Deep Learning-Based Algorithms for Stochastic Control of Jump Diffusion in Finance

Rodney Voskamp

October 2, 2023

Delft University of Technology
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft Institute of Applied Mathematics

MSc Thesis Applied Mathematics

Deep Learning-Based Algorithms for Stochastic Control of Jump Diffusion in Finance

Rodney Voskamp

October 2, 2023

Delft University of Technology

Thesis Committee

Dr. S. Liu,	TU Delft and ING Bank, Supervisor
Prof.dr.ir, C. Vuik,	TU Delft
Dr. F. Yu,	TU Delft

To be defended on October 9, 2023

Delft, the Netherlands

Acknowledgements

I would like to express my gratitude to Dr. Shuaiqiang Liu for the guidance he gave me during this thesis. He provided important background material and his knowledge of neural networks proved invaluable. I would like to thank Prof.dr.ir Kees Vuik for being the responsible professor. At last, I would like to mention Dr. Fenghui Yu for being part of the examination committee.

Abstract

PDEs, like HJB-equations, can be solved using grid-based methods. These methods are inefficient for solving high-dimensional HJB-equation, because they suffer from the Curse of Dimensionality. Neural networks may overcome this problem. In this research, we solve high dimensional Partial Integro Differential Equations (PIDE) using neural networks. PIDE are PDEs that are associated with a jump-diffusion process. In this work, we only use finite activity jump processes. This means that the jump has a compensation component that to make it a martingale. We show two methods to solve PIDEs: a forward method (H-dBSDE, dBSDE-Jump) and a backward method (DBDP-MC). Both methodologies use neural networks to regress the solution and its derivative. The DBDP-MC is extended to jumps by calculating the compensation of the jumps with an offline Monte Carlo simulation. We tested this methodology on Bermudan basket options with 50 dimensions. The method was able to price them correctly. The dBSDE was extended by adding a new set of neural networks. These networks are learned with a different extra loss function. We argue that we can learn the two losses in a hierarchical way, leading to the Hierarchical dBSDE (H-dBSDE) method. Other work was done by minimizing the two loss functions simultaneously by using the sum of them. Easier problems like pricing European option can be solved correctly by the dBSDE-Jump method. However, we show that this can lead to wrong terminal fits, which makes it difficult to solve complex problems efficiently.

Keywords— HJB-equation, Lévy process, DBDP, dBSDE

Table of Contents

1 Introduction **1**

2 The Hamilton-Jacobi-Bellman equation and the curse of dimensionality **4**
2.1 Basics of Stochastic Calculus 4
2.2 The Hamilton-Jacobi-Bellman equation and the dynamic programming principle 5
2.2.1 Dynamic Programming Principle 5
2.2.2 Deterministic Hamilton-Jacobi-Bellman equation 5
2.2.3 General Hamilton-Jacobi-Bellman equation 6
2.3 Curse of dimensionality 7

3 Numerical schemes for PDEs in stochastic control with Itô-processes **8**
3.1 Neural Networks 8
3.1.1 Feed Forward 8
3.1.2 Back-propagation 10
3.1.3 Universal Approximation Theorem 11
3.2 Numerical schemes 11
3.2.1 Deep learning Backward Dynamic Programming (DBDP) . . . 13
3.2.2 deep Backward Stochastic Differential Equation (dBSDE) . . . 14
3.2.3 Comparing the dBSDE and DBDP methods 15
3.2.4 Other neural network algorithms 16
3.3 Connection between the HJB-equation and neural networks 17
3.3.1 Barron Space 17
3.3.2 Other Arguments 18

4 Jump Diffusion **20**
4.1 HJB for jump processes 20
4.1.1 Lévy Process 20
4.1.2 Bernoulli approximation 22
4.2 Schemes 23
4.2.1 Forward Backward SDE with jumps (FBSDEj) 23
4.2.2 Extensions to DBDP 24
4.2.3 deep BSDE with Jumps 25
4.3 Theoretical results for the FBSDEj and DBDP-MC 29
4.3.1 Uniqueness and Existence FBSDEj 29
4.3.2 Convergence of DBDP-MC 30

5 Numerical Results **33**
5.1 Hyperparameters 33
5.2 Bermudan Options 34
5.2.1 Setting 34
5.2.2 A reference method: SGBM 35
5.2.3 Test Case 35
5.2.4 High dimensional Bermudan Options 36
5.2.5 Other options 37
5.2.6 Poisson process and Bernoulli approximation 39
5.2.7 Conclusion 40

5.3 dBSDE-Jump 40
5.3.1 Conclusion 42

6 Discussion and Conclusion **43**

References **43**

Appendix A: Proofs of section 2 **48**

Appendix B: Proof of Itô lemma with jumps **52**

Appendix C: dBSDE-Jump might lead to an incorrect solution **54**

1 Introduction

Many modern tasks in mathematical finance can be described as an optimal control problem. For example, option pricing can be described by a stochastic control problem. Hedging is also a control problem [10]. Here we have a choice to buy, for example, some stocks, bonds and options. The goal is to minimize the risk under the constraint that we only have a finite amount of capital, the control. But stochastic control can also price Credit Valuation Adjustment (CVA) [25, 29] and is even able to describe the strategy used in High-Frequency-Trading [5, 16].

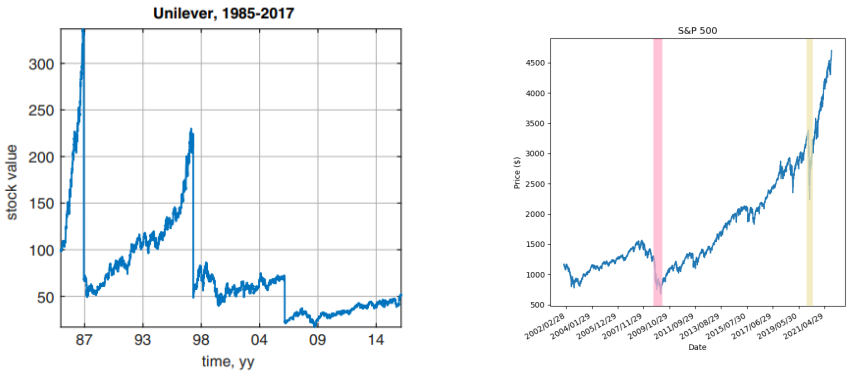
In this work, we price European, American and Bermudan basket options. European options earn a pay-off at the end of their contract, the terminal time. For example, an arithmetic-mean-put option sells the mean of the basket of assets for a certain strike price. The goal is to determine the fair price for this option at the start of the contract. American and Bermudan options can be exercised before the terminal time. Bermudan options have a few early exercise opportunities. Here we are allowed to wait for the next exercise chance or to earn the pay-off immediately. American options are similar but can be exercised at any time before maturity. The early exercise choice makes this a more difficult problem. We can price Bermudan options by comparing the pay-off at an exercise opportunity with the current expected price of the option. Bermudan and American options should therefore be solved backward through time.

We solve the optimal stochastic control problem by rewriting it into a PDE. To this end, we first transform the control problem into a backwards recursive optimization problem using the Dynamic Programming Principle. From here it is easy to derive the PDE, the Hamilton-Jacobi-Bellman (HJB) equation. This HJB equation still has an inner optimization problem. In this work, we solve the optimization problems analytically. However, this analytic solution is not always available and may need to be calculated using some numerical scheme. We substitute the analytic solution of the optimization problem in the PDE, which we can solve using grid-based techniques. From this, we can determine the desired solution to the optimal control problem.

However, when we get a high-dimensional PDE, grid-based methods suffer from the Curse of Dimensionality (CoD). Solving the PDE becomes exponentially harder when we increase the dimension of this PDE. A potential solution is to use neural networks, since they are expected to be able to overcome the CoD. We use the networks as a regression, where we fit the underlying asset to the value of the PDE. The target values can be obtained from the boundary conditions of the PDE. The time is discretized and we can use the PDE to calculate the relation between the solution at different time steps. In this work, we use two methods, one method that is solved backwards in time (Deep Backward Dynamic Programming: DBDP) and another that solves the problem forward in time (deep Backward Forward Stochastic Differential Equation: dBSDE).

Both the DBDP and dBSDE algorithms have shown good results for PDE that can be associated with a simple Itô diffusion. However, this model may not describe reality sufficiently. Sudden large price movements, jumps, often occur in financial markets. For example, a stock can crash due to an economic crisis like in the financial crisis in 2008 or the COVID pandemic. Theoretical and Empirical experiments have demonstrated that these jumps can have a significant impact on various financial problems, like option pricing and risk management. We can simulate these jumps with a Lévy process. The problem becomes a lot more difficult, because we have an additional term to learn. Moreover, in most models, the jumps are modeled using a

distribution. This means that we have two extra types of randomness we have to take into account, namely when the jump happen and the size of these jumps. At last, the paths are not continuous when simulating jumps, which makes the underlying mathematics more complicated. In this work, we only use finite activity jumps. This means that we can turn the processes into a martingale by compensating the jumps. This compensation needs to be calculated analytically or simulated with a Monte Carlo simulation.



(a) Stock values for Unilever. source:[36]

(b) The price of the S&P 500 index.

Figure 1: Jumps happen frequently in the financial markets, for example, during the Asian crisis in 1997-1998 (a) or the 2008 financial crisis and COVID pandemic (b).

While the jumps make the problem more complicated, it still has similarities with the simple Itô-diffusion problems we could solve with the DBDP and dBSDE methods. This suggests that we only need to add extra terms to these algorithms. For the backwards DBDP method, the theoretical groundwork was already done in [17]. A new set of neural networks is introduced to describe the jumps, while the compensation part is obtained with an online Monte Carlo simulation. However, an online Monte Carlo simulation is inefficient and is often too slow, especially for large dimensional problems. Therefore, we change it into an offline Monte Carlo simulation. We test this algorithm by pricing Bermudan basket options. We were able to obtain good results for options with 50 underlying stocks.

Similarly, the dBSDE was adapted in [24] to allow jumps. Here we added an additional set of networks to emulate the compensation of the jumps. These networks need to be learned using a different loss function: the Jump Loss. The original loss function (Terminal Loss) and the Jump Loss are both minimized simultaneously by summing them up. This means that the loss functions try to find some "balance" and that they both are non-zero. We will show that this algorithm is able to price European options correctly, but it will be almost the same as a simple Monte Carlo simulation. The dBSDE-Jump methodology may fit non-initial times incorrectly. This means that it is questionable whether the dBSDE-Jump method is able to solve more complex problems. We argue that we can overcome this problem by either solving the problem in a hierarchical structure or by minimizing both loss function separately.

This work is structured as follows. First, we show some basic stochastic calculus. We use this to prove the Dynamic Programming Principle and to derive the HJB-equations. This is followed by a short explanation of the Curse of Dimensionality. In Chapter 3, we start with an explanation of neural networks. This is then followed by a description of the DBDP and dBSDE algorithms. At last, we show some arguments why neural networks are able to overcome the CoD. In the next chapter, we extend the results from chapters 2 and 3 to jump processes. First, we show some stochastic calculus with jumps, followed by the description of the DBDP and dBSDE with jump methods. This includes the explanation of why the dBSDE-Jump algorithm may not work as intended. At last, the convergence of the DBDP method is shown. Chapter 5 shows the results.

2 The Hamilton-Jacobi-Bellman equation and the curse of dimensionality

In this section, we derive the Hamilton-Jacobi-Bellman equation (HJB-equation). To do this, we first recall some basic stochastic calculus and derive the Itô lemma for Itô-diffusion processes. We continue by defining the optimal control problem. We transform this problem into a backwards recursive relation with the dynamic programming principle. This allows us to derive the HJB-equation. This PDE can easily be solved in low dimensions. In the final section, we show why it is difficult to solve it for higher dimensions. All proofs from this section can be found in Appendix A.

2.1 Basics of Stochastic Calculus

In this section, we derive the Itô lemma for simple diffusion processes. We do this using the quadratic variation:

Definition 1. *Let $0 = t_0 \quad t_1 \quad {}_N = t$ be a partition of [0,t]. Let h be the maximum distance between t_i, t_{i+1}. Then the quadratic (co)variance is*

$$< X, Y >_t = \lim_{h \,!\, 0} \sum_i (X_{t_i} \quad X_{t_{i-1}})(Y_{t_i} \quad Y_{t_{i-1}})$$

In the case of Brownian motion W, we have $\mathsf{E}((W_{t_i} \quad W_{t_{i-1}})^2) = t_i \quad t_{i\ 1}$ and $\mathsf{V}ar((W_{t_i} \quad W_{t_{i-1}})^2) = 2(t_i \quad t_{i\ 1})^2$. This suggests that $< W, W >_t = t$ and similarly $< t, W >= 0$. This is summarized in the Itô table

	dt	dW
dt	0	0
dW	0	dt

From the Itô table and using Taylor's expansion theorem, we can prove Itô lemma. We now state Itô formula:

Theorem 1. *Let X_t be a process. Let f(t,x) be twice di erentiable and $Y_t = g(t, X_t)$. Then*

$$dY_t = \frac{\partial f}{\partial t} dt + \frac{\partial f}{\partial x} dX + \frac{\partial^2 f}{\partial x^2} d < X, X >$$

Which gives us the Itô lemma for the Itô-diffusion process.

Corollary 1.1. *Let X_t be an Itô di usion process*

$$dX_t = \mu(t, x)dt + \sigma(t, x)dW_t$$

Let f(t,x) be twice di erentiable and $Y_t = g(t, X_t)$. Then

$$dY_t = (\frac{\partial f}{\partial t} dt + \mu \frac{\partial f}{\partial x} + \sigma \frac{\partial^2 f}{\partial x^2})dt + \sigma \frac{\partial f}{\partial x} dW_t$$

We can use this theorem to derive the infinitesimal generator of a diffusion process. This generator will appear in the HJB equation.

Lemma 2. *The in nitesimal generator given by*

$$\mathcal{L}f = \lim_{\Delta t \# 0} \frac{\mathsf{E}_x(f(X_{t+\Delta t})) \quad f(X_t)}{\Delta t}$$

of an Itô di usion process is

$$\mathcal{L}f = (\frac{\partial f}{\partial t} + \mu \frac{\partial f}{\partial x} + \sigma^2 \frac{\partial^2 f}{\partial x^2})$$

2.2 The Hamilton-Jacobi-Bellman equation and the dynamic programming principle

2.2.1 Dynamic Programming Principle

In this section, we show the Bellman equation or Dynamic Programming Principle (DPP). The DPP allows us to rewrite a problem backward and recursively. In the current application, we want to optimize actions for a Markovian process. Therefore, we pick a process $X_{t_0}^{t,x}$ with a cost function as follows:

- Markovian process $X_{t_0}^{t,x}$, where $t_0 \quad t \quad T$ and $X_{t_0}^{t_0,x} = x$
- $J(t,x,\alpha) = \mathsf{E}[g(X_{t_0}^{T,x}) + \int_{t_0}^{T} l(s, X_{t_0}^{s,x}, \alpha_s)ds]$
- $v(t,x) = \inf_{a \, 2 \, \mathcal{A}} J(t,x,a)$

Here a show the results from our actions. We have different possible actions, which are restricted to an action space $\mathcal{A}$. The goal is to minimize the cost function J, which consist out of a terminal cost(g) and some instantaneous or holding cost (l). We rewrite the cost function J by stopping the process at τ. This makes it possible to express the cost function recursively:

Theorem 3. *For all stopping times τ 2 $[t_0, T]$,*
$J(t,x,\alpha) = \mathsf{E}[J(\tau, X_{t_0}^{\tau,x}, \alpha) + \int_{t_0}^{\tau} l(s, X_{t_0}^{s,x}, \alpha_s)ds]$

The DPP is similar to the equation above, except here we use the optimal control. We state the DPP and use theorem 3 to prove this result.

Theorem 4. *Let X_t be a controlled Markov process, then for all τ 2 $[t_0, T]$*
$v(t_0,x) = \mathit{inf}_{a \, 2 \, \mathcal{A}} \mathsf{E}[\int_{t_0}^{\tau} l(s, X_{t_0}^{s,x}, \alpha_s)ds + v(\tau, X_{\tau}^{t_0,x})]$

This theorem shows that we solve optimal control problems recursively. We split the problem in two parts by setting a stopping time τ. Theorem (4)shows that obtaining $v(t_0,x)$ requires $v(\tau, X_{\tau}^{t_0,x})$, which is the same problem on a smaller time-interval. On top of that, we need to solve another optimization problem: minimize the right-hand-side of theorem (4). By splitting the problem multiple times, we note that we can solve the problems backwards.

2.2.2 Deterministic Hamilton-Jacobi-Bellman equation

We can derive a PDE from a controlled stochastic process. The idea is to use the DPP. The unused terminal cost term will act as the terminal condition for the PDE. First, we show the derivation for a HJB equation of a deterministic process:

Let there be a controlled deterministic process

$$dX_t = b(X_t, \alpha_t)dt$$

Here α_t is the control defined on a convex set A. An agent tries to minimize a cost function $J(a)$ based on this deterministic process.

$$J(a) = g(x_T) + \int_{t_0}^{T} l(x_s, \alpha_s) ds$$

With t_0 the initial time and T the terminal time. The cost function is split into a terminal cost ($g(X_T)$) and a cost for the duration for the process ($l(x_t, \alpha_t)$. We denote the wanted minimum cost as $u(t_0, x)$:

$$u(t_0, x) = \inf_{a 2 A} g(x_T) + \int_{t_0}^{T} l(x_s, \alpha_s) ds$$

We use the dynamic programming principle (DPP) followed by a Taylor expansion:

$$u(t_0, x) = \inf_{a 2 A} u(t, x) + \int_{t_0}^{t} l(x_s, \alpha_s) ds$$

$$u(t_0 + \Delta t, x(t_0 + \Delta t) = u(t_0, x(t_0)) + \frac{\partial u}{\partial t}\Delta t + \frac{\partial u}{\partial x}\Delta x_t + o(\Delta t)$$

$$= u(t_0, x(t_0)) + \frac{\partial u}{\partial t}\Delta t + \frac{\partial u}{\partial x} b(a, \alpha)\Delta t + o(\Delta t)$$

We substitute this into the DPP for $t = t_0 + \Delta t$

$$u(t_0, x) = \inf_{a 2 A} u(t_0), x(t_0)) + \frac{\partial u}{\partial t}\Delta t + \frac{\partial u}{\partial x} b(a, \alpha)\Delta t + \int_{t_0}^{t} l(x_s, \alpha_s) ds$$

$$0 = \inf_{a 2 A} \frac{\partial u}{\partial t}\Delta t + \frac{\partial u}{\partial x} b(a, \alpha)\Delta t + \int_{t_0}^{t} l(x_s, \alpha_s) ds$$

We divide the equation above by Δt and take its limit to zero, this gives

$$0 = \inf_{a 2 A} \frac{\partial u}{\partial t} + \frac{\partial u}{\partial x} b(a, \alpha) + l(x, \alpha_t)$$

This is the Hamilton-Jacobi-Bellman equation (HJB) of this process.

2.2.3 General Hamilton-Jacobi-Bellman equation

Deriving the HJB equation for a general Itô-diffusion is similar as in the previous section. This results in the following theorem:

Theorem 5. *Let X_t be a Feller (and therefore Markovian) process. Take the cost function*

$$J(a) = \mathsf{E}_x(g(x_T) + \int_{t_0}^{T} l(x_s, \alpha_s) ds)$$

, where

$$\mathsf{E}_x(\ ) = \mathsf{E}(\ jX_0 = x)$$

Then this is equivalent to the PDE

$$inf_{a 2 A} L u(t, x) + l(x_t, \alpha_t)) = 0$$

where L the in nitesimal generator given by

$$L f = \lim_{\Delta t \# 0} \frac{\mathsf{E}_x(f(X_{t+\Delta t})) \quad f(X_t)}{\Delta t}$$

Theorem (5) suggest that we can derive the HJB equation quickly when we can obtain the infinitesimal generator of the process. This generator for the Itô diffusion process was already shown in lemma (2).

2.3 Curse of dimensionality

In this work, we will solve high dimensional PDEs. For low dimensions, we can solve such a PDE with for example, the finite difference or finite elements method. Alternatively, we can use methods that use bundling or basis functions [12, 19, 44]. These methods require a discretization in X. This will lead to an error decay of order $\epsilon^{\mathcal{O}(\frac{1}{d})}$. So for high dimensions, we will need a very fine discretization to solve the PDEs. Therefore, these methods cannot solve high dimensional PDEs. This is called the Curse of Dimensionality (CoD) [9].

One method that is in some cases able to overcome the CoD is the Monte Carlo algorithm. Currently the Longstaff-Schwarz algorithm ([33]) is used to price Bermudan options. This algorithm requires a regression at every exercise opportunity. This is inefficient for high dimensions. Moreover, Monte Carlo algorithms solve the PDE at a single point X_i. If we want to obtain the solution over a region, we notice that we will need to discretize this region. This is again inefficient for high dimensions. In this work, we will use neural networks to solve PDEs. We will also show arguments why it may solve the CoD.

3 Numerical schemes for PDEs in stochastic control with Itô-processes

In this section we show how we can solve high dimensional HJB-equations. The main idea is that neural networks are likely to overcome the curse of dimensionality. Therefore, we start by introducing neural networks. We show that neural networks are able to approximate functions by showing the Universal Approximation Theorem. We can then use the neural networks in numerical schemes to solve PDEs. We discretize the time and show a method that solves the problem backwards through time and one that solves it forwards through time. We continue by showing the differences between the methodologies and quickly note two other neural network methods that can be used. In the final section, we give a short reasoning of why neural networks are able to overcome the curse of dimensionality.

3.1 Neural Networks

High dimensional numerical schemes to solve HJB-equations, like finite differences, suffer from the curse from dimensionality. We try to overcome this problem by introducing schemes that incorporate neural networks. In this section, we first describe describe the main idea behind neural networks. Afterwards, we show some arguments why neural networks can be free from dimensionality.

3.1.1 Feed Forward

Originally, neural networks were inspired by a model of the human brain. The idea was that the many connections between neurons should describe how the brain processes information. This was used to create neural networks. A perceptron or neuron uses simple operations to calculate an output. Many of these neurons are put in layers and these layers are connected in succession. The final layer gives the output of the network.

We describe the perceptron in more detail. It calculates a weighted sum of the output of the previous layer and an extra bias term. This bias term gives more flexibility to the network. For example, without bias, if we take $x_i = 0$, we get that the output always becomes $\sigma(0) = 0$. The summed output is mapped by some nonlinear function σ, the activation function. This gives the network some non-linearity and therefore allows the network to learn a non-linear target. In general, the output in neuron j is given by $\sigma(b_j + \sum_i w_i x_i)$. Next, we show the structure of a whole network. In the next figure, we show a single neuron in one of these layers.

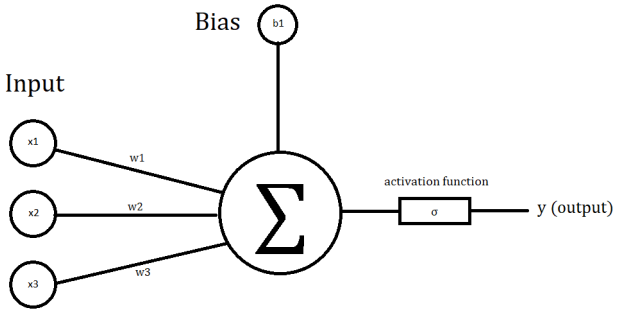


Figure 2: A single neuron, it has 3 inputs from neurons on the previous layer and a bias. This output is $\sigma(w_1x_1 + w_2x_2 + w_3x_3 + b_1)$.

Figure 2 shows a neuron with 3 neurons in the previous layer or as the input for the network. Its output is used in another layer of the network. This is seen in the figure 3.

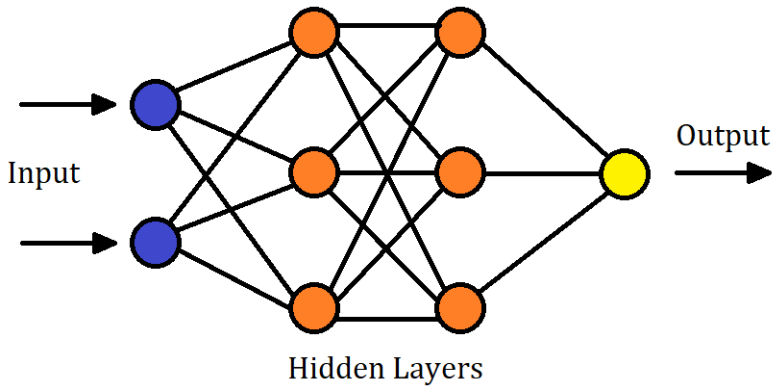


Figure 3: A neural network with a 2 dimensional input (blue) and a one dimensional output (yellow). It has 2 hidden layers with3 neurons each (orange).

We see that the input moves through 2 hidden layers before it gives an output. Writing the weights between layers as a matrix (W_i) and the biases as a vector (B_i), we can calculate this output by

$$A_L \quad \sigma \quad A_{L\ 1} \quad \dots \quad \sigma \quad A_1(x)$$

with

$$A_i(x) = W_i x + B_i$$

The output differs when using different weights and biases. The aim is to adapt the parameters W_i and B_i such that for all input x we get the desired output. This adaption is done using back propagation algorithms. From this point, we set all weights

and biases as $\theta(\theta_i$ per layer). This means that we only need to choose an activation function σ. Szandała [43] gives an overview of the most important functions. In this work, we will use the $Tanh$ and ELU activation function. The $Tanh$ activation function is used since the used algorithms emulate recurrent neural networks. When pricing options, we sometime use ELU, since this is a smooth activation function that is similar to pay-off functions.

3.1.2 Back-propagation

In this section, we describe how we can adept the parameters. For this, we use a loss function $ell(x, y)$, which is 0 when $x = y$. We will try to minimize this loss function. The most common method is Stochastic Gradient Descent (SGD). We calculate the gradient of the loss function. This is done backwards starting from the output and using the chain rule. We decrease the loss by taking step into the direction of the minimum. $\theta_{n+1} = \theta_n \quad \eta_n \, \mathsf{r}_\theta \ell(x, y)$. Here we get another hyper parameter η_n, which we call the learning rate. This leads to the following pseudo-code:

Algorithm 1 Stochastic Gradient Descent

1: initialize θ_0
2: **for** batch **do**
3: sample (x, y)
4: calculate output $\hat{y}$ from input x using the neural network
5: calculate the loss $\ell(y, \hat{y})$
6: adept the parameters: $\theta_{n+1} = \theta_n - \eta_n \nabla_\theta \ell(y, \hat{y})$
7: **end for**

SGD can be slow and it has a constant learning rate. The improved algorithm RMSProp uses an automatically adapting learning rate. The idea is that the learning rate should become smaller depending on how much the parameters have changed.

Algorithm 2 RMSProp

1: initialize θ_0 and let ϵ be some small positive number. Set $v_0 = 0$ and choose decay parameter β.
2: **for** batch **do**
3: sample (x, y)
4: calculate output $\hat{y}$ from input x using the neural network
5: calculate the loss $\ell(y, \hat{y})$
6: $v_t = \beta v_t \ _1 + (1 - \beta)(\nabla_\theta \ell(\mathbf{x},\mathbf{y}))^2$
7: adept the parameters: $\theta_{n+1} = \theta_n - \not{\mathsf{p}} \frac{\eta_n}{\sqrt{v_t + \epsilon}} \nabla_\theta \ell(y, \hat{y})$
8: **end for**

We can extend RMSProp by using the momentum of the previous movements. This means that we remember the previous gradients and use this information in the current step. This can increase the speed of this optimization problem. This algorithm is called ADAM and is the current state of the art for training parameters in neural networks.

Algorithm 3 ADAM

initialize θ_0 and let ϵ be some small positive number. Set $m_0 = v_0 = 0$ and choose decay parameter β_1 and β_2.
for batch **do**
    sample (x, y)
    calculate output $\hat{y}$ from input x using the neural network
    calculate the loss $\ell(y, \hat{y})$
    $v_t = \beta_1 v_t \ _1 + (1 - \beta_1)(\nabla_\theta \ell(y, \hat{y}))^2$
    $m_t = \beta_2 m_t \ _1 + (1 - \beta_2)\nabla_\theta \ell(y, \hat{y})$
    $\hat{v_t} = \frac{v_t}{1 \ \beta_1}$ ▷ Correct bias
    $\hat{m_t} = \frac{m_t}{1 \ \beta_2}$ ▷ Correct bias
    adept the parameters: $\theta_{n+1} = \theta_n - \frac{\eta_n}{\hat{v_t} + \epsilon} \hat{m}_t$
end for

3.1.3 Universal Approximation Theorem

A key part of the algorithms we will use is making neural networks approximate known functions. Therefore, we restate the Universal Approximation Theory (UAT), which shows that neural networks are able to do this.

First, we look into shallow neural networks. This simple neural network structure is well studied and various results for the UAT are proven [20, 30, 32]. We show a sketch of a proof that a shallow neural network with the sigmoid activation function is a universal approximator. A key idea is that we can use multiple activation functions to act as a step- or block- function. This requires appropriate properties on the activation function. We can approximate a function with a simple function (see Lesbeque integration), which means that we can use the block-functions to approximate any function.

While we are able to approximate any function with shallow neural networks, but this requires a lot of neurons. Empirical results show that deep neural networks are more efficient in approximating functions. There already exist some results that show that deep networks are more efficient than shallow networks [34, 46].

3.2 Numerical schemes

In this section, we describe how we can solve high-dimensional HJB-PDE using neural networks. To do this, we discretize the time and use neural networks to approximate the solution at each time step. The function we want to approximate can be calculated by using a different time step and the forward-backward stochastic differential equation (FBSDE). We can easily derive this FBSDE by using Itô lemma. The structure of this section is as follows: First, we start by deriving the FBSDE. Afterwards, we describe the two algorithms that we will use, DBDP and dBSDE. In the final section, we show some other methods that use neural networks, but will not be used in this work.

We will derive three different FBSDEs, which corresponds to different PDEs. First, we derive the classic FBSDE for nonlinear parabolic PDEs. We continue with the derivation of the FBSDE of fully nonlinear PDEs. Afterwards, we show the 2BSDE, an alternative way to obtain a fully nonlinear FBSDE.

We start with a simple Itô diffusion process

$$dX_t = \mu(t, X_t)dt + \sigma(, X_t)dW_t. \tag{1}$$

We have the nonlinear PDE

$$\frac{\partial u}{\partial t} + \mathcal{L}u + f(t, x, u, \nabla_x u) = 0, \tag{2}$$

$$u(T, x, u, \nabla_x u) = g(x, u, \nabla_x u), \tag{3}$$

where $\mathcal{L}$ the infinitesimal generator of the Itô process given by Equation (1). We denote the following doublet of parameters (Y_t, Z_t):

$$Y_t = u(t, X_t) \qquad Z_t = \nabla_x u(t, X_t) \tag{4}$$

Next, we use Itô formula (Y_t on Itô diffusion (Equation 1)):

$$dY_t = [\frac{\partial u}{\partial t} + \mu(t, X_t)\nabla_x u + \frac{1}{2}\mathrm{tr}(\sigma\sigma^T \Delta_x u)]dt + \sigma(t, X_t)\nabla_x u dW_t. \tag{5}$$

Substituting the PDE (Equation (2)) gives:

$$dY_t = \quad f(t, x, y, z)dt + \sigma(t, X_t)Z_t dW_t \tag{6}$$

At last, we only need to integrate this expression to obtain the FBSDE:

$$Y_t = g(X_T) + \int_t^T f(t, x, y, z)ds \quad \int_t^T \sigma(s, X_s)Z_s dW_s \tag{7}$$

.

If we have a control on the volatility term, we get an HJB-PDE that cannot be described by Equation (2). Therefore, we quickly show the FBSDE when using a fully nonlinear PDE:

$$\frac{\partial u}{\partial t} + f(t, x, u, \nabla_x u, \Delta_x u) = 0. \tag{8}$$

The main difference is that the Hessian is part of the PDE. Moreover, the infinitesimal generator is already described in $f(t, x, u, \nabla_x u, \Delta_x u)$. We denote the following triplet of parameters (Y_t, Z_t):

$$Y_t = u(t, X_t) \qquad Z_t = \nabla_x u(t, X_t) \qquad \Gamma_t = \Delta_x u(t, X_t) \tag{9}$$

We can now get the FBSDE by following the same steps as before. This leads to the following FBSDE:

$$Y_t = g(X_T) \quad \int_t^T [\mu(s, X_s)Z_s + \frac{1}{2}\mathrm{tr}(\sigma\sigma^T \Gamma_s) \quad f(t, x, y, z)]ds \quad \int_t^T \sigma(s, X_s)Z_s dW_s \tag{10}$$

There is also an alternative way to write this FBSDE, namely the 2BSDE. This 2BSDE has 2 forward components. One of these components is the already derived FBSDE (Equation (10)). The goal is to obtain a second forward equation such that the triplet (Y_t, Z_t, Γ_t) is unique. To do this, we use similar steps on $Z_t = \nabla_x u$. First we use Itô lemma:

$$dZ_t = \nabla_x z dX_t + [\frac{\partial z}{\partial t} + \frac{1}{2}\mathrm{tr}(\sigma\sigma^T \Delta_x u)]dt. \tag{11}$$

Note that the first term on the right-hand-side gives the Hessian. Now write

$$A_t = \frac{\partial}{\partial t}\frac{\partial u}{\partial x} + \mathcal{L}(\frac{\partial u}{\partial x}). \tag{12}$$

This gives the 2BSDE:

$$Y_t = g(X_T) \quad \int_t^T [\mu(s, X_s)Z_s + \frac{1}{2}\mathrm{tr}(\sigma\sigma^T \Gamma_s) \quad f(t, x, y, z)]ds \quad \int_t^T \sigma(s, X_s)Z_s dW_s \tag{13}$$

$$Z_t = Z_0 + \int_0^t A_s ds + \int_0^t \Gamma_s dX_s \tag{14}$$

3.2.1 Deep learning Backward Dynamic Programming (DBDP)

In this section, we describe the Deep learning Backward Dynamic Programming (DBDP) algorithm. This method was first explored by Huré et al. [31], where they explain it for a simple nonlinear PDE (equation (2)). This means that we use the FBSDE given by equations (1) and (7). First, we discretize the FBSDE with the Euler-Maruyama method:

$$X_{t_{i+1}} = X_{t_i} + \mu(t, X_{t_i})\Delta t_i + \sigma(t, X_{t_i})\delta W_{t_i}. \tag{15}$$

$$Y_{t_{i+1}} = Y_{t_i} \quad f(t_i, x, y, z)\Delta t_i + \sigma(t_i, X_{t_i})Z_{t_i}\Delta W_{t_i} = F(t_i, X_{t_i}, Y_{t_i}, Z_{t_i}, \Delta t_i, \Delta W_{t_i}). \tag{16}$$

The idea is to use neural networks for all Y_{t_i} and Z_{t_i}. When using enough neurons, the universal approximation theorem holds and therefore the networks are able to act as the desired functions.

As the name implies, the algorithm goes backward in time. This means that we first need to learn the terminal networks, which can be done using the terminal condition. This problem is the same as standard regression using neural networks. We can now calculate the wanted output of the networks a time step earlier by using the FBSDE (equation (16)) and the networks of the final layer. We continue this for each time step. This gives the following pseudocode given in Algorithm 4:

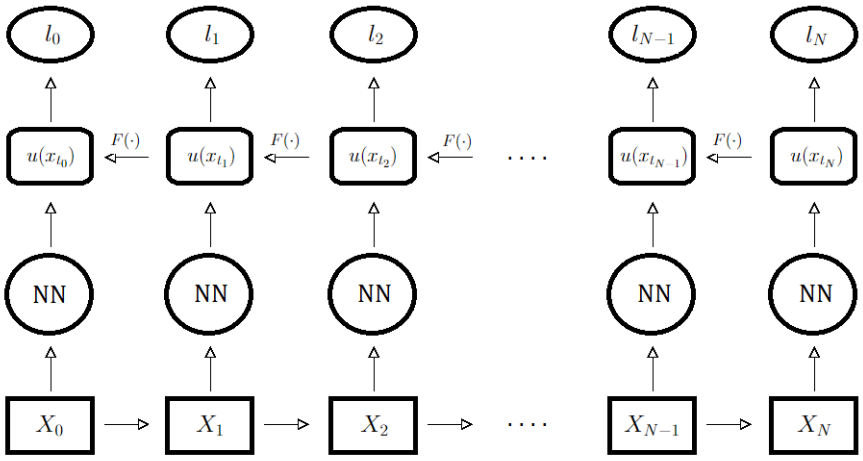


Figure 4: The schematic overview of the DBDP algorithm

Algorithm 4 DBDP

train $\mathcal{U}_N$ and $\mathcal{Z}_N$ by minimizing $\mathsf{E}(\mathcal{U}_N - g(x_{t_N}))^2$ and $\mathsf{E}(\mathcal{Z}_N - Dg(x_{t_N}))^2$
for $i \leftarrow N-1$ to 0 **do**
train $\mathcal{U}_i$ and $\mathcal{Z}_i$ by minimizing $\mathsf{E}(\mathcal{U}_{i+1} - F(t_i, X_{t_i}, \mathcal{U}_i, \mathcal{Z}_i, D\mathcal{Z}_{i+1}, \Delta t_i, \Delta W_{t_i}))^2$
end for

This algorithm has been extended to the fully nonlinear case by Pham et al. [39]. Here they use the FBSDE representation as in equation (10). The Hessian is calculated using automatic differentiation of the Z networks. This guarantees that the triplets $(\mathsf{U}_i, \mathsf{Z}_i, \Gamma_i)$ give unique solutions. For some complex PDEs, the Hessian can become very oscillating at the edge of the attained domain. These oscillations can propagate and lead to a wrong solution or even non-convergence. They solve this by truncating the Hessian to make sure the oscillations cannot become too large. A multistep version of the DBDP algorithm was proposed by Germain et al. [23]. The main difference is that in the simple DBDP method, we learn U_i only from U_{i+1}. The multistep method learns U_i from $\mathsf{U}_{i+1}, \mathsf{U}_{i+2}, \mathsf{U}_{i+3}, \ldots \mathsf{U}_{N\ 1}$ and U_N. The idea is that this reduces error propagation. This means that we are able to obtain a more accurate solution. Moreover, it is able to solve fully nonlinear PDEs.

3.2.2 deep Backward Stochastic Differential Equation (dBSDE)

Some problems do not follow the DPP and cannot be solved by the DBDP method [3]. Also, the DBDP is a backward algorithm and this can make solving certain problems more complicated. Therefore, we introduce a different method to solve HJB-equations: the deep Backward Stochastic Differential Equation (dBSDE) algorithm as introduced by Han et al. [28]. This method solves the corresponding FBSDE in a forward direction, whereas the DBDP does this in a backward direction. Similar as in the DBDP case, we use the Euler-Maruyama method to discretize the FBSDE (equation (7)). We will then, for every time step, use a neural network Z_i to emulate the Z-component. We also let Y_0 be a trainable parameter. We can now use the Euler discretization and Z_i to calculate Y_{t_i} for $t_i > 0$. We then compare the output after all the steps with the terminal condition. This leads to the following pseudocode in Algorithm 5:

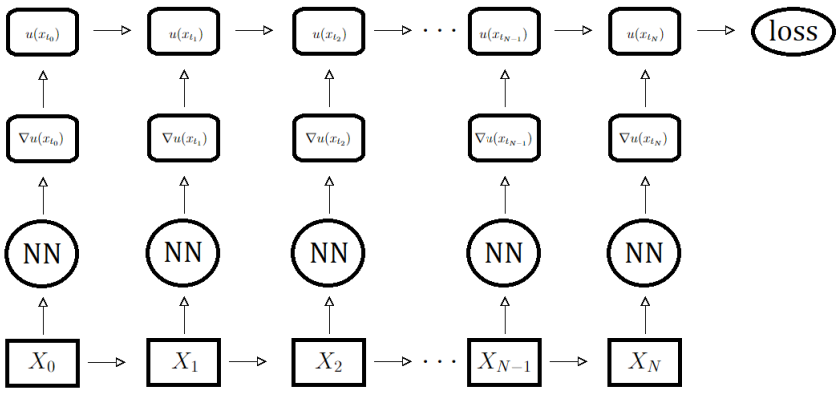


Figure 5: The schematic overview of the dBSDE algorithm

Algorithm 5 dBSDE

initialize networks $\mathcal{Z}_i$ and Y_0.
for iteration **do**
  for $i \leftarrow 0$ to $N-1$ **do**
    Calculate $Y_{i+1} = F(t_i, X_i, Y_i, \mathcal{Z}_{t_i}, \Delta t_i, \Delta W_{t_i})$
  end for
  minimize $\mathsf{E}(Y_N - g(T, X_N))^2$
end for

We can extend this method to solve fully nonlinear PDEs. This was done by Beck et al. [8] by using the 2BSDE (equation (14)).

3.2.3 Comparing the dBSDE and DBDP methods

The main difference between these two methods is that the DBDP goes backward through time, while the dBSDE goes forward. Some problems are easier to solve by choosing the correct direction in time. This is especially true for problems that do not follow the DPP. The DBDP method is not able to solve these problems directly while the dBSDE method can. However, even if the problem follows the DPP, there can still be an algorithm that solves the problem in a more direct and convenient way. For example, coupled FBSDEs can be simulated easily by the forward dBSDE method. The backward DBDP methodology requires us to calculate the Y_i components implicitly. Similarly, barrier or Asian options can be priced easier by using the forward method. In barrier options, we can remember whether the barrier has been reached before and use it directly in the dBSDE networks, see [22]. If we want to price barrier options with the DBDP method, we need a different way to account for the barrier. One way is to calculate the chance of breaking the barrier using a Monte Carlo simulation. This is less efficient than using the dBSDE method. Moreover, we need to use a Brownian bridge construction for this Monte Carlo simulation. For more complex processes, there may not be an easy way to find an equivalent process to this

Brownian bridge. On the other hand, backward processes are able to price American options efficiently. We can rewrite the problem into a DPP-like problem. This means that we will compare the price at the current time to the next (possible) exercise date. This is not possible with forwards methods like the dBSDE.

There are also some other differences between the methods. [1, 23, 31] argue that the DBDP is generally more likely to converge to the desired functions. The dBSDE is similar to a recurrent neural network, which makes it more vulnerable for exploding and vanishing gradients. Changing the algorithm slightly, like [2], may solve this issue. Another advantage for the DBDP method is that the solution is easily obtainable for all time steps. The dBSDE only gives a clear solution for the initial time. For all other time steps, we need all X_i until this time step. Since these are random variables, the obtained solution is also a random variable. Moreover, if the underlying process is complex, we may not be able to create a Brownian bridge like path. This means that we need to find a different way to get the solution at a certain point.

However, the dBDSDE methodology also has an advantage over the DBDP method. In the dBSDE method, all time steps are learned at the same time. Therefore, we can continuously refine the networks. We can check the current loss and decide to continue training. It even allows us to choose a different learning rate if we notice that the networks are not learning correctly. In contrast to the simultaneously learned dBSDE networks, the DBDP method is solved consecutively. This means that we need to stop learning a networks after a certain amount of iterations, which lead to an error. This error propagate through the next networks and can lead to an inaccurate solution. Moreover, when we refine this later network to obtain a more accurate approximation, we will also need to relearn all the earlier other networks.

3.2.4 Other neural network algorithms

DBDP and dBSDE are not the only algorithms that have been developed. In this section, we briefly mention 2 other methods that solve HJB equations using neural networks.

First, we show the Deep Galerkin Method (DGM) [42]. This algorithm does not need a time discretization and is inspired by Physics-Informed Neural Networks (PINN). We let $f(t, x; \theta)$ be a neural network. For the DGM, we sample 3 different type of points:

1. (t_n, x_n) $\mathcal{2}$ $([0, T], \Omega)$: these are the point that describe the PDE. We calculate the loss that corresponds to these points with $\mathsf{E}(\frac{\partial f}{\partial t}(t_n, x_n) + \mathcal{L} f(t_n, x_n))^2$. The derivative and $\mathcal{L}$ can be calculated using automatic differentiation.

2. (τ_n, z_n) $\mathcal{2}$ $([0, T], \partial\Omega)$: These are the boundary points. The loss is calculated as: $\mathsf{E}(f(\tau_n, z_n) \quad g_{\text{boundary}}(\tau_n, z_n))^2$.

3. w_n $\mathcal{2}$ Ω: This is the terminal condition. This has $\mathsf{E}(f(T, w_n) \quad g(w_n))^2$ loss.

In the DGM, we generate the 3 different type of points and sum the (weighted) losses. We then use backward propagation to adapt θ. To make sure that the terminal and boundary conditions are learned correctly, we need to weight the losses. This makes the DGM hard to train. The advantage of this method is that we only need 1 network and there is no time discretization error.

Another method is deep Splitting (DS). This method is similar as the DBDP. It uses a time discretization and the FBSDE is solved backward. As with DBDP, DS uses a new set of neural networks at all t. It differs from DBDP by using a regression argument in the FBSDE. We write out the argument we use to learn network U_i for both the DBDP and the DS methods.

$$\begin{aligned} \text{DBDP: } \mathsf{E}(\mathsf{U}_{i+1}(X_{i+1}) \quad \mathsf{U}_i(X_i) \quad f(t_i, X_i, \mathsf{U}_i, \mathsf{Z}_i)\Delta t \quad \mathsf{Z}_i(X_i)\Delta W_i)^2 \\ \text{DS: } \mathsf{E}(\mathsf{U}_{i+1}(X_{i+1}) \quad \mathsf{U}_i(X_i) \quad f(t_i, X_i, \mathsf{U}_i, D_x\mathsf{U}_i)\Delta t)^2 \end{aligned}$$

So the $\mathsf{Z}\Delta W_i$ term does not appear in the deep splitting method, while it does in the DBDP. Moreover, we cannot use a second type of networks for the derivatives (Z), it needs to be calculated using automatic differentiation. There also exists a DBDP version that uses automatic differentiation [31], but it will not be used in this work.

At last, we note a possible extension to the algorithms. Solving an HJB-equation requires the solution of an optimization problem and the solution of a PDE. We solve the HJB equation by finding an analytic solution to the optimization problem and substituting it in the HJB to obtain a simple PDE. Finding an analytic function is not always easy and using a numerical scheme needs to overcome the curse of dimensionality. [4] solves this problem for the DGM. They solve the optimization problem with a Policy Improvement Algorithm (PIA). They alternatively take a back propagation step between the DGM algorithm and the PIA, creating the hybrid DGM-PIA algorithm. This idea should also be useful for solving HJB-equation without the analytic solution of the optimization problem with the dBSDE, DBDP and DS algorithms.

3.3 Connection between the HJB-equation and neural networks

In this section we describe why neural networks are able to solve high dimensional HJB-equations. Darbon et al. [21] show that some HJB-equations can be written as a special type of neural network, the dynamics of the HJB-PDE are directly described by neural networks. They show that we can approximate the wanted functions using neural networks without relying on the universal approximation theorem. However, we will try to solve more general HJB-equations and cannot initialize the data as described by Darbon et al. [21] Moreover, we will use a $Tanh$ or ELU activation function instead of $argmax$. Therefore, we require other reasons why neural networks can solve high dimensional problems. Proving this is out of the scope of this work, but we will give arguments why it is reasonable to suspect that neural networks can overcome CoD.

3.3.1 Barron Space

First, we show the argument that shallow neural networks are CoD free using Barron spaces. The idea is to write out the whole architecture of this network. We introduce a Barron norm that uses this architecture formula. Then we can show that there exist some functions in this Barron norm and that they can be approximated CoD free. We use the ReLU activation function as an example.

We choose a shallow neural network with the ReLU activation function and n neurons. This can be represented as $\phi(x) = \sum_i w_{1,i}\sigma(w_{2,i}x + b_i)$. Using an appropriate

probability density μ, we can describe this as:

$$\phi_n(x) = \frac{1}{n} \int w_1 \sigma(w_2 x + b) d\mu(w_1, w_2, b) = \mathsf{E}(w_1 \sigma(w_2 x + b)). \tag{17}$$

We introduce the Barron norm for shallow ReLU networks.:

$$jj\phi jj_{\mathcal{B}_p} = \inf_\mu(\mathsf{E}(jw_1 j^p (jjbjj_1 + w_2)^p))^{\frac{1}{p}}, \text{where } 1 \quad p \quad 1 . \tag{18}$$

The Barron Space consist of all functions with a finite Barron norm:

$$\mathcal{B} = f\phi : jj\phi jj_{\mathcal{B}_p} < 1\, g. \tag{19}$$

Not every function has a finite Barron norm. But we will need a finite Barron norm to show that it is possible to approximate this function CoD-free. For this example, we have a finite Barron norm when the following holds:

$$\gamma(g) = \int jjwjj_1^2 j\hat{g}(w) dw < 1 , \tag{20}$$

where $\hat{g}$ the Fourier transform of $g(x)$.

Theorem 6. *If $\gamma(g) < 1$, than there exist a constant c such that $jj\phi_n(x) \quad g(x)jj^2 \frac{cjjgjj_{\mathcal{B}}}{n}$*

The proof of this theorem and other Barron spaces can be found in [7, 11, 15]. It has been shown that for function in the Barron space, it is also learnable without CoD [14]. However, for high dimensions, there are functions that cannot be approximated well by Barron functions, even though they are Lipschitz-continuous [45]. These functions are inefficient to learn and may suffer from the CoD.

Not every problem has been successfully described using Barron spaces, Therefore, we also look at some other reasons why neural networks may overcome the CoD.

3.3.2 Other Arguments

First, we argue the usefulness of deep neural networks. [40] show that in some classes deep neural networks are exponentially better than shallow neural networks. In [40] it is shown that some subclasses of compositional functions (like hierarchical functions) can be regressed CoD free using convolutional neural networks. The main idea is that hierarchical functions can be written in a tree-like structure. Shallow neural networks cannot use this information, while deep neural networks can. These deep neural networks can restructure itself to such a tree-like graph, assuming it has enough width and depth. It does not need to approximate the tree exactly. We can use a different graph in the neural network if this leads to a small enough error. The small subgraphs are neural networks on its own, which means that we can approximate this function with a lower effective dimension. This shows that it can be used to solve high dimensional financial problems. Notice that the payoff of a simple European option can be described by a hierarchical function. This allow us to use DBDP or dBSDE to price options with a lower effective dimension.

A lower effective dimension can also be obtained in a different way. The idea is that we can rewrite the data as if they lay on a lower dimensional manifold. We can solve this problem using finite differences on this lower dimensional manifold. This can, depending on the problem, effectively be emulated by a neural network. Theoretical results have been in shown in [11, 18, 41].

Not all problems fit the requirements in the arguments above. A useful method is to show that a problem is similar or the same as a different problem which has been shown to be CoD-free. For example, [27] shows that the Black-Scholes PDE can be solved by neural networks CoD-free. The proof uses that the drift rate μ and volatility σ of the paths are affine and they relate it to Monte Carlo. For a different example, [26] shows that HJB equations can be solved CoD-free if we take some restrictive assumptions.

4 Jump Diffusion

In this section we extend the work of the previous chapter to Jump processes. We start by defining the Lévy process and derive its corresponding Itô lemma. Afterwards, we extend the DBDP and dBSDE methods. The DBDP method is easily adapted by emulating the jumps with the U-networks and an offline Monte Carlo simulation. However, we cannot extend the dBSDE method in a similar way. We add a new set of neural networks to simulate the jumps. These networks need to be learned, which means that we will get a second loss function. We argue that when we solve them hierarchically or simultaneously, we will get the desired solution. However, some work was already done by minimizing the sum of these loss functions. We show why summing these losses can lead to wrong solutions, but that it is always to price European options correctly. At last, we show that the FBSDE with jumps (FBSDEj) exist and is unique. Moreover, we show results for the convergence of the DBDP method with jumps.

4.1 HJB for jump processes

In this section we show how we can derive HJB-equations for jump (Lévy) processes. We will see that these processes can be split in a continuous part (the Itô diffusion part) and a jump-only part. We continue by deriving the Itô lemma for Lévy processes. This allows us to form a PDE from this Lévy process and to derive its infinitesimal generator. We use this to derive the HJB equation in the same way as we derived the Itô diffusion case.

4.1.1 Lévy Process

First we show the definition of a Lévy process and the definition of a jump in this Lévy process.

Definition 2. *A process η_t that is adapted to its Itration is a Levy process if it satis es the following conditions*

1. η_0 = 0 *almost surely*
2. η_t *is cadlag in probability, e.g. it is right continuous with left limits*
3. η_t *has stationary increments,* $\eta_t \quad \eta_s \overset{d}{=} \eta_{t \quad s}$
4. η_t *has independent increments*

Definition 3. *A jump in a Levy process η_t is given by*

$$\Delta\eta_t = \eta_t \quad \eta_t$$

To simulate the Lévy process, we need to sum the jumps in a certain time interval. The jumps all have a certain size, which can be random or deterministic. In this work, we choose random jump sizes, so the jumps are described using a distribution. We will integrate functions with respect to jumps. This requires us to have a measure to sum the amount of jumps, independent of their size. This is done in the Poisson Random Measure (PRM, jump measure).

Definition 4. *Let U 2 B_0, where B_0 be the Borel sets without 0 in its closure. The Poisson Random Measure (PRM) is given by*

$$N(t,U) = N(t,U,\omega) = \sum_{0<s \quad t} \mathsf{X}_U(\Delta\eta_s)$$

The PRM gives the amount of jumps till time t. From now on, we will not specify that jumps of size 0 are not allowed. Generally all jumps appear in R^d. We use the PRM to define the Lévy measure.

Definition 5. *The Levy measure of η_t is given by*

$$\nu(U) = \mathsf{E}(N(1,U))$$

We will use the Poisson Process to simulate when there is a jump. We will see that this allows us to rewrite any Lévy process with finite Lévy measure in a process that uses the Poisson process. We state the Poisson Process and use this to define the Compound Poisson Process.

Definition 6. *The Poisson process P(t) is a Levy process with intensity factor λ with values in* N *[f*0*g given by*

$$\mathsf{P}(P(t) = n) = \frac{(\lambda t)^n}{n!} e \quad ^{\lambda t}$$

Definition 7. *Let μ_X be a distribution. Now let X(1), X(2) ... X(P(t)) be i.i.d. random variables distributed from μ_x and $P(t)$ an independent Poisson Process. The compound Poisson process is then given by*

$$Y_t = \sum_{i=0}^{P(t)} X(i) = \int_0^T \int_{\mathsf{R}^d} zN(dt,dz)$$

Note that the compound Poisson Process is a Lévy process. From this point, we will assume that the jump process is a compound Poisson process. Therefore, it is useful to calculate the Lévy measure of this process. We also define the compensated PRM, which is a martingale.

Theorem 7. *The Levy measure of a compound Poisson process is $\lambda\mu_x$*

Definition 8. *The compensated PRM is de ned as*

$$\tilde{N}(dt,dz) = N(dt,dz) \quad \nu(dz)dt$$

When the PRM converges, we say that it is a finite activity jump process. If it does not converge, it is an infinite activity process. In the following theorem, it is shown that any Lévy process can be decomposed in a sum of finite and infinite activity processes. In this work, we will only try to solve the finite activity case, but we will show Itô lemma for the general Lévy process. First, we write the Lévy-Itô decomposition.

Theorem 8. *Let R 2* R*. We can decompose a Levy this process in*

$$\begin{aligned} dX_t &= \mu(t,x)dt + \sigma(t,x)dW_t + \int_{jjzjj<R} \gamma(s,z,x)\tilde{N}(dt,dz) + \int_{jjzjj \quad R} \gamma(s,z,x)N(dt,dz) \\ &= \mu(t,x)dt + \sigma(t,x)dW_t + \int_{\mathsf{R}} \gamma(s,z,x)\bar{N}(dt,dz) \end{aligned}$$

[35] claims that this decomposition is always possible for $R = 1$. In our case, we will only use finite activity jumps. This corresponds with $R = \mathbf{1}$. Note that compensating the Poisson process will give a different Itô lemma. If we want the infinite activity case or do not discount the jumps, we have to set $R = 0$. We now show Itô lemma for Lévy processes.

Theorem 9. *The Itô formula of a Levy process as described in theorem (8) is*

$$\begin{aligned} df(X_t) = &(\frac{\partial f}{\partial t} + \mu\frac{\partial f}{\partial x} + \frac{1}{2}\sigma^2\frac{\partial^2 f}{\partial x^2})dt + \sigma\frac{\partial f}{\partial x}dW_t \\ &+ \int_{\mathsf{R}} f(X_t \quad + \gamma(t,z,x)) \quad f(X_t \quad))\bar{N}(dt,dz) \\ &+ \int_{jjzjj<R} [f(X_t \quad + \gamma(t,z,x)) \quad f(X_t \quad)) \quad \gamma(t,z,x)\frac{\partial f}{\partial x}]\nu(dz)dt \end{aligned}$$

This is proven in Appendix B. In this work, we only use finite activity jumps. This means that the Itô formula can be written as

$$\begin{aligned} df(X_t) = &(\frac{\partial f}{\partial t} + \mu\frac{\partial f}{\partial x} + \frac{1}{2}\sigma^2\frac{\partial^2 f}{\partial x^2} \quad \int_{\mathsf{R}} \gamma(t,z,x)\frac{\partial f}{\partial x}\nu(dz))dt + \sigma\frac{\partial f}{\partial x}dW_t \\ &+ \int_{\mathsf{R}} f(X_t \quad + \gamma(t,z,x)) \quad f(X_t \quad))N(dt,dz) \end{aligned}$$

We use the Itô formula to obtain the infinitesimal operator of this Lévy process. We will only show this generator for the finite activity case, e.g. $R = \mathbf{1}$.

Theorem 10. *The in nitesimal generator given by*

$$\mathsf{L}f = \lim_{\Delta t \# 0} \frac{\mathsf{E}_x(f(X_{t+\Delta t})) \quad f(X_t)}{\Delta t}$$

of a nite activity Levy process is

$$\mathsf{L}f = (\frac{\partial f}{\partial t} + \mu\frac{\partial f}{\partial x} + \sigma^2\frac{\partial^2 f}{\partial x^2}) + \int_{\mathsf{R}} [f(X_t \quad + \gamma(t,z,x)) \quad f(X_t \quad)) \quad \gamma(t,z,x)\frac{\partial f}{\partial x}]\nu(dz)$$

The proof is similar as in Lemma (2). Here we use that $\tilde{N}$ is a martingale. We have now derived the HJB-equation for finite activity Lévy processes. We have to plug the infinitesimal generator of Theorem (10) into Theorem (5).

4.1.2 Bernoulli approximation

Let $P(t)$ be a Poisson process. We calculate the chance that k jumps happen in $[t, t + \Delta t]$.

$$\mathsf{P}(P(t+\Delta t) \quad P(t) = k) = \frac{(\lambda\Delta t)^k}{k!}e^{\ \lambda\Delta t}$$

In our schemes, we will discretize the time. This means that Δt becomes small. We note from the equation above that the chance of 2 or more jumps can get very small.

$$\mathsf{P}(P(t+\Delta t) \quad P(t)=1) = \frac{\lambda\Delta t}{1} e^{\ \lambda\Delta t} \quad = \lambda\Delta t + \mathsf{O}(dt)$$
$$\mathsf{P}(P(t+\Delta t) \quad P(t)=2) = \frac{(\lambda\Delta t)^2}{2} e^{\ \lambda\Delta t} = \mathsf{O}(dt^2)$$

Therefore, we can ignore multiple jumps in a time interval. The chance for no jumps is $e^{\ \lambda\Delta t} = 1 \quad \lambda\Delta t + \mathsf{O}(dt)$. This suggest that we can approximate a Poisson process by a Bernoulli distribution with parameter $p = \lambda\Delta t$.

The mean of this Bernoulli distribution is $\mathsf{E}(P_{ber}(\Delta t)) = \lambda\Delta t$. This is the same as the Poisson process. We can now also get the compensated Bernoulli process

$$\hat{P}_{ber}(t) = P_{ber}(t) \quad \lambda t \tag{21}$$

In this work, we will use the Poisson process unless we state that we use the Bernoulli approximation.

4.2 Schemes

Theoretically, extending the DBDP and dBSDE schemes to include jumps should be simple. It can quickly be shown that the FBSDEj (FBSDE with jumps) simply adds a new non-local term to the non-jump FBSDEs. The non-locality combined with the randomness of the jumps makes it a lot more difficult in practise. It may be easy to directly estimate the finite activity Poisson process directly, but the compensation part is often non-trivial. In this section, we will first show the simple FBSDEj. Afterwards, we use an offline Monte Carlo approach to extend the DBDP. In the final section, we show why extending the dBSDE to jumps is more difficult.

4.2.1 Forward Backward SDE with jumps (FBSDEj)

We try to solve the following PDE:

$$\frac{\partial u}{\partial t} = \mathsf{L}\, u + f(t, x, y, \mathsf{I}\,) \tag{22}$$
$$u(T,\ ) = g(\ ) \tag{23}$$

Here L is the infinitesimal generator given by a Lévy process, as derived in Theorem (10). I is an extra term that is related to jumps, as can be seen in the Partial Integro Differential Equation (PIDE):

$$\begin{aligned}\frac{\partial u}{\partial t} = (\mu\frac{\partial u}{\partial x} + \frac{1}{2}\sigma^2\frac{\partial^2 u}{\partial x^2})dt + \sigma\frac{\partial u}{\partial x}dW_t \\ + \int_{\mathsf{R}^d}[u(X_t \quad + \gamma(t,z,x)) \quad u(X_t \ )) \quad \gamma(t,z,x)\frac{\partial u}{\partial x}]\nu(dz)dt \\ + f(t,x,u,\frac{\partial u}{\partial x}, \int_{\mathsf{R}^d}[u(X_t \quad + \gamma(t,z,x)) \quad u(X_t \ ))]\nu(dz)dt)\end{aligned}$$

We will use the same reasoning as in the Itô-Diffusion case. We take the Itô lemma of the underlying Lévy process and we substitute the time derivative using

the PIDE. This gives us an Forward Backwards Stochastic Differential Equation with jumps (FBSDEj). We use the following parameters

$$Y_t = u(t, X_t), \qquad Z_t = \mathsf{r}_x u(t, X_t), \qquad \Gamma_t = u(X_t \quad + \gamma(t, z, x)) \quad u(X_t \quad) \tag{24}$$

$$Y_t = g(X_T) + \int_t^T f(s, X_s, Y_s, Z_s, \int \Gamma_s \nu(dz))ds \quad \int_t^T \sigma(s, X_s) Z_s dW_s \quad \int_t^T \int_{\mathsf{R}^d} \Gamma_s \bar{N}(ds, dz) \tag{25}$$

We note that there are two differences compared to the Itô FBSDE, Equation (7). First, there is an extra integral term in $f(\ )$. This integral will not appear be used in this work. Notice that this is a deterministic integral, which means that we can either calculate it analytically or by using Monte Carlo. The second difference is that we have an extra term in the FBSDEj. The rest of the FBSDEj is the same as the FBSDE, which may suggest that we only need to find an extensions to DBDP and dBSDE.

4.2.2 Extensions to DBDP

As seen in the FBSDEj, we can simply use the Itô-diffusion DBDP method with the addition that we calculate the non-local jumps in some way. We approximate the compensation term using a Monte Carlo algorithm. We pick L paths:

$$\begin{aligned}
\int_{t_i}^{t_{i+1}} u(s, x + \gamma(s, z, x)) \quad u(s, x) d\hat{N} &= \int_{t_i}^{t_{i+1}} u(s, x + \gamma(s, z, x)) \quad u(s, x) dN \\
&\quad \lambda \int_{t_i}^{t_{i+1}} u(s, x + \gamma(s, z, x)) \quad u(s, x) ds \\
&= \sum_{\text{jump in } [t_i, t_{i+1}]} u(s, x + \gamma(s, z, x)) \quad u(s, x) \\
&\quad \lambda \int_{t_i}^{t_{i+1}} u(s, x + \gamma(s, z, x)) \quad u(s, x) ds \\
&= \sum_{\text{jump in } [t_i, t_{i+1}]} u(s, x + \gamma(s, z, x)) \quad u(s, x) \\
&\quad \lambda \Delta t \frac{1}{L} \sum_{L} u(t_{i+1}, x + \gamma(t_{i+1}, z, x)) \quad u(t_{i+1}, x)
\end{aligned}$$

Note that we can choose whether we do the Monte Carlo simulation on time t_i or t_{i+1}. Since we have already trained the networks for time t_{i+1}, we already have the correct solutions at this time. This allows us to do the Monte Carlo simulation offline. Note that this is different from [17], who first published this method.

We now only need to specify how we calculate $u(s, x + \gamma(s, z, x)) \quad u(s, x)$. For this, we propose 3 methods:

1. Deterministic: We substitute $u(s, \ )$ for Neural networks $\mathsf{U}_{i+1}(\ )$. We do this for both the jump and compensation components.
2. U-Net: The substitution for the jump component becomes U_i, while we keep the offline Monte Carlo U_{i+1} in the compensation part. The main difference is that the jump component now enters the learning loop.

3. G-Net: We follow [17] more closely and set $u(s, \ +\gamma(s, \ )) \quad u(s, \ ) = \mathsf{G}_i(\ )$. Here G are new neural networks. Again, we keep the compensation terms offline by using G_{i+1}. For time $t_{N \ 1}$, we need G_N, which has not been learned. Therefore, we will use method as 1 for the $t_{N \ 1}$ time step.

4.2.3 deep BSDE with Jumps

We may think that we can extend the dBSDE algorithm similarly as for the DBDP-MC method. Here, DBDP-MC uses an offline Monte Carlo part to calculate the compensated jumps. In the dBSDE algorithm, we learn all networks at the same time. This means that we cannot pick an already learned network to approximate the compensation component. Therefore, we can only extend the dBSDE method with online Monte Carlo. This is inefficient, as was also noted in [24]. We first show a hierarchical dBSDE method to solve PDEs with an an underlying Lévy process. Afterwards, we show the Gnoatto's method, which is similar to the hierarchical dBSDE.

4.2.3.1 Hierarchical dBSDE In this section we introduce the hierarchical dBSDE (H-dBSDE), which consists of 2 loss functions. We state the general structure of the method and then show how we can calculate or learn its components.

The difference between the FBSDEj and the FBSDE are the jumps. As we already saw in the DBDP-MC method, we only need a way to simulate the compensation part of the jumps. Since theses compensations are deterministic functions of t and X, we can approximate them with a new set of Neural Networks V_i. These networks need to be learned, so we have a second minimization problem. We call the loss that appears in the dBSDE without jumps the Terminal Loss and the other the Jump Loss. This leads to the following optimization problem:

$$\begin{aligned} & \text{minimize}_{\mathsf{U}_i} \text{ Terminal Loss} \\ \text{such that} \quad & \\ & \text{minimize}_{\mathsf{V}_i} \text{ Jump Loss} \\ & \text{FBSDEj} \end{aligned} \tag{26}$$

The hierarchical structure is important. We will see that the Jump Loss is generally non-zero, while we need the Terminal Loss to be zero. Therefore, we can get a different solution if we neglect this hierarchy. We also note that the two losses are not minimized using the same variable. We can also have both minimization problems depend on both the U and the V networks, but following equation (26) is more convenient. The Terminal Loss in equation (26) is the same as in the no-jump dBSDE method. We derive an expression for the Jump Loss, which immediately shows how we can simulate the FBSDEj.

We derive the Jump Loss for a single time step. We use the same idea as in Gnoatto et al. [24]. They start with noting that the integral with respect to the compensated Poisson measure is a martingale.

$$\mathsf{E}(\int_{t_i}^{t_{i+1}} u(s, x + \gamma(s, z, x)) \quad u(s, x) d\hat{N}\mathsf{j}\mathsf{F}_{t_i}) = 0 \tag{27}$$

We know that the conditional expectation is a minimizer for the L^2 loss. When we combine this property with the martingale property, we get

$$\arg\min_{\mathsf{C}} \mathsf{E}((\int_{t_i}^{t_{i+1}} u(s, x+\gamma(s,z,x)) \quad u(s,x)d\hat{N} \quad \mathsf{C})^2) = 0 \tag{28}$$

Therefore, we can obtain the correct solution when we drop the C in the expectation. We can now use the new set of neural networks V to simulate the compensated jump components:

$$\begin{aligned}
\text{Jump Loss} &= \mathsf{E}((\int_{t_i}^{t_{i+1}} u(s, x+\gamma(s,z,x)) \quad u(s,x)d\hat{N})^2) \\
&= \mathsf{E}((\int_{t_i}^{t_{i+1}} u(s, x+\gamma(s,z,x)) \quad u(s,x)dN \\
&\quad \lambda \int_{t_i}^{t_{i+1}} u(s, x+\gamma(s,z,x)) \quad u(s,x)ds)^2) \\
&= \mathsf{E}((\sum_{\text{jump in } [t_i,t_{i+1}]} [\mathsf{U}_i(x+\gamma(s,z,x)) \quad \mathsf{U}_i(x)] \quad \lambda\Delta t \mathsf{V}_i(x))^2)
\end{aligned} \tag{29}$$

This is added to the loss function, while

$$\sum_{\text{jump in } [t_i,t_{i+1}]} [\mathsf{U}_i(x+\gamma(s,z,x)) \quad \mathsf{U}_i(x)] \quad \lambda\Delta t \mathsf{V}_i(x) \tag{30}$$

is used to simulate the jumps in the FBSDEj.

Using equation (29) as the Jump Loss and add equation (30) in the FBSDEj gives us the full expression of our hierarchical structure, equation (26).

$$\text{minimize}_{\mathsf{U}_i} \ \mathsf{E}((g(X_N) \quad Y_N)^2)$$

such that

$$\begin{aligned}
&\text{minimize}_{\mathsf{V}_i} \ \sum_i \mathsf{E}((\sum_{[t_i,t_{i+1}]} [\mathsf{U}_i(x+\gamma(s,z,x)) \quad \mathsf{U}_i(x)] \quad \lambda\Delta t \mathsf{V}_i(x))^2) \\
&X_{t_{i+1}} = X_{t_i} + \mu(t, X_{t_i})\Delta t_i + \sigma(t, X_{t_i})\Delta W_{t_i} \\
&\qquad + \sum_{\text{jump in } [t_i,t_{i+1}]} \gamma(t_i, z, x) \quad \Delta t \int_{\mathsf{R}^d} \gamma(t_i, z, x)\nu(dz) \\
&Y_{t_{i+1}} = Y_{t_i} \quad f(t_i, X_{t_i}, Y_{t_i}, \mathsf{U}_i, \mathsf{V}_i)\Delta t + \sigma(t_i, X_{t_i})Z_{t_i}\Delta W_{t_i} \\
&\qquad + \sum_{\text{jump in } [t_i,t_{i+1}]} [\mathsf{U}_i(X_{t_i}+\gamma(t_i,z,X_{t_i})) \quad \mathsf{U}_i(X_{t_i})] \quad \lambda\Delta t \mathsf{V}_i(X_{t_i})
\end{aligned} \tag{31}$$

Given weights and biases of the neural networks, we can easily calculate both loss functions and the FBSDEj. The X_t-paths have an integral, but it can be simulated directly using Monte Carlo. Remember that we introduced the Jump Loss to simulate the compensation component of the jump only. Therefore, U_i and V_i can be learned independently. We can alternatively solve both optimization problems to minimize both losses simultaneously. This is similar as the PIA method we can use to solve HJB equation, as mentioned in [4].

4.2.3.2 Gnoatto's dBSDE method We compare equations (26) and (31) with Gnoatto et al. [24] method. The latter uses the sum of both loss functions as its function.

$$\begin{aligned}
&\text{minimize} \quad \text{Terminal Loss} \; + \; \text{Jump Loss} \\
&\qquad \mathsf{E}((g(X_N) \quad Y_N)^2) \; + \; \textstyle\sum_i \mathsf{E}((\sum_{[t_i,t_{i+1}]}[U_i(x+\gamma(s,z,x)) \quad U_i(x)] \quad \lambda\Delta t V_i(x))^2) \\
&\text{such that} \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (32) \\
&\qquad X_{t_{i+1}} = X_{t_i} + \mu(t, X_{t_i})\Delta t_i + \sigma(t, X_{t_i})\Delta W_{t_i} \\
&\qquad Y_{t_{i+1}} = Y_{t_i} \quad f(t_i, x, y, z)\Delta t_i + \sigma(t_i, X_{t_i}) Z_{t_i} \Delta W_{t_i} \\
&\qquad\qquad + \sum_{\text{jump in } [t_i,t_{i+1}]} [U_i(x+\gamma(s,z,x)) \quad U_i(x)] \quad \lambda\Delta t V_i(x)
\end{aligned}$$

This structure may produce a wrong solution, since the Terminal Loss should be zero, while the Jump Loss is generally non-zero. We show the latter by looking at a single time step.

$$\mathsf{E}((\sum_{[t_i,t_{i+1}]} [U_i(x+\gamma(s,z,x)) \quad U_i(x)] \quad \lambda\Delta t V_i(x))^2) = \mathsf{E}((U \quad V)^2)$$

Here U shows the effects of the U_i network. This mean that U is a function that depends on the number of jumps and the sizes of these jumps. Both of these are random variables. Therefore, U is a random variable. V on the other hand is a deterministic value. Therefore, the mean squared error can only be 0 when V_i can describe U perfectly. This can only happen when U is not a random variable, which only happens when there cannot be any jumps.

So, when using equation (32), it can converge to a non-zero Jump Loss. Since this structure is minimizing the sum of the loss functions, the optimal Terminal Loss can be non-zero as well. A simplified example for this behaviour is show in Appendix C. This means that it may not fit the Terminal condition well enough, which makes it for example unlikely to price American options correctly with this method. We try to overcome this problem by introducing an extra hyperparameter α 2 $[0, 1]$ to weight the losses. We can substitute the loss function in equation (32) for:

$$\text{minimize } \alpha \quad \text{Terminal Loss} + (1 \quad \alpha) \quad \text{Jump Loss}. \tag{33}$$

The error Equation (32) makes depends on how close α is to 1. If α is almost 1, the dBSDE-Jump method will prioritize the Terminal Loss over the Jump Loss. This is similar to the H-dBSDE method, equation (31). However, when we do this, the Jump Loss is decreased slowly. This means that this algorithm is slow when we want to include jumps.

Despite these problems, the dBSDE-Jump method can still give the desired solution in some special cases. The dBSDE-Jump method is successful when the Terminal Loss is small. This is guaranteed when the Jump Loss is small, which happens when the jump intensity is low or the jump sizes are small. Also, all terms in the Jump Loss depends on Δt. So when we use a fine grid, the Jump Loss gets small too. Gnoatto et al. [24] showed that the dBSDE-Jump method can price European options correctly. In the next section, we show that this holds, although it may simultaneously fit the terminal time incorrectly.

4.2.3.3 Gnoatto's method is able to price European options correctly

In this section, we show that equation (32) prices European options correctly at $t = 0$. We do this by showing that the method is equivalent to a simple Monte Carlo simulation:

$$Y_0 = e^{\ rT}\mathsf{E}(Y_N)$$

We can show this equivalence in two steps. First, we show that $\mathsf{E}(Y_N)$ is equal to $\mathsf{E}(g(X_N))$, where $g(\ )$ the pay-off function. Afterwards, we show that the dBSDE-Jump method emulates the discounting factor.

Mean of the Terminal Fit

First, we show that

$$\mathsf{E}(Y_N) = \mathsf{E}(g(X_N)) \tag{34}$$

We do this by assuming a non-specified regression method $P_i = f(Q_i, \beta) + \epsilon_i$. $f(Q_i, \beta)$ denotes the regression method where the goal is to make ϵ_i as small as possible. Also, we make sure that $f(Q_i, \beta)$ is unbounded before specifying the regression parameters. for example $f(Q_i, \beta) = sin(\beta Q_i)$ is not allowed, because it will always be in $[\ \ 1, 1]$. On the other hand, a constant regression method, $f(Q_i, \beta) = \beta$, is allowed. Do note that the regression is not perfect and may even be bad. For example we can regress points (Q_i, P_i) that lay on a parabola with the constant regression method. We can do this because we are only interested in the mean error: $\mathsf{E}(P_i \quad f(Q_i, \beta))$. At last, note that this does not mean that the mean squared error is 0.

Theorem 11. *Let $f(Q_i, \beta)$ be a t for P_i. Assume $\mathsf{E}(P_i)$ 6= $\mathsf{E}(f(Q_i, \beta))$. Now set $k = \mathsf{E}(P_i \quad f(Q_i, \beta))$ 6= 0. Than we can improve the t in MSE by adding k to $f(Q_i, \beta)$.*

Proof.

$$\begin{aligned}
\mathsf{E}((P_i \quad (f(Q_i, \beta) + k))^2) &= \mathsf{E}((P_i \quad f(Q_i, \beta) \quad k)^2) \\
&= \mathsf{E}((P_i \quad f(Q_i, \beta))^2) \quad 2\mathsf{E}(k(P_i \quad f(Q_i, \beta))) + \mathsf{E}(k^2) \\
&= \mathsf{E}((P_i \quad f(Q_i, \beta))^2) \quad 2k\mathsf{E}((P_i \quad f(Q_i, \beta))) + k^2 \\
&= \mathsf{E}((P_i \quad f(Q_i, \beta))^2) \quad 2k^2 + k^2 \\
&= \mathsf{E}((P_i \quad f(Q_i, \beta))^2) \quad k^2 \\
&< \mathsf{E}((P_i \quad f(Q_i, \beta))^2)
\end{aligned}$$

□

This shows that we can always improve a regression by adding $\mathsf{E}(P_i \quad f(Q_i, \beta))$ to the regression function. Therefore, any regression method will have $\mathsf{E}(P_i \quad f(Q_i, \beta)) = 0$. For the dBSDE method, this means that $\mathsf{E}(Y_N \quad g(X_N)) = 0$, when we are able to add a constant to Y_N. Note that Y_N is a sum of multiple neural networks: $Y_N = Y_0 + G(\mathsf{U}, \mathsf{Z}, X)$. This shows that we can improve the regression by adding k to Y_0. Therefore the dBSDE-Jump algorithm gives $\mathsf{E}(Y_N) = \mathsf{E}(g(X_N))$.

Discounting factor

Next, we show that the dBSDE-Jump method leads to

$$Y_0 = e^{\ rT}\mathsf{E}(Y_N) \tag{35}$$

We start with the FBSDEj, Equations (25) and discretize it.

$$dY_t = \quad f(\ )dt + \sigma Z_t dW_t + \int_{\mathsf{R}^d} \Gamma_t \bar{N}(ds, dz)$$

For option pricing, $f(\ ) = \quad rY_i$.

$$\begin{aligned}\mathsf{E}(Y_{i+1}) &= \mathsf{E}(Y_i + rY_i dt + \sigma Z_t dW_t + \int_{\mathsf{R}^d} \Gamma_t \bar{N}(ds, dz)) \\ &= \mathsf{E}(Y_i + rY_i dt) \\ &= \mathsf{E}((1 + rdt)Y_i)\end{aligned}$$

We use this recursively to obtain $\mathsf{E}(Y_N) = (1 + rdt)^N Y_0$. Here we note that Y_0 is a deterministic variable, so $\mathsf{E}(Y_0) = Y_0$. We remember that $dt = \frac{T}{N}$ and take the limit the amount of time steps to infinity.

$$\begin{aligned}Y_0 &= \frac{1}{(1 + rdt)^N}\mathsf{E}(Y_N) = \frac{1}{(1 + \frac{rT}{N})^N}\mathsf{E}(Y_N) \\ Y_0 &= \lim_{N \,!\, 1} \frac{1}{(1 + \frac{rT}{N})^N}\mathsf{E}(Y_N) = \frac{1}{e^{rT}}\mathsf{E}(Y_N) = e^{\ rT}\mathsf{E}(Y_N)\end{aligned}$$

This shows Equation (35). We can now combine both Equation (34) and Equation (35) to obtain

$$Y_0 = e^{\ rT}\mathsf{E}(Y_N) \tag{36}$$

This means that the dBSDE-Jump algorithm is similar to simple Monte Carlo. The advantage this dBSDE-Jump has over Monte Carlo is that dBSDE-Jump is able to price multiple European options which lay in some initial price range X_0 *2* R^d simultaneously. A simple Monte Carlo simulation requires to divide this range into a grid, which is inefficient for large dimensions. dBSDE-Jump is not able to obtain correct values for Y_i with i ~~6=~~ 0. This makes it impossible to solve problems that uses a DPP-argument to price options (Bermudan option) and coupled FBSDEj.

4.3 Theoretical results for the FBSDEj and DBDP-MC

In this section, we first show existence and uniqueness of the FBSDEj. Afterwards, we show the theorem in Castro et. al [17] that shows convergence of the DBDP-MC method. We introduce the terms used in this theorem and afterwards state this theorem. In this section, we use $dt = h$ and

$$\mathsf{E}_i(\ ) = \mathsf{E}(\ j\mathcal{F}_{t_i})$$

4.3.1 Uniqueness and Existence FBSDEj

In this section we show that the FBSDEj has always a solution and that this solution is unique. We structure the theorems as in [17], so we do not prove any results. First, it is shown that the finite activity Lévy process is unique.

Theorem 12. *There exist a unique nite activity Levy process X_t, such that*

$$\mathsf{E}(\sup_{t<s<T}(X_s \quad X_t)) \quad h(1 + \mathsf{E}(X_t)^2)$$

This result will be used in the proof that shows convergence of the DBDP-MC method. Next, the backwards equation of the FBSDEj is shown to exist.

Theorem 13. *There exists a unique solution of (Y_t, Z_t, Γ_t) (see equation (24)) to the FBSDEj, equation (25).*

[17] does not specify that the solution is unique. however, this follows immediately from theorem 2.1 in [6]. This shows that there exists a unique solution to the FBSDEj given by $(X_t, Y_t, Z_t, \Gamma_t)$.

4.3.2 Convergence of DBDP-MC

We now show that the DBDP-MC converges. Again, we use the structure in [17]. First, we introduce a term for the integral of the compensation part of the compensated Poisson process. We also show how the DBDP-MC uses neural networks to approximate this term.

$$G_t = \int_{\mathsf{R}^d} u(X_t + \gamma(t,z,x)) \quad u(X_t)\lambda(dz) \tag{37}$$

$$\mathsf{G}_i = \int_{\mathsf{R}^d} \mathsf{U}_i(X_t + \gamma(t,z,x)) \quad \mathsf{U}_i(X_t)\lambda(dz) \tag{38}$$

We will now give a measure of the discretization error. We do this by giving the total L^2 error between the desired processes and the average of their discretization.

$$\overline{Z_{t_i}} = \frac{1}{h}\int_{t_i}^{t_{i+1}} Z_t dt \tag{39}$$

$$\overline{G_{t_i}} = \frac{1}{h}\int_{t_i}^{t_{i+1}} G_t dt \tag{40}$$

$$\varepsilon^Z = \mathsf{E}(\sum_{i=0}^{N}\int_{t_i}^{t_{i+1}} jZ_t \quad \overline{Z_{t_i}}j^2 dt) \tag{41}$$

$$\varepsilon^G = \mathsf{E}(\sum_{i=0}^{N}\int_{t_i}^{t_{i+1}} jG_t \quad \overline{G_{t_i}}j^2 dt) \tag{42}$$

The discretization errors ε^Z and ε^G become small fast enough. This is shown in the following theorem.

Theorem 14. *There exist a constant C 0, such that*

$$\varepsilon^Z < Ch$$
$$\varepsilon^G < Ch$$

Time discretization gives an error on Y_t too. [13] proves that its error is $\mathsf{O}(h)$.

$$\sum_{i=0}^{N}\int_{t_i}^{t_{i+1}} (Y_s \quad Y_{t_i})^2 ds) \quad Ch \tag{43}$$

Now we show that neural networks can learn the wanted processes. This requires that these processes must be written as deterministic functions. We do this using a new Lévy processes that are connected to the wanted process. We use the Martingale

Representation Theorem on the martingale $\mathsf{E}_i(\mathsf{U}_{i+1}(X_{t_{i+1}}))$ for $t \in [t_i, t_{i+1}]$. We show both this martingale and the original process.

$$\mathsf{U}_{i+1}(X_{t_{i+1}}) = \mathsf{U}_i(X_{t_i}) \quad f(t, X, \mathsf{U}_i, \mathsf{Z}_i, \int \Gamma_s \nu(dz))\Delta t_i + \int_{t_i}^{t_{i+1}} Z dW_i + \int_{t_i}^{t_{i+1}} \Gamma_s(z) d\hat{N}(ds, dz) \tag{44}$$

$$\mathsf{U}_{i+1}(X_{t_{i+1}}) = \mathsf{E}_i(\mathsf{U}_{i+1}(X_{t_{i+1}})) + \int_{t_i}^{t_{i+1}} \hat{Z}_s dW_s + \int_{t_i}^{t_{i+1}} \hat{\Gamma}_s(z) d\hat{N}(ds, dz) \tag{45}$$

We will write new processes in terms of $\mathsf{E}_i(\mathsf{U}_{i+1}(X_{t_{i+1}}))$. To obtain the deterministic function that will be approximated by U_i, we notice the similarities between the two Lévy processes and propose

$$\hat{V}_i = \mathsf{E}_i(\mathsf{U}_{i+1}(X_{t_{i+1}})) + f(\,)h \tag{46}$$

This process is well-defined, assuming h is small enough. This was proven in [17] using Banach's fixed point theorem. We will need similar processes for Z_i and for the neural network representation of the jumps. We use the martingale obtained by the Martingale Representation Theorem, the left-most two term in Equation (45). We multiply these two terms by $\int_{t_i}^{t_{i+1}} dW_i$ and $\int_{t_i}^{t_{i+1}} \int_{\mathsf{R}^d} d\hat{N}(ds, dz)$. We take the conditional expectation and use Itô isometry. This gives the following processes:

$$\overline{\hat{Z}}_{t_i} h = \mathsf{E}_i(\mathsf{U}_{i+1}(X_{t_{i+1}}) dW_i) = \mathsf{E}_i(\int_{t_i}^{t_{i+1}} \hat{Z}_s ds) \tag{47}$$

$$\overline{\hat{\Gamma}}_{t_i} h = \mathsf{E}_i(\mathsf{U}_{i+1}(X_{t_{i+1}}) d\hat{N}) \;= \mathsf{E}_i(\int_{t_i}^{t_{i+1}} \int_{\mathsf{R}^d} \hat{\Gamma}_s(z) \lambda(dz) ds) \tag{48}$$

$$\overline{\hat{G}}_{t_i} h = \qquad\qquad = \mathsf{E}_i(\int_{t_i}^{t_{i+1}} \int_{\mathsf{R}^d} \hat{\Gamma}_s(z) ds) \tag{49}$$

We will use $\overline{\hat{G}}$ to create a function for the jumps instead of $\overline{\hat{\Gamma}}_{t_i}$. Using conditional Fubini, we see the connection between the two processes:

$$\overline{\hat{\Gamma}}_{t_i} = \int_{\mathsf{R}^d} \overline{\hat{G}}_{t_i} \lambda(dz) \tag{50}$$

We now define the deterministic functions. Notice the similarities between the two Lévy processes.

$$v_i = \hat{V}_i(X_i) \qquad z_i = \overline{\hat{Z}}_{t_i}(X_i) \qquad g_i = \overline{\hat{G}}_{t_i}(X_i) \tag{51}$$

This gives the following error estimates:

$$\mathsf{E}_i^v = \inf \mathsf{E}(v_i(X_i) \quad \mathsf{U}_i(X_i))^2 \tag{52}$$

$$\mathsf{E}_i^z = \inf \mathsf{E}(z_i(X_i) \quad \mathsf{Z}_i(X_i))^2 \tag{53}$$

$$\mathsf{E}_i^g = \inf \mathsf{E}(\int_{\mathsf{R}^d} (g_i(X_i) \quad \mathsf{G}_i(X_i))^2 \lambda(dz)) \tag{54}$$

From the UAT, we can get these error arbitrarily small. We can now state the theorem:

Theorem 15. *There exist a constant* $C \quad 0$ *which is independent of* h*, such that for small* h

$$\max \mathsf{E}(Y_{t_i} \quad U_i(X_i))^2 + \sum_{i=0}^{N} \mathsf{E}(\int_{t_i}^{t_{i+1}} jZ_t \quad \overline{Z_{t_i}}j^2 + jG_t \quad \overline{G_{t_i}}j^2 dt)$$

$$C[\mathsf{E}(g(X_T^h) \quad g(X_T))^2 + h + \varepsilon^Z + \varepsilon^G + \sum_{i=0}^{N}(N\mathsf{E}_i^v + \mathsf{E}_i^z + \mathsf{E}_i^g)] \tag{55}$$

The theorem shows that when we use a finer grid, we can approximate the solution better. The right-hand-side shows the triplet (Y_t, Z_t, Γ_t), where we have the maximum error in Y_t and the average error of the other parameters. Note that all terms on the right-hand-side were constructed to be $\mathsf{O}(h)$. This means that the solution converges to the correct solution with $\mathsf{O}(h)$, assuming the networks are trained correctly.

In our DBDP-MC method, we use an offline Monte Carlo simulation. This means that we need to change the argument slightly. [28] proved that the DBDP without jumps can also be solved using automatic differentiation of Z_t. A similar method can be used to show that our offline Monte Carlo method is valid.

5 Numerical Results

In this section, we show some results for the DBDP-MC and dBDSE-jump methods. But first we start by describing the used structure of these methodologies and the used hyperparameters. Afterwards, we show that the DBDP-MC method is able to price Bermudan options correctly under the Merton-Jump-Diffusion process (MJD). Here we first explain the problem and then show the used parameter sets. This is followed by describing the used reference method before we show the results of the algorithm. We continue by pricing European option with the dBSDE-Jump method under the MJD process. This shows that our theoretical explanation of the dBSDE-Jump method is likely correct, but that the method can still be effective and useful in solving PDEs.

5.1 Hyperparameters

Structure and Hyperparameters

For all experiment, we use 2 hidden layers. Both layers have $30+d$ neurons. We will have 100 networks per DBDP or dBSDE algorithm. This means a time discretization of 100. We will slightly change the amount of networks when pricing a Bermudan or American option. Here we will use the closest integer to 100, such that the amount of early exercise opportunity times is divisible by this number (e.g. choose $N = 104$ when we have 8 early exercise opportunities). In most papers that solve PDEs using the DBDP or dBSDE method use the $Tanh$ activation function. For the dBSDE-Jump algorithm and in the first test example for the DBDP-MC method, we also use this activation function. However, we will use the DBDP-MC methodology on higher dimensional problems, where the $Tanh$ activation function was not able to make a correct fit for the pay-off function. This should be possible due to the UAT, but we decided to use a different activation function: ELU. So we use the $Tanh$ activation function, except when we have a high-dimensional problem, then we use ELU.

$$\text{Tanh: } \sigma(x) = Tanh(x)$$

$$\text{ELU: } \sigma(x) = \begin{cases} x & x > 0 \\ e^x & 1 \quad x \quad 0 \end{cases}$$

Training Neural Networks

We use the ADAM gradient descent method to back-propagate. When using DBDP, we start with an initial learning rate of 0.01 for the terminal time and an initial learning rate of 0.0001 for non-terminal times. If we use the dBSDE method, the learning rate is 0.01 for all networks. Note that the initial learning rate is not an important hyperparameter, since the ADAM algorithm will adapt this rate. We use a batchsize of 1000 and set a maximum amount of iterations: 40000 for the DBDP method and 4000 for the dBSDE. In the DBDP method, we use early exercise opportunity. We check the loss every 100 gradient steps. We then check whether the difference compared to the loss 100 steps ago is less then 0.001. We also check whether the loss is small enough, this is done by setting a threshold which is a tenth of the terminal loss. For the terminal time, the threshold is 0.001.

5.2 Bermudan Options

5.2.1 Setting

In this section, we calculate the price of Bermudan options under the Merton Jump Diffusion process (MJD). This is a Lévy process with $\gamma(z,x) = e^z \quad 1$ and z is normal distributing. Thus the stocks paths are simulated using

$$dX_t^i = rX_t^i dt + \sigma X_t^i dW_t^{\mathsf{Q}} + \int_{\mathsf{R}} [e^z \quad 1]\bar{N}(dt, dz).$$

The problem is d-dimensional. All stocks have the same initial price X_0. The stocks can be correlated with coefficient $\rho_{i,j}$, where i and j denote different stocks. This means that all stocks have the same correlation. If a jump occurs, all stocks jump simultaneously at this time. Therefore we can write the jump intensity with a single parameter: λ. The jump sizes follow a normal distribution $\mathsf{N}(\mu^J, \sigma^J)$ and are also correlated with $\rho_{i,j}^J$. In this section we price European, Bermudan and American options. For a European option, we earn the pay-off function at terminal time T. This pay-off usually depends on a strike price K. In the case of American options, we are allowed to exercise the option at any time $t < T$. Bermudan options can be exercised in some finite time points. We take these early exercise opportunity opportunities equidistant and the amount of opportunities is denoted by M. For example, when $M = 3$, we can early exercise at $\frac{T}{4}, \frac{2T}{4}, \frac{3T}{4}$ and at the terminal time. We will use 3 different options: Arithmetic Put options, Geometric Put options and Put-on-Min options. Next we show the pay-off function for the European equivalent of these 3 options types.

$$\begin{aligned}
\text{Arithmetic Put Option: } & \max(0, K \quad \frac{1}{d}\sum_{i=1}^{d} X_T^i) \\
\text{Geometric Put Option: } & \max(0, K \quad (\prod_{i=1}^{d} X_T^i)^{\frac{1}{d}}) \\
\text{Put-on-Min Option: } & \max(0, K \quad \max_i(X_T^i))
\end{aligned}$$

Unless specified otherwise, we use the Arithmetic mean Put option. We show the used parameter sets in Table 1.

We need to make sure that the DBDP-MC takes the early exercise opportunities into account. We note that we can easily see whether we should exercise by comparing the payoff at this exercise time with its value at the same time. The option contract will be exercised when the current value is higher than the payoff. Set V_i as the value of the option at time t_i

$$V_i = \max\mathsf{f}Y_i, g(X_i)\mathsf{g} \tag{56}$$

We emulate this using neural networks. When we have successfully learned U_i with

the DBDP-MC algorithm and there is an early payoff opportunity at i, we set

$$\text{update } \mathsf{U}_i = \max\mathsf{f}\mathsf{U}_i, g(X_i).\mathsf{g}$$

We continue the DBDP as normal, but in step $i \quad 1$, we will use U_i to learn $\mathsf{U}_{i \ 1}$ and $\mathsf{Z}_{i \ 1}$.

Table 1: Bermudan Sets for arithmetic mean basket put options

Set 1: *A test set from [19]*
$d = 5$, $T = 1$, $M = 8$, $X_0 = 100$, $K = 100$, $r = 0.05$, $\sigma = 0.15$,
$\rho_{ij} = 0.3$, $\lambda = 0.5$, μ^J = [-0.3, -0.2, -0.1, 0.1, 0.2]',
$\sigma^J = 0.1$, ρ_{ij}^J = -0.2

Set 2: *No jump setup*
$d = 1$, $T = 1$, $X_0 = 100$, $K = 100$, $r = 0.1$, $\sigma = 0.2$, $\rho_{ij} = 0.5$

Set 3: *Main set*
$d = 1$, 20 or 50, $T = 1$, $M = 3$, $X_0 = 100$, $K = 100$, $r = 0.1$,
$\sigma = 0.2$, $\rho_{ij} = 0.5$, $\lambda = 0.5$, μ^J = -0.1, $\sigma^J = 0.3$, $\rho_{ij}^J = 0.0$

Set 4: *Intense jumps*
$d = 5$, $T = 1$, $M = 0$, $X_0 = 100$, $K = 100$, $r = 0.1$, $\sigma = 0.2$,
$\rho_{ij} = 0.5$, $\lambda = 5.0$, μ^J = [-.1, -.1, -.1, -.1, -.1]', $\sigma^J = 0.3$, $\rho_{ij}^J = 0.0$

Set 5: *Intense jumps 2*
$d = 5$, $T = 1$, $M = 0$, $X_0 = 100$, $K = 100$, $r = 0.1$, $\sigma = 0.2$,
$\rho_{ij} = 0.5$, $\lambda = 10.0$, μ^J = [-.1, -.1, -.1, -.1, -.1]', $\sigma^J = 0.1$, $\rho_{ij}^J = 0.0$

Set 6:
$d = 20$, $T = 2$, $M = 0$, $X_0 = 100$, $K = 90$, $r = 0.02$, $\sigma = 0.15$,
$\rho_{ij} = 0.3$, $\lambda = 0.3$, $\sigma^J = 0.1$, $\rho_{ij}^J = 0.2$,
μ^J = [-.3, -.2, -.1, .1, .2, -.3, -.2, -.1, .1, .2, -.3, -.2, -.1, .1, .2,
-.3, -.2, -.1, .1, .2]'

5.2.2 A reference method: SGBM

To show whether the DBDP-MC method can solve Bermudan options correctly under the MJD model, we compare the results against a different method, the Stochastic Grid Bundling Method (SGBM) [19]. In this method, Monte Carlo paths are generated and the payoff at terminal time is calculated. Now the paths at the previous payoff time are bundled. The asset values at this time are bundled into B non-overlapping partitions. K basis functions $\phi_k(x)$ are chosen with corresponding regression parameters α_k. All the option values are regressed in the bundle on the basis functions.

$$V_i(X_i) \quad \sum_{k=1}^{K} \alpha_k \phi_k(X_i)$$

This is done for all bundles. Afterwards, continuation value is calculated. This can only be done when appropriate basis functions are used. Equation (56) is used to obtain the real option values at this early exercise opportunity. This process is repeated until the price at initial time is obtained.

There are also some other methods we use to obtain a reference value. For European options, we use a Monte Carlo simulation. When we have a 1D problem, we use the binomial tree method by using [37].

5.2.3 Test Case

In this section, we test the DBDP-MC algorithms. We will use all 3 versions of this algorithm on a 5-dimensional test case. We can than compare it to a reference value

and look at the differences between the results. We use Set 1, which was already solved in [19]. This set is reasonably complicated because it has a different jump mean μ_J for all stocks and all stocks are correlated in both their jumps and its Brownian paths.

Table 2: Set 1 test for 3 methods. We use 10 different runs. s.e. is the standard error and larg diff shows the largest percentage error of the 10 runs. Reference value from [19].

	ref value	mean value	s.e.	larg diff	time (s)
Deterministic	2.576	2.5757	0.0108	2.08%	1267
U-Net	2.576	2.5827	0.0082	1.90%	1300
G-Net	2.576	2.5644	0.0147	2.21%	2097

This shows that all 3 versions of DBDP-MC give the correct solution. The most notable difference is that G-Net is almost twice as slow as the other algorithms. In all versions, the obtained solution is within a standard error of the real solution, which suggest that they work correctly. The differences between the algorithms seems insignificant. In the rest of this work, we will only use the deterministic version.

5.2.4 High dimensional Bermudan Options

We continue to show that we can obtain correct prices for European, American and Bermudan options in low and high dimensions. For these high dimensions, it is hard to fit the terminal condition using the $Tanh$-activation function. The $ReLU$-activation function seems like a natural fit for this arithmetic mean options, but it is difficult to learn the non-terminal networks. Therefore, we use the ELU-activation function.

We use the deterministic DBDP-MC algorithm on sets 2 and 3. We use the European equivalent (E) or the American equivalent (A) of these Bermudan (B)options. Some options do not have a reference value.

Table 3: Results for Bermudan options. Given as (Set, type, dimension). We use 5 different runs. s.e. is the standard error and larg diff shows the largest percentage error of the 5 runs.

	ref value	mean value	s.e.	larg diff	time (s)
Set 2, E, 1	3.7535	3.8061	0.0059	1.92%	1095
Set 2, A, 1	4.8160	4.8744	0.0229	2.45%	1132
Set 3, E, 1	6.52	6.5150	0.0303	1.72%	1213
Set 3, A, 1	-	7.5761	0.0063	-	1268
Set 3, E, 20	2.42	2.4393	0.0384	4.71%	3073
Set 3, B, 20	-	3.2869	0.0418	-	3044
Set 3, A, 20	-	3.5479	0.0271	-	3118
Set 3, E, 50	2.2	2.1899	0.0365	4.91%	1728
Set 3, B, 50	-	3.0624	0.0482	-	1740
Set 3, A, 50	-	3.2956	0.0251	-	1759

This shows that DBDP-MC can correctly price European, American and Bermudan options.

We continue by showing plots from Set 3, European with jumps. Here we plot the option price when the current price is between 50 and 200 (100 points, spaced equidistantly). This is to show that DBDP-MC gives the correct answer at all Y_t.

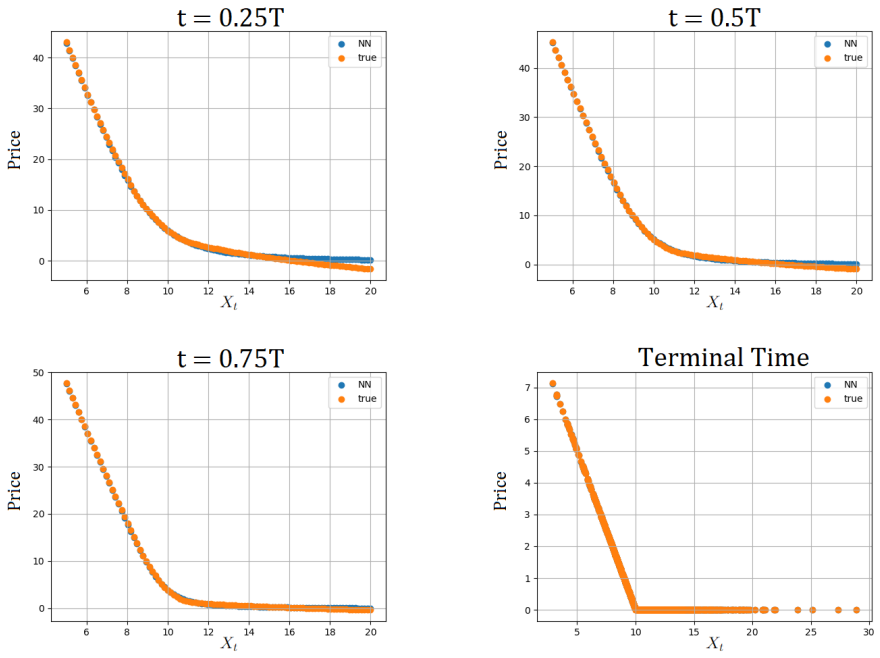


Figure 6: We price a 1-dimensional option using the DBDP-MC method and compare it to simple Monte Carlo simulations. The DBDP-MC prices options correctly for all time steps.

We note that for $t = 0.25T$, the correct option price is not obtained for $X_t \quad 16$. This happens because there was insufficient training data. This can be solved by making sure more X_t-paths reach this value by either using more paths or by using a different initial time. Because X_t is rarely larger than 16 at $t = 0.25T$, this does not lead to a large error for the price at $t = 0$.

5.2.5 Other options

In this section, we show the results of some non-arithmetical mean basket options. We use the geometric mean basket option and the Put-on-Min option. These are more difficult than the arithmetic mean option. We use Set 1, so we can compare the result for the Bermudan geometric option with [19]. We also show the result for the equivalent European option, where we use a Monte Carlo simulation to obtain a reference value. At last, we will try the 20-dimensional geometric basket option from Set 6, which is a more complicated problem.

Table 4: Results for geometric (g) and Put-on-Min (pm) options. Given as (Set, type, option). K represents the strike price. We use 5 different runs. s.e. is the standard error and larg diff shows the largest percentage error of the 5 runs.

	K	ref value	mean value	s.e.	larg diff	time (s)
Set 1, E, g	110	8.19	8.1714	0.0695	3.40%	1172
Set 1, B, g	110	9.8020	9.7978	0.0100	0.34%	1438
Set 1, E, pm	80	3.855	3.8349	0.0450	3.44%	1474
Set 1, B, pm	80	-	4.0601	0.0360	-	1509
Set 6, E, g	90	1.38	1.3301	0.0158	6.11%	3316

First, we note that we do get good results for the Set 1 problems. Therefore, we can conclude that the DBDP-MC can solve 5-dimensional geometric and Put-on-Min options. In the geometric case, the Bermudan option has a significantly lower standard error than the European equivalent. Here, he Bermudan option was often exercised at its earliest opportunity. Therefore, the paths will emulate a European option with a smaller terminal time. This is easy to solve accurately.

The results for Set 6 are not as good as the results from Set 1. The solution is still close to the desired price, but is clearly smaller than it. This happens because the terminal time did not fit correctly. We explain this by plotting the terminal time. Because it is difficult to plot a 20-dimensional fit, we show the terminal fit on a geometric put option for Set 1. To show the problem, we use a smaller batchsize of 100. Moreover, we stop learning before it has fully converged.

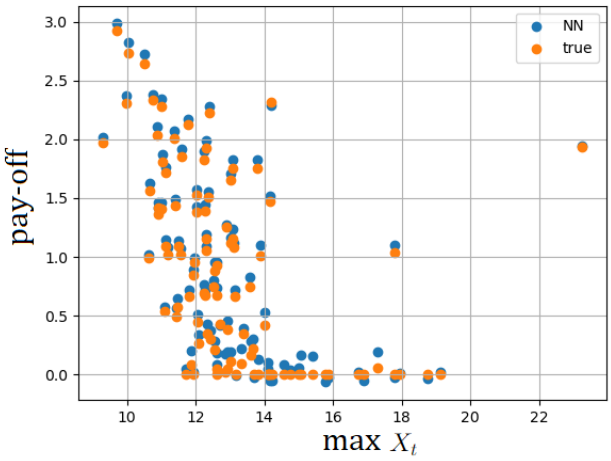


Figure 7: The plot of the fit at terminal time for a geometric put option for Set 1. We use a smaller batchsize and a smaller amount of iterations to demonstrate that the output of the neural network is always higher than the desired result.

Figure 7 shows that the neural network overestimates the terminal time. Since we try to price a put option, this means that we underestimate the price at $t = 0$. The

error that was made at the terminal time propagates backwards through time in all networks. If we want to price this option more accurately, we would need to make sure we have a better terminal fit. This can be achieved with better hyperparameter tuning, since this is a classical regression problem.

The Put-on-Min option is significantly more difficult to price in higher dimensions. As the problem in Set 6 shows, we need to make a good terminal fit. For the Put-on-Min option, only a single stock is used to determine its payoff. The fit requires enough data to describe all possibilities. The Put-on-Min option is more sensitive to this problem than the arithmetic option. The latter will always use all the stocks in its calculation.

5.2.6 Poisson process and Bernoulli approximation

In this section, we price a 5-dimensional arithmetic European basket options as given in Sets 4 and 5. These sets have a higher jump intensity. The goal is to show that the DBDP-MC method is able to price Bermudan options with intense jumps. The approximation error becomes larger when the jumps are larger or there are more jumps. Therefore, we price the Bermudan options in two ways. One where we simulate the Poisson process directly, as in the previous sections and another one using a Bernoulli approximation. We calculate the reference values using the corresponding process.

Table 5: Pricing European options using the Poisson (P) process and it Bernoulli (B) approximation. Errors calculated using 5 paths

	ref value	mean value	s.e.	largest diff	time (s)
Set 4, P	10.12	10.1032	0.0514	1.68%	2688
Set 4, B	10.04	9.9659	0.0758	2.75%	2422
Set 5, P	9.6	9.5793	0.0335	1.90%	2470
Set 5, B	9.1	9.1687	0.0884	4.04%	2294

first, we notice that using the Bernoulli approximation changes the price of the Bermudan option significantly even when using a Monte Carlo simulation. In Set 4, this difference is small. Here, the DBDP-MC algorithm does show that both methods can approach the reference value reasonably well. However, the largest value in the Bernoulli approximation was larger than 10.12 and the smallest in the Poisson algorithm was smaller than 10.04. In Set 5, we got values such that the smallest value of the Poisson process was larger than the largest value in the Bernoulli approximation. Therefore, the results show that while using the Bernoulli approximation creates a bias, as seen in the Monte Carlo reference price, the DBDP-MC algorithm is able to obtain this biased reference price correctly. The error the Bernoulli approximation makes does not come from the DBDP-MC algorithm directly, but from the incorrect sample paths. Both the Bernoulli approximation method as the Poisson process give the desired results. The accuracy is similar as seen in Table 3. Another difference is that the Bernoulli process is slightly faster. This happens because the stocks are easier to simulate using this approximation. so when we have a small jump intensity or we can use a very fine time discretization, the Bernoulli approximation is a more efficient method.

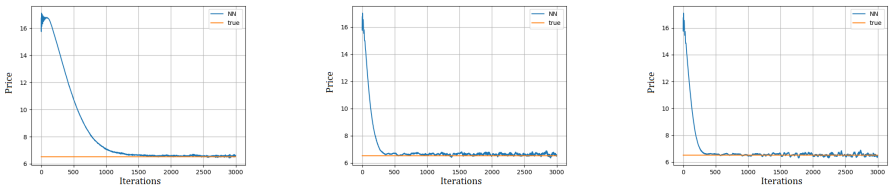
5.2.7 Conclusion

In this section, we demonstrated that the DBDP-MC algorithm is able to price European, Bermudan and American options correctly.

- The algorithm was successful for problems up till dimension 50. We did not try higher dimensional options. Using a strong enough computer and implementing the algorithm in an efficient way, higher dimension, like 100, should be solvable with this method.
- When the jumps are not too large and the jump intensity is low, it is useful to use the Bernoulli approximation to simulate the Poisson process. However, the Bernoulli approximation may lead to a bias when the jump intensity is high or when using a coarse time grid. If the Bernoulli approximation leads to a biased solution, it is better to use the full Poisson process. Both methods were able to solve a 5-dimensional Bermudan options with a high jump intensity.
- The DBDP-MC methodology is sensitive to the accuracy of its terminal fit. Options where this terminal fit are more difficult, like Put-on-Min, can therefore be difficult to price in higher dimensions.

5.3 dBSDE-Jump

In this section, we show some results for the dBSDE-Jump algorithm. First we show that it is able to price European options correctly. This was already shown in [24] using a Bernoulli approximation. We show that it still works in our code. We chose Set 3 with 1 dimension and plot Y_0 for every iteration.



(a) N=100 and equal weights

(b) N = 10 and equal weights

(c) N = 100 and Jump Loss = 10 Terminal Loss

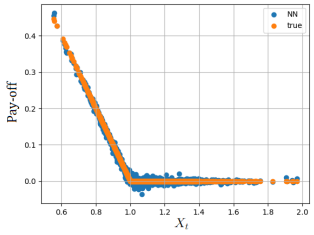
Figure 8: Convergence in iterations of pricing a European option using the dBSDE-Jump algorithm. Large time steps and more wight on the Jump Loss increase the convergence speed.

We see that the dBSDE-Jump algorithm is able to price European options correctly. The convergence is quicker when we put more weight on the Jump Loss or when we take larger time steps.

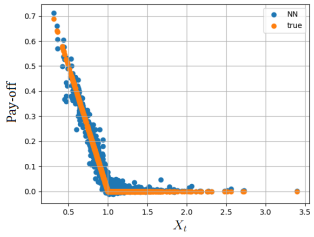
We now show the terminal time fit. We show a non-jump version as a benchmark. This non-jump version is obtained by setting λ very small, such that no jumps occur[1].

[1]The idea is to use the dBSDE-Jump algorithm without jumps. However, setting $\lambda = 0$ gives a division by 0 error. Therefore, we choose a small λ and afterwards inspect whether a jump occurs.

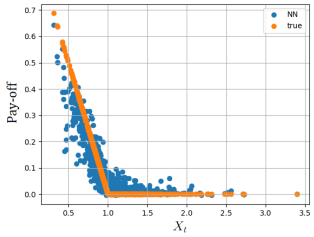
Afterwards, we let $\lambda = 0.5$ and use the dBSDE-Jump method with different weights (see Equation (33)).



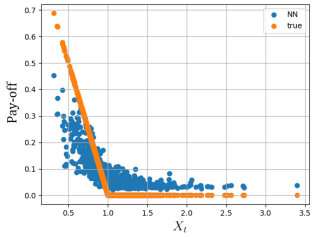
(a) No jumps and equal weights. This is used as the benchmark.



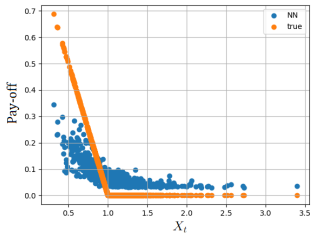
(b) Weight the Jump Loss as much as the Terminal Loss.



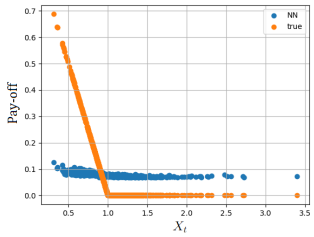
(c) $\alpha = 0.1$



(d) $\alpha = 0.02$



(e) $\alpha = 0.01$

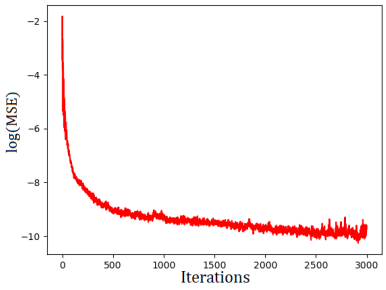


(f) $\alpha = 0.001$

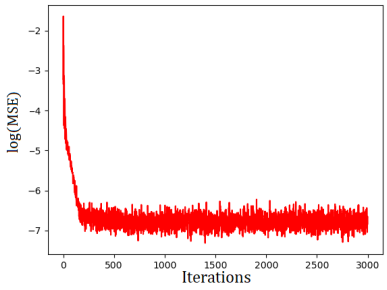
Figure 9: figure (a) shows the terminal plot of the dBSDE-Jump algorithm without jumps. We add jumps in figure (b,c,d,e,f).

We note that when we put a lot of weight on the Jump Loss, the solution becomes close to a constant. This can easily be understood by trying to minimize only the Jump Loss. We can always minimize this loss by setting $U(X) = Y = c$ and $V(X) = 0$. When using equal weights, the terminal plot seems to fit reasonably well. This is not unexpected, since the jump intensity is not that high, while we have a large enough N. If we would do the dBSDE-Jump algorithm in the correct hierarchical setting, the Jump Loss would still be small. Therefore, the fit will always be somewhat close. We compare the loss per iteration for the benchmark no-jump case and the equal weight

case for the dBSDE-Jump algorithm.



(a) No jumps



(b) $\lambda = 0.5$, equal weights

Figure 10: The Terminal Loss of the dBSDE-Jump algorithm. Without jumps, the error seems to get smaller than with jumps. Also, the version with jumps seems to have obtained its minimum value.

The Terminal Loss with jumps is 0.0010, whereas the one without jumps is $5\ 10^{-5}$. The loss of the one with jumps is larger than the one without jumps. This holds even if we only take the Terminal Loss into account, which was 0.0002. The dBSDE-Jump algorithm is not able to minimize this Terminal Loss, because it also has to take the Jump Loss into account. The two loss functions try to balance each other, which leads to a slightly wrong terminal fit. If we can guarantee that the Jump Loss is sufficiently small by using small time steps or not having large or many jumps, the dBSDE algorithm can give solutions that are good enough.

5.3.1 Conclusion

In this section, we quickly state the conclusions that were obtained from numerical experiment with the dBSDE-Jump algorithm.

- The dBSDE-Jump algorithm is able to price European options correctly, as was already shown in [24]. Convergence is faster when we put a lot of weight on the Jump Loss or when we use a coarse time grid.
- The Total Loss can be large when using the dBSDE-Jump algorithm. This means that the pay-off function is not fitted correctly. This issue can be resolved by making sure that the Jump Loss cannot be too large. This happens when the jump intensity is small or when the jump sizes are small. For problems where we have large or many jumps, we can still achieve a small Jump Loss by using a fine grid or by putting most of the weight on the Terminal Loss.

6 Discussion and Conclusion

In this research, we solved high dimensional HJB-equations with jumps. Since grid-based methods suffer from the Curse of Dimensionality, we used methods where the solution is obtained using regression with neural networks. These algorithms discretize the time and use the underlying paths to fit the networks. Different time steps are connected by the FBSDEj, which describes the dynamics of the PDE we try to solve. Since the terminal conditions of the PDE were known, neural networks were able to learn this terminal time. We solved the problem both in the forward direction (dBSDE) and in the backward direction (DBDP).

The used DBDP methodology was slightly changed from the original algorithm by changing the online Monte Carlo simulation into an offline Monte Carlo. We tested this method by pricing Bermudan options. This method was able to price options correctly in 50 dimensions. When we have high intensity jumps or have many jumps, the algorithm showed a bias when we simulate the jumps with a Bernoulli approximation. Using the Poisson process still gave the desired result, but it can be significantly slower. When using the DBDP-MC algorithm, it is important that the neural networks are learned correctly, since we can only improve the solution by redoing (part of) the algorithm. This is important when the terminal payoff function is hard to learn, which occurs in high dimensional pay-off functions. When we do not use large enough batchsizes or iterations, it will likely give a wrong or biased solution. However, by using the DBDP carefully, we were able to solve high dimensional PDEs with this method.

Some problems cannot be directly solved by the DBDP-MC methodology. Other problems are more convenient to solve them in the forward direction. Therefore, we also extend the dBSDE such that it could solve PDEs with jumps. This methodology uses an extra set of neural networks to obtain the discounted jumps. These neural networks could then be learned by adding an extra term in the loss function. However, this leads to the possibility that the dBSDE algorithm does not give the wanted solution. In this work, we showed that the dBSDE can get the correct mean at terminal time, even if a correct fit cannot be made. When we want to price European options, this results in the correct solution at initial time. However, this made the method similar to a plain Monte Carlo simulation. The benefit is that it can price options with different initial asset prices at the same time. This problem can be solved by adding more weight on the terminal time loss and less on the new Jump Loss. This made learning the solution slower.

This work can be improved by trying more complicated examples. For the DBDP-MC, it will be important to tune the hyperparameters carefully. Alternatively, remember that HJB-equations are PDEs with an inner optimization problem. In this work, we solve all optimization problems analytically and substitute the solution into the PDE. However, the analytic solution of the optimization problem can be hard or impossible to calculate. This suggest that it is useful to solve the optimization problem and the HJB-equation simultaneously. The Deep Galerkin Method has been able to do this by alternating between the DGM and a Policy Improvement Algorithm (PIA). It is interesting to show whether the DBDP-PIA algorithm can solve HJB-equations. At last, we noticed that the dBSDE-Jump method may not solve the PDE associated with a jump process correctly. We can overcome this problem by using a hierarchical structure of the loss functions. This structure may be solved learning the Terminal Loss and Jump Loss in an alternating order.

References

[1] Oscar Christian Ameln. "Deep Learning Algorithms for Solving PDEs-Presentation and Implementation of Deep Learning Algorithms for Solving Semi-Linear Parabolic PDEs with an Extension to the Fractional Laplace Operator". MA thesis. NTNU, 2020.

[2] Kristoffer Andersson, Adam Andersson, and Cornelis W Oosterlee. "Convergence of a robust deep FBSDE method for stochastic control". In: *SIAM Journal on Scienti c Computing* 45.1 (2023), A226–A255.

[3] Kristoffer Andersson and Cornelis W Oosterlee. "D-TIPO: Deep time-inconsistent portfolio optimization with stocks and options". In: *arXiv preprint arXiv:2308.10556* (2023).

[4] Ali Al-Aradi et al. "Extensions of the deep Galerkin method". In: *Applied Mathematics and Computation* 430 (2022), p. 127287.

[5] Marco Avellaneda and Sasha Stoikov. "High-frequency trading in a limit order book". In: *Quantitative Finance* 8.3 (2008), pp. 217–224.

[6] Guy Barles, Rainer Buckdahn, and Etienne Pardoux. "Backward stochastic differential equations and integral-partial differential equations". In: *Stochastics: An International Journal of Probability and Stochastic Processes* 60.1-2 (1997), pp. 57–83.

[7] Andrew R Barron. "Universal approximation bounds for superpositions of a sigmoidal function". In: *IEEE Transactions on Information theory* 39.3 (1993), pp. 930–945.

[8] Christian Beck, Weinan E, and Arnulf Jentzen. "Machine learning approximation algorithms for high-dimensional fully nonlinear partial differential equations and second-order backward stochastic differential equations". In: *Journal of Nonlinear Science* 29 (2019), pp. 1563–1619.

[9] Richard Ernest Bellman. "Dynamic Programming". In: (1957).

[10] Alberto Bemporad, Leonardo Bellucci, and Tommaso Gabbriellini. "Dynamic option hedging via stochastic model predictive control based on scenario simulation". In: *Quantitative Finance* 14.10 (2014), pp. 1739–1751.

[11] Julius Berner et al. *The modern mathematics of deep learning*. 2021.

[12] J Frédéric Bonnans, Élisabeth Ottenwaelter, and Housnaa Zidani. "A fast algorithm for the two dimensional HJB equation of stochastic control". In: *ESAIM: Mathematical Modelling and Numerical Analysis* 38.4 (2004), pp. 723–735.

[13] Bruno Bouchard and Romuald Elie. "Discrete-time approximation of decoupled forward–backward SDE with jumps". In: *Stochastic Processes and their Applications* 118.1 (2008), pp. 53–75.

[14] Alina Braun et al. "The smoking gun: Statistical theory improves neural network estimates". In: *arXiv preprint arXiv:2107.09550* (2021).

[15] Andrei Caragea, Philipp Petersen, and Felix Voigtlaender. "Neural network approximation and estimation of classifiers with classification boundary in a Barron class". In: *The Annals of Applied Probability* 33.4 (2023), pp. 3039–3079.

[16] Álvaro Cartea, Sebastian Jaimungal, and Jason Ricci. "Buy low, sell high: A high frequency trading perspective". In: *SIAM Journal on Financial Mathematics* 5.1 (2014), pp. 415–444.

[17] Javier Castro. "Deep learning schemes for parabolic nonlocal integro-differential equations". In: *Partial Di erential Equations and Applications* 3.6 (2022), p. 77.

[18] Ke Chen, Chunmei Wang, and Haizhao Yang. "Deep Operator Learning Lessens the Curse of Dimensionality for PDEs". In: *arXiv preprint arXiv:2301.12227* (2023).

[19] Fei Cong and Cornelis W Oosterlee. "Pricing Bermudan options under Merton jump-diffusion asset dynamics". In: *International Journal of Computer Mathematics* 92.12 (2015), pp. 2406–2432.

[20] George Cybenko. "Approximation by superpositions of a sigmoidal function". In: *Mathematics of control, signals and systems* 2.4 (1989), pp. 303–314.

[21] Jérôme Darbon, Gabriel P Langlois, and Tingwei Meng. "Overcoming the curse of dimensionality for some Hamilton–Jacobi partial differential equations via neural network architectures". In: *Research in the Mathematical Sciences* 7.3 (2020), pp. 1–50.

[22] Narayan Ganesan, Yajie Yu, and Bernhard Hientzsch. "Pricing barrier options with deepBSDEs". In: *arXiv preprint arXiv:2005.10966* (2020).

[23] Maximilien Germain, Huyen Pham, and Xavier Warin. "Deep backward multistep schemes for nonlinear PDEs and approximation error analysis". In: *arXiv preprint arXiv:2006.01496* (2020).

[24] Alessandro Gnoatto, Marco Patacca, and Athena Picarelli. "A deep solver for BSDEs with jumps". In: *arXiv preprint arXiv:2211.04349* (2022).

[25] Alessandro Gnoatto, Athena Picarelli, and Christoph Reisinger. "Deep xVA Solver: A Neural Network–Based Counterparty Credit Risk Management Framework". In: *SIAM Journal on Financial Mathematics* 14.1 (2023), pp. 314–352.

[26] Philipp Grohs and Lukas Herrmann. "Deep neural network approximation for high-dimensional parabolic Hamilton-Jacobi-Bellman equations". In: *arXiv preprint arXiv:2103.05744* (2021).

[27] Philipp Grohs et al. "A proof that artificial neural networks overcome the curse of dimensionality in the numerical approximation of Black-Scholes partial differential equations". In: *arXiv preprint arXiv:1809.02362* (2018).

[28] Jiequn Han, Arnulf Jentzen, and Weinan E. "Solving high-dimensional partial differential equations using deep learning". In: *Proceedings of the National Academy of Sciences* 115.34 (2018), pp. 8505–8510.

[29] Pierre Henry-Labordere. "Deep primal-dual algorithm for BSDEs: Applications of machine learning to CVA and IM". In: *Available at SSRN 3071506* (2017).

[30] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. "Multilayer feedforward networks are universal approximators". In: *Neural networks* 2.5 (1989), pp. 359–366.

[31] Côme Huré, Huyên Pham, and Xavier Warin. "Deep backward schemes for high-dimensional nonlinear PDEs". In: *Mathematics of Computation* 89.324 (2020), pp. 1547–1579.

[32] Moshe Leshno et al. "Multilayer feedforward networks with a nonpolynomial activation function can approximate any function". In: *Neural networks* 6.6 (1993), pp. 861–867.

[33] Francis A Longstaff and Eduardo S Schwartz. "Valuing American options by simulation: a simple least-squares approach". In: *The review of nancial studies* 14.1 (2001), pp. 113–147.

[34] Jianfeng Lu et al. "Deep network approximation for smooth functions". In: *SIAM Journal on Mathematical Analysis* 53.5 (2021), pp. 5465–5506.

[35] Bernt Øksendal and Agnes Sulem. *Stochastic Control of jump di usions*. Springer, 2006.

[36] Cornelis W Oosterlee and Lech A Grzelak. *Mathematical modeling and computation in nance: with exercises and Python and MATLAB computer codes*. World Scientific, 2019.

[37] *Options Calculator*. `http://math.columbia.edu/~smirnov/options13.html`. Accessed: 2023-10-02.

[38] Huyên Pham. *Continuous-time stochastic control and optimization with nancial applications*. Vol. 61. Springer Science & Business Media, 2009.

[39] Huyen Pham, Xavier Warin, and Maximilien Germain. "Neural networks-based backward scheme for fully nonlinear PDEs". In: *SN Partial Di erential Equations and Applications* 2.1 (2021), pp. 1–24.

[40] Tomaso Poggio et al. "Why and when can deep-but not shallow-networks avoid the curse of dimensionality: a review". In: *International Journal of Automation and Computing* 14.5 (2017), pp. 503–519.

[41] Uri Shaham, Alexander Cloninger, and Ronald R Coifman. "Provable approximation properties for deep neural networks". In: *Applied and Computational Harmonic Analysis* 44.3 (2018), pp. 537–557.

[42] Justin Sirignano and Konstantinos Spiliopoulos. "DGM: A deep learning algorithm for solving partial differential equations". In: *Journal of computational physics* 375 (2018), pp. 1339–1364.

[43] Tomasz Szandała. “Review and comparison of commonly used activation functions for deep neural networks”. In: *Bio-inspired neurocomputing*. Springer, 2021, pp. 203–224.

[44] Jan Hendrik Witte and Christoph Reisinger. “A penalty method for the numerical solution of Hamilton–Jacobi–Bellman (HJB) equations in finance”. In: *SIAM Journal on Numerical Analysis* 49.1 (2011), pp. 213–231.

[45] Stephan Wojtowytsch and E Weinan. “Can shallow neural networks beat the curse of dimensionality? a mean field training perspective”. In: *IEEE Transactions on Arti cial Intelligence* 1.2 (2020), pp. 121–129.

[46] Dmitry Yarotsky. “Error bounds for approximations with deep ReLU networks”. In: *Neural Networks* 94 (2017), pp. 103–114.

Appendix A: Proofs of section 2

Theorem 1. *Let X_t be a process. Let f(t,x) be twice di erentiable and $Y_t = g(t, X_t)$. Then*

$$dY_t = \frac{\partial f}{\partial t}dt + \frac{\partial f}{\partial x}dX + \frac{\partial^2 f}{\partial x^2}d < X, X >$$

Proof. Let $0 = t_0 \quad t_1 \quad {}_N = t$ be a partition of [0,t]. Set $dt = t_j \quad t_{j\ 1}$ and $dX = X_{t_j} \quad X_{t_{j-1}}$. We calculate he Taylor expansion of f

$$f(t, X_t) = f(0, X_{t_0}) + \sum_i \frac{\partial f(t_j, X_{t_j})}{\partial t}dt + \sum_i \frac{\partial f(t_j, X_{t_j})}{\partial x}dX + \sum_i \frac{\partial^2 f(t_j, X_{t_j})}{\partial t \partial x}d < t, X >$$

$$+ \sum_i \frac{\partial^2 f(t_j, X_{t_j})}{\partial x^2}d < X, X > + \sum_i R_i$$

With $R_i = o(dt + (dX)^2)$. Since $< t, X > = 0$, the 3rd term is 0. By creating finer meshes, we get

$$\sum_i \frac{\partial f(t_j, X_{t_j})}{\partial t}dt \ \ ! \ \int_0^t \frac{\partial f(s, X_s)}{\partial t}ds$$

$$\sum_i \frac{\partial f(t_j, X_{t_j})}{\partial x}dX \ \ ! \ \int_0^t \frac{\partial f(s, X_s)}{\partial x}dX_s$$

$$\sum_i \frac{\partial^2 f(t_j, X_{t_j})}{\partial x^2}d < X, X > \ \ ! \ \int_0^t \frac{\partial^2 f(s, X_s)}{\partial x^2}d < X, X >_s$$

This shows what we wanted. □

Corollary 1.1. *Let X_t be an Itô di usion process*

$$dX_t = \mu(t, x)dt + \sigma(t, x)dW_t$$

Let f(t,x) be twice di erentiable and $Y_t = g(t, X_t)$. Then

$$dY_t = (\frac{\partial f}{\partial t}dt + \mu\frac{\partial f}{\partial x} + \sigma\frac{\partial^2 f}{\partial x^2})dt + \sigma\frac{\partial f}{\partial x}dW_t$$

Proof. First, we can note that

$$\begin{aligned} d < X, X >_t &= (dX_t)^2 \\ &= (\mu(t, x)dt + \sigma(t, x)dW_t)(\mu(t, x)dt + \sigma(t, x)dW_t) \\ &= \mu^2(dt)^2 + 2\mu\sigma dt dW_t + \sigma^2(dW_t)^2 \end{aligned}$$

From Itô table, we can directly see that

$$d < X, X >_t = \sigma^2 dt$$

Substituting X_t in theorem1 gives.

$$
\begin{aligned}
dY_t &= \frac{\partial f}{\partial t}dt + \frac{\partial f}{\partial x}dX + \frac{\partial^2 f}{\partial x^2}d < X, X > \\
&= \frac{\partial f}{\partial t}dt + \frac{\partial f}{\partial x}d(\mu dt + \sigma dW_t) + \frac{\partial^2 f}{\partial x^2}\sigma^2 dt \\
&= (\frac{\partial f}{\partial t}dt + \mu\frac{\partial f}{\partial x} + \sigma\frac{\partial^2 f}{\partial x^2})dt + \sigma\frac{\partial f}{\partial x}dW_t
\end{aligned}
$$

□

Lemma 2. *The in nitesimal generator given by*

$$
\mathcal{L}f = \lim_{\Delta t \# 0} \frac{\mathsf{E}_x(f(X_{t+\Delta t})) \quad f(X_t)}{\Delta t}
$$

of an Itô di usion process is

$$
\mathcal{L}f = (\frac{\partial f}{\partial t} + \mu\frac{\partial f}{\partial x} + \sigma^2\frac{\partial^2 f}{\partial x^2})
$$

Proof. We use Itô lemma for diffusion processes (corollary 1.1) to show:

$$
\begin{aligned}
\mathcal{L}f &= \lim_{\Delta t \# 0} \frac{\mathsf{E}_x(f(X_{t+\Delta t})) \quad f(X_t)}{\Delta t} \\
&= \lim_{\Delta t \# 0} \frac{\mathsf{E}((\frac{\partial f}{\partial t} + \mu\frac{\partial f}{\partial x} + \sigma^2\frac{\partial^2 f}{\partial x^2})dt + \sigma\frac{\partial f}{\partial x}dW_t)}{\Delta t} \\
&= \lim_{\Delta t \# 0} \frac{(\frac{\partial f}{\partial t} + \mu\frac{\partial f}{\partial x} + \sigma^2\frac{\partial^2 f}{\partial x^2})dt}{\Delta t} \\
&= \frac{\partial f}{\partial t} + \mu\frac{\partial f}{\partial x} + \sigma^2\frac{\partial^2 f}{\partial x^2}
\end{aligned}
$$

Similarly, we can use a Taylor expansion to proof this. □

Theorem 3. *For all stopping times τ 2 $[t_0, T]$,*
$J(t, x, \alpha) = \mathsf{E}[J(\tau, X_{t_0}^{\tau,x}, \alpha) + \int_{t_0}^{\tau} l(s, X_{t_0}^{s,x}, \alpha_s)ds]$

Proof. Fix a stopping time τ then

$$
\begin{aligned}
J(t, x, \alpha) &= \mathsf{E}[[g(X_{t_0}^{T,x}) + \int_{t_0}^{T} l(s, X_{t_0}^{s,x}, \alpha_s)ds] \\
&= \mathsf{E}[\mathsf{E}[g(X_{t_0}^{T,x}) + \int_{t_0}^{T} l(s, X_{t_0}^{s,x}, \alpha_s)ds \mathsf{j}\mathcal{F}_{\tau}^{t_0}]] \\
&= \mathsf{E}[\int_{t_0}^{\tau} l(s, X_{t_0}^{s,x}, \alpha_s)ds + \mathsf{E}[g(X_{t_0}^{T,x}) + \int_{\tau}^{T} l(s, X_{t_0}^{s,x}, \alpha_s)ds \mathsf{j}\mathcal{F}_{\tau}^{t_0}]] \\
&= \mathsf{E}[\int_{t_0}^{\tau} l(s, X_{t_0}^{s,x}, \alpha_s)ds + \mathsf{E}[g(X_{\tau}^{T, X_{t_0}^{\tau,x}}) + \int_{\tau}^{T} l(\tau, X_{\tau}^{s, X_{t_0}^{\tau,x}}, \alpha_s)ds \mathsf{j}\mathcal{F}_{\tau}^{t_0}]] \\
&= \mathsf{E}[\int_{t_0}^{\tau} l(s, X_{t_0}^{s,x}, \alpha_s)ds + J(\tau, X_{\tau}^{t_0,x}, \alpha)]
\end{aligned}
$$

Which is what we wanted. We used Markov property in step 4 and $\mathsf{E}(X) = \mathsf{E}(\mathsf{E}(X|Y))$ in step 2. □

Theorem 4. *Let X_t be a controlled Markov process, then for all $\tau \in [t_0, T]$*
$v(t_0, x) = \inf_{a \in A} \mathsf{E}[\int_{t_0}^{\tau} l(s, X_{t_0}^{s,x}, \alpha_s)ds + v(\tau, X_{\tau}^{t_0,x})]$

Proof. Using theorem 3 and the definition of $v(t, x)$, we get

$$J(t, x, \alpha) = \mathsf{E}[\int_{t_0}^{\tau} l(s, X_{t_0}^{s,x}, \alpha_s)ds + J(\tau, X_{\tau}^{t_0,x}, \alpha)]$$
$$\mathsf{E}[\int_{t_0}^{\tau} l(s, X_{t_0}^{s,x}, \alpha_s)ds + v(\tau, X_{\tau}^{t_0,x})]$$

Since this holds for all cost function, it also holds for $v(t, x)$

$$v(t, x) \quad \mathsf{E}[\int_{t_0}^{\tau} l(s, X_{t_0}^{s,x}, \alpha_s)ds + v(\tau, X_{\tau}^{t_0,x})] \tag{57}$$

We show the converse. We rewrite the control as followed:

$$\hat{\alpha} = \begin{cases} \alpha_s, & s \in [0, \tau] \\ \alpha_s^{\epsilon}, & s \in [\tau, T] \end{cases} \tag{58}$$

Here α_s^{ϵ} is a ϵ-optimal control for $v(t, x)$:

$$v(\tau, X) + \epsilon \quad J(\tau, X, \alpha_s^{\epsilon})$$

The control $\hat{\alpha_s}$ is progressively measurable, as shown in [38]. This control allows the following argument using theorem 3:

$$v(t, x) \quad J(t, x, \hat{\alpha})$$
$$= \mathsf{E}[\int_{t_0}^{\tau} l(s, X_{t_0}^{s,x}, \alpha_s)ds + J(\tau, X_{\tau}^{t_0,x}, \alpha^{\epsilon})]$$
$$\mathsf{E}[\int_{t_0}^{\tau} l(s, X_{t_0}^{s,x}, \alpha_s)ds + v(\tau, X_{\tau}^{t_0,x}) + \epsilon]$$

Letting ϵ go to zero gives the wanted result.

$$v(t, x) \quad \mathsf{E}[\int_{t_0}^{\tau} l(s, X_{t_0}^{s,x}, \alpha_s)ds + v(\tau, X_{\tau}^{t_0,x})] \tag{59}$$

Combining (1) and (2) gives the desired result.

□

Theorem 5. *Let X_t be a Feller (and therefore Markovian) process. Take the cost function*

$$J(a) = \mathsf{E}_x(g(x_T) + \int_{t_0}^{T} l(x_s, \alpha_s)ds)$$

, where

$$\mathsf{E}_x(\ ) = \mathsf{E}(\ jX_0 = x)$$

Then this is equivalent to the PDE

$$inf_{a\,2\,A}\, \mathsf{L}\, u(t,x) + l(x_t, \alpha_t)) = 0$$

where L the in nitesimal generator given by

$$\mathsf{L}\, f = \lim_{\Delta t \# 0} \frac{\mathsf{E}_x(f(X_{t+\Delta t})) \quad f(X_t)}{\Delta t}$$

Proof. Take a starting time t and a stop time $t + \Delta_t$. From theorem 4, we have

$$u(t,x) = inf_{a\,2\,A}\, \mathsf{E}_x(u(t+\Delta t, x) + \int_t^{t+\Delta t} l(x_s, \alpha_s) ds)$$

$$0 = inf_{a\,2\,A}\, \mathsf{E}_x(u(t+\delta t, x)) \quad u(t,x) + \mathsf{E}_x(\int_t^{t+\Delta t} l(x_s, \alpha_s) ds)$$

Now divide by Δt end let it go to zero. This gives

$$0 = inf_{a\,2\,A} + \mathsf{L}\, u(y,x) + l(x,\alpha)$$

□

Appendix B: Proof of Itô lemma with jumps

In this section, we proof Itô lemma for Lévy processes. We will proof it for the combined finite and infinite activity process. First, we show the Itô lemma of a pure, non-compensated jump process.

Theorem 16. *The Itô formula of $dX_t = \int_A \gamma(t,z,x)N(dt,dz)$, where A a set with a lower bound, is*

$$df(X_t) = \int_A f(X_t \quad + \gamma(t,z,x)) \quad f(X_t \quad))N(dt,dz)$$

Proof. We will use that ^ denotes the minimum. Let the time of the jump be given by τ, with $\tau_0 = 0$.

$$\tau_n = \inf\{t > \tau_{n \quad 1}; \Delta X \ 2 \ A\}$$

since, X is a pure jump process, the whole process is described by these jumps at times τ_i.

$$\begin{aligned}
f(X_t) \quad f(X_0) &= \sum_{0 \quad s \quad t} \Delta X \\
&= \sum_{0 \quad s \quad t} f(X_s) \quad f(X_s \quad) \\
&= \sum_{i=1}^{1} f(X_{t \wedge \tau_i}) \quad f(X_{t \wedge \tau_{i-1}}) \\
&= \sum_{i=1}^{1} f(X_{t \wedge \tau_i} \quad + \gamma(t \wedge \tau_i, \Delta X, X) \quad f(X_{t \wedge \tau_i} \quad) \\
&= \int_0^t \int_A f(X_t \quad + \gamma(t,z,x) \quad f(X_t \quad)N(dt,dz)
\end{aligned}$$

Writing this in differential form gives the desired result. □

We want to extend Theorem (16) by adding its continuous Itô-diffusion part. The proof uses a similar decomposition as in the proof of Theorem (16).

Theorem 17. *The Itô formula of $dX_t = \mu(t,x)dt + \sigma(t,x)dW_t + \int_A \gamma(t,z,x)N(dt,dz)$, where A a set with a lower bound, is*

$$df(X_t) = (\frac{\partial f}{\partial t} + \mu\frac{\partial f}{\partial x} + \frac{1}{2}\sigma^2\frac{\partial^2 f}{\partial x^2})dt + \sigma\frac{\partial f}{\partial x}dW_t + \int_A f(X_t \quad + \gamma(t,z,x)) \quad f(X_t \quad))N(dt,dz)$$

Proof. Define τ as in the previous proof.

$$\begin{aligned}
f(X_t) \quad f(X_0) &= \sum_{i=1}^{1} f(X_{t \wedge \tau_i}) \quad f(X_{t \wedge \tau_{i-1}}) \\
&= \sum_{i=1}^{1} f(X_{t \wedge \tau_i}) \quad f(X_{t \wedge \tau_i} \quad) + f(X_{t \wedge \tau_i} \quad) \quad f(X_{t \wedge \tau_{i-1}}) \\
&= \sum_{i=1}^{1} f(X_{t \wedge \tau_i}) \quad f(X_{t \wedge \tau_i} \quad) + \sum_{i=1}^{1} f(X_{t \wedge \tau_i} \quad) \quad f(X_{t \wedge \tau_{i-1}})
\end{aligned}$$

The left-hand-side shows that we have split it between the jump component (left sum) and a continuous part (right sum). The jump sum was already shown in Theorem (16). Note that there are no jumps in the left sum. This means that it is a sum of the Itô lemma for he Itô-diffusion case, Theorem (1). Adding both Itô formula and taking the differential form shows what we want. □

The only difference between the Itô lemma for the process in Theorem (17) and the Itô lemma for the wanted process in Theorem (8) is the compensation of the Poisson process.

Theorem 9. *The Itô formula of a Levy process as described in theorem (8) is*

$$\begin{aligned} df(X_t) = (&\frac{\partial f}{\partial t} + \mu\frac{\partial f}{\partial x} + \frac{1}{2}\sigma^2\frac{\partial^2 f}{\partial x^2})dt + \sigma\frac{\partial f}{\partial x}dW_t \\ &+ \int_{\mathsf{R}} f(X_t \quad + \gamma(t,z,x)) \quad f(X_t \quad))\bar{N}(dt,dz) \\ &+ \int_{jjzjj<R} [f(X_t \quad + \gamma(t,z,x)) \quad f(X_t \quad)) \quad \gamma(t,z,x)\frac{\partial f}{\partial x}]\nu(dz)dt \end{aligned}$$

Proof. First, we rewrite the Lévy process such that the compensation part is put in the drift of the process.

$$dX_t = [\mu(t,x) \quad \int_{jjzjj<R} \gamma(t,z,x)\nu dz]dt + \sigma(t,x)dW_t + \int_{\mathsf{R}} \gamma(s,z,x)N(dt,dz)$$

We can now obtain the Itô formula using Theorem (17). Rearranging finishes the proof. □

Appendix C: A simple example: dBSDE-Jump might lead to an incorrect solution

In this section, we will give a very simple example to show that the dBSDE-Jump method can reach a wrong solution. This example assumes a pure jump process, e.g. $b(t,X) = \sigma(t,X) = 0$ and that we have a deterministic jump sizes $\Gamma(t,X) = 1$. This means that the compensation component of the jumps in this underlying path is $V = \frac{1}{2}$. We assume that there can only be one jump per time step and simulate this using a Bernoulli distribution. The results will no be the same as when we would use a Poisson distribution, but we ignore this to make the example easier. We set the chance for a jump at 0.5, $\lambda = 0.5$. At last, we only look at 1 time step. We set $X_0 = 1$ and we set $g(X_T) = X_T$ as the terminal condition. The solution Y_i, U_i can in this example be written as $G(X) = U(X + \Gamma) \quad U(X)$ and we write the compensation part of the jumps as V. This gives the following FBSDEj.

$$X_1 = X_0 + \sum_{\text{jump}} \Gamma \quad V = \frac{1}{2} + \sum_{\text{jump}} 1$$
$$Y_1 = Y_0 + \sum_{\text{jump}} G(X_0) \quad V$$

We note that we have 3 networks Y_0, G and V. We now write the Terminal Loss and Jump Loss. Note that when no jump occurs, $G = 0$

$$\begin{aligned}
\text{Terminal loss} &= \mathsf{E}((g(X_1) \quad Y_1)^2) \\
&= \mathsf{E}((\frac{1}{2} + \sum_{\text{jump}} 1 \quad (Y_0 + \sum_{\text{jump}} G(X_0) \quad V))^2) \\
&= \frac{1}{2}((\frac{1}{2} \quad (Y_0 \quad V))^2 + (\frac{3}{2} \quad (Y_0 + G \quad V))^2) \\
\text{Jump loss} &= \mathsf{E}((G(X) \quad V(X))^2) \\
&= \frac{1}{2}(V^2 + (G \quad V)^2)
\end{aligned}$$

We show the optimal answer that the dBSDE-Jump algorithm obtains and compare it to the desired values.

Table 6

	$\mathcal{Y}_0$	$\mathcal{G}$	$\mathcal{V}$	Jump Loss	Terminal Loss	Total Loss
dBSDE-Jump	1	$\frac{1}{2}$	$\frac{1}{4}$	$\frac{1}{16}$	$\frac{1}{16}$	$\frac{1}{8}$
Desired	1	1	$\frac{1}{2}$	0	$\frac{1}{4}$	$\frac{1}{4}$

This shows that the dBSDE-Jump obtain incorrect values for G and G, but the correct value for Y_0. Moreover, the Total Loss of this algorithm is less than what we should get when solving the FBSDEj correctly with, for example, the H-dBSDE. The problem is that the Jump and Terminal Loss "balance" each other out. Table 6 demonstrates this.