Machine Learning and Transformer Networks in Stock Market Prediction

Barney Todd

November 2023

This piece of work is a result of my own work and I have complied with the Department's guidance on multiple submission and on the use of AI tools. Material from the work of others not involved in the project has been acknowledged, quotations and paraphrases suitably indicated, and all uses of AI tools have been declared.

Abstract

Transformer neural networks, first proposed in 2017 by *Vaswani et al.* in their paper *'Attention Is All You Need'* [1], have proved to be a revolutionary new neural network architecture in the field of machine learning. Primarily used for natural language processing, the transformer model has consistently outperformed its predecessors, such as recurrent neural networks (RNNs) and long short-term memory networks (LSTMs), in terms of training time and prediction accuracy. The release of ChatGPT in 2022, which is based on the transformer network, exemplified the huge potential of this new network architecture. But could transformer networks revolutionise a wider range of machine learning applications? This report looks into a possible financial application, by investigating whether they can be used to improve stock market predictions.

Contents

Chapter 1

Introduction to Machine Learning and Neural Networks

The field of **artificial intelligence (AI)** has come to the forefront of science and mathematics in recent years. It has been a topic of much discussion and controversy. However, there is no question that it is hugely useful in a wide variety of applications. From ChatGPT to driver-less cars, to cancer diagnosis, there is no doubt that AI has already touched most of our lives. But how is it possible that in these areas, computers have seemingly gained more intelligence than even ourselves, when they do not inherently have the ability to learn?

Whilst computers may not have the natural ability to learn that we humans do, it is possible to teach them via a process called **machine learning**. And once they can learn, they can often do so much faster and more effectively than ourselves. Although machine learning is just one branch of AI, it is an incredibly powerful one, and provides the basis of most of the current applications of AI.

In this report, I will primarily explore the use of **neural networks** for machine learning. Neural networks are a specific type of programming architecture, loosely based on the structure of the human brain, which are able to adapt and learn based on data provided to them. They come in many different shapes and sizes and collectively make up one of the most common methods for machine learning used today.

One recent breakthrough in neural network research came in 2019 with the introduction of **transformer networks** in the paper "*Attention is all you need*" [1]. Transformers are a type of neural network which have revolutionised the field of natural language processing, as their novel structure has led to huge advancements in sequence data processing. Most notably, they were used to create ChatGPT, but they have also been applied in many other areas including speech recognition [2], image processing [3], and even gene analysis [4].

This report focuses on a possible further application of transformer networks in stock market trading. Machine learning is increasingly being used in the financial world due to the ability of neural networks to analyse large datasets, and to spot patterns quicker and more accurately than us. However, despite the recent success of transformer networks in sequence generation applications, there is currently surprisingly little research into the use of transformers in financial applications.

We first discuss the fundamentals of machine learning, before building on that knowledge to explore the evolution of sequence processing neural networks, culminating in transformer networks. After a brief introduction to the stock market in chapter 3, we examine the use of transformer networks in stock market trading, focusing primarily on the two specific transformer based stock market models proposed in [5] and [6].

1.1 Introducing Neural Networks

Let us now go into a bit more detail about neural networks. It is important to understand these structures fully, as they form the basis for the majority of the discussion in this report.

Starting with the most broad overview, neural networks are mathematical structures which aim to transform a set of inputs into an output or set of outputs. Whilst this may currently just sound like any multi-variate function, there is a clever and important difference. The internal parameters of a neural network, which determine the transformation applied to the inputs, are not fixed, and instead are constantly adjusted so that the network can produce a better output. The process of adjusting and optimising these parameters is known as **training** the network.

Training a neural network first involves providing it with some data to learn from. This data has to take the specific form of a set of inputs, paired with a set of (known) corresponding outputs. The network takes in an input, transforms this into a predicted output, compares this to the actual output provided, and then adjusts itself accordingly so that it can give a better prediction next time. This adjustment is how the network 'learns' from each input-output pair.

For example, we might want to build a neural network which can identify handwritten numbers. In this case, we would need a set of images of handwritten numbers as the inputs, and the number shown in each image as the corresponding output. This type of problem is known as a **classification** problem, as the network is trying to assign inputs into different categories, or classes. The other type of problem that we come across in machine learning is **regression**. In this case, the network must predict a specific value from a continuous spectrum as its output. For example, we may want to predict house prices based on a set of variables such as area, size, number of bedrooms etc.

Once the network has been trained, and has learned the appropriate parameters required to transform the inputs into their corresponding output, we can move on to the **inference** stage. This is where we put the neural network to use. We now input values for which we don't know the corresponding outputs. The network will produce a set of predicted outputs which should now be fairly accurate, since the network has learned a transformation which works for similar cases.

This method of learning is incredibly powerful, as it can be adapted for a huge variety of tasks, provided we have a large enough set of data containing known input-output pairs. We do not have to know anything about the underlying transformation which the network is trying to find, nor do we have to tell the program any information about the problem it is trying to solve, other than providing it with the data. Due to the lack of information provided to the network, we refer to this type of learning as **unsupervised learning**.

1.2 Building a Neural Network

Let us now take a look at the inner workings of neural networks, and discuss the method behind the transformation of inputs into outputs. We will start by analysing the most simple neural network configuration, however note that all the ideas discussed in this chapter can be generalised to describe neural networks of any shape and size.

1.2.1 Structure

In the most simple case, a neural network is composed of three layers. Each layer is made up of a series of **nodes** (also known as **neurons**), each containing a value x.

- **Input layer** - the values in this layer form a numerical representation of the network input
- **Hidden layer** - the values in this layer have no meaningful interpretation, but the addition of hidden layers allow networks to model more complex transformations
- **Output layer** - the values in this layer represent the output produced by the network

Each node is connected to every node in the next layer by means of a **weighting** w. The weightings determine how much each node contributes to the value of the connected node in the next layer. Each node also has a **bias** value b, which shifts the value of its associated node. This helps the network to deal with variation and offsets in the input data.

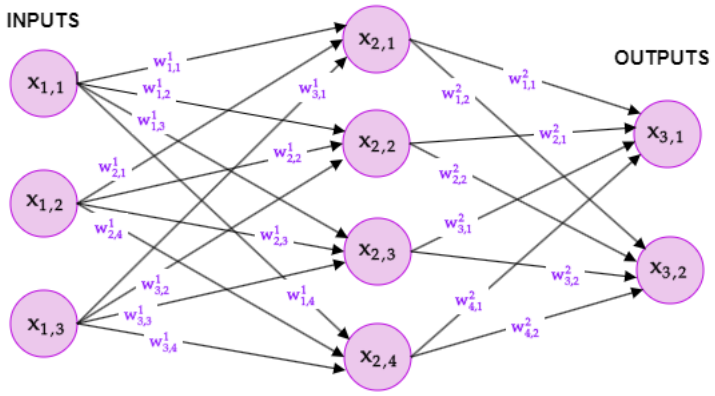

Figure 1.1: *A simple three-layer neural network. Note that this is just an example, the number of nodes in each layer can vary. (figure created by author)*

1.2.2 Forward Pass

Definition 1.1. *The **forward pass** describes the process of calculating the value $x_{i,j}$ of each node from the input nodes to the output nodes.*

To begin this process, we need to provide a set of input values, as well as a set of initial weights and biases.

The input values need to be real numbers which together contain all the necessary information about the current input. For example, if we are inputting words, we might convert each letter to a number between 1 and 26 (each number representing a different letter of the alphabet), and use this sequence of numbers as our input values. Or, if we are inputting images, we might separate the image into pixels, and use a number representation of the colour of each pixel as our input values.

Generally we initialise the weights and biases randomly, as they will be changed later on, so their initial values do not matter. A common way to do this is to randomly sample them from a standard Gaussian.

The values of the nodes in the subsequent layers are then calculated as follows:

$$x_{i,j} = f((\sum_{k=0}^{n_{i-1}} x_{i-1,k} \cdot w_{k,j}^{i-1}) + b_{i,j}) \tag{1.1}$$

where $x_{i,j}$ is the value held by the j^{th} node in the i^{th} layer, n_i is the number of nodes in the i^{th} layer, $w_{k,j}^{i-1}$ is the weight value connecting $x_{i-1,k}$ and $x_{i,j}$, and $b_{i,j}$ is the bias value associated with $x_{i,j}$. The function f is an **activation function** which we will discuss further below. In words, we sum the values of each of the nodes in the previous layer, multiplied by their connecting weight, add the current node's bias value, and then apply an activation function.

When we perform these calculations, we start with the first layer to the right of the input layer. We then move right through the network (i.e. moving forwards through the network from input layer to output layer, this is why we refer to this process as a *forward* pass) computing the values for one layer of nodes at a time. Note that in the network described in section 1.2.1, we only have one hidden layer, however the number of hidden layers can vary. This does not affect the forward pass method though.

1.2.3 Activation Functions

We have just seen how activation functions are used in the forward pass, but we have yet to define what they are, and why they are needed.

Definition 1.2. *An **activation function** is a function applied to the node values in a neural network during the forward pass, to introduce non-linearity to the network.*

This non-linearity is crucial for neural networks to be able to perform successfully, due to the following lemma.

Lemma 1.1. *Without activation functions, a neural network would just be a type of linear regression model.*

Proof. Let $\hat{y}_i$ be the i^{th} output value in an N-layer neural network with no activation functions. We will prove by induction that $\hat{y}_i$ can be written as $\hat{y}_i = (\sum_{k=0}^{n_1} x_{1,k} \cdot W_{k,i}^1) + B_{2,i}$ where $W_{k,i}^1$ and $B_{2,i}$ are constants, therefore showing that each output $\hat{y}_i$ can be written as a linear combination of the input values $x_{1,k}$. The notation in this proof follows the conventions defined in equation (1.1).

First, we define

$$W_{k,i}^{N-1} = w_{k,i}^{N-1} \tag{1.2}$$

and

$$B_{N,i} = b_{N,i} \tag{1.3}$$

so that (base step)

$$\hat{y}_i = (\sum_{l=0}^{n_{N-1}} x_{N-1,l} \cdot W_{l,i}^{N-1}) + B_{N,i} \tag{1.4}$$

We then define

$$W_{l,i}^{N-m} = \sum_{k=0}^{n_{N-m+1}} w_{l,k}^{N-m} \cdot W_{k,i}^{N-m+1} \qquad for\ m \in \{2, 3, ..., N-1\} \tag{1.5}$$

and

$$B_{N-m,i} = (\sum_{k=0}^{n_{N-m}} b_{N-m,k} \cdot W_{k,i}^{N-m}) + B_{N-m+1,i} \qquad for\ m \in \{1, 2, ..., N-2\} \tag{1.6}$$

Now, assume that $\hat{y}_i$ can be written as

$$\hat{y}_i = (\sum_{k=0}^{n_{N-m}} x_{N-m,k} \cdot W_{k,i}^{N-m}) + B_{N-m+1,i} \tag{1.7}$$

then, by substituting $x_{N-m,k} = (\sum_{k=0}^{n_{N-m-1}} x_{N-m-1,k} \cdot w_{k,j}^{N-m-1}) + b_{N-m,j}$ from (1.1), we have

$$\begin{aligned}
\hat{y}_i &= (\sum_{k=0}^{n_{N-m}} ((\sum_{l=0}^{n_{N-m-1}} x_{N-m-1,l} \cdot w_{l,k}^{N-m-1}) + b_{N-m,k}) \cdot W_{k,i}^{N-m}) + B_{N-m+1,i} \\
&= (\sum_{l=0}^{n_{N-m-1}} x_{N-m-1,l} \cdot (\sum_{k=0}^{n_{N-m}} w_{l,k}^{N-m-1} \cdot W_{k,i}^{N-m})) + (\sum_{k=0}^{n_{N-m}} b_{N-m,k} \cdot W_{k,i}^{N-m}) + B_{N-m+1,i} \\
&= (\sum_{l=0}^{n_N-m-1} x_{N-m-1,l} \cdot W_{l,i}^{N-m-1}) + B_{N-m,i} \\
&= (\sum_{k=0}^{n_N-m-1} x_{N-m-1,k} \cdot W_{k,i}^{N-m-1}) + B_{N-m,i}
\end{aligned} \tag{1.8}$$

Therefore, by induction, $\hat{y}_i$ can be written as $\hat{y}_i = (\sum_{k=0}^{n_{N-m}} x_{N-m,k} \cdot W_{k,i}^{N-m}) + B_{N-m+1,i}$ for any $m \in$\{2, 3, ..., N-1\}. Taking $m = N-1$, we can write

$$\hat{y}_i = (\sum_{l=0}^{n_1} x_{1,l} \cdot W_{l,i}^1) + B_{2,i} \tag{1.9}$$

□

Introducing non-linearity allows the network to extract more complex relationships between the inputs and outputs than just linear transformations. It is important to note that all the nodes in any particular layer must use the same activation function, however, different layers within a network may use different activation functions. In practice, we tend to use the same activation function for all hidden layers, but often use a different one for the output layer.

1.2.4 Types of Activation Function

There are many different types of activation function. In fact, in theory, any non-linear piecewise-differentiable monotonically-increasing function can be used as an activation function. However, there are a few specific functions which are commonly used in neural networks.

ReLU

ReLU or Rectified Linear Unit is the function defined by

$$f(x) = max(0, x) \tag{1.10}$$

It is a very commonly used activation function for the hidden layers of a network, as it provides a very simple non-linearity, whilst also deactivating some of the nodes in the network (any nodes which hold a negative value). The simplicity of the operations involved in the function combined with the fact that not all of the nodes are active, make the computations involved in the forward pass quick compared to other activation functions. This makes ReLU a very computationally efficient activation function. The other reason that ReLU is so popular is that it does not suffer from a phenomenon known as the **vanishing gradient problem**. We will discuss what this is, and why it is an issue, later in this chapter (see section 1.4.7).

Two other variations of the ReLU function are also commonly used. The first is the **shifted ReLU**, defined by

$$f(x) = max(0, x - a) \tag{1.11}$$

where a is any real number. This can be used to activate a different range of nodes, as the activation threshold is now a instead of 0. The second is the **leaky ReLU** defined by

$$f(x) = max(0.1x, x) \tag{1.12}$$

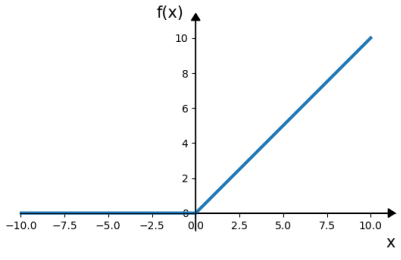

Figure 1.2: *ReLU*

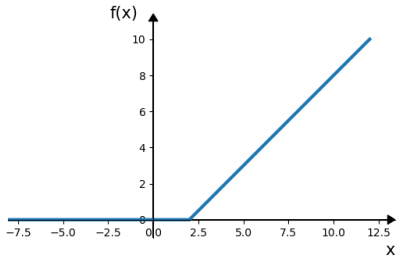

Figure 1.3: *Shifted ReLU*

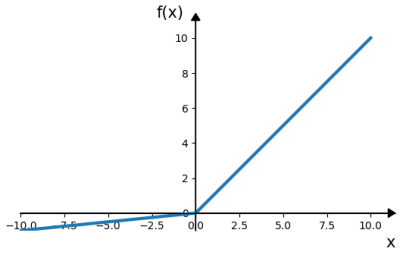

Figure 1.4: *Leaky ReLU*

(*figures created by author*)

Dead neurons - neurons which are never activated - can be an issue with ReLU. Too many dead neurons decrease the model's ability to fit itself to the data fully, as there are a reduced number of nodes contributing to the output values. This problem can be solved by using leaky ReLU instead, as this function does not deactivate any of the nodes.

Sigmoid

The sigmoid function is defined by

$$f(x) = \frac{1}{1 + e^{-x}} \tag{1.13}$$

It transforms all values to a value between 0 and 1. This is often used in binary classification problems where the outputs should be either 0 or 1. The predicted output produced by the network can then be interpreted as the probability of the true output being 1. Sigmoid functions are also sometimes used in the hidden layers, but ReLU tends to be more popular, due to the reasons discussed above.

Tanh

The tanh function is defined by

$$f(x) = tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{1.14}$$

This is very similar to the sigmoid function, with the key difference being that it outputs values between -1 and 1 instead of 0 and 1. Whilst this means it can't be used for binary classification, tanh has the advantage of being centered around 0, which makes learning for subsequent layers more stable. For this reason, it's often preferred over the sigmoid function in the hidden layers of a network. In certain situations, such as in recurrent neural networks (see 2.1), bounded node values can be advantageous. In these cases, tanh may be used in favour of ReLU.

Softmax

The softmax function is defined by

$$f(z_i) = \frac{e^{z_i}}{\sum_j e^{z_j}} \tag{1.15}$$

This converts each output into a probability. This function is used in the output layer of the network, for classification problems where there are a pre-defined number of possible outcomes. It produces a probability for each possible outcome, describing how likely it is that this outcome is the desired output class. In most cases, the class with the highest probability is taken as the output of the network.

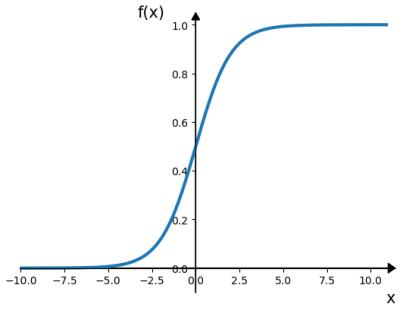

Figure 1.5: *Sigmoid*

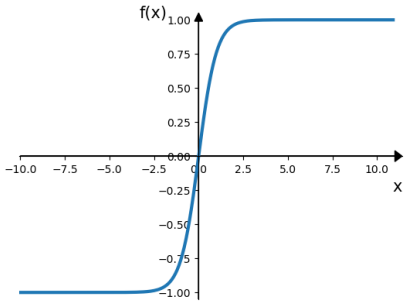

Figure 1.6: *Tanh*

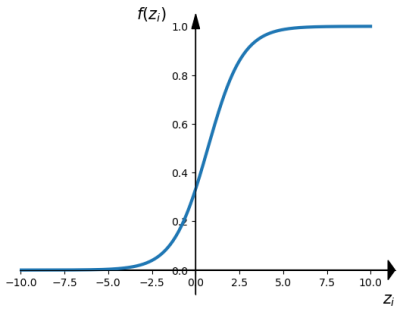

Figure 1.7: *Softmax*

(*figures created by author*)

1.2.5 Loss Functions

The final stage of the forward pass is calculating the **loss** associated with our outputs. The loss is a measure of how far our predicted outputs differ from the expected output. We can theoretically use any function with the following properties as a loss function:

1. The function must be positive for all cases where the outputs are *not* identically equal to the expected output
2. The function must equal zero if the outputs *are* identically equal to the expected output
3. The function must be continuous and piecewise-differentiable
4. The function must increase, as we get further from the minimum. An optional, but desirable extension to this, is that the function should be **convex** - i.e. the gradient $f'(x)$ increases as x increases

There are many such functions which satisfy the above properties, however, much like with the activation functions, there are a set of loss functions which are most commonly used in neural networks.

In the following definitions, we denote the i^{th} predicted output produced by the network as $\hat{y}_i$, and the i^{th} true output from the dataset as y_i.

Mean Square Error (MSE)

The MSE function takes the difference between each output and the corresponding expected output, squares it, and then averages across all the outputs.

$$L_{MSE} = \frac{1}{n}\sum_{1}^{n}(y_i - \hat{y}_i)^2 \tag{1.16}$$

This function is the most commonly used loss function for regression problems, as it is convex, differentiable, and easy to interpret.

Mean Absolute Error (MAE)

The mean absolute error function takes the absolute value of the difference between each output and the corresponding expected output, and then averages across all the outputs.

$$L_{MAE} = \frac{1}{n}\sum_{1}^{n}|y_i - \hat{y}_i| \tag{1.17}$$

This is very similar to the mean square error function, and also used for regression problems. It is often used in cases where there is a lot of noise in the data, as this can cause a significant number of the differences to be quite large, and MAE is less sensitive to large differences than MSE. However, MAE is not convex, and not differentiable everywhere. Therefore, MSE is usually preferred.

Binary Cross-entropy

The binary cross-entropy (BCE) function is the most commonly used loss function for binary classification problems. These problems have only two possible classes of outputs, 0 or 1, with the network producing a probability for each of the classes. BCE is specifically designed for this type of problem, penalising low probabilities for the expected output class.

$$L_{BCE} = -(ylog(\hat{y}) + (1-y)log(1-\hat{y})) \tag{1.18}$$

where $y \in \{0, 1\}$ is the expected class, and $\hat{y} \in (0, 1)$ is the outputted probability of that class.

Categorical Cross-entropy

The categorical cross-entropy (CCE) function is used for classification problems in which there are more than 2 possible output classes. Again, for this type of problem, the outputs of our network will be probabilities for each class. As with BCE, CCE is designed to only consider the expected output class, and penalise low probabilities for this class.

$$L_{CCE} = -\sum_{i} y_i log(\hat{y}_i) \tag{1.19}$$

where $y_i = 1$ if the expected class corresponds to the i^{th} output, and 0 otherwise.

1.3 Training a Neural Network

As briefly discussed in section 1.1, the secret to the success of neural networks lies in the fact that its internal parameters are adjustable. We will now explore the method involved in adjusting these parameters so that the network can model the transformation required to map the inputs to the desired outputs as effectively as possible.

Before training, the output of our network is essentially random due to the random initialisation of our weight and bias parameters. It is these parameters that we now need to adjust so that the network can give better output predictions. The way we do this is by trying to minimise the loss function via a process known as **gradient descent**.

1.3.1 Splitting the dataset

Before we take a closer look at gradient descent, we must briefly digress to discuss an important issue regarding the data we are inputting into the network. Prior to beginning the training process, it is crucial that we split our dataset into three sections:

- The **training set** contains the data we use during the training process. This typically makes up about $60 - 80\%$ of the whole dataset
- The **validation set** contains the data used to optimise the hyperparameters in the network (see section 1.4). This typically makes up about $10 - 20\%$ of the whole dataset
- The **test set** contains the data used to test the performance of the network after training and optimisation. This also makes up about $10 - 20\%$ of the whole dataset

The reason we do this is so that we can keep these three processes completely separate. Re-using any data for more than one of these functions can lead to the illusion that the network is performing better than it actually is, which leads to some undesirable consequences. The reason for this depends on which processes we mix together.

If training and optimising are not kept separate, the network may fit itself too closely to the training data, and in doing so, lose some of its ability to generalise to unseen data in a phenomenon known as **overfitting**. We will discuss this in more detail in section 1.4, but as a broad overview, this leads to the impression during training that the network is performing very well, whilst during inference it does not perform nearly so well.

Inevitably, any network will perform better on training data and validation data, than it will on unseen data, as this is the data that it has been specifically trained or optimised to perform well on. Therefore, if we re-use any of this data in the test set, we will get better results than we would on unseen data. However, this is very dangerous, as we are then given the impression that the network predictions are more accurate than they actually are. Therefore, when using the network to predict outputs for which we do not know the true value, we will be inclined to trust these predictions more than we should.

1.3.2 Gradient Descent

We can now begin to explore the workings of the training process, starting with an important definition.

Definition 1.3. ***Gradient descent*** *is the process of adjusting the parameters in a neural network in such a way that the network outputs are shifted towards the real outputs, resulting in a movement down the gradient of the loss function towards its minimum.*

We can visualise this using a uni-variate MSE function, $f(\hat{y}) = (y - \hat{y})^2$, plotted in figure 1.8.

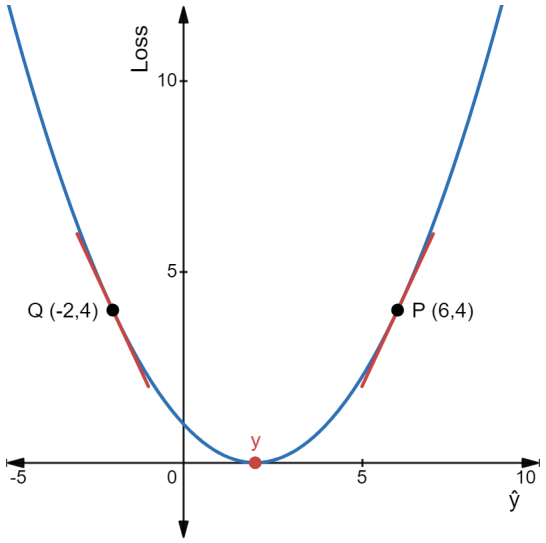

Figure 1.8: *A uni-variate MSE loss function, demonstrating gradient descent (figure created by author)*

If initially our network produces a predicted output of 6, the resulting point on the loss curve is P. The prediction of 6 is too high, and so the objective is to adjust the parameters to reduce $\hat{y}$, and therefore bring the corresponding point on the loss curve down the gradient towards the minimum. Likewise, if our prediction was -2, with the corresponding point Q, the objective would be to increase the prediction, moving the point on the loss curve down the gradient towards the minimum at y.

The next step is to change each of the weights and biases such that the predicted output moves towards the true output. To do this, we consider how a small change in each of the parameters will affect the loss. Note that this is exactly the information held by the partial derivative of the loss function with respect to each of the parameters. Also note that these partial derivatives can always be calculated as all of the operations in the forward pass are at least piecewise differentiable.

Lemma 1.2. *Moving each parameter θ_i by a small amount in the opposite direction to the sign of $\frac{\partial L}{\partial \theta_i}$ will result in movement towards the minimum of the loss curve.*

Whilst this may seem trivial, especially when looking at figure 1.8, we will prove it for the general case, as this is a very important result in the topic of neural networks.

Proof. Let L be the loss function, and θ_i be an adjustable parameter.

$$\frac{\partial L}{\partial \theta_i} = \lim_{h \to 0^+} \frac{L(\theta_1, ..., \theta_i + h, ..., \theta_n) - L(\theta_1, ..., \theta_i, ..., \theta_n)}{h}$$

Therefore,

$$\begin{aligned}
\frac{\partial L}{\partial \theta_i} > 0 &\implies \lim_{h \to 0^+} \frac{L(\theta_1, ..., \theta_i + h, ..., \theta_n) - L(\theta_1, ..., \theta_i, ..., \theta_n)}{h} > 0 \\
&\implies \lim_{h \to 0^+} L(\theta_1, ..., \theta_i + h, ..., \theta_n) - L(\theta_1, ..., \theta_i, ..., \theta_n) > 0 \\
&\implies \lim_{h \to 0^+} L(\theta_1, ..., \theta_i + h, ..., \theta_n) > L(\theta_1, ..., \theta_i, ..., \theta_n)
\end{aligned}$$

Therefore increasing θ_i by a small amount results in an increase in L.
Likewise,

$$\begin{aligned}
\frac{\partial L}{\partial \theta_i} > 0 &\implies \lim_{h \to 0^-} \frac{L(\theta_1, ..., \theta_i + h, ..., \theta_n) - L(\theta_1, ..., \theta_i, ..., \theta_n)}{h} > 0 \\
&\implies \lim_{k \to 0^+} \frac{L(\theta_1, ..., \theta_i - k, ..., \theta_n) - L(\theta_1, ..., \theta_i, ..., \theta_n)}{-k} > 0 \quad for\ k = -h \\
&\implies \lim_{k \to 0^+} \frac{L(\theta_1, ..., \theta_i, ..., \theta_n) - L(\theta_1, ..., \theta_i - k, ..., \theta_n)}{k} > 0 \\
&\implies \lim_{k \to 0^+} L(\theta_1, ..., \theta_i, ..., \theta_n) - L(\theta_1, ..., \theta_i - k, ..., \theta_n) > 0 \\
&\implies L(\theta_1, ..., \theta_i, ..., \theta_n) > \lim_{k \to 0^+} L(\theta_1, ..., \theta_i - k, ..., \theta_n)
\end{aligned}$$

Therefore decreasing θ_i by a small amount results in a decrease in L.
By symmetry, if $\frac{\partial L}{\partial \theta_i} < 0$, increasing θ_i by a small amount results in a decrease in L.
This, combined with the condition on loss functions that they must increase the further we get from the minimum, proves the result.

□

We therefore adjust each of the parameters θ_i via the following transformation:

$$\theta_i \to \theta_i - \alpha \frac{\partial L}{\partial \theta_i}(\boldsymbol{\theta}) \tag{1.20}$$

where $\boldsymbol{\theta}$ represents the current values of all the network parameters. Here, α is a **hyperparameter**, called the **learning rate**, which we are free to choose ourselves. This determines the size of the jumps as we move down the gradient. We will come back to this later on in this chapter.

Definition 1.4. *A **hyperparameter** is a parameter within a neural network which does not get adjusted automatically during training, but instead is manually chosen by the programmer before initialising the training process.*

There are two different methods we can use when completing gradient descent: standard gradient descent and stochastic gradient descent. These two methods differ as to when the updates to the parameters are made.

Definition 1.5. *The **standard gradient descent algorithm (GD)**, usually just known as **gradient descent**, is a method of gradient descent which involves passing the entire training dataset through the network before updating the network parameters. The updates are then made using the average partial derivative values over the whole training set.*

Definition 1.6. *The **stochastic gradient descent algorithm (SGD)** is a method of gradient descent which involves updating the network parameters after each individual training datapoint is passed through the network.*

There is also a variation of SGD which is commonly used.

Definition 1.7. *The **minibatch stochastic gradient descent algorithm** also known as the **batch stochastic gradient descent algorithm** is a method of gradient descent which involves splitting the training dataset into small **batches** each consisting of a subset of the training data. The network parameters are then updated after each batch has been passed through the network.*

It is important to note that with each method, the full training dataset is passed through the network many times to allow the parameters to fully adjust themselves towards the optimum values.

There are advantages and disadvantages to using each method. GD leads to a more stable gradient descent due to taking an average of many datapoints before each update. However, too much stability can cause issues. Due to the loss function depending on a large number of variables, it is likely that there will be local minima within the loss function. It requires some random noise in the gradient descent to escape these local minima. GD has very little random noise, and so is liable to get caught in these local minima. Another issue with GD is that it requires more memory than the other methods, as it has to store values for every datapoint in the training set before using them for the updates. This makes GD the least computationally efficient gradient descent algorithm.

SGD introduces far more randomness into the gradient descent than GD. This is because it updates the parameters individually for every datapoint, even those which may be considered as outliers or anomalies. This causes a large amount of noise in the updates, which helps the algorithm to escape from local minima. However, this also means that SGD often never converges fully to the global minimum, and instead oscillates around it.

Batch SGD provides a compromise between the two methods, converging more fully than SGD, but using less memory than GD, and also introducing enough noise to escape from most local minima. This is the method which we will use for the rest of this report.

1.3.3 Backwards Pass

We have now introduced the gradient descent method, which is used to update our network parameters in such a way that we converge to the global minimum of our loss function. However, we have skipped over one important point. So far, we have assumed that we can easily calculate the partial derivatives of the loss function with respect to each of the network parameters. However, in practice, they can be very difficult and time consuming to compute directly, especially for large, complex networks. In this section, we introduce a method for calculating these partial derivatives in stages, which greatly speeds up the process.

Definition 1.8. *The **backwards pass** is a method for computing the partial derivatives of the loss function with respect to each of the parameters in a neural network. It involves the following process:*

1. *Calculate $\frac{\partial L}{\partial \hat{y}_j}(\hat{y}_j)$ for each of the network outputs $\hat{y}_i$.*
2. *Using the chain rule, moving recursively backwards through the layers of the network, calculate*

$$\frac{\partial L}{\partial x_{i,j}}(\boldsymbol{\theta}) = \sum_{k=0}^{n_{i+1}} \frac{\partial L}{\partial x_{i+1,k}}(\boldsymbol{\theta}) \cdot \frac{\partial x_{i+1,k}}{\partial x_{i,j}}(\boldsymbol{\theta}) \tag{1.21}$$

for each of the nodes $x_{i,j}$ in the network. (We take $x_{N,i} = \hat{y}_i$ for an N layer network).

3. Again using the chain rule, calculate

$$\frac{\partial L}{\partial w^i_{k,j}}(\boldsymbol{\theta}) = \frac{\partial L}{\partial x_{i+1,j}}(\boldsymbol{\theta}) \cdot \frac{\partial x_{i+1,j}}{\partial w^i_{k,j}}(\boldsymbol{\theta}) \tag{1.22}$$

$$\frac{\partial L}{\partial b_{i,j}}(\boldsymbol{\theta}) = \frac{\partial L}{\partial x_{i,j}}(\boldsymbol{\theta}) \cdot \frac{\partial x_{i,j}}{\partial b_{i,j}}(\boldsymbol{\theta}) \tag{1.23}$$

for each of the parameters $w^i_{k,j}$ and $b_{i,j}$ in the network.

Using this method, we can calculate the partial derivatives whilst only requiring knowledge of the rules to differentiate equation (1.1) for the activation functions used, and the loss function. This makes it much easier to teach our program how to calculate the partial derivatives for all of the network parameters. It is also much more computationally efficient to calculate the derivatives in layers, as we do using the backwards pass method, than to try to calculate them all directly from the loss function.

By teaching our program this method, we can automate the entire gradient descent algorithm, letting the computer calculate each of the partial derivatives, and then updating each of the parameters using (1.20). We can then instruct the program to repeat this process for a pre-determined number of iterations of the whole training dataset. At this point, we have taught the network to train itself, and thus our machine can now learn.

1.3.4 Inference

Once the network is trained, we can then use it to make predictions using input data for which we don't know what the output should be. To do this, we simply input the data as before, and complete a forward pass to produce predicted outputs. This time there is no need to calculate the loss, or complete a backwards pass, as we are no longer adjusting the parameters. Instead we are using the trained parameters to produce informed output predictions. This process is known as **inference**. This is the main aim of neural networks, that they can make accurate predictions based on the patterns they have established during the training phase.

That concludes the basic outline of how a neural network works. But there are still many different ways we can fine tune the network to improve training speed and prediction accuracy.

1.4 Optimisation of a Neural Network

We have already briefly introduced **hyperparameters**, in section 1.3.2. However, so far, we have only met one example, the learning rate, and we have not yet discussed how to choose the values for these parameters. In this section, we will introduce several more examples of hyperparameters, and examine their importance, as well as how to choose the optimum values for them.

Unfortunately, for the most part, there are no mathematical formulas, or any sort of concrete guidance to help us decide how to choose these values, so we fall back on trial and error. It is also important to note that changing any of the hyperparameters may affect the optimal value of the other hyperparameters. This makes hyperparameter tuning a very delicate and time-consuming process, and so it is often the case of finding values which are "good enough" rather than perfect.

To choose values for our hyperparameters, we need to test the performance of the network using a range of different values. As discussed in section 1.3.1, it is inadvisable to use the training set for this testing process, and so, for everything in this section, we will be using the validation dataset as input data.

1.4.1 Overfitting

Before we discuss tuning of the hyperparameters any further, let us first take a look in more detail at the phenomenon of overfitting, as this is one of the key factors determining our choices for most of the hyperparameters.

As a reminder, overfitting occurs when our network fits itself too closely to the training data, and in doing so, loses some of its ability to generalise to unseen data. We can observe this in action in figures 1.9 and 1.10, where we can

see that fitting our model too closely to the training data has a disastrous impact on predictions with new unseen data.

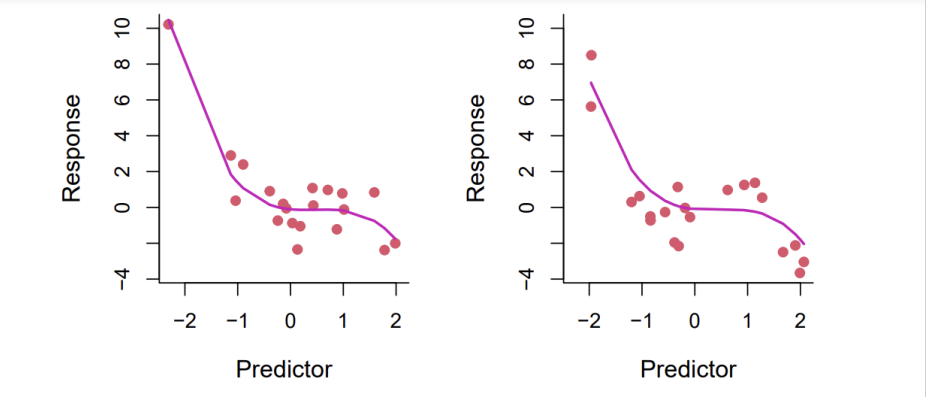

Figure 1.9: *Input data fitted correctly*

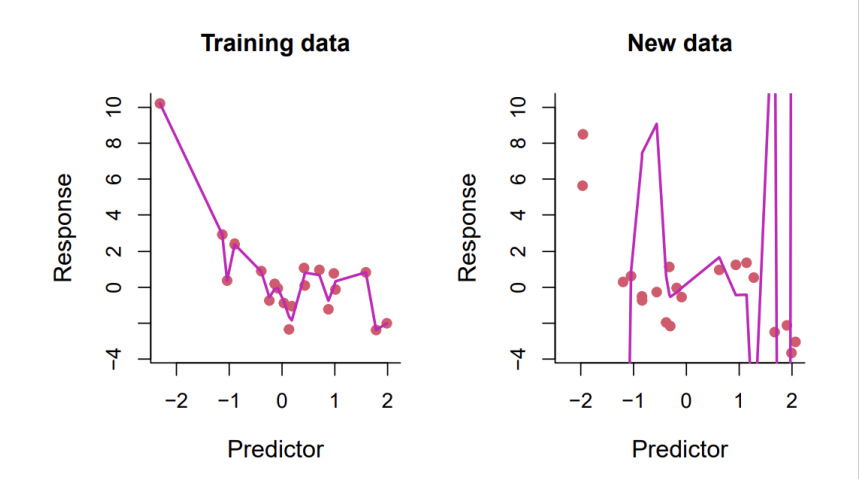

Figure 1.10: *Input data over-fitted*
(figures taken from Durham University MATH3431_2023 course notes [7, Chapter 8, pg. 158])

By analysing the loss values during training, we can tell when overfitting starts to occur. As the network is constantly adjusting itself to the training data, training loss will always decrease the more we train the network. However once the network starts to overfit, its performance on unseen data will start to decrease, as it loses its ability to generalise. We can test this by analysing loss values using the validation dataset as input. Note that during this testing phase we do not update the network parameters, as we do not want the network to fit itself to the validation set.

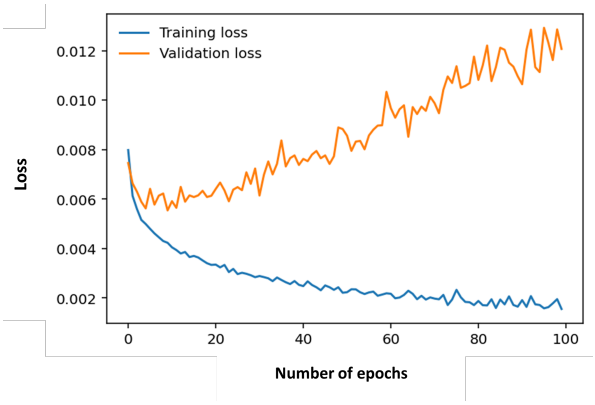

Figure 1.11: *Loss values during training when using either the training set or validation set as input data - a demonstration of overfitting*
(figure created by author)

Definition 1.9. *An **epoch** describes passing a full iteration of the training dataset through a neural network*

Here, we can see that after about 15 epochs, the validation losses start to increase. This is a sure sign of overfitting.

Overfitting can be a major issue when training a neural network, as we want to train the network as fully as possible, without falling into the trap of overfitting. There are two main ways to get around this issue:

1. Increase the amount of training data - this results in the network taking longer to overfit itself to the network, whilst not increasing the time taken to train the network. Therefore the model is less likely to reach the point of overfitting before it is has finished training

2. If no more training data is available, use dropout on the data (see section 1.4.2 below)

Early-stopping - stopping the training before overfitting starts to occur - can also be effective. However, without further preventative measures, it is often the case that the model starts overfitting before it has fully converged. This occurs when the model starts fitting itself to the noise in the training data rather than the underlying patterns. In these cases, early-stopping will not be an effective method to reduce the effects of overfitting, as the model will never get a chance to fully converge.

1.4.2 Dropout

Dropout is a technique used to counteract over-fitting by hiding some of the training data during each epoch. This prevents the network from fitting itself too closely to the training data, as it is trained on a slightly different subset of the training data during each iteration, so cannot fully fit itself to the entire training set.

The dropout value, determining how much of the training data is hidden during each iteration, is a hyperparameter for us to choose. A dropout value of $d/100$, means that $d\%$ of the data is hidden each time. Below, we have two graphs produced in the same way as figure 1.11, but this time using the dropout technique.

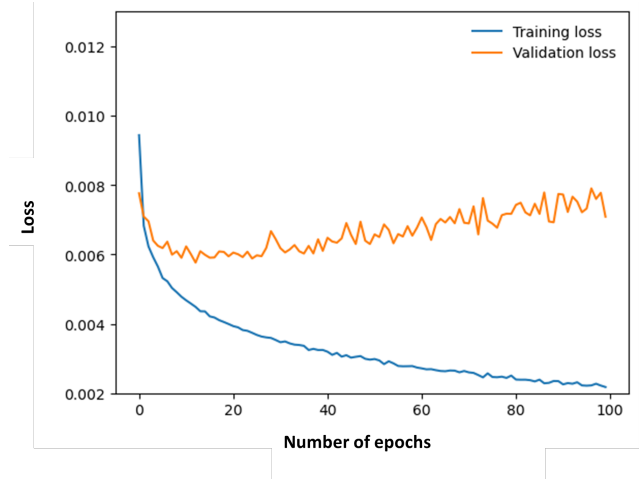

Figure 1.12: *Training and validation losses using a 0.2 dropout value*

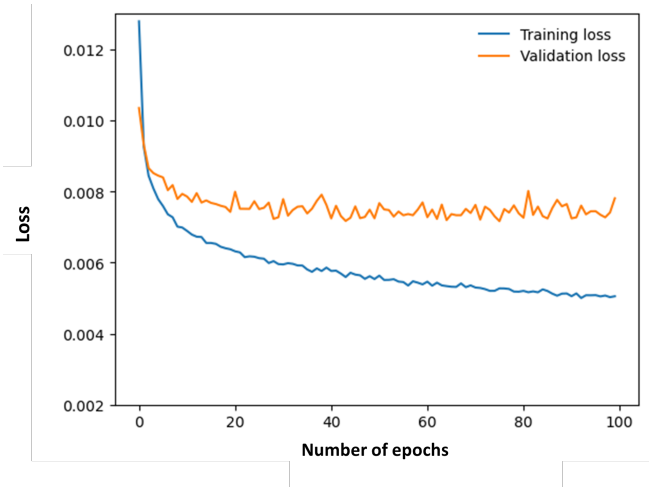

Figure 1.13: *Training and validation losses using a 0.5 dropout value*

(*figures created by author*)

We can see from the graphs, that the overfitting is significantly reduced compared to figure 1.11. With a dropout value of 0.2, we still see some overfitting, whereas with 0.5, we don't see a general increase in validation losses at all. However, we can see in figure 1.13, neither the training losses nor the validation losses go as low as those in figures 1.11 and 1.12. This suggests that a value of 0.5 may be too high for this particular neural network, prohibiting its ability to fit itself properly to the data provided.

The choice of dropout value mainly depends on how much training data we have available. A large training set means less dropout is required, and vice versa. Other factors such as the complexity of the data, and the complexity of the network also affect which value will be optimal, and so it requires careful analysis of results using different values to find the best one. Values between 0.1 and 0.5 are the most commonly used.

1.4.3 Number of Layers and Nodes

To choose values for the number of layers and the number of nodes in each layer, we need to consider an important trade-off. The higher the number of nodes in our network, the more information the network can extract from the input data, and with a higher number of layers, our network will be able to model more complex relationships between the inputs and outputs. However, a higher number of layers and nodes increases the risk of overfitting, whilst also resulting in an increase in computational cost and training time.

To achieve maximum prediction accuracy, we need to find a suitably high number of nodes and layers such that all the necessary information is extracted from the inputs and used to full effect, without incurring any over-fitting.

1.4.4 Learning rate

We introduced learning rate earlier in this chapter as α in equation (1.20). As previously mentioned, the learning rate determines the size of the jumps we make down the gradient of the loss function during gradient descent. When choosing a learning rate, there are two important factors to consider:

1. speed of convergence
2. success of convergence

A higher learning rate will descend the gradient of the loss function quicker and will therefore usually result in quicker convergence, giving us a shorter training time. However, a higher learning rate is also less likely to result in a successful convergence. A learning rate which is too high can result in skipping over the minimum of the loss function and oscillation around it rather than convergence.

In the cases where there is more than one minimum in the loss function, it is very hard to ensure that gradient descent results in convergence at the global minimum rather than at a local minimum. However, a learning rate that is too small is much more likely to cause us to get caught at a local minimum.

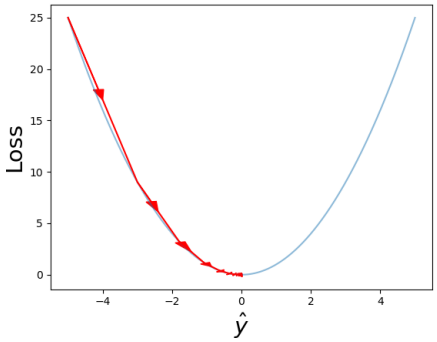

Figure 1.14: *Desirable gradient descent converging at the global minimum*

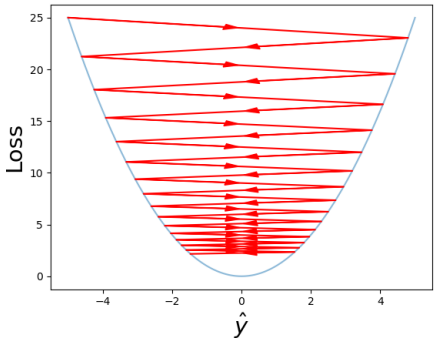

Figure 1.15: *Learning rate too high, resulting in oscillation around the minimum*

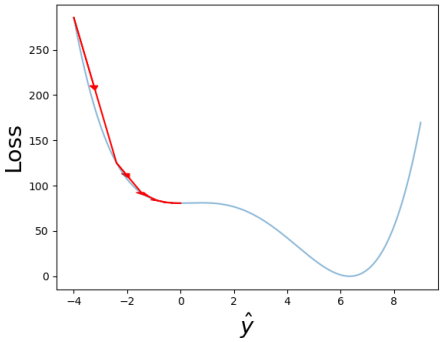

Figure 1.16: *Learning rate too low, resulting in convergence at a different local minimum*

(*figures created by author*)

For most networks, the optimal learning rate is between 0.001 and 1. Using trial and improvement, we can ascertain a good learning rate, by choosing the value which gives us the lowest convergence using the validation data as input. We use this same method to choose values for each of the other hyperparameters as well.

1.4.5 Number of Epochs

The number of epochs is easier to set than the other hyperparameters, as we are not bound to stick with the number of epochs we initially set. We can stop training at any time, if we determine the network to be fully trained, and we can also continue training if we determine that the network is not yet fully trained, even if we've completed the set number of epochs.

To determine when a network is fully trained, we can analyse the average loss value on the validation dataset for each epoch. We then use the method of early-stopping to ensure that the training process results in the minimum possible validation loss. Once the validation loss values start increasing, we stop training, as we know over-fitting is starting to occur.

1.4.6 Batches

As previously mentioned in section 1.3.2, batch SGD offers a compromise between SGD and GD, in terms of memory usage and stability of convergence. Using batches also has the advantage of not needing to load the entire dataset at once, therefore saving more memory by loading only one batch at a time. We are also able to speed up the training process in comparison to SGD, as we can parallelise the training of each batch, i.e. simulataneously pass the entire batch through the network each time. This also allows hardware accelerators such as GPU's to be used effectively to further reduce training time. Note, it is possible to pass the input data through the network in batches, even when using GD. This allows us to capitalise on the increased computational efficiency resulting from parrallelisation, whilst still only updating parameters after a full epoch.

The batch size in batch SGD should be chosen to strike the right balance between the properties of GD and those of SGD. It may not always be obvious which batch size is optimum, but apart from the increased possibility of getting stuck in a local minimum with larger batch sizes, the size of the batch should not significantly affect the end result. Batch sizes of 16, 32 or 64 are most commonly used.

1.4.7 Normalisation

Whilst normalisation in neural networks does not involve any hyperparameters, it is still an important topic to discuss in this section. Normalisation of the node values helps to improve convergence during training, and is therefore a very commonly used practice.

Definition 1.10. ***Normalisation*** *is the process of scaling a set of values to either fit within a certain range, or fit to a standard distribution.*

The most commonly used example of normalisation is known as **Z-score normalisation**. This method scales the values such that the resulting set has a mean of 0 and a standard deviation of 1, effectively fitting the dataset to the standard Gaussian distribution. This is achieved via the following transformation:

$$x_i \to \frac{x_i - \mu}{\sigma} \tag{1.24}$$

where μ and σ are the mean and standard deviation of the original dataset. In neural networks, Z-score normalisation is commonly used on specific sets of node values due to two main benefits:

- It scales the node values so that they are all within a similar range. When the input values within the training set are on different scales, this can have several problems including extra sensitivity to initial weight values, and reduced speed and success of the convergence. Normalisation helps to mitigate these issues
- It helps to avoid the vanishing gradient problem

There are two common ways to group the nodes for normalisation. The first is known as **batch normalisation**, where we calculate the mean and standard deviation individually for each node in the network, by grouping the different values over a whole batch of data. The second method is **layer normalisation**, where we group the node values by layer to calculate a mean and standard deviation over each layer.

1.4.8 The vanishing gradient problem

We can now define what we mean by the vanishing gradient problem.

Definition 1.11. *The **vanishing gradient problem** describes an issue during back propagation in which the gradients get smaller with each layer. With enough layers in the network, this can cause the gradients in some of the layers to effectively vanish.*

If a large number of nodes have very small gradients with respect to the loss function, this means that their values will only be adjusted by very small amounts during each iteration of gradient descent. This will often lead to very slow training. In some cases, it can also destabilise the training process, as these nodes may continue to hold values far from their optimal values, whilst other nodes converge normally. This can cause offset output values, and may result in convergence at a different minimum than the desired one.

The vanishing gradient problem can occur when using either the sigmoid or tanh function as our hidden layer activation function. From figures 1.5 and 1.6, we can see that the gradients of both the sigmoid and tanh functions become very small around the extremes. By analysing the gradient calculations within the back propagation process, we can understand why this becomes a problem.

Looking at equation (1.21), we first discuss $\frac{\partial x_{i+1,k}}{\partial x_{i,j}}(\boldsymbol{\theta})$. Evaluating this using equation (1.1), we get:

$$\frac{\partial x_{i+1,k}}{\partial x_{i,j}}(\boldsymbol{\theta}) = f'((\sum_{l=0}^{n_i} x_{i,l} \cdot w^i_{l,k}) + b_{i+1,k}) \cdot w^i_{j,k} \tag{1.25}$$

Therefore, if $(\sum_{l=0}^{n_i} x_{i,l} \cdot w^i_{l,k}) + b_{i+1,k}$ falls in the extreme ends of the sigmoid or tanh functions, then this value will be very small. This is likely to happen if the input values are very spread out, or on different scales.

Looking again at equation (1.21), we can see that if most of the $\frac{\partial x_{i+1,k}}{\partial x_{i,j}}(\boldsymbol{\theta})$ are very small, then $\frac{\partial L}{\partial x_{i,j}}(\boldsymbol{\theta})$ is likely to be smaller than any of the $\frac{\partial L}{\partial x_{i+1,k}}(\boldsymbol{\theta})$ values. This becomes a recurring problem, and so the gradients in general become smaller and smaller as we move back through the layers of our network. This is a significant problem because if the gradients become too small, the parameters associated with them will only change by tiny amounts, if at all, during each adjustment. This causes conversion to be incredibly slow, and can prevent us from converging to the minimum of the loss function at all.

Normalisation of node values helps to avoid this problem, as it ensures that the node values are both centered around 0, and squeezed towards the center of the tanh and sigmoid functions. This means that very few node values end up in the extreme ends of the activation functions, and so we do not end up with a vanishing gradient problem.

Note that the ReLU activation function does not suffer from this problem, as for all live nodes, the gradient of ReLU is 1.

Chapter 2

Sequence Processing Neural Networks

So far, the neural networks we have been looking at fall under the class of **feed-forward neural networks**. These networks take in a set of fixed-size inputs, and via a single-directional path, process these inputs forward through the layers of the network to produce an individual fixed-size output for each input. In other words, each output is produced directly from an individual input.

In 1982, John Hopfield had an innovative idea to create feedback loops within a neural network [8], and in doing so, became the father of a new type of network, the **recurrent neural network (RNN)**. This new network architecture was able to deal with sequence data as its input, and was able to capture dependencies between the individual datapoints within the sequence. Now, outputs were generated not only depending on the individual datapoints, but also on their relationship with each of the other datapoints in their input sequence. RNNs were also novel in their ability to take in a variable input length. These new features were revolutionary for a whole range of machine learning fields including natural language processing, generative AI, and time series analysis.

In this chapter we will examine the evolution of sequence processing neural networks, discussing RNNs, **long short-term memory (LSTM) networks**, and finally introducing the **transformer architecture** that will form the basis of the remainder of this report.

2.1 Recurrent Neural Networks

The RNN was the first model of its kind to successfully use feedback loops to combine datapoints in the network, and in doing so establish relationships between sequential input datapoints. It achieved this via a simple recursion method as shown in figure 2.1.

2.1.1 Input Embedding

Before we take a look at the structure of RNNs, it is first necessary to briefly discuss input embedding. In all sequence processing models, the datapoints from the original input sequence will first be processed through an **embedding layer**, converting each datapoint into an n-dimensional vector. This means that each datapoint is now represented by n numbers, each number containing slightly different information about their corresponding datapoint. This allows our network to extract all the necessary information from each datapoint, whilst also facilitating comparison between them.

The n-dimensional vector can be thought of as coordinates in the n-dimensional **embedding space**. Vectors closer together in the embedding space represent datapoints which are more closely related. The value of n is a hyperparameter to be chosen with commonly used values including 16, 32, 64 and 128 depending on how complex the input data is.

The embedding layer usually takes the form of a simple linear layer, or in some cases, when dealing with more complex data, a full feed-forward network. The parameters in the embedding layer will be trained along with the rest of the network, so that the network can learn the appropriate transformation such that the distance between

datapoints in the embedding space reflects the strength of the relationships between them.

For numerical input data, the data can be immediately passed through the embedding layer without the need for any pre-processing. However, for categorical data such as words or musical notes, we must first convert the data into a numerical representation before passing it through the embedding layer. This involves first establishing a **vocabulary**, which is a full list of all the possible categories, for example every letter in the alphabet. We then map each category in the vocabulary to a unique number, and use this mapping to convert our input data into numerical form.

2.1.2 Structure of RNNs

We are now ready to introduce the structure of RNNs, and establish how the structure leads to effective sequence handling.

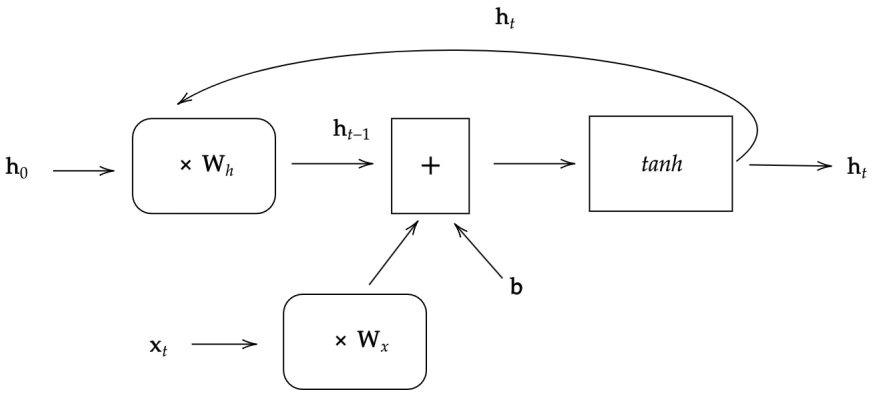

Figure 2.1: *A recurrent neural network*
(figure created by author, based on the diagram found in [9, Section 21.1.3, pg. 506])

Each iteration of the block displayed in figure 2.1 takes two inputs, a **hidden state** vector $\mathbf{h}_{t-1}$ and an embedded datapoint $\mathbf{x}_t$ from the input data sequence. To begin the forward pass, $\mathbf{h}_0 = 0$ and $\mathbf{x}_1$ are inputted. They are then multiplied by the weights $\mathbf{W}_h$ and $\mathbf{W}_x$ respectively before being combined via addition. At this point a bias value $\mathbf{b}$ may also be added. The tanh activation function (or another suitable activation function) is then applied, producing the next hidden state $\mathbf{h}_1$. This is then used as the next hidden state input, and combined with $\mathbf{x}_2$ in the same process as that which we've just described for $\mathbf{h}_0$ and $\mathbf{x}_1$, as well as being outputted from the network. This process is repeated, inputting the next datapoint from the sequence with each subsequent iteration, until we reach the end of the input data sequence. In summary,

$$\mathbf{h}_t = tanh(\mathbf{W}_x\mathbf{x}_t + \mathbf{W}_h\mathbf{h}_{t-1} + \mathbf{b}) \tag{2.1}$$

[9, equation (21.10), pg. 506]. Each output $\mathbf{h}_t$ contains information from all the datapoints up to and including $\mathbf{x}_t$ in the input sequence, rather than just a single datapoint. This makes RNNs far more useful than feed-forward networks for any task requiring sequence analysis.

As the structure of the RNN is iterative, and it applies the same weights during each iteration, the network can take in variable input lengths, simply terminating when the end of the sequence is reached. However, there is a very important drawback to this, which significantly impacts the effectiveness of RNNs in dealing with long sequences, and their ability to capture long term dependencies. This drawback manifests itself in the vanishing and exploding gradient problems.

2.1.3 The vanishing and exploding gradient problems

We have met the vanishing gradient problem before in chapter 1, however the problem is slightly different when it comes to RNNs. The issue again comes during the back propagation stage of training, but this time it affects how the different datapoints in the input sequence are able to interact with each other.

Before we proceed, it is worth proving a result which we will need to use in this section.

Lemma 2.1. *For a square matrix $\boldsymbol{M} \in \mathbb{R}^{nxn}$, a vector $\boldsymbol{v} \in \mathbb{R}^n$, and a differentiable element-wise function f,*

$$\frac{d}{d\boldsymbol{v}}(f(\boldsymbol{M}\boldsymbol{v})) = diag(f'(\boldsymbol{M}\boldsymbol{v})) \cdot \boldsymbol{M} \tag{2.2}$$

where $diag(\cdot)$ makes a diagonal matrix from a vector. (The importance of this stems from [10, Section 1.1, equation (5)].

Proof. Let $\boldsymbol{g}(\boldsymbol{v}) = f(\boldsymbol{M}\boldsymbol{v})$. Then $\frac{d\boldsymbol{g}}{d\boldsymbol{v}}$ is the Jacobian matrix,

$$\frac{d\boldsymbol{g}}{d\boldsymbol{v}} = \begin{bmatrix} \frac{dg_1}{dv_1} & \frac{dg_1}{dv_2} & \cdots & \frac{dg_1}{dv_n} \\ \frac{dg_2}{dv_1} & \frac{dg_2}{dv_2} & \cdots & \frac{dg_2}{dv_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{dg_n}{dv_1} & \frac{dg_n}{dv_n} & \cdots & \frac{dg_n}{dv_n} \end{bmatrix}$$

Let $\mathbf{M}$ be formed of the row vectors $\mathbf{m}_1, \mathbf{m}_2, ..., \mathbf{m}_n$. Then,

$$\frac{dg_i}{dv_j} = \frac{d}{dv_j}(f(\mathbf{m}_i\mathbf{v})) = f'(\mathbf{m}_i\mathbf{v}) \cdot \mathbf{M}_{i,j}$$

Therefore,

$$\frac{d\boldsymbol{g}}{d\boldsymbol{v}} = \begin{bmatrix} f'(\mathbf{m}_1\mathbf{v}) & 0 & \cdots & 0 \\ 0 & f'(\mathbf{m}_2\mathbf{v}) & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & f'(\mathbf{m}_n\mathbf{v}) \end{bmatrix} \cdot \mathbf{M} \;=\; diag(f'(\boldsymbol{M}\boldsymbol{v})) \cdot \boldsymbol{M}$$

□

We now have the tools to fully explore the vanishing and exploding gradient problems with RNNs. 10 For the purposes of this section, we will assume that the RNN is configured so that

$$\mathbf{h}_t = f(\mathbf{W}_h \cdot \mathbf{h}_{t-1} + \mathbf{W}_x \cdot \mathbf{x}_t + \mathbf{b}) \tag{2.3}$$

where f represents a general element-wise activation function. The following argument was inspired by the arguments in [10, Sections 1.1,2.1], [11, *'The Case of the Vanishing Gradients'*] and [12, Section 1].

For any $0 \leq s < t \leq T$, using the chain rule, we have

$$\frac{\partial \mathbf{h}_t}{\partial \mathbf{h}_s} = \prod_{i=s+1}^{t} \frac{\partial \mathbf{h}_i}{\partial \mathbf{h}_{i-1}} \tag{2.4}$$

and from (2.3) and lemma 2.1 we have

$$\frac{\partial \mathbf{h}_i}{\partial \mathbf{h}_{i-1}} = diag(f'(\mathbf{W}_h \cdot \mathbf{h}_{i-1} + \mathbf{W}_x \cdot \mathbf{x}_i + \mathbf{b})) \cdot \mathbf{W}_h \tag{2.5}$$

[11]. Therefore,

$$\frac{\partial \mathbf{h}_t}{\partial \mathbf{h}_s} = \prod_{i=s+1}^{t} diag(f'(\mathbf{W}_h \cdot \mathbf{h}_{i-1} + \mathbf{W}_x \cdot \mathbf{x}_i + \mathbf{b})) \cdot \mathbf{W}_h \tag{2.6}$$

and so,

$$
\begin{aligned}
||\frac{\partial \mathbf{h}_t}{\partial \mathbf{h}_s}|| &= || \prod_{i=s+1}^{t} diag(f'(\mathbf{W}_h \cdot \mathbf{h}_{i-1} + \mathbf{W}_x \cdot \mathbf{x}_i + \mathbf{b})) \cdot \mathbf{W}_h|| \\
&\leq \prod_{i=s+1}^{t} ||diag(f'(\mathbf{W}_h \cdot \mathbf{h}_{i-1} + \mathbf{W}_x \cdot \mathbf{x}_i + \mathbf{b})) \cdot \mathbf{W}_h|| \\
&\leq \prod_{i=s+1}^{t} ||f'(\mathbf{W}_h \cdot \mathbf{h}_{i-1} + \mathbf{W}_x \cdot \mathbf{x}_i + \mathbf{b})|| \cdot ||\mathbf{W}_h|| \\
&= ||\mathbf{W}_h||^{t-s-1} \prod_{i=s+1}^{t} ||f'(\mathbf{W}_h \cdot \mathbf{h}_{i-1} + \mathbf{W}_x \cdot \mathbf{x}_i + \mathbf{b})|| \qquad (2.7)
\end{aligned}
$$

where $||\cdot||$ represents the l_2 norm. This means that for any of the commonly used activation functions, where $||f'(\cdot)||$ is bounded by 1, if $||\mathbf{W}_h|| < 1$ then as t gets further away from s, the size of the gradient $\frac{\partial \mathbf{h}_t}{\partial \mathbf{h}_s}$ will decrease by at least a factor of $1/||\mathbf{W}_h||$ with each time step. This means that changes in $\mathbf{h}_s$ have an exponentially diminishing effect on the hidden states moving forward from $\mathbf{h}_s$ in the sequence. This is the **vanishing gradient problem** in RNNs.

The result is that inputs too far behind the current time step will have little or no influence on the current hidden state. This means that long term dependencies between the datapoints in the input sequence are lost.

The **exploding gradient problem** occurs in the opposite case, in which $||\mathbf{W}_h||$ is too large. In this scenario, the threshold for $||\mathbf{W}_h||$ is not so well defined, due to the fact that with most activation functions, $||f'(\cdot)||$ can achieve very small values, which can cancel out the expanding value of $||\mathbf{W}_h||^{t-s-1}$. However, if $1/||f'(\mathbf{W}_h \cdot \mathbf{h}_{t-1} + \mathbf{W}_x \cdot \mathbf{x}_t + \mathbf{b}||)$ is consistently smaller than $||\mathbf{W}_h||$, then we will fall victim to the exploding gradient problem.

This has the opposite effect to the vanishing gradient problem, resulting in increasing gradients as we go back in time. This is a very undesirable effect as it means that each hidden state has an exponentially *increasing* dependence on the hidden states going further back in the sequence. At best, this will cause a high level of instability during training, but in most cases, this will cause some very strange and likely incorrect results.

It is very difficult to control the size of $||\mathbf{W}_h||$ enough to avoid both the vanishing and exploding gradient problems, and so generally RNNs are only suitable for dealing with short input sequences so that the gradients have a limited distance to propagate through the network.

2.2 LSTM Networks

In 1997, an advancement was made in the field of sequence processing neural networks [13, *'Neural Networks'*, pg. 17] when Sepp Hochreiter and Jürgen Schmidhuber first proposed the LSTM network [14], as an improvement on the RNN structure. With LSTM networks, they were able to remove the problem of vanishing or exploding gradients, allowing them to capture long term dependencies far more succesfully than was possible with RNNs. In this section, we will show how they were able to achieve these improvements.

2.2.1 Structure

The structure of LSTM networks is similar to that of RNNs, in that it contains feedback loops such that once each input has been processed, it is fed back into the network as a hidden state, to be combined with the next input and then re-processed. However, there are two major differences.

The first is in how each input is processed. Each iteration in an RNN is effectively just a single layer feed-forward network with two inputs, $\mathbf{x}_t$ and h_{t-1}. In LSTM networks, the process in each cell is far more complex (see section 2.2.2 for details).

The second difference lies in the long term memory. LSTM networks keep a running vector throughout the network which contains information about each of the previous cells. This vector, known as the **cell state**, is not directly

influenced by weights, but instead is slightly modified in each cell by removing information which is no longer useful, whilst adding important information from the current input. This allows the network to keep track of long-term dependencies between the inputs far more effectively than RNNs are able to do.

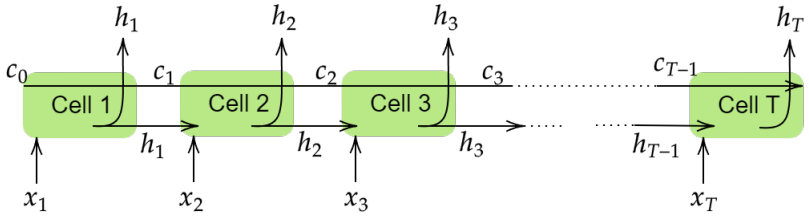

Figure 2.2: *A "folded out" LSTM neural network (figure created by author)*

Figure 2.2 shows the overall structure of an LSTM network. Here, $\mathbf{c}$ represents the cell state, whilst the hidden states are represented by $\mathbf{h}$. Each cell outputs its hidden state, whilst also sending both the hidden state and cell state on to the next cell. It is important to note here that each cell contains the same parameters, so each of the hidden states are effectively fed back into the same cell that has produced them. The above diagram represents a "folded out" version of the LSTM network, in which the cells are drawn out in a chain, rather than drawing a feedback loop as in figure 2.1.

Let us now take a look at the process that occurs in the cells, and we will see how this process mitigates the issues of vanishing and exploding gradients.

2.2.2 LSTM Cells

Inside each cell we see the following structure containing 3 distinct sections:

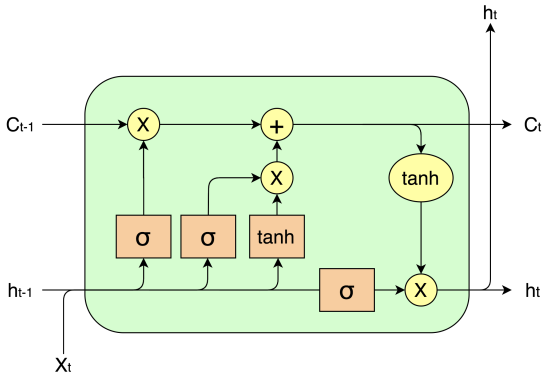

Figure 2.3: *An LSTM cell [15]*

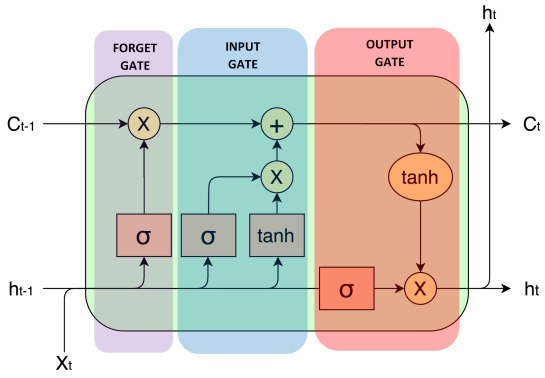

Figure 2.4: *The three sections of an LSTM cell*

The cell takes in three inputs: the cell state $\mathbf{c}_{t-1}$, the hidden state from the previous cell $\mathbf{h}_{t-1}$, and the embedded data point from the input sequence $\mathbf{x}_t$, corresponding to the current time step t. These inputs are then passed through the three different gates shown in figure 2.4. We now examine exactly what happens within each gate in the cell.

Forget Gate

The first gate determines which information from the cell state is still relevant at the current time step, and which should be discarded, or forgotten. It does this via a feed-forward layer, taking $\mathbf{x}_t$ and $\mathbf{h}_{t-1}$ as inputs, and using the sigmoid function as an activation function to produce a vector $\mathbf{f}_t$ of values between 0 and 1.

$$\mathbf{f}_t = \sigma(\mathbf{W}_{f,1} \cdot \mathbf{x}_t + \mathbf{W}_{f,2} \cdot \mathbf{h}_{t-1} + \mathbf{b}_f) \tag{2.8}$$

The cell state is then multiplied element-wise by $\mathbf{f}_t$, meaning a value of 0 in $\mathbf{f}_t$ corresponds to forgetting everything and a value of 1 corresponds to retaining everything.

Input Gate

The role of the input gate is to determine which new information from the current datapoint $\mathbf{x}_t$ and the previous hidden state $\mathbf{h}_{t-1}$ should be added to the cell state. It achieves this via two separate feed-forward layers. The first produces a candidate cell state vector $\tilde{\mathbf{c}}_t$, which contains all the information from $\mathbf{x}_t$ and $\mathbf{h}_{t-1}$, calculated by:

$$\tilde{\mathbf{c}}_t = tanh(\mathbf{W}_{\tilde{c},1} \cdot \mathbf{x}_t + \mathbf{W}_{\tilde{c},2} \cdot \mathbf{h}_{t-1} + \mathbf{b}_{\tilde{c}}) \tag{2.9}$$

The second layer is very similar to the layer in the forget gate. It produces a vector $\mathbf{i}_t$ which holds values between 0 and 1. This determines how much of the new information we keep, and how much we discard.

$$\mathbf{i}_t = \sigma(\mathbf{W}_{i,1} \cdot \mathbf{x}_t + \mathbf{W}_{i,2} \cdot \mathbf{h}_{t-1} + \mathbf{b}_i) \tag{2.10}$$

$\mathbf{i}_t$ and $\tilde{\mathbf{c}}_t$ are then multiplied together element-wise, before the result is added to the cell state. This produces the next cell state $\mathbf{c}_t$. In summary,

$$\mathbf{c}_t = \mathbf{f}_t \odot \mathbf{c}_{t-1} + \mathbf{i}_t \odot \tilde{\mathbf{c}}_t \tag{2.11}$$

where $\odot$ represents element-wise multiplication.

Output Gate

The output gate determines which information from the current cell state should be saved to the hidden state $\mathbf{h}_t$. The output gate uses a feed-forward layer in the same fashion as the forget gate to produce a vector $\mathbf{o}_t$ with values between 0 and 1.

$$\mathbf{o}_t = \sigma(\mathbf{W}_{o,1} \cdot \mathbf{x}_t + \mathbf{W}_{o,2} \cdot \mathbf{h}_{t-1} + \mathbf{b}_o) \tag{2.12}$$

This is then multiplied by the hyperbolic tangent of the cell state to produce the next hidden state $\mathbf{h}_t$.

$$\mathbf{h}_t = \mathbf{o}_t \odot tanh(\mathbf{c}_t) \tag{2.13}$$

2.2.3 Mitigation of the vanishing and exploding gradient problems

Now that we understand the structure of LSTM networks, we can discuss why they do not suffer from the gradient problems associated with RNNs. However, before we proceed, we must again prove a useful result.

Lemma 2.2. *For two vector functions $\boldsymbol{v}(\boldsymbol{x}), \boldsymbol{u}(\boldsymbol{x}) \in \mathbb{R}^n$,*

$$\frac{d}{d\boldsymbol{x}}(\boldsymbol{v}(\boldsymbol{x}) \odot \boldsymbol{u}(\boldsymbol{x})) = diag(\boldsymbol{u}(\boldsymbol{x})) \cdot \frac{d\boldsymbol{v}}{d\boldsymbol{x}} + \frac{d\boldsymbol{u}}{d\boldsymbol{x}} \cdot diag(\boldsymbol{v}(\boldsymbol{x})) \tag{2.14}$$

where $\odot$ represents element-wise multiplication, and $\frac{d\boldsymbol{f}}{d\boldsymbol{x}}$ represents the Jacobian matrix formed by $\boldsymbol{f}$ and $\boldsymbol{x}$.

Proof. let $\mathbf{w}(\mathbf{x}) = \mathbf{v}(\mathbf{x}) \odot \mathbf{u}(\mathbf{x})$.

$$\frac{d\mathbf{w}(\mathbf{x})}{d\mathbf{x}} = \begin{bmatrix} \frac{dw_1(\mathbf{x})}{dx_1} & \frac{dw_1(\mathbf{x})}{dx_2} & \cdots & \frac{dw_1(\mathbf{x})}{dx_n} \\ \frac{dw_2(\mathbf{x})}{dx_1} & \frac{dw_2(\mathbf{x})}{dx_2} & \cdots & \frac{dw_2(\mathbf{x})}{dx_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{dw_n(\mathbf{x})}{dx_1} & \frac{dw_n(\mathbf{x})}{dx_n} & \cdots & \frac{dw_n(\mathbf{x})}{dx_n} \end{bmatrix} \tag{2.15}$$

where

$$\frac{dw_i(\mathbf{x})}{dx_j} = u_i(\mathbf{x}) \cdot \frac{dv_i(\mathbf{x})}{dx_j} + \frac{du_i(\mathbf{x})}{dx_j} \cdot v_i(\mathbf{x}) \tag{2.16}$$

Therefore,

$$\frac{d\mathbf{w}(\mathbf{x})}{d\mathbf{x}} = \begin{bmatrix} u_1(\mathbf{x}) \cdot \frac{dv_1(\mathbf{x})}{dx_1} & u_1(\mathbf{x}) \cdot \frac{dv_1(\mathbf{x})}{dx_2} & \cdots & u_1(\mathbf{x}) \cdot \frac{dv_1(\mathbf{x})}{dx_n} \\ u_2(\mathbf{x}) \cdot \frac{dv_2(\mathbf{x})}{dx_1} & u_2(\mathbf{x}) \cdot \frac{dv_2(\mathbf{x})}{dx_2} & \cdots & u_2(\mathbf{x}) \cdot \frac{dv_2(\mathbf{x})}{dx_n} \\ \vdots & \vdots & \ddots & \vdots \\ u_n(\mathbf{x}) \cdot \frac{dv_n(\mathbf{x})}{dx_1} & u_n(\mathbf{x}) \cdot \frac{dv_n(\mathbf{x})}{dx_2} & \cdots & u_n(\mathbf{x}) \cdot \frac{dv_n(\mathbf{x})}{dx_n} \end{bmatrix} \tag{2.17}$$

$$\tag{2.18}$$

$$+ \begin{bmatrix} \frac{du_1(\mathbf{x})}{dx_1} \cdot v_1(\mathbf{x}) & \frac{du_1(\mathbf{x})}{dx_2} \cdot v_1(\mathbf{x}) & \cdots & \frac{du_1(\mathbf{x})}{dx_n} \cdot v_1(\mathbf{x}) \\ \frac{du_2(\mathbf{x})}{dx_1} \cdot v_2(\mathbf{x}) & \frac{du_2(\mathbf{x})}{dx_2} \cdot v_2(\mathbf{x}) & \cdots & \frac{du_2(\mathbf{x})}{dx_n} \cdot v_2(\mathbf{x}) \\ \vdots & \vdots & \ddots & \vdots \\ \frac{du_n(\mathbf{x})}{dx_1} \cdot v_n(\mathbf{x}) & \frac{du_n(\mathbf{x})}{dx_2} \cdot v_n(\mathbf{x}) & \cdots & \frac{du_n(\mathbf{x})}{dx_n} \cdot v_n(\mathbf{x}) \end{bmatrix} \tag{2.19}$$

$$\tag{2.20}$$

$$= diag(\mathbf{u}(\mathbf{x})) \cdot \frac{d\mathbf{v}}{d\mathbf{x}} + \frac{d\mathbf{u}}{d\mathbf{x}} \cdot diag(\mathbf{v}(\mathbf{x})) \tag{2.21}$$

□

We are now ready to examine why LSTM networks do not suffer from the vanishing and exploding gradient problems associated with RNNs. The following argument is original, using the equivalent analysis for RNNs as inspiration.

Again, we look at the general derivative $\frac{\partial \mathbf{h}_t}{\partial \mathbf{h}_s}$ for $0 \leq s < t \leq T$. By the chain rule,

$$\frac{\partial \mathbf{h}_t}{\partial \mathbf{h}_s} = \prod_{j=s+1}^{t} \frac{\partial \mathbf{h}_j}{\partial \mathbf{h}_{j-1}} \tag{2.22}$$

We now have $\mathbf{h}_t = \mathbf{o}_t \odot tanh(\mathbf{c}_t)$ where $\mathbf{o}_t = \sigma(\mathbf{W}_{o,1} \cdot \mathbf{x}_t + \mathbf{W}_{o,2} \cdot \mathbf{h}_{t-1} + \mathbf{b}_o)$ and so, using lemma 2.2 and lemma 2.1,

$$\frac{\partial \mathbf{h}_j}{\partial \mathbf{h}_{j-1}} = diag(\mathbf{o}_j) \cdot \frac{\partial (tanh(\mathbf{c}_j))}{\partial \mathbf{h}_{j-1}} + diag(\sigma'(\mathbf{W}_{o,1} \cdot \mathbf{x}_j + \mathbf{W}_{o,2} \cdot \mathbf{h}_{j-1} + \mathbf{b}_o)) \cdot \mathbf{W}_{o,2} \cdot diag(tanh(\mathbf{c}_j)) \tag{2.23}$$

Now, using equation (2.11) and lemma 2.2, we have

$$\frac{\partial (tanh(\mathbf{c}_j))}{\partial \mathbf{h}_{j-1}} = tanh'(\mathbf{c}_j) \cdot \frac{\partial \mathbf{c}_j}{\partial \mathbf{h}_{j-1}} \tag{2.24}$$

$$\frac{\partial \mathbf{c}_j}{\partial \mathbf{h}_{j-1}} = diag(\mathbf{f}_j) \cdot \frac{\partial \mathbf{c}_{j-1}}{\partial \mathbf{h}_{j-1}} + \frac{\partial \mathbf{f}_j}{\partial \mathbf{h}_{j-1}} \cdot diag(\mathbf{c}_{j-1}) + diag(\mathbf{i}_j) \cdot \frac{\partial \tilde{\mathbf{c}}_j}{\partial \mathbf{h}_{j-1}} + \frac{\partial \mathbf{i}_j}{\partial \mathbf{h}_{j-1}} \cdot diag(\tilde{\mathbf{c}}_j) \tag{2.25}$$

We can evaluate this using lemma 2.1, and by noting that $\mathbf{c}_{j-1}$ does not depend on $\mathbf{h}_{j-1}$ and so $\frac{\partial \mathbf{c}_{j-1}}{\partial \mathbf{h}_{j-1}} = 0$. Putting everything together, we have

$$\begin{aligned} \frac{\partial \mathbf{h}_t}{\partial \mathbf{h}_s} = \prod_{j=s+1}^{t} (&diag(\mathbf{o}_j) \cdot tanh'(\mathbf{c}_j) \cdot (diag(\sigma'(\mathbf{W}_{f,1} \cdot \mathbf{x}_j + \mathbf{W}_{f,2} \cdot \mathbf{h}_{j-1} + \mathbf{b}_f)) \cdot \mathbf{W}_{f,2} \cdot diag(\mathbf{c}_{j-1}) \\ &+ diag(\mathbf{i}_j) \cdot diag(tanh'(\mathbf{W}_{\tilde{c},1} \cdot \mathbf{x}_j + \mathbf{W}_{\tilde{c},2} \cdot \mathbf{h}_{j-1} + \mathbf{b}_{\tilde{c}})) \cdot \mathbf{W}_{\tilde{c},2} \\ &+ diag(\sigma'(\mathbf{W}_{i,1} \cdot \mathbf{x}_j + \mathbf{W}_{i,2} \cdot \mathbf{h}_{j-1} + \mathbf{b}_i)) \cdot \mathbf{W}_{i,2} \cdot diag(\tilde{\mathbf{c}}_j)) \\ &+ diag(\sigma'(\mathbf{W}_{o,1} \cdot \mathbf{x}_j + \mathbf{W}_{o,2} \cdot \mathbf{h}_{j-1} + \mathbf{b}_o)) \cdot \mathbf{W}_{o,2} \cdot diag(tanh(\mathbf{c}_j))) \end{aligned} \tag{2.26}$$

This is clearly far more complex than the corresponding equation for RNNs. The first thing to note is that the product now involves four separate weights $\mathbf{W}_{f,2}, \mathbf{W}_{\tilde{c},2}, \mathbf{W}_{i,2}$ and $\mathbf{W}_{o,2}$. This means that the network has much more freedom to adjust these weights whilst not allowing the overall product to vanish or explode. In other words, the network can self-regulate its gradients more effectively to avoid unwanted vanishing or exploding.

We also notice that the product depends on the gate outputs $\mathbf{o}_j, \mathbf{i}_j$ and $\tilde{\mathbf{c}}_j$ and the cell state $\mathbf{c}_{j-1}$. Each of these vectors change with each time step (i.e. with the introduction of each new input). As each factor in the above product refers to a different time step, this variation in the vectors helps to prevent factors from being consistently above or below 1, which would cause exploding and vanishing gradients respectively.

However, the gate outputs have a much more important role here. If we define the matrices $\mathbf{A}_j$ such that $\frac{\partial \mathbf{h}_t}{\partial \mathbf{h}_s} = \prod_{j=s+1}^{t} \mathbf{A}_j$, then the gate outputs (containing values between 0 and 1) effectively determine how strongly each of the weights $\mathbf{W}_{f,2}, \mathbf{W}_{\tilde{c},2}, \mathbf{W}_{i,2}$ and $\mathbf{W}_{o,2}$ are represented in $\mathbf{A}_j$ at each time step j. This means that these gate outputs essentially control the propagation of gradients back through the network. The gate outputs are also trainable (via adjustment of their corresponding weights and biases), meaning that the network can adjust itself to allow as much or as little gradient propagation as required [11, '*Preventing Vanishing Gradients with LSTMs*']. This was the revolutionary property of LSTM networks which allowed them to surpass the original RNN architecture in sequence processing applications.

2.3 Transformer Networks

In 2017, Ashish Vaswani, along with a team of seven other researchers, proposed yet another sequence processing architecture, the **transformer network**. Their paper, *Attention Is All You Need* [1], introduced a completely different mechanism for establishing relationships between different datapoints in an input sequence, known as **attention**. Their method allowed entire sequences to be processed in parallel rather than one datapoint at a time, which significantly reduced the training time required compared to LSTM networks and RNNs. This also enabled direct comparison between all of the inputs in the sequence which resulted in an improved ability to capture long term dependencies. The parallelisation used in transformer networks also removes the need for recurrence, which means that the exploding and vanishing gradient problems seen in RNNs are no longer as much of an issue.

Let us now take a look at the structure of transformer networks, to understand how they were able to outperform all previous sequence data architectures

2.3.1 Overall Structure

Transformer networks take a different approach to LSTM networks, instead of implementing a long chain of cells, they use a two-part structure known as an **encoder-decoder** architecture. The full structure is shown in figure 2.5. They were designed with generative AI in mind, which describes models that take in a sequence of data and predict what should come next.

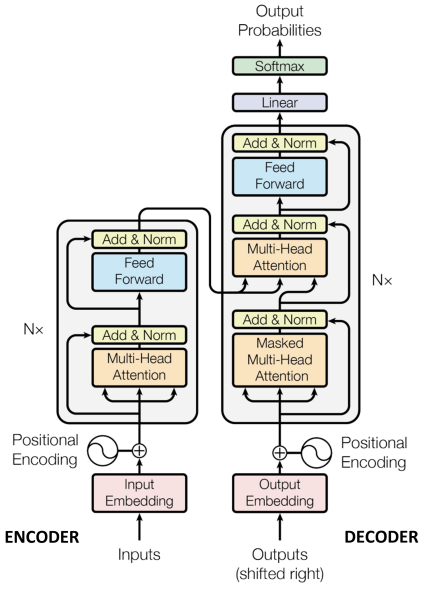

Figure 2.5: *The structure of the transformer network, as proposed in [1]*

The aim of the encoder is to extract all the necessary information from the input data. The decoder attempts to find a mapping between the input data and the outputs. We will start by analysing the structure of the encoder. Note, the remainder of this chapter is based on the ideas introduced in [1, Section 3].

2.3.2 Encoder

Inputs to the encoder are first transformed into the embedding space, in the same way as for RNNs and LSTM networks. The embedded input vectors are then given positional context in the **positional encoder**, which provides each vector with information about the corresponding datapoint's position in the input sequence. These vectors are then passed into the main encoder block, which consists of a **multi-head attention block** and a feed-forward network. This is repeated N times (here, N is a hyperparameter for us to choose) before the final output is passed on to the decoder.

Positional Encoding

As transformer networks do not involve any feedback loops, they need to use an alternative method to encode information about each datapoint's position in the input sequence. The positional encoder achieves this by adding a positional encoding vector to each of the embedding vectors. The positional encoding vectors have the same dimension as the embedding vectors to facilitate the vector addition. Each positional encoding vector is calculated using the following equations:

$$PE_{(pos,2i)} = \sin\left(\frac{pos}{10000^{2i/d_{model}}}\right) \tag{2.27}$$

$$PE_{(pos,2i+1)} = \cos\left(\frac{pos}{10000^{2i/d_{model}}}\right) \tag{2.28}$$

where $PE_{(t,k)}$ is the k^{th} entry of the positional encoding vector corresponding to x_t in the input sequence. We therefore have $i \in \{1, 2, ..., \frac{d_{model}}{2}\}$, and pos being the position of the corresponding input in the input sequence, where d_{model} is the embedding dimension that we chose in the previous layer.

This formulation of the positional encoding vector was chosen for this application for several reasons:

- It allows for input sequences of any length, whilst maintaining fixed bounds for the vector entries (entries take values between 0 and $sin(pos)$)
- The same vector is produced for each input position every time, so the model can easily identify which position each embedding vector had in the input sequence
- It allows the network to easily establish the relative positioning of embedding vectors, as "for any fixed offset k, PE_{pos+k} can be represented as a linear function of PE_{pos}" [1, Section 3.5, pg. 6] (see proof below)

Proof. (based on proof in [16, *'Derivation'*])
We need to find the linear transformation $\mathbf{M}_k \in \mathbb{R}^{2\times 2}$ independent of t, such that

$$\mathbf{M}_k \cdot \begin{bmatrix} sin(w_i \cdot t) \\ cos(w_i \cdot t) \end{bmatrix} = \begin{bmatrix} sin(w_i \cdot (t+k)) \\ cos(w_i \cdot (t+k)) \end{bmatrix} \tag{2.29}$$

where $w_i = 1/10000^{2i/d_{model}}$. Let $\mathbf{M}_k = \begin{bmatrix} m_1 & m_2 \\ m_3 & m_4 \end{bmatrix}$ then we have,

$$\begin{bmatrix} m_1 & m_2 \\ m_3 & m_4 \end{bmatrix} \cdot \begin{bmatrix} sin(w_i \cdot t) \\ cos(w_i \cdot t) \end{bmatrix} = \begin{bmatrix} sin(w_i \cdot t)\ cos(w_i \cdot k) \ + \ cos(w_i \cdot t)\ sin(w_i \cdot k) \\ cos(w_i \cdot t)\ cos(w_i \cdot k) \ - \ sin(w_i \cdot t)\ sin(w_i \cdot k) \end{bmatrix} \tag{2.30}$$

Therefore $\mathbf{M}_k = \begin{bmatrix} cos(w_i \cdot k) & sin(w_i \cdot k) \\ -\ sin(w_i \cdot k) & cos(w_i \cdot k) \end{bmatrix}$ is independent of t as required.

□

Multi-head Attention

Multi-head attention is the key feature of transformer networks which allow the processing of entire input sequences in parallel, whilst also ascertaining relationships between the datapoints in each input sequence.

This is achieved by creating a matrix of **attention scores** which represent the importance of each of the datapoints to each of the other datapoints in the context of the input sequence. For example, if the sentence "I saw a big red dog" was used as the input sequence, the words "I" and "saw" would have high attention scores with each other, as they are closely related in the sentence, despite the individual words having very different meanings. Likewise "red" and "dog" would have high attention scores with each other. These attention scores are then multiplied by vector representations of the original datapoints to give these vectors context about the relationships between their corresponding datapoint and the other datapoints in the input sequence.

This method of establishing relationships between the datapoints allows each datapoint to be compared to each of the others in exactly the same way, meaning that long term dependencies can be captured just as successfully as the short term ones. This is especially advantageous for natural language processing, as words in a sentence are not necessarily close to those which they are most related to, meaning it is essential to consider each of the possible datapoint pairings equally, no matter the distance between the datapoints involved. Neither RNNs nor LSTM networks were able to do this, and that is why transformer models were able to exhibit a much higher level of success than these previous models in natural langauge processing tasks.

Let us now look at how the attention scores are actually calculated. We'll first discuss a **single-headed attention block**, before looking at how multiple heads are applied.

The first step is to pass the input vectors through three parallel linear layers, each outputting a matrix with same dimensions as the input matrix. The three output matrices are known as the **query**, **key**, and **value** matrices.

$$\mathbf{Q} = \mathbf{X} \cdot \mathbf{W}^Q \tag{2.31}$$

$$\mathbf{K} = \mathbf{X} \cdot \mathbf{W}^K \tag{2.32}$$

$$\mathbf{V} = \mathbf{X} \cdot \mathbf{W}^V \tag{2.33}$$

where $\mathbf{X} \in \mathbb{R}^{l \times d_{model}}$ is the matrix formed by stacking the input vectors such that each row in $\mathbf{X}$ corresponds to a datapoint from the input sequence. Here, l is used to denote the length of the input sequence.
To compute the attention scores, we apply the **scaled dot-product attention** equation:

$$\mathbf{A} = softmax(\frac{\mathbf{Q} \cdot \mathbf{K}^T}{\sqrt{d_K}}) \tag{2.34}$$

where d_K is the dimension of $\mathbf{K}$, and the softmax function is applied separately to each row of the matrix $\frac{\mathbf{Q} \cdot \mathbf{K}^T}{\sqrt{d_K}}$. These scores are then multiplied by the matrix $\mathbf{V}$ to give the attention based output $Attention(\mathbf{Q}, \mathbf{K}, \mathbf{V})$.

$$Attention(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \mathbf{A} \cdot \mathbf{V} \tag{2.35}$$

To understand why equation (2.35) gives us the desired output, we must first understand what attention is, and how it works.

Definition 2.1. *An **attention mechanism** describes the creation of a context vector $\boldsymbol{c}_i$ equal to a weighted summation of input vectors $\boldsymbol{h}_j$.*

$$\boldsymbol{c}_i = \sum_{j=0}^{T} \alpha_{ij} \cdot \boldsymbol{h}_j \tag{2.36}$$

where the weights α_{ij} encapsulate the strength of relationship between $\boldsymbol{h}_i$ and $\boldsymbol{h}_j$ [17, Section 3.1, equation (5)].

Lemma 2.3. *$Attention(\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V})$ is an attention mechanism under definition 2.1.*

Proof. Let us begin by considering one row $\mathbf{q}_i$ of $\mathbf{Q}$, corresponding to the i^{th} datapoint in the input sequence. First, we multiply $\mathbf{q}_i$ by $\mathbf{K}^T$. Noting that the columns of $\mathbf{K}^T$ represent each datapoint, this produces a row vector $\boldsymbol{\alpha}_i$ in which the j^{th} entry α_{ij} represents how closely the row vectors $\mathbf{q}_i$ and $\mathbf{k}_j$ match (using the dot product as the mode

of comparison). The vector $\boldsymbol{\alpha}_i$ now holds the attention scores for $\mathbf{q}_i$.

This is then scaled by $1/\sqrt{d_K}$ to push the values away from the extreme ends of the softmax function, where the gradient is very small. The softmax function then re-scales $\boldsymbol{\alpha}_i$ to ensure that its entries add to 1. This ensures that in general, we do not increase the size of the vectors in $\mathbf{V}$ when we apply the attention scores.

The final step is to multiply $\boldsymbol{\alpha}_i$ by $\mathbf{V}$. By the rules of matrix multiplication, this will produce a row vector $\mathbf{c}_i = \sum_{j=0}^{T} \alpha_{ij} \cdot \mathbf{v}_j$. Now applying the same reasoning to the entire matrix $\mathbf{Q}$, the result will be a matrix $\mathbf{C}$, with the rows $\mathbf{c}_i = \sum_{j=0}^{T} \alpha_{ij} \cdot \mathbf{v}_j$ corresponding to the rows $\mathbf{q}_i$ of $\mathbf{Q}$.

□

The three separate linear layers producing $\mathbf{Q}$,$\mathbf{K}$ and $\mathbf{V}$ allow the model to extract the information from $\mathbf{X}$ which is the most suited to each of matrices' respective functions in the attention mechanism.

Now that we understand how attention works for a single head, let us move on to discuss multi-head attention. This involves splitting the attention mechanism into h different heads, which each compute a separate attention matrix in parallel. The h different attention matrices are then joined back together at the end of the process.

The first step is to split the input matrix $\mathbf{X}$ into h smaller matrices $\mathbf{X}_i \in \mathbb{R}^{l \times d_{model}/h}$ where l again represents the number of datapoints in the input sequence. We then split each of the weight matrices from the linear layers in the same way, resulting in h sets of $\mathbf{W}_i^Q, \mathbf{W}_i^K, \mathbf{W}_i^V \in \mathbb{R}^{d_{model}/h \times d_{model}/h}$. The result is h different sets of key, query and value matrices $\mathbf{Q}_i, \mathbf{K}_i, \mathbf{V}_i \in \mathbb{R}^{l \times d_{model}/h}$.

Now, for $i \in 1, ..., h$, we have

$$\mathbf{head}_i = softmax(\frac{\mathbf{Q}_i \cdot \mathbf{K}_i^T}{\sqrt{d_{K_i}}}) \cdot \mathbf{V}_i \tag{2.37}$$

Once the attention matrices for each head have been calculated, these are then concatenated to reform an $l \times d_{model}$ dimensional matrix.

$$Multihead(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = Concat(\mathbf{head}_1, \mathbf{head}_2, ..., \mathbf{head}_h) \tag{2.38}$$

We then pass the result through one more linear layer to produce the final output of the multi-head attention block.

$$output = Multihead(\mathbf{Q}, \mathbf{K}, \mathbf{V}) \cdot \mathbf{W}^O \tag{2.39}$$

for $\mathbf{W}^O \in \mathbb{R}^{d_{model} \times d_{model}}$.

There is a **residual connection** around the multi-head attention layer, meaning that the input matrix $\mathbf{X}$ is added to the output of the layer. Transformers can suffer from vanishing gradients as a result of implementing large stacks of similar layers [18, Section 3, pg. 3]. Residual connections help to mitigate these vanishing gradients, as the addition of the identity function $\mathbf{X}$ results in the addition of a constant 1 to the corresponding gradients. However, in the original architecture, the result of the residual connection is then layer normalised, and as shown in [18, Section 3, equation (3)], the order of these computations means that the vanishing gradient problem is not entirely removed.

Since the original proposal of transformer networks, a new method known as the **Pre-LN transformer** has been suggested as a way to remove the issue of vanishing gradients entirely. In this method, the input $\mathbf{X}$ is normalised *before* being processed by each of the sublayers in the encoder block, for example, for the multi-head attention sublayer,

$$PostLN(\mathbf{X}) = LayerNorm(Multihead(\mathbf{X}) + \mathbf{X}) \tag{2.40}$$

$$PreLN(\mathbf{X}) = Multihead(LayerNorm(\mathbf{X})) + \mathbf{X} \tag{2.41}$$

where Post-LN represents the original transformer architecture. [18, Section 3, equation (3)] shows that the Pre-LN method is effective at completely removing the vanishing gradient problem. Due to this, for deep transformers, Pre-LN transformers are more stable than Post-LN and therefore are usually preferable as the number of layers

increases [19, Section 3.3, *'Post-LN Transformer v.s. Pre-LN Transformer*]. For shallower models, the majority of research suggests that Post-LN transformers still perform better than Pre-LN. In these cases, if the model becomes too unstable due to vanishing gradients, it may be necessary to consider other variations such as those proposed in [18] and [20].

Feed-Forward network

The feed-forward section of the encoder block is a regular feed-forward network, as we've previously seen in Chapter 1. The purpose of this is to provide non-linearity between the iterations of the multi-head attention block. This allows each iteration to extract new information. Without this, performing multi-head attention N times would have limited benefit over performing it once.

There is another residual connection around this layer, and the result is again layer normalised before being passed back to the multi-head attention block. Once we have completed N iterations of the main encoder block, the result is then sent to the decoder. Here, N is a hyperparameter for us to choose.

2.3.3 Decoder

The structure of the decoder is very similar to that of the encoder, with a few key differences. As in the encoder, the input sequence to the the decoder is embedded into a higher dimension before going through the positional encoding. The result of this is then passed into the main decoder block.

Decoder Input

The first difference comes in the input to the decoder. The form of the decoder input depends on the problem we are trying to solve. The problems are split into two categories: those requiring a vector (or single value) output, and those requiring a sequential output.

For **sequence to vector** problems, the aim of the transformer is to predict the next datapoint in the sequence. To do this, it attempts to find a mapping between each datapoint and its immediate successor. Applying this mapping to the final datapoint in the sequence leads to a prediction of the next datapoint. To find this mapping, the transformer takes a decoder input of the original sequence shifted one place forwards (i.e. starting with x_2 and ending with x_T), which it later compares to the output of the encoder, which represents the original sequence.

During inference, the network will first predict the next value in the sequence, giving the output $\hat{x}_{T+1}$. To produce the next output $\hat{x}_{T+2}$ we will re-run the entire network, this time with the encoder input $\{x_2, ..., x_{T+1}\}$ if we now know the true value of x_{T+1}, or $\{x_2, ..., x_T, \hat{x}_{T+1}\}$ if we don't.

For **sequence to sequence** problems, we aim to produce an entire output sequence from the input sequence. Examples of this type of problem include language translation or generative language models. In these applications the decoder input depends on whether we are in the training or inference phase.

During training, the input is the full expected model output sequence, shifted one position to the right by adding a **starting token** (usually denoted as <sos> standing for "start of sequence") to the start of the sequence. The network then aims to output a vector for each input in the sequence (including the starting token) which is a prediction of the next word in the expected output. Each input is restricted from viewing the inputs ahead of it in the sequence, as this would make the problem trivial (see section below on masked multi-head attention).

During the inference phase, the transformer behaves as an **auto-regressive model**, meaning that the model generates the output sequence one datapoint at a time, each time using the output sequence generated so far as input to the decoder to produce the next datapoint. We will use an example to illustrate this point more clearly. Imagine the machine was trying to predict the german translation of "I love you" - "Ich liebe dich". Initially the sequence "I love you" would be encoded in the encoder. Whilst the input to the decoder would just be a starting token, <sos>. The network should then produce the output "Ich". We would then re-run the decoder, this time inputting the two-element sequence {<sos>, "Ich"}, which should then produce the output "liebe". The next decoder input would be {<sos>, "Ich", "liebe"}, and so on. On the final iteration, the network should produce an **end of sequence token** <eos> as its output, indicating that we have reached the end of the output sequence, and we should

terminate the loop.

We now move into the main encoder block, to discuss the next difference between the encoder and the decoder.

Masked Multi-head Attention

As in the encoder, the first section of the decoder block is a multi-head attention block. For sequence to vector tasks, this block is no different from the equivalent block in the encoder. However, for sequence to sequence tasks, we use **masked multi-head attention** here. This means that each input is only compared to the inputs that come before it in the input sequence when calculating the attention scores.

The reason we do this is to mimic the situation during the inference phase, where the model only has access to previously generated outputs when predicting the next output. By inputting the entire output sequence at once, but masking future datapoints, the model can effectively train the generation of an entire sequence in parallel. The decoder will produce an output for each of its inputs, each output a prediction of the immediate successor of its corresponding input in the sequence.

To acheive this masking, we calculate $\frac{\mathbf{Q}\cdot\mathbf{K}^T}{\sqrt{d_K}}$ as normal, except that we set all of the values above the leading diagonal to $-\inf$. This way, all of these values will be transformed to 0 by the softmax function, whilst the non-zero attention scores in each row still add to 1.

This masking is not required for sequence to vector problems, as in these problems, we do not input any information into the decoder that we wouldn't have access to during inference.

Encoder-decoder Multi-head Attention

The result of the masked multi-head attention block is then passed into another multi-head attention block. This is where we combine the encoder output with the decoder vectors by using the masked multi-head attention output for the query matrix, whilst the key and value matrices come from the encoder output.

This allows the decoder vectors to attend to each of the vectors produced by the encoder, which is essential to combine the information from the inputs with the information from the previously generated outputs. It also helps the model to align inputs with generated outputs, allowing it to effectively create a mapping between the two.

Apart from these key differences, the decoder is structured in the same way as the encoder, with a feed forward network at the end of the main block, residual connections around each of the blocks within the main block, and N repetitions of the main block. The result is then processed to create an output.

Output

The output of the main decoder block is passed through a final linear layer to adjust the size of the matrix to the desired output size. It then produces an output using the method best suited to the problem we are trying to solve.

For categorical data, the output is passed through a softmax layer, to produce probabilities for each possible outcome. The most likely outcome is then chosen as the output of the network.

For regression problems, softmax is not required. A linear layer is applied to produce a single output value.

Chapter 3

The Stock Market

Before we explore the use of transformer networks on stock market data, we first need to gain some general understanding of how the stock market works. The purpose of this chapter is to provide a simplistic overview of the key concepts governing the prices of stocks in the stock market.

3.1 Introducing the Stock Market

We begin with a few important definitions to start introducing how the stock market works.

Definition 3.1. *A company's **capital** refers to the financial resources it has to operate and expand its business.*

Definition 3.2. *A **security** is a tradable financial asset which holds monetary value. Securities are split into two distinct categories:*

1. ***Equity** - a type of security which represents ownership in a company, and entitles the holder to a proportion of the company's assets and distributed earnings*
2. ***Debt** - a loan made by investors to companies, with a guaranteed returned interest rate*

Definition 3.3. *A **stock** (also known as a **share**) in a company is a type of equity security which represents ownership in that company in such a way that a shareholder owning $x\%$ of the total number of issued (or outstanding) shares for a company, owns $x\%$ of that company.*

Companies issue financial securities to raise capital. These are sold to investors, who are then free to resell them to other investors if they choose to.

The **stock market** is the virtual arena in which financial securities are traded. It contains a set of rules and methods to facilitate the issue and trading of these securities.

Now that we understand some key definitions, we need to ascertain exactly what it means to own shares in a corporation.

3.1.1 Owning Shares

Whilst public companies are technically entirely owned by their shareholders, they are legally considered to be their own entities. This has two important legal implications:

1. The assets of the company legally belong to the company, not the shareholders. Therefore shareholders have no direct control or ownership over the company's assets.
2. There is a separation of liability between the company and the shareholders. This means that the shareholders are not legally or financially responsible for the company, i.e. if the company gets sued, or goes into debt, the shareholders are not required to pay any money from their own pockets. Likewise, if a shareholder finds themselves in financial difficulty, they cannot sell the company's assets for personal monetary gain.

Therefore, it is more accurate to say that shareholders have *indirect* ownership of the company. However, whilst the shareholders do not directly own the assets of the company, their share ownership does entitle them to certain rights and privileges. In some cases these privileges include voting rights within the company and dividend payouts - a proportion of the company's profits paid to the shareholders. However, stock market investors usually consider the most important privilege to be **capital appreciation**. This describes the benefit brought about by increases in the stock price, making the ownership of these assets more valuable.

3.2 Stock Pricing

Due to the profitability of capital appreciation, investors are always seeking to buy stocks which they think will increase in price. In order for us to predict which stocks will increase in price, we must first understand exactly what determines the price of each stock.

3.2.1 Initial pricing

When a company first goes public, that is, first allows members of the public to buy their shares, they need to go through a process known as an **initial public offering (IPO)**. This is a complex process which culminates in the company offering its shares to investors for a set price.

To determine how much it can expect investors to pay, the company must look at several key factors including its financial performance and growth prospects, overall market conditions, current trends in the industry, and the price of its competitors' shares. It will become clear as we progress through this section why these particular variables are important to the investors.

The company sells its initial set of shares via the **primary market**.

Definition 3.4. *The **primary market** is the market in which companies sell shares directly to investors. This market is used both during IPOs, and if a company decides to sell more shares to the public at any subsequent point in time to increase its capital*

The investors are then free to sell these shares via the **secondary market**.

Definition 3.5. *The **secondary market** is the market in which investors can trade the shares that were initially purchased via the primary market, with each other, with no direct involvement of the company. This is primarily done via stock exchanges, and is how most stock traders acquire and sell their stocks*

3.2.2 Subsequent price changes

The stock market is a highly complex environment in which a large variety of different factors affect prices. This makes it virtually impossible to predict prices with even close to 100% accuracy. We will begin by focusing on the direct cause of the fluctuations of stock prices before exploring a few of the factors affecting the fundamental value of shares later on in the section.

In the broadest sense, the price of every stock in the market is determined by a simple case of supply vs. demand. In the secondary market there are potential buyers and potential sellers for any particular stock. If there are more buyers than sellers, the value of the stock is increased, with the converse happening when there are more sellers than buyers.

The driving force of the prices of stocks is therefore the demand for each stock. To gain a deeper understanding therefore, we need to analyse what factors cause an increase or decrease in demand.

3.2.3 Factors Affecting Demand

It is well known that companies which are performing well and growing often see a corresponding increase in their stock prices. Conversely companies which are struggling to stay afloat, or are losing profits, tend to see a decrease in their stock prices. We begin this section by exploring why a company's performance is one of the key factors driving the demand for its shares.

Whilst the ownership of shares technically entitles the shareholder to a percentage of the company's profits and assets, a large proportion of shareholders never receive direct payouts via dividends. However, shares still hold underlying value due to the potential for future payouts. This can come in many forms, including:

- Future dividends - whilst companies which are in the growth stage of their existence generally will not pay out dividends, most large companies do, and so a company which is growing and is deemed to have a large future potential, holds the possibility of future payouts via dividends
- A takeover - if another company enters into a takeover, they may buy all or a proportion of the shares of the previous company. Usually the company taking over will pay a premium to buy the shares, to convince the shareholders to sell
- Share buybacks - in some situations, a company may buy back some of its shares from the shareholders. Therefore, the remaining shares are likely to go up in value due to a reduced supply. In this situation, a shareholder who keeps their shares will effectively profit via the increase in value of their assets. This usually occurs when a company perceives itself to be doing well.

As a general rule, these payouts are more likely to happen when a company is successful, and the payout is likely to be more significant for companies with larger profits or higher value assets. This is what gives shares their intrinsic value and is why share prices are typically linked to a company's performance.

Definition 3.6. *The **intrinsic value** of a share is the fundamental worth of the share based on its future potential profitability.*

Investors aiming to make a profit via capital appreciation aim to find discrepancies between the current price of a stock, and its intrinsic value.

In general, it is not possible to calculate the intrinsic values of stocks exactly, as they usually depend on a huge variety of factors, with complex dependencies on each factor. However, investors can try to estimate intrinsic values based on a collection of main factors usually spanning individual stock data, factors affecting specific market sectors, and those affecting the overall market. These often include company management, competitive dynamics between companies and industry trends, as well as interest rates, government policies, and a multitude of others. These factors are analysed to create a picture of how each company is likely to perform in the future, which is then used to compute the estimation of the intrinsic value.

Estimating the intrinsic value of stocks provides a basis for how investors choose which stock to buy. A stock whose intrinsic value is perceived to be higher than its market value is **undervalued**, and those for which their intrinsic value is lower than the market value, are **overvalued**. If any shares are indeed undervalued, when investors realise that these stocks are worth more than they cost, they will want to buy them. This increases demand for these stocks and therefore the prices will go up. Therefore an equivalence can be drawn between assessing a stock to be undervalued and predicting the price to go up. Likewise, assessing a stock to be undervalued can be thought of as equivalent to predicting that the price will go down.

Definition 3.7. *For shares that are deemed to be overvalued, investors may perform a transaction known as a **short sell**. This involves 'borrowing' shares from a broker with the promise that they will give them back at some future time. The investor will then immediately sell the borrowed shares, with the hope that the shares will decrease in price. They can then buy them back at a reduced price at a later date, before returning them to the broker.*

By buying undervalued shares and short-selling overvalued shares, investors can profit from all shares which they perceive to be misvalued (i.e. the intrinsic value does not equal the market value).

3.3 Predicting stock prices

The ability to predict stock prices has the potential to make large profits, depending on the level of success of the predictions. Investors do not need to get all of our predictions right, they simply need to get enough of them right that the profits from these predictions outweigh the losses from incorrect predictions. However, achieving this degree of success is very difficult, as we will see that stock prices already take into account their company's perceived future earning potential [21, *'Reasons for stock prices being a leading indicator'*, pg. 8], and thus closely

match their intrinsic values.

There are two main methods for predicting stock prices: technical analysis and fundamental analysis. The two differ in which information they use to make their predictions.

Definition 3.8. ***Technical analysis*** *of the stock market involves analysis of past stock price data to predict new prices. This method relies on identifying trends and patterns in past data which can be applied to current prices.*

Definition 3.9. ***Fundamental analysis*** *of the stock market involves attempting to ascertain the intrinsic value of stocks, by analysing factors such as financial performance, industry trends, growth prospects, management quality, and many others. This method relies on the ability to directly locate stocks which are under or overvalued.*

In this report, we will mainly focus on using technical analysis to predict prices. However, the ideas in this section will show that it is very difficult to predict prices no matter which method we use.

Theory 3.1. *The* ***random walk theory*** *suggests that stock prices fluctuate randomly between consecutive time steps.*

This theory, first popularised by Burton Malkiel in his 1973 book, *A Random Walk Down Wall Street* [22], contains the implication that it is not possible to predict the short-term changes in stock prices using past data. The efficient market hypothesis builds on this idea to make a stronger statement.

Theory 3.2. *The* ***efficient market hypothesis (EMH)*** *suggests that the market is efficient in reflecting all past information in asset prices. This comes in three forms [23, Section I, pg. 383]:*

1. ***Weak efficiency*** *- the asset prices reflect all historical prices. This means that it is impossible to predict new prices by using technical analysis.*
2. ***Semi-strong efficiency*** *- the asset prices reflect all historical publicly available data. This means that neither technical analysts, nor fundamental analysts working with publicly available data will have any success.*
3. ***Strong efficiency*** *- the asset prices reflect all historical publicly available data as well as all historical insider data, i.e. they reflect all historical data known to anyone. This means that no form of stock market analysis will lead to successful predictions.*

To explain this further, if all the data that we are using to estimate the intrinsic value of stocks is already represented in their current prices, this means that if we use all the data available to us and process it using a flawless estimation technique, our intrinsic value estimate will equal the current price. In other words, the best possible estimation we can achieve for the intrinsic value of any stock is its current price. Combine this with the random walk theory, and we are left with no information to help us predict future prices.

Market efficiency comes about due to the fact that once discovered, undervalued stocks will have a high demand, and overvalued stocks, a low demand. This means that each stock price is constantly pushed towards its intrinsic value. In efficient markets, this push occurs instantaneously as stock prices deviate from their intrinsic value, and thus the price is immediately restored to its intrinsic value. Stock market prices are generally accepted to follow the rules of semi-strong efficiency. If we accept this to be the case, then attempting to predict prices without insider information becomes futile. However, in practice, markets are rarely 100% efficient.

Theory 3.3. *The* ***intrinsic value theory*** *suggests that the intrinsic value of stocks can differ from their market value due to inefficiencies in the market. This is contrary to the EMH, and allows the possibility for investors to consistently outperform the market via technical and fundamental analysis.*

The intrinsic value theory introduces the idea of **market inefficiencies**. These can be caused by two main reasons. Both occur due the fact that it is the collective group of investors who have the predominant influence over stock prices. This means that rather than reflecting all the information available, it would be more accurate to say that stock prices reflect the *perceived implications* of all the available information by investors.

The first cause of inefficiency is the implausibility of investors *instantaneously* processing information and investing accordingly when the information becomes available. This causes a short **lag** between information becoming available and the prices adjusting to their new intrinsic values. This means that those who are able to find the information quicker than the majority of other investors, will be able to act on it before the prices have changed to

reflect that information.

The other main cause is a phenomenon known as **trading bias**. This is an effect that comes about due to investors not always being entirely rational. This means that they will make uninformed decisions which lead to stock prices deviating from their exact representation of all the available information. If enough investors make the same irrational decisions, then it can often result in stock prices moving significantly away from their intrinsic value.

There are several different reasons why these inefficiencies occur [24, *'The most common trading biases'*]. The first is the **herd mentality bias**, which is a phenomenon where people tend to follow the crowd, even if they don't agree with the crowd's decisions. Investors are also likely to want to stick to trading strategies which have worked in the past, even when the market changes. This leads to some prices being very slow to react to changes in the overall market.

This slow reaction is reinforced by the **endowment effect**, which is an effect caused by the sentimentality of people leading to them perceiving that items they own are more valuable than equivalent items which they do not own. This can often result in investors holding on to shares longer than they should.

Investors may also fall for the **gambler's fallacy** which is the false idea that events which have happened at an unusually high rate in the past will be less likely to happen again in the future, despite the past events not affecting the probabilities of future events. For example, if a particular stock, for which the price is normally relatively stable, has increased in price for five consecutive days, investors may believe that this indicates that it is more likely to fall in the near future.

These are just a few of the trading biases which investors are prone to, but they give us an idea of how inefficiencies can open up in the market. Whilst the semi-strong EMH is widely accepted as a good descriptor of the stock market, it is generally not thought to describe the stock market perfectly. In other words, it is generally accepted that inefficiencies do occur within the market.

Market inefficiencies open up an opportunity for both technical and fundamental analysis to be used profitably. This is because accurate estimates of the intrinsic values of stocks can enable investors to identify the instances of misvalued stocks caused by these inefficiencies.

In the remainder of this report, we will explore the use of transformer networks to apply technical analysis to historical price data. (See appendix A for details of the different types of historical price data available). These networks will primarily aim to spot the patterns exhibited by market inefficiencies, and use these patterns to identify mismatches between current stock prices and their intrinsic values. In this way, we aim to predict stock prices with enough accuracy to be able make a profit using our predictions.

Chapter 4

Using A Transformer Network To Predict Closing Prices

In this chapter, we move on to explore the practical use of transformer networks by implementing a transformer network to predict stock prices. Note that the network design in this question is based on the model proposed in [5], however, the work from section 4.1.1 onwards is original. After initial results we look at improvements we need to make, before exploring the successes and drawbacks of the network design.

4.1 Network Design

In 2022, Chaojie Wang and his fellow researchers found success using the transformer architecture to predict stock market prices [5]. They used a standard sequence-to-vector transformer model, and inputted a sequence of closing prices with the aim of predicting the next day's closing price.

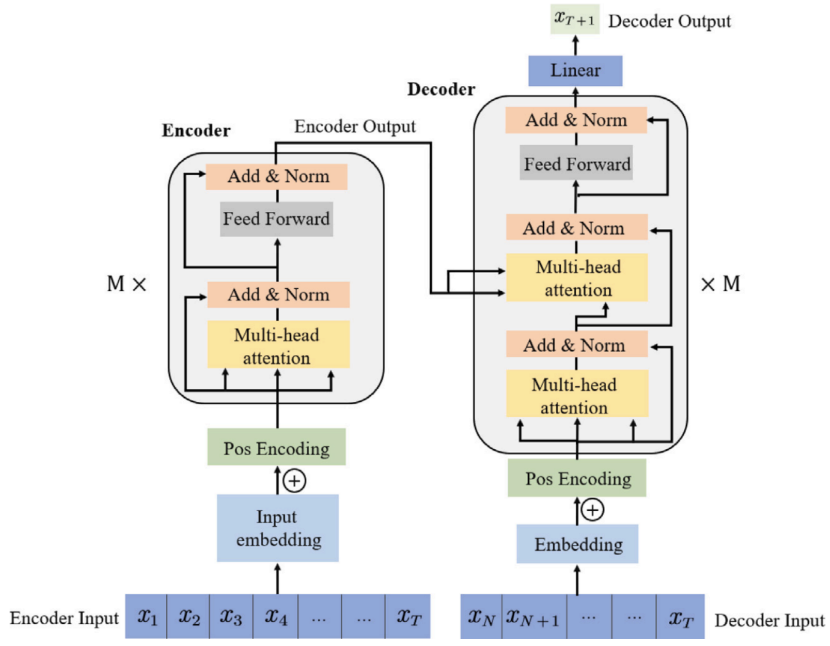

Figure 4.1: *The transformer model used by Wang et al. to predict stock market prices in [5]*

Both the training and test datasets were first normalised by taking

$$\hat{x}_{ut} = \frac{x_{ut} - \mu_u}{\sigma_u}$$

where $\hat{x}_{ut}$ is the normalised closing price for stock u at time t, x_{ut} is the original closing price, and μ_u and σ_u are the mean and standard deviation respectively of the set of prices from stock u. This normalisation improves the stability of the model, and increases the generalisability to allow the model to deal with different stocks.

In their paper, Wang et al. claim to be able to able to execute a successful trading strategy using the predictions from their model. Here, we will use a slightly different simulation method than they used, however, this should not change the overall outcome. If y_t represents the actual stock price at time t, and $\hat{y}_t$ represents the predicted stock price at time t, we simulate the following strategy:

- if $\hat{y}_{t+1} > y_t$, we buy the number of units of the stock equal to the total value of our portfolio
- if $\hat{y}_{t+1} < y_t$, we short sell the number of units of the stock equal to the total value of our portfolio
- if $\hat{y}_{t+1} = y_t$, we do nothing

We then calculate the return at time $t+1$ to be

$$R_{t+1} = \frac{y_{t+1} - y_t}{y_t} \cdot sign(\hat{y}_{t+1} - y_t) \tag{4.1}$$

Following the trading strategy detailed above, R_{t+1} represents the percentage change in portfolio value from time t to time $t+1$

$$R_{t+1} = \frac{V_{t+1} - V_t}{V_t} \tag{4.2}$$

where V_t is the value of our portfolio at time t. Therefore,

$$1 + R_{t+1} = \frac{V_{t+1}}{V_t} \tag{4.3}$$

and so,

$$\%\ profit = -1 + \prod_{t=1}^{T}(1 + R_t) = \frac{V_T - V_0}{V_0} \tag{4.4}$$

In this chapter, we will explore whether the results of Wang et al. can be repeated, or even improved upon, and in doing so, ascertain whether the transformer architecture provides a viable model for predicting stock prices.

4.1.1 Initial results

Using the network design proposed by Wang et al. on a selection of 5 different pre-selected stocks yielded the results shown in figure 4.2.

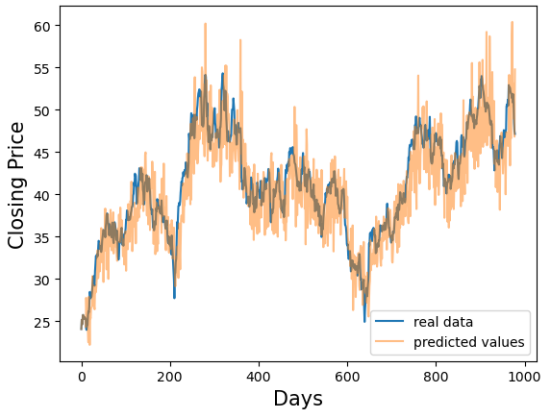

Figure 4.2: *Initial price predictions using data from aal.us*

To test the results, we have run the simulated trading strategy introduced in section 4.1 for 1000 days on 20 separate stocks. These stocks have been pre-selected by choosing stocks with relatively low volatility, which should allow the model to perform to its maximum ability. Running this simulation, the mean MSE was 11.14 and the mean

percentage profit was 0.047. Whilst this model has already resulted in a profit (assuming there are no brokerage fees on trades), the margin is very small, and the profit could therefore easily fall below 0 for other test data sets. We would ideally like to improve these results to improve the profit margin and therefore increase the robustness of the trading strategy.

One of the immediate problems we can see here is that there is far too much noise in the predictions. This is a by-product of having a large quantity of noise in the input data, which is one of the main problems involved in stock market predictions. Neural networks are in general not very effective at dealing with noisy data. They are designed to identify patterns in the data, and excessive noise greatly inhibits their ability to do so.

For this reason it is essential to try to reduce the noise as much as possible in the input data without losing any of the key information. There are several ways in which we can do this, with one of the most effective methods being smoothing.

4.2 Smoothing

In this section, we smooth the input data to reduce the peaks and troughs caused by random day to day fluctuations. This allows the network to focus on finding overall trends in the value of each stock without being distracted by noise. There are several different methods for doing this.

4.2.1 Moving average

The moving average is the most simple method of smoothing in which each of the inputs x_t is replaced by

$$\tilde{x}_t = \frac{\sum_{i=0}^{s} x_{t-i}}{s} \tag{4.5}$$

where in our case, x_t is the closing price at time t, and s is the smoothing constant which determines the length of the moving average.

We can take this one step further by adding a weighting to each data point in a process known as **weighted moving average**. The weightings can be chosen based on the characteristics of the data, and how much importance we wish to give to each data point in the average, however they must always add up to 1. Usually, the weightings increase with t, as data points closer to the current time tend to hold more significance. We then calculate the new input values as follows:

$$\tilde{x}_t = \frac{\sum_{i=0}^{s} a_{t-i} \cdot x_{t-i}}{\sum_{i=1}^{s} a_{t-i}} \tag{4.6}$$

where the a_i are the weightings given to each data point.

One common extension of this is the **exponential moving average**, which assigns a set of weights which decrease exponentially as we go further back in time. In this case, we calculate the smoothed values as follows:

$$\tilde{x}_t = \alpha \cdot x_t + (1-\alpha) \cdot \tilde{x}_{t-1} \tag{4.7}$$

where α is a smoothing constant between 0 and 1. A value closer to 1 will give more weight to recent observations, whilst a value closer to 0 has the opposite effect. By expanding (4.7) we see that the weights do indeed decrease exponentially as we move backwards in time:

$$\begin{aligned}
\tilde{x}_t &= \alpha \cdot x_t + (1-\alpha) \cdot \tilde{x}_{t-1} \\
&= \alpha \cdot x_t + \alpha(1-\alpha) \cdot x_{t-1} + (1-\alpha)^2 \cdot \tilde{x}_{t-2} \\
&= ... \\
&= \alpha \cdot (x_t + (1-\alpha) \cdot x_{t-1} + (1-\alpha)^2 \cdot x_{t-2} + ... + (1-\alpha)^i \cdot x_{t-i} + ... + (1-\alpha)^{t-1} \cdot x_1) + (1-\alpha)^t \cdot x_0
\end{aligned}$$

and by summing the weights, we notice that:

$$\sum weights = \frac{\alpha(1-(1-\alpha)^t}{1-(1-\alpha)} + (1-\alpha)^t = 1 - (1-\alpha)^t + (1-\alpha)^t = 1 \tag{4.8}$$

meaning that the exponential moving average is indeed a valid type of moving average.

Whilst the moving average method is effective at removing noise from the data, one drawback we find is that they are slow to react to new information or trends, resulting in a lag between the general curve of the real data, and the moving average.

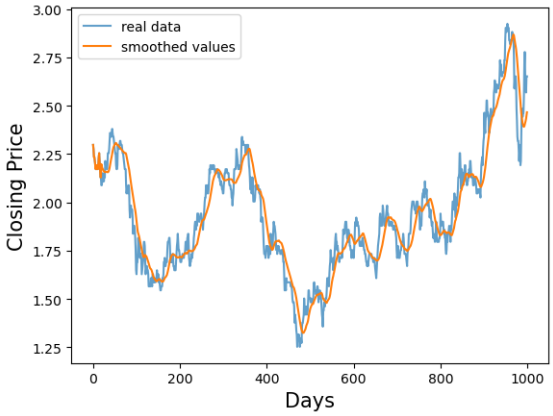

Figure 4.3: *A moving average of length 20 applied to stock price data*

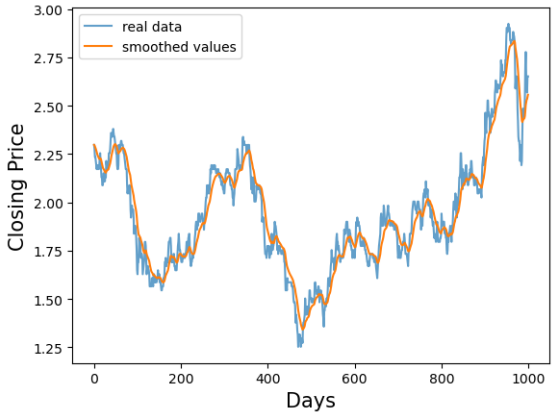

Figure 4.4: *An exponential moving average with $\alpha = 0.1$ applied to stock price data*

(*figures created by author*)

With a moving average of length s, if we denote the value of the moving average at time t as $\tilde{x}_t$, we have:

$$\tilde{x}_{t+1} - \tilde{x}_t = \frac{\sum_{i=0}^{s} x_{t+1-i}}{s} - \frac{\sum_{i=0}^{s} x_{t-i}}{s} \tag{4.9}$$

$$= \frac{x_{t+1} - x_{t-s}}{s} \tag{4.10}$$

Therefore,

$$|\tilde{x}_{t+1} - \tilde{x}_t| > |x_{t+1} - x_t| \iff |x_{t+1} - x_{t-s}| > s|x_{t+1} - x_t| \tag{4.11}$$

In most cases, the right hand inequality will not hold, and so the change in the moving average at each time step will be smaller than the change in the original prices. This is exactly what we want in general, as it means the moving average will be less affected by day to day noise. However, when we observe an upwards or downwards trend for a sustained period of time, this is when the lag effect occurs, as the moving average is slow to catch up.

A commonly used method of reducing this lag effect, is by centering the average around the current datapoint, i.e. we calculate the new input values using,

$$\frac{\sum_{i=t-\lceil s/2 \rceil}^{t+\lfloor s/2 \rfloor} x_i}{s} \tag{4.12}$$

Now, during an upwards trend, the moving average will be pulled upwards by the future values to match the original price values, and will likewise be pulled downwards during a downwards trend.

Note, it is more difficult to do the same correction for an exponential moving average, as due to the increasing number of terms in the sequence of previous prices, the centre of each average does not move by a whole day with each successive calculation.

Centering the moving average is effective at mitigating the lag (as shown in table 4.1). However to compute the centered moving average, we rely on knowledge of future prices. During the training phase, it is possible to use this method of smoothing, as we have access to the entire dataset from the start, however, during inference, we will not have access to any future values. This is a significant issue, as, if we train our network on smoothed data, we will get the best results during inference by again smoothing the input data in the same way. However, this is not

possible when we don't have access to future values. Therefore this is not a viable method of reducing lag for our particular application.

From figure 4.3, we note that the lag works as follows. When we observe an upwards trend in the data, the prediction of the moving average is too low, and vice versa. Thus it seems that a viable solution to this, without using future data, is to add a term to the moving average proportional to the current slope of the curve $\tilde{x}(t) = f(t)$, i.e. we set

$$\tilde{x}'_t = \tilde{x}_t + \beta \cdot \frac{d\tilde{x}_t}{dt}(t) \tag{4.13}$$

The first issue here is that we do not know the equation of the the curve $f(t)$. However, we can estimate it by saying that $\frac{d\tilde{x}_t}{dt}(t) \approx \tilde{x}_t - \tilde{x}_{t-1}$. We also have the issue that even in the smoothed curve $f(t)$, the gradient is constantly changing, meaning that adding a term proportional to this gradient will increase the level of noise, which is exactly the opposite of what we are trying to achieve. To reduce this, we note that the lag is worst during steep sections of the curve. Therefore we can set a minimum gradient, γ such that the extra term is only added when $\tilde{x}_t - \tilde{x}_{t-1} > \gamma$. This way, the noise in flat sections will not increase, but in steep sections where the lag is worst, we can reduce some of the lag. Using these techniques, we now have

$$\tilde{x}'_t = \tilde{x}_t + \beta \cdot (\tilde{x}_t - \tilde{x}_{t-1}) \cdot \mathbb{1}(\tilde{x}_t - \tilde{x}_{t-1} > \gamma) \tag{4.14}$$

We are then left with the task of choosing a value for β and γ. This can be solved via machine learning, by minimising the MSE or bias (see bias definition below). This technique, which we will call the **adjusted moving average**, may also be applied to the exponential moving average, as it does not depend on the length of the average.

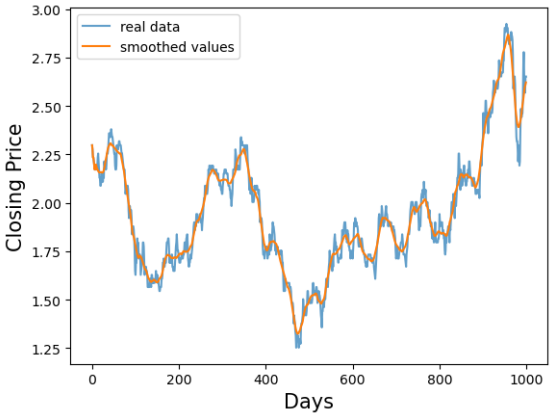

Figure 4.5: *A centered moving average of length 20 applied to stock price data*

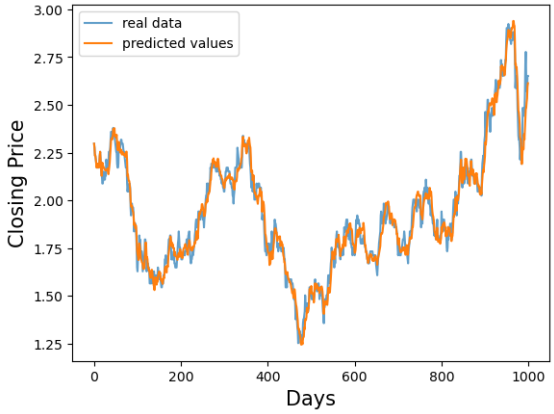

Figure 4.6: *An adjusted moving average of length 20 applied to stock price data*

(*figures created by author*)

We can see from figures 4.5 and 4.6 that centering the moving average provides the best way of reducing the lag effect. The adjusted moving average also successfully mitigates the lag effect, however, it adds some noise back to the smoothed curve.

We can measure the effectiveness of each smoothing technique using two different metrics. The first is the mean squared error (MSE) calculated by

$$\frac{1}{T}\sum_{t=1}^{t=T}(\tilde{x}_t - x_t)^2 \tag{4.15}$$

This determines how closely the smoothed curve follows the original data. The second metric we refer to as the **bias**, and is calculated in the following way:

```
bias = 0;
count = 0;
for t in [1,T]; do
    if sign(x̃_t − x_t) = sign(x̃_{t−1} − x_{t−1}) then
        count = count + 1;
        bias = bias + count^2;
    else
        count = 0;
```

This algorithm adds a quadratically increasing amount to the bias for every consecutive day the smoothed curve is the same side (above or below) of the original curve. This ensures that long periods of time for which the smoothed curve is above or below the curve are penalised, for instance when lag is present. Therefore, this metric effectively determines how well the smoothed curve actually fits the data.

An effective smoothing method should result in a small bias value. Given a small bias value, the MSE value is effectively a measure of smoothness, with larger values corresponding to smoother curves. This is because smoother curves track less of the volatility of the original data, and therefore receive a higher contribution to the MSE due to the volatility of the stock.

	s	Moving av. (MA)	centered MA	Adjusted MA		α	Exponential MA
MSE	3	0.5909	0.1929	0.5707		0.25	1.4573
	5	1.3402	0.3941	0.6675		0.20	2.0293
	10	3.2700	0.9369	3.2700		0.15	2.9771
	20	7.1204	1.8982	3.4420		0.10	4.8281
Bias	3	4336	546	4103		0.25	56881
	5	17469	1852	8883		0.20	96575
	10	65812	8429	65812		0.15	145547
	20	274304	20576	106333		0.10	312568

Table 4.1: *The MSE and bias scores of each of the different moving average techniques*

These results are as we would expect, with the centered moving average producing significantly lower bias values than the other methods, due to the reduction of the lag effect. Whilst the adjusted moving average does somewhat reduce the bias compared to the original moving average, the reduction is not significant compared to that of the centered moving average.

These moving average techniques do produce smooth curves for the most part, however, we have seen that each of the variations has certain issues. MA and exponential MA result in the lag effect, centered MA relies on future data, and adjusted MA adds noise back into the curve. Ideally we would want to find a smoothing technique which replicates the results of centered MA whilst not relying on future data.

4.2.2 Piecewise regression smoothing

The other main type of smoothing technique involves fitting a piecewise function to the data. We first divide the dataset into m equal sized sections, with the set of datapoints in the i^{th} section denoted as D_i. The contents of this section were inspired by [7, Chapter 10], with the definitions coming directly from this source.
In the simplest case - piecewise linear regression smoothing - we fit a separate linear regression line to each section of the data

$$\tilde{x}_t = \hat{\alpha}_i + \hat{\beta}_i \cdot t \qquad for\ t\ s.t.\ x_t \in D_i \tag{4.16}$$

with

$$x_t = \hat{\alpha}_i + \hat{\beta}_i \cdot t + \epsilon_t \tag{4.17}$$

where α_i and β_i are the estimated constant parameters calculated using OLS regression. This is a simple machine learning problem to find the optimum $\hat{\alpha}_i$ and $\hat{\beta}_i$ values, which minimise $\sum_{t=0}^{T} \epsilon_t^2$.

Definition 4.1. *The boundaries between the different sections of a piecewise function are known as* ***knots****.*

In stock market data, we hardly ever see linearity, even within small time frames. Therefore piecewise linear regression may not be the best modelling technique for our purposes.

Piecewise polynomial regression may be better suited to fit our data. As it suggests in the name, polynomial regression fits a polynomial curve to the data instead of being confined by linearity. In the case of piecewise polynomial regression, we fit the curves

$$\hat{x}_t = \hat{\beta}_{i,0} + \hat{\beta}_{i,1} \cdot t + \hat{\beta}_{i,2} \cdot t^2 + \hat{\beta}_{i,3} \cdot t^3 + ... + \hat{\beta}_{i,d} \cdot t^d \qquad for\ t\ s.t.\ x_t \in D_i \tag{4.18}$$

where similarly

$$x_t = \hat{\beta}_{i,0} + \hat{\beta}_{i,1} \cdot t + \hat{\beta}_{i,2} \cdot t^2 + \hat{\beta}_{i,3} \cdot t^3 + ... + \hat{\beta}_{i,d} \cdot t^d + \epsilon_t \tag{4.19}$$

Here, the degree d is up to us to choose, with $d <= 4$ being the most commonly used - values higher than 4 tend not to give a significant improvement on results, whilst also increasing the number of parameters we need to optimise. Once we have chosen d, we can again treat this as a machine learning problem, selecting the values of the parameters which minimise the sum of the residuals squared.

Whilst this method should give better results than the linear regression, we still have an issue with what to do at the knots. Currently, we are likely to have a discontinuity at each of the knots, which is not ideal, as although our data points occur at discrete time values, stock market prices move almost continuously, and so it makes more sense to model them with a continuous curve.

We can solve this problem by applying constraints to the equations of our polynomial regression curves. For example, if we set $d = 3$, on the first knot between D_0 and D_1, we set:

$$\hat{\beta}_{0,0} + \hat{\beta}_{0,1} \cdot t_1 + \hat{\beta}_{0,2} \cdot t_1^2 + \hat{\beta}_{0,3} \cdot t_1^3 = \hat{\beta}_{1,0} + \hat{\beta}_{1,1} \cdot t_1 + \hat{\beta}_{1,2} \cdot t_1^2 + \hat{\beta}_{1,3} \cdot t_1^3 \tag{4.20}$$

where we've defined t_1 to be the time at the knot between D_0 and D_1. Rearranging this, we get

$$\hat{\beta}_{0,0} = \hat{\beta}_{1,0} + (\hat{\beta}_{1,1} - \hat{\beta}_{0,1}) \cdot t_1 + (\hat{\beta}_{1,2} - \hat{\beta}_{0,2}) \cdot t_1^2 + (\hat{\beta}_{1,3} - \hat{\beta}_{0,3}) \cdot t_1^3 \tag{4.21}$$

Thus, by setting this constraint, we have determined the value of $\hat{\beta}_{0,0}$, and removed one degree of freedom from our problem. This has the added advantage of resulting in one fewer parameter to optimise, reducing the computational resources required to fit the regression curves.

It is also of interest to us to analyse the gradient of our curve, as one of the important questions we need to ask is whether the price will go up or down. It is therefore important to match the first derivatives of the curves either side of each knot. Applying this further constraint to the first knot, we get

$$\hat{\beta}_{0,1} = \hat{\beta}_{1,1} + 2 \cdot (\hat{\beta}_{1,2} - \hat{\beta}_{0,2}) \cdot t_1 + 3 \cdot (\hat{\beta}_{1,3} - \hat{\beta}_{0,3}) \cdot t_1^2 \tag{4.22}$$

So we can see that constraining the first derivatives to be equal over each knot results in the loss of another degree of freedom, and gives us one fewer parameter to optimise.

Applying these constraints may slightly reduce how well the regression curve fits the data. However, with the computational resources required to fit the regression model being proportional to the degrees of freedom, each constraint that we apply is beneficial to reduce the time and space requirements. We also end up with a curve which bears a far closer resemblance to how stock market prices actually vary. Therefore, in most cases, we consider it worthwhile to smooth over the knots as much as possible when fitting a piecewise polynomial regression curve.

But how smooth can we actually achieve? We can constrain the second derivatives to be the same at the knots, again reducing the degrees of freedom by 1. But setting the third derivatives to be equal results in the equation $6(\hat{\beta}_{1,3} - \hat{\beta}_{0,3}) = 0$ or in other words, $\hat{\beta}_{1,3} = \hat{\beta}_{0,3}$. This constrains the first two coefficients of t^3 to be equal, and likewise, following the same constraint at the next knot, enforces the third coefficient to be the same again, and so

on. Therefore, every coefficient of t^3 for all the different polynomials must be equal. This sets too big a constraint on the curve, and can result in a large loss in accuracy of fit.

Considering any higher derivatives is futile, as we simply get the equation $0 = 0$, and so the maximum smoothness we can get for a cubic piecewise polynomial whilst maintaining an accurate fit, is achieved by applying constraints at each knot, up to the second derivative. The same is true for a general piecewise polynomial of degree d, where we can apply a maximum of d continuity constraints.

Definition 4.2. *A continuous piecewise polynomial of degree d, which has continuity in the first $d-1$ derivatives at every knot, is known as a* ***degree-d spline****.*

Theorem 4.1. *A degree d spline with K knots has $(d+1)+K$ degrees of freedom.*

Proof. A degree d spline with K knots is composed of $K+1$ polynomials, each with $d+1$ parameters to optimise.

At each knot d degrees of freedom are removed by the continuity constraints.

We are therefore left with $(K+1)(d+1) - Kd = (d+1) + K$ degrees of freedom. □

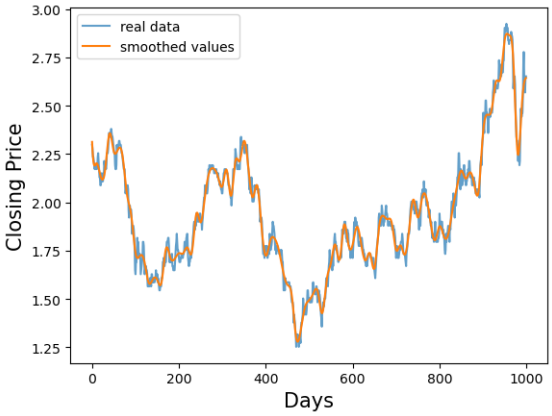

Figure 4.7: *Spline smoothing length 10 applied to stock prices*

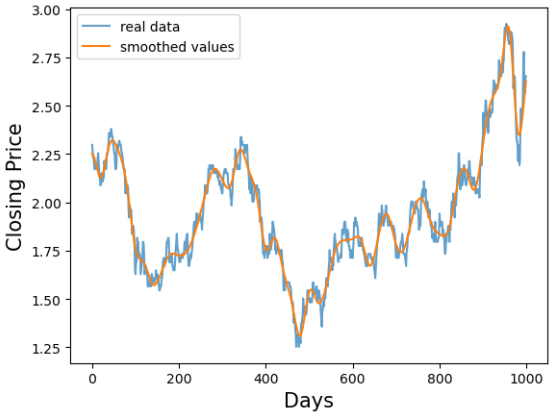

Figure 4.8: *Spline smoothing length 20 applied to stock prices*

(*figures created by author*)

As shown in figures 4.7 and 4.8, spline smoothing produces a curve which follows the trends in the data whilst removing the randomness of the day to day fluctuations, which are very difficult if not impossible to predict. However, this is much more computationally expensive than computing moving averages, and so the results need to be significantly better than those from using the moving average methods, to be able to justify this.

	length	Spline smoothing
MSE	5	0.5070
	10	1.0679
	15	1.9850
	20	2.2276
Bias	5	2485
	10	9099
	15	20340
	20	29316

Table 4.2: *The MSE and bias scores of different lengths of spline smoothing, where the length refers to the number of days in each section of the piecewise curve*

We can see from comparing table 4.2 with table 4.1 that spline smoothing generates very similar results to centered MA, whilst not relying on any future data. This is exactly what we were trying to achieve. Based on these results, we would expect spine smoothing to give the best results when used as a pre-processing layer to our transformer network.

4.2.3 Bagging

The other noise reduction method which we're going to explore in this section is known as bootstrap aggregating, or **bagging**. This involves training a number of different models on bootstrap samples of the original data, and then averaging the predictions of the set of models, to produce a final prediction.

Definition 4.3. *A **bootstrap sample** of a dataset size n is a random sample, with replacement, of size n taken from the original dataset.*

By taking bootstrap samples for each model, this ensures that no model overfits itself to the training set, and no two models are trained on exactly the same data, leading to variation between the different models. Bagging helps to reduce unnecessary noise in the predictions by averaging out inaccuracies caused by the model overfitting itself to noise in the input data.

4.3 Results

The following graphs display the test results for each of the different methods described in this chapter. To produce the results, a smoothed version of the test data was used as the network input, implementing the same smoothing method in each case, as the network was trained with. The blue lines show the real (unsmoothed) values that the network was trying to predict, and the orange lines show the predicted values. The graphs show the results for one particular stock (aal.us), whilst the performance results in table 4.3 are averages over a set of 20 stocks.

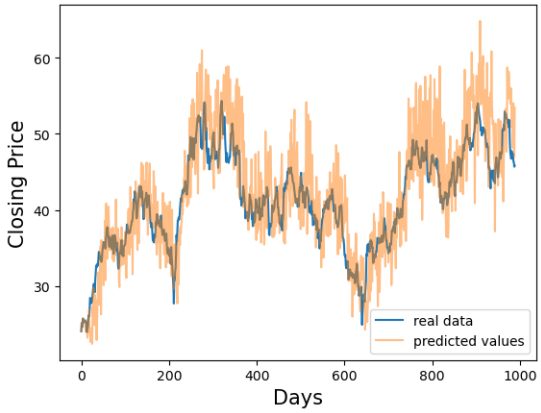

Figure 4.9: *Moving average length 5 results*

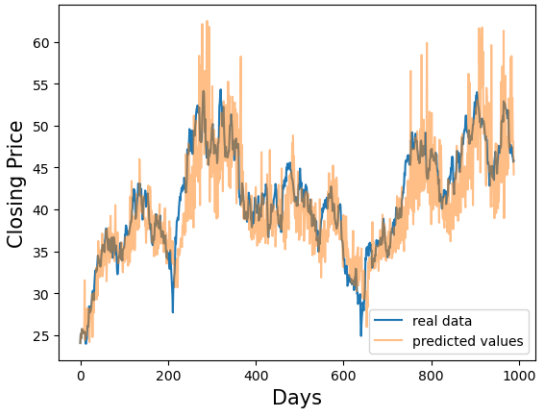

Figure 4.10: *Moving average length 10 results*

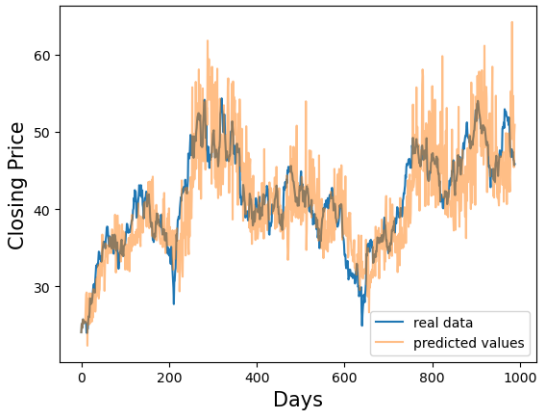

Figure 4.11: *Moving average length 20 results*

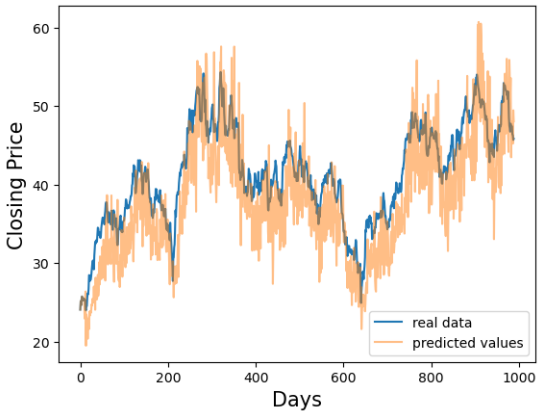

Figure 4.12: *Adjusted moving average length 5 results*

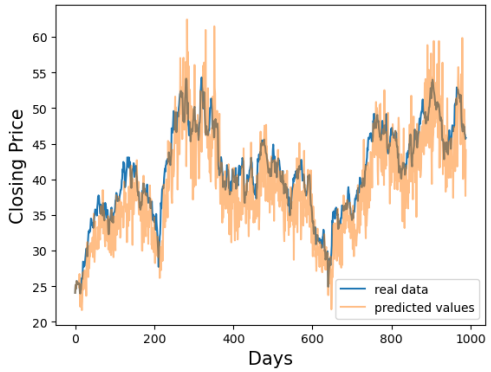

Figure 4.13: *Adjusted moving average length 10 results*

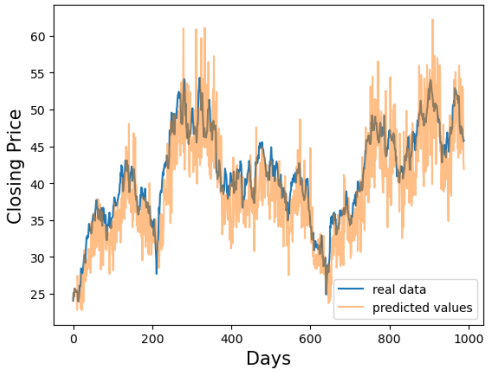

Figure 4.14: *Adjusted moving average length 20 results*

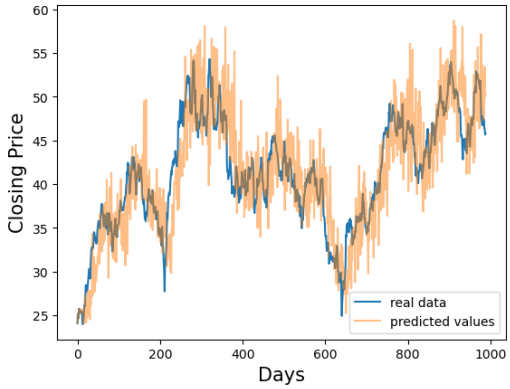

Figure 4.15: *Exponential moving average* $\alpha = 0.1$ *results*

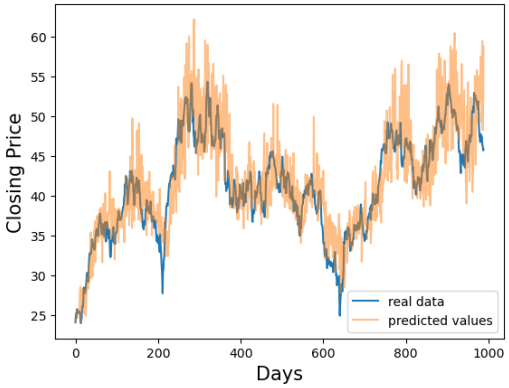

Figure 4.16: *Exponential moving average* $\alpha = 0.15$ *results*

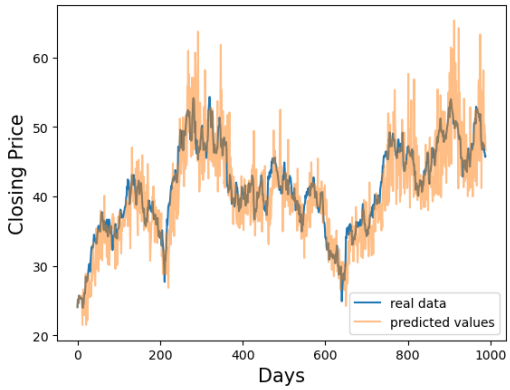

Figure 4.17: *Exponential moving average* $\alpha = 0.25$ *results*

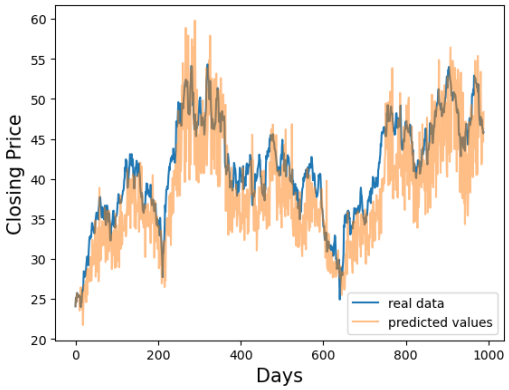

Figure 4.18: *Spline smoothing length 5 results*

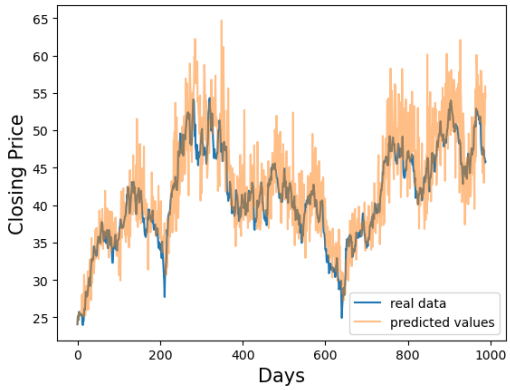

Figure 4.19: *Spline smoothing length 10 results*

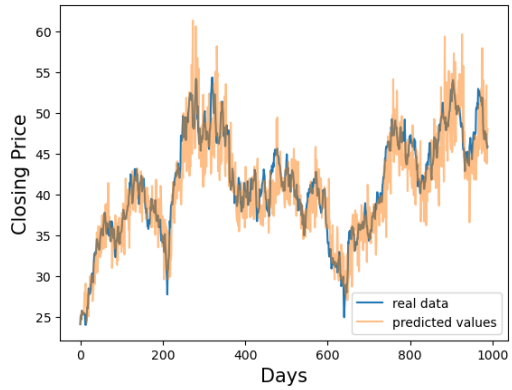

Figure 4.20: *Spline smoothing length 20 results*

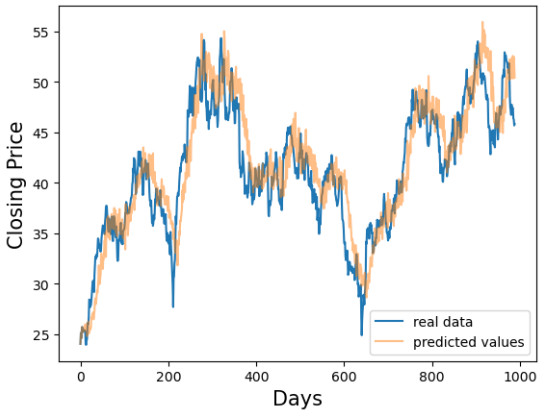

Figure 4.21: *Exponential moving average $\alpha = 0.1$ results with bagging*

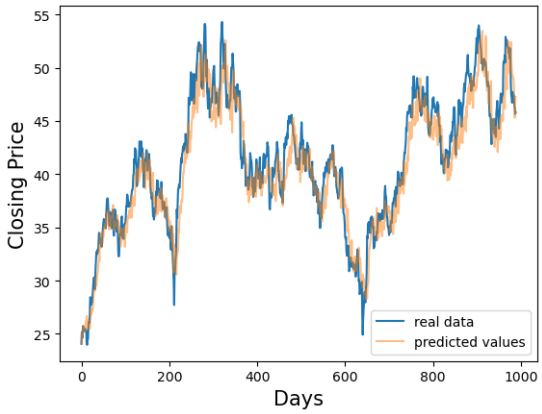

Figure 4.22: *Spline smoothing length 20 results with bagging*

	length	Moving av. (MA)	Adjusted MA	Spline		α	Exponential MA	
MSE	5	14.91	14.78	15.81		0.10	10.35	3.991
	10	14.76	14.15	16.05		0.15	13.62	
	20	15.63	14.34	9.973	2.454	0.25	14.55	
%profit	5	0.9653	1.023	1.052		0.10	1.150	1.015
	10	1.057	1.044	1.089		0.15	1.134	
	20	1.047	1.072	1.134	1.150	0.25	1.074	

Table 4.3: *The returns of the simulated portfolios using predictions from the above models. The R_t values are calculated as in equation (4.1), with the 'Return' values calculated as in equation (4.4). These values have been calculated by summing over a portfolio of 10 different stocks, which were pre-selected to have low volatility. The second set of values for spline 20 and exponential 0.1 are from the inclusion of bagging using 10 separate models.*

From the above graphs, it seems that the majority of smoothing methods were ineffective at reducing the noise seen in the original prediction results. This is backed up by the MSE results in table 4.3 in which we see that only the spline length 20 and exponential moving average with $\alpha = 0.1$ models were able to achieve a reduction in MSE from the original model. This is unexpected, as reducing the day-to-day variation in the training data means that the corresponding outputs for each input will be closer to the input values. This should mean that the network predictions fall closer to the values of the previous days, and therefore exhibit significantly reduced variation.

The failure to reduce variation in the predictions likely indicates that the hyperparameters in each of these networks require further tuning. Each different model may require individual optimisation as the different smoothing methods can affect the optimal values for the hyperparameters in different ways.

Furthermore, whilst we can see from table 4.3 that nearly every method resulted in a return larger than 1 (indicating a profit), the majority of methods were unable to produce a significantly larger return than that of the original model. This can be explained by our analysis of the smoothing methods themselves. As shown in tables 4.1 and 4.2, with the optimal results being displayed by the centered moving average and spline smoothing methods for comparison, none of the other methods were able to achieve the desired combination of a small bias value with a large MSE value. In other words, none of these other methods were able to significantly reduce the noise in the data without inducing a lag effect. The lag effect will have a detrimental effect on results, as it means that for the sections of data where this effect is observed, the model will be 'predicting' price changes which have already occurred rather than current price movements.

Contrary to expectations, the exponential moving average models with α values of 0.1 and 0.15 produced a significantly larger return than the other moving average methods, despite displaying high bias values in table 4.1. However, when bagging was applied to the model with $\alpha = 0.1$, the return result decreased to below most of the other moving average results. This result fits better with our suggestion that higher bias values, corresponding to

larger lag effects, should decrease the accuracy of results. We also note that in figure 4.21, there is a noticeable lag effect in the predictions. This observation, combined with the bagging return result shows that despite the initial elevated results of the exponential moving average, this smoothing method does not produce reliable predictions in general.

In contrast, the results for spline smoothing were initially high, whilst also being further improved by the introduction of bagging. This suggests that this type of smoothing method consistently produces an increased return compared to the initial network. Furthermore, the graph for this method closely follows the real data, showing that the predictions from this model were reliably accurate. This suggests that the elevated return values produced by this method were not a result of chance, but rather a consequence of higher quality predictions. The combination of the results in table 4.3 and figure 4.22 therefore indicate that the spline smoothing method of length 20 is effective for predicting stock prices when applied with bagging to a standard transformer model.

4.4 Concluding on the effectiveness of transformer networks and multi-head attention for stock market predictions

In the previous section, we showed that a transformer network trained on spline smoothed data can consistently produce accurate enough predictions such that we can achieve, on average, a 15% profit by running an investment strategy based on these predictions. Wang et. al were also able to show that their transformer network consistently performed better than the RNN, LSTM network, and convolutional neural network (CNN) designs, which are often associated with stock market predictions [5, Table 2, pg. 7].

The reason for the success of the transformer model over the other models is likely due to its attention mechanism. This is the defining feature of transformer networks which sets them apart from other types of sequence processing model. In the conclusion of their paper, Wang et. al suggest that the attention mechanism provides "an efficient way to capture important information while filtering out irrelevant noise" [5, Section 6., pg. 7]. It is likely that transformer networks do indeed filter out the noise of stock market data more effectively than RNNs and LSTM networks, as the multi-head attention relies less strongly on the relationships between consecutive inputs in the input sequence than RNNs and LSTM networks do. The ability of multi-head attention to attend to each of the inputs in the sequence equally may allow it to better identify the overall trends in the 10 day input sequences. This could be the key to the success of the transformer model in this application.

This theory that the multi-head attention mechanism is able to improve stock price predictions is backed up by a range of other research. Whilst there is a distinct lack of research into the use of entire transformer networks for stock market predictions, the use of the multi-head attention mechanism as part of other network designs appears to be an emerging area of research. In the past year, there has been a noticeable increase in the amount of research exploring this area, with several papers producing successful results. Li and Qian [25] applied the attention mechanism to an LSTM and GRU (similar to a RNN) network. Norasaed and Siriborvornratanakul [26] and Lai et. al [27] explored the combination of multi-head attention with different types of CNN network, where the model in [27] also uses attention in a preprocessing layer to reduce noise. Shuzhen Wang [28] looked into a combination of a CNN, an LSTM network, and a transformer network to make predictions. Tao, Wu and Wang [29] implemented an adapted transformer model. The results in these papers reinforce the idea that multi-head attention can be used effectively to improve stock market predictions.

Chapter 5

Further Applications Of Transformer Networks To Improve Predictions

In the previous chapter, we showed that transformer networks can be effective at improving predictions. Each of the networks discussed in section 4.4 implemented the multi-head attention mechanism to establish correlations between the entries in a sequence of consecutive stock prices. These correlations were then used with effect, as part of a larger network, to aid predictions. This is one area of emerging research into the use of multi-head attention for stock price predictions, however, this is not the only way that transformer mechanisms can be utilised to improve predictions. In this chapter, we discuss two more methods for applying transformer networks to stock market trading.

5.1 Alternative applications of transformer networks to effectively utilise multi-head attention.

In the previous chapter, we hypothesised that the multi-head attention mechanism was the main cause of the success that we observed when applying transformers to stock price predictions. This is also the defining feature of transformer networks which allows them to excel in natural language processing. When finding new applications for transformer networks in stock market trading we must therefore focus on methods that will use the multi-head attention mechanism to full effect. There are two main areas of research into alternative applications of multi-head attention in stock trading. Both look to use this mechanism to uncover new information about the stocks, reaching further than their individual historical prices.

The first method uses transformer networks for a task much more similar to that which they were designed to do. It applies the natural language processing capabilities of these networks to identify and interpret news articles and social media posts which may be related to stock price movement. When used as part of larger networks, this can allow models to bridge the gap between fundamental and technical analysis, taking advantage of both historical price trends and recent external events to make predictions.

To understand how news articles and social media posts can relate to stock prices, we first recall from chapter 3 that stock prices do not depend solely on events occurring within the companies themselves, but also depend on external factors such as interest rates, government policies, industry trends, currency exchange rates, and a whole variety of others. National and global news events such as Brexit and the Covid-19 pandemic can also cause significant shifts in stock market prices. These factors, occurring in real time, are difficult to account for using technical analysis of past data. However, information about these events can be found in news articles and social media posts, which can be used to increase the accuracy of prediction models.

Transformer networks have been shown in several research papers to provide an effective method of identifying and interpreting these articles and posts for use in stock prediction models [30] [31] [32]. The advanced capability of these networks for natural language processing tasks, makes them the natural choice for this application. The multi-head attention mechanism allows them to interpret these sources much more effectively than other network designs.

Transformer networks can also analyse text far more quickly than humans, allowing them to search through a wider range of material than we are capable of. By using transformer networks for this method, it is therefore possible to provide the prediction models with a wide selection of information regarding recent events and current market rumours, which can indicate shifts in investor sentiment. We recall from chapter 3 that there is usually a lag period between occurring events and the corresponding shifts in stock prices. The increased speed of analysis brought about by using transformer networks as opposed to human analysis, can often allow the models to make stock movement predictions related to recent events before the stock prices have adjusted.

The other emerging area of research into transformers in stock market trading involves using the multi-head attention mechanism to draw correlations between different stocks. This allows price predictions for each stock to be based on not only the individual stock's data, but also on the data from a variety of other stocks. It has been widely hypothesised that using this method to provide a larger quantity of meaningful input data for each prediction, should result in an increase in the stability and prediction accuracy of stock prediction models. However, as eluded to in [33, Abstract], [34, Abstract] and [35, Abstract], the current methods for capturing stock correlations are not entirely effective.

The multi-head attention mechanism may provide a new way to better capture the relationships between different stocks. The reason that the transformer network was able to outperform its predecessors in natural language processing stems from the ability of mutli-head attention to equally attend to each of the entries in the input sequence. This allows transformers to establish long-term correlations between inputs as effectively as it captures short-term dependencies. This is a property that RNNs and LSTM networks were unable to achieve. Due to this property, transformer networks are able to capture each of the relationships between a range of different stocks with equal consideration .

Using multi-head for this application allows for many different stock correlations to be captured in parallel. This means that training time is significantly reduced compared to models which analyse the data sequentially, allowing for the input of a larger number of stocks at once. In other words, correlations can be considered over a wider range of different stocks. The results in [6], [36] and [37], show that multi-head attention can indeed be used effectively in this application. For the remainder of this report, we will further explore the use of transformer networks for stock correlations.

5.2 Applying a transformer network to capture stock correlations

In this section, we introduce another network design, initially proposed by Yoo et al. in [6]. This network aims to make stock movement predictions using stock correlations. Section 5.2.1 is based on [6, Section 3], whilst the remaining sections are original work.

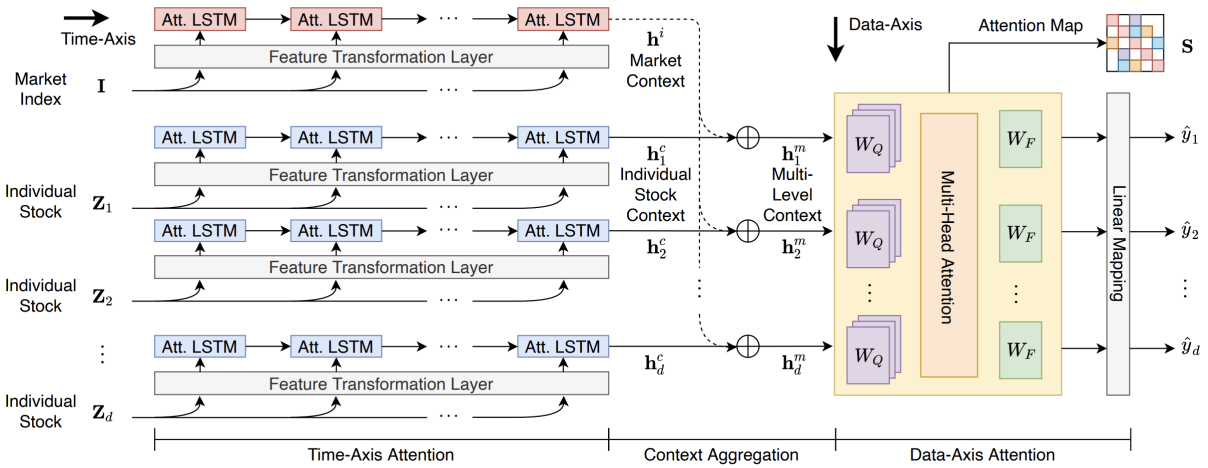

Figure 5.1: *Network proposed by Yoo et al. in [6]*

5.2.1 Network design

In their network design, Yoo et al. combine elements from both LSTM and transformer networks to create the network shown in figure 5.2.1. This network takes in a selection of data from d different stocks, as well as market index data (see definition 5.1 below). The aim is to draw correlations between the different stocks and use these correlations to predict whether the stock price will go up or down between the current time step and the next day.

Definition 5.1. *A **market index** tracks the overall performance of a market or group of assets by simulating a portfolio of a pre-defined selection of assets, which are seen to represent the overall market in question.*

For each stock u, each day is inputted as a feature vector $z_{u,t}$ composed of a set of 11 ratios:

1. $z_{open} = \frac{open_t}{close_t} - 1$
2. $z_{high} = \frac{high_t}{close_t} - 1$
3. $z_{low} = \frac{low_t}{close_t} - 1$
4. $z_{close} = \frac{close_t}{close_{t-1}} - 1$
5. $z_{adj\_close} = \frac{adj\_close_t}{adj\_close_{t-1}} - 1$

6–11. $z_{dk} = \frac{\sum_{i=0}^{k} adj\_close_{t-i}}{k \cdot adj\_close_t} - 1 \qquad for\ k \in \{5, 10, 15, 20, 25, 30\}$

see appendix A for definitions of these variables.

These feature vectors are then embedded into a higher dimension in the feature transform layer before being passed as inputs into the attention LSTM network, which aims to produce a single stock context vector for each stock.

The attention LSTM layer initially operates in the same way as a standard LSTM network, only differing in the processing of the outputs:

- First, each cell output h_i is given an attention score $\alpha_i = \frac{exp(h_i^T h_T)}{\sum_{j=1^T} exp(h_j^T h_T)}$
- The final output for each stock is then computed as $\tilde{h^c} = \sum_{i=0}^{T} \alpha_i h_i$

Here, the attention score determines how important each input is with regards to the final input, and helps to improve the ability of the LSTM network to retain long-term dependencies.

The stock context vectors are then normalised, to increase the stability of the next stages in the process. Each is then combined with the market index context vector using $h_u^m = h_u^c + \beta h^i$. This allows correlations to be influenced by the individual stocks' relationship with the overall market, which improves the ability of the multi-head attention block to effectively capture these stock correlations.

Finally, the vectors are passed as inputs to the multi-head attention block which draws correlations between the different stocks. This is structured in exactly the same way as the multi-head attention blocks we have previously seen in this report, with the exception that the attention map $S = softmax(\frac{QK^T}{\sqrt{h}})$ is also released as an output, describing how much each stock contributes to the predictions of the other stocks. In the same way as transformers, the standard outputs of the multi-head attention block are then passed through a feedforward network, with residual connections and layer normalisation around the multi-head attention block and feedforward network. This section is therefore structured in the same way as a single iteration of the main transformer encoder block.

The output is the product of one final layer, $\hat{y} = \sigma(H_p W_p + b_p)$ where H_p is the output of the encoder block, and σ represents the sigmoid function. $\hat{y}$ represents the probability that the price will increase by the closing of the stock market the following day.

5.2.2 Results

Despite the promising results described by Yoo et al. while using this network structure, my implementation of the same structure only exhibited a prediction success rate of 50.18% when tested on a large dataset of 50 stocks, using one year of data from each. Furthermore, when tested on smaller subsets of this data, it produced results ranging from 48% to 52%, including 49.19% for the set of stocks that it was trained with, showing that this network cannot be consistently relied upon to produce a success rate larger than 50%. In addition, when performing the simulation strategy detailed in the previous chapter, it produced a mean percentage profit of -0.29 over the set of 50 stocks, and -0.22 over the training stocks.

		Actual Values	
		0	1
Predicted Values	0	10089	10440
	1	7747	8229

Table 5.1: *The predictions made by my network on a test dataset of 50 stocks*

		Actual Values	
		0	1
Predicted Values	0	771	842
	1	672	695

Table 5.2: *The predictions made by my network on a test dataset of data from the stocks used for training*

As a silver lining to these somewhat poor results, we can note that a network of this type has only failed to be useful if it is unreliable as to whether it will produce a positive profit or not. This network seems to consistently produce a negative return, and so by simply reversing the predictions, we can achieve an average percentage profit of 0.33.

However, the fact remains that the network has not performed as it was designed to do, and has struggled to predict stock movements with any accuracy. This could be due to a number of reasons. Firstly, we note that if we accept the random walk theory (theory 3.1) to be generally correct, then it should be very difficult to achieve a prediction accuracy of significantly larger than 50%. However, if this was the only problem we would expect the prediction results to be balanced between predictions of 1 and 0. Instead we observe a bias towards predictions of 0. This indicates that there may be an issue with the network itself.

The depth of the network may be an issue here. In my implementation, I used sequences of 10 days of feature vectors as input data. This means that the network consisted of a 10-layer LSTM network, along with a single multi-head attention block with a final feedforward layer. This is not a particularly deep network and may not provide enough complexity to deal with the data provided. It is possible that results could be improved by inputting longer sequences of data, to increase the depth of the LSTM network. However, this may not be effective as the increasing quantity of input data to be analysed may counteract the benefit of extending the network. It may therefore be better to modify the second part of the network.

We note that transformer networks pass their data through the multi-head attention blocks a number of times to ensure that each input can fully attend to the others in the input sequence. In this network, the data is only passed through a single multi-head attention block to compute the stock correlations. Further iterations of the multi-head attention may improve the accuracy of the stock correlations. Analysis of the attention scores outputted from the model gives further evidence for this hypothesis.

Figure 5.2 shows an example of the attention scores outputted by the network during inference. We can immediately see that the noise in the results is very high. This represents an issue with the mechanism of the network, as each attention score is calculated using 10 days of input data, meaning that consecutive attention scores share 9 out of 10 input days. Whilst the LSTM network will interpret these shared days slightly differently for these two consecutive input sequences, due to the differing positions in the sequence, we would still expect the correlations between the stocks to be similar for consecutive inputs. If the sequences were passed through further iterations of the multi-head attention block, we would expect the resulting attention scores to be more stable, resulting in a reduction in the noise of the results.

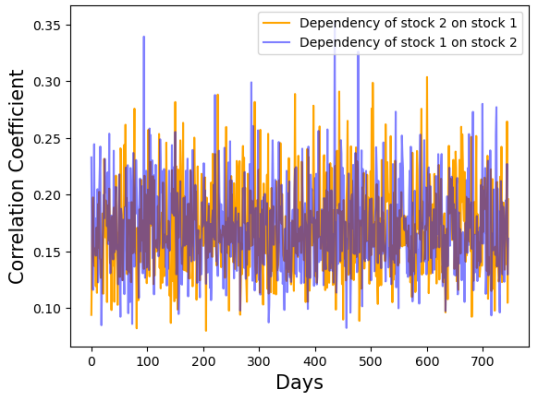

Figure 5.2: *Attention scores showing the dependencies between two stocks over time*

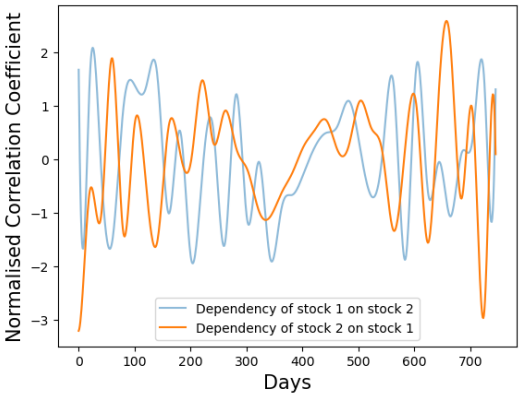

Figure 5.3: *Normalised and smoothed attention scores showing the dependencies between two stocks over time*

We also note that the values in figure 5.2 are higher than we might expect. To create this graph, 5 stocks were inputted into the network. These results suggest that on average around 1/6 of the variation of stock 1 can be determined using the data from stock 2 and vice versa. Furthermore, the diagonal entries of the attention matrices, representing the reliance of each stock's movement on its own data, were only 0.31 on average. This seems very low, as we would expect each stock's own data to form the majority of the weighted averages used to predict its movement.

Further to this, in general, it seems that the correlation coefficients representing the dependency of stock 1 on stock 2 moved in opposite directions to those representing the dependency of stock 2 on stock 1. We can observe this in figure 5.3, where we have provided smoothed and normalised curves using the same data as in figure 5.2. This is unexpected, as we would expect that during periods of high correlation between the two stocks, both sets of attention scores should increase in general, and vice versa. These points again indicate that the single iteration of multi-head attention may not have been able to fully capture the nature of the the correlations between stocks.

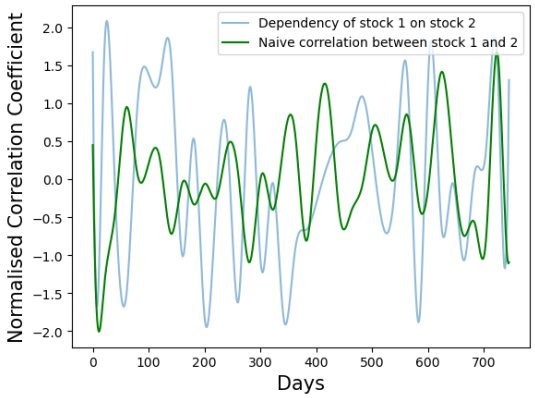

Figure 5.4: *A comparison of the attention scores from stock 1 with the results from a naive correlation method*

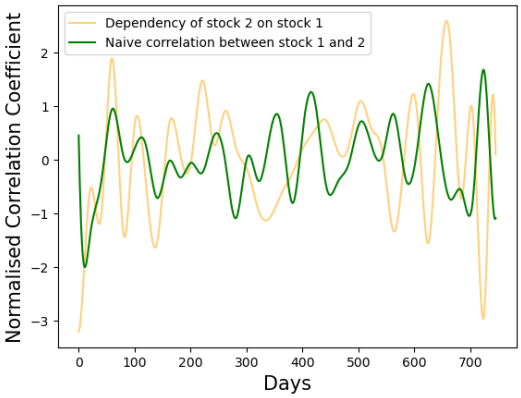

Figure 5.5: *A comparison of the attention scores from stock 2 with the results from a naive correlation method*

However, when comparing the attention scores to a naive method of calculating stock correlation, we notice distinct similarities. This naive approach involves giving a score of 1 to days in which the stock prices both move in the same direction, and a score of -1 in the opposite case. The scores are then summed over a moving window of 10 days, where each result is then associated with the final day in the sequence of 10. These scores are then smoothed and normalised in the same way as the attention scores were. Figures 5.4 and 5.5 show the results. We notice that the large peaks and troughs of the attention scores mostly correspond with peaks and troughs in the results from the naive method. This suggests that while the attention scores produced by the network may not be perfect, they are at least giving a rough indication of the time periods in which correlations are strongest and weakest.

Another important consideration, to describe the results of this network, is that, as mentioned in [36], "stock correlations often occur momentarily and in a cross-time manner". This means that a simple comparison of 10 days of

"time-aligned" data may not be very effective at capturing stock correlations. Using this method, we fail to capture any "cross-time" correlations. The network will also learn parameters to represent the general correlation of the input stocks over the entire training set. Whilst this may successfully capture long term relationships between stocks, it is not an effective way to model momentary correlations, where the nature of the correlations may change significantly over long time periods.

However, despite the shortcomings of this model design, figures 5.4 and 5.5 indicate that there may be some useful information contained within the attention scores outputted by this network, meaning that we may still be able to use the network effectively for applications other than direct prediction.

5.3 Combining Designs

Despite not achieving the results that we might have hoped for in the previous section, we may be able to use the attention scores produced by the network to improve the prediction accuracy of other models.

To test this hypothesis, we can implement these attention scores as a second set of inputs to our model from chapter 4. For clarity we will refer to our original network as "network 1" and the correlations network as "network 2". To combine the networks, we expand network 1 to take a set of 5 different stocks as inputs, and produce 5 separate outputs. We then input the same data into both network 1 and network 2, and simply multiply the attention scores, produced by network 2, by the embedded inputs of network 1. We then pass these through the remainder of network 1 as normal.

Using this design, we produced the predictions shown in figures 5.6 and 5.7. We can observe that the additional input of stock correlations has successfully stabilised the model, almost entirely removing the excess noise in the predictions. The predictions do not quite follow the real data as closely as those in figure 4.20, and the bagging model here was not able to achieve as high a return, with a percentage profit of 0.11. However, these lower results can almost certainly be explained by the sub-optimal attention scores produced by the correlation network. There is plenty of scope to improve the quality of the stock correlation attention scores, using the methods described in the previous section. These improvements would likely be able to improve this combined model to the extent that we could produce results significantly higher than those seen in chapter 4.

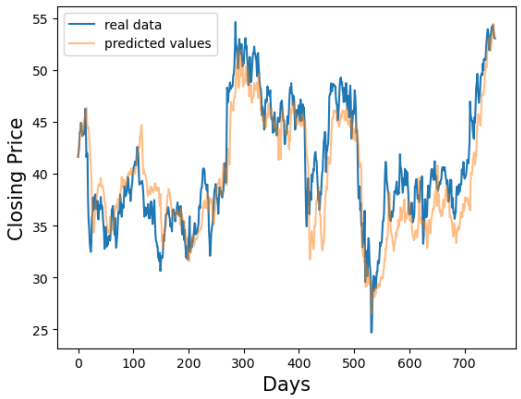

Figure 5.6: *A comparison of the attention scores from stock 1 with the results from a naive correlation method*

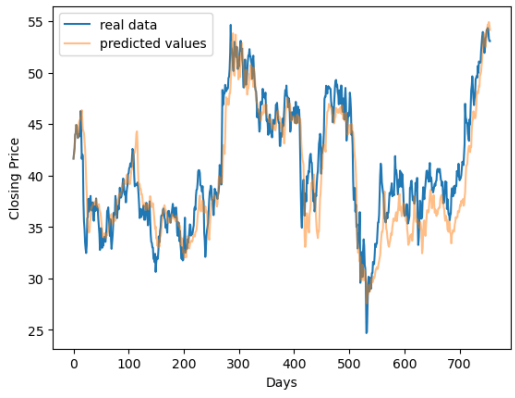

Figure 5.7: *A comparison of the attention scores from stock 2 with the results from a naive correlation method*

We may be able to further improve this model by considering not only the correlations between the stocks during the current input time-frame, but also the long term correlations between the input stocks. By analysing the patterns in correlations over time, it may be possible to give a better indication of how the stocks may correlate in the current time-frame.

Chapter 6

Conclusion

In chapter 4 we showed that a simple and direct use of transformers for modelling stock market prices can produce both profitable results and predictions which closely follow the real data. These results are backed up by the success of Wang et al. in their application of the same model in [5]. We were able to provide some level of noise reduction to the original network predictions via a combination of spline smoothing and bagging, which helped to stabilise the model, and improve its performance. Introducing stock correlation data produced by the network design in [5] was successful in further stabilising the model, and was able to almost entirely remove the excess noise in the predictions. Whilst this was not able to improve the performance of the initial model in terms of the return achieved when implementing the simulated trading strategy detailed in chapter 4, we are confident that improvements to the method of generating the stock correlation data would result in a significant improvement to this model's performance.

In chapters 4 and 5, we were able to show that transformer networks can indeed be effective at predicting stock market prices, even when implementing them in relatively simple ways. We also noted that there has recently been an increase in the quantity of research into the use of multi-head attention for stock market trading. The majority of this research has concurred with our findings on the effectiveness of multi-head attention for stock market predictions. It seems that the parallelised structure of the multi-head attention block can produce high quality results for a wide variety of sequence processing tasks, whether it be direct time-series analysis of stock prices, capturing correlations between stocks, or even scraping news and social media sites for information related to stock movement. With research into using transformers for stock correlations and news scraping only emerging very recently, and already achieving some success, it seems likely that these networks may start to be used more widely in the field of stock market trading in the near future.

Bibliography

[1] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention Is All You Need. 2017. arXiv:1706.03762 [cs].

[2] Marc Evrard. Transformers in Automatic Speech Recognition. In Mohamed Chetouani, Virginia Dignum, Paul Lukowicz, and Carles Sierra, editors, *Human-Centered Artificial Intelligence: Advanced Lectures*, Lecture Notes in Computer Science ((LNAI,volume 13500)), pages 123–139. Springer International Publishing, 2023.

[3] Ch. Sita Kameswari, Kavitha J, T. Srinivas Reddy, Balaswamy Chinthaguntla, Senthil Kumar Jagatheesaperumal, Silvia Gaftandzhieva, and Rositsa Doneva. An Overview of Vision Transformers for Image Processing: A Survey. *International Journal of Advanced Computer Science and Applications*, 14(8), 2023.

[4] Sanghyuk Roy Choi and Minhyeok Lee. Transformer Architecture and Attention Mechanisms in Genome Data Analysis: A Comprehensive Review. *Biology 2023*, 12(1033), July 2023.

[5] Chaojie Wang, Yuanyuan Chen, Shuqi Zhang, and Qiuhui Zhang. Stock Market Index Prediction using Deep Transformer Model. *Expert Systems with Applications*, 208:118–128, December 2022.

[6] Jaemin Yoo, Yejun Soun, Yong-chan Park, and U Kang. Accurate Multivariate Stock Movement Prediction via Data-Axis Transformer with Multi-Level Contexts. In *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining*, KDD '21, pages 2037–2045, New York, NY, USA, August 2021. Association for Computing Machinery.

[7] Durham University. Math3431 - Machine Learning and Neural Networks III Michaelmas Term Course Notes. 2023/24.

[8] J J Hopfield. Neural networks and physical systems with emergent collective computational abilities. *Proceedings of the National Academy of Sciences*, 79(8):2554–2558, April 1982.

[9] Robert DiPietro and Gregory D. Hager. Chapter 21 - Deep learning: RNNs and LSTM. In S. Kevin Zhou, Daniel Rueckert, and Gabor Fichtinger, editors, *Handbook of Medical Image Computing and Computer Assisted Intervention*, The Elsevier and MICCAI Society Book Series, pages 503–519. Academic Press, January 2020.

[10] Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. On the difficulty of training Recurrent Neural Networks. February 2013. arXiv:1211.5063 [cs].

[11] Noah Weber. Why LSTMs Stop Your Gradients From Vanishing: A View from the Backwards Pass. *weberna's blog*, November 2017.

[12] M. Mattheakis and P. Protopapas. Recurrent Neural Networks: Exploding, Vanishing Gradients & Reservoir Computing.pdf, March 2019.

[13] D. Snow. Historical Note of Machine Learning in Finance and Economics, January 2021.

[14] Sepp Hochreiter and Jürgen Schmidhuber. Long Short-term Memory. *Neural computation*, 9:1735–80, December 1997.

[15] Raymond Cheng. LSTM Text Classification Using Pytorch. *Medium*, July 2020.

[16] Timo Denk. Linear Relationships in the Transformer's Positional Encoding, January 2019.

[17] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural Machine Translation by Jointly Learning to Align and Translate, May 2016. arXiv:1409.0473 [cs, stat].

[18] Sho Takase, Shun Kiyono, Sosuke Kobayashi, and Jun Suzuki. On Layer Normalizations and Residual Connections in Transformers, June 2022. arXiv:2206.00330 [cs] version: 1.

[19] Ruibin Xiong, Yunchang Yang, Di He, Kai Zheng, Shuxin Zheng, Chen Xing, Huishuai Zhang, Yanyan Lan, Liwei Wang, and Tie-Yan Liu. On Layer Normalization in the Transformer Architecture, June 2020. arXiv:2002.04745 [cs, stat].

[20] Liyuan Liu, Xiaodong Liu, Jianfeng Gao, Weizhu Chen, and Jiawei Han. Understanding the Difficulty of Training Transformers, October 2023. arXiv:2004.08249 [cs, stat].

[21] Douglas K. Pearce. Stock prices and the economy. *Federal Reserve Bank of Kansas City - Economic Review*, November 1983.

[22] Burton G Malkiel. A RANDOM WALK DOWN WALL STREET - The Time-Tested Strategy for Successful Investing. 1973.

[23] Eugene F. Fama. Efficient Capital Markets: A Review of Theory and Empirical Work. *The Journal of Finance*, 25(2):383–417, 1970.

[24] Oddmund Groette. Trading Bias And Behavioral Mistakes– The Most Common Trading Biases: Strategies to Overcome Them - Quantified Strategies. March 2024. arXiv:2312.15235v1.

[25] Chengyu Li and Guoqi Qian. Stock Price Prediction Using a Frequency Decomposition Based GRU Transformer Neural Network. *Applied Sciences*, 13(1):222, January 2023. Number: 1 Publisher: Multidisciplinary Digital Publishing Institute.

[26] Market movement prediction using chart patterns and attention mechanism | Discover Analytics.

[27] Shijie Lai, Mingxian Wang, Shengjie Zhao, and Gonzalo R. Arce. Predicting High-Frequency Stock Movement with Differential Transformer Neural Network. *Electronics*, 12(13):2943, January 2023. Number: 13 Publisher: Multidisciplinary Digital Publishing Institute.

[28] Shuzhen Wang. A Stock Price Prediction Method Based on BiLSTM and Improved Transformer. *IEEE Access*, 11:104211–104223, 2023.

[29] Zicheng Tao, Wei Wu, and Jianxin Wang. Series decomposition Transformer with period-correlation for stock market index prediction. *Expert Systems with Applications*, 237:121424, March 2024.

[30] Yawei Li, Shuqi Lv, Xinghua Liu, and Qiuyue Zhang. Incorporating Transformers and Attention Networks for Stock Movement Prediction. *Complexity*, 2022:e7739087, February 2022. Publisher: Hindawi.

[31] Jincheng Hu. Local-constraint transformer network for stock movement prediction. *International Journal of Computational Science and Engineering*, 24(4):429–437, January 2021.

[32] Jintao Liu, Hongfei Lin, Xikai Liu, Bo Xu, Yuqi Ren, Yufeng Diao, and Liang Yang. Transformer-Based Capsule Network For Stock Movement Prediction. In Chung-Chi Chen, Hen-Hsen Huang, Hiroya Takamura, and Hsin-Hsi Chen, editors, *Proceedings of the First Workshop on Financial Technology and Natural Language Processing*, pages 66–73, Macao, China, August 2019.

[33] Jingwei Hong, Ping Han, Abdur Rasool, Hui Chen, Zhiling Hong, Zhong Tan, Fan Lin, Steven X. Wei, and Qingshan Jiang. A Correlational Strategy for the Prediction of High-Dimensional Stock Data by Neural Networks and Technical Indicators. In Yuan Tian, Tinghuai Ma, Qingshan Jiang, Qi Liu, and Muhammad Khurram Khan, editors, *Big Data and Security*, pages 405–419, Singapore, 2023. Springer Nature.

[34] Ruirui Liu, Haoxian Liu, Huichou Huang, Bo Song, and Qingyao Wu. Multimodal multiscale dynamic graph convolution networks for stock price prediction. *Pattern Recognition*, 149:110211, May 2024.

[35] Haodong Han, Liang Xie, Shengshuang Chen, and Haijiao Xu. Stock trend prediction based on industry relationships driven hypergraph attention networks. *Applied Intelligence*, 53(23):29448–29464, December 2023.

[36] Tong Li, Zhaoyang Liu, Yanyan Shen, Xue Wang, Haokun Chen, and Sen Huang. MASTER: Market-Guided Stock Transformer for Stock Price Forecasting. December 2023.

[37] Zhen Yang, Tianlong Zhao, Suwei Wang, and Xuemei Li. MDF-DMC: A stock prediction model combining multi-view stock data features with dynamic market correlation information. *Expert Systems with Applications*, 238:122134, March 2024.

[38] ARUNMOHAN_003. Transformer from scratch using pytorch, 2023. Available online at `https://kaggle.com/code/arunmohan003/transformer-from-scratch-using-pytorch`.

[39] Andrej Karpathy. The spelled-out intro to neural networks and backpropagation: building micrograd, August 2022. Available at `https://www.youtube.com/watch?v=VMj-3S1tku0`.

[40] Udacity. intro-to-pytorch, 2021. Available at `https://github.com/udacity/deep-learning-v2-pytorch/blob/master/intro-to-pytorch/Part%206%20-%20Saving%20and%20Loading%20Models.ipynb`.

Appendices

A Stock market price data

Stock exchanges are not open for 24 hours a day, but typically open for between 6 and 9 hours per day, and only on weekdays. This set up leads to the four main categories that historical stock price data is usually grouped into:

- **Opening price** - the price of each stock when the market opens in the morning
- **Low price** - the lowest price each stock reaches during a trading day
- **High price** - the highest price each stock reaches during a trading day
- **Closing price** - the price of each stock when the market closes for the day

Other values which are also normally provided include:

- **Volume** - the total number of shares traded in each stock during a trading day
- **Adjusted closing price** - a version of the closing price adjusted for corporate actions such as dividends or **stock splits** (where the company releases more stocks via the primary market).

Other values may be provided, but none of these others are relevant to this report.

B Code used in this report

Below is a link to the github repository containing the code used to produce the results in chapters 4 and 5. Large parts of the code were based on code in [38], [39] and [40]. Some code was copied directly from these sources.

`https://github.com/barneytodd/Project-III`