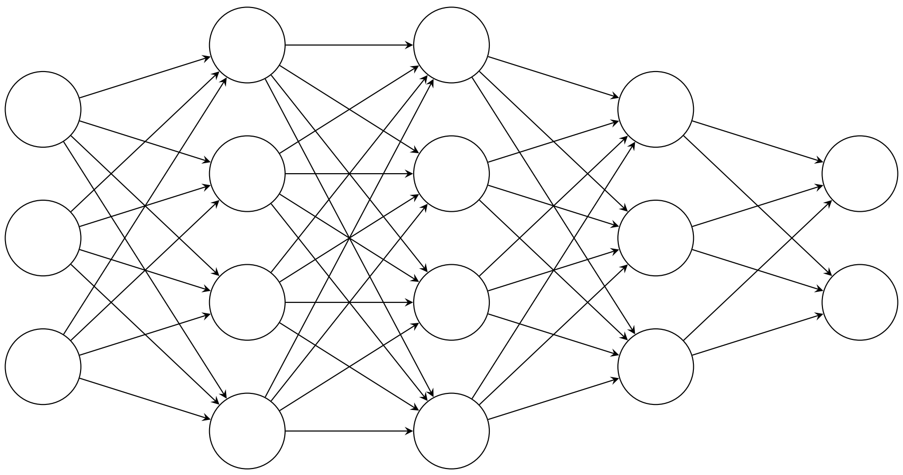
A Mathematical Overview of Neural Networks

Joseph A. Davies



Abstract

This project provides an introduction to the mathematics underlying neural networks. It covers the fundamental components of neural networks, explaining their interactions through mathematical models. After establishing a foundational understanding of how neural networks process information and learn from this information, the project demonstrates the application of these concepts by building neural networks based solely on the concepts learned. The project then explores advanced topics beyond the core concepts, uncovering insights that could enhance neural network performance. This work aims to deepen the understanding of neural network behaviour both theoretically and practically.

Contents

1 Introduction

The fields of neural networks, machine learning, and artificial intelligence have ingrained themselves as commonplace in modern life, affecting practically anyone who uses computing devices. As AI technology continues to drive industry evolution and reshape the world as we know it, there is no better time to appreciate the mathematically rich foundation that underpins these advancements – without which none of these technologies would be possible.

The groundwork for neural networks as a mathematical concept can be traced back to 1943 with the work of Warren McCulloch and Walter Pitts. In their influential paper 'A Logical Calculus of the Ideas Immanent in Nervous Activity' [1], McCulloch and Pitts took a logical, computational approach to an understanding of biological neurons and how they transmit information. While this paper did not initially gain traction in its intended field of neurobiology, it is now recognised as the foundation for the field of neural networks.

Building on these ideas, it is worth mentioning Donald Hebb's 1949 work 'The Organization of Behaviour' [2]. In this work, Hebb introduces the concept that connections strengthen between neurons depending on the frequency with which they transmit information – commonly summarised by "Neurons that fire together wire together". This concept would later be key in developing artificial neural networks.

Another notable contribution to the world of neural networks is the groundbreaking 1958 paper by Frank Rosenblatt, 'The Perceptron: A Probabilistic Model for Information Storage and Organization in the Brain' [3]. In this paper, we first catch a glimpse of the neural network models as we know them today. Despite its later-found limitations, this milestone in the evolution of artificial neural networks represented a pivotal moment in the field. The perceptron model introduced a computational model capable of learning from data. This innovation would later inspire the development of backpropagation algorithms and ultimately affected the neural network and AI technology we use today. [4]

This project will begin by laying out the mathematical concepts that form the basis of neural networks. Through a step-by-step exploration, we will examine these concepts and how they work together to form functional networks. After establishing a solid understanding of the theory, we will apply the learned principles by developing neural networks for handwritten digit classification. This use case will serve as a practical example, helping to illustrate how the theoretical foundations of neural networks translate into working models. Alongside this, we will delve into some advanced topics to explore how neural networks can be further enhanced.

For a comprehensive overview of the topic, please reference [5, 6, 7].

Before we dive into the mathematics behind neural networks, we will first introduce a specific use case that will be referred to throughout the project to help contextualise the concepts being explained.

For this project, we will explore the concepts of neural networks through the lens of a digit classification neural network. The specific details of its implementation will be discussed in later sections. However, simply put, our digit classification neural network is a model designed to receive a 28-by-28 pixel grayscale image of a handwritten digit (from 0 to 9) as an input and output a prediction of which digit has been drawn.

While to most this may seem like a trivial task, it poses a significant challenge to a computer, which lacks the inherent ability to recognise intricacies with varying handwriting styles. As a result, for the computer, this becomes a complex task of computation.

2 Fundamentals of Neural Network Architecture

In this section, we will explore the mathematical foundations of neural networks, delving into the concepts and structures that form a functional network. By shifting the focus away from the traditional computer science perspective, we aim to develop a deeper understanding of the topic from a mathematical standpoint.

2.1 Elements of Neural Networks

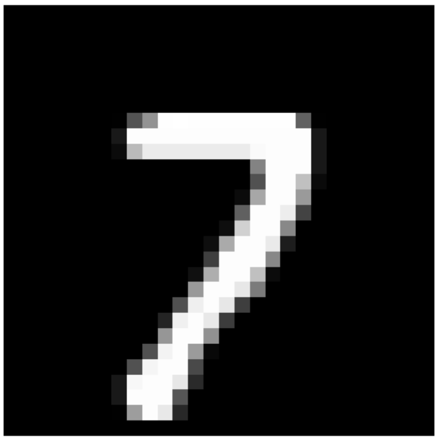
First and foremost, to understand the mathematics behind neural networks, it is vital to grasp the elements that make up a neural network and the roles these elements play in the larger scheme. Without this knowledge, the formulas introduced later to describe the exact workings of the network would be interpreted with little to no meaning.

To answer the question, "What is a neural network?" we first need to define what it means to be neural – specifically, what neurons are – and then explore how these neurons are assembled to create a network.

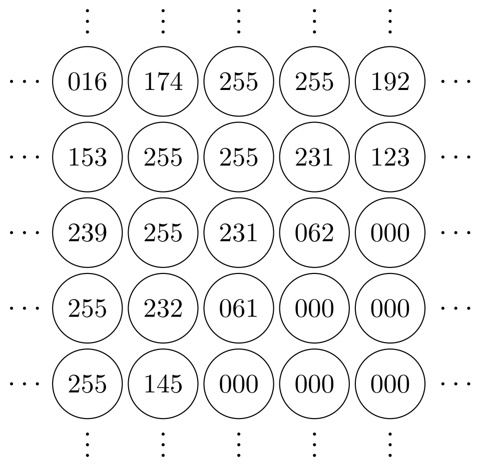
2.1.1 Neurons

In its simplest form, we can envision the neurons in a neural network as placeholders for values. These values correspond to how active each neuron is in the network, which we will refer to as the **activation** of the neuron. Later, we will explore different conventions used: in some cases, a neuron's activation is bounded between 0 and 1; in others it ranges from -1 to 1; while in other cases, activations can take on any positive real value. The reasoning behind these different approaches will become clear later, but for now, think of a neuron as a placeholder for a specific value.

In our digit classifier model, we start with 784 neurons, each corresponding to a single pixel in the 28 by 28 pixel image. The activations of these neurons range from 0 to 255, where 0 represents a completely black pixel and 255 represents a completely white pixel.



(a) Input image example.



(b) Section of neurons of the image.

Figure 1: Visualisation of the activations for a section of an input image.

2.1.2 Layers

Now that we have established what neurons are, how do they form a network? Layers are the first structural concept in neural networks. With this layered organisation, we introduce an order to the neurons by defining which are first in data transfer and which are last. The layered structure means that as data is transmitted from one layer to the next, the neurons in the following layer are a product of the previous layer, and so forth. Importantly, every neuron in one layer is connected to every neuron in the subsequent layer, creating a fully connected structure between layers.

In neural networks, we have three types of layers: **input layers**, which, as the name suggests, are the layers that initially take in the input data. In our case, this layer comprises 784 neurons, corresponding to the 784 pixels in the input image. Skipping to the end of the network, we have **output layers**, which make predictions or conclusions based on the processed data. For example, if a network is designed to output a Boolean response (yes or no, on or off, etc.), the output layer would consist of two neurons, one for each possible outcome. In our example, the output layer will have ten neurons to represent the ten digits 0 to 9.

Between these two layers are the **hidden layers**. While it is not crucial to understand exactly what these layers do at this stage, it is important to note that they sit between the input and output layers and follow the same rules, as mentioned above, for data transmission. The number and size of these hidden layers are not constrained by the application or purpose of the neural network; rather, they are a design choice made during network construction. While these choices are not completely arbitrary in practice, we can consider them as such at this stage.

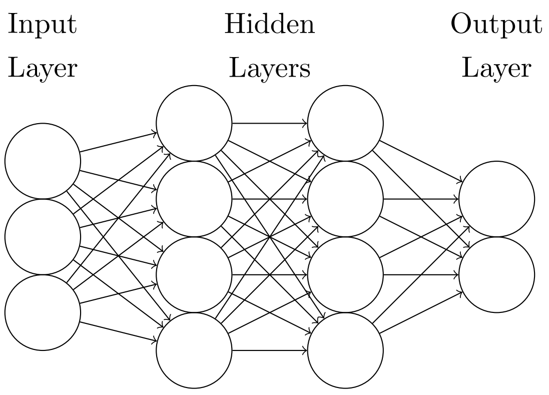


Figure 2: Simple example showing the layered structure of a neural network.

2.1.3 Weights

As mentioned before, activations of neurons are a "product" of the activations in the previous layer. While this is true, we need to introduce additional complexity between these stages; otherwise, the output layer would be an overly simple transformation of the preceding layers. To address this, we introduce the concept of weights.

Weights in a neural network quantify the strength of connections between individual neurons by acting as coefficients for the activation values. Since weights are adjustable variables, they allow the network to determine which neurons in the previous layer are most influential when computing the activation of neurons in the current layer.

At this stage, this concept may seem abstract, so while not the complete model, it is worth developing a simplified mathematical model to understand how a neurons activation is computed.

$$a_j^{(l)} = \sum_{i=1}^{n} w_{j,i}^{(l)} \cdot a_i^{(l-1)} \in \mathbb{R} \tag{1}$$

Where j is the index of the j^{th} neuron in the l^{th} layer, i is the index of the i^{th} neuron in the l^{th} layer and $w_{j,i}^{(l)} \in \mathbb{R}$ is the weight that connects these two neurons.

In words, for each neuron in layer l, we sum up all the activation values from layer $l-1$, each with their respective weight coefficient.

For example, in our case, each neuron in the second layer of the network would have 784 weights – one for each neuron in the input layer. If the first hidden layer has only 16 neurons, we already have 12,544 adjustable parameters to optimise for the network to function as intended, and that is just for the data transfer between the input layer and the first hidden layer.

2.1.4 Biases

Biases work alongside weights as another adjustable parameter used to compute the activation of neurons in subsequent layers. Unlike weights, which adjust the summation of activations from all previous neurons, biases are added directly into the sum (1). This is more clearly illustrated in the adjusted model below.

$$a_j^{(l)} = b_j^{(l)} + \sum_{i=1}^{n} w_{j,i}^{(l)} \cdot a_i^{(l-1)} \in \mathbb{R} \tag{2}$$

Where $b_j^{(l)} \in \mathbb{R}$ is the bias term used to calculate $a_j^{(l)}$.

The purpose of the bias is to allow the network to adjust the final activation value independently of the weighted sum of the previous activations. For example, if the summation results in a value x and the network has determined that while the ratios of the weights properly reflect the relative importance of the neurons, the activation should be shifted to a different value y, the bias b represents the difference between x and y. By adding or subtracting the bias term b, we can achieve the desired final activation.

Referring back to the example from the weights section, in the first hidden layer of the network, we will have 16 biases – one for each of the 16 summations. This brings the total number of adjustable parameters at this stage to 12,560.

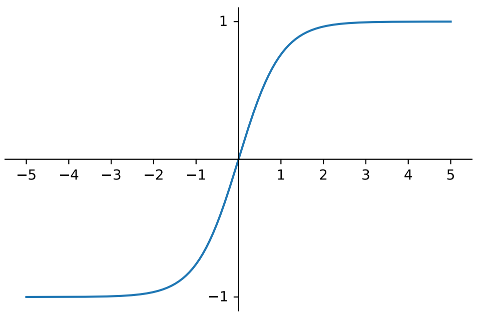
2.1.5 Activation Functions

To complete the model we have been developing, we need to incorporate activation functions. These functions vary widely, but their primary purpose is to introduce non-linearity into the model, making training more efficient [8]. In practice, the activation function is applied to process the mathematical model we have built so far.

Numerous activation functions have been developed throughout the history of neural networks; however, in this section, we will focus on three of the most widely used functions.

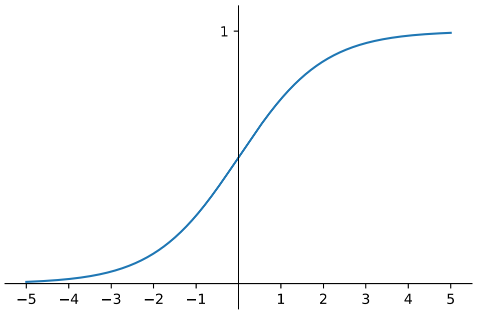
The first of these activation functions is the **tanh** function, a familiar function from trigonometry. It takes a real-valued input and maps it to the range $[-1, 1]$, making it useful for centring data (evenly distributing values around 0). The tanh function is defined as follows:

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$



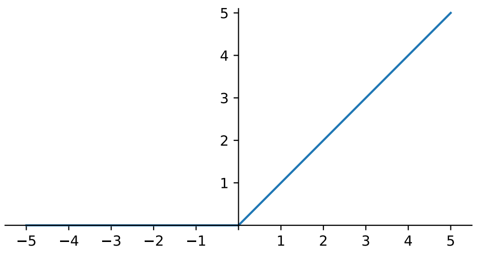
The second function is the **Sigmoid** function. Like the tanh function, it accepts real values as input but maps them to the range $[0, 1]$. This is particularly useful in cases where we need probability-like output. The Sigmoid function is defined as follows:

$$\sigma(x) = \frac{e^x}{1+e^x} = \frac{1}{1+e^{-x}}$$



Lastly, we have the **ReLU** (Rectified Linear Unit) function, which differs from the previous two. It uses the same domain of real numbers but maps them to the range $[0, \infty)$. ReLU is commonly used in modern networks due to its simplicity and efficiency, particularly in deep learning models. The ReLU function is defined as follows:

$$\mathrm{ReLU}(x) = \begin{cases} 0 & \text{if } x \leq 0 \\ x & \text{if } x > 0 \end{cases}$$



For any arbitrary activation function, $f(x)$, we can now complete our model (2) for computing activations as shown below:

$$a_j^{(l)} = f(Z_j^{(l)}) \in \mathbb{R} \quad \text{where,} \quad Z_j^{(l)} = b_j^{(l)} + \sum_{i=1}^{n} w_{j,i}^{(l)} \cdot a_i^{(l-1)} \in \mathbb{R} \tag{3}$$

2.2 Mathematics of Forward Propagation

In neural networks, **forward propagation** refers to the process of transmitting data from the input to the output layer, along with the relevant calculations performed at each step [9]. While we have developed a model for calculating the activation of an individual neuron, imagining a network calculating each neuron's activation individually, as demonstrated in Equation 3, would be quite cumbersome. To streamline this process, we will introduce linear algebra techniques, which allow us to simplify and scale the computations involved in forward propagation for larger networks.

First, we redefine the parameters we have introduced so far as elements in linear algebra, rather than as individual variables. To begin, we take all the activations from layer $l-1$ and arrange them into a vector as shown below:

$$a^{(\vec{l-1})} = \begin{bmatrix} a_1^{(l-1)} \\ a_2^{(l-1)} \\ \vdots \\ a_n^{(l-1)} \end{bmatrix}$$

Next, we construct a matrix containing all the weights associated with the connections between layer $l-1$ and layer l. This matrix is structured such that each row corresponds to the weights connecting to a specific neuron in layer l, while each column corresponds to the weights connecting from a particular neuron in layer $l-1$, as shown below:

$$W^{(l)} = \begin{bmatrix} w_{1,1}^{(l)} & w_{1,2}^{(l)} & \cdots & w_{1,n}^{(l)} \\ w_{2,1}^{(l)} & w_{2,2}^{(l)} & \cdots & w_{2,n}^{(l)} \\ \vdots & \vdots & \ddots & \vdots \\ w_{k,1}^{(l)} & w_{k,2}^{(l)} & \cdots & w_{k,n}^{(l)} \end{bmatrix}$$

We then assemble the associated biases into a vector:

$$b^{(\vec{l})} = \begin{bmatrix} b_1^{(l)} \\ b_2^{(l)} \\ \vdots \\ b_k^{(l)} \end{bmatrix}$$

It is important to clarify how we apply activation functions in this linear algebra context. For an arbitrary activation function $f(x)$, we define it as follows:

$$\text{For, } \vec{v} = \begin{bmatrix} v_1 \\ v_2 \\ v_3 \end{bmatrix} \text{ , } f(\vec{v}) = \begin{bmatrix} f(v_1) \\ f(v_2) \\ f(v_3) \end{bmatrix}$$

With all these elements defined, we can now construct a model to reformulate the model introduced earlier in Equation 3. However, the key difference here is that we are no longer calculating the activation of a single neuron in layer l, but rather the activations for the entire layer l at once.

$$a^{(\vec{l})} = f\left(\begin{bmatrix} w_{1,1}^{(l)} & w_{1,2}^{(l)} & \cdots & w_{1,n}^{(l)} \\ w_{2,1}^{(l)} & w_{2,2}^{(l)} & \cdots & w_{2,n}^{(l)} \\ \vdots & \vdots & \ddots & \vdots \\ w_{k,2}^{(l)} & w_{k,2}^{(l)} & \cdots & w_{k,n}^{(l)} \end{bmatrix} \cdot \begin{bmatrix} a_1^{(l-1)} \\ a_2^{(l-1)} \\ \vdots \\ a_n^{(l-1)} \end{bmatrix} + \begin{bmatrix} b_1^{(l)} \\ b_2^{(l)} \\ \vdots \\ b_k^{(l)} \end{bmatrix} \right) = f\left(W^{(l)} \cdot a^{(\vec{l-1})} + b^{(\vec{l})}\right) \tag{4}$$

2.2.1 Softmax Function

Now that we have a linear algebra model of forward propagation, it is worth briefly revisiting the ReLU activation function. While ReLU is popular due to its simplicity and effectiveness, it tends to produce extremely large activation values as we move deeper into the network. To address this issue, ReLU is often used in tandem with the softmax function, particularly in the output layer, to handle these large values and normalise the results [10].

The softmax function transforms a vector of real numbers into a vector of probabilities that sum to 1, making it especially useful for classification tasks where the goal is to assign a probability to each class. By converting activations into a probability distribution, the network becomes more computationally stable and efficient [11]. The softmax function is defined as follows:

$$\text{softmax}(\vec{x})_i = \frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_j}} \in [0,1]$$

3 Training Methods of Neural Networks

3.1 Concepts of Back Propagation

Now that we have a solid understanding of forward propagation in a neural network, we can turn to the next question: how do we optimise a model that involves hundreds of neurons, and thousands of parameters (weights and biases), to perform its desired task? The answer lies in a process known as back propagation.

Before diving into the mathematics of back propagation, it is helpful to imagine a scenario where we only understand forward propagation, as discussed in the previous chapter, but have no knowledge of back propagation. In this case, for a model with an input layer of 784 neurons, two hidden layers of 16 neurons each, and an output layer of 10 neurons, we would have to manually adjust 13,002 parameters through trial and error to achieve an efficient network. This example illustrates the immense value of back propagation in automating and optimising the learning process within a neural network.

3.1.1 Cost Functions

To train a network and optimise its parameters, we first need a method to evaluate how well the current set of parameters (weights and biases) is performing. This is where cost functions come into play, acting as a bridging point between forward propagation and back propagation. At its most basic level, the cost function compares the output layer of the network after forward propagation to the 'ideal' or 'optimal' output layer.

For example, if our network is given an image of the digit '4' and the output layer produces the following vector:

$$\vec{X} = \begin{bmatrix} 0.1 & 0.2 & 0.4 & 0.1 & 0.8 & 0.6 & 0.2 & 0 & 0.6 & 0.4 \end{bmatrix}^T$$

We can see that the network correctly predicts '4' as the most likely classification, signified by the largest element of $\vec{X}$ being in the fifth position. However, there is still some noise suggesting a small probability that the digit could be another classification, although less likely. The cost function evaluates this output by comparing it to the 'optimal' expected output layer, which looks like:

$$\vec{Y} = \begin{bmatrix} 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}^T$$

One method of calculating the cost of this training example is as follows:

$$\text{Cost}(\vec{X}) = (x_1 - y_1)^2 + (x_2 - y_2)^2 + \cdots + (x_{10} - y_{10})^2 = \sum_{i=1}^{10} (x_i - y_i)^2$$

Which for the vectors exampled above is:

$$\text{Cost}(\vec{X}) = (0.1 - 0)^2 + \cdots + (0.8 - 1)^2 + \cdots + (0.4 - 0)^2 = 1.18$$

In words, the cost is the sum of the squared differences between the activations in the output layer with their respective desired activation. As we can see, the closer the network's output is to the 'optimal' output, the smaller the cost will be. Vice versa, the further the networks output deviates from the optimal result, the higher the cost. Therefore, the goal of optimising the network is equivalent to minimising the cost function.

The above section is an example of the l^2 cost function, which is just one of many types of cost functions we can choose from. Other cost functions, such as the l^1 and the l^4 cost functions, are similar in that they both measure the difference between the predicted and actual values. However, the l^1 cost function calculates the absolute difference $|(x_i - y_i)|$, while the l^4 cost function raises the difference to the fourth power. For now, however, we will focus on the l^2 cost function as the example.

This is the cost of just one training example. In practice, however, the cost is computed as the average of the costs over all the training data.

Another useful way to think of the cost is as being dependent on the network's weights and biases, which are the only adjustable parameters. If we imagine the cost function as a high dimensional hyper-surface, our objective is to find a minima (local or global) on the hyper-surface, where the network performs optimally.

3.1.2 Impact of Parameter Adjustments

Before delving into how we fine-tune the parameters of our network to navigate toward a desired minima of the cost function, it is useful to first explore the implications of making these adjustments. By understanding how changes to a network's parameters influence its outputs, we can develop a clearer perspective on what should be considered when optimising a neural network.

To explore this, let us consider a simpler network. Suppose we have a classification network with only three neurons in the output layer and five neurons in the hidden layer directly before the output layer. Imagine that, after inputting some data, we expect the first output neuron to be the most active. Instead, our network produces a different result, as shown below:

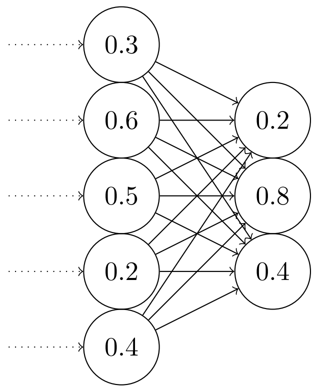


Figure 3: Example of the last hidden layer and the output layer of the neural network.

The question is, how can we adjust this network so the output aligns with our expectations? To address this, it is worth reconsidering Equation 3, more specifically its components: the weights, the biases and the activations. This gives us three primary options when changing the desired neuron's activation: We can increase the bias for the desired neuron, increase the weights for the desired neuron, or increase the activations of the previous layer. Let us look at the implications of each of these approaches.

Adjustment to the biases.

Since biases are independent terms in the activation equation, adjusting them is the simplest approach. By increasing the bias of the desired output neuron, we can directly increase its activation; similarly, we can decrease activations for other neurons by lowering their biases. However, this approach does have its limitations. If the biases favour one output neuron, the network will struggle to handle variations across different classifications. Therefore, while adjusting biases is part of optimisation, it cannot be solely relied upon to achieve an effective network.

Adjustment to the weights.

When adjusting weights, we must consider their role as coefficients for activations in the previous layer. Instead of increasing all the weights indiscriminately, we should prioritise increasing the weights coming from neurons with large activations and reduce those with small or negative activations. This approach, reminiscent of Hebb's principle – "Neurons that fire together wire together"[2] – ensures that the most active neurons in the previous layer make the most significant contributions to the desired neuron in the output layer. In short, the strongest adjustments to weights come from the most active neurons and connect to the neuron we aim to activate.

Adjustment to the previous activations.

While we cannot directly alter the activations in the previous layer, adjusting weights and biases in earlier layers allows us to indirectly influence them. This idea is what brings us to the essence of back propagation: we "propagate" adjustments backwards through the network, starting from the output layer, moving towards the input layer.

To illustrate, let us assume that the weights and biases between the final hidden layer and the output layer are already optimal. To optimise the output, we look at these weights and identify that, to achieve the optimal result, large weights should come from highly active neurons, while small or negative weights should come from relatively inactive neurons. Following this logic, we apply the previously discussed approaches to manipulate weights and biases between the penultimate and final hidden layers, thereby changing the activations in the final hidden layer as we desire.

It should be noted that, in practice, we do not assume that the weights and biases are "optimal" or fix them to specific values. However, this illustrates the main concept of back propagation.

Back propagation combines all three primary adjustment approaches: biases, weights, and activations. While the method is consistent regardless of the number of hidden layers, adjustments at different stages carry different implications. For instance, adjusting a weight between the final hidden layer and the output layer affects only a single neuron in the output layer, while changing a weight between the input layer and the first hidden layer in a deep network influences every neuron in every subsequent layer. We will dive into the mathematical representation of these processes later, but for now, it is important to recognise that while each adjustment method is straightforward in isolation, blending them together in a complete back propagation algorithm becomes difficult.

3.1.3 Gradient Descent

Having explored the impact of parameter adjustment within a neural network, we can now focus on how we can systematically navigate to a minima of the cost function we want to find. To do this, we turn to gradient descent, a fundamental optimisation method used to minimise the cost functions.

Let us start by considering a simplified version of the cost function – a function with only one parameter to optimise. While this does not directly reflect the cost functions we deal with in practical neural networks, it is a helpful way to visualise how optimisation works in these models.

Imagine we have a cost function represented by the following plot, with our initial parameter at x_0.

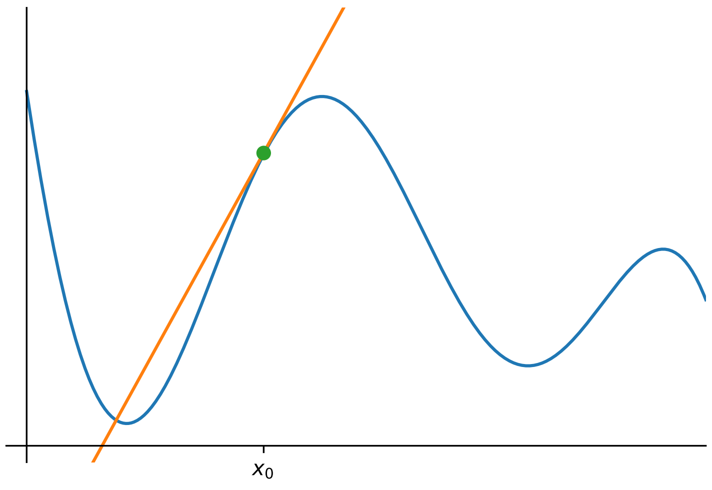


Figure 4: A visualisation of the cost function with a marker point for x_0 along with the gradient at that point.

Although we may not be able to directly calculate the minima, we can approach it using numerical methods. If we find the slope of the function at x_0 (by differentiating it), we can adjust x_0 according to the slope:

- If the slope is negative, we increase x_0.
- If the slope is positive, we decrease x_0.

By repeating this process until the slope reaches zero, we can approach a point in the set of all minima (local and global). To prevent overshooting the minima, the adjustments made to x_0 are proportional to the slopes – this ensures smaller steps are made when the slope is small, and larger steps are made when the slope is large.

This same approach extends similarly to higher-dimensional cost functions. For instance, if our cost function had two parameters, it could be visualised as a surface in $\mathbb{R}^3$. In this case, instead of differentiating at a single point x_0, we calculate the gradient (∇C) at the point (x_0, y_0). The gradient tells us the direction of the steepest ascent on the surface, so we adjust our parameters in the opposite direction, or ‘downhill’ to move to the minima. This two-dimensional case scales directly to higher dimensions, which is essential for practical neural networks, where the cost function might involve thousands of parameters.

To summarise, gradient descent works by:

1. Computing the gradient of the cost function.
2. Taking a small step in the negative direction of the gradient.
3. Repeating the process until a minima is found.

Though this may seem abstract - finding the ‘steepest’ direction in a space with thousands of dimensions - we can simplify our understanding by focusing on the mathematical representation. Consider a weight vector $\vec{w}$, and suppose we have computed the negative gradient of the cost function with respect to $\vec{w}$, resulting in the vector below:

$$\vec{w} = \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_n \end{bmatrix} \quad , \; -\nabla C(\vec{w}) = \begin{bmatrix} 0.13 \\ -2.67 \\ \vdots \\ 0.72 \end{bmatrix}$$

Looking at these vectors, the signs of the elements in the computed vector tell us the direction in which to adjust the weights (negative for a decrease and positive for an increase). The magnitudes of the elements indicate how much to adjust each weight - larger magnitudes signify weights that need bigger adjustments, while smaller ones need only slight tweaks. For example, we see that w_1 needs only a small positive adjustment, while w_2 requires a large negative adjustment.

By interpreting the negative gradient vector in this way, we can see it as a guide for how to adjust the weights to optimise the network. While the concept of “finding the ‘steepest’ direction in a space with thousands of dimensions” is challenging to visualise, thinking of the negative gradient as producing a vector of proportional adjustments makes it far more intuitive. Nonetheless, it is still important to keep the original idea of finding the minima of a function in mind, as this nicely links back to the overall goal of minimising the cost.

3.1.4 Learning Rates

In the training of neural networks, we know that the backpropagation algorithm - an application of the gradient descent algorithm - is iterated to optimise the network. However, a common issue in this process is a phenomenon called overfitting. Overfitting occurs when the network becomes too specialised to the training data, achieving high accuracy on this data but performing poorly on unseen datasets [12].

To mitigate overfitting, one of the first techniques we can employ is the introduction of learning rates in the process of backpropagation. During each iteration of backpropagation, the cost function is computed as the average error over all the training examples. From this, we calculate the negative gradient of the cost to determine the proportional adjustments needed for the network's parameters. These adjustments are then applied, preparing the network for the next iteration.

Without learning rates, we can often observe erratic behaviour in the parameter updates, with the parameters 'jumping' back and forth rather than converging steadily. Learning rates address this issue by acting as a coefficient - typically between 0 and 1 - applied to the adjustment vector (the negative gradient of the cost function). This scaling reduces the magnitude of the updates while preserving their direction and proportionality.

By dampening the updates, learning rates ensure that parameter adjustments become smaller and more stable over successive iterations. As the parameters approach an 'optimal' value, the adjustment vector's magnitude naturally decreases, leading to smoother convergence. Consequently, the network becomes more accurate as its task and less likely to overfit. The learning rate modifies the backpropagation update rule as shown in the following equation:

$$\vec{p_{i+1}} = \vec{p_i} - \alpha \nabla C(\vec{p_i}) \tag{5}$$

Where, $\vec{p_i}$ represents the current parameter vector, $\vec{p_{i+1}}$ represents the updated parameter vector of the next iteration and α is the learning rate.

There are numerous strategies for selecting and implementing learning rates, including:

- **Fixed Learning Rate**, where the learning rate remains constant throughout the entire training process.
- **Time-Based Decay**, where the learning rate decreases over time as training progresses.
- **Adaptive Learning Rate**, where the learning rate dynamically adjusts, increasing or decreasing based on whether parameters frequently converge around specific regions.

These are just a few examples but there are many other learning rate strategies commonly used in neural networks [13].

3.2 Mathematics of Back Propagation

Before we dive fully into the mathematics of backpropagation, let us consider a simplified network where we have one neuron per layer. This will help us build our understanding step-by-step.

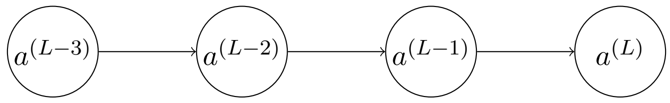


Figure 5: Simple 4 layer network with 1 neuron per layer.

In this network, we denote the activation of the output layer as $a^{(L)}$, with L indicating the layer the neuron is in, such that $a^{(L-1)}$ is the activation of the neuron in the previous layer, and so forth. Let us also define the desired output as y. The cost of this network, denoted as C_0, would then be calculated as follows:

$$C_0 = (a^{(L)} - y)^2$$

We also know that $a^{(L)}$ is calculated using an activation function, f, as follows:

$$a^{(L)} = f(z^{(L)}) \text{ , where } z^{(L)} = w^{(L)}a^{(L-1)} + b(L)$$

We can visualise the calculation of C_0 using the following function-chain diagram:

$$\begin{tikzcd}
 & \boxed{w^{(L)}} \arrow[dr] & & & \\
\cdots \arrow[r, dotted, dash] & \boxed{a^{(L-1)}} \arrow[r] & \boxed{z^{(L)}} \arrow[r] & \boxed{a^{(L)}} \arrow[r] & \boxed{C_0} \\
 & \boxed{b^{(L)}} \arrow[ur] & & \boxed{y} \arrow[ur] &
\end{tikzcd}$$

Figure 6: Chain diagram for the calculation of C_0.

This setup can be interpreted as follows: the weight parameter $w^{(L)}$, the activation $a^{(L-1)}$, and the bias $b^{(L)}$ together allow us to compute $z^{(L)}$. This $z^{(L)}$, combined with the activation function f, gives us $a^{(L)}$, which, along with the expected output y, allows us to calculate C_0. The dotted line going from $a^{(L-1)}$ to earlier layers indicates that this pattern continues; though for simplicity, we will focus only on this section.

Calculating the Effect of Small Changes in $w^{(L)}$ on C_0.

Let us examine how a small change in $w^{(L)}$ affects C_0, which can be expressed as the partial derivative $\frac{\partial C_0}{\partial w^{(L)}}$. Looking at our chain diagram, we see that this derivative is actually the result of several smaller effects: a change in $w^{(L)}$ causes a change in $z^{(L)}$, which changes $a^{(L)}$, ultimately affecting C_0. Applying the chain rule, we can decompose $\frac{\partial C_0}{\partial w^{(L)}}$ as:

$$\frac{\partial C_0}{\partial w^{(L)}} = \frac{\partial C_0}{\partial a^{(L)}} \cdot \frac{\partial a^{(L)}}{\partial z^{(L)}} \cdot \frac{\partial z^{(L)}}{\partial w^{(L)}}$$

Now, let us calculate each term in this chain.

- **Calculating** $\frac{\partial C_0}{\partial a^{(L)}}$: Since $C_0 = (a^{(L)} - y)^2$, we can differentiate with respect to $a^{(L)}$.

$$\frac{\partial C_0}{\partial a^{(L)}} = 2(a^{(L)} - y)$$

- **Calculating** $\frac{\partial a^{(L)}}{\partial z^{(L)}}$: Given $a^{(L)} = f(z^{(L)})$, this derivative is just $f'(z^{(L)})$, where f' is the derivative of the activation function.

$$\frac{\partial a^{(L)}}{\partial z^{(L)}} = f'(z^{(L)})$$

- **Calculating** $\frac{\partial z^{(L)}}{\partial w^{(L)}}$: Since $z^{(L)} = w^{(L)}a^{(L-1)} + b^{(L)}$, the derivative with respect to $w^{(L)}$ is $a^{(L-1)}$.

$$\frac{\partial z^{(L)}}{\partial w^{(L)}} = a^{(L-1)}$$

Combining these terms, we get:

$$\frac{\partial C_0}{\partial w^{(L)}} = a^{(L-1)} \cdot f'(z^{(L)}) \cdot 2(a^{(L)} - y)$$

Let us pause and interpret this result. The term $a^{(L-1)}$ in the derivative shows us that the most significant weight adjustments will come from neurons with high activation, which once again aligns with Hebb's principle that "Neurons that fire together wire together"[2].

Calculating the Effect of Small Changes in $b^{(L)}$ on C_0.

We can use a similar approach to understand how the cost changes with respect to $b^{(L)}$, that is, $\frac{\partial C_0}{\partial b^{(L)}}$. The chain rule decomposition is almost identical to what we saw for $w^{(L)}$, but instead the $\frac{\partial z^{(L)}}{\partial w^{(L)}}$ term is instead replaced with $\frac{\partial z^{(L)}}{\partial b^{(L)}}$. Since $z^{(L)} = w^{(L)}a^{(L-1)} + b^{(L)}$, the derivative with respect to $b^{(L)}$ is simply 1. Hence:

$$\frac{\partial C_0}{\partial b^{(L)}} = f'(z^{(L)}) \cdot 2(a^{(L)} - y)$$

Calculating the Effect of Small Changes in $a^{(L-1)}$ on C_0.

As we have previously discussed, the essence of backpropagation is that we desire to directly adjust the neuron's activation. However, since we cannot directly modify the activations of the neurons, we instead propagate the adjustments backward through the network, altering the weights and biases in the previous layers in such a way that we achieve the desired activations.

With this principle in mind, let us look at the partial derivative $\frac{\partial C_0}{\partial a^{(L-1)}}$, which we can compute using the chain rule. As with earlier calculations, we follow the same reasoning. Specifically, the change in cost C_0 is influenced by changes in $a^{(L-1)}$, and we want to quantify how sensitive the cost function is to changes in the activations of the previous layer, $a^{(L-1)}$.

Using the chain rule, we can express the partial derivative as:

$$\frac{\partial C_0}{\partial a^{(L-1)}} = \frac{\partial C_0}{\partial a^{(L)}} \cdot \frac{\partial a^{(L)}}{\partial z^{(L)}} \cdot \frac{\partial z^{(L)}}{\partial a^{(L-1)}}$$

Since $z^{(L)} = w^{(L)} a^{(L-1)} + b^{(L)}$, the derivative term, $\frac{\partial z^{(L)}}{\partial a^{(L-1)}}$ is $w^{(L)}$. As we already know the other terms in the equation above, we can conclude:

$$\frac{\partial C_0}{\partial a^{(L-1)}} = w^{(L)} \cdot f'(z^{(L)}) \cdot 2(a^{(L)} - y)$$

This equation tells us how sensitive the cost function C_0 is to small changes in the activation of the previous layer, $a^{(L-1)}$. We can use this result to propagate the error backward through the network, adjusting the weights and biases in the previous layer to minimise the cost function. With this calculation, we can now move on to compute the partial derivatives of the cost function with respect to the weights and biases in the layer prior to $L-1$, namely $L-2$, continuing the backpropagation process.

So far, we have calculated these partial derivatives for a single training example. In practice, we compute the partial derivative of the overall cost by averaging over many training examples, as shown below:

$$\frac{\partial C}{\partial w^{(l)}} = \frac{1}{n} \sum_{i=1}^{n} \frac{\partial C_i}{\partial w^{(l)}}$$

$$\frac{\partial C}{\partial b^{(l)}} = \frac{1}{n} \sum_{i=1}^{n} \frac{\partial C_i}{\partial b^{(l)}}$$

where n is the number of training examples and l is the layer you are calculating for. These terms represent the individual contributions of each training example to the gradient of the cost function. More specifically, the partial derivatives we have calculated, on the left-hand side of the above equations, are the elements that make up the gradient of the cost function, ∇C.

By definition, the gradient of the cost function is a vector of the partial derivatives of C with respect to all the different weights and biases in the network. This gradient vector, ∇C, contains the information needed for gradient descent to update the weights and biases in the network. For the four-layer network we specified in this section, ∇C looks like:

$$\nabla C = \begin{bmatrix} \frac{\partial C}{\partial w^{(L-2)}} & \frac{\partial C}{\partial b^{(L-2)}} & \frac{\partial C}{\partial w^{(L-1)}} & \frac{\partial C}{\partial b^{(L-1)}} & \frac{\partial C}{\partial w^{(L)}} & \frac{\partial C}{\partial b^{(L)}} \end{bmatrix}^T$$

This vector, comprised of the partial derivatives we have calculated allows us to efficiently perform updates to the parameters (weights and biases) in the direction that locally minimises the overall cost, as previously shown in section 3.1.3.

While its helpful to see how our partial derivatives work in the gradient ∇C and play a role in the gradient descent in our single-neuron-per-layer example, it is worth noting that, in practical networks, we mainly have multiple neurons per layer. Extending this to a more general case, however, does not introduce a lot of additional complexity; rather, it builds on what we have already established.

Lets consider a network with p neurons in the output layer and q neurons in the hidden layer just before it:

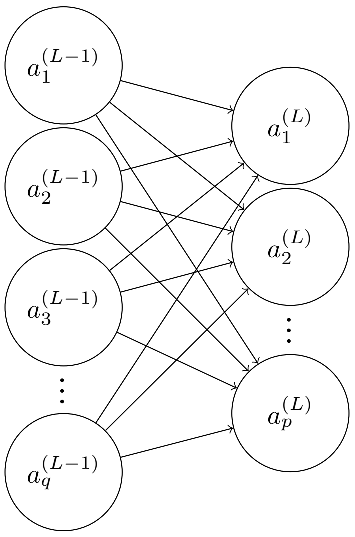


Figure 7: Example of the last hidden layer and the output layer for a general network.

To explain the workings of this network mathematically, we need to introduce subscript notation for clarity. The activations will now have a subscript to indicate the specific neuron; for example, $a_k^{(L-1)}$ refers to the activation of the k^{th} neuron in the $(L-1)^{th}$ layer.

Similarly, we will introduce subscripts to the weights and biases. For biases, the subscript identifies the neuron it corresponds to; for instance, when calculating $a_j^{(L)}$, we use the bias, $b_j^{(L)}$. For weights, we use a two-index notation to denote connections between layers. Specifically, the weight connecting $a_k^{(L-1)}$ to $a_j^{(L)}$ is denoted as $w_{j,k}^{(L)}$, consistent with the weight matrix notation introduced in section 2.2.

In this network, the cost function is generalised as:

$$C = \sum_{i=1}^{p} (a_i^{(L)} - y_i)^2 \text{ where } y_i \text{ are elements of } \vec{Y}\text{, the expected output vector.}$$

As before, the activations $a^{(L)}$ are computed using an arbitrary activation function f, but the term $z^{(L)}$ now represents a sum over all q activations in the $(L-1)^{th}$ layer:

$$z_j^{(L)} = w_{j,1}^{(L)} a_1^{(L-1)} + w_{j,2}^{(L)} a_2^{(L-1)} + \cdots + w_{j,q}^{(L)} a_q^{(L-1)} + b_j^{(L)}$$

Although this setup appears more complex, we will see that the structure of our calculations remains consistent with the simpler model. Let us begin by calculating the sensitivity of the cost function with respect to changes in a weight between layer $(L-1)$ and L:

$$\frac{\partial C}{\partial w_{j,k}^{(L)}} = \frac{\partial C}{\partial a_j^{(L)}} \cdot \frac{\partial a_j^{(L)}}{\partial z_j^{(L)}} \cdot \frac{\partial z_j^{(L)}}{\partial w_{j,k}^{(L)}}$$

When calculating this partial derivative, we obtain the following equation:

$$\frac{\partial C}{\partial w_{j,k}^{(L)}} = a_k^{(L-1)} \cdot f'(z_j^{(L)}) \cdot 2(a_j^{(L)} - y)$$

As we can see, this form is structurally identical to what we obtained in the single-neuron-per-layer example.

Similarly, when we calculate the sensitivity of the cost function with respect to the bias $b_j^{(L)}$, the result mirrors the form seen in the simpler example:

$$\frac{\partial C}{\partial b_j^{(L)}} = f'(z_j^{(L)}) \cdot 2(a_j^{(L)} - y)$$

Where we begin to see differences is when calculating the sensitivity of the cost function with respect to the activations in the layer $L-1$. While this element does not appear in the ∇C vector, it is important to calculate it, as it enables us to propagate backwards through the network to calculate all the other sensitivities.

When calculating the sensitivity of C to the activation $a_k^{(L-1)}$, we must account for its effect on all activations in layer L. Therefore, we sum the influences of $a_k^{(L-1)}$ across all neurons in layer L. The resulting partial derivative can be expressed as follows:

$$\frac{\partial C}{\partial a_k^{(L-1)}} = \sum_{i=1}^{p} \frac{\partial C}{\partial a_i^{(L)}} \cdot \frac{\partial a_i^{(L)}}{\partial z_i^{(L)}} \cdot \frac{\partial z_i^{(L)}}{\partial a_k^{(L-1)}}$$

With these results, we are now equipped to assemble the gradient vector, ∇C, containing all partial derivatives of C with respect to each weight and bias in the network. This vector allows us to implement gradient descent across the entire network efficiently. A representation of ∇C for our network would look as follows:

$$\nabla C = \begin{bmatrix} \vdots \\ \frac{\partial C}{\partial w_{j,k}^{(l)}} \\ \frac{\partial C}{\partial b_j^{(l)}} \\ \vdots \end{bmatrix}$$

For these arbitrary elements of the ∇C vector, we can express them as the following iterative calculations:

- For an arbitrary weight, we express the partial derivative as:

$$\frac{\partial C}{\partial w_{j,k}^{(l)}} = a_k^{(l-1)} \cdot f'(z_j^{(l)}) \cdot \frac{\partial C}{\partial a_j^{(l)}}$$

  where,

$$\frac{\partial C}{\partial a_j^{(l)}} = \sum_{j=1}^{n_{l+1}} w_{j,k}^{(l+1)} \cdot f'(z_j^{(l+1)}) \cdot \frac{\partial C}{\partial a_j^{(l+1)}}$$

  or for the final layer L,

$$\frac{\partial C}{\partial w_{j,k}^{(L)}} = a_k^{(L-1)} \cdot f'(z_j^{(L)}) \cdot 2(a_j^{(L)} - y)$$

- For an arbitrary bias, we express the partial derivative as:

$$\frac{\partial C}{\partial b_j^{(l)}} = f'(z_j^{(l)}) \cdot \frac{\partial C}{\partial a_j^{(l)}}$$

  where again,

$$\frac{\partial C}{\partial a_j^{(l)}} = \sum_{j=1}^{n_{l+1}} w_{j,k}^{(l+1)} \cdot f'(z_j^{(l+1)}) \cdot \frac{\partial C}{\partial a_j^{(l+1)}}$$

  or for the final layer L,

$$\frac{\partial C}{\partial b_j^{(L)}} = f'(z_j^{(L)}) \cdot 2(a_j^{(L)} - y)$$

With this detailed structure of ∇C, we now fully understand the backpropagation process at its mathematical core. By iteratively updating weights and biases in the direction of $-\nabla C$, we move the network's parameters toward a configuration that minimises the overall cost, as explored in section 3.1.3.

4 Creating a Neural Network from a Mathematical Perspective

Now that we have covered the core mathematical principles behind neural networks, let us explore how these principles are applied in a coded model. Typically, when creating neural networks in Python, popular libraries like TensorFlow and PyTorch are commonly used. These packages can be extremely helpful, simplifying the process by removing the need for much of the mathematics, allowing you to focus more on the design choices rather than on the underlying computations.

While these libraries are powerful, it is essential to remember that the mathematics that we have covered in sections 2 and 3 are the foundations of neural networks. Building a neural network from scratch - without relying on these packages - can deepen our understanding of how each mathematical operation contributes to the network's functionality.

In this chapter, we will take a 'mathematician's approach' to coding, avoiding conventional neural network libraries. Instead, we will create a neural network for digit classification using only basic Python mathematics libraries, specifically NumPy. For our input data, we will use the MNIST dataset [14], which is freely available online and provides labelled images along with arrays containing the pixel values for each image. The MNIST dataset we will use can be found in CSV format in the below GitHub repository.

Throughout this chapter, we will highlight specific sections of the code and link them back to relevant mathematical concepts from the previous chapters. For access to the complete code, visit:

https://github.com/JosephDavies427/Neural-Networks-Project

4.1 Coding Explanation

In chapter 2, we discussed the essential elements of a neural network's structure. Now in Python, we can streamline the process of defining these elements. Let us start by recalling equation 4 from section 2.2:

$$\vec{a^{(l)}} = f\left(\begin{bmatrix} w_{1,1}^{(l)} & w_{1,2}^{(l)} & \cdots & w_{1,n}^{(l)} \\ w_{2,1}^{(l)} & w_{2,2}^{(l)} & \cdots & w_{2,n}^{(l)} \\ \vdots & \vdots & \ddots & \vdots \\ w_{k,2}^{(l)} & w_{k,2}^{(l)} & \cdots & w_{k,n}^{(l)} \end{bmatrix} \cdot \begin{bmatrix} a_1^{(l-1)} \\ a_2^{(l-1)} \\ \vdots \\ a_n^{(l-1)} \end{bmatrix} + \begin{bmatrix} b_1^{(l)} \\ b_2^{(l)} \\ \vdots \\ b_k^{(l)} \end{bmatrix}\right) = f\left(W^{(l)} \cdot \vec{a^{(l-1)}} + \vec{b^{(l)}}\right)$$

In our theoretical exploration, we established that the activations cannot be directly adjusted; they are initially provided by our input data (in this case, the MNIST dataset) and subsequently derived through calculations across network layers. Thus, our primary task is to define the weight matrices, bias vectors, and activation functions that compute these activations across the network. We will begin with the first two components: weights and biases.

4.1.1 Weights and Biases

To initialise weights and biases, we first need to make design choices on the network's architecture: the number of hidden layers and the number of neurons per layer. For this example, we will use two hidden layers, each with 16 neurons. The input layer requires 784 neurons to match the dimensions of our input data, and the output layer will have 10 neurons, corresponding to the 10 possible digit classifications. For each layer, the number of parameters in a bias vector equals the number of neurons in that layer. The code snippet below shows how we initialise these structures:

```
def init_params():
    W1 = np.random.rand(16, 784) - 0.5
    b1 = np.random.rand(16, 1) - 0.5
    W2 = np.random.rand(16, 16) - 0.5
    b2 = np.random.rand(16, 1) - 0.5
    W3 = np.random.rand(10, 16) - 0.5
    b3 = np.random.rand(10, 1) - 0.5
    return W1, b1, W2, b2, W3, b3
```

When creating weight matrices and bias vectors, we assign random initial values between -0.5 and 0.5. This range prevents any starting bias and avoids extreme initial values, which helps the network train more consistently by keeping the learning process stable.

Making a comparison back to the general formulas we saw in chapter 2, the `init_params` function generates three weight matrices in the following form:

$$W^{(l)} = \begin{bmatrix} w_{1,1}^{(l)} & w_{1,2}^{(l)} & \cdots & w_{1,n}^{(l)} \\ w_{2,1}^{(l)} & w_{2,2}^{(l)} & \cdots & w_{2,n}^{(l)} \\ \vdots & \vdots & \ddots & \vdots \\ w_{k,1}^{(l)} & w_{k,2}^{(l)} & \cdots & w_{k,n}^{(l)} \end{bmatrix}$$

where:

- For `W1`: $n = 784, k = 16$
- For `W2`: $n = 16, k = 16$
- For `W3`: $n = 16, k = 10$

Similarly, `init_params` generates three bias vectors of the form:

$$\vec{b^{(l)}} = \begin{bmatrix} b_1^{(l)} \\ b_2^{(l)} \\ \vdots \\ b_k^{(l)} \end{bmatrix}$$

where:

- For `b1`: $k = 16$
- For `b2`: $k = 16$
- For `b3`: $k = 10$

This initialisation step lays the structure of the network implicitly through the dimensions of the weight matrices. This will be the key structural foundation of the coded model.

4.1.2 Activation Functions

To complete the forward propagation, we need to implement the activation functions for our neural network. In our code, we will create multiple neural networks using various activation functions from section 2.1.5. However, the general structure will remain consistent: we will apply specific activation functions between most layers, but we will always use the softmax function between the second hidden layer and the output layer to improve classification performance. The code snippets implementing these activation functions are shown below:

- tanh code:

```
def tanh(Z):
    return np.tanh(Z)
```

- Sigmoid code:

```
def sigmoid(Z):
    return np.exp(Z) / (1 + np.exp(Z))
```

- ReLU code:

```
def ReLU(Z):
    return np.maximum(Z, 0)
```

- Softmax code:

```
def softmax(Z):
    A = np.exp(Z) / sum(np.exp(Z))
    return A
```

As we can see, these code snippets are straightforward applications of the mathematical functions explored in chapter 2.

4.1.3 Forward Propagation

Now that we have developed the individual components, let us bring them together to create a complete function for forward propagation. As mentioned earlier, we will create multiple neural networks with different activation functions to examine their impact on the networks performance. However, the application process remains consistent across activation functions, so for now, let us examine the code snippet using the tanh function as an example:

```
def forward_prop_tanh(W1, b1, W2, b2, W3, b3, X):
    Z1 = W1.dot(X) + b1
    A1 = tanh(Z1)
    Z2 = W2.dot(A1) + b2
    A2 = tanh(Z2)
    Z3 = W3.dot(A2) + b3
    A3 = softmax(Z3)
    return Z1, A1, Z2, A2, Z3, A3
```

In this code, we see the application of our forward propagation equation (Equation 4) across all layers in our network. Each layer computes its activations using the weight matrices, bias vectors, and the chosen activation function (tanh). In this code, `Z1`, `Z2`, and `Z3` correspond to $z^{(l)}$ we saw in the function chain figure (Figure 6), while `A1`, `A2`, and `A3` represent the $a^{(l)}$ activations we saw in the same figure.

4.1.4 Back Propagation

Now that the forward propagation process is defined, the next essential step for the network's code is backpropagation. As mentioned earlier in Section 3.1.1, we first need to implement a cost function to bridge the gap between forward propagation and backpropagation.

The following function, `cost_exp`, creates a one-hot encoded array for the 'optimal' outputs, as defined earlier. The array is arranged in a one-hot style with a single 1 entry per column in the row corresponding to the target output.

```
def cost_exp(Y):
    cost_exp_Y = np.zeros((Y.size, Y.max() + 1))
    cost_exp_Y[np.arange(Y.size), Y] = 1
    cost_exp_Y = cost_exp_Y.T
    return cost_exp_Y
```

With a method to evaluate the network's output established, the next step is to compute the derivatives of the activation functions defined earlier, which are essential to construct a backpropagation algorithm. These derivatives are implemented as follows:

- tanh derivative code:

```
def tanh_deriv(Z):
    return 1 - (np.tanh(Z))**2
```

- Sigmoid derivative code:

```
def sigmoid_deriv(Z):
    return np.exp(Z) / (1 + np.exp(Z))**2
```

- ReLU derivative code:

```
def ReLU_deriv(Z):
    return Z > 0
```

Now that we have defined these components, we can implement a functioning backpropagation algorithm. By combining the parts outlined above, we arive at the following function, `backward_prop_tanh`, which implements backpropagation for a network using the `tanh` activation function:

```
def backward_prop_tanh(Z1, A1, Z2, A2, Z3, A3, W1, W2, W3, X, Y, m):
    cost_exp_Y = cost_exp(Y)

    dZ3 = A3 - cost_exp_Y
    dW3 = 1 / m * dZ3.dot(A2.T)
    dB3 = 1 / m * np.sum(dZ3)

    dZ2 = W3.T.dot(dZ3) * tanh_deriv(Z2)
    dW2 = 1 / m * dZ2.dot(A1.T)
    dB2 = 1 / m * np.sum(dZ2)

    dZ1 = W2.T.dot(dZ2) * tanh_deriv(Z1)
    dW1 = 1 / m * dZ1.dot(X.T)
    dB1 = 1 / m * np.sum(dZ1)
    return dW1, dB1, dW2, dB2, dW3, dB3
```

This implementation is a straightforward application of the mathematical framework developed throughout this project. However, one notable feature is worth highlighting. Looking at the line `dZ3 = A3 - cost_exp_Y`, it may initially appear that this is an incorrect application of the cost function, as we might expect to see an absolute value or squared difference. The reason this approach works lies in the earlier implementation of the softmax function between the last hidden layer and the output layer. By using softmax, we can take advantage of a concept called Cross-Entropy Loss Functions. For a complete overview, refer to [15]. A brief explanation is as follows:

- The softmax function outputs probabilities, which are positive and normalised.
- Cross-entropy measures the difference between two probability distributions:
  1. The true labels (one-hot encoded vector, Y).
  2. The predicted probabilities (output from the softmax layer, A).

  For a single training example, the cross-entropy loss is defined as:

  $$L = -\sum_{i=1}^{k} Y_i \log(A_i)$$

  Since Y is one-hot encoded (a single 1 and the rest are zeros), the equation simplifies to:

  $$L = -\log(A_{\text{correct}})$$

- From this, we observe the following:
  1. If A_{correct} is close to 1, L approaches 0 since $\log(1) = 0$.
  2. If A_{correct} is close to 0, L becomes very large as $\lim_{x\to 0}\log(x) = -\infty$.
- When softmax and cross-entropy are combined, the gradient of the loss simplifies significantly. The gradient is given by:

  $$\frac{\partial L}{\partial Z_i} = A_i - Y_i$$

  Here, Z_i represents the normalised output of the last hidden layer before the softmax function is applied.

4.1.5 Learning Rates

With a backpropagation algorithm defined, we now need a method to update the network's parameters after each iteration of forward and backward propagation. In this network, we use a fixed learning rate that remains constant throughout the training process. The parameter update function is shown below, where `alpha` represents the learning rate:

```
def update_params(W1, B1, W2, B2, W3, B3, dW1, dB1, dW2, dB2, dW3, dB3, alpha):
    W1 = W1 - alpha * dW1
    B1 = B1 - alpha * dB1
    W2 = W2 - alpha * dW2
    B2 = B2 - alpha * dB2
    W3 = W3 - alpha * dW3
    B3 = B3 - alpha * dB3
    return W1, B1, W2, B2, W3, B3
```

4.1.6 Gradient Descent

Finally, with all the components defined, we can combine them into an automated learning algorithm for the network. The `grad_dec_tanh` function below demonstrates this process:

```
def grad_dec_tanh(X, Y, alpha, iterations, m):
    W1, B1, W2, B2, W3, B3 = init_params()
    accuracies = []
    for i in range(iterations):
        Z1, A1, Z2, A2, Z3, A3 = forward_prop_tanh(W1, B1, W2, B2, W3, B3, X)
        dW1, dB1, dW2, dB2, dW3, dB3 = backward_prop_tanh(Z1, A1, Z2, A2, Z3, A3,
                                                           W1, W2, W3, X, Y, m)
        W1, B1, W2, B2, W3, B3 = update_params(W1, B1, W2, B2, W3, B3,
                                               dW1, dB1, dW2, dB2, dW3, dB3, alpha)
        if i % 10 == 0:
            predictions = get_predictions(A3)
            accuracy = get_accuracy(predictions, Y)
            accuracies.append(accuracy)
    return W1, B1, W2, B2, W3, B3, accuracies
```

This function includes additional lines for performance monitoring, but its key steps are:

1. The parameters of the network are created using the `init_params` function, which defines the structure of the network, including the weights and biases for each layer.
2. Next, the following process is iterated for a given number of iterations:
   - The network processes the input data through the layers using the `forward_prop_tanh` function, which computes the activations at each layer.
   - The gradients of the cost function with respect to the network parameters are computed using the `backward_prop_tanh` function.
   - The network parameters (weights and biases) are updated using the gradients and the learning rate (`alpha`) in the `update_params` function.
3. At the end of the process, the optimised parameters of the network (`W1`, `B1`, `W2`, `B2`, `W3`, `B3`) are returned.

This example covers the process when the activation function is the tanh function, though the same logic applies when the activation function is set to sigmoid or ReLU.

4.2 Network Analysis

The code presented above shows a selected portion of the complete code used to create the working digit classifier network. If you would like to explore the tools developed for the network, along with the analysis tools created for the following section, please refer to the full code available at:

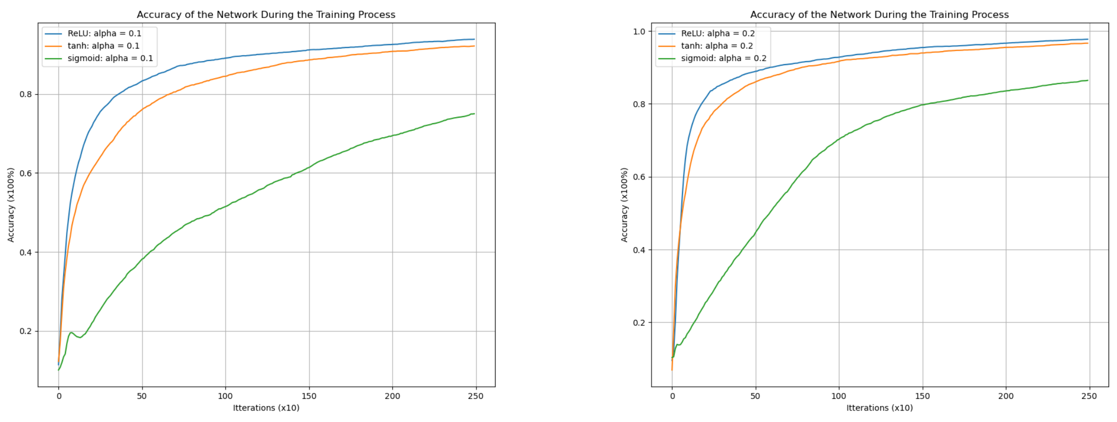
`https://github.com/JosephDavies427/Neural-Networks-Project`

Now that we have demonstrated how to create working code for neural networks based on our mathematical understanding of their workings, we can begin training digit classifier models. Additionally, we can use a selection of developed analysis tools to deepen our understanding of the concepts we have explored during this project and observe the implications and behaviours these concepts have on a functioning network.

4.2.1 Performance Analysis of Activation Functions

As we know, a critical design choice while creating a neural network is the choice of activation functions. In this section, we will analyse the impact of different activation functions on the performance and accuracy of our digit classifier network.

To facilitate this analysis, we trained three distinct networks on the same dataset, each utilising a different activation function. Throughout the training process, we recorded the evolution of the network's prediction accuracy over time. By comparing these metrics, we aim to uncover how different activation functions influence the learning behaviour and final performance of the network. The subsequent analysis will provide insights into the mathematical and practical implications of these design choices.



(a) Learning rate of 0.1.

(b) Learning rate of 0.2.

Figure 8: Plots of the accuracies of the networks during the training process for varying activation functions with 2500 training iterations.

In figure 8, we observe the accuracy behaviour patterns of the different activation functions. It is evident that the ReLU and tanh activation functions significantly outperform the sigmoid function. Both ReLU and tanh exhibit a rapid learning pattern during the initial stages of training, followed by a gradual rise in accuracy that stabilises consistently over time. In contrast, the sigmoid activation function demonstrates a much slower learning pattern, characterised by a gradual rise in accuracy that remains substantially lower than the other activation functions, even at the final stage of training.

While the ReLU model outperforms the tanh model in terms of final accuracy, it is worth carefully considering the behaviour of the plots before concluding that the ReLU function is objectively superior to the tanh function. The accuracy progression in the tanh model is notably smoother compared to the ReLU model, which exhibits more pronounced fluctuations during training. Although this variability in the ReLU model does not significantly affect its overall improvement or final performance in this specific case, it is an important characteristic to note. In other network configurations or in completely different use cases of neural networks, this 'jumpy' behaviour of ReLU could potentially become problematic.

To better understand the behaviour of the models during the later stages of training, we examine the plot in figure 9. This plot illustrates the accuracy progression for the three activation functions over a longer training duration, with the number of iterations significantly increased compared to the previous models seen in figure 8. By extending the training period, we aim to observe how the models optimise their configurations and whether their learning behaviour stabilises or continues to improve.

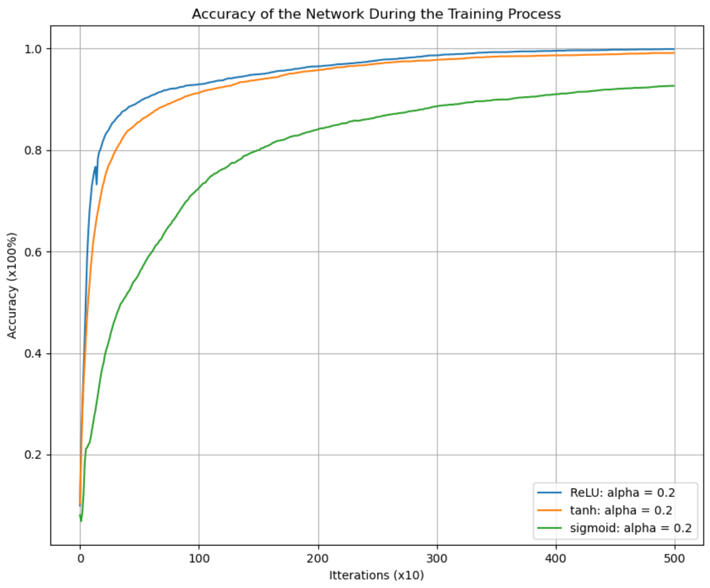


Figure 9: Plot of the accuracies between activation functions with 5000 training iterations.

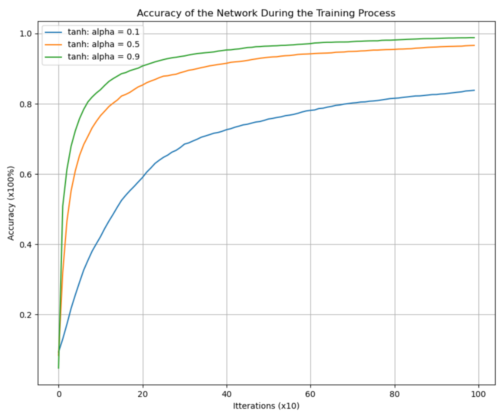
Figure 9 reinforces the conclusions drawn earlier while providing further insights into the plateauing behaviour of the models as training progresses. Both the ReLU and tanh models reach near-perfect accuracy, approaching 1, while the sigmoid model appears to stabilize at a lower accuracy of approximately 0.9. Although an accuracy of 0.9 is still a good result, it remains considerably below that of the other two models and may not suffice for tasks requiring very high reliability.

4.2.2 Performance Analysis of Learning Rates

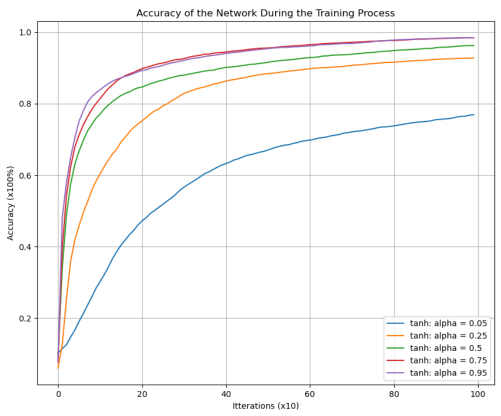
Another essential design consideration when training a neural network is the selection of an appropriate learning rate. Choosing the learning rate requires carefully balancing the speed of convergence to a reliable accuracy with maintaining the stability of the training process. In this section, we will examine how different learning rates affect the performance of the digit classifier network across various activation functions.

For this analysis, we will evaluate the accuracies of a series of networks, drawing comparisons similar to those made in section 4.2.1. However, in this case, we will observe the behaviour of neural networks while keeping the activation function fixed and varying the learning rate during the training process. This analysis will be performed for all three activation functions used in the previous sections. By comparing these results, we aim to identify patterns and trade-offs that arise with different learning rate configurations, offering insights into their influence on network behaviour and practical guidelines for tuning this parameter.

Effects of Varying Learning Rates with the Tanh Activation Function.



(a) Tanh activation function at 3 differing learning rates.

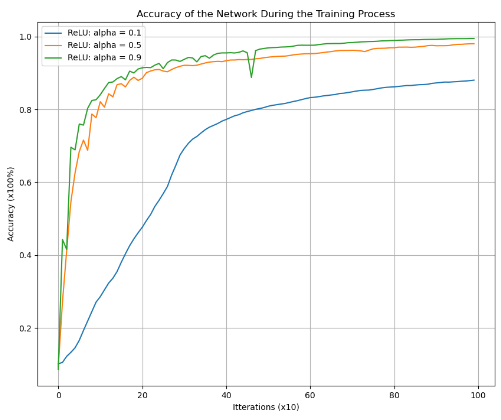


(b) Tanh activation function at 5 differing learning rates.

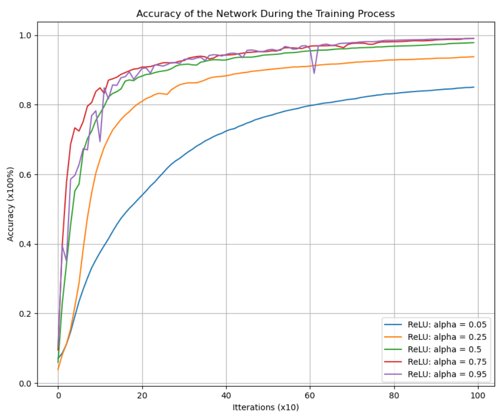
Figure 10: Plots of the accuracies of the tanh function in the training process at differing learning rates.

From figure 10, we observe various patterns in the learning behaviour of networks using the tanh activation function as the learning rate changes. Lower learning rates provide greater stability during the training process, with gradual and consistent improvements but fail to achieve the highest possible accuracy. In contrast, higher learning rates enable much faster convergence, allowing the model to reach high accuracy in fewer iterations, but at the risk of instability, which could become problematic in more complex scenarios. This analysis suggests that for the tanh activation function, the optimal learning rate lies approximately within the range of 0.5 to 0.75. However, this range is not definitive, as factors such as the dataset size, quality, and training conditions can greatly influence these results.

Effects of Varying Learning Rates with the ReLU Activation Function.



(a) ReLU activation function at 3 differing learning rates.

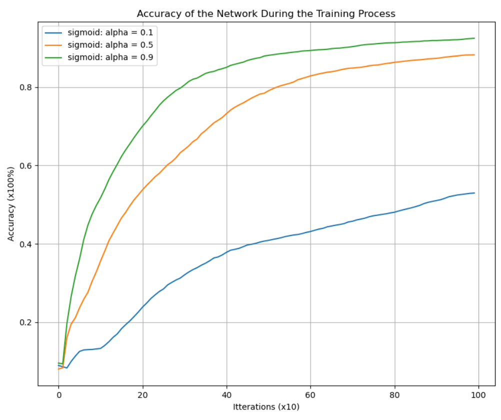


(b) ReLU activation function at 5 differing learning rates.

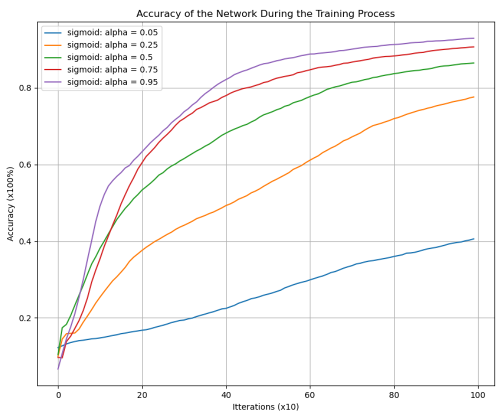
Figure 11: Plots of the accuracies of the ReLU function in the training process at differing learning rates.

Figure 11 highlights the impact of increasing the learning rate beyond the optimal range, with evident, pronounced fluctuations in accuracy for larger learning rates - particularly the $\alpha = 0.9$ line in figure 11a, and the $\alpha = 0.95$ line in figure 11b. Conversely, lower learning rates exhibit a stable learning pattern similar to that observed for the tanh model. However, as with previous observations, these rates result in a lower final accuracy by the end of the training process. Interestingly, the ReLU activation function demonstrates better performance at lower learning rates compared to tanh, achieving quicker convergence and higher final accuracy under the same conditions. Overall, the optimal learning rate for the ReLU model appears to be around 0.5, offering a reasonable balance between stability and accuracy. However, this value should be considered flexible, as many factors influence the choice of an appropriate learning rate, as we saw before.

Effects of Varying Learning Rates with the Sigmoid Activation Function.



(a) sigmoid activation function at 3 differing learning rates.



(b) sigmoid activation function at 5 differing learning rates.

Figure 12: Plots of the accuracies of the sigmoid function in the training process at differing learning rates.

The pattern of lower learning rates resulting in smoother but slower convergence, while higher learning rates provide faster convergence at the cost of less stable learning patterns, remains consistent in figure 12. However, unlike the ReLU results (Figure 11), the sigmoid networks do not exhibit significant instability, as the accuracy plots generally display a smooth progression throughout training. Despite this stability, the sigmoid function shows unpredictable and inconsistent behaviour across learning rates, making it more challenging to determine an exact optimal range. Based on the results, the estimated optimal learning rate range for sigmoid lies between 0.5 and 0.95. Overall, while the sigmoid function offers stable learning patterns, this stability comes at the expense of lower accuracy models, rendering it less reliable and effective for this particular application.

4.2.3 Analysis of Classification Errors

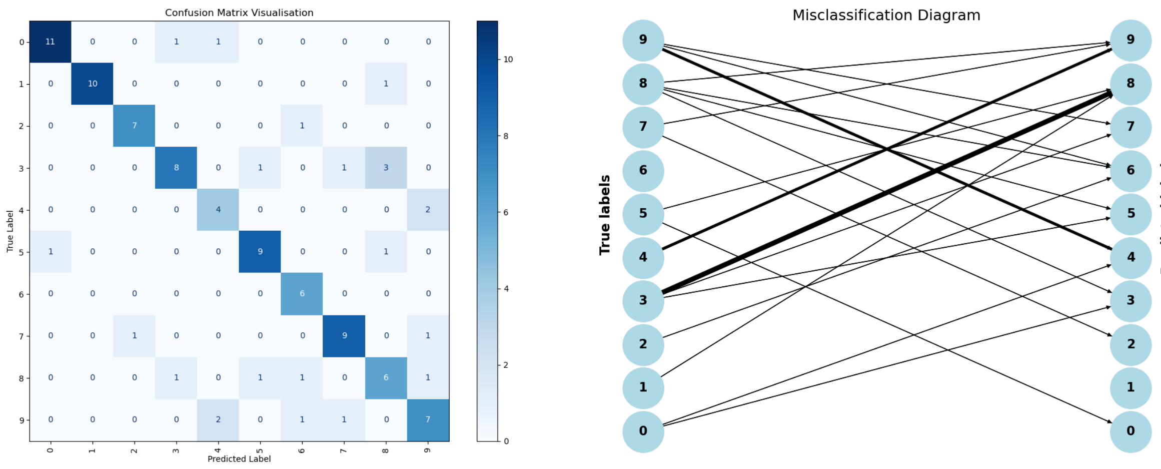
Now that we explored the behaviour of different activation functions during the training process and examined how they differ from one another, we turn our attention to the performance of these networks after training. Specifically, we aim to identify where the errors occur in their ability to label digits. To conduct this analysis, several tools have been developed.

First, a confusion matrix tool allows us to input a specified number of digit examples that differ from the training examples. This tool records both the correct and predicted labels for all the examples and displays the results in a confusion matrix. In this matrix, the labels on the left correspond to the correct labels, while the labels on the bottom correspond to the predictions made by the network. Second, a misclassification diagram tool provides an alternative visualisation of the same data. In this diagram, the correct labels are displayed on the left side, and the predicted labels are displayed on the right. Arrows between the two sides represent misclassification, with the thickness of the arrows indicating the frequency of specific errors. Finally, a tool is available to visualise individual training examples alongside their correct and predicted labels.

For this analysis, we trained networks for each activation function to achieve a consistent final accuracy, rather than being trained on a fixed number of training iterations as in the previous sections. The final accuracies under consideration are 80% and 95%. Using the tools mentioned, we aim to investigate where and why the networks fail, shedding light on the sources of their errors.

Understanding Errors Seen in a ReLU Network.

To begin, we analyse the errors made by a network using ReLU as its activation function, trained to an accuracy of 80% training accuracy. We will first examine the confusion matrix and misclassification diagram to identify common mislabelling patterns. While this provides insight into where the network struggles, it does not explain why these errors occur. To address this, we will then inspect specific misclassified examples to uncover potential reasons behind these mistakes.



(a) Confusion matrix. (b) Misclassification diagram.

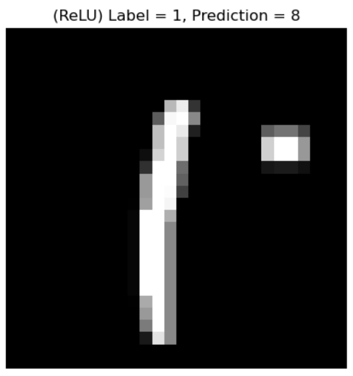
Figure 13: Plots from the error analysis tools based on a network with a ReLU activation function, trained to an accuracy of 80%.

From Figure 13, we can observe that the network performs well overall, as indicated by the strong diagonal in the confusion matrix (Figure 13a), reflecting a high rate of correct classifications. However, our primary focus is on the misclassifications. Notably, 3 is most commonly mistaken for 8, while 4 and 9 are frequently confused with each other.

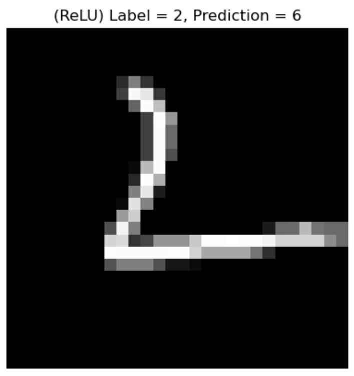
These errors become more intuitive when considering how digits are written. Small variations in handwriting style can cause one digit to resemble another, leading to misclassification. Even subtle differences in stroke placement or curvature can make a 3 appear more like an 8 or a 4 resemble a 9, which explains why these particular mistakes occur.

This also explains why digits like 0 and 1 are classified with high accuracy - their simple and distinct shapes make them less susceptible to stylistic variations that could cause misclassification.

Next, we will examine specific image examples that led to the errors observed, aiming to understand the underlying causes of these misclassifications.



(a) Misclassification of a 1.



(b) Misclassification of a 2.

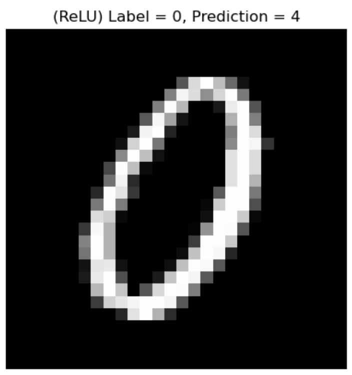
Figure 14: Examples of the errors made by the ReLU model trained to an accuracy of 80%

For our first example of errors, shown in figure 14, we can identify two key causes of misclassifications in our network.

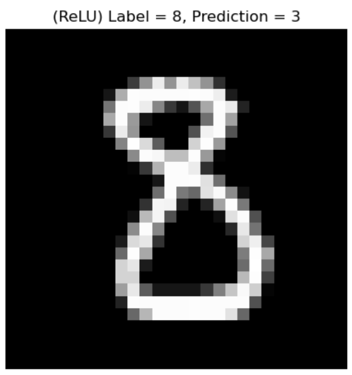
In the first case (Figure 14a), an image of a '1' has been mistaken for an '8'. This misclassification can be attributed to the presence of an artifact - a stray dot to the right of the digit. As humans, we can easily recognise this as irrelevant and still identify the digit correctly. However, since the neural network processes images purely based on the grayscale pixel values in the image, it lacks the ability to ignore these artifacts. The unexpected feature disrupts the network's learned pattern recognition, causing an incorrect prediction.

In the second case (Figure 14b), we see an image of a '2' that has been misclassified as a '6'. This misclassification likely stems from the way the digit was written—its shape is irregular and does not strongly resemble the typical form of a '2'. Instead, its loose and inconsistent structure makes it visually ambiguous, increasing the likelihood of confusion with other digits. Since the network relies on learned patterns from training data, an input that deviates too far from these patterns can lead to incorrect predictions, as the model has no deeper understanding of what makes a digit a '2' beyond its pixel arrangement.

Overall, these examples highlight that our neural network classifies digits purely based on the grayscale values of the image rather than any conceptual understanding of the numbers themselves. As a result, the network is highly susceptible to errors when faced with abnormal handwriting styles or noise within the examples, as it lacks the ability to disregard irrelevant artifacts or interpret inconsistencies the way a human would.



(a) Misclassification of a 0.

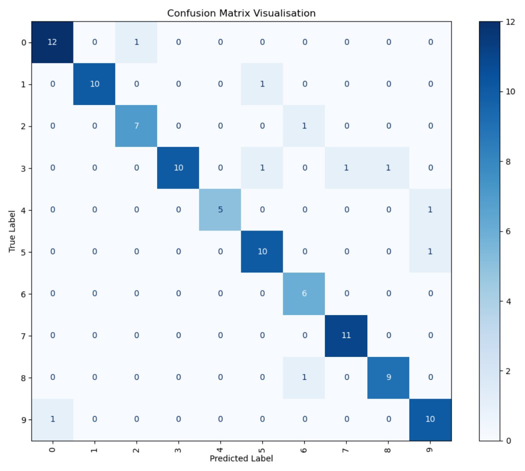


(b) Misclassification of an 8.

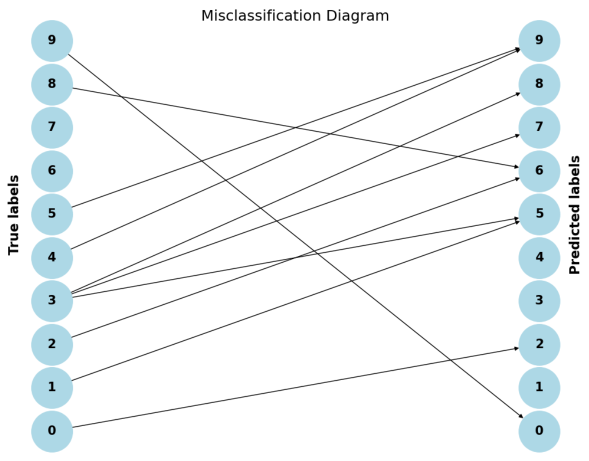
Figure 15: Examples of the errors made by the ReLU model trained to an accuracy of 80%

Not all errors can be attributed to the presence of artifacts or poor handwriting. In Figure 15, we see cases where the network failed to classify seemingly well-formed digits correctly. The first example (Figure 15a), an image of a 0 was misclassified as a 4, despite appearing to be a clear and properly shaped 0. Similarly, in the second example (Figure 15b) an 8 was mistaken for a 3, even though it maintains a well-defined structure. These cases highlight that, while many misclassifications can be explained by distortions or artifacts, some errors arise even when the digit appears unambiguous to the human eye, indicating inherent limitations in the network's learned representations.

To further explore these errors, we will now examine misclassifications in a different trained network model. As before, we will use the ReLU activation function, but this time, the model has been trained to an accuracy of 95%.



(a) Confusion matrix.

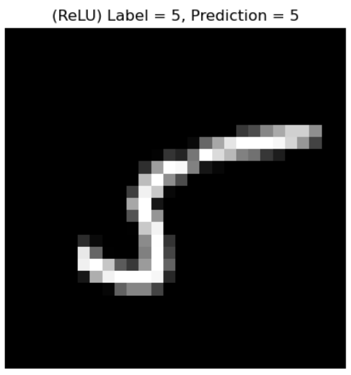


(b) Misclassification diagram.

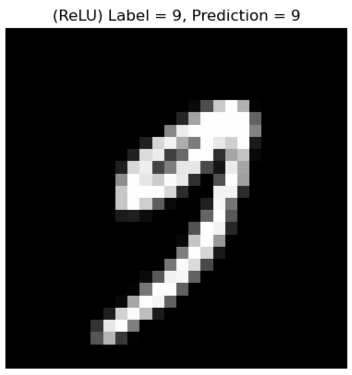
Figure 16: Plots from the error analysis tools based on a network with a ReLU activation function, trained to an accuracy of 95%.

As expected, we see a significant reduction in errors compared to the previous model, as shown in Figure 16. In this case, there is no clear pattern of recurring errors, suggesting a more balanced classification performance, as indicated by the non-diagonal entries in the confusion matrix never exceeding 1. However, we do observe multiple misclassifications involving the digit 3. As previously discussed, this may not necessarily indicate a weakness in the network itself, but rather that the digit 3 shares visual similarities with multiple other digits, making it inherently more challenging to classify.

Similar to our analysis of the previous model, we will now examine specific examples of misclassified digits, as well as some correctly classified cases.



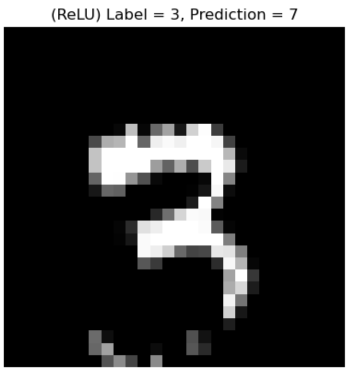
(a) Correct classification of a 5.



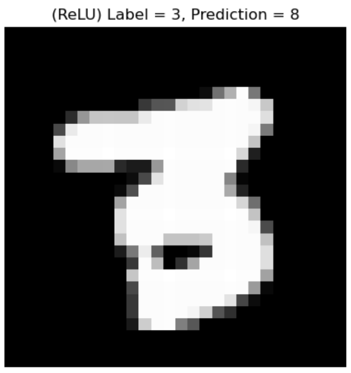
(b) Correct classification of a 9.

Figure 17: Examples of correct classification made by the 95% accurate model where the 80% model failed.

Figure 17 presents two notable cases of correct classification. In our previous 80% accurate model, these specific images were misclassified, likely due to poor handwriting, as previously discussed. Given this, it is remarkable that despite only a moderate increase in training, the new model correctly identifies these challenging digits. This suggests that, even with an imperfect model, the network has improved its ability to generalise across a wider range of handwriting styles, purely by refining its parameters to better accommodate a wider range of examples.



(a) Misclassification of a 3.



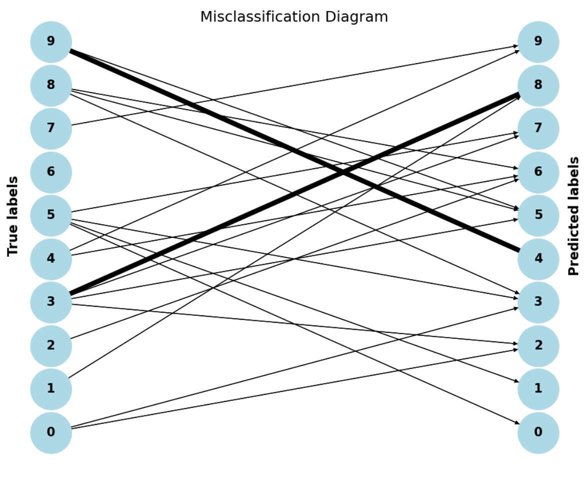
(b) Misclassification of a 3.

Figure 18: Examples of the errors made by the ReLU model trained to an accuracy of 95%

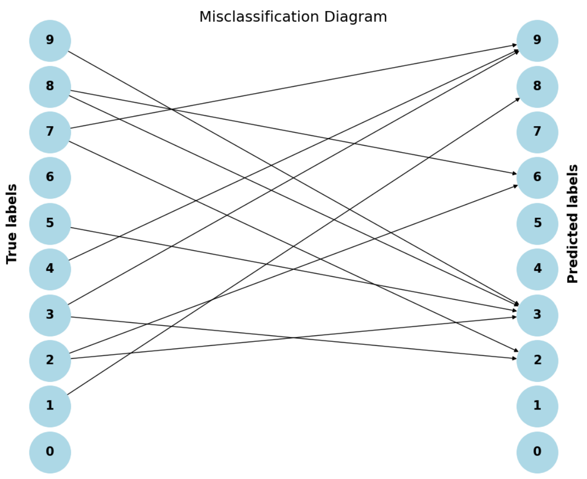
Despite the improvements in accuracy, the model still remains susceptible to errors when presented with images containing noise or handwriting styles that deviate from those seen during training. This is evident in Figure 18, where two different instances of the digit 3 have been misclassified. The first example (Figure 18a) demonstrates how the network struggles with noisy input, leading to incorrect predictions, similar to previously observed cases. In the second example (Figure 18b), the increased stroke thickness causes the digit to resemble a different number, making it more difficult for the model to distinguish. These cases highlight the ongoing challenges posed by handwriting variability and emphasise the model's limitations in generalising across diverse writing styles.

Looking at the Differences in Tanh and Sigmoid Networks.

In the following section we will take a general look at how our other activation functions, tanh and sigmoid, perform under the same form of analysis. While we will not delve into the same level of detail as before, we will take a general overview of their performance using misclassification diagrams. This broader comparison still provides valuable insights into how different activation functions influence model performance and generalisation. By examining their misclassification patterns, we can identify recurring challenges and assess whether the trends observed in ReLU-based networks persist across alternative activation functions.



(a) 80% accuracy model.

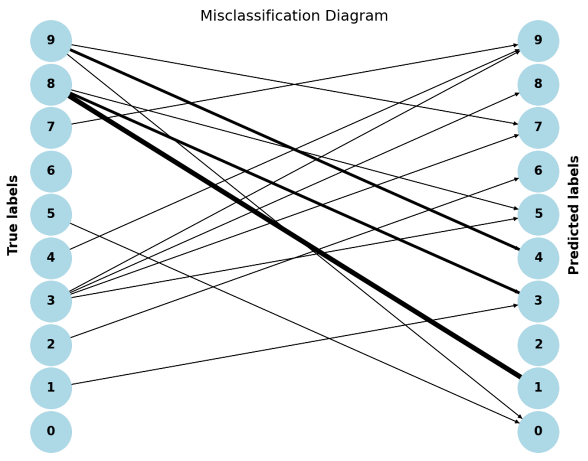


(b) 95% accuracy model.

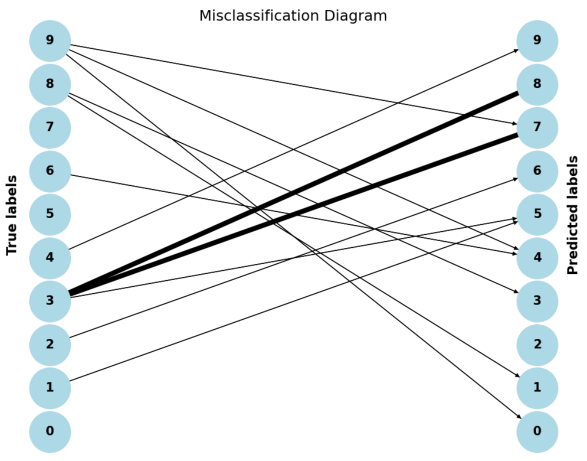
Figure 19: Misclassification diagrams from two tanh models trained to different accuracies.

Figure 19 shows that the tanh networks follow a similar pattern to what we observed with the ReLU networks, demonstrating improved generalisation as training progresses. This is both expected and desirable, as increased training (up to a limit) should lead to better model performance. While the overall misclassification trends align with those seen in the ReLU networks, the reasoning behind these errors also remains consistent. The primary weaknesses of the networks can still be attributed to the same understandable challenges, image noise and/or misleading handwriting style, as discussed in prior analysis.

Notably, the same key digits - such as 3s, 5s, 8s, and 9s - continue to pose difficulties for the model. This is likely due to their curvy forms, which can vary significantly depending on handwriting style, making them more prone to being mistaken for one another.



(a) 80% accuracy model.



(b) 95% accuracy model.

Figure 20: Misclassification diagrams from two sigmoid models trained to different accuracies.

Figure 20 highlights the differing pattern that emerges in the sigmoid models. Similar to what was observed in previous analyses with ReLU and tanh, the sigmoid model trained to 80% accuracy exhibits a wide range of errors, with certain digits being misclassified more frequently than others. However, this pattern shifts when examining the sigmoid model trained to 95% accuracy, which does not follow the expected trend of improved generalisation seen in other activation functions.

While the total number of errors decreases with higher training accuracy, the model still exhibits significant weaknesses - most notably in misclassifying the digit 3. This suggests that the network may have encountered overfitting issues during training. Overfitting occurs when optimisation focuses primarily on improving accuracy on the training set at the expense of generalisation, leading to weakness in handling new or varied examples.

4.2.4 Analysis Conclusion

Before drawing conclusions about our findings and their implications for neural networks, it is important to first acknowledge the limitations and key considerations that should be kept in mind when interpreting this research.

First and foremost, the observed performances of these networks are entirely constrained by the specific conditions under which they were tested: such as the chosen network architecture, training duration, and the particular task of handwritten digit classification. Additionally, the purpose of this analysis was not to provide an exhaustive investigation but rather to offer a surface-level examination of working networks. This was intended to supplement the foundational understanding developed earlier (Sections 2 and 3), which helps to contextualise the theoretical concepts we have explored.

With this in mind, the insights drawn from our observations should not be taken as universally applicable conclusions about neural networks and activation functions. Instead, they should be viewed as complementary insights that align with and reinforce the theoretical principles derived from prior studies.

With these considerations established, we will now summarise our key observations from the analysis, identify emerging patterns, and explore potential explanations for the trends we have seen.

For this application, it was evident that networks using the sigmoid activation function were not the best suited for the intended purpose. We observed that the sigmoid networks required significantly more training to achieve a reasonable accuracy, which remained consistent, even when a range of different learning rates were tested. Furthermore, even when given sufficient time to train to a competitive accuracy, the sigmoid networks demonstrated signs of overfitting issues when encountering examples outside the training dataset. Given these factors, it is clear that the sigmoid function would not be a preferred choice for a digit classification network structured in this manner.

This leaves us with the ReLU and tanh activation functions, both of which proved to be well-suited for the task. They exhibited strong learning capabilities, rapid convergence, and good generalisation to new examples beyond the training data. In terms of overall performance, there was little that separated the two, as both showed strong learning patterns and achieved similar levels of accuracy. When deciding between these two functions for this specific network, the differences in their advantages and drawbacks must be considered.

ReLU demonstrated a rapid learning pattern that was unmatched by sigmoid or tanh, making it highly desirable, especially given the potential risks of overfitting with prolonged training periods. The ability to quickly reach high accuracy is beneficial, as excessive training time can lead to diminishing returns through weak generalisation, as seen with the sigmoid networks. However, ReLU also exhibited some instability in accuracy during training, particularly at higher learning rates. Although this instability did not prevent the network from ultimately achieving high accuracy, it raises concerns about potential overfitting if pushed to extreme training accuracies (e.g., above 99%).

On the other hand, the tanh activation function also produced strong results, performing comparably to ReLU while demonstrating greater stability during training. Although its learning speed was slightly slower than ReLU, it was still significantly better than sigmoid. The consistency in learning patterns suggests that tanh-based networks may produce more repeatable and reliable results. This increased stability could be advantageous in ensuring robust network performance across different training setups. However, as with ReLU, there remains the concern that pushing to extreme training accuracies could lead to overfitting, particularly if the network is trained for excessively long periods.

Overall, both ReLU and tanh prove to be strong choices for this application, with ReLU offering rapid convergence and tanh providing more stable learning behaviour. The decision between them would largely depend on the specific priorities of the task — whether a faster learning rate is preferred despite potential instability, or whether a more consistent but slightly slower learning process is more desirable. In either case, both activation functions exhibit clear advantages over sigmoid in this context, reinforcing the importance of activation function selection in neural network performance.

5 Literature Review

At this point, we have developed a strong understanding of the key concepts that underpin neural networks, both through our theoretical review and hands-on experience with working models. In this chapter, we will examine several research papers that explore specific subtopics within neural networks and machine learning in greater depth.

The purpose of reviewing these papers is not only to reinforce the knowledge we have already built throughout this project, but also to broaden our understanding of the field. The selected papers address topics that were initially discussed only briefly or left unsolved (such as advanced activation functions or techniques for preventing overfitting) because a deeper understanding of these concepts was not essential for grasping the core principles of neural networks. This literature review aims to provide those more advanced insights.

Additionally, this review will introduce more complex machine learning concepts that extend beyond the traditional neural network structures explored in this project. By doing so, it will offer a glimpse into the advancements at the forefront of the neural network and machine learning field, providing a foundation for further exploration of the field's evolving methodologies.

5.1 Dropout: A Simple Way to Prevent Neural Networks from Overfitting

When analysing our networks in section 4.2, we encountered the problem of overfitting. But what exactly is overfitting, and how can we prevent it? The following paper explores dropout, a widely used technique designed to mitigate overfitting in neural networks.

As we have previously discussed, overfitting is a common challenge when training neural networks. It occurs when a network's parameters become overly specialised to the training data, preventing the model from generalising effectively to unseen datasets with the same intended purpose.

One of the main difficulties in addressing overfitting is determining when a network has been trained sufficiently, without excessive computational overhead. A classic approach could involve cyclical training and testing, where the model is trained in small batches, and training continues until improvements in performance on the training set begin to degrade performance on the test set. However, for larger and more complex networks, balancing training effectiveness while keeping computational and financial costs manageable remains a significant challenge.

To tackle overfitting, researchers have developed various regularisation techniques, one of which is dropout. In the following section, we will examine the paper "Dropout: A Simple Way to Prevent Neural Networks from Overfitting" [16] to explore how this method can help mitigate overfitting and improve model generalisation.

How Dropout Works

During training, dropout can be applied as a regularisation technique by randomly deactivating neurons. In each cycle of forward and backward propagation, a neuron is either included or excluded from the network with some probability p (typically $0.5 \leq p < 1$). This means that, during training, the model optimises its parameters using a thinned-out version of the network, where different subsets of neurons are active in each cycle.

This approach prevents specific neurons from becoming overly specialised in recognising certain patterns, reducing the risk of the network developing biased dependencies on particular subsets of neurons. Instead, the entire network is encouraged to learn generalisable features, leading to improved performance on unseen test data.

At test time, since all neurons are now active, the parameters must be scaled by p to prevent the network from becoming overly active. This scaling ensures that the overall expected output remains consistent between training and testing, preserving the benefits of dropout without introducing instability.

Mathematical model of dropout

To formally describe how dropout modifies forward propagation, we revisit our forward propagation model 4. To represent dropout, we introduce a dropout mask, $\vec{r^{(l)}}$, where:

$$r_j^{(l)} \sim \text{Bernoulli}(p)$$

Each element of $\vec{r^{(l)}}$ is independently sampled from a Bernoulli distribution with probability p. This means that each neuron is kept ($r_j^{(l)} = 1$) with probability p and dropped ($r_j^{(l)} = 0$) with probability $1 - p$.

As we have already seen, in a fully connected network, the standard forward propagation equation for each neuron is given by:

$$a_j^{(l)} = f(Z_j^{(l)}) \quad \text{where,} \quad Z_j^{(l)} = b_j^{(l)} + \sum_{i=1}^{n} w_{j,i}^{(l)} \cdot a_i^{(l-1)}$$

To apply dropout, we modify the equation by incorporating the dropout mask as follows:

$$a_j^{(l)} = f(Z_j^{(l)}) \cdot r_j^{(l)} \quad \text{where,} \quad Z_j^{(l)} = b_j^{(l)} + \sum_{i=1}^{n} w_{j,i}^{(l)} \cdot a_i^{(l-1)} \cdot r_i^{(l-1)}$$

This ensures that during training, only a subset of neurons contribute to each forward pass, as some neurons are randomly deactivated.

For the vectorised form, as we have previously seen, forward propagation is given by:

$$\vec{a^{(l)}} = f\left(W^{(l)} \cdot \vec{a^{(l-1)}} + \vec{b^{(l)}}\right)$$

With dropout applied, this becomes:

$$\vec{a^{(l)}} = \vec{r^{(l)}} \odot f\left(W^{(l)} \cdot (\vec{r^{(l-1)}} \odot \vec{a^{(l-1)}}) + \vec{b^{(l)}}\right)$$

Where $\odot$ represents the Hadamard product (element-wise multiplication), which operates as follows:

$$\begin{bmatrix} a \\ b \\ c \end{bmatrix} \odot \begin{bmatrix} d \\ e \\ f \end{bmatrix} = \begin{bmatrix} a \cdot d \\ b \cdot e \\ c \cdot f \end{bmatrix}$$

This formulation ensures that, during training, different neurons are randomly deactivated in each iteration, improving generalisation.

At test time, dropout is no longer applied, meaning all neurons are active. However, to maintain the expected value of activations (since fewer neurons were used in training), the network scales the activations by p as seen below:

$$\vec{a^{(l)}} = p \cdot f\left(W^{(l)} \cdot \vec{a^{(l-1)}} + \vec{b^{(l)}}\right)$$

This ensures that the expected activation values remain consistent between training and testing, preventing the network from becoming over active.

Experimental Results & Effectiveness of Dropout

In this paper, dropout was tested across a range of tasks, from simple to complex, using multiple benchmark datasets. These included the MNIST dataset, which we have previously used, as well as more advanced datasets such as TIMIT - a widely used speech recognition dataset - and ImageNet, a large-scale dataset of natural images (similar to those found in CAPTCHA tests).

To evaluate the effectiveness of dropout, the authors trained multiple networks on these datasets, both with and without dropout, across different network sizes. The trained models were then tested on unseen data, and their classification error rates were recorded. Across all experiments, networks that utilised dropout achieved lower error rates, demonstrating improved generalisation. The benefits of dropout were especially visible in more complex tasks, such as image classification on ImageNet, compared to simpler tasks like digit recognition on MNIST. However, performance improvements were observed across all datasets.

Strengths & Limitations of Dropout

One major downside of the dropout method of regularisation is the increase in training time. Training a model with dropout to achieve the same level of accuracy as a standard model can take 2 to 3 times longer. This is because the final trained network is essentially a combination of many randomly generated network architectures with shared components. Since each randomly 'thinned' network has its own unique gradient during training, the model must optimise across many different gradient paths. As a result, additional training time is required to ensure that the final network configuration reaches a suitable minimum from fine-tuning across these smaller, random architecture-specific gradients.

Another challenge with dropout is the sensitivity to the dropout rate p. While dropout helps prevent weights and biases from drifting towards overfitting, the introduction of the hyper parameter p creates a new tuning challenge. If the dropout rate is too high, the model underfits, failing to learn enough useful features, which leads to low overall accuracy. If the dropout rate is too low, the training process behaves similarly to a standard network, reducing the effectiveness of dropout. However, when p is chosen appropriately, and a balance is found between training time and overfitting prevention, dropout proves to be a highly valuable regularisation technique. Where applicable, it remains a widely used and effective addition to most neural networks.

Implications for Our Neural Networks

The insights gained from this paper provide valuable considerations for revisiting the neural networks constructed in section 4. A key challenge encountered during training was overfitting, where the networks performed well on training data but struggled to generalise to unseen data. Incorporating dropout could serve as an effective solution by reducing reliance on specific features, thereby enhancing the network's ability to generalise. While this approach would come at the cost of increased training time, applying these techniques could significantly improve the robustness of our earlier models - specifically our sigmoid models.

5.2 A Novel Activation Function of Deep Neural Network

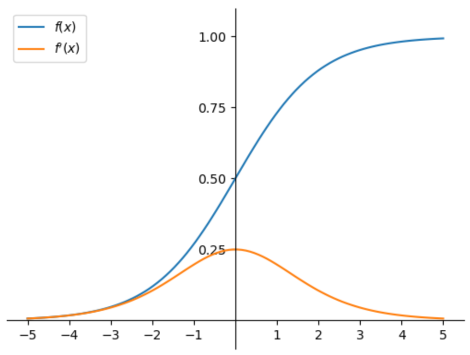
In section 2.1.5, we briefly mentioned that the activation functions covered earlier in this project represent only a small subset of the vast number of possible activation functions. In this section, we explore the motivation and process behind designing custom activation functions, providing insight into the key considerations when selecting and implementing them within neural networks.

While the activation functions discussed so far are widely used and effective in many scenarios, they also come with inherent limitations. These limitations drive ongoing research into new activation functions that aim to mitigate their downsides. To understand why new activation functions are needed, we must first examine the drawbacks of the commonly used ones.

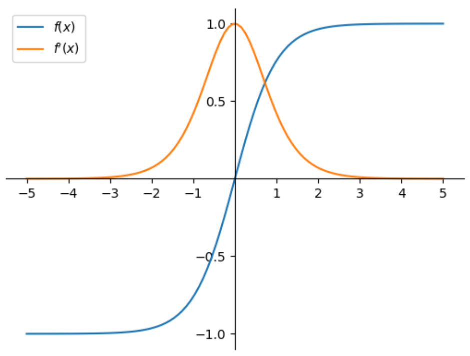
Several studies, including "A Novel Activation Function of Deep Neural Network" [17], have analysed the limitations of traditional activation functions and their impact on deep network performance. In this section, we extend that discussion by revisiting some of the most common activation functions and analysing their behaviour, particularly in the context of backpropagation.

Vanishing Gradient Problem in Sigmoid and Tanh

Let us revisit the sigmoid and tanh activation functions, this time also considering their derivatives.



(a) $f(x) = \text{sigmoid}(x)$



(b) $f(x) = \tanh(x)$

Figure 21: Plots of the sigmoid and tanh functions along with their derivatives.

Both sigmoid and tanh have desirable properties when considering forward propagation, such as smooth gradients and bounded outputs. However, their derivatives introduce a fundamental problem in deep neural networks. Observing figure 21, we see that the derivative of each function is strictly positive and returns values within a limited range:

- The sigmoid derivative is always in the range $(0, 0.25]$, achieving its maximum at $x = 0$.
- The tanh derivative in the range $(0, 1]$, with a maximum at $x = 0$ and decreasing rapidly for larger absolute values of x.

To understand why this is a problem, let us recall the gradient equations derived in section 3.2:

$$\frac{\partial C}{\partial w_{j,k}^{(l)}} = a_k^{(l-1)} \cdot f'(z_j^{(l)}) \cdot \frac{\partial C}{\partial a_j^{(l)}} \qquad \text{and} \qquad \frac{\partial C}{\partial b_j^{(l)}} = f'(z_j^{(l)}) \cdot \frac{\partial C}{\partial a_j^{(l)}}$$

$$\text{where} \quad \frac{\partial C}{\partial a_j^{(l)}} = \sum_{j=1}^{n_{l+1}} w_{j,k}^{(l+1)} \cdot f'(z_j^{(l+1)}) \cdot \frac{\partial C}{\partial a_j^{(l+1)}}$$

for an arbitrary activation function $f(x)$.

Each gradient term depends on the derivative $f'(z)$. In deep networks, this derivative appears repeatedly as gradients are propagated backward through many layers. Since the derivatives of sigmoid and tanh are always smaller than 1, the repeated multiplication of small values causes an exponential decay in the gradient magnitude as we move towards the earlier layers.

This results in vanishing gradients, where the updates to the parameters of earlier layers become negligibly small, causing them to learn extremely slowly. Consequently, deep networks relying on sigmoid or tanh activation functions struggle to efficiently train because the earlier layers fail to capture meaningful features.

Dying Neuron Problem in ReLU

ReLU is one of the most widely used activation functions in modern neural networks primarily due to its simplicity and computational efficiency. Let us recall the definition of the ReLU function:

$$f(x) = \begin{cases} 0 & \text{if } x \leq 0 \\ x & \text{if } x > 0 \end{cases}$$

ReLU is favoured over earlier activation functions such as sigmoid and tanh because it avoids the vanishing gradient problem. Unlike these functions, ReLU does not saturate for large positive values, and its derivative remains 1 in this region, allowing gradients to propagate effectively during back propagation. However, despite these advantages, ReLU introduces another issue: the dying neuron problem.

The dying neuron problem occurs when a large number of neurons in a network consistently output zero, effectively removing them from the learning process. This happens because the derivative of ReLU is zero for all negative inputs:

$$f'(x) = \begin{cases} 0 & \text{if } x \leq 0 \\ 1 & \text{if } x > 0 \end{cases}$$

During training, if a neuron's input becomes negative, its output is set to zero. This means that, in subsequent training iterations, its gradient will also be zero, preventing it from updating its weights. If this continues over multiple updates, the neuron effectively stops learning, leading to what is known as a dead neuron.

This issue can become particularly severe in deep networks, when large negative weight updates can cause entire layers of neurons to become inactive. Once a neuron dies, it no longer contributes to the learning process, reducing the network's overall capacity and expressiveness.

The TSin Activation function

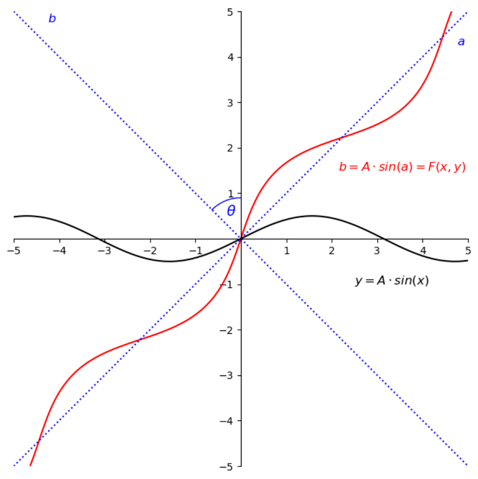
In this paper, the authors propose a novel activation function, the TSin function, designed to introduce controlled periodicity and smooth transformations in certain models. Unlike traditional activation functions such as ReLU, tanh, or sigmoid, TSin exhibits oscillatory behaviour, allowing for a more expressive representation of non-linear transformations while maintaining desirable differentiability properties.

The TSin function is defined by the implicit equation:

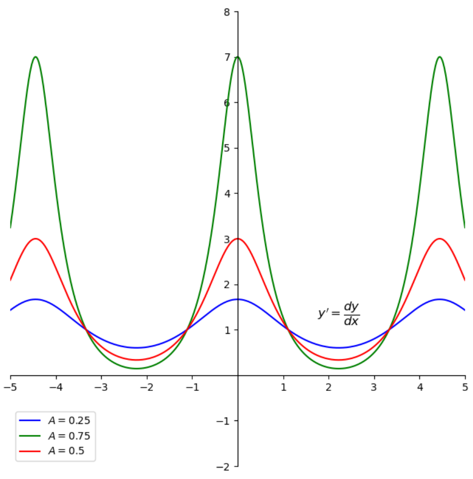
$$F(x,y) = (y-x) - \sqrt{2}A\sin\left(\frac{\sqrt{2}}{2}(x+y)\right) = 0 \qquad where, \qquad (0 < A < 1)$$

This equation implicitly defines the output y in terms of x, incorporating a sine function modulated by the parameter A. The presence of the sine term introduces controlled oscillations, distinguishing TSin from conventional activation functions.

Geometrically, the TSin function can be viewed as a sine wave that has undergone a $\pi/4$ anti-clockwise rotation. This transformation results in a structured periodic pattern that balances smooth variation with nonlinearity, offering a distinctive alternative to sine-based functions. This behaviour is illustrated in figure 22a.



(a) TSin Function (Red): $A = 0.5, \theta = \pi/4$



(b) TSin derivative for varying A values.

Figure 22: Visualisation of the TSin activation function and its derivative.

The derivative of the TSin function is given by:

$$y' = \frac{dy}{dx} = \frac{1 + Acos((x + y)/\sqrt{2}))}{1 - Acos((x + y)/\sqrt{2}))}$$

Figure 22b illustrates the characteristics of the TSin derivative. Notably, as long as the parameter A is chosen away from its upper bound of 1, the derivative avoids values near zero. This directly addresses the vanishing gradient problem that severely affects activation functions like sigmoid and tanh. Unlike these traditional functions, TSin maintains a stable gradient flow, enabling more effective backpropagation.

Another crucial advantage of the TSin derivative is its periodic behaviour, which helps prevent the dying neuron issue commonly seen in ReLU-based networks. The periodic nature of TSin ensures that the function remains expressive across the entire real line, keeping all neurons engaged throughout the training process. Additionally, the bounded nature of its derivative helps regulate weight updates, avoiding the instability from exploding gradients, provided that A remains sufficiently below 1. These properties collectively make TSin a well-suited alternative to conventional activation functions, particularly in deep networks where gradient stability is crucial for effective learning.

Experimental Insights of TSin

The practical evaluation of the TSin function was conducted in this paper across multiple benchmark tasks, comparing its performance against ReLU, Leaky ReLU (LReLU, a variation of ReLU where the negative section is $\alpha \cdot x$ for some $0 < \alpha < 1$ instead of being set to zero as in the standard ReLU function), and Sin activation functions. Initial tests were performed on a simplified 5 by 5 pixel digit recognition task, where the TSin-based model significantly outperformed all other activation functions. This demonstrated its effectiveness in learning small-scale image representations with high precision.

Further experiments extended to the familiar MNIST dataset (28-by-28 pixel digit images), where TSin once again outperformed the other activation functions. The results showed not only a faster convergence rate but also a higher final accuracy, indicating that TSin provides more stable gradient updates and effective feature recognition in deep learning models. Additional testing on the CIFAR-10 dataset, which consists of more complex natural images, revealed a smaller performance gap between TSin and the other activation functions. However, TSin still achieved the best overall performance, suggesting that while its advantages persist in more challenging scenarios, the increased complexity of natural image data may somewhat limit the extent of its benefits.

Implications for Our Neural Networks

The findings in this paper highlight the range of factors that must be considered when selecting an activation function. While TSin may not be a universal solution for our networks, it is important to assess the impact of the limitations discussed in this paper on the models we have already developed. As demonstrated in the experimental results, TSin helped maintain a stable gradient flow. If applied to our networks, this could lead to improved leaning dynamics, faster convergence, and greater accuracy.

6 Conclusions

This project has provided a comprehensive study of neural networks, focusing on their mathematical foundations, practical applications, and performance optimisation. We began by exploring the core mathematical principles that underpin neural networks, including their individual components, mathematical models, and the mechanisms of forward and backward propagation. These theoretical concepts were then applied in practice by constructing and analysing our own neural networks, allowing us to gain first-hand experience in how networks learn. Through these implementations, we observed both the strengths and challenges of training neural networks, such as overfitting and gradient-related issues, reinforcing the need for careful network design.

Following this, the project investigated potential improvements to neural network performance. Our experimental work highlighted the importance of activation function and learning rate choices for achieving better generalisation and stability. Additionally, through our literature review, we gained insights into dropout regularisation, which provides a method for reducing overfitting by randomly deactivating neurons during training. We also explored the potential of alternative activation functions, such as TSin, a function designed to maintain gradient flow while avoiding limitations found in classic activation functions. While these solutions demonstrated their ability to create more effective and robust networks, they also introduced trade-offs, such as increased training time and the need for careful parameter tuning.

Beyond these specific improvements, this project emphasised the crucial role of mathematical reasoning in neural network development. By analysing how networks process and optimise information, we gained a deeper understanding of the mechanisms driving their performance. Rather than viewing neural networks as rigid and fragile structures that should not deviate from their foundational designs, this approach encouraged a more critical and informed evaluation of alternative design choices. This was further reinforced by the literature review, which introduced various alternative approaches to network design, highlighting the importance of continuous research and iterative improvements in network architectures and learning mechanisms.

While this project has laid a strong foundation, many directions for future exploration remain. Expanding the range of experiments, such as creating and testing networks on more complex datasets, experimenting with additional activation functions, or exploring alternative regularisation methods, would provide deeper insights into network performance, strengths, and limitations, as we have already seen in this project. For instance, given the performance variability observed with different learning rates and activation functions, future work could investigate adaptive optimisers such as Adam or RMSProp [18, 19], or experiment with functions like ELU or leaky ReLU to address the limitations seen in ReLU [20, 21].

Additionally, further research could extend into the broader field of machine learning, exploring architectures such as transformers, generative models, and reinforcement learning systems. Each of these builds upon the mathematical principles studied in this project, illustrating how fundamental concepts translate into cutting-edge advancements. Bridging this project's foundational approach with more advances models could not only validate the core mathematics but also offer practical insights into current AI techniques.

Ultimately, this project has reinforced the idea that neural networks are not just computational models but deeply mathematical structures. Understanding their inner workings enables more informed decision-making when designing them. By bridging theory with hands-on implementation and critical evaluation, this work provides a solid foundation for further research and exploration in deep learning.

References

[1] W.S. McCulloch, W. Pitts: *(1943)*, A Logical Calculus of the Ideas Immanent in Nervous Activity. The Bulletin of Mathematical Biophysics, 5(4). doi: https://doi.org/10.1007/bf02478259.

[2] Donald Olding Hebb: *(1949)*, The Organization of Behavior. John Wiley & Sons.

[3] F. Rosenblatt: *(1958)*, The Perceptron: A Probabilistic Model for Information Storage and Organization in the brain. Psychological Review, 65(6). doi: https://doi.org/10.1037/h0042519.

[4] S. Haykin: *(2009)*, Neural Networks and Learning Machines. New York: Prentice Hall/Pearson, pp.77-78.

[5] M.A. Nielsen: *(2015)*, Neural Networks and Deep Learning. Self Published.

[6] I. Goodfellow, Y. Bengio and A. Courville: *(2016)*, Deep Learning. MIT Press..

[7] G. Sanderson: *(2017)*, 3blue1brown. [online] www.3blue1brown.com Available at: https://www.3blue1brown.com/topics/neural-networks [Accessed Jun. 2024].

[8] S.R. Dubey, S.K. Singh and B.B. Chaudhuri: *(2022)*, Activation Functions in Deep learning: A Comprehensive Survey and Benchmark. Neurocomputing, 503. doi: https://doi.org/10.1016/j.neucom.2022.06.111.

[9] A. Zhang, Z.C. Lipton, M. Li and A.J. Smola: *(2021)*, Forward Propagation, Backward Propagation, and Computational Graphs. [online] d2l.ai Available at: https://d2l.ai/chapter_multilayer-perceptrons/backprop.html. [Accessed Oct. 2024].

[10] A.F. Agarap: *(2018)*, Deep Learning using Rectified Linear Units (ReLU) Neural and Evolutionary Computing. doi: https://doi.org/10.48550/arXiv.1803.08375

[11] M. Franke and J. Degen: *(2023)*, The softmax function: Properties, motivation, and interpretation *. [online] Available at: https://alpslab.stanford.edu/papers/FrankeDegen_submitted.pdf [Accessed 2 Oct. 2024].

[12] X. Ying: *(2019)*, An Overview of Overfitting and its Solutions Journal of Physics: Conference Series. Available at: https://www.researchgate.net/publication/331677125_An_Overview_of_Overfitting_and_its_Solutions

[13] V. Bhattbhatt: *(2024)*, Learning Rate and Its Strategies in Neural Network Training. [online]Available at: https://medium.com/thedeephub/learning-rate-and-its-strategies-in-neural-network-training-270a91ea0e5c.

[14] L. Deng: *(2012)*, The mnist database of handwritten digit images for machine learning research. IEEE Signal Processing Magazine, 29(6), 141–142..

[15] A. Mao, M. Mohri and Y. Zhong: *(2023)*, Cross-Entropy Loss Functions: Theoretical Analysis and Applications. arXiv. Available at: https://arxiv.org/abs/2304.07288

[16] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever and R. Salakhutdinov: *(2014)*, Dropout: A Simple Way to Prevent Neural Networks from Overfitting. Available at: https://dl.acm.org/doi/10.5555/2627435.2670313.

[17] L. Xiangyang, Q. Xing, Z. Han and C. Feng: *(2023)*, A Novel Activation Function of Deep Neural Network. doi: https://doi.org/10.1155/2023/3873561.

[18] D. P. Kingma and J. Lei Ba: *(2025)*, Adam: A Method for Stochastic Optimization. doi: https://doi.org/10.48550/arXiv.1412.6980.

[19] Z. Hong: *(2024)*, Adaptive Learning Rate Scheduling: Optimizing Training in Deep Networks [online]Available at: https://medium.com/@zhonghong9998/adaptive-learning-rate-scheduling-optimizing-training-in-deep-networks-14d4f95a45d6.

[20] P. Ramachandran, B. Zoph and Q. V. Le: *(2017)*, Searching for Activation Functions. doi: https://doi.org/10.48550/arXiv.1710.05941.

[21] P. Baheti: *(2021)*, Activation Functions in Neural Networks [12 Types & Use Cases] [online]Available at: https://www.v7labs.com/blog/neural-networks-activation-functions.

Email: joseph.a.davies@students.plymouth.ac.uk