An Introduction to Neural Networks

Daniel Zack Brodtman

Supervisor: Dr. Marija Zamaklar

Declaration of Authorship

This piece of work is a result of my own work except where it forms an assessment based on group project work. In the case of a group project, the work has been prepared in collaboration with other members of the group. Material from the work of others not involved in the project has been acknowledged and quotations and paraphrases suitably indicated.

Contents

1 Introduction to Machine Learning

> Artificial intelligence is the new electricity.
>
> *Andrew Ng*

1.1 What is Machine Learning?

Machine learning (ML) is a subset of artificial intelligence (AI) which refers to software which becomes more accurate at modelling data without being explicitly programmed by a set of rules. A useful non-example is traditional chess engines. These AI machines were built by giving them the rules of chess and huge memories. They analyse board positions and give them ratings. After an opponent moves, the engine simultaneously runs many simulations of possible legal moves, and potential moves following these moves, to see what position the game could develop into. It then chooses the best move by ranking all of these possible outcomes. It does this repeatedly for every move. Some chess engines also have tables of opening and endgame move sequences. Chess engines have become highly skilled, repeatedly beating the best grandmasters. However, when the best chess engine, Stockfish, played against a machine learning based chess computer, Alphazero, in a 100-game match, it did not manage to win a single game and lost 28 [1]. Furthermore, Alphazero has also been used to master the game of Go, a game which has more possible board positions than there are atoms in the observable universe [2].

Machine learning software learns how to model data without being told anything about the relationships within the data. It is given training examples, from which it extracts which features of the data are most important in defining its distribution. The chess problem is useful for seeing this. Alphazero selectively analyses only the most relevant possible move sequences. It searches through just 80,000 positions per second during a chess match, compared to 70,000,000 for the Stockfish engine. This means it can select the best move with substantially less processing power. We call chess engines like Stockfish *weak AI* because they perform simple tasks. Alphazero is an example of *strong AI* because it can be applied to many different tasks with high performance.

1.2 Why is it Important?

In 2015, Google CEO Sundar Pichai said "Machine learning is a core, transformative way by which we're rethinking how we're doing everything". One year earlier they bought DeepMind, the company that created Alphazero, for $500 million. There are applications for machine learning in almost every industry. Amazon and Facebook have thousands of ML engineers working in highly specialised research teams with millions of dollars of funding. The US Air Force has partnered with the UK's Defence Science and Technology Laboratory to research machine learning algorithms' applications in the military. TikTok's use of the algorithms to suggest 30 second videos to people so that they stay on the app (and watch the lucrative advertising) has resulted in it having the 5th highest number of active users per day of any social media platform, overtaking Twitter and LinkedIn with ease [3].

Data Science is now offered as a degree at many of the top universities [4] as increasingly diverse companies build out their technology teams to avoid being swallowed by the Big Tech companies. With so much data being collected, machine learning offers a new, powerful tool for fast, deep data analysis. Data-driven decision making is often what allows companies to outperform their competitors. Quant trading firms which use advanced ML techniques and have just a few hundred employees are recording yearly profits of almost $1 billion [5]. For comparison, the investment bank Deutsche Bank recorded a pre-tax profit of $3.4 billion, but required 82,961 employees to do so [6]. Artificial intelligence and machine learning are becoming increasingly politically relevant (figure 1) as more traditional companies adopt machine learning and become technology companies.

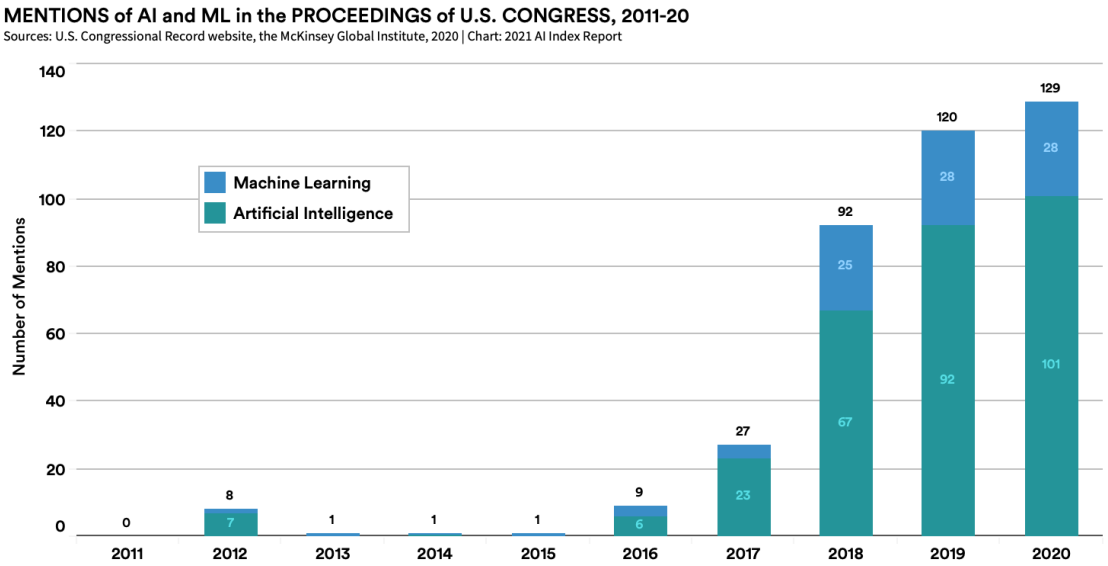


Figure 1: Mentions of machine learning in US Congress has increased 28x in five years [7].

Machine learning techniques are now relatively accessible to non-mathematicians. However, just because it is possible to put a dataset through a machine learning program, it doesn't mean one should. Understanding how ML programs work is important for selecting which technique is appropriate for a problem. This thesis will take a deep look at neural networks, one prominent technique in machine learning, and offer an introduction to their use in the field of image classification.

1.3 Aims and Motivations

In this thesis we delve into the mathematics behind a subsection of machine learning: *neural networks*. One of the key technological developments of the modern world, the first working algorithm for a neural network was posed by Ivakhnenko and Lapa in 1967. The purpose of neural networks is to mimic the brain's capacity for learning by taking in information and deducing patterns to model a dataset. To correctly model complex datasets it is often necessary to use an amount of historical data that is far greater than the human brain can handle in any reasonable time. Computers' ability to process many-dimensional data sets in milliseconds is where the advantage of neural networks is demonstrated. Because computers can analyse such vast datasets, they can spot hidden patterns and relationships within a dataset, helping us to improve the data-based decisions we make. This could be analysing a person's medical history to predict their risk of serious disease, or more complex versions of this such as analysing photos of people's skin to detect cancer. By taking advantage of advances in processing power, neural networks and deep learning have been applied to many industries like the health sector with considerable success in the last 20 years. The reason Alphazero could be applied successfully to different games with different rules is that it is a neural network. It learns how to win games by being fed data about thousands of previous games and playing against itself repeatedly to learn which strategies are most effective in different scenarios. Case studies like Alphazero have shown the potential value of neural networks, and since 2010 research into them has been increasing exponentially (figure 2).

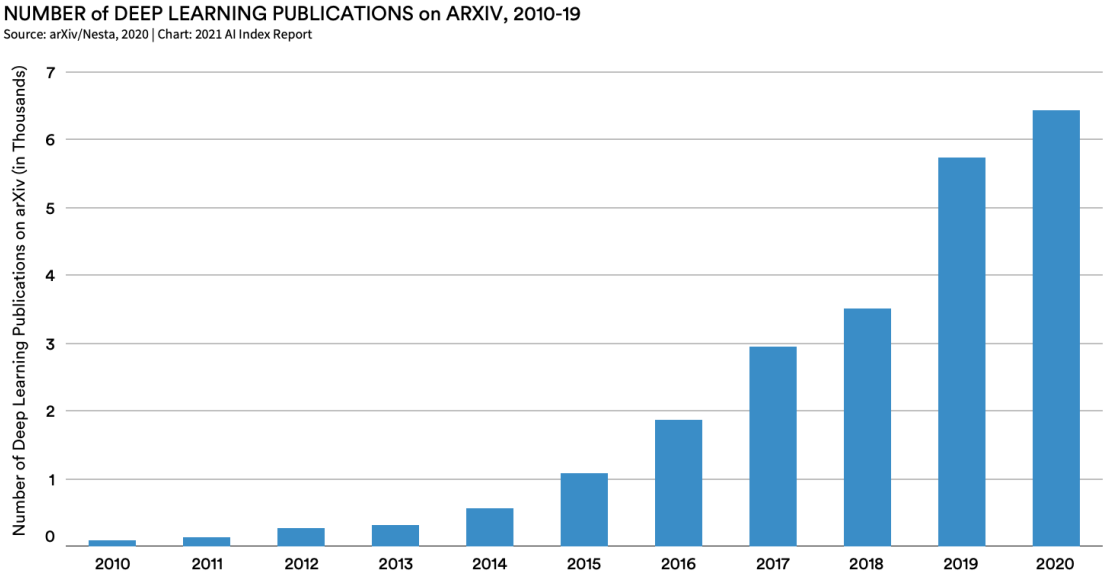


Figure 2: Deep learning papers grew sixfold in 5 years [7].

Neural networks learn by analysing training examples. A *training* set is passed through the network to produce an output and this is repeated for many iterations. They have a set of parameters that become more and more specific to the dataset as it is repeatedly iterated through the network, until the best parameters have been learnt. This is the crucial concept behind neural networks. We discuss the construction and training of neural networks before looking at the methods used to evaluate and improve them. In the final Chapter, we will see a new class of neural network which has been particularly effective for computer vision. We will use this to create our own network which will detect COVID-19 by looking at a patient's lung X-ray. Throughout the thesis we will construct various networks to demonstrate the concepts that are being discussed. For this, we will use the Fashion MNIST dataset, a common benchmark of machine learning.

1.4 The Fashion MNIST Dataset

Throughout this thesis we will discuss the mathematics that allows neural networks to work. We will also use real-world data to demonstrate how these ideas work in practice. We will only look at supervised learning scenarios. This is when the datasets we are using are labeled. Examples of this are a set of pictures where we know what is in the picture. We choose this subsection of machine learning in order to focus on making predictions about new data, such as predicting disease from X-ray images. To illustrate the concepts discussed in this paper we will be using the Fashion MNIST dataset (figure 3). It is a set of 70,000 images of clothing items. Each image has 28x28 greyscale pixels and there are 10 categories. The Fashion MNIST dataset is often used as a benchmark in machine learning research to test a neural network because of its large size and simple visualisation. It follows the same structure as the popular MNIST dataset of handwritten numbers. It was built by Zalando to be a harder alternative for researchers to use as achieving accuracies of 97%+ on the standard MNIST dataset was becoming too easy and the clothing images were more similar to real-world problems of computer vision than the handwritten numbers.

Figure 3: Some example images from the Fashion MNIST dataset

1.5 Notation, Definitions and Code

Throughout this thesis we will follow the notation from Andrew Ng's Standard Notation for Deep Learning from his Coursera lectures [8] which were extremely helpful to the development of this thesis. This is provided in Appendix A for reference and will be useful to look at throughout the reading of this thesis.

We will also frequently refer to the *accuracy* of a network. This means the number of correct predictions out of the total number of predictions made when a network has been tested on a group of test examples. We also frequently refer to the *features* of a dataset. This is sometimes referred to as the variables of a dataset and refers to the measurable pieces of data. For example, the features of a dataset which gives house prices would be (for example) floorspace x_1, garden size x_2 and distance to centre of town x_3. In the case of images like in the Fashion MNIST dataset, a feature is the colour measurement of each pixel. So for Fashion MNIST where each image is of size $28 \times 28 = 784$ pixels, there are 784 features.

The final notice to make here is that we will include code snippets at the ends of Chapters 3, 4 and 5. These are to demonstrate how the concepts we are discussing work in practice and we make substantial use of the Tensorflow library [9] and the Keras interface [10], with the programming done in Python [11]. The full code is available in a GitHub repository created for this project [12]. Additionally, all images, diagrams and graphs were made by me unless stated.

2 Structure of a network

In this Chapter we talk about the building blocks of neural networks. We begin with a look at the essential component of a network, the node, before zooming out to look at the network architecture. We discuss how training examples are passed through the network to produce an output and what the most efficient way to program this is.

Neural networks are organized into layers, and each layer is populated by nodes (figure 4). Each node has a weight, w, and bias, b, and these are the *parameters* of the network. These parameters control the output of the neural network, and so control how well it performs. They also each have an activation function, $f(z)$, associated with them to produce an output, a. Parameters are learnt by *training the network*. The goal of training the network is to find the parameters that best describe the underlying relationships between the data. Any choices we make about the network's architecture are called *hyperparameters*. The activation function of a node is an example of this. Other examples include the number of layers or nodes per layer, the training algorithm used and the learning rate.

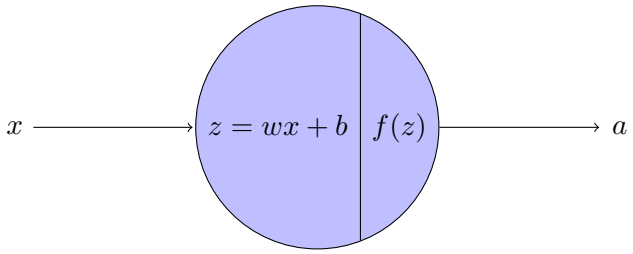


Figure 4: A node

A node takes an input x, multiplies it by a weight w and adds a bias b, before passing it through an activation function to give an activation value a. The image above represents this when the node has an input and output which are both only 1 dimension. Most neural networks have inputs and outputs with many dimensions. So if the input of a node is $x \in \mathbb{R}^n$ this process is done using matrix multiplication and addition. There is an individual weight for every input so the node instead has a weight vector: $w \in \mathbb{R}^n$. The equations that happen 'within' the node are then

$$z = w^T \cdot x + b$$
$$a = f(z)$$

2.1 Shallow and Deep Networks

The name *neural* network was chosen because they were inspired by the brain. In biological neural networks, information is passed between brain cells and sensory neurons via synapses. Artificial neural networks simulate this process. Just as a brain does not require being given instructions for how to learn, neural networks do not need to be given instructions, just training examples. To build a neural network, we construct layers populated with nodes. We then connect each node in the first layer to every node in the second layer, and repeat this for every layer until we reach the output layer, replicating the neuron structure in the brain. This output layer is only connected to nodes in the layer preceding it. It outputs some information that it has deduced based on the data that has been passed through the

network. For example, if a picture of a shoe was passed through the network, it could output the probability that a shoe is in that image, based on what it has learnt about pictures of shoes from the training examples.

When building the neural network we decide how many layers it should have, and how many nodes should be in each layer. This number can be different for each layer. It is convention that the number of layers of a network refers to all the layers that have nodes. That is, we do not include the input layer which just contains data, but we do include our output layer. Figure 5 is an example of a *shallow neural network* with two layers. It has one 'hidden' layer of nodes before its output layer. The term *shallow* refers to the fact that we only have one hidden layer. If we have many hidden layers before our output layer then we describe the network as *deep*.

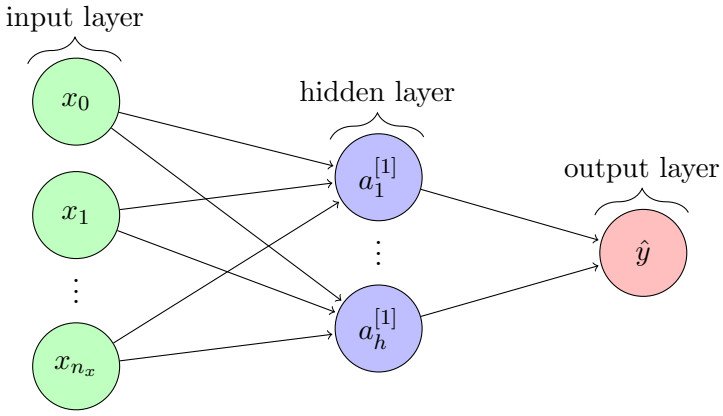


Figure 5: A two-layer neural network with n_x input units and 1 output unit. The hidden layer has h nodes.

Because it has a very simple structure, we only use shallow neural networks for simple tasks. For example, suppose we want to predict if an item of clothing such as a shoe is present in an image. We can measure the RGB properties (out of 255) of the pixels of 70000 images and label these images with whether or not they contain a shoe. We then feed the majority of the images as training examples into our network and it will start with a random guess of the nodes' parameters to predict whether or not the image is a shoe, $\hat{y}$. It will then improve on this guess as the training process goes on. $\hat{y}$ will be a number between 0 and 1. Typically we would say that if $\hat{y} < 0.5$ then the network has guessed that the image did not contain a shoe, otherwise it has predicted that the image does contain a shoe. We then use the rest of the images to test our network. This is an example of *binary classification*: we are classifying an item as one of two labels.

We can extend this further to classify the image as one of several different clothing types. This is an example of multinomial logistic regression, and for this task we can use the Fashion MNIST dataset. Since the dataset is now far more complicated to describe (we have 10 categories of clothing, whereas before we just had 'shoe' or 'not shoe') we would use a *deep neural network* with several hidden layers to account for the additional complexity of the problem. Figure 6 is an illustration of this. These networks are called *fully connected neural networks* (FCNNs) because every node in one layer is connected to every node in the next layer. Later in this thesis we will revisit this example in more detail and use Python to program the network.

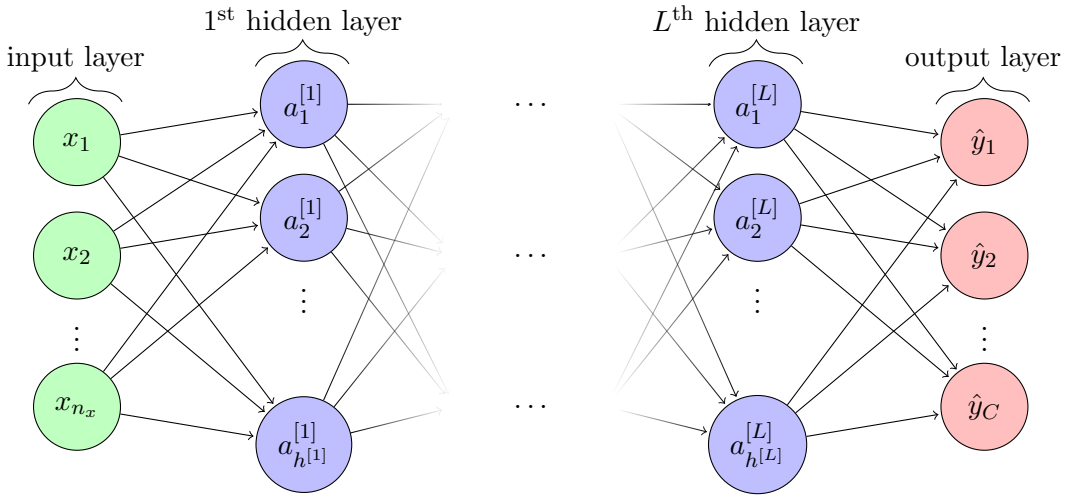


Figure 6: A $(L+1)$-layer neural network with n_x input units and C output units. The l^{th} hidden layer contains $h^{[l]}$ hidden units.

2.2 Forward Propagation

Forward propagation is the process by which data is passed through the network to produce an output. We will only discuss feed forward networks in this Chapter. Feed forward networks are when data is only passed in one direction through the network. The input data is fed through each node of the first layer layer to give activation values that can then be passed to each node in the second layer. The process is repeated for all layers through the network and then an output is produced for each example in the original dataset. The reviewing of these outputs to improve the network is explained in the backpropagation section of the next Chapter.

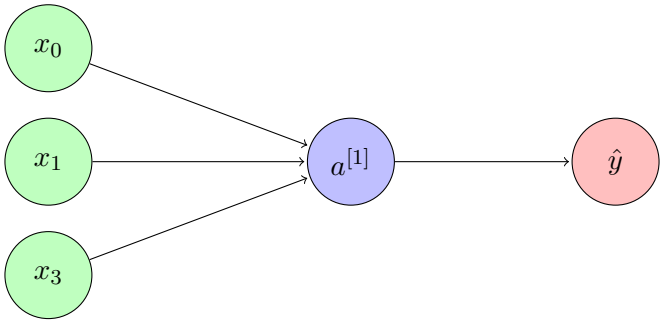


Figure 7: Forward propagation. The equations a training example go through in this network are given below.

We use an example of a 2 layer network with a single node in the hidden layer to illustrate forward propagation. For this example we can imagine a housing dataset which has categories of floorspace x_1, garden size x_2 and distance to centre of town x_3. The network is trained so that it can predict the price of the house. Figure 7 shows the network, while the equations below show the process for a single example $x \in \mathbb{R}^3$ of the dataset (e.g.

one house) passing through the network.

1. $z^{[1]} = w_1^{[1]}x_1 + w_2^{[1]}x_2 + w_3^{[1]}x_3 + b^{[1]}$
2. $a^{[1]} = f(z^{[1]})$
3. $z^{[2]} = w^{[2]T}a^{[1]} + b^{[2]}$
4. $\hat{y} = f(z^{[2]})$

2.3 Vectorisation

Up until now we have only considered the equations for a single example being passed through the network. However, during the training process, many thousands of training examples need to be passed through the network. Traditionally in computing, if we need to repeat an identical process multiple times we would create a *for* loop. This would result in one training example passing through the network, followed by the next and then the next until all training examples had gone through the network. This would take an unnecessarily long time. The essential mathematical operation of neural networks is matrix multiplication. With matrix multiplication we can take advantage of the parallel capabilities of the computer rather than iterate with for loops. Training a network requires m training examples of an n_x-dimensional dataset to go through each node of each layer of a network. Each training example is a vector, $x^{(i)} \in \mathbb{R}^{n_x}$. These are combined into a dataset matrix, X, which looks like the following $n_x \times m$ matrix:

$$X = \begin{bmatrix} \vdots & \vdots & & \vdots \\ x^{(1)} & x^{(2)} & \dots & x^{(m)} \\ \vdots & \vdots & & \vdots \end{bmatrix}$$

Each node i from a layer $[l]$ has a weight vector and bias, $w_i^{[l]} \in \mathbb{R}^{n_x}$ & $b_i^{[l]}$ which is 1-dimensional. We can combine the transpose of the weight vector to create a weight matrix for each layer. We turn the bias into a row vector of identical elements so that we can combine these into a bias matrix. These are both of size $h \times n_x$ where h is the number of nodes in that layer.

$$W^{[l]} = \begin{bmatrix} w_1^{[l]T} \\ \vdots \\ w_h^{[l]T} \end{bmatrix} \qquad B^{[l]} = \begin{bmatrix} b_1^{[l]} \\ \vdots \\ b_h^{[l]} \end{bmatrix}$$

For a network with 1 hidden layer the vectorised equations for passing m training examples through the network at once are shown below and can be trivially extended for many-layer neural networks. $f(x)$ represents the activation function chosen.

$$\begin{aligned} Z^{[1]} &= W^{[1]}X + B^{[1]} \\ A^{[1]} &= f(Z^{[1]}) \\ Z^{[2]} &= W^{[2]}A^{[1]} + B^{[2]} \\ A^{[2]} &= f(Z^{[2]}) \qquad\qquad (= \hat{Y}) \end{aligned}$$

We have now looked at the structure and components of neural networks in some detail and have covered enough ground to build a neural network. The next Chapter details the training of a network to find the parameters that best model a dataset. We also introduce the choices one has for the activation function, $f(x)$, and parameter initialisation of a node.

3 Training a network

In the previous Chapter we discussed the components and structure of neural networks. In this Chapter we look at the process of training neural networks. The purpose of a neural network is to model complex datasets. The network does this by learning a set of parameters (weights and biases) that best allow it to grasp the important features of the dataset. It does this by learning from training examples. The process of training the network is best summarised by figure 8.

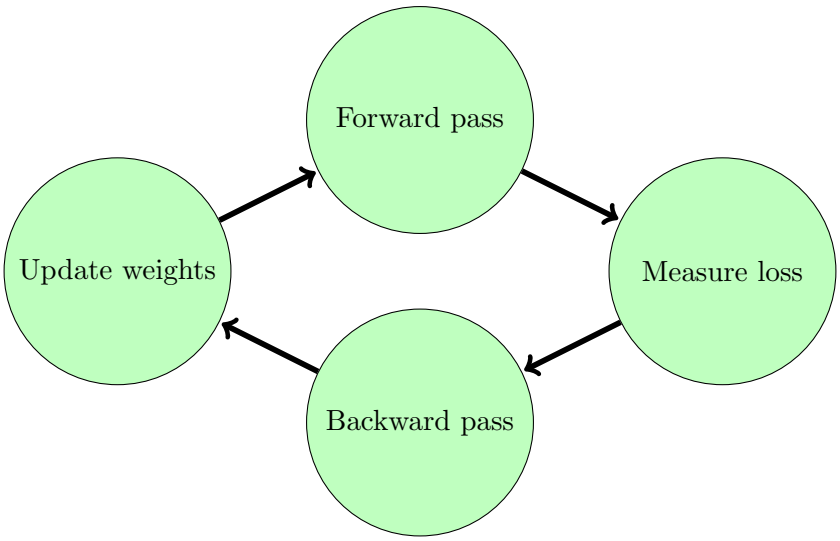


Figure 8: Training cycle for a neural network

This cycle is repeated until we reach a satisfactory loss. In more detail the process is as follows:

1. The number of layers and nodes per layer is chosen. We choose activation functions for each layer and the weights are randomly initialised. The bias of each layer is initialised to the 0 vector.

2. A training set of m examples is fed through the network. This is one forward pass or *forward propagation.*

3. The network gives an output $\hat{y}$. This could be a binary value referring to a yes/no answer or a continuous value such as house prices. Since this is training data, we know the correct value, y, for every training example. The difference between that and $\hat{y}$ is calculated as the *loss.* A *cost function* is calculated using the loss.

4. We move backward through the network to update the weights and bias. This is called *backpropagation.* The new parameters are chosen by minimising the cost function with *gradient descent.*

5. The first four steps are repeated for a predetermined number of iterations. One iteration is defined as one forward pass and one backward pass of the entire training dataset through the network.

6. The network is evaluated using a *validation dataset* that the network hasn't previously seen.

7. *Hyperparameters* are tweaked and the entire process can be repeated to improve the network. We will discuss this only in Chapter 4.

3.1 Splitting the data

Before training a network the dataset is (randomly) split into 3 categories:

- Training set: This subset of the data is used to train the network and optimise the weights. This is what is used in forward propagation and backpropagation.

- Development (dev) or validation set: This subset of the data is used to test out different hyperparameters (and hence different models).

- Test set: Once the best model has been chosen, this is then evaluated using the test set and this mimics the process of using the network in a 'real world' scenario.

The proportions of these sets depend on how much data we have. We want all three sets to show the same trend. If we have a small amount of data (less than a million training examples) we generally split the data 6:2:2 or 7:2:1 because we need the validation and test set to be big enough to show the trends in the data. If we have more than a million examples in our dataset we can split it with more data in the training set e.g. 98:1:1. This is an engineering choice and using more or less training data can alter the performance of a network.

It is important to ensure there are no data leaks between training and test data. This is because if a model has been exposed to any of the test data then it is inherently biased when being tested with it. This means that when tested on real data it could perform poorer than expected based on the test data evaluation. There a few ways to prevent this.

- **Clean the data**: Scan through the data to remove any duplicates that could end up being in more than one subset when the data is split.

- **Pre-processing data after splitting**: We often need to scale data in some way before using it. We want the training set to contain no information contained in the test set. If we transform the data before splitting it into training and dev sets, then when we come to use the training set it has already been influenced by data from the dev and test sets. This is because we have used information from the whole dataset, such as the mean or maximum and minimum values, to standardise or normalise the data. Even though we pre-process after splitting the data, we must remember to use the same transformation techniques on each of the training, dev and test sets.

3.2 Loss and Cost Functions

Neural networks optimise their parameters by minimising a *cost function*. Each time data is input into the model and a prediction is output we use a cost function to evaluate the performance of the model in that instance based on the current *parameters*. Since we pass several datapoints (e.g. several images) through our network, we have a *loss* for each datapoint and a cost for the whole dataset. The distinction between loss and cost is important: loss is a characteristic of a single prediction, $\hat{y}^{(i)}$, and a single true value, $y^{(i)}$,

whereas cost is an average of all the losses when an entire training set is passed through a network.

Going back to our binary classification shoe image problem, we would most likely use the *Binary Cross-Entropy Loss*.

$$L_{BCE}(y^{(i)}, \hat{y}^{(i)}) = -(y^{(i)} log(\hat{y}^{(i)}) + (1 - y^{(i)}) log(1 - \hat{y}^{(i)}))$$

This is useful for binary classification because it takes the value 1 when the prediction $\hat{y}^{(i)}$ is incorrect, and 0 when $\hat{y}^{(i)}$ is correct, whether $y^{(i)}$ is 0 or 1. We would then compute the cost as

$$\begin{aligned} J_{BCE}(y^{(i)}, \hat{y}^{(i)}) &= \frac{1}{m} \sum_{i=1}^{m} L_{BCE} \\ &= \frac{1}{m} \sum_{i=1}^{m} -(y^{(i)} log(\hat{y}^{(i)}) + (1 - y^{(i)}) log(1 - \hat{y}^{(i)}) \end{aligned}$$

For categorical classification (such as predicting the item of clothing in an image) we use a generalised version of the binary cross-entropy loss. Let C be the number of target labels and $y_j^{(i)}$ be the j-th entry in the label vector $y^{(i)}$ of the i-th training example. The label vector takes the value 1 for the category the training example is in and 0 for all other values of $j \in \{1, ..., C\}$ (this is called one-hot encoding and we discuss it in the next Chapter). $\hat{y}_j^{(i)}$ is the j-th entry in the softmax prediction vector $\hat{y}^{(i)}$ of the i-th training example. Then the cross-entropy loss and cost are

$$\begin{aligned} L_{CE}(y^{(i)}, \hat{y}^{(i)}) &= -\sum_{j=1}^{C} y_j^{(i)} log(\hat{y}_j^{(i)}) \\ J_{CE}(y^{(i)}, \hat{y}^{(i)}) &= \frac{1}{m} \sum_{i=1}^{m} L_{CE}(y^{(i)}, \hat{y}^{(i)}) \end{aligned}$$

For continuous label problems, such as if we are trying to predict the price of a house, we would use a different cost function. The two most common options are *mean square error* (MSE)

$$J_{MSE}(y^{(i)}, \hat{y}^{(i)}) = \frac{1}{m} \sum_{i=1}^{m} (y^{(i)} - \hat{y}^{(i)})^2$$

and *mean absolute error* (MAE)

$$J_{MAE}(y^{(i)}, \hat{y}^{(i)}) = \frac{1}{m} \sum_{i=1}^{m} |y^{(i)} - \hat{y}^{(i)}|$$

We use the square or absolute value because it ensure that all the errors are positive so that when they are summed up there is no possibility of negative and postive errors cancelling each other out to give an error of zero. In general we prefer MAE because it is more *robust to outliers*. This is because if we have an outlier and use MSE, then squaring the loss magnifies the impact it has on the cost function. This characteristic is important when dealing with data that has lots of noise (outliers).

3.2.1 Origins of the Cross-Entropy Loss

The cross-entropy loss function comes from the concepts of entropy and information. In information theory, a surprising event contains more *information* than an unsurprising event. This is because if we find out that an unlikely event has happened, this tells us more information than if a likely event occurred. It is more likely that we find out less information. The information of an event can be calculated by using its probability, $p(x)$.

$$h(x) = -\log_2 p(x)$$

Clearly the information $h(x)$ is monotonically decreasing in $p(x)$ between 0 and 1 (the values that $p(x)$ can take). This agrees with the theory: as the probability of an event increases, the amount of information it gives us decreases. When the probability of an event is 1, there is no surprise if it happens so it gives us no information and $h(x) = 0$. We can extend the concept of information to random variables.

The information of a random variable is called its *entropy*, $H(X)$. It is the expected amount of information of an event x from the discrete random variable X.

$$H(X) = -\sum_x p(x) \log_2 p(x)$$

The unit of information when using $\log_2$ is bits. If we use the natural logarithm then it is nats.

If we have a probability distribution, $P(x)$, and an approximation to this probability distribution, $Q(x)$, we may seek to calculate the amount of additional information needed to represent an event using Q instead of P. This is the *cross-entropy* between P and Q.

$$H(P, Q) = -\sum_x P(x) \log_2 Q(x)$$

Since we are dealing here with neural networks, let us translate this language into something that is useful for this context. The additional information required to represent an event can also be thought of as how much information is lacking when we try to represent x using Q instead of P, or the 'lost' information when going from P to Q. For neural networks, this translates to how far away Q was from correctly modelling the events x in the units of bits. This is an ideal candidate for the loss. Going back to the Fashion MNIST example, we know the distribution $P(x)$ for every image in the training data, where x is the event that the image is from a particular item of clothing. For example, figure 9 gives the known probability distribution for an image of a dress.

x	T-shirt	Trouser	Pullover	Dress	Coat	Sandal	Shirt	Sneaker	Bag	Ankle boot
P(x)	0	0	0	1	0	0	0	0	0	0

Figure 9: Probability distribution of the labels of one image

We store this information as a vector, $y^{(i)} \in \mathbb{R}^{10}$, and this is the known label of our image of the dress. This vector $y^{(i)}$ can be thought of as $P(x)$. Now, for this problem we would use the softmax activation function in our final output layer. This outputs a probability for each label x that the training example could take. In this context, our network outputs the vector $\hat{y}^{(i)} \in \mathbb{R}^{10}$ which has the probability that the image is each of t-shirt, trouser, etc.

We can think of this $\hat{y}^{(i)}$ as the approximation probability distribution, $Q(x)$, and it could look like figure 10.

x	T-shirt	Trouser	Pullover	Dress	Coat	Sandal	Shirt	Sneaker	Bag	Ankle boot
Q(x)	0.121	0.002	0.038	0.715	0.016	0.044	0.011	0.003	0.008	0.042

Figure 10: Predicted probability distribution of the dress image

$Q(x)$ is represented by the network as the vector $\hat{y}^{(i)}$. We simply plug the vectors $\hat{y}^{(i)}$ and $y^{(i)}$ into our formula for $H(P, Q)$ and sum over the 10 events X (the different clothing items) to get the cross-entropy loss.

$$\begin{aligned} H(y^{(i)}, \hat{y}^{(i)}) &= -\sum_{j=1}^{10} y_j^{(i)} \log \hat{y}_j^{(i)} \\ &= L_{CE}(y^{(i)}, \hat{y}^{(i)}) \end{aligned}$$

3.3 Backpropagation

Backpropagation is the process of moving backwards through the network to update the parameters (weights and bias). We perform this after every forward pass of the training data through the network. The neural network calculates the gradient of the cost function with respect to the parameters of each node. This is then used in *gradient descent*. The training data is then fed through the network again, and the process is repeated for a chosen number of iterations or epochs. The number of iterations is a *hyperparameter* and we will discuss the tuning of these in Chapter 4. Gradient descent is just one optimisation algorithm, of which there are others that are effective for different contexts.

3.3.1 Gradient Descent

The most basic algorithm to optimise the parameters of a neural network is gradient descent. The idea is to find the parameters that minimise the cost function, which we can think of as a surface. We set a *learning rate*, α, multiply it by the gradient of our cost 'surface' with respect to the parameter, and take this away from the parameter to edge closer to the minimum of the function. We then forward propagate and backpropagate again.

$$w_{new} = w_{old} - \alpha \frac{\partial J(w, b)}{\partial w}$$

$$b_{new} = b_{old} - \alpha \frac{\partial J(w, b)}{\partial b}$$

The parameters we start with are initialised randomly and then updated with each iteration. Gradient descent received its name due to the idea that the gradient of a function points in the direction it is increasing in most. Therefore, if we move in the opposite direction to this we can move to the local minimum of our function. Ideally, we want to move to a global minimum, and we talk about increasing the chance of this in the next Chapter. Neural networks can have millions of parameters so the cost function can be million-dimensional. Therefore, minimizing it completely is very difficult. Figure 11 shows a visual representation of this.

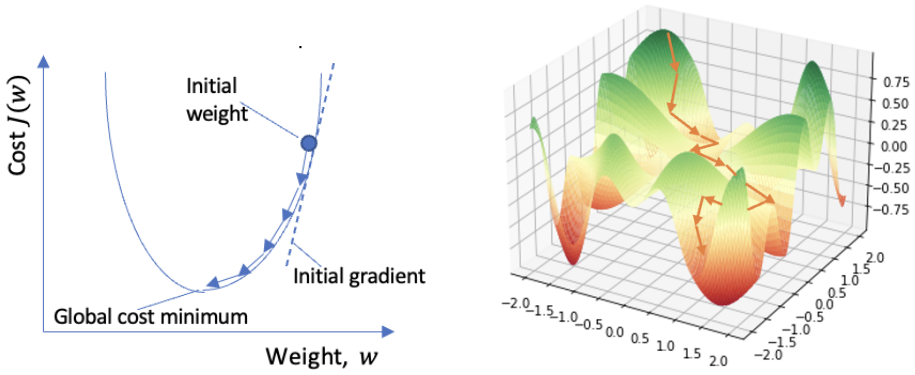


Figure 11: Gradient descent in two and three dimensions

The only hyperparameter of gradient descent is the *learning rate*, α. This controls the size of each 'step' towards the minimum cost in backpropagation. There is an optimum value for this step size, and it can be different for different datasets and network architectures.

- High learning rates allow us to get close to the minimum quickly but risk overshooting the minimum. If this happens we never actually reach the minimum cost because the steps keep oscillating around it.
- Low learning rates ensure we are always moving in the right direction towards the minimum, however take a long time to reach it. If the number of iterations is high enough then we can get very close to the minimum, but this requires a lot of time.

Often the best way to find the optimum learning rate for a problem is trial and error. This is a common theme when building neural networks because there are so many variables that the engineer can choose to change. In simple networks, the learning rate is fixed. However, it is clearly helpful to make large steps when we are far away from the minimum of the cost, and then smaller steps as we get closer. This can be implemented as gradient descent with variable learning rates and we will discuss this in the next Chapter.

3.4 Activation Functions

The nodes of each layer of a neural network have a pre-chosen activation function, $f(z)$. Activation functions are fed with the weighted (and biased) sum from the previous layer's output and output a value. Activation functions are never linear functions because their purpose is to prevent linearity. Without them the data would pass through the nodes and layers of the network only going through linear functions ($f(x) = w \cdot x + b$). The composition of these linear functions, $f(f(...f(x)))$, is again a linear function so we may as well have just used one layer. Complex data cannot be modelled well by linear functions (figure 12) so we fix this by using nonlinear activation functions in each node.

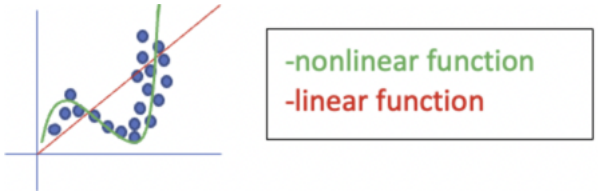


Figure 12: A simple representation of the benefits of nonlinear activation functions

Here we discuss some of the common activation functions and the reasoning behind choosing them in different scenarios. There are a few characteristics that we would like our activation functions to have.

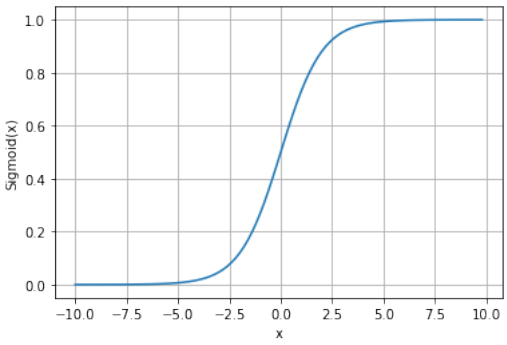
- Activation functions must be differentiable. We use their derivatives in gradient descent when we update the parameters of the network.
- Computationally cheap: many networks require activation values to be calculated millions of times. For this reason we want the functions to be as simple and computationally inexpensive as possible.
- Zero-centered: if outputs of a node are roughly symmetrical about $(0, 0)$ then parameter optimisation is faster. We discuss this further in the normalisation section of Chapter 4.
- Vanishing/exploding gradient problem: Consider a small three layer network with just one node per layer. The output $\hat{y}$ is a function of the parameters from every previous node, and therefore the cost, $J(\hat{y}, y)$, is too. When we calculate the gradients of the cost function with respect to the parameters in nodes from the first layers of the network we have to perform chain rule several times. This is how we get the step size to update parameters with in gradient descent. Because the step size of a parameter is a result of many multiplications of the gradients from layers after that parameter's layer, if these gradients are very small or large, then repeated multiplications result in extremely small or large step size. This results in slow convergence to optimum parameter values, which we want to avoid.

Output nodes need different activation functions to nodes in the hidden layers of a network. This is because we usually want the output of a neural network to be useful for the context. For example, if we are building the neural network to classify whether or not an image is of a shoe, we want the output to be a probability value between 0 and 1. For this we would use the sigmoid function as the activation function.

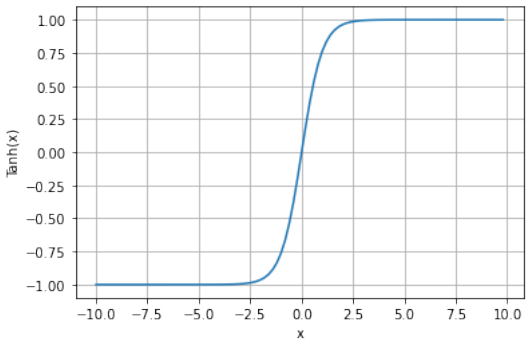
The range of the sigmoid function is $(0, 1)$. One can set a threshold probability value, such as 0.5, above which the output is classed as a "yes, there is a shoe in the image" (1) and below which the output is classed as "no shoe" (0).

There are, however, several disadvantages to using the sigmoid function.

- The function is centred around $\frac{1}{2}$, not 0. This makes parameter optimisation slower for layers after a node with sigmoid activation function.
- Large parts of the function are almost flat and so have very small gradients. This leads to vanishing gradient problem (also called saturation) and makes gradient descent extremely slow.
- Calculating exponents require lots of processing power which can slow down larger networks.



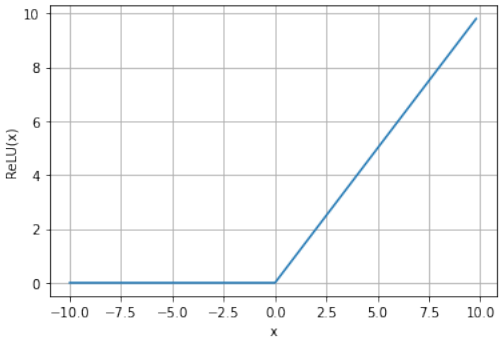
(a) Sigmoid: $S(x) = \frac{1}{1+e^{-x}}$



(b) Hyperbolic tangent: $\tanh(x) = \frac{e^{x}-e^{-x}}{e^{x}+e^{-x}}$

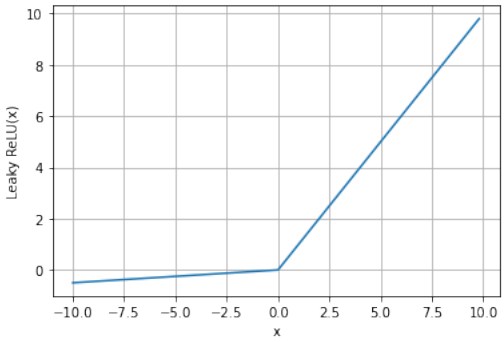
Figure 13: Two traditional activation functions

Hyperbolic tangent (tanh) is almost always preferable to the sigmoid function, which in reality is only ever used as the output activation function in binary classification. Both functions are shown in figure 13. It is zero-centred (we say it is normalized) so it produces zero-centred outputs, making parameter optimisation faster. The range of *tanh* is (-1,1). This means for binary classification we wouldn't use it for the output activation function instead of sigmoid, but we can use it in the hidden layers. However, like sigmoid, it suffers from saturation and is computationally expensive due to the exponents.



(a) ReLU:

$$f(x) = \begin{cases} 0 & x < 0 \\ x & x \geq 0 \end{cases}$$



(b) Leaky ReLU:

$$f(x) = \begin{cases} \alpha x & x < 0 \\ x & x \geq 0 \end{cases}$$

Figure 14: The ReLU functions

Rectified Linear Unit (ReLU) and the Leaky ReLU solve the issues faced by tanh and are shown in figure 14. ReLU has a range of $[0, \infty)$. As long as x is positive, there are no areas of saturation so the gradient descent process is very fast. The function is easy to compute and so requires little processing power.

The only issues with ReLU are that it is not zero-centred and if the input is negative, the node 'dies' (becomes inactive due to outputting the same value every time, 0 in this case). The input could be negative due to the network having a large negative bias term in that layer causing $w^T \cdot x + b < 0$. ReLU has a gradient of 0 in the negative part of the function so once a ReLU node has died, it cannot be recovered since it will not be able to

adjust the parameters in gradient descent (the steps would be of size 0 every time).

Leaky ReLU solves that issue. It has range of $(-\infty, \infty)$. The shallow slope for negative x is created by using a small value α (usually 0.01). This keeps nodes with negative inputs active and ensures the function is zero-centred (normalized). For these reasons ReLU and leaky ReLU are the most popular activation functions for hidden layers.

3.4.1 Softmax

We discussed using the sigmoid function as the output activation function when performing binary classification ("shoe or no shoe"), but what about when we are classifying many different categories such as the items of clothing in the Fashion MNIST dataset? For this we need to use a function that can assign a probability of the image being each category of clothing.

The *softmax* function is a generalised version of the sigmoid function which can be used for classification problems when we have more than 2 target labels. For the clothing example, we would need the output layer to have 10 nodes, each giving the probability that the image is of a different type of clothing (so node 1 would give the probability of a shoe, node 2 the probability of a hat, etc.). We could use sigmoid as the activation function for this layer, although then we run into the issue of classifying an image as multiple types of clothing if there are multiple nodes with an output above our chosen threshhold (usually 0.5). To solve this, the softmax function outputs a vector of probabilities which sum to 1. This vector assigns a probability for each target label which takes into account the fact that the total probability of all the different clothing types occurring has to sum to 1. This is important because the sigmoid function does not do this by itself. The softmax function uses the sigmoid values z_i and then generates *relative probabilities* using the function below for a network with C classes. This ensures that the final output of the network is not that the image is likely to be of a shoe, a hat and trousers but instead that the image is MOST likely to be a shoe (for example).

$$\sigma(z)_i = \frac{\exp(z_i)}{\sum_{c=1}^{C} \exp(z_c)}$$

This is the equation that takes place in each of the output nodes, so that the network now outputs a vector of probabilities $\hat{y}^{(i)} \in \mathbb{R}^C$ rather than a single value $\hat{y}^{(i)} \in \mathbb{R}$.

3.5 Initialisation

Neural networks predict an output by learning the parameters that best fit a dataset. In order to learn the best parameters, the network makes a random initial guess for the weights and bias of each node and then improves upon this using gradient descent to find the best parameters to model the data. The wrong kind of initial guess can lead to a slow or useless network and so there is a framework that ML engineers must work within.

The first suggestion might be to initialise all the weights and biases to 0. This is a tempting option because it is easy to implement. However if we begin with the weights all $= 0$ (or any identical number) then every node will behave exactly the same and give equal outputs. So no matter the path that the data takes through the network, the output (and hence the loss) is the same. Therefore, during backpropagation the weights and biases from the same layer will undergo identical gradient descent and reach identical optimum values after training. All nodes in a layer will then look identical, meaning all the nodes of that

layer do the same thing. We may as well have just used one node per layer. We need to "break the symmetry" of the nodes. To do this we have to assign random initial weights and biases. However, we still have to be careful with how we do this:

- If the initial weights are very large then the variance of outputs gets larger and larger as we get to the final layers of the network, because the outputs of the final layers are compositions of the outputs from previous layers. If using a sigmoid or tanh activation function this means we can get stuck in the flat parts of the line, where the gradient is very small. This makes learning very slow because the steps in the gradient descent are too small. This is an occurrence of vanishing gradient.
- If the weights are too small then we may get stuck in the approximately linear parts of our activation functions when using sigmoid or tanh. This means we lose the benefit of activation functions, that is they introduce nonlinearity.
- Since optimisation uses the chain rule to work out gradients for the weights of each layer, we also see vanishing (or exploding) gradients if our weights and hence gradients are too small (or large). This is because when working out the gradients of the early layers closer to the input the calculation involves the multiplication of several layers of weights so the result becomes very small (or large). This leads to slow or even impossible convergence as the steps in gradient descent are too small (or large).
- We want the average output of any layer to be 0, and the variance of the outputs to stay constant from layer to layer. This is also achieved with normalisation, which we discuss along with its motivations in the next Chapter.

In general, an engineer chooses between random sampling from a normal or uniform distribution for the weights to break the symmetry and initialises all biases to 0. This is a hyperparameter that the engineer can tweak to find the best choice for the dataset.

3.5.1 Xavier and Kaiming Initialisation

We want to keep the variance of outputs constant between layers. Here we derive what the variance of our sampling distribution for the initial weights should be to achieve this. Consider two layers of a neural network where we just look at one node from the layer $[l+1]$ and all the nodes in layer l that lead into it (figure 15). For simplicity we will perform this derivation for the scenario where the nodes do not have an activation function.

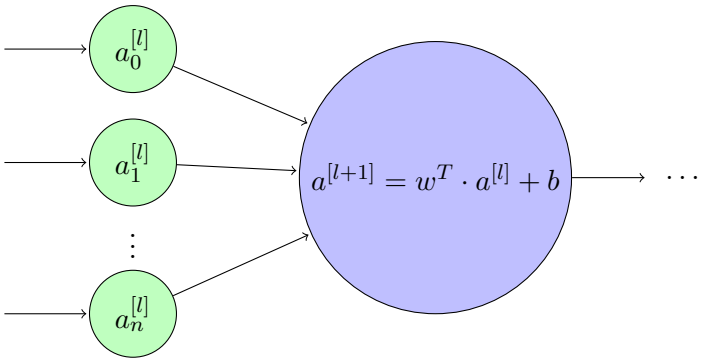


Figure 15

Let $a^{[l]} = (a_0^{[l]}, a_1^{[l]}, \dots a_n^{[l]})^T$ and $a^{[l+1]} \in \mathbb{R}$ be the input and output of the second layer's node respectively. It is the case in this linear scenario that the output of the l-th layer is equal to the input of the $(l+1)$-th layer's node. We want to ensure the variance of each $a_i^{[l]}$ is equal to the variance of $a^{[l+1]}$. $a^{[l]}$ is n-dimensional (i.e. the first layer has n nodes it outputs into the second layer's node) and we randomly sample our weights, w, from a normal distribution with mean 0.

$$\begin{aligned} a^{[l+1]} &= w^T \cdot a^{[l]} + b \\ &= w_0 a_0^{[l]} + w_1 a_1^{[l]} + ... + w_n a_n^{[l]} + b \end{aligned}$$

So let us look at the variance of $a^{[l+1]}$ and the variance of $a_i^{[l]}$

$$var(a^{[l+1]}) = var(w_0 a_0^{[l]} + w_1 a_1^{[l]} + ... + w_n a_n^{[l]} + b) \tag{1}$$

$$= var(w_0 a_0^{[l]}) + var(w_1 a_1^{[l]}) + ... + var(w_n a_n^{[l]}) + 0 \tag{2}$$

because we know that $w_i a_i^{[l]}$ are independent so any covariance terms vanish and $var(b) = 0$ as b is a constant. Let us look at the term $var(w_i a_i^{[l]})$

$$var(w_i a_i^{[l]}) = var(w_i) \cdot var(a_i^{[l]})$$

where we have used that w_i and $a_i^{[l]}$ are independent so have covariance$(w_i, a_i^{[l]}) = 0$. Subbing this back into (2) we see

$$var(a^{[l+1]}) = n \cdot var(w_i) \cdot var(a_i^{[l]})$$

So clearly if we want $var(a^{[l+1]}) = var(a_i^{[l]})$ we need

$$n \cdot var(w_i) = 1 \Leftrightarrow var(w_i) = 1/n \tag{3}$$

So we need to choose random numbers for our weights from a normal distribution with mean 0 and variance $\frac{1}{n}$ where n is the number of nodes in the preceding layer.

$$W \sim \mathcal{N}(0, \frac{1}{n^{[l-1]}})$$

This method of initialisation works for sigmoid and tanh activation functions because for small values of x, $tanh(x) = sigmoid(x) = x$ so our work resulting in equation (3) is valid even though we did it assuming there was no activation function in the nodes.

However, for ReLU and Leaky ReLU we use Kaiming initialisation [13]. We sample from a normal distribution with variance being a function of the number of nodes in the preceding layer again, but with a different numerator.

$$W \sim \mathcal{N}\left(0, \frac{2}{n^{[l-1]}}\right)$$

Weights can also be sampled from other distributions such as the uniform distribution, where Xavier initialisation [14] suggests we use the parameters

$$W \sim \mathcal{U}\left(-\frac{\sqrt{6}}{\sqrt{n^{[l-1]} + n^{[l]}}}, \frac{\sqrt{6}}{\sqrt{n^{[l-1]} + n^{[l]}}}\right)$$

as an alternative option. The aim of these initialisation techniques is to have output variances of 1. In all these methods the biases are initialised to zero. Initialisation is another hyperparameter to tweak when trying to improve the performance of a network and these are only the most common methods used.

3.6 Explicit Gradient Descent Example with Fashion MNIST

The Fashion MNIST dataset is a dataset of 70000 28×28 images of clothing items. Here, we create a basic neural network to classify the 10 clothing categories and describe the mathematics of each step. We will build a simple network with 1 hidden layer of 16 nodes with tanh activation function, and an output layer of 10 nodes with the softmax activation function. The architecture of this network is shown in figure 16. Note that there are 784 input nodes. This is a result of the fact that each image has 28×28 pixels.

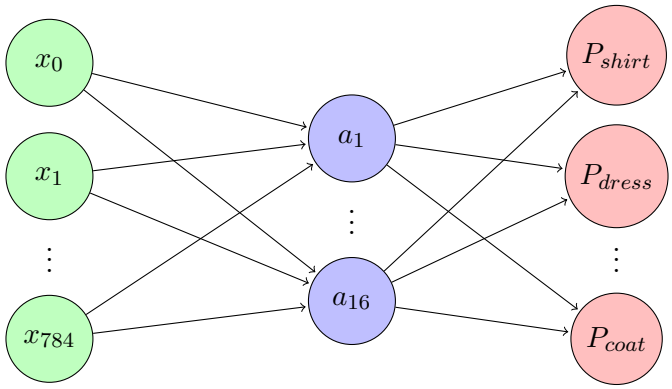


Figure 16: Architecture for a network to classify the Fashion MNIST dataset

1. First we split the data into a training, validation and test set. We use a test set of 10000 images, and a training set of 60000 images. 20% of our test set is then set aside as validation data. We then initialise our weight matrices for each layer, $W^{[l]}$, and our bias vector for each layer, $b^{[l]}$, with $l \in \{1, 2\}$, using Xavier initialisation.

2. We then forward propagate all 60000 training examples. Note $W^{[l]}$ is made up of row vectors $w_k^{[l]T}$ where each $w_k^{[l]}$ is the weight vector of the k-th node in layer $[l]$. Each training example, $x^{(i)}$, will go through the following functions

$$\begin{aligned} z^{[1](i)} &= w^{[1]T} x^{(i)} + b^{[1]} \\ a^{[1](i)} &= \tanh(z^{[1](i)}) \\ z^{[2](i)} &= w^{[2]T} a^{[1](i)} + b^{[2]} \\ a^{[2](i)} &= \sigma(z^{[2](i)}) \\ &= \hat{y}^{(i)} \end{aligned}$$

3. We then calculate the loss and cost. For logistic regression we use the categorical cross-entropy loss L_{CE} with $C = 10$.

$$L_{CE}(y^{(i)}, \hat{y}^{(i)}) = -\sum_{j=1}^{C} y_j^{(i)} log(\hat{y}_j^{(i)})$$

The cost function is

$$J(y^{(i)}, \hat{y}^{(i)}) = \frac{1}{m} \sum_{i=1}^{m} L_{CE}(y^{(i)}, \hat{y}^{(i)})$$

Note also that

$$\frac{\partial J}{\partial L} = \frac{1}{m}$$

4. We then need to differentiate the cost J with respect to the weight and bias parameters, θ. Note the notation

$$d\theta := \frac{\partial J}{\partial \theta}$$

The derivation of $d\theta$ requires chain rule

$$d\theta = \frac{\partial J}{\partial L}\frac{\partial L}{\partial z^{[2](i)}}\frac{\partial z^{[2](i)}}{\partial \theta} \tag{4}$$

We now work out each of the three components that make up the product in equation (4). Note that $y^{(i)}$, $\hat{y}^{(i)} \in \mathbb{R}^{10}$ because we have 10 label categories.

We know the first factor $\frac{\partial J}{\partial L} = \frac{1}{m}$. We will use the following fact about the softmax function σ which is proven in Appendix B. For a vector $\hat{y}$ with $\hat{y}_i = \sigma(z)_i$,

$$\begin{aligned}\frac{\partial \hat{y}_i}{\partial z_j} &= \sigma(z)_i\left(\delta_{ij} - \sigma(z)_j\right) \\ &= \hat{y}_i^{(i)}(\delta_{ij} - \hat{y}_j^{(i)})\end{aligned}$$

We now work out the second factor in equation (4). For $L = -\sum_j y_j^{(i)} log(\hat{y}_j^{(i)})$,

$$\begin{aligned}\frac{\partial L}{\partial z_i^{[2](i)}} &= -\sum_{k=1}^{10} y_k^{(i)}\frac{\partial log(\hat{y}_k^{(i)})}{\partial z_i^{[2](i)}} \\ &= -\sum_{k=1}^{10} y_k^{(i)}\frac{\partial log(\hat{y}_k^{(i)})}{\partial \hat{y}_k^{(i)}} \times \frac{\partial \hat{y}_k^{(i)}}{\partial z_i^{[2](i)}} \\ &= -\sum_{k=1}^{10} y_k^{(i)}\frac{1}{\hat{y}_k^{(i)}} \times \hat{y}_i^{(i)}(\delta_{ik} - \hat{y}_k^{(i)}) \\ &= -y_i^{(i)}(1 - \hat{y}_i^{(i)}) + \sum_{k\neq i} y_k^{(i)} \cdot \hat{y}_i^{(i)} \\ &= \hat{y}_i^{(i)} - y_i^{(i)}\end{aligned}$$

since $\sum_k y_k^{(i)} = 1$ as $y^{(i)}$ is a one-hot encoded vector (so every entry is 0 except for one which is 1 (see figure 9 for an example)). So

$$\frac{\partial L}{\partial z^{[2](i)}} = \hat{y}^{(i)} - y^{(i)}$$

We now derive the final component of equation (4). For $z^{[2](i)} = w^{[2]T} \cdot a^{[1](i)} + b^{[2]}$,

$$\begin{aligned}\frac{\partial z^{[2](i)}}{\partial w^{[2]}} &= a^{[1]} \\ \frac{\partial z^{[2](i)}}{\partial b^{[2]}} &= 1\end{aligned}$$

Now we can finally fill in the three partial derivatives from equation (4).

$$\begin{aligned}dw^{[2]} &= \frac{1}{m} \cdot (\hat{y}^{(i)} - y^{(i)}) \cdot a^{[1]} \\ db^{[2]} &= \frac{1}{m} \cdot (\hat{y}^{(i)} - y^{(i)}) \cdot 1\end{aligned}$$

It can be shown similarly using further chain rule that for the first layer we have

$$dw^{[1]} = \frac{1}{m} \cdot (\hat{y}^{(i)} - y^{(i)}) \cdot x$$
$$db^{[1]} = \frac{1}{m} \cdot (\hat{y}^{(i)} - y^{(i)})$$

5. We then update the parameters. This means updating the weights of every node in each layer and the bias of each layer.

$$w_{new} = w_{old} - \alpha dw$$
$$b_{new} = b_{old} - \alpha db$$

6. Finally, we repeat the process for our chosen number of epochs. We now discuss the programming of this network using Python.

3.7 Implementing Neural Networks with Python

The mathematics we have seen explain how neural networks work. Understanding this is a huge advantage when building machine learning models. It allows engineers to evaluate different methods and choose the best kind of model for a problem. However, when implementing these models with Python there are helpful public libraries that can be taken advantage of. Tensorflow's Keras is an application programming interface (API) which allows an engineer to quickly build neural networks without having to define functions to perform all the mathematical tasks described above. This makes it easy to run experiments quickly and try out new approaches to a project. To build a neural network with Keras we define a model as a 'sequential' object and then add layers to this. In each layer added we specify the hyperparameters, such as activation function or initialisation method, before we finally compile and fit the model to a set of training data. For example, to build and train a 2-layer network with 16 nodes in the hidden layer to classify the Fashion MNIST set we would do the code in listing 1. By the 16th epoch the network reached a test set accuracy of 80.71% and took just 89.80 seconds to train.

```
# define the model and add layers
model = tf.keras.Sequential()
model.add(tf.keras.layers.Flatten(input_shape=(28, 28)))
model.add(tf.keras.layers.Dense(units=16, activation='tanh',
    kernel_initializer = keras.initializers.GlorotNormal))
model.add(tf.keras.layers.Dense(units=10, activation='softmax',
    kernel_initializer = keras.initializers.GlorotNormal))

# compile and fit the model to the training data
model.compile(optimizer=keras.optimizers.Adam(), loss=tf.keras.losses.
    SparseCategoricalCrossentropy(), metrics = ['accuracy'])
model.fit(x_train,y_train, epochs=16, validation_split=0.2)
```

Listing 1: Python code that defines, builds and fits our neural network to the training data

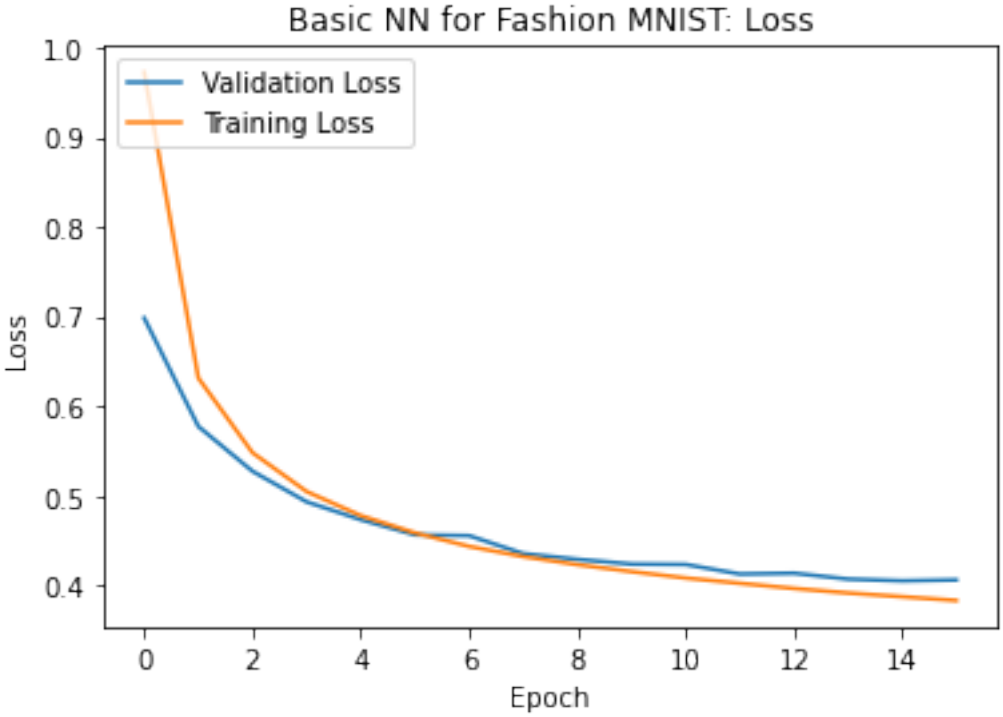


Figure 17: The error of our model

In this Chapter we discussed how neural networks are trained. After each epoch of training, the model is tested on a small set of validation data. If we look at the loss graph (figure 17) we notice that after the 9th epoch the validation loss stops decreasing but the training loss continues to decrease. This shows that we may have overfit the network to the training data so that it starts to perform less well on the unseen validation data. We discuss this and other problems that may come up when building a neural network in the next Chapter. We then offer some solutions and steps we can take to improve the efficiency of our networks, before improving the network we have just built.

4 Evaluating and Improving the Performance of Networks

The aim of a network is to classify real-world data as accurately as possible. In this Chapter we introduce the metrics used to evaluate neural networks and discuss how to get the best performance from them. We will see common issues we encounter when training a network and describe their solutions. We then finally apply these optimisation techniques to the network we built to classify the Fashion MNIST dataset.

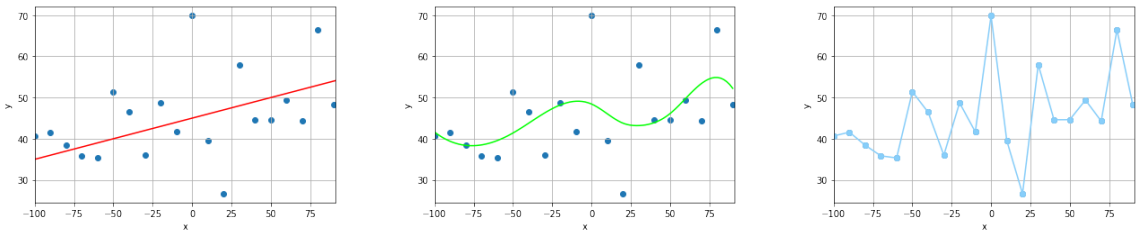
When we evaluate networks using validation data, there are several model characteristics we look at. To understand these, we need to define *bias* and *variance*.

4.1 The Bias-Variance Trade Off

Bias: High bias or *underfitting* is when the network is too simple to grasp the distribution of the data (shown by figure 18a). It has a low accuracy when it comes to using it on real-world examples. To diagnose high bias we look at a validation loss-epoch graph such as in figure 17. If validation loss is still decreasing as we train the network then we know we have underfitting because we have not yet found the best model for the data. We need to make the model more complex by either iterating the training for longer or increasing the number of layers and nodes of the network.

Variance: High variance or *overfitting* is when the network is too sensitive to the training data, and cannot generalise to real-world data well (shown by figure 18c). Again, we use a loss-epoch graph. If validation loss is increasing as we further train the network, while training loss is decreasing, then we know we have overfitting because the model is becoming too specific to the training data. Models with high variance are too complex and too sensitive to the training data, and cannot generalise to real-world data well.

We need to simplify our model and reduce the influence of training data outliers (noise) on our network by using more training data to reduce the amount of noise relative to the total amount of data, or *regularisation*. This can involve adding a penalty term, $\lambda f(w)$, to the cost function or using *dropout* to randomly turn off some nodes during each training iteration. These techniques reduce the likelihood of high weights which are a sign that the network has become too complex.



(a) High bias & low variance: Underfitting

(b) Medium variance & medium bias

(c) High variance & low bias: Overfitting

Figure 18: Characteristics of different models

An ideal model has both low bias and low variance. However, the two are intrinsically inversely linked. If we modify the network so that it fits the training data better we will reduce the bias but increase the variance, and vice versa. Figure 18b shows a good balance between the two. When choosing our hyperparameters to adjust bias and variance we need

to carefully try to balance the impact on the two factors. The effects of altering model complexity are shown explicitly in figure 19.

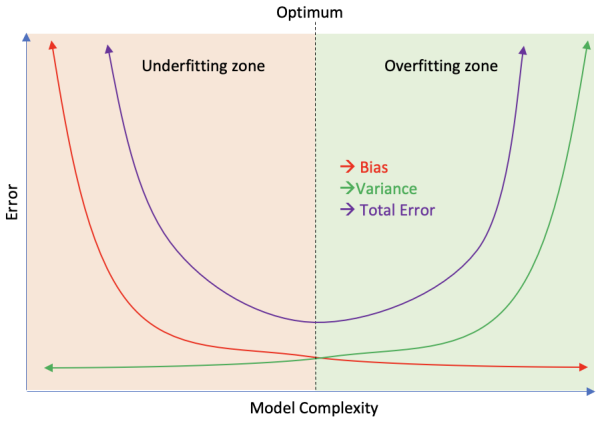


Figure 19: The Bias-Variance Trade Off

4.2 Regularisation

We have mentioned before that if we have a high variance, one method to reduce this is to use more training data. We will discuss data augmentation in the next section, and this is the first step one should take to reduce variance. However, this can become expensive or even impossible depending on the context. An easier method involves altering the network to simplify the model and reduce the impact noise has on the network. There are various ways to do this, though the common thread is that high weights are penalised. This is because high weights signify a more complex model that has overfit to the specific features or patterns of the training data. These methods help reduce weight magnitudes, effectively reducing the influence individual nodes have on a network and in some cases reducing the number of active nodes, if the weights are pushed to 0. We discuss these methods, and then consider data augmentation. First, however, we introduce the most low-tech method to reduce overfitting.

4.2.1 Early Stopping

As we iterate the optimisation process, the network becomes more and more specific to the training data. At first, this results in an improvement on performance of test data classification. However, as training goes on, the improvement slows and even stops. This is because the model is now overfitting to the specific features of the training set and so will perform poorly on test/new data. If we stop training before this, we can prevent the model from overfitting. This gives us a simple way of preventing the network becoming too specific to the training data: *early stopping*. We can visualise this with an error-epoch graph (figure 20).

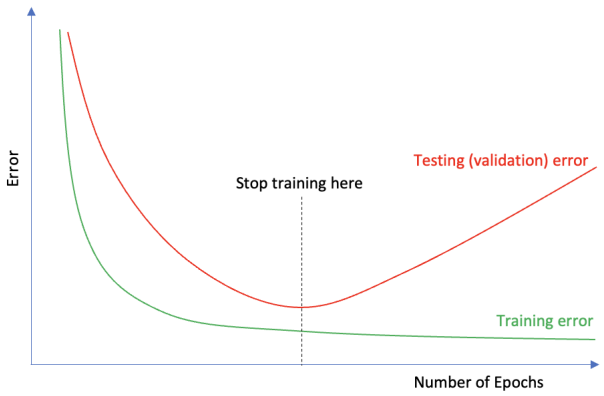


Figure 20: Early Stopping

4.2.2 L1 and L2 Regularisation

When training a network to learn the best weights we try to minimise a cost function

$$J(w^{[1]}, b^{[1]}, ..., w^{[l]}, b^{[l]}) = \frac{1}{m}\sum_{i=1}^{m} L(\hat{y}^{(i)}, y^{(i)})$$

We can add a term to this equation to penalise large weights. This comes in the form of a norm function such as

$$L_1 : \|w^{[l]}\|_1 = \sum_{i=1}^{n^{[l]}} |w_i^{[l]}|$$

$$L_2 : \|w^{[l]}\|_2^2 = \sum_{i=1}^{n^{[l]}} (w_i^{[l]})^2$$

$$Frobenius : \|w^{[l]}\|_F^2 = \sum_{i=1}^{n^{[l]}} \sum_{j=1}^{n^{[l-1]}} (w_{i,j}^{[l]})^2$$

We then use a regularisation hyperparameter, λ, and this controls how much we want to penalise large weights. λ is generally small (≈ 0.01). The cost function becomes

$$J(w^{[1]}, b^{[1]}, ..., w^{[l]}, b^{[l]}) = \frac{1}{m}\sum_{i=1}^{m} L(\hat{y}^{(i)}, y^{(i)}) + \lambda \sum_{l=1}^{L} \|w^{[l]}\|_j$$

where j denotes the norm function used. We then perform gradient descent as usual, but with a new term to account for in our derivative of the cost function. The derivative of the new term for L1 regularisation is

$$\frac{d}{dw}\left(\lambda \sum_{l=1}^{L} \|w^{[l]}\|_1\right) = \frac{d}{dw}\left(\lambda \sum_{l=1}^{L}\sum_{i=1}^{n^{[l]}} |w_i^{[l]}|\right) = \begin{cases} -\lambda & w^{[l]} < 0 \\ \lambda & w^{[l]} > 0 \end{cases}$$

(where $|w|$ is only differentiable when $w \neq 0$). So the gradient descent equation becomes

$$\begin{aligned} w_{new} &= w_{old} - \alpha \frac{\partial J(w,b)}{\partial w} \\ &= \begin{cases} w_{old} - \alpha(dw - \lambda) & w < 0 \\ w_{old} - \alpha(dw + \lambda) & w > 0 \end{cases} \end{aligned}$$

where dw is the derivative of the cost function with respect to w before we implemented regularisation. We see that if $w < 0$ then our new hyperparameter λ makes w less negative by being added to w_{old}, while if $w > 0$ then λ makes w less positive by being taken away from w_{old}. This makes weights with high magnitude less likely to appear.

L2 regularisation works in a similar fashion. The derivative of our new term is

$$\frac{d}{dw}\left(\lambda \sum_{l=1}^{L} \|w^{[l]}\|_2^2\right) = \frac{d}{dw}\left(\lambda \sum_{l=1}^{L}\sum_{i=1}^{n^{[l]}} (w_i^{[l]})^2\right) = 2\lambda w$$

so the gradient descent equations become

$$\begin{aligned} w_{new} &= w_{old} - \alpha \frac{\partial J(w,b)}{\partial w} \\ &= w_{old} - \alpha(dw + 2\lambda w_{old}) \\ &= (1 - 2\alpha\lambda)w_{old} - \alpha \cdot dw \end{aligned}$$

where again dw is the derivative of the cost function with respect to w before we implemented regularisation. This penalises weights of high magnitude because it makes the number that $\alpha \cdot dw$ is taken away from smaller in magnitude than it would be without regularisation (because $1 - 2\alpha\lambda < 1$).

4.2.3 Dropout Regularisation

Another method of reducing overfitting and hence variance of a network is by randomly 'killing' some nodes, an idea first proposed by Hinton et al [15]. We decide the threshold probability of keeping each node live. For example, if we set the threshold at 0.5, approximately 50% of the nodes will be 'killed' and the rest will stay live. This makes the network much smaller (simpler). We do this repeatedly for each iteration of forward and backpropagation. During each iteration a different group of nodes is kept live because the dropout process is random (figure 21 shows an instance of this). We allow the network to perform gradient descent as usual to optimise the weights and bias.

This simplifies the model because the network is less likely to give high weights to nodes if there is a chance they will not be active. Therefore the influence of noise is minimised. Dropout is sometimes called *dilution* because it 'thins' the weight matrix. We usually assign a threshold closer to 1 for the input nodes because we want the information from the training data in the network. Once the network has been fully trained, we do not use dropout when processing new unseen or test data.

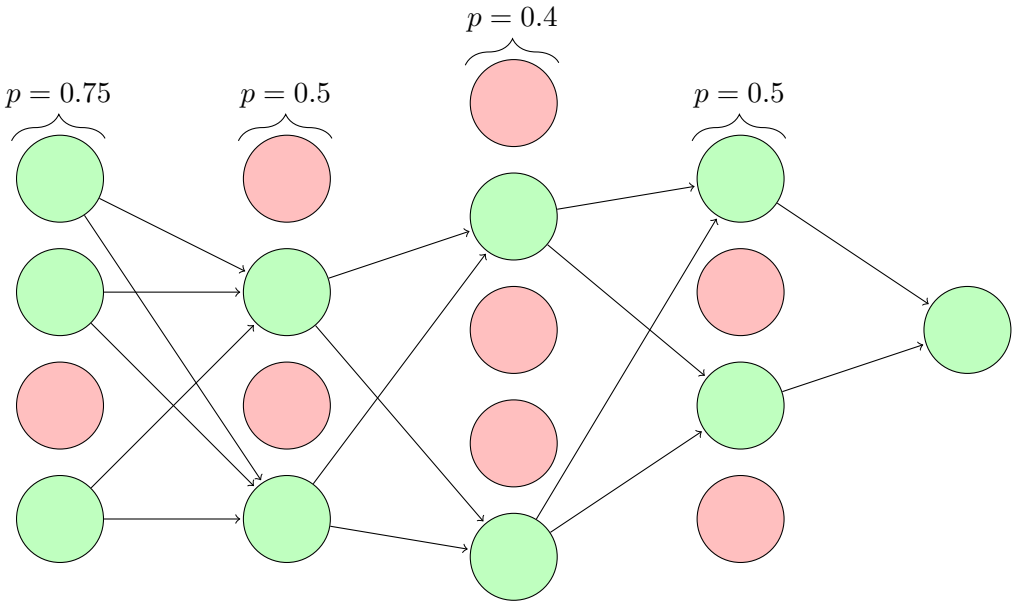


Figure 21: Dropout regularisation on a deep neural network. p is the probability of keeping a node alive in the network.

4.2.4 Data augmentation

If we do not have enough data to train the network to a high enough accuracy we can use techniques to alter the data we do have and create new data. A simple example is images. We can flip, rotate, blur, add or erase random noise and zoom in to images to create 'new' images. Examples of this applied to a photo I took are in figure 22. We can even create synthetic data for certain applications, such as speech recognition, where it has been shown that using speaker data generated by *recurrent neural networks* (RNNs) can improve the learning of new networks [16]. Using more data can help to reduce the variance of networks because it reduces the relative amount of noise in the training set.

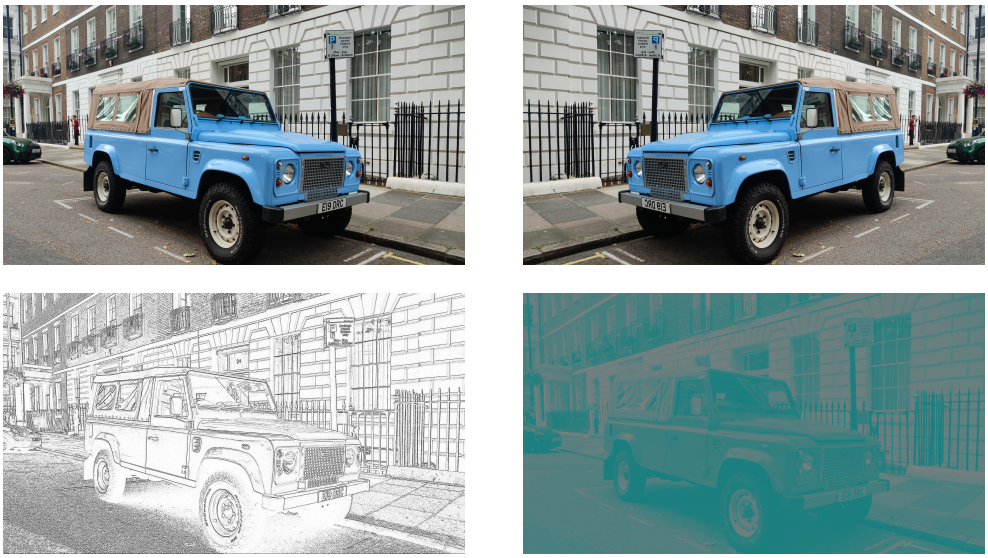


Figure 22: Examples of data augmentation applied to images

4.3 Data Pre-Processing

Preparing data for use in a neural network is an important step in maximising the efficiency of a network for several reasons:

- There may not be enough data to train the network to a high enough accuracy
- The data may not be in a suitable format for a network (e.g. images before their data is tabulated into a color model like RGB or categorical labels that need to be converted to numerical data)
- The data may need to be normalised or standardised to speed up learning
- Large datasets may contain features that don't add new information. To reduce the number of features we can perform *principal component analysis* (PCA)

It is important to pre-process training and test data as well as any data that is put through the network in the exact same way. If we only pre-process the training data then the network will be fitted to that format of data. When being tested or used on new data, it will perform terribly because the new data is not in a format that the network recognises.

4.3.1 Transforming the data

Once we have a large enough dataset we need to make sure it is tabulated into numerical form. This may mean encoding categorical data, or converting colours into numbers via an RGB scale. The most common encoding methods are ordinal encoding and one-hot encoding. Ordinal encoding gives each category a number, while one-hot encoding creates a new feature for each category, and an example from the dataset will take the value 1 for one of these features and 0 for all the other features. Figure 23 shows the difference between the two methods.

Image ID	Label	Label_shoe	Label_shirt	Label_hat
1	Shoe	1	0	0
2	Shirt	0	1	0
3	Hat	0	0	1
4	Shoe	1	0	0

(a) One hot encoding

Image ID	Label	Label_category
1	Shoe	1
2	Shirt	2
3	Hat	3
4	Shoe	1

(b) Ordinal encoding

Figure 23: Two label encoding methods

To convert colours into numbers we first choose the scale that best fits the context (grayscale, RGB, CMYK, etc.) and then use this for our dataset. For a clothing image classification network we could split each image into 28x28 pixels all with red, green and blue values between 0 and 255. If we have 10,000 images this leads to $10000 \times 28 \times 28 \times 3 = 2352000$ individual datapoints.

4.3.2 Normalisation

The next step is to ensure the data is on a common scale. In the Fashion MNIST example we could implement this by dividing each data entry by 255 to get a value between 0 and 1. This is particularly important for problems with features that have very different scales, as neural networks use magnitude to learn without taking units into account. If we were predicting house prices for example, distance to centre of town would be measured in miles and would be between 0 and 50 for houses in one town, while number of bedrooms would only run from 1 to 6 or 7, so the distance to town centre would have a much larger effect on the calculations made by the network than the number of bedrooms. This in turn affects how the network learns because it becomes biased towards features with larger magnitudes. The general formula for normalisation is applied to every feature. Note that x_{min} and x_{max} refer only to the minimum and maximum values of the feature being normalised, not the entire dataset.

$$x' = \frac{x - x_{min}}{x_{max} - x_{min}}$$

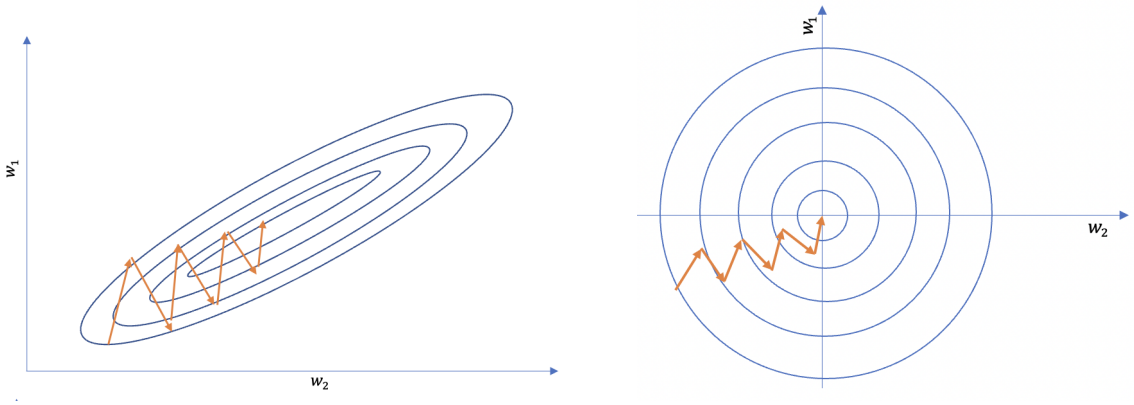
4.3.3 Standardisation

Standardisation is the process of rescaling the data so that it has a mean of 0 and standard deviation of 1. We use the equation

$$x' = \frac{x - \mu}{\sigma}$$

where μ is the mean of the feature and σ is the standard deviation.

The standard deviation is lower if we use normalisation, so outliers are more likely to affect the data. We say that standardisation is more 'robust to outliers' than normalisation. The reason we standardise the input data is because it speeds up the convergence of the model to the best parameters. Recall (from Section 3.6) that we update the weights of the first layer using the partial derivatives of the cost function, $dw_i^{[1]} = x_i^{(i)}(\hat{y}^{(i)} - y^{(i)})\frac{1}{m}$. The size of the steps between each update then clearly depends on the magnitude of x_i. If the features x_i are on different scales then some weights will converge to their optimum values much faster than others. This is why we standardise input data to ensure it has a common scale. Figure 24 shows theoretical cost contour maps before and after standardisation.



(a) Before standardising

(b) After standardising

Figure 24: Contour maps showing gradient descent in a parameter space

4.3.4 Principal Component Analysis

Often datasets can be so high dimensional that it has features that don't give the network new information. A basic example is a health dataset that has features of weight, height, age and BMI. As BMI is a combination of the other features, it is redundant because we can describe a patient in the exact same way with or without their BMI. For complex datasets with hundreds or thousands of features it is important to only use data that is going to help the network learn to avoid wasting resources (time, power, memory) and an overly large network. *Principal component analysis* (PCA) reduces the dimensionality of a dataset by calculating which features explain a predetermined proportion of the variance. It then creates a new set of features called *principal components* which adequately describe the data. PCA also helps to remove noise, reducing variance. The algorithm is performed as follows

1. Calculate the covariance matrix of the standardised (Section 4.3.3) features and then the *eigenvectors* and *eigenvalues*, λ, of this matrix. There will be an eigenvector for

each feature of the dataset. These eigenvectors give the directions in which the data has variance, and the eigenvalues represent these variances. Hence, the sum of the eigenvalues represent the overall variance in the data.

2. Sort the eigenvalues in decreasing order. The largest eigenvalue corresponds to the eigenvector of the first principal component. Each principal component captures some of the variance of the data, with larger eigenvalues corresponding to capturing more of the variance.

3. Choose how much of the variance we wish to capture and use this to decide how many principal components to use as our new set of 'features' to input into the network. To work out how much of the variance is explained by a principal component with eigenvalue λ_j we just need to calculate the fraction

$$\frac{\lambda_j}{\sum_{i=1}^{n_x} \lambda_i}$$

Cumulative explained variance graphs are useful for visualising this.

4.4 Optimisation

We have talked so far about how to make an effective neural network and how to improve its performance. In this section we talk about how to use our resources (time, processing power, memory, etc.) as efficiently as possible.

4.4.1 Mini-Batch Gradient Descent

Until now we have only mentioned *Batch Gradient Descent* (BGD). This is when we feed all the training examples into the network, work out the cost and update the parameters. We repeat this with each iteration to optimise the parameters. This allows us to move towards the minimum of the cost function but we may be doing more calculations than we need to do. Using every training example for each iteration certainly allows us to capture the loss of the model, but it's also possible we could have done this with fewer training examples. If this is possible it is preferable since passing all m training examples through the network requires a lot of computations to be done, taking up time and memory of the computer.

Another form of this is *Stochastic Gradient Descent* (SGD). This is when we feed one (randomly selected) training example through the network, calculate the cost and its gradient, update the parameters, before repeating with the next training example and so on until we have used all training examples. SGD is useful for high-dimensional problems because it reduces the computational power needed for each iteration as the input is of the order $(1 \times n^x)$ rather than $(m \times n^x)$. On the other hand, it is harder to reach optimum parameters with SGD because the error 'jumps around' (has a high variance) with each iteration due to noise in the dataset. Making m updates to the model parameters is also computationally inefficient, particularly for large m.

Mini-Batch Gradient Descent strikes a balance between the two methods. We split the training data into batches and use these for our gradient descent. We introduce a new hyperparameter t, the size of the mini-batches we use. Each iteration is done using only t training examples, selected randomly. Typically, we use powers of 2 to match the memory of GPU or CPU hardware. Small (4,16 or 32) batches are preferable for a wide range of applications because they lead to faster training [17]. Mini-batch gradient descent helps

to avoid being trapped in local minima of the cost function because of the randomness of the batch contents and the fact that we update the parameters more frequently than with BGD. On average as we train using each mini-batch, we reduce the cost and optimise the parameters. However, as figure 25 shows, some mini-batches will produce higher costs than previous mini-batches, due to differences in the randomly selected data.

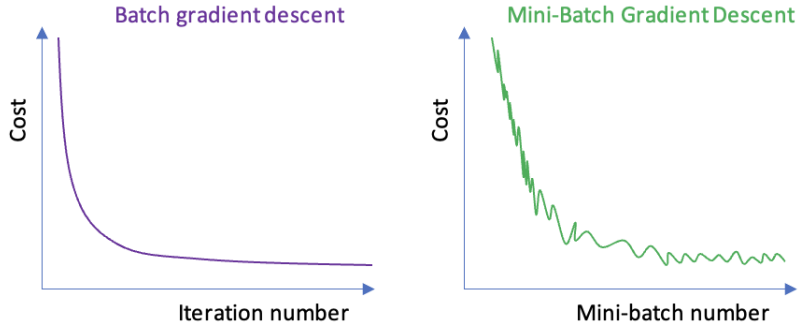


Figure 25: Gradient descent process when using different batch sizes

4.4.2 Batch Normalisation

As data passes through the network, the distribution of the data that is input into each layer changes due to the randomness of the parameter initialisation. This is called *internal covariate shift* and is seen as the change of mean and variance of the data between layers. For the same reasons that we need to standardize our input data, we must do this in between each layer. *Batch Normalisation* [18] centers and scales the data and speeds up learning in the process. We implement this for each mini-batch separately and the process is as follows

1. Calculate the mean, μ, and standard deviation, σ, of the activation function input, Z, of the mini batch for each layer. The calculation in each layer is

$$\mu = \frac{1}{t}\sum_{i=1}^{t} z^{(i)}$$
$$\sigma^2 = \frac{1}{t}\sum_{i=1}^{t}(z^{(i)} - \mu)^2$$

   where t is the mini-batch size.

2. We then normalise these input values

$$\hat{z}^{(i)} = \frac{z^{(i)} - \mu}{\sqrt{\sigma^2 + \epsilon}}$$

   where small ϵ is used to prevent the denominator vanishing.

3. Finally, we introduce two new parameters, γ and β, which the network optimises with gradient descent the same way it optimises the weights and bias.

$$\tilde{z}^{(i)} = \gamma\hat{z}^{(i)} + \beta$$

When we normalise the data before passing it through the network we do it so that the mean is zero and we have unit variance. For batch normalisation the two learnable parameters, γ and β, allow the network to scale the activation inputs to whatever mean and variance give the best (lowest loss) predictions. There is some debate about whether it is best to implement batch normalisation before or after the activation functions. When building a network it can be helpful to test both and compare results. Batch normalisation has been proven empirically to speed up model convergence and training, though the theory that this is because it reduces internal covariate shift is not fully accepted and was challenged by an MIT experiment [19]. The cost function is so high dimensional that it can have many local minima and maxima, making it a hard terrain to find the global minimum of. This paper found that batch normalisation actually smoothed the landscape of the cost function, making optimization easier and thus speeding up training.

4.4.3 Learning Rate Decay

As we get closer to our minimum in gradient descent we want to be careful not to overshoot our target. If the steps between each parameter update are too large, we end up oscillating around the optimum values, never reaching them. However, we also want to have big enough steps that we move quickly from our initial values to the area where our optimum values are located. A fairly low-tech way to do this is to introduce a hyperparameter, the decay rate, which allows us to reduce the learning rate, α, as we get closer to the minimum of our cost function. There are a few ways to do this involving updating the learning rate using a function of the training epochs passed, e (i.e. with each weight update the learning rate gets a bit smaller).

- $\alpha_{new} = \dfrac{1}{1 + \text{decay rate} \times e}\alpha_0$
- $\alpha_{new} = \text{decay rate}^e \times \alpha_0$
- $\alpha_{new} = \dfrac{\text{decay rate}}{\sqrt{e}} \times \alpha_0$

We can also update our learning rate manually by looking at the graph of our cost function improvement over the iterations and deciding when to reduce the learning rate. In other words, if our cost stops improving as we perform more iterations then maybe reducing the learning rate will help to minimise it. There are more sophisticated methods that we can use to hone in on our target parameters, and the ones mentioned in this thesis will all make use of the following concept.

4.4.4 Exponentially Weighted Moving Average

Exponentially weighted moving average (EWMA) is a way to describe the change of a variable with time (or iterations in the case of neural networks). We introduce a hyperparameter, the *smoothing factor* $0 < \beta < 1$, and implement the following

$$\begin{aligned} v_0 &= 0 \\ v_t &= \beta v_{t-1} + (1 - \beta)\theta_t \end{aligned}$$

where v_t is the moving average and θ_t is the measurement of the variable at time t. It is called exponentially weighted because the contribution of older data becomes exponentially

smaller as t increases. For example, with $\beta = 0.9$

$$\begin{aligned} v_3 &= 0.9v_2 + 0.1\theta_3 \\ &= 0.9^3 v_0 + 0.9^2 \cdot 0.1\theta_1 + 0.9 \cdot 0.1\theta_2 + 0.1\theta_3 \end{aligned}$$

The contribution of θ_{k-n} is smaller by a factor of β^n than the contribution of θ_k.

We have that $(1-\epsilon)^{\frac{1}{\epsilon}} \approx \frac{1}{e} = 0.37$ for $|\epsilon| < 1$. This is because $\lim_{\epsilon \to 0}(1-\epsilon)^{\frac{1}{\epsilon}} = e^{-1}$ by using the power series expansion of the natural logarithm. So for $\epsilon = 1-\beta$ we see that the weight of the $\frac{1}{1-\beta}$th term in the weighted sum v_t has decayed to about 37% of the weight of the latest measurement θ_t. So v_t provides a moving average over approximately the last $\left(\frac{1}{1-\beta}\right)$ datapoints. To choose the smoothing factor we need to decide how much previous data we want to be averaging over. For example, with $\beta = 0.9$, $\frac{1}{1-\beta} = 10$ so we have meaningful contributions from the previous 10 datapoints before the contributions become negligible. If we use a smaller β we are taking into account a smaller amount of previous data (e.g. with $\beta = 0.75$, $\frac{1}{1-\beta} = 4$) whereas if use a larger β we take into account more previous data. Therefore, if β is too large recent changes in the value of our parameter θ are not reflected in our value v_t. If β is too small then the EWMA adds nothing to a traditional record of the parameter values because we are just averaging over the last few datapoints.

Since we begin with $v_0 = 0$ we introduce a bias into the system. This is because the parameter value usually does not start from 0 but some initial value we measure, θ_1. Therefore the first few terms have a lower EWMA than they should. To correct this bias we divide our values v_t by $(1-\beta^t)$. As t gets large $(1-\beta^t) \to 1$ so this bias factor stops impacting our v_t, just as our initial value v_0 stops impacting the values v_t after large t. Figure 26 shows EWMA applied three different ways to a dataset.

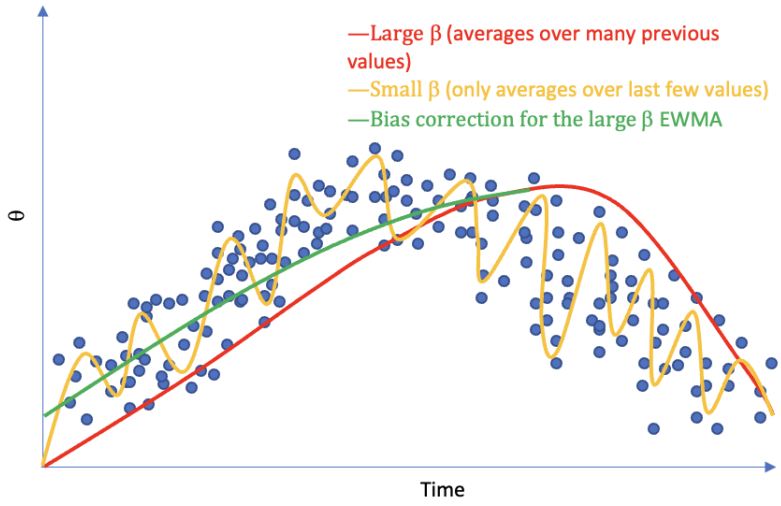


Figure 26: EWMA with and without bias correction applied to data

Optimisation algorithms are the algorithms by which we find the optimum parameters that minimise the cost function. We have seen the standard gradient descent algorithm in Chapter 2. We now take a look at three optimisation algorithms that make use of EWMAs. These all aim to speed up model convergence to the optimum parameters, though come with their own issues which we will now compare.

4.4.5 Gradient Descent with Momentum

We know that different parameters will take different amounts of iterations to optimise. We can visualise this by plotting two parameters, θ_1 and θ_2, on a contour map (figure 27).

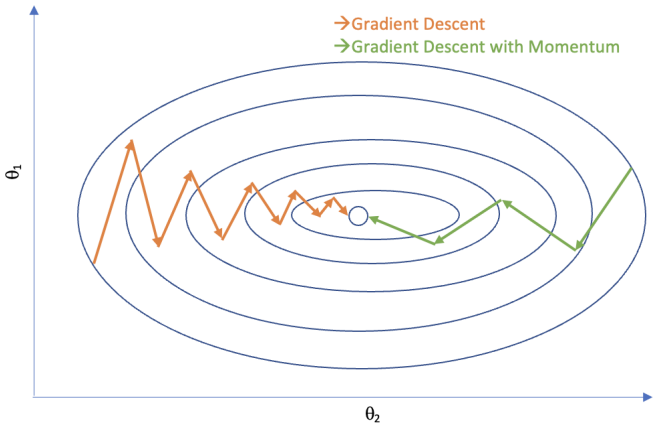


Figure 27: The difference between gradient descent with and without momentum

It is more effective to use a lower learning rate in the θ_1-axis and a higher learning rate in the θ_2-axis. We want to get to the minimum of the surface quickly, but without missing our target in the θ_2-axis by taking steps that are too big. *Gradient descent with momentum* [20] makes use of EWMAs to modify the learning rate as we get closer to our minimum. The typical analogy is of a ball rolling down the side of a bowl: it moves fastest in the direction of the greatest slope. This is what gradient descent with momentum is. We set $V_0 = 0$ and calculate $d\theta$ as normal (θ is any parameter we wish to optimise such as θ_1 or θ_2 from the diagram or a weight or bias from the network). We calculate the EWMA:

$$V_t = \beta V_{t-1} + (1 - \beta) d\theta_t$$

We then update the parameter using the EWMA instead of just $d\theta$:

$$\theta = \theta - \alpha V_t$$

before repeating with each iteration. This works because if we are just oscillating up and down in one axis, such as in θ_1 in the diagram, the EWMA value, V_t, will be around zero so our learning rate is dampened reducing the step size in the θ_1 direction. Similarly, if we are moving repeatedly in the same direction across iterations then we 'gain momentum' and our V_t increases, increasing the steps we take in each iteration, as is the case in the diagram for θ_2. This is akin to the ball speeding up as it moves down a steep part of the bowl. We can tune our new hyperparameter, β, although 0.9 is known to be effective in most contexts. Using momentum helps to speed up training by reducing the amount of time spent oscillating in the wrong direction when converging to optimum parameters. It is faster than traditional gradient descent and makes the convergence process smoother, however it adds a new hyperparameter, β, which has to be selected correctly.

4.4.6 RMSProp

Root mean square propagation (RMSProp) [21] is very similar to Momentum. It also adapts the learning rate for different parameters θ_i depending on how quickly we are getting to

our minimum. As in figure 27 we want to increase the size of our steps in the direction of the θ_2-axis, while dampening our steps in the θ_1-axis. We do this by calculating $d\theta$ as usual and again starting with $S_0 = 0$. We then compute a modified EWMA (it is slightly different to what we did in Momentum as we are now squaring $d\theta$)

$$S_t = \beta S_{t-1} + (1-\beta) {d\theta_t}^2$$

The next stage is to update our parameter. We divide our learning rate and gradient by a function of our EWMA S_t and a small term ϵ ($\approx 10^{-8}$) to prevent any division by 0 errors.

$$\theta = \theta - \alpha \frac{d\theta}{\sqrt{S_t + \epsilon}} \tag{5}$$

RMSProp balances the step size when updating the parameters. If $d\theta$ becomes very large, then when it is multiplied by the term $\frac{1}{\sqrt{S_t+\epsilon}}$, the EWMA S_t carries data from previous values to reduce the magnitude of the term multiplied by α. This ensures we never end up taking too big of a step because the gradient with respect to one parameter is very large and missing the global minimum of the cost function. Therefore this solves the exploding gradient problem. Likewise, if $d\theta$ is becoming very small, then multiplying it by $\frac{1}{\sqrt{S_t+\epsilon}}$ will make the term multiplied by α bigger. This ensures we never get stuck moving too slowly to ever find the minimum of the cost function just because we encounter an area with small gradients in the axis of one parameter. Therefore, this solves the vanishing gradient problem.

On the other hand, the $d\theta_t^2$ term in the RMSProp EWMA has the potential to amplify any noise in the data. In other words, if there are some outliers which give large or tiny partial derivates $d\theta_t$, the effect of these on S_t is amplified by being squared. Also, just like with Momentum, RMSProp requires the careful selection of an additional hyperparameter β when compared with standard gradient descent.

4.4.7 Adam Optimisation

Adaptive Moment Estimation (Adam) is an improvement on RMSProp and Momentum by using the two together [22]. It combines the speed and smoothness of the Momentum algorithm with the balanced step sizes from RMSProp. This means it also prevents vanishing or exploding gradients. We calculate $d\theta_t$ as usual, initialise $V_t = S_t = 0$, before calculating the bias-corrected versions of V_t and S_t:

$$V_t = \frac{\beta_1 V_t + (1-\beta_1) d\theta_t}{1-\beta_1^t}$$
$$S_t = \frac{\beta_2 S_t + (1-\beta_2) {d\theta_t}^2}{1-\beta_2^t}$$

We then update the parameter, θ, using a modified version of the RMSProp equation (5). ϵ plays the same role here as in RMSProp to prevent division by 0.

$$\theta = \theta - \alpha \frac{V_t}{\sqrt{S_t + \epsilon}}$$

Adam does have a couple of disadvantages. Our new hyperparameters, β_1 and β_2, need to be tuned, so now we have 2 extra hyperparameters compared with standard gradient

descent. In general $\beta_1 = \beta_2 = 0.9$ works quite well for most applications. Also, repeatedly computing two EWMAs is computationally expensive and requires more memory to be used to store the previous values of V_t and S_t.

For the reasons mentioned at the start of this subsection, Adam usually provides the fastest algorithm for optimising a network. However, choosing an optimisation algorithm does depend on the context of the problem, and if memory and computational power are limited it may be better to choose a different algorithm.

We discuss the process of choosing the best hyperparameters (which includes which algorithms to use) in this next section.

4.5 Tuning the Hyperparameters

Throughout this Chapter we have introduced several new hyperparameters to adjust the performance of our model. Clearly not all hyperparameters have the same influence on the performance of a network. It is important in machine learning to work efficiently to make the best use of resources. Therefore, when training neural networks we often train several models, all with different combinations of hyperparameters and preferences, at the same time. We then evaluate these using our validation set and choose the best model, which is then evaluated with the test set.

To choose our hyperparameter combinations we can perform a *grid search*, which involves training a model for every combination of choices. This is obviously inefficient and would require both time and power. A more successful method is *random search*. We define a distribution and range for each hyperparameter. We then randomly select a sample of hyperparamaters on our "grid" of the hyperparameter space, and train these models. We repeat the process and evaluate for a chosen number of iterations to choose our model. There are more complex algorithms such as *Bayesian Optimisation* which involve using the posterior expectation of the best hyperparameter combination using the previous iteration.

4.6 Tuning the Hyperparameters for the Fashion MNIST Dataset

We have introduced several hyperparameters and methods for improving a neural network. We now apply these to the neural network we created to classify the Fashion MNIST dataset in Chapter 3. We start by using random search to choose the optimum learning rate and number of nodes in our first layer. Following this, we will optimise our mini-batch sizes, test various optimisation methods and activation functions, and add regularization. Before tuning these hyperparameters we reached a test set accuracy of 80.71% in 89.80 seconds.

To find the optimal number of nodes in the hidden layer and the best learning rate we use Keras' random search tool to iterate through multiple models and choose the best one (listing 2). The random search chose a model with 224 nodes in the hidden layer with a learning rate of 0.01 (figure 28) which gives a test accuracy of 84.17% and required 117 seconds to train.

```
tuner = kt.RandomSearch(build_the_model,
                        objective='val_accuracy',
                        max_trials=10,
                          directory = 'f')

stop_early = tf.keras.callbacks.EarlyStopping(monitor='val_loss', patience
    =5)
```

```
7
8  tuner.search(x_train, y_train, epochs=16, validation_split=0.2, callbacks=[
       stop_early])
9
10 # Get the optimal hyperparameters
11 best_hps=tuner.get_best_hyperparameters(num_trials=1)[0]
12 best_model = tuner.get_best_models()[0]
13
14 print(f"""
15 The hyperparameter search is complete. The optimal number of units in the
       first layer is {best_hps.get('units')} and the optimal learning rate for
        the optimizer is {best_hps.get('learning_rate')}.
16 """)
17 model.summary()
```

Listing 2: Random search code

```
The hyperparameter search is complete. The optimal number of units in
the first densely-connected
layer is 224 and the optimal learning rate for the optimizer
is 0.01.
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
flatten_17 (Flatten)         (None, 784)               0
_________________________________________________________________
dense_31 (Dense)             (None, 224)               175840
_________________________________________________________________
dropout_3 (Dropout)          (None, 224)               0
_________________________________________________________________
dense_32 (Dense)             (None, 10)                2250
=================================================================
Total params: 178,090
Trainable params: 178,090
Non-trainable params: 0
_________________________________________________________________
```

Figure 28: The optimal architecture found by random search

Notice the huge number of trainable parameters in this network - there are 178,090! This means that for this very basic network the surface of the cost function is 178,090-dimensional. This shows how complex the problem of minimising the cost function is, and how impressive it is that neural networks work so well.

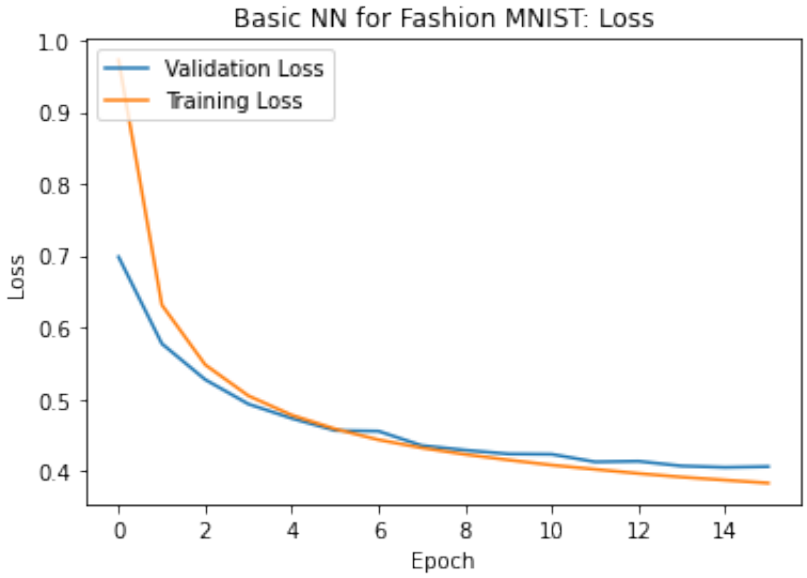


Figure 29: The error of our model

When the model was trained in Chapter 3, we saw that the training accuracy kept increasing but the validation accuracy stopped increasing after the first 9 epochs. This can be seen in figure 29. This indicated that we were overfitting the model to the training data. To improve our model we need to simplify it to reduce overfitting and variance. In this Chapter we discussed several methods to do this through regularisation. We try a few of these here and see how they affect our network. We also alter the hyperparameters of the network and some combinations that were tried are shown in the table below. These include the activation function of the hidden layer, the optimisation algorithm, the initialisation method, the mini-batch size, t, whether or not we use batch normalisation and most importantly which regularisation method is used to reduce the overfitting. It is fairly simple to program these changes using Python and the Keras API. To add in dropout or batch normalisation we treat them the same way we treat the layers of the network and insert where we want them. To add L1 or L2 regularisation we pass our choice to the kernel_regularizer argument in the layer we want to use it. We also do this when we choose our initialisation method and activation function (and any hyperparameters of this e.g. α). This is shown in listing 3.

```
model.add(tf.keras.layers.Dense(units=224, activation=keras.layers.LeakyReLU
    (alpha=0.1), kernel_initializer = keras.initializers.GlorotNormal,
    kernel_regularizer='l2'))
tf.keras.layers.BatchNormalization()
model.add(tf.keras.layers.Dropout(0.3))
model.add(tf.keras.layers.Dense(units=10, activation='softmax',
    kernel_initializer = keras.initializers.HeUniform))
```

Listing 3: Python code which changes hyperparameters

Model	Mini-batch size	Regularisation	Accuracy
tanh, Xavier, Momentum	128	Dropout	85.62
tanh, He, Momentum	128	L1	71.73
ReLU, Xavier, Momentum	128	L2	84.40
Leaky ReLU, Xavier, Momentum	128	Dropout	85.64
Leaky ReLU, Xavier, RMS Prop	128	L1	76.17
Leaky ReLU, Xavier, Adam	64	L2	82.43
Leaky ReLU, Xavier, Adam, batch norm	64	Dropout	86.86
Leaky ReLU, Xavier, Adam	32	L2	83.23
Leaky ReLU, Xavier, Adam, batch norm	32	L1	87.47

Figure 30: Different hyperparameter combinations and their accuracy

We have seen in this Chapter how to improve neural network performance. This involved both accuracy and time taken to reach a high accuracy. After we used random search to test different network architectures we improved the accuracy of our basic network by 3.46%. As can be seen in the figure 30, adding in regularisation only increased the accuracy of our network by a further 3.30%. Some changes such as using L1 regularisation instead of Dropout even drastically worsened the model. To achieve better results for the Fashion MNIST dataset we need to introduce a new type of network, convolutional neural networks.

5 Convolutional Neural Networks

In this section we discuss an important class of neural network, the *convolutional neural network* (CNN). The fully connected neural networks (FCNN) that we have looked at perform well at classifying data when the number of features is low, such as the small images in the Fashion MNIST dataset or the Boston Housing dataset which only has 14 features (crime rate, rooms per house, etc.). For datasets with many features, solving problems with machine learning becomes far more difficult.

CNNs have proven to be extremely successful in modelling datasets with thousands or millions of features. This has applications in *computer vision* as well as natural language processing and financial market prediction. In this report we only talk about their application to image classification, and will build a CNN to classify X-ray images to demonstrate their potential for high accuracy in pattern detection.

5.1 Why do we need a new class of network for images?

As CCTV and social media become omnipresent in modern society, creating endless streams of visual data, the problem of image classification has become a principal driver in machine learning developments. The main issue that the FCNNs that we have looked at in the previous Chapters face is pre-proccessing images into usable data. High definition images are easier for the human brain to process than low definition images due to the increased clarity, hence cameras have become better and more detailed. However, this increased clarity comes at the price of more pixels in the image. These extra pixels mean the image carries more information which makes it harder for neural networks to process. Each pixel is one feature.

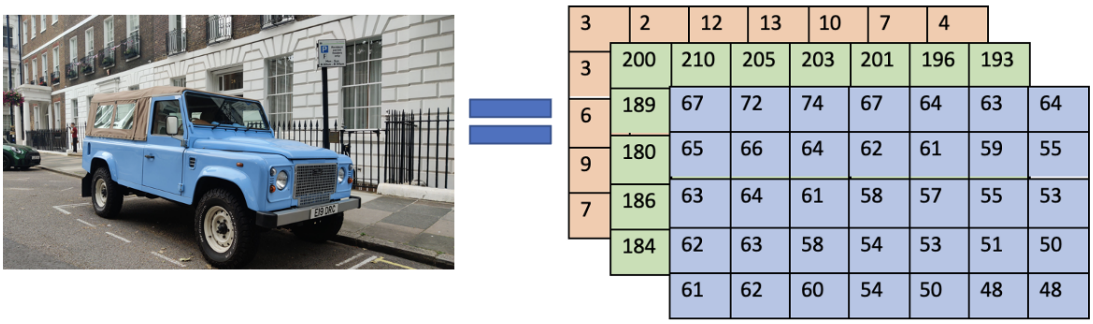


Figure 31: Coloured images are usually represented by 3 channels: red, green and blue

An image is just a matrix of pixels, with each pixel holding a numerical value. For grayscale images, this is a value between 0 (black) and 100 (white). For coloured images this is different. We split the image into multiple matrices, as in figure 31, for different colour channels. When using the RGB model there are three channels (red, green and blue) and each pixel has a score out of 255 for each channel. If we flattened these matrices into a vector to put into a FCNN, it would be very long. For example, a 64×64 pixel image corresponds to a vector of length $64 \cdot 64 \cdot 3 = 12228$ separate features to be fed into a network. If our FCNN has 500 nodes in the first layer then the first layer alone has $12228 \cdot 500 = 6114000$ weights to learn. If we use the CMYK colour model there are four channels, and many images have a resolution much higher than 64×64, with total pixels reaching 33,177,600

for ultra-high-definition cameras. Learning hundreds of millions of parameters is difficult for standard FCNNs and requires a lot of processing power, training data and time, all of which is costly. To solve this problem, CNNs were invented.

CNNs learn and assign importance to features of an image using *convolution* layers. Training data for CNNs requires less pre-processing than FCNNs because the convolution layers, together with *pooling* layers, reduce the dimensionality (number of features) of the data by making the aforementioned matrices smaller. Once the dimensionality of the matrices is reduced, they can be flattened into a vector and passed through a FCNN to be modelled effectively.

5.2 Structure of CNNs

In his 1998 paper Yann LeCun proposed the first CNN [23]. When applied to the MNIST dataset of handwritten numbers his CNN (LeNet) had an error rate of less than half of what the best fully connected neural networks were capable of. His papers is one of the most cited of all time. To this day, CNNs consistently outperform other machine learning techniques for computer vision tasks, achieving almost perfect accuracy on the MNIST and Fashion MNIST datasets.

Reducing the dimensionality of the input data without losing any of the important features that define it is the main problem that CNNs solve. They do this with convolution layers, pooling layers and fully connected layers. Usually there are a series of alternating convolution and pooling layers before the fully connected layers (figure 32).

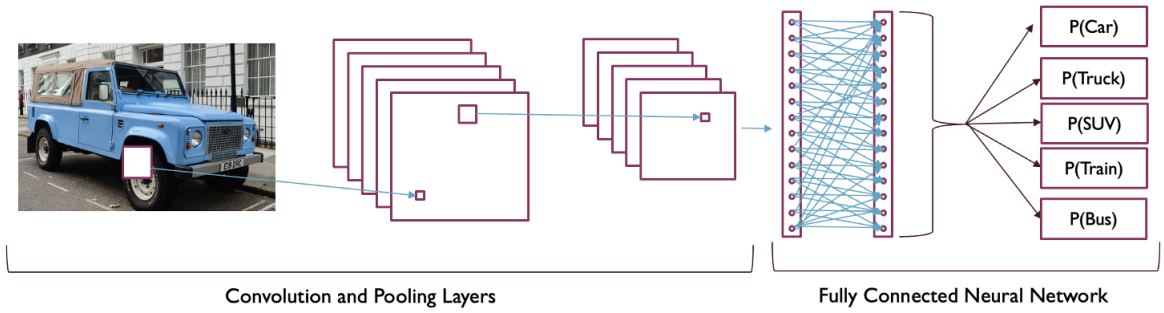


Figure 32: The structure of CNNs

5.3 The Convolution Layer

The strict definition of CNNs is a neural network where at least one layer uses the *convolution operation*. In general, the convolution operation reduces the dimensions of an input. We use a convolution *filter* or *kernel* which is a matrix of dimensions $f \times f$ (generally smaller than the input data matrix). Its elements get learned in the same way as the weights and biases of nodes in FCNNs. These filters are used to distinguish important features of an image such as blur or edges. The filter "strides" along the input matrix, being applied to each $f \times f$ dimension sub-matrix of it. It passes along each row and column of the input matrix and the output is then passed through an activation function, typically ReLU. If we consider a sub-matrix X of the input matrix which has dimension $f \times f$ and a convolution filter C (also with dimensions $f \times f$) then the mathematical operation between these two

matrices is:

$$C * X = \sum_{i=1}^{f} \sum_{j=1}^{f} c_{ij} x_{ij}$$

Each element of the output matrix is produced from one convolution operation. Figure 33 demonstrates a 3×3 vertical edge detection filter convoluted with a 6×6 image. The red box shows how the $(1, 4)$th element (the green box) of the output feature map is calculated.

$$\begin{pmatrix} 0 & 1 & 1 & 1_{x1} & 0_{x0} & 0_{x1} \\ 0 & 0 & 1 & 1_{x0} & 1_{x1} & 0_{x0} \\ 0 & 0 & 0 & 1_{x1} & 1_{x0} & 1_{x1} \\ 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 \end{pmatrix} * \begin{pmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 4 & 3 & 4 & 1 \\ 1 & 2 & 4 & 3 & 3 \\ 1 & 2 & 3 & 4 & 1 \\ 1 & 3 & 3 & 1 & 1 \\ 3 & 3 & 1 & 1 & 0 \end{pmatrix}$$

Figure 33: The convolution operation

The example above only details the process for an image in a 1-dimensional colour model such as grayscale. In the case of RGB images, we have three matrices that represent the image. For this our convolution filter becomes a 3-dimensional 'volume' of matrices to match the dimensions of the image, i.e. it is 3 matrices stacked on top of each other. We can say that this "cuboid" moves along each row and column of our input "cuboid" to give the 3D output feature map. Figure 34 is an example of an edge detection filter (the Laplacian filter) applied to an image.

Figure 34: The Laplacian filter is $\begin{pmatrix} 1 & 0 & 1 \\ 0 & -4 & 0 \\ 1 & 0 & 1 \end{pmatrix}$

There are a few components of the filter that affect the size that our output matrix will be. In the example above we used a *stride*, s, of 1. This is the number of pixels we move the filter along/down between applying the convolution each time. We have also used a *padding*, p, of 0. This the number of rows and columns of empty pixels we add to the outside of the image to alter the dimensions of the output map. If we use no padding, the filter is called *valid*. If we use a padding of $p = \frac{f-1}{2}$ and a stride $s = 1$ then the feature map is the same size as the input image. We call this *same* padding. In general the dimensions

of the output feature map are

$$\lfloor \frac{n+2p-f}{s} + 1 \rfloor \times \lfloor \frac{n+2p-f}{s} + 1 \rfloor$$

The amount of padding, stride and the size of the filter are the hyperparameters of convolution layers. Figure 35 demonstrates a stride of 2, padding of 1 and a grayscale 5×5 image.

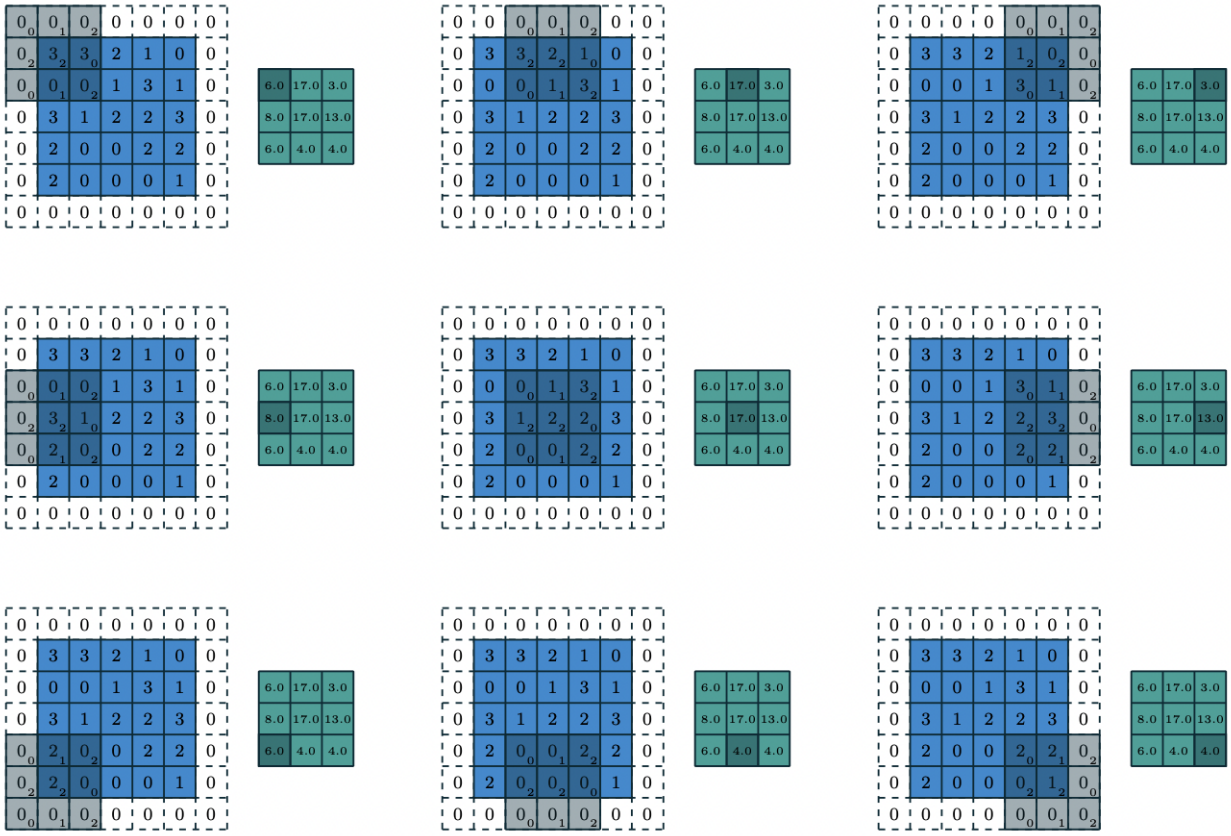


Figure 35: The blue box is the input feature map and the green box is the output feature map. This diagram was created by Dumoulin and Visin [24]

The convolution layers contain several convolution filters. The entries of these filters are learnt like the weights of a FCNN using backpropagation. It learns which filters best detect the defining features in the images that are important. This could be filters that detect vertical edges for example. These filters detect important features such as specific shapes, for example by detecting vertical edges, wherever they appear in the image. Therefore one filter can detect many different features that are important in modelling the dataset, such as the many vertical edges that make up the shape of a shoe. This is different to FCNNs, where each defining feature will have nodes (and hence parameters) associated only with that feature and which cannot assign any importance to other features. This is called *translational invariance* of convolution filters and is one reason why CNNs need to learn fewer parameters than FCNNs to detect the same amount of information from an input.

CNNs have tens or even hundreds of convolution filters and they learn which filters best detect important features of images learning the weights with gradient descent. Convolution layers still need activation functions for the same reason that FCNNs need them: to prevent linearity. Convolution is a linear map so without a nonlinear function in the network it would be no more useful than a linear model. The activation function of convolution filters is generally ReLU because it is easy to compute and doesn't suffer from vanishing gradient problem.

5.4 Pooling

Another function that we can use to reduce the dimensionality of an image is *pooling*. This splits the input image into regions, which it then summarizes using a predetermined method. There are various ways to do this, such as using averages or maximum values. Pooling filters also require a size, $f \times f$, and a stride, s, to be chosen. However, in contrast to convolution filters, the network does not learn these values, they are chosen by the engineer and are hyperparameters. Figure 36 is an example of 2x2 average pooling and 2x2 max pooling.

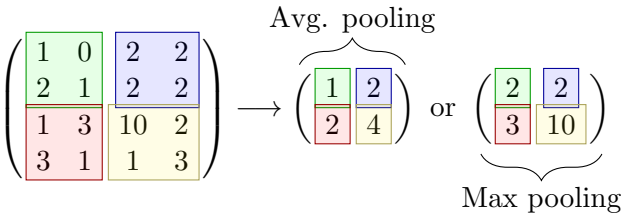


Figure 36: 2×2 pooling example

Pooling helps us to maximise our use of resources because it reduces the size of the input for the layers that follow. This means there are less parameters to be learnt in these layers, and less computing power and time needed. Since pooling can be thought of as summarising the key information from regions of the image, it helps the network to generalise to new data and reduces variance and overfitting.

5.5 Further Layers

CNNs consist of several filter layers which can be a mix of different convolutions and pooling layers. The convolution layers learn important features of the images, and the pooling layers reduce the dimensionality of the images. We then flatten the final feature map matrix (or matrices when using RGB for example) into a vector and pass this through fully connected layers as described in the previous Chapters with standard neural networks. These layers have the same set of hyperparameters and work in the same way as those discussed in Chapters 2-4 of this report. We now introduce one of the most important CNNs of the past 20 years before building our own network to classify X-ray images.

5.6 LeNet-5

Introduced in 1998 by Yann LeCun, Leon Bottou, Yoshua Bengio, and Patrick Haffner, LeNet-5 was a pioneering network in the CNN space. It uses a sequence of 4 convolution and pooling layers before 2 fully connected layers. With 60,000,000 parameters to be learnt, it is relatively small compared to today's CNNs which can have hundreds of millions of parameters. However, it still performs far better than a FCNN. The structure is displayed in figure 37.

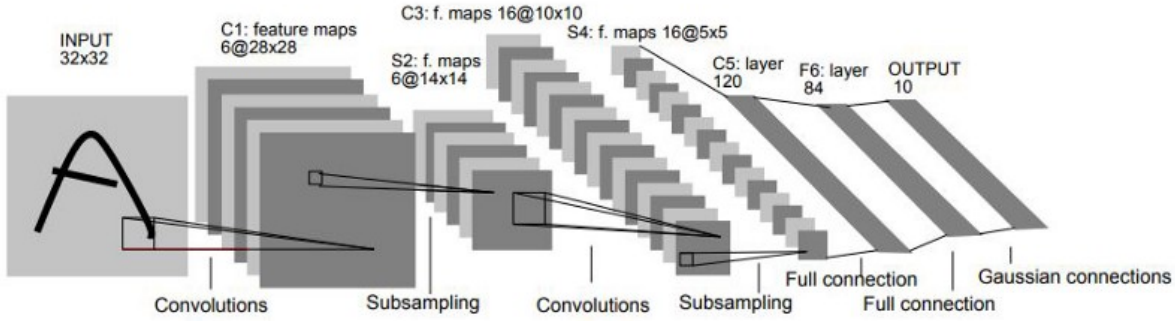


Figure 37: The architecture of LeNet-5. This diagram was first published with the original 1998 paper introducing CNNs [23]

The network was trained with the MNIST dataset which are 32×32 pixel grayscale (so one channel) images. Subsampling in the diagram refers to pooling layers. We see that the network is made from a convolution layer with 6 5×5 filters, an average pooling layer, a convolution layer with 16 5×5 filters, another average pooling layer before a final convolution layer with 120 5×5 filters. This final feature map is then flattened into a vector so that it can be passed through a fully connected neural network. The activation function used for each layer is tanh, partly because it was built before the ReLU function became mainstream. Every convolution filter has a stride of $s = 1$. Each average pooling filter is 2×2 and has a stride of $s = 2$. Figure 38 contains information about the size of the output of each layer. This is useful for visualising how CNNs take input data with many features and reduce their dimensions into something that requires less memory to process but which still contains the important information required to successfully train a network.

Layer	Filters/nodes	Filter size	Stride	Output size
Input				$32 \times 32 \times 1$
Conv 1	6	5×5	1	$28 \times 28 \times 6$
Pooling 1	1	2×2	2	$14 \times 14 \times 6$
Conv 2	16	5×5	1	$10 \times 10 \times 16$
Pooling 2	1	2×2	2	$5 \times 5 \times 16$
Conv 3	120	5×5	1	120
FCNN 1	84			84
FCNN 2	10			10

Figure 38: The architecture of LeNet-5

LeNet-5 has achieved an accuracy above 90% on the Fashion MNIST dataset. We will now build a CNN using the structure of LeNet-5 as inspiration, combined with the concepts covered in the first four Chapters for constructing successful neural networks. We will demonstrate that a simple CNN vastly outperforms FCNNs on the Fashion MNIST dataset, before developing it further and training it on a set of X-ray images to classify lung diseases.

5.7 Building a CNN and Comparison with FCNN

The actual implementation of a CNN is not that different from that of a FCNN. When using an API like Tensorflow's Keras it is in fact only a few additional lines of code when compared to FCNNs. One note is that it is important to ensure that the input is in the right format for the network and we do this in listing 4. We reshape the training image data to specify that it consists of 60,000 examples, each of dimension $28 \times 28 \times 1$. We have to specify the '1' because the network is expecting to be told how many channels each image has and before being processed the images come with a defined shape of 28×28. We do this also for the test images. We also use one-hot encoding for the labels of the images and convert the pixel values from integers to floats so that we can normalise them by dividing by 255.

```
# load dataset
(trainX, trainY), (testX, testY) = fashion_mnist.load_data()
# reshape dataset to have one channel
trainX = trainX.reshape(60000, 28, 28, 1))
testX = testX.reshape((10000, 28, 28, 1))
# encode labels
trainY = to_categorical(trainY)
testY = to_categorical(testY)}
# convert from integers to floats and normalise to range 0-1
train_norm = trainX.astype('float32') / 255
test_norm = testX.astype('float32') / 255
```

Listing 4: Loading and pre-processing the dataset

The next step is to build our model (listing 5). In the Keras API we define the model as a sequential object and then add each layer to this object. To classify the Fashion MNIST dataset we are going to use a fairly simple network architecture. Our network has two convolution layers, two average pooling layers and two fully connected layers. The details of each layer are shown in figure 39. We will not use any regularisation at first to see how much benefit the convolution layers add when compared with the tuned and regularised FCNN we built in Chapter 4.

```
# define cnn model
model = Sequential()
model.add(Conv2D(filters=128, kernel_size=3, padding='same', activation='
    relu', input_shape=(28,28,1)))
model.add(AveragePooling2D(pool_size=2))
model.add(Conv2D(filters=16, kernel_size=3, padding='same', activation='relu
    '))
model.add(AveragePooling2D(pool_size=2))
model.add(Flatten())
model.add(Dense(128, activation='relu'))
model.add(Dense(10, activation='softmax'))
# compile the model
```

```
model.compile(optimizer='Adam', loss='categorical_crossentropy', metrics=['
    accuracy'])
# print the model architecture
model.summary()
```

Listing 5: Building the CNN

```
Model: "sequential_18"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_37 (Conv2D)           (None, 28, 28, 128)       1280
_________________________________________________________________
max_pooling2d_14 (MaxPooling (None, 14, 14, 128)       0
_________________________________________________________________
conv2d_38 (Conv2D)           (None, 14, 14, 16)        18448
_________________________________________________________________
average_pooling2d_21 (Averag (None, 7, 7, 16)          0
_________________________________________________________________
flatten_17 (Flatten)         (None, 784)               0
_________________________________________________________________
dense_35 (Dense)             (None, 128)               100480
_________________________________________________________________
dense_36 (Dense)             (None, 10)                1290
=================================================================
Total params: 121,498
Trainable params: 121,498
Non-trainable params: 0
_________________________________________________________________
```

Figure 39: The architecture of the CNN to classify the Fashion MNIST dataset

The test set accuracy of this network was 91.96%, demonstrating a significant improvement on the FCNN from Chapter 4 where we only reached 87.47% after tuning the hyperparameters using random search and adding in dropout and L2 regularisation. Recall that the FCNN we built and optimised in Chapter 4 had 178,090 trainable parameters while this CNN has 121,498 trainable parameters. With only 68% the number of parameters and no optimisation or regularisation we have trained a model that can classify the clothing images with 4.49% higher accuracy. For high definition images large convolutional neural networks in use nowadays can have more than 100 times as many trainable parameters. VGG16 is one such popular network which has 138 million trainable parameters. With sufficient regularisation and careful hyperparameter tuning, CNNs can achieve accuracies higher than 96% on the Fashion MNIST dataset.

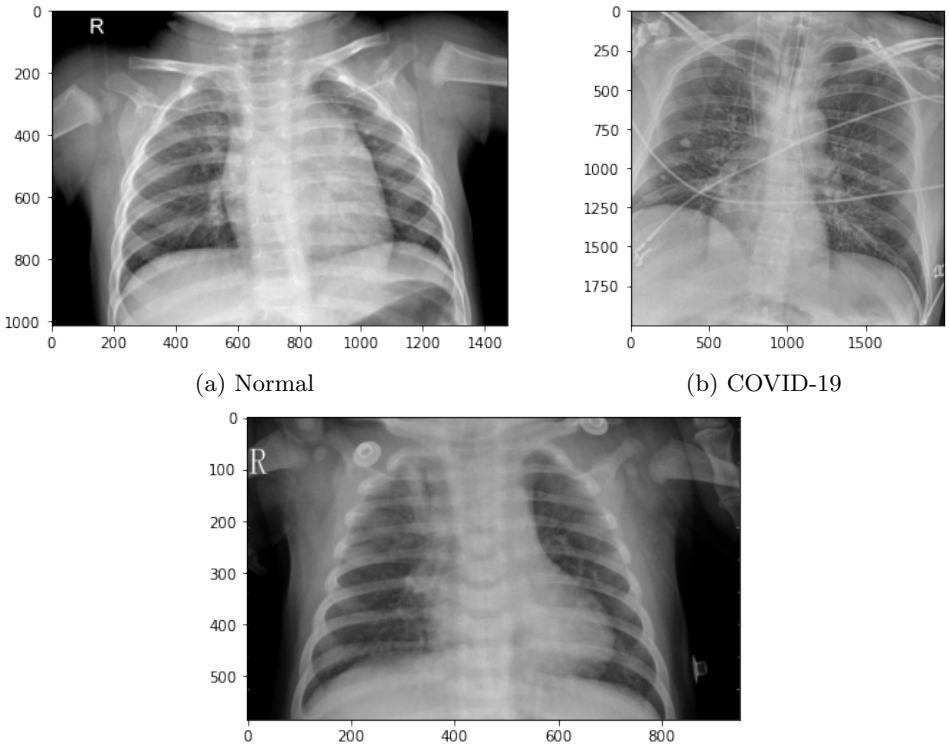
We will tune the hyperparameters of a CNN, though we are going to attempt a trickier classification task. We will classify X-ray scans as healthy, infected with COVID-19 or otherwise unhealthy.

5.8 Building a CNN to Detect COVID-19

During the pandemic, testing for COVID-19 via nasal and throat swab has become the norm. However, there are several reasons that it is sub-optimal. It requires lab processing, has transport costs, requires trained staff, uses single-use plastics which harm the environment and some people have health reasons that do not permit them to perform this kind of test. We will build a CNN that uses X-rays of lungs to diagnose COVID-19. Though these scans also require expensive infrastructure they do have advantages for diagnosing COVID-19. If a patient arrives at the emergency room struggling to breath, it is of vital importance to diagnose them as soon as possible. It usually takes a few hours to get a PCR test result, as well as needing staff to process it. However, if the patient gets a scan then no staff or additional time are needed to obtain the result. They just need to input the image into the CNN and almost instantly it can give a result, hopefully with good accuracy if we build it well.

5.8.1 Visualising and Pre-Processing the Data

The data we are using is combined from three sources. We use Paul Mooney's pneumonia chest X-ray dataset [25], and two COVID-19 chest X-ray datasets: one from 2020 [26] and one from 2022 [27]. There are 576 COVID-19 chest X-rays, 4273 Non-COVID infections (Viral or Bacterial Pneumonia), and 1583 normal chest X-rays. Note how similar the X-rays look (figure 40)- we need a CNN to find the relationships in the images that we don't notice. We will use 80% of the images to train the network. The remaining 20% will be used as validation and test data.



(a) Normal

(b) COVID-19

(c) Pneumonia

Figure 40: Lung X-rays

Also note that the images are all different sizes. We need to do some pre-processing to standardize the data. In order to deal with the relatively small amount of training data (5144 training images compared to 60,000 in the Fashion MNIST dataset) we need to use data augmentation to increase the amount of training data we have. The Keras API has a helpful function that allows us to do this. We use the ImageDataGenerator to generate new images from the existing ones in listing 6. The arguments such as width_shift_range and zoom_range all define the methods of data augmentation. We also set it to divide the value of each pixel by 255 to standardize the data and specify that we want the images to be grayscale (i.e. just have one channel). Once we have defined our generator we apply it to the training set and test set, and at the same time we reshape the images to be of size 400×400.

```
#Data Augmentation
generator = ImageDataGenerator(
    width_shift_range=0.09,
    height_shift_range=0.09,
    shear_range=0.1,
    zoom_range=0.15,
    fill_mode='nearest',
    rescale=1/255,
)

# Apply the generator to the training images.
# Here we also reshape the images to grayscale 400x400 and choose a batch
    size = 16.
# Batch size specifies how many new images are created per original image.

train_generator = generator.flow_from_directory (
    train_directory,
    target_size=(400,400),
    color_mode='grayscale',
    class_mode='categorical',
    batch_size=16,
    shuffle=True,
)

# Repeat for the test images

test_generator = generator.flow_from_directory (
    test_directory,
    target_size=(400,400),
    color_mode='grayscale',
    class_mode='categorical',
    batch_size=16,
    shuffle=True,
)
```

Listing 6: Data augmentation applied to the test and training data

5.8.2 Programming the CNN

The next stage is to define the hyperparameters of our network and build the structure (listing 7). We use three convolution and max pooling layers each, before a FCNN, also with three layers. The output layer has 3 nodes with the softmax activation function so that we can classify not just COVID-19 or healthy, but also if there is Pneumonia present. We

will use batch normalisation and early stopping. We apply L1 regularization with $\lambda = 0.01$. Figure 41 gives a scaled down diagram of the structure.

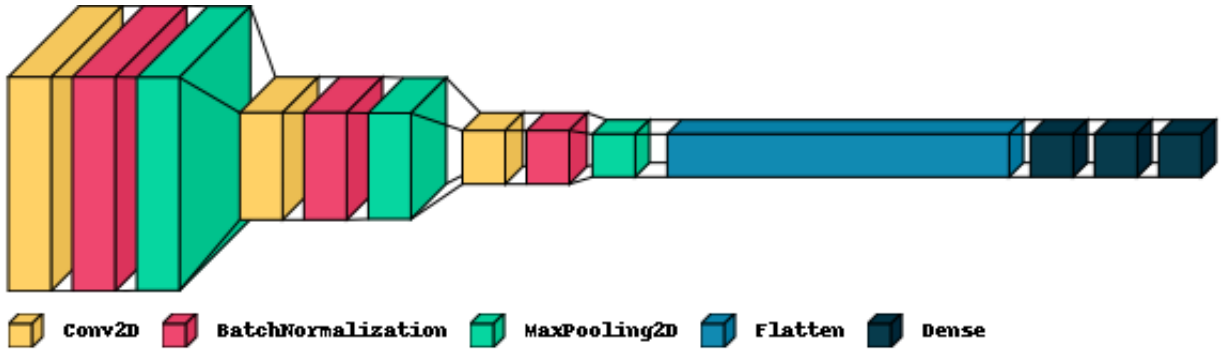


Figure 41: The structure of the COVID-19 CNN classifier

```
# build model
model = Sequential()
# convolution layer and batch norm
model.add(Conv2D(filters = 16, padding = "same", kernel_size = 2, strides = 2, kernel_initializer = 'HeNormal',
                 activation = "relu", input_shape = (400,400,1)))
model.add(BatchNormalization())
# max pooling layer
model.add(MaxPool2D(pool_size = (2,2), strides = 1))

# convolution layer and batch norm
model.add(Conv2D(filters = 32, padding = "same", kernel_size = 2, strides = 2, kernel_initializer = 'HeNormal',
                 activation = "relu"))
model.add(BatchNormalization())
# max pooling layer
model.add(MaxPool2D(pool_size = (2,2), strides = 1))

# convolution layer and batch norm
model.add(Conv2D(filters = 64, padding = "same", kernel_size = 2, strides = 2, kernel_initializer = 'HeNormal',
                 activation = "relu"))
model.add(BatchNormalization())
# map pooling layer
model.add(MaxPool2D(pool_size = (2,2), strides = (2,2)))

# flatten the output feature map into a vector before passing through FCNN
model.add(Flatten())
# fully connected layer 1
model.add(Dense(units = 128, kernel_regularizer = L1(0.01), kernel_initializer = 'HeNormal',
                activation = "relu"))
# fully connected layer 2
model.add(Dense(units = 64, kernel_regularizer = L1(0.01), kernel_initializer = 'HeNormal',
                activation = "relu"))
# output layer
model.add(Dense(units = 3, activation = "softmax"))

```

```
# choose Adam optimisation
model.compile(loss = "categorical_crossentropy", optimizer = "adam", metrics
    = ["accuracy"])

# early stopping
early_stop = EarlyStopping(monitor="val_loss", mode="min", patience=6)

# fit model to training data
history = model.fit(train_generator, validation_data=test_generator, epochs
   =1, callbacks=[early_stop])
```

Listing 7: Building, training and evaluating the CNN

This CNN had an accuracy of 89.52% on the test set, a very encouraging result. To achieve this accuracy we have pulled together the work from all of the previous 4 Chapters. We used what we learnt in Chapters 2 and 3 to build and train the model and we improved this model with batch normalisation, regularisation and Adam optimisation as discussed in Chapter 4. Additionally from Chapter 4, we used data augmentation to deal with a lack of data and pre-processed the images into a uniform size, since they all came from different sources with different sizes. This presents an exciting new way of testing for COVID-19, though as with any medical tool would have to go through rigorous testing and ethical frameworks. It was fairly simple to create: we only needed 5144 training images and Tensorflow's Keras public library. This shows the immense power of CNNs in solving complex problems. The detailed architecture of the CNN is given in figure 42.

```
Model: "sequential_2"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_6 (Conv2D)            (None, 200, 200, 16)      80
_________________________________________________________________
batch_normalization_6 (Batch (None, 200, 200, 16)      64
_________________________________________________________________
max_pooling2d_6 (MaxPooling2 (None, 199, 199, 16)      0
_________________________________________________________________
conv2d_7 (Conv2D)            (None, 100, 100, 32)      2080
_________________________________________________________________
batch_normalization_7 (Batch (None, 100, 100, 32)      128
_________________________________________________________________
max_pooling2d_7 (MaxPooling2 (None, 99, 99, 32)        0
_________________________________________________________________
conv2d_8 (Conv2D)            (None, 50, 50, 64)        8256
_________________________________________________________________
batch_normalization_8 (Batch (None, 50, 50, 64)        256
_________________________________________________________________
max_pooling2d_8 (MaxPooling2 (None, 25, 25, 64)        0
_________________________________________________________________
flatten_2 (Flatten)          (None, 40000)             0
_________________________________________________________________
dense_6 (Dense)              (None, 128)               5120128
_________________________________________________________________
dense_7 (Dense)              (None, 64)                8256
_________________________________________________________________
dense_8 (Dense)              (None, 3)                 195
=================================================================
Total params: 5,139,443
Trainable params: 5,139,219
Non-trainable params: 224
```

Figure 42: The architecture of the CNN to classify X-ray images

This Chapter has introduced convolutional neural networks, defined their structure and explained their processes. The example of LeNet-5 was introduced to teach the reader how to create a successful CNN. We then took this a step further, using X-ray images to create a useful CNN that can detect COVID-19 with a high enough accuracy that it could realistically be used in a medical setting.

6 Conclusion and Future Research

In this thesis we have discussed how to build successful neural networks, a powerful tool that enables machine learning. We discussed the fundamentals of the mathematics behind neural networks before delving deeper into their architecture to see how effective they can become. The proficiency for modelling complex datasets can add significant value to important industries, and this was demonstrated by the CNN built in Chapter 5 to classify X-rays as COVID-19 positive or not. To get to this point, we began by introducing the basic ideas behind neural networks, and the motivation for using them in Chapter 1.

Chapter 2 introduced the structure of neural networks and the essential concept of forward propagation. We introduced the process of backpropagation in Chapter 3, looking at the mathematics behind how we find the best parameters for a network using gradient descent. We discussed the basic choices one has when putting together the building blocks of a network, such as which activation function to use for different nodes or how to initialise weights. We discussed the mathematical processes in explicit detail with regards to a non-binary classification problem. By this point we had learnt enough to build and train a neural network. At the end of this Chapter we looked at how a neural network is implemented using the Python programming language and applied what we had learnt so far to the Fashion MNIST dataset.

We built a decent network in Chapter 3, but it was not as accurate as it could be. In Chapter 4, we established methods for inspecting the performance of neural networks and improving this performance. Common issues, most notably high bias or high variance, were discussed and we looked in detail at various regularisation techniques which could reduce variance. One of the most practical subsections of this thesis was contained in Chapter 4: Data Pre-Processing. This introduced the best practice for preparing large datasets for use in a neural network. We then looked at different optimisation techniques to train networks as quickly and efficiently as possible, before applying what we had learnt in this Chapter to the network we had built in Chapter 3. We saw a significant improvement (6.76%) in accuracy on the Fashion MNIST dataset due to this.

After we tuned the hyperparameters of our network, we still only achieved an 87% classification accuracy on the Fashion MNIST images. To achieve better results, we introduced a new type of network in Chapter 5, the convolutional neural network. This combined two new operations, convolutions and pooling, with the fully connected neural networks we had studied in the first four Chapters. CNNs can achieve remarkably high accuracy on extremely complex datasets and without any tuning of the hyperparameters or regularisation we built a CNN that achieved an accuracy of 91% on the Fashion MNIST dataset, providing an improvement on the FCNN. CNNs have applications ranging from modelling financial time series to natural language processing. We focused on their application to image classification only, though they have several other applications within the field of computer vision such as object segmentation within images. We built a CNN to look at patients' lung X-rays and classify the image as indicating a presence of COVID-19, another disease (such as pneumonia) or no disease detected. This CNN had a remarkable test set accuracy of 89.52% with a relatively small amount of training data, offering a new option for small scale COVID-19 testing.

In large part the success of CNNs is not yet fully explained. There are logical reasons why they should outperform FCNNs, but not to the extent that they are managing to do.

Current research is focused on this, as well as less abstract ideas. There is always pressure to build smaller, faster and simpler networks that can generalise well to new datasets. Optimising algorithms so that networks can be trained using less time and processing power remains a key objective for financial officers in machine learning start ups. As ever, applying these CNNs to new contexts is often where the largest gains can be made, a challenge only limited by our imaginations. Self-driving cars is one such area where the benefit of CNNs will be felt. The ability of the car to accurately detect objects around it will be essential to its safety. Research at the moment is focused on using CNNs in combination with other types of neural networks such as *recurrent neural networks* (RNNs) to solve complex problems. For example, a combined neural network made up of a CNN and RNN achieved a 96% accuracy at separating healthy eyes and eyes with glaucoma [28].

Neural networks are one tool in a large toolbox that data scientists can utilise today. They offer potential in a range of industries, and, as the amount of data collected continues to increase, become an increasingly useful tool. As of April 2022 there are more than 9,000 machine learning focused companies [29], and countless non-tech companies are incorporating machine learning into their business models. Neural networks are becoming essential to modern business. This thesis has offered an introduction to them, with a strong focus on the mathematics required for them to work, and this is motivated by their huge potential to disrupt industry and promote technological change in every sector of industry, public or private.

7 References

[1] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel *et al.*, "A general reinforcement learning algorithm that masters chess, shogi, and go through self-play," *Science*, vol. 362, no. 6419, pp. 1140–1144, 2018.

[2] K.-F. Lee, *AI superpowers: China, Silicon Valley, and the new world order*. Houghton Mifflin, 2018.

[3] S. Xiaoye, "Analysis of bytedance," Ph.D. dissertation, Ph. D. Dissertation. Swiss Federal Institute of Technology Zurich, 2019.

[4] P. Anderson, J. McGuffee, and D. Uminsky, "Data science as an undergraduate degree," in *Proceedings of the 45th ACM technical symposium on Computer science education*, 2014, pp. 705–706.

[5] R. Wigglesworth, "Jane street: the top wall street firm 'no one's heard of'," *Financial Times*, 2021.

[6] D. B. D. Bank, "Financial data supplement q3 2021," 2021.

[7] D. Zhang, S. Mishra, E. Brynjolfsson, J. Etchemendy, D. Ganguli, B. Grosz, T. Lyons, J. Manyika, J. C. Niebles, M. Sellitto *et al.*, "The ai index 2021 annual report," *arXiv preprint arXiv:2103.06312*, 2021.

[8] A. Ng, "Deep learning— coursera," 2017.

[9] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015, software available from tensorflow.org. [Online]. Available: https://www.tensorflow.org/

[10] F. Chollet *et al.*, "Keras," https://keras.io, 2015.

[11] G. Van Rossum and F. L. Drake Jr, *Python reference manual*. Centrum voor Wiskunde en Informatica Amsterdam, 1995.

[12] D. Z. Brodtman, "An introduction to neural networks," https://github.com/zackb9/Project-III.git , 2022.

[13] K. He, X. Zhang, S. Ren, and J. Sun, "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification," in *2015 IEEE International Conference on Computer Vision (ICCV)*, 2015, pp. 1026–1034.

[14] X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks," in *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, ser. Proceedings of Machine Learning Research, Y. W. Teh and M. Titterington, Eds., vol. 9. Chia Laguna Resort,

Sardinia, Italy: PMLR, 13–15 May 2010, pp. 249–256. [Online]. Available: https://proceedings.mlr.press/v9/glorot10a.html

[15] G. E. Hinton, N. Srivastava, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Improving neural networks by preventing co-adaptation of feature detectors," *ArXiv*, vol. abs/1207.0580, 2012.

[16] J. J. Bird, D. R. Faria, C. Premebida, A. Ekárt, and P. P. S. Ayrosa, "Overcoming data scarcity in speaker identification: Dataset augmentation with synthetic mfccs via character-level rnn," in *2020 IEEE International Conference on Autonomous Robot Systems and Competitions (ICARSC)*, 2020, pp. 146–151.

[17] D. Masters and C. Luschi, "Revisiting small batch training for deep neural networks," 2018.

[18] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," *CoRR*, vol. abs/1502.03167, 2015. [Online]. Available: http://arxiv.org/abs/1502.03167

[19] S. Santurkar, D. Tsipras, A. Ilyas, and A. Madry, "How does batch normalization help optimization?" *Advances in neural information processing systems*, vol. 31, 2018.

[20] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *Nature*, vol. 323, pp. 533–536, 1986.

[21] T. Tieleman, G. Hinton *et al.*, "Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude," *COURSERA: Neural networks for machine learning*, vol. 4, no. 2, pp. 26–31, 2012.

[22] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," 2017.

[23] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[24] V. Dumoulin and F. Visin, "A guide to convolution arithmetic for deep learning," *arXiv preprint arXiv:1603.07285*, 2016.

[25] D. S. Kermany, M. Goldbaum, W. Cai, C. C. Valentim, H. Liang, S. L. Baxter, A. McKeown, G. Yang, X. Wu, F. Yan *et al.*, "Identifying medical diagnoses and treatable diseases by image-based deep learning," *Cell*, vol. 172, no. 5, pp. 1122–1131, 2018.

[26] J. P. Cohen, P. Morrison, and L. Dao, "Covid-19 image data collection," *arXiv*, 2020. [Online]. Available: https://github.com/ieee8023/covid-chestxray-dataset

[27] L. Wang, A. Wong, Z. Q. Lin, J. Lee, P. McInnis, A. Chung, M. Ross, B. van Berlo, and A. Ebadi, "Figure 1 covid-19 chest x-ray dataset initiative," *Accessed: April*, vol. 19, 2022.

[28] S. Gheisari, S. Shariflou, J. Phu, P. J. Kennedy, A. Agar, M. Kalloniatis, and S. M. Golzan, "A combined convolutional and recurrent neural network for enhanced glaucoma detection," *Scientific reports*, vol. 11, no. 1, pp. 1–11, 2021.

[29] S. Ghosh, "Top machine learning startups to watch in 2021," 2021. [Online]. Available: https://neptune.ai/blog/top-machine-learning-startups-to-watch

A Notation

Standard notations for Deep Learning

This document has the purpose of discussing a new standard for deep learning mathematical notations.

1 Neural Networks Notations.

General comments:

· superscript (i) will denote the i^{th} training example while superscript [l] will denote the l^{th} layer

Sizes:

·m : number of examples in the dataset

·n_x : input size

·n_y : output size (or number of classes)

·$n_h^{[l]}$: number of hidden units of the l^{th} layer

In a for loop, it is possible to denote $n_x = n_h^{[0]}$ and $n_y = n_h{}^{[\text{number of layers } +1]}$.

·L : number of layers in the network.

Objects:

·$X \in \mathbb{R}^{n_x \times m}$ is the input matrix

·$x^{(i)} \in \mathbb{R}^{n_x}$ is the i^{th}example represented as a column vector

· $Y \in \mathbb{R}^{n_y \times m}$ is the label matrix

· $y^{(i)} \in \mathbb{R}^{n_y}$ is the output label for the i^{th} example

· $W^{[l]} \in \mathbb{R}^{\text{number of units in next layer} \times \text{number of units in the previous layer}}$ is the weight matrix,superscript [l] indicates the layer

· $b^{[l]} \in \mathbb{R}^{\text{number of units in next layer}}$ is the bias vector in the l^{th} layer

· $\hat{y} \in \mathbb{R}^{n_y}$ is the predicted output vector. It can also be denoted $a^{[L]}$ where L is the number of layers in the network.

Common forward propagation equation examples:

$a = g^{[l]}(W_x x^{(i)} + b_1) = g^{[l]}(z_1)$ where $g^{[l]}$ denotes the l^{th} layer activation function

$\hat{y}^{(i)} = softmax(W_h h + b_2)$

· General Activation Formula: $a_j^{[l]} = g^{[l]}(\sum_k w_{jk}^{[l]} a_k^{[l-1]} + b_j^{[l]}) = g^{[l]}(z_j^{[l]})$

· $J(x, W, b, y)$ or $J(\hat{y}, y)$ denote the cost function.

Examples of cost function:

· $J_{CE}(\hat{y}, y) = -\sum_{i=0}^{m} y^{(i)} \log \hat{y}^{(i)}$

· $J_1(\hat{y}, y) = \sum_{i=0}^{m} \mid y^{(i)} - \hat{y}^{(i)} \mid$

B Derivative of the softmax function

The softmax function is defined for each node as

$$\sigma(z)_i = \frac{\exp(z_i)}{\sum_{c=1}^{C} \exp(z_c)}$$

which outputs a vector $\hat{y}^{(i)} \in \mathbb{R}^C$. We now show that the derivative of this function is $\frac{\partial \sigma(z)_i}{\partial z_j} = \sigma(z_i)\left(\delta_{ij} - \sigma(z_j)\right)$. There are two cases to consider, both of which are solved using the quotient rule for differentiation. We first consider the case when $i = j$:

$$\begin{aligned}
\frac{\partial \sigma(z)_i}{\partial z_j} &= \frac{\partial}{\partial z_j}\left(\frac{\exp(z_i)}{\sum_c \exp(z_c)}\right) \\
&= \frac{\exp(z_i)\sum_c \exp(z_c) - \exp(z_i)\exp(z_j)}{\left(\sum_c \exp(z_c)\right)^2} \\
&= \sigma(z_i)\frac{\sum_c \exp(z_c) - \exp(z_j)}{\sum_c \exp(z_c)} \\
&= \sigma(z_i)\left(1 - \sigma(z_j)\right)
\end{aligned}$$

The next case to consider is when $i \neq j$:

$$\begin{aligned}
\frac{\partial \sigma(z)_i}{\partial z_j} &= \frac{\partial}{\partial z_j}\left(\frac{\exp(z_i)}{\sum_c \exp(z_c)}\right) \\
&= \frac{0 \cdot \sum_c \exp(z_c) - \exp(z_i)\exp(z_j)}{\left(\sum_c \exp(z_c)\right)^2} \\
&= \sigma(z_i)\frac{-\exp(z_j)}{\sum_c \exp(z_c)} \\
&= \sigma(z_i)\left(-\sigma(z_j)\right)
\end{aligned}$$

We can use the Kronecker delta function to combine these two cases into one function. The Kronecker delta function is defined as

$$\delta_{ij} = \begin{cases} 1 & \text{for } i = j \\ 0 & \text{for } i \neq j \end{cases}$$

So it is now simple to see that

$$\frac{\partial \sigma(z)_i}{\partial z_j} = \sigma(z_i)\left(\delta_{ij} - \sigma(z_j)\right)$$

as claimed.