Efficiently Pricing Financial Derivatives Using Neural Networks

Connor Tracy

***Supervisors*: Dr Marija Zamaklar and Dr Kasper Peeters**

Durham University
Department of Mathematical Sciences

April 2021

Declaration

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

Acknowledgments

I most of all want to thank my supervisors Dr Marija Zamaklar and Dr Kasper Peeters for their helpful support and feedback. I would also like to thank my future PhD supervisors for interesting preliminary discussion on this research question, and express my gratitude to my partner and family for their continued support of my academic pursuits.

Abstract

This thesis aims to first present a pedagogical introduction to the study of neural networks, before providing an example implementation and concluding with novel research applying neural networks to pricing financial contracts. This report is written for a 4H-level target audience. The introduction to neural networks will discuss the key features, techniques and optimisations in the construction of modern neural networks, accompanied by an illustrative and accessible example implementation, visualisations and brief historical notes.

Following this, foundational option pricing theory from mathematical finance and Monte Carlo simulation will be discussed in preparation for the research component of this thesis. The performance of fully-connected neural networks are measured against incumbent techniques to learn the mapping of market data to call option contract prices: first, accuracy is evaluated on historical quotes and benchmarked against Black-Scholes, and then the training and inference time is compared to Monte Carlo pricing under the Heston-model for high-dimensional basket options.

Contents

Notation

Table 1: Unvectorised Notation

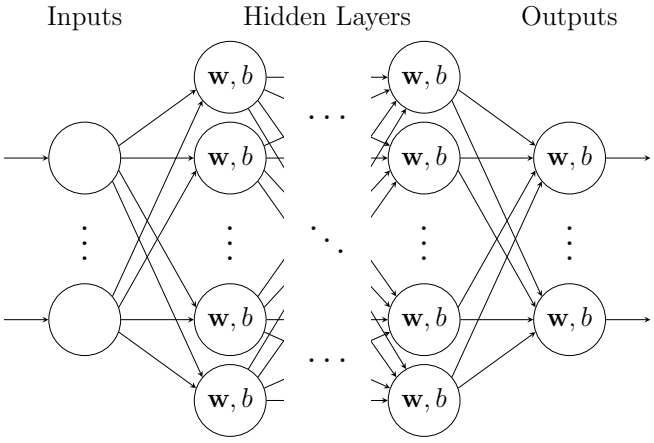
N	Dimension of input feature vector.
m	Number of training examples.
x_i	i^{th} training example.
$a^{(i)[\ell]}$	Output activation for the i^{th} node in layer ℓ.
n_ℓ	Number of nodes (layer width) in layer ℓ.
$z^{(i)[\ell]}$	Weighted sum for the i^{th} node in layer ℓ.
$w_j^{(i)[\ell]}$	Weight of $a^{(j)[\ell-1]}$ in the weighted sum $z^{(i)[\ell]}$.
$b^{(i)[\ell]}$	Bias term in the weighted sum $z^{(i)[\ell]}$.

Table 2: Vectorised Notation

X	$=$	$\begin{pmatrix} x_1 & x_2 & \cdots & x_m \end{pmatrix}^T$	Column vector of training examples.
$\mathbf{w}^{(i)[\ell]}$	$=$	$\begin{pmatrix} w_1^{(i)[\ell]} & w_2^{(i)[\ell]} & \cdots & w_{n_\ell}^{(i)[\ell]} \end{pmatrix}$	Weight vector for the i^{th} node on layer ℓ.
$W^{[\ell]}$	$=$	$\begin{pmatrix} \mathbf{w}^{(1)[\ell]} & \mathbf{w}^{(2)[\ell]} & \cdots & \mathbf{w}^{(n_\ell)[\ell]} \end{pmatrix}^T$	Matrix of weight vectors of layer ℓ.
$B^{[\ell]}$	$=$	$\begin{pmatrix} b^{(1)[\ell]} & b^{(2)[\ell]} & \cdots & b^{(n_\ell)[\ell]} \end{pmatrix}^T$	Column vector of bias terms of layer ℓ.
$Z^{[\ell]}$	$=$	$\begin{pmatrix} z^{(1)[\ell]} & z^{(2)[\ell]} & \cdots & z^{(n_\ell)[\ell]} \end{pmatrix}^T$	Column vector of weighted sums in layer ℓ.
$A^{[\ell]}$	$=$	$\begin{pmatrix} a^{(1)[\ell]} & a^{(2)[\ell]} & \cdots & a^{(n_\ell)[\ell]} \end{pmatrix}^T$	Column vector of activations for layer ℓ.

1 Introduction

Artificial intelligence describes a class of algorithms which can solve sophisticated problems by learning in a way analogous to a human. Machine learning is one such class of techniques in which an algorithm is able to learn patterns and relationships from data. Neural networks provide a powerful method to learn non-linear problems from training data via a model originally designed to mimic human learning and brain structure. This structure arranges artificial neurons in connected layers to represent components of a composite function, dynamically learning weights and biases at each neuron to fit the output mapping to the target distribution. Finally, deep learning concerns neural network models with a large number of connected layers, allowing for very complex relationships between inputs and outputs to be learned. This report presents an introduction to the algorithms and structures behind neural networks, some techniques which make deep networks possible and applies this theory in an implementation to significantly outperform existing methods for accurately and efficiently pricing financial option contracts.



1.1 Neural Networks

Neural networks are able to learn a mapping from a set of input features to output values by applying calculus and linear algebra to labelled training data in a process called *supervised learning*. The problem of learning a target mapping is often categorised a *regression* problem for continuous output values or a *classification* problem if the outputs are discrete values or categories. Once a network is trained, it can very quickly estimate the output for a given set of input features, often much faster than alternative methods and in many cases this may be the only tractable method. A classification task such as image recognition or filtering spam emails will attempt to predict which class a given input belongs to. A regression network will attempt to predict the true value of a dependent variable as a function of input feature values. The title problem of this report is the regression problem of learning the mapping from market data to the price of a financial contract known as an *option*. Such pricing does not always admit an analytical solution and so a neural network can be employed as an efficient method for learning this mapping.

The foundational structure of a neural network that underpins most modern adaptations is that of a graph with nodes called *neurons* in *layers* with each layer fully connected to the next. Each node represents the application of a non-linear *activation function* to a weighted sum of the outputs of the previous layer, plus a bias term. These *weight vectors* $\mathbf{w}$ and *biases* b are

learnable parameters which are iteratively updated according to a learning algorithm derived from the minimisation technique called *gradient descent*, informed by sample *training data* of pairs of inputs and outputs which follow the *target mapping*. This process converges the output of the final layer to match the target mapping.

1.2 Literature and History of Deep Learning

The first study of neural networks originated from attempts to model the design of an animal/human brain, beginning with the 1943 paper by McCulloch and Pitts [28] and gained in popularity through to the end of the 1960s when Minsky and Papert released a damning paper [36] in 1969 which highlighted material limitations in the common approaches to the field at the time [5]. A resurgence came in the late 1980s motivated by new ideas in cognitive science. Neural networks require significant computation and so it was not until the right combination of complementary architecture, computation optimisations and modern computing power that neural networks were able to address many interesting applications. For example, the first visual recognition of hand-written digits and one of the first applications of a network learning from data was achieved in 1989 by LeCun et al. [48] using a powerful new architecture called a *convolutional neural network* and learning algorithm called *backpropagation*. More recently, DeepMind have achieved significant milestones in artificial intelligence and academic (and human) achievement. In 2015, AlphaGo was able to beat a professional human Go player without handicap for the first time [46], with later iterations beating the best human Go player in the world. In 2020, AlphaFold [45] was able to model protein folding significantly more accurately than any existing method, experimental or otherwise, which has important real-world consequences. An article [6] in Nature describes the achievement as a "gigantic leap" which "will change everything".

It has been proven [13, 51] that even simple neural networks are *universal approximators*, that is they can uniformly approximate a wide class of functions, in particular any continuous function between two Euclidean spaces. In [13] the authors prove that arbitrarily wide networks are universal approximators, while in [51] the authors prove that under only very mild conditions, a sufficiently deep *residual neural network* is a universal approximator for a continuous mapping from a compact set to $\mathbb{R}^n$, with a minimum width of just $n + 1$ nodes in each layer.

1.3 Problem Statement

The primary objective of this report is to introduce the important concepts and techniques in the standard neural network architecture and compare the performance of a neural network implementation against two incumbent methods. This report also briefly discusses options pricing theory and Monte Carlo as a benchmark technique before concluding by comparing the accurate and efficient performance of a neural network implementation to price options contracts with these techniques.

Analytical solutions do not always exist for option contract prices and generally rely on models of market dynamics, which contain strong assumptions and must be calibrated to current market conditions. This research is therefore further motivated, in addition to the efficiency of neural networks over Monte Carlo, by an investigation of the use of neural networks as a model-free method to price options. Industry practitioners invest significant resources in calibrating models to current market conditions but it has been shown [32] that neural networks can generalise well in this setting, thereby obviating the need for a model-based approach to options pricing.

2 Basics of Neural Networks

This section builds on the heuristics of section 1.1 and aims to provide the foundational mathematical concepts and techniques underpinning how fully-connected neural networks work. The roadmap in figure 1 visualises the constructions and techniques described in section 2. The *neural network structure* (§2.1) is as a graph of nodes arranged in connected layers, each representing an *activation function* (§2.2) applied to a weighted sum of the outputs of the previous layer. The weights are calibrated using the function-minimisation technique of *gradient descent* (§2.3) which minimises the error from a target mapping and the network output. The network output is calculated using forward *propagation* and the gradient descent calculations are performed by backward *propagation* (§2.4) through the network. This error is defined by a *loss function* (§2.5), while the many calculations of each node in a layer (and each training example in a batch) are *vectorised* (§2.6) to be computed in parallel.

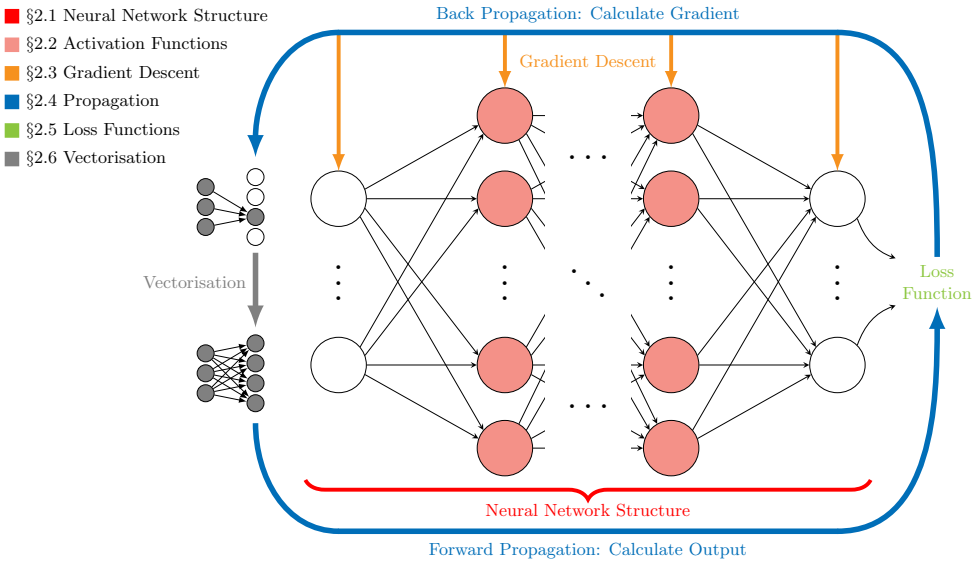


Figure 1: Roadmap visualising the structure of Chapter 2.

2.1 Fully-Connected Network Structure

A *fully-connected neural network* or *multilayer perceptron* constructs an adjustable composite function with tunable parameters called *weights* and *biases*. Changing these parameters in the right way will allow the neural network to approximate a large class of target functions. Specifically, the neural network is a graph consisting of m *layers* in which each layer is fully-connected with the previous layer. There are n_ℓ nodes called *neurons* in layer $\ell = 1, \dots, L$, with a non-linear *activation function* $a(z)$, adjustable weight vector and bias term

$$\mathbf{w}^{(i)[\ell]} = \left(w_1^{(i)[\ell]}, \dots, w_{n_{\ell-1}}^{(i)[\ell]}\right) \text{ and } b^{(i)[\ell]}$$

respectively associated to each node indexed by i in every layer indexed by ℓ. Each node of each layer applies its activation function to a weighted sum $z^{(i)[\ell]}$ of the outputs of all neurons in the previous layer. The linear combination z is the sum of the bias term of the node with the dot product of the output vector from the previous layer's neurons with the weight vector of the current node. The bias term provides each node with a trainable value to shift the activation

function 'to the left or right'. Note that the weight vector is a row vector containing as entries exactly one weight corresponding to each node in the previous layer, hence $\mathbf{w}^{(i)[\ell]} \in \mathbb{R}^{n^{[\ell-1]}}$. This process begins with the first layer having exactly as many neurons as input features and similarly the output layer has exactly as many neurons as the dimension of the codomain of the mapping $\mathbb{R}^{n_0} \to \mathbb{R}^{n_L}$ to be approximated. In this way the output is the composition of many non-linear activation functions, each which are themselves compositions of activation functions and so on, with the learned weights and biases parameters providing complexity to control the resultant non-linear function.

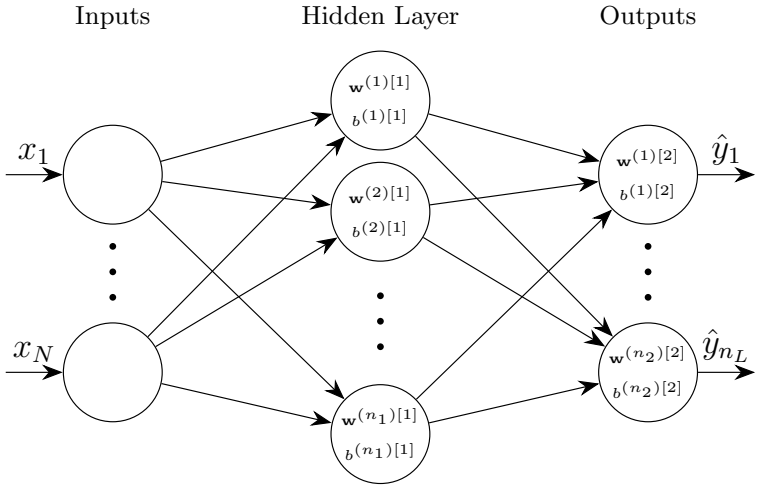


Figure 2: Single-layer shallow neural network structure for $f : \mathbb{R}^N \to \mathbb{R}^{n_2}$.

The layers between the input and output are called *hidden layers* and in enumerating the number of layers L, the input layer is not counted, writing instead that the input layer is the 0^{th} layer, since it only holds the input data with fixed width. Note that inputs and outputs are by convention column vectors, so weight vectors are row vectors[1].

2.2 Activation Functions

An *activation function* is the function which defines the output of a neuron given the weighted sum as input. Because the outputs of each layer are passed as inputs to the next layer, the output of the network is a composition of the activation functions. This means an activation function must be non-linear as otherwise the neural network would be equivalent to linear regression, since a linear combination or composition of linear functions itself defines a linear function. The activation functions give rise to the non-linearity in the neural network estimate function, so the choice of activation function can be an important decision in neural network design.

In addition to non-linearity, it is important for an activation function to be monotonic increasing so that an increase in the weight of a node corresponds to an increased activation, allowing the network to learn which features best predict the target output. Section 2.3 discusses how the network minimises the error of its estimates using gradient descent. Gradient descent and its variants require the calculation of the derivative of the output function, which as a composition of activation functions via the chain rule requires derivatives of all activation functions. Thus, it is valuable for an activation function to be continuously differentiable. Due to the repeated

[1]See Table 1 for a summary of this notation

evaluation of the derivative of an activation function, it is also important for its derivative to be fast to compute so that the learning process is able to iterate in realistic time.

Two common activation functions with bounded codomain include the *sigmoid* function and *tanh* with form as in the right of Figure 1. The sigmoid activation function is defined as

$$\sigma(z) = (1 + e^z)^{-1}, \ z \in \mathbb{R}.$$

Both sigmoid and *tanh* suffer from near-zero gradients for extreme input values which saturate the functions. Due to the repeated composition of activations in a neural network, small and large inputs or derivatives in earlier layers will propagate through the network to become vanishingly small or exploding in value, introducing numerical instability and critically slowing learning. The *vanishing gradient problem* can be addressed by using activation functions with constant gradient, regularisation (see section 3.4) to shrink weights, and more advanced architectures.

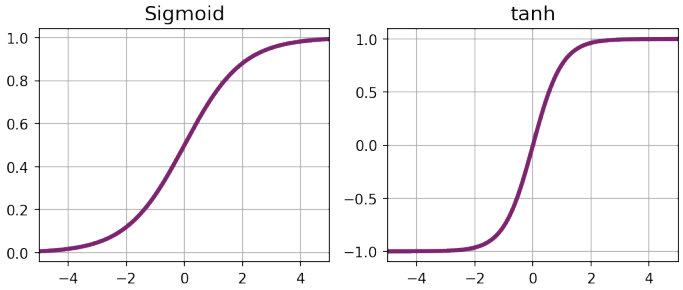


Figure 3: Bounded activation functions.

The *rectified linear unit (ReLU)* activation function was introduced by [37] and is defined to be a linear function for non-negative values and zero elsewhere. This has the benefit of having a constant gradient of 1 for positive input values, but it is not differentiable at 0 and has zero gradient for negative values. The '*leaky-ReLU*' activation function uses a small positive linear function for negative values so that the gradient is never zero, but the derivative is still not defined at 0. This leads to the *exponential linear unit (ELU)* activation function proposed by [9], which takes the further step of connecting the two linear pieces by a smooth curve around 0, thereby addressing the final issue being continuously differentiable everywhere, at the cost of adding a learnable hyperparameter γ to control this curve. The ELU activation function is defined as

$$a(z) = \begin{cases} z, & z < 0 \\ \gamma(e^z - 1), & z \leq 0. \end{cases}$$

A similar activation function is *softplus* which is defined to be $a(z) = \ln 1 + \exp z$. This is easily differentiable, unbounded and grows linearly for large input. The optimal choice of activation function depends on the problem it is applied to and so in practice several different activation functions should be considered in hyperparameter tuning (see section 3.8). The above unbounded activation functions are plotted in Figure 4.

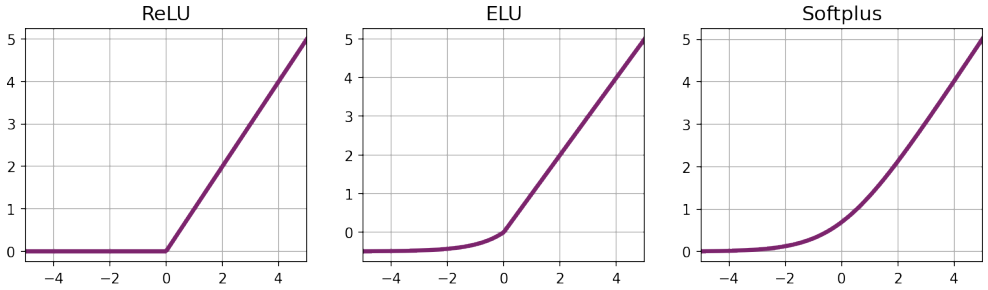


Figure 4: Example unbounded activation functions,

2.3 Gradient Descent Minimisation

Gradient descent is a powerful method to minimize a convex function by iteratively updating the inputs to traverse the image surface in the direction of the steepest gradient, in a process called gradient descent. Convex differentiable functions can guarantee convergence to the unique global minimum but non-convex functions can have many local minima [4], in which case the algorithm may not converge to the optimal minimum. This section will describe how vanilla gradient descent minimises a function. Denote by θ an input parameter of the function to be minimised.

The derivative of a function with respect to some input parameter θ is the slope of the function in the positive direction of this parameter. Thus, subtracting from each parameter some multiple α of its partial derivative at that point will move to a new point lower on the image surface. This scaling factor α is called the *learning rate* since a larger learning rate requires less jumps to travel the same distance along the curved surface. The changes to each parameter are called *parameter updates* and are performed iteratively to traverse the surface towards a (local) minimum in jumps. The intuition for why this method will find a (local) minimum on a smooth function is as a ball rolling on the surface under gravity: the ball will roll down the slope until resting at the lowest point. This analogy will be extended in section 3.6 to optimisations of gradient descent which may be thought of as incorporating momentum and friction. Notice that the steeper the jumps made are bigger for larger gradients and so conversely when close to a local minimum, the update steps will become increasingly small to allow the algorithm to converge closer towards the minimum, or travel slowly on a plateau.

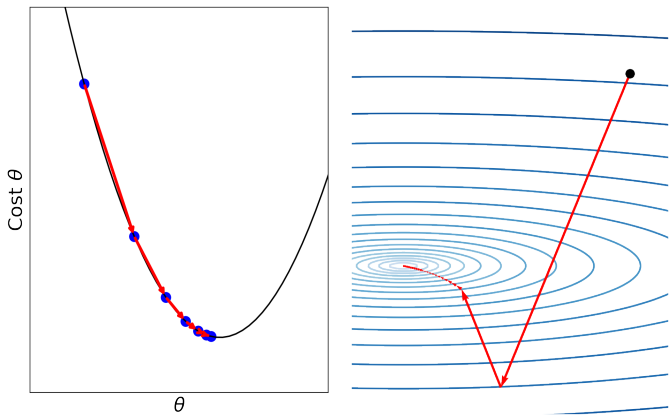


Figure 5: Visualisations of gradient descent in 2D and on a 3d contour.

If the partial derivative of $f(\underline{\theta})$ with respect to θ is positive, then the value of f can be decreased by increasing θ by some small amount. Conversely, if the partial derivative is negative, then the value of f can be reduced by decreasing θ by some small amount. Thus, when this derivative is non-zero, the value of $f(\underline{\theta})$ will be decreased when some small positive value with the same sign as the partial derivative is subtracted from θ, hence the learning rate is positive and this multiple of the derivative is subtracted to perform one step of gradient descent. The learning rate α is a hyperparameter (see section 3.8) which can be tuned manually for a given problem to identify more optimal convergence. The update step for parameter θ is then given by

$$\theta \leftarrow \theta - \alpha \frac{\partial}{\partial \theta} f(\underline{\theta}).$$

The issues of local minima and saddle points in non-convex functions have been mentioned, and it is clear that the composed functions must be differentiable, but even in the case of minimising a convex function, there are still potential issues. Using a very small learning rate will lead to unnecessarily small update steps and therefore slow convergence, while a very large learning rate is able to repeatedly overshoot a minimum. Multiplying many small or large derivatives may also be numerically unstable, with the product becoming very small or large, known as the *vanishing gradient* and *exploding gradient* problems respectively. Extreme input values can saturate bounded functions and therefore lead to vanishing gradients, although techniques such as *weight initialisation* and *batch normalisation* can help address this issue and will be discussed in sections 3.1 and 3.4 respectively. Differentiability can sometimes be addressed by making a minor modification to the function around the non-differentiable point. The manual work of determining the optimal learning rate can be automated using an algorithm to search the hyperparameter space. The limitations of a fixed learning rate can be addressed by decaying the learning rate over time, a technique which will be discussed in section 3.6. In addition to the risk of converging to a suboptimal local minimum, features of curved surfaces such as plateaus and saddle points can slow learning due to a near-zero gradient making update steps small, despite not reaching a minimum. In higher dimensions, saddle points are exponentially more likely to occur than local minima [14]. Strongly convex functions have no saddle points but to apply this technique in more general settings than convex functions, various optimisations exist and will be discussed in section 3.6.

2.4 Forward and Backward Propagation

A function may be expressed as the composition of intermediate functions. For example, the function

$$f(x, y, x) = x^2 y + 3z$$

is the composition of the maps

$$\begin{aligned} &x \mapsto t = x^2, \quad y \mapsto u = ty, \text{ and} \\ &z \mapsto u + 3z = x^2 y + 3z = f(x, y, z). \end{aligned}$$

The value of a function at a point is evaluated from the 'inside' outwards in that written as a composition of functions, the right-most or inner-most function is evaluated first, then substituted in to the next function and so on until terminating when the outermost function is applied. This process is called *forward propagation*. A *computation graph* stores this information by representing each intermediate function as a node and connecting the nodes with directed edges which present the input dependency structure. The computation graph of the function

$f(x, y, x) = x^2y + 3z$ is given in Figure 6 and uses orange arrows to signal the inputs to each intermediate function. The computation graph of a function is also useful in calculating its partial derivative with respect to any of the input parameters in an analogous reversed process called *backward propagation*.

The chain rule is used to calculate the gradient of a composite function, evaluating the derivatives for the innermost functions first and then using these to compute the derivatives of the outer (leftmost) functions in the composite function. This process takes the reverse direction along the paths of the computation graph, hence its name of backward propagation. For a given input θ and vector of inputs $\underline{\theta}$, the gradient of the composition $f \circ g\left(\underline{\theta}\right) = f\big(g\left(\underline{\theta}\right)\big)$ of smooth functions f and g in the direction of θ gives, by the chain rule, the change in f as a function of the parameter θ to be

$$\frac{\partial}{\partial\theta}\big(f \circ g\left(\underline{\theta}\right)\big) = g'\left(\underline{\theta}\right) \cdot f'\big(g\left(\underline{\theta}\right)\big) = \frac{\partial f}{\partial g} \cdot \frac{\partial g}{\partial \theta}\left(\underline{\theta}\right).$$

Notice that for a function which is the composition of many individual functions, this process is repeated until the partial derivative is expressed in terms of known derivatives that can be computed explicitly. Writing $\circledcirc f_i = F_n$ for the composition of a sequence of n functions $\{f_i\}_{i=1}^n$, and defining $F_0 = \theta$, the partial derivative of $F_n := \circledcirc f_i := f_n \circ f_{n-1} \circ \cdots \circ f_1\left(\underline{\theta}\right)$ with respect to θ is given by

$$\frac{\partial}{\partial\theta}F_n\left(\underline{\theta}\right) = \frac{\partial F_n}{\partial F_{n-1}} \cdot \frac{\partial F_{n-1}}{\partial F_{n-2}} \cdot \ \cdots \ \cdot \frac{\partial F_2}{\partial F_1} \cdot \frac{\partial F_1}{\partial \theta}\left(\underline{\theta}\right) = \prod_{i=1}^{n} \frac{\partial F_i}{\partial F_{i-1}}\left(\underline{\theta}\right).$$

Hence for a learning rate $\alpha > 0$, minimising the composite function F_n using gradient descent would update the input θ with the update step

$$\theta \leftarrow \theta - \alpha\frac{\partial}{\partial\theta}F\left(\underline{\theta}\right) = \theta - \alpha\prod_{i=1}^{n} \frac{\partial F_i}{\partial F_{i-1}}\left(\underline{\theta}\right).$$

In this way, the value of the gradient with respect to a given input can be efficiently calculated as the product of the chain of partial derivatives of each intermediate function in the composition. The computation graph caches the gradient of each intermediate function with respect to each of its respective inputs, and denotes this by directed edges between a node and the corresponding input node. Thus to compute a gradient with respect to an input θ, an algorithm can propagate backwards through the computation graph and return the product of the derivatives along the path from the output to the node representing the input θ. Figure 7 shows the backward propagation edges in blue for the example function $f(x, y, x) = x^2y + 3z$.

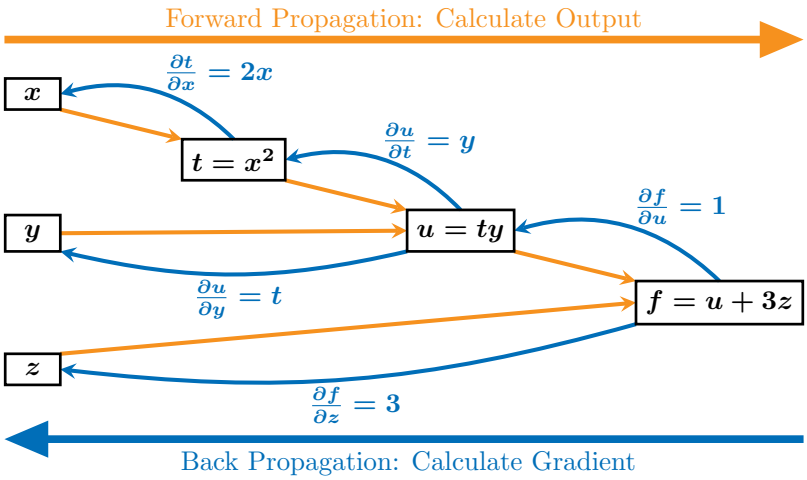


Figure 6: Computation graph for the function $f(x, y, z) = x^2y + 3z$.

The computation graph for one node in a neural network is given in Figure 7. A *forward pass* computes the output prediction for the network by computing the sequence of each layers' activations from the weighted sums of the previous layers' activations, using its current weights and biases. A *backward pass* computes the gradient using the known derivatives of the activation functions used at each node of the network.

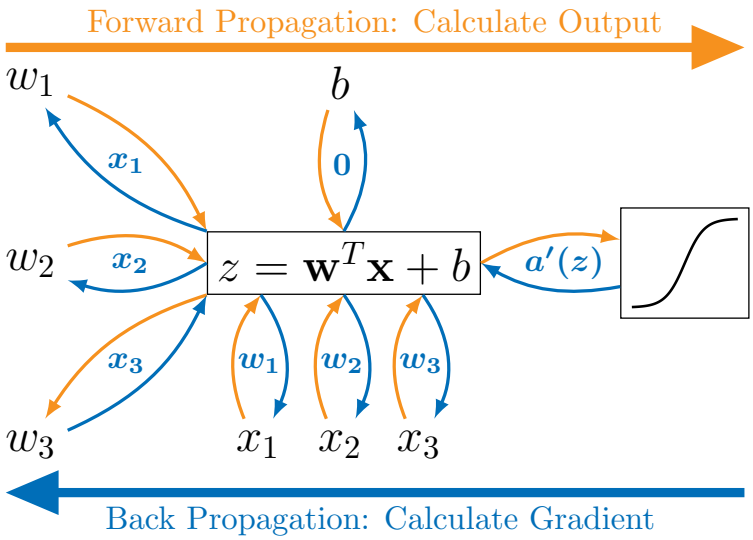


Figure 7: Single neuron computation graph with 3 nodes in the previous layer.

In addition to providing a useful tool to visualise the forward and backward passes in neural network training, a computation graph is also a powerful method employed by the popular neural network libraries such as Keras[2], TensorFlow and PyTorch, or scientific computing libraries such as NumPy and SciPy. This structure allows these libraries to store the instructions of each

[2]Keras is now incorporated into TensorFlow but still exists as a legacy standalone tool.

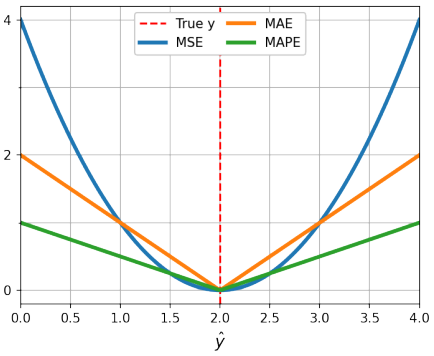
computation required to compute the output of a network or compute a gradient only when required. Because the same operations are repeated any times propagating through the network, this can be a much more efficient way to perform operations on matrix arrays, known as Tensors in TensorFlow.

2.5 Loss Functions

Regression Problems The network output is able to converge to the target mapping when tuning the weights and biases decreases a measure of the difference between the two distributions. The *loss* (error) of an individual prediction $\hat{y}$ with true value y is denoted by $\mathcal{L}(y, \hat{y})$. Denoting the current learnable parameters of the network by $\{w_i, bi\}$, the *cost function* is computed by averaging the losses over a collection of predictions, thus the cost is given by

$$J(\{w_i, b_i\}) = \frac{1}{m} \sum_{i=1}^{m} \mathcal{L}(y_i, \hat{y_i}) .$$

Gradient descent is used to minimise the cost function of a network, which may be visualised as a ball moving under gravity in the direction of steepest slope of the cost surface. Hence to minimise the cost function, the gradient of the surface must be estimated. Via the chain rule, this requires the activation functions and loss function to be differentiable (Non-differentiable functions are instead incorporated via approximated or piecewise derivatives.). Subject to these technical requirements, the optimal choice of loss function can also depend on the specific problem. For example, neural networks used in classification problems will require loss functions for discrete values, whereas regression problems consider continuous values. Moreover, certain applications may be more or less sensitive to extreme outliers, and as such it may be desirable to penalise significant outliers more or less heavily. The most common choices for the cost function are the *mean squared error* (MSE or $L1-$loss), *mean absolute error* (MAE or $L2-$loss) and *mean absolute percentage error* (MAPE). Note that in general the output of a neural network will be multidimensional and so the 'mean' in these loss functions is an average over the m dimensions of the output.



$$\mathcal{L}_{MSE}(y_i, \hat{y_i}) = \frac{1}{m} \sum_{i=1}^{m} (y_i - \hat{y}_i)^2 = \frac{1}{m} \|y - \hat{y}\|_2^2,$$

$$\mathcal{L}_{MAE}(y_i, \hat{y_i}) = \frac{1}{m} \sum_{i=1}^{m} |y_i - \hat{y}_i| = \frac{1}{m} \|y - \hat{y}\|_1,$$

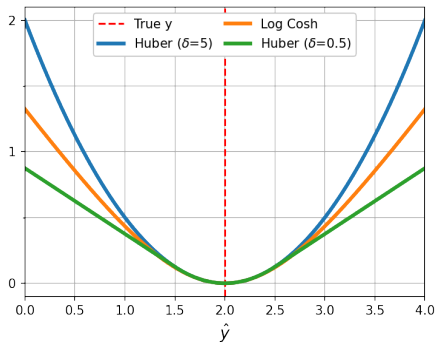
$$\mathcal{L}_{MAPE}(y_i, \hat{y_i}) = \frac{1}{m} \sum_{i=1}^{m} \left| \frac{y_i - \hat{y}_i}{y_i} \right|.$$

Figure 8: Example regression loss functions.

Notice that a network consisting of a single neuron an MSE loss function is equivalent to a least-squares linear regression, minimising sums of the squared losses of each datapoint by adjusting the single weight and bias to give the linear fit.

The MSE loss is less robust to extreme outliers than MAE but it has a simple derivative and has a small gradient for small errors. This is important for convergence as the fixed gradient of MAE does not allow vanilla gradient descent to converge to a minimum by taking smaller update steps. These loss functions are also very simple to implement. The MAPE loss is popular for its ability to capture the absolute relative error independent of scales which may be more appropriate for unnormalised data, however it also has several shortcomings. MAPE can be numerically unstable for data containing near-zero values, producing very large gradient steps or result in division-by-zero errors. Moreover, a ratio may be meaningless for some units (for example temperature) and sometimes absolute error can be more important than relative error. For example, an interest rate change of 50% from 1% to 0.5% may be considered less significant than a price drop of the same percentage. It is therefore clear that the choice of loss function must take careful consideration of the nature of the data, such as its units, scale and the presence of zeros or outliers.

In practice, a value can be manually defined to be the gradient at a single non-differentiable point but it is preferable to be differentiable everywhere. Specifically, MAE is not differentiable at zero so a value of $\{-1, 0, 1\}$ is typically chosen for the derivative at zero, although a better solution can be to identify a loss function with the same benefits while being differentiable everywhere. In addition to the most common loss functions MSE, MAE and MAPE, two further loss functions are increasingly often used: log-cosh and Huber.



$$\mathcal{L}_{LC}(y_i, \hat{y}_i) = \sum_{i=1}^{m} \log\left(\cosh(y_i - \hat{y}_i)\right),$$

$$\mathcal{L}_{Huber}(y_i, \hat{y}_i|\delta) = \begin{cases} \frac{1}{2}(y - \hat{y})^2 & \text{if } |y - \hat{y}| \leq \delta, \\ \delta|y - \hat{y}| - \frac{1}{2}\delta^2 & \text{otherwise.} \end{cases}$$

Figure 9: Example classification loss functions.

Both log cosh and the Huber loss are differentiable everywhere while incorporating the benefits of both of the standard loss functions MSE and MAE. These alternative loss functions have smooth small gradients near zero like MSE but grow like MAE to be more robust to extreme outliers. The δ hyperparameter for the Huber loss allows for the function to be tuned to each specific problem but this necessarily increases the work in hyperparameter tuning.

Classification Problems Classification problems consider discrete events which is distinct to the continuous setting of regression problems and so more appropriate loss functions exist for these problems. Two popular loss functions for classification are the categorical cross-entropy and categorical hinge. There is a tradeoff between the two: cross-entropy better estimates the probability or confidence of an estimate, at the detriment of the prediction accuracy, while the hinge loss reverses this tradeoff. The Hinge loss function and the binary cross-entropy with two

categories $y \in \{0, 1\}$ are given by

$$\mathcal{L}_{Hinge}(y, \hat{y}) = \max\left(0, 1 - (2y - 1)(2\hat{y} - 1)\right),$$
$$\mathcal{L}_{Entropy}(y, \hat{y}) = \begin{cases} -\log(1 - \hat{y}) & \text{if } y = 0, \\ -\log(\hat{y}) & \text{if } y = 1, \end{cases}$$

and it is clear to see that these loss functions are indeed minimised when $\hat{y} = y$. Notice that the binary cross-entropy loss can be conveniently written as $-y \log(\hat{y}) - (1 - y) \log(1 - \hat{y})$.

Loss Functions Domain knowledge of the problem or data may inform the choice of a specific or custom loss function. For example, no-arbitrage (see section 4.1 for a definition and discussion of arbitrage) conditions can be encoded in the loss function as in [7]. Another application of neural networks closely related to options pricing is for the hedging of options contracts. For this problem, the mean absolute tracking error (MATE) and prediction error (PE) were introduced by [14] to incorporate the temporal nature of hedging errors in the loss, although these these losses are typically used alongside the traditional loss functions of MSE, MAE and MAPE and will not be used in the research component of this thesis.

The principle method of identifying the global minimum of the cost function is by a variant of *gradient descent*, which updates the weights to travel along the cost surface in the direction of the steepest gradient to reduce the value of the cost function as fast as possible. Local minima can be avoided by using a convex loss function because if a function is convex, then it has a unique global minimum and so there exist strong theoretical guarantees [35] that learning algorithms based on gradient descent will converge to the global minimum. Without a convex loss function, training a model may converge to a local minimum which is necessarily less optimal than a global minimum. More advanced learning algorithms will be discussed in sections 3.6 and can be used to help avoid local minima.

When the convexity of a loss function is discussed, it is specifically the convexity with respect to the learnable parameters which is important. Gradient descent will attempt to minimise a cost function by updating the learnable parameters and so convexity with respect to these parameters will ensure each update step always decreases the cost towards the global minimum, with no intermediate local minima. One typically thinks of the loss function $\mathcal{L}(\hat{y}, y)$ as a function of the output of the network, but this is in turn a function of the weights and biases parameterising the function. Being convex in $\hat{y}$ does not guarantee convexity in the weights and biases. Even without a theoretical guarantee of no local minima, a neural network may still be optimised to converge to a 'good' local or global minimum.

2.6 Vectorisation

Suppressing indices for a particular node and layer, for a bias term b, row vector of weights $\mathbf{w}$ and column vector of inputs X, the weighted sum z of X can be computed as the vector equation

$$z = w_1 x_1 + \cdots + w_n x_n + b = \mathbf{w}X + b.$$

Then repeating this for n weights sums ($n_{\ell-1} = n$) and introducing a superscript $\mathbf{w}^{(i)}$ to index the sums by $i = 0, \ldots, n$, then matrix multiplication allows for all sums to be computed in the

single matrix expression

$$Z = \begin{pmatrix} z^{(1)} \\ z^{(2)} \\ \vdots \\ z^{(N)} \end{pmatrix} = \begin{pmatrix} \mathbf{w}^{(1)}X + b^{(1)} \\ \mathbf{w}^{(2)}X + b^{(2)} \\ \vdots \\ \mathbf{w}^{(N)}X + b^{(N)} \end{pmatrix} = \begin{pmatrix} w_1^{(1)} & w_2^{(1)} & \cdots & w_N^{(1)} \\ w_1^{(2)} & w_2^{(2)} & \cdots & w_N^{(2)} \\ \vdots & \vdots & \ddots & \vdots \\ w_1^{(n)} & w_2^{(n)} & \cdots & w_N^{(n)} \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_N \end{pmatrix} + \begin{pmatrix} b^{(1)} \\ b^{(2)} \\ \vdots \\ b^{(N)} \end{pmatrix}.$$

We can also write A for the activation functions applied to each entry in the column vector of weighted sums $z^{(i)}$ with

$$A = a\left(Z\right) = a\begin{pmatrix} z^{(1)} \\ z^{(2)} \\ \vdots \\ z^{(N)} \end{pmatrix} = \begin{pmatrix} a\left(z^{(1)}\right) \\ a\left(z^{(2)}\right) \\ \vdots \\ a\left(z^{(N)}\right) \end{pmatrix} = \begin{pmatrix} a^{(1)} \\ a^{(2)} \\ \vdots \\ a^{(N)} \end{pmatrix}.$$

Now introducing superscripts to index the i^{th} node of layer ℓ and a subscript j if y is a weight corresponding to the j^{th} node in layer $\ell-1$, the following notation convention is used:

$$y_j^{(i)[\ell]} \begin{cases} \ell & \text{Layer number,} \\ i & \text{Node number in layer } \ell, \\ j & \text{Corresponding node number in layer } \ell\text{-}1. \end{cases}$$

The indices for these quantities have the following ranges:

- A network has layers $\ell = 0, 1, \ldots, L$, where the 0^{th} layer is the input layer,
- Layer ℓ has nodes index by $i = 1, \ldots, n_\ell$, where $N = n_0$.

Thus Table 1 summarises the notation for unvectorised quantities.

The corresponding quantities of each node in a layer can be collected in a column vector to allow for simultaneous computation via matrix operations. These vectors are described in Table 2, noting that each node has a row vector $\mathbf{w}^{(i)[\ell]}$ of its weights and the output of layer 0 is a single training example X of dimension N, thus $n_0 = N$ and define $A^{[0]} := X$.

Proposition 2.1. The weighted sums of all nodes in layer ℓ can be computed simultaneously via the matrix equation

$$Z^{[\ell]} = W^{[\ell]}A^{[\ell\text{-}1]} + B^{[\ell]}.$$

Proof. First observe that the weighted sum $z^{(i)[\ell]}$ for the i^{th} node in the ℓ^{th} layer can be written

$$\begin{aligned} z^{(i)[\ell]} &= w_1^{(i)[\ell]}a^{(1)[\ell\text{-}1]} + w_2^{(i)[\ell]}a^{(2)[\ell\text{-}1]} + \cdots + w_{n_{\ell\text{-}1}}^{(i)[\ell]}a^{(n_{\ell\text{-}1})[\ell\text{-}1]} + b^{(i)[\ell]} \\ &= \mathbf{w}^{(i)[\ell]}A^{[\ell\text{-}1]} + b^{(i)[\ell]}. \end{aligned} \tag{1}$$

Substituting equation 1 into the definition for $Z^{[\ell]}$ yields

$$\begin{aligned} Z^{[\ell]} &= \begin{pmatrix} z^{(1)[\ell]} \\ z^{(2)[\ell]} \\ \vdots \\ z^{(n_\ell)[\ell]} \end{pmatrix} \overset{1}{=} \begin{pmatrix} \mathbf{w}^{(1)[\ell]} A^{[\ell\text{-}1]} + b^{(1)[\ell]} \\ \mathbf{w}^{(2)[\ell]} A^{[\ell\text{-}1]} + b^{(2)[\ell]} \\ \vdots \\ \mathbf{w}^{(n_\ell)[\ell]} A^{[\ell\text{-}1]} + b^{(n_\ell)[\ell]} \end{pmatrix} \\ &= \begin{pmatrix} \mathbf{w}^{(1)[\ell]} & \mathbf{w}^{(2)[\ell]} & \dots & \mathbf{w}^{(n_\ell)[\ell]} \end{pmatrix}^T \cdot A^{[\ell\text{-}1]} + \begin{pmatrix} b^{(1)[\ell]} & b^{(2)[\ell]} & \dots & b^{(n_\ell)[\ell]} \end{pmatrix}^T \\ &= W^{[\ell]} A^{[\ell\text{-}1]} + B^{[\ell]}. \end{aligned}$$

□

The above demonstrates how to propagate forwards through the layers of a network to compute the output efficiently using vectorisation. Gradient descent calculations are computed propagating gradients backwards through the network and are also vectorised. Recall from section 2.3 that one step of gradient descent updates a learnable parameter θ via

$$\theta \leftarrow \theta - \alpha \frac{\partial}{\partial \theta} F\left(\underline{\theta}\right) = \theta - \alpha \prod_{i=1}^{n} \frac{\partial F_i}{\partial F_{i-1}} \left(\underline{\theta}\right),$$

where $F\left(\underline{\theta}\right)$ is the output of the network as a function of the learnable parameters $\underline{\theta}$. Most terms in this chain rule to compute the gradient are the derivatives of each activation function used (which are often all the same) and the derivatives of weighted sums of activations. Each gradient can be computed via multiplication of matrices of these terms and the linear subtraction from each parameter can again be expressed in this vectorised matrix form.

Hence the forward and backward propagation steps through every node can be computed in parallel as a matrix operation. A GPU or TPU can compute matrix operations on thousands of threads simultaneously, so the calculations for each node in a layer can be done in parallel. Using dedicated mathematical or matrix optimised libraries such as Numpy or CuPy can optimise the matrix operation calculations themselves. Together this results in a speedup of many many times over performing each computation sequentially.

The purpose of vectorisation is to accelerate computations and thereby allow for more complex networks to be trained or for more data to be used given the same computational resources. Without such parallelisation, many important problems are intractable and neural networks become less useful. An important reason for the recent surge of progress made within deep learning in recent years is the improvement of computational power, in particular in GPUs and TPUs. Another sense in which vectorisation is used for this purpose is in replacing 'for' loops. For example, an optimised function from a scientific computing library such as NumPy computes quantities such as means using vectorised operations, which is faster than running a loop to sum all values in an array. One benefit of using high-level neural network tools such as TensorFlow and PyTorch is that these low-level operations are already well-optimised.

2.7 Example Network

Consider the simple example of estimating the period ω and amplitude A of a sine curve $f_{A,\omega}(x) = A \sin \omega$ given the values $X_{A,\omega} = \{f_{A,\omega}(\theta_1), \dots, f_{A,\omega}(\theta_N)\} = \{x_1, \dots, x_N\}$ on a set of N sample points $\Theta = \{\theta_1, \dots, \theta_N\}$ from the interval $[-4, 4]$. Here the training set is a collection of pairs $(X_{A,\omega}, (A, \omega))$ and the target mapping is $\mathbb{A}^N \to \mathbb{A}^2$; $X_{A,\omega} \mapsto (A, \omega)$. A network will then require N nodes in the input layer and two nodes in the output layer to reflect the target mapping N inputs values to two real output values.

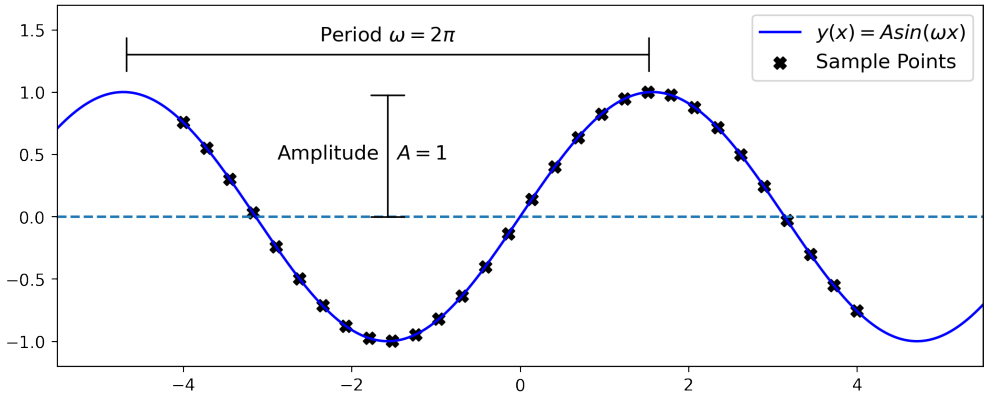


Figure 10: Sampling visualisation for sine curve $A\sin(\omega x)$.

Let the network have just one hidden layer, write $a(z)$ for the activation function to apply at each node of the hidden layer and take the activation function on the output layer to be the identity function. Write $A^{[1]}$ for the column vector of activations $a\left(\mathbf{w}^{(i)[1]}X + b^{(i)[1]}\right)$ from the single hidden layer. The neural network to learn this mapping is illustrated in Figure 11.

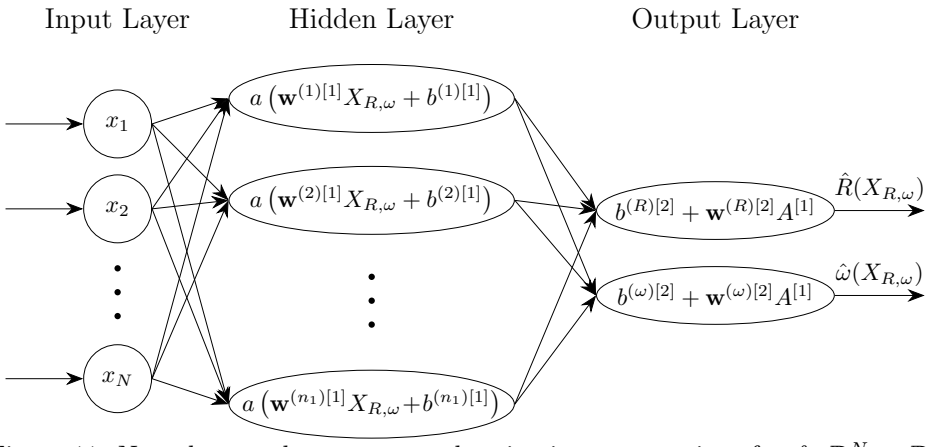


Figure 11: Neural network structure and activationcomputations for $f : \mathbb{R}^N \to \mathbb{R}^2$.

The network will aim to choose the learnable parameters $\mathbf{w}^{(i)[1]}$, $b^{(i)[1]}$, $\mathbf{w}^{(A)[2]}$, $\mathbf{w}^{(\omega)[2]}$, $b^{(A)[2]}$, $b^{(\omega)[2]}$ such that its output predictions $\hat{A}$, $\hat{\omega}$ are close to the true values A, ω. The two predictions for the amplitude A and period ω from this network given the N-dimensional input $X_{A,\omega} = \{x_1, \ldots, x_N\}$ will be the one-dimensional values

$$\begin{aligned}
\hat{A} = b^{(A)[2]} + \mathbf{w}^{(A)[2]}A^{[1]} &= b^{(A)[2]} + \mathbf{w}^{(A)[2]}\begin{pmatrix} a\left(b^{(1)[1]} + \mathbf{w}^{(1)[1]}X_{A,\omega}\right) \\ a\left(b^{(2)[1]} + \mathbf{w}^{(2)[1]}X_{A,\omega}\right) \end{pmatrix} \in \mathbb{R}, \\
\hat{\omega} = b^{(\omega)[2]} + \mathbf{w}^{(\omega)[2]}A^{[1]} &= b^{(\omega)[2]} + \mathbf{w}^{(\omega)[2]}\begin{pmatrix} a\left(b^{(1)[1]} + \mathbf{w}^{(1)[1]}X_{R,\omega}\right) \\ a\left(b^{(2)[1]} + \mathbf{w}^{(2)[1]}X_{A,\omega}\right) \end{pmatrix} \in \mathbb{R},
\end{aligned} \tag{2}$$

where $X_{A,\omega} \in \mathbb{R}^N$ is the input column feature vector and $\mathbf{w}^{(i)[\ell]} \in \mathbb{R}^{n_{[\ell-1]}}$ the row weight vector for the i^{th} neuron in the ℓ^{th} layer.

Even with such a small network, there are many degrees of freedom arising from the bias terms and weight vectors. With a non-linear activation function, this could already be a complicated function which is capable of fitting a wide class of possible target mappings. Adding more nodes in the hidden layer will increase the number of terms in each weighted sum $z^{(i)[\ell]} = \mathbf{w}^{(i)[\ell]}A^{[\ell-1]} + b^{(i)[\ell]}$ (where $A^{[\ell-1]}$ is the column vector of activations for layer $\ell - 1$, not an amplitude A), while adding more hidden layers will repeatedly compose a function of the same form as the equations in (2).

This is a trial-and-error process in which the network computes an estimate using the current set of weights and biases, the loss value for a regression loss function such as the mean squared error is compared with a training example, and this difference is then used to inform an update to the weights and biases to closer fit the training data. This update mechanism is called *gradient descent* and described in section 2.3. It is clear already that more complex networks will have more complex output functions, providing more degrees of freedom in their output to fit the data, and that providing more data will better inform the network of the true underlying mapping. For example, if the training set consisted of only two data points, there is not enough information to capture the shape of a target curve more complex than a straight line. To decide how to change the weights and biases to better fit the output of a network to a target mapping, gradient descent is used.

3 Optimising Neural Networks

In addition to the fundamental features of a standard neural network described above, there are other, often more configurable additional optimisations available to optimise a network during the development stage or its architecture.

3.1 Weight Initialisation

A network learns from the errors of its estimates with successive choices of the weights and biases, and before any estimate can be calculated, the weights must be initialised to some values. Deep networks can experience vanishing or exploding activations or vanishing gradients and it is possible to address this issue with certain choices of weight initialisations.

Even with lots of training data, training time and an appropriate architecture, a deep network is still liable to fail to learn if activations explode or vanish through forward propagation. In addition to numerical instability such as overflow errors, this saturates some non-linear activation functions and causes vanishing gradient so learning is slow or cannot converge with gradient descent. Similarly in backward propagation, the repeated operations at each layer of the network can lead to vanishing gradients. As such, it is important to avoid these things which can stop convergence. This can actually be achieved via *weight initialisation*. There are some clear general principals for choosing how to initialise weights, namely to small and random values, and for specific choices of activation function it has been proven that a specific choice of initialisation on each layer can prevent layer activations or gradients from exploding or vanishing.

There is a specific choice of weight initialisation to be avoided: if weights are set to be some constant value, the gradient will be the same for every neuron so each update step will act the same on each neuron and thereby introduce unhelpful redundancy [1] (such neurons are called *symmetric*) as each node then acts the same. In particular, all-zero parameters will give an output which is always zero and so no learning can occur.

Instead randomly initiated weights should be used to break this symmetry and the variance of the random distribution should be small because this speeds up convergence for some reason. It used to be common to initialise weights to a small-variance centred Gaussian, however then the variance of activations and back-propagated gradients grow with the square root of the number of input neurons [33] (the number of neurons in the previous layer for a fully connected graph), so it is better to scale the weight of nodes in a layer by dividing by the square root of the number of nodes in the previous layer n_{l-1}. This makes sense intuitively as approximating activations by linear/identity function, a forward propagation step (via a matrix multiplication) computes the weighted sums of the $n^{[\ell-1]}$ activations of layer $\ell-1$. On average the weighted sum then has a variance on the order of $n^{[\ell-1]}$ times the variance of the activations of layer $\ell-1$. Hence initialising to have unit variance and scaling the weights of each layer by something of the order of $1/\sqrt{n^{[\ell-1]}}$ might be expected to yield consistent activation variances between layers.

In fact papers by Glorot and Bengio in 2010 [18] and He et al. in 2015 [22] proposed what are now very widely used weight initialisations known as 'Xavier' and 'He' for the *tanh* and ReLU activation functions respectively. These aim to ensure the expected variance between layers is constant both for activations in the forward pass and for gradients in the backward pass.

The Xavier initialisation randomly initialises weights in layer ℓ with distributions

$$\sqrt{\frac{6}{n^{[\ell-1]}+n^{[\ell]}}}\mathcal{U}(-1,1) \quad \text{or} \quad \sqrt{\frac{2}{n^{[\ell-1]}}}\mathcal{N}(0,1).$$

The He initialisation randomly initialises the weights in layer ℓ with the distribution

$$\sqrt{\frac{2}{n^{[\ell-1]}}}\mathcal{N}(0,1).$$

Before these methods were proposed, it was common to take the naive approach of initialising the weights of layer ℓ with the distributions

$$\sqrt{\frac{1}{n^{[\ell-1]}}}\mathcal{U}(-1,1) \quad \text{or} \quad \sqrt{\frac{1}{n^{[\ell-1]}}}\mathcal{N}(0,1).$$

It is instructive to see how the Xavier/He initialisation normalisation factor is derived to ensure constant expected variance in the activations as well as the gradients. Note the paper by He [22] explains that the derivation of the Xavier initialisation in [18] does not account for non-linear activations and is therefore inappropriate for ReLU activation functions. Moreover, He et al. demonstrate that although performance is similar on shallower networks (i.e. up to 10 layers), the Xavier initialisation failed to converge on very deep networks while the network using He initialisation was still able to converge.

Theorem 1. *Assume a neural network uses symmetric activation functions, and both the weight initialisation and input features are i.i.d., centred, small and independent. Then Xavier weight initialisation approximately maintains activation variances and back-propagated gradient variance as one propagates forward or backward through the network.*

Proof. First consider forward propagation. The activation of a node will take the form

$$a = \tanh w_1 x_1 + \cdots + w_{n_i n} x_{n_i n},$$

where the w_i are the weights for the $n^{[\ell-1]}$ activations from the previous layer. For small initialised weights w_i, the tanh activation is approximately the identity, as in figure 17. The variance

of the activation a is therefore approximated by

$$\begin{aligned}
Var(a) &= Var\left(w_1 x_1 + \cdots + w_{n^{[\ell-1]}} x_{n^{[\ell-1]}}\right) \\
&= n^{[\ell-1]} \cdot \mathbb{E}\left[\left(w_1 x_1 - \mathbb{E}\left(w_1 x_1\right)\right)^2\right], && \text{since all terms are i.i.d.} \\
&= n^{[\ell-1]} \cdot \mathbb{E}\left[\left(w_1 x_1 - \mathbb{E}\left(w_1\right)\mathbb{E}\left(x_1\right)\right)^2\right], && \text{by independence of weights and inputs} \\
&= n^{[\ell-1]} \cdot \mathbb{E}\left[\left(w_1 x_1\right)^2\right], && \text{weights and inputs are centred} \\
&= n^{[\ell-1]} \cdot \mathbb{E}\left[\left(w_1 - \mathbb{E}\left(w_1\right)\right)^2\right] \cdot \mathbb{E}\left[\left(x_1 - \mathbb{E}\left(x_1\right)\right)^2\right], && \text{centred and independent} \\
&= n^{[\ell-1]} \cdot Var\left(w_1\right) \cdot Var\left(x_1\right).
\end{aligned}$$

Hence to ensure approximately equal variances between layers, the condition

$$Var(a) = Var(x_i) \implies Var(w_i) = \frac{1}{n^{[\ell-1]}}$$

should be imposed. Similarly, it is shown in [18] that for backward propagation gradients to have equal variance between layers, the related condition $Var(w_i) = \frac{1}{n^{[\ell]}}$ should also be imposed. In the case of networks with constant layer widths, we are done. Otherwise, a harmonic mean of these two conditions can be taken to yield

$$Var(w_i) = \frac{2}{n^{[\ell-1]} + n^{[\ell]}}$$

It is immediate that the classical initialisations with uniform or Normal sampling have variances of $1/3n$ and $1/n$ respectively and so as scalar multiples, the weight initialisations which approximately achieve constant unit variance of activations and backward propagated gradients are as claimed. □

3.2 Data Partitioning

In developing a neural network, available data are partitioned into three disjoint sets: *training data*, *development or validation data*, *test data*. The training data are generally the largest partition and is used to run gradient descent for learning. The validation set is used to evaluate the *generalisation error* on unseen data during training. Finally the test data are used to evaluate the performance of a fully trained model on unseen data, for comparison between different models. When gradient descent has been performed once on every datapoint in the training set, the network is said to have completed one *epoch*. Section 3.5 discusses the use of minibatches to take each gradient descent update step using only subsets of the training set, in which case each step or pass of a minibatch is called a single *iteration*.

We will discuss in section 3.4 how a network can begin to overfit the training data and become biased away from the target distribution is a sufficiently complex network is trained for too long. The validation error is then a useful tool for judging when a network has trained long enough to begin overfitting the training set, so that the parameters used at the point of lowest error can be used for the trained model. Section 3.3 discusses how this point can be determined graphically or using TensorFlow callbacks in section 3.9.

The training and validation sets must be disjoint as the network learns directly from the training data and in particular can memorise individual examples without learning the underlying pattern. Of interest is instead the generalisation performance beyond the training examples,

and not how well a network learns the training data. The validation error can be used to choose hyperparameters: training many models with different hyperparameters and then choosing the one with the optimal hyperparameters will ensure our network has not only learned the training data, but the architecture is also optimised for the target distribution to further improve performance. The test set should ideally also be disjoint from both the training and test sets so that the assessment of the performance of the network can reflect the expected performance on unseen data. However, it is not uncommon for the validation error to be reported instead of the test error, though this risks the hyperparameters being optimised for the validation set only and not truly evaluating the generalisation performance of the network. In this case, the dual-purpose development and test set is more often referred to as the *validation data*.
[3]

There is no universal rule for determining the relative sizes of training, development and test sets, however early work [21] suggested the validation split should scale with a ratio of complexity between the inference optimisation problem and the hyperparameter optimisation problem. It is however somewhat common to partition an available dataset into a round 60/20/20% split to form the training, development and test sets respectively, or 60/20/20% split if the validation error is reported in place of a test error. Andrew Ng is an early pioneer in deep learning and recommends [41, 40, 40] that for very large datasets, a split closer to a 98/1/1% partition can be appropriate to aid training.

The issue of partitioning the data such that the three datasets are disjoint can be more subtle than just repeating data between the sets: namely that of *data tagging* or *data leakage*. Suppose financial data is used in training with the objective to predict or estimate the price of a contract as a function of input features. If the date and time were to be included as input features, then the model could learn to account for historical trends according to the date instead of learning the underlying distribution, in this case the date tags the training examples. Entries in the validation sets could then be correlated by their dates to artificially reduce the error of the estimate despite no increased 'understanding' of the underlying target mapping. Data leakage describes the repetition of training examples in the validation sets, which would artificially lower the generalisation error as the network will have already seen the correct mapping for this input during training. In the case of market data, data leakage could occur from a contract being traded on multiple exchanges or reported under multiple identifiers due to mergers or acquisitions. The literature review [43] explains that partitioning the data into training and validation sets should not violate the time structure of the data. The time structure of the data is violated by an 'even-odd' selection in which adjacent training examples are selected for different sets, however random sampling or a chronological division would prevent data leakage by preserving the time structure of the data. The neural network implemented in this report uses chronological data partitioning and the overall design is described in section 5.1.

3.3 Underfitting, Overfitting and Loss Plots

A network is said to have *high bias* if it *underfits* the training data or *high variance* if it *overfits* the training data. A network overfitting a training set will follow the spurious patterns specific to the training set rather than following the more general underlying pattern of the target mapping. If the training set is not representative of the target distribution, a network will necessarily learn this sampling bias and not generalise well. A network can also overfit the training data if it is

[3] Idk if this is mentioned here already but talk about using lower quality data in training and higher quality in validation, or maybe proportions of them skewed towards savouring the higher-value data. I also like the idea of mentioning medical x-rays where the negative result is far more common than the positive result and so you put most positive results in the validation set.

sufficiently complex and has trained for long enough to learn the individual datapoints or nuanced patterns found only in the training set. A network may underfit the data if the architecture is not appropriate or not sufficiently complex to capture the target mapping, if the network has not trained for long enough to converge to the target mapping or if the regularisation techniques (see section 3.4) constrain the network too much. Figure 12 compares the an underfitting, overfitting and well-fit output function for a network with high bias, high variance or an optimised balance against the training data.

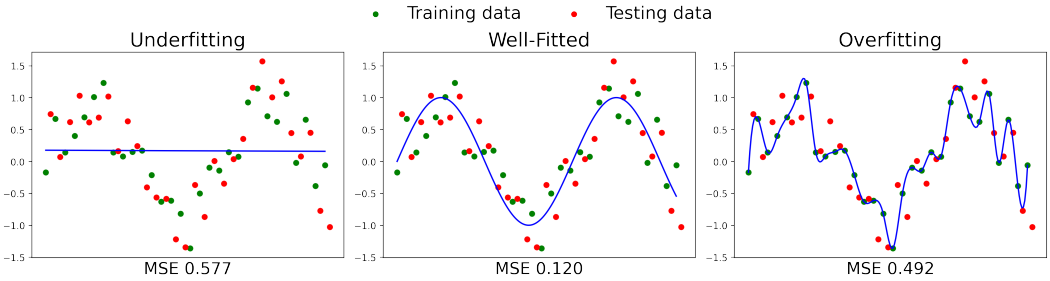


Figure 12: A simple (linear) model underfits the data, the perfect fit has residual error due to noise, and an overfitting model learns only the training set.

A very useful diagnosis tool when building neural networks, especially when building directly from the fundamentals, is the *loss plot* which graphs training and validation cost value against epoch number. In this way the trajectory of generalisation performance can be estimated by the cost value evolution on unseen data changes as the model trains for longer. This is a powerful tool for the role it plays in the early stopping regularisation technique, identifying an appropriate learning rate, diagnosing over or underfitting, as well as to compare different models. Additionally, if building a model from the fundamentals, unexpected behaviour in this plot can help identify bugs in the code. Note the cost is plotted against against *epochs* which represents the average number of times a given datapoint has been used in training, rather than *iterations* which depends on the choice of batchsize. The use of dropout or minibatches (see sections 3.4 and 3.5 respectively will add noise to the plot because a minibatch uses different samples with their own distributions, while dropout changes the network each iteration. The overall trend of the loss should be generally monotonic decreasing, however when using minibatches or dropout it is common to use an exponentially weighted moving average to smooth the variations to better visualise the underlying trend. In general, the validation error will be greater than the training error and a large deviation from this could suggest data leakage has occurred.

An overfitting model can be identified by a validation loss which is much higher than the training loss, since the model begins to learn the training set and not the ability to generalise to the unseen data of the validation set. If a network is trained for too many epochs, it may be able to learn the specific datapoints or local structure in the specific training data sample without observing more data to generalise these trends. Overfitting from too much training can be diagnosed by the validation loss initially decreasing as the network begins to learn and then beginning to increase after some number of epochs when the network begins overfitting. At this inflexion point, the network can be stopped early in a process called *early-stopping* (see section 3.4) and the trained network can use the weights and biases at the stopped time when generalisation error was minimised.

The effect of additional data on generalisation error depends on whether a network is over or underfitting. An overly simple network without the complexity to fully describe the target mapping (for example fitting linear regression to a quadratic trend) will be unable to learn the

target mapping effectively, irrespective of the quantity of training data. The high bias of the model will ensure a high test loss for any training set size. The test loss of a network with excess capacity can slowly slowly to a better test loss as the training set size increases because the training becomes a more complete representation of the target mapping. However initially the additional complexity will learn patterns or datapoints specific to the training set, and result in a non-zero contribution of variance towards the test loss, via the bias-variance tradeoff (see 3.3). A well-fitted network will minimise both the contributions of bias and variance to the test loss and therefore minimise the generalisation error to as close to the intrinsic noise as possible. Sourcing additional data can be expensive and so it is useful to know when to instead invest time into improving network design to address high bias instead. See figure 13 for an illustration of these three cases.

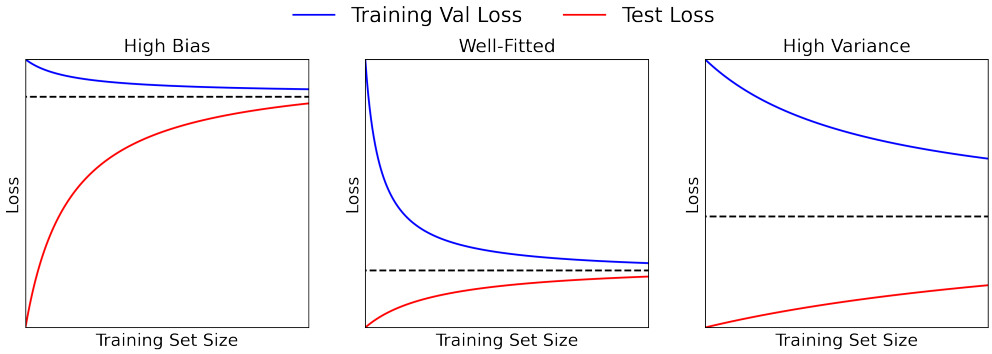


Figure 13: Simple models never learn a complex mapping irrespective of data volume. Complex models generalise given enough data, but a well-fitted model converges to the smallest loss.

In addition to the 'double-dip' paradox discussed in the following section, if training or testing data sample different distributions, the training error and validation error will naturally converge to different values as more data is added. This is because a network will be optimised through a learning algorithm to minimise the loss on the training set, but not optimised to minimise loss on the differently-distributed validation set. Data cleaning and thoughtful data collection are therefore important considerations in the development of a neural network, that is, training data should be representative of the testing data or expected real-world inputs. However, tuning of hyperparameters is an optimisation problem on a validation set so if this validation set is chosen to use higher quality or more representative examples, then the final trained model will be better suited for minimising the test loss. More discussion on this point is included in section 5.2.

It is possible for validation and test losses to be smaller than training loss in practice. For example, weight regularisation terms are not applied during inference, thereby artificially increasing training loss relative to the validation or test losses. Moreover, during training loss is averaged over each minibatch while the model is updated and improved, while validation loss is (often) evaluated at the end of each minibatch or epoch and test loss is evaluated on the final fully-trained model and so more learning has occurred between the evaluation of validation or test loss, as compared to the training loss. A validation or test loss which is significantly less than the training loss however may be a symptom of data leakage, whereby the 'unseen' validation or test set contains examples or information present in the training set.

In addition to diagnosing over and underfitting, alternative loss plots can also be useful for determining some hyperparameters manually (See section 3.8 for discussion of automatic methods of hyperparameter tuning.). In particular, the learning rate used in gradient descent controls how quickly a network can converge to a minimum. A learning rate which is too small may lead

to being stuck on a plateau for many epochs or being stuck in local minima, while a learning rate which is too large can sometimes lead to divergence. It is generally more desirable to have a lower learning rate than a larger learning rate as it is better to learn slowly than not at all. A loss plot for a network with a small learning rate may have a validation loss which does not decrease or decreases slowly. A very large learning date which diverges would have an increasing validation loss curve, while an intermediate size learning rate may still converge and at a faster initial rate but may overshoot the minimum to converge more slowly overall. Finally an optimal learning rate will lead the validation loss to decrease quickly to a global minimum.

In addition to loss plots, learning rate may also be informed by plotting the validation loss of a trained model against the learning rate. Typically the learning rate is considered on a logarithmic scale since a priori it is not known on which scale the optimal value lies. In the leftmost section of figure 14, the learning rate is too small for convergence to occur. It may be possible the network is unable to escape a local minimum, or not enough epochs have been provided to traverse sufficiently far along the loss surface to a minimum. With a learning rate of 10^{7} to $10^{9/2}$, the validation loss decreases with increasing learning rate. Some networks may then experience a transient increase in validation loss called the 'double-dip' paradox and is discussed in more detail at the end of this subsection 3.3. Excessively large learning rates can lead to divergence and so the validation loss increases unbounded. It is preferable to use a large learning rate to converge as quickly as possible, however it is common to choose a learning rate slightly less than the absolute minimum to reduce the risk of divergence with a marginally slower convergence. Popularised in *fast.ai*, a related method is to record the loss of a network after each training batch and increase the learning rate exponentially between each batch. Plotting the loss against learning rate, it has been proposed [47] that the optimal choice of learning rate is on the steepest slope of this plot. As such, given a plot as in figure 14, the optimal learning rate may be 10^{-3} to 10^{-2} or more likely 10^{-6} to 10^{-4}. The consequences of choosing a bad or large learning rate are reduced or avoided by using learning rate decay, as discussed in section 3.6, and the challenge of identifying a single optimal learning rate alongside other hyperparameters is addressed in section 3.8.

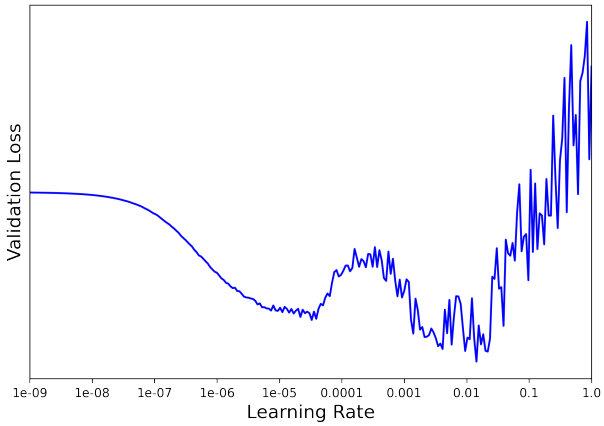


Figure 14: Determining the optimal learning rate.

Overlaying loss plots for models of different configurations is also a useful tool for directly comparing many different hyperparameter configurations at once. In particular, running training in parallel for speed and using TensorBoard callbacks to automatically log and plot these runs is especially helpful. The loss and epoch number are expected to trend exponentially [12] and so a log-log plot can also be useful to more easily interpret and compare linear results.

Bias-Variance Tradeoff We may formalise the concepts of bias, variance and error to discuss the *bias-variance tradeoff* in terms of a *bias-variance decomposition* for minimising a given loss function. More generally, the variance of an estimator describes how concentrated or dispersed its estimates are while the bias of an estimator describes the offset or error from the true value being estimated, and these intuitions correlate with the concepts of overfitting and underfitting from neural networks.

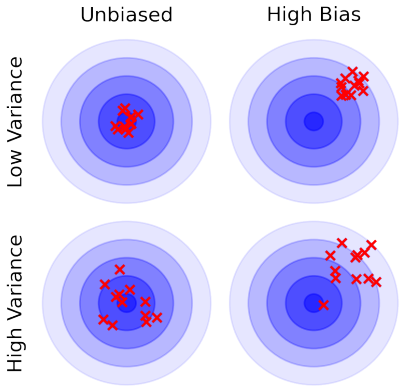


Figure 15: Classic visualisation of bias and variance in sampling.

Definition 3.1. Let $\hat{\theta}$ be an estimator for the deterministic function θ. Define the *bias* and *variance* of $\hat{\theta}$ by

$$\begin{aligned}\text{Bias}_D(\hat{\theta}) &= \mathbb{E}_D[\hat{\theta}] - \theta, \\ \text{Var}_D(\hat{\theta}) &= \mathbb{E}_D[\hat{\theta}^2] - \left(\mathbb{E}_D[\hat{\theta}]\right)^2 = \mathbb{E}_D\left[\left(\hat{\theta} - \mathbb{E}_D[\hat{\theta}]\right)^2\right].\end{aligned}$$

Here $\mathbb{E}_D[\cdot]$ denotes the expectation over all possible training sets D as samples of the target distribution. This also gives rise to a definition for the sign of a non-zero bias. If the target mapping fails to be injective or deterministic, there will be an unavoidable uncertainty as θ will take values probabilistically. The target distribution can therefore be modelled as a random variable, and since the optimal deterministic estimator $\hat{\theta}$ will never be able to fit the random variable perfectly, this also gives rise to a definition for the *noise* intrinsic to the distribution.

Definition 3.2. Let Y be a random variable parameterised by x and define the *noise* of Y by

$$\text{Noise}(Y) = \mathbb{E}_x\left[(Y(x) - \mathbb{E}[Y(x)])^2\right].$$

There exists a *bias-variance decomposition* of the expected loss of an estimator on unseen values of a probabilistic distribution (such as the neural network output estimating a target mapping containing noise) and it will be instructive to derive this for the 1-dimensional squared-error (MSE) loss function, demonstrating more clearly how underfitting is balanced against overfitting. Consider sampling an unseen value from the probability distribution of a random variable Y. Denote by $\mathbb{E}_Y[\cdot]$ the expectation over probability distribution of Y from which the unseen value is sampled and write $\mathbb{E}[\cdot] := \mathbb{E}_Y\left[\mathbb{E}_D[\cdot]\right] = \mathbb{E}_D\left[\mathbb{E}_Y[\cdot]\right]$ for the expectation over all unseen values and training sets that could be taken from this distribution. It is assumed that the order of expectations may be swapped.

Proposition 3.3 (Bias-Variance Decomposition)**.** Let $\hat{y}$ be a 1-dimensional deterministic estimator for a random variable Y. The expected MSE loss of $\hat{y}$ over unseen values and training sets is given by

$$\begin{aligned}\mathbb{E}\left[\mathcal{L}\left(\hat{y}, Y\right)\right] &= \mathbb{E}_Y\left[\mathbb{E}_D\left[\left(\hat{y} - Y\right)^2\right]\right] \\ &= \mathbb{E}_Y\left[\text{Bias}\left(\hat{y}, Y\right)^2 + \text{Var}(\hat{y})\right] + \text{Noise}(Y).\end{aligned}$$

Proof. Observe first that for a random variable X,

$$\begin{aligned}(X-a)^2 &= \left[(X - \mathbb{E}[X]) + (\mathbb{E}[X] - a)\right]^2 \\ &= (X - \mathbb{E}[X])^2 + 2\left(X - \mathbb{E}[X]\right) \cdot (\mathbb{E}[X] - a) + (\mathbb{E}[X] - a)^2 \\ \implies \mathbb{E}[(X-a)^2] &= \mathbb{E}\left[(X - \mathbb{E}[X])^2\right] + 0 + \mathbb{E}\left[(\mathbb{E}[X] - a)^2\right]. \end{aligned} \tag{3}$$

Note that values of $\hat{y}$ depend on its training set, while Y is independent of any training set, in particular $(\mathbb{E}_D[\hat{y}] - Y)$ and $\left((\mathbb{E}_Y[Y] - Y)^2\right)$ are both constant w.r.t. $\mathbb{E}_D[\cdot]$. Applying equation 3 to $\mathbb{E}_Y\left[(\hat{y} - Y)^2\right]$ and $\mathbb{E}_D\left[(\hat{y} - \mathbb{E}_Y[Y])^2\right]$ yields

$$\begin{aligned}\mathbb{E}\left[(\hat{y} - Y)^2\right] &\overset{3}{=} \mathbb{E}\left[(\hat{y} - \mathbb{E}_Y[Y])^2 + (\mathbb{E}_Y[Y] - Y)^2\right] \\ &\overset{3}{=} \mathbb{E}\left[(\hat{y} - \mathbb{E}_D[\hat{y}])^2 + (\mathbb{E}_D[\hat{y}] - \mathbb{E}_Y[Y])^2\right] + \mathbb{E}\left[(\mathbb{E}_Y[Y] - Y)^2\right] \\ &\overset{const.}{=} \mathbb{E}_Y\left[\mathbb{E}_D\left[(\hat{y} - \mathbb{E}_D[\hat{y}])^2\right] + (\mathbb{E}_D[\hat{y}] - Y)^2\right] + \mathbb{E}_Y\left[(\mathbb{E}_Y[Y] - Y)^2\right] \\ &= \mathbb{E}_Y\left[\text{Var}(\hat{y}) + \text{Bias}\left(\hat{y}, Y\right)^2\right] + \text{Noise}(Y). \quad \square\end{aligned}$$

[4]

In the form of the bias-variance decomposition, the following interpretations of bias, variance and noise are presented. The variance of the model can be seen as a measure of how much the error changes if a different training set was used, that is, how much better is the current model from the average predictor? Thus the variance is a measure of the generalisation performance beyond the training data and so indicates overfitting. The bias can be seen as the minimum error achieved after infinite training due to the design of the model adding an unavoidable bias to the results. For example, applying linear regression to data following a quadratic relationship will be certain to have error due to the bias of the inflexible simple model being fitted. Similarly, the learnable parameters of an insufficiently complex network may only parameterise a function space of possible output mappings which does not include the target mapping, and so even with unlimited training, the closest approximation will still contain error inherent to the model design. This is the bias of the model. Finally the noise of the data derives from the ambiguity in the target distribution and cannot be overcome by changes to the model or training. Thus for non-zero noise, a model cannot minimise the cost objective function less than the error due to noise in the underlying data, and only trading off bias and variance above this minimum in a kind of 'uncertainty principle' [20].

The tradeoff between bias and variance arises due to the conflict between the two terms. A model can only optimise the objective cost function by changing its output predictions y_hat. However in general, bias and variance will not be minimised at the same point. For example,

[4]The outer expectations may be instead expressed as integrals with a density over the data inside the integrands to express expectations with respect to the training data, but I feel expectations are clearer and more pedagogical.

consider minimising $(x-a)^2+(x-b)^2$ to a non-zero value using an optimal x which is not a or b if $a \neq b$. The model which minimises bias is therefore in general different to the model which minimises variance as these expressions are minimised by different inputs. A minimum value of expected loss is therefore achieved as the sum of the noise and a minimum sum of variance and bias achieved by a model optimised balancing the two curves for bias and variance.

Conventionally, the bias-variance tradeoff has been taught as being the sum of two monotonic curves, leading to a predictable minimum. However recent work [53] presents a more subtle analysis which answers another long-standing paradox termed the 'double dip'. It has been observed [38] that making networks more complex can initially increase the generalisation error, up to a critical point after which further increasing complexity can begin improving the generalisation error to below the previous minimum, as illustrated in figure 16. This challenges the conventional belief that very complex networks will always overfit the training data and suggests previous neural network results could have been improved by increasing the complexity of the network past the initial increase in generalisation error. In fact this can be explained using the right perspective on the bias-variance tradeoff. Proposition 3.3 gives that the error due to the model is the sum of the bias and variance. Consider the generalisation error as a function of model complexity, a 1-dimensional error curve constructed as the sum of a constant offset due to noise and the two curves for bias and variance as functions of model complexity. Under this framework, it is somewhat immediate that the shape of the error curve depends on the individual bias and variance curves. As demonstrated in figure 16, some curves give rise to an initial minimum, then maximum before tending to a monotonic decreasing function.[5]

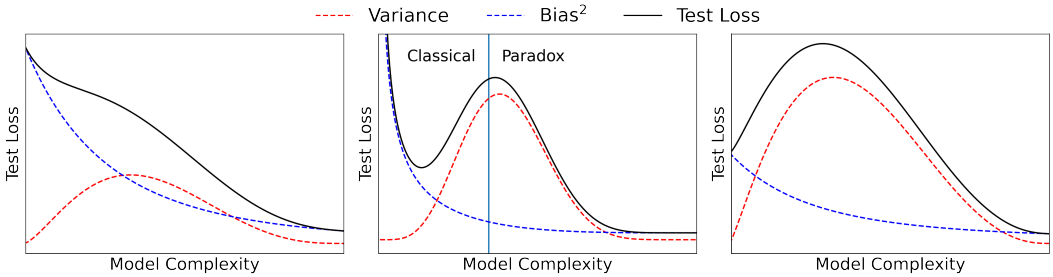


Figure 16: Bias-variance decomposition double-dip phenomena visualisation.

3.4 Regularisation

Recall from section 3.3 that a network is said to have *high bias* if it *underfits* the training data or *high variance* if it *overfits* the training data, and that these properties can be diagnosed by examining the *loss plots* on the training set and development set during training. *Regularisation techniques* are changes or additions to the network architecture or learning algorithm employed to reduce overfitting, and tuning the hyperparameters describing these changes allows for the network to be optimised to balance bias and variance.

There are two primary reasons a network may overfit the training data: network complexity and training time. Network complexity may be address by either shrinking the network or restricting the space of mappings the parameters allow the mapping to access. The output of a network is a composite mapping determined by a great many learnable parameters, each providing a degree of freedom to contort the mapping to the specific patterns or even datapoints found in the training data. The number of degrees of freedom and therefore complexity of the mapping is increased by the size of the network. The composite function can explore a much

[5]For visual clarity these plots omit intrinsic noise, which would raise the black test loss line uniformly.

larger space of possible mappings if the constituent functions are non-linear, since the composition of linear functions is itself linear, the space of curves the mapping can follow are restricted with less linearity in the network activation functions. Finally as mentioned in section 3.3, once the network has trained for long enough to learn the distribution as best as it can, it can then only improve the training loss on which it is optimised by learn the patterns or values specific to the training data and so the validation loss can increases as the network no longer generalises as well beyond the training set to unseen data.

A contrapositive to reducing network complexity to address overfitting is to reduce the dimensionality of the target mapping to address high bias from a simple network. If a target mapping has more input features, then learning this mapping becomes a higher-dimensional problem and so requires a more complex network. Thus if a distribution can be reformulated using less input features, a less complex network is required, or conversely an underfitting network will more easily be able to learn the target mapping. See section 5.1 for how this applies to the neural network implemented in the research component of this thesis.

Adding additional high-quality data will in general improve generalisation performance to serve as the best regularisation technique since the network can learn from a more complete sample of the target distribution without as many extraneous patterns arising from a larger training set. However, data collection can be expensive or additional data may not exist, for example in the case of medical imaging or historically needing to outsource image recognition to humans to label this data. Data augmentation will be discussed in section 3.7 to address a limited supply of data, however regularisation techniques are also available to improve generalisation performance given a dataset. To address complexity by shrinking the network, the width or number of layers may be reduced, *dropout* or *batch (re)normalisation* layers can be included. To address complexity by linearising the network, a *regularisation term* can be added to the loss function or new *random noise* can be added to the data on each epoch. Finally, overfitting arising from excess training time can be addressed with *early-stopping* or adding additional training data. These techniques must be balanced with the need to not underfit the data, occurring when the network is overly constrained and doesn't have the flexibility to properly reflect the patterns in the data. In developing a neural network, it is advised to begin with a minimal model with appropriate architecture and sufficient capacity to learn the target mapping and using loss plots to inform the iterative inclusion and tuning of regularisation techniques to optimise the development error.

The principal of *orthogonalisation* says that changes to the learning algorithm should be made in isolation to implementing or adjusting any regularisation techniques. For example, the learning rate should not be changed at the same time as the weight initialisation policy or gradient descent optimiser (see section 3.6). In addition to being able to isolate the effects of an individual technique, this is important because changes to the learning algorithm will directly impact how the network learns and therefore the generalisation error.

Dropout *Dropout* was introduced in 2014 by [49] and is a widely used technique for regularising an overfitting neural network. For each batch of training data (one *iteration*), dropout removes nodes independently during training, each with some probability p, and uses all nodes at test time. This regularises the network in two ways. Intuitively, a network with less nodes will be less complex and so less able to overfit the training data. Moreover, the probabilistic removal of any node ensures that no later activation can weight any specific node with too much significance, as it will not always be able to expect it to be present. In practice, dropout can be achieved by including a *dropout layer* between each layer of the network and setting values to zero, effectively eliminating the incoming and outgoing connections and calculations associated with a removed node. This has the effect of spreading weights more evenly across a layer without concentrating

as much on any one node. An additional consequence documented in the original paper [49] is to prevent the phenomenon in which nodes of later layers learn to account for sub-optimal outputs from previous layers, which blocks later learning from correct the mistakes. Nodes conspiring in this way create co-adaptations which may not generalise to unseen data, even if a more optimal weighting is possible without the coupling, since learning cannot change it. Note that because different nodes are used on each iteration, the loss plot will not be a smooth curve, and so a weighted average is often used. For the weights to be compatible at testing with the weights at the training stage, the expected value of activations are preserved by dividing the activation functions by $1 - p$, sometimes referred to as *inverted dropout*. In practice, a simpler model is first considered to confirm the network can learn effectively with dropout probability $p = 0$ and complexity or regularisation such as increasing p is only introduced later if regularisation is required, with the optimal value of p being a hyperparameter.
[6] [7]

Network Depth vs Width It is worth noting that it is often preferable to increase the number of layers (depth) of a neural network than to increase the number of nodes in each layer (width). As mentioned before, the output of a neural network is ultimately a composition of many (non-linear) activation functions, each weighted to produce a mapping approximating the target distribution. It is in this sense that a deeper and wider network has a greater *capacity* because the space of possible functions attainable by varying the weights is larger and so it is more likely that there is a closer fit to the target mapping within this space, if the right combination of weights is found. Although both increasing the width or depth of a network allows the overall mapping to be more complex, a wider network is better able to learn the individual datapoints in the training set, while a deeper network is better able to learn features at different levels of abstraction. For example, a neural network with just one hidden layer and as many nodes as training examples would be capable of eventually learning the exact mapping for each datapoint in the training set to achieve a zero test loss but without generalising beyond the test data[8]. It would instead be better to use a deeper network so that it has the capacity to learn more complex underlying patterns in the structure of the data. There is a limit to this policy however as with sufficient capacity and insufficient regularisation, the model will instead learn the training data and not be able to generalise.

Batch Normalisation *Batch normalisation* or *batchnorm* normalises the activation of a given node over the current minibatch and rescales to set the mean and variance to be equal to two learnable parameters. This was proposed in 2015 [26] to address the shifting of the inter-dependent distributions of activations (known as *internal covariate shift*) through the layers of a deep network and the consequence of this on gradient descent. An activation[9] taking values $a_1, \ldots, a_m$ over a minibatch $\mathcal{B}$ is normalised and rescaled as

$$\begin{gathered} \hat{a} = \frac{a - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}}, \\ \text{where } \mu_{\mathcal{B}} = \frac{1}{m}\sum_{j=1}^{m} a_j, \text{ and } \sigma_{\mathcal{B}}^2 = \frac{1}{m}\sum_{j=1}^{m}(a_j - \mu_{\mathcal{B}})^2, \end{gathered} \tag{4}$$

[6] Include an image of nodes not being included randomly.
[7] There's probably some maths to include for dropout.
[8] This is unless the test set included all possible input-output pairs for an injective target mapping.
[9] Here the superscript notation $a^{(i)[\ell]}$ for layer number ℓ and index i is dropped for readability

where the small value ϵ is included to prevent divide-by-zero errors and is often taken to be 10^{-8}. Now that the activation has been normalised, it is given a learnable standard deviation γ and mean β through multiplying 4 by γ and adding β.

$$\frac{a_i - \mu}{\sigma}\dot{\gamma} + \beta.$$

Without normalising the activations, a parameter update in an earlier layer could significantly change the distributions of the outputs of later layers. This is an issue because gradient descent updates weights via backward propagation through the layers but assumes that the weights are updates simultaneously, because the weight updates depend on the estimate and gradients derived using the current weights. However, once the later layers are updated, the estimates and gradients would in fact be changed and so gradient descent would now compute different parameter updates. This moving target presents an issue because the weights of one layer are tuned to expect a certain distribution of activations from the previous layer, but an update to the parameters of the previous layers cause compounding changes to these distributions through the layers. In particular, parameters in later layers will not necessarily receive inputs following the distribution they have already been tuned for, thus making the earlier learning obsolete. The changing distribution of activations through the layers of a network is known as *internal covariate shift*. With batch renormalisation however, a gradient descent step can update weights and biases without affecting the standardised distribution of activations later in the network. This makes learning more robust and improves the speed of convergence. Moreover, normalising activations between layers prevents them from exploding to large values or vanishing to small values and so non-linear activations will not be saturated, thus working to prevent vanishing gradient.

Full normalisation removes potentially valuable information and so the standard mean and variance are added as learnable parameters (that is, they are updated by gradient descent alongside the weights and biases) to reintroduce this. Since batch-normalisation completely determines the bias in the data, there is no need to have bias terms in the network itself. It is for this reason that it is common to use the terms 'parameters' and 'weights' interchangeably. Moreover, the original paper [26] claims that batch normalisation makes dropout redundant due to the additive and multiplicative noise being incorporated by each mini-batches' own mean and variance. Batchnorm is also an efficient technique to implement in most cases, however there is a specialised class of neural networks called *recurrent neural networks (RNNs)* which allow some activations to bypass intermediate layers in deep networks, thereby complicating the implementation of batchnorm. This thesis considers exclusively the fully-connected neural network architecture as an ideal foundation for a 4H-level introduction to deep learning and the research component of this thesis implements a fully-connected network to demonstrate a baseline potential for the technique, hence RNNs will not be discussed further. Both dropout and batchnorm are very powerful and widely, however it is best practice to not use the two techniques together as their effects on variance shift can conflict [31].

Batch Re-Normalisation Batch normalisation applies differently during training and inference (when making predictions or during testing) and batch renormalisation addresses this inconsistency. During training, the normalisation statistics on each layer for batchnorm are computed individually for each batch, however at testing time the training examples are not passed in batches but instead the forward propagation is computed on individual datapoints. During testing, batchnorm could instead 'normalise' each activation using the statistics over the whole training set, but in practice the normalisation statistics used during inference are instead weighted moving averages of the minibatch statistics logged during training. However, the summary statistics of small or non-i.i.d. minibatches are more likely to differ from the aggregate,

thus introducing an inconsistency between training and testing. This issue can be resolved using *batch renormalisation*, first introduced in 2017 [25] by the same author[10] who proposed batchnorm. Batch renormalisation addresses this difference in normalisation by applying an affine transform of the normalisation using the weighted moving averages such that the expectation is consistent between training and testing. In effect, activations are calculated using the moving average normalisation while gradient descent is calculated using a fixed transformation of the batch-normalisation because batch re-normalisation proposes to consider this transformation constant for the purpose of calculating gradients. The author claims that batch renormalisation also makes weight initialisation unnecessary while additionally helping to avoid exploding or vanishing gradients because in expectation, the norm of the gradient is constant, thereby allowing for larger learning rates and so faster learning.

$$\begin{gathered}\frac{a_i - \mu}{\sigma} = \frac{a_i - \mu_B}{\sigma} \cdot r + d, \\ \text{where } r := \sigma_B/\sigma,\ d := (\mu_B - \mu)/\sigma.\end{gathered} \tag{5}$$

As in batchnorm, now that the activation has been normalised, it is given a learnable standard deviation γ and mean β through multiplying 5 by γ and adding β.

$$\hat{a_i} \cdot \gamma + \beta.$$

The principle being employed is to normalise using the moving averages μ and σ by treating r and d as constants during gradient descent. Now the normalisation of activations during both training and inference uses the moving average, but with gradient descent modified. Naively using the moving averages for normalisation during training was shown in the original batchnorm paper to lead to issues such as normalisation cancelling-out the effect of parameter updates on the loss function so no learning can occur.

Regularisation Term Increasing the weight of an input to a node will increase its contribution to the activation, while a subset of weights being over-weight may allow for the recognition of a pattern in the data. Thus, distributing weights more evenly will have a regularising effect by limiting the ability of the network to learn individual datapoints or specific patterns. Moreover, for non-linear activation functions such as *tanh* and the sigmoid function, the gradient is close to 1 near zero and so the functions are approximately linear. Recall that the composition of linear functions is itself linear and so if most weights are small, then the activation function will act approximately linearly on the small input value, thus the overall composite function will be linearised, reducing the complexity and function space of the mapping, thereby regularising the network. Small weights provide the additional benefit of avoiding vanishing gradient by reducing the saturation of activations, thereby increasing the speed of convergence.

[10] Interestingly, the author works for Google who were awarded a patent for this (widely used) technique in 2019.

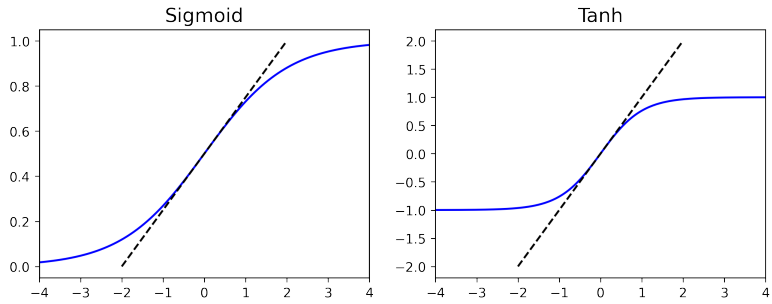


Figure 17: Small input values (weights) linearise sigmoid and tanh activation functions.

To minimise weights, a *regularisation term* or *regulariser* can be added to the cost function to penalise a large weight vector norm. The choice of norm and a factor λ scaling this norm are hyperparameters of the model to be tuned to optimise generalisation performance on the development set. The $L1$ and $L2$ norms (known as *lasso* and *ridge* regularisation respectively), are common choices and come with their own respective benefits, while *elastic regularisation* was proposed by [56] and uses a linear combination of the two to achieve a compromise and benefit from both forms. The form of the loss function built from an MSE loss and a regularisation term and the L1 and L2 norms are given by

$$\mathcal{L}(y, \hat{y}) = \mathcal{L}_{\mathbf{MSE}}(y, \hat{y}) + \frac{\lambda}{2m}\|w\|^2, \text{ with norms}$$
$$\|\underline{x}\|_1^2 = \frac{1}{n}\sum_{i=1}^{n}|x_i|, \qquad \|\underline{x}\|_2^2 = \frac{1}{n}\sum_{i=1}^{n}(x_i - \bar{x})^2 .$$

Equivalently, $\frac{\lambda}{2}\|w\|^2$ can be added to the cost function. Note the factor of $1/2$ is included by convention so that λ is the derivative of the regularisation term in gradient descent to make its meaning more explicit.

Gradient descent minimises the cost function of the network by updating its parameters, so the regularisation term encourages the learning algorithm to minimise weights as well as the sum of losses. The $L2$ norm will penalise particularly anomalous weights especially strongly while the $L1$ norm encourages *sparsity*, when features which carry little information are made very small which prunes less useful quantities while reducing the complexity of the network with zero values. The $L1$ and $L2$ norms can also be referred to as the *Frobenius norm* and *spectral norm* respectively in the context of vectorising using matrices.

Random Noise Random noise can be added directly to the normalised input data for a regularisation effect to help generalise the model by making overfitting more difficult. The input data should however be normalised or standardised first to ensure the noise has the same effect on all of the input features.

Adding noise to training data will help the network to learn to identify the underlying pattern and tolerate deviations from real-world data. Heuristically, each training data point will be harder to pinpoint behind noise and so the model will not be as easily able to learn the training data, instead better learning the underlying mapping as desired. In [52] the author explains that adding random noise to the inputs can be equivalent to the method of adding a 'Tikhonov' (L2) regularisation term to the cost function. In practice, this noise is added each time a training example is used in learning so that a different value is called in each epoch, allowing the network to better generalise as it cannot learn specific datapoints if they change on each epoch. The random

noise should not however be added in validation or testing, unless used for data augmentation to better reflect the anticipated real-world data, such as grainy images. Usually Gaussian noise is used where the variance of this distribution is another hyperparameter of the network, since a large variance could dominate the smaller underlying pattern.

Recall from Proposition 3.3 the bias-variance tradeoff for the expected MSE-loss[11]

$$\mathbb{E}[\mathcal{L}(y, \hat{y})] = \mathbb{E}[(y - \hat{y})^2] = \text{Bias}(\hat{y})^2 + \text{Var}(\hat{y}) + \sigma^2,$$

where $\hat{y}$ is an estimate for y and σ^2 is the noise in the data. Our network only has control over $\hat{y}$ and so a lower-bound for the minimum expected loss is the noise σ^2 which comes from the data directly. Thus adding random noise to the data will add to the intrinsic noise of the data, and therefore to the minimum expected loss, via the variance of the noise. The training loss is therefore expected to be higher in a network using random noise for regularisation, independent of the underlying accuracy. Because random noise is not added during testing or inference, these loss values will not be affected.

Early-Stopping If left to train for too long, the loss plot for the development set may begin to rise after initially decreasing when the network begins to learn the validation set itself and not the underlying pattern of the target distribution. *Early-stopping* is the process of using the weights and biases from the point of inflexion in the final trained network. Training a network for too long can allow the model to overfit the training data and thereby reduce the ability of the model to generalise. TensorBoard has the functionality to update this plot live[12] which is a useful way to terminate training once a plateau or divergence begins to occur, saving training time. Otherwise, a network can be designed to log its parameters each epoch and once training has completed the programmed number of epochs, the trained network automatically uses only the parameters from the point of lowest validation error. On very large datasets, a relatively small network may not be expected to have the complexity to overfit the training data, in which case early-stopping may not present a benefit as was found empirically by [55].

Additional Data Adding more training examples provides the model with a more representative sample of the target mapping to approximate, and so high-variance (overfitting) models will benefit from more data as spurious patterns found in the training set become less prevalent than those reflective of the true distribution, thus improving generalisation. A high-bias (underfitting) model however will not necessarily benefit from additional data as it is already too simple to capture the patterns in the current training set. Collecting new data can be expensive or additional data may not exist, for example in the case of medical imaging with only a limited number of x-ray scans showing a particular condition, and historically image data has been labelled by outsourcing to humans. In such cases two methods to increase the training data are to generate synthetic data or to augment existing data.

An advanced technique in artificial intelligence called Generative Adversarial Networks (GANs)

There exists an advanced technique called *generative adversarial networks (GANs)* first proposed in 2014 [19] using two agents working in opposition via a zero-sum game to generate synthetic data. This method can be very powerful for this use case and is widely used in generating synthetic market data. Specifically in the setting of options pricing, the many parameters determining a unique options contract and the frequency of trading makes tick-by-tick data archiving impractical so data are typically recorded at the end of each trading day. This report uses 18 years of daily-quoted data from 2002-2020 with nearly 500,000 training examples and so generating additional synthetic data will not be necessary.

[11] Technically, the loss is the expectation of this expression with respect to the data.

[12] Refreshing manually or every 30 seconds automatically.

3.5 Minibatches

Batch gradient descent calculates the cost and averages the gradient on the whole training set so that each gradient descent step is taken after the whole training set is processed. In this setting, iterations and epochs coincide. This makes every step well-informed to travel in the right direction but is computationally expensive as many calculations and training examples are used to make each step of progress, even with vectorisation. Instead of waiting to process every training example to make just a single step, a smaller number of training examples per gradient update step will be used, called a *minibatch*. This reduces the number of calculations required per step, at the expense of each gradient estimate being less accurate due to small minibatches sampling the distribution with more bias. There is a tradeoff for convergence speed to be made in determining the minibatch size hyperparameter: gradient estimation accuracy, vectorised computation and update step frequency. The forward and backward propagation steps for every training example in a batch will be vectorised to be computed in parallel to be more efficient per training example, while smaller minibatches require less overall computation and can make more steps per epoch. The overall convergence time will depend on the exact balance of these factors.

A popular neural network design choice is to use *stochastic gradient descent(SGD)* together with *momentum*, where SGD is the use of minibatches of size 1 and momentum is an optimisation of gradient descent which will be discussed in section 3.6 to smooth variations in the stochastic update steps arising from small minibatch sizes. It is noted in [24] that a large minibatch size may lead to poor generalisation performance, known as the *generalisation gap*. In [27], the authors observe that deep networks typically contain many sharp minima which large minibatches are unable to escape.

In the case of a very large training data set, smaller minibatches may be required to store the entire minibatch in the local memory[13] to avoid a critical slowing down due to the inefficiency cost occurred in transferring data between disk and local memory. Since computers operate in binary, it is optimal to use minibatch sizes which are a power of 2. A balance should then be found as a power of 2 batch size which is small enough to generalise sufficiently well and fit on the GPU, but large enough to benefit overall convergence speed.

3.6 Gradient Descent Optimisers

There exist *gradient descent optimisation algorithms* such as those named *RMS Prop* and *ADAM* which are designed to improve convergence speed over vanilla gradient descent as described in section 2.3 using optimisation techniques such as *momentum* and *learning rate decay*. Momentum incorporates a weighted average of gradients to smooth variation in parameter updates to take a more direct path to a minimum, while learning rate decay reduces the learning rate on each gradient descent step to decrease the update step size and improve convergence speed. As mentioned in section 3.5, SGD together with momentum is a popular gradient descent optimisation algorithm. Otherwise, learning is usually performed by *minibatch gradient descent* with an optimiser such as ADAM. More advanced adaptations exist, including AMSGrad and AdaMax derived from ADAM and Nesterov derived from SGD. Finally Nadam is a combination of Nesterov and ADAM, with ADAM being short for ADAptive Moment estimation and Nadam being short for Nesterov-accelerated ADAptive Moment estimation. The very recent review [44] of deep learning optimisers found SGD, momentum and Adam to be the most frequently mentioned optimisers in ArXiv abstracts and in extensive benchmarking tests, the best optimiser depends heavily on the specific problem, but that Adam is not consistently outperformed by newer optimisers. The

[13] If a minibatch does not fit in CPU/GPU/TPU memory, the inefficiency cost in transferring data between disk and local memory will cause performance to 'fall off a cliff'.

authors also observe that evaluating multiple optimisers using default hyperparameters generally produces comparable results to careful tuning of the hyperparameters of a single fixed optimiser.

A useful analogy imagines gradient descent as the motion of a ball rolling on a hill under gravity which extends well to momentum and ADAM. Since gradient descent updates the inputs to traverse the cost surface in the direction of steepest gradient and steeper gradients lead to larger update steps, consider a ball which is free to roll on a curved surface and represent one update step by moving under gravity along the steepest negative gradient in one time step. After enough time steps the ball eventually settles in a (local) minimum.

Figure 18 provides visualises gradient descent paths for different optimisers discussed in this section.[14]

Momentum Momentum is similar to vanilla gradient descent except instead of subtracting the learning rate times the gradient at the current point, an exponentially weighted moving average of the past gradients is used in place of the gradient at the current point. This ensures the update step is in the prevailing direction of the previous steps. In particular, when the minibatch size is small, the gradient updates using momentum can allow the stochastic variation in each gradient to cancel out so that each step is instead taken in the prevailing direction of the recent gradients, generating a more direct path towards the minimum. Once the algorithm has traversed in a consistent direction for several steps, this indicates it is indeed travelling down the greatest slope and this creates a heavy weighting in the sum, providing resistance to small variations in future steps. Momentum derives its name from the analogy of a ball rolling on a hill in which this process is similar to the momentum in the ball which resists changes to its velocity due to inertia. In particular, once the update steps reach the bottom of a valley and the slope begins to increase, momentum means the ball initially continues to roll in the same direction under momentum, only changing velocity gradually. The t^{th} update step for a parameter w can be [2] written

$$\begin{aligned} v_t &= \mu v_{t-1} - \alpha_{t-1} \nabla f\left(\theta_{t-1} + \mu v_{t-1}\right) \\ \theta_t &= \theta_{t-1} + v_t, \end{aligned}$$

where μ is the momentum coefficient, α_{t-1} is the learning rate at step t and v_t is the velocity. The notation $\nabla f\left(\theta\right)$ denotes the partial derivative of f w.r.t. the parameter θ.

[14] See `https://imgur.com/gallery/Zw2sTpk` for a pleasant animation of these visualised gradient descent paths.

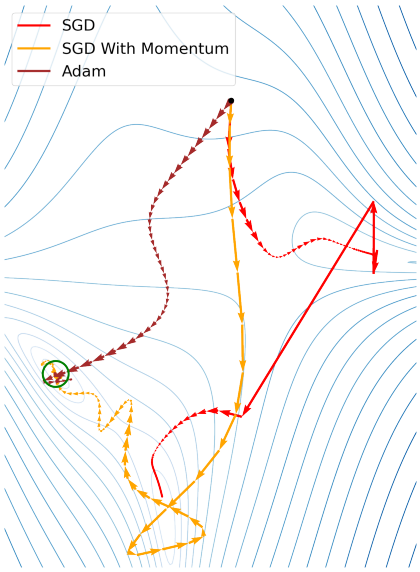


Figure 18: Momentum smooths stochastic jumps and the Adam optimiser traverses the surface efficiently.

Nesterov Nesterov [39] is an improvement to the standard momentum optimiser. An update step under momentum is equivalent to taking a step in the direction of the local gradient and then taking a second step in the direction of the momentum. Conversely, a Nesterov update step takes the momentum update step before calculating the current gradient, to then take a second step in the direction of the gradient at the new position after the momentum step. [50] This change ensures that each sub-step is taken in the direction of steepest gradient from the point where it is made, whereas in momentum both the slop update and the momentum update are calculated from the initial position. The momentum step is the same in both algorithms, however the slope step is more appropriate at the time it is made in a Nesterov update. The overall update step is the combination of the two sub-steps and is given by

$$
\begin{aligned}
v_t &= \mu v_{t-1} - \alpha_{t-1} \nabla f\left(\theta_{t-1}\right) \\
\theta_t &= \theta_{t-1} + v_t = \theta_{t-1} + \mu v_t - \alpha_{t-1} \nabla f\left(\theta_{t-1}\right),
\end{aligned}
$$

Within the ball analogy, this could be imagined as two sentient balls which make discrete jumps rather than continuous rolls: the ball representing the momentum algorithm will make a jump in the average direction of the previous jumps and then make a second jump in the direction of steepest slope as calculated from its *previous* position and not its current position, while the Nesterov ball jumps in the average direction of the previous jumps and then jumps in the direction of greatest slope from where it currently lays. In reality the jumps of the momentum-ball can be made in either order but this ordering highlights the contrast to the Nesterov-ball strategy.

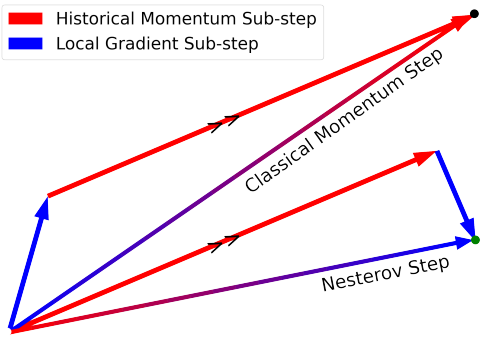


Figure 19: Nesterov sub-steps are computed locally for a better resultant update step.

Learning Rate Decay Learning rate decay (also known as a *learning rate schedule*) is an optimisation of gradient descent in which the fixed learning rate is instead replaced by a variable learning rate which decreases on each update step to improve convergence speed. Intuitively, the randomly initialised weights will not be optimal at the beginning of training, while smaller update steps will allow the network to converge faster to the minimum by limiting overshooting. This is commonly achieved by either dividing the original learning rate by $\sqrt{t}$, or geometrically as $\alpha = p^t \alpha_0$, where t is the iteration or epoch number and p a hyperparameter controlling the rate of learning rate decrease. With a fixed learning rate, once the local neighbourhood of a minimum is reached, large learning rates can lead to overshooting the minimum and so continually reducing this jump size allows faster convergence arbitrarily close to this minimum, if the parameters can reach the area of the minimum before the step size becomes too small. This can also be used together with momentum and can help avoid momentum update steps from overrunning the minimum by decreasing the step size on each step. It is also possible to use different learning rates for each input feature, all decaying independently. Advanced optimisations of gradient descent incorporate learning rate decay in conjunction with scaling the learning rate dynamically to optimise convergence, known as an *adaptive learning rate*.

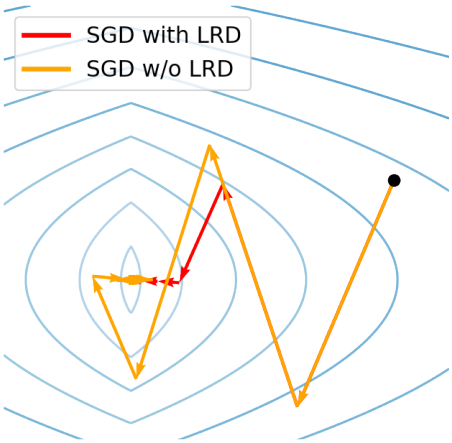


Figure 20: Learning rate decay can accelerate convergence near a minimum.

Adam The Adam optimiser combines both learning rate decay, averaging by the first-moment of gradients via momentum, as well as scaling update steps by the second moment of gradients,

hence the name derives from ADAptive Moment estimation. In the rolling ball analogy, a ball which has built up significant momentum from a long steep slope will overshoot a minimum by a large margin and may be unable to change direction in time. Dividing the update step size by the weighted average of the squared gradients will limit the step size after large gradients and increase the step size after a region of all gradients. This also addresses plateaus on the cost surface in which a region of small gradient leads to slow learning because update steps in vanilla gradient descent are proportional to the size of a near-zero gradient, much like a ball travelling slowly on a nearly flat plateauing surface. Scaling by the second moment of gradient is analogous to friction with air[15] in the rolling ball analogy introducing a terminal velocity. The original 2014 paper [29] suggests several possible benefits of Adam to include:

> Straightforward to implement. Computationally efficient. Little memory requirements. Well suited for problems that are large in terms of data and/or parameters. Appropriate for non-stationary objectives. Appropriate for problems with very noisy/or sparse gradients. Hyper-parameters have intuitive interpretation and typically require little tuning.

The weights sums for the first and second moments of gradient introduce two hyperparameters β_1, β_2, however the default values of 0.9 and 0.999 respectively are empirically found to be very robust in many applications so it is typically worthwhile to instead focus on other hyperparameters during tuning. The update step for a parameter w using the ADAM optimiser is given by

adam update step equations.

3.7 Data Normalisation & Augmentation

Data Normalisation If the units of one feature of the input data are such that the numerical values of each coordinate differ by orders of magnitude, then the larger values will dominate weights and activations. This limits the contribution of features with smaller values, especially if the cost function incorporates a regularisation term to penalise large weights. The net effect is to lose information from smaller features and therefore impacting the convergence of gradient descent. Moreover, an affine coordinate transformation may reveal two axes with numeric values in the training set which differ by orders of magnitude. Visually, for a training set with 2-dimensional inputs, this would be plotted as similar to an eccentric ellipse with some orientation. Intuitively, gradient descent on such a surface would be dominated by traversing the longer axis. Moreover, minimising a symmetric loss function leads to errors of similar magnitude, which is often not appropriate for unnormalised data when it is the percentage error which is a more appropriate measure of loss. In the absence of an a priori understanding of the relative importance of different features, it is therefore common to normalise each feature of training data to have zero mean and unit variance. To address the issue of non-feature axes existing on different scales, it is in general better to use principal component analysis (PCA) to normalise all training data via projection on to a unit sphere when possible. If data are normalised on a per-feature level, then the resulting distribution is a unit $N-$cube, where N is the dimension of the input feature vector, but then the range of data along the main diagonal axis $\sqrt{N}$, while data along a coordinate axis is contained in a unit-length interval.

[15] The friction is considered to be with the air as friction with the surface would depend on the weight of the ball via its normal force to the surface.

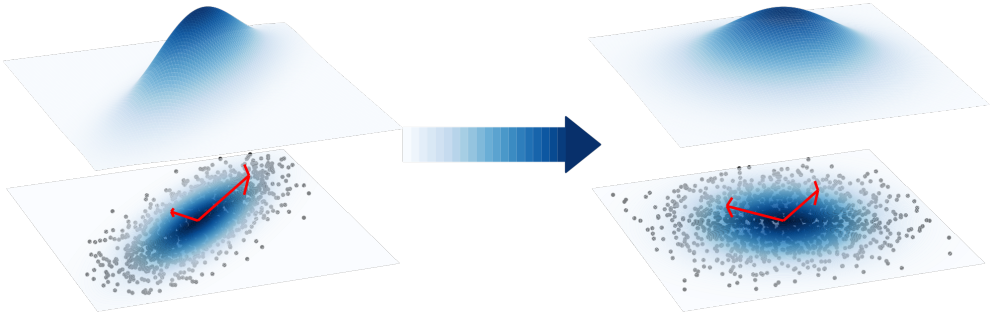


Figure 21: Normalise over all axes for direction-agnostic frequencies and scales.

This allows gradient descent to take a more direct path towards the a minimum. Usually a vector x of all instances of an input feature across the entire dataset will be normalised to have zero mean and unit variance or rescaled to the unit interval $[0, 1]$ through either

$$x' := \frac{x - \bar{x}}{std(x)}, \quad \text{or} \quad x' := \frac{x - \min(x)}{\max(x) - \min(x)},$$

which can be less computationally expensive. It is important to normalise the whole dataset and not just the training set so that the network is trained on data with the same distribution as its test set. During inference or testing, the same normalisation values must be used as in training. Centralising the data reduces the average modulus of the values and can improves symmetry to reflect the symmetric nature of non-linear activation functions to limit saturating activations which would cause vanishing gradient.

One application of a neural network can be for parameter estimation to calibrate a model. It is not uncommon for models to assume normal distributions and so it can be useful in these instances to transform the data, for example using the Box-Cox method, to more closely follow a normal distribution.

Data Quality High quality training data should follow the same distribution as the anticipated real-world data. Once data are collected, it is important to pre-process this data to remove duplicate training examples or those with missing entries, as well as other possible context-specific issues requiring data cleaning. Data pre-processing includes normalisation and auditing the data to confirm it follows the expected distribution, which may include checking the expected content or quality of a subset of images used to train an image classification network, for example. In the case of the options market data used in this research, company mergers can lead to options contracts been quoted under both companies' identifiers and so duplicate entries were filtered out. Moreover, there is theoretical justification for using the ratio of two contract parameters (see section 5.2) as a single input feature instead of the two parameters individually. Thus once the data was sources from a database, one step in the pre-processing included deriving this new quantity. Another data pre-processing step could be the addition of random noise to training data to aid generalisation (although this is more commonly applied in each epoch during training.

Data Augmentation An inexpensive method to generate additional training data is to augment existing data, for example by rotating, reflecting, blurring or applying filters to images. In addition to generating additional training data, this can also have the specific benefit of teaching the network which patterns are not important for determining the desired output, for example

a cat classifier should identify an image of a cat irrespective of rotation or reflection, while blurring and filters also help simulate distortions which may be expected in real-world data, such as images taken personally from a mobile phone rather than a professional stock image which may be more or less easy to source labelled data for. Generally, augmented or synthetic data are of a lower quality or adds less new information than a wholly original new training example. It can be good practice to use lower quality data in training and higher-quality data which is likely less numerous but more representative of the expected real-world data in the validation and test set. This has the benefit of training the network on a large volume of training examples, while hyperparameter tuning optimises the network for the anticipated data.

3.8 Hyperparameter Optimisation

Hyperparameter tuning selects parameters to configure the network, such as the learning rate and dropout probability, to optimally converge to a minimal validation loss (generalisation error). This is therefore an important stage of neural network design and development. Different hyperparameter configurations can differentiate a network which learns a target mapping to a high accuracy and networks which fail to learn at all or settles on a local minimum. Inference from loss plots as discussed in section 3.3 is an important tool for informing changes to hyperparameters by diagnosing under and overfitting. An additional symptom of an underfitting network (or bad data) is a near-constant output at some average of the training data. This is an easy sub-optimal minimum for a simple network to learn and minimise its losses with just a single output value to learn.

Gradient descent considers only the loss on the training set and so to avoid overfitting, the network can be optimised for generalisation error on an unseen validation or development set by minimising this validation loss as a function of controllable hyperparameters, to help generalise beyond the training set. There are often many hyperparameters so can be itself a high-dimensional optimisation problem. Common approaches to efficiently trial different hyperparameter configurations include *grid search*, *random search* or *Bayesian optimisation* to explore the *hyperparameter space* and identify an optional set of hyperparameters. Training networks can be computationally expensive and so hyperparameter tuning methods are often judged by the test loss of the best configuration found after a fixed number of trials. Hyperparameters can be tuned manually by experts using rules of thumb, experimentation, intuition and an understanding of the effects of each change, however advanced systematic methods regularly outperform human experts [30] due to the complex inter-dependence between many hyperparameters. A general rule of thumb is that the learning rate is the most important hyperparameter and so initially this should be the first focus for tuning. For example, figure 22 demonstrates how small differences in the learning rate can differentiate between convergence to the minimum and divergence.

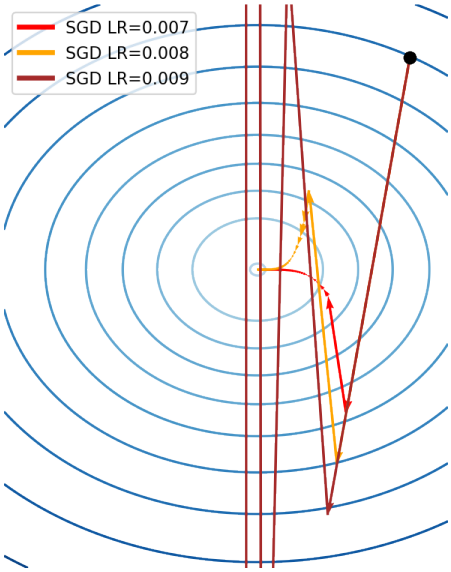


Figure 22: An optimal learning rate can determine both the speed and possibility of convergence.

More comprehensively, in the literature review [54], the authors suggest the following orders of preference for which hyperparameters to tune first: learning rate, momentum, minibatch size, number of layers, learning rate decay, regularisation term.

Grid Search Grid search is the most simple method to systematically explore a hyperparameter space and uses a *coarse-to-fine* approach to identify an optimal set of hyperparameters. Equally-spaced sample points along each dimension (hyperparameter) in the high-dimensional hyperparameter space are sampled so that together these points define a regular square lattice or grid over the hyperparameter space. Each set of hyperparameters corresponding to a point on this grid is individually trialled and the region of lowest cost is considered next at a finer resolution. For example, figure 23 depicts a 2-dimensional coarse-to-fine search process whereby each iteration zooming in on the best quarter-grid, retaining half the previous range of each hyperparameters. The new smaller subspace is again tested along a grid and each successive lowest-cost region becomes the next smaller subspace. In this way, an initially coarse grid is made increasingly fine in a localised region of minimal loss as a smaller subset of the points in the space are considered. Once this process completes some determined number of iterations, the set of hyperparameters corresponding to the gridpoint with the with the least cost is used to train the final model.

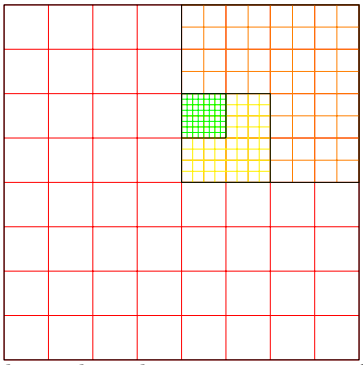


Figure 23: 'Coarse-to-fine' grid search evaluates points on a finer sub-grid at each iteration.

There are two significant limitations of grid search. Firstly, the number of configurations in a grid grows as n^d for a grid of n points in a $d - dimensional$ hyperparameter space, making it infeasible to sample more than a small number of points for each hyperparameter. Although this method searches a space in a systematic and intuitive way, there is significant computational waste arising from each hyperparameter value being re-trialled n^{d-1} times. For a hyperparameter which has macro behaviour largely independent of other hyperparameters (for example, a very large learning rate which always diverges), many trials will be wasted evaluate all n^{d-1} possible choices of the different hyperparameters when suboptimal learning will result each time this hyperparameter is set to a poor value. Both of these issues are addressed by random search.[16]

Random Search In each trial of a random search, random points are chosen for each hyperparameter. Given the same network and computational resources, it is found [3] theoretically and empirically that random search finds models which are as good or better than models identified by grid search. Random search allows for the whole space to be sampled on average but with trials on hyperparameters which one would not intuitively choose, and so it is suggested that this benefit arises from considering a larger space than being restricting to a grid. Intuitively, as depicted in figure 24, the global minimum of the surface being sampled may be in a very steep neighbourhood in an otherwise homogeneous region of the hyperparameter space. In such a scenario, a grid search may not identify this locally minimal area in a coarse grid search and subsequent grids will zoom in on a region which does not contain this minimum. Random search addresses the above-mentioned limitations of grid search by more densely sampling each hyperparameter. A grid-search of n^d trials would trial n different values of each hyperparameter, whereas random search would consider n^d different values. Moreover, the standard implementation of grid search (without any random selection) will limit the choice of n by the available computational and time resources to complete n^d trials, whereas random search can choose any number m samples, which is more practical.

[16]Before systematic search methods are employed, manual trials and tuning should be completed to identify sensible ranges and high-level patterns in more choices.

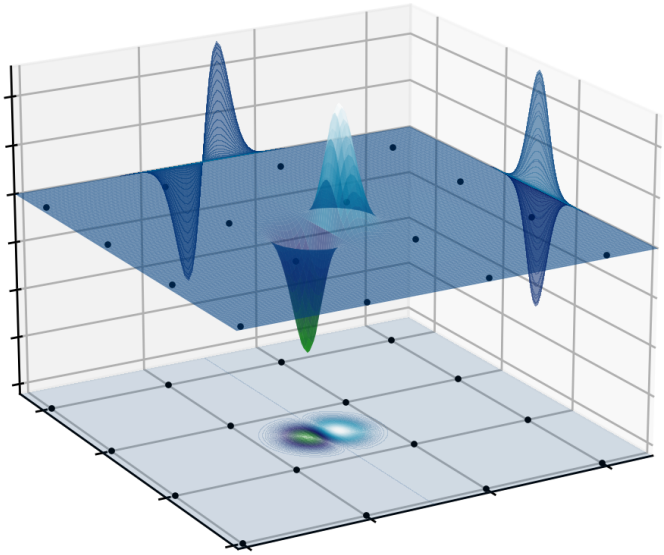


Figure 24: A sparse grid search can easily miss a narrow minimum.

Bayesian Search Bayesian search runs trials on points in the hyperparameter space which, given the previous trials, are likely to have small loss values or require further exploration. For example, continuous function by definition maps similar inputs to nearby points in the image, thus an under-explored region or points near other small values will be valuable choices for future optimisation trials. This heuristic is implemented systematically by Bayesian search. Modern Bayesian optimisation techniques use Gaussian processes which is unfortunately beyond the scope of a report aimed at a general 4H student[17]. Bayesian optimisation assumes a prior distribution for the target function and uses evaluations of sample points to update this prior to a posterior distribution on the space of possible target (*objective*) functions. The next sample point is chosen by an *acquisition function* according to the posterior distribution. The Bayesian optimiser can be designed using different acquisition functions, such as the expected improvement or probability of improvement (in the objective of maximising or minimising the function). In the case of a minimisation problem such as minimising the cost as a function of the hyperparameters of a neural network, the point in the input space which maps to the least cost is the set of hyperparameters used to train the final network.

Each hyperparameter optimisation technique has its own pros and cons, namely the grid search method is very simple to implement but underperforms random search which is able to explore the hyperparameter space more thoroughly without being much more complex. Bayesian optimisation applied to hyperparameter tuning can dramatically outperform both grid and random search, however it is significantly more advanced and is only suitable for features of no more than 20 dimensions due to an exponential increase in the number of required observations and optimisation complexity. Because all of these optimisers and more are implemented in the standard neural network libraries such as TensorFlow and PyTorch, it is not meaningfully more difficult to apply any particular method in practice, as long as their limitations and benefits are understood. In the network implemented by this project, appropriate ranges for each hyperparameter were determined using manual tuning, before random search and TensorBoard visualisation were used to narrow these ranges and then Bayesian optimisation was performed within the space defined by these informed limits.

[17]However, Gaussian processes are useful and interesting so the interested reader is directed to [16]

3.9 Example Network Optimised

Recall from section 2.7 our example network to estimate the amplitude A and period ω of a sine curve given a set of sample points along the curve. Each training example is assumed to be a vector of 30 points $A \sin \omega x$, where the samples are always evaluated at the same x points. This section will demonstrate many of the optimisation techniques discussed in chapter 3 on a tangible pedagogical example, while introducing example code for this implementation in Python 3 using the TensorFlow library. The example is chosen to provide an accessible summary before discussion of financial theory and results of the title research is introduced.

Defining & Training a Model For a full list of imports, see `https://github.com/Connor-Tracy/Neural-Network-Option_pricing`. Some libraries and modules used in the snippets below were imported with common abbreviations for readability.

```
1 import numpy as np
2 from tensorboard.plugins.hparams import api as hp
3 import kerastuner as kt
```

All concepts from chapter 2 are automatically implemented as methods in the TensorFlow library, which is built on and compatible with the Keras library. Initially a model object is defined using the `Sequential` method of the `keras` library. Then each fully-connected and dropout layer is added with individual calls of the Dropout and Dense methods within the `layers` module of the Keras library. Each layer accepts arguments determining the width, weight initializer, activation function and regularisation term and multiple layers can be easily added in a `for` loop. The final layer should typically use a linear activation for regression problems. This network uses the sigmoid activation function and so weights are initialised using Xavier weight initialization. This network uses the elastic regulariser to capture the benefits of both L1 and L2 regularisation, for which the two hyperparameters considered hyperparameters.[18] Finally the `compile` method of the model is passed an optimizer and loss function as arguments.

Below provides a template for building a Keras model, with values corresponding to the network used in the sine example.

First provide the training data to the model to save normalisation parameters from, which are then applied universally across training and inference.

```
4 norm_layer = Normalization().adapt(df()[0])
```

Create a fully connected network using the `Sequential` method and add the `Input` layer shaped to accept the input feature vector of length 30.

```
5 model = keras.Sequential()
6 model.add( InputLayer(input_shape=(30,)) )
```

For each of the 2 `Dense` layers, add layers for either `BatchNormalization` or `Dropout` as

[18] If the network generalises best under only L1 or L2 regularisation, one might expect the tuned values for the coefficient hyperparameters to be close to 0 and 1 to reflect this.

appropriate. Weight initialisation and regularisation methods are called from Keras classes.

```
initializer = tf.keras.initializers.glorot_uniform()   # Initialiser for sigmoid activation
regularizer = keras.regularizers.l1_l2(0.000342, 0.000391)
for layer in range(2):
    model.add( Dense( units = 12, initializer, regularizer, 'sigmoid')
    #model.add( BatchNormalization(renorm = True) )
    model.add( Dropout(.234) )
```

Add the output layer with two nodes to output the two period and amplitude estimates.

```
model.add( Dense(units = 2, activation = 'linear') )
```

Compile the model object with the desired learning rate decay schedule, Adam optimiser, and MAPE loss function, to be trained later.

```
lr_schedule = tf.keras.optimizers.schedules.ExponentialDecay(
    learning_rate = 0.00487, decay_rate = 0.903, decay_steps = 4000, staircase = True)
model.compile(
    optimizer = keras.optimizers.Adam(learning_rate = lr_schedule),
    loss      = tf.keras.losses.MeanAbsolutePercentageError()) # Train, test loss & objective.
```

This code has been lightly simplified for presentation purposes and the complete scripts used in this research can be found at `https://github.com/Connor-Tracy/Neural-Network-Option_pricing`. Specifically, this is most useful wrapped in a `model_builder(hp)` function to create a model given a set of hyperparameters.

Once a model object is defined, it can be trained using the `fit` method, passing the full dataset, minibatch size, number of epochs and proportion of data use for the validation set as arguments. The results can be saved to a variable. TensorBoard `callbacks` are used to log quantities such as weights and cost values at each epoch. The loss plots for a run can be displayed using TensorBoard to diagnose under and overfitting, as well as to compare multiple logged runs, in order to inform manual hyperparameter tuning. Moreover, callbacks can be used to perform early stopping by ending training early after a set number of epochs fail to improve the validation loss. In section 6, the networks are trained until reaching a threshold MAPE value of 1%, and this was achieved using a custom callback.

```
logdir = "logs\\scalars\\" + datetime.now().strftime("%Y%m%d-%H%M-%S")
tensorboard_callback = keras.callbacks.TensorBoard(log_dir=logdir)

model = model_builder()
training_history = model.fit(*df, batch_size, epochs, validation_split,
                             verbose, callbacks=[tensorboard_callback])
```

Plotting the error surface of the trained model as a function of period and amplitude, observe in figure 25 how relative error is greatest closer to degenerate cases of near-zero amplitude or period, and small periodic increases occur where the period is near a multiple of 2π.[19] In these

[19]This is not merely an artifact of dividing by a small value in a MAPE calculation as similar patterns are observed using both MSE and MAPE losses.

regions there is insufficient information in the samples to deduce the amplitude or period from a small number of sample points. Note this is different to the high-dimensional cost surface as a function of hyperparameters which is minimised during hyperparameter tuning (and is the space sampled by trial runs during hyperparameter tuning), as instead this is the error surface for one trained model as a function of the two input features. Input data without a well-defined solution (infinitely many sine curves have zero amplitude, but they will all have the same input sample) in these degenerate cases contributes to the noise in the bias-variance decomposition derived in Proposition 3.3.

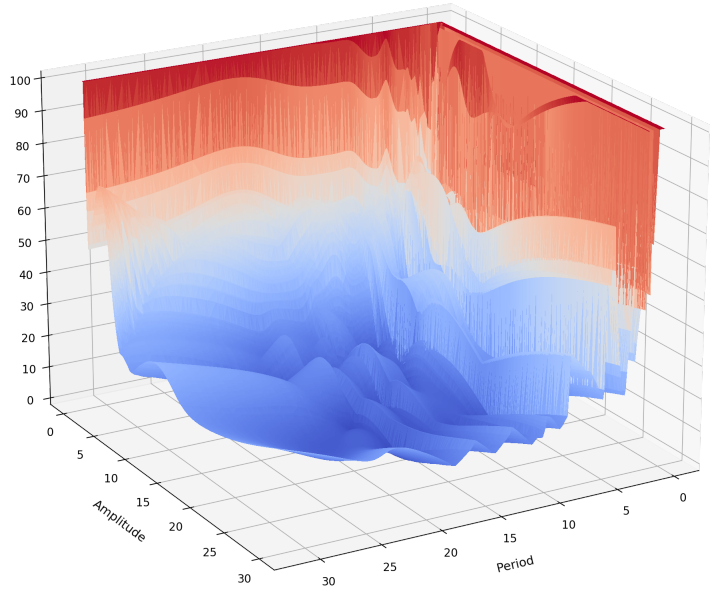


Figure 25: Degenerate cases increase the loss without well-defined solutions.

Hyperparameter Optimisation Automatic hyperparameter optimisation can be performed with the `kerastuner` library, using different optimisers such as grid search, random search and Bayesian optimisation by first defining a `HyperParameter` objects and specifying each hyperparameter with their respective ranges of values to be considered. These ranges of hyperparameters should first be identified by manual trials, and can be narrowed quickly using random search. The learning rate should be sampled on a logarithmic scale using `sampling = 'LOG'` because it acts multiplicatively and the range of values being considered spans many magnitudes. More generally, when the best scale for a hyperparameter is not known, first determine the magnitude by searching on a logarithmic scale. A minimal example for three hyperparameters could be implemented as below.

```
hp = HyperParameters()
hp.Float( 'learning_rate', min_value = 0.000001, max_value = 0.05, sampling='LOG' )
hp.Int( 'num_layers', min_value=1, max_value=2**6, step=1 )
hp.Choice( 'act_func', ['sigmoid','relu','elu'] )
```

Using the appropriate method from the `kerastuner` library, the desired optimiser object is defined and passed a model builder function, loss function, the HyperParameters object, maxi-

mum number of trials to be run, and a naming scheme to log the results of each run.

```
tuner = kt.BayesianOptimization(model_builder, objective = 'mse',
                                hyperparameters = hp, max_trials = 50,
                                project_name = "bayes" + datetime.now().strftime("%Y%m%d-%H%M"))
```

The `search` method of the optimiser object (here named `tuner`) is passed the same arguments as used for the `fit` method demonstrated above to train a single network. Manual tuning and random search should be used initially to identify sensible ranges for each hyperparameter and then the best performance is generally achieved using Bayesian search over a narrows hyperparameter space.

```
tuner.search(*df, batch_size, epochs, verbose, validation_split)
best_hps = tuner.get_best_hyperparameters(num_trials = 1)[0]
```

A dictionary of the optimal combination of hyperparameters found in the trials considered can be returned from the optimiser object after the runs are complete, and passed directly to a custom model builder function to be trained with these hyperparameters.

```
model = model_builder(best_hps)
training_history = model.fit(*df, batch_size, epochs, validation_split,
                             verbose, callbacks=[tensorboard_callback])
print(f"Average val loss:", np.average(training_history.history['val_loss'][10:]))
```

We can save a trained model to the current directory to use later with the `save` method and load a previously saved model as below.

```
model.save("trained_sine_model")
saved_model = keras.models.load_model("trained_sine_model")
```

The GitHub repository `https://github.com/Connor-Tracy/Neural-Network-Option_pricing` contains saved models for the sine example, options pricing accuracy network as well as each network used for timing pricing on different basket sizes.

4 Options Pricing and Monte Carlo Benchmark

4.1 Options Pricing Theory

Option Contracts A *derivative* is a financial contract which derives its value from another traded asset such as a stock price. An *option* is a derivative contract which entitles the buyer of the contract to the right, but not the obligation, to purchase or sell an underlying asset at a pre-agreed price up to a pre-agreed date. A *call option* gives the buyer of the contract the right to buy the underlying asset and a *put option* gives the buyer of the contract the right to sell the underlying asset. The pre-agreed price is called the *strike price* K and the terminal date is called the *expiration date* T or *maturity* as the buyer has the right to *exercise* the contract no later than the expiration date. A *European-style* option allows the contract to be exercised only at the expiry, whereas an *American-style* option allows the contract to be exercised at any time up to

the expiry[20]. More complex option contract styles exist and are generally called *exotic options*, while calls and puts are called *vanilla options*. Exotic options can often be decomposed into a an equivalent portfolio composing of vanilla contracts and cash with returns an interest rate known as the *risk-free rate*, and so it is common to study only vanilla options. A *basket option* is a contract which gives the buyer of the contract the right, but not the obligation, to purchase or sell a collection of assets at the expiry for a single agreed strike price, often representing the arithmetic or geometric mean of the constituent asset prices. An increasingly popular form of basket option is an *index option* in which the basket is a weighted collection of the values of each stock in an index such as the S&P 500, and so basket options can be very high-dimensional. making numerical estimates of the fair value of a contract computationally expensive.[21]

Options contracts are traded by corporations, hedge funds and individuals, usually to either hedge risk in another position or to speculate on a future price movement. An international corporation may wish to hedge its exposure to currency exchange rates arising from treasury, revenue or costs and may use a basket option to guarantee a strike price at a future time. A liquidity provider or market maker will typically hedge against price movements in positions resulting from serving a client or market by simultaneously entering into another contract to remain market-neutral.

Contract Pricing A trader may speculate that the price of an asset will be lower in the future by *short selling* the asset. This involves the market participant borrowing the asset to sell at the current market price, before closing the short position at a later date by repurchasing the asset at the new market price to return to the lender. If the price indeed lowered in this time, the trader can make a profit. An *arbitrage* opportunity exists if a market participant is able to construct a portfolio with zero initial value but with a positive probability of positive terminal value and has non-negative terminal value with probability one.[22] An arbitrage opportunity may arise if, for example, an asset such as a currency is traded on different markets at different prices simultaneously, since a market participant could in theory sell short the currency on the market providing the higher price and use the money received from the sale of this borrowed asset to buy the same asset on the second market, earning the risk-free-rate of return on the difference and returning the purchased asset to close the original short position.

This concept of shorting the expensive version and buying (or '*going long*') the cheaper version also underpins the more general *Law of One Price*. This says that two portfolios with equal terminal values must have equal initial values to avoid an arbitrage opportunity. The presence of an arbitrage opportunity, after market frictions are taken into account, allows a market participant to make risk-free profit until the arbitrage is closed, for example by this trading moving prices itself, which comes at the cost to the writer of the contract. Thus it is well-motivated to seek a theoretical understanding of arbitrage-free options pricing.

At the expiry of a European-style options contract, the holder can choose to either exercise or not exercise the contract. If a European call option is exercised, the contract holder pays the strike price K and receives the underlying asset. It is assumed this can be sold immediately at its current market price of S_T. If $K > S_T$, then the contract holder could instead buy the same asset on the open market for less than the strike price and so it is assumed the holder will choose to not exercise the contract, making the minimum possible terminal value $V_T = 0$, excluding

[20]European and American contracts are traded across al major options markets, not only in their namesakes. Exotic options are also named after other countries and continents.

[21]One technical distinction between an index option and a basket option exists: an index option is typically cash-settled without trading directly in the underlying assets, whereas basket options more often involve direct ownership of the underlying asset.

[22]In simpler terms, an arbitrage is the existence of an opportunity to make a guaranteed profit without any initial investment.

initial *premium* paid or opportunity cost. However if $K < S_T$, then the contract holder will be able to make a profit of $V_T = S_T - K$ by exercising the contract. Thus at the terminal time T, the call contract has *intrinsic value*

$$V_T = (S_T - K) \cdot \mathbb{1}\{S_T > K\} = \min(S_T - K, 0). \tag{6}$$

Similarly, a European put option will allow the holder to choose to sell an asset with underlying terminal value S_T at the strike price K, which is profitable to exercise if and only if $K > S_T$. Thus at the terminal time T, the put contract has *intrinsic value*

$$V_T = (K - S_T) \cdot \mathbb{1}\{S_T < K\} = \min(K - S_T, 0). \tag{7}$$

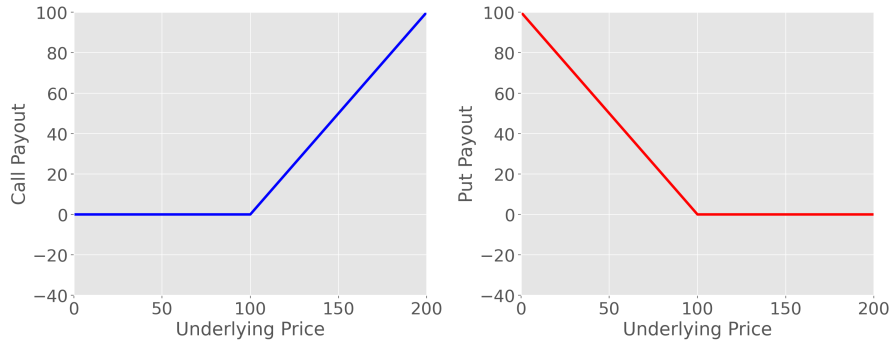


Figure 26: Payoff for $K = 100$ strike European options (excluding premium paid).

Because the option holder gains the right to potentially profit from a future market movement, the writer of the contract receives a *premium* in exchange for the contract, representing the fair value at that time. The value of an option contract at time $0 \leq t \leq T$ can be thought of as the sum of the *intrinsic value* and the *time value*, where the intrinsic value is the value of exercising the contract at the current market price of the underlying asset and the time value is the difference between the premium and the intrinsic value. The time value then represents the value of the optionality provided by the contract at the expense of the contract writer. The intrinsic value is $\min S_t - K, 0$ for a call and $\min K - S_t, 0$ for a put, where S_t is the price of the underlying asset at time t, and the time value represents the value of the opportunity for the intrinsic value to increase by expiry. A contract which has negative intrinsic value is said to be *out of the money (OTM)*, an option with its strike price equal to the current value of the underlying asset is said to be *at of the money (ATM)*, and a contract which has positive intrinsic value is said to be *in the money (ITM)*.

The *risk-free rate* r is an interest rate assumed to be available to all market participants as an alternative guaranteed return on any investment made instead of buying a contract. In financial modelling, uninvested cash reserves are assumed to return the risk-free rate. This return compounds over time so future values are considered according to the *time value of money* which says that the *present value PV* (also known as the *discounted value* of any future value *FV* at time T is related via

$$\mathrm{FV} = (1 + r)^T \mathrm{PV} \tag{8}$$

for discrete (e.g. daily) compounding, or

$$\mathrm{FV} = \exp(rT)\,\mathrm{PV} \tag{9}$$

for continuous compounding.

For a positive risk-free rate, a future payout therefore has less present value than an equal amount held at the current time. Thus combining the Law of One Price, no-arbitrage conditions and the time value of money, the initial value of a contract is the discounted expected terminal payout, where the expectation considers returns in excess of the risk-free rate. This is known as the *Fundamental Theorem of Asset Pricing (FTAP)* [15]. More specifically, the expectation is taken with respect to a *risk-neutral* or *martingale measure*, a unique implied probability measure such that the discounted expectation of an asset value is equal to the current value, known as the *martingale property*. This reflects that information regarding the evolution of a stock price ends at the current time, that is, no information can be known of prices at future dates, since otherwise there would be an arbitrage opportunity. The FTAP also states that a market is arbitrage-free if and only if a risk-neutral measure exists. The proofs of the Fundamental Theorem of Asset Pricing and the uniqueness of the risk-neutral measure are unfortunately not pursuant to the narrative of this thesis. An intuition for these could be provided by considering the premium of a contract as compensation to the writer for the level of risk being taken by issuing this optionality, with discounted values capturing the opportunity cost of investing cash into an options contract, comparing both against a risk-free return available as an alternative.

Theorem 2 (Fundamental Theorem of Asset Pricing I)**.** *A financial market is arbitrage-free if and only if there exists a martingale measure.*

Vanilla option prices with the same underlying asset, strike and expiry must satisfy the *put-call parity* to avoid an arbitrage opportunity on the underlying stock price.

Proposition 4.1 (Put-Call Parity)**.** Denote by C and P the initial fair values (no-arbitrage prices) of European call and put option contracts with strike price K and expiry T on an underlying asset with initial value S that does not pay dividends, and let r be the risk-free rate. Then

$$P + S = C + Ke^{-rT} \text{ or } P + S = C + K\left(1 + r\right)^{-T} \tag{10}$$

for continuous and discrete compounding respectively.

Proof. Consider two portfolios: one portfolio consists of a share and a European put option, and the second portfolio has Ke^{-rT} cash earning the risk-free rate r and a European call option. Both options have the same strike K and maturity T. The two portfolios have equal terminal values

$$\min\left(K - S_T, 0\right) + S_T = \max\left(S_T, K\right) = \min\left(S_T - K, 0\right) + K.$$

By the Law of One Price, the no-arbitrage value of the two portfolios must have been equal at the initial time, thus

$$P + S = C + Ke^{-rT}.$$

The discrete-compounding case is completely analogous with $\left(1 + r\right)^{-T}$ replacing e^{-rT}. □

This provides a *no-arbitrage condition* on the pricing of vanilla options. If an arbitrage opportunity exists, another market participant may continually exploit this source of guaranteed profits at the expense of the option contract seller. In real-world markets, market frictions, liquidity constraints, dividends and other factors also apply.

Motivation There are different participants in financial markets which can broadly be categorised into a *buy-side* and *sell-side*. The buy-side firms generally trade securities for the benefit of themselves, either to hedge risk as in the currency risk to international corporations mentioned earlier, or in proprietary trading seeking to generate a positive profit. An important role from

the sell-side is *market making*, whereby a firm agrees or decides to stand ready to simultaneously offer a buy or sell price (the *bid* and *ask* respectively) to the market. In addition to other market participants writing derivatives contracts, market makers provide liquidity in these markets by actively offering a guaranteed and competitive buy (*bid*) and sell (*ask*) price at all times. It is unlikely a buyer and a seller of a specific security will come to market at the same price at the same time and the fewer active trades that are available, the less *liquid* the market is said to be. For this reason, the market maker provides a service to the market by always providing liquidity so reasonably sized market orders can be filled reliably. The market maker is compensated for this service by collecting the *bid-ask spread* between the sale and purchase price. High competition means this spread is generally very small in liquid markets so the market maker must be very confident of which prices to offer and accept, while providing liquidity in illiquid markets can generate arbitrage opportunities without proper price discovery. It is therefore important for those who make markets in option contracts to be able to be able to accurately and robustly price options and it is valuable for other market participants to be able to identify mispricings to capture arbitrage opportunities.

SDE Pricing Models The cornerstone pricing model is the *Black-Scholes model (B-S)* which is described by a *stochastic differential equation (SDE)*, admitting an analytical solution derived from modelling the underlying asset price as a geometric Brownian motion (so the log of the price can be considered a random walk) and some key assumptions such as constant volatility, which is not found to be true empirically in market data. Unfortunately deep discussion on SDEs is not relevant to the neural-network focus of this thesis. Key assumptions of the standard B-S model are as follows:

- The risk-free rate and volatility are both constant and known exactly;
- No dividends or transaction costs, including for short positions;
- Underlying asset returns follow a stationary log-normal distribution;
- Trading and prices are continuous with infinite liquidity;
- No market participant can predict future price movements.

Although extensions to the model exist which address some of these strict assumptions, none of these assumptions are always empirically true and so it is important to understand this is only an approximation to real-world trading. Hence the formula and indeed all models do not define the true fair value outside the respective model[23].

Theorem 3. *Write N for the standard Normal density function. In the Black-Scholes model, the no-arbitrage price for a European call option with time T to maturity, volatility σ, strike price K, price of the underlying S and risk-free-rate r is given by:*

$$C = N(d_1)S_t - N(d_2)Ke^{-rT},$$
$$\textit{where } d_1 := \frac{\ln S_t/K + \left(r + \sigma^2/2\right)T}{\sigma\sqrt{T}}, \quad d_2 := d_1 - \sigma\sqrt{T},$$

[23]The extensive use of this flawed but easy to apply model in the options market has been widely criticised and has been linked to the irresponsibility which led to the 2007 Financial Crisis. The size of the overall derivatives market traded in 2020 to the end of June was $607 Trillion notional, or $15.5 Trillion net of positive and negative values [42]

In real-world market data, volatility is not constant. The Black-Scholes formula may be inverted to derive an *implied volatility* from an observed option price and if the assumptions of the Black-Scholes model were to always hold, one would expect the resulting implied volatility to be constant. Instead, the implied volatility increases for extreme strike prices, and this is known as the *volatility smile*. Specifically, the implied volatility increases for small maturity values or extreme values of the ratio of the strike price to the current price of the underlying asset, known as the *moneyness*, forming a *volatility surface* as a function of moneyness and time to maturity. Moneyness will be important later as an input feature for the neural network in the research component of this thesis. As the time to maturity tends to infinity, the implied volatility does converge to a constant value.

Alternative models exist which attempt to capture this volatility surface and they must be calibrated to current market conditions. The a popular choice is the *Heston model* [23], characterised by assuming the variance is itself a stochastic process, as well as the asset returns. The Heston model includes its own assumptions, notably that the volatility follows a Cox-Ingersoll-Ross process [10] which includes parameters to describe market phenomenon such as mean reversion, hence the need for calibration to changing market conditions. Assuming mean reversion prevents asset volatilities decaying to zero or exploding to infinity, which are not observed in markets. Moreover, the model is able to recreate the correlated shocks observed between volatility and stock prices. [11]

The SDE given by the Heston model does not admit an analytic solution due to a difficult integral. Moreover, even once evaluated numerically (eg Monte Carlo or finite-differences), the valuation model does not give accurate prices for options close to expiry as it fails to properly capture the high implied volatility. There also exists more advanced models building on the Heston model, for example additionally modelling the risk-free rate as a stochastic process. The process of calibrating the Heston model is not straightforward and so it is desirable[24] to have a model-free (non-parametric) alternative method. Models which do not admit analytic solutions can sometimes be applied using finite-difference or other numerical methods to infer prices but again this can be difficult and computationally expensive. Avoiding calibration and expensive computation is the primary motivation for using neural networks to learn option pricing from market data, in addition to increasing accuracy without strong assumptions imposed by a model.

As the time until maturity decreases, there is less time for an underlying price to change value by a given amount and so an OTM option will decrease in value as the time to expiry decreases since the likelihood of expiring ITM decreases. There are multiple types of *volatility* for an asset but generally speaking, the volatility is usually the standard deviation of some sequence of prices for the asset. An underlying asset with higher volatility will be more likely to have sufficient prive movement (*price action*) to reach a given level, such as an OTM strike price. Consequently, an OTM contract may be considered to have a higher probability of expiring ITM if volatility is high, and so the fair value is higher since more risk is being taken by the contract writer. The dependency of the fair value of an option contract on the strike price, underlying asset price, time to maturity, volatility and risk-free rate indicate quantities which are required to evaluate the fair value of the contract, at least in the simple B-S model. Without a model, a neural network can attempt to learn the mapping from these input features to the price of an options contract.

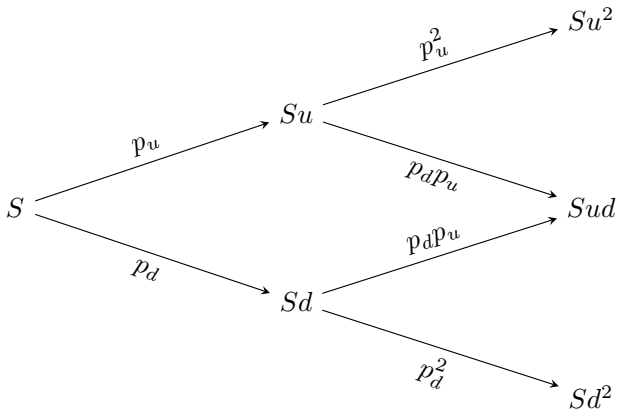
To demonstrate the ability of a neural network to price options contracts, section 5 describes training a neural network to learn an options pricing mapping form historical market data. The mean average percentage error is benchmarked against the closed-form solution arising from the B-S model. Section 6 will report the improvement offered by a trained neural network. Mean

[24] In the early stages of preparation for this thesis, I discussed the topic of option pricing with industry practitioners who consistently expressed a desire for a fast and non-parametric method which did not require considerable effort and resources for calibration.

average percentage error against the Black-Scholes formula is used so that this value can be taken as a threshold accuracy level to which networks are trained to achieve on higher-dimensional basket options with multiple underlying stocks. The minimal time taken to train and price using such a network is compared in section 6 against the incumbent method of Monte Carlo simulation under the Heston model.

4.2 Monte Carlo & Binomial Tree Pricing

Using a model to describe probabilistically the evolution of an underlying asset modelled as a stochastic process, a Monte Carlo numerical pricing method performs many simulations, and then the terminal value of the contract after each simulation is calculated. These are discounted to their present values and averaged to estimate the initial fair value of the contract. An important introductory method for option pricing theory is the *binomial tree model* and provides a useful analogy to model how the Monte Carlo method is able to estimate a fair price by estimating the probabilities of each outcome at the terminal time to allow an expectation to be calculated.



The *binomial tree model* of option pricing models the price evolution of a stock price in discrete time steps according to the following algorithm:

1. At time $t = 0$, the stock has an initial price $S_0 = S$.
2. At time $t = 1$, the stock can either go up by a factor of u to $S_1 = uS$ with probability p_u, or go down by a factor of d to $S_1 = dS$ with probability p_d, such that $p_u + p_d = 1$.
3. From each possible price at time $t = 1, \ldots, T-1$, the price can either go up or down by a factor of u or d with probabilities p_u or p_d respectively, as before.
4. This forms a tree graph with nodes presenting each possible price at each time from $t = 0$ to expiry at time $t = T$.
5. The payout of an option contract is evaluated on each of the time $t = T$ node prices.
6. Iterating backwards from time $t = T - 1$ to time $t = 0$, for each node at time t, evaluate the expected discounted value of the incident pair of nodes in layer $t + 1$.
7. The value at the time $t = 0$ node is the fair price of the options contract at the initial time.

Recall that the risk-neutral measure is used to evaluate an expected value and this is defined such that the expected discount stock price is equal to the current stock price. Thus the risk-neutral measure $\{p_u, p_d\}$ is such that it satisfies

$$S_t = \frac{1}{1+r}\left(q_u u S_t + q_d d S_t\right), \tag{11}$$

since $S_{t+1} = uS_t$ with probability q_u or $S_{t+1} = dS_t$ with probability $q_d = 1 - q_u$. Thus the risk-neural measure is uniquely determined to be

$$q_u = \frac{1+r-d}{u-d}, \quad q_d = 1 - q_u = \frac{u-(1+r)}{u-d}.$$

In more generality, u_t, d_t can easily depend on the time t, in which case the risk-neutral probabilities will in general be different at different nodes of the tree. For simplicity, the below assumes $u_t \cong u,\ d_t \cong d$ and so $q_u,\ q_d$ will also be independent of all times and nodes. Now this process may be iterated backwards along the tree, using the risk-neutral measure to calculate fair prices for the contract at each node, in particular the first node representing the fair price at the initial time.

This understanding of how a binomial model is able to estimate the fair value of option contracts will help us later understand how the Monte Carlo simulations estimate this price. The binomial model is a powerful technique in that it can immediately generalise to exotic path-dependent options with no additional work. For example, the payout of an *Asian option* depends on the average price of the underlying security in the time before expiration. In particular, the payout depends on the entire price history or path from time $t = 0$ to expiry at time T. In this situation, the payout at each node is evaluated according to the prices on each node of the path taken from the node at time $t = 0$. Note the simplification to $u_t \equiv u,\ d_t \equiv d$ means there will be $\binom{t}{a}$ paths leading to the node with stock price $u^a d^{t-a} S$. The computations for this case are not much more difficult but not beneficial for an understanding of the benchmark techniques used in this research.

The exponential growth in the number of nodes as time increases helps to understand why Monte Carlo pricing is very inefficient. The Law of Large Numbers allows for the expectation value of a random process to be estimated as an average of a large number of trials. To generate each step of a simulated price evolution, a value must be sampled from a probability distribution for each stochastic process at each time step, and to model a basket of underlying return processes or additional stochastic processes, correlations between each process must also be accounted for. Even in the simplest form, this is many random samples. However, the many possible values after a large time horizon means a given simulation becomes less representative of the expected value. Thus more simulation runs are required to converge the average to the expected value being estimated. For a visualisation of many Monte Carlo stock price evolution paths, see figure 27. Variance reduction techniques exist, such as importance sampling, stratified sampling, conditioning, control variates and bootstrap, however these can add additional computation and do not overcome the breakdown of convergence in the limit. Even for small simulations, section 6 demonstrates this method is much slower than a neural network, and scales linearly with the number of underlying assets. For an index option representing possibly hundreds of underlying stocks, this is too slow for all but the longest-maturity contracts.

The main takeaway from section 4.1 should be that there is significant need for a fast, accurate and possibly model-free method of pricing derivative contracts, in particular basket options, which does not require regular calibration. A neural network is able to easily capture the complexity of a price mapping and generalise well to different market conditions and therefore presents a more scalable alternative to existing methods.

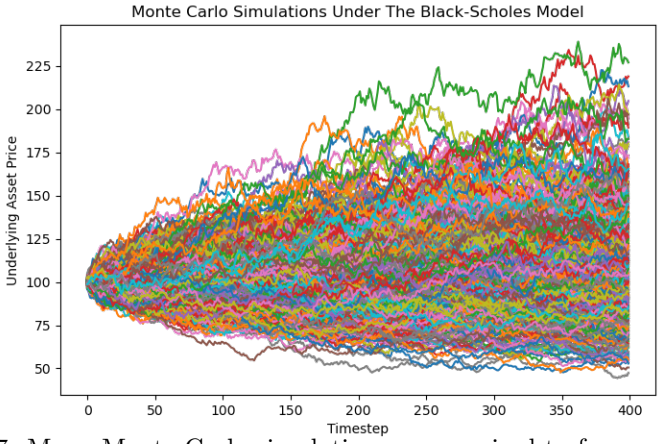


Figure 27: Many Monte Carlo simulations are required to for convergence.

5 Methodology

The motivation for the exploration of using neural networks to price options contract comes principally from improved efficiency and the absence of model-calibration requirements. This thesis presents a demonstration and evaluation of the potential for improvements in efficiency over an incumbent method. This is measured by averaged the wall-clock time taken to both train a network learning on (simulated) options quotes to a threshold error level, combined with the averaged inference time to estimate prices on a large number of contracts. The training data and benchmark times are provided by a Monte Carlo simulation pricer under the Heston model. The threshold error is chosen by evaluating the mean absolute percentage error on historical quotes for single-stock call options.

To justify the use of neural networks for this problem, it must first be established that the method can be at least as accurate as standard alternatives, before being able to demonstrate they are faster to at this level of accuracy. Hence, the ability of a trained neural network to outperform the accuracy of the Black-Scholes theoretical model, a standard benchmark in both industry and academia, is first demonstrated on historical options quotes. Once this is established, exploration of the efficiency of similar neural networks against the incumbent method of Monte Carlo pricing with the Heston model is justified. Many design choices have been made with the expressed intention of favouring the benchmark methods. Despite this, the neural networks used in this research vastly outperform these benchmark methods for both time efficiency and accuracy to observed market prices. Several key such choices are as follows: using a CPU instead of a GPU despite natural acceleration of neural network computations in parallel; the removal of extreme values from the evaluation test set for the Black-Scholes accuracy benchmark; near-constant number of repetitions of Monte Carlo simulations for all contracts despite convergence issues and worsening training data quality for the neural networks; use of a simplified correlation structure and exclusion of Monte Carlo variance reduction techniques to eliminate overhead calculations; contracts for multiple maturities were simulated on a single simulation simultaneously, as well as simultaneous pricing of different strikes at each maturity; and inclusion of the one-off training time of the neural networks, in addition to the inference time required to price contracts.

As a basket size increases, the number of Monte Carlo evolutions which need to be simulated for a given basket simulation increases linearly and so if convergence considerations are ignored, one might expect the time taken to price a basket of stocks to grow linearly with the number of underlyings in the basket. Conversely, the complexity and vectorisation embedded

within neural network architecture could allow for a network to accommodate an increasingly high-dimensional mapping with only small adjustments to the number of nodes or choices of hyperparameters. This could lead to a lesser increase in training and inference times for a larger input dimension compared to Monte Carlo. It is noteworthy that once a network is trained, it may be reused to price a new option contract in different market conditions, whereas the Monte Carlo method requires new evaluations to price each new contract considered, and further calibration for changing market conditions, and so including the training time in the efficiency comparison generously favours the benchmark, while also providing interesting insight into how the separate training and inference steps scale differently. Although Python is not a naturally fast programming language, to remain consistent with the wider project group and for simplicity, both the Monte Carlo simulations and the neural networks were written in Python 3. The TensorFlow 2.3 library developed by Google [34] was used to implement the neural networks and hyperparameter tuning was aided by TensorBoard.

Once accuracy is established, the potential efficiency benefits will be demonstrated using as Monte Carlo simulations both to generate synthetic call option price data and for providing benchmark pricing times. The Monte Carlo algorithm employed simulates underlying return evolutions under the Heston model to generate prices of basket options defined by contract and market parameters sampled randomly, where possible from the historical market data collected during the accuracy stage. The time taken for the Monte Carlo simulations to estimate these prices is logged and the generated data are used to train a neural network for each basket size to learn to predict these simulated contract prices in place of real-world historical prices. The neural networks are trained until a MAPE error threshold is reached, chosen to significantly improve upon the MAPE error achieved by Black-Scholes on the real-world market data. Specifically, the B-S pricer achieves a MAPE value of 98%, while the basket option pricing neural networks were trained to a MAPE value of just 1%. The networks trained on simulated data were tuned with hyperparameters able to achieve MAPE values of 0.38-0.59% given sufficient training, but for timing purposes, training was stopped once epoch validation losses decreased below the threshold value of 1%. The average time over 20 repetitions to achieve the threshold error level is recorded and the time required to evaluate the network's predictions on all synthetic prices generated by the Heston model Monte Carlo simulations, scaled up to 1 million sampled, averaged over 20 repetitions is added.

5.1 Neural Network Design

Fully connected networks were used throughout with uniform hidden layer widths. The intention of the research component of this thesis is to study the scale of potential benefits of neural networks over conventional options pricing techniques and so using a standard fully-connected network provides a baseline level of performance which one might reasonably expect to be approved upon with more complex architectures, in particular for pricing accuracy. The neural network used to learn historical prices accept the 5 input features from the Black-Scholes model with a single price output value, 3 hidden layers, 14 fully-connected nodes per hidden layer, an ELU activation layer and a minibatch size of 2^8. The hyperparameters used in training to 3 s.f. are listed in the following table.

Initial LR	LR Decay Rate	Dropout Prob.	L1-Reg. Coeff.	L2-Reg. Coeff.
0.00316	N/A	N/A	3.98×10^{-9}	3.98×10^{-8}

Because a fully-connected neural network accepts only a fixed input feature vector size, a

separate network is used for each basket size. This analysis considers baskets of size 1, 4, 7, 10, 13 and 16, which correspond to 7 + 2d input features for a basket of d underlyings since each basket option is described by a d-vector of spot prices, a d-vector of correlations between each random underlying return process and its associated random volatility process, as well as 7 other market statistics, namely: time to maturity, initial variance, long-term average price variance, mean reversion rate of variance, volatility of variance, the risk-free rate. The input features for the accuracy and efficiency tests reflect the inputs required by the Black-Scholes and Heston models used as benchmarks in the respective tests. In more generality a Heston model simulation requires a full correlation matrix between each random underlying return process, however this is simplified in our implementation as this reduces computation time in generating each simulation and thereby making it a more difficult benchmark to beat.

The neural network architectures used for the multidimensional basket options had depths of 3-6 hidden layers, 5-19 nodes per hidden layer, and all used the ELU activation function and Adam optimiser, using batch renormalisation and no dropout layers. Complete tables of hyperparameters used in each neural network are listed in the table at the end of this section. As described above, training data were generated via a Monte Carlo simulation method for each basket size (400,000-500,000 training examples) and each network was trained until a threshold MAPE of 1% was achieved.

Hyperparameter Tuning For the optimisation of the networks used in this thesis, first sensible values were considered and loss plots examined to diagnose under or over fitting, which in turn informed new choices of hyperparameters to trial in an iterative process relying on an understanding of the impact of each hyperparameter on the network and building this intuition by studying the results of each trial. Once such a range of sensible values is established in this way, a random search is performed to trial the performance of multiple networks configured with different collections of hyperparameters. TensorBoard is then a powerful tool to help visualise patterns in the outcomes of these trials to narrow the hyperparameter space to search to identify the optimal configuration. At first, direct relationships between error and a single hyperparameter may be visible and this will quickly narrow the range to continue searching, however once these observations are exhausted, patterns in loss values arising from pairings or combinations of hyperparameters may be identified with sufficient samples and insight. Once the hyperparameter space is shrunk, further random searches can be performed on the subspace to better sample the region of better minimised loss values. Eventually relationships between hyperparameters and their effect on the loss become more imperceptible and a final intelligent Bayesian search is performed in this small subspace of possible hyperparameters to search, if the number of hyperparameters is less than 15-20. In the networks constructed in this thesis, there were 5 hyperparameters (excluding network depth and layer width) and so it was possible to apply Bayesian optimisation to identify a configuration of hyperparameters which are very well optimised to minimise the validation loss. This process is illustrated in example screenshots saved in the GitHub repository `https://github.com/Connor-Tracy/Neural-Network-Option_pricing` for this project, taken directly from the development of the networks used in this thesis.

Basket Size	Features	No. Layers	Width	LR	Decay Rate	L1-Reg.	L2-Reg	Min MAPE(%)
1	9	4	5	0.00205	0.926	1.82×10^{-7}	2.22×10^{-6}	0.59
4	15	5	11	0.00334	1.0	1.27×10^{-8}	1.00×10^{-7}	0.46
7	21	3	8	0.00841	0.937	1.14×10^{-7}	8.85×10^{-8}	0.42
10	27	4	8	0.00545	0.909	1.44×10^{-7}	1.81×10^{-7}	0.54
13	33	5	19	0.00383	0.822	4.43×10^{-8}	2.56×10^{-8}	0.53
16	39	6	19	0.00477	0.900	4.28×10^{-7}	9.279×10^{-7}	0.38

5.2 Data: Sourcing, Pre-processing & Cleaning

Historical data of traded call option quotes was sourced form OptionMetrics using a WRDS subscription, a leading database supplier for market data. This very useful service has vast databases of historical financial data, however many data cleaning and pre-processing steps in a surprisingly laborious process[25] were required. Briefly, large database tables were queried using SQL within a WRDS API and the results of these queries were combined into a Pandas dataframe. Interest rate data was available on a daily frequency but not for all required time horizons and so cubic splicing was used to interpolate the rates to obtain interest rate values corresponding to all maturities and dates present in the data. It was necessary to account for more database conventions (many poorly or not documented) than it would be worthwhile to describe here but the simplest ones were dividing by a multiplier used for strike prices and accounting for standard option contracts being for 100 shares in the underlying so the data can be compared with the Black-Scholes benchmark which assumes a single share is traded.

OptionMetrics provides historical realised volatility as well as risk-free rate estimates from zero-coupon LIBOR interest rates. The training data included interest rates matching the maturity of the contract interpolated via cubic spline, and quotes traded on days on which there were insufficient interest rate values (4 are required) available to perform cubic spline were discarded. Volatility was taken to be the 30-day realised value calculated as the standard deviation of closing mid-price log-returns. It was hoped that put option quotes could be transformed into call option quotes from the historical data, but despite excluding dividend paying stocks, the historical data did not satisfy the naive put-call parity closely enough for this to be useful for increasing the training set size.

Data cleaning was also important. For example, extreme outliers from periods of market stress lead to Black-Scholes estimates with error orders of magnitude larger than the typical error so these were removed to improve the benchmark performance. Database conventions lead to various types of duplicate data entries being present so these were removed to prevent data leakage artificially lowering the test loss of the neural network. Contract and underlying prices near tick-size violate the continuity assumption required for Black-Scholes pricing and so affected quotes were also removed.

Simulated data of basket call option prices was generated using Monte Carlo simulations estimated the price of random contracts under the Heston stochastic volatility model. Input

[25]The code for sourcing, processing and cleaning the OptionMetrics data is comparable in length and time investment to all other coding tasks in this project.

quantities defining each contract were sampled from the real historical data where possible (eg spot prices and interest rates). Informed by [8], realistic ranges were chosen from which to sample the remaining quantities. In training, I used random splitting into training and validation sets in accordance with the recommendation by the literature review [43].

The *moneyness* of an option contract is the ratio of the spot and strike price of contract. This is a useful quantity because many models assume the underlying return process distribution to be independent of the spot price, thus only the stationary moneyness value is required for option pricing in these models. This provides a small advantage to a neural network since removing an input feature reduces the dimension of the target mapping. As such this is a standard reparameterisation of the problem and it has been widely observed [17] to improve network generalisation, as well as being expressly recommended in option-pricing neural network design in the literature review [43].

In the design of these demonstrations, there are in many places in which either the benchmark or the network times and accuracies could be easily influenced almost at will. For example, Monte Carlo pricing requires the averaging of many individual simulated stock evolutions. The more repetitions that are performed, the prices will be more converged/meaningful/accurate but the time taken will scale linearly. Moreover, prices which are less converged will be less representative of the underlying target distribution for the neural network training and so this will hinder its accuracy, making it take longer to reach the threshold accuracy or unable to learn such a noisy poorly sampled distribution. As the dimension of a basket option increases, the variance in the simulation increases so the number of repetitions must increase to maintain a comparable level of convergence. To give the best chance to the benchmark and make things as difficult as possible for the proposed neural network method, almost no additional repetitions will be required for higher basket dimensions and a very minimal number of repetitions will be used. Despite this, the neural network is able to learn these prices very accurately and much faster than the Monte Carlo method. Further design choices following this philosophy include excluding quotes reflecting times of extreme market stress during which Black-Scholes scores an error orders of magnitude larger than normal, applying a significant time-saving optimisation to price contracts with a range of maturities and strikes simultaneously with a single set of Monte Carlo simulations, not applying variance reduction techniques, using a simplified correlation structure and not running large simulations which would improve convergence at the cost of time.

6 Results

Accuracy: Black-Scholes Benchmark On historical US-listed call option quotes from 2002-2020, the standard Black-Scholes formula achieved a mean squared error (MSE) of 0.1496 and a mean absolute percentage error (MAPE) of 98.72%. Notably, this is after extreme outliers and periods of heightened market stress were removed, when the observed error is several orders of magnitude larger than the typical error. Nevertheless, this is significantly larger than the MSE of 0.0089 and MAPE of 33.50% achieved by a small fully-connected neural network on an unseen validation data set, averaged over the last three training epochs. The training of the network took approximately 60.9 seconds and optimised for minimising MAPE.

	MSE	MAPE (%)
Black-Scholes	0.1496	98.72
Neural Network	0.0089	33.50

Such significant outperformance of a simple neural network over a widely used standard benchmark establishes neural networks as a credible alternative to existing methods, thereby providing justification for pursuing an analysis of how much more efficiently

Efficiency: Monte Carlo Heston Benchmark Table displaying the times for each method to do each pricing task and describing qualitatively that neural network learning is faster than Monte Carlo in all cases but much faster for high-dimensional options. Find out whether the model-free network was better than the models and which inputs worked best.

Neural network scaled exceptionally well (almost flat) with an increase of input dimension since even small networks had sufficient complexity to capture the pricing formula even for higher dimensions. Conversely, the Monte Carlo pricing method scaled linearly with input dimension under very generous design considerations chosen to improve the efficiency of the simulations. If convergence had been considered, the time taken for the Monte Carlo simulations would have grown faster than linearly. Interestingly, even for a single underlying and including the one-off training time of the neural network, the Monte Carlo method was 19.4 times slower than the neural network, with this multiple increasing to 74.2 times for a basket with 16 underlyings.

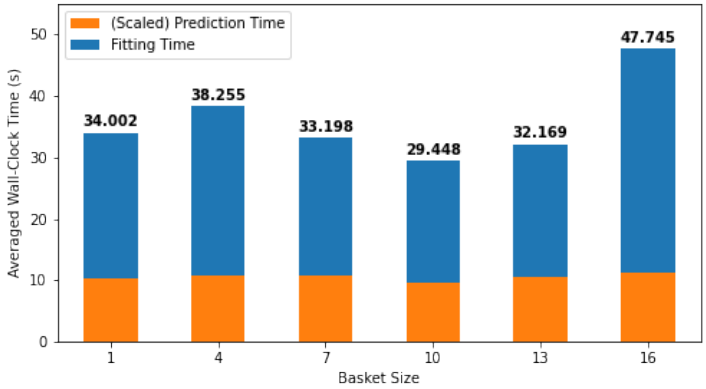


Figure 28: Neural network prediction and one-off training time scaled very well with basket size.

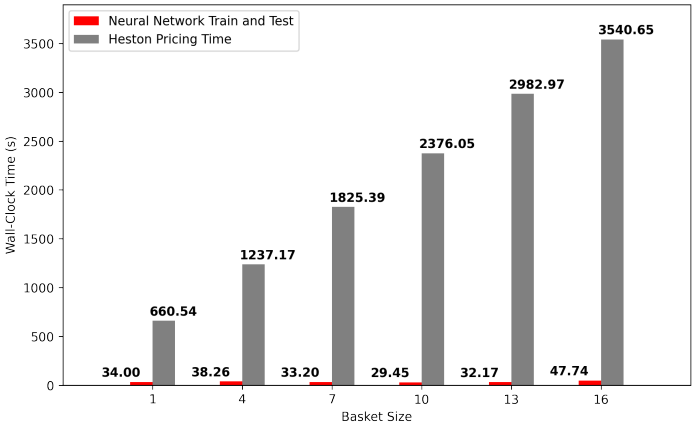


Figure 29: Neural network and Monte Carlo pricing times for increasing basket size.

7 Conclusion

This thesis has presented a pedagogical introduction to deep neural networks aimed at the audience of a typical 4H student, before applying these techniques to explore an active area of research in options pricing, with results indicating significant potential for this technique to address existing issues with incumbent methods. Every neural network used in this research vastly outperformed its benchmark, especially in regards to efficiency, despite very significant design choices to favour the Black-Scholes SDE model and Heston model Monte Carlo benchmarks for accuracy and efficiency respectively. Inference time was almost constant across networks for all basket sizes due to similarly-sized networks accommodating the more complex mappings. In a practical context, a trained method may be continually reused for inference with prediction time almost independent of the basket size for a high-dimensional option, presenting an even more significant improvement over Monte Carlo methods which scale at least linearly with basket size.

References

[1] Yoshua Bengio. "Practical recommendations for gradient-based training of deep architectures". In: *Neural networks: Tricks of the trade*. Springer, 2012, pp. 437–478.

[2] Yoshua Bengio, Nicolas Boulanger-Lewandowski, and Razvan Pascanu. "Advances in optimizing recurrent networks". In: *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*. IEEE. 2013, pp. 8624–8628.

[3] James Bergstra and Yoshua Bengio. "Random search for hyper-parameter optimization." In: *Journal of machine learning research* 13.2 (2012).

[4] Christopher M Bishop. *Pattern recognition and machine learning*. springer, 2006.

[5] Mark JM Bishop. "History and philosophy of neural networks". In: (2015).

[6] Ewen Callaway. "'It will change everything': DeepMind's AI makes gigantic leap in solving protein structures." In: *Nature* (2020).

[7] Marc Chataigner, Stéphane Crépey, and Matthew Dixon. "Deep Local Volatility". In: *Risks* 8.3 (2020), p. 82.

[8] Pavel Cízek, Wolfgang Härdle, and Rafal Weron. *Statistical Tools for Finance and Insurance: 7.4 Calibration*. Mar. 3, 2005. URL: http://sfb649.wiwi.hu-berlin.de/fedc_homepage/xplore/tutorials/stfhtmlnode48.html (visited on 02/01/2021).

[9] Djork-Arné Clevert, Thomas Unterthiner, and Sepp Hochreiter. "Fast and accurate deep network learning by exponential linear units (elus)". In: *arXiv preprint arXiv:1511.07289* (2015).

[10] John C Cox, Jonathan E Ingersoll Jr, and Stephen A Ross. "A theory of the term structure of interest rates". In: *Theory of valuation*. World Scientific, 2005, pp. 129–164.

[11] Ricardo Crisóstomo. "An Analyisis of the Heston Stochastic Volatility Model: Implementation and Calibration Using Matlab". In: (2014).

[12] CS231n. *Convolutional Neural Networks for Visual Recognition*. 2021.

[13] George Cybenko. "Mathematics of control". In: *Signals and Systems* 2 (1989), p. 303.

[14] Yann Dauphin et al. "Identifying and attacking the saddle point problem in high-dimensional non-convex optimization". In: *arXiv preprint arXiv:1406.2572* (2014).

[15] Freddy Delbaen and Walter Schachermayer. "A general version of the fundamental theorem of asset pricing". In: *Mathematische annalen* 300.1 (1994), pp. 463–520.

[16] Peter I Frazier. "A tutorial on Bayesian optimization". In: *arXiv preprint arXiv:1807.02811* (2018).

[17] Eric Ghysels et al. *Nonparametric methods and option pricing*. CIRANO, 1997.

[18] Xavier Glorot and Yoshua Bengio. "Understanding the difficulty of training deep feedforward neural networks". In: *Proceedings of the thirteenth international conference on artificial intelligence and statistics*. JMLR Workshop and Conference Proceedings. 2010, pp. 249–256.

[19] Ian J Goodfellow et al. "Generative adversarial networks". In: *arXiv preprint arXiv:1406.2661* (2014).

[20] Ulf Grenander. "On empirical spectral analysis of stochastic processes". In: *Arkiv för Matematik* 1.6 (1952), pp. 503–531.

[21] Isabelle Guyon et al. "A scaling law for the validation-set training-set size ratio". In: *AT&T Bell Laboratories* 1.11 (1997).

[22] Kaiming He et al. "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification". In: *Proceedings of the IEEE international conference on computer vision*. 2015, pp. 1026–1034.

[23] Steven L Heston. "A closed-form solution for options with stochastic volatility with applications to bond and currency options". In: *The review of financial studies* 6.2 (1993), pp. 327–343.

[24] Elad Hoffer, Itay Hubara, and Daniel Soudry. "Train longer, generalize better: closing the generalization gap in large batch training of neural networks". In: *arXiv preprint arXiv:1705.08741* (2017).

[25] Sergey Ioffe. "Batch renormalization: Towards reducing minibatch dependence in batch-normalized models". In: *arXiv preprint arXiv:1702.03275* (2017).

[26] Sergey Ioffe and Christian Szegedy. "Batch normalization: Accelerating deep network training by reducing internal covariate shift". In: *International conference on machine learning*. PMLR. 2015, pp. 448–456.

[27] Nitish Shirish Keskar et al. "On large-batch training for deep learning: Generalization gap and sharp minima". In: *arXiv preprint arXiv:1609.04836* (2016).

[28] Walter Julius Michael Kickert. *Decision making about decision making: Metamodels and metasystems*. Vol. 7. Routledge, 1987.

[29] Diederik P Kingma and Jimmy Ba. "Adam: A method for stochastic optimization". In: *arXiv preprint arXiv:1412.6980* (2014).

[30] Brent Komer, James Bergstra, and Chris Eliasmith. "Hyperopt-sklearn: automatic hyperparameter configuration for scikit-learn". In: *ICML workshop on AutoML*. Vol. 9. Citeseer. 2014, p. 50.

[31] Xiang Li et al. "Understanding the disharmony between dropout and batch normalization by variance shift". In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2019, pp. 2682–2690.

[32] Shuaiqiang Liu et al. "A neural network-based framework for financial model calibration". In: *Journal of Mathematics in Industry* 9.1 (2019), pp. 1–28.

[33] Andrew L Maas, Awni Y Hannun, and Andrew Y Ng. "Rectifier nonlinearities improve neural network acoustic models". In: *Proc. icml*. Vol. 30. 1. Citeseer. 2013, p. 3.

[34] Martın Abadi et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015. URL: `http://tensorflow.org/`.

[35] Panayotis Mertikopoulos et al. "On the almost sure convergence of stochastic gradient descent in non-convex problems". In: *arXiv preprint arXiv:2006.11144* (2020).

[36] Marvin Minsky and Seymour Papert. "Perceptron: an introduction to computational geometry". In: *The MIT Press, Cambridge, expanded edition* 19.88 (1969), p. 2.

[37] Vinod Nair and Geoffrey E Hinton. "Rectified linear units improve restricted boltzmann machines". In: *Icml*. 2010.

[38] Preetum Nakkiran et al. "Deep double descent: Where bigger models and more data hurt". In: *arXiv preprint arXiv:1912.02292* (2019).

[39] Yurii E Nesterov. "A method for solving the convex programming problem with convergence rate O $(1/k^2)$". In: *Dokl. akad. nauk Sssr*. Vol. 269. 1983, pp. 543–547.

[40] Andrew Ng. *Improving Deep Neural Networks: Hyperparameter Tuning, Regularization and Optimization (Course 2) in CS230*. 2017.

[41] Andrew Ng. *Neural Networks and Deep Learning (Course 1) in CS230*. 2017.

[42] *OTC derivatives statistics at end-June 2020*. Nov. 9, 2020. URL: `https://www.bis.org/publ/otc_hy2011.htm%5C#:~:text=Notional%5C%20amounts%5C%20of%5C%20all%5C%20OTC,%5C%2C%5C%20right%5C%2Dhand%5C%20panel` (visited on 02/01/2021).

[43] Johannes Ruf and Weiguan Wang. "Neural networks for option pricing and hedging: a literature review". In: *Journal of Computational Finance, Forthcoming* (2020).

[44] Robin M Schmidt, Frank Schneider, and Philipp Hennig. "Descending through a Crowded Valley–Benchmarking Deep Learning Optimizers". In: *arXiv preprint arXiv:2007.01547* (2020).

[45] Andrew W Senior et al. "Improved protein structure prediction using potentials from deep learning". In: *Nature* 577.7792 (2020), pp. 706–710.

[46] David Silver et al. "Driessche, G. vd,... Hassabis, TGD (2016). Mastering the game of Go with deep neural networks and tree search". In: *Nature* 529 (), pp. 484–489.

[47] Leslie N Smith. "Cyclical learning rates for training neural networks". In: *2017 IEEE winter conference on applications of computer vision (WACV)*. IEEE. 2017, pp. 464–472.

[48] Nitish Srivastava et al. "Dropout: a simple way to prevent neural networks from overfitting". In: *The journal of machine learning research* 15.1 (2014), pp. 1929–1958.

[49] Nitish Srivastava et al. "Dropout: a simple way to prevent neural networks from overfitting". In: *The journal of machine learning research* 15.1 (2014), pp. 1929–1958.

[50] Ilya Sutskever et al. "On the importance of initialization and momentum in deep learning". In: *International conference on machine learning*. PMLR. 2013, pp. 1139–1147.

[51] Paulo Tabuada and Bahman Gharesifard. "Universal approximation power of deep neural networks via nonlinear control theory". In: *arXiv preprint arXiv:2007.06007* (2020).

[52] Andrey Nikolayevich Tikhonov. "On the stability of inverse problems". In: *Dokl. Akad. Nauk SSSR*. Vol. 39. 1943, pp. 195–198.

[53] Zitong Yang et al. "Rethinking bias-variance trade-off for generalization of neural networks". In: *International Conference on Machine Learning*. PMLR. 2020, pp. 10767–10777.

[54] Tong Yu and Hong Zhu. "Hyper-parameter optimization: A review of algorithms and applications". In: *arXiv preprint arXiv:2003.05689* (2020).

[55] Chiyuan Zhang et al. "Understanding deep learning requires rethinking generalization". In: *arXiv preprint arXiv:1611.03530* (2016).

[56] Hui Zou and Trevor Hastie. "Regularization and variable selection via the elastic net". In: *Journal of the royal statistical society: series B (statistical methodology)* 67.2 (2005), pp. 301–320.