Maria Rebecca Rothe

Numerical Experiments on a Deep Learning Approach to Solving High-Dimensional Partial Differential Equations

Master's thesis in Applied Physics and Mathematics
Supervisor: Helge Holden
June 2020

Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Mathematical Sciences

Norwegian University of Science and Technology

Maria Rebecca Rothe

Numerical Experiments on a Deep Learning Approach to Solving High-Dimensional Partial Differential Equations

Master's thesis in Applied Physics and Mathematics
Supervisor: Helge Holden
June 2020

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Mathematical Sciences

NTNU
Kunnskap for en bedre verden

Summary

This Master's thesis explores a deep learning approach to solving high-dimensional partial differential equations. Numerical analysis of partial differential equations is of great importance as it can describe essential physical phenomenons like heat distribution and wave propagation. The complexity of traditional numerical methods usually increases exponentially with the dimensionality of the problem, limiting numerical analysis and modeling in the high-dimensional case. In this thesis, the method introduced in the paper *Solving high-dimensional partial differential equations using deep learning* (Han et al., 2018) is explored. The proposed method aims to solve high-dimensional partial differential equations with lower computational cost than traditional numerical methods. To do so, a technique of artificial intelligence called deep learning is utilized. The method is named the deep BSDE method, from the utilization of deep learning and backward stochastic differential equations (BSDEs). The method considers the class of semilinear parabolic partial differential equations.

This thesis presents the necessary background theory for understanding the deep BSDE method. This includes an introduction to artificial intelligence, where deep neural networks are explained. The connection between semilinear parabolic partial differential equations and backward stochastic differential equations is presented. The methodology is described along with an explanation of the neural network architecture. Implementation details are provided, and essential algorithms are presented and discussed. The method is tested on the Allen-Cahn equation and the Hamilton-Jacobi-Bellman equation. The Allen-Cahn equation is a reaction-diffusion equation that describes phase separation processes, and the Hamilton-Jacobi-Bellman equation is a result of applying dynamic programming to continuous optimal control problems. Further numerical experiments are also conducted, exploring how different features of the method affect the performance.

The deep BSDE method is implemented in the machine learning platform TensorFlow, and the numerical results are satisfying, achieving both high accuracy and low computational cost. The method achieved a relative approximation error of 0.20% for the Allen-Cahn equation, and 0.22% for the Hamilton-Jacobi-Bellman equation. The promising results open up the possibility of solving more complex and demanding problems in several areas, such as economics, finance, operational research, and physics.

Sammendrag

Denne masteroppgaven utforsker en metode for å løse høydimensjonale partielle differentialligninger ved bruk av dyp læring. Numerisk analyse av partielle differentialligninger er av stor betydning ettersom det kan beskrive viktige fysiske fenomener slik som varmefordeling og bølgeforplantning. Kompleksiteten i tradisjonelle numeriske metoder øker vanligvis eksponensielt med dimensjonaliteten til problemet, og begrenser dermed numerisk analyse og modellering i høydimensjonale tilfeller. I denne oppgaven vil metoden som introduseres i artikkelen *Solving high-dimensional partial differential equations using deep learning* (Han et al., 2018) bli utforsket. Den foreslåtte metoden har som mål å løse høydimensjonale partielle differentialligninger med lavere beregningskostnader enn tradisjonelle numeriske metoder. For å oppnå dette benyttes en teknikk innen kunstig intelligens som kalles dyp læring. Metoden heter *deep BSDE*-metoden, fra bruken av dyp læring og bakover stokastiske differentialligninger (BSDEs). Metoden tar for seg gruppen med semilineære parabolske partielle differentialligninger.

Denne oppgaven presenterer den nødvendige bakgrunnsteorien for å forstå *deep BSDE*-metoden. Dette inkluderer en introduksjon til kunstig intelligens, der dype nevrale nettverk blir forklart. Forbindelsen mellom semilinære parabolske partielle differensialligninger og bakover stokastiske differensialligninger blir presentert. Metodikken blir beskrevet sammen med en forklaring av den nevrale nettverksarkitekturen. Implementeringsdetaljer blir gitt, og viktige algoritmer blir presentert og diskutert. Metoden blir testet på Allen-Cahn-ligningen og Hamilton-Jacobi-Bellman-ligningen. Allen-Cahn-ligningen er en reaksjon-diffusjonsligning som beskriver faseseparasjonsprosesser, og Hamilton-Jacobi-Bellman-ligningen er et resultat av å bruke dynamisk programmering på kontinuerlige optimale kontrollproblemer. Ytterligere numeriske eksperimenter blir også utført for å undersøke hvordan forskjellige egenskaper ved metoden påvirker ytelsen.

Deep BSDE-metoden implementeres i maskinlæringsplattformen TensorFlow, og de numeriske resultatene er tilfredsstillende med både høy nøyaktighet og lave beregningskostnader. Metoden oppnådde en relativ tilnærmingsfeil på 0.20% for Allen-Cahn-ligningen, og 0.22% for Hamilton-Jacobi-Bellman-ligningen. Resultatene er lovende og åpner for muligheten for å løse mer komplekse og krevende problemer på flere områder, som økonomi, finans, operativ forskning og fysikk.

Preface

This thesis is written during the spring of 2020 as the final assignment of my M.Sc. degree in Applied Physics and Mathematics with specialization in Industrial Mathematics at the Department of Mathematical Sciences, Norwegian University of Science and Technology (NTNU). It has offered me the opportunity to utilize knowledge that I have acquired from my studies, as well as attain a deeper understanding of stochastic differential equations and deep learning. It has been motivating to work with a recently developed method that uses highly relevant technology.

The thesis aims to explore a deep learning approach to solving high-dimensional partial differential equations through numerical experiments. The method is implemented in the machine learning platform TensorFlow, and tested on two high-dimensional partial differential equations.

I would like to thank my academic supervisor, Helge Holden, for the guidance and feedback he has provided. I would also like to thank my fellow students for contributing to a motivating learning environment, and making my time at NTNU enjoyable. Lastly, I want to give special thanks to my parents for their continuous support and encouragement.

Maria Rebecca Rothe
Trondheim, June 2020

Table of Contents

Summary i

Sammendrag iii

Preface v

Table of Contents ix

List of Tables xii

List of Figures xvi

Abbreviations xvii

1 Introduction 1
- 1.1 Motivation 1
- 1.2 Related Work 2
- 1.3 Thesis Structure 3

2 Background Theory 5
- 2.1 Deep Learning 5
  - 2.1.1 Introduction to Deep Learning 5
  - 2.1.2 Artificial Neural Networks 6
  - 2.1.3 Multilayer Feedforward Perceptron Model 9
  - 2.1.4 Deep Reinforcement Learning 10
  - 2.1.5 The Expressive Power of Deep Neural Networks 13
- 2.2 Viewing Partial Differential Equations as Stochastic Control Problems . . 14
  - 2.2.1 Stochastic Control Problems 14
  - 2.2.2 Backward Stochastic Differential Equation Reformulation 14
- 2.3 Example Partial Differential Equations 20
  - 2.3.1 The Allen-Cahn Equation 20
  - 2.3.2 The Hamilton-Jacobi-Bellman Equation 21

3 Methodology 25
3.1 Equation Reformulation 25
3.2 Approximation Using Deep Learning 26
3.3 Network Architecture 27

4 Implementation 29
4.1 Activation Function 29
4.2 Optimization Algorithm 30
4.3 Batch Normalization 32
4.4 Exact Solution Methods 32
4.4.1 Branching-Diffusion Method for the Allen-Cahn Equation 33
4.4.2 Monte Carlo Method for the Hamilton-Jacobi-Bellman Equation . 33
4.5 Machine Learning Platform 33
4.6 The Deep BSDE Method Implementation 34

5 Numerical Experiments 41
5.1 The Allen-Cahn Equation 41
5.1.1 Model Variables 42
5.1.2 Numerical Results 42
5.2 The Hamilton-Jacobi-Bellman Equation 47
5.2.1 Model Variables 47
5.2.2 Numerical Results 48
5.3 Model Features 52
5.3.1 Type of Activation Function 52
5.3.2 Batch Size 54
5.3.3 Temporal Discretization Steps 57

6 Discussion 61
6.1 On Numerical Results 61
6.2 On Model Features 62
6.3 On Convergence 63
6.3.1 A Posteriori Error Estimation 64
6.3.2 Upper Bound of Optimal Loss 64

7 Conclusion 67
7.1 Summary 67
7.2 Future Work 68

Bibliography 69

Appendix 73
A Python code for learning the sine function 73
B Python code for solving the Allen-Cahn equation using the deep BSDE method 74
C Python code for solving the Hamilton-Jacobi-Bellman equation using the deep BSDE method 78

D Numerical results for the Allen-Cahn equation 82
E Numerical results for the Hamilton-Jacobi-Bellman equation 83

List of Tables

4.1 The variables that are defined in the `__init__` function in the solver classes. 35

5.1 The values of the model variables used to solve the Allen-Cahn equation using the deep BSDE method. 42
5.2 Numerical results after solving the Allen-Cahn equation with values in **Table 5.1** and for five independent runs using random seeds $\{1, 2, 3, 4, 5\}$. The runtime is given for one of the runs in seconds. 43
5.3 Numerical results after solving the Allen-Cahn equation with values in **Table 5.1** and for five independent runs using random seeds $\{6, 7, 8, 9, 10\}$. The runtime is given for one of the runs in seconds. 44
5.4 The values of the model variables used to solve the HJB equation using the deep BSDE method. 48
5.5 Numerical results after solving the HJB equation with values in **Table 5.4** and for five independent runs using random seeds $\{1, 2, 3, 4, 5\}$. The runtime is given for one of the runs in seconds. 49
5.6 Numerical results after solving the HJB equation with values in **Table 5.4** and for five independent runs using random seeds $\{6, 7, 8, 9, 10\}$. The runtime is given for one of the runs in seconds. 50
5.7 Numerical results after solving the Allen-Cahn equation with values in **Table 5.1** for five independent runs using four different activation functions. The runtime is given for one of the runs in seconds. 53
5.8 Numerical results after solving the HJB equation with values in **Table 5.4** for five independent runs using four different activation functions. The runtime is given for one of the runs in seconds. 54
5.9 Numerical results after solving the Allen-Cahn equation for different batch sizes. The rest of the model variables are set to the values in **Table 5.1**. The runtime is given in seconds. 57
5.10 Numerical results after solving the Allen-Cahn equation for different number of discretization steps. The rest of the model variables are set to the values in **Table 5.1**. The runtime is given in seconds. 60

List of Figures

2.1 Deep learning as an approach to AI; deep learning is a type of representation learning, which is a type of machine learning, which is an approach to AI. 6

2.2 An illustration of a simple ANN from chapter 18.7 in (Russell and Norvig, 2016). 1 and 2 are units in the input layer, 3 and 4 are units in the hidden layer, and lastly 5 and 6 are units in the output layer. The weight for the link between unit i and j is denoted by $w_{i,j}$. 7

2.3 A simple mathematical model for a neuron from chapter 18.7 in (Russell and Norvig, 2016). 7

2.4 An illustration of a simple ANN with two input units and one output unit. The weight for the link between unit i and j is denoted by $w_{i,j}$. With appropriate weights, this neural network can model the logical connectives AND and inclusive OR. 8

2.5 A plot of the approximation of the sine function using an MLP model with two hidden layers, each with 100 units. The green line is the function to be approximated $f^*(x) = \sin(x)$, and the blue circles mark the values predicted by the deep neural network for the respective values of x. 9

2.6 The agent-environment interaction in RL for a Markov process. 11

3.1 An illustration of an example of a subnetwork at time t_n. The subnetwork is fully connected, and the problem equation is 3-dimensional (3 units in the input and output layer). The neural network consists of 2 hidden layers, denoted by h_n^1 and h_n^2, with 5 units in each. Every line represents a parameter (or weight) that is to be optimized. All the parameters in this subnetwork are gathered in the set θ_n. 26

3.2 An illustration of the network architecture. The arrows show the flow of information in the network. Each column represents a time step in the discretization, t_n for $n = 0, 1, \ldots, N$. Each MLP subnetwork for $n = 1, 2, \ldots, N-1$ contains H hidden layers, denoted by h_n^i for $i = 1, 2, \ldots, H$. The different types of connections in the network are marked with (i), (ii) and (iii) corresponding to the given definitions. This illustration is based on the figure included in (Han et al., 2018). 28

4.1 Plots of different activation functions. **(a)** The rectified linear unit (ReLU). **(b)** The softplus function. **(c)** The sigmoid function. **(d)** The tanh function. 30
4.2 Illustration of the structure of the program code (for the entire program code, see Appendix B and C). The blue boxes refer to functions, and the white boxes describe the tasks that are performed in the parenting function. 36
4.3 Illustration of connection (ii) from Subsection 3.3. In the implementation, this connection is created in function `build` in variable scope `forward connections` (for the entire program code, see Appendix B and C). . . 36
4.4 Illustration of connection (i) from Subsection 3.3. In the implementation, this connection is created in function `build` using function `subnetwork` and `add_layer` (for the entire program code, see Appendix B and C). . 37
4.5 Illustration of connection (iii) from Subsection 3.3. In the implementation, this connection is created in function `train` using function `sample_path` (for the entire program code, see Appendix B and C). 37

5.1 A plot of the relative approximation error when solving the Allen-Cahn equation using the deep BSDE method as a function of number of iteration steps. The dark line is the mean of the relative error after five independent runs using the random seeds $\{1, 2, 3, 4, 5\}$, and the shaded area shows the mean $\pm$ the standard deviation of the relative error. 43
5.2 A plot of the relative approximation error when solving the Allen-Cahn equation using the deep BSDE method as a function of number of iteration steps. The dark line is the mean of the relative error after five independent runs using the random seeds $\{6, 7, 8, 9, 10\}$, and the shaded area shows the mean $\pm$ the standard deviation of the relative error. 44
5.3 A plot of the approximated initial value when solving the Allen-Cahn equation using the deep BSDE method as a function of number of iteration steps. The dark line is the mean of $\hat{u}_\theta$ after five independent runs, and the shaded area shows the mean $\pm$ the standard deviation of $\hat{u}_\theta$. . . . 45
5.4 A plot of the loss function when solving the Allen-Cahn equation using the deep BSDE method as a function of number of iteration steps. The dark line is the mean of the loss function after five independent runs, and the shaded area shows the mean $\pm$ the standard deviation of the loss function. 46
5.5 A plot of the time evolution of the solution to the Allen-Cahn equation, $u(t, (0, 0, ..., 0))$, using both the deep BSDE method and the branching-diffusion method. The blue graph is barely visible as the two graphs are superimposed. 46

5.6 A plot of the relative approximation error when solving the HJB equation using the deep BSDE method as a function of number of iteration steps. The dark line is the mean of the relative error after five independent runs using the random seeds $\{1, 2, 3, 4, 5\}$, and the shaded area shows the mean $\pm$ the standard deviation of the relative error. 49
5.7 A plot of the relative approximation error when solving the HJB equation using the deep BSDE method as a function of number of iteration steps. The dark line is the mean of the relative error after five independent runs using the random seeds $\{6, 7, 8, 9, 10\}$, and the shaded area shows the mean $\pm$ the standard deviation of the relative error. 50
5.8 A plot of the approximated initial value when solving the HJB equation using the deep BSDE method as a function of number of iteration steps. The dark line is the mean of $\hat{u}_\theta$ after five independent runs, and the shaded area shows the mean $\pm$ the standard deviation of $\hat{u}_\theta$. 51
5.9 A plot of the loss function when solving the HJB equation using the deep BSDE method as a function of number of iteration steps. The dark line is the mean of the loss function after five independent runs, and the shaded area shows the mean $\pm$ the standard deviation of the loss function. 52
5.10 A plot of the mean relative approximation error when solving the Allen-Cahn equation as a function of number of iteration steps. It is solved using the deep BSDE method for five independent runs using four different activation functions. 53
5.11 A plot of the mean relative approximation error when solving the HJB equation as a function of number of iteration steps. It is solved using the deep BSDE method for five independent runs using four different activation functions. The different graphs are difficult to identify as they are superimposed. 54
5.12 A plot of the relative approximation error when solving the Allen-Cahn equation as a function of number of iteration steps. It is solved using the deep BSDE method with different batch sizes. 55
5.13 A plot of the loss function when solving the Allen-Cahn equation as a function of number of iteration steps. It is solved using the deep BSDE method with different batch sizes. 56
5.14 A plot of the loss function after 4000 iterations for solving the Allen-Cahn equation as a function of batch size. 56
5.15 A plot of the relative approximation error when solving the Allen-Cahn equation as a function of number of iteration steps. It is solved using the deep BSDE method for different number of temporal discretization steps. 58
5.16 A plot of the relative approximation error after 4000 iteration steps for solving the Allen-Cahn equation as a function of number of temporal discretization steps. 58
5.17 A plot of the loss function when solving the Allen-Cahn equation as a function of number of iteration steps. It is solved using the deep BSDE method for different number of temporal discretization steps. 59

5.18 A plot of the loss function after 4000 iterations for solving the Allen-Cahn equation as a function of number of temporal discretization steps. 59

Abbreviations

AI	=	artificial intelligence
ANN	=	artificial neural network
BSDE	=	backward stochastic differential equation
HJB	=	Hamilton-Jacobi-Bellman
MLP	=	multilayer feedforward perceptron
PDE	=	partial differential equation
ReLU	=	rectified linear unit
RL	=	reinforcement learning
SDE	=	stochastic differential equation
SGD	=	stochastic gradient descent

Chapter 1

Introduction

This Master's thesis explains the method that is introduced in the paper *Solving high-dimensional partial differential equations using deep learning*, published by Proceedings of the National Academy of Sciences (Han et al., 2018). The goal is to implement the method and test it on two high-dimensional partial differential equations (PDEs). The thesis provides the reader with an understanding of the mathematical background and of the methodology that Han et al. present. Different features of the method are investigated through numerical experiments. This chapter presents the motivation for the work, and related work on the topic of solving high-dimensional partial differential equations. Finally, the structure of the thesis is presented.

1.1 Motivation

Machine learning is an old concept, but its use has recently exploded in the industry with a growing commitment to artificial intelligence solutions. With enormous data generation in today's society, numerous applications for artificial intelligence have been explored. In healthcare, it is used in medical diagnosis in for example determining breast cancer risk from mammograms (Wired, 2016). Speech recognition is another artificial intelligence application that Apple uses in their virtual assistant "Siri" (SiriTeam, 2017). Tesla also uses this technology in image analysis for autonomous cars (Marr, 2018a). Another interesting application is to optimize power control by short-term predictions on wind power production (Qureshi, 2017). After years of research the systems have become increasingly reliable, and are now employed in many industries. In 2015, an artificial intelligence system outperformed humans in a challenge of object classification in images. In 2016, an artificial intelligence system defeated the world champion in the game Go. And in 2018, the first autonomous cars hit the roads. These achievements are taken from the list *The most amazing artificial intelligence milestones so far* in (Marr, 2018b). The International Data Corporation predicts that the global spending on artificial intelligence systems will reach $37.5 billion in 2019, and grow to reach $97.9 billion in 2023 (IDC, 2019). Artificial intelligence is a highly relevant technology, and because of the great achievements

artificial intelligence systems are providing, it will undoubtedly be an important part of our future.

A recent field of study is using machine learning to solve PDEs, with the goal of solving higher-dimensional problems with lower computational cost. Traditional numerical methods suffer the so called curse of dimensionality. The curse of dimensionality means that the computational cost increases the more dimensions that are involved in the equation. The term was first introduced by Bellman in 1957. Deep learning is a technique of machine learning that has proven to be very effective in dealing with complex and high-dimensional problems such as image analysis and speech recognition. Han et al. (2018) shows promising results in achieving lower computational cost for solving high-dimensional PDEs using deep learning. Their presented method is named the deep BSDE method, from the utilization of deep learning and backward stochastic differential equation (BSDE) theory.

Numerical analysis of PDEs plays a key role in research and many industries. PDEs express connections between rates of change with respect to multiple independent variables. They can describe phenomenons like for example distribution of heat by the famous heat equation, and wave propagation by the famous wave equation. The Allen-Cahn equation can describe phase separation in alloys, and analysis of this process can be used to optimize materials (Kostorz, 1995). Due to the curse of dimensionality, the analysis can sometimes be limited when using traditional numerical methods. Sparsity as a consequence of increased dimensionality can lead to unreliable results, and growing computational cost can lead to overly time consuming solving and require unobtainable amounts of memory. Learning cost efficient ways of solving and analyzing PDEs in higher dimensions is therefore of great importance and extremely beneficial.

1.2 Related Work

Traditional numerical methods such as finite differences method and finite elements method suffer the curse of dimensionality. The problem usually becomes significant already when the equation exceeds four dimensions, and these methods are therefore usually unavailable in the high-dimensional case. Complex problems are today commonly tackled with simplifications and ad hoc assumptions to enable more practical solving methods.

There exist some efficient algorithms for solving high-dimensional problems. Specifically, for linear parabolic PDEs one can find the solution at any space-time point using Monte Carlo methods based on the Feynman-Kac formula. For a class of inviscid Hamilton-Jacobi equations, (Darbon and Osher, 2016) presents an effective algorithm based on results from compressed sensing and on the Hopf formulas for the Hamilton-Jacobi equations. It also exists a general algorithm for nonlinear parabolic PDEs that is based on the Feynman-Kac and Bismut-Elworthy-Li formula and a multilevel decomposition of Picard iteration (Hutzenthaler et al., 2017). It has proven to be very efficient in solving nonlinear parabolic PDEs in finance and physics. Semilinear PDEs with polynomial nonlinearity can be solved using the branching diffusion method (Henry-Labordere et al., 2014). This method exploits that the solution can be given as an expectation of a functional of branching diffusion processes. It does not suffer from the curse of dimensionality, but the approximated solutions can blow up in finite time and the method is

therefore limited in its applicability.

1.3 Thesis Structure

The thesis is structured as follows. Chapter 2 presents the necessary background theory to understand the deep BSDE method. The methodology is described stepwise in Chapter 3 together with the network architecture. Implementation details are included in Chapter 4. The numerical results are presented and discussed in Chapter 5 and 6, respectively. Lastly, Chapter 7 provides the conclusion of this thesis.

Chapter 2

Background Theory

The deep BSDE method aims to solve high-dimensional PDEs efficiently by using the technique deep learning. To do so, the PDE is first reformulated as two stochastic processes, and then an artificial neural network is used to approximate the unknown coefficients in these processes. In this chapter, the background theory for the deep BSDE method is laid down. Deep learning is explained, and the reformulation of the PDE is shown using Itô's formula. Lastly, example PDEs are presented and reformulated.

2.1 Deep Learning

In this section, an introduction to deep learning is given. The layout of artificial neural networks is presented, and the idea of learning is explained. Specifically, the multilayer feedforward perceptron model is described and presented for a simple example problem. A combination technique, deep reinforcement learning, is also described as it bears resemblance to the methodology of the deep BSDE method.

2.1.1 Introduction to Deep Learning

The field of artificial intelligence (AI) attempts to create intelligent systems that think and act humanly and rationally (Russell and Norvig, 2016, chap. 1.1). An important aspect is that the system learns and improves from data. AI systems can be used to solve problems that are challenging for humans, but straight-forward for computers. These are problems that are relatively simple to describe mathematically, and because of large amounts of data, or complex or many calculations are difficult for humans to solve. Another type of problems are the ones that are easy to solve for humans, but difficult to describe formally. It can be intuitive problems like speech recognition or image analysis (Goodfellow et al., 2016, chap. 1).

Machine learning is an example of an approach to AI. In machine learning the goal is to find statistical patterns in data, and possibly use it to make rational future decisions. An example of this could be in medical diagnosis, where clinical parameters of patients

can be used to predict the progression of diseases. Machine learning that is not limited to map input to output, but also learn the representations of the data, is called representation learning. The representations are made up by functions and vectors. Complex and abstract patterns in raw data can be extremely challenging to find. Deep learning solves this problem by expressing the representations in terms of several simpler representations. These simpler representations can be seen as layers, and building them on top of each other results in a deep graph. It is because of this, the approach has been given the name deep learning. It is common to describe the functions in deep learning as hierarchical because of the layers. Adding more layers allows representation of increasingly complex functions. **Fig. 2.1** puts deep learning into perspective in the AI field. Deep learning as an approach to AI is described in further detail in chapter 1 of (Goodfellow et al., 2016).

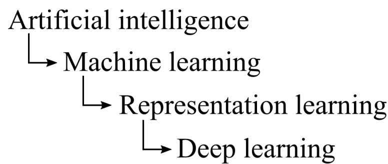


Figure 2.1: Deep learning as an approach to AI; deep learning is a type of representation learning, which is a type of machine learning, which is an approach to AI.

2.1.2 Artificial Neural Networks

Artificial neural networks (ANNs) are inspired by biological neural networks, and aim to approximate a function $f^*(x)$. The network is provided with a set of input examples x and their corresponding output values $y = f^*(x)$ that are called target values. The function $f^*(x)$ is therefore not known, but the target values for a finite set of examples are known. The network is equipped with parameters or "weights" θ and proceeds to define a mapping $f(x;\theta)$ based on the examples, where the network learns the value of the parameters θ that lead to the most accurate approximation of the function f^*, mapping the input examples to their target values. This iterative process of adjusting the parameters θ is called training, and the provided data of examples is therefore called training data. A sufficient amount of training data is needed to create a good representation of the function. After a network has been trained on data, the defined parameters θ are used to compute output from new input. These outputs can be seen as predicted values based on the information provided in the training data. Before describing exactly how the training is performed, some terms and notations will be introduced first.

ANNs consist of layers, where each layer consists of units. The first layer in the network is the input layer for which all variables in the input is given a unit. The final layer is the output layer that provides the result, being the approximated function values. The layers connecting the input and output are called hidden layers. This is because the output for each of the layers is not known. The layers can be interpreted as functions that together form the final approximation. For example three hidden layers of functions f_1, f_2, f_3 connected in a chain, would form the approximation $f(x;\theta) = f_3(f_2(f_1(x;\theta_1);\theta_2);\theta_3)$. The structure of a simple ANN, consisting of one hidden layer, is shown in **Fig. 2.2**. The term deep learning is usually used for ANNs with more than one hidden layer.

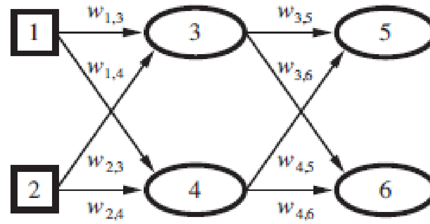


Figure 2.2: An illustration of a simple ANN from chapter 18.7 in (Russell and Norvig, 2016). 1 and 2 are units in the input layer, 3 and 4 are units in the hidden layer, and lastly 5 and 6 are units in the output layer. The weight for the link between unit i and j is denoted by $w_{i,j}$.

Each unit in the hidden layers is made to act like a neuron. Neurons are the core components of the nervous system in the human body. A neuron is an electrically excitable cell that communicates with other cells through connections called synapses. A simple mathematical model of a neuron is shown in **Fig. 2.3**.

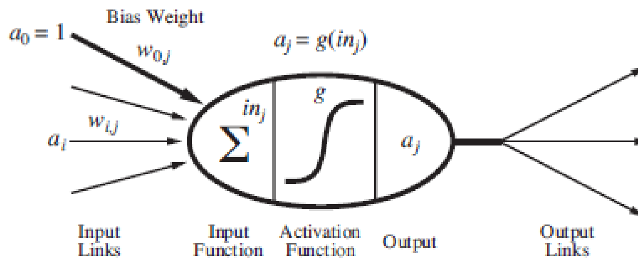


Figure 2.3: A simple mathematical model for a neuron from chapter 18.7 in (Russell and Norvig, 2016).

It is from this idea the name artificial neural network arises. The term a_i is the activation from a unit i, and $w_{i,j}$ represents the weighted link between units i and j. The input function computes a weighted sum of the inputs $\text{in}_j = \sum_{i=0}^{n} w_{i,j} a_i$. The activation function $g(\text{in}_j)$ is applied to derive the output a_j. Each unit also has a dummy input $a_0 = 1$ with associated weight $w_{0,j}$. Typical choices for the activation function are the rectified linear unit (ReLU) $g(z) = \max\{z, 0\}$, the sigmoid function $g(z) = (1 + \exp(-z))^{-1}$, the tanh function $g(z) = \tanh(z)$, and the softplus function $g(z) = \ln(1 + \exp(z))$. Given the input vector (x_1, x_2), the output of the single hidden layer network in figure 2.2 at unit 5 is given by

$$\begin{aligned} a_5 &= g(w_{0,5} + w_{3,5}a_3 + w_{4,5}a_4) \\ &= g(w_{0,5} + w_{3,5}g(w_{0,3} + w_{1,3}a_1 + w_{2,3}a_2) + w_{4,5}g(w_{0,4} + w_{1,4}a_1 + w_{2,4}a_2)) \\ &= g(w_{0,5} + w_{3,5}g(w_{0,3} + w_{1,3}g(w_{0,1}) + w_{2,3}g(w_{0,2})) \\ &\qquad + w_{4,5}g(w_{0,4} + w_{1,4}g(w_{0,1}) + w_{2,4}g(w_{0,2}))). \end{aligned}$$

Two simple examples of problems that can be solved using ANNs are the logical connectives AND and inclusive OR. In these examples the learning of the networks is not included, only the resulting weights that solve the problems. Suppose you have two input units, u_1 and u_2, that take on values in $\{0, 1\}$, one output unit, u_3, and a threshold of 1 as activation function, $g(z) = 1$ if $z \geq 1$ otherwise $g(z) = 0$. **Fig. 2.4** shows this example network.

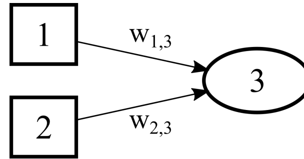


Figure 2.4: An illustration of a simple ANN with two input units and one output unit. The weight for the link between unit i and j is denoted by $w_{i,j}$. With appropriate weights, this neural network can model the logical connectives AND and inclusive OR.

With the weights $w_{1,3} = 0.6$ and $w_{2,3} = 0.6$, the input function only exceeds 1 if both inputs are 1, which satisfies the logical connective AND:

$$\begin{aligned} a_3 &= g(1 \cdot 0.6 + 1 \cdot 0.6) = 1, \\ a_3 &= g(1 \cdot 0.6 + 0 \cdot 0.6) = 0, \\ a_3 &= g(0 \cdot 0.6 + 0 \cdot 0.6) = 0. \end{aligned}$$

However, when $w_{1,3} = 1.1$ and $w_{2,3} = 1.1$, it is sufficient that one of the inputs is 1 for the input function to exceed 1, and therefore this creates an ANN for inclusive OR:

$$\begin{aligned} a_3 &= g(1 \cdot 1.1 + 1 \cdot 1.1) = 1, \\ a_3 &= g(1 \cdot 1.1 + 0 \cdot 1.1) = 1, \\ a_3 &= g(0 \cdot 1.1 + 0 \cdot 1.1) = 0. \end{aligned}$$

The ANN learns by adjusting the weights in the network in a way that the approximation of the function iteratively becomes more accurate. This is done using a gradient descent based algorithm. After each iteration, the prediction error is computed using a loss function, and a gradient is estimated to update the weights. In machine learning a loss function expresses the utility. The loss function, $L(f(x;\theta), f^*(x))$, is defined as the amount of utility that is lost by using the predicted value from the model, $f(x;\theta)$, instead of the correct one, $f^*(x)$, (Russell and Norvig, 2016, chap. 18.4.2). In other words, it expresses how inaccurate the model is. A loss function could for example be $L(f(x;\theta), f^*(x)) = |f(x;\theta) - f^*(x)|$, expressing the difference between the approximated values and the target values. The optimization algorithms aim to minimize this loss function. To do so, a gradient estimate is computed by taking the gradient of the loss function with respect to the parameters θ. This estimate indicates the direction in which the loss function increases faster. It is therefore used to update the parameters θ in a way that promises a decrease of the loss function by moving in the opposite direction. The gradient

estimate over a set of m examples, $\{x_1, x_2, \dots, x_m\}$, is given by

$$\hat{g} = \frac{1}{m} \nabla_\theta \sum_{i=1}^{m} L(f(x_i;\theta), f^*(x_i)),$$

(Goodfellow et al., 2016, chap. 8.1.3). Since all units in the hidden layers have contributed to the final error shown in the output layer, the error gradient is propagated backwards through all the units from the output layer to the input layer. In high-dimensional problems, a stochastic gradient descent-type (SGD) algorithm is typically used to compute the error gradient and backpropagation. An SGD method is usually preferred over normal gradient descent since it requires less computation. SGD algorithms will be further described in Section 4.2, and specifically the Adam optimizer will be presented and described in detail.

2.1.3 Multilayer Feedforward Perceptron Model

The multilayer feedforward perceptron (MLP) model is one of the more commonly used deep learning models (Goodfellow et al., 2016, chap. 6). The model is an ANN that consists of several layers. The term feedforward is used to describe that the network has connections in only one direction, and the model therefore graphically forms a directed acyclic graph that describes how the layers are composed together. The ANN is fully connected, meaning that all units are connected to all the units in the next layer. In **Fig. 2.5** the sine function is approximated using an MLP model. The python code to generate this example can be found in Appendix A.

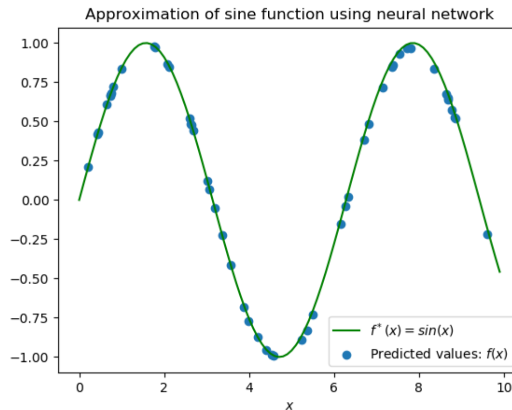


Figure 2.5: A plot of the approximation of the sine function using an MLP model with two hidden layers, each with 100 units. The green line is the function to be approximated $f^*(x) = \sin(x)$, and the blue circles mark the values predicted by the deep neural network for the respective values of x.

The pseudo code for creating the MLP model that approximates the sine function in **Fig. 2.5** is given in **Algorithm 1**. The model is implemented in the machine learning platform TensorFlow. The set $\{x_1, x_2, \dots, x_{100}\}$ denotes the set of input examples of the

model, and the set $\{y_1, y_2, \ldots, y_{100}\}$ denotes the set of target values, where $y_i = \sin(x_i)$ for $i = 1, 2, \ldots, 100$. Together they form the training data. First, the deep ANN graph is created. The input layer has one unit, taking on the value of x. Two hidden layers, h_1 and h_2, are then defined, each with 100 units. The output layer also has one unit, returning the resulting approximation $f(x; \theta)$. The layers are created with the function `tensorflow.contrib.layers.fully_connected`, that adds a fully connected layer where the input, number of units and activation function is specified. In this example the ReLU is used. Next, the loss function $L(f(x; \theta), y)$ is set to be $\frac{\sum(f(x;\theta)-y)^2}{2}$. Then, an optimization algorithm is chosen. The goal is to minimize the loss function on the training data. This is done by updating the parameters θ using the optimization algorithm. This process is called training the MLP model. To understand how the parameters θ are updated using the Adam optimizer, see Section 4.2. In the implementation, the training is simply performed for 10000 iterations with the function `tensorflow.Session().run` where the training data and optimization algorithm is specified. After the model is trained, a new set of test data is generated, $xtest$, and the function `tensorflow.Session().run` is used again to obtain the model's predicted target values.

Algorithm 1: An MLP model for approximating the sine function

Data: Training data: $\{x_1, x_2, \ldots, x_{100}\}, \{y_1, y_2, \ldots, y_{100}\}$;
Result: Trained MLP model with optimized parameters θ based on the training data

1 x = input layer (1 unit);
2 h_1 = hidden layer (fully connected from x, 100 units, activation function = ReLU);
3 h_2 = hidden layer (fully connected from h_1, 100 units, activation function = ReLU);
4 result = output layer (fully connected from h_2, 1 unit);
5 loss $= \frac{\sum(\text{result}-y)^2}{2}$;
6 optimizer $=$ Adam optimizer;
7 Initialize model;
8 **for** 0 **to** 10000 **do**
9 | $xtrain$ = 100 random numbers from 0 to 10 ;
10 | $ytrain$ = $\sin(xtrain)$;
11 | Train MLP model;
12 **end**
13 $xtest$ = 50 random numbers from 0 to 10;
14 Test MLP model on $xtest$;

2.1.4 Deep Reinforcement Learning

Machine learning is typically split into three categories; supervised learning, unsupervised learning and reinforcement learning (RL). In supervised learning the data consists of labeled examples that can be used to label new data, and thereby predict future events. Contrary to this method, unsupervised learning uses unlabeled data and instead tries to define a function that describes an initially unknown structure in the data. RL aims to sequentially

produce actions that maximizes the total reward, based on trial and error and feedback on own actions and experiences. To better understand the mindset in RL, essential terms are here explained:

Agent Agent is a term used to explain the active decision making. An agent executes actions, and receives observations and rewards.

State (S) A state explains the situation the agent is in for the specific space and time location. This includes a summary of the previous observations, rewards and actions.

Action (A) An agent takes actions based on a set of possible actions, which represent all the possible moves the agent can make.

Reward (R) A reward is a measurement of how successful the action of the agent was. After the agent takes a specific action, the new state gives a corresponding reward.

Environment The environment is the entire space where the agent moves. It takes the agent's current state and action as input, and returns the reward and next state as output.

Policy (π) The policy is the agent's strategy on which action to take next based on the current state.

Value function ($Q(S, A)$) The value function expresses the expected accumulated reward from a state S and action A.

The interaction between agent and environment in a Markov process can be seen in **Fig. 2.6**. In a Markov process the next step only depends on the current state.

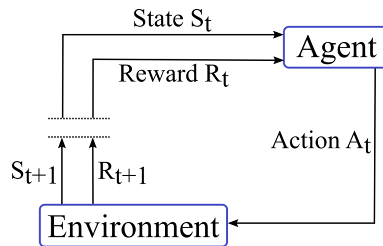


Figure 2.6: The agent-environment interaction in RL for a Markov process.

While ANNs aim to learn a function representation from training data, RL cover how to train an agent in the absence of training data (Russell and Norvig, 2016, chap. 21.1). The basic idea in RL is that the agent, after taking action, receives feedback in the form of a reward, or *reinforcement*, indicating how successful the action was. More actions and observations lead to increased knowledge of the environment and the reward function. The system may even have no prior knowledge of this. It is like playing a new game without knowing any of the rules, and then suddenly being told by the opponent that you lost. How frequently the reward is received can vary. For example, when playing chess the reward is only received at the end of the game. However, with the goal of reaching a destination,

every step reducing the distance to the destination can be considered a positive reward. In practice, the agent takes action based on a policy. In RL, the agent seeks to find the optimal policy that maximizes the expected total reward. The optimal policy is learned through observed rewards. If again the goal is to reach a destination, the agent would after trying various steps learn that steps in the direction of the target result in better rewards. With this new knowledge the agent's policy would be updated to direct the agent to the desired destination.

The value function $Q(S, A)$ expresses the expected accumulated reward from a state S and action A. To denote expected value, the symbol $\mathbb{E}$ is used. Conditional expectation expresses the expected value of a term given the value of some other term. For instance, the conditional expectation $\mathbb{E}[X|Y]$ denotes the expected value of X given Y. The value function under the policy π is then expressed as

$$Q^{\pi}(S, A) = \mathbb{E}[R + \gamma R_{t+1} + \gamma^2 R_{t+2} + \ldots |S, A],$$

where γ denotes a discount factor (Mnih et al., 2015) that states the importance of future rewards R_{t+i} to the current state. The value function sums up all expected future rewards with importance given that the agent is in state S and executes action A. Using S' and A' to denote the state arrived at after action A, and the action picked at state S', respectively, the value function can be written as

$$Q^{\pi}(S, A) = \mathbb{E}_{S',A'}[R + \gamma Q^{\pi}(S', A')|S, A].$$

The subscript in the expectation describes under which distribution the expectation is being taken. RL models aim to find the optimal policy that maximizes the expected total reward. The optimal value function, being the highest value function available, is achieved by the action A' that maximizes the value function at $Q^*(S', A')$. So the optimal policy (the goal of the model) is achieved by the action A that maximizes the value function $Q^*(S, A)$:

$$\begin{aligned} Q^*(S, A) &= \mathbb{E}_{S'}\left[R + \gamma \max_{A'} Q^*(S', A')|S, A\right], \\ \pi^*(S) &= \arg\max_{A} Q^*(S, A). \end{aligned}$$

In traditional RL, the value function is updated iteratively to obtain the optimal one as follows,

$$Q(S, A) \leftarrow Q(S, A) + \alpha \cdot \big(\underbrace{R + \gamma \max_{A'} Q(S', A')}_{\text{Target}} - \underbrace{Q(S, A)}_{\text{Prediction}}\big),$$

where α denotes a learning rate. Since the last terms subtract the prediction from the target, this represents the loss.

There exists a method for exploiting the benefits from both deep learning and RL called deep RL (François-Lavet et al., 2018). This method unites function approximation and target optimization. While traditional RL is limited to low-dimensional input, deep ANNs are efficient at extracting abstract representation from high-dimensional input. In deep RL, an ANN can therefore be used to approximate the value function (or policy),

$$Q(S, A; \theta) \approx Q^*(S, A).$$

The loss of the network at iteration i, using mean squared error, is given by

$$L_i(\theta_i) = \mathbb{E}_{S,A,R,S'}\left[\left(R + \gamma \max_{A'} Q(S', A'; \theta_i^*) - Q(S, A; \theta_i)\right)^2\right],$$

where θ_i denotes the network parameters at iteration i, and θ_i^* denotes the network parameters used to compute the target at iteration i. The learning in deep RL is still performed by iteratively adjusting the weights θ along gradients that promise less error.

2.1.5 The Expressive Power of Deep Neural Networks

The effectiveness of multilayer ANNs is somewhat unclear, and there still lacks a complete theoretical framework for explaining it. There has, however, been done extensive research on the expressive power of ANNs with regard to both the width (number of units in the hidden layers) and the depth (number of hidden layers) of the network through approximation theory. For instance, (Eldan and Shamir, 2016) with references therein state that sufficiently large ANNs with 2 hidden layers can approximate any continuous function on a bounded domain when using appropriate activation functions. Although, the required width of the network may be exponential in the dimension. Such networks are impractical and also prone to overfitting, as they discuss in the introduction of the paper. They then proceed to show that there exists a function on $\mathbb{R}^d$ that can be expressed by a relatively small MLP with 3 hidden layers, that cannot be expressed by any bounded network consisting of 2 hidden layers.

It is important that the deep ANN is sufficiently large, so that it is able to represent the best approximation function that is available. As discussed, this is ensured with 2 hidden layers, but because it may require exponential width, it raises the question whether a lower bound with respect to the width for universal approximators can be found. Lu et al. (2017) actually formulate a theorem stating that MLPs with bounded width and ReLU as activation function are universal approximators. Specifically, any Lebesgue-integrable function $f^* : \mathbb{R}^d \to \mathbb{R}$ can be expressed by this kind of network with a width smaller than $4 + d$, where d is the dimension of the function to be approximated. Here, representation of the best available approximation function is ensured with a practical network without risking overfitting as a consequence of exponential width. Lu et al. (2017) proved that a ReLU network with width less than or equal to d cannot approximate all functions $f^* : \mathbb{R}^d \to \mathbb{R}$. The universal approximation theorem for width-bounded ReLU networks is rendered here.

Theorem 1 (Universal Approximation Theorem for Width-Bounded ReLU Networks)**.** *For any Lebesgue-integrable function $f^* : \mathbb{R}^d \to \mathbb{R}$ and any $\epsilon > 0$, there exists a fully-connected ReLU network with width $d_m \leq d + 4$, such that the function f represented by this network satisfies*

$$\int_{\mathbb{R}^d} |f^*(x) - f(x; \theta)| dx < \epsilon.$$

It is clear that a network's expressive power increases with both increasing width and depth. It is not given whether a shallow and wide or a narrow and deep network will be

the best architectural option, and optimal choices will depend on the specific model and problem characteristics.

2.2 Viewing Partial Differential Equations as Stochastic Control Problems

The reformulation of PDEs into stochastic processes is inspired by viewing the PDEs as stochastic control problems. This section briefly explains stochastic control problems and connects them to PDEs. The reformulation from a PDE to a BSDE is given as a general formula with the nonlinear Feynman-Kac formula, and shown using Itô's lemma.

2.2.1 Stochastic Control Problems

Deep learning has proven to work great with high-dimensional problems such as speech recognition and image analysis, because of its efficient learning and expressive power. This motivates applying deep learning to other problems that suffer the curse of dimensionality. In (Han et al., 2016), they study the case of solving high-dimensional stochastic control problems using deep neural networks. The results are promising in terms of overcoming the curse of dimensionality.

Control theory covers issues regarding controllability, "one may find at least one way to achieve a goal", and optimality, "one hopes to find the best way, in some sense, to achieve the goal" (Lu and Zhang, 2016). Control theory for stochastic systems is typically described by stochastic differential equations (SDEs). SDEs are simply differential equations that contain randomness, or "noise". As a consequence, the solution also contains randomness, and is given as a probability distribution. Stochastic control problems are therefore given by SDEs along with an objective function. The objective is the desired goal, and may for example be a minimization. In (Han et al., 2016) they use a deep feed-forward neural network to solve the stochastic control problem by having the objective function play the role of the loss function in the deep neural network. Their obtained solutions are satisfactory in terms of both accuracy and extendability to high-dimensional case.

Semilinear parabolic PDEs can be written equivalently as BSDEs, and there are significant similarities between stochastic control problems and BSDEs. This motivates using deep learning to solve these PDEs, which is what Han et al. (2018) proceeded to explore in (Han et al., 2018).

2.2.2 Backward Stochastic Differential Equation Reformulation

Semilinear parabolic PDEs can be written equivalently as BSDEs, and can therefore be reformulated into a stochastic control problem. The connection between parabolic PDEs and stochastic processes offers a method of solving the PDEs by simulating random paths of a stochastic process. Reversely, the expectations of some stochastic processes can be computed by deterministic methods. To motivate the connection between BSDEs and PDEs, a proof of the (linear) Feynman-Kac formula will be shown.

The Feynman-Kac formula states that the solution of certain PDEs can be represented as a probabilistic expectation value with respect to some Itô diffusion process. The probability space $(\Omega, \mathcal{F}, \mathbb{P})$ on a finite time interval $t \in [0, T]$ is considered, where $\{\mathcal{F}_t\}_{t\in[0,T]}$ denotes the normal filtration generated by the Brownian motion $\{W_t\}_{t\in[0,T]}$. A d-dimensional Itô diffusion X_t is a solution to the SDE

$$dX_t = \mu(t, X_t)dt + \sigma(t, X_t)dW_t, \tag{2.1}$$

where $\mu \in \mathbb{R}^d$, $\sigma \in \mathbb{R}^{d\times m}$, and W_t denotes the m-dimensional Brownian motion, where $W : [0, T] \times \Omega \to \mathbb{R}^m$ is described by the m-dimensional Wiener process whose components $W_t^{(i)}$ are independent, standard one-dimensional Wiener processes with the following properties:

1. $W_0 = 0$,
2. $t \to W_t$ is continuous in t, with probability 1,
3. the process has stationary, independent increments, and
4. the increments $W_{t+s} - W_s$ are normally distributed with mean 0 and variance t, $W_{t+s} - W_s \sim \mathcal{N}(0, t)$.

The probability density function of the normal distribution $\mathcal{N}(\mu, \sigma^2)$ is

$$f(x|\mu, \sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}}\exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right),$$

where μ is the mean value and σ^2 is the variance. This is not to be confused with the earlier presented μ and σ representing terms in an SDE. The mean value and variance will not be further discussed, only formally presented to give a definition of the normal distribution. If a function $f : [0, T] \times \mathbb{R}^d \to \mathbb{R}$ is twice continuously differentiable, $f(t, x) \in C^2$, then by Itô's formula, $f(t, X_t)$, where X_t is given by equation (2.1), is also an Itô process with differential

$$df(t, X_t) = \frac{\partial f}{\partial t}dt + (\nabla f)^{\mathrm{T}} dX_t + \frac{1}{2}(dX_t)^{\mathrm{T}}(\mathrm{Hess}_x f)dX_t. \tag{2.2}$$

In this expression, ∇ denotes the gradient of a function with respect to x, and Hess_x denotes the Hessian of a function with respect to x. Consider the d-dimensional Itô diffusion $\{X_t\}_{t\in[0,T]}$ given by equation (2.1), and let the generator A be defined as

$$(Af)(t, x) = \sum_{i=0}^{d} \mu_i(t, x)\frac{\partial f}{\partial x_i} + \frac{1}{2}\sum_{i,j=1}^{d}(\sigma\sigma^{\mathrm{T}})_{i,j}(t, x)\frac{\partial^2 f}{\partial x_i \partial x_j}. \tag{2.3}$$

Itô's formula is applied to $e^{-\int_0^t r(s,X_s)ds}u(t, X_t)$, where $u \in C^{1,2}(\mathbb{R}^+ \times \mathbb{R}^d)$ and $r \in C^0(\mathbb{R}^+ \times \mathbb{R}^d)$,

$$\begin{aligned} d(e^{-\int_0^t r(s,X_s)ds}u(t, X_t)) &= e^{-\int_0^t r(s,X_s)ds}du(t, X_t) + u(t, X_t)d(e^{-\int_0^t r(s,X_s)ds}) \\ &= e^{-\int_0^t r(s,X_s)ds}\left(\frac{\partial u}{\partial t} + Au - ru\right)(t, X_t)dt. \end{aligned}$$

Then, both sides are integrated,

$$e^{-\int_0^t r(s,X_s)ds}u(t,X_t) = u(t,x) + \int_0^t e^{-\int_0^s r(v,X_v)dv}\left(\frac{\partial u}{\partial s} + Au - ru\right)(s,X_s)ds.$$

By rearranging the equation, the following expression for u is obtained,

$$u(t,x) = e^{-\int_0^t r(s,X_s)ds}u(t,X_t) - \int_0^t e^{-\int_0^s r(v,X_v)dv}\left(\frac{\partial u}{\partial s} + Au - ru\right)(s,X_s)ds.$$

This stochastic process is a martingale see (Lehmann, n.d.), and will be denoted $u(t,x) = M_t$:

$$M_t = e^{-\int_0^t r(s,X_s)ds}u(t,X_t) - \int_0^t e^{-\int_0^s r(v,X_v)dv}\left(\frac{\partial u}{\partial s} + Au - ru\right)(s,X_s)ds, \quad (2.4)$$

meaning that the conditional expectation of the next value of the process, given all prior values, is equal to the present value. The term was first introduced by Ville in 1939, and this definition is taken from (Grimmett et al., 2001): *A sequence* $Y = \{Y_n : n \geq 0\}$ *is a martingale with respect to the sequence* $X = \{X_n : n \geq 0\}$ *if, for all* $n \geq 0$, $\mathbb{E}|Y_n| < \infty$, *and* $\mathbb{E}[Y_{n+1}|X_0, X_1, \ldots, X_n] = Y_n$.

Again, consider the Itô diffusion $\{X_t\}_{t\in[0,T]}$ given by equation (2.1) with initial condition $X_t^{t,x} = x$, and generator A given by equation (2.3). Let $u(t,x) \in C^{1,2}([0,T]\times\mathbb{R}^d)$ satisfy the following PDE

$$\begin{aligned}\frac{\partial u}{\partial t} + Au - ru = 0,\\ u(T,x) = h(x),\end{aligned} \quad (2.5)$$

then by the Feynman-Kac formula, the solution can be probabilistically represented as

$$\begin{aligned}u(t,x) &= \mathbb{E}\left[e^{-\int_t^T r(s,X_S)ds}h(X_T)|X_t = x\right]\\ &= \mathbb{E}\left[e^{-\int_t^T r(s,X_S^{t,x})ds}h(X_T^{t,x})\right],\end{aligned} \quad (2.6)$$

for all $(t,x) \in [0,T]\times\mathbb{R}^d$. To prove the linear Feynman-Kac formula (2.6), first, it is noted that since (2.4) is a martingale, the process $\{M_{t'}\}_{t\leq t'\leq T}$ given by

$$\begin{aligned}M_{t'} &= e^{-\int_t^{t'} r(s,X_s^{t,x})ds}u(t',X_{t'}^{t,x}) - \int_t^{t'} e^{-\int_t^s r(v,X_v^{t,x})dv}\underbrace{\left(\frac{\partial u}{\partial s} + Au - ru\right)}_{=0 \text{ by eq. (2.5)}}(s,X_s^{t,x})ds\\ &= e^{-\int_t^{t'} r(s,X_s^{t,x})ds}u(t',X_{t'}^{t,x}),\end{aligned}$$

is also a martingale. The solution of the PDE (2.5) can therefore be written as

$$\begin{aligned} u(t,x) &= M_t \\ &= \mathbb{E}[M_t] \\ &= \mathbb{E}[M_T] \\ &= \mathbb{E}\left[e^{-\int_t^T r(s,X_s^{t,x})ds} u(T, X_T^{t,x})\right] \\ &= \mathbb{E}\left[e^{-\int_t^T r(s,X_s^{t,x})ds} h(X_T^{t,x})\right], \end{aligned}$$

proving the Feyman-Kac formula $u(t,x) = \mathbb{E}\left[e^{-\int_t^T r(s,X_s^{t,x})ds} h(X_T^{t,x})\right]$.

The deep BSDE method considers the general class of PDEs called semilinear parabolic PDEs, and these are connected to BSDEs through the nonlinear Feynman-Kac formula. The proof of the nonlinear Feynman-Kac formula is more complicated and will not be shown in this thesis, however, it can be found in (Pardoux and Peng, 1992). Semilinear parabolic PDEs of the following form is considered,

$$\begin{aligned} \frac{\partial u}{\partial t}(t,x) + \frac{1}{2}\mathrm{Tr}\left[\sigma\sigma^{\mathrm{T}}(t,x)(\mathrm{Hess}_x u(t,x))\right] + \nabla u(t,x)\cdot\mu(t,x) \\ + f\left(t,x,u(t,x),\sigma^{\mathrm{T}}(t,x)\nabla u(t,x)\right) = 0, \end{aligned} \tag{2.7}$$

with the terminal condition $u(T,x) = h(x)$, where $T > 0$ and h is continuous. The function u is assumed to be once continuously differentiable with respect to the time variable $t \in [0,T]$ and twice continuously differentiable with respect to the space variable $x \in \mathbb{R}^d$, i.e. $u(t,x) \in C^{1,2}([0,T] \times \mathbb{R}^d)$. The variable σ denotes a $d \times d$ matrix-valued function, μ is a vector valued function and f is a continuous nonlinear function. The symbol Tr denotes the trace of a matrix, Hess_x denotes the Hessian of a function with respect to x, and ∇ denotes the gradient of a function with respect to x. Again, the probability space $(\Omega, \mathcal{F}, \mathbb{P})$ on a finite time interval $t \in [0,T]$ is considered. With $x \in \mathbb{R}^d$, $h(x) \in C^3(\mathbb{R}^d)$ and $f(s,x,y,z) \in C^3([0,T] \times \mathbb{R}^d \times \mathbb{R}^k \times \mathbb{R}^{k\times d})$, the following BSDE

$$Y_s = h(X_T) + \int_s^T f(X_r, Y_r, Z_r)dr - \int_s^T (Z_r)^{\mathrm{T}} dW_r, \quad t \le s \le T, \tag{2.8}$$

has the unique solution process given by $\{(Y_s, Z_s); t \le s \le T\}_{t\ge 0, x\in\mathbb{R}^d}$ (Pardoux and Peng, 1992). The following is a rendered theorem from (Pardoux and Peng, 1992), that gives the connection between BSDEs and semilinear parabolic PDEs:

Theorem 2. *If $u \in C^{1,2}([0,T] \times \mathbb{R}^d)$ solves equation* (2.7)*, then $u(t,x) = Y_t$, $t \ge 0$, $x \in \mathbb{R}^d$, where $\{(Y_s, Z_s); t \le s \le T\}_{t\ge 0, x\in\mathbb{R}^d}$ is the unique solution of the BSDE* (2.8).

In the deep BSDE method, the aim is to approximate the process $\{(Z_s); t \le s \le T\}_{t\ge 0, x\in\mathbb{R}^d}$ through deep learning, and then use a form of equation (2.8) to compute the process $\{(Y_s); t \le s \le T\}_{t\ge 0, x\in\mathbb{R}^d}$, thereby finding the solution of a semilinear parabolic PDE. To make the connection between BSDEs and semilinear parabolic PDEs clear, Itô's formula is applied to $u(s, X_s)$ between $s = t$ and $s = T$, and then it is noted that $\{Y_s, Z_s\} =$

$\{u(s, X_s), (\nabla u\sigma)(s, X_s)\}$ solves the BSDE. All the calculations are now shown. The following stochastic process $\{X_t\}_{t\in[0,T]}$ is considered,

$$X_t = \xi + \int_0^t \mu(s, X_s)ds + \int_0^t \sigma(s, X_s)dW_s, \tag{2.9}$$

where μ, $\sigma \in C^3([0, T]\times\mathbb{R}^d)$. In this process, the solution of the semilinear parabolic PDE is examined at the point $x = \xi$, for some $\xi \in \mathbb{R}^d$. By writing this process in differential form, the equation (2.1) is obtained. Now, Itô's formula, given by equation (2.2), is applied to the solution of the semilinear parabolic PDE, $u(s, X_s)$, that by assumption is once continuously differentiable with respect to t and twice continuously differentiable with respect to x,

$$du(s, X_s) = \frac{\partial u}{\partial s}ds + (\nabla u)^{\mathrm{T}}dX_s + \frac{1}{2}(dX_s)^{\mathrm{T}}(\mathrm{Hess}_x u)dX_s. \tag{2.10}$$

In stochastic calculus, with s denoting a time variable and W_s denoting a Brownian motion, the following rules apply

$$(dW_s)^2 = ds, \tag{2.11}$$

$$dW_s ds = 0, \tag{2.12}$$

$$(ds)^2 = 0. \tag{2.13}$$

Inserting equation (2.9) into equation (2.10) and using the substitutions from the rules (2.11)-(2.13) give

$$du(s, X_s) = \left\{\frac{\partial u}{\partial s} + (\nabla u)^{\mathrm{T}}\mu + \frac{1}{2}\mathrm{Tr}\left[\sigma^{\mathrm{T}}(\mathrm{Hess}_x u)\sigma\right]\right\}ds + (\nabla u)^{\mathrm{T}}\sigma dW_s.$$

The solution $u(s, X_s)$ is now viewed between $s = t$ and $s = T$,

$$\begin{aligned} u(T, X_T) - u(s, X_s) = &\int_s^T \underbrace{\frac{\partial u}{\partial r} + (\nabla u)^{\mathrm{T}}\mu + \frac{1}{2}\mathrm{Tr}\left[\sigma^{\mathrm{T}}(\mathrm{Hess}_x u)\sigma\right]}_{=-f \text{ by eq. (2.7)}} dr \\ &+ \int_s^T (\nabla u)^{\mathrm{T}}\sigma dW_r, \qquad t \le s \le T. \end{aligned} \tag{2.14}$$

The term $u(T, X_T)$ is the terminal condition $h(X_T)$. Since $u(s, X_s)$ solves equation (2.7), the first integral can be rewritten using this equation, and the whole BSDE (2.14) can therefore be written as

$$u(s, X_s) = h(X_T) + \int_s^T f dr - \int_s^T (\nabla u)^{\mathrm{T}}\sigma dW_r, \quad t \le s \le T. \tag{2.15}$$

By comparing this equation to the general form of BSDEs in equation (2.8), it is evident that the solution process is given by $\{Y_s, Z_s\} = \{u(s, X_s), \sigma^{\mathrm{T}}(s, X_s)\nabla u(s, X_s)\}$ $\mathbb{P}$-a.s. So it is shown that if $u \in C^{1,2}([0, T] \times \mathbb{R}^d)$ solves equation (2.7), then $u(t, x) = Y_t$, as

stated in **Thm. 2**. The identity $\{Y_t, Z_t\} = \{u(t, X_t), \sigma^{\mathrm{T}}(t, X_t)\nabla u(t, X_t)\}$ is known as the nonlinear Feynman-Kac formula. The reverse of **Thm. 2** also holds, and this proof can be found in (Pardoux and Peng, 1992).

A semilinear parabolic PDE can now be written as a stochastic control problem. Looking at the solution at the point $x = \xi$, for some $\xi \in \mathbb{R}^d$, we now have the following BSDEs to represent a semilinear parabolic PDE,

$$X_t = \xi + \int_0^t \mu(s, X_s)ds + \int_0^t \sigma(s, X_s)dW_s, \tag{2.16}$$

$$Y_t = h(X_T) + \int_t^T f(s, X_s, Y_s, Z_s)ds - \int_t^T (Z_s)^{\mathrm{T}} dW_s. \tag{2.17}$$

The goal is then to find the $\{\mathcal{F}_t\}_{t\in[0,T]}$-adapted solution process $\{X_t, Y_t, Z_t\}_{t\in[0,T]}$. Inserting the nonlinear Feynman-Kac formula into equation (2.17) and writing the equation forwardly give

$$\begin{aligned} u(t, X_t) - u(0, X_0) = &- \int_0^t f(s, X_s, u(s, X_s), \sigma^{\mathrm{T}}(s, X_s)\nabla u(s, X_s))ds \\ &+ \int_0^t [\nabla u(s, X_s)]^{\mathrm{T}} \sigma(s, X_s) dW_s. \end{aligned} \tag{2.18}$$

Existence and uniqueness of solutions can be proved under suitable regularity assumptions on the functions μ, σ and f (Pardoux and Peng, 1992).

Next, a temporal discretization is applied on the time interval $[0, T] : 0 = t_0 < t_1 < \cdots < t_N = T$, using the simple Euler scheme for $n = 1, \ldots, N-1$. The forward Euler scheme consists of sampling at t_n and approximating the time derivative by forward difference, $\partial u_{t_n}/\partial t \approx (u_{t_{n+1}} - u_{t_n})/\Delta t_n$. The discretized version of equation (2.18) is then

$$\begin{aligned} &u(t_{n+1}, X_{t_{n+1}}) - u(t_n, X_{t_n}) \\ &\approx - f\left(t_n, X_{t_n}, u(t_n, X_{t_n}), \sigma^{\mathrm{T}}(t_n, X_{t_n})\nabla u(t_n, X_{t_n})\right) \Delta t_n \\ &\quad + [\nabla u(t_n, X_{t_n})]^{\mathrm{T}} \sigma(t_n, X_{t_n}) \Delta W_n, \end{aligned} \tag{2.19}$$

where $\Delta t_n = t_{n+1} - t_n$ and $\Delta W_n = W_{t_{n+1}} - W_{t_n}$, and the discretized stochastic process $\{X_t\}_{t\in[0,T]}$ is

$$X_{t_{n+1}} - X_{t_n} \approx \mu(t_n, X_{t_n})\Delta t_n + \sigma(t_n, X_{t_n})\Delta W_n. \tag{2.20}$$

In the BSDE reformulation of the semilinear parabolic PDE (equation (2.19)) the gradient of the solution, $\{\sigma^{\mathrm{T}}\nabla u(t, X_t)\}_{t\in[0,T]}$, is unknown at each discretization step, t_n for $n = 1, \ldots, N-1$. In addition, the solution at the initial time, $u(0, X_0)$, is also unknown. These are the coefficients that are to be approximated through deep learning.

The problem can be interpreted as a RL problem where the process $\{Z_t\}_{t\in[0,T]} = \{\sigma^{\mathrm{T}}\nabla u(t, X_t)\}_{t\in[0,T]}$ is the policy function, and the values of it thereby determine how well the method performs. The deep BSDE method then resembles the idea in deep RL, with the BSDE and the gradient of the solution playing the role of the Markov decision

model and the optimal policy function, respectively. In deep RL, an ANN is used to approximate the policy function. An MLP model is therefore used to approximate the unknown coefficients at each discretization step, and uses equation (2.19) and (2.20) to obtain the solution at $x = \xi$, $t = 0$. The BSDE reformulation of the general semilinear parabolic PDE can now be viewed as a stochastic control problem where the objective function is the loss function in the ANN. The methodology is described stepwise in Chapter 3.

2.3 Example Partial Differential Equations

Here, two examples of semilinear parabolic PDEs that can be solved using the deep BSDE method are presented, along with a short discussion on the derivation, application and importance of the PDEs. Using the formula derived in the previous section, the reformulations into BSDEs are given.

2.3.1 The Allen-Cahn Equation

The Allen-Cahn equation is a reaction-diffusion equation that models phase separation processes. It has applications in areas such as material science, biology, geology and image analysis (Benner and Stoll, 2013). In material science, it can for example be used to model phase separation in alloys. Since even small phase separations in alloys may may lead to significant property changes, analysis of this process can be used to optimize materials and to study the stability of the alloys (Kostorz, 1995). The Allen-Cahn equation is a type of semilinear parabolic PDE on the form

$$u_t = \Delta u - f(u). \tag{2.21}$$

It can be derived by taking the functional derivative of the Ginzburg-Landau energy functional when $\epsilon = 1$ and $f = -F'$,

$$\mathcal{E}(u) = \int_\Omega \left(\frac{\epsilon^2}{2} |\nabla u|^2 + F(u) \right) dx,$$

in a given domain $\Omega \subset \mathbb{R}^d$ as described in (Karasözen et al., 2018). For multi-component alloy systems in material science, the solution u denotes the phase state between materials, and is given by the concentration of one of the components of the alloy. The parameter ϵ defines the thickness of the interfaces separating the phases. The Allen-Cahn equation can be rescaled by ϵ to only study the case $\epsilon = 1$, as in equation (2.21). An additional mobility function could be considered in an analytic approach of the equation. The derivation of the equation and theoretical analysis is beyond the scope of this thesis, but an extensive discussion on the analytical properties can be read in chapter 6 (p. 153-182) of (Bartels, 2015). The nonlinear term $f(u)$ in equation (2.21) is given by the derivative of a free energy functional $F(u)$,

$$f = -F'.$$

A typical choice for the free energy functional F is the convex quartic double-well potential,

$$F(u) = \frac{1}{4} \left(1 - u^2\right)^2.$$

This functional is bi-stable, in the sense that $F(\pm 1) = 0$ and $F(s) > 0$ if $s \neq \pm 1$.

The d-dimensional Allen-Cahn equation with the double well potential and initial condition $u(0, x) = h(x)$ is written as

$$\frac{\partial u}{\partial t}(t, x) = \Delta u(t, x) + u(t, x) - [u(t, x)]^3,$$

where $t \in (0, T]$ is the time variable, $x = (x_1, x_2, \ldots, x_d) \in \Omega \subset \mathbb{R}^d$ is the d-dimensional space variable and $u(t, x) \in C^{1,2}([0, T] \times \mathbb{R}^d)$. By applying a transformation of the time variable $t \mapsto T - t$, a terminal condition $h(T, x)$ is obtained as well as a representation of the Allen-Cahn equation on the semilinear parabolic PDE form presented in equation (2.7),

$$\frac{\partial u}{\partial t}(t, x) + \Delta u(t, x) + u(t, x) - [u(t, x)]^3 = 0.$$

Terminal value problems can easily be transformed to initial value problems, and vice versa. The Allen-Cahn equation is obtained from the general semilinear parabolic PDE in (2.7) by setting

$$\begin{aligned} \mu(t, x) &= \mu = 0, \\ \sigma &= \sqrt{2}, \\ f(t, x, u(t, x), \sigma^{\mathrm{T}}(t, x)\nabla u(t, x)) &= f(u(t, x)) = u(t, x) - [u(t, x)]^3. \end{aligned}$$

It can be helpful to write out the following equality,

$$\mathrm{Tr}(\mathrm{Hess}_x u(t, x)) = \sum_{i=1}^{d} \frac{\partial^2 u}{\partial x_i^2}(t, x) = \Delta u(t, x).$$

Using the formulas in equations (2.19) and (2.20), the following set of discretized BSDEs is obtained for the Allen-Cahn equation,

$$X_{t_{n+1}} - X_{t_n} \approx \sqrt{2}\Delta W_n, \tag{2.22}$$

$$\begin{aligned} u(t_{n+1}, X_{t_{n+1}}) - u(t_n, X_{t_n}) \approx &- \left(u(t_n, X_{t_n}) - [u(t_n, X_{t_n})]^3\right) \Delta t_n \\ &+ \sqrt{2}[\nabla u(t_n, X_{t_n})]^{\mathrm{T}} \Delta W_n. \end{aligned} \tag{2.23}$$

The unknown exact solution of the Allen-Cahn equation can be found using the branching-diffusion method.

2.3.2 The Hamilton-Jacobi-Bellman Equation

The Hamilton-Jacobi-Bellman equation (HJB equation) is a second-order PDE, and is well known for its application in optimal control theory, where the goal is to find a control for a dynamical system such that an objective function is optimized. The equation is a result of applying dynamic programming to continuous optimal control problems. Once the HJB equation is derived, the optimal control can be found by taking the maximizer/minimizer of the (generalized) Hamiltonian involved in the HJB equation (Yong and Zhou, 1999, chap.

4). Given a filtered probability space $(\Omega, \mathcal{F}, \mathbb{P})$ and a terminal condition $u(T, x) = h(x)$, the HJB equation can be written as follows,

$$-\frac{\partial u}{\partial t}(t, x) + \sup_{v \in V'} G\left(t, x, v, -\mathrm{Hess}_x u(t, x), -\nabla u(t, x)\right) = 0,$$

where $t \in [0, T)$ is the time variable, $x = (x_1, x_2, \ldots, x_d) \in \Omega \subset \mathbb{R}^d$ is the d-dimensional space variable and $u(t, x) \in C^{1,2}([0, T] \times \mathbb{R}^d)$. The functions μ, σ, f, h are uniformly continuous. The function G is called the generalized Hamiltonian, and is

$$G(t, x, v, p, P) = \frac{1}{2}\mathrm{Tr}(P\sigma(t, x, v)\sigma(t, x, v)^{\mathrm{T}}) + p \cdot \mu(t, x, v) - f(t, x, v).$$

In the case where $\sigma\sigma^{\mathrm{T}}(t, x, v)$ is uniformly positive definite, the HJB equation admits a classical solution (Yong and Zhou, 1999, chap. 4).

(Han et al., 2018) consider a simple linear-quadratic-Gaussian control problem, meaning that the system is linear, the criterion quadratic and that disturbances are Gaussian. The state process is set to be

$$dX_t = 2\sqrt{\lambda} m_t dt + \sqrt{2} dW_t,$$

where $t \in [0, T]$, $X_0 = x$, λ is a positive constant that represents the level of control, and W_t denotes a Brownian motion. The cost functional is given by

$$J(\{m_t\}_{t \in [0,T]}) = \mathbb{E}\left[\int_0^T ||m_t||^2 dt + h(X_T)\right].$$

The control process is denoted by $\{m_t\}_{t \in [0,T]}$, and the goal of the linear-quadratic-Gaussian control problem is to minimize the cost functional through the control process. The optimal control is given by

$$m_t^* = \frac{\nabla u(t, x)}{\sqrt{2\lambda}},$$

and the HJB equation for this problem is then

$$\frac{\partial u}{\partial t}(t, x) + \Delta u(t, x) - \lambda ||\nabla u(t, x)||^2 = 0 \tag{2.24}$$

(Yong and Zhou, 1999, chap. 4). It is obtained from the general semilinear parabolic PDE in equation (2.7) by setting

$$\mu(t, x) = \mu = 0,$$
$$\sigma = \sqrt{2},$$
$$f(t, x, u(t, x), \sigma^{\mathrm{T}}(t, x)\nabla u(t, x)) = f(u(t, x)) = -\lambda ||\nabla u(t, x)||^2.$$

In the context of BSDE for control, $Y_t = u(t, X_t)$ denotes the optimal value and $Z_t = \sigma^{\mathrm{T}}(t, X_t)\nabla u(t, X_t)$ denotes the optimal control. Given that the state starts from x, the

value of the solution $u(t,x)$ at $t=0$ gives the optimal cost. The discretized set of BSDEs for the HJB equation is found using the formulas in equations (2.19) and (2.20),

$$X_{t_{n+1}} - X_{t_n} \approx \sqrt{2}\Delta W_n, \tag{2.25}$$

$$\begin{aligned} u(t_{n+1}, X_{t_{n+1}}) - u(t_n, X_{t_n}) \approx & - (-\lambda ||\nabla u(t_n, X_{t_n})||^2)\Delta t_n \\ & + \sqrt{2}[\nabla u(t_n, X_{t_n})]^{\mathrm{T}} \Delta W_n. \end{aligned} \tag{2.26}$$

The dimensionality of the HJB equation equals the state space of the control problem, and these high-dimensional problems occur in for instance game theory with multiple players, where each player must solve a high-dimensional HJB type equation to obtain their optimal strategy. The explicit solution of the HJB equation (2.24) can be derived by applying Itô's formula (see (Chassagneux et al., 2016) Subsection 4.2), and is given by

$$u(t,x) = -\frac{1}{\lambda} \ln\left(\mathbb{E}\left[\exp\left(-\lambda h(x+\sqrt{2}W_{T-t})\right)\right]\right).$$

The explicit solution will be used to test the accuracy of the deep BSDE method.

Chapter 3

Methodology

The deep BSDE method solves terminal value problems expressed by a semilinear parabolic PDE equipped with a terminal condition. As in deep RL, an MLP model is used to approximate the policy function, in this case being the gradient of the solution. It is approximated at each discretization point, and the solution is computed using the discretized BSDEs. The BSDE reformulation of the general semilinear parabolic PDE can be viewed as a stochastic control problem where the objective function is given by the loss function. In this chapter, the methodology of the deep BSDE method is described stepwise. The entire network architecture for the model is illustrated, and the different connections within it are explained.

3.1 Equation Reformulation

The deep BSDE method first reformulates a terminal value problem, given by a PDE and a terminal condition, into a stochastic control problem. A general representation of a d-dimensional semilinear parabolic PDE is given by

$$\frac{\partial u}{\partial t}(t,x) + \frac{1}{2}\mathrm{Tr}\left[\sigma\sigma^{\mathrm{T}}(t,x)(\mathrm{Hess}_x u(t,x))\right] + \nabla u(t,x)\cdot\mu(t,x) \\ + f\left(t,x,u(t,x),\sigma^{\mathrm{T}}(t,x)\nabla u(t,x)\right) = 0,$$

with $t \in [0,T]$, $x \in \mathbb{R}^d$ and terminal condition $u(T,x) = h(x)$. This PDE can be written equally as a BSDE, given a process $\{X_t\}_{t\in[0,T]}$ and looking at the solution at $x = \xi$, for some $\xi \in \mathbb{R}^d$. The BSDEs are obtained by applying Itô's formula (see Section 2.2 for details),

$$\begin{aligned} X_t &= \xi + \int_0^t \mu(s,X_s)ds + \int_0^t \sigma(s,X_s)dW_s, \\ Y_t &= h(X_T) + \int_t^T f(s,X_s,Y_s,Z_s)ds - \int_t^T (Z_s)^{\mathrm{T}}dW_s. \end{aligned} \tag{3.1}$$

The solution process of the BSDE (3.1) is given by the nonlinear Feynman-Kac formula, $\{Y_t, Z_t\} = \{u(t, X_t), \sigma^{\mathrm{T}}(t, X_t)\nabla u(t, X_t)\}$. Inserting this solution and applying a temporal discretization on the time interval $[0, T] : 0 = t_0 < t_1 < \cdots < t_N = T$ using the Euler scheme gives

$$X_{t_{n+1}} - X_{t_n} \approx \mu(t_n, X_{t_n})\Delta t_n + \sigma(t_n, X_{t_n})\Delta W_n, \tag{3.2}$$

and

$$\begin{aligned} &u(t_{n+1}, X_{t_{n+1}}) - u(t_n, X_{t_n}) \\ \approx &- f\left(t_n, X_{t_n}, u(t_n, X_{t_n}), \sigma^{\mathrm{T}}(t_n, X_{t_n})\nabla u(t_n, X_{t_n})\right)\Delta t_n \\ &+ [\nabla u(t_n, X_{t_n})]^{\mathrm{T}}\sigma(t_n, X_{t_n})\Delta W_n, \end{aligned} \tag{3.3}$$

for $n = 1, \ldots, N-1$.

3.2 Approximation Using Deep Learning

Next, deep learning is used to approximate the function $x \mapsto \sigma^{\mathrm{T}}(t, x)\nabla u(t, x)$. This is done by letting an MLP model learn the value of $\sigma^{\mathrm{T}}(t, x)\nabla u(t, x)$ in every discretization point. Therefore, the model consists of $N-1$ subnetworks,

$$\sigma^{\mathrm{T}}(t_n, X_{t_n})\nabla u(t_n, X_{t_n}) = \sigma^{\mathrm{T}}\nabla u(t_n, X_{t_n}) \approx \sigma^{\mathrm{T}}\nabla u(t_n, X_{t_n}|\theta_n),$$

where $n = 1, 2, \ldots, N-1$, and θ_n denotes the network parameters for the subnetwork at $t = t_n$. In **Fig. 3.1** an example subnetwork is illustrated.

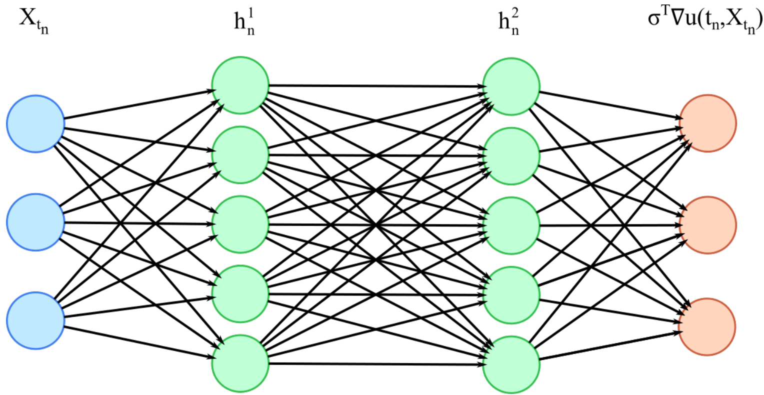


Figure 3.1: An illustration of an example of a subnetwork at time t_n. The subnetwork is fully connected, and the problem equation is 3-dimensional (3 units in the input and output layer). The neural network consists of 2 hidden layers, denoted by h_n^1 and h_n^2, with 5 units in each. Every line represents a parameter (or weight) that is to be optimized. All the parameters in this subnetwork are gathered in the set θ_n.

The input of the MLP model is the set of processes

$$\{X_{t_n}\}_{0 \leq n \leq N}, \quad \{W_{t_n}\}_{0 \leq n \leq N}.$$

These processes are sampled using equation (3.2). The solution at each discretization point $u(t_n, X_{t_n})$ for $n = 1, \dots, N$ is then computed using equation (3.3). The output of the model is the approximated solution at the terminal time,

$$\hat{u}(\{X_{t_n}\}_{0\leq n\leq N}, \{W_{t_n}\}_{0\leq n\leq N}).$$

A loss function is used to evaluate how well the model performs, and also to update the network parameters accordingly (see Subsection 2.1.2 for details). In the deep BSDE method, the expected loss function is defined to be the mean squared error between the output and the given terminal condition,

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

The initial values $u(0, X_0) \approx \theta_{u_0}$ and $\nabla u(0, X_0) \approx \theta_{\nabla u_0}$ are treated as parameters in the model, and the total set of network parameters is then $\theta = \{\theta_{u_0}, \theta_{\nabla u_0}, \theta_1, \dots, \theta_{N-1}\}$. The model updates these parameters iteratively using an SGD algorithm. More details on implementation is in Chapter 4. During training, Monte Carlo samples of the processes $\{X_{t_n}\}_{0\leq n\leq N}, \{W_{t_n}\}_{0\leq n\leq N}$ are performed to find the optimal network parameters θ. The solution $u(0, \xi)$ is then obtained by simulating the processes in the model.

3.3 Network Architecture

The network includes three types of connections. Arrows are used to show the flow of information.

Connection (i) $X_{t_n} \rightarrow h_n^1 \rightarrow h_n^2 \rightarrow \cdots \rightarrow h_n^H \rightarrow \sigma^{\mathrm{T}}\nabla u(t_n, X_{t_n})$. Each h_n^i, for $i = 1, \dots, H$, denotes a hidden layer in the MLP subnetwork at time step n. The subnetwork takes the sampled process X_{t_n} as input, and through training finds the optimal parameters θ_n in the H hidden layers. The solution process with the spatial gradient, $\sigma^{\mathrm{T}}\nabla u(t_n, X_{t_n})$, is approximated at each time step n by the corresponding MLP subnetwork for $n = 1, 2, \dots, N-1$.

Connection (ii) $(u(t_n, X_{t_n}), \sigma^{\mathrm{T}}\nabla u(t_n, X_{t_n}), \Delta W_n) \rightarrow u(t_{n+1}, X_{t_{n+1}})$. The solution for each time step is computed using the solution processes in the previous time step, along with the difference in the current and the previously sampled Brownian motion. The iteration is computed using equation (3.3).

Connection (iii) $(X_{t_n}, \Delta W_n) \rightarrow X_{t_{n+1}}$. The process $X_{t_{n+1}}$ is sampled using equation (3.2), where the previous processes X_{t_n} and W_n are used.

The architecture of the network is shown in **Fig. 3.2**, where the different connections are also marked. As seen from the figure, the network has in total $(H+1)(N-1)$ layers with optimization parameters. The output layer is $u(t_N, X_{t_N})$, where the loss function $L(\theta)$ is computed. The parameters θ are adjusted to minimize this loss function during training. It should be emphasized that the loss function is not measured at the spatial gradient of the solution at each time step. While connection (ii) and (iii) are straightforward computations, connection (i) contains optimization parameters. Connection (i) was illustrated in **Fig. 3.1** as an example of a subnetwork.

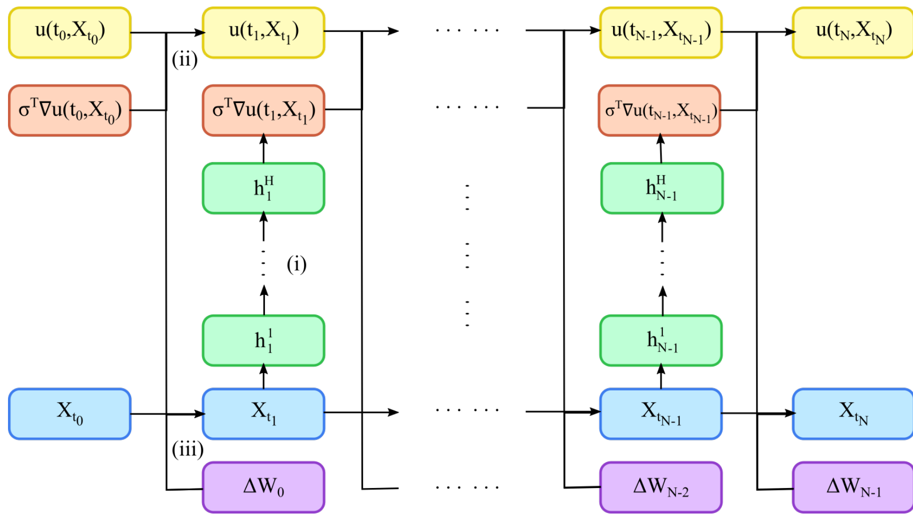


Figure 3.2: An illustration of the network architecture. The arrows show the flow of information in the network. Each column represents a time step in the discretization, t_n for $n = 0, 1, \dots, N$. Each MLP subnetwork for $n = 1, 2, \dots, N - 1$ contains H hidden layers, denoted by h_n^i for $i = 1, 2, \dots, H$. The different types of connections in the network are marked with (i), (ii) and (iii) corresponding to the given definitions. This illustration is based on the figure included in (Han et al., 2018).

Chapter 4

Implementation

In this chapter, specifics regarding the implementation of the deep BSDE method are explained. There are several options and alternatives to what is presented here, but a thorough description of the choices made in (Han et al., 2018) is provided to be able to both recreate and verify their obtained results. Different activation functions and the SGD algorithm Adam optimizer are discussed. To accelerate the training of the MLP, batch normalization is used, and the algorithm is explained here. The methods for finding the exact solution of both the Allen-Cahn equation and the HJB equation are described. These solutions are used when measuring the approximation error in the numerical experiments in the next chapter. Lastly, the actual implementation of the deep BSDE method in TensorFlow is described.

4.1 Activation Function

The ReLU is used as activation function $g(\mathrm{in}_j)$ in (Han et al., 2018), and was introduced in Subsection 2.1.2. It is defined by $g(z) = \max\{z, 0\}$, see **Fig. 4.1(a)**. This activation function is cheap to compute. It also converges quickly and unlike some other functions, like the sigmoid and tanh function, it does not suffer from the vanishing gradient problem (Nwankpa et al., 2018). The adverse effects of the vanishing gradient problem increase in deep neural networks with many hidden layers. It describes the situation when the error gradient that backpropogates through the network diminishes so much that it is too small when it reaches the initial layers. It will then have little effect and it will be more difficult to adjust the weights. It is because of the cheap computation, quick convergence and the persistent error gradient that the ReLU is commonly used in deep learning.

However, the ReLU has a disadvantage called "the dying ReLU". This term describes the situation when a unit continuously takes on negative values and therefore only returns zero. The unit will then be rendered useless. However, SGD methods compute the gradient over several data points, so unless they are all zero the problem is not critical and the approximation will continue to converge. Another challenge with ReLU is that it is not bounded from above. This can cause the activation to blow up. The sigmoid function

and tanh function do not risk this since they are both bounded. However, these activation functions are more computationally expensive. The tanh function differs from the sigmoid function in that it is zero centered, which makes it easier to handle strongly negative, neutral and strongly positive inputs. The softplus activation function is very similar to the ReLU, and has smoothing and nonzero gradient properties. Compared to the ReLU it is more expensive to compute. **Fig. 4.1** shows the four commonly used activation functions that have been mentioned. The choice of activation function depends on the characteristics of the problem to be solved.

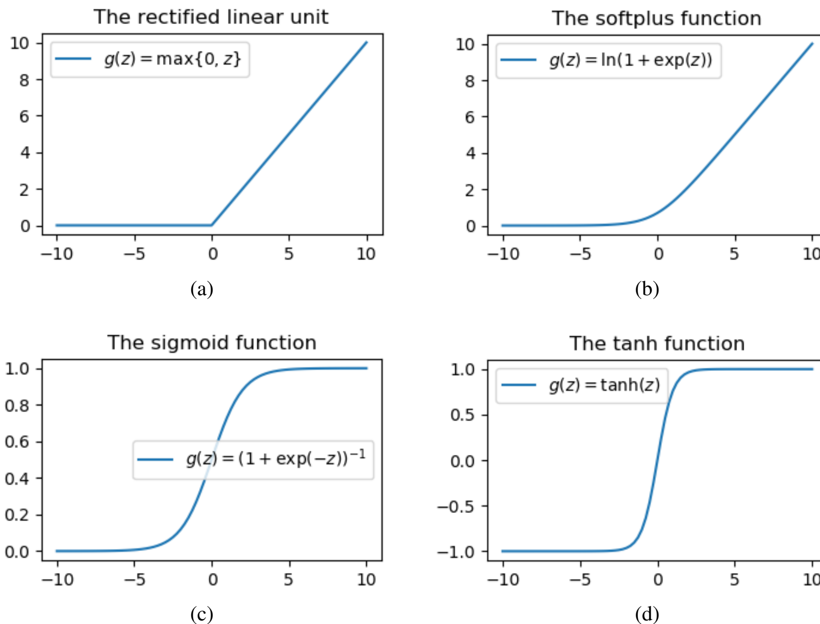


Figure 4.1: Plots of different activation functions. **(a)** The rectified linear unit (ReLU). **(b)** The softplus function. **(c)** The sigmoid function. **(d)** The tanh function.

4.2 Optimization Algorithm

As mentioned an SGD method is used to find the neural network parameters that minimize the given loss function. In an SGD method, the error gradient is computed using only a random sample from the complete data set, resulting in a much less computationally expensive operation. Since the entire data set is not accounted for at each iteration step, the path towards the minima is usually more “noisy” than for the normal gradient descent optimization, meaning that it requires more iterations. However, for larger data sets the SGD is usually preferred over gradient descent because it requires less computation time. This is the case in the learning of most neural networks.

Han et al. (2018) use the Adam optimizer in the training of each subnetwork in the model. This is a first order SGD-based optimization algorithm (Kingma and Ba, 2014),

meaning that it only uses the first order information to obtain the local minima of a function. First order information is retrieved from the first derivative term in the Taylor expansion of this function. The Adam optimizer only uses the gradient (first order) with respect to the network parameters to obtain the minima of the loss function. **Algorithm 2** gives the pseudo code for the Adam optimizer. The variable α denotes the step size (or learning rate). The step size is the amount the parameters are allowed to be changed at each iteration (often between 0 and 1). The loss function is given by $L(\theta)$, and θ_0 are the initialized parameters. The result is of course the optimized parameters θ_i that minimize the loss function. The evaluation of the loss function is done at random subsamples (mini-batches) of the network. The variable i simply denotes the iteration step. The gradient of the loss function with respect to θ at iteration i is stored in the variable $g_i = \nabla_\theta L_i(\theta)$.

The Adam optimizer uses estimates of first and second moments of the gradients to compute individual adaptive learning rates. The method gets its name from adaptive moment estimation. At each iteration it computes estimates of the first moment of the gradient m_i, and second moment of the gradient v_i. The variables β_1, $\beta_2 \in [0, 1)$ control the exponential decay rates of m_i and v_i. Because of the initialization, the moment estimates are biased towards zero. This is counteracted in the bias-corrected estimates $\hat{m}_i$ and $\hat{v}_i$. Lastly, at each iteration the network parameters are updated by $\theta_i = \theta_{i-1} - \alpha \cdot \frac{\hat{m}}{(\sqrt{\hat{v}_i}+\epsilon)}$.

Algorithm 2: The Adam optimizer (from Kingma and Ba (2014))

Data: $\alpha, \beta_1, \beta_2, L(\theta), \theta_0$;
Result: θ_i

1 $m_0 \leftarrow 0$;
2 $v_0 \leftarrow 0$;
3 $i \leftarrow 0$;
4 **while** θ_i *not convergent* **do**
5 $\quad i \leftarrow i + 1$;
6 $\quad g_i \leftarrow \nabla_\theta L_i(\theta_{i-1})$;
7 $\quad m_i \leftarrow \beta_1 \cdot m_{i-1} + (1 + \beta_1) \cdot g_i$;
8 $\quad v_i \leftarrow \beta_2 \cdot v_{i-1} + (1 - \beta_2) \cdot g_i^2$;
9 $\quad \hat{m}_i \leftarrow m_i/(1 - \beta_1^i)$;
10 $\quad \hat{v}_i \leftarrow v_i/(1 - \beta_2^i)$;
11 $\quad \theta_i \leftarrow \theta_{i-1} - \alpha \cdot \hat{m}_i/(\sqrt{\hat{v}_i} + \epsilon)$;
12 **end**

There are several advantages with the Adam optimizer. It is both memory and computationally efficient. In addition it combines advantages from other popular optimization algorithms, as it does not require a stationary loss function, and works well with sparse gradients. While classical SGD works with a constant single learning rate for all network parameters, the Adam optimizer computes individual learning rates for all of the parameters, that are also adapted during training. This contributes to achieving good results fast.

4.3 Batch Normalization

The training of a neural network consists of iteratively adjusting the network parameters using an SGD-type optimizer. A significant challenge with this process is the fact that the input of each layer changes between iterations after the parameters are updated. This challenge is called the internal covariate shift (Ioffe and Szegedy, 2015). It makes the training slower, and again networks with a large number of hidden layers are more prone to this negative effect. To accelerate the training, a technique of batch normalization can be used. This is performed on each dimension in the input, before the activation function is applied. Batch normalization makes sure that the distribution of the inputs remains stable during training by fixing the means and variances of each layer input. The goal of batch normalization is therefore to mitigate the internal covariate shift and thereby accelerate the network training.

To achieve fixed distributions of inputs, the inputs of each layer are linearly transformed to have zero means and unit variances. **Algorithm 3** gives the pseudo code for the batch normalization. In line **1** and **2** the mean and the variance of the mini-batch is computed. Each dimension in the input is then normalized using the mean and variance in line **3**. In the last line, line **4**, the parameters γ and β scale and shift the normalized value. These parameters are learned by the network along with the model parameters θ. By learning these parameters the expressive power of the network remains. This can be seen by setting $\gamma_i = \sqrt{\text{Var}(x_i)}$ and $\beta_i = \text{E}(x_i)$, which would retrieve the original input. In the pseudo code $\hat{x}_i$ denotes the normalized value, and y_i denotes the linear transformation.

Algorithm 3: The batch normalization transform (from Ioffe and Szegedy (2015))

Data: Values of x over a mini-batch: $\mathcal{B} = \{x_{1\ldots m}\}$;
Parameters to be learned: γ, β;
Result: $\{y_i = \text{BN}_{\gamma,\beta}(x_i)\}$

1. $\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i$;
2. $\sigma_{\mathcal{B}}^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2$;
3. $\hat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}}$;
4. $y_i \leftarrow \gamma\hat{x}_i + \beta \equiv \text{BN}_{\gamma,\beta}(x_i)$;

After normalization the training does not depend as much on the scale of the gradient or of the parameter initialization. It is therefore possible to have a higher learning rate without risking divergence. Normally if the learning rate is too large, the optimal solution may be passed and the algorithm can fail to converge or even diverge. With a learning rate that is too small, the convergence can be very slow resulting in time consuming training. With batch normalization, parameters can be easily initialized from a normal or uniform distribution without pretraining.

4.4 Exact Solution Methods

In the numerical experiments in the next chapter, the accuracy of the model is measured by comparing the approximated solution of the semilinear parabolic PDEs to a given exact

solution. In this section, the approaches for obtaining these exact solutions are explained.

4.4.1 Branching-Diffusion Method for the Allen-Cahn Equation

The solution of the Allen-Cahn equation is approximated using the branching-diffusion method given in Appendix B in (Weinan et al., 2017). The methodology is introduced, but a thorough explanation is beyond the scope of this thesis. The method exploits that the solution of semilinear PDEs with polynomial nonlinearity can be represented as an expectation of a functional of branching diffusion processes. As explained in Subsection 2.2.2, BSDEs provide a nonlinear Feynman-Kac formula for semilinear parabolic PDEs in the Markovian case. To solve this BSDE, a branching diffusion process is constructed as follows: a particle starts at time t, in position x, performs a diffusion process, dies after a mean β exponential time and produces k i.i.d. descendants with probability p_k. The descendants perform the diffusion process driven by independent Brownian motions, and die and reproduce i.i.d. descendants independently after independent exponential times, etc, (Henry-Labordere et al., 2014). A diffusion process is a solution to an SDE, and in this case it refers to the process $\{X_t\}_{t\in[0,T]}$ from Subsection 2.2.2. The branching process is therefore constructed by independent Brownian motions and exponential random variables. The solution of this branching diffusion is a viscosity solution to a corresponding semilinear path dependent PDE. By uniqueness, the numerical solution is the solution of the corresponding BSDE.

The obtained solution of the Allen-Cahn equation with terminal time $T = 0.3$ and terminal condition $h(x) = 1/(2 + 0.4||x||^2)$, at $t = 0$, $x = (0, 0, \dots, 0) \in \mathbb{R}^{100}$ for $M = 10^7$ Monte Carlo simulations is $u(0, (0, 0, \dots, 0)) \approx 0.0528$ with a runtime of 1316 seconds. This will act like the not explicitly known exact solution.

4.4.2 Monte Carlo Method for the Hamilton-Jacobi-Bellman Equation

The exact solution of the HJB equation is given by

$$u(t,x) = -\frac{1}{\lambda} \ln\left(\mathbb{E}\left[\exp\left(-\lambda h(x + \sqrt{2}W_{T-t})\right)\right]\right).$$

The solution of the HJB equation with terminal time $T = 1$ and terminal condition $h(x) = \ln((1 + ||x||^2)/2)$, at $t = 0$, $x = (0, 0, \dots, 0) \in \mathbb{R}^{100}$ for $\lambda = 1$ is computed using the Monte Carlo method given in Appendix B in (Weinan et al., 2017). The solution is obtained from $M = 10^7$ Monte Carlo simulations to be $u(0, (0, 0, \dots, 0)) \approx 4.5901$ with a runtime of 171 seconds.

4.5 Machine Learning Platform

TensorFlow is a system for large-scale machine learning (Abadi et al., 2016). This system is used to implement the deep BSDE method. The TensorFlow platform is end-to-end open source and therefore available to anyone. It was specifically developed to conduct

machine learning and deep neural networks research, and because of its generality is applicable to various machine learning tasks. The system creates dataflow graphs on which computations are managed. It works very well with training large models, and offers the user to explore different optimization algorithms and model architectures without having to modify the core system. Further reading on the TensorFlow system can be found in (Abadi et al., 2016).

4.6 The Deep BSDE Method Implementation

The MLP model is implemented as a python class for the specific semilinear parabolic PDE to be solved (for example `class SolveAllenCahn`), like in (Weinan et al., 2017). This is beneficial since the model contains numerous variables that within the class are accessible to all class functions. When running the model, two main functions are called: `build` and `train`. In this section significant functions in the implementation are described. The entire program for solving the Allen-Cahn equation is provided in Appendix B, and for solving the HJB equation in Appendix C. The model is implemented with the same initialization values and hyperparameter values as in (Weinan et al., 2017).

First some basics about the setup in TensorFlow is explained. Computations in TensorFlow are performed on a graph. In order to create the graph, the data structures involved have to be defined. The data structures in TensorFlow are called tensors, and can store vectors and matrices. There are different kinds of tensors, and in this model *placeholders* and *variables* are used. Placeholders are usually used for the model input, and their value is provided when the graph is run. The value of variables, on the other hand, can change during the run of a graph, and they need to be initialized before a run. *get_variable* is also used, and special for this tensor is that it only creates a new variable the first time it is run. If it is run again it just retrieves the existing one with belonging parameters. In the TensorFlow framework, all computations on the graph are done in a so called *session*.

The PDE solver class contains an `__init__` function that initializes the class with the model variables shown in **Table 4.1**.

Table 4.1: The variables that are defined in the `__init__` function in the solver classes.

Variable	Definition
`d`	Dimension of problem equation
`d_h`	Dimension of hidden layers
`T`	Terminal time
`N`	Temporal discretization steps
`l`	Step size `T/N`
`time_stamp`	Array containing all `N+1` time stamps from 0 to `T`
`batch_size`	Number of random paths used in training
`M`	Number of random paths used in testing
`n_maxstep`	Maximum iteration steps
`n_displaystep`	Constant used to display results periodically
`learning_rate`	Learning rate used in training the neural network
`extra_train_ops`	Array storing extra training operations

After the model is defined, the functions `build` and `train` are called. **Fig. 4.2** shows the structure of the program code by presenting the tasks in each function. Descriptions of the functions `build` and `train` are given here:

`build` This function creates the whole neural network graph. It defines the placeholders `X` and `dW` that are used to store the input of the model. The variables `Y0` and `Z0` are also defined to store the values of $u(t_0, X_{t_0})$ and $\sigma^{\mathrm{T}} \nabla u(t_0, X_{t_0})$ respectively. Next, the forward connections (ii) are created (see **Fig. 4.3**). The function `subnetwork` is called to create connection (i) (see **Fig. 4.4**). In `subnetwork`, the function `add_layer` is called to create the layers in each subnetwork. It creates the network weights, defines the belonging input function, performs batch normalization using the function `batch_norm`, and for the hidden layers applies the ReLU activation function. Then, the loss function is defined as the squared difference between the terminal condition and output value. The difference is clipped to a specified minimum and maximum value to avoid letting extreme cases affect the learning negatively.

`train` This function trains the model to solve the semilinear parabolic PDE. It starts by defining the Adam optimizer as optimizer to minimize the loss with respect to all trainable variables/weights. The extra training operations consist of computing the moving averages of variables in the batch normalization (this is only performed when the model is training). Then M number of Monte Carlo simulations of `X` and `dW` (connection (iii)) are performed using the function `sample_path` (see **Fig. 4.5**) to create the test data. Next, all global variables are initialized, and the model is ready to train. The model is trained iteratively on a batch size of `X` and `dW` (connection (iii), see **Fig. 4.5**) using the defined training operations. At each display step, the loss is tested against the Monte Carlo simulations, the current value of `Y0`, and the total runtime of building and training the model up until the current iteration step are displayed.

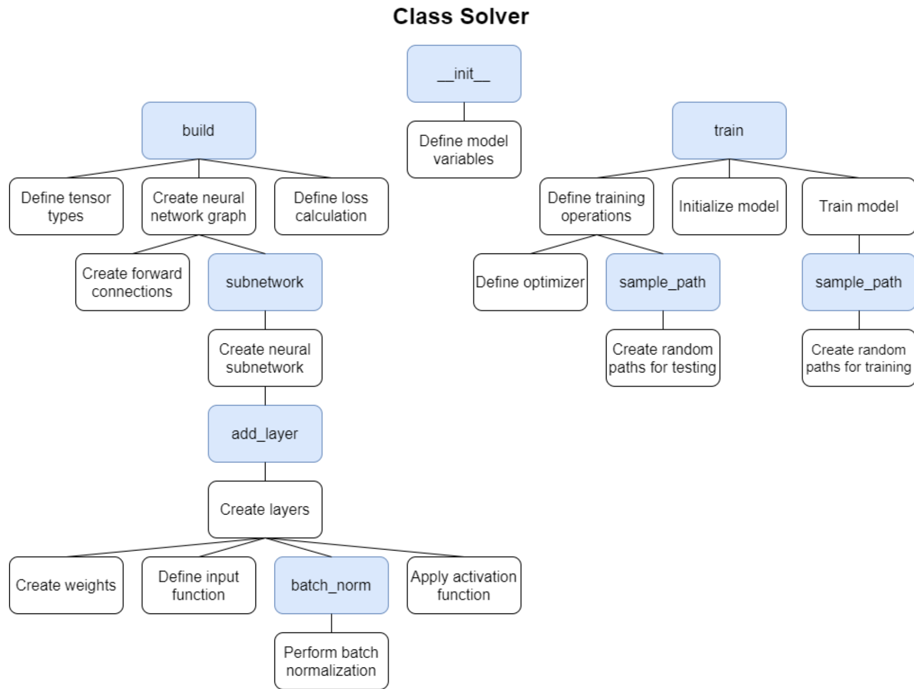


Figure 4.2: Illustration of the structure of the program code (for the entire program code, see Appendix B and C). The blue boxes refer to functions, and the white boxes describe the tasks that are performed in the parenting function.

Fig. 4.3, **4.4** and **4.5** show where the different connections in the network architecture in **Fig. 3.2** is created in the program code.

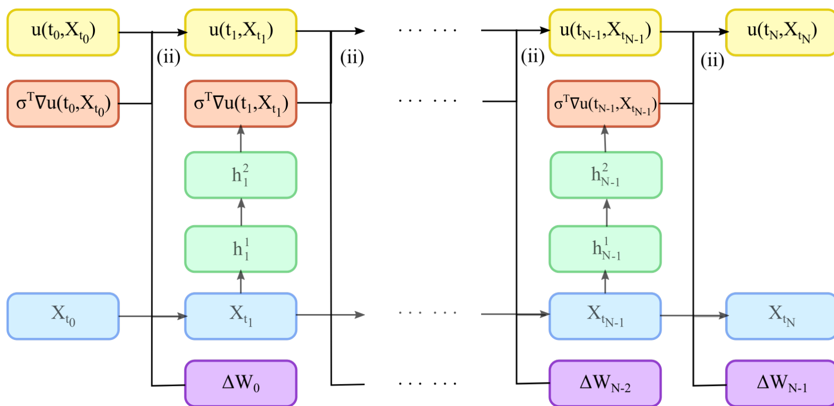


Figure 4.3: Illustration of connection (ii) from Subsection 3.3. In the implementation, this connection is created in function `build` in variable scope `forward connections` (for the entire program code, see Appendix B and C).

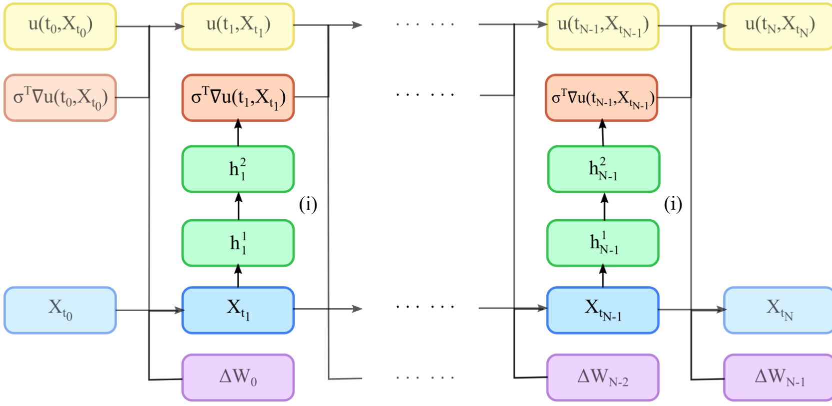


Figure 4.4: Illustration of connection (i) from Subsection 3.3. In the implementation, this connection is created in function `build` using function `subnetwork` and `add_layer` (for the entire program code, see Appendix B and C).

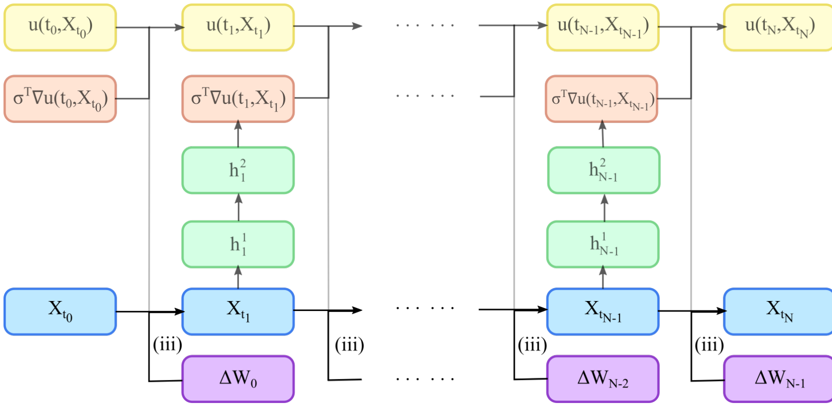


Figure 4.5: Illustration of connection (iii) from Subsection 3.3. In the implementation, this connection is created in function `train` using function `sample_path` (for the entire program code, see Appendix B and C).

Pseudocodes for all mentioned functions are presented in **Algorithm 4**, **5**, **6**, **7** and **8**. The batch normalization was discussed in Section 4.3.

Algorithm 4: Pseudocode for the function `build` in the deep BSDE implementation in Appendix B and C.

Data: Class variables defined in `__init__`;
Result: Neural network graph
1 X placeholder, dW placeholder, Y_0 variable, Z_0 variable;
2 **for** $n \leftarrow 0$ **to** $N-2$ **do**
3 $\quad Y_{t_{n+1}} = Y_{t_n} - f(t_n, X_{t_n}, Y_{t_n}, Z_{t_n})l + Z_{t_n} dW_n$;
4 $\quad Z_{t_{n+1}} = \texttt{subnetwork}(X_{t_n})$;
5 **end**
6 $Y_{t_N} = Y_{t_{N-1}} - f(t_{N-1}, X_{t_{N-1}}, Y_{t_{N-1}}, Z_{t_{N-1}})l + Z_{t_{N-1}} dW_{N-1}$;
7 loss = $|Y_{t_N} - g(X_T)|^2$;

Algorithm 5: Pseudocode for the function `subnetwork` in the deep BSDE implementation in Appendix B and C.

Data: x; Class variables defined in `__init__`;
Result: z_layer connected to x through two hidden layers
1 x_layer = `batch_norm`(x);
2 $layer1$ = `add_layer`(x, dh, ReLU) (hidden layer fully connected from x, dh units, activation function = ReLU);
3 $layer2$ = `add_layer`($layer1$, dh, ReLU) (hidden layer fully connected from $layer1$, dh units, activation function = ReLU);
4 z_layer = `add_layer`($layer2$, dh, None) (hidden layer fully connected from $layer2$, dh units);

Algorithm 6: Pseudocode for the function `add_layer` in the deep BSDE implementation in Appendix B and C.

Data: $input$,dim,$activation$; Class variables defined in `__init__`;
Result: Network layer
1 w Variable (weights with dim units);
2 $layer = w \cdot input$ (input function);
3 $layer_bn = \texttt{batch_norm}(layer)$;
4 activation($layer_bn$)

Algorithm 7: Pseudocode for the function `train` in deep BSDE implementation in Appendix B and C.

Data: Class variables defined in `__init__`;
Result: Trained model

1. $grads$ = gradient of loss w.r.t. trainable variables (weights);
2. $optimizer$ = Adam optimizer;
3. dW_test, X_test =`sample_path`(M);
4. Initialize model;
5. **for** 0 **to** $n_maxstep$ **do**
6. dW_train, X_train =`sample_path`($batch_size$);
7.   Train model;
8. **end**

Algorithm 8: Pseudocode for the function `sample_path` in the deep BSDE implementation in Appendix B and C.

Data: n_sample; Class variables defined in `__init__`;
Result: n_sample number of random paths for dW and X

1. **for** $i \leftarrow 0$ **to** $N-1$ **do**
2. $dW_{i+1} \sim \mathcal{N}$(mean=0, cov=1, size=$n_sample$)$\cdot l$;
3. $X_{i+1} = X_i + \sqrt{2} \cdot dW_i$
4. **end**

Chapter 5

Numerical Experiments

This chapter presents the results from numerical experiments using the deep BSDE method to solve both the Allen-Cahn equation and the HJB equation. The method is implemented in TensorFlow on a Lenovo with a 2.4 GHz Intel Core i5 CPU and 8 GB RAM.

Some numerical results are obtained by running several independent runs. This is done using different random seeds for the NumPy and TensorFlow libraries, so that the sampled paths and randomized initialization change for each run. To measure the performance of the method, the approximated values at the initial time of the terminal value problems are compared to given solutions. The approximated value at $u(0,\xi)$ obtained from the deep BSDE method is denoted $\hat{u}_\theta$. The relative approximation error in the numerical experiments are given in the $L1$-norm as the following,

$$E = \left|\left|\frac{u(0,\xi) - \hat{u}_\theta}{u(0,\xi)}\right|\right|_1 .$$

5.1 The Allen-Cahn Equation

The d-dimensional Allen-Cahn equation for $x \in \mathbb{R}^d$ is given by

$$\frac{\partial u}{\partial t}(t,x) + \Delta u(t,x) + u(t,x) - [u(t,x)]^3 = 0, \quad t \in [0,T),$$
$$u(T,x) = h(x) = \frac{1}{2 + 0.4||x||^2}.$$

The model variables that express this problem equation from the general formulation in Subsection 2.2.2 are therefore

$$\mu(t,x) = \mu = 0,$$
$$\sigma = \sqrt{2},$$
$$f(t,x,u(t,x),\sigma^{\mathrm{T}}(t,x)\nabla u(t,x)) = f(u(t,x)) = u(t,x) - [u(t,x)]^3.$$

5.1.1 Model Variables

The MLP model for the deep BSDE method is implemented with a total of four layers for each subnetwork, of which two are hidden layers. The temporal discretization is set to $N = 20$ equidistant time steps from initial time $t_0 = 0$ to the terminal time $t_N = T = 0.3$. The total number of layers with parameters to be optimized is then $(H + 1)(N - 1) = (2+1)(20-1) = 57$. Each subnetwork is fully connected as described in Subsection 2.1.3 about MLP models. The input and output layer are both d-dimensional, and the hidden layers will be $d + 10$-dimensional. In the numerical experiments, the 100-dimensional Allen-Cahn equation is considered. The solution is examined in the space point $\xi = (0, 0, \ldots, 0)$. The model is trained with a learning rate of $\alpha = 5e - 4$ on 64 sample paths, and tested against 256 Monte Carlo samples. All variable values are summarized in **Table 5.1**. The ReLU is used as activation function, and the Adam optimizer is used for training the ANN. The not explicitly known exact solution is approximated using the branching-diffusion method (see Subsection 4.4.1). The approximation error is therefore measured relative to $u(0, (0, 0, \ldots, 0)) \approx 0.0528$.

Table 5.1: The values of the model variables used to solve the Allen-Cahn equation using the deep BSDE method.

Variable	Symbol	Value
Dimension of problem equation	d	100
Number of hidden layers	H	2
Dimension of hidden layers	d_h	110
Temporal discretization steps	N	20
Terminal time	T	0.3
Space point of interest	ξ	$(0, 0, \ldots, 0)$
Terminal condition	$h(x)$	$1/(2 + 0.4\|\|x\|\|^2)$
Batch size	m	64
Monte Carlo samples	M	256
Learning rate	α	$5e - 4$

5.1.2 Numerical Results

The model variables are set to the values shown in **Table 5.1**. **Table 5.2** presents the numerical results when solving the 100-dimensional Allen-Cahn equation on five independent runs using the specific random seeds $\{1, 2, 3, 4, 5\}$. From the table, one can see that the method obtains a relative error of 0.20% after 4000 iteration steps. The relative approximation error history is shown in **Fig. 5.1**. To further examine how much the randomization affects the method, the same experiment was conducted using the five new random seeds $\{6, 7, 8, 9, 10\}$. **Table 5.3** shows the numerical results, and **Fig. 5.2** presents the corresponding history of relative approximation error. This time, the method obtains a relative error of 0.42%. The values at all display steps (every 100th iteration) are included in Appendix D.

Table 5.2: Numerical results after solving the Allen-Cahn equation with values in **Table 5.1** and for five independent runs using random seeds $\{1, 2, 3, 4, 5\}$. The runtime is given for one of the runs in seconds.

Number of iteration steps	**0**	**1000**	**2000**	**3000**	**4000**
Mean of $\hat{u}_\theta$	0.4163	0.1122	0.0568	0.0530	0.0529
Standard deviation of $\hat{u}_\theta$	0.0748	0.0450	0.0058	0.0002	0.0001
Mean of rel. approx. error	6.8841	1.1256	0.0767	0.0054	0.0020
Standard deviation of rel. approx. error	1.4161	0.8517	0.1091	0.0023	0.0018
Mean of loss	0.087874	0.003335	0.000273	0.000221	0.000178
Standard deviation of loss	0.038434	0.003743	0.000055	0.000023	0.000029
Runtime in seconds	107	280	413	531	721

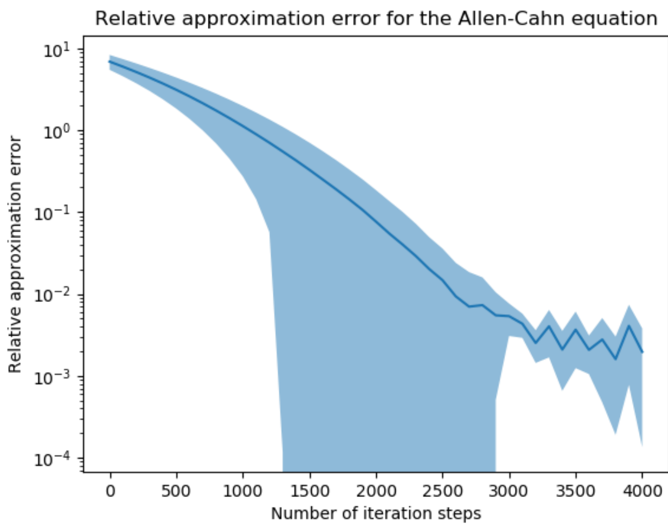


Figure 5.1: A plot of the relative approximation error when solving the Allen-Cahn equation using the deep BSDE method as a function of number of iteration steps. The dark line is the mean of the relative error after five independent runs using the random seeds $\{1, 2, 3, 4, 5\}$, and the shaded area shows the mean $\pm$ the standard deviation of the relative error.

Table 5.3: Numerical results after solving the Allen-Cahn equation with values in **Table 5.1** and for five independent runs using random seeds $\{6, 7, 8, 9, 10\}$. The runtime is given for one of the runs in seconds.

Number of iteration steps	**0**	**1000**	**2000**	**3000**	**4000**
Mean of $\hat{u}_\theta$	0.5034	0.1711	0.0693	0.0535	0.0530
Standard deviation of $\hat{u}_\theta$	0.0968	0.0652	0.0128	0.0005	0.0002
Mean of rel. approx. error	8.5335	2.2410	0.3116	0.0138	0.0042
Standard deviation of rel. approx. error	1.8331	1.2356	0.2430	0.0088	0.0027
Mean of loss	0.143701	0.010732	0.000499	0.000227	0.000194
Standard deviation of loss	0.055645	0.007724	0.000218	0.000016	0.000024
Runtime in seconds	102	288	410	538	693

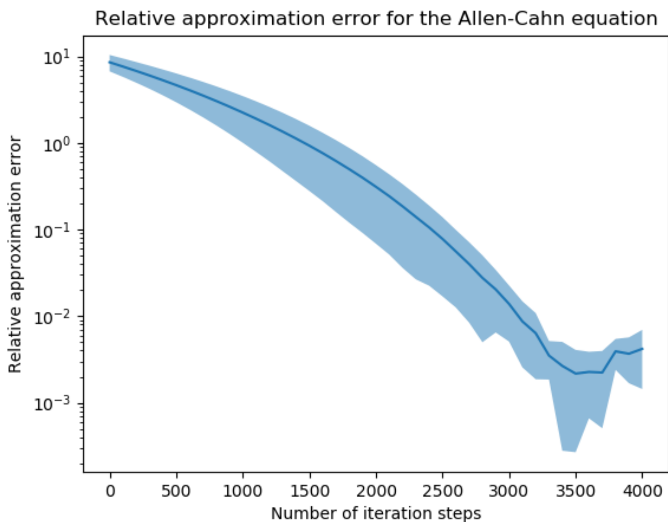


Figure 5.2: A plot of the relative approximation error when solving the Allen-Cahn equation using the deep BSDE method as a function of number of iteration steps. The dark line is the mean of the relative error after five independent runs using the random seeds $\{6, 7, 8, 9, 10\}$, and the shaded area shows the mean $\pm$ the standard deviation of the relative error.

The approximated initial value, $\hat{u}_\theta$, for five independent runs of the deep BSDE method is shown in **Fig. 5.3** against number of iteration steps. The loss function is given by $L(\theta) = \mathbb{E}\left[|h(X_T) - \hat{u}(\{X_{t_n}\}_{0\leq n\leq N}, \{W_{t_n}\}_{0\leq n\leq N})|^2\right]$ as described in Section 3.2, taking the difference between the terminal condition $h(X_T)$ and the approximated solution at the terminal time $\hat{u}(\{X_{t_n}\}_{0\leq n\leq N}, \{W_{t_n}\}_{0\leq n\leq N})$. The gradient of this loss function with respect to θ is used by the Adam optimizer to update the parameters θ at each iteration step. **Fig. 5.4** shows the loss function against the number of iteration steps. The deep BSDE method is also used to compute the time evolution of $u(t,(0,0,\ldots,0))$ for $t \in [0, 0.3]$. The result is shown in **Fig. 5.5** together with the corresponding result obtained by the branching-diffusion method.

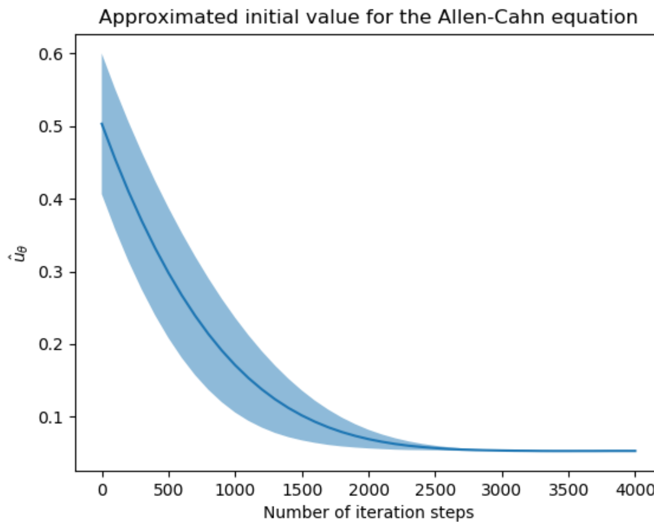


Figure 5.3: A plot of the approximated initial value when solving the Allen-Cahn equation using the deep BSDE method as a function of number of iteration steps. The dark line is the mean of $\hat{u}_\theta$ after five independent runs, and the shaded area shows the mean $\pm$ the standard deviation of $\hat{u}_\theta$.

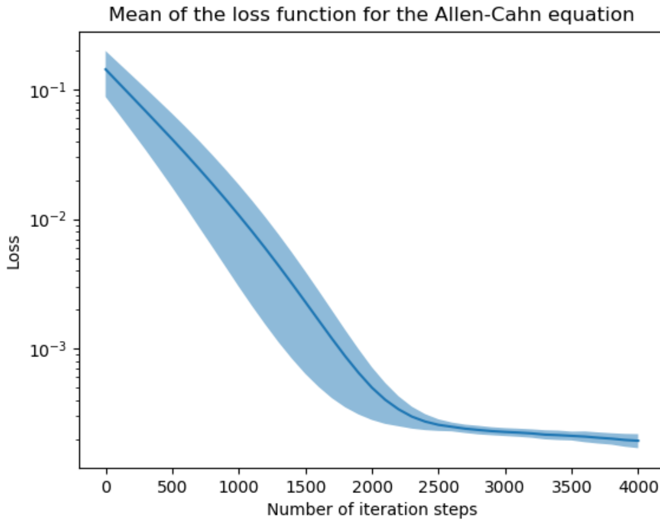


Figure 5.4: A plot of the loss function when solving the Allen-Cahn equation using the deep BSDE method as a function of number of iteration steps. The dark line is the mean of the loss function after five independent runs, and the shaded area shows the mean $\pm$ the standard deviation of the loss function.

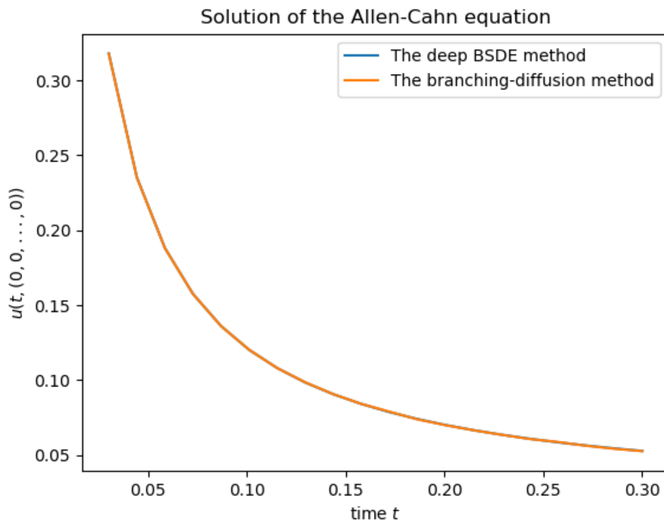


Figure 5.5: A plot of the time evolution of the solution to the Allen-Cahn equation, $u(t,(0,0,...,0))$, using both the deep BSDE method and the branching-diffusion method. The blue graph is barely visible as the two graphs are superimposed.

5.2 The Hamilton-Jacobi-Bellman Equation

The d-dimensional HJB equation for $x \in \mathbb{R}^d$ is given by

$$\frac{\partial u}{\partial t}(t,x) + \Delta u(t,x) - \lambda ||\nabla u(t,x)||^2 = 0, \quad t \in [0,T),$$
$$u(T,x) = h(x) = \ln\left(\frac{1+||x||^2}{2}\right).$$

The model variables that express this problem equation from the general formula are therefore

$$\mu(t,x) = \mu = 0,$$
$$\sigma = \sqrt{2},$$
$$f(t,x,u(t,x),\sigma^{\mathrm{T}}(t,x)\nabla u(t,x)) = f(u(t,x)) = -\lambda ||\nabla u(t,x)||^2.$$

5.2.1 Model Variables

The MLP model for the deep BSDE method is implemented with a total of four layers for each subnetwork, of which two are hidden layers. The temporal discretization is set to $N = 20$ equidistant time steps from initial time $t_0 = 0$ to the terminal time $t_N = T = 1$. The total number of layers with parameters to be optimized is then $(H+1)(N-1) = (2+1)(20-1) = 57$. Each subnetwork is fully connected as described in Subsection 2.1.3 about MLP models. The input and output layer are both d-dimensional, and the hidden layers will be $d+10$-dimensional. In the numerical experiments, the 100-dimensional HJB equation for $\lambda = 1$ is considered. The solution is examined in the space point $\xi = (0,0,\dots,0)$. The model is trained with a learning rate of $\alpha = 1e-02$ on 64 sample paths, and tested against 256 Monte Carlo samples. All variable values are summarized in **Table 5.4**. The ReLU is used as activation function, and the Adam optimizer is used for training the ANN. The exact solution is derived using Itô's formula and given by

$$u(t,x) = -\frac{1}{\lambda}\ln\left(\mathbb{E}\left[\exp\left(-\lambda h(x+\sqrt{2}W_{T-t})\right)\right]\right).$$

The solution at the initial time at the origin is found by Monte Carlo simulations to be $u(0,(0,0,\dots,0)) \approx 4.5901$ (see Subsection 4.4.2).

Table 5.4: The values of the model variables used to solve the HJB equation using the deep BSDE method.

Variable	**Symbol**	**Value**
Dimension of problem equation	d	100
Number of hidden layers	H	2
Dimension of hidden layers	d_h	110
Temporal discretization steps	N	20
Level of control	λ	1
Terminal time	T	1
Space point of interest	ξ	$(0, 0, \ldots, 0)$
Terminal condition	$h(x)$	$\ln\big((1 + \|\|x\|\|^2)/2\big)$
Batch size	m	64
Monte Carlo samples	M	256
Learning rate	α	$1e-02$

5.2.2 Numerical Results

The model variables are set to the values shown in **Table 5.4**. **Table 5.5** presents the numerical results when solving the 100-dimensional HJB equation on five independent runs using the specific random seeds $\{1, 2, 3, 4, 5\}$. From the table, one can see that the method obtains a relative error of 0.22% after 2000 iteration steps. The relative approximation error history is shown in **Fig. 5.6**. To further examine how much the randomization affects the method, the same experiment was conducted using the five new random seeds $\{6, 7, 8, 9, 10\}$. **Table 5.6** shows the numerical results, and **Fig. 5.7** presents the corresponding history of relative approximation error. This time, the method obtains a relative error of 0.23%. The values at all display steps (every 100th iteration) are included in Appendix E.

Table 5.5: Numerical results after solving the HJB equation with values in **Table 5.4** and for five independent runs using random seeds $\{1, 2, 3, 4, 5\}$. The runtime is given for one of the runs in seconds.

Number of iteration steps	**0**	**500**	**1000**	**1500**	**2000**
Mean of $\hat{u}_\theta$	0.4163	3.3151	4.5146	4.5991	4.6001
Standard deviation of $\hat{u}_\theta$	0.0748	0.0858	0.0168	0.0010	0.0008
Mean of rel. approx. error	0.9093	0.2778	0.0165	0.0020	0.0022
Standard deviation of rel. approx. error	0.0163	0.0187	0.0037	0.0002	0.0002
Mean of loss	17.470558	1.749224	0.030793	0.020826	0.020852
Standard deviation of loss	0.582694	0.185690	0.003093	0.000864	0.000747
Runtime in seconds	101	284	359	506	639

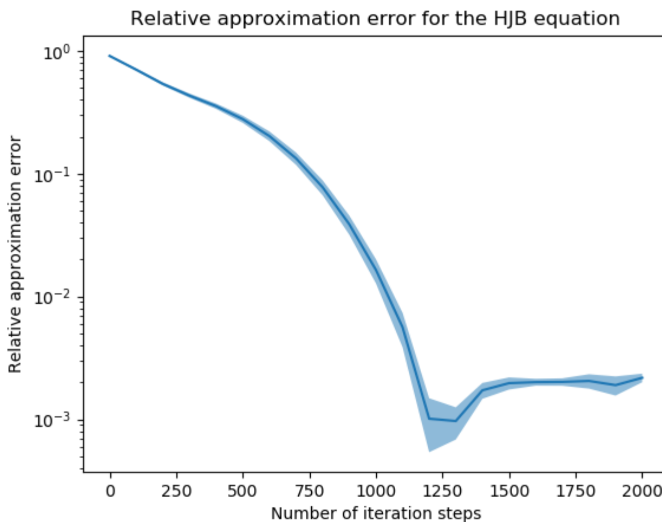


Figure 5.6: A plot of the relative approximation error when solving the HJB equation using the deep BSDE method as a function of number of iteration steps. The dark line is the mean of the relative error after five independent runs using the random seeds $\{1, 2, 3, 4, 5\}$, and the shaded area shows the mean $\pm$ the standard deviation of the relative error.

Table 5.6: Numerical results after solving the HJB equation with values in **Table 5.4** and for five independent runs using random seeds $\{6, 7, 8, 9, 10\}$. The runtime is given for one of the runs in seconds.

Number of iteration steps	**0**	**500**	**1000**	**1500**	**2000**
Mean of $\hat{u}_\theta$	0.5034	3.4062	4.5319	4.5989	4.6008
Standard deviation of $\hat{u}_\theta$	0.0968	0.1056	0.0201	0.0010	0.0017
Mean of rel. approx. error	0.8903	0.2579	0.0127	0.0019	0.0023
Standard deviation of rel. approx. error	0.0211	0.0230	0.0044	0.0002	0.0004
Mean of loss	16.681912	1.494773	0.029087	0.021636	0.021417
Standard deviation of loss	0.760256	0.245718	0.007968	0.001342	0.001430
Runtime in seconds	103	226	319	410	494

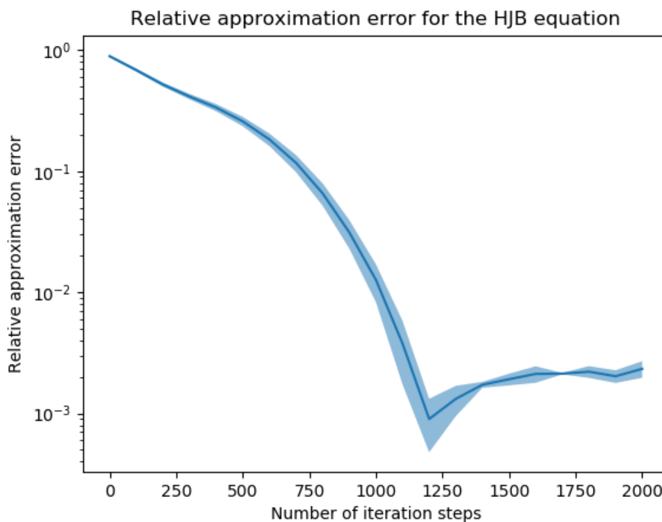


Figure 5.7: A plot of the relative approximation error when solving the HJB equation using the deep BSDE method as a function of number of iteration steps. The dark line is the mean of the relative error after five independent runs using the random seeds $\{6, 7, 8, 9, 10\}$, and the shaded area shows the mean $\pm$ the standard deviation of the relative error.

The approximated initial value, $\hat{u}_\theta$, for five independent runs of the deep BSDE method is shown in **Fig. 5.8** against number of iteration steps. The loss function is given by $L(\theta) = \mathbb{E}\left[|h(X_T) - \hat{u}(\{X_{t_n}\}_{0\leq n\leq N}, \{W_{t_n}\}_{0\leq n\leq N})|^2\right]$ as described in Section 3.2, taking the difference between the terminal condition $h(X_T)$ and the approximated solution at the terminal time $\hat{u}(\{X_{t_n}\}_{0\leq n\leq N}, \{W_{t_n}\}_{0\leq n\leq N})$. The gradient of this loss function with respect to θ is used by the Adam optimizer to update the parameters θ at each iteration step. **Fig. 5.9** shows the loss function against the number of iteration steps.

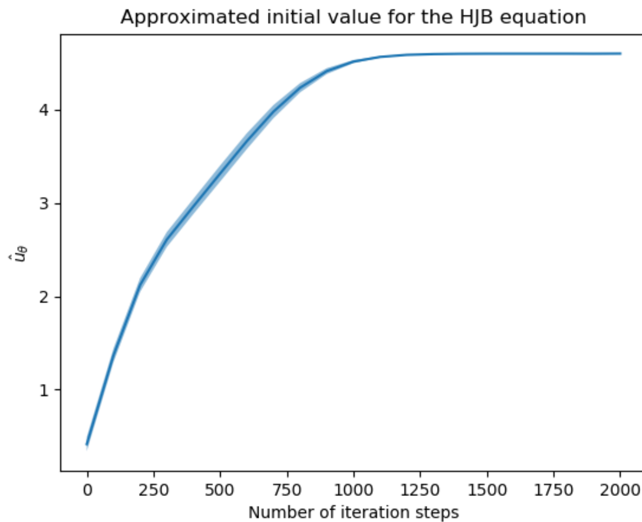


Figure 5.8: A plot of the approximated initial value when solving the HJB equation using the deep BSDE method as a function of number of iteration steps. The dark line is the mean of $\hat{u}_\theta$ after five independent runs, and the shaded area shows the mean $\pm$ the standard deviation of $\hat{u}_\theta$.

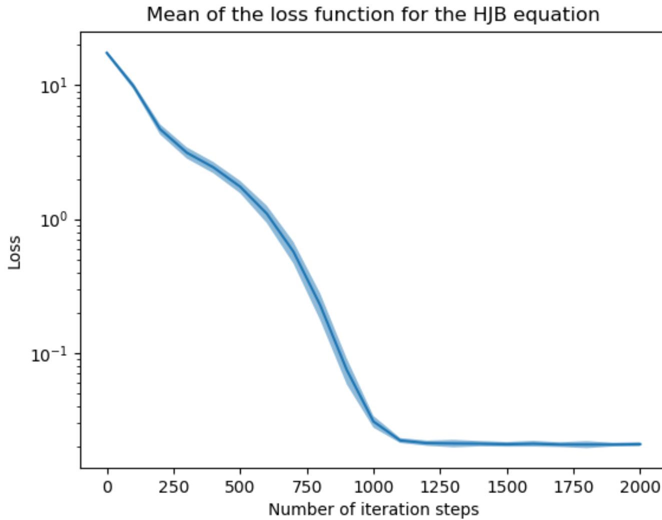


Figure 5.9: A plot of the loss function when solving the HJB equation using the deep BSDE method as a function of number of iteration steps. The dark line is the mean of the loss function after five independent runs, and the shaded area shows the mean $\pm$ the standard deviation of the loss function.

5.3 Model Features

In the previous sections, the deep BSDE method was tested by solving the Allen-Cahn equation and the HJB equation using the same variable values as in (Han et al., 2018). Now, the method will be further explored by varying the different features included in the model. The features that are going to be tested are the type of activation function, the batch size, and the number of temporal discretization steps.

5.3.1 Type of Activation Function

Different activation functions with their advantages and disadvantages were discussed in Section 4.1. The choice of activation function depends on the nature of the data. **Fig. 5.10** shows the mean relative approximation error for solving the Allen-Cahn equation after five independent runs using the different activation functions. The numerical results after 4000 iterations are shown in **Table 5.7**. The same results when solving the HJB equation can be found in **Fig. 5.11** and **Table 5.11**.

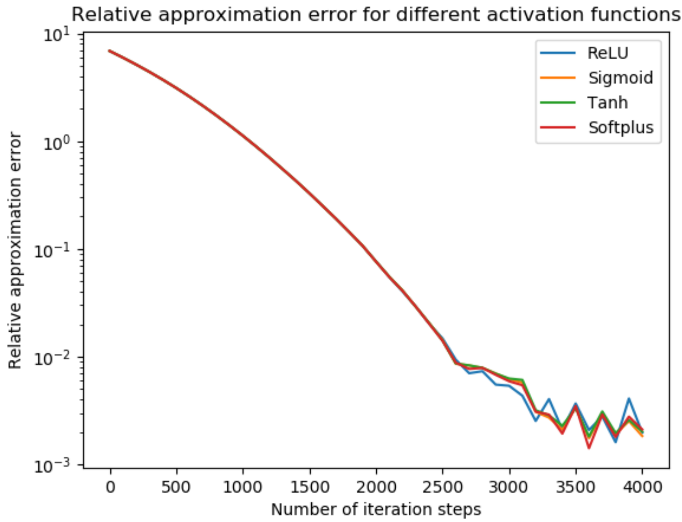


Figure 5.10: A plot of the mean relative approximation error when solving the Allen-Cahn equation as a function of number of iteration steps. It is solved using the deep BSDE method for five independent runs using four different activation functions.

Table 5.7: Numerical results after solving the Allen-Cahn equation with values in **Table 5.1** for five independent runs using four different activation functions. The runtime is given for one of the runs in seconds.

Activation function	**Mean of rel. approx. error**	**Standard deviation of rel. approx. error**	**Runtime**
ReLU	0.0020	0.0018	721
Softplus	0.0021	0.0014	785
Sigmoid	0.0018	0.0011	1085
Tanh	0.0020	0.0011	1110

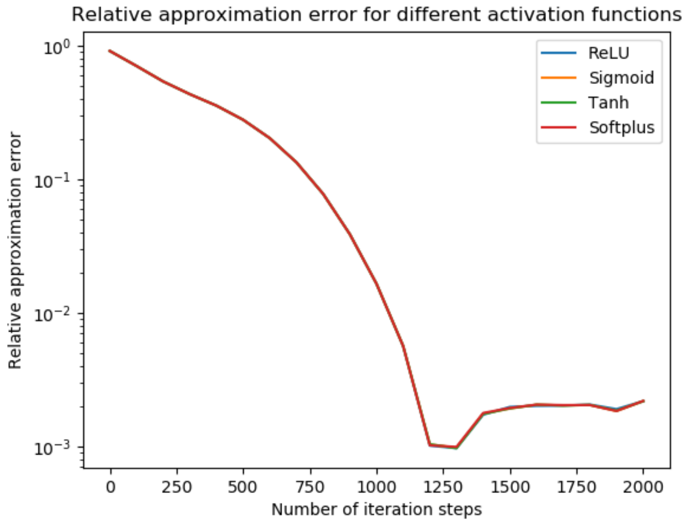


Figure 5.11: A plot of the mean relative approximation error when solving the HJB equation as a function of number of iteration steps. It is solved using the deep BSDE method for five independent runs using four different activation functions. The different graphs are difficult to identify as they are superimposed.

Table 5.8: Numerical results after solving the HJB equation with values in **Table 5.4** for five independent runs using four different activation functions. The runtime is given for one of the runs in seconds.

Activation function	**Mean of rel. approx. error**	**Standard deviation of rel. approx. error**	**Runtime**
ReLU	0.0022	0.0002	639
Softplus	0.0022	0.0002	546
Sigmoid	0.0022	0.0002	694
Tanh	0.0022	0.0002	551

5.3.2 Batch Size

In all machine learning problems, it is essential that the model is provided with sufficient amount of training data to be able to give the best available representation of the data. When the model is unable to represent the data well because of limited training data, it is called underfitting. Overfitting can also occur when the model captures noise in the data along with the underlying pattern. It is therefore interesting to see how the amount of training data affects the numerical results. The batch size denotes the number of input paths that are sampled and used for training at each iteration step. **Fig. 5.12** and **Fig.**

5.13 show how the relative approximation error and the loss function change for different batch sizes when solving the Allen-Cahn equation, respectively. A plot of the loss function after 4000 iterations against different batch sizes is presented in **Fig. 5.14**. The numerical results after 4000 iterations can be found in **Table 5.9**.

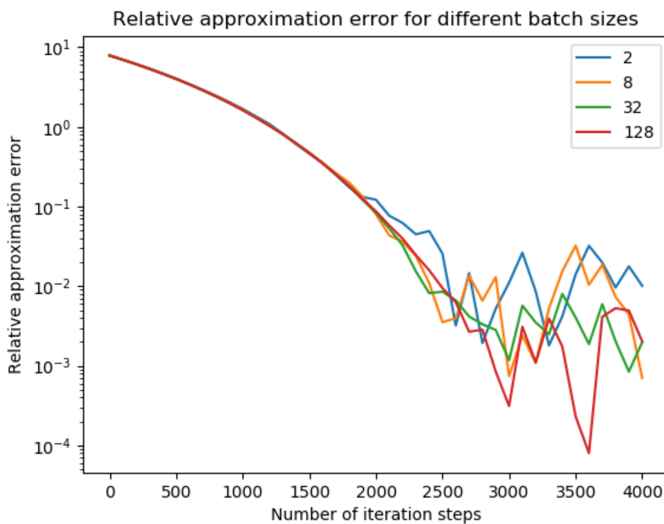


Figure 5.12: A plot of the relative approximation error when solving the Allen-Cahn equation as a function of number of iteration steps. It is solved using the deep BSDE method with different batch sizes.

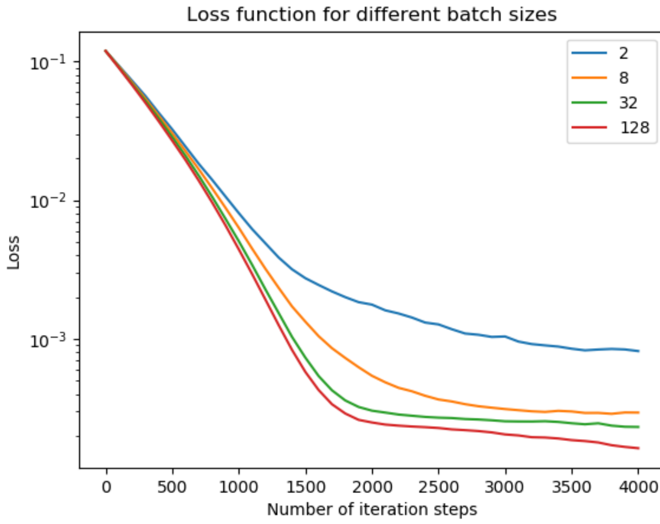


Figure 5.13: A plot of the loss function when solving the Allen-Cahn equation as a function of number of iteration steps. It is solved using the deep BSDE method with different batch sizes.

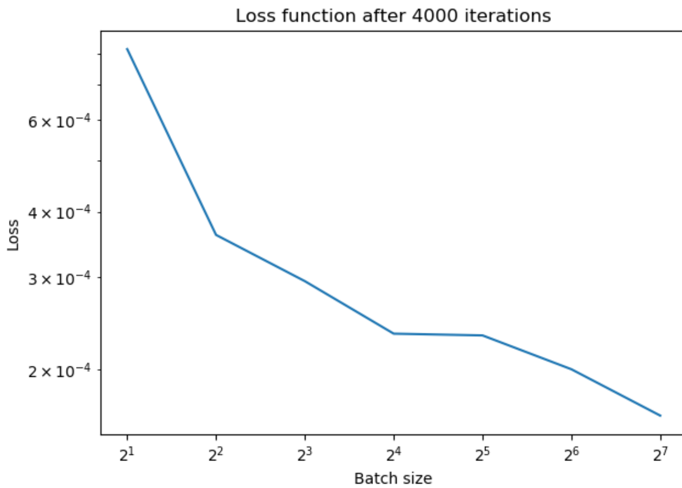


Figure 5.14: A plot of the loss function after 4000 iterations for solving the Allen-Cahn equation as a function of batch size.

Table 5.9: Numerical results after solving the Allen-Cahn equation for different batch sizes. The rest of the model variables are set to the values in **Table 5.1**. The runtime is given in seconds.

Batch size	Rel. approx. error	Loss	Runtime
2	0.0101	0.000818	720
4	0.0012	0.000361	809
8	0.0007	0.000295	660
16	0.0119	0.000234	865
32	0.0020	0.000231	813
64	0.0007	0.000200	721
128	0.0020	0.000163	1103

5.3.3 Temporal Discretization Steps

The deep BSDE method aims to approximate the solution of a semilinear parabolic PDE at time $t = 0$ given a terminal condition at time $t = T$. The BSDE is discretized with N temporal discretization steps between 0 and T. The accuracy of traditional numerical methods, like for example the finite difference method, depends on the fineness of the discretization grid. The relative approximation error as a function of iteration steps when solving the Allen-Cahn equation using the deep BSDE method for different number of temporal discretization steps is shown in **Fig. 5.15**. **Fig. 5.16** shows the relative approximation error after 4000 iterations against number of temporal discretization steps. A plot of the loss function against number of iteration steps for different number of temporal discretization steps, and the loss after 4000 iterations against number of temporal discretization steps are shown in **Fig. 5.17**, and **Fig. 5.18**, respectively. Numerical results after solving the Allen-Cahn equation for different number of temporal discretization steps are presented in **Table 5.10**.

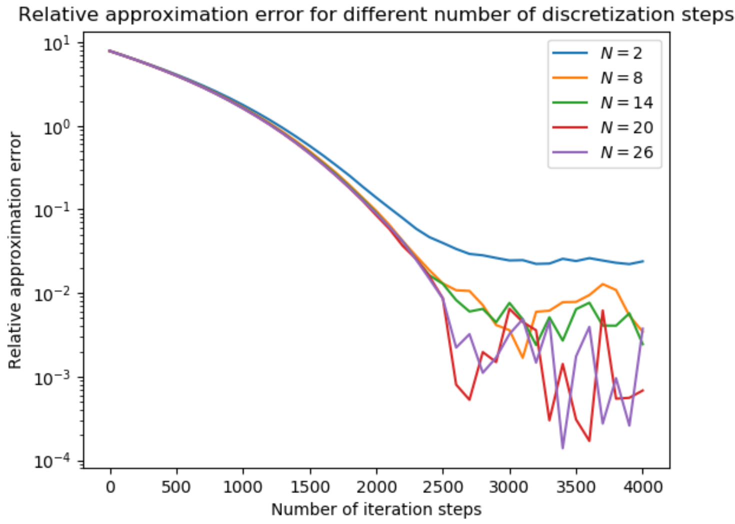


Figure 5.15: A plot of the relative approximation error when solving the Allen-Cahn equation as a function of number of iteration steps. It is solved using the deep BSDE method for different number of temporal discretization steps.

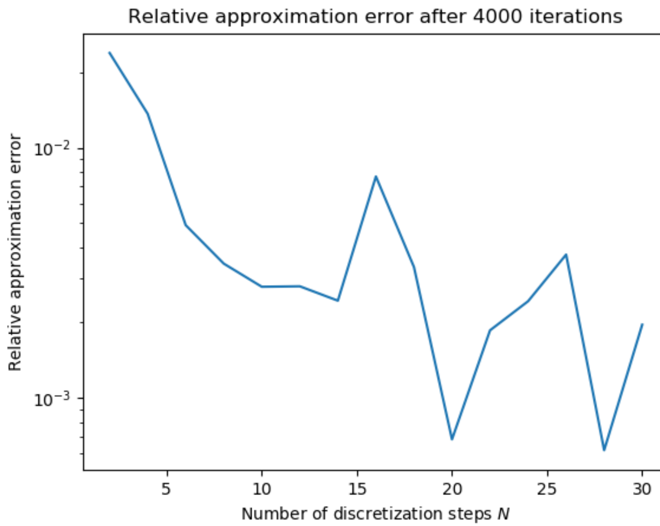


Figure 5.16: A plot of the relative approximation error after 4000 iteration steps for solving the Allen-Cahn equation as a function of number of temporal discretization steps.

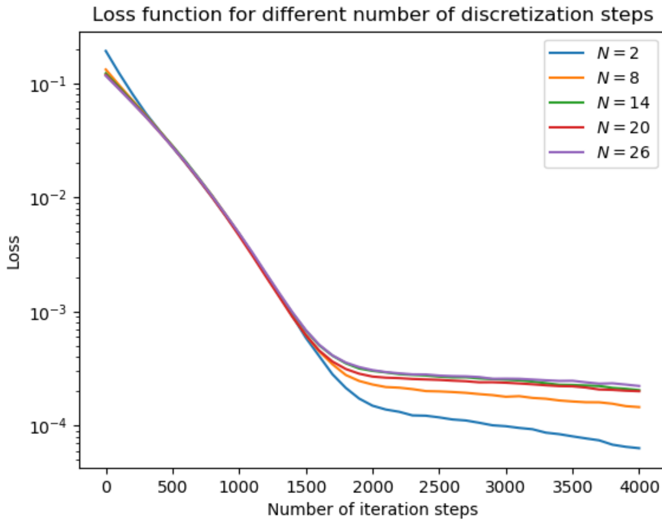


Figure 5.17: A plot of the loss function when solving the Allen-Cahn equation as a function of number of iteration steps. It is solved using the deep BSDE method for different number of temporal discretization steps.

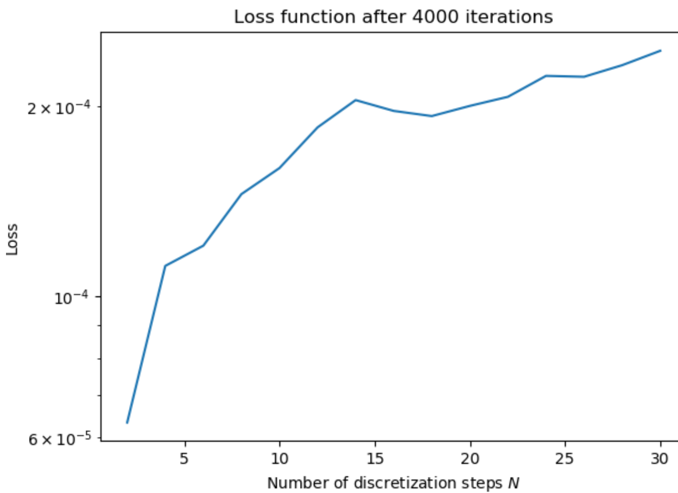


Figure 5.18: A plot of the loss function after 4000 iterations for solving the Allen-Cahn equation as a function of number of temporal discretization steps.

Table 5.10: Numerical results after solving the Allen-Cahn equation for different number of discretization steps. The rest of the model variables are set to the values in **Table 5.1**. The runtime is given in seconds.

Number of discretization steps	**Rel. approx. error**	**Loss**	**Runtime**
2	0.0239	0.000063	92
4	0.0137	0.000112	113
6	0.0049	0.000120	194
8	0.0034	0.000145	239
10	0.0028	0.000160	342
12	0.0028	0.000185	493
14	0.0024	0.000204	422
16	0.0077	0.000196	474
18	0.0033	0.000193	956
20	0.0007	0.000200	721
22	0.0019	0.000207	853
24	0.0024	0.000223	1093
26	0.0037	0.000222	971
28	0.0006	0.000232	2083
30	0.0020	0.000244	2430

Chapter 6

Discussion

The obtained numerical results are discussed in this chapter. Further, the convergence of the deep BSDE method is discussed based on the analysis conducted by Han and Long in (Han and Long, 2018).

6.1 On Numerical Results

The obtained results in Chapter 5 for the Allen-Cahn equation and the HJB equation correspond well with the presented results in (Han et al., 2018). In (Han et al., 2018) the deep BSDE method achieved a relative error of 0.30% after 4000 iteration steps for the Allen-Cahn equation. In Section 5.1, it achieved a relative approximation error of 0.20% after 4000 iteration steps using the same variable values as in (Han et al., 2018). However, when the same experiment was conducted with five new random seeds, the relative approximation error increased to 0.42%. Since the two separate results are quite accurate, it is reasonable to believe that the method is correctly implemented and that the deviations in the results can be explained as a consequence of the randomization included in the model. However, this deviation is relatively small and it is evident that the method works for solving the high-dimensional PDE since the relative approximation error and the loss function converge to an acceptable accuracy (as seen in **Fig 5.1, 5.2** and **5.4**). The runtime for one of the independent runs of 4000 iterations was 721 seconds, which is quite efficient. As a comparison, the branching-diffusion method had a runtime of 1361 seconds.

In **Fig 5.3** the approximated initial value for the Allen-Cahn equation against number of iteration steps is presented. From the figure one can see that the method converges towards a solution, and after about 2500 iterations the standard deviation becomes significantly smaller. The same change can be seen for the loss function in **Fig. 5.4**. At this point, the method has reached a relative approximation error of 0.078. After about 3000 iterations, there are oscillations in the relative approximation error in both **Fig 5.1** and **5.2**. The oscillations occur when the relative approximation errors have gone below 10^{-2}, and they are contained within the interval $[10^{-3}, 10^{-2}]$. This indicates that the method has become slightly unstable, and will not converge further with more iterations. However, since

the solution obtained by the branching-diffusion method only contains three significant digits, it would be futile to iterate further after these three digits are obtained by the deep BSDE method. **Fig 5.5** also shows the high performance of the deep BSDE method, since the approximated time evolution is visibly overlapping with the curve provided using the branching-diffusion method.

The results obtained for the HJB equation are more consistent with regard to the two different experiments using different sets of random seeds in each. The relative approximation errors are slightly higher than in (Han et al., 2018) after 2000 iteration steps, with 0.22% and 0.23% against 0.17%. Again, the deviations are relatively small and most likely due to the randomization. The relative approximation error and the loss function also converge to an acceptable accuracy as seen in **Fig 5.6, 5.7** and **5.9**. The runtime for one of the independent runs of 2000 iterations was 639 seconds, which is again quite efficient.

In **Fig 5.8** the approximated initial value for the HJB equation against number of iteration steps is presented. From the figure one can see that the method converges towards a solution, and after around 1000 iterations the curve flattens out. The same flattening can be seen for the loss function in **Fig. 5.9**. The relative approximation error is 0.016 after 1000 iterations. After around 1250 iterations, there is a clear cusp in the relative approximation error in both **Fig 5.6** and **5.7**. The cusp is followed by slight oscillations and occur when the relative approximation errors have reached 10^{-3}. The oscillations are contained within the interval $[10^{-3}, 3 \cdot 10^{-3}]$. Again, this is an indication of instability in the method, but since the solution obtained by Monte Carlo simulations only contains five significant digits, it would be futile to iterate further after these digits are obtained by the deep BSDE method.

6.2 On Model Features

In Section 5.3, different model features are tested to see how they impact the model results. **Fig. 5.10** and **Fig. 5.11** show the relative approximation error when solving the Allen-Cahn equation and the HJB equation, respectively, for different activation functions. From the figures, it is clear that the choice of activation function has minimal significance to the model, as they all achieve relatively similar accuracy after the same number of iteration steps. However, from **Table 5.7** and **Table 5.8** one can see that they have different runtimes. There is only a slight difference, and it is most visible in **Table 5.7** when solving the Allen-Cahn equation where the runtime is longer overall. Here, the advantage of the cheap computational cost of the ReLU becomes apparent.

The relative approximation error for different batch sizes is shown in **Fig. 5.12**. Generally, the approximation error is smaller for higher batch sizes, which is what would be expected since the model is trained on more sample paths and should have a better representation of the expected value of the stochastic processes. This is the same logic as in Monte Carlo methods. Because of the oscillations in the last iteration steps, the order of the final approximation errors achieved for the different batch sizes is somewhat random, as seen in **Table 5.9**. Unexpectedly, the method achieves a better approximation for a batch size of 4 than 128, proving that the batch size does not affect the method as much as it might in other deep learning problems. This is most likely due to the fact that the sample paths represent the same process. In other deep learning problems, the training data might

represent a complex function, and more data is needed to cover enough of the domain and the range of the function to capture the trends. In **Fig. 5.13** and **Fig. 5.14** one can see that the higher the batch size is, the more the model manages to minimize the loss function. Again, this is logical as the model then has a better representation of the expected value of the stochastic processes.

The number of temporal discretization steps clearly affects the runtime of the method, as seen in **Table 5.10**, where the runtime for $N = 2$ is 92 seconds, and for $N = 30$ is 2430 seconds. This is obviously because the ANN becomes significantly larger for higher number of discretization steps. Generally, the approximation error is smaller for more discretization steps as seen in **Fig. 5.15** and **Fig. 5.16**, which complies with traditional numerical methods. Because of the oscillations in the last iteration steps, the order of the final approximation errors achieved for the different number of discretization steps is somewhat random. In **Fig. 5.17** and **Fig. 5.18** the loss function is shown for different number of discretization steps, and here one can see that the model minimizes the loss function more for fewer number of discretization steps. One reason for this could be because of the size of the ANN, and that for many discretization steps there are more parameters to optimize against the loss function.

6.3 On Convergence

The convergence of the deep BSDE method has been investigated in (Han and Long, 2018), and Han and Long have formulated two main theorems that will be rendered and explained in this section. The proofs are quite extensive and will not be included in this thesis, however, they can be found in (Han and Long, 2018). In their paper, they extend the method to include coupled BSDEs and a wider class of quasilinear parabolic PDEs. This extension will no be considered in this thesis, and simplifications of their theorems will therefore apply.

The deep BSDE method with notation is revisited to understand the theorems. As mentioned, only decoupled forward BSDEs (the variables μ and σ are independent of Y) is considered, and the following is taken from Subsection 2.2.2,

$$\begin{aligned} X_t &= \xi + \int_0^t \mu(s, X_s)ds + \int_0^t \sigma(s, X_s)dW_s, \\ Y_t &= h(X_T) + \int_t^T f(s, X_s, Y_s, Z_s)ds - \int_t^T (Z_s)^{\mathrm{T}} dW_s, \end{aligned}$$

where $X_t \in \mathbb{R}^m$, $Y_t \in \mathbb{R}$, $Z_t \in \mathbb{R}^d$. The BSDEs are written forwardly using the Euler scheme from Subsection 2.2.2, giving the set of equations,

$$\begin{aligned} & X_0^\pi = \xi, \quad Y_0^\pi = \eta_0^\pi(\xi), \\ & X_{t_{n+1}}^\pi = X_{t_n}^\pi + \mu(t_n, X_{t_n}^\pi)\Delta t_n + \sigma(t_n, X_{t_n}^\pi)\Delta W_n, \\ & Z_{t_n}^\pi = \phi_n^\pi(X_{t_n}^\pi), \\ & Y_{t_{n+1}}^\pi = Y_{t_n}^\pi - f(t_n, X_{t_n}^\pi, Y_{t_n}^\pi, Z_{t_n}^\pi)\Delta t_n + (Z_{t_n}^\pi)^{\mathrm{T}}\Delta W_n. \end{aligned}$$

The discretization uses a partition of the time interval $[0,T]$, $\pi : 0 = t_0 < t_1 < \cdots < t_N = T$, with equidistant time steps $\Delta t_n = T/N = l$, where each time step is given by $t_n = nl$. The symbol π is used in superscript to denote a value approximation based on the time partition interval π.

The goal is to find appropriate functions $\eta_0^\pi : \mathbb{R}^m \to \mathbb{R}$ and $\phi_n^\pi(X_{t_n}^\pi) : \mathbb{R}^m \times \mathbb{R} \to \mathbb{R}^d$ for $n = 0, 1, \ldots, N-1$ through deep learning such that $\eta_0^\pi(\xi)$ and $\phi_n^\pi(X_{t_n}^\pi)$ can approximate Y_0 and Z_{t_n}, respectively. To achieve this, a stochastic optimizer is employed. In the numerical experiments in Chapter 5, the Adam optimizer was used. A mathematical formulation of the stochastic optimization problem is given as,

$$\inf_{\eta_0^\pi \in \mathcal{N}_0', \phi_n^\pi \in \mathcal{N}_n} F(\eta_0^\pi, \phi_0^\pi, \ldots, \phi_{N-1}^\pi) = \mathbb{E}|h(X_T^\pi) - Y_T^\pi|^2, \tag{6.1}$$

where $\mathcal{N}_0', \mathcal{N}_n$, $n = 0, \ldots, N-1$ are parametric function spaces generated by the ANN. The goal is to minimize the loss function (right-hand side of (6.1)) with respect to the network parameters approximating η_0^π and ϕ_n^π for $n = 0, 1, \ldots, N-1$.

6.3.1 A Posteriori Error Estimation

A posteriori error estimator uses the approximated solution itself to derive an estimate of the solution error. The following is a posteriori error estimation of the deep BSDE method, taken from (Han and Long, 2018).

Theorem 3 (A posteriori estimates)**.** *Under some assumptions, there exists a constant C, independent of l, d, and m, such that for sufficiently small l,*

$$\begin{aligned}&\sup_{t\in[0,T]} \left(\mathbb{E}|X_t - \hat{X}_t^\pi|^2 + \mathbb{E}|Y_t - \hat{Y}_t^\pi|^2\right) + \int_0^T \mathbb{E}|Z_t - \hat{Z}_t^\pi|^2 dt \\ \leq &C\left[l + \mathbb{E}|h(X_T^\pi) - Y_T^\pi|^2\right]\end{aligned} \tag{6.2}$$

where $\hat{X}_t^\pi = X_{t_n}^\pi$, $\hat{Y}_t^\pi = Y_{t_n}^\pi$, $\hat{Z}_t^\pi = Z_{t_n}^\pi$ for $t \in [t_n, t_{n+1})$.

The left-hand side of equation (6.2) in the theorem represents the simulation error. The theorem therefore states that the simulation error is bounded. The upper bound is given by the value of the objective function presented in equation (6.1). If the objective function is optimized to be close to zero, then by **Thm. 3** the approximated solution obtained by the deep BSDE method is close to the actual solution. This is the goal of the deep learning process. This means that the accuracy of numerical solution is effectively indicated by the value of objective function. The proof of **Thm. 3** can be found in (Han and Long, 2018).

6.3.2 Upper Bound of Optimal Loss

The second result in (Han and Long, 2018) is a theorem providing an upper bound of the optimal loss of the deep BSDE method. The theorem states that the objective function can be optimized to be close to zero, and thereby fulfilling the demand in **Thm. 3**, and attaining an approximated solution close to the actual solution. The theorem is rendered here:

Theorem 4 (Upper bound of optimal loss)**.** *Under some assumptions, there exists a constant C, independent of l, d, and m, such that for sufficiently small l,*

$$\begin{aligned} &\inf_{\eta_0^\pi \in \mathcal{N}_0', \phi_n^\pi \in \mathcal{N}_n} \mathbb{E}|h(X_t^\pi) - Y_t^\pi|^2 \\ \leq & C \left\{ l + \inf_{\eta_0^\pi \in \mathcal{N}_0', \phi_n^\pi \in \mathcal{N}_n} \mathbb{E}|Y_0 - \eta_0^\pi(\xi)|^2 + \sum_{n=0}^{N-1} \mathbb{E}|\mathbb{E}[\tilde{Z}_{t_n}|X_{t_n}^\pi] - \phi_n^\pi(X_{t_n}^\pi)|^2 l \right\} \end{aligned} \tag{6.3}$$

where $\tilde{Z}_{t_n} = l^{-1}\mathbb{E}[\int_{t_n}^{t_{n+1}} Z_t dt | \mathcal{F}_{t_n}]$.

The theorem relates the infimum of the objective function (left-hand side of equation (6.3)) to the expressive power of ANNs. It states that the optimal value of the objective function is small if the approximation capability of the parametric function spaces $\mathcal{N}_0'$, $\mathcal{N}_n$, $n = 0, \ldots, N-1$, is high. The approximation capability of ANNs was discussed in Subsection 2.1.5, and sufficiently large ANNs with 2 hidden layers are so called universal approximators. **Thm. 1** also presented a universal approximation theorem for width-bounded ANNs. Because of the universal approximation property, there exists ANNs with suitable network parameters such that the obtained numerical solution is approximately accurate. The proof of **Thm. 4** can be found in (Han and Long, 2018).

Chapter 7

Conclusion

The goal of the thesis was to implement the deep BSDE method, and test it on two different PDEs. In the thesis, necessary background theory is presented to understand the structure of ANNs, deep learning, and also the reformulation from semilinear parabolic PDEs to BSDEs. Implementation details are given, and numerical results presented and discussed. To conclude, a summary of the methodology and the numerical results is given in this chapter, and lastly, future work is discussed.

7.1 Summary

The presented deep BSDE method solves terminal value problems for semilinear parabolic PDEs using deep learning. By Itô's formula, a semilinear parabolic PDE can be written equivalently as a BSDE for a given stochastic process, in the sense that the solution of the BSDE also solves the semilinear parabolic PDE. The connected solutions are given by the nonlinear Feynman-Kac formula. The BSDEs are temporally discretized using the Euler scheme, and the unknown coefficient (spatial gradient of the solution) is approximated by an MLP at each time step. The input of the ANN is then sampled paths of the stochastic process, and the output is the approximated value of the solution at terminal time. The loss function of the network is given by the difference between the output and the terminal condition. The goal of the learning is to minimize this loss function with respect to the network parameters that are included in the MLPs at each discretization step.

The deep BSDE method is implemented to solve the 100-dimensional Allen-Cahn equation and the 100-dimensional HJB equation, both with 20 temporal discretization steps. The MLPs consist of two 110-dimensional hidden layers, equipped with the ReLU as activation function. The ANN is trained on 64 sampled paths and tested against 256 Monte Carlo simulations. The training is performed using the Adam optimizer, which is an SGD type algorithm, to minimize the loss function. The method achieved a relative approximation error of 0.20% after 4000 iteration steps for the Allen-Cahn equation, and 0.22% after 2000 iteration steps for the HJB equation. The runtime of a single run when solving the Allen-Cahn and the HJB equation was 721 and 639 seconds, respectively.

The model performed well for several activation functions, and the conclusion would be to choose the most computationally efficient one to make use of this advantage. After testing the method on different batch sizes, the method proved not to be as prone to underfitting for small batch sizes as one could expect from a deep learning perspective. The accuracy does, however, generally improve for larger batch sizes. The accuracy also improves for more discretization steps, but also leads to a higher runtime. The choice of the number of discretization steps is therefore controlled by a tradeoff between runtime and accuracy.

The numerical results for the deep BSDE method are great in terms of both accuracy and computational cost. Analysis of PDEs will continue to be important, and methods to minimize the computational cost of the analysis are continuously subject to research. Traditional numerical methods suffer the curse of dimensionality, and can therefore become too computationally expensive or give unreliable results in higher dimensions. The deep BSDE method may therefore be a preferred option to achieve the most correct representation of real world phenomenons, by solving PDEs of higher dimensions than previously possible. It also opens the possibility of solving more complex and demanding problems in various areas, such as economics, finance, operational research, and physics. For example in operational research, one can consider many more participating entities directly, without having to make ad hoc assumptions. The method could also potentially relieve computational capacity, since today a lot of computing power is used to solve PDEs numerically.

7.2 Future Work

A posteriori error estimates and an upper bound of optimal loss was presented in this thesis. It follows that the numerical solution obtained by the deep BSDE method is analytically proved to be approximately accurate, for which the numerical results comply. However, there does not exist a complete theoretical framework for explaining the effectiveness of ANNs. This is currently being eagerly investigated. More numerical experiments on the different features of the ANN part of the method would be interesting to delve further into with regard to how it impacts the results. The method can also be further optimized through hyperparameter tuning of the ANN. It would also be valuable to test the method on more semilinear parabolic PDEs with varying characteristics, with focus on the affect on computational cost. (Han and Long, 2018) extends the deep BSDE method to solve coupled BSDEs and a wider class of quasilinear parabolic PDEs. Future work could also include exploration of the possibility of using deep learning to solve other types of PDEs inspired by the mindset in the deep BSDE method.

Bibliography

Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., et al., 2016. Tensorflow: A system for large-scale machine learning, in: 12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16), pp. 265–283.

Bartels, S., 2015. Numerical methods for nonlinear partial differential equations. volume 47. Springer.

Bellman, R., 1957. Dynamic programming. Princeton Univ Press .

Benner, P., Stoll, M., 2013. Optimal control for allen-cahn equations enhanced by model predictive control, in: 1st IFAC Workshop on Control of Systems Governed by Partial Differential Equations-CPDE 2013, pp. 139–143.

Chassagneux, J.F., Richou, A., et al., 2016. Numerical simulation of quadratic bsdes. The Annals of Applied Probability 26, 262–304.

Darbon, J., Osher, S., 2016. Algorithms for overcoming the curse of dimensionality for certain hamilton–jacobi equations arising in control theory and elsewhere. Research in the Mathematical Sciences 3, 19.

Eldan, R., Shamir, O., 2016. The power of depth for feedforward neural networks, in: Conference on learning theory, pp. 907–940.

François-Lavet, V., Henderson, P., Islam, R., Bellemare, M.G., Pineau, J., et al., 2018. An introduction to deep reinforcement learning. Foundations and Trends® in Machine Learning 11, 219–354.

Goodfellow, I., Bengio, Y., Courville, A., 2016. Deep Learning. MIT Press. `http://www.deeplearningbook.org`.

Grimmett, G., Grimmett, G.R., Stirzaker, D., et al., 2001. Probability and random processes. Oxford university press.

Han, J., Jentzen, A., Weinan, E., 2018. Solving high-dimensional partial differential equations using deep learning. Proceedings of the National Academy of Sciences 115, 8505–8510.

Han, J., Long, J., 2018. Convergence of the deep bsde method for coupled fbsdes. arXiv preprint arXiv:1811.01165 .

Han, J., et al., 2016. Deep learning approximation for stochastic control problems. arXiv preprint arXiv:1611.07422 .

Henry-Labordere, P., Tan, X., Touzi, N., 2014. A numerical algorithm for a class of bsdes via the branching process. Stochastic Processes and their Applications 124, 1112–1140.

Hutzenthaler, M., Jentzen, A., Kruse, T., et al., 2017. On multilevel picard numerical approximations for high-dimensional nonlinear parabolic partial differential equations and high-dimensional nonlinear backward stochastic differential equations. arXiv preprint arXiv:1708.03223 .

IDC, 2019. Worldwide spending on artificial intelligence systems will be nearly $98 billion in 2023, according to new idc spending guide. https://www.idc.com/getdoc.jsp?containerId=prUS45481219.

Ioffe, S., Szegedy, C., 2015. Batch normalization: Accelerating deep network training by reducing internal covariate shift. arXiv preprint arXiv:1502.03167 .

Karasözen, B., Uzunca, M., Sariaydin-Filibelioğlu, A., Yücel, H., 2018. Energy stable discontinuous galerkin finite element method for the allen–cahn equation. International Journal of Computational Methods 15, 1850013.

Kingma, D.P., Ba, J., 2014. Adam: A method for stochastic optimization. arXiv preprint arXiv:1412.6980 .

Kostorz, G., 1995. Metals and alloys: Phase separation and defect agglomeration, in: Modern Aspects of Small-Angle Scattering. Springer, pp. 255–266.

Lehmann, D., n.d. Chapter 17: The feynman-kac formula. http://hsrm-mathematik.de/WS201516/master/option-pricing/Feynman-Kac-Formula.pdf.

Lu, Q., Zhang, X., 2016. A mini-course on stochastic control .

Lu, Z., Pu, H., Wang, F., Hu, Z., Wang, L., 2017. The expressive power of neural networks: A view from the width, in: Advances in neural information processing systems, pp. 6231–6239.

Marr, B., 2018a. The amazing ways tesla is using artificial intelligence and big data. https://www.forbes.com/sites/bernardmarr/2018/01/08/the-amazing-ways-tesla-is-using-artificial-intelligence-and-big-data/#60872a314270.

Marr, B., 2018b. The most amazing artificial intelligence milestones so far. https://www.forbes.com/sites/bernardmarr/2018/12/31/the-most-amazing-artificial-intelligence-milestones-so-far/#19fc24727753.

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A.A., Veness, J., Bellemare, M.G., Graves, A., Riedmiller, M., Fidjeland, A.K., Ostrovski, G., et al., 2015. Human-level control through deep reinforcement learning. Nature 518, 529–533.

Nwankpa, C., Ijomah, W., Gachagan, A., Marshall, S., 2018. Activation functions: Comparison of trends in practice and research for deep learning. arXiv preprint arXiv:1811.03378 .

Pardoux, E., Peng, S., 1992. Backward stochastic differential equations and quasilinear parabolic partial differential equations, in: Stochastic partial differential equations and their applications. Springer, pp. 200–217.

Qureshi, A.S., 2017. Wind power prediction using deep neural network based meta regression and transfer learning. Applied Soft Computing 58, 742–755.

Russell, S.J., Norvig, P., 2016. Artificial intelligence: a modern approach. Malaysia; Pearson Education Limited.

SiriTeam, 2017. Hey siri: An on-device dnn-powered voice trigger for apple's personal assistant. https://machinelearning.apple.com/2017/10/01/hey-siri.html.

Ville, J., 1939. Etude critique de la notion de collectif. Bull. Amer. Math. Soc 45, 824.

Weinan, E., Han, J., Jentzen, A., 2017. Deep learning-based numerical methods for high-dimensional parabolic partial differential equations and backward stochastic differential equations. Communications in Mathematics and Statistics 5, 349–380.

Wired, 2016. This ai software can tell if you're at risk from cancer before symptoms appear. https://www.wired.co.uk/article/cancer-risk-ai-mammograms.

Yong, J., Zhou, X.Y., 1999. Stochastic controls: Hamiltonian systems and HJB equations. volume 43. Springer Science & Business Media.

Appendix

A Python code for learning the sine function

```
#Import necessary libraries
import matplotlib.pyplot as plt
import tensorflow as tf
import numpy as np

#Define placeholders
x = tf.placeholder('float', [None, 1])
y = tf.placeholder('float', [None, 1])

#Create neural network graph
h1 = tf.contrib.layers.fully_connected(x, 100, activation_fn=tf.nn.relu)
h2 = tf.contrib.layers.fully_connected(h1, 100, activation_fn=tf.nn.relu)
result = tf.contrib.layers.fully_connected(h2, 1, activation_fn=None)

#Define loss function
loss = tf.nn.l2_loss(result - y)

#Define an optimizer to minimize the loss
optimizer = tf.train.AdamOptimizer().minimize(loss)

#Initialize variables
init = tf.global_variables_initializer()

#Run 10 000 sessions
sess = tf.Session()
sess.run(init)
for i in range(10000):
    xtrain = np.random.rand(100) * 10
    ytrain = np.sin(xtrain)
    sess.run(optimizer, feed_dict={x: xtrain[:, None], y: ytrain[:, None
    ]})

#Get predictions on test data
xtest = np.random.rand(50) * 10
ypred = sess.run(result, feed_dict={x: xtest[:, None]})

#Plot sine function together with predictions
plt.scatter(xtest, ypred, marker='o', label='Predicted values: $f(x)$')
plt.plot(np.arange(0,10,1/10), np.sin(np.arange(0,10,1/10)), color='green'
    , label='$f^*(x)=sin(x)$')
plt.title('Approximation of sine function using neural network')
```

```python
plt.xlabel('$x$')
plt.legend(numpoints=1)
plt.show()
```

Listing A: Python code for creating a neural network that learns the sine function using TensorFlow. The result is shown in **Fig. 2.5** in Subsection 2.1.3.

A placeholder is a data structure in TensorFlow that can store vectors and matrices. This has to be defined in order to create the neural network graph. In the example above, the graph consists of two hidden layers, `h1` and `h2`, that are fully connected from the input layer `x` to the output layer `result`. The ReLU is used as the activation function in the hidden layers. After the graph is created, a loss function and optimizer is defined. In the TensorFlow framework, all computations on the graph are done in a session. Firstly, the variables are initialized and then 10 000 sessions are run, each training the network on 100 data points from 0 to 10. After the network has been trained, it can perform predictions on new data.

B Python code for solving the Allen-Cahn equation using the deep BSDE method

```python
#Import necessary libraries
import time
import numpy as np
import tensorflow as tf
import matplotlib.pyplot as plt
from scipy.stats import multivariate_normal as normal
from tensorflow import constant_initializer as const_init
from tensorflow import random_normal_initializer as norm_init
from tensorflow import random_uniform_initializer as unif_init
from tensorflow.python.ops import control_flow_ops
from tensorflow.python.training.moving_averages import
    assign_moving_average

class SolveAllenCahn:

    def __init__(self, sess):
        self.sess = sess
        #Define variables
        self.d = 100
        self.dh = self.d + 10
        self.T = 0.3
        self.N = 20
        self.l = (self.T + 0.0) / self.N
        self.time_stamp = np.arange(0, self.N) * self.l
        self.batch_size = 64
        self.M = 256
        self.n_maxstep = 4000
        self.n_displaystep = 100
        self.learning_rate = 5e-4
        self.extra_train_ops = []

    def build(self):
        start_time = time.time()
        #Define tensor types
```

```
34         self.X = tf.placeholder(dtype=tf.float64, shape=[None, self.d,
    self.N+1], name='X')
35         self.dW = tf.placeholder(dtype=tf.float64, shape=[None, self.d,
    self.N], name='dW')
36         self.Y0 = tf.Variable(initial_value=tf.random_uniform(shape=[1],
    dtype=tf.float64, minval=0.3, maxval=0.6))
37         self.Z0 = tf.Variable(initial_value=tf.random_uniform(shape=[1,
    self.d], dtype=tf.float64, minval=-0.1, maxval=0.1))
38         self.is_training = tf.placeholder(dtype=tf.bool)
39         #Create neural network graph
40         self.allones = tf.ones(shape=tf.stack([tf.shape(self.dW)[0], 1]),
    dtype=tf.float64)
41         Y = self.allones * self.Y0
42         Z = tf.matmul(self.allones, self.Z0)
43         with tf.variable_scope('forward_connections'):
44             for n in range(0, self.N-1):
45                 Y = Y - self.f(self.time_stamp[n], self.X[:, :, n], Y, Z)
    * self.l
46                 Y = Y + tf.reduce_sum(Z * self.dW[:, :, n], axis=1,
    keepdims=True)
47                 Z = self.subnetwork(self.X[:, :, n+1], str(n+1)) / self.d
48             #Terminal time
49             Y = Y - self.f(self.time_stamp[self.N-1], self.X[:, :, self.N
    -1], Y, Z) * self.l
50             Y = Y + tf.reduce_sum(Z * self.dW[:, :, self.N-1], axis=1,
    keepdims=True)
51             #Loss calculation
52             delta = Y - self.h(self.T, self.X[:, :, self.N])
53             self.clipped_delta = tf.clip_by_value(delta, clip_value_min
    =-50.0, clip_value_max=50.0)
54             self.loss = tf.reduce_mean(self.clipped_delta**2)
55         self.time_build = time.time() - start_time
56 
57     def f(self, t, X, Y, Z):
58         #Nonlinear term
59         return Y - tf.pow(Y, 3)
60 
61     def h(self, t, X):
62         #Terminal condition
63         return 1 / (2 + 0.4 * (tf.norm(X, ord=2, axis=1, keepdims=True))
    **2)
64 
65     def subnetwork(self, x, name):
66         #Create a subnetwork
67         with tf.variable_scope(name):
68             x_layer = self.batch_norm(x, name='initial')
69             layer1 = self.add_layer(x_layer, dim=self.dh, name='layer1')
70             layer2 = self.add_layer(layer1, dim=self.dh, name='layer2')
71             z_layer = self.add_layer(layer2, dim=self.d, activation=None,
    name='result')
72 
73         return z_layer
74 
75     def add_layer(self, input_, dim, activation=tf.nn.relu, name='linear')
    :
76         #Create a single layer
77         with tf.variable_scope(name):
```

```
78            #Input function
79            shape = input_.get_shape().as_list()
80            w = tf.get_variable('weights', shape=[shape[1], dim], dtype=tf
    .float64, initializer=norm_init(mean=0.0, stddev=5.0/np.sqrt(shape[1]+
    dim)))
81            layer = tf.matmul(input_, w)
82            #Batch normalization
83            layer_bn = self.batch_norm(layer, name='normalization')
84            #Activation
85            if activation != None:
86
87                return activation(layer_bn)
88            else:
89
90                return layer_bn
91
92    def batch_norm(self, x, name):
93        #Perform batch normalization
94        with tf.variable_scope(name):
95            #Define tensor types
96            params_shape = [x.get_shape()[-1]]
97            beta = tf.get_variable('beta', shape=params_shape, dtype=tf.
    float64, initializer=norm_init(mean=0.0, stddev=0.1, dtype=tf.float64)
    )
98            gamma = tf.get_variable('gamma', shape=params_shape, dtype=tf.
    float64, initializer=unif_init(minval=0.1, maxval=0.5, dtype=tf.
    float64))
99            mv_mean = tf.get_variable('moving_mean', shape=params_shape,
    dtype=tf.float64, initializer=const_init(value=0.0, dtype=tf.float64),
     trainable=False)
100           mv_var = tf.get_variable('moving_variance', shape=params_shape
    , dtype=tf.float64, initializer=const_init(value=1.0, dtype=tf.float64
    ), trainable=False)
101           #Fix mean and variance of layer x
102           mean, variance = tf.nn.moments(x, axes=[0], name='moments')
103           self.extra_train_ops.append(assign_moving_average(mv_mean,
    mean, decay=0.99))
104           self.extra_train_ops.append(assign_moving_average(mv_var,
    variance, decay=0.99))
105           mean, variance = control_flow_ops.cond(self.is_training,
    lambda: (mean, variance), lambda: (mv_mean, mv_var))
106           y = tf.nn.batch_normalization(x, mean, variance, offset=beta,
    scale=gamma, variance_epsilon=1e-06)
107           y.set_shape(x.get_shape())
108
109           return y
110
111   def train(self):
112       start_time = time.time()
113       #Define training operations
114       self.global_step = tf.get_variable('global_step', shape=[], dtype=
    tf.int32, initializer=const_init(value=1), trainable=False)
115       grads = tf.gradients(self.loss, tf.trainable_variables())
116       optimizer = tf.train.AdamOptimizer(self.learning_rate)
117       train_ops = [optimizer.apply_gradients(zip(grads, tf.
    trainable_variables()), global_step=self.global_step)] + self.
    extra_train_ops
```

```
118         self.train_op = tf.group(*train_ops)
119         self.loss_history = []
120         self.init_history = []
121         self.runtime_history = []
122         dW_test, X_test = self.sample_path(self.M)
123         feed_dict_test = {self.dW: dW_test, self.X: X_test, self.
    is_training: False}
124         #Initialization
125         step = 1
126         self.sess.run(tf.global_variables_initializer())
127         temp_loss = self.sess.run(self.loss, feed_dict=feed_dict_test)
128         temp_init = self.Y0.eval()[0]
129         runtime = time.time()-start_time+self.time_build
130         self.loss_history.append(temp_loss)
131         self.init_history.append(temp_init)
132         self.runtime_history.append(runtime)
133         #Training
134         for _ in range(self.n_maxstep+1):
135             step = self.sess.run(self.global_step)
136             dW_train, X_train = self.sample_path(self.batch_size)
137             self.sess.run(self.train_op, feed_dict={self.dW: dW_train,
    self.X: X_train, self.is_training: True})
138             if step % self.n_displaystep == 0:
139                 temp_loss = self.sess.run(self.loss, feed_dict=
    feed_dict_test)
140                 temp_init = self.Y0.eval()[0]
141                 runtime = time.time()-start_time+self.time_build
142                 self.loss_history.append(temp_loss)
143                 self.init_history.append(temp_init)
144                 self.runtime_history.append(runtime)
145                 print('step: ', step, '\t loss: ', temp_loss, '\t Y0: ',
    temp_init, '\t runtime: ', runtime)
146             step += 1
147         end_time = time.time()
148 
149     def sample_path(self, n_sample):
150         #Create sample paths
151         dW_sample = np.zeros([n_sample, self.d, self.N])
152         X_sample = np.zeros([n_sample, self.d, self.N+1])
153         for i in range(self.N):
154             dW_sample[:, :, i] = np.reshape(normal.rvs(mean=np.zeros(self.
    d), cov=1, size=n_sample) * np.sqrt(self.l), (n_sample, self.d))
155             X_sample[:, :, i+1] = X_sample[:, :, i] + np.sqrt(2) *
    dW_sample[:, :, i]
156 
157         return dW_sample, X_sample
158 
159 def main():
160     tf.reset_default_graph()
161     with tf.Session() as sess:
162         tf.set_random_seed(1)
163         model = SolveAllenCahn(sess)
164         model.build()
165         model.train()
166         results = np.zeros((len(model.init_history), 4))
167         results[:, 0] = np.arange(len(model.init_history)) * model.
    n_displaystep
```

```
        results[:, 1] = model.loss_history
        results[:, 2] = model.init_history
        results[:, 3] = model.runtime_history
        mat = np.matrix(results)
        with open('Allen_Cahn_results_seed1.txt', 'w') as f:
            for line in mat:
                np.savetxt(f, line)

if __name__ == '__main__':
    np.random.seed(1)
    main()
```

Listing B: Python code for solving the Allen-Cahn equation with the deep BSDE method implemented in TensorFlow.

C Python code for solving the Hamilton-Jacobi-Bellman equation using the deep BSDE method

```
#Import necessary libraries
import time
import numpy as np
import tensorflow as tf
import matplotlib.pyplot as plt
from scipy.stats import multivariate_normal as normal
from tensorflow import constant_initializer as const_init
from tensorflow import random_normal_initializer as norm_init
from tensorflow import random_uniform_initializer as unif_init
from tensorflow.python.ops import control_flow_ops
from tensorflow.python.training.moving_averages import
    assign_moving_average

class SolveHJB:

    def __init__(self, sess):
        self.sess = sess
        #Define variables
        self.lambda_ = 1
        self.d = 100
        self.dh = self.d + 10
        self.T = 1
        self.N = 20
        self.l = (self.T + 0.0) / self.N
        self.time_stamp = np.arange(0, self.N) * self.l
        self.batch_size = 64
        self.M = 256
        self.n_maxstep = 2000
        self.n_displaystep = 100
        self.learning_rate = 1e-2
        self.extra_train_ops = []

    def build(self):
        start_time = time.time()
        #Define tensor types
        self.X = tf.placeholder(dtype=tf.float64, shape=[None, self.d,
    self.N+1], name='X')
```

```
36          self.dW = tf.placeholder(dtype=tf.float64, shape=[None, self.d,
    self.N], name='dW')
37          self.Y0 = tf.Variable(initial_value=tf.random_uniform(shape=[1],
    dtype=tf.float64, minval=0.3, maxval=0.6))
38          self.Z0 = tf.Variable(initial_value=tf.random_uniform(shape=[1,
    self.d], dtype=tf.float64, minval=-0.1, maxval=0.1))
39          self.is_training = tf.placeholder(dtype=tf.bool)
40          #Create neural network graph
41          self.allones = tf.ones(shape=tf.stack([tf.shape(self.dW)[0], 1]),
    dtype=tf.float64)
42          Y = self.allones * self.Y0
43          Z = tf.matmul(self.allones, self.Z0)
44          with tf.variable_scope('forward_connections'):
45              for n in range(0, self.N-1):
46                  Y = Y - self.f(self.time_stamp[n], self.X[:, :, n], Y, Z)
    * self.l
47                  Y = Y + tf.reduce_sum(Z * self.dW[:, :, n], axis=1,
    keepdims=True)
48                  Z = self.subnetwork(self.X[:, :, n+1], str(n+1)) / self.d
49              #Terminal time
50              Y = Y - self.f(self.time_stamp[self.N-1], self.X[:, :, self.N
    -1], Y, Z) * self.l
51              Y = Y + tf.reduce_sum(Z * self.dW[:, :, self.N-1], axis=1,
    keepdims=True)
52              #Loss calculation
53              delta = Y - self.h(self.T, self.X[:, :, self.N])
54              self.clipped_delta = tf.clip_by_value(delta, clip_value_min
    =-50.0, clip_value_max=50.0)
55              self.loss = tf.reduce_mean(self.clipped_delta**2)
56          self.time_build = time.time() - start_time
57
58      def f(self, t, X, Y, Z):
59          #Nonlinear term
60          return - self.lambda_ * tf.norm(Z/np.sqrt(2), ord=2, axis=1,
    keepdims=True)**2
61
62      def h(self, t, X):
63          #Terminal condition
64          return tf.math.log((1 + (tf.norm(X, ord=2, axis=1, keepdims=True))
    **2) / 2)
65
66      def subnetwork(self, x, name):
67          #Create a subnetwork
68          with tf.variable_scope(name):
69              x_layer = self.batch_norm(x, name='initial')
70              layer1 = self.add_layer(x_layer, dim=self.dh, name='layer1')
71              layer2 = self.add_layer(layer1, dim=self.dh, name='layer2')
72              z_layer = self.add_layer(layer2, dim=self.d, activation=None,
    name='result')
73
74          return z_layer
75
76      def add_layer(self, input_, dim, activation=tf.nn.relu, name='linear')
    :
77          #Create a single layer
78          with tf.variable_scope(name):
79              #Input function
```

```
80            shape = input_.get_shape().as_list()
81            w = tf.get_variable('weights'+name, shape=[shape[1], dim],
    dtype=tf.float64, initializer=norm_init(mean=0.0, stddev=5.0/np.sqrt(
    shape[1]+dim)))
82            layer = tf.matmul(input_, w)
83            #Batch normalization
84            layer_bn = self.batch_norm(layer, name='normalization')
85            #Activation
86            if activation != None:
87
88                return activation(layer_bn)
89            else:
90
91                return layer_bn
92
93    def batch_norm(self, x, name):
94        #Perform batch normalization
95        with tf.variable_scope(name):
96            #Define tensor types
97            params_shape = [x.get_shape()[-1]]
98            beta = tf.get_variable('beta', shape=params_shape, dtype=tf.
    float64, initializer=norm_init(mean=0.0, stddev=0.1, dtype=tf.float64)
    )
99            gamma = tf.get_variable('gamma', shape=params_shape, dtype=tf.
    float64, initializer=unif_init(minval=0.1, maxval=0.5, dtype=tf.
    float64))
100           mv_mean = tf.get_variable('moving_mean', shape=params_shape,
    dtype=tf.float64, initializer=const_init(value=0.0, dtype=tf.float64),
     trainable=False)
101           mv_var = tf.get_variable('moving_variance', shape=params_shape
    , dtype=tf.float64, initializer=const_init(value=1.0, dtype=tf.float64
    ), trainable=False)
102           #Fix mean and variance of layer x
103           mean, variance = tf.nn.moments(x, axes=[0], name='moments')
104           self.extra_train_ops.append(assign_moving_average(mv_mean,
    mean, decay=0.99))
105           self.extra_train_ops.append(assign_moving_average(mv_var,
    variance, decay=0.99))
106           mean, variance = control_flow_ops.cond(self.is_training,
    lambda: (mean, variance), lambda: (mv_mean, mv_var))
107           y = tf.nn.batch_normalization(x, mean, variance, offset=beta,
    scale=gamma, variance_epsilon=1e-06)
108           y.set_shape(x.get_shape())
109
110           return y
111
112   def train(self):
113       start_time = time.time()
114       #Define training operations
115       self.global_step = tf.get_variable('global_step', shape=[], dtype=
    tf.int32, initializer=const_init(value=1), trainable=False)
116       grads = tf.gradients(self.loss, tf.trainable_variables())
117       optimizer = tf.train.AdamOptimizer(self.learning_rate)
118       train_ops = [optimizer.apply_gradients(zip(grads, tf.
    trainable_variables()), global_step=self.global_step)] + self.
    extra_train_ops
119       self.train_op = tf.group(*train_ops)
```

```
120         self.loss_history = []
121         self.init_history = []
122         self.runtime_history = []
123         dW_test, X_test = self.sample_path(self.M)
124         feed_dict_test = {self.dW: dW_test, self.X: X_test, self.
    is_training: False}
125         #Initialization
126         step = 1
127         self.sess.run(tf.global_variables_initializer())
128         temp_loss = self.sess.run(self.loss, feed_dict=feed_dict_test)
129         temp_init = self.Y0.eval()[0]
130         runtime = time.time()-start_time+self.time_build
131         self.loss_history.append(temp_loss)
132         self.init_history.append(temp_init)
133         self.runtime_history.append(runtime)
134         #Training
135         for _ in range(self.n_maxstep+1):
136             step = self.sess.run(self.global_step)
137             dW_train, X_train = self.sample_path(self.batch_size)
138             self.sess.run(self.train_op, feed_dict={self.dW: dW_train,
    self.X: X_train, self.is_training: True})
139             if step % self.n_displaystep == 0:
140                 temp_loss = self.sess.run(self.loss, feed_dict=
    feed_dict_test)
141                 temp_init = self.Y0.eval()[0]
142                 runtime = time.time()-start_time+self.time_build
143                 self.loss_history.append(temp_loss)
144                 self.init_history.append(temp_init)
145                 self.runtime_history.append(runtime)
146                 print('step: ', step, '\t loss: ', temp_loss, '\t Y0: ',
    temp_init, '\t runtime: ', runtime)
147             step += 1
148         end_time = time.time()
149 
150     def sample_path(self, n_sample):
151         #Create sample paths
152         dW_sample = np.zeros([n_sample, self.d, self.N])
153         X_sample = np.zeros([n_sample, self.d, self.N+1])
154         for i in range(self.N):
155             dW_sample[:, :, i] = np.reshape(normal.rvs(mean=np.zeros(self.
    d), cov=1, size=n_sample) * np.sqrt(self.l), (n_sample, self.d))
156             X_sample[:, :, i+1] = X_sample[:, :, i] + np.sqrt(2) *
    dW_sample[:, :, i]
157 
158         return dW_sample, X_sample
159 
160 def main():
161     tf.reset_default_graph()
162     with tf.Session() as sess:
163         tf.set_random_seed(1)
164         model = SolveHJB(sess)
165         model.build()
166         model.train()
167         results = np.zeros((len(model.init_history), 4))
168         results[:, 0] = np.arange(len(model.init_history)) * model.
    n_displaystep
169         results[:, 1] = model.loss_history
```

```
        results[:, 2] = model.init_history
        results[:, 3] = model.runtime_history
        mat = np.matrix(results)
        with open('HJB_results_seed1.txt', 'w') as f:
            for line in mat:
                np.savetxt(f, line)

if __name__ == '__main__':
    np.random.seed(1)
    main()
```

Listing C: Python code for solving the Hamilton-Jacobi-Bellman equation with the deep BSDE method implemented in TensorFlow.

D Numerical results for the Allen-Cahn equation

Numerical results when solving the Allen-Cahn equation using random seeds $1, 2, 3, 4, 5$:

```
step, Y0 mean, Y0 std, rel error mean, rel error std, loss mean, loss std, runtime
0, 4.1628e-01, 7.4769e-02, 6.8841e+00, 1.4161e+00, 8.787443e-02, 3.843434e-02, 107
100, 3.6803e-01, 7.4520e-02, 5.9703e+00, 1.4114e+00, 6.535385e-02, 3.200258e-02, 150
200, 3.2422e-01, 7.3803e-02, 5.1405e+00, 1.3978e+00, 4.826244e-02, 2.658619e-02, 165
300, 2.8473e-01, 7.2391e-02, 4.3925e+00, 1.3710e+00, 3.548269e-02, 2.193004e-02, 176
400, 2.4917e-01, 7.0387e-02, 3.7191e+00, 1.3331e+00, 2.582817e-02, 1.785633e-02, 187
500, 2.1759e-01, 6.7610e-02, 3.1210e+00, 1.2805e+00, 1.867666e-02, 1.438847e-02, 198
600, 1.8970e-01, 6.4209e-02, 2.5928e+00, 1.2161e+00, 1.338557e-02, 1.141761e-02, 209
700, 1.6545e-01, 6.0125e-02, 2.1335e+00, 1.1387e+00, 9.538020e-03, 8.914355e-03, 230
800, 1.4464e-01, 5.5477e-02, 1.7393e+00, 1.0507e+00, 6.722671e-03, 6.779927e-03, 240
900, 1.2698e-01, 5.0395e-02, 1.4049e+00, 9.5444e-01, 4.750389e-03, 5.107160e-03, 266
1000, 1.1223e-01, 4.4970e-02, 1.1256e+00, 8.5170e-01, 3.335155e-03, 3.743206e-03, 280
1100, 9.9963e-02, 3.9527e-02, 8.9324e-01, 7.4862e-01, 2.336927e-03, 2.680810e-03, 291
1200, 8.9977e-02, 3.4164e-02, 7.0411e-01, 6.4704e-01, 1.640629e-03, 1.874275e-03, 302
1300, 8.1842e-02, 2.9036e-02, 5.5003e-01, 5.4992e-01, 1.167012e-03, 1.288745e-03, 313
1400, 7.5384e-02, 2.4199e-02, 4.2773e-01, 4.5831e-01, 8.412520e-04, 8.654213e-04, 325
1500, 7.0165e-02, 1.9903e-02, 3.2889e-01, 3.7696e-01, 6.270977e-04, 5.720391e-04, 336
1600, 6.6034e-02, 1.6104e-02, 2.5063e-01, 3.0500e-01, 4.849912e-04, 3.692172e-04, 347
1700, 6.2860e-02, 1.2772e-02, 1.9052e-01, 2.4189e-01, 3.920034e-04, 2.329697e-04, 358
1800, 6.0361e-02, 9.9254e-03, 1.4321e-01, 1.8798e-01, 3.325983e-04, 1.427663e-04, 375
1900, 5.8419e-02, 7.6349e-03, 1.0642e-01, 1.4460e-01, 2.972717e-04, 8.864378e-05, 399
2000, 5.6849e-02, 5.7607e-03, 7.6688e-02, 1.0910e-01, 2.727888e-04, 5.467092e-05, 413
2100, 5.5698e-02, 4.3096e-03, 5.4886e-02, 8.1621e-02, 2.592792e-04, 3.691842e-05, 424
2200, 5.4925e-02, 3.1949e-03, 4.0254e-02, 6.0510e-02, 2.505639e-04, 2.771001e-05, 436
2300, 5.4335e-02, 2.2672e-03, 2.9079e-02, 4.2940e-02, 2.443694e-04, 2.258483e-05, 453
2400, 5.3855e-02, 1.5528e-03, 2.0205e-02, 2.9262e-02, 2.407724e-04, 2.079190e-05, 465
2500, 5.3511e-02, 1.1608e-03, 1.4776e-02, 2.1130e-02, 2.384644e-04, 2.153510e-05, 475
2600, 5.3156e-02, 8.4490e-04, 9.3571e-03, 1.4627e-02, 2.349280e-04, 2.085178e-05, 487
2700, 5.3075e-02, 6.5503e-04, 7.0108e-03, 1.1488e-02, 2.322493e-04, 2.178433e-05, 499
2800, 5.3103e-02, 5.1575e-04, 7.3202e-03, 8.6459e-03, 2.275417e-04, 2.135036e-05, 511
2900, 5.2965e-02, 3.5528e-04, 5.4954e-03, 4.9846e-03, 2.238018e-04, 2.381590e-05, 521
3000, 5.3026e-02, 2.0888e-04, 5.3655e-03, 2.2915e-03, 2.207812e-04, 2.330269e-05, 531
3100, 5.3029e-02, 7.5024e-05, 4.3348e-03, 1.4209e-03, 2.164597e-04, 2.516891e-05, 541
3200, 5.2906e-02, 9.9076e-05, 2.5274e-03, 1.0869e-03, 2.124972e-04, 2.557801e-05, 562
3300, 5.2872e-02, 2.3588e-04, 4.0345e-03, 2.3505e-03, 2.098231e-04, 2.566001e-05, 590
3400, 5.2881e-02, 1.0750e-04, 2.1009e-03, 1.4434e-03, 2.044831e-04, 2.658910e-05, 612
3500, 5.2872e-02, 2.2110e-04, 3.6714e-03, 2.4272e-03, 2.010136e-04, 2.810828e-05, 628
3600, 5.2795e-02, 1.2257e-04, 2.0828e-03, 1.0288e-03, 1.975197e-04, 2.696372e-05, 652
3700, 5.2835e-02, 1.8758e-04, 2.7816e-03, 2.3066e-03, 1.921259e-04, 2.666384e-05, 680
3800, 5.2783e-02, 1.1167e-04, 1.6050e-03, 1.4143e-03, 1.862181e-04, 2.801429e-05, 697
3900, 5.2845e-02, 2.7343e-04, 4.0803e-03, 3.3015e-03, 1.827567e-04, 2.829002e-05, 709
4000, 5.2898e-02, 1.0245e-04, 1.9682e-03, 1.8327e-03, 1.781489e-04, 2.864298e-05, 721
```

Numerical results when solving the Allen-Cahn equation using random seeds $6, 7, 8, 9, 10$:

```
step, Y0 mean, Y0 std, rel error mean, rel error std, loss mean, loss std, runtime
```

```
0, 5.0337e-01, 9.6788e-02, 8.5335e+00, 1.8331e+00, 1.437011e-01, 5.564459e-02, 102
100, 4.5486e-01, 9.6561e-02, 7.6148e+00, 1.8288e+00, 1.121758e-01, 4.753493e-02, 169
200, 4.1033e-01, 9.5887e-02, 6.7714e+00, 1.8160e+00, 8.761520e-02, 4.043618e-02, 181
300, 3.6965e-01, 9.4485e-02, 6.0009e+00, 1.7895e+00, 6.841093e-02, 3.410527e-02, 197
400, 3.3226e-01, 9.2453e-02, 5.2929e+00, 1.7510e+00, 5.323576e-02, 2.858179e-02, 216
500, 2.9822e-01, 8.9687e-02, 4.6482e+00, 1.6986e+00, 4.141740e-02, 2.381148e-02, 230
600, 2.6721e-01, 8.6094e-02, 4.0607e+00, 1.6306e+00, 3.203349e-02, 1.958802e-02, 241
700, 2.3916e-01, 8.1794e-02, 3.5296e+00, 1.5491e+00, 2.461082e-02, 1.588291e-02, 252
800, 2.1388e-01, 7.6770e-02, 3.0507e+00, 1.4540e+00, 1.879751e-02, 1.265861e-02, 264
900, 1.9126e-01, 7.1174e-02, 2.6224e+00, 1.3480e+00, 1.426094e-02, 9.962493e-03, 275
1000, 1.7113e-01, 6.5237e-02, 2.2410e+00, 1.2356e+00, 1.073172e-02, 7.723961e-03, 288
1100, 1.5337e-01, 5.8984e-02, 1.9048e+00, 1.1171e+00, 8.021467e-03, 5.891384e-03, 302
1200, 1.3778e-01, 5.2597e-02, 1.6094e+00, 9.9615e-01, 5.940456e-03, 4.410208e-03, 313
1300, 1.2405e-01, 4.6349e-02, 1.3495e+00, 8.7781e-01, 4.357074e-03, 3.242002e-03, 325
1400, 1.1216e-01, 4.0196e-02, 1.1243e+00, 7.6129e-01, 3.169324e-03, 2.336460e-03, 337
1500, 1.0190e-01, 3.4374e-02, 9.2984e-01, 6.5102e-01, 2.289760e-03, 1.652354e-03, 349
1600, 9.3034e-02, 2.8993e-02, 7.6202e-01, 5.4911e-01, 1.650863e-03, 1.145534e-03, 361
1700, 8.5436e-02, 2.4226e-02, 6.1811e-01, 4.5882e-01, 1.192021e-03, 7.790693e-04, 372
1800, 7.9041e-02, 1.9913e-02, 4.9698e-01, 3.7714e-01, 8.698701e-04, 5.188309e-04, 384
1900, 7.3729e-02, 1.6107e-02, 3.9638e-01, 3.0505e-01, 6.490267e-04, 3.398815e-04, 396
2000, 6.9253e-02, 1.2828e-02, 3.1161e-01, 2.4295e-01, 4.986863e-04, 2.182453e-04, 410
2100, 6.5619e-02, 1.0106e-02, 2.4279e-01, 1.9141e-01, 4.017790e-04, 1.403724e-04, 421
2200, 6.2590e-02, 7.8849e-03, 1.8543e-01, 1.4934e-01, 3.391697e-04, 8.843842e-05, 433
2300, 6.0187e-02, 5.9830e-03, 1.3991e-01, 1.1331e-01, 2.977515e-04, 5.743995e-05, 445
2400, 5.8379e-02, 4.3932e-03, 1.0566e-01, 8.3205e-02, 2.725372e-04, 3.854861e-05, 458
2500, 5.6917e-02, 3.2225e-03, 7.7965e-02, 6.1032e-02, 2.571286e-04, 2.653649e-05, 479
2600, 5.5716e-02, 2.3307e-03, 5.5897e-02, 4.3295e-02, 2.484813e-04, 1.957296e-05, 492
2700, 5.4858e-02, 1.7261e-03, 3.9931e-02, 3.1509e-02, 2.402114e-04, 1.786722e-05, 503
2800, 5.4261e-02, 1.1985e-03, 2.7679e-02, 2.2680e-02, 2.347535e-04, 1.785761e-05, 514
2900, 5.3874e-02, 7.3153e-04, 2.0335e-02, 1.3855e-02, 2.299998e-04, 1.644973e-05, 527
3000, 5.3531e-02, 4.6246e-04, 1.3843e-02, 8.7587e-03, 2.265953e-04, 1.604588e-05, 538
3100, 5.3258e-02, 3.2660e-04, 8.7056e-03, 6.1447e-03, 2.239844e-04, 1.631363e-05, 550
3200, 5.3132e-02, 2.4233e-04, 6.3600e-03, 4.4920e-03, 2.208020e-04, 1.659344e-05, 561
3300, 5.2929e-02, 1.5912e-04, 3.5060e-03, 1.6525e-03, 2.160555e-04, 1.752887e-05, 572
3400, 5.2872e-02, 1.7456e-04, 2.6637e-03, 2.3844e-03, 2.140824e-04, 1.796415e-05, 584
3500, 5.2869e-02, 1.3607e-04, 2.1716e-03, 1.9030e-03, 2.113229e-04, 1.659046e-05, 596
3600, 5.2899e-02, 1.0886e-04, 2.2698e-03, 1.6084e-03, 2.086347e-04, 1.998030e-05, 607
3700, 5.2903e-02, 1.0801e-04, 2.2335e-03, 1.7256e-03, 2.044841e-04, 2.031164e-05, 629
3800, 5.2976e-02, 1.3665e-04, 3.9254e-03, 1.5311e-03, 2.010309e-04, 2.036221e-05, 650
3900, 5.2994e-02, 1.0560e-04, 3.6811e-03, 2.0000e-03, 1.964408e-04, 2.234523e-05, 671
4000, 5.2977e-02, 1.9666e-04, 4.1898e-03, 2.7481e-03, 1.937513e-04, 2.408719e-05, 693
```

E Numerical results for the Hamilton-Jacobi-Bellman equation

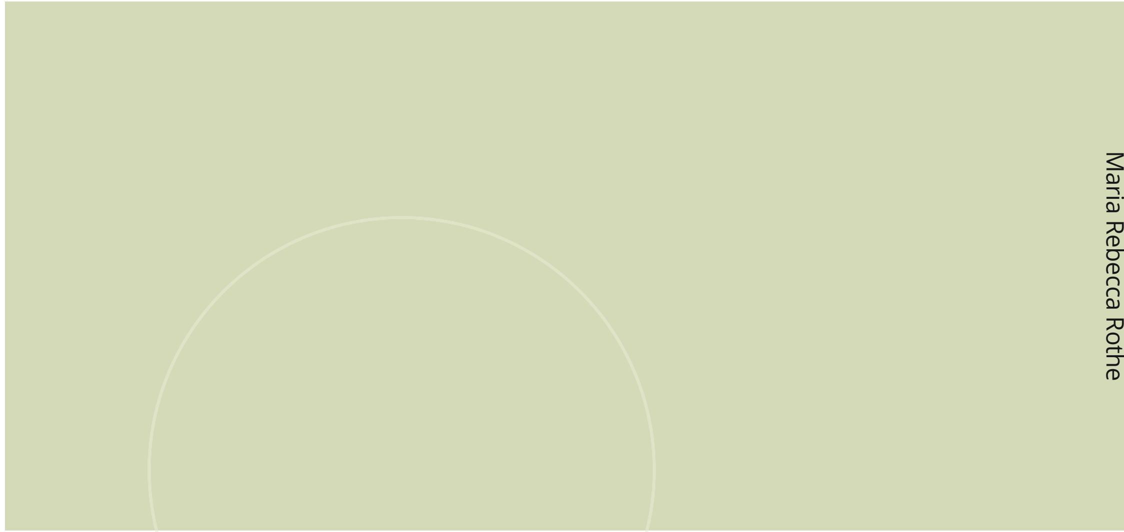
Numerical results when solving the HJB equation using random seeds $1, 2, 3, 4, 5$:

```
step, Y0 mean, Y0 std, rel error mean, rel error std, loss mean, loss std, runtime
0, 4.1628e-01, 7.4769e-02, 9.0931e-01, 1.6289e-02, 1.747056e+01, 5.826944e-01, 101
100, 1.3667e+00, 7.3501e-02, 7.0225e-01, 1.6013e-02, 9.866117e+00, 5.304473e-01, 165
200, 2.1217e+00, 7.3926e-02, 5.3777e-01, 1.6106e-02, 4.700103e+00, 4.148533e-01, 195
300, 2.6081e+00, 7.9677e-02, 4.3180e-01, 1.7358e-02, 3.139560e+00, 3.026362e-01, 225
400, 2.9641e+00, 8.1349e-02, 3.5424e-01, 1.7723e-02, 2.436543e+00, 2.351695e-01, 254
500, 3.3151e+00, 8.5775e-02, 2.7777e-01, 1.8687e-02, 1.749224e+00, 1.856897e-01, 284
600, 3.6616e+00, 8.4640e-02, 2.0229e-01, 1.8440e-02, 1.099425e+00, 1.630479e-01, 296
700, 3.9785e+00, 7.0377e-02, 1.3324e-01, 1.5332e-02, 5.716744e-01, 1.055683e-01, 307
800, 4.2346e+00, 4.9612e-02, 7.7454e-02, 1.0808e-02, 2.282552e-01, 5.158093e-02, 318
900, 4.4125e+00, 3.1511e-02, 3.8693e-02, 6.8650e-03, 7.464607e-02, 1.634928e-02, 330
1000, 4.5146e+00, 1.6807e-02, 1.6459e-02, 3.6617e-03, 3.079260e-02, 3.093240e-03, 359
1100, 4.5643e+00, 8.0359e-03, 5.6168e-03, 1.7507e-03, 2.224122e-02, 9.612427e-04, 388
1200, 4.5866e+00, 3.7256e-03, 1.0106e-03, 4.7152e-04, 2.123183e-02, 9.579685e-04, 418
1300, 4.5945e+00, 1.3024e-03, 9.6716e-04, 2.8375e-04, 2.110954e-02, 1.372498e-03, 447
1400, 4.5980e+00, 1.1659e-03, 1.7216e-03, 2.5400e-04, 2.104015e-02, 9.784806e-04, 477
1500, 4.5991e+00, 1.0234e-03, 1.9687e-03, 2.2295e-04, 2.082558e-02, 8.644073e-04, 506
```

```
1600, 4.5993e+00, 5.9528e-04, 2.0061e-03, 1.2969e-04, 2.103400e-02, 1.039068e-03, 536
1700, 4.5993e+00, 6.4562e-04, 2.0141e-03, 1.4066e-04, 2.072233e-02, 8.715655e-04, 565
1800, 4.5995e+00, 1.2581e-03, 2.0510e-03, 2.7409e-04, 2.072133e-02, 1.265973e-03, 594
1900, 4.5988e+00, 1.5457e-03, 1.8959e-03, 3.3675e-04, 2.071271e-02, 6.700979e-04, 622
2000, 4.6001e+00, 8.0372e-04, 2.1762e-03, 1.7510e-04, 2.085202e-02, 7.466265e-04, 639
```

Numerical results when solving the HJB equation using random seeds $6, 7, 8, 9, 10$:

```
step, Y0 mean, Y0 std, rel error mean, rel error std, loss mean, loss std, runtime
0, 5.0337e-01, 9.6788e-02, 8.9034e-01, 2.1086e-02, 1.668191e+01, 7.602561e-01, 103
100, 1.4521e+00, 9.5866e-02, 6.8365e-01, 2.0885e-02, 9.073578e+00, 5.612733e-01, 166
200, 2.2050e+00, 9.3795e-02, 5.1961e-01, 2.0434e-02, 4.172950e+00, 3.202017e-01, 187
300, 2.6923e+00, 9.8522e-02, 4.1346e-01, 2.1464e-02, 2.769139e+00, 2.827191e-01, 198
400, 3.0507e+00, 1.0550e-01, 3.3538e-01, 2.2985e-02, 2.134726e+00, 2.768237e-01, 213
500, 3.4062e+00, 1.0558e-01, 2.5793e-01, 2.3002e-02, 1.494773e+00, 2.457177e-01, 226
600, 3.7501e+00, 9.9548e-02, 1.8300e-01, 2.1688e-02, 8.994417e-01, 1.935383e-01, 237
700, 4.0524e+00, 8.6611e-02, 1.1715e-01, 1.8869e-02, 4.386448e-01, 1.294924e-01, 258
800, 4.2890e+00, 6.3095e-02, 6.5590e-02, 1.3746e-02, 1.698321e-01, 6.802638e-02, 287
900, 4.4466e+00, 3.7894e-02, 3.1272e-02, 8.2557e-03, 5.866228e-02, 2.466546e-02, 301
1000, 4.5319e+00, 2.0061e-02, 1.2672e-02, 4.3705e-03, 2.908684e-02, 7.967692e-03, 319
1100, 4.5728e+00, 9.3442e-03, 3.7716e-03, 2.0357e-03, 2.300493e-02, 2.855015e-03, 337
1200, 4.5891e+00, 4.4502e-03, 8.9943e-04, 4.2101e-04, 2.200930e-02, 1.624966e-03, 356
1300, 4.5962e+00, 1.6994e-03, 1.3194e-03, 3.7022e-04, 2.215333e-02, 1.520778e-03, 374
1400, 4.5980e+00, 4.5254e-04, 1.7254e-03, 9.8590e-05, 2.194876e-02, 1.132205e-03, 393
1500, 4.5989e+00, 9.6933e-04, 1.9156e-03, 2.1118e-04, 2.163554e-02, 1.341660e-03, 410
1600, 4.5998e+00, 1.5161e-03, 2.1201e-03, 3.3031e-04, 2.150515e-02, 1.124084e-03, 421
1700, 4.5999e+00, 1.3676e-04, 2.1309e-03, 2.9794e-05, 2.171788e-02, 1.412311e-03, 432
1800, 4.6003e+00, 1.1297e-03, 2.2116e-03, 2.4611e-04, 2.170305e-02, 1.176512e-03, 447
1900, 4.5994e+00, 1.1152e-03, 2.0269e-03, 2.4296e-04, 2.125515e-02, 1.296377e-03, 467
2000, 4.6008e+00, 1.6792e-03, 2.3350e-03, 3.6583e-04, 2.141667e-02, 1.429779e-03, 494
```



Maria Rebecca Rothe

Numerical Experiments on a Deep Learning Approach to Solving High-Dimensional Partial Differential Equations

NTNU
Norwegian University of
Science and Technology