NEURAL NETWORKS FROM SCRATCH IN PYTHON

By Harrison Kinsley & Daniel Kukiela

Feel free to post comments/questions by highlighting the text and clicking the comment button to the right. Looks like this: `

Do not resolve comments that are not yours.

PDFs are being distributed usually within **24 hrs** of order.

Links to chapters:

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.

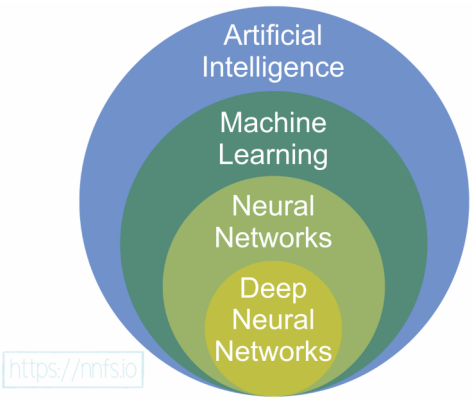

Fig 1.01: Depicting the various fields of artificial intelligence and where they fit in overall.

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.

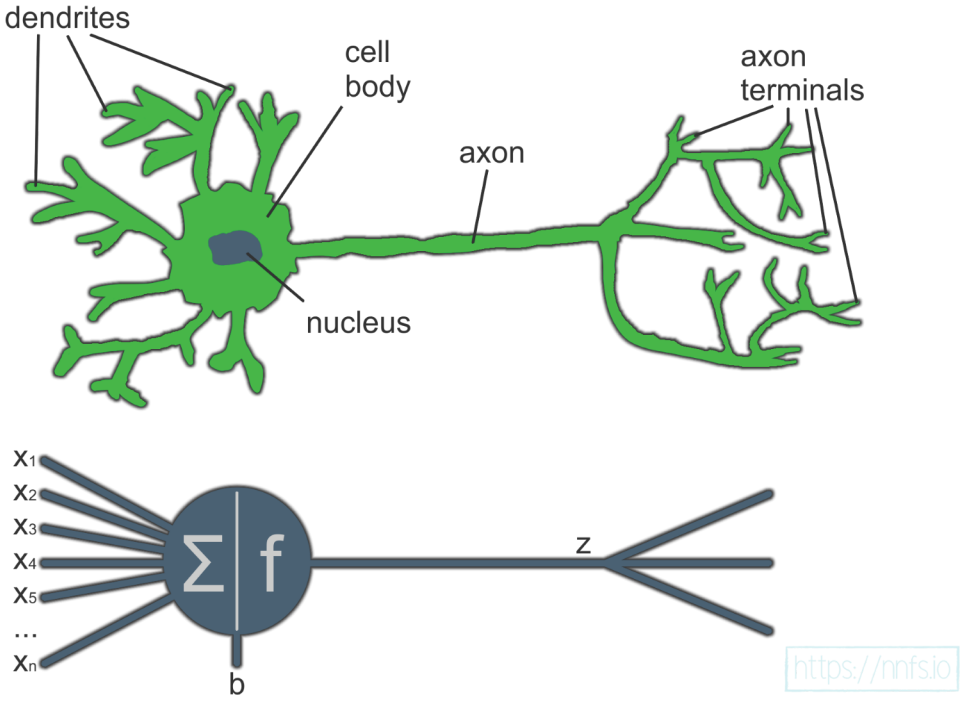

Fig 1.02: Comparing a biological neuron to an artificial neuron.

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.

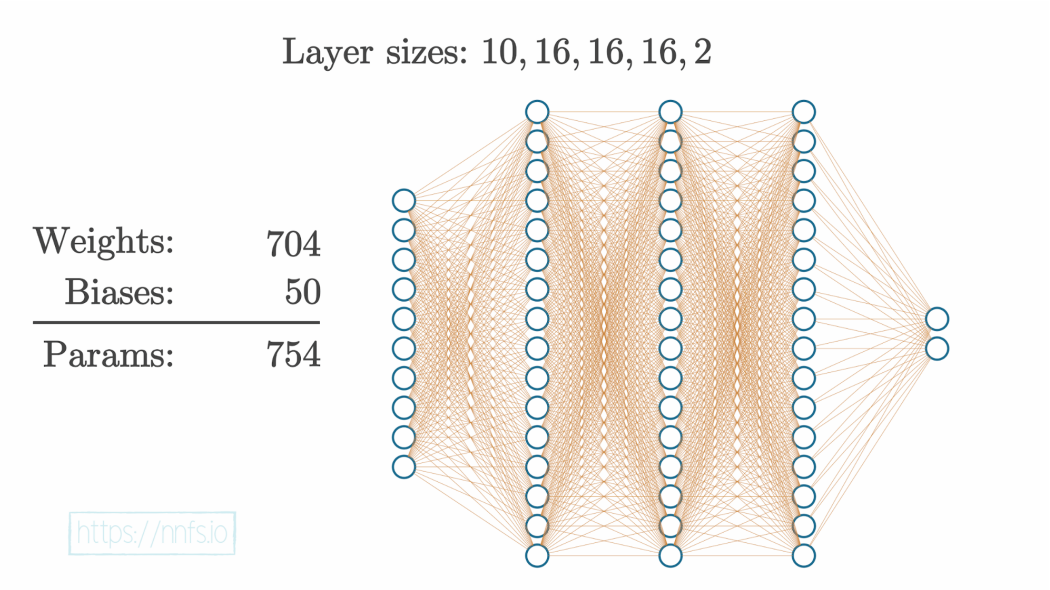

Fig 1.03: Example of a neural network with 3 hidden layers of 16 neurons each.

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:

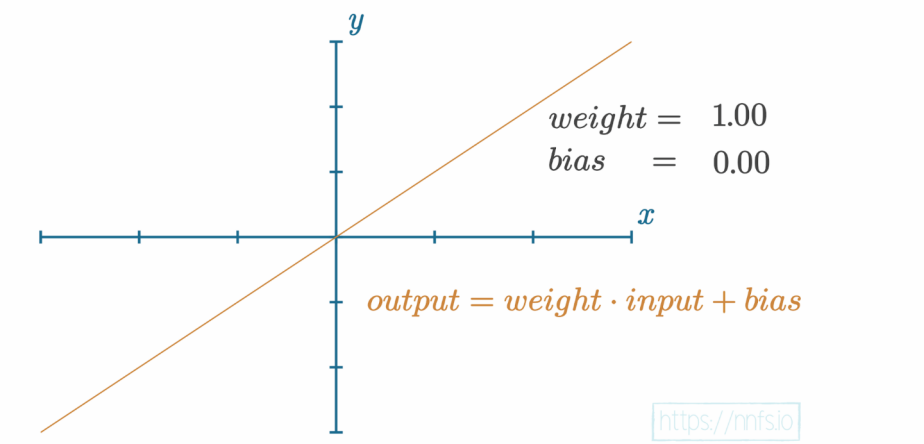

Fig 1.04: Graph of a single-input neuron's output with a weight of 1, bias of 0 and input *x*.

Adjusting the weight will impact the slope of the function:

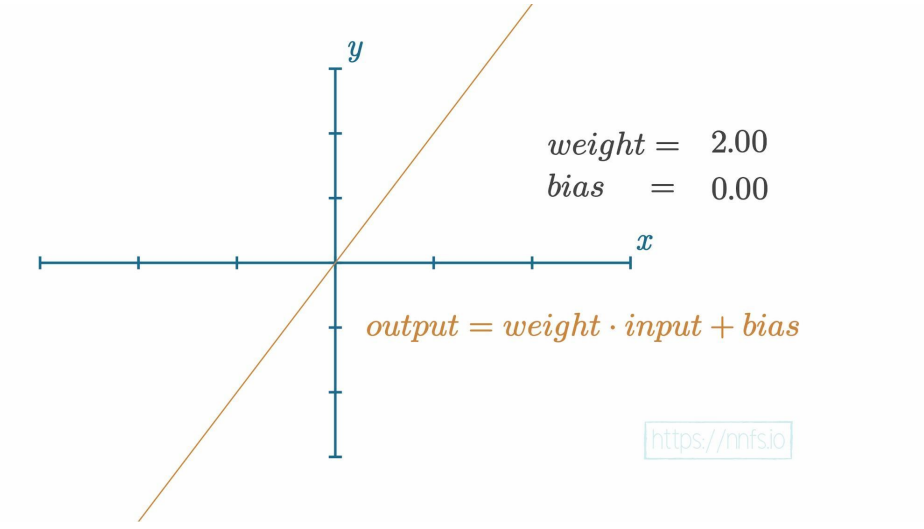

Fig 1.05: Graph of a single-input neuron's output with a weight of 2, bias of 0 and input *x*.

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:

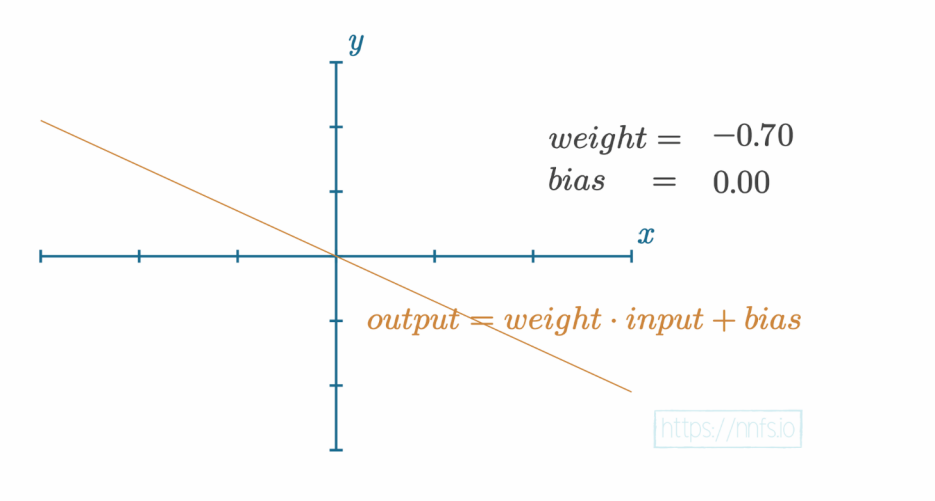

Fig 1.06: Graph of a single-input neuron's output with a weight of -0.70, bias of 0 and input *x*.

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:

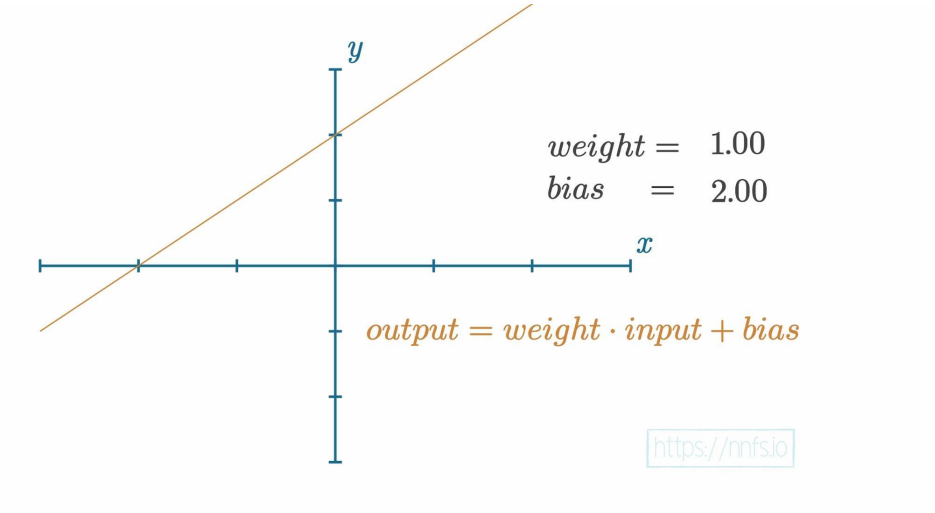

Fig 1.07: Graph of a single-input neuron's output with a weight of 1, bias of 2 and input *x*.

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:

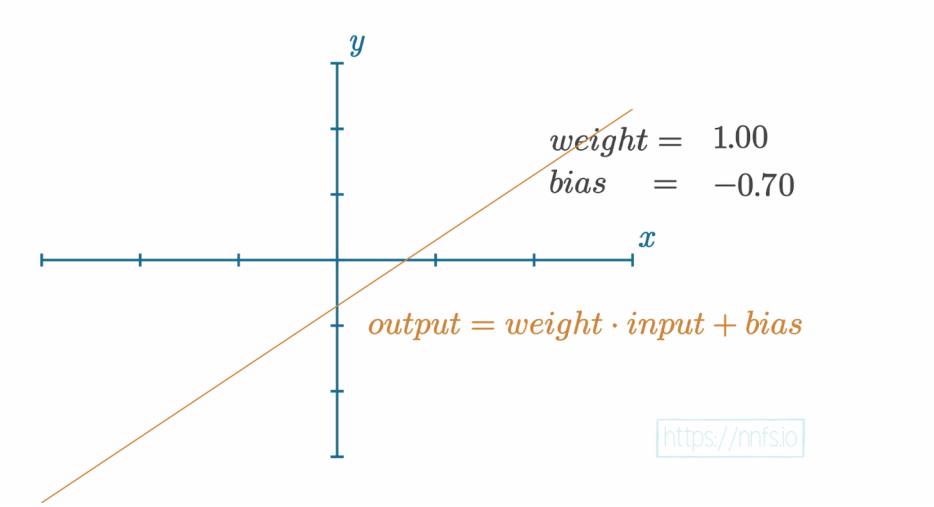

Fig 1.08: Graph of a single-input neuron's output with a weight of 1.0, bias of -0.70 and input *x*.

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 is 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.

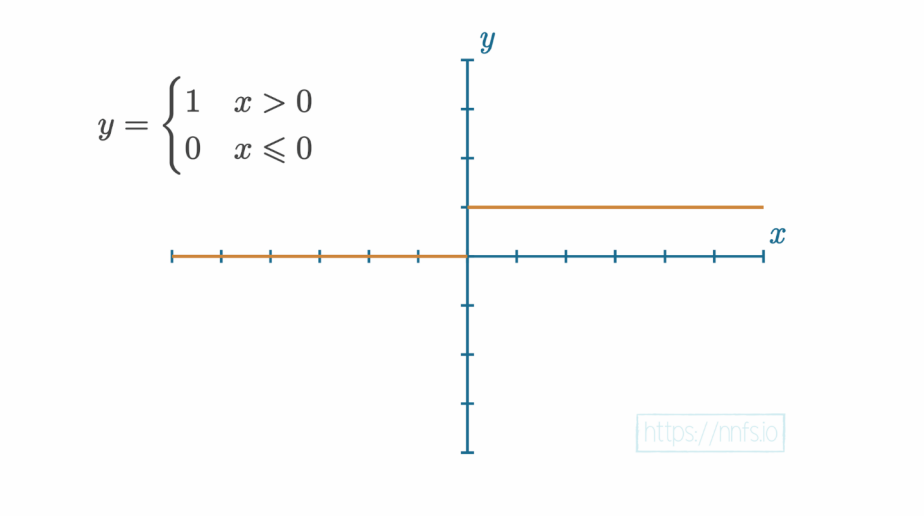

Fig 1.09: Graph of a step function.

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.

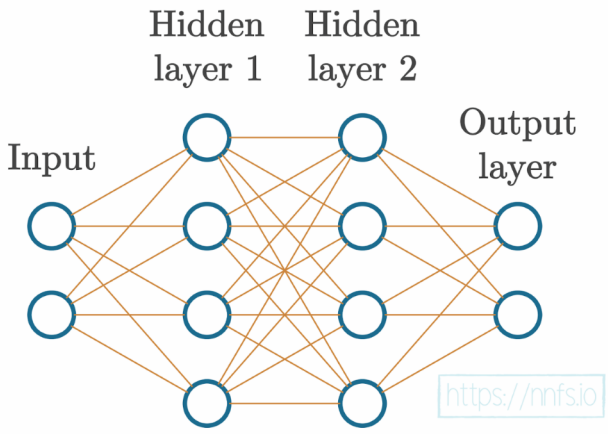

Fig 1.10: Example basic neural network.

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."

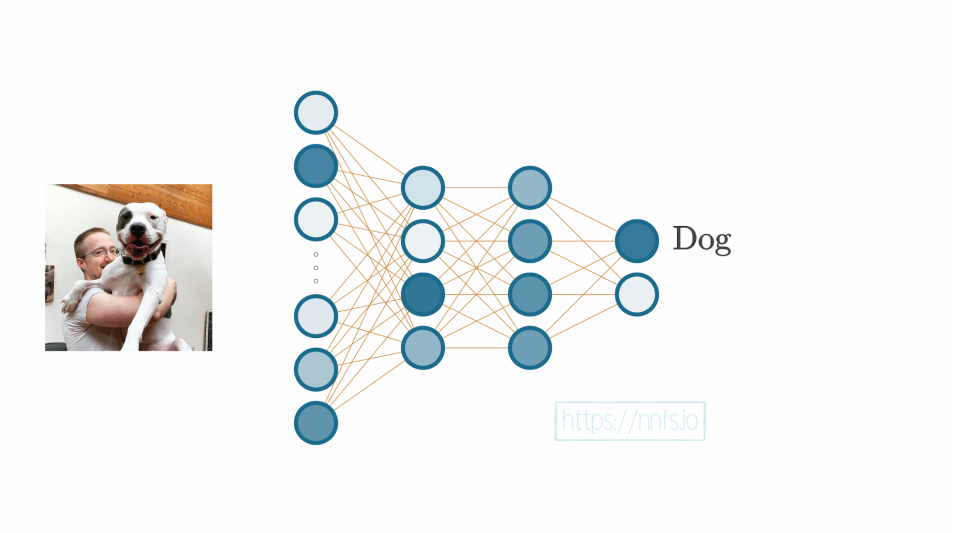

Fig 1.11: Visual depiction of passing image data through a neural network, getting a classification

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.

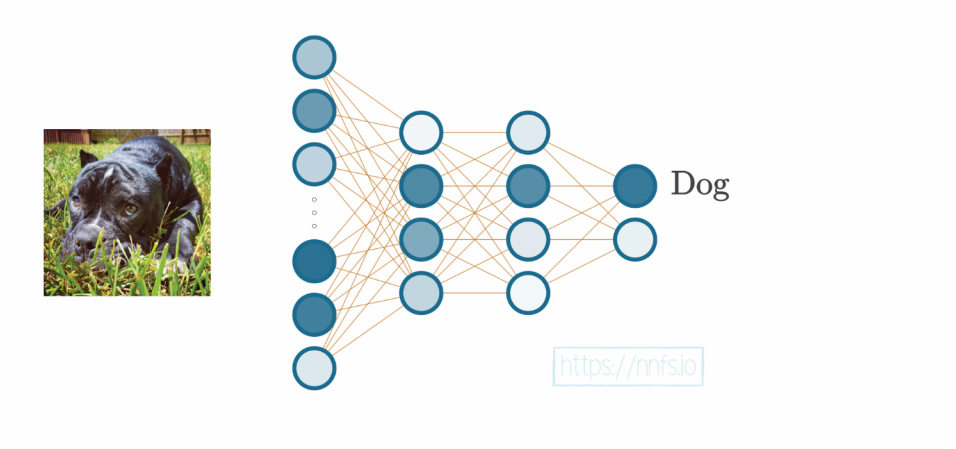

Fig 1.12: Visual depiction of passing image data through a neural network, getting a classification

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.