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.

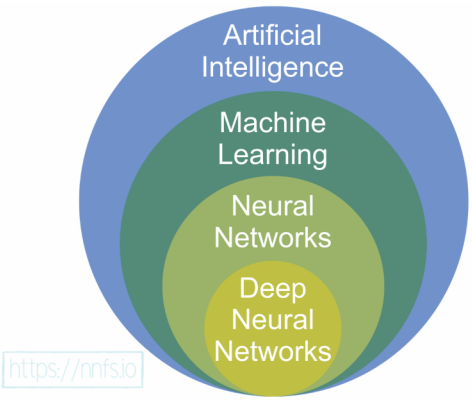

Fig 1.01: Depicting the various fields of artificial intelligence and where they fit in overall.

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.

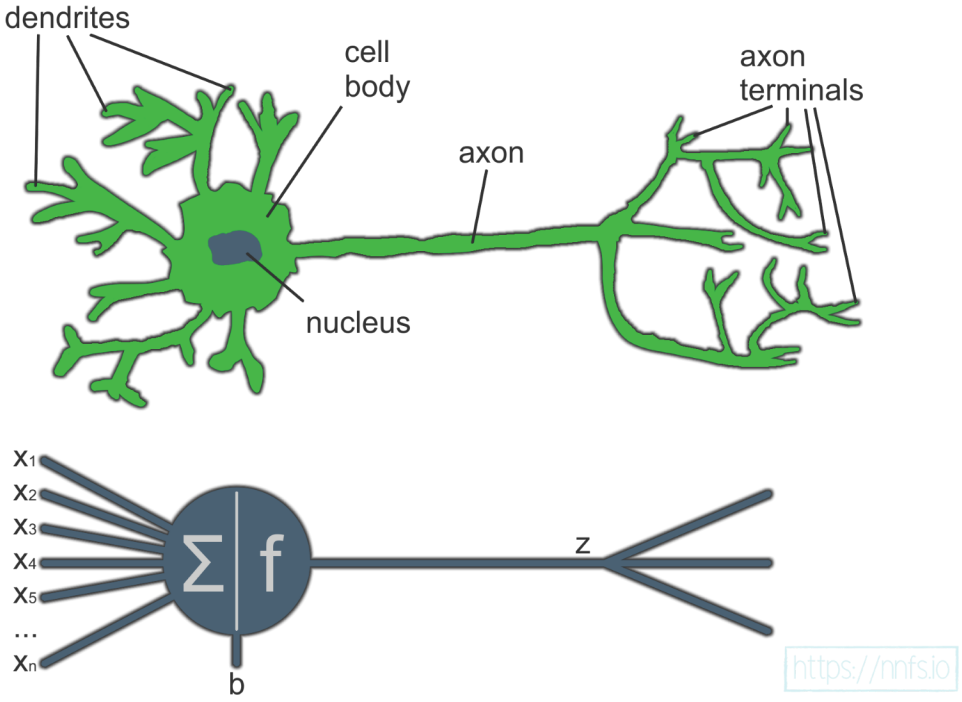

Fig 1.02: Comparing a biological neuron to an artificial neuron.

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.

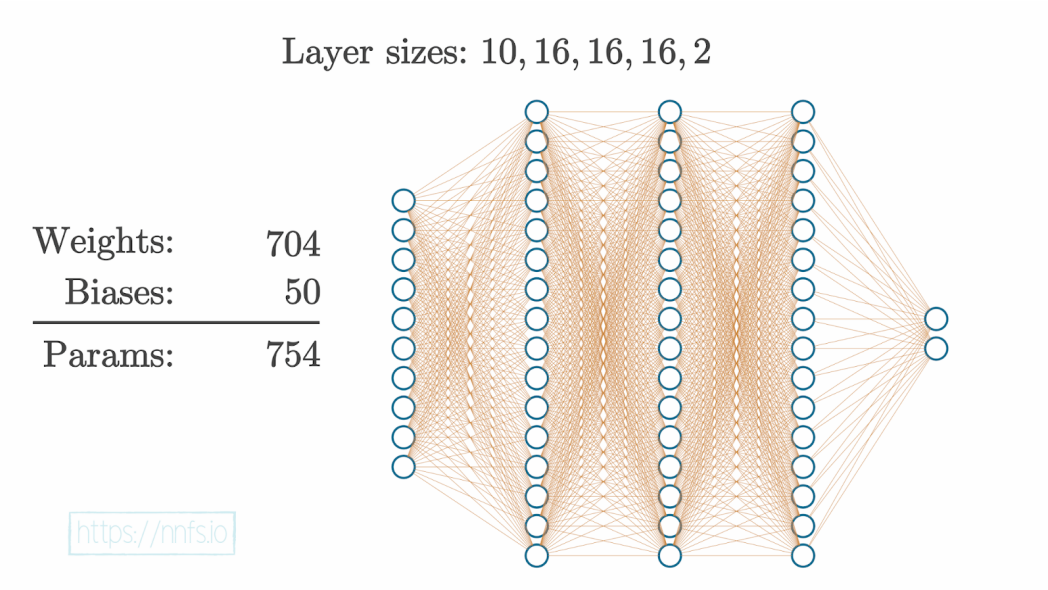

Fig 1.03: Example of a neural network with 3 hidden layers of 16 neurons each.

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:

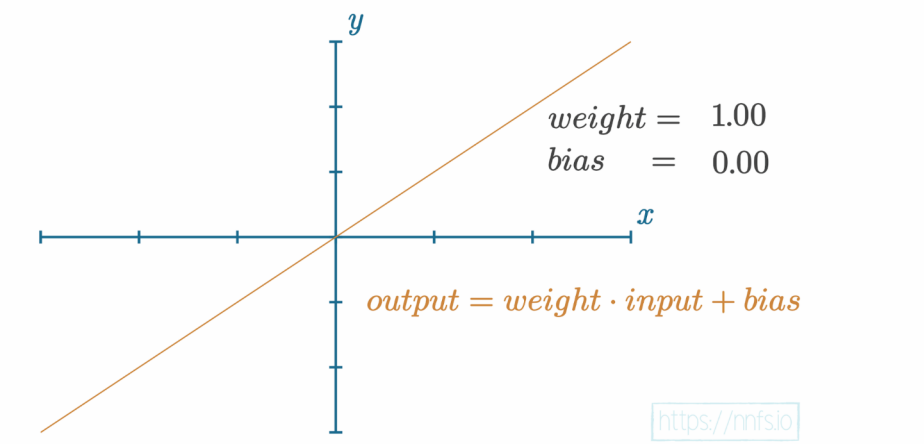

Fig 1.04: Graph of a single-input neuron's output with a weight of 1, bias of 0 and input *x*.

Adjusting the weight will impact the slope of the function:

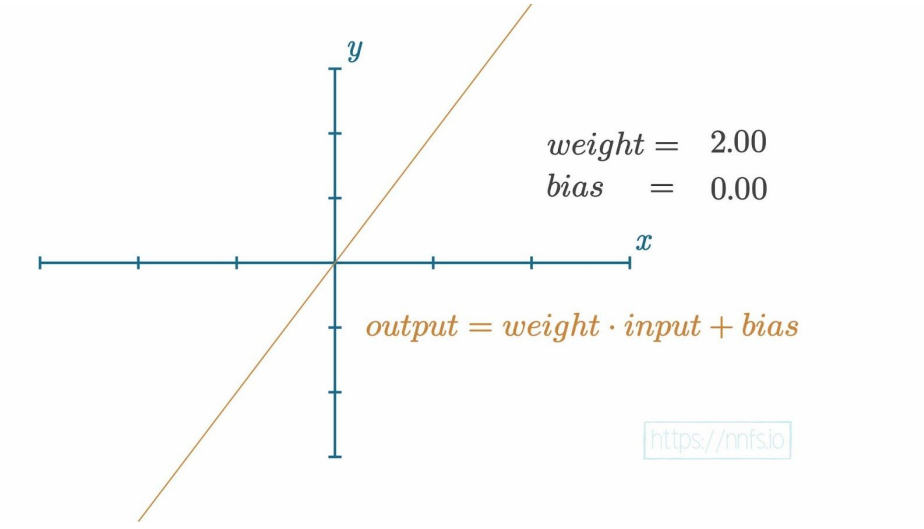

Fig 1.05: Graph of a single-input neuron's output with a weight of 2, bias of 0 and input *x*.

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:

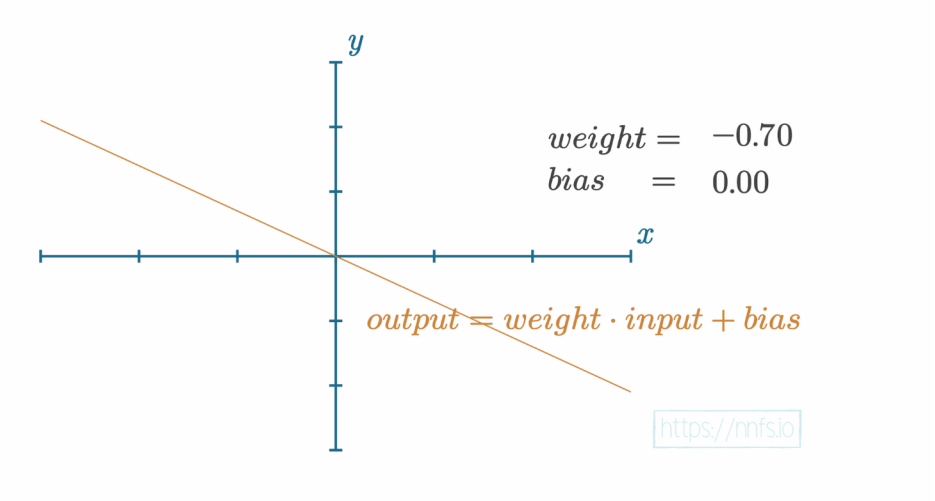

Fig 1.06: Graph of a single-input neuron's output with a weight of -0.70, bias of 0 and input *x*.

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:

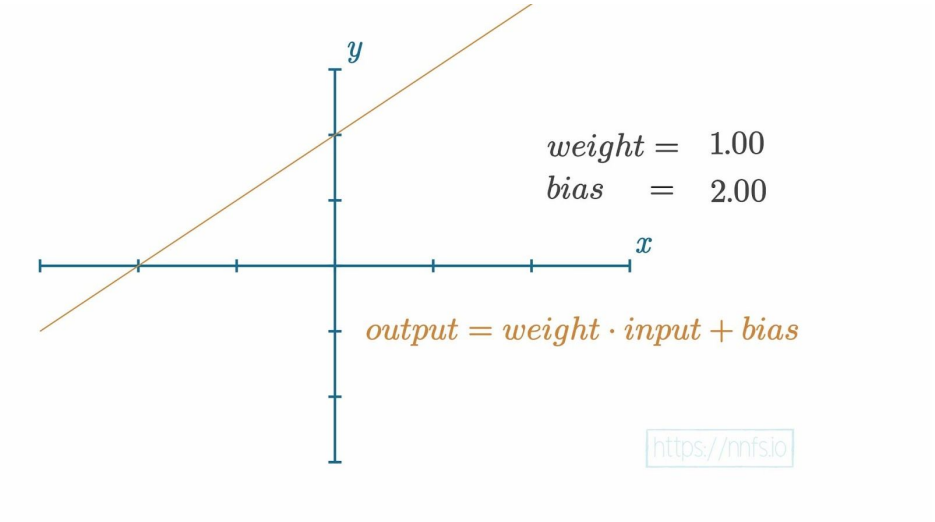

Fig 1.07: Graph of a single-input neuron's output with a weight of 1, bias of 2 and input *x*.

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:

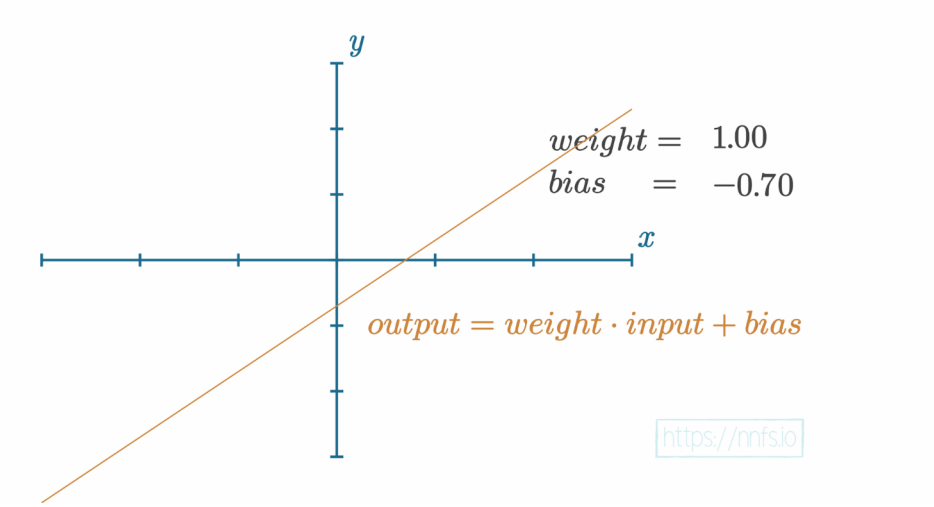

Fig 1.08: Graph of a single-input neuron's output with a weight of 1.0, bias of -0.70 and input *x*.

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.

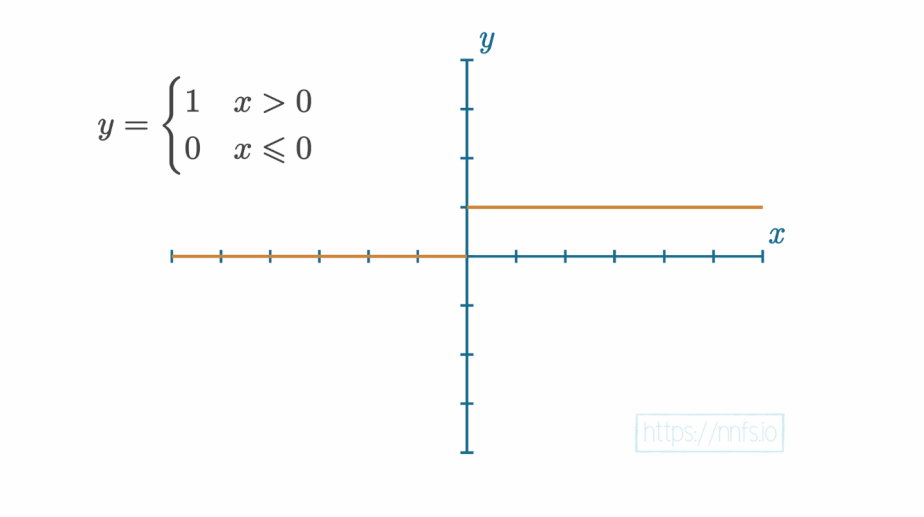

Fig 1.09: Graph of a step function.

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.

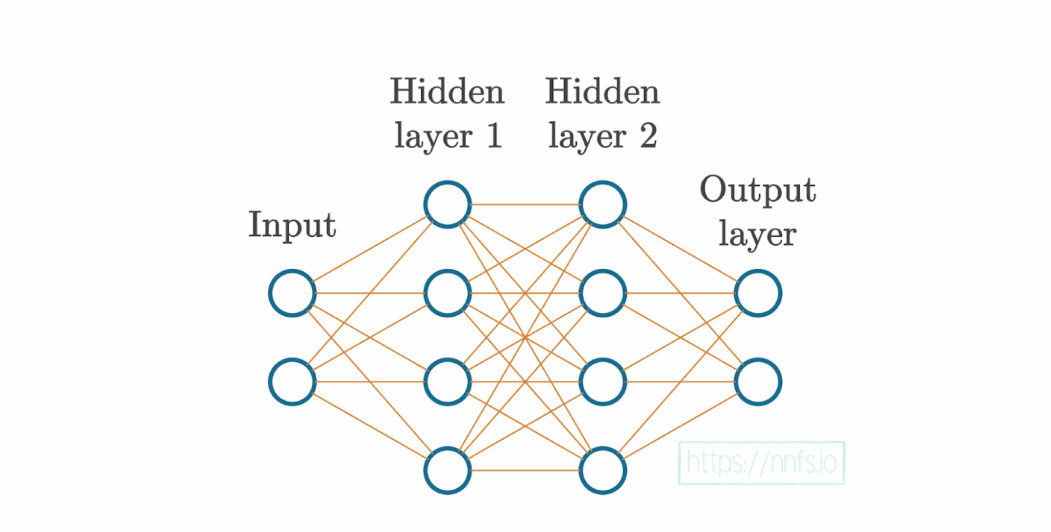

Fig 1.10: Example basic neural network.

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."

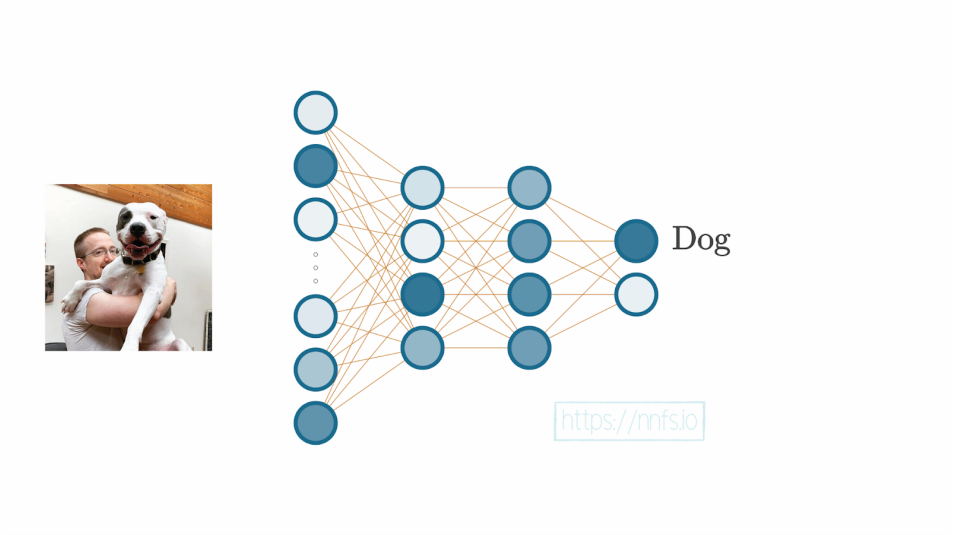

Fig 1.11: Visual depiction of passing image data through a neural network, getting a classification

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.

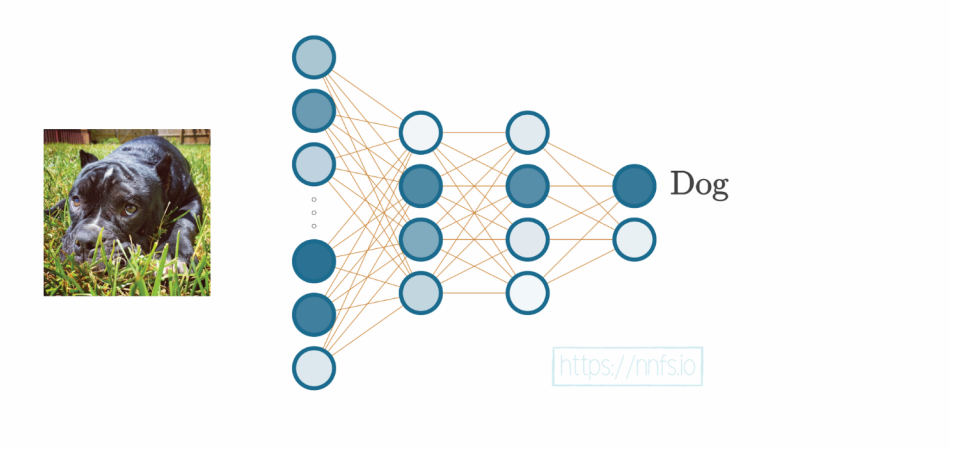

Fig 1.12: Visual depiction of passing image data through a neural network, getting a classification

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.

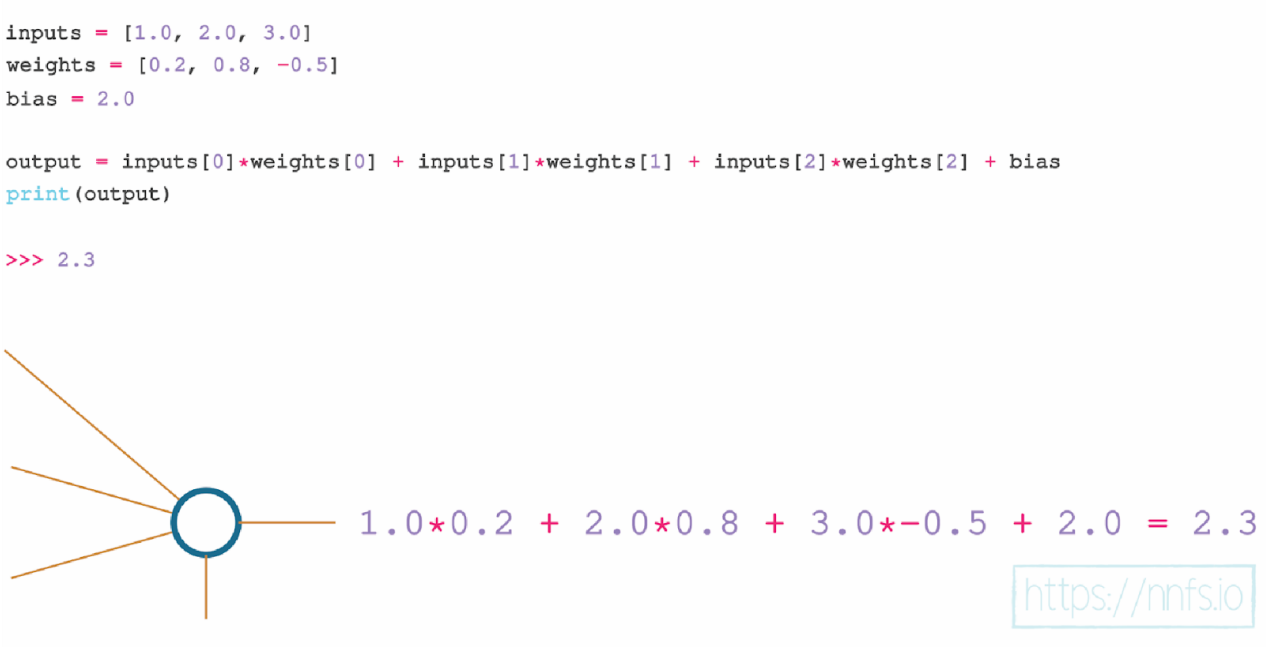

Fig 2.01: Visualizing the code that makes up the math of a basic neuron.

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:

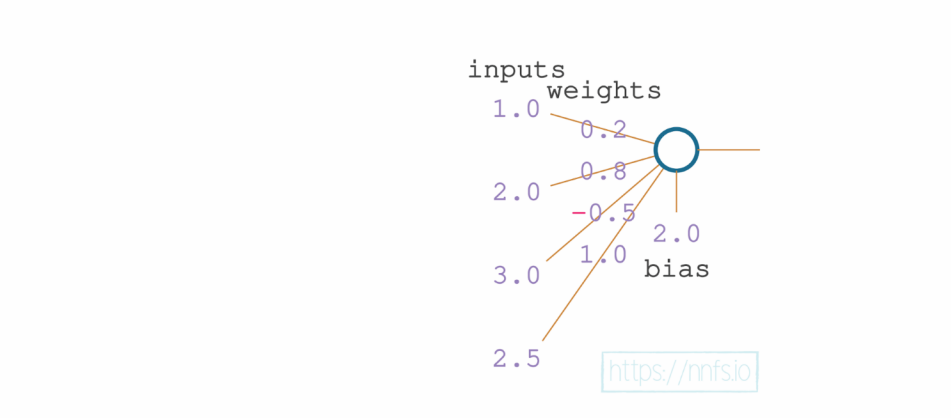

Fig 2.02: Visualizing how the inputs, weights, and biases from the code interact with the neuron.

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:

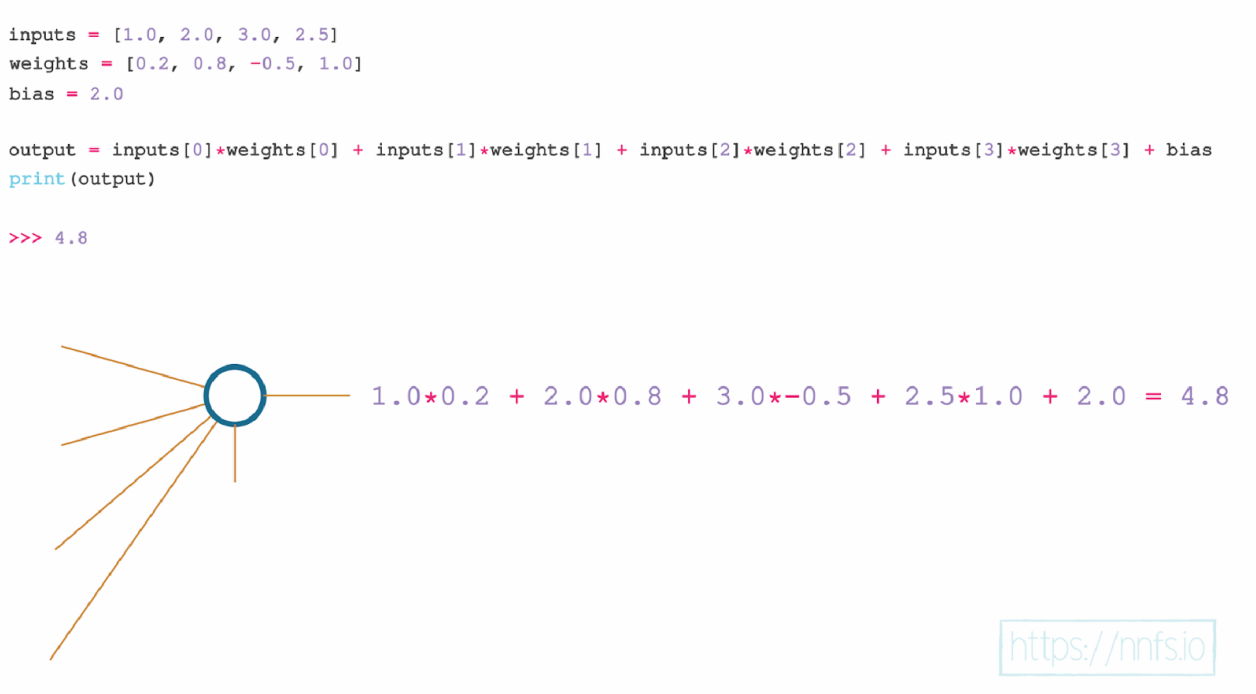

Fig 2.03: Visualizing the code that makes up a basic neuron, with 4 inputs this time.

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:

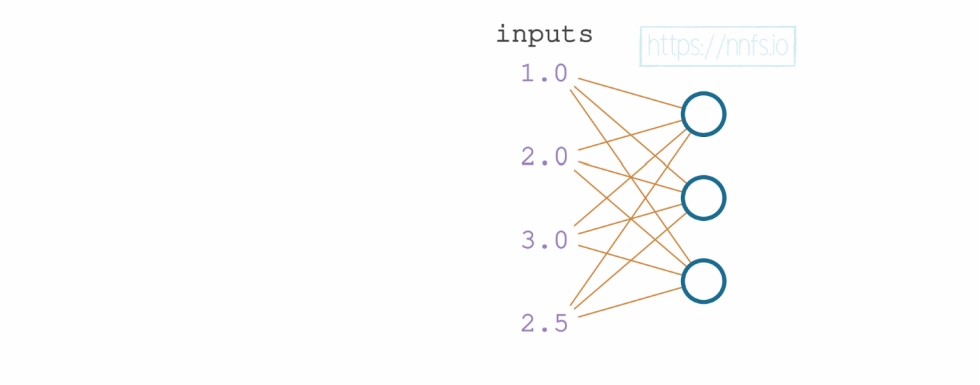

Fig 2.04: Visualizing a layer of neurons with common input.

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]
```


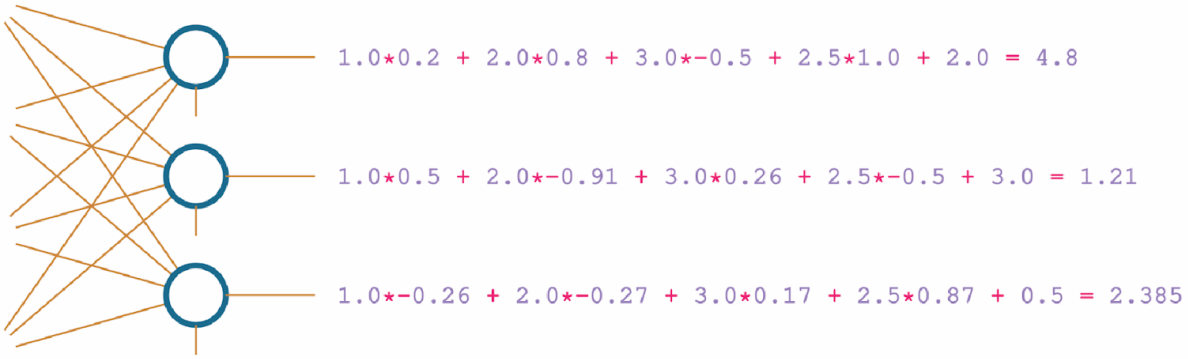

Fig 2.04.2: Code, math and visuals behind a layer of neurons.

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.

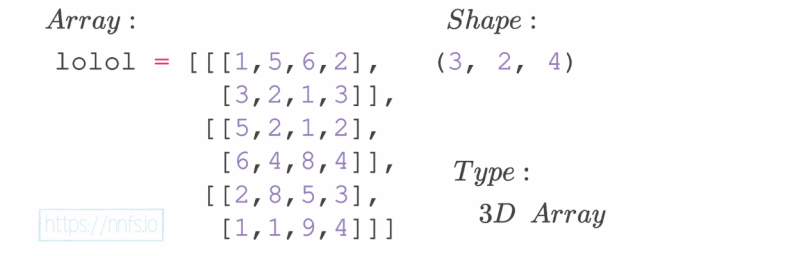

Fig 2.05: Example of a 3-dimensional array.

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
```

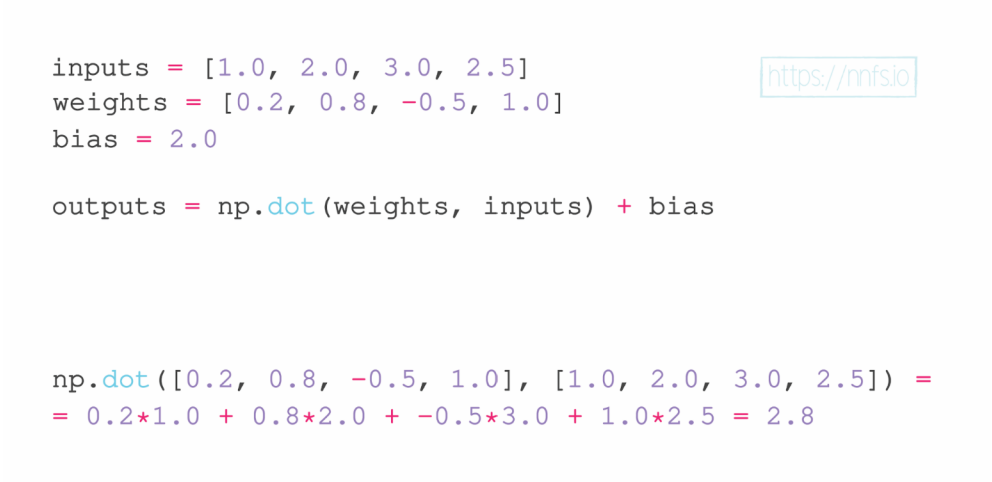

Fig 2.07: Visualizing the math of the dot product of inputs and weights for a single neuron.

```
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])
```

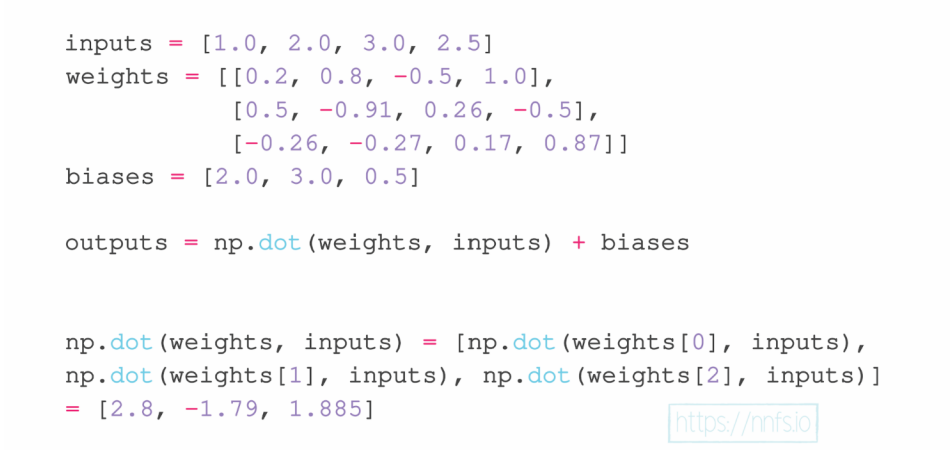

Fig 2.09: Code and visuals for the dot product applied to the layer of neurons.

```
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**.

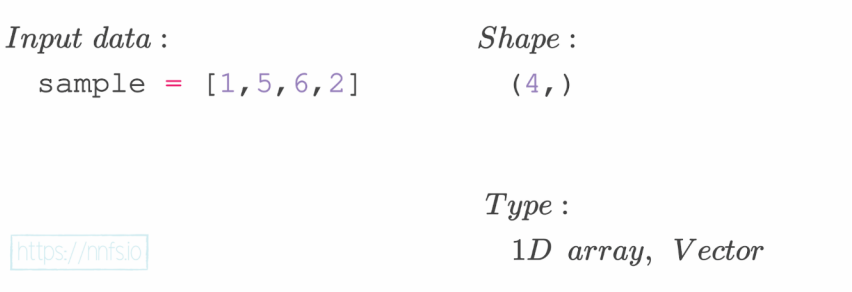

Fig 2.11: Visualizing a 1D array.

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:

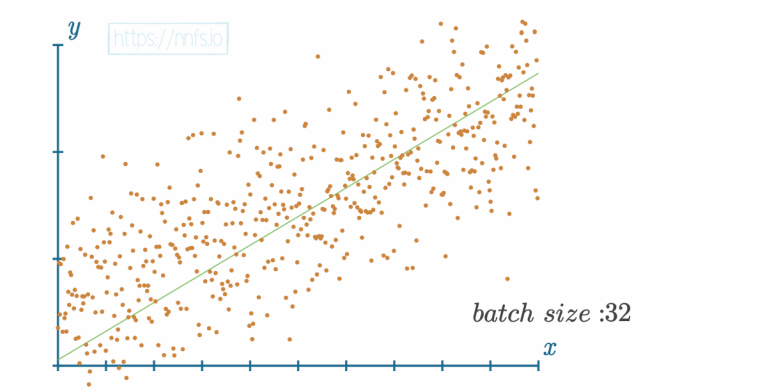

Fig 2.12: Example of a linear equation fitting batches of 32 chosen samples. See animation below for other sizes of samples at a time to see how much of a difference batch size can make.

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**:

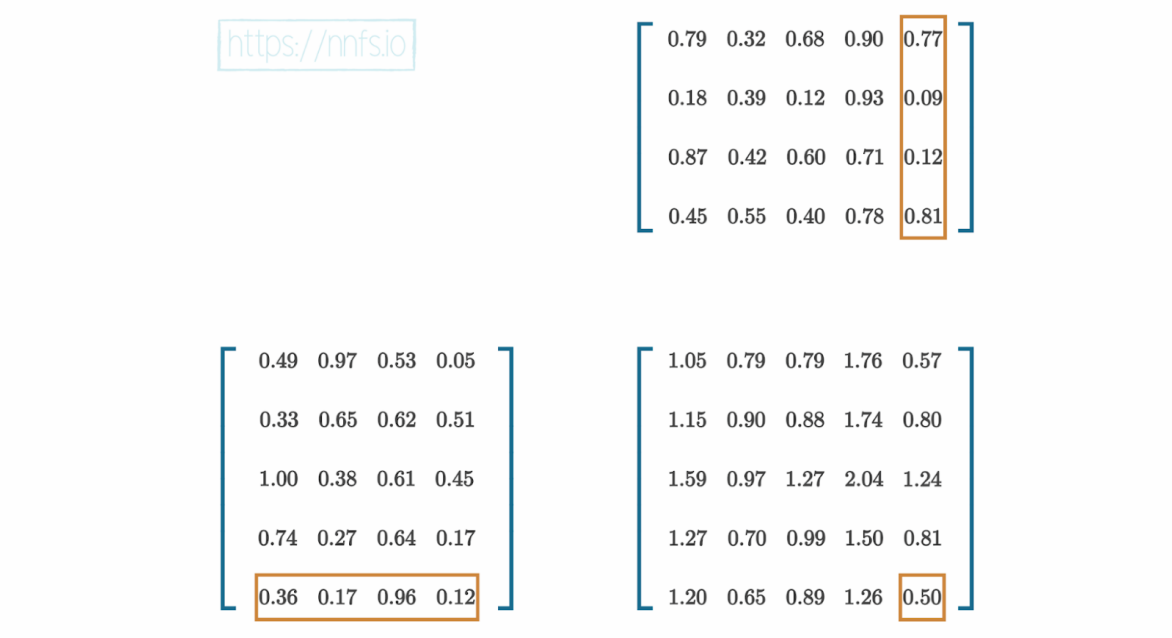

Fig 2.14: Visualizing how a single element in the resulting matrix from matrix product is calculated. See animation for the full calculation of each element.

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:

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

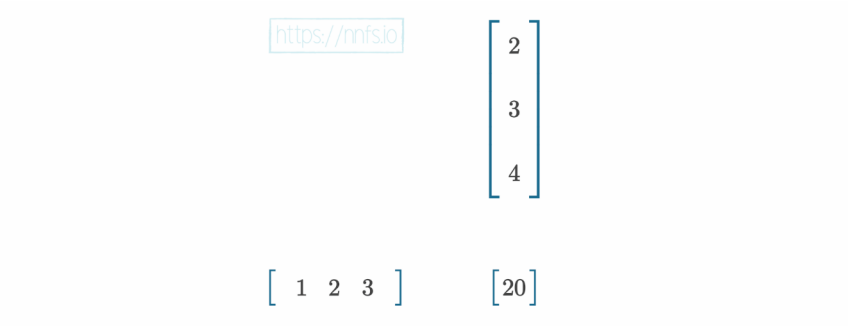

Fig 2.15: Product of row and column vectors.

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

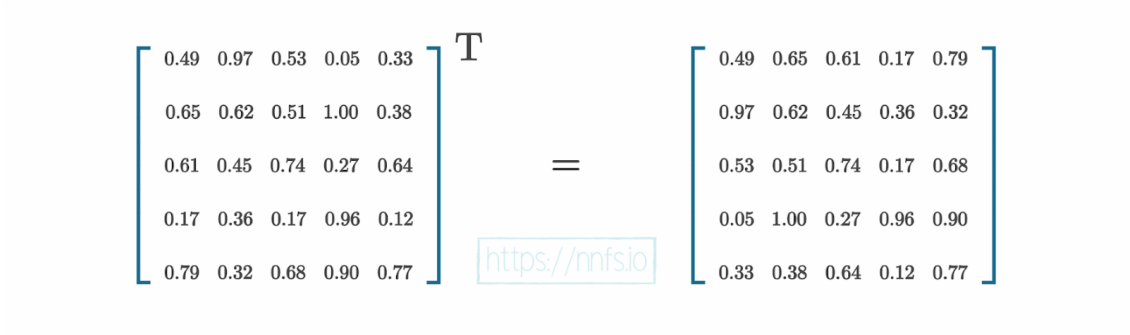

Fig 2.17: Another example of an array transposition.

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]]
```

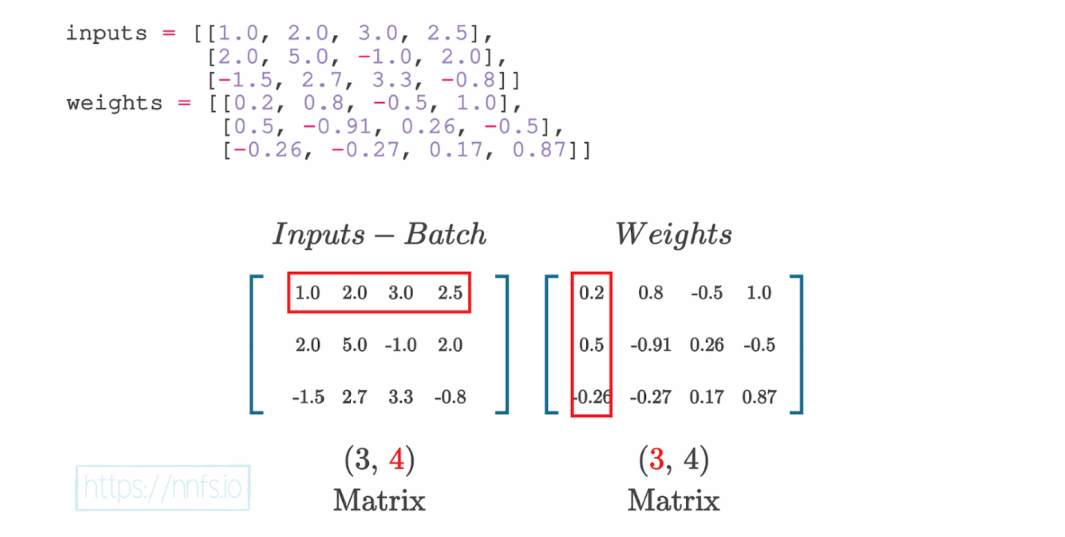

Fig 2.18: Depiction of why we need to transpose to perform the matrix product.

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]]
```

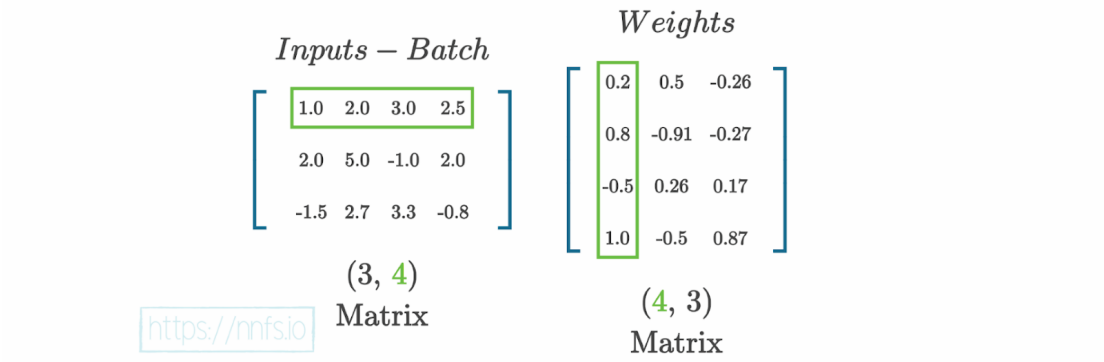

Fig 2.19: After transposition, we can perform the matrix product.

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.

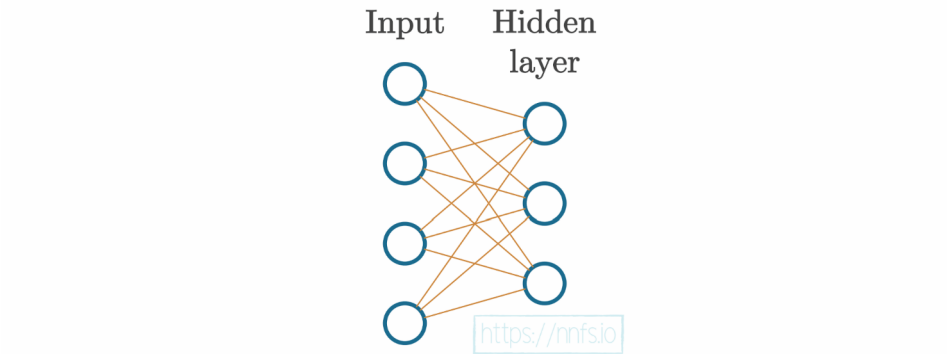

Fig 3.01: Input layer with 4 features into a hidden layer with 3 neurons.

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:

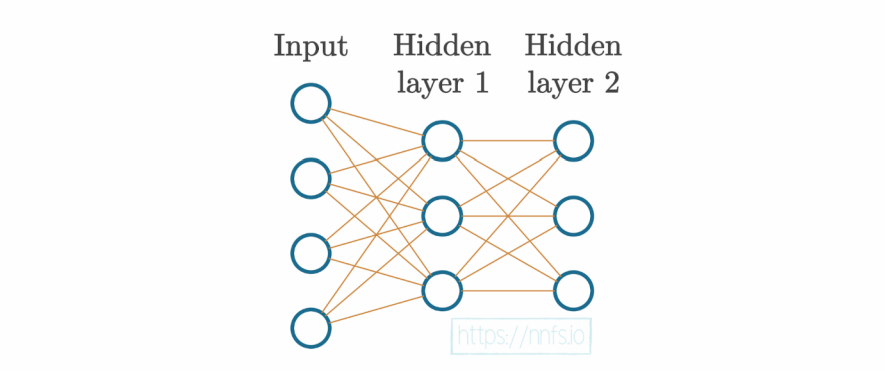

Fig 3.02: 4 features input into 2 hidden layers of 3 neurons each.

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.

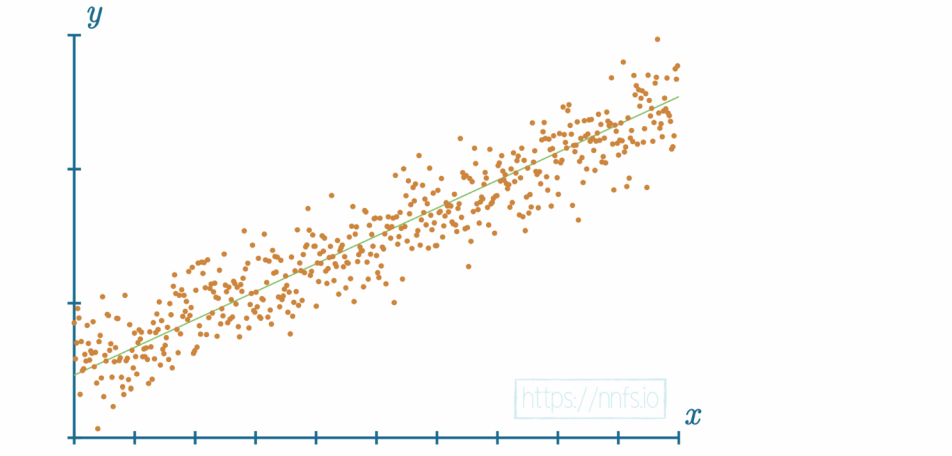

Fig 3.03: Example of data (orange dots) that can be represented (fit) by a straight line (green line).

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

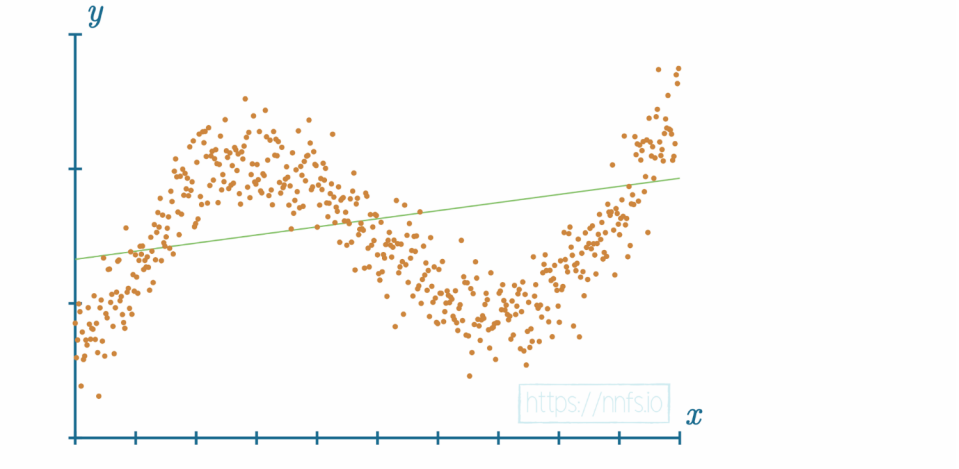

Fig 3.04: Example of data (orange dots) that is not well fit by a straight line.

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()
```

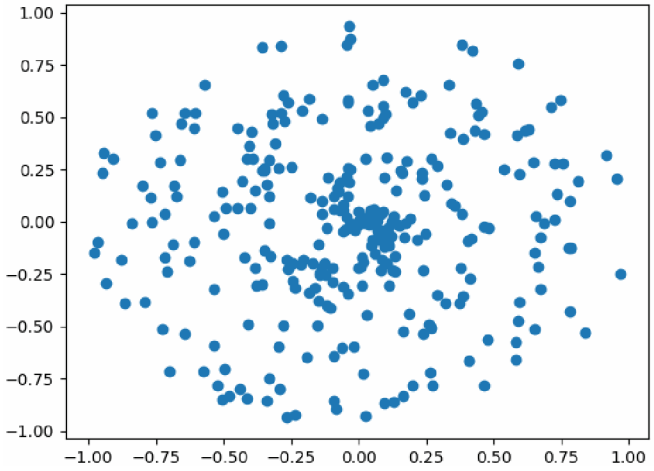

Fig 3.05: Uncolored spiral dataset.

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()
```

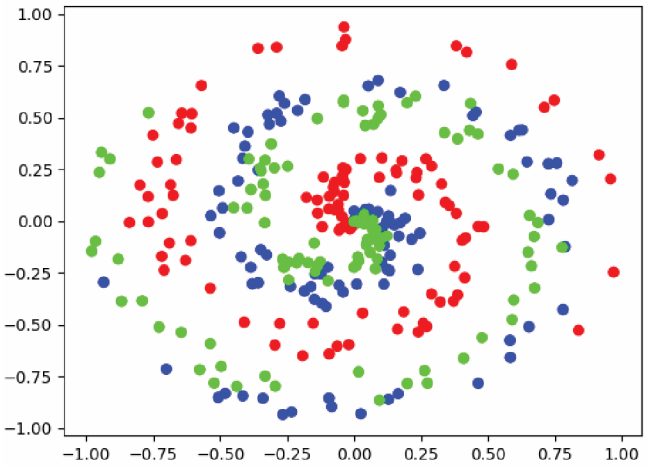

Fig 3.06: Spiral dataset colored by class.

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.

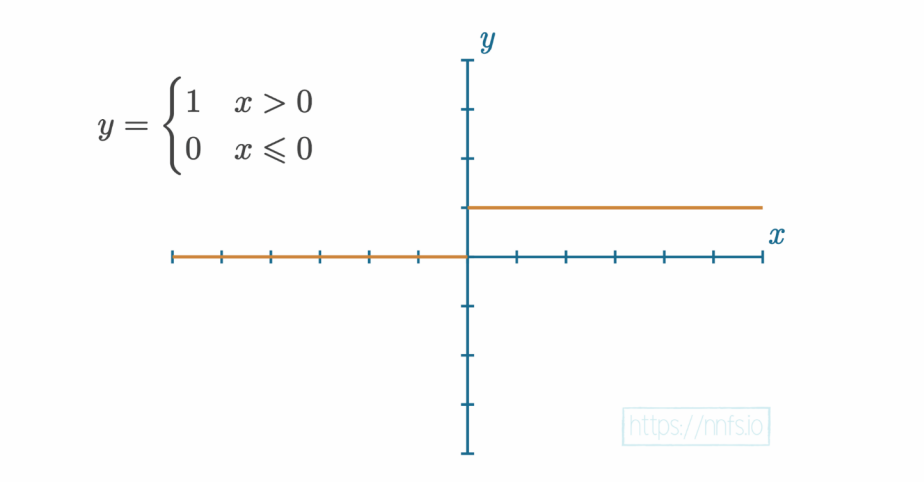

Fig 3.07: Graph of a step function.

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.

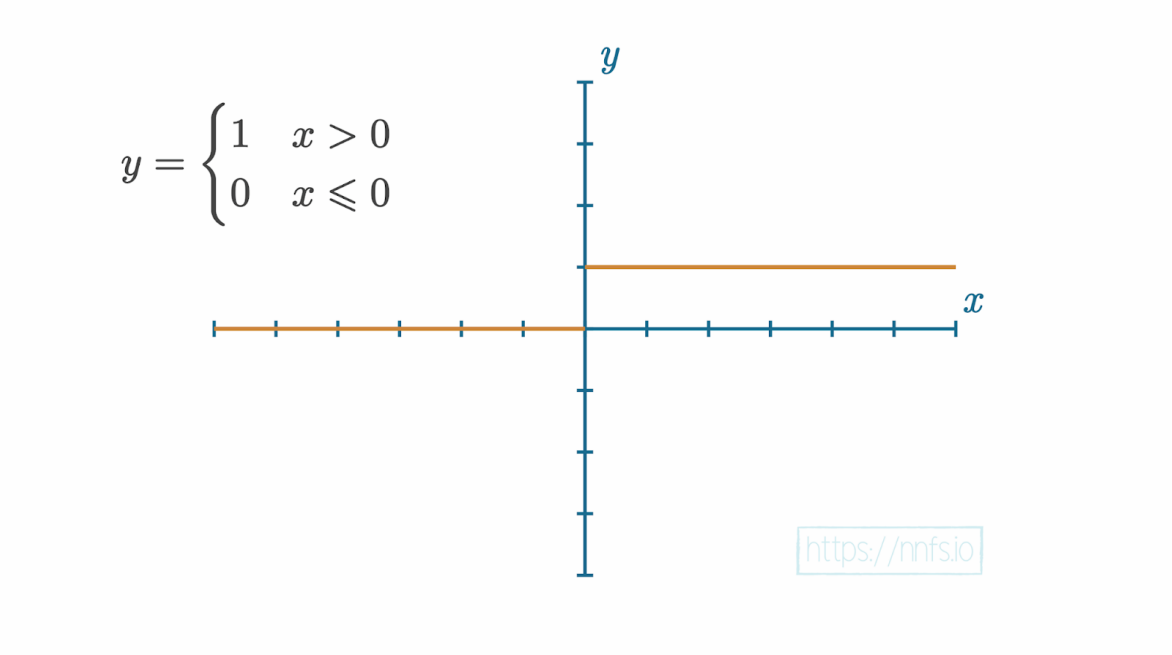

Fig 4.01: Step function graph.

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.

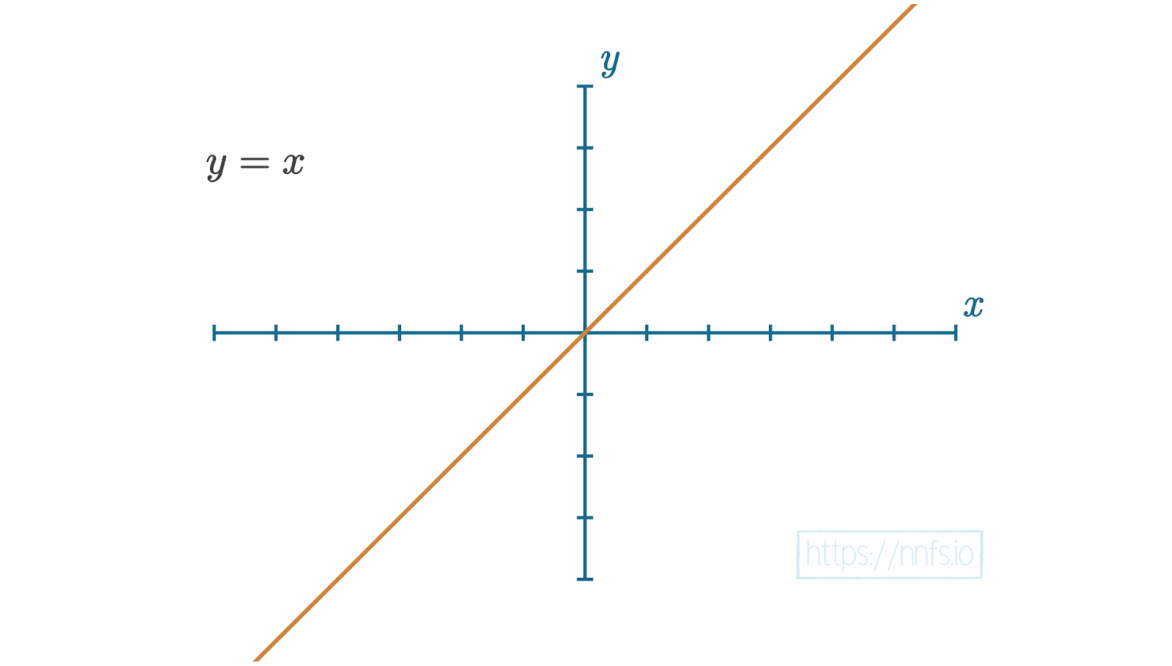

Fig 4.02: Linear function graph.

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:

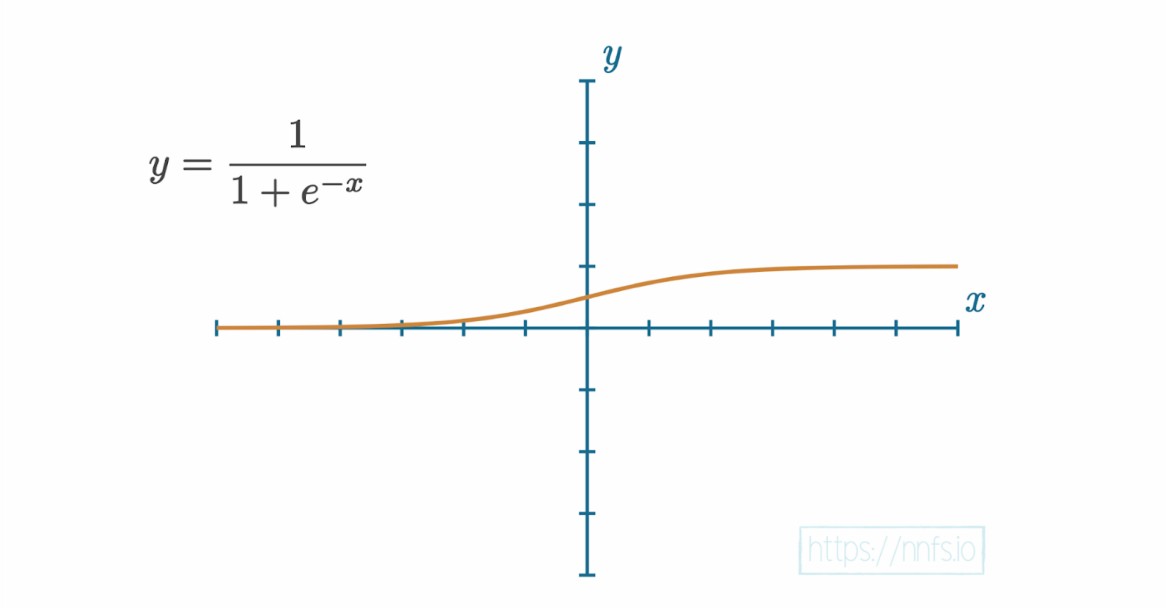

Fig 4.03: Sigmoid function graph.

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

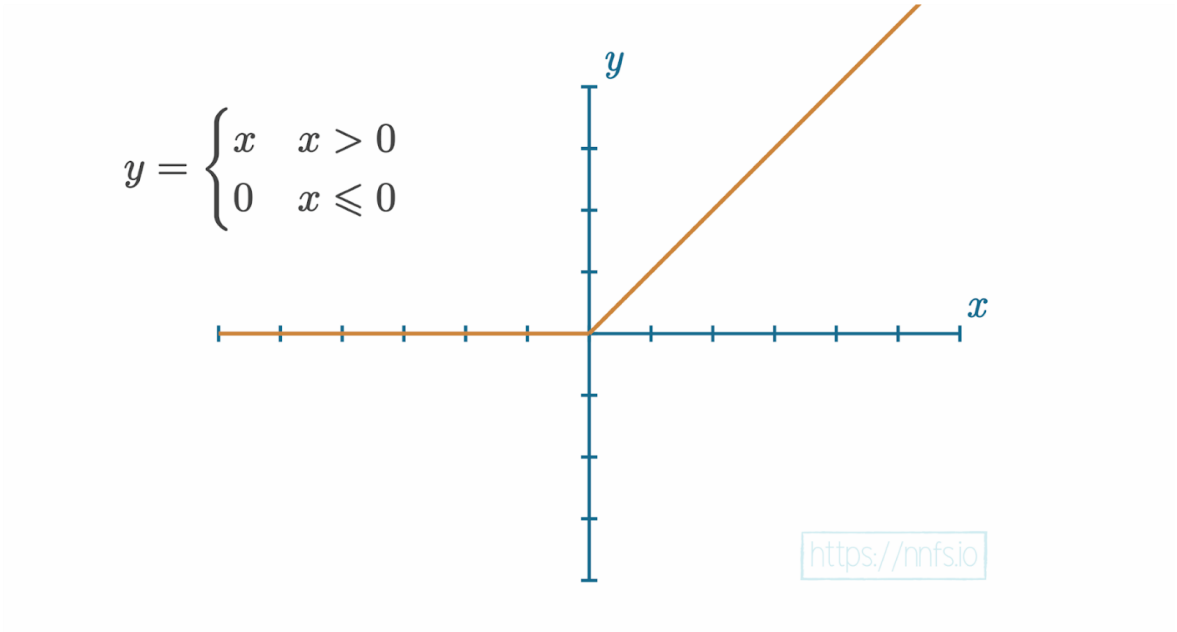

Fig 4.04: Graph of the ReLU activation function.

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:

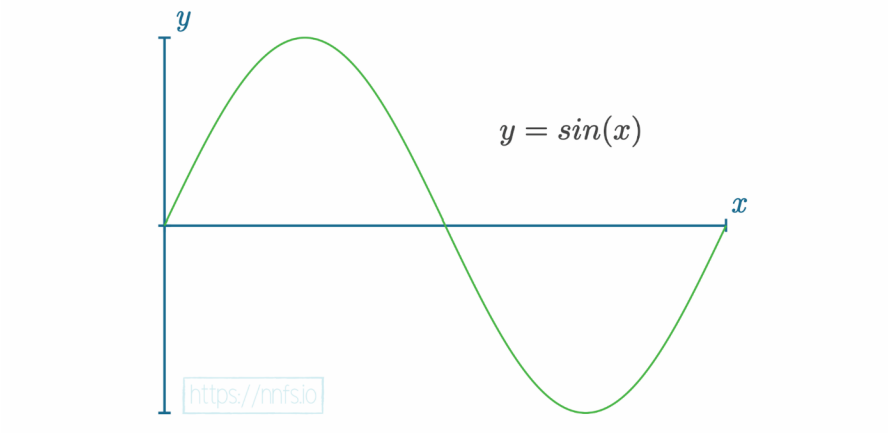

Fig 4.05: Graph of y=sin(x)

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:

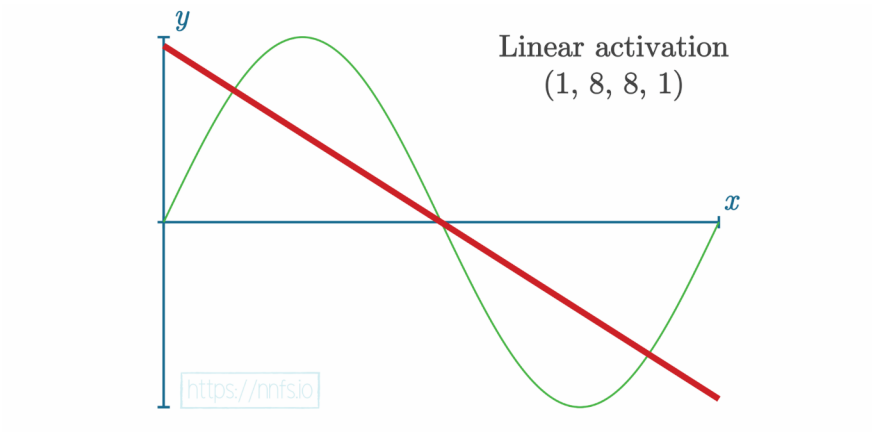

Fig 4.06: Neural network with linear activation functions in hidden layers attempting to fit y=sin(x)

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

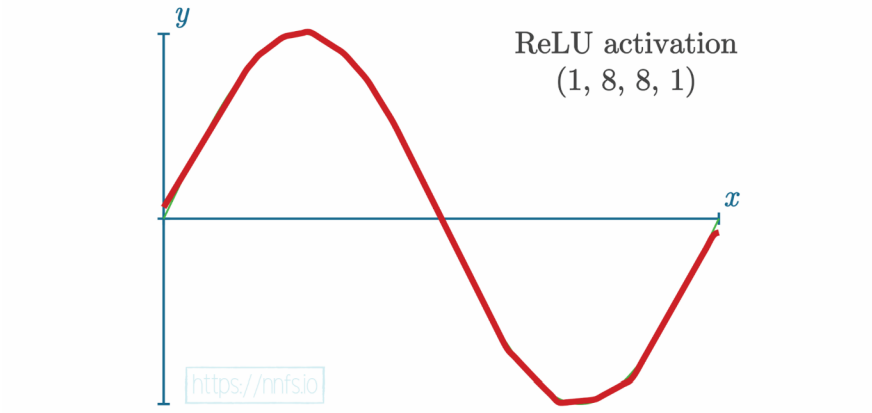

Fig 4.07: ReLU activation functions in hidden layers attempting to fit y=sin(x)

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:

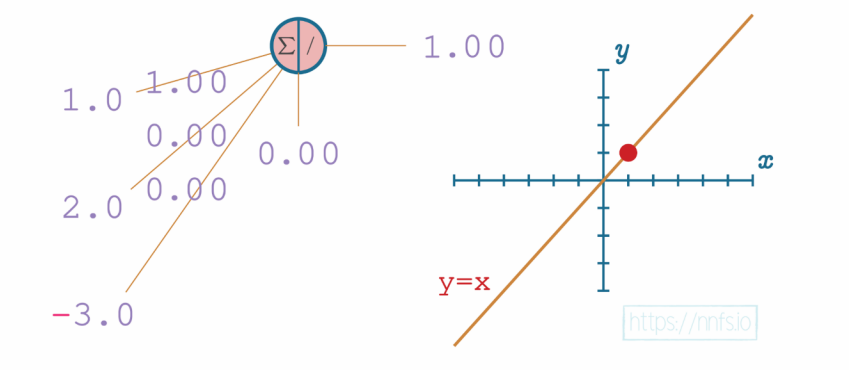

Fig 4.08: Example of output with a neuron using a linear activation function.

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

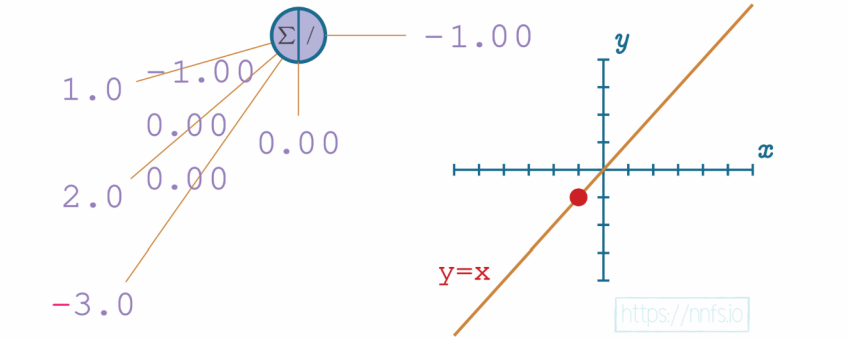

Fig 4.09: Example of output with a neuron using a linear activation function, updated weight.

And updating weights and additionally a bias:

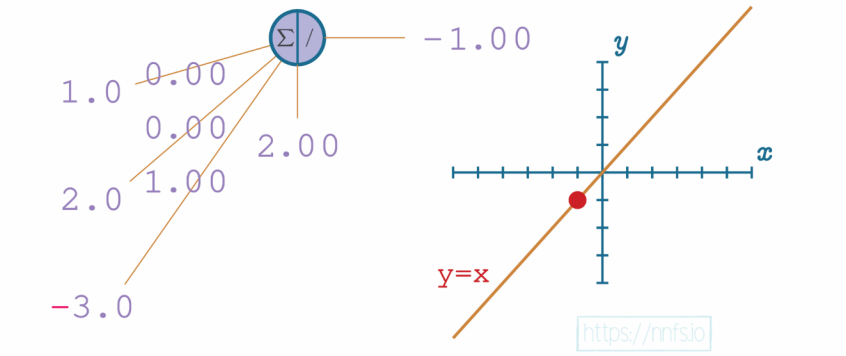

Fig 4.10: Example of output with a neuron using a linear activation function, updated another weight.

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:

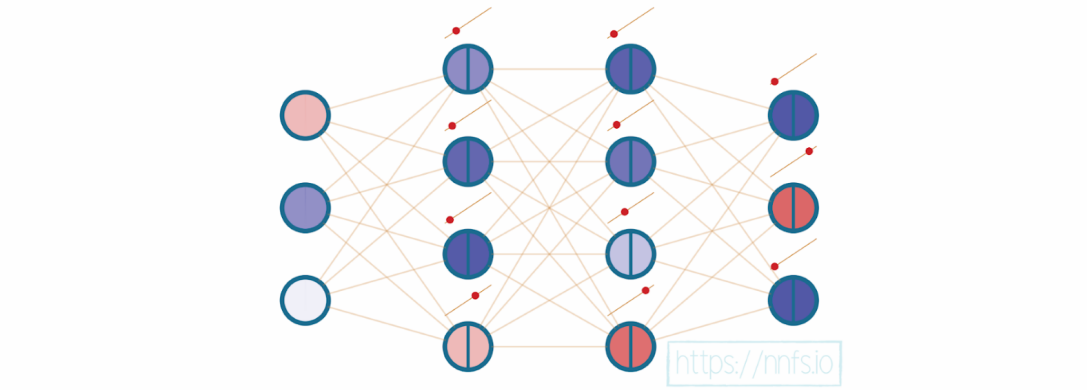

Fig 4.11: A neural network with all linear activation functions.

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:

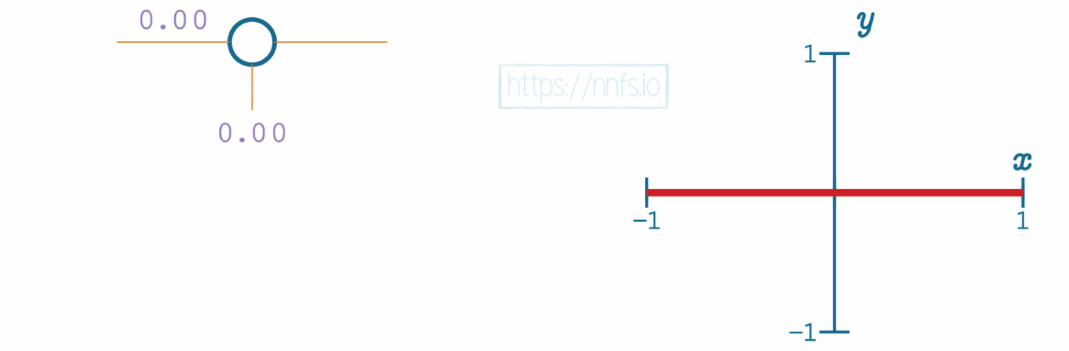

Fig 4.12: Single neuron with single input (zeroed weight) and ReLU activation function.

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:

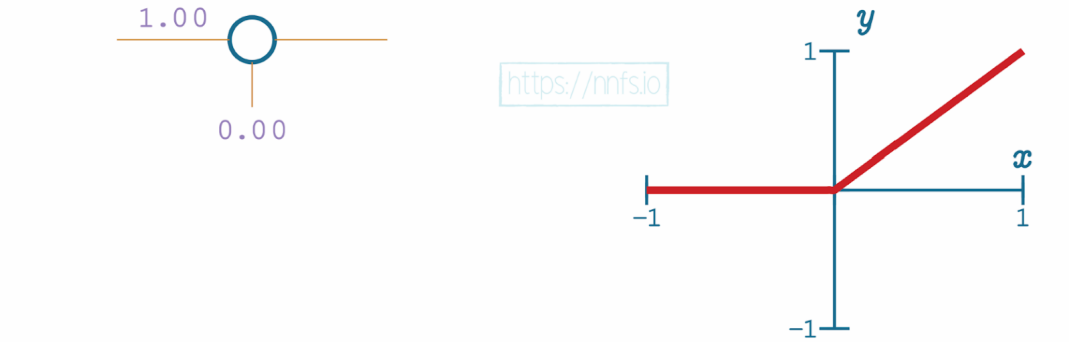

Fig 4.13: Single neuron with single input and ReLU activation function, weight set to 1.0.

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

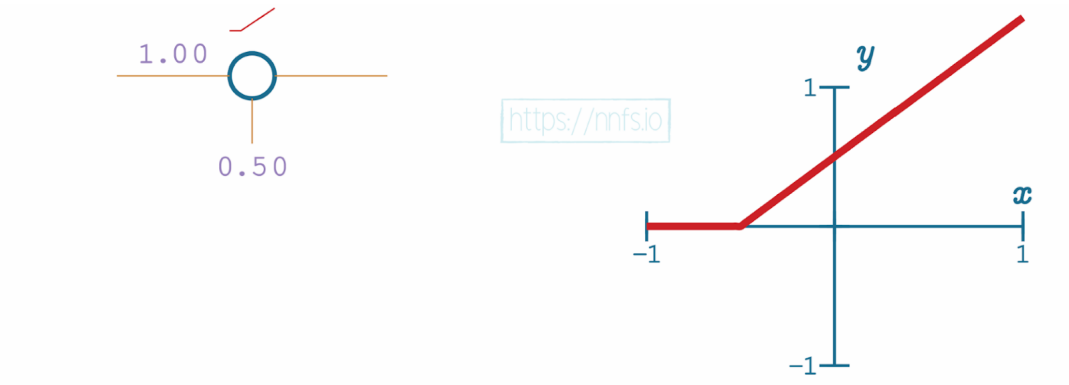

Fig 4.14: Single neuron with single input and ReLU activation function, bias applied.

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?

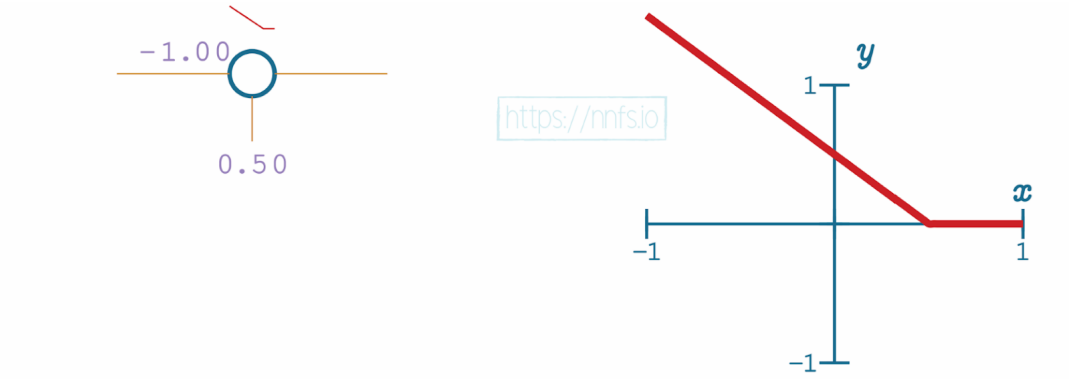

Fig 4.15: Single neuron with single input and ReLU activation function, negative weight.

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:

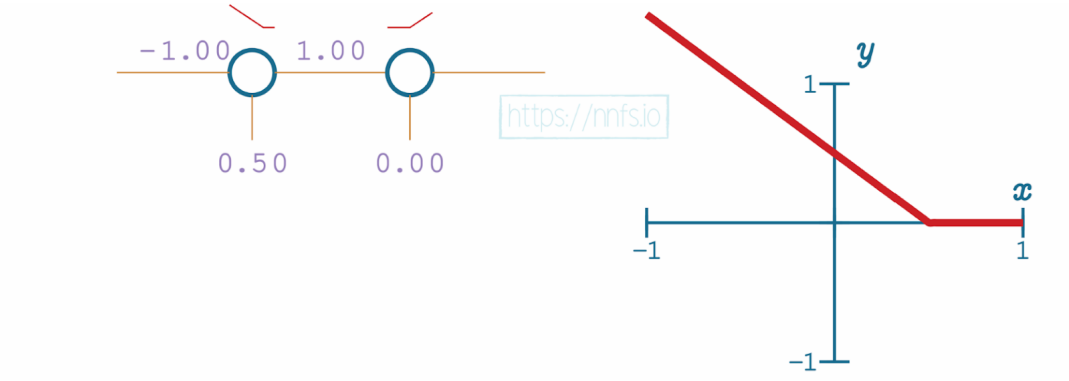

Fig 4.16: Pair of neurons with single inputs and ReLU activation functions.

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:

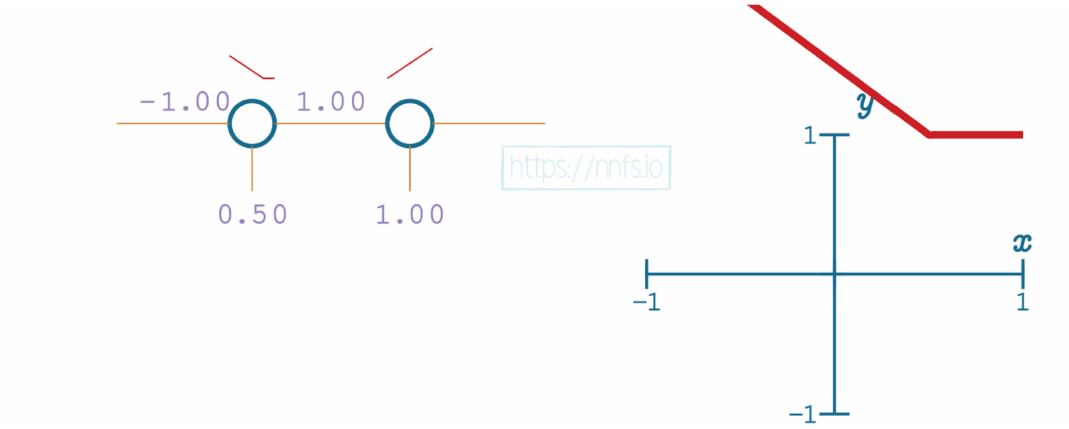

Fig 4.17: Pair of neurons with single inputs and ReLU activation functions, other bias applied.

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?

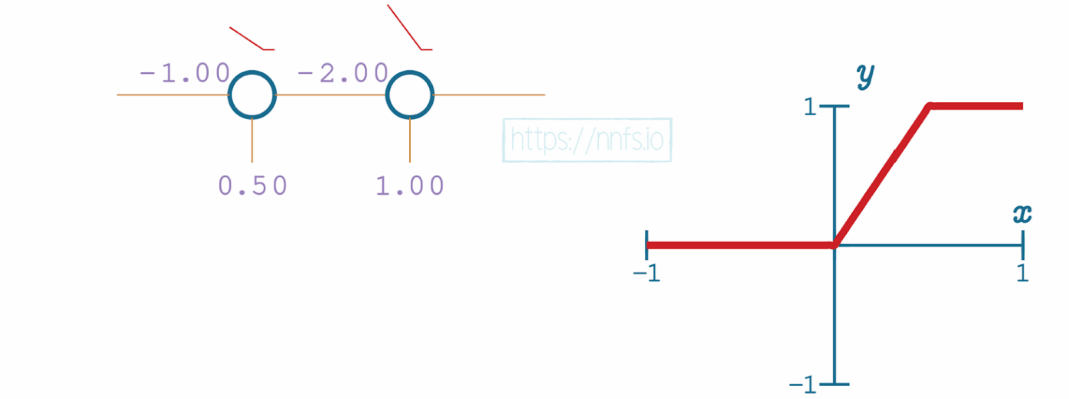

Fig 4.18: Pair of neurons with single inputs and ReLU activation functions, other negative weight.

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:

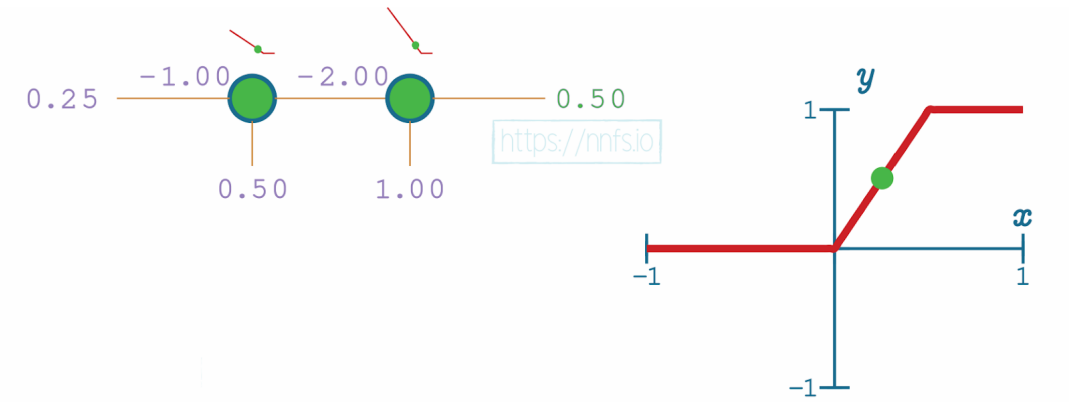

Fig 4.19: Pair of neurons with single inputs and ReLU activation functions, area of effect.

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:

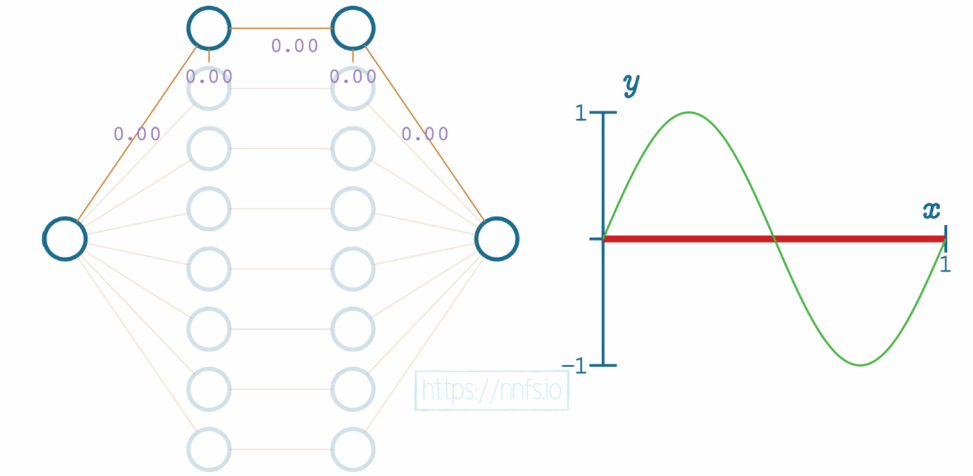

Fig 4.20: Hand-tuning a neural network starting with the first pair of neurons.

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:

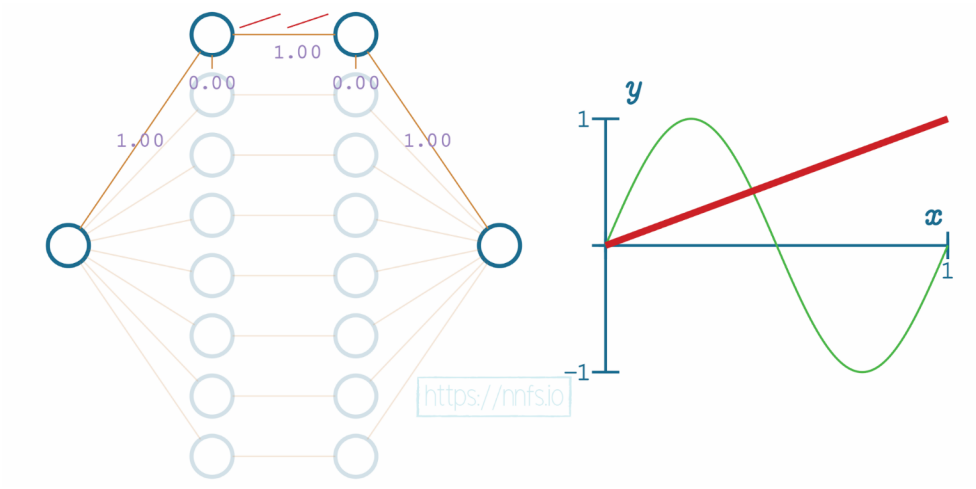

Fig 4.21: Adjusting weights for the first/top pair of neurons all to 1.

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:

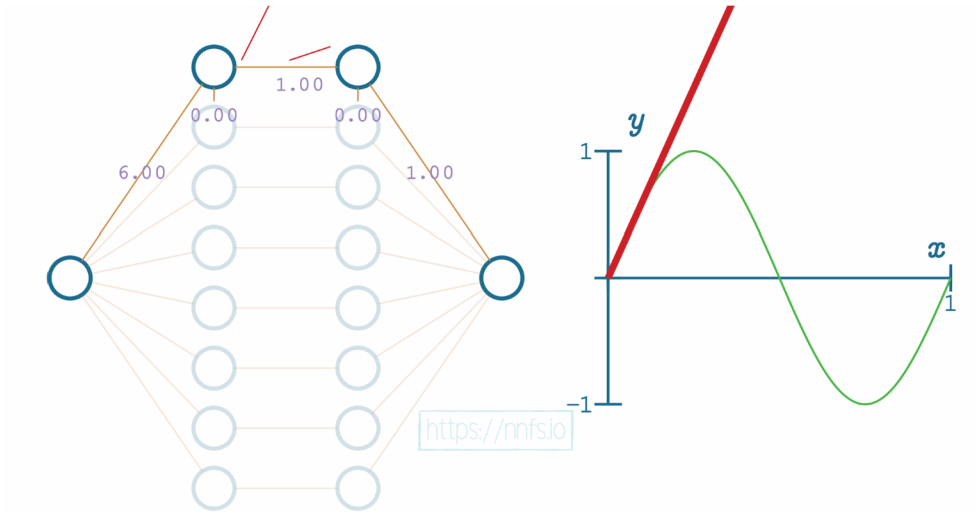

Fig 4.22: Setting weight for first hidden neuron to 6.

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*:

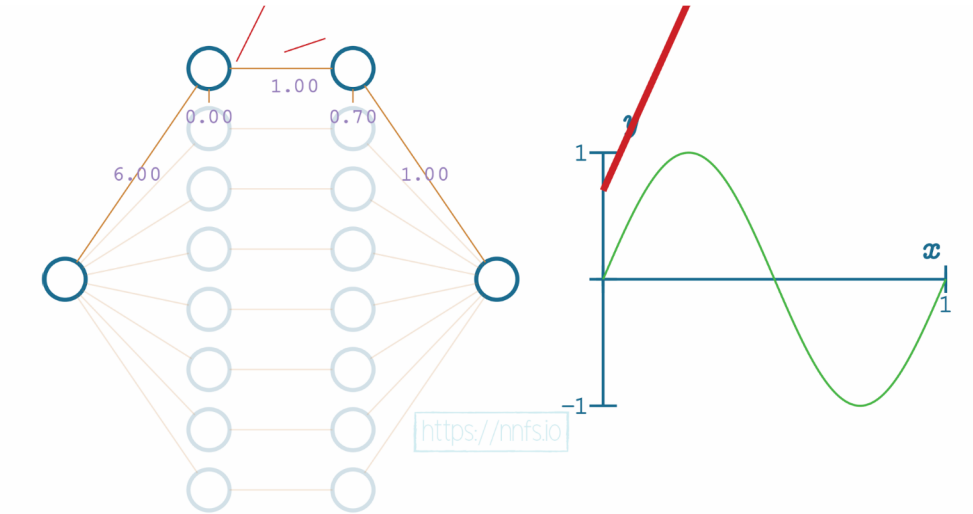

Fig 4.23: Using the bias for the 2nd hidden neuron in the top pair to offset function vertically.

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:

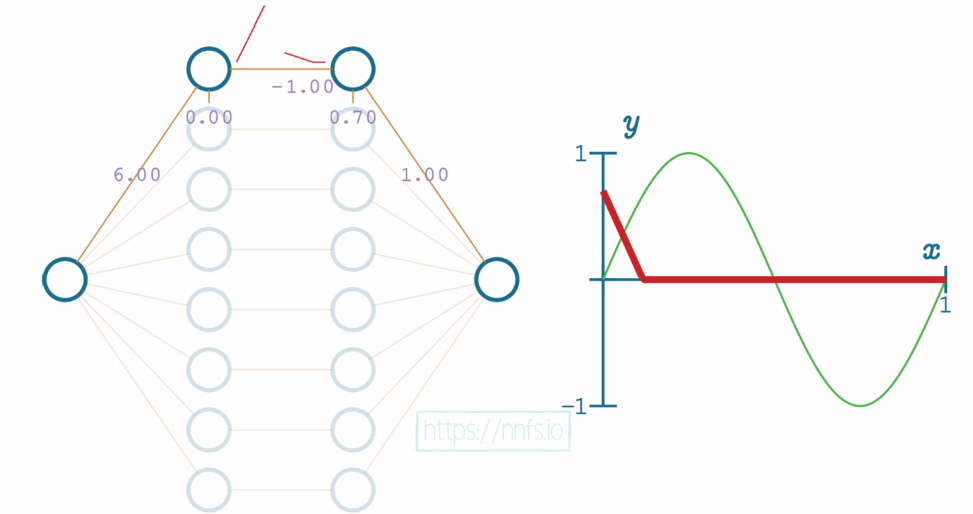

Fig 4.24: Setting the weight for the 2nd neuron in the top pair to -1.

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:

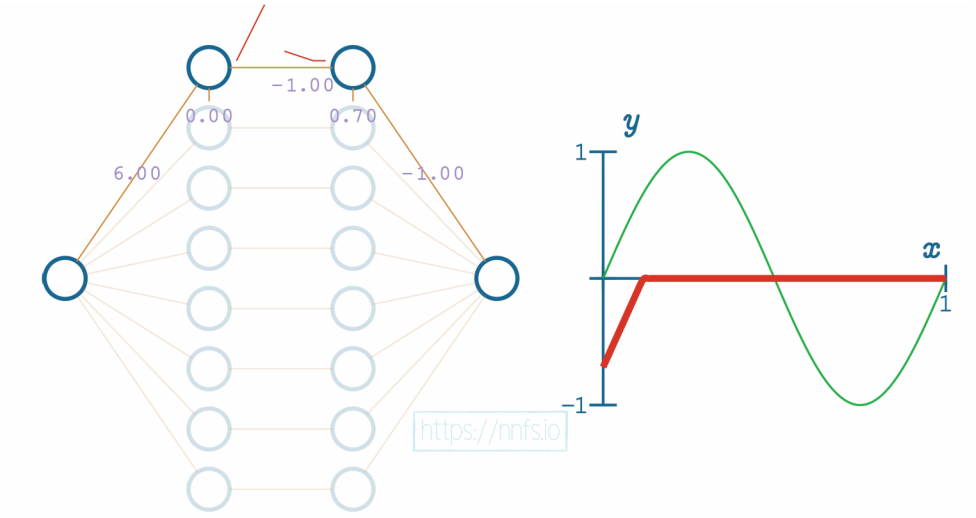

Fig 4.25: Setting the weight to the output neuron to -1.

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:

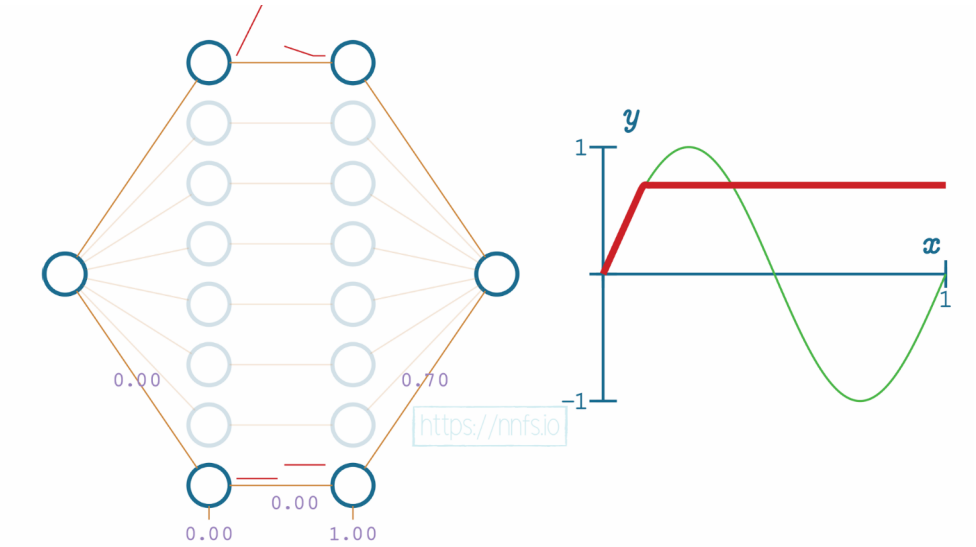

Fig 4.26: Using the bottom pair of neurons to offset the entire neural network function.

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:

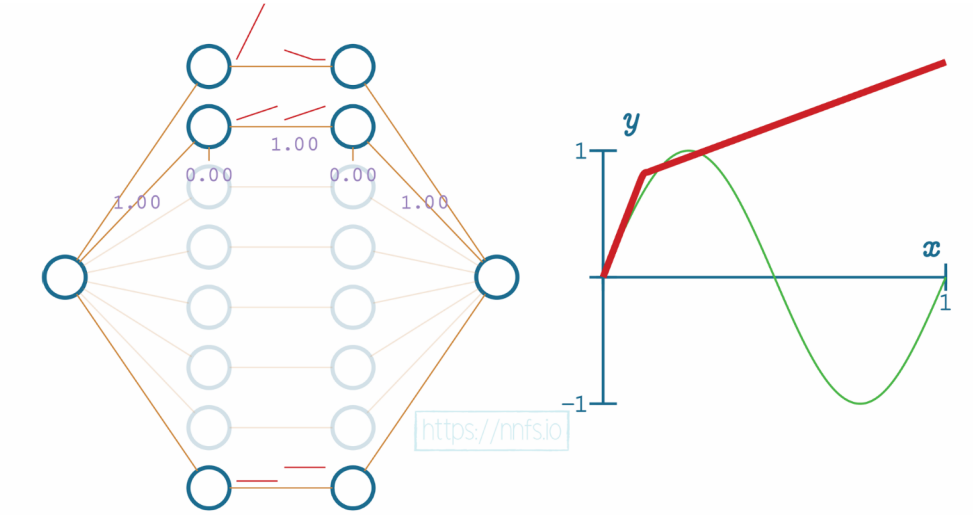

Fig 4.27: Starting to adjust the 2nd pair of neurons (from the top) for the next segment of the overall function.

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:

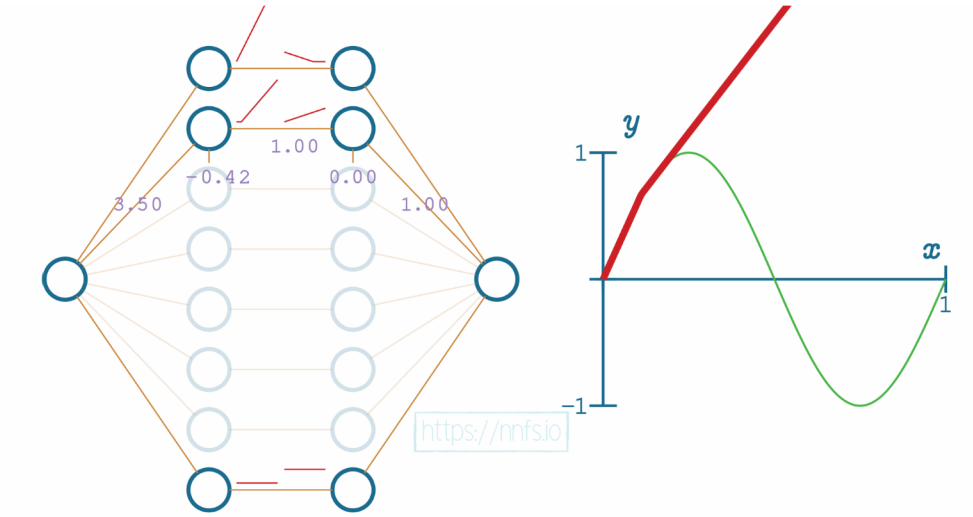

Fig 4.28: Adjusting the weight and bias into the first neuron of the 2nd pair.

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.

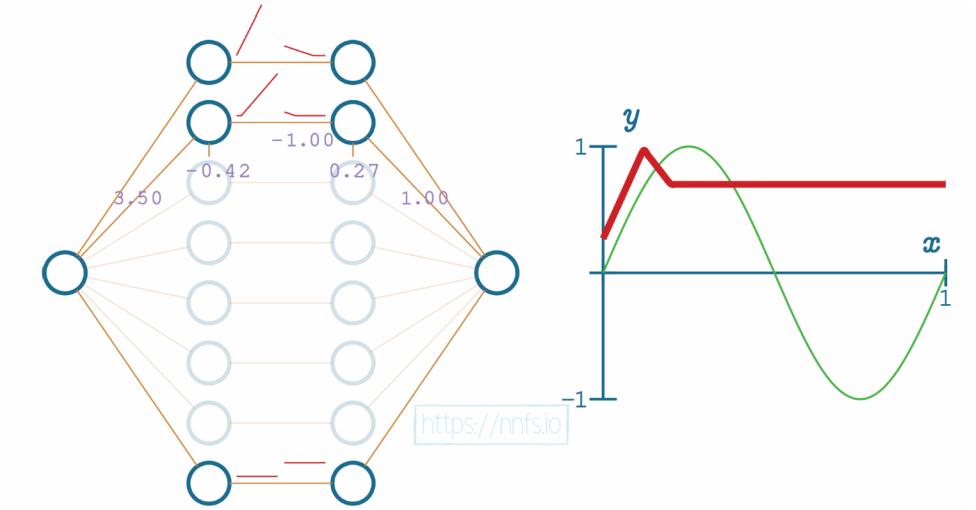

Fig 4.29: Adjusting the bias of the 2nd neuron in the 2nd pair.

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:

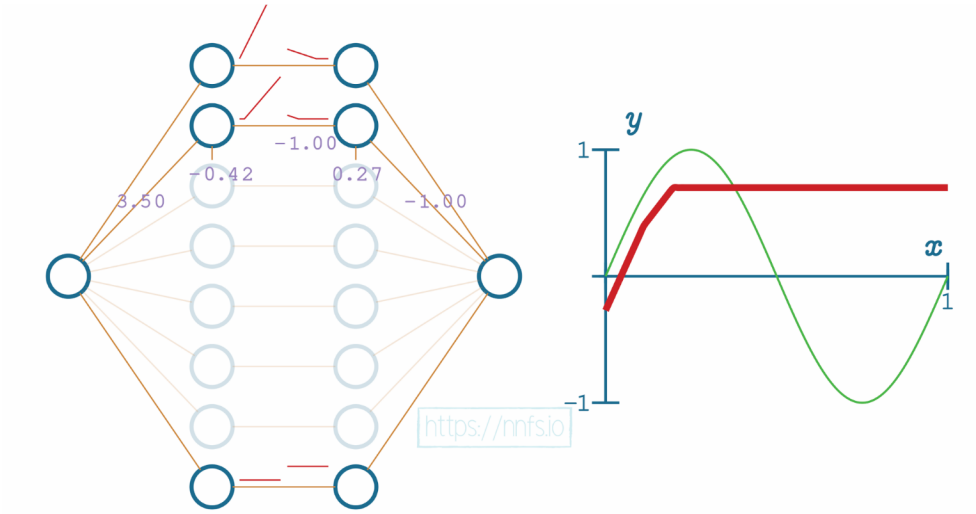

Fig 4.30: Flipping the 2nd pair's function segment, flipping the weight to the output neuron.

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

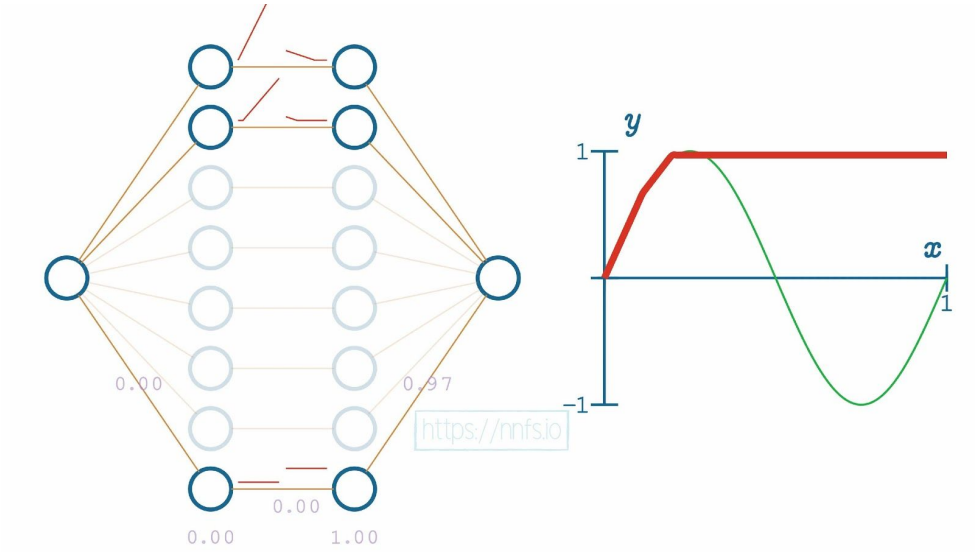

Fig 4.31: Using the bottom pair of neurons to adjust the network's overall function.

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:

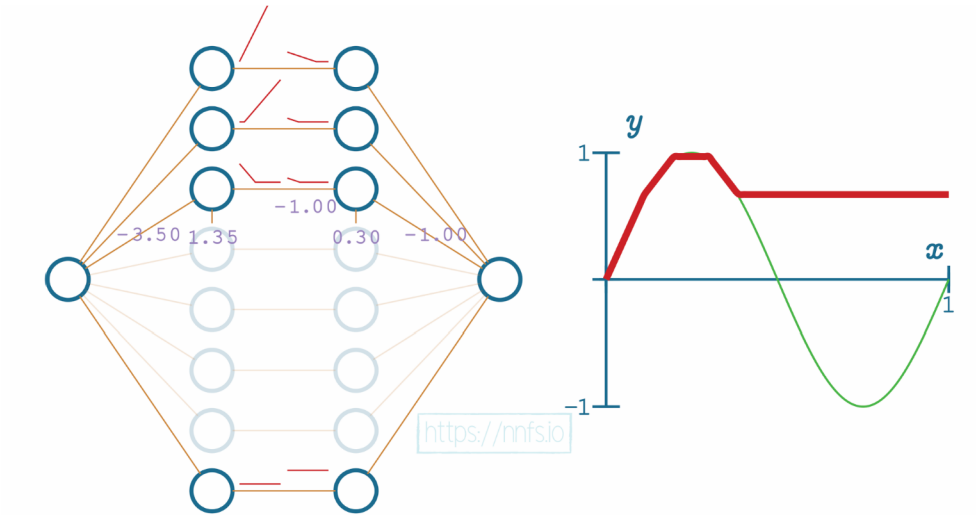

Fig 4.32: Adjusting the 3rd pair of neurons for the next segment.

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

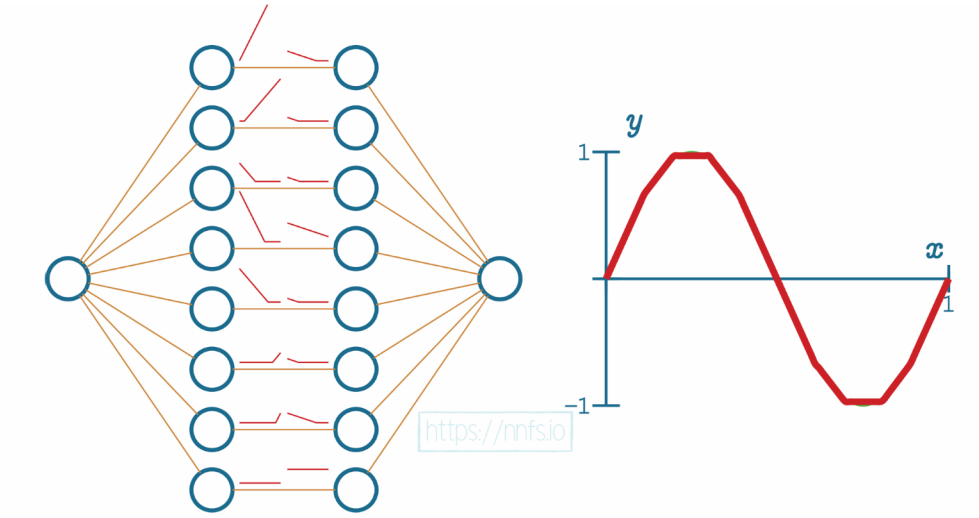

Fig 4.33: The completed process (see anim for all values).

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:

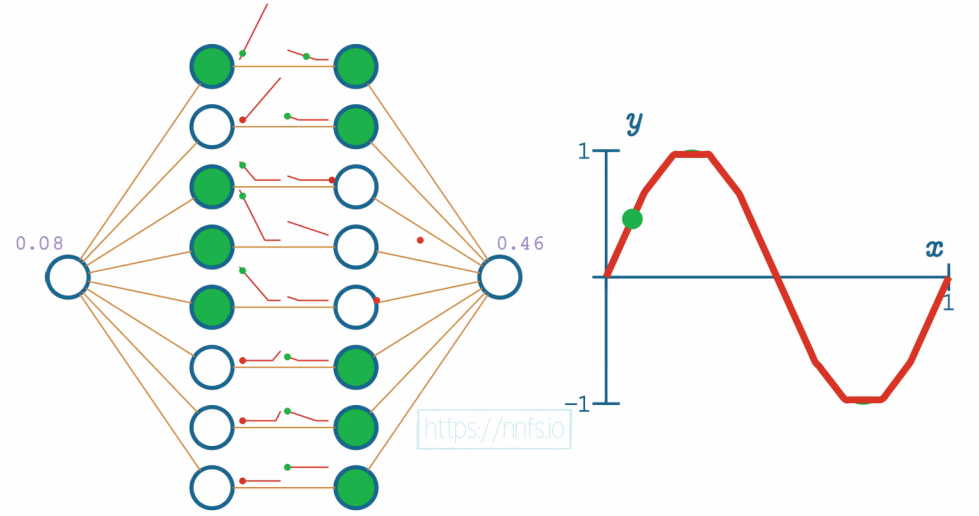

Fig 4.34: Example of data passing through this hand-crafted model.

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:

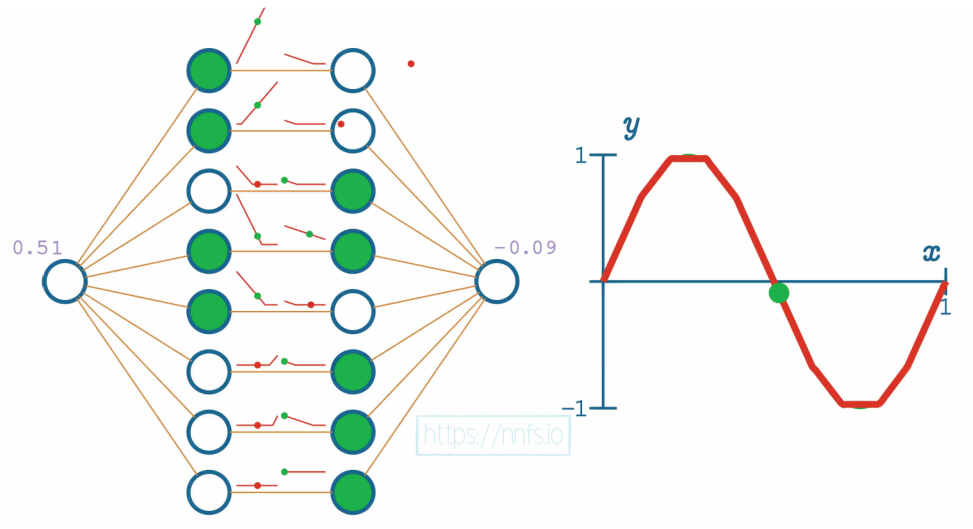

Fig 4.35: Example of data passing through this hand-crafted model.

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:

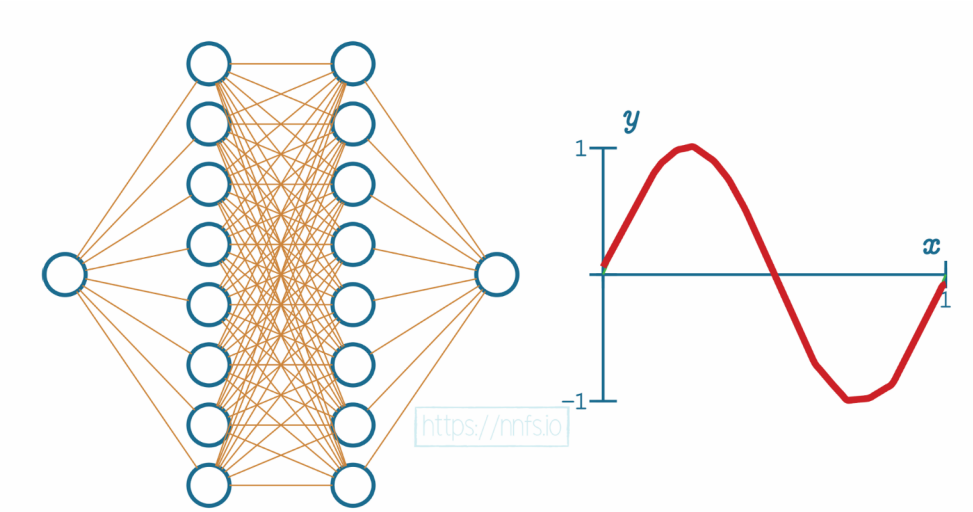

Fig 4.36: Example of fitment after fully-connecting the neurons and using an optimizer.

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:

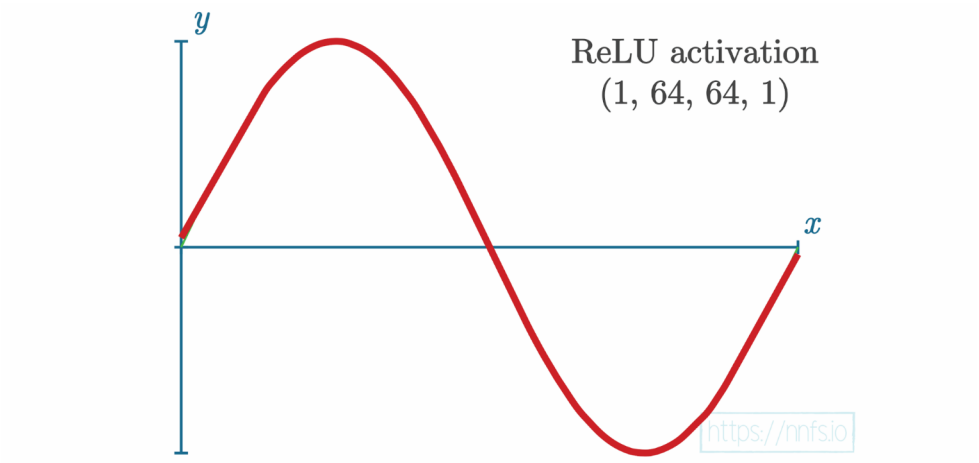

Fig 4.37: Fitment with 2 hidden layers of 64 neurons each, fully connected, with optimizer.

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:

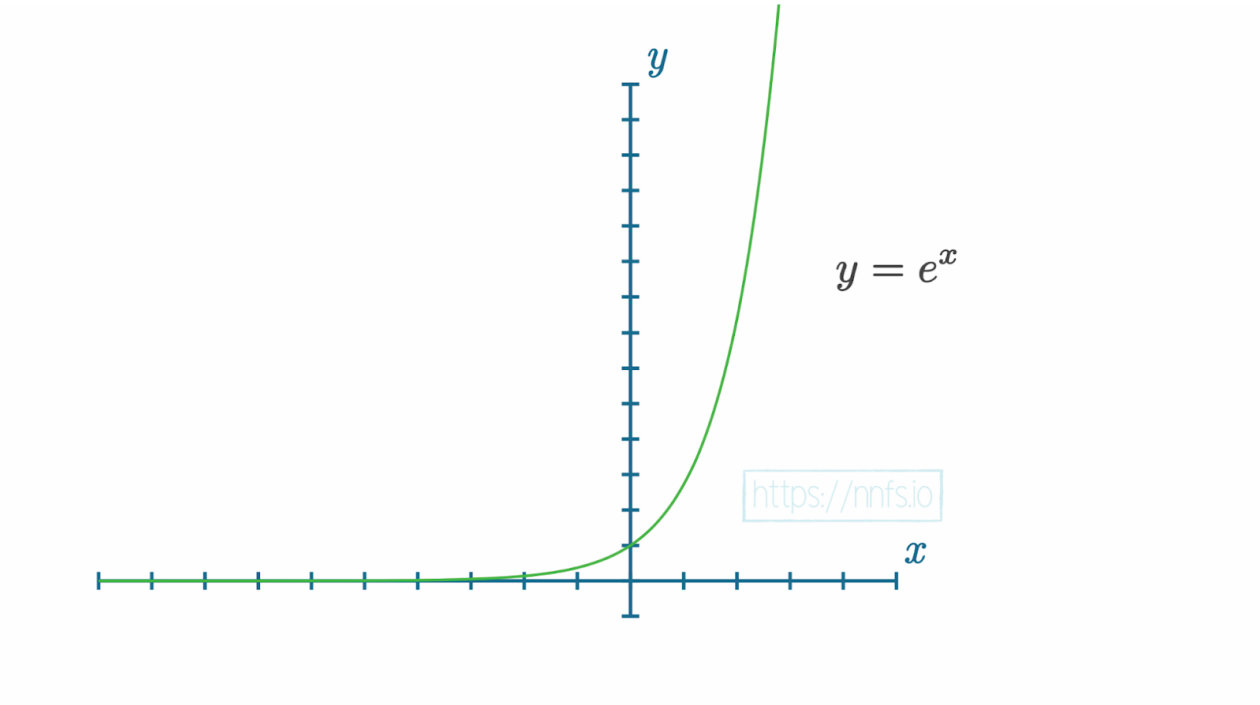

Fig 4.38: Graph of an exponential function.

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
```