Distribution Agreement

In presenting this thesis or dissertation as a partial fulfillment of the requirements for an advanced degree from Emory University, I hereby grant to Emory University and its agents the non-exclusive license to archive, make accessible, and display my thesis or dissertation in whole or in part in all forms of media, now or hereafter known, including display on the world wide web. I understand that I may select some access restrictions as part of the online submission of this thesis or dissertation. I retain all ownership rights to the copyright of the thesis or dissertation. I also retain the right to use in future works (such as articles or books) all or part of this thesis or dissertation.

Signature:

_____ _____

Xingjian Li Date

Improving Sampling and Function Approximation in Machine Learning Methods for Solving Partial Differential Equations

By

Xingjian Li
Doctor of Philosophy

Mathematics

Lars Ruthotto, Ph.D.
Advisor

Yuanzhe Xi, Ph.D.
Committee Member

Elizabeth Newman, Ph.D.
Committee Member

Accepted:

Kimberly Jacob Arriola, Ph.D.
Dean of the James T. Laney School of Graduate Studies

Date

Improving Sampling and Function Approximation in Machine Learning Methods for Solving Partial Differential Equations

By

Xingjian Li
B.S., Xiamen University, 2019

Advisor: Lars Ruthotto, Ph.D.

An abstract of
A dissertation submitted to the Faculty of the
James T. Laney School of Graduate Studies of Emory University
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy
in Mathematics
2024

Abstract

Improving Sampling and Function Approximation in Machine Learning Methods for Solving Partial Differential Equations
By Xingjian Li

Numerical solutions to partial differential equations (PDEs) remain one of the main focuses in the field of scientific computing. Deep learning and neural network based methods for solving PDEs have gained much attention and popularity in recent years. The nonlinear structures and universal approximation property of neural networks allow for a cheaper approximation of functions in high dimensions compared to many traditional numerical methods. Reformulating PDE problems as optimization tasks also enables straightforward setup and implementation and can sometimes circumvent stability concerns common for classic numerical methods that rely on explicit or semi-explicit time discretization. However low accuracy and convergence difficulty stand as challenges to deep learning based schemes and fine-tuning neural networks can also be computationally expensive at times.

We present some of our findings using machine learning methods for solving certain PDEs. Since low and high dimensional PDEs often require very different numerical methods to solve, we divide our work into two main sections based on the dimensionality of a problem. In the first half we focus on the popular Physics Informed Neural Networks (PINNs) framework, specifically in problems with dimensions less than or equal to three. We present an alternative optimization based algorithm using a B-spline polynomial function approximator and accurate numerical integration with a grid based sampling scheme. With implementation using popular machine learning libraries, our approach serves as a direct substitute for PINNs, and through performance comparison between the two methods over a wide selection of examples, we find that for low dimensional problems, our proposed method can improve both accuracy and reliability when compared to PINNs. In the second half, we focus on a general class of stochastic optimal control (SOC) problems. By leveraging the underlying theory we propose a neural network solver that solves the SOC problem and the corresponding Hamilton–Jacobi–Bellman (HJB) equation simultaneously. Our method utilizes the stochastic Pontryagin maximum principle and is thus unique in the sampling strategy, this combined with modifying the loss function enables us to tackle high-dimensional problems efficiently.

Improving Sampling and Function Approximation in Machine Learning Methods for Solving Partial Differential Equations

By

Xingjian Li
B.S., Xiamen University, 2019

Advisor: Lars Ruthotto, Ph.D.

A dissertation submitted to the Faculty of the
James T. Laney School of Graduate Studies of Emory University
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy
in Mathematics
2024

Acknowledgments

I would like to express my deepest gratitude to my advisor Prof. Lars Ruthotto for his constant support and caring throughout my Ph.D. studies. Beyond the academic and mathematical training that I received from Prof. Lars Ruthotto, he also passed on to me the value of integrity and honesty and helped me through many down times by being a role model. I could not have finished the Ph.D. without his help and I feel extremely fortunate to be his student.

I would like to thank some of my collaborators including Prof. Samy Wu Fung, Prof. Levon Nurbekyan, Dr. Deepanshu Verma, and Dr. Derek Onken. They have helped me immensely during my time at Emory and taught me many valuable lessons that have well-equipped me for my Ph.D. studies.

My thanks also go to my thesis committee members Prof. Yuanzhe Xi and Prof. Elizabeth Newman, who has given me valuable suggestions for my thesis and advice for my career path.

I would like to thank many of the friends that I made at Emory: Chang Meng, Nicole Yang, Abbey Julian, Malvern Madondo, Kelvin Kan, Haley Rosso, Chris Keyes, Shifan Zhao, Ayush Basu, Maxwell Auerbach, Benjamin Yellin, and many more. They have supported me throughout my time here at Emory and I value my friendship with them a lot.

Finally, I would like to thank my family for supporting me in my life in general, for encouraging me when I encounter difficulties, and for listening to all my complaints.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Numerical methods for solving partial differential equations (PDEs) play a crucial role in simulating and understanding a wide range of phenomena in science and engineering [36, 30, 27, 6]. Partial differential equations are ubiquitous in math as well as other scientific fields such as physics, engineering, economics, etc. Analytical solutions to many PDEs are often elusive or impractical, sometimes impossible to obtain, necessitating the development of numerical methods for approximation of said solutions, with examples such as [115, 95, 62, 37]. Many different approaches have been proposed over the years for different PDE problems. One way to classify these methods is to separate them into Eulerian and Lagrangian methods, see [114, 33, 114].

Eulerian and Lagrangian methods are two distinct approaches used in the numerical solution of PDEs. To briefly summarize, in Eulerian methods, the focus is on fixed points in space, and the solution is tracked at specific locations over time. This usually corresponds to methods such as finite difference or finite element methods, where the computational domain often remains stationary, and values at discrete grid points are updated as the simulation progresses. On the other hand, Lagrangian methods focus on following the movement of individual particles or elements in the system, effectively translating the PDE problem into integrating some ODE system and thus

are not necessarily tied to specific grid discretizations of the domain, making the method appealing for many high dimensional problems. On the other hand, not all PDEs can be solved using a Lagrangian method, which is the main limitation of this class of methods. We refer to [114, 33, 59] for more details regarding the differences between the two approaches. The comparison here provides an important point of view for our work.

In recent years deep learning based methods for numerically solving PDEs have gained much attention with works such as Physics Informed Neural Networks (PINNs) [92, 93, 94] being developed and broadly tested on many different applications. By directly approximating the solution to a PDE with a neural network and reformulating the PDE solver as some optimization task, PINNs and many similar methods no longer tie themselves to specific grids in the domain, and thus are much easier to extend and apply to problems in higher dimensions. However deep learning based methods are not without their own challenges in the field, as a tradeoff, the main difficulty now shifts to accurately solving a non-convex optimization problem, given little theory support, much work remains to be done for such methods to achieve competitive accuracy and efficiency, see [43, 23, 24].

We divide our work into two main sections, in the first part we focus solely on PINNs and its applications to problems with spacial dimension smaller than or equal to 3. Traditionally these problems are often solved with Eulerian methods such as the finite element method (FEM) or the finite difference method (FDM), etc. While PINNs has achieved success in the area over a wide range of problems such as [65, 48], what can not be ignored is that successful implementation and fine tuning of a PINNs model can be difficult with many failures of convergence documented in [55, 23, 24]. As demonstrated in the pioneering work of [43] when directly compared against Eulerian methods such as the finite element method over several benchmark examples, it is also evident that PINNs cannot attain on-par accuracy to classic

numerical methods, showing another limitation of the methods.

In our work, we aim to provide a more detailed review of PINNs specifically on low dimensional problems, while answering the more important question, of whether neural networks are necessary for a machine learning based numerical PDE solver and if other function approximators can achieve comparable results. To do so we propose an alternative spline structure formulated as tensor products for function approximation similar to PINNs, we finalize the algorithm with our choice of sampling and optimization scheme for the model. We implement our proposed solver under some of the most popular machine learning libraries thus making it a direct substitute for PINNs. We test our proposed approach and compare it against PINNs over a wide selection of different PDE examples, we notice from the numerical results that our proposed approach can achieve solutions with improved accuracy in most examples when compared against PINNs. We also find that the polynomial structure of the B-spline function approximator can be optimized with reduced need for hyperparameter tuning and opens the opportunity for additional numerical techniques to be applied, such as domain decomposition and projection methods for fitting boundary conditions.

As we establish the case for low dimensional PDEs, the difficulties in solving high dimensional PDEs are differernt and also need addressing. Neural networks have demonstrated their potential in solving problems with inputs in high dimensions where grid discretization of the space can not often be relied upon. Such problems can include the Black–Scholes equation [76], the Allen-Cahn equation, and many other examples. In the second half of this thesis, we shift our focus to a specific type of Hamilton-Jacobi-Bellman (HJB) equations that can have high spatial dimensions and propose a neural network approach that can solve them efficiently. Solving the HJB equation is often considered a global solution method for certain (stochastic) optimal control problems, where for each problem the goal is to find a policy to control some

randomly perturbed dynamical systems to optimize a given objective functional.

As mentioned earlier one of the biggest challenges for efficiently solving the HJB equation is to overcome the Curse-of-Dimensionality (CoD). With the space dimension of a HJB equation easily reaching tens or even hundreds, numerical methods that rely on grid discretization of the spatial domain such as [13, 31, 57, 58, 88, 105] are not always practical. In our method, we propose to directly parameterize the solution to the HJB equation, also known as the value function using a neural network. Due to their universal approximation property, the use of neural networks as function approximators for HJB equations has been tested in [44, 91] and received some success, however their methods' applicability is still limited amongst optimal control problems due to their formulation of the learning problem.

One of the key differences that separates our method from similar ones is our sampling strategy for training the neural networks. In existing methods a random sampling scheme of the state-time space is usually used for obtaining the collocation points needed, we instead borrow ideas from Lagrangian methods and propose an alternative sampling scheme of the domain. To be more specific we use the stochastic Pontryagin maximum principle (PMP) [86] to link the sampling and the value function approximation; more precisely, we carefully define a Forward-Backward Stochastic Differential Equations (FBSDE) system which can serve as a substitute for the HJB equation. This allows us to follow the movement of each agent in the space-time domain and sample accordingly in only the relevant region of the domain. In fact for dynamical systems without noise, our choice of the FBSDE system is precisely the characteristics curves of the problem. We use several numerical experiments to demonstrate the effectiveness of our methods particularly for higher dimensional problems.

We briefly summarize our main contributions as follows. For low dimensional PDEs, we propose a Spline based function approximator and optimization method

that relies on accurate numerical integration, which can improve both accuracy and reproducibility when compared to PINNs. We demonstrate this through extensive numerical experiments. Our method partially fills in the gap between optimization based methods such as PINNs and traditional numerical methods using polynomial interpolation and solving linear systems. For HJB equations that can often have high dimensions, our proposed method builds upon existing solvers with improvements in both sampling strategy and loss function formulation. We demonstrate through various examples that our modified solver can largely increase convergence speed and model accuracy, most importantly our proposed method allows for tackling many problems that can not be solved with existing solvers.

We also want to mention some of the work that I participated in during my Ph.D. that I will not discuss in full in this thesis. In [79, 80] we propose a neural network approach for solving high dimensional optimal control problems that follow deterministic dynamics. The method shares a similar idea to our work in [63], we will therefore briefly cover the difference between deterministic and stochastic optimal control problems in relevant sections. My contribution to this work includes coding, numerical experimentation, and writing. One of the areas of application for optimal control and optimal transport theory is in generative modeling, specifically in the method of Continuous Normalizing Flow (CNF). We propose OT-Flow in [77], where we introduce an optimal transport reformulation of the generic CNF formula by adding artificial transport cost to the existing loss functional. Our solver also shares some similarities to [63] by penalizing the violation of the resulting HJB equation. We will however not discuss the application in detail in this work for the reason that both the optimal transport problem and the HJB equation are not always solved with high accuracy in practice, we therefore find it not suitable to include in this work. My contribution to the project primarily focuses on numerical experimentation and coding.

1.1 Overview of Thesis

We structure the thesis as follows. In Chapter 2, we present the mathematical background relevant to the thesis. We first introduce the necessary background on neural networks and deep learning. We follow this up by providing a brief overview of Physics Informed Neural Networks (PINNs) and FEM as two methods of interest for solving many PDEs in low dimensions. We will also introduce and discuss the relevant theory for stochastic optimal control problems and HJB equations, these set up the foundations for our method in the following chapters. In Chapter 3 we first provide a literature review regarding PINNs and its properties. We then propose and describe an alternative spline based model and training scheme to optimize the model. We follow this up by testing our spline structure and comparing the results with PINNs over a wide range of different numerical experiments. we also introduce some numerical techniques that can further improve model accuracy in this chapter. The related work has yet to be published. In Chapter 4 we propose a neural network based approach for solving stochastic optimal control problems and corresponding HJB equations at the same time. We test our proposed method over several different examples and present the numerical results in the latter half of the chapter, with more information and details in relevant publications. In the last Chapter, we conclude our work and point out several open questions and possible future directions for continuation of our current work.

Chapter 2

Mathematical Background

For the second chapter, We provide the necessary mathematical backgrounds to better understand and provide theoretical support for our work. We divide the chapter into several different sections discussing PDEs with dimensions from low to high. We start with a brief introduction to deep learning methods in section 2.1, in particular how they can be used in general optimization problems. We also touch on definitions such as neural networks, training and validation of a model, as well as common optimization methods used in the process. This forms the basis for much of the following work and discussions. We then include a section describing Physics Informed Neural Networks (PINNs) in section 2.2, a deep learning based method for solving general partial differential equations. Aside from the definition we will also describe some implementation details. We also provide some brief notes on the classic finite element method for solving PDEs in section 2.3 using a simple Poisson equation example, we will rely heavily on the FEM for problems without analytic solutions. Here for both PINNs and FEM, we look at PDEs with dimensions smaller than or equal to three. Lastly, we attribute a main section to the concept of deterministic and stochastic optimal control problems in section 2.4, we follow it up with two crucial and relevant results for solving these problems, namely the Pontryagin Maximum

Principle and the HJB equation with its modified form, both of which will be of high importance in the following work.

2.1 Neural Networks

Aside from its applications in the field of imaging, generative modeling and natural language processing, machine learning and deep learning methods also play important roles in applied mathematics,physics and engineering. We hereby provide a brief introduction of neural networks (NN), as well as training, validation, and some applications of neural networks that are crucial to some of our work.

At its core, neural networks represent a class of function approximators that use a combination of linear and nonlinear transformations to satisfy different tasks such as regression or classification. Such formulation allows for additional flexibility and function approximation in high dimensions, which is not always possible for traditional methods that rely on basis functions. However, the added complexity also brings many challenges to convergence and optimization.

2.1.1 Definition

Consider the following definition

$$\mathcal{NN}_0(\boldsymbol{x};\boldsymbol{\theta}) = \mathrm{act}(\boldsymbol{K}\boldsymbol{x} + \boldsymbol{b}). \tag{2.1}$$

This forms the basis of a neural network layer, also denoted as a single layer network, which combines the linear transformation $\boldsymbol{K}\boldsymbol{x}+\boldsymbol{b}$ with a nonlinear activation function. Here the input $\boldsymbol{x} \in \mathbb{R}^d$. Matrices $\boldsymbol{K} \in \mathbb{R}^{m\times d}$ and $\boldsymbol{b} \in \mathbb{R}^m$ can be arbitrary. We often use $\boldsymbol{\theta} = (\boldsymbol{K}, \boldsymbol{b})$ to denote all the linear parameters (trainable weights). The main difference that separates neural networks and a simple affine transformation is the activation function, here denoted by $\mathrm{act}(\cdot)$. Activation functions are nonlinear

functions applied element-wise to the input $\mathrm{act}(\boldsymbol{x}) = [\mathrm{act}(\mathrm{x}_1), \mathrm{act}(\mathrm{x}_2), \ldots, \mathrm{act}(\mathrm{x}_\mathrm{d})]^\top$. Some common choices for activation functions include Softmax, ReLU, and leaky ReLU, etc, see [50, 2, 110]. In our work we primarily focus on the Tanh function, also known as the hyperbolic tangent function, which reads

$$\tanh(\boldsymbol{x}) = \frac{\sinh(\boldsymbol{x})}{\cosh(\boldsymbol{x})} = \frac{e^{2\boldsymbol{x}} - 1}{e^{2\boldsymbol{x}} + 1}. \tag{2.2}$$

In general different activation functions are used for different tasks. Lastly the output of a single layer network $\mathcal{NN}_0(\boldsymbol{x}; \boldsymbol{\theta}) \in \mathbb{R}^m$.

A single layer network is also referred to as a layer or a dense layer when used to construct more complicated forms of neural networks. The most common form of neural networks is the Multi-layer Perceptron (MLP), which takes the form

$$\begin{aligned} \boldsymbol{a}_0 &= \mathrm{act}(\boldsymbol{K}_0 \boldsymbol{x} + \boldsymbol{b}_0) \\ \boldsymbol{a}_{i+1} &= \mathrm{act}(\boldsymbol{K}_{\mathrm{i}+1} \boldsymbol{a}_\mathrm{i} + \boldsymbol{b}_{\mathrm{i}+1}), \quad 0 \leq \mathrm{i} \leq \mathrm{M} - 2 \\ \mathcal{NN}(\boldsymbol{x}; \boldsymbol{\theta}_{\mathcal{NN}}) &= \mathrm{act}(\boldsymbol{K}_\mathrm{M} \boldsymbol{a}_{\mathrm{M}-1} + \boldsymbol{b}_\mathrm{M}), \end{aligned} \tag{2.3}$$

given input $\boldsymbol{x}$ the output is calculated through the combination of M different layers. We refer to this as a M-layer perceptron, see [87]. The parameters of the network $\boldsymbol{\theta} = (\boldsymbol{K}_0, \boldsymbol{b}_0, \ldots, \boldsymbol{K}_M, \boldsymbol{b}_M)$ consists of parameters from each layer, each $\boldsymbol{K}_i \in \mathbb{R}^{m_i \times m_{i-1}}$ and $\boldsymbol{b}_i \in \mathbb{R}^{m_i}$ can take different shapes.

ResNet One architecture that we also focus on is the Deep Residual Network (ResNet), introduced in [45], which has the formulation

$$\begin{aligned} \boldsymbol{a}_0 &= \mathrm{act}(\boldsymbol{K}_0 \boldsymbol{x} + \boldsymbol{b}_0) \\ \boldsymbol{a}_{i+1} &= \boldsymbol{a}_i + \mathrm{act}(\boldsymbol{K}_{\mathrm{i}+1} \boldsymbol{a}_\mathrm{i} + \boldsymbol{b}_{\mathrm{i}+1}), \quad 0 \leq \mathrm{i} \leq \mathrm{M} - 2 \\ \mathcal{NN}(\boldsymbol{x}; \boldsymbol{\theta}_{\mathcal{NN}}) &= \boldsymbol{a}_{M-1} + \mathrm{act}(\boldsymbol{K}_\mathrm{M} \boldsymbol{a}_{\mathrm{M}-1} + \boldsymbol{b}_\mathrm{M}). \end{aligned} \tag{2.4}$$

Notice here at each layer information from the previous layer is also added. ResNet has shown wide applicability across many applications, we also adopt it for some of our tests.

In general, increasing the width of a layer (increasing m in a single layer example) and adding more layers improves the expressiveness of neural networks, i.e. it allows the network to have more complexity and can approximate complex functions. However, it will also increase the computation overhead and difficulty of a given problem.

2.1.2 Training of a Neural Network

Mathematically, a deep learning problem can often be formulated as an optimization problem

$$\min_{\boldsymbol{\theta}} J(\mathcal{NN}(\boldsymbol{x}, \boldsymbol{\theta})). \tag{2.5}$$

Here $\mathcal{NN}$ is the output of the neural network, and $\boldsymbol{x}$ is the input of the neural network model, we use J or *Loss* to represent the objective function, which is a measurement of the discrepancy between the current output of the model and the desired function values. If explicit labeled data is given for J, that is for each $\boldsymbol{x}$ a label $\boldsymbol{y}$ is provided. We often refer to the problem as a supervised learning problem and eq. (2.5) can be further written as minimizing $J(\mathcal{NN}(\boldsymbol{x}, \boldsymbol{\theta}), \boldsymbol{y})$. However, if only partial data exist or we are given without data and use other metrics for J, it is sometimes called a semi-supervised learning problem or unsupervised learning problem. In our work, we primarily look at problems in the latter two categories.

We refer to the process of solving the minimization problem as "training" a neural network. Note that due to the nonconvex nature of a neural network, it is generally a difficult problem to solve with limited convergence theory. We here provide some discussion that will be useful for later sections.

Back Propagation The forward problem in a machine learning setting usually means evaluating the objective J given some $\boldsymbol{x}$, while the backward process is where one updates the trainable parameters of a neural network. Gradient evaluation is therefore crucial in the resulting optimization step, for neural networks we resort to Automatic Differentiation (AD) for calculating the gradient, see [71]. AD offers an efficient way of calculating gradient information regardless of architecture. Additional tools and resources are also available such as [75] if Hessian information is also needed.

Stochastic Gradient Method Gradient descent method for training neural networks has the general form

$$\boldsymbol{\theta} = \boldsymbol{\theta} - \eta \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}, \boldsymbol{x}). \tag{2.6}$$

Here η is the learning rate or step size and is usually picked by the user. $\boldsymbol{x}$ is the input of the network. When using the entire dataset as input we refer to it as batch gradient descent. However, it is much more common to use only a portion of the total inputs (training data) at each update, this saves computation resources yet remains effective in most instances.

There have been many alternatives to the standard stochastic gradient descent (SGD) methods, with many listed in [96]. In our work, we also use Adaptive Moment Estimation (Adam), which is a momentum-based subgradient method. Newton or quasi-Newton methods such as L-BFGS are also sometimes used, though mostly for sample average methods and full batch training.

2.2 Physics Informed Neural Networks

In this section, we offer a brief introduction to the idea of Physics Informed Neural Networks. PINNs is a deep learning based method for solving partial differential equations. The method differs from traditional numerical PDE methods such as

finite difference method (FDM) or finite element method (FEM) in a few key ways. Instead of relying on a mesh discretization of the space, PINNs directly parameterize the solution to some differential equation using a neural network. The problem can then be formulated as an optimization problem with respect to the residual of the given PDE, which can be solved with a stochastic gradient descent method based on some sampling strategy over the given domain.

2.2.1 Neural Network Approximation

Being the major difference between traditional methods and PINNs, PINNs approximates the solution using a neural network, unlike FD which solves a discretized version of the PDE, or FEM which also relies on some mesh for creating a function space for its solution. Neural networks allow one to get rid of the need for meshing, which can be appealing since meshing remains the main challenge for traditional methods to tackle high dimensional problems.

Development in network architecture for PINNs is rather sparse. For most applications such as [43, 94, 23, 24, 29], multilayer perceptrons or their variants are used with different activation functions. Empirically deep networks with a number of layers over 5 are often needed for learning complicated functions, see [43, 92, 93], which can lead to challenging optimization problems.

2.2.2 Loss Function

Consider a very general class of partial differential equations

$$\mathcal{A}u(\boldsymbol{x}, t) = f(\boldsymbol{x}, t), \qquad x \in \Omega, t \in [0, T]. \tag{2.7}$$

Here the function $u : \bar{\Omega} \times [0, T] \to \mathbb{R}^n$ is the solution to the given PDE. Here we assume the spatial domain Ω is open, bounded, and connected. $[0, T]$ is the time interval and

here we assume time starts at 0. We use $\mathcal{A}$ to denote the differential operator and nonlinear function $f : \Omega \times [0, T] \to \mathbb{R}^n$ for the source term. The boundary and initial condition for the problem are also defined

$$\begin{cases} \mathcal{B}u(\boldsymbol{x}, t) &= g(\boldsymbol{x}, t), \qquad \boldsymbol{x} \in \partial\Omega, t \in [0, T], \\ u(\boldsymbol{x}, 0) &= h(\boldsymbol{x}), \qquad \boldsymbol{x} \in \Omega. \end{cases} \tag{2.8}$$

It is important to note for stationary problems such as the Poisson equation, where the solution u is not time-dependent, one will need to correspondingly modify eq. (2.7) and eq. (2.8). The state variable $\boldsymbol{x}$ can be of different dimensions depending on the problems.

As mentioned in previous sections in PINNs we choose to approximate the solution u with a neural network, usually denoted by $u_\theta(\boldsymbol{x}, t)$ with θ being the network weights that are optimized during training. For the objective functional, PINNs directly minimizes the residual of the strong form, that is eq. (2.7). We can write the loss function as

$$\begin{aligned} J(\theta) =& \alpha_1 \left(\frac{1}{N_f} \sum_{i=1}^{N_f} \| \mathcal{A} u_\theta(\boldsymbol{x}_f^i, t_f^i) - f(\boldsymbol{x}_f^i, t_f^i) \|^2 \right) \\ &+ \alpha_2 \left(\frac{1}{N_g} \sum_{j=1}^{N_g} \| \mathcal{B} u_\theta(\boldsymbol{x}_g^j, t_g^j) - g(\boldsymbol{x}_g^j, t_g^j) \|^2 \right) + \alpha_3 \left(\frac{1}{N_h} \sum_{k=1}^{N_h} \| u_\theta(\boldsymbol{x}_h^k, 0) - h(\boldsymbol{x}_h^k) \|^2 \right). \end{aligned} \tag{2.9}$$

Here, N_f is the number of collocation points $(\boldsymbol{x}_f^i, t_f^i) \in \Omega \times [0, T]$ for $i = 1, 2, \ldots, N_f$ sampled for the PDE residual in the loss. Similarly, $(\boldsymbol{x}_g^j, t_g^j) \in \partial\Omega \times [0, T]$ for $j = 1, 2, \ldots, N_g$ are points sampled from the boundary and $\boldsymbol{x}_h^k$ for $k = 1, 2, \ldots, N_h$ are sampled for learning the initial condition. $\alpha_1, \alpha_2, \alpha_3$ are weights that balance between different terms in the loss function and are usually provided by the user. In practice, properly choosing $\{\alpha_i\}_{i=1,2,3}$ is often crucial to the accuracy of a PINNs

model. Notation wise $Loss(\theta)$ or $Loss(\boldsymbol{x};\theta)$ are also commonly used in literature for eq. (2.9). Lastly, the minimization problem is often presented as

$$\min_{\theta} \mathbb{E}\left[J(\theta)\right], \tag{2.10}$$

given fixed collocation points $\{\boldsymbol{x}_f, t_f, \boldsymbol{x}_g, t_g, \boldsymbol{x}_h\}$ the loss function eq. (2.9) forms a deterministic optimization problem. However in most cases the collocation points are resampled at each epoch, we therefore instead solve the problem in eq. (2.10), where the expected value is taken w.r.t. the sample points. More details can be found in [92, 93, 94]. We usually refer to the problem as solved given the best parameters θ obtained through training, however, unlike methods such as the FEM, solving a PDE using PINNs does not guarantee high solution accuracy in most cases.

2.2.3 Sampling and Training

Given the non-convexity nature of the loss function in eq. (2.9) when $\mathcal{A}$ is nonlinear and the use of neural networks, small batch training is often required for optimizing a PINNs model. Suitable techniques for sampling and resampling collocation points in eq. (2.9) are important in PINNs setting, aside from the uniform sampling from the given domain, Latin Hypercube Sampling [102] is another commonly used method.

For optimizing the parameters θ, automatic differentiation is the go-to method for gradient evaluation of the loss function. SGD, ADAM, and L-BFGS are some of the widely used methods for updating the parameters. In practice, it is also common to combine ADAM and L-BFGS into a 2-step training scheme for better accuracy, as illustrated in [43].

To sum up, alongside the choice of function approximators, sampling and optimization of the learning problem also play an important role in the performance of a PINNs model, we will demonstrate said point with more details in the following

chapters.

2.3 Finite Element Method

The finite element method (FEM) is one of the standard and most widely used methods for spacial discretization of many partial differential equations, particularly for those without an analytic solution. Throughout the remainder of the paper we will constantly be referring back to the FEM for numerical solutions to multiple examples, as such we find it necessary to provide some basics of the FEM in this section.

We believe it is the best way to introduce the key components of FEM through an example as in [43, 81]. Consider the simple Poisson equation

$$\Delta u(\boldsymbol{x}) = f(\boldsymbol{x}), \quad \boldsymbol{x} \in \Omega,$$

with homogeneous Dirichlet boundary $u(\boldsymbol{x}) = 0$ for $\boldsymbol{x} \in \partial\Omega$. We also assume the right hand side $f \in \mathcal{L}^2(\Omega)$.

At its core, the FEM aims to find a weak solution to the problem. That is, we want to find the solution u on some function space U such that for all functions $v(\boldsymbol{x}) \in V$, we have

$$\int_\Omega v(\boldsymbol{x})(\Delta u(\boldsymbol{x}) - f(\boldsymbol{x}))\mathrm{d}\boldsymbol{x} = 0.$$

By applying the Green's Formula and taking into account the Dirichlet boundary condition, we can get

$$\int_\Omega (\langle\, \nabla v(\boldsymbol{x}), \nabla u(\boldsymbol{x})\rangle \; - v(\boldsymbol{x}) f(\boldsymbol{x}))\mathrm{d}\boldsymbol{x} = 0.$$

the most common choices for both U and V would be the Sobolev space $H_0^1(\Omega)$ of functions defined on $\Omega \to \mathbb{R}$ that have a weak derivative. Boundary conditions are also taken into consideration as indicated by the subscript 0. With the finite element

method, in order to solve the problem numerically, we replace $H_0^1(\Omega)$ with some finite dimensional space H, H can be characterized by its basis $\{\psi\}_{i=1}^N$, given the finite dimensional space we can rewrite the weak form into

$$-\int_\Omega \langle \nabla\psi_i(\boldsymbol{x}), \nabla \sum_{j=1}^N a_j\psi_j(\boldsymbol{x})\rangle \mathrm{d}\boldsymbol{x} = \int_\Omega \psi_i(\boldsymbol{x}) f(\boldsymbol{x}) \mathrm{d}\boldsymbol{x}$$

for all $i = 1, 2, \ldots, N$. The above equations can also be written as a linear system $Ba = F$ which has $N \times N$ matrix B with entry

$$B_{i,j} = -\int_\Omega \langle \nabla\psi_i(\boldsymbol{x}), \nabla\psi_j(\boldsymbol{x})\rangle \mathrm{d}\boldsymbol{x},$$

and right hand side vector F as

$$F = \left(\int_\Omega \psi_i(\boldsymbol{x}) f(\boldsymbol{x}) \mathrm{d}\boldsymbol{x}\right)_{i=1}^N.$$

By solving the linear system a numerical approximation of u on H takes the form $\hat{u} = \sum_{i=1}^N a_i\psi_i(\boldsymbol{x})$.

Notice the above formulation is only valid for homogeneous Dirichlet boundary conditions. With Neumann condition, the additional $\int_{\partial\Omega} vu\mathrm{d}x$ term needs to be added to the bilinear form, a more general function space $U = V = H^1(\Omega)$ will be used instead as well. Mixed boundary conditions can be treated similarly. Stationary PDEs other than the Poisson equation can be solved in a similar fashion by deriving the weak formulation using the Green's Formula.

2.3.1 Non-Stationary PDEs

For time dependent, non-stationary PDEs, one can apply FEM to the problem by treating time as an additional dimension for the space variable, however, a more common way for time dependent problems will be to apply FEM on only the space

variable after discretizing the system in time. As an example, we consider a general heat equation

$$\frac{\partial u(\boldsymbol{x},t)}{\partial t} = \Delta u(\boldsymbol{x},t) + F(u(\boldsymbol{x},t)), \qquad \boldsymbol{x} \in \Omega, t \in [0,T],$$

with initial condition $u(\boldsymbol{x},0) = u_0(\boldsymbol{x})$ for $\boldsymbol{x} \in \Omega$ and boundary condition of choice.

The main difference between stationary and non-stationary PDEs is the introduction of time discretization methods, here for the heat equation we consider a more stable implicit Euler scheme

$$u_{n+1} = u_n + \mathrm{d}t\Delta u_{n+1} + \mathrm{d}tF(u_{n+1}) \quad \text{for } n = 0, 1, \ldots, N_t.$$

Here we use n to denote the time stepping and $\mathrm{d}t$ is the size of the time steps. We use u_n to denote the approximation of the solution u at each time step. The weak form will then become

$$\int_\Omega v u_{n+1} \mathrm{d}x = \int_\Omega \left(v u_n \mathrm{d}\boldsymbol{x} - \mathrm{d}t\langle \nabla v, \nabla u_{n+1}\rangle + \mathrm{d}t v F(u_{n+1})\right) \mathrm{d}\boldsymbol{x}.$$

for $n = 0, 1, \ldots, N_t$. Note here we once again assume 0 Dirichlet boundary for simplicity. Given u_n a each time step, we can solve the weak problem with respect to u_{n+1} using a finite element discretization of the space. Also note that due to the use of an implicit time integration scheme, the resulting equation will be nonlinear, and iterative methods such as Newton's method will be required for recovering u_{n+1}. Other time stepping methods such as semi-explicit methods are also sometimes used, however we will not focus on those in our work. We refer to [81, 115] for a more thorough introduction to the method.

2.4 Stochastic Optimal Control and HJB Equations

One of the main focuses of the thesis is solving HJB equations, the importance of which is tied to their underlying stochastic optimal control (SOC) problems. In this section, we describe a classic type of SOC problems, provide some basics, and review the key results from SOC theory that are crucial to some of our work. Our discussion primarily follows [112] and we refer to this textbook for a more comprehensive background and some details that we omit here.

To be more specific, we first introduce the definition of the SOC problems we aim to solve, followed by a few key results: the stochastic Pontryagin Maximum Principle (PMP), the corresponding Hamilton-Jacobi-Bellman (HJB) equation, and its reformulation into a system of forward-backward stochastic differential equations (FBSDEs) obtained from a nonlinear version of the Feynman-Kac formula. Lastly, we will also highlight some major differences between SOC problems and their deterministic counterpart.

2.4.1 Stochastic Optimal Control Problem

Let $(\Omega, \mathcal{F}, \mathbb{F} = \{\mathcal{F}_t\}_{t\geq 0}, \mathbb{P})$ be a given complete probability space, $W(s)$ be a d-dimensional Brownian motion on $(\Omega, \mathcal{F}, \mathbb{F}, \mathbb{P})$ where we use s to denote the time. For a fixed initial state $\boldsymbol{x}$ at some time $0 < t < T < \infty$, we seek to control the randomly perturbed dynamical system some times also known as a stochastic differential equation (SDE)

$$\begin{cases} \mathrm{d}\boldsymbol{z}_{t,\boldsymbol{x}}(s) = f(s, \boldsymbol{z}_{t,\boldsymbol{x}}(s), \boldsymbol{u}_{t,\boldsymbol{x}}(s, \boldsymbol{z}_{t,\boldsymbol{x}}(s)))\mathrm{d}s + \sigma(s, \boldsymbol{z}_{t,\boldsymbol{x}}(s))\mathrm{d}W(s), s \in [t, T], \\ \boldsymbol{z}_{t,\boldsymbol{x}}(t) = \boldsymbol{x}. \end{cases} \tag{2.11}$$

Here, $\boldsymbol{z}_{t,\boldsymbol{x}} : [t,T] \to \mathbb{R}^d$ describes the state and $\boldsymbol{u}_{t,\boldsymbol{x}} : [t,T] \times \mathbb{R}^d \to U$ describes the control of the system, the function $\sigma : [t,T] \times \mathbb{R}^d \to \mathbb{R}^{d\times d}$ represents the diffusion coefficient, and $f : [t,T] \times \mathbb{R}^d \times U \to \mathbb{R}^d$ represents the drift of the system. We assume that the set of admissible controls $U \subset \mathbb{R}^k$ is closed. Above condition guarantees uniqueness of solution to the SDE eq. (2.11). Note here we require the control $\boldsymbol{u}$ to be independent of the diffusion σ, which is commonly true in many applications.

We seek to minimize the objective functional

$$J_{t,\boldsymbol{x}}(\boldsymbol{u}_{t,\boldsymbol{x}}) = \mathbb{E}\left\{G\big(\boldsymbol{z}_{t,\boldsymbol{x}}(T)\big) + \int_t^T L\big(s, \boldsymbol{z}_{t,\boldsymbol{x}}(s), \boldsymbol{u}_{t,\boldsymbol{x}}(s, \boldsymbol{z}_{t,\boldsymbol{x}}(s))\big)\,\mathrm{d}s\right\}, \tag{2.12}$$

which is comprised of the running cost $L : [t,T] \times \mathbb{R}^d \times U \to \mathbb{R}$ and the terminal cost $G : \mathbb{R}^d \to \mathbb{R}$. Here, the expectation is taken with respect to perturbance of the dynamics eq. (2.11) over all admissible control processes $\boldsymbol{u}_{t,\boldsymbol{x}}(s, \boldsymbol{z}_{t,\boldsymbol{x}}(s))$ that is described by the probability measure $\mathbb{P}$. We assume sufficient regularity conditions on f, σ, G, and L, see [112, Chapter 2] for a list of assumptions.

The value function assigns the optimal cost-to-go to any initial state is defined by

$$\Phi(t,\boldsymbol{x}) = \inf_{\boldsymbol{u}_{t,\boldsymbol{x}}} J_{t,\boldsymbol{x}}(\boldsymbol{u}_{t,\boldsymbol{x}}), \tag{2.13}$$

and a solution $\boldsymbol{u}^*_{t,\boldsymbol{x}}$ to eq. (2.13) incurring this minimum value is called an optimal control. Here to differentiate from the general PDE solutions in previous sections we use Φ to denote the value function.

The Generalized Hamiltonian The functional $H : [t,T] \times \mathbb{R}^d \times \mathbb{R}^d \times \mathbb{R}^{d\times d} \to \mathbb{R} \cup \{\infty\}$, is a key ingredient for the SOC theory in some of the following sections and the backbone of some of the work we did. For the problem defined in eq. (2.11) and

eq. (2.12) we can write H as

$$H(s, \boldsymbol{z}, \boldsymbol{p}, \boldsymbol{M}) = \sup_{\boldsymbol{u} \in U} \mathcal{H}(s, \boldsymbol{z}, \boldsymbol{p}, \boldsymbol{M}, \boldsymbol{u}), \tag{2.14}$$

where $\boldsymbol{p}$ and $\boldsymbol{M}$ are called adjoint variables and

$$\mathcal{H}(s, \boldsymbol{z}, \boldsymbol{p}, \boldsymbol{M}, \boldsymbol{u}) = \frac{1}{2}\mathrm{tr}\left(\sigma(s, \boldsymbol{z})\boldsymbol{M}\right) + \boldsymbol{p} \cdot f(s, \boldsymbol{z}, \boldsymbol{u}) - L(s, \boldsymbol{z}, \boldsymbol{u}).$$

It can be assumed that there exists a unique minimizer of the Hamiltonian eq. (2.14) given some assumptions from [112, Chapter 3] .

To make it notation wise convenient, in the rest of the paper, we drop the second argument for the controls and denote controls by $\boldsymbol{u}_{t,\boldsymbol{x}}(s)$. The Hamiltonian is of great importance in both the Pontryagin Maximum Principle as well as the Dynamic Programming Principle, as can be seen in the following sections.

2.4.2 Stochastic Pontryagin Maximum Principle

The stochastic Pontryagin Maximum Principle (PMP) provides first-order necessary conditions for the SOC problem and also states that the optimal control $\boldsymbol{u}^*_{t,\boldsymbol{x}}$ must satisfy an (extended) Hamiltonian system along the optimal state and adjoint trajectory. This is of high importance in solving SOC problems and is made precise by the following result from [112, Theorem 3.2, Chapter 3].

Theorem 2.4.1. *[112, Theorem 3.2, Chapter 3] Assume that $(\boldsymbol{z}^*_{t,\boldsymbol{x}}, \boldsymbol{u}^*_{t,\boldsymbol{x}})$ is an optimal pair that solves eq.* (2.11) *and eq.* (2.12)*. Then there exist adjoint states $\boldsymbol{p}_{t,\boldsymbol{x}} \colon [t, T] \to \mathbb{R}^d$ and $\boldsymbol{M}_{t,\boldsymbol{x}} \colon [t, T] \to \mathbb{R}^{d \times d}$ satisfying the adjoint equation*

$$\begin{cases} \mathrm{d}\boldsymbol{p}_{t,\boldsymbol{x}}(s) = & \boldsymbol{M}_{t,\boldsymbol{x}}(s)\mathrm{d}W(s) - \nabla_{\boldsymbol{z}}\mathcal{H}\big(s, \boldsymbol{z}^*_{t,\boldsymbol{x}}(s), \boldsymbol{p}_{t,\boldsymbol{x}}(s), \boldsymbol{M}_{t,\boldsymbol{x}}(s), \boldsymbol{u}^*_{t,\boldsymbol{x}}(s)\big)\mathrm{d}s \\ \boldsymbol{p}_{t,\boldsymbol{x}}(T) = & -\nabla_{\boldsymbol{z}} G\big(\boldsymbol{z}^*_{t,\boldsymbol{x}}(T)\big), \end{cases} \tag{2.15}$$

where $s \in [t, T]$ and the optimal control satisfies

$$\boldsymbol{u}^*_{t,\boldsymbol{x}}(s) = \arg\max_{\boldsymbol{u} \in U} \ \mathcal{H}\big(s, \boldsymbol{z}^*_{t,\boldsymbol{x}}(s), \boldsymbol{p}_{t,\boldsymbol{x}}(s), \boldsymbol{M}_{t,\boldsymbol{x}}(s), \boldsymbol{u}(s)\big) \tag{2.16}$$

for almost all $s \in [t, T]$, $\mathbb{P}$-almost surely.

We hereby highlight the fact that the optimal control defined in eq. (2.16) only depends on the adjoint variable $\boldsymbol{p}_{t,\boldsymbol{x}}$ but not on $\boldsymbol{M}_{t,\boldsymbol{x}}$ since $\sigma(\cdot, \cdot)$ does not depend on the control. This is key to reducing the complexity of the SOC solutions.

We further assume that there exists a unique continuous closed-form solution to eq. (2.16). Although not demonstrated in this work, this assumption can be weakened to include implicitly defined functions as long as they can be obtained efficiently; this allows, for example, modeling more general convex running costs.

We note that when the control satisfies eq. (2.16), the dynamics in eq. (2.11) can be rewritten in terms of the Hamiltonian and is equal to

$$\begin{cases} \mathrm{d}\boldsymbol{z}^*_{t,\boldsymbol{x}}(s) = \nabla_{\boldsymbol{p}} \mathcal{H}\big(s, \boldsymbol{z}^*_{t,\boldsymbol{x}}(s), \boldsymbol{p}_{t,\boldsymbol{x}}(s), \boldsymbol{M}_{t,\boldsymbol{x}}(s), \boldsymbol{u}^*_{t,\boldsymbol{x}}(s)\big)\mathrm{d}s + \sigma(s, \boldsymbol{z}^*_{t,\boldsymbol{x}}(s))\mathrm{d}W(s), \\ \boldsymbol{z}^*_{t,\boldsymbol{x}}(t) = \boldsymbol{x}. \end{cases} \tag{2.17}$$

The system of equations eqs. (2.15) to (2.17) is called the stochastic Hamiltonian system, where the maximum condition eq. (2.16) corresponds to the variational inequality for the control.

Despite its importance, finding a tuple $(\boldsymbol{z}^*_{t,\boldsymbol{x}}, \boldsymbol{u}^*_{t,\boldsymbol{x}}, \boldsymbol{p}_{t,\boldsymbol{x}}, \boldsymbol{M}_{t,\boldsymbol{x}})$ that satisfies the PMP can be difficult. However, when the value function Φ is differentiable, $(\boldsymbol{p}_{t,\boldsymbol{x}}, \boldsymbol{M}_{t,\boldsymbol{x}})$ satisfying eq. (2.15) can be directly obtained from Φ, this is formalized in the following theorem that, with weaker assumptions, can be found in [112, Chapter 5].

Theorem 2.4.2. *[112, Chapter 5] Assume that $\boldsymbol{u}^*_{t,\boldsymbol{x}}$ is an optimal control and $\Phi \in$*

$C^{1,3}([t,T]\times\mathbb{R}^d)$. *Then*

$$\boldsymbol{p}_{t,\boldsymbol{x}}(s) = -\nabla_{\boldsymbol{z}}\Phi\big(s,\boldsymbol{z}^*_{t,\boldsymbol{x}}(s)\big) \;\; \textit{and} \;\; \boldsymbol{M}_{t,\boldsymbol{x}}(s)) = -\sigma(s,\boldsymbol{z}^*_{t,\boldsymbol{x}}(s))^\top\nabla^2_{\boldsymbol{z}}\Phi\left(s,\boldsymbol{z}^*_{t,\boldsymbol{x}}(s)\right) \quad (2.18)$$

solve eq. (2.15).

Theorem 2.4.2 along with eq. (2.16) collectively serve to express the optimal control $\boldsymbol{u}^*$ as

$$\boldsymbol{u}^*_{t,\boldsymbol{x}}(s) = \boldsymbol{u}^*_{t,\boldsymbol{x}}\left(s,\boldsymbol{z}^*_{t,\boldsymbol{x}}(s), -\nabla_{\boldsymbol{z}}\Phi\big(s,\boldsymbol{z}^*_{t,\boldsymbol{x}}(s)\big)\right). \quad (2.19)$$

for almost all $s\in[t,T]$ and $\mathbb{P}$-almost surely. This relation along with eq. (2.17) is one of the key ingredients of some of our work. Additionally, there are a few things we would like to point out here. Equation (2.19) characterizes the optimal control in what some would call a feedback or feedback closed-loop form, which is of utmost importance in many real-life applications. Once the value function is recovered with its gradient readily available, the optimal control $\boldsymbol{u}^*$ can be quickly calculated at any given point in time and space, avoiding the re-computation overhead for cases when multiple evaluations are needed for different times or states. Using the feedback form also means separately solving and storing the adjoint variables are no longer necessary.

2.4.3 Hamilton-Jacobi-Bellman Equation

In the previous section, we studied how the solution to the problem eq. (2.12) for initial states can be obtained from the values function Φ. To help approximate the value function Φ, we also use the fact that Φ satisfies the Hamilton-Jacobi-Bellman (HJB) equation, which is a result of the Dynamic Programming (DP) method or Bellman's principle. We state the following result taken from [112] under suitable assumptions, see also [84, Remark 3.4.4, Theorem 3.5.2].

Theorem 2.4.3. *[112, Propositon 3.5, Chapter 4] Assume that the value function* $\Phi \in C^{1,2}([t,T] \times \mathbb{R}^d)$. *Then* Φ *satisfies the HJB equation*

$$\begin{aligned} &-\partial_s \Phi(s,\boldsymbol{z}) + H\big(s,\boldsymbol{x},-\nabla_{\boldsymbol{z}}\Phi(s,\boldsymbol{z}),-\sigma(s,\boldsymbol{z})^\top \nabla^2_{\boldsymbol{z}}\Phi(s,\boldsymbol{z})\big) = 0,\ \forall (s,\boldsymbol{z}) \in [t,T)\times\mathbb{R}^d, \\ &\Phi(T,\boldsymbol{z}) = G(\boldsymbol{z}). \end{aligned} \tag{2.20}$$

Furthermore, if $\Psi \in C^{1,2}([0,T]\times\mathbb{R}^d)$ *is a solution of eq.* (2.20) *and* $\boldsymbol{u}^*_{t,\boldsymbol{x}}$ *is such that*

$$\boldsymbol{u}^*_{t,\boldsymbol{x}}(s) \in \underset{\boldsymbol{u}\in U}{\arg\max}\ \mathcal{H}\big(s,\boldsymbol{z}^*_{t,\boldsymbol{x}}(s),-\nabla_{\boldsymbol{z}}\Psi\big(s,\boldsymbol{z}^*_{t,\boldsymbol{x}}(s)\big),-\nabla^2_{\boldsymbol{z}}\Psi\big(s,\boldsymbol{z}^*_{t,\boldsymbol{x}}(s)\big),\boldsymbol{u}\big) \tag{2.21}$$

for almost all $s \in [t,T]$ *and* $\mathbb{P}$*-almost surely. Then* $\Psi = \Phi$, *and* $\boldsymbol{u}^*_{t,\boldsymbol{x}}$ *is an optimal control.*

The smoothness of Φ can be relaxed to continuity in the weaker sense of viscosity solutions [112, Section 5, Chapter 4]. Using the definition of the Hamiltonian in eq. (2.20) we get the HJB equation as the following second-order parabolic PDE:

$$\begin{cases} -\partial_s \Phi(s,\boldsymbol{z}) - \dfrac{1}{2}\mathrm{tr}(\sigma(s,\boldsymbol{z})\sigma(s,\boldsymbol{z})^\top \nabla^2_{\boldsymbol{z}}\Phi(s,\boldsymbol{z})) - \nabla_{\boldsymbol{z}}\Phi(s,\boldsymbol{z})\cdot f(s,\boldsymbol{z},\boldsymbol{u}^*) \\ \qquad\qquad - L(s,\boldsymbol{z},\boldsymbol{u}^*) = 0,\ \forall\,(s,\boldsymbol{z}) \in [t,T)\times\mathbb{R}^d, \\ \Phi(T,\boldsymbol{z}) = G(\boldsymbol{z}(T)). \end{cases} \tag{2.22}$$

In addition, by the envelope theorem, it follows that $\nabla_{\boldsymbol{p}}\mathcal{H} = \nabla_{\boldsymbol{p}}H$ and $\nabla_{\boldsymbol{M}}\mathcal{H} = \nabla_{\boldsymbol{M}}H$. This simplifies the computation of optimal trajectories, which can now be

expressed via the value function as:

$$\begin{cases} \mathrm{d}\boldsymbol{z}^*_{t,\boldsymbol{x}}(s) = \nabla_{\boldsymbol{p}} H\big(s, \boldsymbol{z}^*_{t,\boldsymbol{x}}(s), -\nabla_{\boldsymbol{z}}\Phi\big(s, \boldsymbol{z}^*_{t,\boldsymbol{x}}(s)\big), -\sigma(s, \boldsymbol{z}^*_{t,\boldsymbol{x}}(s))^\top \nabla^2_{\boldsymbol{z}}\Phi\big(s, \boldsymbol{z}^*_{t,\boldsymbol{x}}(s)\big)\big)\mathrm{d}s \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad + \sigma(s, \boldsymbol{z}^*_{t,\boldsymbol{x}}(s))\mathrm{d}W(s), \\ \boldsymbol{z}^*_{t,\boldsymbol{x}}(t) = \boldsymbol{x}. \end{cases} \tag{2.23}$$

Compare to eq. (2.11) these modified dynamics do not explicitly involve the control, which reduces the problem solely to the state variables. This idea was proposed for deterministic optimal control problems in [78], which we will elaborate further in later sections.

Despite it being appealing, HJB equations are intrinsically hard to solve for various reasons, in particular for the curse of dimensionality (CoD). Notice in eq. (2.20) that the dimension of the PDE depends on the state variable. For SOC problems it is common to have state variables with dimensions beyond a grid based method can handle. A core objective for some of our work is to address this issue, which we will discuss in the following chapters.

2.4.4 FBSDE Formulation

Regarding the CoD as described for general HJB equations, one way to avoid the need for a spatial discretization of the HJB equation eq. (2.22) is to use a non-linear version of the Feynman-Kac formula and obtain an equivalent system of stochastic differential equations. To be more specific the resulting system has an easily exploitable forward-backward structure and is most commonly referred to as a FBSDE system. This idea has been applied to a variety of nonlinear parabolic/elliptic PDEs; see, for example, [7, 69, 22, 82, 84].

The most obvious choice for formulating the FBSDE system given an HJB equa-

tion is to use eq. (2.23) as the forward system to sample trajectories. Along those trajectories, we note that the solution to the HJB equation eq. (2.22) must satisfy the backward SDE

$$\begin{cases} \mathrm{d}\Phi(s, \boldsymbol{z}(s)) & = \nabla_{\boldsymbol{z}}\Phi(s, \boldsymbol{z}(s))^\top \sigma(s, \boldsymbol{z}(s))\, \mathrm{d}W(s) - L(s, \boldsymbol{z}(s), \boldsymbol{u}^*(s))\mathrm{d}s, \\ \Phi(T, \boldsymbol{z}(T)) & = G(\boldsymbol{z}(T)). \end{cases} \tag{2.24}$$

It is important to point out that FBSDEs can be derived for general semi-linear parabolic PDEs and are not limited to HJB equations, moreover for each defined PDE, given different forward dynamics we correspondingly get different BSDE for the system. Choosing the proper FBSDE system often requires additional consideration, for HJB equation eq. (2.22) our choice of the forward system eq. (2.23) is the key difference from other existing works. As a more common choice, for example, [44] and [91] remove the drift term entirely from the forward dynamics and use only the standard Brownian motion for exploration, which read

$$\begin{cases} \mathrm{d}\boldsymbol{z}^*_{t,\boldsymbol{x}}(s) = \sigma(s, \boldsymbol{z}^*_{t,\boldsymbol{x}}(s))\mathrm{d}W(s), \\ \boldsymbol{z}^*_{t,\boldsymbol{x}}(t) = \boldsymbol{x} \end{cases} \tag{2.25}$$

with corresponding BSDE. Despite this being a valid FBSDE system for eq. (2.22), we advocate for including the control in the dynamics as motivated by stochastic PMP eq. (2.17). As we will demonstrate in later chapters through numerical experiments, focusing the sampling along optimal trajectories can lead to more accurate and efficient value function approximations.

2.4.5 Relation to Deterministic Optimal Control

Given the context, we think it is appropriate to also briefly introduce the deterministic counterpart to SOC problems, how they relate to each other and some of the key

differences that affect how one treats them.

Definition of a deterministic optimal control problem can be easily derived from eq. (2.11) by removing the volatility term

$$\mathrm{d}\boldsymbol{z}_{t,\boldsymbol{x}}(s) = f(s, \boldsymbol{z}_{t,\boldsymbol{x}}(s), \boldsymbol{u}_{t,\boldsymbol{x}}(s, \boldsymbol{z}_{t,\boldsymbol{x}}(s)))\mathrm{d}s, \qquad \boldsymbol{z}_{t,\boldsymbol{x}}(t) = \boldsymbol{x}. \tag{2.26}$$

for $s \in [t, T]$ with similar assumptions to eq. (2.11). The objective function to be minimized is defined as

$$J_{t,\boldsymbol{x}}(\boldsymbol{u}_{t,\boldsymbol{x}}) = G\big(\boldsymbol{z}_{t,\boldsymbol{x}}(T)\big) + \int_t^T L\big(s, \boldsymbol{z}_{t,\boldsymbol{x}}(s), \boldsymbol{u}_{t,\boldsymbol{x}}(s, \boldsymbol{z}_{t,\boldsymbol{x}}(s))\big)\,\mathrm{d}s, \tag{2.27}$$

notice compare to eq. (2.12) the expected value is no longer needed. This is because for deterministic problems, given initial state and time pair $(t, \boldsymbol{x})$ without the volatility term the optimal solution $\boldsymbol{z}_{t,\boldsymbol{x}}(s)$ is uniquely defined. Value function $\Phi(s, \boldsymbol{z})$ for deterministic OC problem is defined exactly as eq. (2.13), as for the Hamiltonian it has form

$$\mathcal{H}(s, \boldsymbol{z}, \boldsymbol{p}, \boldsymbol{u}) = \boldsymbol{p} \cdot f(s, \boldsymbol{z}, \boldsymbol{u}) - L(s, \boldsymbol{z}, \boldsymbol{u}).$$

This differs from eq. (2.14) in the second order term. Also notice here in the deterministic case the adjoint variables only consist of $\boldsymbol{p}$. With this observation, the PMP for deterministic OC problems can be very much simplified to a system of forward-backward ODEs

$$\begin{cases} \mathrm{d}\boldsymbol{z}^*_{t,\boldsymbol{x}}(s) = \nabla_{\boldsymbol{p}}\mathcal{H}\big(s, \boldsymbol{z}_{t,\boldsymbol{x}}(s), \boldsymbol{p}^*_{t,\boldsymbol{x}}(s), \boldsymbol{u}^*_{t,\boldsymbol{x}}(s)\big)\mathrm{d}s, \\ \mathrm{d}\boldsymbol{p}^*_{t,\boldsymbol{x}}(s) = -\nabla_{\boldsymbol{z}}\mathcal{H}\big(s, \boldsymbol{z}^*_{t,\boldsymbol{x}}(s), \boldsymbol{p}_{t,\boldsymbol{x}}(s), \boldsymbol{u}^*_{t,\boldsymbol{x}}(s)\big)\mathrm{d}s, \\ \boldsymbol{z}^*_{t,\boldsymbol{x}}(t) = \boldsymbol{x}. \\ \boldsymbol{p}^*_{t,\boldsymbol{x}}(T) = -\nabla_{\boldsymbol{z}}\Phi(T, \boldsymbol{z}^*_{t,\boldsymbol{x}}(T)). \end{cases} \tag{2.28}$$

for $t \le s \le T$. eq. (2.28) is a set of necessary first order optimality conditions for

eq. (2.27), more details and assumptions can be found in [14, 112].

The difference in PMP led us to the first key difference between a SOC problem and its deterministic counterpart. For deterministic OC problems, the PMP serves as a *local solution*, that is given some fixed $(t, \boldsymbol{x})$ pair, we can obtain the optimal state and adjoint pair $(\boldsymbol{z}^*, \boldsymbol{p}^*)$ by discretizing and integrating eq. (2.28). Optimal control $\boldsymbol{u}^*$ can also be derived consequently. We refer to the solution obtained this way as *local* in the sense that the solution is tied to the initial input of the system, changing the initial pair $(t, \boldsymbol{x})$ or moving off the optimal state path $\boldsymbol{z}^*(s)$ will both render the optimal control obsolete.

Such local methods are generally not available for SOC problems with few exceptions such as [61]. Due to the existence of noise, even for fixed initial state $(t, \boldsymbol{x})$, the optimal state path $\boldsymbol{z}_{t,\boldsymbol{x}}(s)$ is not fixed. Therefore a direct discretization method is not sufficient for evaluating the optimal control, a function approximator for $\boldsymbol{u}^*$ is often necessary even for relatively simple problems. This difference poses a major challenge for solving SOC problems.

Under similar assumptions to theorem 2.4.2, for details see [38, Theorem I.6.2] we can aslo write the HJB equation as

$$\begin{cases} -\partial_s \Phi(s, \boldsymbol{z}) + H\big(s, \boldsymbol{x}, -\nabla_{\boldsymbol{z}} \Phi(s, \boldsymbol{z})\big) = -\partial_s \Phi(s, \boldsymbol{z}) - \nabla_{\boldsymbol{z}} \Phi(s, \boldsymbol{z}) \cdot f(s, \boldsymbol{z}, \boldsymbol{u}^*) \\ \qquad\qquad\qquad\qquad\qquad\qquad - L(s, \boldsymbol{z}, \boldsymbol{u}^*) = 0, \ \forall\, (s, \boldsymbol{z}) \in [t, T) \times \mathbb{R}^d, \\ \Phi(T, \boldsymbol{z}) = G(\boldsymbol{z}(T)). \end{cases} \tag{2.29}$$

Similar to SOC problems, HJB equations for deterministic OC problems also suffer from CoD. In addition to that, the difference in HJB equations eq. (2.22) and eq. (2.29) also affects the regularity of the solutions. To be more specific eq. (2.29) involves a fully nonlinear first order PDE which can have *viscous solutions*, while

eq. (2.22) is a quasilinear parabolic PDE, Assuming the noise is regular enough, the second order term effectively regularizes the problem and eq. (2.22) will admit smooth solution. One should expect the solution to eq. (2.22) to converge to the solution of eq. (2.29) as $\sigma \to 0$. The technique is the method of *vanishing viscosity*. For more details and precise results, we refer to [36, Chapter 10]. This difference is crucial to some of our work. While it is generally a challenging task to approximate a non-smooth function using neural networks, when dealing with SOC problems and its related HJB equation, as long as the diffusion term satisfies the smoothness assumption, we do not have such concern.

To summarize, in this chapter we provide the necessary mathematical background for the thesis. We first introduced the concept of machine learning and neural networks, this forms the basis to understanding the thesis. We then discussed PINNs and FEM, two very different methods for solving generic PDE problems. We rely on FEM as a benchmark solution method for many of the problems in this thesis, while for PINNs we will provide a deeper look in the next chapter. Lastly we provided background for optimal control problems and HJB equations, we will detail our proposed method for solving such problems in chapter 4.

Chapter 3

A Spline Based Alternative Model for PINNs

General Partial Differential Equations (PDEs) and how to solve them is one of the main focuses of the math community with applications arising from physics, chemistry and engineering, etc. With analytic solutions absent in many instances, finding suitable numerical solvers that can solve different PDEs with good accuracy and efficiency remains the task for many applications in the field.

Deep neural networks gained their attraction in the field of PDEs through popular work such as the Physics Informed Neural Networks (PINNs) [94] and Deep Ritz Method [34]. The introduction of neural networks to approximate solution functions is one of the key features that set the methods apart from traditional numerical PDE methods. Given the universal approximation property one can approximate arbitrary functions using neural networks without the need for any discretization of the space-time domain. Another advantage of using function approximators is that it can reduce or remove some of the stability concerns one may have when selecting time discretization schemes with traditional methods. Reformulating the PDE problem as an optimization problems also reduces the implementation difficulty, especially for

non-experts in the area.

That said neural network based methods for PDE solving are not without their challenges, first of all, though neural networks have proven to be effective on many high dimensional problems, it is unclear if they pose any advantages over other function approximators such as polynomials for low dimensional problems. The complex and nonlinear structure of neural networks almost always yields non-convex learning problems, which also makes solving them much harder. The Monte Carlo integration scheme that PINNs and other methods rely on is also inferior in accuracy to methods such as the midpoint rule or the Simpson's rule. In our work, we aim to take a deeper look at some of the mentioned challenges and hope to bring some insights into PINNs and other alternative optimization based methods for PDE solving.

3.1 A Deeper Look into PINNs

In this section, we provide a brief review of PINNs, including some state-of-the-art results in the field, in particular some recent work that tries to introduce classical numerical PDE methods into PINNs. Through studying these related works on PINNs we aim to gain a better understanding of the methods, specifically the strength of PINNs as well as some of the issues one may encounter while using the method.

Neural network based methods for solving PDEs have gained much attention over the past few years, with some more popular results such as the Deep Ritz method [34], the Deep Galerkin method[100] and the Physics Informed Neural Networks (PINNs) [94, 92, 93] all proposed after 2017. PINNs and related methods often share similar ideals in principle, where given some PDE, a neural network is used to model the solution of the differential equation with space and time variables as inputs, solving the PDE then translates to minimizing some energy functional of the residual of the PDE, as well as its initial and boundary conditions. PINNs as a method aims to

address and has the potential to overcome some of the challenges of traditional numerical PDE methods such as the finite element method (FEM). For instance, since using neural networks for function approximating does not rely on grid discretization in either space or time, PINNs is another option to alleviate the curse of dimensionality and can be directly applied to problems with dimensions greater than 3. This is generally not true for classical methods such as FEM or FDM, where special discretization in space will usually be needed for each problem. The grid-free property of PINNs also allows for easier handling of problems with irregular domains, notice for classical methods discretization of domains with irregular shape is one of the main challenges for performance, this is generally not the case for PINNs.

Another advantage of PINNs is that evaluating a trained model can be easy and cheap. Though training a neural network can be a hard task given the non-convex nature of the learning problem, once a model is trained, evaluating new data points in the domain can be both simple in practice and also efficient.

Much work in testing and applying PINNs has been done regarding different types of PDEs and received success to a certain degree, see [70, 101, 51, 28].

3.1.1 State of the Art in PINNs

In this section we provide a brief overview of some of the state-of-the-art models and results for PINNs, in particular, we will look at some work that aims to combine classical numerical PDE methods into PINNs.

The development of PINNs has branched into several directions including changes to sampling, problem formulation as well as models for function approximation. For sampling training points to feed into the neural network, [70] showed that sampling clusters of points in smaller regions instead of random sampling over the entire domain can improve model accuracy in parts of the domain where training can be hard, however, this often requires deeper knowledge of the problem for one to correctly

identify the sampling needs. In the absence of a reference solution, this strategy can be much harder to implement. In [109] it is shown that alternative sampling methods such as Latin hypercube sampling [67], Halton sequence [108] or Hammersley sequence can sometimes produce better results compared to the common uniform sampling methods. These results are often problem dependent though.

For proposed changes in models, Jagtap et al. in [49] introduced conservative PINNs (cPINNs), in the work they aim to tackle problems with complex geometries through a decomposition of the spatial domain and train multiple different neural networks to solve the problem, one of each subdomain. The idea of domain decomposition was later generalized in the work of extended PINNs (XPINNs), see [29], where a more general space-time domain decomposition was discussed, special care on the boundaries was also discussed such that each neural network trained for the problem is consistent with its neighbors. XPINNs is now commonly used for problems with large spatial and time domains. Follow-up and similar work also include [47] and [52]. Finite Basis Physics-Informed Neural Networks (FBPINNs) [74] borrowed ideas from the finite element methods and aim to reduce the spectral bias in PINNs [90], we find it to be in many ways similar to the above-mentioned XPINNs where a decomposition of the domain is used with the obtained solution being a combination of multiple different neural networks.

Other results in PINNs also include competitive PINNs [113] which adopt the idea from generative adversarial networks, aside from approximating the solution using a neural network, another neural network is added to measure the performance of the solution, the learning problem is also consequently reformulated as a saddle point optimization problem. Bayesian PINNs (BPINNs) [111] utilizes a Bayesian neural network and mainly targets problems with noisy data. These methods all showed some success in certain test problems, but are limited in applicability due to high computation cost and model limitations.

In particular, in recent years there have been attempts to introduce techniques and ideas from traditional numerical PDE methods to PINNs models. we hereby discuss a few of them. Kharazmi et al. proposed variational PINNs (VPINNs) in [51]. The main contribution of VPINNs lies in its modified version of the loss functional, where instead of minimizing the direct residual of the ODE, VPINNs form the loss functional based on the variational form of the given PDE and use Legendre polynomials for test functions. The work claims that the variational loss functional is more effective for solving certain PDEs with non-smoothness. The idea of Spline-PINN is proposed in [104], where a combination of Hermite Splines and convolutional neural networks is used to model the solution in space and time. Training of the model remains the same as regular PINNs. The method showed some success in problems such as the Navier-Stokes equation. We find this to be an interesting idea, however since the method still uses neural networks for constructing their function approximator, it faces similar challenges to other neural network based approaches. A more recent idea to combine the use of neural networks and FEM comes from [8]. Here a neural network is used for learning the solution, however, instead of calculating the loss directly, an interpolation of the neural network over some FEM space is first introduced, before the loss functional is evaluated accordingly over the finite element space with parameter gradient passing back to update the neural network.

Unlike methods such as finite element or finite difference, due to the use of neural networks, convergence theories for PINNs are generally sparse. In [99] Shin et al. showed convergence results of PINNs with respect to the number of training points. They showed that for linear elliptic and parabolic PDEs, strong convergence can be achieved in C^0 given i.i.d. sampled training data. In [72] the authors developed upper bounds on the generalization error of PINNs, the results however are limited to PDEs that satisfy certain stability requirements. Similar error estimates are also discussed in [97], here the authors looked at incompressible Navier-Stokes equations

and provided an upper bound on the total error of approximation. However, this is only true for specific neural network architectures with two hidden layers and uses tanh as activation functions. Such constraints on neural networks are not uncommon for error estimates of PINNs.

In short, despite gaining much popularity in recent years, convergence results and error analysis are still lacking for PINNs, especially for a more general class of neural network models. Even for many commonly used test problems such as in [28], little theory can be relied on.

3.2 Pros and Cons of PINNs

In this section, we want to provide our understanding of PINNs, namely the pros and cons of the method through both a literature review and our own experience experimenting with different examples. For some of the advantages and disadvantages of PINNs mentioned in previous sections such as its ability to tackle the curse of dimensionality, we shall not repeat them here.

We find one of the strengths of PINNs lies in its low entry bar for non-PDE experts and the fact that it can be set up and implemented rather easily. Unlike traditional numerical PDE methods whose successful implementation requires not only a deep understanding of the PDE itself but also means to solve the occurring linear system. With PINNs, minimal knowledge of the problem as well as neural network training is sufficient to set up a solver. This property has inspired many works and experiments beyond the numerical PDE community such as in [65, 68]. Implementation-wise, PINNs models usually rely on open-source libraries such as Pytorch, Tensorflow, or JAX [83, 1, 18], which all have large communities with many available resources. This results in a much lower threshold for troubleshooting when compared against packages for traditional numerical PDE methods such as FEniCS [5, 66] or NGSolve

[39].

Another accompanying strength of PINNs to many users is the stability of its solution. For classic numerical PDE methods such as the FEM and FDM, carefully selected step sizes for spacial and time discretization are often needed to avoid blow-up in the solution, such as meeting the CFL condition [27] for certain hyperbolic PDEs. Despite the difficulties in training, when approximating the solution using a neural network this is generally not of concern, in fact exploitation of such property has been done in [106, 107]. The reason for this is similar to that of time-dependent finite element methods.

Despite its success in many problems, PINNs is not without any issues. In fact, many works have been published detailing the challenges and difficulties of applying PINNs to different applications. In [43] Grossmann et al. showed that PINNs cannot achieve competitive accuracy when compared against the finite element method through several examples and extensive experiments. [23] focuses primarily on problems arising in fluid dynamics and details their failure with PINNs where models failed to converge despite much effort dedicated to training. Similar results are also found in [24]. In the seminal work by Krishnapriyan et al. in [55] the authors aim to explore and understand the failures of PINNs to converge. They conclude that neural networks are expressive enough to approximate the solutions to most PDEs, the challenge mainly resides in solving the consequent optimization problem. The paper also proposes a few strategies to address the issue, which we will discuss in later sections.

Due to the lack of convergence results and relevant theory to fall back on, we summarize the main disadvantage of PINNs the difficulty to properly train a neural network model. As laid out in earlier sections, formulation of the loss functional, network architecture and size, choice of sampling, and optimization scheme can all affect the outcome of the learning problem. Finding the appropriate hyperparameters

and learning setup can be time-consuming and affect solution accuracy.

It is also worth mentioning that though PINNs can achieve acceptable accuracy for many applications, for problems that require high precision, PINNs has yet to demonstrate its ability to attain so, which is also a limitation to its applicability.

3.3 A Trainable Spline Model for PDEs in Low Dimensions

As described in earlier sections, the introduction of neural networks as function approximators brings versatility but also creates more challenging optimization problems. We want to point out that finite dimensional spaces such as $P_1(\Omega)$ in FEM have been applied to a much wider range of PDEs with dimension $d \leq 3$ and have proven approximation theory for functions under some regularity constraints. In our following work, we aim to answer the question: **For low dimensional PDEs** ($d \leq 3$)**, is using a neural network necessary for PINNs or similar deep learning algorithms?**

Inspired by relevant work in the FEM [81, 115] and work such as [104], we propose the following cubic B-spline model for approximating functions in low dimensions. Consider some function $\Phi(x) \in C^0(\Omega)$ (note that we use $\Phi(x)$ to denote arbitrary continuous functions only in this section, which is different from the value function $\Phi(s, \boldsymbol{z})$ defined for HJB equations) with some scalar input x. we aim to construct the approximation

$$\Phi(x) \approx \Phi^{\text{spline}}(x) = \sum_{j=1}^{m} \theta_j b_j(x), \tag{3.1}$$

here m is the total number of basis functions, we use θ_j and b_j to represent the coefficients and the corresponding basis functions. Here each basis function $b_j(x)$ is a translated version of some $b(x)$, which is sometimes referred to as a "mother" spline.

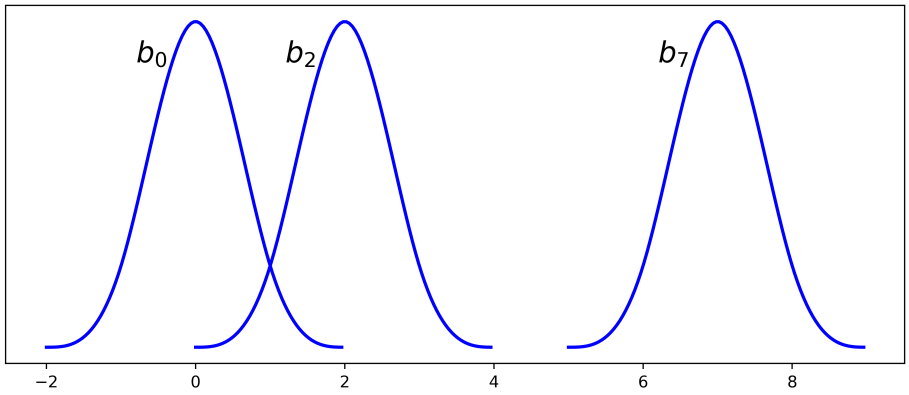


Figure 3.1: Mother spline b_0 and basis functions b_2 and b_7 in 1D.

we define $b(x)$ as

$$b(x) = \begin{cases} (x+2)^3, & -2 \leq x < -1, \\ -x^3 - 2(x+1)^3 + 6(x+1), & -1 \leq x < 0, \\ x^3 + 2(x-1)^3 - 6(x-1), & 0 \leq x < 1, \\ (2-x)^3, & 1 \leq x < 2, \\ 0, & \text{else.} \end{cases} \tag{3.2}$$

The translation is defined by $b_j(x) = b(x-j)$, allowing $\Phi^{\text{spline}}(x)$ to be defined for any given x, another way to denote the spline function is $\Phi_\theta(x)$. We visualize the basis functions in 1D in fig. 3.1.

3.3.1 Spline Interpolation

One of the key differences between polynomials and neural networks is their ability to interpolate existing data. We here briefly describe the interpolation of our proposed B-spline model, for more details of polynomial interpolation one can refer to [3, 85, 103].

We consider the following interpolation problem for function $\Phi(x)$

$$\min \int_{\Omega} (\Phi''(x))^2 dx \quad \text{subject to} \quad \Phi(x_j) = \text{dataT}(j),\ j = 1, 2, \ldots, m, \tag{3.3}$$

where dataT represents the given data points, we aim to minimize the bending energy while fitting the given data. Let $\theta = [\theta_1; \theta_2; \ldots; \theta_m]^\top$, expanding eq. (3.1) at the cell centers $x_j = j$ gives the interpolation condition

$$\text{dataT}(j) = \Phi^{\text{spline}}(x_j) = \sum_{k=1}^{m} \theta_k b_k(j) = [b_1(j), \ldots, b_m(j)]\theta, \quad j = 1, 2, \ldots, m.$$

By concatenating each data points j we can obtain the following linear system

$$\text{dataT} = B_m \theta$$

where

$$B_m = [b^k(x_j)] = \begin{bmatrix} 4 & 1 & & 0 \\ 1 & \ddots & \ddots & \\ & \ddots & \ddots & 1 \\ 0 & & 1 & 4 \end{bmatrix},$$

here matrix $B_m \in \mathbb{R}^{m \times m}$. We refer to [73] for a more detailed introduction to the spline model as well as implementation in Matlab.

Evaluating the spline model $\Phi^{\text{spline}}(x)$ is also efficient compared to a neural network, notice that the "mother" spline $b(x) = 0$ for $x \notin (-2, 2)$, therefore for any given x at most four basis functions are nonzero, reducing the computational cost largely.

3.3.2 Spline Model for Higher Dimensions

For higher dimensional function approximation, namely $d = 2$ and $d = 3$, we resort to a Kronecker product approach. In short we will replace eq. (3.1) with the following

$$\Phi(\boldsymbol{x}) \approx \Phi^{\text{spline}}(\boldsymbol{x}) = \sum_{j_d=1}^{m_d} \cdots \sum_{j_1=1}^{m_1} \theta_{j_1,\ldots,j_d} b_{j_1}(x_1) \ldots b_{j_d}(x_d), \tag{3.4}$$

where $d = 2$ or $d = 3$.

Similar to the 1D case, interpolation results can be derived, we omit them here and point to [73] for those who are interested.

Implementation wise it is most common and efficient to treat the polynomial approximation as a tensor product, where a d-dimensional tensor θ is used to represent the coefficients $\theta_{j_1,\ldots,j_d}$, for each input $\boldsymbol{x} = [x_1, \ldots, x_d]$, spline value $\Phi_j^{\text{spline}}(x_j)$ is calculated before combined together as a tensor product. We will discuss our implementation in more detail in the following sections.

For time dependent problems we can derive a similar structure by treating time as additional dimension in the Kronecker product, which will have the form

$$\Phi(\boldsymbol{x}, t) \approx \Phi^{\text{spline}}(\boldsymbol{x}, t) = \sum_{j_d=1}^{m_d} \cdots \sum_{j_1=1}^{m_1} \theta_{j_1,\ldots,j_d,j_t} b_{j_1}(x_1) \ldots b_{j_d}(x_d) \tilde{b}_{j_t}(t), \tag{3.5}$$

here $d = 1, 2, 3$. we use j_t to index the coefficients corresponding the to time input and $\tilde{b}_{j_t}(t)$ to denote the basis functions. Given related theory in FEM we here differentiate the basis function in time by using a linear function instead of a cubic spline function, namely we define

$$\tilde{b}(t) = \begin{cases} t + 1, & -1 \leq t \leq 0, \\ 1 - t, & 0 \leq t \leq 1, \\ 0, & \text{else.} \end{cases} \tag{3.6}$$

Each basis function $\tilde{b}_{j_t}$ can be translated from $\tilde{b}$ following $\tilde{b}_{j_t}(x) = \tilde{b}(x - j)$ similar to

$b(x)$. Such a definition allows us to treat time dependent variables in the same way as spatial variables, this is the same as PINNs, avoiding the need for time discretization as is often for classical methods such as FDM or FEM. Interpolation in time can also be done easily as a piecewise linear interpolation problem.

3.3.3 Derivatives and Laplacians

When approximating the solutions $\Phi(\cdot)$ using neural networks and minimizing the residuals of the given PDEs, evaluating the gradient $\nabla\Phi$ and Laplacian $\Delta\Phi$ with respect to the input x and t plays an essential role. With PINNs this is usually done through the use of automatic differentiation [11], this usually requires at least a full backward propagation of the entire neural network model. While calculating the gradient term can be done this way relatively easily, computationally it can be much more expensive when retrieving second order terms such as the Hessian or the Laplacian using automatic differentiation. One possible solution to alleviate some of the computation cost is to introduce packages like hessQuik [75].

For our proposed cubic spline model this can be done much easier. Given the Kronecker product of 1D basis functions definition in eq. (3.4), for dimension index q we have

$$\partial_q \Phi^{\text{spline}}(\boldsymbol{x}) = \sum_{j_d=1}^{m_d} \cdots \sum_{j_1=1}^{m_1} \theta_{j_1,\ldots,j_d} b_{j_1}(x_1) \ldots (b_{j_q}(x_q))' \ldots b_{j_d}(x_d), \tag{3.7}$$

b'_{j_1} can be derived through the "mother" spline in eq. (3.2) as

$$b'(x) = \begin{cases} 3(x+2)^2, & -2 \le x < -1, \\ -9x^2 - 12x, & -1 \le x < 0, \\ 9x^2 - 12x, & 0 \le x < 1, \\ -3(2-x)^2, & 1 \le x < 2, \\ 0, & \text{else}, \end{cases} \quad \text{and} \quad b''(x) = \begin{cases} 6x + 12, & -2 \le x < -1, \\ -18x - 12, & -1 \le x < 0, \\ 18x - 12, & 0 \le x < 1, \\ 12 - 6x, & 1 \le x < 2, \\ 0, & \text{else}. \end{cases}$$

Similarly, the Laplacian can also be calculated directly given any input x using the formula above without the need for automatic differentiation. It is also important to note that given any fixed input x and t, the spline interpolation, its gradient and Hessian are linear with respect to the weights $\{\theta_i\}$.

Implementation Our implementation of the B-spline model follows the above-described setup. Though defined for functions for any dimension, we restrict this in our code to problems with spatial dimension of a maximum of 3 with additional time variable if necessary. Our implementation also allows for parameterizations of functions with vector outputs, which can be seen as a concatenation of 2 or more scalar output functions.

In order to reduce the complexity of choosing the appropriate model, which is often of concern in PINNs applications, our model can be defined and set up automatically given only the domain information, as well as the number of basis functions in each dimension. Our model allows for simultaneous calculation of both the gradient and Laplacian with respect to the input along the forward propagation, reducing a large portion of the computation cost.

We implement our model in Pytorch which allows for easy substitution for neural networks in many PINNs related applications. We reference [73] and its Matlab code

for our spline structure, we have since made modifications to satisfy our need for PDE problems.

3.3.4 Sampling and Optimization

In this section, we briefly describe the sampling and optimization strategy we use in our experiments to accompany our selected spline function approximator.

Note that for PINNs and similar methods that rely on the the use of neural networks, it is most common to use Stochastic Approximation (SA) methods for training, which use mini-batch sampling of training points in the domain with optimization methods such as SGD or Adam. For our testing we opt for a Sample Average Approximation (SAA) [53] scheme for optimization. We base our sampling of collocation points $\{\boldsymbol{x}_f^i, t_f^i\}_{i=1}^{N_f}$ on a grid discretization of the state-time domain. Boundary points and initial points $\{\boldsymbol{x}_g^i, t_g^i\}_{i=1}^{N_g}$ and $\{\boldsymbol{x}_h^i\}_{i=1}^{N_h}$ can also be treated in the same way with grid discretization of their respective surfaces. We note that this is tractable given problems in lower dimensions. With a grid based sampling method, the problem can then be formulated as a deterministic optimization problem with respect to the samples $\{\boldsymbol{x}_f, t_f, \boldsymbol{x}_g, t_g, \boldsymbol{x}_h\}$, we can then solve the problem using methods with super-linear convergence rate with line search such as L-BFGS.

We want to emphasize the two main reasons that motivate our formulation of the training problem. First, an SA method is not suitable for optimizing our spline model, notice that given each input in space at most 4 coefficients in each dimension will be updated at each iteration, meaning only a small portion of all weights will receive updates when trained with a mini-batch sampling scheme. Secondly, by using a grid based sampling scheme, we can use numerical integration methods such as the midpoint rule, which ensures higher accuracy compared to the Monte Carlo integration method. We can also reduce the number of hyperparameters by formulating the problem as a deterministic optimization problem. In our testing we explore whether

we can maintain the accuracy of PINNs.

3.3.5 Outline of Our Method for Testing

In this section, we provide an outline of the method we aim to test for an arbitrary PDE. We aim to maintain the optimization structure of PINNs by formulating the solver as an optimization problem of the PDE's residual while replacing the neural network with a polynomial model. We address some of the ensuing issues by modifying the classical PINNs algorithm. Consider a time dependent PDE defined in general form from section 2.2

$$\mathcal{A}u(\boldsymbol{x}, t) = f(\boldsymbol{x}, t), \qquad \boldsymbol{x} \in \Omega, t \in [0, T]. \tag{3.8}$$

with boundary and initial condition

$$\begin{cases} \mathcal{B}u(\boldsymbol{x}, t) &= g(\boldsymbol{x}, t), \qquad \boldsymbol{x} \in \partial\Omega, t \in [0, T], \\ u(\boldsymbol{x}, 0) &= h(\boldsymbol{x}), \qquad \boldsymbol{x} \in \Omega. \end{cases} \tag{3.9}$$

A learning problem can be set up with the following steps.

Create Spline Model and Sample points We construct a spline model given the problem domain $\Omega \times [0, T]$ with a chosen number of nodes for both $\boldsymbol{x}$ and t. We follow the sampling scheme described in earlier sections by performing a grid discretization of the domain, which gives $\{\boldsymbol{x}_f, t_f, \boldsymbol{x}_g, t_g, \boldsymbol{x}_h\}$. We notice through various experiments that increasing the number of splines generally improves model accuracy. This is not necessarily true for PINNs as adding to the width and depth of a neural network also increases the complexity of the optimization and may not always lead to a better approximation of the solution, as demonstrated in[43].

Initialization As pointed out in [55] finding appropriate initialization plays an important role in PINNs. While with neural networks it is generally hard to locate a good initialization of the model, more can be tested when using a polynomial based model. In particular for time dependent problems with given initial conditions, we propose initializing the spline nodes using interpolation of the initial condition. We repeat this for every spline node in time, which can be cheaply done, effective having $u_\theta(\boldsymbol{x}, t) = h(\boldsymbol{x})$ for $\forall t \in [0, T]$. By doing so we directly satisfy the initial condition without any update, we find that this also provides a better initial guess for the following optimization problem compared to random initialization.

Loss Function and Optimization PINNs and similar methods minimize the residual of some PDE as an energy function which has the form

$$\min_\theta \int_\Omega \int_0^T (\mathcal{A}u_\theta(\boldsymbol{x}, t) - f(\boldsymbol{x}, t))^2 d\boldsymbol{x}dt \tag{3.10}$$

where u_θ denotes the function approximation of the solution u. While PINNs relies on a Monte Carlo approximation of the integral in eq. (3.10), given our choice of sample points we instead minimize the following loss function

$$\begin{aligned} Loss(\theta) =& \alpha_1 \left(\sum_{i=1}^{N_f} h_x h_t (\mathcal{A}u_\theta(\boldsymbol{x}_f^i, t_f^i) - f(\boldsymbol{x}_f^i, t_f^i))^2 \right) \\ &+ \alpha_2 \left(\sum_{j=1}^{N_g} h_t (\mathcal{B}u_\theta(\boldsymbol{x}_g^j, t_g^j) - g(\boldsymbol{x}_g^j, t_g^j))^2 \right) + \alpha_3 \left(\sum_{k=1}^{N_h} h_x (u_\theta(\boldsymbol{x}_h^k, 0) - h(\boldsymbol{x}_h^k))^2 \right). \end{aligned} \tag{3.11}$$

which is essentially the numerical integration of eq. (3.10) using the midpoint integration rule given the grid discretization of sample points. Here $\{\alpha_1, \alpha_2, \alpha_3\}$ are hyperparameters, in practice we usually set $\alpha_1 = 1$. Since our sampling relies on a grid discretization of the space-time domain, we use h_x and h_t to denote the step sizes

used.

Notice that given fixed sample points the loss function *eq.* (3.11) is deterministic with respect to the trainable weights θ, we therefore can solve the minimization problem using methods such as L-BFGS.

With the spline architecture defined in eq. (3.4), we can easily verify the following results for linear PDEs.

Corollary 3.3.1. *Given the general form of PDEs defined in eq.* (2.7) *with boundary and initial condition following eq.* (2.8) *over domain* $\Omega \times [0,T]$ *with* $\mathcal{A}$ *and* $\mathcal{B}$ *being linear operators. Let* $u_\theta(x,t)$ *denote the spline function approximator from eq.* (3.5) *defined over the same domain with* θ *being the nodal weights. For any selection of collocation points* $\{\boldsymbol{x}_f^i, t_f^i\}_{i=1}^{N_f} \in \Omega \times [0,T]$, $\{\boldsymbol{x}_g^j, t_g^j\}_{j=1}^{N_g} \in \partial\Omega \times [0,T]$ *and* $\{\boldsymbol{x}_h^k\}_{k=1}^{N_h} \in \Omega$. *The minimization problem*

$$\begin{aligned} Loss(\theta) = &\sum_{i=1}^{N_f} (\mathcal{A}u_\theta(\boldsymbol{x}_f^i, t_f^i) - f(\boldsymbol{x}_f^i, t_f^i))^2 + \sum_{j=1}^{N_g} (\mathcal{B}u_\theta(\boldsymbol{x}_g^j, t_g^j) - g(\boldsymbol{x}_g^j, t_g^j))^2 \\ &+ \sum_{k=1}^{N_h} (u_\theta(\boldsymbol{x}_h^k, 0) - h(\boldsymbol{x}_h^k))^2 \end{aligned}$$

is convex with respect to θ.

Notice here given that corollary 3.3.1 holds for any choice of sampling schemes, this also applies to the grid based sampling and exact integration method used in eq. (3.11). Though some convergence results for PINNs have been established for certain PDEs such as the Poisson equation, we are not aware of similar results for a more general class of PDEs. Considering that linear PDEs consist of a wide range of different problems, we argue that our proposed spline function approximator and optimization scheme is advantageous over PINNs in accuracy, we will demonstrate this in the following sections with numerical examples.

3.4 Numerical Experiments for Different PDEs

In this section, we test our discussed method and compare the results against some PINNs solutions on a series of different benchmark PDE problems.

After reviewing many related works, we decide to primarily follow [43] for numerical experiments and comparisons. The authors provided detailed comparisons between trained PINNs solutions and finite element solutions on several different examples from 1D to 3D. Unlike many other works, in [43] the authors not only provide hyperparameters for the best results, but also a detailed training routine used to obtain the optimal model, including different network architectures and optimization schemes tested for each problem. Though their network models may not necessarily be state-of-the-art in terms of accuracy, we find this to be the most accurate depiction of a normal user's experience with PINNs where much effort and time are often needed to properly train a neural network model.

In total we borrow 4 different examples from [43], namely for elliptic equations we consider the Poisson equation in both two and three space dimensions. For parabolic equations, we consider a time dependent Allen-Cahn equation in one space dimension, and for hyperbolic equations, we test on a semilinear Schrödinger equation with one space dimension. For these examples we primarily follow the experimental setup from [43] while also testing our proposed scheme.

For problems introduced with explicit analytic solutions we directly compare the PINNs results and our spline model results to the analytic solution for error measurement, no additional numerical solution will be needed. However, for problems that do not have such solutions, we rely on FEM solutions for comparison. We generally rely on section 2.3 for setting up the FEM solver with additional details added for each problem. In all instances, we use very fine discretization for both space and time in order to ensure our FEM solution is as close as possible to the ground truth solution. Implementation wise we use FEniCS [66] for all FEM solutions.

For obtaining the necessary PINNs models, we follow [43] closely. The PINNs models tested here share the structure and loss function as the vanilla approach as described in Raissi et al. [92, 93]. Unlike the vanilla approach, a two-step optimization scheme is used where the Adam optimizer is used in the first phase of training with random sampling at each epoch, followed by L-BFGS on a fixed set of collocation points to refine the results. This training setup usually produces models with the highest accuracy from our testing. We utilize the optimal learning rate selected in [43] for each example. For sampling of collocation points the Latin Hypercube sampling [102] is used as experiments have suggested that such sampling scheme often produces better results. All derivatives in the loss function calculation as well as parameter updates are computed using automatic differentiation. For harder problems such as the Allen-Cahn equation and the Schrödinger equation, an additional optimization step is added at the beginning of the training, where the network is trained to fit only the initial condition. In [43] the authors claim such practice can improve accuracy on problems tested.

Finding the optimal neural network is one of the biggest challenges for PINNs, for each example in [43] multiple neural networks are tested with different depths and widths. Different network architectures can lead to significant differences in error results, system memory cost, training time and model evaluation time.

To further demonstrate our method and for more extensive comparisons we will also include additional examples beyond those used in [43], Including a two dimensional Navier-Stokes equation examples from [24] and one dimensional Burger's equation example also used in [94]. we shall provide more details of each of these examples in the relevant sections.

For implementation details, [43] has the PINNs models coded using JAX [18], in our experiments we will use FEniCs for FEM solutions and Pytorch [83] for training our proposed spline models.

For error calculation we mainly use 2 metrics, first is the most common l_2 relative error used in most PINNs related papers, given function $u(\boldsymbol{x}, t)$ and its approximation $u_\theta(\boldsymbol{x}, t)$ with collocations points $\{\boldsymbol{x}_i, t_i\}_{i=1}^N$, by stacking the collocation points as vector inputs $\boldsymbol{x}, \boldsymbol{t}$, the l_2 relative error of the approximation can be measured using the following formula

$$l_2 \text{ relative eror} = \frac{\|u(\boldsymbol{x}, \boldsymbol{t}) - u_\theta(\boldsymbol{x}, \boldsymbol{t})\|^2}{\|u(\boldsymbol{x}, \boldsymbol{t})\|^2}.$$

Note here the collocation points can come from either random sampling or a grid based discretization. We also provide average absolute error which takes the form $\frac{1}{N}\sum_{i=1}^N |u(\boldsymbol{x}_i, t_i) - u_\theta(\boldsymbol{x}_i, t_i)|$, which we also find to be useful at times.

3.4.1 2D Poisson Equation

In this section we consider the two-dimensional Poisson equations which is defined as:

$$\begin{aligned} \Delta u(x, y) =& 2(x^4(3y-2) + x^3(4-6y) + x^2(6y^3 - 12y^2 + 9y - 2) \\ & - 6x(y-1)^2 y + (y-1)^2 y), \qquad (x, y) \in (0, 1)^2. \end{aligned} \tag{3.12}$$

The problem is defined with mixed boundary condition

$$\begin{cases} \partial_{\vec{n}} u(0, y) = 0, & y \in [0, 1] \\ \partial_{\vec{n}} u(1, y) = 0, & y \in [0, 1] \\ u(x, 0) = 0, & x \in [0, 1] \\ \partial_{\vec{n}} u(x, 1) = 0, & x \in [0, 1] \end{cases} \tag{3.13}$$

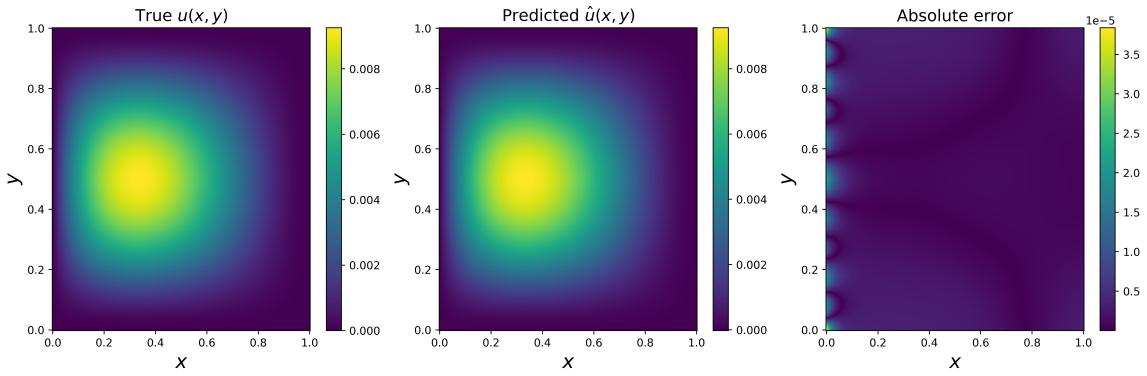


Figure 3.2: Approximation results of the 2D Poisson equation. Left: visualization of the analytic solution to the problem. Middle: Approximated solution using our spline based model. Right: absolute error between the learned solution and the true solution over the domain. Notice here the spline function approximator achieves consistently low error across the entire domain.

Such formulation of the problem allows for an analytic solution which reads

$$u(x, y) = x^2(x-1)^2 y(y-1)^2$$

for all (x, y) in the given domain. We visualize the true solution to the problem in fig. 3.2. Notice here the solution u has values between 0 and 0.01 over the entire domain, accurately approximating the solution can be challenging given the precision needed. We refer to [43] for a carefully selected learning rate and training routine for the problem.

Spline Approximation For setting up the learning problem for the spline model, we follow the steps described in earlier chapters, here we define the loss function as

$$\begin{aligned} Loss(\theta) = &\sum_{i=1}^{N_f} h^2(\Delta u_\theta(x_f^i, y_f^i) - f(x_f^i, y_f^i))^2 + 100 \sum_{j=1}^{N_g} \Big(h(\partial_{\vec{n}} u_\theta(0, y_g^j))^2 \\ &+ h(\partial_{\vec{n}} u_\theta(1, y_g^j))^2 + h(\partial_{\vec{n}} u_\theta(x_g^j, 1))^2 + h(u_\theta(x_g^j, 0))^2 \Big), \end{aligned} \tag{3.14}$$

here h is the step size for the discretization we use in sampling and f is the right hand side function, we also set the weight for the boundary condition to 100. For

sampling we use a 150×150 grid to obtain all collocation points, we construct the spline model with 32 nodes for each dimension. Since we use L-BFGS with line search for optimizing the loss, no additional tuning for the learning rate is needed.

We display the training results in fig. 3.2 as well as a comparison to the analytic solution. Our method showed high accuracy with the absolute approximation error being below the 10^{-5} range, in fact when evaluated on a finer mesh, the average absolute error of the spline model over the domain is 1.6×10^{-6} across all collocation points.

PINNs For PINNs results we follow closely to the work of [43] where they minimize the l_2 residual of the PDE as well as the boundary condition. A total of 2250 points are sampled at each epoch with $N_f = 2000$ and $N_g = 250$. A multi-layer perceptron (MLP) is used as the forward function approximator, see section 4.1 for more details, with tanh as activation functions. The following architectures are tested: [20,1], [60,1], [20,20,1], [60,60,1], [20,20,20,1], [60,60,60,1], [20,20,20,20,1], [60,60,60,60,1], [20,20,20,20,20,1], [60,60,60,60,60,1], and [120,120,120,120,120,1]. We aim to find the optimal solution that can be achieved by PINNs. For training Adam optimizer is used for 20K epochs with a learning rate of 1×10^{-3}, followed by L-BFGS with line search until no update for network weights can be made.

We use l_2 relative error as our primary metric and display the training results in table 3.1 alongside the spline approximation results. It is clear that the spline solution produces errors that are at least a magnitude lower than any of the trained PINNs models. We also note that for PINNs selecting the optimal NN architecture can be challenging, simply adding to the depth of the neural network or increasing the size of each layer does not always improve accuracy. In fact, the most complex architecture tested in this experiment is [120,120,120,120,120,1] which showed relatively low accuracy compared to small neural networks. This result is consistent with the ones

Methods	architecture	l_2 relative error (lower is better)
Spline model	32×32	0.00065
PINNs	$[20, 1]$	0.11
	$[60, 1]$	0.058
	$[20, 20, 1]$	0.025
	$[60, 60, 1]$	0.023
	$[20 \times 3, 1]$	0.013
	$[60 \times 3, 1]$	0.004
	$[20 \times 4, 1]$	0.012
	$[60 \times 4, 1]$	0.006
	$[20 \times 5, 1]$	0.010
	$[60 \times 5, 1]$	0.003
	$[120 \times 5, 1]$	0.011

Table 3.1: Error results of both spline models and PINNs when compared against the analytic solution, we display the l_2 relative error using different architectures. Here the spline approximation results are at least a magnitude better than any PINNs model.

from [55].

3.4.2 3D Poisson Equation

In this section we consider the Poisson equation in three space dimensions. The problem is defined on the unit cube and reads as

$$\Delta u(x, y, z) = -3\pi^2 \sin(\pi x) \sin(\pi y) \sin(\pi z) \qquad (x, y, z) \in (0, 1)^3, \tag{3.15}$$

with homogeneous Dirichlet boundary condition

$$u(x, y, z) = 0 \qquad (x, y, z) \in \partial[0, 1]^3.$$

Analytic solution to the problem can be written as

$$u_{\text{true}}(x, y, z) = \sin(\pi x) \sin(\pi y) \sin(\pi z),$$

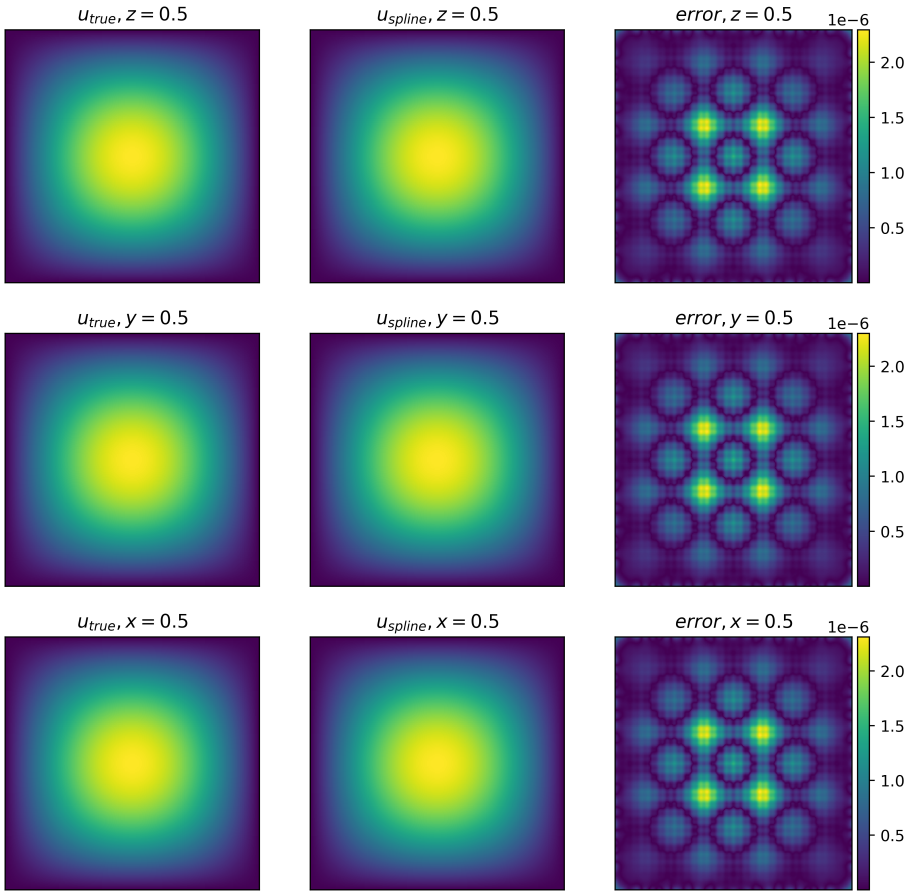


Figure 3.3: Approximation results of the 3D Poisson equation using the spline model proposed in section 3.3. Here we visualize the solution and compare at three slices in space, namely at $x = 0.5$, $y = 0.5$ and $z = 0.5$. We show the analytic solution in the first column, the approximation results in the second column and the absolute measured in the last column, notice here the errors are consistently in the 10^{-6} range, showing high approximation accuracy.

for any pair (x, y, z) in the given domain. We visualize the solution to the problem in fig. 3.3 over a few slices in space.

Spline Approximation We set up the spline model and corresponding learning problem using the loss function

$$\begin{aligned} Loss(\theta) = &\sum_{i=1}^{N_f} h^3(\Delta u_\theta(x_f^i, y_f^i, z_f^i) + 3\pi^2 \sin(\pi x_f^i)\sin(\pi y_f^i)\sin(\pi z_f^i))^2 \\ &+ 100\sum_{j=1}^{N_g}\Big(h^2(u_\theta(0, y_g^j, z_g^j))^2 + h^2(u_\theta(1, y_g^j, z_g^j))^2 + h^2(u_\theta(x_g^j, 0, z_g^j))^2 \\ &+ h^2(u_\theta(x_g^j, 1, z_g^j))^2 + h^2(u_\theta(x_g^j, y_g^j, 0))^2 + h^2(u_\theta(x_g^j, y_g^j, 1))^2\Big), \end{aligned} \tag{3.16}$$

with h being the step size used for spacial discretization. Here in this experiment, we use a $100 \times 100 \times 100$ grid for generating all the collocation points. We consider a rather simple $32 \times 32 \times 32$ spline architecture for approximating the solution as we find it is sufficient for this problem. We use L-BFGS for updating parameters until convergence.

We present some of the final training results in fig. 3.3 where we also compare the solution against the analytic solution for $u(x, y, z)$. We choose to display three slices of the domain at $x = 0.5$, $y = 0.5$ and $z = 0.5$ respectively, notice the absolute error on each slice is at 10^{-6} level or lower. We further calculate the average absolute error of the approximated solution over 5000 randomly sampled points within the unit cube, the average error is 4.36×10^{-7}, indicating that the learned solution has high accuracy over the given domain.

PINNs For PINNs training we use the l_2 residual of the PDE and the boundary condition to construct the loss function as presented in [43]. We select $N_f = 1000$ and $N_g = 100$ in training each epoch. We use MLP as our primary network structure for testing and consider the following different architectures: [20,20,1], [60,60,1], [20,20,20,1], [60,60,60,1], [20,20,20,20,1], [60,60,60,60,1], [20,20,20,20,20,1], and [60,60,60,60,60,1]. We run Adam optimizer with learning rate 1×10^{-3} for 20K iteration before moving to L-BFGS for fine tuning. All gradient and Hessian required

Methods	architecture	l_2 relative error (lower is better)
Spline model	$32 \times 32 \times 32$	1.86×10^{-6}
PINNs	$[20, 20, 1]$	0.002
	$[60, 60, 1]$	0.0008
	$[20 \times 3, 1]$	0.011
	$[60 \times 3, 1]$	0.0005
	$[20 \times 4, 1]$	0.0015
	$[60 \times 4, 1]$	0.0009
	$[20 \times 5, 1]$	0.0021
	$[60 \times 5, 1]$	0.0009

Table 3.2: Error results of both spline models and PINNs when compared against the analytic solution of the 3D Poisson equation, we display the l_2 relative error using different architectures. Here the spline approximation results are over 2 magnitudes better than all PINNs models tested.

for evaluating the loss or updating the parameters are calculated using automatic differentiation.

We show the PINNs results as well as comparisons against the spline models in table 3.2. For this example, PINNs can achieve relatively high accuracy with l_2 relative errors between 10^{-2} and 10^{-3} for all the architectures tested. However, the spline approximation showed a much lower error of 1.86×10^{-6}, which is over 2 magnitudes lower than any PINNs results.

To briefly summarize the 2 presented Poisson equation examples, notice regardless of the right hand side both problems are linear PDEs. For fixed collocations points used for training, the resulting loss function is, therefore, convex with respect to the spline coefficients, as discussed in corollary 3.3.1, guaranteeing a reachable global minimizer for the problem. For this reason, our spline approximation can achieve much lower error than any PINNs architecture tested. Similar results should be expected for other linear PDEs such as the heat equation or the Helmholtz equation. In short, we believe for linear PDEs in low dimensions, our proposed spline model and optimization scheme is preferable compared to PINNs in accuracy.

In the following sections, we will primarily focus on nonlinear PDEs which will

result in non-convex optimization problems in general, finding a good approximation of the solution will therefore become much more difficult, we will investigate the performance of our proposed method one different examples and compare against PINNs in accuracy.

3.4.3 1D Schrödinger Equation

For examples on hyperbolic equations we here experiment on a semilinear Schrödinger equation example often used as a benchmark problem in works such as [43, 92, 93] and remains one of the more challenging problems to solve with neural networks. We consider the following problem

$$i\frac{\partial h(x,t)}{\partial t} = -0.5\Delta h(x,t) - |h(x,t)|^2 h(x,t), \qquad x \in [-5,5], t \in [0,\frac{\pi}{2}], \tag{3.17}$$

with boundary and initial consition

$$\begin{cases} h(x,0) = 2\text{sech}(x), & x \in [-5,5], \\ h(5,t) = h(-5,t), & t \in [0,\frac{\pi}{2}], \\ \dfrac{\partial h(5,t)}{\partial x} = \dfrac{\partial h(-5,t)}{\partial x}, & t \in [0,\frac{\pi}{2}]. \end{cases}$$

Here $h(x,t)$ is a complex valued function and can be written as $h(x,t) = u_R(x,t) + i \cdot u_I(x,t)$. It is in practice most common to solve for u_I and u_R individually before combining them to get $h(x,t)$.

FEM Solution It is important to note that analytic solutions for eq. (3.17) does not exist and therefore we resort to a FEM solution for measuring errors. We here provide a brief outline for the FEM solution, we refer to [43] for more details.

Here we consider a semi-explicit Euler scheme for time discretization, the weak

form for both the real and imaginary part of the PDE can be formulated as

$$\int_{-5}^{5}(u_I^{t+1}(x)-u_I^t(x))v_R(x)dx-\frac{1}{2}\int_{-5}^{5}\langle\,\nabla u_R^{t+1}(x),\nabla v_R(x)\rangle dx-|h^t(x)|^2\int_{-5}^{5}u_R^{t+1}(x)v_R(x)dx=0,$$
$$\int_{-5}^{5}(u_R^{t+1}(x)-u_R^t(x))v_I(x)dx+\frac{1}{2}\int_{-5}^{5}\langle\,\nabla u_I^{t+1}(x),\nabla v_I(x)\rangle dx+|h^t(x)|^2\int_{-5}^{5}u_I^{t+1}(x)v_I(x)dx=0$$

for test functions v_R and v_I, here we assume Dirichlet boundary for simplicity. To obtain an accurate solution to the problem we select step size $dt = 1 \times 0^{-3}$ for time discretization. For spatial discretization we use a total of 2048 cells, We choose P_1 finite element and solve the problem using FEniCs. We display the visualized FEM solution of $|h(x,t)|$ in fig. 3.4.

Spline Approximation In order to solve the problem using our spline setup, we consider the following loss function

$$\begin{aligned}
Loss(\theta) = &\frac{1}{N_f}\sum_{i=1}^{N_f}h_x h_t((\frac{\partial u_I^\theta(x_f^i,t_f^i)}{\partial t}-\frac{1}{2}\Delta u_R^\theta(x_f^i,t_f^i)-|h^\theta(x_f^i,t_f^i)|^2u_R^\theta(x_f^i,t_f^i))^2\\
&+(\frac{\partial u_R^\theta(x_f^i,t_f^i)}{\partial t}+\frac{1}{2}\Delta u_I^\theta(x_f^i,t_f^i)+|h^\theta(x_f^i,t_f^i)|^2u_I^\theta(x_f^i,t_f^i))^2)\\
&+\frac{1}{N_g}\sum_{j=1}^{N_g}h_t((u_R^\theta(-5,t_g^j)-u_R^\theta(5,t_g^j))^2+(u_I^\theta(-5,t_g^j)-u_I^\theta(5,t_g^j))^2)\\
&+\frac{1}{N_g}\sum_{j=1}^{N_g}h_t((\frac{\partial u_R^\theta(-5,t_g^j)}{\partial x}-\frac{\partial u_R^\theta(5,t_g^j)}{\partial x})^2+(\frac{\partial u_I^\theta(-5,t_g^j)}{\partial x}-\frac{\partial u_I^\theta(5,t_g^j)}{\partial x})^2)\\
&+\frac{1}{N_h}\sum_{k=1}^{N_h}h_x((u_R^\theta(x_h^k,0)-2\mathrm{sech}(x_h^k))^2+(u_I^\theta(x_h^k,0))^2),
\end{aligned}\tag{3.18}$$

by treating the real and imaginary part of the solution separately using different function approximators. Here h_x and h_t represent discretizations used for space and time sampling respectively. In our experiments we use a 300×250 grid for generating all the collocation points. we test three different spline structures, namely 128×96,

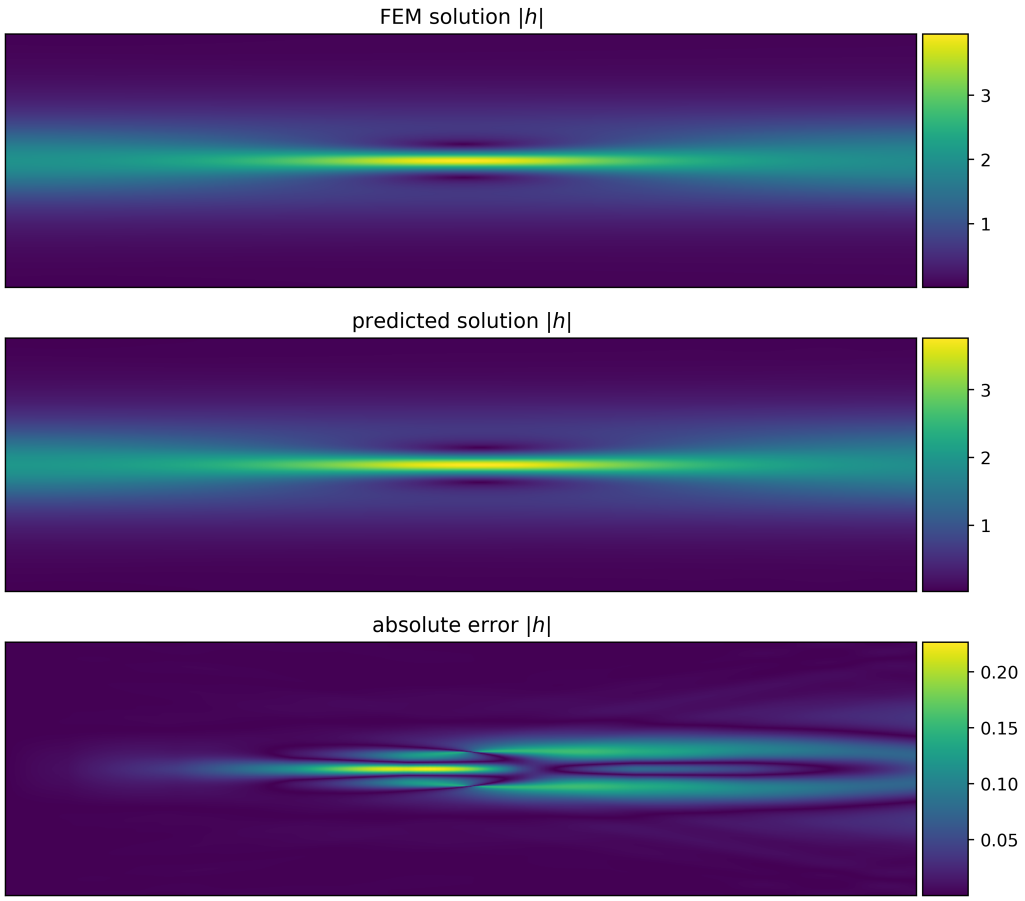


Figure 3.4: Visualization of the solved solution to eq. (3.17), Top: FEM solution of $|h(x,t)|$ using a fine grid and time discretization. Middle: the learned solution using our spline based model. Bottom: Absolute error between the FEM solution and the spline model.

192×128 and 256×192, in all test cases we use L-BFGS for optimization until convergence.

We show the approximation results of our spline model in fig. 3.4 here the most accurate solution is obtained using 256×192 nodes. In fig. 3.4 we display the comparison between the FEM solution and the spline solution of $|h(x,t)|$ over the domain, as well as the absolute error between them evaluated over a 500×500 grid. Here the average absolute error for the approximation is 0.016. We also notice that the errors are low initially and increase as time increases. This shows that our initialization of the spline architecture using the given initial condition is useful in improving the

Methods	architecture	l_2 relative error (lower is better)
Spline Model	128×96	0.058
	192×128	0.055
	256×192	0.038
PINNs	$[20 \times 3, 2]$	0.057
	$[20 \times 6, 2]$	0.092
	$[100 \times 4, 2]$	0.075
	$[100 \times 6, 2]$	0.104

Table 3.3: Error results of both spline models and PINNs for the Schrödinger equation, here the errors are measured with respect to $|h(x,t)|$. We compare the solutions to the FEM results obtained using a fine mesh. The l_2 relative error is used as the metric, notice that here the spline solution can achieve slightly better accuracy compared to the best PINNs results.

accuracy of the model.

PINNs The PINNs experiment follows the instruction given in [43], using a loss function that minimizes the l_2 residual of the PDE similar to eq. (3.18), two neural networks $u_I(x,t;\theta_1)$ and $u_R(x,t;\theta_2)$ are used to learn the real and imaginary part of the solution. We use $N_f = 20000$ collocation points sampled within the domain, $N_g = 50$ for boundary points and $N_h = 50$ for initial conditions. Similar to previous examples we use MLP architectures for both approximating $u_I(x,t;\theta_1)$ and $u_R(x,t;\theta_2)$. 8 different architectures are tested in [43] with networks of 20 and 100 neurons per layer and different depth. We train each network a total of 50K iterations using Adam optimizer with a learning rate of 1×10^{-4} before finalizing the model with L-BFGS.

We present some of the PINNs results as well as comparisons against the FEM solution and the spline model results in table 3.3. Here one thing to note is that trained PINNs models with different architectures have little difference in accuracy, we therefore only include ones with the highest and lowest error to be displayed. As can be observed the best spline results we obtained have a l_2 relative error of 0.038, which is lower than all the PINNs models trained, though the difference is not as big

as some of the other examples we presented. We also note that based on our test results, increasing the number of nodes of the spline model can in most cases improve accuracy. Here in our testing, the simple 128×96 model can still produce results that are similar to the best PINNs model trained.

In general, we find both methods struggling to achieve high accuracy for the problem, we attribute this to the complex loss function defined for the problem. Additional consideration for optimization may be needed for both our spline architecture and PINNs if higher accuracy is desired.

3.4.4 Allen-Cahn equation

In this section we discuss a one dimensional Allen-Cahn equation which is used in [43] and is another challenging problems for PINNs to solve accurately. We consider the following PDE

$$\frac{\partial u(x,t)}{\partial t} = \epsilon \Delta u - \frac{2}{\epsilon} u((x,t)(1-u((x,t))(1-2u((x,t)), \quad x \in \Omega = [0,1],\ t \in [0,T] \tag{3.19}$$

with periodic boundary condition and initial condition given as

$$\begin{cases} u(0,t) = u(1,t), \quad t \in [0,T] \\ u(x,0) = \frac{1}{2}(\frac{1}{2}\sin(x2\pi) + \frac{1}{2}\sin(x16\pi)) + \frac{1}{2}, \quad x \in \Omega, \end{cases}$$

here we select $T = 0.05$ and $\epsilon = 0.01$. Here the choice of a smaller ϵ will yield a solution that is closer to a piecewise constant function, which increases the difficulty of solving the problem, especially for PINNs and similar methods, we refer to [43, 4] for more details on the Allen-Cahn equation.

FEM Solution Similar to the Schrödinger equation for the Allen-Cahn equation an analytic solution is not readily available to us, hence we again resort to a finite

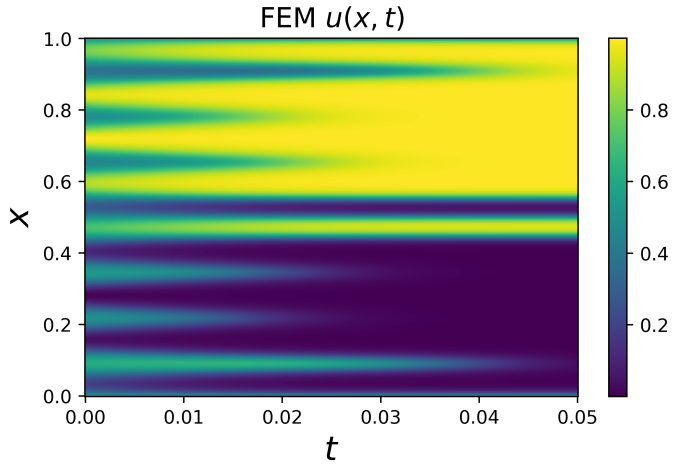


Figure 3.5: Solutions to the Allen-Cahn equation obtained through the finite element method, here we use a fine grid for both space and time discretization to ensure that the numerical solution is as accurate as possible.

element solution for accurately solving the problem and compare against solutions obtained from the spline model or PINNs. For time discretization we consider a semi-explicit Euler scheme. Assuming Dirichlet boundary condition we can write the weak form of the problem as

$$\begin{aligned}\int_0^1 (u_{t+1}(x) - u_t(x))v(x)dx + 0.01 \cdot \int_0^1 \langle \nabla u_{t+1}(x), \nabla v(x) \rangle dx \\ + \frac{2}{0.01} \int_0^1 u_t(x)(1 - u_t(x))(1 - 2u_t(x))v(x)dx = 0,\end{aligned} \tag{3.20}$$

for all test functions $v(x) \in H_0^1([0, 1])$. In order to solve the problem with high accuracy, we select the times step size $dt = 1 \times 10^{-4}$. For the finite element mesh in space, we use a total of 2048 cells over the given domain. We use P_1 finite element for the solution space and solve the nonlinear variational problem using a Newton solver. Implementation is done using FEniCs. We display the solved FEM solution in fig. 3.5.

Spline Approximation For using a spline model to approximate the solution to eq. (3.19), we consider the following loss function

$$\begin{aligned} Loss(\theta) = \frac{1}{N_f}\sum_{i=1}^{N_f} h_x h_t (\frac{\partial u_\theta(x_f^i, t_f^i)}{\partial t} - 0.01 \cdot \Delta u_\theta(x_f^i, t_f^i) \\ + \frac{2}{0.01} u_\theta(x_f^i, t_f^i)(1 - u_\theta(x_f^i, t_f^i))(1 - 2u_\theta(x_f^i, t_f^i)))^2 \\ + \frac{1}{N_g}\sum_{j=1}^{N_g} h_t (u_\theta(0, t_g^j) - u_\theta(1, t_g^j))^2 \\ + \frac{1}{N_h}\sum_{k=1}^{N_h} h_x (u_\theta(x_h^k, 0) - \frac{1}{2} - \frac{1}{2}(\frac{1}{2}\sin(2\pi x_h^k) + \frac{1}{2}\sin(16\pi x_h^k)))^2. \end{aligned} \tag{3.21}$$

We use h_x and h_t to denote the step size used for generating all the collocation points, in practice we also raise the weight on the boundary and initial condition in the loss to 1000 for a faster and more accurate approximation of them. We use a 300×300 grid for sampling all points used for training and test models using different numbers of spline nodes. We find the best result is obtained when using a 250×200 spline architecture. For training we simply use L-BFGS until convergence.

We show the spline approximation results in fig. 3.6 together with both the absolute and relative when compared against our obtained FEM solution. Notice for this problem we get relatively low errors over the entire domain with the average absolute error being 0.00037 when compared against the FEM solution over a 500×500 grid of the same domain.

PINNs In [43] the authors find applying PINNs for the Allen-Cahn equation defined in eq. (3.19) can be particularly challenging. We here provide a brief overview of the optimal PINNs setup being used, we refer to [43] for more details. The loss function is defined similarly to eq. (3.21) where the l_2 residual of the PDE is minimized, here additional weight of 1000 is placed over the initial condition for more accurate results. The neural network is trained with $N_f = 20000$ sample points $(x_f^i, t_f^i) \in$

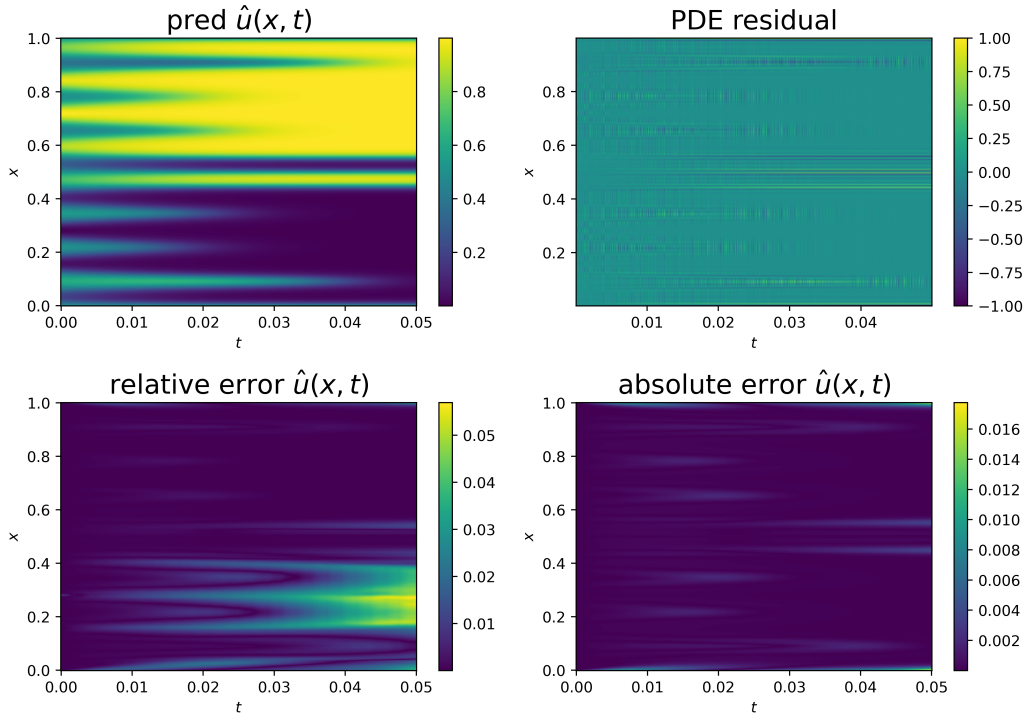


Figure 3.6: Spline approximation results for the Allen-Cahn equation, we display the predicted solution as well as the PDE residual after training in the top row. In the bottom are plots of absolute and relative error for the problem, notice both remains relative low over the domain.

$[0,1] \times [0, 0.05]$, For boundary and initial condition $N_g = 250$ and $N_h = 500$ are chosen. All sample points are collected using Latin Hypercube sampling at each training iteration.

Our network type of choice is MLP and we test over 10 different architectures using 20,100 or 500 neurons per layer with different numbers of layers. We omit a few test cases from [43] in particular deep networks with 500 weights for each layer, we find the time and memory cost to train such networks get increasingly high while the performance actually decreases. Unlike some of the previous examples, we employ a three-step training scheme to solve the minimization problem. 7K steps of Adam with learning rate 1×10^{-4} is applied to only the initial loss, followed by 50K iterations on the full loss using Adam, and lastly L-BFGS is applied for fine tuning the results. Our testing results coincide with that of [43] in that an accurate approximation of

Methods	architecture	l_2 relative error (lower is better)
Spline Model	256×192	0.0014
PINNs	$[20 \times 3, 1]$	0.59
	$[20 \times 5, 1]$	0.099
	$[20 \times 7, 1]$	0.57
	$[100 \times 3, 1]$	0.56
	$[100 \times 5, 1]$	0.025
	$[100 \times 6, 1]$	0.053
	$[500 \times 3, 1]$	0.079
	$[500 \times 4, 1]$	0.045

Table 3.4: Error results of both spline models and PINNs for the Allen-Cahn equation, here the errors are measured with respect to the FEM results obtained using a fine mesh. The l_2 relative error is used as the metric, For PINNs we present the best results for each architecture type and omit some of the results that are less accurate.

the initial and boundary condition is necessary for the PINNs model to converge to the true solution.

We present error results for both PINNs and the spline approximation in table 3.4. Here the errors are measured using the FEM solution. We notice that the spline model can achieve a l_2 relative error of 0.0014, which is a magnitude lower than all the trained PINNs models. The lowest error is achieved in PINNs using a 5 layer network with 100 neurons at each layer. Notice here that increasing the depth of the neural network does not monotonically increase accuracy, we attribute this to the added complexity of the optimization problem.

We notice that the computation and time cost get increasingly high when training with larger networks, Here when running on a NVIDIA Tesla T4 GPU, training a network with architecture [500,500,500,500,1] takes around 8 hours while for the architecture [100,100,100,100,100,1] which gives the highest accuracy, training time is around an hour. Training the proposed spline model in this case takes less than 15 minutes with the same hardware. We don't find the time comparison to be particularly meaningful given the differences between these 2 methods, however consider the hyperparameter tuning requirement for PINNs, for harder problems time and

computation overhead remain a challenge for PINNs.

3.4.5 2D Taylor-Green Vortex Problem

In this section we consider the 2D incompressible Navier-Stokes equations, which takes the general form

$$\begin{cases} \nabla \cdot \boldsymbol{u} = 0 \\ \frac{\partial \boldsymbol{u}}{\partial t} + (\boldsymbol{u} \cdot \nabla)\boldsymbol{u} = -\frac{1}{\rho}\nabla p + \nu \nabla^2 \boldsymbol{u} \end{cases} \tag{3.22}$$

with some boundary and initial condition. Here $\boldsymbol{u} = [u, v]^\top$, p, ν and ρ denote the velocity vector, pressure, kinematic viscosity, and density, respectively. Assuming the fluid properties ρ and ν are given, our goal is to recover the velocity and pressure $[u(x, y, t), v(x, y, t), p(x, y, t)]^\top$ given some domain $(x, y) \in \Omega$ and $t \in [0, T]$.

We refer to [26, 17, 89] for more details on the Navier-Stokes equations, many numerical methods for solving the Navier-Stokes equation have been developed such as [25, 41]. PINNs and similar methods have also been applied and tested on such problems in works such as those from[23, 104], we here place our focus mainly on the work [24], we find the authors detailing of their experience applying PINNs to solve the Navier-Stokes equation very inspiring, as they demonstrated some of the strengths and challenges regarding PINNs.

In this section we consider a 2D Taylor-Green vortex (TGV) problem at Reynolds number $Re = 100$ which is also used in [24], here the Reynolds number measures the ratio between inertial and viscous forces. With periodic boundary conditions, the

problem yields a closed form analytic solution that reads

$$\begin{cases} u(x,y,t) = \cos(x)\sin(y)\exp(-2\nu t) \\ v(x,y,t) = -\sin(x)\cos(y)\exp(-2\nu t) \\ p(x,y,t) = \dfrac{\rho}{4}(\cos(2x)+\cos(2y))\exp(-4\nu t) \end{cases} \tag{3.23}$$

with $\nu = 0.01$ and $\rho = 1$ assume to be given. We define the space and time domain to be $x, y \in [-\pi, \pi]$ and $t \in [0, 100]$. Here since we mostly focus on approximation accuracy of learning based algorithms, we omit most of the details and some properties of the Navier-Stokes equations, we refer to [24] for a more thorough review of the problem.

Spline Approximation Given that the solution to the TGV problem has a vector output, we therefore use different spline approximator for each of $u(x,y,t)$, $v(x,y,t)$ and $p(x,y,t)$ respectively. For loss function, we consider the following residual terms

$$\begin{cases} L_1 = \nabla \cdot \boldsymbol{u}_\theta \\ L_2 = \partial_t u_\theta + \boldsymbol{u}_\theta \cdot \nabla u_\theta + \partial_x p_\theta - 0.01\nabla^2 u_\theta \\ L_3 = \partial_t v_\theta + \boldsymbol{u}_\theta \cdot \nabla v_\theta + \partial_x p_\theta - 0.01\nabla^2 v_\theta \\ L_4 = u_\theta - u_0 \\ L_5 = v_\theta - v_0 \\ L_6 = p_\theta - p_0 \\ L_7 = u_\theta - u_D \\ L_8 = v_\theta - v_D \end{cases} \tag{3.24}$$

Here L_1, L_2, L_3 represents the PDE residual within the given domain, L_4, L_5, L_6 are initial conditions with $t = 0$, and L_7, L_8 are boundary conditions, we assume Dirichlet

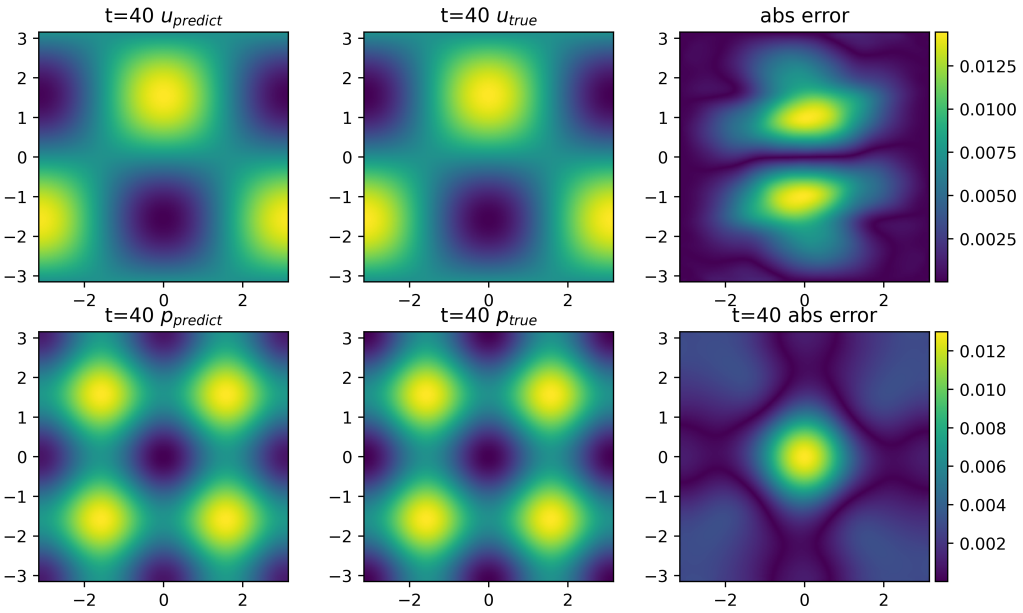


Figure 3.7: Spline approximation results for the 2D Taylor-Green Vortex, Similar to [24] we choose to visualize both u_θ and p_θ at $t = 40$ over the spatial domain. we compare the results to the analytic solution for error measurement. For both u and p the absolute error at $t = 40$ is in the 10^{-3} range.

boundary for simplicity in our testing. we can write the loss function as

$$Loss(\theta) = \int_\Omega \int_0^T (L_1^2 + L_2^2 + L_3^2) \mathrm{d}\boldsymbol{x}\mathrm{d}t + \int_\Omega (L_4^2 + L_5^2 + L_6^2) \mathrm{d}\boldsymbol{x} + \int_0^T \int_{\partial\Omega} (L_7^2 + L_8^2) \mathrm{d}\boldsymbol{x}\mathrm{d}t. \tag{3.25}$$

To evaluate the loss function we opt for a grid based sampling scheme and use the midpoint rule for integration, we choose a $200 \times 200 \times 250$ grid for generating all the collocation points. regarding the spline model, we tested several different structures and noticed insignificant differences between them, in the end we selected a model with $40 \times 40 \times 200$ nodes in each dimension, we found this model sufficient to approximate the solution, further increasing in number of splines increases computation cost and does not yield better solution in our testing. For optimization, we rely on L-BFGS until no updates to the parameters can be made.

We display some training results in fig. 3.7, similar to [24] we visualize both u_θ and p_θ over a slice of the domain at $t = 40$ using a 500×500 grid. We compare the

Methods	architecture	l_2 error for u at $t = 40$	l_2 error for u at $t = 0$
Spline model	$40 \times 40 \times 200$	0.022	1.84×10^{-5}
PINNs	$[20 \times 3, 3]$	0.031	0.009
	$[20 \times 4, 3]$	0.069	0.012
	$[20 \times 5, 3]$	0.054	0.008
	$[100 \times 2, 3]$	0.037	0.005
	$[100 \times 4, 3]$	0.016	0.003
	results from [24]	≈ 0.02	$\approx 2 \times 10^{-4}$

Table 3.5: Error results of both spline models and PINNs when compared against the analytic solution for the 2D Taylor-Green Vortex, here similar to [24] we show the l_2 relative error of $u(x, y, t)$ measured against the analytic solution at different times. We also include results from [24] for a fine tuned PINNs model.

learned solution to the analytic solution in eq. (3.23), and we get the average absolute error for u and p at 0.0035 and 0.0023 respectively. Here velocity v has almost the same results as u. We find the errors to be relatively low given the large domain of the problem.

PINNs Without any available PINNs model and codes to work with, we opt for setting up our own training routine. We follow a similar idea to [43] for testing PINNs. For loss function we consider a similar l_2 residual penalization to eq. (3.25). We use MLP neural networks with tanh as activation functions. We test the following architectures: [20,20,20,3], [20,20,20,20,3], [20,20,20,20,20,3], [100,100,3], [100,100,100,3] and [100,100,100,100,3]. We use a two-step training scheme for optimizing our neural networks of choice, in the first step we apply 30K iterations using Adam optimizer with the learning rate set to 1×10^{-3}, here we select $N_f = 2000$, $N_g = 250$ and $N_h = 250$. We follow this up with L-BFGS for refining the network weights, at this step we increase the batch size to $N_f = 5000$,$N_g = 500$ and $N_h = 500$.

In [24] a fine tuned PINNs solution is presented with state-of-the-art accuracy for the TGV problem. In their work, the proposed network uses the MLP structure with 3 hidden layers and 128 neurons per layer. For sampling at each iteration 8192 collocation points are used for both the PDE loss and the boundary/initial conditions.

The optimization runs for a total of 400K iterations using Pytorch's Adam optimizer with $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\epsilon = 10^{-8}$. The learning rate of choice for the problem starts at 1×10^{-3} and gradually decreases over time, we refer to [24] for more details on the learning rate scheduler. We include the results in table 3.5 for comparison against some of our results.

We present error results for both PINNs and the spline approximation in table 3.5. We compare the trained solution to the analytic solution presented in eq. (3.23). Here similar to [24] we focus on error comparison at 2 different time stamps, namely $t = 0$ and $t = 40$, where l_2 errors are calculated with respect to $u_\theta(x, y, t)$. Here notice that most of our trained PINNs models are worse in terms of accuracy when compared to the state-of-the-art results, particularly at the initial time. On the other hand, the spline model shows very similar accuracy when measured at $t = 40$, and lower error at $t = 0$ when compared against the state-of-the-art PINNs result. We conclude that the spline model can achieve on-par accuracy to PINNs even on complex problems with relatively large domains like eq. (3.22), whilst avoiding some of the difficulty for fine tuning a neural network.

We recognize that Navier-Stokes equations represent a large class of PEDs in the field of computational fluid dynamics, further explorations of other types of problems are necessary for a deeper understanding of the proposed methods. However, we find it more reasonable to leave it for future work and study.

3.4.6 Additional Numerical Schemes

Given the polynomial structure of the spline model we use, unlike with neural networks, additional numerical schemes can be tested and may result in improvement in accuracy or convergence speed. We here provide a brief description of some of the simple experiments we did, though this is not the primary focus of our work we hope to inspire more discussion regarding the topic.

A Projection Method for Boundary Conditions

For learning based algorithms such as PINNs, accurately approximating the boundary condition is in many cases crucial to the overall performance of the methods. With PINNs this is usually done by re-weighting the loss function or modifying the training process, as shown in the example eq. (3.19). It usually takes trial and error to find the optimal hyperparameters.

Given the interpolation property of our proposed spline model we propose a projection method that can accurately match the boundary condition of a given PDE while maintaining the key features of the learning algorithm. We consider a simplified version of the general PDE in eq. (2.7) with boundary condition $\mathcal{B}u(\boldsymbol{x}) = g(\boldsymbol{x})$ for $\boldsymbol{x} \in \partial\Omega$, the loss function can be written in integration form similar to eq. (2.9)

$$Loss(\theta) = \int_{\Omega} (\mathcal{A}u_\theta(\boldsymbol{x}) - f(\boldsymbol{x}))^2 \mathrm{d}\boldsymbol{x} + \int_{\partial\Omega} (\mathcal{B}u_\theta(\boldsymbol{x}) - g(\boldsymbol{x}))^2 \mathrm{d}\boldsymbol{x}$$

where θ denotes the trainable parameters. We find it possible to translate the problem into a constraint optimization problem with the boundary condition being the equality constraints. Here notice that the boundary condition by definition is linear in terms of θ, given $\{\boldsymbol{x}_g^j\}_{j=1}^{N_g}$, the interpolation problem $\mathcal{B}u_\theta(\boldsymbol{x}_g^j) = g(\boldsymbol{x}_g^j)$ for $j = 1, 2, \ldots, N_g$ can be solved directly or for a least square solution.

Assuming θ_0 solves the minimization problem given collocation points $\{\boldsymbol{x}_g^j\}_{j=1}^{N_g}$

$$\min_{\theta} \sum_{j=1}^{N_g} (Bu_\theta(\boldsymbol{x}_g^j) - g(\boldsymbol{x}_g^j))^2.$$

Here since the operator $\mathcal{B}$ is linear, we can write it in matrix form as B. The solution to the original problem has the form $\theta = \theta_0 + N\hat{\theta}$ with N representing the null space of B. We can then reduce the original problem into a single objective optimization

Methods	average absolute error on boundaries	l_2 error
Default, $w = 1$	0.0017	0.45
Default, $w = 10$	8.68×10^{-5}	0.024
Default, $w = 100$	5.02×10^{-6}	0.00065
With projection	9.84×10^{-8}	3.29×10^{-5}

Table 3.6: Comparison between the original learning problem and when using projection for treating boundary conditions, here we use w to denote the added weight for the boundary loss, with higher w we emphasize optimizing the boundary loss to the PDE loss, therefore one should expect lower error on the boundary. In this example using the projection method allows us to achieve the highest accuracy on both the boundaries and interior.

problem which reads

$$Loss(\hat{\theta}) = \sum_{i=1}^{N_f} (\mathcal{A}u_{\theta_0 + N\hat{\theta}}(\boldsymbol{x}_f^i) - f(\boldsymbol{x}_f^i))^2.$$

We note that in practice this is equivalent to fixing the nodal values on the boundary and optimizing only the spline nodes in the interior of the domain. Time dependent problems can be treated similarly.

We use the 2D Poisson equation from eq. (3.12) to verify our idea. Notice here the analytic solution has 0 values on all boundaries. We compare training the spline model in its default setting and using the projection method, some results are presented in table 3.6. Her when treating the boundary as part of the loss function we test different weights for the boundary loss, we use the same spline model as presented in table 3.1. We notice that in table 3.6 by increasing the boundary weight the model achieves higher accuracy on the boundary, more importantly, the l_2 error over the entire domain also decreases as we approximate the boundary condition better. By using the projection method we can achieve the lowest errors both on the boundary and within the domain, showing the potential of the idea.

Domain Decomposition

While domain decomposition based methods have seen much development for traditional numerical PDE methods, it has also gained traction in recent years in the field of PINNs, with ideas like [29] being widely tested. We here primarily focus on the work [55], where a sequence-to-sequence learning scheme is proposed to address possible convergence issues originating from some complex problems. In this section we conduct some preliminary testing of the same idea on our spline architecture. We aim to investigate whether such an idea can result in improvement in solution accuracy.

We consider the following 1D Burger's equation as our main example

$$\begin{cases} \dfrac{\partial u}{\partial t} + u\dfrac{\partial u}{\partial x} - (0.01/\pi)\dfrac{\partial^2 u}{\partial x^2} = 0, \qquad x \in [-1,1], t \in [0,1] \\ u(x,0) = -\sin(\pi x) \\ u(1,t) = u(-1,t) = 0. \end{cases} \tag{3.26}$$

Notice here the same problem is a commonly used benchmark problem also seen in [92, 94, 9]. It's worth pointing out that the problem does not yield an analytic solution, however accurate numerical solutions to the problem is readily available through the *Chebfun* library, we refer to [10, 32] for a detailed guide to the numerical solutions.

Given a defined spline model, we consider the following loss function

$$\begin{aligned} Loss(\theta_n) = &\sum_{i=1}^{N_f} h_x h_t (\partial_t u_{\theta_n}(x_f^i, t_i) + u_{\theta_n}(x_f^i, t_i)\partial_x u_{\theta_n}(x_f^i, t_i) - (0.01/\pi)\Delta u_{\theta_n}(x_f^i, t_i))^2 \\ &+ \sum_{j=1}^{N_g} h_t (u_{\theta_n}(1, t_g^j) - u_{\theta_n}(-1, t_g^j))^2 + \sum_{k=1}^{N_h} h_x (u_{\theta_n}(x_h^k, n\Delta t) - u(x_h^k, n\Delta t))^2. \end{aligned}$$

Note here instead of solving the problem on the entire domain, we discretize the domain in time into N sub-intervals. We then solve the problem sequentially on each

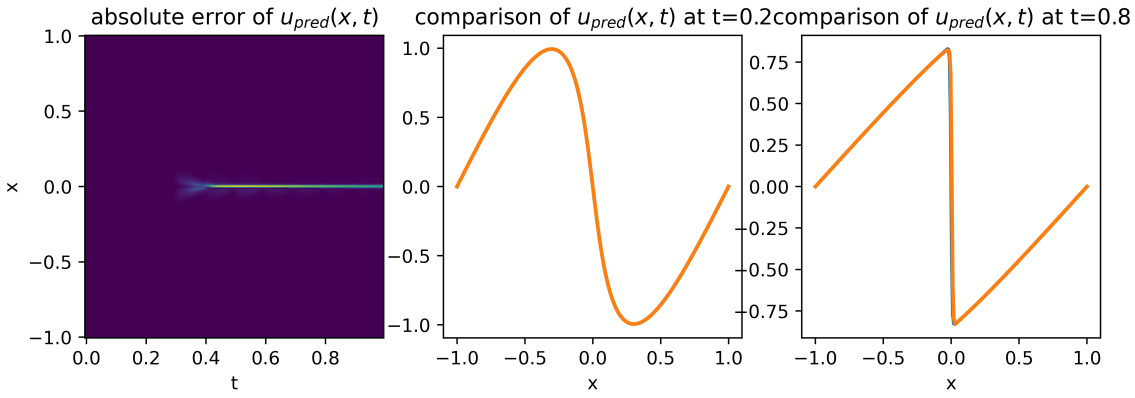


Figure 3.8: Approximation results for the 1D Burgers equation, here we visualize at 2 different time $t = 0.2$ and $t = 0.8$, in both cases we can capture the function well, notice for the problem we get highest error near the discontinuity.

sub-domain $[-1, 1] \times [n\Delta t, (n+1)\Delta t]$ for $n = 0, 1, \ldots, N-1$. Notice here for each sub-interval, we use the approximation results from the previous one as the initial condition, thus finalizing the loss function. It's also worth pointing out that for the spline model, nodal values do not overlap, meaning we can write $\theta = [\theta_0, \theta_1, \ldots, \theta_{N-1}]$ and only θ_n is optimized over each sub-domain. We train the function approximator at each sub-domain following the same scheme as previously proposed, note here we also correspondingly reduce the sampling for each sub-domain, the total number of spline nodes and sample points remain the same as training directly on the full domain.

For our experiment, we consider splitting the domain into 10 sub-domains of the same size, before solving on them one by one in a sequential order. We use a spline model with 192×100 nodes, we find using fewer nodes does not yield good results for the problem. We show some of the learning results in fig. 3.8. Here we specifically visualize the solution at $t = 0.2$ and $t = 0.8$, in both cases the solutions are captured relatively well, we also note that we get the highest error near the "shock". We believe further study into learning discontinuous functions or PDEs with viscous solutions may be necessary for our polynomial model.

We measure the l_2 relative errors of our spline model trained on the full domain

Methods	l_2 relative error
Spline model $[192, 100]$	0.061
Spline model $[256, 192]$	0.042
Spline model $[192, 100]$, seq2seq	0.049
PINNs, Raissi et al. [92]	0.045

Table 3.7: Comparison of L_2 relative error between different spline model and corresponding PINNs results. We notice that when using the same spline architecture, the sequence-to-sequence learning tested can improve model accuracy especially near the "shock", The overall accuracy of our model is also similar to that of PINNs.

and using the sequence-to-sequence learning technique and compare them in table 3.7. Here we notice that using the domain decomposition scheme does reduce model error similar to [55], in particular, we find that with domain decomposition we can capture the "shock" with much higher accuracy. Comparing our results to available PINNs results from [92] also showed similar accuracy. We want to point out that in a more recent work [9] the state-of-the-art PINNs results for the example have been further improved, however since our main objective for the experiment is not to contest against PINNs in accuracy, we will leave this for future work and discussion.

3.5 Summary

In our work we try to gain a better understanding of the pros and cons of PINNs and similar machine learning methods for solving PDEs, more importantly we aim to answer the following question, whether neural networks are necessary for the optimization algorithm to be effective in low dimensional problems and how does PINNs compare against other alternative methods. We investigate this by proposing an alternative spline based function approximator which serves as a direct substitute for neural networks in low dimensional problems, we also propose an optimization algorithm that relies on exact numerical integration instead of Monte Carlo integration. Our implementation uses existing machine learning libraries and requires little knowledge of classical numerical PDE methods.

We test our methods on multiple examples over different classes of PDEs and compare the obtained solutions to some of the most accurate PINNs models available. In all the tested examples we find that using a polynomial model we can achieve as good or much better accuracy compared to the PINNs solutions, while at the same time partially reducing the need for hyperparameter tuning. Convergence to global minima can also be shown for linear PDEs, which is not always true for PINNs.

The spline structure allows for additional numerical schemes that are not possible or easily implementable with PINNs, such as interpolation with existing data; using projection methods to satisfy boundary conditions; or domain decomposition based ideas. We also tested these ideas and received expected improvement in solution accuracy. We believe further exploration of similar ideas may be worthwhile.

To briefly summarize we find through extensive experimentation that for many low dimensional PDEs neural networks are not necessary to achieve accurate approximation using an optimization based algorithm, In fact, polynomial based models can be just as if not more effective while having much better reproducibility. We want to use this opportunity to open up more discussion on the topic, we believe a deeper look into similar polynomial models can yield more insights and lead to potential developments of PINNs as well.

Chapter 4

Deep Learning Approach for SOC problems and HJB Equations

In this Chapter we describe our proposed deep learning approach for solving high dimensional (stochastic) optimal control problems and the corresponding HJB equations. The theoretical foundation of our framework is given by the PMP, FBSDE, and Dynamic Programming as presented in the previous chapter. The key idea is to approximate the value function Φ by a neural network and compute the control using the feedback form. What distinguishes our framework from similar approaches such as [44, 91] is the use of the feedback form to guide the sampling during training. Thereby we seek to learn to explore the relevant part of the state space. This followed by combining both the original control objective and HJB equation into the learning problem allows us to tackle a given problem with better accuracy and efficiency. We demonstrate our methods through various examples and comparisons.

4.1 Neural Network Approximation

The first building block of our framework is to parameterize the value function using a neural network. Since finding an effective network architecture for any learning task

is both crucial and an open research topic, we treat this as a modular component. Our framework can be used with any scalar-valued neural network that takes inputs in $\mathbb{R}^{d+1}$ as long as it is twice differentiable with respect to its last d inputs; this is to allow computations of $\nabla\Phi$ and $\nabla^2\Phi$.

Among the networks we use in our numerical experiments is the multi-layer perceptron (MLP) model used in [91, 44]. As an alternative, which also satisfies the regularity needed, we propose the residual network also used for deterministic control in [78], relevant work in [77] and showed satisfying results. The network is given by

$$\Phi(\boldsymbol{y};\boldsymbol{\theta}) = \boldsymbol{w}^\top \mathcal{NN}(\boldsymbol{y};\boldsymbol{\theta}_{\mathcal{NN}}) + \frac{1}{2}\boldsymbol{y}^\top(\boldsymbol{A}^\top\boldsymbol{A})\boldsymbol{y} + \boldsymbol{b}^\top\boldsymbol{y} + c, \tag{4.1}$$

with trainable weights $\boldsymbol{\theta} = (\boldsymbol{w}, \boldsymbol{\theta}_{\mathcal{NN}}, \boldsymbol{A}, \boldsymbol{b}, c)$. Assuming the state space has dimension d, here the inputs $\boldsymbol{y} = (s, \boldsymbol{z}) \in \mathbb{R}^{d+1}$ correspond to time-space variables, $\mathcal{NN}(\boldsymbol{y};\boldsymbol{\theta}_{\mathcal{NN}})\colon \mathbb{R}^{d+1} \to \mathbb{R}^m$ is a neural network, and $\boldsymbol{\theta}$ contains the trainable weights: $\boldsymbol{w} \in \mathbb{R}^m$, $\boldsymbol{\theta}_{\mathcal{NN}} \in \mathbb{R}^p$, $\boldsymbol{A} \in \mathbb{R}^{\gamma\times(d+1)}$, $\boldsymbol{b} \in \mathbb{R}^{d+1}$, $c \in \mathbb{R}$, where rank $\gamma = \min(10, d+1)$ limits the number of parameters in $\boldsymbol{A}^\top\boldsymbol{A}$. Here, $\boldsymbol{A}$, $\boldsymbol{b}$, and c model quadratic potentials, that is, linear dynamics; $\mathcal{NN}$ models nonlinear dynamics. For certain experiments, we may choose to omit the quadratic potential terms $\boldsymbol{A}$, $\boldsymbol{b}$ and c for comparison or simplicity reasons.

In all of our experiments, for $\mathcal{NN}$, we either use a MLP from [42], which can be defined with

$$\begin{aligned} \boldsymbol{a}_0 &= \mathrm{act}(\boldsymbol{K}_0\boldsymbol{y} + \boldsymbol{b}_0) \\ \boldsymbol{a}_{i+1} &= \mathrm{act}(\boldsymbol{K}_{\mathrm{i}+1}\boldsymbol{a}_{\mathrm{i}} + \boldsymbol{b}_{\mathrm{i}+1}), \quad 0 \le \mathrm{i} \le \mathrm{M}-2 \\ \mathcal{NN}(\boldsymbol{y};\boldsymbol{\theta}_{\mathcal{NN}}) &= \mathrm{act}(\boldsymbol{K}_{\mathrm{M}}\boldsymbol{a}_{\mathrm{M}-1} + \boldsymbol{b}_{\mathrm{M}}), \end{aligned} \tag{4.2}$$

or a residual neural network (ResNet), presented in [46]

$$\begin{aligned} \boldsymbol{a}_0 &= \mathrm{act}(\boldsymbol{K}_0\boldsymbol{y} + \boldsymbol{b}_0) \\ \boldsymbol{a}_{i+1} &= \boldsymbol{a}_i + \mathrm{act}(\boldsymbol{K}_{\mathrm{i}+1}\boldsymbol{a}_{\mathrm{i}} + \boldsymbol{b}_{\mathrm{i}+1}), \quad 0 \leq \mathrm{i} \leq \mathrm{M}-2 \\ \mathcal{NN}(\boldsymbol{y};\boldsymbol{\theta}_{\mathcal{NN}}) &= \boldsymbol{a}_{M-1} + \mathrm{act}(\boldsymbol{K}_{\mathrm{M}}\boldsymbol{a}_{\mathrm{M}-1} + \boldsymbol{b}_{\mathrm{M}}), \end{aligned} \tag{4.3}$$

with neural network weights $\boldsymbol{\theta}_{\mathcal{NN}}=(\boldsymbol{K}_0, \ldots, \boldsymbol{K}_M, \boldsymbol{b}_0, \ldots, \boldsymbol{b}_M)$ where $\boldsymbol{b}_i \in \mathbb{R}^m \ \forall i$, $\boldsymbol{K}_0 \in \mathbb{R}^{m\times(d+1)}$, and $\{\boldsymbol{K}_1, \ldots, \boldsymbol{K}_M\} \in \mathbb{R}^{m\times m}$ with M being the depth of the network. The choice of the element-wise nonlinearity $\mathrm{act}(\cdot)$ is discussed in the respective experiments.

As a brief summary, our learning problem has little restriction on neural network architectures, as such, one can experiment with any other network architecture as long as the selected function approximators have the expressiveness to learn the desired value function. In our test cases both neural network models showed sufficiently accurate convergence results.

4.2 Formulation of the Training Problem

In this section we describe our formulation of the learning problem, we aim to solve both the SOC problem as well as its corresponding HJB equation at the same time.

Ideally, we would choose $\boldsymbol{\theta}$ such that $\Phi(s, \boldsymbol{z}; \boldsymbol{\theta})$ is equal to the value function of the control problem globally, that is, for all $(s, \boldsymbol{z}) \in [t, T] \times \mathbb{R}^d$. Since this is known to be cursed by the dimensionality for reasonable problem sizes, we resort to a semi-global approach, which enforces this property at randomly sampled points in the space-time domain. Selecting these sampled regions will be of key importance.

To generate samples, we first obtain initial states $\boldsymbol{x} \sim \rho$ from some (possibly Dirac) distribution ρ and then use an Euler Maruyama scheme with $N+1$ equidistant time points $s_0, \ldots, s_N$ and step size $\mathrm{d}s = (T-t)/N$. This yields a state trajectory starting

at $\boldsymbol{z}_0 = \boldsymbol{x}$ via

$$\boldsymbol{z}_{i+1} = \boldsymbol{z}_i + f(s_i, \boldsymbol{z}_i, \boldsymbol{u}_i)\mathrm{d}s + \sigma(s_i, \boldsymbol{z}_i)\mathrm{d}\boldsymbol{W}_i, \quad i = 0, \dots, N-1 \tag{4.4}$$

corresponding to eq. (2.11) where $\mathrm{d}\boldsymbol{W}_i \sim \mathcal{N}(\boldsymbol{0}, \mathrm{d}s \cdot \boldsymbol{I}_d)$, and $\boldsymbol{u}_i = \boldsymbol{u}^*_{t,x}(s_i, \boldsymbol{z}_i)$ is the optimal control obtained from the feedback, that is, form eq. (2.16)

$$\boldsymbol{u}^*_i \in \arg\max_{\boldsymbol{u} \in U} \mathcal{H}\left(s_i, \boldsymbol{z}_i, -\nabla\Phi(s_i, \boldsymbol{z}_i; \boldsymbol{\theta}), -\sigma(s_i, \boldsymbol{z}_i)^\top \nabla^2\Phi(s_i, \boldsymbol{z}_i; \boldsymbol{\theta}), \boldsymbol{u}\right).$$

A few comments are in place. First, it is important to note that due to the feedback form, the sampled trajectories depend on the parameters of the value function, different parametrizations of the value function will yield different forward trajectories. Second, the addition of this drift term, motivated by control theory, is one of the key differences to neural network solvers for the more general class of semi-linear elliptic PDEs as presented in [44, 91]. Third, another way to view our choice of the drift term is by the fact that for $\sigma \to 0$, the trajectories defined above approximate the characteristic curves of the non-viscous HJB equation corresponding to the deterministic counterpart of the control problem, thus making our SOC approach consistent with that for deterministic OC in [78].

It is however worth pointing out some key differences compared to the similar method proposed for deterministic OC problems in [78]. Due to the random noise introduced even for a fixed initial state, the optimal path may vary at each step of time integration. Unlike in the deterministic case, in order to avoid being stuck in less desirable local minima, repeated re-sampling and parameter updating will be needed to properly optimize the model, this will generally result in more iteration and time for training.

To further simplify the notations, we omit the subscript $\boldsymbol{z}$ in $\nabla_{\boldsymbol{z}}\Phi$ and $\nabla^2_{\boldsymbol{z}}\Phi$ for the rest of the paper. Furthermore, we collect the states, control, and noise along the

discrete trajectories in eq. (4.4) column-wise in the matrices

$$\boldsymbol{Z} \in \mathbb{R}^{d\times N}, \quad \boldsymbol{U} \in \mathbb{R}^{k\times N}, \quad \mathbf{dW} \in \mathbb{R}^{d\times N}.$$

To learn the parameters of the neural networks in an unsupervised way (that is, assuming neither analytic values of Φ nor optimal control trajectories), we approximately solve the minimization problem

$$\begin{aligned}\min_{\boldsymbol{\theta}} \mathbb{E}_{\boldsymbol{x}\sim\rho}\left\{\mathbb{E}_{\boldsymbol{Z},\boldsymbol{U},\mathbf{dW}|\boldsymbol{x}}\left\{\beta_1 P^p_{\mathrm{BSDE}}(\boldsymbol{Z},\boldsymbol{U},\mathbf{dW}) + \beta_2 P^p_{\mathrm{HJB}}(\boldsymbol{Z}) + \beta_3 J(\boldsymbol{Z},\boldsymbol{U})\right.\right.\\ \left.\left.+\beta_4|G(\boldsymbol{z}_N) - \Phi(s_N,\boldsymbol{z}_N;\boldsymbol{\theta})|^p + \beta_5|\nabla G(\boldsymbol{z}_N) - \nabla\Phi(s_N,\boldsymbol{z}_N;\boldsymbol{\theta})|^p\right\}\right\},\end{aligned} \tag{4.5}$$

where the terms in the objective function consist of penalty functions for violations of the BSDE system and the HJB equation, the control objective, and penalty terms for the terminal condition, respectively, and are defined below. The exponent $p \in \{1,2\}$ allows one to choose between different norms for the loss function. In our numerical examples, we notice that using $p = 1$ favors the minimization of the control objective and therefore gives much faster convergence, on the other hand choosing $p = 2$ emphasizes minimization of the BSDE and HJB loss, which can have higher accuracy for approximating the value function but converges much slower. The relative influence of each term is controlled by the components of $\beta \in \mathbb{R}^5_+$ and is of high importance in hyperparameter tuning. Different choices of β allow us to experiment with different learning approaches; for example, setting $\beta_1 = \beta_4 = \beta_5 = 1$ and $\beta_2 = \beta_3 = 0$ provides the same loss function as in [91] while $\beta_1 = 0$ and $\beta_i > 0,\ i \in \{2,3,4,5\}$ gives the loss function used for deterministic OC problems in [78].

We penalize the violation of the BSDE eq. (2.24) via

$$P_{\mathrm{BSDE}}(\boldsymbol{Z},\boldsymbol{U},\mathbf{dW}) = \sum_{i=0}^{N-1} |\Phi_{i+1}(\boldsymbol{\theta}) - \Phi_i(\boldsymbol{\theta}) + L(s_i,\boldsymbol{z}_i,\boldsymbol{u}_i)\mathrm{d}s - \nabla\Phi_i(\boldsymbol{\theta})^\top \sigma(s_i,\boldsymbol{z}_i)\mathrm{d}\boldsymbol{W}_i| \tag{4.6}$$

where we use the abbreviations $\Phi_i(\boldsymbol{\theta}) := \Phi(s_i, \boldsymbol{z}_i; \boldsymbol{\theta})$ and $\nabla\Phi_i(\boldsymbol{\theta}) := \nabla\Phi(s_i, \boldsymbol{z}_i; \boldsymbol{\theta})$. Similarly, the HJB penalty term reads

$$P_{\mathrm{HJB}}(\boldsymbol{Z}) = \mathrm{d}s \sum_{i=1}^{N} |H(s_i, \boldsymbol{z}_i, -\nabla\Phi_i(\theta), -\sigma(s_i, \boldsymbol{z}_i)^\top \nabla^2\Phi_i(\theta)) - \partial_s\Phi_i(\boldsymbol{\theta})|, \tag{4.7}$$

where $\nabla^2\Phi_i(\theta) := \nabla^2\Phi(s_i, \boldsymbol{z}_i; \theta)$, $\partial_s\Phi_i(\theta) := \partial_s\Phi(s_i, \boldsymbol{z}_i; \theta)$. Finally, we approximate the objective functional via

$$J(\boldsymbol{Z}, \boldsymbol{U}) = G(\boldsymbol{z}_N) + \mathrm{d}s \sum_{i=1}^{N} L(s_i, \boldsymbol{z}_i, \boldsymbol{u}_i).$$

In principle, any stochastic approximation approach can be used to approximately solve the above optimization problem. Here, we use Adam [54] and sample a mini-batch of trajectories originating in i.i.d. samples from ρ at each optimization step.

We find this to be the appropriate section to discuss more details of our proposed approach. Our method shares much similarity with the PINNs approach to solving the same HJB equation in both the definition of the loss function and a sample based optimization scheme. The differences lie in two main ways, first our inclusion of the control objective is unique to SOC problems and greatly changes the optimization process compared to minimizing only the HJB loss. Unlike in PINNs where a random sampling scheme over the entire space-time domain is usually used for batch minimization, our PMP inspired sampling scheme focuses only on a small yet relevant portion of the state-time space where the optimal solution lies, this difference can be amplified as the dimension of the problem grows.

4.3 Numerical Experiments for SOC Problems and HJB Equations

We test the efficacy of our proposed algorithm on several different (Stochastic) OC problems. First, we introduce a two-dimensional trajectory planning problem to visualize the difference between purely random exploration and our proposed sampling scheme. To illustrate the accuracy of the learned value function, we compare it with the value function obtained by solving the corresponding HJB PDE using a finite element method (FEM). The goal of this experiment is to compare the accuracy of the neural network and FEM approximation, and not to compete, with the FEM. Secondly, we introduce a 100-dimensional benchmark problem and compare our approach to those in [35, 44] through the benchmark problem. For the original version of this problem, our method shows faster initial convergence and time-to-solution with comparable accuracy. We modify the terminal cost of this problem to further highlight the importance of the feedback form in the sampling, while our method can still recover a reasonable solution to the modified problem, approaches without the feedback form can not. Lastly, we also test our method on a 12-dimensional problem with nonlinear dynamics, comparing with solution obtained from the deterministic version of the problem in [78], showing that our method generates relatively accurate solutions under complex dynamics.

4.3.1 Implementation Details

We implement and test our proposed approach in two software environments. To obtain a direct comparison with [91] we modify the FBSNN code accompanying the paper. To this end, we created a publicly available fork at `https://github.com/EmoryMLIP/FBSNNs`. Our two main modifications are adding the proposed drift to the forward dynamics and adding the control objective in the training loss. Other

parameters, including the choice of neural network model, are kept unchanged.

In order to further simplify the experimentation, we also implement our own PyTorch code available at `https://github.com/EmoryMLIP/NeuralSOC`. Our implementation contains all loss terms in eq. (4.5). We implement both sampling techniques: pure random walk and the proposed one informed by PMP. This facilitates comparisons of our approach with other available methods and simplifies developing new examples.

We tested most of our examples using either Intel Xeon E5-4627 CPU or Nvidia P100 GPU.

4.3.2 2D Trajectory Planning Problem

To visualize the behavior of our PMP-based sampling approach, we consider a two-dimensional test problem.

The problem consists of planning an optimal trajectory from the initial state that follows a Gaussian distribution $\boldsymbol{x} \sim \rho = \mathcal{N}((-1.5, -1.5)^\top, 0.4 \cdot \boldsymbol{I}_2)$ to the target $\boldsymbol{x}_{\text{target}} = (1.5, 1.5)^\top$. To make the problem more interesting, a hill is placed at the origin, denoted by $Q(\boldsymbol{z})$, which adds height-dependent cost for traveling around that region. In our experiments, $Q(\boldsymbol{z})$ is defined by a two-dimensional Gaussian density with mean zero and covariance of $0.4 \cdot \boldsymbol{I}_2$ scaled by a factor of 50.

The dynamics for the problem read

$$f(s, \boldsymbol{z}, \boldsymbol{u}) = \boldsymbol{u} \quad \text{and} \quad \sigma = \begin{bmatrix} 0.2 & -0.4 \\ -0.4 & 0.2 \end{bmatrix}. \tag{4.8}$$

The choice of non-scalar σ adds to the complexity of the problem by changing the behavior of the standard Brownian motion, see fig. 4.1.

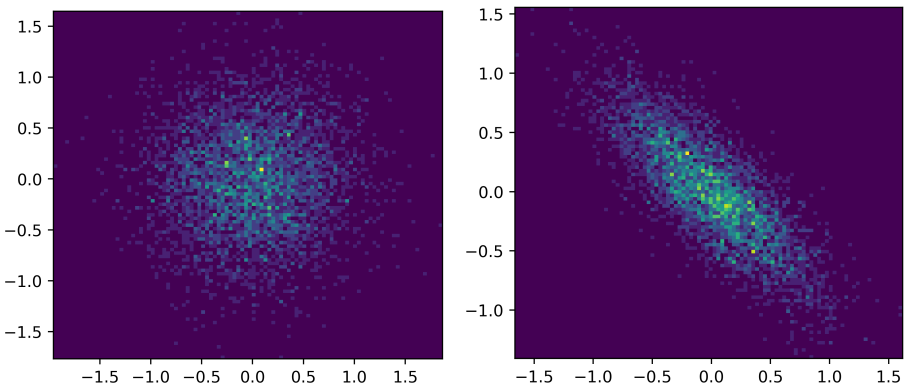


Figure 4.1: Action of σ in (4.8) on standard Gaussian distribution (Left) warps it diagonally (Right). This would affect the solution of the problem.

The running cost and terminal cost of the problem are given, respectively, by

$$L(s, \boldsymbol{z}, \boldsymbol{u}) = \frac{1}{2}\|\boldsymbol{u}\|^2 + Q(\boldsymbol{z}) \quad \text{and} \quad G(\boldsymbol{z}) = 50 \cdot \|\boldsymbol{z} - \boldsymbol{x}_{\text{target}}\|^2. \tag{4.9}$$

Our objective is to find an optimal path between each given initial state and the target state, the traveling agent should also balance between taking the shortest path and avoiding the obstacle.

The corresponding HJB equation can be derived as

$$\partial_s \Phi(s, \boldsymbol{z}) + \frac{1}{2}\text{tr}(\sigma\sigma^\top \nabla^2 \Phi(s, \boldsymbol{z})) - \frac{1}{2}\|\nabla \Phi(s, \boldsymbol{z})\|^2 + Q(\boldsymbol{z}) = 0. \tag{4.10a}$$

with terminal condition

$$\Phi(T, \boldsymbol{z}) = G(\boldsymbol{z}). \tag{4.10b}$$

Finite Element Method

Since it is not obvious how to solve the HJB equation eq. (4.10) analytically, we resort to approximately solving it using a finite element method (FEM) to obtain a baseline for this problem.

The HJB equation is defined over the entire state space without an explicit bound-

ary condition, for simplicity we approximate the value function by solving the HJB PDE eq. (4.10) on the restricted domain $\Omega = [-3, 3] \times [-3, 3]$ with homogeneous Neumann boundary conditions,

$$\frac{\partial \Phi}{\partial \hat{\boldsymbol{n}}}(s, \boldsymbol{z}) = 0, \text{ on } \partial\Omega, \ \forall s < T,$$

where $\hat{\boldsymbol{n}}$ denotes the unit normal vector. Since the diffusion coefficient σ is independent of time and space, $\text{tr}(\sigma\sigma^\top \nabla^2 \Phi(s, \boldsymbol{z})) = \text{div}(\sigma\sigma^\top \nabla\Phi(s, \boldsymbol{z}))$, which we can then use to derive a weak form of the PDE. Using the implicit Euler discretization in time on a partition of $[0, T]$ into N sub-intervals with uniform step size, $\mathrm{d}s$, yields

$$\frac{\Phi_{n+1} - \Phi_n}{\mathrm{d}s} + \frac{1}{2}\text{div}(\sigma\sigma^\top \nabla\Phi_n) - \frac{1}{2}\|\nabla\Phi_n\|^2 + Q = 0, \quad n = N, N-1, \ldots, 0,$$

where Φ_n denotes the approximated solution $\Phi(t_n, \cdot)$, at $t_n = n \cdot \mathrm{d}s$ and $\Phi_{N+1} = G(\cdot)$. Then, using Green's formula, the weak problem at the n-th time step consists of finding $\Phi_n \in H^1(\Omega)$ such that

$$\int_\Omega (\Phi_{n+1} - \Phi_n) v \mathrm{d}\boldsymbol{z} - \mathrm{d}s\frac{1}{2}\int_\Omega \sigma\sigma^\top \nabla\Phi_n \cdot \nabla v \mathrm{d}\boldsymbol{z} + \mathrm{d}s \int_\Omega \left(Q - \frac{1}{2}\|\nabla\Phi_n)\|^2\right) v \mathrm{d}\boldsymbol{z} = 0,$$

for all test functions $v \in H^1(\Omega)$. Here, $H^1(\Omega)$ denotes the Hilbert Sobolev space defined by $H^1(\Omega) = \{v \in L^2(\Omega) | \nabla v \in L^2(\Omega)\}$.

To solve the problem in weak form numerically we use FEniCS [60], we create a triangular mesh for Ω and use $\mathcal{P}_1$ Lagrange finite elements to discretize Φ in space. We discretize Ω using 150 mesh points in each dimension, summing up to a total of 22,500 degrees of freedom, and use the step size of $\mathrm{d}s = 0.001$ in time. At each time step, we use Newton's method to solve for Φ_n, with relative error and absolute error tolerance for the solver set to 10^{-6} and 10^{-10}, respectively. We denote the FEM solution by Φ_{FEM}.

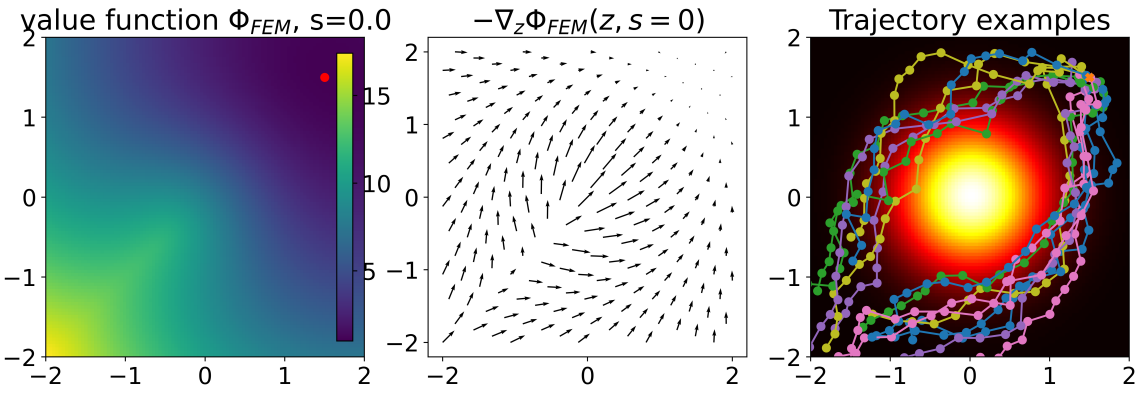


Figure 4.2: Results of the two-dimensional test problem. Left: Value function approximation $\Phi_{\text{FEM}}(0, \cdot)$. Middle: Quiver plot of optimal controls at $s = 0$. Right: Trajectories generated from randomly chosen initial states.

In fig. 4.2 we plot the solution Φ_{FEM} as well as the optimal control policy at initial time $s = 0$, which we obtained via the feedback form. We also present trajectory examples originating from some randomly chosen initial states following the optimal policy. As expected, the trajectories travel from the initial points to the target while avoiding the obstacle in the center of the domain. It's also worth noting that given the relatively large noise we used for the example, the resulting trajectories can vary greatly despite sharing the same initial states.

In table 4.2, we evaluate the control objective, J, for some fixed initial state. We notice that the estimated value matches with $\Phi_{\text{FEM}}(0)$, suggesting that the FEM solution is an accurate approximation of the true Φ.

Another thing we want to point out is that FEM is sufficient and suitable for the 2D parabolic equation we have here since a variational form is explicitly available. However for problems without an easily accessible variational form, FEM may not be an ideal choice and one may want to resort to methods mentioned in [19, 57] for baseline solutions.

Neural Network Approach

For the problem defined in eq. (4.8) and eq. (4.9), the forward SDE eq. (4.4) simplifies to

$$\boldsymbol{z}_{i+1} = \boldsymbol{z}_i - \nabla\Phi(s_i, \boldsymbol{z}_i)\mathrm{d}s + \sigma\mathrm{d}\boldsymbol{W}_i, \tag{4.11}$$

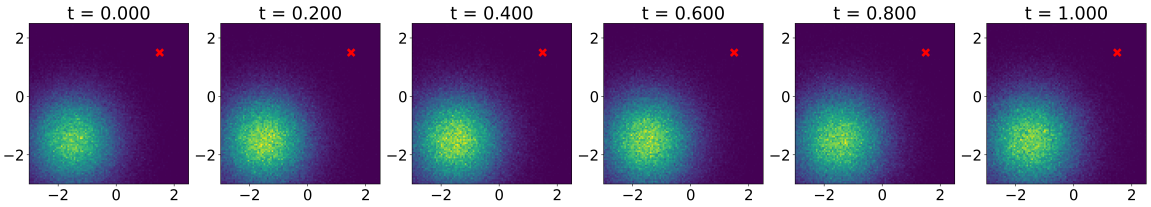
with feedback form which reads

$$\boldsymbol{u} = -\nabla\Phi(s, \boldsymbol{z}). \tag{4.12}$$

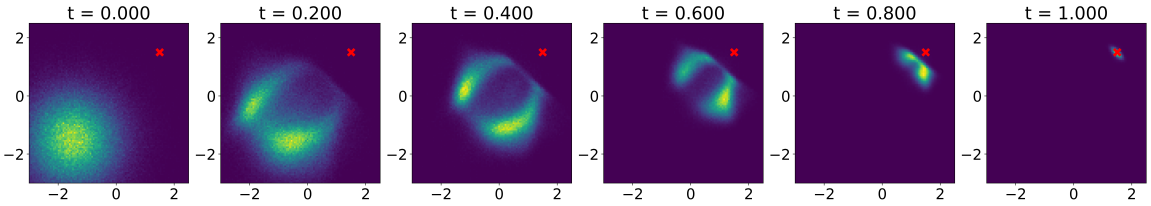
Following our proposed method in eq. (4.1), we approximate the value function using a three-layer residual neural network with 32 neurons per layer. We do not include the quadratic terms in the network for this experiment since we find the simpler structure was already sufficient for solving this problem. As a result, the model overall consists of 1217 trainable parameters. We choose tanh as the activation function for all but the final layer of the network, the final layer does not have an activation function.

For penalty parameters, we select $\beta = (1.0.1.0, 1.0, 1.0, 0.0)$, that is, we enable both the penalty terms, P_{BSDE} and P_{HJB} in eq. (4.5) along with the control objective. To approximately solve eq. (4.5) we train a total of 6,000 steps with a batch size of 64, we use Adam optimizer for each update with no additional weight decay. For learning rate scheduling, we start with a learning rate of 0.01 and divide it by 10 every 1800 iterations. The average cost per iteration is about 0.22s when running on a NVIDIA P100 GPU.

Note that for the chosen σ in eq. (4.8) which is non-scalar, full Hessian information of the value function, $\nabla^2\Phi$, is required to calculate P_{HJB}. To this end, though it is possible to obtain the full Hessian using automatic differentiation, we instead use the efficient implementation in the package "hessQuik" [75]. We will refer to the neural network approximated solution as Φ_{NN} in the remainder of the section. We also notice



(a) Training samples with pure random walk as FSDE as also used in [44] and [91].



(b) Training samples with PMP-based drift term.

Figure 4.3: We visualize training samples of a pure random walk sampler (top row) and our proposed PMP-based sampler (bottom row) for the two-dimensional test problem. At six time points (left to right), we visualize the sampled states as two-dimensional histograms. As expected, the pure random walk explores the area around the initial state in all (even suboptimal) directions, while the proposed approach learns to sample around approximately optimal trajectories.

that sampling the initial states from a slightly larger area than what is given during training often helps the robustness of the learned model.

Given the stochastic nature of the problem and the random initialization of neural network weights, each training sequence can produce a slightly different model. To account for this, we repeat the training ten times and obtain neural network approximations of the value functions $\Phi_{\mathrm{NN}}^{(j)}$, where $1 \leq j \leq 10$. We compare the resulting 10 models to the FEM solution in the following subsections.

To gain more insight into the sampling, we store all states visited during training and plot them as two-dimensional histograms for different time points (left to right) in fig. 4.3. We compare our proposed PMP-based sampling (fig. 4.3b) to the purely noise driven dynamics (fig. 4.3a), that is, without the drift term altogether, as used in works such as [44, 91]. As expected, the use of purely noisy dynamics leads to the sampling of points only around the initial states in all (even sub-optimal) directions with almost no samples close to the target. On the other hand, with the use of the

drift term, the sampled states visit the paths between the initial and target states.

Another way to interpret the histogram plots in fig. 4.3 is by observing the semi-global nature of our neural network approach for SOC problems. Since the loss function in eq. (4.5) only minimizes the HJB and BSDE losses in a neighborhood of points sampled using the forward SDE, one would expect the trained model to be more reliable in regions that are frequently visited. On the other hand, with the purely noise driven dynamics models are only optimized near the initial sampling region, one should not expect the model to have high accuracy beyond that region, especially near the target.

Comparison

In this subsection, we compare the neural network models $\Phi_{\text{NN}}^{(1)}, \ldots, \Phi_{\text{NN}}^{(10)}$ and Φ_{FEM} obtained from previous subsections along the approximately optimal trajectories. Specifically, we randomly sample initial states from ρ and simulate the trajectories using the trained models. We believe that this approach enables a meaningful comparison since the training procedure focuses on those parts of the state space visited by the trajectories and hence the neural networks approximate the value function semi-globally. It is in our opinion important to focus on the relevant space when comparing accuracy of solutions.

For each trained model, we record all the sampled states visited at times $s \in \{0, 0.5, 0.9\}$ while following the corresponding learned policy. We then compare the learned value functions Φ_{NN}^{j} with the reference solution Φ_{FEM} at all these points.

In fig. 4.4, we plot the comparison for three different times $s \in \{0, 0.5, 0.9\}$ along the rows. The first, second, and fourth columns represent neural network models $\Phi_{\text{NN}}^{(1)}$, $\Phi_{\text{NN}}^{(2}$), and Φ_{FEM} at the sampled points, respectively. We observe that value function estimates look similar to the finite element reference solution. The third column shows the average results of the ten learned value functions obtained from

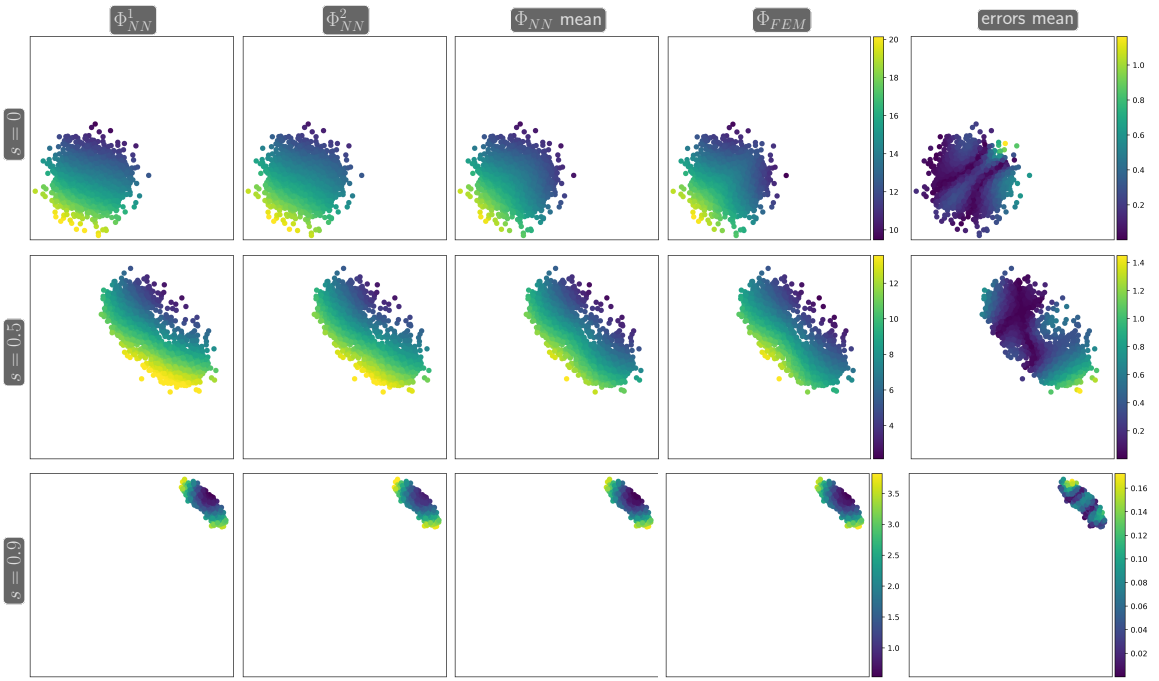


Figure 4.4: Comparison between learned value function (First 3 columns, including both individual model and model average) and the FEM solution (fourth column) at different time shots $s = 0, 0.5$ and 0.9. From the errors (fifth column), the neural network solution matches the FEM solution closely over the sampled region.

Time snapshot	$s = 0$	$s = 0.5$	$s = 0.9$
AE (mean $\pm$ std)	0.31 ± 0.17	0.47 ± 0.44	0.17 ± 0.18
RE (mean $\pm$ std)	0.02 ± 0.01	0.06 ± 0.06	0.15 ± 0.17

Table 4.1: Average absolute and relative error between $\Phi^{(1)}_{\text{NN}}, \ldots, \Phi^{(10)}_{\text{NN}}$ and Φ_{FEM} across all sampled points at different time steps.

the ten training sequences. Lastly, the last column displays the average absolute mean errors between the learned value functions and Φ_{FEM}.

In table 4.1, we compare mean of the absolute and relative errors between Φ_{FEM} and $\Phi^{(j)}_{\text{NN}}$, $1 \leq j \leq 10$, across the sampled points shown in fig. 4.4 for all ten trained models, computed via

$$\text{AE}(s) = \frac{1}{n_{\text{samples}} \times n_{\text{models}}} \sum_{i=1}^{n_{\text{samples}}} \sum_{j=1}^{n_{\text{models}}} \left| \Phi^{(j)}_{\text{NN}}(s, \boldsymbol{z}_i) - \Phi_{\text{FEM}}(s, \boldsymbol{z}_i) \right| \tag{4.13}$$

Initial state	$\Phi_{\text{FEM}}(0)$	$\Phi_{\text{NN}}(0)$	J_{FEM}	J_{NN}
$\boldsymbol{x}_{\text{init}} = (-1.5, -1.5)^\top$	14.67	14.48	14.68	15.33

Table 4.2: Discrepancy between the value function Φ and the control objective J at some initial state.

and

$$\text{RE}(s) = \frac{1}{n_{\text{samples}} \times n_{\text{models}}} \sum_{i=1}^{n_{\text{samples}}} \sum_{j=1}^{n_{\text{models}}} \frac{\left|\Phi_{\text{NN}}^{(j)}(s, \boldsymbol{z}_i) - \Phi_{\text{FEM}}(s, \boldsymbol{z}_i)\right|}{|\Phi_{\text{FEM}}(s, \boldsymbol{z}_i)|}. \tag{4.14}$$

Our observations indicate that the relative error is smallest at the initial time and increases over time, while the average absolute error remains fairly constant across all states and time intervals. We believe one possible explanation for such difference is the fact that the true value function decreases as time increases, the relative errors are therefore amplified given how they are calculated. Inclusion of the control objective, J, in the training loss function may also play a role in the errors observed, as the optimization problem has multiple objectives. Furthermore, the errors across all the trained models display a relatively low standard deviation, indicating that our proposed training scheme is robust to random initialization.

In table 4.2, we compare the value function approximation for one of the trained models, $\Phi_{\text{NN}}^{(1)}$, to the value of the control objective, $J(-\nabla\Phi_{\text{NN}})$, at the initial state $\boldsymbol{x} = (-1.5, -1.5)^\top$ and time $t = 0$. Since the system dynamics are stochastic, we generate $12,000$ trajectories starting from $\boldsymbol{x}$ using the learned feedback control to calculate the control objective J for each trajectory. We then use the sample average as a proxy for the expected value. We use a finer step size of $\mathrm{d}s = 0.005$ than the one used in training to get a more accurate approximation of J. We observe that the discrepancy between the value estimate and the actual cost is almost negligible for the FEM solution, which is to be expected. For the neural network approximation, the value estimate is about 4% smaller than the actual control objective, which indicates

that the value estimates can be overly optimistic. We consider this to be mostly satisfactory as Φ_{NN} still approximates J fairly well, despite the fact that in our training problem we do not explicitly penalize the discrepancy between $\Phi_{\mathrm{NN}}(0)$ and J.

On the hardware used for our experiments, both approaches showed comparable time-to-solution. The neural network training took approximately 20 minutes using the GPU, while the FEM solution was obtained in roughly one hour using the CPU. However, the FEM approach requires a computational mesh, making it infeasible for $d > 4$, which is the primary use case for our proposed method.

Impact of Penalty Parameter Selection

In this section we briefly describe the importance of selecting appropriate hyperparameters, namely the β. Though solving both the SOC problem and the HJB equation accurately gives the same value function, in practice solving the multi-objective minimization problem can be very hard and carefully choosing hyperparameters is often needed for ideal results. From our observation only using the control objective and excluding both HJB and BSDE penalties from the loss function can lead to a fast convergence, however the solution will be suboptimal with the value function being incorrect. On the other hand only minimizing the HJB/BSDE loss will often result in the value function being minimized in a wrong solution region, rendering the final solution obsolete.

4.3.3 100-dimensional example

In this section we consider a 100-dimensional benchmark SOC problem also used in [35, 44] with fixed initial state $\boldsymbol{x} = (0, 0, \ldots, 0)^\top \in \mathbb{R}^{100}$ corresponding to time $t = 0$. The drift and diffusion of the system are given by

$$f(s, \boldsymbol{z}, \boldsymbol{u}) = 2\boldsymbol{u} \quad \text{and} \quad \sigma = \sqrt{2},$$

respectively. The terminal and Lagrangian cost for the problem are

$$G(\boldsymbol{z}) = \ln\left(\frac{1+\|\boldsymbol{z}\|^2}{2}\right), \quad \text{and} \quad L(s,\boldsymbol{z},\boldsymbol{u}) = \|\boldsymbol{u}\|^2, \tag{4.15}$$

respectively. We can compute the Hamiltonian eq. (2.14) of the system as

$$\begin{aligned} H(s,\boldsymbol{z},\boldsymbol{p},\boldsymbol{M}) &= \sup_{\boldsymbol{u}\in U}\left\{\frac{\sigma}{2}\mathrm{tr}\left(\boldsymbol{M}\right) + \boldsymbol{p}\cdot f(s,\boldsymbol{z},\boldsymbol{u}) - L(s,\boldsymbol{z},\boldsymbol{u})\right\} \\ &= \sup_{\boldsymbol{u}\in U}\left\{\frac{1}{\sqrt{2}}\mathrm{tr}\left(\boldsymbol{M}\right) + \boldsymbol{p}\cdot 2\boldsymbol{u} - \|\boldsymbol{u}\|^2\right\}. \end{aligned}$$

Using the first-order necessary condition of the Hamiltonian we get

$$0 = 2\boldsymbol{u} - 2\boldsymbol{p} \implies \boldsymbol{u} = \boldsymbol{p},$$

and using this closed form for $\boldsymbol{u}$, the Hamiltonian can be then simplified to

$$H(s,\boldsymbol{z},\boldsymbol{p},\boldsymbol{M}) = \frac{1}{\sqrt{2}}\mathrm{tr}\left(\boldsymbol{M}\right) + \|\boldsymbol{p}\|^2.$$

Hence, the HJB equation satisfied by the value function, $\Phi(\cdot,\cdot)$, reads

$$\frac{\partial}{\partial s}\Phi(s,\boldsymbol{z}) + \Delta\Phi(s,\boldsymbol{z}) - \|\nabla\Phi(s,\boldsymbol{z})\|^2 = 0, \tag{4.16}$$

with terminal condition

$$\Phi(T,\boldsymbol{z}) = G(\boldsymbol{z}).$$

For this particular problem, an explicit solution can be obtained through the Cole-Hopf transformation, see [21, 30] and has the form

$$\Phi(s,\boldsymbol{z}) = -\ln\left(\mathbb{E}\left(\exp\left(-G\left(\boldsymbol{z} + \sqrt{2}\,\mathrm{d}W(T-s)\right)\right)\right)\right), \tag{4.17}$$

which we can use to test the performance of our method.

Finally, we note that the forward SDE eq. (4.4) we propose to use for sampling the state space simplifies to

$$\boldsymbol{z}_{i+1} = \boldsymbol{z}_i - 2\,\nabla_{\boldsymbol{z}}\Phi(s_i, \boldsymbol{z}_i)\mathrm{d}s + \sqrt{2}\mathrm{d}\boldsymbol{W}_i. \tag{4.18}$$

The importance of sampling

To demonstrate the impact of using the feedback form to sample the state space, as well as our proposed loss function, we conduct a direct comparison to the method proposed in [91] on the benchmark problem.

For approximating the value function, we use the same neural network model as described in [91], which is given by a five-layer feed-forward neural network with 256 neurons per hidden layer to approximate the solution $\Phi(s, \boldsymbol{z})$. We partition the time interval $[0, 1]$ using 50 uniformly spaced points.

For penalty terms, we use the same penalty parameters as in the original code with added control objective for fair comparison, that is, $\beta = (1, 0, 20, 1, 1)$. We use the Adam optimizer [54] to update the parameters of the network with a batch size of 64 using a total of 50,000 iterations. This results in the average cost per 100 iterations being around 27s using the CPU. For the following experiments, notice the main differences between our approach and [91] lie in two ways, first is the use of a PMP driven dynamics versus a pure noise driven one, second is the inclusion of the control objective alongside the HJB penalty in the loss function.

Method	20k iterations		50k iterations	
	RE	RE_0	RE	RE_0
FBSNN	0.54%	0.12%	0.39%	0.045%
Ours	0.48%	0.0083%	0.39%	0.012%

Table 4.3: Relative errors for eq. (4.16) obtained using our method and method in [91]

Given the explicit solution from eq. (4.17) we can evaluate the accurate value func-

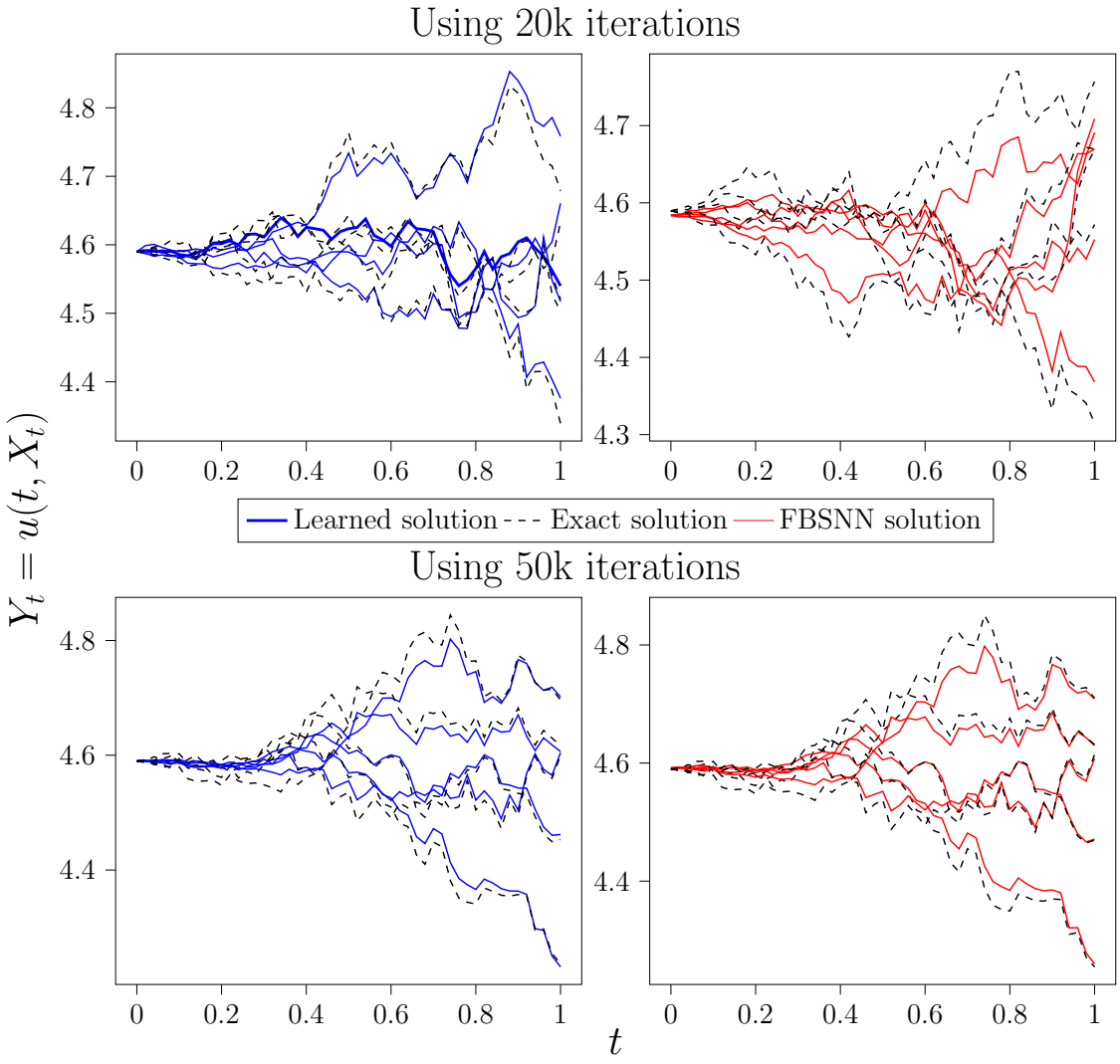


Figure 4.5: Solution to eq. (4.16) obtained using our method (left column) and the method in [91] (right column)

tion given arbitrary time and state, for our experiments the true value function $\Phi(s, \boldsymbol{z})$ is calculated as the expected value over 10K random samples following eq. (4.17). In Figure 4.5 we plot the exact solution (black-dashed line) eq. (4.17), the learned solution using our approach (blue-solid line) and the solution learned using FBSNNs (red-solid line) [91] along five random trajectories. In the top row, we present the results obtained after training the networks for 20,000 iterations with a learning rate of 10^{-3} and the bottom row presents the results after training the networks for 20K and 30K iterations with learning rates 10^{-3} and 10^{-4}, respectively. From the figure it

is clear that our approach approximates the value function better, especially in early iterations, as compared to the FBSNNs which has observable higher errors.

In table 4.3, we also compare the learned solutions to the exact solution Φ in eq. (4.17) by computing the average relative errors,

$$RE = \frac{\|\Phi(\cdot,\cdot;\boldsymbol{\theta}) - \Phi(\cdot,\cdot)\|_2}{\|\Phi\|_2}, \quad RE_0 = \frac{|\Phi(0,\boldsymbol{z}(0);\boldsymbol{\theta}) - \Phi(0,\boldsymbol{z}(0))|}{|\Phi(0,\boldsymbol{z}(0))|},$$

for ten random trajectories. Our method attains lower errors, especially for the initial values and at the earlier iterations. We attribute this to the inclusion of the control objective into the loss function, similar to the 2D example, we observe that having the control objective as part of the minimization problem can allow the model to converge much faster.

Initial states from a distribution

We use this section to demonstrate the versatility of our method beyond fixed initial states, especially in addressing input states following a given distribution. Specifically, given the same problem as presented in previous sections, instead of using a fixed initial state, we sample $\boldsymbol{x}$ from a distribution $\rho = \mathcal{N}(\mathbf{0}, 0.5 \cdot \boldsymbol{I}_{100})$.

We repeat the training process with 20k iterations, maintaining most of the same hyperparameters, but increasing the batch size to 512 from 64 and choosing $\beta_3 = 50$. In fig. 4.6, we present the mean and variance of the relative errors of the errors relative to eq. (4.17) in the learned value function for ten random trajectories. As expected to the much higher complexity of the modified problem, the maximum relative error over the time interval increased to around 1.5%, which is slightly larger than in the original problem. Nevertheless given the results we can still conclude that our method is valid when given multiple initial states, though additional effort in training is needed for higher accuracy.

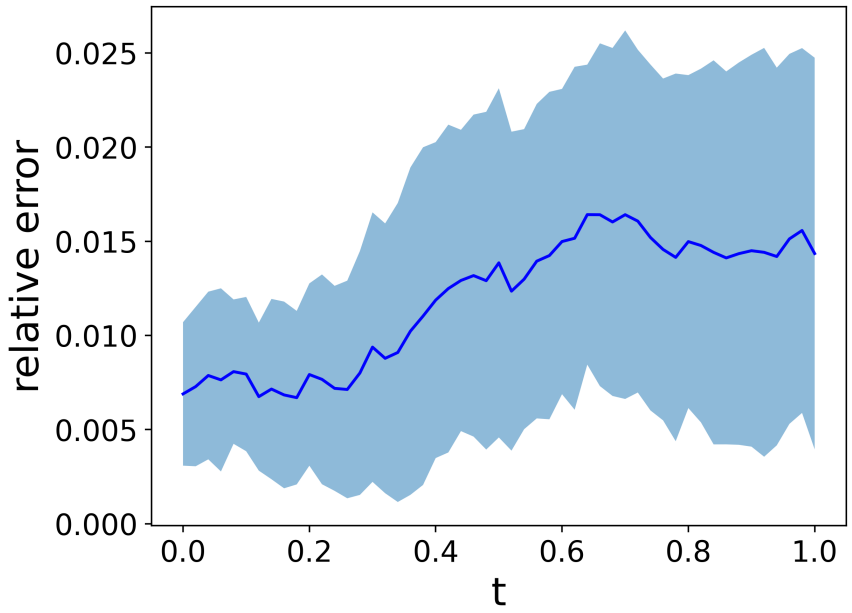


Figure 4.6: Mean and variance of the errors relative to eq. (4.17) in the learned value function for ten random trajectories obtained by sampling initial states from a distribution using our method after 20k iteration.

Shifted target

In the example above, one thing to note is that the minimizer of the terminal function coincides with the initial state at $\boldsymbol{x} = (0, 0, \ldots, 0)^\top$. Therefore, even a random walk without any drift (which is used in [91, 44]) will sample around the optimal terminal state similar to the demonstration in fig. 4.3a, which is critical to accurately approximate the value function. This also means that after training using our approach, the drift term in the sampler is relatively small and that the above experiment does not fully show the advantages of our method.

To shed more light on the importance of sampling, we modify the terminal cost to

$$G(z) = 1000 \ln \left(\frac{1 + \|\boldsymbol{z} - \boldsymbol{z}_{\text{target}}\|^2}{2} \right),$$

with $\boldsymbol{z}_{\text{target}} = (3, 3, \ldots, 3)^T$, so that the target for the state variable $\boldsymbol{z}$ at final time T no longer coincides with the initial state. Similar to the two-dimensional test problem

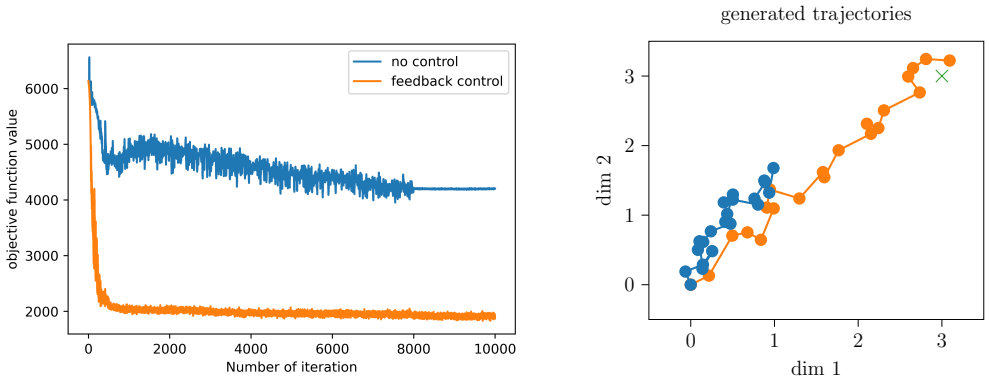


Figure 4.7: Computational results for the modified 100-dimensional benchmark problem in section 4.3.3. Left: Control objective for both methods given the same initial state, the blue line represents results using FBSNNs in [91], and the orange line denotes our method. Right: Trajectory examples generated using learned value functions on two randomly selected dimensions. The orange line represents our method and blue line FBSNNs.

in section 4.3.2, solving the modified problem now requires sampling states near the target and we expect to benefit from the added drift term in our PMP inspired forward dynamics.

We compare our method to FBSNNs on the modified problem while keeping the same network structure and most of the hyper-parameters. We use a smaller $\sigma = \frac{2\sqrt{2}}{5}$ to improve training speed. We evaluate the performance of the methods using the objective functional J defined in eq. (2.12) at the control obtained from the feedback form via the respective value function approximations. For this experiment, we use a GPU to train and the results of this comparison are shown in fig. 4.7.

To reduce the effect of the Brownian motion, we run the experiments for each method on the same problem five times and plot the average values corresponding to training iterations. Furthermore, since the primary goal for this example is to explore the difference between sampling strategies, we select much higher weights for the control objective such that we have faster initial convergence for the control variable.

As can be seen in fig. 4.7 (left), our method not only yields faster initial conver-

gence but also achieves a considerably lower control objective. This indicates that the controls obtained from our approach are more effective, that is, they are closer to optimal. It is also worth pointing out that due to the high terminal cost we assigned when designing the problem, it takes very few iterations to locate the correct state-time region that the optimal solution resides in. Since FBSNNs use a Brownian motion with no drift, the sampling is unlikely to discover the target. Consequently, the generated trajectories in fig. 4.7 (right) from our method approximately reach the target, while the trajectories obtained from the FBSNN method stay closer to the initial state. Do note our primary goal for the added experiment is to highlight the importance of PMP dynamics in exploring the solution space, additional hyperparameter tuning and training will be needed if one aims to solve the underlying HJB equation accurately as well.

4.3.4 12D Quadcopter Path Finding Problem with Nonlinear Dynamics

In this section we introduce a quadcopter path finding problem with stochastic and nonlinear dynamics, a similar deterministic version of the problem is also used in [64, 78]. Our goal for adding the example is to test our proposed method's ability to deal with nonlinear dynamics. The problem has a state dimension of 12 and 4 control variables. We choose values $\boldsymbol{x} = [-1.5, -1.5, -1.5, 0, \ldots, 0]^\top \in \mathbb{R}^{12}$ and $\boldsymbol{x}_{\text{target}} = [2, 2, 2, 0, \ldots, 0]^\top \in \mathbb{R}^{12}$, we assume the initial states follows a Gaussian distribution centered at $\boldsymbol{x}$. Given the state variable $\boldsymbol{z} = [z_1, z_2, z_3, z_4, z_5, z_6, z_7, z_8, z_9, z_{10}, z_{11}, z_{12}]^\top$

the dynamics read

$$f(s, \boldsymbol{z}, \boldsymbol{u}) = \begin{cases} z_7 \\ z_8 \\ z_9 \\ z_{10} \\ z_{11} \\ z_{12} \\ \dfrac{u_1}{m} f_7(z_4, z_5, z_6) = \dfrac{u_1}{m}(\sin(z_4)\sin(z_6) + \cos(z_4)\sin(z_5)\cos(z_6)) \\ \dfrac{u_1}{m} f_8(z_4, z_5, z_6) = \dfrac{u_1}{m}(-\cos(z_4)\sin(z_6) + \sin(z_4)\sin(z_5)\cos(z_6)) \\ \dfrac{u_1}{m} f_9(z_5, z_6) - g = \dfrac{u_1}{m}(\cos(z_5)\cos(z_6)) - g \\ u_2 \\ u_3 \\ u_4 \end{cases}$$

Here $z_1, z_2, \ldots, z_6$ represents the spacial and angular position of the quadcopter [40]. The controls of the problem are $\boldsymbol{u} = [u_1, u_2, u_3, u_4]^\top \in \mathbb{R}^4$. We assume that both the mass of the quadcopter $m = 1$ and gravity $g = 9.81$ are given and remain constant. We select the diffusion coefficient $\sigma = 0.2$ for the problem. The control objective encompasses the energy term

$$L(\boldsymbol{u}(s, \boldsymbol{z})) = 2 + \|\boldsymbol{u}(s, \boldsymbol{z})\|^2 = 2 + u_1^2 + u_2^2 + u_3^2 + u_4^2,$$

and the terminal cost

$$G(\boldsymbol{z}(T)) = 2500 \cdot \|\boldsymbol{z}(T) - \boldsymbol{x}_{\text{target}}\|^2.$$

The Hamiltonian of the system has the form

$$\begin{aligned}\mathcal{H}(s,\boldsymbol{z},\boldsymbol{p},\boldsymbol{M},\boldsymbol{u}) &= \frac{1}{2}\mathrm{tr}\left(\sigma\boldsymbol{M}\right) + \boldsymbol{p}\cdot f(s,\boldsymbol{z},\boldsymbol{u}) - L(s,\boldsymbol{z},\boldsymbol{u}) \\ &= \frac{1}{2}\mathrm{tr}\left(\sigma\boldsymbol{M}\right) + p_1 z_7 + p_2 z_8 + p_3 z_9 + p_4 z_{10} + p_5 z_{11} + p_6 z_{12} + p_7\frac{u_1}{m}f_7 \\ &+ p_8\frac{u_1}{m}f_8 + p_9\frac{u_1}{m}f_9 - p_9 g + p_{10}u_2 + p_{11}u_3 + p_{12}u_4\end{aligned}$$

given the adjoint variable $\boldsymbol{p}$ and $\boldsymbol{M}$. By taking the first order optimality condition of the generalized Hamiltonian and using the results from theorem 2.4.2, we can derive the feedback form of the controls $\boldsymbol{u}$ in terms of the value function $\Phi(\cdot)$ as

$$u_1 = \frac{-1}{2m}\left(f_7\frac{\partial\Phi}{\partial z_7} + f_8\frac{\partial\Phi}{\partial z_8} + f_9\frac{\partial\Phi}{\partial z_9}\right),$$
$$u_2 = -\frac{1}{2}\frac{\partial\Phi}{\partial z_{10}},\quad u_3 = -\frac{1}{2}\frac{\partial\Phi}{\partial z_{11}},\quad u_4 = -\frac{1}{2}\frac{\partial\Phi}{\partial z_{12}}.$$

The HJB equation and BSDE can be derived using the feedback form accordingly under section 2.4.3.

Neural Network Approach for the SOC Problem

We test our proposed approach on the SOC quadcopter problem, for our model we use the network architecture in eq. (4.1) featuring two layers and 128 neurons per layer for the ResNet, we also enable the quadratic terms as described in eq. (4.1). For training we select penalty terms $\beta = (0.1, 0.1, 1.0, 0.1, 0.1)$. We train a total of 6000 iterations using Adam optimizer with a batch size of 128. For learning rate scheduling we start with a learning rate of 0.01 and have it halved every 1600 iterations. Since the dynamics in this example are more complex, we discretize the SDE with 100 equidistant steps between $t = 0$ and $T = 1$ to enhance accuracy. By doing so, on average, every training iteration took around 2 seconds on the GPU.

We visualize the training results by showing some flight paths starting from randomly generated initial states following the learned policy. In fig. 4.8 we notice that

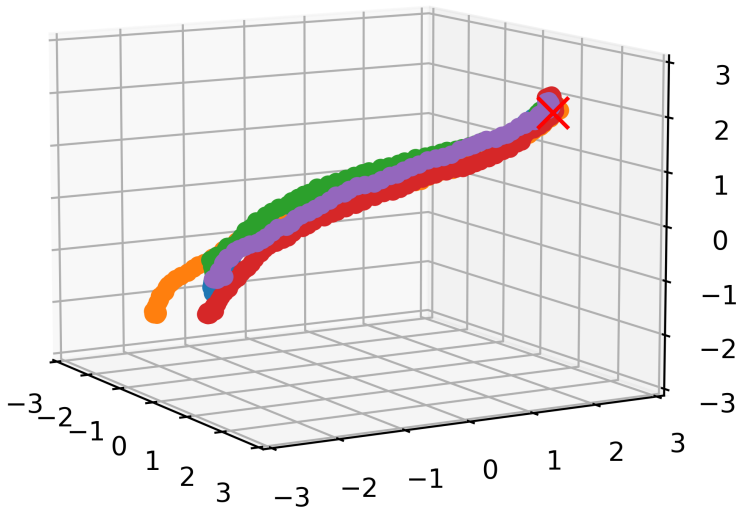


Figure 4.8: Flight path examples using the learned controller. The target is depicted by a red cross.

all trajectories converge to the target regardless of the initial states, indicating our controller is successful in solving the path finding problem.

Comparison and remarks against the Deterministic OC solution

One of the main challenges for the quadcopter problem is the lack of a reference solution, as described in section 2.4.2, SOC problems with nonlinear dynamics usually do not admit analytic solutions. Since the problem has a state dimension of 12 we are also not aware of any numerical methods for either local or global solutions.

To account for this we opt for a simple comparison against a neural network model trained on the deterministic version of the same problem. Assuming the dynamics insection 4.3.4 remain the same while the noise σ is reduced to 0. The problem then reduces to that from [78]. Notice in this case changing the problem from stochastic to deterministic does not affect the feedback form, however the Hamiltonian and the HJB equation will lose the diffusion term, the BSDE system will also no longer be necessary.

We have solved the deterministic version of the problem in [78] where the ex-

J at $\boldsymbol{z}_0$	evaluated at $\sigma = 0.2$	evaluated at $\sigma = 0$
Deterministic Model	9.33×10^3	2.18×10^3
Our Model	3.34×10^3	-
Accurate J (deterministic)	-	2.18×10^3

Table 4.4: Approximated control objective J for initial state $\boldsymbol{z}_0 = [-1.5, -1.5, -1.5, 0, \ldots, 0]^\top$. Note the deterministic solution is trained with $\sigma = 0$ while ours with $\sigma = 0.2$. Value for the accurate solution comes from [78].

act same neural network is used, the penalty parameters used for the deterministic problem reads $\beta = (0, 0.1, 1.0, 0, 0)$. Here only the HJB loss is used and the training primarily focuses on minimizing the control cost. The trained model has an accurate approximation of J given a fixed initial state.

Having both models trained, we can then compare their performance on the SOC problem. Fixing an initial state of $\boldsymbol{x} = [-1.5, -1.5, -1.5, 0, \ldots, 0]$, for each policy, we compute the average value of the control objective over 15,000 randomly chosen trajectories, each using 200 time steps and report the results in table 4.4. As one would expect, while the deterministic model approximates the true solution well for $\sigma = 0$, its performance drops notably when the objective is evaluated with $\sigma = 0.2$, falling behind our SOC model in performance.

We use this experiment to emphasize the following point, that taking the stochasticity of the dynamics into account is crucial when handling noisy dynamics. The deterministic solution, though accurate without any diffusion, is not robust enough when injected with noise or consistent disturbance.

Having experimented with several examples in both stochastic and deterministic OC problems. Some remarks can be made regarding training a neural network model. For applications that require high accuracy of the Value function or solution to the HJB equation, additional weights to $\beta_1, \beta_2, \beta_4, \beta_5$ are needed as well as longer training time and iterations since optimizing the value function to high accuracy s generally a hard task. On the other hand for applications where only the controls are needed, such as most examples presented in [78], focusing on the control loss J can lead to

very fast convergence to relatively accurate results. This is the motivation behind our choice of hyperparameters in [78].

4.4 Summary

We propose a neural network approach for approximately solving Hamilton-Jacobi-Bellman PDEs arising in high-dimensional stochastic optimal control. Similar to existing approaches [91, 44], we parameterize the value function with a neural network and experiment with different losses to train the network weights. One of the main differences that set our work apart from these works is the use of feedback form given by the stochastic Pontryagin maximum principle to design the forward SDE used to explore the state space during training. We also differentiate our method from similar ideas by including the control objective in the learning problem, by solving the SOC problem directly alongside the HJB equation we can more efficiently explore the state space and gain faster convergence of the model.

Using an intuitive two-dimensional test problem, we visualize that the improved sampling strategy allows us to effectively learn the value function and determine the relevant regions of the state space; see section 4.3.2. We further demonstrate this point through a modified version of the 100-dimensional test problem. Theoretically, our proposed forward SDE compared to purely random exploration is that it coincides with the characteristic curves of the HJB equation as the stochasticity of the system is reduced. Therefore, our work can be seen as an extension of the neural network approaches for deterministic control problems in [78].

Our choice of loss function allows us to gain much faster convergence while maintaining accuracy compared to results in [91, 44], as we showed through the benchmark example in section 4.3.3. Using a 12-dimensional quadcopter example whose dynamic is nonlinear in the states, we also demonstrate that our model can handle complicated

dynamics; see section 4.3.4.

Compared to neural network approaches for semi-linear elliptic/parabolic PDEs such as [91, 44] it is important to highlight that our approach is limited to HJB equations arising in stochastic optimal control. Since our forward SDE is derived from optimality principles, with the lack of similar property, extending it to other high-dimensional PDEs (for example, Black Scholes [15] and Allen Cahan [98, 12] equations) is not obvious and may be impossible.

Chapter 5

Conclusions and Future Work

In this dissertation, we present some results and findings on deep learning based algorithms for solving different PDEs. We divided our work into two parts, in the first half we investigate general PDEs with dimensions smaller than or equal to three and focus particularly on the popular PINNs algorithm. Through experimentation, we find that by using a polynomial based function approximator and more accurate numerical integration scheme, one can maintain the optimization structure of a machine learning method and achieve as good if not much better accuracy than PINNs in a variety of tested examples, while avoiding some of the common difficulties when applying PINNs. In the second half of our work, we focus primarily on stochastic optimal control problems and their corresponding HJB equations. By utilizing the underlying control theory we propose a deep learning based method that can handle problems in high dimensions with relatively high efficiency.

There remains abundant space for future work in the field regarding both topics. For problems with dimensions smaller than or equal to 3, we believe there is much that can be done to follow up our work. First of all, additional numerical experiments are valuable to further understand and improve the current method, especially on problems with complex domain or non-homogeneous PDEs. Finding the appropriate

numerical solutions to such problems and comparing them against PINNs would yield meaningful results and conclusions. Finding other ways to leverage the polynomial structures of our spline model to improve training efficiency and accuracy is another topic worth paying attention to. Considering the non-convexity of many problems in the field finding the appropriate means to implement stochastic optimization methods will also help reduce memory cost and potentially improve convergence results and model accuracy.

It is worth mentioning that despite several major differences, our proposed optimization method shares many similarities with the least square finite element method (see [16]) often used in PDE applications. We believe a deeper look into the differences between our approach and LSFEM could result in ideas that can further improve our method, as well as bring more insights into the general topic. In fact, we think such discussion could add benefits to the development of PINNs as well.

For high dimensional HJB equations and (stochastic) optimal control problems, one interesting topic often discussed in the area relates to problems with viscous solutions, additional consideration and choices of networks may be necessary for accurately and efficiently learning discontinuous solutions. Another possible future direction will be to focus on problems with infinite time horizons, which yield stationary HJB equations with relevant theory. Some early work has been proposed in [56, 20] for problems in low dimensions, while for more challenging high dimensional problems results are still lacking.

Bibliography

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

[2] Abien Fred Agarap. Deep learning using rectified linear units (relu), 2019.

[3] Akram Aldroubi, Murray Eden, and Michael Unser. Discrete spline filters for multiresolutions and wavelets of l_2. *SIAM Journal on Mathematical Analysis*, 25(5):1412–1432, 1994.

[4] Samuel Miller Allen and John W Cahn. Ground state structures in ordered binary alloys with second neighbor interactions. *Acta Metallurgica*, 20(3):423–433, 1972.

[5] Martin S. Alnaes, Jan Blechta, Johan Hake, August Johansson, Benjamin

Kehlet, Anders Logg, Chris N. Richardson, Johannes Ring, Marie E. Rognes, and Garth N. Wells. The FEniCS project version 1.5. *Archive of Numerical Software*, 3, 2015. doi: 10.11588/ans.2015.100.20553.

[6] William F Ames. *Nonlinear partial differential equations in engineering*. Academic press, 1965.

[7] Fabio Antonelli. Backward-forward stochastic differential equations. *Ann. Appl. Probab.*, 3(3):777–793, 1993. ISSN 1050-5164. URL `http://links.jstor.org/sici?sici=1050-5164(199308)3:3<777:BSDE>2.0.CO;2-5&origin=MSN`.

[8] Santiago Badia, Wei Li, and Alberto F. Martín. Finite element interpolated neural networks for solving forward and inverse problems, 2023.

[9] Reza Akbarian Bafghi and Maziar Raissi. Pinns-tf2: Fast and user-friendly physics-informed neural networks in tensorflow v2. 2023. URL `https://api.semanticscholar.org/CorpusID:265043331`.

[10] Zachary Battles and Lloyd N Trefethen. An extension of matlab to continuous functions and operators. *SIAM Journal on Scientific Computing*, 25(5):1743–1770, 2004.

[11] Atilim Gunes Baydin, Barak A. Pearlmutter, Alexey Andreyevich Radul, and Jeffrey Mark Siskind. Automatic differentiation in machine learning: a survey, 2018.

[12] Nils Berglund. An introduction to singular stochastic pdes: Allen-cahn equations, metastability and regularity structures, 2019.

[13] D.P. Bertsekas and J.N. Tsitsiklis. Neuro-dynamic programming: an overview. In *Proceedings of 1995 34th IEEE Conference on Decision and Control*, volume 1, pages 560–564 vol.1, 1995. doi: 10.1109/CDC.1995.478953.

[14] Leonhard Bittner. L. s. pontryagin, v. g. boltyanskii, r. v. gamkrelidze, e. f. mishechenko, the mathematical theory of optimal processes. viii + 360 s. new york/london 1962. john wiley & sons. preis 90/–. *Zamm-zeitschrift Fur Angewandte Mathematik Und Mechanik*, 43:514–515, 1963. URL `https://api.semanticscholar.org/CorpusID:122390952`.

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

[16] Pavel B Bochev and Max D Gunzburger. *Least-squares finite element methods*, volume 166. Springer Science & Business Media, 2009.

[17] Franck Boyer and Pierre Fabrie. *Mathematical Tools for the Study of the Incompressible Navier-Stokes Equations andRelated Models*, volume 183. Springer Science & Business Media, 2012.

[18] James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. JAX: composable transformations of Python+NumPy programs, 2018. URL `http://github.com/google/jax`.

[19] Fabio Camilli and Maurizio Falcone. An approximation scheme for the optimal control of diffusion processes. *M2AN - Modélisation mathématique et analyse numérique*, 29(1):97–122, 1995. URL `http://www.numdam.org/item/M2AN_1995__29_1_97_0/`.

[20] Eduardo Casas and Karl Kunisch. Infinite horizon optimal control for a general class of semilinear parabolic equations. *Applied Mathematics & Optimization*, 88(2):47, 2023.

[21] Jean-François Chassagneux and Adrien Richou. Numerical simulation of quadratic bsdes. *The Annals of Applied Probability*, 26(1), February 2016.

ISSN 1050-5164. doi: 10.1214/14-aap1090. URL http://dx.doi.org/10.1214/14-AAP1090.

[22] Patrick Cheridito, H. Mete Soner, Nizar Touzi, and Nicolas Victoir. Second-order backward stochastic differential equations and fully nonlinear parabolic PDEs. *Comm. Pure Appl. Math.*, 60(7):1081–1110, 2007. ISSN 0010-3640. doi: 10.1002/cpa.20168. URL https://doi.org/10.1002/cpa.20168.

[23] Pi-Yueh Chuang and Lorena A. Barba. Experience report of physics-informed neural networks in fluid simulations: pitfalls and frustration, 2022.

[24] Pi-Yueh Chuang and Lorena A. Barba. Predictive limitations of physics-informed neural networks in vortex shedding, 2023.

[25] Pi-Yueh Chuang, Olivier Mesnard, Anush Krishnan, and Lorena A. Barba. PetIBM: toolbox and applications of the immersed-boundary method on distributed-memory architectures. *The Journal of Open Source Software*, 3(25): 558, may 2018. doi: 10.21105/joss.00558. URL https://doi.org/10.21105/joss.00558.

[26] Peter Constantin and Ciprian Foiaş. *Navier-stokes equations*. University of Chicago press, 1988.

[27] Richard Courant, Kurt Friedrichs, and Hans Lewy. On the partial difference equations of mathematical physics. *IBM journal of Research and Development*, 11(2):215–234, 1967.

[28] Salvatore Cuomo, Vincenzo Schiano di Cola, Fabio Giampaolo, Gianluigi Rozza, Maziar Raissi, and Francesco Piccialli. Scientific machine learning through physics-informed neural networks: Where we are and what's next, 2022.

[29] Ameya D. Jagtap and George Em Karniadakis. Extended physics-informed neural networks (xpinns): A generalized space-time domain decomposition based deep learning framework for nonlinear partial differential equations. *Communications in Computational Physics*, 28(5):2002–2041, 2020. ISSN 1991-7120. doi: https://doi.org/10.4208/cicp.OA-2020-0164. URL `http://global-sci.org/intro/article_detail/cicp/18403.html`.

[30] L. Debnath. *Nonlinear Partial Differential Equations for Scientists and Engineers*. Birkhäuser Boston, 2011. ISBN 9780817682651. URL `https://books.google.com/books?id=Ir4yXgBesAsC`.

[31] Hongjie Dong and Nicolai V. Krylov. The rate of convergence of finite-difference approximations for parabolic Bellman equations with Lipschitz coefficients in cylindrical domains. *Appl. Math. Optim.*, 56(1):37–66, 2007. ISSN 0095-4616. doi: 10.1007/s00245-007-0879-4. URL `https://doi.org/10.1007/s00245-007-0879-4`.

[32] T. A Driscoll, N. Hale, and L. N. Trefethen. *Chebfun Guide*. Pafnuty Publications, 2014. URL `http://www.chebfun.org/docs/guide/`.

[33] Franz Durst, D Miloievic, and Bernhard Schönung. Eulerian and lagrangian predictions of particulate two-phase flows: a numerical study. *Applied Mathematical Modelling*, 8(2):101–115, 1984.

[34] Weinan E and Bing Yu. The deep ritz method: A deep learning-based numerical algorithm for solving variational problems, 2017.

[35] Weinan E, Jiequn Han, and Arnulf Jentzen. Deep learning-based numerical methods for high-dimensional parabolic partial differential equations and backward stochastic differential equations. *Commun. Math. Stat.*, 5(4):349–

380, 2017. ISSN 2194-6701. doi: 10.1007/s40304-017-0117-6. URL https://doi.org/10.1007/s40304-017-0117-6.

[36] L.C. Evans. *Partial Differential Equations*. Graduate studies in mathematics. American Mathematical Society, 2010. ISBN 9780821849743. URL https://books.google.com/books?id=Xnu0o_EJrCQC.

[37] Robert Eymard, Thierry Gallouët, and Raphaèle Herbin. Finite volume methods. *Handbook of numerical analysis*, 7:713–1018, 2000.

[38] W.H. Fleming and H.M. Soner. *Controlled Markov Processes and Viscosity Solutions*. Stochastic Modelling and Applied Probability. Springer New York, 2006. ISBN 9780387310718. URL https://books.google.com/books?id=4Bjz2iWmLyQC.

[39] Peter Gangl, Kevin Sturm, Michael Neunteufel, and Joachim Schöberl. Fully and semi-automated shape differentiation in ngsolve, 2020.

[40] Luis Rodolfo Garcia Carrillo, Alejandro Dzul, R. Lozano, and Claude Pégard. *Quad Rotorcraft Control. Vision-Based Hovering and Navigation*. 01 2012.

[41] Paola Gervasio, Fausto Saleri, and Alessandro Veneziani. Algebraic fractional-step schemes with spectral methods for the incompressible navier–stokes equations. *Journal of Computational Physics*, 214(1):347–365, 2006.

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

[43] Tamara G. Grossmann, Urszula Julia Komorowska, Jonas Latz, and Carola-Bibiane Schönlieb. Can physics-informed neural networks beat the finite element method?, 2023.

[44] Jiequn Han, Arnulf Jentzen, and Weinan E. Solving high-dimensional partial differential equations using deep learning. *Proc. Natl. Acad. Sci. USA*, 115 (34):8505–8510, 2018. ISSN 0027-8424. doi: 10.1073/pnas.1718942115. URL https://doi.org/10.1073/pnas.1718942115.

[45] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition, 2015.

[46] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2016.

[47] Zheyuan Hu, Ameya D. Jagtap, George Em Karniadakis, and Kenji Kawaguchi. When do extended physics-informed neural networks (XPINNs) improve generalization? *SIAM Journal on Scientific Computing*, 44(5):A3158–A3182, sep 2022. doi: 10.1137/21m1447039. URL https://doi.org/10.1137%2F21m1447039.

[48] Yunona Iwasaki and Ching-Yao Lai. Clustering behavior of physics-informed neural networks: Inverse modeling of an idealized ice shelf.

[49] Ameya D. Jagtap, Ehsan Kharazmi, and George Em Karniadakis. Conservative physics-informed neural networks on discrete domains for conservation laws: Applications to forward and inverse problems. *Computer Methods in Applied Mechanics and Engineering*, 365:113028, 2020. ISSN 0045-7825. doi: https://doi.org/10.1016/j.cma.2020.113028. URL https://www.sciencedirect.com/science/article/pii/S0045782520302127.

[50] Kelvin Kan, James G. Nagy, and Lars Ruthotto. Lsemink: A modified newton-krylov method for log-sum-exp minimization, 2023.

[51] E. Kharazmi, Z. Zhang, and G. E. Karniadakis. Variational physics-informed neural networks for solving partial differential equations, 2019.

[52] Ehsan Kharazmi, Zhongqiang Zhang, and George E.M. Karniadakis. hp-vpinns: Variational physics-informed neural networks with domain decomposition. *Computer Methods in Applied Mechanics and Engineering*, 374:113547, February 2021. ISSN 0045-7825. doi: 10.1016/j.cma.2020.113547. URL `http://dx.doi.org/10.1016/j.cma.2020.113547`.

[53] Sujin Kim, Raghu Pasupathy, and Shane G Henderson. A guide to sample average approximation. *Handbook of simulation optimization*, pages 207–243, 2015.

[54] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[55] Aditi S. Krishnapriyan, Amir Gholami, Shandian Zhe, Robert M. Kirby, and Michael W. Mahoney. Characterizing possible failure modes in physics-informed neural networks, 2021.

[56] Karl Kunisch and Donato Vásquez-Varas. Smooth approximation of feedback laws for infinite horizon control problems with non-smooth value functions. *arXiv preprint arXiv:2312.11981*, 2023.

[57] Harold J. Kushner. Numerical methods for stochastic control problems in continuous time. *SIAM J. Control Optim.*, 28(5):999–1048, 1990. ISSN 0363-0129. doi: 10.1137/0328056. URL `https://doi.org/10.1137/0328056`.

[58] Harold J. Kushner and Paul Dupuis. *Numerical methods for stochastic control problems in continuous time*, volume 24 of *Applications of Mathematics (New York)*. Springer-Verlag, New York, second edition, 2001. ISBN 0-387-

95139-3. doi: 10.1007/978-1-4613-0007-6. URL https://doi.org/10.1007/978-1-4613-0007-6. Stochastic Modelling and Applied Probability.

[59] Alvin CK Lai and FZ Chen. Comparison of a new eulerian model with a modified lagrangian approach for particle distribution and deposition indoors. *Atmospheric Environment*, 41(25):5249–5256, 2007.

[60] Hans Petter Langtangen and Anders Logg. *Solving PDEs in Python*. Springer, 2017. ISBN 978-3-319-52461-0. doi: 10.1007/978-3-319-52462-7.

[61] N. Lehtomaki, N. Sandell, and M. Athans. Robustness results in linear-quadratic gaussian based multivariable control designs. *IEEE Transactions on Automatic Control*, 26(1):75–93, 1981. doi: 10.1109/TAC.1981.1102565.

[62] Randall J LeVeque. Finite difference methods for differential equations. *Draft version for use in AMath*, 585(6):112, 1998.

[63] Xingjian Li, Deepanshu Verma, and Lars Ruthotto. A neural network approach for stochastic optimal control. *arXiv preprint arXiv:2209.13104*, 2022.

[64] Alex Tong Lin, Yat Tin Chow, and Stanley Osher. A splitting method for overcoming the curse of dimensionality in hamilton-jacobi equations arising from nonlinear optimal control and differential games with applications to trajectory generation, 2018.

[65] Joseph W Lockwood, Ning Lin, Michael Oppenheimer, and Ching-Yao Lai. Using neural networks to predict hurricane storm surge and to assess the sensitivity of surge to storm characteristics. *Journal of Geophysical Research: Atmospheres*, 127(24):e2022JD037617, 2022.

[66] Anders Logg, Kent-Andre Mardal, Garth N. Wells, et al. *Automated Solution*

of Differential Equations by the Finite Element Method. Springer, 2012. doi: 10.1007/978-3-642-23099-8.

[67] Wei-Liem Loh. On latin hypercube sampling. *The annals of statistics*, 24(5): 2058–2080, 1996.

[68] Björn Lütjens, Catherine H. Crawford, Mark Veillette, and Dava Newman. Pce-pinns: Physics-informed neural networks for uncertainty propagation in ocean modeling, 2021.

[69] Jin Ma, Philip Protter, and Jiong Min Yong. Solving forward-backward stochastic differential equations explicitly—a four step scheme. *Probab. Theory Related Fields*, 98(3):339–359, 1994. ISSN 0178-8051. doi: 10.1007/BF01192258. URL `https://doi.org/10.1007/BF01192258`.

[70] Zhiping Mao, Ameya Dilip Jagtap, and George Em Karniadakis. Physics-informed neural networks for high-speed flows. *Computer Methods in Applied Mechanics and Engineering*, 2020. URL `https://api.semanticscholar.org/CorpusID:212755458`.

[71] Charles C. Margossian. A review of automatic differentiation and its efficient implementation. *WIREs Data Mining and Knowledge Discovery*, 9(4), mar 2019. doi: 10.1002/widm.1305. URL `https://doi.org/10.1002%2Fwidm.1305`.

[72] Siddhartha Mishra and Roberto Molinaro. Estimates on the generalization error of physics informed neural networks (pinns) for approximating a class of inverse problems for pdes, 2023.

[73] J. Modersitzki. *FAIR: Flexible Algorithms for Image Registration*. SIAM, Philadelphia, 2009.

[74] Ben Moseley, Andrew Markham, and Tarje Nissen-Meyer. Finite basis physics-informed neural networks (fbpinns): a scalable domain decomposition approach for solving differential equations, 2021.

[75] Elizabeth Newman and Lars Ruthotto. 'hessquik': Fast hessian computation of composite functions. *Journal of Open Source Software*, 7(72):4171, 2022. doi: 10.21105/joss.04171. URL `https://doi.org/10.21105/joss.04171`.

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

[77] Derek Onken, Samy Wu Fung, Xingjian Li, and Lars Ruthotto. Ot-flow: Fast and accurate continuous normalizing flows via optimal transport, 2021.

[78] Derek Onken, Levon Nurbekyan, Xingjian Li, Samy Wu Fung, Stanley Osher, and Lars Ruthotto. A neural network approach for high-dimensional optimal control, 2021.

[79] Derek Onken, Levon Nurbekyan, Xingjian Li, Samy Wu Fung, Stanley Osher, and Lars Ruthotto. A neural network approach applied to multi-agent optimal control. In *2021 European Control Conference (ECC)*, pages 1036–1041. IEEE, 2021.

[80] Derek Onken, Levon Nurbekyan, Xingjian Li, Samy Wu Fung, Stanley Osher, and Lars Ruthotto. A neural network approach for high-dimensional optimal control applied to multiagent path finding. *IEEE Transactions on Control Systems Technology*, 31(1):235–251, 2022.

[81] N.S. Ottosen and H. Petersson. *Introduction to the Finite Element Method*. Prentice Hall, 1992. ISBN 9780134738772. URL `https://books.google.com/books?id=_5FRAAAAMAAJ`.

[82] Etienne Pardoux and Shanjian Tang. Forward-backward stochastic differential equations and quasilinear parabolic PDEs. *Probab. Theory Related Fields*, 114 (2):123–150, 1999. ISSN 0178-8051. doi: 10.1007/s004409970001. URL `https://doi.org/10.1007/s004409970001`.

[83] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library, 2019.

[84] Huyên Pham. *Continuous-time stochastic control and optimization with financial applications*, volume 61 of *Stochastic Modelling and Applied Probability*. Springer-Verlag, Berlin, 2009. ISBN 978-3-540-89499-5. doi: 10.1007/978-3-540-89500-8. URL `https://doi.org/10.1007/978-3-540-89500-8`.

[85] Les Piegl and Wayne Tiller. *The NURBS book*. Springer Science & Business Media, 2012.

[86] L. S. Pontryagin, V. G. Boltyanskii, R. V. Gamkrelidze, and E. F. Mishchenko. *The Mathematical Theory of Optimal Processes*. Translated by K. N. Trirogoff; edited by L. W. Neustadt. Interscience Publishers John Wiley & Sons, Inc. New York-London, 1962.

[87] Marius-Constantin Popescu, Valentina E Balas, Liliana Perescu-Popescu, and Nikos Mastorakis. Multilayer perceptron and neural networks. *WSEAS Transactions on Circuits and Systems*, 8(7):579–588, 2009.

[88] Warren B. Powell. *Approximate dynamic programming*. Wiley Series in Probability and Statistics. Wiley-Interscience [John Wiley & Sons], Hoboken, NJ,

2007. ISBN 978-0-470-17155-4. doi: 10.1002/9780470182963. URL https://doi.org/10.1002/9780470182963. Solving the curses of dimensionality.

[89] Luigi Quartapelle. *Numerical solution of the incompressible Navier-Stokes equations*, volume 113. Birkhäuser, 2013.

[90] Nasim Rahaman, Aristide Baratin, Devansh Arpit, Felix Draxler, Min Lin, Fred A. Hamprecht, Yoshua Bengio, and Aaron Courville. On the spectral bias of neural networks, 2019.

[91] Maziar Raissi. Forward-backward stochastic neural networks: Deep learning of high-dimensional partial differential equations. *arXiv preprint arXiv:1804.07010*, 2018.

[92] Maziar Raissi, Paris Perdikaris, and George Em Karniadakis. Physics informed deep learning (part i): Data-driven solutions of nonlinear partial differential equations. *arXiv preprint arXiv:1711.10561*, 2017.

[93] Maziar Raissi, Paris Perdikaris, and George Em Karniadakis. Physics informed deep learning (part ii): Data-driven discovery of nonlinear partial differential equations. *arXiv preprint arXiv:1711.10566*, 2017.

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

[95] Singiresu S Rao. *The finite element method in engineering*. Butterworth-heinemann, 2017.

[96] Sebastian Ruder. An overview of gradient descent optimization algorithms.

ArXiv, abs/1609.04747, 2016. URL https://api.semanticscholar.org/CorpusID:17485266.

[97] Tim De Ryck, Ameya D. Jagtap, and Siddhartha Mishra. Error estimates for physics informed neural networks approximating the navier-stokes equations, 2023.

[98] Jie Shen and Xiaofeng Yang. Numerical approximations of allen-cahn and cahn-hilliard equations. *Discrete Contin. Dyn. Syst*, 28(4):1669–1691, 2010.

[99] Yeonjong Shin. On the convergence of physics informed neural networks for linear second-order elliptic and parabolic type pdes. *Communications in Computational Physics*, 28(5):2042–2074, June 2020. ISSN 1991-7120. doi: 10.4208/cicp.oa-2020-0193. URL http://dx.doi.org/10.4208/cicp.OA-2020-0193.

[100] Justin Sirignano and Konstantinos Spiliopoulos. Dgm: A deep learning algorithm for solving partial differential equations. *Journal of Computational Physics*, 375:1339–1364, December 2018. ISSN 0021-9991. doi: 10.1016/j.jcp.2018.08.029. URL http://dx.doi.org/10.1016/j.jcp.2018.08.029.

[101] Jonthan D Smith, Zachary E Ross, Kamyar Azizzadenesheli, and Jack B Muir. Hyposvi: Hypocentre inversion with stein variational inference and physics informed neural networks. *Geophysical Journal International*, 228(1):698–710, August 2021. ISSN 1365-246X. doi: 10.1093/gji/ggab309. URL http://dx.doi.org/10.1093/gji/ggab309.

[102] Michael L. Stein. Large sample properties of simulations using latin hypercube sampling. *Technometrics*, 29:143–151, 1987. URL https://api.semanticscholar.org/CorpusID:121392444.

[103] P Thévenaz, T Blu, and M Unser. Image interpolation and resampling, handbook of medical image processing, 2003.

[104] Nils Wandel, Michael Weinmann, Michael Neidlin, and Reinhard Klein. Spline-pinn: Approaching pdes without data using fast, physics-informed hermite-spline cnns, 2022.

[105] J. Wang and P. A. Forsyth. Maximal use of central differencing for Hamilton-Jacobi-Bellman PDEs in finance. *SIAM J. Numer. Anal.*, 46(3):1580–1601, 2008. ISSN 0036-1429. doi: 10.1137/060675186. URL https://doi.org/10.1137/060675186.

[106] Yongji Wang, Ching-Yao Lai, Javier Gómez-Serrano, and Tristan Buckmaster. Self-similar blow-up profile for the boussinesq equations via a physics-informed neural network. *arXiv preprint arXiv:2201.06780*, 2022.

[107] Yongji Wang, Ching-Yao Lai, Javier Gómez-Serrano, and Tristan Buckmaster. Asymptotic self-similar blow-up profile for three-dimensional axisymmetric euler equations using neural networks, 2023.

[108] Tien-Tsin Wong, Wai-Shing Luk, and Pheng-Ann Heng. Sampling with hammersley and halton points. *Journal of graphics tools*, 2(2):9–24, 1997.

[109] Chenxi Wu, Min Zhu, Qinyang Tan, Yadhu Kartha, and Lu Lu. A comprehensive study of non-adaptive and residual-based adaptive sampling for physics-informed neural networks. *Computer Methods in Applied Mechanics and Engineering*, 403:115671, January 2023. ISSN 0045-7825. doi: 10.1016/j.cma.2022.115671. URL http://dx.doi.org/10.1016/j.cma.2022.115671.

[110] Bing Xu, Naiyan Wang, Tianqi Chen, and Mu Li. Empirical evaluation of rectified activations in convolutional network, 2015.

[111] Liu Yang, Xuhui Meng, and George Em Karniadakis. B-pinns: Bayesian physics-informed neural networks for forward and inverse pde problems with

noisy data. *Journal of Computational Physics*, 425:109913, January 2021. ISSN 0021-9991. doi: 10.1016/j.jcp.2020.109913. URL http://dx.doi.org/10.1016/j.jcp.2020.109913.

[112] Jiongmin Yong and Xun Yu Zhou. *Stochastic controls*, volume 43 of *Applications of Mathematics (New York)*. Springer-Verlag, New York, 1999. ISBN 0-387-98723-1. doi: 10.1007/978-1-4612-1466-3. URL https://doi.org/10.1007/978-1-4612-1466-3. Hamiltonian systems and HJB equations.

[113] Qi Zeng, Yash Kothari, Spencer H. Bryngelson, and Florian Schäfer. Competitive physics informed networks, 2022.

[114] Zhao Zhang and Qingyan Chen. Comparison of the eulerian and lagrangian methods for predicting particle transport in enclosed spaces. *Atmospheric environment*, 41(25):5236–5248, 2007.

[115] Olek C Zienkiewicz, Robert L Taylor, and Jian Z Zhu. *The finite element method: its basis and fundamentals*. Elsevier, 2005.