UNIVERSITY OF CALIFORNIA
Santa Barbara

Fast Physics-Informed Neural Networks on Edge Devices

A Thesis submitted in partial satisfaction
of the requirements for the degree of

Master of Science

in

Electrical and Computer Engineering

by

Zhixiong Chen

Committee in Charge:

Professor Zheng Zhang, Chair

Professor Haewon Jeong

Professor Li-C Wang

June 2024

The Thesis of Zhixiong Chen is approved:

Professor Haewon Jeong

Professor Li-C Wang

Professor Zheng Zhang, Committee Chair

June 2024

Fast Physics-Informed Neural Networks on Edge Devices

by

Zhixiong Chen

Acknowledgements

I would like to express my sincere gratitude to several individuals who have been instrumental in the completion of this project.

Firstly, I extend my heartfelt thanks to Professor Zheng Zhang, my academic advisor, for his invaluable guidance and support throughout this endeavor. His expertise and encouragement have been pivotal in shaping the direction of this project.

I am also deeply grateful to all the members in Professor Zhang's research group, especially Alvin Liu, Xinling Yu, and. Yequan Zhao, for their unwavering assistance and insightful feedback. Their contributions have greatly enriched the quality of this work.

I am truly fortunate to have had the opportunity to collaborate with such talented and supportive individuals. Their contributions have undoubtedly made a significant difference, and for that, I am profoundly grateful.

Abstract

Fast Physics-Informed Neural Networks on Edge Devices

Zhixiong Chen

Training end-to-end models for solving partial differential equations (PDEs) using deep learning methods, such as deep neural networks, demands substantial computing resources, including power supply, memory space, and advanced computing platforms. However, edge devices typically lack these resources, making such training paradigms unattainable on these devices. Transfer learning, which involves leveraging pre-trained models on one dataset to perform inference on another dataset by fine-tuning model parameters, offers a solution by extensively pre-training models on modern GPUs and requiring fewer computing resources during fine-tuning. By applying transfer learning to solve PDEs with neural networks, we address the demand for real-time response from PDE solvers in scientific and engineering problems. In this project, we propose utilizing transfer learning for Physics-Informed Neural Networks (PINNs) to address problems in reachability analysis. We first pre-train a modified PINN on standard GPUs and subsequently fine-tune the model with constrained computing resources. During fine-tuning, we compute gradients of the loss analytically to reduce dependency on existing libraries, thereby enhancing the method's generalizability across other edge devices. Through experimentation with multiple PDE examples and the reachability problem, our results demonstrate that transfer learning with limited computing resources achieves comparable accuracy levels to the end-to-end training paradigm while requiring significantly fewer computing resources.

Contents

Acknowledgements **iv**

Abstract **v**

List of Figures **ix**

1 Introduction **1**

2 Background and Related Work **5**

2.1 Differential Equations 5

2.1.1 ODEs and PDEs 5

2.1.2 Solving Differential Equations 7

2.2 Deep Neural Networks 9

2.2.1 Activation Functions 11

2.2.2 Train, validation, and testing of neural networks 15

2.2.3 Optimization 17

2.2.4 Advanced Optimization Methods 19

2.2.5 Backward-Propagation 22

2.3 Transfer Learning 24

2.4 Physics-Informed Neural Networks 27

2.4.1 Applications 31

2.4.2 Reachability in High Dimensions 33

3 Methodology **40**

3.1 Transfer Learning of PINNs 40

3.1.1 Poisson Equation 43

3.1.2 Reachability Problem 44

3.2 Finetuning Analytic Gradient Computation 45

4 Methodology 50
4.1 Transfer Learning of PINNs 50
4.1.1 Poisson Equation 53
4.1.2 Reachability Problem 54
4.2 Finetuning Analytic Gradient Computation 55

5 Numerical Results 60
5.1 Poisson Equation 60
5.2 Hamilton-Jacobi-Bellman Equation 69
5.3 3-D Reachability Problem 74

6 Discussion and Conlusion 77

List of Figures

2.1 MLP with 2 hidden layers. RAY, Pinti, and Oberai 2024 12
2.2 Different activation functions. RAY, Pinti, and Oberai 2024 . . . 14
2.3 Gradient Descent prefers flatter minimas. RAY, Pinti, and Oberai 2024 18
2.4 Computational graph. RAY, Pinti, and Oberai 2024 23
2.5 Computational graph to evaluate derivatives w.r.t network input. RAY, Pinti, and Oberai 2024 28
2.6 Structure of Physics-Informed Neural Networks 29
2.7 Structure of Transfer Learning of Physics-Informed Neural Networks. Hidden layers are fixed during finetuning and only the output layer can be modified 39

5.1 Pretraining loss values in log scale. y-axis is loss values in log scale, and x-axis represents iteration 63
5.2 Finetuning loss values in log scale. y-axis is loss values in log scale, and x-axis represents iteration 64
5.3 Predicted and ground truth solution for the Poisson equation at $k = \mathbf{5}$ 65
5.4 Error of the predicted result. 65
5.5 Predicted solution for the Poisson equation at k=5 by training a PINN from scratch. 66
5.6 Error of the predicted solution for the PINN trained from scratch. 66
5.7 Predicted and ground truth solution for the Poisson equation at $k = \mathbf{6}$ after finetuning. 67
5.8 Error of solution for the Poisson equation at $k = \mathbf{6}$ after finetuning. 67
5.9 Predicted and ground truth solution for the Poisson equation at $k = \mathbf{3.5}$ after finetuning. 68
5.10 Error of solution for the Poisson equation at $k = \mathbf{6}$ after finetuning. 68

5.11 Pretraining training loss. The pretraining loss is calculated over $\mu \in [\mathbf{1}, \mathbf{1.2}, \mathbf{1.4}, \mathbf{1.6}]$ 71

5.12 Finetuning test loss. The testing is performed for $\mu = \mathbf{1.33}$(blue line), $\mu = \mathbf{1.527}$(yellow line), and $\mu = \mathbf{1.423}$(red line). 72

5.13 Test loss for $\mu = \mathbf{1.423}$ at finetuning stage using analytical gradient computation 73

5.14 Test loss for $\mu = \mathbf{1.423}$ at finetuning stage using backward propagation 73

5.15 Ground truth solution for 3-D reachability problem. Left figure shows the solution space where the blue region denotes the reachable set. The right figure shows the overlap of this solution with the true solution. Since this is ground truth, thus it shows a fully overlap. . . . 75

5.16 Predicted solution for 3-D reachability problem (Left). Comparison of predicted solution with ground truth (Right) where shaded area represents the difference. 76

Chapter 1

Introduction

Differential equations are fundamental tools in various scientific fields, describing relationships between variables and their rates of change. In physics, differential equations are ubiquitous, modeling the behavior of systems in classical mechanics, electromagnetism, thermodynamics, and quantum mechanics. For instance, Newton's second law of motion can be expressed as a second-order ordinary differential equation. In engineering, differential equations are extensively used to model and analyze systems across domains such as electrical circuits, fluid dynamics, structural mechanics, control theory, and signal processing. The Navier-Stokes equations, for example, describe fluid flow, while the heat equation governs the distribution of heat in a material. In biology, differential equations model population dynamics, biochemical reactions, neural networks, and the spread of diseases. Examples include the Lotka-Volterra equations for predator-prey interactions and the Hodgkin-Huxley model for action potentials in neurons. In engineering, partial differential equations (PDEs) are often used to describe complex systems, making their solution a primary concern. Simple cases can sometimes be solved using analytic methods like Fourier series. However, for more complex PDEs with higher dimensions or intricate boundary conditions, numerical methods are preferred. Despite their accuracy, numerical methods can be computationally expensive for solving challenging PDEs. To address this, Physics-Informed Neural Networks (PINNs) have been proposed Raissi, Perdikaris, and George E Karniadakis 2019. Utilizing modern GPUs, PINNs show superior performance in solving high-dimensional PDEs and various areas where solving PDEs is required, such as inverse problem in Lu, Pestourie, et al. 2021, Jagtap et al. 2022, and J. Yu et al. 2022, control problems in Bansal and Tomlin 2021 and Mowlavi and Nabi 2023, and simulation tasks in Liu, X. Yu, and Z. Zhang 2022 and Oldenburg et al. 2022 facilitating further research in areas like inverse design and simulations. Details of PINNs will be discussed in the Preliminary section. However, PINNs are not without limitations. They face the curse of dimensionality due to the exponential growth of the number of forward and backward propagation when solving high-dimensional PDEs and require significant computing resources for complex problems. Additionally, their generalizability is limited. A PINN trained on specific PDE parameters must be retrained from scratch if these parameters change. This retraining requirement poses challenges in engineering scenarios requiring real-time PDE solutions on edge devices, such as microcontrollers, FPGAs, and edge GPUs, which have limited computing resources and energy constraints. Efforts have been made to develop models capable of solving different PDEs after a single training session. One such framework, DeepONet by Lu, Jin, and George Em Karniadakis 2019, aims to learn linear and nonlinear operators implicitly as neural networks. Leveraging the Universal Approximation Theorem for Operators, DeepONet represents various operators, including those from classical and fractional calculus, through neural networks trained on multi-fidelity data and heterogeneous sources of experimental and simulation data. This approach efficiently discovers new operators and predicts complex dynamics of multiscale and multiphysics operators. However, DeepONet's hardware memory requirements remain a challenge. Another alternative is applying transfer learning to PINNs. This approach involves pretraining a PINN on a family of PDEs with different parameters and fine-tuning the model for new PDEs. This method, described in Desai et al. 2021, has proven fast and effective for PDEs with a single varying parameter but is limited when multiple parameters change in real-world scenarios. Additionally, its performance on edge devices has not been tested. Therefore, this project proposes the following contributions: 1. During the fine-tuning stage, only the parameters of the last layer are optimized. We implemented an analytic method to compute the gradients, eliminating dependence on existing gradient computation libraries. 2. We extend the transfer learning method to handle more complex tasks by pretraining PINNs on PDEs with multiple varying parameters. 3. To our knowledge, this is the first work aiming to solve a real-world optimal control problem on edge devices. We successfully addressed a problem from Bansal and Tomlin 2021 on edge devices using transfer learning while maintaining accuracy.

Chapter 2

Background and Related Work

2.1 Differential Equations

2.1.1 ODEs and PDEs

Ordinary differential equations (ODEs) refer to differential equations that are dependent on only a single independent variable. The general form of an ODE is:

$$F\left(x, y, y', y'', \ldots, y^{(n)}\right) = 0 \tag{2.1}$$

where y is the unknown function of the independent variable $x; y', y'', \ldots, y^{(n)}$ represent the first, second, $\ldots, n$th derivatives of y with respect to $x; F$ is a given function relating y and its derivatives. The ODE is said to be linear if it can be written as a linear combination of the derivatives of y. A classic example of ODE from physics is the ODE describing the simple harmonic motion. The motion of a mass attached to a spring is governed by a second-order ODE. The equation is given by:

$$m\frac{d^2x}{dt^2} + c\frac{dx}{dt} + kx = 0 \tag{2.2}$$

This second-order ODE describes the relation between the displacement $x(t)$ of the mass and its acceleration.

Efforts to find solutions for ODEs have been extensive. To analytically solve first-order ODEs, one method is the separation of variables, which basically separates the variables so that x and y will only appear on one side of the equation and thus reducing the difficulty of solving such an equation.

Partial differential equations (PDEs) are equations that involve multiple independent variables and their partial derivatives. They are fundamental in describing physical phenomena that vary in more than one dimension, such as heat diffusion, fluid flow, electromagnetic fields, and quantum mechanics. The general form of a PDE is:

$$F\left(x_1, x_2, \ldots, x_n, u, \frac{\partial u}{\partial x_1}, \frac{\partial u}{\partial x_2}, \ldots, \frac{\partial^2 u}{\partial x_1^2}, \frac{\partial^2 u}{\partial x_1 x_2}, \ldots, \frac{\partial^i u}{\partial x_n^i}\right) = 0 \tag{2.3}$$

Approaches for solving PDEs have also been investigated extensively. Methods like finite difference method, finite volume method, and spectral method can provide an accurate solution to PDEs.

2.1.2 Solving Differential Equations

In many cases, solving differential equations exactly is not a choice for people. Some equations are complicated, and numeric approximations are normally enough. Using the first-order ordinary differential equations as an example. The reason that we can only consider first-order here is that higher-order ODEs can be converted to a larger system of first-order ODEs by introducing extra variables. For example, the second-order equation $y'' = -y$ can be rewritten as two first-order equations: $y' = z$ and $z' = -y$. Numerical methods for solving first-order ODEs often fall into one of two large categories: Linear multistep methods, or Runge-Kutta methods by Kutta 1901. Each method can be further divided into those that are explicit as discussed in Verwer 1996 and those that are implicit as discussed in Ascher, Ruuth, and Wetton 1995. There are some well-known methods like the Euler method, the backward Euler method, the first-order exponential integrator method mentioned in Biswas et al. 2013, and so on.

For PDEs, exact solution is also hard to obtain and not necessary in most engineering and science problems. One famous method is finite difference method, and it represents functions by their values at certain grid points and derivatives are approximated through differences in these values. The method of lines is also widely used, and the main idea is that it discretizes all dimensions except one. The finite element method uses variational methods to minimize an error function, produce a stable solution, and find approximate solutions to boundary value problems. The finite volume method, similar to the finite difference method or finite element method, calculates values at discrete places on a meshed geometry. “Finite volume” refers to the small volume surrounding each node point on a mesh. Spectral methods are techniques used to solve certain differential equations, often involving the use of the fast Fourier transform. The idea is to write the solution of the differential equation as a sum of certain basis functions, and then to choose the coefficients in the sum that best satisfy the differential equation. There are also some mesh-free methods that do not require a mesh connecting the data points of the simulation domain.

There are certain shortcomings associated with these numerical methods. One obvious point is the curse of dimensionality (CoD). In numerical methods for PDEs, this phenomenon manifests when higher-dimensional spaces require significantly more computational effort to achieve accurate solutions. For example, in a one-dimensional problem, a relatively small number of grid points may suffice to approximate the solution. However, as the dimensionality increases to two, three, or higher, the number of grid points, and thus the computational workload, grows exponentially. This exponential growth results in enormous memory and processing demands, making high-dimensional problems infeasible to solve with standard numerical techniques. Additionally, the curse of dimensionality can lead to significant challenges in maintaining numerical stability and accuracy, as errors can propagate and magnify across the expanded computational grid. Another constraint associated with numerical methods is the high computation cost demanded when solving inverse problems or PDE-constrained optimization methods in that the PDE needs to be solved many times.

2.2 Deep Neural Networks

A popular approach to solving complicated problems is to use deep neural networks (DNNs). The simplest network architecture is known as multilayer perceptron (MLP) mentioned in Rosenblatt 1958 and Rumelhart, Hinton, and Williams 1986. If we define our objective as the approximation of a function $f : x \in \mathbb{R}^d \mapsto y \in \mathbb{R}^D$ using an MLP, which we denote as $\mathcal{F}$. Computing units of an MLP, which are known as neurons, are stacked in several consecutive layers. The zeroth layer is called the input layer, which is responsible for providing an input to the network. The last layer is known as the output layer, which outputs the network's prediction. Every other layer in between is known as a hidden layer. A schematic demonstration of such architecture is shown in Figure 1. Here we can use some notations to help us better understand the operations occurring inside an MLP. We consider a network with L hidden layers, and the width of each layer is denoted as H_l for $l = 0, 1, \ldots, L+ 1$. To be consistent with the function we try to approximate in our previous definition, we must have $H_0 = d$ and $H_{L+1} = D$. The output vector for l-th layer can be represented by $x^{(l)} \in \mathbb{R}^{H_l}$, and it will also serve as the input to the next layer. We can also set $x^{(0)} \in \mathbb{R}^d$ as the input signal provided by the input layer. In each layer $l, 1 \leq l \leq L + 1$, the i-th neuron performs an affine transformation on that layer input $x^{(l-1)}$ followed by a non-linear transformation:

$$x_i^{(l)} = \sigma(\underbrace{W_{ij}^{(l)} x_j^{(l-1)}}_{\text{Einstein sum}} + b_i^{(l)}), \quad 1 \leq i \leq H_l, \quad 1 \leq j \leq H_{l-1} \tag{2.4}$$

where $W_{ij}^{(l)}$ and $b_i^{(l)}$ are known as the weights and bias associated with i-th neuron of layer l, while the function $\sigma(\cdot)$ is known as the activation function and plays an important role in helping the network to represent non-linear complex functions. If we use $\boldsymbol{W}^{(l)} \in \mathbb{R}^{H_{L-1} \times H_l}$ to be the weight matrix for layer l and $\boldsymbol{b}^{(l)} \in \mathbb{R}^{H_l}$ to be the bias vector for layer l, then we can re-write the action of the whole layer as

$$x^{(l)} = \sigma\left(\mathcal{A}^{(\ell)}\left(x^{(l-1)}\right)\right), \quad \mathcal{A}^{(\ell)}\left(x^{(l-1)}\right) = W^{(l)} x^{(l-1)} + b^{(l)} \tag{2.5}$$

where the activation function is applied component-wise. Finally, the computation of the whole network $\mathcal{F} : \mathbb{R}^d \mapsto \mathbb{R}^D$ can be seen as a composition of alternating affine transformations and component-wise activations:

$$\mathcal{F}(\boldsymbol{x}) = \mathcal{A}^{(\mathcal{L}+1)} \circ \sigma \circ \mathcal{A}^{(\mathcal{L})} \circ \sigma \circ \mathcal{A}^{(\mathcal{L}-1)} \circ \cdots \circ \sigma \circ \mathcal{A}^{(1)}(x). \tag{2.6}$$

The parameters of the network is all the weights and biases, which we will represent as

$$\theta = \left\{W^{(l)}, b^{(l)}\right\}_{l=1}^{L+1} \in R^{N_\theta} \tag{2.7}$$

where N_θ denotes the total number of parameters of the network. The network $\boldsymbol{F}(\boldsymbol{x};\boldsymbol{\theta})$ represents a family of parameterized functions, where $\boldsymbol{\theta}$ needs to be suitably chosen such that, the network approximates the target function $f(x)$ at the input x.

2.2.1 Activation Functions

One component that could be the most important is the activation function of an MLP. There are various types of activations investigated by literature and each of them has its own advantages and disadvantages. We will introduce some of them in the following paragraphs.

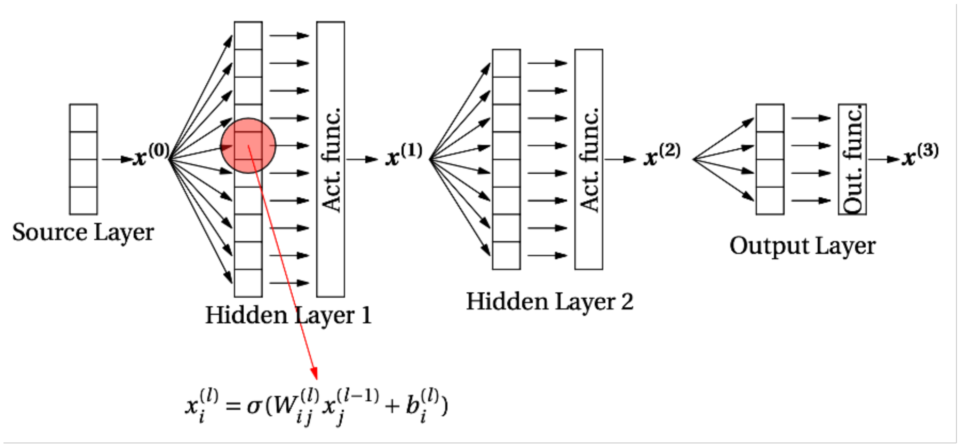


Figure 2.1: MLP with 2 hidden layers. RAY, Pinti, and Oberai 2024

The simplest activation can be written as $\sigma(\xi) = \xi$. This function is infinitely smooth, ranging from negative infinity to positive infinity, with all derivatives beyond the second order are zero. Using this linear activation function for all layers in an MLP will reduce the entire network to a single affine transformation of the input. Therefore, the network will be nothing more than a linear approximation of the target function, and it will be totally useless if the target function is highly non-linear.

One of the most well-known and widely used activation function is called Rectified Linear Unit (ReLU). This is a piece function and is defined as

$$\sigma(\xi) = \max\{0, \xi\} = \begin{cases} \xi, & \text{if } \xi \geq 0 \\ 0, & \text{if } \xi < 0 \end{cases} \tag{2.8}$$

The function is continuous, while its derivative will be piecewise constant with a jump $\xi = 0$. The second derivative will be a dirac function concentrated at $\xi = 0$. Or in other words, the higher-order derivates are not well-defined. The range of the function is from 0 to positive infinity. To overcome the issue of ReLU that it can lead to a null output from a neuron if the affine transformation of the neuron is negative, a leaky version of ReLU mentioned by Xu et al. 2015 was designed.

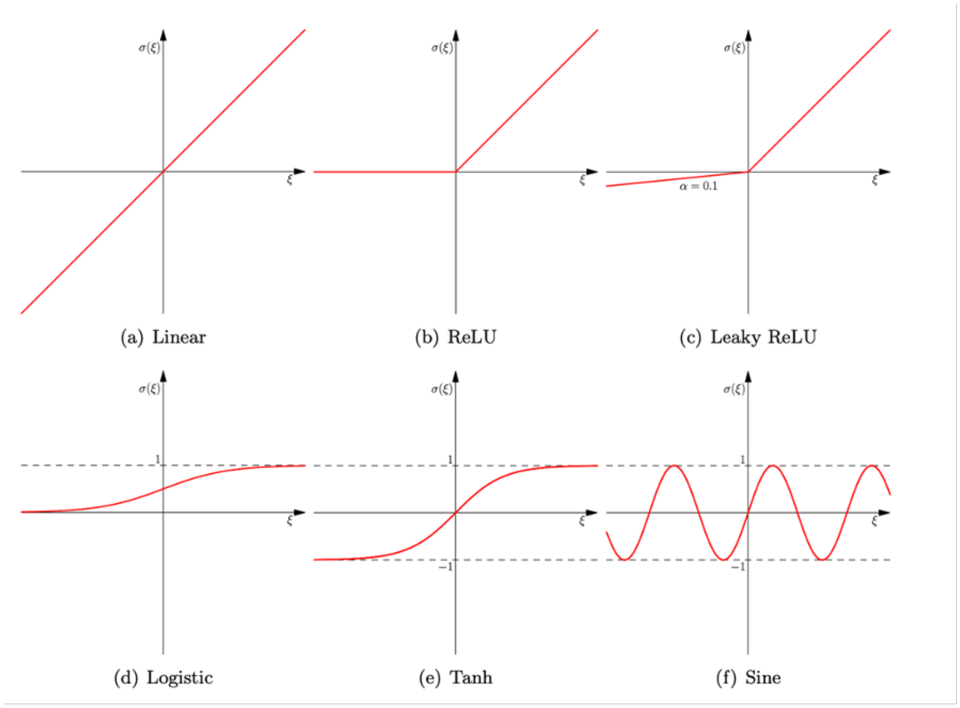
$$\sigma(\xi; \alpha) = \max\{0, \xi\} = \begin{cases} \xi, & \text{if } \xi \geq 0 \\ \alpha\xi, & \text{if } \xi < 0 \end{cases} \tag{2.9}$$

where α becomes a network hyper-parameter. The derivates of Leaky ReLU behave the same way as those for ReLU, while its range now is from negative infinity to positive infinity.

Another famous choice is the logistic function, or Sigmoid function. The Sigmoid activation function is given by

$$\sigma(\xi) = \frac{1}{1 + e^{-\xi}} \tag{2.10}$$

and it has the following properties: The function is infinitely smooth and monotonic; the range of the function is from 0 to 1; the derivative quickly decays



(a) Linear (b) ReLU (c) Leaky ReLU

(d) Logistic (e) Tanh (f) Sine

Figure 2.2: Different activation functions. RAY, Pinti, and Oberai 2024

to zero away from $\xi = 0$, this activation function can lead to slow convergence of the network when training.

Tanh is also a frequently used activation function, and it can be seen as a symmetric extension of the sigmoid function

$$\sigma(\xi) = \frac{e^{\xi} - e^{-\xi}}{e^{\xi} + e^{-\xi}} \tag{2.11}$$

This function is also infinitely smooth and monotonic, while its range is from -1 to 1 . Similar to the logistic function, the derivative of tanh quickly decays to zero away from $\xi = 0$ and can also lead to slow convergence for training networks.

2.2.2 Train, validation, and testing of neural networks

With all the previous knowledge, we now have a sufficient understanding of the architecture of MLPs. One question that is left unanswered is how the parameters of these networks are set to approximate some target function. The discussion here will be focused on the framework of supervised learning. Let us assume that we are given a dataset of pairwise samples $\mathcal{S} = \{(x_i, y_i) : 1 \leq i \leq N\}$ corresponding to a target function $\boldsymbol{f} : \boldsymbol{x} \mapsto \boldsymbol{y}$. We wish to approximate the function using the neural network $\mathcal{F}(\boldsymbol{x}; \boldsymbol{\theta}, \boldsymbol{\Theta})$ where $\boldsymbol{\theta}$ are the network parameters; $\boldsymbol{\Theta}$ corresponds to the hyperparameters of the network such as depth, width, activation function type, etc. The method to design a robust network involves three main steps: 1. Find the optimal values of $\boldsymbol{\theta}$ in the training phase; 2. Find the optimal value of Θ in the validation phase; 3. Test the performance of the network on unseen data on the testing phase. Correspondingly, the dataset $\mathcal{S}$ will also be divided into three distinct parts corresponding to these three steps. In the following paragraphs, these three steps will be discussed in detail.

Training the network makes use of the training dataset to solve the following optimization problem: Find

$$\boldsymbol{\theta}^{*} = \arg\min_{\boldsymbol{\theta}} \Pi_{\text{train}}\left(\boldsymbol{\theta}\right), \text{ where } \Pi_{\text{train}}\left(\boldsymbol{\theta}\right) = \frac{1}{N_{\text{train}}} \sum_{(\boldsymbol{x}_i,\boldsymbol{y})\in\delta_{\text{train}}}^{N_{\text{train}}} \left\|\boldsymbol{y}_i - \mathcal{F}\left(\boldsymbol{x}_i;\boldsymbol{\theta},\boldsymbol{\theta}\right)\right\|^2 \tag{2.12}$$

for some fixed $\boldsymbol{\Theta}$. The optimal $\boldsymbol{\theta}^{*}$ is obtained using a proper gradient-based algorithm. The function Π_{train} is referred to as the loss function. In this example, the mean-squared loss function is used here. There are also other types indeed.

Validation process involves using the validation dataset to solve another optimization problem that

$$\boldsymbol{\Theta}^{*} = \arg\min_{\boldsymbol{\Theta}} \Pi_{\text{val}}(\boldsymbol{\Theta}), \text{ where } \Pi_{\text{val}}\left(\boldsymbol{\Theta}\right) = \frac{1}{N_{\text{val}}} \sum_{\substack{i=1\\(\boldsymbol{x}_i y_i)\in\mathcal{S}_{\text{val}}}}^{N_{\text{val}}} \left\|\boldsymbol{y}_i - \mathcal{F}\left(\boldsymbol{x}_i;\boldsymbol{\theta}^{*},\boldsymbol{\Theta}\right)\right\|^2 \tag{2.13}$$

Through this process, the optimal $\boldsymbol{\Theta}^{*}$ is obtained. Once the best network, characterized by $\boldsymbol{\theta}^{*}$ and Θ^{*}, is obtained, it is evaluated on the test dataset to estimate the networks performance on data not used during the first two phases.

$$\Pi_{\text{test}} = \frac{1}{N_{\text{test}}} \sum_{\substack{i=1\\(\boldsymbol{x}_i\boldsymbol{y}_i)\in\mathcal{S}_{\text{test}}}}^{N_{\text{test}}} \left\|\boldsymbol{y}_i - \mathcal{F}\left(\boldsymbol{x_i};\boldsymbol{\theta}^{*},\boldsymbol{\Theta}^{*}\right)\right\|^2 \tag{2.14}$$

This error will be the test error and is also known as the generalizing error of the network.

2.2.3 Optimization

In the previous content, we can conclude that the network is modified to best approximate a function by solving some minimization problems in different phases. Practically, this minimization problem can be solved using gradient descent (GD). Consider the Taylor expansion about initial network parameter θ_0

$$\Pi\left(\boldsymbol{\theta}_0+\Delta\boldsymbol{\theta}\right)=\Pi\left(\boldsymbol{\theta}_0\right)+\frac{\partial\Pi}{\partial\boldsymbol{\theta}}\left(\boldsymbol{\theta}_0\right)\cdot\Delta\boldsymbol{\theta}+\frac{\partial^2\Pi}{\partial\theta_i\theta_j}(\widehat{\boldsymbol{\theta}})\Delta\theta_i\Delta\theta_j \tag{2.15}$$

for some $\widehat{\boldsymbol{\theta}}$ in a small neighborhood of $\boldsymbol{\theta}_0$. When $|\Delta\boldsymbol{\theta}|$ is small and summing $\frac{\partial^2\Pi}{\partial\theta_i\theta_j}$ is bounded, we can neglect the second order term and just consider the approximation

$$\Pi\left(\boldsymbol{\theta}_0+\Delta\boldsymbol{\theta}\right)\approx\Pi\left(\boldsymbol{\theta}_0\right)+\frac{\partial\Pi}{\partial\boldsymbol{\theta}}\left(\boldsymbol{\theta}_0\right)\cdot\Delta\boldsymbol{\theta} \tag{2.16}$$

To decrease the value of the loss function to the maximum extent compared to its evaluation at $\boldsymbol{\theta}_0$, we need to choose the step $\boldsymbol{\Delta}$ in the opposite direction of the gradient. For instance:

$$\Delta\boldsymbol{\theta}=-\eta\frac{\partial\Pi}{\partial\boldsymbol{\theta}}\left(\boldsymbol{\theta}_0\right)$$

where η is known as the learning-rate. This is also one of the hyper-parameters that we would like to tune during the validation phase.

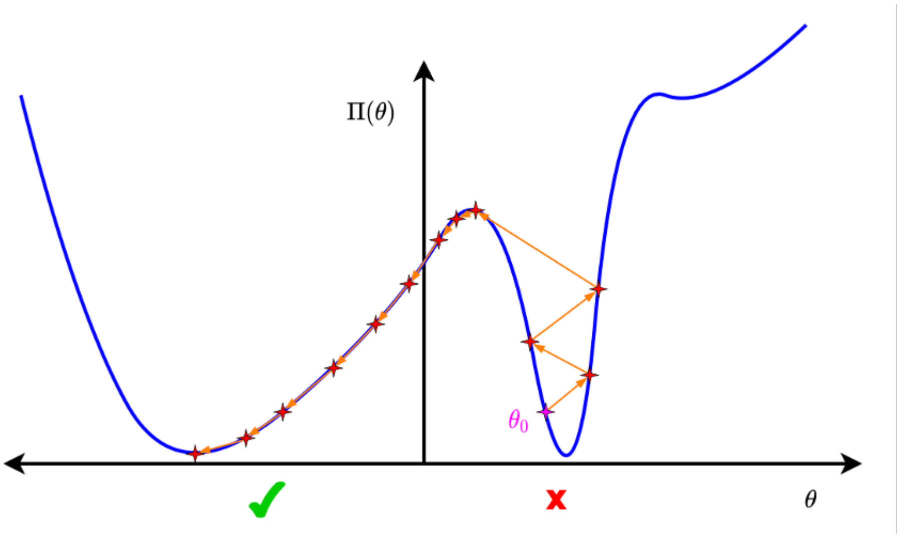


Figure 2.3: Gradient Descent prefers flatter minimas. RAY, Pinti, and Oberai 2024

If we assume the loss function $\Pi(\theta)$ is convex and differentiable, and its gradient is Lipschitz continuous with Lipschitz constant $\mathcal{K}$. Then for a $\eta \leq 1/\mathcal{K}$, the gradient descent updates converge as

$$\|\boldsymbol{\theta}^* - \boldsymbol{\theta}_k\|_2 \leq \frac{C}{k}$$

This works for the case where only one minima exists. But what if more than one minima exist? What kind of minima does gradient descent like to reach? A direct answer to this question is that gradient descent prefers flatter minimas, as can be seen in figure 2.3.

Starting from the function $\Pi(\theta)$ in the figure, and we assume each value of $\Pi(\theta)$ can be approximated by a parabola

$$\Pi(\theta) \approx \frac{1}{2}a\theta^2$$

where $a > 0$ is the curvature of each valley. If we use a constant learning rate η and start from $\boldsymbol{\theta}_0$ in either of the valleys. Then,

$$\frac{\partial \Pi}{\partial \theta}(\theta_0) = a\theta_0$$

and the new point after an update will be $\theta_1 = \theta_0(1-a\eta)$. The subsequent iterates will be similar to this. To reach convergence, we need

$$\left|\frac{\theta_{k+1}}{\theta_k}\right| < 1 \Rightarrow |1 - a\eta| < 1$$

Since $a > 0$ in the valleys, we will need the following condition on the learning rate such that $a\eta < 2$. In other words, the curvature a should satisfy that $a < 2/\eta$. Therefore, larger curvature requires a smaller learning rate, and vice versa. If we fix the learning rate, the gradient descent will naturally prefer the smaller curvature, and thus the flatter minimas.

2.2.4 Advanced Optimization Methods

In the previous content, we are mainly focusing on the situation in which the optimization problem is solved through gradient descent. However, there are also some other optimization techniques that exist, and they are mostly motivated by gradient descent.

Let us recall the update formula mentioned before, as most optimization algorithms make use of it:

$$[\boldsymbol{\theta}_{k+1}]_i = [\boldsymbol{\theta}_k]_i - [\boldsymbol{\eta}_k]_i \, [\boldsymbol{g}_k]_i \, , 1 \leq i \leq N_\theta,$$

where $[\boldsymbol{\eta}_k]_i$ is the component-wise learning rate and the vector-valued function $\boldsymbol{g}$ approximates the gradient. From the previous section, we can see that the learning rate can affect the choice of minimas for gradient descent. This can lead to a problem that if the learning rate is too large and the objective function landscape has sharp gradients, the update will keep zigzagging its way toward the minima. Ideally, we want to make sure the update only happens on one side of the function so that we can reach the minima faster. Zig-zagging will impede us from reaching the minima or make the convergence slower.

To resolve the issues faced by gradient descent, there are two popular methods. First category includes the momentum methods. Rather than just using the gradient at the previous step, these methods make use of the history of the gradient. The formula for the update is given by

$$[\boldsymbol{\eta}_k]_i = \eta, \boldsymbol{g}_k = \beta_1 \boldsymbol{g}_{k-1} + (1 - \beta_1) \frac{\partial \Pi}{\partial \boldsymbol{\theta}} (\boldsymbol{\theta}_k) \, , \boldsymbol{g}_{-1} = 0 \tag{2.17}$$

where $\boldsymbol{g}_k$ is a weighted moving average of the gradient, and it is expected to smoothen out the zig-zagging by canceling out the components of the gradient along the other side of the function and move more smoothly towards the minima. Another method that makes use of the history of the gradient is called Adam and was introduced by Kingma and Ba 2014. What Adam is different is that it also uses the second moment of the gradient. The updates are given by

$$\begin{aligned} \boldsymbol{g}_k &= \beta_1 \boldsymbol{g}_{k-1} + (1-\beta_1)\frac{\partial \Pi}{\partial \boldsymbol{\theta}}(\boldsymbol{\theta}_k) \\ [\boldsymbol{G}_k]_i &= \beta_2 [\boldsymbol{G}_{k-1}]_i + (1-\beta_2)\left(\frac{\partial \Pi}{\partial \boldsymbol{\theta}_i}(\boldsymbol{\theta}_k)\right)^2 \\ [\boldsymbol{\eta}_k]_i &= \frac{\eta}{\sqrt{[\boldsymbol{G}_k]_i} + \epsilon} \end{aligned} \tag{2.18}$$

where $\boldsymbol{g}_k$ and $\boldsymbol{G}_k$ are the weighted running averages of the gradients and the square of the gradients. Another widely-used optimization method is called Stochastic optimization. To understand this method, we need to look back to the form of the training loss:

$$\Pi(\boldsymbol{\theta}) = \frac{1}{N_{\text{train}}} \sum_{i=1}^{N_{\text{train}}} \Pi_i(\boldsymbol{\theta}), \Pi_i(\boldsymbol{\theta}) = \|\boldsymbol{y}_i - \mathcal{F}(\boldsymbol{x}_i; \boldsymbol{\theta}, \boldsymbol{\theta})\|^2 \tag{2.19}$$

Then we can obtain the gradient of loss by

$$\frac{\partial \Pi}{\partial \boldsymbol{\theta}}(\boldsymbol{\theta}) = \frac{1}{N_{\text{train}}} \sum_{i=1}^{N_{\text{train}}} \frac{\partial \Pi_i}{\partial \boldsymbol{\theta}}(\boldsymbol{\theta}) \tag{2.20}$$

However, one issue with this calculation is that N_{train} is typically very large and this summation could be very expensive. To avoid this problem, we randomly choose i in Π_i for each update step. This is known as stochastic gradient descent. Stochastic optimization algorithms are still not perfect, as the loss function fluctuates in a chaotic manner and never manages to reach the minima, and handling only one sample at a time significantly underutilizes the computational and memory resources. An alternating solution is to use mini-batch optimization, where the training dataset N_{train} is split into disjoint subsets known as minibatches. Therefore, the gradient of the loss function then can be approximated by

$$\frac{\partial \Pi}{\partial \boldsymbol{\theta}}(\boldsymbol{\theta}) = \frac{1}{N_{\text{train}}} \sum_{i=1}^{N_{\text{train}}} \frac{\partial \Pi_i}{\partial \boldsymbol{\theta}}(\boldsymbol{\theta}) \approx \frac{1}{\bar{N}_{\text{train}}} \sum_{i \in \text{ batch } (j)} \frac{\partial \Pi_i}{\partial \boldsymbol{\theta}}(\boldsymbol{\theta}) \tag{2.21}$$

2.2.5 Backward-Propagation

In the previous section, one thing that we discussed the most is the gradient. However, how the gradients are actually evaluated while training the neural network is a crucial point that we need to understand. Let us revisit the output of an MLP given by

$$\begin{aligned} \xi_i^{(l+1)} &= W_{ij}^{(l+1)} x_j^{(l)} + b_i^{(l+1)}, 1 \leq i \leq H_{l+1} \\ x_i^{(l+1)} &= \sigma\left(\xi_i^{(l+1)}\right), 1 \leq i \leq H_{l+1} \end{aligned}$$

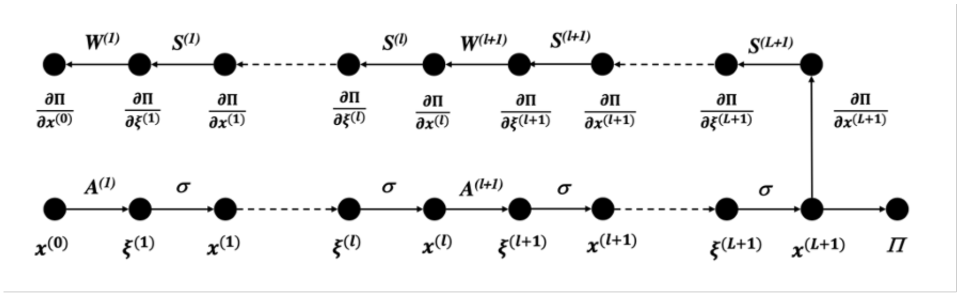

$W^{(1)}$
$S^{(1)}$
$S^{(l)}$
$W^{(l+1)}$
$S^{(l+1)}$
$S^{(L+1)}$
$\frac{\partial \Pi}{\partial x^{(0)}}$
$\frac{\partial \Pi}{\partial \xi^{(1)}}$
$\frac{\partial \Pi}{\partial x^{(1)}}$
$\frac{\partial \Pi}{\partial \xi^{(l)}}$
$\frac{\partial \Pi}{\partial x^{(l)}}$
$\frac{\partial \Pi}{\partial \xi^{(l+1)}}$
$\frac{\partial \Pi}{\partial x^{(l+1)}}$
$\frac{\partial \Pi}{\partial \xi^{(L+1)}}$
$\frac{\partial \Pi}{\partial x^{(L+1)}}$
$A^{(1)}$
σ
σ
$A^{(l+1)}$
σ
σ
$x^{(0)}$
$\xi^{(1)}$
$x^{(1)}$
$\xi^{(l)}$
$x^{(l)}$
$\xi^{(l+1)}$
$x^{(l+1)}$
$\xi^{(L+1)}$
$x^{(L+1)}$
Π

Figure 2.4: Computational graph. RAY, Pinti, and Oberai 2024

Given a training sample (x, y), set $x^{(0)} = x$. The value of the loss can be computed using the forward pass, which is basically evaluate the equations above in order, and then evaluate the loss function for the given sample as

$$\Pi(\boldsymbol{\theta}) = \|\boldsymbol{y} - \mathcal{F}(\boldsymbol{x}; \boldsymbol{\theta}, \boldsymbol{\Theta})\|^2$$

This process will indeed be repeated for all the samples, but here we will just focus on dealing with a single sample.

To update the network parameters, we need to compute $\frac{\partial \Pi}{\partial \theta}$, or more precisely, $\frac{\partial \Pi}{\partial \boldsymbol{W}^{(l)}}$ and $\frac{\partial \Pi}{\partial \boldsymbol{b}^{(l)}}$. To compute them, we need to first derive the expression for $\frac{\partial \Pi}{\partial \xi^{(l)}}$ and $\frac{\partial \Pi}{\partial \boldsymbol{x}^{(l)}}$. This derivation can be easily understood with the help of the computational graph shown in the figure.

Applying the chain rule repeatedly yields

$$\frac{\partial \Pi}{\partial \xi^{(l)}} = \frac{\partial \Pi}{\partial \boldsymbol{x}^{(L+1)}} \cdot \frac{\partial \boldsymbol{x}^{(L+1)}}{\partial \boldsymbol{\xi}^{(L+1)}} \cdot \frac{\partial \boldsymbol{\xi}^{(L+1)}}{\partial \boldsymbol{x}^{(L)}} \cdots \frac{\partial \boldsymbol{x}^{(l+1)}}{\partial \boldsymbol{\xi}^{(l+1)}} \cdot \frac{\partial \boldsymbol{\xi}^{(l+1)}}{\partial \boldsymbol{x}^{(l)}} \cdot \frac{\partial \boldsymbol{x}^{(l)}}{\partial \boldsymbol{\xi}^{(l)}}$$

To further evaluate this expression, we need to evaluate these terms:

$$\begin{aligned}
\frac{\partial \Pi}{\partial \boldsymbol{x}^{(L+1)}} &= -2\left(\boldsymbol{y} - \boldsymbol{x}^{(L+1)}\right)^T \\
\frac{\partial \boldsymbol{\xi}^{(l+1)}}{\partial \boldsymbol{x}^{(l)}} &= \boldsymbol{W}^{(l+1)} \\
\frac{\partial \boldsymbol{x}^{(l)}}{\partial \boldsymbol{\xi}^{(l)}} &= \boldsymbol{S}^{(l)} \equiv \operatorname{diag}\left[\sigma'\left(\xi_1^{(l)}\right), \cdots, \sigma'\left(\xi_{H_l}^{(l)}\right)\right]
\end{aligned}$$

Using these terms, we can then represent $\frac{\partial \Pi}{\partial \xi(l)}$ as

$$\frac{\partial \Pi}{\partial \boldsymbol{\xi}^{(l)}} = \frac{\partial \Pi}{\partial \boldsymbol{x}^{(L+1)}} \cdot \boldsymbol{S}^{(L+1)} \cdot \boldsymbol{W}^{(L+1)} \cdots \boldsymbol{S}^{(l+1)} \cdot \boldsymbol{W}^{(l+1)} \cdot \boldsymbol{S}^{(l)} \tag{2.22}$$

Applying transpose and recognizing that $\boldsymbol{\Sigma}^{(l)}$ is diagonal and therefore symmetric, we can have a final representation

$$\frac{\partial \Pi}{\partial \boldsymbol{\xi}^{(l)}} = \boldsymbol{S}^{(l)} \boldsymbol{W}^{(l+1)T} \boldsymbol{S}^{(l+1)} \cdots \boldsymbol{W}^{(L+1)T} \boldsymbol{S}^{(L+1)} \left[-2\left(\boldsymbol{y} - \boldsymbol{x}^{(L+1)}\right)\right] \tag{2.23}$$

Finally, we can have the expression for $\frac{\partial \Pi}{\partial \boldsymbol{W}^{(2)}}$ as

$$\frac{\partial \Pi}{\partial \boldsymbol{W}^{(l)}} = \frac{\partial \Pi}{\partial \boldsymbol{\xi}^{(l)}} \cdot \frac{\partial \boldsymbol{\xi}^{(l)}}{\partial \boldsymbol{W}^{(l)}} = \frac{\partial \Pi}{\partial \boldsymbol{\xi}^{(l)}} \otimes \boldsymbol{x}^{(l-1)} \tag{2.24}$$

2.3 Transfer Learning

In previous sections, we know how machine learning models, especially neural networks, approximate a target function based on a given dataset and task. It is certain that the prediction performance will be degraded when the distribution of the dataset is shifted or changed. This issue in some cases is unavoidable as obtaining training data that matches the feature space and predicted data distribution characteristics of the test data can be difficult and expensive. Therefore, there is an urgent demand for a high-quality model to predict a target domain that is trained from a related, rather the same, source domain. This need summarizes the motivation for transfer learning.

The above description may sound not feasible, but real-world experiences tell us that transfer learning happens quite frequently. For example, there are two people who wish to learn violin. One person has no previous experience playing any musical instrument, while the other knows how to play Erhu, a Chinese musical instrument which has basically the same mechanism as violin, very well. Then person who knows how to play Erhu will be able to learn the violin in a more efficient manner by transferring previously learned knowledge to the task of learning how to play the violin. In the context of machine learning, there is also a concrete example about predicting text sentiment of product reviews. If the training data and the target data are both obtained for digital camera reviews, then some traditional methods can help to achieve good prediction results. However, if the case is changed to the situation that training data and target data is from food quality reviews, then the prediction results are mostly likely to degrade due to domain differences. Still, the reviews for digital cameras provide valuable information for predicting food quality review sentiments as they share a considerable number of common characteristics. Thus, transfer learning can be used to potentially improve the results of a target learner. The advantages of transfer learning are more than what we have seen. In general, transfer learning mainly benefits scenarios for edge computing. Training machine learning models on edge platforms, like Micro-Controller Units, FPGA, etc., face two major constraints: Limited data and limited computing resources. Some edge computing tasks require the platform to collect and store data locally, and thus prohibit the model from being trained with rich data. Transfer learning, however, can help to overcome this issue in that it uses knowledge from the pre-trained model that is trained on a larger dataset, and thus helps the model to adapt to the limited data case. To resolve the limitation of computing resources, transfer learning can help us greatly reduce the training time since it allows models to leverage pre-trained weights. On platforms where low latency models are required, such a reduction in training time is especially beneficial as it requires much less power to support the entire training process. Furthermore, training a whole model requires a large amount of matrix computation, and doing such computation without using GPUs can be extremely inefficient. Transfer learning helps to reduce the amount of computation we need to perform by order of tens or even more. As edge platforms face various tasks, the better generalizability brought by transfer learning also benefits the model a lot.

2.4 Physics-Informed Neural Networks

From previous knowledge, we can see that MLPs can be used to approximate functions. In the context of Partial Differential Equations (PDEs), will MLP still be able to approximate the solution function? In fact, the idea of using neural networks to solve PDEs was introduced very early in the 1990-2000s by Lagaris, Likas, and Fotiadis 1998. Recently, with the improvements in the hardware and more mature machine learning tools, this idea was rediscovered in 2019 by Raissi, Perdikaris, and George E Karniadakis 2019, and was the term Physics-Informed Neural Networks (PINNs). The fundamental idea of PINNs is similar to regression, except that the loss function contains differential operators arising in the PDE we are dealing with. The main process for solving a one-dimensional scalar PDE is described in the following content. One remark is that for multi-dimensional systems of PDEs, this process can be easily extended. The whole process starts with selecting a neural network as the representation of the PDE solution function $u = \mathcal{F}(x; \boldsymbol{\theta})$. Then we need to find $\boldsymbol{\theta}$ such that the PDE is satisfied in a proper form. An important distinction between training a PINN from training a normal

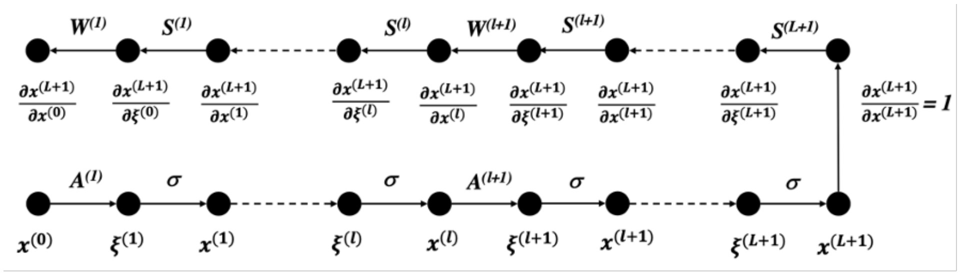

$W^{(1)}$
$S^{(1)}$
$S^{(l)}$
$W^{(l+1)}$
$S^{(l+1)}$
$S^{(L+1)}$
$\frac{\partial x^{(L+1)}}{\partial x^{(0)}}$
$\frac{\partial x^{(L+1)}}{\partial \xi^{(0)}}$
$\frac{\partial x^{(L+1)}}{\partial x^{(1)}}$
$\frac{\partial x^{(L+1)}}{\partial \xi^{(l)}}$
$\frac{\partial x^{(L+1)}}{\partial x^{(l)}}$
$\frac{\partial x^{(L+1)}}{\partial \xi^{(l+1)}}$
$\frac{\partial x^{(L+1)}}{\partial x^{(l+1)}}$
$\frac{\partial x^{(L+1)}}{\partial \xi^{(L+1)}}$
$\frac{\partial x^{(L+1)}}{\partial x^{(L+1)}} = 1$
$A^{(1)}$
σ
σ
$A^{(l+1)}$
σ
σ
$x^{(0)}$
$\xi^{(1)}$
$x^{(1)}$
$\xi^{(l)}$
$x^{(l)}$
$\xi^{(l+1)}$
$x^{(l+1)}$
$\xi^{(L+1)}$
$x^{(L+1)}$

Figure 2.5: Computational graph to evaluate derivatives w.r.t network input. RAY, Pinti, and Oberai 2024

MLP is that training a PINN requires us to find the derivatives that is a part of the loss function. We can use a similar computational graph to show the idea in figure 2.5.

As we set $\Pi = \boldsymbol{x}^{(L+1)}$ in the graph, one main difference for this computational graph with the previous one is that we are computing derivatives through backward-propagation with respect to different things. Previous it was calculating over the loss function. Here, the derivative is respecting to the output of the network. From this graph, we can easily see that

$$\frac{\partial \boldsymbol{x}^{(L+1)}}{\partial \boldsymbol{x}^{(0)}} = \boldsymbol{W}^{(L+1)}\boldsymbol{S}^{(L+1)}\boldsymbol{W}^{(L)}\boldsymbol{S}^{(L)} \dots \boldsymbol{W}^{(2)}\boldsymbol{S}^{(2)}\boldsymbol{W}^{(1)}\boldsymbol{S}^{(1)} \tag{2.25}$$

As we can see from the computational graph the calculation of $\frac{\mathrm{d}u}{\mathrm{d}x}$ requires one backward branch. To evaluate the second-order derivative, we will need another backward branch that is extended from the first backward branch. The same logic applies to all higher orders of derivatives. Another schematic demonstration of

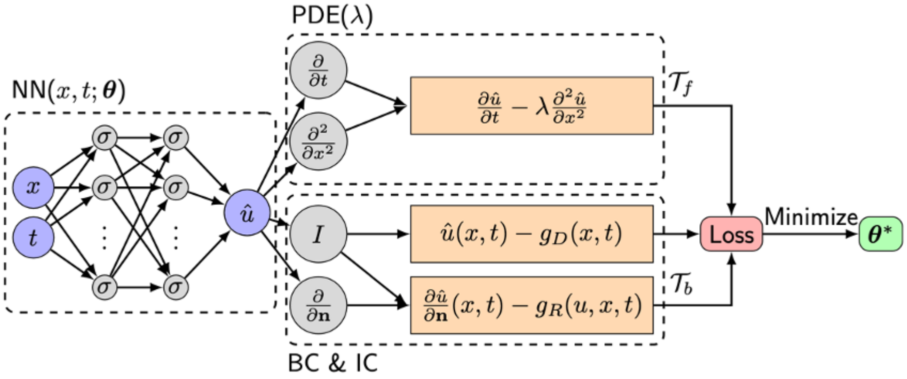

PDE(λ)
NN($x, t; \boldsymbol{\theta}$)
x
t
σ
$\hat{u}$
$\frac{\partial}{\partial t}$
$\frac{\partial^2}{\partial x^2}$
$\frac{\partial \hat{u}}{\partial t} - \lambda \frac{\partial^2 \hat{u}}{\partial x^2}$
$\mathcal{T}_f$
I
$\hat{u}(x,t) - g_D(x,t)$
Loss
Minimize
$\boldsymbol{\theta}^*$
$\frac{\partial}{\partial \mathbf{n}}$
$\frac{\partial \hat{u}}{\partial \mathbf{n}}(x,t) - g_R(u,x,t)$
$\mathcal{T}_b$
BC & IC

Figure 2.6: Structure of Physics-Informed Neural Networks

how PINNs work is demonstrated in figure 2.6. We can further understand how PINNs work for solving PDEs by using a general example.

Consider a PDE that:

$$L(u(x)) = f(x), x \in \Omega$$

$$B(u(x)) = g(x), x \in \partial\Omega$$

where L is the differential operator and B is the boundary operator. f is known as the forcing term, and g is known as the boundary condition. In this example, we use the 3-D Navier-Stokes equation solving for the velocity field $v = [v_1, v_2, v_3]$ and pressure p on $\Omega = \Omega_s \times [0, T]$. Here Ω_s is the spatial domain and $[0, T]$ is the time interval we care about. The equation is given as

$$\begin{gathered}\frac{\partial v}{\partial t} + \boldsymbol{v} \cdot \nabla v + \nabla p - \mu \Delta v = \boldsymbol{f}, \forall (\boldsymbol{s}, t) \in \Omega \\ \nabla \cdot \boldsymbol{u} = 0, \forall (\boldsymbol{s}, t) \in \Omega \\ \boldsymbol{v} = \boldsymbol{0}, \forall (\boldsymbol{s}, t) \in \partial \Omega_S \times [0, T] \\ \boldsymbol{v}(\boldsymbol{s}, 0) = \boldsymbol{v}_0(\boldsymbol{s}), \forall \boldsymbol{s} \in \Omega_S.\end{gathered} \tag{2.26}$$

In this equation, the first one is the balance of linear moment. The second equation ensures the conservation of mass. The third and fourth one indicates the boundary and initial condition. To design a PINN for this specific equation, the input dimension should be the same as the coordinates that are involved in this equation, and the output will be the solution vector. More specifically, the input here will just be $x = [s_1, s_2, s_3, t] \in \mathbb{R}^4$ and the output will be $u = [v_1, v_2, v_3, p] \in \mathbb{R}^4$. The next step will be constructing the loss function. The loss function here will consist of two main parts: interior residual and boundary residual. Interior residual will just be $R(u) = L(u) - f$ and the boundary residual is $R_b(u) = B(u) - g$. Thus, the total loss function is

$$\mathcal{L}(\theta) = R(u) + \lambda_b R_b(\theta)$$

Then, to train the network we just need to solve this optimization problem $\boldsymbol{\theta}^* = \underset{\boldsymbol{\theta}}{\arg\min}\, \mathcal{L}(\boldsymbol{\theta})$ using algorithms mentioned before. After extensive training and hyper-parameter tuning, we will then be confident to say that this PINN is able to solve this Navier-Stokes equation precisely in the given domain.

2.4.1 Applications

Besides solving partial differential equations directly, PINNs can accomplish a lot of different tasks in both science and engineering domains. Engineering problems often involve solving complex differential equations that describe physical phenomena such as fluid dynamics, heat transfer, and structural mechanics. PINNs excel in these areas by integrating the governing equations into the neural network training process. This integration allows for more accurate and efficient solutions compared to traditional numerical methods. For instance, in fluid dynamics, PINNs are used to model fluid flow in various contexts, including aerodynamics, hydrodynamics, and weather forecasting. They can predict flow patterns, turbulence, and pressure distribution with high accuracy, which is crucial for designing efficient aircraft, and ships, and predicting natural disasters. In structural engineering, PINNs help analyze stress and strain in materials under various loads, essential for designing safe and resilient structures, such as bridges, buildings, and mechanical components.

PINNs have shown great promise in the healthcare sector by improving the modeling of biological systems and medical imaging processes. These applications can lead to better diagnosis, treatment planning, and understanding of complex biological interactions. In medical imaging, PINNs enhance the accuracy of techniques like MRI shown by X. Yu et al. 2023 and Herten et al. 2022 by incorporating physical principles of image formation, leading to higher-resolution images and more precise diagnostic capabilities. In cardiovascular modeling, PINNs are used to model blood flow and pressure in the cardiovascular system, aiding in the diagnosis and treatment of heart diseases, and simulating the effects of medical interventions to help in the planning of surgeries and other treatments.

Environmental scientists leverage PINNs to model and predict complex environmental systems as shown by Waheed et al. 2021 and Y. Zhang, Zhu, and Gao 2023, which is crucial for addressing climate change, pollution, and natural resource management. In climate modeling, PINNs improve models by integrating physical laws governing atmospheric and oceanic processes, as shown by Lütjens et al. 2021, leading to more accurate predictions of climate patterns and helping policymakers make informed decisions to combat climate change. In hydrogeology, PINNs are used to simulate groundwater flow and contamination spread like the work by X. Zhang et al. 2022, vital for managing water resources and mitigating the impact of pollutants on the environment. Understanding and predicting the behavior of materials under different conditions is essential for developing new materials and improving existing ones. PINNs play a crucial role in this field by providing accurate models of material behavior at various scales. In molecular dynamics shown by J. Li, Chen, and B. Li 2022, PINNs simulate the interactions between atoms and molecules, helping researchers understand material properties at the atomic level, which is crucial for designing new materials with desired properties. Additionally, they model phase transitions in materials, such as the melting of metals or the crystallization of polymers, aiding in optimizing manufacturing processes and developing materials with specific characteristics.

In the finance sector, PINNs are utilized to model complex financial systems and processes, as demonstrated by Noguer i Alonso and Maxwell 2023, improving risk management, investment strategies, and market analysis. They enhance traditional models for pricing financial derivatives by incorporating the underlying physical and stochastic processes, leading to more accurate pricing and better risk management strategies. By modeling the behavior of financial markets, PINNs help in understanding market trends, volatility, and the impact of external factors, supporting the development of robust trading strategies and financial regulations.

2.4.2 Reachability in High Dimensions

One engineering problem that is solved by the framework of PINNs is the high-dimensional reachability problem. In autonomous systems, with the urgent need for safety guarantees and controllers for these systems, a verification method that computes both the safe configurations and the corresponding safe controller for the system is used, which is known as Hamilton-Jacobi reachability analysis. In this analysis, the Backward Reachable Tube (BRT) of a dynamical system is computed and it will give a set of states. Trajectories that start from this set will eventually reach some given targets set despite the worst-case disturbance. An example would be an aerial vehicle under the disturbance of wind or another adversarial aircraft flying nearby. In this situation, the target set could be the destination of the vehicle. The BRT, therefore, provides both the set of states from which the aerial vehicle can safely reach its destination and a robust controller for the vehicle. To actually solve this problem, the traditional method formulates a zero-sum dynamic game between the control and the disturbance whose value function can be used to synthesize the BRT and the safety controller. This process involves solving a Hamilton-Jacobi PDE on a grid representing a discretization of the state space. The work DeepReach [cite] solves this problem by using a modified PINN framework that uses periodic, or sinusoidal activation functions to solve this problem. The details of this problem will be shown in the following paragraphs.

Consider an agent in an environment in the presence of external disturbance. The agane tis modeled as a dynamics system with state $x \in \mathbb{R}^n$, control u, and disturbance d. The state evolves according to the dynamics:

$$\dot{x} = f(x, u, d), u \in \mathcal{U}, d \in \mathcal{D}.$$

Here, the disturbance could be either an external input or represent the model and environment uncertainty. Let $\xi_{x,t}^{u,d}(\tau)$ denote the state achieved at time τ by starting at initial state and initial time and applying input functions $u(\cdot)$ and $d(\cdot)$ over $[t, \tau]$. The target set $\mathcal{L}$ that is crucial to the agent is contained in the environment. This target set can be either a set of goal states or a set of unsafe states. The work DeepReach mainly concerns about computing the BRT and the Backward Reach-Avoid Tube (BRAT) of $\mathcal{L}$.

Previously we know the definition of BRT, which is the set of initial states that will eventually reach the target set with the time horizon $[t, T]$. Mathematically, it can be represented as

$$\mathcal{V}(t) = \Big\{x : \forall u(\cdot), \exists d(\cdot), \exists \tau \in [t, T], \xi_{x,t}^{u,d}(\tau) \in \mathcal{L}\Big\}$$

The definition of BRAT is slightly different. When the target set represents the goal states, BRAT is the set of initial states of the agent from which it can eventually reach the target set while always avoiding some unsafe set of states $\mathcal{G}$. Mathematically,

$$\begin{aligned} \mathcal{V}(t) = \Big\{x : \forall d(\cdot), \exists u(\cdot), \forall s \in [t, T], \xi_{x,t}^{u,d}(s) \notin \mathcal{G} \\ \exists \tau \in [t, T], \xi_{x,t}^{u,d}(\tau) \in \mathcal{L}\Big\} \end{aligned}$$

If there is no constraint present in the system, the set is then reduced to a BRT.

The example that is used in the work is a collision avoidance problem between two identical vehicles. The dynamics between the two vehicles are:

$$\dot{x}_1 = -v_e + v_p \cos x_3 + \omega_e x_2$$

$$\dot{x}_2 = v_p \sin x_3 - \omega_e x_1$$

$$\dot{x}_3 = \omega_p - \omega_e$$

where x_1, x_2 represent the relative position between vehicles, and x_3 represents the relative heading. v_e and v_p are the linear velocities of the evader and pursuer respectively. These two velocities are equal and constant. Similarly, ω_p and ω_e denotes the angular velocities of two agents and are the disturbance and input in the system respectively. The BRT we are interested in computing is

$$\mathcal{L} = \{x : \|(x_1, x_2)\| \leq \beta\}$$

Computing the BRT or BRAT is through Hamilton-Jacobi reachability. Firstly, a target function $l(x)$ is defined whose sub-zero level set is the target set $\mathcal{L}$. The BRT aims to compute all states that could enter $\mathcal{L}$ at any time point. This is accomplished by finding the minimum distance to $\mathcal{L}$ over time:

$$J(x, t, u(\cdot), d(\cdot)) = \min_{\tau \in [t,T]} l\left(\xi_{x,t}^{u,d}(\tau)\right)$$

The ultimate goal is to capture this minimum distance for optimal trajectories of the system. Therefore, we then compute the optimal control that maximizes this distance and the worstcase disturbance signal that minimizes the distance. The value function corresponding to this control problem is:

$$V(x,t) = \inf_{d(\cdot)} \sup_{u(\cdot)} \{J(x,t,u(\cdot),d(\cdot))\}$$

Computing through dynamic programming, we end with a final value Hamilton-Jacobi-Isaacs Variational Inequality:

$$\min\{D_t V(x,t) + H(x,t), l(x) - V(x,t)\} = 0$$

$$V(x,T) = l(x)$$

Here D_t represents the time gradients of the value function. H is the Hamiltonian and can be written as:

$$H(x,t) = \max_{u} \min_{d} \langle \nabla V(x,t), f(x,u,d) \rangle$$

Once we compute the value function, the BRT is then given as the sub-zero level set of the value function:

$$\mathcal{V}(t) = \{x : V(x,t) \leq 0\}$$

The corresponding safety control can be written as

$$u^*(x,t) = \arg \max_{u} \min_{d} \langle \nabla V(x,t), f(x,u,d) \rangle$$

Applying this optimal control at the BRT boundary and make sure the starting point is outside the BRT can ensure the system maintaining safety with any control applied.

Another main problem here is how to solve HJI VI. Typically, solving this involves computing value function over a grid representing a discretization of state space and time. However, this will result in an exponential scaling complexity both in computation and memory and limits its direct use to low-dimensional systems. Therefore, using the framework of PINNs can help to combat this challenge.

Recall that in a common PINN, the network's output is the predicted solution for the PDE we are solving, and we need to have a residual function for that PDE to represent the loss function. In this reachability problem, the situation is quite different. The input is a state vector x and a time point t, which is the same as the common case. The output here is the corresponding value $V_\theta(x,t)$, and θ represents the parameters of the neural network. Specifically, given an input (x_i, t_i), the loss function $h\left(x_i, t_i; \theta\right)$ for training the neural network is given as:

$$
\begin{aligned}
& h\left(x_i, t_i; \theta\right) = h_1\left(x_i, t_i; \theta\right) + \lambda h_2\left(x_i, t_i; \theta\right) \\
& h_1\left(x_i, t_i; \theta\right) = \left\|V_\theta\left(x_i, t_i\right) - l\left(x_i\right)\right\| \nVdash \left(t_i = T\right) \\
& h_2\left(x_i, t_i; \theta\right) = \| \min \left\{D_t V_\theta\left(x_i, t_i\right) + H\left(x_i, t_i\right)\right. \\
& \left. l\left(x_i\right) - V_\theta\left(x_i, t_i\right)\right\} \|.
\end{aligned}
$$

From the equation, we can see that the loss function depends on the time and spatial gradients of the value function. Therefore, we expect the network to not only represent the value function accurately but also its gradients. In practice, the ReLU-based network fails to represent the gradients well, thus leading to

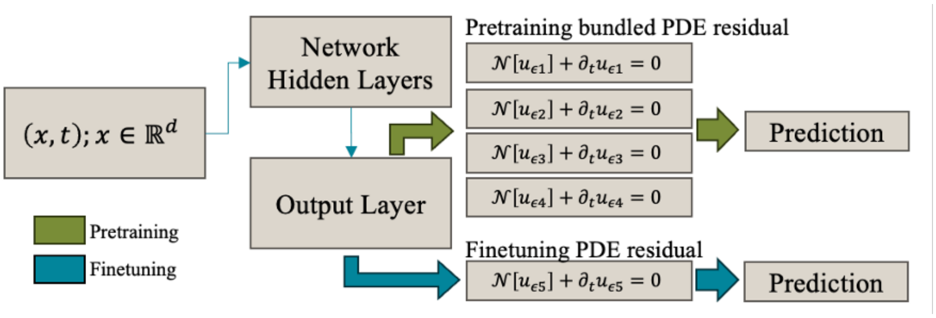

$(x,t); x \in \mathbb{R}^d$
Network Hidden Layers
Output Layer
Pretraining
Finetuning
Pretraining bundled PDE residual
$\mathcal{N}[u_{\epsilon 1}] + \partial_t u_{\epsilon 1} = 0$
$\mathcal{N}[u_{\epsilon 2}] + \partial_t u_{\epsilon 2} = 0$
$\mathcal{N}[u_{\epsilon 3}] + \partial_t u_{\epsilon 3} = 0$
$\mathcal{N}[u_{\epsilon 4}] + \partial_t u_{\epsilon 4} = 0$
Prediction
Finetuning PDE residual
$\mathcal{N}[u_{\epsilon 5}] + \partial_t u_{\epsilon 5} = 0$
Prediction

Figure 2.7: Structure of Transfer Learning of Physics-Informed Neural Networks. Hidden layers are fixed during finetuning and only the output layer can be modified

poor estimation of the value function. The solution used is to adopt a sinusoidal activation function. Therefore, the network is then able to learn a highly accurate approximation of the value function.

Chapter 3

Methodology

3.1 Transfer Learning of PINNs

Problem Formulation With our knowledge about the solving reachability problem in autonomous systems using the framework of PINNs from previous sections, we can conclude that this solution is promising and bearing with many advantages over traditional methods. One point that is worth noticing is that this solution trains a network from scratch. To solve a problem involving three or more agents, the training time can take up to more than one day. Furthermore, the solution that can be obtained from this well-trained network is for a particular HJI PDE. If the condition that influences this autonomous system changes, some parameters in the PDE will also be changed, causing the network to fail in predicting a correct solution for the new PDE, and thus being unable to solve the reachability problem. Therefore, we consider a solution that pre-trains a model with multiple conditions as training objectives, and only finetune the model when it is actually used by the autonomous system. Therefore, the problem here becomes applying transfer learning to PINNs.

We start introducing the method by defining a neural network that approximate network solution $\psi(x,t)$ at time points t is $\psi(x,t) = H(x,t)_{\theta_H} W_{\theta_W} + B_{\theta_B}$, where $H \in \mathbb{R}^{(x,t)\times h}$. To be more straightforward, the neural network is parameterized by $\theta = [\theta_H, \theta_W, \theta_B]$. To train this network, the final weight layer is designed to have multiple outputs such that $W_{\theta_W} \in \mathbb{R}^{h\times q}$. The network is designed such that multiple (q) solutions, $\psi(x,t) \in \mathbb{R}^{(x,t)\times q}$, can be estimated and simultaneously trained to satisfy equations that have different linear operators defined by different coefficients as well as different initial conditions. This training paradigm is like putting multiple equations into one bundle. Such bundle training allows us to achieve two goals: 1 . We can integrate the training of multiple equations into a single network; 2. We can encourage the hidden networks $H(x,t)$ to be versatile across equations. Optimization of this network involves using a special form of loss function. For an original PINN, the loss function is formulated by:

$$\mathcal{L} = \mathcal{L}_{\text{diffeq}} + \mathcal{L}_{IC} + \mathcal{L}_{BC}$$

$$\mathcal{L}_{\text{diffeq}} = D_t\left[u_{\varepsilon}(x,t)\right] + \mathcal{N}\left[u_{\varepsilon}(x,t)\right]$$

where $\varepsilon \epsilon \{\varepsilon_1, \varepsilon_2, \ldots, \varepsilon_q\}$ is the one element from a parameter set A of this PDE. The number of this parameter set is the same as the number of solutions that this network can give.

After this network is trained to a certain convergence condition, we will then proceed to finetune. At the finetuning stage, the weights for the hidden layers, or fixed layers, are frozen and the hidden layer's output is computed at specific input $\hat{t}$. The solution is then $\psi(\hat{t}) = H(\hat{x}, \hat{t})W_{\text{out}}$ where the W_{out} of the output layer is trainable and $H(\hat{x}, \hat{t}) \in \mathbb{R}^{(x,t)\times 1}$. A schematic demonstration of this framework is shown in Figure 3. To finetune W_{out} , we will still use the same form of loss functions as before, but the parameter set here, denoted as B, satisfies the condition that $A \cap B = \emptyset$. In this case, we are only transferring knowledge we learned from PDEs that has only one parameter being altered. To enhance the capability of this method, we propose to include multiple sets for each parameter in a PDE. Given a PDE's residual function:

$$\omega u_t(x,t) + \varepsilon u_x(x,t) = 0$$

Here the parameters that could be changed for this family of PDEs are ω and ε. By formulating these two parameters' set as vectors and plug directly into the PDE residual function, we will be able to have a loss function that can train over a family of PDEs with multiple varying parameters.

3.1.1 Poisson Equation

Here we will first use a widely used PDE, Poisson equation, to demonstrate the most fundamental idea of this method. The Poisson equation is typically used to identify an electrostatic potential ψ given a charge distribution ρ. In $2-\mathrm{D}$, it is described by:

$$\frac{\partial^2\psi}{\partial x^2}+\frac{\partial^2\psi}{\partial y^2}=\rho(x,y)$$

This PDE is defined in the domain $x\in[x_L,x_R]$ and $y\in[y_B,y_T]$ with BCs:

$$\psi\left(x_L,y\right)=\psi\left(x_R,y\right)=\psi\left(x,y_B\right)=\psi\left(x,y_T\right)=0$$

The charge distribution is given by:

$$\rho(x,y)=\sin(k\pi x)\sin(k\pi y)$$

For different Poisson equations, the parameter that could be changed is k. Therefore, applying the idea of bundle training to this PDE, we can train the network on four different charge distributions for $k\in 1,2,3,4$. Then we can evaluate the performance of the network on a harder testing force function of the form:

$$\rho_{\text{test }}=\frac{1}{4}\sum_{k=1}^{4}(-1)^{k+1}2k\sin(k\pi x)\sin(k\pi y)$$

The result will be shown in the next section, and from the result we can see the bundle pretraining and finetuning can successfully make the network to solve PDEs that it has never seen before.

3.1.2 Reachability Problem

Now I will discuss how this method is applied to the reachability problem we have already seen. Recall that the loss function $h\left(x_i, t_i; \theta\right)$ is described by:

$$
\begin{aligned}
&h\left(x_i, t_i; \theta\right) = h_1\left(x_i, t_i; \theta\right) + \lambda h_2\left(x_i, t_i; \theta\right), \\
&h_1\left(x_i, t_i; \theta\right) = \left\|V_\theta\left(x_i, t_i\right) - l\left(x_i\right)\right\| \nVdash \left(t_i = T\right), \\
&h_2\left(x_i, t_i; \theta\right) = \| \min\left\{D_t V_\theta\left(x_i, t_i\right) + H\left(x_i, t_i\right),\right. \\
&\left. l\left(x_i\right) - V_\theta\left(x_i, t_i\right)\right\} \|.
\end{aligned}
$$

To let the autonomous system to be able to find the optimal control and also ensure its safety under various influences, we need to include different Hamiltonian $H\left(x_i, t_i\right)$ into this loss function and perform bundle training to the network. Previously, we only knew the dynamics between two vehicles. The Hamiltonian can be derived from the dynamics as

$$
\begin{aligned}
H(x, t) = p_1\left(-v_e + v_p \cos x_3\right) + p_2\left(v_p \sin x_3\right) \\
- \bar{\omega}\left\|p_1 x_2 - p_2 x_1 - p_3\right\| + \bar{\omega} p_3
\end{aligned}
$$

Similar to the previous example, we want to create a set of parameters so that we can then have multiple loss functions for us to perform bundle training. However, in this Hamiltonian, there are two parameter that could describe the influence on the vehicles: v_e (or v_p, since they are always equal), and $\bar{\omega}$. In this case, creating one set of parameters, though could help the model to handle more cases, is not sufficiently generalizable because we cannot assume the other influence will not change in real cases. Therefore, the approach we adopted here is we create two set of parameters $v_e \in v_{e1}, v_{e2}, v_{e3}, v_{e4}$ and $\bar{\omega} \in \bar{\omega}_1, \bar{\omega}_2, \bar{\omega}_3, \bar{\omega}_4$. Here we only create a set with four different parameters just for demonstration. In practice, the number of parameters in the set can be more. The influence of including more parameters will be discussed in the result section. Finally, the loss function becomes:

$$h\left(x_i, t_i; \theta\right) = h_1\left(x_i, t_i; \theta\right) + \lambda h_2\left(x_i, t_i; \theta\right),$$

$$h_1\left(x_i, t_i; \theta\right) = \left\|V_\theta\left(x_i, t_i\right) - l\left(x_i\right)\right\| \nVdash \left(t_i = T\right),$$

$$h_2\left(x_i, t_i; \theta\right) = \| \min \Big\{ D_t V_\theta\left(x_i, t_i\right) + \left[H_1\left(x_i, t_i\right), H_2\left(x_i, t_i\right), H_3\left(x_i, t_i\right), H_4\left(x_i, t_i\right)\right]^T,$$

$$l\left(x_i\right) - V_\theta\left(x_i, t_i\right)\} \|$$

After the model is pre-trained on this bundle of loss functions, we can leverage this model to perform finetuning and solve more cases in much less time than training a model from scratch.

3.2 Finetuning Analytic Gradient Computation

Computing hardware deployed in most autonomous systems has a lot of limitations compared with modern GPUs. For instance, the edge hardware does not possess enough memory space to store large amount of data. This fact makes the normal training of a deep neural network on such hardware infeasible as the backward propagation requires the hardware to store all the intermediate result. Besides, most edge hardware does not have strong computing units like GPU and can be extremely slow in handling large scale high dimensional matrix computation. Another limitation is that existing compliers and libraries used in deep learning like PyTorch, TensorFlow, etc., are not supported on a lot of edge hardware. To overcome these limitations, we propose to leverage the second phase of transfer learning we discussed before: Finetuning. Same as the techniques mentioned in previous section, at finetuning stage, only the output layer's parameters are allowed to be optimized, which means we only need to perform backward propagation for one layer, and this process does not require the hardware to store any intermediate result at all. Meanwhile, the matrix computation associated with optimizing only one layer will not bring too much computation overhead for the hardware, and thus will not decrease the efficiency significantly. By using the analytical gradient computation method, we also can get rid of our dependency on existing libraries for performing computations related to backward propagation. We show this process by using an example PDE.

Consider a Hamilton-Jacobi-Bellman equation that:

$$\partial_t u_\theta(x,t) + \Delta u(x,t) - \mu \left\| \nabla_x u_\theta(x,t) \right\|^2 = a$$

For simplicity, we are considering a case where boundary conditions are not involved, and only the terminal condition is considered. Here $\|\cdot\|_p$ denotes an l_p norm. This PDE's residual function can be written as

$$R(\theta) = \partial_t u_\theta(x,t) + \Delta u(x,t) - \mu \left\|\nabla_x u_\theta(x,t)\right\|^2 - a$$

By writing out each part of the partial derivatives in the finite difference form, we can the have the expressions:

$$\begin{aligned} \partial_t u_\theta(x,t) &= \frac{u_\theta(x,t+\Delta t) + u_\theta(x,t-\Delta t)}{2\Delta t} \\ \Delta u_\theta &= \frac{u_\theta\left(x+\Delta x_1,t\right) + u_\theta\left(x-\Delta x_1,t\right) - 2u_\theta(x,t)}{\Delta x_1^2} + \cdots \\ &\quad + \frac{u_\theta(x,t+\Delta t) + u_\theta(x,t-\Delta t) - 2u_\theta(x,t)}{\Delta t^2}, \cdots \Bigg) \\ \nabla_x u_\theta(x,t) &= \Bigg(\frac{u_\theta\left(x+\Delta x_1 t\right) - u_\theta\left(x-\Delta x_1,t\right)}{2\Delta x_1} \end{aligned}$$

These three equations give us the expression for computing each part of the residual function without the need of using any differential operations and we only need to evaluate the model's output. We can then update the model's parameter by computing the gradient of loss with respect to the model's current parameter. In practical training of PINNs, the loss function is typically given by:

$$\mathcal{L}(\theta) = \frac{1}{N}\sum_{i=0}^{N} \|R(\theta)\|^2$$

Therefore, the gradient of loss with respect to model's parameters can be computed as:

$$\frac{\partial \mathcal{L}}{\partial \theta} = 2R(\theta)\frac{\partial R(\theta)}{\partial \theta}$$

From the previous section, we know that in this framework, the network is divided into hidden layers and output layer. Therefore, we can rewrite the output of the entire network as:

$$u_\theta(x,t) = w_\theta N(x,t) + b_\theta$$

where $N(x,t)$ denotes the hidden layers' output, w_θ and b_θ represent parameters in the hidden layer. Combining the equation for gradient of loss with respect to the parameters with the equation for each term in the residual function, we can then have a new expression for the first term as:

$$\frac{\partial}{\partial\theta}\left(\partial_t u_\theta(x,t)\right) = \frac{\partial}{\partial\theta}\left(\frac{u_\theta(x,t+\Delta t) + u_\theta(x,t-\Delta t)}{2\Delta t}\right)$$

$$\frac{\partial}{\partial\theta}\left(\partial_t u_\theta(x,t)\right) = \frac{\frac{\partial u_\theta(x,t+\Delta t)}{\partial\theta} + \frac{\partial u_\theta(x,t-\Delta t)}{\partial\theta}}{2\Delta t}$$

According to the equation for the output of the entire network, we can easily rewrite the equation as:

$$\frac{\partial u_\theta(x,t+\Delta t)}{\partial\theta} = N(x,t+\Delta t)$$

And this same form can be applied to all other terms as well. Finally, we will end up with the final equation for the gradient of loss with respect to the parameters as

$$
\begin{aligned}
&\frac{\partial R}{\partial \theta} = 2R(\theta)(\frac{N(x,t+\Delta t)+N(x,t-\Delta t)}{2\Delta t}+ \\
&\sum_{i=0}^{d}\left(\frac{N\left(x+\Delta x_i,t\right)-2N(x,t)}{\Delta x_i^2}\right)+\frac{N(x,t+\Delta t)}{\Delta t^2}- \\
&2\mu \| w\left(\frac{N\left(x+\Delta x_1,t\right)-N\left(x-\Delta x_1,t\right)}{2\Delta x_1}+\right. \\
&\left.\cdots+\frac{N\left(x+\Delta x_d,t\right)-N\left(x-\Delta x_d,t\right)}{2\Delta x_d}\right)+b\|\cdot \\
&\left(\frac{N\left(x+\Delta x_1,t\right)-N\left(x-\Delta x_1,t\right)}{2\Delta x_1},\ldots,\frac{N(x,t+\Delta t)-N(x,t-\Delta t)}{2\Delta t}\right))
\end{aligned}
\tag{3.1}
$$

After calculating the gradients, the parameters can be updated by:

$$\theta^* = \theta + \eta g$$

where η denotes a tunable learning rate. Iteratively computing will enable us to have a loss value that is smaller than a predefined threshold, where we consider to be a sufficiently accurate stopping point. From the experimental results shown in the next section, we can see that this process reaches the same level of accuracy compared with training from scratch, while saving more than 10 times of training time.

Chapter 4

Methodology

4.1 Transfer Learning of PINNs

Problem Formulation With our knowledge about the solving reachability problem in autonomous systems using the framework of PINNs from previous sections, we can conclude that this solution is promising and bearing with many advantages over traditional methods. One point that is worth noticing is that this solution trains a network from scratch. To solve a problem involving three or more agents, the training time can take up to more than one day. Furthermore, the solution that can be obtained from this well-trained network is for a particular HJI PDE. If the condition that influences this autonomous system changes, some parameters in the PDE will also be changed, causing the network to fail in predicting a correct solution for the new PDE, and thus being unable to solve the reachability problem. Therefore, we consider a solution that pre-trains a model with multiple conditions as training objectives, and only finetune the model when it is actually used by the autonomous system. Therefore, the problem here becomes applying transfer learning to PINNs.

We start introducing the method by defining a neural network that approximate network solution $\psi(x,t)$ at time points t is $\psi(x,t) = H(x,t)_{\theta_H} W_{\theta_W} + B_{\theta_B}$, where $H \in \mathbb{R}^{(x,t)\times h}$. To be more straightforward, the neural network is parameterized by $\theta = [\theta_H, \theta_W, \theta_B]$. To train this network, the final weight layer is designed to have multiple outputs such that $W_{\theta_W} \in \mathbb{R}^{h\times q}$. The network is designed such that multiple (q) solutions, $\psi(x,t) \in \mathbb{R}^{(x,t)\times q}$, can be estimated and simultaneously trained to satisfy equations that have different linear operators defined by different coefficients as well as different initial conditions. This training paradigm is like putting multiple equations into one bundle. Such bundle training allows us to achieve two goals: 1 . We can integrate the training of multiple equations into a single network; 2. We can encourage the hidden networks $H(x,t)$ to be versatile across equations. Optimization of this network involves using a special form of loss function. For an original PINN, the loss function is formulated by:

$$\mathcal{L} = \mathcal{L}_{\text{diffeq}} + \mathcal{L}_{IC} + \mathcal{L}_{BC}$$

$$\mathcal{L}_{\text{diffeq}} = D_t \left[u_{\varepsilon}(x,t)\right] + \mathcal{N}\left[u_{\varepsilon}(x,t)\right]$$

where $\varepsilon \epsilon \{\varepsilon_1, \varepsilon_2, \ldots, \varepsilon_q\}$ is the one element from a parameter set A of this PDE. The number of this parameter set is the same as the number of solutions that this network can give.

After this network is trained to a certain convergence condition, we will then proceed to finetune. At the finetuning stage, the weights for the hidden layers, or fixed layers, are frozen and the hidden layer's output is computed at specific input $\hat{t}$. The solution is then $\psi(\hat{t}) = H(\hat{x}, \hat{t})W_{\text{out}}$ where the W_{out} of the output layer is trainable and $H(\hat{x}, \hat{t}) \in \mathbb{R}^{(x,t)\times 1}$. A schematic demonstration of this framework is shown in Figure 3. To finetune W_{out} , we will still use the same form of loss functions as before, but the parameter set here, denoted as B, satisfies the condition that $A \cap B = \emptyset$. In this case, we are only transferring knowledge we learned from PDEs that has only one parameter being altered. To enhance the capability of this method, we propose to include multiple sets for each parameter in a PDE. Given a PDE's residual function:

$$\omega u_t(x,t) + \varepsilon u_x(x,t) = 0$$

Here the parameters that could be changed for this family of PDEs are ω and ε. By formulating these two parameters' set as vectors and plug directly into the PDE residual function, we will be able to have a loss function that can train over a family of PDEs with multiple varying parameters.

4.1.1 Poisson Equation

Here we will first use a widely used PDE, Poisson equation, to demonstrate the most fundamental idea of this method. The Poisson equation is typically used to identify an electrostatic potential ψ given a charge distribution ρ. In $2-\mathrm{D}$, it is described by:

$$\frac{\partial^2\psi}{\partial x^2}+\frac{\partial^2\psi}{\partial y^2}=\rho(x,y)$$

This PDE is defined in the domain $x \in [x_L, x_R]$ and $y \in [y_B, y_T]$ with BCs:

$$\psi\left(x_L, y\right)=\psi\left(x_R, y\right)=\psi\left(x, y_B\right)=\psi\left(x, y_T\right)=0$$

The charge distribution is given by:

$$\rho(x,y)=\sin(k\pi x)\sin(k\pi y)$$

For different Poisson equations, the parameter that could be changed is k. Therefore, applying the idea of bundle training to this PDE, we can train the network on four different charge distributions for $k \in 1,2,3,4$. Then we can evaluate the performance of the network on a harder testing force function of the form:

$$\rho_{\text{test}} = \frac{1}{4}\sum_{k=1}^{4}(-1)^{k+1}2k\sin(k\pi x)\sin(k\pi y)$$

The result will be shown in the next section, and from the result we can see the bundle pretraining and finetuning can successfully make the network to solve PDEs that it has never seen before.

4.1.2 Reachability Problem

Now I will discuss how this method is applied to the reachability problem we have already seen. Recall that the loss function $h\left(x_i, t_i; \theta\right)$ is described by:

$$
\begin{aligned}
& h\left(x_i, t_i; \theta\right) = h_1\left(x_i, t_i; \theta\right) + \lambda h_2\left(x_i, t_i; \theta\right), \\
& h_1\left(x_i, t_i; \theta\right) = \left\| V_\theta\left(x_i, t_i\right) - l\left(x_i\right) \right\| \nVdash \left(t_i = T\right), \\
& h_2\left(x_i, t_i; \theta\right) = \left\| \min\left\{ D_t V_\theta\left(x_i, t_i\right) + H\left(x_i, t_i\right), \right.\right. \\
& \left.\left. l\left(x_i\right) - V_\theta\left(x_i, t_i\right) \right\} \right\|.
\end{aligned}
$$

To let the autonomous system to be able to find the optimal control and also ensure its safety under various influences, we need to include different Hamiltonian $H\left(x_i, t_i\right)$ into this loss function and perform bundle training to the network. Previously, we only knew the dynamics between two vehicles. The Hamiltonian can be derived from the dynamics as

$$
\begin{aligned}
H(x,t) = {} & p_1\left(-v_e + v_p \cos x_3\right) + p_2\left(v_p \sin x_3\right) \\
& - \bar{\omega} \left\| p_1 x_2 - p_2 x_1 - p_3 \right\| + \bar{\omega} p_3
\end{aligned}
$$

Similar to the previous example, we want to create a set of parameters so that we can then have multiple loss functions for us to perform bundle training. However, in this Hamiltonian, there are two parameter that could describe the influence on the vehicles: v_e (or v_p, since they are always equal), and $\bar{\omega}$. In this case, creating one set of parameters, though could help the model to handle more cases, is not sufficiently generalizable because we cannot assume the other influence will not change in real cases. Therefore, the approach we adopted here is we create two set of parameters $v_e \in v_{e1}, v_{e2}, v_{e3}, v_{e4}$ and $\bar{\omega} \in \bar{\omega}_1, \bar{\omega}_2, \bar{\omega}_3, \bar{\omega}_4$. Here we only create a set with four different parameters just for demonstration. In practice, the number of parameters in the set can be more. The influence of including more parameters will be discussed in the result section. Finally, the loss function becomes:

$$h\left(x_i, t_i; \theta\right) = h_1\left(x_i, t_i; \theta\right) + \lambda h_2\left(x_i, t_i; \theta\right),$$

$$h_1\left(x_i, t_i; \theta\right) = \left\|V_\theta\left(x_i, t_i\right) - l\left(x_i\right)\right\| \nVdash \left(t_i = T\right),$$

$$h_2\left(x_i, t_i; \theta\right) = \| \min \Big\{ D_t V_\theta\left(x_i, t_i\right) + \left[H_1\left(x_i, t_i\right), H_2\left(x_i, t_i\right), H_3\left(x_i, t_i\right), H_4\left(x_i, t_i\right)\right]^T,$$

$$l\left(x_i\right) - V_\theta\left(x_i, t_i\right)\} \|$$

After the model is pre-trained on this bundle of loss functions, we can leverage this model to perform finetuning and solve more cases in much less time than training a model from scratch.

4.2 Finetuning Analytic Gradient Computation

Computing hardware deployed in most autonomous systems has a lot of limitations compared with modern GPUs. For instance, the edge hardware does not possess enough memory space to store large amount of data. This fact makes the normal training of a deep neural network on such hardware infeasible as the backward propagation requires the hardware to store all the intermediate result. Besides, most edge hardware does not have strong computing units like GPU and can be extremely slow in handling large scale high dimensional matrix computation. Another limitation is that existing compliers and libraries used in deep learning like PyTorch, TensorFlow, etc., are not supported on a lot of edge hardware. To overcome these limitations, we propose to leverage the second phase of transfer learning we discussed before: Finetuning. Same as the techniques mentioned in previous section, at finetuning stage, only the output layer's parameters are allowed to be optimized, which means we only need to perform backward propagation for one layer, and this process does not require the hardware to store any intermediate result at all. Meanwhile, the matrix computation associated with optimizing only one layer will not bring too much computation overhead for the hardware, and thus will not decrease the efficiency significantly. By using the analytical gradient computation method, we also can get rid of our dependency on existing libraries for performing computations related to backward propagation. We show this process by using an example PDE.

Consider a Hamilton-Jacobi-Bellman equation that:

$$\partial_t u_\theta(x,t) + \Delta u(x,t) - \mu \left\| \nabla_x u_\theta(x,t) \right\|^2 = a$$

For simplicity, we are considering a case where boundary conditions are not involved, and only the terminal condition is considered. Here $\|\cdot\|_p$ denotes an l_p norm. This PDE's residual function can be written as

$$R(\theta) = \partial_t u_\theta(x,t) + \Delta u(x,t) - \mu \left\| \nabla_x u_\theta(x,t) \right\|^2 - a$$

By writing out each part of the partial derivatives in the finite difference form, we can the have the expressions:

$$\partial_t u_\theta(x,t) = \frac{u_\theta(x,t+\Delta t) + u_\theta(x,t-\Delta t)}{2\Delta t}$$

$$\Delta u_\theta = \frac{u_\theta\left(x+\Delta x_1,t\right) + u_\theta\left(x-\Delta x_1,t\right) - 2u_\theta(x,t)}{\Delta x_1^2} + \cdots$$

$$+\frac{u_\theta(x,t+\Delta t) + u_\theta(x,t-\Delta t) - 2u_\theta(x,t)}{\Delta t^2}, \cdots \Bigg)$$

$$\nabla_x u_\theta(x,t) = \Bigg(\frac{u_\theta\left(x+\Delta x_1 t\right) - u_\theta\left(x-\Delta x_1,t\right)}{2\Delta x_1}$$

These three equations give us the expression for computing each part of the residual function without the need of using any differential operations and we only need to evaluate the model's output. We can then update the model's parameter by computing the gradient of loss with respect to the model's current parameter. In practical training of PINNs, the loss function is typically given by:

$$\mathcal{L}(\theta) = \frac{1}{N}\sum_{i=0}^{N} \|R(\theta)\|^2$$

Therefore, the gradient of loss with respect to model's parameters can be computed as:

$$\frac{\partial \mathcal{L}}{\partial \theta} = 2R(\theta)\frac{\partial R(\theta)}{\partial \theta}$$

From the previous section, we know that in this framework, the network is divided into hidden layers and output layer. Therefore, we can rewrite the output of the entire network as:

$$u_\theta(x,t) = w_\theta N(x,t) + b_\theta$$

where $N(x,t)$ denotes the hidden layers' output, w_θ and b_θ represent parameters in the hidden layer. Combining the equation for gradient of loss with respect to the parameters with the equation for each term in the residual function, we can then have a new expression for the first term as:

$$\frac{\partial}{\partial\theta}\left(\partial_t u_\theta(x,t)\right) = \frac{\partial}{\partial\theta}\left(\frac{u_\theta(x,t+\Delta t) + u_\theta(x,t-\Delta t)}{2\Delta t}\right)$$

$$\frac{\partial}{\partial\theta}\left(\partial_t u_\theta(x,t)\right) = \frac{\frac{\partial u_\theta(x,t+\Delta t)}{\partial\theta} + \frac{\partial u_\theta(x,t-\Delta t)}{\partial\theta}}{2\Delta t}$$

According to the equation for the output of the entire network, we can easily rewrite the equation as:

$$\frac{\partial u_\theta(x,t+\Delta t)}{\partial\theta} = N(x,t+\Delta t)$$

And this same form can be applied to all other terms as well. Finally, we will end up with the final equation for the gradient of loss with respect to the parameters as

$$\begin{aligned}
&\frac{\partial R}{\partial \theta} = 2R(\theta)(\frac{N(x,t+\Delta t)+N(x,t-\Delta t)}{2\Delta t}+ \\
&\sum_{i=0}^{d}\left(\frac{N\left(x+\Delta x_i,t\right)-2N(x,t)}{\Delta x_i^2}\right)+\frac{N(x,t+\Delta t)}{\Delta t^2}- \\
&2\mu\|w\left(\frac{N\left(x+\Delta x_1,t\right)-N\left(x-\Delta x_1,t\right)}{2\Delta x_1}+\right. \\
&\left.\cdots+\frac{N\left(x+\Delta x_d,t\right)-N\left(x-\Delta x_d,t\right)}{2\Delta x_d}\right)+b\|\cdot \\
&\left(\frac{N\left(x+\Delta x_1,t\right)-N\left(x-\Delta x_1,t\right)}{2\Delta x_1},\ldots,\frac{N(x,t+\Delta t)-N(x,t-\Delta t)}{2\Delta t}\right))
\end{aligned} \tag{4.1}$$

After calculating the gradients, the parameters can be updated by:

$$\theta^* = \theta + \eta g$$

where η denotes a tunable learning rate. Iteratively computing will enable us to have a loss value that is smaller than a predefined threshold, where we consider to be a sufficiently accurate stopping point. From the experimental results shown in the next section, we can see that this process reaches the same level of accuracy compared with training from scratch, while saving more than 10 times of training time.

Chapter 5

Numerical Results

In this section, we will demonstrate our experiment and result for solving different PDEs and problems. We use the same hardware for both the pretraining and finetuning stage, and we record the computing time and memory usage in both stages to illustrate the effectiveness of this framework on edge devices. The GPU we used in all our experiment is NVIDIA GeForce RTX 3090, with 24 GB memory.

5.1 Poisson Equation

We firstly demonstrate our proposed method by solving the Poisson equation demonstrated before. The network we used here is an MLP that has five hidden layers with two input dimensions representing two spatial dimensions. The dimensions of the output for the pretraining and finetuning stage are different and will be discussed in detail. The width of each layer is 128. The activation function we used in the network is the sinusoidal activation function. The PDE is defined as:

$$\frac{\partial^2 \psi}{\partial x^2} + \frac{\partial^2 \psi}{\partial y^2} = \rho(x, y) \tag{5.1}$$

with $x \in [0, 1]$ and $y \in [0, 1]$. The charge distribution $\rho(x, y)$ is given by:

$$\rho(x, y) = \sin(k\pi x)\sin(k\pi y)$$

And the boundary condition is described by:

$$\psi(x, y) = 0, (x, y) \in \partial\Omega$$

Then, we can write the residual function as $\nabla^2\Psi - \sin(k\pi x)\sin(k\pi y) = 0$. Now let $u_\theta(x, y)$ be the network, to ensure the network's prediction not only is accurate, but also satisfies the boundary condition, we use a transformation to let the network meet the Dirichlet boundary condition:

$$\psi_\theta(x, y) = x(x - 1)y(y - 1)u_\theta(x, y).$$

In the pretraining stage, we set the output dimension to be four, representing the predicted solution for Poisson equations with different values of k in the charge distribution. Specifically, we let the values for k to be $[1, 2, 3, 4]$. For each value of k, we can formulate a residual function. Therefore, we will have four residual functions corresponding to four different values of k in the PDE. By substituting the output value from each output dimension of the network into each residual function and compute an averaged loss value across these four residual functions, we can then train the network based on its ability to approximate all four PDEs. We fed 1,000 two dimensional coordinates into the network and they are sampled randomly in the square domain bounded by $(0, 0)$ and $(1, 1)$.

For the pretraining stage, we used 10,000 iterations to train the model, and the optimizer we used is Adam [cite] from the PyTorch library. The learning rate at the beginning of the training is $1e - 3$ and will decay every 1,000 iterations by a factor of 0.9. After we finish with the pretraining stage, we then move to the finetuning stage where we load the model from the last iteration in the pretraining stage and replace the last layer with a single output layer that has randomly initialized parameters. While most of the settings are unchanged, we required only the last layer of the network to be trainable and frozen all other layers. Then we performed finetuning for 1000 iterations. In the finetuning stage, we set the value of k in the residual function to be 5, which is a number that has never been seen by the network. The loss performance for the pretraining stage and the finetuning stage are shown in figure 4.1 and figure 4.2 respectively.

To better visualize the performance of the transfer learning, we plot the heat map for the ground truth with $k = 5$, and the ground truth is given by the analytic

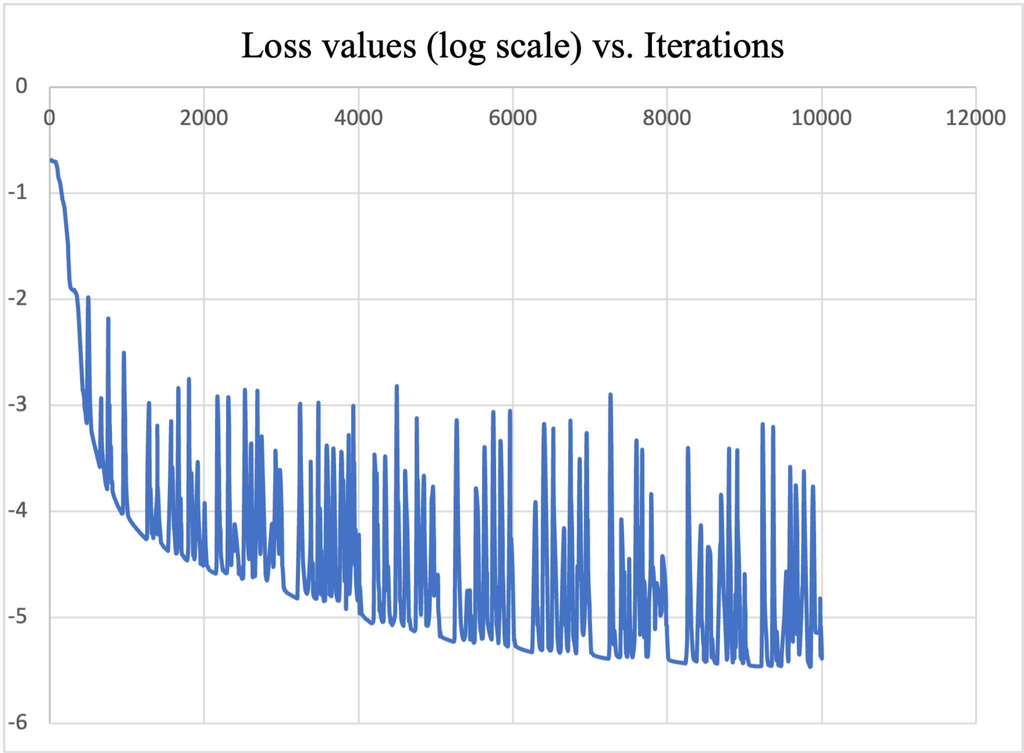


Figure 5.1: Pretraining loss values in log scale. y-axis is loss values in log scale, and x-axis represents iteration

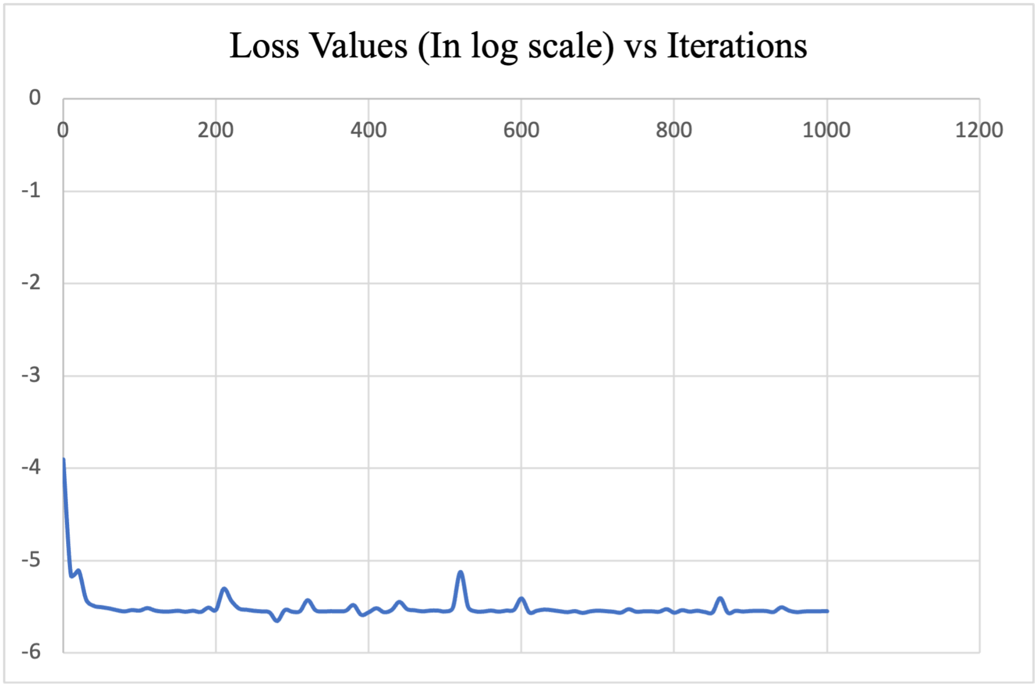


Figure 5.2: Finetuning loss values in log scale. y-axis is loss values in log scale, and x-axis represents iteration

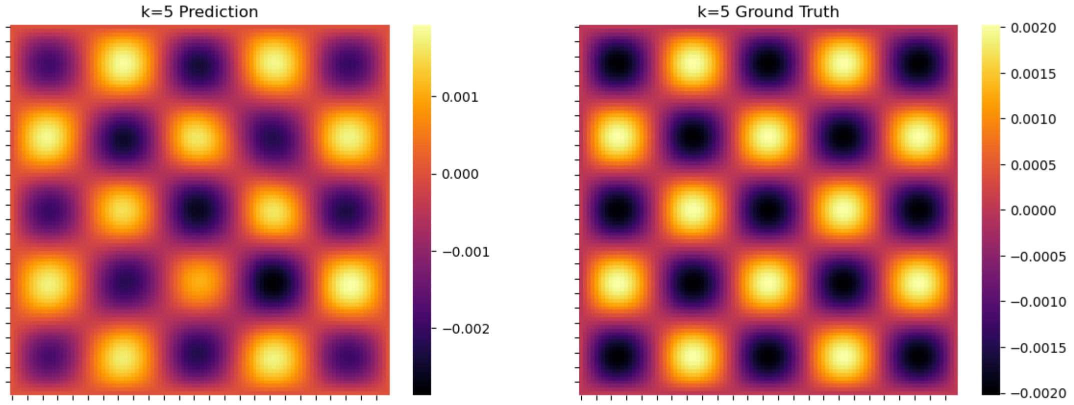


Figure 5.3: Predicted and ground truth solution for the Poisson equation at $k = \mathbf{5}$

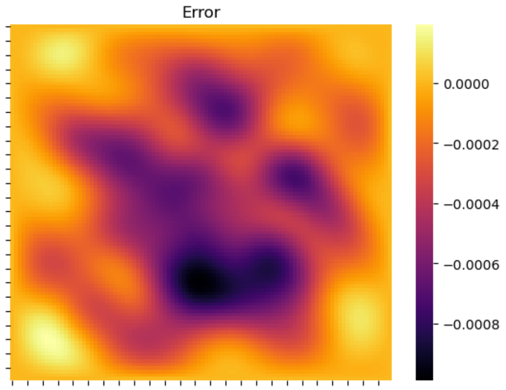


Figure 5.4: Error of the predicted result.

solution in this form:

$$\psi(x,y) = \frac{-1}{2(k\pi)^2}\sin(k\pi x)\sin(k\pi y)$$

We also include the heatmap for the predicted solution as well. The plot is shown in figure 4.3.

And the error between these two is also visualized by figure 4.4.

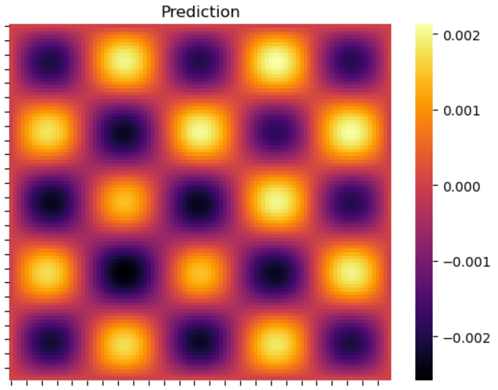


Figure 5.5: Predicted solution for the Poisson equation at k=5 by training a PINN from scratch.

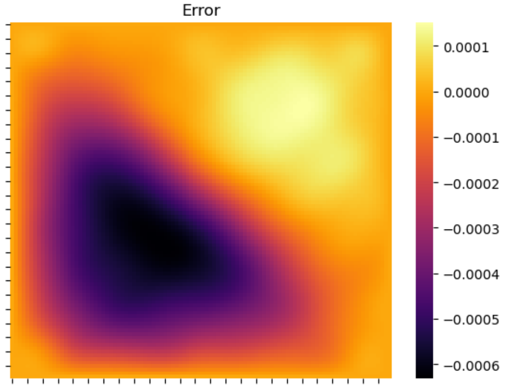


Figure 5.6: Error of the predicted solution for the PINN trained from scratch.

The mean value of the error in the whole field is 0.000280962 . If we train a PINN from scratch for $k = 5$, we will end up with a prediction heat map and error heat map shown in figure 4.5 and figure 4.6.

The average error of the field for training a PINN from scratch for 10,000 iterations is 0.0001916107 , which is at the same level and slightly lower compared with our transfer learning method. We also tried finetuning for other values of k

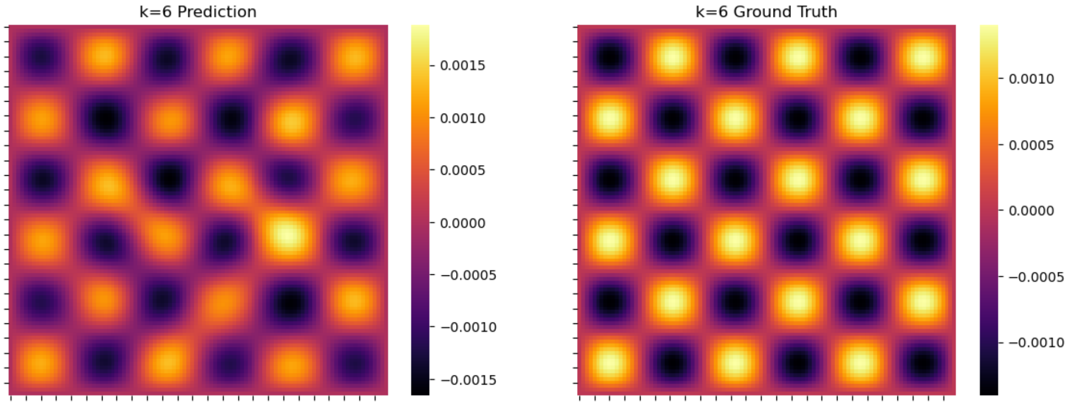


Figure 5.7: Predicted and ground truth solution for the Poisson equation at $k = \mathbf{6}$ after finetuning.

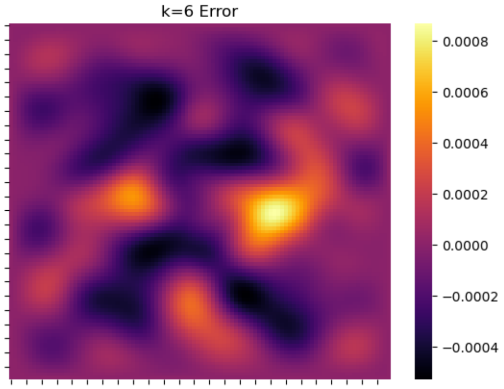


Figure 5.8: Error of solution for the Poisson equation at $k = \mathbf{6}$ after finetuning.

to evaluate the generalizability of this framework. The results are shown in figure 4.7 to 4.10

From all these results, we can see that the transfer learning framework can effectively learn the PDE given a pretrained model even though this PDE has some other parameters. Another advantage of finetuning is that it can be significantly faster when solving some new cases. In experiment, the time used for Finetuning a

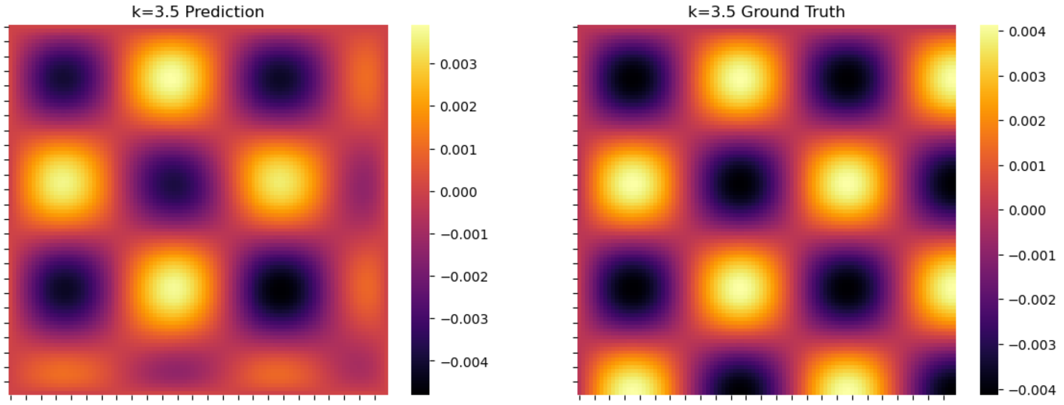


Figure 5.9: Predicted and ground truth solution for the Poisson equation at $k = \mathbf{3.5}$ after finetuning.

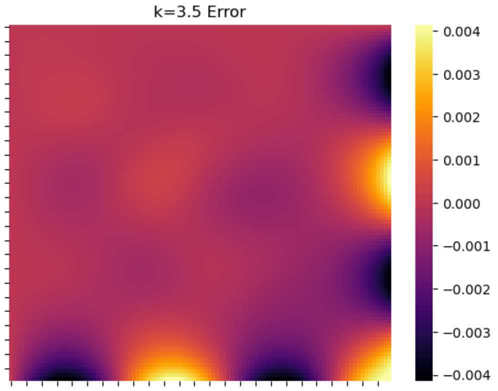


Figure 5.10: Error of solution for the Poisson equation at $k = \mathbf{6}$ after finetuning.

model for 1,000 iterations is 22.92 seconds, while training a PINN from scratch for 10,000 iterations to solve this Poisson equation requires more than five minutes. Overall, this framework is able to predict the solution for Poisson equations with different parameters fast and accurately.

5.2 Hamilton-Jacobi-Bellman Equation

Another example to demonstrate the capability of the proposed method is a Hamilton-Jacobi-Bellman (HJB) equation. The HJB equation is a cornerstone of modern control theory. It describes the condition of optimality for a control problem, offering a precise mathematical formulation to determine optimal strategies in systems that evolve over time. The HJB equation relates the value of an optimal policy at any given state to the maximum return achievable from that state. Typically, the HJB equations can be challenging to solve due to its complexity and the high dimensionality in practical applications. In different scenarios, the number of state variables, each represents an aspect of the system that can change over time and needs to be controlled, can be different and potentially high. Here the HJB equation we considered is a 21-dimensional one given as:

$$\partial_t u(\boldsymbol{x}, t) + \Delta u(\boldsymbol{x}, t) - \mu \left\| \nabla_{\boldsymbol{x}} u(\boldsymbol{x}, t) \right\|_2^2 = -2, \boldsymbol{x} \in [0,1]^{20}, t \in [0,1] \tag{5.2}$$

with final condition and domain given as:

$$u(x,1) = \|x\|_1, x \in [0,1]^{20}$$

Here $\|\cdot\|_p$ denotes the L_p norm for $p = 1, 2$. The exact solution is $u(\boldsymbol{x}, t) = \|\boldsymbol{x}\|_1 + 1 - t$ if $\mu = 0.01$. Training a traditional PINN for solving this equation can be time consuming as it involves computations for all the 21 dimensions. To make the PINN more efficient for solving the same equation with other values of μ, we then leverage the idea of transfer learning for this equation. Same as the previous example, we firstly choose multiple values for μ as $\mu \in [1, 1.2, 1.4, 1.6]$. As the equation has 21 dimensions, the mesh grids we create as the input to the network will also have 21 dimensions, and we have 1024 points in total. As the input dimension of the PINN should be equal to the dimension of the PDE itself, the PINN we used for the pretraining has input size as 21 , where the first 20 inputs denoting all the spatial coordinates and the last one denoting the temporal coordinate. The width and depth of the network is the same as before, which is 128 and 4 respectively. The output dimension, as we discussed before, should correspond to the number of parameters we choose to pretraining over, which will be 4 . The other hyper-parameters such as optimizer, learning rate, etc., are the same as the last example.

In the pretraining stage, we trained the model for 7,000 iterations and it takes about 5 hours. After that, we choose some other values for μ and finetune the

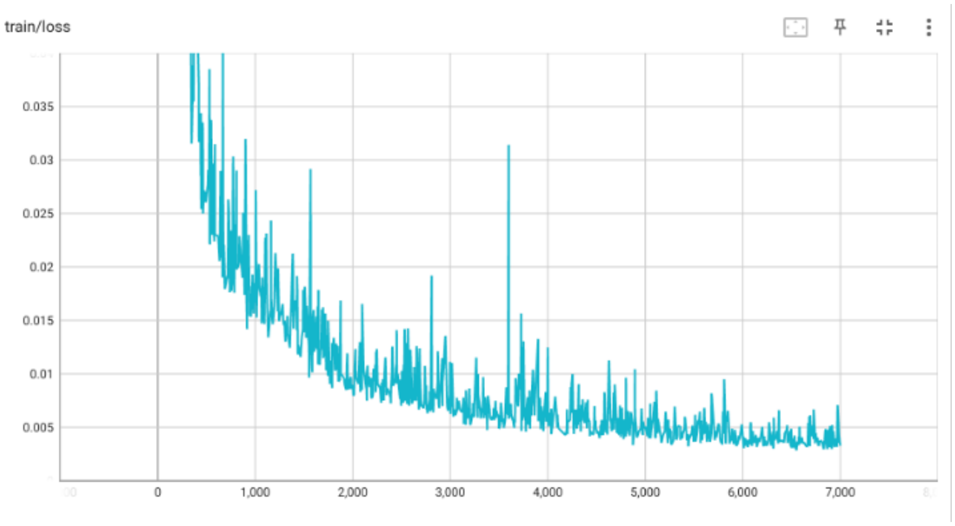


Figure 5.11: Pretraining training loss. The pretraining loss is calculated over $\mu \in [\mathbf{1}, \mathbf{1.2}, \mathbf{1.4}, \mathbf{1.6}]$

model with only 1,000 iterations. We use the vanilla PINN as a benchmark to demonstrate the effectiveness of the transfer learning. The performance of both stages is shown in figure 4.11 and 4.12.

By looking at the test loss of different tested values, we can see that it can reach the same accuracy level as the vanilla PINN, while saving more than 10x training time, and we can conclude that the transfer learning method can indeed solve various HJB equations efficiently and accurately than training a PINN from scratch.

As the current model relies heavily on existing tool libraries such as PyTorch to compute the gradient, and these libraries are commonly not available on some edge devices, we further demonstrate our proposed analytic computation of gradients

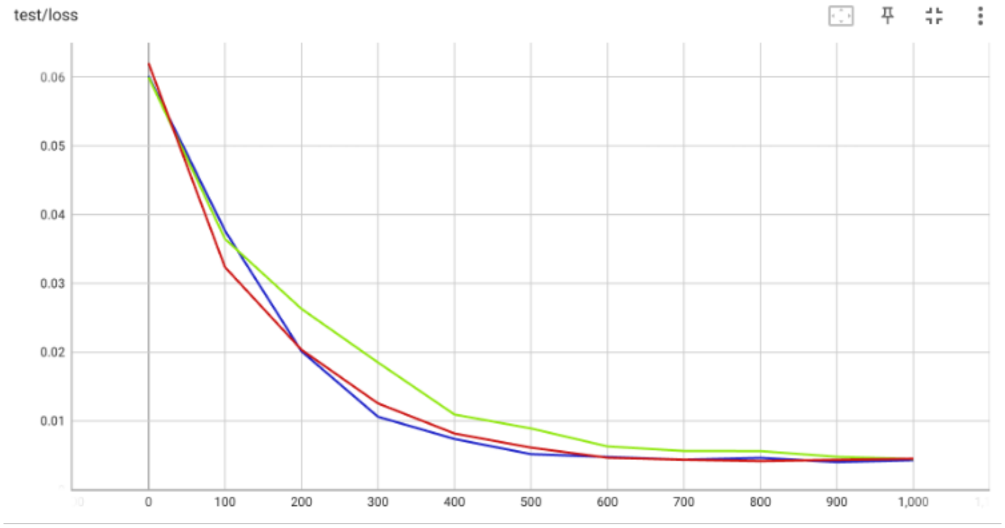


Figure 5.12: Finetuning test loss. The testing is performed for $\mu = \mathbf{1.33}$(blue line), $\mu = \mathbf{1.527}$(yellow line), and $\mu = \mathbf{1.423}$(red line).

so that we can perform finetuning on edge devices easily. To achieve this goal, we mainly solved two issues: 1. The calculation of the gradients of loss with respect to the model parameters requires existing libraries; 2. The differential operators involved in the PDE itself also requires tools from existing libraries. The first one can be solved with the previously mentioned analytic gradient computation. The solution to the second issue is using the central finite difference instead of backward propagation to compute the gradients associated with the differential operators. With the performance of the test loss shown in figure 4.13 and 4.14, we can see that we can still achieve the same finetuning performance without using existing libraries to compute the gradients.

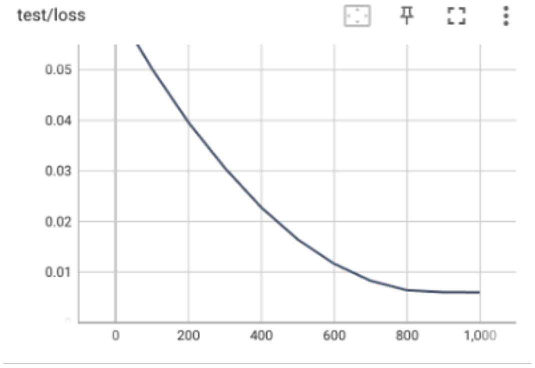


Figure 5.13: Test loss for $\mu = \mathbf{1.423}$ at finetuning stage using analytical gradient computation

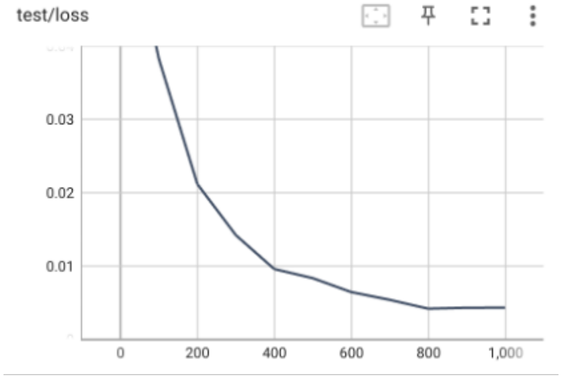


Figure 5.14: Test loss for $\mu = \mathbf{1.423}$ at finetuning stage using backward propagation

From the result, we can see that using analytic gradient computation method, we can still finetuning the model to the same accuracy level. This makes the model friendly to hardware that have limited access to existing deep learning libraries.

5.3 3-D Reachability Problem

With the demonstration from previous examples, we now investigate a more realistic problem that we introduced at the beginning: The 3-D reachability problem. Recall that the Hamiltonian for the 3-D reachability problem can be written as:

$$\begin{aligned} H(x,t) &= p_1\left(-v_e + v_p \cos x_3\right) + p_2\left(v_p \sin x_3\right) \\ &- \bar{\omega}\left\|p_1 x_2 - p_2 x_1 - p_3\right\| + \bar{\omega} p_3 \end{aligned} \tag{5.3}$$

To solve this equation, we will need to select values for the velocity and $\bar{\omega}$. For the situation where velocity equals to 0.75 and $\bar{\omega}$ equals to 3.0 , we can visualize the reachability using figure 4.15.

To validate the effectiveness of transfer learning for solving this problem, we design the experiment as follows. Firstly, we pretrain the model by choosing multiple parameters. We chose four values for velocity as $v_e \in [0.735, 0.745, 0.755, 0.765]$, and four values for $\bar{\omega}$ that $\bar{\omega} \in [2.85, 2.95, 3.05, 3.15]$. The purpose of choosing these values is that we can bypass using the values for the ground truth case. It

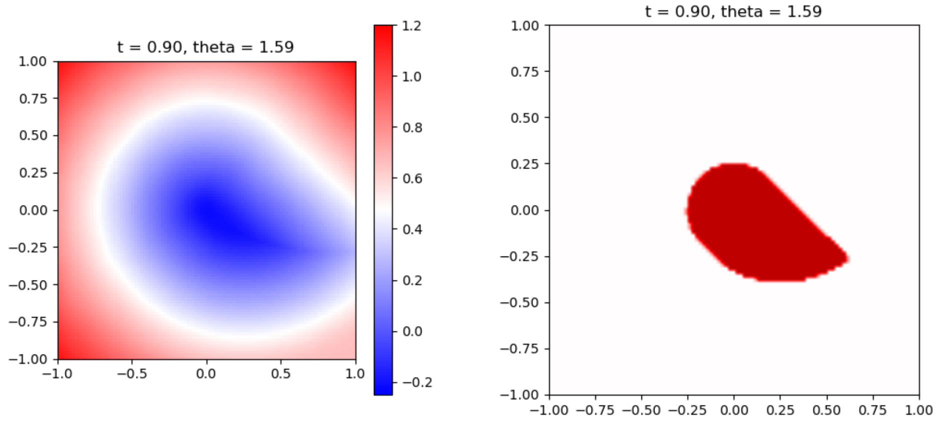


Figure 5.15: Ground truth solution for 3-D reachability problem. Left figure shows the solution space where the blue region denotes the reachable set. The right figure shows the overlap of this solution with the true solution. Since this is ground truth, thus it shows a fully overlap.

is obviously that our model will have 4 output dimensions. The input dimension for this problem is four as well since it includes three spatial dimensions and one temporal dimension. We perform 120,000 iterations of pretraining with 65,000 points. The optimizer we use for this case is Stochastic Gradient Descent (SGD). After pretraining, we leverage the saved model from last epoch of the pretraining stage and replace its last layer with a single output layer. Then we finetune the model for 10,000 iterations. The resulting solution from the model is shown in figure 4.16.

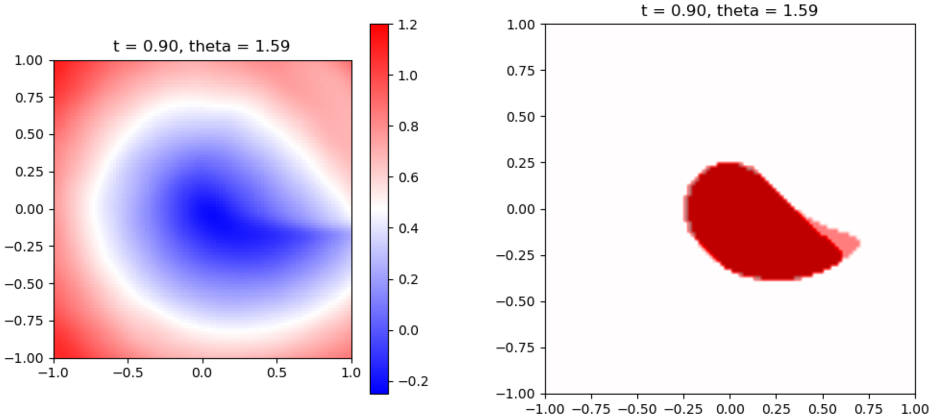


Figure 5.16: Predicted solution for 3-D reachability problem (Left). Comparison of predicted solution with ground truth (Right) where shaded area represents the difference.

From this result, we can conclude that the transfer learning meets our expectation that it can saves more than 10x training time for solving a new case, while the prediction accuracy is comparable with training from scratch.

Chapter 6

Discussion and Conlusion

Through the experiment, we validate using multiple examples that the transfer learning can work as desired for efficiently solving new PDEs given some prior knowledge about that PDE through pretraining. There are some points worth discussing from the experiment. The number of parameters that we used for pretraining will have a crucial influence on the entire framework. In the pretraining case, we typically select four different parameters as a pretraining bundle. In fact, the number of parameters we choose for pretraining bundle can have respective advantages and limitations. Choosing more parameters will require us to have more output dimensions at the last layer and will results in larger model size and more computation. The benefit of doing so is also obvious: as we feed more cases for pretraining, we can expect the network to be more generalizable since wider range of data can improve the performance of transfer learning [cite: Do Better ImageNet Models Transfer Better?]. From the other side, reducing the number of parameters in the pretraining bundle can help to save a lot of memory and computing time, but the performance of transfer learning will be limited if the target data distribution deviates much from the pretraining bundle. As a conclusion, we demonstrated the main idea and the capability of transfer learning using a relatively simple Poisson equation. The model after finetuning successfully solved a new Poisson equation with the same level of accuracy compared to training from scratch, and the finetuning also helps to save a significant amount of time as only 1/10 iterations is needed for finetuning. We further demonstrated the capability of transfer learning in solving more complicated PDEs like a 20-dimensional HJB equation. In this experiment, we also demonstrated the effectiveness of the analytic gradient computation method, which enables us to deploy this framework onto devices where common Python libraries are not available. In the last experiment, where we solved the 3-D reachability problem, we demonstrated that we could solve real world optimal control problems efficiently and accurately using this transfer learning framework. This work also leaves a lot of spaces for future work. This current hardware-friendly framework can only solve a single family of PDEs. Therefore, its usage could be very limited if the situation is more complicated. A more generalized framework is needed. This issue is especially important for the analytical gradient computation part. To analytically compute the gradients, we need to derive the expression according to the residual function.

Deriving such expression for a single family of PDEs can be easy, but to have a more generalized and automated system, we need to improve the way we find the exact gradient to make the framework more generalizable.

Bibliography

Ascher, Uri M, Steven J Ruuth, and Brian TR Wetton (1995). “Implicit-explicit methods for time-dependent partial differential equations”. In: *SIAM Journal on Numerical Analysis* 32.3, pp. 797–823.

Bansal, Somil and Claire J Tomlin (2021). “Deepreach: A deep learning approach to high-dimensional reachability”. In: *2021 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, pp. 1817–1824.

Biswas, BN et al. (2013). “A discussion on Euler method: A review”. In: *Electronic Journal of Mathematical Analysis and Applications* 1.2, pp. 2090–2792.

Desai, Shaan et al. (2021). “One-shot transfer learning of physics-informed neural networks”. In: *arXiv preprint arXiv:2110.11286*.

Herten, Rudolf LM van et al. (2022). “Physics-informed neural networks for myocardial perfusion MRI quantification”. In: *Medical Image Analysis* 78, p. 102399.

Jagtap, Ameya D et al. (2022). “Physics-informed neural networks for inverse problems in supersonic flows”. In: *Journal of Computational Physics* 466, p. 111402.

Kingma, Diederik P and Jimmy Ba (2014). “Adam: A method for stochastic optimization”. In: *arXiv preprint arXiv:1412.6980*.

Kutta, Wilhelm (1901). *Beitrag zur näherungsweisen Integration totaler Differentialgleichungen*. Teubner.

Lagaris, Isaac E, Aristidis Likas, and Dimitrios I Fotiadis (1998). “Artificial neural networks for solving ordinary and partial differential equations”. In: *IEEE transactions on neural networks* 9.5, pp. 987–1000.

Li, Jiaheng, Junchao Chen, and Biao Li (2022). “Gradient-optimized physics-informed neural networks (GOPINNs): a deep learning method for solving the complex modified KdV equation”. In: *Nonlinear Dynamics* 107, pp. 781–792.

Liu, Ziyue, Xinling Yu, and Zheng Zhang (2022). “TT-PINN: a tensor-compressed neural PDE solver for edge computing”. In: *arXiv preprint arXiv:2207.01751*.

Lu, Lu, Pengzhan Jin, and George Em Karniadakis (2019). “Deeponet: Learning nonlinear operators for identifying differential equations based on the universal approximation theorem of operators”. In: *arXiv preprint arXiv:1910.03193*.

Lu, Lu, Raphael Pestourie, et al. (2021). “Physics-informed neural networks with hard constraints for inverse design”. In: *SIAM Journal on Scientific Computing* 43.6, B1105–B1132.

Lütjens, Björn et al. (2021). “Pce-pinns: Physics-informed neural networks for uncertainty propagation in ocean modeling”. In: *arXiv preprint arXiv:2105.02939*.

Mowlavi, Saviz and Saleh Nabi (2023). “Optimal control of PDEs using physics-informed neural networks”. In: *Journal of Computational Physics* 473, p. 111731.

Noguer i Alonso, Miquel and Daniel Maxwell (2023). “Physics-Informed Neural Networks (PINNs) in Finance”. In: *Daniel, Physics-Informed Neural Networks (PINNs) in Finance (October 10, 2023)*.

Oldenburg, Jan et al. (2022). “Geometry aware physics informed neural network surrogate for solving Navier–Stokes equation (GAPINN)”. In: *Advanced Modeling and Simulation in Engineering Sciences* 9.1, p. 8.

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

RAY, DEEP PINTI, Orazio Pinti, and Assad A Oberai (2024). *DEEP LEARNING AND COMPUTATIONAL PHYSICS.* Springer.

Rosenblatt, Frank (1958). “The perceptron: a probabilistic model for information storage and organization in the brain.” In: *Psychological review* 65.6, p. 386.

Rumelhart, David E, Geoffrey E Hinton, and Ronald J Williams (1986). “Learning representations by back-propagating errors”. In: *nature* 323.6088, pp. 533–536.

Verwer, Jan G (1996). “Explicit Runge-Kutta methods for parabolic partial differential equations”. In: *Applied Numerical Mathematics* 22.1-3, pp. 359–379.

Waheed, Umair bin et al. (2021). “PINNtomo: Seismic tomography using physics-informed neural networks”. In: *arXiv preprint arXiv:2104.01588*.

Xu, Bing et al. (2015). “Empirical evaluation of rectified activations in convolutional network”. In: *arXiv preprint arXiv:1505.00853*.

Yu, Jeremy et al. (2022). “Gradient-enhanced physics-informed neural networks for forward and inverse PDE problems”. In: *Computer Methods in Applied Mechanics and Engineering* 393, p. 114823.

Yu, Xinling et al. (2023). “Pifon-ept: Mr-based electrical property tomography using physics-informed fourier networks”. In: *IEEE Journal on Multiscale and Multiphysics Computational Techniques*.

Zhang, Xiaoping et al. (2022). “GW-PINN: A deep learning algorithm for solving groundwater flow equations”. In: *Advances in Water Resources* 165, p. 104243.

Zhang, Yijie, Xueyu Zhu, and Jinghuai Gao (2023). “Seismic inversion based on acoustic wave equations using physics-informed neural network”. In: *IEEE transactions on geoscience and remote sensing* 61, pp. 1–11.