Solving High-Dimensional Nonlinear Partial Differential Equations Using Deep Learning

by

Tara Aghajani (CID: 01072173)

Department of Mathematics
Imperial College London
London SW7 2AZ
United Kingdom

Thesis submitted as part of the requirements for the award of the MSc in Mathematics and Finance, Imperial College London, 2018-2019

IMPERIAL COLLEGE LONDON

THESIS DECLARATION FORM

(for candidates whose degree will be awarded by **Imperial College**)

Declaration: I hereby confirm that the work contained in this thesis is my own work unless other wises stated.

Name Tara Aghajani **CID** 01072173

Title of Thesis Solving High-Dimensional Nonlinear Partial Differential Equations Using Deep Learning

Month and year of submission for examination 09/2019

Date 09/09/2019

Signature of Candidate Aghajani

Acknowledgements

I would like to thank my supervisor, Dr Antoine Jacquier, for his support throughout this project, but also throughout the year as a great lecturer.

Contents

[T1]fontenc upquote

1 Introduction

This paper considers the deep learning-based approach for approximation of the solution of high-dimension nonlinear parabolic PDEs suggested in [1] and [2]. In these papers, PDEs are reformulated using backward stochastic differential equations and the gradient of the unknown solution is approximated by neural networks, with the gradient acting as the policy function. Numerical simulations are performed for the approximation of the solution of nonlinear Black-Scholes PDEs, whose nonlinearity is a result of more realistic market assumptions.

Unlike classical numerical approaches such as finite difference methods, this method does not run into difficulties involving computational cost and stability, especially in high dimensions. The main idea behind the finite difference method is to employ a Taylor series expansion to replace the partial derivatives in the PDE by difference quotients. The interior of the region where the PDE is defined is replaced by a finite mesh grid of points and the partial derivatives in the PDE are approximated at each grid point. This grid grows exponentially in the number of dimensions of the PDE, which makes this method computationally heavy, even for $d = 4$. This is the so called 'curse of dimensionality', expression coined by Richard E. Bellman when considering problems in dynamic programming [3].

Other methods have recently been developed to attempt to solve high-dimensional nonlinear PDEs. One of these methods consists in using multilevel decomposition of Picard iteration [4]. I will compare the efficiency of this method with the efficiency of the neural network algorithm used in this paper. Another method, which also uses deep neural networks is the so called "Deep Galerkin Method (DGM)", which was developed by Sirignano and Spiliopoulos in [5] and further developed in [6]. This method consists in training the neural network using randomly sampled time and space points, which makes the algorithm meshfree unlike commonly used numerical methods such as finite difference methods.

2 Introduction to Partial Differential Equations

2.1 General Overview

A partial differential equation (PDE) is an equation involving unknown multivariable functions and their partial derivatives. PDEs are used to mathematically formulate many phenomena from the natural sciences (electromagnetism, Maxwell's equations) to the social sciences (financial markets, Black-Scholes model).

We will define the PDE for the unknown function $u : \mathbb{R}^n \to \mathbb{R}$. This is the function we wish to solve for.

A k-th order partial differential equation is an equation of the form :

$$f\left(x, u(x), Du(x), \ldots, D^{k-1}u(x), D^k u(x)\right) = 0, \qquad x \in \mathbb{R}^n,$$

where D^k is the collection of all partial derivatives of order k and x is the independent variable. The order of the PDE is the order of the highest order derivative that appears in the equation. If f is a linear function of u and its derivatives, then the PDE is called linear, i.e. the coefficients depend at most on the independent variable. If the coefficients of the terms involving the highest order derivative of u depend only on the independent variable, not on u or its derivatives, the PDE is called semilinear. If f is linear in the highest order derivative with coefficients that depend on lower order derivatives the PDE is called quasilinear, i.e. the coefficients of the highest-order terms may depend on $x, u, Du, \ldots, D^{k-1}u$, but not on $D^k u$.

2.2 The Black-Scholes Partial Differential Equation

The Black-Scholes equation, perhaps the most well-known equation in quantitative finance, and the associated Black-Scholes PDE were developed by three economists: Fischer Black, Myron Scholes and Robert Merton. They were introduced in their paper "The Pricing of Options and Corporate Liabilities," published in the Journal of Political Economy in 1973 [7].

They are used for the pricing of financial derivatives, such as European-style contingent claims. A contingent claim is a financial instrument whose payoff depends on the realisation of some uncertain future event. For European derivatives, this payoff depends on the level of an agreed-upon underlying financial asset or a set of assets at a pre-determined maturity date. Common underlying instruments include stocks, bonds, currencies and commodities.

The Black-Scholes model makes the following set of assumptions:

- Markets are arbitrage-free.
- Markets are friction-less (i.e. no transaction costs, no taxes, etc.).
- No dividends are paid out during the life of the option.
- Any fraction of a share can be bought or sold and it is possible to hold a negative number of shares (i.e. short-selling).
- The drift and the volatility of the underlying are known and constant.
- The risk-free rate is constant.

To price an option, we assume that the price dynamics for the underlying asset are given by the Black-Scholes market model. Let $X = (X_t)_{t \geq 0}$ be the price process of the risky asset. In the Black-Scholes model, this process follows a geometric Brownian motion (GBM) with constant drift and volatility. Consider a given probability space $(\Omega, \mathcal{F}, \mathbb{P})$ supporting a Brownian motion $(W_t)_{t\geq 0}$. Under the 'real world' probability measure $\mathbb{P}$, the price process and the bank account process are given by:

$$\begin{aligned} \tfrac{dX_t}{X_t} &= \mu dt + \sigma dW_t \\ \tfrac{dB_t}{B_t} &= r dt. \end{aligned} \tag{2.1}$$

Here, the drift μ represents the expected return on the asset and σ is the variance of the returns on the asset.

We want to price a claim written on the asset X. Assume that the claim's price is given by $u(t,x)$, which gives the value of the claim at time t when the underlying asset is at the level $X_t = x$. Assume this function is sufficiently smooth. Define the payoff function by $g(x)$, i.e. $u(T,x) = g(x)$, where T is the expiration date. By a dynamic hedging and no-arbitrage argument, it can be shown that u must satisfy the Black-Scholes PDE:

$$\begin{cases} \partial_t u(t,x) + rx \cdot \partial_x u(t,x) + \frac{1}{2}\sigma^2 x^2 \cdot \partial_{xx} u(t,x) = r \cdot u(t,x) \\ u(T,x) = g(x). \end{cases} \tag{2.2}$$

Now that the Black-Scholes PDE is derived, it can be solved for an exact formula in the case of a European option and for certain other simple derivatives. In other cases, one can use the Feynman-Kac formula and Monte Carlo methods to approximate the solution, as long as the PDE is linear.

3 Introduction to Deep Learning

Conventional programming involves precisely defining tasks for the computer to perform. By contrast, in the field of machine learning (ML) computer algorithms learn by experience and acquire skills without human involvement. Deep learning is a subset of ML where artificial neural networks (ANN) learn from large amounts of data. ANN is inspired by biological neural networks that make up the human brain. Similarly to how a human being learns from experience, a deep learning algorithm repeatedly performs a task which is associated to a performance measure. The goal is to improve this performance by tweaking the way the task is performed each time. The word "deep" refers to the multiple layers of the neural network which enable learning. The recent increased interest in deep learning is due to the increased amount of data generation in recent years, as well as the stronger computing power available. This has allowed deep learning capabilities to grow in recent years, leading to its success in a wide range of applications such as image recognition, speech recognition and natural language processing.

Learning can be supervised or unsupervised. Supervised learning refers to the case where the training data points are input-output pairs and the task of the algorithm is to learn the mapping from the input to the output. The goal is to then use this mapping for new examples, where the output variables are predicted for new input data. In unsupervised learning, the algorithms are left to their own devices to discover previously unknown patterns and structures in the data, without pre-existing labels. In this case, we only have input data and no corresponding output. Other branches of ML include semi-supervised and reinforcement learning. In this paper we will focus on supervised learning.

3.1 Neural Networks

The power of the human brain is due to the sheer complexity of the connections between neurons. The brain exhibits huge parallelism, with each neuron connected to many other neurons. This is reflected in the design of ANNs. It is this parallelism that leads to robust networks. In fact, due to this high degree of parallelism and multiple computation paths, a small number of errors can be tolerated without affecting the result of the computation.

We can describe the ANN as a directed graph with computation units situated at the vertices and weights on the directed edges. Some of the nodes may be distinguished as input nodes, which receive signals from the data, and some as the output nodes. The nodes have activations and their activation influences those of their neighbours. The degree to which the activation of one node influences those of its neighbours is determined by the weights on the edges. The process of 'learning' is the adjustment of these weights.

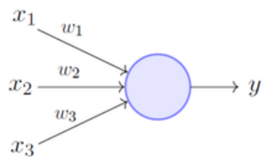

Figure 1: Graphical representation of a neuron [8].

Frank Rosenblatt set the foundations for neural networks with his paper entitled 'The Perceptron: A Perceiving and Recognizing Automaton' in 1957 [9]. The perceptron takes several binary inputs $x_1, x_2, \ldots$ and produces a single binary output. The weights $w_1, w_2, \ldots$ are real numbers expressing the importance of the respective inputs. The output, which can take the values 0 or 1, is determined by whether the weighted sum is less than or greater than some threshold value:

$$\text{output} = \begin{cases} 0 & \text{if } \sum_j w_j x_j \leq \text{ threshold} \\ 1 & \text{if } \sum_j w_j x_j > \text{ threshold.} \end{cases}$$

To simplify the notation, we can write the weighted sum $\sum_j w_j x_j$ as a dot product $w \cdot x$, where w and x are the weight and input vectors respectively. We also move the threshold to the other side of the inequality and replace it by what is known as the perceptron's bias such that, $b = -\text{threshold}$. Now the above equation can be written as:

$$\text{output} = \begin{cases} 0 & \text{if } w \cdot x + b \leq 0 \\ 1 & \text{if } w \cdot x + b > 0. \end{cases}$$

In supervised learning, we can write learning algorithms which choose the optimal weights and biases, by comparing the neural network's output with the training data output. The algorithm will bring small changes to the weights or biases, which should lead to a small corresponding changes in the output. But this is not the case with the perceptron, as a small change in these parameters can sometimes cause the output to flip between 0 and 1.

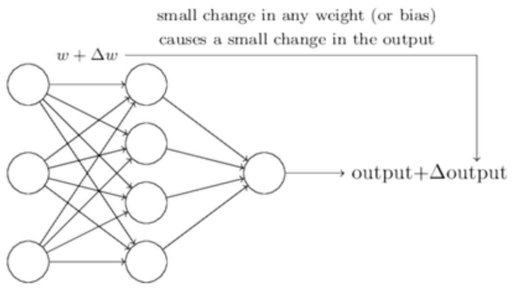
small change in any weight (or bias)
causes a small change in the output
$w + \Delta w$
output+Δoutput

Figure 2: Effect of parameter change on the output [8].

This problem was overcome by introducing the sigmoid neuron. In this case the output can take any real value between 0 and 1. In fact, the output is now $\sigma(w \cdot x + b)$, where σ is the sigmoid function, defined by:

$$\sigma(z) = \frac{1}{1 + \mathrm{e}^{-z}}.$$

The sigmoid function acts an activation function and is very commonly-used.

There exists a wide variety of neural network architectures, as can be observed in figure 3.

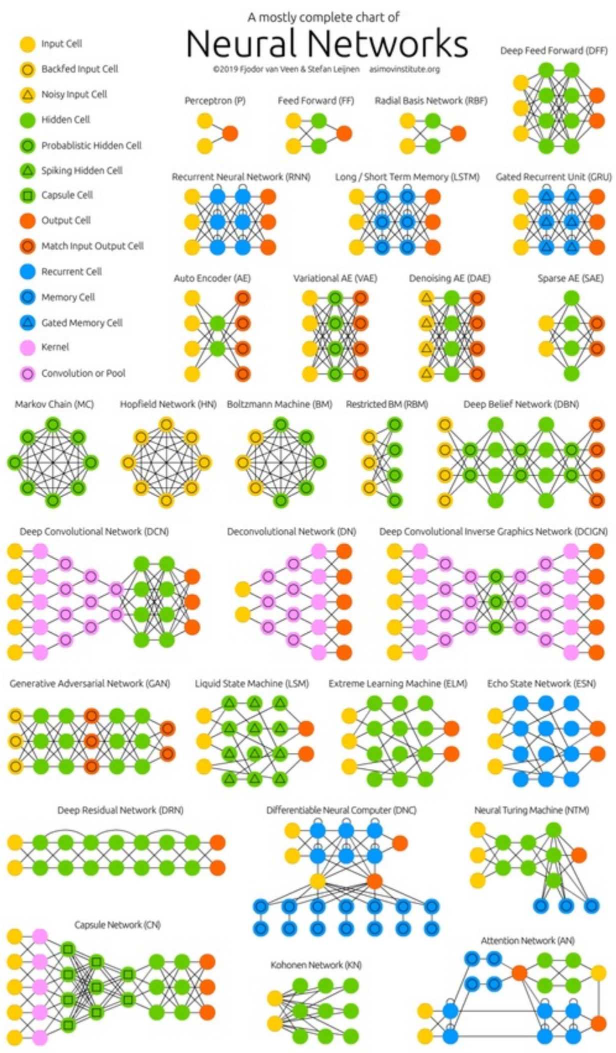
A mostly complete chart of
Neural Networks
©2019 Fjodor van Veen & Stefan Leijnen asimovinstitute.org
Input Cell
Backfed Input Cell
Noisy Input Cell
Hidden Cell
Probablistic Hidden Cell
Spiking Hidden Cell
Capsule Cell
Output Cell
Match Input Output Cell
Recurrent Cell
Memory Cell
Gated Memory Cell
Kernel
Convolution or Pool
Perceptron (P)
Feed Forward (FF)
Radial Basis Network (RBF)
Deep Feed Forward (DFF)
Recurrent Neural Network (RNN)
Long / Short Term Memory (LSTM)
Gated Recurrent Unit (GRU)
Auto Encoder (AE)
Variational AE (VAE)
Denoising AE (DAE)
Sparse AE (SAE)
Markov Chain (MC)
Hopfield Network (HN)
Boltzmann Machine (BM)
Restricted BM (RBM)
Deep Belief Network (DBN)
Deep Convolutional Network (DCN)
Deconvolutional Network (DN)
Deep Convolutional Inverse Graphics Network (DCIGN)
Generative Adversarial Network (GAN)
Liquid State Machine (LSM)
Extreme Learning Machine (ELM)
Echo State Network (ESN)
Deep Residual Network (DRN)
Differentiable Neural Computer (DNC)
Neural Turing Machine (NTM)
Capsule Network (CN)
Kohonen Network (KN)
Attention Network (AN)

Figure 3: Different architectures of neural networks.

In this paper we focus on "Deep Feedforward Networks", which are the most common type of neural network in practical applications. Here, the underlying directed graph is acyclic, which means that there are no directed cycles.

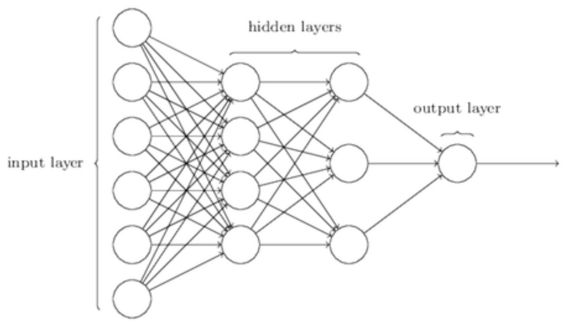

Figure 4: Deep feedforward network example [8].

In figure 4, we can observe the architecture of a simple feedforward network. The leftmost layer is called the input layer and the neurons within this layer are called input neurons. The rightmost layer is called output layer and contains the output neurons. The middle layers are called hidden layers. This network is a four-layer network with two hidden layers. A network is called 'deep' if it has more than one hidden layer. The number of layers in the network is referred to as the 'depth' of the neural network and the number of neurons in a layer is referred to as the 'width' of that particular layer. So the depth of this particular example is 4 and the hidden layers have a width of 4 and 3 respectively.

Each of the neurons receive and transmit information forward, along the relevant edges of the directed graph. Each edge has a weight w_{ij} that represents the strength of the connection between the neurons i and j. For each non-input neuron j, there is a corresponding nonlinear activation function ϕ_j and a bias b_j. The information arriving to neuron j is aggregated by taking the weighted sum according to the weights on the edges leading to this neuron. The value of this information is given by: $\sum_i w_{ij}x_i$. The information transmitted from this neuron is: $\phi_j\left(b_j + \sum_i w_{ij}x_i\right)$ for $i = 1, 2, \ldots, d$, where d is the number of neurons in the previous layer.

I will describe these calculations for a four-layer network with m_1, m_2, m_3 and m_4 being the respective widths of the layers. First, I look at the information transmitted by a neuron j on the first hidden layer.

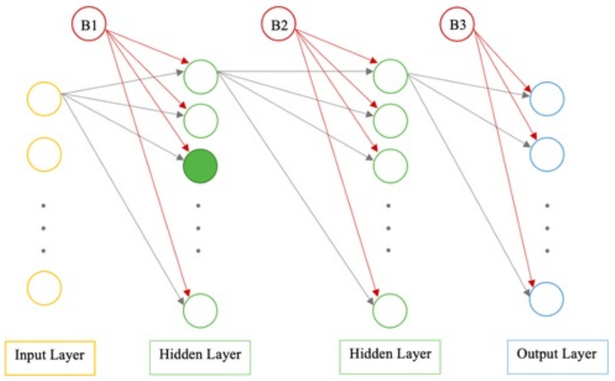

Figure 5: jth neuron on the first hidden layer of a four-layer network represented in green.

In this case, the jth neuron of the first hidden layer is connected to the input vector $\mathbf{x}$ via a biased weighted sum and an activation function ϕ_j:

$$l_j^{(1)} = \phi_j \left(b_j^{(1)} + \sum_{i=1}^{m_1} w_{ij}^{(1)} x_i \right).$$

Now, we look at the kth neuron of the second hidden layer:

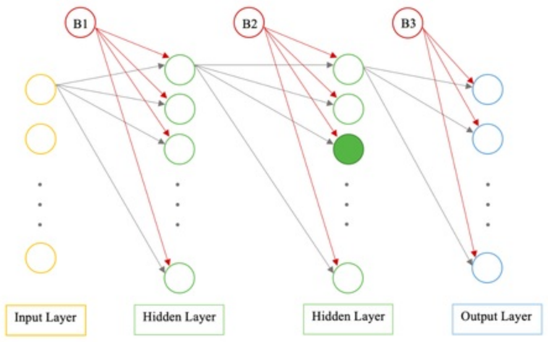

Figure 6: kth neuron on the second hidden layer of a four-layer network represented in green.

In this case the information transmitted is given by:

$$l_k^{(2)} = \psi_k \left(b_k^{(2)} + \sum_{j=1}^{m_2} w_{jk}^{(2)} \cdot \phi_j \left(b_j^{(1)} + \sum_{i=1}^{m_1} w_{ij}^{(1)} x_j \right) \right).$$

Finally, let's look at the hth neuron of the output layer:

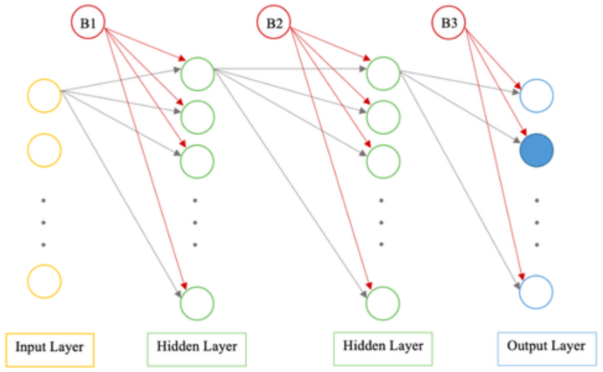

Figure 7: hth neuron on the output layer of a four-layer network represented in blue.

In this case the information transmitted is given by:

$$y_h = \sigma_h \left(b_h^{(3)} + \sum_{l=1}^{m_3} w_{lh}^{(3)} \cdot \psi_l \left(b_l^{(2)} + \sum_{j=1}^{m_2} w_{jk}^{(2)} \cdot \phi_k \left(b_j^{(1)} + \sum_{i=1}^{m_1} w_{ij}^{(1)} x_j \right)\right)\right).$$

Here, $\phi, \psi, \sigma : \mathbb{R} \to \mathbb{R}$ are activation functions for the respective neurons and the bracketed superscripts refer to the layer in question. In the above figures, $B1, B2$ and $B3$ represent the bias parameters. Only the directed edges of the first neuron in each layer is represented to keep the figure easy to understand.

For the above example of four-layer network, the total number of possible combinations for these parameters (i.e. the weights, biases and the activation functions) is:

$$m_1 \cdot m_2 + m_2 \cdot m_3 + m_3 \cdot m_4 + 2 \cdot m_2 + 2 \cdot m_3 + 2 \cdot m_4,$$

where the first three terms are due to weight combinations and the last three terms are due to bias and activation function combinations. These are the parameters that are optimised during the 'learning' process. We will call this parameter set θ. For this optimisation, we need to define a loss function $L(\theta; x, y)$, which will determine the performance of a given parameter set θ. The goal is to find θ^* which minimises our loss function.

3.2 Stochastic Gradient Descent

We briefly touched upon the subject of optimising the neural network parameters θ in the previous section. This optimisation is a minimisation problem. To minimise the loss function $L(\theta; x, y)$, we need to resort to a numerical solution as it is impossible to find an exact analytical solution. The most popular optimisation techniques are variants of gradient descent.

Consider a function $f : \mathbb{R}^d \to \mathbb{R}$ which takes a d-dimensional vector $\mathbf{x} = [x_1, \ldots, x_d]^\top$ as input. The gradient of f with respect to $\mathbf{x}$ is defined by the following vector:

$$\nabla_{\mathbf{x}} f(\mathbf{x}) = \left[\frac{\partial f(\mathbf{x})}{\partial x_1}, \frac{\partial f(\mathbf{x})}{\partial x_2}, \ldots, \frac{\partial f(\mathbf{x})}{\partial x_d}\right]^\top,$$

where the $\partial f(\mathbf{x})/\partial x_i$ represent the partial derivatives of f with respect to x_i, for $i \in \{1, \ldots, d\}$. To simplify our notation we write $\nabla_{\mathbf{x}} f(\mathbf{x})$ as $\nabla f(\mathbf{x})$.

The directional derivative of f along a given vector $\mathbf{u}$ at a given point $\mathbf{x}$ represents the rate at which the function changes at the given point in the direction $\mathbf{u}$:

$$D_{\mathbf{u}} f(\mathbf{x}) = \lim_{h \to 0} \frac{f(\mathbf{x} + h\mathbf{u}) - f(\mathbf{x})}{h}.$$

This can also be rewritten as:

$$D_{\mathbf{u}} f(\mathbf{x}) = \nabla f(\mathbf{x}) \cdot \mathbf{u}.$$

To minimise f, we need to find the direction in which f can be reduced fastest. Since $D_{\mathbf{u}} f(\mathbf{x})$ gives the rates of change of f at the point $\mathbf{x}$ in all possible directions, minimising f comes down to minimising $D_{\mathbf{u}} f(\mathbf{x})$ with respect to $\mathbf{u}$.

We have that:

$$D_{\mathbf{u}} f(\mathbf{x}) = \|\nabla f(\mathbf{x})\| \cdot \|\mathbf{u}\| \cdot \cos(\theta) = \|\nabla f(\mathbf{x})\| \cdot \cos(\theta),$$

where θ is the angle between $\nabla f(\mathbf{x})$ and $\mathbf{u}$. The cosine function $\cos(\theta)$ is minimised when $\theta = \pi$. This means that $D_{\mathbf{u}} f(\mathbf{x})$ is minimised when $\mathbf{u}$ and $\nabla f(\mathbf{x})$ are in opposite directions. This iterative formula allows us to reduce the value of f:

$$\mathbf{x} := \mathbf{x} - \gamma \nabla f(\mathbf{x}),$$

where γ is the learning rate.

When the training data is large, the classical gradient descent algorithm is computationally too expensive. Thus, a stochastic version of the algorithm is often used in the context of neural networks. When training neural network models, the objective function is given by the sum of n functions:

$$L(\mathbf{x}) = \frac{1}{n} \sum_{i=1}^{n} L_i(\mathbf{x}),$$

where $L_i(\mathbf{x})$ is a loss function corresponding to the training data instance indexed by i. Note that for each iteration, the computational cost increases linearly with the training data set size n. This is the reason why the computation cost is very high when the training data is large. Stochastic gradient descent (SGD) offers a solution to this problem. At each iteration, SGD computes $\nabla L_i(\mathbf{x})$ instead of $\nabla L(\mathbf{x})$, where the i is uniformly sampled. $\nabla L_i(\mathbf{x})$ is an unbiased estimator of $\nabla L(\mathbf{x})$ since:

$$\mathbb{E}_i \nabla L_i(\mathbf{x}) = \frac{1}{n} \sum_{i=1}^{n} \nabla L_i(\mathbf{x}) = \nabla L(\mathbf{x}).$$

At each iteration a different mini-batch $\mathcal{B}$ is sampled from the training data. In this case, we have

$$\nabla L_{\mathcal{B}}(\mathbf{x}) = \frac{1}{|\mathcal{B}|} \sum_{i \in \mathcal{B}} \nabla L_i(\mathbf{x}).$$

Then $\mathbf{x}$ is updated as:

$$\mathbf{x} := \mathbf{x} - \gamma \nabla L_{\mathcal{B}}(\mathbf{x}),$$

where $|\mathcal{B}|$ denotes the size of the mini-batch and γ is the learning rate. As previously, $\nabla L_{\mathcal{B}}(\mathbf{x})$ is an unbiased estimator for $\nabla L(\mathbf{x})$:

$$\mathbb{E}_{\mathcal{B}} \nabla L_{\mathcal{B}}(\mathbf{x}) = \nabla L(\mathbf{x}).$$

The computational cost is $\mathcal{O}(|\mathcal{B}|)$ for each iteration. Thus, when the size of the mini-batch is small, the computational cost remains low.

3.3 Universal Approximation Theorem

Neural networks have proven themselves to perform extremely well empirically, but so far we have not discussed the theoretical results underlying the approximation capabilities of artificial neural networks.

First we discuss the Kolmogorov-Arnold representation theorem, one of the classical theorems in approximation theorey. This theorem could explain the motivation of using layered feedforward networks to approximate functions.

Theorem 3.1 (Kolmogorov-Arnold representation theorem)**.** *Any continuous real-valued function $f(x_1, x_2, \ldots, x_n)$ defined on $[0, 1]$, with $n \geq 2$, can be represented in the form:*

$$f(\mathbf{x}) = f(x_1, x_2, \ldots, x_n) = \sum_{j=1}^{2n+1} g_j \left(\sum_{i=1}^{n} \phi_{ij}(x_i) \right),$$

where the g_j are properly chosen continuous functions of one variable and the ϕ_{ij} are continuous monotonically increasing functions independent of f.

This theorem can be explained by the following network architecture:

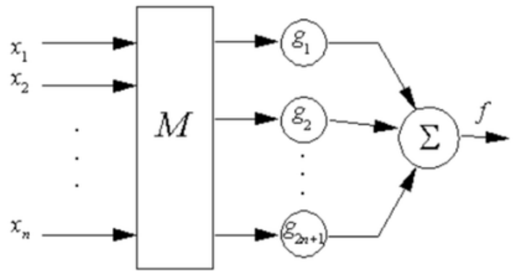

Figure 8: Graphical representation of Kolmogorov's theorem.

Here M maps $\mathbb{R}^n$ into several uni-dimensional transformations. The theorem states that one can express a continuous multivariate function on a compact set in terms of sums and compositions of a finite number of single variable functions.

Girosi and Poggio [10], pointed out that the ϕ_{ij} functions are highly non-smooth and the functions g_j are not parameterised, which means that the Kolmogorov-Arnold representation theorem cannot be relevant to neural networks. KurkovÃą [11] contradicted this statement by saying that non-smooth functions can be approximated as sums of infinite series of smooth functions, thus one should be able to approximately implement ϕ_{ij} and g_j with parameterised networks. Later, Lin and Unbehauen [12] showed that an "approximate" implementation of g_j does not lead to an approximate implementation of f. Even if Kolmogorov's theorem cannot be directly applied to proving the universality of neural networks as function approximators, it points to the feasibility of using parallel and layered network structures for multivariate function mappings.

Later, rigorous mathematical proofs were given to show that feedforward layered neural networks employing continuous sigmoid type, as well as other more general, activation functions can approximate continuous functions. The universal approximation theorem states that a feed-forward neural network with a single hidden layer, containing a finite number of neurons can approximate continuous functions on compact subsets of $\mathbb{R}^n$. George Cybenko [13] proved this theorem for sigmoid activation function in 1989. In 1991, Kurt Hornik [14] proved that it is the architecture of the neural network and not the choice of the activation function that makes neural networks into universal approximators.

In 2017, Lu et al. proved the universal approximation theorem for width-bounded deep neural networks. They showed that a width of less or equal to n+4 is enough to approximate any Lebesgue-integrable function of n-dimensional input variables with ReLU activation function.

Later in 2017, Hanin showed that a width of n+1 is sufficient to approximate any continuous convex function of n-dimensional input variables.

The universal approximation theorem can be expressed mathematically:

Theorem 3.2 (Universal Approximation Theorem)**.** *Let $\phi : \mathbb{R} \to \mathbb{R}$ be a non-constant, bounded and continiuos function and I_m be the m-dimension unit hypercube. Then, given any ϵ and any continuous, real-valued function $f \in C(I_m)$, there exists $N \in \mathbb{N}$ and $v_i, b_i \in \mathbb{R}$ and $\mathbf{w}_i \in \mathbb{R}^m$ for $i = 1, \dots, N$, such that:*

$$F(\mathbf{x}) = \sum_{i=1}^{N} v_i \phi(\mathbf{w}_i \cdot \mathbf{x} + b_i),$$

satisfies $|F(\mathbf{x}) - f(\mathbf{x})| < \epsilon$ for all $\mathbf{x} \in I_m$.

4 Algorithm

The PDEs I am interested in solving are semilinear parabolic PDEs. These can be represented by:

$$\begin{cases} \partial_t u(t,x) + \frac{1}{2}\mathrm{Tr}\left(\sigma\sigma^T(t,x)(\mathrm{Hess}_x u)(x,t)\right) + \nabla u(t,x)\cdot\mu(t,x) + f\left(t,x,u(t,x),\sigma^T(t,x)\nabla u(t,x)\right) = 0 \\ u(T,x) = g(x). \end{cases} \tag{4.1}$$

Here t and x represent the time and the d-dimensional space variables respectively, μ is a known vector-valued function, σ is a known $d \times d$ matrix-valued function, σ^T is the transpose of σ, ∇u and $\mathrm{Hess}_x u$ denote the gradient and the Hessian of u with respect to x, Tr denotes the trace of a matrix and f is a known nonlinear function. $g(x)$ is the terminal condition. We are interested in solving u at $t = 0$ and $x = \xi$ for $\xi \in \mathbb{R}^d$.

The first step of the algorithm consists in reformulating the PDE as BSDE. The BSDE is viewed as a stochastic control problem with the gradient of the solution being the policy function. This policy function is then approximated by a deep neural network.

4.1 Introduction to BSDE

Define the probability space $(\Omega, \mathcal{F}, \mathbb{P})$, let $W : [0,T] \times \Omega \to \mathbb{R}^d$ be a $\mathbb{R}^d$-valued standard Brownian motion, let $\{\mathcal{F}_t\}_{t\geq 0}$ be the normal filtration generated by W and let $\xi \in \mathbb{R}^d$ be a $\mathcal{F}_0$-measurable random variable. Then consider the following stochastic forward differential equation:

$$X_t = \xi + \int_0^t \mu(s, X_s)ds + \int_0^t \sigma(s, X_s)dW_s, \tag{4.2}$$

where $\mu : \mathbb{R}_+ \times \mathbb{R}^d \to \mathbb{R}^d$ and $\sigma : \mathbb{R}_+ \times \mathbb{R}^d \to \mathbb{R}^{d\times d}$.

For a terminal time $T > 0$, the solution of a backward stochastic differential equation (BSD) is a pair of square integrable adapted processes $(Y_t, Z_t)_{t\leq T}$ such that:

$$Y_t = \zeta + \int_t^T f(s, Y_s, Z_s)ds - \int_t^T (Z_s)^T dW_s, \quad t \leq T. \tag{4.3}$$

Here ζ is the terminal value, a $\mathcal{F}_T$-measureable random variable. Contrary to forward SDEs, the solution is not known at the initial time 0 but at the terminal time T. The function f is called the generator.

BSDEs were introduced in 1973 by J.M.Bismut [15] as equation for the adjoint process in the stochastic version of Pontryagin maximum principle. In this case f was linear with respect to (Y, Z). In 1990, Pardoux and Peng [16] generalised the notion and considered general BSDEs and proved existence and uniqueness of the solution.

Since then, BSDEs have been regularly used in mathematical finance, as any pricing problem can be written in terms of linear BSDEs or nonlinear BSDEs when constraints are taken into account.

4.2 The Nonlinear Feynman-Kac Formula

In this section, I introduce the link between nonlinear parabolic PDEs and BSDEs. This link has been studied in depth in the literature (cf. [17–19]). For this, we consider the Markovian BSDE. This is a special case where the terminal condition and the generator depend on the solution of a SDE. These Markovian BSDEs give a nonlinear Feynman-Kac representation of some nonlinear parabolic PDEs. Keeping the definitions introduce in the previous section, conside the following Forward-Backward Stochastic Differential Equation (FBSDE):

$$\begin{cases} X_t = \xi + \int_0^t \mu(s, X_s)ds + \int_0^t \sigma(s, X_s)dW_s & \text{(4.4.a)} \\ Y_t = g(X_T) + \int_t^T f(s, X_s, Y_s, Z_s)ds - \int_t^T (Z_s)^T dW_s. & \text{(4.4.b)} \end{cases}$$

In this system, X is called the forward component and (Y, Z) is the backward component. This system is called a "decoupled" FBSDE, since the solution of the backward component does not appear in the coefficients of the forward component. We want to find the $\{\mathcal{F}_t\}_{t\in[0,T]}$-adapted solution process $\{(X_t, Y_t, Z_t)\}_{t\in[0,T]}$ with values in $\mathbb{R}^d \times \mathbb{R} \times \mathbb{R}^d$. It has been proven that under suitable regularity assumptions on the coefficient functions μ, σ and f, there exists an up-to-indistinguishabiliy unique solution (see [17] and [19]). In addition, the solution of the FBSDE will be a solution of the associated PDE (4.1), i.e. for all $t \in [0, T]$ it holds $\mathbb{P}$-a.s. that:

$$Y_t = u(t, X_t) \in \mathbb{R} \qquad\qquad Z_t = \sigma^T(t, X_t)\nabla u(t, X_t) \in \mathbb{R}^d. \tag{4.5}$$

Now, we plug (4.5) into (4.1) and rewrite the equation forwardly:

$$\begin{aligned} &\text{u(t,X}_t) = \\ &\text{u}(0,\, \xi) - \textstyle\int_0^t f(s, X_s, u(s, X_s), \sigma^T(s, X_s)\nabla u(s, X_s))ds + \int_0^t [\nabla u(s, X_s)]^T \sigma(s, X_s)dW_s. \end{aligned} \tag{4.6}$$

Now, to compute $u(0, \xi)$ we can solve the BSDE and find the value of Y_0.

4.3 Discretisation of the Backward Stochastic Differential Equation

To derive the algorithm to solve the FBSDE, we a apply a discretisation to (4.4) and (4.6).

For $N \in \mathbb{N}$, let $t_0, t_1, \ldots, t_N \in [0, T]$ be real number such that:

$$0 = t_0 < t_1 < \cdots < t_N = T.$$

Then for sufficiently large N, we consider an Euler Scheme for $n = 1, \dots, N-1$:

$$X_{t_{n+1}} \approx X_{t_n} + \mu(t_n, X_{t_n})\Delta t_n + \sigma(t_n, X_{t_n})\Delta W_n, \tag{4.7}$$

and

$$\begin{aligned} &\mathrm{u}(\ t_{n+1}, X_{t_{n+1}}) \approx \\ &\mathrm{u}(\ t_n, X_{t_n}) - f\big(t_n, X_{t_n}, u(t_n, X_{t_n}), \sigma^T(t_n, X_{t_n})\nabla u(t_n, X_{t_n})\big)\Delta t_n + \big[\nabla u(t_n, X_{t_n})\big]^T \sigma(t_n, X_{t_n})\Delta W_n, \end{aligned} \tag{4.8}$$

where $\Delta t_n = t_{n+1} - t_n$ and $\Delta W_n = W_{n+1} - W_n$.

The path $\{X_{t_n}\}_{0 \le n \le N}$ can be sampled using Monte Carlo methods using (4.7). Now, to approximate $\{u(t_n, X_{t_n})\}_{0 \le n \le N}$, we only need to approximate the function $x \mapsto \sigma^T(t, X_t)\nabla u(t, X_t)$ at each time step t_n.

4.4 Neural Network Based Approximation

In this step, we use deep learning to approximate $\sigma^T(t_n, x)\nabla u(t_n, x) \in \mathbb{R}^d$ for $x \in \mathbb{R}^d$ and $n \in \{0, 1, \dots, N\}$. Using this approximation, we can then approximate $u(t_n, x) \in \mathbb{R}$ recursively by using (4.8). To approximate the function $\sigma^T(t, X_t)\nabla u(t, X_t)$, we use a multilayer feedforward neural network:

$$\sigma^T(t, X_{t_n})\nabla u(t, X_{t_n}) = (\sigma^T\nabla u)(t, X_{t_n}) \approx (\sigma^T\nabla u)(t, X_{t_n} \mid \theta_n), \tag{4.9}$$

where θ_n denotes the parameters of the neural network at $t = t_n$.

In the examples which we will study, σ satisfies that for all $x \in \mathbb{R}^d$, $\sigma(x) = \mathrm{Id}_{\mathbb{R}^d}$. This means that we only aim to approximate the spatial gradients $\nabla u(t, X_t)$. The initial "guess" of $u(0, \xi) \approx \theta_{u_0}$ and $\nabla u(0, \xi) \approx \theta_{\nabla u_0}$ are parameters of the model, which we aim to optimise. A total of $1 + d$ parameters are necessary for this approximation (1 for θ_{u_0} and d for $\theta_{\nabla u_0}$).

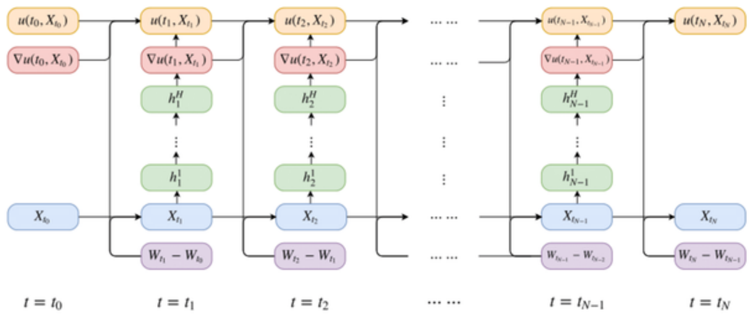

Figure 9: Network architecture for solving the semilinear parabolic PDE [1] .

In the above figure, $h_n^1, \dots, h_n^H$ represent the multilayer feedforward neural network approximating $\nabla u(t, X_t)$ at time $t = t_n$. The network learns the appropriate parameter set θ that leads to an optimal function approximation. This approximation is done with a composition of a chain of simple functions. In total, we employ $N-1$ neural networks. In each of these $N-1$ neural networks, we employ H hidden layers. This means that each network has a total of $H+2$ layers, where $X_{t_n} \in \mathbb{R}^d$ represents the input layer and $\nabla u(t_n, X_{t_n}) \in \mathbb{R}^d$ represents the output layer. The input and the output layers are d-dimensional layers, i.e. they have a width of d. The width of the hidden layers can be chosen to optimise the approximation. In the papers studied ([1] and [2]), a width of $d+10$ was chosen. I will explore a variety of width in later sections of this paper. We adopt a general notation of $d+k$ for the width of the hidden layers. The number of parameters (here weights) used to describe the linear transformation from the d-dimensional first layer to the $(d+k)$-dimensional first hidden layer is $d(d+k)$. The number of parameters used for the linear transformation from one $(d+k)$-dimensional hidden layer to the next $(d+k)$-dimensional hidden layer is $(d+k)^2$. Finally, the number of parameters used from the final $(d+k)$-dimensional hidden layer to the d-dimensional output layer is $d(d+k)$. This means that $2d(d+10)+H(d+k)^2$ parameters are necessary to represent the weights of one neural network. As we employ a total of $N-1$ neural networks, we have a total number of $(N-1)\big(2d(d+k)+H(d+k)^2\big)$ weight parameters to optimise. After each linear transformation described above, we also add a bias (affine linear transformation). We use $d+k$ parameters between each linear transformation and the application of the activation function on the hidden layers. We use d parameters after the final linear transformation. This means that we have a total number of $(N-1)\big(H(d+k)+d\big)$ bias parameters to optimise. The chosen activation function is the rectified linear function (also called ReLU) : $x \mapsto \max\{0, x\}$. Summing the number of parameters discussed above, we get a total number of parameters: $\rho = 1 + d + (N-1)\big(2d(d+k)+H(d+k)^2\big)(N-1)\big(H(d+k)+d\big)$.

All the parameters $\theta = (\theta_1, \dots, \theta_\rho) \in \mathbb{R}^\rho$ are initialized through a normal or a uniform distribution without any pre-training.

4.5 Stochastic Optimization Algorithm

The above described algorithm takes the paths $\{X_{t_n}\}_{0\le n\le N}$ and $\{W_{t_n}\}_{0\le n\le N}$ as input data and gives as final output an approximation of $u(t_N, X_{t_N})$, denoted by $\hat{u}(\{X_{t_n}\}_{0\le n\le N}, \{W_{t_n}\}_{0\le n\le N})$. Now we can define the loss function by the squared absolute difference between this approximation and the actual final value of $u(T, X_T) = g(X_T)$:

$$L(\theta) = \mathbb{E}\left[\left|g(X_T) - \hat{u}(\{X_{t_n}\}_{0\le n\le N}, \{W_{t_n}\}_{0\le n\le N})\right|^2\right]. \tag{4.10}$$

Now, I use a variant of stochastic gradient descent, called Adam optimiser to optimise the parameters θ by minimising the loss function. This variant has been used in [1] and [2], as recommended in [20]. The name Adam is derived from "adaptive moment estimation". While classical stochastic gradient descent maintains a constant learning rate for all weight updates, Adam adapts the learning rates to the parameters from estimates of first and second moments of the gradients.

The authors of [20] describe Adam as combining the advantages of two other extensions of stochastic gradient descent, Adaptive Gradient Algorithm (AdaGrad) and Root Mean Square Propagation (RMSProp). AdaGrad maintains a per-parameter learning rate that improves performance on problems with sparse gradients (e.g. natural language and computer vision problems). RMSProp also maintains per-parameter learning rates that are adapted based on the average of recent magnitudes of the gradients for the weight (e.g. how quickly it is changing). This means that the algorithm does well on online and non-stationary problems (e.g. noisy). Adam offers the benefits of both AdaGrad and RMSProp. Empirical results have shown that Adam works well in practice and is now widely used among practitioners.

The recommended batch size is 64. Denote by $\theta^* \in \mathbb{R}^\rho$ the real vector which minimises the loss function. Denote by $m \in \mathbb{N}_0$ the number of steps in the gradient descent. Through the optimisation, we obtain random approximations $\Theta_0, \Theta_1, \cdots : \Omega \to \mathbb{R}^\rho$ of θ^*. In particular, define $\phi^m : \mathbb{R}^\rho \times \Omega \to \mathbb{R}$ by:

$$\phi^m(\theta, \omega) = |g(X_T^m(\omega) - \hat{u}(\{X_{t_n}^m(\omega)\}_{0 \le n \le N}, \{W_{t_n}^m(\omega)\}_{0 \le n \le N})|^2, \tag{4.11}$$

for all $\theta \in \mathbb{R}^\rho, \omega \in \Omega$ and $m \in \mathbb{N}_0$. Define the function $\Phi^m : \mathbb{R}^\rho \times \Omega \to \mathbb{R}^\rho$ by:

$$\Phi^m(\theta, \omega) = (\nabla_\theta \phi^m)(\theta, \omega), \tag{4.12}$$

and let $\Theta : \mathbb{N}_0 \times \Omega \to \mathbb{R}^\rho$ be a stochastic process such that:

$$\Theta_m = \Theta_{m-1} - \gamma \cdot \Phi^m(\Theta_{m-1}), \tag{4.13}$$

where $\gamma \in (0, \infty)$ is the learning rate. We approximate $u(0, \xi)$ for sufficiently large $\rho, N, m \in \mathbb{N}$ and sufficiently small $\gamma \in (0, \infty)$.

5 Examples for Nonlinear PDEs and Nonlinear BSDEs

In this section, I study extensions of the Black-Scholes model which take in consideration important risk factors in the real markets, such as defaultable securities and higher interest rates for borrowing than for lending. I use the above described algorithm to price European derivatives with two extented Black-Scholes models.

5.1 Pricing of European Financial Derivatives with Different Interest Rates for Borrowing and Lending

This example is based on an example studied in [1]. In this market, the risk free bank account has different interest rates for borrowing and lending. The pricing model takes into account a basket with 100 underlying assets, resulting in a high-dimensional nonlinear PDE. The parameters used here are the same as the ones chosen in the original paper: $\bar{\mu} = \frac{6}{100}$, $\bar{\sigma} = \frac{2}{10}$, $R^l = \frac{4}{100}$, $R^b = \frac{6}{100}$, $d = 100$, $x = (x_1, \ldots, x_d)$, $y \in \mathbb{R}$, $z \in \mathbb{R}^{1 \times d}$, $m \in \mathbb{N}_0$, $T = \frac{1}{2}$, $N = 20$, $\gamma_m = 0.005$, $\mu(t,x) = \bar{\mu}x$, $\sigma(t,x) = \bar{\sigma}\operatorname{diag}_{\mathbb{R}^{d\times d}}(x_1, \ldots, x_d)$, $\xi = (100, \ldots, 100) \in \mathbb{R}^d$. The payoff function is defined by:

$$g(x) = \left(\max_{1\le i\le 100} x_i - 120\right)^+ - 2\left(\max_{1\le i\le 100} x_i - 150\right)^+. \tag{5.1}$$

The nonlinear function f is defined by:

$$f(t,x,y,z) = -R^l y - \frac{\bar{\mu} - R^l}{\bar{\sigma}} \sum_{i=1}^{d} z_i + (R^b - R^l)\left(\frac{1}{\bar{\sigma}} \sum_{i=1}^{d} z_i - y\right)^+. \tag{5.2}$$

The input paths $\{X_{t_n}\}_{0\le n\le N}$ are sampled using an Euler Scheme. I assume that the asset prices follow a geometric Brownian motion:

$$X_{t_{n+1}} = X_{t_n} \exp\left(\left(\bar{\mu} - \frac{\bar{\sigma}^2}{2}\right)\Delta t_n + \bar{\sigma}\Delta W_n\right) \tag{5.3}$$

Now we use the algorithm to numerically approximate $u(0,\xi)$. Denote this approximation value by $\mathcal{U}^{\Theta_m} \approx u(0,\xi)$. This approximation is run 6 times independently with different random seeds. Initially, we choose the number of hidden layers H to be 2 and the width of each hidden layer to be $d+10$. In the below table, I give the mean for $\mathcal{U}^{\Theta_m}$ for $m \in \{0, 1000, 2000, 3000, 4000\}$ based on 256 Monte Carlo samples (validation sample), as well as the mean of the loss function. I also give the standard deviation for these values and the average runtime in seconds.

Number of steps m	Mean of $\mathcal{U}^{\Theta_m}$	Stand. dev. of $\mathcal{U}^{\Theta_m}$	Mean of loss	Stand. dev. of loss	Average runtime in sec.
0	18.786	2.620	50.577	8.091	
1000	20.722	0.552	34.098	4.152	120
2000	21.236	0.058	33.244	4.397	202
3000	21.266	0.0279	33.235	4.534	285
4000	21.273	0.044	33.170	4.534	368

Table 1: Numerical simulation for two hidden layers and d+10 neurons per hidden layer.

I compare the approximation of $u(0,\xi)$ with the value calculated by means of multilevel-Picard approximation in [4], in Section 4.3, Table 6. The value found using this alternative method is 21.299, which is very close to $\mathcal{U}^{\Theta_{4000}} = 21.273$. It is interesting to note that the average runtime for 7 Picard iterations is 8825, which is higher than the average runtime with this method. The standard deviation with the Picard approximation is 0.467, which is nearly 10 times higher.

In the below plots, we can see the mean of $\mathcal{U}^{\Theta_m}$ and the mean of the loss for $m \in \{1, 2, 3, \ldots, 4000\}$

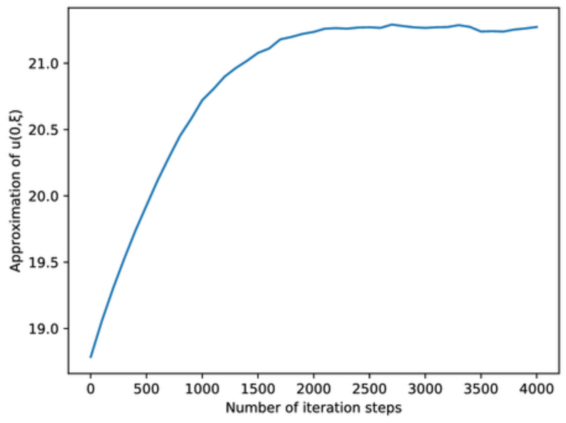

Figure 10: Mean of the approximation $\mathcal{U}^{\Theta_m}$ for $m \in \{1, 2, 3 \ldots, 4000\}$.

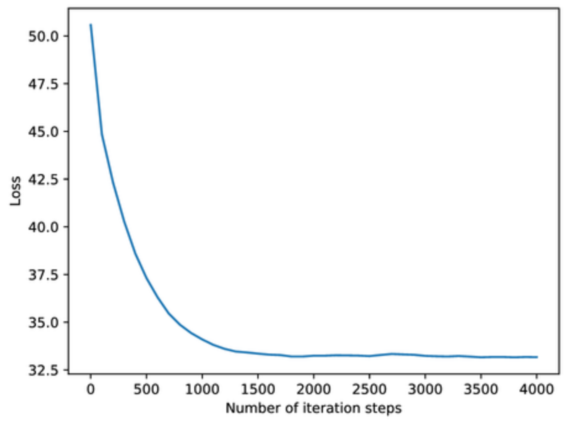

Figure 11: Mean of the loss function for $m \in \{1, 2, 3 \ldots, 4000\}$.

5.2 Pricing of European Financial Derivatives with Default Risk

This example is based on an example studied in [2]. In this case, I consider the fair price of a European option with 100 underlying assets with default risk. If the option issuer defaults, the option holder will only receive a fraction $\delta \in [0, 1)$ of the current option value. The default is modeled by the first jump time of a Poisson process with intensity $\mathcal{Q}$. The default probability is a decreasing function of the current option value, i.e., the default becomes more likely as the value of the option decreases. The parameters used here are the same as the ones chosen in the original paper: $\bar{\mu} = \frac{2}{100}$, $\bar{\sigma} = \frac{2}{10}$, $v^h = 50$, $v^l = 70$, $\gamma^h = 0.2$, $\gamma^l = 0.02$, $d = 100$, $x = (x_1, \ldots, x_d)$, $y \in \mathbb{R}$, $z \in \mathbb{R}^{1\times d}$, $m \in \mathbb{N}_0$, $T = 1$, $N = 40$, $\delta = \frac{2}{3}$, $R = 0.02$, $\gamma_m = 0.008$, $\mu(t, x) = \bar{\mu}x$, $\sigma(t, x) = \bar{\sigma}\mathrm{diag}_{\mathbb{R}^{d\times d}}(x_1, \ldots, x_d)$, $\xi = (100, \ldots, 100) \in \mathbb{R}^d$. The payoff function is defined by:

$$g(x) = \min\{x_1, \ldots, x_d\}. \tag{5.4}$$

The nonlinear function f is defined by:

$$f(t, x, y, z) = -Ry - (1 - \delta)\mathcal{Q}(y)y. \tag{5.5}$$

The intensity function $\mathcal{Q}$ is defined as a piece-wise linear function of the current value of the option within three regions:

$$\mathcal{Q}(y) = \mathbb{1}_{(-\infty,v^h)}(y)\gamma^h + \mathbb{1}_{[v^h,v^l)}(y)\left[\frac{\gamma^h - \gamma^l}{v^h - v^l}\left(y - v^h\right) + \gamma^h\right] + \mathbb{1}_{[v^l,\infty)}(y)\gamma^l. \tag{5.6}$$

The input paths $\{X_{t_n}\}_{0\leq n\leq N}$ are sampled using an Euler Scheme. I assume that the asset prices follow a geometric Brownian motion:

$$X_{t_{n+1}} = X_{t_n}\exp\left(\left(\bar{\mu} - \frac{\bar{\sigma}^2}{2}\right)\Delta t_n + \bar{\sigma}\Delta W_n\right) \tag{5.7}$$

Now, I use the algorithm to numerically approximate $u(0, \xi)$. As in the previous example, the approximation is based on 6 independent runs with different random seeds. The number of hidden layers H is 2 and the width of each hidden layer is $d + 10$. In the below table, I give the mean for $\mathcal{U}^{\Theta_m}$ for $m \in \{0, 1000, 2000, 3000, 4000, 5000, 6000\}$ based on 256 Monte Carlo samples (validation sample), as well as the mean of the loss function. I also give the standard deviation for these values and the average runtime in seconds.

Number of steps m	Mean of $\mathcal{U}^{\Theta_m}$	Stand. dev. of $\mathcal{U}^{\Theta_m}$	Mean of loss	Stand. dev. of loss	Runtime in sec.
0	44.399	1.399	301.031	53.750	110
1000	51.133	1.280	116.255	21.843	462
2000	55.990	1.124	55.867	9.150	714
3000	59.129	0.741	37.339	2.030	972
4000	60.552	0.304	33.825	1.996	1226
5000	60.938	0.052	33.293	2.292	1474
6000	60.977	0.062	33.028	2.275	1713

Table 2: Numerical simulation for two hidden layers and d+10 neurons per hidden layer.

I compare the approximation of $u(0,\xi)$ with the value calculated by means of multilevel-Picard approximation in [4], in Section 4.1, Table 2. The value found using this alternative method is 58.113, which is very close to $\mathcal{U}^{\Theta_{4000}} = 60.977$. The average runtime for 7 Picard iterations is 8453, which is higher than the average runtime with this method. The standard deviation with the Picard approximation is 0.035, which nearly 2 times higher.

In the below plots, we can see the mean of $\mathcal{U}^{\Theta_m}$ and the mean of the loss for $m \in \{1, 2, 3, \ldots, 6000\}$

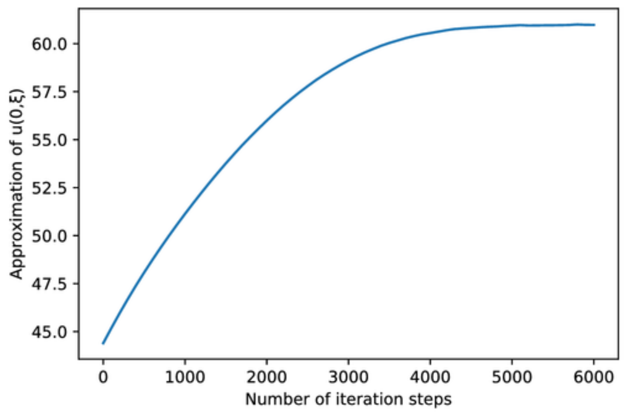

Figure 12: Mean of the approximation $\mathcal{U}^{\Theta_m}$ for $m \in \{1, 2, 3 \ldots, 6000\}$.

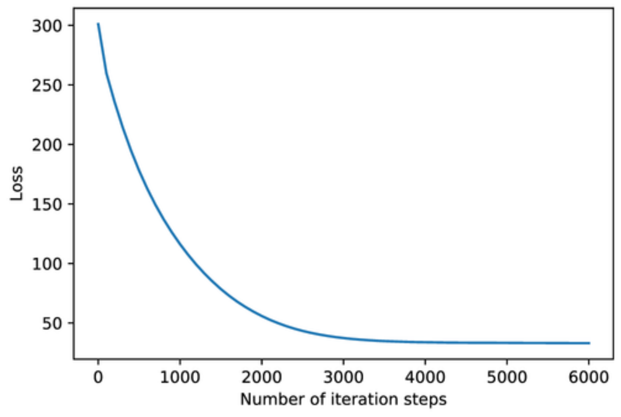

Figure 13: Mean of loss function for $m \in \{1, 2, 3 \ldots, 6000\}$.

6 "Wide" vs "Deep"

The question of whether shallow or deep networks perform better has been discussed by multiple papers, mostly in the context of image recognition. Shallow circuits can require exponentially more components than deeper circuits, which is why the authors of ResNet in [21] tried to make them as thin as possible in favor of increasing their depth and having less parameters. Goodfellow et al. [22] also showed that an increased number of layers results in higher accuracy, stating that "the depth was crucial to our success". They also believe that such deep networks need a large amount of data to train successfully. They did point out however that even though deeper architectures may obtain better accuracy, the returns are diminishing, as can be observed in figure 14.

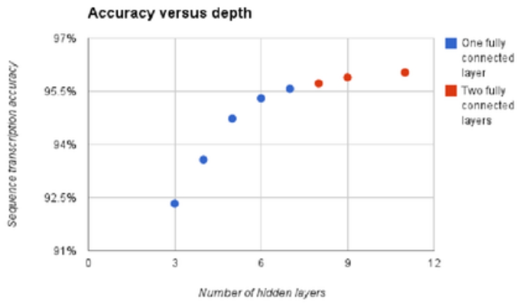

Figure 14: Performance analysis shows the increasing accuracy of deep architectures, with diminishing returns [22].

However, Zagoruyko and Komodakis [23] noted that doubling the number of layers only led to a fraction of a percent of improvement in accuracy, which means that "training very deep residual networks has a problem of diminishing feature reuse, which makes these networks very slow to train". They showed that wide and shallow networks are superior to thin and deep networks. As an example, they showed that a 16-layer network outperforms even a 1000-layer network both in accuracy and efficiently. This can be observed in figure 15.

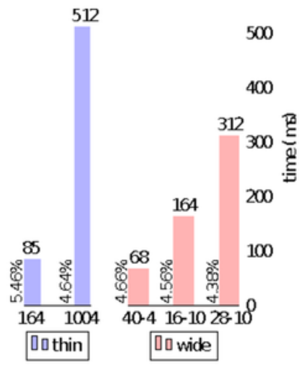

Figure 15: Wide network is 8 times faster than thin network with approximately the same accuracy [23].

Most of the literature discusses the question of depth versus width in the context of image recognition but [24] briefly discussed this problem in the context of option pricing. Their findings show that wide networks lead to higher performance but there are diminishing returns. This was not specifically mentioned in the paper, but it seems like the performance starts to deteriorate after 140 neurons per layer.

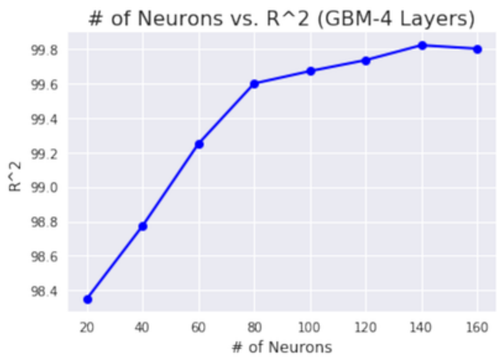

Figure 16: Performance analysis shows the increasing accuracy of wide architectures, with diminishing returns [24].

More interestingly, their analysis shows that an increase in the number of layers leads to higher accuracy only up to 4 layers. Any number of layer greater or equal to 5 leads to lower accuracy. Their interpretation of this is that too many layers lead to over-parametrisation of the problem.

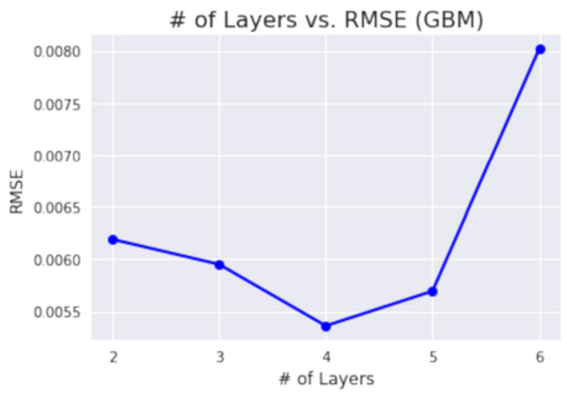

Figure 17: Performance increases up to 4 layers but decreases after 5 layers [24].

6.1 "Wide" vs "Deep" for Example of Different Interest Rates for Borrowing and Lending

I carry out a similar analysis to understand the effect of different numbers of layers as well as different numbers of neurons per hidden layer on the accuracy of the approximation. In figure 11 we can see that the loss stabilises at around $m = 2500$ iterations, so I choose the maximum number of steps to be 2500. First, I keep the number of layers fixed at 2 hidden layers (i.e. fixed depth) and vary the number of neurons per layer (i.e. varying width). Figure 18 shows how the loss changes as the number of neurons per layer increases.

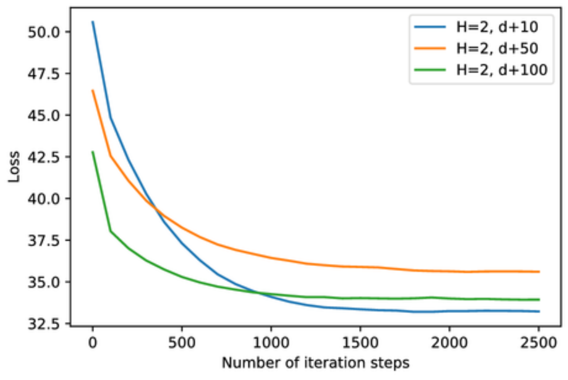

Figure 18: Loss with fixed depth and varying width.

We can observe that the loss converges to the smallest value for d+10 after $m = 2500$ iterations.

Now, I keep the width fixed and vary the depth. Figure 19 shows how the loss changes as the number of hidden layers increases.

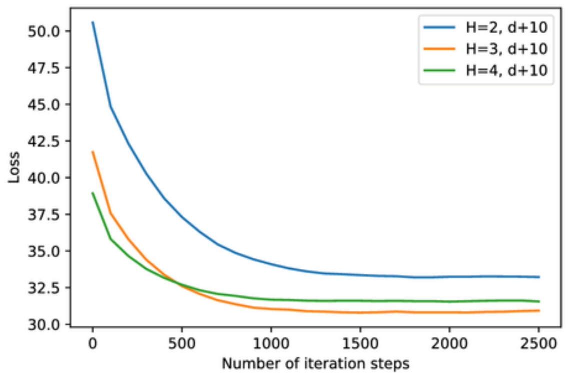

Figure 19: Loss with fixed width and varying depth.

We observe that a higher number of hidden layers results in lower loss, but it is not clear whether a higher number of layers always leads to a better performance, as 3 hidden layers show lower loss than 4 hidden layers.

To get a better overall understanding of the effect of width and depth, I will compare a variety of different combinations of width and depth for the second example.

6.2 "Wide" vs "Deep" for Example of Default Risk

As previously, I analyse the accuracy of the approximation depending on the width and depth.

In figure 13, we can see that the loss stabilises at around $m = 3400$ iterations, so I choose the maximum number of steps to be 3400. Now, I compare a variety of different combinations of width and depth:

	H=2	H=3	H=4	H=5	H=6
d+10	34.550	35.091	36.382	34.443	34.852
d+20	37.279	33.570	33.367	34.091	34.216
d+30	35.049	32.622	34.277	33.726	35.698
d+40	34.691	32.916	34.570	36.486	34.915
d+50	37.775	34.756	34.936	33.902	35.953

Table 3: Numerical simulation for varying width and depth.

I plotted these values as a 3-D plot to visualise the effectiveness of different combinations.

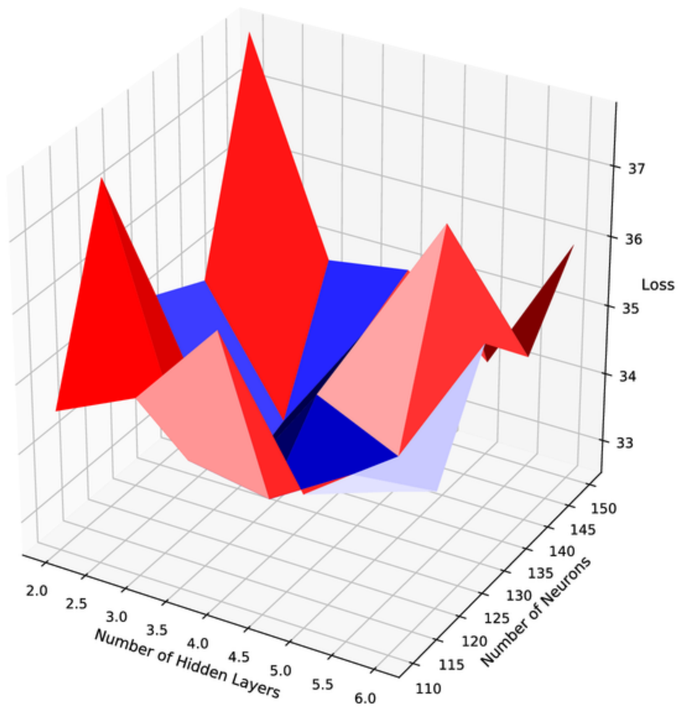

Figure 20: 3-D surface representing the loss for different width and depth combinations.

In this second plot, I used interpolation to get a smooth 3-D plot to make the visualisation clearer.

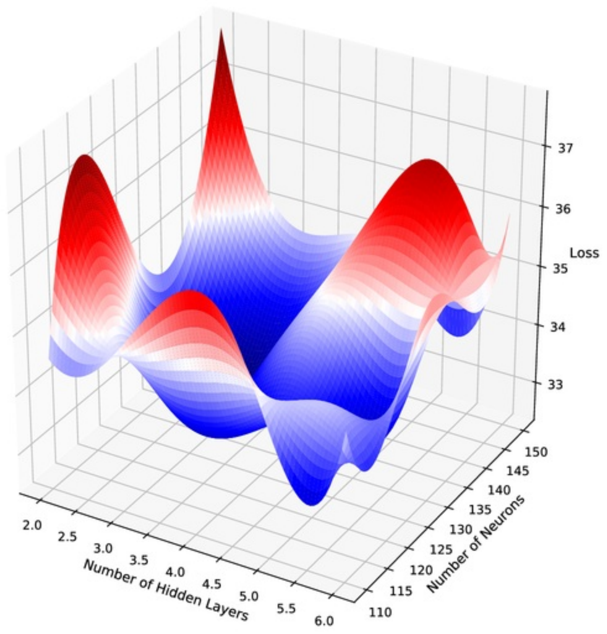

Figure 21: Smooth 3-D surface representing the loss for different width and depth combinations.

The result seems to be similar to [24], as we can observe that the loss increases for more extreme numbers of layers and numbers of neurons per layer. In fact, a combination of a high number of layers and a high number of neurons per layer leads to the highest loss. Lower number of layers, combined with a lower number of neurons per layer also seem to lead to a higher loss level. The best performance seems to be concentrated in the middle of the 3-D surface, around a number of layer equal to 3 or 4 and a number of neurons per hidden layer of 130 to 140.

7 Activation Functions

In this section, I will study different activation functions and analyse the performance of different activation functions in the context of option pricing. Firstly, I will describe the most commonly used activation functions and their advantages and disadvantages.

7.1 Sigmoid Activation Function

Firstly, I will describe the sigmoid function, which I briefly introduced previously:

$$\sigma(z) = \frac{1}{1 + \mathrm{e}^{-z}}.$$

This function was one of the first activation functions used in the context of neural networks, but it is rarely used nowadays, due to a big disadvantage called the "vanishing gradient problem". As sigmoid function's values lie within the interval $[0, 1]$ and due to the S-shape of the function, applying this function to any small or large value will return a value close to zero or one respectively. This means that the gradient becomes close to zero so that it would effectively kill the neuron and prevent the network from learning. This is not as much a problem for shallow networks but with deep networks the progressively dying gradient strongly affects the training.

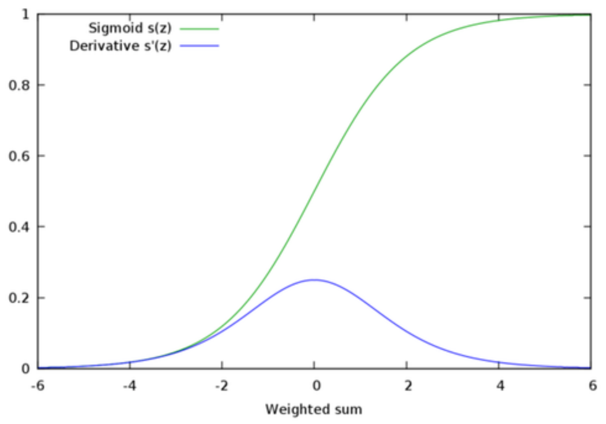

Figure 22: The sigmoid function and its derivative.

7.2 Tanh Activation Function

Tanh is very similar to the sigmoid function, but returns value in the range $[-1, 1]$ instead of $[0, 1]$. This function can be interpreted as a scaled up version of sigmoid, but it still suffers from the vanishing gradient problem. Nowadays, tanh is sometimes used in the final layer but rarely in earlier layers.

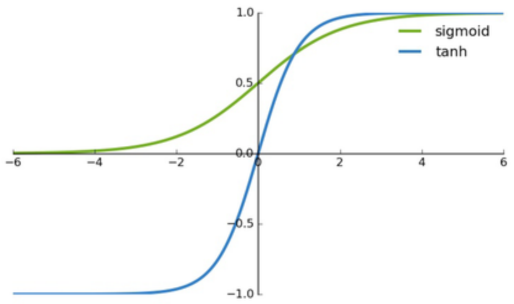

Figure 23: Comparison of tanh and sigmoid functions.

7.3 ReLU Activation Function

The rectified linear activation (ReLU) function is one of the reasons deep neural networks have recently achieved such outstanding results. As of 2017, it is the most widely used activation function for deep neural networks. It was proven in 2011 that this function did a better job at training deep neural networks compared to previously used activation functions [25].

This function is defined by the positive part of its input:

$$f(x) = \max(x, 0)$$

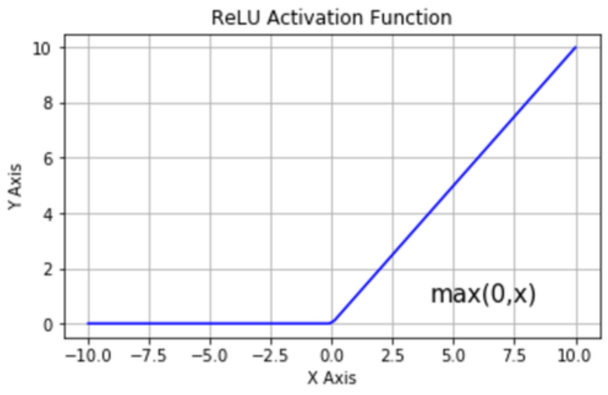

Figure 24: ReLU Activation Function.

This function solves the vanishing gradient problem mentioned before, since its derivative is given by:

$$f'(x) = \begin{cases} 0 & \text{for} \quad x < 0 \\ 1 & \text{for} \quad x \geq 0 \end{cases}$$

In fact, as long as the value of the input is greater or equal to zero, the gradient of the activation function will be 1, which means that the neural network can continue to learn. When the input is less than zero, the output is zero, which means that that specific neuron will not be activated. This is called a sparse representation. It can simplify and accelerate the learning process, but in some cases might hinder the learning. This is called "dying ReLU" problem. This problem can cause several neurons to stop responding, making a substantial part of the network passive. Some variations of ReLU were developed to mitigate this issue, which I will discuss in the coming sections.

7.4 LReLU Activation Function

Leaky ReLU (LReLU) is a modification of ReLU which was introduced to overcome the "dying ReLU" problem. This function allows for small negative values, as we can see in the figure and formula below:

$$f(x) = \begin{cases} ax & \text{for} \quad x < 0 \\ x & \text{for} \quad x \geq 0 \end{cases},$$

where $a \in [0, 1)$.

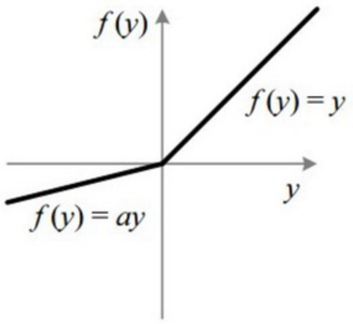

Figure 25: LReLU Activation Function.

This means that the gradient will be small but non-zero. This reduces the sparsity but makes the gradient more robust for optimisation, since now the weights will be adjusted for those neurons that were not active with ReLU.

This modification of the ReLU activation function was introduced in [26], but the analysis of the authors did not show any significant improvement from ReLU.

7.5 ELU Activation Function

Exponential Linear Unit (ELU) is another activation function inspired by ReLU, which has the same aim as LReLU, i.e. increasing the learning speed by not deactivating some neurons. But it also has the advantage of speeding up the process by deactivating some of the neurons, since the slope in most of its negative domain approaches 0.

The function is given by:

$$f(x) = \begin{cases} \alpha(\exp(x) - 1) & \text{for} \quad x \leq 0 \\ x & \text{for} \quad x > 0 \end{cases}$$

.

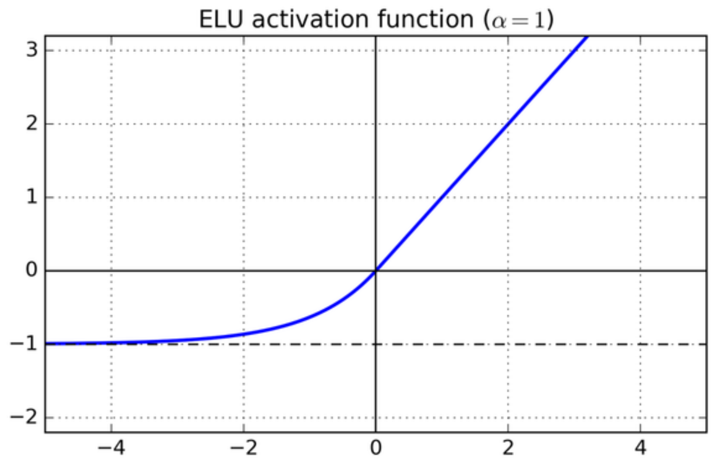

Figure 26: ELU Activation Function.

This activation function was introduced in [27] and the numerical analysis in the paper shows significant improvement from ReLU.

7.6 Application of Activation Functions to the Algorithm

Now I apply the sigmoid, ReLU, LReLU and ELU activation functions to the first option pricing example (different borrowing and lending rates) introduced above, for $d = 50$ and $d = 100$. The algorithm is run 6 times independently for $m \in \{0, 1, \ldots, 2500\}$.

We observe in figures 27 and 28 that in both cases, the sigmoid function shows the worse performance and the ELU function shows the best performance. It is not clear whether LReLU shows any improvement from ReLU as in the case of $d = 50$, ReLU outperforms LReLU.

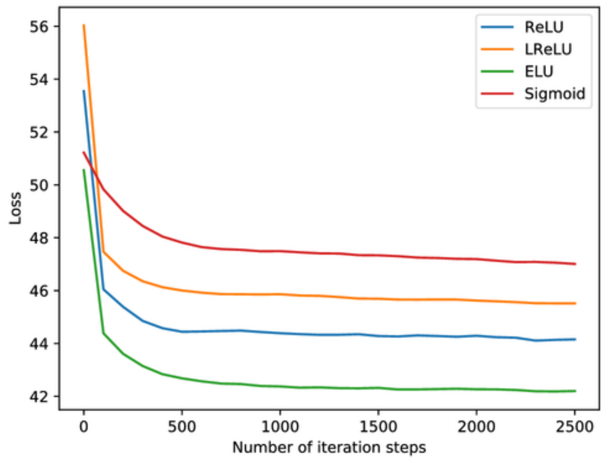

Figure 27: Performance analysis of different activation functions for d=50.

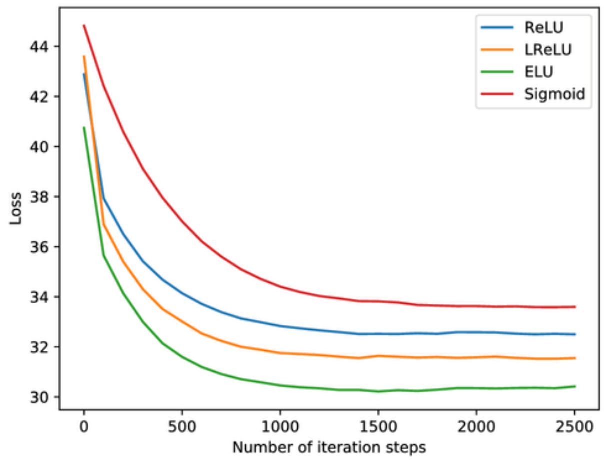

Figure 28: Performance analysis of different activation functions for d=100.

This result is in agreement with the literature (cf. [27–29]), where the authors have shown that nonlinear versions of ReLU (such as ELU) show a better performance than both ReLU and LReLU.

8 Runtime

Now I will analyse the runtime of this algorithm. Firstly, I analyse how the runtime grows with the space dimension of the considered PDE (i.e. the number of underlying assets). I have done the analysis for the example of a European derivative with default risk. The number of iterations is $m = 3400$. In figure 29, we can observe that the runtime only grows linearly with respect to the dimension.

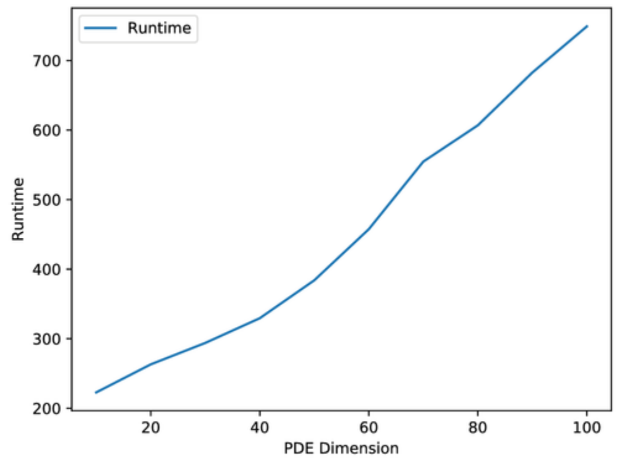

Figure 29: Linear runtime growth with respect to PDE dimension.

This linear growth is the main advantage that this algorithm offers, as it does not suffer from the curse of dimensionality.

I also analyse how the runtime grows with respect to the width and the depth of the neural network. This analysis was done for the example of a European derivative with default risk. The number of iterations is $m = 3400$ and the dimension is $d = 100$. As shown in figure 30, the runtime is high for neural networks with higher number of layers. This should be one of the considerations when choosing the number of layers.

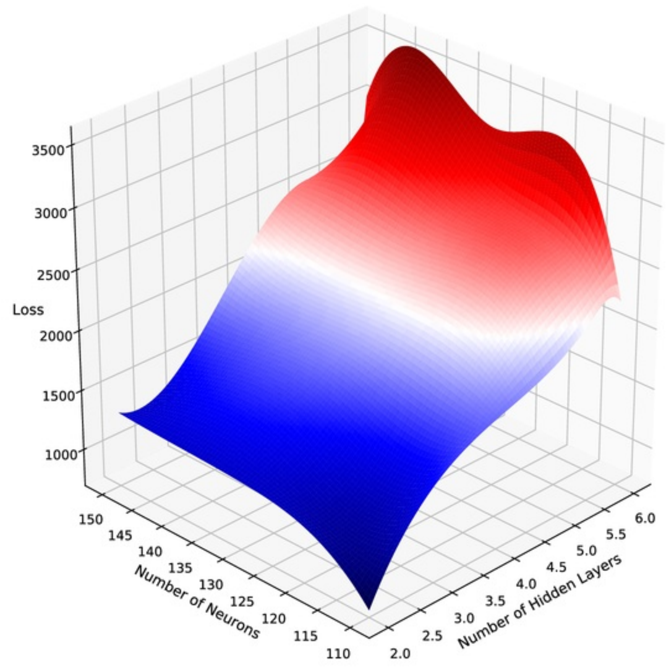

Figure 30: Runtime growth with respect to width and depth.

Conclusion

In this paper, I explore the benefits of implementing neural networks to solve high-dimensional nonlinear PDEs. My numerical analysis shows that this method offers an effective approximation, with results close to the values found with alternative methods. In addition to the accuracy of the approximation, this method allows one to decrease the runtime considerably, which is the main advantage of this method. In fact, the runtime only grows linearly with respect to the dimension of the PDE, which makes this method computationally inexpensive. I analysed different variations of neural network structures by varying the depth and the width of the networks. I observed that the accuracy of the approximation increased with the layers up to a certain point, but diminished eventually. To be precise, a number of hidden layers of 3 or 4 seems to be the optimal. The change in accuracy with respect to the width was less clear. My analysis of the runtime with respect to the different depth and width combinations shows that the runtime increases significantly with the depth and the width. This is another factor to be taken in consideration when choosing the optimal depth and width. When taking into consideration both the accuracy of the approximation and the runtime, I believe that a number of hidden layers of $H = 3$ and a number of neurons per hidden layer of $d + 10$ offers the optimal combination. I also analysed the performance of a variety of activation functions. I observed a significant improvement in the accuracy of the approximation with the nonlinear version of ReLU (ELU).

Finally, I would like to add that the parameters for the neural networks need to be modified based on the problem. The optimal combination of depth and width is dependent on the dimension of the PDE. The optimal choice of the activation function is also dependent on the dimension, as we observed above. Therefore, experimentation and reflection is key to improving the result.

References

[1] W. E, J. Han and A. Jentzen: Deep Learning-Based Numerical Methods for High-Dimensional Parabolic Partial Differential Equations and Backward Stochastic Differential Equations. Preprint available at arXiv:1706.04702.

[2] J. Han, A. Jentzen and W. E: Solving high-dimensional partial differential equations using deep learning. Preprint available at arXiv:1707.02568.

[3] R. E. Bellman: Dynamic Programming (Princeton University Press, 1957).

[4] W. E, M. Hutzenthaler, A. Jentzen and T. Kruse: On full history recursive multilevel Picard approximations and numerical approximations for high-dimensional nonlinear parabolic partial differential equations and high-dimensional nonlinear backward stochastic differential equations (2017). Preprint available at arXiv:1607.03295v2.

[5] J. Sirignano, K. Spiliopoulos: DGM: A deep learning algorithm for solving partial differential equations (2017). Preprint available at arXiv:1708.07469.

[6] A. Al-Aradi, A. Correia, D. Naiff, G. Jardim and Y. Saporito: Solving Nonlinear and High-Dimensional Partial Differential Equations via Deep Learning (2018). Preprint available at arXiv:1811.08782.

[7] F. Black and M. Scholes: The Pricing of Options and Corporate Liabilities, The Journal of Political Economy, Vol. 81, No. 3 (May - Jun., 1973), pp. 637-654.

[8] neuralnetworksanddeeplearning.com, by Michael Nielsen (2019).

[9] F. Rosenblatt: The Perceptron: A Perceiving and Recognizing Automaton, Report 85-60-1, Cornell Aeronautical Laboratory, New York (1957).

[10] F. Girosi and T. Poggio: Representation Properties of Networks: Kolmogorov's Theorem is Irrelevant, Neural Computation, 1(4), 465-469 (1989).

[11] V. Kurkova: Kolmogorov's theorem and multilayer neural networks. Neural Networks 5(3): 501-506 (1992).

[12] J.-N. Lin and R. Unbehauen: On the Realization of a Kolmogorov Network, Neural Computation, 5(1), 21-31 (1993).

[13] G. Cybenko: Approximation by Superpositions of a Sigmoidal Function, Math. Control Signals Systems, 2, 303-314 (1989).

[14] K. Hornik: Approximation Capabilities of Multilayer Feedforward Networks, Neural Networks, 4(2), 251-257 (1991).

[15] J.M. Bismut: Conjugate Convex Functions in Optimal Stochastic Control, J. Math. Anal. Appl.,44, 384-404 (1973).

[16] E. Pardoux and S. Peng : Adapted Solutions of Backward stochastic diïňĂerential equation, Systems and Control Letters 14, 55-61 (1990).

[17] E. Pardoux and S. Peng: Backward stochastic differential equations and quasilinear parabolic partial differential equations. In Stochastic Partial Differential Equations and their Applications (Charlotte, NC, 1991), vol. 176 of Lecture Notes in Control and Inform. Sci., 200-217 (Springer, Berlin, 1992).

[18] E. Pardoux and S. Tang: Forward-backward stochastic differential equations and quasilinear parabolic PDEs. Probability Theory and Related Fields 114, 123-150 (1999).

[19] N. El Karoui, S. Peng and M.-C. Quenez: Backward stochastic differential equations in finance. Mathematical Finance 7, 1-71 (1997).

[20] D. P. Kingma and J. Ba: Adam: a method for stochastic optimization (2015). Preprint available at arXiv:1412.6980.

[21] K. He, X. Zhang, S. Ren and J. Sun: Deep Residual Learning for Image Recognition (2015). Preprint available at arXiv:1512.03385.

[22] I. J. Goodfellow, Y. Bulatov, J. Ibarz, S. Arnoud and V. Shet: Multi-digit Number Recognition from Street View Imagery using Deep Convolutional Neural Networks (2014). Preprint available at arXiv:1312.6082.

[23] S. Zagoruyko and N. Komodakis: Wide Residual Networks (2016). Preprint available at arXiv:1605.07146.

[24] A. Hirsa, T. Karatas and A. Oskoui: Supervised Deep Neural Networks (DNNs) for Pricing/Calibration of Vanilla/Exotic Options Under Various Different Processes (2019). Preprint available at arXiv:1902.05810.

[25] X. Glorot, A. Bordes and Y. Bengio: Deep Sparse Rectifier Neural Networks (2011).

[26] A. L. Maas, A.Y. Hannun and A.Y. Ng: Rectifier Nonlinearities Improve Neural Network Acoustic Models (2013).

[27] D. Clevert, T. Unterthiner and S. Hochreiter: Fast and Accurate Deep Network Learning by Exponential Linear Units (ELUs) (2016).

[28] D. Pedamonti: Comparison of nonlinear activation functions for deep neural networks on MNIST classification task (2018). Preprint available at arXiv:1804.02763.

[29] S. Hayou, A. Doucet, J. Rousseau: On the Impact of the Activation Function on Deep Neural Networks Training (2019). Preprint available at arXiv:1902.06853.

AGHAJANI_TARA_01072173

GRADEMARK REPORT

FINAL GRADE

/0

GENERAL COMMENTS

Instructor

PAGE 1

PAGE 2

PAGE 3

PAGE 4

PAGE 5

PAGE 6

PAGE 7

PAGE 8

PAGE 9

PAGE 10

PAGE 11

PAGE 12

PAGE 13

PAGE 14

PAGE 15

PAGE 16

PAGE 17

PAGE 18

PAGE 19

PAGE 20

PAGE 21

PAGE 22

PAGE 23

PAGE 24

PAGE 25

PAGE 26

PAGE 27

PAGE 28

PAGE 29

PAGE 30

PAGE 31

PAGE 32

PAGE 33

PAGE 34

PAGE 35

PAGE 36

PAGE 37

PAGE 38

PAGE 39

PAGE 40

PAGE 41

PAGE 42