Solving Nonlinear and High-Dimensional Partial Differential Equations via Deep Learning

TEAM One

ALI AL-ARADI, University of Toronto
ADOLFO CORREIA, Instituto de Matemática Pura e Aplicada
DANILO NAIFF, Universidade Federal do Rio de Janeiro
GABRIEL JARDIM, Fundação Getulio Vargas

Supervisor:
YURI SAPORITO, Fundação Getulio Vargas

EMAp, Fundação Getulio Vargas, Rio de Janeiro, Brazil

Contents

1 Introduction **4**

2 An Introduction to Partial Differential Equations **6**
- 2.1 Overview . . . 6
- 2.2 The Black-Scholes Partial Differential Equation . . . 8
- 2.3 The Fokker-Planck Equation . . . 10
- 2.4 Stochastic Optimal Control and Optimal Stopping . . . 11
- 2.5 Mean Field Games . . . 18

3 Numerical Methods for PDEs **21**
- 3.1 Finite Difference Method . . . 21
- 3.2 Galerkin methods . . . 25
- 3.3 Finite Element Methods . . . 26
- 3.4 Monte Carlo Methods . . . 27

4 An Introduction to Deep Learning **29**
- 4.1 Neural Networks and Deep Learning . . . 30
- 4.2 Stochastic Gradient Descent . . . 34
- 4.3 Backpropagation . . . 34
- 4.4 Summary . . . 36
- 4.5 The Universal Approximation Theorem . . . 37
- 4.6 Other Topics . . . 37

5 The Deep Galerkin Method **41**
- 5.1 Introduction . . . 41
- 5.2 Mathematical Details . . . 42
- 5.3 A Neural Network Approximation Theorem . . . 44
- 5.4 Implementation Details . . . 44

6 Implementation of the Deep Galerkin Method **47**
- 6.1 How this chapter is organized . . . 48
- 6.2 European Call Options . . . 49
- 6.3 American Put Options . . . 51

6.4 Fokker-Planck Equations 54
6.5 Stochastic Optimal Control Problems 57
6.6 Systemic Risk 63
6.7 Mean Field Games 67
6.8 Conclusions and Future Work 71

Chapter 1

Introduction

In this work we present a methodology for numerically solving a wide class of partial differential equations (PDEs) and PDE systems using deep neural networks. The PDEs we consider are related to various applications in quantitative finance including option pricing, optimal investment and the study of mean field games and systemic risk. The numerical method is based on the **Deep Galerkin Method** (DGM) described in Sirignano and Spiliopoulos (2018) with modifications made depending on the application of interest.

The main idea behind DGM is to represent the unknown function of interest using a deep neural network. Noting that the function must satisfy a known PDE, the network is trained by minimizing losses related to the differential operator, the initial/terminal conditions and the boundary conditions given in the initial value and/or boundary problem. The training data for the neural network consists of different possible inputs to the function and is obtained by sampling randomly from the region on which the PDE is defined. One of the key features of this approach is the fact that, unlike other commonly used numerical approaches such as finite difference methods, it is *mesh-free*. As such, it does not suffer (as much as other numerical methods) from the curse of dimensionality associated with high-dimensional PDEs and PDE systems.

The **main goals** of this paper are to:

1. Present a brief overview of PDEs and how they arise in quantitative finance along with numerical methods for solving them.

2. Present a brief overview of deep learning; in particular, the notion of neural networks, along with an exposition of how they are trained and used.

3. Discuss the theoretical foundations of DGM, with a focus on the justification of why this method is expected to perform well.

4. Elucidate the features, capabilities and limitations of DGM by analyzing aspects of its implementation for a number of different PDEs and PDE systems.

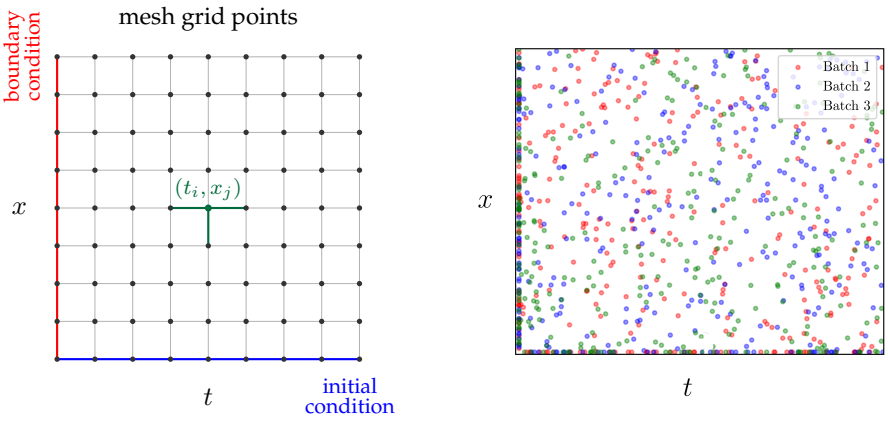

Figure 1.1: *Grid-based finite differences method (left) vs. Deep Galerkin Method (right)*

We present the results in a manner that highlights our own learning process, where we show our failures and the steps we took to remedy any issues we faced. The **main messages** can be distilled into **three main points**:

1. **Sampling method matters**: DGM is based on random sampling; where and how the sampled random points used for training are chosen are the single most important factor in determining the accuracy of the method.

2. **Prior knowledge matters**: similar to other numerical methods, having information about the solution that can guide the implementation dramatically improves the results.

3. **Training time matters**: neural networks sometimes need more time than we afford them and better results can be obtained simply by letting the algorithm run longer.

Chapter 2

An Introduction to Partial Differential Equations

2.1 Overview

Partial differential equations (PDE) are ubiquitous in many areas of science, engineering, economics and finance. They are often used to describe natural phenomena and model multidimensional dynamical systems. In the context of finance, finding solutions to PDEs is crucial for problems of **derivative pricing**, **optimal investment**, **optimal execution**, **mean field games** and many more. In this section, we discuss some introductory aspects of partial differential equations and motivate their importance in quantitative finance with a number of examples.

In short, PDEs describe a relation between a multivariable function and its partial derivatives. There is a great deal of variety in the types of PDEs that one can encounter both in terms of form and complexity. They can vary in **order**; they may be **linear** or **nonlinear**; they can involve various types of **initial/terminal conditions** and **boundary conditions**. In some cases, we can encounter **systems of coupled PDEs** where multiple functions are connected to one another through their partial derivatives. In other cases, we find **free boundary problems** or **variational inequalities** where both the function and its domain are unknown and both must be solved for simultaneously.

To express some of the ideas in the last paragraph mathematically, let us provide some definitions. A k**-th order partial differential equation** is an expression of the form:

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

where D^k is the collection of all partial derivatives of order k and $u : \Omega \to \mathbb{R}$ is the unknown function we wish to solve for.

PDEs can take one of the following forms:

1. **Linear PDE**: derivative coefficients and source term do not depend on any derivatives:
$$\sum_{|\alpha| \leq k} \underbrace{a_\alpha(x) \cdot D^\alpha u}_{\text{linear in derivatives}} = \underbrace{f(x)}_{\text{source term}}$$

2. **Semi-linear PDE**: coefficients of highest order derivatives do not depend on lower order derivatives:
$$\sum_{|\alpha| = k} \underbrace{a_\alpha(x) \cdot D^\alpha u}_{\text{linear in highest order derivatives}} + \underbrace{a_0\left(D^{k-1}u, ..., Du, u, x\right)}_{\text{source term}} = 0$$

3. **Quasi-linear PDE**: linear in highest order derivative with coefficients that depend on lower order derivatives:
$$\sum_{|\alpha| = k} \underbrace{a_\alpha\left(D^{k-1}u, ..., Du, u, x\right)}_{\text{coefficient term of highest order derivative}} \cdot D^\alpha u + \underbrace{a_0\left(D^{k-1}u, ..., Du, u, x\right)}_{\text{source term does not depend on highest order derivative}} = 0$$

4. **Fully nonlinear PDE**: depends nonlinearly on the highest order derivatives.

A **system of partial differential equations** is a collection of several PDEs involving multiple unknown functions:

$$\mathbf{F}\left(D^k \boldsymbol{u}(x), D^{k-1}\boldsymbol{u}(x), ..., D\boldsymbol{u}(x), \boldsymbol{u}(x), x\right) = 0 \qquad x \in \Omega \subset \mathbb{R}^n$$

where $\boldsymbol{u} : \Omega \to \mathbb{R}^m$.

Generally speaking, the PDE forms above are listed in order of increasing difficulty. Furthermore:

- Higher-order PDEs are more difficult to solve than lower-order PDEs;
- Systems of PDEs are more difficult to solve than single PDEs;
- PDEs increase in difficulty with more state variables.

In certain cases, we require the unknown function u to be equal to some known function on the boundary of its domain $\partial\Omega$. Such a condition is known as a **boundary condition** (or an **initial/terminal condition** when dealing with a time dimension). This will be true of the form of the PDEs that we will investigate in Chapter 5.

Next, we present a number of examples to demonstrate the prevalence of PDEs in financial applications. Further discussion of the basics of PDEs (and more advanced topics) such as well-posedness, existence and uniqueness of solutions, classical and weak solutions and regularity can be found in Evans (2010).

2.2 The Black-Scholes Partial Differential Equation

One of the most well-known results in quantitative finance is the **Black-Scholes Equation** and the associated **Black-Scholes PDE** discussed in the seminal work of Black and Scholes (1973). Though they are used to solve for the price of various financial derivatives, for illustrative purposes we begin with a simple variant of this equation relevant for pricing a European-style contingent claim.

2.2.1 European-Style Derivatives

European-style contingent claims are financial instruments written on a source of uncertainty with a payoff that depends on the level of the underlying at a predetermined maturity date. We assume a simple market model known as the **Black-Scholes model** wherein a risky asset follows a geometric Brownian motion (GBM) with constant drift and volatility parameters and where the short rate of interest is constant. That is, the dynamics of the price processes for a risky asset $X = (X_t)_{t\geq 0}$ and a riskless bank account $B = (B_t)_{t\geq 0}$ under the "real-world" probability measure $\mathbb{P}$ are given by:

$$\begin{aligned} \frac{dX_t}{X_t} &= \mu\, dt + \sigma\, dW_t \\ \frac{dB_t}{B_t} &= r\, dt \end{aligned}$$

where $W = (W_t)_{t\geq 0}$ is a $\mathbb{P}$-Brownian motion.

We are interested in pricing a claim written on the asset X with payoff function $G(x)$ and with an expiration date T. Then, assuming that the claim's price function $g(t, x)$ - which determines the value of the claim at time t when the underlying asset is at the level $X_t = x$ - is sufficiently smooth, it can be shown by dynamic hedging and no-arbitrage arguments that g must satisfy the **Black-Scholes PDE**:

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

This simple model and the corresponding PDE can extend in several ways, e.g.

- incorporating additional sources of uncertainty;
- including *non-traded* processes as underlying sources of uncertainty;
- allowing for richer asset price dynamics, e.g. jumps, stochastic volatility;
- pricing more complex payoffs functions, e.g. path-dependent payoffs.

2.2.2 American-Style Derivatives

In contrast to European-style contingent claims, American-style derivatives allow the option holder to exercise the derivative *prior* to the maturity date and receive the payoff immediately based on the prevailing value of the underlying. This can be described as an **optimal stopping problem** (more on this topic in Section 2.4).

To describe the problem of pricing an American option, let $\mathcal{T}[t,T]$ be the set of admissible stopping times in $[t,T]$ at which the option holder can exercise, and let $\mathbb{Q}$ be the risk-neutral measure. Then the price of an American-style contingent claim is given by:

$$g(t,x) = \sup_{\tau \in \mathcal{T}[t,T]} \mathbb{E}^{\mathbb{Q}} \left[e^{-r(\tau - t)} G(X_\tau) \;\middle|\; X_t = x \right]$$

Using dynamic programming arguments it can be shown that optimal stopping problems admit a **dynamic programming equation**. In this case, the solution of this equation yields the price of the American option. Assuming the same market model as the previous section, it can be shown that the price function for the American-style option $g(t,x)$ with payoff function $G(x)$ - assuming sufficient smoothness - satisfies the **variational inequality**:

$$\max\left\{ (\partial_t + \mathcal{L} - r)\, g,\; G - g \right\} = 0, \qquad \text{for } (t,x) \in [0,T] \times \mathbb{R}$$

where $\mathcal{L} = rx \cdot \partial_x + \frac{1}{2}\sigma^2 x^2 \cdot \partial_{xx}$ is a differential operator.

The last equation has a simple interpretation. Of the two terms in the curly brackets, one will be equal to zero while the other will be negative. The first term is equal to zero when $g(t,x) > G(x)$, i.e. when the option value is greater than the intrinsic

(early exercise) value, the option is not exercised early and the price function satisfies the usual Black-Scholes PDE. When the second term is equal to zero we have that $g(t,x) = G(x)$, in other words the option value is equal to the exercise value (i.e. the option is exercised). As such, the region where $g(t,x) > G(x)$ is referred to as the **continuation region** and the curve where $g(t,x) = G(x)$ is called the **exercise boundary**. Notice that it is not possible to have $g(t,x) < G(x)$ since both terms are bounded above by 0.

It is also worth noting that this variational inequality can be written as follows:

$$\begin{cases} \partial_t g + rx \cdot \partial_x g + \frac{1}{2}\sigma^2 x^2 \cdot \partial_{xx} g - r \cdot g = 0 & \{(t,x) : g(t,x) > G(x)\} \\ g(t,x) \geq G(x) & (t,x) \in [0,T] \times \mathbb{R} \\ g(T,x) \ = \ G(x) & x \in \mathbb{R} \end{cases}$$

where we drop the explicit dependence on (t,x) for brevity. The **free boundary** set in this problem is $F = \{(t,x) : g(t,x) = G(x)\}$ which must be determined alongside the unknown price function g. The set F is referred to as the **exercise boundary**; once the price of the underlying asset hits the boundary, the investor's optimal action is to exercise the option immediately.

2.3 The Fokker-Planck Equation

We now turn out attention to another application of PDEs in the context of stochastic processes. Suppose we have an Itô process on $\mathbb{R}^d$ with time-independent drift and diffusion coefficients:

$$d\boldsymbol{X}_t = \mu(\boldsymbol{X}_t)dt + \sigma(\boldsymbol{X}_t)dW_t$$

and assume that the initial point is a random vector $\boldsymbol{X}_0$ with distribution given by a probability density function $f(\boldsymbol{x})$. A natural question to ask is: *"what is the probability that the process is in a given region $A \subset \mathbb{R}^d$ at time t?"* This quantity can be computed as an integral of the probability density function of the random vector $\boldsymbol{X}_t$, denoted by $p(t,\boldsymbol{x})$:

$$\mathbb{P}\left(\boldsymbol{X}_t \in A\right) = \int_A p(t,\boldsymbol{x})\, d\boldsymbol{x}$$

The **Fokker-Planck equation** is a partial differential equation that $p(t,\boldsymbol{x})$ can be shown to satisfy:

$$\begin{cases} \partial_t p(t,\boldsymbol{x}) + \sum_{j=1}^d \partial_j(\mu_j(\boldsymbol{x}) \cdot p(t,\boldsymbol{x})) \\ \qquad\qquad - \frac{1}{2}\sum_{i,j=1}^d \partial_{ij}(\sigma_{ij}(\boldsymbol{x}) \cdot p(t,\boldsymbol{x})) = 0 & (t,\boldsymbol{x}) \in \mathbb{R}_+ \times \mathbb{R}^d \\ p(0,\boldsymbol{x}) = f(\boldsymbol{x}) & \boldsymbol{x} \in \mathbb{R}^d \end{cases}$$

where ∂_j and ∂_{ij} are first and second order partial differentiation operators with respect to x_j and x_i and x_j, respectively. Under certain conditions on the initial distribution f, the above PDE admits a unique solution. Furthermore, the solution satisfies the property that $p(t, \boldsymbol{x})$ is positive and integrates to 1, which is required of a probability density function.

As an example consider an **Ornstein-Uhlenbeck** (OU) process $X = (X_t)_{t \geq 0}$ with a random starting point distributed according to an independent normal random variable with mean 0 and variance v. That is, X satisfies the stochastic differential equation (SDE):

$$dX_t = \kappa(\theta - X_t)\, dt + \sigma\, dW_t\,, \qquad X_0 \sim N(0, v)$$

where θ and κ are constants representing the mean reversion level and rate. Then the probability density function $p(t, x)$ for the location of the process at time t satisfies the PDE:

$$\begin{cases} \partial_t p + \kappa \cdot p + \kappa(x - \theta) \cdot \partial_x p - \frac{1}{2}\sigma^2 \cdot \partial_{xx} p = 0 & (t, x) \in \mathbb{R}_+ \times \mathbb{R} \\ p(0, x) = \frac{1}{\sqrt{2\pi v}} \cdot e^{-\frac{x^2}{2v}} \end{cases}$$

Since the OU process with a fixed starting point is a Gaussian process, using a normally distributed random starting point amounts to combining the conditional distribution process with its (conjugate) prior, implying that X_t is normally distributed. We omit the derivation of the exact form of $p(t, x)$ in this case.

2.4 Stochastic Optimal Control and Optimal Stopping

Two classes of problems that heavily feature PDEs are **stochastic optimal control** and **optimal stopping** problems. In this section we give a brief overview of these problems along with some examples. For a thorough overview, see Touzi (2012), Pham (2009) or Cartea et al. (2015).

In stochastic control problems, a controller attempts to maximize a measure of success - referred to as a **performance criteria** - which depends on the path of some stochastic process by taking actions (choosing controls) that influence the dynamics of the process. In optimal stopping problems, the performance criteria depends on a stopping time chosen by the agent; the early exercise of American options discussed earlier in this chapter is an example of such a problem.

To discuss these in concrete terms let $X = (X_t)_{t \geq 0}$ be a controlled Itô process satisfying the stochastic differential equation:

$$dX_t^u = \mu(t, X_t^u, u_t)\, dt + \sigma(t, X_t^u, u_t)\, dW_t\,, \qquad X_0^u = 0$$

where $u = (u_t)_{t \geq 0}$ is a control process chosen by the controller from an admissible set $\mathcal{A}$. Notice that the drift and volatility of the process are influenced by the controller's actions. For a given control, the agent's performance criteria is:

$$H^u(x) = \mathbb{E}\Bigg[\underbrace{\int_0^T F(s, X_s^u, u_s)\, ds}_{\text{running reward}} + \underbrace{G(X_T^u)}_{\text{terminal reward}} \Bigg]$$

The key to solving optimal control problems and finding the optimal control u^* lies in the **dynamic programming principle** (DPP) which involves embedding the original optimization problem into a larger class of problems indexed by time, with the original problem corresponding to $t = 0$. This requires us to define:

$$H^u(t, x) = \mathbb{E}_{t,x}\left[\int_t^T F(s, X_s^u, u_s)\, ds + G(X_T^u) \right]$$

where $\mathbb{E}_{t,x}[\cdot] = \mathbb{E}[\cdot | X_t^u = x]$. The **value function** is the value of the performance criteria when the agent adopts the optimal control:

$$H(t, x) = \sup_{u \in \mathcal{A}} H^u(t, x)$$

Assuming enough regularity, the value function can be shown to satisfy a **dynamic programming equation** (DPE) also called a **Hamilton-Jacobi-Bellman** (HJB) equation. This is a PDE that can be viewed as an infinitesimal version of the DPP. The HJB equation is given by:

$$\begin{cases} \partial_t H(t, x) + \sup\limits_{u \in \mathcal{A}} \{\mathcal{L}_t^u H(t, x) + F(t, x, u)\} = 0 \\ H(T, x) = G(x) \end{cases}$$

where the differential operator $\mathcal{L}_t^u$ is the **infinitesimal generator** of the controlled process X^u - an analogue of derivatives for stochastic processes - given by:

$$\mathcal{L} f(t, X_t) = \lim_{h \downarrow 0} \frac{\mathbb{E}_t[f(t + h, X_{t+h})] - f(t, X_t)}{h}$$

Broadly speaking, the optimal control is obtained as follows:

1. Solve the first order condition (inner optimization) to obtain the optimal control in terms of the derivatives of the value function, i.e. in feedback form;
2. Substitute the optimal control back into the HJB equation, usually yielding a highly nonlinear PDE and solve the PDE for the unknown value function;
3. Use the value function to derive an explicit expression for the optimal control.

For optimal stopping problems, the optimization problem can be written as:

$$\sup_{\tau \in \mathcal{T}} \mathbb{E}\left[G(X_\tau)\right]$$

where $\mathcal{T}$ is the set of admissible stopping times. Similar to the optimal control problem, we can derive a DPE for optimal stopping problem in the form of a **variational inequality** assuming sufficient regularity in the value function H. Namely,

$$\max\left\{(\partial_t + \mathcal{L}_t)H,\ G - H\right\} = 0, \qquad \text{on } [0,T] \times \mathbb{R}$$

The interpretation of this equation was discussed in Section 2.2.2 for American-style derivatives where we discussed how the equation can be viewed as a free boundary problem.

It is possible to extend the problems discussed in this section in many directions by considering multidimensional processes, infinite horizons (for running rewards), incorporating jumps and combining optimal control and stopping in a single problem. This will lead to more complex forms of the corresponding dynamic programming equation.

Next, we discuss a number of examples of HJB equations that arise in the context of problems in quantitative finance.

2.4.1 The Merton Problem

In the **Merton problem**, an agent chooses the proportion of their wealth that they wish to invest in a risky asset and a risk-free asset through time. They seek to maximize the expected utility of terminal wealth at the end of their investment horizon; see Merton (1969) for the investment-consumption problem and Merton (1971) for extensions in a number of directions. Once again, we assume the Black-Scholes market model:

$$\begin{aligned}\frac{dS_t}{S_t} &= \mu\, dt + \sigma\, dW_t \\ \frac{dB_t}{B_t} &= r\, dt\end{aligned}$$

The wealth process X_t^π of a portfolio that invests a proportion π_t of wealth in the risky asset and the remainder in the risk-free asset satisfies the following SDE:

$$dX_t^\pi = (\pi_t(\mu - r) + rX_t^\pi)\, dt + \sigma\pi_t\, dW_t$$

The investor is faced with the following optimal stochastic control problem:

$$\sup_{\pi \in \mathcal{A}} \mathbb{E}\left[U(X_T^\pi)\right]$$

where $\mathcal{A}$ is the set of admissible strategies and $U(x)$ is the investor's utility function. The value function is given by:

$$H(t,x) = \sup_{\pi \in \mathcal{A}} \mathbb{E}\left[U(X_T^\pi) \mid X_t^\pi = x\right]$$

which satisfies the following HJB equation:

$$\begin{cases} \partial_t H + \sup\limits_{\pi \in \mathcal{A}} \left\{ \left((\pi(\mu - r) + rx) \cdot \partial_x + \frac{1}{2}\sigma^2\pi^2\partial_{xx} \right) H \right\} = 0 \\ H(T,x) = U(x) \end{cases}$$

If we assume an exponential utility function with risk preference parameter γ, that is $U(x) = -e^{-\gamma x}$, then the value function and the optimal control can be obtained in closed-form:

$$H(t,x) = -\exp\left[-x\gamma e^{r(T-t)} - \tfrac{\lambda^2}{2}(T-t)\right]$$
$$\pi_t^* = \frac{\lambda}{\gamma\sigma} e^{-r(T-t)}$$

where $\lambda = \frac{\mu - r}{\sigma}$ is the market price of risk.

It is also worthwhile to note that the solution to the Merton problem plays an important role in the **substitute hedging** and **indifference pricing** literature, see e.g. Henderson and Hobson (2002) and Henderson and Hobson (2004).

2.4.2 Optimal Execution with Price Impact

Stochastic optimal control, and hence PDEs in the form of HJB equations, feature prominently in the algorithmic trading literature, such as in the classical work of Almgren and Chriss (2001) and more recently Cartea and Jaimungal (2015) and Cartea and Jaimungal (2016) to name a few. Here we discuss a simple algorithmic trading problem with an investor that wishes to liquidate an inventory of shares but is subject to price impact effects when trading too quickly. The challenge then involves balancing this effect with the possibility of experiencing a negative market move when trading too slowly.

We begin by describing the dynamics of the main processes underlying the model. The agent can control their (liquidation) **trading rate** ν_t which in turn affects their **inventory level** Q_t^ν via:

$$dQ_t^\nu = -\nu_t \, dt, \qquad Q_0^\nu = q$$

Note that negative values of ν indicate that the agent is buying shares. The price of the underlying asset S_t is modeled as a Brownian motion that experiences a

permanent price impact due to the agent's trading activity in the form of a linear increase in the drift term:

$$dS_t^\nu = -b\nu_t\, dt + \sigma\, dW_t, \qquad S_0^\nu = S$$

By selling too quickly the agent applies increasing downward pressure (linearly with factor $b > 0$) on the asset price which is unfavorable to a liquidating agent. Furthermore, placing larger orders also comes at the cost of increased **temporary price impact**. This is modeled by noting that the cashflow from a particular transaction is based on the **execution price** $\widehat{S}_t$ which is linearly related to the fundamental price (with a factor of $k > 0$):

$$\widehat{S}_t^\nu = S_t^\nu - k\nu_t$$

The **cash process** X_t^ν evolves according to:

$$dX_t^\nu = \widehat{S}_t^\nu \nu_t\, dt, \qquad X_0^\nu = x$$

With the model in place we can consider the agent's performance criteria, which consists of maximizing their terminal cash and penalties for excess inventory levels both at the terminal date and throughout the liquidation horizon. The performance criteria is

$$H^\nu(t,x,S,q) = \mathbb{E}_{t,x,S,q}\Bigg[\underbrace{X_T^\nu}_{\substack{\text{terminal}\\ \text{cash}}} + \underbrace{Q_T^\nu\left(S_T^\nu - \alpha Q_T^\nu\right)}_{\substack{\text{terminal}\\ \text{inventory}}} - \underbrace{\phi \int_t^T \left(Q_u^\nu\right)^2 du}_{\text{running inventory}} \Bigg]$$

where α and ϕ are preference parameters that control the level of penalty for the terminal and running inventories respectively. The value function satisfies the HJB equation:

$$\begin{cases} (\partial_t + \frac{1}{2}\sigma^2\partial_{SS})H - \phi q^2 \\ \qquad + \sup_\nu \left\{ (\nu(S - k\nu)\partial_x - b\nu\cdot\partial_S - \nu\partial_q)\, H \right\} = 0 \\ H(t,x,S,q) = x + Sq - \alpha q^2 \end{cases}$$

Using a carefully chosen ansatz we can solve for the value function and optimal control:

$$H(t,x,S,q) = x + qS + \left(h(t) - \tfrac{b}{2}\right) q^2$$

$$\nu_t^* = \gamma \cdot \frac{\zeta e^{\gamma(T-t)} + e^{-\gamma(T-t)}}{\zeta e^{\gamma(T-t)} - e^{-\gamma(T-t)}} \cdot Q_t^{\nu^*}$$

$$\text{where} \quad h(t) = \sqrt{k\phi} \cdot \frac{1 + \zeta e^{2\gamma(T-t)}}{1 - \zeta e^{2\gamma(T-t)}}, \qquad \gamma = \sqrt{\frac{\phi}{k}}, \qquad \zeta = \frac{\alpha - \frac{1}{2}b + \sqrt{k\phi}}{\alpha - \frac{1}{2}b - \sqrt{k\phi}}$$

For other optimal execution problems the interested reader is referred to Chapter 6 of Cartea et al. (2015).

2.4.3 Systemic Risk

Yet another application of PDEs in optimal control is the topic of Carmona et al. (2015). The focus in that paper is on **systemic risk** - the study of instability in the entire market rather than a single entity - where a number of banks are borrowing and lending with the central bank with the target of being at or around the average monetary reserve level across the economy. Once a characterization of optimal behavior is obtained, questions surrounding the stability of the system and the possibility of multiple defaults can be addressed. This is an example of a **stochastic game**, with multiple players determining their preferred course of action based on the actions of others. The object in stochastic games is usually the determination of **Nash equilibria** or sets of strategies where no player has an incentive to change their action.

The main processes underlying this problem are the log-monetary reserves of each bank denoted $X^i = \left(X_t^i\right)_{t\geq 0}$ and assumed to satisfy the SDE:

$$dX_t^i = \left[a\left(\overline{X}_t - X_t^i\right) + \alpha_t^i\right] dt + \sigma\, d\widetilde{W}_t^i$$

where $\widetilde{W}_t^i = \rho W_t^0 + \sqrt{1-\rho^2} W_t^i$ are Brownian motions correlated through a common noise process, $\overline{X}_t$ is the average log-reserve level and α_t^i is the rate at which bank i borrows from or lends to the central bank. The interdependence of reserves appears in a number of places: first, the drift contains a mean reversion term that draws each bank's reserve level to the average with a mean reversion rate a; second, the noise terms are driven partially by a common noise process.

The agent's control in this problem is the borrowing/lending rate α^i. Their aim is to remain close to the average reserve level at all times over some fixed horizon. Thus, they penalize any deviations from this (stochastic) average level in the interim and at the end of the horizon. They also penalize borrowing and lending from the central bank at high rates as well as borrowing (resp. lending) when their own reserve level is above (resp. below) the average level. Formally, the performance criterion is given by:

$$J^i\left(\alpha^1, ..., \alpha^N\right) = \mathbb{E}\left[\int_0^T f_i\left(\boldsymbol{X}_t, \alpha_t^i\right) dt + g_i\left(X_T^i\right)\right]$$

where the running penalties are:

$$f_i(\boldsymbol{x}, \alpha^i) = \underbrace{\tfrac{1}{2}\left(\alpha^i\right)^2}_{\substack{\text{excessive lending} \\ \text{or borrowing}}} - \underbrace{q\alpha^i\left(\overline{x} - x^i\right)}_{\substack{\text{borrowing/lending in} \\ \text{“the wrong direction”}}} + \underbrace{\tfrac{\epsilon}{2}\left(\overline{x} - x^i\right)^2}_{\substack{\text{deviation from the} \\ \text{average level}}}$$

and the terminal penalty is:

$$g_i(\boldsymbol{x}) = \underbrace{\tfrac{c}{2}\left(\overline{x} - x^i\right)^2}_{\substack{\text{deviation from the} \\ \text{average level}}}$$

where c, q, and ϵ represent the investor's preferences with respect to the various penalties. Notice that the performance criteria for each agent depends on the strategies and reserve levels of all the agents including themselves. Although the paper discusses multiple approaches to solving the problem (Pontryagin stochastic maximum principle and an alternative forward-backward SDE approach), we focus on the HJB approach as this leads to a system of nonlinear PDEs. Using the dynamic programming principle, the HJB equation for agent i is:

$$\begin{cases} \partial_t V^i + \inf\limits_{\alpha^i} \Big\{ \sum\limits_{j=1}^{N} \left[a(\overline{x} - x^j) + \alpha^j \right] \partial_j V^i \\ \qquad\qquad + \dfrac{\sigma^2}{2} \sum\limits_{j,k=1}^{N} \left(\rho^2 + \delta_{jk}(1-\rho^2) \right) \partial_{jk} V^i \\ \qquad\qquad + \frac{(\alpha^i)^2}{2} - q\alpha^i(\overline{x} - x^i) + \frac{\epsilon}{2}\left(\overline{x} - x^i\right)^2 \Big\} = 0 \\ V^i(T, \boldsymbol{x}) = \frac{c}{2}\left(\overline{x} - x^i\right)^2 \end{cases}$$

Remarkably, this system of PDEs can be solved in closed-form to obtain the value function and the optimal control for each agent:

$$\begin{aligned} V^i(t, \boldsymbol{x}) &= \frac{\eta(t)}{2}\left(\overline{x} - x^i\right)^2 + \mu(t) \\ \alpha_t^{i,*} &= \left(q + \left(1 - \tfrac{1}{N}\right) \cdot \eta(t) \right) \left(\overline{X}_t - X_t^i \right) \\ \text{where} \qquad \eta(t) &= \frac{-(\epsilon - q)^2 \left(e^{(\delta^+ - \delta^-)(T-t)} - 1 \right) - c\left(\delta^+ e^{(\delta^+ - \delta^-)(T-t)} - \delta^- \right)}{\left(\delta^- e^{(\delta^+ - \delta^-)(T-t)} - \delta^+ \right) - c\left(1 - \frac{1}{N^2}\right)\left(e^{(\delta^+ - \delta^-)(T-t)} - 1 \right)} \\ \mu(t) &= \tfrac{1}{2}\sigma^2(1-\rho^2)\left(1 - \tfrac{1}{N}\right) \int_t^T \eta(s)\, ds \\ \delta^{\pm} &= -(a+q) \pm \sqrt{R}, \qquad\qquad R = (a+q)^2 + \left(1 - \tfrac{1}{N^2}\right)(\epsilon - q^2) \end{aligned}$$

2.5 Mean Field Games

The final application of PDEs that we will consider is that of **mean field games** (MFGs). In financial contexts, MFGs are concerned with modeling the behavior of a large number of small interacting market participants. In a sense, it can be viewed as a limiting form of the Nash equilibria for finite-player stochastic game (such as the interbank borrowing/lending problem from the previous section) as the number of participants tends to infinity. Though it may appear that this would make the problem more complicated, it is often the case that this simplifies the underlying control problem. This is because in MFGs, agents need not concern themselves with the actions of every other agent, but rather they pay attention only to the aggregate behavior of the other agents (the mean field). It is also possible in some cases to use the limiting solution to obtain approximations for Nash equilibria of finite player games when direct computation of this quantity is infeasible. The term ''mean field" originates from mean field theory in physics which, similar to the financial context, studies systems composed of large numbers of particles where individual particles have negligible impact on the system. A mean field game typically consists of:

1. An **HJB equation** describing the optimal control problem of an individual;
2. A **Fokker-Planck equation** which governs the dynamics of the aggregate behavior of all agents.

Much of the pioneering work in MFGs is due to Huang et al. (2006) and Lasry and Lions (2007), but the focus of our exposition will be on a more recent work by Cardaliaguet and Lehalle (2017). Building on the optimal execution problem discussed earlier in this chapter, Cardaliaguet and Lehalle (2017) propose extensions in a number of directions. First, traders are assumed to be part of a mean field game and the price of the underlying asset is impacted permanently, not only by the actions of the agent, but by the aggregate behavior of all agents acting in an optimal manner. In addition to this aggregate permanent impact, an individual trader faces the usual temporary impact effects of trading too quickly. The other extension is to allow for varying preferences among the traders in the economy. That is, traders may have different tolerance levels for the size of their inventories both throughout the investment horizon and at its end. Intuitively, this framework can be thought of as the agents attempting to "trade optimally within the crowd."

Proceeding to the mathematical description of the problem, we have the following dynamics for the various agents' **inventory and cash processes** (indexed by a superscript a):

$$\begin{aligned} dQ_t^a &= \nu_t^a \, dt, & Q_0^a &= q^a \\ dX_t^a &= -\nu_t^a \left(S_t + k\nu_t^a\right) \, dt, & X_0^a &= x^a \end{aligned}$$

An important deviation from the previous case is the fact that the permanent price impact is due to the **net sum of the trading rates of all agents**, denoted by μ_t:

$$dS_t = \kappa \mu_t \, dt + \sigma \, dW_t$$

Also, the value function associated with the optimal control problem for agent a is given by:

$$H^a(t,x,S,q) = \sup_\nu \mathbb{E}_{t,x,S,q}\Big[\underbrace{X_T^a}_{\substack{\text{terminal} \\ \text{cash}}} + \underbrace{Q_T^a \left(S_T - \alpha^a Q_T^a\right)}_{\substack{\text{terminal} \\ \text{inventory}}} - \underbrace{\phi^a \int_t^T \left(Q_u^a\right)^2 du}_{\text{running inventory}} \Big]$$

Notice that each agent a has a *different* value of α^a and ϕ^a demonstrating their differing preferences. As a consequence, an agent can be represented by their preferences $a = (\alpha^a, \phi^a)$. The HJB equation associated with the agents' control problem is:

$$\begin{cases} \left(\partial_t + \frac{1}{2}\sigma^2 \partial_{SS}\right) H^a - \phi^a q^2 + \kappa\mu \cdot \partial_S H^a + \sup_\nu \left\{ \left(\nu \cdot \partial_q - \nu(S + k\nu) \cdot \partial_x\right) H^a \right\} = 0 \\ H^a(T,x,S,q;\mu) = x + q(S - \alpha^a q) \end{cases}$$

This can be simplified using an ansatz to:

$$\begin{cases} -\kappa\mu q = \partial_t h^a - \phi^a q^2 + \sup_\nu \left\{ \nu \cdot \partial_q h^a - k\nu^2 \right\} \\ h^a(T,q) = -\alpha^a q^2 \end{cases}$$

Notice that the PDE above requires agents to know the net trading flow of the mean field μ, but that this quantity itself depends on the value function of each agent which we have yet to solve for. To resolve this issue we first write the optimal control of each agent in feedback form:

$$\nu^a(t,q) = \frac{\partial_q h^a(t,q)}{2k}$$

Next, we assume that the distribution of inventories and preferences of agents is captured by a density function $m(t, dq, da)$. With this, the net flow μ_t is simply given by the aggregation of all agents' optimal actions:

$$\mu_t = \int_{(q,a)} \underbrace{\frac{\partial h^a(t,q)}{2k}}_{\substack{\text{trading rate of agent} \\ \text{with inventory } q \\ \text{and preferences } a}} \underbrace{m(t,dq,da)}_{\substack{\text{aggregated according to} \\ \text{distribution of agents}}}$$

In order to compute the quantity at different points in time we need to understand the evolution of the density m through time. This is just an application of the Fokker-Planck equation, as m is a density that depends on a stochastic process (the inventory level). If we assume that the initial density of inventories and preferences is $m_0(q, a)$, we can write the Fokker-Planck equation as:

$$\begin{cases} \partial_t m + \partial_q \left(m \cdot \underbrace{\frac{\partial h^a(t,q)}{2k}}_{\substack{\text{drift of inventory} \\ \text{process } Q_t^a \text{ under} \\ \text{optimal controls}}} \right) = 0 \\ m(0, q, a) = m_0(q, a) \end{cases}$$

The full system for the MFG in the problem of Cardaliaguet and Lehalle (2017) involves the combined HJB and Fokker-Planck equations with the appropriate initial and terminal conditions:

$$\begin{cases} -\kappa\mu q = \partial_t h^a - \phi^a q^2 + \dfrac{(\partial_q h^a)^2}{4k} & \text{(HJB equation - optimality)} \\ H^a(T, x, S, q; \mu) = x + q(S - \alpha^a q) & \text{(HJB terminal condition)} \\ \\ \partial_t m + \partial_q \left(m \cdot \dfrac{\partial h^a(t,q)}{2k} \right) = 0 & \text{(FP equation - density flow)} \\ m(0, q, a) = m_0(q, a) & \text{(FP initial condition)} \\ \\ \mu_t = \displaystyle\int_{(q,a)} \frac{\partial h^a(t,q)}{2k}\, m(t, dq, da) & \text{(net trading flow)} \end{cases}$$

Assuming identical preferences $\alpha^a = \alpha, \phi^a = \phi$ allows us to find a closed-form solution to this PDE system. The form of the solution is fairly involved so we refer the interested reader to the details in Cardaliaguet and Lehalle (2017).

Chapter 3

Numerical Methods for PDEs

Although it is possible to obtain closed-form solutions to PDEs, more often we must resort to numerical methods for arriving at a solution. In this chapter we discuss some of the approaches taken to solve PDEs numerically. We also touch on some of the difficulties that may arise in these approaches involving stability and computational cost, especially in higher dimensions. This is by no means a comprehensive overview of the topic to which a vast amount of literature is dedicated. Further details can be found in Burden et al. (2001), Achdou and Pironneau (2005) and Brandimarte (2013).

3.1 Finite Difference Method

It is often the case that differential equations cannot be solved analytically, so one must resort to numerical methods to solve them. One of the most popular numerical methods is the **finite difference method**. As its name suggests, the main idea behind this method is to approximate the differential operators with difference operators and apply them to a discretized version of the unknown function in the differential equation.

3.1.1 Euler's Method

Arguably, the simplest finite difference method is **Euler's method** for **ordinary differential equations** (ODEs). Suppose we have the following initial value problem

$$\begin{cases} y'(t) = f(t) \\ y(0) = y_0 \end{cases}$$

for which we are trying to solve for the function $y(t)$. By the Taylor series expansion, we can write

$$y(t+h) = y(t) + \frac{y'(t)}{1!} \cdot h + \frac{y''(t)}{2!} \cdot h^2 + \cdots$$

for any infinitely differentiable real-valued function y. If h is small enough, and if the derivatives of y satisfy some regularity conditions, then terms of order h^2 and higher are negligible and we can make the approximation

$$y(t+h) \approx y(t) + y'(t) \cdot h$$

As a side note, notice that we can rewrite this equation as

$$y'(t) \approx \frac{y(t+h) - y(t)}{h}$$

which closely resembles the definition of a derivative;

$$y'(t) = \lim_{h \to 0} \frac{y(t+h) - y(t)}{h}.$$

Returning to the original problem, note that we know the exact value of $y'(t)$, namely $f(t)$, so that we can write

$$y(t+h) \approx y(t) + f(t) \cdot h.$$

At this point, it is helpful to introduce the notation for the discretization scheme typically used for finite difference methods. Let $\{t_i\}$ be the sequence of values assumed by the time variable, such that $t_0 = 0$ and $t_{i+1} = t_i + h$, and let $\{y_i\}$ be the sequence of approximations of $y(t)$ such that $y_i \approx y(t_i)$. The expression above can be rewritten as

$$y_{i+1} \approx y_i + f(t_i) \cdot h,$$

which allows us to find an approximation for the value of $y\left(t_{i+1}\right) \approx y_{i+1}$ given the value of $y_i \approx y(t_i)$. Using Euler's method, we can find numerical approximations for $y(t)$ for any value of $t > 0$.

3.1.2 Explicit versus implicit schemes

In the previous section, we developed Euler's method for a simple initial value problem. Suppose one has the slightly different problem where the source term f is now a function of both t and y.

$$\begin{cases} y'(t) = f(t, y) \\ y(0) = y_0 \end{cases}$$

A similar argument as before will now lead us to the expression for y_{i+1}

$$y_{i+1} \approx y_i + f(t_i, y_i) \cdot h,$$

where y_{i+1} is *explicitly* written as a sum of terms that depend only on time t_i. Schemes such as this are called **explicit**. Had we used the approximation

$$y(t-h) \approx y(t) - y'(t) \cdot h$$

instead, we would arrive at the slightly different expression for y_{i+1}

$$y_{i+1} \approx y_i + f(t_{i+1}, y_{i+1}) \cdot h,$$

where the term y_{i+1} appears in both sides of the equation and no explicit formula for y_{i+1} is possible in general. Schemes such as this are called **implicit**. In the general case, each step in time in an implicit method requires solving the expression above for y_{i+1} using a root finding technique such as **Newton's method** or other fixed point iteration methods.

Despite being easier to compute, explicit methods are generally known to be numerically unstable for a large range of equations (especially so-called **stiff problems**), making them unusable for most practical situations. Implicit methods, on the other hand, are typically both more computationally intensive and more numerically stable, which makes them more commonly used. An important measure of numerical stability for finite difference methods is **A-stability**, where one tests the stability of the method for the (linear) test equation $y'(t, y) = \lambda y(t)$, with $\lambda < 0$. While the implicit Euler method is stable for all values of $h > 0$ and $\lambda < 0$, the explicit Euler method is stable only if $|1 + h\lambda| < 1$, which may require using a small value for h if the absolute value of λ is high. Of course, all other things being equal, a small value for h is undesirable since it means a finer grid is required, which in turn makes the numerical method more computationally expensive.

3.1.3 Finite difference methods for PDEs

In the previous section, we focused our discussion on methods for numerically solving ODEs. However, finite difference methods can be used to solve PDEs as well and the concepts presented above can also be applied in PDEs solving methods. Consider the boundary problem for the heat equation in one spatial dimension, which describes the dynamics of heat transfer in a rod of length l:

$$\begin{cases} \partial_t u = \alpha^2 \cdot \partial_{xx} u \\ u(0, x) = u_0(x) \\ u(t, 0) = u(t, l) = 0 \end{cases}$$

We could approximate the differential operators in the equation above using a **forward difference** operator for the partial derivative in time and a **second-order central difference** operator for the partial derivative in space. Using the notation

$u_{i,j} \approx u(t_i, x_j)$, with $t_{i+1} = t_i + k$ and $x_{j+1} = x_j + h$, we can rewrite the equation above as a system of linear equations

$$\frac{u_{i+1,j} - u_{i,j}}{k} = \alpha^2 \left(\frac{u_{i,j-1} - 2u_{i,j} + u_{i,j+1}}{h^2} \right),$$

where $i = 1, 2, \ldots, N$ and $j = 1, 2, \ldots, N$, assuming we are using the same number of discrete points on both dimensions. In this two dimensional example, the points (t_i, x_j) form a two dimensional grid of size $\mathcal{O}\left(N^2\right)$. For a d-dimensional problem, a d-dimensional grid with size $\mathcal{O}\left(N^d\right)$ would be required. In practice, the exponential growth of the grid in the number of dimensions rapidly makes the method unmanageable, even for $d = 4$. *This is an important shortcoming of finite difference methods in general.*

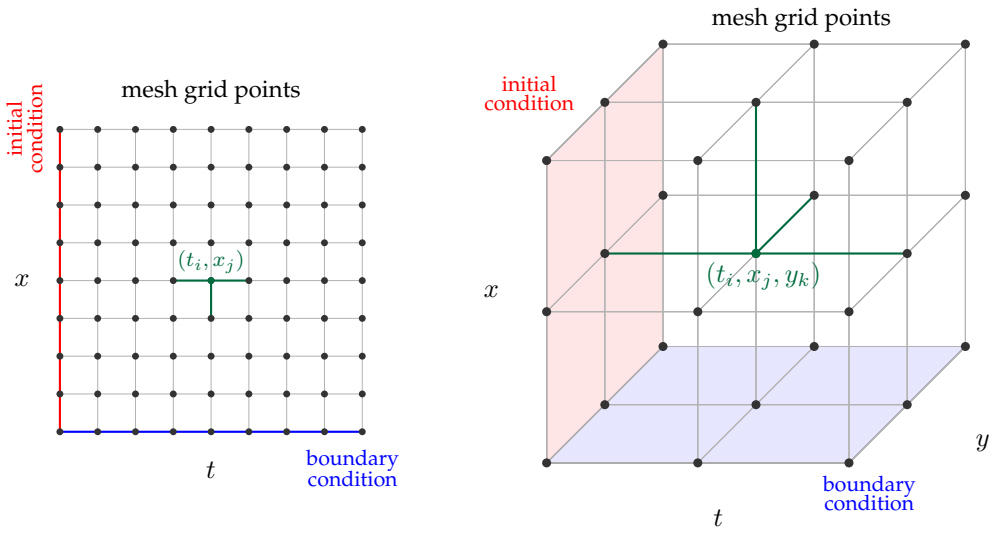

Figure 3.1: *Illustration of finite difference methods for solving PDEs in two (left) and three (right) dimensions. The known function value on the boundaries is combined with finite differences to solve for the value of function on a grid in the interior of the region where it is defined.*

The scheme developed above is known as the **forward difference** method or **FTCS (forward in time, central in space)**. It is easy to verify that this scheme is explicit in time, since we can write the $u_{i+1,\cdot}$ terms as a linear combination of previously computed $u_{i,\cdot}$ terms. The number of operations necessary to advance each step in time with this method should be $\mathcal{O}\left(N^2\right)$. Unfortunately, this scheme is also known to be unstable if h and k do not satisfy the inequality $\alpha^2 \frac{k}{h^2} \leq \frac{1}{2}$.

Alternatively, we could apply the **Backward Difference** method or **BTCS (backward in time, central in space)** using the following equations:

$$\frac{u_{i+1,j} - u_{i,j}}{k} = \alpha^2 \left(\frac{u_{i+1,j-1} - 2u_{i+1,j} + u_{i+1,j+1}}{h^2} \right).$$

This scheme is implicit in time since it is not possible to write the $u_{i+1,\cdot}$ terms as a function of just the previously computed $u_{i,\cdot}$ terms. In fact, each step in time requires solving system of linear equations of size $\mathcal{O}\left(N^2\right)$. The number of operations necessary to advance each step in time with this method is $\mathcal{O}\left(N^3\right)$ when using methods such as Gaussian elimination to solve the linear system. On the other hand, this scheme is also known to be *unconditionally stable*, independently of the values for h and k.

3.1.4 Higher order methods

All numerical methods for solving PDEs have errors due to many sources of inaccuracies. For instance, **rounding error** is related to the floating point representation of real numbers. Another important category of error is **truncation error**, which can be understood as the error due to the Taylor series expansion truncation. Finite difference methods are usually classified by their respective truncation errors.

All finite methods discussed so far are low order methods. For instance, the Euler's methods (both explicit and implicit varieties) are **first-order methods**, which means that the global truncation error is proportional to h, the discretization granularity. However, a number of alternative methods have lower truncation errors. For example, the **Runge-Kutta 4th-order method**, with a global truncation error proportional to h^4, is widely used, being the most known method of a family of finite difference methods, which cover even 14th-order methods. Many Runge-Kutta methods are specially suited for solving stiff problems.

3.2 Galerkin methods

In finite difference methods, we approximate the continuous differential operator by a discrete difference operator in order to obtain a numerical approximation of the function that satisfies the PDE. The function's domain (or a portion of it) must also be discretized so that numerical approximations for the value of the solution can be computed at the points of the so defined spatial-temporal grid. Furthermore, the value of the function on off-grid points can also be approximated by techniques such as interpolation.

Galerkin methods take an alternative approach: given a finite set of basis functions on the same domain, the goal is to find a linear combination of the basis functions that approximates the solution of the PDE on the domain of interest. This problem translates into a variational problem where one is trying to find maxima or minima

of functionals.

More precisely, suppose we are trying to solve the equation $F(x) = y$ for x, where x and y are members of spaces of functions X and Y respectively and that $F : X \to Y$ is a (possibly non-linear) functional. Suppose also that $\{\phi_i\}_{i=1}^{\infty}$ and $\{\psi_j\}_{j=1}^{\infty}$ form linearly independent bases for X and Y. According to the Galerkin method, an approximation for x could be given by

$$x_n = \sum_{i=1}^{n} \alpha_i \phi_i$$

where the α_i coefficients satisfy the equations

$$\left\langle F\left(\sum_{i=1}^{n} \alpha_i \phi_i\right), \psi_j \right\rangle = \langle y, \psi_j \rangle ,$$

for $j = 1, 2, \ldots, n$.[1] Since the inner products above usually involve non-trivial integrals, one should carefully choose the bases to ensure the equations are more manageable.

3.3 Finite Element Methods

Finite element methods can be understood as a special case of Galerkin methods. Notice that in the general case presented above, the approximation x_n may not be well-posed, in the sense that the system of equations for α_i may have no solution or it may have multiple solutions depending on the value of n. Additionally, depending on the choice of ϕ_i and ψ_j, x_n may not converge to x as $n \to \infty$. Nevertheless, one could discretize the domain in small enough regions (called elements) so that the approximation is locally satisfactory in each region. Adding boundary consistency constraints for each region intersection (as well as for the outer boundary conditions given by the problem definition) and solving for the whole domain of interest, one can come up with a globally fair numerical approximation for the solution to the PDE.

In practice, the domain is typically divided in triangles or quadrilaterals (two-dimensional case), tetrahedra (three-dimensional case) or more general geometrical shapes in higher dimensions in a process known as **triangulation**. Typical choices for ϕ_i and ψ_j are such that the inner product equations above reduce to a system of algebraic equations for steady state problems or a system of ODEs in the case of time-dependent problems. If the PDE is linear, those systems will be linear

[1] https://www.encyclopediaofmath.org/index.php/Galerkin_method

as well, and they can be solved using methods such as Gaussian elimination or iterative methods such as Jacobi or Gauss-Seidel. If the PDE is not linear, one may need to solve systems of nonlinear equations, which are generally more computationally expensive. One of the major advantages of the finite element methods over finite difference methods, is that finite elements can effortlessly handle complex boundary geometries, which typically arise in physical or engineering problems, whereas this may be very difficult to achieve with finite difference algorithms.

3.4 Monte Carlo Methods

One of the more fascinating aspects of PDEs is how they are intimately related to stochastic processes. This is best exemplified by the **Feynman-Kac theorem**, which can be viewed in two ways:

- It provides a solution to a certain class of linear PDEs, written in terms of an expectation involving a related stochastic process;
- It gives a means by which certain expectations can be computed by solving an associated PDE.

For our purposes, we are interested in the first of these two perspectives.

The theorem is stated as follows: the solution to the partial differential equation

$$\begin{cases} \partial_t h + a(t,x) \cdot \partial_x h + \frac{1}{2} b(t,x)^2 \cdot \partial_{xx} h + g(t,x) \cdot h(t,x) \;=\; c(t,x) \cdot h(t,x) \\ h(T,x) \;=\; H(x) \end{cases}$$

admits a stochastic representation given by

$$h(t,x) \;=\; \mathbb{E}^{\mathbb{P}^*}_{t,x} \left[\int_t^T e^{-\int_t^u c(s,X_s)\, ds} \cdot g(u, X_u)\, du + H(X_T) \cdot e^{-\int_t^T c(s,X_s)\, ds} \right]$$

where $\mathbb{E}_{t,x}[\ \cdot\ ] = \mathbb{E}[\ \cdot\ |X_t = x]$ and the process $X = (X_t)_{t \geq 0}$ satisfies the SDE:

$$dX_t \;=\; a(t, X_t)\, dt + b(t, X_t)\, dW_t^{\mathbb{P}^*}$$

where $W^{\mathbb{P}^*} = \left(W_t^{\mathbb{P}^*}\right)_{t \geq 0}$ is a standard Brownian motion under the probability measure $\mathbb{P}^*$. This representation suggests the use of **Monte Carlo methods** to solve for unknown function h. Monte Carlo methods are a class of numerical techniques based on simulating random variables used to solve a range of problems, such as numerical integration and optimization.

Returning to the theorem, let us now discuss its statement:

- When confronted with a PDE of the form above, we can define a (fictitious) process X with drift and volatility given by the processes $a(t, X_t)$ and $b(t, X_t)$, respectively.
- Thinking of c as a "discount factor," we then consider the conditional expectation of the discounted terminal condition $H(X_T)$ and the running term $g(t, X_t)$ given that the value of X at time t is equal to a known value, x. Clearly, this conditional expectation is a function of t and x; for every value of t and x we have some conditional expectation value.
- This function (the conditional expectation as a function of t and x) is precisely the solution to the PDE we started with and can be estimated via Monte Carlo simulation of the process X.

A class of Monte Carlo methods have also been developed for nonlinear PDEs, but this is beyond the scope of this work.

Chapter 4

An Introduction to Deep Learning

The tremendous strides made in computing power and the explosive growth in data collection and availability in recent decades has coincided with an increased interest in the field of **machine learning** (ML). This has been reinforced by the success of machine learning in a wide range of applications ranging from image and speech recognition, medical diagnostics, email filtering, fraud detection and many more.

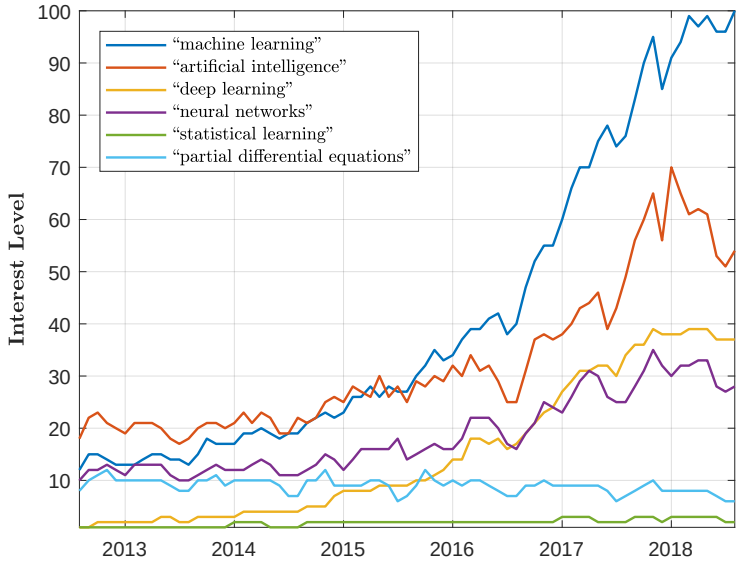

Figure 4.1: *Google search frequency for various terms. A value of 100 is the peak popularity for the term; a value of 50 means that the term is half as popular.*

As the name suggests, the term machine learning refers to computer algorithms that learn from data. The term "learn" can have several meanings depending on

the context, but the the common theme is the following: a computer is faced with a task and an associated performance measure, and its goal is to improve its performance in this task with experience which comes in the form of examples and data.

ML naturally divides into two main branches. **Supervised learning** refers to the case where the data points include a label or target and tasks involve predicting these labels/targets (i.e. classification and regression). **Unsupervised learning** refers to the case where the dataset does not include such labels and the task involves learning a useful structure that relates the various variables of the input data (e.g. clustering, density estimation). Other branches of ML, including semi-supervised and reinforcement learning, also receive a great deal of research attention at present. For further details the reader is referred to Bishop (2006) or Goodfellow et al. (2016).

An important concept in machine learning is that of **generalization** which is related to the notions of **underfitting** and **overfitting**. In many ML applications, the goal is to be able to make meaningful statements concerning data that the algorithm has not encountered - that is, to generalize the model to unseen examples. It is possible to calibrate an assumed model "too well" to the training data in the sense that the model gives misguided predictions for new data points; this is known as overfitting. The opposite case is underfitting, where the model is not fit sufficiently well on the input data and consequently does not generalize to test data. Striking a balance in the trade-off between underfitting and overfitting, which itself can be viewed as a tradeoff between bias and variance, is crucial to the success of a ML algorithm.

On the theoretical side, there are a number of interesting results related to ML. For example, for certain tasks and hypothesized models it may be possible to obtain the minimal sample size to ensure that the training error is a faithful representation of the generalization error with high confidence (this is known as **Probably Approximately Correct (PAC) learnability**). Another result is the **no-free-lunch theorem**, which implies that there is no universal learner, i.e. that every learner has a task on which it fails even though another algorithm can successfully learn the same task. For an excellent exposition of the theoretical aspects of machine learning the reader is referred to Shalev-Shwartz and Ben-David (2014).

4.1 Neural Networks and Deep Learning

Neural networks are machine learning models that have received a great deal of attention in recent years due to their success in a number of different applications. The typical way of motivating the approach behind neural network models is to compare the way they operate to the human brain. The building blocks of the

brain (and neural networks) are basic computing devices called **neurons** that are connected to one another by a complex communication network. The communication links cause the activation of a neuron to activate other neurons it is connected to. From the perspective of learning, training a neural network can be thought of as determining which neurons "fire" together.

Mathematically, a neural network can be defined as a directed graph with vertices representing neurons and edges representing links. The input to each neuron is a function of a weighted sum of the output of all neurons that are connected to its incoming edges. There are many variants of neural networks which differ in architecture (how the neurons are connected); see Figure 4.2. The simplest of these forms is the **feedforward neural network**, which is also referred to as a **multilayer perceptron** (MLP).

MLPs can be represented by a directed acyclic graph and as such can be seen as feeding information forward. Usually, networks of this sort are described in terms of layers which are chained together to create the output function, where a layer is a collection of neurons that can be thought of as a unit of computation. In the simplest case, there is a single **input layer** and a single **output layer**. In this case, output j (represented by the jth neuron in the output layer), is connected to the input vector $\mathbf{x}$ via a biased weighted sum and an **activation function** ϕ_j:

$$y_j = \phi_j \left(b_j + \sum_{i=1}^{d} w_{i,j} \mathbf{x}_i \right)$$

It is also possible to incorporate additional **hidden layers** between the input and output layers. For example, with one hidden layer the output would become:

$$y_k = \underbrace{\phi \left[b_k^{(2)} + \sum_{i=1}^{m_2} w_{j,k}^{(2)} \cdot \underbrace{\psi \left(b_j^{(1)} + \sum_{i=1}^{m_1} w_{i,j}^{(1)} \mathbf{x}_j \right)}_{\text{input layer to hidden layer}} \right]}_{\text{hidden layer to output layer}}$$

where $\phi, \psi : \mathbb{R} \to \mathbb{R}$ are nonlinear activation functions for each layer and the bracketed superscripts refer to the layer in question. We can visualize an extension of this the process as a simple application of the chain rule, e.g.

$$f(\mathbf{x}) = \psi_d(\cdots \psi_2(\psi_1(\mathbf{x})))$$

Here, each layer of the network is represented by a function ψ_i, incorporating the weighted sums of previous inputs and activations to connected outputs. The number of layers in the graph is referred to as the **depth** of the neural network and the

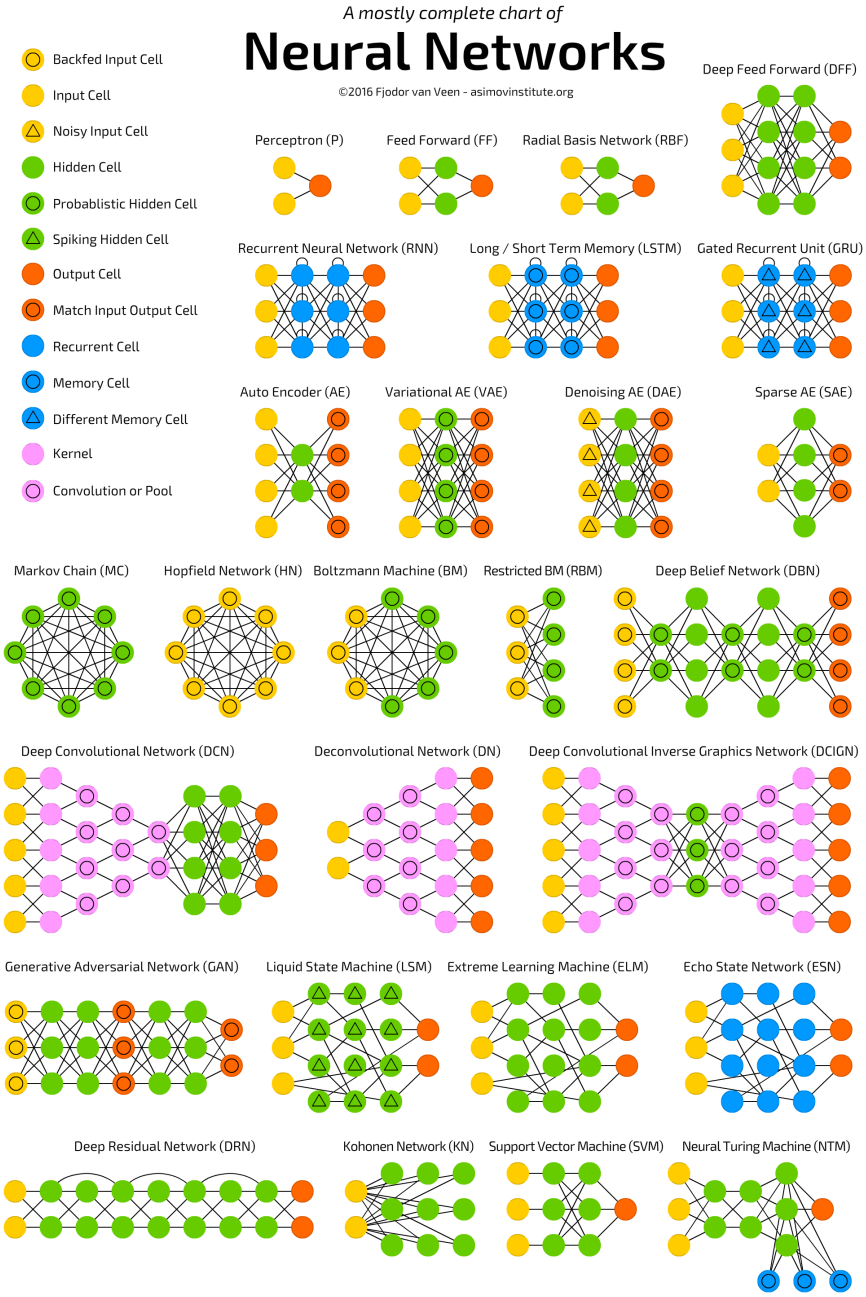

Figure 4.2: *Neural network architectures. Source: "Neural Networks 101" by Paul van der Laken (https://paulvanderlaken.com/2017/10/16/neural-networks-101)*

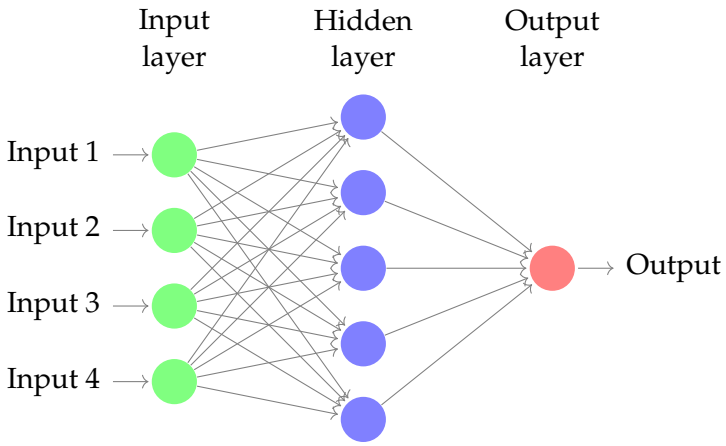

Figure 4.3: *Feedforward neural network with one hidden layer.*

number of neurons in a layer represents the **width** of that particular layer; see Figure 4.3.

The term "deep" neural network and **deep learning** refer to the use of neural networks with many hidden layers in ML problems. One of the advantages of adding hidden layers is that depth can exponentially reduce the computational cost in some applications and exponentially decrease the amount of training data needed to learn some functions. This is due to the fact that some functions can be represented by smaller deep networks compared to wide shallow networks. This decrease in model size leads to improved statistical efficiency.

It is easy to imagine the tremendous amount of flexibility and complexity that can be achieved by varying the structure of the neural network. One can vary the depth or width of the network, or have varying activation functions for each layer or even each neuron. This flexibility can be used to achieve very strong results but can lead to opacity that prevents us from understanding why any strong results are being achieved.

Next, we turn to the question of how the parameters of the neural network are estimated. To this end, we must first define a **loss function**, $L(\boldsymbol{\theta}; \mathbf{x}, \mathbf{y})$, which will determine the performance of a given parameter set $\boldsymbol{\theta}$ for the neural network consisting of the weights and bias terms in each layer. The goal is to find the parameter set that minimizes our loss function. The challenge is that the highly nonlinear nature of neural networks can lead to non-convexities in the loss function. Non-convex optimization problems are non-trivial and often we cannot guarantee that a candidate solution is a global optimizer.

4.2 Stochastic Gradient Descent

The most commonly used approach for estimating the parameters of a neural network is based on **gradient descent** which is a simple methodology for optimizing a function. Given a function $f : \mathbb{R}^d \to \mathbb{R}$, we wish to determine the value of $\boldsymbol{x}$ that achieves the minimum value of f. To do this, we begin with an initial guess $\boldsymbol{x}_0$ and compute the gradient of f at this point. This gives the direction in which the largest increase in the function occurs. To minimize the function we move in the opposite direction, i.e. we iterate according to:

$$\boldsymbol{x}_n = \boldsymbol{x}_{n-1} - \eta \cdot \nabla_{\boldsymbol{x}} f\left(\boldsymbol{x}_{n-1}\right)$$

where η is the step size known as the **learning rate**, which can be constant or decaying in n. The algorithm converges to a critical point when the gradient is equal to zero, though it should be noted that this is not necessarily a global minimum. In the context of neural networks, we would compute the derivatives of the loss functional with respect to the parameter set $\boldsymbol{\theta}$ (more on this in the next section) and follow the procedure outlined above.

One difficulty with the use of gradient descent to train neural networks is the computational cost associated with the procedure when training sets are large. This necessitates the use of an extension of this algorithm known as **stochastic gradient descent** (SGD). When the loss function we are minimizing is additive, it can be written as:

$$\nabla L(\boldsymbol{\theta}; \mathbf{x}, \mathbf{y}) = \frac{1}{m} \sum_{i=1}^{m} \nabla_{\boldsymbol{\theta}} L_i \left(\boldsymbol{\theta}; \mathbf{x}^{(i)}, \mathbf{y}^{(i)}\right)$$

where m is the size of the training set and L_i is the per-example loss function. The approach in SGD is to view the gradient as an expectation and approximate it with a random subset of the training set called a **mini-batch**. That is, for a fixed mini-batch of size m' the gradient is estimated as:

$$\nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta}; \mathbf{x}, \mathbf{y}) \approx \frac{1}{m'} \nabla_{\boldsymbol{\theta}} \sum_{i=1}^{m'} L_i \left(\boldsymbol{\theta}; \mathbf{x}^{(i)}, \mathbf{y}^{(i)}\right)$$

This is followed by taking the usual step in the opposite direction (steepest descent).

4.3 Backpropagation

The stochastic gradient descent optimization approach described in the previous section requires repeated computation of the gradients of a highly nonlinear function. **Backpropagation** provides a computationally efficient means by which this

can be achieved. It is based on recursively applying the chain rule and on defining computational graphs to understand which computations can be run in parallel.

As we have seen in previous sections, a feedforward neural network can be thought of as receiving an input $\mathbf{x}$ and computing an output y by evaluating a function defined by a sequence of compositions of simple functions. These simple functions can be viewed as operations between nodes in the neural network graph. With this in mind, the derivative of y with respect to $\mathbf{x}$ can be computed analytically by repeated applications of the chain rule, given enough information about the operations between nodes. The backpropagation algorithm traverses the graph, repeatedly computing the chain rule until the derivative of the output y with respect to $\mathbf{x}$ is represented symbolically via a second computational graph; see Figure 4.4.

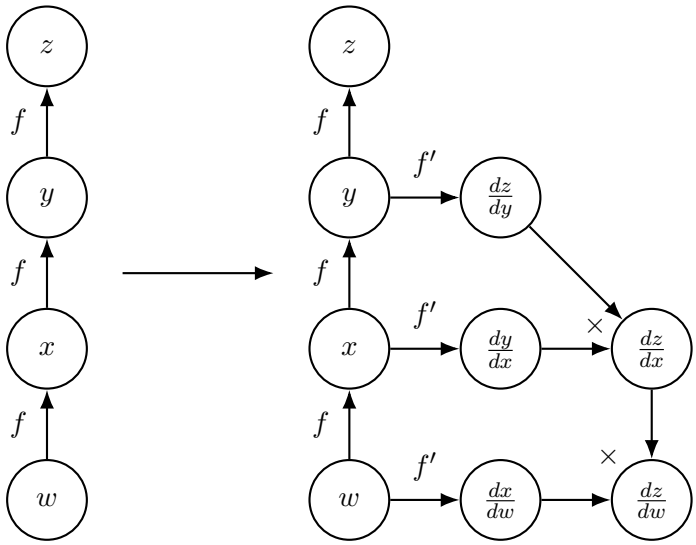

Figure 4.4: *Visualization of backpropagation algorithm via computational graphs. The left panel shows the composition of functions connecting input to output; the right panel shows the use of the chain rule to compute the derivative. Source: Goodfellow et al. (2016)*

The two main approaches for computing the derivatives in the computational graph is to input a numerical value then compute the derivatives at this value, returning a number as done in PyTorch (`pytorch.org`), or to compute the derivatives of a symbolic variable, then store the derivative operations into new nodes added to the graph for later use as done in TensorFlow (`tensorflow.org`). The advantage of the latter approach is that higher-order derivatives can be computed from this extended graph by running backpropagation again.

The backpropagation algorithm takes at most $\mathcal{O}\left(n^2\right)$ operations for a graph with n nodes, storing at most $\mathcal{O}\left(n^2\right)$ new nodes. In practice, most feedforward neural networks are designed in a chain-like way, which in turn reduces the number of operations and new storages to $\mathcal{O}\left(n\right)$, making derivatives calculations a relatively cheap operation.

4.4 Summary

In summary, training neural networks is broadly composed of three ingredients:

1. Defining the architecture of the neural network and a loss function, also known as the **hyperparameters** of the model;
2. Finding the loss minimizer using stochastic gradient descent;
3. Using backpropagation to compute the derivatives of the loss function.

This is presented in more mathematical detail in Figure 4.5.

---

1. Define the architecture of the neural network by setting its depth (number of layers), width (number of neurons in each layer) and activation functions
2. Define a loss functional $L(\boldsymbol{\theta}; \mathbf{x}, \mathbf{y})$, mini-batch size m' and learning rate η
3. Minimize the loss functional to determine the optimal $\boldsymbol{\theta}$:
   (a) `Initialize the parameter set,` $\boldsymbol{\theta}_0$
   (b) `Randomly sample a mini-batch of` m' `training examples` $\left(\mathbf{x}^{(i)}, \mathbf{y}^{(i)}\right)$
   (c) `Compute the loss functional for the sampled mini-batch` $L\left(\boldsymbol{\theta}_i; \mathbf{x}^{(i)}, \mathbf{y}^{(i)}\right)$
   (d) `Compute the gradient` $\nabla_{\boldsymbol{\theta}} L\left(\boldsymbol{\theta}_i; \mathbf{x}^{(i)}, \mathbf{y}^{(i)}\right)$ `using backpropagation`
   (e) `Use the estimated gradient to update` θ_i `based on SGD:`

$$\boldsymbol{\theta}_{i+1} = \boldsymbol{\theta}_i - \eta \cdot \nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta}_i; \mathbf{x}^{(i)}, \mathbf{y}^{(i)})$$

   (f) `Repeat steps (b)-(e) until` $\|\boldsymbol{\theta}_{i+1} - \boldsymbol{\theta}_i\|$ `is small.`

Figure 4.5: *Parameter estimation procedure for neural networks.*

4.5 The Universal Approximation Theorem

An important theoretical result that sheds some light on why neural networks perform well is the **universal approximation theorem**, see Cybenko (1989) and Hornik (1991). In simple terms, this result states that any continuous function defined on a compact subset of $\mathbb{R}^n$ can be approximated arbitrarily well by a feedforward network with a single hidden layer.

Mathematically, the statement of the theorem is as follows: let ϕ be a nonconstant, bounded, monotonically-increasing continuous function and let I_m denote the m-dimensional unit hypercube. Then, given any $\epsilon > 0$ and any function f defined on I_m, there exists $N, v_i, b_i, \mathbf{w}$ such that the approximation function:

$$F(\boldsymbol{x}) = \sum_{i=1}^{N} v_i \phi \left(\mathbf{w} \cdot \boldsymbol{x} + \mathbf{b}_i \right)$$

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

A remarkable aspect of this result is the fact that the activation function is independent of the function we wish to approximate! However, it should be noted that the theorem makes no statement on the number of neurons needed in the hidden layer to achieve the desired approximation error, nor whether the estimation of the parameters of this network is even feasible.

4.6 Other Topics

4.6.1 Adaptive Momentum

Recall that the stochastic gradient descent algorithm is parametrized by a learning rate η which determines the step size in the direction of steepest descent given by the gradient vector. In practice, this value should decrease along successive iterations of the SGD algorithm for the network to be properly trained. For a network's parameter set to be properly optimized, an appropriately chosen learning rate schedule is in order, as it ensures that the excess error is decreasing in each iteration. Furthermore, this learning rate schedule can depend on the nature of the problem at hand.

For the reasons discussed in the last paragraph, a number of different algorithms have been developed to find some heuristic capable of guiding the selection of an effective sequence of learning rate parameters. Inspired by physics, many of these algorithms interpret the gradient as a velocity vector, that is, the direction and speed at which the parameters move through the parameter space. **Momentum algorithms**, for example, calculate the next velocity as a weighted sum of the

gradient from the last iteration and the newly calculated one. This helps minimize instabilities caused by the high sensitivity of the loss function with respect to some directions of the parameter space, at the cost of introducing two new parameters, namely a decay **factor**, and an initialization parameter η_0. Assuming these sensitivities are axis-dependent, we can apply different learning rate schedules to each direction and adapt them throughout the training session.

The work of Kingma and Ba (2014) combines the ideas discussed in this section in a single framework referred to as **Adaptative Momentum** (Adam). The main idea is to increase/decrease the learning rates based on the past magnitudes of the partial derivatives for a particular direction. Adam is regarded as being robust to its hyperparameter values.

4.6.2 The Vanishing Gradient Problem

In our analysis of neural networks, we have established that the addition of layers to a network's architecture can potentially lead to great increases in its performance: increasing the number of layers allows the network to better approximate increasingly more complicated functions in a more efficient manner. In a sense, the success of deep learning in current ML applications can be attributed to this notion.

However, this improvement in power can be counterbalanced by the **Vanishing Gradient Problem**: due to the the way gradients are calculated by backpropagation, the deeper a network is the smaller its loss function's derivative with respect to weights in early layers becomes. At the limit, depending on the activation function, the gradient can underflow in a manner that causes weights to not update correctly.

Intuitively, imagine we have a deep feedforward neural network consisting of n layers. At every iteration, each of the network's weights receives an update that is proportional to the gradient of the error function with respect to the current weights. As these gradients are calculated using the chain rule through backpropagation, the further back a layer is, the more it is multiplied by an already small gradient.

4.6.3 Long-Short Term Memory and Recurrent Neural Networks

Applications with time or positioning dependencies, such as speech recognition and natural language processing, where each layer of the network handles one time/positional step, are particularly prone to the vanishing gradient problem. In particular, the vanishing gradient might mask long term dependencies between

observation points far apart in time/space.

Colloquially, we could say that the neural network is not able to accurately remember important information from past layers. One way of overcoming this difficulty is to incorporate a notion of memory for the network, training it to learn which inputs from past layers should flow through the current layer and pass on to the next, i.e. how much information should be "remembered" or "forgotten." This is the intuition behind **long-short term memory** (LSTM) networks, introduced by Hochreiter and Schmidhuber (1997).

LSTM networks are a class of **recurrent neural networks** (RNNs) that consists of layers called **LSTM units**. Each layer is composed of a **memory cell**, an **input gate**, an **output gate** and a **forget gate** which regulates the flow of information from one layer to another and allows the network to learn the optimal remembering/forgetting mechanism. Mathematically, some fraction of the gradients from past layers are able to pass through the current layer directly to the next. The magnitude of the gradient that passes through the layer unchanged (relative to the portion that is transformed) as well as the discarded portion, is also learned by the network. This embeds the memory aspect in the architecture of the LSTM allowing it to circumvent the vanishing gradient problem and learn long-term dependencies; refer to Figure 4.6 for a visual representation of a single LSTM unit.

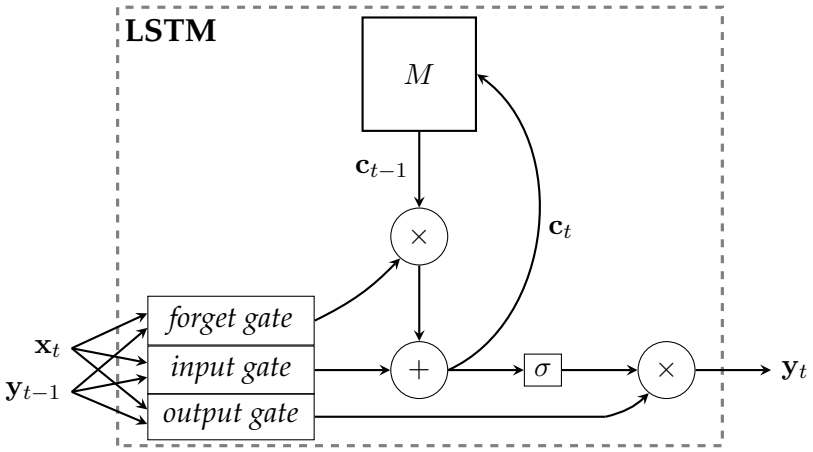

Figure 4.6: *Architecture of an LSTM unit: a new input $\mathbf{x}_t$ and output of the last unit $\mathbf{y}_{t-1}$ are combined with past memory information $\mathbf{c}_{t-1}$ to produce new output $\mathbf{y}_t$ and store new memory information $\mathbf{c}_t$. Source: "A trip down long-short memory lane" by Peter Veličković (https://www.cl.cam.ac.uk/~pv273/slides/LSTMslides.pdf)*

Inspired by the effectiveness of LSTM networks and given the rising importance of deep architectures in modern ML, Srivastava et al. (2015) devised a network that allows gradients from past layers to flow through the current layer. **Highway networks** use the architecture of LSTMs for problems where the data is not sequential. By adding an "information highway," which allows gradients from early layers to flow unscathed through intermediate layers to the end of the network, the authors are able to train incredibly deep networks, with depth as high as a 100 layers without vanishing gradient issues.

Chapter 5

The Deep Galerkin Method

5.1 Introduction

We now turn our attention to the application of neural networks to finding solutions to PDEs. As discussed in Chapter 3, numerical methods that are based on grids can fail when the dimensionality of the problem becomes too large. In fact, the number of points in the mesh grows exponentially in the number of dimensions which can lead to computational intractability. Furthermore, even if we were to assume that the computational cost was manageable, ensuring that the grid is set up in a way to ensure stability of the finite difference approach can be cumbersome.

With this motivation, Sirignano and Spiliopoulos (2018) propose a *mesh-free* method for solving PDEs using neural networks. The **Deep Galerkin Method** (DGM) approximates the solution to the PDE of interest with a deep neural network. With this parameterization, a loss function is set up to penalize the fitted function's deviations from the desired differential operator and boundary conditions. The approach takes advantage of computational graphs and the backpropagation algorithm discussed in the previous chapter to efficiently compute the differential operator while the boundary conditions are straightforward to evaluate. For the training data, the network uses points randomly sampled from the region where the function is defined and the optimization is performed using stochastic gradient descent.

The main insight of this approach lies in the fact that the training data consists of randomly sampled points in the function's domain. By sampling mini-batches from different parts of the domain and processing these small batches sequentially, the neural network "learns" the function without the computational bottleneck present with grid-based methods. This circumvents the curse of dimensionality which is encountered with the latter approach.

5.2 Mathematical Details

The form of the PDEs of interest are generally described as follows: let u be an unknown function of time and space defined on the region $[0,T]\times\Omega$ where $\Omega\subset\mathbb{R}^d$, and assume that u satisfies the PDE:

$$\begin{cases} (\partial_t+\mathcal{L})\,u(t,\boldsymbol{x})=0, & (t,\boldsymbol{x})\in[0,T]\times\Omega \\ u(0,\boldsymbol{x})=u_0(\boldsymbol{x}), & \boldsymbol{x}\in\Omega \qquad \text{(initial condition)} \\ u(t,\boldsymbol{x})=g(t,\boldsymbol{x}), & (t,\boldsymbol{x})\in[0,T]\times\partial\Omega \qquad \text{(boundary condition)} \end{cases}$$

The goal is to approximate u with an approximating function $f(t,\boldsymbol{x};\boldsymbol{\theta})$ given by a deep neural network with parameter set $\boldsymbol{\theta}$. The loss functional for the associated training problem consists of three parts:

1. A measure of how well the approximation satisfies the **differential operator**:

$$\left\|\,(\partial_t+\mathcal{L})\,f(t,\boldsymbol{x};\boldsymbol{\theta})\right\|^2_{[0,T]\times\Omega,\,\nu_1}$$

   Note: parameterizing f as a neural network means that the differential operator can be computed easily using backpropagation.

2. A measure of how well the approximation satisfies the **boundary condition**:

$$\left\|f(t,\boldsymbol{x};\boldsymbol{\theta})-g(t,\boldsymbol{x})\right\|^2_{[0,T]\times\partial\Omega,\,\nu_2}$$

3. A measure of how well the approximation satisfies the **initial condition**:

$$\left\|f(0,\boldsymbol{x};\boldsymbol{\theta})-u_0(\boldsymbol{x})\right\|^2_{\Omega,\,\nu_3}$$

In all three terms above the error is measured in terms of L^2-norm, i.e. using $\|h(y)\|^2_{\mathcal{Y},\nu}=\int_{\mathcal{Y}}|h(y)|^2\nu(y)dy$ with $\nu(y)$ being a density defined on the region $\mathcal{Y}$. Combining the three terms above gives us the cost functional associated with training the neural network:

$$L(\boldsymbol{\theta})=\underbrace{\left\|\,(\partial_t+\mathcal{L})\,f(t,\boldsymbol{x};\boldsymbol{\theta})\right\|^2_{[0,T]\times\Omega,\nu_1}}_{\text{differential operator}}+\underbrace{\left\|f(t,\boldsymbol{x};\boldsymbol{\theta})-g(t,\boldsymbol{x})\right\|^2_{[0,T]\times\partial\Omega,\nu_2}}_{\text{boundary condition}}+\underbrace{\left\|f(0,\boldsymbol{x};\boldsymbol{\theta})-u_0(\boldsymbol{x})\right\|^2_{\Omega,\nu_3}}_{\text{initial condition}}$$

The next step is to minimize the loss functional using stochastic gradient descent. More specifically, we apply the algorithm defined in Figure 5.1. The description given in Figure 5.1 should be thought of as a general outline as the algorithm should be modified according to the particular nature of the PDE being considered.

---

1. Initialize the parameter set $\boldsymbol{\theta}_0$ and the learning rate α_n.
2. Generate random samples from the domain's interior and time/spatial boundaries, i.e.
   - `Generate` (t_n, x_n) `from` $[0,T] \times \Omega$ `according to` ν_1
   - `Generate` (τ_n, z_n) `from` $[0,T] \times \partial\Omega$ `according to` ν_2
   - `Generate` w_n `from` Ω`, according to` ν_3
3. Calculate the loss functional for the current mini-batch (the randomly sampled points $s_n = \{(t_n, x_n), (\tau_n, z_n), w_n\}$):
   - `Compute` $L_1(\boldsymbol{\theta}_n; t_n, x_n) = \left((\partial_t + \mathcal{L})\, f(\boldsymbol{\theta}_n; t_n, x_n)\right)^2$
   - `Compute` $L_2(\boldsymbol{\theta}_n; \tau_n, z_n) = \left(f(\tau_n, z_n) - g(\tau_n, z_n)\right)^2$
   - `Compute` $L_3(\boldsymbol{\theta}_n; w_n) = \left(f(0, w_n) - u_0(w_n)\right)^2$
   - `Compute` $L(\boldsymbol{\theta}_n; s_n) = L_1(\boldsymbol{\theta}_n; t_n, x_n) + L_2(\boldsymbol{\theta}_n; \tau_n, z_n) + L_3(\boldsymbol{\theta}_n; z_n)$
4. Take a descent step at the random point s_n with Adam-based learning rates:

$$\theta_{n+1} = \theta_n - \alpha_n \nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta}_n; s_n)$$

5. Repeat steps (2)-(4) until $\|\theta_{n+1} - \theta_n\|$ is small.

Figure 5.1: *Deep Galerkin Method (DGM) algorithm.*

It is important to notice that the problem described here is strictly an optimization problem. This is unlike typical machine learning applications where we are concerned with issues of underfitting, overfitting and generalization. Typically, arriving at a parameter set where the loss function is equal to zero would not be desirable as it suggests some form of overfitting. However, in this context a neural network that achieves this is the goal as it would be the solution to the PDE at hand. The only case where generalization becomes relevant is when we are unable to sample points everywhere within the region where the function is defined, e.g. for functions defined on unbounded domains. In this case, we would be interested in examining how well the function satisfies the PDE in those unsampled regions. The results in the next chapter suggest that this generalization is often very poor.

5.3 A Neural Network Approximation Theorem

Theoretical motivation for using neural networks to approximate solutions to PDEs is given as an elegant result in Sirignano and Spiliopoulos (2018) which is similar in spirit to the Universal Approximation Theorem. More specifically, it is shown that *deep neural network approximators converge to the solution of a class of quasilinear parabolic PDEs as the number of hidden layers tends to infinity*. To state the result in more precise mathematical terms, define the following:

- $L(\boldsymbol{\theta})$, the loss functional measuring the neural network's fit to the differential operator and boundary/initial/terminal conditions;
- $\mathfrak{C}^n$, the class of neural networks with n hidden units;
- $f^n = \underset{f \in \mathfrak{C}^n}{\arg\min}\, L(\boldsymbol{\theta})$, the best n-layer neural network approximation to the PDE solution.

The main result is the convergence of the neural network approximators to the *true* PDE solution:

$$f^n \to u \qquad \text{as} \qquad n \to \infty$$

Further details, conditions, statement of the theorem and proofs are found in Section 7 of Sirignano and Spiliopoulos (2018). It is should be noted that, similar to the Universal Approximation Theorem, this result does not prescribe a way of designing or estimating the neural network successfully.

5.4 Implementation Details

The architecture adopted by Sirignano and Spiliopoulos (2018) is similar to that of LSTMs and Highway Networks described in the previous chapter. It consists of three layers, which we refer to as **DGM layers**: an input layer, a hidden layer and an output layer, though this can be easily extended to allow for additional hidden layers.

From a bird's-eye perspective, each DGM layer takes as an input the original mini-batch inputs $\boldsymbol{x}$ (in our case this is the set of randomly sampled time-space points) and the output of the previous DGM layer. This process culminates with a vector-valued output $\boldsymbol{y}$ which consists of the neural network approximation of the desired function u evaluated at the mini-batch points. See Figure 5.2 for a visualization of the overall architecture.

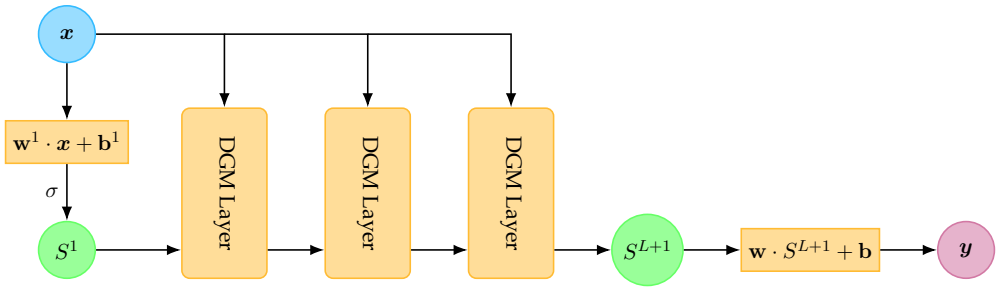

Figure 5.2: *Bird's-eye perspective of overall DGM architecture.*

Within a DGM layer, the mini-batch inputs along with the output of the previous layer are transformed through a series of operations that closely resemble those in Highway Networks. Below, we present the architecture in the equations along with a visual representation of a single DGM layer in Figure 5.3:

$$
\begin{aligned}
S^1 &= \sigma\left(\mathbf{w}^1 \cdot \boldsymbol{x} + \mathbf{b}^1\right) \\
Z^\ell &= \sigma\left(\mathbf{u}^{z,\ell} \cdot \boldsymbol{x} + \mathbf{w}^{z,\ell} \cdot S^\ell + \mathbf{b}^{z,\ell}\right) && \ell = 1, ..., L \\
G^\ell &= \sigma\left(\mathbf{u}^{g,\ell} \cdot \boldsymbol{x} + \mathbf{w}^{g,\ell} \cdot S^\ell + \mathbf{b}^{g,\ell}\right) && \ell = 1, ..., L \\
R^\ell &= \sigma\left(\mathbf{u}^{r,\ell} \cdot \boldsymbol{x} + \mathbf{w}^{r,\ell} \cdot S^\ell + \mathbf{b}^{r,\ell}\right) && \ell = 1, ..., L \\
H^\ell &= \sigma\left(\mathbf{u}^{h,\ell} \cdot \boldsymbol{x} + \mathbf{w}^{h,\ell} \cdot \left(S^\ell \odot R^\ell\right) + \mathbf{b}^{h,\ell}\right) && \ell = 1, ..., L \\
S^{\ell+1} &= \left(1 - G^\ell\right) \odot H^\ell + Z^\ell \odot S^\ell && \ell = 1, ..., L \\
f(t, \boldsymbol{x}; \boldsymbol{\theta}) &= \mathbf{w} \cdot S^{L+1} + \mathbf{b}
\end{aligned}
$$

where $\odot$ denotes Hadamard (element-wise) multiplication, L is the total number of layers, σ is an activation function and the $\mathbf{u}$, $\mathbf{w}$ and $\mathbf{b}$ terms with various superscripts are the model parameters.

Similar to the intuition for LSTMs, each layer produces weights based on the last layer, determining how much of the information gets passed to the next layer. In Sirignano and Spiliopoulos (2018) the authors also argue that including repeated element-wise multiplication of nonlinear functions helps capture "sharp turn" features present in more complicated functions. Note that at every iteration the original input enters into the calculations of every intermediate step, thus decreasing the probability of vanishing gradients of the output function with respect to x.

Compared to a Multilayer Perceptron (MLP), the number of parameters in each hidden layer of the DGM network is roughly eight times bigger than the same

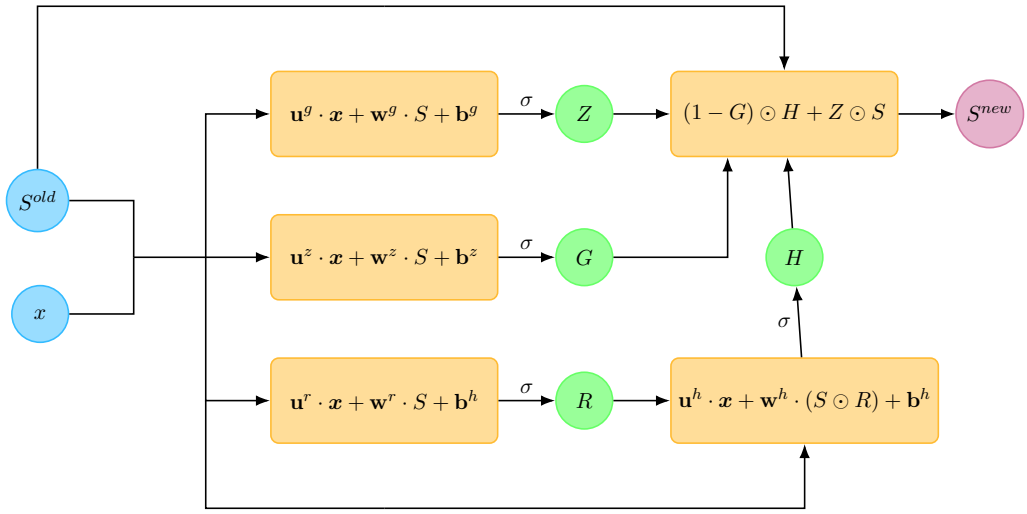

Figure 5.3: *Operations within a single DGM layer.*

number in an usual dense layer. Since each DGM network layer has 8 weight matrices and 4 bias vectors while the MLP network only has one weight matrix and one bias vector (assuming the matrix/vector sizes are similar to each other). Thus, the DGM architecture, unlike a deep MLP, is able to handle issues of vanishing gradients, while being flexible enough to model complex functions.

Remark on Hessian implementation: second-order differential equations call for the computation of second derivatives. In principle, given a deep neural network $f(t, \boldsymbol{x}; \boldsymbol{\theta})$, the computation of higher-order derivatives by automatic differentiation is possible. However, given $\boldsymbol{x} \in \mathbb{R}^n$ for $n > 1$, the computation of those derivatives becomes computationally costly, due to the quadratic number of second derivative terms and the memory-inefficient manner in which the algorithm computes this quantity for larger mini-batches. For this reason, we implement a finite difference method for computing the Hessian along the lines of the methods discussed in Chapter 3. In particular, for each of the sample points x, we compute the value of the neural net and its gradients at the points $x + he_j$ and $x - he_j$, for each canonical vector e_j, where h is the step size, and estimate the Hessian by central finite differences, resulting in a precision of order $\mathcal{O}(h^2)$. The resulting matrix H is then symmetrized by the transformation $0.5(H + H^T)$.

Chapter 6

Implementation of the Deep Galerkin Method

In this chapter we apply the Deep Galerkin Method to solve various PDEs that arise in financial contexts, as discussed in Chapter 2. The application of neural networks to the problem of numerically solving PDEs (and other problems) requires a great deal of experimentation and implementation decisions. Even with the basic strategy of using the DGM method, there are already numerous decisions to make, including:

- the network architecture;
- the size of the neural network to use to achieve a good balance between execution time and accuracy;
- the choice of activation functions and other hyperparameters;
- the random sampling strategy, selection of optimization and numerical (e.g. differentiation and integration) algorithms, training intensity;
- programming environment.

In light of this, our approach was to begin with simple and more manageable PDEs and then, as stumbling blocks are gradually surpassed, move on to more challenging ones. We present the results of applying DGM to the following problems:

1. *European Call Options*:
   We begin with the Black-Scholes PDE, a **linear PDE** which has a simple analytical solution and is a workhorse model in finance. This also creates the basic setup for the remaining problems.

2. *American Put Options*:
   Next, we tackle American options, whose main challenge is the **free boundary problem**, which needs to be found as part of the solution of the problem. This requires us to adapt the algorithm (particularly, the loss function) to handle this particular detail of the problem.

3. *The Fokker-Plank Equation*:
   Subsequently, we address the Fokker-Planck equation, whose solution is a probability density function that has special **constraints** (such as being positive on its domain and integrating to one) that need to met by the method.

4. *Stochastic Optimal Control Problems*:
   For even more demanding challenges, we focus on HJB equations, which can be highly **nonlinear**. In particular, we consider two optimal control problems: the Merton problem and the optimal execution problem.

5. *Systemic Risk*:
   The systemic risk problem allows us to apply the method to a **multidimensional system of HJB equations**, which involves multiple variables and equations with a high degree of nonlinearity.

6. *Mean Field Games*:
   Lastly, we close our work with mean field games, which are formulated in terms of **conversant HJB and Fokker-Planck equations**.

The variety of problems we manage to successfully apply the method to attests to the power and flexibility of the DGM approach.

6.1 How this chapter is organized

Each section in this chapter highlights one of the case studies mentioned in the list above. We begin with the statement of the PDE and its analytical solution and proceed to present (possibly several) attempted numerical solutions based on the DGM approach. The presentation is done in such a way as to highlight the *experiential* aspect of our implementation. As such the first solution we present is by no means the best, and the hope is to demonstrate the learning process surrounding the DGM and how our solutions improve along the way. Each example is intended to highlight a different challenge faced - usually associated with the difficulty of the problem, which is generally increasing in examples - and a proverbial "moral of the story."

An important caveat is that, in some cases, we don't tackle the full problem in the sense that the PDEs that are given at the start of each section are not always in their primal form. The reason for this is that the PDEs may be too complex to implement in the DGM framework directly. This is especially true with HJB equations which involve an optimization step as part of the first order condition. In these cases we resort to to simplified versions of the PDEs obtained using simplifying ansatzes, but we emphasize that even these can be of significant difficulty.

Remark (a note on implementation): in all the upcoming examples we use the same network architecture used by Sirignano and Spiliopoulos (2018) presented in Chapter 5, initializing the weights with Xavier initialization. The network was trained for a number of iterations (epochs) which may vary by example, with random resampling of points for the interior and terminal conditions every 10 iterations. We also experimented with regular dense feedforward neural networks and managed to have some success fitting the first problem (European options) but we found them to be less likely to fit more irregular functions and more unstable to hyperparameters changes as well.

6.2 European Call Options

1: One-Dimensional Black-Scholes PDE

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

Solution:

$$g(t,x) \;=\; x\,\Phi(d_+) - Ke^{-r(T-t)}\Phi(d_-)$$

$$\text{where} \quad d_\pm \;=\; \frac{\ln(x/K) + (r \pm \frac{1}{2}\sigma^2)(T-t)}{\sigma\sqrt{T-t}}$$

As a first example of the DGM approach, we trained the network to learn the value of a European call option. For our experiment, we used the interest rate $r = 5\%$, the volatility $\sigma = 25\%$, the initial stock price $S_0 = 50$, the maturity time $T = 1$ and the option's strike price $K = 50$. In Figure 6.2 we present the true and estimated option values at different times to maturity.

First, we sampled uniformly on the time domain and according to a lognormal distribution on the space domain as this is the exact distribution that the stock

prices follow in this model. We also sampled uniformly at the terminal time point. However, we found that this did not yield good results for the estimated function. These sampled points and fits can be seen in the green dots and lines in Figure 6.1 and Figure 6.2.

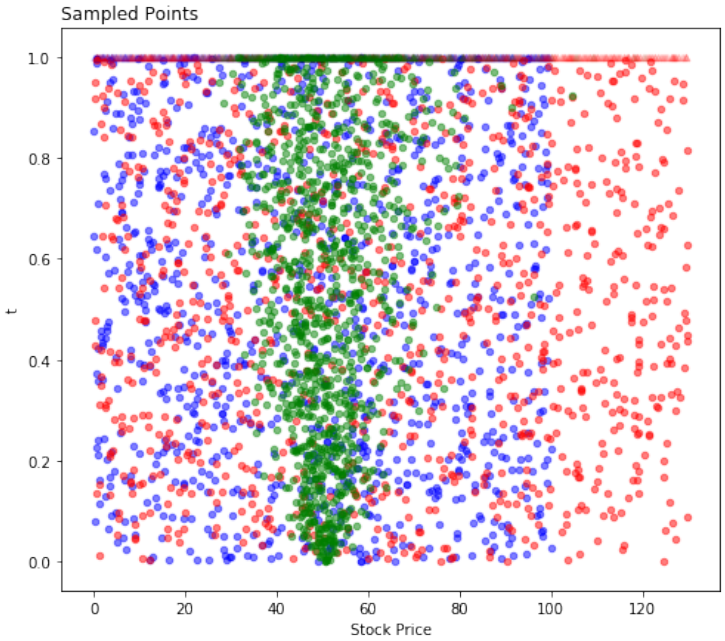

Figure 6.1: *Different sampling schemes: lognormal (green), uniform on* $[0, 1] \times [0, 100]$ *(blue) and uniform on* $[0, 1] \times [0, 130]$ *(red)*

Since the issues seemed to appear at regions that were not well-sampled we returned to the approach of Sirignano and Spiliopoulos (2018) and sampled uniformly in the region of interest $[0, 1] \times [0, 100]$. This improved the fit, as can be seen in the blue lines of Figure 6.2, however, there were still issues on the right end of the plots with the fitted solution dipping too early.

Finally, we sampled uniformly *beyond* the region of interest on $[0, 1] \times [0, 130]$ to show the DGM network points that lie to the right of the region of interest. This produced the best fit, as can be seen by the red lines in Figure 6.2.

Another point worth noting is that the errors are smaller for times that are closer to maturity. This reason for this behavior could be due to the fact that the estimation process is "drawing information" from the terminal condition. Since this term is both explicitly penalized and heavily sampled from, this causes the estimated function to behave well in this region. As we move away from this time point, this stabilizing effect diminishes leading to increased errors.

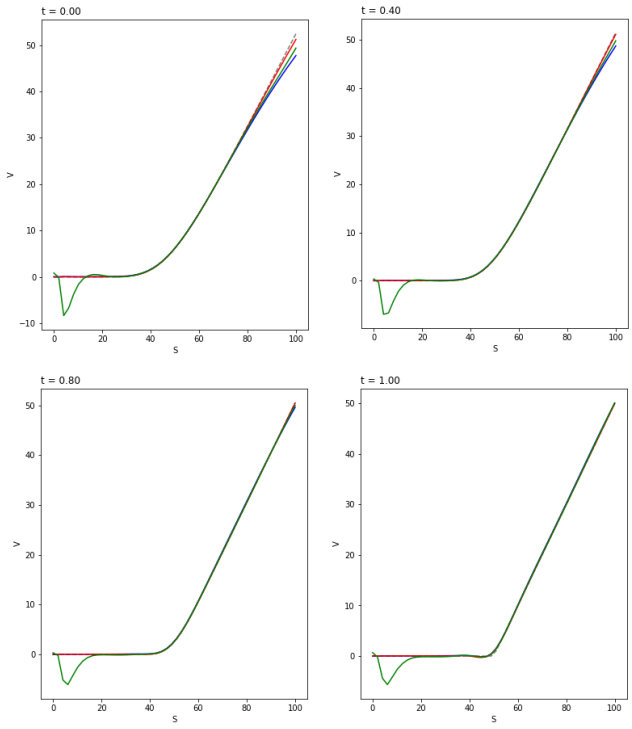

Figure 6.2: *Call Prices as a function of Stock Price: the black dashed line is the true value function, calculated using the Black and Scholes Formula; the green, blue and red lines correspond to the three sampling methodologies described above.*

Moral: sampling methodology matter!

6.3 American Put Options

2: Black-Scholes PDE with Free Boundary

$$\begin{cases} \partial_t g + rx \cdot \partial_x g + \frac{1}{2}\sigma^2 x^2 \cdot \partial_{xx} g - r \cdot g = 0 & \{(t,x) : g(t,x) > G(x)\} \\ g(t,x) \geq G(x) & (t,x) \in [0,T] \times \mathbb{R} \\ g(T,x) \ = \ G(x) & x \in \mathbb{R} \end{cases}$$

where $G(x) = (K - x)_+$

Solution: No analytical solution.

In order to further test the capabilities of DGM nets, we trained the network to learn the value of American-style put options. This is a step towards increased complexity, compared to the European variant, as the American option PDE formulation includes a free boundary condition. We utilize the same parameters as the European call option case: $r = 5\%$, $\sigma = 25\%$, $S_0 = 50$, $T = 1$ and $K = 50$.

In our first attempt, we trained the network using the method prescribed by Sirignano and Spiliopoulos (2018). The approach for solving free boundary problems there is to sample uniformly over the region of interest ($t \in [0, 1], S \in [0, 100]$ in our case), and accept/reject training examples for that particular batch of points, depending on whether or not they are inside or outside the boundary region implied by the last iteration of training. This approach was able to correctly recover option values.

As an alternative approach, we used a different formulation of the loss function that takes into account the free boundary condition instead of the acceptance/rejection methodology. In particular, we applied a loss to all points that violated the condition $\{(t, x) : g(t, x) \geq G(x)\}$ via:

$$\left\| \max\{-\left(f(t, x; \boldsymbol{\theta}) - (K - x)_+\right), 0\} \right\|^2_{[0,T]\times\Omega,\ \nu_1}$$

Figure 6.3 compares the DGM fitted option prices obtained using the alternative inequality loss for different maturities compared to the finite difference method approach. The figure shows that we are successful at replicating the option prices with this loss function.

Figure 6.4 depicts the absolute error between the estimated put option values and the analytical prices for corresponding European puts given by the Black-Scholes formula. Since the two should be equal in the continuation region, this can be a way of indirectly obtaining the early exercise boundary. The black line is the boundary obtained by the finite difference method and we see that is it closely matched by our implied exercise boundary. The decrease in the difference between the two option prices below the boundary as time passes reflects the deterioration of the early exercise optionality in the American option.

Moral: loss functions matter!

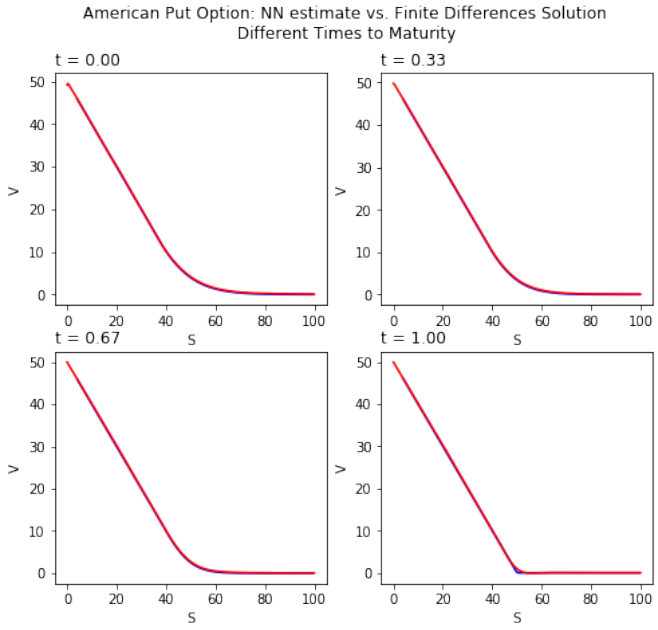

Figure 6.3: *Comparison of American put option prices at various maturities computed using DGM (red) vs. finite difference methods (blue)*

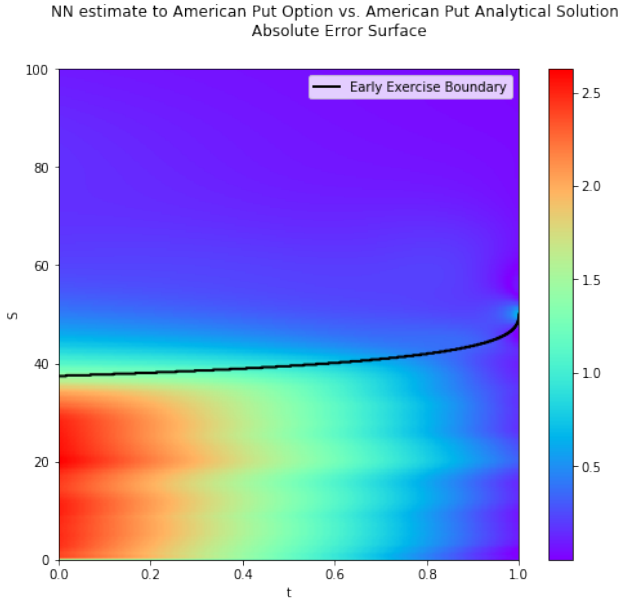

Figure 6.4: *Absolute difference between DGM-estimated American put option prices and analytical solution for corresponding European put options.*

6.4 Fokker-Planck Equations

3: Fokker-Planck Equation for OU process with random Gaussian start

$$\begin{cases} \partial_t p + \kappa \cdot p + \kappa(x-\theta) \cdot \partial_x p - \frac{1}{2}\sigma^2 \cdot \partial_{xx} p = 0 & (t,x) \in \mathbb{R}_+ \times \mathbb{R} \\ p(0,x) = \frac{1}{\sqrt{2\pi v}} \cdot e^{-\frac{x^2}{2v}} \end{cases}$$

Solution: Gaussian density function.

The Fokker-Planck equation introduces a new difficulty in the form of a constraint on the solution. We applied the DGM method to the Fokker-Planck equation for the Ornstein–Uhlenbeck mean-reverting process. If the process begins at a fixed point x_0, i.e. its initial distribution is a Dirac delta at x_0, then the solution for this PDE is known to have the normal distribution

$$X_T \sim N\left(x_0 \cdot e^{-\kappa(T-t)} + \theta\left(1 - e^{-\kappa(T-t)}\right), \frac{\sigma^2}{2\kappa}\left(1 - e^{-2\kappa(T-t)}\right)\right)$$

Since it is not possible to directly represent the initial delta numerically, one would have to approximate it, e.g. with a normal distribution with mean x_0 and a small variance. In the case where the starting point is Gaussian, we use Monte Carlo simulation to determine the distribution at every point in time, but we note that the distribution should be Gaussian since we are essentially using a conjugate prior.

For the DGM algorithm, we used loss function terms for the differential equation itself, the initial condition and added a penalty to reflect the non-negativity constraint. Though we intended to include another term to force the integral of the solution to equal one, this proved to be too computationally expensive, since an integral must be numerically evaluated at each step of the network training phase. For the parameters $\theta = 0.5$, $\sigma = 2$, $T = 1$, $\kappa = 0$, Figure 6.5 shows the density estimate p as a function of position x at different time points compared to the simulated distribution. As can be seen from these figures, the fitted distributions had issues around the tails and with the overall height of the fitted curve, i.e. the fitted densities did not integrate to 1. The neural network estimate, while correctly approximating the initial condition, is not able to conserve probability mass and the Gaussian bell shape across time.

To improve on the results, we apply a change of variables:

$$p\,(t,x) = \frac{e^{-u(t,x)}}{c(t)}$$

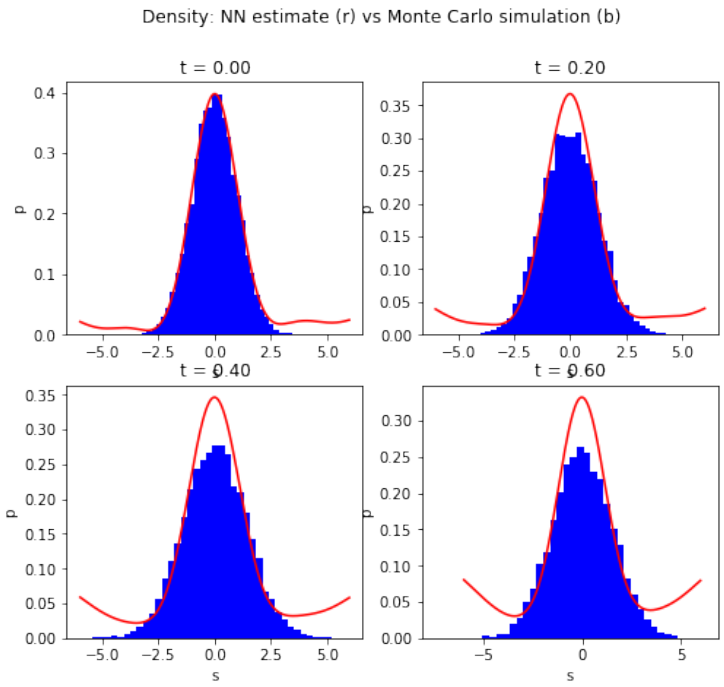

Figure 6.5: *Distribution of X_t at different times. Blue bars correspond to histograms of simulated values; red lines correspond to the DGM solution of the required Fokker-Planck equation.*

where $c(t)$ is a normalizing constant. This amounts to fitting an exponentiated normalized neural network guaranteed to remain positive and integrate to unity. This approach provides an alternative PDE to be solved by the DGM method:

$$\partial_t u + \kappa(x-\theta)\partial_x u - \frac{\sigma^2}{2}\left[\partial_{xx}u - (\partial_x u)^2\right] = \kappa + \frac{\int(\partial_t u)e^{-u}dx}{\int e^{-u}dx}$$

Notice that the new equation is a non-linear PDE dependent on a integral term. To handle the integral term and avoid the costly operation of numerically integrating at each step, we first uniformly sample $\{t_j\}_{j=1}^{N_t}$ from $t \in [0,T]$ and $\{x_k\}_{q=1}^{N_x}$ from $[x_{min}, x_{max}]$, then, for each t_j, we use **importance sampling** to approximate the expectation term by

$$I_t := \sum_{k=1}^{N_x}(\partial_t u(t_j, x_k))w(x_k)$$

where

$$w(x_k) = \frac{e^{u(t_j,x_k)}}{\sum_{k=1}^{N_x} e^{u(t_j,x_k)}}$$

Note that since the density for uniform distribution is constant within the sampling region, the denominator terms for the weights are cancelled. The L_1 loss is then

approximated by:

$$\frac{1}{N_t}\frac{1}{N_x}\sum_{j=1}^{N_t}\sum_{k=1}^{N_x}(\partial_t + \mathcal{L})u(t_j, x_k, I_t, \theta)$$

Even though the resulting equation turns out to be more complex, using this technique to train the network by solving for $u(x,t)$ and transforming back to $p(x,t)$ allowed us to achieve stronger results as evidence by the plots in Figure 6.6 .

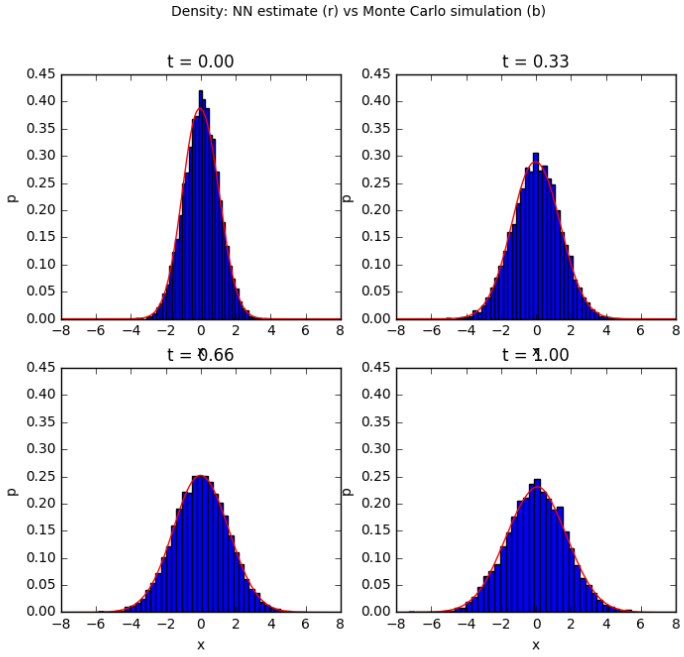

Figure 6.6: *Distribution of X_t at different times. Blue bars correspond to histograms of simulated values; red lines correspond to the DGM solution of the required Fokker-Planck equation using the modified approach.*

Notice that the network was able to accurately recover the shape and preserve the probability mass across time steps.

It is interesting to note that, in this example, the loss of linearity in the PDE was not as important to being able to solve the problem than imposing the appropriate structure on the desired function.

Moral: prior knowledge matters!

6.5 Stochastic Optimal Control Problems

In this section we tackle a pair of nonlinear HJB equations. The interest is in both the value function as well as the optimal control. The original form of the HJB equations contains an optimization term (first-order condition) which can be difficult to work with. Here we are working with the simplified PDE once the optimal control in feedback form is substituted back in and an ansatz is potentially used to simplify further. Since we are interested in both the value function and the optimal control, and the optimal control is written in terms of derivatives of the value function, a further step of numerically differentiating the DGM output (based on finite differences) is required for the optimal control.

6.5.1 Merton problem

4: Merton Problem - Optimal Investment with Exponential Utility

$$\begin{cases} \partial_t H - \frac{\lambda^2}{2\sigma^2}\frac{(\partial_x H)^2}{\partial_{xx} H} + rxH = 0 & (t,x) \in \mathbb{R}_+ \times \mathbb{R} \\ H(T,q) = -\alpha q^2 \end{cases}$$

Solution (value function and optimal control):

$$H(t,x) = -\exp\left[-x\gamma e^{r(T-t)} - \tfrac{\lambda^2}{2}(T-t)\right]$$

$$\pi_t^* = \frac{\lambda}{\gamma\sigma} e^{-r(T-t)}$$

where $\lambda = \dfrac{\mu - r}{\sigma}$

In this section, we attempt to solve the HJB equation for the Merton problem with exponential utility. In our first attempts, we found that the second-order derivative appearing in the denominator in the above equation generates large instabilities in the numerical resolution of the problem. Thus, we rewrite the equation by multiplying out to obtain:

$$-\frac{\lambda^2}{2\sigma^2}(\partial_x H)^2 + \partial_{xx} H\left(\partial_t H - \frac{\lambda^2}{2\sigma^2} + rxH\right) = 0$$

In this formulation, the equation becomes a quasi-linear PDE which was more stable numerically. The equation was solved with parameters $r = 0.05$, $\sigma = 0.25$, $\mu = 0.2$ and $\gamma = 1$, with terminal time $T = 1$, in the region $(t,x) \in [0,1]^2$, with oversampling of 50% in the x-axis.

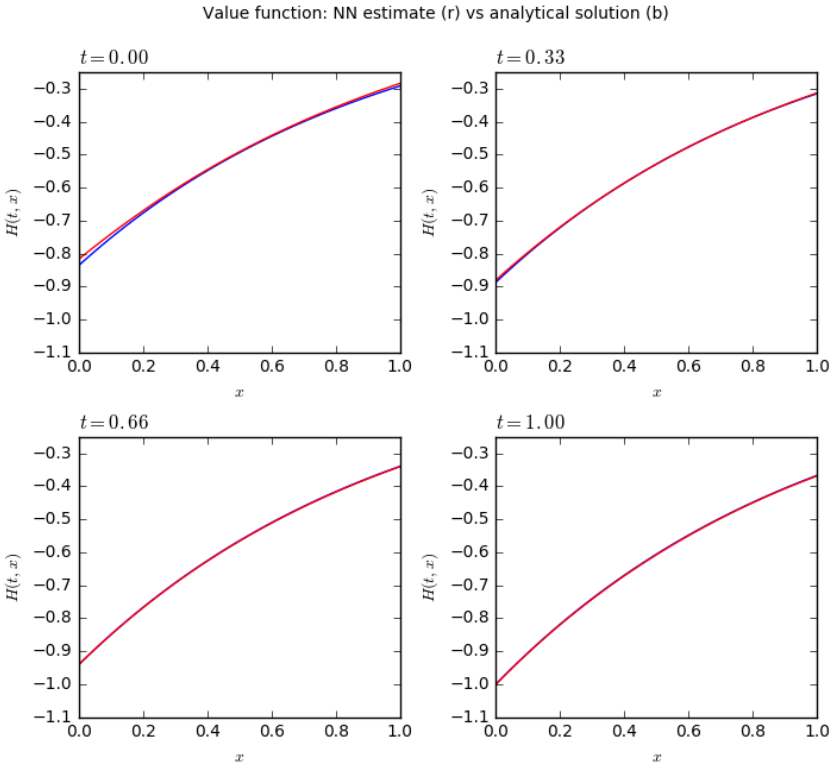

Figure 6.7: *Approximate (red) vs. analytical (blue) value function for the Merton problem.*

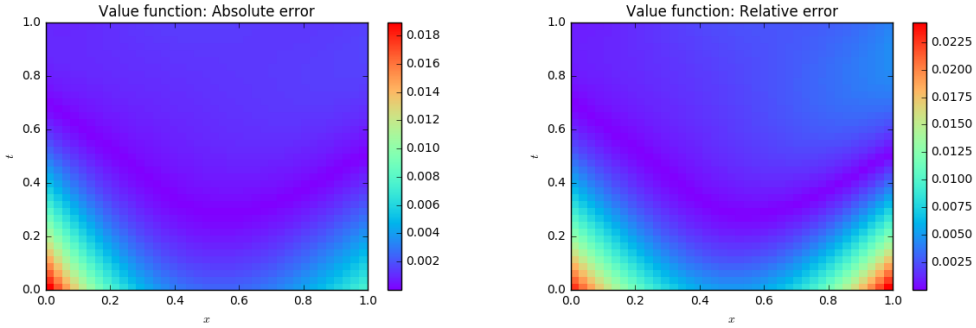

Figure 6.8: *Absolute (left panel) and relative (right panel) error between approximate and analytical solutions of the Merton problem value function.*

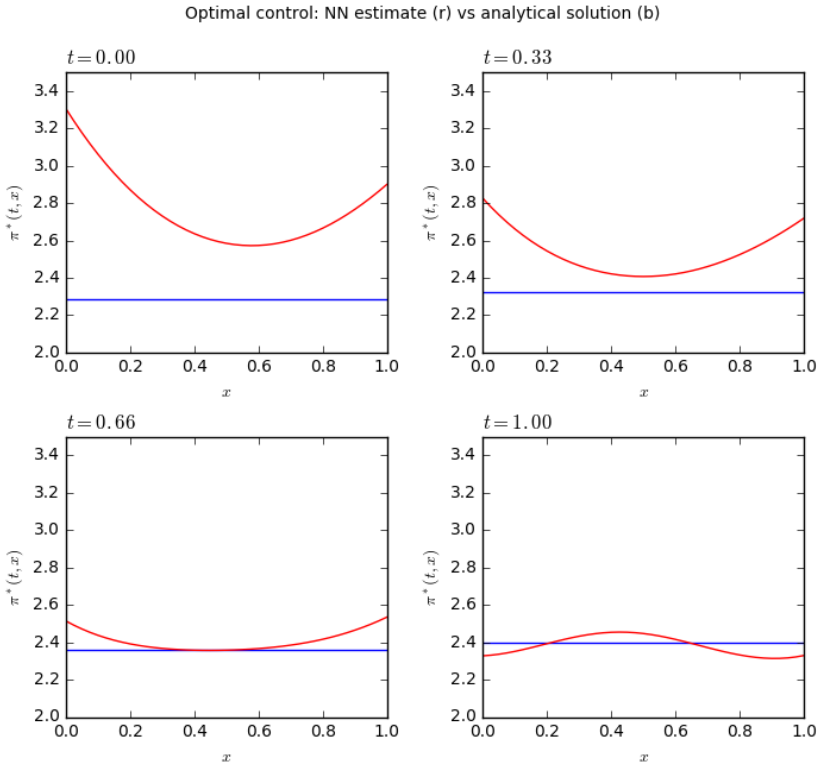

Figure 6.9: *Approximate (red) vs. analytical (blue) optimal control for the Merton problem.*

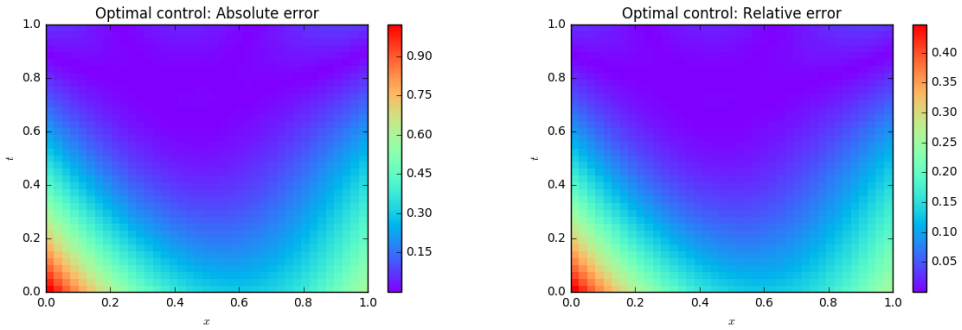

Figure 6.10: *Absolute (left panel) and relative (right panel) error between approximate and analytical solutions of the optimal control.*

The estimated value function (Figures 6.7 and 6.8) and optimal control (Figure 6.9) are compared with the analytical solution below. Examining the plots, we find that the value function is estimated well by the neural network. Notice, however that at $t = 0$, the error between the approximate and analytical solutions is larger, but within an acceptable range. This may once again be due to the fact that the terminal condition has a stabilizing effect on the solution that diminished as we move away from that time point.

In general, we are interested in the optimal control associated with the HJB equation. In this context, the optimal control involves dividing by the second-order derivative of the value function which appears to be small in certain regions. This causes a propagation of errors in the computed solution, as seen in Figures 6.9 and 6.10. The approximation appears to be reasonably close at $t = 1$, but diverges quickly as t goes to 0. Notice that regions with small errors in the value function solution correspond to large errors in the optimal control.

6.5.2 Optimal Execution

5: Optimal Liquidation with Permanent and Temporary Price Impact

$$\begin{cases} \partial_t h(t,q) - \phi q^2 + \frac{1}{4\kappa}\left(bq + \partial_q h(t,q)\right)^2 = 0 & (t,q) \in \mathbb{R}_+ \times \mathbb{R} \\ h(T,q) = -\alpha q^2 \end{cases}$$

Solution:

$$h(t) = \sqrt{k\phi} \cdot \frac{1 + \zeta e^{2\gamma(T-t)}}{1 - \zeta e^{2\gamma(T-t)}} \cdot q^2$$

$$\text{where} \qquad \gamma = \sqrt{\frac{\phi}{k}}, \qquad \zeta = \frac{\alpha - \frac{1}{2}b + \sqrt{k\phi}}{\alpha - \frac{1}{2}b - \sqrt{k\phi}}$$

For the second nonlinear HJB equation, the optimal execution problem was solved with parameters with $k = 0.01$, $b = 0.001$, $\phi = 0.1$, $\alpha = 0.1$, from $t = 0$ to terminal time $t = T = 1$, with $q \in [0, 5]$, with oversampling of 50% in the q-axis. The approximation in the plots below shows a good fit to the true value function. The optimal control solution for the equation only depends on the first derivative of the solution, so the error propagation is not as large as in the previous problem, as shown in the computed solution, where there is a good fit, worsening when q goes to 0 and t goes to T.

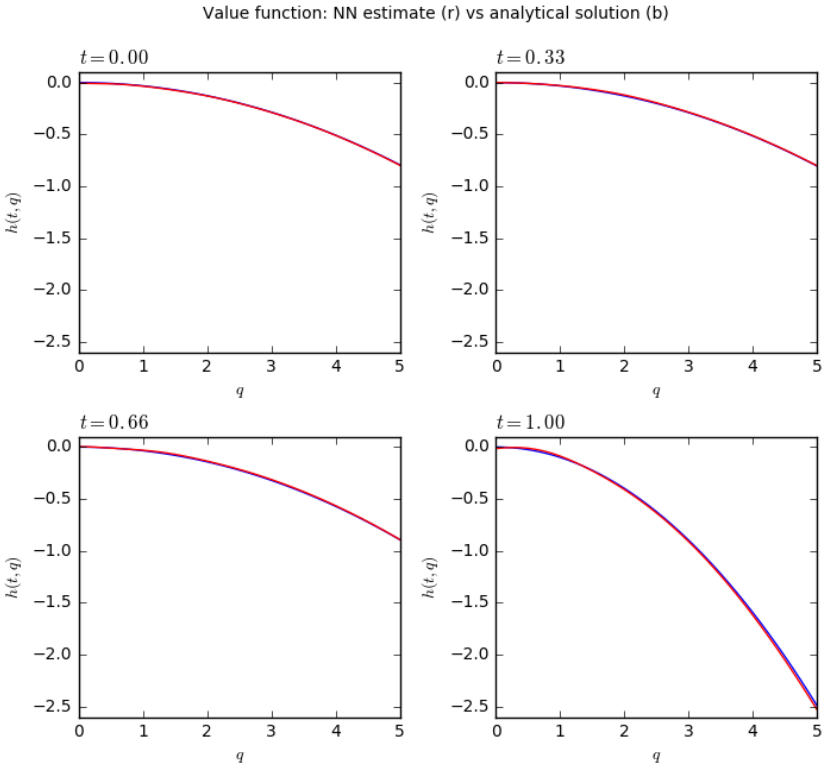

Figure 6.11: *Approximate (red) vs. true (blue) value function for the optimal execution problem.*

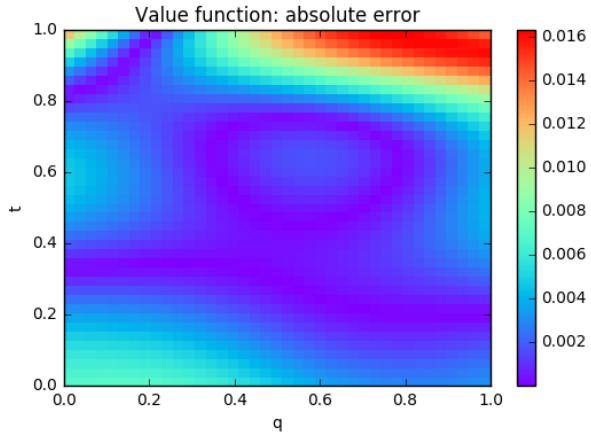

Figure 6.12: *Absolute error between approximate and analytical solutions for the value function of optimal execution problem.*

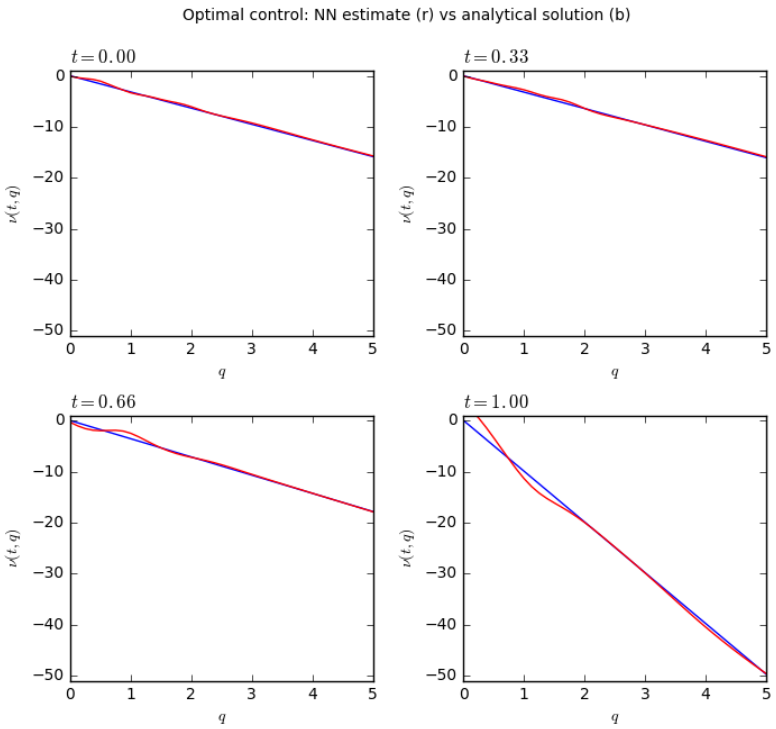

Figure 6.13: *Approximate (red) vs. true (blue) optimal trading rate for the optimal execution problem.*

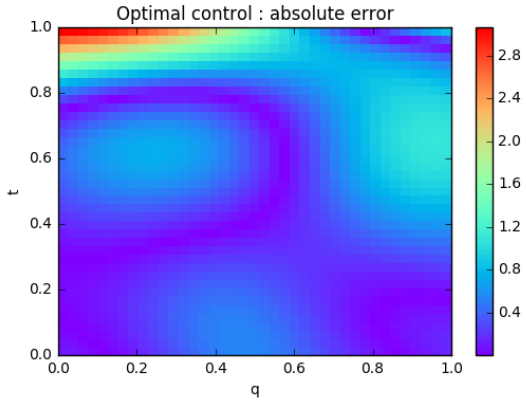

Figure 6.14: *Absolute error between approximate and analytical solutions for the optimal control in optimal execution problem.*

Moral: going from value function to optimal control is nontrivial!

6.6 Systemic Risk

6: Systemic Risk

$$\begin{cases} \partial_t V^i + \sum_{j=1}^{N} \left[a(\overline{x} - x^j) - \partial_j V^j \right] \partial_j V^i \\ \qquad + \frac{\sigma^2}{2} \sum_{j,k=1}^{N} \left(\rho^2 + \delta_{jk}(1-\rho^2) \right) \partial_{jk} V^i \\ \qquad + \frac{1}{2}(\epsilon - q)^2 \left(\overline{x} - x^i \right)^2 + \frac{1}{2} \left(\partial_i V^i \right)^2 = 0 \\ V^i(T, \boldsymbol{x}) = \frac{c}{2} \left(\overline{x} - x^i \right)^2 \qquad \text{for} \quad i = 1, ..., N. \end{cases}$$

Solution:

$$V^i(t, \boldsymbol{x}) = \frac{\eta(t)}{2} \left(\overline{x} - x^i \right)^2 + \mu(t)$$

$$\alpha_t^{i,*} = \left(q + \left(1 - \tfrac{1}{N} \right) \cdot \eta(t) \right) \left(\overline{X}_t - X_t^i \right)$$

where

$$\eta(t) = \frac{-(\epsilon - q)^2 \left(e^{(\delta^+ - \delta^-)(T-t)} - 1 \right) - c \left(\delta^+ e^{(\delta^+ - \delta^-)(T-t)} - \delta^- \right)}{\left(\delta^- e^{(\delta^+ - \delta^-)(T-t)} - \delta^+ \right) - c(1 - \frac{1}{N^2}) \left(e^{(\delta^+ - \delta^-)(T-t)} - 1 \right)}$$

$$\mu(t) = \tfrac{1}{2}\sigma^2(1-\rho^2) \left(1 - \tfrac{1}{N} \right) \int_t^T \eta(s)\, ds$$

$$\delta^{\pm} = -(a+q) \pm \sqrt{R}, \qquad R = (a+q)^2 + \left(1 - \tfrac{1}{N^2} \right)(\epsilon - q^2)$$

The systemic risk problems brings our first system of HJB equations (which happen to also be nonlinear). This was solved for the two-player ($N = 2$) case with correlation $\rho = 0.5$, $\sigma = 0.1$, $a = 1$, $q = 1$, $\epsilon = 10$, $c = 1$, from $t = 0$ to terminal time $t = T = 1$, with $(x_1, x_2) \in [0, 10] \times [0, 10]$, and results were compared with the analytical solution.

Note that the analytical solution has two symmetries, one between the value functions for both players, and one around the $x_1 = x_2$ line. The neural network solution captures both symmetries, fitting the analytical solution for this system. The regions with the largest errors were found in the symmetry axis, as t goes to 0, but away from those regions the error in the solution becomes very low. Once again this may be attributed to the influence of the terminal condition.

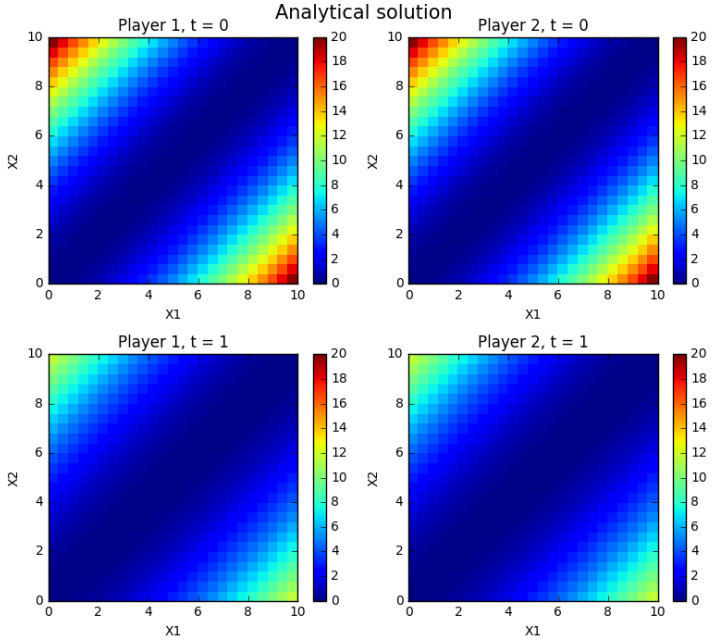

Figure 6.15: *Analytical solution to the systemic risk problem.*

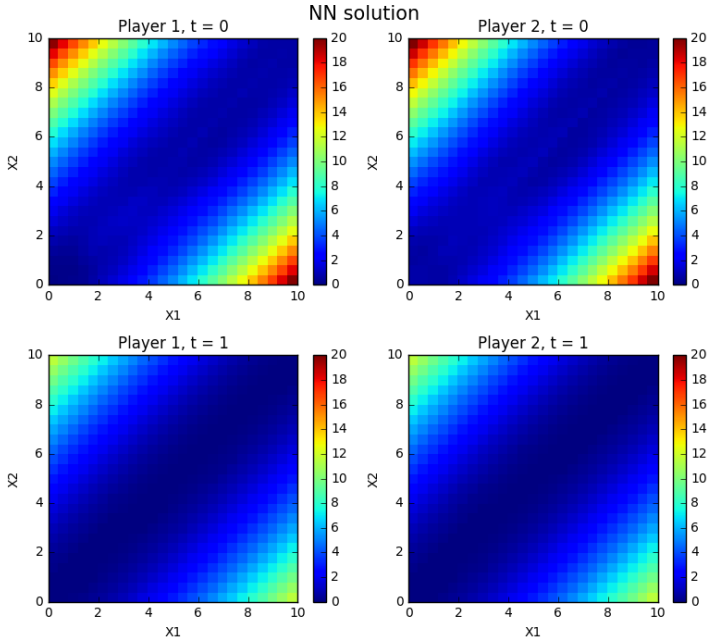

Figure 6.16: *Neural network solution to the systemic risk problem.*

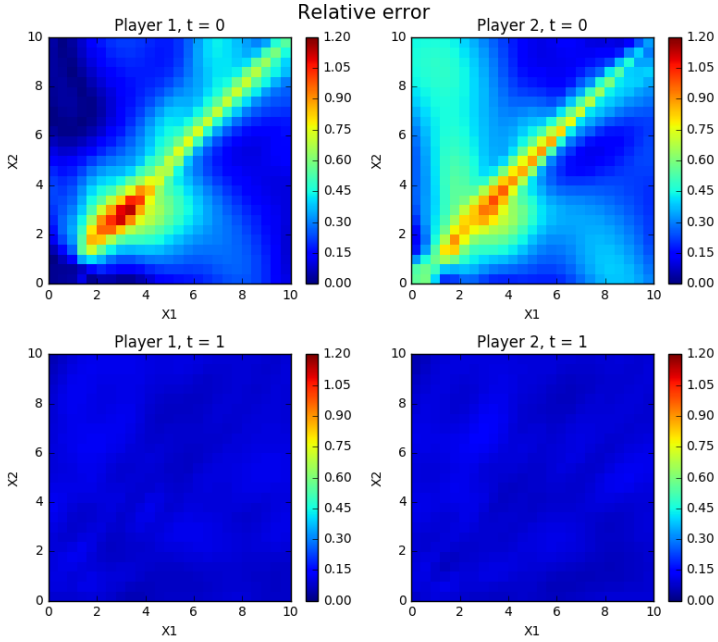

Figure 6.17: *Absolute error between approximate and analytical solutions for the systemic risk problem.*

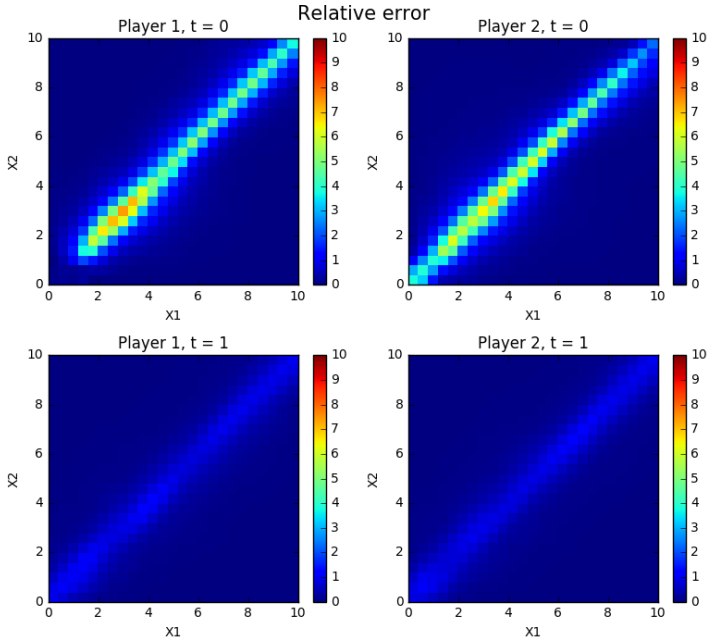

Figure 6.18: *Relative error between approximate and analytical solutions for the systemic risk problem.*

The systemic risk problem was also solved for five players with the same parameters as above to test the method's capability with higher dimensionality both in terms of the number of variables and the number of equations in the system. In the figures below, we compare the value function for a player when he deviates by Δx from the initial state from x_0, with $x_0 = 5$. Note that all players have the same value function by symmetry. The plots show that the neural network trained using the DGM approach is beginning to capture the overall shape of the solution, although there is still a fair amount of deviation from the analytical solution. This suggests that more training time, or a better training procedure, should eventually capture the true solution with some degree of accuracy.

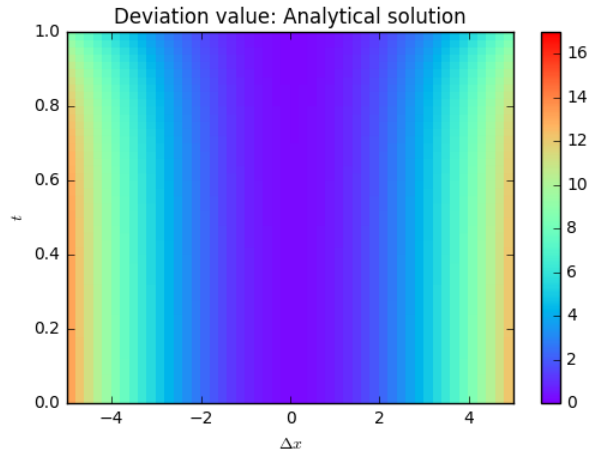

Figure 6.19: *Analytical solution to the systemic risk problem with five players.*

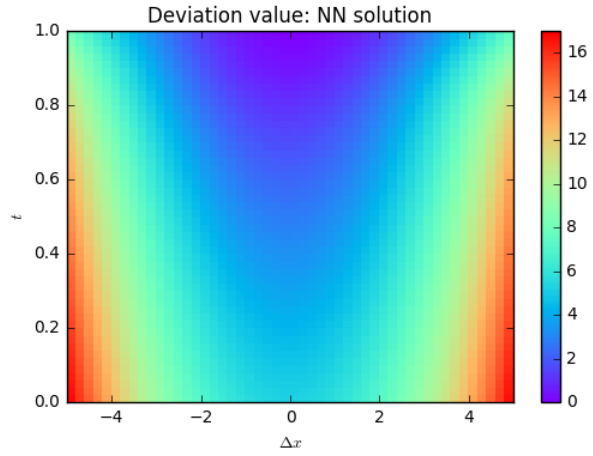

Figure 6.20: *Neural network solution to the systemic risk problem with five players.*

6.7 Mean Field Games

7: Optimal Liquidation in a Mean Field with Identical Preferences

$$
\begin{cases}
-\kappa\mu q = \partial_t h^a - \phi^a q^2 + \dfrac{(\partial_q h^a)^2}{4k} & \text{(HJB equation - optimality)} \\
H^a(T, x, S, q; \mu) = x + q(S - \alpha^a q) & \text{(HJB terminal condition)} \\
\partial_t m + \partial_q \left(m \cdot \dfrac{\partial h^a(t,q)}{2k} \right) = 0 & \text{(FP equation - density flow)} \\
m(0, q, a) = m_0(q, a) & \text{(FP initial condition)} \\
\mu_t = \displaystyle\int_{(q,a)} \frac{\partial h^a(t,q)}{2k}\, m(t, dq, da) & \text{(net trading flow)}
\end{cases}
$$

Solution: see Cardaliaguet and Lehalle (2017).

The main challenge with the MFG problem is that it involves both an HJB equation and a Fokker-Planck equation. Furthermore, the density governed by Fokker-Planck equation must remain positive on its domain and integrate to unity as we previously saw. The naïve implementation of the MFG problem yields poor results given that the integral term μ_t is expensive to compute and the density in the Fokker-Planck equation has some constraints that must be satisfied. Using the same idea of exponentiating and normalizing used in Section 6.4, we rewrite the density $m(t, q, a) = \frac{1}{c(t)} e^{-u(t,q,a)}$ to obtain a PDE for the function u:

$$-\partial_t u + \frac{1}{2k}(-\partial_q u \partial_q v + \partial_{qq} v) + \frac{\int (\partial_t u) e^{-u} dx}{\int e^{-u} dx} = 0$$

Both integral terms are handled by *importance sampling* as in the Fokker-Planck equation with exponential transformation.

The equation was solved numerically with parameters $A, \phi, \alpha, k = 1$, with terminal time $T = 1$. The initial mass distribution was a normal distribution with mean $E_0 = 5$ and variance 0.25. Results were calculated for $t \in [0, 1]$ and $q \in [0, 10]$. The value function, optimal control along with the expected values of the mass through time were compared with the analytical solution (an analytical solution for the probability mass is not available; however the expected value of this distribution can be computed analytically).

The analytical solutions for the value function and the optimal control were in an acceptable range for the problem, though it should be noted that for $t = 0$, the approximation diverges as q grows, but still fits reasonably well. The implied expected value from the fitted density showed a very good fit with the analytical solution. The probability mass could not be compared with an analytical solution, but it's reasonable to believe that it is close to the true solution given the remaining results.

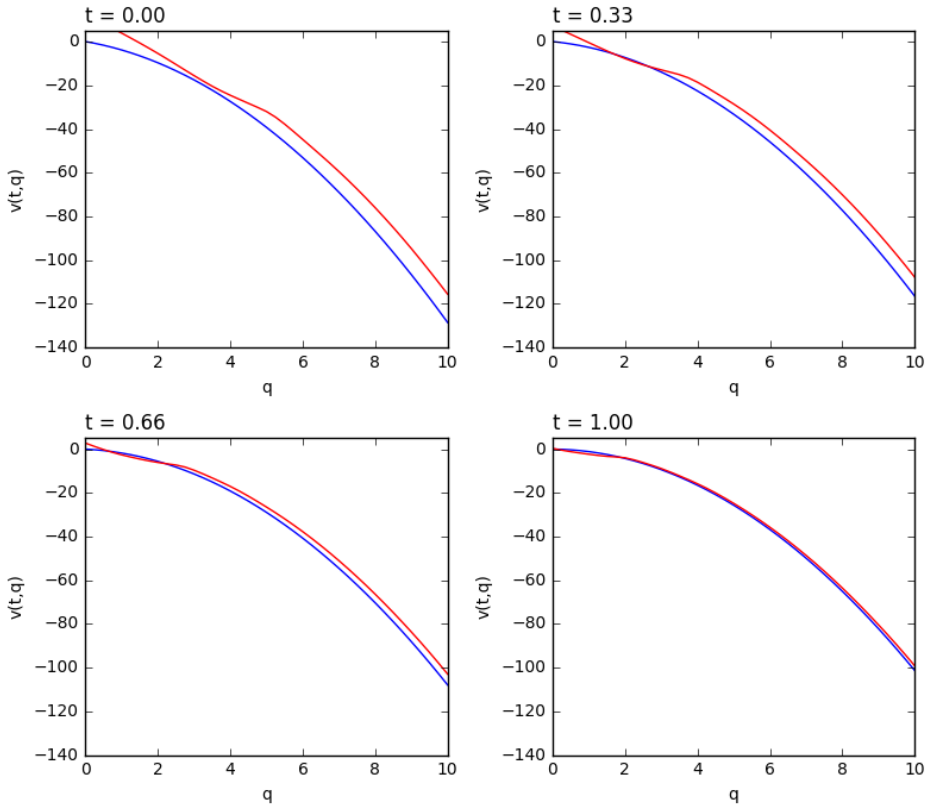

Figure 6.21: *Approximate (red) vs. analytical solution for the value function of the MFG problem.*

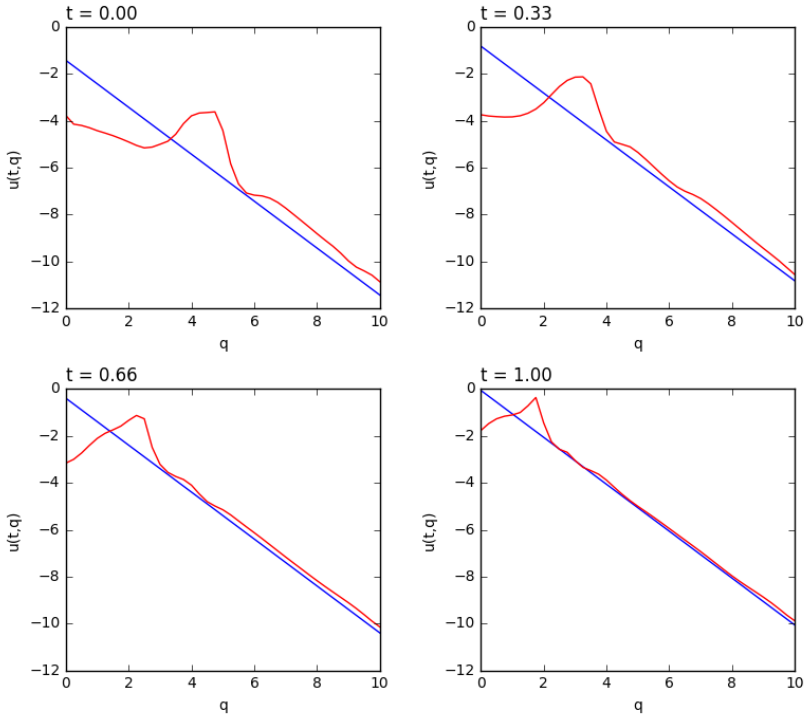

Figure 6.22: *Approximate (red) vs. analytical solution for the optimal control for the MFG problem.*

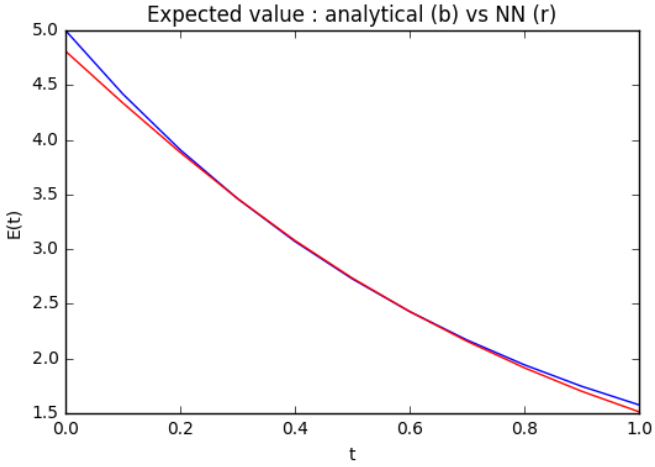

Figure 6.23: *Approximate (red) vs. analytical solution for the expected value of the distribution of agents for the MFG problem.*

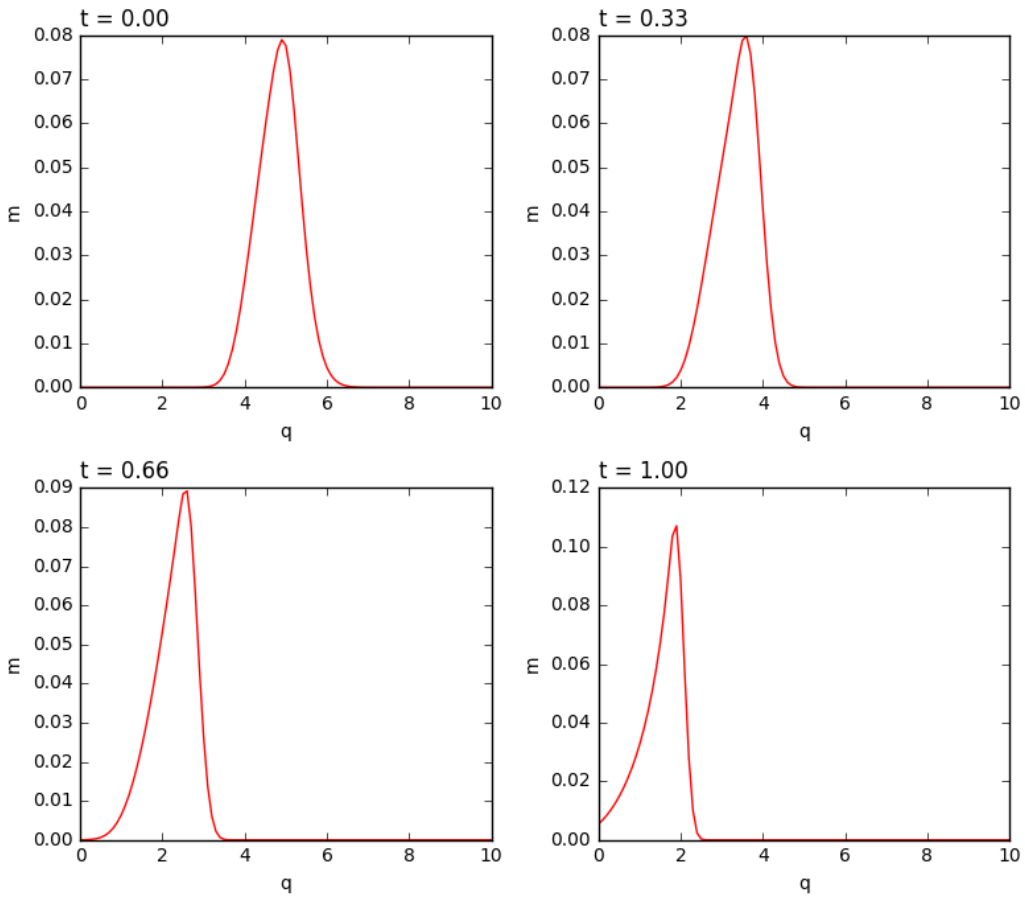

Figure 6.24: *Unnormalized probability mass of inventories for MFG; the curve shifts left as all traders are liquidating.*

6.8 Conclusions and Future Work

The main messages from the implementation of DGM can be distilled into **three main points**:

1. **Sampling method matters**: similar to choosing a grid in finite difference methods, where and how the sampled random points used for training are chosen is the single most important factor in determining the quality of the results.

2. **Prior knowledge matters**: having some information about the solution can dramatically improve the accuracy of the approximations. This proved to be instrumental in the Fokker-Planck and MFG applications. It also rings true for finite difference methods and even Monte Carlo methods (a good analogy is the use of control variates).

3. **Training time matters**: in some cases, including some of our earlier attempts, the loss functions appeared to be decreasing with iterations and the shape of solutions seemed to be moving in the right direction. As is the case with neural networks, and SGD-based optimization in general, sometimes the answer is to let the optimizer run longer. As a point of reference, Sirignano and Spiliopoulos (2018) ran the algorithm on a supercomputer with a GPU cluster and achieve excellent results in up to 200 dimensions.

The last point regarding runtime is particularly interesting. While finite difference methods are memory-intensive, training the DGM network can take a long amount of time. This hints at a notion known in computer science as **space-time tradeoff**. However it should be noted that the finite difference approach will simply not run for high dimensionality, whereas the DGM (when properly executed) will arrive at a solution, though the runtime may be long. It would be interesting to study the space-time tradeoff for numerical methods used to solve PDEs.

As discussed earlier in this work, generalization in our context refers to how well the function satisfies the conditions of the PDE for points or regions in the function's domain that were not sampled in the training phase. Our experiments show that the DGM method does not generalize well in this sense; the function fits well on regions that are well-sampled (in a sense, this can be viewed as finding a solution to a similar yet distinct PDE defined over the region where sampling occurs). This is especially problematic for PDEs defined on unbounded domains, since it is impossible to sample everywhere for these problems using uniform distributions. Even when sampling from distributions with unbounded support, we may undersample relevant portions of the domain (or oversample regions that are not as relevant). Choosing the best distribution to sample from may be part of the problem,

i.e. it may not be clear which is the appropriate distribution to use in the general case when applying DGM. As such, it would be interesting to explore efficient ways of tackling the issue of choosing the sampling distribution. On a related note, one could also explore more efficient methods of random sampling particularly in higher dimensions, e.g. quasi-Monte Carlo methods, Latin hypercube sampling.

Also, it would be interesting to understand what class of problems DGM can (or cannot) generalize to; a concept we refer to as **meta-generalization**. Is there an architecture or training method that yields better results for a wider range of PDEs?

Another potential research question draws inspiration from transfer learning, where knowledge gained from solving one problem can be applied to solving a different but related problem. In the context of PDEs and DGM, does knowing the solution to a simpler related PDE and using this solution as training data improve the performance of the DGM method for a more complex PDE?

Finally, we remark that above all neural networks are rarely a "one-size-fits-all" tool. Just as is the case with numerical methods, they need to be modified based on the problem. Continual experimentation and reflection is key to improving results, but a solid understanding of the underlying processes is vital to avoiding "black-box" opacity.

A Note On Performance

In order to have a sense on how sensitive DGM is to the computational environment used to train the neural networks, we benchmarked training times both using and not using graphical processing units (GPUs). It is well established among machine learning practitioners that GPUs are able to achieve much higher throughput on typical neural net training workloads due to parallelization opportunities at the numerical processing level. On the other hand, complex neural network architectures such as those of LSTM models may be harder to parallelize. Some disclaimers are relevant at this point: these tests are not meant to be exhaustive nor detailed. The goal is only to evaluate how much faster using GPUs can be for the model at hand. Other caveats are that we are using relatively small scale training sessions and we are running on a relatively low performance GPU (GeForce 830M).

First test scenario

Here we start with a DGM network with 3 hidden layers and 50 neurons per layer. At first, we train the network as usual and then with no resampling in the training loop to verify that the resampling procedure is not significantly impacting the training times. The numerical values are given in seconds per optimization step.

Seconds / optimization steps	CPU	GPU
Regular training	0.100	0.119
Training without resampling	0.099	0.112

Table 6.1: *In loop resampling impact*

Surprisingly, however, we also verify that the GPU is actually running slower than the CPU!
Next, we significantly increase the size of the network to check the impact on the training times. We train networks with 10 and 20 layers, with 200 neurons in both cases.

Seconds / optimization steps	CPU	GPU
10 layers	5.927	3.873
20 layers	13.458	8.943

Table 6.2: *Network size impact*

Now we see that the (regular) training times increase dramatically and that the GPU is running faster then the CPU as expected. We hypothesize that, given the complexity of DGM network architecture, the GPU engine implementation is not

able to achieve enough parallelization in the computation graph to run faster than the CPU engine implementation when the network is small.

Second test scenario

We begin this section by noting that each hidden layer in the DGM network is roughly eight times bigger than a multilayer perceptron network, since each DGM network layer has 8 weight matrices and 4 bias vectors while the MLP network only has one weight matrix and one bias vector per layer. So here we train a MLP network with 24 hidden layers and 50 neurons per layer (which should be roughly equivalent with respect to the number of parameters to the 3 layered DGM network above). We also train a bigger MLP network with 80 layers and 200 neurons per layer (which should be roughly equivalent with respect to the number of parameters to the 10 layered DGM network above).

Seconds / optimization steps	CPU	GPU
24 layers	0.129	0.077
80 layers	5.617	2.518

Table 6.3: *Network size impact*

From the results above we verify that the GPU has a clear performance advantage over the CPU even for small MLP networks.
We also note that, while the CPU training times for the different network architectures (with comparable number of parameters) were roughly equivalent, the GPU engine implementation is much more sensitive to the complexity of the network architecture.

Bibliography

Achdou, Y., Pironneau, O., 2005. Computational methods for option pricing. Vol. 30. Siam.

Almgren, R., Chriss, N., 2001. Optimal execution of portfolio transactions. Journal of Risk 3, 5–40.

Bishop, C. M., 2006. Pattern Recognition and Machine Learning. Springer.

Black, F., Scholes, M., 1973. The pricing of options and corporate liabilities. Journal of Political Economy 81 (3), 637–654.

Brandimarte, P., 2013. Numerical methods in finance and economics: a MATLAB-based introduction. John Wiley & Sons.

Burden, R. L., Faires, J. D., Reynolds, A. C., 2001. Numerical analysis. Brooks/cole Pacific Grove, CA.

Cardaliaguet, P., Lehalle, C.-A., 2017. Mean field game of controls and an application to trade crowding. Mathematics and Financial Economics, 1–29.

Carmona, R., Sun, L.-H., Fouque, J.-P., 2015. Mean field games and systemic risk. Communications in Mathematical Sciences 14 (4), 911–933.

Cartea, Á., Jaimungal, S., 2015. Optimal execution with limit and market orders. Quantitative Finance 15 (8), 1279–1291.

Cartea, Á., Jaimungal, S., 2016. Incorporating order-flow into optimal execution. Mathematics and Financial Economics 10 (3), 339–364.

Cartea, Á., Jaimungal, S., Penalva, J., 2015. Algorithmic and high-frequency trading. Cambridge University Press.

Cybenko, G., 1989. Approximation by superpositions of a sigmoidal function. Mathematics of control, signals and systems 2 (4), 303–314.

Evans, L. C., 2010. Partial Differential Equations. American Mathematical Society, Providence, R.I.

Goodfellow, I., Bengio, Y., Courville, A., Bengio, Y., 2016. Deep Learning. Vol. 1. MIT press Cambridge.

Henderson, V., Hobson, D., 2002. Substitute hedging. Risk Magazine 15 (5), 71–76.

Henderson, V., Hobson, D., 2004. Utility indifference pricing: An overview. Volume on Indifference Pricing.

Hochreiter, S., Schmidhuber, J., 1997. Long short-term memory. Neural computation 9 (8), 1735–1780.

Hornik, K., 1991. Approximation capabilities of multilayer feedforward networks. Neural networks 4 (2), 251–257.

Huang, M., Malhamé, R. P., Caines, P. E., 2006. Large population stochastic dynamic games: closed-loop Mckean-Vlasov systems and the Nash certainty equivalence principle. Commun. Inf. Syst. 6 (3), 221–252.

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

Lasry, J.-M., Lions, P.-L., 2007. Mean field games. Japanese Journal of Mathematics 2 (1), 229–260.

Merton, R., 1971. Optimum consumption and portfolio-rules in a continuous-time framework. Journal of Economic Theory.

Merton, R. C., 1969. Lifetime portfolio selection under uncertainty: The continuous-time case. The Review of Economics and Statistics, 247–257.

Pham, H., 2009. Continuous-Time Stochastic Control and Optimization with Financial Applications. Vol. 61. Springer Science & Business Media.

Shalev-Shwartz, S., Ben-David, S., 2014. Understanding Machine Learning: From Theory to Algorithms. Cambridge University Press.

Sirignano, J., Spiliopoulos, K., 2018. DGM: A deep learning algorithm for solving partial differential equations. arXiv preprint arXiv:1708.07469.

Srivastava, R. K., Greff, K., Schmidhuber, J., 2015. Highway networks. arXiv preprint arXiv:1505.00387.

Touzi, N., 2012. Optimal Stochastic Control, Stochastic Target Problems, and Backward SDE. Vol. 29. Springer Science & Business Media.