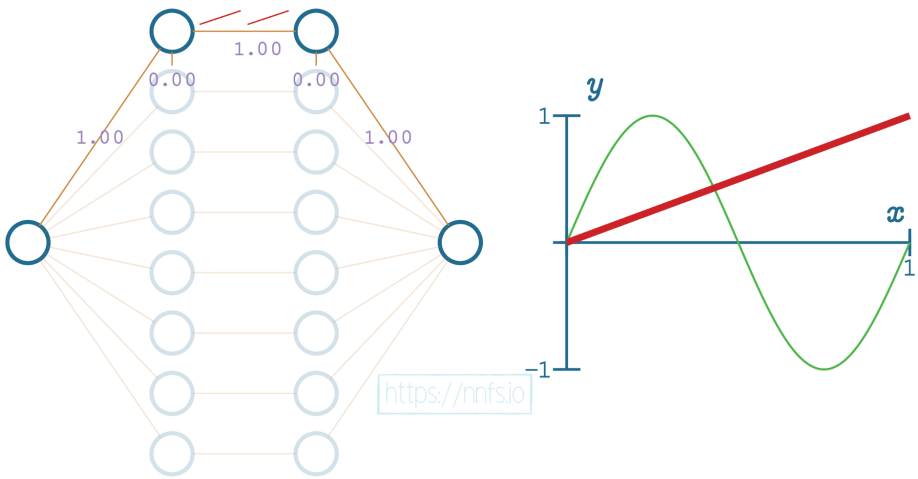


Fig 4.21: Adjusting weights for the first/top pair of neurons all to 1.

In this case, we can see that the slope of the overall function is impacted. We can further increase this slope by adjusting the weight for the first neuron of the first layer to 6.0:

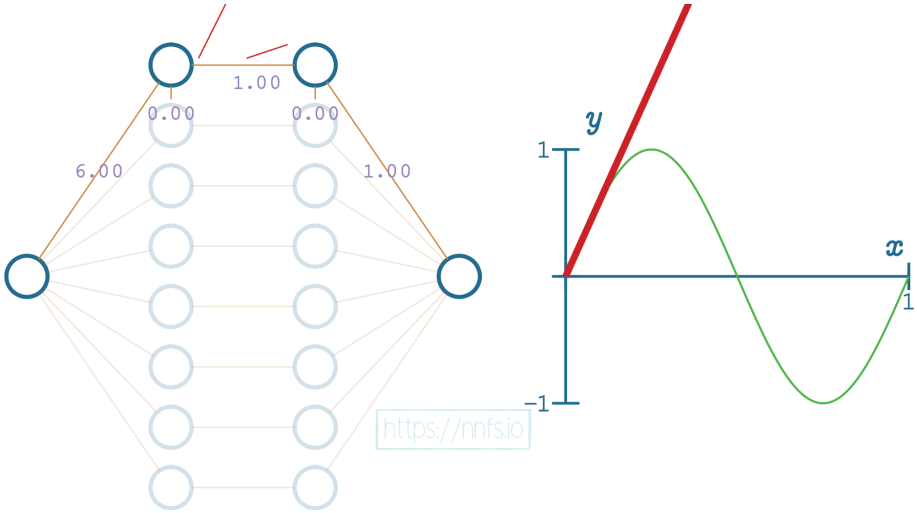


Fig 4.22: Setting weight for first hidden neuron to 6.

We can now see, for example, that the initial slope of this function is what we'd like, but we have a problem. Currently, this function never ends because this neuron pair never *deactivates*. We can visually see where we'd like the deactivation to occur. It's where the red fitment line (our current neural network's output) diverges initially from the green sine wave. So now, while we have the correct slope, we need to set this spot as our deactivation point. To do that, we start by increasing

the bias for the 2nd neuron of the hidden layer pair to 0.70. Recall that this offsets the overall function *vertically*:

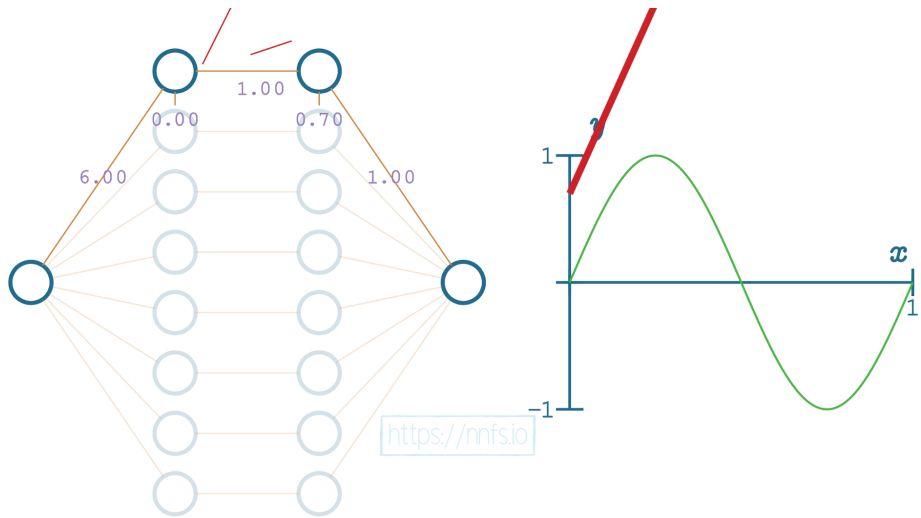


Fig 4.23: Using the bias for the 2nd hidden neuron in the top pair to offset function vertically.

Now we can set the weight for the 2nd neuron to -1, causing a deactivation point to occur, at least horizontally, where we want it:

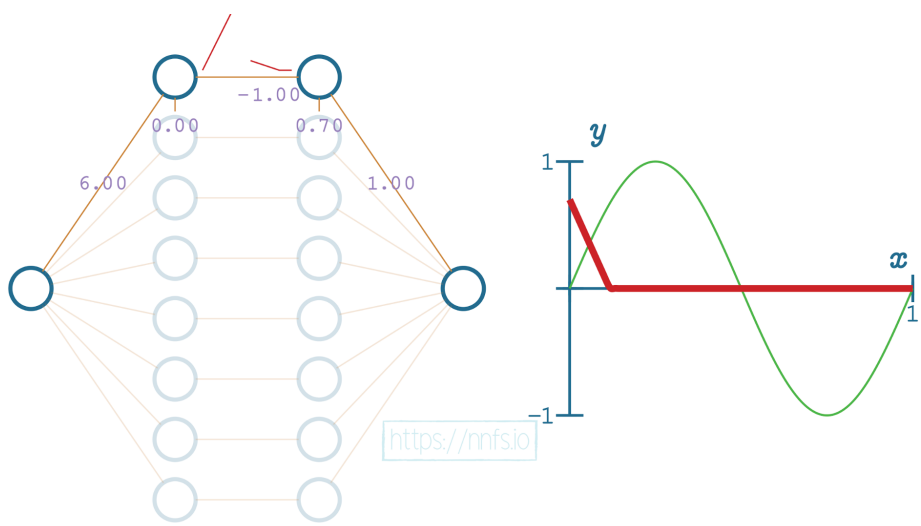


Fig 4.24: Setting the weight for the 2nd neuron in the top pair to -1.

Now we'd like to flip this slope back. How might we flip the output of these two neurons? It seems like we can take the weight of the connection to the output neuron, which is currently a 1.0, and just flip it to a -1, and that flips the function:

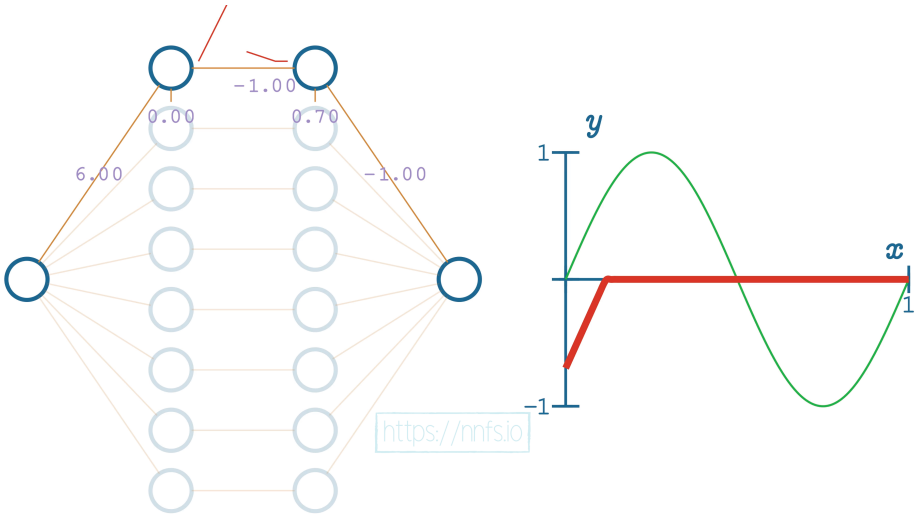


Fig 4.25: Setting the weight to the output neuron to -1.

We're certainly getting closer to making this first section fit how we want. Now, all we need to do is offset this up a bit. For this hand-optimized example, we're going to use the first 7 pairs of neurons in the hidden layers to create the sine wave's shape, then the bottom pair to offset everything vertically. If we set the bias of the 2nd neuron in the bottom pair to 1.0 and the weight to the output neuron as 0.7, we can vertically shift the line like so:

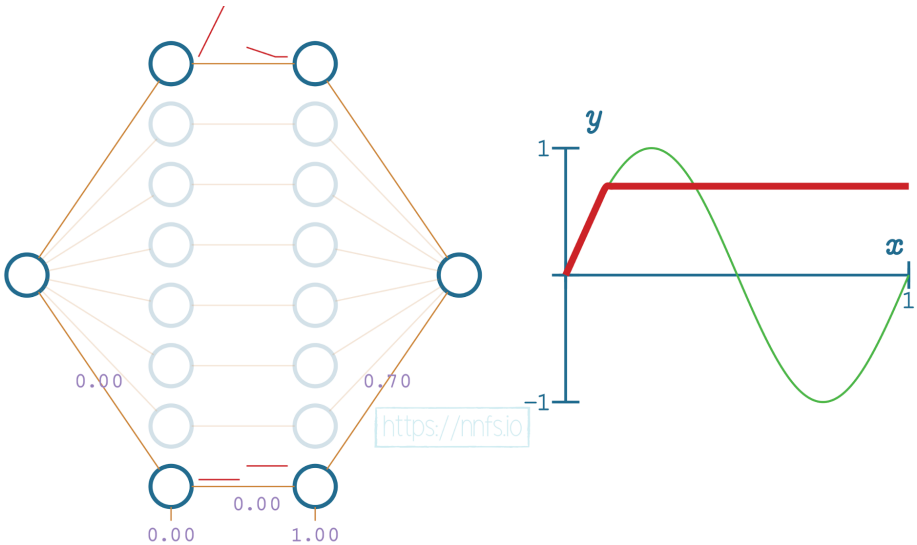


Fig 4.26: Using the bottom pair of neurons to offset the entire neural network function.

At this point, we have completed the first section with an "area of effect" being the first upward section of the sine wave. We can start on the next section that we wish to do. We can start by setting all weights for this 2nd pair of neurons to 1, including the output neuron:

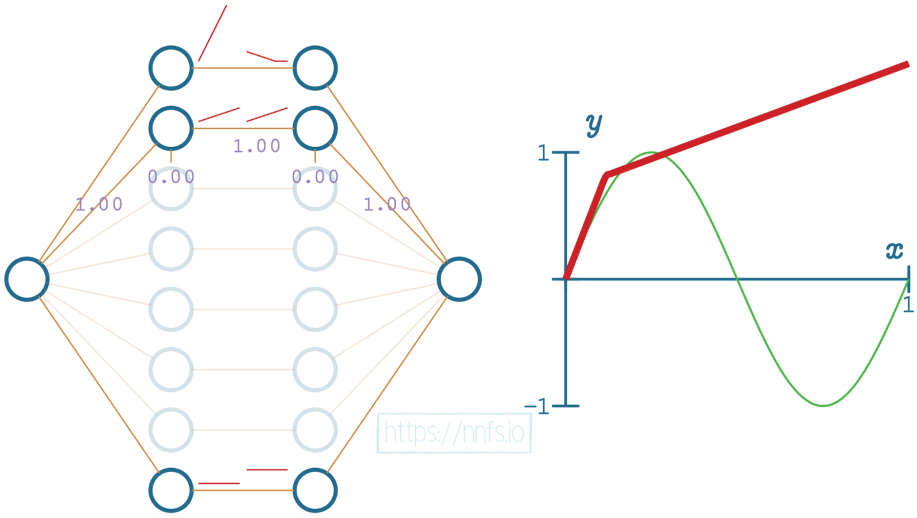


Fig 4.27: Starting to adjust the 2nd pair of neurons (from the top) for the next segment of the overall function.

At this point, this 2nd pair of neurons' activation is beginning too soon, which is impacting the "area of effect" of the top pair that we already aligned. To fix this, we want this second pair to start influencing the output where the first pair deactivates, so we want to adjust the function horizontally. As you can recall from earlier, we adjust the first neuron's bias in this neuron pair to achieve this. Also, to modify the slope, we'll set the weight coming into that first neuron for the 2nd pair, setting it to 3.5. This is the same method we used to set the slope for the first section, which is controlled by the top pair of neurons in the hidden layer. After these adjustments:

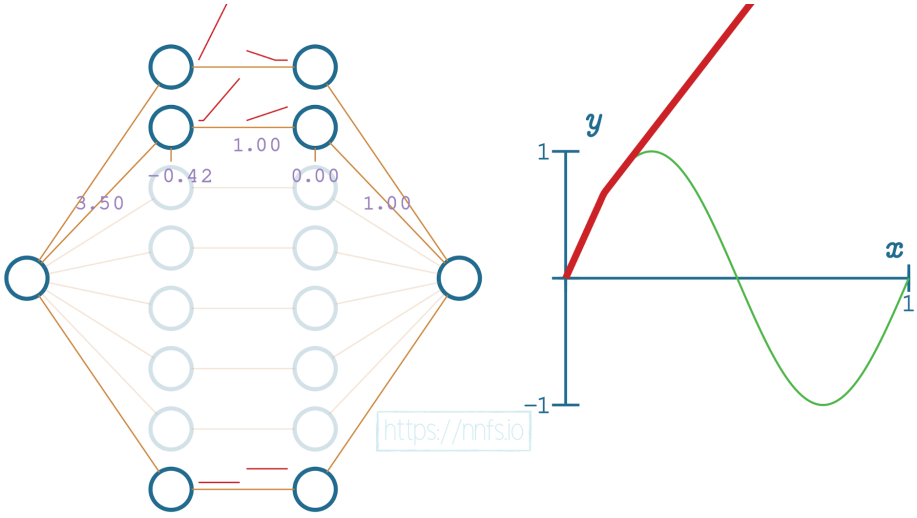


Fig 4.28: Adjusting the weight and bias into the first neuron of the 2nd pair.

We will now use the same methodology as we did with the first pair to set the deactivation point. We set the weight for the 2nd neuron in the hidden layer pair to -1 and the bias to 0.27.

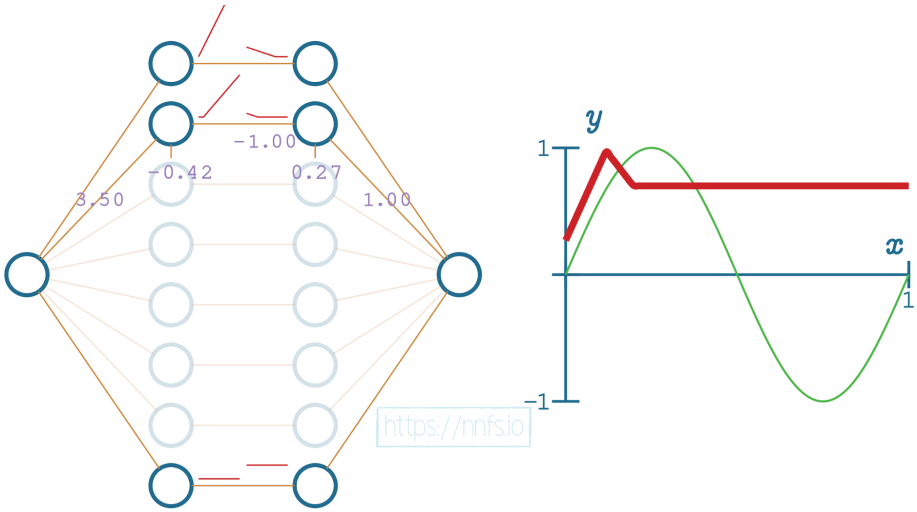


Fig 4.29: Adjusting the bias of the 2nd neuron in the 2nd pair.

Then we can flip this section's function, again the same way we did with the first one, by setting the weight to the output neuron from 1.0 to -1.0:

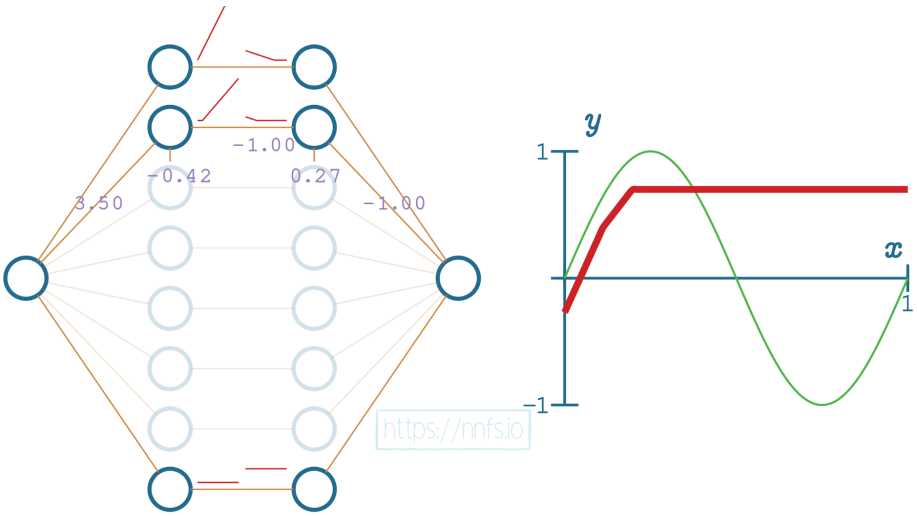


Fig 4.30: Flipping the 2nd pair's function segment, flipping the weight to the output neuron.

And again, just like the first pair, we will use the bottom pair to fix the vertical offset:

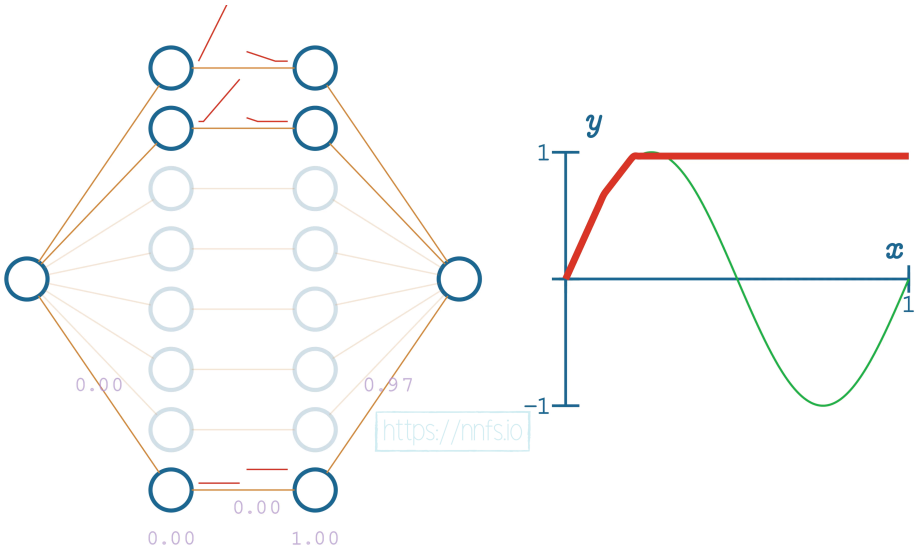


Fig 4.31: Using the bottom pair of neurons to adjust the network's overall function.

We then just continue with this methodology. We'll leave it flat for the top section, which means we will only begin the activation for the 3rd pair of hidden layer neurons when we wish for the slope to start going down:

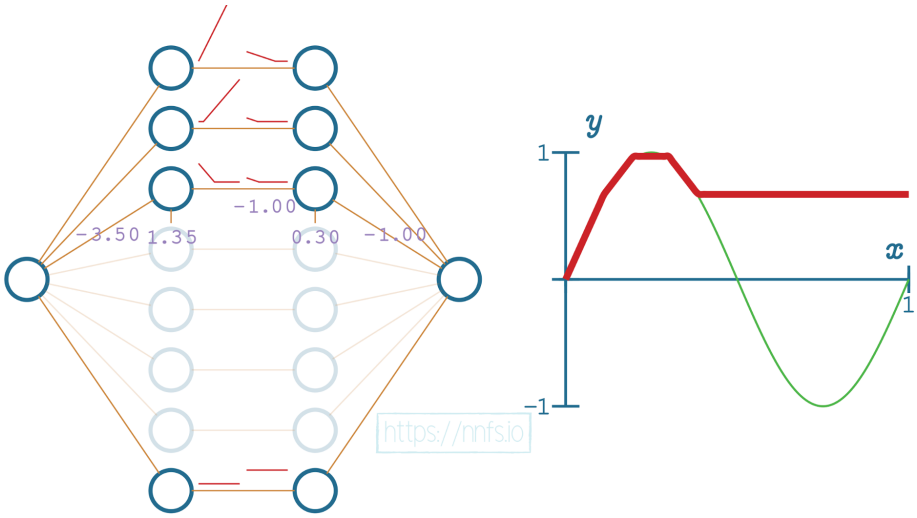


Fig 4.32: Adjusting the 3rd pair of neurons for the next segment.

This process is simply repeated for each section, giving us a final result:

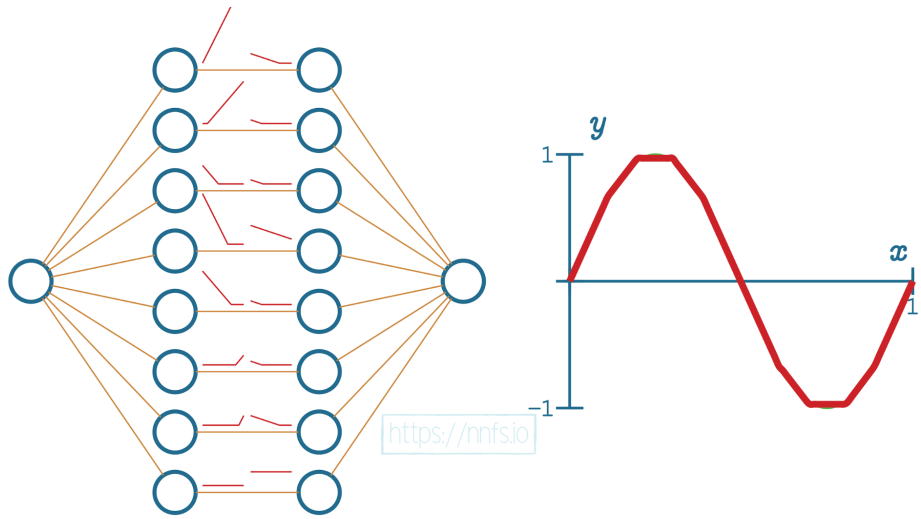


Fig 4.33: The completed process (see anim for all values).

We can then begin to pass data through to see how these neuron's areas of effect come into play — only when both neurons are activated based on input:

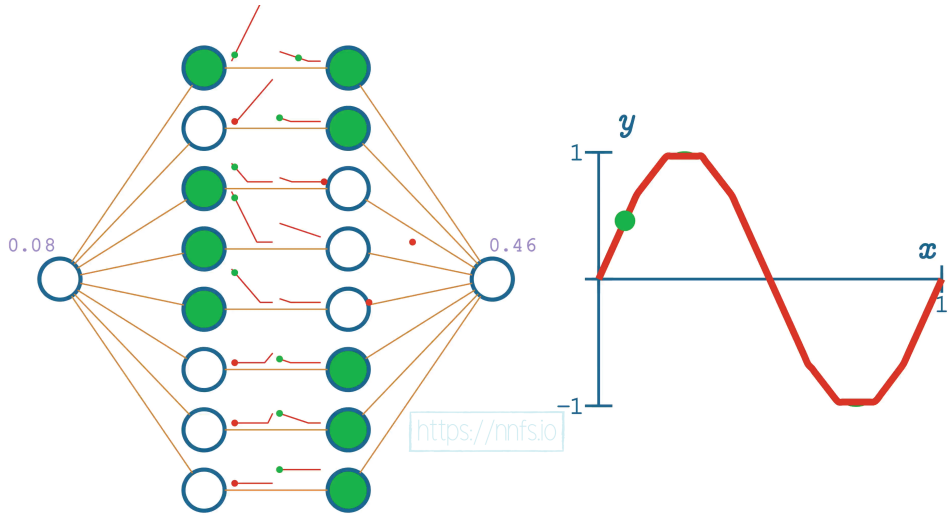


Fig 4.34: Example of data passing through this hand-crafted model.

In this case, given an input of 0.08, we can see the only pairs activated are the top ones, as this is their area of effect. Continuing with another example:

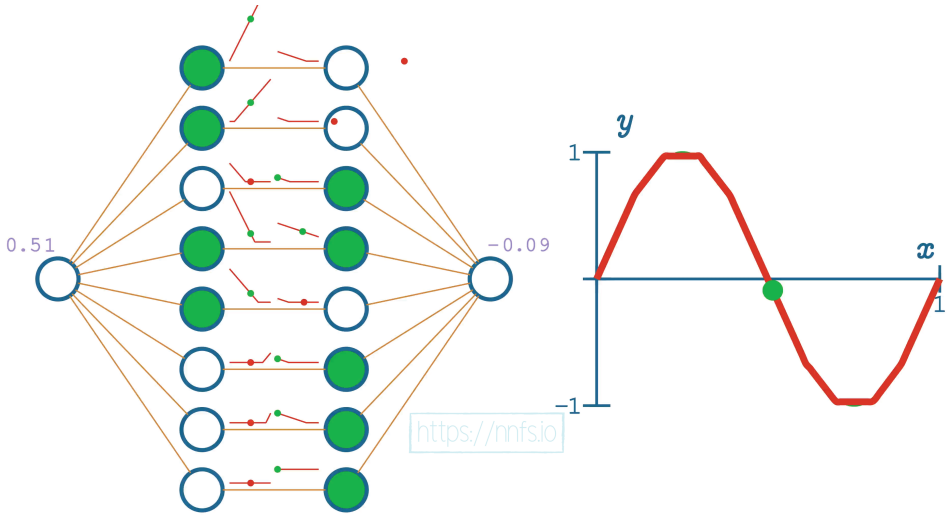


Fig 4.35: Example of data passing through this hand-crafted model.

In this case, only the fourth pair of neurons is activated. As you can see, even without any of the other weights, we've used some crude properties of a pair of neurons with rectified linear activation functions to fit this sine wave pretty well. If we enable all of the weights now and allow a mathematical optimizer to train, we can see even better fitment:

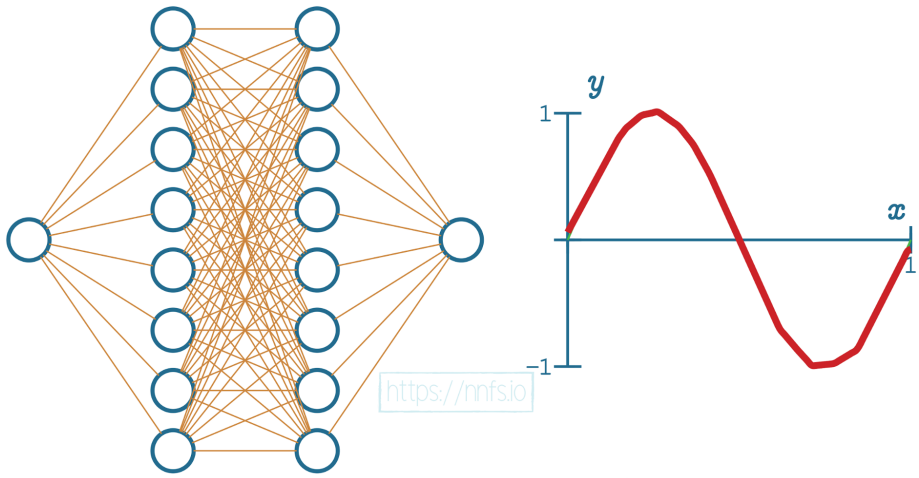


Fig 4.36: Example of fitment after fully-connecting the neurons and using an optimizer.

Animation for the entirety of the concept of ReLU fitment:

Anim 4.12-4.36: https://nnfs.io/mvp

It should begin to make more sense to you now how more neurons can enable more unique areas of effect, why we need two or more hidden layers, and why we need nonlinear activation functions to map nonlinear problems. For further example, we can take the above example with 2 hidden layers of 8 neurons each, and instead use 64 neurons per hidden layer, seeing the even further continued improvement:

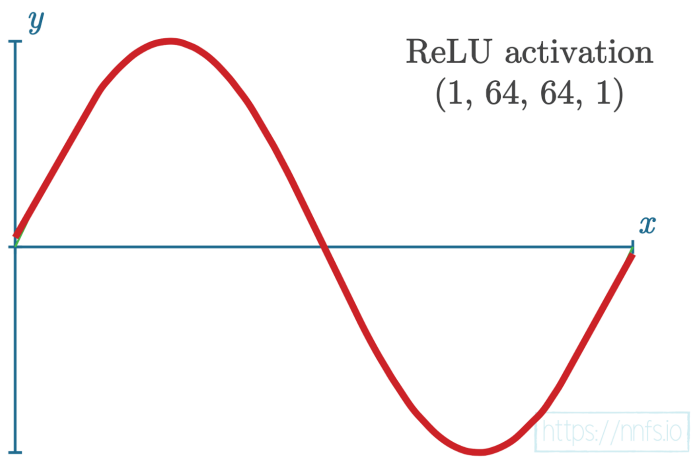


Fig 4.37: Fitment with 2 hidden layers of 64 neurons each, fully connected, with optimizer.

Anim 4.37: https://nnfs.io/moo

ReLU Activation Function Code

Despite the fancy sounding name, the rectified linear activation function is straightforward to code. Most closely to its definition:

```
inputs = [0, 2, -1, 3.3, -2.7, 1.1, 2.2, -100]

output = []
for i in inputs:
    if i > 0:
        output.append(i)
    else:
        output.append(0)

print(output)

>>>
[0, 2, 0, 3.3, 0, 1.1, 2.2, 0]
```

We made up a list of values to start. The ReLU in this code is a loop where we're checking if the current value is greater than 0. If it is, we're appending it to the output list, and if it's not, we're appending 0. This can be written more simply, as we just need to take the largest of two values: 0 or neuron value. For example:

```
inputs = [0, 2, -1, 3.3, -2.7, 1.1, 2.2, -100]

output = []
for i in inputs:
    output.append(max(0, i))

print(output)

>>>
[0, 2, 0, 3.3, 0, 1.1, 2.2, 0]
```

NumPy contains an equivalent — `np.maximum()`:

```
import numpy as np

inputs = [0, 2, -1, 3.3, -2.7, 1.1, 2.2, -100]
output = np.maximum(0, inputs)
print(output)

>>>
[0.  2.  0.  3.3 0.  1.1 2.2 0. ]
```

This method compares each element of the input list (or an array) and returns an object of the same shape filled with new values. We will use it in our new rectified linear activation class:

```
# ReLU activation
class Activation_ReLU:

    # Forward pass
    def forward(self, inputs):
        # Calculate output values from input
        self.output = np.maximum(0, inputs)
```

Let's apply this activation function to the dense layer's outputs in our code:

```
# Create dataset
X, y = spiral_data(samples=100, classes=3)

# Create Dense layer with 2 input features and 3 output values
dense1 = Layer_Dense(2, 3)

# Create ReLU activation (to be used with Dense layer):
activation1 = Activation_ReLU()

# Make a forward pass of our training data through this layer
dense1.forward(X)

# Forward pass through activation func.
# Takes in output from previous layer
activation1.forward(dense1.output)
```

```
# Let's see output of the first few samples:
print(activation1.output[:5])
```

```
>>>
[[0.         0.         0.        ]
 [0.         0.00011395 0.        ]
 [0.         0.00031729 0.        ]
 [0.         0.00052666 0.        ]
 [0.         0.00071401 0.        ]]
```

As you can see, negative values have been **clipped** (modified to be zero). That's all there is to the rectified linear activation function used in the hidden layer. Let's talk about the activation function that we are going to use on the output of the last layer.

The Softmax Activation Function

In our case, we're looking to get this model to be a classifier, so we want an activation function meant for classification. One of these is the Softmax activation function. First, why are we bothering with another activation function? It just depends on what our overall goals are. In this case, the rectified linear unit is unbounded, not normalized with other units, and exclusive. "Not normalized" implies the values can be anything, an output of *[12, 99, 318]* is without context, and "exclusive" means each output is independent of the others. To address this lack of context, the softmax activation on the output data can take in non-normalized, or uncalibrated, inputs and produce a normalized distribution of probabilities for our classes. In the case of classification, what we want to see is a prediction of which class the network "thinks" the input represents. This distribution returned by the softmax activation function represents **confidence scores** for each class and will add up to 1. The predicted class is associated with the output neuron that returned the largest confidence score. Still, we can also note the other confidence scores in our overarching algorithm/program that uses this network. For example, if our network has a confidence distribution for two classes: *[0.45, 0.55]*, the prediction is the 2nd class, but the confidence in this prediction isn't very high. Maybe our program would not act in this case since it's not very confident.

Here's the function for the **Softmax**:

$$S_{i,j} = \frac{e^{z_{i,j}}}{\sum_{l=1}^{L} e^{z_{i,l}}}$$

That might look daunting, but we can break it down into simple pieces and express it in Python code, which you may find is more approachable than the formula above. To start, here are example outputs from a neural network layer:

```
layer_outputs = [4.8, 1.21, 2.385]
```

The first step for us is to "exponentiate" the outputs. We do this with Euler's number, *e,* which is roughly *2.71828182846* and referred to as the "exponential growth" number. Exponentiating is taking this constant to the power of the given parameter:

$$y = e^x$$

Both the numerator and the denominator of the Softmax function contain *e* raised to the power of *z*, where *z*, given indices, means a singular output value — the index *i* means the current sample and the index *j* means the current output in this sample. The numerator exponentiates the current output value and the denominator takes a sum of all of the exponentiated outputs for a given sample. We need then to calculate these exponentiates to continue:

```
# Values from the previous output when we described
# what a neural network is
layer_outputs = [4.8, 1.21, 2.385]

# e - mathematical constant, we use E here to match a common coding
# style where constants are uppercased
E = 2.71828182846  # you can also use math.e

# For each value in a vector, calculate the exponential value
exp_values = []
for output in layer_outputs:
    exp_values.append(E ** output)  # ** - power operator in Python
print('exponentiated values:')
print(exp_values)
```

```
>>>
exponentiated values:
[121.51041751893969, 3.3534846525504487, 10.85906266492961]
```

Exponentiation serves multiple purposes. To calculate the probabilities, we need non-negative values. Imagine the output as `[4.8, 1.21, -2.385]` — even after normalization, the last value will still be negative since we'll just divide all of them by their sum. A negative probability (or confidence) does not make much sense. An exponential value of any number is always non-negative — it returns 0 for negative infinity, 1 for the input of 0, and increases for positive values:

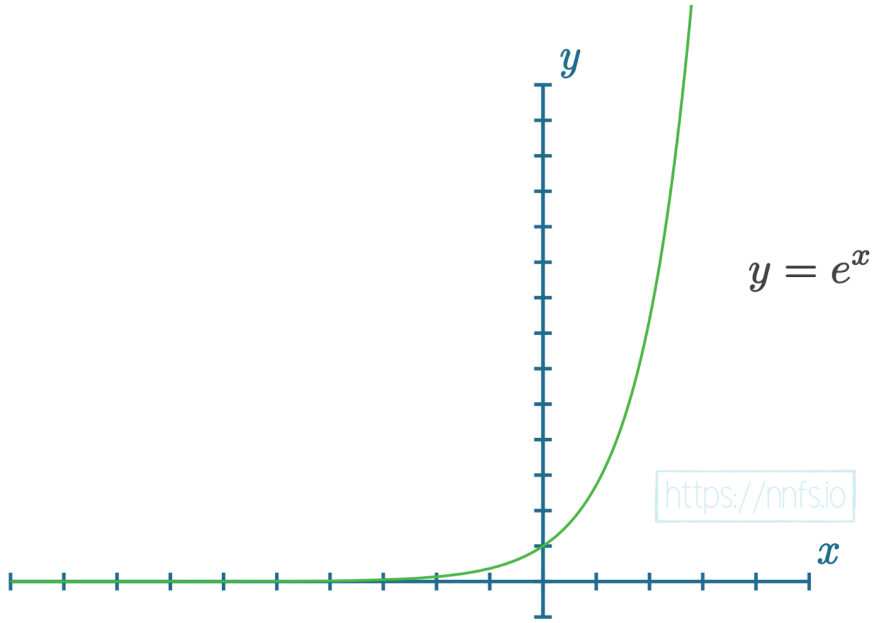


Fig 4.38: Graph of an exponential function.

The exponential function is a monotonic function. This means that, with higher input values, outputs are also higher, so we won't change the predicted class after applying it while making sure that we get non-negative values. It also adds stability to the result as the normalized exponentiation is more about the difference between numbers than their magnitudes. Once we've exponentiated, we want to convert these numbers to a probability distribution (converting the values into the vector of confidences, one for each class, which add up to 1 for everything in the vector). What that means is that we're about to perform a normalization where we take a given value and divide it by the sum of all of the values. For our outputs, exponentiated at this stage, that's what the equation of the Softmax function describes next — to take a given exponentiated value and divide it by the sum of all of the exponentiated values. Since each output value normalizes to a fraction of the sum, all of the values are now in the range of 0 to 1 and add up to 1 — they share the probability of 1 between themselves. Let's add the sum and normalization to the code:

```
# Now normalize values
norm_base = sum(exp_values)  # We sum all values
norm_values = []
for value in exp_values:
    norm_values.append(value / norm_base)
print('Normalized exponentiated values:')
print(norm_values)

print('Sum of normalized values:', sum(norm_values))
```

```
>>>
Normalized exponentiated values:
[0.8952826639573506, 0.024708306782070668, 0.08000902926057876]
Sum of normalized values: 1.0
```

We can perform the same set of operations with the use of NumPy in the following way:

```
import numpy as np

# Values from the earlier previous when we described
# what a neural network is
layer_outputs = [4.8, 1.21, 2.385]

# For each value in a vector, calculate the exponential value
exp_values = np.exp(layer_outputs)
print('exponentiated values:')
print(exp_values)

# Now normalize values
norm_values = exp_values / np.sum(exp_values)
print('normalized exponentiated values:')
print(norm_values)
print('sum of normalized values:', np.sum(norm_values))

>>>
exponentiated values:
[121.51041752   3.35348465  10.85906266]
normalized exponentiated values:
[0.89528266 0.02470831 0.08000903]
sum of normalized values: 0.9999999999999999
```

Notice the results are similar, but faster to calculate and the code is easier to read with NumPy. We can exponentiate all of the values with a single call of the `np.exp()`, then immediately normalize them with the sum. To train in batches, we need to convert this functionality to accept layer outputs in batches. Doing this is as easy as:

```
# Get unnormalized probabilities
exp_values = np.exp(inputs)

# Normalize them for each sample
probabilities = exp_values / np.sum(exp_values, axis=1, keepdims=True)
```

We have some new functions. Specifically, `np.exp()` does the `E**output` part. We should also address what *axis* and *keepdims* mean in the above. Let's first discuss the *axis*. Axis is easier to show than tell, but, in a 2D array/matrix, axis 0 refers to the rows, and axis 1 refers to the columns. Let's see some examples of how *axis* affects the sum using NumPy. First, we will just show the default, which is *None.*

```
import numpy as np


layer_outputs = np.array([[4.8, 1.21, 2.385],
                          [8.9, -1.81, 0.2],
                          [1.41, 1.051, 0.026]])

print('Sum without axis')
print(np.sum(layer_outputs))

print('This will be identical to the above since default is None:')
print(np.sum(layer_outputs, axis=None))
```

```
>>>
Sum without axis
18.172
This will be identical to the above since default is None:
18.172
```

With no axis specified, we are just summing all of the values, even if they're in varying dimensions. Next, `axis=0`. This means to sum row-wise, along axis 0. In other words, the output has the same size as this axis, as at each of the positions of this output, the values from all the other dimensions at this position are summed to form it. In the case of our 2D array, where we have only a single other dimension, the columns, the output vector will sum these columns. This means we'll perform *4.8+8.9+1.41* and so on.

```
print('Another way to think of it w/ a matrix == axis 0: columns:')
print(np.sum(layer_outputs, axis=0))
```

```
>>>
Another way to think of it w/ a matrix == axis 0: columns:
[15.11   0.451  2.611]
```

This isn't what we want, though. We want sums of the rows. You can probably guess how to do this with NumPy, but we'll still show the "from scratch" version:

```
print('But we want to sum the rows instead, like this w/ raw py:')

for i in layer_outputs:
    print(sum(i))

>>>
But we want to sum the rows instead, like this w/ raw py:
8.395
7.29
2.4869999999999997
```

With the above, we could append these to some list in any way we want. That said, we're going to use NumPy. As you probably guessed, we're going to sum along axis 1:

```
print('So we can sum axis 1, but note the current shape:')
print(np.sum(layer_outputs, axis=1))

>>>
So we can sum axis 1, but note the current shape:
[8.395 7.29 2.487]
```

As pointed out by "`note the current shape`," we did get the sums that we expected, but actually, we want to simplify the outputs to a single value per sample. We're trying to sum all the outputs from a layer for each sample in a batch; converting the layer's output array with row length equal to the number of neurons in the layer, to just one value. We need a column vector with these values since it will let us normalize the whole batch of samples, sample-wise, with a single calculation.

```
print('Sum axis 1, but keep the same dimensions as input:')
print(np.sum(layer_outputs, axis=1, keepdims=True))

>>>
Sum axis 1, but keep the same dimensions as input:
[[8.395]
 [7.29 ]
 [2.487]]
```

With this, we keep the same dimensions as the input. Now, if we divide the array containing a batch of the outputs with this array, NumPy will perform this sample-wise. That means that it'll divide all of the values from each output row by the corresponding row from the sum array. Since this sum in each row is a single value, it'll be used for the division with every value from the corresponding output row). We can combine all of this into a softmax class, like:

```
# Softmax activation
class Activation_Softmax:

    # Forward pass
    def forward(self, inputs):

        # Get unnormalized probabilities
        exp_values = np.exp(inputs - np.max(inputs, axis=1,
                                            keepdims=True))
        # Normalize them for each sample
        probabilities = exp_values / np.sum(exp_values, axis=1,
                                            keepdims=True)

        self.output = probabilities
```

Finally, we also included a subtraction of the largest of the inputs before we did the exponentiation. There are two main pervasive challenges with neural networks: "dead neurons" and very large numbers (referred to as "exploding" values). "Dead" neurons and enormous numbers can wreak havoc down the line and render a network useless over time. The exponential function used in softmax activation is one of the sources of exploding values. Let's see some examples of how and why this can easily happen:

```
import numpy as np

print(np.exp(1))

>>>
2.718281828459045


print(np.exp(10))

>>>
22026.465794806718
```

```
print(np.exp(100))

>>>
2.6881171418161356e+43


print(np.exp(1000))

>>>
__main__:1: RuntimeWarning: overflow encountered in exp
inf
```

It doesn't take a very large number, in this case, a mere *1,000*, to cause an overflow error. We know the exponential function tends toward 0 as its input value approaches negative infinity, and the output is 1 when the input is 0 (as shown in the chart earlier):

```
import numpy as np

print(np.exp(-np.inf), np.exp(0))

>>>
0.0 1.0
```

We can use this property to prevent the exponential function from overflowing. Suppose we subtract the maximum value from a list of input values. We would then change the output values to always be in a range from some negative value up to 0, as the largest number subtracted by itself returns 0, and any smaller number subtracted by it will result in a negative number — exactly the range discussed above. With Softmax, thanks to the normalization, we can subtract any value from all of the inputs, and it will not change the output:

```
softmax = Activation_Softmax()

softmax.forward([[1, 2, 3]])
print(softmax.output)

>>>
[[0.09003057 0.24472847 0.66524096]]
```

```
softmax.forward([[-2, -1, 0]])  # subtracted 3 - max from the list
print(softmax.output)

>>>
[[0.09003057 0.24472847 0.66524096]]
```

This is another useful property of the exponentiated and normalized function. There's one more thing to mention in addition to these calculations. What happens if we divide the layer's output data, `[1, 2, 3]`, for example, by 2?

```
softmax.forward([[0.5, 1, 1.5]])
print(softmax.output)

>>>
[[0.18632372 0.30719589 0.50648039]]
```

The output confidences have changed due to the nonlinearity nature of the exponentiation. This is one example of why we need to scale all of the input data to a neural network in the same way, which we'll explain in further detail in chapter 22.

Now, we can add another dense layer as the output layer, setting it to contain as many inputs as the previous layer has outputs and as many outputs as our data includes classes. Then we can apply the softmax activation to the output of this new layer:

```
# Create dataset
X, y = spiral_data(samples=100, classes=3)

# Create Dense layer with 2 input features and 3 output values
dense1 = Layer_Dense(2, 3)

# Create ReLU activation (to be used with Dense layer):
activation1 = Activation_ReLU()

# Create second Dense layer with 3 input features (as we take output
# of previous layer here) and 3 output values
dense2 = Layer_Dense(3, 3)

# Create Softmax activation (to be used with Dense layer):
activation2 = Activation_Softmax()

# Make a forward pass of our training data through this layer
dense1.forward(X)
```

```
# Make a forward pass through activation function
# it takes the output of first dense layer here
activation1.forward(dense1.output)

# Make a forward pass through second Dense layer
# it takes outputs of activation function of first layer as inputs
dense2.forward(activation1.output)

# Make a forward pass through activation function
# it takes the output of second dense layer here
activation2.forward(dense2.output)

# Let's see output of the first few samples:
print(activation2.output[:5])
```

```
>>>
[[0.33333334 0.33333334 0.33333334]
 [0.33333316 0.3333332  0.33333364]
 [0.33333287 0.3333329  0.33333418]
 [0.3333326  0.33333263 0.33333477]
 [0.33333233 0.3333324  0.33333528]]
```

As you can see, the distribution of predictions is almost equal, as each of the samples has ~33% (0.33) predictions for each class. This results from the random initialization of weights (a draw from the normal distribution, as not every random initialization will result in this) and zeroed biases. These outputs are also our "confidence scores." To determine which classification the model has chosen to be the prediction, we perform an *argmax* on these outputs, which checks which of the classes in the output distribution has the highest confidence and returns its index - the predicted class index. That said, the confidence score can be as important as the class prediction itself. For example, the argmax of `[0.22, 0.6, 0.18]` is the same as the argmax for `[0.32, 0.36, 0.32]`. In both of these, the argmax function would return an index value of 1 (the 2nd element in Python's zero-indexed paradigm), but obviously, a 60% confidence is much better than a 36% confidence.

Full code up to this point:

```
import numpy as np
import nnfs
from nnfs.datasets import spiral_data

nnfs.init()


# Dense layer
class Layer_Dense:

    # Layer initialization
    def __init__(self, n_inputs, n_neurons):
        # Initialize weights and biases
        self.weights = 0.01 * np.random.randn(n_inputs, n_neurons)
        self.biases = np.zeros((1, n_neurons))

    # Forward pass
    def forward(self, inputs):
        # Calculate output values from inputs, weights and biases
        self.output = np.dot(inputs, self.weights) + self.biases


# ReLU activation
class Activation_ReLU:

    # Forward pass
    def forward(self, inputs):
        # Calculate output values from inputs
        self.output = np.maximum(0, inputs)
```

```
# Softmax activation
class Activation_Softmax:

    # Forward pass
    def forward(self, inputs):

        # Get unnormalized probabilities
        exp_values = np.exp(inputs - np.max(inputs, axis=1,
                                            keepdims=True))
        # Normalize them for each sample
        probabilities = exp_values / np.sum(exp_values, axis=1,
                                            keepdims=True)

        self.output = probabilities


# Create dataset
X, y = spiral_data(samples=100, classes=3)

# Create Dense layer with 2 input features and 3 output values
dense1 = Layer_Dense(2, 3)

# Create ReLU activation (to be used with Dense layer):
activation1 = Activation_ReLU()

# Create second Dense layer with 3 input features (as we take output
# of previous layer here) and 3 output values
dense2 = Layer_Dense(3, 3)

# Create Softmax activation (to be used with Dense layer):
activation2 = Activation_Softmax()

# Make a forward pass of our training data through this layer
dense1.forward(X)

# Make a forward pass through activation function
# it takes the output of first dense layer here
activation1.forward(dense1.output)

# Make a forward pass through second Dense layer
# it takes outputs of activation function of first layer as inputs
dense2.forward(activation1.output)

# Make a forward pass through activation function
# it takes the output of second dense layer here
activation2.forward(dense2.output)
```

```
# Let's see output of the first few samples:
print(activation2.output[:5])
```

```
>>>
[[0.33333334 0.33333334 0.33333334]
 [0.33333316 0.3333332  0.33333364]
 [0.33333287 0.3333329  0.33333418]
 [0.3333326  0.33333263 0.33333477]
 [0.33333233 0.3333324  0.33333528]]
```

We've completed what we need for forward-passing data through our model. We used the **Rectified Linear (ReLU**) activation function on the hidden layer, which works on a per-neuron basis. We additionally used the **Softmax** activation function for the output layer since it accepts non-normalized values as input and outputs a probability distribution, which we're using as confidence scores for each class. Recall that, although neurons are interconnected, they each have their respective weights and biases and are not "normalized" with each other.

As you can see, our example model is currently random. To remedy this, we need a way to calculate how wrong the neural network is at current predictions and begin adjusting weights and biases to decrease error over time. Thus, our next step is to quantify how wrong the model is through what's defined as a **loss function**.

Supplementary Material: https://nnfs.io/ch4
Chapter code, further resources, and errata for this chapter.

Chapter 5

Calculating Network Error with Loss

With a randomly-initialized model, or even a model initialized with more sophisticated approaches, our goal is to train, or teach, a model over time. To train a model, we tweak the weights and biases to improve the model's accuracy and confidence. To do this, we calculate how much error the model has. The **loss function**, also referred to as the **cost function**, is the algorithm that quantifies how wrong a model is. **Loss** is the measure of this metric. Since loss is the model's error, we ideally want it to be 0.

You may wonder why we do not calculate the error of a model based on the argmax accuracy. Recall our earlier example of confidence: `[0.22, 0.6, 0.18]` vs `[0.32, 0.36, 0.32]`. If the correct class were indeed the middle one (index 1), the model accuracy would be identical between the two above. But are these two examples *really* as accurate as each other? They are not, because accuracy is simply applying an argmax to the output to find the index of the biggest value. The output of a neural network is actually confidence, and more confidence in the correct

answer is better. Because of this, we strive to increase correct confidence and decrease misplaced confidence.

Categorical Cross-Entropy Loss

If you're familiar with linear regression, then you already know one of the loss functions used with neural networks that do regression: **squared error** (or **mean squared error** with neural networks).

We're not performing regression in this example; we're classifying, so we need a different loss function. The model has a softmax activation function for the output layer, which means it's outputting a probability distribution. **Categorical cross-entropy** is explicitly used to compare a "ground-truth" probability (*y* or "*targets*") and some predicted distribution (*y-hat* or "*predictions*"), so it makes sense to use cross-entropy here. It is also one of the most commonly used loss functions with a softmax activation on the output layer.

The formula for calculating the categorical cross-entropy of *y* (actual/desired distribution) and *y-hat* (predicted distribution) is:

$$L_i = -\sum_j y_{i,j} log(\hat{y}_{i,j})$$

Where L_i denotes sample loss value, *i* is the i-th sample in the set, *j* is the label/output index, *y* denotes the target values, and *y-hat* denotes the predicted values.

Once we start coding the solution, we'll simplify it further to *-log(correct_class_confidence)*, the formula for which is:

$$L_i = -\log\left(\hat{y}_{i,k}\right) \qquad where\ \mathbf{k}\ is\ an\ index\ of\ "true"\ probability$$

Where L_i denotes sample loss value, *i* is the i-th sample in a set, *k* is the index of the target label (ground-true label), *y* denotes the target values and *y-hat* denotes the predicted values.

You may ask why we call this cross-entropy and not **log loss**, which is also a type of loss. If you do not know what log loss is, you may wonder why there is such a fancy looking formula for what looks to be a fairly basic description.

In general, the log loss error function is what we apply to the output of a binary logistic regression model (which we'll describe in chapter 16) — there are only two classes in the distribution, each of them applying to a single output (neuron) which is targeted as a 0 or 1. In our case, we have a classification model that returns a probability distribution over all of the outputs. Cross-entropy compares two probability distributions. In our case, we have a softmax output, let's say it's:

```
softmax_output = [0.7, 0.1, 0.2]
```

Which probability distribution do we intend to compare this to? We have 3 class confidences in the above output, and let's assume that the desired prediction is the first class (index 0, which is currently 0.7). If that's the intended prediction, then the desired probability distribution is `[1, 0, 0]`. Cross-entropy can also work on probability distributions like `[0.2, 0.5, 0.3]`; they wouldn't have to look like the one above. That said, the desired probabilities will consist of a 1 in the desired class, and a 0 in the remaining undesired classes. Arrays or vectors like this are called **one-hot**, meaning one of the values is "hot" (on), with a value of 1, and the rest are "cold" (off), with values of 0. When comparing the model's results to a one-hot vector using cross-entropy, the other parts of the equation zero out, and the target probability's log loss is multiplied by 1, making the cross-entropy calculation relatively simple. This is also a special case of the cross-entropy calculation, called categorical cross-entropy. To exemplify this — if we take a softmax output of `[0.7, 0.1, 0.2]` and targets of `[1, 0, 0]`, we can apply the calculations as follows:

$$L_i = -\sum_j y_{i,j} log(\hat{y}_{i,j}) = -(1 \cdot log(0.7) + 0 \cdot log(0.1) + 0 \cdot log(0.2)) =$$
$$= -(-0.35667494393873245 + 0 + 0) = 0.35667494393873245$$

Let's see the Python code for this:

```
import math


# An example output from the output layer of the neural network
softmax_output = [0.7, 0.1, 0.2]
# Ground truth
target_output = [1, 0, 0]

loss = -(math.log(softmax_output[0])*target_output[0] +
         math.log(softmax_output[1])*target_output[1] +
         math.log(softmax_output[2])*target_output[2])

print(loss)

>>>
0.35667494393873245
```

That's the full categorical cross-entropy calculation, but we can make a few assumptions given one-hot target vectors. First, what are the values for `target_output[1]` and `target_output[2]` in this case? They're both 0, and anything multiplied by 0 is 0. Thus, we don't need to calculate these indices. Next, what's the value for `target_output[0]` in this case? It's 1. So this can be omitted as any number multiplied by 1 remains the same. The same output then, in this example, can be calculated with:

```
loss = -math.log(softmax_output[0])
```

Which still gives us:

```
>>>
0.35667494393873245
```

As you can see with one-hot vector targets, or scalar values that represent them, we can make some simple assumptions and use a more basic calculation — what was once an involved formula reduces to the negative log of the target class' confidence score — the second formula presented at the beginning of this chapter.

As we've already discussed, the example confidence level might look like `[0.22, 0.6, 0.18]` or `[0.32, 0.36, 0.32]`. In both cases, the *argmax* of these vectors will return the second class as the prediction, but the model's confidence about these predictions is high only for one of them. The **Categorical Cross-Entropy Loss** accounts for that and outputs a larger loss the lower the confidence is:

```
import math

print(math.log(1.))
print(math.log(0.95))
print(math.log(0.9))
print(math.log(0.8))
print('...')
print(math.log(0.2))
print(math.log(0.1))
print(math.log(0.05))
print(math.log(0.01))
```

```
>>>
0.0
-0.05129329438755058
-0.10536051565782628
-0.2231435513142097
...
-1.6094379124341003
-2.3025850929940455
-2.995732273553991
-4.605170185988091
```

We've printed different log values for a few example confidences. When the confidence level equals *1*, meaning the model is 100% "sure" about its prediction, the loss value for this sample equals *0*. The loss value raises with the confidence level, approaching 0. You might also wonder why we did not print the result of *log(0)* — we'll explain that shortly.

So far, we've applied log() to the softmax output, but have neither explained what "log" is nor why we use it. We will save the discussion of "why" until the next chapter, which covers derivatives, gradients, and optimizations; suffice it to say that the log function has some desirable properties. **Log** is short for **logarithm** and is defined as the solution for the x-term in an equation of the form $a^x = b$. For example, $10^x = 100$ can be solved with a log: $log_{10}(100)$, which evaluates to 2. This property of the log function is *especially* beneficial when *e* (Euler's number or *~2.71828*) is used in the base (where 10 is in the example). The logarithm with *e* as its base is referred to as the **natural logarithm**, **natural log**, or simply **log** — you may also see this written as **ln**: $ln(x) = log(x) = log_e(x)$ The variety of conventions can make this confusing, so to simplify things, **any mention of log will always be a natural logarithm throughout this book**. The natural log represents the solution for the x-term in the equation $e^x = b$; for example, $e^x = 5.2$ is solved by *log(5.2)*.

In Python code:

```
import numpy as np

b = 5.2
print(np.log(b))

>>>
1.6486586255873816
```

We can confirm this by exponentiating our result:

```
import math

print(math.e ** 1.6486586255873816)

>>>
5.199999999999999
```

The small difference is the result of floating-point precision in Python. Getting back to the loss calculation, we need to modify our output in two additional ways. First, we'll update our process to work on batches of softmax output distributions; and second, make the negative log calculation dynamic to the target index (the target index has been hard-coded so far).

Consider a scenario with a neural network that performs classification between three classes, and the neural network classifies in batches of three. After running through the softmax activation function with a batch of 3 samples and 3 classes, the network's output layer yields:

```
# Probabilities for 3 samples
softmax_outputs = np.array([[0.7, 0.1, 0.2],
                            [0.1, 0.5, 0.4],
                            [0.02, 0.9, 0.08]])
```

We need a way to dynamically calculate the categorical cross-entropy, which we now know is a negative log calculation. To determine which value in the softmax output to calculate the negative log from, we simply need to know our target values. In this example, there are 3 classes; let's say we're trying to classify something as a "dog," "cat," or "human." A dog is class 0 (at index 0), a cat class 1 (index 1), and a human class 2 (index 2). Let's assume the batch of three sample inputs to this neural network is being mapped to the target values of a dog, cat, and cat. So the targets (as

a list of target indices) would be *[0, 1, 1]*.

```
softmax_outputs = [[0.7, 0.1, 0.2],
                   [0.1, 0.5, 0.4],
                   [0.02, 0.9, 0.08]]

class_targets = [0, 1, 1]  # dog, cat, cat
```

The first value, 0, in `class_targets` means the first softmax output distribution's intended prediction was the one at the 0th index of `[0.7, 0.1, 0.2]`; the model has a `0.7` confidence score that this observation is a dog. This continues throughout the batch, where the intended target of the 2nd softmax distribution, `[0.1, 0.5, 0.4]`, was at an index of 1; the model only has a `0.5` confidence score that this is a cat — the model is less certain about this observation. In the last sample, it's also the 2nd index from the softmax distribution, a value of `0.9` in this case — a pretty high confidence.

With a collection of softmax outputs and their intended targets, we can map these indices to retrieve the values from the softmax distributions:

```
softmax_outputs = [[0.7, 0.1, 0.2],
                   [0.1, 0.5, 0.4],
                   [0.02, 0.9, 0.08]]

class_targets = [0, 1, 1]

for targ_idx, distribution in zip(class_targets, softmax_outputs):
    print(distribution[targ_idx])

>>>
0.7
0.5
0.9
```

The `zip()` function, again, lets us iterate over multiple iterables at the same time in Python. This can be further simplified using NumPy (we're creating a NumPy array of the Softmax outputs this time):

```
softmax_outputs = np.array([[0.7, 0.1, 0.2],
                            [0.1, 0.5, 0.4],
                            [0.02, 0.9, 0.08]])
class_targets = [0, 1, 1]
```

```
print(softmax_outputs[[0, 1, 2], class_targets])

>>>
[0.7 0.5 0.9]
```

What are the 0, 1, and 2 values? NumPy lets us index an array in multiple ways. One of them is to use a list filled with indices and that's convenient for us — we could use the `class_targets` for this purpose as it already contains the list of indices that we are interested in. The problem is that this has to filter data rows in the array — the second dimension. To perform that, we also need to explicitly filter this array in its first dimension. This dimension contains the predictions and we, of course, want to retain them all. We can achieve that by using a list containing numbers from 0 through all of the indices. We know we're going to have as many indices as distributions in our entire batch, so we can use a `range()` instead of typing each value ourselves:

```
print(softmax_outputs[
    range(len(softmax_outputs)), class_targets
])

>>>
[0.7 0.5 0.9]
```

This returns a list of the confidences at the target indices for each of the samples. Now we apply the negative log to this list:

```
print(-np.log(softmax_outputs[
    range(len(softmax_outputs)), class_targets
]))

>>>
[0.35667494 0.69314718 0.10536052]
```

Finally, we want an average loss per batch to have an idea about how our model is doing during training. There are many ways to calculate an average in Python; the most basic form of an average is the **arithmetic mean**: *sum(iterable) / len(iterable)*. NumPy has a method that computes this average on arrays, so we will use that instead. We add NumPy's average to the code:

```
neg_log = -np.log(softmax_outputs[
              range(len(softmax_outputs)), class_targets
          ])
average_loss = np.mean(neg_log)
print(average_loss)

>>>
0.38506088005216804
```

We have already learned that targets can be one-hot encoded, where all values, except for one, are zeros, and the correct label's position is filled with 1. They can also be sparse, which means that the numbers they contain are the correct class numbers — we are generating them this way with the `spiral_data()` function, and we can allow the loss calculation to accept any of these forms. Since we implemented this to work with sparse labels (as in our training data), we have to add a check if they are one-hot encoded and handle it a bit differently in this new case. The check can be performed by counting the dimensions — if targets are single-dimensional (like a list), they are sparse, but if there are 2 dimensions (like a list of lists), then there is a set of one-hot encoded vectors. In this second case, we'll implement a solution using the first equation from this chapter, instead of filtering out the confidences at the target labels. We have to multiply confidences by the targets, zeroing out all values except the ones at correct labels, performing a sum along the row axis (axis *1*). We have to add a test to the code we just wrote for the number of dimensions, move calculations of the log values outside of this new *if* statement, and implement the solution for the one-hot encoded labels following the first equation:

```
import numpy as np

softmax_outputs = np.array([[0.7, 0.1, 0.2],
                            [0.1, 0.5, 0.4],
                            [0.02, 0.9, 0.08]])
class_targets = np.array([[1, 0, 0],
                          [0, 1, 0],
                          [0, 1, 0]])


# Probabilities for target values -
# only if categorical labels
if len(class_targets.shape) == 1:
    correct_confidences = softmax_outputs[
        range(len(softmax_outputs)),
        class_targets
    ]
```

```
# Mask values - only for one-hot encoded labels
elif len(class_targets.shape) == 2:
    correct_confidences = np.sum(
        softmax_outputs*class_targets,
        axis=1)

# Losses
neg_log = -np.log(correct_confidences)

average_loss = np.mean(neg_log)
print(average_loss)

>>>
0.38506088005216804
```

Before we move on, there is one additional problem to solve. The softmax output, which is also an input to this loss function, consists of numbers in the range from 0 to 1 - a list of confidences. It is possible that the model will have full confidence for one label making all the remaining confidences zero. Similarly, it is also possible that the model will assign full confidence to a value that wasn't the target. If we then try to calculate the loss of this confidence of 0:

```
import numpy as np

print(-np.log(0))

>>>
__main__:1: RuntimeWarning: divide by zero encountered in log
inf
```

Before we explain this, we need to talk about *log(0)*. From the mathematical point of view, *log(0)* is undefined. We already know the following dependence: if $y=log(x)$, then $e^y=x$. The question of what the resulting y is in $y=log(0)$ is the same as the question of what's the y in $e^y=0$. In simplified terms, the constant e to any power is always a positive number, and there is no y resulting in $e^y=0$. This means the *log(0)* is undefined. We need to be aware of what the *log(0)* is, and "undefined" does not mean that we don't know anything about it. Since *log(0)* is undefined, what's the result for a value very close to *0*? We can calculate the limit of a function. How to exactly calculate it exceeds this book, but the solution is:

$$\lim_{x \to 0^+} log(x) = -\infty$$

We read it as the limit of a natural logarithm of x, with x approaching *0* from a positive (it is

impossible to calculate the natural logarithm of a negative value) equals negative infinity. What this means is that the limit is negative infinity for an infinitely small *x*, where *x* never reaches *0*.

The situation is a bit different in programming languages. We do not have limits here, just a function which, given a parameter, returns some value. The negative natural logarithm of *0*, in Python with NumPy, equals an infinitely big number, rather than undefined, and prints a warning about a division by *0* (which is a result of how this calculation is done). If `-np.log(0)` equals `inf`, is it possible to calculate e to the power of negative infinity with Python?

```
print(np.e**(-np.inf))

>>>
0.0
```

In programming, the fewer things that are undefined, the better. Later on, we'll see similar simplifications, for example when calculating a derivative of the absolute value function, which does not exist for an input of *0* and we'll have to make some decisions to work around this.

Back to the result of `inf` for `-np.log(0)` — as much as that makes sense, since the model would be fully wrong, this will be a problem for us to do further calculations with. Later, with optimization, we will also have a problem calculating gradients, starting with a mean value of all sample-wise losses since a single infinite value in a list will cause the average of that list to also be infinite:

```
import numpy as np

print(np.mean([1, 2, 3, -np.log(0)]))

>>>
__main__:1: RuntimeWarning: divide by zero encountered in log
inf
```

We could add a very small value to the confidence to prevent it from being a zero, for example, *1e-7*:

```
print(-np.log(1e-7))

>>>
16.11809565095832
```

Adding a very small value, one-tenth of a million, to the confidence at its far edge will insignificantly impact the result, but this method yields an additional 2 issues. First, in the case where the confidence value is *1*:

```
print(-np.log(1+1e-7))

>>>
-9.999999505838704e-08
```

When the model is fully correct in a prediction and puts all the confidence in the correct label, loss becomes a negative value instead of being 0. The other problem here is shifting confidence towards *1*, even if by a very small value. To prevent both issues, it's better to clip values from both sides by the same number, *1e-7* in our case. That means that the lowest possible value will become *1e-7* (like in the demonstration we just performed) but the highest possible value, instead of being *1+1e-7*, will become *1-1e-7* (so slightly less than *1*):

```
print(-np.log(1-1e-7))

>>>
1.0000000494736474e-07
```

This will prevent loss from being exactly *0*, making it a very small value instead, but won't make it a negative value and won't bias overall loss towards *1*. Within our code and using numpy, we'll accomplish that using `np.clip()` method:

```
y_pred_clipped = np.clip(y_pred, 1e-7, 1 - 1e-7)
```

This method can perform clipping on an array of values, so we can apply it to the predictions directly and save this as a separate array, which we'll use shortly.

The Categorical Cross-Entropy Loss Class

In the later chapters, we'll be adding more loss functions and some of the operations that we'll be performing are common for all of them. One of these operations is how we calculate the overall loss — no matter which loss function we'll use, the overall loss is always a mean value of all sample losses. Let's create the `Loss` class containing the `calculate` method that will call our loss object's forward method and calculate the mean value of the returned sample losses:

```
# Common loss class
class Loss:

    # Calculates the data and regularization losses
    # given model output and ground truth values
    def calculate(self, output, y):

        # Calculate sample losses
        sample_losses = self.forward(output, y)

        # Calculate mean loss
        data_loss = np.mean(sample_losses)

        # Return loss
        return data_loss
```

In later chapters, we'll add more code to this class, and the reason for it to exist will become more clear. For now, we'll use it for this single purpose.

Let's convert our loss code into a class for convenience down the line:

```
# Cross-entropy loss
class Loss_CategoricalCrossentropy(Loss):

    # Forward pass
    def forward(self, y_pred, y_true):

        # Number of samples in a batch
        samples = len(y_pred)

        # Clip data to prevent division by 0
        # Clip both sides to not drag mean towards any value
        y_pred_clipped = np.clip(y_pred, 1e-7, 1 - 1e-7)

        # Probabilities for target values -
        # only if categorical labels
        if len(y_true.shape) == 1:
            correct_confidences = y_pred_clipped[
                range(samples),
                y_true
            ]

        # Mask values - only for one-hot encoded labels
        elif len(y_true.shape) == 2:
            correct_confidences = np.sum(
                y_pred_clipped*y_true,
                axis=1
            )

        # Losses
        negative_log_likelihoods = -np.log(correct_confidences)
        return negative_log_likelihoods
```

This class inherits the `Loss` class and performs all the error calculations that we derived throughout this chapter and can be used as an object. For example, using the manually-created output and targets:

```
loss_function = Loss_CategoricalCrossentropy()
loss = loss_function.calculate(softmax_outputs, class_targets)
print(loss)

>>>
0.38506088005216804
```

Combining everything up to this point:

```
import numpy as np
import nnfs
from nnfs.datasets import spiral_data

nnfs.init()


# Dense layer
class Layer_Dense:

    # Layer initialization
    def __init__(self, n_inputs, n_neurons):
        # Initialize weights and biases
        self.weights = 0.01 * np.random.randn(n_inputs, n_neurons)
        self.biases = np.zeros((1, n_neurons))

    # Forward pass
    def forward(self, inputs):
        # Calculate output values from inputs, weights and biases
        self.output = np.dot(inputs, self.weights) + self.biases


# ReLU activation
class Activation_ReLU:

    # Forward pass
    def forward(self, inputs):
        # Calculate output values from inputs
        self.output = np.maximum(0, inputs)
```

```
# Softmax activation
class Activation_Softmax:

    # Forward pass
    def forward(self, inputs):

        # Get unnormalized probabilities
        exp_values = np.exp(inputs - np.max(inputs, axis=1,
                                            keepdims=True))
        # Normalize them for each sample
        probabilities = exp_values / np.sum(exp_values, axis=1,
                                            keepdims=True)

        self.output = probabilities


# Common loss class
class Loss:

    # Calculates the data and regularization losses
    # given model output and ground truth values
    def calculate(self, output, y):

        # Calculate sample losses
        sample_losses = self.forward(output, y)

        # Calculate mean loss
        data_loss = np.mean(sample_losses)

        # Return loss
        return data_loss


# Cross-entropy loss
class Loss_CategoricalCrossentropy(Loss):

    # Forward pass
    def forward(self, y_pred, y_true):

        # Number of samples in a batch
        samples = len(y_pred)

        # Clip data to prevent division by 0
        # Clip both sides to not drag mean towards any value
        y_pred_clipped = np.clip(y_pred, 1e-7, 1 - 1e-7)
```

```
        # Probabilities for target values -
        # only if categorical labels
        if len(y_true.shape) == 1:
            correct_confidences = y_pred_clipped[
                range(samples),
                y_true
            ]

        # Mask values - only for one-hot encoded labels
        elif len(y_true.shape) == 2:
            correct_confidences = np.sum(
                y_pred_clipped*y_true,
                axis=1
            )

        # Losses
        negative_log_likelihoods = -np.log(correct_confidences)
        return negative_log_likelihoods



# Create dataset
X, y = spiral_data(samples=100, classes=3)

# Create Dense layer with 2 input features and 3 output values
dense1 = Layer_Dense(2, 3)

# Create ReLU activation (to be used with Dense layer):
activation1 = Activation_ReLU()

# Create second Dense layer with 3 input features (as we take output
# of previous layer here) and 3 output values
dense2 = Layer_Dense(3, 3)

# Create Softmax activation (to be used with Dense layer):
activation2 = Activation_Softmax()

# Create loss function
loss_function = Loss_CategoricalCrossentropy()

# Perform a forward pass of our training data through this layer
dense1.forward(X)

# Perform a forward pass through activation function
# it takes the output of first dense layer here
activation1.forward(dense1.output)
```

```
# Perform a forward pass through second Dense layer
# it takes outputs of activation function of first layer as inputs
dense2.forward(activation1.output)

# Perform a forward pass through activation function
# it takes the output of second dense layer here
activation2.forward(dense2.output)

# Let's see output of the first few samples:
print(activation2.output[:5])

# Perform a forward pass through activation function
# it takes the output of second dense layer here and returns loss
loss = loss_function.calculate(activation2.output, y)

# Print loss value
print('loss:', loss)
```

```
>>>
[[0.33333334 0.33333334 0.33333334]
 [0.33333316 0.3333332  0.33333364]
 [0.33333287 0.3333329  0.33333418]
 [0.3333326  0.33333263 0.33333477]
 [0.33333233 0.3333324  0.33333528]]
loss: 1.0986104
```

Again, we get ~*0.33* values since the model is random, and its average loss is also not great for these data, as we've not yet trained our model on how to correct its errors.

Accuracy Calculation

While loss is a useful metric for optimizing a model, the metric commonly used in practice along with loss is the **accuracy**, which describes how often the largest confidence is the correct class in terms of a fraction. Conveniently, we can reuse existing variable definitions to calculate the accuracy metric. We will use the *argmax* values from the *softmax outputs* and then compare these to the targets. This is as simple as doing (note that we slightly modified the `softmax_outputs` for the purpose of this example):

```
import numpy as np

# Probabilities of 3 samples
softmax_outputs = np.array([[0.7, 0.2, 0.1],
                            [0.5, 0.1, 0.4],
                            [0.02, 0.9, 0.08]])
# Target (ground-truth) labels for 3 samples
class_targets = np.array([0, 1, 1])

# Calculate values along second axis (axis of index 1)
predictions = np.argmax(softmax_outputs, axis=1)
# If targets are one-hot encoded - convert them
if len(class_targets.shape) == 2:
    class_targets = np.argmax(class_targets, axis=1)
# True evaluates to 1; False to 0
accuracy = np.mean(predictions == class_targets)

print('acc:', accuracy)
```

```
>>>
acc: 0.6666666666666666
```

We are also handling one-hot encoded targets by converting them to sparse values using `np.argmax()`.

We can add the following to the end of our full script above to calculate its accuracy:

```
# Calculate accuracy from output of activation2 and targets
# calculate values along first axis
predictions = np.argmax(activation2.output, axis=1)
if len(y.shape) == 2:
    y = np.argmax(y, axis=1)
accuracy = np.mean(predictions == y)

# Print accuracy
print('acc:', accuracy)
```

```
>>>
acc: 0.34
```

Now that you've learned how to perform a forward pass through our network and calculate the metrics to signal if the model is performing poorly, we will embark on optimization in the next chapter!

Supplementary Material: https://nnfs.io/ch5
Chapter code, further resources, and errata for this chapter.

Chapter 6

Introducing Optimization

Now that the neural network is built, able to have data passed through it, and capable of calculating loss, the next step is to determine how to adjust the weights and biases to decrease the loss. Finding an intelligent way to adjust the neurons' input's weights and biases to minimize loss is the main difficulty of neural networks.

The first option one might think of is randomly changing the weights, checking the loss, and repeating this until happy with the lowest loss found. To see this in action, we'll use a simpler dataset than we've been working with so far:

```
import matplotlib.pyplot as plt
import nnfs
from nnfs.datasets import vertical_data

nnfs.init()
```

```
X, y = vertical_data(samples=100, classes=3)
plt.scatter(X[:, 0], X[:, 1], c=y, s=40, cmap='brg')
plt.show()
```

Which looks like:

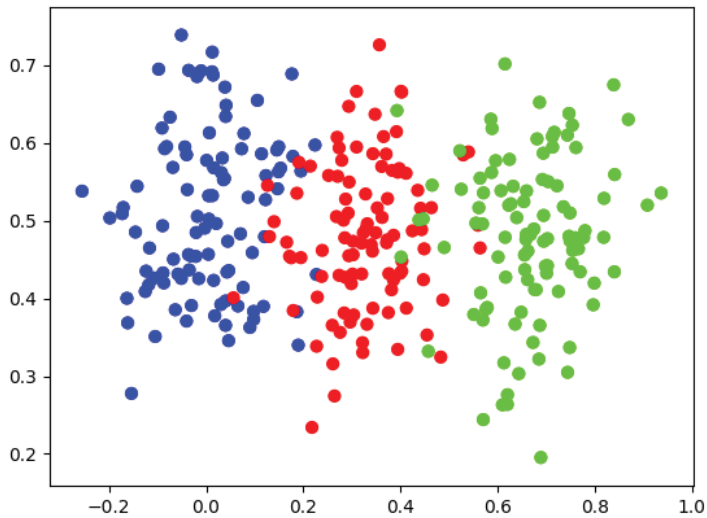


Fig 6.01: "Vertical data" graphed.

Using the previously created code up to this point, we can use this new dataset with a simple neural network:

```
# Create dataset
X, y = vertical_data(samples=100, classes=3)

# Create model
dense1 = Layer_Dense(2, 3)  # first dense layer, 2 inputs
activation1 = Activation_ReLU()
dense2 = Layer_Dense(3, 3)  # second dense layer, 3 inputs, 3 outputs
activation2 = Activation_Softmax()

# Create loss function
loss_function = Loss_CategoricalCrossentropy()
```

Then create some variables to track the best loss and the associated weights and biases:

```
# Helper variables
lowest_loss = 9999999  # some initial value
best_dense1_weights = dense1.weights.copy()
best_dense1_biases = dense1.biases.copy()
best_dense2_weights = dense2.weights.copy()
best_dense2_biases = dense2.biases.copy()
```

We initialized the loss to a large value and will decrease it when a new, lower, loss is found. We are also copying weights and biases (`copy()` ensures a full copy instead of a reference to the object). Now we iterate as many times as desired, pick random values for weights and biases, and save the weights and biases if they generate the lowest-seen loss:

```
for iteration in range(10000):

    # Generate a new set of weights for iteration
    dense1.weights = 0.05 * np.random.randn(2, 3)
    dense1.biases = 0.05 * np.random.randn(1, 3)
    dense2.weights = 0.05 * np.random.randn(3, 3)
    dense2.biases = 0.05 * np.random.randn(1, 3)

    # Perform a forward pass of the training data through this layer
    dense1.forward(X)
    activation1.forward(dense1.output)
    dense2.forward(activation1.output)
    activation2.forward(dense2.output)

    # Perform a forward pass through activation function
    # it takes the output of second dense layer here and returns loss
    loss = loss_function.calculate(activation2.output, y)

    # Calculate accuracy from output of activation2 and targets
    # calculate values along first axis
    predictions = np.argmax(activation2.output, axis=1)
    accuracy = np.mean(predictions == y)

    # If loss is smaller - print and save weights and biases aside
    if loss < lowest_loss:
        print('New set of weights found, iteration:', iteration,
              'loss:', loss, 'acc:', accuracy)
        best_dense1_weights = dense1.weights.copy()
        best_dense1_biases = dense1.biases.copy()
        best_dense2_weights = dense2.weights.copy()
        best_dense2_biases = dense2.biases.copy()
        lowest_loss = loss
```

```
>>>
New set of weights found, iteration: 0 loss: 1.0986564 acc:
0.3333333333333333
New set of weights found, iteration: 3 loss: 1.098138 acc:
0.3333333333333333
New set of weights found, iteration: 117 loss: 1.0980115 acc:
0.3333333333333333
New set of weights found, iteration: 124 loss: 1.0977516 acc: 0.6
New set of weights found, iteration: 165 loss: 1.097571 acc:
0.3333333333333333
New set of weights found, iteration: 552 loss: 1.0974693 acc: 0.34
New set of weights found, iteration: 778 loss: 1.0968257 acc:
0.3333333333333333
New set of weights found, iteration: 4307 loss: 1.0965533 acc:
0.3333333333333333
New set of weights found, iteration: 4615 loss: 1.0964499 acc:
0.3333333333333333
New set of weights found, iteration: 9450 loss: 1.0964295 acc:
0.3333333333333333
```

Loss certainly falls, though not by much. Accuracy did not improve, except for a singular situation where the model randomly found a set of weights yielding better accuracy. Still, with a fairly large loss, this state is not stable. Running an additional 90,000 iterations for 100,000 in total:

```
New set of weights found, iteration: 13361 loss: 1.0963014 acc:
0.3333333333333333
New set of weights found, iteration: 14001 loss: 1.0959858 acc:
0.3333333333333333
New set of weights found, iteration: 24598 loss: 1.0947444 acc:
0.3333333333333333
```

Loss continued to drop, but accuracy did not change. This doesn't appear to be a reliable method for minimizing loss. After running for 1 billion iterations, the following was the best (lowest loss) result:

```
New set of weights found, iteration: 229865000 loss: 1.0911305 acc:
0.3333333333333333
```

Even with this basic dataset, we see that randomly searching for weight and bias combinations will take far too long to be an acceptable method. Another idea might be, instead of setting parameters with randomly-chosen values each iteration, apply a fraction of these values to parameters. With this, weights will be updated from what currently yields us the lowest loss instead of aimlessly randomly. If the adjustment decreases loss, we will make it the new point to adjust from. If loss instead increases due to the adjustment, then we will revert to the previous point. Using similar code from earlier, we will first change from randomly selecting weights and biases to randomly *adjusting* them:

```
# Update weights with some small random values
dense1.weights += 0.05 * np.random.randn(2, 3)
dense1.biases += 0.05 * np.random.randn(1, 3)
dense2.weights += 0.05 * np.random.randn(3, 3)
dense2.biases += 0.05 * np.random.randn(1, 3)
```

Then we will change our ending `if` statement to be:

```
# If loss is smaller - print and save weights and biases aside
if loss < lowest_loss:
    print('New set of weights found, iteration:', iteration,
          'loss:', loss, 'acc:', accuracy)
    best_dense1_weights = dense1.weights.copy()
    best_dense1_biases = dense1.biases.copy()
    best_dense2_weights = dense2.weights.copy()
    best_dense2_biases = dense2.biases.copy()
    lowest_loss = loss
# Revert weights and biases
else:
    dense1.weights = best_dense1_weights.copy()
    dense1.biases = best_dense1_biases.copy()
    dense2.weights = best_dense2_weights.copy()
    dense2.biases = best_dense2_biases.copy()
```

Full code up to this point:

```
# Create dataset
X, y = vertical_data(samples=100, classes=3)

# Create model
dense1 = Layer_Dense(2, 3)  # first dense layer, 2 inputs
activation1 = Activation_ReLU()
dense2 = Layer_Dense(3, 3)  # second dense layer, 3 inputs, 3 outputs
activation2 = Activation_Softmax()

# Create loss function
loss_function = Loss_CategoricalCrossentropy()

# Helper variables
lowest_loss = 9999999  # some initial value
best_dense1_weights = dense1.weights.copy()
best_dense1_biases = dense1.biases.copy()
best_dense2_weights = dense2.weights.copy()
best_dense2_biases = dense2.biases.copy()

for iteration in range(10000):

    # Update weights with some small random values
    dense1.weights += 0.05 * np.random.randn(2, 3)
    dense1.biases += 0.05 * np.random.randn(1, 3)
    dense2.weights += 0.05 * np.random.randn(3, 3)
    dense2.biases += 0.05 * np.random.randn(1, 3)

    # Perform a forward pass of our training data through this layer
    dense1.forward(X)
    activation1.forward(dense1.output)
    dense2.forward(activation1.output)
    activation2.forward(dense2.output)

    # Perform a forward pass through activation function
    # it takes the output of second dense layer here and returns loss
    loss = loss_function.calculate(activation2.output, y)
```

```
    # Calculate accuracy from output of activation2 and targets
    # calculate values along first axis
    predictions = np.argmax(activation2.output, axis=1)
    accuracy = np.mean(predictions == y)

    # If loss is smaller - print and save weights and biases aside
    if loss < lowest_loss:
        print('New set of weights found, iteration:', iteration,
              'loss:', loss, 'acc:', accuracy)
        best_dense1_weights = dense1.weights.copy()
        best_dense1_biases = dense1.biases.copy()
        best_dense2_weights = dense2.weights.copy()
        best_dense2_biases = dense2.biases.copy()
        lowest_loss = loss
    # Revert weights and biases
    else:
        dense1.weights = best_dense1_weights.copy()
        dense1.biases = best_dense1_biases.copy()
        dense2.weights = best_dense2_weights.copy()
        dense2.biases = best_dense2_biases.copy()
```

```
>>>
New set of weights found, iteration: 0 loss: 1.0987684 acc:
0.3333333333333333
...
New set of weights found, iteration: 29 loss: 1.0725244 acc:
0.5266666666666666
New set of weights found, iteration: 30 loss: 1.0724432 acc:
0.3466666666666667
...
New set of weights found, iteration: 48 loss: 1.0303522 acc:
0.6666666666666666
New set of weights found, iteration: 49 loss: 1.0292586 acc:
0.6666666666666666
...
New set of weights found, iteration: 97 loss: 0.9277446 acc:
0.7333333333333333
...
New set of weights found, iteration: 152 loss: 0.73390484 acc:
0.8433333333333334
New set of weights found, iteration: 156 loss: 0.7235515 acc: 0.87
New set of weights found, iteration: 160 loss: 0.7049076 acc:
0.9066666666666666
...
New set of weights found, iteration: 7446 loss: 0.17280102 acc:
0.9333333333333333
New set of weights found, iteration: 9397 loss: 0.17279711 acc: 0.93
```

Loss descended by a decent amount this time, and accuracy raised significantly. Applying a fraction of random values actually lead to a result that we could almost call a solution. If you try 100,000 iterations, you will not progress much further:

```
>>>
...
New set of weights found, iteration: 14206 loss: 0.1727932 acc:
0.9333333333333333
New set of weights found, iteration: 63704 loss: 0.17278232 acc:
0.9333333333333333
```

Let's try this with the previously-seen spiral dataset instead:

```
# Create dataset
X, y = spiral_data(samples=100, classes=3)

>>>
New set of weights found, iteration: 0 loss: 1.1008677 acc:
0.3333333333333333
...
New set of weights found, iteration: 31 loss: 1.0982264 acc:
0.37333333333333335
...
New set of weights found, iteration: 65 loss: 1.0954362 acc:
0.38333333333333336
New set of weights found, iteration: 67 loss: 1.093989 acc:
0.4166666666666667
...
New set of weights found, iteration: 129 loss: 1.0874122 acc:
0.42333333333333334
...
New set of weights found, iteration: 5415 loss: 1.0790575 acc: 0.39
```

This training session ended with almost no progress. Loss decreased slightly and accuracy is barely above the initial value. Later, we'll learn that the most probable reason for this is called a local minimum of loss. The data complexity is also not irrelevant here. It turns out hard problems are hard for a reason, and we need to approach this problem more intelligently.

Supplementary Material: https://nnfs.io/ch6
Chapter code, further resources, and errata for this chapter.

Chapter 7

Derivatives

Randomly changing and searching for optimal weights and biases did not prove fruitful for one main reason: the number of possible combinations of weights and biases is infinite, and we need something smarter than pure luck to achieve any success. Each weight and bias may also have different degrees of influence on the loss — this influence depends on the parameters themselves as well as on the current sample, which is an input to the first layer. These input values are then multiplied by the weights, so the input data affects the neuron's output and affects the impact that the weights make on the loss. The same principle applies to the biases and parameters in the next layers, taking the previous layer's outputs as inputs. This means that the impact on the output values depends on the parameters as well as the samples — which is why we are calculating the loss value per each sample separately. Finally, the function of *how* a weight or bias impacts the overall loss is not necessarily linear. In order to know *how* to adjust weights and biases, we first need to understand their impact on the loss.

One concept to note is that we refer to weights and biases and their impact on the loss function. The loss function doesn't contain weights or biases, though. The input to this function is the output of the model, and the weights and biases of the neurons influence this output. Thus, even though we calculate loss from the model's output, not weights/biases, these weights and biases

directly impact the loss.

In the coming chapters, we will describe exactly how this happens by explaining partial derivatives, gradients, gradient descent, and backpropagation. Basically, we'll calculate how much each singular weight and bias changes the loss value (how much of an impact it has on it) given a sample (as each sample produces a separate output, thus also a separate loss value), and how to change this weight or bias for the loss value to decrease. Remember — our goal here is to decrease loss, and we'll do this by using gradient descent. Gradient, on the other hand, is a result of the calculation of the partial derivatives, and we'll backpropagate it using the chain rule to update all of the weights and biases. Don't worry if that doesn't make much sense yet; we'll explain all of these terms and how to perform these actions in this and the coming chapters.

To understand partial derivatives, we need to start with derivatives, which are a special case of partial derivatives — they are calculated from functions taking single parameters.

The Impact of a Parameter on the Output

Let's start with a simple function and discover what is meant by "impact."
A very simple function $y=2x$, which takes x as an input:

```
def f(x):
    return 2*x
```

Now let's create some code around this to visualize the data — we'll import NumPy and Matplotlib, create an array of 5 input values from 0 to 4, calculate the function output for each of these input values, and plot the result as lines between consecutive points. These points' coordinates are inputs as x and function outputs as y:

```
import matplotlib.pyplot as plt
import numpy as np


def f(x):
    return 2*x
```

```
x = np.array(range(5))
y = f(x)

print(x)
print(y)
```

```
>>>
[0 1 2 3 4]
[0 2 4 6 8]
```

```
plt.plot(x, y)
plt.show()
```

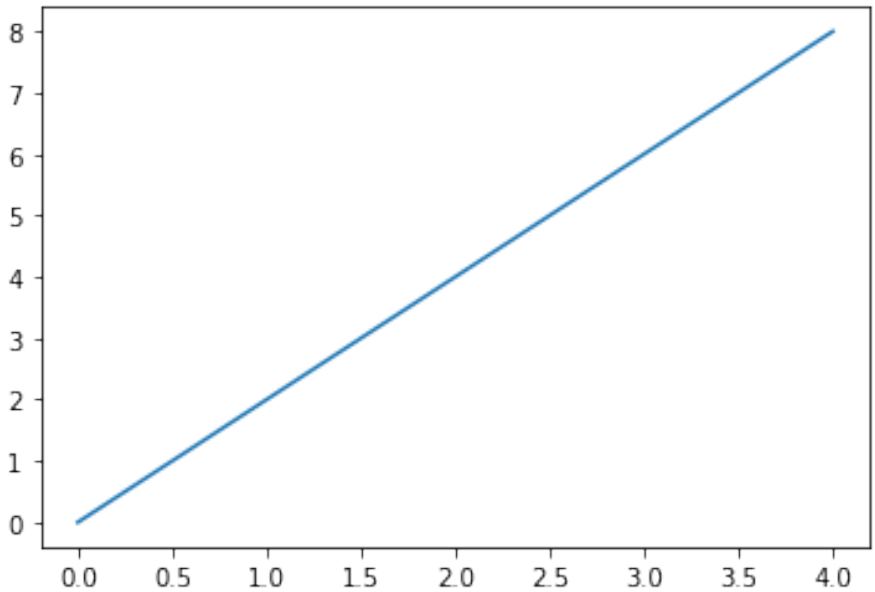


Fig 7.01: Linear function y=2x graphed

The Slope

This looks like an output of the *f(x) = 2x* function, which is a line. How might you define the *impact* that *x* will have on *y*? Some will say, *"y* is double *x.*" Another way to describe the *impact* of a linear function such as this comes from algebra: the **slope**. "Rise over run" might be a phrase you recall from school. The slope of a line is:

$$\frac{Change\ in\ y}{Change\ in\ x} = \frac{\Delta y}{\Delta x}$$

It is change in *y* divided by change in *x,* or, in math — *delta y* divided by *delta x*. What's the slope of *f(x) = 2x* then?
To calculate the slope, first we have to take any two points lying on the function's graph and subtract them to calculate the change. Subtracting the points means to subtract their x and y dimensions respectively. Division of the change in y by the change in x returns the slope:

$$\begin{array}{c} x\ \ y \\ \downarrow\ \downarrow \end{array}$$
$$\begin{cases} p_1 = [0, 0] \\ p_2 = [1, 2] \end{cases}$$

$$\Delta x = p_{2x} - p_{1x} = 1 - 0 = 1$$
$$\Delta y = p_{2y} - p_{1y} = 2 - 0 = 2$$

$$slope = \frac{\Delta y}{\Delta x} = \frac{2}{1} = 2$$

Continuing the code, we keep all values of x in a single-dimensional NumPy array, `x`, and all results in a single-dimensional array, `y`. To perform the same operation, we'll take `x[0]` and `y[0]` for the first point, then `x[1]` and `y[1]` for the second one. Now we can calculate the slope between them:

```
print((y[1]-y[0]) / (x[1]-x[0]))

>>>
2.0
```

It is not surprising that the slope of this line is 2. We could say the measure of the impact that x has on y is 2. We can calculate the slope in the same way for any linear function, including linear functions that aren't as obvious.
What about a nonlinear function like $f(x)=2x^2$?

```
def f(x):
    return 2*x**2
```

This function creates a graph that does not form a straight line:

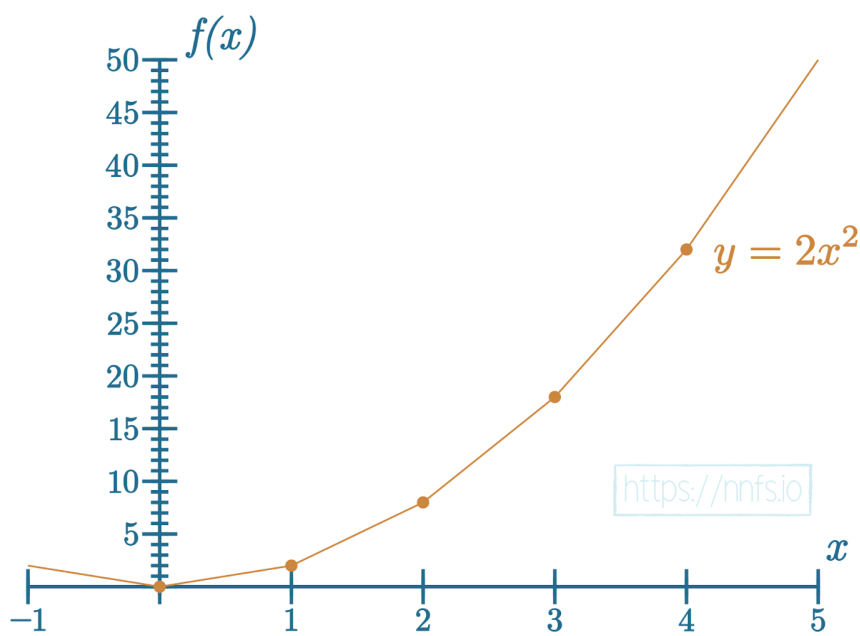


Fig 7.02: Approximation of the parabolic function $y=2x^2$ graphed

Can we measure the slope of this curve? Depending on which 2 points we choose to use, we will measure varying slopes:

```
y = f(x)  # Calculate function outputs for new function

print(x)
print(y)

>>>
[0 1 2 3 4]
[ 0  2  8 18 32]
```

Now for the first pair of points:

```
print((y[1]-y[0]) / (x[1]-x[0]))

>>>
2
```

And for another one:

```
print((y[3]-y[2]) / (x[3]-x[2]))

>>>
10
```

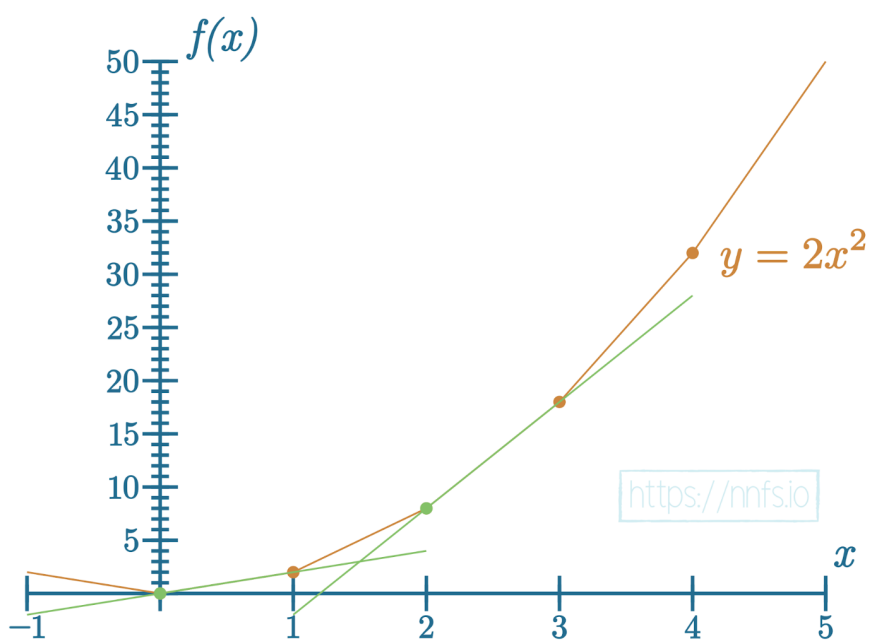


Fig 7.03: Approximation of the parabolic function's example tangents

Anim 7.03: https://nnfs.io/bro

How might we measure the impact that x has on y in this nonlinear function? Calculus proposes that we measure the slope of the **tangent line** at *x* (for a specific input value to the function), giving us the **instantaneous slope** (slope at this point)**,** which is the **derivative**. The **tangent line** is created by drawing a line between two points that are "infinitely close" on a curve, but this curve has to be differentiable at the derivation point. This means that it has to be continuous and smooth (we cannot calculate the slope at something that we could describe as a "sharp corner," since it contains an infinite number of slopes). Then, because this is a curve, there is no single slope. Slope depends on where we measure it. To give an immediate example, we can approximate a derivative of the function at *x* by using this point and another one also taken at x, but with a very small delta added to it, such as *0.0001*. This number is a common choice as it does not introduce too large an error (when estimating the derivative) or cause the whole expression to be numerically unstable (Δx might round to 0 due to floating-point number resolution). This lets us perform the same calculation for the slope as before, but on two points that are very close to each other, resulting in a good approximation of a slope at x:

```
p2_delta = 0.0001

x1 = 1
x2 = x1 + p2_delta  # add delta

y1 = f(x1)  # result at the derivation point
y2 = f(x2)  # result at the other, close point

approximate_derivative = (y2-y1)/(x2-x1)
print(approximate_derivative)
```

```
>>>
4.0001999999987845
```

As we will soon learn, the derivative of $2x^2$ at $x=1$ should be exactly *4*. The difference we see (~4.0002) comes from the method used to compute the tangent. We chose a delta small enough to

approximate the derivative as accurately as possible but large enough to prevent a rounding error. To elaborate, an infinitely small delta value will approximate an accurate derivative; however, the delta value needs to be numerically stable, meaning, our delta can not surpass the limitations of Python's floating-point precision (can't be too small as it might be rounded to *0* and, as we know, dividing by *0* is "illegal"). Our solution is, therefore, restricted between estimating the derivative and remaining numerically stable, thus introducing this small but visible error.

The Numerical Derivative

This method of calculating the derivative is called **numerical differentiation** — calculating the slope of the tangent line using two *infinitely* close points, or as with the code solution — calculating the slope of a tangent line made from two points that were "sufficiently close." We can visualize why we perform this on two close points with the following:

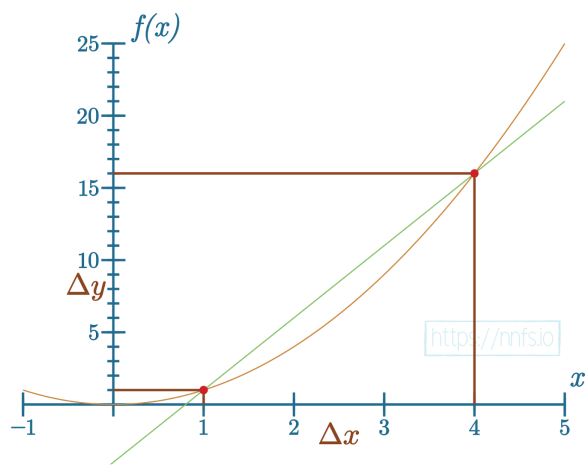


Fig 7.04: Why we want to use 2 points that are sufficiently close — large delta inaccuracy.

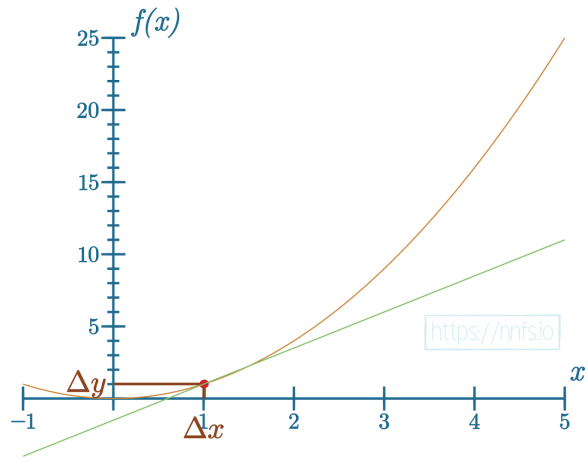


Fig 7.05: Why we want to use 2 points that are sufficiently close — very small delta accuracy.

Anim 7.04-7.05: https://nnfs.io/cat

We can see that the closer these two points are to each other, the more correct the tangent line appears to be.

Continuing with **numerical differentiation**, let us visualize the tangent lines and how they change depending on where we calculate them. To begin, we'll make the graph of this function more granular using Numpy's `arange()`, allowing us to plot with smaller steps. The `np.arange()` function takes in *start, stop,* and *step* parameters, allowing us to take fractions of a step, such as *0.001* at a time:

```
import matplotlib.pyplot as plt
import numpy as np


def f(x):
    return 2*x**2


# np.arange(start, stop, step) to give us smoother line
x = np.arange(0, 5, 0.001)
y = f(x)
```

```
plt.plot(x, y)

plt.show()
```

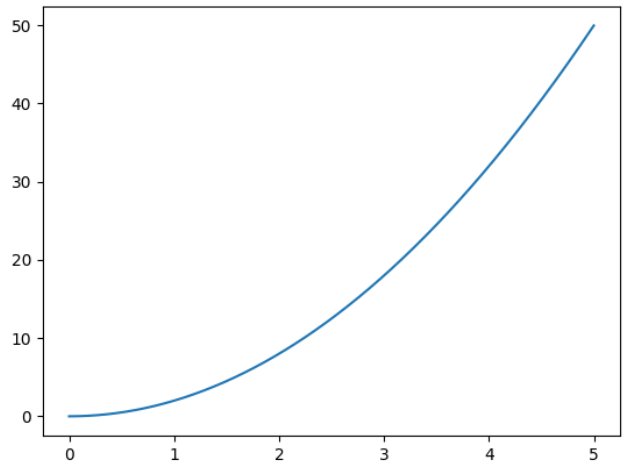


Fig 7.06: Matplotlib output that you should see from graphing $y=2x^2$.

To draw these tangent lines, we will derive the function for the tangent line at a point and plot the tangent on the graph at this point. The function for a straight line is $y = mx+b$. Where m is the slope or the *approximatex_derivative* that we've already calculated. And x is the input which leaves b, or the y-intercept, for us to calculate. The slope remains unchanged, but currently, you can "move" the line up or down using the y-intercept. We already know x and m, but b is still unknown. Let's assume $m=1$ for the purpose of the figure and see what exactly it means:

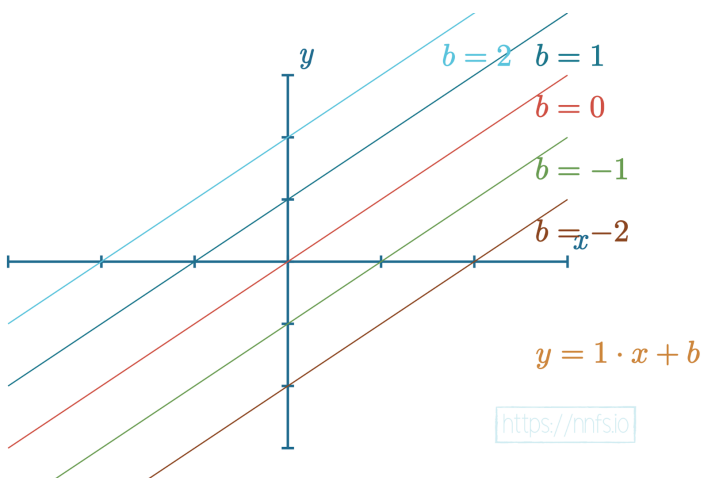


Fig 7.07: Various biases graphed where slope = 1.

Anim 7.07: https://nnfs.io/but

To calculate *b,* the formula is $b = y - mx$:

$$y = mx + b$$

$$y - mx = b$$

$$b = y - mx$$

So far we've used two points — the point that we want to calculate the derivative at and the "close enough" to it point to calculate the approximation of the derivative. Now, given the above equation for *b*, the approximation of the derivative and the same "close enough" point (its *x* and *y* coordinates to be specific), we can substitute them in the equation and get the y-intercept for the tangent line at the derivation point. Using code:

```
b = y2 - approximate_derivative*x2
```

Putting everything together:

```
import matplotlib.pyplot as plt
import numpy as np


def f(x):
    return 2*x**2


# np.arange(start, stop, step) to give us smoother line
x = np.arange(0, 5, 0.001)
y = f(x)
plt.plot(x, y)
```

```
# The point and the "close enough" point
p2_delta = 0.0001
x1 = 2
x2 = x1+p2_delta

y1 = f(x1)
y2 = f(x2)

print((x1, y1), (x2, y2))

# Derivative approximation and y-intercept for the tangent line
approximate_derivative = (y2-y1)/(x2-x1)
b = y2 - approximate_derivative*x2


# We put the tangent line calculation into a function so we can call
# it multiple times for different values of x
# approximate_derivative and b are constant for given function
# thus calculated once above this function
def tangent_line(x):
    return approximate_derivative*x + b


# plotting the tangent line
# +/- 0.9 to draw the tangent line on our graph
# then we calculate the y for given x using the tangent line function
# Matplotlib will draw a line for us through these points
to_plot = [x1-0.9, x1, x1+0.9]
plt.plot(to_plot, [tangent_line(i) for i in to_plot])

print('Approximate derivative for f(x)',
      f'where x = {x1} is {approximate_derivative}')

plt.show()

>>>
(2, 8) (2.0001, 8.000800020000002)
Approximate derivative for f(x) where x = 2 is 8.000199999998785
```

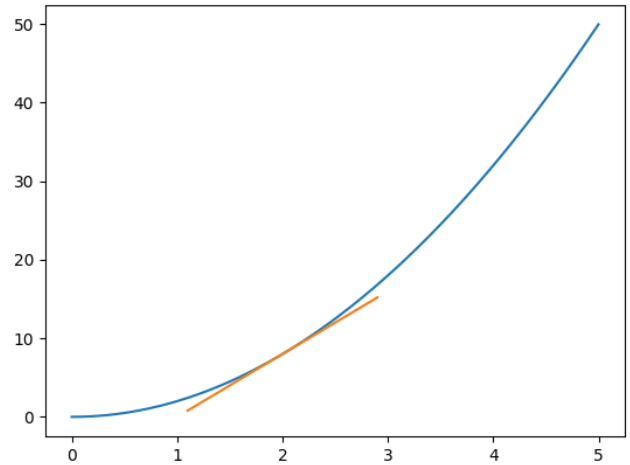


Fig 7.08: Graphed approximate derivative for f(x) where x=2

The orange line is the approximate tangent line at $x=2$ for the function $f(x) = 2x^2$. Why do we care about this? You will soon find that we care only about the *slope* of this tangent line but both visualizing and understanding the **tangent line** are very important. We care about the slope of the tangent line because it informs us about the *impact* that x has on this function at a particular point, referred to as the **instantaneous rate of change**. We will use this concept to determine the effect of a specific weight or bias on the overall loss function given a sample. For now, with different values for x, we can observe resulting impacts on the function. We can continue the previous code to see the tangent line for various inputs (x) - we put a part of the code in a loop over example x values and plot multiple tangent lines:

```
import matplotlib.pyplot as plt
import numpy as np


def f(x):
    return 2*x**2


# np.arange(start, stop, step) to give us a smoother curve
x = np.array(np.arange(0, 5, 0.001))
y = f(x)

plt.plot(x, y)

colors = ['k', 'g', 'r', 'b', 'c']


def approximate_tangent_line(x, approximate_derivative):
    return (approximate_derivative*x) + b
```

```
for i in range(5):
    p2_delta = 0.0001
    x1 = i
    x2 = x1+p2_delta

    y1 = f(x1)
    y2 = f(x2)

    print((x1, y1), (x2, y2))
    approximate_derivative = (y2-y1)/(x2-x1)
    b = y2-(approximate_derivative*x2)

    to_plot = [x1-0.9, x1, x1+0.9]

    plt.scatter(x1, y1, c=colors[i])
    plt.plot([point for point in to_plot],
             [approximate_tangent_line(point, approximate_derivative)
                 for point in to_plot],
             c=colors[i])

    print('Approximate derivative for f(x)',
          f'where x = {x1} is {approximate_derivative}')

plt.show()
```

```
>>>
(0, 0) (0.0001, 2e-08)
Approximate derivative for f(x) where x = 0 is 0.00019999999999999998
(1, 2) (1.0001, 2.00040002)
Approximate derivative for f(x) where x = 1 is 4.0001999999987845
(2, 8) (2.0001, 8.000800020000002)
Approximate derivative for f(x) where x = 2 is 8.000199999998785
(3, 18) (3.0001, 18.001200020000002)
Approximate derivative for f(x) where x = 3 is 12.000199999998785
(4, 32) (4.0001, 32.00160002)
Approximate derivative for f(x) where x = 4 is 16.000200000016548
```

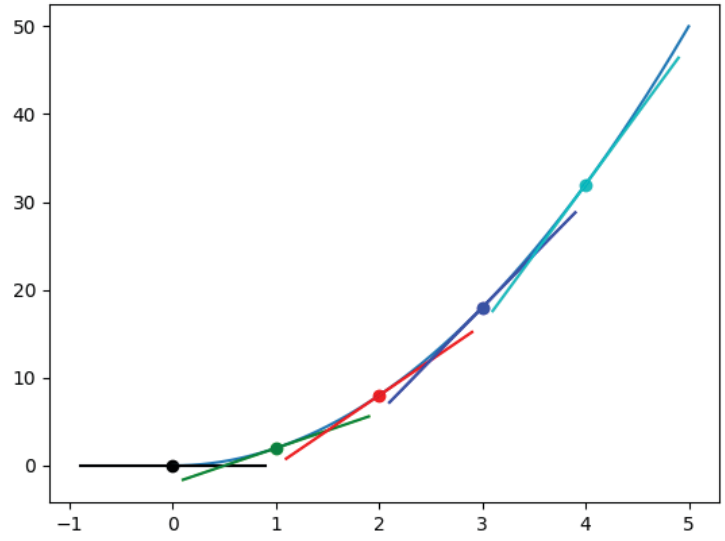


Fig 7.09: Derivative calculated at various points.

For this simple function, $f(x) = 2x^2$, we didn't pay a high penalty by approximating the derivative (i.e., the slope of the tangent line) like this, and received a value that was close enough for our needs.

The problem is that the *actual* function employed in our neural network is not so simple. The loss function contains all of the layers, weights, and biases — it's an absolutely massive function operating in multiple dimensions! Calculating derivatives using **numerical differentiation** requires multiple forward passes for a single parameter update (we'll talk about parameter updates in chapter 10). We need to perform the forward pass as a reference, then update a single parameter by the delta value and perform the forward pass through our model again to see the change of the loss value. Next, we need to calculate the **derivative** and revert the parameter change that we made for this calculation. We have to repeat this for every weight and bias and for every sample, which will be very time-consuming. We can also think of this method as brute-forcing the derivative calculations. To reiterate, as we quickly covered many terms, the **derivative** is the **slope** of the **tangent line** for a function that takes a single parameter as an input. We'll use this ability to calculate the slopes of the loss function at each of the weight and bias points — this brings us to the multivariate function, which is a function that takes multiple parameters and is a topic for the next chapter — the partial derivative.

The Analytical Derivative

Now that we have a better idea of what a derivative *is*, how to calculate the numerical (also called universal) derivative, and why it's not a good approach for us, we can move on to the **Analytical Derivative**, the actual solution to the derivative that we'll implement in our code.

In mathematics, there are two general ways to solve problems: **numerical** and **analytical** methods. Numerical solution methods involve coming up with a number to find a solution, like the above approach with `approximate_derivative`. The numerical solution is also an approximation. On the other hand, the analytical method offers the exact and much quicker, in terms of calculation, solution. However, identifying the analytical solution for the derivative of a given function, as we'll quickly learn, will vary in complexity, whereas the numerical approach never gets more complicated — it's always calling the method twice with two inputs to calculate the approximate derivative at a point. Some analytical solutions are quite obvious, some can be calculated with simple rules, and some complex functions can be broken down into simpler parts and calculated using the so-called **chain rule**. We can leverage already-proven derivative solutions for certain functions, and others — like our loss function — can be solved with combinations of the above.

To compute the derivative of functions using the analytical method, we can split them into simple, elemental functions, finding the derivatives of those and then applying the **chain rule**, which we will explain soon, to get the full derivative. To start building an intuition, let's start with simple functions and their respective derivatives.

The derivative of a simple constant function:

$$f(x) = 1 \quad \rightarrow \quad \frac{d}{dx} f(x) = \frac{d}{dx} 1 = 0$$

$$f(x) = 1$$

$$f'(x) = \frac{d}{dx} 1$$

$$f'(x) = 0$$

https://nnfs.io

Fig 7.10: Derivative of a constant function — calculation steps.

Anim 7.10: https://nnfs.io/cow

When calculating the derivative of a function, recall that the derivative can be interpreted as a slope. In this example, the result of this function is a horizontal line as the output value for any x is 1:

By looking at it, it becomes evident that the derivative equals 0 since there's no change from one value of x to any other value of x (i.e., there's no slope).

So far, we are calculating derivatives of the functions by taking a single parameter, *x* in our case, in each example. This changes with partial derivatives since they take functions with multiple parameters, and we'll be calculating the derivative with respect to only one of them at a time. For now, with derivatives, it's always with respect to a single parameter. To denote the derivative, we can use prime notation, where, for the function $f(x)$, we add a prime (') like $f'(x)$. For our example, $f(x) = 1$, the derivative $f'(x) = 0$. Another notation we can use is called the Leibniz's notation — the dependence on the prime notation and multiple ways of writing the derivative with the Leibniz's notation is as follows:

$$f'(x) = \frac{d}{dx}f(x) = \frac{df}{dx}(x) = \frac{df(x)}{dx}$$

Each of these notations has the same meaning — the derivative of a function (with respect to *x*).

In the following examples, we use both notations, since sometimes it's convenient to use one notation or another. We can also use both of them in a single equation.

In summary: the derivative of a constant function equals *0*:

$$f(x) = 1 \quad \rightarrow \quad f'(x) = 0$$

The derivative of a linear function:

$$f(x) = x \quad \rightarrow \quad \frac{d}{dx}f(x) = \frac{d}{dx}x = \frac{d}{dx}x^1 = 1 \cdot x^{1-1} = 1 \cdot x^0 = 1 \cdot 1 = 1$$

$$\begin{aligned}
f(x) &= x \\
f'(x) &= \frac{d}{dx}x \\
f'(x) &= \frac{d}{dx}x^1 \\
f'(x) &= \quad x^{1-1} \\
f'(x) &= 1 \cdot x^0 \\
f'(x) &= 1 \cdot 1 \\
f'(x) &= 1
\end{aligned}$$

https://nnfs.io

Fig 7.11: Derivative of a linear function — calculation steps.

Anim 7.11: https://nnfs.io/tob

In this case, the derivative is 1, and the intuition behind this is that for every change of x, y changes by the same amount, so y changes one times the x.

The derivative of the linear function equals *1* (but not in every case, which we'll explain next):

$$f(x) = x \quad \rightarrow \quad f'(x) = 1$$

What if we try 2x, which is also a linear function?

$$f(x) = 2x \quad \rightarrow \quad \frac{d}{dx}f(x) = \frac{d}{dx}2x = 2 \cdot \frac{d}{dx}x = 2 \cdot 1x^{1-1} = 2 \cdot 1x^0 = 2 \cdot 1 = 2$$

$$f(x) = 2x$$

$$f'(x) = \frac{d}{dx}2x$$

$$f'(x) = 2 \cdot \frac{d}{dx}x^1$$

$$f'(x) = 2 \cdot x^{1-1}$$

$$f'(x) = 2 \cdot 1x^0$$

$$f'(x) = 2 \cdot 1 \cdot 1$$

$$f'(x) = 2$$

https://nnfs.io

Fig 7.12: Derivative of another linear function — calculation steps.

Anim 7.12: https://nnfs.io/pop

When calculating the derivative, we can take any constant that function is multiplied by and move it outside of the derivative — in this case it's 2 multiplied by the derivative of x. Since we already determined that the derivative of *f(x) = x* was *1*, we now multiply it by *2* to give us the result.

The derivative of a linear function equals the slope, *m*. In this case *m = 2*:

$$f(x) = 2x \quad \rightarrow \quad f'(x) = 2$$

If you associate this with numerical differentiation, you're absolutely right — we already concluded that the derivative of a linear function equals its slope:

$$f(x) = mx \quad \rightarrow \quad f'(x) = m$$

m, in this case, is a constant, no different than the value *2,* as it's not a parameter — every non-parameter to the function can't change its value; thus, we consider it to be a constant. We have just found a simpler way to calculate the derivative of a linear function and also generalized it for the equations of different slopes, *m*. It's also an exact derivative, not an approximation, as with the numerical differentiation.

What happens when we introduce exponents to the function?

$$f(x) = 3x^2 \quad \rightarrow \quad \frac{d}{dx}f(x) = \frac{d}{dx}3x^2 = 3 \cdot \frac{d}{dx}x^2 = 3 \cdot 2x^{2-1} = 3 \cdot 2x^1 = 6x$$

$$f(x) = 3x^2$$

$$f'(x) = \frac{d}{dx}3x^2$$

$$f'(x) = 3 \cdot x^{2-1}$$

$$f'(x) = 3 \cdot 2x^1$$

$$f'(x) = 6x$$

https://nnfs.io

Fig 7.13: Derivative of quadratic function — calculation steps.

Anim 7.13: https://nnfs.io/rok

First, we are applying the rule of a constant — we can move the coefficient (the value that multiplies the other value) outside of the derivative. The rule for handling exponents is as follows: take the exponent, in this case a *2*, and use it as a coefficient for the derived value, then, subtract 1 from the exponent, as seen here: *2 - 1 = 1*.

If $f(x) = 3x^2$ then $f'(x) = 3 \cdot 2x^1$ or simply $6x$. This means the slope of the tangent line, at any point, x, for this quadratic function, will be $6x$. As discussed with the numerical solution of the quadratic function differentiation, the derivative of a quadratic function depends on the x and in this case it equals $6x$:

$$f(x) = 3x^2 \quad \rightarrow \quad f'(x) = 6x$$

A commonly used operator in functions is addition, how do we calculate the derivative in this case?

$$f(x) = 3x^2 + 5x \quad \rightarrow \quad \frac{d}{dx}f(x) = \frac{d}{dx}[3x^2 + 5x] =$$

$$= \frac{d}{dx}3x^2 + \frac{d}{dx}5x^1 =$$

$$= 3 \cdot \frac{d}{dx}x^2 + 5 \cdot \frac{d}{dx}x^1 =$$

$$= 3 \cdot 2x^{2-1} + 5 \cdot 1x^{1-1} =$$

$$= 3 \cdot 2x^1 + 5 \cdot x^0 =$$

$$= 6x + 5$$

$$
\begin{aligned}
f(x) &= 3x^2 + 5x \\
f'(x) &= \frac{d}{dx}[3x^2 + 5x] \\
f'(x) &= \frac{d}{dx}3x^2 + \frac{d}{dx}5x \\
f'(x) &= 3 \cdot x^{2-1} + 5 \cdot x^{1-1} \\
f'(x) &= 3 \cdot 2x^1 + 5 \cdot 1x^0 \\
f'(x) &= 6x + 5
\end{aligned}
$$

https://nnfs.io

Fig 7.14: Derivative of quadratic function with addition — calculation steps.

Anim 7.14: https://nnfs.io/mob

The derivative of a sum operation is the sum of derivatives, so we can split the derivative of a more complex sum operation into a sum of the derivatives of each term of the equation and solve the rest of the derivative using methods we already know.

The derivative of a sum of functions equals their derivatives:

$$\frac{d}{dx}[f(x) + g(x)] = \frac{d}{dx}f(x) + \frac{d}{dx}g(x) = f'(x) + g'(x)$$

In this case, we've shown the rule using both notations.

Let's try a couple more examples:

$$
\begin{aligned}
f(x) = 5x^5 + 4x^3 - 5 \quad \rightarrow \quad \frac{d}{dx}f(x) &= \frac{d}{dx}[5x^5 + 4x^3 - 5] = \\
&= \frac{d}{dx}5x^5 + \frac{d}{dx}4x^3 - \frac{d}{dx}5 = \\
&= 5 \cdot \frac{d}{dx}x^5 + 4 \cdot \frac{d}{dx}x^3 - \frac{d}{dx}5 = \\
&= 5 \cdot 5x^{5-1} + 4 \cdot 3x^{3-1} - 0 = \\
&= 5 \cdot 5x^4 + 4 \cdot 3x^2 = \\
&= 25x^4 + 12x^2
\end{aligned}
$$

$$
\begin{aligned}
f(x) &= 5x^5 + 4x^3 - 5 \\
f'(x) &= \frac{d}{dx}[5x^5 + 4x^3 - 5] \\
f'(x) &= \frac{d}{dx}5x^5 + \frac{d}{dx}4x^3 - \frac{d}{dx}5 \\
f'(x) &= 5 \cdot x^{5-1} + 4 \cdot x^{3-1} - 0 \\
f'(x) &= 5 \cdot 5x^4 + 4 \cdot 3x^2 \\
f'(x) &= 25x^4 + 12x^2
\end{aligned}
$$

https://nnfs.io

Fig 7.15: Analytical derivative of multi-dimensional function example — calculation steps.

Anim 7.15: https://nnfs.io/tom

The derivative of a constant 5 equals 0, as we already discussed at the beginning of this chapter. We also have to apply the other rules that we've learned so far to perform this calculation.

$$
\begin{aligned}
f(x) = x^3 + 2x^2 - 5x + 7 \quad \rightarrow \quad \frac{d}{dx}f(x) &= \frac{d}{dx}[x^3 + 2x^2 - 5x + 7] = \\
&= \frac{d}{dx}x^3 + \frac{d}{dx}2x^2 - \frac{d}{dx}5x + \frac{d}{dx}7 = \\
&= \frac{d}{dx}x^3 + 2 \cdot \frac{d}{dx}x^2 - 5 \cdot \frac{d}{dx}x + \frac{d}{dx}7 = \\
&= 3x^{3-1} + 2 \cdot 2x^{2-1} - 5 \cdot 1x^{1-1} + 0 = \\
&= 3x^2 + 2 \cdot 2x^1 - 5 \cdot 1 + 0 = \\
&= 3x^2 + 4x - 5
\end{aligned}
$$

$$
\begin{aligned}
f(x) &= x^3 + 2x^2 - 5x + 7 \\
f'(x) &= \frac{d}{dx}[x^3 + 2x^2 - 5x + 7] \\
f'(x) &= \frac{d}{dx}x^3 + \frac{d}{dx}2x^2 - \frac{d}{dx}5x + \frac{d}{dx}7 \\
f'(x) &= x^{3-1} + 2 \cdot x^{2-1} - 5 \cdot x^{1-1} + 0 \\
f'(x) &= 3x^2 + 2 \cdot 2x^1 - 5 \cdot 1x^0 \\
f'(x) &= 3x^2 + 4x - 5
\end{aligned}
$$

https://nnfs.io

Fig 7.16: Analytical derivative of another multi-dimensional function example — calculation steps.

Anim 7.16: https://nnfs.io/sun

This looks relatively straight-forward so far, but, with neural networks, we'll work with functions that take multiple parameters as inputs, so we're going to calculate the partial derivatives as well.

Summary

Let's summarize some of the solutions and rules that we have learned in this chapter.

Solutions:

The derivative of a constant equals 0 (m is a constant in this case, as it's not a parameter that we are deriving with respect to, which is *x* in this example):

$$\frac{d}{dx}1 = 0$$

$$\frac{d}{dx}m = 0$$

The derivative of *x* equals 1:

$$\frac{d}{dx}x = 1$$

The derivative of a linear function equals its slope:

$$\frac{d}{dx}mx + b = m$$

Rules:

The derivative of a constant multiple of the function equals the constant multiple of the function's

derivative:

$$\frac{d}{dx}[k \cdot f(x)] = k \cdot \frac{d}{dx}f(x)$$

The derivative of a sum of functions equals the sum of their derivatives:

$$\frac{d}{dx}[f(x) + g(x)] = \frac{d}{dx}f(x) + \frac{d}{dx}g(x) = f'(x) + g'(x)$$

The same concept applies to subtraction:

$$\frac{d}{dx}[f(x) - g(x)] = \frac{d}{dx}f(x) - \frac{d}{dx}g(x) = f'(x) - g'(x)$$

The derivative of an exponentiation:

$$\frac{d}{dx}x^n = n \cdot x^{n-1}$$

We used the value x instead of the whole function f(x) here since the derivative of an entire function is calculated a bit differently. We'll explain this concept along with the chain rule in the next chapter.

Since we've already learned what derivatives are and how to calculate them analytically, which we'll later implement in code, we can go a step further and cover partial derivatives in the next chapter.

Supplementary Material: https://nnfs.io/ch7
Chapter code, further resources, and errata for this chapter.

Chapter 8

Gradients, Partial Derivatives, and the Chain Rule

Two of the last pieces of the puzzle, before we continue coding our neural network, are the related concepts of **gradients** and **partial derivatives**. The derivatives that we've solved so far have been cases where there is only one independent variable in the function — that is, the result depended solely on, in our case, x. However, our neural network consists, for example, of neurons, which have multiple inputs. Each input gets multiplied by the corresponding weight (a function of 2 parameters), and they get summed with the bias (a function of as many parameters as there are inputs, plus one for a bias). As we'll explain soon in detail, to learn the impact of all of the inputs, weights, and biases to the neuron output and at the end of the loss function, we need to calculate the derivative of each operation performed during the forward pass in the neuron and the whole model. To do that and get answers, we'll need to use the **chain rule**, which we'll explain soon in this chapter.

The Partial Derivative

The **partial derivative** measures how much impact a single input has on a function's output. The method for calculating a partial derivative is the same as for derivatives explained in the previous chapter; we simply have to repeat this process for each of the independent inputs.

Each of the function's inputs has some impact on this function's output, even if the impact is 0. We need to know these impacts; this means that we have to calculate the derivative with respect to each input separately to learn about each of them. That's why we call these partial derivatives with respect to given input — we are calculating a partial of the derivative, related to a singular input. The partial derivative is a single equation, and the full multivariate function's derivative consists of a set of equations called the **gradient**. In other words, the **gradient** is a vector of the size of inputs containing partial derivative solutions with respect to each of the inputs. We'll get back to gradients shortly.

To denote the partial derivative, we'll be using Euler's notation. It's very similar to Leibniz's notation, as we only need to replace the differential operator *d* with ∂. While the *d* operator might be used to denote the differentiation of a multivariate function, its meaning is a bit different — it can mean the rate of the function's change in relation to the given input, but when other inputs might change as well, and it is used mostly in physics. We are interested in the partial derivatives, a situation where we try to find the impact of the given input to the output while treating all of the other inputs as constants. We are interested in the impact of singular inputs since our goal, in the model, is to update parameters. The ∂ operator means explicitly that — the partial derivative:

$$f(x, y, z) \quad \rightarrow \quad \frac{\partial}{\partial x} f(x, y, z), \frac{\partial}{\partial y} f(x, y, z), \frac{\partial}{\partial z} f(x, y, z)$$

The Partial Derivative of a Sum

Calculating the partial derivative with respect to a given input means to calculate it like the regular derivative of one input, just while treating other inputs as constants. For example:

$$f(x,y) = x + y \quad \rightarrow \quad \frac{\partial}{\partial x} f(x,y) = \frac{\partial}{\partial x}[x+y] = \frac{\partial}{\partial x}x + \frac{\partial}{\partial x}y = 1 + 0 = 1$$

$$\frac{\partial}{\partial y} f(x,y) = \frac{\partial}{\partial y}[x+y] = \frac{\partial}{\partial y}x + \frac{\partial}{\partial y}y = 0 + 1 = 1$$

First, we applied the sum rule — the derivative of a sum is the sum of derivatives. Then, we already know that the derivative of *x* with respect to *x* equals *1*. The new thing is the derivative of *y* with respect to *x*. As we mentioned, *y* is treated as a constant, as it does not change when we are deriving with respect to *x*, and the derivative of a constant equals *0*. In the second case, we derived with respect to *y*, thus treating *x* as constant. Put another way, regardless of the value of y in this example, the slope of *x* does not depend on *y*. This will not always be the case, though, as we will soon see.

Let's try another example:

$$f(x,y) = 2x + 3y^2 \quad \rightarrow \quad \frac{\partial}{\partial x} f(x,y) = \frac{\partial}{\partial x}[2x + 3y^2] = \frac{\partial}{\partial x}2x + \frac{\partial}{\partial x}3y^2 =$$

$$= 2 \cdot \frac{\partial}{\partial x}x + 3 \cdot \frac{\partial}{\partial x}y^2 = 2 \cdot 1 + 3 \cdot 0 = 2$$

$$\frac{\partial}{\partial y} f(x,y) = \frac{\partial}{\partial y}[2x + 3y^2] = \frac{\partial}{\partial y}2x + \frac{\partial}{\partial y}3y^2 =$$

$$= 2 \cdot \frac{\partial}{\partial y}x + 3 \cdot \frac{\partial}{\partial y}y^2 = 2 \cdot 0 + 3 \cdot 2y^1 = 6y$$

In this example, we also applied the sum rule first, then moved constants to the outside of the derivatives and calculated what remained with respect to *x* and *y* individually. The only difference to the non-multivariate derivatives from the previous chapter is the "partial" part, which means we

are deriving with respect to each of the variables separately. Other than that, there is nothing new here.

Let's try something seemingly more complicated:

$$f(x,y) = 3x^3 - y^2 + 5x + 2 \quad \rightarrow$$

$$\frac{\partial}{\partial x} f(x,y) = \frac{\partial}{\partial x}[3x^3 - y^2 + 5x + 2] = \frac{\partial}{\partial x}3x^3 - \frac{\partial}{\partial x}y^2 + \frac{\partial}{\partial x}5x + \frac{\partial}{\partial x}2 =$$

$$= 3\cdot\frac{\partial}{\partial x}x^3 - \frac{\partial}{\partial x}y^2 + 5\cdot\frac{\partial}{\partial x}x + \frac{\partial}{\partial x}2 = 3\cdot 3x^2 - 0 + 5\cdot 1 + 0 = 9x^2 + 5$$

$$\frac{\partial}{\partial y} f(x,y) = \frac{\partial}{\partial y}[3x^3 - y^2 + 5x + 2] = \frac{\partial}{\partial y}3x^3 - \frac{\partial}{\partial y}y^2 + \frac{\partial}{\partial y}5x + \frac{\partial}{\partial y}2 =$$

$$= 3\cdot\frac{\partial}{\partial y}x^3 - \frac{\partial}{\partial y}y^2 + 5\cdot\frac{\partial}{\partial y}x + \frac{\partial}{\partial y}2 = 3\cdot 0 - 2y^1 + 5\cdot 0 + 0 = -2y$$

Pretty straight-forward — we're constantly applying the same rules over and over again, and we did not add any new calculation or rules in this example.

The Partial Derivative of Multiplication

Before we move on, let's introduce the partial derivative of multiplication operation:

$$f(x,y) = x \cdot y \quad \rightarrow \quad \frac{\partial}{\partial x} f(x,y) = \frac{\partial}{\partial x}[x \cdot y] = y\frac{\partial}{\partial x}x = y \cdot 1 = y$$

$$\frac{\partial}{\partial y} f(x,y) = \frac{\partial}{\partial y}[x \cdot y] = x\frac{\partial}{\partial y}y = x \cdot 1 = x$$

We have already mentioned that we need to treat the other independent variables as constants, and we also have learned that we can move constants to the outside of the derivative. That's exactly how we solve the calculation of the partial derivative of multiplication — we treat other variables as constants, like numbers, and move them outside of the derivative. It turns out that when we derive with respect to *x*, *y* is treated as a constant, and the result equals *y* multiplied by the derivative of *x* with respect to *x,* which is *1*. The whole derivative then results with *y*. The intuition behind this example is when calculating the partial derivative with respect to *x*, every change of *x* by *1* changes the function's output by *y*. For example, if *y=3* and *x=1*, the result is *1·3=3*. When we change *x* by *1* so *y=3* and *x=2*, the result is *2·3=6*. We changed *x* by *1* and the result changed by *3*, by the *y*. That's what the partial derivative of this function with respect to *x* tells us.

Let's introduce a third input variable and add multiplication of variables for another example:

$$f(x,y,z) = 3x^3z - y^2 + 5z + 2yz \quad \rightarrow$$

$$\begin{aligned}
\frac{\partial}{\partial x} f(x,y,z) &= \frac{\partial}{\partial x}[3x^3z - y^2 + 5z + 2yz] = \\
&= \frac{\partial}{\partial x}3x^3z - \frac{\partial}{\partial x}y^2 + \frac{\partial}{\partial x}5z + \frac{\partial}{\partial x}2yz = \\
&= 3z \cdot \frac{\partial}{\partial x}x^3 - \frac{\partial}{\partial x}y^2 + 5 \cdot \frac{\partial}{\partial x}z + 2 \cdot \frac{\partial}{\partial x}yz = \\
&= 3z \cdot 3x^2 - 0 + 5 \cdot 0 + 2 \cdot 0 = 9x^2z
\end{aligned}$$

$$f(x, y, z) = 3x^3z - y^2 + 5z + 2yz \quad \rightarrow$$

$$\begin{aligned}
\frac{\partial}{\partial y} f(x, y, z) &= \frac{\partial}{\partial y}[3x^3z - y^2 + 5z + 2yz] = \\
&= \frac{\partial}{\partial y}3x^3z - \frac{\partial}{\partial y}y^2 + \frac{\partial}{\partial y}5z + \frac{\partial}{\partial y}2yz = \\
&= 3 \cdot \frac{\partial}{\partial y}x^3z - \frac{\partial}{\partial y}y^2 + 5 \cdot \frac{\partial}{\partial y}z + 2z \cdot \frac{\partial}{\partial y}y = \\
&= 3 \cdot 0 - 2y + 5 \cdot 0 + 2z \cdot 1 = -2y + 2z
\end{aligned}$$

$$f(x, y, z) = 3x^3z - y^2 + 5z + 2yz \quad \rightarrow$$

$$\begin{aligned}
\frac{\partial}{\partial z} f(x, y, z) &= \frac{\partial}{\partial z}[3x^3z - y^2 + 5z + 2yz] = \\
&= \frac{\partial}{\partial z}3x^3z - \frac{\partial}{\partial z}y^2 + \frac{\partial}{\partial z}5z + \frac{\partial}{\partial z}2yz = \\
&= 3x^3 \cdot \frac{\partial}{\partial z}z - \frac{\partial}{\partial z}y^2 + 5 \cdot \frac{\partial}{\partial z}z + 2y \cdot \frac{\partial}{\partial z}z = \\
&= 3x^3 \cdot 1 - 0 + 5 \cdot 1 + 2y \cdot 1 = 3x^3 + 5 + 2y
\end{aligned}$$

The only new operation here is, as mentioned, moving variables other than the one that we derive with respect to, outside of the derivative. The results in this example appear more complicated, but only because of the existence of other variables in them — variables that are treated as constants during derivation. Equations of the derivatives are longer, but not necessarily more complicated.

The reason to learn about partial derivatives is we'll be calculating the partial derivatives of multivariate functions soon, an example of which is the neuron. From the code perspective and the *Dense* layer class, more specifically, the *forward* method of this class, we're passing in a single variable — the input array, containing either a batch of samples or outputs from the previous layer. From the math perspective, each value of this single variable (an array) is a separate input — it contains as many inputs as we have data in the input array. For example, if we pass a vector of *4* values to the neuron, it's a singular variable in the code, but *4* separate inputs in the equation. This forms a function that takes multiple inputs. To learn about the impact that each input makes to the function's output, we'll need to calculate the partial derivative of this function with respect to each of its inputs, which we'll explain in detail in the next chapter.

The Partial Derivative of *Max*

Derivatives and partial derivatives are not limited to addition and multiplication operations, or constants. We need to derive them for the other functions that we used in the forward pass, one of which is the derivative of the *max()* function:

$$f(x, y) = max(x, y) \quad \rightarrow \quad \frac{\partial}{\partial x} f(x, y) = \frac{\partial}{\partial x} max(x, y) = 1(x > y)$$

The max function returns the greatest input. We know that the derivative of *x* with respect to x equals *1,* so the derivative of this function with respect to *x* equals 1 if *x* is greater than *y*, since the function will return *x*. In the other case, where *y* is greater than *x* and will get returned instead, the derivative of *max()* with respect to *x* equals 0 — we treat *y* as a constant, and the derivative of *y* with respect to *x* equals 0. We can denote that as *1(x > y)*, which means *1* if the condition is met, and *0* otherwise. We could also calculate the partial derivative of *max()* with respect to *y*, but we won't need it anywhere in this book.

One special case for the derivative of the *max()* function is when we have only one variable parameter, and the other parameter is always constant at *0*. This means that we want whichever is bigger in return — *0* or the input value, effectively clipping the input value at *0* from the positive side. Handling this is going to be useful when we calculate the derivative of the **ReLU** activation function since that activation function is defined as *max(x, 0)*:

$$f(x) = max(x, 0) \quad \rightarrow \quad \frac{d}{dx} f(x) = \frac{d}{dx} max(x, 0) = 1(x > 0)$$

Notice that since this function takes a single parameter, we used the *d* operator instead of the ∂ to calculate the non-partial derivative. In this case, the derivative is *1* when *x* is greater than *0,* otherwise, it's *0*.

The Gradient

As we mentioned at the beginning of this chapter, the gradient is a **vector** composed of all of the partial derivatives of a function, calculated with respect to each input variable.

Let's return to one of the partial derivatives of the sum operation that we calculated earlier:

$$f(x, y, z) = 3x^3z - y^2 + 5z + 2yz \quad \rightarrow$$

$$\frac{\partial}{\partial x} f(x, y, z) = 9x^2z$$

$$\frac{\partial}{\partial y} f(x, y, z) = -2y + 2z$$

$$\frac{\partial}{\partial z} f(x, y, z) = 3x^3 + 5 + 2y$$

If we calculate all of the partial derivatives, we can form a gradient of the function. Using different notations, it looks as follows:

$$\nabla f(x, y, z) = \begin{bmatrix} \frac{\partial}{\partial x} f(x, y, z) \\ \frac{\partial}{\partial y} f(x, y, z) \\ \frac{\partial}{\partial z} f(x, y, z) \end{bmatrix} = \begin{bmatrix} \frac{\partial}{\partial x} \\ \frac{\partial}{\partial y} \\ \frac{\partial}{\partial z} \end{bmatrix} f(x, y, z) = \begin{bmatrix} 9x^2z \\ -2y + 2z \\ 3x^3 + 5 + 2y \end{bmatrix}$$

That's all we have to know abxout the **gradient** - it's a vector of all of the possible partial derivatives of the function, and we denote it using the $\bigtriangledown$ — nabla symbol that looks like an inverted delta symbol.

We'll be using **derivatives** of single-parameter functions and **gradients** of multivariate functions to perform **gradient descent** using the **chain rule,** or, in other words, to perform the **backward pass**, which is a part of the model training. How exactly we'll do that is the subject of the next chapter.

The Chain Rule

During the forward pass, we're passing the data through the neurons, then through the activation function, then through the neurons in the next layer, then through another activation function, and so on. We're calling a function with an input parameter, taking an output, and using that output as an input to another function. For this simple example, let's take 2 functions: *f* and *g*:

$$z = f(x)$$
$$y = g(z)$$

x is the input data, z is an output of the function f, but also an input for the function g, and y is an output of the function g. We could write the same calculation as:

$$y = g(f(x))$$

In this form, we do not use the intermediate z variable, showing that function g takes the output of function f directly as an input. This does not differ much from the above 2 equations but shows an important property of functions chained this way — since x is an input to the function f and then the output of the function f is an input to the function g, the output of the function g is influenced by x in some way, so there must exist a derivative which can inform us of this influence.

The forward pass through our model is a chain of functions similar to these examples. We are passing in samples, the data flows through all of the layers, and activation functions to form an output. Let's bring the equation and the code of the example model from chapter 1:

$$L = -\sum_{l=1}^{N} y_l log(\forall_{j=1}^{n_3} \frac{e^{\sum_{i=1}^{n_2}(\forall_{j=1}^{n_2} max(0,\sum_{i=1}^{n_1}(\forall_{j=1}^{n_1} max(0,\sum_{i=1}^{n_0} X_i w_{1,i,j}+b_{1,j}))_i w_{2,i,j}+b_{2,j}))_i w_{3,i,j}+b_{3,j}}{\sum_{k=1}^{n_3} e^{\sum_{i=1}^{n_2}(\forall_{j=1}^{n_2} max(0,\sum_{i=1}^{n_1}(\forall_{j=1}^{n_1} max(0,\sum_{i=1}^{n_0} X_i w_{1,i,j}+b_{1,k}))_i w_{2,i,j}+b_{2,k}))_i w_{3,i,k}+b_{3,k}}})$$

https://nnfs.io

```
loss = -np.log(
    np.sum(
        y * np.exp(
            np.dot(
                np.maximum(
                    0,
                    np.dot(
                        np.maximum(
                            0,
                            np.dot(
                                X,
                                w1.T
                            ) + b1
                        ),
                        w2.T
                    ) + b2
                ),
                w3.T
            ) + b3
        ) /
        np.sum(
            np.exp(
                np.dot(
                    np.maximum(
                        0,
                        np.dot(
                            np.maximum(
                                0,
                                np.dot(
                                    X,
                                    w1.T
                                ) + b1
                            ),
                            w2.T
                        ) + b2
                    ),
                    w3.T
                ) + b3
            ),
            axis=1,
            keepdims=True
        )
    )
)
```

https://nnfs.io

Fig 8.01: Code for a forward pass of an example neural network model.

If you look closely, you'll see that we are presenting the loss as a big function, or a chain of functions, of multiple inputs — input data, weights, and biases. We are passing input data to the first layer where we also have that layer's weights and biases, then the outputs flow through the ReLU activation function, and another layer, which brings more weights and biases, and another ReLU activation, up to the end — the output layer and softmax activation. The model output, along with the targets, is passed to the loss function, which returns the model's error. We can look at the loss function not only as a function that takes the model's output and targets as parameters to produce the error, but also as a function that takes targets, samples, and all of the weights and biases as inputs if we chain all of the functions performed during the forward pass as we've just shown in the images. To improve loss, we need to learn how each weight and bias impacts it. How to do that for a chain of functions? By using the chain rule. This rule says that the derivative of a function chain is a product of derivatives of all of the functions in this chain, for example:

$$\frac{d}{dx}f(g(x)) = \frac{df(g(x))}{dg(x)} \cdot \frac{dg(x)}{dx} = f'(g(x)) \cdot g'(x)$$

First, we wrote the derivative of the outer function, *f(g(x))*, with respect to the inner function, *g(x)*, as this inner function is its parameter. Next, we multiplied it by the derivative of the inner function, *g(x)*, with respect to its parameters, *x*. We also denoted this derivative using 2 different notations. With 3 functions and multiple inputs, the partial derivative of this function with respect to *x* is as follows (we can't use the prime notation in this case since we have to mention which variable we are deriving with respect to):

$$\frac{\partial}{\partial x}f(g(y, h(x, z))) = \frac{\partial f(g(y, h(x, z)))}{\partial g(y, h(x, z))} \cdot \frac{\partial g(y, h(x, z))}{\partial h(x, z)} \cdot \frac{\partial h(x, z)}{\partial x}$$

To calculate the partial derivative of a chain of functions with respect to some parameter, we take the partial derivative of the outer function with respect to the inner function in a chain to the parameter. Then multiply this partial derivative by the partial derivative of the inner function with respect to the more inner function in a chain to the parameter, then multiply this by the partial derivative of the more inner function with respect to the other function in the chain. We repeat this all the way down to the parameter in question. Notice, for example, how the middle derivative is with respect to *h(x, z)* and not *y* as *h(x, z)* is in the chain to the parameter *x*. The **chain rule** turns out to be the most important rule in finding the impact of singular input to the output of a chain of functions, which is the calculation of loss in our case. We'll use it again in the next chapter when we discuss and code backpropagation. For now, let's cover an example of the chain rule.

Let's solve the derivative of $h(x) = 3(2x^2)^5$. The first thing that we can notice here is that we have a complex function that can be split into two simpler functions. First is an equation part contained inside the parentheses, which we can write as $g(x) = 2x^2$. That's the inside function that we exponentiate and multiply with the rest of the equation. The remaining part of the equation can then be written as $f(y) = 3(y)^5$. *y* in this case is what we denoted as $g(x)=2x^2$ and when we combine it back, we get $h(x) = f(g(x)) = 3(2x^2)^5$. To calculate a derivative of this function, we start by taking that outside exponent, the ^{5}, and place it in front of the component that we are exponentiating to multiply it later by the leading 3, giving us 15. We then subtract 1 from the ^{5} exponent, leaving us with a ^{4}.

$$h(x) = f(g(x)) = 3(2x^2)^5 \quad \rightarrow \quad f'(g(x)) = 3 \cdot 5(2x^2)^{5-1} = 15(2x^2)^4$$

Then the chain rule informs us to multiply the above derivative of the outer function, with the derivative of the interior function, giving us:

$$\rightarrow \quad h'(x) = f'(g(x)) \cdot g'(x) = 15(2x^2)^4 \cdot \frac{d}{dx}2x^2 = 15(2x^2)^4 \cdot 2 \cdot \frac{d}{dx}x^2 =$$

$$= 15(2x^2)^4 \cdot 2 \cdot 2x^1 = 15(2x^2)^4 \cdot 4x$$

Recall that $4x$ was the derivative of $2x^2$, which is the inner function, $g(x)$. This highlights the **chain rule** concept in an example, allowing us to calculate the derivatives of more complex functions by chaining together the derivatives. Note that we multiplied by the derivative of that interior function, but left the interior function *unchanged* within the derivative of the outer function.

In theory, we could just stop here with a perfectly-usable derivative of the function. We can enter some input into $15(2x^2)^4 \cdot 4x$ and get the answer. That said, we can also go ahead and simplify this function for more practice. Coming back to the original problem, so far we've found:

$$f(x) = 3(2x^2)^5 \quad \rightarrow \quad f'(x) = 15(2x^2)^4 \cdot 4x$$

To simplify this derivative function, we first take $(2x^2)^4$ and distribute the ^{4} exponent:

$$f'(x) = 15(2x^2)^4 \cdot 4x = 15 \cdot (2^4 \cdot x^{2 \cdot 4}) \cdot 4x = 15 \cdot 2^4 \cdot x^8 \cdot 4x$$

Combine the *x's*:

$$f'(x) = 15 \cdot 2^4 \cdot x^8 \cdot 4x = 15 \cdot 2^4 \cdot 4 \cdot x^{8+1} = 15 \cdot 2^4 \cdot 4 \cdot x^9$$

And the constants:

$$f'(x) = 15 \cdot 2^4 \cdot 4 \cdot x^9 = 15 \cdot 16 \cdot 4 \cdot x^9 = 960x^9$$

We'll simplify derivatives later as well for faster computation — there's no reason to repeat the same operations when we can solve them in advance.

Hopefully, now you understand what derivatives and partial derivatives are, what the gradient is, what the derivative of the loss function with respect to weights and biases means, and how to use the chain rule. For now, these terms might sound disconnected, but we're going to use them all to perform gradient descent in the backpropagation step, which is the subject of the next chapters.

Summary

Let's summarize the rules that we have learned in this chapter.

The partial derivative of the sum with respect to any input equals 1:

$$f(x, y) = x + y \quad \rightarrow \quad \frac{\partial}{\partial x} f(x, y) = 1$$
$$\frac{\partial}{\partial y} f(x, y) = 1$$

The partial derivative of the multiplication operation with 2 inputs, with respect to any input, equals the other input:

$$f(x, y) = x \cdot y \quad \rightarrow \quad \frac{\partial}{\partial x} f(x, y) = y$$
$$\frac{\partial}{\partial y} f(x, y) = x$$

The partial derivative of the max function of 2 variables with respect to any of them is 1 if this variable is the biggest and 0 otherwise. An example of x:

$$f(x, y) = max(x, y) \quad \rightarrow \quad \frac{\partial}{\partial x} f(x, y) = 1(x > y)$$

The derivative of the max function of a single variable and 0 equals 1 if the variable is greater than 0 and 0 otherwise:

$$f(x) = max(x, 0) \quad \rightarrow \quad \frac{d}{dx} f(x) = 1(x > 0)$$

The derivative of chained functions equals the product of the partial derivatives of the subsequent functions:

$$\frac{d}{dx}f(g(x)) = \frac{d}{dg(x)}f(g(x)) \cdot \frac{d}{dx}g(x) = f'(g(x)) \cdot g'(x)$$

The same applies to the partial derivatives. For example:

$$\frac{\partial}{\partial x}f(g(y, h(x, z))) = f'(g(y, h(x, z))) \cdot g'(y, h(x, z)) \cdot h'(x, z)$$

The gradient is a vector of all possible partial derivatives. An example of a triple-input function:

$$\nabla f(x, y, z) = \begin{bmatrix} \frac{\partial}{\partial x}f(x, y, z) \\ \frac{\partial}{\partial y}f(x, y, z) \\ \frac{\partial}{\partial z}f(x, y, z) \end{bmatrix} = \begin{bmatrix} \frac{\partial}{\partial x} \\ \frac{\partial}{\partial y} \\ \frac{\partial}{\partial z} \end{bmatrix} f(x, y, z)$$

Supplementary Material: https://nnfs.io/ch8
Chapter code, further resources, and errata for this chapter.

Chapter 9

Backpropagation

Now that we have an idea of how to measure the impact of variables on a function's output, we can begin to write the code to calculate these partial derivatives to see their role in minimizing the model's loss. Before applying this to a complete neural network, let's start with a simplified forward pass with just one neuron. Rather than backpropagating from the loss function for a full neural network, let's backpropagate the ReLU function for a single neuron and act as if we intend to minimize the output for this single neuron. We're first doing this only as a demonstration to simplify the explanation, since minimizing the output from a ReLU activated neuron doesn't serve any purpose other than as an exercise. Minimizing the loss value is our end goal, but in this case, we'll start by showing how we can leverage the chain rule with derivatives and partial derivatives to calculate the impact of each variable on the ReLU activated output. We'll also start by minimizing this more basic output before jumping to the full network and overall loss.

Let's quickly recall the forward pass and atomic operations that we need to perform for this single neuron and ReLU activation. We'll use an example neuron with 3 inputs, which means that it also has 3 weights and a bias:

```
x = [1.0, -2.0, 3.0]  # input values
w = [-3.0, -1.0, 2.0]  # weights
b = 1.0  # bias
```

We then start with the first input, `x[0]`, and the related weight, `w[0]`:

Fig 9.01: Beginning a forward pass with the first input and weight.

We have to multiply the input by the weight:

```
x = [1.0, -2.0, 3.0]  # input values
w = [-3.0, -1.0, 2.0]  # weights
b = 1.0  # bias


xw0 = x[0] * w[0]
print(xw0)


>>>
-3.0
```

Visually:

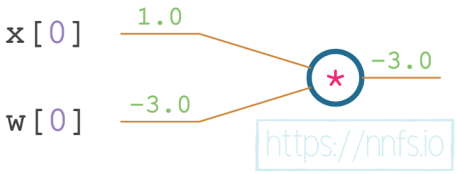


Fig 9.02: The first input and weight multiplication.

We repeat this operation for `x1`, `w1` and `x2`, `w2` pairs:

```
xw1 = x[1] * w[1]
xw2 = x[2] * w[2]
print(xw1, xw2)
```

```
>>>
2.0 6.0
```

Visually:

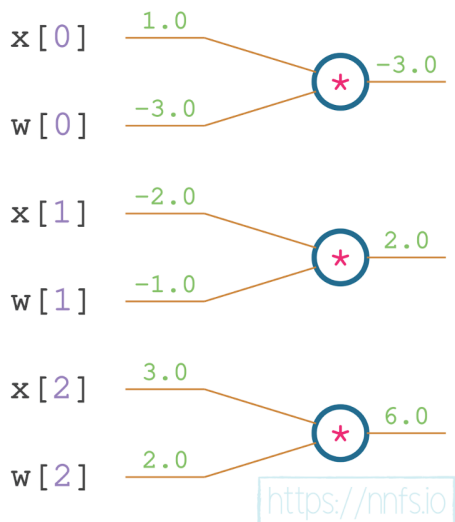


Fig 9.03: Input and weight multiplication of all of the inputs.

Code all together:

```
# Forward pass
x = [1.0, -2.0, 3.0]  # input values
w = [-3.0, -1.0, 2.0]  # weights
b = 1.0  # bias

# Multiplying inputs by weights
xw0 = x[0] * w[0]
xw1 = x[1] * w[1]
xw2 = x[2] * w[2]
print(xw0, xw1, xw2)
```

```
>>>
-3.0 2.0 6.0
```

The next operation to perform is a sum of all weighted inputs with a bias:

```
# Forward pass
x = [1.0, -2.0, 3.0]  # input values
w = [-3.0, -1.0, 2.0]  # weights
b = 1.0  # bias

# Multiplying inputs by weights
xw0 = x[0] * w[0]
xw1 = x[1] * w[1]
xw2 = x[2] * w[2]
print(xw0, xw1, xw2, b)

# Adding weighted inputs and a bias
z = xw0 + xw1 + xw2 + b
print(z)
```

```
>>>
-3.0 2.0 6.0 1.0
6.0
```

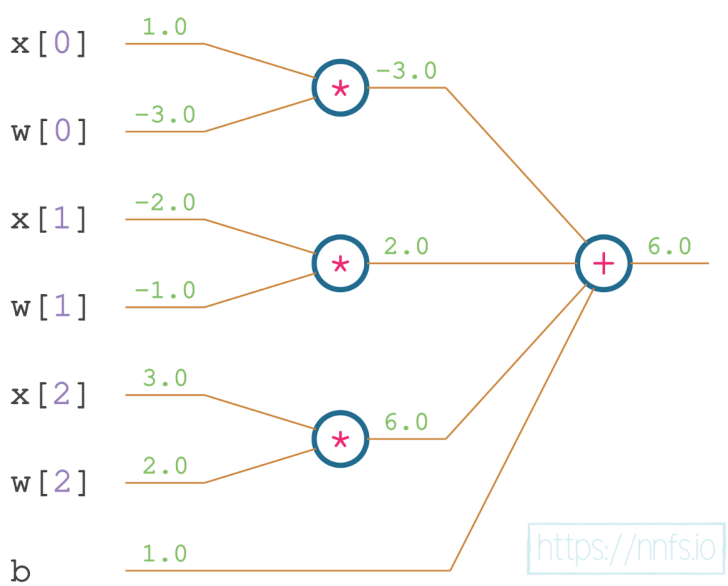


Fig 9.04: Weighted inputs and bias addition.

This forms the neuron's output. The last step is to apply the ReLU activation function on this output:

```
# Forward pass
x = [1.0, -2.0, 3.0]  # input values
w = [-3.0, -1.0, 2.0]  # weights
b = 1.0  # bias

# Multiplying inputs by weights
xw0 = x[0] * w[0]
xw1 = x[1] * w[1]
xw2 = x[2] * w[2]
print(xw0, xw1, xw2, b)

# Adding weighted inputs and a bias
z = xw0 + xw1 + xw2 + b
print(z)

# ReLU activation function
y = max(z, 0)
print(y)
```

```
>>>
-3.0 2.0 6.0 1.0
6.0
6.0
```

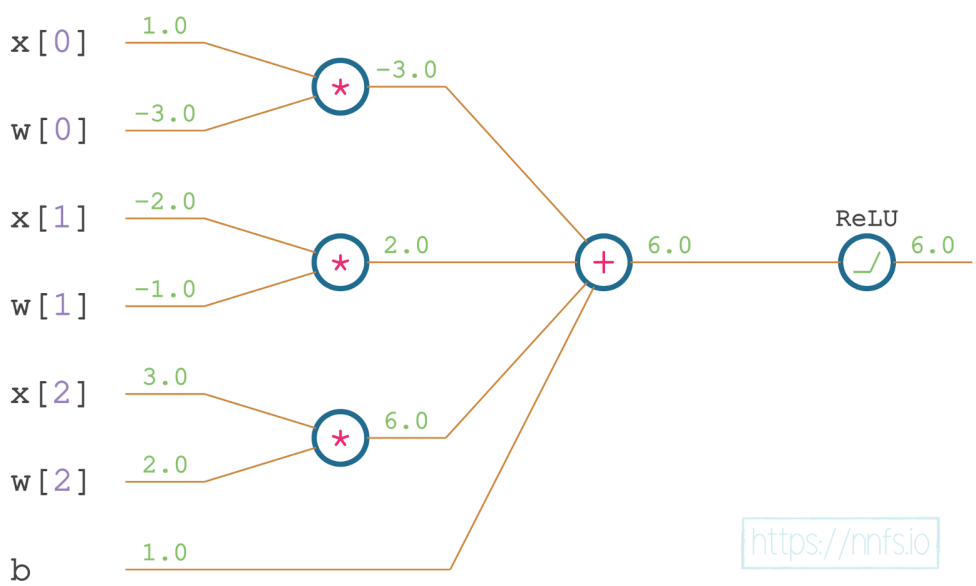


Fig 9.05: ReLU activation applied to the neuron output.

This is the full forward pass through a single neuron and a ReLU activation function. Let's treat all of these chained functions as one big function which takes input values (x), weights (w), and bias (b), as inputs, and outputs y. This big function consists of multiple simpler functions — there is a multiplication of input values and weights, sum of these values and bias, as well as a *max* function as the ReLU activation — 3 chained functions in total:

The first step is to backpropagate our gradients by calculating derivatives and partial derivatives with respect to each of our parameters and inputs. To do this, we're going to use the **chain rule**. Recall that the chain rule for a function stipulates that the derivative for nested functions like *f(g(x))* solves to:

$$\frac{d}{dx}f(g(x)) = \frac{d}{dg(x)}f(g(x)) \cdot \frac{d}{dx}g(x) = f'(g(x)) \cdot g'(x)$$

This big function that we just mentioned can be, in the context of our neural network, loosely interpreted as:

$$ReLU(\sum[inputs \cdot weights] + bias)$$

Or in the form that matches code more precisely as:

$$ReLU(x_0w_0 + x_1w_1 + x_2w_2 + b)$$

Our current task is to calculate how much each of the inputs, weights, and a bias impacts the output. We'll start by considering what we need to calculate for the partial derivative of w_0, for example. But first, let's rewrite our equation to the form that will allow us to determine how to calculate the derivatives more easily:

$y = ReLU(sum(mul(x_0, w_0), mul(x_1, w_1), mul(x_2, w_2), b))$

The above equation contains 3 nested functions: *ReLU*, a sum of weighted inputs and a bias, and multiplications of the inputs and weights. To calculate the impact of the example weight, w_0, on the output, the chain rule tells us to calculate the derivative of *ReLU* with respect to its parameter, which is the sum, then multiply it with the partial derivative of the sum operation with respect to its $mul(x_0, w_0)$ input, as this input contains the parameter in question. Then, multiply this with the partial derivative of the multiplication operation with respect to the x_0 input. Let's see this in a simplified equation:

$$\frac{\partial}{\partial x_0}[ReLU(sum(mul(x_0, w_0), mul(x_1, w_1), mul(x_2, w_2), b))] =$$

$$\frac{dReLU()}{dsum()} \cdot \frac{\partial sum()}{\partial mul(x_0, w_0)} \cdot \frac{\partial mul(x_0, w_0)}{\partial x_0}$$

For legibility, we did not denote the *ReLU*() parameter, which is the full sum, and the sum parameters, which are all of the multiplications of inputs and weights. We excluded this because the equation would be longer and harder to read. This equation shows that we have to calculate the derivatives and partial derivatives of all of the atomic operations and multiply them to acquire the impact that x_0 makes on the output. We can then repeat this to calculate all of the other remaining impacts. The derivatives with respect to the weights and a bias will inform us about their impact and will be used to update these weights and bias. The derivatives with respect to inputs are used to chain more layers by passing them to the previous function in the chain.

We'll have multiple chained layers of neurons in the neural network model, followed by the loss function. We want to know the impact of a given weight or bias on the loss. That means that we will have to calculate the derivative of the loss function (which we'll do later in this chapter) and apply the chain rule with the derivatives of all activation functions and neurons in all of the consecutive layers. The derivative with respect to the layer's inputs, as opposed to the derivative with respect to the weights and biases, is not used to update any parameters. Instead, it is used to chain to another layer (which is why we backpropagate to the previous layer in a chain).

During the backward pass, we'll calculate the derivative of the loss function, and use it to multiply with the derivative of the activation function of the output layer, then use this result to multiply by the derivative of the output layer, and so on, through all of the hidden layers and activation functions. Inside these layers, the derivative with respect to the weights and biases will form the gradients that we'll use to update the weights and biases. The derivatives with respect to inputs will form the gradient to chain with the previous layer, This layer can calculate the impact of its weights and biases on the loss and backpropagate gradients on inputs further.

For this example, let's assume that our neuron receives a gradient of *1* from the next layer. We're making up this value for demonstration purposes, and a value of *1* won't change the values, which means that we can more easily show all of the processes. We are going to use the color of red for derivatives:

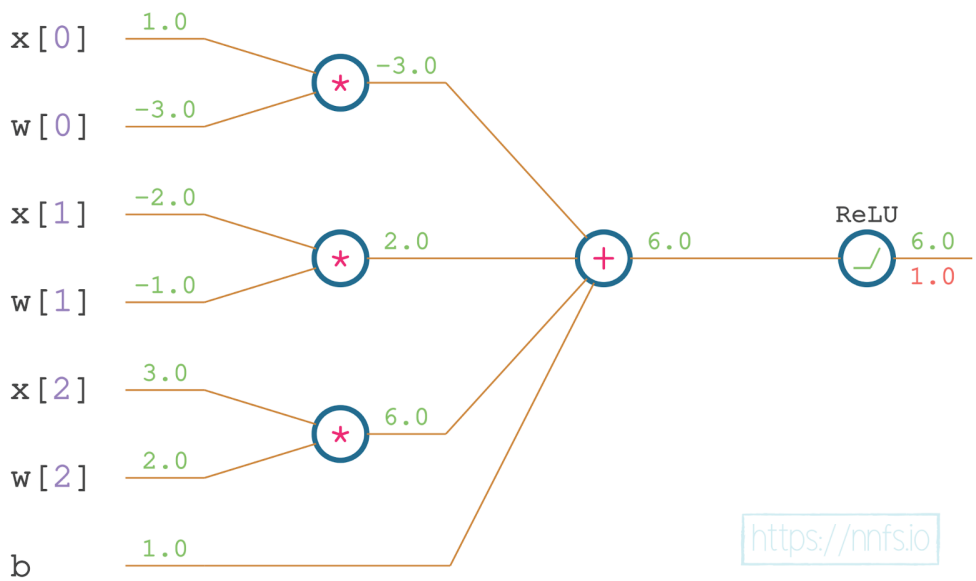


Fig 9.06: Initial gradient (received during backpropagation).

Recall that the derivative of *ReLU()* with respect to its input is *1,* if the input is greater than *0,* and *0* otherwise:

$$f(x) = max(x, 0) \quad \rightarrow \quad \frac{d}{dx} f(x) = 1(x > 0)$$

We can write that in Python as:

```
drelu_dz = (1. if z > 0 else 0.)
```

Where the `drelu_dz` means the derivative of the *ReLU* function with respect to *z* — we used *z* instead of *x* from the equation since the equation denotes the *max* function in general, and we are applying it to the neuron's output, which is *z*.

The input value to the *ReLU* function is *6*, so the derivative equals *1*. We have to use the chain rule and multiply this derivative with the derivative received from the next layer, which is *1* for the purpose of this example:

```
# Forward pass
x = [1.0, -2.0, 3.0]  # input values
w = [-3.0, -1.0, 2.0]  # weights
b = 1.0  # bias
```

```
# Multiplying inputs by weights
xw0 = x[0] * w[0]
xw1 = x[1] * w[1]
xw2 = x[2] * w[2]

# Adding weighted inputs and a bias
z = xw0 + xw1 + xw2 + b

# ReLU activation function
y = max(z, 0)

# Backward pass

# The derivative from the next layer
dvalue = 1.0

# Derivative of ReLU and the chain rule
drelu_dz = dvalue * (1. if z > 0 else 0.)
print(drelu_dz)
```

```
>>>
1.0
```

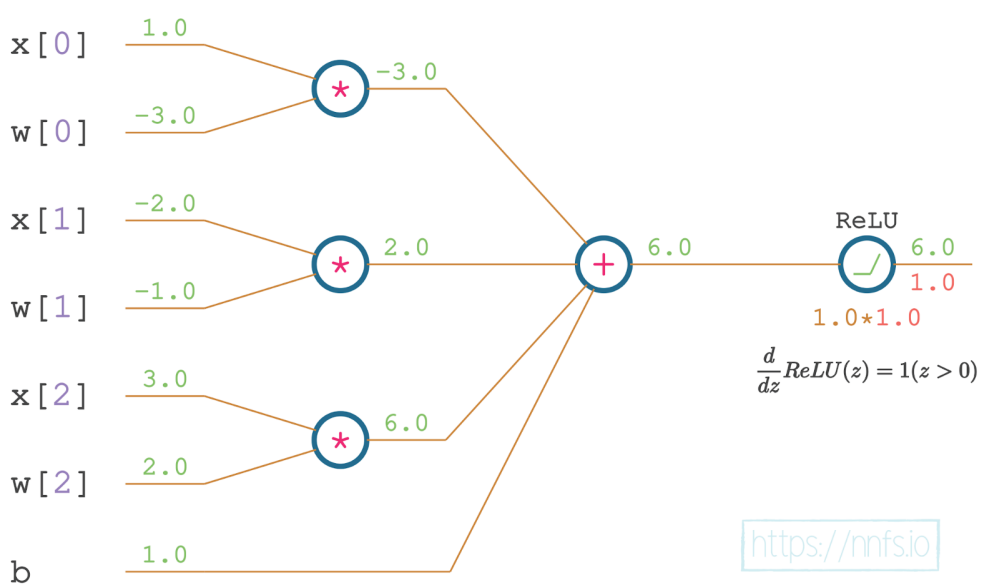


Fig 9.07: Derivative of the ReLU function and chain rule.

This results with the derivative of *1*:

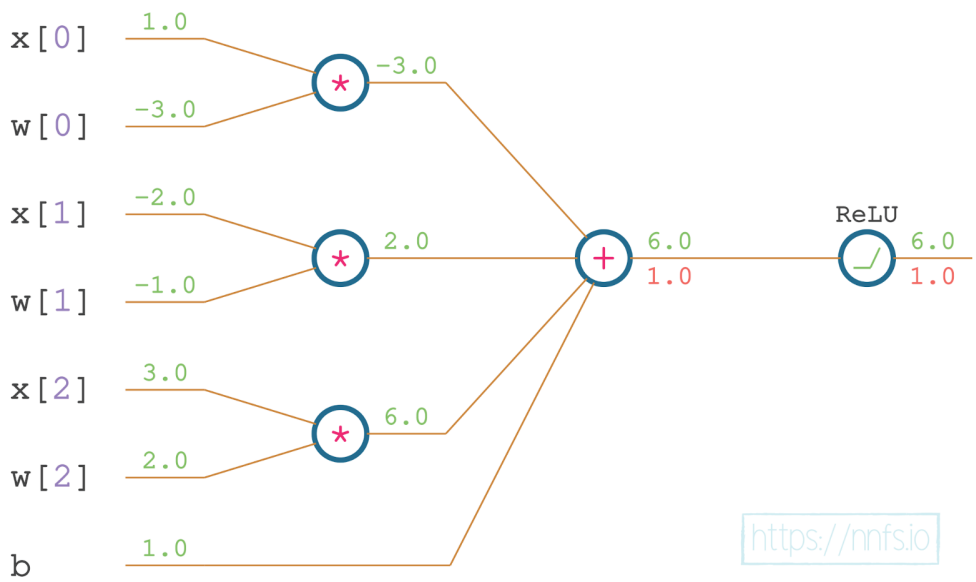


Fig 9.08: ReLU and chain rule gradient.

Moving backward through our neural network, what is the function that comes immediately before we perform the activation function?

It's a sum of the weighted inputs and bias. This means that we — want to calculate the partial derivative of the sum function, and then, using the chain rule, multiply this by the partial derivative of the subsequent, outer, function, which is *ReLU*. We'll call these results the:

- `drelu_dxw0` — the partial **d**erivative of the **ReLU** w.r.t. the first weighed input, $\boldsymbol{w_0 x_0}$,
- `drelu_dxw1` — the partial **d**erivative of the **ReLU** w.r.t. the second weighed input, $\boldsymbol{w_1 x_1}$,
- `drelu_dxw2` — the partial **d**erivative of the **ReLU** w.r.t. the third weighed input, $\boldsymbol{w_2 x_2}$,
- `drelu_db` — the partial **d**erivative of the **ReLU** with respect to the bias, $\boldsymbol{b}$.

The partial derivative of the sum operation is always *1*, no matter the inputs:

$$f(x, y) = x + y \quad \rightarrow \quad \frac{\partial}{\partial x} f(x, y) = 1$$

$$\frac{\partial}{\partial y} f(x, y) = 1$$

The weighted inputs and bias are summed at this stage. So we will calculate the partial derivatives of the sum operation with respect to each of these, multiplied by the partial derivative for the subsequent function (using the chain rule), which is the *ReLU* function, denoted by `drelu_dz`.

For the first partial derivative:

```
dsum_dxw0 = 1
drelu_dxw0 = drelu_dz * dsum_dxw0
```

To be clear, the `dsum_dxw0` above means the partial **d**erivative of the **sum** with respect to the **x** (input), **w**eighted, for the **0**th pair of inputs and weights. *1* is the value of this partial derivative, which we multiply, using the chain rule, with the derivative of the subsequent function, which is the *ReLU* function.

Again, we have to apply the chain rule and multiply the derivative of the ReLU function with the partial derivative of the sum, with respect to the first weighted input:

```
# Forward pass
x = [1.0, -2.0, 3.0]  # input values
w = [-3.0, -1.0, 2.0]  # weights
b = 1.0  # bias

# Multiplying inputs by weights
xw0 = x[0] * w[0]
xw1 = x[1] * w[1]
xw2 = x[2] * w[2]

# Adding weighted inputs and a bias
z = xw0 + xw1 + xw2 + b

# ReLU activation function
y = max(z, 0)

# Backward pass

# The derivative from the next layer
dvalue = 1.0

# Derivative of ReLU and the chain rule
drelu_dz = dvalue * (1. if z > 0 else 0.)
print(drelu_dz)

# Partial derivatives of the multiplication, the chain rule
dsum_dxw0 = 1
drelu_dxw0 = drelu_dz * dsum_dxw0
print(drelu_dxw0)
```

```
>>>
1.0
1.0
```

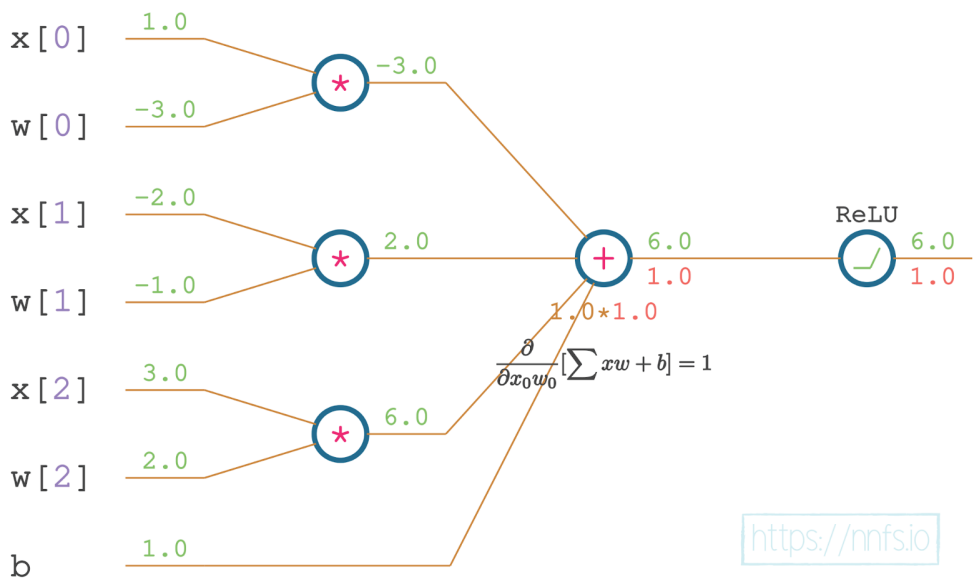


Fig 9.09: Partial derivative of the sum function w.r.t. the first weighted input; the chain rule.

This results with a partial derivative of 1 again:

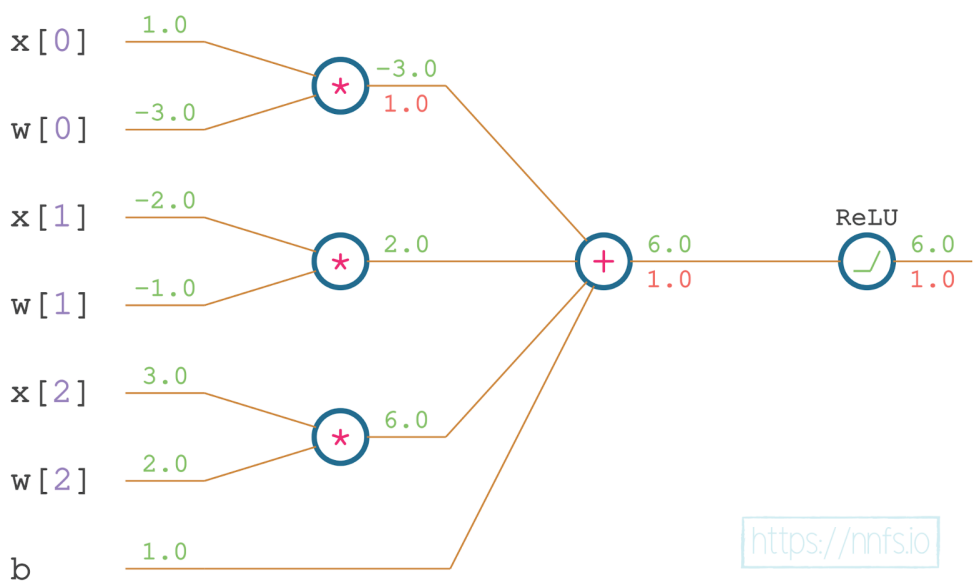


Fig 9.10: The sum and chain rule gradient (for the first weighted input).

We can then perform the same operation with the next weighed input:

```
dsum_dxw1 = 1
drelu_dxw1 = drelu_dz * dsum_dxw1
```

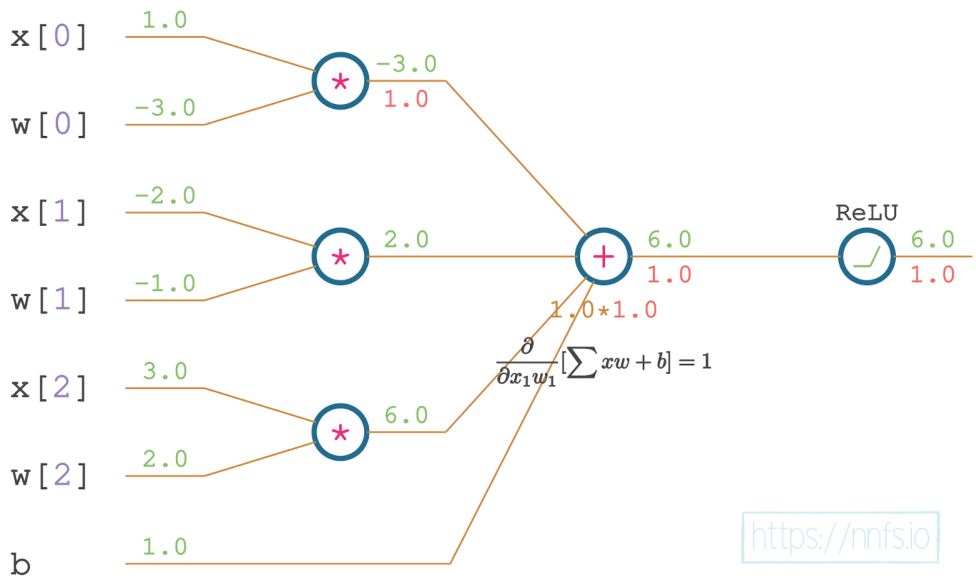


Fig 9.11: Partial derivative of the sum function w.r.t. the second weighted input; the chain rule.

Which results with the next calculated partial derivative:

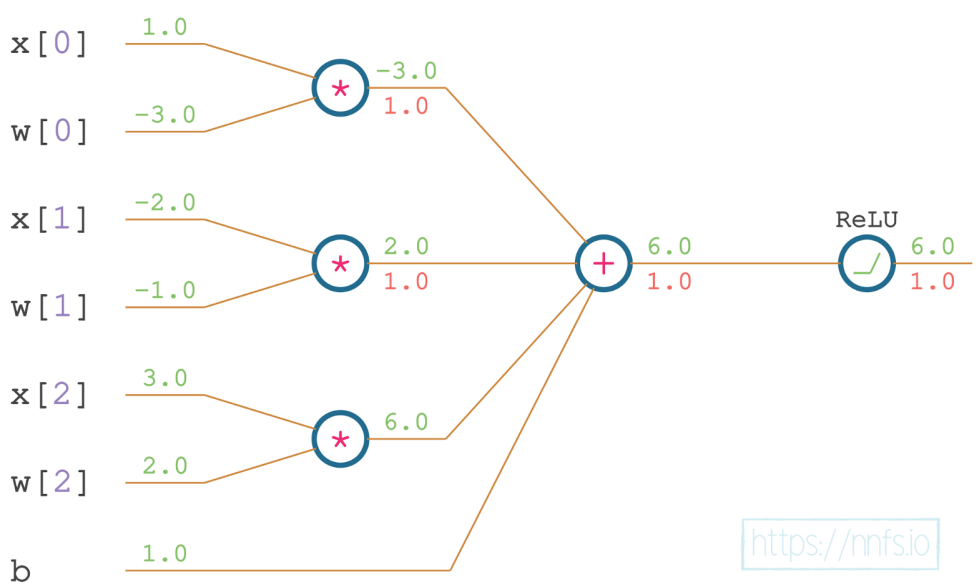


Fig 9.12: The sum and chain rule gradient (for the second weighted input).

And the last weighted input:

```
dsum_dxw2 = 1
drelu_dxw2 = drelu_dz * dsum_dxw2
```

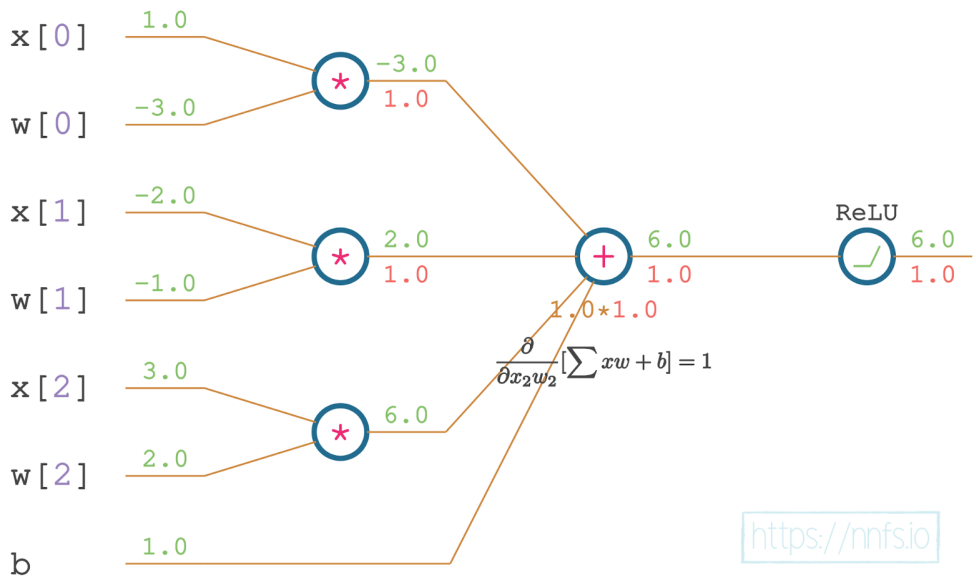


Fig 9.13: Partial derivative of the sum function w.r.t. the third weighted input; the chain rule.

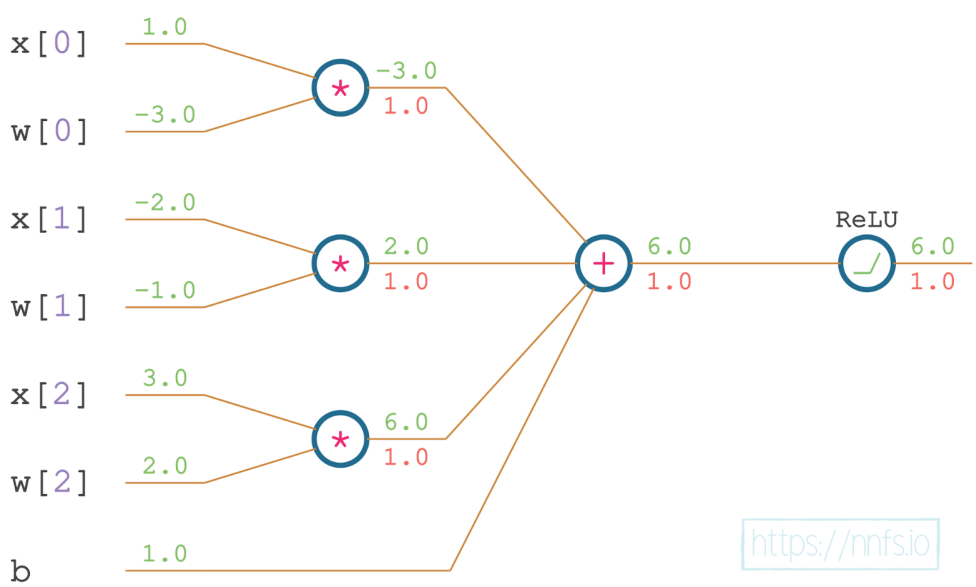


Fig 9.14: The sum and chain rule gradient (for the third weighted input).

Then the bias:

```
dsum_db = 1
drelu_db = drelu_dz * dsum_db
```

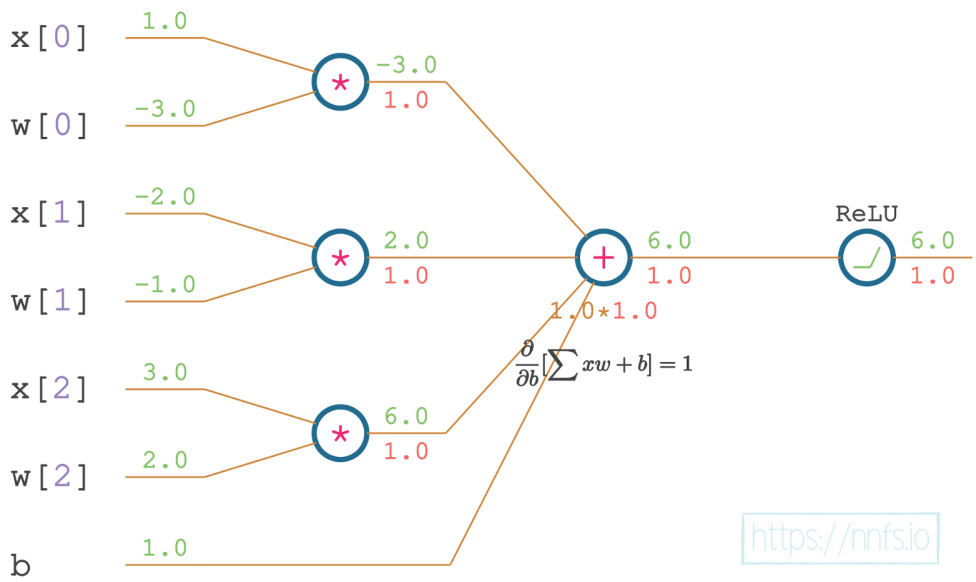


Fig 9.15: Partial derivative of the sum function w.r.t. the bias; the chain rule.

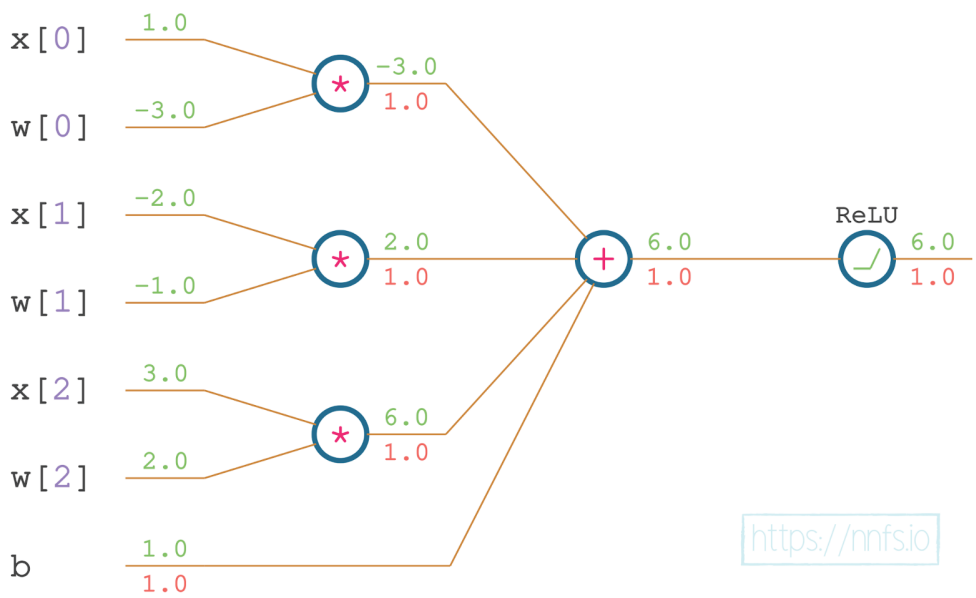


Fig 9.16: The sum and chain rule gradient (for the bias).

Let's add these partial derivatives, with the applied chain rule, to our code:

```
# Forward pass
x = [1.0, -2.0, 3.0]  # input values
w = [-3.0, -1.0, 2.0]  # weights
b = 1.0  # bias

# Multiplying inputs by weights
xw0 = x[0] * w[0]
xw1 = x[1] * w[1]
xw2 = x[2] * w[2]

# Adding weighted inputs and a bias
z = xw0 + xw1 + xw2 + b

# ReLU activation function
y = max(z, 0)

# Backward pass

# The derivative from the next layer
dvalue = 1.0

# Derivative of ReLU and the chain rule
drelu_dz = dvalue * (1. if z > 0 else 0.)
print(drelu_dz)

# Partial derivatives of the multiplication, the chain rule
dsum_dxw0 = 1
dsum_dxw1 = 1
dsum_dxw2 = 1
dsum_db = 1
drelu_dxw0 = drelu_dz * dsum_dxw0
drelu_dxw1 = drelu_dz * dsum_dxw1
drelu_dxw2 = drelu_dz * dsum_dxw2
drelu_db = drelu_dz * dsum_db
print(drelu_dxw0, drelu_dxw1, drelu_dxw2, drelu_db)
```

```
>>>
1.0
1.0 1.0 1.0 1.0
```

Continuing backward, the function that comes before the sum is the multiplication of weights and inputs. The derivative for a product is whatever the input is being multiplied by. Recall:

$$f(x, y) = x \cdot y \quad \rightarrow \quad \frac{\partial}{\partial x} f(x, y) = y$$

$$\frac{\partial}{\partial y} f(x, y) = x$$

The partial derivative of *f* with respect to *x* equals *y*. The partial derivative of *f* with respect to *y* equals *x*. Following this rule, the partial derivative of the first *weighted input* with respect to the *input* equals the *weight* (the other input of this function). Then, we have to apply the chain rule and multiply this partial derivative with the partial derivative of the subsequent function, which is the sum (we just calculated its partial derivative earlier in this chapter):

```
dmul_dx0 = w[0]
drelu_dx0 = drelu_dxw0 * dmul_dx0
```

This means that we are calculating the partial derivative with respect to the x_0 input, the value of which is w_0, and we are applying the chain rule with the derivative of the subsequent function, which is `drelu_dxw0`.

This is a good time to point out that, as we apply the chain rule in this way — working backward by taking the *ReLU()* derivative, taking the summing operation's derivative, multiplying both, and so on, this is a process called **backpropagation** using the **chain rule**. As the name implies, the resulting output function's gradients are passed back through the neural network, using multiplication of the gradient of subsequent functions from later layers with the current one. Let's add this partial derivative to the code and show it on the chart:

```
# Forward pass
x = [1.0, -2.0, 3.0]  # input values
w = [-3.0, -1.0, 2.0]  # weights
b = 1.0  # bias

# Multiplying inputs by weights
xw0 = x[0] * w[0]
xw1 = x[1] * w[1]
xw2 = x[2] * w[2]

# Adding weighted inputs and a bias
z = xw0 + xw1 + xw2 + b

# ReLU activation function
y = max(z, 0)

# Backward pass
```

```
# The derivative from the next layer
dvalue = 1.0

# Derivative of ReLU and the chain rule
drelu_dz = dvalue * (1. if z > 0 else 0.)
print(drelu_dz)

# Partial derivatives of the multiplication, the chain rule
dsum_dxw0 = 1
dsum_dxw1 = 1
dsum_dxw2 = 1
dsum_db = 1
drelu_dxw0 = drelu_dz * dsum_dxw0
drelu_dxw1 = drelu_dz * dsum_dxw1
drelu_dxw2 = drelu_dz * dsum_dxw2
drelu_db = drelu_dz * dsum_db
print(drelu_dxw0, drelu_dxw1, drelu_dxw2, drelu_db)

# Partial derivatives of the multiplication, the chain rule
dmul_dx0 = w[0]
drelu_dx0 = drelu_dxw0 * dmul_dx0
print(drelu_dx0)
```

```
>>>
1.0
1.0 1.0 1.0 1.0
-3.0
```

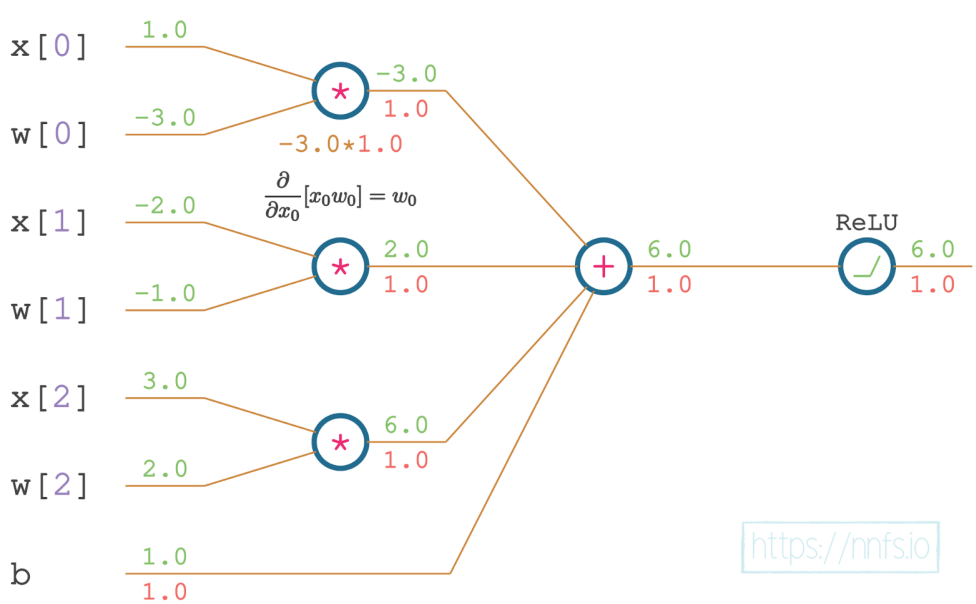


Fig 9.17: Partial derivative of the multiplication function w.r.t. the first input; the chain rule.

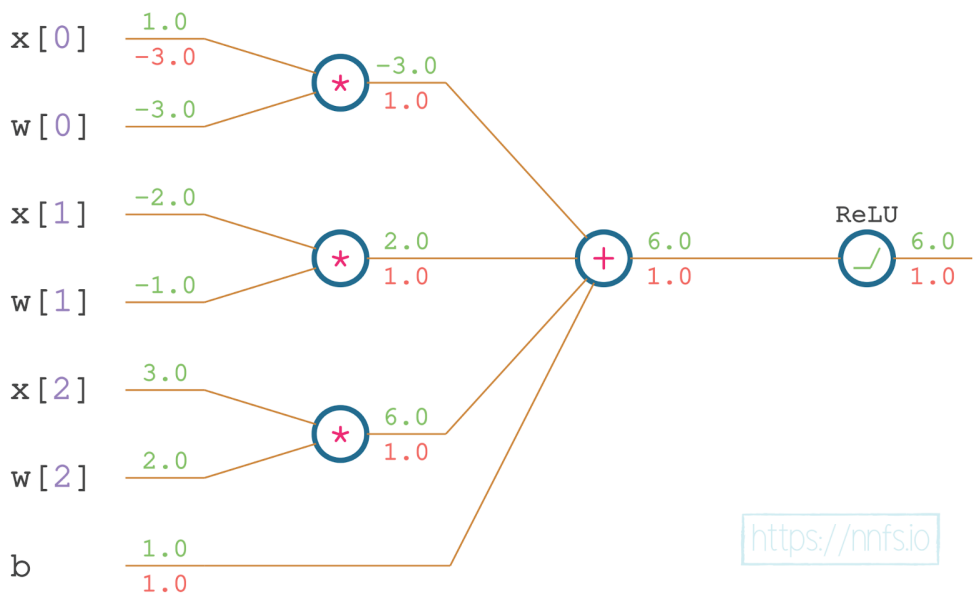


Fig 9.18: The multiplication and chain rule gradient (for the first input).

We perform the same operation for other inputs and weights:

```
# Forward pass
x = [1.0, -2.0, 3.0]  # input values
w = [-3.0, -1.0, 2.0]  # weights
b = 1.0  # bias

# Multiplying inputs by weights
xw0 = x[0] * w[0]
xw1 = x[1] * w[1]
xw2 = x[2] * w[2]

# Adding weighted inputs and a bias
z = xw0 + xw1 + xw2 + b

# ReLU activation function
y = max(z, 0)

# Backward pass
# The derivative from the next layer
dvalue = 1.0
```

```
# Derivative of ReLU and the chain rule
drelu_dz = dvalue * (1. if z > 0 else 0.)
print(drelu_dz)

# Partial derivatives of the multiplication, the chain rule
dsum_dxw0 = 1
dsum_dxw1 = 1
dsum_dxw2 = 1
dsum_db = 1
drelu_dxw0 = drelu_dz * dsum_dxw0
drelu_dxw1 = drelu_dz * dsum_dxw1
drelu_dxw2 = drelu_dz * dsum_dxw2
drelu_db = drelu_dz * dsum_db
print(drelu_dxw0, drelu_dxw1, drelu_dxw2, drelu_db)

# Partial derivatives of the multiplication, the chain rule
dmul_dx0 = w[0]
dmul_dx1 = w[1]
dmul_dx2 = w[2]
dmul_dw0 = x[0]
dmul_dw1 = x[1]
dmul_dw2 = x[2]
drelu_dx0 = drelu_dxw0 * dmul_dx0
drelu_dw0 = drelu_dxw0 * dmul_dw0
drelu_dx1 = drelu_dxw1 * dmul_dx1
drelu_dw1 = drelu_dxw1 * dmul_dw1
drelu_dx2 = drelu_dxw2 * dmul_dx2
drelu_dw2 = drelu_dxw2 * dmul_dw2
print(drelu_dx0, drelu_dw0, drelu_dx1, drelu_dw1, drelu_dx2, drelu_dw2)
```

```
>>>
1.0
1.0 1.0 1.0 1.0
-3.0 1.0 -1.0 -2.0 2.0 3.0
```

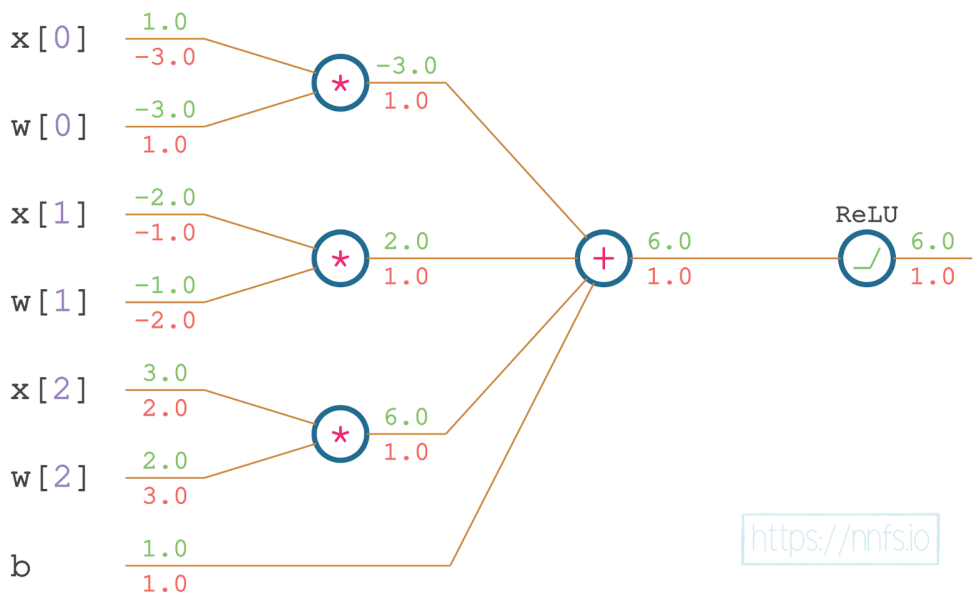


Fig 9.19: Complete backpropagation graph.

Anim 9.01-9.19: https://nnfs.io/pro

That's the complete set of the activated neuron's partial derivatives with respect to the inputs, weights and a bias.

Recall the equation from the beginning of this chapter:

$$\frac{\partial}{\partial x_0}[ReLU(sum(mul(x_0, w_0), mul(x_1, w_1), mul(x_2, w_2), b))] =$$

$$\frac{dReLU()}{dsum()} \cdot \frac{\partial sum()}{\partial mul(x_0, w_0)} \cdot \frac{\partial mul(x_0, w_0)}{\partial x_0}$$

Since we have the complete code and we are applying the chain rule from this equation, let's see what we can optimize in these calculations. We applied the chain rule to calculate the partial

derivative of the ReLU activation function with respect to the first input, x_0. In our code, let's take the related lines of the code and simplify them:

```
drelu_dx0 = drelu_dxw0 * dmul_dx0
```

where:

```
dmul_dx0 = w[0]
```

then:

```
drelu_dx0 = drelu_dxw0 * w[0]
```

where:

```
drelu_dxw0 = drelu_dz * dsum_dxw0
```

then:

```
drelu_dx0 = drelu_dz * dsum_dxw0 * w[0]
```

where:

```
dsum_dxw0 = 1
```

then:

```
drelu_dx0 = drelu_dz * 1 * w[0] = drelu_dz * w[0]
```

where:

```
drelu_dz = dvalue * (1. if z > 0 else 0.)
```

then:

```
drelu_dx0 = dvalue * (1. if z > 0 else 0.) * w[0]
```

```
drelu_dx0 = drelu_dxw0 * dmul_dx0

dmul_dx0 = w[0]

drelu_dxw0 = drelu_dz * dsum_dxw0

dsum_dxw0 = 1

drelu_dz = dvalue * (1. if z > 0 else 0.)
```

https://nnfs.io

Fig 9.20: How to apply the chain rule for the partial derivative of ReLU w.r.t. first input

```
drelu_dx0 = dvalue * (1. if z > 0 else 0.) * w[0]
```

Fig 9.21: The chain rule applied for the partial derivative of ReLU w.r.t. first input

Anim 9.20-9.21: https://nnfs.io/com

In this equation, starting from the left-hand side, is the derivative calculated in the next layer, with respect to its inputs — this is the gradient backpropagated to the current layer, which is the derivative of the *ReLU* function, and the partial derivative of the neuron's function with respect to the x_0 input. This is all multiplied by applying the chain rule to calculate the impact of the input to the neuron on the whole function's output.

The partial derivative of a neuron's function, with respect to the weight, is the input related to this weight, and, with respect to the input, is the related weight. The partial derivative of the neuron's function with respect to the bias is always 1. We multiply them with the derivative of the subsequent function (which was *1* in this example) to get the final derivatives. We are going to code all of these derivatives in the Dense layer's class and the ReLU activation class for the backpropagation step.

All together, the partial derivatives above, combined into a vector, make up our gradients. Our gradients could be represented as:

```
dx = [drelu_dx0, drelu_dx1, drelu_dx2]  # gradients on inputs
dw = [drelu_dw0, drelu_dw1, drelu_dw2]  # gradients on weights
db = drelu_db  # gradient on bias...just 1 bias here
```

For this single neuron example, we also won't need our `dx`. With many layers, we will continue backpropagating to preceding layers with the partial derivative with respect to our inputs.

Continuing the single neuron example, we can now apply these gradients to the weights to hopefully minimize the output. This is typically the purpose of the **optimizer** (discussed in the following chapter), but we can show a simplified version of this task by directly applying a negative fraction of the gradient to our weights. We apply a negative fraction to this gradient since we want to decrease the final output value, and the gradient shows the direction of the steepest ascent. For example, our current weights and bias are:

```
print(w, b)

>>>
[-3.0, -1.0, 2.0] 1.0
```

We can then apply a fraction of the gradients to these values:

```
w[0] += -0.001 * dw[0]
w[1] += -0.001 * dw[1]
w[2] += -0.001 * dw[2]
b += -0.001 * db

print(w, b)


>>>
[-3.001, -0.998, 1.997] 0.999
```

Now, we've slightly changed the weights and bias in such a way so as to decrease the output somewhat intelligently. We can see the effects of our tweaks on the output by doing another forward pass:

```
# Multiplying inputs by weights
xw0 = x[0] * w[0]
xw1 = x[1] * w[1]
xw2 = x[2] * w[2]
```

```
# Adding
z = xw0 + xw1 + xw2 + b

# ReLU activation function
y = max(z, 0)
print(y)
```

```
>>>
5.985
```

We've successfully decreased this neuron's output from 6.000 to 5.985. Note that it does not make sense to decrease the neuron's output in a real neural network; we were doing this purely as a simpler exercise than the full network. We want to decrease the loss value, which is the last calculation in the chain of calculations during the forward pass, and it's the first one to calculate the gradient during the backpropagation. We've minimized the ReLU output of a single neuron only for the purpose of this example to show that we actually managed to decrease the value of chained functions intelligently using the derivatives, partial derivatives, and chain rule. Now, we'll apply the one-neuron example to the list of samples and expand it to an entire layer of neurons. To begin, let's set a list of 3 samples for input, where each sample consists of 4 features. For this example, our network will consist of a single hidden layer, containing 3 neurons (lists of 3 weight sets and 3 biases). We're not going to describe the forward pass again, but the backward pass, in this case, needs further explanation.

So far, we have performed an example backward pass with a single neuron, which received a singular derivative to apply the chain rule. Let's consider multiple neurons in the following layer. A single neuron of the current layer connects to all of them — they all receive the output of this neuron. What will happen during backpropagation? Each neuron from the next layer will return a partial derivative of its function with respect to this input. The neuron in the current layer will receive a vector consisting of these derivatives. We need this to be a singular value for a singular neuron. To continue backpropagation, we need to sum this vector.

Now, let's replace the current singular neuron with a layer of neurons. As opposed to a single neuron, a layer outputs a vector of values instead of a singular value. Each neuron in a layer connects to all of the neurons in the next layer. During backpropagation, each neuron from the current layer will receive a vector of partial derivatives the same way that we described for a single neuron. With a layer of neurons, it'll take the form of a list of these vectors, or a 2D array. We know that we need to perform a sum, but what should we sum and what is the result supposed to be? Each neuron is going to output a gradient of the partial derivatives with respect to all of its inputs, and all neurons will form a list of these vectors. We need to sum along the inputs — the first input to all of the neurons, the second input, and so on. We'll have to sum columns.

To calculate the partial derivatives with respect to inputs, we need the weights — the partial derivative with respect to the input equals the related weight. This means that the array of partial derivatives with respect to all of the inputs equals the array of weights. Since this array is transposed, we'll need to sum its rows instead of columns. To apply the chain rule, we need to multiply them by the gradient from the subsequent function.

In the code to show this, we take the transposed weights, which are the transposed array of the derivatives with respect to inputs, and multiply them by their respective gradients (related to given neurons) to apply the chain rule. Then we sum along with the inputs. Then we calculate the gradient for the next layer in backpropagation. The "next" layer in backpropagation is the previous layer in the order of creation of the model:

```
import numpy as np

# Passed-in gradient from the next layer
# for the purpose of this example we're going to use
# a vector of 1s
dvalues = np.array([[1., 1., 1.]])

# We have 3 sets of weights - one set for each neuron
# we have 4 inputs, thus 4 weights
# recall that we keep weights transposed
weights = np.array([[0.2, 0.8, -0.5, 1],
                    [0.5, -0.91, 0.26, -0.5],
                    [-0.26, -0.27, 0.17, 0.87]]).T

# Sum weights related to the given input multiplied by
# the gradient related to the given neuron
dx0 = sum([weights[0][0]*dvalues[0][0], weights[0][1]*dvalues[0][1],
           weights[0][2]*dvalues[0][2]])
dx1 = sum([weights[1][0]*dvalues[0][0], weights[1][1]*dvalues[0][1],
           weights[1][2]*dvalues[0][2]])
dx2 = sum([weights[2][0]*dvalues[0][0], weights[2][1]*dvalues[0][1],
           weights[2][2]*dvalues[0][2]])
dx3 = sum([weights[3][0]*dvalues[0][0], weights[3][1]*dvalues[0][1],
           weights[3][2]*dvalues[0][2]])

dinputs = np.array([dx0, dx1, dx2, dx3])

print(dinputs)


>>>
[ 0.44 -0.38 -0.07  1.37]
```

`dinputs` is a gradient of the neuron function with respect to inputs.

We defined the gradient of the subsequent function (dvalues) as a row vector, which we'll explain shortly. From NumPy's perspective, and since both weights and dvalues are NumPy arrays, we can simplify the dx0 to dx3 calculation. Since the weights array is formatted so that the rows contain weights related to each input (weights for all neurons for the given input), we can multiply them by the gradient vector directly:

```
import numpy as np

# Passed-in gradient from the next layer
# for the purpose of this example we're going to use
# a vector of 1s, 3 derivatives - one for each neuron
dvalues = np.array([[1., 1., 1.]])

# We have 3 sets of weights - one set for each neuron
# we have 4 inputs, thus 4 weights
# recall that we keep weights transposed
weights = np.array([[0.2, 0.8, -0.5, 1],
                    [0.5, -0.91, 0.26, -0.5],
                    [-0.26, -0.27, 0.17, 0.87]]).T

# Sum weights related to the given input multiplied by
# the gradient related to the given neuron
dx0 = sum(weights[0]*dvalues[0])
dx1 = sum(weights[1]*dvalues[0])
dx2 = sum(weights[2]*dvalues[0])
dx3 = sum(weights[3]*dvalues[0])

dinputs = np.array([dx0, dx1, dx2, dx3])

print(dinputs)
```

```
>>>
[ 0.44 -0.38 -0.07  1.37]
```

You might already see where we are going with this — the sum of the multiplication of the elements is the dot product. We can achieve the same result by using the `np.dot` function. For this to be possible, we need to match the "inner" shapes and decide the first dimension of the result, which is the first dimension of the first parameter. We want the output of this calculation to be of the shape of the gradient from the subsequent function — recall that we have one partial derivative for each neuron and multiply it by the neuron's partial derivative with respect to its input. We then want to multiply each of these gradients with each of the partial derivatives that are related to this neuron's inputs, and we already noticed that they are rows. The dot product takes rows from the first argument and columns from the second to perform multiplication and sum; thus, we need to transpose the weights for this calculation:

```
import numpy as np

# Passed-in gradient from the next layer
# for the purpose of this example we're going to use
# a vector of 1s
dvalues = np.array([[1., 1., 1.]])

# We have 3 sets of weights - one set for each neuron
# we have 4 inputs, thus 4 weights
# recall that we keep weights transposed
weights = np.array([[0.2, 0.8, -0.5, 1],
                    [0.5, -0.91, 0.26, -0.5],
                    [-0.26, -0.27, 0.17, 0.87]]).T

# sum weights of given input
# and multiply by the passed-in gradient for this neuron
dinputs = np.dot(dvalues[0], weights.T)

print(dinputs)
```

```
>>>
[ 0.44 -0.38 -0.07  1.37]
```

We have to account for one more thing — a batch of samples. So far, we have been using a single sample responsible for a single gradient vector that is backpropagated between layers. The row vector that we created for `dvalues` is in preparation for a batch of data. With more samples, the layer will return a list of gradients, which we *almost* handle correctly for. Let's replace the singular gradient `dvalues[0]` with a full list of gradients, `dvalues`, and add more example gradients to this list:

```
import numpy as np

# Passed-in gradient from the next layer
# for the purpose of this example we're going to use
# an array of an incremental gradient values
dvalues = np.array([[1., 1., 1.],
                    [2., 2., 2.],
                    [3., 3., 3.]])

# We have 3 sets of weights - one set for each neuron
# we have 4 inputs, thus 4 weights
# recall that we keep weights transposed
weights = np.array([[0.2, 0.8, -0.5, 1],
                    [0.5, -0.91, 0.26, -0.5],
                    [-0.26, -0.27, 0.17, 0.87]]).T
```

```
# sum weights of given input
# and multiply by the passed-in gradient for this neuron
dinputs = np.dot(dvalues, weights.T)
print(dinputs)

>>>
[[ 0.44 -0.38 -0.07  1.37]
 [ 0.88 -0.76 -0.14  2.74]
 [ 1.32 -1.14 -0.21  4.11]]
```

Calculating the gradients with respect to weights is very similar, but, in this case, we're going to be using gradients to update the weights, so we need to match the shape of weights, not inputs. Since the derivative with respect to the weights equals inputs, weights are transposed, so we need to transpose inputs to receive the derivative of the neuron with respect to weights. Then we use these transposed inputs as the first parameter to the dot product — the dot product is going to multiply rows by inputs, where each row, as it is transposed, contains data for a given input for all of the samples, by the columns of `dvalues`. These columns are related to the outputs of singular neurons for all of the samples, so the result will contain an array with the shape of the weights, containing the gradients with respect to the inputs, multiplied with the incoming gradient for all of the samples in the batch:

```
import numpy as np

# Passed-in gradient from the next layer
# for the purpose of this example we're going to use
# an array of an incremental gradient values
dvalues = np.array([[1., 1., 1.],
                    [2., 2., 2.],
                    [3., 3., 3.]])

# We have 3 sets of inputs - samples
inputs = np.array([[1, 2, 3, 2.5],
                   [2., 5., -1., 2],
                   [-1.5, 2.7, 3.3, -0.8]])

# sum weights of given input
# and multiply by the passed-in gradient for this neuron
dweights = np.dot(inputs.T, dvalues)

print(dweights)


>>>
[[ 0.5  0.5  0.5]
 [20.1 20.1 20.1]
 [10.9 10.9 10.9]
 [ 4.1  4.1  4.1]]
```

This output's shape matches the shape of weights because we summed the inputs for each weight and then multiplied them by the input gradient. `dweights` is a gradient of the neuron function with respect to the weights.

For the biases and derivatives with respect to them, the derivatives come from the sum operation and always equal 1, multiplied by the incoming gradients to apply the chain rule. Since gradients are a list of gradients (a vector of gradients for each neuron for all samples), we just have to sum them with the neurons, column-wise, along axis 0.

```
import numpy as np

# Passed-in gradient from the next layer
# for the purpose of this example we're going to use
# an array of an incremental gradient values
dvalues = np.array([[1., 1., 1.],
                    [2., 2., 2.],
                    [3., 3., 3.]])

# One bias for each neuron
# biases are the row vector with a shape (1, neurons)
biases = np.array([[2, 3, 0.5]])

# dbiases - sum values, do this over samples (first axis), keepdims
# since this by default will produce a plain list -
# we explained this in the chapter 4
dbiases = np.sum(dvalues, axis=0, keepdims=True)

print(dbiases)
```

```
>>>
[[6. 6. 6.]]
```

`keepdims` lets us keep the gradient as a row vector — recall the shape of biases array.

The last thing to cover here is the derivative of the ReLU function. It equals *1* if the input is greater than *0* and *0* otherwise. The layer passes its outputs through the *ReLU()* activation during the forward pass. For the backward pass, *ReLU()* receives a gradient of the same shape. The derivative of the ReLU function will form an array of the same shape, filled with 1 when the related input is greater than 0, and 0 otherwise. To apply the chain rule, we need to multiply this array with the gradients of the following function:

```python
import numpy as np

# Example layer output
z = np.array([[1, 2, -3, -4],
              [2, -7, -1, 3],
              [-1, 2, 5, -1]])

dvalues = np.array([[1, 2, 3, 4],
                    [5, 6, 7, 8],
                    [9, 10, 11, 12]])

# ReLU activation's derivative
drelu = np.zeros_like(z)
drelu[z > 0] = 1

print(drelu)

# The chain rule
drelu *= dvalues

print(drelu)
```

```
>>>
[[1 1 0 0]
 [1 0 0 1]
 [0 1 1 0]]
[[ 1  2  0  0]
 [ 5  0  0  8]
 [ 0 10 11  0]]
```

To calculate the ReLU derivative, we created an array filled with zeros. `np.zeros_like` is a NumPy function that creates an array filled with zeros, with the shape of the array from its parameter, the `z` array in our case, which is an example output of the neuron. Following the *ReLU()* derivative, we then set the values related to the inputs greater than *0* as 1. We then print this table to see and compare it to the gradients. In the end, we multiply this array with the gradient of the subsequent function and print the result.

We can now simplify this operation. Since the *ReLU()* derivative array is filled with 1s, which do not change the values multiplied by them, and 0s that zero the multiplying value, this means that we can take the gradients of the subsequent function and set to 0 all of the values that correspond to the *ReLU()* input and are equal to or less than 0:

```
import numpy as np

# Example layer output
z = np.array([[1, 2, -3, -4],
              [2, -7, -1, 3],
              [-1, 2, 5, -1]])

dvalues = np.array([[1, 2, 3, 4],
                    [5, 6, 7, 8],
                    [9, 10, 11, 12]])

# ReLU activation's derivative
# with the chain rule applied
drelu = dvalues.copy()
drelu[z <= 0] = 0

print(drelu)

>>>
[[ 1  2  0  0]
 [ 5  0  0  8]
 [ 0 10 11  0]]
```

The copy of `dvalues` ensures that we don't modify it during the ReLU derivative calculation.

Let's combine the forward and backward pass of a single neuron with a full layer and batch-based partial derivatives. We'll minimize ReLU's output, once again, only for this example:

```
import numpy as np

# Passed-in gradient from the next layer
# for the purpose of this example we're going to use
# an array of an incremental gradient values
dvalues = np.array([[1., 1., 1.],
                    [2., 2., 2.],
                    [3., 3., 3.]])

# We have 3 sets of inputs - samples
inputs = np.array([[1, 2, 3, 2.5],
                   [2., 5., -1., 2],
                   [-1.5, 2.7, 3.3, -0.8]])
# We have 3 sets of weights - one set for each neuron
# we have 4 inputs, thus 4 weights
# recall that we keep weights transposed
weights = np.array([[0.2, 0.8, -0.5, 1],
                    [0.5, -0.91, 0.26, -0.5],
                    [-0.26, -0.27, 0.17, 0.87]]).T
```

```
# One bias for each neuron
# biases are the row vector with a shape (1, neurons)
biases = np.array([[2, 3, 0.5]])

# Forward pass
layer_outputs = np.dot(inputs, weights) + biases  # Dense layer
relu_outputs = np.maximum(0, layer_outputs)  # ReLU activation

# Let's optimize and test backpropagation here
# ReLU activation - simulates derivative with respect to input values
# from next layer passed to current layer during backpropagation
drelu = relu_outputs.copy()
drelu[layer_outputs <= 0] = 0

# Dense layer
# dinputs - multiply by weights
dinputs = np.dot(drelu, weights.T)
# dweights - multiply by inputs
dweights = np.dot(inputs.T, drelu)
# dbiases - sum values, do this over samples (first axis), keepdims
# since this by default will produce a plain list -
# we explained this in the chapter 4
dbiases = np.sum(drelu, axis=0, keepdims=True)

# Update parameters
weights += -0.001 * dweights
biases += -0.001 * dbiases

print(weights)
print(biases)

>>>
[[ 0.179515   0.5003665 -0.262746 ]
 [ 0.742093  -0.9152577 -0.2758402]
 [-0.510153   0.2529017  0.1629592]
 [ 0.971328  -0.5021842  0.8636583]]
[[1.98489  2.997739 0.497389]]
```

In this code, we replaced the plain Python functions with NumPy variants, created example data, calculated the forward and backward passes, and updated the parameters. Now we will update the dense layer and ReLU activation code with a `backward` method (for backpropagation), which we'll call during the backpropagation phase of our model.

```
# Dense layer
class Layer_Dense:

    # Layer initialization
    def __init__(self, inputs, neurons):
        self.weights = 0.01 * np.random.randn(inputs, neurons)
        self.biases = np.zeros((1, neurons))

    # Forward pass
    def forward(self, inputs):
        self.output = np.dot(inputs, self.weights) + self.biases


# ReLU activation
class Activation_ReLU:

    # Forward pass
    def forward(self, inputs):
        self.output = np.maximum(0, inputs)
```

During the `forward` method for our `Layer_Dense` class, we will want to remember what the inputs were (recall that we'll need them when calculating the partial derivative with respect to weights during backpropagation), which can be easily implemented using an object property (`self.inputs`):

```
# Dense layer
class Layer_Dense:
    ...
    # Forward pass
    def forward(self, inputs):
        ...
        self.inputs = inputs
```

Next, we will add our backward pass (backpropagation) code that we developed previously into a new method in the layer class, which we'll call `backward`:

```
class Layer_Dense:
    ...
    # Backward pass
    def backward(self, dvalues):
        # Gradients on parameters
        self.dweights = np.dot(self.inputs.T, dvalues)
        self.dbiases = np.sum(dvalues, axis=0, keepdims=True)
        # Gradient on values
        self.dinputs = np.dot(dvalues, self.weights.T)
```

We then do the same for our ReLU class:

```
# ReLU activation
class Activation_ReLU:

    # Forward pass
    def forward(self, inputs):
        # Remember input values
        self.inputs = inputs
        self.output = np.maximum(0, inputs)

    # Backward pass
    def backward(self, dvalues):
        # Since we need to modify the original variable,
        # let's make a copy of the values first
        self.dinputs = dvalues.copy()

        # Zero gradient where input values were negative
        self.dinputs[self.inputs <= 0] = 0
```

By this point, we've covered everything we need to perform backpropagation, except for the derivative of the Softmax activation function and the derivative of the cross-entropy loss function.

Categorical Cross-Entropy loss derivative

If you are not interested in the mathematical derivation of the Categorical Cross-Entropy loss, feel free to skip to the code implementation, as derivatives are known for common loss functions, and you won't necessarily need to know how to solve them. It is a good exercise if you plan to create custom loss functions, though.

As we learned in chapter 5, the Categorical Cross-Entropy loss function's formula is:

$$L_i = -log(\hat{y}_{i,k}) \quad where\ \mathbf{k}\ is\ an\ index\ of\ "true"\ probability$$

Where L_i denotes sample loss value, *i* — *i*-th sample in a set, *k* — index of the target label (ground-true label), *y* — target values and *y-hat* — predicted values.

This formula is convenient when calculating the loss value itself, as all we need is the output of the Softmax activation function at the index of the correct class. For the purpose of the derivative calculation, we'll use the full equation mentioned back in chapter 5:

$$L_i = -\sum_j y_{i,j} log(\hat{y}_{i,j})$$

Where L_i denotes sample loss value, *i* — *i*-th sample in a set, *j* — label/output index, *y* — target values and *y-hat* — predicted values.

We'll use this full function because our current goal is to calculate the gradient, which is composed of the partial derivatives of the loss function with respect to each of its inputs (being the outputs of the Softmax activation function). This means that we cannot use the equation, which takes just the value at the index of the correct class (the first equation above). To calculate partial derivatives with respect to each of the inputs, we need an equation that takes all of them as parameters, thus the choice to use the full equation.

First, let's define the gradient equation:

$$\frac{\partial L_i}{\partial \hat{y}_{i,j}} = \frac{\partial}{\partial \hat{y}_{i,j}}[-\sum_j y_{i,j} log(\hat{y}_{i,j})] =$$

We defined the equation here as the partial derivative of the loss function with respect to each of its inputs. We already learned that the derivative of the sum equals the sum of the derivatives. We also learned that we can move constants. An example is $y_{i,j}$, as it is not what we are calculating the derivative with respect to. Let's apply these transforms:

$$= -\sum_j y_{i,j} \cdot \frac{\partial}{\partial \hat{y}_{i,j}} log(\hat{y}_{i,j}) =$$

Now we have to solve the derivative of the logarithmic function, which is the reciprocal of its parameter, multiplied (using the chain rule) by the partial derivative of this parameter — using prime (also called Lagrange's) notation:

$$f(x) = log(h(x)) \quad \rightarrow \quad f'(x) = \frac{1}{h(x)} \cdot h'(x)$$

We can solve it further (using Leibniz's notation in this case):

$$f(x) = log(x) \quad \rightarrow \quad \frac{d}{dx} f(x) = \frac{d}{dx} log(x) = \frac{1}{x} \cdot \frac{d}{dx} x = \frac{1}{x} \cdot 1 = \frac{1}{x}$$

Let's apply this derivative:

$$= -\sum_j y_{i,j} \cdot \frac{1}{\hat{y}_{i,j}} \cdot \frac{\partial}{\partial \hat{y}_{i,j}} \hat{y}_{i,j} =$$

The partial derivative of a value with respect to this value equals 1:

$$= -\sum_j y_{i,j} \cdot \frac{1}{\hat{y}_{i,j}} \cdot 1 = -\sum_j \frac{y_{i,j}}{\hat{y}_{i,j}} =$$

Since we are calculating the partial derivative with respect to the *y* given *j*, the sum is being performed over a single element and can be omitted:

$$= -\frac{y_{i,j}}{\hat{y}_{i,j}}$$

Full solution:

$$\frac{\partial L_i}{\partial \hat{y}_{i,j}} = \frac{\partial}{\partial \hat{y}_{i,j}}[-\sum_j y_{i,j} log(\hat{y}_{i,j})] = -\sum_j y_{i,j} \cdot \frac{\partial}{\partial \hat{y}_{i,j}} log(\hat{y}_{i,j}) =$$

$$= -\sum_j y_{i,j} \cdot \frac{1}{\hat{y}_{i,j}} \cdot \frac{\partial}{\partial \hat{y}_{i,j}} \hat{y}_{i,j} = -\sum_j y_{i,j} \cdot \frac{1}{\hat{y}_{i,j}} \cdot 1 = -\sum_j \frac{y_{i,j}}{\hat{y}_{i,j}} = -\frac{y_{i,j}}{\hat{y}_{i,j}}$$

The derivative of this loss function with respect to its inputs (predicted values at the i-th sample, since we are interested in a gradient with respect to the predicted values) equals the negative ground-truth vector, divided by the vector of the predicted values (which is also the output vector of the softmax function).

Categorical Cross-Entropy loss derivative code implementation

Since we derived this equation and have found that it solves to a simple division operation of 2 values, we know that, with NumPy, we can extend this operation to the sample-wise vectors of ground truth and predicted values, and further to the batch-wise arrays of them. From the coding perspective, we need to add a backward method to the Loss_CategoricalCrossentropy class. We need to pass the array of predictions and the array of true values into it and calculate the negated division of them:

```
# Cross-entropy loss
class Loss_CategoricalCrossentropy(Loss):
    ...
    # Backward pass
    def backward(self, dvalues, y_true):

        # Number of samples
        samples = len(dvalues)
        # Number of labels in every sample
        # We'll use the first sample to count them
        labels = len(dvalues[0])

        # If labels are sparse, turn them into one-hot vector
        if len(y_true.shape) == 1:
            y_true = np.eye(labels)[y_true]

        # Calculate gradient
        self.dinputs = -y_true / dvalues
        # Normalize gradient
        self.dinputs = self.dinputs / samples
```

Along with the partial derivative calculation, we are performing two additional operations. First, we're turning numerical labels into one-hot encoded vectors — prior to this, we need to check how many dimensions y_true consists of. If the shape of the labels returns a single dimension

(which means that they are shaped like a list and not like an array), they consist of discrete numbers and need to be converted to a list of one-hot encoded vectors — a two-dimensional array. If that's the case, we need to turn them into one-hot encoded vectors. We'll use the `np.eye` method which, given a number, *n*, returns an *n*x*n* array filled with ones on the diagonal and zeros everywhere else. For example:

```
import numpy as np

print(np.eye(5))

>>>
array([[1., 0., 0., 0., 0.],
       [0., 1., 0., 0., 0.],
       [0., 0., 1., 0., 0.],
       [0., 0., 0., 1., 0.],
       [0., 0., 0., 0., 1.]])
```

We can then index this table with the numerical label to get the one-hot encoded vector that represents it:

```
print(np.eye(5)[1])

>>>
array([0., 1., 0., 0., 0.])


print(np.eye(5)[4])

>>>
array([0., 0., 0., 0., 1.])
```

If *y_true* is already one-hot encoded, we do not perform this step.

The second operation is the gradient normalization. As we'll learn in the next chapter, optimizers sum all of the gradients related to each weight and bias before multiplying them by the learning rate (or some other factor). What this means, in our case, is that the more samples we have in a dataset, the more gradient sets we'll receive at this step, and the bigger this sum will become. As a consequence, we'll have to adjust the learning rate according to each set of samples. To solve this problem, we can divide all of the gradients by the number of samples. A sum of elements divided by a count of them is their mean value (and, as we mentioned, the optimizer will perform the sum) — this way, we'll effectively normalize the gradients and make their sum's magnitude invariant to the number of samples.

Softmax activation derivative

The next calculation that we need to perform is the partial derivative of the Softmax function, which is a bit more complicated task than the derivative of the Categorical Cross-Entropy loss. Let's remind ourselves of the equation of the Softmax activation function and define the derivative:

$$S_{i,j} = \frac{e^{z_{i,j}}}{\sum_{l=1}^{L} e^{z_{i,l}}} \quad \rightarrow \quad \frac{\partial S_{i,j}}{\partial z_{i,k}} = \frac{\partial \frac{e^{z_{i,j}}}{\sum_{l=1}^{L} e^{z_{i,l}}}}{\partial z_{i,k}}$$

Where $S_{i,j}$ denotes j-th Softmax's output of i-th sample, z — input array which is a list of input vectors (output vectors from the previous layer), $z_{i,j}$ — j-th Softmax's input of i-th sample, L — number of inputs, $z_{i,k}$ — k-th Softmax's input of i-th sample.

As we described in chapter 4, the Softmax function equals the exponentiated input divided by the sum of all exponentiated inputs. In other words, we need to exponentiate all of the values first, then divide each of them by the sum of all of them to perform the normalization. Each input to the Softmax impacts each of the outputs, and we need to calculate the partial derivative of each output with respect to each input. From the programming side of things, if we calculate the impact of one list on the other list, we'll receive a matrix of values as a result. That's exactly what we'll calculate here — we'll calculate the **Jacobian matrix** (which we'll explain later) of the vectors, which we'll dive deeper into soon.

To calculate this derivative, we need to first define the derivative of the division operation:

$$f(x) = \frac{g(x)}{h(x)} \quad \rightarrow \quad f'(x) = \frac{g'(x) \cdot h(x) - g(x) \cdot h'(x)}{[h(x)]^2}$$

In order to calculate the derivative of the division operation, we need to take the derivative of the numerator multiplied by the denominator, subtract the numerator multiplied by the derivative of the denominator from it, and then divide the result by the squared denominator.

We can now start solving the derivative:

$$\frac{\partial S_{i,j}}{\partial z_{i,k}} = \frac{\partial \frac{e^{z_{i,j}}}{\sum_{l=1}^{L} e^{z_{i,l}}}}{\partial z_{i,k}} =$$

Let's apply the derivative of the division operation:

$$= \frac{\frac{\partial}{\partial z_{i,k}} e^{z_{i,j}} \cdot \sum_{l=1}^{L} e^{z_{i,l}} - e^{z_{i,j}} \cdot \frac{\partial}{\partial z_{i,k}} \sum_{l=1}^{L} e^{z_{i,l}}}{[\sum_{l=1}^{L} e^{z_{i,l}}]^2} =$$

At this step, we have two partial derivatives present in the equation. For the one on the right side of the numerator (right side of the subtraction operator):

$$\frac{\partial}{\partial z_{i,k}} \sum_{l=1}^{L} e^{z_{i,l}}$$

We need to calculate the derivative of the sum of the constant,*e* (Euler's number), raised to power $z_{i,l}$ (where *l* denotes consecutive indices from *1* to the number of the Softmax outputs — *L*) with respect to the $z_{i,k}$. The derivative of the sum operation is the sum of derivatives, and the derivative of the constant *e* raised to power *n* (e^n) with respect to *n* equals e^n:

$$\frac{d}{dn} e^n = e^n \cdot \frac{d}{dn} n = e^n \cdot 1 = e^n$$

It is a special case when the derivative of an exponential function equals this exponential function itself, as its exponent is exactly what we are deriving with respect to, thus its derivative equals *1*. We also know that the range *1...L* contains *k* (*k* is one of the indices from this range) exactly once and then, in this case, the derivative is going to equal *e* to the power of the $z_{i,k}$ (as *j* equals *k*) and *0* otherwise (when *j* does not equal *k* as $z_{i,l}$ won't contain $z_{i,k}$ and will be treated as a constant — The derivative of the constant equals *0*):

$$\frac{\partial}{\partial z_{i,k}} \sum_{l=1}^{L} e^{z_{i,l}} = \frac{\partial}{\partial z_{i,k}} e^{z_{i,1}} + \frac{\partial}{\partial z_{i,k}} e^{z_{i,2}} + \ldots + \frac{\partial}{\partial z_{i,k}} e^{z_{i,k}} + \ldots + \frac{\partial}{\partial z_{i,k}} e^{z_{i,L-1}} + \frac{\partial}{\partial z_{i,k}} e^{z_{i,L}}$$

$$= 0 + 0 + \ldots + e^{z_{i,k}} + \ldots + 0 + 0 = e^{z_{i,k}}$$

The derivative on the left side of the subtraction operator in the denominator is a slightly different case:

$$\frac{\partial}{\partial z_{i,k}} e^{z_{i,j}}$$

It does not contain the sum over all of the elements like the derivative we solved moments ago, so it can become either *0* if $j \neq k$ or *e* to the power of the $z_{i,j}$ if $j=k$. That means, starting from this step, we need to calculate the derivatives separately for both cases. Let's start with $j=k$.

In the case of $j=k$, the derivative on the left side is going to equal *e* to the power of the $z_{i,j}$ and the derivative on the right solves to the same value in both cases. Let's substitute them:

$$= \frac{e^{z_{i,j}} \cdot \sum_{l=1}^{L} e^{z_{i,l}} - e^{z_{i,j}} \cdot e^{z_{i,k}}}{[\sum_{l=1}^{L} e^{z_{i,l}}]^2} =$$

The numerator contains the constant *e* to the power of $z_{i,j}$ in both the minuend (the value we are subtracting from) and subtrahend (the value we are subtracting from the minuend) of the subtraction operation. Because of this, we can regroup the numerator to contain this value multiplied by the subtraction of their current multipliers. We can also write the denominator as a multiplication of the value instead of using the power of *2*:

$$= \frac{e^{z_{i,j}} \cdot (\sum_{l=1}^{L} e^{z_{i,l}} - e^{z_{i,k}})}{\sum_{l=1}^{L} e^{z_{i,l}} \cdot \sum_{l=1}^{L} e^{z_{i,l}}} =$$

Then let's split the whole equation into 2 parts:

$$= \frac{e^{z_{i,j}}}{\sum_{l=1}^{L} e^{z_{i,l}}} \cdot \frac{\sum_{l=1}^{L} e^{z_{i,l}} - e^{z_{i,k}}}{\sum_{l=1}^{L} e^{z_{i,l}}} =$$

We moved *e* from the numerator and the sum from the denominator to its own fraction, and the content of the parentheses in the numerator, and the other sum from the denominator as another fraction, both joined by the multiplication operation. Now we can further split the "right" fraction into two separate fractions:

$$= \frac{e^{z_{i,j}}}{\sum_{l=1}^{L} e^{z_{i,l}}} \cdot (\frac{\sum_{l=1}^{L} e^{z_{i,l}}}{\sum_{l=1}^{L} e^{z_{i,l}}} - \frac{e^{z_{i,k}}}{\sum_{l=1}^{L} e^{z_{i,l}}}) =$$

In this case, as it's a subtraction operation, we separated both values from the numerator, dividing them both by the denominator and applying the subtraction operation between new fractions. If we

look closely, the "left" fraction turns into the Softmax function's equation, as well as the "right" one, with the middle fraction solving to *1* as the numerator and the denominator are the same values:

$$= S_{i,j} \cdot (1 - S_{i,k})$$

Note that the "left" Softmax function carries the $_j$ parameter, and the "right" one $_k$ — both came from their numerators, respectively.

Full solution:

$$\frac{\partial S_{i,j}}{\partial z_{i,k}} = \frac{\partial \frac{e^{z_{i,j}}}{\sum_{l=1}^{L} e^{z_{i,l}}}}{\partial z_{i,k}} = \frac{\frac{\partial}{\partial z_{i,k}} e^{z_{i,j}} \cdot \sum_{l=1}^{L} e^{z_{i,l}} - e^{z_{i,j}} \cdot \frac{\partial}{\partial z_{i,k}} \sum_{l=1}^{L} e^{z_{i,l}}}{[\sum_{l=1}^{L} e^{z_{i,l}}]^2} =$$

$$\frac{e^{z_{i,j}} \cdot \sum_{l=1}^{L} e^{z_{i,l}} - e^{z_{i,j}} \cdot e^{z_{i,k}}}{[\sum_{l=1}^{L} e^{z_{i,l}}]^2} = \frac{e^{z_{i,j}} \cdot (\sum_{l=1}^{L} e^{z_{i,l}} - e^{z_{i,k}})}{\sum_{l=1}^{L} e^{z_{i,l}} \cdot \sum_{l=1}^{L} e^{z_{i,l}}} =$$

$$\frac{e^{z_{i,j}}}{\sum_{l=1}^{L} e^{z_{i,l}}} \cdot \frac{\sum_{l=1}^{L} e^{z_{i,l}} - e^{z_{i,k}}}{\sum_{l=1}^{L} e^{z_{i,l}}} = \frac{e^{z_{i,j}}}{\sum_{l=1}^{L} e^{z_{i,l}}} \cdot (\frac{\sum_{l=1}^{L} e^{z_{i,l}}}{\sum_{l=1}^{L} e^{z_{i,l}}} - \frac{e^{z_{i,k}}}{\sum_{l=1}^{L} e^{z_{i,l}}}) =$$

$$S_{i,j} \cdot (1 - S_{i,k})$$

Now we have to go back and solve the derivative in the case of *j≠k*. In this case, the "left" derivative of the original equation solves to *0* as the whole expression is treated as a constant:

$$= \frac{0 \cdot \sum_{l=1}^{L} e^{z_{i,l}} - e^{z_{i,j}} \cdot e^{z_{i,k}}}{[\sum_{l=1}^{L} e^{z_{i,l}}]^2} =$$

The difference is that now the whole subtrahend solves to *0*, leaving us with just the minuend in the numerator:

$$= \frac{-e^{z_{i,j}} \cdot e^{z_{i,k}}}{[\sum_{l=1}^{L} e^{z_{i,l}}]^2} =$$

Now, exactly like before, we can write the denominator as the multiplication of the values instead of using the power of *2*:

$$= \frac{-e^{z_{i,j}} \cdot e^{z_{i,k}}}{\sum_{l=1}^{L} e^{z_{i,l}} \cdot \sum_{l=1}^{L} e^{z_{i,l}}} =$$

That lets us to split this fraction into 2 fractions, using the multiplication operation:

$$= -\frac{e^{z_{i,j}}}{\sum_{l=1}^{L} e^{z_{i,l}}} \cdot \frac{e^{z_{i,k}}}{\sum_{l=1}^{L} e^{z_{i,l}}} =$$

Now both fractions represent the Softmax function:

$$= -S_{i,j} \cdot S_{i,k}$$

Note that the left Softmax function carries the $_j$ parameter, and the "right" one has $_k$ — both came from their numerators, respectively.

Full solution:

$$\frac{\partial S_{i,j}}{\partial z_{i,k}} = \frac{\partial \frac{e^{z_{i,j}}}{\sum_{l=1}^{L} e^{z_{i,l}}}}{\partial z_{i,k}} = \frac{\frac{\partial}{\partial z_{i,k}} e^{z_{i,j}} \cdot \sum_{l=1}^{L} e^{z_{i,l}} - e^{z_{i,j}} \cdot \frac{\partial}{\partial z_{i,k}} \sum_{l=1}^{L} e^{z_{i,l}}}{[\sum_{l=1}^{L} e^{z_{i,l}}]^2} =$$

$$\frac{0 \cdot \sum_{l=1}^{L} e^{z_{i,l}} - e^{z_{i,j}} \cdot e^{z_{i,k}}}{[\sum_{l=1}^{L} e^{z_{i,l}}]^2} = \frac{-e^{z_{i,j}} \cdot e^{z_{i,k}}}{[\sum_{l=1}^{L} e^{z_{i,l}}]^2} = \frac{-e^{z_{i,j}} \cdot e^{z_{i,k}}}{\sum_{l=1}^{L} e^{z_{i,l}} \cdot \sum_{l=1}^{L} e^{z_{i,l}}} =$$

$$-\frac{e^{z_{i,j}}}{\sum_{l=1}^{L} e^{z_{i,l}}} \cdot \frac{e^{z_{i,k}}}{\sum_{l=1}^{L} e^{z_{i,l}}} = -S_{i,j} \cdot S_{i,k}$$

As a summary, the solution of the derivative of the Softmax function with respect to its inputs is:

$$\frac{\partial S_{i,j}}{\partial z_{i,k}} = \begin{cases} S_{i,j} \cdot (1 - S_{i,k}) & j = k \\ -S_{i,j} \cdot S_{i,k} & j \neq k \end{cases}$$

That's not the end of the calculation that we can perform here. When left in this form, we'll have 2 separate equations to code and use in different cases, which isn't very convenient for the speed of calculations. We can, however, further morph the result of the second case of the derivative:

$$-S_{i,j} \cdot S_{i,k} = S_{i,j} \cdot (-S_{i,k}) = S_{i,j} \cdot (0 - S_{i,k})$$

In the first step, we moved the second Softmax along the minus sign into the brackets so we can add a zero inside of them and right before this value. That does not change the solution, but now:

$$\frac{\partial S_{i,j}}{\partial z_{i,k}} = \begin{cases} S_{i,j} \cdot (1 - S_{i,k}) & j = k \\ S_{i,j} \cdot (0 - S_{i,k}) & j \neq k \end{cases}$$

Both solutions look very similar, they differ only in a single value. Conveniently, there exists **Kronecker delta** function (which we'll explain soon) whose equation is:

$$\delta_{i,j} = \begin{cases} 1 & i = j \\ 0 & i \neq j \end{cases}$$

We can apply it here, simplifying our equation further to:

$$\frac{\partial S_{i,j}}{\partial z_{i,k}} = S_{i,j} \cdot (\delta_{j,k} - S_{i,k})$$

That's the final math solution to the derivative of the Softmax function's outputs with respect to each of its inputs. To make it a little bit easier to implement in Python using NumPy, let's transform the equation for the last time:

$$\frac{\partial S_{i,j}}{\partial z_{i,k}} = S_{i,j} \cdot (\delta_{j,k} - S_{i,k}) = S_{i,j}\delta_{j,k} - S_{i,j}S_{i,k}$$

We basically multiplied $S_{i,j}$ by both sides of the subtraction operation from the parentheses.

Softmax activation derivative code implementation

This lets us code the solution using just two NumPy functions, which we'll explain now step by step:

Let's make up a single sample:

```
softmax_output = [0.7, 0.1, 0.2]
```

And shape it as a list of samples:

```
import numpy as np

softmax_output = [0.7, 0.1, 0.2]

softmax_output = np.array(softmax_output).reshape(-1, 1)
print(softmax_output)

>>>
array([[0.7],
       [0.1],
       [0.2]])
```

The left side of the equation is Softmax's output multiplied by the Kronecker delta. The Kronecker delta equals *1* when both inputs are equal, and *0* otherwise. If we visualize this as an array, we'll have an array of zeros with ones on the diagonal — you might remember that we already have implemented such a solution using the `np.eye` method:

```
print(np.eye(softmax_output.shape[0]))

>>>
array([[1., 0., 0.],
       [0., 1., 0.],
       [0., 0., 1.]])
```

Now we'll do the multiplication of both of the values from the equation part:

```
print(softmax_output * np.eye(softmax_output.shape[0]))

>>>
array([[0.7, 0. , 0. ],
       [0. , 0.1, 0. ],
       [0. , 0. , 0.2]])
```

It turns out that we can gain some speed by replacing this by the `np.diagflat` method call, which computes the same solution — the diagflat method creates an array using an input vector as the diagonal:

```
print(np.diagflat(softmax_output))

>>>
array([[0.7, 0. , 0. ],
       [0. , 0.1, 0. ],
       [0. , 0. , 0.2]])
```

The other part of the equation is $S_{i,j}S_{i,k}$ — the multiplication of the Softmax outputs, iterating over the $_j$ and $_k$ indices respectively. Since, for each sample (the $_i$ index), we'll have to multiply the values from the Softmax function's output (in all of the combinations), we can use the dot product operation. For this, we'll just have to transpose the second argument to get its row vector form (as described in chapter 2):

```
print(np.dot(softmax_output, softmax_output.T))

>>>
array([[0.49, 0.07, 0.14],
       [0.07, 0.01, 0.02],
       [0.14, 0.02, 0.04]])
```

Finally, we can perform the subtraction of both arrays (following the equation):

```
print(np.diagflat(softmax_output) -
      np.dot(softmax_output, softmax_output.T))

>>>
array([[ 0.21, -0.07, -0.14],
       [-0.07,  0.09, -0.02],
       [-0.14, -0.02,  0.16]])
```

The matrix result of the equation and the array solution provided by the code is called the **Jacobian matrix**. In our case, the Jacobian matrix is an array of partial derivatives in all of the combinations of both input vectors. Remember, we are calculating the partial derivatives of every output of the Softmax function with respect to each input separately. We do this because each input influences each output due to the normalization process, which takes the sum of all the exponentiated inputs. The result of this operation, performed on a batch of samples, is a list of the Jacobian matrices, which effectively forms a 3D matrix — you can visualize it as a column whose levels are Jacobian matrices being the sample-wise gradient of the Softmax function.

This raises a question — if sample-wise gradients are the Jacobian matrices, how do we perform the chain rule with the gradient back-propagated from the loss function, since it's a vector for each sample? Also, what do we do with the fact that the previous layer, which is the Dense layer, will expect the gradients to be a 2D array? Currently, we have a 3D array of the partial derivatives — a list of the Jacobian matrices. The derivative of the Softmax function with respect to any of its inputs returns a vector of partial derivatives (a row from the Jacobian matrix), as this input influences all the outputs, thus also influencing the partial derivative for each of them. We need to sum the values from these vectors so that each of the inputs for each of the samples will return a single partial derivative value instead. Because each input influences all of the outputs, the returned vector of the partial derivatives has to be summed up for the final partial derivative with respect to this input. We can perform this operation on each of the Jacobian matrices directly, applying the chain rule at the same time (applying the gradient from the loss function) using `np.dot()` — For each sample, it'll take the row from the Jacobian matrix and multiply it by the corresponding value from the loss function's gradient. As a result, the dot product of each of these vectors and values will return a singular value, forming a vector of the partial derivatives sample-wise and a 2D array (a list of the resulting vectors) batch-wise.

Let's code the solution:

```
# Softmax activation
class Activation_Softmax:
    ...
    # Backward pass
    def backward(self, dvalues):

        # Create uninitialized array
        self.dinputs = np.empty_like(dvalues)

        # Enumerate outputs and gradients
        for index, (single_output, single_dvalues) in \
                enumerate(zip(self.output, dvalues)):
            # Flatten output array
            single_output = single_output.reshape(-1, 1)
            # Calculate Jacobian matrix of the output
            jacobian_matrix = np.diagflat(single_output) - \
                              np.dot(single_output, single_output.T)
            # Calculate sample-wise gradient
            # and add it to the array of sample gradients
            self.dinputs[index] = np.dot(jacobian_matrix,
                                         single_dvalues)
```

First, we created an empty array (which will become the resulting gradient array) with the same shape as the gradients that we're receiving to apply the chain rule. The `np.empty_like` method creates an empty and uninitialized array. Uninitialized means that we can expect it to contain meaningless values, but we'll set all of them shortly anyway, so there's no need for initialization (for example, with zeros using `np.zeros()` instead). In the next step, we're going to iterate sample-wise over pairs of the outputs and gradients, calculating the partial derivatives as described earlier and calculating the final product (applying the chain rule) of the Jacobian matrix and gradient vector (from the passed-in gradient array), storing the resulting vector as a row in the dinput array. We're going to store each vector in each row while iterating, forming the output array.

Common Categorical Cross-Entropy loss and Softmax activation derivative

At the moment, we have calculated the partial derivatives of the Categorical Cross-Entropy loss and Softmax activation functions, and we can finally use them, but there is still one more step that we can perform to speed the calculations up. Different books and tutorials usually mention the derivative of the loss function with respect to the Softmax inputs, or even weight and biases of the output layer directly and don't go into the details of the partial derivatives of these functions separately. This is partially because the derivatives of both functions combine to solve a simple equation — the whole code implementation is simpler and faster to execute. When we look at our current code, we perform multiple operations to calculate the gradients and even include a loop in the backward step of the activation function.

Let's apply the chain rule to calculate the partial derivative of the Categorical Cross-Entropy loss function with respect to the Softmax function inputs. First, let's define this derivative by applying the chain rule:

$$\frac{\partial L_i}{\partial z_{i,k}} = \frac{\partial L_i}{\partial \hat{y}_{i,j}} \cdot \frac{\partial S_{i,j}}{\partial z_{i,k}} =$$

This partial derivative equals the partial derivative of the loss function with respect to its inputs, multiplied (using the chain rule) by the partial derivative of the activation function with respect to its inputs. Now we need to systematize semantics — we know that the inputs to the loss function, $y\text{-}hat_{i,j}$, are the outputs of the activation function, $S_{i,j}$:

$$\hat{y}_{i,j} = S_{i,j}$$

That means that we can update the equation to the form of:

$$= \frac{\partial L_i}{\partial \hat{y}_{i,j}} \cdot \frac{\partial \hat{y}_{i,j}}{\partial z_{i,k}} =$$

Now we can substitute the equation for the partial derivative of the Categorical Cross-Entropy function, but, since we are calculating the partial derivative with respect to the Softmax inputs, we'll use the one containing the sum operator over all of the outputs — it will soon become clear why. The derivative:

$$\frac{\partial L_i}{\partial \hat{y}_{i,j}} = -\sum_j \frac{y_{i,j}}{\hat{y}_{i,j}}$$

After substitution to the combined derivative's equation:

$$= -\sum_j \frac{y_{i,j}}{\hat{y}_{i,j}} \cdot \frac{\partial \hat{y}_{i,j}}{\partial z_{i,k}} =$$

Now, as we calculated before, the partial derivative of the Softmax activation, before applying Kronecker delta to it:

$$\frac{\partial S_{i,j}}{\partial z_{i,k}} = \begin{cases} S_{i,j} \cdot (1 - S_{i,k}) & j = k \\ -S_{i,j} \cdot S_{i,k} & j \neq k \end{cases}$$

Let's actually do the substitution of the $S_{i,j}$ with *y-hat*$_{i,j}$ here as well:

$$\frac{\partial \hat{y}_{i,j}}{\partial z_{i,k}} = \begin{cases} \hat{y}_{i,j} \cdot (1 - \hat{y}_{i,k}) & j = k \\ -\hat{y}_{i,j} \cdot \hat{y}_{i,k} & j \neq k \end{cases}$$

The solution is different depending on if $j=k$ or $j \neq k$. To handle for this situation, we have to split the current partial derivative following these cases — when they both match and when they do not:

$$-\sum_j \frac{y_{i,j}}{\hat{y}_{i,j}} \cdot \frac{\partial \hat{y}_{i,j}}{\partial z_{i,k}} = -\frac{y_{i,k}}{\hat{y}_{i,k}} \cdot \frac{\partial \hat{y}_{i,k}}{\partial z_{i,k}} - \sum_{j \neq k} \frac{y_{i,j}}{\hat{y}_{i,j}} \cdot \frac{\partial \hat{y}_{i,j}}{\partial z_{i,k}}$$

For the $j \neq k$ case, we just updated the sum operator to exclude k and that's the only change:

$$-\sum_{j \neq k} \frac{y_{i,j}}{\hat{y}_{i,j}} \cdot \frac{\partial \hat{y}_{i,j}}{\partial z_{i,k}}$$

For the $j=k$ case, we do not need the sum operator as it will sum only one element, of index k. For

the same reason, we also replace *j* indices with *k*:

$$-\frac{y_{i,k}}{\hat{y}_{i,k}} \cdot \frac{\partial \hat{y}_{i,k}}{\partial z_{i,k}}$$

Back to the main equation:

$$= -\frac{y_{i,k}}{\hat{y}_{i,k}} \cdot \frac{\partial \hat{y}_{i,k}}{\partial z_{i,k}} - \sum_{j \neq k} \frac{y_{i,j}}{\hat{y}_{i,j}} \cdot \frac{\partial \hat{y}_{i,j}}{\partial z_{i,k}} =$$

Now we can substitute the partial derivatives of the activation function for both cases with the newly-defined solutions:

$$= -\frac{y_{i,k}}{\hat{y}_{i,k}} \cdot \hat{y}_{i,k} \cdot (1 - \hat{y}_{i,k}) - \sum_{j \neq k} \frac{y_{i,j}}{\hat{y}_{i,j}} (-\hat{y}_{i,j} \hat{y}_{i,k}) =$$

We can cancel out the *y-hat*$_{i,k}$ from both sides of the subtraction in the equation — both contain it as part of the multiplication operations and in their denominators. Then on the "right" side of the equation, we can replace 2 minus signs with the plus one and remove the parentheses:

$$= -y_{i,k} \cdot (1 - \hat{y}_{i,k}) + \sum_{j \neq k} y_{i,j} \hat{y}_{i,k} =$$

Now let's multiply the -$y_{i,k}$ with the content of the parentheses on the "left" side of the equation:

$$= -y_{i,k} + y_{i,k} \hat{y}_{i,k} + \sum_{j \neq k} y_{i,j} \hat{y}_{i,k} =$$

Now let's look at the sum operation — it adds up $y_{i,j}$*y-hat*$_{i,k}$ over all possible values of index *j* except for when it equals *k*. Then, on the left of this part of the equation, we have $y_{i,k}$*y-hat*$_{i,k}$, which contains $y_{i,k}$ — the exact element that is excluded from the sum. We can then join both expressions:

$$= -y_{i,k} + \sum_{j} y_{i,j} \hat{y}_{i,k} =$$

Now the sum operator iterates over all of the possible values of *j* and, since we know that $y_{i,j}$ for each *i* is the one-hot encoded vector of ground-truth values, the sum of all of its elements equals *1*. In other words, following the earlier explanation in this chapter — this sum will multiply *0* by the $y\text{-}hat_{i,k}$ except for a single situation, the true label, where it'll multiply *1* by this value. We can then simplify it further to:

$$= -y_{i,k} + \hat{y}_{i,k} = \hat{y}_{i,k} - y_{i,k}$$

Full solution:

$$\frac{\partial L_i}{\partial z_{i,k}} = \frac{\partial L_i}{\partial \hat{y}_{i,j}} \cdot \frac{\partial S_{i,j}}{\partial z_{i,k}} = \frac{\partial L_i}{\partial \hat{y}_{i,j}} \cdot \frac{\partial \hat{y}_{i,j}}{\partial z_{i,k}} = -\sum_j \frac{y_{i,j}}{\hat{y}_{i,j}} \cdot \frac{\partial \hat{y}_{i,j}}{\partial z_{i,k}} =$$

$$= -\frac{y_{i,k}}{\hat{y}_{i,k}} \cdot \frac{\partial \hat{y}_{i,k}}{\partial z_{i,k}} - \sum_{j \neq k} \frac{y_{i,j}}{\hat{y}_{i,j}} \cdot \frac{\partial \hat{y}_{i,j}}{\partial z_{i,k}} = -\frac{y_{i,k}}{\hat{y}_{i,k}} \cdot \hat{y}_{i,k} \cdot (1 - \hat{y}_{i,k}) - \sum_{j \neq k} \frac{y_{i,j}}{\hat{y}_{i,j}} (-\hat{y}_{i,j}\hat{y}_{i,k}) =$$

$$= -y_{i,k} \cdot (1 - \hat{y}_{i,k}) + \sum_{j \neq k} y_{i,j}\hat{y}_{i,k} = -y_{i,k} + y_{i,k}\hat{y}_{i,k} + \sum_{j \neq k} y_{i,j}\hat{y}_{i,k} =$$

$$= -y_{i,k} + \sum_j y_{i,j}\hat{y}_{i,k} = -y_{i,k} + \hat{y}_{i,k} = \hat{y}_{i,k} - y_{i,k}$$

As we can see, when we apply the chain rule to both partial derivatives, the whole equation simplifies significantly to the subtraction of the predicted and ground truth values. It is also multiple times faster to compute.

Common Categorical Cross-Entropy loss and Softmax activation derivative - code implementation

To code this solution, nothing in the forward pass changes — we still need to perform it on the activation function to receive the outputs and then on the loss function to calculate the loss value. For backpropagation, we'll create the backward step containing the implementation of the new equation, which calculates the combined gradient of the loss and activation functions. We'll code the solution as a separate class, which initializes both the Softmax activation and the Categorical Cross-Entropy objects, calling their forward methods respectively during the forward pass. Then the new backward pass is going to contain the new code:

```
# Softmax classifier - combined Softmax activation
# and cross-entropy loss for faster backward step
class Activation_Softmax_Loss_CategoricalCrossentropy():

    # Creates activation and loss function objects
    def __init__(self):
        self.activation = Activation_Softmax()
        self.loss = Loss_CategoricalCrossentropy()

    # Forward pass
    def forward(self, inputs, y_true):
        # Output layer's activation function
        self.activation.forward(inputs)
        # Set the output
        self.output = self.activation.output
        # Calculate and return loss value
        return self.loss.calculate(self.output, y_true)

    # Backward pass
    def backward(self, dvalues, y_true):

        # Number of samples
        samples = len(dvalues)
```

```
        # If labels are one-hot encoded,
        # turn them into discrete values
        if len(y_true.shape) == 2:
            y_true = np.argmax(y_true, axis=1)

        # Copy so we can safely modify
        self.dinputs = dvalues.copy()
        # Calculate gradient
        self.dinputs[range(samples), y_true] -= 1
        # Normalize gradient
        self.dinputs = self.dinputs / samples
```

To implement the solution $y\text{-}hat_{i,k}\text{-}y_{i,k}$, instead of performing the subtraction of the full arrays, we're taking advantage of the fact that the *y* being `y_true` in the code consists of one-hot encoded vectors, which means that, for each sample, there is only a singular value of *1* in these vectors and the remaining positions are filled with zeros.

This means that we can use NumPy to index the prediction array with the sample number and its true value index, subtracting *1* from these values. This operation requires discrete true labels instead of one-hot encoded ones, thus the additional code that performs the transformation if needed — If the number of dimensions in the ground-truth array equals *2*, it means that it's a matrix consisting of one-hot encoded vectors. We can use `np.argmax()`, which returns the index of the maximum value (index for *1* in this case), but we need to perform this operation sample-wise to get a vector of indices:

```
import numpy as np

y_true = np.array([[1, 0, 0], [0, 0, 1], [0, 1, 0]])
print(np.argmax(y_true))

>>>
0


print(np.argmax(y_true, axis=1))

>>>
[0, 2, 1]
```

For the last step, we normalize the gradient in exactly the same way and for the same reason as described along with the Categorical Cross-Entropy gradient normalization.

Let's summarize the code for each of the classes that we have updated:

```
# Softmax activation
class Activation_Softmax:

    # Forward pass
    def forward(self, inputs):
        # Remember input values
        self.inputs = inputs

        # Get unnormalized probabilities
        exp_values = np.exp(inputs - np.max(inputs, axis=1,
                                            keepdims=True))
        # Normalize them for each sample
        probabilities = exp_values / np.sum(exp_values, axis=1,
                                            keepdims=True)

        self.output = probabilities

    # Backward pass
    def backward(self, dvalues):

        # Create uninitialized array
        self.dinputs = np.empty_like(dvalues)

        # Enumerate outputs and gradients
        for index, (single_output, single_dvalues) in \
                enumerate(zip(self.output, dvalues)):
            # Flatten output array
            single_output = single_output.reshape(-1, 1)
            # Calculate Jacobian matrix of the output
            jacobian_matrix = np.diagflat(single_output) - \
                              np.dot(single_output, single_output.T)
            # Calculate sample-wise gradient
            # and add it to the array of sample gradients
            self.dinputs[index] = np.dot(jacobian_matrix,
                                         single_dvalues)


# Cross-entropy loss
class Loss_CategoricalCrossentropy(Loss):

    # Forward pass
    def forward(self, y_pred, y_true):

        # Number of samples in a batch
        samples = len(y_pred)
```

```
        # Clip data to prevent division by 0
        # Clip both sides to not drag mean towards any value
        y_pred_clipped = np.clip(y_pred, 1e-7, 1 - 1e-7)

        # Probabilities for target values -
        # only if categorical labels
        if len(y_true.shape) == 1:
            correct_confidences = y_pred_clipped[
                range(samples),
                y_true
            ]
        # Mask values - only for one-hot encoded labels
        elif len(y_true.shape) == 2:
            correct_confidences = np.sum(
                y_pred_clipped * y_true,
                axis=1
            )
        # Losses
        negative_log_likelihoods = -np.log(correct_confidences)

        return negative_log_likelihoods

    # Backward pass
    def backward(self, dvalues, y_true):
        # Number of samples
        samples = len(dvalues)
        # Number of labels in every sample
        # We'll use the first sample to count them
        labels = len(dvalues[0])
        # If labels are sparse, turn them into one-hot vector
        if len(y_true.shape) == 1:
            y_true = np.eye(labels)[y_true]
        # Calculate gradient
        self.dinputs = -y_true / dvalues
        # Normalize gradient
        self.dinputs = self.dinputs / samples


# Softmax classifier - combined Softmax activation
# and cross-entropy loss for faster backward step
class Activation_Softmax_Loss_CategoricalCrossentropy():

    # Creates activation and loss function objects
    def __init__(self):
        self.activation = Activation_Softmax()
        self.loss = Loss_CategoricalCrossentropy()
```

```
    # Forward pass
    def forward(self, inputs, y_true):
        # Output layer's activation function
        self.activation.forward(inputs)
        # Set the output
        self.output = self.activation.output
        # Calculate and return loss value
        return self.loss.calculate(self.output, y_true)

    # Backward pass
    def backward(self, dvalues, y_true):
        # Number of samples
        samples = len(dvalues)
        # If labels are one-hot encoded,
        # turn them into discrete values
        if len(y_true.shape) == 2:
            y_true = np.argmax(y_true, axis=1)
        # Copy so we can safely modify
        self.dinputs = dvalues.copy()
        # Calculate gradient
        self.dinputs[range(samples), y_true] -= 1
        # Normalize gradient
        self.dinputs = self.dinputs / samples
```

We can now test if the combined backward step returns the same values compared to when we backpropagate gradients through both of the functions separately. For this example, let's make up an output of the Softmax function and some target values. Next, let's backpropagate them using both solutions:

```
softmax_outputs = np.array([[0.7, 0.1, 0.2],
                            [0.1, 0.5, 0.4],
                            [0.02, 0.9, 0.08]])

class_targets = np.array([0, 1, 1])

softmax_loss = Activation_Softmax_Loss_CategoricalCrossentropy()
softmax_loss.backward(softmax_outputs, class_targets)
dvalues1 = softmax_loss.dinputs

activation = Activation_Softmax()
activation.output = softmax_outputs
loss = Loss_CategoricalCrossentropy()
loss.backward(softmax_outputs, class_targets)
activation.backward(loss.dinputs)
dvalues2 = activation.dinputs
```

```
print('Gradients: combined loss and activation:')
print(dvalues1)
print('Gradients: separate loss and activation:')
print(dvalues2)

>>>
Gradients: combined loss and activation:
[[-0.1         0.03333333  0.06666667]
 [ 0.03333333 -0.16666667  0.13333333]
 [ 0.00666667 -0.03333333  0.02666667]]
Gradients: separate loss and activation:
[[-0.09999999  0.03333334  0.06666667]
 [ 0.03333334 -0.16666667  0.13333334]
 [ 0.00666667 -0.03333333  0.02666667]]
```

The results are the same. The small difference between values in both arrays results from the precision of floating-point values in raw Python and NumPy. To answer the question of how many times faster this solution is, we can take advantage of Python's timeit module, running both solutions multiple times and combining the execution times. A full description of the timeit module and the code used here is outside of the scope of this book, but we include this code purely to show the speed deltas:

```
from timeit import timeit

softmax_outputs = np.array([[0.7, 0.1, 0.2],
                            [0.1, 0.5, 0.4],
                            [0.02, 0.9, 0.08]])

class_targets = np.array([0, 1, 1])


def f1():
    softmax_loss = Activation_Softmax_Loss_CategoricalCrossentropy()
    softmax_loss.backward(softmax_outputs, class_targets)
    dvalues1 = softmax_loss.dinputs

def f2():
    activation = Activation_Softmax()
    activation.output = softmax_outputs
    loss = Loss_CategoricalCrossentropy()
    loss.backward(softmax_outputs, class_targets)
    activation.backward(loss.dinputs)
    dvalues2 = activation.dinputs
```

```
t1 = timeit(lambda: f1(), number=10000)
t2 = timeit(lambda: f2(), number=10000)
print(t2/t1)
```

```
>>>
6.922146504409747
```

Calculating the gradients separately is about 7 times slower.. This factor can differ from a machine to a machine, but it clearly shows that it was worth putting in additional effort to calculate and code the optimized solution of the combined loss and activation function derivative.

Let's take the code of the model and initialize the new class of combined accuracy and loss class' object:

```
# Create Softmax classifier's combined loss and activation
loss_activation = Activation_Softmax_Loss_CategoricalCrossentropy()
```

Instead of the previous:

```
# Create Softmax activation (to be used with Dense layer):
activation2 = Activation_Softmax()

# Create loss function
loss_function = Loss_CategoricalCrossentropy()
```

Then replace the forward pass calls over these objects:

```
# Perform a forward pass through activation function
# takes the output of second dense layer here
activation2.forward(dense2.output)
...
# Calculate sample losses from output of activation2 (softmax activation)
loss = loss_function.forward(activation2.output, y)
```

With the forward pass call on the new object:

```
# Perform a forward pass through the activation/loss function
# takes the output of second dense layer here and returns loss
loss = loss_activation.forward(dense2.output, y)
```

And finally add the backward step and printing gradients:

```
# Backward pass
loss_activation.backward(loss_activation.output, y)
dense2.backward(loss_activation.dinputs)
activation1.backward(dense2.dinputs)
dense1.backward(activation1.dinputs)

# Print gradients
print(dense1.dweights)
print(dense1.dbiases)
print(dense2.dweights)
print(dense2.dbiases)
```

Full model code:

```
# Create dataset
X, y = spiral_data(samples=100, classes=3)

# Create Dense layer with 2 input features and 3 output values
dense1 = Layer_Dense(2, 3)

# Create ReLU activation (to be used with Dense layer):
activation1 = Activation_ReLU()

# Create second Dense layer with 3 input features (as we take output
# of previous layer here) and 3 output values (output values)
dense2 = Layer_Dense(3, 3)

# Create Softmax classifier's combined loss and activation
loss_activation = Activation_Softmax_Loss_CategoricalCrossentropy()

# Perform a forward pass of our training data through this layer
dense1.forward(X)

# Perform a forward pass through activation function
# takes the output of first dense layer here
activation1.forward(dense1.output)

# Perform a forward pass through second Dense layer
# takes outputs of activation function of first layer as inputs
dense2.forward(activation1.output)
```

```
# Perform a forward pass through the activation/loss function
# takes the output of second dense layer here and returns loss
loss = loss_activation.forward(dense2.output, y)

# Let's see output of the first few samples:
print(loss_activation.output[:5])

# Print loss value
print('loss:', loss)

# Calculate accuracy from output of activation2 and targets
# calculate values along first axis
predictions = np.argmax(loss_activation.output, axis=1)
if len(y.shape) == 2:
    y = np.argmax(y, axis=1)
accuracy = np.mean(predictions==y)

# Print accuracy
print('acc:', accuracy)

# Backward pass
loss_activation.backward(loss_activation.output, y)
dense2.backward(loss_activation.dinputs)
activation1.backward(dense2.dinputs)
dense1.backward(activation1.dinputs)

# Print gradients
print(dense1.dweights)
print(dense1.dbiases)
print(dense2.dweights)
print(dense2.dbiases)
```

```
>>>
[[0.33333334 0.33333334 0.33333334]
 [0.33333316 0.3333332  0.33333364]
 [0.33333287 0.3333329  0.33333418]
 [0.3333326  0.33333263 0.33333477]
 [0.33333233 0.3333324  0.33333528]]
loss: 1.0986104
acc: 0.34
[[ 1.5766358e-04  7.8368575e-05  4.7324404e-05]
 [ 1.8161036e-04  1.1045571e-05 -3.3096316e-05]]
[[-3.6055347e-04  9.6611722e-05 -1.0367142e-04]]
[[ 5.4410957e-05  1.0741142e-04 -1.6182236e-04]
 [-4.0791339e-05 -7.1678100e-05  1.1246944e-04]
 [-5.3011299e-05  8.5817286e-05 -3.2805994e-05]]
[[-1.0732794e-05 -9.4590941e-06  2.0027626e-05]]
```

Full code up to this point:

```
import numpy as np
import nnfs
from nnfs.datasets import spiral_data

nnfs.init()


# Dense layer
class Layer_Dense:

    # Layer initialization
    def __init__(self, n_inputs, n_neurons):
        # Initialize weights and biases
        self.weights = 0.01 * np.random.randn(n_inputs, n_neurons)
        self.biases = np.zeros((1, n_neurons))

    # Forward pass
    def forward(self, inputs):
        # Remember input values
        self.inputs = inputs
        # Calculate output values from input ones, weights and biases
        self.output = np.dot(inputs, self.weights) + self.biases

    # Backward pass
    def backward(self, dvalues):
        # Gradients on parameters
        self.dweights = np.dot(self.inputs.T, dvalues)
        self.dbiases = np.sum(dvalues, axis=0, keepdims=True)
        # Gradient on values
        self.dinputs = np.dot(dvalues, self.weights.T)


# ReLU activation
class Activation_ReLU:

    # Forward pass
    def forward(self, inputs):
```

```
        # Remember input values
        self.inputs = inputs
        # Calculate output values from inputs
        self.output = np.maximum(0, inputs)

    # Backward pass
    def backward(self, dvalues):
        # Since we need to modify original variable,
        # let's make a copy of values first
        self.dinputs = dvalues.copy()

        # Zero gradient where input values were negative
        self.dinputs[self.inputs <= 0] = 0


# Softmax activation
class Activation_Softmax:

    # Forward pass
    def forward(self, inputs):
        # Remember input values
        self.inputs = inputs

        # Get unnormalized probabilities
        exp_values = np.exp(inputs - np.max(inputs, axis=1,
                                            keepdims=True))
        # Normalize them for each sample
        probabilities = exp_values / np.sum(exp_values, axis=1,
                                            keepdims=True)

        self.output = probabilities

    # Backward pass
    def backward(self, dvalues):

        # Create uninitialized array
        self.dinputs = np.empty_like(dvalues)

        # Enumerate outputs and gradients
        for index, (single_output, single_dvalues) in \
                enumerate(zip(self.output, dvalues)):
            # Flatten output array
            single_output = single_output.reshape(-1, 1)
            # Calculate Jacobian matrix of the output
            jacobian_matrix = np.diagflat(single_output) - \
                              np.dot(single_output, single_output.T)
```

```
            # Calculate sample-wise gradient
            # and add it to the array of sample gradients
            self.dinputs[index] = np.dot(jacobian_matrix,
                                         single_dvalues)


# Common loss class
class Loss:

    # Calculates the data and regularization losses
    # given model output and ground truth values
    def calculate(self, output, y):

        # Calculate sample losses
        sample_losses = self.forward(output, y)

        # Calculate mean loss
        data_loss = np.mean(sample_losses)

        # Return loss
        return data_loss


# Cross-entropy loss
class Loss_CategoricalCrossentropy(Loss):

    # Forward pass
    def forward(self, y_pred, y_true):

        # Number of samples in a batch
        samples = len(y_pred)

        # Clip data to prevent division by 0
        # Clip both sides to not drag mean towards any value
        y_pred_clipped = np.clip(y_pred, 1e-7, 1 - 1e-7)

        # Probabilities for target values -
        # only if categorical labels
        if len(y_true.shape) == 1:
            correct_confidences = y_pred_clipped[
                range(samples),
                y_true
            ]
```

```
        # Mask values - only for one-hot encoded labels
        elif len(y_true.shape) == 2:
            correct_confidences = np.sum(
                y_pred_clipped * y_true,
                axis=1
            )

        # Losses
        negative_log_likelihoods = -np.log(correct_confidences)
        return negative_log_likelihoods

    # Backward pass
    def backward(self, dvalues, y_true):

        # Number of samples
        samples = len(dvalues)
        # Number of labels in every sample
        # We'll use the first sample to count them
        labels = len(dvalues[0])

        # If labels are sparse, turn them into one-hot vector
        if len(y_true.shape) == 1:
            y_true = np.eye(labels)[y_true]

        # Calculate gradient
        self.dinputs = -y_true / dvalues
        # Normalize gradient
        self.dinputs = self.dinputs / samples


# Softmax classifier - combined Softmax activation
# and cross-entropy loss for faster backward step
class Activation_Softmax_Loss_CategoricalCrossentropy():

    # Creates activation and loss function objects
    def __init__(self):
        self.activation = Activation_Softmax()
        self.loss = Loss_CategoricalCrossentropy()

    # Forward pass
    def forward(self, inputs, y_true):
        # Output layer's activation function
        self.activation.forward(inputs)
        # Set the output
        self.output = self.activation.output
        # Calculate and return loss value
        return self.loss.calculate(self.output, y_true)
```

```
    # Backward pass
    def backward(self, dvalues, y_true):

        # Number of samples
        samples = len(dvalues)

        # If labels are one-hot encoded,
        # turn them into discrete values
        if len(y_true.shape) == 2:
            y_true = np.argmax(y_true, axis=1)

        # Copy so we can safely modify
        self.dinputs = dvalues.copy()
        # Calculate gradient
        self.dinputs[range(samples), y_true] -= 1
        # Normalize gradient
        self.dinputs = self.dinputs / samples


# Create dataset
X, y = spiral_data(samples=100, classes=3)

# Create Dense layer with 2 input features and 3 output values
dense1 = Layer_Dense(2, 3)

# Create ReLU activation (to be used with Dense layer):
activation1 = Activation_ReLU()

# Create second Dense layer with 3 input features (as we take output
# of previous layer here) and 3 output values (output values)
dense2 = Layer_Dense(3, 3)

# Create Softmax classifier's combined loss and activation
loss_activation = Activation_Softmax_Loss_CategoricalCrossentropy()

# Perform a forward pass of our training data through this layer
dense1.forward(X)

# Perform a forward pass through activation function
# takes the output of first dense layer here
activation1.forward(dense1.output)

# Perform a forward pass through second Dense layer
# takes outputs of activation function of first layer as inputs
dense2.forward(activation1.output)

# Perform a forward pass through the activation/loss function
# takes the output of second dense layer here and returns loss
loss = loss_activation.forward(dense2.output, y)
```

```
# Let's see output of the first few samples:
print(loss_activation.output[:5])

# Print loss value
print('loss:', loss)

# Calculate accuracy from output of activation2 and targets
# calculate values along first axis
predictions = np.argmax(loss_activation.output, axis=1)
if len(y.shape) == 2:
    y = np.argmax(y, axis=1)
accuracy = np.mean(predictions == y)

# Print accuracy
print('acc:', accuracy)

# Backward pass
loss_activation.backward(loss_activation.output, y)
dense2.backward(loss_activation.dinputs)
activation1.backward(dense2.dinputs)
dense1.backward(activation1.dinputs)

# Print gradients
print(dense1.dweights)
print(dense1.dbiases)
print(dense2.dweights)
print(dense2.dbiases)
```

At this point, thanks to gradients and backpropagation using the chain rule, we're able to adjust the weights and biases with the goal of lowering loss, but we'd be doing it in a very rudimentary way. This process of adjusting weights and biases using gradients to decrease loss is the job of the optimizer, which is the subject of the next chapter.

Supplementary Material: https://nnfs.io/ch9
Chapter code, further resources, and errata for this chapter.

Chapter 10

Optimizers

Once we have calculated the gradient, we can use this information to adjust weights and biases to decrease the measure of loss. In a previous toy example, we showed how we could successfully decrease a neuron's activation function's (ReLU) output in this manner. Recall that we subtracted a fraction of the gradient for each weight and bias parameter. While very rudimentary, this is still a commonly used optimizer called **Stochastic Gradient Descent (SGD)**. As you will soon discover, most optimizers are just variants of SGD.

Stochastic Gradient Descent (SGD)

There are some naming conventions with this optimizer that can be confusing, so let's walk through those first. You might hear the following names:

- Stochastic Gradient Descent, SGD
- Vanilla Gradient Descent, Gradient Descent, GD, or Batch Gradient Descent, BGD
- Mini-batch Gradient Descent, MBGD

The first name, **Stochastic Gradient Descent**, historically refers to an optimizer that fits a single sample at a time. The second optimizer, **Batch Gradient Descent**, is an optimizer used to fit a whole dataset at once. The last optimizer, **Mini-batch Gradient Descent**, is used to fit slices of a dataset, which we'd call batches in our context. The naming convention can be confusing here for multiple reasons.

First, in the context of deep learning and this book, we call slices of data **batches**, where, historically, the term to refer to slices of data in the context of Stochastic Gradient Descent was **mini-batches**. In our context, it does not matter if the batch contains a single sample, a slice of the dataset, or the full dataset — as a batch of the data. Additionally, with the current code, we are fitting the full dataset; following this naming convention, we would use **Batch Gradient Descent**. In a future chapter, we'll introduce data slices, or **batches**, so we should start by using the **Mini-batch Gradient Descent** optimizer. That said, current naming trends and conventions with Stochastic Gradient Descent in use with deep learning today have merged and normalized all of these variants, to the point where we think of the **Stochastic Gradient Descent** optimizer as one that assumes a batch of data, whether that batch happens to be a single sample, every sample in a dataset, or some subset of the full dataset at a time.

In the case of Stochastic Gradient Descent, we choose a learning rate, such as *1.0*. We then subtract the *learning_rate · parameter_gradients* from the actual parameter values. If our learning rate is 1, then we're subtracting the exact amount of gradient from our parameters. We're going to start with 1 to see the results, but we'll be diving more into the learning rate shortly. Let's create the SGD optimizer class code. The initialization method will take hyper-parameters, starting with the learning rate, for now, storing them in the class' properties. The `update_params` method, given a layer object, performs the most basic optimization, the same way that we performed it in the previous chapter — it multiplies the gradients stored in the layers by the negated learning rate

and adds the result to the layer's parameters. It seems that, in the previous chapter, we performed SGD optimization without knowing it. The full class so far:

```
# SGD optimizer
class Optimizer_SGD:

    # Initialize optimizer - set settings,
    # learning rate of 1. is default for this optimizer
    def __init__(self, learning_rate=1.0):
        self.learning_rate = learning_rate

    # Update parameters
    def update_params(self, layer):
        layer.weights += -self.learning_rate * layer.dweights
        layer.biases += -self.learning_rate * layer.dbiases
```

To use this, we need to create an optimizer object:

```
optimizer = Optimizer_SGD()
```

Then update our network layer's parameters after calculating the gradient using:

```
optimizer.update_params(dense1)
optimizer.update_params(dense2)
```

Recall that the layer object contains its parameters (weights and biases) and also, at this stage, the gradient that is calculated during backpropagation. We store these in the layer's properties so that the optimizer can make use of them. In our main neural network code, we'd bring the optimization in after backpropagation. Let's make a 1x64 densely-connected neural network (1 hidden layer with 64 neurons) and use the same dataset as before:

```
# Create dataset
X, y = spiral_data(samples=100, classes=3)

# Create Dense layer with 2 input features and 64 output values
dense1 = Layer_Dense(2, 64)

# Create ReLU activation (to be used with Dense layer):
activation1 = Activation_ReLU()

# Create second Dense layer with 64 input features (as we take output
# of previous layer here) and 3 output values (output values)
dense2 = Layer_Dense(64, 3)

# Create Softmax classifier's combined loss and activation
loss_activation = Activation_Softmax_Loss_CategoricalCrossentropy()
```

The next step is to create the optimizer's object:

```
# Create optimizer
optimizer = Optimizer_SGD()
```

Then perform a **forward pass** of our sample data:

```
# Perform a forward pass of our training data through this layer
dense1.forward(X)

# Perform a forward pass through activation function
# takes the output of first dense layer here
activation1.forward(dense1.output)

# Perform a forward pass through second Dense layer
# takes outputs of activation function of first layer as inputs
dense2.forward(activation1.output)

# Perform a forward pass through the activation/loss function
# takes the output of second dense layer here and returns loss
loss = loss_activation.forward(dense2.output, y)

# Let's print loss value
print('loss:', loss)

# Calculate accuracy from output of activation2 and targets
# calculate values along first axis
predictions = np.argmax(loss_activation.output, axis=1)
if len(y.shape) == 2:
    y = np.argmax(y, axis=1)
accuracy = np.mean(predictions==y)

print('acc:', accuracy)
```

Next, we do our **backward pass**, which is also called **backpropagation**:

```
# Backward pass
loss_activation.backward(loss_activation.output, y)
dense2.backward(loss_activation.dinputs)
activation1.backward(dense2.dinputs)
dense1.backward(activation1.dinputs)
```

Then we finally use our optimizer to update weights and biases:

```
# Update weights and biases
optimizer.update_params(dense1)
optimizer.update_params(dense2)
```

This is everything we need to train our model! But why would we only perform this optimization once, when we can perform it lots of times by leveraging Python's looping capabilities? We will repeatedly perform a forward pass, backward pass, and optimization until we reach some stopping point. Each full pass through all of the training data is called an **epoch**. In most deep learning tasks, a neural network will be trained for multiple epochs, though the ideal scenario would be to have a perfect model with ideal weights and biases after only one epoch. To add multiple epochs of training into our code, we will initialize our model and run a loop around all the code performing the forward pass, backward pass, and optimization calculations:

```
# Create dataset
X, y = spiral_data(samples=100, classes=3)

# Create Dense layer with 2 input features and 64 output values
dense1 = Layer_Dense(2, 64)

# Create ReLU activation (to be used with Dense layer):
activation1 = Activation_ReLU()

# Create second Dense layer with 64 input features (as we take output
# of previous layer here) and 3 output values (output values)
dense2 = Layer_Dense(64, 3)

# Create Softmax classifier's combined loss and activation
loss_activation = Activation_Softmax_Loss_CategoricalCrossentropy()

# Create optimizer
optimizer = Optimizer_SGD()

# Train in loop
for epoch in range(10001):

    # Perform a forward pass of our training data through this layer
    dense1.forward(X)

    # Perform a forward pass through activation function
    # takes the output of first dense layer here
    activation1.forward(dense1.output)

    # Perform a forward pass through second Dense layer
    # takes outputs of activation function of first layer as inputs
    dense2.forward(activation1.output)

    # Perform a forward pass through the activation/loss function
    # takes the output of second dense layer here and returns loss
    loss = loss_activation.forward(dense2.output, y)
```

```
    # Calculate accuracy from output of activation2 and targets
    # calculate values along first axis
    predictions = np.argmax(loss_activation.output, axis=1)
    if len(y.shape) == 2:
        y = np.argmax(y, axis=1)
    accuracy = np.mean(predictions==y)

    if not epoch % 100:
        print(f'epoch: {epoch}, ' +
              f'acc: {accuracy:.3f}, ' +
              f'loss: {loss:.3f}')

    # Backward pass
    loss_activation.backward(loss_activation.output, y)
    dense2.backward(loss_activation.dinputs)
    activation1.backward(dense2.dinputs)
    dense1.backward(activation1.dinputs)

    # Update weights and biases
    optimizer.update_params(dense1)
    optimizer.update_params(dense2)
```

This gives us an update of where we are (epochs), the model's accuracy, and loss every 100 epochs. Initially, we can see consistent improvement:

```
epoch: 0, acc: 0.360, loss: 1.099
epoch: 100, acc: 0.400, loss: 1.087
epoch: 200, acc: 0.417, loss: 1.077
...
epoch: 1000, acc: 0.407, loss: 1.058
...
epoch: 2000, acc: 0.403, loss: 1.038
epoch: 2100, acc: 0.447, loss: 1.022
epoch: 2200, acc: 0.467, loss: 1.023
epoch: 2300, acc: 0.437, loss: 1.005
epoch: 2400, acc: 0.497, loss: 0.993
epoch: 2500, acc: 0.513, loss: 0.981
...
epoch: 9500, acc: 0.590, loss: 0.865
epoch: 9600, acc: 0.627, loss: 0.863
epoch: 9700, acc: 0.630, loss: 0.830
epoch: 9800, acc: 0.663, loss: 0.844
epoch: 9900, acc: 0.627, loss: 0.820
epoch: 10000, acc: 0.633, loss: 0.848
```

Additionally, we've prepared animations to help visualize the training process and to convey the impact of various optimizers and their hyperparameters. The left part of the animation canvas

contains dots, where color represents each of the 3 classes of the data, the coordinates are features, and the background colors show the model prediction areas. Ideally, the points' colors and the background should match if the model classifies correctly. The surrounding area should also follow the data's "trend" — which is what we'd call generalization — the ability of the model to correctly predict unseen data. The colorful squares on the right show weights and biases — red for positive and blue for negative values. The matching areas right below the Dense 1 bar and next to the Dense 2 bar show the updates that the optimizer performs to the layers. The updates might look overly strong compared to the weights and biases, but that's because we've visually normalized them to the maximum value, or else they would be almost invisible since the updates are quite small at a time. The other 3 graphs show the loss, accuracy, and current learning rate values in conjunction with the training time, epochs in this case.

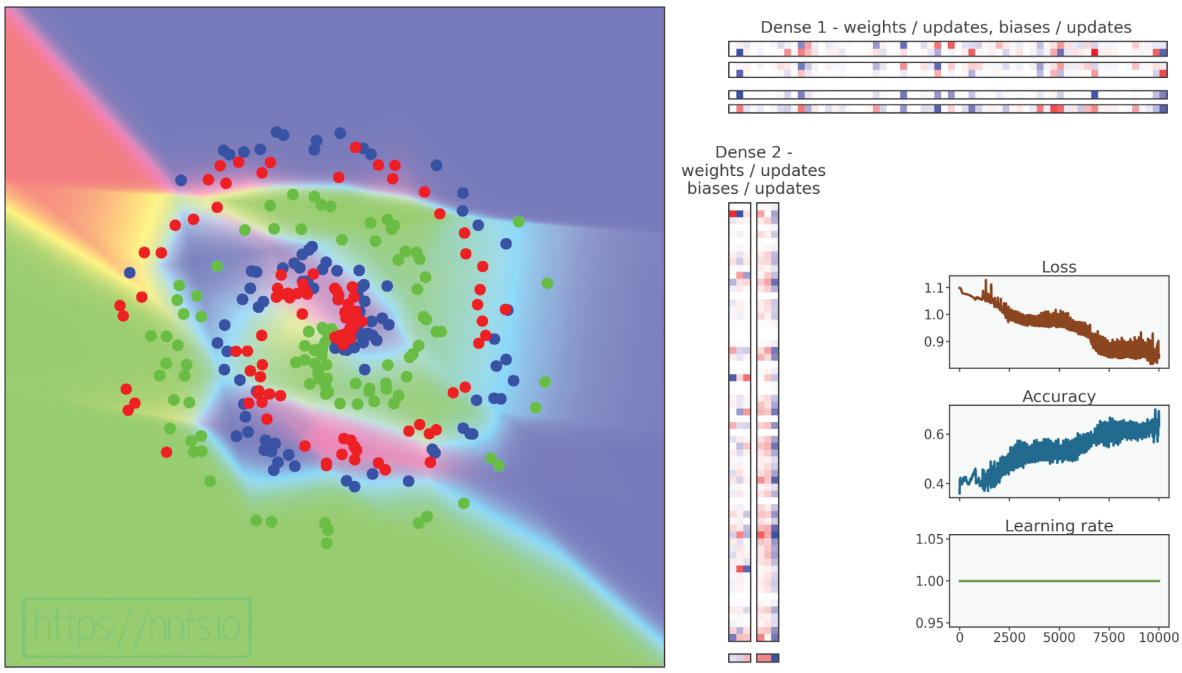


Fig 10.01: Model training with Stochastic Gradient Descent optimizer.

Epilepsy Warning, there are quick flashing colors in the animation:

Anim 10.01: https://nnfs.io/pup

Our neural network mostly stays stuck at around a loss of 1 and later 0.85-0.9, and an accuracy around 0.60. The animation also has a "flashy wiggle" effect, which most likely means we chose too high of a learning rate. Given that loss didn't decrease much, we can assume that this learning rate, being too high, also caused the model to get stuck in a **local minimum**, which we'll learn more about soon. Iterating over more epochs doesn't seem helpful at this point, which tells us that we're likely stuck with our optimization. Does this mean that this is the most we can get from our optimizer on this dataset?

Recall that we're adjusting our weights and biases by applying some fraction, in this case, *1.0*, to the gradient and subtracting this from the weights and biases. This fraction is called the **learning rate** (LR) and is the primary adjustable parameter for the optimizer as it decreases loss. To gain an intuition for adjusting, planning, or initially setting the learning rate, we should first understand how the learning rate affects the optimizer and output of the loss function.

Learning Rate

So far, we have a gradient of a model and the loss function with respect to all of the parameters, and we want to apply a fraction of this gradient to the parameters in order to descend the loss value.

In most cases, we won't apply the negative gradient as is, as the direction of the function's steepest descent will be continuously changing, and these values will usually be too big for meaningful model improvements to occur. Instead, we want to perform small steps — calculating the gradient, updating parameters by a negative fraction of this gradient, and repeating this in a loop. Small steps ensure that we are following the direction of the steepest descent, but these steps can also be too small, causing learning stagnation — we'll explain this shortly.

Let's forget, for a while, that we are performing gradient descent of an n-dimensional function (our loss function), where n is the number parameters (weights and biases) that the model contains, and assume that we have just one dimension to the loss function (a singular input). Our goal for the following images and animations is to visualize some concepts and gain an intuition; thus, we will not use or present certain optimizer settings, and instead will be considering things in more general terms. That said, we've used a real SGD optimizer on a real function to prepare all of the following examples. Here's the function where we want to determine what input to it will result in the lowest possible output:

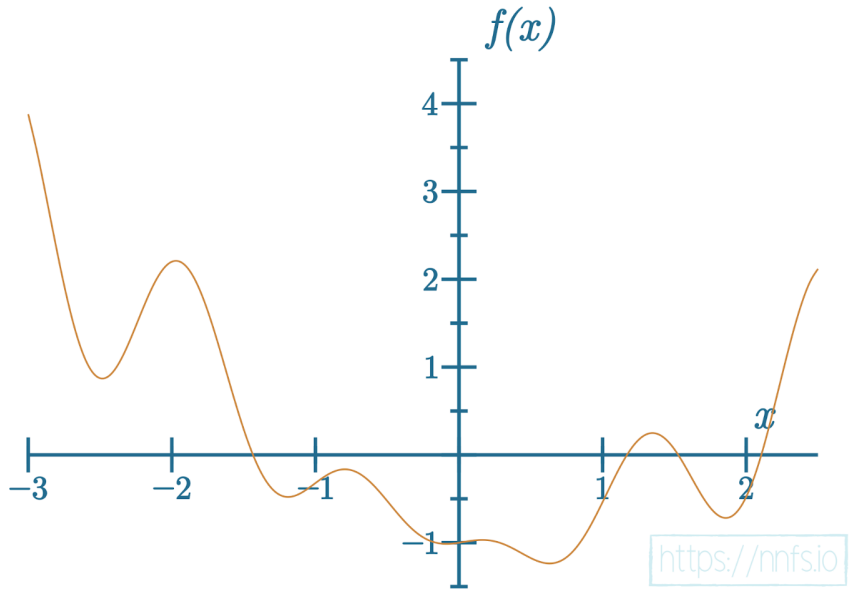


Fig 10.02: Example function to minimize the output.

We can see the **global minimum** of this function, which is the lowest possible *y* value that this function can output. This is the goal — to minimize the function's output to find the global minimum. The values of the axes are not important in this case. The goal is only to show the function and the learning rate concept. Also, remember that this one-dimensional function example is being used merely to aid in visualization. It would be easy to solve this function with simpler math than what is required to solve the much larger n-dimensional loss function for neural networks, where n (which is the number of weights and biases) can be in the millions or even billions (or more). When we have millions of, or more, dimensions, gradient descent is the best-known way to search for a global minimum.

We'll start descending from the left side of this graph. With an example learning rate:

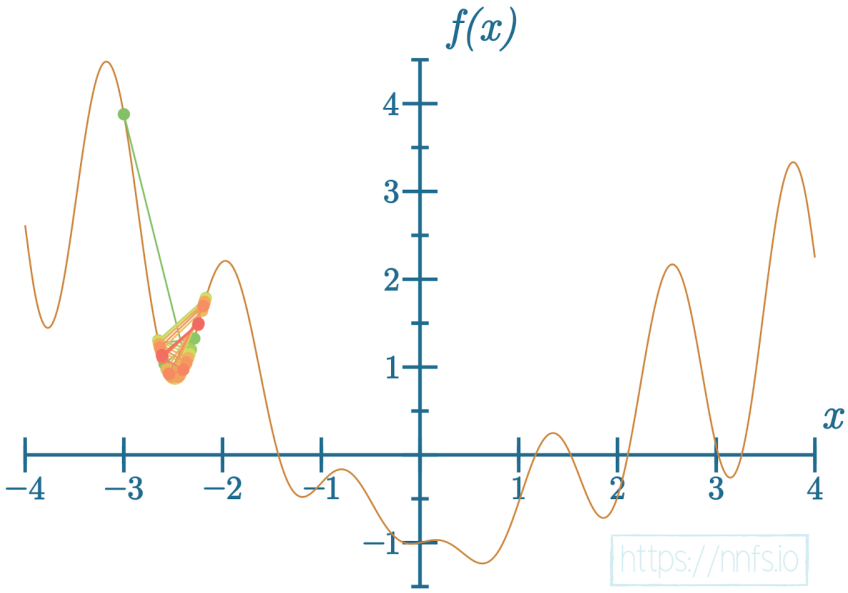


Fig 10.03: Stuck in the first local minimum.

Anim 10.03: https://nnfs.io/and

The learning rate turned out to be too small. Small updates to the parameters caused stagnation in the model's learning — the model got stuck in a local minimum. The **local minimum** is a minimum that is near where we look but isn't necessarily the global minimum, which is the absolute lowest point for a function. With our example here, as well as with optimizing full neural networks, we do not know where the global minimum is. How do we know if we've reached the global minimum or at least gotten close? The loss function measures how far the model is with its predictions to the real target values, so, as long as the loss value is not *0* or very close to *0*, and the model stopped learning, we're at some local minimum. In reality, we almost never approach a loss of *0* for various reasons. One reason for this may be imperfect neural network hyperparameters. Another reason for this may be insufficient data. If you did reach a loss of 0 with a neural network, you should find it suspicious, for reasons we'll get into later in this book.

We can try to modify the learning rate:

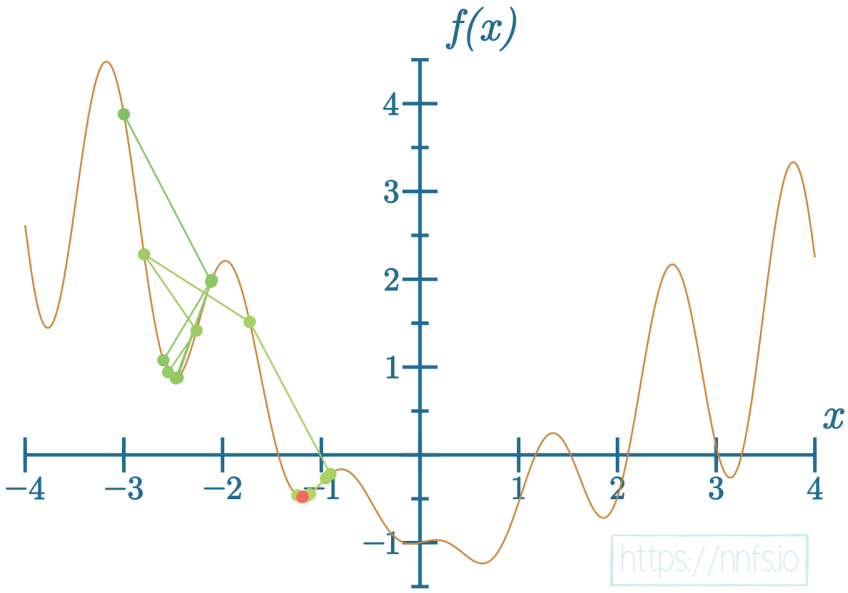


Fig 10.04: Stuck in the second local minimum.

Anim 10.04: https://nnfs.io/xor

This time, the model escaped this local minimum but got stuck at another one. Let's see one more example after another learning rate change:

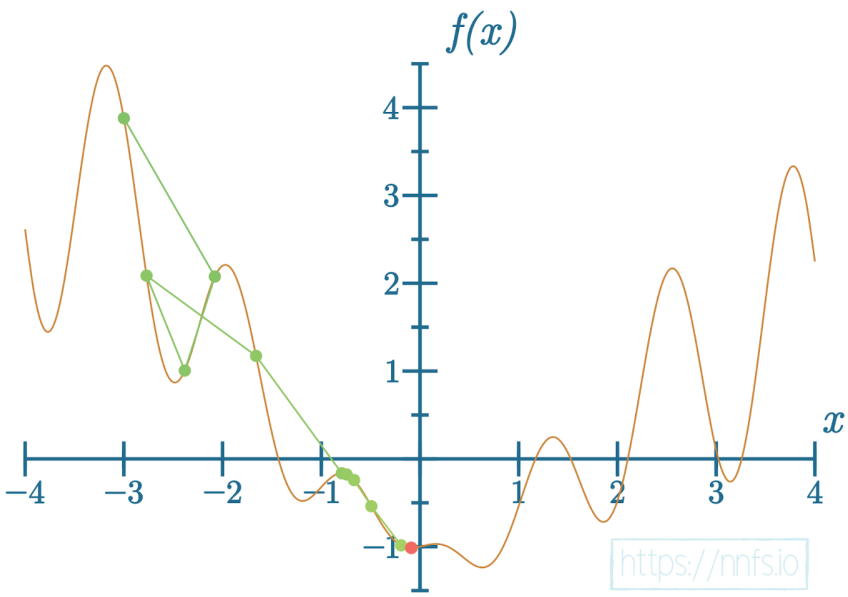


Fig 10.05: Stuck in the third local minimum, near the global minimum.

Anim 10.05: https://nnfs.io/tho

This time the model got stuck at a local minimum near the global minimum. The model was able to escape the "deeper" local minimums, so it might be counter-intuitive why it is stuck here. Remember, the model follows the direction of steepest descent of the loss function, no matter how large or slight the descent is. For this reason, we'll introduce momentum and the other techniques to prevent such situations.

Momentum, in an optimizer, adds to the gradient what, in the physical world, we could call inertia — for example, we can throw a ball uphill and, with a small enough hill or big enough applied force, the ball can roll-over to the other side of the hill. Let's see how this might look with the model in training:

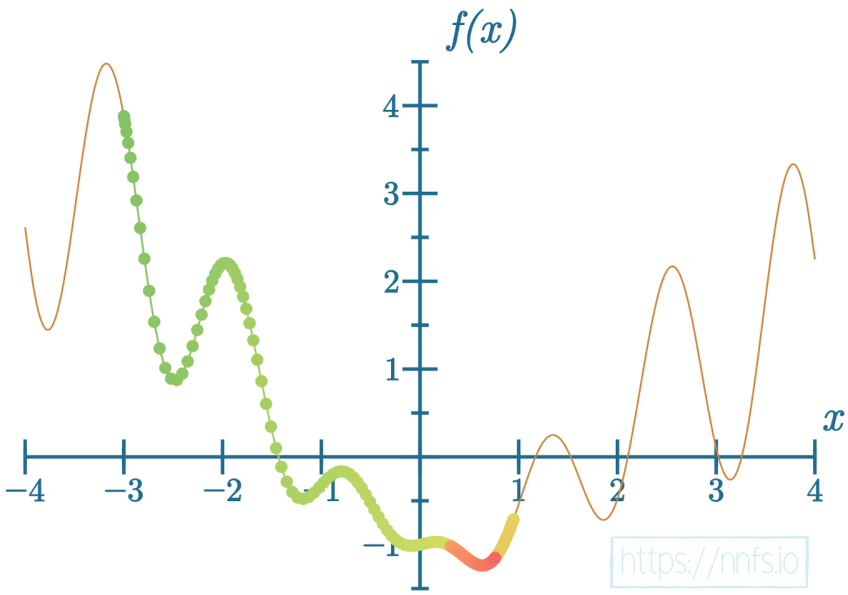


Fig 10.06: Reached the global minimum, too low learning rate.

Anim 10.06: https://nnfs.io/pog

We used a very small learning rate here with a large momentum. The color change from green, through orange to red presents the advancement of the gradient descent process, the steps. We can see that the model achieved the goal and found the global minimum, but this took many steps. Can this be done better?

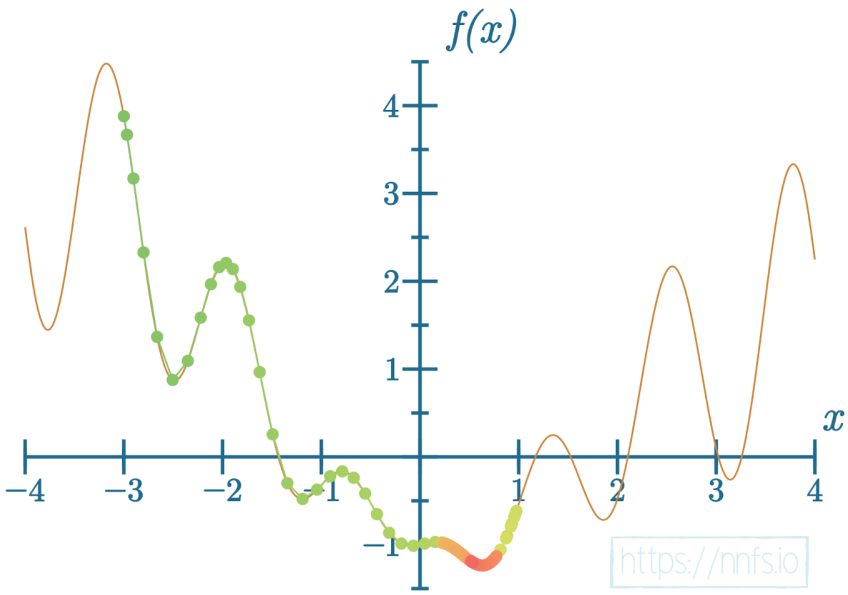


Fig 10.07: Reached the global minimum, better learning rate.

Anim 10.07: https://nnfs.io/jog

And even further:

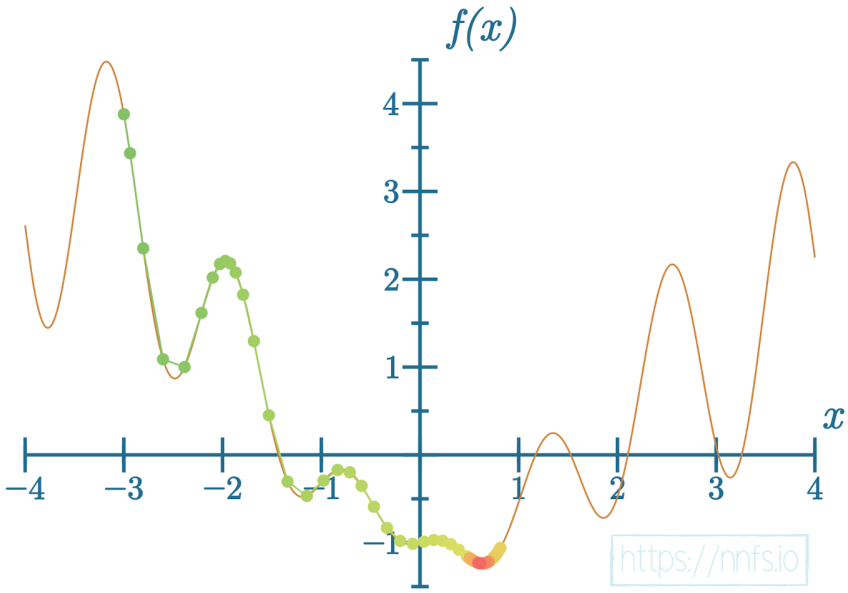


Fig 10.08: Reached the global minimum, significantly better learning rate.

Anim 10.08: https://nnfs.io/mog

With these examples, we were able to find the global minimum in about 200, 100, and 50 steps, respectively, by modifying the learning rate and the momentum. It's possible to significantly shorten the training time by adjusting the parameters of the optimizer. However, we have to be careful with these hyper-parameter adjustments, as this won't necessarily always help the model:

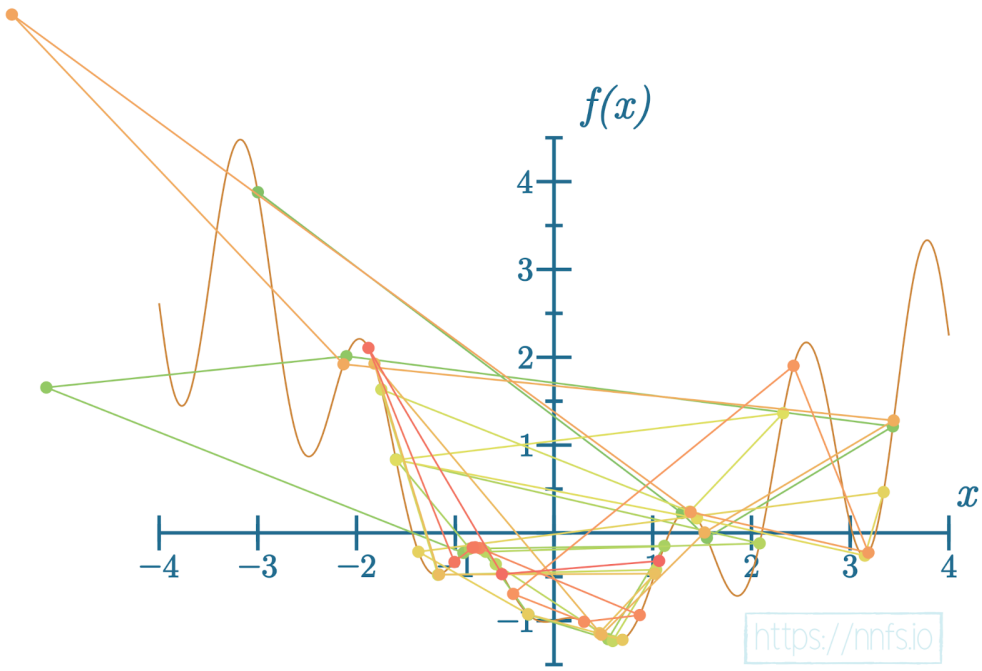


Fig 10.09: Unstable model, learning rate too big.

Anim 10.09: https://nnfs.io/log

With the learning rate set too high, the model might not be able to find the global minimum. Even, at some point, if it does, further adjustments could cause it to jump out of this minimum. We'll see this behavior later in this chapter — try to take a close look at results and see if you can find it, as well as the other issues we've described, from the different optimizers as we work through them.

In this case, the model was "jumping" around some minimum and what this might mean is that we should try to lower the learning rate, raise the momentum, or possibly apply a learning rate decay (lowering the learning rate during training), which we'll describe in this chapter. If we set the learning rate far too high:

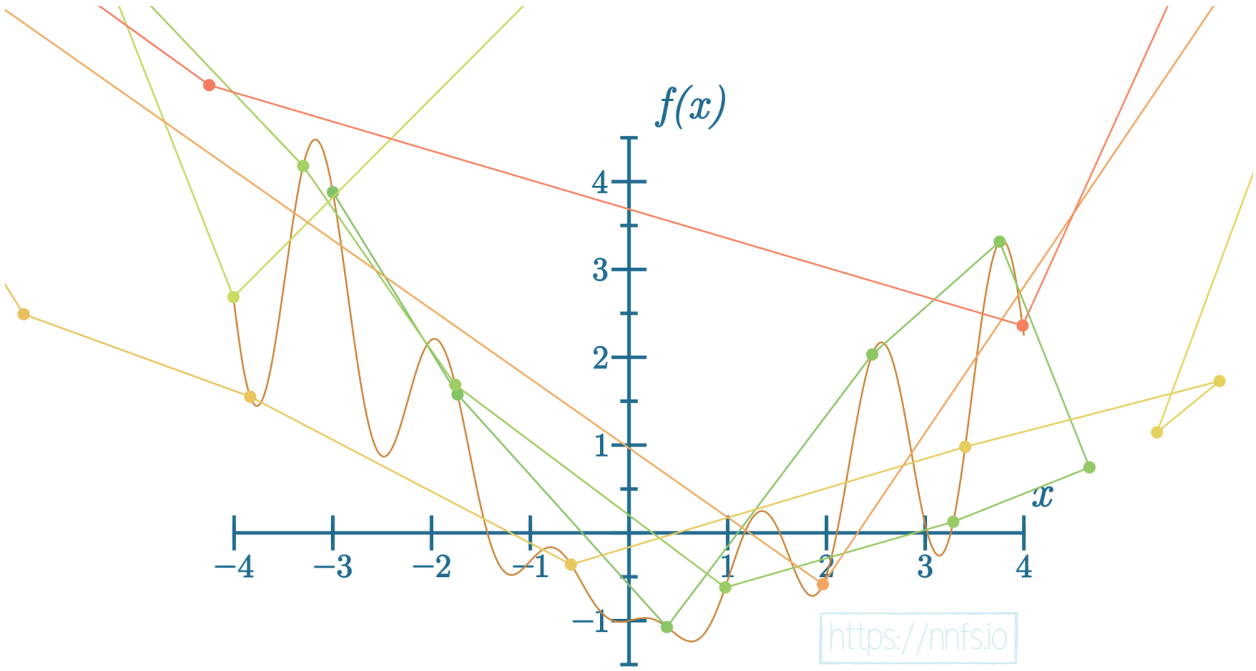


Fig 10.10: Unstable model, learning rate significantly too big.

Anim 10.10: https://nnfs.io/sog

In this situation, the model starts "jumping" around, and moves in what we might observe as random directions. This is an example of "overshooting," with every step — the direction of a change is correct, but the amount of the gradient applied is too large. In an extreme situation, we could cause a **gradient explosion**:

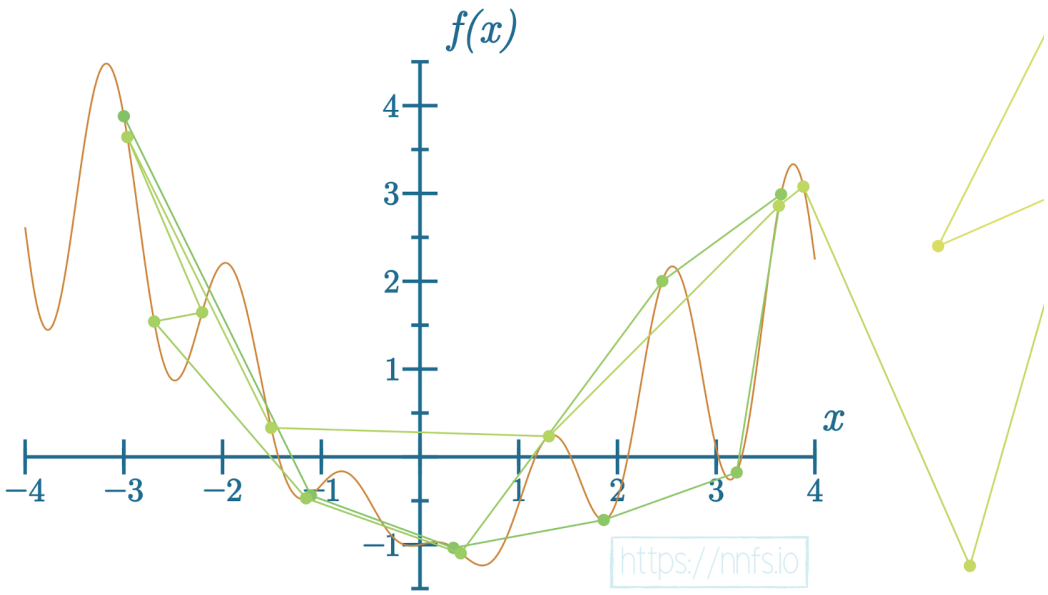


Fig 10.11: Broken model, learning rate critically too big.

Anim 10.11: https://nnfs.io/bog

A gradient explosion is a situation where the parameter updates cause the function's output to rise instead of fall, and, with each step, the loss value and gradient become larger. At some point, the floating-point variable limitation causes an overflow as it cannot hold values of this size anymore, and the model is no longer able to train. It's crucial to recognize this situation forming during training, especially for large models, where the training can take days, weeks, or more. It is possible to tune the model's hyper-parameters in time to save the model and to continue training.

When we choose the learning rate and the other hyper-parameters correctly, the learning process can be relatively quick:

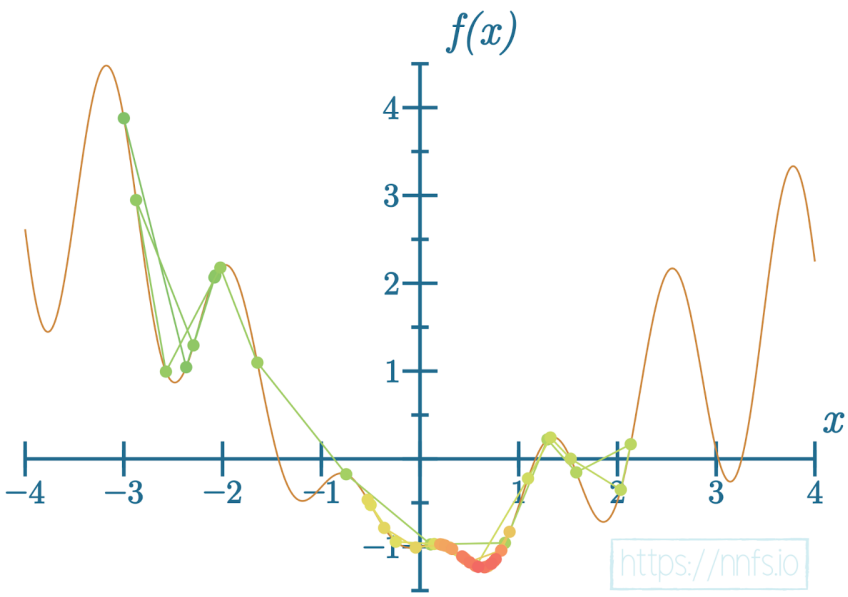


Fig 10.12: Model learned, good learning rate, can be better.

Anim 10.12: https://nnfs.io/cog

This time it took significantly less time for the model to find the global minimum, but it can always be better:

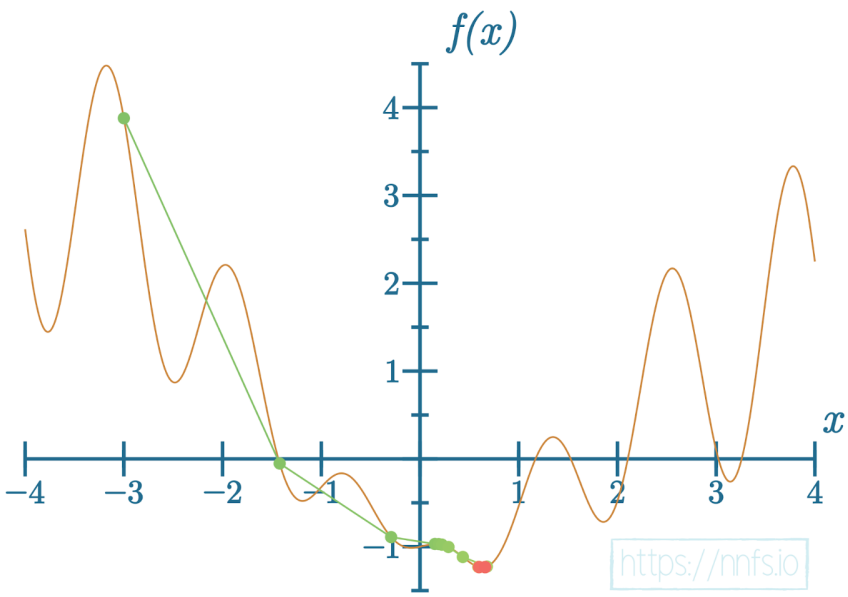


Fig 10.13: An efficient learning example.

Anim 10.13: https://nnfs.io/rog

This time the model needed just a few steps to find the global minimum. The challenge is to choose the hyper-parameters correctly, and it is not always an easy task. It is usually best to start with the optimizer defaults, perform a few steps, and observe the training process when tuning different settings. It is not always possible to see meaningful results in a short-enough period of time, and, in this case, it's good to have the ability to update the optimizer's settings during training. How you choose the learning rate, and other hyper-parameters, depends on the model, data, including the amount of data, the parameter initialization method, etc. There is no single, best way to set hyper-parameters, but experience usually helps. As we mentioned, one of the challenges during the training of a neural network model is to choose the right settings. The

difference can be anything from a model not learning at all to learning very well.

For a summary of learning rates — if we plot the loss along an axis of steps:

Fig 10.14: Graphs of the loss in a function of steps, different rates

We can see various examples of relative learning rates and what loss will ideally look like as a graph over time (steps) of training.

Knowing what the learning rate should be to get the most out of your training process isn't possible, but a good rule is that your initial training will benefit from a larger learning rate to take initial steps faster. If you start with steps that are too small, you might get stuck in a local minimum and be unable to leave it due to not making large enough updates to the parameters. For example, what if we make the learning rate 0.85 rather than 1.0 with the SGD optimizer?

```
# Create dataset
X, y = spiral_data(samples=100, classes=3)

# Create Dense layer with 2 input features and 64 output values
dense1 = Layer_Dense(2, 64)

# Create ReLU activation (to be used with Dense layer):
activation1 = Activation_ReLU()
```

```
# Create second Dense layer with 64 input features (as we take output
# of previous layer here) and 3 output values (output values)
dense2 = Layer_Dense(64, 3)

# Create Softmax classifier's combined loss and activation
loss_activation = Activation_Softmax_Loss_CategoricalCrossentropy()

# Create optimizer
optimizer = Optimizer_SGD(learning_rate=.85)

# Train in loop
for epoch in range(10001):

    # Perform a forward pass of our training data through this layer
    dense1.forward(X)

    # Perform a forward pass through activation function
    # takes the output of first dense layer here
    activation1.forward(dense1.output)

    # Perform a forward pass through second Dense layer
    # takes outputs of activation function of first layer as inputs
    dense2.forward(activation1.output)

    # Perform a forward pass through the activation/loss function
    # takes the output of second dense layer here and returns loss
    loss = loss_activation.forward(dense2.output, y)

    # Calculate accuracy from output of activation2 and targets
    # calculate values along first axis
    predictions = np.argmax(loss_activation.output, axis=1)
    if len(y.shape) == 2:
        y = np.argmax(y, axis=1)
    accuracy = np.mean(predictions==y)

    if not epoch % 100:
        print(f'epoch: {epoch}, ' +
              f'acc: {accuracy:.3f}, ' +
              f'loss: {loss:.3f}')

    # Backward pass
    loss_activation.backward(loss_activation.output, y)
    dense2.backward(loss_activation.dinputs)
    activation1.backward(dense2.dinputs)
    dense1.backward(activation1.dinputs)

    # Update weights and biases
    optimizer.update_params(dense1)
    optimizer.update_params(dense2)
```

```
>>>
epoch: 0, acc: 0.360, loss: 1.099
epoch: 100, acc: 0.403, loss: 1.091
...
epoch: 2000, acc: 0.437, loss: 1.053
epoch: 2100, acc: 0.443, loss: 1.026
epoch: 2200, acc: 0.377, loss: 1.050
epoch: 2300, acc: 0.433, loss: 1.016
epoch: 2400, acc: 0.460, loss: 1.000
epoch: 2500, acc: 0.493, loss: 1.010
epoch: 2600, acc: 0.527, loss: 0.998
epoch: 2700, acc: 0.523, loss: 0.977
...
epoch: 7100, acc: 0.577, loss: 0.941
epoch: 7200, acc: 0.550, loss: 0.921
epoch: 7300, acc: 0.593, loss: 0.943
epoch: 7400, acc: 0.593, loss: 0.940
epoch: 7500, acc: 0.557, loss: 0.907
epoch: 7600, acc: 0.590, loss: 0.949
epoch: 7700, acc: 0.590, loss: 0.935
...
epoch: 9100, acc: 0.597, loss: 0.860
epoch: 9200, acc: 0.630, loss: 0.842
...
epoch: 10000, acc: 0.657, loss: 0.816
```

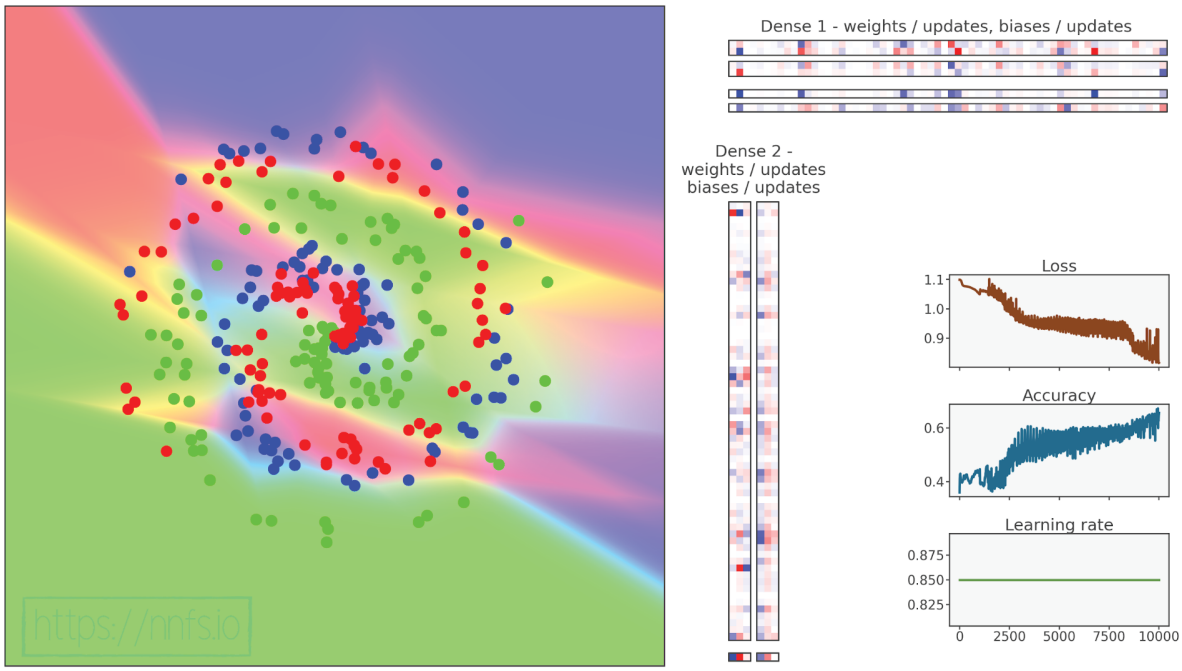


Fig 10.15: Model training with SGD optimizer and lowered learning rate.

Epilepsy Warning (quick flashing colors).

Anim 10.15: https://nnfs.io/cup

As you can see, the neural network did slightly better in terms of accuracy, and it achieved a lower loss; lower loss is not always associated with higher accuracy. Remember, even if we desire the best accuracy out of our model, the optimizer's task is to decrease loss, not raise accuracy directly. Loss is the mean value of all of the sample losses, and some of them could drop significantly, while others might rise just slightly, changing the prediction for them from a correct to an incorrect class at the same time. This would cause a lower mean loss in general, but also more incorrectly predicted samples, which will, at the same time, lower the accuracy. A likely reason for this model's lower accuracy is that it found another local minimum by chance — the descent path has changed, due to smaller steps. In a direct comparison of these two models in training, different learning rates did not show that the lower this learning rate value is, the better. In most cases, we want to start with a larger learning rate and decrease the learning rate over time/steps.

A commonly-used solution to keep initial updates large and explore various learning rates during training is to implement a **learning rate decay**.

Learning Rate Decay

The idea of a **learning rate decay** is to start with a large learning rate, say 1.0 in our case, and then decrease it during training. There are a few methods for doing this. One is to decrease the learning rate in response to the loss across epochs — for example, if the loss begins to level out/plateau or starts "jumping" over large deltas. You can either program this behavior-monitoring logically or simply track your loss over time and manually decrease the learning rate when you deem it appropriate. Another option, which we will implement, is to program a **Decay Rate**, which steadily decays the learning rate per batch or epoch.

Let's plan to decay per step. This can also be referred to as **1/t decaying** or **exponential decaying**. Basically, we're going to update the learning rate each step by the reciprocal of the step count fraction. This fraction is a new hyper-parameter that we'll add to the optimizer, called the **learning rate decay**. How this decaying works is it takes the step and the decaying ratio and multiplies them. The further in training, the bigger the step is, and the bigger result of this multiplication is. We then take its reciprocal (the further in training, the lower the value) and multiply the initial learning rate by it. The added *1* makes sure that the resulting algorithm never raises the learning rate. For example, for the first step, we might divide 1 by the learning rate, *0.001* for example, which will result in a current learning rate of *1000*. That's definitely not what we wanted. 1 divided by the 1+fraction ensures that the result, a fraction of the starting learning rate, will always be less than or equal to 1, decreasing over time. That's the desired result — start with the current learning rate and make it smaller with time. The code for determining the current decay rate:

```
starting_learning_rate = 1.
learning_rate_decay = 0.1
step = 1

learning_rate = starting_learning_rate * \
                (1. / (1 + learning_rate_decay * step))
print(learning_rate)
```

```
>>>
0.9090909090909091
```

In practice, 0.1 would be considered a fairly aggressive decay rate, but this should give you a sense of the concept. If we are on step 20:

```
starting_learning_rate = 1.
learning_rate_decay = 0.1
step = 20

learning_rate = starting_learning_rate * \
                (1. / (1 + learning_rate_decay * step))
print(learning_rate)
```

```
>>>
0.3333333333333333
```

We can also simulate this in a loop, which is more comparable to how we will be applying learning rate decay:

```
starting_learning_rate = 1.
learning_rate_decay = 0.1

for step in range(20):
    learning_rate = starting_learning_rate * \
                    (1. / (1 + learning_rate_decay * step))
    print(learning_rate)
```

```
>>>
1.0
0.9090909090909091
0.8333333333333334
0.7692307692307692
0.7142857142857143
0.6666666666666666
0.625
0.588235294117647
0.5555555555555556
0.5263157894736842
0.5
0.47619047619047616
0.45454545454545453
0.4347826086956522
0.41666666666666663
0.4
0.3846153846153846
0.37037037037037035
0.35714285714285715
0.3448275862068965
```

This learning rate decay scheme lowers the learning rate each step using the mentioned formula. Initially, the learning rate drops fast, but the change in the learning rate lowers each step, letting the model sit as close as possible to the minimum. The model needs small updates near the end of training to be able to get as close to this point as possible. We can now update our SGD optimizer class to allow for the learning rate decay:

```
# SGD optimizer
class Optimizer_SGD:

    # Initialize optimizer - set settings,
    # learning rate of 1. is default for this optimizer
    def __init__(self, learning_rate=1., decay=0.):
        self.learning_rate = learning_rate
        self.current_learning_rate = learning_rate
        self.decay = decay
        self.iterations = 0

    # Call once before any parameter updates
    def pre_update_params(self):
        if self.decay:
            self.current_learning_rate = self.learning_rate * \
                (1. / (1. + self.decay * self.iterations))

    # Update parameters
    def update_params(self, layer):
        layer.weights += -self.current_learning_rate * layer.dweights
        layer.biases += -self.current_learning_rate * layer.dbiases

    # Call once after any parameter updates
    def post_update_params(self):
        self.iterations += 1
```

We've updated a few things in the SGD class. First, in the `__init__` method, we added handling for the current learning rate, and `self.learning_rate` is now the initial learning rate. We also added attributes to track the decay rate and the number of iterations that the optimizer has gone through. Next, we added a new method called `pre_update_params`. This method, if we have a decay rate other than 0, will update our `self.current_learning_rate` using the prior formula. The `update_params` method remains unchanged, but we do have a new `post_update_params` method that will add to our `self.iterations` tracking. With our updated SGD optimizer class, we've added printing the current learning rate, and added pre and post optimizer method calls. Let's use a decay rate of 1e-2 (0.01) and train our model again:

```
# Create dataset
X, y = spiral_data(samples=100, classes=3)

# Create Dense layer with 2 input features and 64 output values
dense1 = Layer_Dense(2, 64)

# Create ReLU activation (to be used with Dense layer):
activation1 = Activation_ReLU()

# Create second Dense layer with 64 input features (as we take output
# of previous layer here) and 3 output values (output values)
dense2 = Layer_Dense(64, 3)

# Create Softmax classifier's combined loss and activation
loss_activation = Activation_Softmax_Loss_CategoricalCrossentropy()

# Create optimizer
optimizer = Optimizer_SGD(decay=1e-2)

# Train in loop
for epoch in range(10001):

    # Perform a forward pass of our training data through this layer
    dense1.forward(X)

    # Perform a forward pass through activation function
    # takes the output of first dense layer here
    activation1.forward(dense1.output)

    # Perform a forward pass through second Dense layer
    # takes outputs of activation function of first layer as inputs
    dense2.forward(activation1.output)

    # Perform a forward pass through the activation/loss function
    # takes the output of second dense layer here and returns loss
    loss = loss_activation.forward(dense2.output, y)

    # Calculate accuracy from output of activation2 and targets
    # calculate values along first axis
    predictions = np.argmax(loss_activation.output, axis=1)
    if len(y.shape) == 2:
        y = np.argmax(y, axis=1)
    accuracy = np.mean(predictions==y)

    if not epoch % 100:
        print(f'epoch: {epoch}, ' +
              f'acc: {accuracy:.3f}, ' +
              f'loss: {loss:.3f}, ' +
              f'lr: {optimizer.current_learning_rate}')
```

```
    # Backward pass
    loss_activation.backward(loss_activation.output, y)
    dense2.backward(loss_activation.dinputs)
    activation1.backward(dense2.dinputs)
    dense1.backward(activation1.dinputs)

    # Update weights and biases
    optimizer.pre_update_params()
    optimizer.update_params(dense1)
    optimizer.update_params(dense2)
    optimizer.post_update_params()
```

```
>>>
epoch: 0, acc: 0.360, loss: 1.099, lr: 1.0
epoch: 100, acc: 0.403, loss: 1.095, lr: 0.5025125628140703
epoch: 200, acc: 0.397, loss: 1.084, lr: 0.33444816053511706
epoch: 300, acc: 0.400, loss: 1.080, lr: 0.2506265664160401
epoch: 400, acc: 0.407, loss: 1.078, lr: 0.2004008016032064
epoch: 500, acc: 0.420, loss: 1.078, lr: 0.1669449081803005
epoch: 600, acc: 0.420, loss: 1.077, lr: 0.14306151645207438
epoch: 700, acc: 0.417, loss: 1.077, lr: 0.1251564455569462
epoch: 800, acc: 0.413, loss: 1.077, lr: 0.11123470522803114
epoch: 900, acc: 0.410, loss: 1.077, lr: 0.10010010010010009
epoch: 1000, acc: 0.417, loss: 1.077, lr: 0.09099181073703366
...
epoch: 2000, acc: 0.420, loss: 1.076, lr: 0.047641734159123386
...
epoch: 3000, acc: 0.413, loss: 1.075, lr: 0.03226847370119393
...
epoch: 4000, acc: 0.407, loss: 1.075, lr: 0.02439619419370578
...
epoch: 5000, acc: 0.403, loss: 1.074, lr: 0.019611688566385566
...
epoch: 7000, acc: 0.400, loss: 1.073, lr: 0.014086491055078181
...
epoch: 10000, acc: 0.397, loss: 1.072, lr: 0.009901970492127933
```

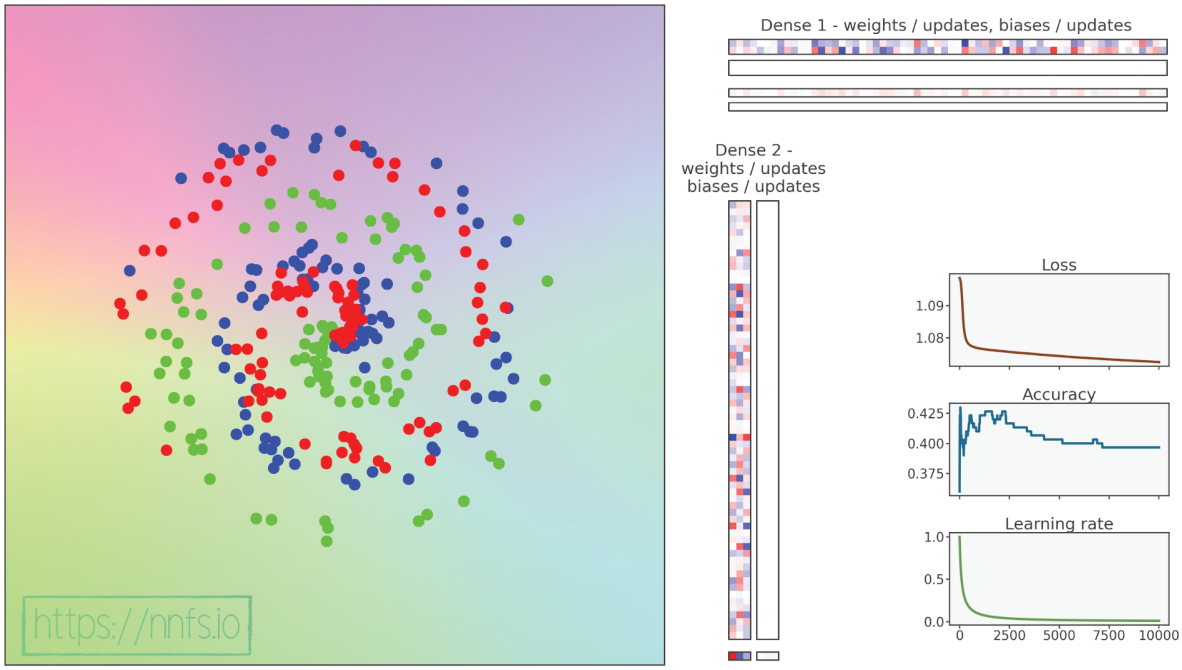


Fig 10.16: Model training with SGD optimizer and and learning rate decay set too high.

Epilepsy Warning (quick flashing colors)

Anim 10.16: https://nnfs.io/zuk

This model definitely got stuck, and the reason is almost certainly because the learning rate decayed far too quickly and became too small, trapping the model in some local minimum. This is most likely why, rather than wiggling, our accuracy and loss stopped changing *at all*.

We can, instead, try to decay a bit slower by making our decay a smaller number. For example, let's go with 1e-3 (0.001):

```
# Create dataset
X, y = spiral_data(samples=100, classes=3)

# Create Dense layer with 2 input features and 64 output values
dense1 = Layer_Dense(2, 64)

# Create ReLU activation (to be used with Dense layer):
activation1 = Activation_ReLU()

# Create second Dense layer with 64 input features (as we take output
# of previous layer here) and 3 output values (output values)
dense2 = Layer_Dense(64, 3)

# Create Softmax classifier's combined loss and activation
loss_activation = Activation_Softmax_Loss_CategoricalCrossentropy()

# Create optimizer
optimizer = Optimizer_SGD(decay=1e-3)

# Train in loop
for epoch in range(10001):

    # Perform a forward pass of our training data through this layer
    dense1.forward(X)

    # Perform a forward pass through activation function
    # takes the output of first dense layer here
    activation1.forward(dense1.output)

    # Perform a forward pass through second Dense layer
    # takes outputs of activation function of first layer as inputs
    dense2.forward(activation1.output)

    # Perform a forward pass through the activation/loss function
    # takes the output of second dense layer here and returns loss
    loss = loss_activation.forward(dense2.output, y)

    # Calculate accuracy from output of activation2 and targets
    # calculate values along first axis
    predictions = np.argmax(loss_activation.output, axis=1)
    if len(y.shape) == 2:
        y = np.argmax(y, axis=1)
    accuracy = np.mean(predictions==y)

    if not epoch % 100:
        print(f'epoch: {epoch}, ' +
              f'acc: {accuracy:.3f}, ' +
              f'loss: {loss:.3f}, ' +
              f'lr: {optimizer.current_learning_rate}')
```

```
    # Backward pass
    loss_activation.backward(loss_activation.output, y)
    dense2.backward(loss_activation.dinputs)
    activation1.backward(dense2.dinputs)
    dense1.backward(activation1.dinputs)

    # Update weights and biases
    optimizer.pre_update_params()
    optimizer.update_params(dense1)
    optimizer.update_params(dense2)
    optimizer.post_update_params()
```

```
>>>
epoch: 0, acc: 0.360, loss: 1.099, lr: 1.0
epoch: 100, acc: 0.400, loss: 1.088, lr: 0.9099181073703367
epoch: 200, acc: 0.423, loss: 1.078, lr: 0.8340283569641367
...
epoch: 1700, acc: 0.450, loss: 1.025, lr: 0.3705075954057058
epoch: 1800, acc: 0.470, loss: 1.017, lr: 0.35727045373347627
epoch: 1900, acc: 0.460, loss: 1.008, lr: 0.3449465332873405
epoch: 2000, acc: 0.463, loss: 1.000, lr: 0.33344448149383127
epoch: 2100, acc: 0.490, loss: 1.005, lr: 0.32268473701193934
...
epoch: 3200, acc: 0.493, loss: 0.983, lr: 0.23815194093831865
...
epoch: 5000, acc: 0.577, loss: 0.900, lr: 0.16669444907484582
...
epoch: 6000, acc: 0.633, loss: 0.860, lr: 0.1428775539362766
...
epoch: 8000, acc: 0.647, loss: 0.799, lr: 0.11112345816201799
...
epoch: 9800, acc: 0.663, loss: 0.773, lr: 0.09260116677470137
epoch: 9900, acc: 0.663, loss: 0.772, lr: 0.09175153683824203
epoch: 10000, acc: 0.667, loss: 0.771, lr: 0.09091735612328393
```

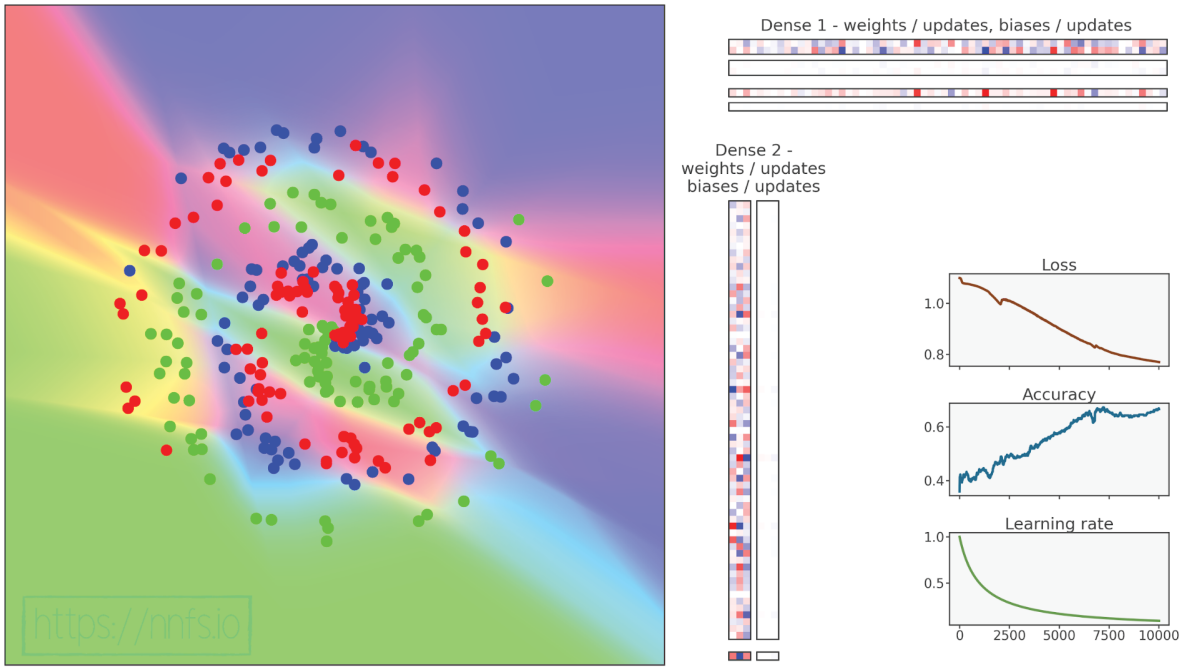


Fig 10.17: Model training with SGD optimizer and more proper learning rate decay.

Epilepsy Warning (quick flashing colors)

Anim 10.17: https://nnfs.io/muk

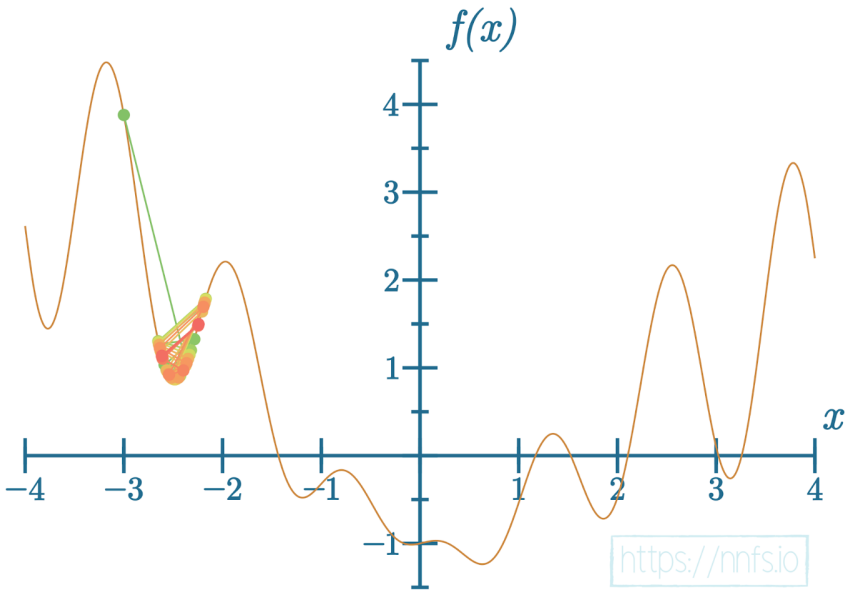
In this case, we've achieved our lowest loss and highest accuracy thus far, but it still should be possible to find parameters that will give us even better results. For example, you may suspect that the initial learning rate is too high. It can make for a great exercise to attempt to find better settings. Feel free to try!

Stochastic Gradient Descent with learning rate decay can do fairly well but is still a fairly basic optimization method that only follows a gradient without any additional logic that could potentially help the model find the **global minimum** to the loss function. One option for improving the SGD optimizer is to introduce **momentum**.

Stochastic Gradient Descent with Momentum

Momentum creates a rolling average of gradients over some number of updates and uses this average with the unique gradient at each step. Another way of understanding this is to imagine a ball going down a hill — even if it finds a small hole or hill, momentum will let it go straight through it towards a lower minimum — the bottom of this hill. This can help in cases where you're stuck in some local minimum (a hole), bouncing back and forth. With momentum, a model is more likely to pass through local minimums, further decreasing loss. Simply put, momentum may still point towards the global gradient descent direction.

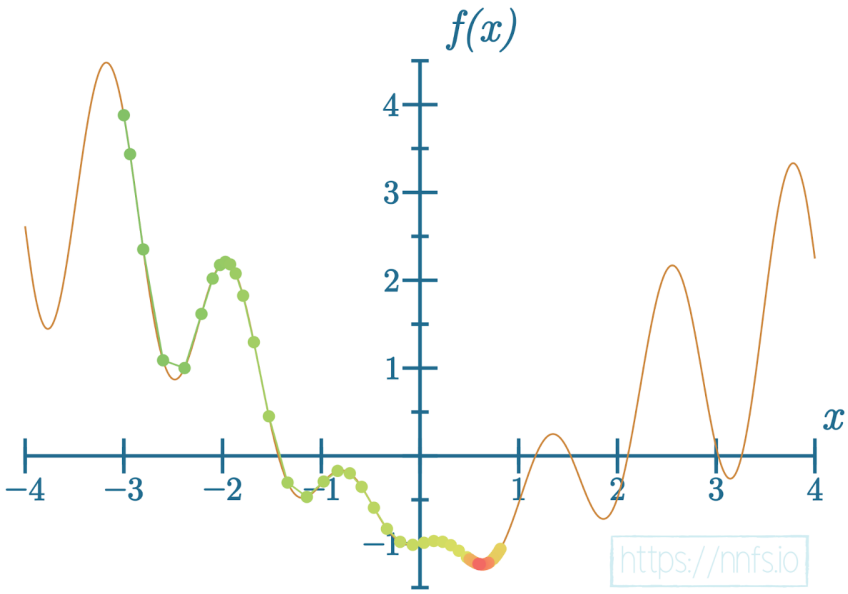
Recall this situation from the beginning of this chapter:



With regular updates, the SGD optimizer might determine that the next best step is one that keeps the model in a local minimum. Remember that the gradient points toward the current steepest loss ascent for that step — taking the negative of the gradient vector flips it toward the current steepest descent, which may not necessarily follow descent towards the global minimum — the current steepest descent may point towards a local minimum. So this step may decrease loss for that update but might not get us out of the local minimum. We might wind up with a gradient

that points in one direction and then the opposite direction in the next update; the gradient could continue to bounce back and forth around a local minimum like this, keeping the optimization of the loss stuck. Instead, momentum uses the previous update's direction to influence the next update's direction, minimizing the chances of bouncing around and getting stuck.

Recall another example shown in this chapter:



We utilize momentum by setting a parameter between 0 and 1, representing the fraction of the previous parameter update to retain, and subtracting (adding the negative) our actual gradient, multiplied by the learning rate (like before), from it. The update contains a portion of the gradient from preceding steps as our momentum (direction of previous changes) and only a portion of the current gradient; together, these portions form the actual change to our parameters and the bigger the role that momentum takes in the update, the slower the update can change the direction. When we set the momentum fraction too high, the model might stop learning at all since the direction of the updates won't be able to follow the global gradient descent. The code for this is as follows:

```
weight_updates = self.momentum * layer.weight_momentums - \
                 self.current_learning_rate * layer.dweights
```

The hyperparameter, `self.momentum`, is chosen at the start and the `layer.weight_momentums` start as all zeros but are altered during training as:

```
layer.weight_momentums = weight_updates
```

This means that the momentum is always the previous update to the parameters. We will perform the same operations as the above with the biases. We can then update our SGD optimizer class' `update_params` method with the momentum calculation, applying with the parameters, and retaining them for the next steps as an alternative chain of operations to the current code. The difference is that we only calculate the updates and we add these updates with the common code:

```
# Update parameters
def update_params(self, layer):

    # If we use momentum
    if self.momentum:

        # If layer does not contain momentum arrays, create them
        # filled with zeros
        if not hasattr(layer, 'weight_momentums'):
            layer.weight_momentums = np.zeros_like(layer.weights)
            # If there is no momentum array for weights
            # The array doesn't exist for biases yet either.
            layer.bias_momentums = np.zeros_like(layer.biases)

        # Build weight updates with momentum - take previous
        # updates multiplied by retain factor and update with
        # current gradients
        weight_updates = \
            self.momentum * layer.weight_momentums - \
            self.current_learning_rate * layer.dweights
        layer.weight_momentums = weight_updates

        # Build bias updates
        bias_updates = \
            self.momentum * layer.bias_momentums - \
            self.current_learning_rate * layer.dbiases
        layer.bias_momentums = bias_updates

    # Vanilla SGD updates (as before momentum update)
    else:
        weight_updates = -self.current_learning_rate * \
                         layer.dweights
        bias_updates = -self.current_learning_rate * \
                       layer.dbiases

    # Update weights and biases using either
    # vanilla or momentum updates
    layer.weights += weight_updates
    layer.biases += bias_updates
```

Making our full SGD optimizer class:

```
# SGD optimizer
class Optimizer_SGD:

    # Initialize optimizer - set settings,
    # learning rate of 1. is default for this optimizer
    def __init__(self, learning_rate=1., decay=0., momentum=0.):
        self.learning_rate = learning_rate
        self.current_learning_rate = learning_rate
        self.decay = decay
        self.iterations = 0
        self.momentum = momentum

    # Call once before any parameter updates
    def pre_update_params(self):
        if self.decay:
            self.current_learning_rate = self.learning_rate * \
                (1. / (1. + self.decay * self.iterations))

    # Update parameters
    def update_params(self, layer):

        # If we use momentum
        if self.momentum:

            # If layer does not contain momentum arrays, create them
            # filled with zeros
            if not hasattr(layer, 'weight_momentums'):
                layer.weight_momentums = np.zeros_like(layer.weights)
                # If there is no momentum array for weights
                # The array doesn't exist for biases yet either.
                layer.bias_momentums = np.zeros_like(layer.biases)

            # Build weight updates with momentum - take previous
            # updates multiplied by retain factor and update with
            # current gradients
            weight_updates = \
                self.momentum * layer.weight_momentums - \
                self.current_learning_rate * layer.dweights
            layer.weight_momentums = weight_updates

            # Build bias updates
            bias_updates = \
                self.momentum * layer.bias_momentums - \
                self.current_learning_rate * layer.dbiases
            layer.bias_momentums = bias_updates
```

```
        # Vanilla SGD updates (as before momentum update)
        else:
            weight_updates = -self.current_learning_rate * \
                              layer.dweights
            bias_updates = -self.current_learning_rate * \
                            layer.dbiases

        # Update weights and biases using either
        # vanilla or momentum updates
        layer.weights += weight_updates
        layer.biases += bias_updates

    # Call once after any parameter updates
    def post_update_params(self):
        self.iterations += 1
```

Let's show an example illustrating how adding momentum changes the learning process. Keeping the same starting **learning rate** (1) and **decay** (1e-3) from the previous training attempt and using a momentum of 0.5:

```
# Create dataset
X, y = spiral_data(samples=100, classes=3)

# Create Dense layer with 2 input features and 64 output values
dense1 = Layer_Dense(2, 64)

# Create ReLU activation (to be used with Dense layer):
activation1 = Activation_ReLU()

# Create second Dense layer with 64 input features (as we take output
# of previous layer here) and 3 output values (output values)
dense2 = Layer_Dense(64, 3)

# Create Softmax classifier's combined loss and activation
loss_activation = Activation_Softmax_Loss_CategoricalCrossentropy()

# Create optimizer
optimizer = Optimizer_SGD(decay=1e-3, momentum=0.5)

# Train in loop
for epoch in range(10001):

    # Perform a forward pass of our training data through this layer
    dense1.forward(X)

    # Perform a forward pass through activation function
    # takes the output of first dense layer here
    activation1.forward(dense1.output)
```

```
    # Perform a forward pass through second Dense layer
    # takes outputs of activation function of first layer as inputs
    dense2.forward(activation1.output)

    # Perform a forward pass through the activation/loss function
    # takes the output of second dense layer here and returns loss
    loss = loss_activation.forward(dense2.output, y)

    # Calculate accuracy from output of activation2 and targets
    # calculate values along first axis
    predictions = np.argmax(loss_activation.output, axis=1)
    if len(y.shape) == 2:
        y = np.argmax(y, axis=1)
    accuracy = np.mean(predictions==y)

    if not epoch % 100:
        print(f'epoch: {epoch}, ' +
              f'acc: {accuracy:.3f}, ' +
              f'loss: {loss:.3f}, ' +
              f'lr: {optimizer.current_learning_rate}')

    # Backward pass
    loss_activation.backward(loss_activation.output, y)
    dense2.backward(loss_activation.dinputs)
    activation1.backward(dense2.dinputs)
    dense1.backward(activation1.dinputs)

    # Update weights and biases
    optimizer.pre_update_params()
    optimizer.update_params(dense1)
    optimizer.update_params(dense2)
    optimizer.post_update_params()
```

```
>>>
epoch: 0, acc: 0.360, loss: 1.099, lr: 1.0
epoch: 100, acc: 0.427, loss: 1.078, lr: 0.9099181073703367
epoch: 200, acc: 0.423, loss: 1.075, lr: 0.8340283569641367
...
epoch: 1800, acc: 0.483, loss: 0.978, lr: 0.35727045373347627
epoch: 1900, acc: 0.547, loss: 0.984, lr: 0.3449465332873405
...
epoch: 3100, acc: 0.593, loss: 0.883, lr: 0.2439619419370578
epoch: 3200, acc: 0.570, loss: 0.878, lr: 0.23815194093831865
epoch: 3300, acc: 0.563, loss: 0.863, lr: 0.23261223540358225
epoch: 3400, acc: 0.607, loss: 0.860, lr: 0.22732439190725165
...
epoch: 4600, acc: 0.670, loss: 0.761, lr: 0.1786033220217896
epoch: 4700, acc: 0.690, loss: 0.749, lr: 0.1754693805930865
```

```
...
epoch: 6000, acc: 0.743, loss: 0.661, lr: 0.1428775539362766
...
epoch: 8000, acc: 0.763, loss: 0.586, lr: 0.11112345816201799
...
epoch: 10000, acc: 0.800, loss: 0.539, lr: 0.09091735612328393
```

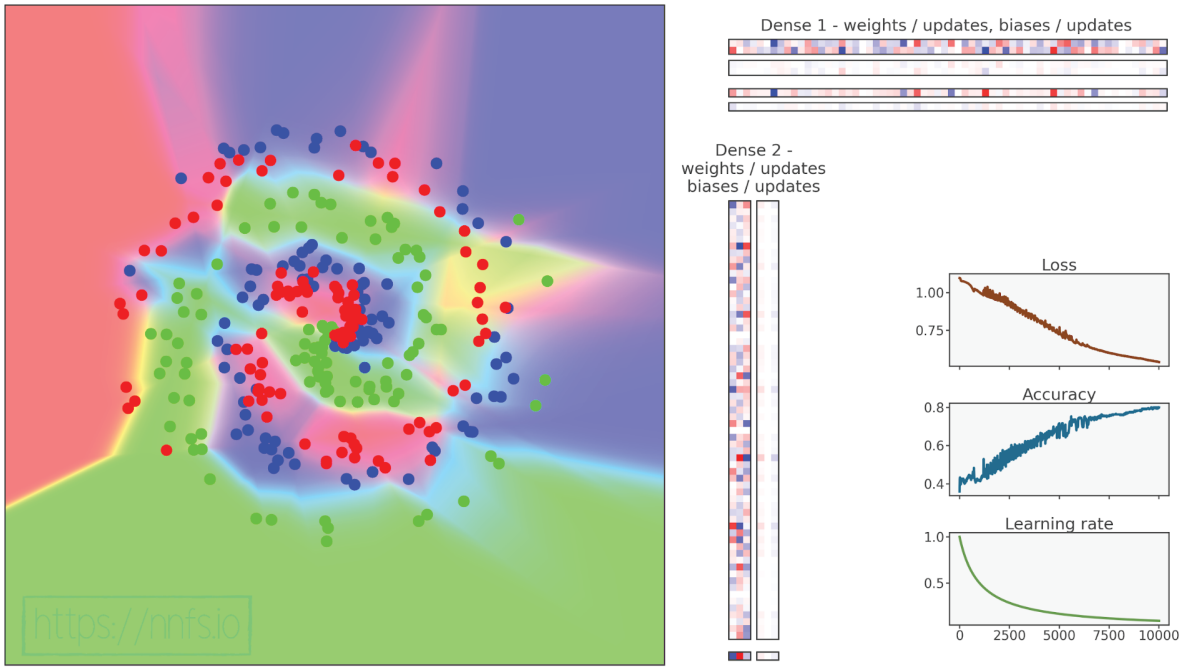


Fig 10.18: Model training with SGD optimizer, learning rate decay and Momentum.

Epilepsy Warning (quick flashing colors)

Anim 10.18: https://nnfs.io/ram

The model achieved the lowest loss and highest accuracy that we've seen so far, but can we do even better? Sure we can! Let's try to set the momentum to 0.9:

```
# Create dataset
X, y = spiral_data(samples=100, classes=3)

# Create Dense layer with 2 input features and 64 output values
dense1 = Layer_Dense(2, 64)

# Create ReLU activation (to be used with Dense layer):
activation1 = Activation_ReLU()

# Create second Dense layer with 64 input features (as we take output
# of previous layer here) and 3 output values (output values)
dense2 = Layer_Dense(64, 3)

# Create Softmax classifier's combined loss and activation
loss_activation = Activation_Softmax_Loss_CategoricalCrossentropy()

# Create optimizer
optimizer = Optimizer_SGD(decay=1e-3, momentum=0.9)

# Train in loop
for epoch in range(10001):

    # Perform a forward pass of our training data through this layer
    dense1.forward(X)

    # Perform a forward pass through activation function
    # takes the output of first dense layer here
    activation1.forward(dense1.output)

    # Perform a forward pass through second Dense layer
    # takes outputs of activation function of first layer as inputs
    dense2.forward(activation1.output)

    # Perform a forward pass through the activation/loss function
    # takes the output of second dense layer here and returns loss
    loss = loss_activation.forward(dense2.output, y)

    # Calculate accuracy from output of activation2 and targets
    # calculate values along first axis
    predictions = np.argmax(loss_activation.output, axis=1)
    accuracy = np.mean(predictions==y)
```

```
    if not epoch % 100:
        print(f'epoch: {epoch}, ' +
              f'acc: {accuracy:.3f}, ' +
              f'loss: {loss:.3f}, ' +
              f'lr: {optimizer.current_learning_rate}')

    # Backward pass
    loss_activation.backward(loss_activation.output, y)
    dense2.backward(loss_activation.dinputs)
    activation1.backward(dense2.dinputs)
    dense1.backward(activation1.dinputs)

    # Update weights and biases
    optimizer.pre_update_params()
    optimizer.update_params(dense1)
    optimizer.update_params(dense2)
    optimizer.post_update_params()
```

```
>>>
epoch: 0, acc: 0.360, loss: 1.099, lr: 1.0
epoch: 100, acc: 0.443, loss: 1.053, lr: 0.9099181073703367
epoch: 200, acc: 0.497, loss: 0.999, lr: 0.8340283569641367
epoch: 300, acc: 0.603, loss: 0.810, lr: 0.7698229407236336
epoch: 400, acc: 0.700, loss: 0.700, lr: 0.7147962830593281
epoch: 500, acc: 0.750, loss: 0.595, lr: 0.66711140760507
epoch: 600, acc: 0.810, loss: 0.496, lr: 0.6253908692933083
epoch: 700, acc: 0.810, loss: 0.466, lr: 0.5885815185403178
epoch: 800, acc: 0.847, loss: 0.384, lr: 0.5558643690939411
epoch: 900, acc: 0.850, loss: 0.364, lr: 0.526592943654555
epoch: 1000, acc: 0.877, loss: 0.344, lr: 0.5002501250625312
...
epoch: 2200, acc: 0.900, loss: 0.242, lr: 0.31259768677711786
...
epoch: 2900, acc: 0.910, loss: 0.216, lr: 0.25647601949217746
...
epoch: 3800, acc: 0.920, loss: 0.202, lr: 0.20837674515524068
...
epoch: 7100, acc: 0.930, loss: 0.181, lr: 0.12347203358439313
...
epoch: 10000, acc: 0.933, loss: 0.173, lr: 0.09091735612328393
```

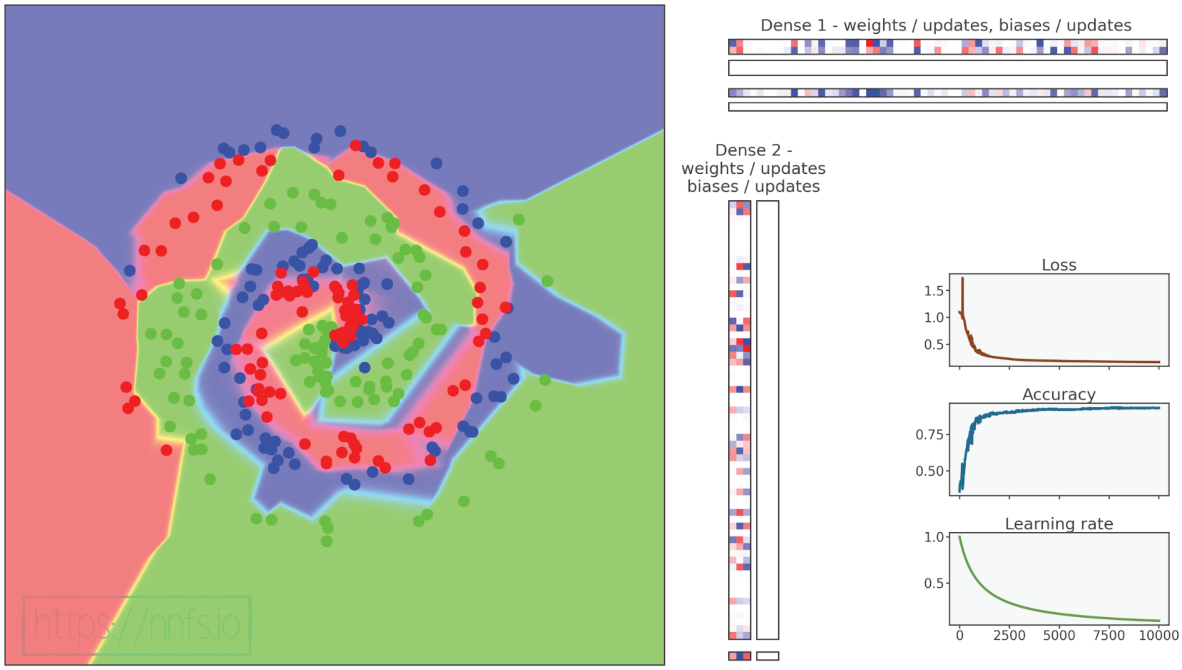


Fig 10.19: Model training with SGD optimizer, learning rate decay and Momentum (tuned).

Epilepsy Warning (quick flashing colors)

Anim 10.19: https://nnfs.io/map

This is a decent enough example of how momentum can prove useful. The model achieved an accuracy of almost 88% in the first 1000 epochs and improved further, ending with an accuracy of 93.3% and a loss of 0.173. These results are a great improvement. The SGD optimizer with momentum is usually one of 2 main choices for an optimizer in practice next to the Adam optimizer, which we'll talk about shortly. First, we have 2 other optimizers to talk about. The next modification to Stochastic Gradient Descent is **AdaGrad**.

AdaGrad

AdaGrad, short for **adaptive gradient**, institutes a per-parameter learning rate rather than a globally-shared rate. The idea here is to normalize updates made to the features. During the training process, some weights can rise significantly, while others tend to not change by much. It is usually better for weights to not rise too high compared to the other weights, and we'll talk about this with regularization techniques. AdaGrad provides a way to normalize parameter updates by keeping a history of previous updates — the bigger the sum of the updates is, in either direction (positive or negative), the smaller updates are made further in training. This lets less-frequently updated parameters to keep-up with changes, effectively utilizing more neurons for training. The concept of AdaGrad can be contained in the following two lines of code:

```
cache += parm_gradient ** 2
parm_updates = learning_rate * parm_gradient / (sqrt(cache) + eps)
```

The `cache` holds a history of squared gradients, and the `parm_updates` is a function of the learning rate multiplied by the gradient (basic SGD so far) and then is divided by the square root of the cache plus some **epsilon** value. The division operation performed with a constantly rising cache might also cause the learning to stall as updates become smaller with time, due to the monotonic nature of updates. That's why this optimizer is not widely used, except for some specific applications. The **epsilon** is a **hyperparameter** (pre-training control knob setting) preventing division by 0. The epsilon value is usually a small value, such as *1e-7*, which we'll be defaulting to. You might also notice that we are summing the squared value, only to calculate the square root later, which might look counter-intuitive as to why we do this. We are adding squared values and taking the square root, which is not the same as just adding the value, for example:

$$\sqrt{1^2 + 3^2} = \sqrt{1 + 9} = \sqrt{10} \approx 2.16$$

$$1 + 3 = 4$$

The resulting cache value grows slower, and in a different way, taking care of the negative numbers (we would not want to divide the update by the negative number and flip its sign). Overall, the impact is the learning rates for parameters with smaller gradients are decreased slowly, while the parameters with larger gradients have their learning rates decreased faster.

To implement AdaGrad, we start by copying and pasting our SGD optimizer class, changing the name, adding a property for **epsilon** with a default of 1e-7 to the `__init__` method, and removing the momentum. Next, inside the `update_params` method, we'll replace the momentum code with:

```
    # Update parameters
    def update_params(self, layer):

        # If layer does not contain cache arrays,
        # create them filled with zeros
        if not hasattr(layer, 'weight_cache'):
            layer.weight_cache = np.zeros_like(layer.weights)
            layer.bias_cache = np.zeros_like(layer.biases)

        # Update cache with squared current gradients
        layer.weight_cache += layer.dweights**2
        layer.bias_cache += layer.dbiases**2

        # Vanilla SGD parameter update + normalization
        # with square rooted cache
        layer.weights += -self.current_learning_rate * \
                         layer.dweights / \
                         (np.sqrt(layer.weight_cache) + self.epsilon)
        layer.biases += -self.current_learning_rate * \
                        layer.dbiases / \
                        (np.sqrt(layer.bias_cache) + self.epsilon)
```

We added the cache and its updates, then added dividing the updates by the square root of the cache. Full code for the AdaGrad optimizer:

```
# Adagrad optimizer
class Optimizer_Adagrad:

    # Initialize optimizer - set settings
    def __init__(self, learning_rate=1., decay=0., epsilon=1e-7):
        self.learning_rate = learning_rate
        self.current_learning_rate = learning_rate
        self.decay = decay
        self.iterations = 0
        self.epsilon = epsilon

    # Call once before any parameter updates
    def pre_update_params(self):
        if self.decay:
            self.current_learning_rate = self.learning_rate * \
                (1. / (1. + self.decay * self.iterations))
```

```
    # Update parameters
    def update_params(self, layer):

        # If layer does not contain cache arrays,
        # create them filled with zeros
        if not hasattr(layer, 'weight_cache'):
            layer.weight_cache = np.zeros_like(layer.weights)
            layer.bias_cache = np.zeros_like(layer.biases)

        # Update cache with squared current gradients
        layer.weight_cache += layer.dweights**2
        layer.bias_cache += layer.dbiases**2

        # Vanilla SGD parameter update + normalization
        # with square rooted cache
        layer.weights += -self.current_learning_rate * \
                         layer.dweights / \
                         (np.sqrt(layer.weight_cache) + self.epsilon)
        layer.biases += -self.current_learning_rate * \
                        layer.dbiases / \
                        (np.sqrt(layer.bias_cache) + self.epsilon)

    # Call once after any parameter updates
    def post_update_params(self):
        self.iterations += 1
```

Testing this optimizer now with decaying set to *1e-4* as well as *1e-5* works better than *1e-3*, which we have used previously. This optimizer with our dataset works better with lesser decaying:

```
# Create dataset
X, y = spiral_data(samples=100, classes=3)

# Create Dense layer with 2 input features and 64 output values
dense1 = Layer_Dense(2, 64)

# Create ReLU activation (to be used with Dense layer):
activation1 = Activation_ReLU()

# Create second Dense layer with 64 input features (as we take output
# of previous layer here) and 3 output values (output values)
dense2 = Layer_Dense(64, 3)

# Create Softmax classifier's combined loss and activation
loss_activation = Activation_Softmax_Loss_CategoricalCrossentropy()

# Create optimizer
#optimizer = Optimizer_SGD(decay=8e-8, momentum=0.9)
optimizer = Optimizer_Adagrad(decay=1e-4)
```

```
# Train in loop
for epoch in range(10001):

    # Perform a forward pass of our training data through this layer
    dense1.forward(X)

    # Perform a forward pass through activation function
    # takes the output of first dense layer here
    activation1.forward(dense1.output)

    # Perform a forward pass through second Dense layer
    # takes outputs of activation function of first layer as inputs
    dense2.forward(activation1.output)

    # Perform a forward pass through the activation/loss function
    # takes the output of second dense layer here and returns loss
    loss = loss_activation.forward(dense2.output, y)

    # Calculate accuracy from output of activation2 and targets
    # calculate values along first axis
    predictions = np.argmax(loss_activation.output, axis=1)
    if len(y.shape) == 2:
        y = np.argmax(y, axis=1)
    accuracy = np.mean(predictions==y)

    if not epoch % 100:
        print(f'epoch: {epoch}, ' +
              f'acc: {accuracy:.3f}, ' +
              f'loss: {loss:.3f}, ' +
              f'lr: {optimizer.current_learning_rate}')

    # Backward pass
    loss_activation.backward(loss_activation.output, y)
    dense2.backward(loss_activation.dinputs)
    activation1.backward(dense2.dinputs)
    dense1.backward(activation1.dinputs)

    # Update weights and biases
    optimizer.pre_update_params()
    optimizer.update_params(dense1)
    optimizer.update_params(dense2)
    optimizer.post_update_params()


>>>
epoch: 0, acc: 0.360, loss: 1.099, lr: 1.0
epoch: 100, acc: 0.457, loss: 1.012, lr: 0.9901970492127933
epoch: 200, acc: 0.527, loss: 0.936, lr: 0.9804882831650161
epoch: 300, acc: 0.600, loss: 0.874, lr: 0.9709680551509855
```

```
...
epoch: 1200, acc: 0.700, loss: 0.640, lr: 0.892936869363336
...
epoch: 1700, acc: 0.750, loss: 0.579, lr: 0.8547739123001966
...
epoch: 4700, acc: 0.800, loss: 0.464, lr: 0.6803183890060548
...
epoch: 5100, acc: 0.810, loss: 0.454, lr: 0.6622955162593549
...
epoch: 6700, acc: 0.820, loss: 0.426, lr: 0.5988382537876519
...
epoch: 7500, acc: 0.830, loss: 0.412, lr: 0.5714612263557918
...
epoch: 9900, acc: 0.847, loss: 0.381, lr: 0.5025378159706518
epoch: 10000, acc: 0.847, loss: 0.379, lr: 0.5000250012500626
```

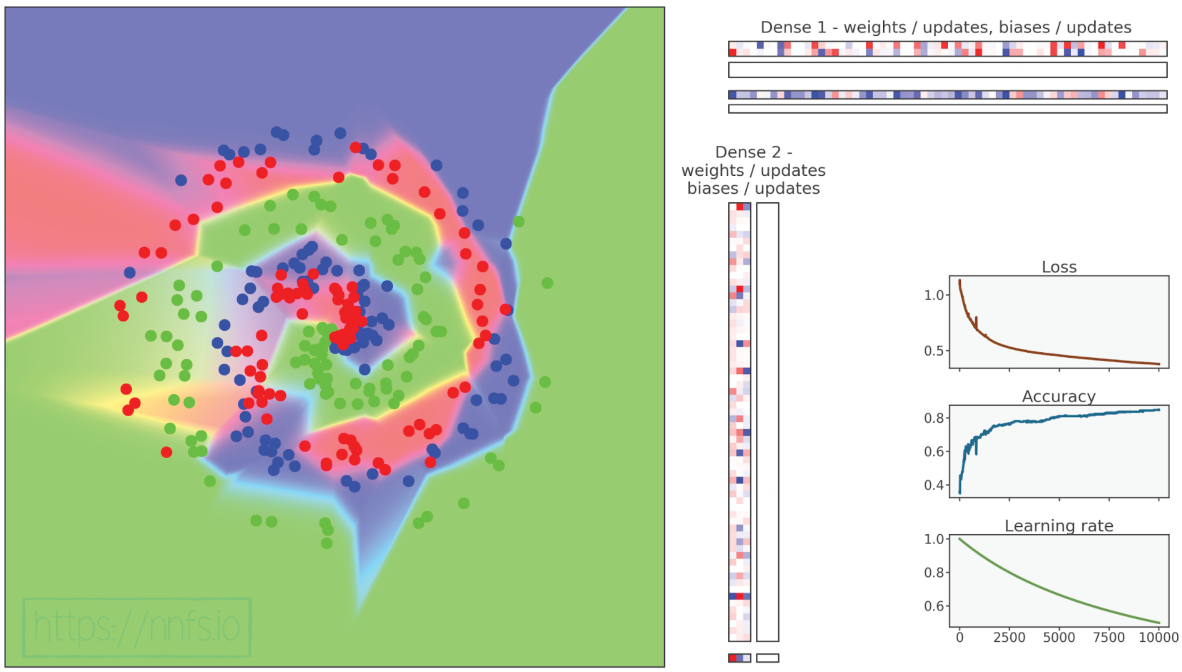


Fig 10.20: Model training with AdaGrad optimizer.

Epilepsy Warning (quick flashing colors)

Anim 10.20: https://nnfs.io/bop

AdaGrad worked quite well here, but not as good as SGD with momentum, and we can see that loss consistently fell throughout the entire training process. It is interesting to note that AdaGrad initially took a few more epochs to reach similar results to Stochastic Gradient Descent with momentum.

RMSProp

Continuing with Stochastic Gradient Descent adaptations, we reach **RMSProp**, short for **Root Mean Square Propagation**. Similar to AdaGrad, RMSProp calculates an adaptive learning rate per parameter; it's just calculated in a different way than AdaGrad.

Where AdaGrad calculates the cache as:

```
cache += gradient ** 2
```

RMSProp calculates the cache as:

```
cache = rho * cache + (1 - rho) * gradient ** 2
```

Note that this is similar to both momentum with the SGD optimizer and cache with the AdaGrad. RMSProp adds a mechanism similar to momentum but also adds a per-parameter adaptive learning rate, so the learning rate changes are smoother. This helps to retain the global direction of changes and slows changes in direction. Instead of continually adding squared gradients to a cache (like in Adagrad), it uses a moving average of the cache. Each update to the cache retains a part of the cache and updates it with a fraction of the new, squared, gradients. In this way, cache contents "move" with data in time, and learning does not stall. In the case of this optimizer, the per-parameter learning rate can either fall or rise, depending on the last updates and current gradient. RMSProp applies the cache in the same way as AdaGrad does.

The new hyperparameter here is *rho*. *Rho* is the cache memory decay rate. Because this optimizer, with default values, carries over so much momentum of gradient and the adaptive learning rate updates, even small gradient updates are enough to keep it going; therefore, a default learning rate of *1* is far too large and causes instant model instability. A learning rate that becomes stable again and gives fast enough updates is around *0.001* (that's also the default value for this optimizer used in well-known machine learning frameworks). That's what we'll use as default from now on too. The following is the full code for RMSProp optimizer class:

```
# RMSprop optimizer
class Optimizer_RMSprop:

    # Initialize optimizer - set settings
    def __init__(self, learning_rate=0.001, decay=0., epsilon=1e-7,
                 rho=0.9):
        self.learning_rate = learning_rate
        self.current_learning_rate = learning_rate
        self.decay = decay
        self.iterations = 0
        self.epsilon = epsilon
        self.rho = rho

    # Call once before any parameter updates
    def pre_update_params(self):
        if self.decay:
            self.current_learning_rate = self.learning_rate * \
                (1. / (1. + self.decay * self.iterations))

    # Update parameters
    def update_params(self, layer):

        # If layer does not contain cache arrays,
        # create them filled with zeros
        if not hasattr(layer, 'weight_cache'):
            layer.weight_cache = np.zeros_like(layer.weights)
            layer.bias_cache = np.zeros_like(layer.biases)

        # Update cache with squared current gradients
        layer.weight_cache = self.rho * layer.weight_cache + \
            (1 - self.rho) * layer.dweights**2
        layer.bias_cache = self.rho * layer.bias_cache + \
            (1 - self.rho) * layer.dbiases**2
```

```
        # Vanilla SGD parameter update + normalization
        # with square rooted cache
        layer.weights += -self.current_learning_rate * \
                         layer.dweights / \
                         (np.sqrt(layer.weight_cache) + self.epsilon)

        layer.biases += -self.current_learning_rate * \
                        layer.dbiases / \
                        (np.sqrt(layer.bias_cache) + self.epsilon)

    # Call once after any parameter updates
    def post_update_params(self):
        self.iterations += 1
```

Changing the optimizer used in our main neural network testing code:

```
optimizer = Optimizer_RMSprop(decay=1e-4)
```

And running this code gives us:

```
>>>
epoch: 0, acc: 0.360, loss: 1.099, lr: 0.001
epoch: 100, acc: 0.417, loss: 1.077, lr: 0.0009901970492127933
epoch: 200, acc: 0.457, loss: 1.072, lr: 0.0009804882831650162
epoch: 300, acc: 0.480, loss: 1.062, lr: 0.0009709680551509856
...
epoch: 1000, acc: 0.597, loss: 0.961, lr: 0.0009091735612328393
...
epoch: 4800, acc: 0.703, loss: 0.767, lr: 0.0006757213325224677
...
epoch: 5800, acc: 0.713, loss: 0.744, lr: 0.0006329514526235838
...
epoch: 7100, acc: 0.720, loss: 0.718, lr: 0.0005848295221942804
...
epoch: 10000, acc: 0.730, loss: 0.668, lr: 0.0005000250012500625
```

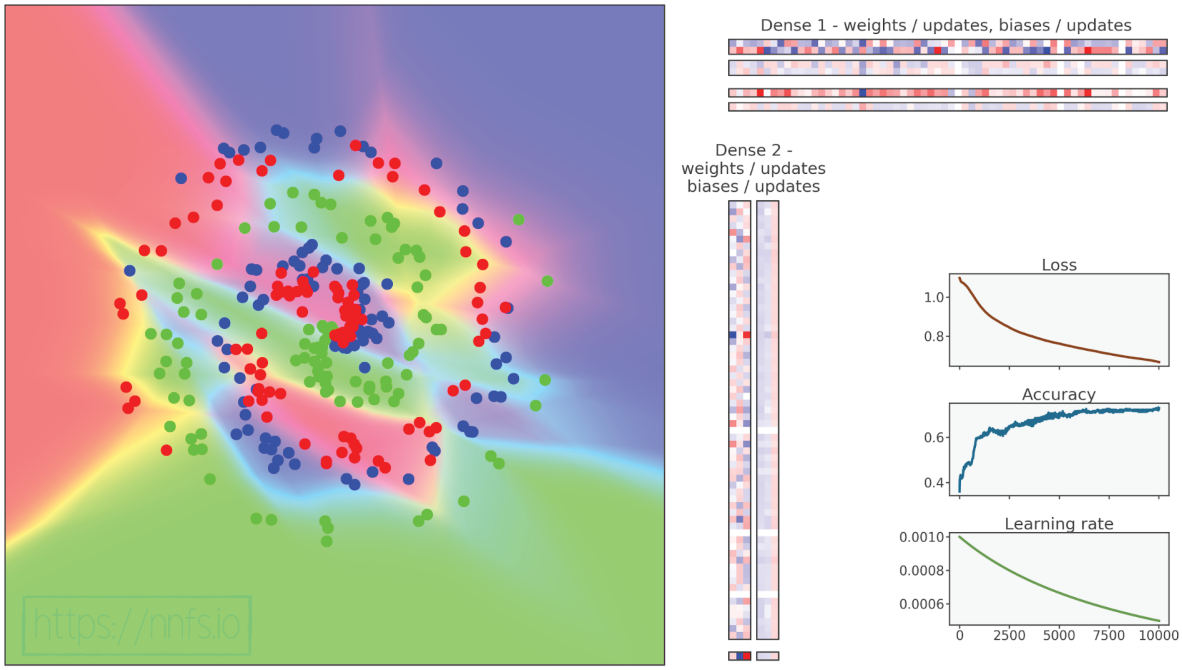


Fig 10.21: Model training with RMSProp optimizer.

Epilepsy Warning (quick flashing colors)

Anim 10.21: https://nnfs.io/pun

The results are not the greatest, but we can slightly tweak the hyperparameters:

```
optimizer = Optimizer_RMSprop(learning_rate=0.02, decay=1e-5,
                              rho=0.999)
```

```
>>>
epoch: 0, acc: 0.360, loss: 1.099, lr: 0.02
epoch: 100, acc: 0.467, loss: 1.014, lr: 0.01998021958261321
epoch: 200, acc: 0.530, loss: 0.959, lr: 0.019960279044701046
...
epoch: 600, acc: 0.623, loss: 0.762, lr: 0.019880913329158343
...
epoch: 1000, acc: 0.710, loss: 0.634, lr: 0.019802176259170884
...
epoch: 1800, acc: 0.810, loss: 0.475, lr: 0.01964655841412981
...
epoch: 3800, acc: 0.850, loss: 0.351, lr: 0.01926800836231563
...
epoch: 6200, acc: 0.870, loss: 0.286, lr: 0.018832569044906263
...
epoch: 6600, acc: 0.903, loss: 0.262, lr: 0.018761902081633034
...
epoch: 7100, acc: 0.900, loss: 0.274, lr: 0.018674310684506857
...
epoch: 9500, acc: 0.890, loss: 0.244, lr: 0.018265006986365174
epoch: 9600, acc: 0.893, loss: 0.241, lr: 0.018248341681949654
epoch: 9700, acc: 0.743, loss: 0.794, lr: 0.018231706761228456
epoch: 9800, acc: 0.917, loss: 0.213, lr: 0.018215102141185255
epoch: 9900, acc: 0.907, loss: 0.225, lr: 0.018198527739105907
epoch: 10000, acc: 0.910, loss: 0.221, lr: 0.018181983472577025
```

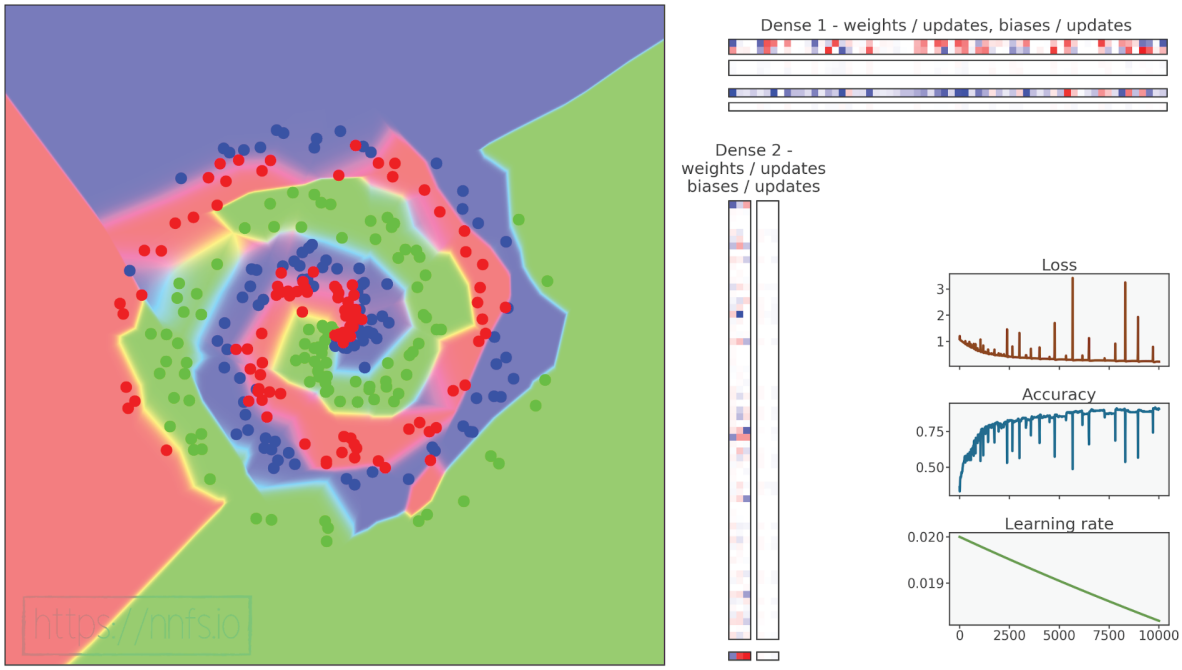


Fig 10.22: Model training with RMSProp optimizer (tuned).

Epilepsy Warning (quick flashing colors)

Anim 10.22: https://nnfs.io/not

Pretty good result, close to SGD with momentum but not as good. We still have one final adaptation to stochastic gradient descent to cover.

Adam

Adam, short for **Adaptive Momentum**, is currently the most widely-used optimizer and is built atop RMSProp, with the momentum concept from SGD added back in. This means that, instead of applying current gradients, we're going to apply momentums like in the SGD optimizer with momentum, then apply a per-weight adaptive learning rate with the cache as done in RMSProp.

The Adam optimizer additionally adds a bias correction mechanism. Do not confuse this with the layer's bias. The bias correction mechanism is applied to the cache and momentum, compensating for the initial zeroed values before they warm up with initial steps. To achieve this correction, both momentum and caches are divided by $1\text{-}beta^{step}$. As step raises, $beta^{step}$ approaches *0* (a fraction to the power of a rising value decreases), solving this whole expression to a fraction during the first steps and approaching *1* as training progresses. For example, *beta 1*, a fraction of momentum to apply, defaults to 0.9. This means that, during the first step, the correction value equals:

$$1 - 0.9^1 = 1 - 0.9 = 0.1$$

With training progression, as step count rises:

$$1 - \lim_{step \to \infty} 0.9^{step} = 1 - 0 = 1$$

The same applies to the cache and the *beta 2* — in this case, the starting value is 0.001 and also approaches *1*. These values divide the momentums and the cache, respectively. Division by a fraction causes them to be multiple times bigger, significantly speeding up training in the initial stages before both tables warm up during multiple initial steps. We also previously mentioned that both of these bias-correcting coefficients go towards a value of *1* as training progresses and return parameter updates to their typical values for the later training steps. To get parameter updates, we divide the scaled momentum by the scaled square-rooted cache.

The code for the Adam Optimizer is based on the RMSProp optimizer. It adds the cache seen from the SGD along with the *beta 1* hyper-parameter. Next, it introduces the bias correction mechanism for both the momentum and the cache. We've also modified the way the parameter updates are calculated — using corrected momentums and corrected caches, instead of gradients and caches. The full list of changes made from RMSProp are posted after the following code:

```
# Adam optimizer
class Optimizer_Adam:

    # Initialize optimizer - set settings
    def __init__(self, learning_rate=0.001, decay=0., epsilon=1e-7,
                 beta_1=0.9, beta_2=0.999):
        self.learning_rate = learning_rate
        self.current_learning_rate = learning_rate
        self.decay = decay
        self.iterations = 0
        self.epsilon = epsilon
        self.beta_1 = beta_1
        self.beta_2 = beta_2

    # Call once before any parameter updates
    def pre_update_params(self):
        if self.decay:
            self.current_learning_rate = self.learning_rate * \
                (1. / (1. + self.decay * self.iterations))

    # Update parameters
    def update_params(self, layer):

        # If layer does not contain cache arrays,
        # create them filled with zeros
        if not hasattr(layer, 'weight_cache'):
            layer.weight_momentums = np.zeros_like(layer.weights)
            layer.weight_cache = np.zeros_like(layer.weights)
            layer.bias_momentums = np.zeros_like(layer.biases)
            layer.bias_cache = np.zeros_like(layer.biases)

        # Update momentum  with current gradients
        layer.weight_momentums = self.beta_1 * \
                                 layer.weight_momentums + \
                                 (1 - self.beta_1) * layer.dweights
        layer.bias_momentums = self.beta_1 * \
                               layer.bias_momentums + \
                               (1 - self.beta_1) * layer.dbiases
        # Get corrected momentum
        # self.iteration is 0 at first pass
        # and we need to start with 1 here
        weight_momentums_corrected = layer.weight_momentums / \
            (1 - self.beta_1 ** (self.iterations + 1))
        bias_momentums_corrected = layer.bias_momentums / \
            (1 - self.beta_1 ** (self.iterations + 1))
        # Update cache with squared current gradients
        layer.weight_cache = self.beta_2 * layer.weight_cache + \
            (1 - self.beta_2) * layer.dweights**2
```

```
        layer.bias_cache = self.beta_2 * layer.bias_cache + \
            (1 - self.beta_2) * layer.dbiases**2
        # Get corrected cache
        weight_cache_corrected = layer.weight_cache / \
            (1 - self.beta_2 ** (self.iterations + 1))
        bias_cache_corrected = layer.bias_cache / \
            (1 - self.beta_2 ** (self.iterations + 1))

        # Vanilla SGD parameter update + normalization
        # with square rooted cache
        layer.weights += -self.current_learning_rate * \
                         weight_momentums_corrected / \
                         (np.sqrt(weight_cache_corrected) +
                             self.epsilon)
        layer.biases += -self.current_learning_rate * \
                         bias_momentums_corrected / \
                         (np.sqrt(bias_cache_corrected) +
                             self.epsilon)

    # Call once after any parameter updates
    def post_update_params(self):
        self.iterations += 1
```

The following changes were made from copying the RMSProp class code:

1. renamed class from `Optimizer_RMSprop` to `Optimizer_Adam`
2. renamed the *rho* hyperparameter and property to *beta_2* in `__init__`
3. added *beta_1* hyperparameter and property in `__init__`
4. added *momentum* array creation in *update_params()*
5. added *momentum* calculation
6. renamed `self.rho` to `self.beta_2` with cache calculation code in `update_params`
7. added `*_corrected` variables as corrected momentums and weights
8. replaced `layer.dweights`, `layer.dbiases`, `layer.weight_cache`, and `layer.bias_cache` with corrected arrays of values in parameter updates with momentum arrays

Back to our main neural network code. We can now set our optimizer to Adam, run the code, and see what impact these changes had:

```
optimizer = Optimizer_Adam(learning_rate=0.02, decay=1e-5)
```

With our default settings, we end with:

```
>>>
epoch: 0, acc: 0.360, loss: 1.099, lr: 0.02
epoch: 100, acc: 0.683, loss: 0.772, lr: 0.01998021958261321
epoch: 200, acc: 0.793, loss: 0.560, lr: 0.019960279044701046
epoch: 300, acc: 0.850, loss: 0.458, lr: 0.019940378268975763
```

```
epoch: 400, acc: 0.873, loss: 0.374, lr: 0.01992051713662487
epoch: 500, acc: 0.897, loss: 0.321, lr: 0.01990069552930875
epoch: 600, acc: 0.893, loss: 0.286, lr: 0.019880913329158343
epoch: 700, acc: 0.900, loss: 0.260, lr: 0.019861170418772778
...
epoch: 1700, acc: 0.930, loss: 0.164, lr: 0.019665876753950384
...
epoch: 2600, acc: 0.950, loss: 0.132, lr: 0.019493367381748363
...
epoch: 9900, acc: 0.967, loss: 0.078, lr: 0.018198527739105907
epoch: 10000, acc: 0.963, loss: 0.079, lr: 0.018181983472577025
```

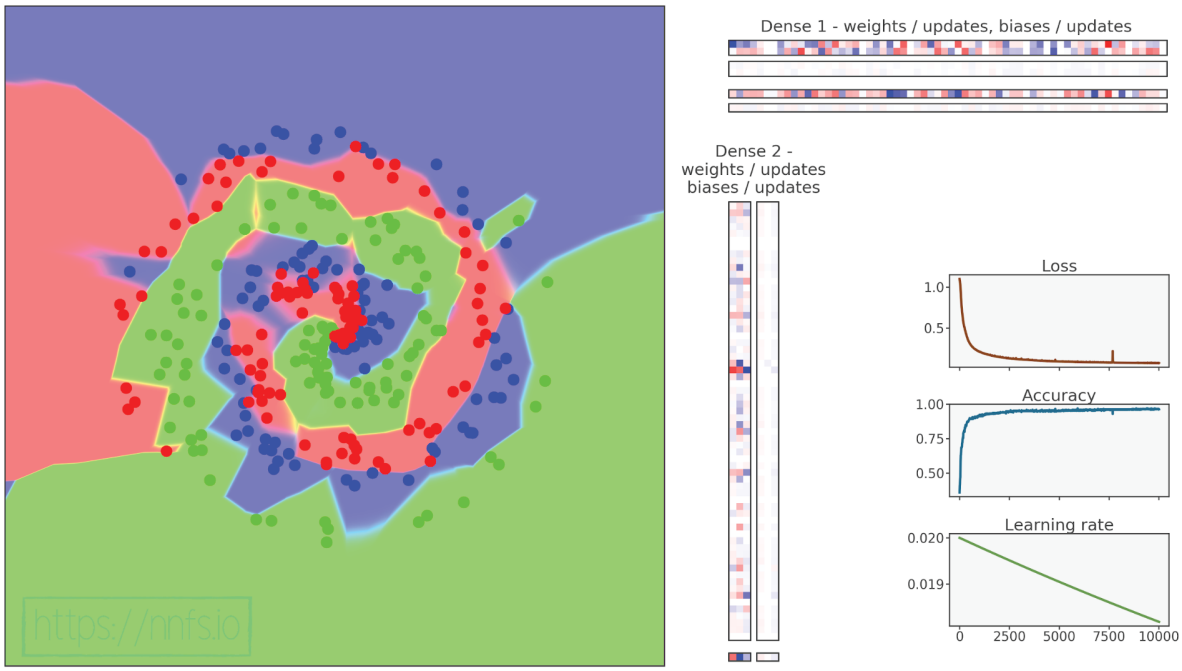


Fig 10.23: Model training with Adam optimizer.

Epilepsy Warning (quick flashing colors)

Anim 10.23: https://nnfs.io/you

This is the best result so far, but let's adjust the learning rate to be a bit higher, to *0.05* and change decay to *5e-7*:

```
optimizer = Optimizer_Adam(learning_rate=0.05, decay=5e-7)
```

In this case, loss and accuracy slightly improved, ending on:

```
>>>
epoch: 0, acc: 0.360, loss: 1.099, lr: 0.05
epoch: 100, acc: 0.713, loss: 0.684, lr: 0.04999752512250644
epoch: 200, acc: 0.827, loss: 0.511, lr: 0.04999502549496326
...
epoch: 700, acc: 0.907, loss: 0.264, lr: 0.049982531105378675
epoch: 800, acc: 0.897, loss: 0.278, lr: 0.04998003297682575
epoch: 900, acc: 0.923, loss: 0.230, lr: 0.049977535097973466
...
epoch: 2000, acc: 0.930, loss: 0.170, lr: 0.04995007490013731
...
epoch: 3300, acc: 0.950, loss: 0.136, lr: 0.04991766081847992
...
epoch: 7800, acc: 0.973, loss: 0.089, lr: 0.04980578235171948
epoch: 7900, acc: 0.970, loss: 0.089, lr: 0.04980330185930667
epoch: 8000, acc: 0.980, loss: 0.088, lr: 0.04980082161395499
...
epoch: 9900, acc: 0.983, loss: 0.074, lr: 0.049753743844839965
epoch: 10000, acc: 0.983, loss: 0.074, lr: 0.04975126853296942
```

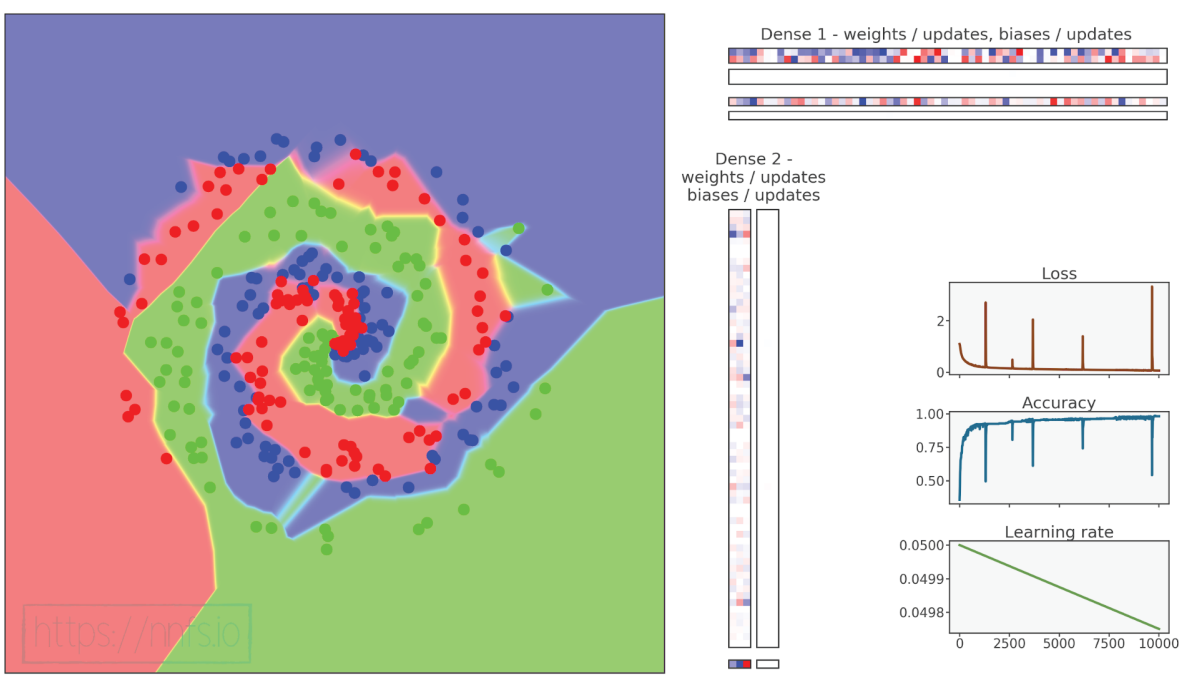


Fig 10.24: Model training with Adam optimizer (tuned).

Epilepsy Warning (quick flashing colors)

Anim 10.24 : https://nnfs.io/car

It doesn't get much better, both for accuracy and loss. While Adam has performed the best here and is usually the best optimizer of those shown, that's not always the case. It's usually a good idea to try the Adam optimizer first but to also try the others, especially if you're not getting the results you hoped for. Sometimes simple SGD or SGD + momentum performs better than Adam. Reasons why will vary, but keep this in mind.

We will cover choosing various hyperparameters (such as the learning rate) when training, but a general starting learning rate for SGD is 1.0, with a decay down to 0.1. For Adam, a good starting LR is 0.001 (1e-3), decaying down to 0.0001 (1e-4). Different problems may require different values here, but these are decent to start.

We achieved 98.3% accuracy on the generated dataset in this section, and a loss approaching perfection (0). Rather than being excited, you will soon learn to fear results this good, or at least approach them cautiously. There are cases where you can truly achieve valid results as good as these, but, in this case, we've been ignoring a major concept in machine learning: out-of-sample testing data (which can shed light on over-fitting), which is the subject of the next section.

Full code up to this point:

```
import numpy as np
import nnfs
from nnfs.datasets import spiral_data

nnfs.init()
```

```python
# Dense layer
class Layer_Dense:

    # Layer initialization
    def __init__(self, n_inputs, n_neurons):
        # Initialize weights and biases
        self.weights = 0.01 * np.random.randn(n_inputs, n_neurons)
        self.biases = np.zeros((1, n_neurons))

    # Forward pass
    def forward(self, inputs):
        # Remember input values
        self.inputs = inputs
        # Calculate output values from inputs, weights and biases
        self.output = np.dot(inputs, self.weights) + self.biases

    # Backward pass
    def backward(self, dvalues):
        # Gradients on parameters
        self.dweights = np.dot(self.inputs.T, dvalues)
        self.dbiases = np.sum(dvalues, axis=0, keepdims=True)
        # Gradient on values
        self.dinputs = np.dot(dvalues, self.weights.T)


# ReLU activation
class Activation_ReLU:

    # Forward pass
    def forward(self, inputs):
        # Remember input values
        self.inputs = inputs
        # Calculate output values from inputs
        self.output = np.maximum(0, inputs)

    # Backward pass
    def backward(self, dvalues):
        # Since we need to modify original variable,
        # let’s make a copy of values first
        self.dinputs = dvalues.copy()

        # Zero gradient where input values were negative
        self.dinputs[self.inputs <= 0] = 0
```

```
# Softmax activation
class Activation_Softmax:

    # Forward pass
    def forward(self, inputs):
        # Remember input values
        self.inputs = inputs

        # Get unnormalized probabilities
        exp_values = np.exp(inputs - np.max(inputs, axis=1,
                                            keepdims=True))
        # Normalize them for each sample
        probabilities = exp_values / np.sum(exp_values, axis=1,
                                            keepdims=True)

        self.output = probabilities

    # Backward pass
    def backward(self, dvalues):

        # Create uninitialized array
        self.dinputs = np.empty_like(dvalues)

        # Enumerate outputs and gradients
        for index, (single_output, single_dvalues) in \
                enumerate(zip(self.output, dvalues)):
            # Flatten output array
            single_output = single_output.reshape(-1, 1)
            # Calculate Jacobian matrix of the output
            jacobian_matrix = np.diagflat(single_output) - \
                              np.dot(single_output, single_output.T)
            # Calculate sample-wise gradient
            # and add it to the array of sample gradients
            self.dinputs[index] = np.dot(jacobian_matrix,
                                         single_dvalues)


# SGD optimizer
class Optimizer_SGD:

    # Initialize optimizer - set settings,
    # learning rate of 1. is default for this optimizer
    def __init__(self, learning_rate=1., decay=0., momentum=0.):
        self.learning_rate = learning_rate
        self.current_learning_rate = learning_rate
        self.decay = decay
        self.iterations = 0
        self.momentum = momentum
```

```
# Call once before any parameter updates
def pre_update_params(self):
    if self.decay:
        self.current_learning_rate = self.learning_rate * \
            (1. / (1. + self.decay * self.iterations))

# Update parameters
def update_params(self, layer):

    # If we use momentum
    if self.momentum:

        # If layer does not contain momentum arrays, create them
        # filled with zeros
        if not hasattr(layer, 'weight_momentums'):
            layer.weight_momentums = np.zeros_like(layer.weights)
            # If there is no momentum array for weights
            # The array doesn't exist for biases yet either.
            layer.bias_momentums = np.zeros_like(layer.biases)

        # Build weight updates with momentum - take previous
        # updates multiplied by retain factor and update with
        # current gradients
        weight_updates = \
            self.momentum * layer.weight_momentums - \
            self.current_learning_rate * layer.dweights
        layer.weight_momentums = weight_updates

        # Build bias updates
        bias_updates = \
            self.momentum * layer.bias_momentums - \
            self.current_learning_rate * layer.dbiases
        layer.bias_momentums = bias_updates

    # Vanilla SGD updates (as before momentum update)
    else:
        weight_updates = -self.current_learning_rate * \
                         layer.dweights
        bias_updates = -self.current_learning_rate * \
                       layer.dbiases

    # Update weights and biases using either
    # vanilla or momentum updates
    layer.weights += weight_updates
    layer.biases += bias_updates
```

```
    # Call once after any parameter updates
    def post_update_params(self):
        self.iterations += 1


# Adagrad optimizer
class Optimizer_Adagrad:

    # Initialize optimizer - set settings
    def __init__(self, learning_rate=1., decay=0., epsilon=1e-7):
        self.learning_rate = learning_rate
        self.current_learning_rate = learning_rate
        self.decay = decay
        self.iterations = 0
        self.epsilon = epsilon

    # Call once before any parameter updates
    def pre_update_params(self):
        if self.decay:
            self.current_learning_rate = self.learning_rate * \
                (1. / (1. + self.decay * self.iterations))

    # Update parameters
    def update_params(self, layer):

        # If layer does not contain cache arrays,
        # create them filled with zeros
        if not hasattr(layer, 'weight_cache'):
            layer.weight_cache = np.zeros_like(layer.weights)
            layer.bias_cache = np.zeros_like(layer.biases)

        # Update cache with squared current gradients
        layer.weight_cache += layer.dweights**2
        layer.bias_cache += layer.dbiases**2

        # Vanilla SGD parameter update + normalization
        # with square rooted cache
        layer.weights += -self.current_learning_rate * \
                         layer.dweights / \
                         (np.sqrt(layer.weight_cache) + self.epsilon)
        layer.biases += -self.current_learning_rate * \
                        layer.dbiases / \
                        (np.sqrt(layer.bias_cache) + self.epsilon)

    # Call once after any parameter updates
    def post_update_params(self):
        self.iterations += 1
```

```python
# RMSprop optimizer
class Optimizer_RMSprop:

    # Initialize optimizer - set settings
    def __init__(self, learning_rate=0.001, decay=0., epsilon=1e-7,
                 rho=0.9):
        self.learning_rate = learning_rate
        self.current_learning_rate = learning_rate
        self.decay = decay
        self.iterations = 0
        self.epsilon = epsilon
        self.rho = rho

    # Call once before any parameter updates
    def pre_update_params(self):
        if self.decay:
            self.current_learning_rate = self.learning_rate * \
                (1. / (1. + self.decay * self.iterations))

    # Update parameters
    def update_params(self, layer):

        # If layer does not contain cache arrays,
        # create them filled with zeros
        if not hasattr(layer, 'weight_cache'):
            layer.weight_cache = np.zeros_like(layer.weights)
            layer.bias_cache = np.zeros_like(layer.biases)

        # Update cache with squared current gradients
        layer.weight_cache = self.rho * layer.weight_cache + \
            (1 - self.rho) * layer.dweights**2
        layer.bias_cache = self.rho * layer.bias_cache + \
            (1 - self.rho) * layer.dbiases**2

        # Vanilla SGD parameter update + normalization
        # with square rooted cache
        layer.weights += -self.current_learning_rate * \
                         layer.dweights / \
                         (np.sqrt(layer.weight_cache) + self.epsilon)
        layer.biases += -self.current_learning_rate * \
                        layer.dbiases / \
                        (np.sqrt(layer.bias_cache) + self.epsilon)

    # Call once after any parameter updates
    def post_update_params(self):
        self.iterations += 1
```

```
# Adam optimizer
class Optimizer_Adam:

    # Initialize optimizer - set settings
    def __init__(self, learning_rate=0.001, decay=0., epsilon=1e-7,
                 beta_1=0.9, beta_2=0.999):
        self.learning_rate = learning_rate
        self.current_learning_rate = learning_rate
        self.decay = decay
        self.iterations = 0
        self.epsilon = epsilon
        self.beta_1 = beta_1
        self.beta_2 = beta_2

    # Call once before any parameter updates
    def pre_update_params(self):
        if self.decay:
            self.current_learning_rate = self.learning_rate * \
                (1. / (1. + self.decay * self.iterations))

    # Update parameters
    def update_params(self, layer):

        # If layer does not contain cache arrays,
        # create them filled with zeros
        if not hasattr(layer, 'weight_cache'):
            layer.weight_momentums = np.zeros_like(layer.weights)
            layer.weight_cache = np.zeros_like(layer.weights)
            layer.bias_momentums = np.zeros_like(layer.biases)
            layer.bias_cache = np.zeros_like(layer.biases)

        # Update momentum  with current gradients
        layer.weight_momentums = self.beta_1 * \
                                 layer.weight_momentums + \
                                 (1 - self.beta_1) * layer.dweights
        layer.bias_momentums = self.beta_1 * \
                               layer.bias_momentums + \
                               (1 - self.beta_1) * layer.dbiases
        # Get corrected momentum
        # self.iteration is 0 at first pass
        # and we need to start with 1 here
        weight_momentums_corrected = layer.weight_momentums / \
            (1 - self.beta_1 ** (self.iterations + 1))
        bias_momentums_corrected = layer.bias_momentums / \
            (1 - self.beta_1 ** (self.iterations + 1))
        # Update cache with squared current gradients
        layer.weight_cache = self.beta_2 * layer.weight_cache + \
            (1 - self.beta_2) * layer.dweights**2
```

```
        layer.bias_cache = self.beta_2 * layer.bias_cache + \
            (1 - self.beta_2) * layer.dbiases**2
        # Get corrected cache
        weight_cache_corrected = layer.weight_cache / \
            (1 - self.beta_2 ** (self.iterations + 1))
        bias_cache_corrected = layer.bias_cache / \
            (1 - self.beta_2 ** (self.iterations + 1))

        # Vanilla SGD parameter update + normalization
        # with square rooted cache
        layer.weights += -self.current_learning_rate * \
                         weight_momentums_corrected / \
                         (np.sqrt(weight_cache_corrected) +
                             self.epsilon)
        layer.biases += -self.current_learning_rate * \
                         bias_momentums_corrected / \
                         (np.sqrt(bias_cache_corrected) +
                             self.epsilon)

    # Call once after any parameter updates
    def post_update_params(self):
        self.iterations += 1


# Common loss class
class Loss:

    # Calculates the data and regularization losses
    # given model output and ground truth values
    def calculate(self, output, y):

        # Calculate sample losses
        sample_losses = self.forward(output, y)

        # Calculate mean loss
        data_loss = np.mean(sample_losses)

        # Return loss
        return data_loss


# Cross-entropy loss
class Loss_CategoricalCrossentropy(Loss):

    # Forward pass
    def forward(self, y_pred, y_true):

        # Number of samples in a batch
        samples = len(y_pred)
```

```
        # Clip data to prevent division by 0
        # Clip both sides to not drag mean towards any value
        y_pred_clipped = np.clip(y_pred, 1e-7, 1 - 1e-7)

        # Probabilities for target values -
        # only if categorical labels
        if len(y_true.shape) == 1:
            correct_confidences = y_pred_clipped[
                range(samples),
                y_true
            ]

        # Mask values - only for one-hot encoded labels
        elif len(y_true.shape) == 2:
            correct_confidences = np.sum(
                y_pred_clipped * y_true,
                axis=1
            )

        # Losses
        negative_log_likelihoods = -np.log(correct_confidences)
        return negative_log_likelihoods

    # Backward pass
    def backward(self, dvalues, y_true):

        # Number of samples
        samples = len(dvalues)
        # Number of labels in every sample
        # We'll use the first sample to count them
        labels = len(dvalues[0])

        # If labels are sparse, turn them into one-hot vector
        if len(y_true.shape) == 1:
            y_true = np.eye(labels)[y_true]

        # Calculate gradient
        self.dinputs = -y_true / dvalues
        # Normalize gradient
        self.dinputs = self.dinputs / samples
```

```
# Softmax classifier - combined Softmax activation
# and cross-entropy loss for faster backward step
class Activation_Softmax_Loss_CategoricalCrossentropy():

    # Creates activation and loss function objects
    def __init__(self):
        self.activation = Activation_Softmax()
        self.loss = Loss_CategoricalCrossentropy()

    # Forward pass
    def forward(self, inputs, y_true):
        # Output layer's activation function
        self.activation.forward(inputs)
        # Set the output
        self.output = self.activation.output
        # Calculate and return loss value
        return self.loss.calculate(self.output, y_true)

    # Backward pass
    def backward(self, dvalues, y_true):

        # Number of samples
        samples = len(dvalues)

        # If labels are one-hot encoded,
        # turn them into discrete values
        if len(y_true.shape) == 2:
            y_true = np.argmax(y_true, axis=1)

        # Copy so we can safely modify
        self.dinputs = dvalues.copy()
        # Calculate gradient
        self.dinputs[range(samples), y_true] -= 1
        # Normalize gradient
        self.dinputs = self.dinputs / samples


# Create dataset
X, y = spiral_data(samples=100, classes=3)

# Create Dense layer with 2 input features and 64 output values
dense1 = Layer_Dense(2, 64)

# Create ReLU activation (to be used with Dense layer):
activation1 = Activation_ReLU()

# Create second Dense layer with 64 input features (as we take output
# of previous layer here) and 3 output values (output values)
dense2 = Layer_Dense(64, 3)
```

```
# Create Softmax classifier's combined loss and activation
loss_activation = Activation_Softmax_Loss_CategoricalCrossentropy()

# Create optimizer
optimizer = Optimizer_Adam(learning_rate=0.05, decay=5e-7)

# Train in loop
for epoch in range(10001):

    # Perform a forward pass of our training data through this layer
    dense1.forward(X)

    # Perform a forward pass through activation function
    # takes the output of first dense layer here
    activation1.forward(dense1.output)

    # Perform a forward pass through second Dense layer
    # takes outputs of activation function of first layer as inputs
    dense2.forward(activation1.output)

    # Perform a forward pass through the activation/loss function
    # takes the output of second dense layer here and returns loss
    loss = loss_activation.forward(dense2.output, y)

    # Calculate accuracy from output of activation2 and targets
    # calculate values along first axis
    predictions = np.argmax(loss_activation.output, axis=1)
    if len(y.shape) == 2:
        y = np.argmax(y, axis=1)
    accuracy = np.mean(predictions==y)

    if not epoch % 100:
        print(f'epoch: {epoch}, ' +
              f'acc: {accuracy:.3f}, ' +
              f'loss: {loss:.3f}, ' +
              f'lr: {optimizer.current_learning_rate}')

    # Backward pass
    loss_activation.backward(loss_activation.output, y)
    dense2.backward(loss_activation.dinputs)
    activation1.backward(dense2.dinputs)
    dense1.backward(activation1.dinputs)

    # Update weights and biases
    optimizer.pre_update_params()
    optimizer.update_params(dense1)
    optimizer.update_params(dense2)
    optimizer.post_update_params()
```

Supplementary Material: https://nnfs.io/ch10
Chapter code, further resources, and errata for this chapter.

Chapter 11

Testing with Out-of-Sample Data

Up to this point, we've created a model that is seemingly 98% accurate at predicting the testing dataset that we've generated. These generated data are created based on a very clear set of rules outlined in the *spiral_data* function. The expectation is that a well-trained neural network can learn a representation of these rules and use this representation to predict classes of additional generated data.

Imagine that you've trained a neural network model to read license plates on vehicles. The expectation for a well-trained model, in this case, would be that it could see future examples of license plates and still accurately predict them (a prediction, in this case, would be correctly identifying the characters on the license plate).

The complexity of neural networks is their biggest issue and strength. By having a massive amount of tunable parameters, they are exceptional at "fitting" to data. This is a gift, a curse,

and something that we must constantly try to balance. With enough neurons, a model can easily memorize a dataset; however, it can not generalize the data with too few. This is one reason why we do not simply solve problems with neural networks by using the most neurons or biggest models possible.

At the moment, we're uncertain whether our latest neural network's 98% accuracy is due to learning to meaningfully represent the underlying data-generation function or instead **overfitting** the data. So far, we have only tuned hyper-parameters to achieve the highest possible accuracy on the training data, and have never tried to challenge the model with the previously unseen data.

Overfitting is effectively just memorizing the data without any understanding of it. An overfit model will do very well predicting the data that it has already seen, but often significantly worse on unseen data.

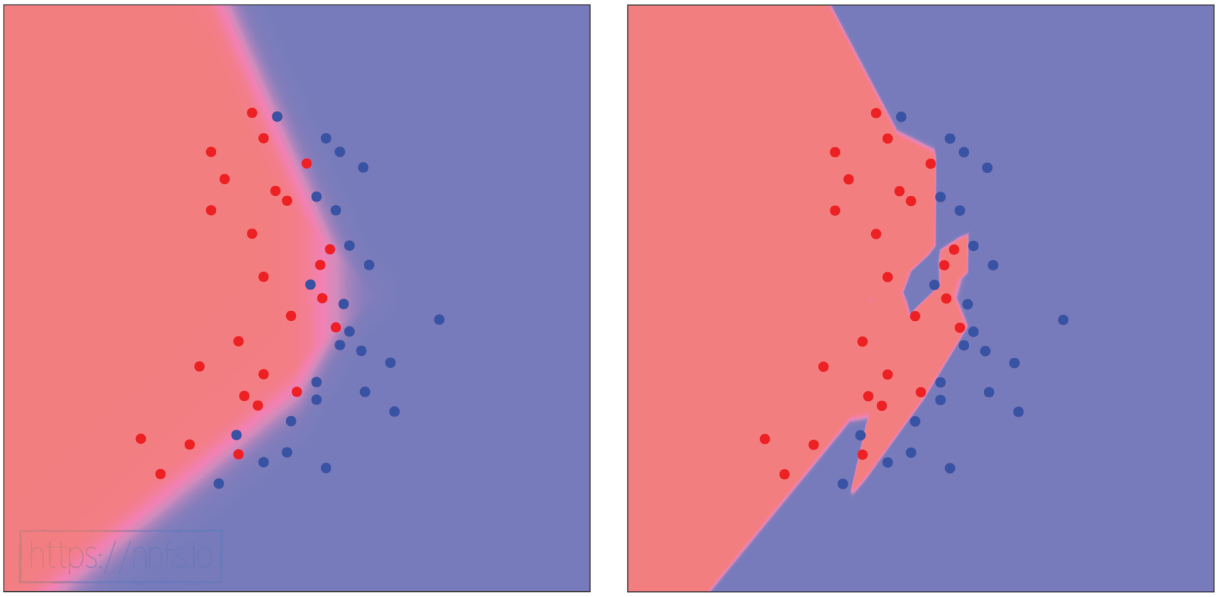


Fig 11.01: Good generalization (left) and overfitting (right) on the data

The left image shows an example of generalization. In this example, the model learned to separate red and blue data points, even if some of them will be predicted incorrectly. One reason for this might be the data that contains some "confusing" samples. When you look at the image, you can see that, for example, some of these blue dots might not be there, which would raise the data quality and make it easier to fit. A good dataset is one of the biggest challenges with neural networks. The image on the right shows the model that memorized the data, fitting them perfectly and ruining generalization.

Without knowing if a model overfits the training data, we cannot trust the model's results. For this reason, it's essential to have both **training** and **testing data** as separate sets for different purposes.

Training data should only be used to train a model. The **testing,** or **out-of-sample** data, should only be used to validate a model's performance after training (we are using the testing data during training later in this chapter for demonstration purposes only). The idea is that some data are reserved and withheld from the training data for testing the model's performance.

In many cases, one can take a random sampling of available data to train with and make the remaining data the testing dataset. You still need to be very careful about information leaking through. One common area where this can be problematic is in time-series data. Consider a scenario where you have data from sensors collected every second. You might have millions of observations collected, and randomly selecting your data for the **testing** data might result in samples in your **testing** dataset that are only a second in time apart from your **training** data, thus are very similar. This means overfitting can spill into your testing data, and the model can achieve good results on both the training and the testing data, which won't mean it generalized well. Randomly allocating time-series data as testing data may be very similar to training data. Both datasets must differ enough to prove the model's ability to generalize. In time-series data, a better approach is to take multiple slices of your data, entire blocks of time, and reserve those for testing.

Other biases like these can sneak into your testing dataset, and this is something you must be vigilant about, carefully considering if data leakage has occurred and how to truly isolate **out-of-sample** data.

In our case, we can use our data-generating function to create new data that will serve as out-of-sample/testing data:

```
# Create test dataset
X_test, y_test = spiral_data(samples=100, classes=3)
```

Given what was just said about overfitting, it may look wrong to only generate more data, as the testing data could look similar to the training data. Intuition and experience are both important to spot potential issues with out-of-sample data. By looking at the image representation of the data, we can see that another set of data generated by the same function will be adequate. This is just about as safe as it gets for out-of-sample data as the classes are partially mixing at the edges (also, we're quite literally using the "underlying function" to make more data).

With these data, we evaluate the model's performance by doing a forward pass and calculating loss and accuracy the same as before:

```
# Validate the model

# Create test dataset
X_test, y_test = spiral_data(samples=100, classes=3)

# Perform a forward pass of our testing data through this layer
dense1.forward(X_test)

# Perform a forward pass through activation function
# takes the output of first dense layer here
activation1.forward(dense1.output)

# Perform a forward pass through second Dense layer
# takes outputs of activation function of first layer as inputs
dense2.forward(activation1.output)

# Perform a forward pass through the activation/loss function
# takes the output of second dense layer here and returns loss
loss = loss_activation.forward(dense2.output, y_test)

# Calculate accuracy from output of activation2 and targets
# calculate values along first axis
predictions = np.argmax(loss_activation.output, axis=1)
if len(y_test.shape) == 2:
    y_test = np.argmax(y_test, axis=1)
accuracy = np.mean(predictions == y_test)

print(f'validation, acc: {accuracy:.3f}, loss: {loss:.3f}')
```

```
>>>
...
epoch: 9800, acc: 0.983, loss: 0.075, lr: 0.04975621940303483
epoch: 9900, acc: 0.983, loss: 0.074, lr: 0.049753743844839965
epoch: 10000, acc: 0.983, loss: 0.074, lr: 0.04975126853296942
validation, acc: 0.803, loss: 0.858
```

While 80.3% accuracy and a loss of *0.858* is not terrible, this contrasts with our training data that achieved 98% accuracy and a loss of 0.074. This is evidence of over-fitting. In the following image, the training data is dimmed, and validation data points are shown on top of it at the same positions for both the well-generalized (on the left) and overfitted (on the right) models.

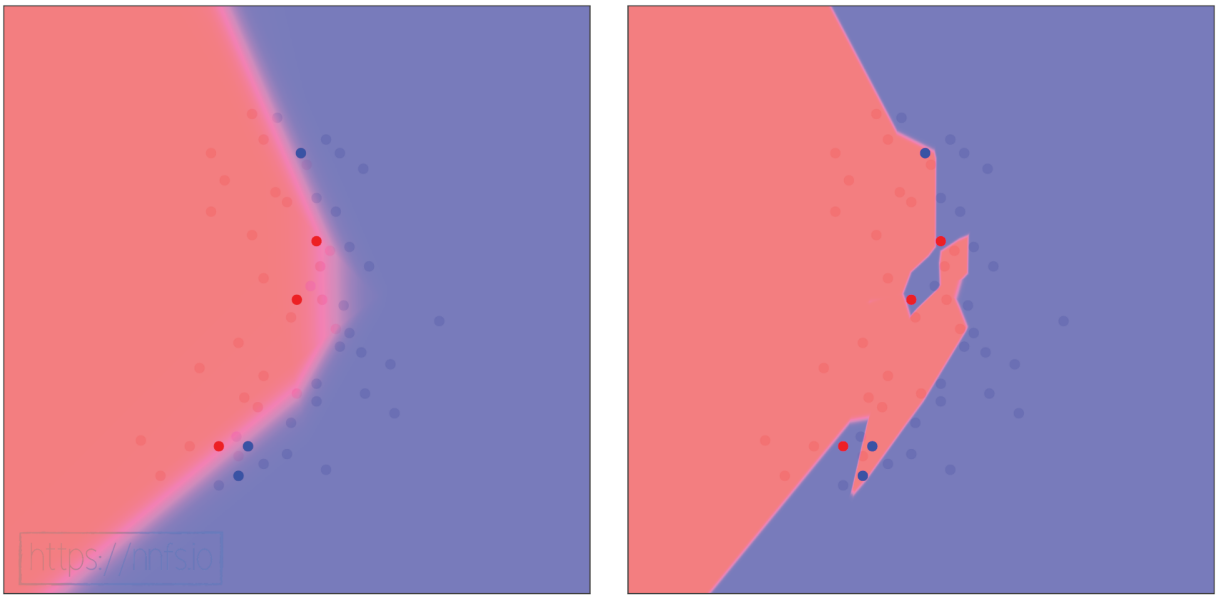


Fig 11.02: Left - prediction with well-generalized model; right - prediction mistakes with an overfit model.

We can recognize overfitting when testing data results begin to diverge in trend from training data. It will usually be the case that performance against your training data is better, but having training loss differ from test performance by over 10% approximately is a common sign of serious overfitting from our anecdotal experience. Ideally, both datasets would have identical performance. Even a small difference means that the model did not correctly predict some testing samples, implying slight overfitting of training data. In most cases, modest overfitting is not a serious problem, but something we hope to minimize.

Let's see the training process of this model once again, but with the training data, training accuracy, and loss plots dimmed. We add the test data and its loss and accuracy plotted on top of the training counterparts to show this model overfitting:

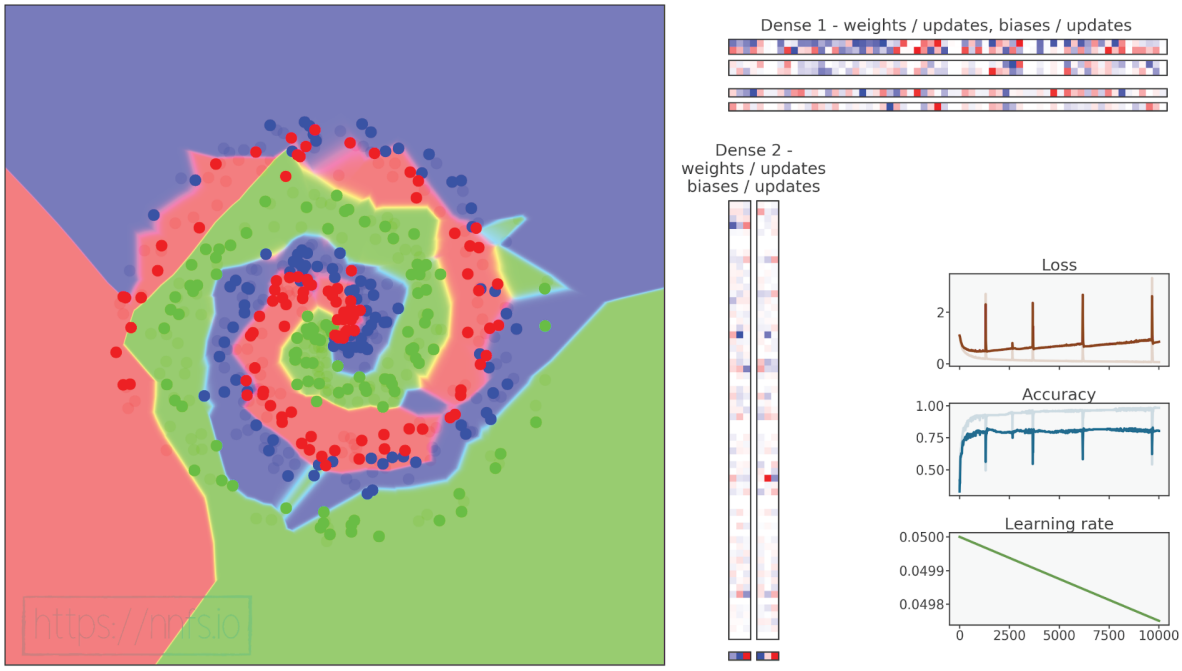


Fig 11.03: Prediction issues on the testing data — overfitted model.

Anim 11.03: https://nnfs.io/zog

This is a classic example of overfitting — the validation loss falls down, then starts rising once the model starts overfitting. The dots representing classes in the validation data can be spotted over areas of effect of other classes. Previously, we weren't aware that this was happening; we were just seeing very good training results. That's why we usually should use the testing data to test the model after training. The model is currently tuned to achieve the best possible score on the training data, and most likely the learning rate is too high, there are too many training epochs, or the model is too big. There are other possible causes and ways to fix this, but this is the topic of the following chapters. In general, the goal is to have the testing loss identical to the training loss, even if that means higher loss and lower accuracy on the training data. Similar performance on both datasets means that model generalized instead of overfitting on the training data.

As mentioned, one option to prevent overfitting is to change the model's size. If a model is not learning at all, one solution might be to try a larger model. If your model is learning, but there's a divergence between the training and testing data, it could mean that you should try a smaller model. One general rule to follow when selecting initial model hyperparameters is to find the smallest model possible that still learns. Other possible ways to avoid overfitting are regularization techniques we'll discuss in chapter 14, and the *Dropout* layer explained in chapter 15. Often the divergence of the training and testing data can take a long time to occur. The process of trying different model settings is called hyperparameter searching. Initially, you can very quickly (usually within minutes) try different settings (e.g., layer sizes) to see if the models are learning *something*. If they are, train the models fully — or at least significantly longer — and compare results to pick the best set of hyperparameters. Another possibility is to create a list of different hyperparameter sets and train the model in a loop using each of those sets at a time to pick the best set at the end. The reasoning here is that the fewer neurons you have, the less chance you have that the model is memorizing the data. Fewer neurons can mean it's easier for a neural network to generalize (actually learn the meaning of the data) compared to memorizing the data. With enough neurons, it's easier for a neural network to memorize the data. Remember that the neural network wants to decrease training loss and follows the path of least resistance to meet that objective. Our job as the programmer is to make the path to generalization the easiest path. This can often mean our job is actually to make the path to lowering loss for the model more challenging!

Supplementary Material: https://nnfs.io/ch11
Chapter code, further resources, and errata for this chapter.

Chapter 12

Validation Data

In the chapter on optimization, we used hyperparameter tuning to select hyperparameters that lead to better results, but one more thing requires clarification. We *should not* check different hyperparameters using the test dataset; if we do that, we're going to be manually optimizing the model to the test dataset, biasing it towards overfitting these data, and these data are supposed to be used only to perform the last check if the model trains and generalizes well. In other words, if we're tuning our network's parameters to fit the testing data, then we're essentially optimizing our network on the testing data, which is another way for overfitting on these data.

Thus, hyperparameter tuning using the test dataset is a mistake. The test dataset should only be used as unseen data, not informing the model in any way, which hyperparameter tuning is, other than to test performance.

Hyperparameter tuning can be performed using yet another dataset called **validation data**. The test dataset needs to contain real out-of-sample data, but with a validation dataset, we have more freedom with choosing data. If we have a lot of training data and can afford to use some for validation purposes, we can take it as an out-of-sample dataset, similar to a test dataset. We can now search for parameters that work best using this new validation dataset and test our model

at the end using the test dataset to see if we really tuned the model or just overfitted it to the validation data.

There are situations when we'll be short on data and cannot afford to create yet another dataset from the training data. In those situations, we have two options:

The first is to temporarily split the training data into a smaller training dataset and validation dataset for hyperparameter tuning. Afterward, with the final hyperparameter set, train the model on all the training data. We allow ourselves to do that as we tune the model to the part of training data that we put aside as validation data. Keep in mind that we still have a test dataset to check the model's performance after training.

The second possibility in situations where we are short on data is a process called **cross-validation**. Cross-validation is primarily used when we have a small training dataset and cannot afford any data for validation purposes. How it works is we split the training dataset into a given number of parts, let's say 5. We now train the model on the first 4 chunks and validate it on the last. So far, this is similar to the case described previously — we are also only using the training dataset and can validate on data that was not used for training. What makes cross-validation different is that we then swap samples. For example, if we have 5 chunks, we can call them chunks A, B, C, D, and E. We may first train on A, B, C, and D, then validate on E. We'll then train on A, B, C, E, and validate on D, doing this until we've validated on each of the 5 sample groups. This way, we do not lose any training data. We validate using the data that was not used for training during any given iteration and validate on more data than if we just temporarily split the training dataset and train on all of the samples. This validation method is often called k-fold cross-validation; here, our k is 5. Here's an example of 2 steps of cross-validation:

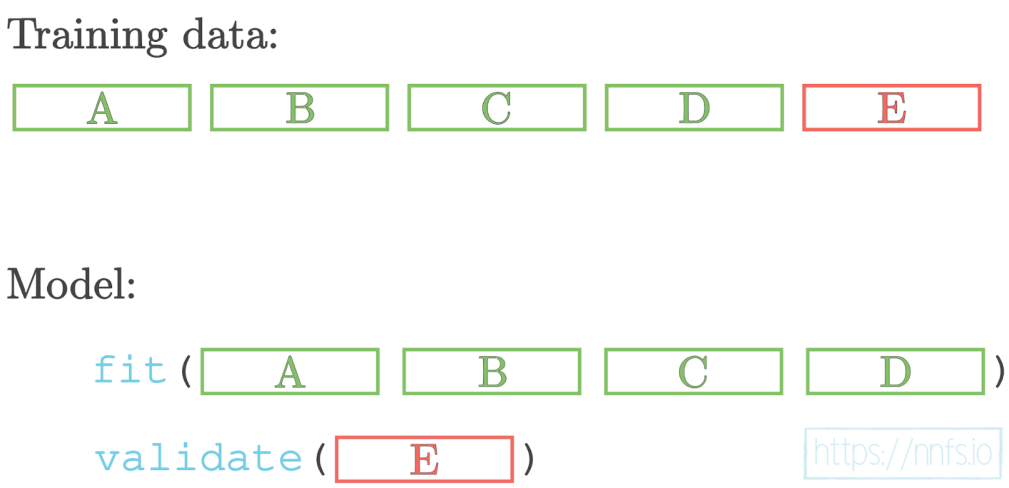


Fig 12.01: Cross-validation, first step.

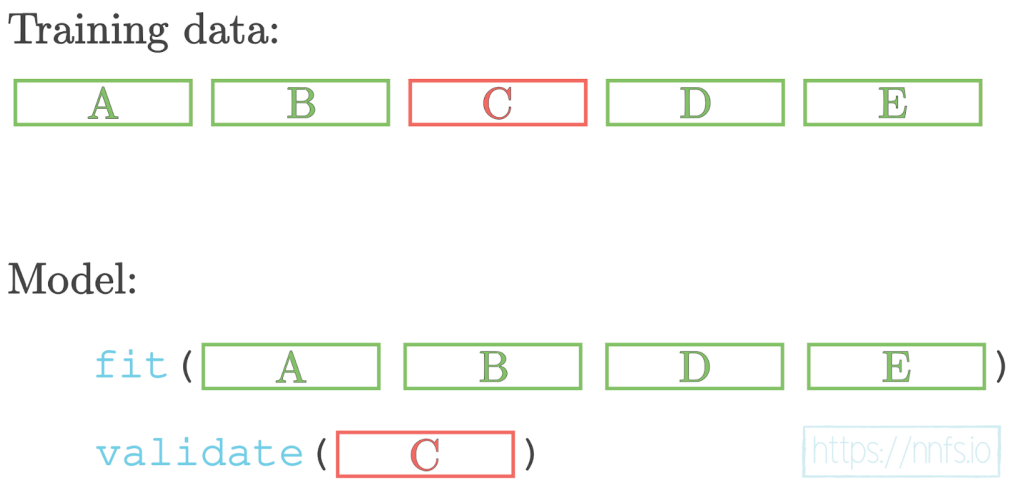


Fig 12.02: Cross-validation, third step.

Anim 12.01-12.02: https://nnfs.io/lho

When using a validation dataset and cross-validation, it is common to loop over different hyperparameter sets, leaving the code to run training multiple times, applying different settings each run, and reviewing the results to choose the best set of hyperparameters. In general, we should not loop over *all* possible setting combinations that we would like to check unless training is exceptionally fast. It's usually better to check some settings that we suspect will work well, pick the best combination of those settings, tweak them to create the next list of setting sets, and train the model on new sets. We can repeat this process as many times as we'd like.

Supplementary Material: https://nnfs.io/ch12

Chapter code, further resources, and errata for this chapter.

Chapter 13

Training Dataset

Since we are talking about datasets and testing, it's worth mentioning a few things about the training dataset and operations that we can perform on it; this technique is referred to as **preprocessing**. However, it's important to remember that any preprocessing we do to our training data also needs to be done to our validation and testing data and later done to the prediction data.

Neural networks usually perform best on data consisting of numbers in a range of 0 to 1 or -1 to 1, with the latter being preferable. Centering data on the value of 0 can help with model training as it attenuates weight biasing in some direction. Models can work fine with data in the range of 0 to 1 in most cases, but sometimes we're going to need to rescale them to a range of -1 to 1 to get training to behave or achieve better results.

Speaking of the data range, the values do not have to strictly be in the range of -1 and 1 — the model will perform well with data slightly outside of this range or with just some values being many times bigger. The case here is that when we multiply data by a weight and sum the results with a bias, we're usually passing the resulting output to an activation function. Many activation functions behave properly within this described range. For example, *softmax* outputs a vector of probabilities containing numbers in the range of 0 to 1; *sigmoid* also has an output range of 0 to 1,

but *tanh* outputs a range from -1 to 1.

Another reason why this scaling is ideal is a neural network's reliance on many multiplication operations. If we multiply by numbers above 1 or below -1, the resulting value is larger in scale than the original one. Within the -1 to 1 range, the result becomes a fraction, a smaller value. Multiplying big numbers from our training data with weights might cause floating-point overflow or instability — weights growing too fast. It's easier to control the training process with smaller numbers.

There are many terms related to data **preprocessing**: standardization, scaling, variance scaling, mean removal (as mentioned above), non-linear transformations, scaling to outliers, etc., but they are out of the scope of this book. We're only going to scale data to a range by simply dividing all of the numbers by the maximum of their absolute values. For the example of an image that consists of numbers in the range between *0* and *255*, we divide the whole dataset by *255* and return data in the range from *0* to *1*. We can also subtract *127.5* (to get a range from *-127.5* to *127.5*) and divide by 127.5, returning data in the range from -1 to 1.

We need to ensure identical scaling for all the datasets (same scale parameters). For example, we can find the maximum for training data and divide training, validation and testing data by this number. In general, we should prepare a scaler of our choice and use its instance on every dataset. It is important to remember that once we train our model and want to predict using new samples, we need to scale those new samples by using the same scaler instance we used on the training, validation, and testing data. In most cases, when we are working with data (e.g., sensor data), we will need to save the scaler object along with the model and use it during prediction as well; otherwise, results are likely to vary as the model might not effectively recognize these data without being scaled. It is usually fine to scale datasets that consist of larger numbers than the training data using a scaler prepared on the training data. If the resulting numbers are slightly outside of the *-1* to *1* range, it does not affect validation or testing negatively, since we do not train on these data. Additionally, for linear scaling, we can use different datasets to find the maximum as well, but be aware that non-linear scaling can leak the information from other datasets to the training dataset and, in this case, the scaler should be prepared on the training data only.

In cases where we do not have many training samples, we could use **data augmentation**. One easy way to understand augmentation is in the case of images. Let's imagine that our model's goal is to detect rotten fruits — apples, for example. We will take a photo of an apple from different angles and predict whether it's rotten. We should get more pictures in this case, but let's assume that we cannot. What we could do is to take photos that we have, rotate, crop, and save those as worthy data too. This way, we have added more samples to the dataset, which can help with model generalization. In general, if we use augmentation, then it's only useful if the augmentations that we make are similar to variations that we could see in reality. For example, we may refrain from using a rotation when creating a model to detect road signs as they are not being rotated in real-life scenarios (in most cases, anyway). The case of a rotated road sign, however, is one you better

consider if you're making a self-driving car. Just because a bolt came loose on a stop sign, flipping it over, doesn't mean you no longer need to stop there!

How many samples do we need to train the model? There is no single answer to this question — one model might require just a few per class, and another may require a few million or billion. Usually, a few thousand per class will be necessary, and a few tens of thousands should be preferable to start. The difference depends on the data complexity and model size. If the model has to predict sensor data with 2 simple classes, for example, if an image contains a dark area or does not, hundreds of samples per class might be enough. To train on data with many features and several classes, tens of thousands of samples are what you should start with. If you're attempting to train a chatbot the intricacies of written language, then you're going to likely want at least millions of samples.

Supplementary Material: https://nnfs.io/ch13
Chapter code, further resources, and errata for this chapter.

Chapter 14

L1 and L2 Regularization

Regularization methods are those which reduce generalization error. The first forms of regularization that we'll address are **L1** and **L2 regularization**. L1 and L2 regularization are used to calculate a number (called a **penalty**) added to the loss value to penalize the model for large weights and biases. Large weights might indicate that a neuron is attempting to memorize a data element; generally, it is believed that it would be better to have many neurons contributing to a model's output, rather than a select few.

Forward Pass

L1 regularization's penalty is the sum of all the absolute values for the weights and biases. This is a linear penalty as regularization loss returned by this function is directly proportional to parameter values. L2 regularization's penalty is the sum of the squared weights and biases. This non-linear approach penalizes larger weights and biases more than smaller ones because of the square function used to calculate the result. In other words, L2 regularization is commonly used as it does not affect small parameter values substantially and does not allow the model to grow weights too large by heavily penalizing relatively big values. L1 regularization, because of its linear nature, penalizes small weights more than L2 regularization, causing the model to start being invariant to small inputs and variant only to the bigger ones. That's why L1 regularization is rarely used alone and usually combined with L2 regularization if it's even used at all. Regularization functions of this type drive the sum of weights and the sum of parameters towards *0*, which can also help in cases of exploding gradients (model instability, which might cause weights to become very large values). Beyond this, we also want to dictate how much of an impact we want this regularization penalty to carry. We use a value referred to as **lambda** in this equation — where a higher value means a more significant penalty.

L1 weight regularization:

$$L_{1w} = \lambda \sum_{m} |w_m|$$

L1 bias regularization:

$$L_{1b} = \lambda \sum_{n} |b_n|$$

L2 weight regularization:

$$L_{2w} = \lambda \sum_{m} w_m^2$$

L2 bias regularization:

$$L_{2b} = \lambda \sum_n b_n^2$$

Overall loss:

$$Loss = DataLoss + L_{1w} + L_{1b} + L_{2w} + L_{2b}$$

Using code notation:

```
l1w = lambda_l1w * sum(abs(weights))
l1b = lambda_l1b * sum(abs(biases))
l2w = lambda_l2w * sum(weights**2)
l2b = lambda_l2b * sum(biases**2)
loss = data_loss + l1w + l1b + l2w + l2b
```

Regularization losses are calculated separately, then summed with the data loss, to form the overall loss. Parameter *m* is an arbitrary iterator over all of the weights in a model, parameter *n* is the bias equivalent of this iterator, w_m is the given weight, and b_n is the given bias.

To implement regularization in our neural network code, we'll start with the `__init__` method of the `Dense` layer's class, which will house the **lambda** regularization strength hyperparameters, since these can be set separately for every layer:

```
# Layer initialization
def __init__(self, n_inputs, n_neurons,
             weight_regularizer_l1=0, weight_regularizer_l2=0,
             bias_regularizer_l1=0, bias_regularizer_l2=0):
    # Initialize weights and biases
    self.weights = 0.01 * np.random.randn(inputs, neurons)
    self.biases = np.zeros((1, neurons))
    # Set regularization strength
    self.weight_regularizer_l1 = weight_regularizer_l1
    self.weight_regularizer_l2 = weight_regularizer_l2
    self.bias_regularizer_l1 = bias_regularizer_l1
    self.bias_regularizer_l2 = bias_regularizer_l2
```

This method sets the lambda hyperparameters. Now we update our loss class to include the additional penalty if we choose to set the lambda hyperparameter for any of the regularizers in the layer's initialization. We will implement this code into the *Loss* class as it is common for the hidden layers. What's more, the regularization calculation is the same, regardless of

the type of loss used. It's only a penalty that is summed with the data loss value resulting in a final, overall loss value. For this reason, we're going to add a new method to a general loss class, which is inherited by all of our specific loss functions (such as our existing `Loss_CategoricalCrossentropy`). For the code of this method, we'll create the layer's regularization loss variable. We'll add to it each of the atomic regularization losses if its corresponding lambda value is greater than *0*. To perform these calculations, we read the lambda hyperparameters, weights, and biases from the passed-in layer object. For our general loss class:

```
# Regularization loss calculation
def regularization_loss(self, layer):

    # 0 by default
    regularization_loss = 0

    # L1 regularization - weights
    # calculate only when factor greater than 0
    if layer.weight_regularizer_l1 > 0:
        regularization_loss += layer.weight_regularizer_l1 * \
                               np.sum(np.abs(layer.weights))

    # L2 regularization - weights
    if layer.weight_regularizer_l2 > 0:
        regularization_loss += layer.weight_regularizer_l2 * \
                               np.sum(layer.weights *
                                      layer.weights)

    # L1 regularization - biases
    # calculate only when factor greater than 0
    if layer.bias_regularizer_l1 > 0:
        regularization_loss += layer.bias_regularizer_l1 * \
                               np.sum(np.abs(layer.biases))

    # L2 regularization - biases
    if layer.bias_regularizer_l2 > 0:
        regularization_loss += layer.bias_regularizer_l2 * \
                               np.sum(layer.biases *
                                      layer.biases)

    return regularization_loss
```

Then we'll calculate the regularization loss and add it to our calculated loss in the training loop:

```
# Calculate loss from output of activation2 so softmax activation
data_loss = loss_function.forward(activation2.output, y)

# Calculate regularization penalty
regularization_loss = loss_function.regularization_loss(dense1) + \
                      loss_function.regularization_loss(dense2)

# Calculate overall loss
loss = data_loss + regularization_loss
```

We created a new `regularization_loss` variable and added all layer's regularization losses to it. This completes the forward pass for regularization, but this also means our overall loss has changed since part of the calculation can include regularization, which must be accounted for in the backpropagation of the gradients. Thus, we will now cover the partial derivatives for both L1 and L2 regularization.

Backward pass

The derivative of L2 regularization is relatively simple:

$$L_{2w} = \lambda \sum_m w_m^2 \quad \rightarrow \quad \frac{\partial L_{2w}}{\partial w_m} = \frac{\partial}{\partial w_m}[\lambda \sum_m w_m^2] =$$

$$= \lambda \frac{\partial}{\partial w_m} w_m^2 = \lambda \cdot 2w_m^{2-1} = 2\lambda w_m$$

This might look complicated, but is one of the simpler derivative calculations that we have to derive in this book. Lambda is a constant, so we can move it outside of the derivative term. We can remove the sum operator since we calculate the partial derivative with respect to the given parameter only, and the sum of one element equals this element. So, we only need to calculate the derivative of w^2, which we know is $2w$. From the coding perspective, we will multiply all of the weights by 2λ. We'll implement this with NumPy directly as it's just a simple multiplication operation.

L1 regularization's derivative, on the other hand, requires more explanation. In the case of L1 regularization, we must calculate the derivative of the absolute value piecewise function, which effectively multiplies a value by -1 if it is less than 0; otherwise, it's multiplied by 1. This is because the absolute value function is linear for positive values, and we know that a linear function's derivative is:

$$f(x) = x \quad \rightarrow \quad f'(x) = 1$$

For negative values, it negates the sign of the value to make it positive. In other words, it multiplies values by -1:

$$f(x) = -x \quad \rightarrow \quad f'(x) = -1$$

When we combine that:

$$abs(x) = \begin{cases} x & x > 0 \\ -x & x < 0 \end{cases} \quad \rightarrow \quad abs'(x) = \begin{cases} 1 & x > 0 \\ -1 & x < 0 \end{cases}$$

And the complete partial derivative of L1 regularization with respect to given weight:

$$L_{1w} = \lambda \sum_m |w_m| \quad \rightarrow \quad L'_{1w} = \frac{\partial}{\partial w_m} \lambda \sum_m |w_m| = \lambda \frac{\partial}{\partial w_m} |w_m| = \lambda \begin{cases} 1 & w_m > 0 \\ -1 & w_m < 0 \end{cases}$$

Like L2 regularization, lambda is a constant, and we calculate the partial derivative of this regularization with respect to the specific input. The partial derivative, in this case, equals 1 or -1 depending on the w_m (weight) value.
We are calculating this derivative with respect to weights, and the resulting gradient, which has the same shape as the weights, is what we'll use to update the weights. To put this into pure Python code:

```
weights = [0.2, 0.8, -0.5]  # weights of one neuron
dL1 = []  # array of partial derivatives of L1 regularization
for weight in weights:
    if weight >= 0:
        dL1.append(1)
    else:
        dL1.append(-1)
print(dL1)
```

```
>>>
[1, 1, -1]
```

You may have noticed that we're using `>= 0` in the code where the equation above clearly depicts *> 0*. If we picture the `np.abs` function, it's a line going down and "bouncing" at the value *0*, like a saw tooth. At the pointed end (i.e., the value of *0*), the derivative of the `np.abs` function is undefined, but we cannot code it this way, so we need to handle this situation and break this rule a bit.

Now let's try to modify this L1 derivative to work with multiple neurons in a layer:

```
weights = [[0.2, 0.8, -0.5, 1],  # now we have 3 sets of weights
           [0.5, -0.91, 0.26, -0.5],
           [-0.26, -0.27, 0.17, 0.87]]
dL1 = []  # array of partial derivatives of L1 regularization
for neuron in weights:
    neuron_dL1 = []  # derivatives related to one neuron
    for weight in neuron:
        if weight >= 0:
            neuron_dL1.append(1)
        else:
            neuron_dL1.append(-1)
    dL1.append(neuron_dL1)
print(dL1)
```

```
>>>
[[1, 1, -1, 1], [1, -1, 1, -1], [-1, -1, 1, 1]]
```

That's the vanilla Python version, now for the NumPy version. With NumPy, we're going to use conditions and binary masks. We'll create the gradient as an array filled with values of *1* and shaped like weights, using `np.ones_like(weights)`. Next, the condition `weights < 0` returns an array of the same shape as `dL1`, containing `0` where the condition is false and `1` where it's true. We're using this as a binary mask to `dL1` to set values to *-1* only where the condition is true (where weight values are less than 0):

```
import numpy as np

weights = np.array([[0.2, 0.8, -0.5, 1],
                    [0.5, -0.91, 0.26, -0.5],
                    [-0.26, -0.27, 0.17, 0.87]])

dL1 = np.ones_like(weights)

dL1[weights < 0] = -1

print(dL1)
```

```
>>>
array([[ 1.,  1., -1.,  1.],
       [ 1., -1.,  1., -1.],
       [-1., -1.,  1.,  1.]])
```

This returned an array of the same shape containing values of 1 and -1 — the partial gradient of the `np.abs` function (we still have to multiply it by the lambda hyperparameter). We can now

take these and update the backward pass method for the dense layer object. For L1 regularization, we'll take the code above and multiply it by λ for weights and perform the same operation for biases. For L2 regularization, as discussed at the beginning of this chapter, all we need to do is take the weights/biases, multiply them by 2λ, and add that product to the gradients:

```
# Dense Layer
class Layer_Dense:
    ...
    # Backward pass
    def backward(self, dvalues):
        # Gradients on parameters
        self.dweights = np.dot(self.inputs.T, dvalues)
        self.dbiases = np.sum(dvalues, axis=0, keepdims=True)

        # Gradients on regularization
        # L1 on weights
        if self.weight_regularizer_l1 > 0:
            dL1 = np.ones_like(self.weights)
            dL1[self.weights < 0] = -1
            self.dweights += self.weight_regularizer_l1 * dL1
        # L2 on weights
        if self.weight_regularizer_l2 > 0:
            self.dweights += 2 * self.weight_regularizer_l2 * \
                                self.weights
        # L1 on biases
        if self.bias_regularizer_l1 > 0:
            dL1 = np.ones_like(self.biases)
            dL1[self.biases < 0] = -1
            self.dbiases += self.bias_regularizer_l1 * dL1
        # L2 on biases
        if self.bias_regularizer_l2 > 0:
            self.dbiases += 2 * self.bias_regularizer_l2 * \
                               self.biases

        # Gradient on values
        self.dinputs = np.dot(dvalues, self.weights.T)
```

With this, we can update our print to include new information — regularization loss and overall loss:

```
print(f'epoch: {epoch}, ' +
      f'acc: {accuracy:.3f}, ' +
      f'loss: {loss:.3f} (' +
      f'data_loss: {data_loss:.3f}, ' +
      f'reg_loss: {regularization_loss:.3f}), ' +
      f'lr: {optimizer.current_learning_rate}')
```

Then we can add weight and bias regularizer parameters when defining a layer:

```
# Create Dense layer with 2 input features and 3 output values
dense1 = Layer_Dense(2, 64, weight_regularizer_l2=5e-4,
                            bias_regularizer_l2=5e-4)
```

We usually add regularization terms to the hidden layers only. Even if we are calling the regularization method on the output layer as well, it won't modify gradients if we do not set the lambda hyperparameters to values other than *0*.

Full code up to this point:

```
import numpy as np
import nnfs
from nnfs.datasets import spiral_data

nnfs.init()


# Dense layer
class Layer_Dense:

    # Layer initialization
    def __init__(self, n_inputs, n_neurons,
                 weight_regularizer_l1=0, weight_regularizer_l2=0,
                 bias_regularizer_l1=0, bias_regularizer_l2=0):
        # Initialize weights and biases
        self.weights = 0.01 * np.random.randn(n_inputs, n_neurons)
        self.biases = np.zeros((1, n_neurons))
        # Set regularization strength
        self.weight_regularizer_l1 = weight_regularizer_l1
        self.weight_regularizer_l2 = weight_regularizer_l2
        self.bias_regularizer_l1 = bias_regularizer_l1
        self.bias_regularizer_l2 = bias_regularizer_l2

    # Forward pass
    def forward(self, inputs):
        # Remember input values
        self.inputs = inputs
        # Calculate output values from inputs, weights and biases
        self.output = np.dot(inputs, self.weights) + self.biases
```

```
    # Backward pass
    def backward(self, dvalues):
        # Gradients on parameters
        self.dweights = np.dot(self.inputs.T, dvalues)
        self.dbiases = np.sum(dvalues, axis=0, keepdims=True)

        # Gradients on regularization
        # L1 on weights
        if self.weight_regularizer_l1 > 0:
            dL1 = np.ones_like(self.weights)
            dL1[self.weights < 0] = -1
            self.dweights += self.weight_regularizer_l1 * dL1
        # L2 on weights
        if self.weight_regularizer_l2 > 0:
            self.dweights += 2 * self.weight_regularizer_l2 * \
                             self.weights
        # L1 on biases
        if self.bias_regularizer_l1 > 0:
            dL1 = np.ones_like(self.biases)
            dL1[self.biases < 0] = -1
            self.dbiases += self.bias_regularizer_l1 * dL1
        # L2 on biases
        if self.bias_regularizer_l2 > 0:
            self.dbiases += 2 * self.bias_regularizer_l2 * \
                            self.biases

        # Gradient on values
        self.dinputs = np.dot(dvalues, self.weights.T)


# ReLU activation
class Activation_ReLU:

    # Forward pass
    def forward(self, inputs):
        # Remember input values
        self.inputs = inputs
        # Calculate output values from inputs
        self.output = np.maximum(0, inputs)

    # Backward pass
    def backward(self, dvalues):
        # Since we need to modify original variable,
        # let's make a copy of values first
        self.dinputs = dvalues.copy()

        # Zero gradient where input values were negative
        self.dinputs[self.inputs <= 0] = 0
```

```
# Softmax activation
class Activation_Softmax:

    # Forward pass
    def forward(self, inputs):
        # Remember input values
        self.inputs = inputs

        # Get unnormalized probabilities
        exp_values = np.exp(inputs - np.max(inputs, axis=1,
                                            keepdims=True))
        # Normalize them for each sample
        probabilities = exp_values / np.sum(exp_values, axis=1,
                                            keepdims=True)

        self.output = probabilities

    # Backward pass
    def backward(self, dvalues):

        # Create uninitialized array
        self.dinputs = np.empty_like(dvalues)

        # Enumerate outputs and gradients
        for index, (single_output, single_dvalues) in \
                enumerate(zip(self.output, dvalues)):
            # Flatten output array
            single_output = single_output.reshape(-1, 1)
            # Calculate Jacobian matrix of the output
            jacobian_matrix = np.diagflat(single_output) - \
                              np.dot(single_output, single_output.T)
            # Calculate sample-wise gradient
            # and add it to the array of sample gradients
            self.dinputs[index] = np.dot(jacobian_matrix,
                                         single_dvalues)


# SGD optimizer
class Optimizer_SGD:

    # Initialize optimizer - set settings,
    # learning rate of 1. is default for this optimizer
    def __init__(self, learning_rate=1., decay=0., momentum=0.):
        self.learning_rate = learning_rate
        self.current_learning_rate = learning_rate
        self.decay = decay
        self.iterations = 0
        self.momentum = momentum
```

```
# Call once before any parameter updates
def pre_update_params(self):
    if self.decay:
        self.current_learning_rate = self.learning_rate * \
            (1. / (1. + self.decay * self.iterations))

# Update parameters
def update_params(self, layer):

    # If we use momentum
    if self.momentum:

        # If layer does not contain momentum arrays, create them
        # filled with zeros
        if not hasattr(layer, 'weight_momentums'):
            layer.weight_momentums = np.zeros_like(layer.weights)
            # If there is no momentum array for weights
            # The array doesn't exist for biases yet either.
            layer.bias_momentums = np.zeros_like(layer.biases)

        # Build weight updates with momentum - take previous
        # updates multiplied by retain factor and update with
        # current gradients
        weight_updates = \
            self.momentum * layer.weight_momentums - \
            self.current_learning_rate * layer.dweights
        layer.weight_momentums = weight_updates

        # Build bias updates
        bias_updates = \
            self.momentum * layer.bias_momentums - \
            self.current_learning_rate * layer.dbiases
        layer.bias_momentums = bias_updates

    # Vanilla SGD updates (as before momentum update)
    else:
        weight_updates = -self.current_learning_rate * \
                         layer.dweights
        bias_updates = -self.current_learning_rate * \
                       layer.dbiases

    # Update weights and biases using either
    # vanilla or momentum updates
    layer.weights += weight_updates
    layer.biases += bias_updates

# Call once after any parameter updates
def post_update_params(self):
    self.iterations += 1
```

```python
# Adagrad optimizer
class Optimizer_Adagrad:

    # Initialize optimizer - set settings
    def __init__(self, learning_rate=1., decay=0., epsilon=1e-7):
        self.learning_rate = learning_rate
        self.current_learning_rate = learning_rate
        self.decay = decay
        self.iterations = 0
        self.epsilon = epsilon

    # Call once before any parameter updates
    def pre_update_params(self):
        if self.decay:
            self.current_learning_rate = self.learning_rate * \
                (1. / (1. + self.decay * self.iterations))

    # Update parameters
    def update_params(self, layer):

        # If layer does not contain cache arrays,
        # create them filled with zeros
        if not hasattr(layer, 'weight_cache'):
            layer.weight_cache = np.zeros_like(layer.weights)
            layer.bias_cache = np.zeros_like(layer.biases)

        # Update cache with squared current gradients
        layer.weight_cache += layer.dweights**2
        layer.bias_cache += layer.dbiases**2

        # Vanilla SGD parameter update + normalization
        # with square rooted cache
        layer.weights += -self.current_learning_rate * \
                         layer.dweights / \
                         (np.sqrt(layer.weight_cache) + self.epsilon)
        layer.biases += -self.current_learning_rate * \
                        layer.dbiases / \
                        (np.sqrt(layer.bias_cache) + self.epsilon)

    # Call once after any parameter updates
    def post_update_params(self):
        self.iterations += 1
```

```python
# RMSprop optimizer
class Optimizer_RMSprop:

    # Initialize optimizer - set settings
    def __init__(self, learning_rate=0.001, decay=0., epsilon=1e-7,
                 rho=0.9):
        self.learning_rate = learning_rate
        self.current_learning_rate = learning_rate
        self.decay = decay
        self.iterations = 0
        self.epsilon = epsilon
        self.rho = rho

    # Call once before any parameter updates
    def pre_update_params(self):
        if self.decay:
            self.current_learning_rate = self.learning_rate * \
                (1. / (1. + self.decay * self.iterations))

    # Update parameters
    def update_params(self, layer):

        # If layer does not contain cache arrays,
        # create them filled with zeros
        if not hasattr(layer, 'weight_cache'):
            layer.weight_cache = np.zeros_like(layer.weights)
            layer.bias_cache = np.zeros_like(layer.biases)

        # Update cache with squared current gradients
        layer.weight_cache = self.rho * layer.weight_cache + \
            (1 - self.rho) * layer.dweights**2
        layer.bias_cache = self.rho * layer.bias_cache + \
            (1 - self.rho) * layer.dbiases**2

        # Vanilla SGD parameter update + normalization
        # with square rooted cache
        layer.weights += -self.current_learning_rate * \
                         layer.dweights / \
                         (np.sqrt(layer.weight_cache) + self.epsilon)
        layer.biases += -self.current_learning_rate * \
                        layer.dbiases / \
                        (np.sqrt(layer.bias_cache) + self.epsilon)

    # Call once after any parameter updates
    def post_update_params(self):
        self.iterations += 1
```

```python
# Adam optimizer
class Optimizer_Adam:

    # Initialize optimizer - set settings
    def __init__(self, learning_rate=0.001, decay=0., epsilon=1e-7,
                 beta_1=0.9, beta_2=0.999):
        self.learning_rate = learning_rate
        self.current_learning_rate = learning_rate
        self.decay = decay
        self.iterations = 0
        self.epsilon = epsilon
        self.beta_1 = beta_1
        self.beta_2 = beta_2

    # Call once before any parameter updates
    def pre_update_params(self):
        if self.decay:
            self.current_learning_rate = self.learning_rate * \
                (1. / (1. + self.decay * self.iterations))

    # Update parameters
    def update_params(self, layer):

        # If layer does not contain cache arrays,
        # create them filled with zeros
        if not hasattr(layer, 'weight_cache'):
            layer.weight_momentums = np.zeros_like(layer.weights)
            layer.weight_cache = np.zeros_like(layer.weights)
            layer.bias_momentums = np.zeros_like(layer.biases)
            layer.bias_cache = np.zeros_like(layer.biases)

        # Update momentum  with current gradients
        layer.weight_momentums = self.beta_1 * \
                                 layer.weight_momentums + \
                                 (1 - self.beta_1) * layer.dweights
        layer.bias_momentums = self.beta_1 * \
                               layer.bias_momentums + \
                               (1 - self.beta_1) * layer.dbiases
        # Get corrected momentum
        # self.iteration is 0 at first pass
        # and we need to start with 1 here
        weight_momentums_corrected = layer.weight_momentums / \
            (1 - self.beta_1 ** (self.iterations + 1))
        bias_momentums_corrected = layer.bias_momentums / \
            (1 - self.beta_1 ** (self.iterations + 1))
        # Update cache with squared current gradients
        layer.weight_cache = self.beta_2 * layer.weight_cache + \
            (1 - self.beta_2) * layer.dweights**2
```

```
        layer.bias_cache = self.beta_2 * layer.bias_cache + \
            (1 - self.beta_2) * layer.dbiases**2
        # Get corrected cache
        weight_cache_corrected = layer.weight_cache / \
            (1 - self.beta_2 ** (self.iterations + 1))
        bias_cache_corrected = layer.bias_cache / \
            (1 - self.beta_2 ** (self.iterations + 1))

        # Vanilla SGD parameter update + normalization
        # with square rooted cache
        layer.weights += -self.current_learning_rate * \
                         weight_momentums_corrected / \
                         (np.sqrt(weight_cache_corrected) +
                             self.epsilon)
        layer.biases += -self.current_learning_rate * \
                         bias_momentums_corrected / \
                         (np.sqrt(bias_cache_corrected) +
                             self.epsilon)

    # Call once after any parameter updates
    def post_update_params(self):
        self.iterations += 1


# Common loss class
class Loss:

    # Regularization loss calculation
    def regularization_loss(self, layer):

        # 0 by default
        regularization_loss = 0

        # L1 regularization - weights
        # calculate only when factor greater than 0
        if layer.weight_regularizer_l1 > 0:
            regularization_loss += layer.weight_regularizer_l1 * \
                                   np.sum(np.abs(layer.weights))

        # L2 regularization - weights
        if layer.weight_regularizer_l2 > 0:
            regularization_loss += layer.weight_regularizer_l2 * \
                                   np.sum(layer.weights *
                                          layer.weights)
```

```
        # L1 regularization - biases
        # calculate only when factor greater than 0
        if layer.bias_regularizer_l1 > 0:
            regularization_loss += layer.bias_regularizer_l1 * \
                                   np.sum(np.abs(layer.biases))

        # L2 regularization - biases
        if layer.bias_regularizer_l2 > 0:
            regularization_loss += layer.bias_regularizer_l2 * \
                                   np.sum(layer.biases *
                                          layer.biases)

        return regularization_loss

    # Calculates the data and regularization losses
    # given model output and ground truth values
    def calculate(self, output, y):

        # Calculate sample losses
        sample_losses = self.forward(output, y)

        # Calculate mean loss
        data_loss = np.mean(sample_losses)

        # Return loss
        return data_loss


# Cross-entropy loss
class Loss_CategoricalCrossentropy(Loss):

    # Forward pass
    def forward(self, y_pred, y_true):

        # Number of samples in a batch
        samples = len(y_pred)

        # Clip data to prevent division by 0
        # Clip both sides to not drag mean towards any value
        y_pred_clipped = np.clip(y_pred, 1e-7, 1 - 1e-7)

        # Probabilities for target values -
        # only if categorical labels
        if len(y_true.shape) == 1:
            correct_confidences = y_pred_clipped[
                range(samples),
                y_true
            ]
```

```
        # Mask values - only for one-hot encoded labels
        elif len(y_true.shape) == 2:
            correct_confidences = np.sum(
                y_pred_clipped * y_true,
                axis=1
            )

        # Losses
        negative_log_likelihoods = -np.log(correct_confidences)
        return negative_log_likelihoods

    # Backward pass
    def backward(self, dvalues, y_true):

        # Number of samples
        samples = len(dvalues)
        # Number of labels in every sample
        # We'll use the first sample to count them
        labels = len(dvalues[0])

        # If labels are sparse, turn them into one-hot vector
        if len(y_true.shape) == 1:
            y_true = np.eye(labels)[y_true]

        # Calculate gradient
        self.dinputs = -y_true / dvalues
        # Normalize gradient
        self.dinputs = self.dinputs / samples


# Softmax classifier - combined Softmax activation
# and cross-entropy loss for faster backward step
class Activation_Softmax_Loss_CategoricalCrossentropy():

    # Creates activation and loss function objects
    def __init__(self):
        self.activation = Activation_Softmax()
        self.loss = Loss_CategoricalCrossentropy()

    # Forward pass
    def forward(self, inputs, y_true):
        # Output layer's activation function
        self.activation.forward(inputs)
        # Set the output
        self.output = self.activation.output
        # Calculate and return loss value
        return self.loss.calculate(self.output, y_true)
```

```
    # Backward pass
    def backward(self, dvalues, y_true):

        # Number of samples
        samples = len(dvalues)

        # If labels are one-hot encoded,
        # turn them into discrete values
        if len(y_true.shape) == 2:
            y_true = np.argmax(y_true, axis=1)

        # Copy so we can safely modify
        self.dinputs = dvalues.copy()
        # Calculate gradient
        self.dinputs[range(samples), y_true] -= 1
        # Normalize gradient
        self.dinputs = self.dinputs / samples


# Create dataset
X, y = spiral_data(samples=100, classes=3)

# Create Dense layer with 2 input features and 64 output values
dense1 = Layer_Dense(2, 64, weight_regularizer_l2=5e-4,
                            bias_regularizer_l2=5e-4)

# Create ReLU activation (to be used with Dense layer):
activation1 = Activation_ReLU()

# Create second Dense layer with 64 input features (as we take output
# of previous layer here) and 3 output values (output values)
dense2 = Layer_Dense(64, 3)

# Create Softmax classifier's combined loss and activation
loss_activation = Activation_Softmax_Loss_CategoricalCrossentropy()

# Create optimizer
optimizer = Optimizer_Adam(learning_rate=0.02, decay=5e-7)

# Train in loop
for epoch in range(10001):

    # Perform a forward pass of our training data through this layer
    dense1.forward(X)

    # Perform a forward pass through activation function
    # takes the output of first dense layer here
    activation1.forward(dense1.output)
```

```
# Perform a forward pass through second Dense layer
# takes outputs of activation function of first layer as inputs
dense2.forward(activation1.output)


# Perform a forward pass through the activation/loss function
# takes the output of second dense layer here and returns loss
data_loss = loss_activation.forward(dense2.output, y)

# Calculate regularization penalty
regularization_loss = \
    loss_activation.loss.regularization_loss(dense1) + \
    loss_activation.loss.regularization_loss(dense2)

# Calculate overall loss
loss = data_loss + regularization_loss

# Calculate accuracy from output of activation2 and targets
# calculate values along first axis
predictions = np.argmax(loss_activation.output, axis=1)
if len(y.shape) == 2:
    y = np.argmax(y, axis=1)
accuracy = np.mean(predictions == y)

if not epoch % 100:
    print(f'epoch: {epoch}, ' +
          f'acc: {accuracy:.3f}, ' +
          f'loss: {loss:.3f} (' +
          f'data_loss: {data_loss:.3f}, ' +
          f'reg_loss: {regularization_loss:.3f}), ' +
          f'lr: {optimizer.current_learning_rate}')

# Backward pass
loss_activation.backward(loss_activation.output, y)
dense2.backward(loss_activation.dinputs)
activation1.backward(dense2.dinputs)
dense1.backward(activation1.dinputs)

# Update weights and biases
optimizer.pre_update_params()
optimizer.update_params(dense1)
optimizer.update_params(dense2)
optimizer.post_update_params()
```

```
# Validate the model

# Create test dataset
X_test, y_test = spiral_data(samples=100, classes=3)
# Perform a forward pass of our testing data through this layer
dense1.forward(X_test)

# Perform a forward pass through activation function
# takes the output of first dense layer here
activation1.forward(dense1.output)

# Perform a forward pass through second Dense layer
# takes outputs of activation function of first layer as inputs
dense2.forward(activation1.output)

# Perform a forward pass through the activation/loss function
# takes the output of second dense layer here and returns loss
loss = loss_activation.forward(dense2.output, y_test)

# Calculate accuracy from output of activation2 and targets
# calculate values along first axis
predictions = np.argmax(loss_activation.output, axis=1)
if len(y_test.shape) == 2:
    y_test = np.argmax(y_test, axis=1)
accuracy = np.mean(predictions == y_test)

print(f'validation, acc: {accuracy:.3f}, loss: {loss:.3f}')
```

```
>>>
...
epoch: 10000, acc: 0.947, loss: 0.217 (data_loss: 0.157, reg_loss: 0.060),
lr: 0.019900507413187767
validation, acc: 0.830, loss: 0.435
```

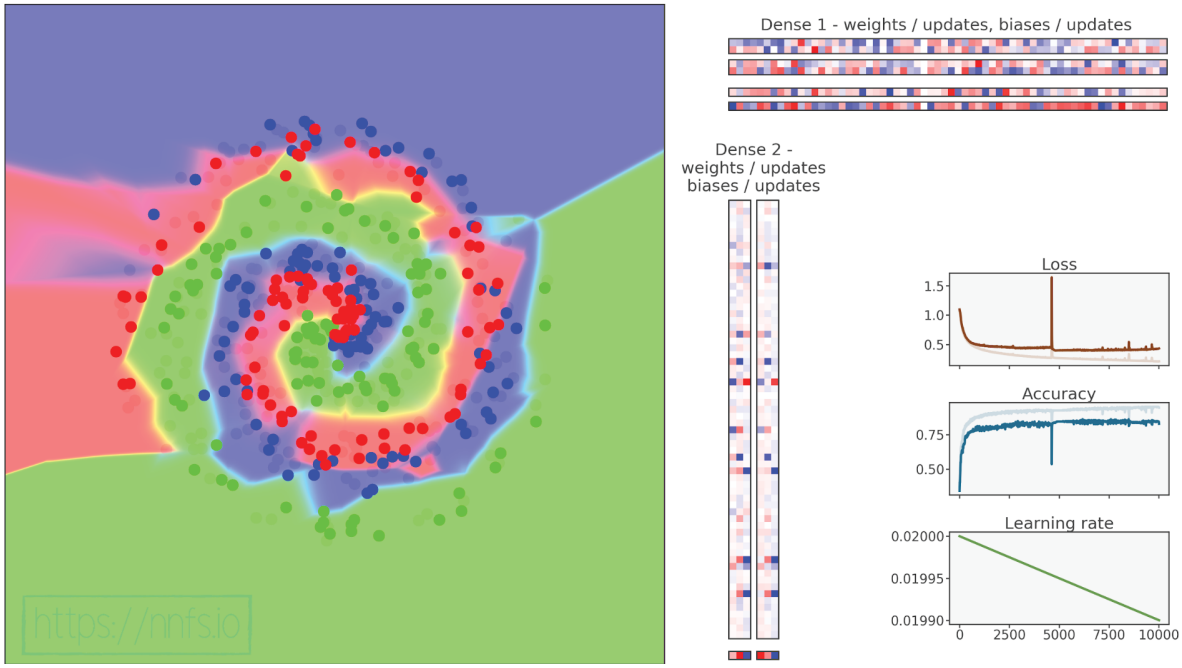


Fig 14.01: Training with regularization

Anim 14.01: https://nnfs.io/abc

This animation shows the training data in the background (dimmed dots) and the validation data in the foreground. After adding the L2 regularization term to the hidden layer, we achieved a lower validation loss (0.858 before adding regularization in, 0.435 now) and higher accuracy (0.803 before, 0.830 now). We can also take a moment to exemplify how a simple increase in data for training can make a large difference. If we grow from 100 samples to 1,000 samples:

```
# Create dataset
X, y = spiral_data(samples=1000, classes=3)
```

And run the code again:

```
>>>
epoch: 10000, acc: 0.895, loss: 0.357 (data_loss: 0.293, reg_loss: 0.063),
lr: 0.019900507413187767
validation, acc: 0.873, loss: 0.332
```

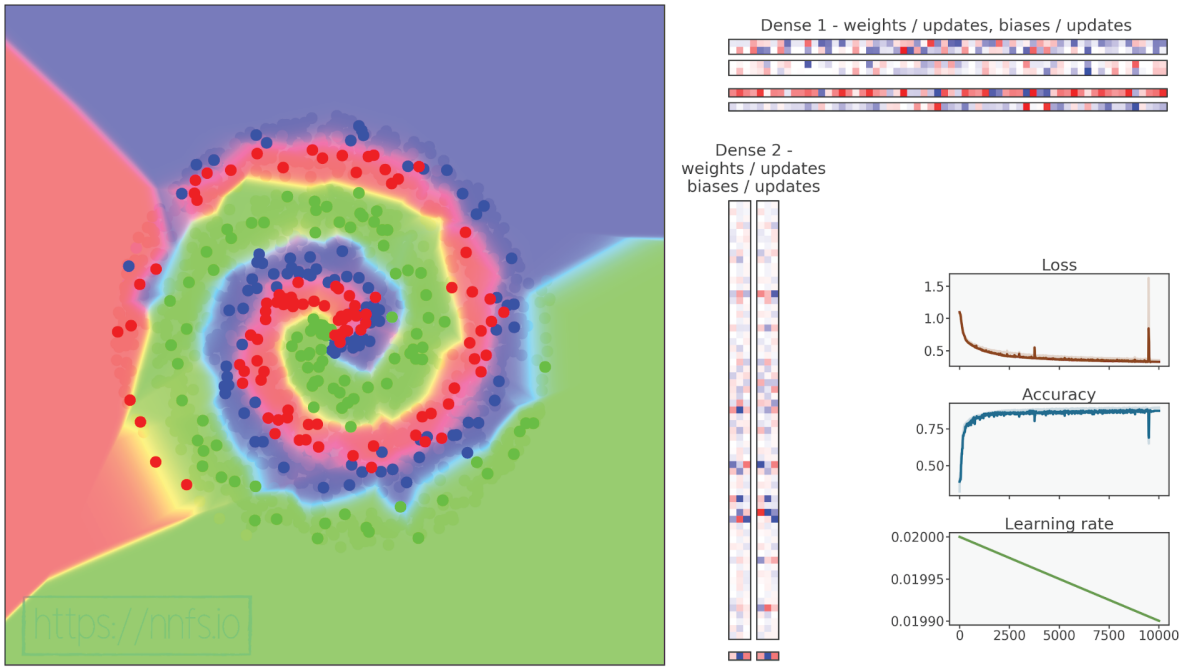


Fig 14.02: Training with regularization and more training data.

Anim 14.02: https://nnfs.io/bcd

We can see that this change alone also had a considerable impact on both validation accuracy in general, as well as the delta between the validation and training accuracies — lower accuracy and higher training loss suggest that the capacity of the model might be too low. A large delta earlier and a small one now suggests that the model was most likely overfitting previously. In theory, this regularization allows us to create much larger models without fear of overfitting (or memorization). We can test this by increasing the number of neurons per layer. Going with 128 or 256 neurons per layer helps with the training accuracy but not that much with the validation accuracy:

```
# Create Dense layer with 2 input features and 256 output values
dense1 = Layer_Dense(2, 256, weight_regularizer_l2=5e-4,
                             bias_regularizer_l2=5e-4)
```

```
# Create ReLU activation (to be used with Dense layer):
activation1 = Activation_ReLU()

# Create second Dense layer with 256 input features (as we take output
# of previous layer here) and 3 output values (output values)
dense2 = Layer_Dense(256, 3)
```

```
>>>
epoch: 10000, acc: 0.924, loss: 0.253 (data_loss: 0.210, reg_loss: 0.043),
lr: 0.019900507413187767
validation, acc: 0.887, loss: 0.331
```

This didn't produce much of a change in results, but raising this number again to 512 did improve validation accuracy and loss as well:

```
# Create Dense layer with 2 input features and 512 output values
dense1 = Layer_Dense(2, 512, weight_regularizer_l2=5e-4,
                             bias_regularizer_l2=5e-4)

# Create ReLU activation (to be used with Dense layer):
activation1 = Activation_ReLU()

# Create second Dense layer with 512 input features (as we take output
# of previous layer here) and 3 output values (output values)
dense2 = Layer_Dense(512, 3)
```

```
>>>
epoch: 10000, acc: 0.918, loss: 0.253 (data_loss: 0.210, reg_loss: 0.043),
lr: 0.019900507413187767
validation, acc: 0.920, loss: 0.256
```

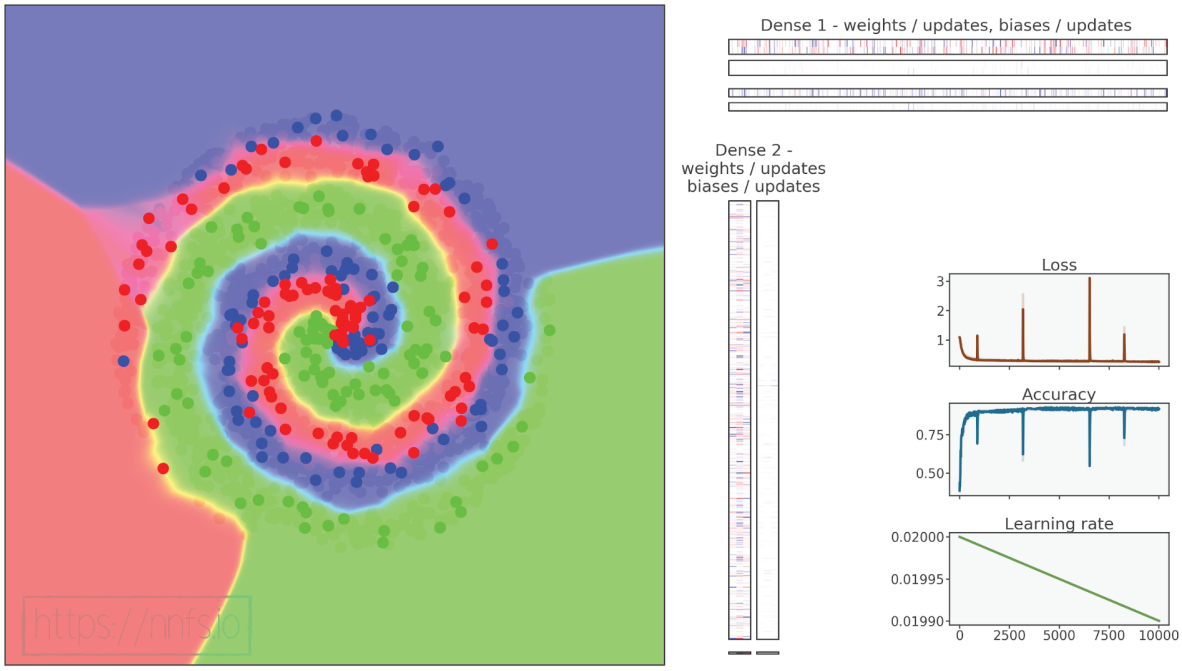


Fig 14.03: Training with regularization and more training data (tuned).

Anim 14.03: https://nnfs.io/cde

In this case, we see that the accuracies and losses for in-sample and out-of-sample data are almost identical. From here, we could add either more layers and neurons or both. Feel free to tinker with this to try to improve it. Next, we're going to cover another regularization method: **dropout**.

Supplementary Material: https://nnfs.io/ch14
Chapter code, further resources, and errata for this chapter.

Chapter 15

Dropout

Another option for neural network regularization is adding a **dropout layer**. This type of layer disables some neurons, while the others pass through unchanged. The idea here, similarly to regularization, is to prevent a neural network from becoming too dependent on any neuron or for any neuron to be relied upon entirely in a specific instance (which can be common if a model overfits the training data). Another problem dropout can help with is **co-adoption**, which happens when neurons depend on the output values of other neurons and do not learn the underlying function on their own. Dropout can also help with **noise** and other perturbations in the training data as more neurons working together mean that the model can learn more complex functions.

The Dropout function works by randomly disabling neurons at a given rate during every forward pass, forcing the network to learn how to make accurate predictions with only a random part of neurons remaining. Dropout forces the model to use more neurons for the same purpose, resulting in a higher chance of learning the underlying function that describes the data. For example, if we disable one half of the neurons during the current step, and the other half during the next step, we are forcing more neurons to learn the data, as only a part of them "sees" the data and gets updates in a given pass. These alternating halves of neurons are an example, and in reality, we'll use a hyperparameter to inform the dropout layer of the number of neurons to disable randomly.

Also, since active neurons are changing, dropout helps prevent overfitting, as the model can't use specific neurons to memorize certain samples. It's also worth mentioning that the dropout layer does not truly disable neurons, but instead zeroes their outputs. In other words, dropout does not decrease the number of neurons used, nor does it make the training process twice as fast when half the neurons are disabled.

Forward Pass

In the code, we will "turn off" neurons with a filter that is an array with the same shape as the layer output but filled with numbers drawn from a Bernoulli distribution. A **Bernoulli distribution** is a binary (or discrete) probability distribution where we can get a value of *1* with a probability of *p* and value of *0* with a probability of *q*. Let's take some random value from this distribution, r_i, then:

$$P(r_i = 1) = p$$

$$P(r_i = 0) = q = 1 - p = 1 - P(r_i = 1)$$

What this means is that the probability of this value being *1* is *p*. The probability of it being *0* is *q* = *1* - *p*, therefore:

$$r_i \sim Bernoulli(p)$$

This means that the given r_i is an equivalent of a value from the Bernoulli distribution with a probability *p* for this value to be *1*. If r_i is a single value from this distribution, a draw from this distribution, reshaped to match the shape of the layer outputs, can be used as a mask to these outputs.

We are returned an array filled with values of *1* with a probability of *p* and otherwise values of *0*. We then apply this filter to the output of a layer we want to add dropout to.

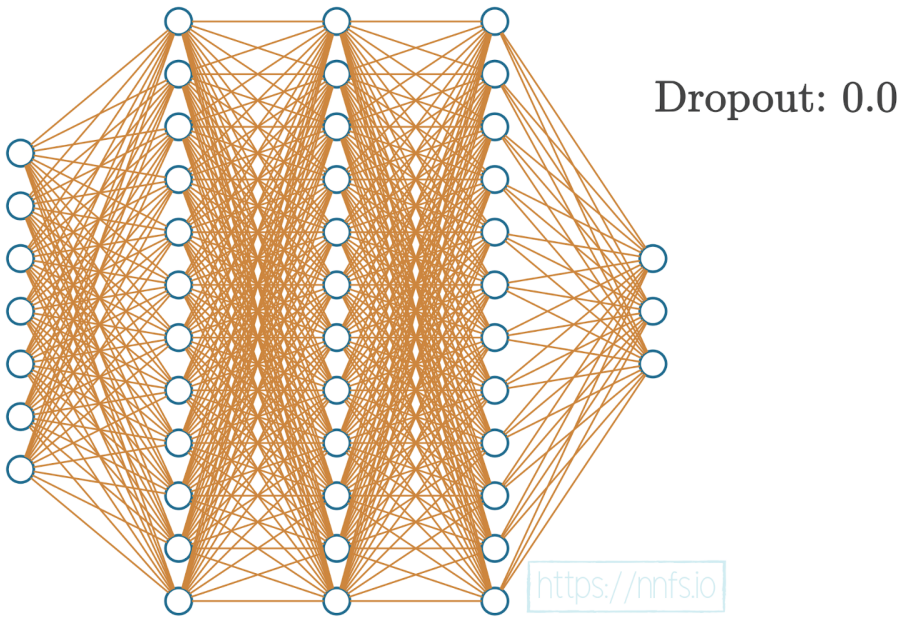


Fig 15.01: Example model with no dropout applied.

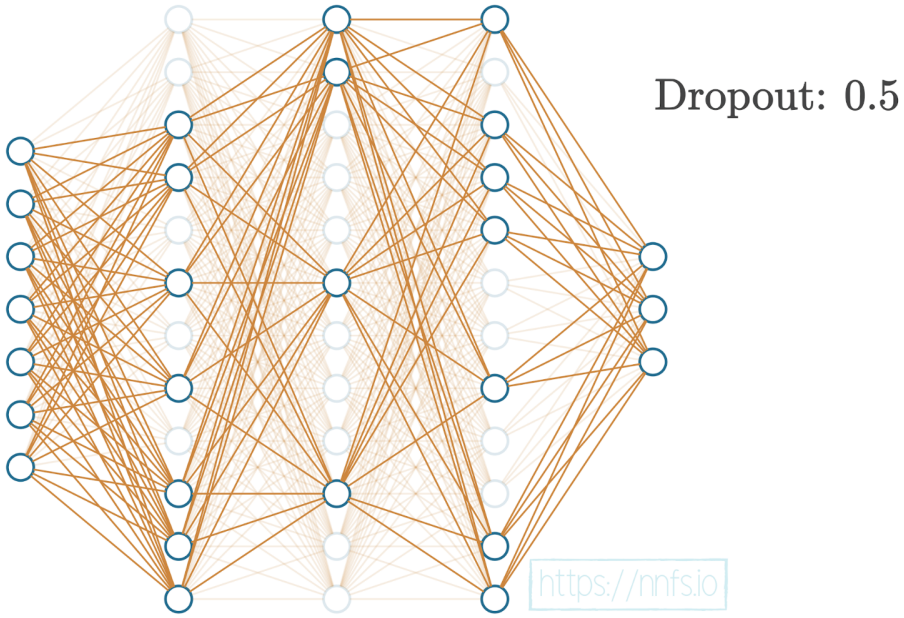


Fig 15.02: Example model with 0.5 dropout.

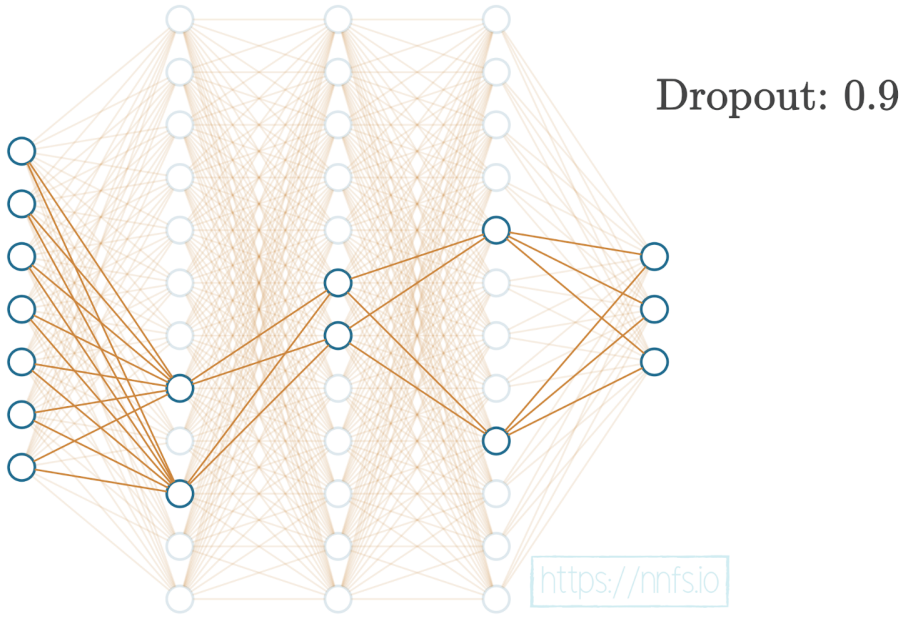


Fig 15.03: Example model with 0.9 dropout.

Anim 15.01-15.03: https://nnfs.io/def

With the code, we have one hyperparameter for a dropout layer. This is a value for the percentage of neurons to disable in that layer. For example, if you chose 0.10 for the dropout parameter, 10% of the neurons will be disabled at random during each forward pass. Before we use NumPy, we'll demonstrate this with an example in pure Python:

```
import random

dropout_rate = 0.5
# Example output containing 10 values
example_output = [0.27, -1.03, 0.67, 0.99, 0.05,
                  -0.37, -2.01, 1.13, -0.07, 0.73]
```

```
# Repeat as long as necessary
while True:

    # Randomly choose index and set value to 0
    index = random.randint(0, len(example_output) - 1)
    example_output[index] = 0

    # We might set an index that already is zeroed
    # There are different ways of overcoming this problem,
    # for simplicity we count values that are exactly 0
    # while it's extremely rare in real model that weights
    # are exactly 0, this is not the best method for sure
    dropped_out = 0
    for value in example_output:
        if value == 0:
            dropped_out += 1

    # If required number of outputs is zeroed - leave the loop
    if dropped_out / len(example_output) >= dropout_rate:
        break

print(example_output)


>>>
[0, -1.03, 0.67, 0.99, 0, -0.37, 0, 0, 0, 0.73]
```

The code is relatively rudimentary, but the idea is to keep zeroing neuron outputs (setting them to 0) randomly until we've disabled whatever target % of neurons we require. If we consider a Bernoulli distribution as a special case of a Binomial distribution with $n=1$ and look at a list of available methods in NumPy, it turns out that there's a much cleaner way to do this using *numpy.random.binomial*. A binomial distribution differs from Bernoulli distribution in one way, as it adds a parameter, *n,* which is the number of concurrent experiments (instead of just one) and returns the number of successes from these *n* experiments.

`np.random.binomial()` works by taking the already discussed parameters *n* (number of experiments) and *p* (probability of the true value of the experiment) as well as an additional parameter *size*: `np.random.binomial(n, p, size)`.

The function itself can be thought of like a coin toss, where the result will be 0 or 1. The *n* is how many tosses of the coin do you want to do. The *p* is the probability for the toss result to be a 1. The overall result is a sum of all toss results. The *size* is how many of these "tests" to run, and the return is a list of overall results. For example:

```
np.random.binomial(2, 0.5, size=10)
```

This will produce an array that is of size 10, where each element will be the sum of 2 coin tosses, where the probability of 1 will be 0.5, or 50%. The resulting array:

```
array([0, 0, 1, 2, 0, 2, 0, 1, 0, 2])
```

We can use this to create our dropout layer. Our goal here is to create a filter where the intended dropout % is represented as 0, with everything else as 1. For example, let's say we have a dropout layer that we'll add after a layer that consists of 5 neurons, and we wish to have a 20% dropout. An example of a dropout layer might look like:

```
[1, 0, 1, 1, 1]
```

As you can see, ⅕ of that list is a 0. This is an example of the filter we're going to apply to the output of the dense layer. If we multiplied a neural network's layer output by this, we'd be effectively disabling the neuron at the same index as the 0.
We can mimic that with `np.random.binomial()` by doing:

```
dropout_rate = 0.20
print(np.random.binomial(1, 1-dropout_rate, size=5))

>>>
array([0, 1, 1, 1, 1])
```

This is based on probabilities, so there will be times when it does not look like the above array. There could be times no neurons zero out, or all neurons zero out. On average, these random draws will tend toward the probability we desire. Also, this was an example using a very small layer (5 neurons). On a realistically sized layer, you should find the probability more consistently matches your intended value.

Assume a neural network layer's output is:

```
example_output = np.array([0.27, -1.03, 0.67, 0.99, 0.05,
                           -0.37, -2.01, 1.13, -0.07, 0.73])
```

Next, let's assume our target dropout rate is 0.3, or 30%. We apply a dropout layer:

```
import numpy as np

dropout_rate = 0.3
example_output = np.array([0.27, -1.03, 0.67, 0.99, 0.05,
                           -0.37, -2.01, 1.13, -0.07, 0.73])

example_output *= np.random.binomial(1, 1-dropout_rate,
                                     example_output.shape)

print(example_output)


>>>
[ 0.27 -1.03  0.00  0.99  0.   -0.37 -2.01  1.13 -0.07  0.  ]
```

Note that our dropout rate is the ratio of neurons we intend to *disable (q)*. Sometimes, the implementation of dropout will include a rate parameter that instead means the fraction of neurons you intend to *keep (p)*. At the time of writing this, the dropout parameter in deep learning frameworks, TensorFlow and Keras, represents the neurons you intend to disable. On the other hand, the dropout parameter in PyTorch and the original paper on dropout (http://www.cs.toronto.edu/~rsalakhu/papers/srivastava14a.pdf) signal the ratio of neurons you intend to keep.

The way it's implemented is not important. What *is* important is that you know which method you're using!

While dropout helps a neural network generalize and is helpful for training, it's not something we want to utilize when predicting. It's not as simple as only omitting it because the magnitude of inputs to the next neurons can be dramatically different. If you have a dropout of 50%, for example, this would suggest that, on average, your inputs to the next layer neurons will be 50% smaller when summed, assuming they are fully-connected. What that means is that we used dropout during training, and, in this example, a random 50% of neurons output a value of 0 at each of the steps. Neurons in the next layer multiply inputs by weights, sum them, and receive values of 0 for half of their inputs. If we don't use dropout during prediction, all neurons will output their values, and this state won't match the state seen during training, since the sums will be statistically about twice as big. To handle this, during prediction, we might multiply all of the outputs by the dropout fraction, but that'd add another step for the forward pass, and there is a better way to achieve this. Instead, we want to scale the data back up after a dropout, during the training phase, to mimic the mean of the sum when all of the neurons output their values. *Example_output* becomes:

```
example_output *= np.random.binomial(1, 1-dropout_rate,
                                     example_output.shape) / \
                  (1-dropout_rate)
```

Notice that we added the division of the dropout's result by the dropout rate. Since this rate is a fraction, it makes the resulting values larger, accounting for the value lost because a fraction of the neuron outputs being zeroed out. This way, we don't have to worry about the prediction and can simply omit the dropout during prediction. In any specific example, you will find that scaling doesn't equal the same sum as before because we're randomly dropping neurons. That said, after enough samples, the scaling will average out overall. To prove this:

```
import numpy as np

dropout_rate = 0.2
example_output = np.array([0.27, -1.03, 0.67, 0.99, 0.05,
                           -0.37, -2.01, 1.13, -0.07, 0.73])
print(f'sum initial {sum(example_output)}')

sums = []
for i in range(10000):

    example_output2 = example_output * \
        np.random.binomial(1, 1-dropout_rate, example_output.shape) / \
        (1-dropout_rate)
    sums.append(sum(example_output2))

print(f'mean sum: {np.mean(sums)}')


>>>
sum initial 0.36000000000000015
mean sum: 0.36282000000000014
```

It's not exact yet, but you should get the idea.

Backward Pass

The last missing piece to implement dropout as a layer is a backward pass method. As before, we need to calculate the partial derivative of the dropout operation:

When the value of element r_i equals *1*, its function and derivative becomes the neuron's output, *z*, compensated for the loss value by *1-q*, where *q* is the dropout rate, as we just described:

$$f(z,q) = \frac{z}{1-q} \quad \rightarrow \quad \frac{\partial}{\partial z}[\frac{z}{1-q}] = \frac{1}{1-q} \cdot \frac{\partial}{\partial z} z = \frac{1}{1-q} \cdot 1 = \frac{1}{1-q}$$

That's because the derivative with respect to *z* of *z* is 1, and we treat the rest as a constant. Whenr_i*=0*:

$$f(z,q) = 0 \quad \rightarrow \quad \frac{\partial}{\partial z} 0 = 0$$

And that's because we are zeroing this element of the dropout filter, and the derivative of any constant value (including *0*) is *0*. Let's combine both cases and denote *Dropout* as *Dr*:

$$Dr_i = \begin{cases} \frac{z_i}{1-q} & r_i = 1 \\ 0 & r_i = 0 \end{cases} \quad \rightarrow \quad \frac{\partial}{\partial z_i} Dr_i = \begin{cases} \frac{1}{1-q} & r_i = 1 \\ 0 & r_i = 0 \end{cases} = \frac{r_i}{1-q}$$

i denotes the index of the given input (and the layer output). When we write a derivative of the dropout function this way, we can simplify it to a value from the Bernoulli distribution divided by *1-q*, which is identical to our scaled mask, the function the dropout applies during the forward pass, as it's also either *1* divided by *1-q*, or *0*. Thus, we can save this mask during the forward pass and use it with the chain rule as the gradient of this function.

The Code

We can now implement this concept in a new layer type, the dropout layer:

```
# Dropout
class Layer_Dropout:

    # Init
    def __init__(self, rate):
        # Store rate, we invert it as for example for dropout
        # of 0.1 we need success rate of 0.9
        self.rate = 1 - rate

    # Forward pass
    def forward(self, inputs):
        # Save input values
        self.inputs = inputs
        # Generate and save scaled mask
        self.binary_mask = np.random.binomial(1, self.rate,
                           size=inputs.shape) / self.rate
        # Apply mask to output values
        self.output = inputs * self.binary_mask

    # Backward pass
    def backward(self, dvalues):
        # Gradient on values
        self.dinputs = dvalues * self.binary_mask
```

Let's take this new dropout layer, and add it between our two dense layers. First defining it:

```
# Create Dense layer with 2 input features and 64 output values
dense1 = Layer_Dense(2, 64, weight_regularizer_l2=5e-4,
                            bias_regularizer_l2=5e-4)

# Create ReLU activation (to be used with Dense layer):
activation1 = Activation_ReLU()

# Create dropout layer
dropout1 = Layer_Dropout(0.1)
```

```
# Create second Dense layer with 64 input features (as we take output
# of previous layer here) and 3 output values (output values)
dense2 = Layer_Dense(64, 3)
```

During the forward pass, add in the dropout:

```
    # Perform a forward pass through Dropout layer
    dropout1.forward(activation1.output)

    # Perform a forward pass through second Dense layer
    # takes outputs of activation function of first layer as inputs
    dense2.forward(dropout1.output)
```

And of course in the backward pass:

```
    dropout1.backward(dense2.dinputs)
    activation1.backward(dropout1.dinputs)
```

Let's also raise the learning rate a bit, from 0.02 to 0.05 and raise the learning rate decaying from 5e-7 to 5e-5 as these parameters work better with our model and dropout layer.

Full code up to now:

```
import numpy as np
import nnfs
from nnfs.datasets import spiral_data

nnfs.init()


# Dense layer
class Layer_Dense:

    # Layer initialization
    def __init__(self, n_inputs, n_neurons,
                 weight_regularizer_l1=0, weight_regularizer_l2=0,
                 bias_regularizer_l1=0, bias_regularizer_l2=0):
        # Initialize weights and biases
        self.weights = 0.01 * np.random.randn(n_inputs, n_neurons)
        self.biases = np.zeros((1, n_neurons))
```

```
        # Set regularization strength
        self.weight_regularizer_l1 = weight_regularizer_l1
        self.weight_regularizer_l2 = weight_regularizer_l2
        self.bias_regularizer_l1 = bias_regularizer_l1
        self.bias_regularizer_l2 = bias_regularizer_l2

    # Forward pass
    def forward(self, inputs):
        # Remember input values
        self.inputs = inputs
        # Calculate output values from inputs, weights and biases
        self.output = np.dot(inputs, self.weights) + self.biases




    # Backward pass
    def backward(self, dvalues):
        # Gradients on parameters
        self.dweights = np.dot(self.inputs.T, dvalues)
        self.dbiases = np.sum(dvalues, axis=0, keepdims=True)

        # Gradients on regularization
        # L1 on weights
        if self.weight_regularizer_l1 > 0:
            dL1 = np.ones_like(self.weights)
            dL1[self.weights < 0] = -1
            self.dweights += self.weight_regularizer_l1 * dL1
        # L2 on weights
        if self.weight_regularizer_l2 > 0:
            self.dweights += 2 * self.weight_regularizer_l2 * \
                                self.weights
        # L1 on biases
        if self.bias_regularizer_l1 > 0:
            dL1 = np.ones_like(self.biases)
            dL1[self.biases < 0] = -1
            self.dbiases += self.bias_regularizer_l1 * dL1
        # L2 on biases
        if self.bias_regularizer_l2 > 0:
            self.dbiases += 2 * self.bias_regularizer_l2 * \
                               self.biases

        # Gradient on values
        self.dinputs = np.dot(dvalues, self.weights.T)
```

```python
# Dropout
class Layer_Dropout:

    # Init
    def __init__(self, rate):
        # Store rate, we invert it as for example for dropout
        # of 0.1 we need success rate of 0.9
        self.rate = 1 - rate

    # Forward pass
    def forward(self, inputs):
        # Save input values
        self.inputs = inputs
        # Generate and save scaled mask
        self.binary_mask = np.random.binomial(1, self.rate,
                           size=inputs.shape) / self.rate
        # Apply mask to output values
        self.output = inputs * self.binary_mask

    # Backward pass
    def backward(self, dvalues):
        # Gradient on values
        self.dinputs = dvalues * self.binary_mask


# ReLU activation
class Activation_ReLU:

    # Forward pass
    def forward(self, inputs):
        # Remember input values
        self.inputs = inputs
        # Calculate output values from inputs
        self.output = np.maximum(0, inputs)

    # Backward pass
    def backward(self, dvalues):
        # Since we need to modify original variable,
        # let's make a copy of values first
        self.dinputs = dvalues.copy()

        # Zero gradient where input values were negative
        self.dinputs[self.inputs <= 0] = 0
```

```
# Softmax activation
class Activation_Softmax:

    # Forward pass
    def forward(self, inputs):
        # Remember input values
        self.inputs = inputs

        # Get unnormalized probabilities
        exp_values = np.exp(inputs - np.max(inputs, axis=1,
                                            keepdims=True))
        # Normalize them for each sample
        probabilities = exp_values / np.sum(exp_values, axis=1,
                                            keepdims=True)

        self.output = probabilities


    # Backward pass
    def backward(self, dvalues):

        # Create uninitialized array
        self.dinputs = np.empty_like(dvalues)

        # Enumerate outputs and gradients
        for index, (single_output, single_dvalues) in \
                enumerate(zip(self.output, dvalues)):
            # Flatten output array
            single_output = single_output.reshape(-1, 1)
            # Calculate Jacobian matrix of the output
            jacobian_matrix = np.diagflat(single_output) - \
                              np.dot(single_output, single_output.T)
            # Calculate sample-wise gradient
            # and add it to the array of sample gradients
            self.dinputs[index] = np.dot(jacobian_matrix,
                                         single_dvalues)


# SGD optimizer
class Optimizer_SGD:

    # Initialize optimizer - set settings,
    # learning rate of 1. is default for this optimizer
    def __init__(self, learning_rate=1., decay=0., momentum=0.):
        self.learning_rate = learning_rate
        self.current_learning_rate = learning_rate
        self.decay = decay
        self.iterations = 0
        self.momentum = momentum
```

```
# Call once before any parameter updates
def pre_update_params(self):
    if self.decay:
        self.current_learning_rate = self.learning_rate * \
            (1. / (1. + self.decay * self.iterations))

# Update parameters
def update_params(self, layer):

    # If we use momentum
    if self.momentum:

        # If layer does not contain momentum arrays, create them
        # filled with zeros
        if not hasattr(layer, 'weight_momentums'):
            layer.weight_momentums = np.zeros_like(layer.weights)
            # If there is no momentum array for weights
            # The array doesn't exist for biases yet either.
            layer.bias_momentums = np.zeros_like(layer.biases)

        # Build weight updates with momentum - take previous
        # updates multiplied by retain factor and update with
        # current gradients
        weight_updates = \
            self.momentum * layer.weight_momentums - \
            self.current_learning_rate * layer.dweights
        layer.weight_momentums = weight_updates

        # Build bias updates
        bias_updates = \
            self.momentum * layer.bias_momentums - \
            self.current_learning_rate * layer.dbiases
        layer.bias_momentums = bias_updates

    # Vanilla SGD updates (as before momentum update)
    else:
        weight_updates = -self.current_learning_rate * \
                         layer.dweights
        bias_updates = -self.current_learning_rate * \
                       layer.dbiases

    # Update weights and biases using either
    # vanilla or momentum updates
    layer.weights += weight_updates
    layer.biases += bias_updates

# Call once after any parameter updates
def post_update_params(self):
    self.iterations += 1
```

```python
# Adagrad optimizer
class Optimizer_Adagrad:

    # Initialize optimizer - set settings
    def __init__(self, learning_rate=1., decay=0., epsilon=1e-7):
        self.learning_rate = learning_rate
        self.current_learning_rate = learning_rate
        self.decay = decay
        self.iterations = 0
        self.epsilon = epsilon

    # Call once before any parameter updates
    def pre_update_params(self):
        if self.decay:
            self.current_learning_rate = self.learning_rate * \
                (1. / (1. + self.decay * self.iterations))

    # Update parameters
    def update_params(self, layer):

        # If layer does not contain cache arrays,
        # create them filled with zeros
        if not hasattr(layer, 'weight_cache'):
            layer.weight_cache = np.zeros_like(layer.weights)
            layer.bias_cache = np.zeros_like(layer.biases)

        # Update cache with squared current gradients
        layer.weight_cache += layer.dweights**2
        layer.bias_cache += layer.dbiases**2

        # Vanilla SGD parameter update + normalization
        # with square rooted cache
        layer.weights += -self.current_learning_rate * \
                         layer.dweights / \
                         (np.sqrt(layer.weight_cache) + self.epsilon)
        layer.biases += -self.current_learning_rate * \
                        layer.dbiases / \
                        (np.sqrt(layer.bias_cache) + self.epsilon)

    # Call once after any parameter updates
    def post_update_params(self):
        self.iterations += 1
```

```python
# RMSprop optimizer
class Optimizer_RMSprop:

    # Initialize optimizer - set settings
    def __init__(self, learning_rate=0.001, decay=0., epsilon=1e-7,
                 rho=0.9):
        self.learning_rate = learning_rate
        self.current_learning_rate = learning_rate
        self.decay = decay
        self.iterations = 0
        self.epsilon = epsilon
        self.rho = rho

    # Call once before any parameter updates
    def pre_update_params(self):
        if self.decay:
            self.current_learning_rate = self.learning_rate * \
                (1. / (1. + self.decay * self.iterations))

    # Update parameters
    def update_params(self, layer):

        # If layer does not contain cache arrays,
        # create them filled with zeros
        if not hasattr(layer, 'weight_cache'):
            layer.weight_cache = np.zeros_like(layer.weights)
            layer.bias_cache = np.zeros_like(layer.biases)

        # Update cache with squared current gradients
        layer.weight_cache = self.rho * layer.weight_cache + \
            (1 - self.rho) * layer.dweights**2
        layer.bias_cache = self.rho * layer.bias_cache + \
            (1 - self.rho) * layer.dbiases**2

        # Vanilla SGD parameter update + normalization
        # with square rooted cache
        layer.weights += -self.current_learning_rate * \
                         layer.dweights / \
                         (np.sqrt(layer.weight_cache) + self.epsilon)
        layer.biases += -self.current_learning_rate * \
                        layer.dbiases / \
                        (np.sqrt(layer.bias_cache) + self.epsilon)

    # Call once after any parameter updates
    def post_update_params(self):
        self.iterations += 1
```

```
# Adam optimizer
class Optimizer_Adam:

    # Initialize optimizer - set settings
    def __init__(self, learning_rate=0.001, decay=0., epsilon=1e-7,
                 beta_1=0.9, beta_2=0.999):
        self.learning_rate = learning_rate
        self.current_learning_rate = learning_rate
        self.decay = decay
        self.iterations = 0
        self.epsilon = epsilon
        self.beta_1 = beta_1
        self.beta_2 = beta_2

    # Call once before any parameter updates
    def pre_update_params(self):
        if self.decay:
            self.current_learning_rate = self.learning_rate * \
                (1. / (1. + self.decay * self.iterations))

    # Update parameters
    def update_params(self, layer):

        # If layer does not contain cache arrays,
        # create them filled with zeros
        if not hasattr(layer, 'weight_cache'):
            layer.weight_momentums = np.zeros_like(layer.weights)
            layer.weight_cache = np.zeros_like(layer.weights)
            layer.bias_momentums = np.zeros_like(layer.biases)
            layer.bias_cache = np.zeros_like(layer.biases)

        # Update momentum  with current gradients
        layer.weight_momentums = self.beta_1 * \
                                 layer.weight_momentums + \
                                 (1 - self.beta_1) * layer.dweights
        layer.bias_momentums = self.beta_1 * \
                               layer.bias_momentums + \
                               (1 - self.beta_1) * layer.dbiases
        # Get corrected momentum
        # self.iteration is 0 at first pass
        # and we need to start with 1 here
        weight_momentums_corrected = layer.weight_momentums / \
            (1 - self.beta_1 ** (self.iterations + 1))
        bias_momentums_corrected = layer.bias_momentums / \
            (1 - self.beta_1 ** (self.iterations + 1))
        # Update cache with squared current gradients
        layer.weight_cache = self.beta_2 * layer.weight_cache + \
            (1 - self.beta_2) * layer.dweights**2
```

```
        layer.bias_cache = self.beta_2 * layer.bias_cache + \
            (1 - self.beta_2) * layer.dbiases**2
        # Get corrected cache
        weight_cache_corrected = layer.weight_cache / \
            (1 - self.beta_2 ** (self.iterations + 1))
        bias_cache_corrected = layer.bias_cache / \
            (1 - self.beta_2 ** (self.iterations + 1))

        # Vanilla SGD parameter update + normalization
        # with square rooted cache
        layer.weights += -self.current_learning_rate * \
                         weight_momentums_corrected / \
                         (np.sqrt(weight_cache_corrected) +
                             self.epsilon)
        layer.biases += -self.current_learning_rate * \
                         bias_momentums_corrected / \
                         (np.sqrt(bias_cache_corrected) +
                             self.epsilon)

    # Call once after any parameter updates
    def post_update_params(self):
        self.iterations += 1


# Common loss class
class Loss:

    # Regularization loss calculation
    def regularization_loss(self, layer):

        # 0 by default
        regularization_loss = 0

        # L1 regularization - weights
        # calculate only when factor greater than 0
        if layer.weight_regularizer_l1 > 0:
            regularization_loss += layer.weight_regularizer_l1 * \
                                   np.sum(np.abs(layer.weights))

        # L2 regularization - weights
        if layer.weight_regularizer_l2 > 0:
            regularization_loss += layer.weight_regularizer_l2 * \
                                   np.sum(layer.weights * \
                                          layer.weights)
```

```
        # L1 regularization - biases
        # calculate only when factor greater than 0
        if layer.bias_regularizer_l1 > 0:
            regularization_loss += layer.bias_regularizer_l1 * \
                                   np.sum(np.abs(layer.biases))

        # L2 regularization - biases
        if layer.bias_regularizer_l2 > 0:
            regularization_loss += layer.bias_regularizer_l2 * \
                                   np.sum(layer.biases * \
                                          layer.biases)

        return regularization_loss

    # Calculates the data and regularization losses
    # given model output and ground truth values
    def calculate(self, output, y):

        # Calculate sample losses
        sample_losses = self.forward(output, y)

        # Calculate mean loss
        data_loss = np.mean(sample_losses)

        # Return loss
        return data_loss


# Cross-entropy loss
class Loss_CategoricalCrossentropy(Loss):

    # Forward pass
    def forward(self, y_pred, y_true):

        # Number of samples in a batch
        samples = len(y_pred)

        # Clip data to prevent division by 0
        # Clip both sides to not drag mean towards any value
        y_pred_clipped = np.clip(y_pred, 1e-7, 1 - 1e-7)

        # Probabilities for target values -
        # only if categorical labels
        if len(y_true.shape) == 1:
            correct_confidences = y_pred_clipped[
                range(samples),
                y_true
            ]
```

```
        # Mask values - only for one-hot encoded labels
        elif len(y_true.shape) == 2:
            correct_confidences = np.sum(
                y_pred_clipped * y_true,
                axis=1
            )

        # Losses
        negative_log_likelihoods = -np.log(correct_confidences)
        return negative_log_likelihoods

    # Backward pass
    def backward(self, dvalues, y_true):

        # Number of samples
        samples = len(dvalues)
        # Number of labels in every sample
        # We'll use the first sample to count them
        labels = len(dvalues[0])

        # If labels are sparse, turn them into one-hot vector
        if len(y_true.shape) == 1:
            y_true = np.eye(labels)[y_true]

        # Calculate gradient
        self.dinputs = -y_true / dvalues
        # Normalize gradient
        self.dinputs = self.dinputs / samples


# Softmax classifier - combined Softmax activation
# and cross-entropy loss for faster backward step
class Activation_Softmax_Loss_CategoricalCrossentropy():

    # Creates activation and loss function objects
    def __init__(self):
        self.activation = Activation_Softmax()
        self.loss = Loss_CategoricalCrossentropy()

    # Forward pass
    def forward(self, inputs, y_true):
        # Output layer's activation function
        self.activation.forward(inputs)
        # Set the output
        self.output = self.activation.output
        # Calculate and return loss value
        return self.loss.calculate(self.output, y_true)
```

```
    # Backward pass
    def backward(self, dvalues, y_true):

        # Number of samples
        samples = len(dvalues)

        # If labels are one-hot encoded,
        # turn them into discrete values
        if len(y_true.shape) == 2:
            y_true = np.argmax(y_true, axis=1)

        # Copy so we can safely modify
        self.dinputs = dvalues.copy()
        # Calculate gradient
        self.dinputs[range(samples), y_true] -= 1
        # Normalize gradient
        self.dinputs = self.dinputs / samples


# Create dataset
X, y = spiral_data(samples=1000, classes=3)

# Create Dense layer with 2 input features and 64 output values
dense1 = Layer_Dense(2, 64, weight_regularizer_l2=5e-4,
                            bias_regularizer_l2=5e-4)

# Create ReLU activation (to be used with Dense layer):
activation1 = Activation_ReLU()

# Create dropout layer
dropout1 = Layer_Dropout(0.1)

# Create second Dense layer with 64 input features (as we take output
# of previous layer here) and 3 output values (output values)
dense2 = Layer_Dense(64, 3)

# Create Softmax classifier's combined loss and activation
loss_activation = Activation_Softmax_Loss_CategoricalCrossentropy()

# Create optimizer
optimizer = Optimizer_Adam(learning_rate=0.05, decay=5e-5)

# Train in loop
for epoch in range(10001):

    # Perform a forward pass of our training data through this layer
    dense1.forward(X)
```

```
# Perform a forward pass through activation function
# takes the output of first dense layer here
activation1.forward(dense1.output)

# Perform a forward pass through Dropout layer
dropout1.forward(activation1.output)
# Perform a forward pass through second Dense layer
# takes outputs of activation function of first layer as inputs
dense2.forward(dropout1.output)

# Perform a forward pass through the activation/loss function
# takes the output of second dense layer here and returns loss
data_loss = loss_activation.forward(dense2.output, y)

# Calculate regularization penalty
regularization_loss = \
    loss_activation.loss.regularization_loss(dense1) + \
    loss_activation.loss.regularization_loss(dense2)

# Calculate overall loss
loss = data_loss + regularization_loss

# Calculate accuracy from output of activation2 and targets
# calculate values along first axis
predictions = np.argmax(loss_activation.output, axis=1)
if len(y.shape) == 2:
    y = np.argmax(y, axis=1)
accuracy = np.mean(predictions == y)

if not epoch % 100:
    print(f'epoch: {epoch}, ' +
          f'acc: {accuracy:.3f}, ' +
          f'loss: {loss:.3f} (' +
          f'data_loss: {data_loss:.3f}, ' +
          f'reg_loss: {regularization_loss:.3f}), ' +
          f'lr: {optimizer.current_learning_rate}')

# Backward pass
loss_activation.backward(loss_activation.output, y)
dense2.backward(loss_activation.dinputs)
dropout1.backward(dense2.dinputs)
activation1.backward(dropout1.dinputs)
dense1.backward(activation1.dinputs)

# Update weights and biases
optimizer.pre_update_params()
optimizer.update_params(dense1)
optimizer.update_params(dense2)
optimizer.post_update_params()
```

```
# Validate the model

# Create test dataset
X_test, y_test = spiral_data(samples=100, classes=3)
# Perform a forward pass of our testing data through this layer
dense1.forward(X_test)

# Perform a forward pass through activation function
# takes the output of first dense layer here
activation1.forward(dense1.output)

# Perform a forward pass through second Dense layer
# takes outputs of activation function of first layer as inputs
dense2.forward(activation1.output)

# Perform a forward pass through the activation/loss function
# takes the output of second dense layer here and returns loss
loss = loss_activation.forward(dense2.output, y_test)

# Calculate accuracy from output of activation2 and targets
# calculate values along first axis
predictions = np.argmax(loss_activation.output, axis=1)
if len(y_test.shape) == 2:
    y_test = np.argmax(y_test, axis=1)
accuracy = np.mean(predictions == y_test)

print(f'validation, acc: {accuracy:.3f}, loss: {loss:.3f}')
```

```
>>>
epoch: 9900, acc: 0.668, loss: 0.733 (data_loss: 0.717, reg_loss: 0.016),
lr: 0.0334459346466437
epoch: 10000, acc: 0.688, loss: 0.727 (data_loss: 0.711, reg_loss: 0.016),
lr: 0.03333444448148271
validation, acc: 0.757, loss: 0.712
```

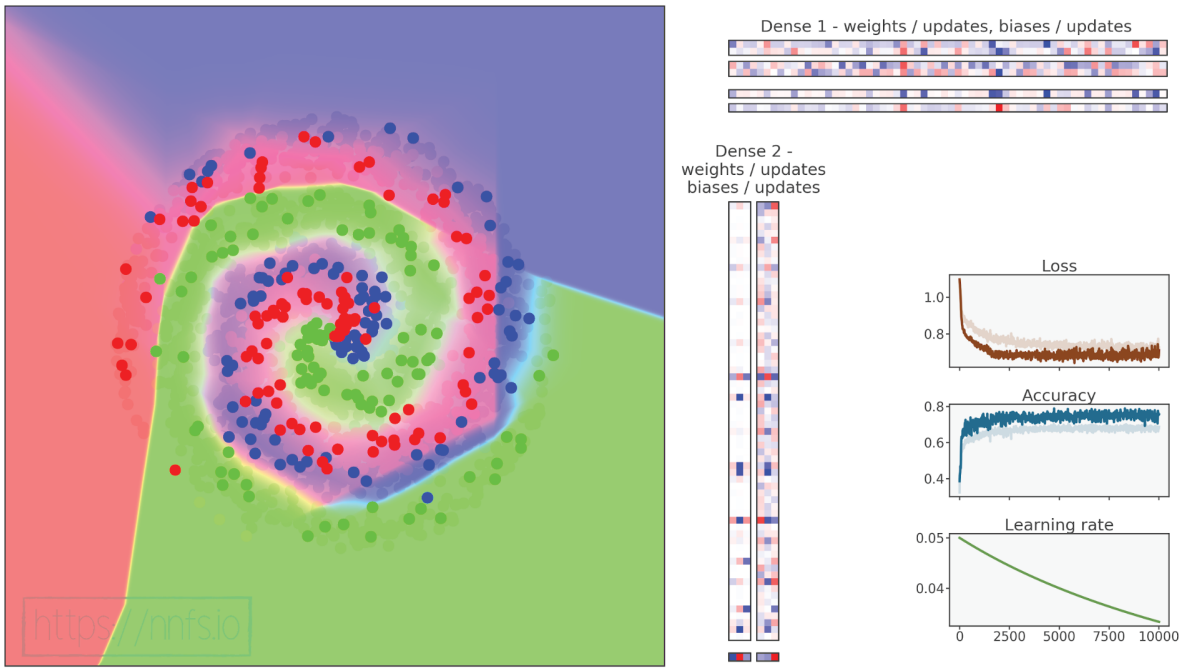


Fig 15.04: Model trained with dropout.

Anim 15.04: https://nnfs.io/efg

While our accuracy and loss have suffered considerably, we've found a scenario where our validation set performs *better* than our in-sample set (because we do not apply dropout when testing so you don't disable some of the connections). Further tweaking would likely fix the accuracy issue; for example, due to our regularization tactics, we can change our layer sizes to 512:

```
# Create Dense layer with 2 input features and 512 output values
dense1 = Layer_Dense(2, 512, weight_regularizer_l2=5e-4,
                             bias_regularizer_l2=5e-4)

# Create ReLU activation (to be used with Dense layer):
activation1 = Activation_ReLU()
```

```
# Create dropout layer
dropout1 = Layer_Dropout(0.1)

# Create second Dense layer with 512 input features
# and 3 output values
dense2 = Layer_Dense(512, 3)
```

Adding more neurons ends with:

```
epoch: 0, acc: 0.373, loss: 1.099 (data_loss: 1.099, reg_loss: 0.000), lr:
0.05
epoch: 100, acc: 0.719, loss: 0.735 (data_loss: 0.672, reg_loss: 0.063), lr:
0.04975371909050202
epoch: 200, acc: 0.782, loss: 0.627 (data_loss: 0.548, reg_loss: 0.079), lr:
0.049507401356502806
epoch: 300, acc: 0.800, loss: 0.603 (data_loss: 0.521, reg_loss: 0.082), lr:
0.0492635105177595
epoch: 400, acc: 0.802, loss: 0.595 (data_loss: 0.513, reg_loss: 0.082), lr:
0.04902201088288642
epoch: 500, acc: 0.809, loss: 0.562 (data_loss: 0.482, reg_loss: 0.079), lr:
0.048782867456949125
epoch: 600, acc: 0.836, loss: 0.521 (data_loss: 0.445, reg_loss: 0.076), lr:
0.04854604592455945
epoch: 700, acc: 0.816, loss: 0.532 (data_loss: 0.457, reg_loss: 0.076), lr:
0.048311512633460556
epoch: 800, acc: 0.839, loss: 0.515 (data_loss: 0.442, reg_loss: 0.073), lr:
0.04807923457858551
epoch: 900, acc: 0.842, loss: 0.499 (data_loss: 0.426, reg_loss: 0.072), lr:
0.04784917938657352
epoch: 1000, acc: 0.837, loss: 0.480 (data_loss: 0.408, reg_loss: 0.071),
lr: 0.04762131530072861
...
epoch: 9800, acc: 0.848, loss: 0.443 (data_loss: 0.391, reg_loss: 0.052),
lr: 0.033558173093056816
epoch: 9900, acc: 0.841, loss: 0.468 (data_loss: 0.416, reg_loss: 0.052),
lr: 0.0334459346466437
epoch: 10000, acc: 0.859, loss: 0.468 (data_loss: 0.417, reg_loss: 0.051),
lr: 0.03333444448148271
validation, acc: 0.857, loss: 0.397
```

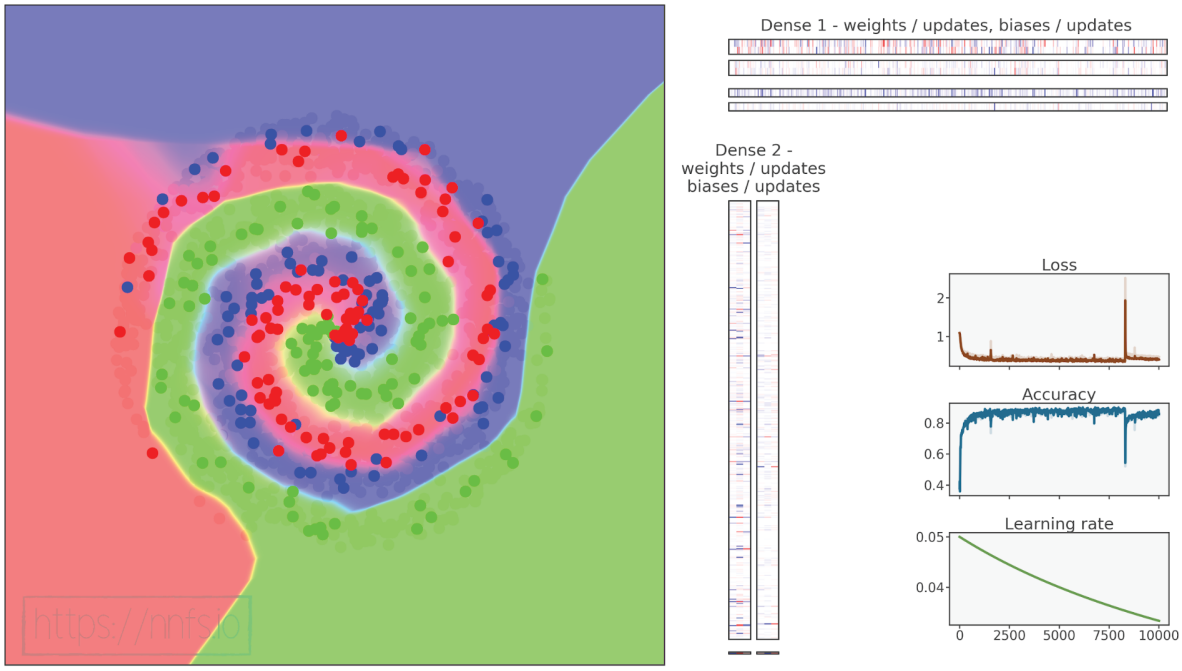


Fig 15.05: Model trained with dropout and bigger hidden layer.

Anim 15.05: https://nnfs.io/fgh

Pretty good result, but worse compared to the "no dropout" model. Interestingly, validation accuracy is close to the training accuracy with dropout — usually validation accuracy will be higher, so we might suspect these as signs of overfitting here (validation loss is lower than expected).

Supplementary Material: https://nnfs.io/ch15
Chapter code, further resources, and errata for this chapter.

Chapter 16

Binary Logistic Regression

Now that we've learned how to create and train a neural network, let's consider an alternative output layer for a neural network. Until now, we've used an output layer that is a probability distribution, where all of the values represent a confidence level of a given class being the correct class, and where these confidences sum to *1*. We're now going to cover an alternate output layer option, where each neuron separately represents two classes — 0 for one of the classes, and a 1 for the other. A model with this type of output layer is called **binary logistic regression**. This single neuron could represent two classes like *cat* vs. *dog*, but it could also represent *cat* vs. *not cat* or any combination of 2 classes, and you could have many of these. For example, a model may have two binary output neurons. One of these neurons could be distinguishing between *person/not person*, and the other neuron could be deciding between *indoors/outdoors*. Binary logistic regression is a regressor type of algorithm, which will differ as we'll use a **sigmoid** activation function for the output layer rather than **softmax**, and **binary cross-entropy** rather than **categorical cross-entropy** for calculating loss.

Sigmoid Activation Function

The sigmoid activation function is used with regressors because it "squishes" a range of outputs from negative infinity to positive infinity to be between 0 and 1. The bounds represent the two possible classes. The sigmoid equation is:

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

For the purpose of neural networks, we'll use our common notation:

$$\sigma_{i,j} = \frac{1}{1 + e^{-z_{i,j}}}$$

The denominator of the **Sigmoid** function contains *e* raised to the power of $z_{i,j}$, where *z*, given indices, means a singular output value of the layer that this activation function takes as input. The index *i* means the current sample, and the index *j* means the current output in this sample.

If we plot the sigmoid function:

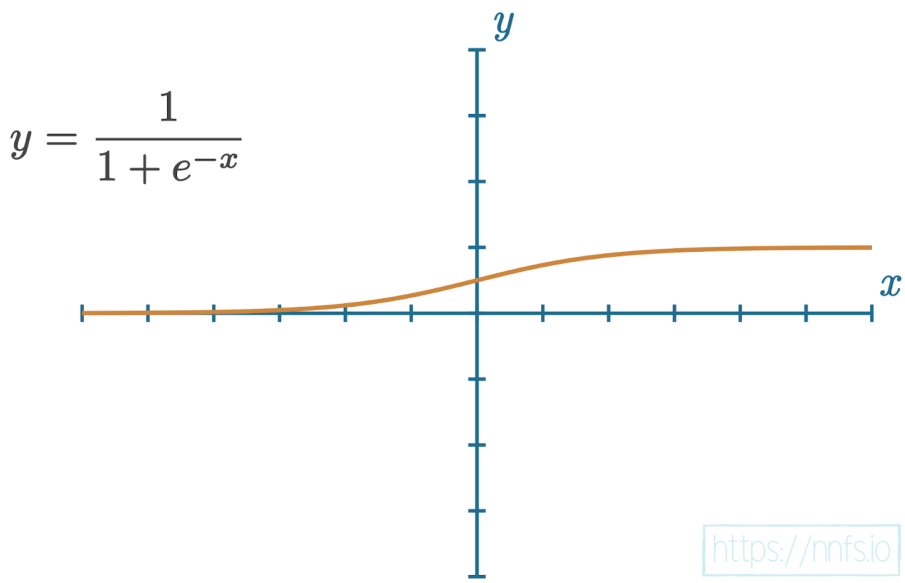


Fig 16.1: The sigmoid function graph.

Note the output from this function averages at *0.5*, and squishes down to a flat line as it approaches *0* or *1*. The sigmoid function approaches both maximum and minimum values exponentially fast. For example, for an input of *2*, the output is *~0.88*, which is already pretty close to *1*. With an input of *3*, the output is *~0.95*, and so on. It's also similar for negative values: $\sigma(-2) \approx 0.12$ and $\sigma(-3) \approx 0.05$. This property makes the sigmoid activation function a good candidate to apply to the final layer's output with a binary logistic regression model.

For commonly-used functions, such as the sigmoid function, the derivatives are almost always public knowledge. Unless you're inventing a function, you won't need to calculate derivatives by hand, but it can still be a good exercise. The sigmoid function's derivative solves to $\sigma_{i,j}(1-\sigma_{i,j})$. If you would like to leverage this fact without diving into the mathematical derivation, feel free to skip to the next section.

Sigmoid Function Derivative

Let's define the derivative of the **Sigmoid** function with respect to its input:

$$\sigma_{i,j} = \frac{1}{1+e^{-z_{i,j}}} \quad \rightarrow \quad \sigma'_{i,j} = \frac{d}{dz_{i,j}}[\frac{1}{1+e^{-z_{i,j}}}]$$

At this point, we might start calculating the derivative of the division operation, but, since the numerator contains just the value *1*, the whole fraction is effectively just a reciprocal of its denominator and can be represented as its negative power:

$$\frac{1}{x} = x^{-1}$$

It's easier to calculate the derivative of the power operation than the derivative of the division operation, so let's update our equation to follow this:

$$\frac{d}{dz_{i,j}}(1+e^{-z_{i,j}})^{-1} =$$

Now, we can calculate the derivative of the expression raised to the power of the *-1*, which equals this exponent multiplied by the expression itself, raised to the power lowered by *1*. Then, following the chain rule, we have to calculate the derivative of the expression itself:

$$= -1 \cdot (1+e^{-z_{i,j}})^{-1-1} \cdot \frac{d}{dz_{i,j}}(1+e^{-z_{i,j}}) =$$

As we already learned, the derivative of the sum operation is the sum of derivatives:

$$= -(1+e^{-z_{i,j}})^{-2} \cdot (\frac{d}{dz_{i,j}}1 + \frac{d}{dz_{i,j}}e^{-z_{i,j}}) =$$

The derivative of *1* with respect to $z_{i,j}$ equals *0*, as the derivative of a constant is always *0*. The

derivative of the constant *e* raised to the power $-z_{i,j}$ equals this value multiplied by the derivative of the exponent:

$$= -(1 + e^{-z_{i,j}})^{-2} \cdot (0 + e^{-z_{i,j}} \cdot \frac{d}{dz_{i,j}}[-z_{i,j}]) =$$

The derivative of the $-z_{i,j}$ with respect to $z_{i,j}$ equals *-1* as *-1* is a constant and can be moved outside of the derivative, leaving us with the derivative of $z_{i,j}$ with respect to $z_{i,j}$ which, as we know, equals *1*:

$$= -(1+e^{-z_{i,j}})^{-2} \cdot (e^{-z_{i,j}} \cdot (-1 \cdot \frac{d}{dz_{i,j}} z_{i,j})) = -(1+e^{-z_{i,j}})^{-2} \cdot (e^{-z_{i,j}} \cdot (-1)) =$$

Now we can move the minus sign outside of the parentheses and cancel out the other minus:

$$= -(1 + e^{-z_{i,j}})^{-2} \cdot (-e^{-z_{i,j}}) = (1 + e^{-z_{i,j}})^{-2} \cdot e^{-z_{i,j}} =$$

Let's rewrite the resulting equation — the expression raised to the power of *-2* can be written as its reciprocal raised to the power of *2*, then the multiplier (the value we multiply by) from the equation can become the numerator of the resulting fraction:

$$= \frac{e^{-z_{i,j}}}{(1 + e^{-z_{i,j}})^2} =$$

The denominator of this fraction can be written as the multiplication of the expression by itself instead of raising it to the power of *2*:

$$= \frac{e^{-z_{i,j}}}{(1 + e^{-z_{i,j}})(1 + e^{-z_{i,j}})} =$$

Now we can split this fraction into two separate ones — one containing *1* in the numerator and the other one *e* to the power of $-z_{i,j}$, both having each of the expressions that are separated by the multiplication operator in the denominator in their respective denominators. We can do this as we are performing the multiplication operation between both fractions:

$$= \frac{1}{1 + e^{-z_{i,j}}} \cdot \frac{e^{-z_{i,j}}}{1 + e^{-z_{i,j}}} =$$

If you remember the equation of the sigmoid function, you might already see where we are going with this — the multiplicand (the value that is being multiplied by the multiplier) is the equation of the sigmoid function. Let's work on this equation further — it'd be ideal if the numerator of

the multiplicator could be represented as some sort of equation containing the sigmoid function's equation as well. What we can do is add *1* and remove *1* from it as it won't change its value:

$$= \frac{1}{1+e^{-z_{i,j}}} \cdot \frac{1+e^{-z_{i,j}}-1}{1+e^{-z_{i,j}}} =$$

What this allows us to do is split the multiplicator into two separate fractions by the minus sign in the multiplicator:

$$= \frac{1}{1+e^{-z_{i,j}}} \cdot (\frac{1+e^{-z_{i,j}}}{1+e^{-z_{i,j}}} - \frac{1}{1+e^{-z_{i,j}}}) =$$

The minuend (the value we are subtracting from) of the multiplicator equals *1* as the numerator, and the denominator of the fraction are equal, and the subtrahend (the value we are subtracting from the minuend) is actually the equation of the sigmoid function as well:

$$= \frac{1}{1+e^{-z_{i,j}}} \cdot (1 - \frac{1}{1+e^{-z_{i,j}}}) = \sigma_{i,j} \cdot (1 - \sigma_{i,j})$$

It turns out that the derivative of the sigmoid function equals this function multiplied by the difference of *1* and this function as well. That allows us to easily write this derivative in the code.

Full solution:

$$\sigma_{i,j} = \frac{1}{1+e^{-z_{i,j}}} \quad \rightarrow \quad \sigma'_{i,j} = \frac{d}{dz_{i,j}}[\frac{1}{1+e^{-z_{i,j}}}] = \frac{d}{dz_{i,j}}(1+e^{-z_{i,j}})^{-1} =$$

$$= -1 \cdot (1+e^{-z_{i,j}})^{-1-1} \cdot \frac{d}{dz_{i,j}}(1+e^{-z_{i,j}}) = -(1+e^{-z_{i,j}})^{-2} \cdot (\frac{d}{dz_{i,j}}1 + \frac{d}{dz_{i,j}}e^{-z_{i,j}}) =$$

$$= -(1+e^{-z_{i,j}})^{-2} \cdot (0 + e^{-z_{i,j}} \cdot \frac{d}{dz_{i,j}}[-z_{i,j}]) =$$

$$= -(1+e^{-z_{i,j}})^{-2} \cdot (e^{-z_{i,j}} \cdot (-1 \cdot \frac{d}{dz_{i,j}}z_{i,j})) = -(1+e^{-z_{i,j}})^{-2} \cdot (e^{-z_{i,j}} \cdot (-1)) =$$

$$= -(1+e^{-z_{i,j}})^{-2} \cdot (-e^{-z_{i,j}}) = (1+e^{-z_{i,j}})^{-2} \cdot e^{-z_{i,j}} =$$

$$= \frac{e^{-z_{i,j}}}{(1+e^{-z_{i,j}})^2} = \frac{e^{-z_{i,j}}}{(1+e^{-z_{i,j}})(1+e^{-z_{i,j}})} = \frac{1}{1+e^{-z_{i,j}}} \cdot \frac{e^{-z_{i,j}}}{1+e^{-z_{i,j}}} =$$

$$= \frac{1}{1+e^{-z_{i,j}}} \cdot \frac{1+e^{-z_{i,j}}-1}{1+e^{-z_{i,j}}} == \frac{1}{1+e^{-z_{i,j}}} \cdot (\frac{1+e^{-z_{i,j}}}{1+e^{-z_{i,j}}} - \frac{1}{1+e^{-z_{i,j}}}) =$$

$$= \frac{1}{1+e^{-z_{i,j}}} \cdot (1 - \frac{1}{1+e^{-z_{i,j}}}) = \sigma_{i,j} \cdot (1 - \sigma_{i,j})$$

Sigmoid Function Code

As with other activation functions, we'll write a forward pass method and a backward pass method. For the forward pass, we'll take the inputs and apply the sigmoid function. For the backward pass, we'll leverage the sigmoid function's derivative, which, as we figured out during derivation of the sigmoid function's derivative, equals the sigmoid output from the forward pass multiplied by the difference of 1 and this output.

```
# Sigmoid activation
class Activation_Sigmoid:

    # Forward pass
    def forward(self, inputs):
        # Save input and calculate/save output
        # of the sigmoid function
        self.inputs = inputs
        self.output = 1 / (1 + np.exp(-inputs))

    # Backward pass
    def backward(self, dvalues):
        # Derivative - calculates from output of the sigmoid function
        self.dinputs = dvalues * (1 - self.output) * self.output
```

Now that we have the new activation function, we need to code our new calculation for the binary cross-entropy loss.

Binary Cross-Entropy Loss

To calculate binary cross-entropy loss, we will continue to use the negative log concept from categorical cross-entropy loss. Rather than only calculating this on the target class, we will sum the log-likelihoods of the correct and incorrect classes for each neuron separately. Because class values are either *0* or *1*, we can simplify the incorrect class to be *1-correct class* as this inverts the value. We can then calculate the negative log-likelihood of the correct and incorrect classes, adding them together. We are presenting two forms of the equation — the first is following the description just given, then the optimized version differentiating only in the minus signs being moved over and redundant parentheses removed:

$$L_{i,j} = (y_{i,j})(-log(\hat{y}_{i,j})) + (1 - y_{i,j})(-log(1 - \hat{y}_{i,j})) =$$

$$= -y_{i,j} \cdot log(\hat{y}_{i,j}) - (1 - y_{i,j}) \cdot log(1 - \hat{y}_{i,j})$$

In code, this will start as (but will be modified shortly, so do not commit this to your codebase yet):

```
sample_losses = -(y_true * np.log(y_pred) +
                  (1 - y_true) * np.log(1 - y_pred))
```

Since a model can contain multiple binary outputs, and each of them, unlike in the cross-entropy loss, outputs its own prediction, loss calculated on a single output is going to be a vector of losses containing one value for each output. What we need is a sample loss and, to achieve that, we need to calculate a mean of all of these losses from a single sample:

$$L_i = \frac{1}{J} \sum_j L_{i,j}$$

Where index *i* means the current sample, the index *j* means the current output in this sample, and the *J* means the number of outputs. Since we are operating on a set of samples (the output is an array containing the set of loss vectors), we can use NumPy to perform this operation on a single call:

```
sample_losses = np.mean(sample_losses, axis=-1)
```

The last parameter, `axis=-1`, informs NumPy to calculate the mean value along the last dimension. To make it easier to visualize, let's use a simple example. Assume that this is an output of the model containing 3 neurons in the output layer, and it's passed through the binary cross-entropy loss function:

```
outputs = np.array([[1,  2,  3],
                    [2,  4,  6],
                    [0,  5,  10],
                    [11, 12, 13],
                    [5,  10, 15]])
```

These numbers are completely made up for this example. We want to take each of the output vectors, `[1,  2,  3]` for example, and calculate a mean value from the numbers they hold, putting the result on the output vector. We then want to repeat this for the other vectors and return the resulting vector, which will be a one-dimensional array. Using NumPy:

```
print(np.mean(outputs, axis=-1))


>>>
array([ 2.,  4.,  5., 12., 10.])
```

If we calculate the mean value of the first output, it's indeed *2*, the mean value of the second output is indeed *4*, and so on.

We are also going to inherit from the **Loss** class, so the overall loss calculation will be handled by the `calculate` method that we already created for the categorical cross-entropy loss class.

Binary Cross-Entropy Loss Derivative

To calculate the gradient from here, we already know that the derivative for the natural logarithm is *1/x* and that the derivative of *1-x* is *-1*. In simplified form, this gives us *-(y_true / y + (1 - y_true) / (1 - y)) · (-1)*.

To calculate the partial derivative of this loss function with respect to the predicted input, we'll use the latter version of the loss equation. It doesn't really matter in this case which one we use:

$$\frac{\partial L_{i,j}}{\partial \hat{y}_{i,j}} = \frac{\partial}{\partial \hat{y}_{i,j}}[-y_{i,j} \cdot log(\hat{y}_{i,j}) - (1 - y_{i,j}) \cdot log(1 - \hat{y}_{i,j})] =$$

The expression that we have to calculate the partial derivative of consists of two sub-expressions, which are components of the sum operation. We can write that as the sum of derivatives:

$$= \frac{\partial}{\partial \hat{y}_{i,j}}[-y_{i,j} \cdot log(\hat{y}_{i,j})] + \frac{\partial}{\partial \hat{y}_{i,j}}[-(1 - y_{i,j}) \cdot log(1 - \hat{y}_{i,j})] =$$

Both components contain $y_{i,j}$ (the target value) inside of their derivatives, which are the constants that we are deriving with respect to $y\text{-}hat_{i,j}$ (the predicted value, which is a different variable), so we can move them outside of the derivative along with the other constants and minus sign:

$$= -y_{i,j} \cdot \frac{\partial}{\partial \hat{y}_{i,j}} log(\hat{y}_{i,j}) - (1 - y_{i,j}) \cdot \frac{\partial}{\partial \hat{y}_{i,j}} log(1 - \hat{y}_{i,j}) =$$

Now, like in the **Categorical Cross-Entropy** loss' derivative, we have to calculate the derivative of the logarithmic function, which equals the reciprocal of its parameter multiplied (following the chain rule) by the derivative of this parameter. Let's apply that to both of the partial derivatives:

$$= -y_{i,j} \cdot \frac{1}{\hat{y}_{i,j}} \cdot \frac{\partial}{\partial \hat{y}_{i,j}} \hat{y}_{i,j} - (1 - y_{i,j}) \cdot \frac{1}{1 - \hat{y}_{i,j}} \cdot \frac{\partial}{\partial \hat{y}_{i,j}}[1 - \hat{y}_{i,j}] =$$

Now the first partial derivative equals *1*, since the value we derive, and the value we derive with respect to, are the same values. The second partial derivative can be written as the difference of

the derivatives:

$$= -y_{i,j} \cdot \frac{1}{\hat{y}_{i,j}} \cdot 1 - (1 - y_{i,j}) \cdot \frac{1}{1 - \hat{y}_{i,j}} \cdot (\frac{\partial}{\partial \hat{y}_{i,j}} 1 - \frac{\partial}{\partial \hat{y}_{i,j}} \hat{y}_{i,j}) =$$

From the two new derivatives, the first one equals *0* as the derivative of the constant always equals *0*, then the second derivative equals *1* as the value we derive, and the value we derive with respect to, are the same values:

$$= -y_{i,j} \cdot \frac{1}{\hat{y}_{i,j}} \cdot 1 - (1 - y_{i,j}) \cdot \frac{1}{1 - \hat{y}_{i,j}} \cdot (0 - 1) =$$

We can finally clean up to get the resulting equation:

$$= -\frac{y_{i,j}}{\hat{y}_{i,j}} + \frac{1 - y_{i,j}}{1 - \hat{y}_{i,j}} = -(\frac{y_{i,j}}{\hat{y}_{i,j}} - \frac{1 - y_{i,j}}{1 - \hat{y}_{i,j}})$$

The partial derivative of the **Binary Cross-Entropy** loss solves to a pretty simple equation that will be easy to implement in code.

Full solution:

$$\frac{\partial L_{i,j}}{\partial \hat{y}_{i,j}} = \frac{\partial}{\partial \hat{y}_{i,j}} [-y_{i,j} \cdot log(\hat{y}_{i,j}) - (1 - y_{i,j}) \cdot log(1 - \hat{y}_{i,j})] =$$

$$= \frac{\partial}{\partial \hat{y}_{i,j}} [-y_{i,j} \cdot log(\hat{y}_{i,j})] + \frac{\partial}{\partial \hat{y}_{i,j}} [-(1 - y_{i,j}) \cdot log(1 - \hat{y}_{i,j})] =$$

$$= -y_{i,j} \cdot \frac{\partial}{\partial \hat{y}_{i,j}} log(\hat{y}_{i,j}) - (1 - y_{i,j}) \cdot \frac{\partial}{\partial \hat{y}_{i,j}} log(1 - \hat{y}_{i,j}) =$$

$$= -y_{i,j} \cdot \frac{1}{\hat{y}_{i,j}} \cdot \frac{\partial}{\partial \hat{y}_{i,j}} \hat{y}_{i,j} - (1 - y_{i,j}) \cdot \frac{1}{1 - \hat{y}_{i,j}} \cdot \frac{\partial}{\partial \hat{y}_{i,j}} [1 - \hat{y}_{i,j}] =$$

$$= -y_{i,j} \cdot \frac{1}{\hat{y}_{i,j}} \cdot 1 - (1 - y_{i,j}) \cdot \frac{1}{1 - \hat{y}_{i,j}} \cdot (\frac{\partial}{\partial \hat{y}_{i,j}} 1 - \frac{\partial}{\partial \hat{y}_{i,j}} \hat{y}_{i,j}) =$$

$$= -y_{i,j} \cdot \frac{1}{\hat{y}_{i,j}} \cdot 1 - (1 - y_{i,j}) \cdot \frac{1}{1 - \hat{y}_{i,j}} \cdot (0 - 1) =$$

$$= -\frac{y_{i,j}}{\hat{y}_{i,j}} + \frac{1 - y_{i,j}}{1 - \hat{y}_{i,j}} = -(\frac{y_{i,j}}{\hat{y}_{i,j}} - \frac{1 - y_{i,j}}{1 - \hat{y}_{i,j}})$$

This partial derivative is a derivative of the single output's loss and, with any type of output, we always need to calculate it with respect to a sample loss, not an atomic output loss, since we have to calculate the mean value of all output losses in a sample to form a *sample loss* during the forward pass:

$$L_i = \frac{1}{J}\sum_j L_{i,j}$$

For backpropagation, we have to calculate the partial derivative of the *sample loss* with respect to each input:

$$\frac{\partial L_i}{\partial \hat{y}_{i,j}} = \frac{\partial L_i}{\partial L_{i,j}} \cdot \frac{\partial L_{ij}}{\partial \hat{y}_{i,j}}$$

We have just calculated the second derivative, the partial derivative of the single output loss with respect to the related prediction. We have to calculate the partial derivative of the sample loss with respect to the single output loss:

$$\frac{\partial L_i}{\partial L_{i,j}} = \frac{\partial}{\partial L_{i,j}}[\frac{1}{J}\sum_j L_{i,j}] =$$

1 divided by *J* (the number of outputs), is a constant and can be moved outside of the derivative. Since we are calculating the derivative with respect to a given output, *j*, the sum of one element equals this element:

$$= \frac{1}{J} \cdot \frac{\partial}{\partial L_{i,j}} L_{i,j} =$$

The remaining derivative equals *1* as the derivative of a variable with respect to the same variable equals *1*.

$$= \frac{1}{J} \cdot 1 = \frac{1}{J}$$

Full solution:

$$\frac{\partial L_i}{\partial L_{i,j}} = \frac{\partial}{\partial L_{i,j}}[\frac{1}{J}\sum_j L_{i,j}] = \frac{1}{J} \cdot \frac{\partial}{\partial L_{i,j}} L_{i,j} = \frac{1}{J} \cdot 1 = \frac{1}{J}$$

Now we can update the equation of the partial derivative of a sample loss with respect to a single output loss by applying the chain rule:

$$\frac{\partial L_i}{\partial \hat{y}_{i,j}} = \frac{\partial L_i}{\partial L_{i,j}} \cdot \frac{\partial L_{ij}}{\partial \hat{y}_{i,j}} = \frac{1}{J} \cdot (-(\frac{y_{i,j}}{\hat{y}_{i,j}} - \frac{1 - y_{i,j}}{1 - \hat{y}_{i,j}})) = -\frac{1}{J} \cdot (\frac{y_{i,j}}{\hat{y}_{i,j}} - \frac{1 - y_{i,j}}{1 - \hat{y}_{i,j}}) =$$

We have to perform this normalization since each output returns its own derivative, and without normalization, each additional input will raise gradients and require changing other hyperparameters, including the learning rate.

Binary Cross-Entropy Code

In our code, this will be:

```
# Number of samples
samples = len(dvalues)
# Number of outputs in every sample
# We'll use the first sample to count them
outputs = len(dvalues[0])

# Calculate gradient
self.dinputs = -(y_true / clipped_dvalues -
                 (1 - y_true) / (1 - clipped_dvalues)) / outputs
```

Similar to what we did in the categorical cross-entropy loss, we need to normalize gradient so it'll become invariant to the number of samples we calculate it for:

```
# Normalize gradient
self.dinputs = self.dinputs / samples
```

Finally, we need to address the numerical instability of the logarithmic function. The sigmoid activation can return a value in the range of *0* to *1* (inclusive), but the *log(0)* presents a slight issue due to how it's calculated and will return *negative infinity*. This alone isn't necessarily a big deal, but any list with *-inf* in it will have a mean of *-inf*, which is the same for any list with positive infinity averaging to infinity.

```
import numpy as np

print(np.log(0))


>>>
__main__:1: RuntimeWarning: divide by zero encountered in log
-inf


print(np.mean([5, 2, 4, np.log(0)]))


>>>
__main__:1: RuntimeWarning: divide by zero encountered in log
-inf
```

This is a similar issue to the one we discussed earlier regarding categorical cross-entropy loss in chapter 5. To prevent this issue, we'll add clipping on the batch of values:

```
# Clip data to prevent division by 0
# Clip both sides to not drag mean towards any value
y_pred_clipped = np.clip(y_pred, 1e-7, 1 - 1e-7)
```

We now will use these clipped values for the forward pass, rather than the originals:

```
# Calculate sample-wise loss
sample_losses = -(y_true * np.log(y_pred_clipped) +
                  (1 - y_true) * np.log(1 - y_pred_clipped))
```

As we perform the division operation during the derivative calculation, the gradient passed in may contain both values, *0* and *1*. Either of these values will cause a problem in either the `y_true / dvalues` or `(1 - y_true) / (1 - dvalues)` parts respectively (*0* in the first and *1-1=0* in the second case will also cause division by *0*), so we need to clip this gradient as well:

```
# Clip data to prevent division by 0
# Clip both sides to not drag mean towards any value
clipped_dvalues = np.clip(dvalues, 1e-7, 1 - 1e-7)
```

Now, similar to the forward pass, we can use these clipped values:

```
# Calculate gradient
self.dinputs = -(y_true / clipped_dvalues -
                 (1 - y_true) / (1 - clipped_dvalues)) / outputs
```

The full code now for the binary cross-entropy:

```
# Binary cross-entropy loss
class Loss_BinaryCrossentropy(Loss):

    # Forward pass
    def forward(self, y_pred, y_true):

        # Clip data to prevent division by 0
        # Clip both sides to not drag mean towards any value
        y_pred_clipped = np.clip(y_pred, 1e-7, 1 - 1e-7)

        # Calculate sample-wise loss
        sample_losses = -(y_true * np.log(y_pred_clipped) +
                          (1 - y_true) * np.log(1 - y_pred_clipped))
        sample_losses = np.mean(sample_losses, axis=-1)

        # Return losses
        return sample_losses

    # Backward pass
    def backward(self, dvalues, y_true):

        # Number of samples
        samples = len(dvalues)
        # Number of outputs in every sample
        # We'll use the first sample to count them
        outputs = len(dvalues[0])

        # Clip data to prevent division by 0
        # Clip both sides to not drag mean towards any value
        clipped_dvalues = np.clip(dvalues, 1e-7, 1 - 1e-7)

        # Calculate gradient
        self.dinputs = -(y_true / clipped_dvalues -
                         (1 - y_true) / (1 - clipped_dvalues)) / outputs
        # Normalize gradient
        self.dinputs = self.dinputs / samples
```

Now that we have this new activation function and loss calculation, we'll make edits to our existing softmax classifier to implement the binary logistic regression model.

Implementing Binary Logistic Regression and Binary Cross-Entropy Loss

With these new classes, our code changes will be in the execution of actual code (instead of modifying the classes). The first change is to make the `spiral_data` object output 2 classes, rather than 3, like so:

```
# Create dataset
X, y = spiral_data(samples=100, classes=2)
```

Next, we'll reshape our labels, as they're not sparse anymore. They're binary, *0* or *1*:

```
# Reshape labels to be a list of lists
# Inner list contains one output (either 0 or 1)
# per each output neuron, 1 in this case
y = y.reshape(-1, 1)
```

Consider the difference here. Initially, the *y* output from the `spiral_data` function would look something like:

```
X, y = spiral_data(samples=100, classes=2)
print(y[:5])
>>>
[0 0 0 0 0]
```

Then we reshape it here for binary logistic regression:

```
y = y.reshape(-1, 1)
print(y[:5])
>>>
[[0]
 [0]
 [0]
 [0]
 [0]]
```

Why have we done this? Initially, with the softmax classifier, the values from `spiral_data` could be used directly as the target labels, as they contain the correct class labels in numerical form — an index of the correct class, where each neuron in the output layer is a separate class, for example `[0, 1, 1, 0, 1]`. In this case, however, we're trying to represent some binary outputs, where each neuron represents 2 possible classes on its own. For the example we're currently working on, we have a single output neuron so the output from our neural network should be a tensor (array), containing one value, of a target value of either *0* or *1*, for example, `[[0], [1], [1], [0], [1]]`. The `.reshape(-1, 1)` means to reshape the data into 2 dimensions, where the second dimension contains a single element, and the first dimension contains how many elements the result will contain (`-1`) following other conditions. You are allowed to use `-1` only once in a shape with NumPy, letting you have that dimension be variable. Thanks to this ability, we do not always need the same number of samples every time, and NumPy can handle the calculation for us. In the case above, they're all *0* because the `spiral_data` function makes the dataset one class at a time, starting with *0*. We will also need to reshape the y-testing data in the same way.

Let's create our layers and use the appropriate activation functions:

```
# Create dataset
X, y = spiral_data(100, 2)

# Reshape labels to be a list of lists
# Inner list contains one output (either 0 or 1)
# per each output neuron, 1 in this case
y = y.reshape(-1, 1)

# Create Dense layer with 2 input features and 64 output values
dense1 = Layer_Dense(2, 64, weight_regularizer_l2=5e-4,
                            bias_regularizer_l2=5e-4)

# Create ReLU activation (to be used with Dense layer):
activation1 = Activation_ReLU()

# Create second Dense layer with 64 input features (as we take output
# of previous layer here) and 1 output value
dense2 = Layer_Dense(64, 1)

# Create Sigmoid activation:
activation2 = Activation_Sigmoid()
```

Notice that we're still using the rectified linear activation for the hidden layer. The hidden layer activation functions don't necessarily need to change, even though we're effectively building a different type of classifier. You should also notice that because this is now a binary classifier, the `dense2` object has only 1 output. Its output represents exactly 2 classes (*0* or *1*) being mapped to one neuron. We can now select a loss function and optimizer. For the **Adam** optimizer settings, we

are going to use the default learning rate and the decaying of *5e-7*:

```
# Create loss function
loss_function = Loss_BinaryCrossentropy()

# Create optimizer
optimizer = Optimizer_Adam(decay=5e-7)
```

While we require a different calculation for loss (since we use a different activation function for the output layer), we can still use the same optimizer as in the softmax classifier. Another small change is how we measure predictions. With probability distributions, we use *argmax* and determine which index is associated with the largest value, which becomes the classification result. With a binary classifier, we are determining if the output is closer to *0* or to *1*. To do this, we simplify the output to:

```
predictions = (activation2.output > 0.5) * 1
```

This results in *True*/*False* evaluations to the statement that the output is above *0.5* for all values. *True* and *False*, when treated as numbers, are *1* and *0*, respectively. For example, if we execute: `int(True)`, the result will be `1` and `int(False)` will be `0`. If we want to convert a list of *True*/*False* boolean values to numbers, we can't just wrap the list in `int()`. However, we *can* perform math operations directly on an array of boolean values and return the arithmetic answer. For example, we can run:

```
import numpy as np

a = np.array([True, False, True])
print(a)

>>>
[ True False  True]
```

And then:

```
b = a*1
print(b)

>>>
[1 0 1]
```

Thus, to evaluate predictive accuracy, we can do the following in our code:

```
predictions = (activation2.output > 0.5) * 1
accuracy = np.mean(predictions==y_test)
```

The `* 1` multiplication turns an array of boolean True/False values into numerical 1/0 values, respectively. We will need to implement this accuracy calculation for validation data too.

Full code up to this point:

```
import numpy as np
import nnfs
from nnfs.datasets import spiral_data

nnfs.init()


# Dense layer
class Layer_Dense:

    # Layer initialization
    def __init__(self, n_inputs, n_neurons,
                 weight_regularizer_l1=0, weight_regularizer_l2=0,
                 bias_regularizer_l1=0, bias_regularizer_l2=0):
        # Initialize weights and biases
        self.weights = 0.01 * np.random.randn(n_inputs, n_neurons)
        self.biases = np.zeros((1, n_neurons))
        # Set regularization strength
        self.weight_regularizer_l1 = weight_regularizer_l1
        self.weight_regularizer_l2 = weight_regularizer_l2
        self.bias_regularizer_l1 = bias_regularizer_l1
        self.bias_regularizer_l2 = bias_regularizer_l2

    # Forward pass
    def forward(self, inputs):
        # Remember input values
        self.inputs = inputs
        # Calculate output values from inputs, weights and biases
        self.output = np.dot(inputs, self.weights) + self.biases

    # Backward pass
    def backward(self, dvalues):
        # Gradients on parameters
        self.dweights = np.dot(self.inputs.T, dvalues)
        self.dbiases = np.sum(dvalues, axis=0, keepdims=True)
```

```
        # Gradients on regularization
        # L1 on weights
        if self.weight_regularizer_l1 > 0:
            dL1 = np.ones_like(self.weights)
            dL1[self.weights < 0] = -1
            self.dweights += self.weight_regularizer_l1 * dL1
        # L2 on weights
        if self.weight_regularizer_l2 > 0:
            self.dweights += 2 * self.weight_regularizer_l2 * \
                             self.weights
        # L1 on biases
        if self.bias_regularizer_l1 > 0:
            dL1 = np.ones_like(self.biases)
            dL1[self.biases < 0] = -1
            self.dbiases += self.bias_regularizer_l1 * dL1
        # L2 on biases
        if self.bias_regularizer_l2 > 0:
            self.dbiases += 2 * self.bias_regularizer_l2 * \
                            self.biases

        # Gradient on values
        self.dinputs = np.dot(dvalues, self.weights.T)


# Dropout
class Layer_Dropout:

    # Init
    def __init__(self, rate):
        # Store rate, we invert it as for example for dropout
        # of 0.1 we need success rate of 0.9
        self.rate = 1 - rate

    # Forward pass
    def forward(self, inputs):
        # Save input values
        self.inputs = inputs
        # Generate and save scaled mask
        self.binary_mask = np.random.binomial(1, self.rate,
                           size=inputs.shape) / self.rate
        # Apply mask to output values
        self.output = inputs * self.binary_mask

    # Backward pass
    def backward(self, dvalues):
        # Gradient on values
        self.dinputs = dvalues * self.binary_mask
```

```python
# ReLU activation
class Activation_ReLU:

    # Forward pass
    def forward(self, inputs):
        # Remember input values
        self.inputs = inputs
        # Calculate output values from inputs
        self.output = np.maximum(0, inputs)

    # Backward pass
    def backward(self, dvalues):
        # Since we need to modify original variable,
        # let's make a copy of values first
        self.dinputs = dvalues.copy()

        # Zero gradient where input values were negative
        self.dinputs[self.inputs <= 0] = 0


# Softmax activation
class Activation_Softmax:

    # Forward pass
    def forward(self, inputs):
        # Remember input values
        self.inputs = inputs

        # Get unnormalized probabilities
        exp_values = np.exp(inputs - np.max(inputs, axis=1,
                                            keepdims=True))
        # Normalize them for each sample
        probabilities = exp_values / np.sum(exp_values, axis=1,
                                            keepdims=True)

        self.output = probabilities

    # Backward pass
    def backward(self, dvalues):

        # Create uninitialized array
        self.dinputs = np.empty_like(dvalues)

        # Enumerate outputs and gradients
        for index, (single_output, single_dvalues) in \
                enumerate(zip(self.output, dvalues)):
            # Flatten output array
            single_output = single_output.reshape(-1, 1)
```

```
            # Calculate Jacobian matrix of the output
            jacobian_matrix = np.diagflat(single_output) - \
                              np.dot(single_output, single_output.T)
            # Calculate sample-wise gradient
            # and add it to the array of sample gradients
            self.dinputs[index] = np.dot(jacobian_matrix,
                                         single_dvalues)


# Sigmoid activation
class Activation_Sigmoid:

    # Forward pass
    def forward(self, inputs):
        # Save input and calculate/save output
        # of the sigmoid function
        self.inputs = inputs
        self.output = 1 / (1 + np.exp(-inputs))

    # Backward pass
    def backward(self, dvalues):
        # Derivative - calculates from output of the sigmoid function
        self.dinputs = dvalues * (1 - self.output) * self.output


# SGD optimizer
class Optimizer_SGD:

    # Initialize optimizer - set settings,
    # learning rate of 1. is default for this optimizer
    def __init__(self, learning_rate=1., decay=0., momentum=0.):
        self.learning_rate = learning_rate
        self.current_learning_rate = learning_rate
        self.decay = decay
        self.iterations = 0
        self.momentum = momentum

    # Call once before any parameter updates
    def pre_update_params(self):
        if self.decay:
            self.current_learning_rate = self.learning_rate * \
                (1. / (1. + self.decay * self.iterations))
```

```
# Update parameters
def update_params(self, layer):

    # If we use momentum
    if self.momentum:

        # If layer does not contain momentum arrays, create them
        # filled with zeros
        if not hasattr(layer, 'weight_momentums'):
            layer.weight_momentums = np.zeros_like(layer.weights)
            # If there is no momentum array for weights
            # The array doesn't exist for biases yet either.
            layer.bias_momentums = np.zeros_like(layer.biases)

        # Build weight updates with momentum - take previous
        # updates multiplied by retain factor and update with
        # current gradients
        weight_updates = \
            self.momentum * layer.weight_momentums - \
            self.current_learning_rate * layer.dweights
        layer.weight_momentums = weight_updates

        # Build bias updates
        bias_updates = \
            self.momentum * layer.bias_momentums - \
            self.current_learning_rate * layer.dbiases
        layer.bias_momentums = bias_updates

    # Vanilla SGD updates (as before momentum update)
    else:
        weight_updates = -self.current_learning_rate * \
                         layer.dweights
        bias_updates = -self.current_learning_rate * \
                       layer.dbiases

    # Update weights and biases using either
    # vanilla or momentum updates
    layer.weights += weight_updates
    layer.biases += bias_updates

# Call once after any parameter updates
def post_update_params(self):
    self.iterations += 1
```

```python
# Adagrad optimizer
class Optimizer_Adagrad:

    # Initialize optimizer - set settings
    def __init__(self, learning_rate=1., decay=0., epsilon=1e-7):
        self.learning_rate = learning_rate
        self.current_learning_rate = learning_rate
        self.decay = decay
        self.iterations = 0
        self.epsilon = epsilon

    # Call once before any parameter updates
    def pre_update_params(self):
        if self.decay:
            self.current_learning_rate = self.learning_rate * \
                (1. / (1. + self.decay * self.iterations))

    # Update parameters
    def update_params(self, layer):

        # If layer does not contain cache arrays,
        # create them filled with zeros
        if not hasattr(layer, 'weight_cache'):
            layer.weight_cache = np.zeros_like(layer.weights)
            layer.bias_cache = np.zeros_like(layer.biases)

        # Update cache with squared current gradients
        layer.weight_cache += layer.dweights**2
        layer.bias_cache += layer.dbiases**2

        # Vanilla SGD parameter update + normalization
        # with square rooted cache
        layer.weights += -self.current_learning_rate * \
                         layer.dweights / \
                         (np.sqrt(layer.weight_cache) + self.epsilon)
        layer.biases += -self.current_learning_rate * \
                        layer.dbiases / \
                        (np.sqrt(layer.bias_cache) + self.epsilon)

    # Call once after any parameter updates
    def post_update_params(self):
        self.iterations += 1
```

```python
# RMSprop optimizer
class Optimizer_RMSprop:

    # Initialize optimizer - set settings
    def __init__(self, learning_rate=0.001, decay=0., epsilon=1e-7,
                 rho=0.9):
        self.learning_rate = learning_rate
        self.current_learning_rate = learning_rate
        self.decay = decay
        self.iterations = 0
        self.epsilon = epsilon
        self.rho = rho

    # Call once before any parameter updates
    def pre_update_params(self):
        if self.decay:
            self.current_learning_rate = self.learning_rate * \
                (1. / (1. + self.decay * self.iterations))

    # Update parameters
    def update_params(self, layer):

        # If layer does not contain cache arrays,
        # create them filled with zeros
        if not hasattr(layer, 'weight_cache'):
            layer.weight_cache = np.zeros_like(layer.weights)
            layer.bias_cache = np.zeros_like(layer.biases)

        # Update cache with squared current gradients
        layer.weight_cache = self.rho * layer.weight_cache + \
            (1 - self.rho) * layer.dweights**2
        layer.bias_cache = self.rho * layer.bias_cache + \
            (1 - self.rho) * layer.dbiases**2

        # Vanilla SGD parameter update + normalization
        # with square rooted cache
        layer.weights += -self.current_learning_rate * \
                         layer.dweights / \
                         (np.sqrt(layer.weight_cache) + self.epsilon)
        layer.biases += -self.current_learning_rate * \
                        layer.dbiases / \
                        (np.sqrt(layer.bias_cache) + self.epsilon)

    # Call once after any parameter updates
    def post_update_params(self):
        self.iterations += 1
```

```
# Adam optimizer
class Optimizer_Adam:

    # Initialize optimizer - set settings
    def __init__(self, learning_rate=0.001, decay=0., epsilon=1e-7,
                 beta_1=0.9, beta_2=0.999):
        self.learning_rate = learning_rate
        self.current_learning_rate = learning_rate
        self.decay = decay
        self.iterations = 0
        self.epsilon = epsilon
        self.beta_1 = beta_1
        self.beta_2 = beta_2

    # Call once before any parameter updates
    def pre_update_params(self):
        if self.decay:
            self.current_learning_rate = self.learning_rate * \
                (1. / (1. + self.decay * self.iterations))

    # Update parameters
    def update_params(self, layer):

        # If layer does not contain cache arrays,
        # create them filled with zeros
        if not hasattr(layer, 'weight_cache'):
            layer.weight_momentums = np.zeros_like(layer.weights)
            layer.weight_cache = np.zeros_like(layer.weights)
            layer.bias_momentums = np.zeros_like(layer.biases)
            layer.bias_cache = np.zeros_like(layer.biases)
        # Update momentum  with current gradients
        layer.weight_momentums = self.beta_1 * \
                                 layer.weight_momentums + \
                                 (1 - self.beta_1) * layer.dweights
        layer.bias_momentums = self.beta_1 * \
                               layer.bias_momentums + \
                               (1 - self.beta_1) * layer.dbiases
        # Get corrected momentum
        # self.iteration is 0 at first pass
        # and we need to start with 1 here
        weight_momentums_corrected = layer.weight_momentums / \
            (1 - self.beta_1 ** (self.iterations + 1))
        bias_momentums_corrected = layer.bias_momentums / \
            (1 - self.beta_1 ** (self.iterations + 1))
        # Update cache with squared current gradients
        layer.weight_cache = self.beta_2 * layer.weight_cache + \
            (1 - self.beta_2) * layer.dweights**2
        layer.bias_cache = self.beta_2 * layer.bias_cache + \
            (1 - self.beta_2) * layer.dbiases**2
```

```
        # Get corrected cache
        weight_cache_corrected = layer.weight_cache / \
            (1 - self.beta_2 ** (self.iterations + 1))
        bias_cache_corrected = layer.bias_cache / \
            (1 - self.beta_2 ** (self.iterations + 1))

        # Vanilla SGD parameter update + normalization
        # with square rooted cache
        layer.weights += -self.current_learning_rate * \
                         weight_momentums_corrected / \
                         (np.sqrt(weight_cache_corrected) +
                             self.epsilon)
        layer.biases += -self.current_learning_rate * \
                         bias_momentums_corrected / \
                         (np.sqrt(bias_cache_corrected) +
                             self.epsilon)

    # Call once after any parameter updates
    def post_update_params(self):
        self.iterations += 1


# Common loss class
class Loss:

    # Regularization loss calculation
    def regularization_loss(self, layer):

        # 0 by default
        regularization_loss = 0

        # L1 regularization - weights
        # calculate only when factor greater than 0
        if layer.weight_regularizer_l1 > 0:
            regularization_loss += layer.weight_regularizer_l1 * \
                                   np.sum(np.abs(layer.weights))

        # L2 regularization - weights
        if layer.weight_regularizer_l2 > 0:
            regularization_loss += layer.weight_regularizer_l2 * \
                                   np.sum(layer.weights *
                                          layer.weights)

        # L1 regularization - biases
        # calculate only when factor greater than 0
        if layer.bias_regularizer_l1 > 0:
            regularization_loss += layer.bias_regularizer_l1 * \
                                   np.sum(np.abs(layer.biases))
```

```
        # L2 regularization - biases
        if layer.bias_regularizer_l2 > 0:
            regularization_loss += layer.bias_regularizer_l2 * \
                                   np.sum(layer.biases *
                                          layer.biases)

        return regularization_loss

    # Calculates the data and regularization losses
    # given model output and ground truth values
    def calculate(self, output, y):

        # Calculate sample losses
        sample_losses = self.forward(output, y)

        # Calculate mean loss
        data_loss = np.mean(sample_losses)

        # Return loss
        return data_loss


# Cross-entropy loss
class Loss_CategoricalCrossentropy(Loss):

    # Forward pass
    def forward(self, y_pred, y_true):

        # Number of samples in a batch
        samples = len(y_pred)

        # Clip data to prevent division by 0
        # Clip both sides to not drag mean towards any value
        y_pred_clipped = np.clip(y_pred, 1e-7, 1 - 1e-7)

        # Probabilities for target values -
        # only if categorical labels
        if len(y_true.shape) == 1:
            correct_confidences = y_pred_clipped[
                range(samples),
                y_true
            ]

        # Mask values - only for one-hot encoded labels
        elif len(y_true.shape) == 2:
            correct_confidences = np.sum(
                y_pred_clipped * y_true,
                axis=1
            )
```

```
        # Losses
        negative_log_likelihoods = -np.log(correct_confidences)
        return negative_log_likelihoods

    # Backward pass
    def backward(self, dvalues, y_true):

        # Number of samples
        samples = len(dvalues)
        # Number of labels in every sample
        # We'll use the first sample to count them
        labels = len(dvalues[0])

        # If labels are sparse, turn them into one-hot vector
        if len(y_true.shape) == 1:
            y_true = np.eye(labels)[y_true]

        # Calculate gradient
        self.dinputs = -y_true / dvalues
        # Normalize gradient
        self.dinputs = self.dinputs / samples

# Softmax classifier - combined Softmax activation
# and cross-entropy loss for faster backward step
class Activation_Softmax_Loss_CategoricalCrossentropy():

    # Creates activation and loss function objects
    def __init__(self):
        self.activation = Activation_Softmax()
        self.loss = Loss_CategoricalCrossentropy()

    # Forward pass
    def forward(self, inputs, y_true):
        # Output layer's activation function
        self.activation.forward(inputs)
        # Set the output
        self.output = self.activation.output
        # Calculate and return loss value
        return self.loss.calculate(self.output, y_true)

    # Backward pass
    def backward(self, dvalues, y_true):

        # Number of samples
        samples = len(dvalues)
```

```
        # If labels are one-hot encoded,
        # turn them into discrete values
        if len(y_true.shape) == 2:
            y_true = np.argmax(y_true, axis=1)

        # Copy so we can safely modify
        self.dinputs = dvalues.copy()
        # Calculate gradient
        self.dinputs[range(samples), y_true] -= 1
        # Normalize gradient
        self.dinputs = self.dinputs / samples


# Binary cross-entropy loss
class Loss_BinaryCrossentropy(Loss):

    # Forward pass
    def forward(self, y_pred, y_true):

        # Clip data to prevent division by 0
        # Clip both sides to not drag mean towards any value
        y_pred_clipped = np.clip(y_pred, 1e-7, 1 - 1e-7)

        # Calculate sample-wise loss
        sample_losses = -(y_true * np.log(y_pred_clipped) +
                          (1 - y_true) * np.log(1 - y_pred_clipped))
        sample_losses = np.mean(sample_losses, axis=-1)

        # Return losses
        return sample_losses

    # Backward pass
    def backward(self, dvalues, y_true):

        # Number of samples
        samples = len(dvalues)
        # Number of outputs in every sample
        # We'll use the first sample to count them
        outputs = len(dvalues[0])

        # Clip data to prevent division by 0
        # Clip both sides to not drag mean towards any value
        clipped_dvalues = np.clip(dvalues, 1e-7, 1 - 1e-7)

        # Calculate gradient
        self.dinputs = -(y_true / clipped_dvalues -
                         (1 - y_true) / (1 - clipped_dvalues)) / outputs
        # Normalize gradient
        self.dinputs = self.dinputs / samples
```

```
# Create dataset
X, y = spiral_data(samples=100, classes=2)

# Reshape labels to be a list of lists
# Inner list contains one output (either 0 or 1)
# per each output neuron, 1 in this case
y = y.reshape(-1, 1)
# Create Dense layer with 2 input features and 64 output values
dense1 = Layer_Dense(2, 64, weight_regularizer_l2=5e-4,
                            bias_regularizer_l2=5e-4)

# Create ReLU activation (to be used with Dense layer):
activation1 = Activation_ReLU()

# Create second Dense layer with 64 input features (as we take output
# of previous layer here) and 1 output value
dense2 = Layer_Dense(64, 1)

# Create Sigmoid activation:
activation2 = Activation_Sigmoid()

# Create loss function
loss_function = Loss_BinaryCrossentropy()

# Create optimizer
optimizer = Optimizer_Adam(decay=5e-7)


# Train in loop
for epoch in range(10001):

    # Perform a forward pass of our training data through this layer
    dense1.forward(X)

    # Perform a forward pass through activation function
    # takes the output of first dense layer here
    activation1.forward(dense1.output)

    # Perform a forward pass through second Dense layer
    # takes outputs of activation function
    # of first layer as inputs
    dense2.forward(activation1.output)

    # Perform a forward pass through activation function
    # takes the output of second dense layer here
    activation2.forward(dense2.output)

    # Calculate the data loss
    data_loss = loss_function.calculate(activation2.output, y)
```

```
    # Calculate regularization penalty
    regularization_loss = \
        loss_function.regularization_loss(dense1) + \
        loss_function.regularization_loss(dense2)

    # Calculate overall loss
    loss = data_loss + regularization_loss

    # Calculate accuracy from output of activation2 and targets
    # Part in the brackets returns a binary mask - array consisting
    # of True/False values, multiplying it by 1 changes it into array
    # of 1s and 0s
    predictions = (activation2.output > 0.5) * 1
    accuracy = np.mean(predictions == y)

    if not epoch % 100:
        print(f'epoch: {epoch}, ' +
              f'acc: {accuracy:.3f}, '+
              f'loss: {loss:.3f} (' +
              f'data_loss: {data_loss:.3f}, ' +
              f'reg_loss: {regularization_loss:.3f}), ' +
              f'lr: {optimizer.current_learning_rate}')

    # Backward pass
    loss_function.backward(activation2.output, y)
    activation2.backward(loss_function.dinputs)
    dense2.backward(activation2.dinputs)
    activation1.backward(dense2.dinputs)
    dense1.backward(activation1.dinputs)

    # Update weights and biases
    optimizer.pre_update_params()
    optimizer.update_params(dense1)
    optimizer.update_params(dense2)
    optimizer.post_update_params()


# Validate the model

# Create test dataset
X_test, y_test = spiral_data(samples=100, classes=2)

# Reshape labels to be a list of lists
# Inner list contains one output (either 0 or 1)
# per each output neuron, 1 in this case
y_test = y_test.reshape(-1, 1)
```

```
# Perform a forward pass of our testing data through this layer
dense1.forward(X_test)

# Perform a forward pass through activation function
# takes the output of first dense layer here
activation1.forward(dense1.output)

# Perform a forward pass through second Dense layer
# takes outputs of activation function of first layer as inputs
dense2.forward(activation1.output)

# Perform a forward pass through activation function
# takes the output of second dense layer here
activation2.forward(dense2.output)

# Calculate the data loss
loss = loss_function.calculate(activation2.output, y_test)

# Calculate accuracy from output of activation2 and targets
# Part in the brackets returns a binary mask - array consisting of
# True/False values, multiplying it by 1 changes it into array
# of 1s and 0s
predictions = (activation2.output > 0.5) * 1
accuracy = np.mean(predictions == y_test)

print(f'validation, acc: {accuracy:.3f}, loss: {loss:.3f}')
```

```
>>>
epoch: 0, acc: 0.500, loss: 0.693 (data_loss: 0.693, reg_loss: 0.000), lr:
0.001
epoch: 100, acc: 0.630, loss: 0.674 (data_loss: 0.673, reg_loss: 0.001), lr:
0.0009999505024501287
epoch: 200, acc: 0.625, loss: 0.669 (data_loss: 0.668, reg_loss: 0.001), lr:
0.0009999005098992651
epoch: 300, acc: 0.650, loss: 0.664 (data_loss: 0.663, reg_loss: 0.002), lr:
0.000999850522346909
epoch: 400, acc: 0.650, loss: 0.659 (data_loss: 0.657, reg_loss: 0.002), lr:
0.0009998005397923115
epoch: 500, acc: 0.675, loss: 0.647 (data_loss: 0.644, reg_loss: 0.004), lr:
0.0009997505622347225
epoch: 600, acc: 0.720, loss: 0.632 (data_loss: 0.625, reg_loss: 0.006), lr:
0.0009997005896733929
...
epoch: 1500, acc: 0.805, loss: 0.503 (data_loss: 0.464, reg_loss: 0.039),
lr: 0.0009992510613295335
...
epoch: 2500, acc: 0.855, loss: 0.430 (data_loss: 0.379, reg_loss: 0.052),
lr: 0.0009987520593019025
```

```
...
epoch: 4500, acc: 0.910, loss: 0.346 (data_loss: 0.285, reg_loss: 0.061),
lr: 0.0009977555488927658
epoch: 4600, acc: 0.905, loss: 0.340 (data_loss: 0.278, reg_loss: 0.062),
lr: 0.000997705775569079
epoch: 4700, acc: 0.910, loss: 0.330 (data_loss: 0.268, reg_loss: 0.062),
lr: 0.0009976560072110577
epoch: 4800, acc: 0.920, loss: 0.326 (data_loss: 0.263, reg_loss: 0.063),
lr: 0.0009976062438179587
...
epoch: 6100, acc: 0.940, loss: 0.291 (data_loss: 0.223, reg_loss: 0.069),
lr: 0.0009969597711777935
...
epoch: 6600, acc: 0.950, loss: 0.279 (data_loss: 0.211, reg_loss: 0.068),
lr: 0.000996711350897713
epoch: 6700, acc: 0.955, loss: 0.272 (data_loss: 0.203, reg_loss: 0.069),
lr: 0.0009966616816971556
epoch: 6800, acc: 0.955, loss: 0.269 (data_loss: 0.200, reg_loss: 0.069),
lr: 0.00099661201744669
epoch: 6900, acc: 0.960, loss: 0.266 (data_loss: 0.197, reg_loss: 0.069),
lr: 0.0009965623581455767
...
epoch: 9800, acc: 0.965, loss: 0.222 (data_loss: 0.158, reg_loss: 0.063),
lr: 0.0009951243880606966
epoch: 9900, acc: 0.965, loss: 0.221 (data_loss: 0.157, reg_loss: 0.063),
lr: 0.0009950748768967994
epoch: 10000, acc: 0.965, loss: 0.219 (data_loss: 0.156, reg_loss: 0.063),
lr: 0.0009950253706593885
validation, acc: 0.945, loss: 0.207
```

The model performed quite well here! You should have some intuition about tweaking the output layer to better fit the problem you're attempting to solve while keeping your hidden layers mostly the same. In the next chapter, we're going to work on regression, where our intended output is not a classification at all, but rather to predict a scalar value, like the price of a house.

Supplementary Material: https://nnfs.io/ch16
Chapter code, further resources, and errata for this chapter.

Chapter 17

Regression

Up until this point, we've been working with classification models, where we try to determine *what* something is. Now we're curious about determining a *specific* value based on an input. For example, you might want to use a neural network to predict what the temperature will be tomorrow or what the price of a car should be. For a task like this, we need something with a much more granular output. This also means that we require a new way to measure loss, as well as a new output layer activation function! It also means our data are different. We need training data that have target scalar values, not classes.

```
import matplotlib.pyplot as plt
import nnfs
from nnfs.datasets import sine_data

nnfs.init()


X, y = sine_data()
```

```
plt.plot(X, y)
plt.show()
```

The data above will produce a graph like:

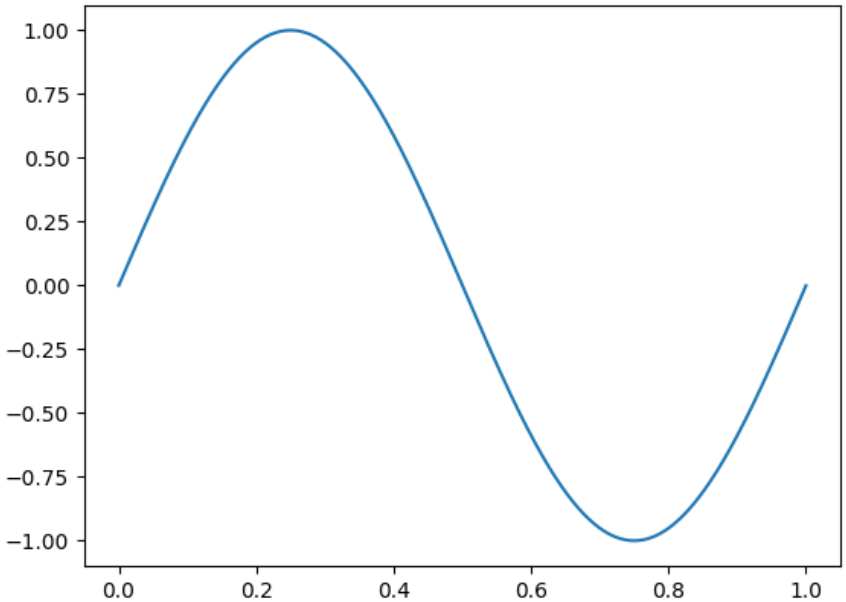


Fig 17.01: The sine data graph.

Linear Activation

Since we're no longer using classification labels and want to predict a scalar value, we're going to use a linear activation function for the output layer. This linear function does not modify its input and passes it to the output: $y=x$. For the backward pass, we already know the derivative of $f(x)=x$ is 1; thus, the full class for our new linear activation function is:

```
# Linear activation
class Activation_Linear:

    # Forward pass
    def forward(self, inputs):
        # Just remember values
        self.inputs = inputs
        self.output = inputs

    # Backward pass
    def backward(self, dvalues):
        # derivative is 1, 1 * dvalues = dvalues - the chain rule
        self.dinputs = dvalues.copy()
```

This might raise a question — why do we even write some code that does nothing? We just pass inputs to outputs for the forward pass and do the same with gradients during the backward pass since, to apply the chain rule, we multiply incoming gradients by the derivative, which is 1. We do this only for completeness and clarity to see the activation function of the output layer in the model definition code. From a computational time point of view, this adds almost nothing to the processing time, at least not enough to noticeably impact training times.

Now we just need to figure out loss!

Mean Squared Error Loss

Since we aren't working with classification labels anymore, we cannot calculate cross-entropy. Instead, we need some new methods. The two main methods for calculating error in regression are **mean squared error** (MSE) and **mean absolute error** (MAE).

With **mean squared error**, you square the difference between the predicted and true values of single outputs (as the model can have multiple regression outputs) and average those squared values.

$$L_i = \frac{1}{J}\sum_j (y_{i,j} - \hat{y}_{i,j})^2$$

Where *y* means the target value, *y-hat* means predicted value, index *i* means the current sample, index *j* means the current output in this sample, and the *J* means the number of outputs.

The idea here is to penalize more harshly the further away we get from the intended target.

Mean Squared Error Loss Derivative

The partial derivative of squared error with respect to the predicted value is:

$$\frac{\partial}{\partial \hat{y}_{i,j}} L_i = \frac{\partial}{\partial \hat{y}_{i,j}} [\frac{1}{J} \sum_j (y_{i,j} - \hat{y}_{i,j})^2] =$$

1 divided by *J* (the number of outputs) is a constant and can be moved outside of the derivative. Since we are calculating the derivative with respect to the given output, *j*, the sum of one element equals this element:

$$= \frac{1}{J} \cdot \frac{\partial}{\partial \hat{y}_{i,j}} (y_{i,j} - \hat{y}_{i,j})^2 =$$

To calculate the partial derivative of an expression to the power of some value, we need to multiply this exponent by the expression, subtract 1 from the exponent, and multiply this by the partial derivative of the inner function:

$$= \frac{1}{J} \cdot 2 \cdot (y_{i,j} - \hat{y}_{i,j})^{2-1} \cdot \frac{\partial}{\partial \hat{y}_{i,j}} [y_{i,j} - \hat{y}_{i,j}] =$$

The partial derivative of the subtraction equals the subtraction of the partial derivatives:

$$= \frac{2}{J} \cdot (y_{i,j} - \hat{y}_{i,j})^1 \cdot (\frac{\partial}{\partial \hat{y}_{i,j}} y_{i,j} - \frac{\partial}{\partial \hat{y}_{i,j}} \hat{y}_{i,j}) =$$

The partial derivative of the ground truth value with respect to the predicted value equals *0* since we treat other variables as constants. The partial derivative of the predicted value with respect to itself equals *1*, which results in *0-1=-1*. This is multiplied by the rest of the equation and forms the solution:

$$= \frac{2}{J} \cdot (y_{i,j} - \hat{y}_{i,j}) \cdot (0 - 1) = \frac{2}{J} \cdot (y_{i,j} - \hat{y}_{i,j}) \cdot (-1) = -\frac{2}{J} \cdot (y_{i,j} - \hat{y}_{i,j})$$

Full solution:

$$\frac{\partial}{\partial \hat{y}_{i,j}} L_i = \frac{\partial}{\partial \hat{y}_{i,j}} [\frac{1}{J} \sum_j (y_{i,j} - \hat{y}_{i,j})^2] = \frac{1}{J} \cdot \frac{\partial}{\partial \hat{y}_{i,j}} (y_{i,j} - \hat{y}_{i,j})^2 =$$

$$= \frac{1}{J} \cdot 2 \cdot (y_{i,j} - \hat{y}_{i,j})^{2-1} \cdot \frac{\partial}{\partial \hat{y}_{i,j}} [y_{i,j} - \hat{y}_{i,j}] =$$

$$= \frac{2}{J} \cdot (y_{i,j} - \hat{y}_{i,j})^1 \cdot (\frac{\partial}{\partial \hat{y}_{i,j}} y_{i,j} - \frac{\partial}{\partial \hat{y}_{i,j}} \hat{y}_{i,j}) = \frac{2}{J} \cdot (y_{i,j} - \hat{y}_{i,j}) \cdot (0 - 1) =$$

$$= \frac{2}{J} \cdot (y_{i,j} - \hat{y}_{i,j}) \cdot (-1) = -\frac{2}{J} \cdot (y_{i,j} - \hat{y}_{i,j})$$

The partial derivative equals *-2*, multiplied by the subtraction of the true and predicted values, and then divided by the number of outputs to normalize the gradients, making their magnitude invariant to the number of outputs.

Mean Squared Error (MSE) Loss Code

The code for MSE includes an implementation of the equation to calculate the sample loss from multiple outputs. `axis=-1` with the mean calculation was explained in the previous chapter in detail and, in short words, it informs NumPy to calculate mean across outputs, for each sample separately. For the backward pass, we implemented the derivative equation, which results in *-2* multiplied by the difference of true and predicted values, and normalized by the number of outputs. Similarly to the other loss function implementations, we also normalize gradients by the number of samples to make them invariant to the batch size, or the number of samples in general:

```
# Mean Squared Error loss
class Loss_MeanSquaredError(Loss):  # L2 loss

    # Forward pass
    def forward(self, y_pred, y_true):

        # Calculate loss
        sample_losses = np.mean((y_true - y_pred)**2, axis=-1)
```

```
        # Return losses
        return sample_losses

    # Backward pass
    def backward(self, dvalues, y_true):

        # Number of samples
        samples = len(dvalues)
        # Number of outputs in every sample
        # We'll use the first sample to count them
        outputs = len(dvalues[0])

        # Gradient on values
        self.dinputs = -2 * (y_true - dvalues) / outputs
        # Normalize gradient
        self.dinputs = self.dinputs / samples
```

Mean Absolute Error Loss

With **mean absolute error**, you take the absolute difference between the predicted and true values in a single output and average those absolute values.

$$L_i = \frac{1}{J} \sum_j |y_{i,j} - \hat{y}_{i,j}|$$

Where *y* means the target value, *y-hat* means predicted value, index *i* means the current sample, index *j* means the current output in this sample, and the *J* means the number of outputs.

This function, used as a loss, penalizes the error linearly. It produces sparser results and is robust to outliers, which can be both advantageous and disadvantageous. In reality, L1 (MAE) loss is used less frequently than L2 (MSE) loss.

Mean Absolute Error Loss Derivative

The partial derivative for absolute error with respect to the predicted values is:

$$\frac{\partial}{\partial \hat{y}_{i,j}} L_i = \frac{\partial}{\partial \hat{y}_{i,j}} [\frac{1}{J} \sum_j |y_{i,j} - \hat{y}_{i,j}|]$$

1 divided by *J* (the number of outputs) is a constant, and can be moved outside of the derivative. Since we are calculating the derivative with respect to the given output, *j*, the sum of one element equals this element:

$$= \frac{1}{J} \cdot \frac{\partial}{\partial \hat{y}_{i,j}} |y_{i,j} - \hat{y}_{i,j}| =$$

We already calculated the partial derivative of an absolute value for the L1 regularization, which is similar to the L1 loss. The derivative of an absolute value equals *1* if this value is greater than *0*, or *-1* if it's less than *0*. The derivative does not exist for a value of *0*:

$$= \frac{1}{J} \cdot \begin{cases} 1 & y_{i,j} - \hat{y}_{i,j} > 0 \\ -1 & y_{i,j} - \hat{y}_{i,j} < 0 \end{cases}$$

Full solution:

$$\frac{\partial}{\partial \hat{y}_{i,j}} L_i = \frac{\partial}{\partial \hat{y}_{i,j}} [\frac{1}{J} \sum_j |y_{i,j} - \hat{y}_{i,j}|] = \frac{1}{J} \cdot \frac{\partial}{\partial \hat{y}_{i,j}} |y_{i,j} - \hat{y}_{i,j}| = \frac{1}{J} \cdot \begin{cases} 1 & y_{i,j} - \hat{y}_{i,j} > 0 \\ -1 & y_{i,j} - \hat{y}_{i,j} < 0 \end{cases}$$

Mean Absolute Error Loss Code

The code for mean absolute error is very similar to the mean squared error. The forward pass includes NumPy's `np.abs()` to calculate absolute values before calculating the mean. For the backward pass, we'll use `np.sign()`, which returns 1 or -1 given the sign of the input and *0* if the parameter equals *0*, then normalize gradients by the number of samples to make them invariant to the batch size, or number of samples in general:

```
# Mean Absolute Error loss
class Loss_MeanAbsoluteError(Loss):  # L1 loss

    # Forward pass
    def forward(self, y_pred, y_true):

        # Calculate loss
        sample_losses = np.mean(np.abs(y_true - y_pred), axis=-1)

        # Return losses
        return sample_losses

    # Backward pass
    def backward(self, dvalues, y_true):

        # Number of samples
        samples = len(dvalues)
        # Number of outputs in every sample
        # We'll use the first sample to count them
        outputs = len(dvalues[0])

        # Calculate gradient
        self.dinputs = np.sign(y_true - dvalues) / outputs
        # Normalize gradient
        self.dinputs = self.dinputs / samples
```

Accuracy in Regression

Now that we've got data, an activation function, and a loss calculation for regression, we'd like to measure performance.

With cross-entropy, we were able to count the number of matches (situations where the prediction equals the ground truth target), and then divide it by the number of samples to measure the model's accuracy. With a regression model, we have two problems: the first problem is that each output neuron in the model (there might be many) is a separate output — like in a binary regression model and unlike in a classifier, where all outputs contribute toward a common prediction. The second problem is that the prediction is a float value, and we can't simply check if the output value equals the ground truth one, as it most likely won't — if it differs even slightly, the accuracy will be a 0. For example, if your model predicts home prices and one of the samples has the target price of $192,500, and the predicted value is $192,495, then a pure "is it equal" assessment would return False. We'd likely consider the predicted price to be correct or "close enough" in this scenario, given the magnitude of the numbers in consideration. There's no perfect way to show accuracy with regression. Still, it is preferable to have some accuracy metric.

For example, Keras, a popular deep learning framework, shows both accuracy and loss for regression models, and we'll also make our own accuracy metric.

First, we need some "limit" value, which we'll call "precision." To calculate this precision, we'll calculate the standard deviation from the ground truth target values and then divide it by *250*. This value can certainly vary depending on your goals. The larger the number you divide by, the more "strict" the accuracy metric will be. *250* is our value of choice. Code to represent this:

```
accuracy_precision = np.std(y) / 250
```

Then we could use this precision value as a sort of "cushion allowance" for regression outputs when comparing targets and predicted values for accuracy. We perform the comparison by applying the absolute value on the difference between the ground truth values and the predictions. Then we check if the difference is smaller than our previously calculated precision:

```
predictions = activation2.output
accuracy = np.mean(np.absolute(predictions - y) <
           accuracy_precision)
```

Regression Model Training

With this new activation function, loss, and way of calculating accuracy, we now create our model:

```
# Create dataset
X, y = sine_data()

# Create Dense layer with 1 input feature and 64 output values
dense1 = Layer_Dense(1, 64)

# Create ReLU activation (to be used with Dense layer):
activation1 = Activation_ReLU()

# Create second Dense layer with 64 input features (as we take output
# of previous layer here) and 1 output value
dense2 = Layer_Dense(64, 1)
# Create Linear activation:
activation2 = Activation_Linear()

# Create loss function
loss_function = Loss_MeanSquaredError()

# Create optimizer
optimizer = Optimizer_Adam()

# Accuracy precision for accuracy calculation
# There are no really accuracy factor for regression problem,
# but we can simulate/approximate it. We'll calculate it by checking
# how many values have a difference to their ground truth equivalent
# less than given precision
# We'll calculate this precision as a fraction of standard deviation
# of al the ground truth values
accuracy_precision = np.std(y) / 250

# Train in loop
for epoch in range(10001):
```

```
# Perform a forward pass of our training data through this layer
dense1.forward(X)

# Perform a forward pass through activation function
# takes the output of first dense layer here
activation1.forward(dense1.output)

# Perform a forward pass through second Dense layer
# takes outputs of activation function
# of first layer as inputs
dense2.forward(activation1.output)

# Perform a forward pass through activation function
# takes the output of second dense layer here
activation2.forward(dense2.output)

# Calculate the data loss
data_loss = loss_function.calculate(activation2.output, y)

# Calculate regularization penalty
regularization_loss = \
    loss_function.regularization_loss(dense1) + \
    loss_function.regularization_loss(dense2)

# Calculate overall loss
loss = data_loss + regularization_loss

# Calculate accuracy from output of activation2 and targets
# To calculate it we're taking absolute difference between
# predictions and ground truth values and compare if differences
# are lower than given precision value
predictions = activation2.output
accuracy = np.mean(np.absolute(predictions - y) <
                   accuracy_precision)

if not epoch % 100:
    print(f'epoch: {epoch}, ' +
          f'acc: {accuracy:.3f}, ' +
          f'loss: {loss:.3f} (' +
          f'data_loss: {data_loss:.3f}, ' +
          f'reg_loss: {regularization_loss:.3f}), ' +
          f'lr: {optimizer.current_learning_rate}')

# Backward pass
loss_function.backward(activation2.output, y)
activation2.backward(loss_function.dinputs)
dense2.backward(activation2.dinputs)
activation1.backward(dense2.dinputs)
dense1.backward(activation1.dinputs)
```

```
    # Update weights and biases
    optimizer.pre_update_params()
    optimizer.update_params(dense1)
    optimizer.update_params(dense2)
    optimizer.post_update_params()
```

```
>>>
epoch: 0, acc: 0.002, loss: 0.500 (data_loss: 0.500, reg_loss: 0.000), lr:
0.001
epoch: 100, acc: 0.003, loss: 0.346 (data_loss: 0.346, reg_loss: 0.000), lr:
0.001
...
epoch: 9900, acc: 0.003, loss: 0.145 (data_loss: 0.145, reg_loss: 0.000),
lr: 0.001
epoch: 10000, acc: 0.004, loss: 0.145 (data_loss: 0.145, reg_loss: 0.000),
lr: 0.001
```

Training didn't work out here very well! Let's add an ability to draw the testing data and let's also do a forward pass on the testing data, drawing output data on the same plot as well:

```
import matplotlib.pyplot as plt

X_test, y_test = sine_data()

dense1.forward(X_test)
activation1.forward(dense1.output)
dense2.forward(activation1.output)
activation2.forward(dense2.output)

plt.plot(X_test, y_test)
plt.plot(X_test, activation2.output)
plt.show()
```

First, we are importing matplotlib, then creating a new set of data. Next, we have 4 lines of the code that are the same as the forward pass from our code above. We could call it a prediction or, in the context of what we are going to do, validation. We'll cover both topics and explain what validation and prediction are in the future chapters. For now, it's enough to know that what we are doing here is predicting on the same feature-set that we've used to train the model in order to see what the model learned and returns for our data — seeing how close outputs are to the training ground-true values. We are then plotting the training data, which are obviously a sine, and prediction data, what we'd hope to form a sine as well. Let's run this code again and take a look at the generated image:

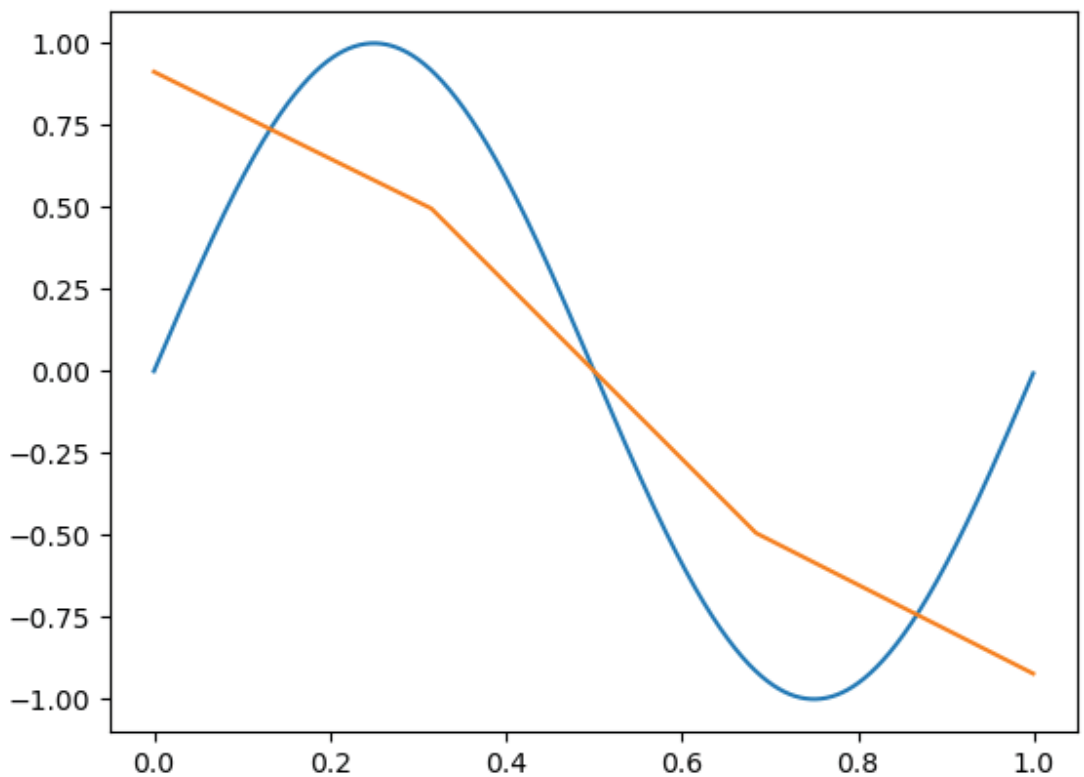


Fig 17.02: Model prediction - could not fit the sine data.

Animation of the training process:

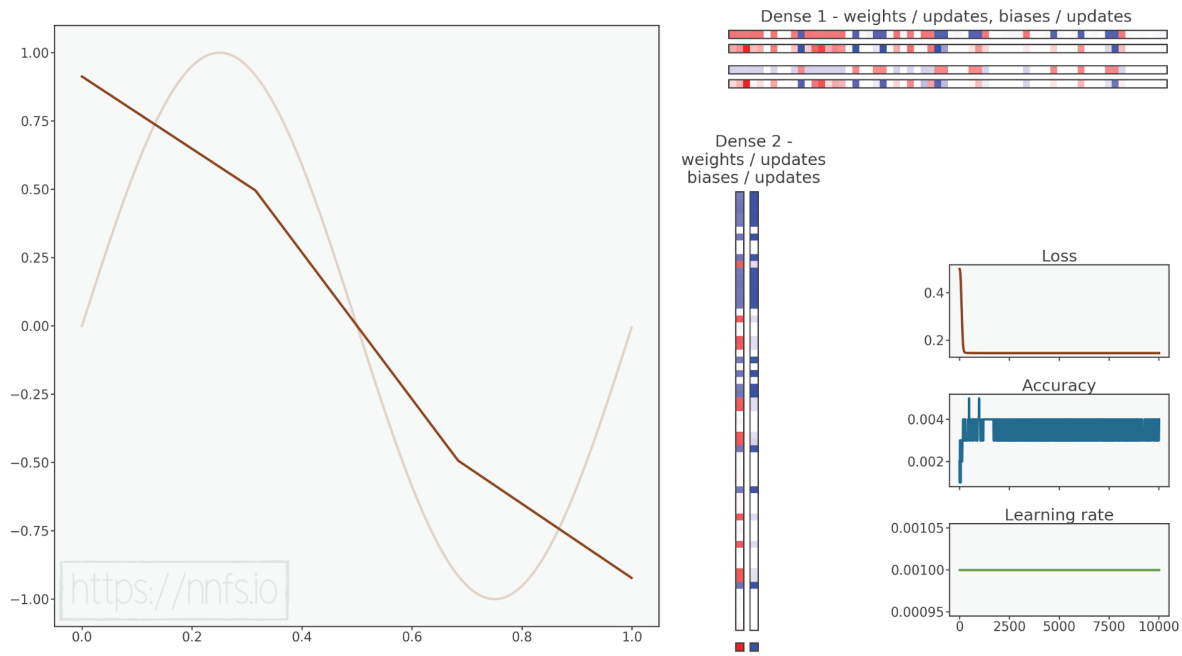


Fig 17.03: Model stoppped training immediately.

Anim 17.03: https://nnfs.io/ghi

Recall the rectified linear activation function and how its nonlinear behavior allowed us to map nonlinear functions, but we also needed two or more hidden layers. In this case, we have only 1 hidden layer followed by the output layer. As we should know by now, this is simply not enough!

If we add just one more layer:

```
# Create dataset
X, y = sine_data()

# Create Dense layer with 1 input feature and 64 output values
dense1 = Layer_Dense(1, 64)

# Create ReLU activation (to be used with Dense layer):
activation1 = Activation_ReLU()

# Create second Dense layer with 64 input features (as we take output
# of previous layer here) and 64 output values
dense2 = Layer_Dense(64, 64)

# Create ReLU activation (to be used with Dense layer):
activation2 = Activation_ReLU()

# Create third Dense layer with 64 input features (as we take output
# of previous layer here) and 1 output value
dense3 = Layer_Dense(64, 1)

# Create Linear activation:
activation3 = Activation_Linear()

# Create loss function
loss_function = Loss_MeanSquaredError()

# Create optimizer
optimizer = Optimizer_Adam()

# Accuracy precision for accuracy calculation
# There are no really accuracy factor for regression problem,
# but we can simulate/approximate it. We'll calculate it by checking
# how many values have a difference to their ground truth equivalent
# less than given precision
# We'll calculate this precision as a fraction of standard deviation
# of al the ground truth values
accuracy_precision = np.std(y) / 250

# Train in loop
for epoch in range(10001):

    # Perform a forward pass of our training data through this layer
    dense1.forward(X)
```

```
# Perform a forward pass through activation function
# takes the output of first dense layer here
activation1.forward(dense1.output)

# Perform a forward pass through second Dense layer
# takes outputs of activation function
# of first layer as inputs
dense2.forward(activation1.output)

# Perform a forward pass through activation function
# takes the output of second dense layer here
activation2.forward(dense2.output)

# Perform a forward pass through third Dense layer
# takes outputs of activation function of second layer as inputs
dense3.forward(activation2.output)

# Perform a forward pass through activation function
# takes the output of third dense layer here
activation3.forward(dense3.output)

# Calculate the data loss
data_loss = loss_function.calculate(activation3.output, y)

# Calculate regularization penalty
regularization_loss = \
    loss_function.regularization_loss(dense1) + \
    loss_function.regularization_loss(dense2) + \
    loss_function.regularization_loss(dense3)

# Calculate overall loss
loss = data_loss + regularization_loss

# Calculate accuracy from output of activation2 and targets
# To calculate it we're taking absolute difference between
# predictions and ground truth values and compare if differences
# are lower than given precision value
predictions = activation3.output
accuracy = np.mean(np.absolute(predictions - y) <
                               accuracy_precision)

if not epoch % 100:
    print(f'epoch: {epoch}, ' +
          f'acc: {accuracy:.3f}, ' +
          f'loss: {loss:.3f} (' +
          f'data_loss: {data_loss:.3f}, ' +
          f'reg_loss: {regularization_loss:.3f}), ' +
          f'lr: {optimizer.current_learning_rate}')
```

```
    # Backward pass
    loss_function.backward(activation3.output, y)
    activation3.backward(loss_function.dinputs)
    dense3.backward(activation3.dinputs)
    activation2.backward(dense3.dinputs)
    dense2.backward(activation2.dinputs)
    activation1.backward(dense2.dinputs)
    dense1.backward(activation1.dinputs)

    # Update weights and biases
    optimizer.pre_update_params()
    optimizer.update_params(dense1)
    optimizer.update_params(dense2)
    optimizer.update_params(dense3)
    optimizer.post_update_params()


import matplotlib.pyplot as plt

X_test, y_test = sine_data()

dense1.forward(X_test)
activation1.forward(dense1.output)
dense2.forward(activation1.output)
activation2.forward(dense2.output)
dense3.forward(activation2.output)
activation3.forward(dense3.output)

plt.plot(X_test, y_test)
plt.plot(X_test, activation3.output)
plt.show()


>>>
epoch: 0, acc: 0.002, loss: 0.500 (data_loss: 0.500, reg_loss: 0.000), lr:
0.001
epoch: 100, acc: 0.003, loss: 0.187 (data_loss: 0.187, reg_loss: 0.000), lr:
0.001
...
epoch: 9900, acc: 0.617, loss: 0.031 (data_loss: 0.031, reg_loss: 0.000),
lr: 0.001
epoch: 10000, acc: 0.620, loss: 0.031 (data_loss: 0.031, reg_loss: 0.000),
lr: 0.001
```

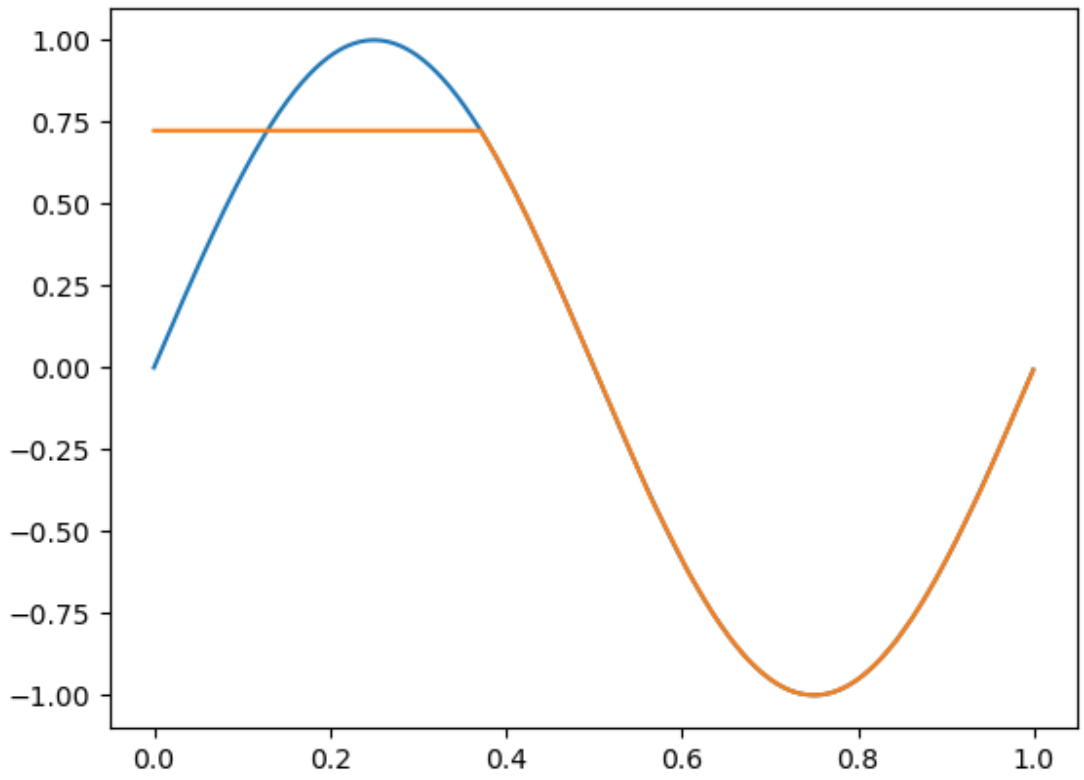


Fig 17.04: Model prediction - better fit to the data.

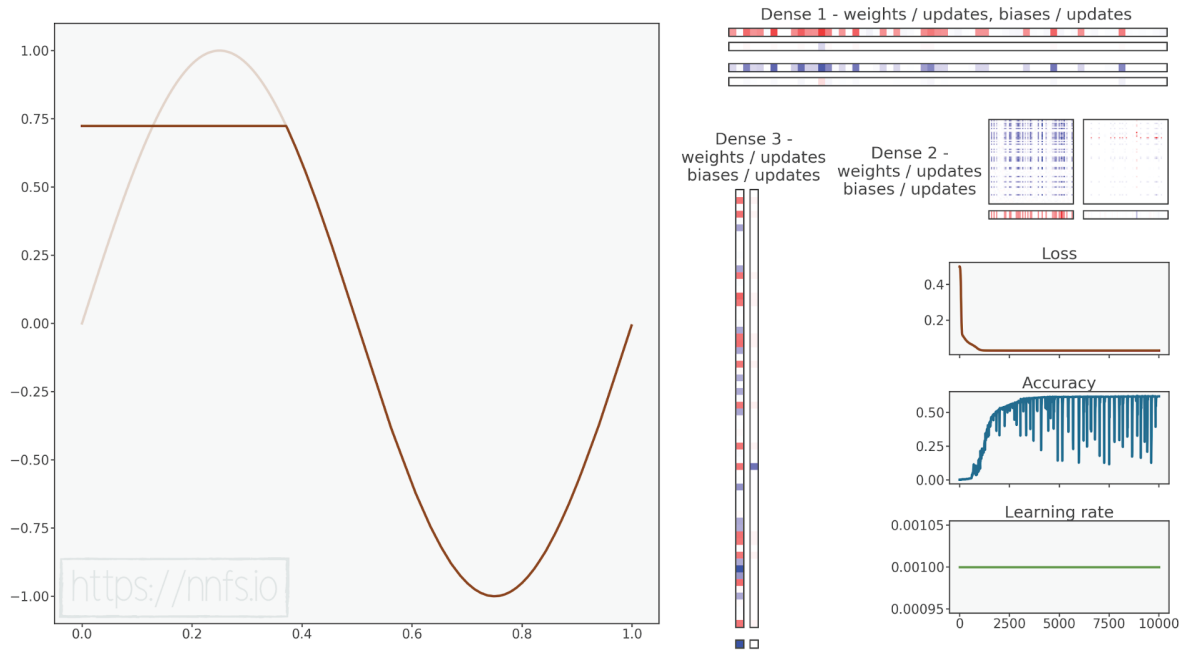


Fig 17.05: Model trainined to better fit the sine data.

Anim 17.05: https://nnfs.io/hij

Our model's accuracy is not very good, and loss seems stuck at a pretty high level for this model. The image shows us why this is the case, the model has some trouble fitting our data, and it looks like it might be stuck in a local minimum. As we have already learned, to try to help the model with being stuck at a local minimum, we might use a higher learning rate and add a learning rate decay. In the previous model, we have used the default learning rate, which is *0.001*. Let's set it to *0.01* and add learning rate decaying:

```
optimizer = Optimizer_Adam(learning_rate=0.01, decay=1e-3)
```

```
>>>
epoch: 0, acc: 0.002, loss: 0.500 (data_loss: 0.500, reg_loss: 0.000), lr:
0.01
epoch: 100, acc: 0.027, loss: 0.061 (data_loss: 0.061, reg_loss: 0.000), lr:
0.009099181073703368
...
epoch: 9900, acc: 0.565, loss: 0.031 (data_loss: 0.031, reg_loss: 0.000),
lr: 0.0009175153683824203
epoch: 10000, acc: 0.564, loss: 0.031 (data_loss: 0.031, reg_loss: 0.000),
lr: 0.0009091735612328393
```

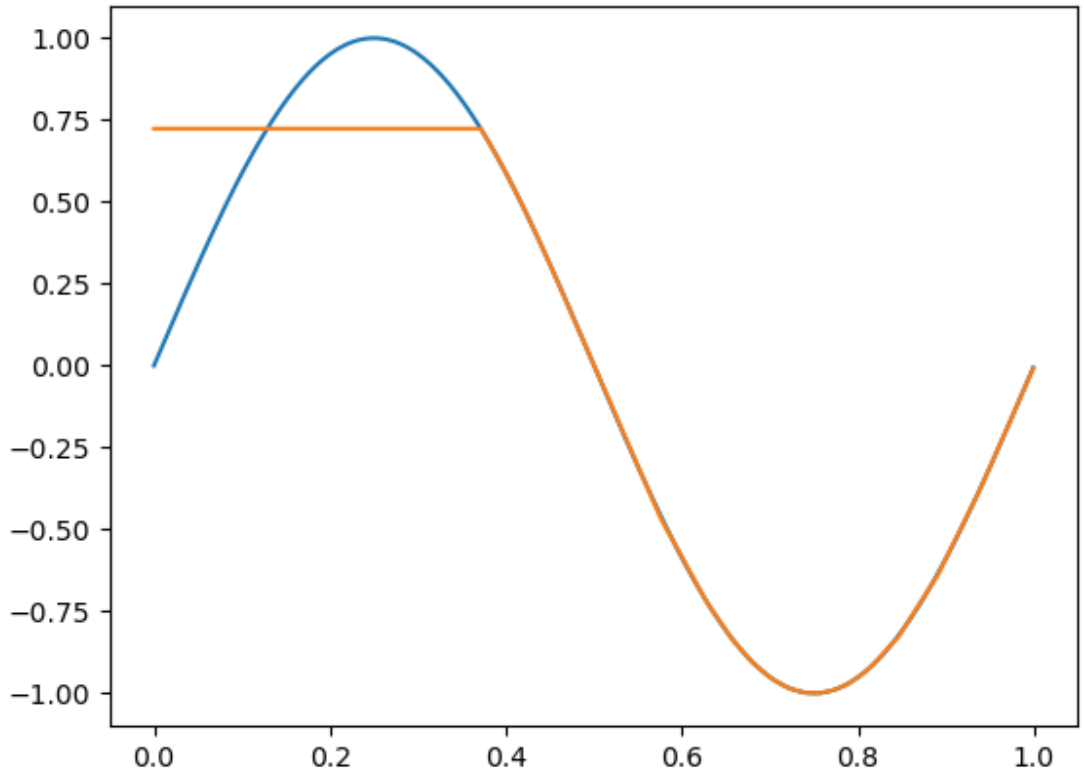


Fig 17.06: Model prediction - similar fit to the data.

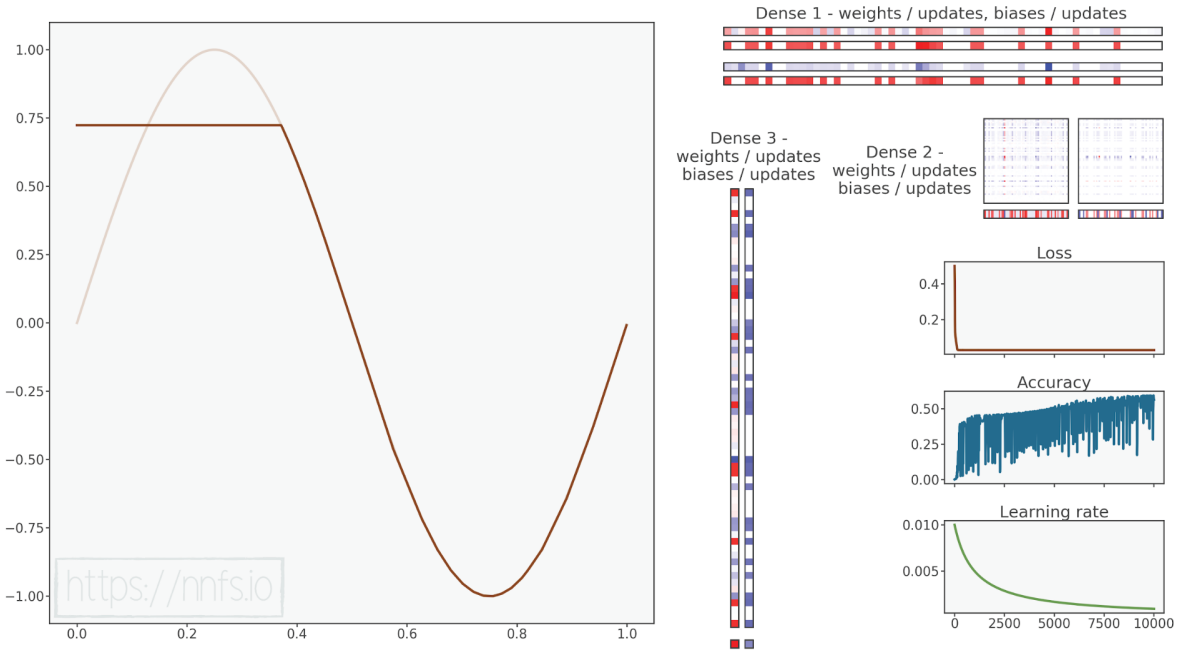


Fig 17.07: Model trainined to fit the sine data, similar fit.

Anim 17.07: https://nnfs.io/ijk

Our model seems to still be stuck with even lower accuracy this time. Let's try to use an even bigger learning rate then:

```
optimizer = Optimizer_Adam(learning_rate=0.05, decay=1e-3)
```

```
>>>
epoch: 0, acc: 0.002, loss: 0.500 (data_loss: 0.500, reg_loss: 0.000), lr:
0.05
epoch: 100, acc: 0.087, loss: 0.031 (data_loss: 0.031, reg_loss: 0.000), lr:
0.04549590536851684
...
epoch: 9900, acc: 0.275, loss: 0.031 (data_loss: 0.031, reg_loss: 0.000),
lr: 0.004587576841912101
epoch: 10000, acc: 0.229, loss: 0.031 (data_loss: 0.031, reg_loss: 0.000),
lr: 0.0045458678061641965
```

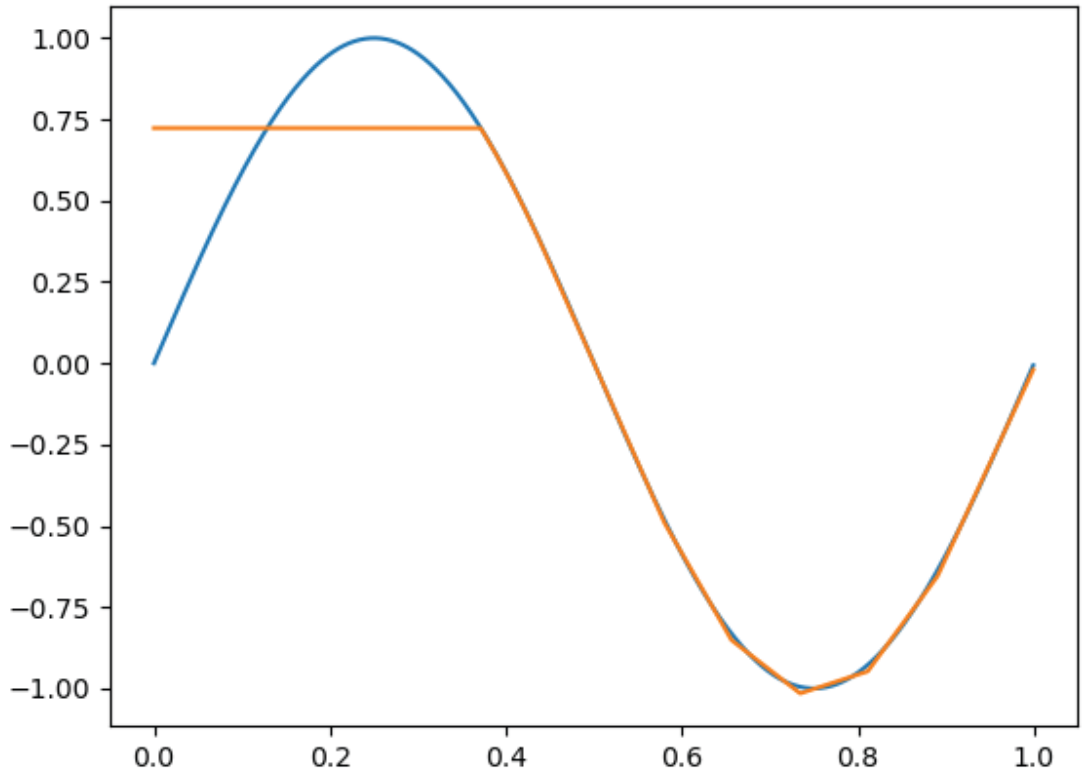


Fig 17.08: Model prediction - similar fit to the data.

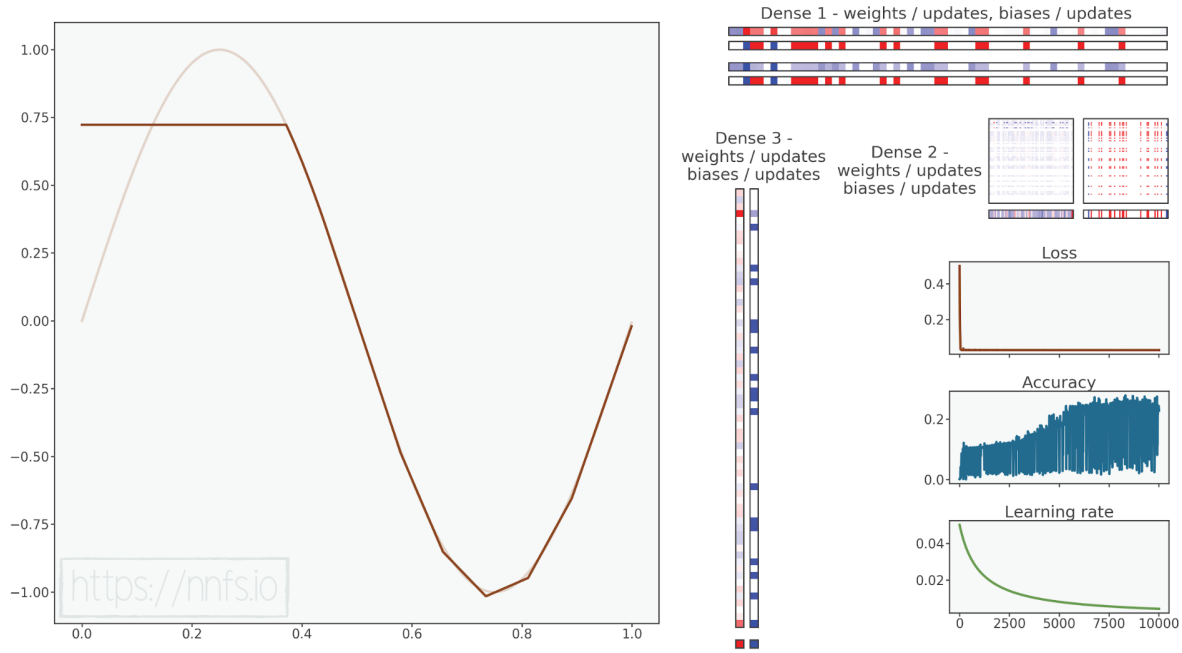


Fig 17.09: Model trainined to fit the sine data, similar fit.

Anim 17.09: https://nnfs.io/jkl

It's getting even worse. Accuracy drops significantly, and we can observe the lower part of the sine being of a worse shape as well. It seems like we are not able to make this model learn the data, but after multiple tests and tuning hyperparameters, we could find a learning rate of 0.005:

```
optimizer = Optimizer_Adam(learning_rate=0.005, decay=1e-3)
```

```
>>>
epoch: 0, acc: 0.002, loss: 0.500 (data_loss: 0.500, reg_loss: 0.000), lr:
0.005
epoch: 100, acc: 0.007, loss: 0.084 (data_loss: 0.084, reg_loss: 0.000), lr:
0.004549590536851684
...
epoch: 9900, acc: 0.964, loss: 0.000 (data_loss: 0.000, reg_loss: 0.000),
lr: 0.00045875768419121016
epoch: 10000, acc: 0.970, loss: 0.000 (data_loss: 0.000, reg_loss: 0.000),
lr: 0.00045458678061641964
```

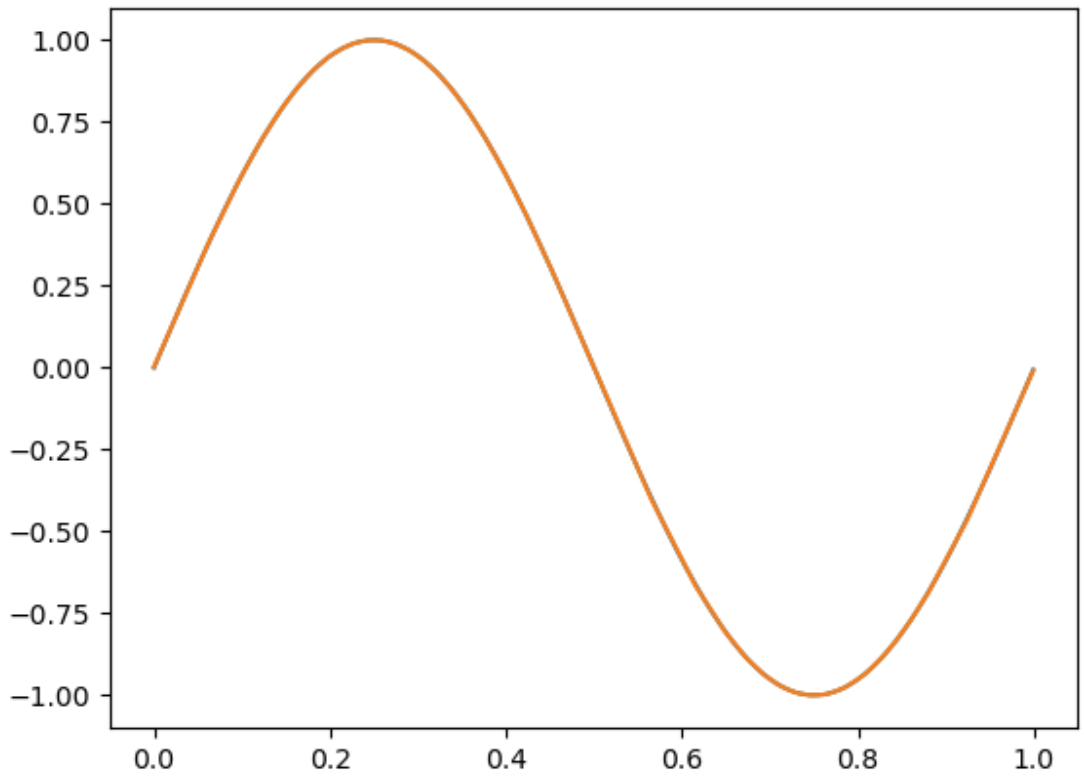


Fig 17.10: Model prediction - good fit to the data.

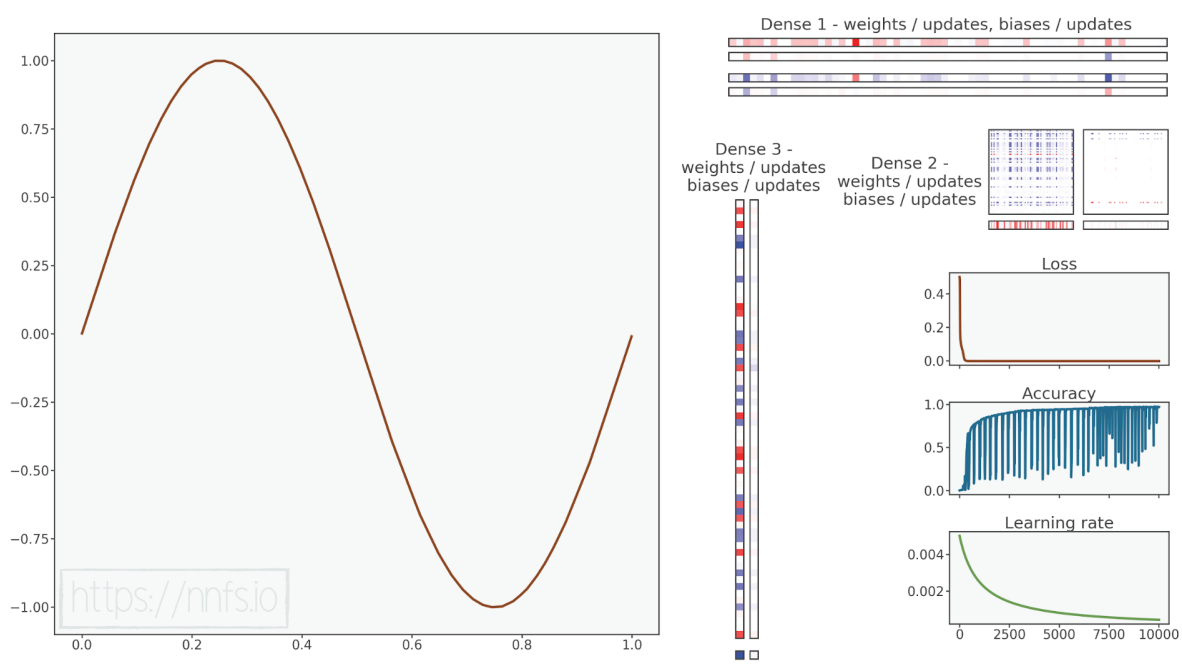


Fig 17.11: Model trainined to fit the sine data.

Anim 17.11: https://nnfs.io/klm

This time model has learned pretty well, but the curious part is that both lower and higher learning rates than what we used here initially caused accuracy to be pretty low and loss be stuck at the same value when the learning rate in between them actually worked.

Debugging such a problem is usually a pretty hard task and out of the scope of this book. The accuracy and loss suggest that updates to the parameters are not big enough, but the rising learning rate makes things only worse, and there is just this single spot that we were able to find that lets our model learn. You might recall that, back in chapter 3, we were discussing parameter initialization methods and why it's important to initialize them wisely. It turns out that, in the current case, we can help the model learn by changing the factor of 0.01 to 0.1 in the Dense layer's weight initialization. But then you might ask — since the learning rate is being used to decide how much of a gradient to apply to the parameters, why does changing these initial values help instead? As you may recall, the back-propagated gradient is calculated using weights, and the learning rate does not affect it. That's why it's important to use right weight initialization, and so far, we have been using the same values for each model.

If we, for example, take a look at the source code of Keras, a neural network framework, we can learn that:

```
def glorot_uniform(seed=None):
    """Glorot uniform initializer, also called Xavier uniform initializer.

    It draws samples from a uniform distribution within [-limit, limit]
    where `limit` is `sqrt(6 / (fan_in + fan_out))`
    where `fan_in` is the number of input units in the weight tensor
    and `fan_out` is the number of output units in the weight tensor.

    # Arguments
        seed: A Python integer. Used to seed the random generator.

    # Returns
        An initializer.

    # References
        Glorot & Bengio, AISTATS 2010
        http://jmlr.org/proceedings/papers/v9/glorot10a/glorot10a.pdf
    """
    return VarianceScaling(scale=1.,
                           mode='fan_avg',
                           distribution='uniform',
                           seed=seed)
```

This code is part of the Keras 2 library. The important part of the above is actually the comment section, which describes how it initializes weights. We can find there are important pieces of information to remember — the fraction that multiplies the draw from the uniform distribution depends on the number of inputs and the number of neurons and is not constant like in our case. This method of initialization is called **Glorot uniform**. We (the authors of this book) actually have had a very similar problem in one of our projects, and changing the way weights were initialized changed the model from not learning at all to a learning state.

For the purposes of this model, let's change the factor multiplying the draw from the normal distribution in the weight initialization of the *Dense* layer to 0.1 and re-run all four of the above attempts to compare results:

```
self.weights = 0.1 * np.random.randn(n_inputs, n_neurons)
```

And all above tests re-ran:

```
optimizer = Optimizer_Adam()
```

```
>>>
epoch: 0, acc: 0.003, loss: 0.496 (data_loss: 0.496, reg_loss: 0.000), lr:
0.001
epoch: 100, acc: 0.005, loss: 0.114 (data_loss: 0.114, reg_loss: 0.000), lr:
0.001
...
epoch: 9900, acc: 0.869, loss: 0.000 (data_loss: 0.000, reg_loss: 0.000),
lr: 0.001
epoch: 10000, acc: 0.883, loss: 0.000 (data_loss: 0.000, reg_loss: 0.000),
lr: 0.001
```

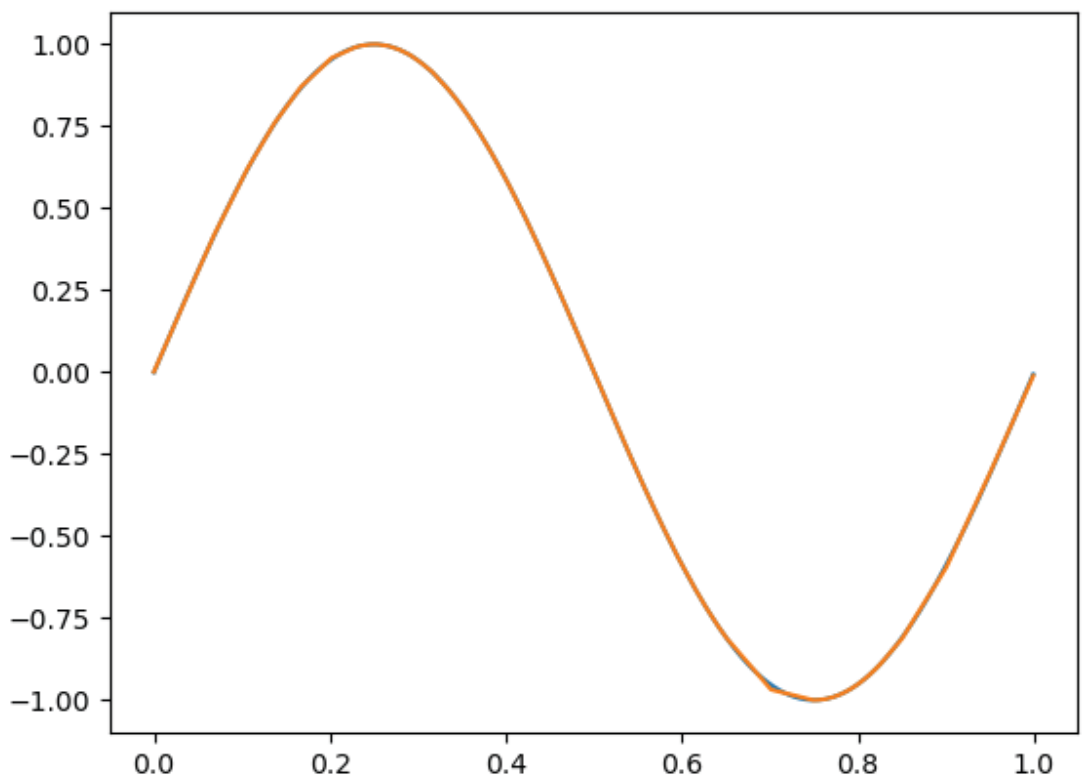


Fig 17.12: Model prediction - good fit to the data with different weight initialization.

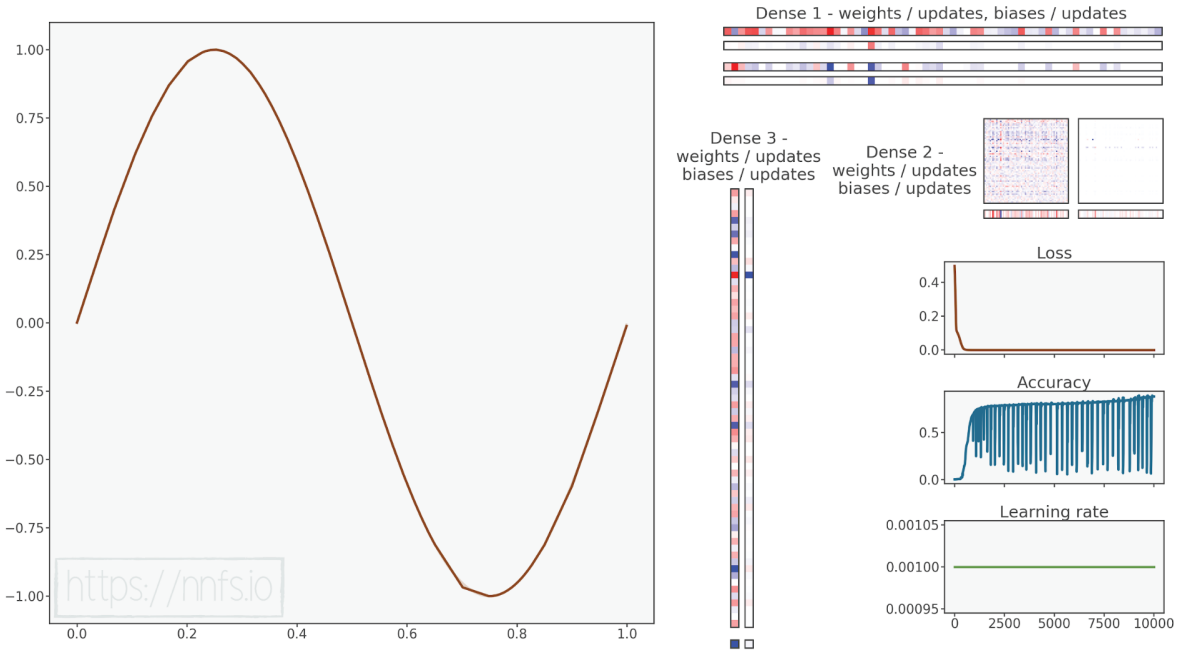


Fig 17.13: Model trained to fit the sine data after replacing weight initialization.

Anim 17.13: https://nnfs.io/lmn

This model was previously stuck and has now achieved high accuracy. There are some visible imperfections like at the bottom side of this sine, but the overall result is better.

```
optimizer = Optimizer_Adam(learning_rate=0.01, decay=1e-3)
```

```
>>>
epoch: 0, acc: 0.003, loss: 0.496 (data_loss: 0.496, reg_loss: 0.000), lr:
0.01
epoch: 100, acc: 0.065, loss: 0.011 (data_loss: 0.011, reg_loss: 0.000), lr:
0.009099181073703368
...
epoch: 9900, acc: 0.958, loss: 0.000 (data_loss: 0.000, reg_loss: 0.000),
lr: 0.0009175153683824203
```

```
epoch: 10000, acc: 0.949, loss: 0.000 (data_loss: 0.000, reg_loss: 0.000),
lr: 0.0009091735612328393
```

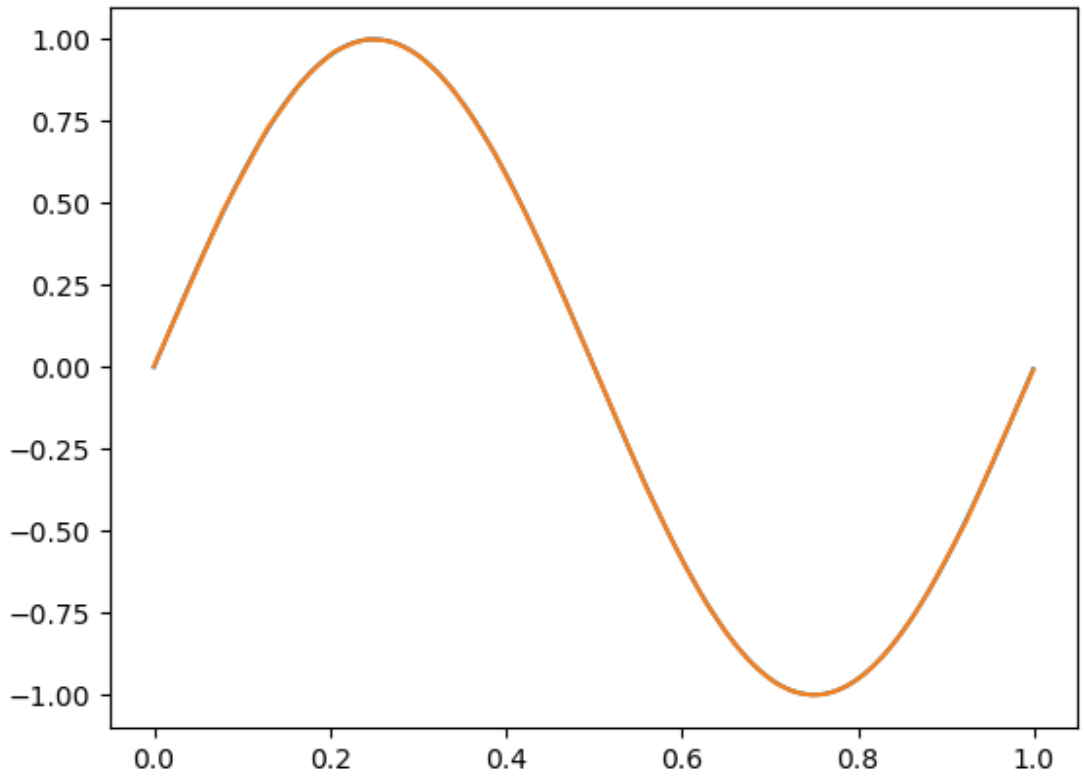


Fig 17.14: Model prediction - good fit to the data with different weight initialization.

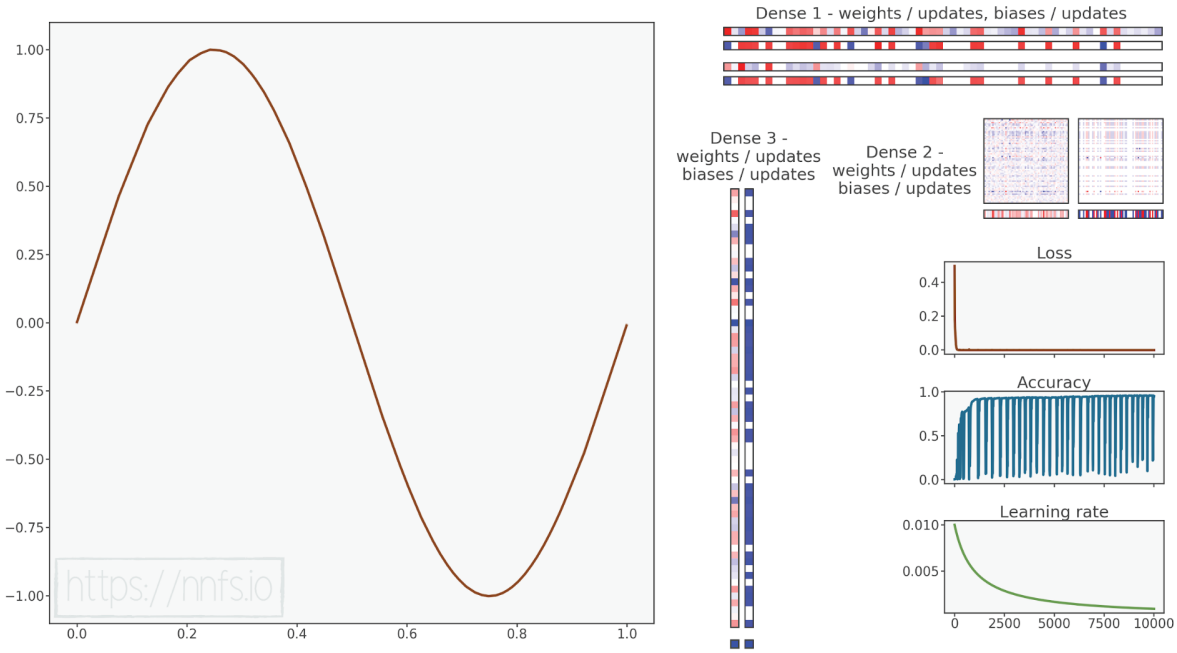


Fig 17.15: Model trained to fit the sine data after replacing weight initialization.

Anim 17.15: https://nnfs.io/mno

Another previously stuck model has trained very well this time, achieving very good accuracy.

```
optimizer = Optimizer_Adam(learning_rate=0.05, decay=1e-3)
```

```
>>>
epoch: 0, acc: 0.003, loss: 0.496 (data_loss: 0.496, reg_loss: 0.000), lr:
0.05
epoch: 100, acc: 0.016, loss: 0.008 (data_loss: 0.008, reg_loss: 0.000), lr:
0.04549590536851684
...
epoch: 9000, acc: 0.802, loss: 0.000 (data_loss: 0.000, reg_loss: 0.000),
lr: 0.005000500050005001
epoch: 9100, acc: 0.233, loss: 0.000 (data_loss: 0.000, reg_loss: 0.000),
lr: 0.004950985246063967
```

```
epoch: 9200, acc: 0.434, loss: 0.000 (data_loss: 0.000, reg_loss: 0.000),
lr: 0.004902441415825081
epoch: 9300, acc: 0.838, loss: 0.000 (data_loss: 0.000, reg_loss: 0.000),
lr: 0.0048548402757549285
epoch: 9400, acc: 0.309, loss: 0.000 (data_loss: 0.000, reg_loss: 0.000),
lr: 0.004808154630252909
epoch: 9500, acc: 0.253, loss: 0.000 (data_loss: 0.000, reg_loss: 0.000),
lr: 0.004762358319839985
epoch: 9600, acc: 0.795, loss: 0.000 (data_loss: 0.000, reg_loss: 0.000),
lr: 0.004717426172280404
epoch: 9700, acc: 0.802, loss: 0.000 (data_loss: 0.000, reg_loss: 0.000),
lr: 0.004673333956444528
epoch: 9800, acc: 0.141, loss: 0.000 (data_loss: 0.000, reg_loss: 0.000),
lr: 0.004630058338735069
epoch: 9900, acc: 0.221, loss: 0.000 (data_loss: 0.000, reg_loss: 0.000),
lr: 0.004587576841912101
epoch: 10000, acc: 0.631, loss: 0.000 (data_loss: 0.000, reg_loss: 0.000),
lr: 0.0045458678061641965
```

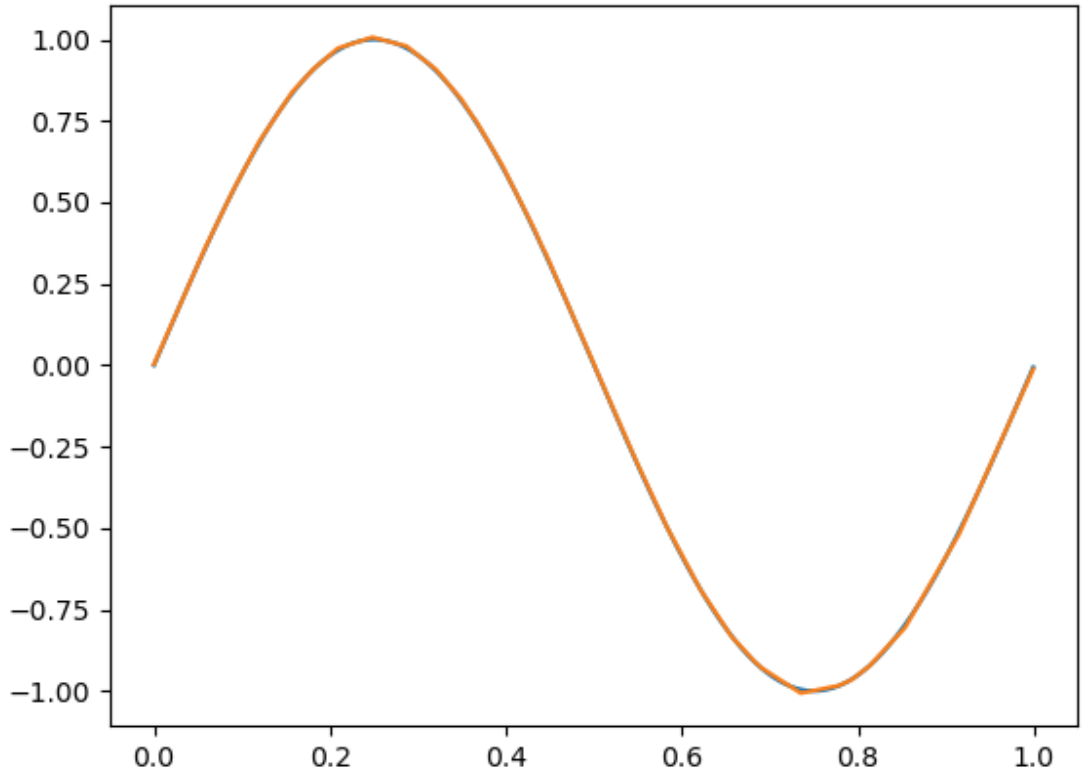


Fig 17.16: Model prediction - good fit to the data with different weight initialization.

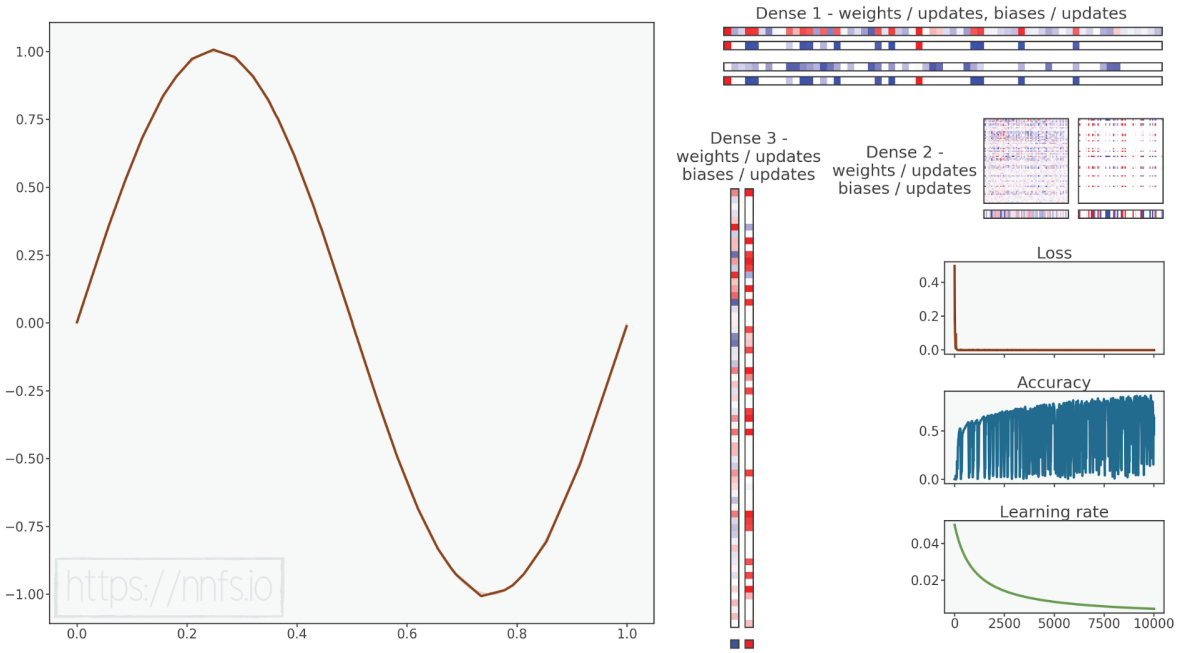


Fig 17.17: Model trained to fit the sine data after replacing weight initialization.

Anim 17.17: https://nnfs.io/nop

The "jumping" accuracy in the case of this set of the optimizer settings shows that the learning rate is way too big, but even then, the model learned the shape of the sine function considerably well.

```
optimizer = Optimizer_Adam(learning_rate=0.005, decay=1e-3)
```

```
>>>
epoch: 0, acc: 0.003, loss: 0.496 (data_loss: 0.496, reg_loss: 0.000), lr:
0.005
epoch: 100, acc: 0.017, loss: 0.048 (data_loss: 0.048, reg_loss: 0.000), lr:
0.004549590536851684
epoch: 200, acc: 0.242, loss: 0.001 (data_loss: 0.001, reg_loss: 0.000), lr:
0.004170141784820684
```

```
epoch: 300, acc: 0.786, loss: 0.000 (data_loss: 0.000, reg_loss: 0.000), lr:
0.003849114703618168
epoch: 400, acc: 0.885, loss: 0.000 (data_loss: 0.000, reg_loss: 0.000), lr:
0.0035739814152966403
...
epoch: 9900, acc: 0.982, loss: 0.000 (data_loss: 0.000, reg_loss: 0.000),
lr: 0.00045875768419121016
epoch: 10000, acc: 0.981, loss: 0.000 (data_loss: 0.000, reg_loss: 0.000),
lr: 0.00045458678061641964
```

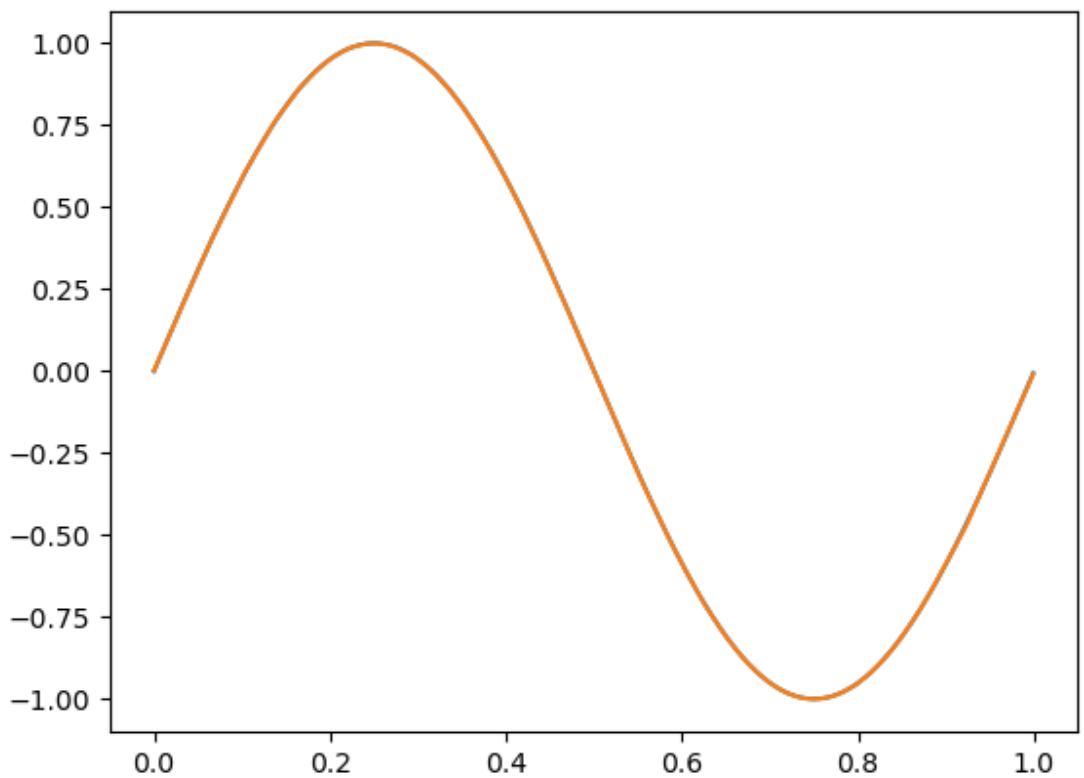


Fig 17.18: Model prediction - best fit to the data with different weight initialization.

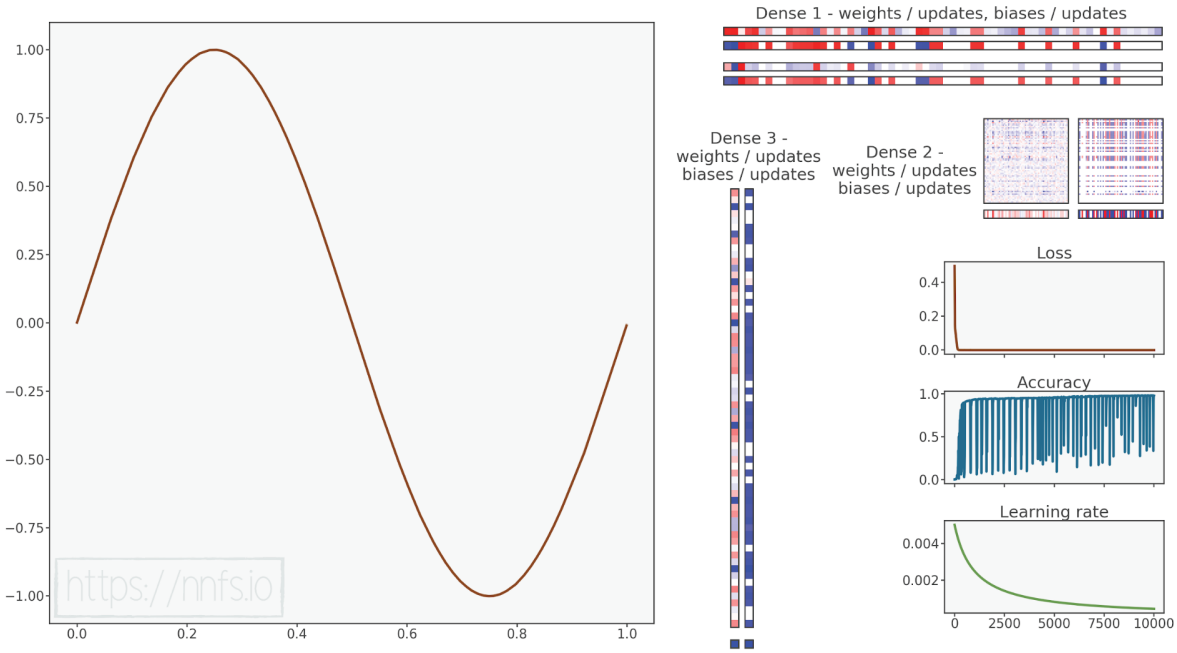


Fig 17.19: Model trained to best fit the sine data after replacing weight initialization.

Anim 17.19: https://nnfs.io/opq

These hyperparameters yielded the best results again, but not by much.

As we can see, this time, our model learned in all cases, using different learning rates, and did not get stuck if any of them. That's how much changing weight initialization can impact the training process.

Full code up to this point:

```
import numpy as np
import nnfs
from nnfs.datasets import sine_data

nnfs.init()


# Dense layer
class Layer_Dense:

    # Layer initialization
    def __init__(self, n_inputs, n_neurons,
                 weight_regularizer_l1=0, weight_regularizer_l2=0,
                 bias_regularizer_l1=0, bias_regularizer_l2=0):
        # Initialize weights and biases
        self.weights = 0.1 * np.random.randn(n_inputs, n_neurons)
        self.biases = np.zeros((1, n_neurons))
        # Set regularization strength
        self.weight_regularizer_l1 = weight_regularizer_l1
        self.weight_regularizer_l2 = weight_regularizer_l2
        self.bias_regularizer_l1 = bias_regularizer_l1
        self.bias_regularizer_l2 = bias_regularizer_l2

    # Forward pass
    def forward(self, inputs):
        # Remember input values
        self.inputs = inputs
        # Calculate output values from inputs, weights and biases
        self.output = np.dot(inputs, self.weights) + self.biases

    # Backward pass
    def backward(self, dvalues):
        # Gradients on parameters
        self.dweights = np.dot(self.inputs.T, dvalues)
        self.dbiases = np.sum(dvalues, axis=0, keepdims=True)
```

```
        # Gradients on regularization
        # L1 on weights
        if self.weight_regularizer_l1 > 0:
            dL1 = np.ones_like(self.weights)
            dL1[self.weights < 0] = -1
            self.dweights += self.weight_regularizer_l1 * dL1
        # L2 on weights
        if self.weight_regularizer_l2 > 0:
            self.dweights += 2 * self.weight_regularizer_l2 * \
                              self.weights
        # L1 on biases
        if self.bias_regularizer_l1 > 0:
            dL1 = np.ones_like(self.biases)
            dL1[self.biases < 0] = -1
            self.dbiases += self.bias_regularizer_l1 * dL1
        # L2 on biases
        if self.bias_regularizer_l2 > 0:
            self.dbiases += 2 * self.bias_regularizer_l2 * \
                             self.biases

        # Gradient on values
        self.dinputs = np.dot(dvalues, self.weights.T)


# Dropout
class Layer_Dropout:

    # Init
    def __init__(self, rate):
        # Store rate, we invert it as for example for dropout
        # of 0.1 we need success rate of 0.9
        self.rate = 1 - rate

    # Forward pass
    def forward(self, inputs):
        # Save input values
        self.inputs = inputs
        # Generate and save scaled mask
        self.binary_mask = np.random.binomial(1, self.rate,
                           size=inputs.shape) / self.rate
        # Apply mask to output values
        self.output = inputs * self.binary_mask

    # Backward pass
    def backward(self, dvalues):
        # Gradient on values
        self.dinputs = dvalues * self.binary_mask
```

```
# ReLU activation
class Activation_ReLU:

    # Forward pass
    def forward(self, inputs):
        # Remember input values
        self.inputs = inputs
        # Calculate output values from inputs
        self.output = np.maximum(0, inputs)

    # Backward pass
    def backward(self, dvalues):
        # Since we need to modify original variable,
        # let's make a copy of values first
        self.dinputs = dvalues.copy()

        # Zero gradient where input values were negative
        self.dinputs[self.inputs <= 0] = 0


# Softmax activation
class Activation_Softmax:

    # Forward pass
    def forward(self, inputs):
        # Remember input values
        self.inputs = inputs

        # Get unnormalized probabilities
        exp_values = np.exp(inputs - np.max(inputs, axis=1,
                                            keepdims=True))
        # Normalize them for each sample
        probabilities = exp_values / np.sum(exp_values, axis=1,
                                            keepdims=True)

        self.output = probabilities

    # Backward pass
    def backward(self, dvalues):

        # Create uninitialized array
        self.dinputs = np.empty_like(dvalues)

        # Enumerate outputs and gradients
        for index, (single_output, single_dvalues) in \
                enumerate(zip(self.output, dvalues)):
            # Flatten output array
            single_output = single_output.reshape(-1, 1)
```

```
            # Calculate Jacobian matrix of the output
            jacobian_matrix = np.diagflat(single_output) - \
                              np.dot(single_output, single_output.T)
            # Calculate sample-wise gradient
            # and add it to the array of sample gradients
            self.dinputs[index] = np.dot(jacobian_matrix,
                                         single_dvalues)


# Sigmoid activation
class Activation_Sigmoid:

    # Forward pass
    def forward(self, inputs):
        # Save input and calculate/save output
        # of the sigmoid function
        self.inputs = inputs
        self.output = 1 / (1 + np.exp(-inputs))

    # Backward pass
    def backward(self, dvalues):
        # Derivative - calculates from output of the sigmoid function
        self.dinputs = dvalues * (1 - self.output) * self.output


# Linear activation
class Activation_Linear:

    # Forward pass
    def forward(self, inputs):
        # Just remember values
        self.inputs = inputs
        self.output = inputs

    # Backward pass
    def backward(self, dvalues):
        # derivative is 1, 1 * dvalues = dvalues - the chain rule
        self.dinputs = dvalues.copy()


# SGD optimizer
class Optimizer_SGD:

    # Initialize optimizer - set settings,
    # learning rate of 1. is default for this optimizer
    def __init__(self, learning_rate=1., decay=0., momentum=0.):
        self.learning_rate = learning_rate
        self.current_learning_rate = learning_rate
        self.decay = decay
```

```
        self.iterations = 0
        self.momentum = momentum

    # Call once before any parameter updates
    def pre_update_params(self):
        if self.decay:
            self.current_learning_rate = self.learning_rate * \
                (1. / (1. + self.decay * self.iterations))

    # Update parameters
    def update_params(self, layer):

        # If we use momentum
        if self.momentum:

            # If layer does not contain momentum arrays, create them
            # filled with zeros
            if not hasattr(layer, 'weight_momentums'):
                layer.weight_momentums = np.zeros_like(layer.weights)
                # If there is no momentum array for weights
                # The array doesn't exist for biases yet either.
                layer.bias_momentums = np.zeros_like(layer.biases)

            # Build weight updates with momentum - take previous
            # updates multiplied by retain factor and update with
            # current gradients
            weight_updates = \
                self.momentum * layer.weight_momentums - \
                self.current_learning_rate * layer.dweights
            layer.weight_momentums = weight_updates

            # Build bias updates
            bias_updates = \
                self.momentum * layer.bias_momentums - \
                self.current_learning_rate * layer.dbiases
            layer.bias_momentums = bias_updates

        # Vanilla SGD updates (as before momentum update)
        else:
            weight_updates = -self.current_learning_rate * \
                             layer.dweights
            bias_updates = -self.current_learning_rate * \
                           layer.dbiases

        # Update weights and biases using either
        # vanilla or momentum updates
        layer.weights += weight_updates
        layer.biases += bias_updates
```

```
    # Call once after any parameter updates
    def post_update_params(self):
        self.iterations += 1


# Adagrad optimizer
class Optimizer_Adagrad:

    # Initialize optimizer - set settings
    def __init__(self, learning_rate=1., decay=0., epsilon=1e-7):
        self.learning_rate = learning_rate
        self.current_learning_rate = learning_rate
        self.decay = decay
        self.iterations = 0
        self.epsilon = epsilon

    # Call once before any parameter updates
    def pre_update_params(self):
        if self.decay:
            self.current_learning_rate = self.learning_rate * \
                (1. / (1. + self.decay * self.iterations))

    # Update parameters
    def update_params(self, layer):

        # If layer does not contain cache arrays,
        # create them filled with zeros
        if not hasattr(layer, 'weight_cache'):
            layer.weight_cache = np.zeros_like(layer.weights)
            layer.bias_cache = np.zeros_like(layer.biases)

        # Update cache with squared current gradients
        layer.weight_cache += layer.dweights**2
        layer.bias_cache += layer.dbiases**2

        # Vanilla SGD parameter update + normalization
        # with square rooted cache
        layer.weights += -self.current_learning_rate * \
                         layer.dweights / \
                         (np.sqrt(layer.weight_cache) + self.epsilon)
        layer.biases += -self.current_learning_rate * \
                        layer.dbiases / \
                        (np.sqrt(layer.bias_cache) + self.epsilon)

    # Call once after any parameter updates
    def post_update_params(self):
        self.iterations += 1
```

```python
# RMSprop optimizer
class Optimizer_RMSprop:

    # Initialize optimizer - set settings
    def __init__(self, learning_rate=0.001, decay=0., epsilon=1e-7,
                 rho=0.9):
        self.learning_rate = learning_rate
        self.current_learning_rate = learning_rate
        self.decay = decay
        self.iterations = 0
        self.epsilon = epsilon
        self.rho = rho

    # Call once before any parameter updates
    def pre_update_params(self):
        if self.decay:
            self.current_learning_rate = self.learning_rate * \
                (1. / (1. + self.decay * self.iterations))

    # Update parameters
    def update_params(self, layer):

        # If layer does not contain cache arrays,
        # create them filled with zeros
        if not hasattr(layer, 'weight_cache'):
            layer.weight_cache = np.zeros_like(layer.weights)
            layer.bias_cache = np.zeros_like(layer.biases)

        # Update cache with squared current gradients
        layer.weight_cache = self.rho * layer.weight_cache + \
            (1 - self.rho) * layer.dweights**2
        layer.bias_cache = self.rho * layer.bias_cache + \
            (1 - self.rho) * layer.dbiases**2

        # Vanilla SGD parameter update + normalization
        # with square rooted cache
        layer.weights += -self.current_learning_rate * \
                         layer.dweights / \
                         (np.sqrt(layer.weight_cache) + self.epsilon)
        layer.biases += -self.current_learning_rate * \
                        layer.dbiases / \
                        (np.sqrt(layer.bias_cache) + self.epsilon)

    # Call once after any parameter updates
    def post_update_params(self):
        self.iterations += 1
```

```
# Adam optimizer
class Optimizer_Adam:

    # Initialize optimizer - set settings
    def __init__(self, learning_rate=0.001, decay=0., epsilon=1e-7,
                 beta_1=0.9, beta_2=0.999):
        self.learning_rate = learning_rate
        self.current_learning_rate = learning_rate
        self.decay = decay
        self.iterations = 0
        self.epsilon = epsilon
        self.beta_1 = beta_1
        self.beta_2 = beta_2

    # Call once before any parameter updates
    def pre_update_params(self):
        if self.decay:
            self.current_learning_rate = self.learning_rate * \
                (1. / (1. + self.decay * self.iterations))

    # Update parameters
    def update_params(self, layer):

        # If layer does not contain cache arrays,
        # create them filled with zeros
        if not hasattr(layer, 'weight_cache'):
            layer.weight_momentums = np.zeros_like(layer.weights)
            layer.weight_cache = np.zeros_like(layer.weights)
            layer.bias_momentums = np.zeros_like(layer.biases)
            layer.bias_cache = np.zeros_like(layer.biases)

        # Update momentum  with current gradients
        layer.weight_momentums = self.beta_1 * \
                                 layer.weight_momentums + \
                                 (1 - self.beta_1) * layer.dweights
        layer.bias_momentums = self.beta_1 * \
                               layer.bias_momentums + \
                               (1 - self.beta_1) * layer.dbiases
        # Get corrected momentum
        # self.iteration is 0 at first pass
        # and we need to start with 1 here
        weight_momentums_corrected = layer.weight_momentums / \
            (1 - self.beta_1 ** (self.iterations + 1))
        bias_momentums_corrected = layer.bias_momentums / \
            (1 - self.beta_1 ** (self.iterations + 1))
        # Update cache with squared current gradients
        layer.weight_cache = self.beta_2 * layer.weight_cache + \
            (1 - self.beta_2) * layer.dweights**2
```

```
        layer.bias_cache = self.beta_2 * layer.bias_cache + \
            (1 - self.beta_2) * layer.dbiases**2
        # Get corrected cache
        weight_cache_corrected = layer.weight_cache / \
            (1 - self.beta_2 ** (self.iterations + 1))
        bias_cache_corrected = layer.bias_cache / \
            (1 - self.beta_2 ** (self.iterations + 1))

        # Vanilla SGD parameter update + normalization
        # with square rooted cache
        layer.weights += -self.current_learning_rate * \
                         weight_momentums_corrected / \
                         (np.sqrt(weight_cache_corrected) +
                             self.epsilon)
        layer.biases += -self.current_learning_rate * \
                         bias_momentums_corrected / \
                         (np.sqrt(bias_cache_corrected) +
                             self.epsilon)

    # Call once after any parameter updates
    def post_update_params(self):
        self.iterations += 1


# Common loss class
class Loss:

    # Regularization loss calculation
    def regularization_loss(self, layer):

        # 0 by default
        regularization_loss = 0

        # L1 regularization - weights
        # calculate only when factor greater than 0
        if layer.weight_regularizer_l1 > 0:
            regularization_loss += layer.weight_regularizer_l1 * \
                                   np.sum(np.abs(layer.weights))

        # L2 regularization - weights
        if layer.weight_regularizer_l2 > 0:
            regularization_loss += layer.weight_regularizer_l2 * \
                                   np.sum(layer.weights * \
                                          layer.weights)
```

```
        # L1 regularization - biases
        # calculate only when factor greater than 0
        if layer.bias_regularizer_l1 > 0:
            regularization_loss += layer.bias_regularizer_l1 * \
                                   np.sum(np.abs(layer.biases))

        # L2 regularization - biases
        if layer.bias_regularizer_l2 > 0:
            regularization_loss += layer.bias_regularizer_l2 * \
                                   np.sum(layer.biases * \
                                          layer.biases)

        return regularization_loss

    # Calculates the data and regularization losses
    # given model output and ground truth values
    def calculate(self, output, y):

        # Calculate sample losses
        sample_losses = self.forward(output, y)

        # Calculate mean loss
        data_loss = np.mean(sample_losses)

        # Return loss
        return data_loss


# Cross-entropy loss
class Loss_CategoricalCrossentropy(Loss):

    # Forward pass
    def forward(self, y_pred, y_true):

        # Number of samples in a batch
        samples = len(y_pred)

        # Clip data to prevent division by 0
        # Clip both sides to not drag mean towards any value
        y_pred_clipped = np.clip(y_pred, 1e-7, 1 - 1e-7)

        # Probabilities for target values -
        # only if categorical labels
        if len(y_true.shape) == 1:
            correct_confidences = y_pred_clipped[
                range(samples),
                y_true
            ]
```

```
        # Mask values - only for one-hot encoded labels
        elif len(y_true.shape) == 2:
            correct_confidences = np.sum(
                y_pred_clipped * y_true,
                axis=1
            )

        # Losses
        negative_log_likelihoods = -np.log(correct_confidences)
        return negative_log_likelihoods

    # Backward pass
    def backward(self, dvalues, y_true):

        # Number of samples
        samples = len(dvalues)
        # Number of labels in every sample
        # We'll use the first sample to count them
        labels = len(dvalues[0])

        # If labels are sparse, turn them into one-hot vector
        if len(y_true.shape) == 1:
            y_true = np.eye(labels)[y_true]

        # Calculate gradient
        self.dinputs = -y_true / dvalues
        # Normalize gradient
        self.dinputs = self.dinputs / samples


# Softmax classifier - combined Softmax activation
# and cross-entropy loss for faster backward step
class Activation_Softmax_Loss_CategoricalCrossentropy():

    # Creates activation and loss function objects
    def __init__(self):
        self.activation = Activation_Softmax()
        self.loss = Loss_CategoricalCrossentropy()

    # Forward pass
    def forward(self, inputs, y_true):
        # Output layer's activation function
        self.activation.forward(inputs)
        # Set the output
        self.output = self.activation.output
        # Calculate and return loss value
        return self.loss.calculate(self.output, y_true)
```

```
    # Backward pass
    def backward(self, dvalues, y_true):

        # Number of samples
        samples = len(dvalues)

        # If labels are one-hot encoded,
        # turn them into discrete values
        if len(y_true.shape) == 2:
            y_true = np.argmax(y_true, axis=1)

        # Copy so we can safely modify
        self.dinputs = dvalues.copy()
        # Calculate gradient
        self.dinputs[range(samples), y_true] -= 1
        # Normalize gradient
        self.dinputs = self.dinputs / samples


# Binary cross-entropy loss
class Loss_BinaryCrossentropy(Loss):

    # Forward pass
    def forward(self, y_pred, y_true):

        # Clip data to prevent division by 0
        # Clip both sides to not drag mean towards any value
        y_pred_clipped = np.clip(y_pred, 1e-7, 1 - 1e-7)

        # Calculate sample-wise loss
        sample_losses = -(y_true * np.log(y_pred_clipped) +
                          (1 - y_true) * np.log(1 - y_pred_clipped))
        sample_losses = np.mean(sample_losses, axis=-1)

        # Return losses
        return sample_losses

    # Backward pass
    def backward(self, dvalues, y_true):

        # Number of samples
        samples = len(dvalues)
        # Number of outputs in every sample
        # We'll use the first sample to count them
        outputs = len(dvalues[0])

        # Clip data to prevent division by 0
        # Clip both sides to not drag mean towards any value
        clipped_dvalues = np.clip(dvalues, 1e-7, 1 - 1e-7)
```

```
        # Calculate gradient
        self.dinputs = -(y_true / clipped_dvalues -
                         (1 - y_true) / (1 - clipped_dvalues)) / outputs

        # Normalize gradient
        self.dinputs = self.dinputs / samples


# Mean Squared Error loss
class Loss_MeanSquaredError(Loss):  # L2 loss

    # Forward pass
    def forward(self, y_pred, y_true):

        # Calculate loss
        sample_losses = np.mean((y_true - y_pred)**2, axis=-1)

        # Return losses
        return sample_losses

    # Backward pass
    def backward(self, dvalues, y_true):

        # Number of samples
        samples = len(dvalues)
        # Number of outputs in every sample
        # We'll use the first sample to count them
        outputs = len(dvalues[0])

        # Gradient on values
        self.dinputs = -2 * (y_true - dvalues) / outputs
        # Normalize gradient
        self.dinputs = self.dinputs / samples


# Mean Absolute Error loss
class Loss_MeanAbsoluteError(Loss):  # L1 loss

    # Forward pass
    def forward(self, y_pred, y_true):

        # Calculate loss
        sample_losses = np.mean(np.abs(y_true - y_pred), axis=-1)

        # Return losses
        return sample_losses
```

```
    # Backward pass
    def backward(self, dvalues, y_true):

        # Number of samples
        samples = len(dvalues)
        # Number of outputs in every sample
        # We'll use the first sample to count them
        outputs = len(dvalues[0])

        # Calculate gradient
        self.dinputs = np.sign(y_true - dvalues) / outputs
        # Normalize gradient
        self.dinputs = self.dinputs / samples


# Create dataset
X, y = sine_data()

# Create Dense layer with 1 input feature and 64 output values
dense1 = Layer_Dense(1, 64)

# Create ReLU activation (to be used with Dense layer):
activation1 = Activation_ReLU()

# Create second Dense layer with 64 input features (as we take output
# of previous layer here) and 64 output values
dense2 = Layer_Dense(64, 64)

# Create ReLU activation (to be used with Dense layer):
activation2 = Activation_ReLU()

# Create third Dense layer with 64 input features (as we take output
# of previous layer here) and 1 output value
dense3 = Layer_Dense(64, 1)

# Create Linear activation:
activation3 = Activation_Linear()

# Create loss function
loss_function = Loss_MeanSquaredError()

# Create optimizer
optimizer = Optimizer_Adam(learning_rate=0.005, decay=1e-3)
```

```
# Accuracy precision for accuracy calculation
# There are no really accuracy factor for regression problem,
# but we can simulate/approximate it. We'll calculate it by checking
# how many values have a difference to their ground truth equivalent
# less than given precision
# We'll calculate this precision as a fraction of standard deviation
# of all the ground truth values
accuracy_precision = np.std(y) / 250

# Train in loop
for epoch in range(10001):

    # Perform a forward pass of our training data through this layer
    dense1.forward(X)

    # Perform a forward pass through activation function
    # takes the output of first dense layer here
    activation1.forward(dense1.output)

    # Perform a forward pass through second Dense layer
    # takes outputs of activation function
    # of first layer as inputs
    dense2.forward(activation1.output)

    # Perform a forward pass through activation function
    # takes the output of second dense layer here
    activation2.forward(dense2.output)

    # Perform a forward pass through third Dense layer
    # takes outputs of activation function of second layer as inputs
    dense3.forward(activation2.output)

    # Perform a forward pass through activation function
    # takes the output of third dense layer here
    activation3.forward(dense3.output)

    # Calculate the data loss
    data_loss = loss_function.calculate(activation3.output, y)

    # Calculate regularization penalty
    regularization_loss = \
        loss_function.regularization_loss(dense1) + \
        loss_function.regularization_loss(dense2) + \
        loss_function.regularization_loss(dense3)

    # Calculate overall loss
    loss = data_loss + regularization_loss
```

```
    # Calculate accuracy from output of activation2 and targets
    # To calculate it we're taking absolute difference between
    # predictions and ground truth values and compare if differences
    # are lower than given precision value
    predictions = activation3.output
    accuracy = np.mean(np.absolute(predictions - y) <
                       accuracy_precision)

    if not epoch % 100:
        print(f'epoch: {epoch}, ' +
              f'acc: {accuracy:.3f}, ' +
              f'loss: {loss:.3f} (' +
              f'data_loss: {data_loss:.3f}, ' +
              f'reg_loss: {regularization_loss:.3f}), ' +
              f'lr: {optimizer.current_learning_rate}')

    # Backward pass
    loss_function.backward(activation3.output, y)
    activation3.backward(loss_function.dinputs)
    dense3.backward(activation3.dinputs)
    activation2.backward(dense3.dinputs)
    dense2.backward(activation2.dinputs)
    activation1.backward(dense2.dinputs)
    dense1.backward(activation1.dinputs)

    # Update weights and biases
    optimizer.pre_update_params()
    optimizer.update_params(dense1)
    optimizer.update_params(dense2)
    optimizer.update_params(dense3)
    optimizer.post_update_params()


import matplotlib.pyplot as plt

X_test, y_test = sine_data()

dense1.forward(X_test)
activation1.forward(dense1.output)
dense2.forward(activation1.output)
activation2.forward(dense2.output)
dense3.forward(activation2.output)
activation3.forward(dense3.output)

plt.plot(X_test, y_test)
plt.plot(X_test, activation3.output)
plt.show()
```

Supplementary Material: https://nnfs.io/ch17
Chapter code, further resources, and errata for this chapter.

Chapter 18

Model Object

We built a model that can perform the forward pass, backward pass, and ancillary tasks like measuring accuracy. We have built all this by writing a fair bit of code and making modifications in some decently-sized blocks of code. It's beginning to make more sense to make our model an object itself, especially since we will want to do things like save and load this object to use for future prediction tasks. We will also use this object to cut down on some of the more common lines of code, making it easier to work with our current code base and build new models. To do this model object conversion, we'll use the last model we were working on, the regression model with sine data:

```
from nnfs.datasets import sine_data
X, y = sine_data()
```

Once we have the data, our first step for the model class is to add in the various layers we want. Thus, we can begin our model class by doing:

```
# Model class
class Model:

    def __init__(self):
        # Create a list of network objects
        self.layers = []

    # Add objects to the model
    def add(self, layer):
        self.layers.append(layer)
```

This allows us to use the `add` method of the model object to add layers. This alone will help with legibility considerably. Let's add some layers:

```
# Instantiate the model
model = Model()

# Add layers
model.add(Layer_Dense(1, 64))
model.add(Activation_ReLU())
model.add(Layer_Dense(64, 64))
model.add(Activation_ReLU())
model.add(Layer_Dense(64, 1))
model.add(Activation_Linear())
```

We can also query this model now:

```
print(model.layers)

>>>
[<__main__.Layer_Dense object at 0x0000015E9D504BC8>, <__main__.
Activation_ReLU object at 0x0000015E9D504C48>, <__main__.Layer_Dense
object at 0x0000015E9D504C88>, <__main__.Activation_ReLU object at
0x0000015E9D504CC8>, <__main__.Layer_Dense object at 0x0000015E9D504D08>,
<__main__.Activation_Linear object at 0x0000015E9D504D88>]
```

Besides adding layers, we also want to set a loss function and optimizer for the model. To do this, we'll create a method called `set`:

```
    # Set loss and optimizer
    def set(self, *, loss, optimizer):
        self.loss = loss
        self.optimizer = optimizer
```

The use of the asterisk in the parameter definitions notes that the subsequent parameters (*loss* and *optimizer* in this case) are keyword arguments. Since they have no default value assigned, they are required keyword arguments, which means that they have to be passed by names and values, making code more legible.

Now we can add a call to this method into our newly-created model object, and pass the loss and optimizer objects:

```
# Create dataset
X, y = sine_data()

# Instantiate the model
model = Model()

# Add layers
model.add(Layer_Dense(1, 64))
model.add(Activation_ReLU())
model.add(Layer_Dense(64, 64))
model.add(Activation_ReLU())
model.add(Layer_Dense(64, 1))
model.add(Activation_Linear())

# Set loss and optimizer objects
model.set(
    loss=Loss_MeanSquaredError(),
    optimizer=Optimizer_Adam(learning_rate=0.005, decay=1e-3),
)
```

After we've set our model's layers, loss function, and optimizer, the next step is to train, so we'll add a train method. For now, we'll make it a placeholder and fill it in soon:

```
    # Train the model
    def train(self, X, y, *, epochs=1, print_every=1):

        # Main training loop
        for epoch in range(1, epochs+1):

            # Temporary
            pass
```

We can then add a call to the train method in the model definition. We'll pass the training data, the number of epochs (10000, as we've used so far), and an indicator of how often to print a training summary. We do not need or want to print it every step, so we'll make it configurable:

```
# Create dataset
X, y = sine_data()

# Instantiate the model
model = Model()

# Add layers
model.add(Layer_Dense(1, 64))
model.add(Activation_ReLU())
model.add(Layer_Dense(64, 64))
model.add(Activation_ReLU())
model.add(Layer_Dense(64, 1))
model.add(Activation_Linear())

# Set loss and optimizer objects
model.set(
    loss=Loss_MeanSquaredError(),
    optimizer=Optimizer_Adam(learning_rate=0.005, decay=1e-3),
)

model.train(X, y, epochs=10000, print_every=100)
```

To actually train, we need to perform a forward pass. Performing this forward pass in the object is slightly more complicated because we want to do this in a loop over the layers, and we need to know the previous layer's output to pass data properly. One issue with querying the previous layer is that the first layer doesn't have a "previous" layer. The first layer that we're defining is the first **hidden** layer. One option we have then is to create an "input layer." This is considered a layer in a neural network but doesn't have weights and biases associated with it. The input layer only contains the training data, and we'll only use it as a "previous" layer to the first layer during the iteration of the layers in a loop. We'll create a new class and call it in similarly as `Layer_Dense` class — `Layer_Input`:

```
# Input "layer"
class Layer_Input:

    # Forward pass
    def forward(self, inputs):
        self.output = inputs
```

The `forward` method sets training samples as `self.output`. This property is common with other layers. There's no need for a backward method here since we'll never use it. It might seem silly right now to even have this class, but it should hopefully become clear how we're going to use this shortly. The next thing we're going to do is set the previous and next layer properties for each of the model's layers. We'll create a method called `finalize` in the `Model` class:

```
    # Finalize the model
    def finalize(self):

        # Create and set the input layer
        self.input_layer = Layer_Input()

        # Count all the objects
        layer_count = len(self.layers)

        # Iterate the objects
        for i in range(layer_count):

            # If it's the first layer,
            # the previous layer object is the input layer
            if i == 0:
                self.layers[i].prev = self.input_layer
                self.layers[i].next = self.layers[i+1]

            # All layers except for the first and the last
            elif i < layer_count - 1:
                self.layers[i].prev = self.layers[i-1]
                self.layers[i].next = self.layers[i+1]

            # The last layer - the next object is the loss
            else:
                self.layers[i].prev = self.layers[i-1]
                self.layers[i].next = self.loss
```

This code creates an input layer and sets `next` and `prev` references for each layer contained within the `self.layers` list of a model object. We wanted to create the `Layer_Input` class to set the `prev` property of the first hidden layer in a loop since we are going to call all of the layers in a uniform way. The `next` layer for the final layer will be the loss, which we already have created.

Now that we have the necessary layer information for our model object to perform a forward pass, let's add a forward method. We will use this forward method both for when we train and later when we just want to predict, which is also called **model inference**. Continuing the code within the `Model` class:

```
# Forward pass
class Model:
    ...
    # Performs forward pass
    def forward(self, X):

        # Call forward method on the input layer
        # this will set the output property that
        # the first layer in "prev" object is expecting
        self.input_layer.forward(X)

        # Call forward method of every object in a chain
        # Pass output of the previous object as a parameter
        for layer in self.layers:
            layer.forward(layer.prev.output)

        # "layer" is now the last object from the list,
        # return its output
        return layer.output
```

In this case, we take in *X* (input data), then simply pass this data through the `input_layer` in the `Model` object, which creates an `output` attribute in this object. From here, we begin iterating over the `self.layers`, the layers starting with the first hidden layer. We perform a forward pass on the `layer.prev.output`, the output data of the previous layer, for each layer. For the first hidden layer, the `layer.prev` is `self.input_layer`. The `output` attribute is created for each layer when we call the `forward` method, which is then used as input to a `forward` method call on the next layer. Once we've iterated over all of the layers, we return the final layer's output. That's a forward pass, and now let's go ahead and add this forward pass method call to the `train` method in the `Model` class:

```
# Forward pass
class Model:
    ...
    # Train the model
    def train(self, X, y, *, epochs=1, print_every=1):

        # Main training loop
        for epoch in range(1, epochs+1):

            # Perform the forward pass
            output = self.forward(X)

            # Temporary
            print(output)
            exit()
```

Full `Model` class up to this point:

```
# Model class
class Model:

    def __init__(self):
        # Create a list of network objects
        self.layers = []

    # Add objects to the model
    def add(self, layer):
        self.layers.append(layer)

    # Set loss and optimizer
    def set(self, *, loss, optimizer):
        self.loss = loss
        self.optimizer = optimizer

    # Finalize the model
    def finalize(self):

        # Create and set the input layer
        self.input_layer = Layer_Input()

        # Count all the objects
        layer_count = len(self.layers)

        # Iterate the objects
        for i in range(layer_count):

            # If it's the first layer,
            # the previous layer object is the input layer
            if i == 0:
                self.layers[i].prev = self.input_layer
                self.layers[i].next = self.layers[i+1]

            # All layers except for the first and the last
            elif i < layer_count - 1:
                self.layers[i].prev = self.layers[i-1]
                self.layers[i].next = self.layers[i+1]

            # The last layer - the next object is the loss
            else:
                self.layers[i].prev = self.layers[i-1]
                self.layers[i].next = self.loss
```

```
    # Train the model
    def train(self, X, y, *, epochs=1, print_every=1):

        # Main training loop
        for epoch in range(1, epochs+1):

            # Perform the forward pass
            output = self.forward(X)

            # Temporary
            print(output)
            exit()

    # Performs forward pass
    def forward(self, X):

        # Call forward method on the input layer
        # this will set the output property that
        # the first layer in "prev" object is expecting
        self.input_layer.forward(X)

        # Call forward method of every object in a chain
        # Pass output of the previous object as a parameter
        for layer in self.layers:
            layer.forward(layer.prev.output)

        # "layer" is now the last object from the list,
        # return its output
        return layer.output
```

Finally, we can add in the `finalize` method call to the main code (recall this method makes, among other things, the model's layers aware of their previous and next layers).

```
# Create dataset
X, y = sine_data()

# Instantiate the model
model = Model()

# Add layers
model.add(Layer_Dense(1, 64))
model.add(Activation_ReLU())
model.add(Layer_Dense(64, 64))
model.add(Activation_ReLU())
model.add(Layer_Dense(64, 1))
model.add(Activation_Linear())
```

```
# Set loss and optimizer objects
model.set(
    loss=Loss_MeanSquaredError(),
    optimizer=Optimizer_Adam(learning_rate=0.005, decay=1e-3),
)

# Finalize the model
model.finalize()

# Train the model
model.train(X, y, epochs=10000, print_every=100)
```

Running this:

```
>>>
[[ 0.00000000e+00]
 [-1.13209149e-08]
 [-2.26418297e-08]
...
 [-1.12869511e-05]
 [-1.12982725e-05]
 [-1.13095930e-05]]
```

At this point, we've covered the forward pass of our model in the `Model` class. We still need to calculate loss and accuracy along with doing backpropagation. Before doing this, we need to know which layers are "trainable," which means layers with weights and biases that we can tweak. To do this, we need to check if the layer has either a `weights` or `biases` attribute. We can check this with the following code:

```
# If layer contains an attribute called "weights",
# it's a trainable layer -
# add it to the list of trainable layers
# We don't need to check for biases -
# checking for weights is enough
if hasattr(self.layers[i], 'weights'):
    self.trainable_layers.append(self.layers[i])
```

Where `i` is the index for the layer in the list of layers. We'll put this code into the `finalize` method. The full code for that method so far:

```
# Finalize the model
def finalize(self):

    # Create and set the input layer
    self.input_layer = Layer_Input()

    # Count all the objects
    layer_count = len(self.layers)

    # Initialize a list containing trainable layers:
    self.trainable_layers = []

    # Iterate the objects
    for i in range(layer_count):

        # If it's the first layer,
        # the previous layer object is the input layer
        if i == 0:
            self.layers[i].prev = self.input_layer
            self.layers[i].next = self.layers[i+1]

        # All layers except for the first and the last
        elif i < layer_count - 1:
            self.layers[i].prev = self.layers[i-1]
            self.layers[i].next = self.layers[i+1]

        # The last layer - the next object is the loss
        # Also let's save aside the reference to the last object
        # whose output is the model's output
        else:
            self.layers[i].prev = self.layers[i-1]
            self.layers[i].next = self.loss

        # If layer contains an attribute called "weights",
        # it's a trainable layer -
        # add it to the list of trainable layers
        # We don't need to check for biases -
        # checking for weights is enough
        if hasattr(self.layers[i], 'weights'):
            self.trainable_layers.append(self.layers[i])
```

Next, we'll modify the common `Loss` class to contain the following:

```
# Common loss class
class Loss:
    ...
    # Set/remember trainable layers
    def remember_trainable_layers(self, trainable_layers):
        self.trainable_layers = trainable_layers

    # Calculates the data and regularization losses
    # given model output and ground truth values
    def calculate(self, output, y):

        # Calculate sample losses
        sample_losses = self.forward(output, y)

        # Calculate mean loss
        data_loss = np.mean(sample_losses)

        # Return the data and regularization losses
        return data_loss, self.regularization_loss()
```

The `remember_trainable_layers` method in the common `Loss` class "tells" the loss object which layers in the `Model` object are trainable. The `calculate` method was modified to also return the `self.regularization_loss()` during a single call. The `regularization_loss` method currently requires a layer object, but with the `self.trainable_layers` property set in `remember_trainable_layers`, method we can now iterate over the trainable layers to compute regularization loss for the entire model, rather than one layer at a time:

```
class Loss:
    ...
    # Regularization loss calculation
    def regularization_loss(self):

        # 0 by default
        regularization_loss = 0

        # Calculate regularization loss
        # iterate all trainable layers
        for layer in self.trainable_layers:

            # L1 regularization - weights
            # calculate only when factor greater than 0
            if layer.weight_regularizer_l1 > 0:
                regularization_loss += layer.weight_regularizer_l1 * \
                                       np.sum(np.abs(layer.weights))
```

```
            # L2 regularization - weights
            if layer.weight_regularizer_l2 > 0:
                regularization_loss += layer.weight_regularizer_l2 * \
                                       np.sum(layer.weights *
                                              layer.weights)

            # L1 regularization - biases
            # calculate only when factor greater than 0
            if layer.bias_regularizer_l1 > 0:
                regularization_loss += layer.bias_regularizer_l1 * \
                                       np.sum(np.abs(layer.biases))

            # L2 regularization - biases
            if layer.bias_regularizer_l2 > 0:
                regularization_loss += layer.bias_regularizer_l2 * \
                                       np.sum(layer.biases *
                                              layer.biases)

        return regularization_loss
```

For calculating accuracy, we need predictions. So far, predicting has required different code depending on the type of model. For a softmax classifier, we do a `np.argmax()`, but for regression, the prediction is the direct output because of the linear activation function being used in an output layer. Ideally, we'd have a prediction method that would choose the appropriate method for our model. To do this, we'll add a `predictions` method to each activation function class:

```
# Softmax activation
class Activation_Softmax:
    ...
    # Calculate predictions for outputs
    def predictions(self, outputs):
        return np.argmax(outputs, axis=1)


# Sigmoid activation
class Activation_Sigmoid:
    ...
    # Calculate predictions for outputs
    def predictions(self, outputs):
        return (outputs > 0.5) * 1
```

```
# Linear activation
class Activation_Linear:
    ...
    # Calculate predictions for outputs
    def predictions(self, outputs):
        return outputs
```

All the computations made inside the `predictions` functions are the same as those performed with appropriate models in previous chapters. While we have no plans for using the ReLU activation function for an output layer's activation function, we'll include it here for completeness:

```
# ReLU activation
class Activation_ReLU:
    ...
    # Calculate predictions for outputs
    def predictions(self, outputs):
        return outputs
```

We still need to set a reference to the activation function for the final layer in the `Model` object. We can later call the `predictions` method, which will return predictions calculated from the outputs. We'll set this in the `Model` class' `finalize` method:

```
# Model class
class Model:
    ...
    def finalize(self):
        ...
            # The last layer - the next object is the loss
            # Also let's save aside the reference to the last object
            # whose output is the model's output
            else:
                self.layers[i].prev = self.layers[i-1]
                self.layers[i].next = self.loss
                self.output_layer_activation = self.layers[i]
```

Just like the different prediction methods, we also calculate accuracy in different ways. We're going to implement this in a way similar to the specific loss class' objects implementation — we'll create specific accuracy classes and their objects, which we'll associate with models.

First, we'll write a common `Accuracy` class containing (for now) just a single method, `calculate`, returning an accuracy calculated from comparison results. We've already added a call to the `self.compare` method that does not exist yet, but we'll create it soon in other classes that will inherit from this `Accuracy` class. For now, it's enough to know that it will return a list of *True* and *False* values, indicating if a prediction matches the ground-truth value. Next, we calculate the mean value (which treats *True* as *1* and *False* as *0*) and return it as an accuracy. The

code:

```
# Common accuracy class
class Accuracy:

    # Calculates an accuracy
    # given predictions and ground truth values
    def calculate(self, predictions, y):

        # Get comparison results
        comparisons = self.compare(predictions, y)

        # Calculate an accuracy
        accuracy = np.mean(comparisons)

        # Return accuracy
        return accuracy
```

Next, we can work with this common `Accuracy` class, inheriting from it, then building further for specific types of models. In general, each of these classes will contain two methods: `init` (not to be confused with a Python class' `__init__` method) for initialization from inside the model object and `compare` for performing comparison calculations. For regression, the `init` method will calculate an accuracy precision, the same as we have written previously for the regression model, and have been running before the training loop. The `compare` method will contain the actual comparison code we have implemented in the training loop itself, which uses `self.precision`. Note that initialization won't recalculate precision unless forced to do so by setting the *`reinit`* parameter to `True`. This allows for multiple use-cases, including setting `self.precision` independently, calling `init` whenever needed (e.g., from outside of the model during its creation), and even calling it multiple times (which will become handy soon):

```
# Accuracy calculation for regression model
class Accuracy_Regression(Accuracy):

    def __init__(self):
        # Create precision property
        self.precision = None

    # Calculates precision value
    # based on passed-in ground truth
    def init(self, y, reinit=False):
        if self.precision is None or reinit:
            self.precision = np.std(y) / 250

    # Compares predictions to the ground truth values
    def compare(self, predictions, y):
        return np.absolute(predictions - y) < self.precision
```

We can then set the accuracy object from within the `set` method in our `Model` class the same way as the loss and optimizer currently:

```
# Model class
class Model:
    ...
    # Set loss, optimizer and accuracy
    def set(self, *, loss, optimizer, accuracy):
        self.loss = loss
        self.optimizer = optimizer
        self.accuracy = accuracy
```

Then we can finally add the loss and accuracy calculations to our model right after the completed forward pass' code. Note that we also initialize the accuracy with `self.accuracy.init(y)` at the beginning of the `train` method, which can be called multiple times — as noted earlier. In the case of regression accuracy, this will invoke a precision calculation a single time during the first call. The code of the `train` method with implemented loss and accuracy calculations:

```
# Model class
class Model:
    ...
    # Train the model
    def train(self, X, y, *, epochs=1, print_every=1):

        # Initialize accuracy object
        self.accuracy.init(y)

        # Main training loop
        for epoch in range(1, epochs+1):

            # Perform the forward pass
            output = self.forward(X)

            # Calculate loss
            data_loss, regularization_loss = \
                self.loss.calculate(output, y)
            loss = data_loss + regularization_loss

            # Get predictions and calculate an accuracy
            predictions = self.output_layer_activation.predictions(
                              output)
            accuracy = self.accuracy.calculate(predictions, y)
```

Finally, we'll add a call to the previously created method `remember_trainable_layers` with the `Loss` class' object, which we'll do in the `finalize` method (`self.loss.remember_trainable_layers(self.trainable_layers)`). The full model class code so far:

```
# Model class
class Model:

    def __init__(self):
        # Create a list of network objects
        self.layers = []

    # Add objects to the model
    def add(self, layer):
        self.layers.append(layer)

    # Set loss, optimizer and accuracy
    def set(self, *, loss, optimizer, accuracy):
        self.loss = loss
        self.optimizer = optimizer
        self.accuracy = accuracy

    # Finalize the model
    def finalize(self):

        # Create and set the input layer
        self.input_layer = Layer_Input()

        # Count all the objects
        layer_count = len(self.layers)

        # Initialize a list containing trainable layers:
        self.trainable_layers = []

        # Iterate the objects
        for i in range(layer_count):

            # If it's the first layer,
            # the previous layer object is the input layer
            if i == 0:
                self.layers[i].prev = self.input_layer
                self.layers[i].next = self.layers[i+1]

            # All layers except for the first and the last
            elif i < layer_count - 1:
                self.layers[i].prev = self.layers[i-1]
                self.layers[i].next = self.layers[i+1]
```

```
        # The last layer - the next object is the loss
        # Also let's save aside the reference to the last object
        # whose output is the model's output
        else:
            self.layers[i].prev = self.layers[i-1]
            self.layers[i].next = self.loss
            self.output_layer_activation = self.layers[i]

        # If layer contains an attribute called "weights",
        # it's a trainable layer -
        # add it to the list of trainable layers
        # We don't need to check for biases -
        # checking for weights is enough
        if hasattr(self.layers[i], 'weights'):
            self.trainable_layers.append(self.layers[i])

    # Update loss object with trainable layers
    self.loss.remember_trainable_layers(
        self.trainable_layers
    )

# Train the model
def train(self, X, y, *, epochs=1, print_every=1):

    # Initialize accuracy object
    self.accuracy.init(y)

    # Main training loop
    for epoch in range(1, epochs+1):

        # Perform the forward pass
        output = self.forward(X)

        # Calculate loss
        data_loss, regularization_loss = \
            self.loss.calculate(output, y)
        loss = data_loss + regularization_loss

        # Get predictions and calculate an accuracy
        predictions = self.output_layer_activation.predictions(
                          output)
        accuracy = self.accuracy.calculate(predictions, y)
```

```
    # Performs forward pass
    def forward(self, X):

        # Call forward method on the input layer
        # this will set the output property that
        # the first layer in "prev" object is expecting
        self.input_layer.forward(X)

        # Call forward method of every object in a chain
        # Pass output of the previous object as a parameter
        for layer in self.layers:
            layer.forward(layer.prev.output)

        # "layer" is now the last object from the list,
        # return its output
        return layer.output
```

Full code for the `Loss` class:

```
# Common loss class
class Loss:

    # Regularization loss calculation
    def regularization_loss(self):

        # 0 by default
        regularization_loss = 0

        # Calculate regularization loss
        # iterate all trainable layers
        for layer in self.trainable_layers:

            # L1 regularization - weights
            # calculate only when factor greater than 0
            if layer.weight_regularizer_l1 > 0:
                regularization_loss += layer.weight_regularizer_l1 * \
                                       np.sum(np.abs(layer.weights))

            # L2 regularization - weights
            if layer.weight_regularizer_l2 > 0:
                regularization_loss += layer.weight_regularizer_l2 * \
                                       np.sum(layer.weights *
                                              layer.weights)
```

```
        # L1 regularization - biases
        # only calculate when factor greater than 0
        if layer.bias_regularizer_l1 > 0:
            regularization_loss += layer.bias_regularizer_l1 * \
                                   np.sum(np.abs(layer.biases))

        # L2 regularization - biases
        if layer.bias_regularizer_l2 > 0:
            regularization_loss += layer.bias_regularizer_l2 * \
                                   np.sum(layer.biases *
                                          layer.biases)

    return regularization_loss

# Set/remember trainable layers
def remember_trainable_layers(self, trainable_layers):
    self.trainable_layers = trainable_layers

# Calculates the data and regularization losses
# given model output and ground truth values
def calculate(self, output, y):

    # Calculate sample losses
    sample_losses = self.forward(output, y)

    # Calculate mean loss
    data_loss = np.mean(sample_losses)

    # Return the data and regularization losses
    return data_loss, self.regularization_loss()
```

Now that we've done a full forward pass and have calculated loss and accuracy, we can begin the backward pass. The `backward` method in the `Model` class is structurally similar to the `forward` method, just in reverse and using different parameters. Following the backward pass in our previous training approach, we need to call the `backward` method of a loss object to create the `dinputs` property. Next, we'll loop through all the layers in reverse order, calling their `backward` methods with the `dinputs` property of the next layer (in normal order) as a parameter, effectively backpropagating the gradient returned by that next layer. Remember that we have set the loss object as a `next` layer in the last, output layer.

```
# Model class
class Model:
    ...
    # Performs backward pass
    def backward(self, output, y):

        # First call backward method on the loss
        # this will set dinputs property that the last
        # layer will try to access shortly
        self.loss.backward(output, y)

        # Call backward method going through all the objects
        # in reversed order passing dinputs as a parameter
        for layer in reversed(self.layers):
            layer.backward(layer.next.dinputs)
```

Next, we'll add a call of this `backward` method to the end of the `train` method:

```
            # Perform backward pass
            self.backward(output, y)
```

After this backward pass, the last action to perform is to optimize. We have previously been calling the optimizer object's `update_params` method as many times as we had trainable layers. We have to make this code universal as well by looping through the list of trainable layers and calling `update_params()` in this loop:

```
            # Optimize (update parameters)
            self.optimizer.pre_update_params()
            for layer in self.trainable_layers:
                self.optimizer.update_params(layer)
            self.optimizer.post_update_params()
```

Then we can output useful information — here's where this last parameter to the `train` method becomes handy:

```
            # Print a summary
            if not epoch % print_every:
                print(f'epoch: {epoch}, ' +
                      f'acc: {accuracy:.3f}, ' +
                      f'loss: {loss:.3f} (' +
                      f'data_loss: {data_loss:.3f}, ' +
                      f'reg_loss: {regularization_loss:.3f}), ' +
                      f'lr: {self.optimizer.current_learning_rate}')
```

Making the full `train` method:

```
# Train the model
def train(self, X, y, *, epochs=1, print_every=1):

    # Initialize accuracy object
    self.accuracy.init(y)

    # Main training loop
    for epoch in range(1, epochs+1):

        # Perform the forward pass
        output = self.forward(X)

        # Calculate loss
        data_loss, regularization_loss = \
            self.loss.calculate(output, y)
        loss = data_loss + regularization_loss

        # Get predictions and calculate an accuracy
        predictions = self.output_layer_activation.predictions(
                          output)
        accuracy = self.accuracy.calculate(predictions, y)

        # Perform backward pass
        self.backward(output, y)

        # Optimize (update parameters)
        self.optimizer.pre_update_params()
        for layer in self.trainable_layers:
            self.optimizer.update_params(layer)
        self.optimizer.post_update_params()

        # Print a summary
        if not epoch % print_every:
            print(f'epoch: {epoch}, ' +
                  f'acc: {accuracy:.3f}, ' +
                  f'loss: {loss:.3f} (' +
                  f'data_loss: {data_loss:.3f}, ' +
                  f'reg_loss: {regularization_loss:.3f}), ' +
                  f'lr: {self.optimizer.current_learning_rate}')
```

We can now pass the accuracy class' object into the model and test our model's performance:

```
# Create dataset
X, y = sine_data()

# Instantiate the model
model = Model()

# Add layers
model.add(Layer_Dense(1, 64))
model.add(Activation_ReLU())
model.add(Layer_Dense(64, 64))
model.add(Activation_ReLU())
model.add(Layer_Dense(64, 1))
model.add(Activation_Linear())

# Set loss, optimizer and accuracy objects
model.set(
    loss=Loss_MeanSquaredError(),
    optimizer=Optimizer_Adam(learning_rate=0.005, decay=1e-3),
    accuracy=Accuracy_Regression()
)

# Finalize the model
model.finalize()

# Train the model
model.train(X, y, epochs=10000, print_every=100)
```

```
>>>
epoch: 100, acc: 0.006, loss: 0.085 (data_loss: 0.085, reg_loss: 0.000), lr:
0.004549590536851684
epoch: 200, acc: 0.032, loss: 0.035 (data_loss: 0.035, reg_loss: 0.000), lr:
0.004170141784820684
...
epoch: 9900, acc: 0.934, loss: 0.000 (data_loss: 0.000, reg_loss: 0.000),
lr: 0.00045875768419121016
epoch: 10000, acc: 0.970, loss: 0.000 (data_loss: 0.000, reg_loss: 0.000),
lr: 0.00045458678061641964
```

Our new model is behaving well, and now we're able to make new models more easily with our `Model` class. We have to continue to modify these classes to handle for entirely new models. For example, we haven't yet handled for binary logistic regression. For this, we need to add two things. First, we need to calculate the categorical accuracy:

```
# Accuracy calculation for classification model
class Accuracy_Categorical(Accuracy):

    def __init__(self, *, binary=False):
        # Binary mode?
        self.binary = binary

    # No initialization is needed
    def init(self, y):
        pass

    # Compares predictions to the ground truth values
    def compare(self, predictions, y):
        if not self.binary and len(y.shape) == 2:
            y = np.argmax(y, axis=1)
        return predictions == y
```

This is the same as the accuracy calculation for classification, just wrapped into a class and with an additional switch parameter. This switch disables one-hot to sparse label conversion while this class is used with the binary cross-entropy model, since this model always require the ground-true values to be a 2D array and they're not one-hot encoded. Note that we do not perform any initialization here, but the method needs to exist since it's going to be called from the `train` method of the `Model` class. The next thing that we need to add is the ability to validate the model using validation data. Validation requires only a forward pass and calculation of loss (just data loss). We'll modify the `calculate` method of the `Loss` class to let it calculate the validation loss as well:

```
# Common loss class
class Loss:
    ...
    # Calculates the data and regularization losses
    # given model output and ground truth values
    def calculate(self, output, y, *, include_regularization=False):

        # Calculate sample losses
        sample_losses = self.forward(output, y)

        # Calculate mean loss
        data_loss = np.mean(sample_losses)

        # If just data loss - return it
        if not include_regularization:
            return data_loss

        # Return the data and regularization losses
        return data_loss, self.regularization_loss()
```

We've added a new parameter and condition to return just the data loss, as regularization loss is not being used in this case. To run it, we'll pass predictions and targets the same way as with the training data. We will not return regularization loss by default, which means we need to update the call to this method in the `train` method to include regularization loss during training:

```
        # Calculate loss
        data_loss, regularization_loss = \
            self.loss.calculate(output, y,
                                include_regularization=True)
```

Then we can add the validation code to the `train` method in the `Model` class. We added the *`validation_data`* parameter to the function, which takes a tuple of validation data (samples and targets), the `if` statement to check if the validation data is present, and if it is — the code to perform a forward pass over these data, calculate loss and accuracy in the same way as during training and print the results:

```
# Model class
class Model:
    ...
    # Train the model
    def train(self, X, y, *, epochs=1, print_every=1,
              validation_data=None):
        ...
        # If there is the validation data
        if validation_data is not None:

            # For better readability
            X_val, y_val = validation_data

            # Perform the forward pass
            output = self.forward(X_val)

            # Calculate the loss
            loss = self.loss.calculate(output, y_val)

            # Get predictions and calculate an accuracy
            predictions = self.output_layer_activation.predictions(
                              output)
            accuracy = self.accuracy.calculate(predictions, y_val)

            # Print a summary
            print(f'validation, ' +
                  f'acc: {accuracy:.3f}, ' +
                  f'loss: {loss:.3f}')
```

The full `train` method for the `Model` class:

```
# Model class
class Model:
    ...
    # Train the model
    def train(self, X, y, *, epochs=1, print_every=1,
              validation_data=None):

        # Initialize accuracy object
        self.accuracy.init(y)

        # Main training loop
        for epoch in range(1, epochs+1):

            # Perform the forward pass
            output = self.forward(X)

            # Calculate loss
            data_loss, regularization_loss = \
                self.loss.calculate(output, y,
                                    include_regularization=True)
            loss = data_loss + regularization_loss

            # Get predictions and calculate an accuracy
            predictions = self.output_layer_activation.predictions(
                              output)
            accuracy = self.accuracy.calculate(predictions, y)

            # Perform backward pass
            self.backward(output, y)

            # Optimize (update parameters)
            self.optimizer.pre_update_params()
            for layer in self.trainable_layers:
                self.optimizer.update_params(layer)
            self.optimizer.post_update_params()

            # Print a summary
            if not epoch % print_every:
                print(f'epoch: {epoch}, ' +
                      f'acc: {accuracy:.3f}, ' +
                      f'loss: {loss:.3f} (' +
                      f'data_loss: {data_loss:.3f}, ' +
                      f'reg_loss: {regularization_loss:.3f}), ' +
                      f'lr: {self.optimizer.current_learning_rate}')
```

```
        # If there is the validation data
        if validation_data is not None:

            # For better readability
            X_val, y_val = validation_data

            # Perform the forward pass
            output = self.forward(X_val)

            # Calculate the loss
            loss = self.loss.calculate(output, y_val)

            # Get predictions and calculate an accuracy
            predictions = self.output_layer_activation.predictions(
                              output)
            accuracy = self.accuracy.calculate(predictions, y_val)

            # Print a summary
            print(f'validation, ' +
                  f'acc: {accuracy:.3f}, ' +
                  f'loss: {loss:.3f}')
```

Now we can create the test data and test the binary logistic regression model with the following code:

```
# Create train and test dataset
X, y = spiral_data(samples=100, classes=2)
X_test, y_test = spiral_data(samples=100, classes=2)

# Reshape labels to be a list of lists
# Inner list contains one output (either 0 or 1)
# per each output neuron, 1 in this case
y = y.reshape(-1, 1)
y_test = y_test.reshape(-1, 1)

# Instantiate the model
model = Model()

# Add layers
model.add(Layer_Dense(2, 64, weight_regularizer_l2=5e-4,
                             bias_regularizer_l2=5e-4))
model.add(Activation_ReLU())
model.add(Layer_Dense(64, 1))
model.add(Activation_Sigmoid())
```

```
# Set loss, optimizer and accuracy objects
model.set(
    loss=Loss_BinaryCrossentropy(),
    optimizer=Optimizer_Adam(decay=5e-7),
    accuracy=Accuracy_Categorical(binary=True)
)

# Finalize the model
model.finalize()

# Train the model
model.train(X, y, validation_data=(X_test, y_test),
            epochs=10000, print_every=100)
```

```
>>>
epoch: 100, acc: 0.625, loss: 0.675 (data_loss: 0.674, reg_loss: 0.001), lr:
0.0009999950502450128 7
epoch: 200, acc: 0.630, loss: 0.669 (data_loss: 0.668, reg_loss: 0.001), lr:
0.0009999900509899265 1
...
epoch: 9900, acc: 0.905, loss: 0.312 (data_loss: 0.276, reg_loss: 0.037),
lr: 0.0009950748768967994
epoch: 10000, acc: 0.905, loss: 0.312 (data_loss: 0.275, reg_loss: 0.036),
lr: 0.0009950253706593885
validation, acc: 0.775, loss: 0.423
```

Now that we're streamlining the forward and backward pass code, including validation, this is a good time to reintroduce dropout. Recall that dropout is a method to disable, or filter out, certain neurons in an attempt to regularize and improve our model's ability to generalize. If dropout is employed in our model, we want to make sure to leave it out when performing validation and inference (predictions); in our previous code, we left it out by not calling its `forward` method during the forward pass during validation. Here we have a common method for performing a forward pass for both training and validation, so we need a different approach for turning off dropout — to inform the layers if we are during the training and let them "decide" on calculation to include. The first thing we'll do is include a *`training`* boolean argument for the `forward` method in all the layer and activation classes, since we are calling them in a unified form:

```
    # Forward pass
    def forward(self, inputs, training):
```

When we're not training, we can set the output to the input directly in the `Layer_Dropout` class and return from the method without changing outputs:

```
        # If not in the training mode - return values
        if not training:
            self.output = inputs.copy()
            return
```

When we are training, we will engage the dropout:

```
# Dropout
class Layer_Dropout:
    ...
    # Forward pass
    def forward(self, inputs, training):
        # Save input values
        self.input = inputs

        # If not in the training mode - return values
        if not training:
            self.output = inputs.copy()
            return

        # Generate and save scaled mask
        self.binary_mask = np.random.binomial(1, self.rate,
                           size=inputs.shape) / self.rate
        # Apply mask to output values
        self.output = inputs * self.binary_mask
```

Next, we modify the `forward` method of our `Model` class to add the *`training`* parameter and a call to the `forward` methods of the layers to take this parameter's value:

```
# Model class
class Model:
    ...
    # Performs forward pass
    def forward(self, X, training):

        # Call forward method on the input layer
        # this will set the output property that
        # the first layer in "prev" object is expecting
        self.input_layer.forward(X, training)

        # Call forward method of every object in a chain
        # Pass output of the previous object as a parameter
        for layer in self.layers:
            layer.forward(layer.prev.output, training)
```

```
        # "layer" is now the last object from the list,
        # return its output
        return layer.output
```

We also need to update the `train` method in the `Model` class since the *`training`* parameter in the forward method call will need to be set to `True`:

```
            # Perform the forward pass
            output = self.forward(X, training=True)
```

Then set to `False` during validation:

```
            # Perform the forward pass
            output = self.forward(X_val, training=False)
```

Making the full `train` method in the `Model` class:

```
# Model class
class Model:
    ...
    # Train the model
    def train(self, X, y, *, epochs=1, print_every=1,
              validation_data=None):

        # Initialize accuracy object
        self.accuracy.init(y)

        # Main training loop
        for epoch in range(1, epochs+1):

            # Perform the forward pass
            output = self.forward(X, training=True)

            # Calculate loss
            data_loss, regularization_loss = \
                self.loss.calculate(output, y,
                                    include_regularization=True)
            loss = data_loss + regularization_loss

            # Get predictions and calculate an accuracy
            predictions = self.output_layer_activation.predictions(
                              output)
            accuracy = self.accuracy.calculate(predictions, y)

            # Perform backward pass
            self.backward(output, y)
```

```
        # Optimize (update parameters)
        self.optimizer.pre_update_params()
        for layer in self.trainable_layers:
            self.optimizer.update_params(layer)
        self.optimizer.post_update_params()
        # Print a summary
        if not epoch % print_every:
            print(f'epoch: {epoch}, ' +
                  f'acc: {accuracy:.3f}, ' +
                  f'loss: {loss:.3f} (' +
                  f'data_loss: {data_loss:.3f}, ' +
                  f'reg_loss: {regularization_loss:.3f}), ' +
                  f'lr: {self.optimizer.current_learning_rate}')

    # If there is the validation data
    if validation_data is not None:

        # For better readability
        X_val, y_val = validation_data

        # Perform the forward pass
        output = self.forward(X_val, training=False)

        # Calculate the loss
        loss = self.loss.calculate(output, y_val)

        # Get predictions and calculate an accuracy
        predictions = self.output_layer_activation.predictions(
                          output)
        accuracy = self.accuracy.calculate(predictions, y_val)
        # Print a summary
        print(f'validation, ' +
              f'acc: {accuracy:.3f}, ' +
              f'loss: {loss:.3f}')
```

The last thing that we have to take care of, with the `Model` class, is the combined Softmax activation and CrossEntropy loss class that we created for faster gradient calculation. The challenge here is that previously we have been defining forward and backward passes by hand for every model separately. Now, however, we have loops over layers in both directions of calculations, a unified way of calculating outputs and gradients, and other improvements. We cannot just simply remove the Softmax activation and CrossEntropy loss and replace them with an object combining both. It won't work with the code that we have so far, since we are handling the output activation function and loss in a specific way. Since the combined object contains just a backward pass optimization, let's leave the forward pass as is, using separate Softmax activation and Categorical Cross-Entropy loss objects, and handle just for the backward pass.

To start, we want to automatically decide if the current model is a classifier and if it uses the

Softmax activation and Categorical Cross-Entropy loss. This can be achieved by checking the class name of the last layer's object, which is an activation function's object, and by checking the class name of the loss function's object. We'll add this check to the end of the `finalize` method:

```
# If output activation is Softmax and
# loss function is Categorical Cross-Entropy
# create an object of combined activation
# and loss function containing
# faster gradient calculation
if isinstance(self.layers[-1], Activation_Softmax) and \
   isinstance(self.loss, Loss_CategoricalCrossentropy):
    # Create an object of combined activation
    # and loss functions
    self.softmax_classifier_output = \
        Activation_Softmax_Loss_CategoricalCrossentropy()
```

To make this check, we are using Python's `isinstance` function, which returns *True* if a given object is an instance of a given class. If both of the tests return *True*, we are setting a new property containing an object of the `Activation_Softmax_Loss_CategoricalCrossentropy` class. We also want to initialize this property with a value of `None` in the `Model` class' constructor:

```
# Softmax classifier's output object
self.softmax_classifier_output = None
```

The last step is, during the backward pass, to check if this object is set and, if it is, to use it. To do so, we need to handle this case separately with a slightly modified version of the current code of the backward pass. First, we call the `backward` method of the combined object, then, since we won't call the `backward` method of the activation function (the last object on a list of layers), set the `dinputs` of the object of this function with the gradient calculated within the activation/loss object. At the end, we can iterate all of the layers except for the last one and perform the backward pass on them:

```
# If softmax classifier
if self.softmax_classifier_output is not None:
    # First call backward method
    # on the combined activation/loss
    # this will set dinputs property
    self.softmax_classifier_output.backward(output, y)

    # Since we'll not call backward method of the last layer
    # which is Softmax activation
    # as we used combined activation/loss
    # object, let's set dinputs in this object
    self.layers[-1].dinputs = \
        self.softmax_classifier_output.dinputs
```

```
            # Call backward method going through
            # all the objects but last
            # in reversed order passing dinputs as a parameter
            for layer in reversed(self.layers[:-1]):
                layer.backward(layer.next.dinputs)

            return
```

Full `Model` class up to this point:

```
# Model class
class Model:

    def __init__(self):
        # Create a list of network objects
        self.layers = []
        # Softmax classifier's output object
        self.softmax_classifier_output = None

    # Add objects to the model
    def add(self, layer):
        self.layers.append(layer)

    # Set loss, optimizer and accuracy
    def set(self, *, loss, optimizer, accuracy):
        self.loss = loss
        self.optimizer = optimizer
        self.accuracy = accuracy

    # Finalize the model
    def finalize(self):

        # Create and set the input layer
        self.input_layer = Layer_Input()

        # Count all the objects
        layer_count = len(self.layers)

        # Initialize a list containing trainable layers:
        self.trainable_layers = []

        # Iterate the objects
        for i in range(layer_count):

            # If it's the first layer,
            # the previous layer object is the input layer
            if i == 0:
                self.layers[i].prev = self.input_layer
```

```
            self.layers[i].next = self.layers[i+1]

        # All layers except for the first and the last
        elif i < layer_count - 1:
            self.layers[i].prev = self.layers[i-1]
            self.layers[i].next = self.layers[i+1]

        # The last layer - the next object is the loss
        # Also let's save aside the reference to the last object
        # whose output is the model's output
        else:
            self.layers[i].prev = self.layers[i-1]
            self.layers[i].next = self.loss
            self.output_layer_activation = self.layers[i]

        # If layer contains an attribute called "weights",
        # it's a trainable layer -
        # add it to the list of trainable layers
        # We don't need to check for biases -
        # checking for weights is enough
        if hasattr(self.layers[i], 'weights'):
            self.trainable_layers.append(self.layers[i])

    # Update loss object with trainable layers
    self.loss.remember_trainable_layers(
        self.trainable_layers
    )

    # If output activation is Softmax and
    # loss function is Categorical Cross-Entropy
    # create an object of combined activation
    # and loss function containing
    # faster gradient calculation
    if isinstance(self.layers[-1], Activation_Softmax) and \
       isinstance(self.loss, Loss_CategoricalCrossentropy):
        # Create an object of combined activation
        # and loss functions
        self.softmax_classifier_output = \
            Activation_Softmax_Loss_CategoricalCrossentropy()

# Train the model
def train(self, X, y, *, epochs=1, print_every=1,
          validation_data=None):

    # Initialize accuracy object
    self.accuracy.init(y)

    # Main training loop
    for epoch in range(1, epochs+1):
```

```
        # Perform the forward pass
        output = self.forward(X, training=True)

        # Calculate loss
        data_loss, regularization_loss = \
            self.loss.calculate(output, y,
                                include_regularization=True)
        loss = data_loss + regularization_loss

        # Get predictions and calculate an accuracy
        predictions = self.output_layer_activation.predictions(
                          output)
        accuracy = self.accuracy.calculate(predictions, y)

        # Perform backward pass
        self.backward(output, y)

        # Optimize (update parameters)
        self.optimizer.pre_update_params()
        for layer in self.trainable_layers:
            self.optimizer.update_params(layer)
        self.optimizer.post_update_params()

        # Print a summary
        if not epoch % print_every:
            print(f'epoch: {epoch}, ' +
                  f'acc: {accuracy:.3f}, ' +
                  f'loss: {loss:.3f} (' +
                  f'data_loss: {data_loss:.3f}, ' +
                  f'reg_loss: {regularization_loss:.3f}), ' +
                  f'lr: {self.optimizer.current_learning_rate}')

    # If there is the validation data
    if validation_data is not None:

        # For better readability
        X_val, y_val = validation_data

        # Perform the forward pass
        output = self.forward(X_val, training=False)

        # Calculate the loss
        loss = self.loss.calculate(output, y_val)

        # Get predictions and calculate an accuracy
        predictions = self.output_layer_activation.predictions(
                          output)
        accuracy = self.accuracy.calculate(predictions, y_val)
```

```
        # Print a summary
        print(f'validation, ' +
              f'acc: {accuracy:.3f}, ' +
              f'loss: {loss:.3f}')

    # Performs forward pass
    def forward(self, X, training):

        # Call forward method on the input layer
        # this will set the output property that
        # the first layer in "prev" object is expecting
        self.input_layer.forward(X, training)

        # Call forward method of every object in a chain
        # Pass output of the previous object as a parameter
        for layer in self.layers:
            layer.forward(layer.prev.output, training)

        # "layer" is now the last object from the list,
        # return its output
        return layer.output

    # Performs backward pass
    def backward(self, output, y):

        # If softmax classifier
        if self.softmax_classifier_output is not None:
            # First call backward method
            # on the combined activation/loss
            # this will set dinputs property
            self.softmax_classifier_output.backward(output, y)

            # Since we'll not call backward method of the last layer
            # which is Softmax activation
            # as we used combined activation/loss
            # object, let's set dinputs in this object
            self.layers[-1].dinputs = \
                self.softmax_classifier_output.dinputs

            # Call backward method going through
            # all the objects but last
            # in reversed order passing dinputs as a parameter
            for layer in reversed(self.layers[:-1]):
                layer.backward(layer.next.dinputs)

            return
```

```
        # First call backward method on the loss
        # this will set dinputs property that the last
        # layer will try to access shortly
        self.loss.backward(output, y)

        # Call backward method going through all the objects
        # in reversed order passing dinputs as a parameter
        for layer in reversed(self.layers):
            layer.backward(layer.next.dinputs)
```

Also we won't need the initializer and the forward method of the `Activation_Softmax_Loss_CategoricalCrossentropy` class anymore so we can remove them leaving in just the backward pass:

```
# Softmax classifier - combined Softmax activation
# and cross-entropy loss for faster backward step
class Activation_Softmax_Loss_CategoricalCrossentropy():

    # Backward pass
    def backward(self, dvalues, y_true):

        # Number of samples
        samples = len(dvalues)

        # If labels are one-hot encoded,
        # turn them into discrete values
        if len(y_true.shape) == 2:
            y_true = np.argmax(y_true, axis=1)

        # Copy so we can safely modify
        self.dinputs = dvalues.copy()
        # Calculate gradient
        self.dinputs[range(samples), y_true] -= 1
        # Normalize gradient
        self.dinputs = self.dinputs / samples
```

Now we can test our updated `Model` object with dropout:

```
# Create dataset
X, y = spiral_data(samples=1000, classes=3)
X_test, y_test = spiral_data(samples=100, classes=3)

# Instantiate the model
model = Model()
```

```
# Add layers
model.add(Layer_Dense(2, 512, weight_regularizer_l2=5e-4,
                              bias_regularizer_l2=5e-4))
model.add(Activation_ReLU())
model.add(Layer_Dropout(0.1))
model.add(Layer_Dense(512, 3))
model.add(Activation_Softmax())

# Set loss, optimizer and accuracy objects
model.set(
    loss=Loss_CategoricalCrossentropy(),
    optimizer=Optimizer_Adam(learning_rate=0.05, decay=5e-5),
    accuracy=Accuracy_Categorical()
)

# Finalize the model
model.finalize()

# Train the model
model.train(X, y, validation_data=(X_test, y_test),
            epochs=10000, print_every=100)
```

```
>>>
epoch: 100, acc: 0.716, loss: 0.726 (data_loss: 0.666, reg_loss: 0.060), lr:
0.04975371909050202
epoch: 200, acc: 0.787, loss: 0.615 (data_loss: 0.538, reg_loss: 0.077), lr:
0.049507401356502806
...
epoch: 9900, acc: 0.861, loss: 0.436 (data_loss: 0.389, reg_loss: 0.046),
lr: 0.0334459346466437
epoch: 10000, acc: 0.880, loss: 0.394 (data_loss: 0.347, reg_loss: 0.047),
lr: 0.03333444448148271
validation, acc: 0.867, loss: 0.379
```

It seems like everything is working as intended. Now that we've got this `Model` class, we're able to define new models without writing large amounts of code repeatedly. Rewriting code is annoying and leaves more room to make small, hard-to-notice mistakes.

Full code up to this point:

```
import numpy as np
import nnfs
from nnfs.datasets import spiral_data

nnfs.init()


# Dense layer
class Layer_Dense:

    # Layer initialization
    def __init__(self, n_inputs, n_neurons,
                 weight_regularizer_l1=0, weight_regularizer_l2=0,
                 bias_regularizer_l1=0, bias_regularizer_l2=0):
        # Initialize weights and biases
        self.weights = 0.01 * np.random.randn(n_inputs, n_neurons)
        self.biases = np.zeros((1, n_neurons))
        # Set regularization strength
        self.weight_regularizer_l1 = weight_regularizer_l1
        self.weight_regularizer_l2 = weight_regularizer_l2
        self.bias_regularizer_l1 = bias_regularizer_l1
        self.bias_regularizer_l2 = bias_regularizer_l2

    # Forward pass
    def forward(self, inputs, training):
        # Remember input values
        self.inputs = inputs
        # Calculate output values from inputs, weights and biases
        self.output = np.dot(inputs, self.weights) + self.biases

    # Backward pass
    def backward(self, dvalues):
        # Gradients on parameters
        self.dweights = np.dot(self.inputs.T, dvalues)
        self.dbiases = np.sum(dvalues, axis=0, keepdims=True)
```

```
        # Gradients on regularization
        # L1 on weights
        if self.weight_regularizer_l1 > 0:
            dL1 = np.ones_like(self.weights)
            dL1[self.weights < 0] = -1
            self.dweights += self.weight_regularizer_l1 * dL1
        # L2 on weights
        if self.weight_regularizer_l2 > 0:
            self.dweights += 2 * self.weight_regularizer_l2 * \
                             self.weights
        # L1 on biases
        if self.bias_regularizer_l1 > 0:
            dL1 = np.ones_like(self.biases)
            dL1[self.biases < 0] = -1
            self.dbiases += self.bias_regularizer_l1 * dL1
        # L2 on biases
        if self.bias_regularizer_l2 > 0:
            self.dbiases += 2 * self.bias_regularizer_l2 * \
                            self.biases

        # Gradient on values
        self.dinputs = np.dot(dvalues, self.weights.T)


# Dropout
class Layer_Dropout:

    # Init
    def __init__(self, rate):
        # Store rate, we invert it as for example for dropout
        # of 0.1 we need success rate of 0.9
        self.rate = 1 - rate

    # Forward pass
    def forward(self, inputs, training):
        # Save input values
        self.inputs = inputs

        # If not in the training mode - return values
        if not training:
            self.output = inputs.copy()
            return

        # Generate and save scaled mask
        self.binary_mask = np.random.binomial(1, self.rate,
                           size=inputs.shape) / self.rate
        # Apply mask to output values
        self.output = inputs * self.binary_mask
```

```
    # Backward pass
    def backward(self, dvalues):
        # Gradient on values
        self.dinputs = dvalues * self.binary_mask


# Input "layer"
class Layer_Input:

    # Forward pass
    def forward(self, inputs, training):
        self.output = inputs


# ReLU activation
class Activation_ReLU:

    # Forward pass
    def forward(self, inputs, training):
        # Remember input values
        self.inputs = inputs
        # Calculate output values from inputs
        self.output = np.maximum(0, inputs)

    # Backward pass
    def backward(self, dvalues):
        # Since we need to modify original variable,
        # let's make a copy of values first
        self.dinputs = dvalues.copy()

        # Zero gradient where input values were negative
        self.dinputs[self.inputs <= 0] = 0

    # Calculate predictions for outputs
    def predictions(self, outputs):
        return outputs


# Softmax activation
class Activation_Softmax:

    # Forward pass
    def forward(self, inputs, training):
        # Remember input values
        self.inputs = inputs

        # Get unnormalized probabilities
        exp_values = np.exp(inputs - np.max(inputs, axis=1,
                                            keepdims=True))
```

```
        # Normalize them for each sample
        probabilities = exp_values / np.sum(exp_values, axis=1,
                                            keepdims=True)

        self.output = probabilities

    # Backward pass
    def backward(self, dvalues):

        # Create uninitialized array
        self.dinputs = np.empty_like(dvalues)

        # Enumerate outputs and gradients
        for index, (single_output, single_dvalues) in \
                enumerate(zip(self.output, dvalues)):
            # Flatten output array
            single_output = single_output.reshape(-1, 1)
            # Calculate Jacobian matrix of the output
            jacobian_matrix = np.diagflat(single_output) - \
                              np.dot(single_output, single_output.T)
            # Calculate sample-wise gradient
            # and add it to the array of sample gradients
            self.dinputs[index] = np.dot(jacobian_matrix,
                                         single_dvalues)

    # Calculate predictions for outputs
    def predictions(self, outputs):
        return np.argmax(outputs, axis=1)


# Sigmoid activation
class Activation_Sigmoid:

    # Forward pass
    def forward(self, inputs, training):
        # Save input and calculate/save output
        # of the sigmoid function
        self.inputs = inputs
        self.output = 1 / (1 + np.exp(-inputs))

    # Backward pass
    def backward(self, dvalues):
        # Derivative - calculates from output of the sigmoid function
        self.dinputs = dvalues * (1 - self.output) * self.output

    # Calculate predictions for outputs
    def predictions(self, outputs):
        return (outputs > 0.5) * 1
```

```python
# Linear activation
class Activation_Linear:

    # Forward pass
    def forward(self, inputs, training):
        # Just remember values
        self.inputs = inputs
        self.output = inputs

    # Backward pass
    def backward(self, dvalues):
        # derivative is 1, 1 * dvalues = dvalues - the chain rule
        self.dinputs = dvalues.copy()

    # Calculate predictions for outputs
    def predictions(self, outputs):
        return outputs


# SGD optimizer
class Optimizer_SGD:

    # Initialize optimizer - set settings,
    # learning rate of 1. is default for this optimizer
    def __init__(self, learning_rate=1., decay=0., momentum=0.):
        self.learning_rate = learning_rate
        self.current_learning_rate = learning_rate
        self.decay = decay
        self.iterations = 0
        self.momentum = momentum

    # Call once before any parameter updates
    def pre_update_params(self):
        if self.decay:
            self.current_learning_rate = self.learning_rate * \
                (1. / (1. + self.decay * self.iterations))

    # Update parameters
    def update_params(self, layer):

        # If we use momentum
        if self.momentum:
            # If layer does not contain momentum arrays, create them
            # filled with zeros
            if not hasattr(layer, 'weight_momentums'):
                layer.weight_momentums = np.zeros_like(layer.weights)
                # If there is no momentum array for weights
                # The array doesn't exist for biases yet either.
                layer.bias_momentums = np.zeros_like(layer.biases)
```

```
            # Build weight updates with momentum - take previous
            # updates multiplied by retain factor and update with
            # current gradients
            weight_updates = \
                self.momentum * layer.weight_momentums - \
                self.current_learning_rate * layer.dweights
            layer.weight_momentums = weight_updates

            # Build bias updates
            bias_updates = \
                self.momentum * layer.bias_momentums - \
                self.current_learning_rate * layer.dbiases
            layer.bias_momentums = bias_updates

        # Vanilla SGD updates (as before momentum update)
        else:
            weight_updates = -self.current_learning_rate * \
                             layer.dweights
            bias_updates = -self.current_learning_rate * \
                           layer.dbiases

        # Update weights and biases using either
        # vanilla or momentum updates
        layer.weights += weight_updates
        layer.biases += bias_updates

    # Call once after any parameter updates
    def post_update_params(self):
        self.iterations += 1


# Adagrad optimizer
class Optimizer_Adagrad:

    # Initialize optimizer - set settings
    def __init__(self, learning_rate=1., decay=0., epsilon=1e-7):
        self.learning_rate = learning_rate
        self.current_learning_rate = learning_rate
        self.decay = decay
        self.iterations = 0
        self.epsilon = epsilon

    # Call once before any parameter updates
    def pre_update_params(self):
        if self.decay:
            self.current_learning_rate = self.learning_rate * \
                (1. / (1. + self.decay * self.iterations))
```

```
    # Update parameters
    def update_params(self, layer):

        # If layer does not contain cache arrays,
        # create them filled with zeros
        if not hasattr(layer, 'weight_cache'):
            layer.weight_cache = np.zeros_like(layer.weights)
            layer.bias_cache = np.zeros_like(layer.biases)

        # Update cache with squared current gradients
        layer.weight_cache += layer.dweights**2
        layer.bias_cache += layer.dbiases**2

        # Vanilla SGD parameter update + normalization
        # with square rooted cache
        layer.weights += -self.current_learning_rate * \
                         layer.dweights / \
                         (np.sqrt(layer.weight_cache) + self.epsilon)
        layer.biases += -self.current_learning_rate * \
                        layer.dbiases / \
                        (np.sqrt(layer.bias_cache) + self.epsilon)

    # Call once after any parameter updates
    def post_update_params(self):
        self.iterations += 1


# RMSprop optimizer
class Optimizer_RMSprop:

    # Initialize optimizer - set settings
    def __init__(self, learning_rate=0.001, decay=0., epsilon=1e-7,
                 rho=0.9):
        self.learning_rate = learning_rate
        self.current_learning_rate = learning_rate
        self.decay = decay
        self.iterations = 0
        self.epsilon = epsilon
        self.rho = rho

    # Call once before any parameter updates
    def pre_update_params(self):
        if self.decay:
            self.current_learning_rate = self.learning_rate * \
                (1. / (1. + self.decay * self.iterations))
```

```
    # Update parameters
    def update_params(self, layer):

        # If layer does not contain cache arrays,
        # create them filled with zeros
        if not hasattr(layer, 'weight_cache'):
            layer.weight_cache = np.zeros_like(layer.weights)
            layer.bias_cache = np.zeros_like(layer.biases)

        # Update cache with squared current gradients
        layer.weight_cache = self.rho * layer.weight_cache + \
            (1 - self.rho) * layer.dweights**2
        layer.bias_cache = self.rho * layer.bias_cache + \
            (1 - self.rho) * layer.dbiases**2

        # Vanilla SGD parameter update + normalization
        # with square rooted cache
        layer.weights += -self.current_learning_rate * \
                         layer.dweights / \
                         (np.sqrt(layer.weight_cache) + self.epsilon)
        layer.biases += -self.current_learning_rate * \
                        layer.dbiases / \
                        (np.sqrt(layer.bias_cache) + self.epsilon)

    # Call once after any parameter updates
    def post_update_params(self):
        self.iterations += 1


# Adam optimizer
class Optimizer_Adam:

    # Initialize optimizer - set settings
    def __init__(self, learning_rate=0.001, decay=0., epsilon=1e-7,
                 beta_1=0.9, beta_2=0.999):
        self.learning_rate = learning_rate
        self.current_learning_rate = learning_rate
        self.decay = decay
        self.iterations = 0
        self.epsilon = epsilon
        self.beta_1 = beta_1
        self.beta_2 = beta_2

    # Call once before any parameter updates
    def pre_update_params(self):
        if self.decay:
            self.current_learning_rate = self.learning_rate * \
                (1. / (1. + self.decay * self.iterations))
```

```
# Update parameters
def update_params(self, layer):

    # If layer does not contain cache arrays,
    # create them filled with zeros
    if not hasattr(layer, 'weight_cache'):
        layer.weight_momentums = np.zeros_like(layer.weights)
        layer.weight_cache = np.zeros_like(layer.weights)
        layer.bias_momentums = np.zeros_like(layer.biases)
        layer.bias_cache = np.zeros_like(layer.biases)

    # Update momentum  with current gradients
    layer.weight_momentums = self.beta_1 * \
                             layer.weight_momentums + \
                             (1 - self.beta_1) * layer.dweights
    layer.bias_momentums = self.beta_1 * \
                           layer.bias_momentums + \
                           (1 - self.beta_1) * layer.dbiases
    # Get corrected momentum
    # self.iteration is 0 at first pass
    # and we need to start with 1 here
    weight_momentums_corrected = layer.weight_momentums / \
        (1 - self.beta_1 ** (self.iterations + 1))
    bias_momentums_corrected = layer.bias_momentums / \
        (1 - self.beta_1 ** (self.iterations + 1))
    # Update cache with squared current gradients
    layer.weight_cache = self.beta_2 * layer.weight_cache + \
        (1 - self.beta_2) * layer.dweights**2
    layer.bias_cache = self.beta_2 * layer.bias_cache + \
        (1 - self.beta_2) * layer.dbiases**2
    # Get corrected cache
    weight_cache_corrected = layer.weight_cache / \
        (1 - self.beta_2 ** (self.iterations + 1))
    bias_cache_corrected = layer.bias_cache / \
        (1 - self.beta_2 ** (self.iterations + 1))

    # Vanilla SGD parameter update + normalization
    # with square rooted cache
    layer.weights += -self.current_learning_rate * \
                     weight_momentums_corrected / \
                     (np.sqrt(weight_cache_corrected) +
                         self.epsilon)
    layer.biases += -self.current_learning_rate * \
                     bias_momentums_corrected / \
                     (np.sqrt(bias_cache_corrected) +
                         self.epsilon)
```

```
    # Call once after any parameter updates
    def post_update_params(self):
        self.iterations += 1


# Common loss class
class Loss:

    # Regularization loss calculation
    def regularization_loss(self):

        # 0 by default
        regularization_loss = 0

        # Calculate regularization loss
        # iterate all trainable layers
        for layer in self.trainable_layers:

            # L1 regularization - weights
            # calculate only when factor greater than 0
            if layer.weight_regularizer_l1 > 0:
                regularization_loss += layer.weight_regularizer_l1 * \
                                       np.sum(np.abs(layer.weights))

            # L2 regularization - weights
            if layer.weight_regularizer_l2 > 0:
                regularization_loss += layer.weight_regularizer_l2 * \
                                       np.sum(layer.weights *
                                              layer.weights)

            # L1 regularization - biases
            # calculate only when factor greater than 0
            if layer.bias_regularizer_l1 > 0:
                regularization_loss += layer.bias_regularizer_l1 * \
                                       np.sum(np.abs(layer.biases))

            # L2 regularization - biases
            if layer.bias_regularizer_l2 > 0:
                regularization_loss += layer.bias_regularizer_l2 * \
                                       np.sum(layer.biases *
                                              layer.biases)

        return regularization_loss

    # Set/remember trainable layers
    def remember_trainable_layers(self, trainable_layers):
        self.trainable_layers = trainable_layers
```

```
    # Calculates the data and regularization losses
    # given model output and ground truth values
    def calculate(self, output, y, *, include_regularization=False):

        # Calculate sample losses
        sample_losses = self.forward(output, y)

        # Calculate mean loss
        data_loss = np.mean(sample_losses)

        # If just data loss - return it
        if not include_regularization:
            return data_loss

        # Return the data and regularization losses
        return data_loss, self.regularization_loss()


# Cross-entropy loss
class Loss_CategoricalCrossentropy(Loss):

    # Forward pass
    def forward(self, y_pred, y_true):

        # Number of samples in a batch
        samples = len(y_pred)

        # Clip data to prevent division by 0
        # Clip both sides to not drag mean towards any value
        y_pred_clipped = np.clip(y_pred, 1e-7, 1 - 1e-7)

        # Probabilities for target values -
        # only if categorical labels
        if len(y_true.shape) == 1:
            correct_confidences = y_pred_clipped[
                range(samples),
                y_true
            ]

        # Mask values - only for one-hot encoded labels
        elif len(y_true.shape) == 2:
            correct_confidences = np.sum(
                y_pred_clipped * y_true,
                axis=1
            )

        # Losses
        negative_log_likelihoods = -np.log(correct_confidences)
        return negative_log_likelihoods
```

```
    # Backward pass
    def backward(self, dvalues, y_true):

        # Number of samples
        samples = len(dvalues)
        # Number of labels in every sample
        # We'll use the first sample to count them
        labels = len(dvalues[0])

        # If labels are sparse, turn them into one-hot vector
        if len(y_true.shape) == 1:
            y_true = np.eye(labels)[y_true]

        # Calculate gradient
        self.dinputs = -y_true / dvalues
        # Normalize gradient
        self.dinputs = self.dinputs / samples


# Softmax classifier - combined Softmax activation
# and cross-entropy loss for faster backward step
class Activation_Softmax_Loss_CategoricalCrossentropy():

    # Backward pass
    def backward(self, dvalues, y_true):

        # Number of samples
        samples = len(dvalues)

        # If labels are one-hot encoded,
        # turn them into discrete values
        if len(y_true.shape) == 2:
            y_true = np.argmax(y_true, axis=1)

        # Copy so we can safely modify
        self.dinputs = dvalues.copy()
        # Calculate gradient
        self.dinputs[range(samples), y_true] -= 1
        # Normalize gradient
        self.dinputs = self.dinputs / samples
```

```
# Binary cross-entropy loss
class Loss_BinaryCrossentropy(Loss):

    # Forward pass
    def forward(self, y_pred, y_true):

        # Clip data to prevent division by 0
        # Clip both sides to not drag mean towards any value
        y_pred_clipped = np.clip(y_pred, 1e-7, 1 - 1e-7)

        # Calculate sample-wise loss
        sample_losses = -(y_true * np.log(y_pred_clipped) +
                          (1 - y_true) * np.log(1 - y_pred_clipped))
        sample_losses = np.mean(sample_losses, axis=-1)

        # Return losses
        return sample_losses

    # Backward pass
    def backward(self, dvalues, y_true):

        # Number of samples
        samples = len(dvalues)
        # Number of outputs in every sample
        # We'll use the first sample to count them
        outputs = len(dvalues[0])

        # Clip data to prevent division by 0
        # Clip both sides to not drag mean towards any value
        clipped_dvalues = np.clip(dvalues, 1e-7, 1 - 1e-7)

        # Calculate gradient
        self.dinputs = -(y_true / clipped_dvalues -
                         (1 - y_true) / (1 - clipped_dvalues)) / outputs
        # Normalize gradient
        self.dinputs = self.dinputs / samples


# Mean Squared Error loss
class Loss_MeanSquaredError(Loss):  # L2 loss

    # Forward pass
    def forward(self, y_pred, y_true):

        # Calculate loss
        sample_losses = np.mean((y_true - y_pred)**2, axis=-1)

        # Return losses
        return sample_losses
```

```
    # Backward pass
    def backward(self, dvalues, y_true):

        # Number of samples
        samples = len(dvalues)
        # Number of outputs in every sample
        # We'll use the first sample to count them
        outputs = len(dvalues[0])

        # Gradient on values
        self.dinputs = -2 * (y_true - dvalues) / outputs
        # Normalize gradient
        self.dinputs = self.dinputs / samples


# Mean Absolute Error loss
class Loss_MeanAbsoluteError(Loss):  # L1 loss

    def forward(self, y_pred, y_true):

        # Calculate loss
        sample_losses = np.mean(np.abs(y_true - y_pred), axis=-1)

        # Return losses
        return sample_losses

    # Backward pass
    def backward(self, dvalues, y_true):

        # Number of samples
        samples = len(dvalues)
        # Number of outputs in every sample
        # We'll use the first sample to count them
        outputs = len(dvalues[0])

        # Calculate gradient
        self.dinputs = np.sign(y_true - dvalues) / outputs
        # Normalize gradient
        self.dinputs = self.dinputs / samples
```

```python
# Common accuracy class
class Accuracy:

    # Calculates an accuracy
    # given predictions and ground truth values
    def calculate(self, predictions, y):

        # Get comparison results
        comparisons = self.compare(predictions, y)

        # Calculate an accuracy
        accuracy = np.mean(comparisons)

        # Return accuracy
        return accuracy


# Accuracy calculation for classification model
class Accuracy_Categorical(Accuracy):

    def __init__(self, *, binary=False):
        # Binary mode?
        self.binary = binary

    # No initialization is needed
    def init(self, y):
        pass

    # Compares predictions to the ground truth values
    def compare(self, predictions, y):
        if not self.binary and len(y.shape) == 2:
            y = np.argmax(y, axis=1)
        return predictions == y


# Accuracy calculation for regression model
class Accuracy_Regression(Accuracy):

    def __init__(self):
        # Create precision property
        self.precision = None

    # Calculates precision value
    # based on passed-in ground truth values
    def init(self, y, reinit=False):
        if self.precision is None or reinit:
            self.precision = np.std(y) / 250
```

```
    # Compares predictions to the ground truth values
    def compare(self, predictions, y):
        return np.absolute(predictions - y) < self.precision


# Model class
class Model:

    def __init__(self):
        # Create a list of network objects
        self.layers = []
        # Softmax classifier's output object
        self.softmax_classifier_output = None

    # Add objects to the model
    def add(self, layer):
        self.layers.append(layer)

    # Set loss, optimizer and accuracy
    def set(self, *, loss, optimizer, accuracy):
        self.loss = loss
        self.optimizer = optimizer
        self.accuracy = accuracy

    # Finalize the model
    def finalize(self):

        # Create and set the input layer
        self.input_layer = Layer_Input()

        # Count all the objects
        layer_count = len(self.layers)

        # Initialize a list containing trainable layers:
        self.trainable_layers = []

        # Iterate the objects
        for i in range(layer_count):

            # If it's the first layer,
            # the previous layer object is the input layer
            if i == 0:
                self.layers[i].prev = self.input_layer
                self.layers[i].next = self.layers[i+1]

            # All layers except for the first and the last
            elif i < layer_count - 1:
                self.layers[i].prev = self.layers[i-1]
                self.layers[i].next = self.layers[i+1]
```

```
        # The last layer - the next object is the loss
        # Also let's save aside the reference to the last object
        # whose output is the model's output
        else:
            self.layers[i].prev = self.layers[i-1]
            self.layers[i].next = self.loss
            self.output_layer_activation = self.layers[i]

        # If layer contains an attribute called "weights",
        # it's a trainable layer -
        # add it to the list of trainable layers
        # We don't need to check for biases -
        # checking for weights is enough
        if hasattr(self.layers[i], 'weights'):
            self.trainable_layers.append(self.layers[i])

    # Update loss object with trainable layers
    self.loss.remember_trainable_layers(
        self.trainable_layers
    )

    # If output activation is Softmax and
    # loss function is Categorical Cross-Entropy
    # create an object of combined activation
    # and loss function containing
    # faster gradient calculation
    if isinstance(self.layers[-1], Activation_Softmax) and \
       isinstance(self.loss, Loss_CategoricalCrossentropy):
        # Create an object of combined activation
        # and loss functions
        self.softmax_classifier_output = \
            Activation_Softmax_Loss_CategoricalCrossentropy()

# Train the model
def train(self, X, y, *, epochs=1, print_every=1,
          validation_data=None):

    # Initialize accuracy object
    self.accuracy.init(y)

    # Main training loop
    for epoch in range(1, epochs+1):

        # Perform the forward pass
        output = self.forward(X, training=True)
```

```
        # Calculate loss
        data_loss, regularization_loss = \
            self.loss.calculate(output, y,
                                include_regularization=True)
        loss = data_loss + regularization_loss

        # Get predictions and calculate an accuracy
        predictions = self.output_layer_activation.predictions(
                          output)
        accuracy = self.accuracy.calculate(predictions, y)

        # Perform backward pass
        self.backward(output, y)

        # Optimize (update parameters)
        self.optimizer.pre_update_params()
        for layer in self.trainable_layers:
            self.optimizer.update_params(layer)
        self.optimizer.post_update_params()

        # Print a summary
        if not epoch % print_every:
            print(f'epoch: {epoch}, ' +
                  f'acc: {accuracy:.3f}, ' +
                  f'loss: {loss:.3f} (' +
                  f'data_loss: {data_loss:.3f}, ' +
                  f'reg_loss: {regularization_loss:.3f}), ' +
                  f'lr: {self.optimizer.current_learning_rate}')

    # If there is the validation data
    if validation_data is not None:

        # For better readability
        X_val, y_val = validation_data

        # Perform the forward pass
        output = self.forward(X_val, training=False)

        # Calculate the loss
        loss = self.loss.calculate(output, y_val)

        # Get predictions and calculate an accuracy
        predictions = self.output_layer_activation.predictions(
                          output)
        accuracy = self.accuracy.calculate(predictions, y_val)
```

```
        # Print a summary
        print(f'validation, ' +
              f'acc: {accuracy:.3f}, ' +
              f'loss: {loss:.3f}')

    # Performs forward pass
    def forward(self, X, training):

        # Call forward method on the input layer
        # this will set the output property that
        # the first layer in "prev" object is expecting
        self.input_layer.forward(X, training)

        # Call forward method of every object in a chain
        # Pass output of the previous object as a parameter
        for layer in self.layers:
            layer.forward(layer.prev.output, training)

        # "layer" is now the last object from the list,
        # return its output
        return layer.output

    # Performs backward pass
    def backward(self, output, y):

        # If softmax classifier
        if self.softmax_classifier_output is not None:
            # First call backward method
            # on the combined activation/loss
            # this will set dinputs property
            self.softmax_classifier_output.backward(output, y)

            # Since we'll not call backward method of the last layer
            # which is Softmax activation
            # as we used combined activation/loss
            # object, let's set dinputs in this object
            self.layers[-1].dinputs = \
                self.softmax_classifier_output.dinputs

            # Call backward method going through
            # all the objects but last
            # in reversed order passing dinputs as a parameter
            for layer in reversed(self.layers[:-1]):
                layer.backward(layer.next.dinputs)

            return
```

```
        # First call backward method on the loss
        # this will set dinputs property that the last
        # layer will try to access shortly
        self.loss.backward(output, y)

        # Call backward method going through all the objects
        # in reversed order passing dinputs as a parameter
        for layer in reversed(self.layers):
            layer.backward(layer.next.dinputs)


# Create dataset
X, y = spiral_data(samples=1000, classes=3)
X_test, y_test = spiral_data(samples=100, classes=3)

# Instantiate the model
model = Model()

# Add layers
model.add(Layer_Dense(2, 512, weight_regularizer_l2=5e-4,
                              bias_regularizer_l2=5e-4))
model.add(Activation_ReLU())
model.add(Layer_Dropout(0.1))
model.add(Layer_Dense(512, 3))
model.add(Activation_Softmax())

# Set loss, optimizer and accuracy objects
model.set(
    loss=Loss_CategoricalCrossentropy(),
    optimizer=Optimizer_Adam(learning_rate=0.05, decay=5e-5),
    accuracy=Accuracy_Categorical()
)

# Finalize the model
model.finalize()

# Train the model
model.train(X, y, validation_data=(X_test, y_test),
            epochs=10000, print_every=100)
```

Supplementary Material: https://nnfs.io/ch18
Chapter code, further resources, and errata for this chapter.

Chapter 19

A Real Dataset

In practice, deep learning tends to involve massive datasets (often terabytes or more in size), and models can take days, weeks, or even months to train. This is why, so far, we've used programmatically-generated datasets to keep things manageable and fast, while we learn the math and other aspects of deep learning. The main objective of this book is to teach how neural networks work, rather than the application of deep learning to various problems. That said, we'll explore a more realistic dataset now, since this will present new challenges to deep learning that we've not yet had to consider.

If you have explored deep learning before reading this book, you have likely become acquainted (and possibly annoyed) with the MNIST dataset, which is a dataset of images of handwritten digits (0 through 9) at a resolution of 28x28 pixels. It's a relatively small dataset and is reasonably easy for models to learn. This dataset became the "hello world" of deep learning, and it was once a benchmark of machine learning algorithms. The problem with this dataset is that it's comically easy to get 99%+ accuracy on the MNIST dataset, and doesn't provide much room for learning how various parameters impact learning. In 2017, however, a company called Zalando released a dataset (https://arxiv.org/abs/1708.07747) called Fashion MNIST (https://github.com/zalandoresearch/fashion-mnist), which is a drop-in replacement for the regular MNIST dataset.

The Fashion MNIST dataset is a collection of 60,000 training samples and 10,000 testing samples of 28x28 images of 10 various clothing items like shoes, boots, shirts, bags, and more. We'll see some examples shortly, but first, we need the actual data. Since the original dataset consists of binary files containing encoded (in a specific format) image data, for this book, we have prepared and are hosting a preprocessed dataset consisting of .png images instead. It is usually wise to use lossless compression for images since lossy compression, like JPEG, affects images by changing their data). These images are also grouped by labels and separated into training and testing groups. The samples are the images of articles of clothing, and the labels are the classifications. Here are the numeric labels and their respective descriptions:

Label	**Description**
0	T-shirt/top
1	Trouser
2	Pullover
3	Dress
4	Coat
5	Sandal
6	Shirt
7	Sneaker
8	Bag
9	Ankle boot

Data preparation

First, we will retrieve the data from the nnfs.io site. Let's define the URL of the dataset, a filename to save it locally to and the folder for the extracted images:

```
URL = 'https://nnfs.io/datasets/fashion_mnist_images.zip'
FILE = 'fashion_mnist_images.zip'
FOLDER = 'fashion_mnist_images'
```

Next, download the compressed data (if the file is absent under the given path) using the *urllib*, a standard Python library:

```
import os
import urllib
import urllib.request

if not os.path.isfile(FILE):
    print(f'Downloading {URL} and saving as {FILE}...')
    urllib.request.urlretrieve(URL, FILE)
```

From here, we'll unzip the files using another standard Python library called *zipfile*. We'll use the context wrapper (the `with` keyword, which will open and close file for us) to gather the zipped file handler and extract all of the included files using `.extractall` and the given `FOLDER`:

```
from zipfile import ZipFile

print('Unzipping images...')
with ZipFile(FILE) as zip_images:
    zip_images.extractall(FOLDER)
```

The full code for retrieving the data:

```
from zipfile import ZipFile
import os
import urllib
import urllib.request
```

```
URL = 'https://nnfs.io/datasets/fashion_mnist_images.zip'
FILE = 'fashion_mnist_images.zip'
FOLDER = 'fashion_mnist_images'

if not os.path.isfile(FILE):
    print(f'Downloading {URL} and saving as {FILE}...')
    urllib.request.urlretrieve(URL, FILE)

print('Unzipping images...')
with ZipFile(FILE) as zip_images:
    zip_images.extractall(FOLDER)

print('Done!')
```

Running this:

```
>>>
Downloading https://nnfs.io/datasets/fashion_mnist_images.zip and saving as
fashion_mnist_images.zip...
Unzipping images...
Done!
```

You should now have a directory called *fashion_mnist_images*, containing *test* and *train* directories and the data license. Inside of both the *test* and *train* directories, we have ten subdirectories, numbered 0 through 9. These numbers are classifications that correspond to the images within. For example, if we open directory *0*, we can see these are images of shirts with either short sleeves or no sleeves at all. For example:

Fig 19.01: Example t-shirt image from the Fasion MNIST dataset.

Inside directory *7*, we have non-boot shoes, or *sneakers* as the creators of this dataset have classified them. For example:

Fig 19.02: Example sneaker image from the Fasion MNIST dataset.

It's common practice to grayscale (go from 3-channel RGB values per pixel to a single black to white range of 0-255 per pixel) images, though these images are already grayscaled. It is also a common practice to resize images to **normalize** their dimensions, but once again, the Fashion MNIST dataset is prepared so that all the images are already all the same shape (28x28).

Data loading

Next, we have to read these images into Python and associate the image (pixel) data with the respective labels. We can access the directories as follows:

```
import os

labels = os.listdir('fashion_mnist_images/train')
print(labels)

>>>
['0', '1', '2', '3', '4', '5', '6', '7', '8', '9']
```

Since the subdirectory names are the labels themselves, we can reference individual samples for each class by looking through the files in each numbered subdirectory:

```
files = os.listdir('fashion_mnist_images/train/0')
print(files[:10])
print(len(files))
```

```
>>>
['0000.png', '0001.png', '0002.png', '0003.png', '0004.png', '0005.png',
'0006.png', '0007.png', '0008.png', '0009.png']
6000
```

As you can see, we have 6,000 samples of class 0. In total, we have 60,000 samples -- 6,000 per classification. Meaning our dataset is also already **balanced**; each class occurs with the same frequency. If a dataset is not already balanced, the neural network will likely become biased to predict the class containing the most images. This is because neural networks fundamentally seek out the steepest and quickest gradient descent to decrease loss, which might lead to a local minimum making the model unable to find the global loss minimum. We have a total of 10 classes here, so a random prediction, with a balanced dataset, would have an accuracy of about 10%.

Imagine, however, if the balance of classes in the dataset was 64% for class 0 and 4% for 1 through 9. The neural network might very quickly learn to always predict class 0. Though the model would rapidly decrease loss initially, it would likely remain stuck predicting class 0 with an accuracy closer to 64%. In such a case, we're better off trimming away samples from the high-frequency classes so that we have the same number of samples per class.

Another option is to use class weights, weighting classes that occur more often with a fraction of 1 when accounting for loss. Though we have never seen this work well in practice. With image data, another option would be to augment the samples through actions like cropping, rotation, and maybe flipping horizontally or vertically. Before applying such transformations, ensure they will generate valid samples that fit your objectives. Luckily for us, we don't need to worry about that since the Fashion MNIST data are indeed perfectly balanced. We'll now explore our data by looking at individual samples. To handle image data, we're going to make use of the Python package containing OpenCV, under the **cv2** library, which you can install with pip:

```
pip install opencv-python
```

And to load the image data:

```
import cv2

image_data = cv2.imread('fashion_mnist_images/train/7/0002.png',
                        cv2.IMREAD_UNCHANGED)
print(image_data)
```

We read in images with `cv2.imread()`, where the first parameter is the path to the image. The `cv2.IMREAD_UNCHANGED` argument notifies the `cv2` package that we intend to read in these images in the same format as they were saved (grayscale in this case). By default, OpenCV will convert these images to use all 3 color channels, even though this is a grayscale image. As a result, we have a 2D array of numbers — grayscale pixel values. If we formatted this otherwise messy

array before printing with the following line of the code, which will inform NumPy, since the loaded image is a numpy array object, to print more characters in a line:

```
import numpy as np
np.set_printoptions(linewidth=200)
```

We'd still likely be able to identify the subject:

```
[[  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0   1   0   0   0  49 135 182 150  59   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0  78 255 220 212 219 255 246 191 155  87   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   1   0   0  57 206 215 203 191 203 212 216 217 220 211  15   0]
 [  0   0   0   0   0   0   0   0   0   0   1   0   0   0  58 231 220 210 199 209 218 218 217 208 200 215  56   0]
 [  0   0   0   0   1   2   0   0   4   0   0   0   0 145 213 207 199 187 203 210 216 217 215 215 206 215 130   0]
 [  0   0   0   0   1   2   4   0   0   0   3 105 225 205 190 201 210 214 213 215 215 212 211 208 205 207 218   0]
 [  1   5   7   0   0   0   0   0  52 162 217 189 174 157 187 198 202 217 220 223 224 222 217 211 217 201 247  65]
 [  0   0   0   0   0   0  21  72 185 189 171 171 185 203 200 207 208 209 214 219 222 222 224 215 218 211 212 148]
 [  0  70 114 129 145 159 179 196 172 176 185 196 199 206 201 210 212 213 216 218 219 217 212 207 208 200 198 173]
 [  0 122 158 184 194 192 193 196 203 209 211 211 215 218 221 222 226 227 227 226 226 223 222 216 211 208 216 185]
 [ 21   0   0  12  48  82 123 152 170 184 195 211 225 232 233 237 242 242 240 240 238 236 222 209 200 193 185 106]
 [ 26  47  54  18   5   0   0   0   0   0   0   0   0   2   4   6   9   9   8   9   6   6   4   2   0   0   0   0]
 [  0  10  27  45  55  59  57  50  44  51  58  62  65  56  54  57  59  61  60  63  68  67  66  73  77  74  65  39]
 [  0   0   0   0   4   9  18  23  26  25  23  25  29  37  38  37  39  36  29  31  33  34  28  24  20  14   7   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0]]
```

In this case, it's a sneaker. Rather than formatting the raw values to see what we're looking at this way, we can use Matplotlib to visualize this. For example:

```
import matplotlib.pyplot as plt
plt.imshow(image_data)
plt.show()

>>>
```

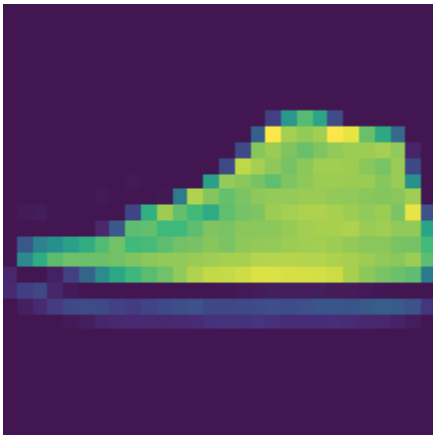


Fig 19.03: Sneaker image shown with Matplotlib after loading with Python

We can check another sample:

```
import matplotlib.pyplot as plt, cv2
image_data = cv2.imread('fashion_mnist_images/train/4/0011.png',
                        cv2.IMREAD_UNCHANGED)
plt.imshow(image_data)
plt.show()

>>>
```

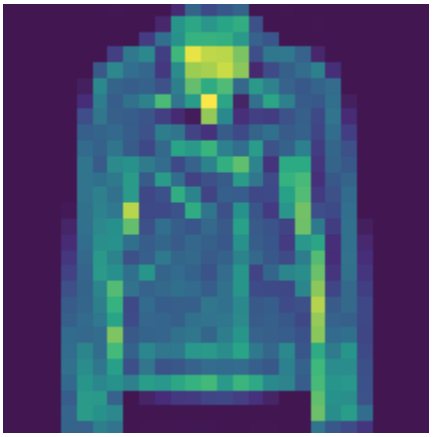


Fig 19.04: Jacket image shown with Matplotlib after loading with Python

It looks like a jacket. If we check our table from before, class 4 is "coat." You might wonder about the strange coloring, but this is just the default from matplotlib not expecting grayscale. We can notify Matplotlib that this is grayscale by specifying a *cmap* (colormap) during the `plt.imshow()` call:

```
import matplotlib.pyplot as plt, cv2
image_data = cv2.imread('fashion_mnist_images/train/4/0011.png',
                        cv2.IMREAD_UNCHANGED)
plt.imshow(image_data, cmap='gray')
plt.show()

>>>
```

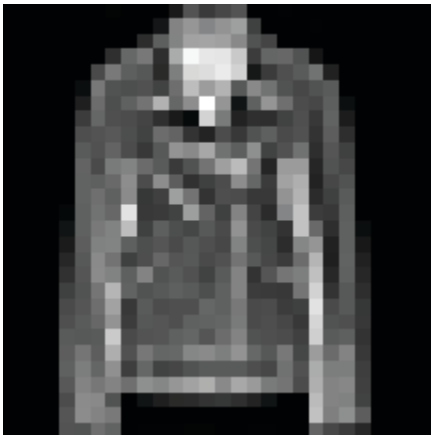


Fig 19.05: Grayscaled jacket image

Now we can iterate over all of the samples, load them, and put them into the input data (**X**) and targets (**y**) lists. First, we scan the train folder, which, as noted before, contains folders named from 0 to 9, which also act as sample labels. We iterate through these folders and the images inside them, appending them to a list variable (named `X`) along with their respective labels to another list variable (named `y`), forming our samples and ground-truth, or target labels:

```
# Scan all the directories and create a list of labels
labels = os.listdir('fashion_mnist_images/train')

# Create lists for samples and labels
X = []
y = []

# For each label folder
for label in labels:
    # And for each image in given folder
    for file in os.listdir(os.path.join(
                    'fashion_mnist_images', 'train', label
    )):
        # Read the image
        image = cv2.imread(os.path.join(
                    'fashion_mnist_images/train', label, file
                ), cv2.IMREAD_UNCHANGED)

        # And append it and a label to the lists
        X.append(image)
        y.append(label)
```

We need to do this operation on both the testing and training data. Again, they are already nicely split up for us. Many times, you will need to separate your data into train and test groups on your own. We'll convert the above code into a function to prevent duplicating the code for training and testing directories. This function will take a dataset type as a parameter: train or test, along with the path where those datasets are located:

```python
import numpy as np
import cv2
import os


# Loads a MNIST dataset
def load_mnist_dataset(dataset, path):

    # Scan all the directories and create a list of labels
    labels = os.listdir(os.path.join(path, dataset))

    # Create lists for samples and labels
    X = []
    y = []

    # For each label folder
    for label in labels:
        # And for each image in given folder
        for file in os.listdir(os.path.join(path, dataset, label)):
            # Read the image
            image = cv2.imread(os.path.join(
                        path, dataset, label, file
                    ), cv2.IMREAD_UNCHANGED)

            # And append it and a label to the lists
            X.append(image)
            y.append(label)

    # Convert the data to proper numpy arrays and return
    return np.array(X), np.array(y).astype('uint8')
```

Since `X` has been defined as a list, and we are adding images represented as NumPy arrays to this list, we'll call `np.array()` on `X` at the end to transform it from a list into a proper NumPy array. We will do the same with the labels (`y`) since they are a list of numbers and additionally inform NumPy that our labels are integer (not float) values.
Then we can write a function that will create and return our train and test data:

```python
# MNIST dataset (train + test)
def create_data_mnist(path):

    # Load both sets separately
    X, y = load_mnist_dataset('train', path)
    X_test, y_test = load_mnist_dataset('test', path)

    # And return all the data
    return X, y, X_test, y_test
```

Code up to this point for our new data:

```
import numpy as np
import cv2
import os


# Loads a MNIST dataset
def load_mnist_dataset(dataset, path):

    # Scan all the directories and create a list of labels
    labels = os.listdir(os.path.join(path, dataset))

    # Create lists for samples and labels
    X = []
    y = []

    # For each label folder
    for label in labels:
        # And for each image in given folder
        for file in os.listdir(os.path.join(path, dataset, label)):
            # Read the image
            image = cv2.imread(os.path.join(path, dataset, label, file),
cv2.IMREAD_UNCHANGED)

            # And append it and a label to the lists
            X.append(image)
            y.append(label)

    # Convert the data to proper numpy arrays and return
    return np.array(X), np.array(y).astype('uint8')


# MNIST dataset (train + test)
def create_data_mnist(path):

    # Load both sets separately
    X, y = load_mnist_dataset('train', path)
    X_test, y_test = load_mnist_dataset('test', path)

    # And return all the data
    return X, y, X_test, y_test
```

Thanks to this function, we can load in our data by doing:

```
# Create dataset
X, y, X_test, y_test = create_data_mnist('fashion_mnist_images')
```

Data preprocessing

Next, we will **scale** the data (not the images, but the data representing them, the numbers). Neural networks tend to work best with data in the range of either 0 to 1 or -1 to 1. Here, the image data are within the range 0 to 255. We have a decision to make with *how* to scale these data. Usually, this process will be some experimentation and trial and error. For example, we could scale images to be between the range of -1 and 1 by taking each pixel value, subtracting half the maximum of all pixel values (i.e., 255/2 = 127.5), then dividing by this same half to produce a range bounded by -1 and 1. We could also scale our data between 0 and 1 by simply dividing it by 255 (the maximum value). To start, we opt to scale between -1 and 1. Before we do that, we have to change the datatype of the NumPy array, which is currently `uint8` (unsigned integer, holds integer values in the range of 0 to 255). If we don't do this, NumPy will convert it to a `float64` data type while our intention is to use `float32`, a 32-bit float value. This can be achieved by calling `.astype(np.float32)` on a NumPy array object. We will leave the labels untouched:

```
# Create dataset
X, y, X_test, y_test = create_data_mnist('fashion_mnist_images')

# Scale features
X = (X.astype(np.float32) - 127.5) / 127.5
X_test = (X_test.astype(np.float32) - 127.5) / 127.5
```

Ensure that you scale both training and testing data using identical methods. Later, when making predictions, you will also need to scale the input data for inference. It can be easy to forget to scale your data in these different places. You also want to be sure that any preprocessing, like scaling, is informed only by your training dataset. In this example, we knew the minimum (min) and maximum (max) values would be 0 and 255, and performed linear scaling. You will often need to first query your dataset for min and max values for use in scaling. You may also use other methods to scale if your dataset has extreme outliers, as min/max may not work well. If this is the case, you might use some combination of the average value and standard deviation to create your scaling method.

A common mistake when scaling is to allow the testing dataset to inform transformations made to the training dataset. There is only one exception to this rule, which is when the data is being scaled linearly, for example, by the mentioned division by a constant number. Any non-linear scaling function could possibly leak information from testing or validation data into training data. Any preprocessing rules should be derived without knowledge of the testing dataset, but then

applied to the testing set. For example, your entire dataset might have a min value of 0 and a max of 125, while the training dataset only has a min of 0 and a max of 100. You will still use the 100 value when scaling your testing dataset. This means that your testing dataset might not fit neatly between the bounds of -1 and 1 after scaling, but this usually should not be a problem. In the case of a bigger difference, you can additionally scale the data linearly by dividing them by some number.

Back to our data, let's check that our data have been scaled:

```
print(X.min(), X.max())
```

```
>>>
-1.0 1.0
```

Next, we check the shape of our input data:

```
print(X.shape)
```

```
>>>
(60000, 28, 28)
```

Our Dense layers work on batches of 1-dimensional vectors. They cannot operate on images shaped as a 28x28, 2-dimensional array. We need to take these 28x28 images and **flatten** them, which means to take every row of an image array and append it to the first row of that array, transforming the 2-dimensional (2D) array of an image into a 1-dimensional (1D) array (i.e., vector), or in other words, we could see this as unwrapping numbers in a 2D array to a list-like form. There are neural network models called convolutional neural networks that will allow you to pass 2D image data "as is," but a dense neural network like we have here expects samples that are 1D. Even in convolutional neural networks, you will usually need to flatten data before feeding them to an output layer or a dense layer. To flatten an array with NumPy, we can reshape using `-1` as a first shape dimension, which means "however many elements are there" and effectively put them all in the first dimension making a flat 1D array. For an example of this concept:

```
example = np.array([[1, 2], [3, 4]])
flattened = example.reshape(-1)

print(example)
print(example.shape)

print(flattened)
print(flattened.shape)
```

```
>>>
[[1 2]
 [3 4]]
(2, 2)
[1 2 3 4]
(4,)
```

We could also use `np.flatten()`, but our intention differs with a batch of samples. In the case of our samples, we still wish to retain all 60,000 of them, so we'd like to reshape our training data to be `(60000, -1)`. This will notify NumPy that we wish to keep the 60,000 samples (first dimension), but flatten the rest (-1 as the second dimension means that we want to put all of the sample data in this single dimension into a 1D array). This will create 60,000 samples of 784 features each. The 784 features are the result of 28·28. To do this, we'll use the number of samples from the training (`X.shape[0]`) and testing (`X_test.shape[0]`) datasets respectively and reshape them:

```
# Reshape to vectors
X = X.reshape(X.shape[0], -1)
X_test = X_test.reshape(X_test.shape[0], -1)
```

You can achieve the same result by explicitly defining the shape, instead of relying on NumPy inference:

```
.reshape(X.shape[0], X.shape[1]*X.shape[2])
```

which would be more explicit, but we find this to be less legible.

Data Shuffling

Our dataset currently consists of samples and their target classifications, in order, from 0 to 9. To illustrate, we can query our *y* data at various points. The first 6000 will all be 0. For instance:

```
print(y[0:10])

>>>
[0 0 0 0 0 0 0 0 0 0]
```

If we then query a bit later:

```
print(y[6000:6010])

>>>
[1 1 1 1 1 1 1 1 1 1]
```

This is a problem if we train our network with data in this order; for the same reason, an imbalanced dataset is problematic.

While we train on the first 6,000 samples, the model will learn that the quickest way to reduce loss is to always predict a 0, since it'll see several batches of the data with class 0 only. Then, between 6,000 and 12,000, the loss will initially spike as the label will change and the model will be predicting, incorrectly, still label 0, and will likely learn that now it needs to always predict class 1 (as that's all it sees in batches of labels which we optimize for). The model will cycle local minimums following whichever label is currently being repeated over batches and will most likely never find a global minimum. This process will continue until we get through all samples, repeating however many epochs we selected.

Preferably, there would be many classifications (ideally some from each class) of samples per fitment to keep the model from becoming biased towards any single class, simply because that's the class it's been seeing lately. Thus, we often will randomly shuffle the data. We didn't need to shuffle our previous training data, like the spiral data, since we were training on the entire dataset at once anyway, rather than individual batches. With this larger dataset that we're training on in batches, we want to shuffle the data, as it's currently organized in order of chunks of 6,000 samples per label. When shuffling, we want to ensure that we shuffle the sample and target

arrays the same; otherwise, we'll have a very confused (and very wrong, in most cases) model as labels will no longer match samples. Hence, we cannot simply call `shuffle()` on both of them separately. There are many ways to achieve this, but what we'll do is gather all of the "keys," which are the same for samples and targets, and then shuffle them.

These keys will be values from 0 to 59999 in this case.

```
keys = np.array(range(X.shape[0]))
print(keys[:10])

>>>
array([0 1 2 3 4 5 6 7 8 9])
```

Then, we can shuffle these keys:

```
import nnfs

nnfs.init()

np.random.shuffle(keys)
print(keys[:10])

>>>
[ 3048 19563 58303  8870 40228 31488 21860 56864   845 25770]
```

Now, this is essentially the new order of indexes, which we can then apply by doing:

```
X = X[keys]
y = y[keys]
```

This tells NumPy to return values of given indices as we would normally index NumPy arrays, but we're using a variable that keeps a list of randomly ordered indices. Then we can check a slice of targets:

```
print(y[:15])

>>>
[0 3 9 1 6 5 3 9 0 4 8 9 0 6 6]
```

They seem to be shuffled. We can check individual samples as well:

```
import matplotlib.pyplot as plt
plt.imshow((X[8].reshape(28, 28)))  # Reshape as image is a vector already
plt.show()
```

```
>>>
```

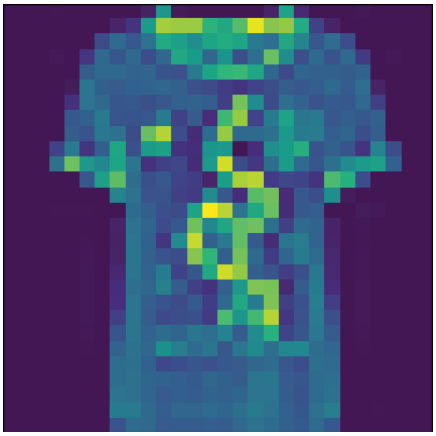


Fig 19.06: Random (shirt) image after shuffling

We can then check the class at the same index:

```
print(y[8])
```

```
>>>
0
```

Class 0 is indeed "shirt," and so these data do look properly shuffled. You may check a few more manually to ensure your data are as expected. If the model does not train or appears to be misbehaving, you will want to double-check how you preprocessed the data.

Batches

So far, we've trained our models by feeding the entire dataset as a single "batch" through the model. We discussed earlier in Chapter 2 why it is preferred to do more than 1 sample at a time, but is there a batch size that would be too big? Our dataset has been small enough for us to get away with the behavior of feeding the entire dataset at once, but real-world datasets can often be terabytes or more in size, which is a nonstarter for the majority of computers to process as a single batch. A batch is a slice of a fixed size from the data. When we train with batches, we iterate through the dataset in a chunk, or "batch, " of data at a time, performing a forward pass, loss calculation, backward pass, and optimization. If the data have been shuffled, and each batch is large enough and somewhat representative of the dataset, it is a fair assumption that each gradient of each batch should be a good approximation of the direction towards a global minimum. If the batch is too small, the direction of the gradient descent can fluctuate too much from batch to batch, causing the model to take a long time to train.

Common batch sizes range between 32 and 128 samples. You can go smaller if you're having trouble fitting everything into memory, or larger if you want training to go faster, but this range is the typical range of batch sizes. You will usually see accuracy and loss improvements by increasing the batch size from say 2 to 8, or 8 to 32. At some point, however, you will see diminishing returns regarding accuracy and loss if you keep increasing the batch size. Additionally, training with large batch sizes will become slow compared to the speed achievable with smaller batch sizes — like our examples earlier with the spiral data, requiring 10 thousand epochs to train! As is often the case with neural networks, it's a lot of trial and error with your specific data and model. For example, imagine we select a batch size of 128, and we opt to do 10 epochs.

This means, for each epoch, we will iterate over our data, fitting 128 samples at a time to train our model. Each batch of samples being trained is referred to as a **step**. We can calculate the number of **steps** by dividing the number of samples by the batch size:

```
steps = X.shape[0] // BATCH_SIZE
```

We use the integer division operator, `//`, (instead of the floating-point division operator, `/`) to return an integer, as the number of steps cannot contain a fraction. This is the number of iterations that we'll be making per epoch in a loop. If there are some straggler samples left over, we can add

them in by simply adding one more step:

```
if steps * BATCH_SIZE < X.shape[0]:
    steps += 1
```

Why we add this 1 can be presented in a simple example:

```
batch_size = 2

X = [1, 2, 3, 4]
print(len(X) // batch_size)

>>>
2


X = [1, 2, 3, 4, 5]
print(len(X) // batch_size)

>>>
2
```

Integer division rounds down; thus, if there are any samples left, we'll add 1 to form the last batch with the remainder.

An example of code leading up to training a model using batches:

```
import nnfs
from nnfs.datasets import spiral_data


nnfs.init()

# Create dataset
X, y = spiral_data(samples=100, classes=3)

EPOCHS = 10
BATCH_SIZE = 128  # We take 128 samples at once

# Calculate number of steps
steps = X.shape[0] // BATCH_SIZE
# Dividing rounds down. If there are some remaining data,
# but not a full batch, this won't include it.
# Add 1 to include the remaining samples in 1 more step.
if steps * BATCH_SIZE < X.shape[0]:
    steps += 1
```

```
for epoch in range(EPOCHS):
    for step in range(steps):
        batch_X = X[step*BATCH_SIZE:(step+1)*BATCH_SIZE]
        batch_y = y[step*BATCH_SIZE:(step+1)*BATCH_SIZE]

        # Now we perform forward pass, loss calculation,
        # backward pass and update parameters
```

We loaded the dataset, defined the number of epochs and a batch size, then calculated the number of steps. Next, we have two loops — an outer one over the epochs and an inner one over the steps. During each step in each epoch, we're selecting a slice of the training data. Now that we know how to train the model in batches, we're interested in the training loss and accuracy for each step and epoch. So far, we've only been calculating loss per fit, but recall that we fitted against the entire dataset at once. Now, we'll be interested in both batch-wise statistics and epoch-wise. For the overall loss and accuracy, we want to calculate a sample-wise average. To do this, we will accumulate the sum of losses from all epoch batches and counts to calculate the mean value at the end of each epoch. We'll start in the common `Loss` class' `calculate` method by adding:

```
        # Add accumulated sum of losses and sample count
        self.accumulated_sum += np.sum(sample_losses)
        self.accumulated_count += len(sample_losses)
```

Making the full `calculate` method in the `Loss` class:

```
    # Calculates the data and regularization losses
    # given model output and ground truth values
    def calculate(self, output, y, *, include_regularization=False):

        # Calculate sample losses
        sample_losses = self.forward(output, y)

        # Calculate mean loss
        data_loss = np.mean(sample_losses)

        # Add accumulated sum of losses and sample count
        self.accumulated_sum += np.sum(sample_losses)
        self.accumulated_count += len(sample_losses)

        # If just data loss - return it
        if not include_regularization:
            return data_loss

        # Return the data and regularization losses
        return data_loss, self.regularization_loss()
```

We're saving the sum and the count so we can calculate the mean at any point. To do that, we'll add a new method called `calculate_accumulated` inside the `Loss` class:

```
# Calculates accumulated loss
def calculate_accumulated(self, *, include_regularization=False):

    # Calculate mean loss
    data_loss = self.accumulated_sum / self.accumulated_count

    # If just data loss - return it
    if not include_regularization:
        return data_loss

    # Return the data and regularization losses
    return data_loss, self.regularization_loss()
```

This method can also return the regularization loss if *`include_regularization`* is set to `True`. The regularization loss does not need to be accumulated as it's calculated from the current state of layer parameters, at the time it's called. We'll be using this ability during training, but not while evaluating and predicting; we'll discuss this in more detail shortly.

Finally, in order to reset the sum and count values for a new epoch, we'll add one last method:

```
# Reset variables for accumulated loss
def new_pass(self):
    self.accumulated_sum = 0
    self.accumulated_count = 0
```

Making our full common `Loss` class:

```
# Common loss class
class Loss:

    # Regularization loss calculation
    def regularization_loss(self):

        # 0 by default
        regularization_loss = 0

        # Calculate regularization loss
        # iterate all trainable layers
        for layer in self.trainable_layers:

            # L1 regularization - weights
            # calculate only when factor greater than 0
            if layer.weight_regularizer_l1 > 0:
                regularization_loss += layer.weight_regularizer_l1 * \
                                       np.sum(np.abs(layer.weights))

            # L2 regularization - weights
            if layer.weight_regularizer_l2 > 0:
                regularization_loss += layer.weight_regularizer_l2 * \
                                       np.sum(layer.weights * \
                                              layer.weights)

            # L1 regularization - biases
            # calculate only when factor greater than 0
            if layer.bias_regularizer_l1 > 0:
                regularization_loss += layer.bias_regularizer_l1 * \
                                       np.sum(np.abs(layer.biases))

            # L2 regularization - biases
            if layer.bias_regularizer_l2 > 0:
                regularization_loss += layer.bias_regularizer_l2 * \
                                       np.sum(layer.biases * \
                                              layer.biases)

        return regularization_loss

    # Set/remember trainable layers
    def remember_trainable_layers(self, trainable_layers):
        self.trainable_layers = trainable_layers
```

```
    # Calculates the data and regularization losses
    # given model output and ground truth values
    def calculate(self, output, y, *, include_regularization=False):

        # Calculate sample losses
        sample_losses = self.forward(output, y)

        # Calculate mean loss
        data_loss = np.mean(sample_losses)

        # Add accumulated sum of losses and sample count
        self.accumulated_sum += np.sum(sample_losses)
        self.accumulated_count += len(sample_losses)

        # If just data loss - return it
        if not include_regularization:
            return data_loss

        # Return the data and regularization losses
        return data_loss, self.regularization_loss()

    # Calculates accumulated loss
    def calculate_accumulated(self, *, include_regularization=False):

        # Calculate mean loss
        data_loss = self.accumulated_sum / self.accumulated_count

        # If just data loss - return it
        if not include_regularization:
            return data_loss

        # Return the data and regularization losses
        return data_loss, self.regularization_loss()

    # Reset variables for accumulated loss
    def new_pass(self):
        self.accumulated_sum = 0
        self.accumulated_count = 0
```

We'll want to implement the same things for the `Accuracy` class now:

```
# Common accuracy class
class Accuracy:

    # Calculates an accuracy
    # given predictions and ground truth values
    def calculate(self, predictions, y):

        # Get comparison results
        comparisons = self.compare(predictions, y)
        # Calculate an accuracy
        accuracy = np.mean(comparisons)

        # Add accumulated sum of matching values and sample count
        self.accumulated_sum += np.sum(comparisons)
        self.accumulated_count += len(comparisons)

        # Return accuracy
        return accuracy

    # Calculates accumulated accuracy
    def calculate_accumulated(self):

        # Calculate an accuracy
        accuracy = self.accumulated_sum / self.accumulated_count

        # Return the data and regularization losses
        return accuracy

    # Reset variables for accumulated accuracy
    def new_pass(self):
        self.accumulated_sum = 0
        self.accumulated_count = 0
```

Here, we've added setting the `accumulated_sum` and `accumulated_count` properties in the `calculate` method for the epoch accuracy calculation, added a new `calculate_accumulated` method that returns this accuracy, and finally added a `new_pass` method to reset the `accumulated_sum` and `accumulated_count` values that we'll use at the beginning of each epoch.

Now, we'll modify the `train` method for our `Model` class. First, we'll add a new parameter called *`batch_size`*:

```
    def train(self, X, y, *, epochs=1, batch_size=None,
              print_every=1, validation_data=None):
```

We'll default this parameter to None, which means to use the entire dataset as the batch. In this case, training will take 1 step per epoch, where that step consists of feeding all the data through the network at once.

```
# Default value if batch size is not set
train_steps = 1

# If there is validation data passed,
# set default number of steps for validation as well
if validation_data is not None:
    validation_steps = 1

    # For better readability
    X_val, y_val = validation_data
```

As discussed, most "real life" datasets will require a batch size smaller than that of all samples. We'll handle that using the method that we described earlier: performing integer division of the number of all samples by the batch size and eventually adding 1 to include any remaining samples that did not form a full batch (we'll do that for both training and validation data):

```
# Calculate number of steps
if batch_size is not None:
    train_steps = len(X) // batch_size
    # Dividing rounds down. If there are some remaining
    # data, but not a full batch, this won't include it
    # Add 1 to include this not full batch
    if train_steps * batch_size < len(X):
        train_steps += 1

    if validation_data is not None:
        validation_steps = len(X_val) // batch_size
        # Dividing rounds down. If there are some remaining
        # data, but nor full batch, this won't include it
        # Add 1 to include this not full batch
        if validation_steps * batch_size < len(X_val):
            validation_steps += 1
```

Next, starting at the top, we'll modify the loop over epochs to print an epoch number and then reset the accumulated epoch loss and accuracy values. Then, inside of here, we'll add a new loop that will iterate over steps in the epoch.

```
    # Print epoch number
    print(f'epoch: {epoch}')

    # Reset accumulated values in loss and accuracy objects
    self.loss.new_pass()
    self.accuracy.new_pass()
```

```
# Iterate over steps
for step in range(train_steps):
```

Inside of each step, we'll need to grab the **batch** of data that we'll use to train — either the full dataset if the `batch_size` parameter is still the default `None` or a slice of size `batch_size`:

```
# If batch size is not set -
# train using one step and full dataset
if batch_size is None:
    batch_X = X
    batch_y = y

# Otherwise slice a batch
else:
    batch_X = X[step*batch_size:(step+1)*batch_size]
    batch_y = y[step*batch_size:(step+1)*batch_size]
```

With each of these batches, we fit and print information, similar to how we were fitting per epoch. The difference now is we use `batch_X` instead of `X` and `batch_y` instead of `y`. The other change is the `if` statement for the summary printing that will account for steps instead of epochs:

```
# Perform the forward pass
output = self.forward(batch_X, training=True)
# Calculate loss
data_loss, regularization_loss = \
    self.loss.calculate(output, batch_y,
                        include_regularization=True)
loss = data_loss + regularization_loss
# Get predictions and calculate an accuracy
predictions = self.output_layer_activation.predictions(
                  output)
accuracy = self.accuracy.calculate(predictions,
                                   batch_y)
# Perform backward pass
self.backward(output, batch_y)

# Optimize (update parameters)
self.optimizer.pre_update_params()
for layer in self.trainable_layers:
    self.optimizer.update_params(layer)
self.optimizer.post_update_params()

# Print a summary
if not step % print_every or step == train_steps - 1:
    print(f'step: {step}, ' +
          f'acc: {accuracy:.3f}, ' +
          f'loss: {loss:.3f} (' +
```

```
                    f'data_loss: {data_loss:.3f}, ' +
                    f'reg_loss: {regularization_loss:.3f}), ' +
                    f'lr: {self.optimizer.current_learning_rate}')
```

Then we'd like to print information like accuracy and loss per epoch:

```
# Get and print epoch loss and accuracy
epoch_data_loss, epoch_regularization_loss = \
    self.loss.calculate_accumulated(
        include_regularization=True)
epoch_loss = epoch_data_loss + epoch_regularization_loss
epoch_accuracy = self.accuracy.calculate_accumulated()
print(f'training, ' +
      f'acc: {epoch_accuracy:.3f}, ' +
      f'loss: {epoch_loss:.3f} (' +
      f'data_loss: {epoch_data_loss:.3f}, ' +
      f'reg_loss: {epoch_regularization_loss:.3f}), ' +
      f'lr: {self.optimizer.current_learning_rate}')
```

If the batch size is set, the chances are that our validation data will be larger than this batch size, so we need to add batching for the validation data as well:

```
# If there is the validation data
if validation_data is not None:

    # Reset accumulated values in loss
    # and accuracy objects
    self.loss.new_pass()
    self.accuracy.new_pass()

    # Iterate over steps
    for step in range(validation_steps):

        # If batch size is not set -
        # train using one step and full dataset
        if batch_size is None:
            batch_X = X_val
            batch_y = y_val

        # Otherwise slice a batch
        else:
            batch_X = X_val[
                step*batch_size:(step+1)*batch_size
            ]
            batch_y = y_val[
                step*batch_size:(step+1)*batch_size
            ]
```

```
            # Perform the forward pass
            output = self.forward(batch_X, training=False)

            # Calculate the loss
            self.loss.calculate(output, batch_y)

            # Get predictions and calculate an accuracy
            predictions = self.output_layer_activation.predictions(
                              output)
            self.accuracy.calculate(predictions, batch_y)

        # Get and print validation loss and accuracy
        validation_loss = self.loss.calculate_accumulated()
        validation_accuracy = self.accuracy.calculate_accumulated()

        print(f'validation, ' +
              f'acc: {validation_accuracy:.3f}, ' +
              f'loss: {validation_loss:.3f}')
```

Compared to our current codebase, we've added calls to the `new_pass` method, of both loss and accuracy objects, which reset values accumulated during the training step. Next, we introduced batches (a loop iterating over steps), and removed catching a return from the loss calculation (we don't care about batch loss during validation, just the final, overall loss). The last steps were to add handling for the overall validation loss and replace `X_val` with `batch_X` and `y_val` to `batch_y` to match the changes made to the training code.

This makes our full `train` method for the `Model` class:

```
# Train the model
def train(self, X, y, *, epochs=1, batch_size=None,
          print_every=1, validation_data=None):

    # Initialize accuracy object
    self.accuracy.init(y)

    # Default value if batch size is not being set
    train_steps = 1

    # If there is validation data passed,
    # set default number of steps for validation as well
    if validation_data is not None:
        validation_steps = 1

        # For better readability
        X_val, y_val = validation_data
```

```
        # Calculate number of steps
        if batch_size is not None:
            train_steps = len(X) // batch_size
            # Dividing rounds down. If there are some remaining
            # data, but not a full batch, this won't include it
            # Add `1` to include this not full batch
            if train_steps * batch_size < len(X):
                train_steps += 1

            if validation_data is not None:
                validation_steps = len(X_val) // batch_size
                # Dividing rounds down. If there are some remaining
                # data, but nor full batch, this won't include it
                # Add `1` to include this not full batch
                if validation_steps * batch_size < len(X_val):
                    validation_steps += 1

        # Main training loop
        for epoch in range(1, epochs+1):

            # Print epoch number
            print(f'epoch: {epoch}')

            # Reset accumulated values in loss and accuracy objects
            self.loss.new_pass()
            self.accuracy.new_pass()

            # Iterate over steps
            for step in range(train_steps):

                # If batch size is not set -
                # train using one step and full dataset
                if batch_size is None:
                    batch_X = X
                    batch_y = y

                # Otherwise slice a batch
                else:
                    batch_X = X[step*batch_size:(step+1)*batch_size]
                    batch_y = y[step*batch_size:(step+1)*batch_size]

                # Perform the forward pass
                output = self.forward(batch_X, training=True)

                # Calculate loss
                data_loss, regularization_loss = \
                    self.loss.calculate(output, batch_y,
                                        include_regularization=True)
                loss = data_loss + regularization_loss
```

```
        # Get predictions and calculate an accuracy
        predictions = self.output_layer_activation.predictions(
                          output)
        accuracy = self.accuracy.calculate(predictions,
                                           batch_y)

        # Perform backward pass
        self.backward(output, batch_y)

        # Optimize (update parameters)
        self.optimizer.pre_update_params()
        for layer in self.trainable_layers:
            self.optimizer.update_params(layer)
        self.optimizer.post_update_params()

        # Print a summary
        if not step % print_every or step == train_steps - 1:
            print(f'step: {step}, ' +
                  f'acc: {accuracy:.3f}, ' +
                  f'loss: {loss:.3f} (' +
                  f'data_loss: {data_loss:.3f}, ' +
                  f'reg_loss: {regularization_loss:.3f}), ' +
                  f'lr: {self.optimizer.current_learning_rate}')

    # Get and print epoch loss and accuracy
    epoch_data_loss, epoch_regularization_loss = \
        self.loss.calculate_accumulated(
            include_regularization=True)
    epoch_loss = epoch_data_loss + epoch_regularization_loss
    epoch_accuracy = self.accuracy.calculate_accumulated()

    print(f'training, ' +
          f'acc: {epoch_accuracy:.3f}, ' +
          f'loss: {epoch_loss:.3f} (' +
          f'data_loss: {epoch_data_loss:.3f}, ' +
          f'reg_loss: {epoch_regularization_loss:.3f}), ' +
          f'lr: {self.optimizer.current_learning_rate}')

    # If there is the validation data
    if validation_data is not None:

        # Reset accumulated values in loss
        # and accuracy objects
        self.loss.new_pass()
        self.accuracy.new_pass()
```

```
# Iterate over steps
for step in range(validation_steps):

    # If batch size is not set -
    # train using one step and full dataset
    if batch_size is None:
        batch_X = X_val
        batch_y = y_val

    # Otherwise slice a batch
    else:
        batch_X = X_val[
            step*batch_size:(step+1)*batch_size
        ]
        batch_y = y_val[
            step*batch_size:(step+1)*batch_size
        ]

    # Perform the forward pass
    output = self.forward(batch_X, training=False)

    # Calculate the loss
    self.loss.calculate(output, batch_y)

    # Get predictions and calculate an accuracy
    predictions = self.output_layer_activation.predictions(
                      output)
    self.accuracy.calculate(predictions, batch_y)

# Get and print validation loss and accuracy
validation_loss = self.loss.calculate_accumulated()
validation_accuracy = self.accuracy.calculate_accumulated()

# Print a summary
print(f'validation, ' +
      f'acc: {validation_accuracy:.3f}, ' +
      f'loss: {validation_loss:.3f}')
```

Training

At this point, we're ready to train using batches and our new dataset. As a reminder, we create the data with:

```
# Create dataset
X, y, X_test, y_test = create_data_mnist('fashion_mnist_images')
```

Then shuffle with:

```
# Shuffle the training dataset
keys = np.array(range(X.shape[0]))
np.random.shuffle(keys)
X = X[keys]
y = y[keys]
```

Then flatten sample-wise and scale to the range of -1 to 1:

```
# Scale and reshape samples
X = (X.reshape(X.shape[0], -1).astype(np.float32) - 127.5) / 127.5
X_test = (X_test.reshape(X_test.shape[0], -1).astype(np.float32) -
              127.5) / 127.5
```

Then construct our model consisting of 2 hidden layers using ReLU activation, an output layer with softmax activation since we're building a classification model, cross-entropy loss, Adam optimizer, and categorical accuracy:

```
# Instantiate the model
model = Model()

# Add layers
model.add(Layer_Dense(X.shape[1], 64))
model.add(Activation_ReLU())
model.add(Layer_Dense(64, 64))
model.add(Activation_ReLU())
model.add(Layer_Dense(64, 10))
model.add(Activation_Softmax())
```

Set loss, optimizer and accuracy objects:

```
# Set loss, optimizer and accuracy objects
model.set(
    loss=Loss_CategoricalCrossentropy(),
    optimizer=Optimizer_Adam(decay=5e-5),
    accuracy=Accuracy_Categorical()
)
```

Finally, we finalize and train!

```
# Finalize the model
model.finalize()

# Train the model
model.train(X, y, validation_data=(X_test, y_test),
            epochs=5, batch_size=128, print_every=100)
```

```
>>>
epoch: 1
step: 0, acc: 0.078, loss: 2.303 (data_loss: 2.303, reg_loss: 0.000), lr:
0.001
step: 100, acc: 0.719, loss: 0.660 (data_loss: 0.660, reg_loss: 0.000), lr:
0.0009950248756218907
step: 200, acc: 0.789, loss: 0.560 (data_loss: 0.560, reg_loss: 0.000), lr:
0.0009900990099009901
step: 300, acc: 0.781, loss: 0.612 (data_loss: 0.612, reg_loss: 0.000), lr:
0.0009852216748768474
step: 400, acc: 0.781, loss: 0.518 (data_loss: 0.518, reg_loss: 0.000), lr:
0.000980392156862745
step: 468, acc: 0.833, loss: 0.400 (data_loss: 0.400, reg_loss: 0.000), lr:
0.0009771350400625367
training, acc: 0.720, loss: 0.746 (data_loss: 0.746, reg_loss: 0.000), lr:
0.0009771350400625367
validation, acc: 0.805, loss: 0.537
epoch: 2
step: 0, acc: 0.859, loss: 0.444 (data_loss: 0.444, reg_loss: 0.000), lr:
0.0009770873027505008
step: 100, acc: 0.789, loss: 0.475 (data_loss: 0.475, reg_loss: 0.000), lr:
0.000972337012008362
step: 200, acc: 0.859, loss: 0.357 (data_loss: 0.357, reg_loss: 0.000), lr:
0.0009676326866321544
step: 300, acc: 0.836, loss: 0.461 (data_loss: 0.461, reg_loss: 0.000), lr:
0.0009629736626703259
step: 400, acc: 0.789, loss: 0.437 (data_loss: 0.437, reg_loss: 0.000), lr:
```

```
0.0009583592888974076
step: 468, acc: 0.885, loss: 0.324 (data_loss: 0.324, reg_loss: 0.000), lr:
0.0009552466924583273
training, acc: 0.832, loss: 0.461 (data_loss: 0.461, reg_loss: 0.000), lr:
0.0009552466924583273
validation, acc: 0.836, loss: 0.458
epoch: 3
step: 0, acc: 0.859, loss: 0.387 (data_loss: 0.387, reg_loss: 0.000), lr:
0.0009552010698251983
step: 100, acc: 0.820, loss: 0.433 (data_loss: 0.433, reg_loss: 0.000), lr:
0.0009506607091928891
step: 200, acc: 0.859, loss: 0.320 (data_loss: 0.320, reg_loss: 0.000), lr:
0.0009461633077869241
step: 300, acc: 0.859, loss: 0.424 (data_loss: 0.424, reg_loss: 0.000), lr:
0.0009417082587814295
step: 400, acc: 0.812, loss: 0.394 (data_loss: 0.394, reg_loss: 0.000), lr:
0.0009372949667260287
step: 468, acc: 0.875, loss: 0.286 (data_loss: 0.286, reg_loss: 0.000), lr:
0.000934317481080071
training, acc: 0.851, loss: 0.407 (data_loss: 0.407, reg_loss: 0.000), lr:
0.000934317481080071
validation, acc: 0.847, loss: 0.422
epoch: 4
step: 0, acc: 0.859, loss: 0.350 (data_loss: 0.350, reg_loss: 0.000), lr:
0.0009342738356612324
step: 100, acc: 0.828, loss: 0.398 (data_loss: 0.398, reg_loss: 0.000), lr:
0.0009299297903008323
step: 200, acc: 0.867, loss: 0.310 (data_loss: 0.310, reg_loss: 0.000), lr:
0.0009256259545517657
step: 300, acc: 0.891, loss: 0.393 (data_loss: 0.393, reg_loss: 0.000), lr:
0.0009213617727000506
step: 400, acc: 0.836, loss: 0.363 (data_loss: 0.363, reg_loss: 0.000), lr:
0.0009171366992250195
step: 468, acc: 0.885, loss: 0.264 (data_loss: 0.264, reg_loss: 0.000), lr:
0.0009142857142857143
training, acc: 0.862, loss: 0.378 (data_loss: 0.378, reg_loss: 0.000), lr:
0.0009142857142857143
validation, acc: 0.855, loss: 0.404
epoch: 5
step: 0, acc: 0.836, loss: 0.333 (data_loss: 0.333, reg_loss: 0.000), lr:
0.0009142439202779302
step: 100, acc: 0.828, loss: 0.368 (data_loss: 0.368, reg_loss: 0.000), lr:
0.0009100837277029487
step: 200, acc: 0.867, loss: 0.307 (data_loss: 0.307, reg_loss: 0.000), lr:
0.0009059612248595759
step: 300, acc: 0.891, loss: 0.380 (data_loss: 0.380, reg_loss: 0.000), lr:
0.0009018759018759019
step: 400, acc: 0.859, loss: 0.342 (data_loss: 0.342, reg_loss: 0.000), lr:
0.0008978272580355541
```

```
step: 468, acc: 0.885, loss: 0.241 (data_loss: 0.241, reg_loss: 0.000), lr:
0.0008950948800572861
training, acc: 0.869, loss: 0.357 (data_loss: 0.357, reg_loss: 0.000), lr:
0.0008950948800572861
validation, acc: 0.860, loss: 0.389
```

The model trained successfully and achieved pretty good accuracy. This was done with a new, real, much more challenging dataset and in just 5 epochs instead of 10000. Training also went faster than with our previous attempts at spiral data, where we trained by fitting the whole dataset at once.

So far, we've only mentioned how important it is to shuffle the training data and what might happen if we attempt to train on non-shuffled data. Now would be a good time to exemplify what happens when we don't shuffle it. We can comment out the shuffling code:

```
# Shuffle the training dataset
# keys = np.array(range(X.shape[0]))
# np.random.shuffle(keys)
# X = X[keys]
# y = y[keys]
```

Running again, we can see that we end on:

```
>>>
epoch: 1
step: 0, acc: 0.000, loss: 2.302 (data_loss: 2.302, reg_loss: 0.000), lr:
0.001
step: 100, acc: 0.000, loss: 2.338 (data_loss: 2.338, reg_loss: 0.000), lr:
0.0009950248756218907
step: 200, acc: 0.000, loss: 2.401 (data_loss: 2.401, reg_loss: 0.000), lr:
0.0009900990099009901
step: 300, acc: 0.000, loss: 2.214 (data_loss: 2.214, reg_loss: 0.000), lr:
0.0009852216748768474
step: 400, acc: 0.000, loss: 2.278 (data_loss: 2.278, reg_loss: 0.000), lr:
0.000980392156862745
step: 468, acc: 1.000, loss: 0.018 (data_loss: 0.018, reg_loss: 0.000), lr:
0.0009771350400625367
training, acc: 0.381, loss: 2.246 (data_loss: 2.246, reg_loss: 0.000), lr:
0.0009771350400625367
validation, acc: 0.100, loss: 6.982
epoch: 2
step: 0, acc: 0.000, loss: 8.201 (data_loss: 8.201, reg_loss: 0.000), lr:
```

```
0.0009770873027505008
step: 100, acc: 0.000, loss: 4.577 (data_loss: 4.577, reg_loss: 0.000), lr:
0.000972337012008362
step: 200, acc: 0.383, loss: 1.821 (data_loss: 1.821, reg_loss: 0.000), lr:
0.0009676326866321544
step: 300, acc: 0.000, loss: 0.964 (data_loss: 0.964, reg_loss: 0.000), lr:
0.0009629736626703259
step: 400, acc: 0.000, loss: 1.545 (data_loss: 1.545, reg_loss: 0.000), lr:
0.0009583592888974076
step: 468, acc: 1.000, loss: 0.013 (data_loss: 0.013, reg_loss: 0.000), lr:
0.0009552466924583273
training, acc: 0.597, loss: 1.573 (data_loss: 1.573, reg_loss: 0.000), lr:
0.0009552466924583273
validation, acc: 0.109, loss: 3.917
epoch: 3
step: 0, acc: 0.000, loss: 3.431 (data_loss: 3.431, reg_loss: 0.000), lr:
0.0009552010698251983
step: 100, acc: 0.000, loss: 3.519 (data_loss: 3.519, reg_loss: 0.000), lr:
0.0009506607091928891
step: 200, acc: 0.859, loss: 0.559 (data_loss: 0.559, reg_loss: 0.000), lr:
0.0009461633077869241
step: 300, acc: 1.000, loss: 0.225 (data_loss: 0.225, reg_loss: 0.000), lr:
0.0009417082587814295
step: 400, acc: 1.000, loss: 0.151 (data_loss: 0.151, reg_loss: 0.000), lr:
0.0009372949667260287
step: 468, acc: 1.000, loss: 0.012 (data_loss: 0.012, reg_loss: 0.000), lr:
0.000934317481080071
training, acc: 0.638, loss: 1.478 (data_loss: 1.478, reg_loss: 0.000), lr:
0.000934317481080071
validation, acc: 0.134, loss: 3.108
epoch: 4
step: 0, acc: 0.031, loss: 2.620 (data_loss: 2.620, reg_loss: 0.000), lr:
0.0009342738356612324
step: 100, acc: 0.000, loss: 4.128 (data_loss: 4.128, reg_loss: 0.000), lr:
0.0009299297903008323
step: 200, acc: 0.000, loss: 1.891 (data_loss: 1.891, reg_loss: 0.000), lr:
0.0009256259545517657
step: 300, acc: 1.000, loss: 0.118 (data_loss: 0.118, reg_loss: 0.000), lr:
0.0009213617727000506
step: 400, acc: 1.000, loss: 0.065 (data_loss: 0.065, reg_loss: 0.000), lr:
0.0009171366992250195
step: 468, acc: 1.000, loss: 0.011 (data_loss: 0.011, reg_loss: 0.000), lr:
0.0009142857142857143
training, acc: 0.644, loss: 1.335 (data_loss: 1.335, reg_loss: 0.000), lr:
0.0009142857142857143
validation, acc: 0.189, loss: 3.050
epoch: 5
step: 0, acc: 0.016, loss: 2.734 (data_loss: 2.734, reg_loss: 0.000), lr:
0.0009142439202779302
```

```
step: 100, acc: 0.000, loss: 2.848 (data_loss: 2.848, reg_loss: 0.000), lr:
0.0009100837277029487
step: 200, acc: 0.547, loss: 1.108 (data_loss: 1.108, reg_loss: 0.000), lr:
0.0009059612248595759
step: 300, acc: 0.992, loss: 0.018 (data_loss: 0.018, reg_loss: 0.000), lr:
0.0009018759018759019
step: 400, acc: 1.000, loss: 0.065 (data_loss: 0.065, reg_loss: 0.000), lr:
0.0008978272580355541
step: 468, acc: 1.000, loss: 0.010 (data_loss: 0.010, reg_loss: 0.000), lr:
0.0008950948800572861
training, acc: 0.744, loss: 0.961 (data_loss: 0.961, reg_loss: 0.000), lr:
0.0008950948800572861
validation, acc: 0.200, loss: 3.311
```

As we can see, this doesn't work well at all. We can observe how the model approached a perfect accuracy of 1 during training, but epoch accuracy remained poor, and the validation accuracy proved that the model did not learn. Training accuracy quickly became high, since the model learned to predict just one label (as it repeatedly saw only that label). Once the label changed in the training data, the model quickly learned to predict only that new label, as that's all it saw in every batch. This process repeated to the end of an epoch and then over all epochs. Epoch accuracy is lower because it took a while for the model to learn the new label after a switch, and it showed a low accuracy during this period. Validation accuracy was calculated after training for a given epoch ended, and as we remember, the model learned to predict just one label. In the case of validation, the label that the model predicted was the last label it had seen — its accuracy was close to 1/10 as our training dataset consists of 10 classes.

Re-enable shuffling, and then you can tinker around with your model to see if you can further improve results. Here is an example with a larger model, a higher learning rate decay, and twice as many epochs:

```
# Add layers
model.add(Layer_Dense(X.shape[1], 128))
model.add(Activation_ReLU())
model.add(Layer_Dense(128, 128))
model.add(Activation_ReLU())
model.add(Layer_Dense(128, 10))
model.add(Activation_Softmax())

# Set loss, optimizer and accuracy objects
model.set(
    loss=Loss_CategoricalCrossentropy(),
    optimizer=Optimizer_Adam(decay=1e-3),
    accuracy=Accuracy_Categorical()
)
```

```
# Finalize the model
model.finalize()

# Train the model
model.train(X, y, validation_data=(X_test, y_test),
            epochs=10, batch_size=128, print_every=100)
```

```
>>>
...
epoch: 10
step: 0, acc: 0.891, loss: 0.263 (data_loss: 0.263, reg_loss: 0.000), lr:
0.0001915341888527102
step: 100, acc: 0.883, loss: 0.257 (data_loss: 0.257, reg_loss: 0.000), lr:
0.00018793459875963167
step: 200, acc: 0.922, loss: 0.227 (data_loss: 0.227, reg_loss: 0.000), lr:
0.00018446781036709093
step: 300, acc: 0.898, loss: 0.282 (data_loss: 0.282, reg_loss: 0.000), lr:
0.00018112660749864155
step: 400, acc: 0.914, loss: 0.299 (data_loss: 0.299, reg_loss: 0.000), lr:
0.00017790428749332856
step: 468, acc: 0.917, loss: 0.192 (data_loss: 0.192, reg_loss: 0.000), lr:
0.00017577781683951485
training, acc: 0.894, loss: 0.291 (data_loss: 0.291, reg_loss: 0.000), lr:
0.00017577781683951485
validation, acc: 0.874, loss: 0.354
```

We improved accuracy and decreased loss a bit by simply increasing the model size, decay, and number of epochs.

Full code up to now:

```
import numpy as np
import nnfs
import os
import cv2

nnfs.init()


# Dense layer
class Layer_Dense:

    # Layer initialization
    def __init__(self, n_inputs, n_neurons,
                 weight_regularizer_l1=0, weight_regularizer_l2=0,
                 bias_regularizer_l1=0, bias_regularizer_l2=0):
        # Initialize weights and biases
        self.weights = 0.01 * np.random.randn(n_inputs, n_neurons)
        self.biases = np.zeros((1, n_neurons))
        # Set regularization strength
        self.weight_regularizer_l1 = weight_regularizer_l1
        self.weight_regularizer_l2 = weight_regularizer_l2
        self.bias_regularizer_l1 = bias_regularizer_l1
        self.bias_regularizer_l2 = bias_regularizer_l2

    # Forward pass
    def forward(self, inputs, training):
        # Remember input values
        self.inputs = inputs
        # Calculate output values from inputs, weights and biases
        self.output = np.dot(inputs, self.weights) + self.biases

    # Backward pass
    def backward(self, dvalues):
        # Gradients on parameters
        self.dweights = np.dot(self.inputs.T, dvalues)
        self.dbiases = np.sum(dvalues, axis=0, keepdims=True)
```

```
        # Gradients on regularization
        # L1 on weights
        if self.weight_regularizer_l1 > 0:
            dL1 = np.ones_like(self.weights)
            dL1[self.weights < 0] = -1
            self.dweights += self.weight_regularizer_l1 * dL1
        # L2 on weights
        if self.weight_regularizer_l2 > 0:
            self.dweights += 2 * self.weight_regularizer_l2 * \
                             self.weights
        # L1 on biases
        if self.bias_regularizer_l1 > 0:
            dL1 = np.ones_like(self.biases)
            dL1[self.biases < 0] = -1
            self.dbiases += self.bias_regularizer_l1 * dL1
        # L2 on biases
        if self.bias_regularizer_l2 > 0:
            self.dbiases += 2 * self.bias_regularizer_l2 * \
                            self.biases

        # Gradient on values
        self.dinputs = np.dot(dvalues, self.weights.T)


# Dropout
class Layer_Dropout:

    # Init
    def __init__(self, rate):
        # Store rate, we invert it as for example for dropout
        # of 0.1 we need success rate of 0.9
        self.rate = 1 - rate

    # Forward pass
    def forward(self, inputs, training):
        # Save input values
        self.inputs = inputs

        # If not in the training mode - return values
        if not training:
            self.output = inputs.copy()
            return

        # Generate and save scaled mask
        self.binary_mask = np.random.binomial(1, self.rate,
                           size=inputs.shape) / self.rate
        # Apply mask to output values
        self.output = inputs * self.binary_mask
```

```
    # Backward pass
    def backward(self, dvalues):
        # Gradient on values
        self.dinputs = dvalues * self.binary_mask


# Input "layer"
class Layer_Input:

    # Forward pass
    def forward(self, inputs, training):
        self.output = inputs


# ReLU activation
class Activation_ReLU:

    # Forward pass
    def forward(self, inputs, training):
        # Remember input values
        self.inputs = inputs
        # Calculate output values from inputs
        self.output = np.maximum(0, inputs)

    # Backward pass
    def backward(self, dvalues):
        # Since we need to modify original variable,
        # let's make a copy of values first
        self.dinputs = dvalues.copy()

        # Zero gradient where input values were negative
        self.dinputs[self.inputs <= 0] = 0

    # Calculate predictions for outputs
    def predictions(self, outputs):
        return outputs


# Softmax activation
class Activation_Softmax:

    # Forward pass
    def forward(self, inputs, training):
        # Remember input values
        self.inputs = inputs

        # Get unnormalized probabilities
        exp_values = np.exp(inputs - np.max(inputs, axis=1,
                                            keepdims=True))
```

```
        # Normalize them for each sample
        probabilities = exp_values / np.sum(exp_values, axis=1,
                                            keepdims=True)

        self.output = probabilities

    # Backward pass
    def backward(self, dvalues):

        # Create uninitialized array
        self.dinputs = np.empty_like(dvalues)

        # Enumerate outputs and gradients
        for index, (single_output, single_dvalues) in \
                enumerate(zip(self.output, dvalues)):
            # Flatten output array
            single_output = single_output.reshape(-1, 1)
            # Calculate Jacobian matrix of the output
            jacobian_matrix = np.diagflat(single_output) - \
                              np.dot(single_output, single_output.T)
            # Calculate sample-wise gradient
            # and add it to the array of sample gradients
            self.dinputs[index] = np.dot(jacobian_matrix,
                                         single_dvalues)

    # Calculate predictions for outputs
    def predictions(self, outputs):
        return np.argmax(outputs, axis=1)


# Sigmoid activation
class Activation_Sigmoid:

    # Forward pass
    def forward(self, inputs, training):
        # Save input and calculate/save output
        # of the sigmoid function
        self.inputs = inputs
        self.output = 1 / (1 + np.exp(-inputs))

    # Backward pass
    def backward(self, dvalues):
        # Derivative - calculates from output of the sigmoid function
        self.dinputs = dvalues * (1 - self.output) * self.output

    # Calculate predictions for outputs
    def predictions(self, outputs):
        return (outputs > 0.5) * 1
```

```
# Linear activation
class Activation_Linear:

    # Forward pass
    def forward(self, inputs, training):
        # Just remember values
        self.inputs = inputs
        self.output = inputs

    # Backward pass
    def backward(self, dvalues):
        # derivative is 1, 1 * dvalues = dvalues - the chain rule
        self.dinputs = dvalues.copy()

    # Calculate predictions for outputs
    def predictions(self, outputs):
        return outputs


# SGD optimizer
class Optimizer_SGD:

    # Initialize optimizer - set settings,
    # learning rate of 1. is default for this optimizer
    def __init__(self, learning_rate=1., decay=0., momentum=0.):
        self.learning_rate = learning_rate
        self.current_learning_rate = learning_rate
        self.decay = decay
        self.iterations = 0
        self.momentum = momentum

    # Call once before any parameter updates
    def pre_update_params(self):
        if self.decay:
            self.current_learning_rate = self.learning_rate * \
                (1. / (1. + self.decay * self.iterations))

    # Update parameters
    def update_params(self, layer):

        # If we use momentum
        if self.momentum:

            # If layer does not contain momentum arrays, create them
            # filled with zeros
            if not hasattr(layer, 'weight_momentums'):
                layer.weight_momentums = np.zeros_like(layer.weights)
```

```
            # If there is no momentum array for weights
            # The array doesn't exist for biases yet either.
            layer.bias_momentums = np.zeros_like(layer.biases)

        # Build weight updates with momentum - take previous
        # updates multiplied by retain factor and update with
        # current gradients
        weight_updates = \
            self.momentum * layer.weight_momentums - \
            self.current_learning_rate * layer.dweights
        layer.weight_momentums = weight_updates

        # Build bias updates
        bias_updates = \
            self.momentum * layer.bias_momentums - \
            self.current_learning_rate * layer.dbiases
        layer.bias_momentums = bias_updates

    # Vanilla SGD updates (as before momentum update)
    else:
        weight_updates = -self.current_learning_rate * \
                         layer.dweights
        bias_updates = -self.current_learning_rate * \
                       layer.dbiases

    # Update weights and biases using either
    # vanilla or momentum updates
    layer.weights += weight_updates
    layer.biases += bias_updates

# Call once after any parameter updates
def post_update_params(self):
    self.iterations += 1


# Adagrad optimizer
class Optimizer_Adagrad:

    # Initialize optimizer - set settings
    def __init__(self, learning_rate=1., decay=0., epsilon=1e-7):
        self.learning_rate = learning_rate
        self.current_learning_rate = learning_rate
        self.decay = decay
        self.iterations = 0
        self.epsilon = epsilon
```

```
    # Call once before any parameter updates
    def pre_update_params(self):
        if self.decay:
            self.current_learning_rate = self.learning_rate * \
                (1. / (1. + self.decay * self.iterations))

    # Update parameters
    def update_params(self, layer):

        # If layer does not contain cache arrays,
        # create them filled with zeros
        if not hasattr(layer, 'weight_cache'):
            layer.weight_cache = np.zeros_like(layer.weights)
            layer.bias_cache = np.zeros_like(layer.biases)

        # Update cache with squared current gradients
        layer.weight_cache += layer.dweights**2
        layer.bias_cache += layer.dbiases**2

        # Vanilla SGD parameter update + normalization
        # with square rooted cache
        layer.weights += -self.current_learning_rate * \
                         layer.dweights / \
                         (np.sqrt(layer.weight_cache) + self.epsilon)
        layer.biases += -self.current_learning_rate * \
                        layer.dbiases / \
                        (np.sqrt(layer.bias_cache) + self.epsilon)

    # Call once after any parameter updates
    def post_update_params(self):
        self.iterations += 1


# RMSprop optimizer
class Optimizer_RMSprop:

    # Initialize optimizer - set settings
    def __init__(self, learning_rate=0.001, decay=0., epsilon=1e-7,
                 rho=0.9):
        self.learning_rate = learning_rate
        self.current_learning_rate = learning_rate
        self.decay = decay
        self.iterations = 0
        self.epsilon = epsilon
        self.rho = rho
```

```
    # Call once before any parameter updates
    def pre_update_params(self):
        if self.decay:
            self.current_learning_rate = self.learning_rate * \
                (1. / (1. + self.decay * self.iterations))

    # Update parameters
    def update_params(self, layer):

        # If layer does not contain cache arrays,
        # create them filled with zeros
        if not hasattr(layer, 'weight_cache'):
            layer.weight_cache = np.zeros_like(layer.weights)
            layer.bias_cache = np.zeros_like(layer.biases)

        # Update cache with squared current gradients
        layer.weight_cache = self.rho * layer.weight_cache + \
            (1 - self.rho) * layer.dweights**2
        layer.bias_cache = self.rho * layer.bias_cache + \
            (1 - self.rho) * layer.dbiases**2

        # Vanilla SGD parameter update + normalization
        # with square rooted cache
        layer.weights += -self.current_learning_rate * \
                         layer.dweights / \
                         (np.sqrt(layer.weight_cache) + self.epsilon)
        layer.biases += -self.current_learning_rate * \
                        layer.dbiases / \
                        (np.sqrt(layer.bias_cache) + self.epsilon)

    # Call once after any parameter updates
    def post_update_params(self):
        self.iterations += 1


# Adam optimizer
class Optimizer_Adam:

    # Initialize optimizer - set settings
    def __init__(self, learning_rate=0.001, decay=0., epsilon=1e-7,
                 beta_1=0.9, beta_2=0.999):
        self.learning_rate = learning_rate
        self.current_learning_rate = learning_rate
        self.decay = decay
        self.iterations = 0
        self.epsilon = epsilon
        self.beta_1 = beta_1
        self.beta_2 = beta_2
```

```
    # Call once before any parameter updates
    def pre_update_params(self):
        if self.decay:
            self.current_learning_rate = self.learning_rate * \
                (1. / (1. + self.decay * self.iterations))

    # Update parameters
    def update_params(self, layer):

        # If layer does not contain cache arrays,
        # create them filled with zeros
        if not hasattr(layer, 'weight_cache'):
            layer.weight_momentums = np.zeros_like(layer.weights)
            layer.weight_cache = np.zeros_like(layer.weights)
            layer.bias_momentums = np.zeros_like(layer.biases)
            layer.bias_cache = np.zeros_like(layer.biases)

        # Update momentum  with current gradients
        layer.weight_momentums = self.beta_1 * \
                                 layer.weight_momentums + \
                                 (1 - self.beta_1) * layer.dweights
        layer.bias_momentums = self.beta_1 * \
                               layer.bias_momentums + \
                               (1 - self.beta_1) * layer.dbiases
        # Get corrected momentum
        # self.iteration is 0 at first pass
        # and we need to start with 1 here
        weight_momentums_corrected = layer.weight_momentums / \
            (1 - self.beta_1 ** (self.iterations + 1))
        bias_momentums_corrected = layer.bias_momentums / \
            (1 - self.beta_1 ** (self.iterations + 1))
        # Update cache with squared current gradients
        layer.weight_cache = self.beta_2 * layer.weight_cache + \
            (1 - self.beta_2) * layer.dweights**2
        layer.bias_cache = self.beta_2 * layer.bias_cache + \
            (1 - self.beta_2) * layer.dbiases**2
        # Get corrected cache
        weight_cache_corrected = layer.weight_cache / \
            (1 - self.beta_2 ** (self.iterations + 1))
        bias_cache_corrected = layer.bias_cache / \
            (1 - self.beta_2 ** (self.iterations + 1))

        # Vanilla SGD parameter update + normalization
        # with square rooted cache
        layer.weights += -self.current_learning_rate * \
                         weight_momentums_corrected / \
                         (np.sqrt(weight_cache_corrected) +
                             self.epsilon)
```

```
        layer.biases += -self.current_learning_rate * \
                        bias_momentums_corrected / \
                        (np.sqrt(bias_cache_corrected) +
                            self.epsilon)

    # Call once after any parameter updates
    def post_update_params(self):
        self.iterations += 1


# Common loss class
class Loss:

    # Regularization loss calculation
    def regularization_loss(self):

        # 0 by default
        regularization_loss = 0

        # Calculate regularization loss
        # iterate all trainable layers
        for layer in self.trainable_layers:

            # L1 regularization - weights
            # calculate only when factor greater than 0
            if layer.weight_regularizer_l1 > 0:
                regularization_loss += layer.weight_regularizer_l1 * \
                                       np.sum(np.abs(layer.weights))

            # L2 regularization - weights
            if layer.weight_regularizer_l2 > 0:
                regularization_loss += layer.weight_regularizer_l2 * \
                                       np.sum(layer.weights * \
                                              layer.weights)

            # L1 regularization - biases
            # calculate only when factor greater than 0
            if layer.bias_regularizer_l1 > 0:
                regularization_loss += layer.bias_regularizer_l1 * \
                                       np.sum(np.abs(layer.biases))

            # L2 regularization - biases
            if layer.bias_regularizer_l2 > 0:
                regularization_loss += layer.bias_regularizer_l2 * \
                                       np.sum(layer.biases * \
                                              layer.biases)

        return regularization_loss
```

```
    # Set/remember trainable layers
    def remember_trainable_layers(self, trainable_layers):
        self.trainable_layers = trainable_layers

    # Calculates the data and regularization losses
    # given model output and ground truth values
    def calculate(self, output, y, *, include_regularization=False):

        # Calculate sample losses
        sample_losses = self.forward(output, y)

        # Calculate mean loss
        data_loss = np.mean(sample_losses)

        # Add accumulated sum of losses and sample count
        self.accumulated_sum += np.sum(sample_losses)
        self.accumulated_count += len(sample_losses)

        # If just data loss - return it
        if not include_regularization:
            return data_loss

        # Return the data and regularization losses
        return data_loss, self.regularization_loss()

    # Calculates accumulated loss
    def calculate_accumulated(self, *, include_regularization=False):

        # Calculate mean loss
        data_loss = self.accumulated_sum / self.accumulated_count

        # If just data loss - return it
        if not include_regularization:
            return data_loss

        # Return the data and regularization losses
        return data_loss, self.regularization_loss()

    # Reset variables for accumulated loss
    def new_pass(self):
        self.accumulated_sum = 0
        self.accumulated_count = 0
```

```python
# Cross-entropy loss
class Loss_CategoricalCrossentropy(Loss):

    # Forward pass
    def forward(self, y_pred, y_true):

        # Number of samples in a batch
        samples = len(y_pred)

        # Clip data to prevent division by 0
        # Clip both sides to not drag mean towards any value
        y_pred_clipped = np.clip(y_pred, 1e-7, 1 - 1e-7)

        # Probabilities for target values -
        # only if categorical labels
        if len(y_true.shape) == 1:
            correct_confidences = y_pred_clipped[
                range(samples),
                y_true
            ]

        # Mask values - only for one-hot encoded labels
        elif len(y_true.shape) == 2:
            correct_confidences = np.sum(
                y_pred_clipped * y_true,
                axis=1
            )

        # Losses
        negative_log_likelihoods = -np.log(correct_confidences)
        return negative_log_likelihoods

    # Backward pass
    def backward(self, dvalues, y_true):

        # Number of samples
        samples = len(dvalues)
        # Number of labels in every sample
        # We'll use the first sample to count them
        labels = len(dvalues[0])

        # If labels are sparse, turn them into one-hot vector
        if len(y_true.shape) == 1:
            y_true = np.eye(labels)[y_true]

        # Calculate gradient
        self.dinputs = -y_true / dvalues
        # Normalize gradient
        self.dinputs = self.dinputs / samples
```

```
# Softmax classifier - combined Softmax activation
# and cross-entropy loss for faster backward step
class Activation_Softmax_Loss_CategoricalCrossentropy():

    # Backward pass
    def backward(self, dvalues, y_true):

        # Number of samples
        samples = len(dvalues)

        # If labels are one-hot encoded,
        # turn them into discrete values
        if len(y_true.shape) == 2:
            y_true = np.argmax(y_true, axis=1)

        # Copy so we can safely modify
        self.dinputs = dvalues.copy()
        # Calculate gradient
        self.dinputs[range(samples), y_true] -= 1
        # Normalize gradient
        self.dinputs = self.dinputs / samples


# Binary cross-entropy loss
class Loss_BinaryCrossentropy(Loss):

    # Forward pass
    def forward(self, y_pred, y_true):

        # Clip data to prevent division by 0
        # Clip both sides to not drag mean towards any value
        y_pred_clipped = np.clip(y_pred, 1e-7, 1 - 1e-7)

        # Calculate sample-wise loss
        sample_losses = -(y_true * np.log(y_pred_clipped) +
                          (1 - y_true) * np.log(1 - y_pred_clipped))
        sample_losses = np.mean(sample_losses, axis=-1)

        # Return losses
        return sample_losses

    # Backward pass
    def backward(self, dvalues, y_true):

        # Number of samples
        samples = len(dvalues)
        # Number of outputs in every sample
        # We'll use the first sample to count them
        outputs = len(dvalues[0])
```

```
        # Clip data to prevent division by 0
        # Clip both sides to not drag mean towards any value
        clipped_dvalues = np.clip(dvalues, 1e-7, 1 - 1e-7)

        # Calculate gradient
        self.dinputs = -(y_true / clipped_dvalues -
                         (1 - y_true) / (1 - clipped_dvalues)) / outputs
        # Normalize gradient
        self.dinputs = self.dinputs / samples


# Mean Squared Error loss
class Loss_MeanSquaredError(Loss):  # L2 loss

    # Forward pass
    def forward(self, y_pred, y_true):

        # Calculate loss
        sample_losses = np.mean((y_true - y_pred)**2, axis=-1)

        # Return losses
        return sample_losses

    # Backward pass
    def backward(self, dvalues, y_true):

        # Number of samples
        samples = len(dvalues)
        # Number of outputs in every sample
        # We'll use the first sample to count them
        outputs = len(dvalues[0])

        # Gradient on values
        self.dinputs = -2 * (y_true - dvalues) / outputs
        # Normalize gradient
        self.dinputs = self.dinputs / samples


# Mean Absolute Error loss
class Loss_MeanAbsoluteError(Loss):  # L1 loss

    def forward(self, y_pred, y_true):

        # Calculate loss
        sample_losses = np.mean(np.abs(y_true - y_pred), axis=-1)

        # Return losses
        return sample_losses
```

```
    # Backward pass
    def backward(self, dvalues, y_true):

        # Number of samples
        samples = len(dvalues)
        # Number of outputs in every sample
        # We'll use the first sample to count them
        outputs = len(dvalues[0])

        # Calculate gradient
        self.dinputs = np.sign(y_true - dvalues) / outputs
        # Normalize gradient
        self.dinputs = self.dinputs / samples


# Common accuracy class
class Accuracy:

    # Calculates an accuracy
    # given predictions and ground truth values
    def calculate(self, predictions, y):

        # Get comparison results
        comparisons = self.compare(predictions, y)

        # Calculate an accuracy
        accuracy = np.mean(comparisons)

        # Add accumulated sum of matching values and sample count
        self.accumulated_sum += np.sum(comparisons)
        self.accumulated_count += len(comparisons)

        # Return accuracy
        return accuracy

    # Calculates accumulated accuracy
    def calculate_accumulated(self):

        # Calculate an accuracy
        accuracy = self.accumulated_sum / self.accumulated_count

        # Return the data and regularization losses
        return accuracy

    # Reset variables for accumulated accuracy
    def new_pass(self):
        self.accumulated_sum = 0
        self.accumulated_count = 0
```

```
# Accuracy calculation for classification model
class Accuracy_Categorical(Accuracy):

    def __init__(self, *, binary=False):
        # Binary mode?
        self.binary = binary

    # No initialization is needed
    def init(self, y):
        pass

    # Compares predictions to the ground truth values
    def compare(self, predictions, y):
        if not self.binary and len(y.shape) == 2:
            y = np.argmax(y, axis=1)
        return predictions == y


# Accuracy calculation for regression model
class Accuracy_Regression(Accuracy):

    def __init__(self):
        # Create precision property
        self.precision = None

    # Calculates precision value
    # based on passed-in ground truth values
    def init(self, y, reinit=False):
        if self.precision is None or reinit:
            self.precision = np.std(y) / 250

    # Compares predictions to the ground truth values
    def compare(self, predictions, y):
        return np.absolute(predictions - y) < self.precision


# Model class
class Model:

    def __init__(self):
        # Create a list of network objects
        self.layers = []
        # Softmax classifier's output object
        self.softmax_classifier_output = None

    # Add objects to the model
    def add(self, layer):
        self.layers.append(layer)
```

```
    # Set loss, optimizer and accuracy
    def set(self, *, loss, optimizer, accuracy):
        self.loss = loss
        self.optimizer = optimizer
        self.accuracy = accuracy

    # Finalize the model
    def finalize(self):

        # Create and set the input layer
        self.input_layer = Layer_Input()

        # Count all the objects
        layer_count = len(self.layers)

        # Initialize a list containing trainable layers:
        self.trainable_layers = []

        # Iterate the objects
        for i in range(layer_count):

            # If it's the first layer,
            # the previous layer object is the input layer
            if i == 0:
                self.layers[i].prev = self.input_layer
                self.layers[i].next = self.layers[i+1]

            # All layers except for the first and the last
            elif i < layer_count - 1:
                self.layers[i].prev = self.layers[i-1]
                self.layers[i].next = self.layers[i+1]

            # The last layer - the next object is the loss
            # Also let's save aside the reference to the last object
            # whose output is the model's output
            else:
                self.layers[i].prev = self.layers[i-1]
                self.layers[i].next = self.loss
                self.output_layer_activation = self.layers[i]

            # If layer contains an attribute called "weights",
            # it's a trainable layer -
            # add it to the list of trainable layers
            # We don't need to check for biases -
            # checking for weights is enough
            if hasattr(self.layers[i], 'weights'):
                self.trainable_layers.append(self.layers[i])
```

```
        # Update loss object with trainable layers
        self.loss.remember_trainable_layers(
            self.trainable_layers
        )

        # If output activation is Softmax and
        # loss function is Categorical Cross-Entropy
        # create an object of combined activation
        # and loss function containing
        # faster gradient calculation
        if isinstance(self.layers[-1], Activation_Softmax) and \
           isinstance(self.loss, Loss_CategoricalCrossentropy):
            # Create an object of combined activation
            # and loss functions
            self.softmax_classifier_output = \
                Activation_Softmax_Loss_CategoricalCrossentropy()

    # Train the model
    def train(self, X, y, *, epochs=1, batch_size=None,
              print_every=1, validation_data=None):

        # Initialize accuracy object
        self.accuracy.init(y)

        # Default value if batch size is not being set
        train_steps = 1

        # If there is validation data passed,
        # set default number of steps for validation as well
        if validation_data is not None:
            validation_steps = 1

            # For better readability
            X_val, y_val = validation_data

        # Calculate number of steps
        if batch_size is not None:
            train_steps = len(X) // batch_size
            # Dividing rounds down. If there are some remaining
            # data but not a full batch, this won't include it
            # Add `1` to include this not full batch
            if train_steps * batch_size < len(X):
                train_steps += 1

            if validation_data is not None:
                validation_steps = len(X_val) // batch_size
```

```
        # Dividing rounds down. If there are some remaining
        # data but nor full batch, this won't include it
        # Add `1` to include this not full batch
        if validation_steps * batch_size < len(X_val):
            validation_steps += 1

    # Main training loop
    for epoch in range(1, epochs+1):

        # Print epoch number
        print(f'epoch: {epoch}')

        # Reset accumulated values in loss and accuracy objects
        self.loss.new_pass()
        self.accuracy.new_pass()

        # Iterate over steps
        for step in range(train_steps):

            # If batch size is not set -
            # train using one step and full dataset
            if batch_size is None:
                batch_X = X
                batch_y = y

            # Otherwise slice a batch
            else:
                batch_X = X[step*batch_size:(step+1)*batch_size]
                batch_y = y[step*batch_size:(step+1)*batch_size]

            # Perform the forward pass
            output = self.forward(batch_X, training=True)

            # Calculate loss
            data_loss, regularization_loss = \
                self.loss.calculate(output, batch_y,
                                    include_regularization=True)
            loss = data_loss + regularization_loss

            # Get predictions and calculate an accuracy
            predictions = self.output_layer_activation.predictions(
                              output)
            accuracy = self.accuracy.calculate(predictions,
                                               batch_y)

            # Perform backward pass
            self.backward(output, batch_y)
```

```
        # Optimize (update parameters)
        self.optimizer.pre_update_params()
        for layer in self.trainable_layers:
            self.optimizer.update_params(layer)
        self.optimizer.post_update_params()

        # Print a summary
        if not step % print_every or step == train_steps - 1:
            print(f'step: {step}, ' +
                  f'acc: {accuracy:.3f}, ' +
                  f'loss: {loss:.3f} (' +
                  f'data_loss: {data_loss:.3f}, ' +
                  f'reg_loss: {regularization_loss:.3f}), ' +
                  f'lr: {self.optimizer.current_learning_rate}')

    # Get and print epoch loss and accuracy
    epoch_data_loss, epoch_regularization_loss = \
        self.loss.calculate_accumulated(
            include_regularization=True)
    epoch_loss = epoch_data_loss + epoch_regularization_loss
    epoch_accuracy = self.accuracy.calculate_accumulated()

    print(f'training, ' +
          f'acc: {epoch_accuracy:.3f}, ' +
          f'loss: {epoch_loss:.3f} (' +
          f'data_loss: {epoch_data_loss:.3f}, ' +
          f'reg_loss: {epoch_regularization_loss:.3f}), ' +
          f'lr: {self.optimizer.current_learning_rate}')

    # If there is the validation data
    if validation_data is not None:

        # Reset accumulated values in loss
        # and accuracy objects
        self.loss.new_pass()
        self.accuracy.new_pass()

        # Iterate over steps
        for step in range(validation_steps):

            # If batch size is not set -
            # train using one step and full dataset
            if batch_size is None:
                batch_X = X_val
                batch_y = y_val
```

```
                # Otherwise slice a batch
                else:
                    batch_X = X_val[
                        step*batch_size:(step+1)*batch_size
                    ]
                    batch_y = y_val[
                        step*batch_size:(step+1)*batch_size
                    ]

                # Perform the forward pass
                output = self.forward(batch_X, training=False)

                # Calculate the loss
                self.loss.calculate(output, batch_y)

                # Get predictions and calculate an accuracy
                predictions = self.output_layer_activation.predictions(
                                  output)
                self.accuracy.calculate(predictions, batch_y)

            # Get and print validation loss and accuracy
            validation_loss = self.loss.calculate_accumulated()
            validation_accuracy = self.accuracy.calculate_accumulated()

            # Print a summary
            print(f'validation, ' +
                  f'acc: {validation_accuracy:.3f}, ' +
                  f'loss: {validation_loss:.3f}')

    # Performs forward pass
    def forward(self, X, training):

        # Call forward method on the input layer
        # this will set the output property that
        # the first layer in "prev" object is expecting
        self.input_layer.forward(X, training)

        # Call forward method of every object in a chain
        # Pass output of the previous object as a parameter
        for layer in self.layers:
            layer.forward(layer.prev.output, training)

        # "layer" is now the last object from the list,
        # return its output
        return layer.output
```

```
    # Performs backward pass
    def backward(self, output, y):

        # If softmax classifier
        if self.softmax_classifier_output is not None:
            # First call backward method
            # on the combined activation/loss
            # this will set dinputs property
            self.softmax_classifier_output.backward(output, y)

            # Since we'll not call backward method of the last layer
            # which is Softmax activation
            # as we used combined activation/loss
            # object, let's set dinputs in this object
            self.layers[-1].dinputs = \
                self.softmax_classifier_output.dinputs

            # Call backward method going through
            # all the objects but last
            # in reversed order passing dinputs as a parameter
            for layer in reversed(self.layers[:-1]):
                layer.backward(layer.next.dinputs)

            return

        # First call backward method on the loss
        # this will set dinputs property that the last
        # layer will try to access shortly
        self.loss.backward(output, y)

        # Call backward method going through all the objects
        # in reversed order passing dinputs as a parameter
        for layer in reversed(self.layers):
            layer.backward(layer.next.dinputs)


# Loads a MNIST dataset
def load_mnist_dataset(dataset, path):

    # Scan all the directories and create a list of labels
    labels = os.listdir(os.path.join(path, dataset))

    # Create lists for samples and labels
    X = []
    y = []
```

```
    # For each label folder
    for label in labels:
        # And for each image in given folder
        for file in os.listdir(os.path.join(path, dataset, label)):
            # Read the image
            image = cv2.imread(
                        os.path.join(path, dataset, label, file),
                        cv2.IMREAD_UNCHANGED)

            # And append it and a label to the lists
            X.append(image)
            y.append(label)

    # Convert the data to proper numpy arrays and return
    return np.array(X), np.array(y).astype('uint8')


# MNIST dataset (train + test)
def create_data_mnist(path):

    # Load both sets separately
    X, y = load_mnist_dataset('train', path)
    X_test, y_test = load_mnist_dataset('test', path)

    # And return all the data
    return X, y, X_test, y_test


# Create dataset
X, y, X_test, y_test = create_data_mnist('fashion_mnist_images')

# Shuffle the training dataset
keys = np.array(range(X.shape[0]))
np.random.shuffle(keys)
X = X[keys]
y = y[keys]

# Scale and reshape samples
X = (X.reshape(X.shape[0], -1).astype(np.float32) - 127.5) / 127.5
X_test = (X_test.reshape(X_test.shape[0], -1).astype(np.float32) -
             127.5) / 127.5

# Instantiate the model
model = Model()
```

```
# Add layers
model.add(Layer_Dense(X.shape[1], 128))
model.add(Activation_ReLU())
model.add(Layer_Dense(128, 128))
model.add(Activation_ReLU())
model.add(Layer_Dense(128, 10))
model.add(Activation_Softmax())

# Set loss, optimizer and accuracy objects
model.set(
    loss=Loss_CategoricalCrossentropy(),
    optimizer=Optimizer_Adam(decay=1e-3),
    accuracy=Accuracy_Categorical()
)

# Finalize the model
model.finalize()

# Train the model
model.train(X, y, validation_data=(X_test, y_test),
            epochs=10, batch_size=128, print_every=100)
```

Supplementary Material: https://nnfs.io/ch19
Chapter code, further resources, and errata for this chapter.

Chapter 20

Model Evaluation

In Chapter 11, Testing or Out-of-Sample Data, we covered the differences between validation and testing data. With our model up to this point, we've validated during training, but currently have no great way to run a test on data or perform a prediction. To begin, we're going to add a new `evaluate` method to the `Model` class:

```
# Evaluates the model using passed-in dataset
def evaluate(self, X_val, y_val, *, batch_size=None):
```

This method takes in samples (*`X_val`*), target outputs (*`y_val`*), and an optional batch size. First, we calculate the number of steps given the length of the data and the *`batch_size`* argument. This is the same as in the `train` method:

```
        # Default value if batch size is not being set
        validation_steps = 1

        # Calculate number of steps
        if batch_size is not None:
            validation_steps = len(X_val) // batch_size
            # Dividing rounds down. If there are some remaining
            # data, but not a full batch, this won't include it
            # Add `1` to include this not full batch
            if validation_steps * batch_size < len(X_val):
                validation_steps += 1
```

Then, we're going to move a chunk of code from the `Model` class' `train` method:

```
# Model class
class Model:
    ...
    def train(self, X, y, *, epochs=1, batch_size=None,
              print_every=1, validation_data=None):
        ...
            ...
            # If there is the validation data
            if validation_data is not None:

                # Reset accumulated values in loss
                # and accuracy objects
                self.loss.new_pass()
                self.accuracy.new_pass()

                # Iterate over steps
                for step in range(validation_steps):

                    # If batch size is not set -
                    # train using one step and full dataset
                    if batch_size is None:
                        batch_X = X_val
                        batch_y = y_val

                    # Otherwise slice a batch
                    else:
                        batch_X = X_val[
                            step*batch_size:(step+1)*batch_size
                        ]
                        batch_y = y_val[
                            step*batch_size:(step+1)*batch_size
                        ]
```

```
            # Perform the forward pass
            output = self.forward(batch_X, training=False)

            # Calculate the loss
            self.loss.calculate(output, batch_y)

            # Get predictions and calculate an accuracy
            predictions = self.output_layer_activation.predictions(
                              output)
            self.accuracy.calculate(predictions, batch_y)

        # Get and print validation loss and accuracy
        validation_loss = self.loss.calculate_accumulated()
        validation_accuracy = self.accuracy.calculate_accumulated()

        # Print a summary
        print(f'validation, ' +
              f'acc: {validation_accuracy:.3f}, ' +
              f'loss: {validation_loss:.3f}')
```

We'll move that code, along with the code parts for the number of steps calculation and resetting accumulated loss and accuracy, to the `evaluate` method, making it:

```
    # Evaluates the model using passed-in dataset
    def evaluate(self, X_val, y_val, *, batch_size=None):

        # Default value if batch size is not being set
        validation_steps = 1

        # Calculate number of steps
        if batch_size is not None:
            validation_steps = len(X_val) // batch_size
            # Dividing rounds down. If there are some remaining
            # data, but not a full batch, this won't include it
            # Add `1` to include this not full minibatch
            if validation_steps * batch_size < len(X_val):
                validation_steps += 1

        # Reset accumulated values in loss
        # and accuracy objects
        self.loss.new_pass()
        self.accuracy.new_pass()

        # Iterate over steps
        for step in range(validation_steps):
```

```
            # If batch size is not set -
            # train using one step and full dataset
            if batch_size is None:
                batch_X = X_val
                batch_y = y_val

            # Otherwise slice a batch
            else:
                batch_X = X_val[
                    step*batch_size:(step+1)*batch_size
                ]
                batch_y = y_val[
                    step*batch_size:(step+1)*batch_size
                ]

            # Perform the forward pass
            output = self.forward(batch_X, training=False)

            # Calculate the loss
            self.loss.calculate(output, batch_y)

            # Get predictions and calculate an accuracy
            predictions = self.output_layer_activation.predictions(
                              output)
            self.accuracy.calculate(predictions, batch_y)

        # Get and print validation loss and accuracy
        validation_loss = self.loss.calculate_accumulated()
        validation_accuracy = self.accuracy.calculate_accumulated()

        # Print a summary
        print(f'validation, ' +
              f'acc: {validation_accuracy:.3f}, ' +
              f'loss: {validation_loss:.3f}')
```

Now, where that block of code once was in the `Model` class' `train` method, we can call the new `evaluate` method:

```
# Model class
class Model:
    ...
    def train(self, X, y, *, epochs=1, batch_size=None,
              print_every=1, validation_data=None):
        ...
            ...
            # If there is the validation data
            if validation_data is not None:
```

```
                # Evaluate the model:
                self.evaluate(*validation_data,
                              batch_size=batch_size)
```

If you're confused about the `*validation_data` part, the asterisk, called the **starred expression**, unpacks the `validation_data` list into singular values. For a simple example of how this works:

```
a = (1, 2)

def test(n1, n2):
    print(n1, n2)

test(*a)


>>>
1 2
```

Now that we have this separate `evaluate` method, we can evaluate the model whenever we please — either during training or on-demand, by passing the validation or testing data. First, we'll create and train a model as usual:

```
# Create dataset
X, y, X_test, y_test = create_data_mnist('fashion_mnist_images')

# Shuffle the training dataset
keys = np.array(range(X.shape[0]))
np.random.shuffle(keys)
X = X[keys]
y = y[keys]

# Scale and reshape samples
X = (X.reshape(X.shape[0], -1).astype(np.float32) - 127.5) / 127.5
X_test = (X_test.reshape(X_test.shape[0], -1).astype(np.float32) -
          127.5) / 127.5

# Instantiate the model
model = Model()

# Add layers
model.add(Layer_Dense(X.shape[1], 128))
model.add(Activation_ReLU())
model.add(Layer_Dense(128, 128))
model.add(Activation_ReLU())
model.add(Layer_Dense(128, 10))
model.add(Activation_Softmax())
```

```
# Set loss, optimizer and accuracy objects
model.set(
    loss=Loss_CategoricalCrossentropy(),
    optimizer=Optimizer_Adam(decay=1e-3),
    accuracy=Accuracy_Categorical()
)

# Finalize the model
model.finalize()

# Train the model
model.train(X, y, validation_data=(X_test, y_test),
            epochs=10, batch_size=128, print_every=100)
```

We can then add code to evaluate. Right now, we don't have any specific testing data besides what we've used for validation data, but we can use this, for now, to test this method:

```
model.evaluate(X_test, y_test)
```

Running this, we get:

```
>>>
...
epoch: 10
step: 0, acc: 0.891, loss: 0.263 (data_loss: 0.263, reg_loss: 0.000), lr:
0.0001915341888527102
step: 100, acc: 0.883, loss: 0.257 (data_loss: 0.257, reg_loss: 0.000), lr:
0.00018793459875963167
step: 200, acc: 0.922, loss: 0.227 (data_loss: 0.227, reg_loss: 0.000), lr:
0.00018446781036709093
step: 300, acc: 0.898, loss: 0.282 (data_loss: 0.282, reg_loss: 0.000), lr:
0.00018112660749864155
step: 400, acc: 0.914, loss: 0.299 (data_loss: 0.299, reg_loss: 0.000), lr:
0.00017790428749332856
step: 468, acc: 0.917, loss: 0.192 (data_loss: 0.192, reg_loss: 0.000), lr:
0.00017577781683951485
training, acc: 0.894, loss: 0.291 (data_loss: 0.291, reg_loss: 0.000), lr:
0.00017577781683951485
validation, acc: 0.874, loss: 0.354
validation, acc: 0.874, loss: 0.354
```

The validation accuracy and loss are repeated twice and show the same values at the end since we're validating during the training and evaluating right after on the same data. You'll often train a model, tweak its hyperparameters, train it all over again, and so on, using training and validation data passed into the training method. Then, whenever you find the model and hyperparameters that appear to perform the best, you'll use that model on testing data and, in the future, to make predictions in production.

Next, we can also run evaluation on the training data:

```
model.evaluate(X, y)
```

Running this prints:

```
>>>
validation, acc: 0.895, loss: 0.285
```

"Validation" here means that we evaluated the model, but we have done this using the training data. We compare that to the result of training on this data which we have just performed:

```
training, acc: 0.894, loss: 0.291 (data_loss: 0.291, reg_loss: 0.000), lr:
0.00017577781683951485
```

You may notice that, despite using the same dataset, there is some difference between accuracy and loss values. This difference comes from the fact that the model prints accuracy and loss accumulated during the epoch, while the model was still learning; thus, mean accuracy and loss differ from the evaluation on the training data that has been run after the last epoch of training. **Running evaluation on the training data at the end of the training process will return the final accuracy and loss.**

In the next chapter, we will add the ability to save and load our models; we'll also construct a way to retrieve and set a model's parameters.

Supplementary Material: https://nnfs.io/ch20
Chapter code, further resources, and errata for this chapter.

Chapter 21

Saving and Loading Models and Their Parameters

Retrieving Parameters

There are situations where we'd like to take a closer look into model parameters to see if we have dead or exploding neurons. To retrieve these parameters, we will iterate over the trainable layers, take their parameters, and put them into a list. The only trainable layer type that we have here is the *Dense* layer. Let's add a method to the `Layer_Dense` class to retrieve parameters:

```
# Dense layer
class Layer_Dense:
    ...
    # Retrieve layer parameters
    def get_parameters(self):
        return self.weights, self.biases
```

Within the `Model` class, we'll add `get_parameters` method, which will iterate over the trainable layers of the model, run their `get_parameters` method, and append returned weights and biases to a list:

```
# Model class
class Model:
    ...
    # Retrieves and returns parameters of trainable layers
    def get_parameters(self):

        # Create a list for parameters
        parameters = []

        # Iterable trainable layers and get their parameters
        for layer in self.trainable_layers:
            parameters.append(layer.get_parameters())

        # Return a list
        return parameters
```

Now, after training a model, we can grab the parameters by running:

```
parameters = model.get_parameters()
```

For example:

```
# Create dataset
X, y, X_test, y_test = create_data_mnist('fashion_mnist_images')

# Shuffle the training dataset
keys = np.array(range(X.shape[0]))
np.random.shuffle(keys)
X = X[keys]
y = y[keys]

# Scale and reshape samples
X = (X.reshape(X.shape[0], -1).astype(np.float32) - 127.5) / 127.5
X_test = (X_test.reshape(X_test.shape[0], -1).astype(np.float32) -
             127.5) / 127.5

# Instantiate the model
model = Model()
# Add layers
model.add(Layer_Dense(X.shape[1], 128))
model.add(Activation_ReLU())
model.add(Layer_Dense(128, 128))
model.add(Activation_ReLU())
model.add(Layer_Dense(128, 10))
model.add(Activation_Softmax())

# Set loss, optimizer and accuracy objects
model.set(
    loss=Loss_CategoricalCrossentropy(),
    optimizer=Optimizer_Adam(decay=1e-3),
    accuracy=Accuracy_Categorical()
)

# Finalize the model
model.finalize()

# Train the model
model.train(X, y, validation_data=(X_test, y_test),
            epochs=10, batch_size=128, print_every=100)

# Retrieve and print parameters
parameters = model.get_parameters()
print(parameters)
```

This will look *something* like (we trim the output to save space):

```
[(array([[ 0.03538642,  0.00794717, -0.04143231, ...,  0.04267325,
        -0.00935107,  0.01872394],
       [ 0.03289384,  0.00691249, -0.03424096, ...,  0.02362755,
        -0.00903602,  0.00977725],
       [ 0.02189022, -0.01362374, -0.01442819, ...,  0.01320345,
        -0.02083327,  0.02499157],
       ...,
       [ 0.0146937 , -0.02869027, -0.02198809, ...,  0.01459295,
        -0.02335824,  0.00935643],
       [-0.00090149,  0.01082182, -0.06013806, ...,  0.00704454,
        -0.0039093 ,  0.00311571],
       [ 0.03660082, -0.00809607, -0.02737131, ...,  0.02216582,
        -0.01710589,  0.01578414]], dtype=float32), array([[-2.24505737e-02,
5.40090213e-03,  2.91307438e-02,
        -1.04323691e-02, -9.52822249e-03, -1.48109728e-02,
       ...,
         0.04158591, -0.01614098, -0.0134403 ,  0.00708392,  0.0284729 ,
         0.00336277, -0.00085383,  0.00163819]], dtype=float32)),
(array([[-0.00196577, -0.00335329, -0.01362851, ...,  0.00397028,
         0.00027816,  0.00427755],
       [ 0.04438829, -0.09197803,  0.02897452, ..., -0.11920264,
         0.03808296, -0.00536136],
       [ 0.04146343, -0.03637529,  0.04973305, ..., -0.13564698,
        -0.08259197, -0.02467288],
       ...,
       [ 0.03495856,  0.03902597,  0.0028984 , ..., -0.10016892,
        -0.11356542,  0.05866433],
       [-0.00857899, -0.02612676, -0.01050871, ..., -0.00551328,
        -0.01432311, -0.00916382],
       [-0.20444085, -0.01483698, -0.09321352, ...,  0.02114356,
        -0.0762504 ,  0.03600615]], dtype=float32), array([[-0.0103433 ,
-0.00158314,  0.02268587, -0.02352985, -0.02144126,
        -0.00777614,  0.00795028, -0.00622872,  0.06918745, -0.00743477]],
      dtype=float32))]
```

Setting Parameters

If we have a method to get parameters, we will likely also want to have a method that will set parameters. We'll do this similar to how we setup the `get_parameters` method, starting with the `Layer_Dense` class:

```
# Dense layer
class Layer_Dense:
    ...
    # Set weights and biases in a layer instance
    def set_parameters(self, weights, biases):
        self.weights = weights
        self.biases = biases
```

Then we can update the `Model` class:

```
# Model class
class Model:
    ...
    # Updates the model with new parameters
    def set_parameters(self, parameters):

        # Iterate over the parameters and layers
        # and update each layers with each set of the parameters
        for parameter_set, layer in zip(parameters,
                                        self.trainable_layers):
            layer.set_parameters(*parameter_set)
```

We are also iterating over the trainable layers here, but what we are doing next needs a bit more explanation. First, the `zip()` function takes in iterables, like lists, and returns a new iterable with pairwise combinations of all the iterables passed in as parameters. In other words (and using our example), `zip()` takes a list of parameters and a list of layers and returns an iterator containing tuples of 0th elements of both lists, then the 1st elements of both lists, the 2nd elements from both lists, and so on. This way, we can iterate over parameters and the layer they belong to at the same time. As our parameters are a tuple of weights and biases, we will unpack them with a starred expression so that our `Layer_Dense` method can take them as separate parameters. This approach gives us flexibility if we'd like to use layers with different numbers of parameter groups.

One difference that presents itself now is that this allows us to have a model that never needed an optimizer. If we don't train a model but, instead, load already trained parameters into it, we won't optimize anything. To account for this, we'll visit the `finalize` method of the `Model` class, changing:

```
# Model class
class Model:
    ...
    # Finalize the model
    def finalize(self):
        ...
        # Update loss object with trainable layers
        self.loss.remember_trainable_layers(
            self.trainable_layers
        )
```

To (we added an `if` statement to set a list of trainable layers to the loss function, only if this loss object exists):

```
# Model class
class Model:
    ...
    # Finalize the model
    def finalize(self):
        ...
        # Update loss object with trainable layers
        if self.loss is not None:
            self.loss.remember_trainable_layers(
                self.trainable_layers
            )
```

Next, we'll change the `Model` class' `set` method to allow us to pass in only given parameters. We'll assign default values and add `if` statements to use parameters only when they're present. To do that, we'll change:

```
    # Set loss, optimizer and accuracy
    def set(self, *, loss, optimizer, accuracy):
        self.loss = loss
        self.optimizer = optimizer
        self.accuracy = accuracy
```

To:

```
    # Set loss, optimizer and accuracy
    def set(self, *, loss=None, optimizer=None, accuracy=None):

        if loss is not None:
            self.loss = loss

        if optimizer is not None:
            self.optimizer = optimizer

        if accuracy is not None:
            self.accuracy = accuracy
```

We can now train a model, retrieve its parameters, create a new model, and set its parameters with those retrieved from the previously-trained model:

```
# Create dataset
X, y, X_test, y_test = create_data_mnist('fashion_mnist_images')

# Shuffle the training dataset
keys = np.array(range(X.shape[0]))
np.random.shuffle(keys)
X = X[keys]
y = y[keys]

# Scale and reshape samples
X = (X.reshape(X.shape[0], -1).astype(np.float32) - 127.5) / 127.5
X_test = (X_test.reshape(X_test.shape[0], -1).astype(np.float32) -
             127.5) / 127.5

# Instantiate the model
model = Model()

# Add layers
model.add(Layer_Dense(X.shape[1], 128))
model.add(Activation_ReLU())
model.add(Layer_Dense(128, 128))
model.add(Activation_ReLU())
model.add(Layer_Dense(128, 10))
model.add(Activation_Softmax())

# Set loss, optimizer and accuracy objects
model.set(
    loss=Loss_CategoricalCrossentropy(),
    optimizer=Optimizer_Adam(decay=1e-3),
    accuracy=Accuracy_Categorical()
)
```

```
# Finalize the model
model.finalize()

# Train the model
model.train(X, y, validation_data=(X_test, y_test),
            epochs=10, batch_size=128, print_every=100)

# Retrieve model parameters
parameters = model.get_parameters()

# New model

# Instantiate the model
model = Model()

# Add layers
model.add(Layer_Dense(X.shape[1], 128))
model.add(Activation_ReLU())
model.add(Layer_Dense(128, 128))
model.add(Activation_ReLU())
model.add(Layer_Dense(128, 10))
model.add(Activation_Softmax())

# Set loss and accuracy objects
# We do not set optimizer object this time - there's no need to do it
# as we won't train the model
model.set(
    loss=Loss_CategoricalCrossentropy(),
    accuracy=Accuracy_Categorical()
)

# Finalize the model
model.finalize()

# Set model with parameters instead of training it
model.set_parameters(parameters)

# Evaluate the model
model.evaluate(X_test, y_test)
```

```
>>>
(model training output removed)
validation, acc: 0.874, loss: 0.354
validation, acc: 0.874, loss: 0.354
```

Saving Parameters

We'll extend this further now by actually saving the parameters into a file. To do this, we'll add a `save_parameters` method in the `Model` class. We'll use Python's built-in *pickle* module to serialize any Python object. Serialization is a process of turning an object, which can be of any abstract form, into a binary representation — a set of bytes that can be, for example, saved into a file. This serialized form contains all the information needed to recreate the object later. *Pickle* can either return the bytes of the serialized object or save them directly to a file. We'll make use of the latter ability, so let's import *pickle*:

```
import pickle
```

Then we'll add a new method to the `Model` class. Before having *pickle* save our parameters into a file, we'll need to create a file-handler by opening a file in binary-write mode. We will then pass this handler along to the data into `pickle.dump()`. To create the file, we need a filename that we'll save the data into; we'll pass it in as a parameter:

```
# Model class
class Model:
    ...
    # Saves the parameters to a file
    def save_parameters(self, path):

        # Open a file in the binary-write mode
        # and save parameters to it
        with open(path, 'wb') as f:
            pickle.dump(self.get_parameters(), f)
```

With this method, you can save the parameters of a trained model by running:

```
model.save_parameters('fashion_mnist.parms')
```

Loading Parameters

Presumably, if we are saving model parameters into a file, we would also like to have a way to load them from this file. Loading parameters is very similar to saving the parameters, just reversed. We'll open the file in a binary-read mode and have *pickle* read from it, deserializing parameters back into a list. Then we call the `set_parameters` method that we created earlier and pass in the loaded parameters:

```
    # Loads the weights and updates a model instance with them
    def load_parameters(self, path):

        # Open file in the binary-read mode,
        # load weights and update trainable layers
        with open(path, 'rb') as f:
            self.set_parameters(pickle.load(f))
```

We set up a model, load in the parameters file (we did not train this model), and test the model to check if it works:

```
# Create dataset
X, y, X_test, y_test = create_data_mnist('fashion_mnist_images')

# Shuffle the training dataset
keys = np.array(range(X.shape[0]))
np.random.shuffle(keys)
X = X[keys]
y = y[keys]

# Scale and reshape samples
X = (X.reshape(X.shape[0], -1).astype(np.float32) - 127.5) / 127.5
X_test = (X_test.reshape(X_test.shape[0], -1).astype(np.float32) -
          127.5) / 127.5

# Instantiate the model
model = Model()
```

```
# Add layers
model.add(Layer_Dense(X.shape[1], 128))
model.add(Activation_ReLU())
model.add(Layer_Dense(128, 128))
model.add(Activation_ReLU())
model.add(Layer_Dense(128, 10))
model.add(Activation_Softmax())

# Set loss and accuracy objects
# We do not set optimizer object this time - there's no need to do it
# as we won't train the model
model.set(
    loss=Loss_CategoricalCrossentropy(),
    accuracy=Accuracy_Categorical()
)

# Finalize the model
model.finalize()

# Set model with parameters instead of training it
model.load_parameters('fashion_mnist.parms')

# Evaluate the model
model.evaluate(X_test, y_test)
```

```
>>>
validation, acc: 0.874, loss: 0.354
```

While we can save and load model parameter values, we still need to define the model. It must be the exact configuration as the model that we're importing parameters from. It would be easier if we could save the model itself.

Saving the Model

Why didn't we save the whole model in the first place? Saving just weights versus saving the whole model has different use cases along with pros and cons. With saved weights, you can, for example, initialize a model with those weights, trained from similar data, and then train that model to work with your specific data. This is called **transfer learning** and is outside of the scope of this book. Weights can be used to visualize the model (like in some animations that we have created for the purpose of this book, starting from chapter 6), identify dead neurons, implement more complicated models (like **reinforcement learning**, where weights collected from multiple models are committed to a single network), and so on. A file containing just weights is also much smaller than an entire model. A model initialized from weights loads faster and uses less memory, as the optimizer and related parts are not created. One downside of loading just weights and biases is that the initialized model does not contain the optimizer's state. It is possible to train the model further, but it's more optimal to load a full model if we intend to train it. When saving the full model, everything related to it is saved as well; this includes the optimizer's state (that allows us to easily continue the training) and model's structure.

We'll create another method in the `Model` class that we'll use to save the entire model. The first thing we'll do is make a copy of the model since we're going to edit it before saving, and we may also want to save a model during the training process as a **checkpoint**.

```
    # Saves the model
    def save(self, path):

        # Make a deep copy of current model instance
        model = copy.deepcopy(self)
```

We import the *copy* module to support this:

```
import copy
```

The *copy* module offers two methods that allow us to copy the model — *copy* and *deepcopy*. While *copy* is faster, it only copies the first level of the object's properties, causing copies of our model objects to have some references common to the original model. For example, our model object has a list of layers — the list is the top-level property, and the layers themselves

are secondary — therefore, references to the layer objects will be shared by both the original and copied model objects. Due to these challenges with *copy*, we'll use the *deepcopy* method to recursively traverse all objects and create a full copy.

Next, we'll remove the accumulated loss and accuracy:

```
# Reset accumulated values in loss and accuracy objects
model.loss.new_pass()
model.accuracy.new_pass()
```

Then remove any data in the input layer, and reset the gradients, if any exist:

```
# Remove data from input layer
# and gradients from the loss object
model.input_layer.__dict__.pop('output', None)
model.loss.__dict__.pop('dinputs', None)
```

Both `model.input_layer` and `model.loss` are class instances. They're attributes of the `Model` object but also objects themselves. One of the dunder properties (called "dunder" because of the double underscores) that exists for all classes is the `__dict__` property. It contains names and values for the class object's properties. We can then use the built-in `pop` method on these values, which means we remove them from that instance of the class' object. The `pop` method will wind up throwing an error if the key we pass as the first parameter doesn't exist, as the `pop` method wants to return the value of the key that it removes. We use the second parameter of the `pop` method — which is the default value that we want to return if the key doesn't exist — to prevent these errors. We will set this parameter to *None* — we do not intend to catch the removed values, and it doesn't really matter what the default value is. This way, we do not have to check if a given property exists, in times like when we'd like to delete it using the *del* statement, and some of them might not exist.

Next, we'll iterate over all the layers to remove their properties:

```
# For each layer remove inputs, output and dinputs properties
for layer in model.layers:
    for property in ['inputs', 'output', 'dinputs',
                     'dweights', 'dbiases']:
        layer.__dict__.pop(property, None)
```

With these things cleaned up, we can save the model object. To do that, we have to open a file in a binary-write mode, and call `pickle.dump()` with the model object and the file handler as parameters:

```
# Open a file in the binary-write mode and save the model
with open(path, 'wb') as f:
    pickle.dump(model, f)
```

This makes the full `save` method:

```
    # Saves the model
    def save(self, path):

        # Make a deep copy of current model instance
        model = copy.deepcopy(self)

        # Reset accumulated values in loss and accuracy objects
        model.loss.new_pass()
        model.accuracy.new_pass()

        # Remove data from the input layer
        # and gradients from the loss object
        model.input_layer.__dict__.pop('output', None)
        model.loss.__dict__.pop('dinputs', None)

        # For each layer remove inputs, output and dinputs properties
        for layer in model.layers:
            for property in ['inputs', 'output', 'dinputs',
                             'dweights', 'dbiases']:
                layer.__dict__.pop(property, None)

        # Open a file in the binary-write mode and save the model
        with open(path, 'wb') as f:
            pickle.dump(model, f)
```

This means we can train a model, then save it whenever we wish with:

```
model.save('fashion_mnist.model')
```

Loading the Model

Loading a model will ideally take place before a model object even exists. What we mean by this is we could load a model by calling a method of the `Model` class instead of the object:

```
model = Model.load('fashion_mnist.model')
```

To achieve this, we're going to use the `@staticmethod` decorator. This decorator can be used with class methods to run them on uninitialized objects, where the *`self`* does not exist (notice that it is missing the function definition). In our case, we're going to use it to immediately create a model object without first needing to instantiate a model object. Within this method, we'll open a file using the passed-in path, in binary-read mode, and use pickle to deserialize the saved model:

```
    # Loads and returns a model
    @staticmethod
    def load(path):

        # Open file in the binary-read mode, load a model
        with open(path, 'rb') as f:
            model = pickle.load(f)

        # Return a model
        return model
```

Since we already have a saved model, let's create the data, and then load a model to see if it works:

```
# Create dataset
X, y, X_test, y_test = create_data_mnist('fashion_mnist_images')

# Shuffle the training dataset
keys = np.array(range(X.shape[0]))
np.random.shuffle(keys)
X = X[keys]
y = y[keys]
```

```
# Scale and reshape samples
X = (X.reshape(X.shape[0], -1).astype(np.float32) - 127.5) / 127.5
X_test = (X_test.reshape(X_test.shape[0], -1).astype(np.float32) -
             127.5) / 127.5

# Load the model
model = Model.load('fashion_mnist.model')

# Evaluate the model
model.evaluate(X_test, y_test)


>>>
validation, acc: 0.874, loss: 0.354
```

Saving the full trained model is a common way of saving a model. It saves parameters (weights and biases) and instances of all the model's objects and the data they generated. That is going to be, for example, the optimizer state like cache, learning rate decay, full model structure, etc. Loading the model, in this case, is as easy as calling one method and the model is ready to use, whether we want to continue training it or use it for a prediction.

Supplementary Material: https://nnfs.io/ch21
Chapter code, further resources, and errata for this chapter.

Chapter 22

Prediction / Inference

While we often spend most of our time focusing on training and testing models, the whole reason we're doing any of this is to have a model that takes new inputs and produces desired outputs. This will typically involve many attempts to train the best model possible, save that model, and load that saved model to do inference, or prediction.

In the case of Fashion MNIST classification, we'd like to load a trained model, show it never-before-seen images, and have it predict the correct classification. To do this, we'll add a new `predict` method to the `Model` class:

```
# Predicts on the samples
def predict(self, X, *, batch_size=None):
```

Note that we predict *`X`* with a possible *`batch_size`*. This means all predictions, including predictions on just one sample, will still be fed in as a list of samples in the form of a NumPy array, whose first dimension is the list samples, and second is sample data. For example, if we would like to predict on a single image, we still need to create a NumPy array mimicking a list containing a single sample — with a shape of *(1, 784)* where *1* is this single sample, and *784* is

the number of features in a sample (pixels per image). Similar to the `evaluate` method, we'll calculate the number of steps we plan to take:

```
# Default value if batch size is not being set
prediction_steps = 1

# Calculate number of steps
if batch_size is not None:
    prediction_steps = len(X) // batch_size
    # Dividing rounds down. If there are some remaining
    # data, but not a full batch, this won't include it
    # Add `1` to include this not full batch
    if prediction_steps * batch_size < len(X):
        prediction_steps += 1
```

Then create a list that we'll populate with the predictions:

```
# Model outputs
output = []
```

We'll iterate over the batches, passing the samples for predictions forward through the network, and populating the `output` with the predictions:

```
# Iterate over steps
for step in range(prediction_steps):

    # If batch size is not set -
    # train using one step and full dataset
    if batch_size is None:
        batch_X = X

    # Otherwise slice a batch
    else:
        batch_X = X[step*batch_size:(step+1)*batch_size]

    # Perform the forward pass
    batch_output = self.forward(batch_X, training=False)

    # Append batch prediction to the list of predictions
    output.append(batch_output)
```

After running this, the `output` is a list of batch predictions. Each of them is a NumPy array, a partial result made by predicting on a batch of samples from the input data array. Any applications, or programs, that will make use of the inference output of our models, we expect to simply pass in a list of samples and get back a list of predictions (both in the form of a NumPy array as mentioned before). Since we're not focused on training, we're only using batches in prediction

to ensure our model can fit into memory, but we're going to get a return that's also in batches of predictions. We can see a simple example of this:

```
import numpy as np

output = []

b = np.array([[1, 2], [3, 4]])
output.append(b)
b = np.array([[5, 6], [7, 8]])
output.append(b)
b = np.array([[9, 10], [11, 12]])
output.append(b)

print(output)

>>>
[array([[1, 2],
       [3, 4]]), array([[5, 6],
       [7, 8]]), array([[ 9, 10],
       [11, 12]])]
```

In this example, we see an output with a batch size of 2 and 6 total samples. The output is a list of arrays, with each array housing a batch of predictions. Instead, we want just 1 list of predictions, no more batches. To achieve this, we're going to use NumPy's `vstack` method:

```
import numpy as np

output = []

b = np.array([[1, 2], [3, 4]])
output.append(b)
b = np.array([[5, 6], [7, 8]])
output.append(b)
b = np.array([[9, 10], [11, 12]])
output.append(b)

output = np.vstack(output)

print(output)

>>>
[[ 1  2]
 [ 3  4]
 [ 5  6]
 [ 7  8]
 [ 9 10]
 [11 12]]
```

It takes a list of objects and stacks them, if possible, creating a homologous array. This is a preferable form of the return from the `predict` method when we pass a list of samples. With plain Python, we might just add to the list each step:

```
output = []

b = [[1, 2], [3, 4]]
output += b
b = [[5, 6], [7, 8]]
output += b
b = [[9, 10], [11, 12]]
output += b

print(output)


>>>
[[1, 2], [3, 4], [5, 6], [7, 8], [9, 10], [11, 12]]
```

We add results to a list and stack them at the end, instead of appending to the NumPy array each batch to avoid a performance penalty. Unlike plain Python, NumPy is written in *C* language and creates data objects in memory differently. That means that there is no easy way of adding data to the existing NumPy array, other than merging two arrays and saving the result as a new array. But this will lead to a performance penalty, since the further in predictions we are, the bigger the resulting array is. The fastest and most optimal way is to append NumPy arrays to a list and stack them vertically at once when we have collected all of the partial results.
We'll add the `np.vstack` to the end of the outputs that we return:

```
        # Stack and return results
        return np.vstack(output)
```

Making our full `predict` method:

```
    # Predicts on the samples
    def predict(self, X, *, batch_size=None):

        # Default value if batch size is not being set
        prediction_steps = 1

        # Calculate number of steps
        if batch_size is not None:
            prediction_steps = len(X) // batch_size
            # Dividing rounds down. If there are some remaining
            # data, but not a full batch, this won't include it
            # Add `1` to include this not full batch
            if prediction_steps * batch_size < len(X):
                prediction_steps += 1

        # Model outputs
        output = []

        # Iterate over steps
        for step in range(prediction_steps):

            # If batch size is not set -
            # train using one step and full dataset
            if batch_size is None:
                batch_X = X

            # Otherwise slice a batch
            else:
                batch_X = X[step*batch_size:(step+1)*batch_size]

            # Perform the forward pass
            batch_output = self.forward(batch_X, training=False)

            # Append batch prediction to the list of predictions
            output.append(batch_output)

        # Stack and return results
        return np.vstack(output)
```

Now we can load the model and test the prediction functionality:

```
# Create dataset
X, y, X_test, y_test = create_data_mnist('fashion_mnist_images')

# Scale and reshape samples
X_test = (X_test.reshape(X_test.shape[0], -1).astype(np.float32) -
          127.5) / 127.5
```

```
# Load the model
model = Model.load('fashion_mnist.model')

# Predict on the first 5 samples from validation dataset
# and print the result
confidences = model.predict(X_test[:5])
print(confidences)
```

```
>>>
[[9.6826810e-01 8.3330568e-05 1.0794386e-03 1.3643305e-03 7.6704117e-07
  5.5963554e-08 2.9197156e-02 8.6661328e-16 6.8134182e-06 1.8056496e-12]
 [7.7293724e-01 2.0613789e-03 9.3451981e-04 9.0647154e-02 3.4899445e-04
  2.0565639e-07 1.3301854e-01 6.3095896e-12 5.2045987e-05 7.7830048e-11]
 [9.4310820e-01 5.1831361e-05 1.4724518e-03 8.1068231e-04 7.9751426e-06
  9.9619001e-07 5.4532889e-02 2.9622423e-13 1.4997837e-05 2.2963499e-10]
 [9.8930722e-01 1.2575739e-04 2.5738587e-04 1.4423713e-04 2.5113836e-06
  5.6183376e-07 1.0156924e-02 2.8593078e-13 5.5162018e-06 1.4746830e-10]
 [9.2869467e-01 7.3713978e-04 1.7579789e-03 2.1864739e-03 1.7945129e-05
  1.9282908e-05 6.6521421e-02 5.1533548e-11 6.5157568e-05 7.2020221e-09]]
```

It looks like it's working! After spending so much time training and finding the best hyperparameters, a common issue people have is actually *using* the model. As a reminder, each of the subarrays in the output is a vector of confidences containing a confidence metric per class. The first thing that we need to do in this case is to gather the argmax values of these confidence vectors. Recall that we're using a softmax classifier, so this neural network is attempting to fit to one-hot vectors, where the correct class is represented by a 1, and the others by 0s. When doing inference, it is unlikely to achieve such a perfect result, but the index associated with the highest value in the output is what we determine the model is predicting; we're just using the argmax. We could write code to do this, but we've already done that in all of the activation function classes, where we added a `predictions` method:

```
# Softmax activation
class Activation_Softmax:
    ...
    # Calculate predictions for outputs
    def predictions(self, outputs):
        return np.argmax(outputs, axis=1)
```

We've also set an attribute in our model with the output layer's activation function, which means we can generically acquire predictions by performing:

```
# Load the model
model = Model.load('fashion_mnist.model')

# Predict on the first 5 samples from validation dataset, print the result
confidences = model.predict(X_test[:5])
predictions = model.output_layer_activation.predictions(confidences)
print(predictions)

# Print first 5 labels
print(y_test[:5])
```

```
>>>
[0 0 0 0 0]
[0 0 0 0 0]
```

In this case, our model predicted all "class 0," and our test labels were all class 0 as well. Since shuffling our testing data isn't essential, we never shuffled them, so they're going in the original order like our training data was. This explains why all these predictions are 0s.

In practice, we don't care what class number something is, we want to know *what* it is. In this case, class numbers map directly to names, so we add the following dictionary to our code:

```
fashion_mnist_labels = {
    0: 'T-shirt/top',
    1: 'Trouser',
    2: 'Pullover',
    3: 'Dress',
    4: 'Coat',
    5: 'Sandal',
    6: 'Shirt',
    7: 'Sneaker',
    8: 'Bag',
    9: 'Ankle boot'
}
```

Then we could get the string classification by performing:

```
for prediction in predictions:
    print(fashion_mnist_labels[prediction])
```

```
>>>
T-shirt/top
T-shirt/top
T-shirt/top
T-shirt/top
T-shirt/top
```

This is great, but we still have to *actually* predict something instead of the training data. When covering deep learning, the training steps often get all the focus; we want to see those accuracy and loss metrics look good! It works well to focus on training for tutorials that aim to show people how to *use* a framework, but one of the larger pain points we see is *applying* the models in production, or just running predictions on new data that was sourced from the wild (especially since outside data is rarely formatted to match your training data).

At the moment, we have a model trained on items of clothing, so we need some truly new samples. Luckily, you're probably a person who owns some clothes; if so, you can take photos of those to start with. If not, use the following sample photos:

https://nnfs.io/datasets/tshirt.png

Fig 20.01: Hand-made t-shirt image for the purpose of inference.

https://nnfs.io/datasets/pants.png

Fig 20.02: Hand-made pants image for the purpose of inference.

You can also try your hand at hand-drawing samples like these. Once you have new images/samples that you wish to use in production, you'll need to preprocess them in the same way the training samples were. Some of these changes are fairly difficult to forget, like the image resolution or number of color channels; we'd get an error if we didn't do those things. Let's start preprocessing our image by loading it in. We'll use the *cv2* package to read in the image:

```
import cv2

image_data = cv2.imread('tshirt.png', cv2.IMREAD_UNCHANGED)
```

We can view the image:

```
import matplotlib.pyplot as plt
plt.imshow(cv2.cvtColor(image_data, cv2.COLOR_BGR2RGB))
plt.show()
```

Fig 20.03: Hand-made t-shirt image loaded with Python.

Note that we're doing *cv2.cvtColor* because OpenCV uses BGR (blue, green, red pixel values) color format by default, but matplotlib uses RGB (red, green, blue), so we're converting the colormap to display the image.

The first thing we'll do is read this image as grayscale instead of RGB. This is in contrast to the Fashion MNIST images, which are grayscaled, and we have used `cv2.IMREAD_UNCHANGED` as a parameter to the `cv2.imread()` to inform OpenCV that our intention is to read images grayscaled and unchanged. Here, we have a color image, and this parameter won't work as "unchanged" means containing all the colors; thus, we'll use `cv2.IMREAD_GRAYSCALE` to force grayscaling when we read in our image:

```
import cv2
image_data = cv2.imread('tshirt.png', cv2.IMREAD_GRAYSCALE)
```

Then we can display it:

```
import matplotlib.pyplot as plt
plt.imshow(image_data, cmap='gray')
plt.show()
```

Note that we use a gray colormap with `plt.imshow()` by passing the `'gray'` argument into the *`cmap`* parameter. The result is a grayscale image:

Fig 20.04: Grayscaled hand-made t-shirt image loaded with Python.

Next, we'll resize the image to be the same 28x28 resolution as our training data:

```
image_data = cv2.resize(image_data, (28, 28))
```

We then display this resized image:

```
plt.imshow(image_data, cmap='gray')
plt.show()
```

Fig 20.05: Grayscaled and scaled down hand-made t-shirt image.

Next, we'll flatten and scale the image. While the scale operation is the same as for the training data, the flattening is a bit different; we don't have a list of images but a single image, and, as previously explained, a single image must be passed in as a list containing this single image. We flatten by applying `.reshape(1, -1)` to the image. The `1` argument represents the number of samples, and the `-1` flattens the image to a vector of length 784. This produces a 1x784 array with our one sample and 784 features (i.e., 28x28 pixels):

```
import numpy as np

image_data = (image_data.reshape(1, -1).astype(np.float32) - 127.5) / 127.5
```

Now we can load in our model and predict on this image data:

```
# Load the model
model = Model.load('fashion_mnist.model')

# Predict on the image
confidences = model.predict(image_data)

# Get prediction instead of confidence levels
predictions = model.output_layer_activation.predictions(confidences)

# Get label name from label index
prediction = fashion_mnist_labels[predictions[0]]

print(prediction)
```

Making our code up to this point that loads, preprocesses, and predicts:

```
# Label index to label name relation
fashion_mnist_labels = {
    0: 'T-shirt/top',
    1: 'Trouser',
    2: 'Pullover',
    3: 'Dress',
    4: 'Coat',
    5: 'Sandal',
    6: 'Shirt',
    7: 'Sneaker',
    8: 'Bag',
    9: 'Ankle boot'
}

# Read an image
image_data = cv2.imread('tshirt.png', cv2.IMREAD_GRAYSCALE)
```

```
# Resize to the same size as Fashion MNIST images
image_data = cv2.resize(image_data, (28, 28))

# Reshape and scale pixel data
image_data = (image_data.reshape(1, -1).astype(np.float32) - 127.5) / 127.5

# Load the model
model = Model.load('fashion_mnist.model')

# Predict on the image
predictions = model.predict(image_data)

# Get prediction instead of confidence levels
predictions = model.output_layer_activation.predictions(predictions)

# Get label name from label index
prediction = fashion_mnist_labels[predictions[0]]

print(prediction)
```

Note that we are using `predictions[0]` as we passed in a single image in the form of a list, and the model returns a list containing a single prediction.

Only one problem…

```
>>>
Ankle boot
```

What's wrong? Let's compare our currently-preprocessed image to the training data:

```
import matplotlib.pyplot as plt
import cv2

mnist_image = cv2.imread('fashion_mnist_images/train/0/0000.png',
                         cv2.IMREAD_UNCHANGED)
plt.imshow(mnist_image, cmap='gray')
plt.show()
```

Fig 20.06: Example t-shirt image from the Fashion MNIST dataset

Now we compare this original and example training image to our's:

Fig 20.07: Grayscaled and scaled down hand-made t-shirt image.

The training data that we've used is color-inverted (i.e., the background is black instead of white, and so on). To invert our image before scaling, we can use pixel math directly instead of using OpenCV. We'll subtract all the pixel values from the maximum pixel value: 255. For example, a value of *0* will become *255 - 0 = 255*, and the value of *255* will become *255 - 255 = 0*.

```
image_data = 255 - image_data
```

With this small change, our prediction code becomes:

```
# Read an image
image_data = cv2.imread('tshirt.png', cv2.IMREAD_GRAYSCALE)

# Resize to the same size as Fashion MNIST images
image_data = cv2.resize(image_data, (28, 28))

# Invert image colors
image_data = 255 - image_data
```

```
# Reshape and scale pixel data
image_data = (image_data.reshape(1, -1).astype(np.float32) - 127.5) / 127.5

# Load the model
model = Model.load('fashion_mnist.model')

# Predict on the image
confidences = model.predict(image_data)

# Get prediction instead of confidence levels
predictions = model.output_layer_activation.predictions(confidences)

# Get label name from label index
prediction = fashion_mnist_labels[predictions[0]]

print(prediction)


>>>
T-shirt/top
```

Now it works! The reason it works now, and not work previously, is from how the *Dense* layers work — they learn feature (pixel in this case) values and the correlation between them. Contrast this with convolutional layers, which are being trained to find and understand features on images (not features as data input nodes, but actual characteristics/traits, such as lines and curves). Because pixel values were very different, the model incorrectly put its "guess" in this case. Convolutional layers may properly predict in this case, as-is.

Let's try the pants:

```
import matplotlib.pyplot as plt
import cv2

image_data = cv2.imread('pants.png', cv2.IMREAD_UNCHANGED)
plt.imshow(cv2.cvtColor(image_data, cv2.COLOR_BGR2RGB))
plt.show()
```

Fig 20.08: Hand-made pants image loaded with Python.

Now we'll preprocess:

```
# Read an image
image_data = cv2.imread('pants.png', cv2.IMREAD_GRAYSCALE)

# Resize to the same size as Fashion MNIST images
image_data = cv2.resize(image_data, (28, 28))

# Invert image colors
image_data = 255 - image_data
```

Let's see what we have:

```
plt.imshow(image_data, cmap='gray')
plt.show()
```

Fig 20.09: Grayscaled and scaled down hand-made t-shirt image.

Making our code:

```
# Label index to label name relation
fashion_mnist_labels = {
    0: 'T-shirt/top',
    1: 'Trouser',
    2: 'Pullover',
    3: 'Dress',
    4: 'Coat',
    5: 'Sandal',
    6: 'Shirt',
    7: 'Sneaker',
    8: 'Bag',
    9: 'Ankle boot'
}

# Read an image
image_data = cv2.imread('pants.png', cv2.IMREAD_GRAYSCALE)

# Resize to the same size as Fashion MNIST images
image_data = cv2.resize(image_data, (28, 28))

# Invert image colors
image_data = 255 - image_data

# Reshape and scale pixel data
image_data = (image_data.reshape(1, -1).astype(np.float32) - 127.5) / 127.5

# Load the model
model = Model.load('fashion_mnist.model')

# Predict on the image
confidences = model.predict(image_data)

# Get prediction instead of confidence levels
predictions = model.output_layer_activation.predictions(confidences)

# Get label name from label index
prediction = fashion_mnist_labels[predictions[0]]

print(prediction)

>>>
Trouser
```

A success again! We have now coded in the last feature of our model, which closes the list of the topics that we covered in this book.

Full code:

```
import numpy as np
import nnfs
import os
import cv2
import pickle
import copy

nnfs.init()


# Dense layer
class Layer_Dense:

    # Layer initialization
    def __init__(self, n_inputs, n_neurons,
                 weight_regularizer_l1=0, weight_regularizer_l2=0,
                 bias_regularizer_l1=0, bias_regularizer_l2=0):
        # Initialize weights and biases
        self.weights = 0.01 * np.random.randn(n_inputs, n_neurons)
        self.biases = np.zeros((1, n_neurons))
        # Set regularization strength
        self.weight_regularizer_l1 = weight_regularizer_l1
        self.weight_regularizer_l2 = weight_regularizer_l2
        self.bias_regularizer_l1 = bias_regularizer_l1
        self.bias_regularizer_l2 = bias_regularizer_l2

    # Forward pass
    def forward(self, inputs, training):
        # Remember input values
        self.inputs = inputs
        # Calculate output values from inputs, weights and biases
        self.output = np.dot(inputs, self.weights) + self.biases

    # Backward pass
    def backward(self, dvalues):
        # Gradients on parameters
        self.dweights = np.dot(self.inputs.T, dvalues)
        self.dbiases = np.sum(dvalues, axis=0, keepdims=True)
```

```
        # Gradients on regularization
        # L1 on weights
        if self.weight_regularizer_l1 > 0:
            dL1 = np.ones_like(self.weights)
            dL1[self.weights < 0] = -1
            self.dweights += self.weight_regularizer_l1 * dL1
        # L2 on weights
        if self.weight_regularizer_l2 > 0:
            self.dweights += 2 * self.weight_regularizer_l2 * \
                             self.weights
        # L1 on biases
        if self.bias_regularizer_l1 > 0:
            dL1 = np.ones_like(self.biases)
            dL1[self.biases < 0] = -1
            self.dbiases += self.bias_regularizer_l1 * dL1
        # L2 on biases
        if self.bias_regularizer_l2 > 0:
            self.dbiases += 2 * self.bias_regularizer_l2 * \
                            self.biases

        # Gradient on values
        self.dinputs = np.dot(dvalues, self.weights.T)

    # Retrieve layer parameters
    def get_parameters(self):
        return self.weights, self.biases

    # Set weights and biases in a layer instance
    def set_parameters(self, weights, biases):
        self.weights = weights
        self.biases = biases


# Dropout
class Layer_Dropout:

    # Init
    def __init__(self, rate):
        # Store rate, we invert it as for example for dropout
        # of 0.1 we need success rate of 0.9
        self.rate = 1 - rate

    # Forward pass
    def forward(self, inputs, training):
        # Save input values
        self.inputs = inputs
```

```
        # If not in the training mode - return values
        if not training:
            self.output = inputs.copy()
            return

        # Generate and save scaled mask
        self.binary_mask = np.random.binomial(1, self.rate,
                                  size=inputs.shape) / self.rate
        # Apply mask to output values
        self.output = inputs * self.binary_mask

    # Backward pass
    def backward(self, dvalues):
        # Gradient on values
        self.dinputs = dvalues * self.binary_mask


# Input "layer"
class Layer_Input:

    # Forward pass
    def forward(self, inputs, training):
        self.output = inputs


# ReLU activation
class Activation_ReLU:

    # Forward pass
    def forward(self, inputs, training):
        # Remember input values
        self.inputs = inputs
        # Calculate output values from inputs
        self.output = np.maximum(0, inputs)

    # Backward pass
    def backward(self, dvalues):
        # Since we need to modify original variable,
        # let's make a copy of values first
        self.dinputs = dvalues.copy()

        # Zero gradient where input values were negative
        self.dinputs[self.inputs <= 0] = 0

    # Calculate predictions for outputs
    def predictions(self, outputs):
        return outputs
```

```
# Softmax activation
class Activation_Softmax:

    # Forward pass
    def forward(self, inputs, training):
        # Remember input values
        self.inputs = inputs

        # Get unnormalized probabilities
        exp_values = np.exp(inputs - np.max(inputs, axis=1,
                                            keepdims=True))
        # Normalize them for each sample
        probabilities = exp_values / np.sum(exp_values, axis=1,
                                            keepdims=True)

        self.output = probabilities

    # Backward pass
    def backward(self, dvalues):

        # Create uninitialized array
        self.dinputs = np.empty_like(dvalues)

        # Enumerate outputs and gradients
        for index, (single_output, single_dvalues) in \
                enumerate(zip(self.output, dvalues)):
            # Flatten output array
            single_output = single_output.reshape(-1, 1)
            # Calculate Jacobian matrix of the output
            jacobian_matrix = np.diagflat(single_output) - \
                              np.dot(single_output, single_output.T)
            # Calculate sample-wise gradient
            # and add it to the array of sample gradients
            self.dinputs[index] = np.dot(jacobian_matrix,
                                         single_dvalues)

    # Calculate predictions for outputs
    def predictions(self, outputs):
        return np.argmax(outputs, axis=1)
```

```python
# Sigmoid activation
class Activation_Sigmoid:

    # Forward pass
    def forward(self, inputs, training):
        # Save input and calculate/save output
        # of the sigmoid function
        self.inputs = inputs
        self.output = 1 / (1 + np.exp(-inputs))

    # Backward pass
    def backward(self, dvalues):
        # Derivative - calculates from output of the sigmoid function
        self.dinputs = dvalues * (1 - self.output) * self.output

    # Calculate predictions for outputs
    def predictions(self, outputs):
        return (outputs > 0.5) * 1


# Linear activation
class Activation_Linear:

    # Forward pass
    def forward(self, inputs, training):
        # Just remember values
        self.inputs = inputs
        self.output = inputs

    # Backward pass
    def backward(self, dvalues):
        # derivative is 1, 1 * dvalues = dvalues - the chain rule
        self.dinputs = dvalues.copy()

    # Calculate predictions for outputs
    def predictions(self, outputs):
        return outputs
```

```python
# SGD optimizer
class Optimizer_SGD:

    # Initialize optimizer - set settings,
    # learning rate of 1. is default for this optimizer
    def __init__(self, learning_rate=1., decay=0., momentum=0.):
        self.learning_rate = learning_rate
        self.current_learning_rate = learning_rate
        self.decay = decay
        self.iterations = 0
        self.momentum = momentum

    # Call once before any parameter updates
    def pre_update_params(self):
        if self.decay:
            self.current_learning_rate = self.learning_rate * \
                (1. / (1. + self.decay * self.iterations))

    # Update parameters
    def update_params(self, layer):

        # If we use momentum
        if self.momentum:

            # If layer does not contain momentum arrays, create them
            # filled with zeros
            if not hasattr(layer, 'weight_momentums'):
                layer.weight_momentums = np.zeros_like(layer.weights)
                # If there is no momentum array for weights
                # The array doesn't exist for biases yet either.
                layer.bias_momentums = np.zeros_like(layer.biases)

            # Build weight updates with momentum - take previous
            # updates multiplied by retain factor and update with
            # current gradients
            weight_updates = \
                self.momentum * layer.weight_momentums - \
                self.current_learning_rate * layer.dweights
            layer.weight_momentums = weight_updates

            # Build bias updates
            bias_updates = \
                self.momentum * layer.bias_momentums - \
                self.current_learning_rate * layer.dbiases
            layer.bias_momentums = bias_updates
```

```
        # Vanilla SGD updates (as before momentum update)
        else:
            weight_updates = -self.current_learning_rate * \
                             layer.dweights
            bias_updates = -self.current_learning_rate * \
                           layer.dbiases

        # Update weights and biases using either
        # vanilla or momentum updates
        layer.weights += weight_updates
        layer.biases += bias_updates

    # Call once after any parameter updates
    def post_update_params(self):
        self.iterations += 1


# Adagrad optimizer
class Optimizer_Adagrad:

    # Initialize optimizer - set settings
    def __init__(self, learning_rate=1., decay=0., epsilon=1e-7):
        self.learning_rate = learning_rate
        self.current_learning_rate = learning_rate
        self.decay = decay
        self.iterations = 0
        self.epsilon = epsilon

    # Call once before any parameter updates
    def pre_update_params(self):
        if self.decay:
            self.current_learning_rate = self.learning_rate * \
                (1. / (1. + self.decay * self.iterations))

    # Update parameters
    def update_params(self, layer):

        # If layer does not contain cache arrays,
        # create them filled with zeros
        if not hasattr(layer, 'weight_cache'):
            layer.weight_cache = np.zeros_like(layer.weights)
            layer.bias_cache = np.zeros_like(layer.biases)

        # Update cache with squared current gradients
        layer.weight_cache += layer.dweights**2
        layer.bias_cache += layer.dbiases**2
```

```
        # Vanilla SGD parameter update + normalization
        # with square rooted cache
        layer.weights += -self.current_learning_rate * \
                         layer.dweights / \
                         (np.sqrt(layer.weight_cache) + self.epsilon)
        layer.biases += -self.current_learning_rate * \
                        layer.dbiases / \
                        (np.sqrt(layer.bias_cache) + self.epsilon)

    # Call once after any parameter updates
    def post_update_params(self):
        self.iterations += 1


# RMSprop optimizer
class Optimizer_RMSprop:

    # Initialize optimizer - set settings
    def __init__(self, learning_rate=0.001, decay=0., epsilon=1e-7,
                 rho=0.9):
        self.learning_rate = learning_rate
        self.current_learning_rate = learning_rate
        self.decay = decay
        self.iterations = 0
        self.epsilon = epsilon
        self.rho = rho

    # Call once before any parameter updates
    def pre_update_params(self):
        if self.decay:
            self.current_learning_rate = self.learning_rate * \
                (1. / (1. + self.decay * self.iterations))

    # Update parameters
    def update_params(self, layer):

        # If layer does not contain cache arrays,
        # create them filled with zeros
        if not hasattr(layer, 'weight_cache'):
            layer.weight_cache = np.zeros_like(layer.weights)
            layer.bias_cache = np.zeros_like(layer.biases)

        # Update cache with squared current gradients
        layer.weight_cache = self.rho * layer.weight_cache + \
            (1 - self.rho) * layer.dweights**2
        layer.bias_cache = self.rho * layer.bias_cache + \
            (1 - self.rho) * layer.dbiases**2
```

```
        # Vanilla SGD parameter update + normalization
        # with square rooted cache
        layer.weights += -self.current_learning_rate * \
                         layer.dweights / \
                         (np.sqrt(layer.weight_cache) + self.epsilon)
        layer.biases += -self.current_learning_rate * \
                        layer.dbiases / \
                        (np.sqrt(layer.bias_cache) + self.epsilon)

    # Call once after any parameter updates
    def post_update_params(self):
        self.iterations += 1


# Adam optimizer
class Optimizer_Adam:

    # Initialize optimizer - set settings
    def __init__(self, learning_rate=0.001, decay=0., epsilon=1e-7,
                 beta_1=0.9, beta_2=0.999):
        self.learning_rate = learning_rate
        self.current_learning_rate = learning_rate
        self.decay = decay
        self.iterations = 0
        self.epsilon = epsilon
        self.beta_1 = beta_1
        self.beta_2 = beta_2

    # Call once before any parameter updates
    def pre_update_params(self):
        if self.decay:
            self.current_learning_rate = self.learning_rate * \
                (1. / (1. + self.decay * self.iterations))

    # Update parameters
    def update_params(self, layer):

        # If layer does not contain cache arrays,
        # create them filled with zeros
        if not hasattr(layer, 'weight_cache'):
            layer.weight_momentums = np.zeros_like(layer.weights)
            layer.weight_cache = np.zeros_like(layer.weights)
            layer.bias_momentums = np.zeros_like(layer.biases)
            layer.bias_cache = np.zeros_like(layer.biases)
```

```
        # Update momentum  with current gradients
        layer.weight_momentums = self.beta_1 * \
                                 layer.weight_momentums + \
                                 (1 - self.beta_1) * layer.dweights
        layer.bias_momentums = self.beta_1 * \
                               layer.bias_momentums + \
                               (1 - self.beta_1) * layer.dbiases
        # Get corrected momentum
        # self.iteration is 0 at first pass
        # and we need to start with 1 here
        weight_momentums_corrected = layer.weight_momentums / \
            (1 - self.beta_1 ** (self.iterations + 1))
        bias_momentums_corrected = layer.bias_momentums / \
            (1 - self.beta_1 ** (self.iterations + 1))
        # Update cache with squared current gradients
        layer.weight_cache = self.beta_2 * layer.weight_cache + \
            (1 - self.beta_2) * layer.dweights**2
        layer.bias_cache = self.beta_2 * layer.bias_cache + \
            (1 - self.beta_2) * layer.dbiases**2
        # Get corrected cache
        weight_cache_corrected = layer.weight_cache / \
            (1 - self.beta_2 ** (self.iterations + 1))
        bias_cache_corrected = layer.bias_cache / \
            (1 - self.beta_2 ** (self.iterations + 1))

        # Vanilla SGD parameter update + normalization
        # with square rooted cache
        layer.weights += -self.current_learning_rate * \
                         weight_momentums_corrected / \
                         (np.sqrt(weight_cache_corrected) +
                             self.epsilon)
        layer.biases += -self.current_learning_rate * \
                         bias_momentums_corrected / \
                         (np.sqrt(bias_cache_corrected) +
                             self.epsilon)

    # Call once after any parameter updates
    def post_update_params(self):
        self.iterations += 1
```

```
# Common loss class
class Loss:

    # Regularization loss calculation
    def regularization_loss(self):

        # 0 by default
        regularization_loss = 0

        # Calculate regularization loss
        # iterate all trainable layers
        for layer in self.trainable_layers:

            # L1 regularization - weights
            # calculate only when factor greater than 0
            if layer.weight_regularizer_l1 > 0:
                regularization_loss += layer.weight_regularizer_l1 * \
                                       np.sum(np.abs(layer.weights))

            # L2 regularization - weights
            if layer.weight_regularizer_l2 > 0:
                regularization_loss += layer.weight_regularizer_l2 * \
                                       np.sum(layer.weights * \
                                              layer.weights)

            # L1 regularization - biases
            # calculate only when factor greater than 0
            if layer.bias_regularizer_l1 > 0:
                regularization_loss += layer.bias_regularizer_l1 * \
                                       np.sum(np.abs(layer.biases))

            # L2 regularization - biases
            if layer.bias_regularizer_l2 > 0:
                regularization_loss += layer.bias_regularizer_l2 * \
                                       np.sum(layer.biases * \
                                              layer.biases)

        return regularization_loss

    # Set/remember trainable layers
    def remember_trainable_layers(self, trainable_layers):
        self.trainable_layers = trainable_layers
```

```
    # Calculates the data and regularization losses
    # given model output and ground truth values
    def calculate(self, output, y, *, include_regularization=False):

        # Calculate sample losses
        sample_losses = self.forward(output, y)

        # Calculate mean loss
        data_loss = np.mean(sample_losses)

        # Add accumulated sum of losses and sample count
        self.accumulated_sum += np.sum(sample_losses)
        self.accumulated_count += len(sample_losses)

        # If just data loss - return it
        if not include_regularization:
            return data_loss

        # Return the data and regularization losses
        return data_loss, self.regularization_loss()

    # Calculates accumulated loss
    def calculate_accumulated(self, *, include_regularization=False):

        # Calculate mean loss
        data_loss = self.accumulated_sum / self.accumulated_count

        # If just data loss - return it
        if not include_regularization:
            return data_loss

        # Return the data and regularization losses
        return data_loss, self.regularization_loss()

    # Reset variables for accumulated loss
    def new_pass(self):
        self.accumulated_sum = 0
        self.accumulated_count = 0


# Cross-entropy loss
class Loss_CategoricalCrossentropy(Loss):

    # Forward pass
    def forward(self, y_pred, y_true):

        # Number of samples in a batch
        samples = len(y_pred)
```

```
        # Clip data to prevent division by 0
        # Clip both sides to not drag mean towards any value
        y_pred_clipped = np.clip(y_pred, 1e-7, 1 - 1e-7)

        # Probabilities for target values -
        # only if categorical labels
        if len(y_true.shape) == 1:
            correct_confidences = y_pred_clipped[
                range(samples),
                y_true
            ]

        # Mask values - only for one-hot encoded labels
        elif len(y_true.shape) == 2:
            correct_confidences = np.sum(
                y_pred_clipped * y_true,
                axis=1
            )

        # Losses
        negative_log_likelihoods = -np.log(correct_confidences)
        return negative_log_likelihoods

    # Backward pass
    def backward(self, dvalues, y_true):

        # Number of samples
        samples = len(dvalues)
        # Number of labels in every sample
        # We'll use the first sample to count them
        labels = len(dvalues[0])

        # If labels are sparse, turn them into one-hot vector
        if len(y_true.shape) == 1:
            y_true = np.eye(labels)[y_true]

        # Calculate gradient
        self.dinputs = -y_true / dvalues
        # Normalize gradient
        self.dinputs = self.dinputs / samples
```

```
# Softmax classifier - combined Softmax activation
# and cross-entropy loss for faster backward step
class Activation_Softmax_Loss_CategoricalCrossentropy():

    # Backward pass
    def backward(self, dvalues, y_true):

        # Number of samples
        samples = len(dvalues)

        # If labels are one-hot encoded,
        # turn them into discrete values
        if len(y_true.shape) == 2:
            y_true = np.argmax(y_true, axis=1)

        # Copy so we can safely modify
        self.dinputs = dvalues.copy()
        # Calculate gradient
        self.dinputs[range(samples), y_true] -= 1
        # Normalize gradient
        self.dinputs = self.dinputs / samples


# Binary cross-entropy loss
class Loss_BinaryCrossentropy(Loss):

    # Forward pass
    def forward(self, y_pred, y_true):

        # Clip data to prevent division by 0
        # Clip both sides to not drag mean towards any value
        y_pred_clipped = np.clip(y_pred, 1e-7, 1 - 1e-7)

        # Calculate sample-wise loss
        sample_losses = -(y_true * np.log(y_pred_clipped) +
                          (1 - y_true) * np.log(1 - y_pred_clipped))
        sample_losses = np.mean(sample_losses, axis=-1)

        # Return losses
        return sample_losses

    # Backward pass
    def backward(self, dvalues, y_true):

        # Number of samples
        samples = len(dvalues)
        # Number of outputs in every sample
        # We'll use the first sample to count them
        outputs = len(dvalues[0])
```

```
        # Clip data to prevent division by 0
        # Clip both sides to not drag mean towards any value
        clipped_dvalues = np.clip(dvalues, 1e-7, 1 - 1e-7)

        # Calculate gradient
        self.dinputs = -(y_true / clipped_dvalues -
                         (1 - y_true) / (1 - clipped_dvalues)) / outputs
        # Normalize gradient
        self.dinputs = self.dinputs / samples


# Mean Squared Error loss
class Loss_MeanSquaredError(Loss):  # L2 loss

    # Forward pass
    def forward(self, y_pred, y_true):

        # Calculate loss
        sample_losses = np.mean((y_true - y_pred)**2, axis=-1)

        # Return losses
        return sample_losses

    # Backward pass
    def backward(self, dvalues, y_true):

        # Number of samples
        samples = len(dvalues)
        # Number of outputs in every sample
        # We'll use the first sample to count them
        outputs = len(dvalues[0])

        # Gradient on values
        self.dinputs = -2 * (y_true - dvalues) / outputs
        # Normalize gradient
        self.dinputs = self.dinputs / samples


# Mean Absolute Error loss
class Loss_MeanAbsoluteError(Loss):  # L1 loss

    def forward(self, y_pred, y_true):

        # Calculate loss
        sample_losses = np.mean(np.abs(y_true - y_pred), axis=-1)

        # Return losses
        return sample_losses
```

```
    # Backward pass
    def backward(self, dvalues, y_true):

        # Number of samples
        samples = len(dvalues)
        # Number of outputs in every sample
        # We'll use the first sample to count them
        outputs = len(dvalues[0])

        # Calculate gradient
        self.dinputs = np.sign(y_true - dvalues) / outputs
        # Normalize gradient
        self.dinputs = self.dinputs / samples


# Common accuracy class
class Accuracy:

    # Calculates an accuracy
    # given predictions and ground truth values
    def calculate(self, predictions, y):

        # Get comparison results
        comparisons = self.compare(predictions, y)

        # Calculate an accuracy
        accuracy = np.mean(comparisons)

        # Add accumulated sum of matching values and sample count
        self.accumulated_sum += np.sum(comparisons)
        self.accumulated_count += len(comparisons)

        # Return accuracy
        return accuracy

    # Calculates accumulated accuracy
    def calculate_accumulated(self):

        # Calculate an accuracy
        accuracy = self.accumulated_sum / self.accumulated_count

        # Return the data and regularization losses
        return accuracy

    # Reset variables for accumulated accuracy
    def new_pass(self):
        self.accumulated_sum = 0
        self.accumulated_count = 0
```

```
# Accuracy calculation for classification model
class Accuracy_Categorical(Accuracy):

    def __init__(self, *, binary=False):
        # Binary mode?
        self.binary = binary

    # No initialization is needed
    def init(self, y):
        pass

    # Compares predictions to the ground truth values
    def compare(self, predictions, y):
        if not self.binary and len(y.shape) == 2:
            y = np.argmax(y, axis=1)
        return predictions == y


# Accuracy calculation for regression model
class Accuracy_Regression(Accuracy):

    def __init__(self):
        # Create precision property
        self.precision = None

    # Calculates precision value
    # based on passed-in ground truth values
    def init(self, y, reinit=False):
        if self.precision is None or reinit:
            self.precision = np.std(y) / 250

    # Compares predictions to the ground truth values
    def compare(self, predictions, y):
        return np.absolute(predictions - y) < self.precision


# Model class
class Model:

    def __init__(self):
        # Create a list of network objects
        self.layers = []
        # Softmax classifier's output object
        self.softmax_classifier_output = None

    # Add objects to the model
    def add(self, layer):
        self.layers.append(layer)
```

```
    # Set loss, optimizer and accuracy
    def set(self, *, loss=None, optimizer=None, accuracy=None):

        if loss is not None:
            self.loss = loss

        if optimizer is not None:
            self.optimizer = optimizer

        if accuracy is not None:
            self.accuracy = accuracy

    # Finalize the model
    def finalize(self):

        # Create and set the input layer
        self.input_layer = Layer_Input()

        # Count all the objects
        layer_count = len(self.layers)

        # Initialize a list containing trainable layers:
        self.trainable_layers = []

        # Iterate the objects
        for i in range(layer_count):

            # If it's the first layer,
            # the previous layer object is the input layer
            if i == 0:
                self.layers[i].prev = self.input_layer
                self.layers[i].next = self.layers[i+1]

            # All layers except for the first and the last
            elif i < layer_count - 1:
                self.layers[i].prev = self.layers[i-1]
                self.layers[i].next = self.layers[i+1]

            # The last layer - the next object is the loss
            # Also let's save aside the reference to the last object
            # whose output is the model's output
            else:
                self.layers[i].prev = self.layers[i-1]
                self.layers[i].next = self.loss
                self.output_layer_activation = self.layers[i]
```

```
        # If layer contains an attribute called "weights",
        # it's a trainable layer -
        # add it to the list of trainable layers
        # We don't need to check for biases -
        # checking for weights is enough
        if hasattr(self.layers[i], 'weights'):
            self.trainable_layers.append(self.layers[i])

    # Update loss object with trainable layers
    if self.loss is not None:
        self.loss.remember_trainable_layers(
            self.trainable_layers
        )

    # If output activation is Softmax and
    # loss function is Categorical Cross-Entropy
    # create an object of combined activation
    # and loss function containing
    # faster gradient calculation
    if isinstance(self.layers[-1], Activation_Softmax) and \
       isinstance(self.loss, Loss_CategoricalCrossentropy):
        # Create an object of combined activation
        # and loss functions
        self.softmax_classifier_output = \
            Activation_Softmax_Loss_CategoricalCrossentropy()

# Train the model
def train(self, X, y, *, epochs=1, batch_size=None,
          print_every=1, validation_data=None):

    # Initialize accuracy object
    self.accuracy.init(y)

    # Default value if batch size is not being set
    train_steps = 1

    # Calculate number of steps
    if batch_size is not None:
        train_steps = len(X) // batch_size
        # Dividing rounds down. If there are some remaining
        # data but not a full batch, this won't include it
        # Add `1` to include this not full batch
        if train_steps * batch_size < len(X):
            train_steps += 1
```

```
        # Main training loop
        for epoch in range(1, epochs+1):

            # Print epoch number
            print(f'epoch: {epoch}')

            # Reset accumulated values in loss and accuracy objects
            self.loss.new_pass()
            self.accuracy.new_pass()

            # Iterate over steps
            for step in range(train_steps):

                # If batch size is not set -
                # train using one step and full dataset
                if batch_size is None:
                    batch_X = X
                    batch_y = y

                # Otherwise slice a batch
                else:
                    batch_X = X[step*batch_size:(step+1)*batch_size]
                    batch_y = y[step*batch_size:(step+1)*batch_size]

                # Perform the forward pass
                output = self.forward(batch_X, training=True)

                # Calculate loss
                data_loss, regularization_loss = \
                    self.loss.calculate(output, batch_y,
                                        include_regularization=True)
                loss = data_loss + regularization_loss

                # Get predictions and calculate an accuracy
                predictions = self.output_layer_activation.predictions(
                                  output)
                accuracy = self.accuracy.calculate(predictions,
                                                   batch_y)

                # Perform backward pass
                self.backward(output, batch_y)

                # Optimize (update parameters)
                self.optimizer.pre_update_params()
                for layer in self.trainable_layers:
                    self.optimizer.update_params(layer)
                self.optimizer.post_update_params()
```

```
            # Print a summary
            if not step % print_every or step == train_steps - 1:
                print(f'step: {step}, ' +
                      f'acc: {accuracy:.3f}, ' +
                      f'loss: {loss:.3f} (' +
                      f'data_loss: {data_loss:.3f}, ' +
                      f'reg_loss: {regularization_loss:.3f}), ' +
                      f'lr: {self.optimizer.current_learning_rate}')

        # Get and print epoch loss and accuracy
        epoch_data_loss, epoch_regularization_loss = \
            self.loss.calculate_accumulated(
                include_regularization=True)
        epoch_loss = epoch_data_loss + epoch_regularization_loss
        epoch_accuracy = self.accuracy.calculate_accumulated()

        print(f'training, ' +
              f'acc: {epoch_accuracy:.3f}, ' +
              f'loss: {epoch_loss:.3f} (' +
              f'data_loss: {epoch_data_loss:.3f}, ' +
              f'reg_loss: {epoch_regularization_loss:.3f}), ' +
              f'lr: {self.optimizer.current_learning_rate}')

        # If there is the validation data
        if validation_data is not None:

            # Evaluate the model:
            self.evaluate(*validation_data,
                          batch_size=batch_size)

    # Evaluates the model using passed-in dataset
    def evaluate(self, X_val, y_val, *, batch_size=None):

        # Default value if batch size is not being set
        validation_steps = 1

        # Calculate number of steps
        if batch_size is not None:
            validation_steps = len(X_val) // batch_size
            # Dividing rounds down. If there are some remaining
            # data but not a full batch, this won't include it
            # Add `1` to include this not full batch
            if validation_steps * batch_size < len(X_val):
                validation_steps += 1

        # Reset accumulated values in loss
        # and accuracy objects
        self.loss.new_pass()
        self.accuracy.new_pass()
```

```
        # Iterate over steps
        for step in range(validation_steps):

            # If batch size is not set -
            # train using one step and full dataset
            if batch_size is None:
                batch_X = X_val
                batch_y = y_val

            # Otherwise slice a batch
            else:
                batch_X = X_val[
                    step*batch_size:(step+1)*batch_size
                ]
                batch_y = y_val[
                    step*batch_size:(step+1)*batch_size
                ]

            # Perform the forward pass
            output = self.forward(batch_X, training=False)

            # Calculate the loss
            self.loss.calculate(output, batch_y)

            # Get predictions and calculate an accuracy
            predictions = self.output_layer_activation.predictions(
                              output)
            self.accuracy.calculate(predictions, batch_y)

        # Get and print validation loss and accuracy
        validation_loss = self.loss.calculate_accumulated()
        validation_accuracy = self.accuracy.calculate_accumulated()

        # Print a summary
        print(f'validation, ' +
              f'acc: {validation_accuracy:.3f}, ' +
              f'loss: {validation_loss:.3f}')

    # Predicts on the samples
    def predict(self, X, *, batch_size=None):

        # Default value if batch size is not being set
        prediction_steps = 1

        # Calculate number of steps
        if batch_size is not None:
            prediction_steps = len(X) // batch_size
```

```
        # Dividing rounds down. If there are some remaining
        # data but not a full batch, this won't include it
        # Add `1` to include this not full batch
        if prediction_steps * batch_size < len(X):
            prediction_steps += 1

    # Model outputs
    output = []

    # Iterate over steps
    for step in range(prediction_steps):

        # If batch size is not set -
        # train using one step and full dataset
        if batch_size is None:
            batch_X = X

        # Otherwise slice a batch
        else:
            batch_X = X[step*batch_size:(step+1)*batch_size]

        # Perform the forward pass
        batch_output = self.forward(batch_X, training=False)

        # Append batch prediction to the list of predictions
        output.append(batch_output)

    # Stack and return results
    return np.vstack(output)

# Performs forward pass
def forward(self, X, training):

    # Call forward method on the input layer
    # this will set the output property that
    # the first layer in "prev" object is expecting
    self.input_layer.forward(X, training)

    # Call forward method of every object in a chain
    # Pass output of the previous object as a parameter
    for layer in self.layers:
        layer.forward(layer.prev.output, training)

    # "layer" is now the last object from the list,
    # return its output
    return layer.output
```

```
    # Performs backward pass
    def backward(self, output, y):

        # If softmax classifier
        if self.softmax_classifier_output is not None:
            # First call backward method
            # on the combined activation/loss
            # this will set dinputs property
            self.softmax_classifier_output.backward(output, y)

            # Since we'll not call backward method of the last layer
            # which is Softmax activation
            # as we used combined activation/loss
            # object, let's set dinputs in this object
            self.layers[-1].dinputs = \
                self.softmax_classifier_output.dinputs

            # Call backward method going through
            # all the objects but last
            # in reversed order passing dinputs as a parameter
            for layer in reversed(self.layers[:-1]):
                layer.backward(layer.next.dinputs)

            return

        # First call backward method on the loss
        # this will set dinputs property that the last
        # layer will try to access shortly
        self.loss.backward(output, y)

        # Call backward method going through all the objects
        # in reversed order passing dinputs as a parameter
        for layer in reversed(self.layers):
            layer.backward(layer.next.dinputs)

    # Retrieves and returns parameters of trainable layers
    def get_parameters(self):

        # Create a list for parameters
        parameters = []

        # Iterable trainable layers and get their parameters
        for layer in self.trainable_layers:
            parameters.append(layer.get_parameters())

        # Return a list
        return parameters
```

```
    # Updates the model with new parameters
    def set_parameters(self, parameters):

        # Iterate over the parameters and layers
        # and update each layers with each set of the parameters
        for parameter_set, layer in zip(parameters,
                                        self.trainable_layers):
            layer.set_parameters(*parameter_set)

    # Saves the parameters to a file
    def save_parameters(self, path):

        # Open a file in the binary-write mode
        # and save parameters into it
        with open(path, 'wb') as f:
            pickle.dump(self.get_parameters(), f)

    # Loads the weights and updates a model instance with them
    def load_parameters(self, path):

        # Open file in the binary-read mode,
        # load weights and update trainable layers
        with open(path, 'rb') as f:
            self.set_parameters(pickle.load(f))

    # Saves the model
    def save(self, path):

        # Make a deep copy of current model instance
        model = copy.deepcopy(self)

        # Reset accumulated values in loss and accuracy objects
        model.loss.new_pass()
        model.accuracy.new_pass()

        # Remove data from the input layer
        # and gradients from the loss object
        model.input_layer.__dict__.pop('output', None)
        model.loss.__dict__.pop('dinputs', None)

        # For each layer remove inputs, output and dinputs properties
        for layer in model.layers:
            for property in ['inputs', 'output', 'dinputs',
                             'dweights', 'dbiases']:
                layer.__dict__.pop(property, None)

        # Open a file in the binary-write mode and save the model
        with open(path, 'wb') as f:
            pickle.dump(model, f)
```

```
    # Loads and returns a model
    @staticmethod
    def load(path):

        # Open file in the binary-read mode, load a model
        with open(path, 'rb') as f:
            model = pickle.load(f)

        # Return a model
        return model


# Loads a MNIST dataset
def load_mnist_dataset(dataset, path):

    # Scan all the directories and create a list of labels
    labels = os.listdir(os.path.join(path, dataset))

    # Create lists for samples and labels
    X = []
    y = []

    # For each label folder
    for label in labels:
        # And for each image in given folder
        for file in os.listdir(os.path.join(path, dataset, label)):
            # Read the image
            image = cv2.imread(
                        os.path.join(path, dataset, label, file),
                        cv2.IMREAD_UNCHANGED)

            # And append it and a label to the lists
            X.append(image)
            y.append(label)

    # Convert the data to proper numpy arrays and return
    return np.array(X), np.array(y).astype('uint8')


# MNIST dataset (train + test)
def create_data_mnist(path):

    # Load both sets separately
    X, y = load_mnist_dataset('train', path)
    X_test, y_test = load_mnist_dataset('test', path)

    # And return all the data
    return X, y, X_test, y_test
```

```
# Label index to label name relation
fashion_mnist_labels = {
    0: 'T-shirt/top',
    1: 'Trouser',
    2: 'Pullover',
    3: 'Dress',
    4: 'Coat',
    5: 'Sandal',
    6: 'Shirt',
    7: 'Sneaker',
    8: 'Bag',
    9: 'Ankle boot'
}

# Read an image
image_data = cv2.imread('pants.png', cv2.IMREAD_GRAYSCALE)

# Resize to the same size as Fashion MNIST images
image_data = cv2.resize(image_data, (28, 28))

# Invert image colors
image_data = 255 - image_data

# Reshape and scale pixel data
image_data = (image_data.reshape(1, -1).astype(np.float32) - 127.5) / 127.5

# Load the model
model = Model.load('fashion_mnist.model')

# Predict on the image
confidences = model.predict(image_data)

# Get prediction instead of confidence levels
predictions = model.output_layer_activation.predictions(confidences)

# Get label name from label index
prediction = fashion_mnist_labels[predictions[0]]

print(prediction)
```

Supplementary Material: https://nnfs.io/ch22
Chapter code, further resources, and errata for this chapter.

Chapter 23

Closing

By now, you should have a full understanding of how data passes forward through neural networks, how error and accuracy are measured, how to perform backpropagation to acquire a gradient, how network parameters are optimized based on the gradient, and how to use a trained model to make predictions. As we close this book, the reality is that you are just getting started, as is this field. There are various types of layers besides the dense layer used here, mainly **recurrent** and **convolutional** layers, and many more use-cases for neural networks than what we covered here (classification and regression). The field of deep learning is constantly evolving and building on top of the foundations introduced in this book. Our hope is that we've equipped you with the ability to move with these advances and to best leverage neural networks of today to solve problems.

Whether you're using a neural network to classify cats vs dogs, to generate audio, or for a chatbot, the challenge is still the same — *how do you best structure your problem so a neural network can fit and generalize it?*

Inevitably, when things don't work the way you hoped — *how do you interpret the network's failure?*

You're now equipped with the knowledge required to get a complete picture of what's happening (or not happening) within the network. What's happening to the input as it goes through layers? Is gradient exploding? You can feed a sample through and see exactly how values throughout the network are calculated. At this point as well, you've seen us break down the calculation of partial derivatives for four loss functions (five, counting softmax and categorical cross-entropy combined). From here, if you were so inclined, you could even attempt to create your own custom loss function. Where to go from here is up to you.

We'd now like to enumerate and acknowledge those who have helped to make this book possible. First, we'd like to thank the teams behind the following packages which have helped ordinary people like us to instantly leverage an incredible amount of knowledge and intellect with respect to deep learning with a simple *pip install*: NumPy, TensorFlow, Keras, Pytorch, Theano, Matplotlib, and OpenCV.

More specifically, we'd like to thank Grant Sanderson (*3Blue1Brown* on YouTube) for the *Manim* package and for pushing the limits of mathematics-based, high quality, visualizations, and sharing this with the world both through his own teaching material as well as through sharing the *Manim* package. We extended this package for use with almost all of our animations and figures.

Finally, we'd like to thank the contributors to this book. Both **Spencer Churchill** and **Thomas Halvorson** have been incredibly helpful as editors of their own volition and free time. With countless improvements and suggestions, Spencer and Thomas stood out with firm grasps of both the English language and the subject matter contained within this book. A huge thank you to you both from the authors.

There have been many other people who have made contributions to the text as well via the Google Doc, which we'll list here:
Kevin C, Wayne Kiely, Chad DeRosier, Christian Liz-Fonts, Steve Martin, Wayne Kiely, David Arkless, William Mailhot, Tom H, Andrew Doud, Larry Cawley, Yuval Omer, Bryan Bischof, Rasmus Larsson, Armando Bezies, Michael Pegnam, Maarten Roos, Adam Walter, Christian Stankovic, Leon Shams-Schaal, Emilie Parent, Andy Brice, Aaron Passchier, Payson Chadrow, Andrew Doud, Andy Passmore, Jyotinder Singh, Coltan Gowan, Steffan Johansson, Dimitris Mermiklis, Philipp Tölle, Chris Gooding, Ernesto Ramos, Pramit Shah, Wayne Poulson, Sai Anish Malla, Mike Gropp, RD Lagos, Dr Butensky Moshe, Nicholas Obert, Adrian Bool, Jyotinder Singh, Lucas Parzianello, Leene 04, Rudy Depena, Jonathan Lindbloom, Ahmed Hassan, Sebastian Wette, Matheus Ferreira, Dimitris Mermiklis, Lukas Heine, Harshith Thota, Adam Strike, Andrew Doud, Kevin Byrne, Luca Passani, Kyler Crank, Chris Hahn, Büşra Emir , Ethan Zemelman, Jozef Alexovic, Tayler Porter, Roberto Tomás, Gavin Reddy, CurryWaffel, Daniel Alves, Mees Broer, David Canaday, Austin Johansen, Jan Kuparinen, Daniel Tsang, Nate Krasner, Everman Robert, Rick Roach, Addison Weatherhead, Christian Liz-Fonts, Christopher Varnon, Michael Peres, Eytan Ohana, Wes Simpson, Tommy Katzenellenbogen, João Pedro Ferreira,

Patrick le, Yannick Poulin-giroux, Alex Duenas, Omota Paul-Daniel, Akira Kurusu, Andrew Yatzkan, Jesús Peña Martín, Ben Schneider, Steve, Stephen O'Brien, Steve, Diego López González, Andrew Doud, Juan Felix, Jeff Echt, Dimitris Mermiklis, Adrian Bool, Ulrich Storkenmaier-Fabinyi, John Smith, Allan McElroy, Leo Liang, and *Harrison McCarty, Matt Williams, Magnus Imber, Ag, Andy Junghyun Kim, Alec Mather, Marijn Morssink, Stijn Schatteman, Yang Liu, Andrew Krcatovich*

Supplementary Material: https://nnfs.io/ch23
Chapter code, further resources, and errata for this chapter.

Index

Symbols
1/t decaying 274

A
accuracy
  categorical 129
  regression 432
Activation Function (concept) 72, 77, 321, 328, 332, 335, 361
Activation Functions
  linear 74, 425
  rectified linear (ReLU) 76, 81, 85, 95
  sigmoid 75, 389
  softmax 98, 220, 230
  step 73
AdaGrad 293
adaptive gradient 293
Adaptive Momentum 304
Adaptive Momentum (Adam) 304
adjustable parameters 23
analytical derivative 154
analytical differentiation 154
array 34, 35

B
backpropagation 180, 196
batch 250
batch (data) 44, 54
batches 549
batch gradient descent 250
Batch Gradient Descent 250
batch size 549
Bernoulli distribution 362
biases 15, 57
binary classification 19
binary cross-entropy (loss) 396
binary logistic regression 388

C
cache (RMSProp) 298
categorical cross-entropy (loss) 112, 114, 215, 230
Chain Rule 174, 196
checkpoint 612
classification 19
classification (model) 12
classifications 12
co-adoption 361
confidence scores 98
cost function 111
cross product 38
cross-validation 329

D
data
  out-of-sample 321
  testing 321
  training 62, 332
  validation 328
data augmentation 333
data balancing 537
data flattening 544
data normalization 536
data preprocessing 332, 543
data scaling 543
data shuffling 545, 546
dead neurons 67
decay rate 274
deep neural network 59
dense (layer) 14, 66
derivative 139
  analytical 154
  numerical 146
differentiation
  analytical 154
  numerical 146
dot product 38, 42, 55
dropout 361
dropout layer 361

E
epsilon (AdaGrad) 293
exponential decaying 274

F
Fashion MNIST 532
feature set 44
forward pass 66
fully connected 32, 66

G
Gaussian distribution 67
generalization 23, 45
global minimum 258
gradient 167, 173
Gradient Descent 250
gradient explosion 267
ground-truth 12

H
hidden layer 59
hidden layers 59
homologous (array) 35

I
inference 479, 617
inputs 54
instantaneous slope 145

L
L1 regularization 336
L2 regularization 336
labels 12
lambda (regularization) 336
layer (neurons) 30, 41, 54, 59
learning rate decay 274
learning rate (LR) 256, 257
Leibniz's notation 155
linear (activation) 74
list (programming) 34
local minimum 259
log 115
logarithm 115
log (logarithm) 115
loss function 111
Loss Functions
  binary cross-entropy 396, 398
  categorical cross-entropy 112, 215, 230
  mean absolute error loss (MAE) 429
  mean squared error loss (MSE) 426

M
MAE (mean absolute error) 429
matrix 34, 35
matrix product 47, 55
Mean Absolute Error 429
mean absolute error (loss) 429
mean squared error (loss) 426
mini-batch 250
Mini-batch Gradient Descent 250
Mini-batch Gradient Descent, MBGD 250
MNIST 532
model evaluation 594
model inference 479
Model Types
  binary logistic regression 388
  classiffication 12
  regression 423
MSE (mean squared error) 426

N
natural logarithm 115
neural network 13
neuron 26, 40
noise 361
numerical derivative 146
numerical differentiation 146
NumPy 25

O
observation 44
one-hot (vector) 113
optimization 131
Optimizers 249
  AdaGrad 293
  Adam 304
  RMSProp 298
  Stochastic Gradient Descent (SGD) 250
  Stochastic Gradient Descent with Momentum 283
out-of-sample data 323
overfitting 23, 322

P
parameters 23
partial derivative 167
prediction 617
preprocess 19
preprocessing 332, 333, 543

R
random initialization 66
rectified linear (activation) 76, 81, 85, 95
regression (model) 423
Regularization 335
  L1 regularization 336
  L2 regularization 336
  lambda 336
ReLU 76, 81, 85, 95
rho (RMSProp) 298
RMSProp 298
Root Mean Square Propagation (RMSProp) 298

S
sample 44
sigmoid (activation) 75, 389
slope 142
softmax (activation) 98, 220, 230
squared error 112
step 549
step activation 73
step function 18
step function (activation) 73
Stochastic Gradient Descent (SGD) 250
Stochastic Gradient Descent with Momentum 283
supervised machine learning 12

T
tangent line 145
targets 12
tensor 34, 37
testing data 321, 323
training 563
training data 62, 323
training dataset 332

transposition 50, 55

U

unsupervised machine learning 12

V

validation accuracy 324
validation data 328
validation loss 324
Vanilla Gradient Descent 250
vector 38
vector addition 39

W

weights 15, 54

Website: https://nnfs.io/