Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

Parallelisation of Differential Neural Networks

In partial fulfillment of the requirements for the degree of Master of Science in the School of Computer Science and Statistics

Name:	**Maurice Kiely**
Student ID:	**23350601**
Supervision:	**Niall O'Sullivan**

Date of Submission:	**03-09-2024**
Project Code:	Differential Machine Learning

Declaration concerning plagiarism

I hereby declare that this thesis is my own work where appropriate citations have been given and that it has not been submitted as an exercise for a degree at this or any other university.

I have read and I understand the plagiarism provisions in the General Regulations of the University Calendar for the current year, found at `http://www.tcd.ie/calendar`.

Name:	**Maurice Kiely**
Student ID:	**23350601**
Signature:	[illegible]
Date:	**03-09-2024**

Acknowledgements

I would like to extend my sincerest thanks to all those who have helped me to this stage of my M.Sc.

I would like to begin by expressing my gratitude to my supervisor, Niall O'Sullivan. His consistent support and invaluable insights were crucial to the successful completion of this project. Without him, this work would not have been possible.

Next, I would like to extend my sincere thanks to the M.Sc. coordinator, Kirk Soodhalter, for his exceptional management of the course from its outset. Kirk's approach to teaching and his engaging interaction with the M.Sc. students created an environment far more interesting and engaging than I could have ever anticipated.

Additionally, I would like to thank my fellow M.Sc. students and friends. Whether it was tackling challenging problems together or simply offering encouragement during tough times, their presence played a pivotal role in my growth and learning throughout the year.

Finally, I would like to extend my sincerest thanks to my parents for their unwavering support from the outset of not just the M.Sc., but my academic life. Their encouragement has been a constant source of motivation, their financial support has made the pursuit of my goals possible, and their endless patience—especially during the more challenging times—has been truly invaluable. I couldn't have done it without them, even if they might still be hoping for a day when my academia will translate into something they can easily explain to their friends.

Abstract

Differential Machine Learning is an extension of supervised learning that trains models on both standard labels and the differential of the standard labels with respect to the inputs. This type of model has many uses in the financial world where the speed of training is crucial. Training a Differential Neural Network involves steps that are suited to shared memory parallelism and adjoint differentiation. The aim of this paper is to derive the process of backpropagation for the differential training and implement both a serial and parallel version of the model. This will be done with the implementation of shared memory parallelism by using multi-threading and efficient memory management and the models will be trained using synthetically generated European call option data. The resulting performance improvements and scalability will be analyzed to demonstrate the advantages of parallelisation of the model.

Github Implementation: *https://github.com/mauricekiely/Differential_ Neural_ Network*

Contents

List of Algorithms

List of Figures

2 Introduction

Neural Networks

Neural Networks are a class of predictive machine-learning models that have seen a significant rise in popularity in recent years. Once a relatively niche concept, these models have become increasingly integrated into daily life. With widespread adoption across areas such as image recognition with Convolutional Neural Networks (CNNs)—and Natural Language Processing (NLP)—utilizing Recurrent Neural Networks (RNNs)—the impact of Neural Networks is undeniable.

One industry that has undergone significant transformation due to Neural Networks is finance. Traditional methods, such as fundamental analysis of financial statements have increasingly given way to sophisticated quantitative models driven by machine learning algorithms. These advanced models are now integral to decision-making processes in financial markets, reflecting a paradigm shift in how investment strategies are developed and executed. Neural Networks have been a fundamental component of this shift.

A Neural Network is modelled on the human brain. It consists of interconnected units called Neurons, which are grouped into layers. These neurons process input "training data", through the network, and adjust the connections between these Neurons based on the training data. Over time, as the network is trained, it learns to recognize patterns in the data, enabling it to make decisions by reinforcing connections that lead to correct outcomes and mitigating those otherwise.

Neural Network Structure

A Neural Network is trained using data composed of input features in the input layer and corresponding labels in the output layer. The input data is generally denoted $X_{train} \in \mathbb{R}^{m \times n}$ where m is the number of training features and n is the number of data points. Label data is typically denoted $Y_{train} \in \mathbb{R}^{k \times n}$ for classification problems where k is the number of possible outcomes and $k = 1$ for regression problems.

Between these two, there are one or more "hidden layers", where nodes in these layers are responsible for learning complex representations of the input data. The term "deep" in a deep Neural Network refers to the presence of multiple hidden layers between the input and output layers. A deeper network, with more hidden layers, allows the model to capture and learn more features, making it capable of modelling more complex patterns in the data.

Each neuron in a hidden layer computes a weighted sum of its inputs and then passes the result through an activation function. The activation function introduces non-linearity, allowing the network to model complex patterns that linear transformations cannot. Some typical activation functions include ReLU, Sigmoid and Tanh. During training, the network adjusts its weights and biases to minimize the error between predicted outputs and actual labels. The final weights and biases should in turn give a more accurate estimate of the outcome of unseen data.

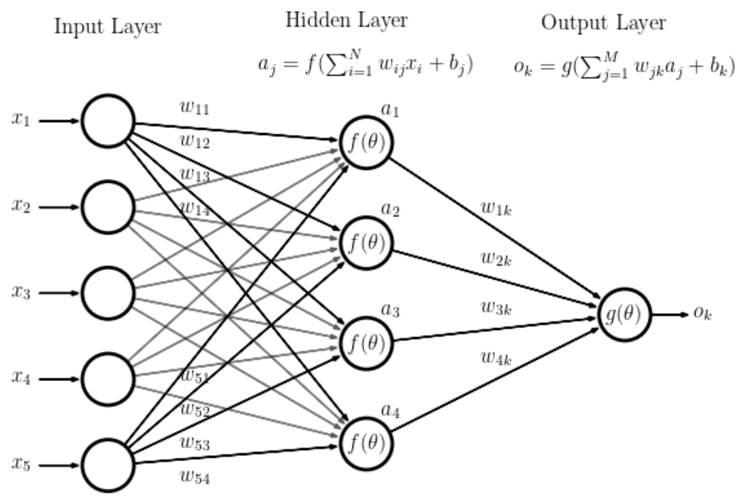

Figure 1: Basic Neural Network

Feedforward Equations

The feedforward equation in a Neural Network is the process of getting a prediction based on the input data and current weights and biases for each node. Each connection between nodes has a weight and each node has a bias.

For 2 layers $(l, l-1)$ that are fully connected, the weight matrix between them is denoted $W_l \in \mathbb{R}^{n_l \times n_{l-1}}$ where $w_{jk}^{(l)} = Node_k^{(l-1)} \to Node_j^{(l)}$.

Each Node has an input value (pre-activated (z)) and output value (activated (a)). The pre-activated value is simply a linear combination of the activated values of the previous layer.

$$z_i^{(l)} = \sum_{j=1}^{n_{l-1}} w_{ij}^{(l)} \cdot a_j^{(l-1)} + b_i^{(l)} \tag{2.0.1}$$

The activated value is then obtained by applying an activation function σ element-wise to the pre-activated value:

$$a_i^{(l_i)} = \sigma(z_i^{(l_i)}) \tag{2.0.2}$$

The activation function (σ) provides the model with the ability to capture non-linear patterns of the training data. Here $a^{(0)} = X_{train}$ and the output of the final layer $a^{(L)} = \hat{y}$. This represents the model's predictions and is used to compute the cost function (C), typically Mean Squared Error (MSE).

Backpropagation

The Neural Network trains by adjusting its weights and biases to minimise the cost function. To do this, we need to find $\frac{\partial C}{\partial W}$ and $\frac{\partial C}{\partial B}$ for each layer. This can be done efficiently by differentiating each step of the feedforward equation and using the chain rule on the adjoints to efficiently compute each partial derivative:

We work backwards along the feedforward equations

$$Z_l = W_l A_{l-1} + B_l$$
$$A_l = \sigma(Z_l)$$

and apply the chain rule. For the output layer L, the derivative of the cost function with respect to the weighted input Z_L is given by:

$$\frac{\partial C}{\partial Z_L} = \frac{\partial C}{\partial A_L} \odot \sigma'(Z_L),$$

where $\odot$ denotes the Hadamard product (element-wise multiplication), and $\sigma'(Z_L)$ is the derivative of the activation function with respect to Z_L.

This derivative is also known as the error term for the output layer, denoted by δ_L:

$$\delta_L = \frac{\partial C}{\partial A_L} \odot \sigma'(Z_L).$$

For the hidden layers $l < L$, the error term is propagated backward as:

$$\delta_l = \left(W_{l+1}^T \delta_{l+1}\right) \odot \sigma'(Z_l),$$

where W_{l+1}^T is the transpose of the weights of the next layer.

Using the error term δ_l, the partial derivatives with respect to the weights and biases are:

$$\frac{\partial C}{\partial W_l} = \delta_l A_{l-1}^T,$$

$$\frac{\partial C}{\partial B_l} = \delta_l.$$

Once these derivatives are computed, they are used to adjust the weights by some scalar learning rate (η) to decrease the difference between $\hat{Y}$ and Y.

$$W_l \leftarrow W_l - \eta \frac{\partial C}{\partial W_l} \tag{2.0.3}$$

$$B_l \leftarrow B_l - \eta \frac{\partial C}{\partial B_l} \tag{2.0.4}$$

This process is called Gradient Descent as we are attempting to reach the lowest point of the Cost function. Generally, η is found using one of several optimising functions (normally ADAM). Some of these introduce a momentum term to avoid the data converging to any local minima and correctly converge. This hazard is called overfitting and it can be mitigated using regularization techniques like penalty terms, drop-out layers or cross-validation.

By combining the feedforward equations with the backpropagation, we can efficiently train a simple Neural Network. However, as aforementioned, Neural Networks take many complex forms. From Convolutional Networks and Recurrent Networks to Generative Adversarial Networks, these models can take many complex forms, each with its respective advantages.

2.1 Twin-Network

The core focus of this paper is a model called a Twin Neural Network. This model is introduced by the paper "Differential Machine Learning"[4], which will discussed in more detail later. The model combines the feedforward equations and a form of backpropagation to train on both standard labels (Y_{train}) and differential labels ($Z_{train} = \frac{\partial Y}{\partial X}$). This structure allows the model to capture the sensitivity of predictions to changes in the input data and provide these sensitivities as additional outputs once trained. This makes it particularly effective for tasks where understanding the underlying sensitivities is crucial.

Twin-Network Structure

As stated, this network trains on both labels and differential labels. As a result, the structure of the model differs slightly from what we previously looked at. The Twin-Network performs similarly to 2 interconnected simple Neural Networks that share the same weights.

The feedforward component of this network is split into 2 processes. The first computes the standard feedforward prediction to find $\hat{Y}$. The second uses a form of backpropagation to compute a prediction for the sensitivities $\frac{\partial \hat{Y}}{\partial X}$. This dual process allows the network to train on output labels as well as sensitivities to inputs which significantly enhances the model's predictive accuracy and robustness.

Once both values are predicted, the backpropagation can be done to update W_l and B_l. The weights are shared and mirrored by both sides of the feedforward network. Thus, during backpropagation, each weight is updated twice

$$C\left(\{w_l, b_l\}_{l=1,\dots,L}\right) = \alpha \cdot \mathrm{MSE}_Y + \beta \cdot \mathrm{MSE}_Z$$

where MSE_Y is the mean squared error of the output predictions and MSE_Z is the mean squared error of the sensitivity predictions. Each weight and bias is updated proportionally based on these derivatives.

This type of training has many benefits including increased accuracy and more efficient learning which will be discussed further shortly.

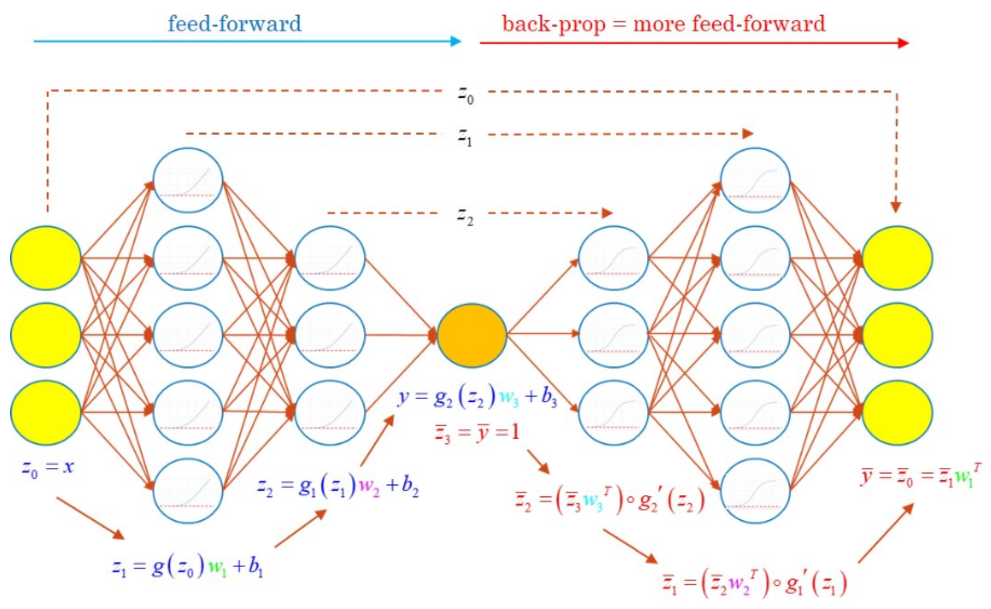
feed-forward
back-prop = more feed-forward
z_0
z_1
z_2
$z_0 = x$
$z_1 = g(z_0) w_1 + b_1$
$z_2 = g_1(z_1) w_2 + b_2$
$y = g_2(z_2) w_3 + b_3$
$\bar{z}_3 = \bar{y} = 1$
$\bar{z}_2 = (\bar{z}_3 w_3^T) \circ g_2'(z_2)$
$\bar{z}_1 = (\bar{z}_2 w_2^T) \circ g_1'(z_1)$
$\bar{y} = \bar{z}_0 = \bar{z}_1 w_1^T$

Figure 2: Twin-Network

Feedforward Equations

The feedforward process in this network can be broken into 2 components:

The first component is the same as the feedforward of a standard Neural Network. We begin with the X_{train} values and pass these forward through the network, applying activation functions, to reach the predicted $\hat{Y}$. This component uses equations 2.0.1 and 2.0.2 to compute the standard label prediction.

The second component involves calculating the adjoints of each A_l and Z_l with respect to Y to compute $\frac{\partial Y}{\partial X}$. This is done by working backwards along the first component of the feedforward equation and using the chain rule as follows:

$$\frac{\partial Y}{\partial Y} = \frac{\partial Y}{\partial A_L} = 1 \tag{2.1.1}$$

$$\frac{\partial Y}{\partial Z_l} = \frac{\partial Y}{\partial A_l} \cdot \frac{\partial A_l}{\partial Z_l} = \frac{\partial Y}{\partial A_l} \odot \sigma'(Z_l) \tag{2.1.2}$$

$$\frac{\partial Y}{\partial A_{l-1}} = \frac{\partial Y}{\partial Z_l} \cdot \frac{\partial Z_l}{\partial A_{l-1}} = W_l^T \cdot \frac{\partial Y}{\partial Z_l} \tag{2.1.3}$$

By working back through the pre-activated and activated values of each node, we can deduce $\frac{\partial Y}{\partial X}$ and use this to compute the cost function listed above.

BackPropagation

Like the feedforward process, the training for this type of Network can also be broken into 2 components. Firstly, the training of the weights and biases on $\frac{\partial Y}{\partial X}$ and then on Y. This is done by splitting the cost function C into its 2 respective components and differentiating each respectively:

$$\frac{\partial C}{\partial W} = \alpha \cdot \frac{\partial MSE_Y}{\partial W} + \beta \cdot \frac{MSE_{\frac{\partial Y}{\partial X}}}{\partial W}$$

$$\frac{\partial C}{\partial B} = \alpha \cdot \frac{\partial MSE_Y}{\partial B} + \beta \cdot \frac{MSE_{\frac{\partial Y}{\partial X}}}{\partial B}$$

The differentiation of the standard labels (Y) is relatively simple. The logic is the same as that discussed in the previous section, with each derivative simply being:

$$\frac{\partial MSE_Y}{\partial W_l} = \delta_l A_{l-1}^T,$$

$$\frac{\partial MSE_Y}{\partial B_l} = \delta_l.$$

,

However, the complexity of this network comes from the training of the differential components.

Unlike standard labels, there are significant interdependencies between the weights on each layer due to differential calculations involving $\left\{\frac{\partial Y}{\partial X_l} = Z_l\right\}_{l=L,L-1,\ldots,k+1}$ for each weight W_k. Because of this, the derivatives with respect to the weights are not simply predetermined matrices acquired by the forward equations; instead, we must take a more advanced approach.

These derivatives cannot be found using matrix multiplication alone, as we have seen in simpler cases. To calculate these derivatives, we must approach the problem element-wise. The first step in this process is to identify the component of the cost function that we are differentiating with respect to, also on an element-wise basis.

In this case, consider the cost function with dimensions $\mathbb{R}^{n_0 \times N}$, where N represents the number of training points, and n_0 represents the number of input parameters. An entry in the cost function, therefore, is expressed as:

$$\text{Cost}_{ij} = \frac{(\hat{Z}_{ij} - Z_{ij})^2}{n},$$

where the indices i and j correspond to the specific elements in the respective matrices.

To determine the derivatives of this cost function with respect to the weights, we must utilize tensors. This approach captures the norms of every entry in the derivative of the cost function with respect to a corresponding weight element W_{ab}.

Let us denote the weight matrix at layer k as $W^{(k)}$, which has dimensions $n_k \times n_{k-1}$. The derivative of the cost function can now be considered a tensor operation, where each element in the resulting tensor is influenced by multiple elements from the weight matrices.

The derivative of the cost function with respect to an individual weight $W_{ab}^{(k)}$ can be expressed as:

$$\frac{\partial \text{Cost}_{ij}}{\partial W_{ab}^{(k)}} = \frac{2}{n}\left(\hat{Z}_{ij} - Z_{\text{train},ij}\right) \cdot \frac{\partial \hat{Z}_{ij}}{\partial W_{ab}^{(k)}}.$$

We will investigate this $\frac{\partial \hat{Z}_{ij}}{\partial W_{ab}^{(k)}}$, derivative later. However, the derivative of the cost function with respect to a particular weight $W_{ab}^{(k)}$ can be visualized using a tensor. The structure of the tensor that represents the derivative of the cost function with respect to each element of $W^{(k)}$, consider the following:

$$\begin{pmatrix}
\begin{pmatrix} \frac{\partial \text{Cost}_{11}}{\partial W_{11}^{(k)}} & \cdots & \frac{\partial \text{Cost}_{1N}}{\partial W_{11}^{(k)}} \\ \frac{\partial \text{Cost}_{21}}{\partial W_{11}^{(k)}} & \cdots & \frac{\partial \text{Cost}_{2N}}{\partial W_{11}^{(k)}} \\ \vdots & \ddots & \vdots \\ \frac{\partial \text{Cost}_{n_0 1}}{\partial W_{11}^{(k)}} & \cdots & \frac{\partial \text{Cost}_{n_0 N}}{\partial W_{11}^{(k)}} \end{pmatrix} &
\begin{pmatrix} \frac{\partial \text{Cost}_{11}}{\partial W_{12}^{(k)}} & \cdots & \frac{\partial \text{Cost}_{1N}}{\partial W_{12}^{(k)}} \\ \frac{\partial \text{Cost}_{21}}{\partial W_{12}^{(k)}} & \cdots & \frac{\partial \text{Cost}_{2N}}{\partial W_{12}^{(k)}} \\ \vdots & \ddots & \vdots \\ \frac{\partial \text{Cost}_{n_0 1}}{\partial W_{12}^{(k)}} & \cdots & \frac{\partial \text{Cost}_{n_0 N}}{\partial W_{12}^{(k)}} \end{pmatrix} & \cdots &
\begin{pmatrix} \frac{\partial \text{Cost}_{11}}{\partial W_{1n_k}^{(k)}} & \cdots & \frac{\partial \text{Cost}_{1N}}{\partial W_{1n_k}^{(k)}} \\ \frac{\partial \text{Cost}_{21}}{\partial W_{1n_k}^{(k)}} & \cdots & \frac{\partial \text{Cost}_{2N}}{\partial W_{1n_k}^{(k)}} \\ \vdots & \ddots & \vdots \\ \frac{\partial \text{Cost}_{n_0 1}}{\partial W_{1n_k}^{(k)}} & \cdots & \frac{\partial \text{Cost}_{n_0 N}}{\partial W_{1n_k}^{(k)}} \end{pmatrix} \\
\vdots & \ddots & \vdots & \vdots \\
\begin{pmatrix} \frac{\partial \text{Cost}_{11}}{\partial W_{n_{k-1}1}^{(k)}} & \cdots & \frac{\partial \text{Cost}_{1N}}{\partial W_{n_{k-1}1}^{(k)}} \\ \frac{\partial \text{Cost}_{21}}{\partial W_{n_{k-1}1}^{(k)}} & \cdots & \frac{\partial \text{Cost}_{2N}}{\partial W_{n_{k-1}1}^{(k)}} \\ \vdots & \ddots & \vdots \\ \frac{\partial \text{Cost}_{n_0 1}}{\partial W_{n_{k-1}1}^{(k)}} & \cdots & \frac{\partial \text{Cost}_{n_0 N}}{\partial W_{n_{k-1}1}^{(k)}} \end{pmatrix} &
\begin{pmatrix} \frac{\partial \text{Cost}_{11}}{\partial W_{n_{k-1}2}^{(k)}} & \cdots & \frac{\partial \text{Cost}_{1N}}{\partial W_{n_{k-1}2}^{(k)}} \\ \frac{\partial \text{Cost}_{21}}{\partial W_{n_{k-1}2}^{(k)}} & \cdots & \frac{\partial \text{Cost}_{2N}}{\partial W_{n_{k-1}2}^{(k)}} \\ \vdots & \ddots & \vdots \\ \frac{\partial \text{Cost}_{n_0 1}}{\partial W_{n_{k-1}2}^{(k)}} & \cdots & \frac{\partial \text{Cost}_{n_0 N}}{\partial W_{n_{k-1}2}^{(k)}} \end{pmatrix} & \cdots &
\begin{pmatrix} \frac{\partial \text{Cost}_{11}}{\partial W_{n_{k-1}n_k}^{(k)}} & \cdots & \frac{\partial \text{Cost}_{1N}}{\partial W_{n_{k-1}n_k}^{(k)}} \\ \frac{\partial \text{Cost}_{21}}{\partial W_{n_{k-1}n_k}^{(k)}} & \cdots & \frac{\partial \text{Cost}_{2N}}{\partial W_{n_{k-1}n_k}^{(k)}} \\ \vdots & \ddots & \vdots \\ \frac{\partial \text{Cost}_{n_0 1}}{\partial W_{n_{k-1}n_k}^{(k)}} & \cdots & \frac{\partial \text{Cost}_{n_0 N}}{\partial W_{n_{k-1}n_k}^{(k)}} \end{pmatrix}
\end{pmatrix}$$

By using this element-wise approach and tensor representation, we can accurately compute the gradients required for backpropagation, even in the presence of complex interdependencies between layers. We take a norm of each inner matrix to quantify these gradients.

Once these derivatives are calculated, we can piece together the 2 components of the cost function derivatives to get values

$$\frac{\partial C}{\partial W} = \alpha \cdot \frac{\partial \text{MSE}Y}{\partial W} + \beta \cdot \frac{\partial \text{MSE}\frac{\partial Y}{\partial X}}{\partial W} \qquad \frac{\partial C}{\partial B} = \alpha \cdot \frac{\partial \text{MSE}Y}{\partial B} + \beta \cdot \frac{\partial \text{MSE}\frac{\partial Y}{\partial X}}{\partial B}$$

$$W_l \leftarrow W_l - \eta \frac{\partial C}{\partial W_l} \qquad B_l \leftarrow B_l - \eta \frac{\partial C}{\partial B_l}$$

Benefits of Twin-Network

The use of a twin-network introduces several advantages. The biggest being the fact that, once trained, the model can compute input sensitivities as quickly as a standard output with precision. This removes the need for computationally intensive Monte Carlo simulations for various derivative calculations.

Another prominent benefit is that it trains on both the shape and the points of the function it attempts to approximate. This dual focus enables the model to capture trends and relationships within the data that might otherwise be lost. This added training information results in a more accurate prediction of the standard labels of the model.

The use of these differentials also mitigates the risks of overfitting. Overfitting, which occurs when a model becomes too closely aligned with the training data to the point where its performance on unseen data becomes worse, is a common pitfall in machine learning. The Twin-Network combats this by also learning the slope of the target function and using it as a regularizer.

This regularization method does not introduce bias into the model, unlike other types. Regularization techniques often work by imposing penalties on the complexity of the model. This can shift the model's predictions away from the optimal solution. In contrast, differential training within a twin-network framework refines the model's learning process without such drawbacks, allowing it to remain unbiased while still avoiding overfitting.

By training the model on both the function's values and its derivatives, it requires less training data, while also becoming more adaptable to new, unseen data. This is particularly important in contexts where consistent and dependable predictions are essential, such as in financial modelling or risk management, which we will explore in more detail shortly.

2.2 Financial Applications

The financial sector is one of the most changed by the machine learning wave as it is an area that is arguably more driven by data than any other. Firms with the most cutting-edge technology generally perform exponentially better than those that are marginally behind. As a result, a financial application of a Twin-Network implementation is a good example to showcase the benefits of this model.

2.3 Differential Machine Learning Paper

The Differential Machine Learning Paper[4] was briefly mentioned earlier as the foundation of this project. This paper explains the importance of the computation of derivatives for the pricing of financial instruments. Typically this is done by Monte Carlo, with Automatic Adjoint Differentiation (AAD)[3] being a recent technique to speed up this process. However, these methods are still quite computationally intensive. The difference with the Twin-Network, is that, although the set-up and training of the model are computationally intensive, once the training is complete, the model can take inputs and almost instantly compute both an accurate product price, as well as all of the input sensitivities.

The paper provides an example of this model for a 5F Bermudan Swaption, a financial derivative with a high level of complexity that cannot be modelled quickly by normal means. Traditionally the price is modelled using Monte Carlo, but this requires heavy computation for each required derivative price. The paper implements the Twin-Network with the use of Tensorflow in Python (**see here**) to train a model that can take inputs and return a derivative price as well as the derivatives of the 5 factors used. The paper efficiently showcases the power of this Differential Neural Network in the context of financial modelling.

The example I will be training my Twin-Network on is that of a simple, synthetically generated European Call Option based on the Black-Scholes formula. I chose this as the training data can be generated quickly with a closed-form solution, and because these derivatives are commonly used in various trading applications. This makes them widely recognized and ideal for showcasing the model's capabilities while still allowing it to perform at its full potential.

Black-Scholes Theory

A European Call Option is a financial contract that gives the holder the right but not the obligation to buy some underlying financial product for some price K (called the strike) at some future time T.

In 1973, Fischer Black and Myron Scholes published the paper which introduced the Black-Scholes model[2]. This model provides a theoretical framework for pricing European options and has become the cornerstone of modern financial derivatives pricing. The model is built on the foundation of stochastic calculus and relies on the assumption of a lognormal distribution of asset prices, making it mathematically robust and widely applicable. Despite its many limitations, the Black-Scholes model remains a widely used tool in the financial industry for the valuation of options.

The model is based on 5 parameters:

1. S: The current price of the underlying asset.
2. K: The strike price.
3. T: The time to maturity of the contract.
4. r: The risk-free interest rate, (usually FED rate).
5. σ: The volatility of the underlying asset's returns.

The Black-Scholes formula for the price of a European call option C is given by:

$$V = S\Phi(d_1) - Ke^{-rT}\Phi(d_2)$$

where

$$d_1 = \frac{\ln\left(\frac{S}{K}\right) + \left(r + \frac{\sigma^2}{2}\right)T}{\sigma\sqrt{T}} \qquad d_2 = d_1 - \sigma\sqrt{T}$$

Greeks

The Greeks are derivatives of this formula that are commonly used in the context of options trading. They are used to give insight into the underlying behaviour of the options value. The 5 we will use in the model are:

Delta (Δ): $\Delta = \frac{\partial V}{\partial S} = \Phi(d_1)$

Dual Delta: $\frac{\partial V}{\partial K} = -\Delta$

Vega (ν): $\nu = \frac{\partial V}{\partial \sigma} = S\phi(d_1)\sqrt{T}$

Theta (Θ): $\Theta = \frac{\partial V}{\partial t} = -\frac{S\phi(d_1)\sigma}{2\sqrt{T}} - rKe^{-rT}\Phi(d_2)$

Rho (ρ): $\rho = \frac{\partial V}{\partial r} = KTe^{-rT}\Phi(d_2)$

European-Call Twin-Network

The analysis performed in this paper will be done on a model trained on a standard European Call option and its first-order derivatives. The target value will be the price of a European Call option generated by implementing the Black-Scholes formula and randomly generating values and their Greeks:

$$S \sim U[90, 110], \quad K \sim U[95, 105], \quad T \sim U[0.1, 3.0], \quad \sigma \sim U[0.05, 0.30], \quad r \sim U[0.001, 0.05]$$

The model consisted of 2 fully-connected hidden layers with 8 nodes in each. It was trained on $N = 1000$ and $N = 5000$ synthetically generated data points and was split into 80% Training Data and 20% Validation Data. Each time the model was trained it was run for 150 Epochs, and variables included; noise-level, trainingRate and λ where $\alpha = (\frac{1}{\lambda+1})$ and $\beta = (\frac{\lambda}{\lambda+1})$. Once trained, this model should be able to take inputs for S, K, T, σ, ρ and output a derivative price V and the 5 sensitivities

We will analyse results from various combinations of noise, and lambda to see which models perform the best in which conditions.

3 Problem Implementation

The implementation of this problem can be divided into the following sections: the Matrix class, the Layer class, the Neural Network class, the ForwardPropagation algorithms, and the BackPropagation algorithms. For the most part, each of these sections builds on from the last, beginning with the Matrix class.

3.1 Matrix Class

Due to the structure of the network, matrix operations are the obvious method of dealing with the construction and training. Nearly every component of the project can be represented as a matrix or vector, including every weight matrix and each set of data, whether activated, pre-activated, or otherwise.

The Matrix class consists of only 3 member variables which are; the number of rows, number of cols and a vector where the entries of the matrix are stored. With performance and memory efficiency in mind, the matrix entries are stored contiguously in memory. This layout improves cache performance during operations, enabling faster matrix computations.

```
template <class T>
class Matrix {
    size_t myRows, myCols;
    vector<T> myVector;

public:
    // Constructors
    Matrix() : myRows(0), myCols(0) {}

    Matrix(const size_t rows, const size_t cols) : myRows(rows),
        myCols(cols), myVector(rows * cols) {}

    Matrix(const size_t rows, const size_t cols, const T val) :
        myRows(rows), myCols(cols), myVector(rows * cols, T(val)) {}
};
```

Since the matrix entries are stored using `std::vector<double>`, there is no need to implement the Rule of Three or Rule of Five. The copy and move operations generated by the compiler already manages memory and resources appropriately.

The complexity of the Matrix class lies in the member functions. Alongside some simple accessor functions, the class has a number of more complex member functions. As the Neural Network will be implemented to use matrix operations, there must be an implementation of these operations to allow for efficient readable code. Most of these need to be implemented to handle Matrix-Matrix, Matrix-Vector and Matrix-Scalar operations. For the most part, these use operator overloading for readability. For example:

```cpp
Matrix operator+(T scalar) const {
    Matrix result(myRows, myCols);
    transform(myVector.begin(), myVector.end(),
       result.myVector.begin(), [scalar](T val) { return val +
       scalar; });
    return result;
}

Matrix operator+(const vector<T>& vec) const {
    Matrix result(myRows, myCols);
    for (size_t i = 0; i < myRows; ++i) {
        for (size_t j = 0; j < myCols; ++j){
            result[i][j] = (*this)[i][j] + vec[i];
        }}
        return result;
    }
Matrix operator+(const Matrix& rhs) const {
    Matrix result(myRows, rhs.num_cols());
    for (size_t i = 0; i < myRows; ++i){
        for (size_t j = 0; j < rhs.num_cols(); ++j) {
            result[i][j] = (*this)[i][j] + rhs[i][j];
            }}
        return result;
    }
```

The class also consists of member functions for `.transpose()`, `.dot(const Matrix&)` as well as `elementwiseMultiply(const Matrix&)` along with some operator overloads for scalar, vector and Matrix operations (+, -, *, [ ]). The use of contiguous memory allows these operations to be easily parallelised, with the incorporation of SIMD being possible for various functions.

This Matrix class will serve as the basis for the following sections.

3.2 Layer Class

As described earlier, the Neural Network is composed of multiple layers, each with distinct components. These include weights, biases, and both pre-activated and activated data. Each layer has an activation function and the number of nodes it contains. In the program, the Layer class was implemented with these in mind.

Each Layer consists of a variety of member variables. These include size, index in the Network as well as Matrices for each weight, activated data and pre-activated set of data. Each of these matrices also has a corresponding derivative, either with respect to the cost function, or the output. Each layer also consists of a bias vector and its derivative.

The Layer class has the following structure:

```cpp
class Layer {
    // Layer Size and Index
    size_t  myNNodes,  myLayerIdx;
    // Matrices
    Matrix<double>  myX,    myZ,    myW;
    vector<double>  myB;
    // Respective Derivatives
    Matrix<double>  mydYdX, mydYdZ, mydCdW;
    vector<double>  mydCdB;
public:
    Layer(const vector<size_t>& layerSizes, const size_t layerIdx,
        const size_t n) :
```

Upon initialization, each layer is passed a `layerSizes` vector as well as an index. This vector contains the size of each layer in order. This is used to construct each `Matrix` with the correct dimensions. The parameter `n` is the argument for the number of data points being passed through each layer.

Using this, a `Layer` (l) can be constructed with `myX, myZ` $\in \mathbb{R}^{n_l \times n}$ and `myW` $\in \mathbb{R}^{n_l \times n_{l-1}}$.

To help with the mitigation of variance in training, the weights are initialized using Xavier Initialization, $W_l \sim \mathcal{N}\left(0, \frac{2}{n_{l-1}+n_l}\right)$. This is done using a `weightInitializer()` member function.

Aside from accessor functions and the weight initializer, there are no member functions in the Layer class. The primary goal of this class is to simplify the training process by organizing all components of the Neural Network. This ensures that both forward and backward propagation are optimized for performance. By keeping all data relatively close in memory, the Layer class reduces the overhead of locating scattered data and improves memory access efficiency. Additionally, by localizing computations within the Layer class, the network benefits from enhanced cache locality.

These Layers are the building blocks of the Network and allow for efficient training and readable code.

3.3 NeuralNetwork Class

The `NeuralNetwork` class is the core component of this project. It is the model being trained to produce accurate predictions based on the provided training data. The class consists of Matrices and Vectors to hold each set of training and test data for X, Y and Z. The network also initializes matrices for predicted Y and Z and the MSEs required to quantify the error.

However, the Network is built around a vector of layers `vector<Layer> myLayers`, where the computation involved in training is done. Training the model involves passing data forward and backwards through this vector. `myLayers` consists of `myNLayers - 1` Layers, due to the omission of the input layer. Note, this is because the input layer does not require a weight matrix or activation data as these are encapsulated by `myLayers[0].myW()` and X_{train} respectively.

```
class NeuralNetwork {
    // Number of Layers and respective sizes
    size_t                myNLayers;
    vector<size_t>        myLayerSizes;

    // Matrices and Vectors for all relevant training data
    Matrix<double>        myXTrain,    myZTrain,    myZPred, myXTest,
        myZTest;
    vector<double>        myYTrain,    myYPred, myYTest;

    // Vector of Layers. Core component of Network
    vector<Layer>         myLayers;

    // Hold MSE of trainig data (Not needed, just for debugging really)
    double                YMSE,        ZMSE;
```

Beyond the standard member functions for data access and propagation, two functions are worth discussing. The first is `updateWeights()`, which iterates over all the layers in the network and updates the weights and biases based on the derivatives found. This is just an implementation of equations 2.0.3 and 2.0.4.

The second is `.train(size_t epochs, double trainingRate, double lambda)`. This function orchestrates the entire training process. It repeatedly executes the forward and backward propagation steps over a specified number of epochs and adjusts the model iteratively. This causes the weights and biases to converge to the set that yield the most accurate predictions.

Together, these components and functions enable the `NeuralNetwork` class to effectively learn from the training data, adjusting its internal parameters to provide increasingly accurate predictions over time.

FeedForward Functions

The forward propagation in the network is divided into two components: the prediction of Y and the prediction of Z. The prediction of Y is carried out according to equations 2.0.1 and 2.0.2, iterating through each layer until the output layer is reached.

For this process, the Softplus activation function is used. Softplus is chosen over ReLU due to the necessity of a second derivative during the backpropagation phase. Specifically for the Z components, which is undefined for ReLU at $x = 0$. Softplus is simply a continuous approximation for ReLU. This smoothness reduces the likelihood of dead neurons that can occur with ReLU. Additionally, the availability of higher-order derivatives with Softplus allows for the calculation of the second-order derivatives required for the backpropagation of the Z component.

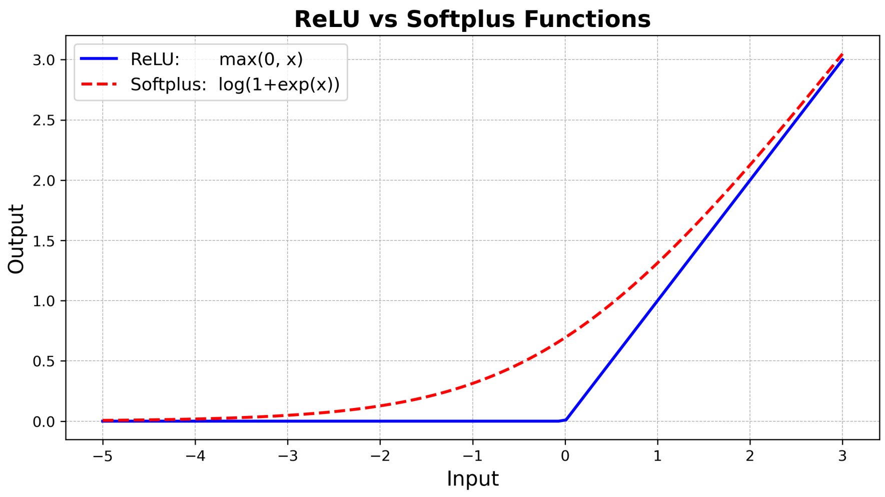

Figure 3: Softplus vs ReLU

Using this process, we can deduce a prediction for the labels ($\hat{Y}$), and with this, we can compute MSE_Y. This Prediction will be used to adjust the weights and biases accordingly.

Algorithm 1 Forward Propagation of Y

```
Require: $X_{train}$ is a Matrix of $n$ training points, each with $n_0$ training features.
1: procedure FORWARDPROP($X_{train}$)
2:     $X^{(0)} = X_{train}$
3:     for $l = 1, .., L$ do
4:         $Z^{(l)} = W^{(l)} \cdot X^{(l-1)} + B^{(l)}$
5:         $X^{(l)} = \sigma(Z^{(l)})$
6:     end for
7:     $Y_{pred} = X^{(L)}$
8: end procedure
```

The second part of the forward propagation is the prediction of Z ($\frac{\partial Y}{\partial X}$). This process is more like a standard backpropagation than another forward equation as it involves using the chain rule and calculation of adjoints to compute the final derivative. Note, there is no need for the use of a tape or other data structure as the computation is straightforward and can be done simply using loops.

The calculation of this is done according to 2.1.3 and 2.1.2, where $\sigma'(x) = \frac{e^x}{(1+e^x)}$. Like the forward propagation of Y, we work through each layer, however here, we reverse the process and work from the output to the input layer to calculate $\hat{Z}$ and in turn, MSE_Z.

Algorithm 2 Forward Propagation of Z

```
1: procedure ForwardProp
2:     $\frac{\partial Y}{\partial X^{(L)}} = 1$
3:     for $l = L, .., 1$ do
4:         $\frac{\partial Y}{\partial Z^{(l)}} = \sigma'\left(Z^{(l)}\right) \odot \frac{\partial Y}{\partial X^{(l)}}$
5:         $\frac{\partial Y}{\partial X^{(l-1)}} = W^{(l)\top} \cdot \frac{\partial Y}{\partial Z^{(l)}}$
6:     end for
7:     $Z_{pred} = X^{(L)}$
8: end procedure
```

At each epoch of training both of these functions are called to calculate the prediction values which are in turn used to compute the cost of the Network via 2.1. With this cost function, we can apply the backpropagation algorithms and train the model.

BackPropagation Equations

Like the feedforward equations, the backpropagation is broken into a Y and Z component. The simpler of the 2 is the Y component due to the ability to be broken into simple matrix operations. Once again, the implementation of this works by processing through layers via the `myLayers` vector. Here, the adjoints are calculated and compounded through the network. This is done for both weights and biases with respect to the Cost function C.

$$C = \alpha \cdot MSE_Y + \beta \cdot MSE_Z$$

We begin the backpropagation by computing the Y components.

Algorithm 3 Back Propagation of Y

```
Require: $Y_{train}$ is a vector of $n$ training points, $Y_{pred}$ is a vector of $n$ predicted points.
1: procedure ForwardProp($Y_{train}$, $Y_{pred}$, $n$)
2:     $\frac{\partial C}{\partial Z^{(L)}} = \frac{2(Y_{pred} - Y_{train})}{n} \odot \sigma'\left(Z^{(L)}\right)$
3:     $adjoints = \frac{\partial C}{\partial Z^{(L)}}$
4:     for $l = L, .., 1$ do
5:         $\frac{\partial C}{\partial W^{(l)}} = adjoints * X^{(l-1)^\top}$
6:         $\frac{\partial C}{\partial B^{(l)}} = SumAlongRows(adjoints)$
7:         if l > 1 then
8:             $result = W^{(l)\top} * result \odot \sigma'\left(Z^{(l-1)}\right)$
9:         end if
10:    end for
11: end procedure
```

The derivatives found here are then assigned to the member variables of their corresponding derivatives with proportion α to account for the proportion of the cost function contributed to by the standard labels.

The backpropagation of the derivative labels is where the majority of the projects computation is done. Unlike the standard label calculation, this can't be done via standard matrix multiplication. This is due to the inclusion of weight terms from high layers in derivatives of lower layers. For example:

A fully connected 3-layer network has an output of:

$$\hat{Y} = X^{(L)} = \sigma(W^{(3)} \cdot \sigma(W^{(2)} \cdot \sigma(W^{(1)} \cdot X_{train} + B^{(1)}) + B^{(2)}) + B^{(3)})$$

and a $\hat{Z}$ of:

$$\frac{\partial Y}{\partial X_{train}} = W^{(1)\top} \cdot \left[\sigma'(Z^{(1)}) \odot W^{(2)\top} \cdot \left[\sigma'(Z^{(2)}) \odot \left[W^{(3)\top} \cdot \sigma'(Z^{(3)})\right]\right]\right]$$

To calculate derivatives with respect to weights for these components of the cost function, each weight involves a complex combination of product and chain rules that can't be accomplished via matrix operations. If we attempt to do this, we can see quite quickly that it doesn't work.

Take the example:

$$\frac{\partial(W^{(3)\top} \cdot \sigma'(Z^{(3)}))}{\partial W^{(3)}} = \frac{\partial(W^{(3)\top} \cdot)}{\partial W^{(3)}} \cdot \sigma'(Z^{(3)}) + W^{(3)\top} \cdot \frac{\partial \sigma'(Z^{(3)})}{\partial W^{(3)}}$$

The dimensionality of these Matrices does not align to allow simple matrix addition. As a result, I deduced an element-wise approach to calculating the derivatives. By creating tensors for each required derivative. We can take a derivative $\frac{\partial Cost_{ij}}{\partial W_{ab}^{(K)}}$ where the outer matrix of the tensor has entries a, b and each inner matrix contains each iteration of the Cost function i, j.

When worked out for a Network with L layers, we can find that the Z component of the cost is as follows:

$$Cost_{ij} = \frac{(\hat{Z}_{ij} - Z_{ij})^2}{n}$$

where:

$$\hat{Z}_{ij} = \sum_{n_0=0}^{N_0} \sum_{n_1=0}^{N_1} \cdots \sum_{n_{L-2}=0}^{N_{L-2}} W_{n_0 i}^{(0)} W_{n_1 n_0}^{(1)} \cdots W_{n_{L-2} n_{L-3}}^{(L-2)} \sigma'(Z_{n_{L-2} j}^{(L-2)}) \sigma'(Z_{n_{L-3} j}^{(L-3)}) \cdots \sigma'(Z_{n_0 j}^{(0)}) - \left(\frac{\partial Y}{\partial X}\right)_{ij}$$

The derivative calculations of this with respect to weight $W_{ab}^{(k)}$ and bias $B_a^{(k)}$ are:

$$\frac{\partial \text{Cost}_{ij}}{W_{ab}^{(k)}} = \frac{2(\hat{Z}_{ij} - Z_{ij})}{n} \cdot \frac{\partial \hat{Z}_{ij}}{W_{ab}^{(k)}}$$

$$\frac{\partial \text{Cost}_{ij}}{B_a^{(k)}} = \frac{2(\hat{Z}_{ij} - Z_{ij})}{n} \cdot \frac{\partial \hat{Z}_{ij}}{B_a^{(k)}}$$

The partial derivatives of $\hat{Z}_{ij}$ are defined as:

$$\frac{\partial \hat{Z}_{ij}}{W_{ab}^{(k)}} = T\left(k\right) \cdot PR_W\left(a,b,k,L\right) + TS\left(a,b,k,L\right) \tag{3.3.1}$$

$$\frac{\partial \hat{Z}_{ij}}{B_{a}^{(k)}} = T\left(k\right) \cdot PR_B\left(a,b,k,L\right) \tag{3.3.2}$$

Here, PR_W and PR_B represent the product rule components of the weight and bias partial derivatives respectively:

$$PR_W\left(a,b,k,L\right) = \left[\sigma''\left(Z_{aj}^{(k)}\right) A_{bj}^{(k-1)} + \sum_{l=k+1}^{L-2} S\left(a,b,k,l\right) A_{bj}^{(l-1)} \cdot P\left(k,L-2\right)\right] \cdot P\left(0,k-1\right)$$

$$PR_B\left(a,b,k,L\right) = \left[\sigma''\left(Z_{aj}^{(k)}\right) + \sum_{l=k+1}^{L-2} S\left(a,b,k,l\right) \cdot P\left(k,L-2\right)\right] \cdot P\left(0,k-1\right)$$

The following functions are used to support these calculations:

$$P\left(p,q\right) = \prod_{m=p}^{q} \sigma'\left(Z_{n_m j}^{(m)}\right)$$

$$T\left(k\right) = \sum_{n_1=0}^{N_1} \cdots \sum_{n_{L-2}=0}^{N_{L-2}} W_{n_1 i}^{(1)} W_{n_2 n_1}^{(2)} \cdots W_{n_{L-2} n_{L-3}}^{(L-2)}$$

$$S\left(a,b,k,l\right) = \sigma''\left(Z_{n_l j}^{(l)}\right) \left[\prod_{m=k+2}^{l} W_{n_m n_{m-1}}^{(m)} \sigma'\left(Z_{n_{m-1} j}^{(m-1)}\right)\right] W_{n_k a}^{(k+1)} \sigma'(Z_{aj}^{(k)})$$

$$TS\left(a,b,k,L\right) = \sum_{n_1=0}^{N_1} \cdots \sum_{n_{k-2}=0}^{N_{k-2}} \sum_{n_{k+1}=0}^{N_{k+1}} \cdots \sum_{n_{L-2}=0}^{N_{L-2}} W_{n_0 i}^{(0)} \cdots W_{b n_{k-2}}^{(k-1)} W_{n_{k-1} a}^{(k+1)} \cdots W_{n_{L-2} n_{L-3}}^{(L-2)} \cdot TP(k,L)$$

$$TP\left(k,L\right) = \sigma'\left(Z_{bj}^{(k-1)}\right) \sigma'\left(Z_{aj}^{(k)}\right) \prod_{\substack{m=0 \\ m\neq k-1,k}}^{L-2} \sigma'\left(Z_{n_m j}^{(m)}\right)$$

These functions give us a closed-form solution to allow us to populate the tensors. By calculating the norms of the inner matrices, we can determine the precise-updates needed for the model's parameters during backpropagation. The implementation of this involves a number of nested functions, the majority of which break down the complex function denoted above.

The code begins `populateWeightTensors(double beta)` This function iterates through each layer of the network and for each, it populates every entry of each inner matrix (derivative of cost entry with respect to corresponding weight entry). A norm of the inner matrix is then taken to quantify the effect each weight has on the cost. It repeats this for every weight in the Network.

To calculate each individual elementwise derivative $\frac{\partial \text{Cost}_{ij}}{\partial W_{ab}^{(k)}}$, a function `individualWeightTensorEntry(size_t i, size_t j, size_t a, size_t b, size_t k, bool isBias)` is called. This function initializes a vector `vector<size_t> indices(myLayerSizes.size()- 1, 0)` to hold the indices required for the recursive calls that compute PR and TS.

These recursive functions are where the main computation in the network takes place. They iterate through each of the nested summations and provide values for PR_W and TS shown above.

Algorithm 4 Weight Derivative Calculation

```
Require: $n$ training points, $n_0$ training features.
 1: procedure CALCULATEWEIGHTTENSORS($n$, $n_0$)
 2:     for $l = 1, .., L$ do
 3:         $r, c = numRows^{(l)}, numCols^{(l)}$
 4:         for $a = 1, .., r$ do
 5:             for $b = 1, .., c$ do
 6:                 $norm = 0.0$
 7:                 for $i = 1, .., n_0$ do
 8:                     for $j = 1, .., n$ do
 9:                         $norm += [individualWeightTensorEntry(i, j, a, b, k)]^2$
10:                     end for
11:                 end for
12:                 $norm = \sqrt{norm}$
13:                 $\frac{\partial C}{\partial W_{ab}^{(k)}} = \frac{2}{n} \cdot norm$
14:             end for
15:         end for
16:     end for
17: end procedure
```

Algorithm 5 Weight Entry Calculation

```
Require: $i, j$ Cost function indices, $a, b, k$ weight matrix indices.
 1: procedure INDIVIDUALWEIGHTTENSORENTRY($i, j, a, b, c$)
 2:     Initialize Indices Vector for recursion.
 3:     $ProdSum = 0.0$
 4:     $ProdSum = recursiveProdSum(i, j, a, b, k, LayerSizes, indices, currentLayer = 0)$
 5:     reset Indices vector to 0
 6:     $trailingSum = 0$
 7:     if k $==$ 0 then
 8:         $trailingSum = recursiveTrailingTerms(i, j, a, b, k, myLayerSizes, indices, 1)$
 9:     else
10:         $trailingSum = recursiveTrailingTerms(i, j, a, b, k, myLayerSizes, indices, 0)$
11:     end if
12:     return $(trailingSum + ProdSum) \cdot (\hat{Z}_{ij} - Z_{ij})$
13: end procedure
```

The recursive functions in this implementation loop through the indices vector until it reaches the max value for each layer, stored in LayerSizes. The other MSE component of the Cost derivative is included in the return value.

The process for the Bias calculations is essentially the same with minor changes corresponding to the formulae above. With these implemented, we can calculate the complete derivatives of each weight and bias with respect to the total cost function (2.1). With these calculated, weights and biases can be updated accordingly and the model can begin to train.

4 Parallelisation

So far, we have looked at a serial implementation. Next, we will look at a parallel implementation of the same network. The parallelisation will be done using shared memory parallelism. The software that was used to implement the parallel model was OpenMP. This will be implemented on a machine with 8 available threads and ARM64 architecture. I chose to use this concurrent approach, due to the reduced overhead when compared with distributed memory parallelism.

The problem can be organised in a way that makes use of multi-threading, efficiently organised memory and vectorisation. The use of a distributed system like OpenMPI would introduce unnecessary overhead and complexity, as its high-level capabilities are excessive for this problem. Instead, shared memory parallelism provides a more suitable and efficient solution.

The Network was implemented with shared memory parallelism in mind and as such, the classes used in the project can make use of the multi-threading capabilities of OpenMP and apply SIMD where appropriate.

4.1 Parallelisation of Matrix

The contiguous structure of the `Matrix` class allows many of its member functions to be efficiently parallelised using SIMD. In the serial implementation, the use of an `-O3` compiler flag should instruct the compiler to apply SIMD where possible which, theoretically should be in almost every Matrix operation due to the contiguous memory structure. The compiler can identify the ability to do this when there are simple, elementwise operations. For example, in areas such as `Matrix operator+(T scalar)` or `Matrix operator-(const Matrix& rhs)`.

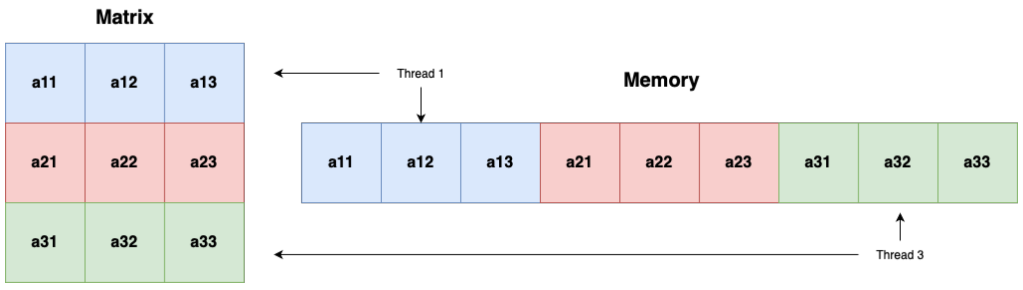

Figure 4: Matrix Memory Structure

The memory access pattern is the key to efficient Matrix operations. Each thread consists of its own contiguous memory which can utilise vectorisation due to each utilising its cache efficiently.

However, for operations with slightly more complex examples, such as `Matrix dot(const Matrix& rhs)`, the inclusion of an OpenMP directive will instruct the compiler to distribute the workload across available threads. Due to the increased complexity of these operations ($\mathcal{O}(n^3)$), parallelisation is important for efficient calculation in the forward propagation step. It can be done as follows:

```cpp
Matrix dot(const Matrix& rhs) const {
    Matrix result(myRows, rhs.num_cols());
    #pragma omp parallel for
    for (size_t i = 0; i < myRows; ++i){
        for (size_t j = 0; j < rhs.num_cols(); ++j) {
            T sum = 0;
            for (size_t k = 0; k < myCols; ++k) {sum += (*this)[i][k]
                * rhs[k][j];}
            result[i][j] = sum;
        }}
    return result;
}
```

By using OpenMP to parallelize the outer loops, the workload is distributed across multiple threads, each of which can process multiple elements in parallel. However, due to the structure of the memory, this computation involves some cache overhead making it slower than the simple element-wise operations seen above. It is for this reason that the compiler does not automatically apply vectorisation.

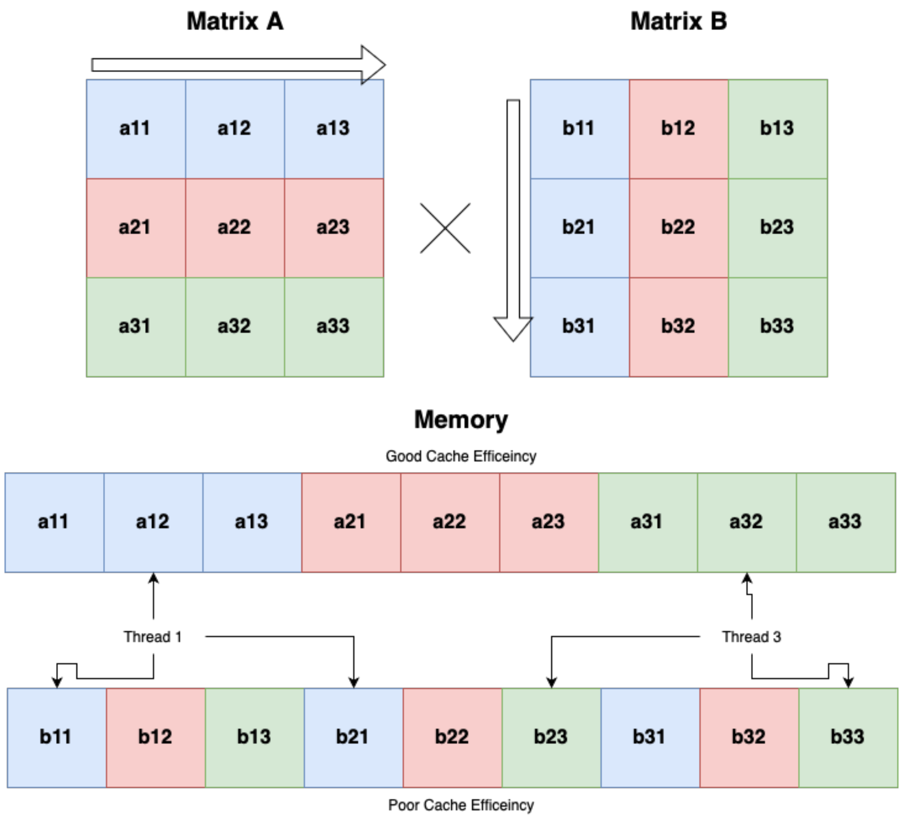

Figure 5: Dot Prod Memory Structure

For consistency, OpenMP directives were applied to the various Matrix operations where necessary. Namely (`#pragma omp parallel for`), (`#pragma omp parallel for collapse(2)`), and (`#pragma omp simd reduction(+:x)`). Although these are made redundant by the compiler with an `-O3` flag, they are implemented as good practice. The Matrix class is essentially optimal for the context of the problem. The memory pattern allows multiple elements to be loaded simultaneously, increasing bandwidth and reducing overhead. It also allows for the ability to apply vectorisation allows for quick calculation of matrix operations.

4.2 Parallelisation of NeuralNetwork

The optimisation of the `NeuralNetwork` class is more complex than that of the `Matrix` class. Even though the Network is organised efficiently using Layer classes, the complexity of the functions restricts the use of SIMD on operations that don't directly involve matrix manipulations. As such, the use of efficient multi-threading is more appropriate.

Beginning with the forward propagation algorithms, due to the dependency of `Layer[l]` on `Layer[l-1]`, this process is not easily parallelizable. Consequently, the primary optimization here relies on the parallel implementation already discussed in the matrix operations.

The main optimisation in the `NeuralNetwork` is done in the backpropagation step. Specifically, in populating the tensors. This step is the only part of the network that does not depend on subsequent calculations for its computation. As a result, we can effectively use multi-threading to call various entries of $\frac{\partial C_{ij}}{\partial W_{ab}^{(k)}}$ for a given k simultaneously. The decision to parallelize over a and b rather than k is because each k has varying size. This would lead to an imbalanced workload for some threads which would be left idle, waiting for others to finish. To maximise efficiency, each thread computes the derivative with respect to a single weight at each loop iteration. Some calculations contain more loop iterations than others and as such, dynamic scheduling is used, however, for lower threads this does not contribute too much to the overhead. This evenly distributes the workload as each thread is operating on a matrix with dimensions $\mathbb{R}^{n \times n_0}$.

The data locality is also worth consideration. When threads operate on adjacent elements of the weight matrix, there is improved data locality, which can lead to better cache utilization. This can reduce cache misses and improve overall performance.

5 Results

As stated, the Neural Network was trained on synthetically generated European Call Option Data. The data is generated upon initialisation of the program, but for the testing of the problem, the data generated and the weights initialized are seeded to be the same in all models. We will investigate the effects of scaling over network size and number of data points.

5.1 Strong Scaling

We can investigate scaling by observing the speedup of the parallel algorithm on p processes when compared to a single serial implementation. Strong scaling is this investigation with a problem of constant size and a variable number of processes.

$$S_p = \frac{T_1}{T_p}$$

The theoretical best-case scenario is called *Linear Speedup* where $S_p = p$. Almost no programs reach this speedup and only tasks that are embarrassingly parallel achieve a speedup that is relatively close.

As stated repeatedly in the previous section, parallelisation of programs involves significant overhead. This includes efficient memory management, communication between threads and load balancing. A program will almost always have these bottlenecks present preventing an optimal speedup. These bottlenecks can drastically reduce the speed increase of the program due to the diminishing nature of the speedup. This follows Amdahls Law [1]:

$$\frac{1}{f_s + \frac{1-f_s}{p}}$$

where f_s is the proportion of the program done in serial.

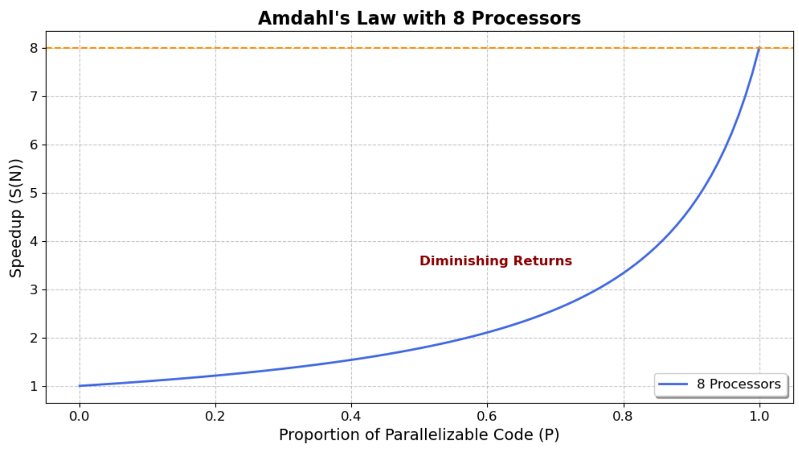

Figure 6: Amdahls Law

We will first investigate the speedup of the Network from 1 to 8 threads on a Network of size `[5,8,8,1]`. This Network size provides an accurate output for both the derivative price and its sensitivities, whilst also keeping training time reasonably low. The output of this model for 150 Epochs, $\lambda = 1.0$ and $trainingRate = 0.1$, when trained on standardised data, gave an $MSE_Y \approx 0.51$ and $MSE_Z \approx 0.81$.

This model tested on 1000 and 5000 training points with a test-train split of 80%-20%.

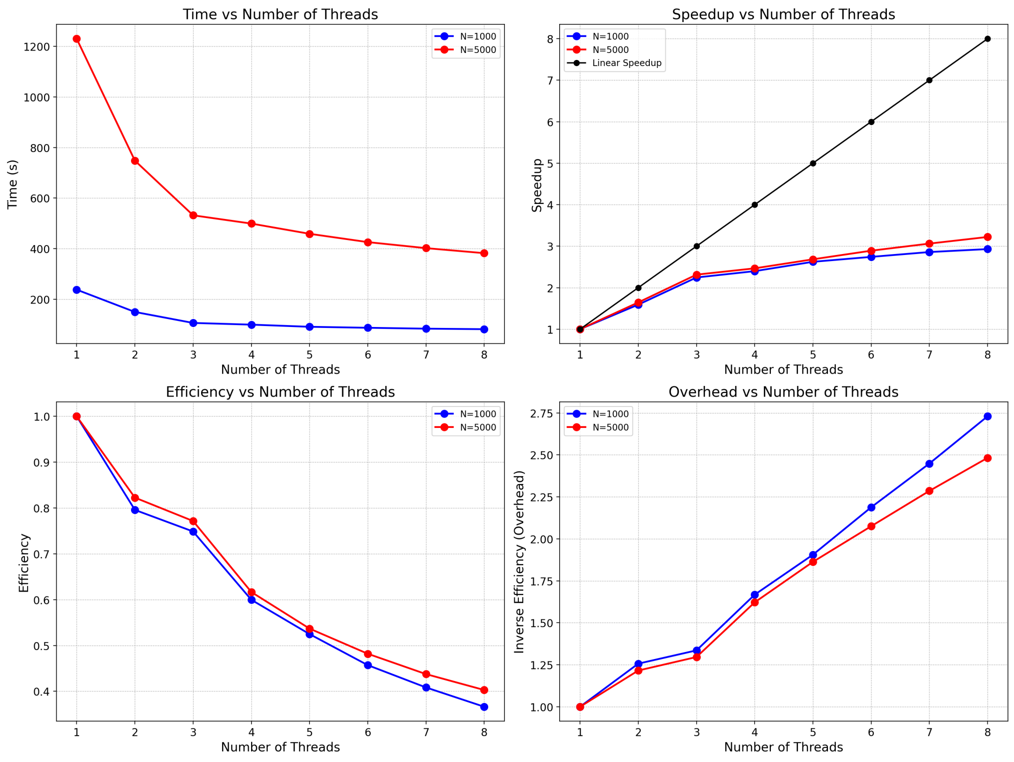

Figure 7: Strong Scaling

There is a clear logarithmic pattern to the speedup as the efficiency of fewer threads is closer to 1 than the efficiency of more threads. Although this pattern is consistent with Amdahls Law, there are other things to consider. This decrease in efficiency is also due to the additional overhead required in communication and synchronization of more threads. Factors like dynamic scheduling to avoid load balancing in the tensor population contribute to overhead, reducing the efficiency.

Another factor is that the computation of tensors involves a large number of memory access from sources that are stored non-contiguously. As threads increase, the competition between threads for memory access can cause the bandwidth to become a bottleneck. Another issue comes from different threads accessing the same cache lines simultaneously. Due to some of the different matrices on each layer being stored contiguously in memory (`myZ, myW`), there will be occurrences of different threads trying to access the same cache line leading to contention (This could be solved by padding).

We can also see that the parallelisation of the larger problem ($N = 5000$), performs marginally better than that of the smaller problem. This is due to the additional benefit received from the extra overhead that is required regardless—for example, the initialization of threads, and synchronisation. The proportion of time spent on the parallelisable components is also larger as the increase in problem size will lead to more time spent in the parallel parts such as gradient computation.

5.2 Weak Scaling

Weak scaling is the investigation of performance of the model when the problem size scales with the number of processes it uses. Theoretically, for perfect parallelisation, the speedup should remain 1. However, like linear speedup, this is not usually achievable.

The scale of the problem here comes from 2 places. The first is the size of the Matrix $\mathbb{R}^{5\times n}$ of training data ($\mathcal{O}(n)$) and the second is the size of the network ($\mathcal{O}(n^L)$). The increase in problem size for both of these should provide similar results due to the proportion of parallel work scaling identically for both.

For consistency, the weak-scaling was investigated by linearly increasing the number of training points (n).

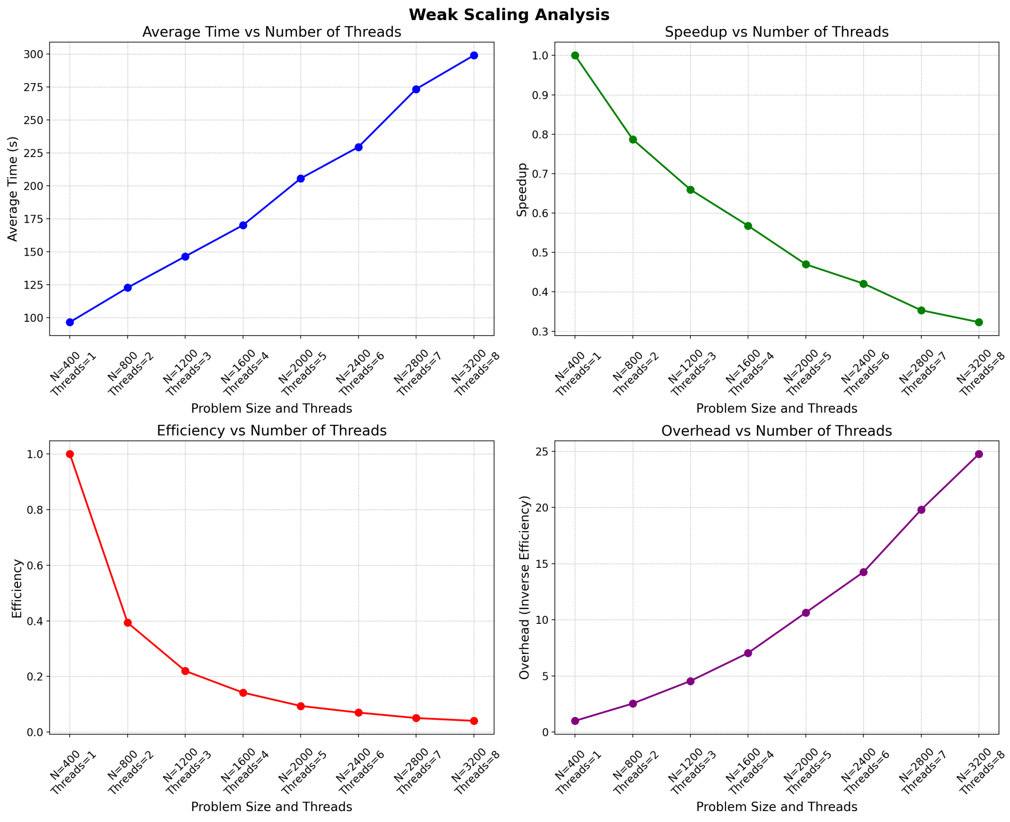

Figure 8: Weak Scaling

We can see that the efficiency of the program drops quickly which is in line with the investigation seen above. The use of additional threads increases the cost of overhead for the same reasons as discussed previously. The bottlenecks arise from synchronisation overhead and cache contention reducing efficiency as the number of threads increases.

6 Discussion

This Neural Network implemented in this project works well, providing accurate predictions of outputs alongside sensitivities. The calculation is quite efficient and makes use of well-structured memory to perform the calculation. However, some areas could be improved.

Further Work

The bulk of the work involved in the program is that of the backpropagation of the differential labels. The method of element-wise calculation of these derivatives is somewhat crude, as it was derived from first principles by manually expanding the network for various sizes to obtain the derivatives formula 3.3.1 and 3.3.2. This derivation, as shown, involves recursion to account for the varying number of layers in the network leading to a complexity of $\mathcal{O}(n^L)$. This contributes to a heavy workload in the gradient calculations and in turn, slower computation.

The largest bottleneck ties back to this recursion. The employment of dynamic programming could drastically speed up the process. Each iteration of the recursion involves multiple loops and conditionals, however, they are the same on each epoch of training. Memoisation could be used to store a derived formula for each matrix entry as it is found on the first epoch. Although this would greatly increase the memory requirement, it would have a significant impact on the speed of the program.

Another solution to this problem would involve a more robust derivation of the derivative formulas, utilising more matrix operations. This would allow the use of SIMD in more areas of the project which would also increase efficiency.

An additional problem that is causing unnecessary overhead is the cache contention involved in the tensor population algorithms. This was mentioned above, but setting up the program to involve padding on the edges of each matrix could eliminate some of the overhead involved in the implementation of equations 3.3.1 and 3.3.2.

One final area that could technically improve the speed would be implementing some momentum-based training optimiser. The ADAM optimiser was briefly mentioned in the introduction, but if implemented, it could result in faster conversion which would reduce the number of required epochs for the model.

7 Conclusion

The goal of this project was to successfully implement both a serial and parallel version of the Differential Neural Network. Both of these were successfully implemented with both models giving accurate predictions for both the outputs and the sensitivities. The model is implemented to allow for a flexible model with a variable number of layers. Each layer can also contain a variable number of nodes. This implementation allows the Network to be applied to any such model with differential training data. In a financial setting where the quick calculation of sensitivities is crucial, this model can offer an alternative to extensive Monte Carlo calculation.

The parallel implementation also provides a respectable speedup of ~ 3 over 8 threads, by leveraging efficient memory management, SIMD and multi-threading where possible. The bottlenecks were identified and possible extensions of the work were discussed.

Overall, a flexible yet efficient differential Neural Network was created that can be run with sufficient speedup in parallel.

Bibliography

[1] Gene M. Amdahl. Computer architecture and amdahl's law. *Computer*, 46(12):38–46, 2013. `doi:10.1109/MC.2013.418`.

[2] Fischer Black and Myron Scholes. The pricing of options and corporate liabilities. In *World Scientific Reference on Contingent Claims Analysis in Corporate Finance: Volume 1: Foundations of CCA and Equity Valuation*, pages 3–21. World Scientific, 2019.

[3] Mike Giles and Paul Glasserman. Smoking adjoints: Fast monte carlo greeks. *Risk*, 19(1):88–92, 2006.

[4] Brian Huge and Antoine Savine. Differential machine learning, 2020. URL: `https://arxiv.org/abs/2005.02347`, `arXiv:2005.02347`.