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.

Copyright

License for Code

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.

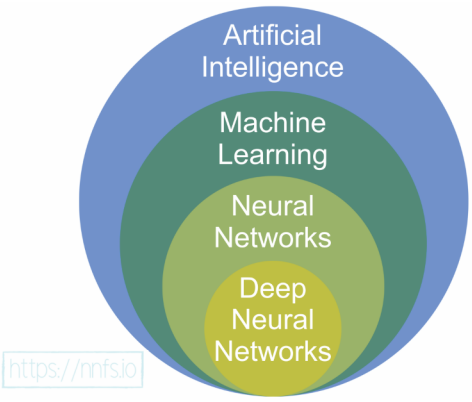


Fig 1.01: Depicting the various fields of artificial intelligence and where they fit in overall.

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

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.

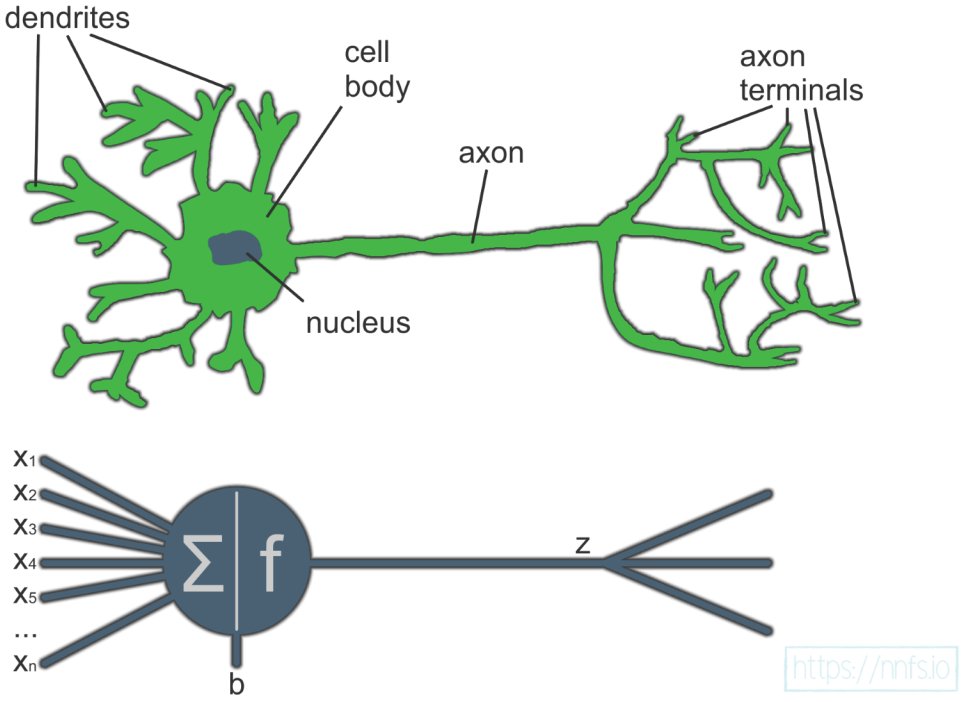


Fig 1.02: Comparing a biological neuron to an artificial neuron.

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.

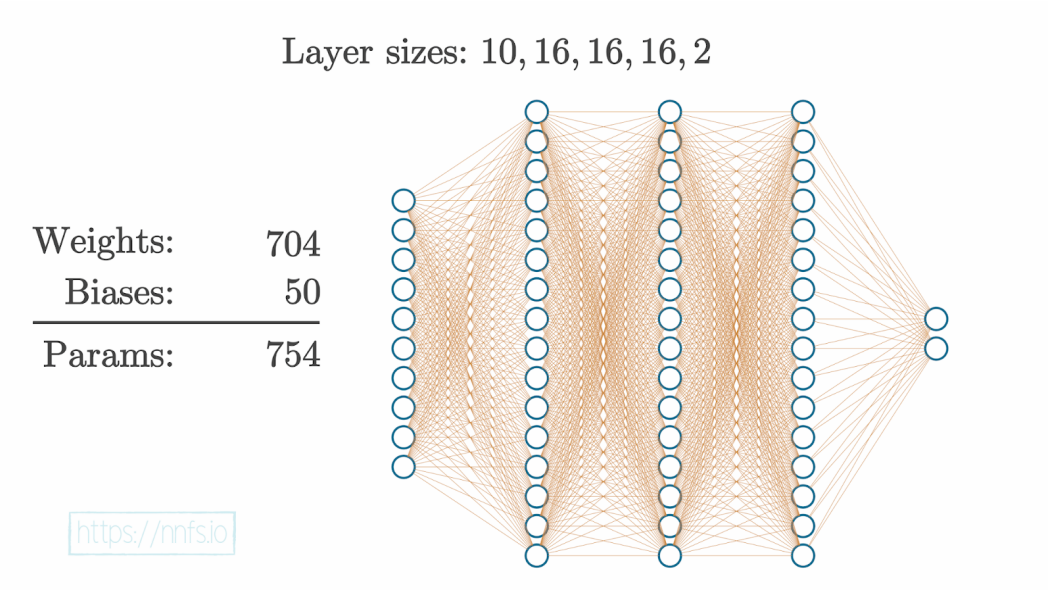


Fig 1.03: Example of a neural network with 3 hidden layers of 16 neurons each.

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:

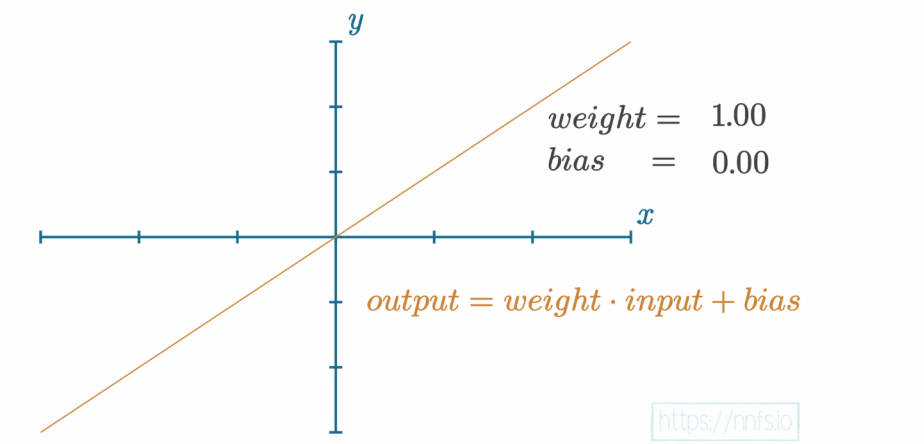


Fig 1.04: Graph of a single-input neuron's output with a weight of 1, bias of 0 and input *x*.

Adjusting the weight will impact the slope of the function:

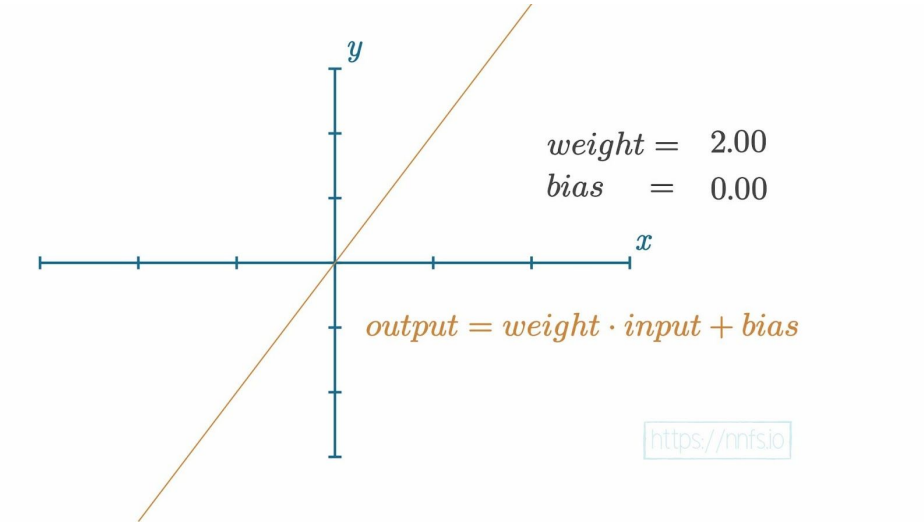


Fig 1.05: Graph of a single-input neuron's output with a weight of 2, bias of 0 and input *x*.

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:

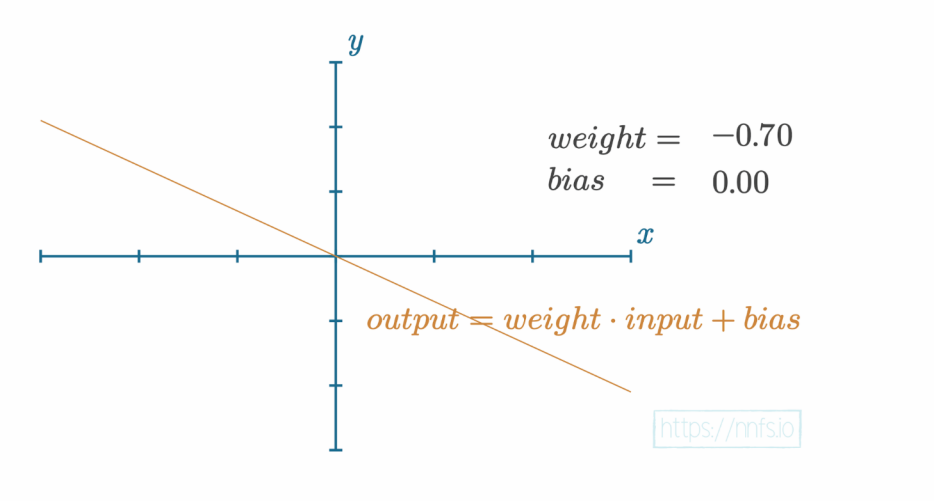


Fig 1.06: Graph of a single-input neuron's output with a weight of -0.70, bias of 0 and input *x*.

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:

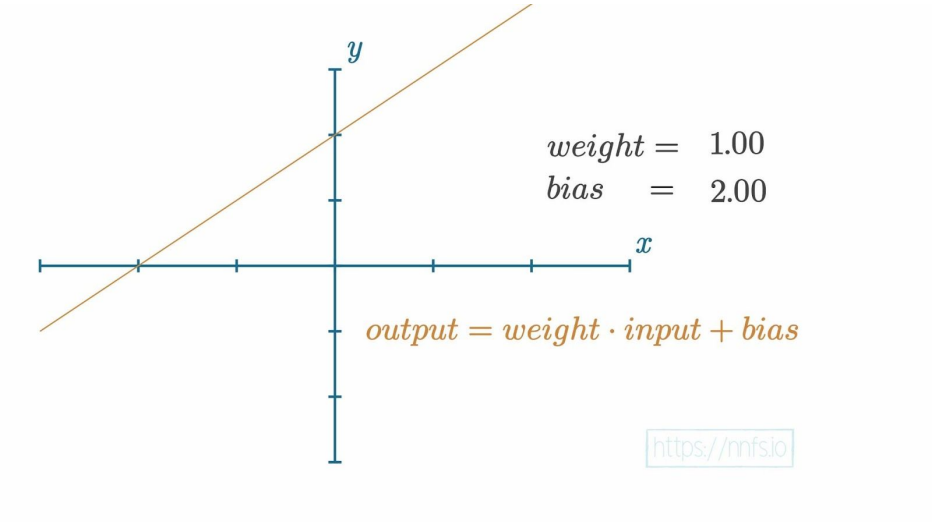


Fig 1.07: Graph of a single-input neuron's output with a weight of 1, bias of 2 and input *x*.

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:

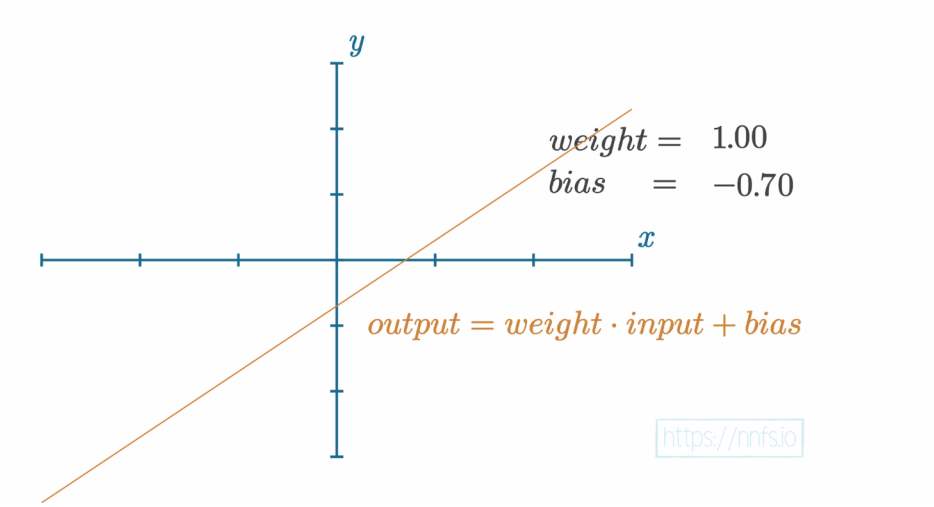


Fig 1.08: Graph of a single-input neuron's output with a weight of 1.0, bias of -0.70 and input *x*.

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.

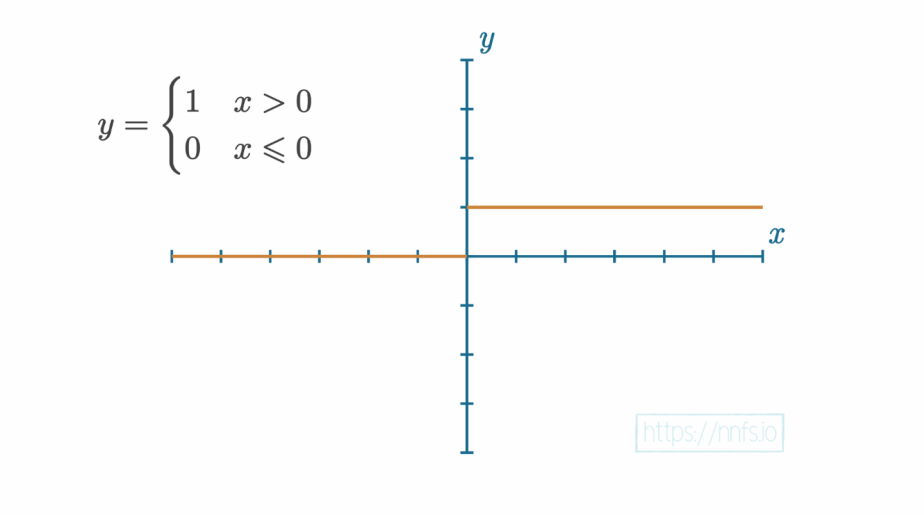


Fig 1.09: Graph of a step function.

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.

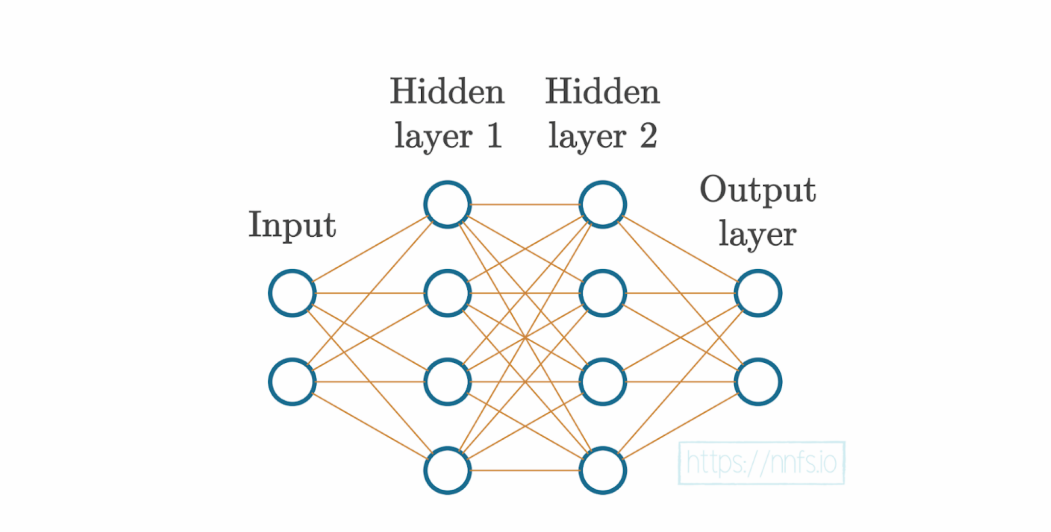


Fig 1.10: Example basic neural network.

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."

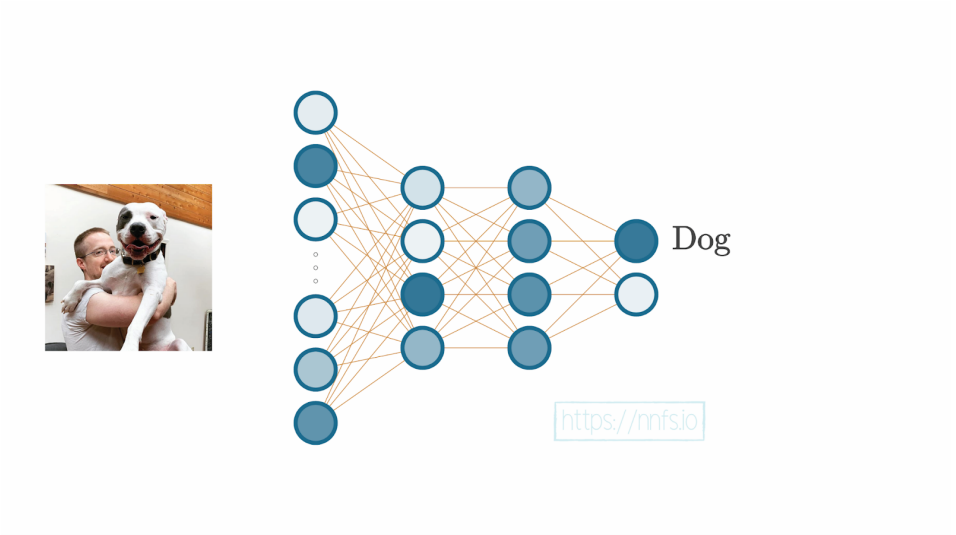


Fig 1.11: Visual depiction of passing image data through a neural network, getting a classification

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.

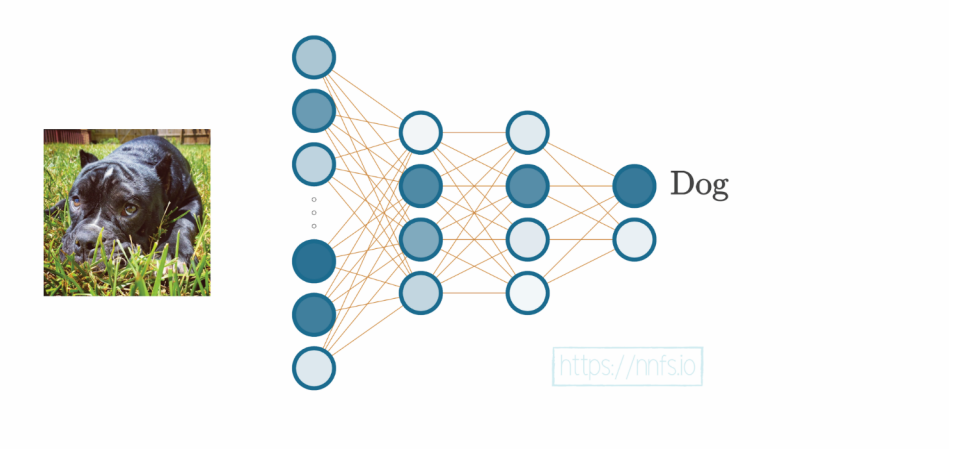


Fig 1.12: Visual depiction of passing image data through a neural network, getting a classification

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
        )
    )
)
```


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.

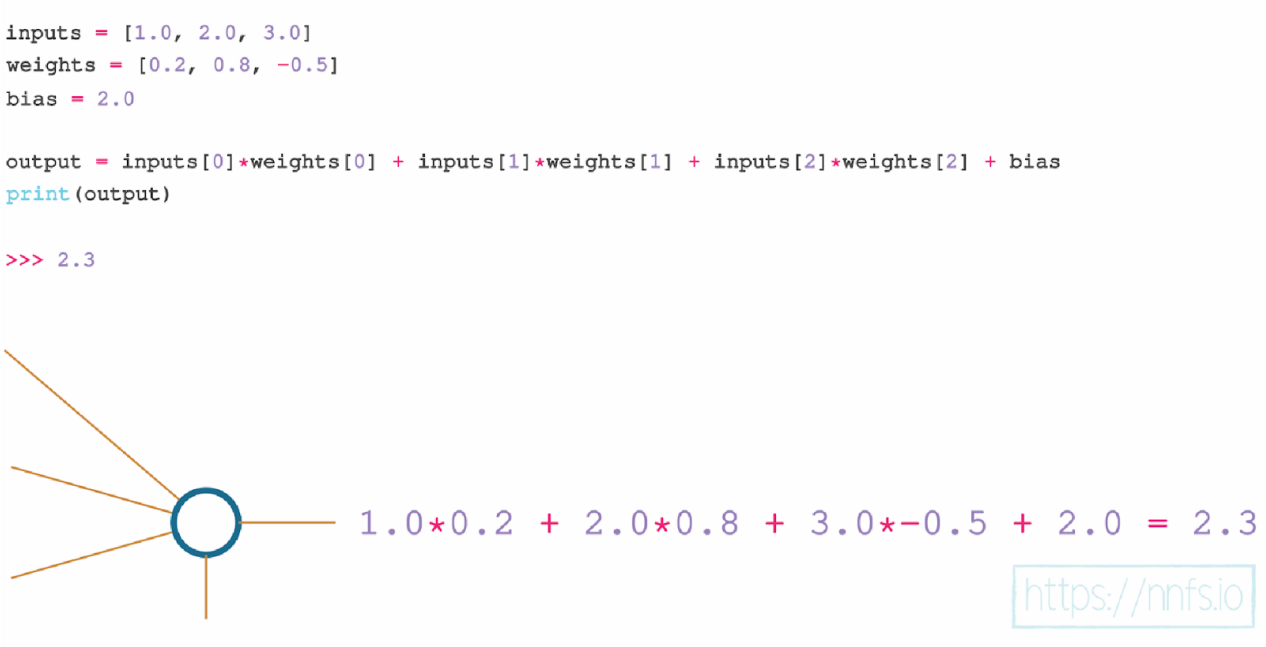


Fig 2.01: Visualizing the code that makes up the math of a basic neuron.

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:

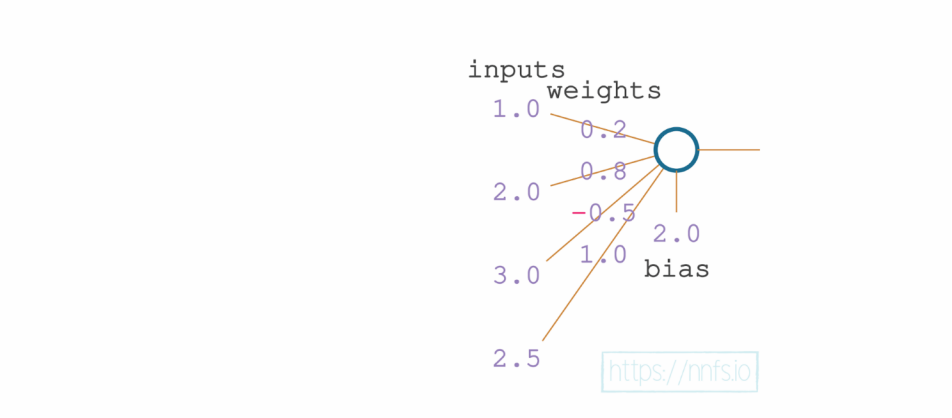


Fig 2.02: Visualizing how the inputs, weights, and biases from the code interact with the neuron.

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:

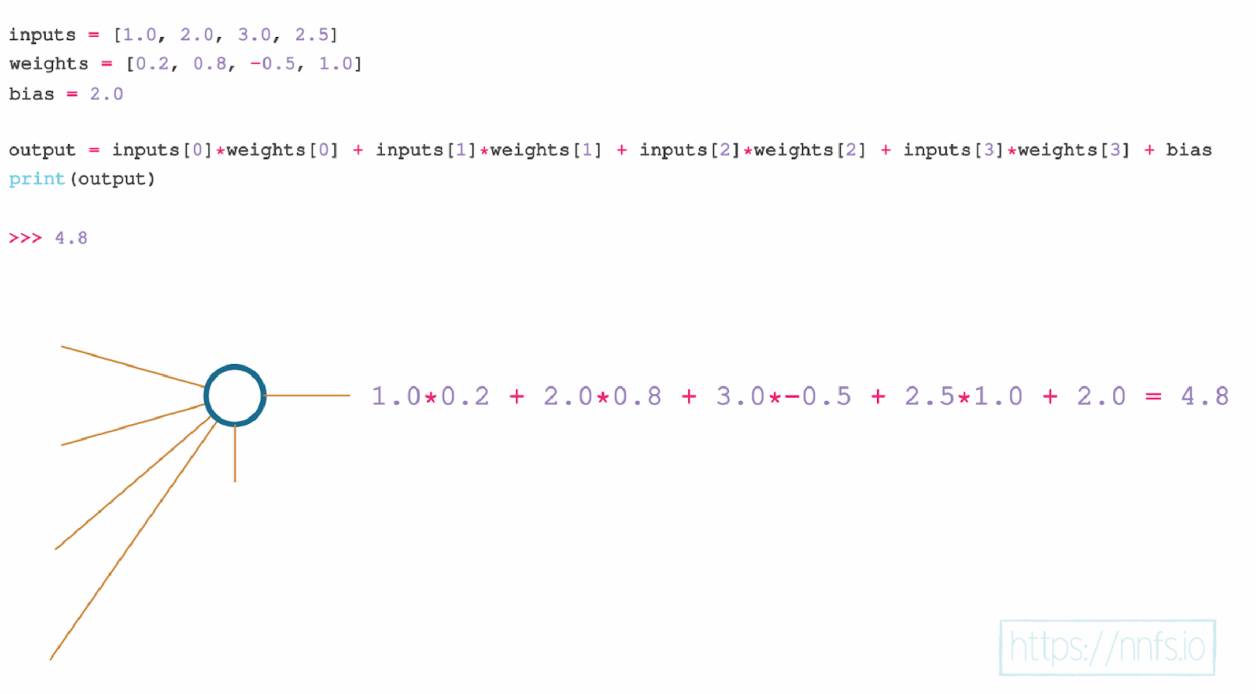


Fig 2.03: Visualizing the code that makes up a basic neuron, with 4 inputs this time.

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:

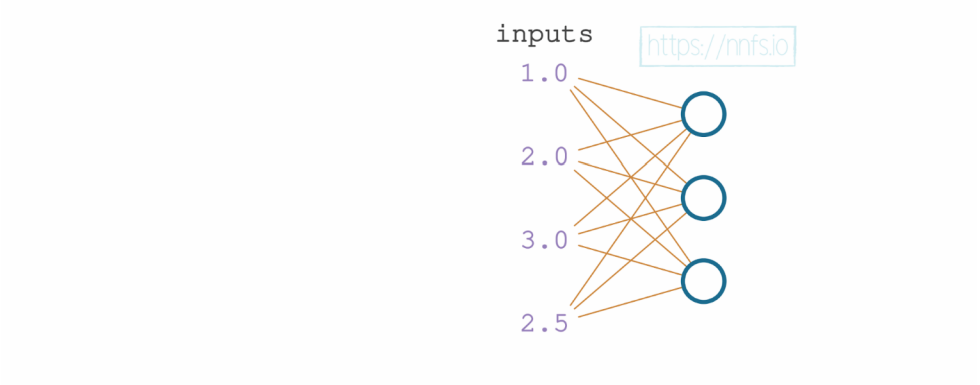


Fig 2.04: Visualizing a layer of neurons with common input.

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]
```

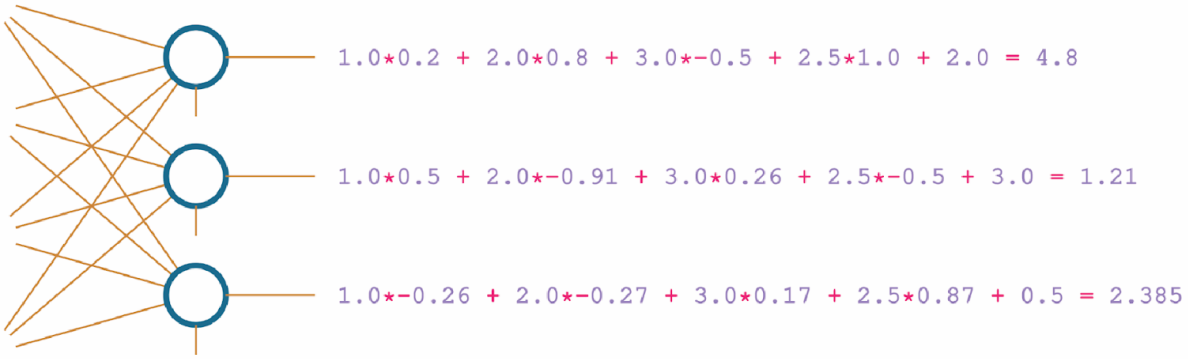



Fig 2.04.2: Code, math and visuals behind a layer of neurons.

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.

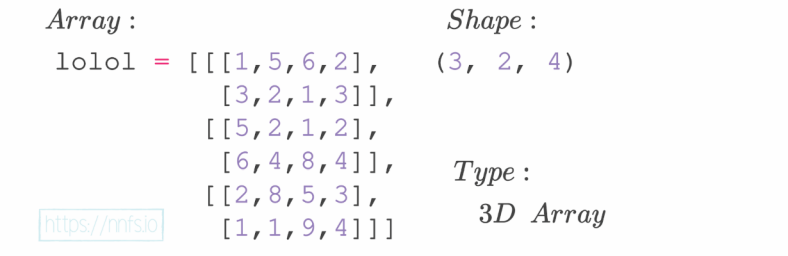


Fig 2.05: Example of a 3-dimensional array.

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
```

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

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
```

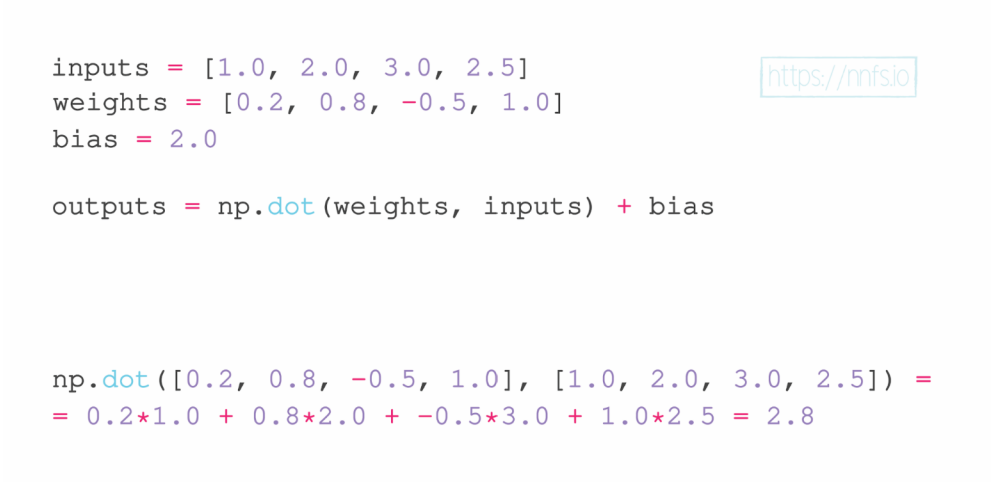


Fig 2.07: Visualizing the math of the dot product of inputs and weights for a single neuron.

```
inputs = [1.0, 2.0, 3.0, 2.5]                    https://nnfs.io
weights = [0.2, 0.8, -0.5, 1.0]
bias = 2.0

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


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 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 input 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])
```

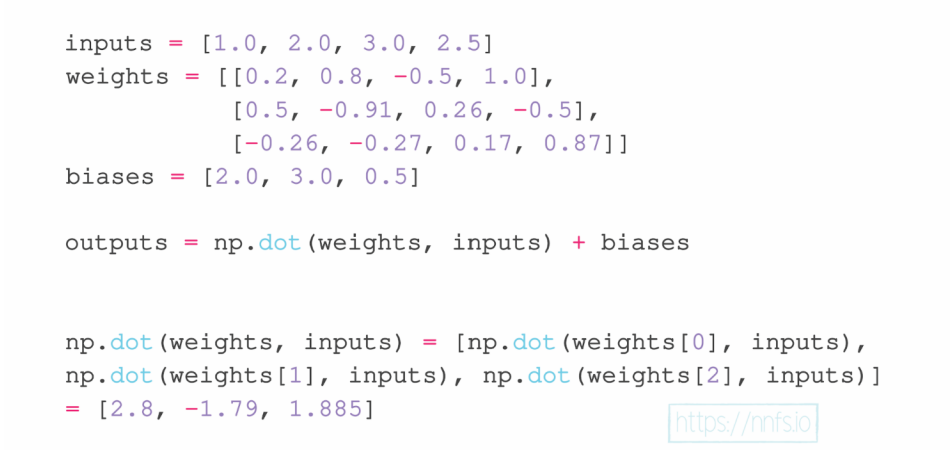


Fig 2.09: Code and visuals for the dot product applied to the layer of neurons.

```
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])
```


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**.

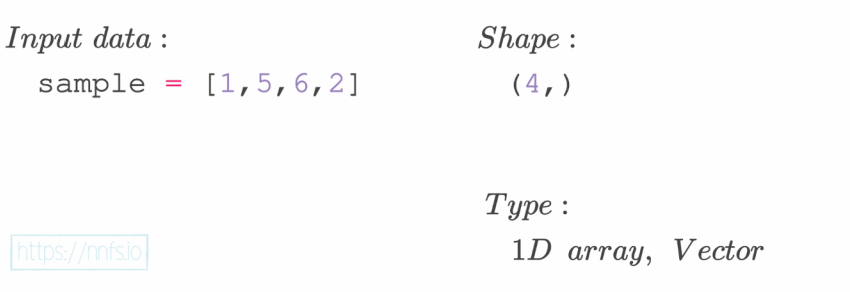


Fig 2.11: Visualizing a 1D array.

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:

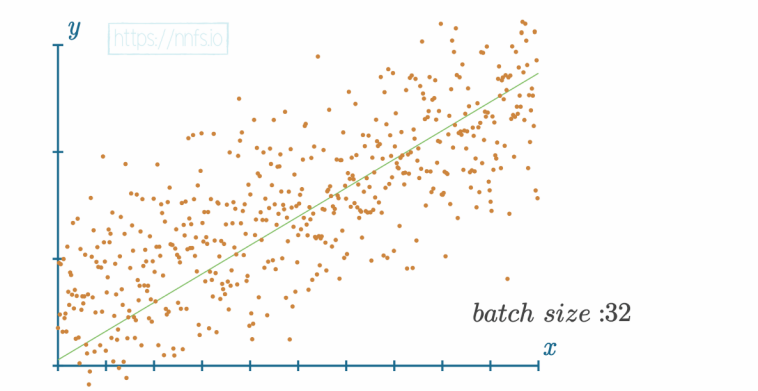


Fig 2.12: Example of a linear equation fitting batches of 32 chosen samples. See animation below for other sizes of samples at a time to see how much of a difference batch size can make.

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

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**:

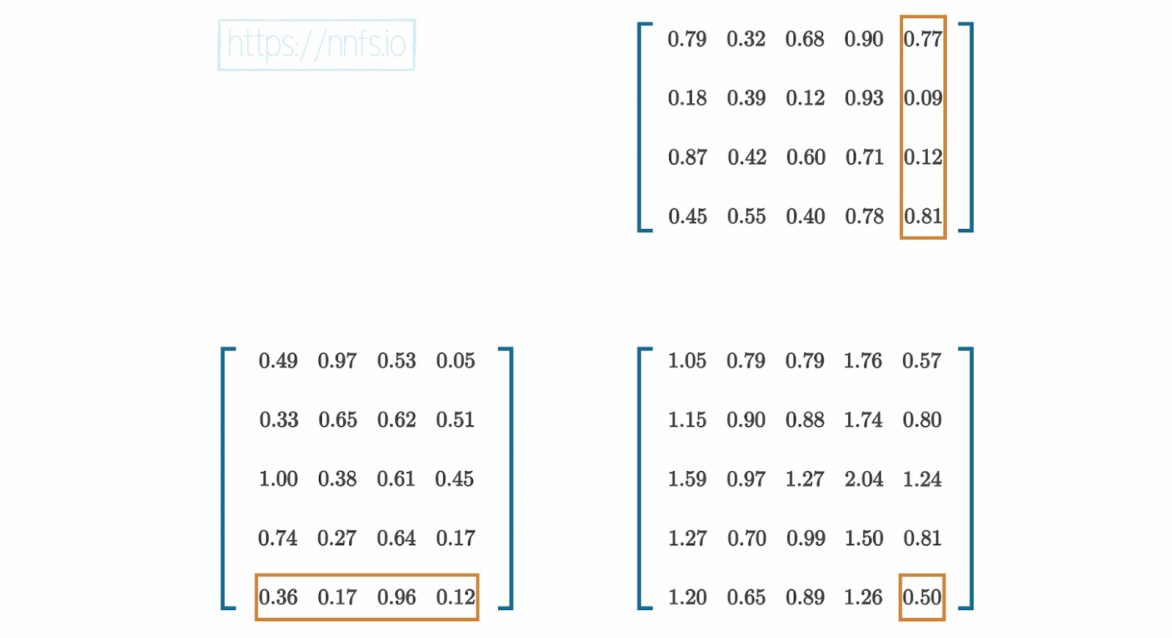


Fig 2.14: Visualizing how a single element in the resulting matrix from matrix product is calculated. See animation for the full calculation of each element.

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:

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

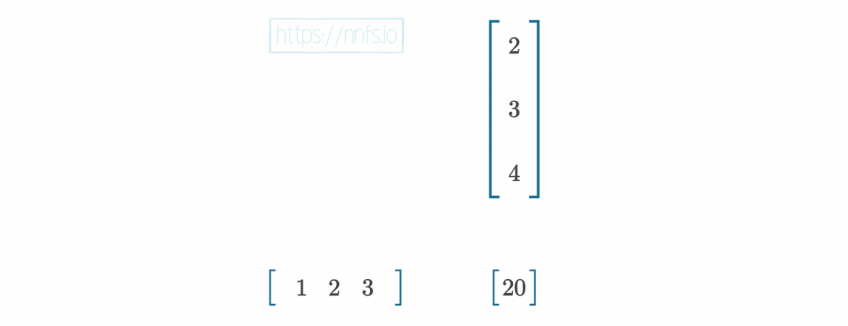


Fig 2.15: Product of row and column vectors.

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:

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

Fig 2.16: Example of an array transposition.

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

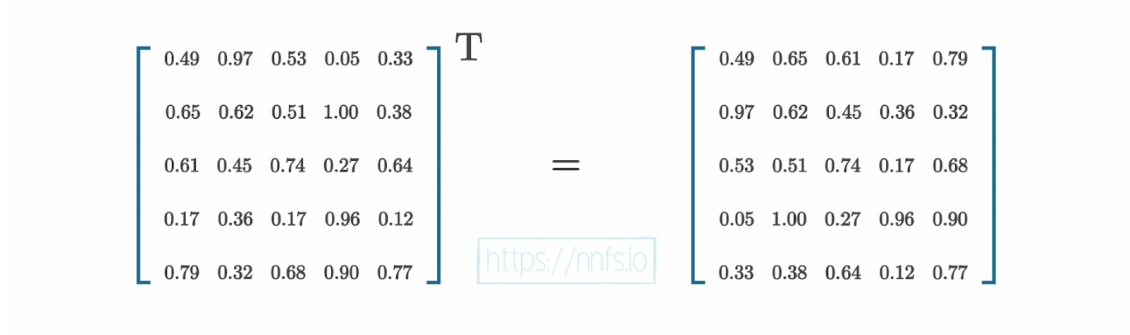


Fig 2.17: Another example of an array transposition.

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]
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]
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


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]]
```

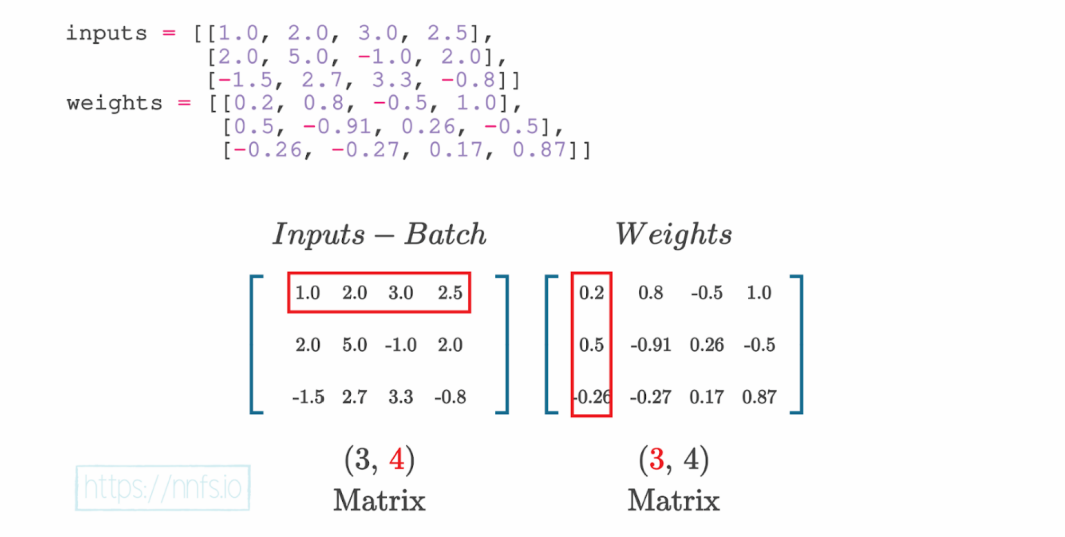


Fig 2.18: Depiction of why we need to transpose to perform the matrix product.

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]]
```

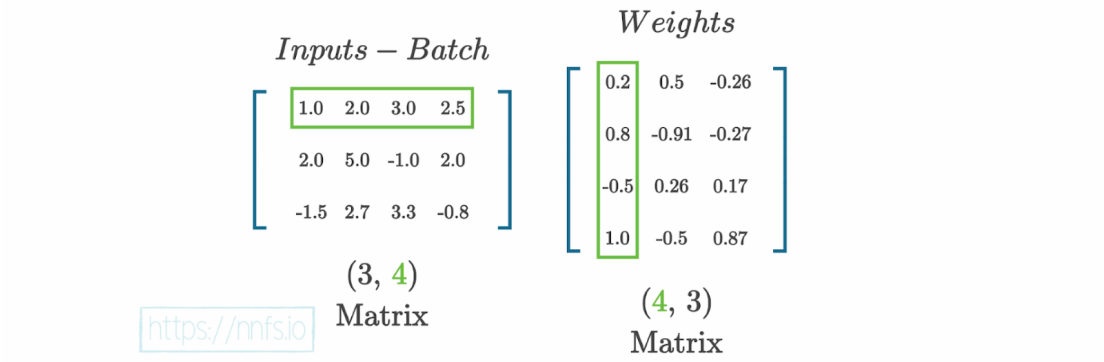


Fig 2.19: After transposition, we can perform the matrix product.

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)
```


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

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]])
```


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

Fig 2.21: Code and visuals for inputs multiplied 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.

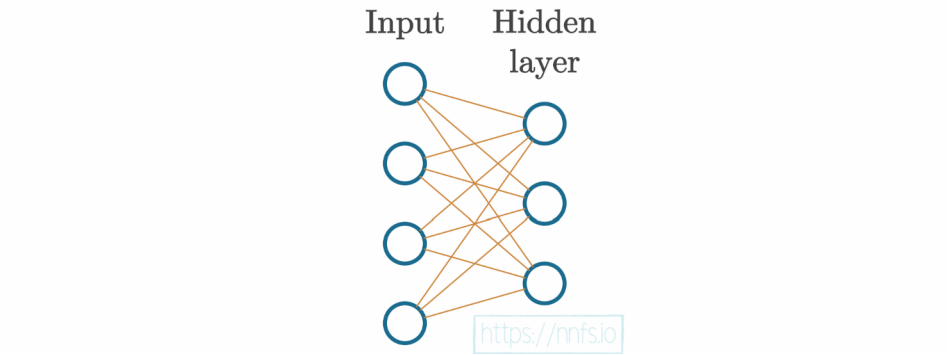


Fig 3.01: Input layer with 4 features into a hidden layer with 3 neurons.

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:

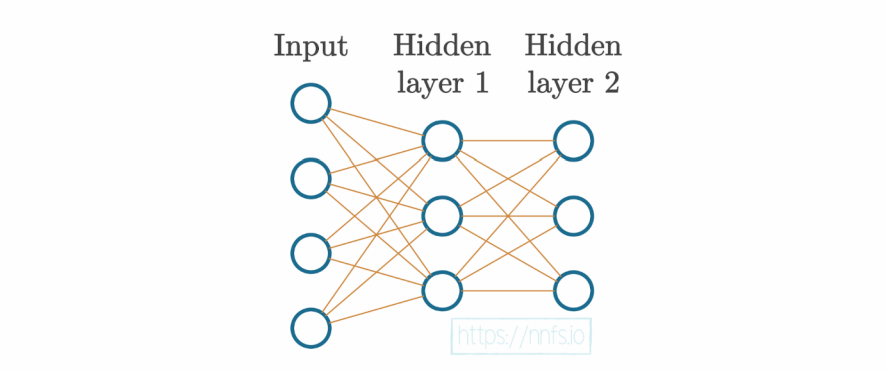


Fig 3.02: 4 features input into 2 hidden layers of 3 neurons each.

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.

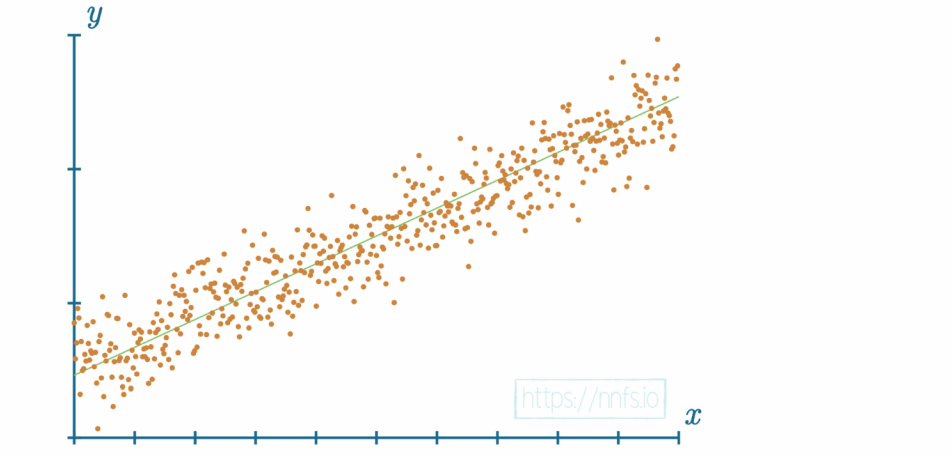


Fig 3.03: Example of data (orange dots) that can be represented (fit) by a straight line (green line).

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

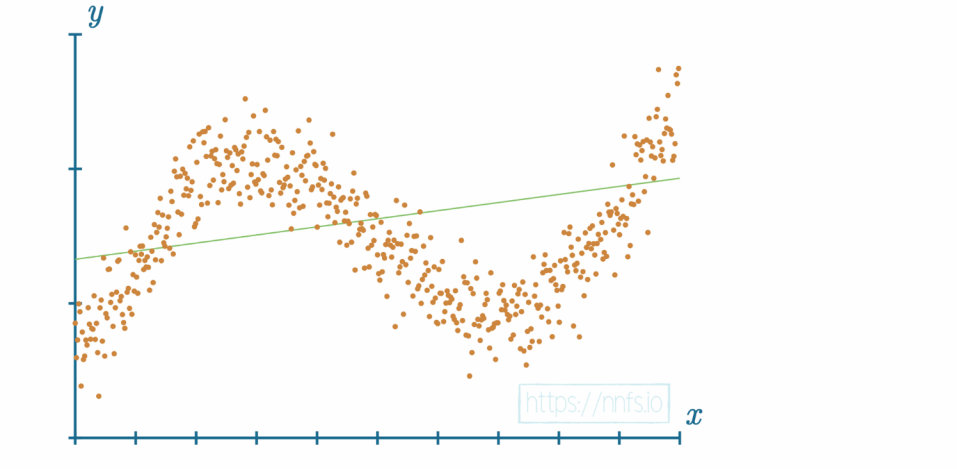


Fig 3.04: Example of data (orange dots) that is not well fit by a straight line.

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()
```

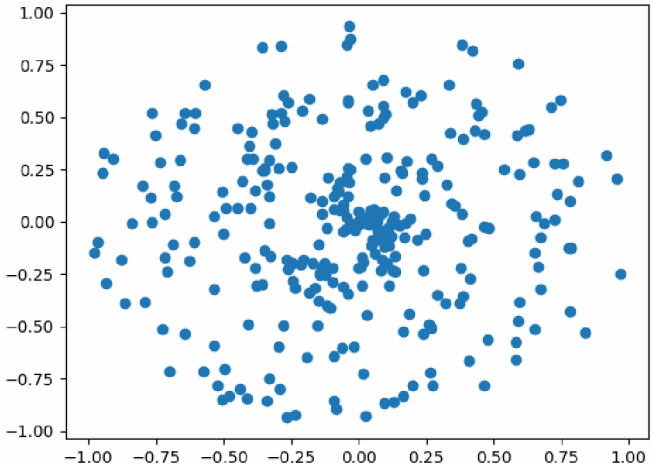


Fig 3.05: Uncolored spiral dataset.

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()
```

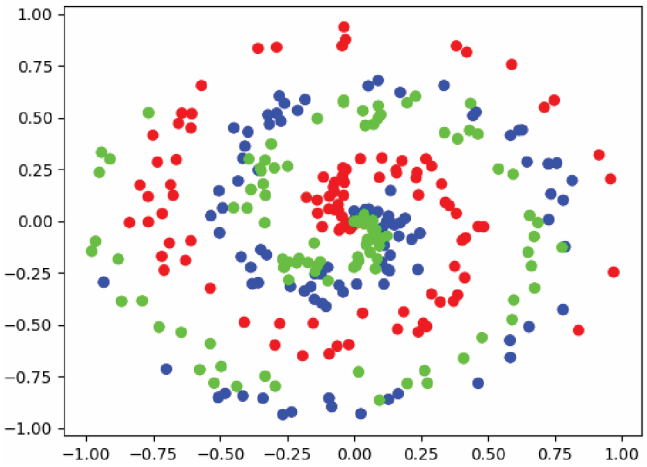


Fig 3.06: Spiral dataset colored by class.

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.

```
class Layer_Dense:

    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 pretrained 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 usual 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.

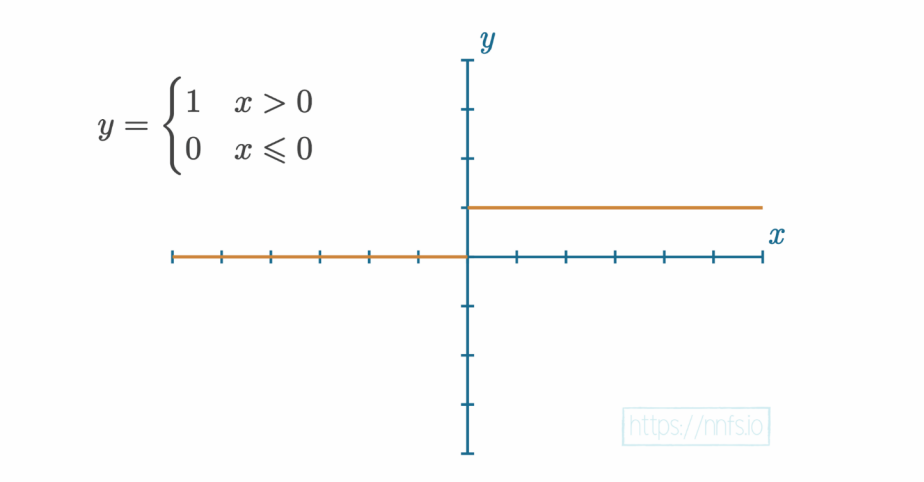


Fig 3.07: Graph of a step function.

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 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:

```
class Layer_Dense:

    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))

    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 the 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.

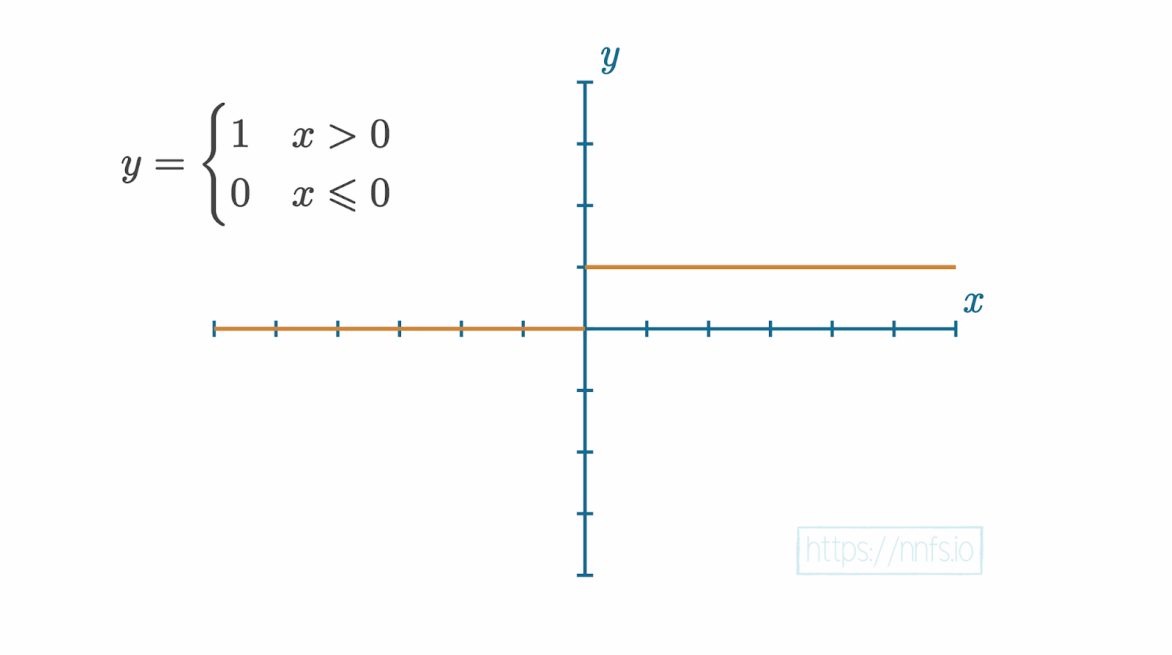


Fig 4.01: Step function graph.

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.

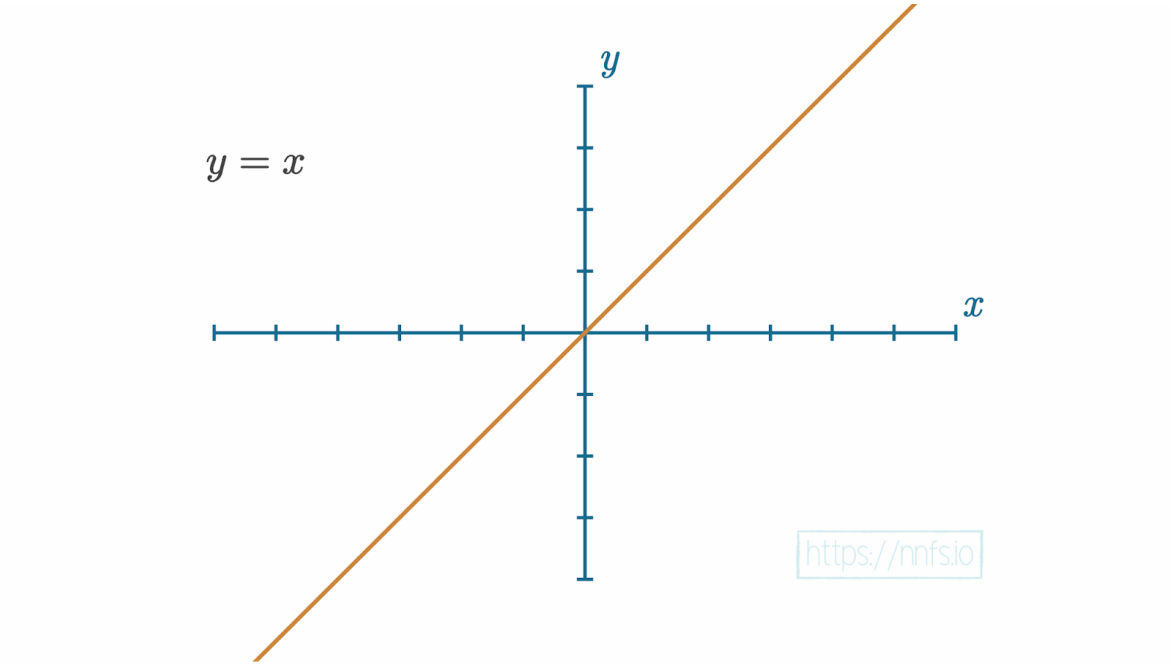


Fig 4.02: Linear function graph.

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:

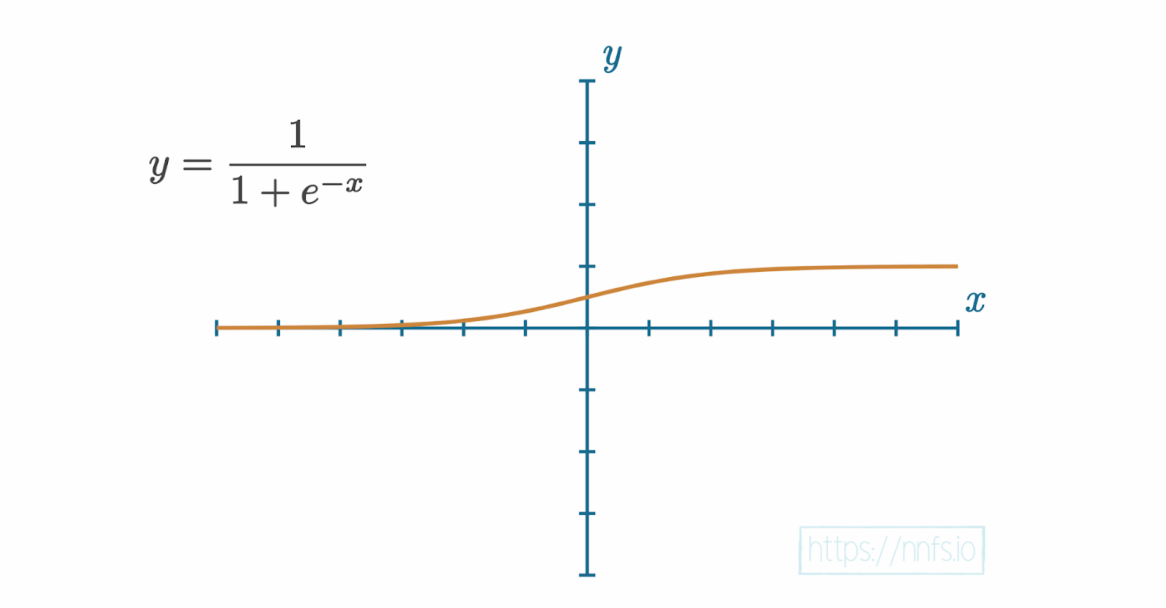


Fig 4.03: Sigmoid function graph.

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

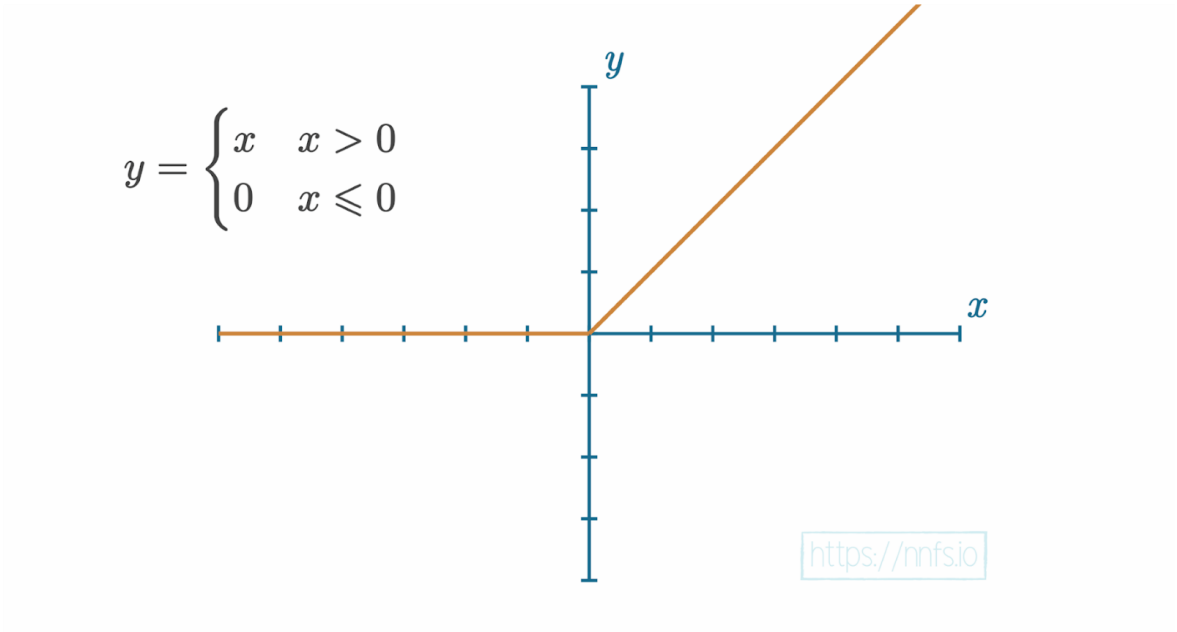


Fig 4.04: Graph of the ReLU activation function.

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:

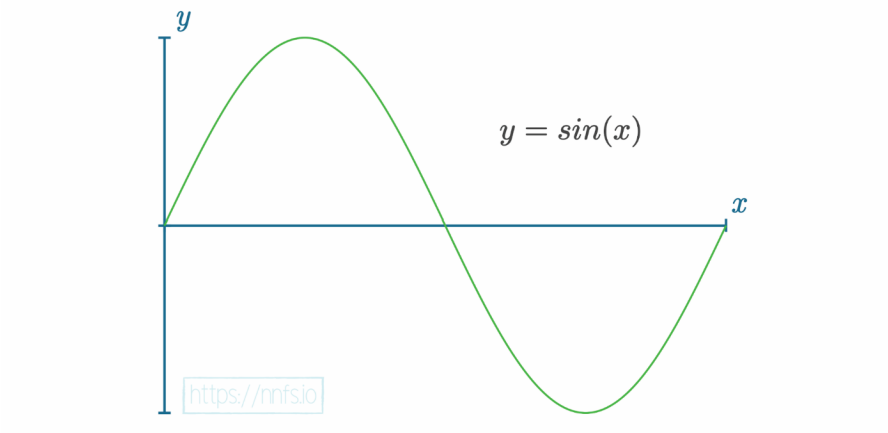


Fig 4.05: Graph of y=sin(x)

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:

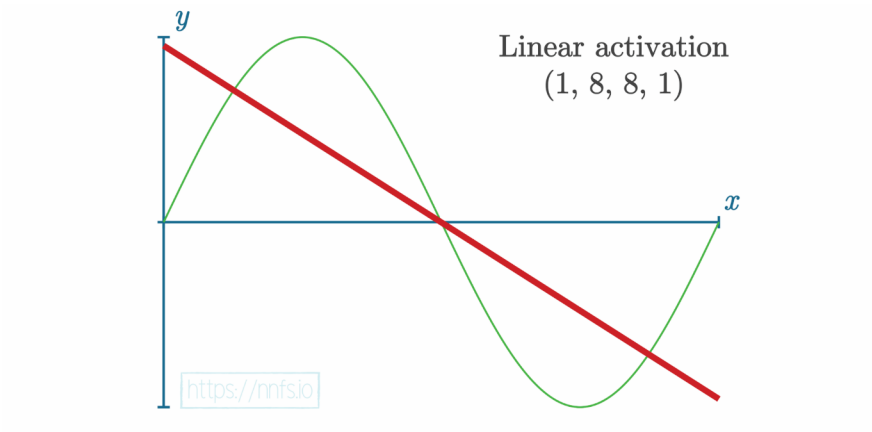


Fig 4.06: Neural network with linear activation functions in hidden layers attempting to fit y=sin(x)

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

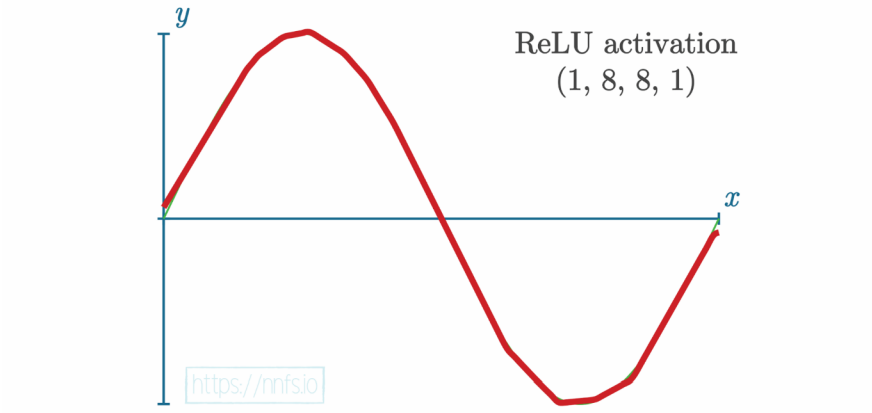


Fig 4.07: ReLU activation functions in hidden layers attempting to fit y=sin(x)

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:

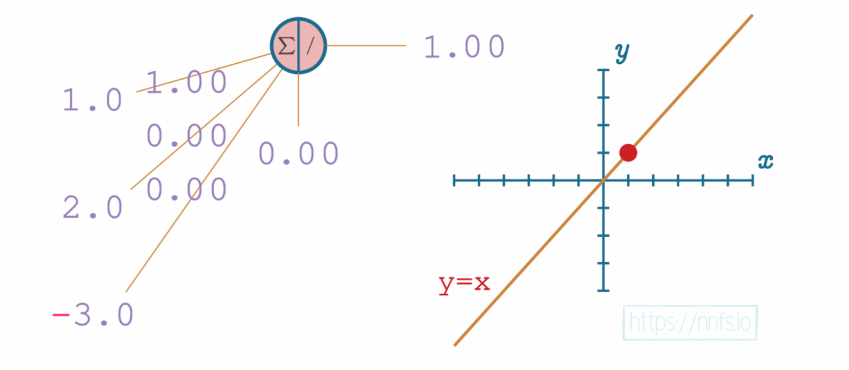


Fig 4.08: Example of output with a neuron using a linear activation function.

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

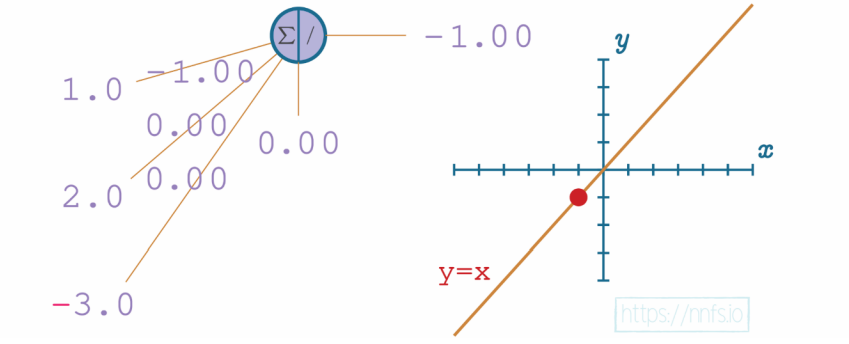


Fig 4.09: Example of output with a neuron using a linear activation function, updated weight.

And updating weights and additionally a bias:

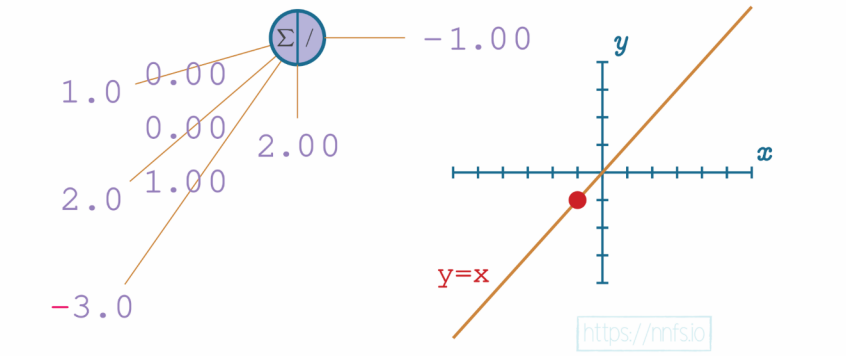


Fig 4.10: Example of output with a neuron using a linear activation function, updated another weight.

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:

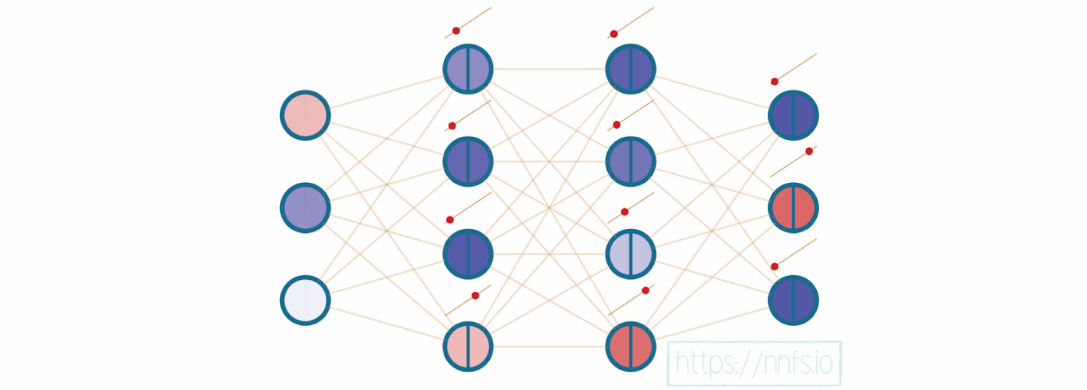


Fig 4.11: A neural network with all linear activation functions.

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:

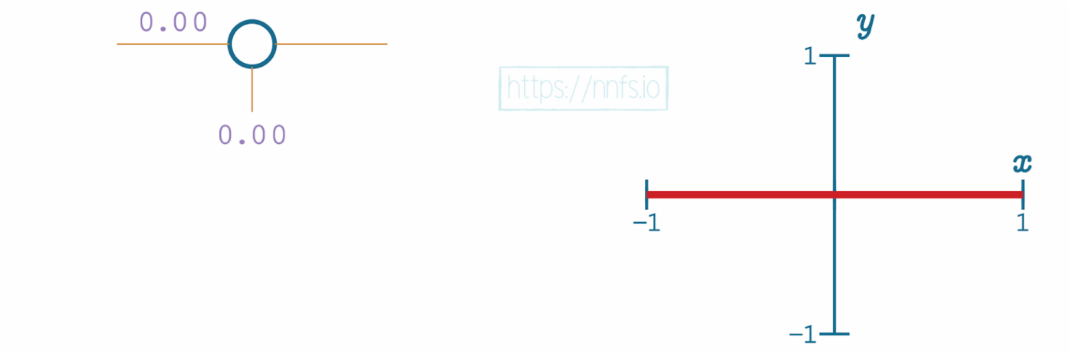


Fig 4.12: Single neuron with single input (zeroed weight) and ReLU activation function.

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:

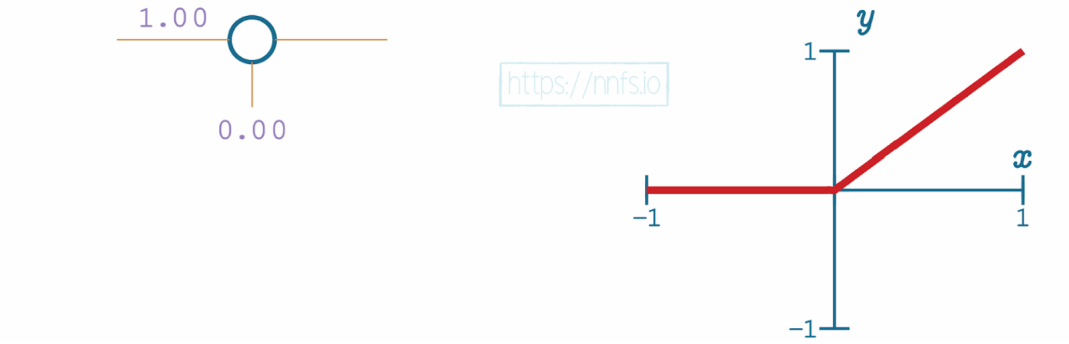


Fig 4.13: Single neuron with single input and ReLU activation function, weight set to 1.0.

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

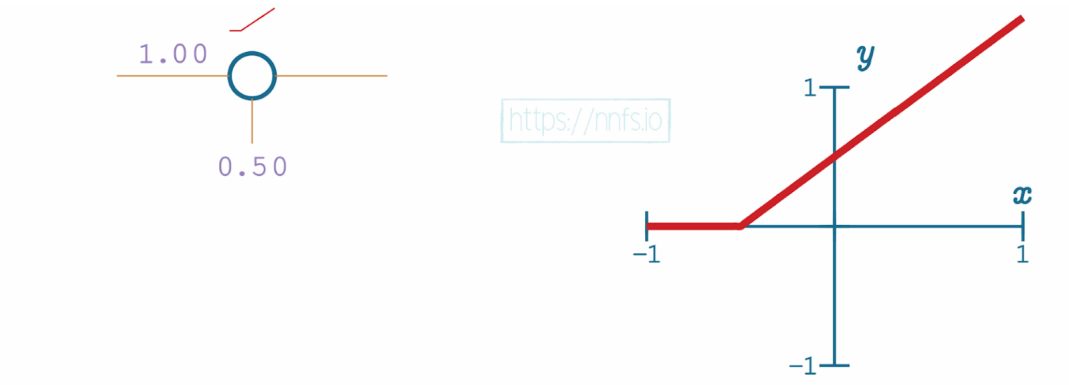


Fig 4.14: Single neuron with single input and ReLU activation function, bias applied.

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?

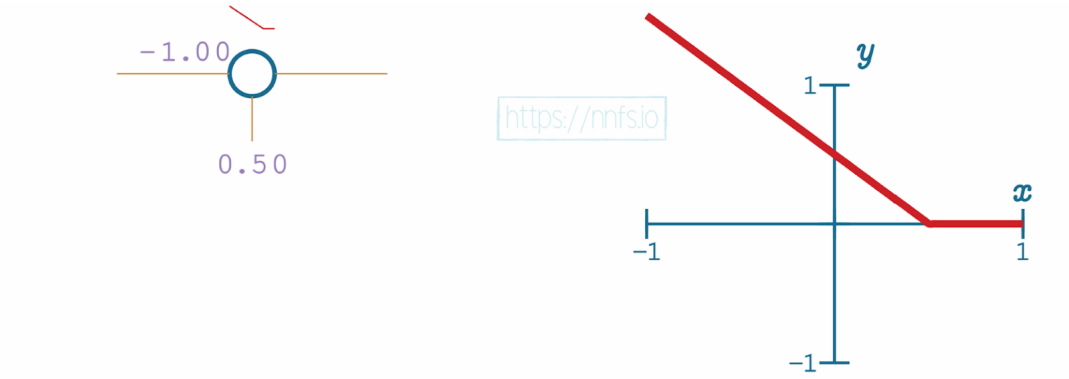


Fig 4.15: Single neuron with single input and ReLU activation function, negative weight.

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:

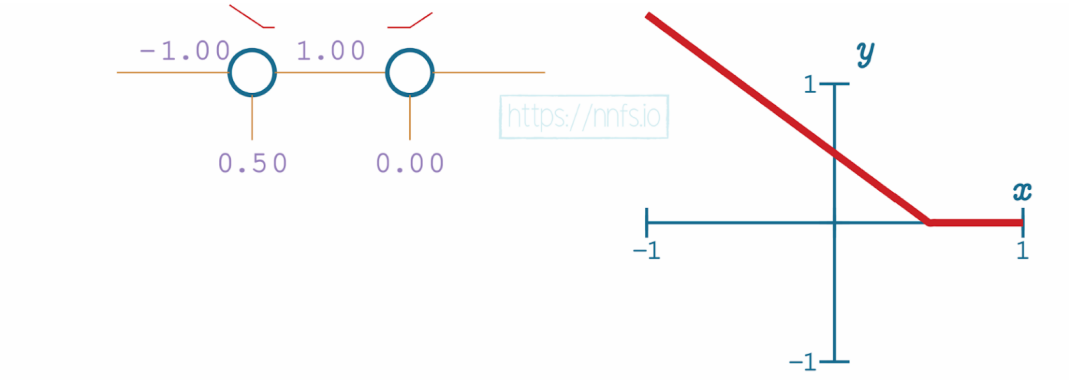


Fig 4.16: Pair of neurons with single inputs and ReLU activation functions.

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:

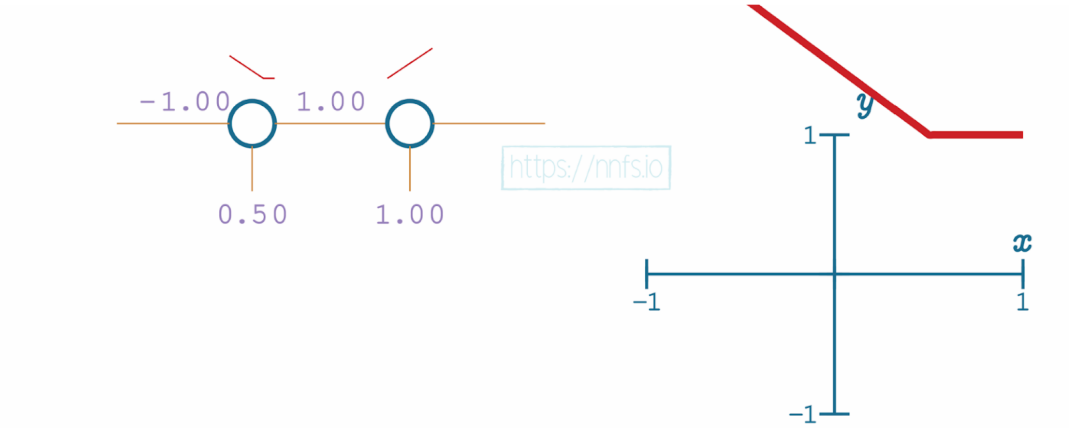


Fig 4.17: Pair of neurons with single inputs and ReLU activation functions, other bias applied.

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?

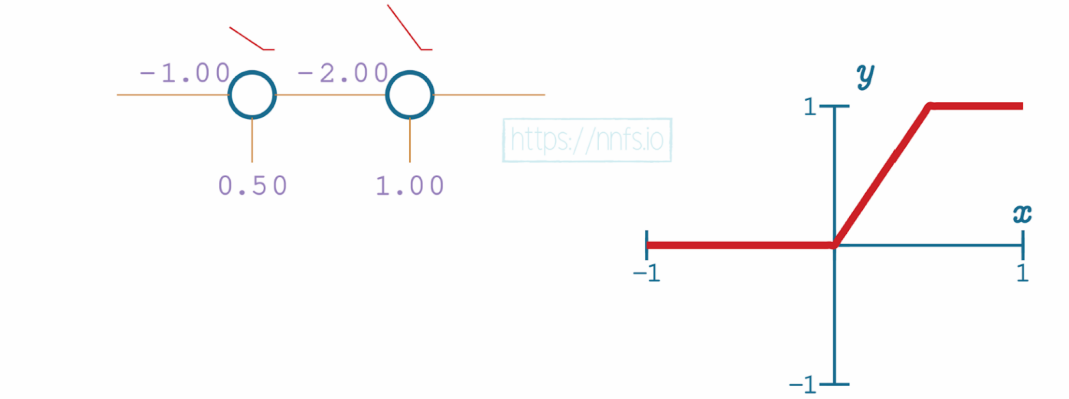


Fig 4.18: Pair of neurons with single inputs and ReLU activation functions, other negative weight.

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:

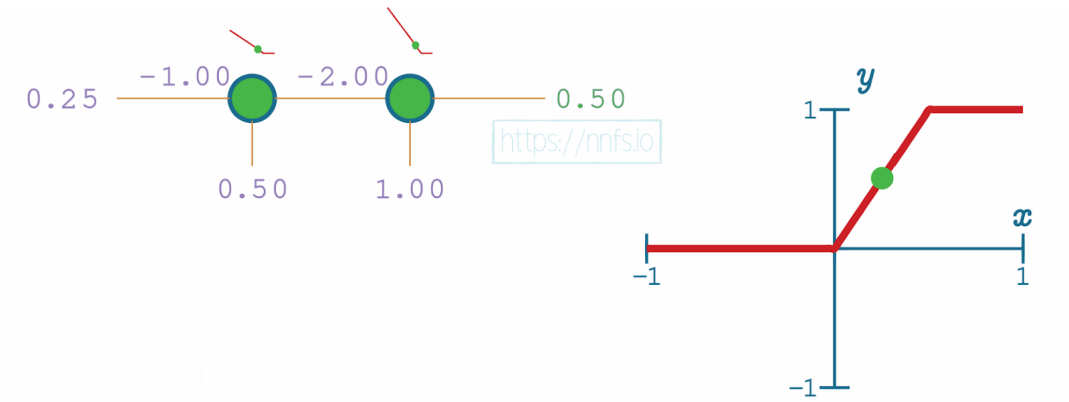


Fig 4.19: Pair of neurons with single inputs and ReLU activation functions, area of effect.

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:

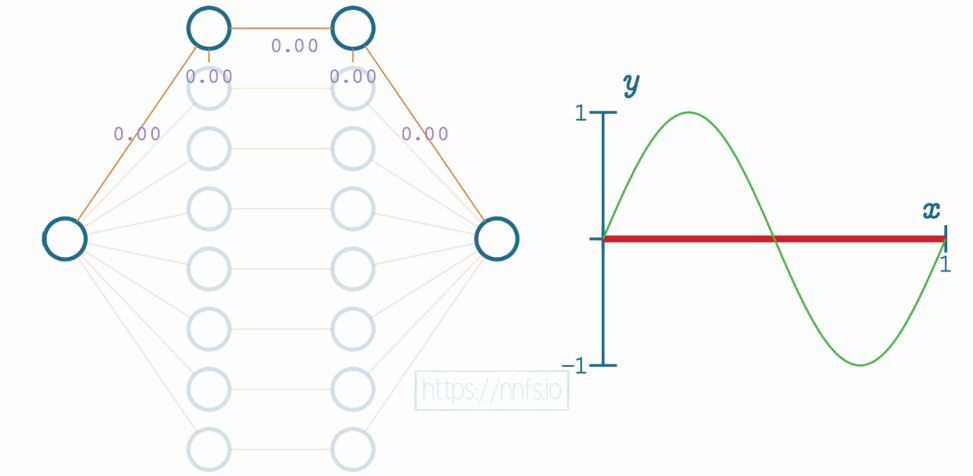


Fig 4.20: Hand-tuning a neural network starting with the first pair of neurons.

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:

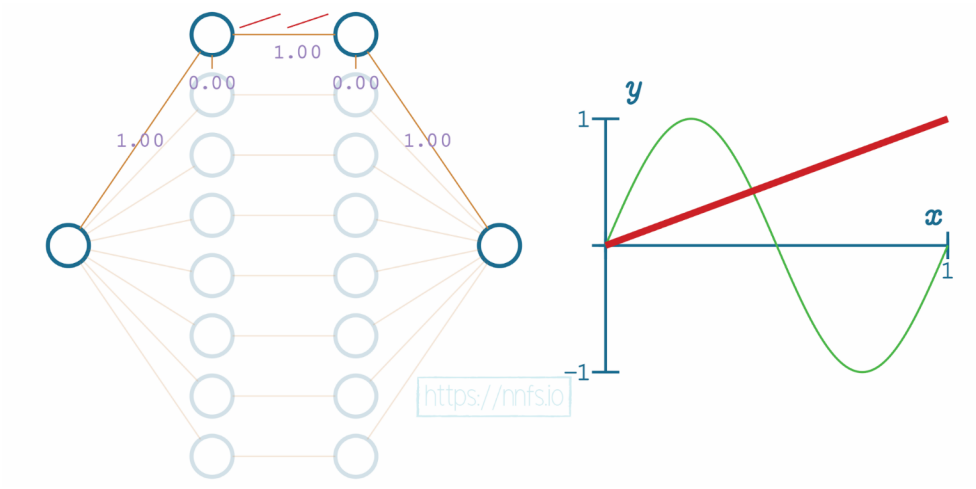


Fig 4.21: Adjusting weights for the first/top pair of neurons all to 1.

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:

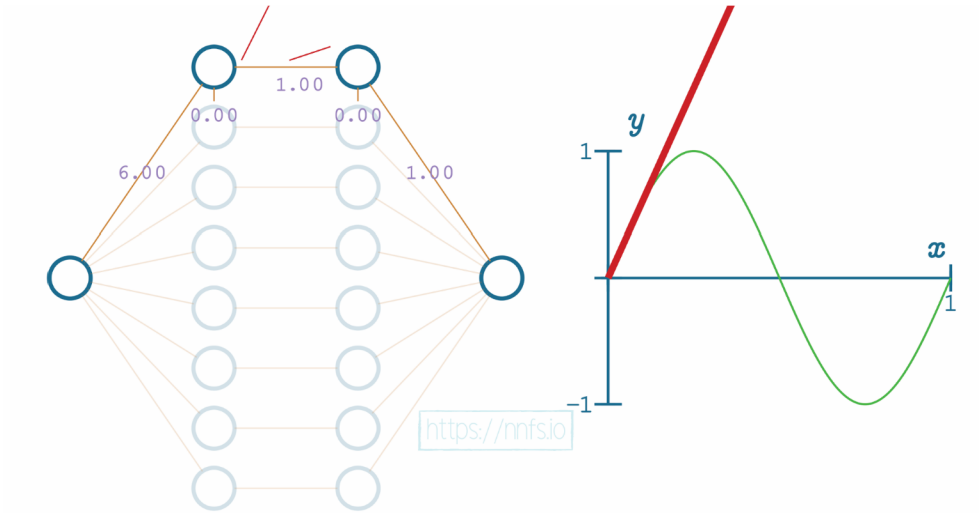


Fig 4.22: Setting weight for first hidden neuron to 6.

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*:

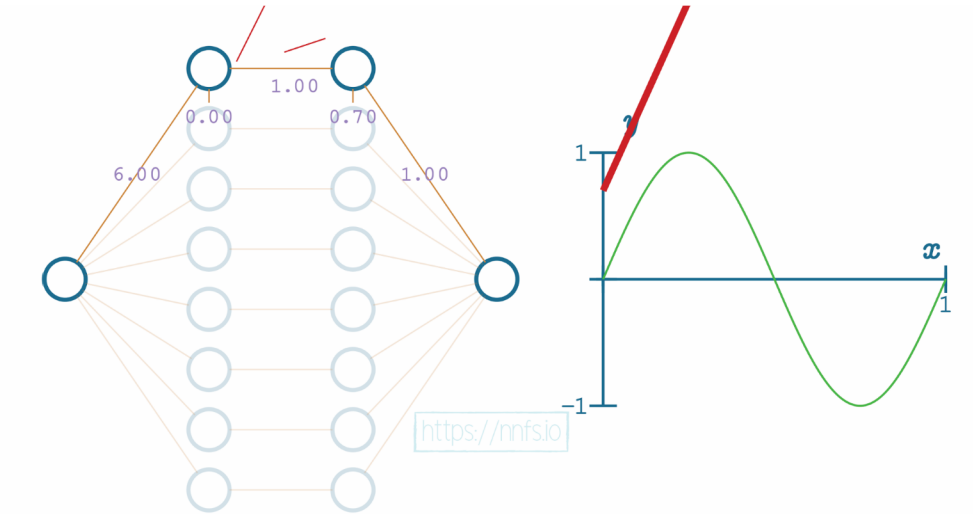


Fig 4.23: Using the bias for the 2nd hidden neuron in the top pair to offset function vertically.

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:

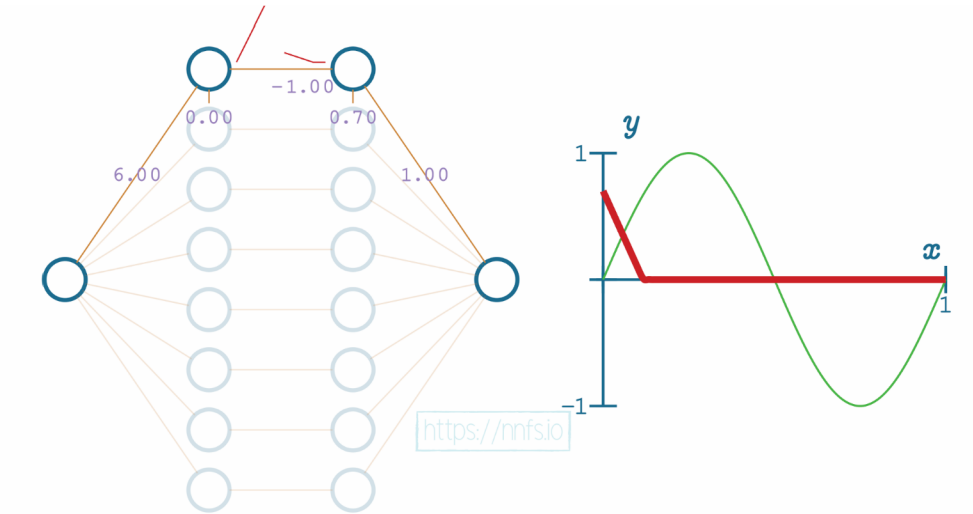


Fig 4.24: Setting the weight for the 2nd neuron in the top pair to -1.

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:

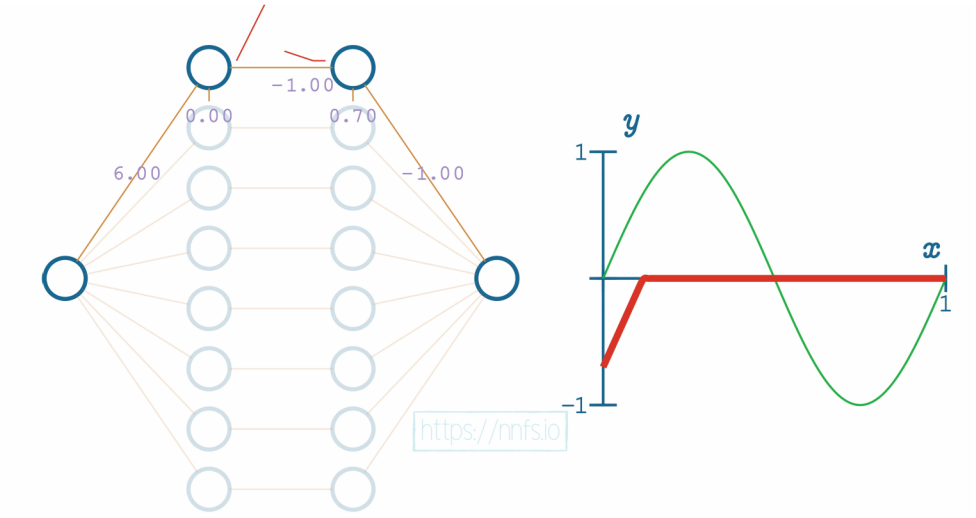


Fig 4.25: Setting the weight to the output neuron to -1.

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:

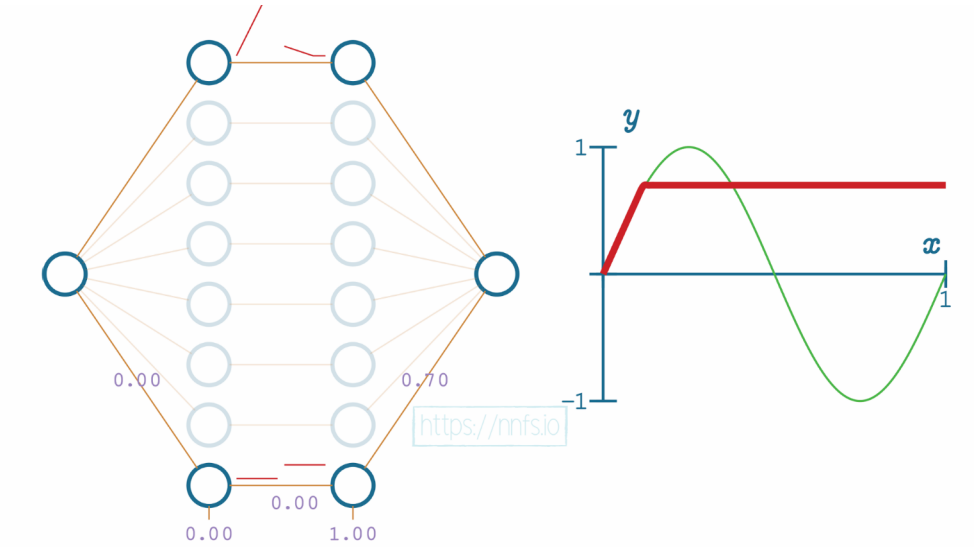


Fig 4.26: Using the bottom pair of neurons to offset the entire neural network function.

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:

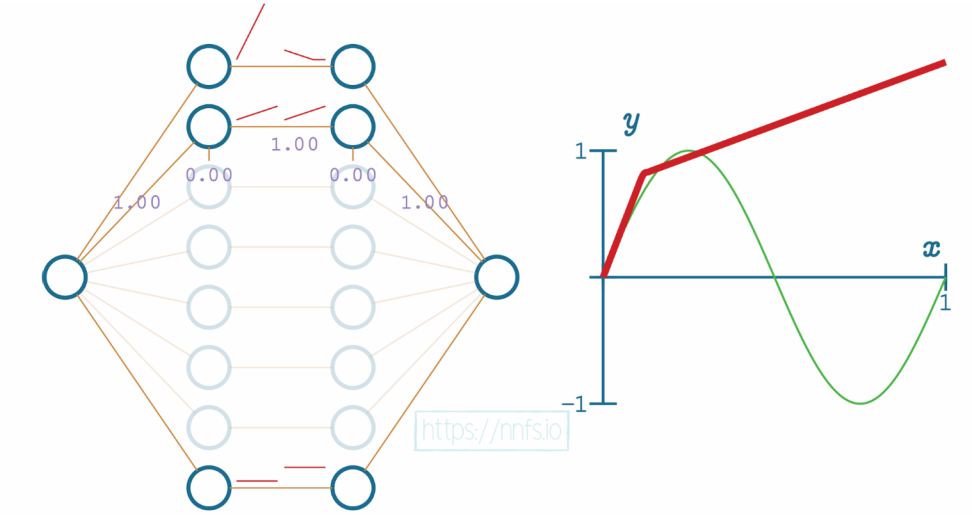


Fig 4.27: Starting to adjust the 2nd pair of neurons (from the top) for the next segment of the overall function.

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:

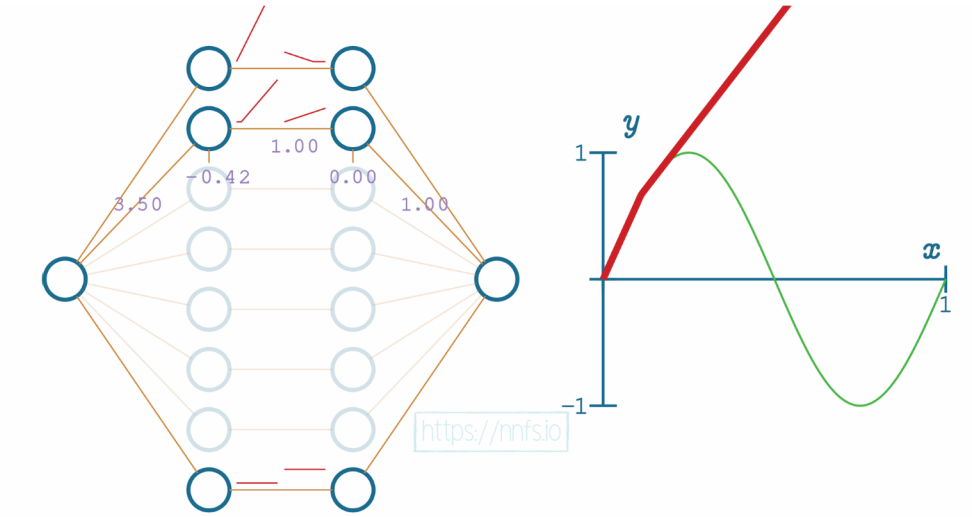


Fig 4.28: Adjusting the weight and bias into the first neuron of the 2nd pair.

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.

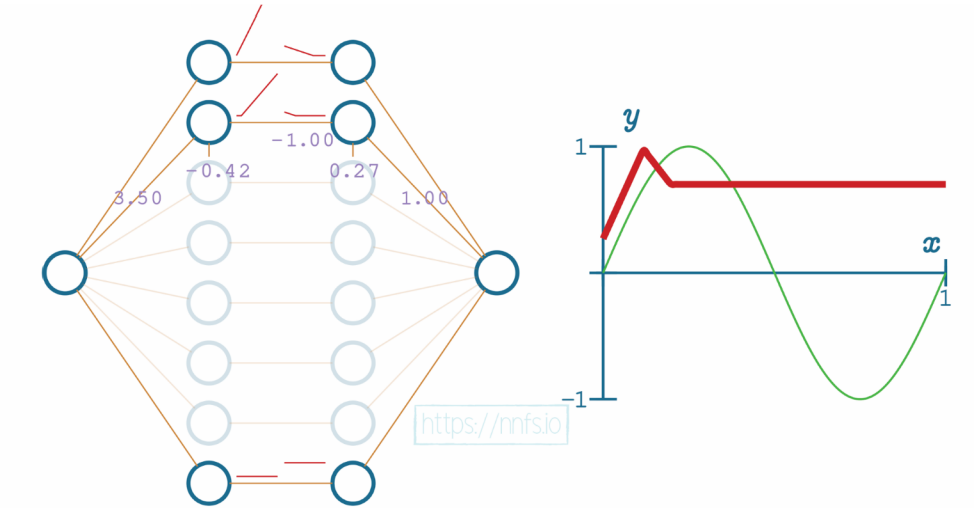


Fig 4.29: Adjusting the bias of the 2nd neuron in the 2nd pair.

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:

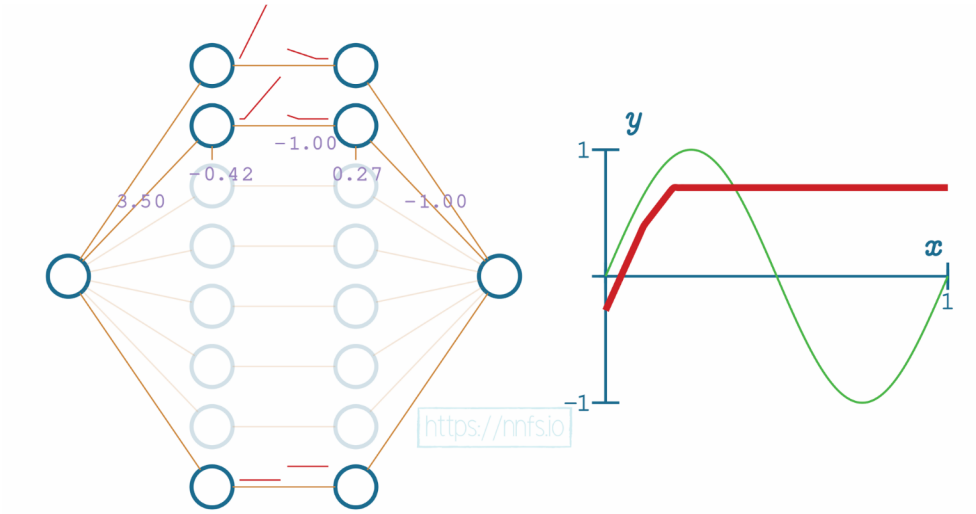


Fig 4.30: Flipping the 2nd pair's function segment, flipping the weight to the output neuron.

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

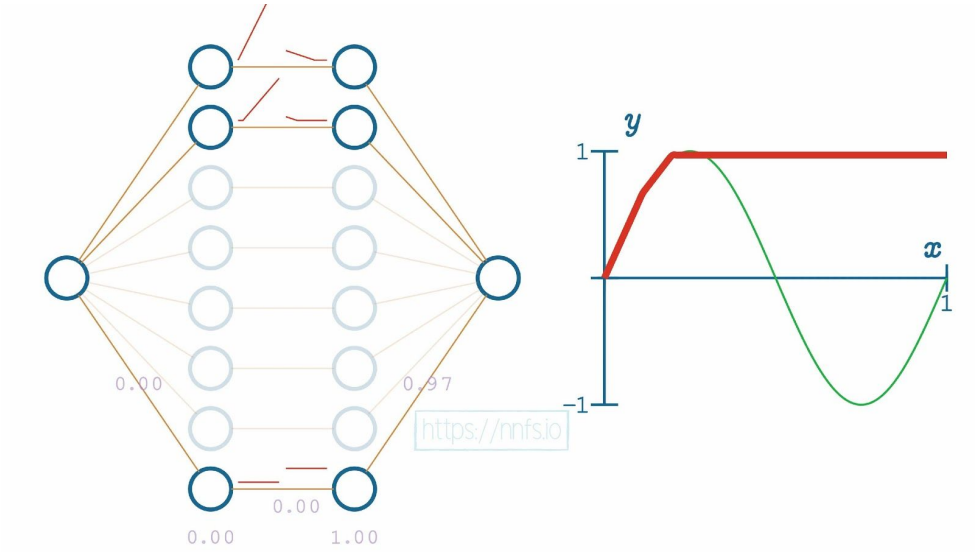


Fig 4.31: Using the bottom pair of neurons to adjust the network's overall function.

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:

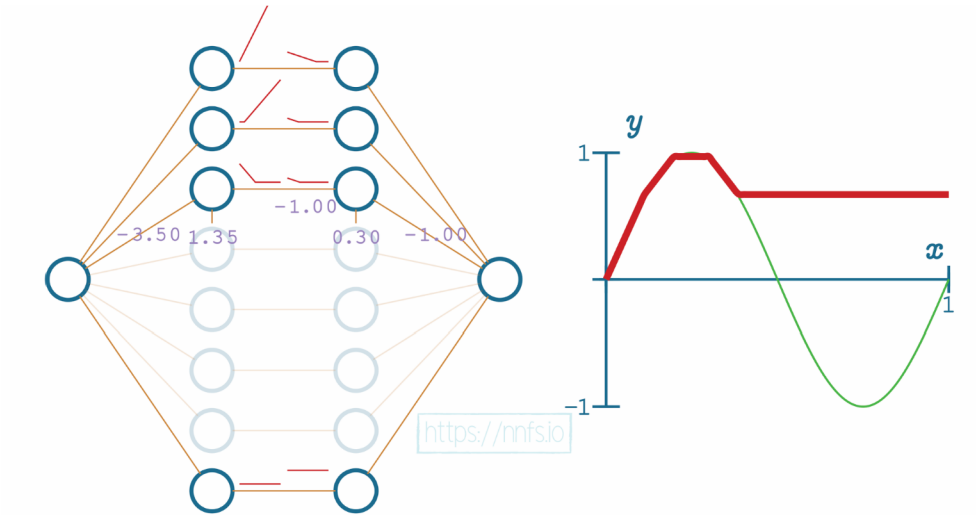


Fig 4.32: Adjusting the 3rd pair of neurons for the next segment.

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

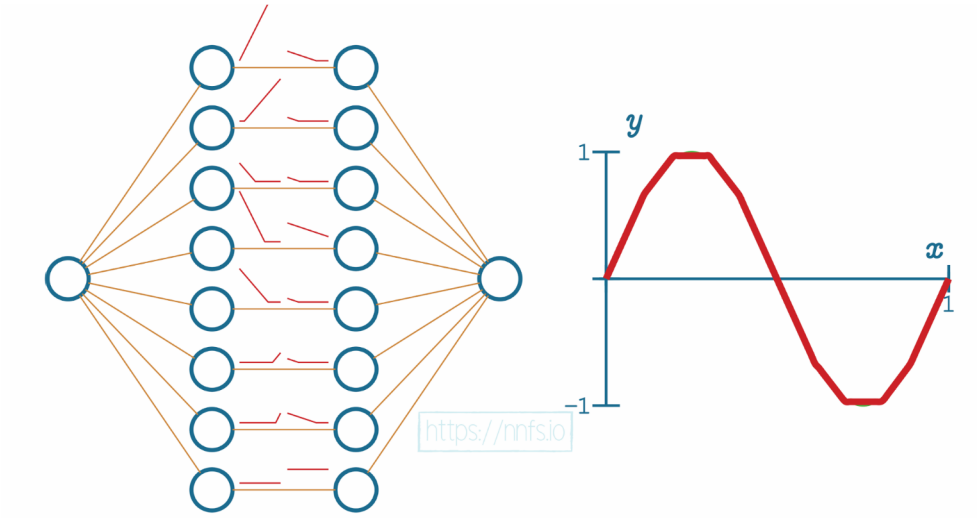


Fig 4.33: The completed process (see anim for all values).

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:

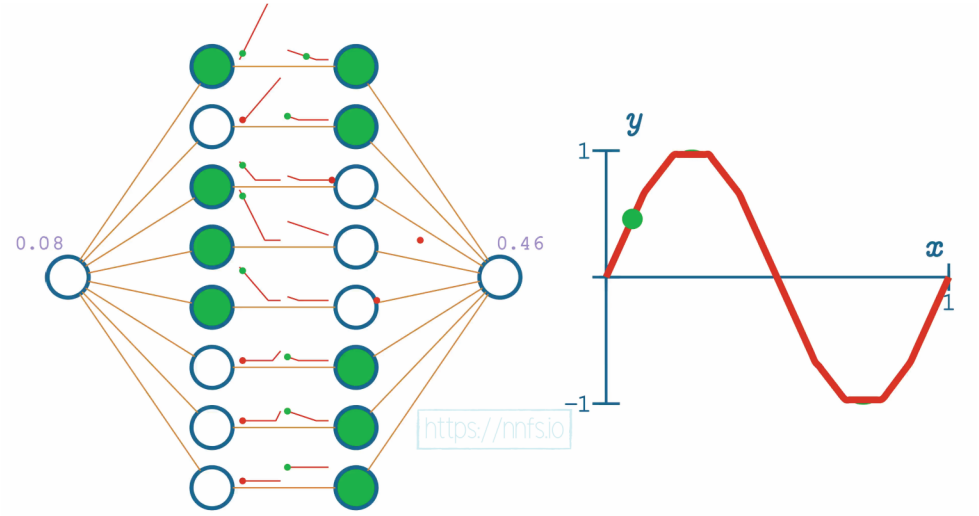


Fig 4.34: Example of data passing through this hand-crafted model.

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:

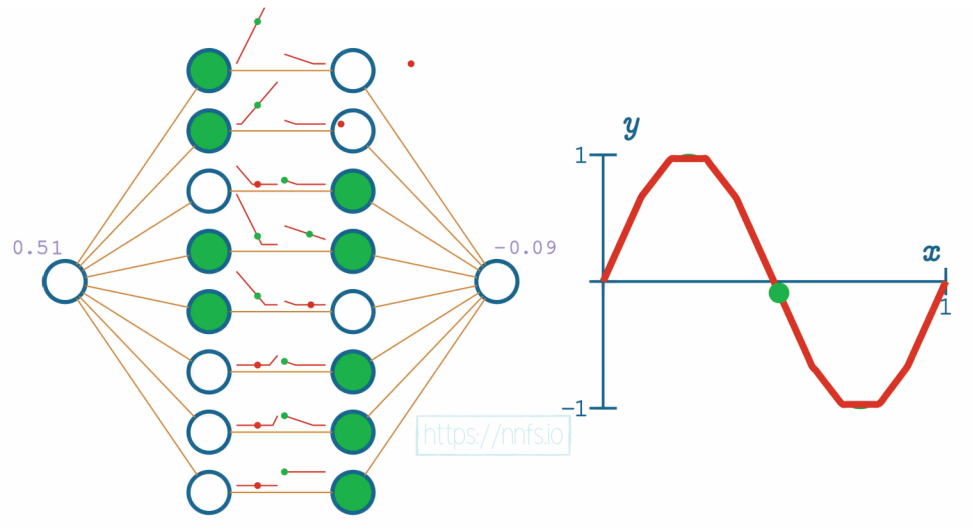


Fig 4.35: Example of data passing through this hand-crafted model.

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:

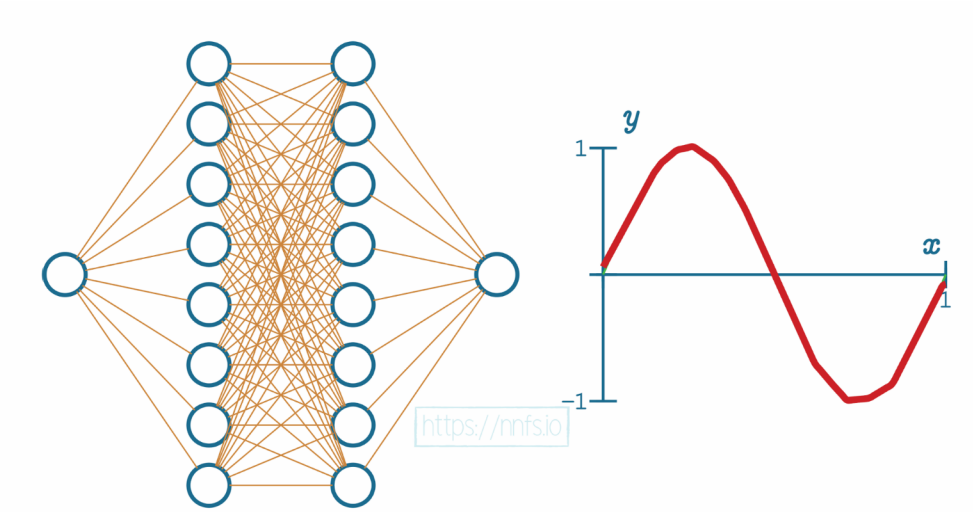


Fig 4.36: Example of fitment after fully-connecting the neurons and using an optimizer.

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:

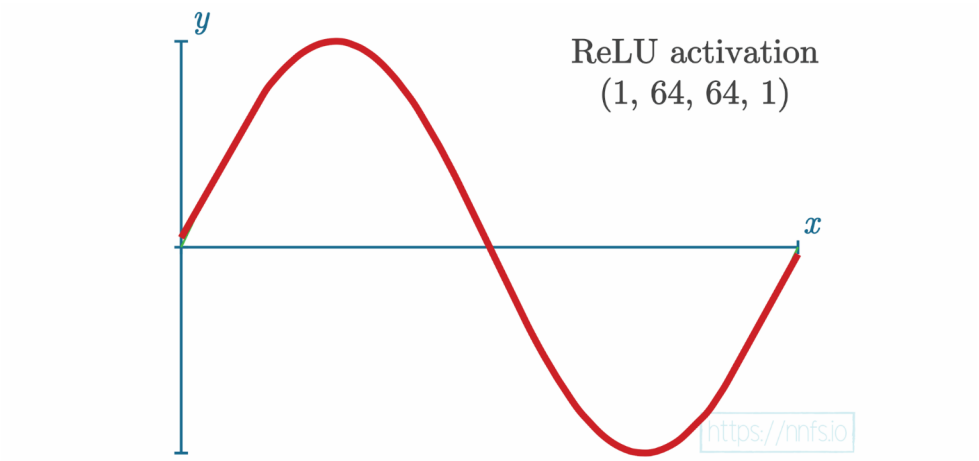


Fig 4.37: Fitment with 2 hidden layers of 64 neurons each, fully connected, with optimizer.

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:

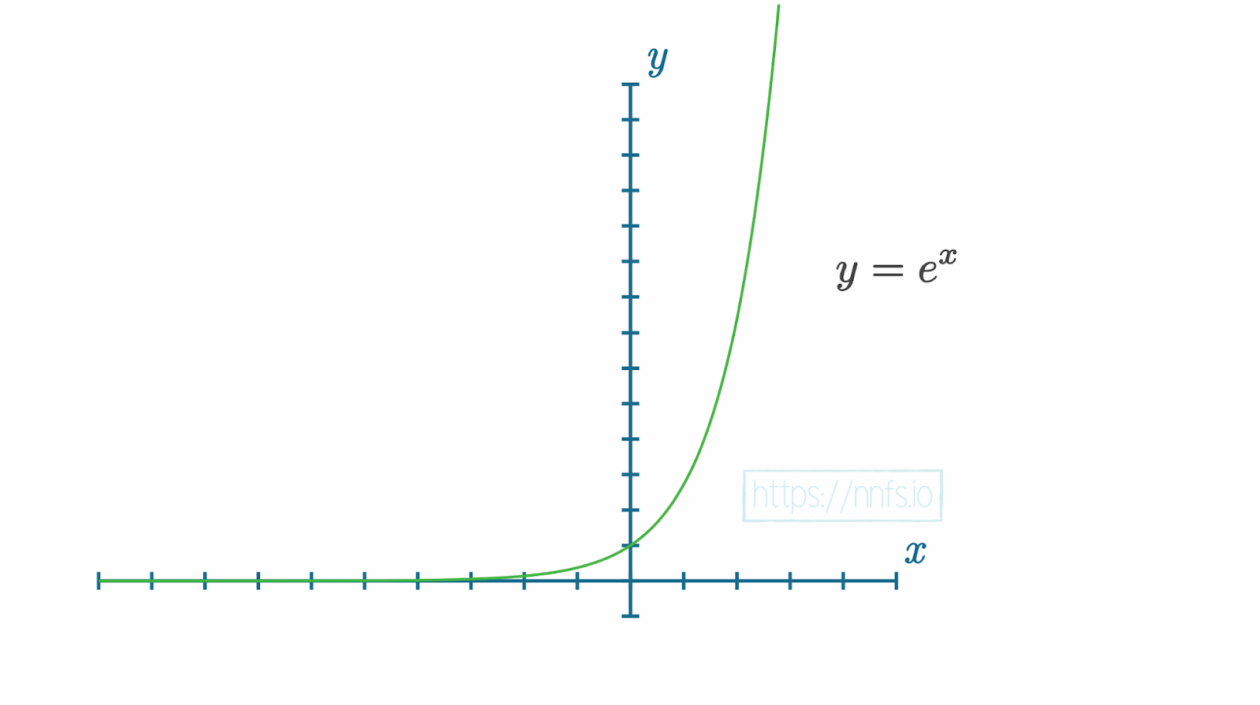


Fig 4.38: Graph of an exponential function.

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 (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:

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

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)
```

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
-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?

```
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
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*:

```
-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*:

```
-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*):

```
-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 loss 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:

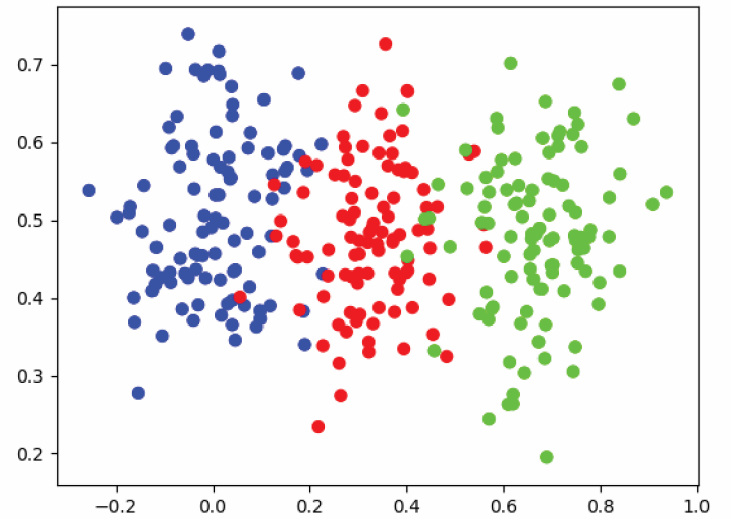


Fig 6.01: "Vertical data" graphed.

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()
```

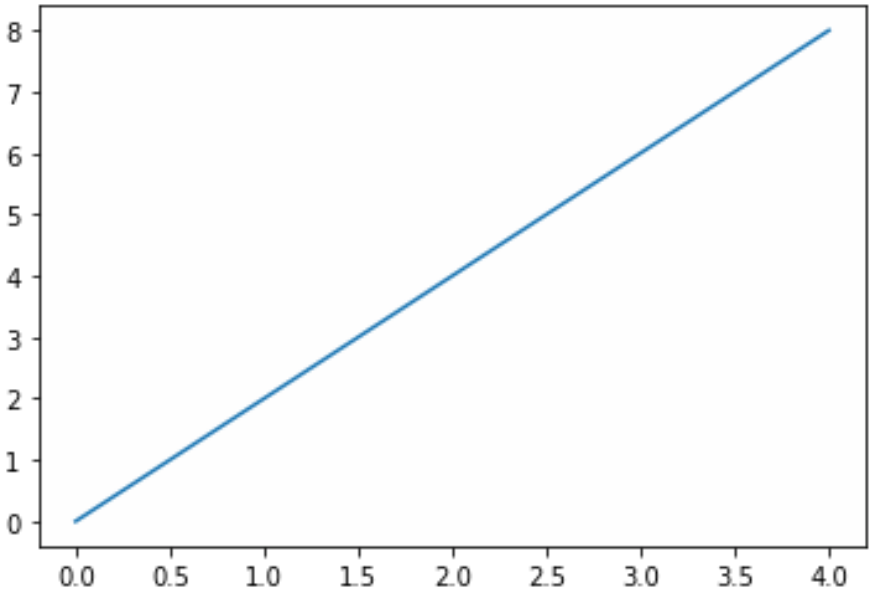


Fig 7.01: Linear function y=2x graphed

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:

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

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

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

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:

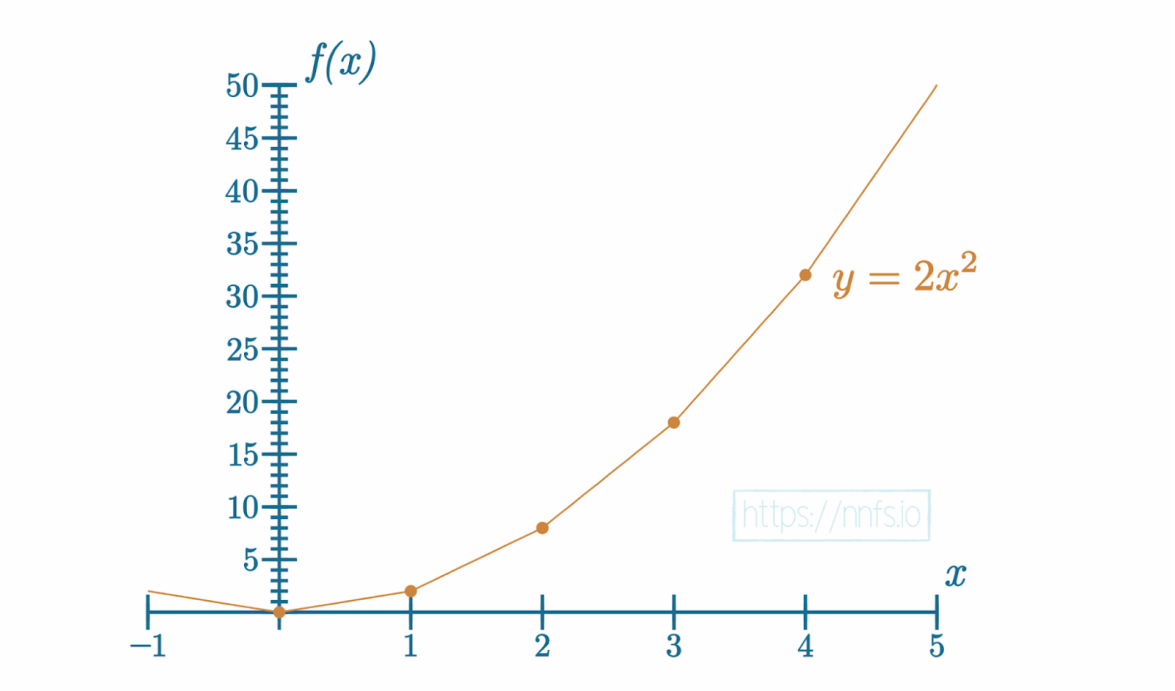


Fig 7.02: Approximation of the parabolic function $y=2x^2$ graphed

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
```

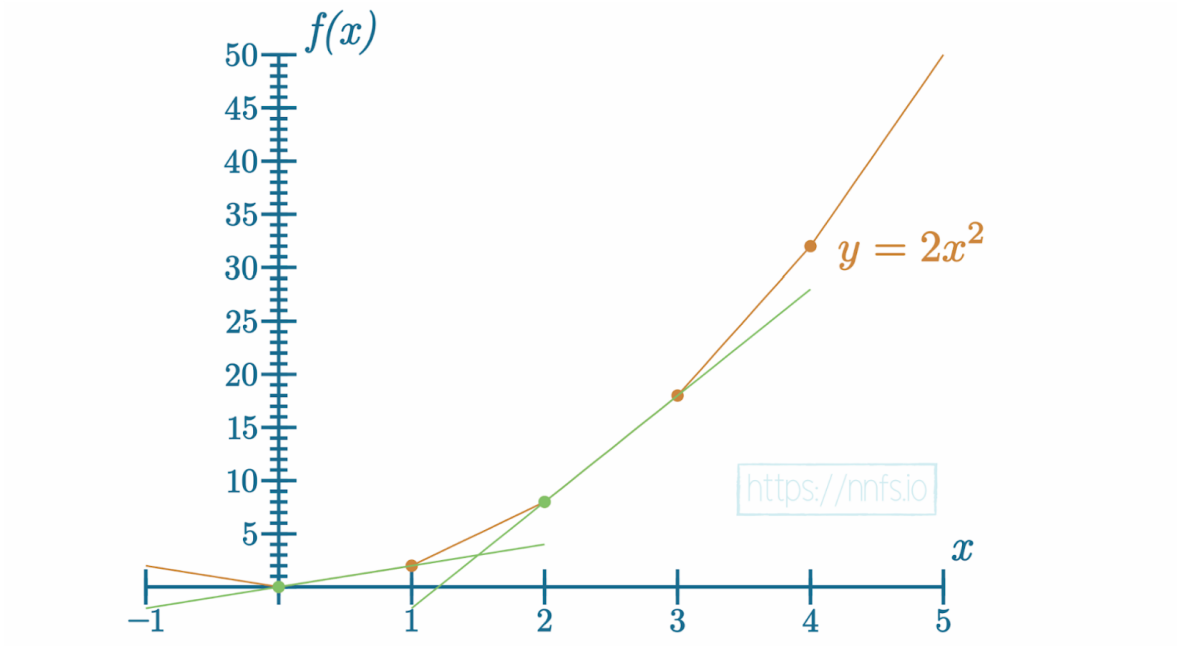


Fig 7.03: Approximation of the parabolic function's example tangents

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:

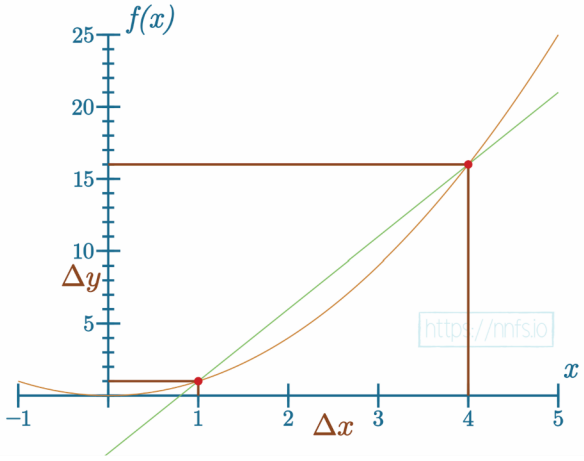


Fig 7.04: Why we want to use 2 points that are sufficiently close — large delta inaccuracy.

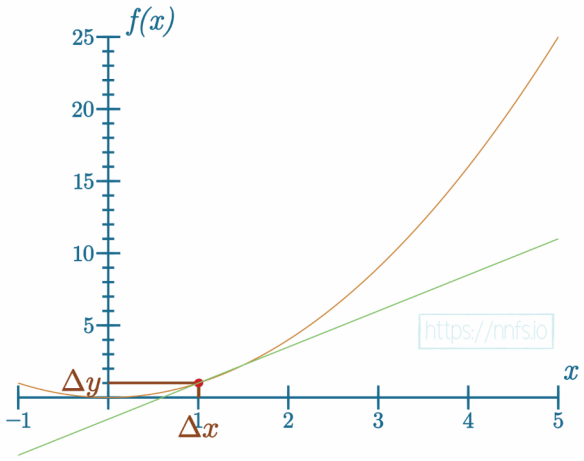


Fig 7.05: Why we want to use 2 points that are sufficiently close — very small delta accuracy.

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()
```

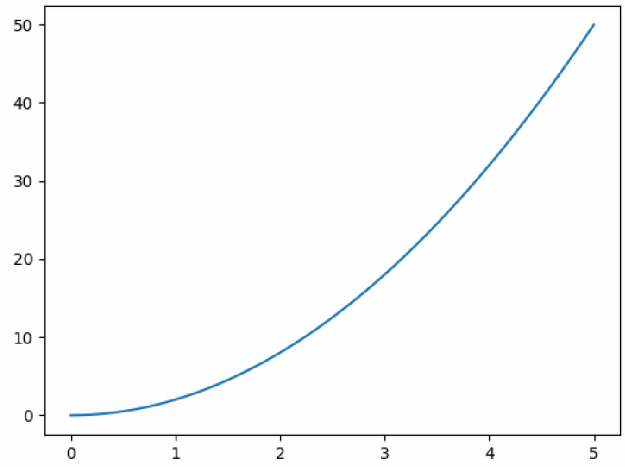


Fig 7.06: Matplotlib output that you should see from graphing $y=2x^2$.

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 *approximate_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:

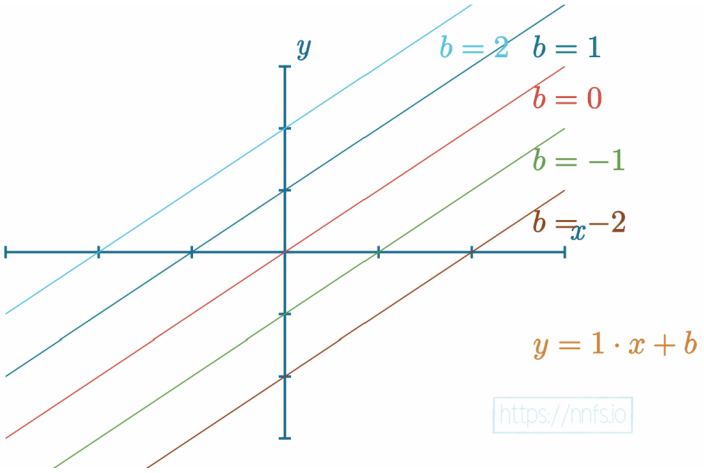


Fig 7.07: Various biases graphed where slope = 1.

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
```

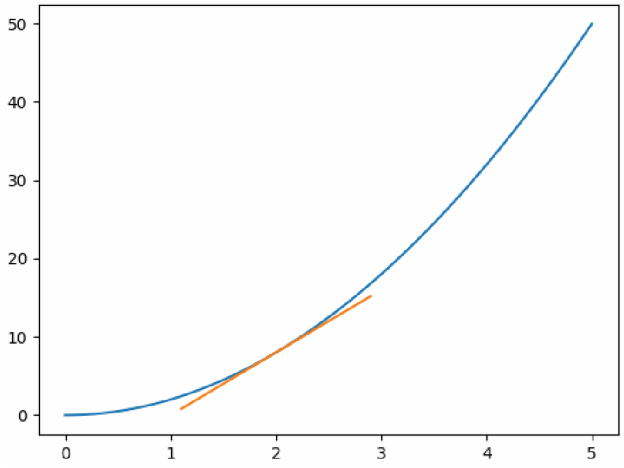


Fig 7.08: Graphed approximate derivative for f(x) where x=2

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
```

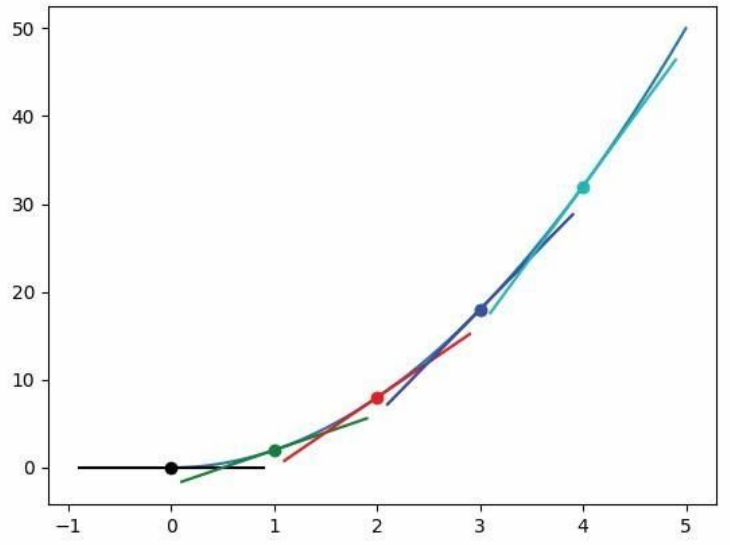


Fig 7.09: Derivative calculated at various points.

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:

$$f(x) = 1 \quad \rightarrow \quad \frac{d}{dx} f(x) = \frac{d}{dx} 1 = 0$$

$$f(x) = 1$$

$$f'(x) = \frac{d}{dx} 1$$

$$f'(x) = 0$$

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:

$$f(x) = x \quad \rightarrow \quad \frac{d}{dx}f(x) = \frac{d}{dx}x = \frac{d}{dx}x^1 = 1 \cdot x^{1-1} = 1 \cdot x^0 = 1 \cdot 1 = 1$$

$$f(x) = x$$
$$f'(x) = \frac{d}{dx}x$$
$$f'(x) = \frac{d}{dx}x^1$$
$$f'(x) = x^{1-1}$$
$$f'(x) = 1 \cdot x^0$$
$$f'(x) = 1 \cdot 1$$
$$f'(x) = 1$$

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?

$$f(x) = 2x \quad \rightarrow \quad \frac{d}{dx}f(x) = \frac{d}{dx}2x = 2 \cdot \frac{d}{dx}x = 2 \cdot 1x^{1-1} = 2 \cdot 1x^{0} = 2 \cdot 1 = 2$$

$$f(x) = 2x$$
$$f'(x) = \frac{d}{dx}2x$$
$$f'(x) = 2 \cdot \frac{d}{dx}x^{1}$$
$$f'(x) = 2 \cdot x^{1-1}$$
$$f'(x) = 2 \cdot 1x^{0}$$
$$f'(x) = 2 \cdot 1 \cdot 1$$
$$f'(x) = 2$$

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?

$$f(x) = 3x^2 \quad \rightarrow \quad \frac{d}{dx}f(x) = \frac{d}{dx}3x^2 = 3 \cdot \frac{d}{dx}x^2 = 3 \cdot 2x^{2-1} = 3 \cdot 2x^1 = 6x$$

$$f(x) = 3x^2$$
$$f'(x) = \frac{d}{dx}3x^2$$
$$f'(x) = 3 \cdot x^{2-1}$$
$$f'(x) = 3 \cdot 2x^1$$
$$f'(x) = 6x$$

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?

$$f(x) = 3x^2 + 5x \quad \rightarrow \quad \frac{d}{dx} f(x) = \frac{d}{dx}[3x^2 + 5x] =$$

$$= \frac{d}{dx} 3x^2 + \frac{d}{dx} 5x^1 =$$

$$= 3 \cdot \frac{d}{dx} x^2 + 5 \cdot \frac{d}{dx} x^1 =$$

$$= 3 \cdot 2x^{2-1} + 5 \cdot 1x^{1-1} =$$

$$= 3 \cdot 2x^1 + 5 \cdot x^0 =$$

$$= 6x + 5$$

$$
\begin{aligned}
f(x) &= 3x^2 + 5x \\
f'(x) &= \frac{d}{dx}[3x^2 + 5x] \\
f'(x) &= \frac{d}{dx}3x^2 + \frac{d}{dx}5x \\
f'(x) &= 3 \cdot x^{2-1} + 5 \cdot x^{1-1} \\
f'(x) &= 3 \cdot 2x^1 + 5 \cdot 1x^0 \\
f'(x) &= 6x + 5
\end{aligned}
$$

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:

$$f(x) = 5x^5 + 4x^3 - 5 \quad \rightarrow \quad \frac{d}{dx}f(x) = \frac{d}{dx}[5x^5 + 4x^3 - 5] =$$

$$= \frac{d}{dx}5x^5 + \frac{d}{dx}4x^3 - \frac{d}{dx}5 =$$

$$= 5 \cdot \frac{d}{dx}x^5 + 4 \cdot \frac{d}{dx}x^3 - \frac{d}{dx}5 =$$

$$= 5 \cdot 5x^{5-1} + 4 \cdot 3x^{3-1} - 0 =$$

$$= 5 \cdot 5x^4 + 4 \cdot 3x^2 =$$

$$= 25x^4 + 12x^2$$

$$f(x) = 5x^5 + 4x^3 - 5$$

$$f'(x) = \frac{d}{dx}[5x^5 + 4x^3 - 5]$$

$$f'(x) = \frac{d}{dx}5x^5 + \frac{d}{dx}4x^3 - \frac{d}{dx}5$$

$$f'(x) = 5 \cdot x^{5-1} + 4 \cdot x^{3-1} - 0$$

$$f'(x) = 5 \cdot 5x^4 + 4 \cdot 3x^2$$

$$f'(x) = 25x^4 + 12x^2$$

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.

$$f(x) = x^3 + 2x^2 - 5x + 7 \quad \rightarrow \quad \frac{d}{dx}f(x) = \frac{d}{dx}[x^3 + 2x^2 - 5x + 7] =$$

$$= \frac{d}{dx}x^3 + \frac{d}{dx}2x^2 - \frac{d}{dx}5x + \frac{d}{dx}7 =$$

$$= \frac{d}{dx}x^3 + 2 \cdot \frac{d}{dx}x^2 - 5 \cdot \frac{d}{dx}x + \frac{d}{dx}7 =$$

$$= 3x^{3-1} + 2 \cdot 2x^{2-1} - 5 \cdot 1x^{1-1} + 0 =$$

$$= 3x^2 + 2 \cdot 2x^1 - 5 \cdot 1 + 0 =$$

$$= 3x^2 + 4x - 5$$

$$f(x) = x^3 + 2x^2 - 5x + 7$$

$$f'(x) = \frac{d}{dx}[x^3 + 2x^2 - 5x + 7]$$

$$f'(x) = \frac{d}{dx}x^3 + \frac{d}{dx}2x^2 - \frac{d}{dx}5x + \frac{d}{dx}7$$

$$f'(x) = x^{3-1} + 2 \cdot x^{2-1} - 5 \cdot x^{1-1} + 0$$

$$f'(x) = 3x^2 + 2 \cdot 2x^1 - 5 \cdot 1x^0$$

$$f'(x) = 3x^2 + 4x - 5$$

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:

$$f(x,y,z) = 3x^3z - y^2 + 5z + 2yz \quad \rightarrow$$
$$\frac{\partial}{\partial x} f(x,y,z) = \frac{\partial}{\partial x}[3x^3z - y^2 + 5z + 2yz] =$$
$$= \frac{\partial}{\partial x}3x^3z - \frac{\partial}{\partial x}y^2 + \frac{\partial}{\partial x}5z + \frac{\partial}{\partial x}2yz =$$
$$= 3z \cdot \frac{\partial}{\partial x}x^3 - \frac{\partial}{\partial x}y^2 + 5 \cdot \frac{\partial}{\partial x}z + 2 \cdot \frac{\partial}{\partial x}yz =$$
$$= 3z \cdot 3x^2 - 0 + 5 \cdot 0 + 2 \cdot 0 = 9x^2z$$

$$f(x,y,z) = 3x^3z - y^2 + 5z + 2yz \quad \rightarrow$$

$$\frac{\partial}{\partial y} f(x,y,z) = \frac{\partial}{\partial y}[3x^3z - y^2 + 5z + 2yz] =$$

$$= \frac{\partial}{\partial y} 3x^3z - \frac{\partial}{\partial y} y^2 + \frac{\partial}{\partial y} 5z + \frac{\partial}{\partial y} 2yz =$$

$$= 3 \cdot \frac{\partial}{\partial y} x^3z - \frac{\partial}{\partial y} y^2 + 5 \cdot \frac{\partial}{\partial y} z + 2z \cdot \frac{\partial}{\partial y} y =$$

$$= 3 \cdot 0 - 2y + 5 \cdot 0 + 2z \cdot 1 = -2y + 2z$$

$$f(x,y,z) = 3x^3z - y^2 + 5z + 2yz \quad \rightarrow$$

$$\frac{\partial}{\partial z} f(x,y,z) = \frac{\partial}{\partial z}[3x^3z - y^2 + 5z + 2yz] =$$

$$= \frac{\partial}{\partial z} 3x^3z - \frac{\partial}{\partial z} y^2 + \frac{\partial}{\partial z} 5z + \frac{\partial}{\partial z} 2yz =$$

$$= 3x^3 \cdot \frac{\partial}{\partial z} z - \frac{\partial}{\partial z} y^2 + 5 \cdot \frac{\partial}{\partial z} z + 2y \cdot \frac{\partial}{\partial z} z =$$

$$= 3x^3 \cdot 1 - 0 + 5 \cdot 1 + 2y \cdot 1 = 3x^3 + 5 + 2y$$

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 about the **gradient** - it's a vector of all of the possible partial derivatives of the function, and we denote it using the ∇ — 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:

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

```
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
        )
    )
)
```


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]`:

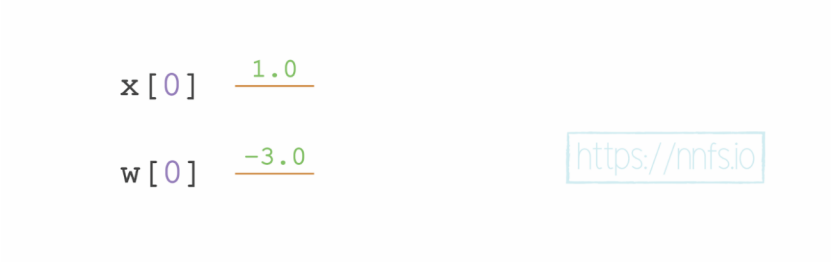


Fig 9.01: Beginning a forward pass with the first input and weight.

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:

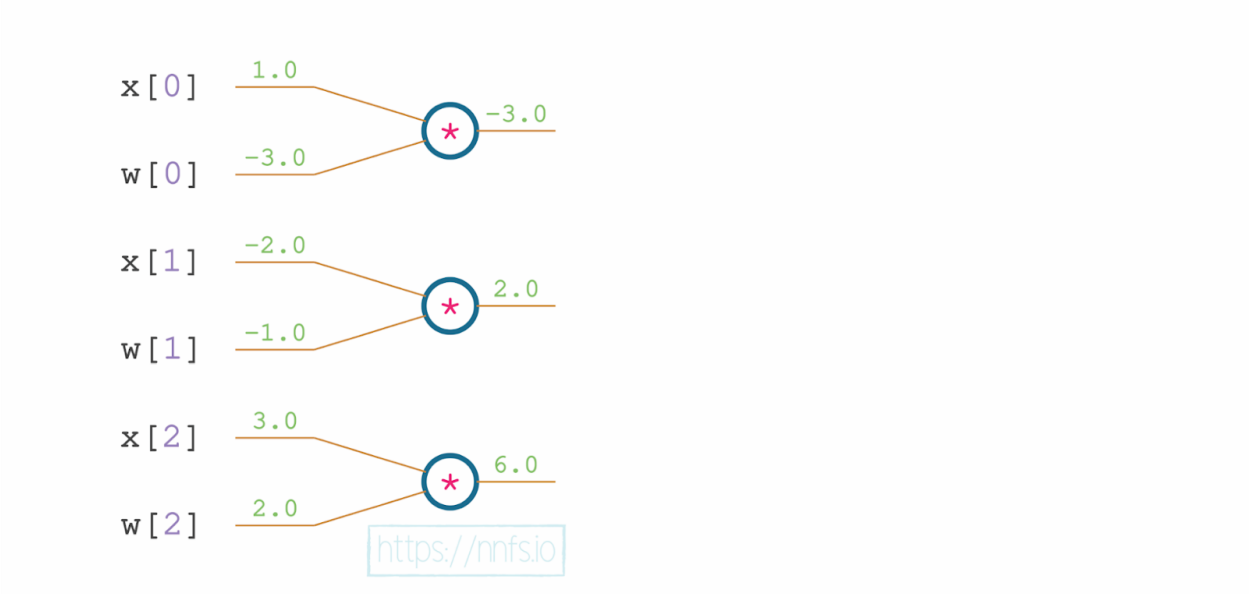


Fig 9.03: Input and weight multiplication of all of the inputs.

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
```

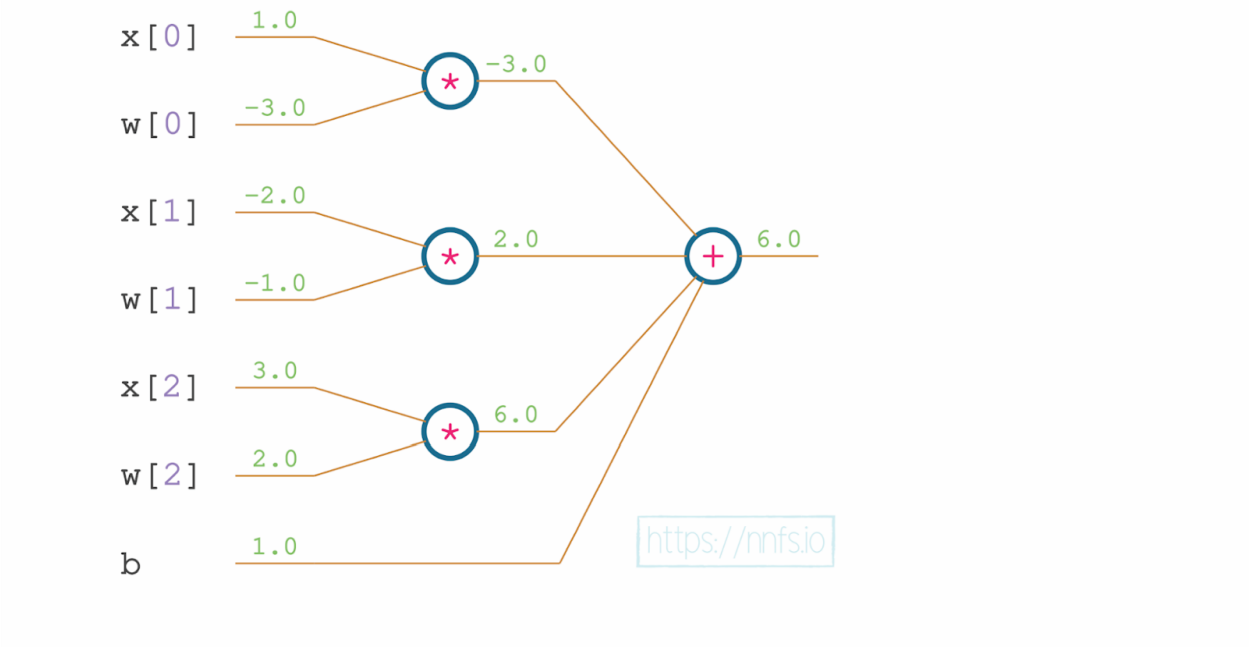


Fig 9.04: Weighted inputs and bias addition.

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
```

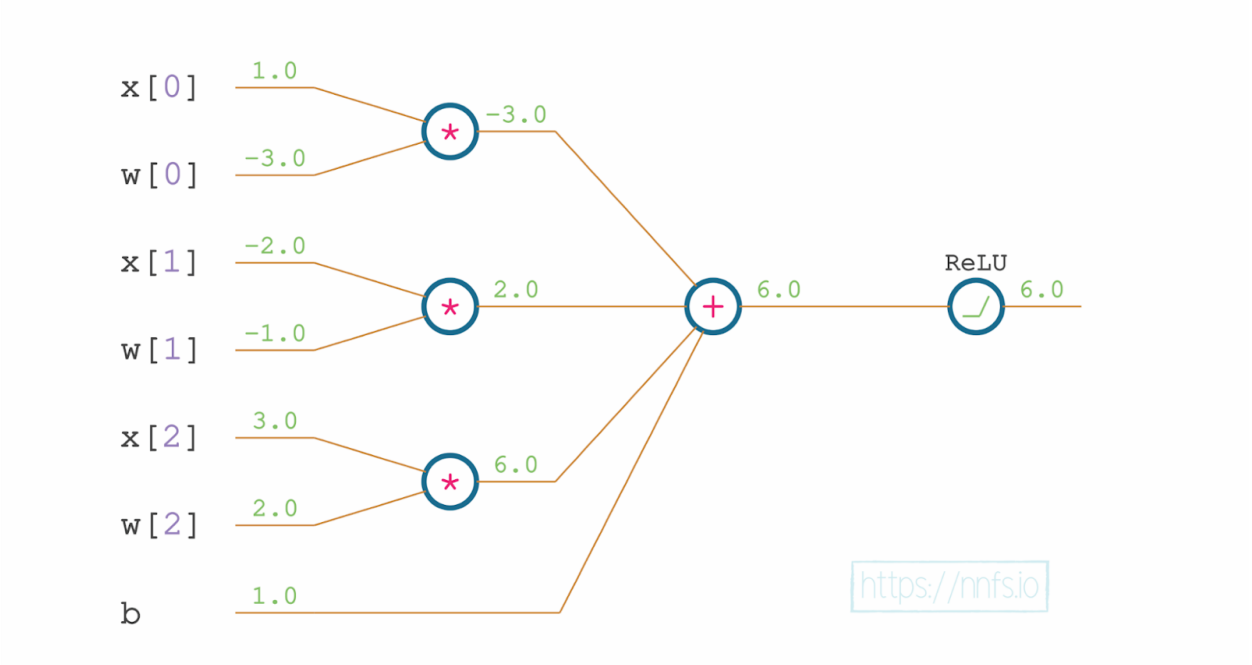


Fig 9.05: ReLU activation applied to the neuron output.

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:

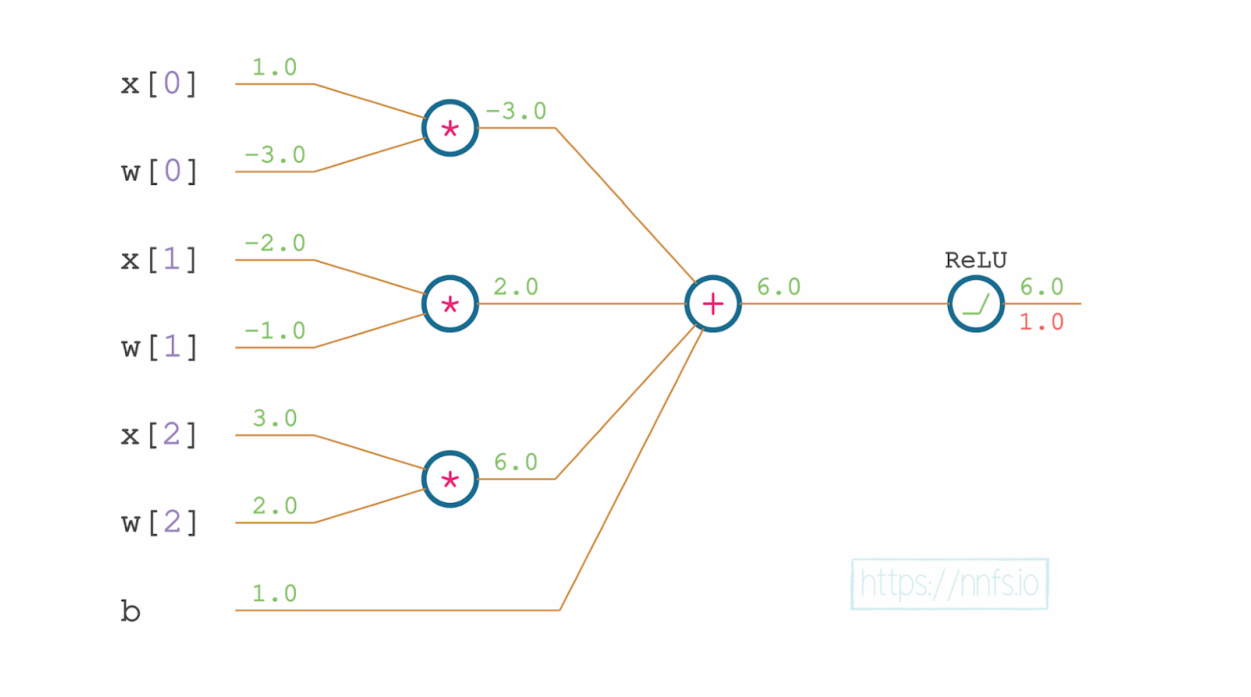


Fig 9.06: Initial gradient (received during backpropagation).

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:

```
relu_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
```

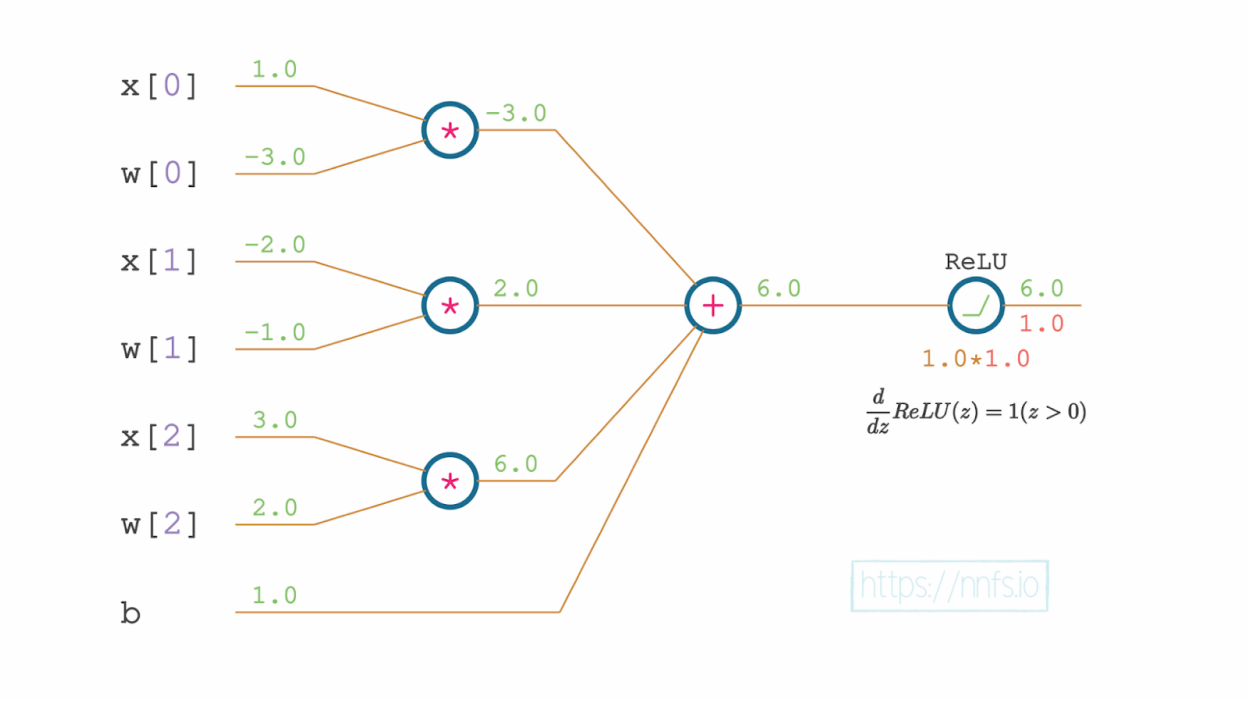


Fig 9.07: Derivative of the ReLU function and chain rule.

This results with the derivative of *1*:

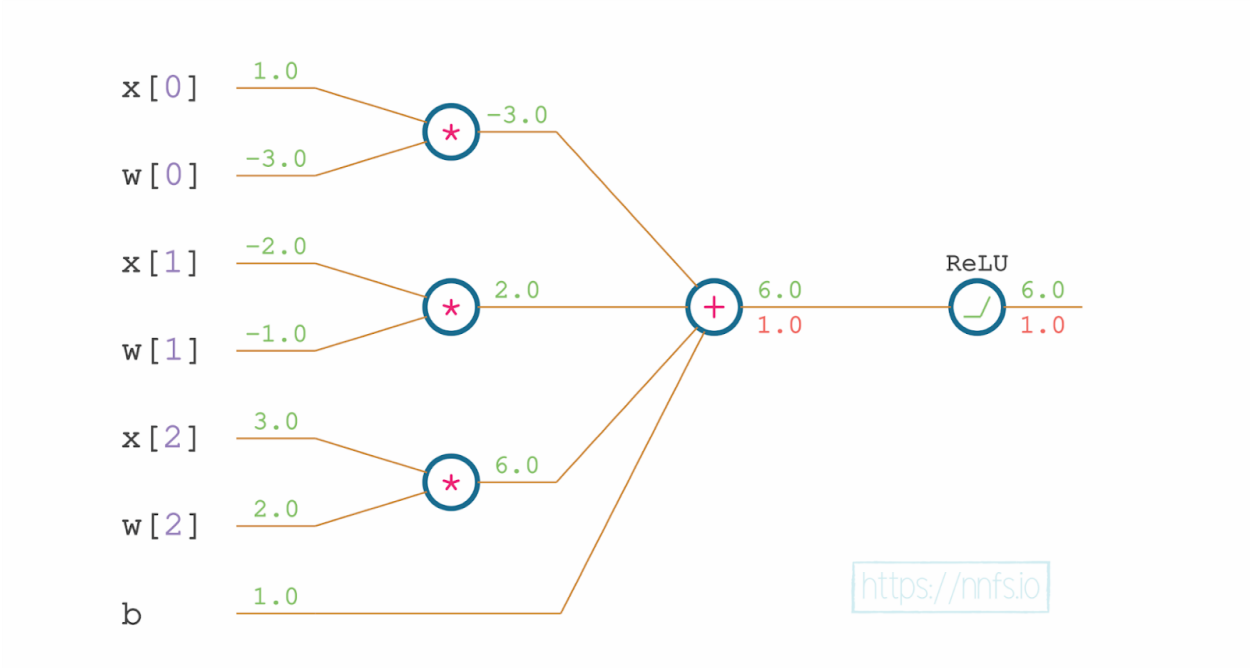


Fig 9.08: ReLU and chain rule gradient.

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
```

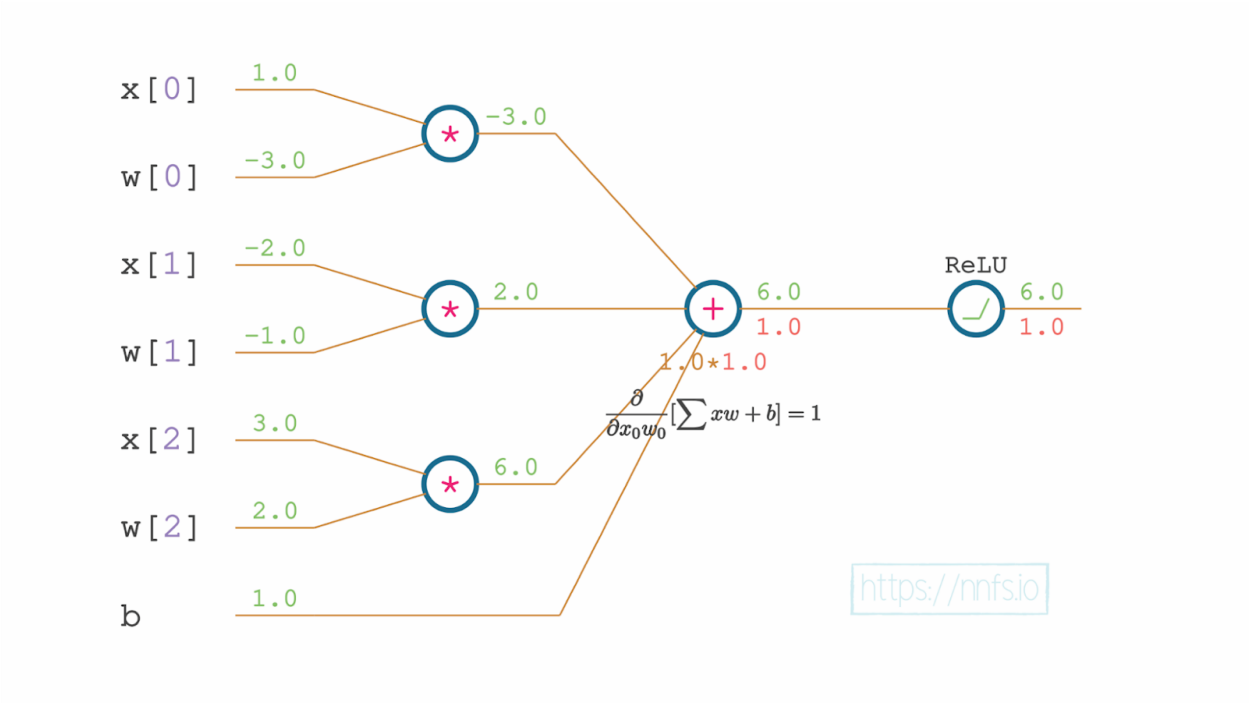


Fig 9.09: Partial derivative of the sum function w.r.t. the first weighted input; the chain rule.

This results with a partial derivative of 1 again:

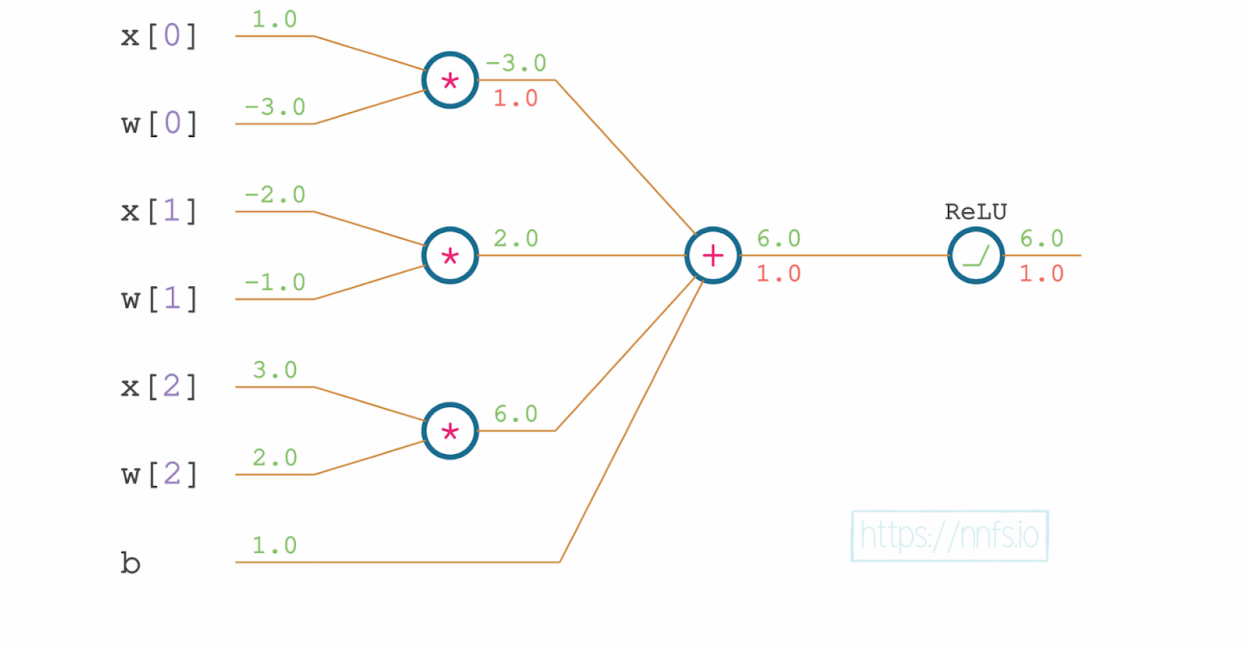


Fig 9.10: The sum and chain rule gradient (for the first weighted input).

We can then perform the same operation with the next weighed input:

```
dsum_dxw1 = 1
drelu_dxw1 = drelu_dz * dsum_dxw1
```

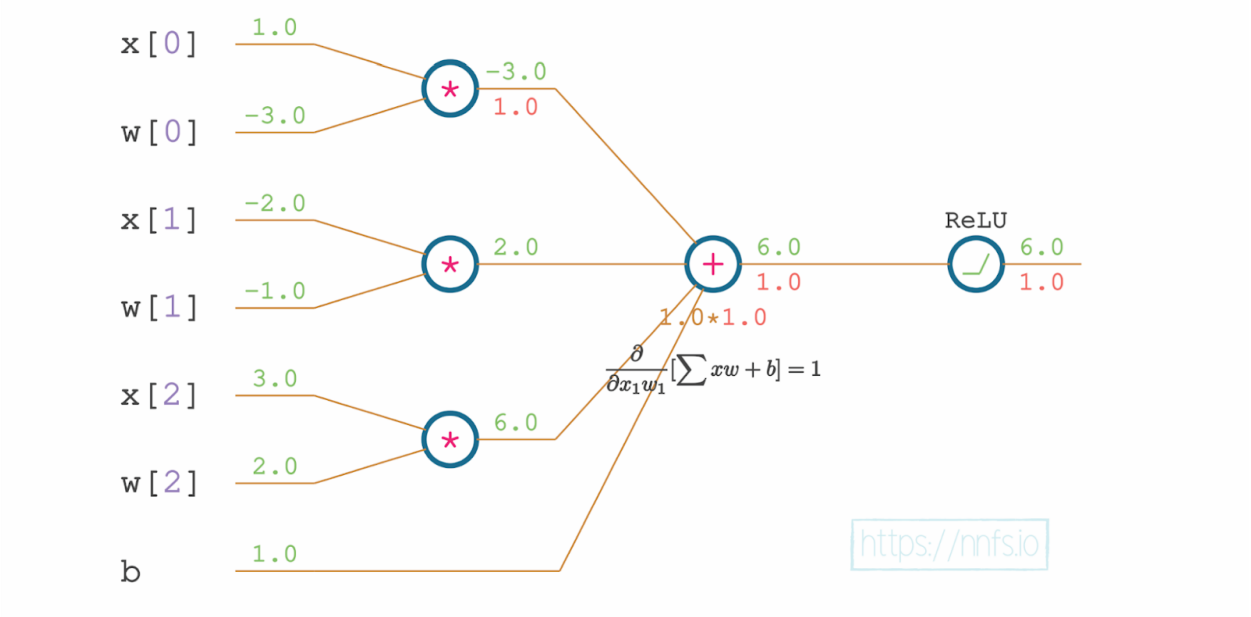


Fig 9.11: Partial derivative of the sum function w.r.t. the second weighted input; the chain rule.

Which results with the next calculated partial derivative:

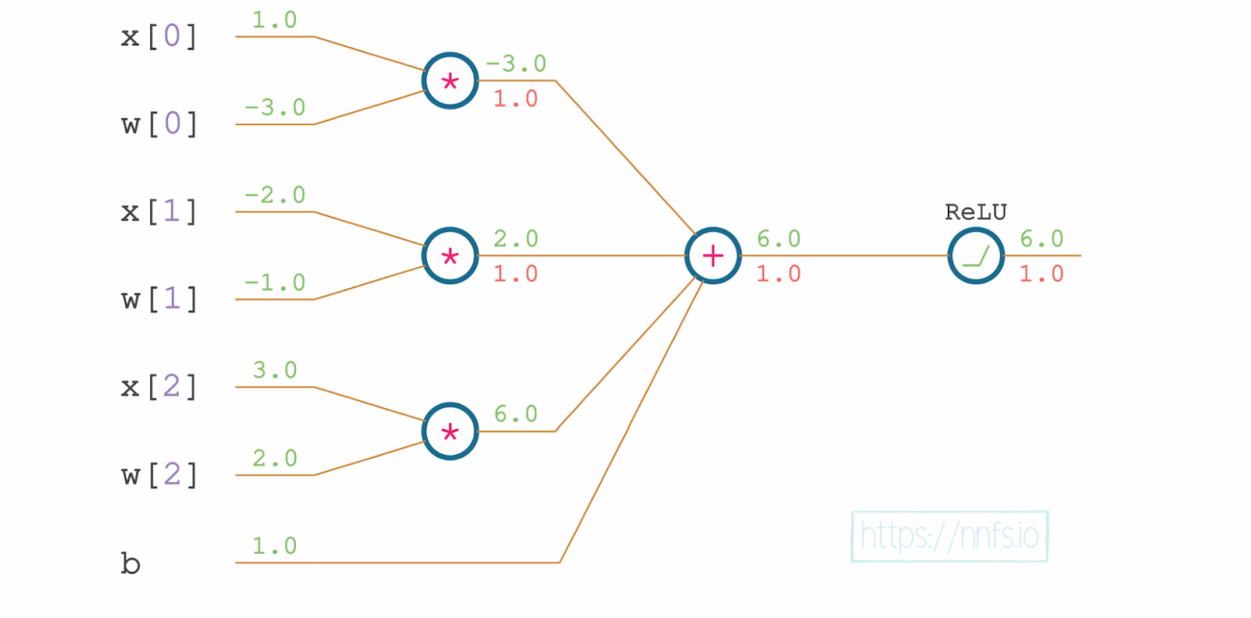


Fig 9.12: The sum and chain rule gradient (for the second weighted input).

And the last weighted input:

```
dsum_dxw2 = 1
drelu_dxw2 = drelu_dz * dsum_dxw2
```

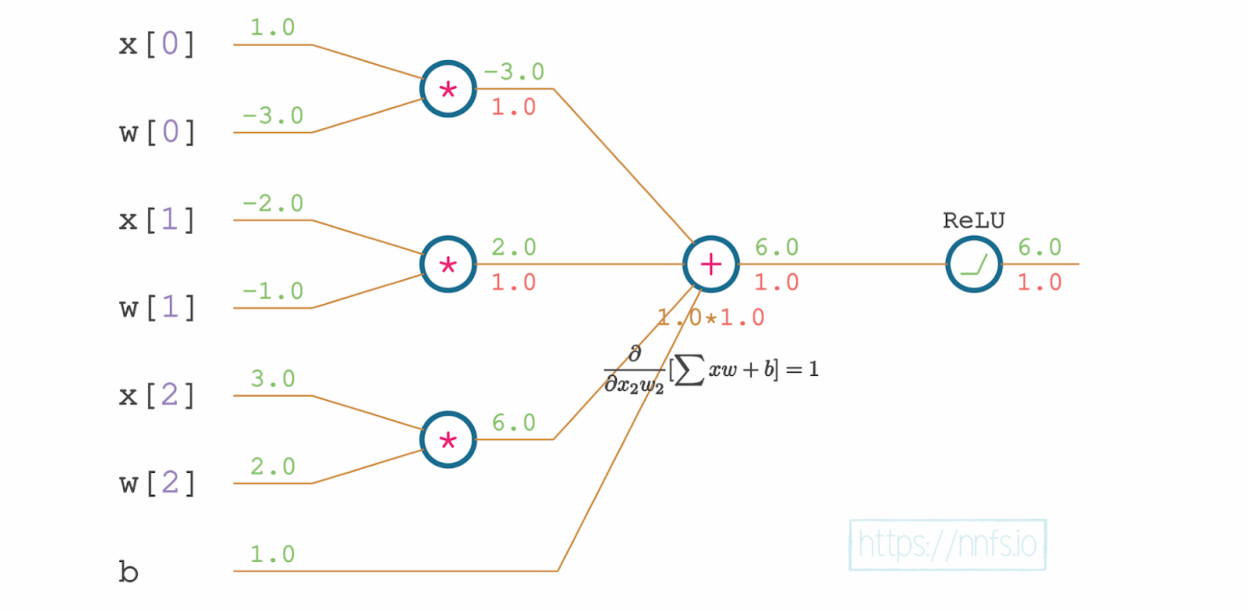


Fig 9.13: Partial derivative of the sum function w.r.t. the third weighted input; the chain rule.

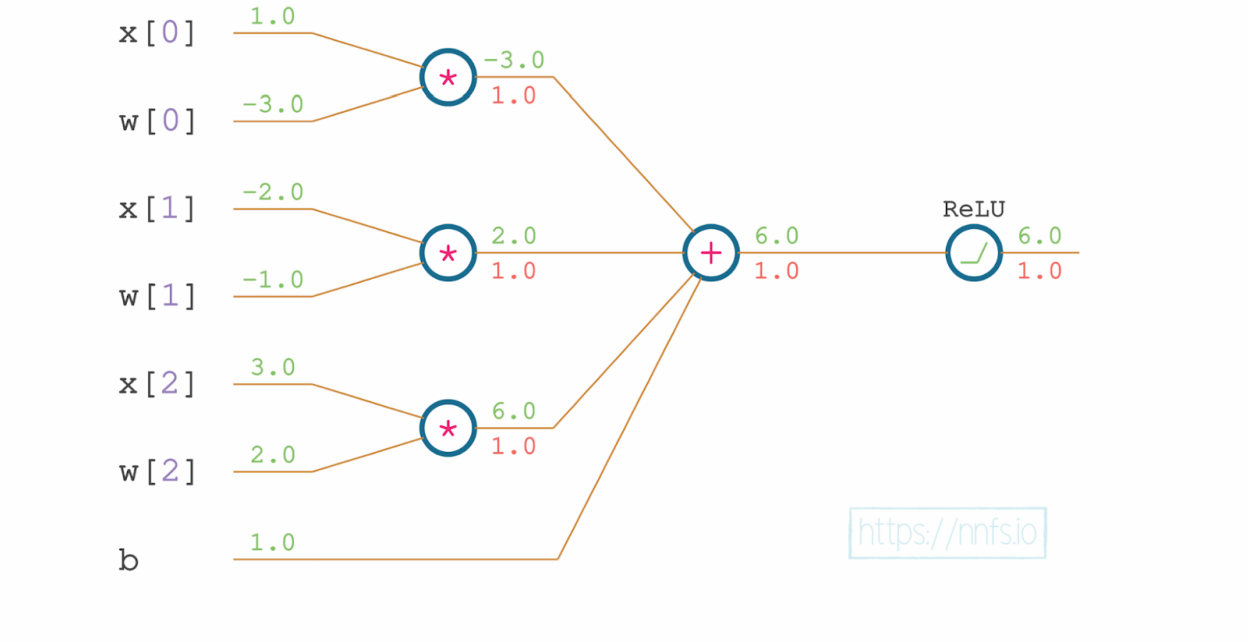


Fig 9.14: The sum and chain rule gradient (for the third weighted input).

Then the bias:

```
dsum_db = 1
drelu_db = drelu_dz * dsum_db
```

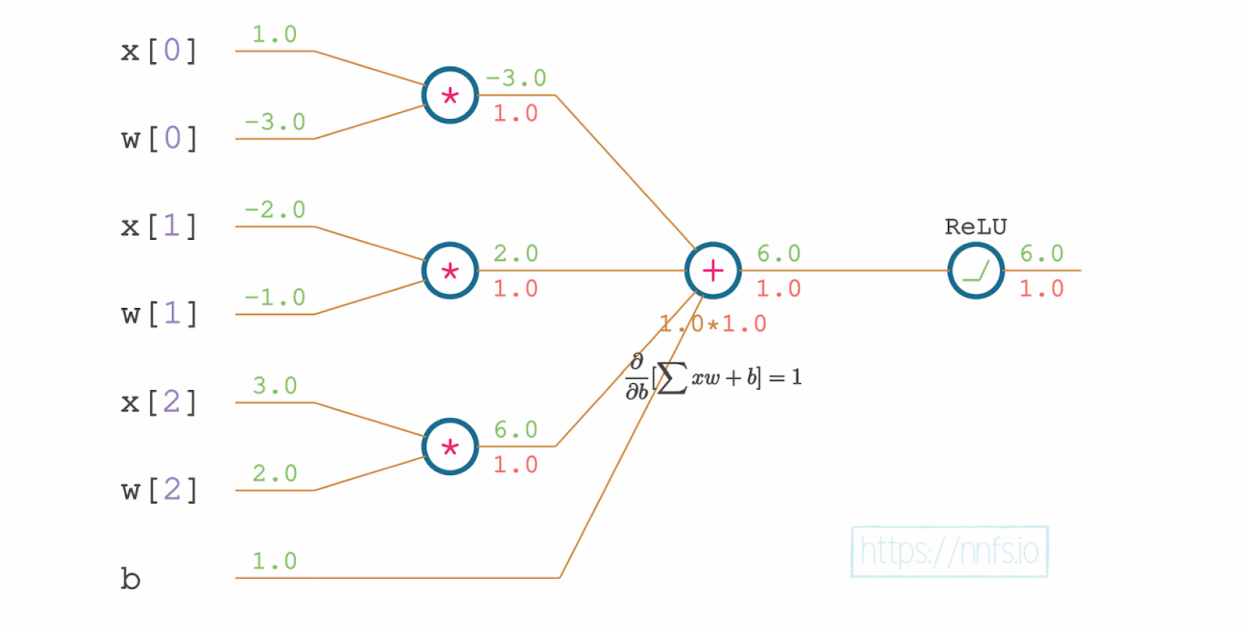


Fig 9.15: Partial derivative of the sum function w.r.t. the bias; the chain rule.

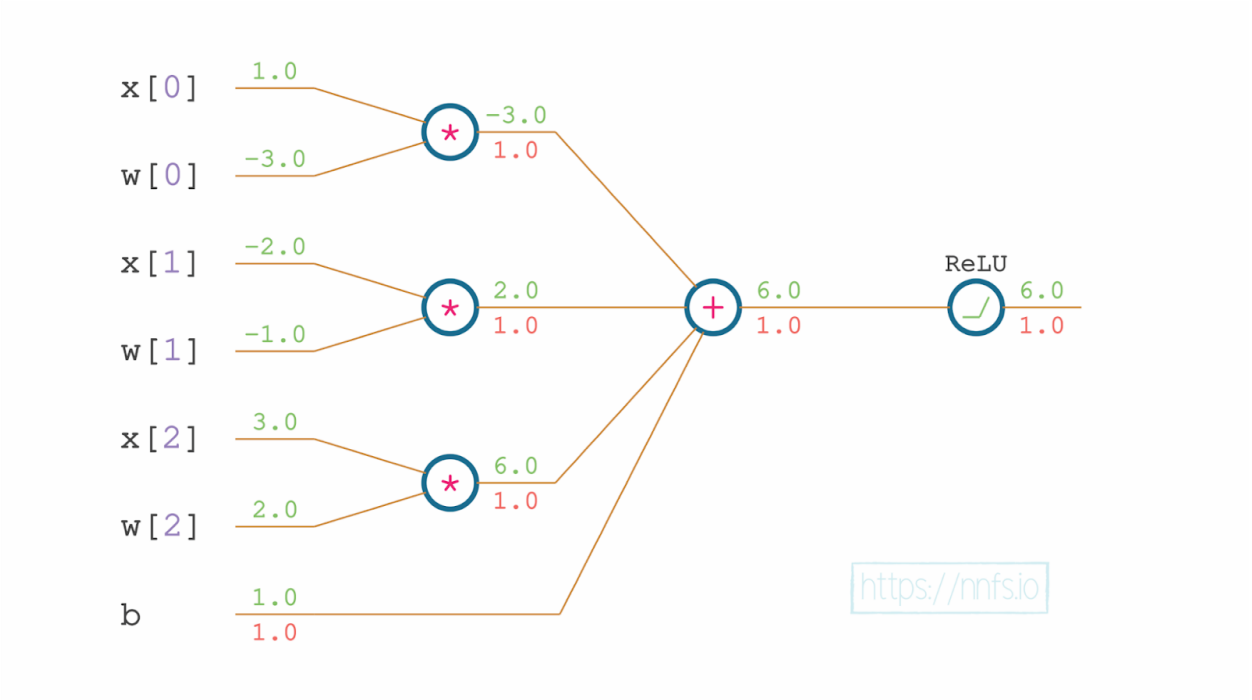


Fig 9.16: The sum and chain rule gradient (for the bias).

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
```

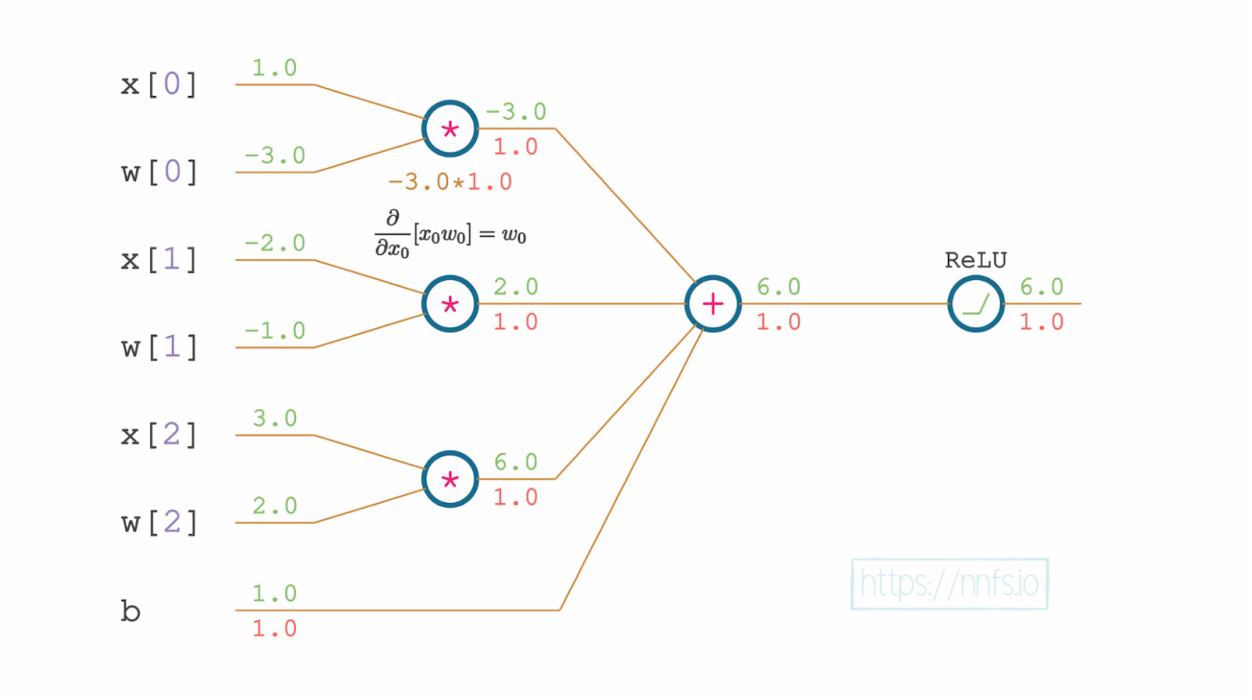


Fig 9.17: Partial derivative of the multiplication function w.r.t. the first input; the chain rule.

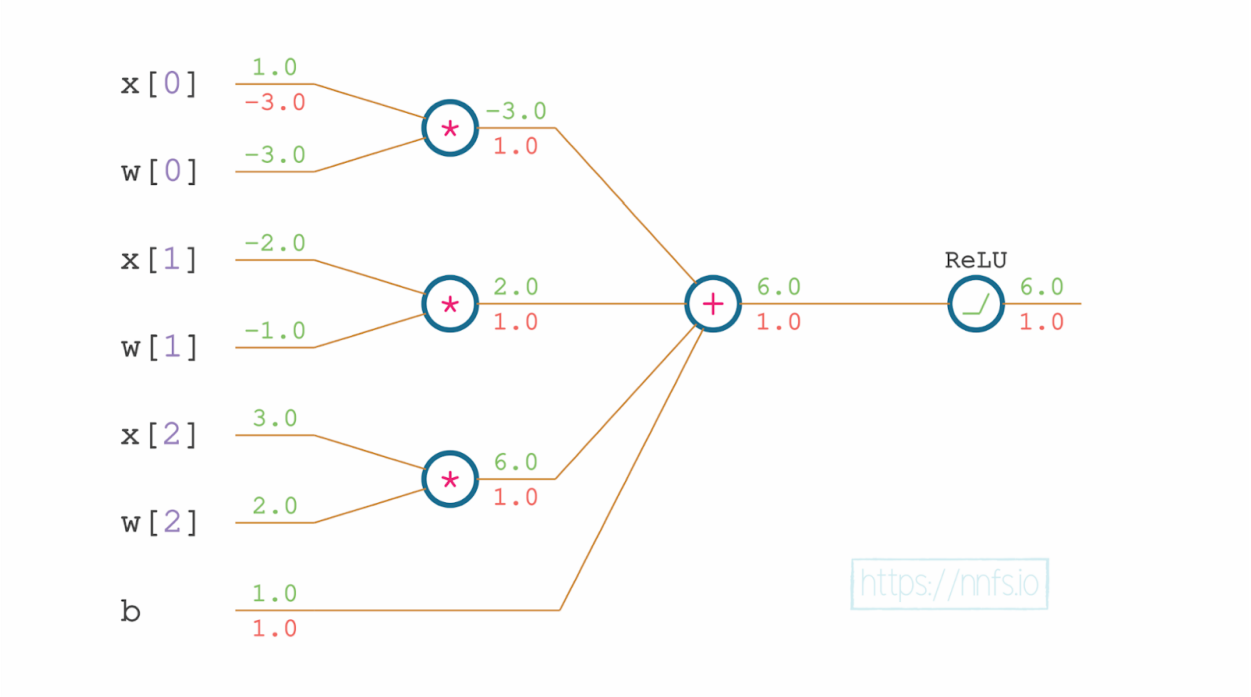


Fig 9.18: The multiplication and chain rule gradient (for the first input).

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
```

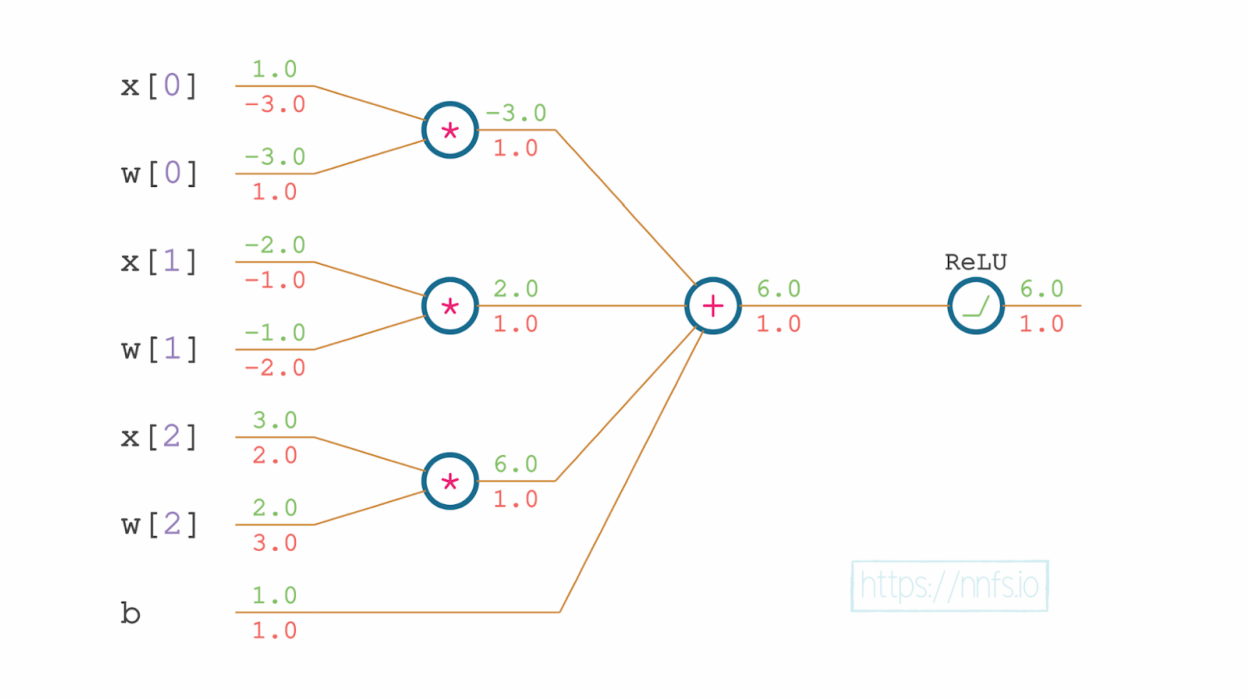


Fig 9.19: Complete backpropagation graph.

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]
```

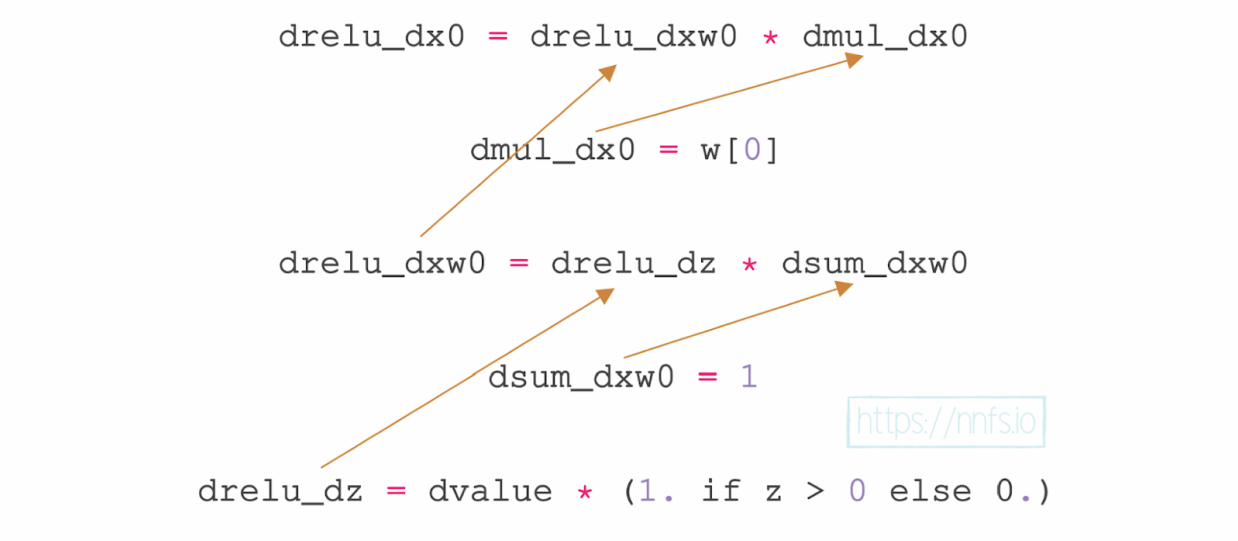


Fig 9.20: How to apply the chain rule for the partial derivative of ReLU w.r.t. first input

```
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 of given input
# and multiply by the passed in gradient for this 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]
```

`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
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
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:

```
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
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:

```
np.eye(5)[1]
```

```
>>>
array([0., 1., 0., 0., 0.])
```

```
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*):

$$\frac{\partial}{\partial z_{i,k}} \sum_{l=1}^{L} e^{z_{i,l}} = \frac{\partial}{\partial z_{i,k}} e^{z_{i,1}} + \frac{\partial}{\partial z_{i,k}} e^{z_{i,2}} + ... + \frac{\partial}{\partial z_{i,k}} e^{z_{i,k}} + ... + \frac{\partial}{\partial z_{i,k}} e^{z_{i,L-1}} + \frac{\partial}{\partial z_{i,k}} e^{z_{i,L}}$$

$$= 0 + 0 + ... + e^{z_{i,k}} + ... + 0 + 0 = e^{z_{i,k}}$$

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 = 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 and
            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, $\hat{y}_{i,j}$, 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-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:

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]])
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 and
            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:

```
import numpy as np
import nnfs

nnfs.init()

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:

```
import numpy as np
from timeit import timeit
import nnfs

nnfs.init()

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 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 and
            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:

```
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.

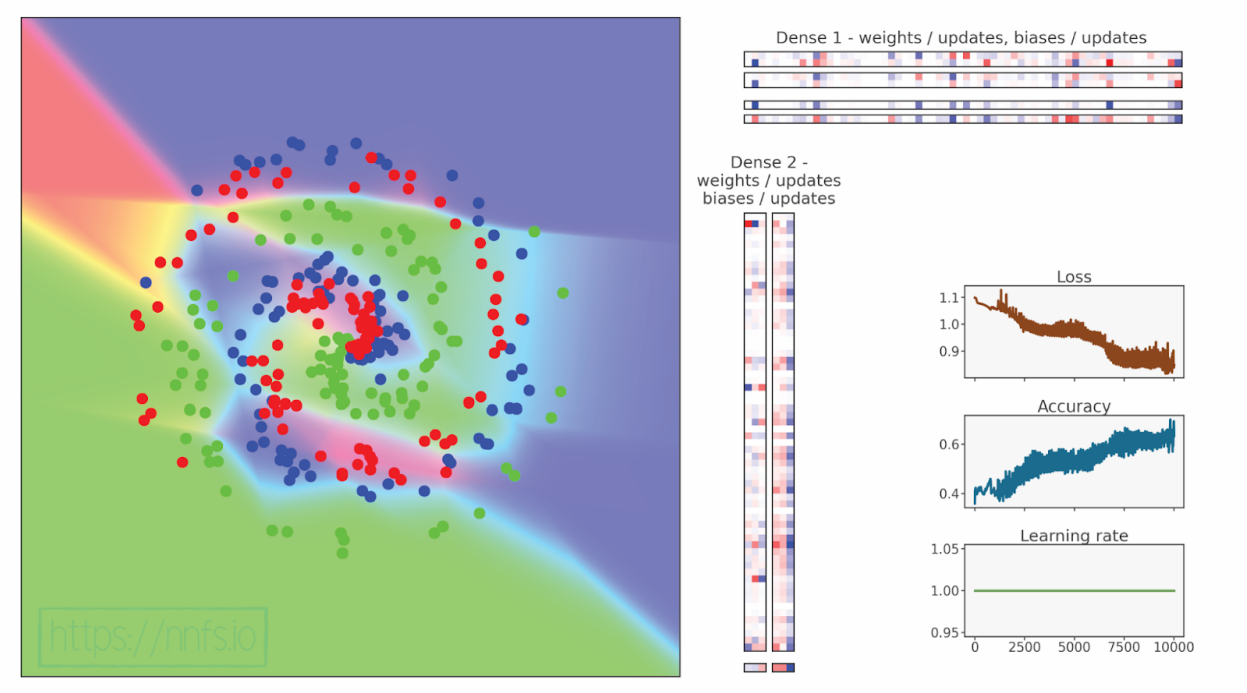


Fig 10.01: Model training with Stochastic Gradient Descent optimizer.

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:

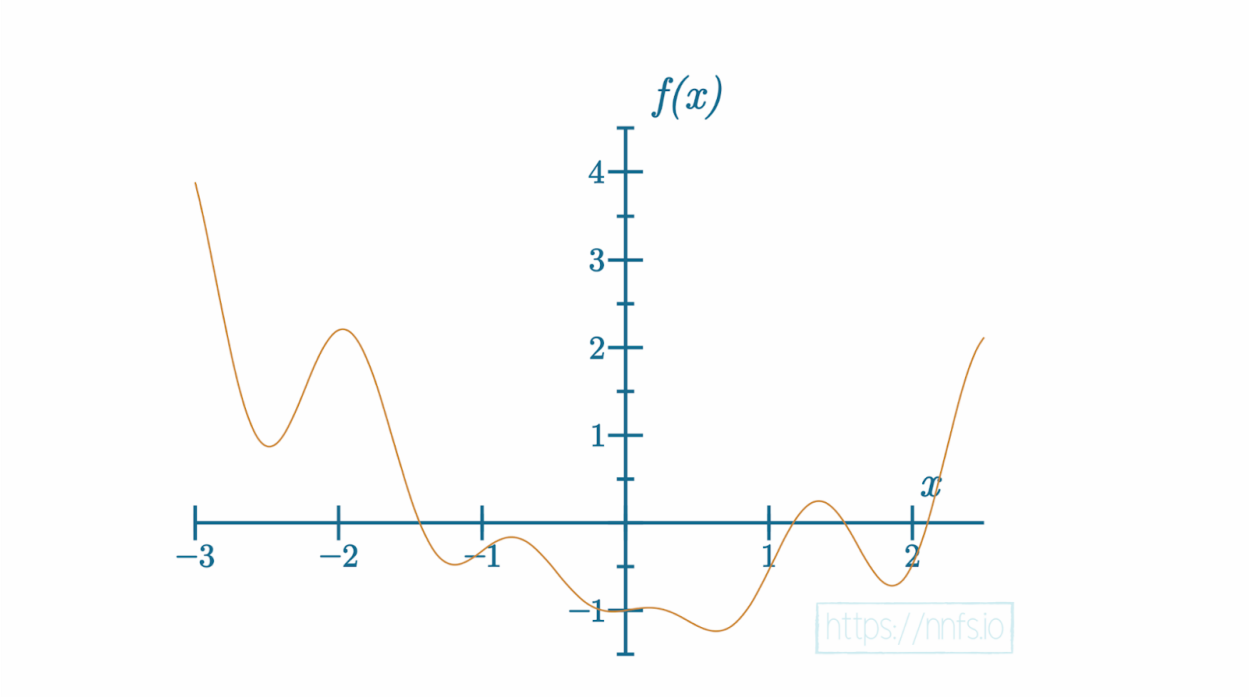


Fig 10.02: Example function to minimize the output.

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:

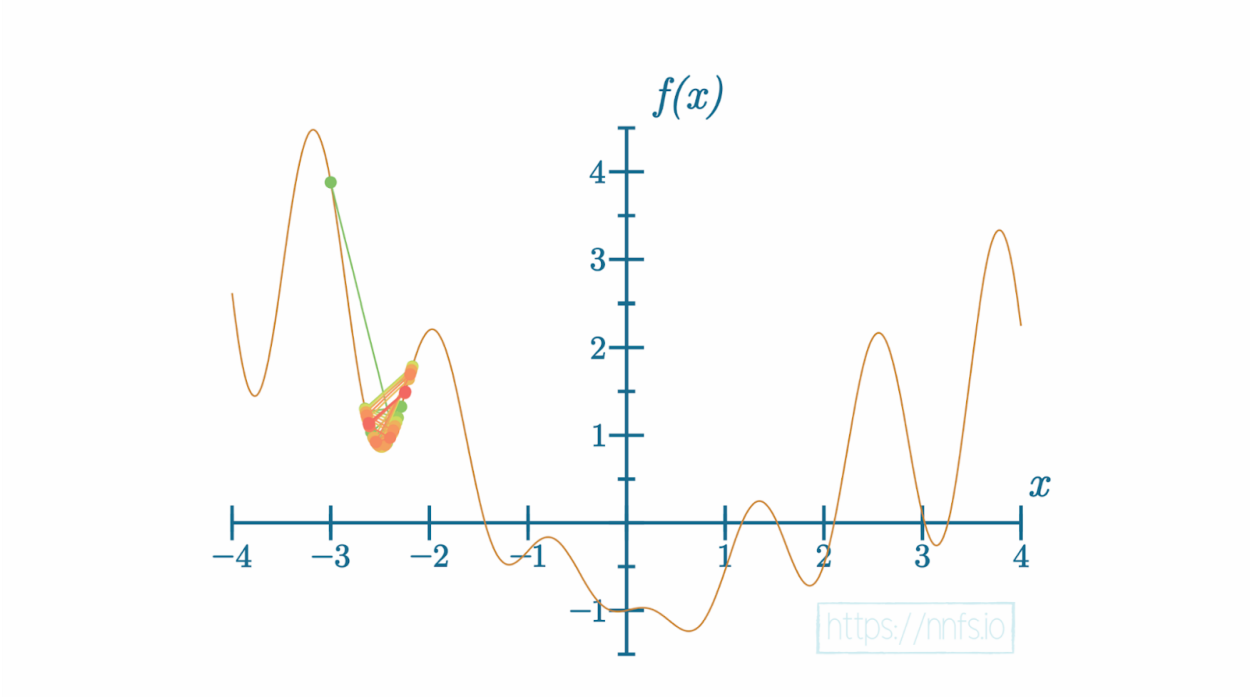


Fig 10.03: Stuck in the first local minimum.

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:

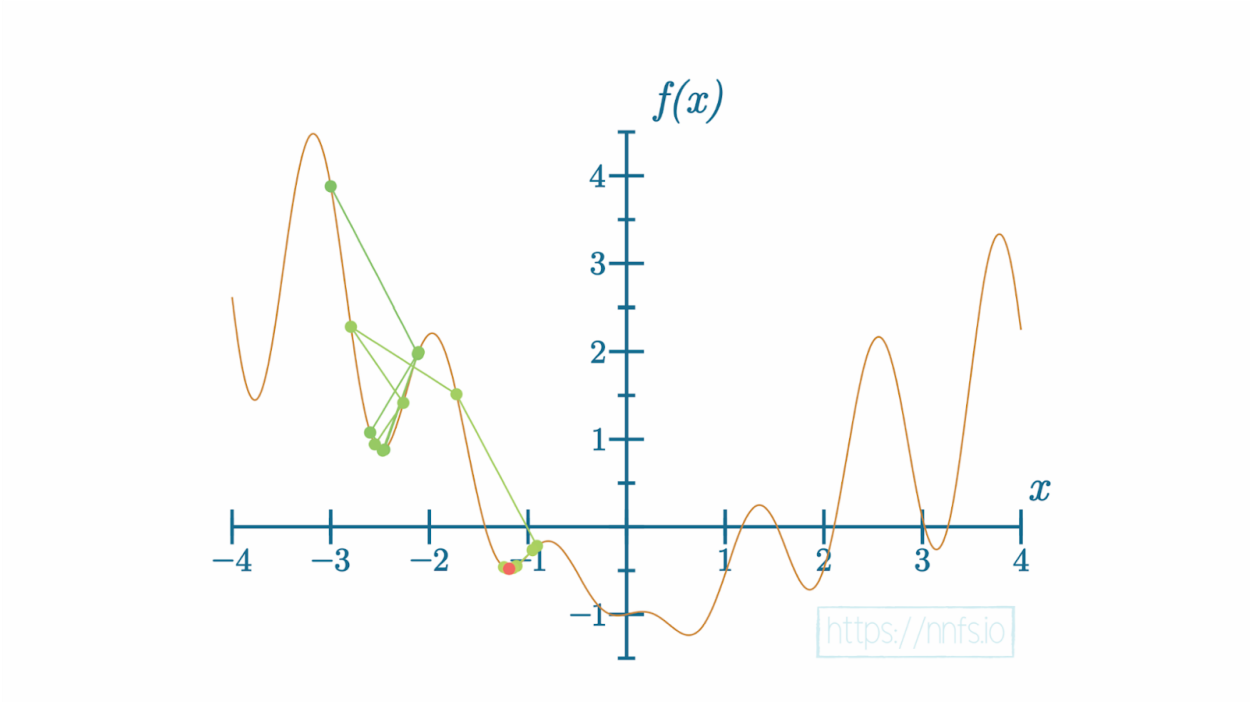


Fig 10.04: Stuck in the second local minimum.

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:

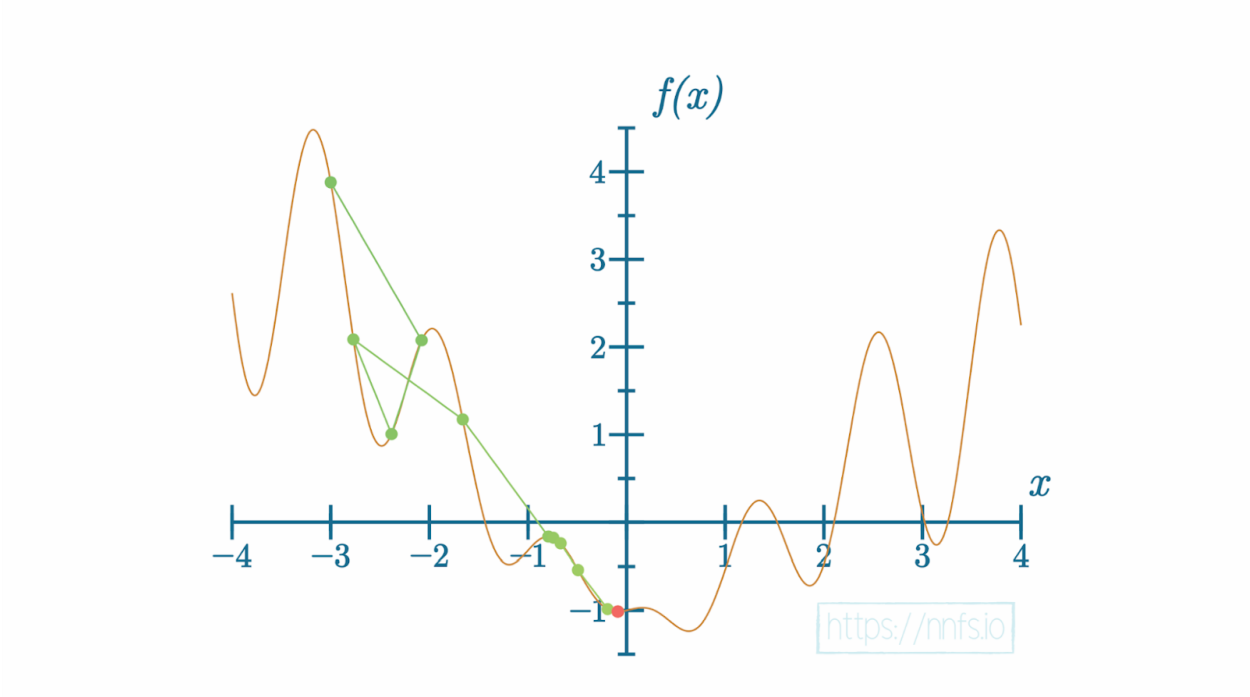


Fig 10.05: Stuck in the third local minimum, near the global minimum.

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:

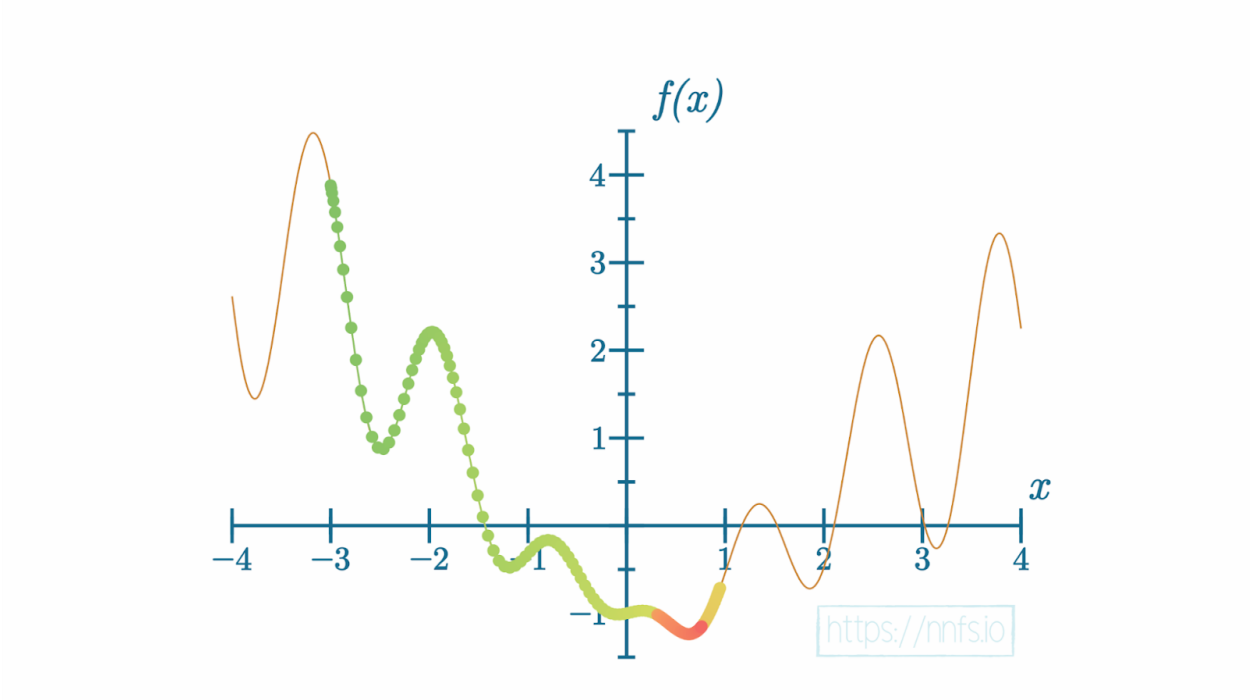


Fig 10.06: Reached the global minimum, too low learning rate.

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?

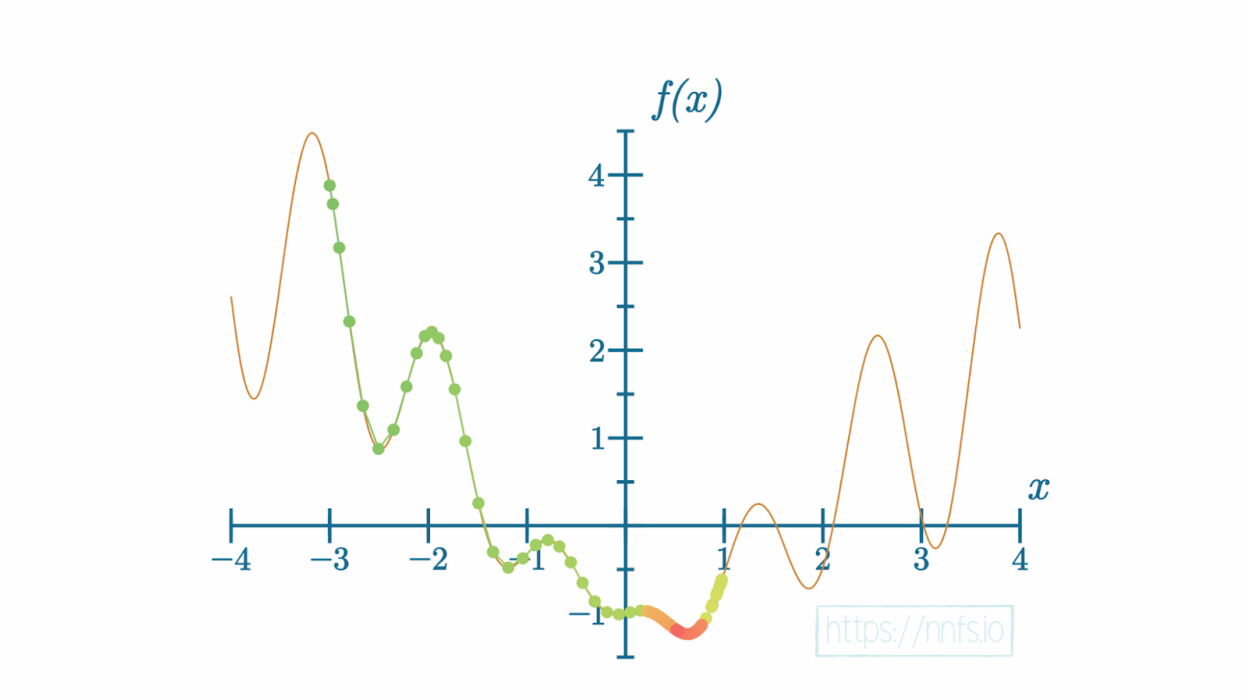


Fig 10.07: Reached the global minimum, better learning rate.

Anim 10.07: https://nnfs.io/jog

And even further:

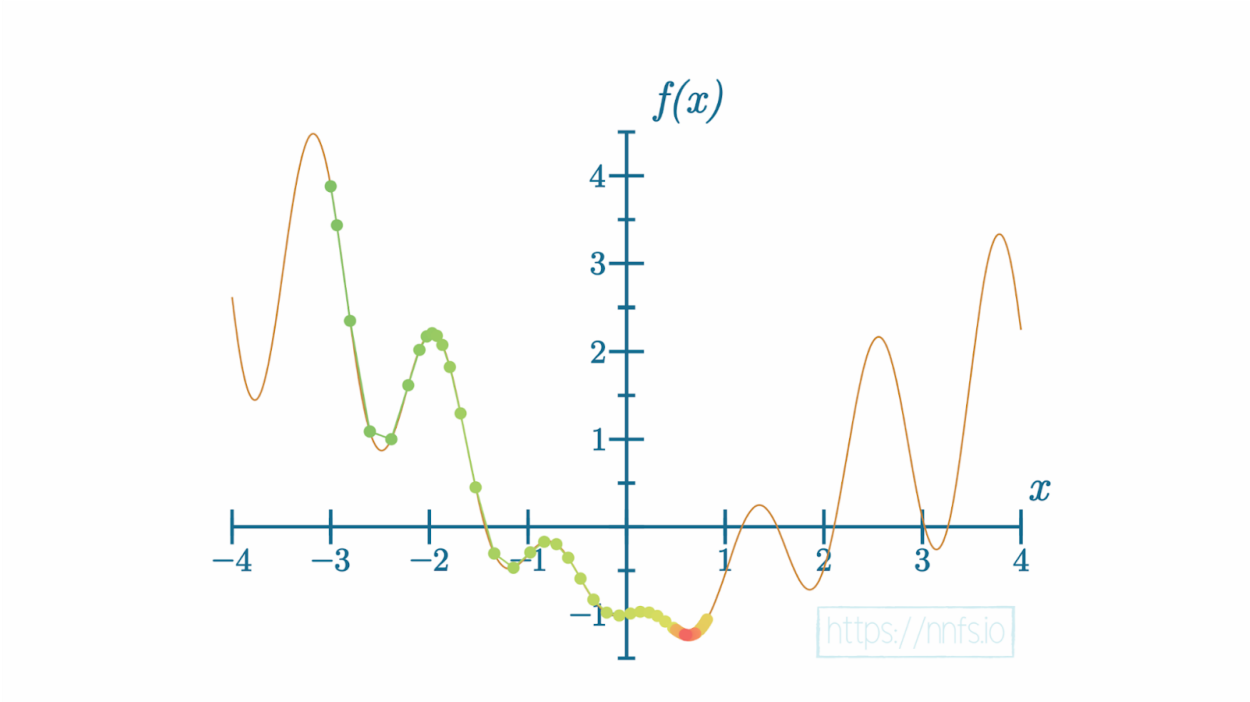


Fig 10.08: Reached the global minimum, significantly better learning rate.

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:

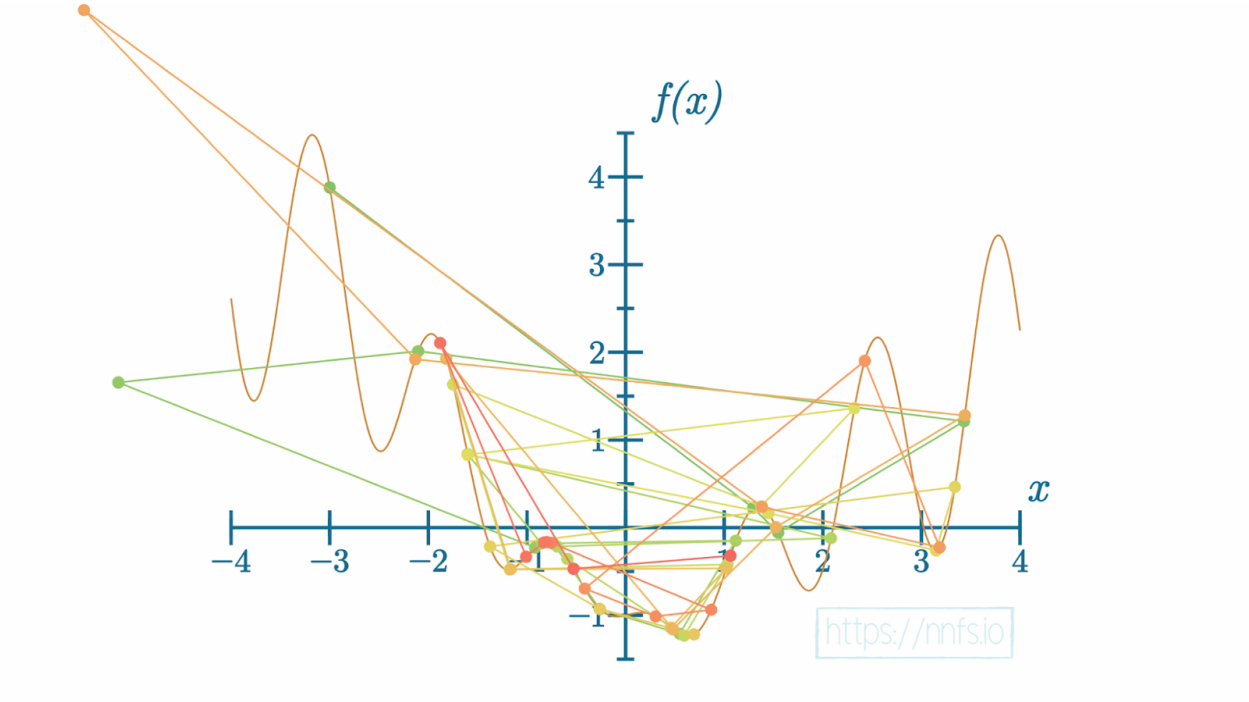


Fig 10.09: Unstable model, learning rate too big.

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:

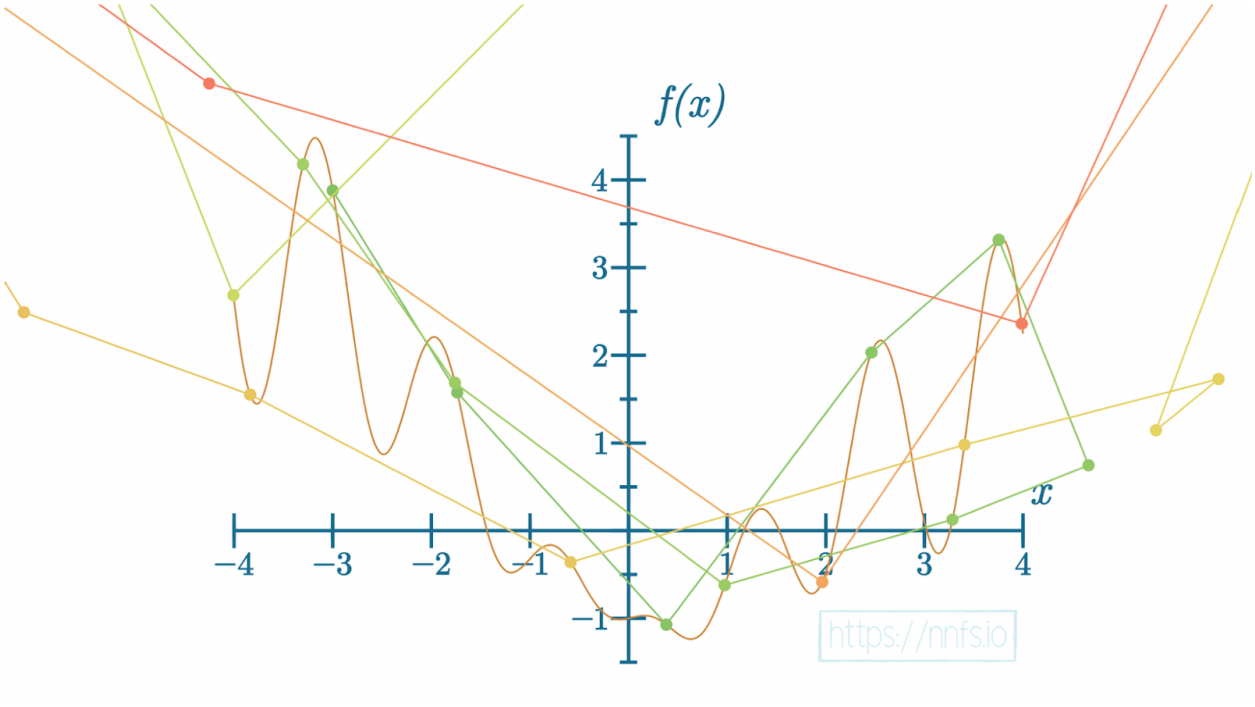


Fig 10.10: Unstable model, learning rate significantly too big.

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**:

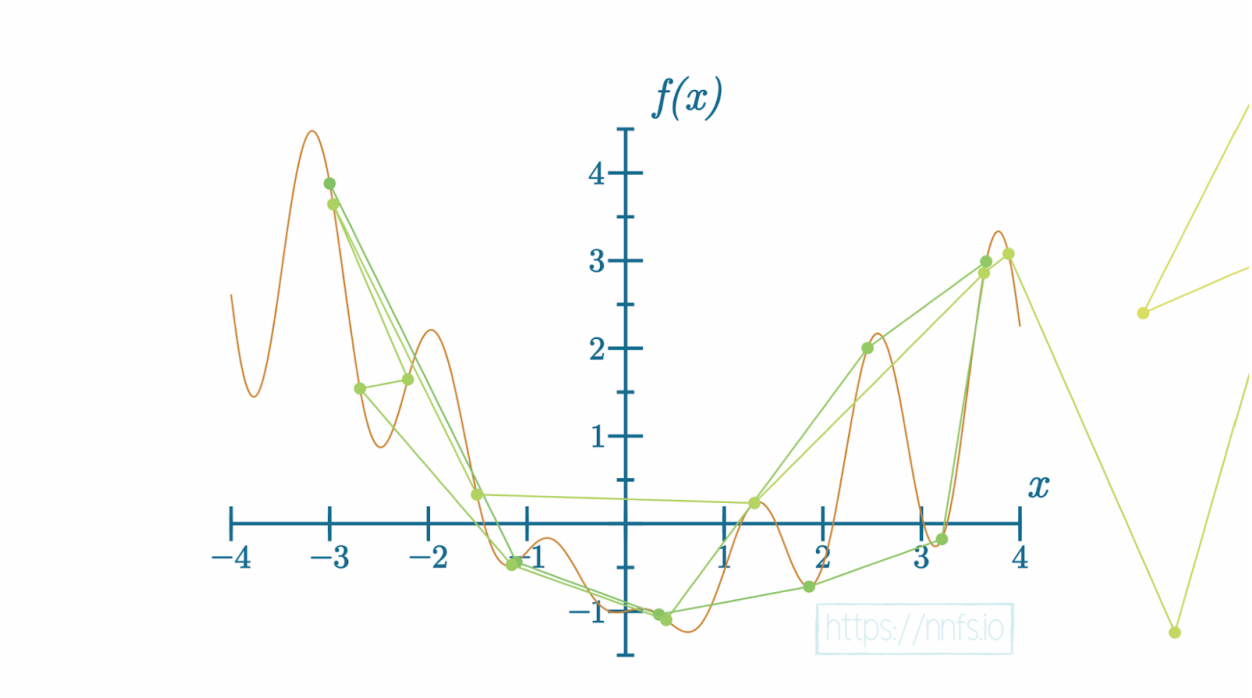


Fig 10.11: Broken model, learning rate critically too big.

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:

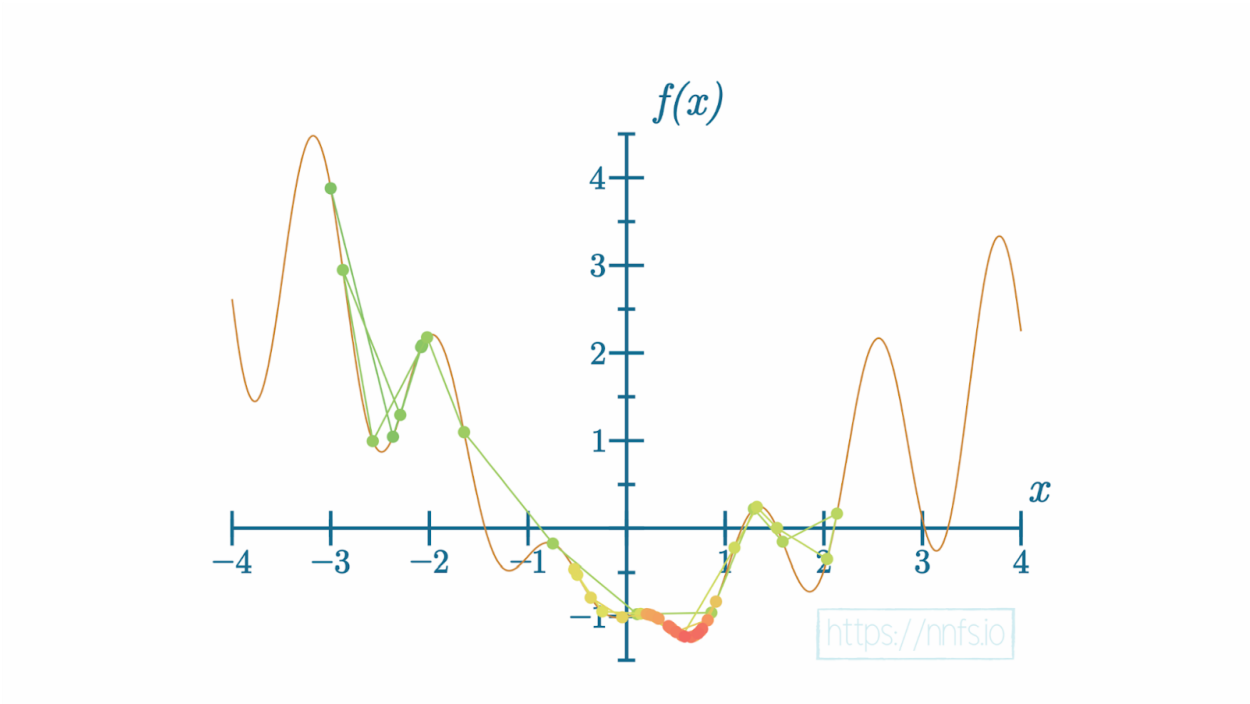


Fig 10.12: Model learned, good learning rate, can be better.

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:

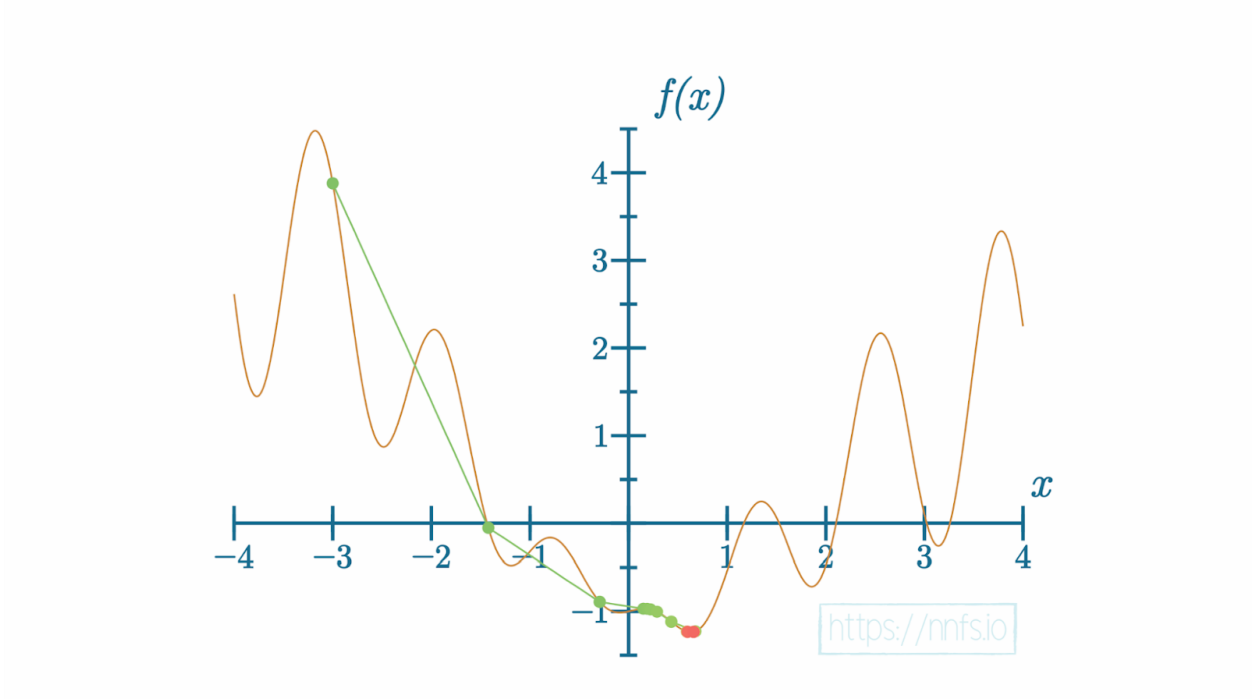


Fig 10.13: An efficient learning example.

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
```

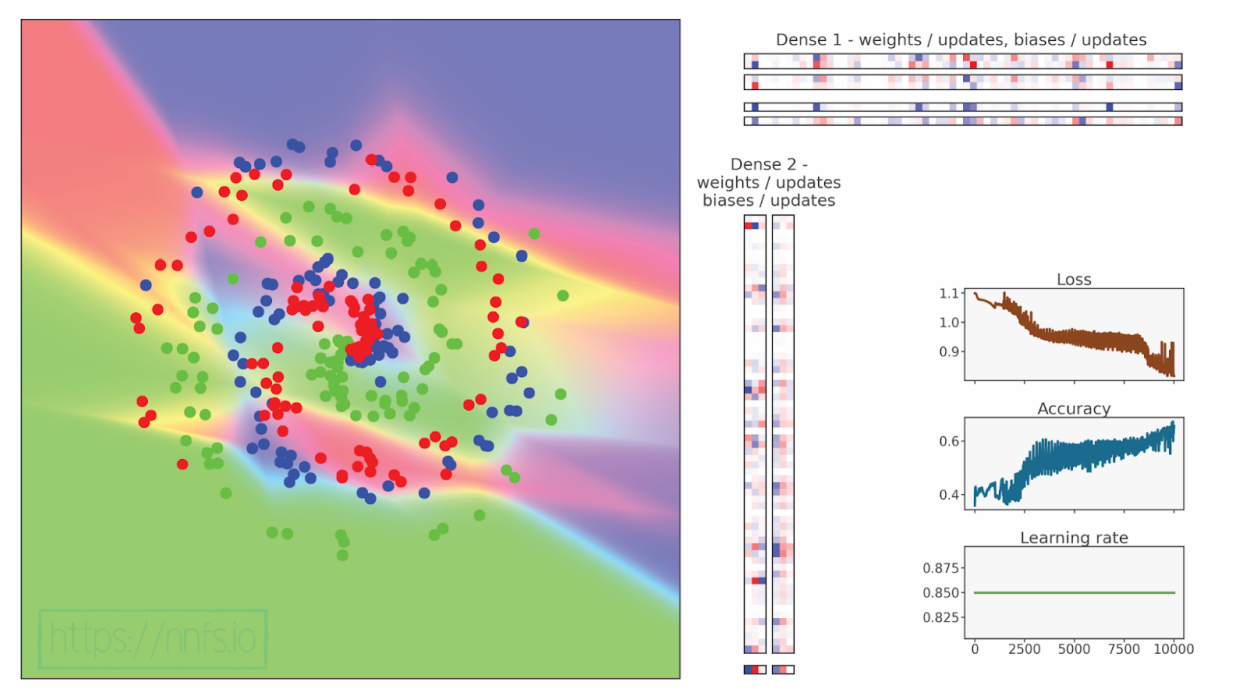


Fig 10.15: Model training with SGD optimizer and lowered learning rate.

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
```

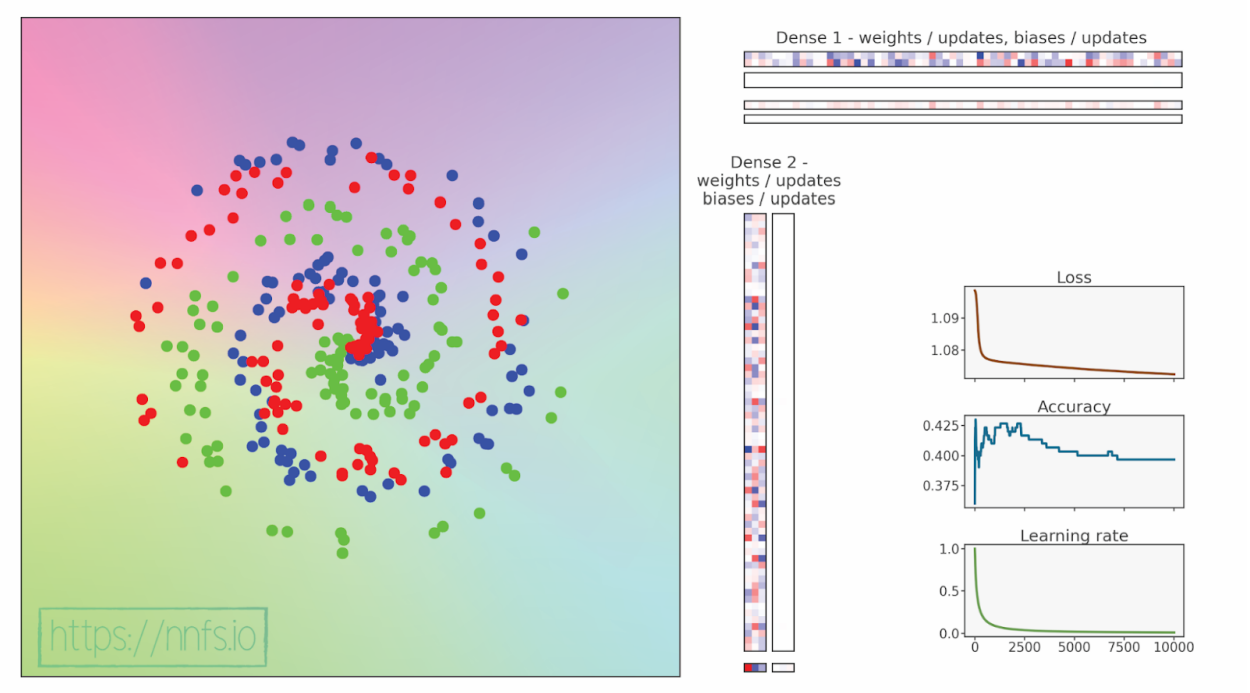


Fig 10.16: Model training with SGD optimizer and and learning rate decay set too high.

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
```

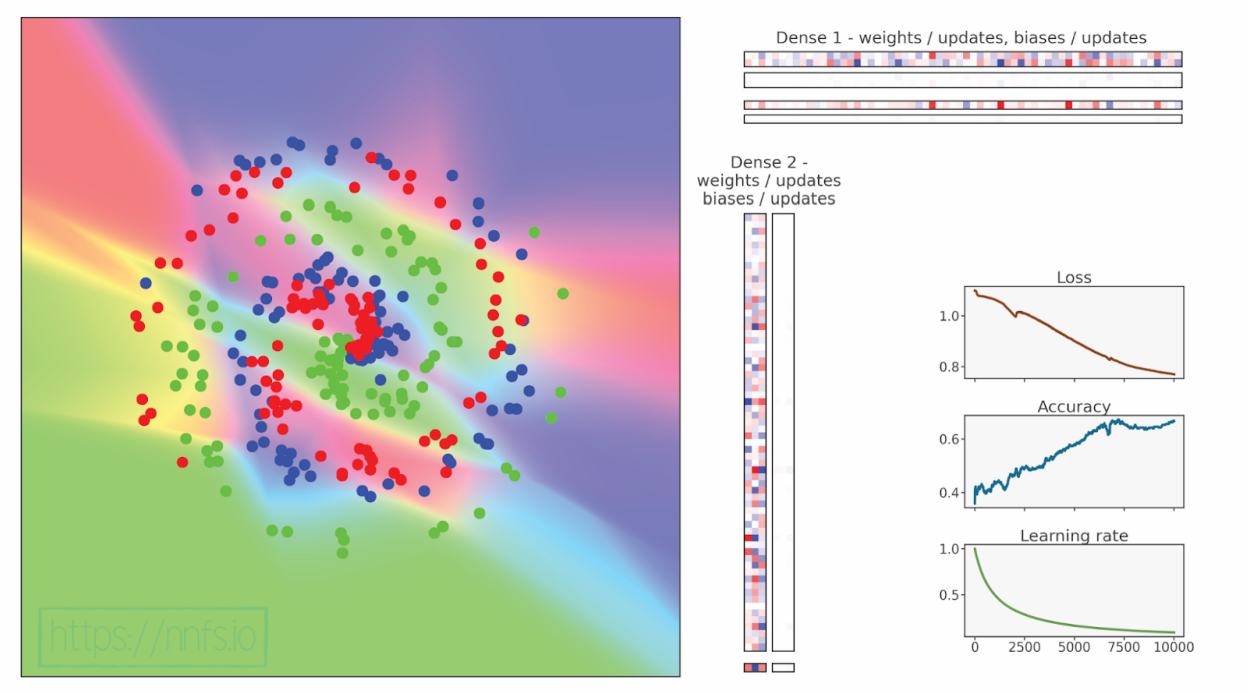


Fig 10.17: Model training with SGD optimizer and more proper learning rate decay.

Epilepsy Warning (quick flashing colors)

Anim 10.17: https://nnfs.io/muk

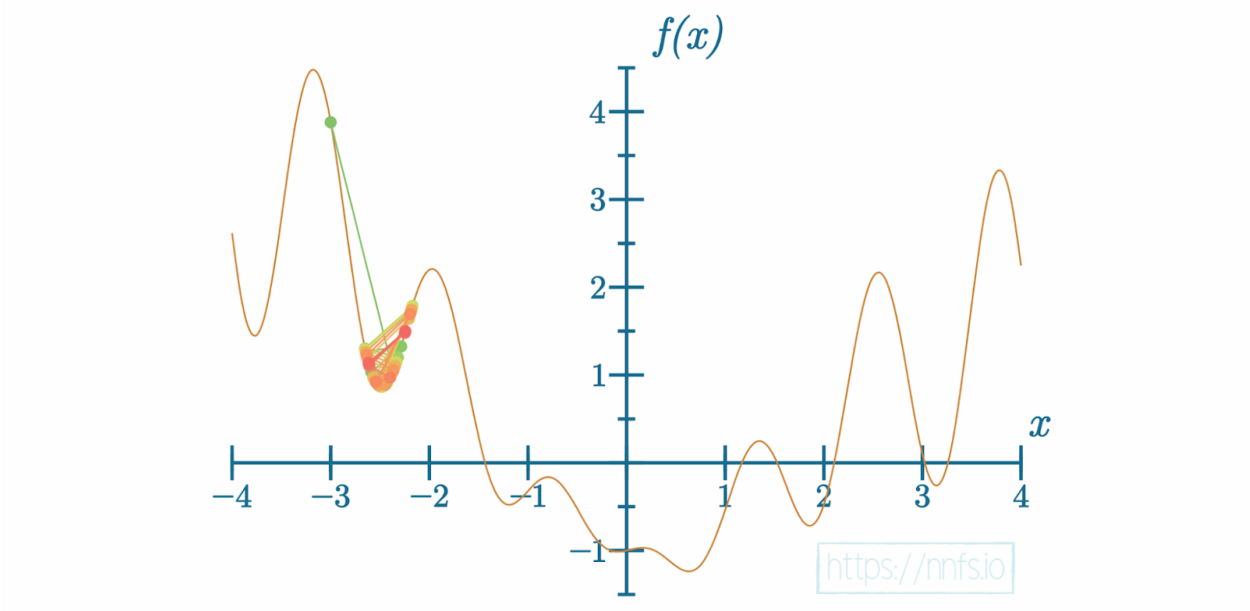
In this case, we've achieved our lowest loss and highest accuracy thus far, but it still should be possible to find parameters that will give us even better results. For example, you may suspect that the initial learning rate is too high. It can make for a great exercise to attempt to find better settings. 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.

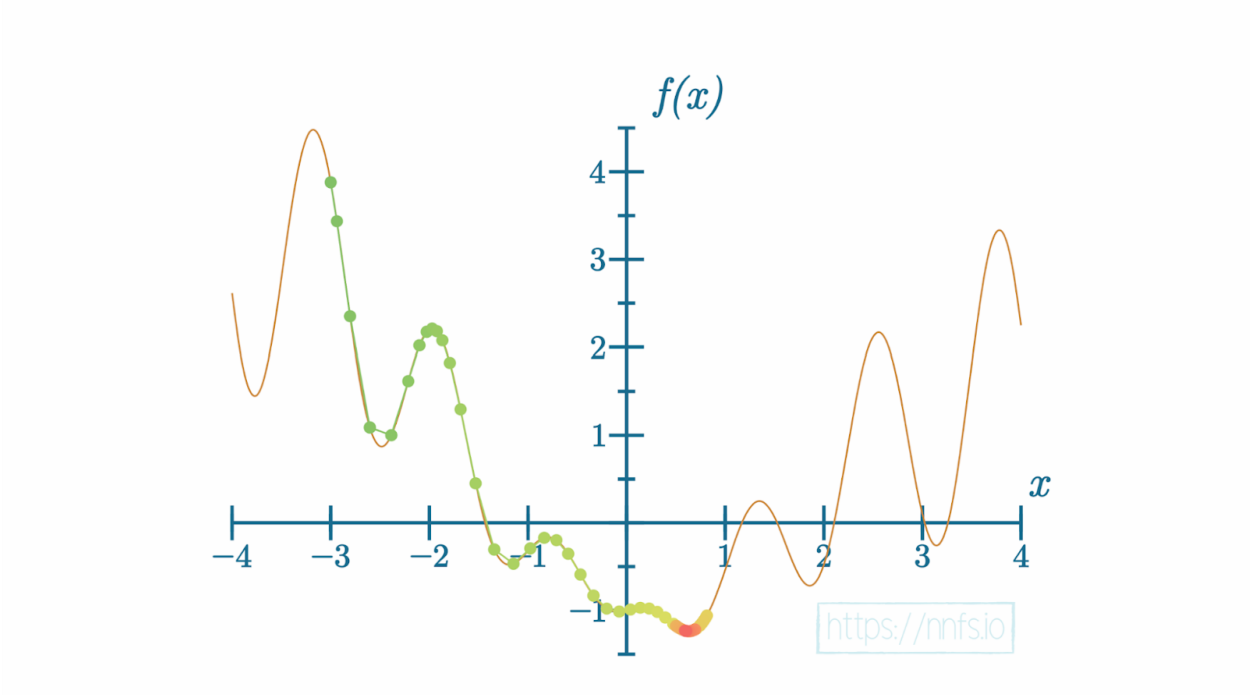
Recall this situation from the beginning of this chapter:



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
```

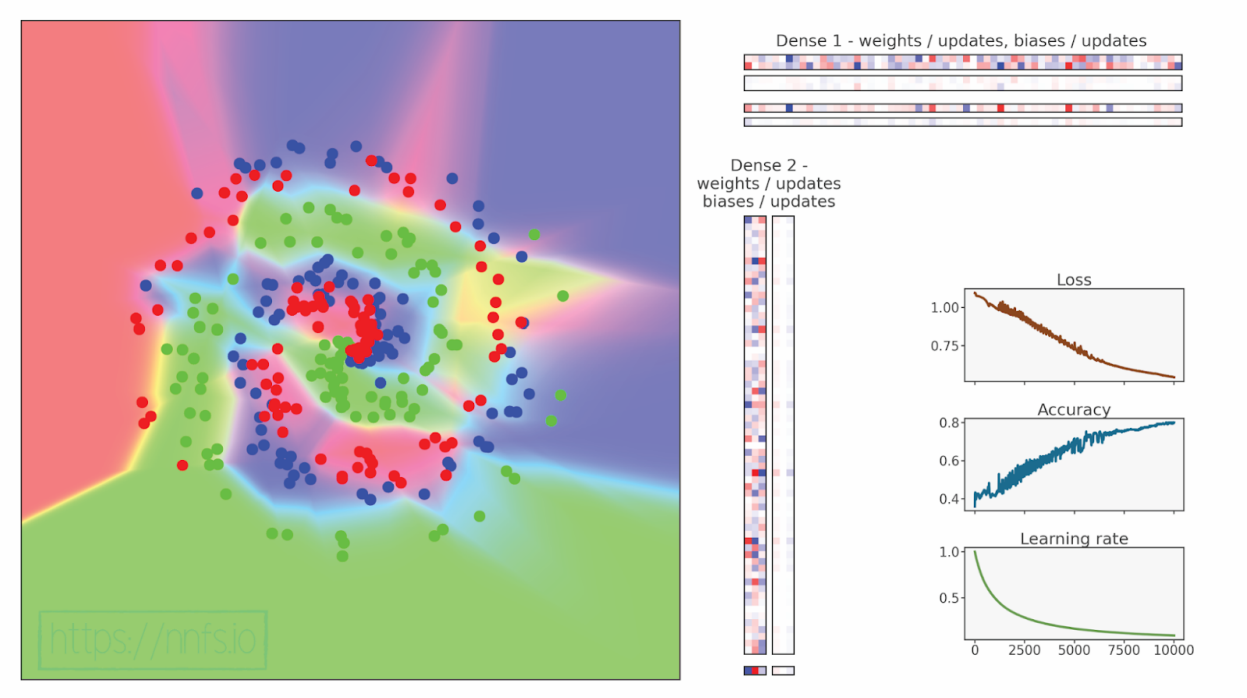


Fig 10.18: Model training with SGD optimizer, learning rate decay and Momentum.

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
```

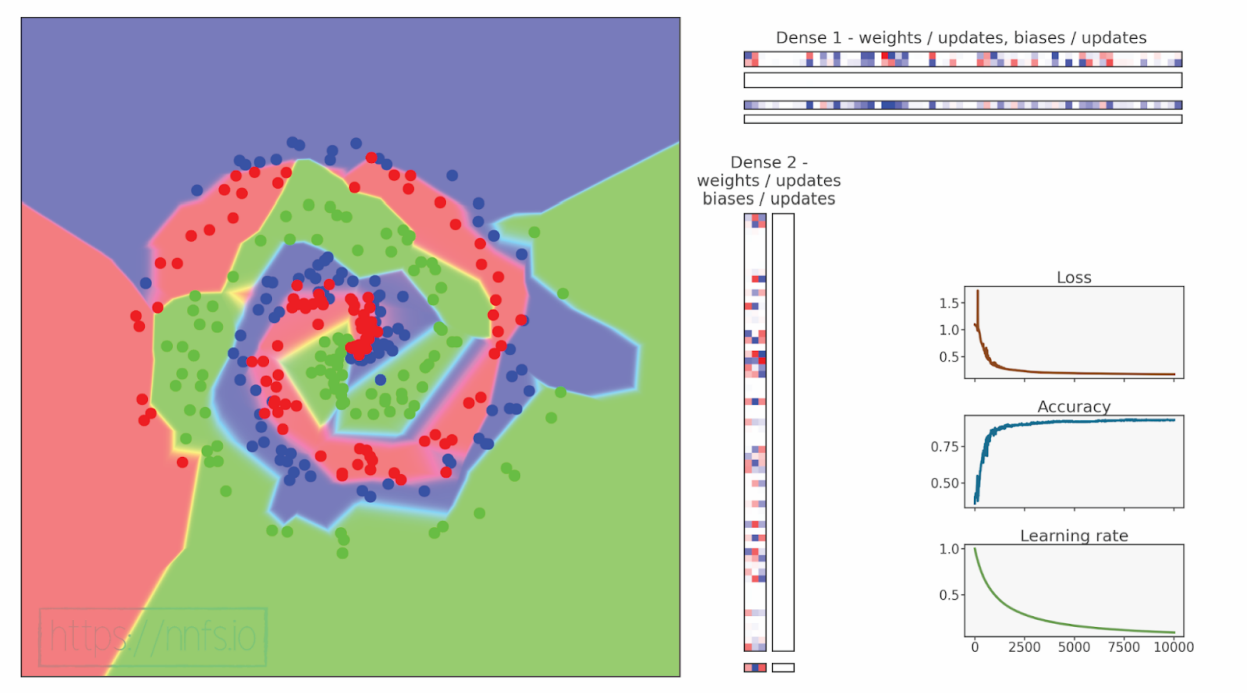


Fig 10.19: Model training with SGD optimizer, learning rate decay and Momentum (tuned).

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
```

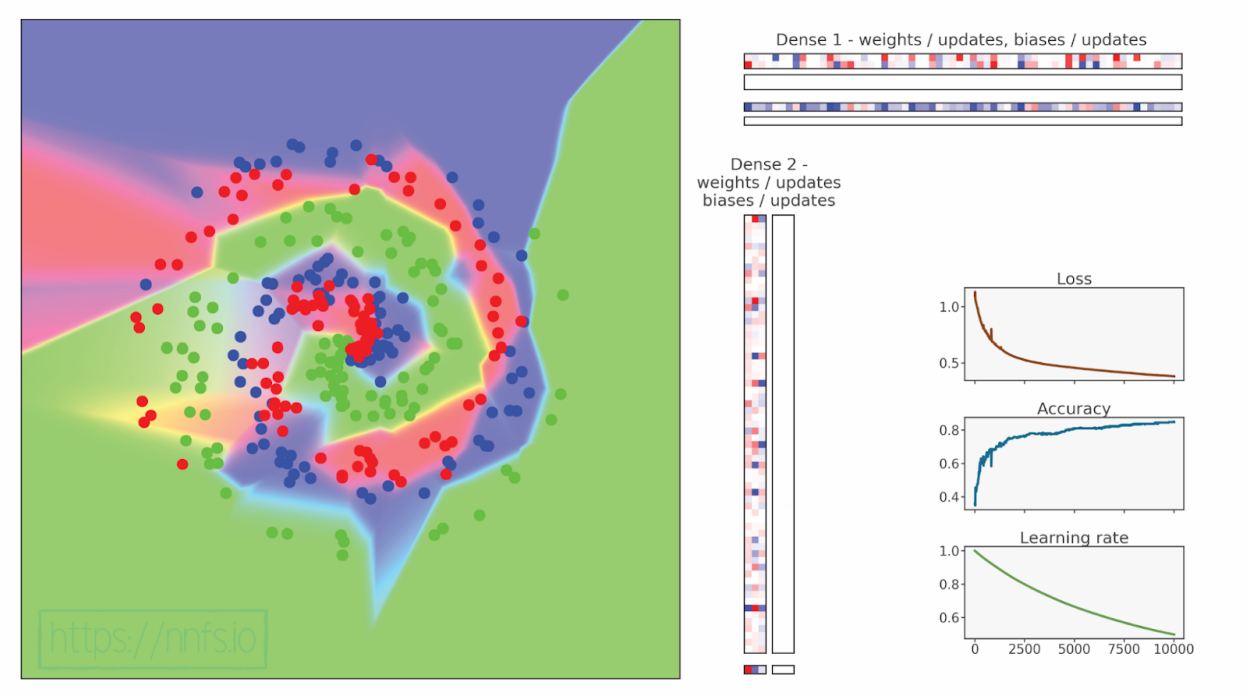


Fig 10.20: Model training with AdaGrad optimizer.

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
```

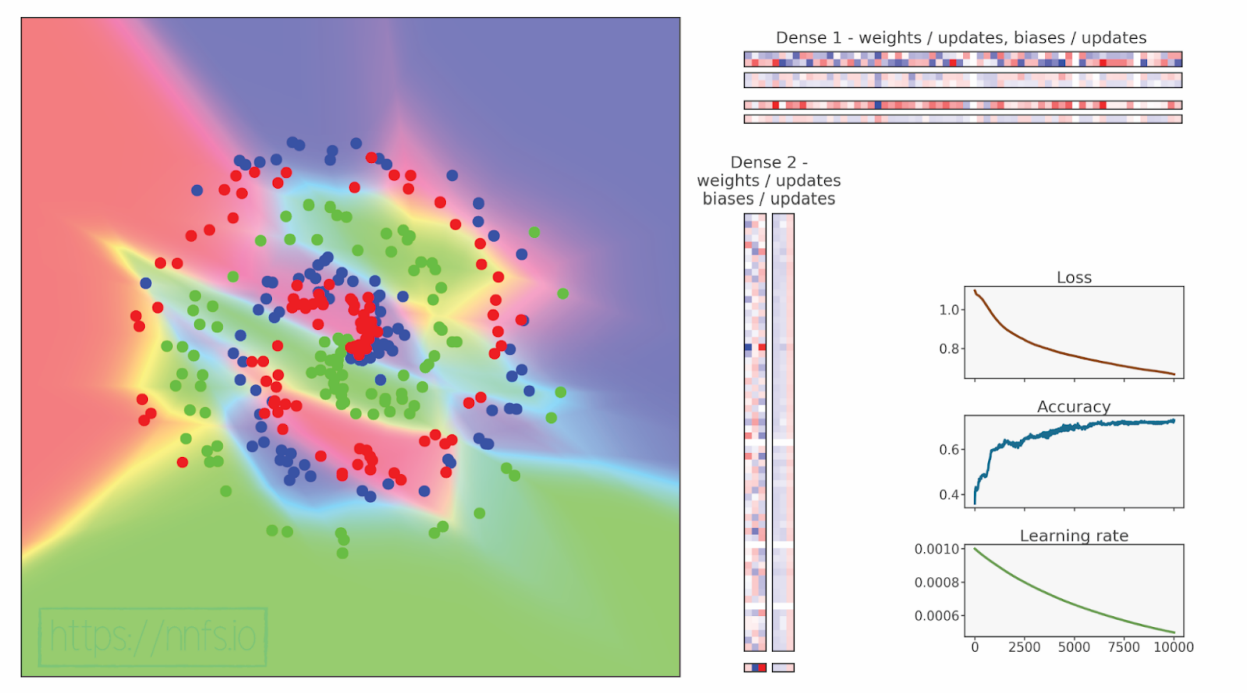


Fig 10.21: Model training with RMSProp optimizer.

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
```