Imperial College London

Department of Mathematics

Deep Solvers

Author: Alberto Moreno de Vega García (CID: 02306375)

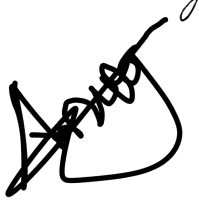
A thesis submitted for the degree of

MSc in Mathematics and Finance, 2022-2023

Declaration

The work contained in this thesis is my own work unless otherwise stated.

Alberto Moreno de Vega García



Acknowledgements

I express my gratitude to my personal tutor and thesis supervisor Damiano Brigo, for his help and support during the project.
I am very grateful to Ming Gao, Anatoli Karolik, and the entire JP Morgan team for their invaluable guidance, insightful contributions, and for giving me the opportunity and resources to participate in this project.
I extend my appreciation to my family, friends, and all those who have played a significant role on my journey thus far. Their unwavering support and presence have been necessary in making this achievement possible.
Finally I own special gratitude to Fundación Ramón Areces for trusting me to receive one of their scholarships for my studies at Imperial College London.

Abstract

In the financial industry it is not uncommon to work with models that have a big number of risk factors leading to a high dimensional pricing problem. An example is the valuation of basket options. Well known numerical techniques such as the finite difference method or the American Monte Carlo have been extensively studied and are widely used in low dimensions. When the dimensionality increases the computational cost of such techniques becomes prohibitively high. This phenomenon is known as the "curse of dimensionality". Moreover in the Counterparty Credit Risk (CCR) space it is required to price a derivative not only at one time instant but through whole simulated paths for the risk factors. For this, it is industry standard to resort to some pricing acceleration techniques that avoid having to compute the price at each time instant for each simulated path.

In this thesis we study Deep Learning techniques that have the potential to solve the existing impediments in high dimensional problems and can be used to accelerate the pricing process. We desire to test if the techniques available in the literature, and two novel ones we propose, can be used in the industry. One of this new approaches allows to work under a parametric approach, making it more generic. This study was carried in the context of the creation of a counterparty credit risk bench-marking library which requires generic pricers that can deal with high dimensionality and different parameters. With this in mind a key advantage of the proposed techniques is that they are general enough to be used under any dynamics of the underlying and payoff structure.

After testing these techniques we have found higher errors than expected and proposed approaches to mitigate this problem that will be studied in further research.

Contents

1 BSDEs **8**
- 1.1 Why BSDEs? . . . 8
- 1.2 Existence, uniqueness and FBSDEs. . . . 10
- 1.3 Generalised Feynman-Kac formula . . . 11

2 Deep Learning **13**
- 2.1 Neural Networks . . . 13
- 2.2 Universal approximation . . . 15
- 2.3 Loss function and Neural network training. . . . 16

3 Solving BSDEs with Neural Networks **18**
- 3.1 A review on the literature. . . . 18
- 3.2 One net at each time . . . 19
  - 3.2.1 A Forward Scheme . . . 19
  - 3.2.2 A Backward Scheme . . . 22
  - 3.2.3 Hybrid approach . . . 23
  - 3.2.4 Sub-network architectures . . . 24
  - 3.2.5 Numerical tests with one network at a time. . . . 25
- 3.3 Common network . . . 27
  - 3.3.1 Approximate $u(t, X_t)$, Raissi's Method. . . . 28
  - 3.3.2 Modelling a shared gradient. . . . 29
  - 3.3.3 Numerical tests on Raissi's method. . . . 29

4 A parametric approach **35**
- 4.1 DeepONet . . . 35
- 4.2 Subnets. . . . 37
- 4.3 Training a DeepONet . . . 37

5 Conclusions and further research. **39**

A Raissi under $r = 10\%$, $\sigma = 3\%$. **40**

B Raissi under $r = 0\%$, $\sigma = 3\%$. **41**

C Raissi under $r = 10\%$, $\sigma = 25\%$, 200 time steps. **42**

Bibliography **46**

List of Figures

1.1 Realisation of diffusion processes under different dynamics. 9

2.1 Feed-forward Neural Network architecture. 14
2.2 ReLU VS ELU activation function. 15

3.1 Data Flow of the Forward BSDE Architecture. 20
3.2 Data Flow of the Backward BSDE Architecture 22
3.3 Price simulated under a forward and backward method. 24
3.4 Sub-network architectures 25
3.5 Comparison of the forward and backward method for pricing a call option. 27
3.6 DEPE and DENE of a 100-D basket call calculated under the forward BSDE solver 27
3.7 Mean of absolute error of pathwise approximation under different Deep Solvers. . . 27
3.8 Data Flow of the BSDE shared gradient Architecture. 30
3.9 Raissi VS Raissi+ELU 30
3.10 Training evolution for Raissi's method. 31
3.11 Evolution of error profile in Raissi method for different time instants. 33
3.12 Pathwise approximation using Raissi's method. 34

4.1 DeepONet Architecture, subnets and sensors. 37

A.1 Evolution of the error profile of Raissi method under 3% volatility, $r = 10\%$. 40

B.1 Evolution of the error profile of Raissi method under 3% volatility, no drift. 41

C.1 Evolution of the error profile of Raissi method under 25% volatility, 200 time steps. 42

List of Tables

3.1 Pricing results under Black Scholes for one net at a time architectures. 26
3.2 Train Raissi method with different architectures. 31

4.1 Parameter Distributions for the DeepONet training. 38

List of Algorithms

1 Stochastic Gradient Descent 17
2 Adam Optimizer 17
3 Deep Forward BSDE Solver 21
4 Deep Backward BSDE Solver 23
5 Deep Raissi BSDE Solver 29

Introduction

A financial derivative is an instrument whose value derives from an underlying asset. These instruments are commonly utilised for hedging and speculation, making the pricing and risk management of derivatives a critical task for financial institutions and investors. As a direct outcome of the financial crisis, significant transformations have taken place in the risk management practices of financial institutions. There is now a widespread acknowledgment that accurate contract valuation must incorporate the risk of counterparty default. This is the Credit Valuation Adjustment or CVA. If an institution considers its own default risk this is done through the Debit Valuation Adjustment or DVA. If relevant, other adjustments like funding costs (FVA), regulatory capital costs (KVA), collateral cost (CollVA), etc. In general, these total valuation adjustments (XVA) are incorporated into the pricing paradigm making calculations more complex. As a starting point it requires valuing the contract not only at time 0 but also at future times.

An example on why the valuation at future times is required can be found in the formulas for CVA or DVA ([12, Def. 2.3.1]):

$$\mathrm{CVA}_0 = \int_0^T \mathbb{E}\left[(1 - Rec_C) D(0,t) \widehat{V}^+(t) 1_{t \leq \tau < t+dt}\right] \tag{0.0.1}$$

$$\mathrm{DVA}_0 = \int_0^T \mathbb{E}\left[(1 - Rec_B) D(0,t) \widehat{V}^-(t) 1_{t \leq \tau < t+dt}\right] \tag{0.0.2}$$

where $D(0,t)$ represents the discount factor between times 0 and time t, Rec_i the recovery from the counterparty (i=C) or the bank (i=B). The above expectation is taken under the risk neutral measure. The terms $V^+(t)$ and $V^-(t)$ are the discounted expected positive and negative exposure, that will be explained in section 3.2.5, and depend on the value of the portfolio in the future.
For this calculation one plausible approach involves simulating a big number of risk factors paths and employing pricing techniques across the different paths. This requires not only the knowledge of prices at valuation time, but also calculating the prices at future time points $t \in (0,T)$.
Then there should be careful consideration to choose the pricing technique for the products in the first place. The most desirable pricing technique will be to have a closed form formula such as the well known Black-Scholes formula ([8]) that computes the price at time t of a European call option on a stock, following a Geometric Brownian Motion (GBM):

$$C_t(S_t, K, \sigma) = S_t \mathcal{N}(d_+) - K \mathrm{e}^{-r(T-t)} \mathcal{N}(d_-), \tag{0.0.3}$$

where

$$d_1 := \frac{ln\frac{S_t}{K} + \left(r + \frac{\sigma^2}{2}\right)(T-t)}{\sigma\sqrt{(T-t)}}, \quad d_2 = d_1 - \sigma\sqrt{(T-t)}$$

and where $\mathcal{N}$ denotes the cumulative distribution function of the Gaussian random variable, K is the strike, σ the volatility, r the risk free rate and T the maturity.

Unfortunately closed form formulas are rarely available for more complex products or underlying dynamics and numerical techniques are required instead. In a front office environment, where only the price at valuation time matters, commonly used techniques include Monte Carlo simulation, finite difference methods or the Longstaff and Schwartz [41] regression, also known as American Monte Carlo. Each technique has some advantages or disadvantages compared with the others. For example finite difference methods are able to provide the so called "Greeks" (sensitivities with respect to the risk factors), that are useful for hedging purposes, American Monte Carlo is able to handle options with early exercise features, such as Bermudan options. This techniques have been

deeply studied and applied in low dimensional problems and are very effective. Their drawback is their difficulties to perform well in high dimensional settings where they become computationally infeasible as their computational complexity increases with the dimension. This is known as the "curse of dimensionality" already postulated in the work of Bellman in [6]. High dimensionality can happen even at an individual trade level if the derivative is affected by a big number of risk factors.
In the counterparty credit risk (CCR) space it will be computationally unfeasible to call a front office pricer for each time instant realization of the risk factors across different paths. As a solution, it is industry standard to resort to fast pricers which use interpolation or regression techniques on prices obtained for a number of points using front office pricers.

An additional layer of complexity is introduced when the assumptions of XVA calculations are relaxed to a more realistic setting. For example in the case of replacement closeout (meaning that at default event the value of the trade is calculated considering default risk of the remaining participant) or asymmetric funding costs the valuation at time t needs to account for the XVA at future times, resulting in a recursive valuation. In this case XVA calculations require very specific mathematical tools as can be found in [50] or [13], where they use variants of the well known American Monte Carlo.

Finally all of the valuation paradigms require a selection of parameters representing the market conditions (volatility, discounting, correlations ...). It would be advantageous to incorporate these parameters as part of our pricing function, enabling us to price derivatives under different market scenarios without the need to re-train the pricing function from the beginning. This means working under a parametric valuation approach. This has been studied in [23] in the context of solving PDEs.

Valuation problems, both with and without default can be formulated as a Backward Stochastic Differential Equation (BSDE), whose solution provides the value of the portfolio at different time instants. The inconvenience is that solving a BSDE is in general not simple. New numerical approaches, that leverage the recent interest and growth in the field of Deep Learning, try to offer solutions to BSDEs, hence opening a set of new derivative valuation techniques. This numerical methods are generally known as Deep (BSDE) Solvers.

In this thesis we investigate different Deep BSDE Solvers present in the literature, and asses their application in the pricing of financial derivatives, which can help in XVA calculations solving some of the above explained challenges. This is of high interest because Deep BSDE Solvers have the potential to combine all the advantages of the traditional numerical methods explained (obtaining the Greeks, handling early exercise options) and offer their owns: not affected by the curse of dimensionality and can be used under any dynamics model. Therefore, once operative, Deep BSDE solvers can be used as generic pricers to work for any product and any model. We have restricted to simple settings for the numerical examples.

The structure of the thesis is as follows: firstly, in chapter 1 we give some background on BSDEs, their usefulness in mathematical finance and connection with PDEs. In chapter 2, we review deep learning covering neural networks, common architectures, important theorems and training techniques. Chapter 3 starts with a summary of the existing literature on the use of neural networks to solve high dimensional problems. We discuss several representative approaches in detail and also propose an improvement to existing methods which significantly reduces numerical errors. Additionally, in this chapter we also include some numerical results to compare numerical accuracy of different methods. This is done in the context of derivative pricing. Finally, in chapter 4, we develop a parametric approach that generalises chapter 3, and is able to deal with the problem of changing market conditions. We do this by using a neural network that not only learns the pricing value as a function of time and state variables, but also parameters. This novel approach has not been studied in the literature in the context of BSDEs and we believe it has great potential to build fast pricing methods.

Chapter 1

BSDEs

In the first chapter we develop the basic background theory for Backward Stochastic Differential Equations BSDEs, their relation with Partial Differential Equations (PDEs) and formulate why BSDEs are of interest in the study of financial mathematics. This part of the dissertation is mainly based on technical references such as the book from Jianfeng Zhang [55], lecture notes from Bouchard [10] and Perkowski [46].

1.1 Why BSDEs?

Differential equations are a well known tool in mathematics, as they are able to represent the dynamics of a deterministic system. They can be understood in terms of relations between how a function changes under infinitesimal changes in its corresponding variables. Although the system is deterministic the relationships can be very complex. When the system stops being deterministic, and we add some noise, we need to use more advanced tools to understand the dynamics of a random system. This is the purpose of Stochastic Differential Equations or SDEs which intuitively characterise the behaviour of a stochastic process over time (in a more formal setting we suppose we are working in a filtered probability space with the usual conditions). The theory of SDEs was developed in the 1940s by Kiyosi Itô [34]. In reality there is no clear notion of what is an infinitesimal change in a random variable. Hence, SDEs have to be understood in an integral sense:

$$\boldsymbol{X}_t = \boldsymbol{X}_0 + \int_0^t \mu\left(s, \boldsymbol{X}_s\right) ds + \int_0^t \sigma\left(s, \boldsymbol{X}_s\right) \boldsymbol{dW}_s,\ 0 \leq t \leq T \tag{1.1.1}$$

where the first integral is in the Lebesgue sense (this term is also known as drift), the second integral is an Itô integral (also known as diffusion term) and $\boldsymbol{X}_0$ is the initial condition (which can be deterministic or random with a known distribution). The term $\boldsymbol{W}_t$ represents n-dimensional Brownian motion. Note that we use the notation now and in the following $\boldsymbol{A}$ instead of A to represent that it can be a multidimensional process (vector) in $\mathbb{R}^d$. The above equation is equivalent to its so called differential notation form:

$$\begin{cases} d\boldsymbol{X}_t &= \mu\left(t, \boldsymbol{X}_t\right) dt + \sigma\left(t, \boldsymbol{X}_t\right) \boldsymbol{dW}_t, \\ X_0 &= \xi \end{cases} \tag{1.1.2}$$

and a solution is a process $\boldsymbol{X}$ adapted to the filtration ($\boldsymbol{X}_t$ is $\mathcal{F}_t$-measurable for all t). It can be shown that under certain suitability hypothesis a process satisfying the above equation exists (see for example: [55, Chapter 3]). Stochastic differential equations have applications in several fields such as biology, physics, and, as it will be seen through this dissertation, in finance (see for instance [44]). In fact, one of the most well known examples is the case where r and σ are constants, with $\mu\left(t, \boldsymbol{X}_t\right) = r$ and $\sigma\left(t, \boldsymbol{X}_t\right) = \sigma \cdot \boldsymbol{X}_t$, where the process represents the price of an asset under the risk neutral measure, with initial price $\boldsymbol{X}_0$. In this case $\boldsymbol{X}_t$ follows a geometric Brownian motion and we can price a call option obtaining formula 0.0.3. An example on how the paths of an SDE evolve is shown in figure 1.1.

A question arises if we would like to study the process's behaviour running backwards in time. In other words, what happens if instead of having an initial condition, we only have information

about the terminal conditions or the value of the process at time T. In the case of deterministic dynamics given by an Ordinary Differential Equation, or ODE, we simply revert time and we would still have an ODE but in the case of randomness in the system, we need to introduce a new object, the **backwards stochastic differential equation** or BSDE. The theory of BSDEs was firstly developed by Bismut in [7] and then extended to the nonlinear case by Pardoux and Peng in [45].

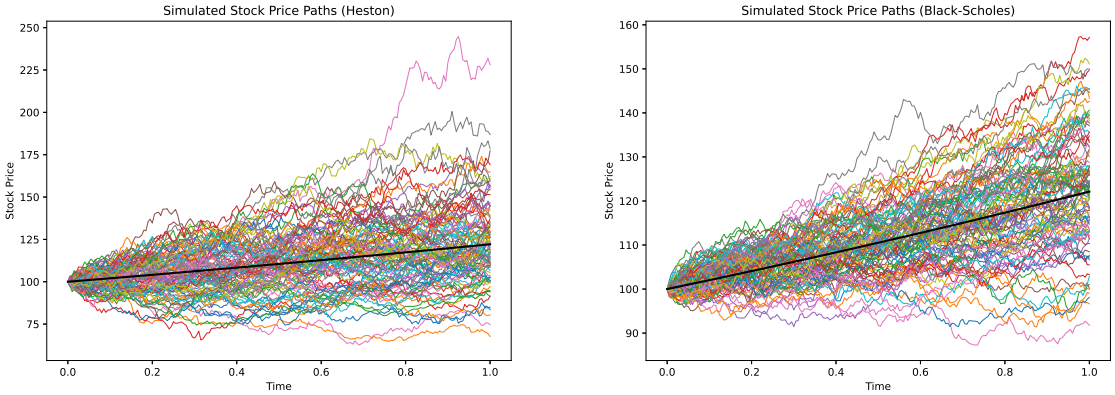


Figure 1.1: Evolution of stock price processes started at $S_0 = 100$ under different diffusion dynamics. On the right we have Black-Scholes dynamics with $\mu(t, X_t) = r, \sigma(t, X_t) = \sigma X_t$ in equation 1.1.1. On the left we have a Heston model where $\mu(t, X_t) = r$ but $\sigma(t, X_t) = \sqrt{V_t} X_t$ being V_t also a stochastic process. The black line represents the evolution of the path without noise ($X_t = e^{rt)} X_0$), and the colored lines represent different realisations of random paths.

To see why this new object is necessary, we study the simplest possible case of such a problem. Let $T < \infty$ be a finite time horizon, and $(\Omega, \{\mathcal{F}\}_t, \mathbb{F}, \mathbb{P})$ be a filtered probability space which supports and n-dimensional Brownian motion $\boldsymbol{W}_t$, where $\mathcal{F}$ is the natural filtration generated by the Brownian motion with assumed right continuity, and completed with $\mathbb{P}$-null sets. We have the following equation:

$$dY_t = 0 \cdot dW_t \qquad Y_T = \xi$$

with ξ $\mathcal{F}_T$ being measurable, we would like its solution to be adapted to the filtration as well. The solution has to be $Y_t = \xi$ but this it not adapted as we would require $\xi \in \mathcal{F}_0$. To avoid this problem we make use of the martingale representation theorem. We recall the definition of a martingale:

Definition 1.1.1 (Martingale.)**.** We say a stochastic process $\boldsymbol{X} : \Omega \times [0, T] \to \mathbb{R}^d$ on a filtered probability space is a martingale if:

1. $\boldsymbol{X}$ is adapted to the filtration
2. $\mathbb{E}[\|\boldsymbol{X}_t\|] < \infty \ \forall t$
3. $\mathbb{E}[\boldsymbol{X}_t \mid \mathcal{F}_s] = X_s$

Theorem 1.1.2 ([55] Theorem 2.5.2)**.** *Let ξ be $\mathcal{F}_T$ measurable in the above probability space. There exists a unique square integrable process (meaning integrable in the $\mathbb{L}^2$ sense) Z_t such that:*

$$\xi = \mathbb{E}[\xi] + \int_0^T Z_s \cdot dW_s \tag{1.1.3}$$

and consequently for any martingale X in the filtration such that $\mathbb{E}\left[|X_T|^2\right] < \infty$ there exists a unique process Z_t such that:

$$X_t = X_0 + \int_0^t Z_s \cdot dW_s \tag{1.1.4}$$

For a $\mathcal{F}_T$- measurable random variable ξ, it can be immediately seen,by the tower property, that the random variable $Y_t = \mathbb{E}[\xi \mid \mathcal{F}_t]$ is a martingale. Hence, by theorem 1.1.2 there exists a unique process Z_t adapted to the filtration such that:

$$\mathbb{E}[\xi \mid \mathcal{F}_t] = \mathbb{E}[\xi \mid \mathcal{F}_0] + \int_0^t Z_s \cdot dW_s \tag{1.1.5}$$

which rearranging leads to:

$$\mathbb{E}\left[\xi \mid \mathcal{F}_t\right] = \mathbb{E}\left[\xi \mid \mathcal{F}_0\right] + \int_0^T Z_s \cdot dW_s - \int_t^T Z_s \cdot dW_s$$
$$Y_t = \xi - \int_t^T Z_s \cdot dW_s$$

using that $\mathbb{E}\left[\xi \mid \mathcal{F}_T\right] = \xi$, by measurably, leading to a BSDE, which in a more general case can have a (possibly random) drift term, hence taking the form:

$$Y_t = \xi + \int_t^T f(s, Y_s, Z_s)ds - \int_t^T Z_s \cdot dW_s \tag{1.1.6}$$

where f is the **generator** and $\boldsymbol{Z}_t$ will be commonly referred to as the **control process**. We write it in differential notation as:

$$\begin{cases} -dY_t &= f(s, Y_t, \boldsymbol{Z}_t)ds - Z_t \cdot dW_t \\ Y_T &= \xi \end{cases}$$

We can extend everything to a multidimensional setting without loss of generality. In the case of BSDEs, a solution is not an individual process verifying the equation but a pair of processes $(Y_t, \boldsymbol{Z}_t)$ verifying equation 1.1.6. The question is if any such pair even exists.

1.2 Existence, uniqueness and FBSDEs.

In this section we show the main results in relation to BSDE solutions and develop BSDEs further to obtain the Forward Backward Stochastic Differential Equation or FBSDE, for short. We begin with some assumptions.

Assumption 1.2.1. The random variable ξ is square integrable, and the generator function is:

1. $\mathcal{F}$-measurable in all variables,
2. uniformly Lipschitz continuous in (y, z) with constant L,
3. $f(0, 0, 0)$ is integrable.

Theorem 1.2.2 ([55] Theorem 4.3.1)**.** *Under the above assumptions the BSDE 1.1.6 has a unique solution.*

Example 1.2.3. We start by showing one of the simplest applications of BSDEs in finance. Consider a financial market with a risky asset whose price, S_t, evolves following a Stochastic Differential Equation (SDE) as in equation 1.1.1, a money market with instantaneous risk free rate r and a contract paying at maturity $g(S_T)$. We want to develop a replicating self-financing trading strategy $(\lambda, h)_t$ where the first entry represents the amount invested in the money market and the second entry the money invested in the risky asset. Then the infinitesimal evolution of the value at time t (V_t) of this trading strategy is: why do you start by showing, there is only this example ????

$$dV_t = \frac{h_t}{S_t}dS_t + \lambda_t r dt = \frac{h_t}{S_t}dS_t + (V_t - h_t)rdt = rV_t dt + h_t\sigma(t, S_t)dW_t$$

with terminal condition $g(S_T)$, in order to replicate the payoff. The calculation of the price of a derivative V_t will represent the numerical examples in following sections and will be referred to as the problem of pricing.

We observe in the above example that the random terminal condition depends on S_T and hence on an SDE. This dependence is formalised in the concept of **FBSDEs**.

Definition 1.2.4 (FBSDE)**.** Let $(\Omega, \{\mathcal{F}\}_t, \mathbb{F}, \mathbb{P})$ be a filtered probability space with a d-dimensional Brownian motion $\{\boldsymbol{dW}_t\}$, ξ a d-dimensional random vector. Let $f : \mathbb{R} \times \mathbb{R}^d \times \mathbb{R} \to \mathbb{R}$, $\mu : \mathbb{R} \times \mathbb{R}^d \to \mathbb{R}^d$, $\sigma : \mathbb{R} \times \mathbb{R}^{d\times d} \to \mathbb{R}^d$ be deterministic functions. Let $\boldsymbol{X}_t$, Y_t and $\boldsymbol{Z}_t$ be square integrable,

adapted stochastic processes. A **decoupled** Forward Backward Stochastic Differential Equation is a system of two stochastic differential equations:

$$\boldsymbol{X}_t = \xi + \int_0^t \mu(s, \boldsymbol{X}_s)ds + \int_0^t \sigma(s, \boldsymbol{X}_s) \cdot \boldsymbol{dW}_s, \tag{1.2.1}$$

$$Y_t = g(\boldsymbol{X}_T) + \int_0^t f(s, \boldsymbol{X}_s, Y_s)ds + \int_0^t \boldsymbol{Z}_t \cdot \boldsymbol{dW}_t \tag{1.2.2}$$

where the first one is called the forward equation (or diffusion) since it starts with an initial condition. The second equation is called backward equation.

It is called decoupled because the forward equation has no dependence on the backward term. This can be extended to add extra dependence in a fully-coupled FBSDE (the term μ in the diffusion depends on Y_t). However, this lies outside our scope of study. Additionally, we have assumed that Y_t is a 1-dimensional real valued random variable, in order to better fit our purpose of pricing derivatives, but this is without lost of generality.

A solution to the above FBSDE is a triple $(\boldsymbol{X}_t, Y_t, \boldsymbol{Z}_t)$ of processes where $\boldsymbol{X}$ is a solution to the SDE, and $(Y_t, \boldsymbol{Z}_t)$ is a pair solving the BSDE. We can change the setting slightly with an initial condition at some other time point $\boldsymbol{X}_t = \xi$ for $t \in (0, T)$.

Assumption 1.2.5. In addition of assuming that f, σ, μ, g are deterministic functions, we may assume:

1. $\mu(\cdot, 0), \sigma(\cdot, 0), g(0), f(\cdot, 0, 0, 0)$ are bounded
2. μ, σ, f, g are uniformly Lipschitz continuous in space variables with Lipschitz constant L
3. μ, σ, f are uniformly Hölder-$\frac{1}{2}$ continuous in t with constant L

By theorem 1.2.2 it follows that the FBSDE has a unique solution under the above assumptions. The last assumption is only required for convergence results that will be used in chapter 3.

1.3 Generalised Feynman-Kac formula

One of the most well known results in the theory of SDE is their connections with PDEs through the Feynman-Kac formula, in the 1-dimensional version:

Theorem 1.3.1 ([56] Theorem 16.)**.** *Let $u(t, x) \in C^{1,2}([0, T], \mathbb{R})$ with the bounded derivative in the second entry, satisfying the PDE:*

$$u_t(t, x) + \mu(t, x)u_x(t, x) + \frac{1}{2}\sigma(t, x)^2 u_{xx}(t, x) - r(t, x)u(t, x) = 0 \quad \forall (t, x) \in [0, T) \times \mathbb{R}$$

with terminal condition $u(T, x) = g(x)$, with r being a non-negative function. Then u has a representation:

$$u(t, x) = \mathbb{E}\left[e^{-\int_t^T r(s, X_s^{t,x})ds} g(X_T^{t,x})\right]$$

where X satisfies the SDE:

$$dX_s^{t,x} = \mu(t, X_s^{t,x})ds + \sigma(s, X_s^{t,x})dW_s$$

where the notation $X^{t,x}$ means the stochastic process started at time t in $X_t = x$.

It would be interesting then to have a similar connection in the case of BSDEs in a generalised version of the Feynman-Kac theorem. Generally, solving a (F)BSDE is not a straightforward task. Therefore, having this connection allows us to transform a PDE problem into a stochastic calculus one, and vice-versa, allowing the application of a wider range of techniques to identify the solution. It is in our particular interest to have this connection since it would allow us to apply any of the numerical techniques in further chapters to a PDE via its identification with the appropriate BSDE, transferring high dimensional problems.

We define the semi-linear parabolic PDE:

$$\begin{cases} \partial_t u(t,x) + \mathcal{L}_t u(t,x) + f(t,x,u(t,x),\nabla_x u(t,x)\sigma(t,x)) = 0 \\ u(T,x) = g(x) \end{cases} \tag{1.3.1}$$

where $\mathcal{L}_t$ is the differential operator:

$$\mathcal{L}_t u = \sum_{i=1}^{d} \mu_i(t,x)\partial_{x_i} u + \sum_{i,j}^{d} \sigma_{i,j}(t,x)\partial_{x_i,x_j} u$$

Definition 1.3.2. Let $u \in C^{1,2}([0,T]\times\mathbb{R}^d)$. We say u is a classical solution of the PDE 1.3.1 if for all $(t,x) \in [0,T)\times\mathbb{R}^d$, u satisfies the interior condition.

Theorem 1.3.3 (Generalised Feynman-Kac formula. [46] Theorem 4.)**.** *Assume $u \in C^{1,2}([0,T]\times\mathbb{R}^d,\mathbb{R})$ is a classical solution of the partial differential equation 1.3.1 Then:*

$$Y_s^{t,x} = u(t, X_s^{t,x}) \qquad Z_s^{t,x} = \nabla_X u(t, X_s^{t,x}) \tag{1.3.2}$$

where the process $(\boldsymbol{X}_s^{t,x}, Y_s^{t,x}, \boldsymbol{Z}_s^{t,x})$ is the unique solution of the FBSDE 1.2.1.

Proof. We simplify notation and only see the 1-dimensional case. By Itô's lemma (using differential notation) we have that:

$$\begin{aligned} du(t,X_t) &= \partial_t u(t,X_t)dt + \partial_x u(t,X_t)dX_t + \frac{1}{2}\partial_{xx}u(t,X_t)dX_t dX_t \\ &= \partial_t u(t,X_t)dt + \partial_x u(t,X_t)(\mu(t,X_t)dt + \sigma(t,X_t)dW_t) + \frac{1}{2}\partial_{xx}u(t,X_t)\sigma(t,X_t)^2 dW_t dW_t \\ &= \left[\partial_t u(t,X_t)dt + \partial_x u(t,X_t)\mu(t,X_t) + \frac{1}{2}\partial_{xx}u(t,X_t)\sigma(t,X_t)^2 dW_t dW_t\right]dt + \partial_x u(t,X_t)\sigma(t,X_t)dW_t \\ &= f(t,X_t,Y_t,\nabla_x u(t,X_t)\sigma(t,X_t))dt + \partial_x u(t,X_t)\sigma(t,X_t)dW_t \end{aligned}$$

where the last equality comes from the PDE solution. Compare to the backwards equation in 1.1.6 and we get that it is a solution if:

$$Z_t = \partial_x u(t,X_t)\sigma(t,X_t)$$

The terminal condition of the BSDE is satisfied by the PDE boundary condition. □

Remark 1.3.4. This result adds some additional information. Noticing that $\nabla_X u$ is in fact the delta of the valuation process, we are obtaining a connection of the control process Z_t and the hedging strategy for a derivative. This relation will prove particularly useful in some of the numerical techniques to be presented.

It naturally follows to investigate when the opposite case will hold, namely, when a solution to the FBSDE represents a solution to the PDE. In case of regularity, as in the conditions of u defined in theorem 1.3.3, we have the opposite implication under assumptions 2.2.1. In a more general setting we will have a weak form solution also known as the viscosity solution. This can be thought of as having a function that behaves slightly differently from a $C^{1,2}$ function which is a classical solution. To avoid delving into very technical details which may require a lengthy development, we restrain to this intuitive notion of viscosity solutions, and we refer to [55] for further details. This is of our interest because through the entirety of this thesis we are solving the Black Scholes PDE which is equation 1.3.1 with $f(t,x,u(t,x),\nabla_x u(t,x)\sigma(t,x)) = -r\cdot u(t,x)$, and the boundary condition given by the payoff function.

Chapter 2

Deep Learning

A crucial part of the techniques used in this project is the use of Deep Learning Models. In this chapter we aim to give the formal background required to develop further ideas. We follow [27] and [28]. In section 2.1 we give the mathematical definition of a (Feed-forward) Neural Network. After that, in section 2.2 we will provide the theoretical results that explain why Neural Networks are a good choice for the purposes we have. Finally, we explain in section 2.3, 2.3 how do Neural networks learn by training them with data.

In general, we can think of a deep learning problem as designing a function $\mathcal{N} : \mathbb{R}^N \to \mathbb{R}^M$ that turns inputs $(x_1, \ldots, x_N)$ into outputs $(y_1, \ldots, y_M)$ in an optimal way which is to be defined. What we will be doing is trying to learn an unknown (possibly non-linear) function f from some available data. A first requirement to being able to learn this function is having $\mathcal{N}$ to live in a sufficiently rich functional space where we can search. To make this a problem that we can computationally tackle, we normally choose $\mathcal{N}$ to be determined by a set of parameters $\theta \in \Theta$ and we try to find the optimal parameters (this is further explained in [28]). For this purpose neural networks are a very helpful resource.

We can see neural networks as a composition of parameterised functions. Therefore, a key concept in a Neural Network is understanding how the information flow of values move through this composition, or internal structure of the network. This inner structure is called **network architecture**, and there are several of them depending on the particular task to be achieved. Although there are general, well known, architectures such as Recurrent Neural Networks, or Long-Short Term Memory Networks, it is necessary to have a good understanding of the problem in order to choose the correct architecture. This could, for example, include using a net to reduce dimensionality to a latent space and another part to bring it back to the original dimension (*auto-encoder* [28, Chapter 14]), or generating new data (*GAN* [28, Chapter 20]) or classification tasks.

2.1 Neural Networks

In the architectures we will be using, we will first focus on the simplest kind of neural network as a building block: The Feed-forward Neural Network (FNN). It is in essence a function created by composing trivial affine functions with non-linear activation functions. A formal definition will be:

Definition 2.1.1 (Feed-forward Neural Network.[27], Def. 2.1)**.** Let $N, M, r \in \mathbb{N}$. A function $\mathcal{N} : \mathbb{R}^N \to \mathbb{R}^M$ is a Feed-forward Neural Network with $r-1$ hidden layers and with $d_i \in \mathbb{N}$ units in the $i-$th hidden layer for $i = 1, \ldots, r-1$, together with the activation functions $\sigma_i : \mathbb{R}^{d_i} \to \mathbb{R}^{d_i}$, where $d_i = M$, if $\mathcal{N}$ is given by the composition:

$$\mathcal{N} = \sigma_r \circ L_r \circ \cdots \circ \sigma_1 \circ L_1 \tag{2.1.1}$$

where the functions $L_i : \mathbb{R}^{d_{i-1}} \to \mathbb{R}^{d_i}$ are affine for any $i = 1, \ldots, r$:

$$L_i(x) := W^i \boldsymbol{x} + \boldsymbol{b}^i, \ x \in \mathbb{R}^{d_{i-1}} \tag{2.1.2}$$

The matrix $W^i = \left[W^i_{j,k}\right]_{j=1,\ldots,d_i;k=1,\ldots,d_{i-1}} \in \mathbb{R}^{d_i \times d_{i-1}}$ is the weight matrix, and the bias vector is $\boldsymbol{b}_i \in \mathbb{R}^{d_i}$ with $d_0 := N$.

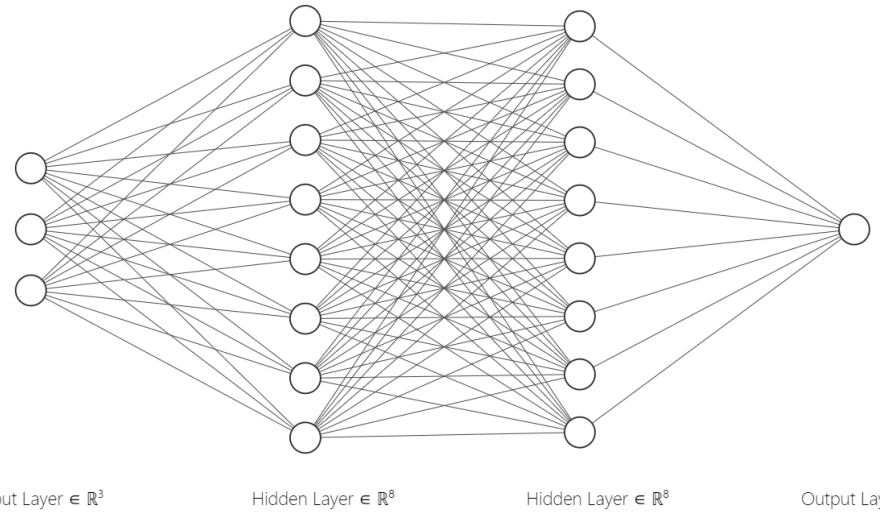


Figure 2.1: Feed-forward Neural Network with 3 hidden layers taking inputs in $\mathbb{R}^5$ and giving a real value. The architecture is $r = 3, d_0 = 5, d_1 = d_2 = 8$ and the output has $d_3 = 1$. The vertex represent the activation function of the network and the lines represent the weights. The image is done using the NN-SVG package from [38].

Definition 2.1.2. The set of all functional defined by the above structure is:

$$\mathcal{N}_r(N, d_1, \dots, d_{r-1}, M; \sigma_1, \dots, \sigma_r)$$

It is common to have activation functions that act component wise, $\sigma(\boldsymbol{x}) = (g(x_1), \dots, \sigma(x_N))$. In this case we will simply use g instead of σ.
Two key concepts to take from the above definition are the parameters, made by the weights and the biases of the network, as well as the hyperparameters which are the dimensions of each vector space in the network. Knowing both of them, the network $\mathcal{N}$ is completely defined and the difference between parameters and hyperparameters is that parameters are learnt, whereas hyperparameters have to be chosen in advance. The **depth** of the network is given by the number of hidden layers, and the dimensionality of each layer is called the **width**.

The following simple lemma will be helpful to compare the complexity of networks in the following chapters:

Lemma 2.1.3. *[27] Proposition 2.4 The number of parameters of* $\mathcal{N} \in \mathcal{N}_r(N, d_1, \dots, d_{r-1}, M; \sigma_1, \dots, \sigma_r)$ *assuming the activation functions do not involve additional parameters is:*

$$\sum_{i=1}^{r} (d_{i-1} + 1) d_i \tag{2.1.3}$$

An additional interesting property of the FNN is that they preserve some differentiability and continuity properties of, at least, the least differentiable of their activation functions. This result is reassuring, given that we will be approximating derivatives of functions in our work, we expect some differentiability properties. The result is trivially true given that we are using the composition of affine functions which are infinitely differentiable with the activation functions. We formalise this in the following:

Lemma 2.1.4 ([27] Proposition 2.6)**.** *Let* $\mathcal{N}$ *be a Feed-forward Neural Network and let its activation function* $\sigma_i \in C^{m_i}(\mathbb{R}^{d_i})$ *for some* $m_i \in \mathbb{N} \cup \{0\} \cup \{\infty\}$ *for any* $i = 1, \dots, r$ *then it holds:*

$$\mathcal{N} \in C^{\min(m_i)}(\mathbb{R}^N, \mathbb{R}^M)$$

A fundamental aspect of the neural network architecture, as demonstrated earlier, lies in the selection of appropriate activation functions, a variety of which can be found in the literature,

with each presenting distinct advantages and drawbacks, necessitating careful considerations when committing to a choice. A comprehensive summary of these activation functions can be found in [54]. For the present study we will solely focus on mentioning activation functions deployed in this paper: **identity**, **ReLU**, and **ELU**.

The identity is the trivial function $Id(x) = x$ which is then C^∞. For our purposes we will use it in the output layer to allow for negative values. Even though its differentiability properties are desirable, it would not be beneficial to use it in every layer, as it will then not make any difference with respect to an affine function. Moreover the derivative of the function is constant 1 which can be detrimental in the learning process as it adds no information to gradient descent.

The Rectified Linear Unit (ReLU), defined as $\sigma(x) = (x)^+$, is a widely adopted activation function in deep learning due to its simplicity and ability to facilitate fast convergence [25]. Despite these advantages, ReLU suffers from a well-known issue referred to as the 'dead-ReLU problem'. This occurs when a layer in the network outputs solely negative values, which, after passing through the ReLU layer, are transformed to 0 in every entry, effectively causing the gradient-based training algorithm to freeze.
To address this limitation, we turn to another activation function, the exponential linear unit or ELU, proposed by Clevert in [19], and defined as:

$$\sigma(x) = \alpha(e^x - 1)\mathbb{1}_{x<0} + x\mathbb{1}_{x\geq 0}$$

which is C^1 if we chose $\alpha = 1$. The ELU and ReLU functions can be seen in 2.2.

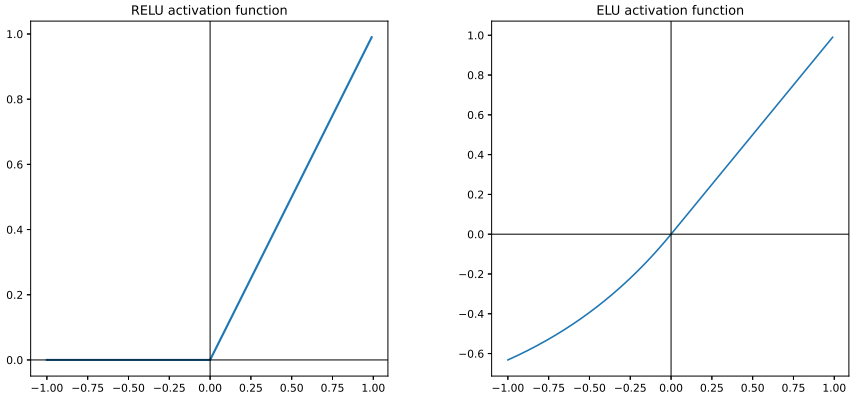


Figure 2.2: ReLU VS ELU activation function. The non-negativity of ReLU has often lead to dead gradients in the training of some methods of section 3.

2.2 Universal approximation

Deep learning became popular under the expectation that our neural networks has the capability to represent the objective function. In this matter there is a reassuring theoretical result that states that every good enough function can be approximated by a neural networks.Furthermore, this paper will show this this can be extended to operators. For the moment we give the theorem for real valued function.

Assumption 2.2.1. Let σ verify:

1. is not polynomial function,
2. is bounded on any finite interval,
3. the closure of the sets of all discontinuity points of σ in $\mathbb{R}$ has zero Lebesgue measure

We immediately observe that the activation functions we have shown verify these assumptions.

Theorem 2.2.2 (Universal approximation theorem for functions. Theorem 1[39])**.** *Let $\sigma : \mathbb{R} \to \mathbb{R}$ be a measurable function which verifies the assumptions 2.2.1, $K \subset \mathbb{R}^N$ a compact set and $\epsilon > 0$. Then:*

1. *For any function $f \in C(K, \mathbb{R})$, there exists $d \in \mathbb{N}$ and $\mathcal{N} \in \mathcal{N}_2(N, d, 1; \sigma, Id)$ such that:*

$$\sup_{x \in K} |f(\boldsymbol{x}) - \mathcal{N}(\boldsymbol{x})| < \epsilon$$

2. *For any $p \geq 1$ and any measurable function $f : K \to \mathbb{R}$ (assuming it is measurable under the $L^P(K)$ norm) there exists $d' \in \mathbb{N}$ and $h \in \mathcal{N}_2(N, d', 1; \sigma, Id$ such that:*

$$\left(\int_K |f(\boldsymbol{x}) - \mathcal{N}(\boldsymbol{x})|^p d\boldsymbol{x} \right)^{1/p} < \epsilon$$

In practice, the result has limited practical applicability as it does not provide the specific network structure required for a given problem, and most utilized architectures differ from those proposed by the theorem. The hope is that this generalises for deeper networks. Furthermore, this result does not ensure the convergence of solutions, as challenges such as numerical noise, local minima, or poorly conditioned problems may impede convergence. Consequently, this theorem merely serves as a theoretical reassurance that approximating functions using neural networks is indeed possible.

2.3 Loss function and Neural network training.

Having identified the objective (learning a function) and the mathematical object to be employed for its attainment, the next step involves contemplating the methodology. A fundamental aspect of Deep Learning lies in evaluating the quality of the approximation function with respect to the true function. This assessment is not inherent in the network's structure but rather found by a loss function, denoted as $\mathcal{L} : \mathbb{R}^M \times \mathbb{R}^M \to \mathbb{R}$. In our settings we will not have information about the true value of the output of the function we want to approximate for a given set of inputs. Therefore we have an unsupervised learning problem and desire to minimise the risk:

$$\mathbb{E}\left[l(\mathcal{N}(\boldsymbol{x}))\right]$$

which will be computed over a sample batch B. Given a set of weights and biases by definition 2.1.1, it is clear that our function is completely determined. Hence, we can write the empirical loss function as a function of the parameters θ:

$$l(\theta) = \frac{1}{\#B} \sum_{i \in B} l(\mathcal{N}(\boldsymbol{x}), \boldsymbol{y})$$

where B is a set of samples or *batch*.

Remark 2.3.1. We will normally use θ to represent the parameters of a neural network, and Θ for the space this parameters are in, which is immediately dependent on the context.

After formulating the loss function, the next crucial step involves devising a method to obtain a minimizer. We provide a concise overview of the most common techniques employed in machine learning for this purpose.

A well known method is gradient descent which updates the parameters θ for a batch B as:

$$\theta_{\text{new}} = \theta_{\text{old}} - \eta \cdot \nabla_\theta(B)$$

where η is a hyperparameter called **learning rate** and ∇ is the gradient of the loss function with respect to the parameters of the neural network.

However, computing the gradient with respect to a whole data set can be costly and can lead to over-fitting. To prevent this from happening the common choice is stochastic gradient descent (SGD) which only does the training over random subsets of the training set called mini-batches. Essentially we will split the data into disjoint subsets, $B = \bigsqcup_{i=1} B_i$ and do the iterations as explained in algorithm 1. Each loop in step 2 of the algorithm is call an**epoch**.

Algorithm 1 Stochastic Gradient Descent

Require: Learning rate η, number of epochs T,
Require: Training data $(x^{(1)}), (x^{(2)}), \ldots, (x^{(n)})$
Initialize model parameters $\theta \in \Theta$
for $t = 1$ to T **do**
  Randomly split training data into n mini-batches B_i.
  for $i = 1$ to n **do**
    Compute the gradient: $\nabla_\theta \mathcal{L}(\theta \mid B_i)$
    Update the parameters: $\theta \leftarrow \theta - \eta \cdot \nabla_\theta \mathcal{L}(\theta \mid B_i)$
  end for
end for
return Learned model parameters θ

The correct behavior of the algorithm is dependent on both the selection of an appropriate set of initial parameters and a suitable learning rate. Choosing an excessively large learning rate may avoid convergence, while an excessively small one may significantly delay the convergence. Additionally, very big scale differences between the initial parameter of the network may lead to local minima avoiding convergence.

To address these challenges, we opt to initialise our weights using the Xavier Initialiser from [24], which employs a random normal distribution with zero mean and variance, determined by the layer's hyperparameters. The randomness avoids symmetric behaviour within the network that can lead to problems in training, and it avoids the weights being too small or too big.

Regarding the learning rate, several interesting approaches are available:
- piece-wise constant function with powers of the form 10^{-i},
- exponential decay function, and
- Adam optimiser (proposed in [35]) as seen in algorithm 2.

Algorithm 2 Adam Optimizer

Require: Learning rate α, β_1, β_2, number of iterations T
Require: Training data $(x^{(1)}), (x^{(2)}), \ldots, (x^{(n)})$
Initialise model parameters θ
Initialise first moment variables: $m \leftarrow 0$
Initialise second moment variables: $v \leftarrow 0$
Initialise time step: $t \leftarrow 0$
for $t = 1$ to T **do**
  Randomly shuffle the training data
  for $i = 1$ to n **do**
    Compute gradient: $grad \leftarrow \nabla_\theta \mathcal{L}(\theta \mid B_i)$
    Update biased first moment estimate: $m \leftarrow \beta_1 \cdot m + (1 - \beta_1) \cdot grad$
    Update biased second moment estimate: $v \leftarrow \beta_2 \cdot v + (1 - \beta_2) \cdot grad \odot grad$
    Correct bias in first moment: $\hat{m} \leftarrow \frac{m}{1-\beta_1^t}$
    Correct bias in second moment: $\hat{v} \leftarrow \frac{v}{1-\beta_2^t}$
    Update parameters: $\theta \leftarrow \theta - \alpha \cdot \frac{\hat{m}}{\sqrt{\hat{v}}+\epsilon}$
  end for
end for
return Learned model parameters θ

The computation of the gradients is done automatically in machine learning. For this purpose we used *TensorFlow*[1] in our code.

Chapter 3

Solving BSDEs with Neural Networks

In this core chapter chapter of the thesis, we explore the details on the numerical schemes to solve BSDEs using machine learning. To this end we start section 3.1 with an in-depth review on the existing literature and applications, providing a comprehensive context for the subsequent discussions. In section 3.2 we explain the schemes that use one network at each time step in the discretisation of the system, both in a forward (subsection 3.2.1) and backward (3.2.2) manner. A third, novel approach, is proposed combining both in order to reduce the error pathwise. After that, we focus on architectures that share the network through time in section 3.3, which will be building block for the next chapter.

3.1 A review on the literature.

There is a considerable literature on deep learning techniques to solve high dimensional problems focusing on (semi-linear elliptic) PDEs and BSDEs. There are different approaches, but in general we could classify them in those using the PDE structure directly, namely Physics Informed Neural Networks (PINN), and those using the BSDE formulation. A summary of the different techniques can be found in [9].

The PINN method relies on minimising the discrepancy of the neural network approximation from verifying the PDE in the inner domain and in its boundary conditions. In [49] and [48] Raissi et al. developed the method and discussed data selection in the context of Burger's, or Navier-Stokes equations.

The BSDE approach is introduced by Han, Jentzen and E in [31] and [20]. They complement the work in further research towards the boundaries of its errors given in [32]. Their method formulates the PDE as a BSDE, use a discretisation of the time under an Euler scheme and then solves it by approximating the control process at each time step with a Feed-forward Neural Network. The results they obtained are very accurate approximations, although it is somewhat obscure whether said methods lead to over-fitting given the high number of parameters and high flexibility of the network. An additional drawback of this method is that it does not return $u(t,x)$ and $Du(t,x)$ (the solution function and its gradient) for t,x outside of the trained trajectories, or trajectories starting from a different initial condition. The final, and main, downside of this method is that they simulate the BSDE in a forward manner and use the terminal condition only as part of the loss function. This does not allow for early exercise features which require a backward simulation, and, in a purely mathematical sense, it breaks the backwards nature of the BSDE.

The problem of early exercise features is addressed in several papers. The first one, which is a natural continuation of Han's et al. work, can be found in paper[52]. In this paper the authors run the BSDE approximation starting on terminal conditions and try to make all paths converge to the same initial value, as Y_0 should be a function of X_0 exclusively, in a Markovian framework. This is done in the context of pricing Bermudan Swaption under a LIBOR Market Model, which creates a high dimensional problem. The authors in [40] claim that in the Bermudan example, there is a bias obtained by overestimating the continuation. They proposed a solution by combining the BSDE approach with the traditional AMC regression to obtain better approximations for the continuation

value. As the authors show, the combination of the two methods removes the bias at the high price of computational efficiency. In their examples, the authors found that for European options their method is on average 6 times slower than forward Monte Carlo methods and 4-5 times slower in American options, compared to AMC. Liang et al. tested the method using computers and high performance servers and concluded that their method is only convenient when memory overflow occurs. In their setting the authors use this happens exclusively when running on a CPU in more than 20 dimensions.

Some error boundaries, of what we will call the Backwards BSDE Solver, are shown in [21] where they prove that the variance at the initial time upperbounds the error. In [37] the authors essentially use a version of American Monte Carlo where they avoid a set of basis functions and use a neural network instead. Very recently in [30] Guo et al. proposed using Doob's decomposition to approximate the American option price and avoid the bias. They consider networks to approximate the martingale term and super-martingale term in the decomposition.

An orthogonal approach to using neural networks to approximate the pricing function can be found in [3], [4], [5] and [29], where they construct the exercise boundary through a neural network instead of the price itself.

Mixed approaches combining ideas from PINN and BSDE are found in [43], [47] and [18] in the case of American options. Essentially, they approximate the pricing function with a neural network instead of the control process but then combine part of the PDE and the BSDE in their loss functions.

In the present study, our attention shall be directed towards methodologies that operate directly with the BSDE without an extra layer of complexity.

3.2 One net at each time

The first approach we follow is approximating the the control processes in BSDE 1.1.6 with a neural network.

3.2.1 A Forward Scheme

For the first method we follow the ideas proposed in [31] and [20]. We start with a decoupled BSDE (possibly a PDE formulated as a BSDE) as in equation 1.2.1 which lives in a filtered probability space $(\Omega, \{\mathcal{F}\}_t, \mathbb{F}, \mathbb{P})$. We are looking for a solution which is a triple $\{(\boldsymbol{X}_t, Y_t, \boldsymbol{Z}_t)\}_{t\in[0,T]}$ verifying 1.2.1. We recall that $\boldsymbol{X}, \boldsymbol{Z}$ are $d-$dimensional stochastic processes and $Y_t \in \mathbb{R}$ is a $1-$dimensional process. As mentioned before $Y, \boldsymbol{Z}$ are functions of X and the following relations holds:

$$Y_t = u(t, \boldsymbol{X}_t) \qquad \boldsymbol{Z}_t = \sigma^T(t, \boldsymbol{X}_t)\nabla u(t, \boldsymbol{X}_t) \tag{3.2.1}$$

We are interested in the computation of $u(t, \boldsymbol{X}_t)$ via a neural network. As explained in chapter 2 we need to decide the architecture and then a loss, which will require training data for the learning process. We first explain the architecture.
Consider an Euler scheme for the forward dynamics:

$$\begin{aligned} \boldsymbol{X}_{t_{n+1}} &= \boldsymbol{X}_{t_n} + \mu(\boldsymbol{X}_{t_n}, t_n)\Delta t_n + \sigma(\boldsymbol{X}_{t_n}, t_n)\cdot \boldsymbol{dW}_{t_n} \\ \boldsymbol{X}_{t_0} &= \boldsymbol{X}_0 \end{aligned}$$

where $n = 0, 1, \ldots, N-1$, t_n is the time at step n over a equidistant partition of $[0, T]$, and $\boldsymbol{X}_0$ is the initial value of the forward process. The Euler scheme can also be applied to the backwards process:

$$-dY = f(t, \boldsymbol{X}_t, Y_t, \boldsymbol{Z}_t)dt - \boldsymbol{Z}_t \cdot \boldsymbol{dW_t}$$

leading to:

$$Y_{t_n} - Y_{t_{n+1}} = f(t_n, \boldsymbol{X}_{t_n}, Y_{t_n}, \boldsymbol{Z}_{t_n})\Delta t_n - \boldsymbol{Z}_{t_n} \cdot \boldsymbol{dW_{t_n}}$$

which can be rewritten as:

$$Y_{t_{n+1}} = Y_{t_n} - f(t_n, \boldsymbol{X}_{t_n}, Y_{t_n}, \boldsymbol{Z}_{t_n})\Delta t_n + \boldsymbol{Z}_{t_n} \cdot \boldsymbol{dW_{t_n}} \tag{3.2.2}$$

Contrary to the Euler scheme, in the forward part we do not know the initial condition Y_0 but rather a terminal condition $g(\boldsymbol{X}_T)$ (coming from the payoff function of a financial derivative). We

also do not know the values of $\boldsymbol{Z}_t$ at the step times and here is precisely where we use the power of neural networks.

The initial condition $\hat{Y}_0$ and $\hat{\boldsymbol{Z}}_0$ are treated as parameters that can be learnt in our model and the control process at each time step t_n, $\boldsymbol{Z}_{t_n}$ is parameterised in [31] by a neural network using the relation 3.2.1:

$$\boldsymbol{Z}_{t_n} = (\sigma^T \nabla u)(t, \boldsymbol{X}_{t_n}) \approx \widehat{\sigma^T \nabla u}_{t_n}(\theta \mid \boldsymbol{X}_{t_n}) \tag{3.2.3}$$

As learnable parameters, $\hat{Y}_0$ and $\hat{\boldsymbol{Z}}_0$ are randomly initialised at the start of training. In our experiments we found that this seemingly random selection of initial values requires careful consideration, as it is a source of high instability. We have found convergence issues can arise, even for simple products like European put options, if the initial choice is too far from the right solution. To ensure a satisfactory initial guess, we propose two approaches.

The first approach (specially in a practitioners side) involves using the last Mark-to-Market (MtM) value obtained. The second performing a Monte Carlo forward simulation to obtain the discounted payoff. The initial value $\boldsymbol{Z}_0$ does not exhibit problematic behavior and does not require any special fit.

Additionally, in the work [31], the authors briefly mention an extension to handle the case where $X_0 = \eta$ with η being a random variable. We have attempted to implement this extension by replacing the random initial values $\hat{Y}_0$ and $\hat{\boldsymbol{Z}}_0$ with an additional FNN $\mathcal{N}(\eta)$. However, we prefer not to study this approach at this time, as it can be formulated as a parametric approach, hence contained in the last chapter framework.

In our work have introduced a subtle modification by using the prior structure and then splitting the volatility term from the gradient in the neural network. We expect this to have some advantages. In terms of implementation, the learning of the volatility at each moment in time would not be embedded inside the control process, thereby allowing us to work with more general volatility structures. Therefore, at each time step our network is:

$$\boldsymbol{Z}_{t_n} = (\sigma^T \nabla u)(t, \boldsymbol{X}_{t_n}) \approx (\sigma^T(t, \boldsymbol{X}_{t_n}) \widehat{\nabla u_{t_n}}(\theta \mid \boldsymbol{X}_{t_n})). \tag{3.2.4}$$

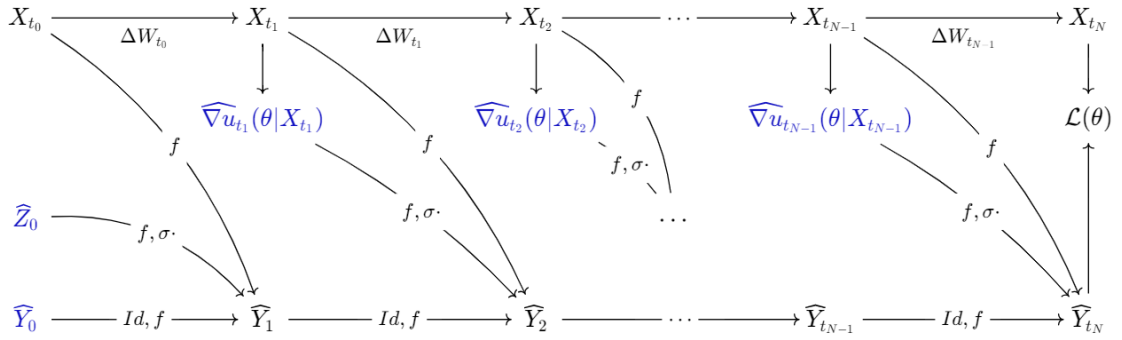


Figure 3.1: Data Flow of the Forward BSDE Architecture: In blue the learnable parts of the high level architecture. The arrows represent flows of information and which functions the variables go through for the next step according to equation 3.2.2. At the end we compare terminal values in the loss function.

Hence we can think of the above scheme as a neural network operator:

$$\mathcal{N} : \Theta \times \mathbb{R}^{d \times N} \times \mathbb{R}^{d \times N} \to \mathbb{R}$$

where $\hat{Y}_0, \hat{\boldsymbol{Z}}_0$ are included in Θ (the space of parameters), which runs all the iterations of the Euler scheme in 3.2.2 requiring the whole path trajectory of the forward SDE and Brownians as input data. The high level architecture of the network is represented in figure 3.1.By the BSDE structure we have that:

$$Y_{t_N} = g(\boldsymbol{X}_{t_N})$$

If it were the case that, at each time, the sub networks were exactly the gradient, $\hat{Y}_0 = Y_0$ and $\hat{\boldsymbol{Z}}_0 = \boldsymbol{Z}_0$and we ignore the discretisation error of the Euler scheme, we would have:

$$\widehat{Y}_{t_N} = \mathcal{N}\left(\boldsymbol{\theta} \mid \{\boldsymbol{X}_i\}_{i=t_0}^{t_N=T}, \{\boldsymbol{dW_i}\}_{i=t_0}^{t_N=T}\right) = g(\boldsymbol{X}_{t_N})$$

Thus it makes sense to consider the loss:

$$\mathcal{L}(\theta) = \mathbb{E}\left[\left(\mathcal{N}\left(\boldsymbol{\theta} \mid \{\boldsymbol{X}_i\}_{i=t_0}^{t_N=T}, \{\boldsymbol{dW_i}\}_{i=t_0}^{t_N=T}\right) - g(\boldsymbol{X}_{t_N})\right)^2\right] \tag{3.2.5}$$

In practice this expectation will be computed by M Monte Carlo sampling paths. Then optimization algorithms are used to get θ. Normally the Adam optimiser is used. The whole training algorithm can be found in algorithm 3. Each $Y_{t_i}^j$ in the algorithm corresponds to $u(t_i, \boldsymbol{X}_{t_i}^j)$ so we can obtain values for realised paths, but for a given pair $t, \boldsymbol{X}_t$, that is not in a simulated path we cannot compute $u(t, \boldsymbol{X}_t)$.

The main advantage of this method is that, after training, we obtain an estimation of the initial Y_0 as well as the gradients. Financially the gradients are the deltas, necessary for hedging purposes, which will need re-evaluation if we were using other Monte Carlo techniques.

Algorithm 3 Deep Forward BSDE Solver

Require: Time grid $t_i, i = 1, \ldots, N$, initial $\boldsymbol{X}_0$, M number of sample paths, A = number of training steps.

1: Simulate data: $\{\boldsymbol{X}_i\}_{i=t_0}^{t_N=T}$ keepBrownians $\{\boldsymbol{dW_i}\}_{i=t_0}^{t_N=T}$
2: Initialize $\mathcal{N}$ parameters $\theta \in \Theta$
3: **for** $a = 1$ to A **do**
4: **for** $i = 0$ to N, $j = 1$ to M **do**
5: $Y_{t_{i+1}}^j = Y_{t_i}^j - f(t_i, \boldsymbol{X}_{t_i}^j, Y_{t_i}^j, \boldsymbol{Z}_{t_i}^j)\Delta t_i + (\sigma(t, \boldsymbol{X}_{t_i}^j)^T \widehat{\nabla u}(\theta \mid t_i, \boldsymbol{X}_{t_i}^j)) \cdot \boldsymbol{dW_{t_i}^j}$
6: **end for**
7: Compute $\mathcal{L}(\theta) = \sum_{j=1}^M \left(\widehat{Y}_{t_N}^j - g(\boldsymbol{X}_{t_N}^j)\right)^2$
8: Update parameters (Adam)
9: **end for**
10: **return** $\widehat{Y}_0, \widehat{\boldsymbol{Z}}_0$

Remark 3.2.1. We are trying to give a general framework where the generator f can take any form. In pricing problems with a unique risk free interest rate and no valuation adjustments the generator will be:

$$f(t, \boldsymbol{x}, y, \boldsymbol{z}) = -r_t \cdot y$$

being r_t the risk free interest rate.

The convergence of the method is justified by the following results which we state in a slightly more general setting than the one above allowing extra dependence on Y, $\widehat{\boldsymbol{Z}}_t = \mathcal{N}(\boldsymbol{X}_t, Y_t)$.

Theorem 3.2.2 ([32], Theorem 1)**.** *Let π be a equidistant partition of the interval $[0,T]$, $\hat{\boldsymbol{X}}^\pi$, $\hat{\boldsymbol{Z}}^\pi$, $\hat{Y}^\pi$ represent the approximated solutions to the BSDE by the above scheme, and $\hat{\boldsymbol{X}}$, $\hat{\boldsymbol{Z}}$, $\hat{Y}$ represent the true solution. Under the assumptions 1.2.5 together with linear growth in the functions $\widehat{\boldsymbol{Z}}_t(\boldsymbol{X}_t, Y_t)$ there exists a constant C independent of the partition norm and d such that:*

$$\sup_{t\in[0,T]}\left(\mathbb{E}\left[|\boldsymbol{X}_t - \hat{\boldsymbol{X}}_t^\pi|^2\right] + \mathbb{E}\left[|Y_t - \hat{Y}_t^\pi|^2\right]\right) + \int_0^T \mathbb{E}\left[|\boldsymbol{Z}_t - \hat{\boldsymbol{Z}}_t^\pi|^2\right] dt \leq C\left(\|\pi\| + \mathbb{E}\left[|g(\boldsymbol{X}_T^\pi) - \hat{Y}_T^\pi|^2\right]\right)$$

for sufficiently small $\|\pi\|$, where $\hat{A}_t^\pi = \hat{A}_{t_i}^\pi$ for $t \in [t_i, t_{i+1})$ $A \in \{X, Y, Z\}$.

Theorem 3.2.2 proves an upper bound on the simulation error by the chosen loss function in our learning process. Theorem 3.2.3 provides an upper bound of the loss function with the approximation capabilities of the functional space. This indicates that if this space is rich enough, such as in Deep Neural Networks, we can have a small loss function, and together with the previous theorem, the approximated solution of the FBSDE should be close to the actual solution.

Theorem 3.2.3 ([32], Theorem 2)**.** *Under the same assumptions there exists a constant C independent of $\|\pi\|$ and d such that:*

$$\inf_{\theta\in\Theta} \mathbb{E}\left[|Y_T^\pi - g(\hat{\boldsymbol{X}}_t^\pi)|^2\right] \leq C\left(\inf_{\theta\in\Theta}\left\{\mathbb{E}\left[\left|Y_0 - \widehat{Y}_0\right|^2\right] + \sum_{i=0}^{N-1}\mathbb{E}\left[\left|\mathbb{E}\left[\widetilde{\boldsymbol{Z}}_{t_i} \mid \hat{\boldsymbol{X}}_t^\pi\right] - \hat{\boldsymbol{Z}}_t^\pi\right|^2\right]\|\pi\|\right\}\right)$$

for $\|\pi\|$ small, where $\widetilde{\boldsymbol{Z}}_{t_i} = \|\pi\|^{-1}\mathbb{E}\left[\int_{t_i}^{t_{i+1}} \boldsymbol{Z_t} dt \mid \mathcal{F}_{t_i}\right]$

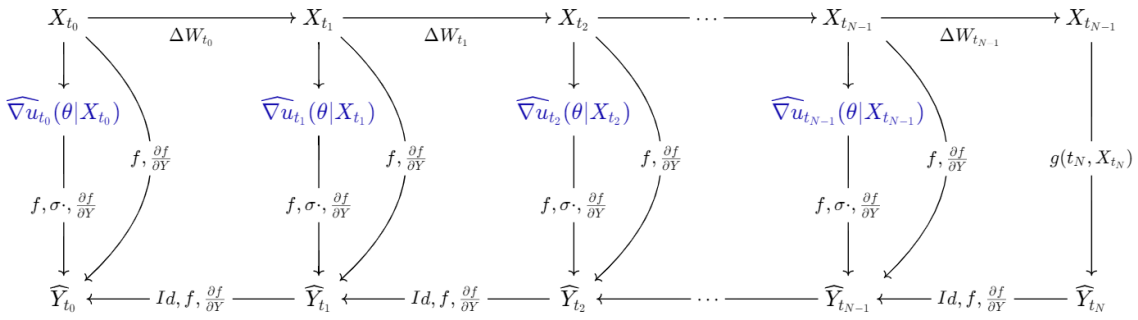


Figure 3.2: Data Flow of the Backward BSDE Architecture: In blue the learnable parts of the high level architecture. The arrows represent flows of information and which functions the variables go through for the next step according to equation 3.2.7. At time 0 we compute the variance of all paths to obtain the loss.

Remark 3.2.4. The above theorems has been proven under a much more general set of assumptions in the cited papers, including fully coupled systems, but we have only shown the result as adapted for our purposes.

3.2.2 A Backward Scheme

The previous method is nonetheless unable to handle early exercise features in financial applications such as Bermudan options, as we need information about prices in future, unseen, times. In a more mathematical sense it is also interesting to use the original backwards structure of the BSDE to solve it starting from terminal condition as it will have a better fit close to maturity. The solution proposed in [52] is running the BSDE backwards. From equation 3.2.2 we can rewrite:

$$Y_{t_n} = Y_{t_{n+1}} + f(t_n, \boldsymbol{X}_{t_n}, Y_{t_n}, \boldsymbol{Z}_{t_n})\Delta t_n - \boldsymbol{Z}_{t_n} \cdot \boldsymbol{dW_{t_n}} \tag{3.2.6}$$

Contrary to the previous case we have an implicit equation so we cannot run the scheme directly. As a brief note this difficulty is not found in the set up of [52] or [51] as they work directly with the discounted portfolio value, hence a martingale, so we have no generator function which makes the equation implicit. To handle the case with non-zero generator function f we use the idea of expanding f as a 1st order Taylor polynomial from [40]. In particular by remark 3.2.1 we are not loosing any information as the 1st order expansion is the exact solution, in the problem of pricing. We then have the Euler scheme going as:

$$Y_{t_n} \approx Y_{t_{n+1}} + \frac{f(t_n, \boldsymbol{X}_{t_n}, Y_{t_n}, \boldsymbol{Z}_{t_n})\Delta t_n - \boldsymbol{Z}_{t_n} \cdot \boldsymbol{dW_{t_n}}}{1 - \frac{\partial f}{\partial Y}\Delta t_n} \tag{3.2.7}$$

We can simulate M forward paths of the state variables and then run the scheme backwards to get a value of the initial price Y_{t_0} which has a representation $u(0, X_{t_0})$. Therefore we expect that all these paths, when run backwards towards a initial condition, converge to the same value, as they only depend on the initial condition of $\boldsymbol{X}_0$, which is shared along samples. As previously this requires omitting possible errors due to the discretization. Similarly to the previous section, the flow of information inside the network is shown in figure 3.2. The network uses as input entire path trajectories of the SDE forward process and its Brownians. We refer to algorithm 4 for a more detailed explanation. Again, we have only represent the gradient by a Neural network instead of the whole control process.

All paths should converge to Y_{t_0} giving:

$$\widehat{Y}_0 = \mathbb{E}\left[\mathcal{N}\left(\boldsymbol{\theta} \mid \{\boldsymbol{X}_i\}_{i=t_0}^{t_N=T}, \{\boldsymbol{dW_i}\}_{i=t_0}^{t_N=T}\right)\right] \tag{3.2.8}$$

Then it naturally comes to consider a loss function measuring how deviated our samples are from the true value, which is precisely the variance:

$$\mathcal{L}(\theta) = \text{Variance}\left(\mathcal{N}\left(\boldsymbol{\theta} \mid \{\boldsymbol{X}_i\}_{i=t_0}^{t_N=T}, \{\boldsymbol{dW_i}\}_{i=t_0}^{t_N=T}\right)\right) \tag{3.2.9}$$

Algorithm 4 Deep Backward BSDE Solver

Require: Time grid $t_i, i = 1, \dots, N$, FBSDE, Initial $\boldsymbol{X}_0$, M numple sample paths, A = number of training steps.

Simulate data: $\{\boldsymbol{X}_i\}_{i=t_0}^{t_N=T}$ keep Brownians $\{\boldsymbol{dW_i}\}_{i=t_0}^{t_N=T}$
Initialize $\mathcal{N}$ parameters $\theta \in \Theta$
for $a = 1$ to A **do**
  for $i = N$ to 0, $j = 1$ to M **do**

$$Y^j_{t_{i-1}} = \frac{Y^j_{t_i} - (\sigma^T(t, X^j_{t_i})\widehat{\nabla u}(\theta|t_i, X^j_{t_i})) \cdot \boldsymbol{dW^j_{t_i}}}{1 + r(t_{i+1} - t_i)}$$

  end for
  Compute $\mathcal{L}(\theta) = \text{Var}\left(\widehat{Y}^j_{t_0}\right) = \frac{1}{M}\sum_{j=1}^{M}\left(\widehat{Y}^j_{t_0} - \bar{Y}_{t_0}\right)^2$
  Use training method (Adam)
end for
return $\widehat{Y}_0$

Similarly to the forward case, there is a theoretical justification of the backward method. In [21], the authors upper bound the error committed when approximating by a neural network scheme. The first theorem proves that the expected error in the numerical approximation of the stochastic process solution to the FBSDE can be bound by the variance at initial time. The second one proves that the loss (variance) can be small if the functional space we take our network from is rich enough.

Theorem 3.2.5 ([21], Theorem 3.6)**.** *Let π be an equidistant partition of the interval $[0, T]$, and $\hat{\boldsymbol{X}}^\pi$, $\hat{\boldsymbol{Z}}^\pi$, $\hat{Y}^\pi$ represent the approximated solutions to the BSDE by the above scheme. Under the assumptions 1.2.5 where the Lipschitz constant verifies $K < \frac{1}{6T^2}$, a constant C exists independently from the time grid norm and d , such that:*

$$\sup_{t\in[0,T]} \mathbb{E}\left[|Y_t - \hat{Y}^\pi_t|^2\right] + \int_0^T \mathbb{E}\left[|\boldsymbol{Z}_t - \hat{\boldsymbol{Z}}^\pi_t|^2\right] dt \leq C\left(\|\pi\| + Var(\hat{Y}_{t_0})\right)$$

for sufficiently small $\|\pi\|$, where $\hat{A}^\pi_t = \hat{A}^\pi_{t_i}$ for $t \in [t_i, t_{i+1})$ $A \in \{X, Y, Z\}$.

Theorem 3.2.6 ([21], Theorem 3.8)**.** *Let π be a equidistant partition of the interval $[0, T]$ and $\hat{\boldsymbol{X}}^\pi$, $\hat{\boldsymbol{Z}}^\pi$, $\hat{Y}^\pi$ represent the approximated solutions to the BSDE by the above scheme. By theorem 1.3.3 we know that $\boldsymbol{Z}_t = \nabla_x u(t, \boldsymbol{X}_t)\sigma(t, \boldsymbol{X}_t)$. Under the assumptions 1.2.5 where the Lipschitz constant verifies $K < \frac{1}{6T^2}$, a constant C exits independently from the time grid norm and d such that:*

$$Var(\hat{Y}_{t_0}) \leq C\left[\|\pi\| + \sum_{0\leq i\leq n-1} \mathbb{E}\left\{|\phi(X^\pi_{t_i}) - Z^\pi_{t_i}|\right\}\right]$$

for sufficiently small $\|\pi\|$, where $\phi(t, x) := \nabla_x u(t, x)\sigma(t, x)$.

3.2.3 Hybrid approach

In relation to the two methods above there are some doubts on which one has better properties when both are available. An important point in the decision is that the forward method has a great amount of flexibility in the initial value, leading to very small relative errors as shown in table 3.1. The reason is that it is a parameter of the model. The model will first fit the initial value and then it will start adjusting the gradients to match the payoff at terminal time. Hence we expect that as numerical noise accumulates, specially over long time horizons, we will still get a good fit in the initial time, but get an increasing error when we are closer to maturity. Quite the opposite effect happens with the backward method. The fit at terminal time is perfect, indeed it is the condition that the method starts running from. Therefore, there is no numerical noise at maturity. As we run backwards in time the numerical noise will accumulate and we will try to fit the model to estimate the true initial condition.

One of the objectives of this work was to investigate whether the advantages of both methods can be combined to mitigate their drawbacks in a new approach. Their negative effects are less relevant in examples where we are calculating the expected exposures, and the averaging cancels some error, but are very relevant if we want to work pathwise (solving the BSDE for a particular realisation of a Brownian motion and evolution of the forward process). This effect is seen in figure 3.3.

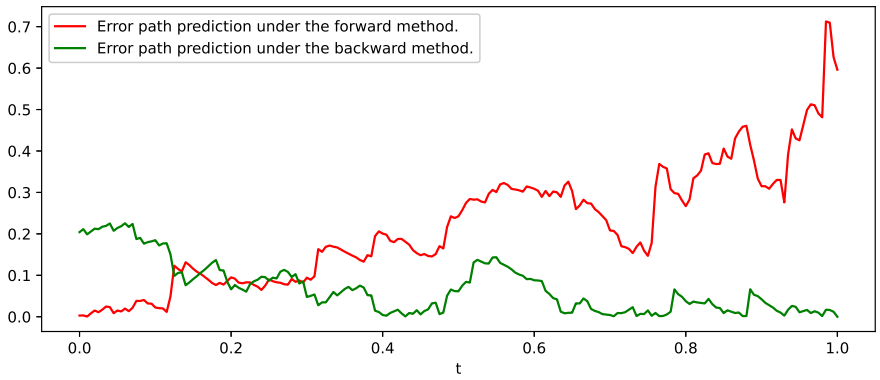


Figure 3.3: Error of simulated path under the forward and backward method, compared with the true path value. The backward method has a perfect fit at maturity and high error at $t = 0$. The forward method has perfect fit at $t = 0$ and high error at maturity.

To give a quick intuition of the above, let's work with a problem where the generator of the BSDE is zero, having no drift. Then our approximation methods at each time step are:

$$\widehat{Y}_{t_i} = \widehat{Y}_{t_0} + \sum_{j=0}^{i-1} \sigma^T(t_j, \boldsymbol{X}_{t_j}) \widehat{\nabla u_{t_n}}(\theta \mid \boldsymbol{X}_{t_j}) \boldsymbol{dW}_j$$

for the forward method, and:

$$\widehat{Y}_{t_i} = g(\boldsymbol{X}_{t_N}) - \sum_{j=i}^{N-1} \sigma^T(t_j, \boldsymbol{X}_{t_j}) \widehat{\nabla u_{t_n}}(\theta \mid \boldsymbol{X}_{t_j}) \boldsymbol{dW}_j$$

for the backward method. Under a perfect fit of the network and the true solution of the FBSDE, it is straightforward to see that they should be the same for every t_i. Based on the explanations given above on how the accumulation of noise affects each method, we try to use the value:

$$\widehat{Y}_{t_i} = \frac{t_i \left(g(\boldsymbol{X})_{t_N} - \sum_{j=i}^{N-1} \sigma^T(t_j, \boldsymbol{X}_{t_j}) \widehat{\nabla u_{t_n}}(\theta \mid \boldsymbol{X}_{t_j}) \boldsymbol{dW}_j \right)}{T} + \frac{(T - t_i) \left(\widehat{Y}_{t_0} + \sum_{j=0}^{i-1} \sigma^T(t_j, \boldsymbol{X}_{t_j}) \widehat{\nabla u_{t_n}}(\theta \mid \boldsymbol{X}_{t_j}) \boldsymbol{dW}_j \right)}{T}$$

The weights for each method are selected to be higher for times closer to the perfect fit, $t = 0$ for the forward method and $t = T$ for the backward method. We observe in the experiments, that it reduces the average of the error in pathwise calculations.

3.2.4 Sub-network architectures

The choice of network architecture plays a crucial role in approximating each of the deltas and significantly impacts the quality of the results. Therefore, we discuss the some possible choices. The architecture proposed in the original papers follows a consistent pattern using four dense layers: one d-dimensional input layer, two (20+d)-dimensional hidden layers, and a one-dimensional output layer. Each of these layers have no bias, employ the ReLU activation function, and incorporate batch normalisation.

Working on the ideas presented in [14], we conducted experiments to explore various approaches. Similarly to the finding of the authors, we have concluded that batch normalisation is unnecessary and can be effectively substituted by using the ELU activation function, which also addresses the well-known ReLU problem.

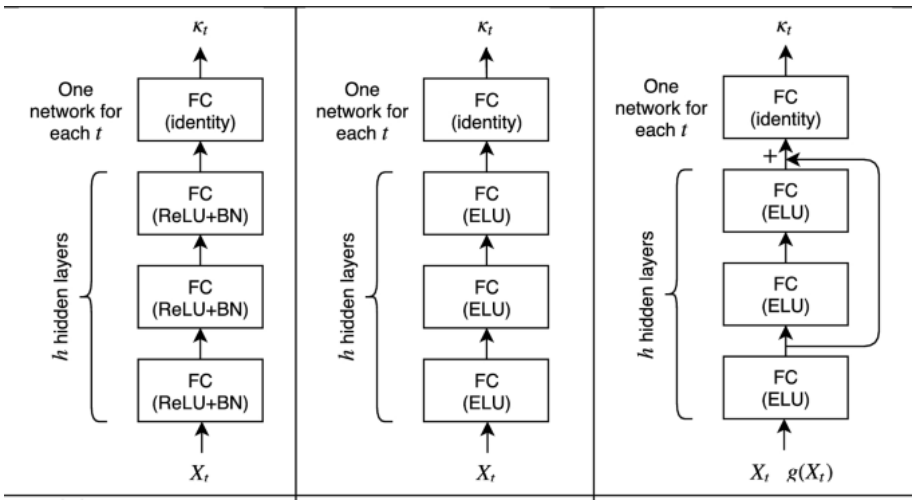


Figure 3.4: Variation of sub-network architectures. From left to right: architecture with batch normalization and ReLU activation function, no batch normalization and ELU activation function, architecture with a shortcut and adding payoff as a control variate. The output κ_t of each network represents the control process at time t. The figure is from [14].

Additionally, we have investigated the potential benefits of implementing shortcuts within the sub-networks. However, our experiments revealed that this approach is less efficient unless larger numbers of hidden layers are used. While this may lead to improved results, it comes at the cost of increased computational complexity, which does not seem justified by the small improvement.

A further addition that is expected to increase stability is adding the payoff $g(t_n, X_{t_n})$ or $Y_{t_{i-1}}$ as an input of this networks. The first architecture has been called residual and the second one ResNet by its resemblance to the ResNet architecture ([33]).

3.2.5 Numerical tests with one network at a time.

We perform some experiments using the above mentioned techniques. We have focused on the problem of pricing and the problem of calculating exposures. The problem of pricing consists of calculating the price, at time t, of a derivative contract paying $g(\boldsymbol{X}_T)$ at maturity, which we denote by V_t. As explained in the first chapter of this thesis, in example 1.2.3, pricing problems can be formulated as a PDE or a BSDE.

We have tested the models for a European call, which pays $(X_T - K)^+$ at maturity, and an arithmetic basket-call with payoff:

$$\left(\sum_{i=1}^{n} w_i \cdot S_T^i - K\right)^+ \tag{3.2.10}$$

under 10-D and 100-D with equal weights. The risk factor processes are uncorrelated Black-Scholes models with parameters $r = 0.10$ and $\sigma = 0.30$, $\boldsymbol{X}_0 = 100.$ across all dimensions and the strike is 100. The true prices are 16.7341, 10.2243, and 9.5222 (the last two values are obtained via Monte Carlo over $1M$ paths). The results are shown in table 3.1 where we observe errors of around 0.05% at time 0 for the methods. It is important to remark that the computational time does not increase from 1D to 10D but does with the 100D call. This is explained by the increase in magnitude on the number of parameters of the network as we go from two layers with 21 neurons (for 1D) or 30 (for 10D) to 120. By lemma 2.1.3 we observe an increment from $200 \times 483 = 9.66e^4$ to $200 \times (14600) = 2.92e^6$ in the number of parameters which is an increase of two (2) orders of magnitude. We should highlight that the loss represents different concepts in the methods. In the backward model it is the dispersion with respect the true value (hence the error we expect

to make when approximating initial value), while in the other three methods it is the error with respect to the terminal condition. Hence, only the losses in the forward, Residual and ResNet are comparable. It is difficult to infer any general properties from the observed data. ResNet is the slowest method and keeps the same magnitude of relative error through all dimensions. In low dimensions, 1 and 10, the Residual architecture has a lower loss than the forward method for a similar computational time, but this effect disappears in high dimensions. Given the similarity in the results of the methods we will use the forward architecture in our experiments.
All networks have been trained under the same training regime of 7000 iterations with training batch of 128 samples and Adam optimiser. The loss is calculated over a validation set of 1024 samples. All these methods have been trained using the same CPU so we believe the time taken is a comparable quantity.

Architecture	Dimension	Learning time (mm:ss)	Price ($)	Rel.error (%) at $t = 0$	Loss
Forward	1	09:08	16.7350	0.0054	3.5541
Backward	1	08:08	16.7461	0.0710	2.5869
ResNet	1	11:47	16.7299	0.0250	2.0023
Residual	1	08:58	16.7440	0.0591	2.3109
Forward	10	10:22	10.2292	0.0489	4.1288
Backward	10	10:35	10.2225	0.0176	3.1982
ResNet	10	15:34	10.2300	0.0557	4.6411
Residual	10	12:09	10.2291	0.0469	3.7133
Forward	100	38:47	9.5068	0.1616	2.9523
Backward	100	38:04	9.4904	0.3337	2.8987
ResNet	100	48:22	9.5209	0.0136	2.8106
Residual	100	39:03	9.5050	0.1809	4.024

Table 3.1: Pricing results under a Black Scholes for One Net at a time architectures for $r = 0.10$, $\sigma = 0.30$, at the money option, strike 100.

The main advantage of the proposed techniques is not just the ability to price at time 0, but the ability to replicate price paths in general. Keeping the advantages in mind, we can make use of the applications proposed in [26] to calculate expected exposures. This is a key part in the calculation of CVA and DVA as expressed in formulas 0.0.1 and 0.0.2. Their intuitive necessity is that they explain how much money we can loose if there is a default event in the future. Therefore, calculating exposures requires knowing the distribution of the price at future times. For this purpose we generate paths with trained neural networks and then compute the **discounted positive** and **negative exposure**:

$$\text{DEPE}(s) := \mathbb{E}\left[e^{-r(s-t)}(V_s)^+ \mid \mathcal{F}_t\right] \quad \text{DENE}(s) := \mathbb{E}\left[e^{-r(s-t)}(V_s)^- \mid \mathcal{F}_t\right]$$

To compute the expectation in our experiments, we are using 5000 outer Monte Carlo paths, computing the mean at each point. In the case of European (basket) call options, the optionality eliminates the negative exposure and we simply obtained discounted prices. This allows us to benchmark the results for the basket call option for which the discounted exposure is constant to the initial price. In figure 3.5 we show the training evolution for the forward and backward method for the 1-D call together with the calculated exposures. We observe an identical exposure profile for both methods with a good fit at time 0 and a higher error closer to maturity. Given their similarity, in figure 3.6 we show a more detailed insight on the error for the exposures of a 100-D basket call calculated only under the forward method.

Although the exposures calculated through the two methods are identical, we see different properties if we work for individual paths. In figure 3.7 we show the average absolute error and standard deviation obtained by approximating the price path of a 1-D call with different methods. We observe that the proposed hybrid method provides a better approximation pathwise compared to the other methods.

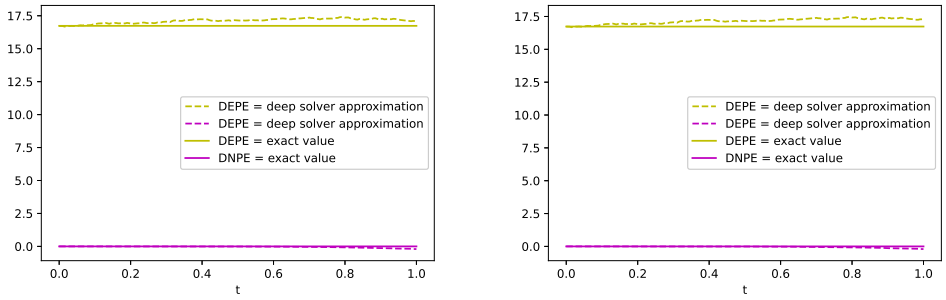


Figure 3.5: Comparison of the forward and backward method for pricing a call option. In the left the exposures are calculated using the forward BSDE solver and in the right with the backward BSDE solver.

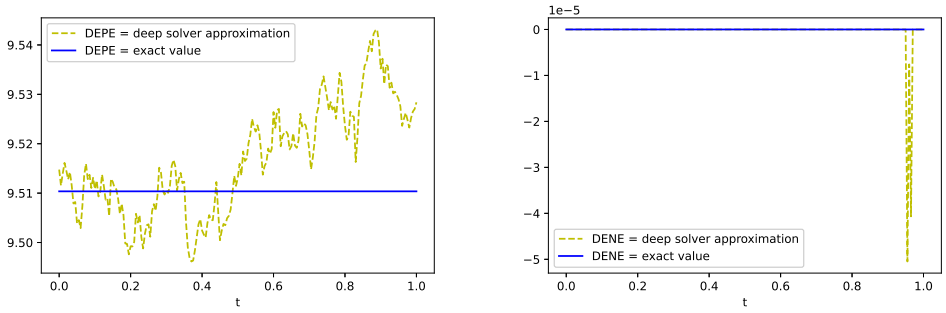


Figure 3.6: DEPE and DENE of a 100-D basket call calculated under the forward BSDE solver

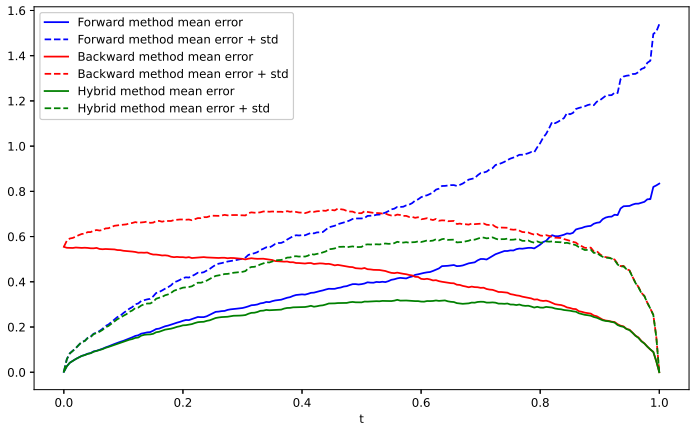


Figure 3.7: Mean of absolute error of pathwise approximation under different Deep Solvers.

3.3 Common network

In the previous section we have focus on techniques that use a different sub-neural network at each time step to approximate the control process. Although this can lead to good results, it does create some complications. Most notably it drastically increases the number of parameters to learn during training, if we attempt to have a finer time grid. This is challenging when dealing for example with a long maturity T. Secondly we cannot calculate the value of $u(t, X_t)$ for any particular pair of pair of inputs $(t, \boldsymbol{X}_t)$. Finally we would expect that given all the functions are approximating the delta at close time points the should have similar weights which is not necessarily guarantee by having many different networks. By sharing the same network through the whole time grid,

this is achieved.

We are then looking for approximations done by a single neural network independently of the time grid. In this architectures it is important to avoid using batch normalization in the networks. This is because $\boldsymbol{X}_t$ will not be stationary. Hence, the normalisation is expected to change through time (in our case with the risk free rate).

3.3.1 Approximate $u(t, X_t)$, Raissi's Method.

The first approach will be to approximate by a single Feed-forward neural network the valuation function $u(t, X_t)$. While training we would like to use the formulation we have developed about the pricing problem as a BSDE. The solution to this is given in the work of Raissi [47]. In the paper the author follows the ideas in PINN method where the solution function of a PDE is approximated by a neural network, but he extends to a loss function that keeps track of the BSDE formulation. With the unknown function approximated by a neural network, we can compute the gradient (delta) by automatic differentiation. We will not delve into the details of this, as this will be done automatically via the code from Machine Learning libraries. For a reference about automatic differentiation see [2].
In particular in that paper, Raissi formulates the approach to work for coupled FBSDE by simulating the steps in the Euler forward scheme at the same time that the loss function is computed.

Let $\mathcal{N} : \Theta \times \mathbb{R}^+ \times \mathbb{R}^d \to \mathbb{R}$ be the neural network, where $\theta \in \Theta$ represents the weights (and biases if we use them), $t \in \mathbb{R}^+$ is a positive time instant, and $\mathbb{R}^d$ represents the state variables. By using the discretization of the system in equation 3.2.2, we require $\widehat{Y}_t := \mathcal{N}(\theta \mid t, \boldsymbol{X}_t)$ to satisfy the BSDE:

$$\widehat{Y}_{t_{n+1}} \approx \widehat{Y}_{t_n} + f(t_n, \boldsymbol{X}_n, \widehat{Y}_{t_n}, \widehat{\boldsymbol{Z}}_{t_n})\Delta t_n + \sigma^T(t_n, \boldsymbol{X}_{t_n})\widehat{\boldsymbol{Z}}_{t_n} \cdot \boldsymbol{dW}_{t_n} \tag{3.3.1}$$

for every step t_n $n = 0, \ldots, N-1$, where $\widehat{\boldsymbol{Z}}_{t_n} = \nabla_X \mathcal{N}(\theta \mid t, \boldsymbol{X}_t)$. Additionally at terminal time we would like the function to match the payoff:

$$g(T, \boldsymbol{X}_T) = \mathcal{N}(\theta \mid T, \boldsymbol{X}_T) \tag{3.3.2}$$

Then the loss function will be the accumulation of errors over all the time steps:

$$\begin{aligned}\mathcal{L}(\theta) = \mathbb{E}\Bigg[\alpha_1 \cdot \sum_{n=0}^{N-1} \Big(Y_{t_{n+1}} - Y_{t_n} - f(t_n, \boldsymbol{X}_{t_n}, \widehat{Y}_{t_n}, \widehat{\boldsymbol{Z}}_{t_n})\Delta t_n - \sigma^T(t_n, \boldsymbol{X}_{t_n})\widehat{\boldsymbol{Z}}_{t_n} \cdot \boldsymbol{dW}_{t_n}\Big)^2 \\ + \alpha_2 \cdot \left(Y_{t_N} - g\left(t_N, \boldsymbol{X}_{t_N}\right)\right)^2\Bigg]\end{aligned}$$

where the expectation is computed, as before, by simulating M paths. We show the method in algorithm 5.

In our experiments we have added the α_i terms, not found in the literature we explored. Their rationale lies in the understanding of the algorithm's intuition. At a first run the loss for a random initialization gives a very small error for the term multiplying α_1 (see table 3.2). This term is called **inner loss** as it is the loss associated with the BSDE and the inner domain. The weights of the neural network, by the Glorot Initialiser, will be on average zero. Consequently, we expect the inner error to be small, because the function will be close to the 0 function. On the other hand, the **external loss** is very high as shown in table 3.2, given we have not learned any function. During the training the algorithm learns the function and the balance of errors changes. Intuitively the network learns a local behaviour and it is only through the external loss that it learns the global behaviour of the function. In this setting we have tested different combinations of the α_i parameters and decided to stay with $\alpha_i = 1$. The reason is that, although a slightly higher value of α_1 has shown improvements in the convergence of some particular tests, we have not found a general rule for the choice.

We found that one of the main struggles in the algorithm is the scale difference between time and state variables. On average at time t the state variables values the algorithm will see, as part of the training, will be $e^{\int_0^t r(s)ds}\boldsymbol{X}_t$ with their spread controlled by the standard deviation, hence the volatility. It is noticeable then, that there is a big difference when comparing with the time

Algorithm 5 Deep Raissi BSDE Solver

Require: Time grid $t_i, i = 1, \dots, N$, initial $\boldsymbol{X}_0$, M number of sample paths, A = number of training steps.

Simulate data: $\{\boldsymbol{X}_i\}_{i=t_0}^{t_N=T}$ keep Brownians $\{\boldsymbol{dW_i}\}_{i=t_0}^{t_N=T}$
Initialize $\mathcal{N}$ parameters $\theta \in \Theta$
for $a = 1$ to A **do**
  $\mathcal{L} = 0$
  $Y^j_{t_i}, Z^j_{t_i} = \mathcal{N}(\theta \mid t_{i+1}, X^j_{t_{i+1}}), \nabla_X \mathcal{N}(\theta \mid t_{i+1}, X^j_{t_{i+1}})$
  for $j = 1$ to M **do**
    for $i = 0$ to $N - 1$ **do**
      $Y^j_{t_{i+1}} = \mathcal{N}(\theta \mid t_{i+1}, X^j_{t_{i+1}})$
      $Z^j_{t_{i+1}} = \nabla_X \mathcal{N}(\theta \mid t_{i+1}, X^j_{t_{i+1}})$
      $\mathcal{L} = \mathcal{L} + \left(Y^j_{t_{i+1}} - Y^j_{t_i} + f(t_i, \boldsymbol{X}^j_{t_i}, Y^j_{t_i}, \boldsymbol{Z}^j_{t_i})\Delta t_i - (\sigma^T(t, \boldsymbol{X}^j_{t_i}) Z^j_{t_i} \cdot \boldsymbol{dW^j_{t_i}}\right)^2$
      $Y^j_{t_i}, Z^j_{t_i} = Y^j_{t_{i+1}}, Z^j_{t_{i+1}}$
    end for
    $\mathcal{L} = \mathcal{L} + \left(Y^j_{t_N} - g\left(t_N, \boldsymbol{X}^j_{t_N}\right)\right)^2$
  end for
  Update parameters (Adam)
end for

grid where values go from 0 to 1. To solve this inconvenience we have re-scale the state variable space using moneyness instead of spot price. Although neural networks are expected to eventually converge to the right result, the moneyness trick accelerates the process and can potentially avoid falling in local minima. In [47] this problem does not take place because the scale of the state variables (which has no drift) is already in the order of magnitude of the units.

In [47] the author additionally imposes a first order condition in the boundary trying to match the gradients:

$$\nabla_X g(T, \boldsymbol{X}_T) = \nabla_X \mathcal{N}(\theta \mid T, \boldsymbol{X}_T)$$

In the setting of pricing derivatives it is somewhat obscure if this is an appropriate addition to the loss. The discontinuity can lead to strange behaviour, by the different smoothness properties. In fact in the numerical example we will observe that it is at the point of discontinuity where most of the error concentrates. This does not happen in the followed literature where all the terminal conditions are given by smooth functions.

3.3.2 Modelling a shared gradient.

The final approach of this chapter is the one formulated in [14]. It approximates the gradient as in the first two methods but, instead of using one different network at each time step, we simplify it by sharing the same network through all time steps: $\mathcal{N} : \Theta \times \mathbb{R}^+ \times \mathbb{R}^d \to \mathbb{R}$. This is justified because the valuation function, and hence its gradient, have the same form for the entire space-time domain. Hence, we only have one neural network and the initial condition for $\widehat{Y}_0$ to learn.
The numerical algorithm is identical to 3 with the only difference that in the 5th step, the gradient is calculated through a share network and that time is not simply an index of which network should be use. The data flow of the network hence should be as in figure 3.8.

Due to the greater generality of Raissi's method which fits our purposes better we will not show numerical experiments on this approach.

3.3.3 Numerical tests on Raissi's method.

As previously explained, using the proposed techniques for option pricing problems present some complications. In this section we start by presenting the example of pricing a one dimensional ATM option, with X_0 and strike 100, using the Raissi method. The dynamics of the underlying model are a Black Scholes model with volatility $\sigma = 25\%$ and interest rate $r = 10\%$. .The correct price at time 0 is 14.9757. We show the results with different width and depth. The compared architectures are FNN with:

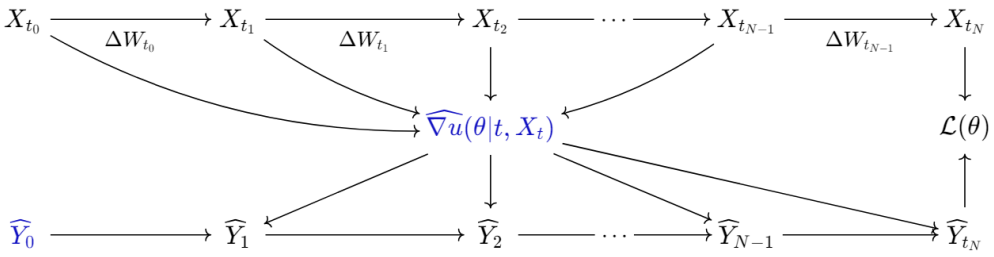


Figure 3.8: Data Flow BSDE shared gradient Architecture: only one network has to be learnt apart from the initial condition. At the end we compare terminal values in the loss function.

1. 4 hidden layers with 20 neurons each
2. 4 hidden layers with 20 neurons each + ELU in the last layer,
3. 6 hidden layers with 20 neurons each,
4. 4 hidden layers with 50 neurons each.

By lemma 2.1.3 the numbers of parameters are: 1321 for the first two architectures, 2161 for the third one and 7711 for the last one. This is a substantial reduction in the number of parameters compared with the previous section architectures.

It is important to justify the usage of the ELU function in the last layer of the network in one of the architectures. We observe in figure 3.9 that the approximated value (in green) goes under zero for a set of paths, which is not realistic as the call option should never have negative price. We have tried to floor the approximation with this extra activation function in the last layer, while keeping differentiability (which we would have lost if using ReLU).

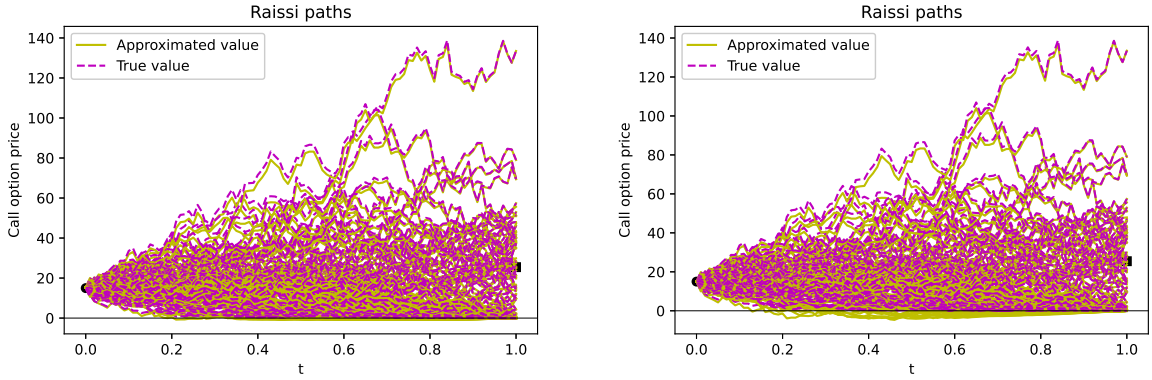


Figure 3.9: In the left the paths approximated adding a ELU activation function in the last layer. In the right the same architecture whiteout ELU activation in the last layer. Both have 4 hidden layers with 20 neurons each.

In table 3.2 we show the training process of different network structures. A more general overview of the training is shown in figure 3.10.

The results show that the addition of the extra ELU function should not be used since it creates a bias in the whole approximation as is shown in the results.

As observed in table 3.2 all architectures give a very accurate initial price. This is not surprising given that the algorithm trains over the pair $t = 0$, $X_0 = 100$. for every pair, so a total of 36000×100 times. In our research we are interested in the approximation error that the method will make for a general (t, X_t). Therefore we study the evolution of the error through time. This is shown in figure 3.11 for $r = 10\%$ and $\sigma = 25\%$ and trained over 100 time steps. For the sake of readability other settings are presented in the appendix. Through all this figures we show the true value minus the approximation, without squaring or taking absolute value, so we discern between over-

Architecture	Iteration	Inner Loss	External loss	Gradient loss	$\widehat{Y}_0$	Time
$[20, 20, 20, 20]$	1	0.1864	1.453×10^5	151.7346	0.6024	$0s$
$[20, 20, 20, 20]$	500	223.6275	69.2435	5.1101	14.8941	$64s$
$[20, 20, 20, 20]$	5000	122.8540	12.2719	2.8769	14.9958	$193s$
$[20, 20, 20, 20]$	10000	122.2420	6.2596	2.4374	14.875	$336s$
$[20, 20, 20, 20]$	20000	119.5698	5.9679	2.2888	14.9491	$625s$
$[20, 20, 20, 20]$	30000	118.8617	6.8222	2.3145	14.9644	$911s$
$[20, 20, 20, 20]$+ELU	1	0.0120	1.8831×10^5	148.0486	-0.1626	$0s$
$[20, 20, 20, 20]$+ELU	500	248.4136	106.6012	6.9701	15.6972	$74s$
$[20, 20, 20, 20]$+ELU	5000	125.1190	16.3481	3.6207	15.3853	$193s$
$[20, 20, 20, 20]$+ELU	10000	123.4179	161.7731	2.6476	14.9065	$327s$
$[20, 20, 20, 20]$+ELU	20000	116.5707	5.6802	2.1055	15.3905	$595s$
$[20, 20, 20, 20]$+ELU	30000	116.3503	5.2700	2.0440	15.3582	$864s$
$[20, 20, 20, 20, 20, 20]$	1	0.4421	1.6342×10^5	154.9823	-0.1989	$0s$
$[20, 20, 20, 20, 20, 20]$	500	130.3159	20.4544	3.1028	15.5226	$85s$
$[20, 20, 20, 20, 20, 20]$	5000	102.2737	5.6755	1.7017	15.0151	$234s$
$[20, 20, 20, 20, 20, 20]$	10000	96.0008	13.2512	1.8145	14.8077	$402s$
$[20, 20, 20, 20, 20, 20]$	20000	100.0730	4.1736	1.2484	14.8885	$740s$
$[20, 20, 20, 20, 20, 20]$	30000	100.7935	3.0847	1.1066	14.9858	$1081s$
$[50, 50, 50, 50]$	1	0.1066	1.7038×10^5	150.8409	0.0229	$0s$
$[50, 50, 50, 50]$	500	145.8189	48.7139	4.2810	15.5513	$72s$
$[50, 50, 50, 50]$	5000	109.1380	7.7543	2.4970	14.9733	$207s$
$[50, 50, 50, 50]$	10000	107.5036	3.4357	1.6579	15.0138	$359s$
$[50, 50, 50, 50]$	20000	105.2589	6.0111	1.7069	14.9242	$668s$
$[50, 50, 50, 50]$	30000	106.6101	3.6583	1.4809	14.9759	$979s$

Table 3.2: Training values for the Raissi method pricing an ATM call option under $r = 0.10$, $\sigma = 0.25$ in a Black Scholes model with strike 100. The true price is 14.9757. The results are given over a validation data set of 250 paths, while we train with batches of 100 paths. The loss is presented as the sum of losses over all paths instead of their mean. We train for 36000 iterations with piece-wise constant learning rate: 3000 iterations with $1e^{-2}$, 10000 with $1e^{-3}$, 10000 with $1e^{-4}$, 7000 with $1e^{-5}$ and 6000 with $1e^{-6}$.

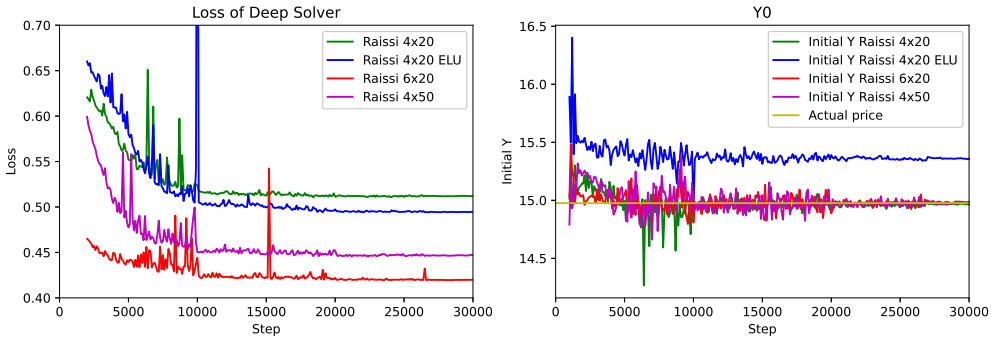


Figure 3.10: Training evolution for Raissi's method.

or underestimation. The averages of the absolute errors obtained ranges from 0.0756 at maturity to 0.8977 at $t = 0.6$. This is slightly higher than the average errors obtained in [47], which are of one order of magnitude less, and, in any case, too high to be used in industry standards.

In order to understand the nature of this error we have undergone several tests. In the literature they solve a Black-Scholes-Barenblatt 100-D BSDE:

$$\begin{cases} d\boldsymbol{X}_t &= 0.4 \cdot \operatorname{diag}(\boldsymbol{X}_t) d\boldsymbol{W}_t \\ dY_t &= 0.05 \left(Y_t - \boldsymbol{Z}_t^T \boldsymbol{X}_t\right) dt - 0.4 \cdot \boldsymbol{Z}_t^T \operatorname{diag}(\boldsymbol{X}_t) d\boldsymbol{W}_t dW_t \\ Y_T &= \|\boldsymbol{X}_T\|^2 \end{cases}$$

with initial $\boldsymbol{X}_0 = (1, 0.5, \ldots, 1, 0.5)$. After testing that our code obtains for the settings in the literature similar results, we have tried to discern the key differences between the two settings and identify four factors:

1. No drift term in the forward SDE in their setting,
2. scale of the state variables in the order of magnitude of the units,
3. smooth payoff,
4. bigger dimension.

We have discarded 1 and 2 as the root of our problem given that the problem persists with $r = 0$ and we are already re-scaling our state variables. Then we have focused on the effects of the remaining two factors. With respect to the payoff differentiability there is a jump at the strike in the derivative of the payoff $(X_T - K)^+$. In our perspective this can lead to bad learning of the correct function at maturity and this error propagates through the whole time dimension when learning the global behaviour of the function. In figure 3.11 we observe that indeed the error at maturity concentrates around strike which may support our hypothesis. We have tried smoothing the payoff function $g(\boldsymbol{X}_t)$ by:

$$g(\boldsymbol{X}_t) \approx \frac{1}{\lambda} \log\left(1 + e^{\lambda(\boldsymbol{X}_t - K)}\right) \tag{3.3.3}$$

with large λ as proposed in [18]. This requires special care to be taken with exploding numerical values in the code, as the exponential part can go to numerical infinity for some paths. Unfortunately, we have observed no reduction in the error so further analysis needs to be done in order to understand if this is indeed the root of the problem.
The implications of the high dimensionality are difficult to understand. In figures A.1, B.1 and C.1 we observe that there is no error reduction when increasing the time granularity but there is a big one when reducing the volatility. We believe that this will explain why, when increasing dimensions, the approximation error is smaller as in essence having a 100-D vector for a symmetric function at terminal condition, is equivalent to having 1-D vector with a much lower volatility.
In figure 3.12 we see the call prices replication and the paths of the underlying. The strike and means of the process are also plotted as our original hypothesis was that the error only happened for paths that end up out of the money and have been under the average price for a long trajectory. We have concluded by the above discussion that this is not the case and that the error is related with the volatility term.

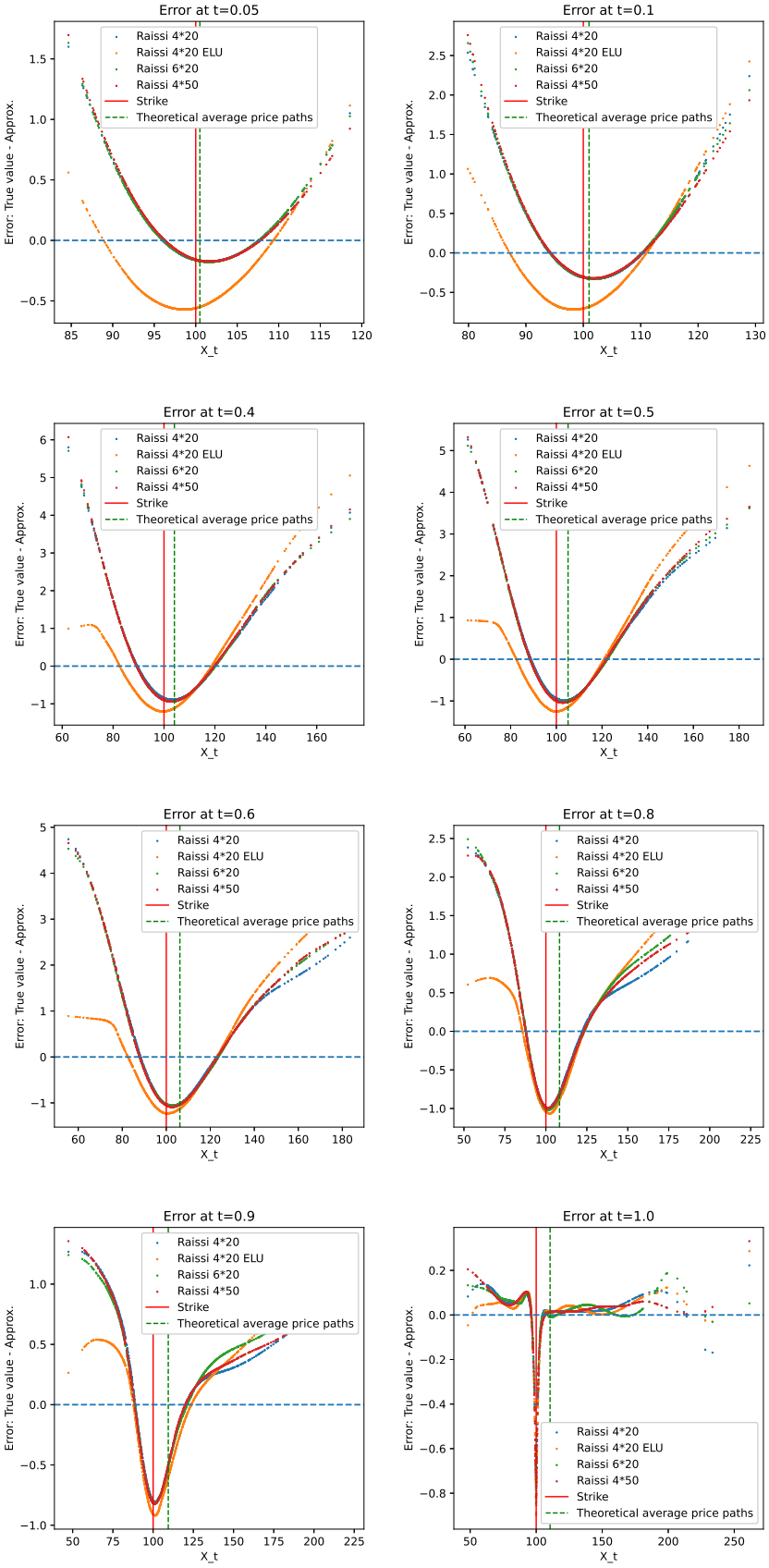
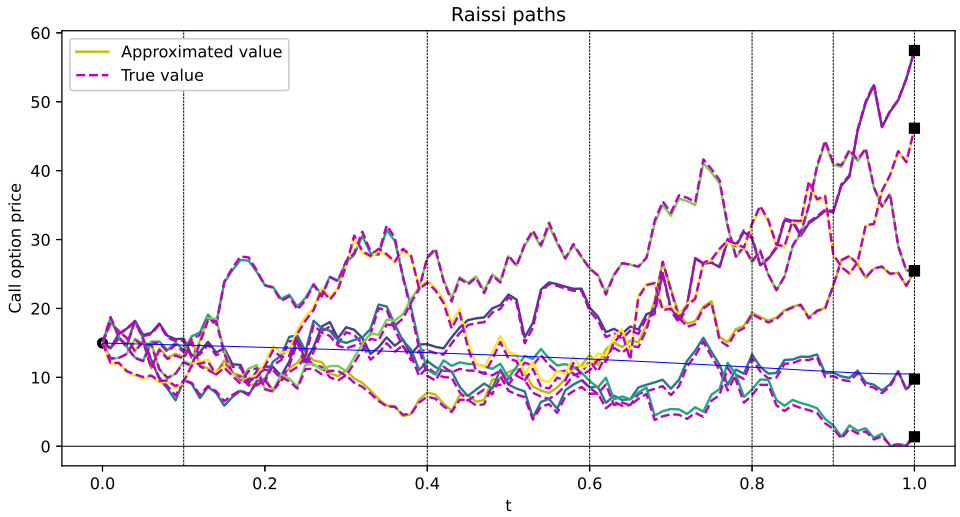
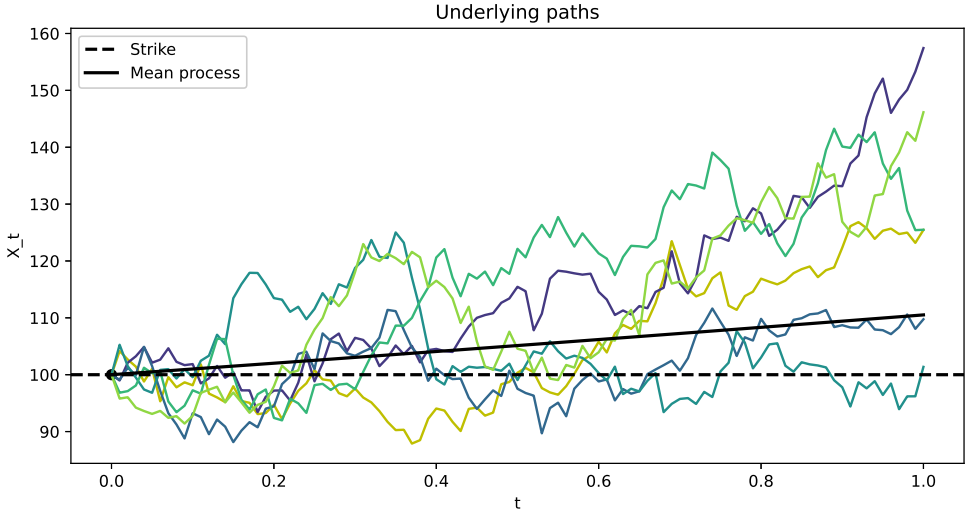


Figure 3.11: Evolution of error profile in Raissi method for different time instants. The vertical lines represent the strike and average of X_t.



(a) Price approximation vs true value. the blue line represents the price of a call whose spot price is the mean of the distribution of X_t for each t. The vertical grey lines are the instants from which the snapshot of the error profile is shown in figures 3.11, A.1, B.1 and C.1.



(b) Underlying price paths, strike and expected underlying path.

Figure 3.12: Pathwise approximation using Raissi's method.

Chapter 4

A parametric approach

In the previous section we study deep solvers that represent the solution process $\left(Y_s^{t,\boldsymbol{X}}, \boldsymbol{Z}_s^{t,\boldsymbol{X}}\right)$ in different ways, which always depend on a previous choice of dynamics and parameters for the FBSDE. Hence, our network only took the state variables $\boldsymbol{X}_t$ as inputs, and the time variable t when we shared a network through time. As a result those techniques fail to price under new market conditions as new training needs to be done. If the interest rate curve, the volatility observed in the market vary or the strike is different, no new prices can be provided quickly.

For this final chapter we propose a novel approach to solve parametric BSDEs using Deep learning. In a different setting, an example of applying parametric learning to solving PDEs can be found in the work of Kathrin Glau and Linus Wunderlich [23]. This requires transforming the previous frameworks into a parametric problem, that is, the inputs of the model are not only state variables and time, but also parameters modeling the dynamics of the FBSDE. In order to have this triple dependence, our model learns an operator that maps between infinite dimensional functional spaces. For example, it will take a short rate curve, a local volatility function and return a function $\mathbb{R}^d \times \mathbb{R}^+ \to \mathbb{R}$ which values a derivative. The core idea, as it will be developed in section 4.1, consists of approximating a function by an infinite series and learning the coefficients and basis functions separately. In section 4.2 we briefly discuss some tips to simplify the network, and in section 4.3 we explain the details on the training.

4.1 DeepONet

As seen in theorem 2.2.2 every "good" function can be approximated by neural networks. A result, which may not be as well known, is that any non-linear continuous functional (see [15], [42]) or transformation between functional spaces (called **operator** see ([17], [16])) can also be approximated by neural networks. In particular we have the following theorem proven in [17]:

Theorem 4.1.1 (Universal approximation theorem for Operator, [17] Theorem 1)**.** *Suppose that σ is a continuous non polynomial function and K_1, K_2 are compact subsets of a Banach Space X and $\mathbb{R}^N$ respectively. Let V be a compact subset in $C(K_1)$ and G a nonlinear continuous operator mapping $V \to C(K_2)$. We recall $C(K_1)$ is the set of continuous functions from K_1 to $\mathbb{R}$. Then, for any $\epsilon > 0$ there are positive integers n, p, m and sets of constants $c_i^k, \xi_{ij}^k, \theta_i^k, \zeta_k \in \mathbb{R}$, $w_k \in \mathbb{R}^N$, $x_j \in K_1$ for $i = 1, \dots, n, k = 1, \dots, p, j = 1, \dots, m$ such that $\forall u \in V, y \in K_2$ it holds:*

$$\left| G(u)(y) - \sum_{k=1}^{p} \left(\sum_{i=1}^{n} c_i^k \sigma \left(\sum_{j=1}^{m} \xi_{ij}^k u(x_j) + \theta_i^k \right) \right) \sigma(w_k \cdot y + \zeta_k) \right| < \epsilon \tag{4.1.1}$$

We will be using the ideas in [23] to construct our network. Theorem 4.1.1 tells us that the operator G can be approximated via the composition of affine operations and the activation function σ, which is again a reassuring result as it means it can be approximated by a neural network. Differently from the Universal Approximation theorem for functions, theorem 4.1.1 will additionally be helpful in the architecture of the network. The main insight of the above theorem is that we find a split between the term taking as input the information about the functional, and the part taking as input the vector y.

For our purposes, if we have a parameterised FBSDE with solution $\left(Y_s^{t,\boldsymbol{X}}(\alpha), \boldsymbol{Z}_s^{t,\boldsymbol{X}}(\alpha)\right)$, where α are the parameters controlling the FBSDE, we try to represent:

$$Y_s^{t,\boldsymbol{X}}(\alpha) = u(\alpha \mid t, \boldsymbol{X}) \approx G(\alpha)(t, \boldsymbol{X})$$

by the function form in the Markovian framework.
Although the numerical examples will be done only for constant parameters we explain the theory in the general case where the dynamics can be given by possibly random functions. Notably a Neural Network cannot take an infinite number of inputs, for the value of the function in its entire domain, so we require the following definition:

Definition 4.1.2. Let $u \in V$ as above and let $\{x_1, \ldots, x_n\} \in X$. We say that $\{x_1, \ldots, x_n\} \in X$ is the **set of sensors** of u in X. The idea is that functions need to be represented by numerical values to act as an input for a neural network and hence we need to work with $\{u(x_1), \ldots, u(x_n)\}$.

We call y, as in theorem 4.1.1, the **external input**.

Determining how the structure of the network should be is not a trivial task. The first idea is to simply input all the parameters and state variables in a FNN and expect the network to learn the correct form. The high complexity of the relations between the parameters, and the state variables in the function will require a very rich net which can possibly lead to over-parametrisation, over-fit or slow training. Going back to the theorem 4.1.1, as mentioned, we observed a split between the part with the sensors and the external input, both being affine functions and activation, which is essentially two different FNN. The same will be done in our architecture.
We believe that learning operators may be seen as too abstract a framework compared to learning a real valued function. To give an intuition on how operators are well known mathematical objects, we provide two examples. Firstly, suppose that we want to map a continuous function to the space of infinite series of polynomials. This map is an operator and we can take for instance the Taylor series of the function around a point a:

$$T_a(f)(y) = \left(\sum_{n=0}^{\infty} \frac{f^{(n)}(a)}{n!} (\bullet - a)^n \right)(y)$$

Another example is the Fourier series of a periodic function:

$$Fourier(f)(y) = \left(A_0 + \sum_{n=1}^{\infty} \left(A_n \cos\left(\frac{2\pi n}{T} \bullet \right) + B_n \sin\left(\frac{2\pi n}{T} \bullet \right) \right) \right)(y)$$

where:

- A_0 is the average value of the function f over one period,
- A_n and B_n are the Fourier coefficients given by:

$$A_n = \frac{2}{T} \int_0^T f(u) \cos\left(\frac{2\pi n u}{T} \right) du \qquad B_n = \frac{2}{T} \int_0^T f(u) \sin\left(\frac{2\pi n u}{T} \right) du$$

Both operators represent a function as a series of coefficients and basis functions. In the first case the coefficients are given by $\left\{ \frac{f^{(n)}(a)}{n!} \right\}_{n=0}^{\infty}$ and basis functions $\{(x-a)^n\}_{n=0}^{\infty}$. In Fourier series the coefficients are given by the A_n and B_n, with sines and cosines as basis functions. We propose a network architecture of the form operator:

$$\mathcal{O}(f)(t, \boldsymbol{X}_t) = \left(\sum_{n=0}^{\infty} \mathcal{B}(\alpha) \cdot \mathcal{K}(\bullet, \bullet) \right)(t, \boldsymbol{X}_t)$$

which is called **DeepONet**. We cannot compute an infinite series so we cap at some index q which is considered big enough to achieve accuracy, as we will do with a Taylor representation. The functions $\mathcal{B}$ and $\mathcal{K}$ are represented by neural networks following theorem 4.1.1. In figure 4.1 we see the data flow of the high level architecture. The neural networks representing the coefficients is the **BranchNet** and the one representing the basis functions is the **TrunkNet**.

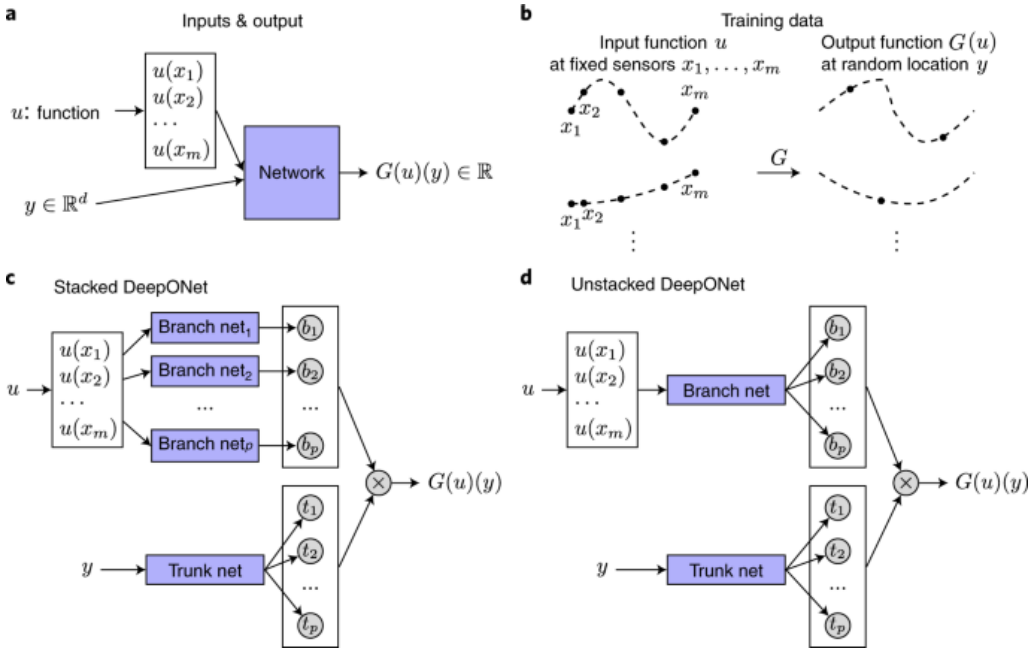


Figure 4.1: Structure of a DeepONet. On top we have the Branch Net which gives the basis functions coefficients, and below the Trunk net which gives the basis functions. We use $\bigotimes$ to denote element wise multiplication.

4.2 Subnets.

The DeepONet is a high level architecture and there are different possibilities to choose the sub-architectures of the Trunk and the Branch nets. In [53], [22] and [36] there are ideas for the input of data, and reducing the number of parameters of the network. We highlight two important additions that will be studied in further research: stacked branch net and Permutation Invariant layer.

The first one is observed in figure 4.1 (c and d). The idea is that each sensor of constant parameter can either go through a different network, or a shared one, in order to be transformed into the coefficients of the basis functions, or BranchNet. The reduction in the number of parameters, specially for a big number of sensors, makes the unstacked (shared) branch net a better choice ([53]).

The second addition, the Permutation Invariant layer causes a reduction in the complexity of the TrunkNet. In the case of pricing an option in a high dimensional setting, there is frequently some symmetry among the state variables at maturity, as seen, for example in equation, 3.2.10. Any permutation of the underlying prices leads to the same payoff. Therefore there are reasons to believe that this is also true at all previous times and that we do not need the information of each individual component.The solution, as proposed in [22], consists of adding a first layer in the TrunkNet that calculates a quantity based on the state variables such as their mean, geometric mean or maximum. The Permutation Invariant layer will be dependant on the payoff.

4.3 Training a DeepONet

Finally, we explain how to train the DeepONet to learn its functional form. The DeepONet has already been studied in the literature [23] and our contribution is using it together with the BSDE structure of the problem. The principles followed are similar to the Raissi method previously discussed. The main difference is that, to train the DeepONet, we require knowledge of the parameters, not only as inputs of the function but also for the simulation of the Forward SDE to generate training data. This is because the parameters are inside the drift and diffusion term. In the case of curves it is necessary to assume that they are parameterised by a finite set of parameters α (for example a local volatility function). The set of parameters follows a probability distribution $\mathcal{D}$ which is known.

Let $\mathcal{G}(\alpha)(t, \boldsymbol{X}_t)$ be the DeepONet, with network parameters θ. To compute the loss we proceed as following:

1. Sample parameters $\alpha^j \sim \mathcal{D}$ for $j = 1, \dots J$. Each form a training super-batch.

2. For each super-batch simulate M paths of the SDE along a partition of the interval $[0, T]$, $\{0 = t_0, t_1 \dots, t_N = T\}$. Using j for the super-batch, m for the path, and t_k for the time instant:

$$\boldsymbol{X}_{t_{i+1}}^{j,m} = \boldsymbol{\mu}\left(\alpha^j \mid t_i, \boldsymbol{X}_{t_i}^{j,m}\right)\Delta_{t_i} + \boldsymbol{\sigma}\left(\alpha^j \mid t_i, \boldsymbol{X}_{t_i}^{j,m}\right) \cdot \Delta W_{t_i}^{j,m} \tag{4.3.1}$$

   where the initial $\boldsymbol{X}_0^j$ can also be a parameter of the model.

3. Initialise for every j, m:

$$Y_{t_0}^{j,m} = \mathcal{G}(\alpha^j)(t_0, \boldsymbol{X}_{t_0}^{j,m}) \quad \boldsymbol{Z}_{t_0}^{j,m} = \nabla_X \mathcal{G}(\alpha^j)(t_0, \boldsymbol{X}_{t_0}^{j,m}) \quad \mathcal{L}^{j,m} = 0$$

4. For $i = 0, 1, \dots N-1$ let $Y_{t_{i+1}}^{j,m} = \mathcal{G}(\alpha^j)(t_{i+1}, \boldsymbol{X}_{t_{i+1}}^{j,m})$, $\boldsymbol{Z}_{t_{i+1}}^{j,m} = \nabla_X \mathcal{G}(\alpha^j)(t_{i+1}, \boldsymbol{X}_{t_{i+1}}^{j,m})$ and:

$$\mathcal{L}^{j,m} = \mathcal{L}^{j,m} + \left(Y_{t_{i+1}}^{j,m} - Y_{t_i}^{j,m} + f\left(t_i, \boldsymbol{X}_{t_i}^{j,m}, Y_{t_i}^{j,m}, \boldsymbol{Z}_{t_i}^{j,m}\right)\Delta_{t_i} - \boldsymbol{\sigma}^T\left(\alpha^j \mid t_i, \boldsymbol{X}_{t_i}^{j,m}\right)\boldsymbol{Z}_{t_i}^{j,m} \cdot \Delta W_{t_i}^{j,m}\right)^2$$

5. At maturity T: $\mathcal{L}^{j,m} = \mathcal{L}^{j,m} + \left(Y_{t_N}^{j,m} - g\left(t_N, \boldsymbol{X}_{t_N}^{j,m}\right)\right)^2$ where g is the payoff function or termincal condition on the BSDE.

We try to minimise.

$$\mathcal{L}(\theta) = \mathbb{E}_{\alpha \sim \mathcal{D}}[\mathcal{L}^{\alpha}(\theta)] \tag{4.3.2}$$

so we train with the network as in other sections.
In the tests done we have trained a DeepONet to price a European call in a Black Scholes model using as parameters r, *sigma* and strike K. Training with r and σ allows pricing under different market conditions, while learning K permits pricing different instruments. The distributions of the parameters is shown in table 4.1 with $X_0 = 100$. The trained network is validated with a validation set of 250 paths within 100 super-batches and 50 time steps. The result is an average squared error, at each time point, with respect to the correct value of 0.5567 after 3 hours of training. In a similar time, if we price an option with $X_0 = 100$ and strikes in a distribution $U(95, 105)$ we obtain an average error of 0.4012 which suggest that the strike is, similarly to Raissi's method, a problematic point.

Parameter	**Distribution**
σ	$U(0.2375, 0.2625)$
r	$U(0.095, 0.105)$
K	$U(95, 105)$

Table 4.1: Parameter Distributions for the DeepONet training.

Chapter 5

Conclusions and further research.

In this thesis we have presented a set of techniques which can be helpful in several areas of financial mathematics, particularly in derivative pricing. In more generality they can be used to solve any BSDE or PDE through the Generalised Feynman-Kac theorem. This techniques have the potential to handle high dimensional problems since they are not dimension dependent. Although this techniques show good results in the literature, we have shown that they require careful considerations in order to be used in general settings, like low dimensions, where their accuracy decreases. In the literature this problems might not arise by their particular setting. For instance, symmetry along the variables, the terminal condition functions are C^∞, solution values far from zero (so the relative error does not explode) and the stochastic process solution of the forward SDE has a similar scale to the time variable. We have shown some variations to achieve better results such as the hybrid approach, re-scaling of the state variables, using the ELU activation function in the last layer, or adding shortcuts in the networks. Some of these modifications are ad-hoc solution which makes them difficult to be used in a new, unknown problem, which is a requirement when building a generic pricer. Additionally a novel, parametric, approach has been proposed an proved in a simple setting. We are particularly interested in this method and test different enhacements to achieve better perfromance.

This work will be continued to create an entire library of Deep Learning based prices that can be used for fast benchmarking purposes in the counterparty credit risk space. The first step will be to improve the numerical accuracy in general settings, and the speed of the methods. It also has to be tested in different models such as a Heston volatility model where the control process in the BSDE becomes $Z_t = (X_t \mathrm{Vol}_t \nabla_X u, \sigma \mathrm{Vol}_t \nabla_{\mathrm{Vol}} u)$.
One immediate continuation is the handling of Asian options since in Markovian frameworks it is possible to use a state variable such as $A_t = \frac{1}{t}\int_0^t X_u du$ in the pricing function. A further development will be the pricing of early exercise options. If the option can be exercised in dates $\tau_1, \ldots, \tau_k$, we modify the FBSDE 1.2.1 to have this extra information (see [21] and [52]):

$$
\begin{aligned}
Y_T &= g(\boldsymbol{X}_T) \\
Y_t &= Y_{\tau_{i+1}} + \int_t^{\tau_i+1} f_s ds - \int_t^{\tau_i+1} \boldsymbol{Z}_s \cdot \boldsymbol{dW}_s, \;\; t \in (\tau_i, \tau_{i+1}), \\
Y_{\tau_i} &= \max\left\{ g(\boldsymbol{X}_{\tau_i}), \int_{\tau_i}^{\tau_i+1} f_s ds - \int_{\tau_i}^{\tau_i+1} \boldsymbol{Z}_s \cdot \boldsymbol{dW}_s \right\}
\end{aligned}
$$

where f is the generator of the BSDE.

Finally, BSDEs are helpful tools in the formulation of XVA problems, particularly if we include collateral and funding making the problem recursive. In [11] a general framework is provided, and the price process including XVA is formulated as a BSDE. Given this thesis has been done in a team whose main focus is XVA calculations, it is of our interest to solve the BSDE problems coming from XVA calculations. Therefore we would like to apply all the above mentioned techniques in the computation of prices including XVA.

Appendix A

Raissi under $r = 10\%$, $\sigma = 3\%$.

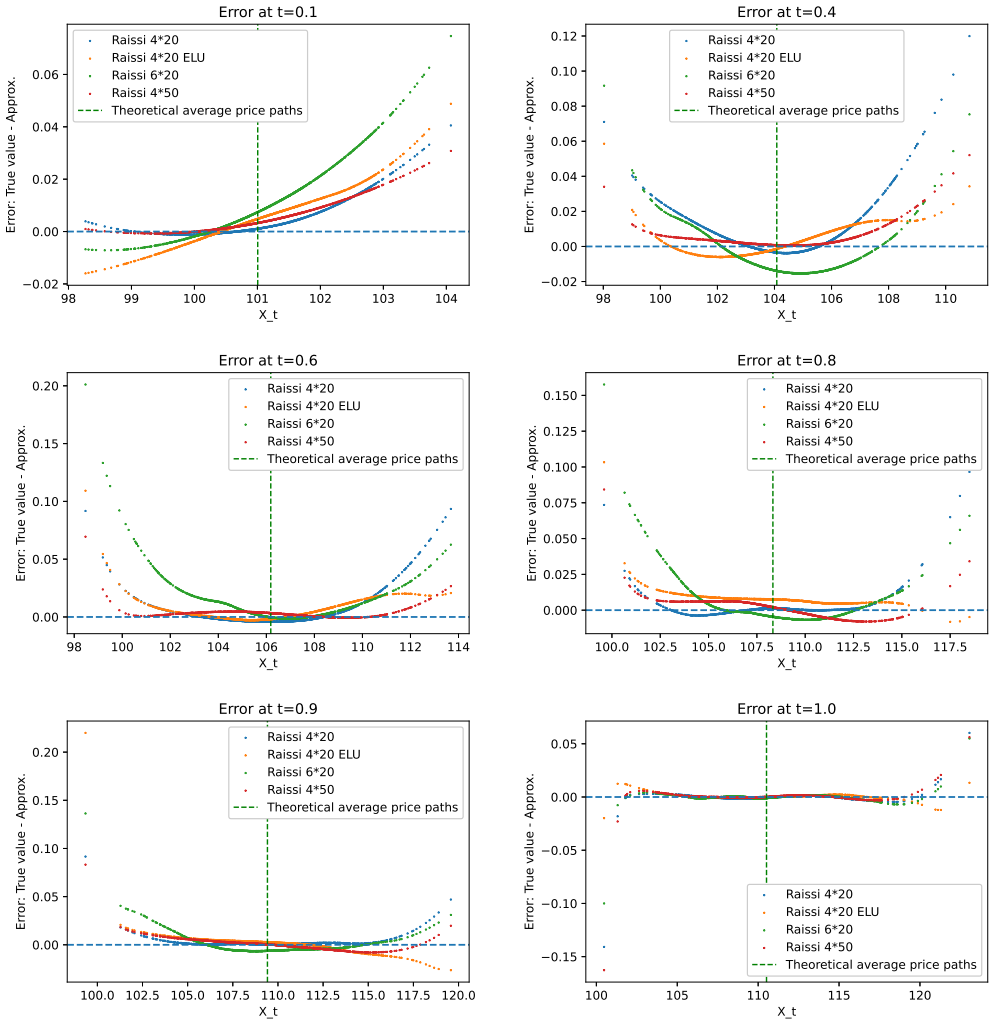


Figure A.1: Evolution of the error profile of Raissi method under 3% volatility. The time grid has 100 steps. Note that most paths end up deeply in the money, where there is very small error.

Appendix B

Raissi under $r = 0\%$, $\sigma = 3\%$.

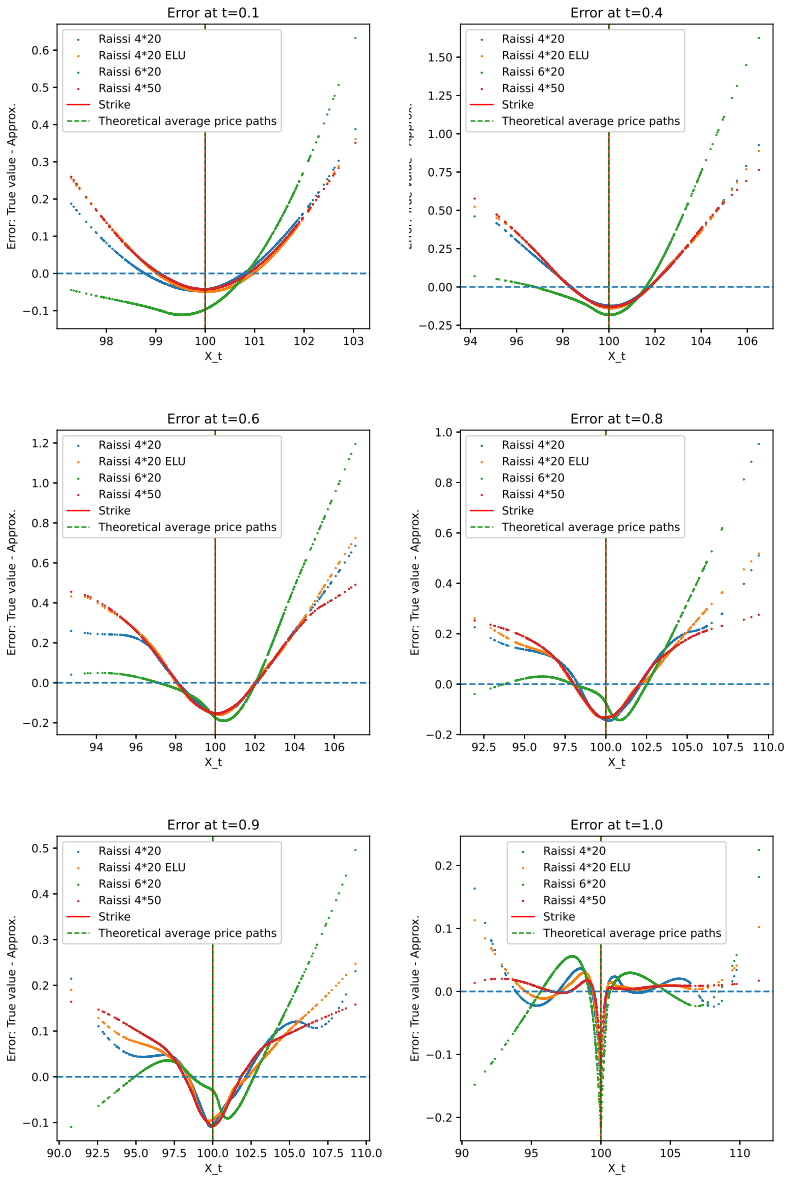


Figure B.1: Evolution of the error profile of Raissi method under 3% volatility and no drift. The time grid has 100 steps.

Appendix C

Raissi under $r = 10\%$, $\sigma = 25\%$, 200 time steps.

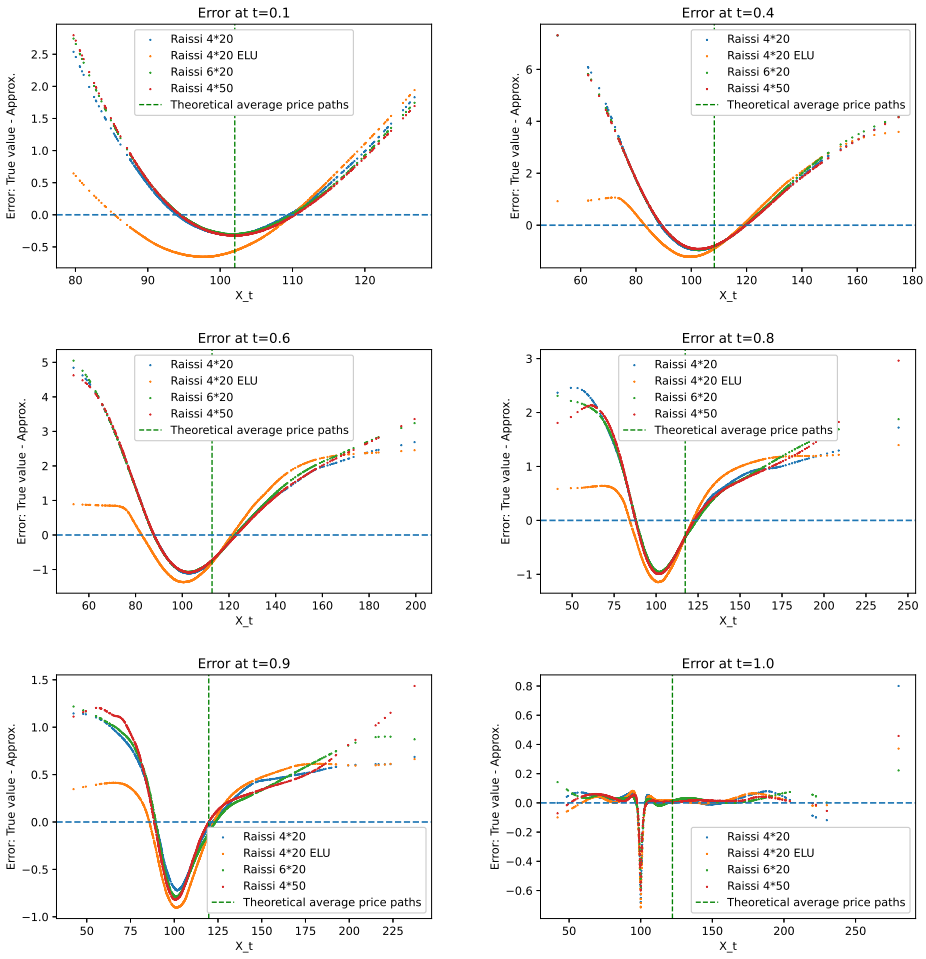


Figure C.1: Evolution of the error profile of Raissi method under 25% volatility and $r = 10\%$. The time grid has 200 steps.

Bibliography

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org: `https://www.tensorflow.org/`.

[2] Atilim Gunes Baydin, Barak A Pearlmutter, Alexey Andreyevich Radul, and Jeffrey Mark Siskind. Automatic differentiation in machine learning: a survey. *Journal of Marchine Learning Research*, 18:1–43, 2018.

[3] Sebastian Becker, Patrick Cheridito, and Arnulf Jentzen. Deep optimal stopping. *The Journal of Machine Learning Research*, 20(1):2712–2736, 2019.

[4] Sebastian Becker, Patrick Cheridito, and Arnulf Jentzen. Pricing and hedging american-style options with deep learning. *Journal of Risk and Financial Management*, 13(7):158, 2020.

[5] Sebastian Becker, Patrick Cheridito, Arnulf Jentzen, and Timo Welti. Solving high-dimensional optimal stopping problems using deep learning. *European Journal of Applied Mathematics*, 32(3):470–514, 2021.

[6] Richard Bellman. Dynamic programming. *Science*, 153(3731):34–37, 1966.

[7] Jean-Michel Bismut. *Théorie probabiliste du contrôle des diffusions*, volume 167. American Mathematical Soc., 1976.

[8] Fischer Black and Myron Scholes. The pricing of options and corporate liabilities. *Journal of political economy*, 81(3):637–654, 1973.

[9] Jan Blechschmidt and Oliver G Ernst. Three ways to solve partial differential equations with neural networks—a review. *GAMM-Mitteilungen*, 44(2):e202100006, 2021.

[10] Bruno Bouchard. Lecture notes on bsdes main existence and stability results, 2015. CEREMADE - Centre de Recherches en Mathématiques de la Decision.

[11] Damiano Brigo, Marco Francischello, and Andrea Pallavicini. Nonlinear valuation under credit, funding, and margins: Existence, uniqueness, invariance, and disentanglement. *European Journal of Operational Research*, 274(2):788–805, 2019.

[12] Damiano Brigo, Massimo Morini, and Andrea Pallavicini. *Counterparty credit risk, collateral and funding: with pricing cases for all asset classes*, volume 478. John Wiley & Sons, 2013.

[13] Mark Broadie, Yiping Du, and Ciamac C Moallemi. Risk estimation via regression. *Operations Research*, 63(5):1077–1097, 2015.

[14] Quentin Chan-Wai-Nam, Joseph Mikael, and Xavier Warin. Machine learning for semi linear pdes. *Journal of scientific computing*, 79(3):1667–1712, 2019.

[15] Tianping Chen and Hong Chen. Approximations of continuous functionals by neural networks with application to dynamic systems. *IEEE Transactions on Neural networks*, 4(6):910–918, 1993.

[16] Tianping Chen and Hong Chen. Approximation capability to functions of several variables, nonlinear functionals, and operators by radial basis function neural networks. *IEEE Transactions on Neural Networks*, 6(4):904–910, 1995.

[17] Tianping Chen and Hong Chen. Universal approximation to nonlinear operators by neural networks with arbitrary activation functions and its application to dynamical systems. *IEEE transactions on neural networks*, 6(4):911–917, 1995.

[18] Yangang Chen and Justin WL Wan. Deep neural network framework based on backward stochastic differential equations for pricing and hedging american options in high dimensions. *Quantitative Finance*, 21(1):45–67, 2021.

[19] Djork-Arné Clevert, Thomas Unterthiner, and Sepp Hochreiter. Fast and accurate deep network learning by exponential linear units (elus). *arXiv preprint arXiv:1511.07289*, 2015. url: `http://arxiv.org/abs/1511.07289`.

[20] Weinan E, Jiequn Han, and Arnulf Jentzen. Deep learning-based numerical methods for high-dimensional parabolic partial differential equations and backward stochastic differential equations. *Communications in Mathematics and Statistics*, 5(4):349–380, nov 2017. url: `https://doi.org/10.1007%2Fs40304-017-0117-6`.

[21] Chengfan Gao, Siping Gao, Ruimeng Hu, and Zimu Zhu. Convergence of the backward deep bsde method with applications to optimal stopping problems. *arXiv preprint arXiv:2210.04118*, 2022. url: `http://arxiv.org/abs/2210.04118`.

[22] Maximilien Germain, Mathieu Laurière, Huyên Pham, and Xavier Warin. Deepsets and their derivative networks for solving symmetric pdes. *Journal of Scientific Computing*, 91(2):63, 2022.

[23] Kathrin Glau and Linus Wunderlich. The deep parametric pde method and applications to option pricing. *Applied Mathematics and Computation*, 432:127355, 2022. url: `https://doi.org/10.1016/j.amc.2022.127355`.

[24] 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*, pages 249–256. JMLR Workshop and Conference Proceedings, 2010.

[25] Xavier Glorot, Antoine Bordes, and Yoshua Bengio. Deep sparse rectifier neural networks. In *Proceedings of the fourteenth international conference on artificial intelligence and statistics*, pages 315–323. JMLR Workshop and Conference Proceedings, 2011.

[26] Alessandro Gnoatto, Athena Picarelli, and Christoph Reisinger. Deep xva solver: A neural network–based counterparty credit risk management framework. *SIAM Journal on Financial Mathematics*, 14(1):314–352, 2023.

[27] Lukas Gonon. Lecture notes on deep learning, 2022. Imperial College London, Department of Mathematics.

[28] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[29] Ludovic Goudenege, Andrea Molent, and Antonino Zanette. Computing xva for american basket derivatives by machine learning techniques. *arXiv preprint arXiv:2209.06485*, 2022. url: `http://arxiv.org/abs/2209.06485`.

[30] Ivan Guo, Nicolas Langrené, and Jiahao Wu. Simultaneous upper and lower bounds of american option prices with hedging via neural networks. *arXiv preprint arXiv:2302.12439*, 2023. url: `http://arxiv.org/abs/2302.12439`.

[31] Jiequn Han, Arnulf Jentzen, and Weinan E. Solving high-dimensional partial differential equations using deep learning. *Proceedings of the National Academy of Sciences*, 115(34):8505–8510, 2018.

[32] Jiequn Han and Jihao Long. Convergence of the deep bsde method for coupled fbsdes. *Probability, Uncertainty and Quantitative Risk*, 5:1–33, 2020.

[33] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition, 2015. url: `http://arxiv.org/abs/1512.03385`.

[34] Kiyosi Itô. Stochastic integral. *Proceedings of the Imperial Academy*, 20(8):519 – 524, 1944.

[35] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014. url: `http://arxiv.org/abs/1412.6980`.

[36] Nikola Kovachki, Zongyi Li, Burigede Liu, Kamyar Azizzadenesheli, Kaushik Bhattacharya, Andrew Stuart, and Anima Anandkumar. Neural operator: Learning maps between function spaces, 2023. url: `http://arxiv.org/abs/2108.08481`.

[37] Bernard Lapeyre and Jérôme Lelong. Neural network regression for bermudan option pricing. *Monte Carlo Methods and Applications*, 27(3):227–247, 2021.

[38] Alexander LeNail. Nn-svg: Publication-ready neural network architecture schematics. *Journal of Open Source Software*, 4(33):747, 2019. url: `https://doi.org/10.21105/joss.00747`.

[39] Moshe Leshno, Vladimir Ya Lin, Allan Pinkus, and Shimon Schocken. Multilayer feedforward networks with a nonpolynomial activation function can approximate any function. *Neural networks*, 6(6):861–867, 1993.

[40] Jian Liang, Zhe Xu, and Peter Li. Deep learning-based least squares forward-backward stochastic differential equation solver for high-dimensional derivative pricing. *Quantitative Finance*, 21(8):1309–1323, 2021.

[41] Francis A Longstaff and Eduardo S Schwartz. Valuing american options by simulation: a simple least-squares approach. *The review of financial studies*, 14(1):113–147, 2001.

[42] Hrushikesh Narhar Mhaskar and Nahmwoo Hahm. Neural networks for functional approximation and system identification. *Neural Computation*, 9(1):143–159, 1997.

[43] Nikolas Nüsken and Lorenz Richter. Interpolating between bsdes and pinns: deep learning for elliptic and parabolic boundary value problems. *arXiv preprint arXiv:2112.03749*, 2021. url: `https://arxiv.org/abs/2112.03749`.

[44] Bernt Oksendal. *Stochastic differential equations: an introduction with applications*. Springer Science & Business Media, 2013.

[45] Etienne Pardoux and Shige Peng. Adapted solution of a backward stochastic differential equation. *Systems & control letters*, 14(1):55–61, 1990.

[46] Nicolas Perkowski. Backward stochastic differential equations: An introduction. Lecture Notes, 2011. University Name, Course Code.

[47] Maziar Raissi. Forward-backward stochastic neural networks: Deep learning of high-dimensional partial differential equations. *arXiv preprint arXiv:1804.07010*, 2018. url: `https://arxiv.org/abs/1804.07010`.

[48] Maziar Raissi, Paris Perdikaris, and George E Karniadakis. Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations. *Journal of Computational physics*, 378:686–707, 2019.

[49] Maziar Raissi, Paris Perdikaris, and George Em Karniadakis. Physics informed deep learning (part i): Data-driven solutions of nonlinear partial differential equations. *arXiv preprint arXiv:1711.10561*, 2017. url: `https://arxiv.org/abs/1711.10561`.

[50] Robert Schöftner. On the estimation of credit exposures using regression-based monte carlo simulation. *The journal of credit risk*, 4(4):37–62, 2008.

[51] Jian-Huang She and Dan Grecu. Neural network for cva: Learning future values. *arXiv preprint arXiv:1811.08726*, 2018. https://arxiv.org/abs/1811.08726.

[52] Haojie Wang, Han Chen, Agus Sudjianto, Richard Liu, and Qi Shen. Deep learning-based bsde solver for libor market model with application to bermudan swaption pricing and hedging. *arXiv preprint arXiv:1807.06622*, 2018. url: https://arxiv.org/abs/1807.06622.

[53] Sifan Wang, Hanwen Wang, and Paris Perdikaris. Learning the solution operator of parametric partial differential equations with physics-informed deeponets, 2021. http://arxiv.org/abs/2103.10974.

[54] Wikipedia. Activation function. https://en.wikipedia.org/wiki/Activation_function. [Online; accessed: 02/08/2023].

[55] Jianfeng Zhang. *Backward stochastic differential equations*. Springer, 2017.

[56] Harry Zheng. Lecture notes: Simulation methods in finance, 2022. Imperial College London, Department of Mathematics.