Master's thesis

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

Albert Johannessen

Modeling Dynamical Systems with Physics Informed Neural Networks with Applications to PDE-Constrained Optimal Control Problems

Master's thesis in Cybernetics and Robotics
Supervisor: Adil Rasheed
January 2024

NTNU
Norwegian University of
Science and Technology

Albert Johannessen

Modeling Dynamical Systems with Physics Informed Neural Networks with Applications to PDE-Constrained Optimal Control Problems

Master's thesis in Cybernetics and Robotics
Supervisor: Adil Rasheed
January 2024

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Engineering Cybernetics

NTNU
Norwegian University of
Science and Technology

Abstract

Creating explicit models of dynamical systems leads to models that will be as accurate as their assumptions. Although many physical laws are still useful for certain use cases but may not be perfectly accurate. Using machine learning methods to learn representations of dynamical systems can be done using much fewer assumptions about the system, but is reliant on having enough data to be accurate. Combining machine learning methods with prior information makes it possible to create models that are more accurate than either method separately.

The main motivation of this master project is to investigate methods of training machine learning models on dynamical systems, while also incorporating prior information derived from first principles to see what challenges exist. This initial investigation is then used to apply the same techniques for other types of problems that can be especially difficult to solve using classical methods.

The machine learning architecture called Physics Informed Neural Networks (PINNs) is a recent method that can successfully merge prior system information with data-driven machine learning. This master project applies PINNs to multiple different types of dynamical systems and investigates what it takes to properly train the machine learning models. There has been many improvements to the basic PINN training method, and introducing causality when training has been shown to give improved results in many cases. It also used PINNs successfully to discover dynamical systems starting from an incomplete equation, both where the equation is known but the parameter values are unknown, as well as the case where there is an unknown term in the equation itself. Finally, knowledge of dynamical systems is then combined with machine learning methods to solve optimal control problems on PDEs, as well as combinations with causal training.

Sammendrag

Å modellere dynamiske systemer eksplisitt vil føre til modeller som er like nøykatige som antagelsene som gikk inn i modelleringen. Mange fysiske lover er nyttige modelleringsverktøy for mange bruksområder, men er ikke alltid perfekte. Å bruke maskinlæringsmetoder for å lære representasjoner av dynamiske systemer kan gjøres ved å bruke mye færre antakelser om systemet, men er avhengig av å ha nok data til å være nøyaktige. Ved å kombinere maskinlæringsmetoder med a priori informasjon gjør det mulig å lage modeller som er mer nøyaktige enn det man kunne fått til med hver metode separat.

Hovedmotivasjonen for denne masteroppgaven er å undersøke metoder for å trene opp maskinlæringsmodeller på dynamiske systemer, samtidig som man tar med tidligere informasjon utredet fra første prinsipper for å se hvilke treningsutfordringer som eksisterer. Resultatene fra denne første undersøkelsen vil dermed brukes for å anvende de samme teknikkene på andre typer problemer som kan være spesielt vanskelig å løse med klassiske metoder.

Maskinlæringsarkitekturen kalt Physics Informed Neural Networks (PINNs) er en ganske ny metode som med klarer å kombinere a priori informasjon om systemet med maskinlæring. Denne masteroppgaven bruker PINNs på flere forskjellige typer dynamiske systemer og undersøker hva som skal til for å trene maskinlæringsmodellene på riktig måte. Det har kommer mange forbedringer til den mest grunnleggende treningsmetoden for PINNs, og det å legge til kausalitet i treningen har vist å gi bedre resultater i mange tilfeller. Den har også brukt PINNs med suksess for å oppdage dynamiske systemer ved å starte fra en ufullstendig ligning, både der ligningen er kjent men at parameterverdier er ukjente, i tillegg til der det er et ukjent ledd i ligningen selv. Til slutt blir kunnskap om dynamiske systemer kombinert med maskinlæringsmetoder for å løse optimale kontrollproblemer med PDEer, samt kombinasjoner med kausal trening.

Preface

The following project and report was completed for the master project in engineering cybernetics. It consists of an individual scientific research project aimed at learning about a chosen specialized field. A literature review of the field is done by reading scientific articles and combining it with previous theoretical knowledge. Multiple experiments are conducted to reproduce and verify results, as well as answer new research questions. The master project functions as a way to gain familiarity with the scientific method, along with producing original research and advancing the field.

I would like to thank my supervisor Adil Rasheed for his guidance and feedback during the project work.

Contents

List of Figures

List of Tables

List of Algorithms

Nomenclature

Abbreviations

ANN	Artificial Neural Network
BVP	Boundary Value Problem
CNN	Convolutional Neural Network
FCNN	Fully-Connected Neural Network
FEM	Finite Element Method
IVP	Initial Value Problem
MLP	Multi-Layered Perceptron
MSE	Mean Squared Error
L-BFGS	Limited-memory - Broyden–Fletcher–Goldfarb–Shanno algorithm
LSTM	Long-Short Term Memory
MPC	Model Predictive Control
ODE	Ordinary Differential Equation
PCA	Principal Component Analysis
PDE	Partial Differential Equation
PINN	Physics Informed Neural Network
RNN	Recurrent Neural Network
SGD	Stochastic Gradient Descent
SQP	Sequentual Quadratic Programming

1 Introduction

1.1 Background and Motivation

Dynamical systems is an area of mathematics used for describing and modeling anything that involves change. It has become ubiquitously used within the fields of engineering, physics, chemistry, economy and many other applied sciences. Systems of interest can be modeled by sets of equations that describe how the system changes over time. These equations can then be used for investigating desired properties of the system, creating simulations to see how the system behaves and modifying aspects to change the behavior. Using physical laws makes it often possible to write down explicit system models starting from first principles. The model of the system will then be a representation of the underlying dynamics which holds for a given set of assumptions. These assumptions must be made explicitly before the model is to be used, and can lead to simpler or more complicated models. For example will Newton's laws of motion be an accurate model for most practical use cases, but Einstein's theory of relativity proved to be a more general and complicated model which conflicts with Newton's laws in extreme cases.

The field of machine learning has become very popular in recent times because of an increased amount of both data and computational power. Machine learning makes it possible for computers to learn from data in order to solve specific problems. This is often accomplished by making the computer create its own internal model of the problem it is trying to solve. Machine learning has already been used extensively for modeling dynamical systems in various ways, which can be particularly useful for systems where it is difficult to write down explicit equations. Playing board games like Chess, or understanding and generating language are both examples of problems where writing down governing equations are challenging.

An interesting problem is then if it is possible to combine the two approaches, where both explicit equations based on first principles are combined with internally learned representations from data to create models that capture reality in a better and more efficient way than either methods separately.

1.2 Contribution

This project explores the potential of using machine learning approaches for learning models of complicated dynamical systems, while also incorporating prior knowledge of the systems derived from first principles. An investigation into what systems are easy to learn and which are more difficult, as well as how to formulate the machine learning problem correctly and efficiently. The main focus is on a deep learning architecture called Physics Informed Neural Networks, which will be explained later in more detail.

This information gained from this initial investigation is then put to use by applying these techniques in clever ways to more complicated problems that might not be easily solved with classical methods. Problems from data-driven discovery of dynamical systems and optimal control of systems governed by partial differential equations are of particular interest.

1.3 Research Questions

A set of research questions are formulated explicitly, to be used for guiding the project in the intended direction.

- What is the state of the art of modeling dynamical systems by combining prior information with machine learning?
- How are machine learning approaches used for data-driven discovery of dynamical systems when starting from incomplete models?
- How can knowledge of dynamical systems be combined with machine learning approaches to find optimal control policies?

1.4 Structure of the Thesis

- Chapter 1 - Introduction: Gives an overview of the main motivations behind the project work, and formulates a set of research objectives.
- Chapter 2 - Theory: Presents the background theory necessary to understand the rest of the project work. Also contains an overview of the current literature.
- Chapter 3 - Method: Includes an explanation and motivation of each individual experiment, along with the necessary details in order to reproduce the results.
- Chapter 4 - Results and Discussion: Shows the results from each experiment presented in the method chapter, and discusses the relevance and meaning of each result.
- Chapter 5 - Conclusion and Further Work: Wraps the experimental results back in with the research objectives presented in the introduction, and comes up with further ideas worth investigating.

2 Theory

2.1 Machine Learning

2.1.1 Introduction

Machine learning is a subfield of artificial intelligence consisting of algorithms that makes computers able to learn to solve problems from data, in contrast to being programmed directly. Artificial intelligence is a wider field about making computers solve complex problems. Machine learning methods are often useful when it is difficult to create explicit models, unlike for example expert systems using handcrafted rules. Machine learning has become ubiquitous in modern computing with the rise of the internet and larger datasets that allows more sophisticated models to be trained. They are used for many different types of problems, some examples include email spam filtering, recommendation systems for online stores, internet search engines, vision systems for autonomous vehicles, chatbots and playing games like chess.

Machine learning was first named by Arthur Samuel in 1959 [1] when he developed a learning algorithm that made a computer able to play the game of checkers. The system consisted of a minimax search algorithm combined with a learned evaluation function. The evaluation function was a linear combination of manually defined board parameters with adjustable weights, where the weights were updated after a win or a loss.

An even earlier example of a machine learning algorithm was developed by Frank Rosenblatt in 1957 [2]. He created a model known as the perceptron, that performed binary classification. It was inspired by the computational model of biological neurons from McCulloch and Pitts from 1943 [3], which formulated that biological neural networks are able to learn things by adjusting the connection strength between the neurons in the network. Rosenblatt's perceptron consisted of multiple input neurons connected to one output neuron, where each connection had an associated weight that could be updated from training data. It also had a nonlinear activation function that worked as a thresholding mechanism.

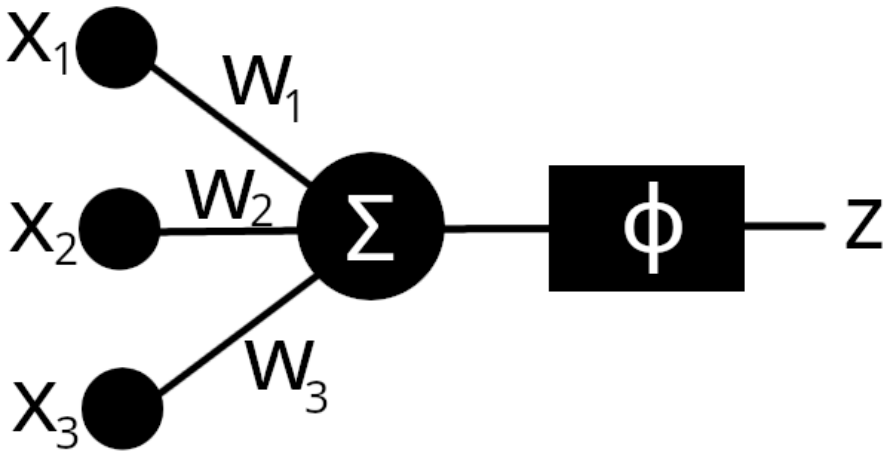


Figure 2.1: Visualization of a perceptron model.

In 1969 the influential book called Perceptrons by Minsky and Papert[4] showed mathematically that the perceptron model was severely limited in what functions could be represented. The conclusion was that the data had to be possible to linearly separate, which essentially means that it must be possible to place a hyperplane in between the datapoints. The XOR function is a simple example of a function that is not linearly separable. The solution to learn such problems was to add additional layers of neurons with weights prior to the input layer, resulting in the model known as the Multi-Layered Perceptron (MLP). The backpropagation algorithm developed by Rumelhart, Hinton and Williams in 1986 [5] made it possible to train MLPs using gradient descent type algorithms commonly used in numerical optimization problems.

2.1.2 Overview

Machine learning consists of three fundamental components. The first is the dataset which is essential to properly learn the desired task. More recently the emphasis on better quality data combined with higher quantities has proven to be more efficient instead of coming up with new learning algorithms or models. The second component is the model, which is a mathematical equation that describes how to apply the data in order to make use of it. The model can also contain a set of learnable parameters. The final component is the learning algorithm that describes how the model with parameters is changed when training on the data.

One of the most widely used rigorous definitions of machine learning is the one from Tom Mitchell [6]. It states: "A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E."

It is common to separate machine learning into three smaller categories depending on how the learning is done, which are called supervised learning, unsupervised learning and reinforcement learning. Supervised learning might be the most commonly applicable type of machine learning and consists of problems where the goal is to learn a mapping from an input to an output. This is achieved by training on a dataset containing both input data and corresponding labels or target values. The two main types of supervised

learning are called regression and classification. Regression is what happens when the input data maps continuously to some output values and the goal is to approximate the underlying function. Classification is what happens when the output values have a discrete and finite domain, and while it might look like a discrete version of regression it will often rely on different types of learning algorithms. A typical regression problem can be to predict some value based on previously collected values, for example in predicting stock prices in finance based on historical data. A typical classification problem can be to determine what class a certain input belongs to, for example detecting fraudulent bank transactions automatically.

Unsupervised learning is about learning things from data without any labels or target values. There can be multiple goals with learning unsupervised. Some of the most common applications are trying to find patterns and structures in the underlying dataset through for example a clustering algorithm like k-means. Another application is to learn better representations or features of the data, for example for the purposes of dimensionality reduction and data compression through Principal Component Analysis (PCA), or as input to another learning algorithm often through autoencoders. The last example of using unsupervised features as input is particularly useful when the dataset is partially labeled, meaning only some of the input data has corresponding labels in what is called semi-supervised learning. The concept of features learned in an unsupervised way is also highly applicable to generative models for both text [7] and images [8] as it is often easier for a model to train on features already determined by another model. The compressed feature space is in this case often called a latent space. In comparison, training generative image models directly on raw pixel values can often make the training process significantly longer and more expensive in terms of compute.

The final category of reinforcement learning deals with problems where the goal is to train an agent to interact with an environment in order to accomplish specific tasks. It has a lot of overlap with control theory in its goals, as the environment and agent often can be modeled together as a dynamical system. The difference comes from that in reinforcement learning the input function is learned while it is often manually designed to meet certain requirements in control theory. According to Sutton, Barto and Williams [9], reinforcement learning can be interpreted as a form of direct adaptive optimal control. Optimal control means that the dynamical system together with its input satisfies some optimality conditions. Direct adaptive control means that the structure and parametric values of the controller input function is learned as an online problem. The way learning is done in reinforcement learning is by making the agent explore and experiment with certain degrees of randomness and getting rewards based on how capable it is of solving its designated tasks. The reward is then used to update the control law. One of the advantages of reinforcement learning over classical control theory is that it does not rely on creating models of the system first, which also means that it can be applied to many other types of systems. One of the first examples of playing games was TD-Gammon [10] which used reinforcement learning to play backgammon. More recently, DeepMind used reinforcement learning with the MuZero model [11] where it is able to both learn optimal strategies as well as the rules of the game, thus giving it the ability to play any game with enough training.

2.1.3 Generalizability

When training machine learning models it is desirable that they are able to generalize outside of the training data. This means that when presented with never before seen data the model should still provide reasonable outputs. For example would an email

spam detection filter not be useful if it only was reliable on emails it had already seen. Generalizing outside the training data is what makes machine learning useful.

To measure how well a model generalizes it is common to split the dataset into two parts: a training set and a testing set. The training set is usually where most of the data ends up with something in the range of $60\% - 80\%$ from the full set. The model is then trained exclusively on the training set while the performance is measured on both the training and testing sets. If the difference in performance measure is small it means that the model is able to generalize. However if the difference starts to increase it is said that the model overfits on the training data which makes it less generalizable.

Overfitting is something that often happens when the model is too complex in comparison to the underlying data distribution. It is also possible for a model to underfit, which can mean that the model is lacking in complexity to represent the data properly. In this case it will generally result in poor performance on both the training and testing set. When choosing models it is therefore desirable that they hit some sweet spot in between underfitting and overfitting. Determining aspects of the data generation process and making assumptions can then lead to more suitable models.

It is often desirable to train multiple different models and compare them against each other to get the best possible model. This could be entirely different types of models, models with different hyperparameters or models trained on separate data. Models often contain parameters which are predetermined instead of learned called hyperparameters which can influence the overall performance of the model. In order to properly compare models, the training set is often split again into a new smaller training set and a validation set. The validation set is then used for testing each model separately in order to compare them against each other, while the final testing set is used to get the actual generalizable performance of the selected model.

2.1.4 Deep Learning

Deep learning is a subfield inside the larger field of machine learning that is becoming increasingly popular in recent years. Whereas in classical machine learning it is more common to manually design the features that go into the models, deep learning instead relies on processing raw data and learning its own feature representations. The deep part of deep learning refers to how models usually have a hierarchy of abstraction levels for the learned feature representations, where the abstraction level of the features increases as the depth of the model increases. These features will build on the features learned from previous layers, which makes it possible to gradually build up complicated representations by increasing the model depth [12]. A simple example could be to train a facial recognition model on images. In this case the shallow layers would learn features such as edges or specific colors, while the deeper layers would combine the edges and colors to create geometric objects followed by all the aspects needed to represent a face.

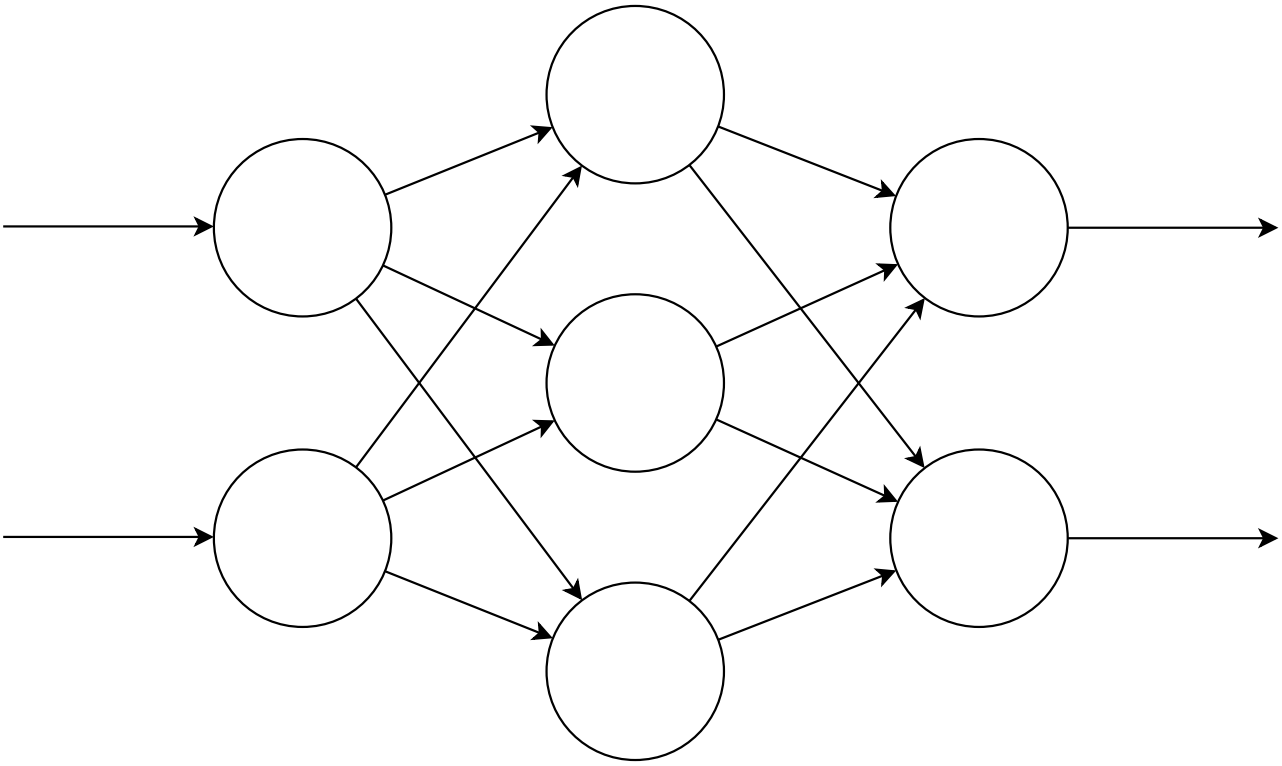


Figure 2.2: Visualization of a simple Fully-Connected Neural Network, represented as a directed graph.

The Multi-Layered Perceptron is perhaps the simplest example of a deep learning based model, where each layer builds upon the output from the previous layers. Additionally, MLPs are also one of the simplest examples of Artificial Neural Networks (ANNs) and are often called Fully-Connected Neural Networks (FCNNs). FCNNs represent neural networks as multiple layers of nodes where each node in one layer is connected to every node in the subsequent layer. This allows for a compact representation of the forward pass using matrix multiplications denoting the connection strength between nodes. The general form of an FCNN can be expressed as:

$$\begin{aligned} \boldsymbol{z}_1 =& \boldsymbol{\phi_1}(\boldsymbol{W}_1\boldsymbol{x} + \boldsymbol{b}_1) \\ \boldsymbol{z}_2 =& \boldsymbol{\phi_2}(\boldsymbol{W}_1\boldsymbol{z}_1 + \boldsymbol{b}_2) \\ & \vdots \\ \boldsymbol{z}_k =& \boldsymbol{\phi_k}(\boldsymbol{W}_k\boldsymbol{z}_{k-1} + \boldsymbol{b}_k) \end{aligned} \tag{2.1}$$

with input $\boldsymbol{x} \in \mathbb{R}^N$, output $\boldsymbol{z}_k \in \mathbb{R}^M$ and intermediate hidden outputs $\boldsymbol{z}_i$. The weight matrices $\boldsymbol{W}_i$ and bias vectors $\boldsymbol{b}_i$ are learnable parameters of the neural network which transform the state layerwise. The $\boldsymbol{\phi}_i$ are called activation functions and are applied at the end of each layer in order to introduce nonlinearity in the function representation. Without nonlinear activation functions it would severely limit the representation capabilities of neural networks. Some common examples of activation functions include the sigmoid function: $\sigma(x) = \frac{1}{1+e^{-x}}$, $\tanh(x)$ and the Rectified Linear Unit (ReLU): $\phi(x) = \max(x, 0)$. Many other types of activation functions also exist but what many have in common is that they are nonlinear and that they somehow restrict the input

variables to some smaller domain. The choice of activation functions can be considered as hyperparameters and are chosen before the training process begins. Which activation function to use can often be dependent on the problem in order to exploit certain properties. Likewise, the number of layers and the number of hidden units are also hyperparameters determined before.

The universal approximation theorem [13] states very simplified that neural networks are able to represent any mapping, assuming that the network is complex enough. Being complex enough is dependent on having a combination of enough layers to represent the given abstraction level, as well as having enough nodes at each layer to represent those specific features. Stated differently, this means that for any given problem there exists some network structure combined with a set of parameter values that can represent the desired mapping up to a given approximation accuracy. However, this theorem does not give any method to discover that network.

Deep learning methods instead rely on solving a numerical optimization problem in order to fit the network to the data from the mapping. This optimization method does not guarantee that the network becomes perfectly accurate, but it does often work well in practice. To do this it is necessary to define an objective function for the optimization problem, usually called a loss function $\mathcal{L}$ in deep learning context. The choice of loss function is problem dependent and there exists many different types of loss functions used in practice. It also can be thought of as a hyperparameter during the training process, although it is not strictly a part of the final network and only used during training. For supervised learning problems the loss function will be a function of the training data labels $\boldsymbol{y}$ and the predicted network outputs $\hat{\boldsymbol{y}} = f(\boldsymbol{x})$ for corresponding input data $\boldsymbol{x}$. The neural network (2.1) is described simply as f. For example is the Mean Squared Error (MSE) loss function defined as:

$$\mathcal{L}(\boldsymbol{y}, \hat{\boldsymbol{y}}) = \frac{1}{M} \sum_{m=1}^{M} (y_m - \hat{y}_m)^2 \tag{2.2}$$

for a single datapoint $\boldsymbol{y} \in \mathbb{R}^M$. Computing the loss over a batch of training data is done by averaging the loss functions of each datapoint. The MSE loss function is very commonly used for regression problems where the goal is to learn a continuous mapping between input and output. For classification problems, the Cross Entropy loss function is one of the most widely used:

$$\mathcal{L}(\boldsymbol{y}, \hat{\boldsymbol{y}}) = -\frac{1}{M} \sum_{m=1}^{M} y_m \ln(\hat{y}_m) \tag{2.3}$$

using that there are M different classes. This is usually accomplished by modeling the output labels as one-hot encoded vectors.

The deep learning optimization can then be stated simply as the unconstrained minimization problem:

$$\min_{\boldsymbol{\theta}} \quad \mathcal{L}(\boldsymbol{y}, \hat{\boldsymbol{y}}) \tag{2.4}$$

where $\hat{\boldsymbol{y}} = \boldsymbol{f}(\boldsymbol{x}; \boldsymbol{\theta})$ for a training set with input points $\boldsymbol{x}$ and outputs $\boldsymbol{y}$. The neural network $\boldsymbol{f}$ is parametrized by $\boldsymbol{\theta}$ containing the weight matrices and bias vectors. The parameter vector $\boldsymbol{\theta}$ is usually very high-dimensional, which leads to a very non-convex loss landscape. In order to find the global minima it is therefore necessary to use global optimization methods, but this is rarely done in practice as it is too computationally

expensive considering both the high amount of parameters and also higher amounts of training data. Instead it is more common to simply treat the problem as it was convex and use gradient descent based methods in order to find a local minima. The performance difference from different local minimas turns out to not necessarily be that important in practice. Higher order optimization methods are also generally not used, as the size of the Hessian matrix would make it too expensive in terms of both computations and memory.

A gradient descent algorithm in its simplest form are described as:

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \alpha \nabla_{\boldsymbol{\theta}} \mathcal{L} \tag{2.5}$$

where the step size α, usually called the learning rate in deep learning, is yet another hyperparameter. In order to perform gradient descent it is therefore necessary to compute the gradients from the loss function to the parameters $\nabla_{\boldsymbol{\theta}} \mathcal{L}$.

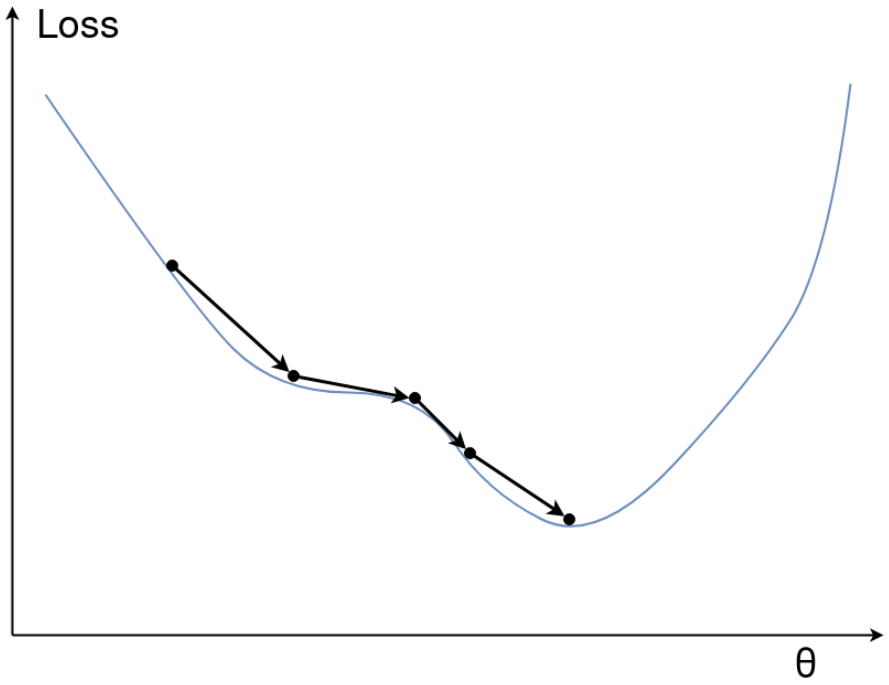


Figure 2.3: Simplified visualization of how gradient descent iteratively minimizes the loss function towards a local minima.

Computing derivatives can be done in different ways. Symbolical gradient representations are generally too computationally expensive for deep neural networks, and numerical methods based on finite differences are too inaccurate to provide useful descent iterations. A middleground is to use automatic differentiation where individual expressions are done symbolically where each symbolic expression is evaluated numerically independent of the overall expression. Automatic differentiation is based on the chain rule from basic calculus where each expression is computed iteratively. It can be done in a forward or reverse mode that both results in the same output but depending on whether to start iterating from the input or the output. Forward mode is usually better if the resulting jacobian matrix is tall, meaning that the dimension of the input is smaller than the dimension of the output. Reverse mode is in contrast better for wide matrices where the input dimension is larger than the output dimension. As the goal is

to compute gradients to the parameters of the neural network, reverse mode automatic differentiation will be significantly less expensive to compute. In deep learning context it is common to refer to reverse mode automatic differentiation as the backpropagation algorithm after it was independently re-discovered by the computer science community [5].

For neural networks with multiple layers it is practical to compute gradients to the parameters of each layer separately. The gradients to previous layers can then be recursively computed based on the gradients to the inputs so far. To simplify notation, write $\nabla_{\boldsymbol{\theta}}\mathcal{L} = \frac{\partial \mathcal{L}}{\partial \boldsymbol{\theta}}$. Gradients to the parameters at layer k are then given as:

$$\frac{\partial \mathcal{L}}{\partial \boldsymbol{\theta}_k} = \frac{\partial \mathcal{L}}{\partial \boldsymbol{z}_k}\frac{\partial \boldsymbol{z}_k}{\partial \boldsymbol{\theta}_k} = \frac{\partial \mathcal{L}}{\partial \boldsymbol{z}_k}\frac{\partial \boldsymbol{\phi}_k}{\partial \boldsymbol{s}_k}\frac{\partial \boldsymbol{s}_k}{\partial \boldsymbol{\theta}_k} \tag{2.6}$$

where $\boldsymbol{s}_k = \boldsymbol{W}_k \boldsymbol{z}_{k-1} + \boldsymbol{b}_k$. All the Jacobians above can be considered well-defined and easily computable analytically. For example will the jacobian $\frac{\partial \boldsymbol{\phi}_k}{\partial \boldsymbol{s}_k}$ be the derivative of the activation function which is implemented alongside the activation function. To get gradients to previous layers, apply the chain rule recursively:

$$\frac{\partial \mathcal{L}}{\partial \boldsymbol{\theta}_{k-1}} = \frac{\partial \mathcal{L}}{\partial \boldsymbol{z}_k}\frac{\partial \boldsymbol{\phi}_k}{\partial \boldsymbol{s}_k}\frac{\partial \boldsymbol{s}_k}{\partial \boldsymbol{z}_{k-1}}\frac{\partial \boldsymbol{z}_{k-1}}{\partial \boldsymbol{\theta}_{k-1}} = \frac{\partial \mathcal{L}}{\partial \boldsymbol{z}_k}\frac{\partial \boldsymbol{\phi}_k}{\partial \boldsymbol{s}_k}\frac{\partial \boldsymbol{s}_k}{\partial \boldsymbol{z}_{k-1}}\frac{\partial \boldsymbol{\phi}_{k-1}}{\partial \boldsymbol{s}_{k-1}}\frac{\partial \boldsymbol{s}_{k-1}}{\partial \boldsymbol{\theta}_{k-1}} \tag{2.7}$$

This generalizes to the recursive formula:

$$\frac{\partial \mathcal{L}}{\partial \boldsymbol{\theta}_i} = \frac{\partial \mathcal{L}}{\partial \boldsymbol{z}_k}\frac{\partial \boldsymbol{z}_k}{\partial \boldsymbol{z}_{k-1}} \cdots \frac{\partial \boldsymbol{z}_{i+1}}{\partial \boldsymbol{z}_i}\frac{\partial \boldsymbol{z}_i}{\partial \boldsymbol{\theta}_i} \tag{2.8}$$

with

$$\frac{\partial \boldsymbol{z}_{i+1}}{\partial \boldsymbol{z}_i} = \frac{\partial \boldsymbol{\phi}_{i+1}}{\partial \boldsymbol{s}_{i+1}}\frac{\partial \boldsymbol{s}_{i+1}}{\partial \boldsymbol{z}_i} \qquad \frac{\partial \boldsymbol{z}_i}{\partial \boldsymbol{\theta}_i} = \frac{\partial \boldsymbol{\phi}_i}{\partial \boldsymbol{s}_i}\frac{\partial \boldsymbol{s}_i}{\partial \boldsymbol{\theta}_i} \tag{2.9}$$

which gives the complete backpropagation algorithm for neural networks. Computing gradients like this is also called to do a backward pass of the network.

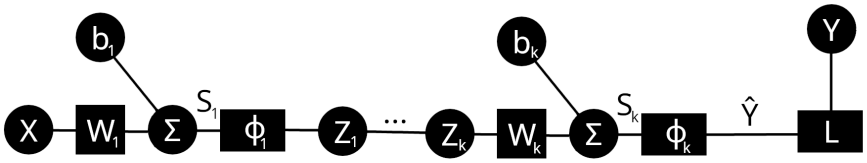


Figure 2.4: Visualization of a deep neural network.

The current gradient descent algorithm will converge to the closest local minimum from the initial parameter values. There are some common techniques that can help to mitigate this somewhat and escape local minimas. As second order optimization methods are generally too expensive these techniques modify the gradient iterations themselves. A standard technique in deep learning is to use Stochastic Gradient Descent (SGD) instead of the normal gradient descent algorithm. The difference is that the stochastic version uses a subset of the training data called a minibatch at each update, compared to the whole data at once. The minibatches are drawn from random partitions of the training set, and it is common to re-randomize or shuffle the minibatch partitions after a full pass of the training data. SGD is particularly useful when dealing with large datasets

for memory reasons, but the stochastic nature makes the loss landscape shift at each iteration, thus making it possible to escape local minimas. Another addition is to add momentum to the iterations so that the gradient updates can overcome short hurdles. Momentum is implemented in practice by averaging the current gradient with the K previously computed gradients and using the averaged gradient as the actual descent direction. It can also result in faster convergence due to reducing oscillations in valleys in the loss landscape. A final technique to mention is that it is common to adjust the learning rate as the learning happens, usually with a larger learning rate at the beginning and then a smaller one towards the end. One of the most common optimizers currently used in deep learning is the Adam (Adaptive Momentum) optimizer [14] using all of these features.

In some cases with smaller datasets it is possible to use approximate second order optimization methods. These are also called quasi-Newton methods due to using an approximation of the Hessian matrix, as computing the full Hessian matrix is not tractable. In deep learning context the most common method is the Limited-memory Broyden–Fletcher–Goldfarb–Shanno (L-BFGS) algorithm [15].

There are also multiple techniques to help deep neural networks generalize better and reduce overfitting. A common method is to add regularization on the network parameters, which effectively reduces the model complexity as the parameter values are driven close to zero. The strength of the regularization is a tunable hyperparameter and can be thought of as a tradeoff between model complexity and data fit after training. Dropout is another technique useful during training that makes it more difficult for networks to simply memorize the training data and instead forces them to learn better representations. This is done in practice by adding a random chance that every node in every layer is simply turned off for that training iteration. The probability is yet another hyperparameter. Dropout is also only used during training, and turned off when the network is used for inference.

To conclude the section on deep learning it is worth mentioning that there exists other types of ANNs than just FCNNs. Convolutional Neural Networks (CNNs) are very common when dealing with data that contains translationally invariant features [16]. Images are the typical example of this, as training a CNN to classify objects in images means that the position of the object inside the image should not impact the classification. CNNS are inspired by linear FIR filters from digital signal processing and image processing, where an input signal sent through a filter results in an output of the mathematical convolution of the input signal and the filter impulse response. In classical image processing such filters were manually designed to detect certain features in images [17], while more modern CNNs are basically learning their own filter representations. Using CNNs instead of FCNNs for image processing tasks will usually result in much fewer total network parameters.

Another type of ANN are Recurrent Neural Networks (RNNs) [16], which are intended for handling sequential data. Typical examples include time-series data, or text and natural language processing. They work by processing one element of the input sequence at a time and modifying an internal hidden state of the RNN containing the latent representation of the sequence. One problem with RNNs is that they heavily suffer from what is known as the vanishing gradient problem [18], which is also encountered for any deeper network. Due to the backpropagation algorithm recursively computing gradients, the gradient magnitudes have a tendency to shrink for each successive layer due to magnitudes smaller than 1 being multiplied together. The Long-Short Term Memory (LSTM) architecture is a modified RNN that solves the problem of vanishing

gradients, and are in practice always preferred over standard RNNs. RNNs have also been used for sequence to sequence tasks where one RNN encodes the sequence into a hidden state and a second RNN decodes the hidden state to an output sequence [19]. This approach used to be state of the art for problems like translating natural languages, but RNNs has more recently been surpassed by the transformer architecture for most sequence modeling tasks [20].

2.2 Partial Differential Equations

2.2.1 Introduction

Dynamical systems can generally be thought of as something that has a state (the system) that changes over time (the dynamics). Dynamical systems are often described using mathematical equations to make them easier to study.

Ordinary Differential Equations (ODEs) are a way to describe systems represented by a state vector $\boldsymbol{x}(t)$ that evolves continuously in time t. ODEs are particularly useful for describing many systems related to physics, engineering, finance and other types of sciences. An ODE can be defined as an equation with a function of a variable together with its derivatives [21]. As an implicit equation this becomes:

$$F(t, x(t), \dot{x}(t), \ddot{x}(t), \dots) = 0 \tag{2.10}$$

with state variable $x(t) \in \mathbb{R}$ and independent variable t. The order of the ODE is defined as the highest order of the derivative present in the equation. For practical applications it is often easier to work with the explicit form:

$$\dot{\boldsymbol{x}} = \boldsymbol{f}(t, \boldsymbol{x}(t)) \tag{2.11}$$

where the state is now a vector $\boldsymbol{x}(t) \in \mathbb{R}^n$. As the ODE now only contains the first order derivative, the order is now defined to be the dimension of the state vector. An ODE containing higher order derivatives can be converted into the form of equation (2.11) by augmenting the state vector with the higher order derivatives of the state, without losing any generality.

ODEs of the form of equation (2.11 have infinitely many solutions for $\boldsymbol{x}(t)$, as the equation only describes how the state changes. Combining the equation with an initial value $\boldsymbol{x}(t_0) = \boldsymbol{x}_0$ results in an Initial Value Problem (IVP). The IVP can be uniquely solved for $\boldsymbol{x}(t)$ from the Picard-Lindelöf theorem [22] provided that the dynamics of the system f is Lipschitz continuous. The solution at time t_1 is found by integrating from the initial time t_0:

$$\boldsymbol{x}(t_1) = \boldsymbol{x}(t_0) + \int_{t_0}^{t_1} \boldsymbol{f}(t, \boldsymbol{x}(t))dt \tag{2.12}$$

which is commonly solved using numerical methods designed specifically for ODEs in practice.

Partial Differential Equations (PDEs) are a way to describe systems represented by a function $u(t, \boldsymbol{x})$ that evolves continuously in time t. In contrast to ODEs where the state was a vector that evolved in time, PDEs use states represented by functions that evolve over time. Functions in the context of functional analysis can be interpreted as vectors with an uncountably infinite dimension, which also means that they can be approximated by a high-dimensional ODE.

The independent variables are commonly denoted as time t and spatial variables $\boldsymbol{x}$. The system state represents some quantity that is distributed over the space, where the distribution also change over time. The system state function u has so far been considered as a scalar, but systems of PDEs are also possible. However, this master thesis will mainly deal with scalar valued PDEs. Because the system state in a PDE is able to capture more information than in an ODE they are able to model more complicated dynamical systems, but are also generally harder to both solve and study.

Similarly to ODEs, PDEs can be described more generally as an implicit equation [23]:

$$F(\boldsymbol{x}, u(\boldsymbol{x}), Du(\boldsymbol{x}), D^2u(\boldsymbol{x}), \dots) = 0 \tag{2.13}$$

defined on an open subset $\Omega \subset \mathbb{R}^n$ called the domain, with independent variables $\boldsymbol{x} \in \Omega$, state variable $u(\boldsymbol{x})$ and differential operator D^k representing the set of all k-th order partial derivatives with respect to the elements of the $\boldsymbol{x}$ vector. This definition does not include the time variable t explicitly, but it can be interpreted as being included in the independent variable vector $\boldsymbol{x}$. The order of the PDE is defined as the order of the highest partial derivative present in the equation, similarly to the order of an ODE.

PDEs in the form of equation (2.13) will also have an infinite number of possible solutions without placing further restrictions. When the time t is included as an independent variable in the PDE it is common to define an initial condition. Initial conditions are simply defined as $u(0, \boldsymbol{x}) = g(\boldsymbol{x}), \boldsymbol{x} \in \Omega$ for a given function g dependent on the specific problem. It is also sometimes necessary to define a set of boundary conditions on $\partial\Omega$. Different types of boundary conditions exists, but one of the most commonly used is the Dirichlet boundary condition that takes the form of: $u(t, \boldsymbol{x}) = \phi(t, \boldsymbol{x})$ defined on $\boldsymbol{x} \in \partial\Omega$ and $t \in [0, \infty)$ for a scalar function ϕ. Another commonly used is called the Neumann boundary condition which takes the form of: $\frac{\partial u(t,\boldsymbol{x})}{\partial \boldsymbol{n}} = \phi(t, \boldsymbol{x})$, which specified the boundary in terms of the rate of change in the normal direction $\boldsymbol{n}$ on the boundary. Boundary conditions can also be more complicated, for example involving higher order derivatives, nonlinearities and different combinations of these. The problem of solving a PDE with a specific set of boundary conditions is called a Boundary Value Problem (BVP).

When defining initial conditions for ODEs, it is necessary to specify the same number of initial conditions as the order of the equation. For example will a second order equation require and initial state and an initial derivative. The same principle works for PDEs, where the initial conditions must be specified up to the highest order partial derivative of the time. For example will the heat equation require one initial temperature distribution as it only contains a first order partial derivative in time, while for example the wave equation requires both an initial amplitude and an initial amplitude velocity as there is a second order partial derivative in time. This is also relevant for boundary conditions, where the order of the partial derivatives in the spatial variables specifies how many boundary conditions with derivatives are required for a unique solution.

As the solution to a PDE has to satisfy the equation itself (2.13), it would seem necessary that potential solutions are differentiable. Although in practice, many PDEs will naturally result in discontinuous solutions, thus also making them non-differentiable. One example of this is the shock wave phenomenon arising from Burgers' equation, which is used frequently for experiments throughout this thesis. This result led to the development of an alternative formulation of PDEs based on what is called a weak derivative, where the main principle is to rewrite the PDE to be represented as an integral equation without including any explicit derivatives. A discontinuous solution satisfying

the weak formulation of the PDE is then called a weak solution. Weak solutions are also shown to not necessarily belong to any commonly used function spaces used for studying differential equations, like an L^p space. A Sobolev space is an extension of L^p spaces where weak solutions can exist, and have become one of the main ways to study PDEs in a mathematically rigorous way [23]. But this is not directly relevant when working with numerical approximations and going into more detail is outside the scope of this master thesis.

2.2.2 Example PDEs

Below are listed some common PDEs that can appear in many practical applications, as well as in experiments with PINNs.

Laplace's equation

$$\nabla^2 u(\boldsymbol{x}) = 0$$

Heat equation

$$\frac{\partial u(t,\boldsymbol{x})}{\partial t} = k\nabla^2 u(t,\boldsymbol{x})$$

Wave equation

$$\frac{\partial^2 u(t,\boldsymbol{x})}{\partial t^2} = k\nabla^2 u(t,\boldsymbol{x})$$

Schrödinger equation

$$i\hbar\frac{\partial \Psi(t,\boldsymbol{x})}{\partial t} = \left[-\frac{\hbar^2}{2m}\nabla^2 + V(t,\boldsymbol{x})\right]\Psi(t,\boldsymbol{x})$$

1D Burgers' equation

$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} = \nu\frac{\partial^2 u}{\partial x^2}$$

1D Allen-Cahn equation

$$\frac{\partial u}{\partial t} - a\frac{\partial^2 u}{\partial x^2} + bu^3 - cu = 0$$

Kuramoto–Sivashinsky equation

$$\frac{\partial u}{\partial t} + \frac{\partial^2 u}{\partial x^2} + \frac{\partial^4 u}{\partial x^4} + u\frac{\partial u}{\partial x} = 0$$

Korteweg - De Vries equation

$$\frac{\partial u}{\partial t} + \frac{\partial^3 u}{\partial x^3} - 6u\frac{\partial u}{\partial x} = 0$$

Maxwell's vacuum equations

$$\begin{cases} \nabla \cdot \boldsymbol{E} = 0, \\ \nabla \cdot \boldsymbol{B} = 0, \\ \nabla \times \boldsymbol{E} = -\frac{\partial \boldsymbol{B}}{\partial t}, \\ \nabla \times \boldsymbol{B} = \mu_0\epsilon_0\frac{\partial \boldsymbol{E}}{\partial t}, \end{cases}$$

Incompressible Navier–Stokes equations

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

2.2.3 Classification of Second Order PDEs

Many PDEs of interest that are derived from physical systems are often second order PDEs, which means that the equation contains second order partial derivatives. In the study of second order PDEs, it is common to classify them into three different types of PDEs, each of which with their own distinct type of dynamics [24]. For a linear second oder PDE defined on two variables (t, x), the general form can be written as:

$$Au_{tt} + Bu_{tx} + Cu_{xx} + Du_t + Eu_x + Fu + G = 0 \tag{2.14}$$

where all the coefficients are functions of (t, x). The equation is called homogeneous if $G = 0$.

To classify such PDEs, consider the quantity known as the discriminant:

$$\Delta = B^2 - 4AC \tag{2.15}$$

The type of the PDE is then classified as follows:

- Hyperbolic if $\Delta > 0$
- Parabolic if $\Delta = 0$
- Elliptic if $\Delta < 0$

The terminology used is analogous to types of curves defined from standard quadratic equations in two variables.

For a general nonlinear second order PDE, the classification can be done in the same way by linearizing the equation to calculate the necessary coefficients. It is also possible to generalize the classifications to higher-dimensional spatial variables.

Hyperbolic and parabolic equations are generally systems that evolve in time. Hyperbolic equations are typically defined by vibrations and periodic motion. The simplest example of a hyperbolic equation is the wave equation. Parabolic equations are typically defined by diffusion, where the state tends to spread out over time. The simplest example of a parabolic equation is the heat equation.

Elliptic equations are generally systems defined to be in some equilibrium state, and are often independent of time. In this case, the classification done with the discriminant can then be done by replacing the variables (t, x) with spatial variables (x, y). The simplest example of an elliptic equation is Laplace's equation.

As the coefficients can be functions of (t, x), it is possible to have PDEs where the type of dynamics changes throughout the domain. For an example of this, consider the Tricomi equation:

$$x\frac{\partial^2 u}{\partial t^2} - \frac{\partial^2 u}{\partial x^2} = 0 \tag{2.16}$$

The discriminant is then calculated as:

$$\Delta = B^2 - 4AC = 0^2 - 4 \cdot x \cdot (-1) = 4x \tag{2.17}$$

which means that it is hyperbolic for $x > 0$, elliptic for $x < 0$ and parabolic on the transition $x = 0$. The Tricomi equation is used to model supersonic aerodynamics, and the change in dynamics correspond to subsonic and supersonic flight, with a sonic boom happening on the transition [24].

2.2.4 Analytical Solutions

Solving PDEs analytically is generally not possible, and it is much more common in practice to use numerical methods based on Finite Differences or the Finite Element Method (FEM). However some simple PDEs combined with simple boundary conditions are analytically solvable. This can be useful for generating data without implementing a full numerical solver.

Separation of Variables

One method of approaching this for linear PDEs is to use separation of variables, which assumes that $u(t, \boldsymbol{x}) = G(t)F(\boldsymbol{x})$. A linear PDE with initial condition and boundary condition will have a unique solution provided that the initial and boundary conditions are sufficiently nice, for example that the constraint functions are continuously differentiable. This means that if the separation of variables results in a solution that satisfies the PDE and conditions it is also a unique solution. Rewriting $u(t, \boldsymbol{x}) = G(t)F(\boldsymbol{x})$ and inserting into the PDE will result in two separate ODEs that can be solved independently. These two solutions can then be combined to give the final solution to the PDE and conditions.

To give an example of the separation of variables method, consider the 1-dimensional heat equation: $\frac{\partial u}{\partial t} = c^2 \frac{\partial^2 u}{\partial x^2}$ where $u = u(t, x)$ defined on $[0, \infty) \times [0, L]$. Use the boundary conditions: $u(t, 0) = u(t, L) = 0$ and the more general initial condition: $u(0, x) = g(x)$. This can be interpreted as modeling the heat distribution of a bar of length L where the ends of the bar are kept at temperature 0, and where the initial temperature distribution is given by $g(x)$ [25].

Start by separating $u(t, x) = G(t)F(x)$ and insert this back into the heat equation:

$$\begin{aligned}
&\frac{\partial u}{\partial t} = c^2 \frac{\partial^2 u}{\partial x^2} \\
&\frac{\partial}{\partial t}\left[G(t)F(x)\right] = c^2 \frac{\partial^2}{\partial x^2}\left[G(t)F(x)\right] \\
&\dot{G}(t)F(x) = c^2 G(t)F''(x) \\
&\frac{\dot{G}(t)}{c^2 G(t)} = \frac{F''(x)}{F(x)}
\end{aligned}$$

The left side of the equation above is a function of t and the right side is a function of x. Because they are equal for all values of t and x, it implies that both sides must be equal to a constant. This means that the expression can be split into two different ODEs:

$$\begin{aligned}
\dot{G}(t) + p^2 c^2 G(t) = 0 \\
F''(x) + p^2 F(x) = 0
\end{aligned}$$

where the constant is denoted as p^2. A negative constant would result in two unstable ODEs which would not be able to satisfy the boundary conditions of the PDE, which is why it can be assumed positive. The bottom equation has the general solution:

$$F(x) = A\cos(px) + B\sin(px)$$

The boundary conditions give $u(t,0) = G(t)F(0) = 0 \implies F(0) = 0$ and $u(t,L) = G(t)F(L) = 0 \implies F(L) = 0$. As the ODE is second order it gives a unique solution with these two constraints. Inserting $x = 0$ gives $F(0) = A\cos(p \cdot 0) + B\sin(p \cdot 0) = A \implies A = 0$. Inserting $x = L$ gives $F(L) = B\sin(p \cdot L) = 0 \implies p = \frac{n\pi}{L}$ for $n = 1, 2, \ldots$.

Solving the top ODE results in the general solution:

$$G(t) = Ce^{-p^2c^2t} = e^{-\lambda_n^2 t}$$

where $\lambda_n = \frac{cn\pi}{L}$ Combining these two solutions with a given n results in a solution to the PDE:

$$u_n(t,x) = B_n \sin\Big(\frac{n\pi}{L}x\Big)e^{-\lambda_n^2 t}$$

Because the PDE is linear it means that linear combinations of solutions will also be a solution. The full solution is then given as:

$$u(t,x) = \sum_{n=1}^{\infty} B_n \sin\Big(\frac{n\pi}{L}x\Big)e^{-\lambda_n^2 t}$$

where the B_n constants depend on the B and C constants from the ODE solutions above, and must be chosen to satisfy the initial condition $u(0,x) = g(x)$. Inserting this into the proposed solution gives:

$$u(0,x) = \sum_{n=1}^{\infty} B_n \sin\Big(\frac{n\pi}{L}x\Big) = g(x)$$

which means that the B_n coefficients are the result from decomposing the initial condition $g(x)$ into a Fourier sine series. An explicit expression for these coefficients is:

$$B_n = \frac{2}{L}\int_0^L g(x)\sin\Big(\frac{n\pi x}{L}\Big)dx$$

Method of Characteristics

Another method that can work for simple PDEs is the method of characteristics [26]. Consider a first order multi-dimensional PDE on the form:

$$\frac{\partial u}{\partial t}(t,\boldsymbol{x}) + \boldsymbol{v}(t,\boldsymbol{x}) \cdot \nabla u(t,\boldsymbol{x}) + w(t,\boldsymbol{x},u) = 0 \tag{2.18}$$

A characteristic of the equation is a trajectory $\boldsymbol{x}(t)$, where the spatial variable $\boldsymbol{x}$ is given as a function of the time variable t, and where the following ODE holds:

$$\frac{d\boldsymbol{x}}{dt}(t) = \boldsymbol{v}(t,\boldsymbol{x}(t)) \tag{2.19}$$

The Lagrangian derivative can then be defined as:

$$\frac{Du}{Dt}(t) = \frac{d}{dt}u(t,\boldsymbol{x}(t)) \tag{2.20}$$

which intuitively can be understood as the rate of the change of the state u on a characteristic trajectory $\boldsymbol{x}(t)$. By expanding the Lagrangian derivative:

$$\frac{d}{dt}u(t,\boldsymbol{x}(t)) = \frac{\partial u}{\partial t} + \frac{d\boldsymbol{x}}{dt}\cdot\nabla u \tag{2.21}$$

and where $\frac{d\boldsymbol{x}}{dt} = \boldsymbol{v}(t,\boldsymbol{x}(t))$, thus giving another ODE:

$$\frac{Du}{Dt}(t) + w(t,\boldsymbol{x}(t),u(t,\boldsymbol{x}(t))) = 0 \tag{2.22}$$

This means that for any equation on the form of (2.18), the PDE can be reduced to a system of two ODEs: (2.19) and (2.22), each of which are defined using the given v and w, and can be solved using conventional ODE methods. The resulting solutions to the ODEs can in most cases be used to construct the full PDE solution.

As an example, consider the simple 1-dimensional transport equation:

$$\frac{\partial u}{\partial t} + a\frac{\partial u}{\partial x} = 0 \tag{2.23}$$

where $v(t,x) = a$ constant and $w(t,x,u) = 0$. Also assume that an initial condition is given as: $u(0,x) = g(x)$. The characteristic ODE (2.19) gives that $x(t) = at + x_0$. Define $z(t) = u(t,x(t))$ so that the Lagrangian derivative becomes: $\frac{dz}{dt} = 0$ and $z(t) = z_0$.

To proceed from here, the initial condition is needed to retrieve a unique solution $u(t,x)$. The initial condition can be used as:

$$z(0) = u(0,x(0)) = u(0,x_0) = g(x_0) \tag{2.24}$$

This also implies that $z(t) = z_0 = g(x_0)$, which means that u is constant along characteristic trajectories, which makes sense as $w = 0$ in the example equation.

For this example, the next step would be to invert the equations for the characteristic, resulting in:

$$\begin{aligned} x(t) &= at + x_0 \\ \implies x_0 &= x - at \end{aligned} \tag{2.25}$$

which can be inserted back into the value for x_0, giving the final solution:

$$u(t,x) = g(x - at) \tag{2.26}$$

This requires that the mapping from the characteristic is invertible, which may not always be the case depending on the resulting ODEs. The transport equation is one of the simplest PDEs, but the method of characteristics can be used similarly for many other more complicated PDEs.

2.3 Optimal Control Theory

2.3.1 Brief Overview

Optimal control can be thought of as a field laying at the intersection of control theory and mathematical optimization. The goal is to use optimization methods in order to find control policies for a dynamical system, where the dynamical system behaves in an optimal way for a given objective function. Consider a dynamical system represented as an ODE on the form:

$$\dot{\boldsymbol{x}} = \boldsymbol{f}(t, \boldsymbol{x}, \boldsymbol{c}) \tag{2.27}$$

with system state $\boldsymbol{x}(t) \in \mathbb{R}^n$, control input $\boldsymbol{c}(t) \in \mathbb{R}^m$ and system dynamics $\boldsymbol{f}$. In control theory it is normally common to use the symbol $\boldsymbol{u}$ for the control input, but here the symbol $\boldsymbol{c}$ is used to avoid confusion with the state of a PDE. The general form of an optimal control problem can then be formulated as:

$$\begin{aligned} \min_{\boldsymbol{c}(t)} \quad & h(t_f, \boldsymbol{x}(t_f), \boldsymbol{c}(t_f)) + \int_{t_0}^{t_f} g(t, \boldsymbol{x}(t), \boldsymbol{c}(t)) dt \\ \text{s.t.} \quad & \dot{\boldsymbol{x}} = \boldsymbol{f}(t, \boldsymbol{x}, \boldsymbol{c}) \\ & \boldsymbol{x}(t_0) = \boldsymbol{x}_0 \end{aligned} \tag{2.28}$$

where the time span goes from an initial time t_0 to final time t_f, initial condition $\boldsymbol{x}_0$, final cost h and instantaneous cost g. The optimization problem has the system ODE (2.27) as a constraint, and is done over a suitable function space, for example the space of functions with finite energy L^2. This formulation assumes that the initial and final times are fixed variables, but it is in some cases also possible to have the final time as part of the optimization variables, which can be useful for example for trajectory optimization when minimizing the time spent is important. It is also common to include control constraints that can take the form of: $\boldsymbol{c}_a \leq \boldsymbol{c}(t) \leq \boldsymbol{c}_b, \forall t \geq t_0$, as the control can often be limited by physical hardware constraints.

Solving a continuous time optimal control problem can in some rare cases be done analytically based on either using Pontryagin's maximum principle or by solving the Hamilton–Jacobi–Bellman equation. One notable special case is the Linear Quadratic Regulator (LQR), where the system ODE is linear, the final cost is zero, the instantaneous cost is quadratic in both state and control input and the final time goes to infinity. This also assumes that the system is controllable. For a Linear Time-Invariant (LTI) system represented as:

$$\dot{\boldsymbol{x}} = \boldsymbol{A}\boldsymbol{x} + \boldsymbol{B}\boldsymbol{c} \tag{2.29}$$

the control input will be given as a full-state feedback controller:

$$\boldsymbol{c} = -\boldsymbol{K}\boldsymbol{x} \tag{2.30}$$

for a control matrix $\boldsymbol{K}$ computed analytically from the algebraic Riccati equation.

Usually it is not possible to find analytical solutions to the optimization problem (2.28), which motivates the use of numerical methods. This is done by discretizing the state and the control input, thus changing the search space from a function space to a vector space. The system ODE is then split up into a set of constraints operating at each of the discretization points. The discretization step size h defines the number of new variables and dynamics constraints as: $N = \frac{t_f - t_0}{h}$. The discretized optimization problem becomes:

$$\begin{aligned} \min_{\boldsymbol{x}_1,\ldots,\boldsymbol{x}_N\in\mathbb{R}^n,\boldsymbol{c}_1,\ldots,\boldsymbol{c}_N\in\mathbb{R}^m} \quad & h(t_N,\boldsymbol{x}_N,\boldsymbol{c}_N)+\sum_{i=0}^{N-1} g(t_i,\boldsymbol{x}_i,\boldsymbol{c}_i) \\ \text{s.t.} \quad & \boldsymbol{x}_N=\boldsymbol{x}_{N-1}+h\boldsymbol{f}(t_{N-1},\boldsymbol{x}_{N-1},\boldsymbol{c}_{N-1}) \\ & \vdots \\ & \boldsymbol{x}_2=\boldsymbol{x}_1+h\boldsymbol{f}(t_1,\boldsymbol{x}_1,\boldsymbol{c}_1) \\ & \boldsymbol{x}_1=\boldsymbol{x}_0 \end{aligned} \tag{2.31}$$

using an explicit Euler discretization. The continuous time system states $\boldsymbol{x}(t)$ and control inputs $\boldsymbol{c}(t)$ are discretized and turn into a collection of points that can be flattened into a vector of dimension: $N \cdot n + N \cdot m$.

When the setup is similar to the LQR with linear constraints and a quadratic cost function, the optimization problem is convex, meaning that there is a globally optimal solution. Because of this it is very common to work with linearized dynamics if possible. For cases where this is not possible, other numerical methods can be used to solve the problem, for example Sequential Quadratic Programming (SQP) [15].

The output from an optimization problem like (2.31) gives a state trajectory and a set of control inputs for a given time horizon. This will result in open-loop control if implemented directly, and is therefore lacking in robustness to possible disturbances and modeling errors. Model Predictive Control (MPC) is a control policy that implements optimal control with feedback, therefore fixing the robustness problem. MPC works by solving the optimization problem (2.31) and using the first control input on the actual plant. The next series of control inputs are then discarded, and the optimization problem is solved again using the next measured state as the new initial state. MPC can be very computationally expensive, but is a very general technique for implementing optimal control with feedback, and is used frequently when appropriate.

2.3.2 PDE-Constrained Optimal Control

Optimal control theory is not limited to systems described by ODEs and can easily be extended to other types of systems, for example stochastic systems or systems represented by PDEs. The overall goal remains the same: find a control input to make the system behave in a way that is optimal with respect to a given objective function.

As an example of why this can be so useful, consider the problem of controlling the temperature in a room. One way to do this is to use a temperature sensor placed at one location in the room, and then construct a model from thermodynamics that models the temperature as a single scalar valued differential equation. This system can then be controlled using any methods from traditional control theory. The main drawback of this approach is that it assumes that the temperature is both constant and distributed equally throughout the room. This can be an acceptable drawback in many circumstances, as almost every house heater does it like this. But for cases where more fine control over specific areas are needed, it is possible to extend the model of the system with the three-dimensional heat equation. The desired temperature can then be represented as a function of space instead of a single scalar value. As the overall problem is more fine-grained, one potential benefit of this approach is to save on total energy consumption, at the cost of increased modeling complexity and computational cost.

Assume the dynamical systems of interest are represented as a PDE on the explicit form of:

$$\frac{\partial u}{\partial t} = f(u(t, \boldsymbol{x}), Du(t, \boldsymbol{x}), \ldots, D^N u(t, \boldsymbol{x}), \boldsymbol{c}) \tag{2.32}$$

where the system state is a scalar valued function of time and position $u(t, \boldsymbol{x})$. The dynamics f is here assumed to be time and space invariant, meaning that it does not depend on t or $\boldsymbol{x}$ directly, and it takes in the state u, as well as partial derivatives up to order N along with a control input $\boldsymbol{c}$. It is often common in texts on PDE-constrained optimal control to use y as the system state and $\boldsymbol{u}$ as the control input to align more with the conventions of control theory. However this thesis adopts the conventions used in texts on PDEs and Physics Informed Neural Networks, as those are the main topics.

The optimal control problem can be defined similarly to (2.28):

$$\begin{aligned} \min_{\boldsymbol{c}} \quad & J(u(t, \boldsymbol{x}), \boldsymbol{c}) \\ \text{s.t.} \quad & \frac{\partial u}{\partial t} = f(u(t, \boldsymbol{x}), Du(t, \boldsymbol{x}), \ldots, D^N u(t, \boldsymbol{x}), \boldsymbol{c}) \\ & u(t_0, \boldsymbol{x}) = g(\boldsymbol{x}), \quad \forall \boldsymbol{x} \in \Omega \\ & u(t, \boldsymbol{x}) = \phi(t, \boldsymbol{x}), \quad \forall \boldsymbol{x} \in \partial\Omega \end{aligned} \tag{2.33}$$

for an objective function $J(u(t, \boldsymbol{x}), \boldsymbol{c})$. The initial condition is given by $g(\boldsymbol{x})$ for an initial time t_0, and the boundary constraint is here given as a Dirichlet boundary condition with $\phi(t, \boldsymbol{x})$, although different types of boundary constraints can be defined in the same way. This formulation of the optimal control problem suggests that the control input $\boldsymbol{c}$ can change the dynamics of the system in some way. If the control input is a function of time and space $\boldsymbol{c} = \boldsymbol{c}(t, \boldsymbol{x})$, then this problem formulation is referred to as volume control, as every point spatial point can influence the dynamics. This is also the formulation that gives the most direct control of the problem. Other alternatives exist, for example that the control input is only dependent on the time $\boldsymbol{c} = \boldsymbol{c}(t)$, or only over a subset of the spatial domain.

The objective function $J(u(t, \boldsymbol{x}), \boldsymbol{c})$ can be represented in many different ways depending on the application. One common choice is to make a system behave in a given way, in the sense that the system state $u(t, \boldsymbol{x})$ is driven towards a desired system state $u_d(\boldsymbol{x})$ as $t \to \infty$, similar to what is done for many control problems on ODEs. One difference is that u_d is a real-valued function of space, instead of just a vector value. It's also possible for the objective function to not only be evaluated at a subset of the spatial domain, for example at specific points, or at the boundaries.

Solving this type of optimal control problem can be done in the same way as for ODEs using a discretization technique. This results in an array representing the points of the system state u, which can then be optimized over using numerical optimization techniques. However, the computational complexity increases rapidly with the number of dimensions of the spatial domain, and can be infeasible to solve in the worst case.

These formulations of the optimal control problem assume that the control input has some influence over the dynamics of the system. But there are other possible formulations as well. One example is that the dynamics themselves no longer depend on the control input, but rather that the problem is to determine the optimal initial condition, thus making the initial condition the control input. Or with an equation: $g(\boldsymbol{x}) = c(\boldsymbol{x})$. This problem formulation would also be possible for systems described by ODEs, but is generally not so practical as most uncontrolled systems already exist in some state without the possibility to get to the initial condition in the first place. So this serves as mostly a theoretical possibility for now.

The final formulation discussed here is done by again making the dynamics no longer depend on the control input, but rather that the boundary conditions are dependent on it. This can make more sense physically, for example using Neumann boundary conditions $\frac{\partial u(t,\boldsymbol{x})}{\partial t} = \phi(t,\boldsymbol{x}), \forall \boldsymbol{x} \in \partial\Omega$, where the boundary condition is given by the control input: $\phi(t,\boldsymbol{x}) = c(t,\boldsymbol{x})$, this has the physical implication of adding or removing flux at through the boundaries. The same setup also works with Dirichlet boundaries, as well as mixed boundaries. It is also possible for the control input to only affect a part of the boundary condition and not the whole boundary, but that would be problem dependent.

2.4 Physics Informed Neural Networks

2.4.1 Overview

As neural networks are universal function approximators they are also able to learn representations of outputs from dynamical systems. One possible approach for ODEs is to model the network as the actual dynamics of the system. The network can then be trained using a simple regression setup where the target values are the derivatives of the input values. The learned dynamics can then be used with any numerical integrator to simulate the learned system. One of the main drawbacks of this approach is that it requires a lot of training data, including the actual derivatives which are often not directly available. The Neural Ordinary Differential Equation (Neural ODE) architecture [27] [28] [29] made it possible to learn ODEs without using derivatives but they still require a lot of training data and are generally slow to converge for more complicated systems. Extensions to the architecture [30] [31] [32] that forces a more specific representation turned out to be significantly increase both the convergence speed when training and also give more accurate representations of the unobserved states. However, the Neural ODE architecture is not easily generalizable to PDEs which limits its total representative capabilities. They also suffer from not being able to generalize well outside the training data.

The Physics Informed Neural Network (PINN) framework [33] [34] [35] is a method for training neural networks to learn the output of dynamical systems. In contrast to the methods described above, PINNs will learn a representation for a specific solution of a dynamical system instead of learning the underlying dynamics. This is accomplished by using explicit prior information of the system dynamics during training. One of the motivations for this approach was that machine learning algorithms would often partially rediscover known systems, but by utilizing known physical laws or other forms of domain expertise it can greatly increase the speed of convergence and increase robustness of the solutions. Data acquisition can often be difficult or costly for complex physical, engineering and biological dynamical systems, but adding prior information can reduce the amount of necessary data to properly learn the systems.

The original PINN problem formulation considered parameterized nonlinear systems of PDEs on the form:

$$\boldsymbol{u}_t + N[\boldsymbol{u};\boldsymbol{\lambda}] = 0 \tag{2.34}$$

for a state $\boldsymbol{u}(t,\boldsymbol{x})$ called the latent solution, nonlinear operator N parameterized by $\boldsymbol{\lambda}$ and for $t \in [0,T], \boldsymbol{x} \in \Omega \subset \mathbb{R}^n$. The subscript notation $\boldsymbol{u}_t$ is in the context of PDEs often used as a shorter notation for $\frac{\partial \boldsymbol{u}}{\partial t}$. This notation is very similar to the general PDE given in (2.13), except that it is an explicit representation with respect to the partial

derivative $\boldsymbol{u}_t$, and that the state $\boldsymbol{u}(t, \boldsymbol{x})$ is now a vector meaning, it can represent a system of PDEs as opposed to a single scalar equation.

$\boldsymbol{u}(t, \boldsymbol{x})$ can then be approximated with a deep neural network and then trained on sampled data from the system. Compared to traditional numerical methods like FEM, this will result in a continuous rather than discrete mapping. Training the neural network can be done by minimizing the loss function:

$$\mathcal{L}_u = \frac{1}{N_u} \sum_{i=1}^{N_u} ||\boldsymbol{u}(t_u^i, \boldsymbol{x}_u^i) - \boldsymbol{u}^i||^2 \tag{2.35}$$

for training data $t_u^i, \boldsymbol{x}_u^i, \boldsymbol{u}^i$ with $i = 1, 2, \ldots, N_u$. This simple regression setup will usually converge provided that there is enough training data. However, for cases with limited data or where data is only available from the initial and boundary conditions the trained network would not generalize well. The solution is to add prior knowledge about the structure of the PDE into the loss function. This of course requires that the PDE is known or can be derived from first principles. The PINN setup works by defining:

$$\boldsymbol{f} = \boldsymbol{u}_t + N[\boldsymbol{u}; \boldsymbol{\lambda}] \tag{2.36}$$

and then combining the the loss function above (2.35) with the physics informed loss:

$$\mathcal{L}_f = \frac{1}{N_f} \sum_{i=1}^{N_f} ||\boldsymbol{f}(t_f^i, \boldsymbol{x}_f^i)||^2 \tag{2.37}$$

so that the final training loss becomes: $\mathcal{L} = \mathcal{L}_u + \beta \cdot \mathcal{L}_f$, for some hyperparameter β that adjusts the tradeoff between data and prior information. This results in solving the following unconstrained minimization problem:

$$\min_{\boldsymbol{\theta}} \quad \mathcal{L}_u + \beta \cdot \mathcal{L}_f \tag{2.38}$$

for neural network parameters $\boldsymbol{\theta}$. This can also be interpreted as solving the constrained minimization problem:

$$\begin{aligned} \min_{\boldsymbol{\theta}} \quad & \mathcal{L}_u \\ \text{s.t.} \quad & \mathcal{L}_f = 0 \end{aligned} \tag{2.39}$$

where β takes the role of the Lagrange multiplier. This constrained minimization problem can be considered an approximation of the continuous version of the problem where the PDE (2.34) is enforced as a constraint.

The physics informed loss (2.37) is evaluated at the collocation points $t_f^i, \boldsymbol{x}_f^i$ with $i = 1, 2, \ldots, N_f$, which is not reliant on data but rather manually chosen or generated before training. The original PINN version [33] used a method called Latin Hypercube Sampling, which works by partitioning the space into a grid with equal spacing, and then randomly sampling points from each cell in the grid. The number of collocation points has a great impact on the generalization ability of PINNs, but has the drawback that they decrease the training speed. Evaluating $\boldsymbol{f}(t_f^i, \boldsymbol{x}_f^i)$ using equation (2.36) can be done using automatic differentiation of the neural network from the output to the input. Additionally, as PINNs can train with very small amounts of training data the original authors trained the networks using the L-BFGS optimization algorithm.

Systems of PDEs can often result in complex dynamical outputs, which necessitates a neural network with enough depth and width to properly represent the complexity of

the output. However, larger models with more parameters also require more data to be able to generalize, as they would easily overfit otherwise. The physics informed loss can in this case be thought of as a regularization term that restricts the overall complexity of the model and makes it follow the prior more closely, which is what makes it possible to rely on relatively small amounts of training data compared to the model complexity. It is also worth noting that if no data is available but the initial and boundary conditions have known expressions, data can be generated by sampling from these expressions. In this case, PINNs does not require any more prior information than other numerical methods for solving PDEs which typically rely on a discretization of the initial and boundary conditions in addition to the PDE equation.

This setup uses that the parameters of the PDE $\boldsymbol{\lambda}$ are known, as they are necessary to evaluate $\boldsymbol{f}$. The original authors also demonstrated an alternative setup where the parameters are unknown [34], where PINNs can be used for data driven discovery. This can be done by setting $\boldsymbol{\lambda}$ as an unknown parameter vector combined with the same PINN setup described above, and then including $\boldsymbol{\lambda}$ in the optimization. Gradients from the loss function $\mathcal{L}$ to parameters $\boldsymbol{\lambda}$ can be computed using automatic differentiation through $\boldsymbol{f}$. A drawback of this method is that it requires significantly more data compared to when the parameters are known, and more specifically that the data is sampled from throughout the entire spatio-temporal domain instead of just from the initial data and boundary.

2.4.2 Literature review

Deeper Investigations

The PINN framework [33] [34] [35] have been studied extensively after they were introduced originally. The PINN approach was used to model fluid dynamics based on the Navier-Stokes equation in an efficient way [36] that is much more flexible with the geometry of the boundary compared to traditional solvers, and much more accurate compared to pure machine learning. But the authors noticed that the PINN model struggles for higher Reynolds numbers indicating more turbulence, as neural networks often struggle with learning higher frequency signals. Multiple surveys on PINNs have been published that discusses some of the current applications and challenges, also highlighting the high-frequency drawbacks of neural networks, with comparisons to classical numerical methods [37], and an overview of novel architecture improvements along with theoretical investigations on convergence and error estimates [38]. Further work going into detail on convergence guarantees framed in mathematical analysis [39] and rigorous estimates on the approximation capabilities and generalization [40] also exists.

It has been shown that PINNs can fail to learn even the simplest of PDEs for certain parameter values [41], even though the network itself has the capacity to learn. The problem is instead that the optimization problem is too difficult to solve with simple gradient based methods leading to a model that fails at approximating the true underlying system. Potential solutions to overcome this is to train the PINNs with different methods. One example mentioned in [41] is to do curriculum learning, where the complexity of the physics information is gradually increased during training. This can be viewed as a form of transfer learning where the PINN is first trained on a simpler system before being trained further on a more complicated [42]. Another approach is to formulate the PINN training as a sequence to sequence learning problem [41]. Instead of learning the mapping for the whole spatio-temporal domain, learn one sequence at a time, for example as a slice at a certain point in time, and then learn the next point

in time by starting from the previous slice. Another proposed technique is to start the optimization problem with the Adam optimizer utilizing the momentum, and switching to the L-BFGS optimizer after a certain number of training steps [42]. L-BFGS seems to be required for learning more complicated PDEs to converge, but L-BFGS does not use momentum and will go directly towards the closest local minimum, which can be avoided by using Adam first. Further investigations done by analyzing the training in the limit of infinitely wide neural networks have also been done [43].

An attempt at fighting back against the effects from the curse of dimensionality was made by introducing separable PINNs [44], where each dimension of the input vector is entered into a separate neural network. The resulting outputs are then combined to form the final output. This makes the training much more efficient, as the automatic differentiation can operate on a much smaller dimensional subspace, compared to the full neural network. Even for as low as three dimensional inputs the authors found a significant speed up in terms of time spent training compared to the vanilla PINN setup.

Some PDEs are defined using periodic boundary conditions. Training a PINN to learn this can be approximated by adding that the values of the PINN output should be equal to each other at the boundaries as a term to the loss function. If the PDE also has a periodic boundary including partial derivatves, this can also be done in a similar way by computing partial derivatives, comparing outputs and adding to the loss. But for higher order, or even infinitely differentiable periodic boundary conditions, this gets increasingly computationally expensive. A solution to this was proposed in [45]. This method works by manually constructing a Fourier embedding of the input as a type of feature engineering, and then sending this embedding as the input to the PINN. This guarantees infinitely differentiable periodic boundary conditions where the accuracy and resolution is influenced by the dimension of the Fourier embedding. The Fourier embedding is constructed as a vector containing sines and cosines with increasing frequencies of the spatial point, where all sines and cosines are set to be periodic on the spatial domain. The number of frequencies relates to the dimension of the embedding and can be considered a hyperparameter. The Fourier embedding can also be extended to multiple dimensions, by considering pairs of products between sines and cosines. As the vector with sines and cosines are sent into the first layer of the neural network, the neural network will then multiply these terms with weights and add them together in different ways, making the overall process resemble a customized Fourier series.

Learning Chaotic Systems

Systems represented as ODEs where the dynamics of different states operate at different frequencies, for example one state with slower dynamics and another state with faster dynamics, are known to be particularly challenging for traditional numerical solvers. Equations like these are often called stiff ODEs and can require specialized solvers in certain cases. The same phenomenon is also present when training PINNs. Similar problems can also exist for systems represented by PDEs, one example of this is the turbulence within the Navier-Stokes equations.

This stiffness problem and how it relates to training PINNs was analyzed by looking at the gradients during training [46]. The authors propose a new learning algorithm that updates the learning rate based on gradient values iteratively during training in a method called learning rate annealing. They also proposed a new neural network structure with some inspirations from skip connections as found in residual networks [47] that are less affected by these training problems.

Another solution to solve such problems is to train using time marching [48]. This

means that the time-domain is partitioned into multiple smaller segments where they cover the original time domain. A separate PINN is then trained on each of these time segments separately until convergence, and sequentially where the next network is initialized with the parameters of the previous network. The initial condition of the first PINN is set to the initial condition of the overall problem, while subsequent networks have initial conditions set to the output of the previous network at the corresponding time. The overall solution to the problem is then gradually built up in time, leading to a higher final accuracy.

The concept of partitioning the time was taken to another level with the introduction of causality during training [49]. With this setup the overall loss function when training is modified in a way that prioritizes learning dynamics in a time-sequential manner. In practice, this means that the time domain is once again partitioned into N_t sets, and the loss function is computed over each of these time domains separately. The final loss is computed as a weighted sum of these individual losses:

$$\mathcal{L}(\boldsymbol{\theta}) = \frac{1}{N_t} \sum_{i=0}^{N_t} w_i \mathcal{L}_i(t_i, \boldsymbol{\theta}) \tag{2.40}$$

where the loss weights w_i are computed as:

$$w_i = \exp\left(-\epsilon \sum_{k=1}^{i-1} \mathcal{L}(t_k, \boldsymbol{\theta}) \right) \tag{2.41}$$

The loss weight w_i can be interpreted as a value that increases when the loss functions on previous time domains are low. So initially when training, all losses will be high, meaning only the first time domain will have any significant weighting. As the loss on the first time domain improves, the weight for the second time domain increases, and so on for the other time domains. The ϵ hyperparameter is a positive value that determines the relative importance of causality. If $\epsilon = 0$, then all loss weights will be 1, thus ignoring causality. If $\epsilon \to \infty$ then only the current time domain will have an influence on the loss function until this local loss goes to zero.

The causal loss function can also be combined with time marching, where the time domain is first split into ranges for training separate PINNs, and then each time range is split further into another set of time ranges for causal training. The authors of [49] combined these techniques with the modified network structure from [46] to achieve high accuracy on the chaotic Lorenz ODE, Kuramoto–Sivashinsky equation, and turbulent Navier-Stokes equation compared to other approaches so far. They also used exact periodic boundary conditions with a Fourier embedding [45] where applicable.

System Discovery

Some methods for performing system discovery with PINNs exists. This makes it possible to learn the dynamics of a system based on data. One of these methods is to perform symbolic regression with MLPs as a part of the symbolic operators [50]. This makes it possible to generate symbolic expressions for the solutions of PDEs, but are often difficult to work with in practice as the authors note.

Another method that learns the expression of an unknown term within a differential equation was recently proposed [51]. This method is particularly useful when the equation for the dynamics is partially known with some potentially missing terms. The authors did this by setting up three different neural networks responsible for learning

different aspects of the problem. The first learns the known dynamics, the second learns the output state of the system and the third learns the unknown part of the boundary condition. These three networks combined makes it possible to explicitly represent the unknown dynamics.

To train networks with this setup, consider three neural networks parametrized by: $\boldsymbol{\theta}_u$ for learning the output values, $\boldsymbol{\theta}_b$ for learning the unknown part of the boundary and $\boldsymbol{\theta}_f$ for learning the unknown part of the dynamics. The three networks are trained together using a loss function $\mathcal{L}(\boldsymbol{\theta}_u, \boldsymbol{\theta}_b, \boldsymbol{\theta}_f)$ equal to a sum of three loss functions: $\mathcal{L}_u(\boldsymbol{\theta}_u)$ for the output values, $\mathcal{L}_b(\boldsymbol{\theta}_u, \boldsymbol{\theta}_b)$ for the boundary and $\mathcal{L}_f(\boldsymbol{\theta}_u, \boldsymbol{\theta}_f)$ for the dynamics. The output value loss function is in this case equal to the standard loss function for given data points (2.35). The boundary loss is computed by inserting output values from the u-network into the known part of the boundary conditions, and then adding in the outputs from the b-network and computing the MSE loss. The dynamics loss is again similar to the physics informed loss (2.37) where the known part of the dynamics is used on collocation points from the u-network like in (2.36), and where the outputs from the f-network is added in before computing the MSE loss.

One thing to note is that the f-network needs to be explicitly given input values containing the relevant partial derivatives, as it is not able to learn arbitrary dependencies between derivatives. These partial derivatives must be computed from the output values of the u-network and differentiated with automatic differentiation. It then requires some prior knowledge of the unknown term as an inductive bias.

Solving Control Problems

PINNs have also been used for control applications. Formulating a PINN on ODEs where the control variable is included as an input to the neural network in addition to the time state makes it possible to use PINNs as part of a Model Predictive Control (MPC) framework using the PINN as the predictive model [52]. The authors here note that the training the Physics Informed Neural nets-based Control (PINC) as they call it makes the MPC run much faster during runtime as it is no longer required to integrate an ODE, because a lot of computational time is traded off when the PINC is trained.

As PINNs are naturally suited for learning on PDEs, they are also suited for PDE-constrained optimal control problems [53]. The underlying system is now described by a PDE, similar to the problem formulations discussed in the previous section. The PINN formulation of PDE-constrained optimal control [53] used two different neural networks that were trained together simultaneously. The first network learns the system state u in the same way as the normal PINN, but where either the physics informed loss through the PDE dynamics or the data loss through the boundary conditions are influenced by the control input. The control input $\boldsymbol{c}$ is represented by the second neural network, and is trained by formulating the whole problem as minimizing a loss function corresponding to a standard PDE-constrained optimal control problem. This setup is very flexible, and can solve any of the different problem configurations related to PDE-constrained optimal control.

3 Method

3.1 Introduction

To gain familiarity with PINNs and explore their limitations, multiple PINNs were trained on data from different types of dynamical systems. The models were in some cases also compared against other models to highlight differences. After an initial investigation into how to work with PINNs, more advanced experiments were conducted that showcased more practical applications as well as better training methods.

All the PINN models were implemented from scratch with PyTorch [54] and functorch [55]. The ODEs were integrated using torchdiffeq [56]. The code is open sourced at GitHub, using the MIT license for free use. The repository includes both code for training models as well as code for generating all the plots used throughout this thesis.

3.2 Data

The initial models were trained on generated synthetic data on relatively simple problems. The ODE systems were integrated using the Dormand-Prince 5 [57] integrator with a step size of $h = 0.01$, where the initial values were both randomly sampled and manually chosen on different systems. The initial PDE systems were only using data generated from the initial and boundary conditions so it was not necessary to use a numerical solver. More data could be sampled from linear PDEs by solving them analytically using the separation of variables method. The analytical solution was also used for comparing the learned systems to the true systems as a form of training validation. Some experiments used datasets obtained from repositories associated with various papers. These datasets were used for validating the learned PINNs, and were originally obtained with traditional numerical solvers

The collocation points for calculating the physics informed loss were randomly sampled throughout the spatio-temporal domain. This can also be done when training PINNs on real data, as the collocation points do not rely on any information of the output of the system. The points were sampled uniformly as this turned out to work well enough instead of the latin hypercube sampling. Some of the simpler systems used linearly spaced collocation points instead of random sampling, but this proved to not be as effective for more complicated systems. More simpler PDEs could get away with sampling collocation points once before the training, while the more demanding PDEs sample collocation points randomly at every training iteration.

A manual seed was also set to ensure that the experiments were reproducible and to minimize the chance that the results are dependent on the random seed.

3.3 Model architecture

The output trajectories of the dynamical systems were modeled as fully-connected neural networks. The tanh activation function was used to introduce nonlinearity and constrain

the outputs of the hidden layers, while also allowing negative values. It is also continuously differentiable which could be an advantage when learning outputs that are also continuously differentiable. The physics informed loss was calculated using automatic differentiation to get first derivatives of the neural network at the collocation inputs, and second derivatives using the already calculated first derivatives as input to the automatic differentiation again. Higher order derivatives can be calculated in a similar manner.

When learning higher order ODEs with PINNs it is possible to convert the system dynamics into a state space representation and use that as the prior when computing the loss. This would however require collocation samples from a higher dimensional space, and potentially also more data corresponding to all the new states to accurately learn. For systems defined with a single state and higher order derivatives of that state it is then more useful to not use the state space representation as it is less data-intensive.

The MSE loss function was used for all purposes during training.

3.4 Hyperparameters

Table 3.1 contains a list of the hyperparameters for each of the conducted experiments. α refers to the learning rate, N_u to the number of datapoints in the training set, N_f to the number of collocation points and β to the relative weighting of the loss functions. Each experiment is explained in more detail in the next section. Every experiment uses the Adam optimizer with some exceptions that use the L-BFGS optimizer, which can be seen in the table below when $\alpha = 1$. In general, it was found that the L-BFGS optimizer converges much faster, but is generally not practical for systems with more complicated dynamics where the optimizer can get stuck in a suboptimal minimum.

Table 3.1: Table of hyperparameters for each experiment.

Experiment	Hidden Layers	Hidden Units	Epochs	α	N_u	N_f	β
Learning Dynamical Systems with PINNs							
Linear ODE	3	20	20000	1e-4	10	100	1e-4
Nonlinear ODE	4	20	10000	1e-4	10	100	1e-1
Time-varying ODE	2	40	1000	1e-3	1	1000	1e-1
1D Linear PDE	4	20	1000	1e-3	100	1000	1e-1
2D Linear PDE	4	20	1000	1e-3	100	1000	1e-3
Nonlinear PDE	5	20	1000	1	100	10000	1e-1
Data-Driven Discovery of Dynamical Systems with PINNs							
1D Linear PDE	4	20	1000	1	1000	1000	1e-2
2D Linear PDE	4	20	1000	1	20000	20000	1e-4
Causal Training							
Simple PDE	5	50	10000	1e-3	1000	10000	1e-3
Chaotic PDE	4	100	150000	1e-3	1000	25000	1e-3
Symbolic Operator Discovery							
Nonlinear PDE	6	50	10000	1e-3	1000	10000	1e-3
Solving PDE-Constrained Optimal Control Problems with PINNs							
Flux Control	5	50	5000	1	100	10000	100
Dirichlet Boundary Control	4	50	10000	1	100	10000	1
Neumann Boundary Control	4	50	10000	1	100	10000	1
Initial Control	4	50	20000	1e-3	400	20000	1
Regularization with the Maximum Principle							
Elliptic PDE	4	50	1000	1	200	10000	1e-1
Causal Optimal Control with PINNs							
Initial Control	5	50	20000	1e-3	1000	10000	1e4
Reversed Initial Control	5	50	20000	1e-3	1000	10000	1e3

Some of the later experiments involve multiple neural networks being trained simultaneously. The listed hyperparameters in table 3.1 will in this case refer to the network that corresponds to the output state u. Other networks will usually have similar or slightly lower hidden layers and hidden units.

3.5 Experimental Setup

The following section describes the setup of each conducted experiment using the hyperparameters listed in table 3.1. Details about the specific dynamical system along with parameters are described, in addition to what the purpose and goal of each experiment is. The results of each experiment are presented in the next chapter.

3.5.1 Learning Dynamical Systems with PINNs

The first experiments are meant to show that PINNs can learn outputs from dynamical systems in a strictly superior way than standard neural networks by generalizing better on less data.

Linear ODE

Linear time-invariant ODEs are generally considered the easiest type of dynamical system to work with. They always have an analytical solution which can be expressed in a closed form, and are always stable or unstable on a global level. A common toy problem when working with ODEs is the second order dynamical system called the mass spring damper, which shows up in a variety of situations related to mechanical systems. The mass spring damper system is defined by the following ODE:

$$m\ddot{x} + d\dot{x} + kx = 0 \tag{3.1}$$

with state: position $x(t)$ and parameters: mass m, damping d and spring constant k. The following experiments use the parameters: $m = 1, d = 5, k = 500$ and initial conditions $x(0) = 1, \dot{x}(0) = 0$.

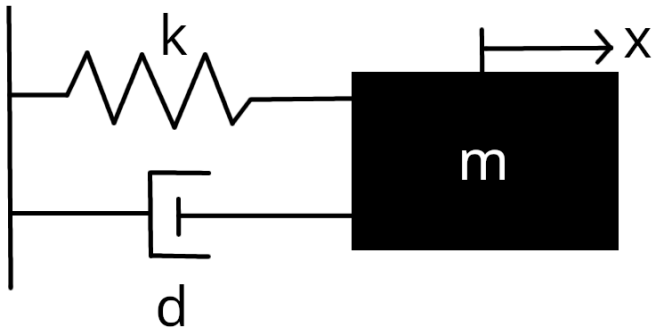


Figure 3.1: Mass spring damper system visualization.

The purpose here is to demonstrate how to train PINNs on outputs from some of the simplest type of dynamical systems. A PINN is compared to a neural network with the same number of parameters trained in the standard way by formulating a regression problem on time inputs and trajectory outputs, to showcase the advantages of also incorporating prior information about the system dynamics. The training is done by sampling 10 datapoints evenly spaced in time in the range $0 < t < 0.4$ seconds from the true output of the system generated numerically. The training data only contains the positions $x(t)$ and not the velocity $\dot{x}(t)$. The mass spring damper is also used to demonstrate the importance of having enough data when training.

Nonlinear ODE

Consider the second order system called the Van der Pol oscillator defined by the nonlinear ODE:

$$\ddot{x} - \mu(1 - x^2)\dot{x} + x = 0 \tag{3.2}$$

with state $x(t)$ and parameter μ. It can be considered as a mass spring damper where $m = 1$, $d = -\mu(1 - x^2)$ and $k = 1$, and is a common example of an ODE with a limit cycle, meaning that every trajectory converges to some periodic function in time.

The parameter: μ is set to 1, and both a PINN and a standard neural network with the same number of parameters is trained on 10 datapoints evenly spaced throughout $0 < t < 20$. The purpose is to show that PINNs are able to easily generalize to nonlinear systems without losing any accuracy.

Time-varying ODE

For the final ODE, consider the non-autonomous Riccati equation defined by [21]:

$$\dot{x} = x^2 - t \tag{3.3}$$

This system has two equilibrium points located at $x(t) = \pm\sqrt{t}$ where the negative equilibrium point is stable and the positive is unstable. This means that every initial condition below zero will approach the trajectory $x(t) = -\sqrt{t}$. This is demonstrated by training a PINN with equation (3.3) as prior but not using any datapoints at all. A second PINN is also trained by additionally using the initial condition as a datapoint.

1D Linear PDE

PINNs were originally formulated based on PDEs, as they are generally more difficult to work with than ODEs. The initial PDE experiment considered the 1-dimensional heat equation defined as follows:

$$u_t = k^2 u_{xx} \tag{3.4}$$

The following experiment is defined by setting $k = 1$, boundary conditions $u(t, 0) = u(t, 1) = 0$ defined on the spatial region $0 < x < 1$ and initial condition $u(0, x) = \sin(\pi x)$. This makes it possible to solve the heat equation analytically using the separation of variables method. The initial condition was chosen strategically to simplify the computation of the resulting Fourier series from equation (2.2.4). The resulting solution has the expression:

$$u(t, x) = \sin(\pi x) e^{-\pi^2 t} \tag{3.5}$$

Figure 3.2 displays a visualization of the state $u(t, x)$ over the spatial domain and the time range $0 < t < 0.2$ seconds, which is generated using the analytical expression (3.5). As the boundary is set to zero it will lead to an initial heat distribution evolving in time by dissipating towards zero.

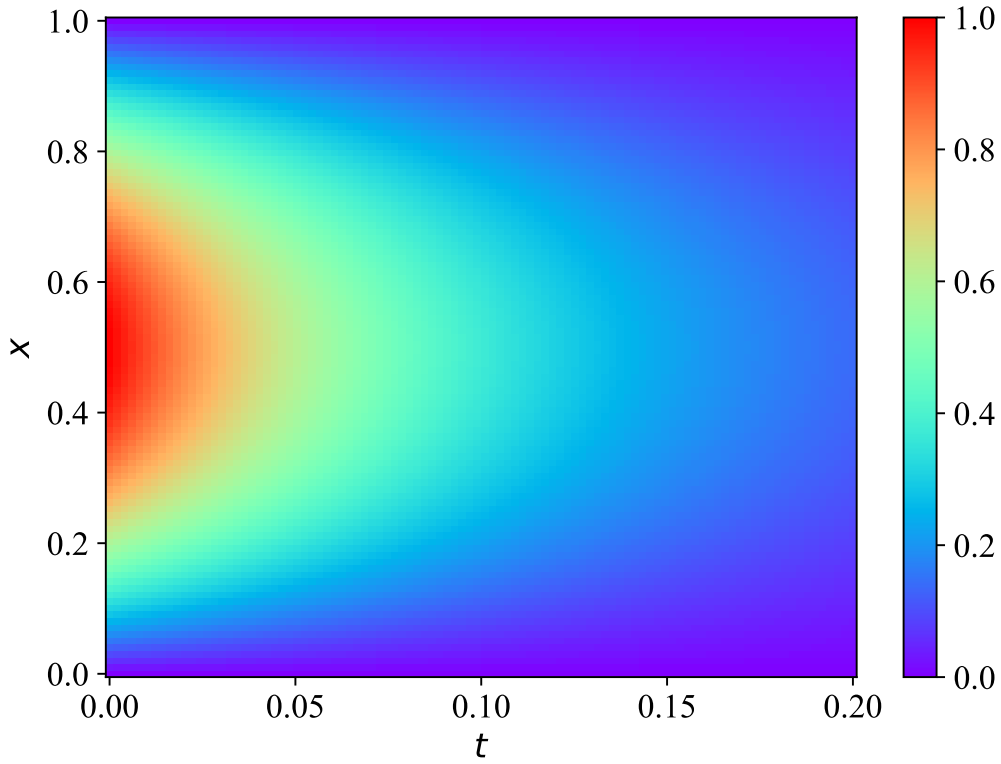


Figure 3.2: True solution of a 1-dimensional heat equation.

Training a PINN is done by only using datapoints sampled randomly and uniformly on the boundary and initial conditions. Many real life problems modeled with PDEs have known expressions for the initial and boundary conditions which means that the current setup is equivalent to standard numerical solvers in terms of how much prior data they need. The collocation points for the physics informed loss are uniformly sampled across the whole spatio-temporal domain.

2D Linear PDE

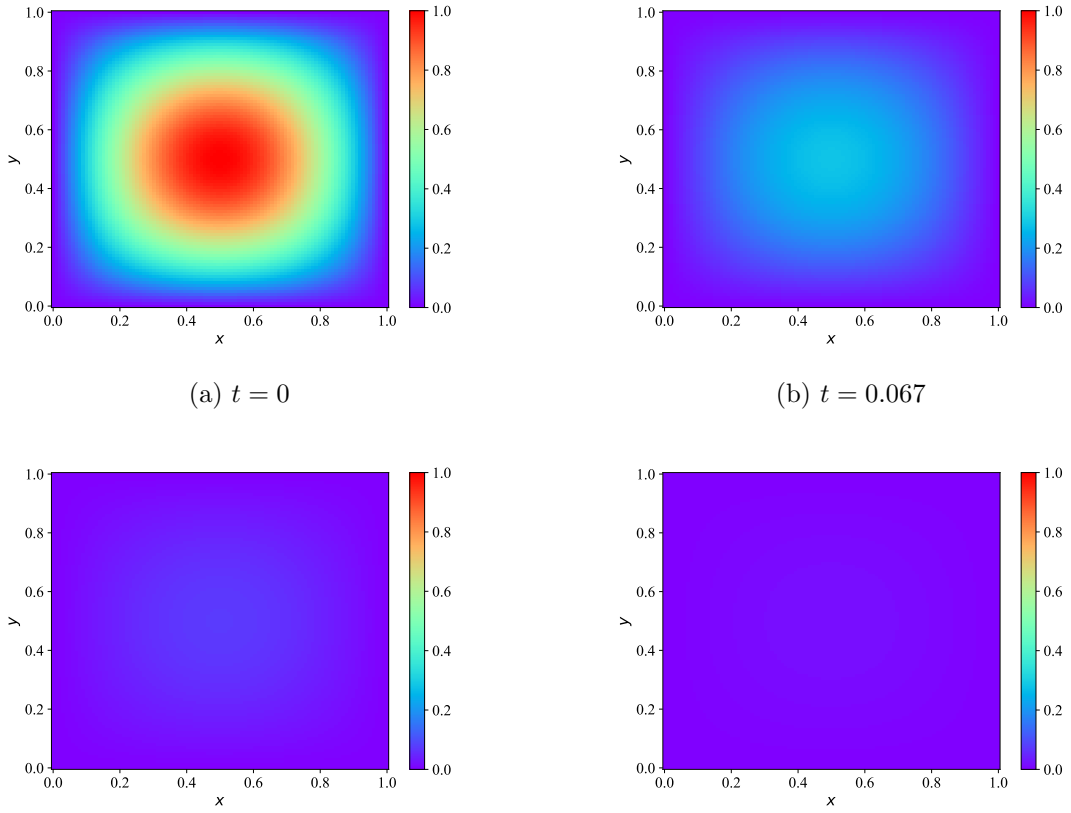
Extending the previous experiment to the 2-dimensional heat equation defined as follows:

$$u_t = k^2(u_{xx} + u_{yy}) \tag{3.6}$$

for spatial variables x and y. The experiment uses a very similar setup where $k = 1$, the initial condition is given as: $u(0, x, y) = \sin(\pi x)\sin(\pi y)$ for $0 < x, y < 1$ and boundary conditions: $u(t, 0, y) = u(t, 1, y) = u(t, x, 0) = u(t, x, 1) = 0$. Similarly to the 1-dimensional case, an analytical solution can be found from the separation of variables method. This results in the expression:

$$u(t, x, y) = \sin(\pi x)\sin(\pi y)e^{-2\pi^2 t} \tag{3.7}$$

The true solution is visualized below in Figure 3.3. As the solution is three-dimensional it was visualized by extracting 2-dimensional slices in time to see how the spatial domain evolves.



(a) $t = 0$ (b) $t = 0.067$

(c) $t = 0.133$ (d) $t = 0.2$

Figure 3.3: True solution of a 2-dimensional heat equation.

A PINN is trained with the exact same method as the previous 1-dimensional case by using uniformly sampled datapoints along the initial and boundary conditions for the regression loss and uniformly sampled collocation points. The goal of this experiment is to demonstrate that PINNs are able to learn higher dimensional PDEs, but with some more difficulty.

Nonlinear PDE

Nonlinear PDEs are usually not solvable analytically and can in many cases also be difficult to accurately solve numerically. The Navier-Stokes equations for describing fluid mechanics has an infamous open problem in mathematics regarding simply the existence of smooth solutions in 3-dimensional space. The homogeneous incompressible Navier-Stokes equations are defined as:

$$\begin{cases} \boldsymbol{u}_t + (\boldsymbol{u} \cdot \nabla)\boldsymbol{u} - \nu \nabla^2 \boldsymbol{u} = -\frac{1}{\rho} \nabla p, \\ \nabla \cdot \boldsymbol{u} = 0, \end{cases} \tag{3.8}$$

for states: velocity field $\boldsymbol{u}(t, \boldsymbol{x})$ and pressure $p(t, \boldsymbol{x})$, with parameters: viscosity ν and density ρ.

The viscous Burgers' equation can be derived from the Navier-Stokes equation by reducing to the 1-dimensional case, setting the pressure equal to zero and removing the constraint that the fluid flow is incompressible. This results in the following PDE:

$$u_t + uu_x - \nu u_{xx} = 0 \tag{3.9}$$

The experiment uses the parameter $\nu = \frac{0.01}{\pi}$, initial condition $u(0, x) = -\sin(\pi x)$ and boundary conditions $u(t, -1) = u(t, 1) = 0$.

The Adam optimizer from the previous experiments are replaced with the L-BFGS optimizer using a line search based on the strong Wolfe conditions. It also requires significantly more collocation points to learn properly, but not any more datapoints along the initial and boundary conditions. The Burgers' equation is generally difficult to solve numerically due to a shock formation that forms after a certain amount of time [33], but is still possible to learn with PINNs, although at an increased computational cost compared to simpler PDEs.

3.5.2 Data-Driven Discovery of Dynamical Systems with PINNs

The next set of experiments are meant to demonstrate an area where PINNs outperforms alternative numerical methods, as it can also be used with partially unknown dynamics. One of the advantages of machine learning compared to many classical alternatives is that the methods do not rely on an explicit model of the target system, which would have been subjected to potentially inaccurate assumptions and modeling errors.

1D Linear PDE

The same heat equation problem formulated in the previous subsection is now revisited, except that the parameter k is now considered unknown, thus leading to an incomplete model of the system dynamics. Using the approach described in [34] it is possible to train a PINN on data from the system and also learn an estimate of the k parameter simultaneously. This is meant to be an experiment that showcases how the method works, which means it can also be applied to real data without knowing the analytical solution, or even the full details of the system dynamics.

The heat equation with the given setup was chosen for this experiment because it has a simple analytical solution that can be used to generate data and also verify the solution. The training data is now generated by random uniform sampling from the entire spatio-temporal domain, and the collocation points for the physics informed loss are chosen to be the same points as where the training data is collected. It therefore requires a lot more data than the case where the parameter is known. The L-BFGS optimizer is also used instead of Adam.

The estimated $\hat{k}$ parameter is randomly initialized by sampling from a standard normal distribution, and is then included as a part of the optimization procedure by augmenting it into the target vector. It is then optimized iteratively with L-BFGS using gradients computed from the physics informed loss function to the parameter $\hat{k}$ with automatic differentiation.

2D Linear PDE

To show that the data-driven discovery method can be applied to higher dimensional systems, the same experiment is now repeated for the 2-dimensional heat equation, also using the same setup as described in the previous subsection. The same changes are made to this experiment by using the L-BFGS optimizer, random uniform sampling of the same training and collocation points from the analytical solution, and initializing the estimated parameter $\hat{k}$ from a standard normal distribution. As the dimension is

higher than the previous experiment, the number of training points has to be greatly increased to compensate.

3.5.3 Causal Training

This and the following subsections are meant to demonstrate more involved training methods and applications of PINNs. This subsection will demonstrate various enhanced techniques that can be applied during training to improve accuracy and robustness, in some cases also convergence speed. This also makes it possible to solve PDEs that standard PINNs have historically failed to properly learn, like certain chaotic systems.

These experiments are meant to demonstrate the theory outlined in the literature review from the previous chapter.

Simple PDE

Once again, consider the Burgers' equation defined as:

$$u_t + uu_x - \nu u_{xx} = 0 \tag{3.10}$$

with the parameter $\nu = \frac{0.01}{\pi}$, initial condition $u(0, x) = -\sin(\pi x)$ and periodic boundary conditions $u(t, -1) = u(t, 1) = 0$.

The purpose now is to demonstrate 3.5.1 how the learned solution can be improved from the previous experiment. Because the boundary conditions are defined to be periodic, this is enforced exactly using a Fourier embedding. The Fourier embedding uses 5 increasing frequencies of sines and cosines in addition to the DC-term, which means that the spatial input dimension of the network becomes 11, in addition to one extra time dimension. Because all the frequencies are set to be periodic on the boundary, the periodic boundary conditions are satisfied automatically without adding any extra boundary term to the loss function.

The neural network itself is implemented as the modified network structure with residual connections, which has been shown to improve gradient flow for many chaotic systems.

This is combined with the causal loss function (2.40) which prioritizes learning earlier time-domains before later time-domains, ensuring that causality in time is prioritized during training. The causal loss is created by dividing the time domain into 20 equally sized subdomains.

The full training procedure, each running a certain amount of epochs, is done for 5 different values of the ϵ hyperparameter. ϵ is first initialized to 0.01, and training is either done until max epochs is reached, or a convergence criterion depending on the loss weightings are met. ϵ is then multiplied by 10 for each subsequent training run, ending with a value of 100 for the final iteration. This makes it so that causality is enforced more for later training runs compared to the first ones. Each training run continues with the same network parameters from the previous run, thus each subsequent run can be thought of as an additional fine-tune that makes everything gradually more causal.

The final enhancement is to add time-marching. The overall time domain is split into 10 different equally sized subdomains, and a separate model is trained on each subdomain. This also means that each of these 10 subdomains are further divided into 20 subdomains from the causal loss. When one network finishes training, the next network is initialized with the parameters of the previous network before restarting on the next time subdomain. This also means that to get the output solution for the whole domain, the outputs of each submodel must be combined together.

Chaotic PDE

Burgers' equation was already possible to solve accurately without the causal enhancements, so the vastly increased training time is not necessarily justified. The previous experiment was meant to show that the new techniques actually work, before attempting a more complicated equation.

Now consider the 1-dimensional Allen-Cahn equation defined as:

$$u_t - 0.0001u_{xx} + 5u^3 - 5u = 0 \tag{3.11}$$

on the domain $0 \leq t \leq 1$ and $-1 \leq x \leq 1$, using initial condition: $u(0, x) = x^2 \cos(\pi x)$ and periodic boundary conditions: $u(t, -1) = u(t, 1)$ and $u_x(t, -1) = u_x(t, 1)$.

This is an example of a chaotic PDE where standard PINNs fail to properly learn the true solution, causing the error between the true solution and the PINN solution to increase along the time axis.

Training a PINN with the same enhanced training techniques as used for the Burger's equation in the previous experiment makes it possible to accurately learn the Allen-Cahn equation. A modified network structure is used alongside Fourier embeddings, this time using 10 increasing frequencies resulting in an input dimension of 22. The causal loss is subdivided into 100 time-domains. Time-marching is not used, as this is one of the more computationally demanding techniques, and is instead replaced with more collocation points and network parameters to compensate. The ϵ hyperparameter is set to the constant 100 instead of iterating through a list.

To verify the accuracy of the solution, the trained PINN is compared to a dataset obtained from a traditional numerical solver [58].

3.5.4 Symbolic Operator Discovery

The previous subsection on data-driven discovery of PDEs demonstrated a use case of PINNs where they perform better than traditional numerical solvers. In that case, the dynamics of the system was known in the form of the full PDE, with the addition that there are unknown parameters to be learned in addition to the full solution.

Full system equations can in many cases be explicitly derived from first principles, which makes the previous experiment useful in many cases. However, when using first principles there are always many assumptions and simplifications being made to make it possible to model the true system. In some cases, this can lead to a model that is not accurate enough to be useful for a given purpose. It is also possible that an equation is derived explicitly, but that some terms of the equation are missing.

The purpose of the following experiment is to make it possible to use data to discover new dynamics and possibly unknown terms of an equation. The symbolic operator discovery method described in the literature review is here used to learn a missing term of a PDE from data.

Nonlinear PDE

Burgers' equation is used yet again to demonstrate the method. Burgers' equation is now assumed to be on the form:

$$u_t - \nu u_{xx} = 0 \tag{3.12}$$

with $\nu = \frac{0.01}{\pi}$ and where the term uu_x is missing from the equation. The goal is to re-learn this term from data. The initial condition is set to the usual $u(0, x) = -\sin(\pi x)$ and boundary conditions are assumed to be periodic.

Because the boundary is assumed to be known explicitly, it is not necessary to use a dedicated network to learn this. The solution network and the unknown dynamics network are both set to standard neural networks with the same number of layers and hidden units, and then trained together. The unknown dynamics has an input dimension of 3, corresponding to the terms: u, u_t and u_x. The true term is equal to the product of u and u_x, meaning that u_t is not actually needed. It is added as an input here, because in real applications it is not known beforehand which terms that goes in the unknown dynamics.

To verify the accuracy of the experiment, a separate PINN is trained on the Burgers' equation using all the same enhanced training techniques described in the previous subsection on causal training, except time-marching to make it easier to compare. The output of this PINN is then compared against a dataset obtained from a traditional numerical solver [59]. For simplicity, this PINN is then used as the ground truth for comparing the learned dynamics in this experiment.

3.5.5 Solving PDE-Constrained Optimal Control Problems

The next set of experiments are meant to demonstrate a practical application of PINNs, and show that the same framework can solve many different types of problems. However, one drawback with this approach compared to traditional numerical methods is that there are fewer guarantees with regards to optimality of solutions and general accuracy of the solutions.

Flux Control

For the first experiment, consider a system governed by the two-dimensional Laplace equation:

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 0 \tag{3.13}$$

defined on the square domain: $(x, y) \in [0, 1] \times [0, 1]$, with boundary conditions:

- $u(x, 0) = \sin(\pi x)$
- $u(x, 1) = c(x)$
- $u(0, y) = u(1, y)$
- $\frac{\partial u}{\partial x}(0, y) = \frac{\partial u}{\partial x}(1, y)$

with a periodic boundary in the x-direction, and where the function $c(x)$ is a control input that affects the entire top side of the domain.

Now consider the optimal control problem with all the above equations as constraints along with the objective function:

$$J = \int_0^1 \left[\frac{\partial u}{\partial y}(x, 1) - q_d(x)\right]^2 dx \tag{3.14}$$

where $q_d(x) = \cos(\pi x)$ is the desired flux. The optimal control problem can be thought of as finding the control input $c(x)$ that results in the desired flux at the top side of the boundary, subject to the dynamics and boundary conditions.

The problem is solved using the method outlined in [53]. Two neural networks are trained simultaneously, the first represents the state u and the second represents the control input $c(x)$. The final loss function to be optimized is a weighted sum of the boundary loss, physics loss and objective function. The integral for the objective function is computed numerically using the trapezoid method of integration using 41 point evaluations along the integral. For this experiment, the weightings of the boundary and physics loss are set to be 1, with a weighting of the objective function set to 100.

Dirichlet Boundary Control

For this next experiment consider the optimal control problem with dynamics governed by the 1-dimensional heat equation:

$$\frac{\partial u}{\partial t} - \frac{\partial^2 u}{\partial x^2} = 0 \tag{3.15}$$

with initial condition:

$$u(0, x) = \sin(\pi x) \tag{3.16}$$

and boundary condition:

$$u(t, 0) = c(t) \tag{3.17}$$

for a control input $c(t)$. This means that the control input can influence the dynamics along the bottom boundary of the domain. The objective function is here defined to be:

$$J = \sum_{k=1}^{N} \left[u(x_k) - u_d(x_k)\right]^2 \tag{3.18}$$

for a desired temperature distribution $u_d(x) = 0.5$. The objective function could be replaced with an integral and computed with the trapezoid method, similarly to the previous experiment. However, simply evaluating the function at a set of points and comparing these with the MSE loss works well enough. The objective function is evaluated at the final time of the temporal domain. This makes it so the system does not necessarily have to approach the target distribution as fast or as energy efficient as possible, which are usually desirable properties of control systems, but could also make the learning procedure more difficult as the the three losses are summed together.

Neumann Boundary Control

Now consider the same experiment as above, except that the boundary condition is given as:

$$\frac{\partial u}{\partial x}(t, 0) = c(t) \tag{3.19}$$

which is usually a more realistic scenario, as influencing the temperature distribution directly without any inertia is more difficult to do than setting the flux along the boundary.

Initial Control

For the final experiment here, consider the system governed by Burgers' equation:

$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} = \nu\frac{\partial^2 u}{\partial x^2} \tag{3.20}$$

with $\nu = 0.01$. The domain is set to be $(t, x) \in [0, 5] \times [0, 4]$. Periodic boundary conditions are used.

It can be verified that the following analytical solution satisfies Burgers' equation [53]:

$$u_a(t, x) = \frac{2\nu\pi e^{-\pi^2\nu(t-5)}\sin(\pi x)}{2 + e^{-\pi^2\nu(t-5)}\cos(\pi x)} \tag{3.21}$$

so that the initial condition can be inferred to be: $u(0, x) = u_a(0, x)$. This analytical solution is visualized in Figure 3.4.

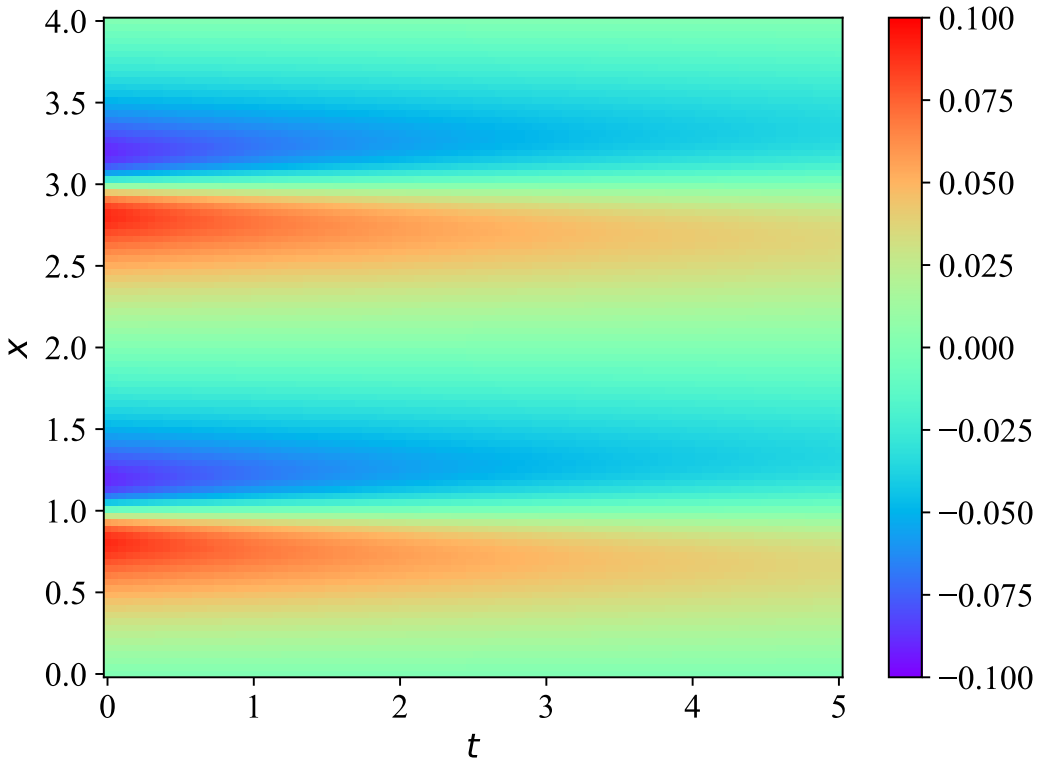


Figure 3.4: An analytical solution of Burgers' equation.

Now consider the optimal control problem by setting the initial condition to:

$$u(0, x) = c(x) \tag{3.22}$$

with control input c(x). The objective function is set to be:

$$J = \frac{1}{2}\int_0^4 \left[u(5, x) - u_a(5, x)\right]^2 dx \tag{3.23}$$

This means that the goal of the optimal control problem is to find the initial condition that results in the final state defined by $u_a(5, x)$. As solutions are unique, this implies

that the resulting solution to the overall control problem will also equal $u_a(t, x)$ on the entirety of the domain.

The integral of the objective function is again computed numerically using trapezoid integration with 41 point evaluations.

3.5.6 Regularization with the Maximum Principle

The following experiment is based on a novel idea from this master project. It is a way to make the training loss converge faster, as well as make it possible to use fewer collocation points, which overall results in faster PINN training.

Elliptic PDE

A function is called harmonic if it satisfies Laplace's equation:

$$\nabla^2 u = 0 \tag{3.24}$$

A function is called subharmonic if it satisfies the following inequality:

$$\nabla^2 u \geq 0 \tag{3.25}$$

and the opposite inequality results in a superharmonic function. Harmonic functions are therefore both subharmonic and superharmonic [26].

For a subharmonic function u defined on a bounded domain $\Omega \subset \mathbb{R}^n$, the strong maximum principle says that:

$$\max_{\overline{\Omega}} u = \max_{\partial\Omega} u \tag{3.26}$$

which either means that the maximum value of u is attained at the boundary of the domain, or that u is constant on the domain [26]. This can be thought of as analogous as to how convex functions have their maximum values on the boundary of their domain, as long as the domain is also convex.

Similarly for superharmonic functions:

$$\min_{\overline{\Omega}} u = \min_{\partial\Omega} u \tag{3.27}$$

which also means that for a harmonic function u:

$$\min_{\partial\Omega} u \leq u \leq \max_{\partial\Omega} u \tag{3.28}$$

meaning that both the maximum and minimum values are attained at the boundary of the domain.

This can be used during PINN training by adding regularization terms to the loss function. The maximum regularization term can be computed by first finding the maximum value along the boundary:

$$u_{\texttt{max}} = \max_{\partial\Omega} u \tag{3.29}$$

and then penalizing values of u that are above the max value on interior points:

$$\mathcal{L}_{r^+} = \sum_{i=1}^{N_f} \left[\max(u(t_f^i, x_f^i) - u_{\texttt{max}}, 0)\right]^2 \tag{3.30}$$

using the same collocation points (t_f^i, x_f^i) as the physics informed loss. If $u(t_f^i, x_f^i) - u_{\texttt{max}} > 0$ for any interior point, this is added to the regularization sum. If it is smaller than zero, it is set to 0 from the max statement and therefore not added to the regularization. The max value is also computed again for every training iteration. If the boundary conditions are known explicitly, it could be possible to use a constant max value computed from this.

The minimum regularization is done similarly by first computing the minimum value on the boundary:

$$u_{\texttt{min}} = \min_{\partial\Omega} u \tag{3.31}$$

and then summing up points:

$$\mathcal{L}_{r^-} = \sum_{i=1}^{N_f} \left[\min(u(t_f^i, x_f^i) - u_{\texttt{min}}, 0)\right]^2 \tag{3.32}$$

Both maximum and minimum regularization use values raised to the second power. For the minimum regularization, this serves as a way to make the values positive, but the maximum values are always positive. Raising to the second power is also differentiable, unlike the alternative solution of using an absolute value for the minimum. This can also be thought of as similar to how L1 versus L2 regularization on network parameters leads to different outcomes. L1 regularization tends to drive the parameters closer to zero, while L2 places less importance on smaller values and more on removing the big outliers.

Combining the maximum and minimum regularization terms leads to the overall regularization:

$$\mathcal{L}_r = \mathcal{L}_{r^+} + \mathcal{L}_{r^-} \tag{3.33}$$

To test the effectiveness of this new regularization term, consider Laplace's equation in two dimensions and the following analytical solution:

$$u_a(x, y) = \cos(\pi x)\sinh(\pi y) \tag{3.34}$$

defined on the domain $(x, y) \in [0, 1] \times [0, 1]$. The analytical solution can be derived using separation of variables and is easily verifiable to satisfy the equation. It is visualized in Figure 3.5. This analytical solution can then be used to obtain boundary conditions and verify the accuracy of the trained PINNs.

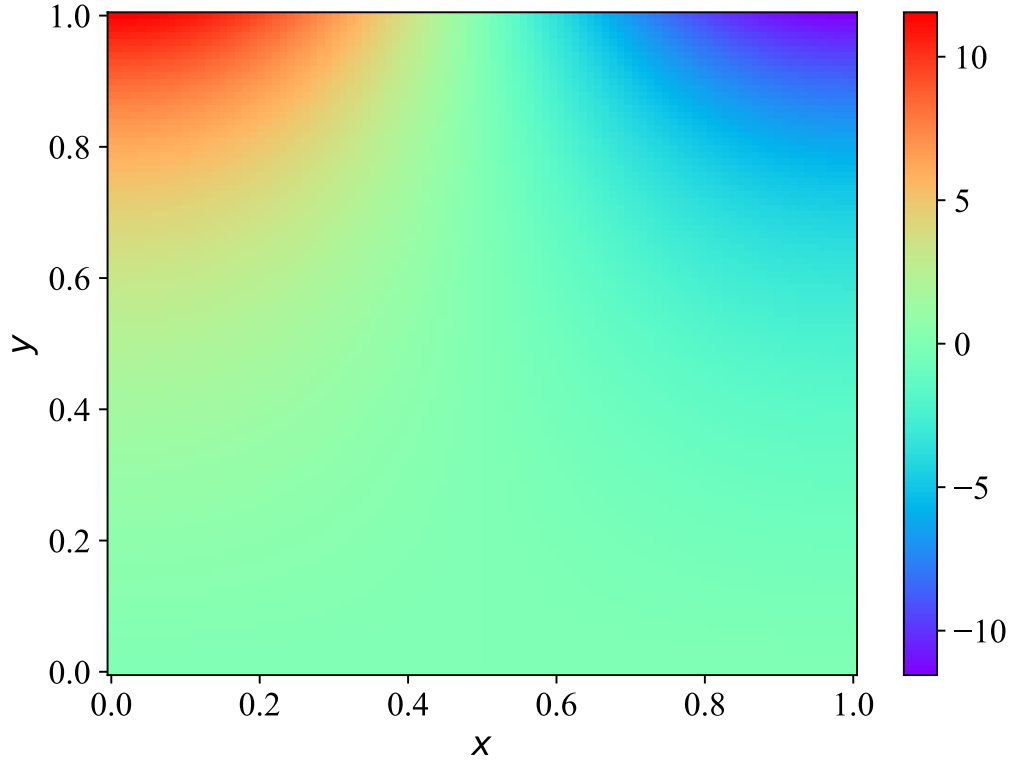


Figure 3.5: An analytical solution of Laplace's equation.

Two PINNs with the same number of parameters are trained independently of each other on Laplace's equation with boundary conditions obtained from the analytical solution. One of the PINNs is trained with the standard PINN setup, while the other one has the exact same setup and hyperparameters with the addition of the regularization term (3.33).

Afterwards, both PINNs are trained again from scratch, only this time using much less collocation points with a reduction from 10000 to 200.

For PDEs that are more complicated than Laplace's equation, it is also possible to generalize a weaker version of the maximum principle. This requires that the PDE is uniformly elliptic. For a second order linear differential operator L, defined as:

$$L = -\sum_{i=1}^{n}\sum_{j=1}^{n} a_{ij}(\boldsymbol{x})\frac{\partial^2}{\partial x_i \partial x_j} + \sum_{j=1}^{n} b_j(\boldsymbol{x})\frac{\partial}{\partial x_j} \tag{3.35}$$

then, L is elliptic if the eigenvalues of the matrix $\left[a_{ij}\right]$ are positive at each point $\boldsymbol{x}$, and L is uniformly positive if the smallest eigenvalue is bounded from below by a constant $\kappa > 0$ at each point $\boldsymbol{x}$. The difference between elliptic and uniformly elliptic is not that important for practical and numerical applications, and is mostly a necessity of mathematical rigor. So the maximum principle could be used for many different types of elliptic PDEs if they satisfy the above requirement.

Also note that the maximum principle for elliptic PDEs is unrelated to the similar sounding Pontryagin's maximum principle [60] from optimal control theory.

Finally, it should be mentioned that a similar maximum statement can be said about the heat equation, where the maximum value is attained either at the boundary of the

domain, or at the initial time [26]. Using this for faster training is less obvious to implement, and could be worth investigating further.

3.5.7 Causal Optimal Control

The final set of experiments use a combination of previous methods for an overall improvement. More specifically, the PINN approach to optimal control theory is combined with causality and related techniques to improve upon the training method.

Initial Control

The same initial control problem described above in the subection on optimal control theory is now re-visited with all the same improvements done when training on the chaotic PDE in the subsection on causal training. Causality leads to a more robust training in general, which should also be helpful in order to learn control policies more accurately.

Reversed Initial Control

For the problem with initial control, it might not necessarily make the most sense to apply causality forwards in time. This is because the initial condition is unknown while the final end state is known. So applying causality on an imperfect initial condition can in this case lead to a loss function with two terms that are competing against each other for most of the training. The term related to the causal physics information is trying to keep the initial condition accurate before moving on to the next time range. While the term related to the objective function of the optimal control problem wants to change the initial condition to fit the final end state, thus breaking the causal training.

A solution to this problem can be to re-formulate it by reversing the direction of the time t, and setting the desired final state as the initial condition. This also means that there is no longer any need for an explicit control policy, so this particular optimal control problem can be learned simply by causal training on the reversed time. The control policy corresponding to the initial condition for the actual problem then becomes equal to the end state of the reversed-time problem.

As the time range goes from $t = 0$ to $t = 5$, define the new reversed time variable as: $\tau = 5 - t$. By setting the state as a function: $u = u(\tau, x)$, then the partial derivative becomes:

$$\frac{\partial u}{\partial t} = \frac{\partial u}{\partial \tau}\frac{\partial \tau}{\partial t} = -\frac{\partial u}{\partial \tau} \tag{3.36}$$

The partial derivative for the spatial variable x remains unchanged. The time-reversed Burgers' equation then becomes:

$$-\frac{\partial u}{\partial \tau} + u\frac{\partial u}{\partial x} = \nu\frac{\partial^2 u}{\partial x^2} \tag{3.37}$$

which can now be used instead of the forward-time version of the equation for the physics informed loss.

4 Results and Discussions

4.1 Results

4.1.1 Learning Dynamical Systems with Physics Informed Neural Networks

Linear ODE

Training both a standard neural network and a PINN with the same number of parameters in the networks, and using them to plot their output trajectories are visualized in Figure 4.1. Both models are able to closely follow the true trajectory at the beginning, as it is a relatively simple system and does not require that much data. However, after $t > 0.4$ seconds the neural network output is diverging from the true output as it was not trained on any data past this, and is therefore not generalizing outside its training set. This can also be interpreted as overfitting on the training points. This is normal behavior from a standard neural network and was to be expected. One possible solution is to add more data, but in many real life cases this is not feasible.

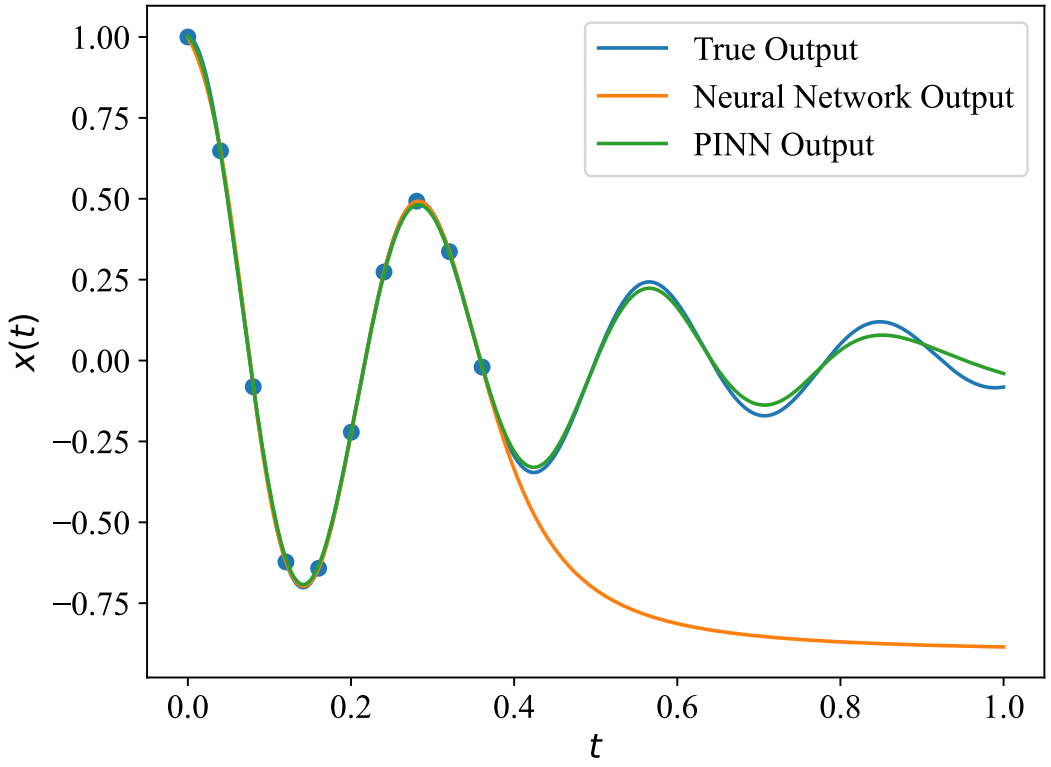


Figure 4.1: Output trajectories from a standard neural network and a PINN trained on a mass spring damper system. Datapoints are highlighted as blue dots.

Adding a physics informed regularizer on the PINN results in much better generaliza-

tion and follows the true system for much longer. Although the PINN is still not perfect towards the end, which is a consequence of the collocation points used during training. PINNs will not generalize further than their collocation points were sampled from, but as the collocation points can be generated at will when training it is necessary to first determine how long the PINN should stay accurate before training.

The validation loss for both models is computed by comparing the model trajectories with the true trajectory at the whole timeline with the MSE loss function. The validation loss per training epoch is shown in Figure 4.2. The standard neural network converges relatively fast, and does not learn much more afterwards. The PINN is stagnant for around 2500 epochs at first, before starting to improve. This observation turns out to be important for training PINNs on ODEs, and is one of the reasons that this experiment requires as many training epochs as it does. Because the mass spring damper is a globally stable autonomous linear system, a valid solution to the equation (3.1) is the trajectory $x(t) = 0$. This solution does obviously not fit with the datapoints, but is the result of a local minimum during training which also happens to be much closer to the initial parameters of the neural network. The β parameter that trades off the data loss and physics informed loss was adjusted to a much lower value compared to many of the other experiments to make the data more important, and was necessary to learn anything at all. So surprisingly, it turns out that learning trajectories from stable linear autonomous ODEs, the easiest possible type of ODEs, are actually kind of difficult to do with PINNs.

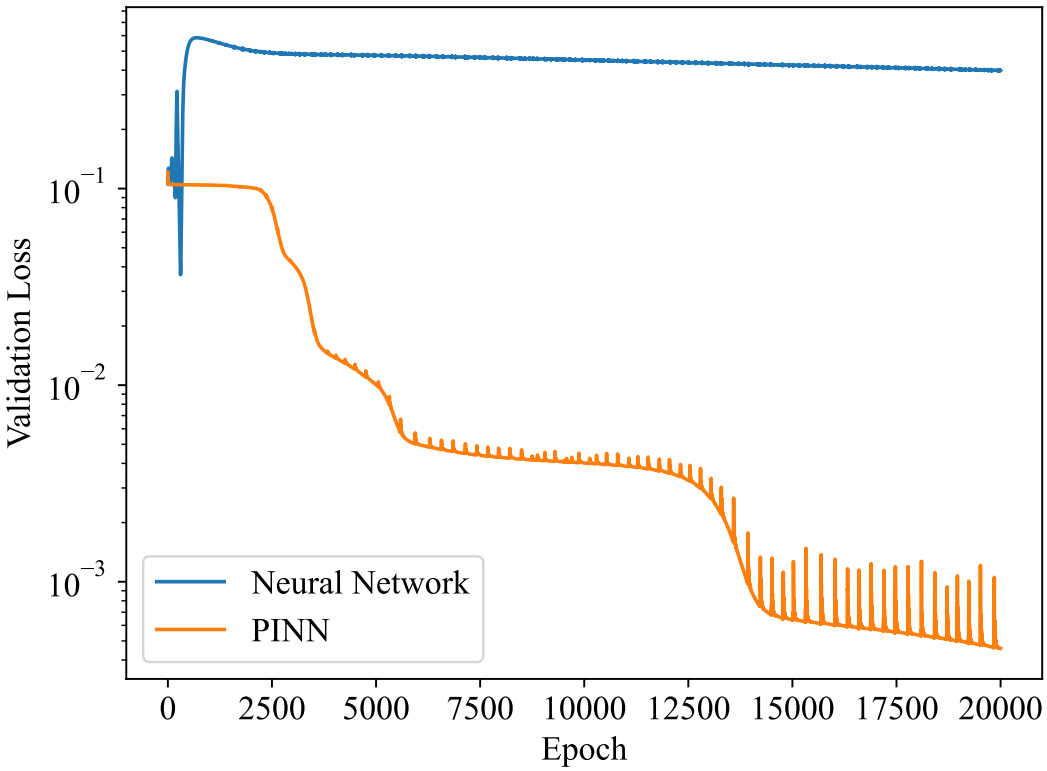


Figure 4.2: Validation loss from a standard neural network and a PINN trained on a mass spring damper system.

Next up, two identical PINNs were trained on the same mass spring damper datapoints as before, except now that one PINN has $N_f = 100$ collocation points as before, and the

other has $N_f = 30$ collocation points. Both sets of collocation points are sampled from the same time interval uniformly. The output trajectories are visualized in Figure 4.3.

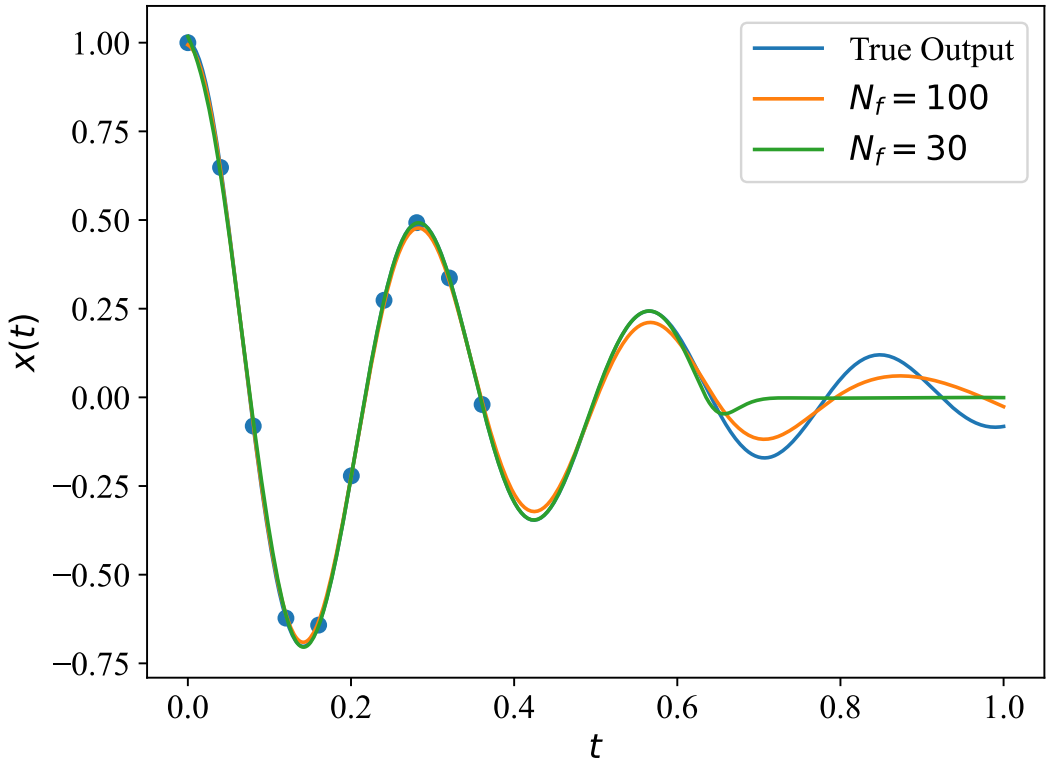


Figure 4.3: Output trajectories from two PINNs trained on a mass spring damper system with different number of collocation points. Datapoints are highlighted as blue dots.

Both PINNs are following the true output for some time after the training data ends, but it can be seen that the PINN with the most collocation points is also the one that generalizes better. This could indicate that increasing the number of points leads to a more robust model in general. One drawback of increasing the points however is the computational training time as the model has to be differentiated with respect to the prior ODE at every point. The validation loss from training these two PINNs are shown in Figure 4.4.

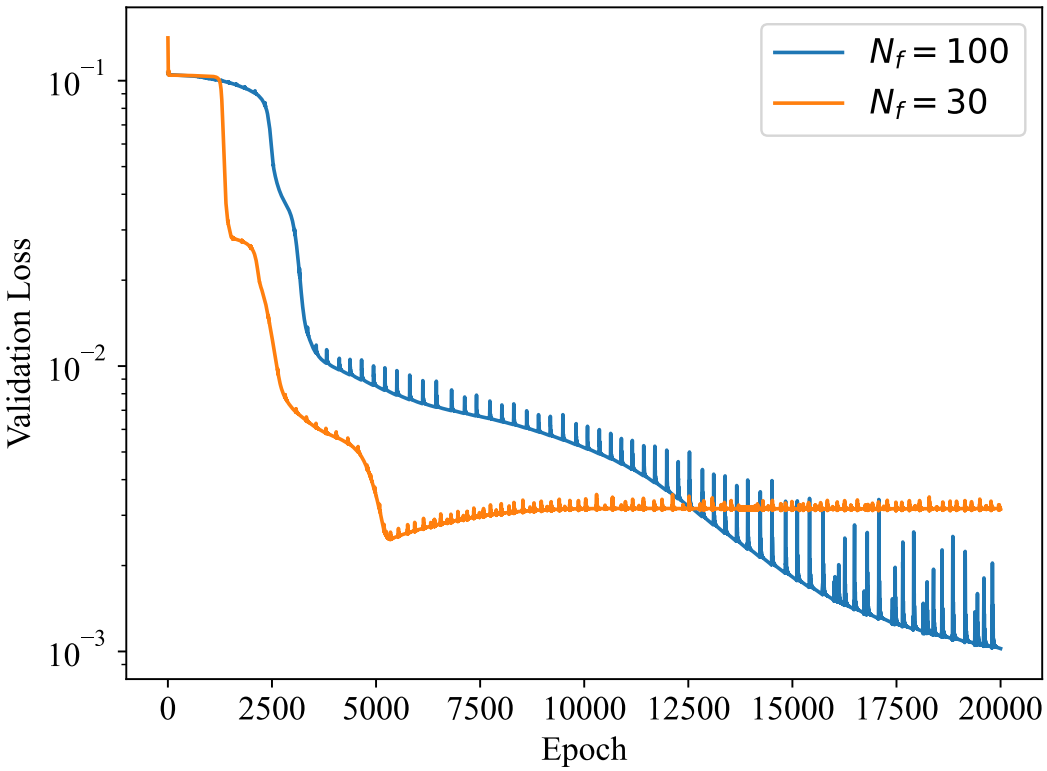


Figure 4.4: Validation loss two PINNs trained on a mass spring damper system with different number of collocation points. Datapoints are highlighted as blue dots.

It can be seen that not only is the PINN with fewer collocation points much faster to train in terms of CPU time, but it also escaped the local minima $x(t) = 0$ faster. However, with enough training steps the PINN with more points does catch up and ends up at an even lower validation loss. The robustness of a PINN model is therefore dependent exclusively on the amount of time willing to spend on training. It also appears that if the number of training steps are expensive enough to be limited it is beneficial to reduce the number of collocation points, or put invertedly, the number of collocation points must increase alongside training epochs to maintain the relative robustness.

Nonlinear ODE

A standard neural network and a PINN trained on the Van der Pol oscillator and computing the output trajectories are shown in Figure 4.5. Both models are able to hit every datapoint perfectly, as expected with enough training. In this case the dynamics are more complicated which results in the standard neural network overfitting heavily on the relatively few datapoints in comparison. This could be solved by using more data, and possibly reducing the model complexity, but it will still struggle to generalize outside the training interval as seen in the previous experiment.

The PINN is following the true output almost perfectly with the same model complexity and datapoints as the standard neural network. As the dynamics are more complicated than the mass spring damper it is easier to train a PINN and can be done in fewer epochs and without trading off the data loss.

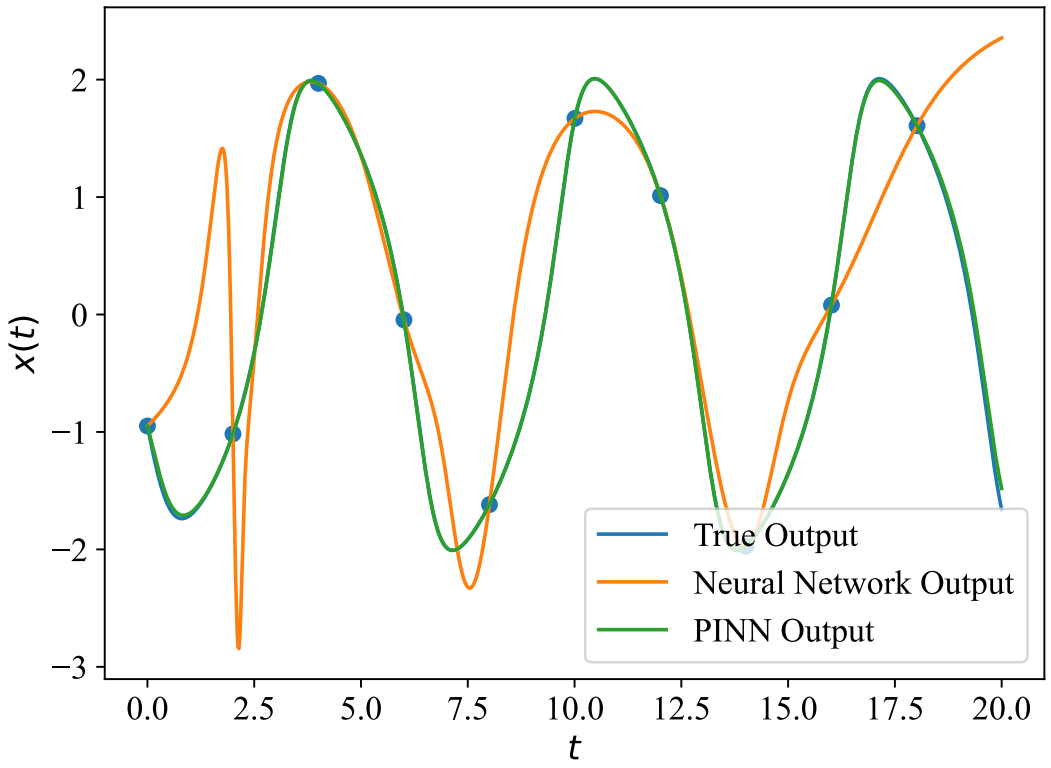


Figure 4.5: Output trajectories from a standard neural network and a PINN trained on a Van der Pol oscillator system. Datapoints are highlighted as blue dots.

Time-varying ODE

Training a PINN on the dynamics of the Riccati equation (3.3) without using any data at all results in Figure 4.6. As there is a time-varying equilibrium point at $x(t) = -\sqrt{t}$ which every trajectory converges to, the PINN trajectory also converges to this. The first initial value of the PINN trajectory is a random point coming from the initialization of the parameters.

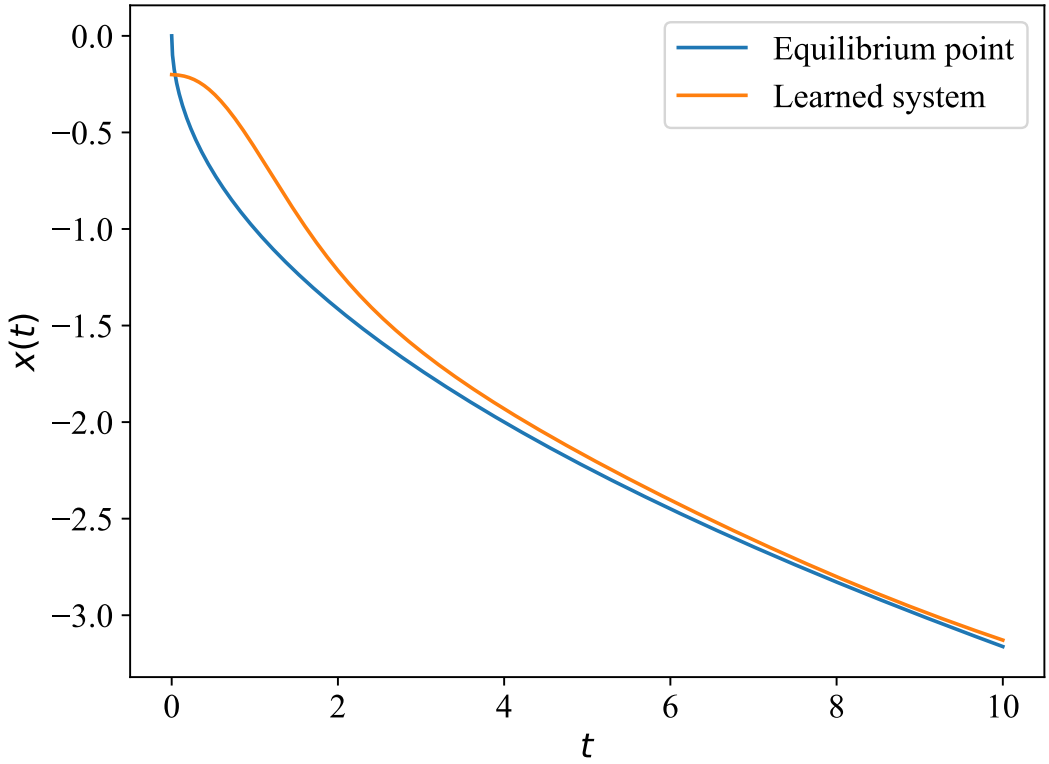


Figure 4.6: Training a physics informed neural network to learn the output trajectory of a Riccati equation without any datapoints.

Adding an initial condition as a single datapoint to the PINN training results in the trajectory visualized in Figure 4.7.

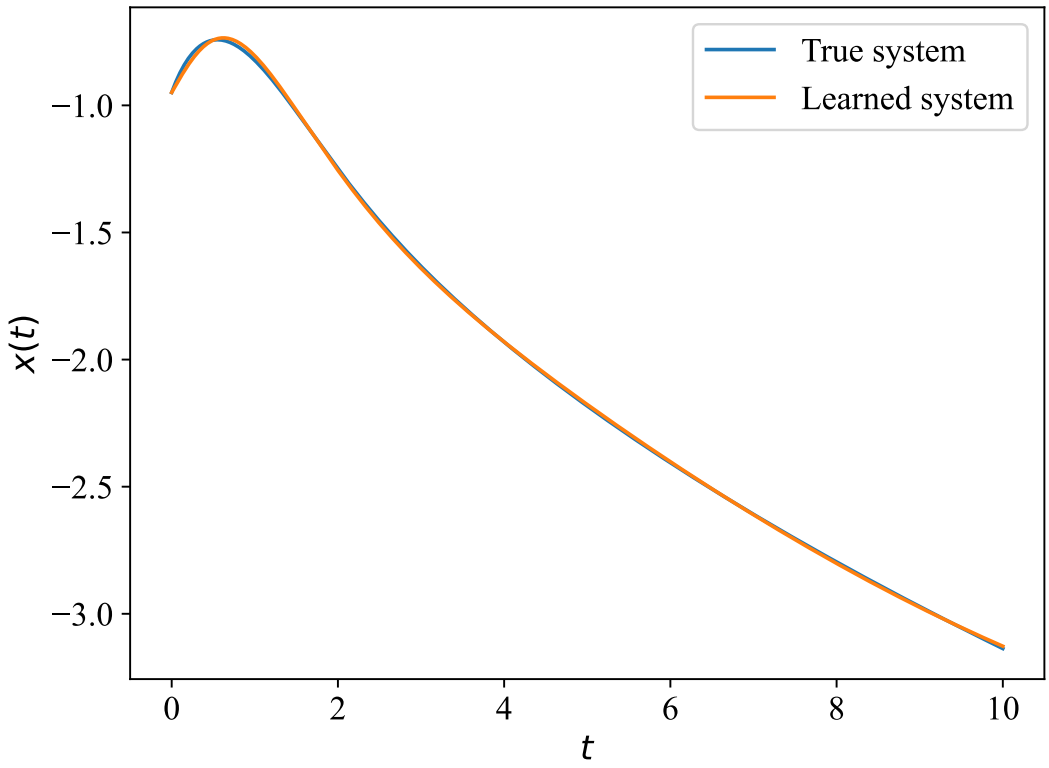


Figure 4.7: Training a physics informed neural network to learn the output trajectory of a Riccati equation based on a single datapoint.

The PINN is now following the true system almost perfectly for as long as the collocation points were sampled from. The result was also reached in only 1000 epochs, compared to the 20000 epochs necessary for the mass spring damper. It appears that more complicated dynamics makes it possible to get away with less data while still learning a robust model. In many real life systems modeled as nonlinear ODEs, small modeling inaccuracies can cause the system to behave in wildly unexpected ways. This is a major drawback of many nonlinear control systems [61] and must also be accounted for when training PINNs on prior dynamics.

1D Linear PDE

The PINN output after training on the 1-dimensional heat equation is shown in Figure 4.8. The initial and boundary conditions are learned accurately, as well as the general trend of the heat dissipation. The output is almost identical to the true solution shown in Figure 3.2. Computing the validation loss of the PINN can be done by comparing outputs against the true analytical solution throughout the whole spatio-temporal domain with the MSE loss function. The final validation loss value after training is complete is as low as: $6.2244 \cdot 10^{-5}$, which confirms the visual comparison that the PINN has learned the true system well.

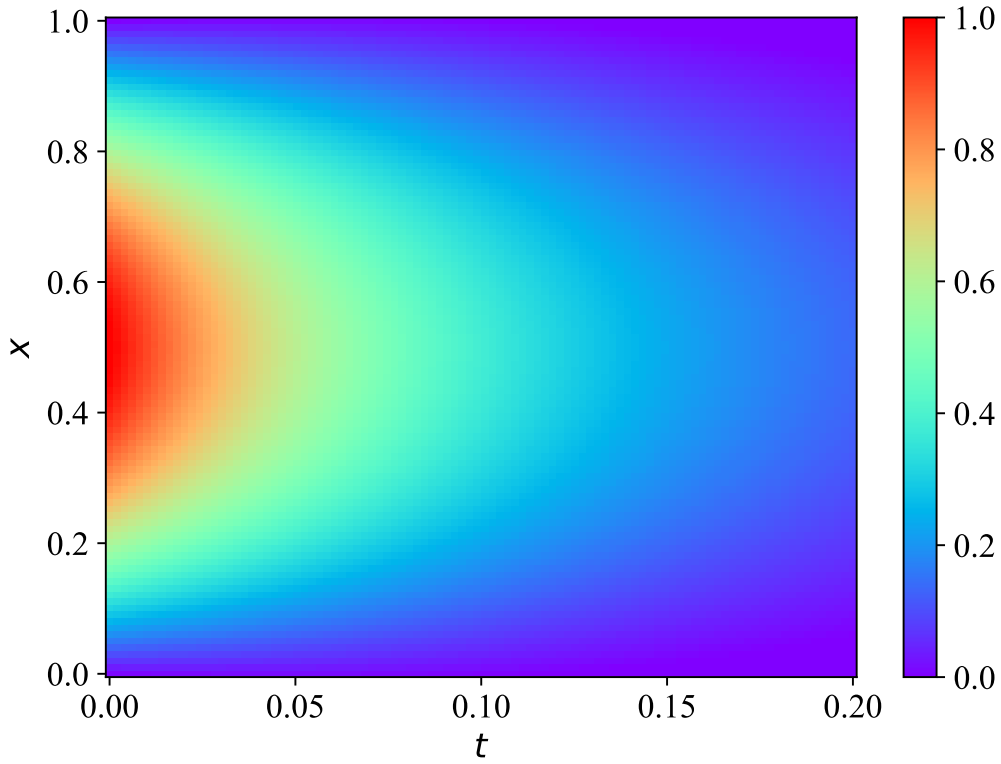
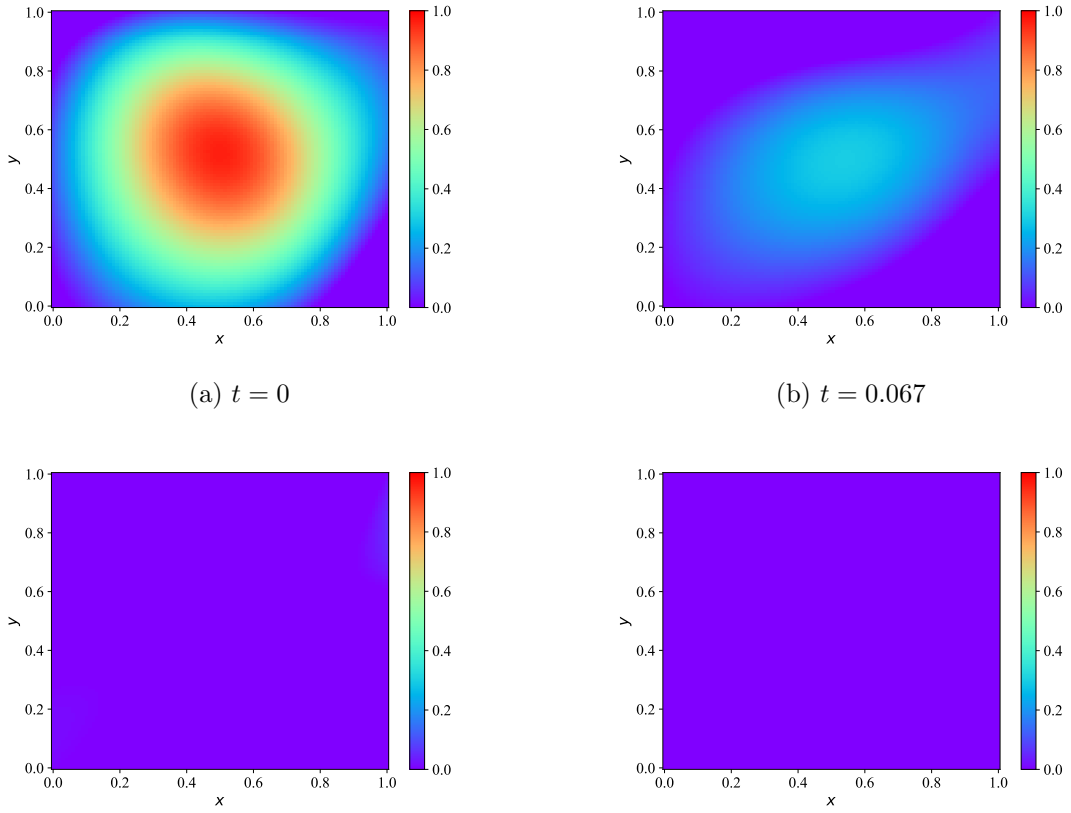


Figure 4.8: PINN output after training on a 1-dimensional heat equation.

The 1-dimensional heat equation is a relatively simple PDE so it is not surprising that the PINN is learning without any difficulties. It is also much easier to learn compared to simple ODEs, using fewer training epochs. The training data also consists entirely of points sampled from the boundary and initial conditions, which can be compared to the single datapoint used when training on the time-varying ODE. Adding interior points would make the training even easier and faster to converge.

2D Linear PDE

The setup now is identical to the previous experiment on the heat equation with the addition of an extra dimension. The extra dimension also requires more collocation points to fight back against the curse of dimensionality. The trained PINN output is seen in Figure 4.9 and is again able to learn the initial and boundary conditions well alongside learning the heat dissipation.



(a) $t = 0$ (b) $t = 0.067$

(c) $t = 0.133$ (d) $t = 0.2$

Figure 4.9: PINN output after training on a 2-dimensional heat equation.

Compared to the true solution it is very similar to the true solution, which is also confirmed by looking at the final validation loss at: 0.0405. It is however not as close to the true solution as the 1-dimensional case. The two experiments were trained on the same number of epochs, but even with the increased collocation points it is still not quite as accurate. The PINN might be trained for longer on even more collocation points to overcome this. It could also be beneficial to replace the optimizer with a better one, as was done for the data-driven discovery experiment. This does however show that even going from one to two dimensions on a simple PDE makes things much more difficult to train on.

Nonlinear PDE

At first all experiments were done using collocation points placed at a linearly spaced grid in the spatio-temporal domain, and the Adam optimizer for the training. However, the resulting output of training a PINN on the Burgers' equation was not accurate at all and seemed to be impossible with that setup. The collocation points were then changed to being sampled uniformly instead, which was also used for the previously discussed experiments, and the L-BFGS optimizer introduced here for the Burgers' equation.

Training a PINN with the improved setup worked, and the output now learns the characteristic shock formation of the Burgers' equation. This can be seen in Figure 4.10. As Burgers' equation is difficult to solve analytically it is also difficult to properly validate the solution without relying on numerical methods for solving PDEs. In this case, the PINN output was compared visually against the output from the paper by

the original authors of the PINN framework [33]. Vertical slices at specific points in time of the output are also visualized separately in Figure 4.11. This final experiment demonstrates the importance of having enough collocation points and a better optimizer for learning more difficult PDEs.

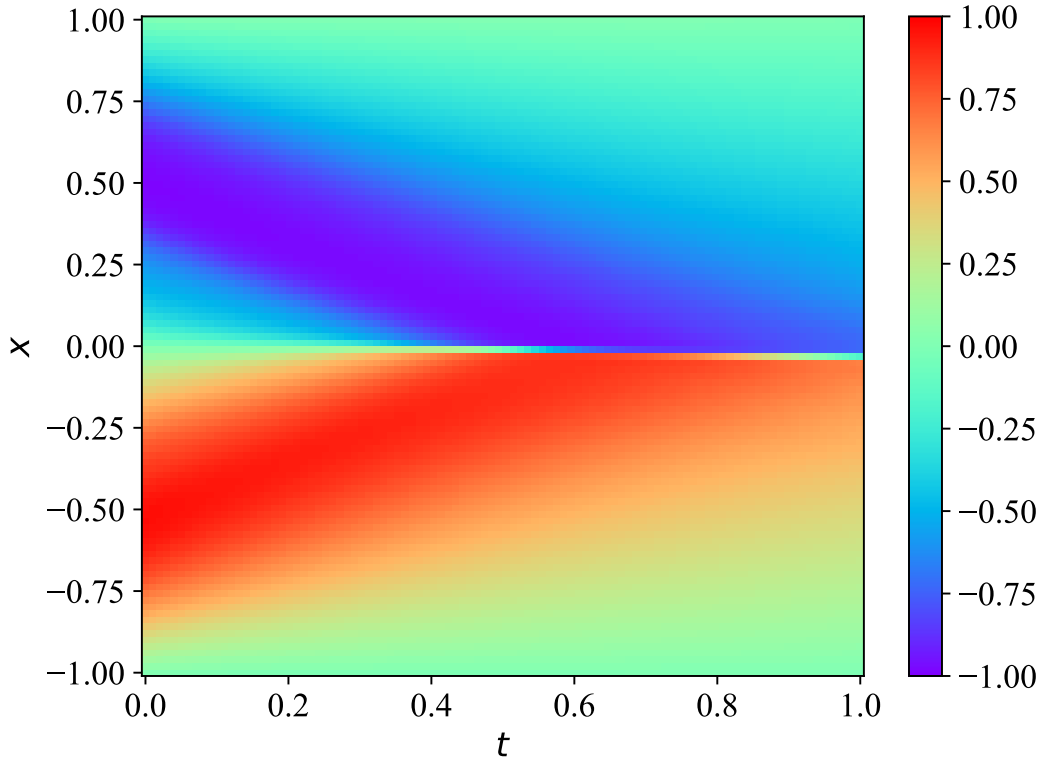
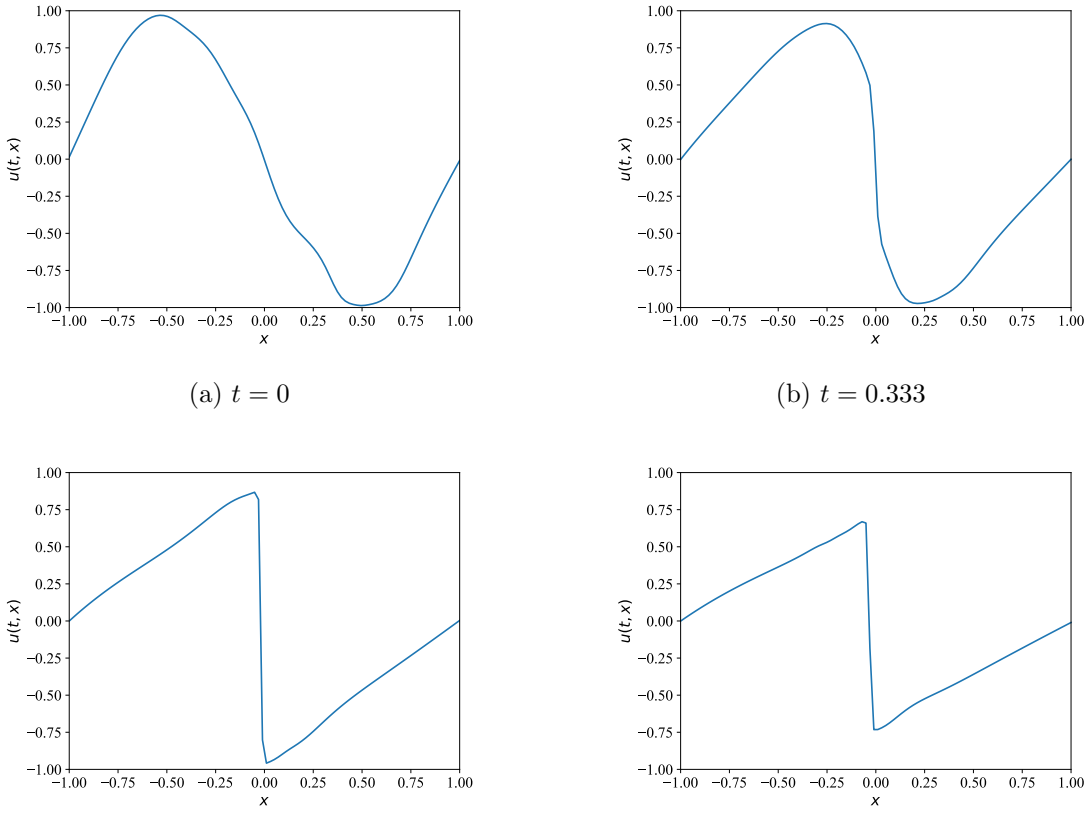


Figure 4.10: PINN output after training on a Burgers' equation.



(a) $t = 0$

(b) $t = 0.333$

(c) $t = 0.667$

(d) $t = 1$

Figure 4.11: Visualization of time-slices from the PINN output after training on a Burgers' equation.

4.1.2 Data-Driven Discovery of Dynamical Systems with PINNs

1D Linear PDE

The output from a PINN trained on the 1-dimensional heat equation is visualized in Figure 4.12. The output looks very similar to both the true solution and the previous experiment, which is also confirmed from the validation loss at: $2.0372 \cdot 10^{-5}$. The final validation loss here is even lower than the previous experiment with the known parameter, which could be a result of replacing the optimizer and increasing the amount of data to cover the interior of the domain.

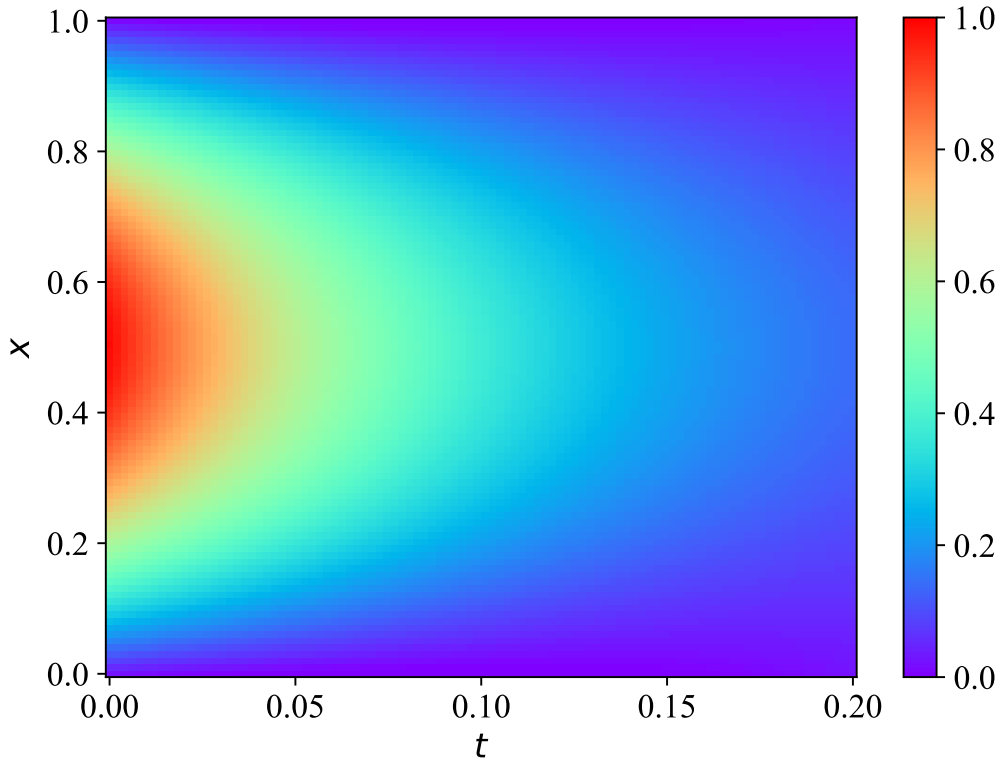


Figure 4.12: PINN output after training on a 1-dimensional heat equation with an unknown parameter.

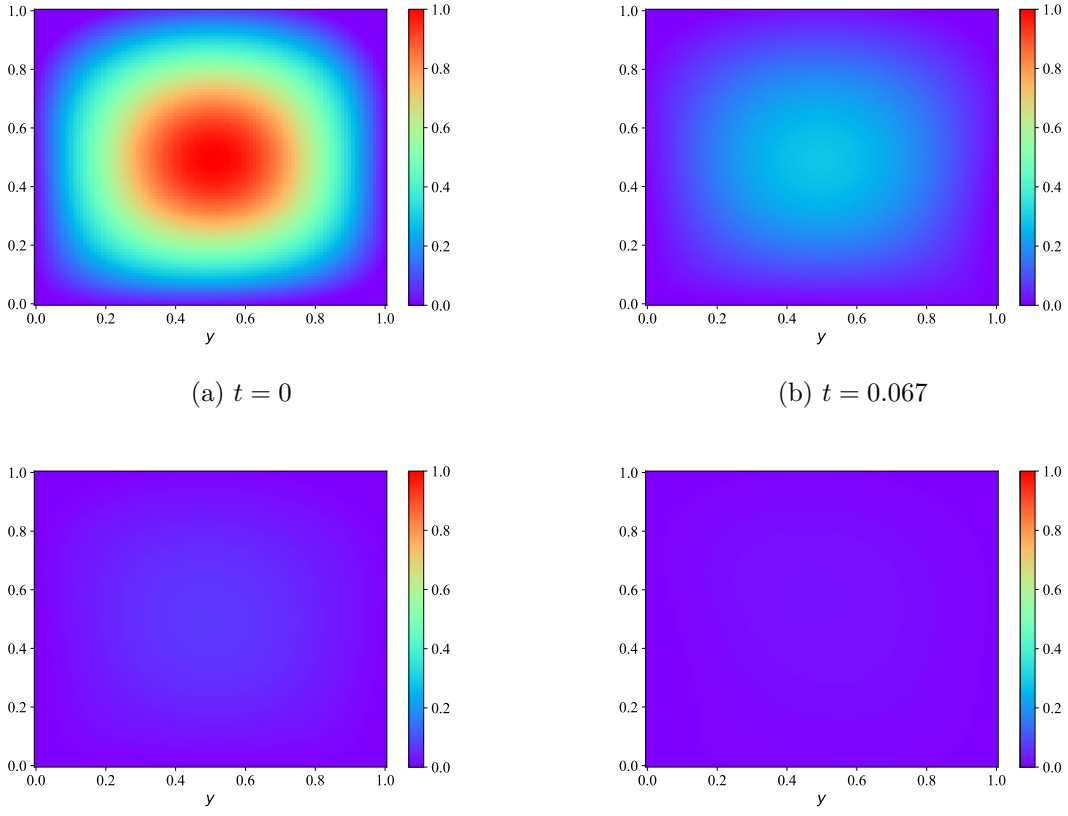
The true value of the parameter was set to $k = 1$ for simplicity, and the final estimate of the parameter after training was: $\hat{k} = 0.989268$, which is very close to the true value. An interesting thing that occasionally happened was that the estimate would converge towards -1 instead, which is equivalent for the heat equation (3.4) where the parameter is squared. The initial value of the estimate was sampled from a standard normal, so it would often converge depending on which side of zero it started at. Having prior information about the parameter could be useful in this case to ensure that it converges to the correct value.

The purpose of this experiment was less about learning the output of the system with a PINN and more about discovering the structure of the underlying dynamics. Because of the increase in data quantity required it might be feasible to train a neural network on the output data without using any prior physics information, but using PINNs it is possible to both learn a representation of the output while also simultaneously estimating the unknown parameters.

2D Linear PDE

The same experiment was now repeated for the 2-dimensional heat equation, and the final output of the trained PINN is shown in Figure 4.9. The output is again very similar to the true solution and output from the previous experiment with the 2-dimensional heat equation. The final validation loss after training is: $5.2727 \cdot 10^{-5}$, which is much lower than the previous experiment with the known parameter, which again could be explained by the change of optimization algorithm and increased data quantity.

The final estimate of the parameter after training ends up at: $\hat{k} = 0.993225$ compared to the true value of $k = 1$ which means that the PINN learned the true system well. However, increasing the dimension requires significantly more data than compared to the previous 1-dimensional case, where the number of training points and collocation points is increased from 1000 to 20000. This also increases the computational cost, but the most significant drawback is the reliance on training data, as more collocation points can always be generated. Collecting data from real systems can often be difficult, so it might not always be feasible to estimate unknown parameters with this method. Although, if there exists some prior information about the value of the parameter in addition to the system dynamics this could also be incorporated into the PINN training. For example if the parameter value itself is unknown, but has a known minimum and maximum value, this prior information can be used during training by clipping the parameter value to this range after every iteration. For the heat equation specifically it can for example be assumed that the heat conductivity is a positive value, thus limiting the range from below by zero.



(a) $t = 0$ (b) $t = 0.067$

(c) $t = 0.133$ (d) $t = 0.2$

Figure 4.13: PINN output after training on a 2-dimensional heat equation with an unknown parameter.

4.1.3 Causal Training

Simple PDE

With all the training enhancements of the modified network structure, Fourier embeddings, causal loss with epsilon annealing and time-marching, the resulting output from

solving the Burger's equation is shown in Figure 4.14. The complete output was created by stitching together the individual outputs from the smaller submodels. The quality is seen to be much higher compared to the previous more default approach shown in Figure 4.10.

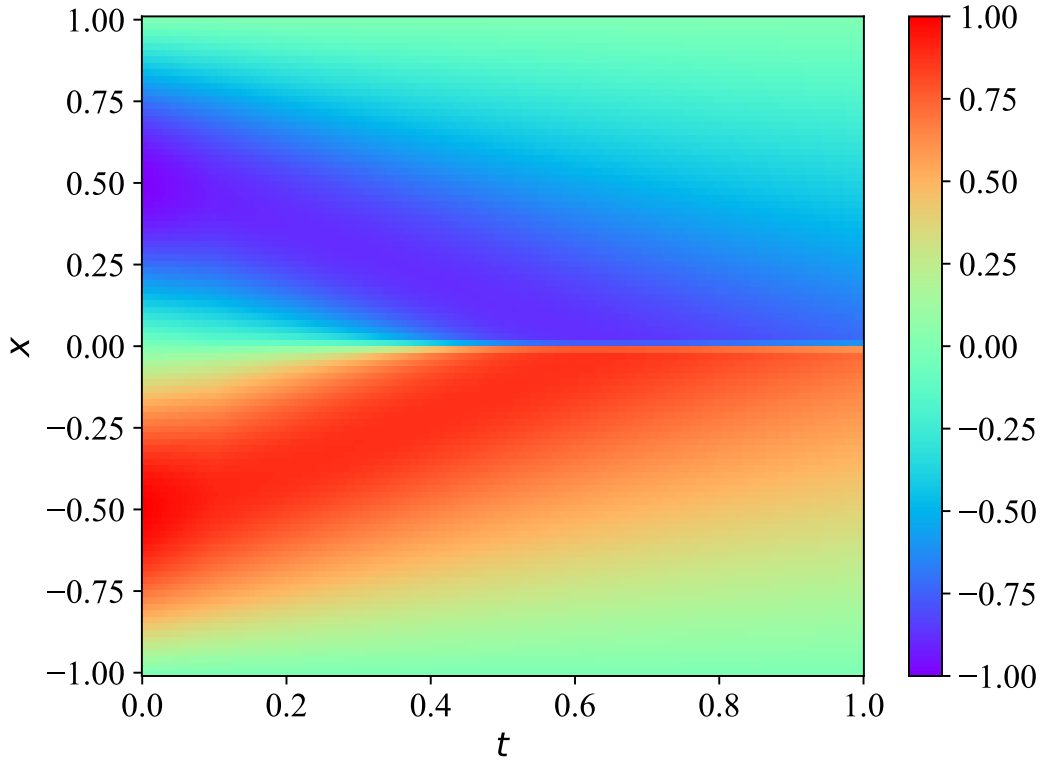
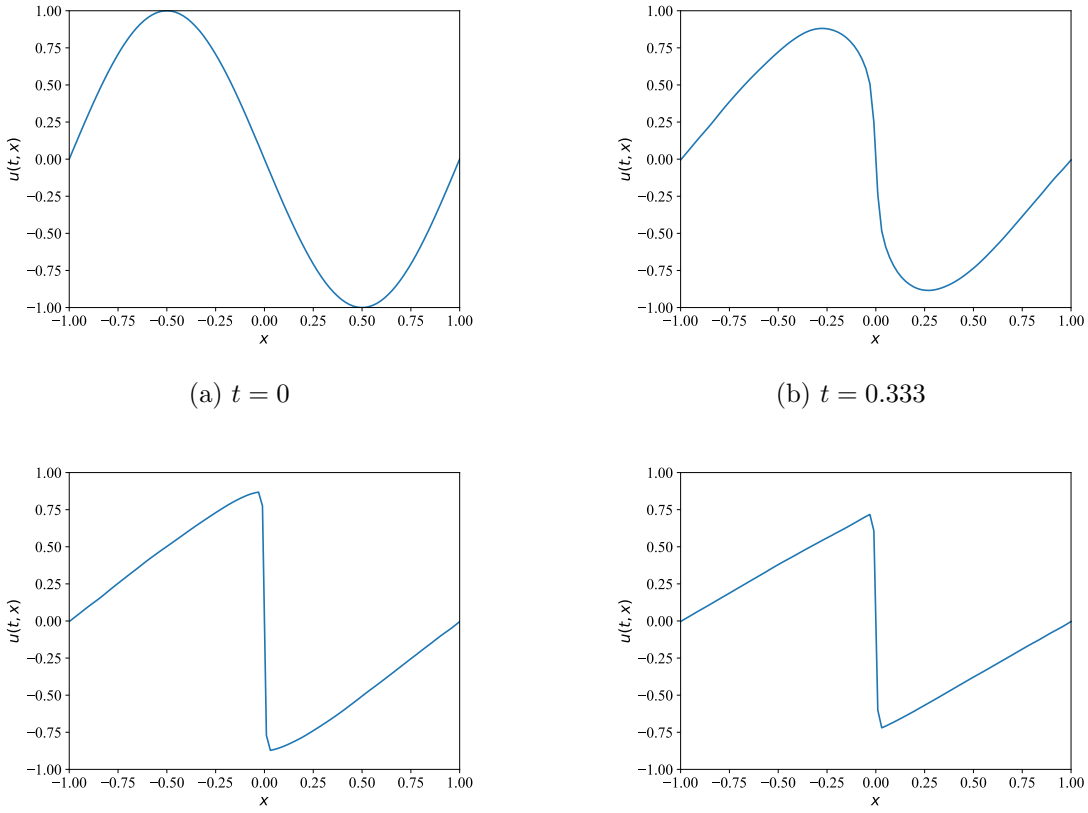


Figure 4.14: Burgers' equation solved with causal PINN training.

Visualizations of the output at specific slices in time are shown below in Figure 4.15. Comparing this to the slices from the default approach in Figure 4.11, the plots look more smooth and symmetric, which indicates a higher accuracy.



(a) $t = 0$

(b) $t = 0.333$

(c) $t = 0.667$

(d) $t = 1$

Figure 4.15: Visualization of time-slices from the PINN output after causal training on a Burgers' equation.

The purpose of this experiment was to show that the enhancements to the PINN training process work and give accurate results. But the increased computational cost may not be worth it for all problems.

Chaotic PDE

Moving on to the more complicated Allen-Cahn equation, training a PINN without any of the enhancements will generally work very poorly. The result of doing this is not shown here, as the plot would be just tending towards zero as the time increases. However, by incorporating some of the enhancements described in the method section and training a PINN, results in the output plot shown below in Figure 4.16.

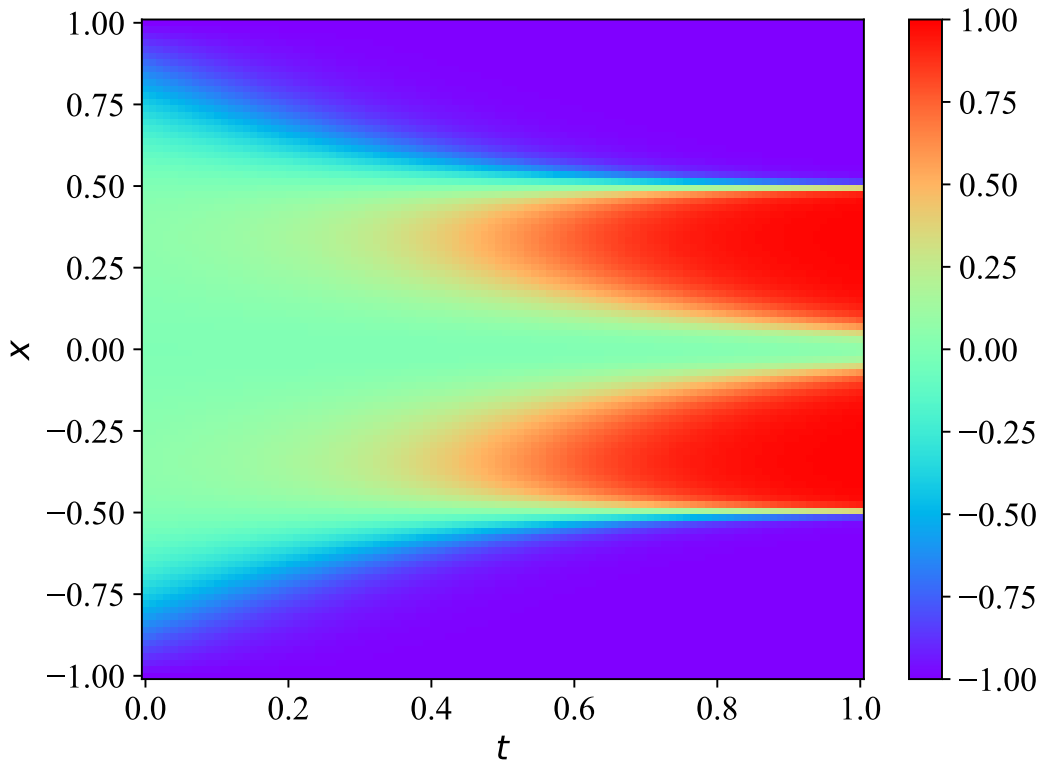


Figure 4.16: The Allen-Cahn equation solved with causal PINN training.

Comparing the pointwise MSE to a numerical solver, the resulting difference becomes: $3.2457 \cdot 10^{-6}$. This is an indication that the causal training method also gives a high accuracy for chaotic systems.

Numerically solving chaotic ODEs and PDEs are often hard to do with any accuracy for longer time intervals. For a system to be chaotic it means that small differences in initial conditions results in wildly different trajectories in time. Because no numerical method is perfectly accurate, for example due to floating point arithmetic on computers, small errors will accumulate during the numerical integration. These small errors might not mean much for a non-chaotic system except that the solutions become less accurate, but for chaotic systems, the solutions will get increasingly wrong with time, making them very unreliable.

Introducing causality is a way to combat this by ensuring that sections in time are accurate enough before proceeding to the next section in time. This prevents errors from accumulating, and reduces the chaotic influence during training.

4.1.4 Symbolic Operator Discovery

Nonlinear PDE

Firstly, a PINN is trained on yet another Burgers' equation. All the same improvements as used with the causal experiment are applied here, except time-marching. Comparing the MSE loss to the numerical solver results in the difference: $2.8914 \cdot 10^{-5}$. This PINN model is then used as the ground truth to validate the accuracy of the symbolic operator discovery. This is done partially because the trained PINN model is a continuous function

that can be evaluated at any point, and is therefore not restricted to the specific grid discretization as the downloaded dataset used. Additionally, having access to the model itself makes it possible to compute partial derivatives with automatic differentiation, thus making the operator accuracy more reliable. The alternative would be to use finite differences on the numerical dataset, which makes it much less reliable. This insight can also be considered a useful application of PINNs compared to numerical solvers.

Afterwards, the symbolic operator discovery is done with two different models. The first is a default PINN. The second is a PINN with the additions of the modified network structure, which seems to generally be an improvement without any noticeable drawbacks, and with a Fourier embedding to handle the known periodic boundary. The second neural network that learns the missing term of the PDE symbolically is kept as a standard neural network for both models.

Plotting the outputs of any of these models would result in yet another plot of the Burgers' equation, which has been shown many times throughout this thesis, so it was not necessary to display here yet again.

To validate the accuracy, a grid of equally spaced points are constructed over the domain to evaluate at. The networks learning the output state of the PDE are compared with the MSE against the ground truth PINN described previously. The default PINN achieves an MSE of $1.28 \cdot 10^{-4}$, and the improved PINN achieves an MSE of $7.7 \cdot 10^{-5}$, which is a slight improvement.

To validate the learned symbolic operators, automatic differentiation is used on the ground truth PINN to calculate the relevant partial derivatives. These are then evaluated on the grid of validation points. This makes it possible to explicitly calculate the ground truth operator by combining the relevant terms together. The network that learns the symbolic representation receives these true partial derivatives and state as input, and the output becomes the learned operator. The operator can then be validated by comparing with the MSE over the grid points. The first model with the standard PINN achieves an MSE of 0.286, and the model with the improved PINN achieves an MSE of 0.194. The improved PINN still has a slight improvement, but neither of the models appear particularly accurate in this case.

So even though both models get a relatively accurate representation of the output, they appear to not learn the symbolic expression for the unknown term very well. Although the MSE values are not necessarily that big either, and it is to a certain extent dependent on the true scale of the values of the system. This can also happen because the MSE is computed as the mean over the validation points. And for Burgers' equation in particular, as there develops a discontinuity along the origin, the derivative at that point approaches infinity as time increases. This value might get large enough to run into numerical errors, but this was not investigated thoroughly. A potential solution could be to take the median value of the MSE instead of the mean, as it is more outlier resistant. But the conclusion is anyway that it is difficult to verify the accuracy of the learned unknown term.

As the two networks are trained together, the outputs from the symbolic network are used to calculate the physics informed loss also during training. If the network was set to a constant zero, then the physics informed loss would not reflect the true PDE as there is a term missing from the equation. Therefore it means that even though the learned symbolic operator in this case might not be that accurate, it is still accurate enough to allow the other network to represent the output state of the system. It is also possible that simply tuning the hyperparameters further would result in a more accurate expression for the unknown term.

4.1.5 Solving PDE-Constrained Optimal Control Problems

Flux Control

The output state of the trained PINN can be seen below in Figure 4.17. The boundary condition when $y = 0$ seems to be satisfied relatively well along with periodic boundary in the x-direction.

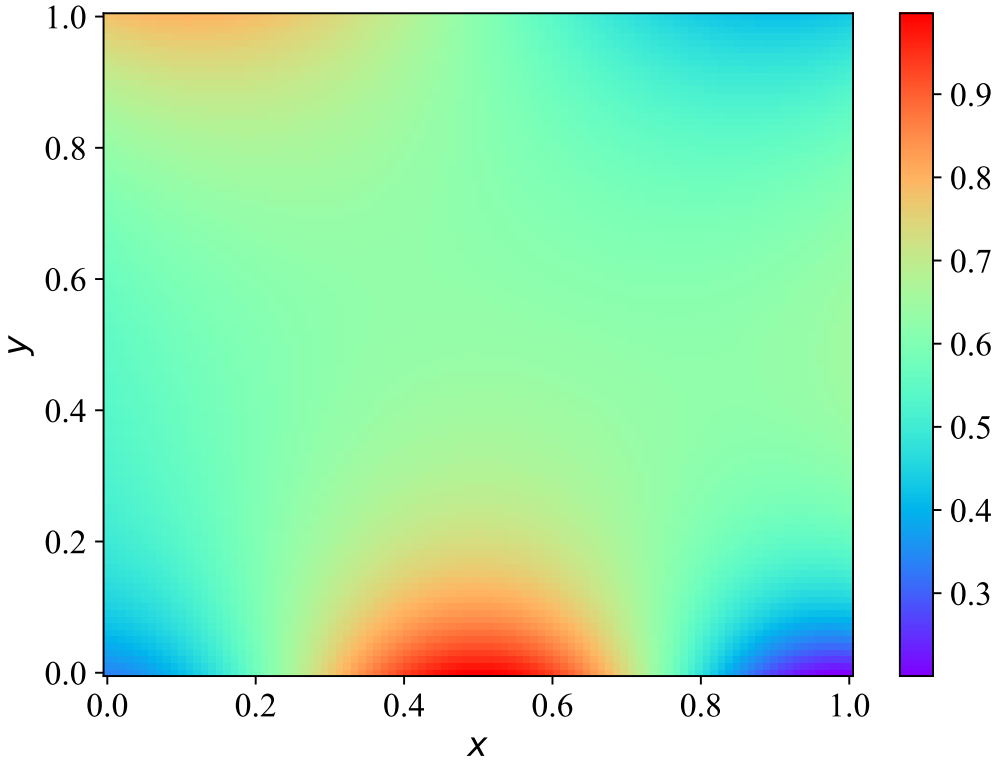


Figure 4.17: PINN output of the solution to the optimal control problem on Laplace's equation.

The learned control input is displayed in Figure 4.18, and is working as the boundary on the top side of the domain when $y = 1$. Verifying that the flux in the y direction goes to the desired flux is not obvious from looking at any of these plots, and must be verified in another way.

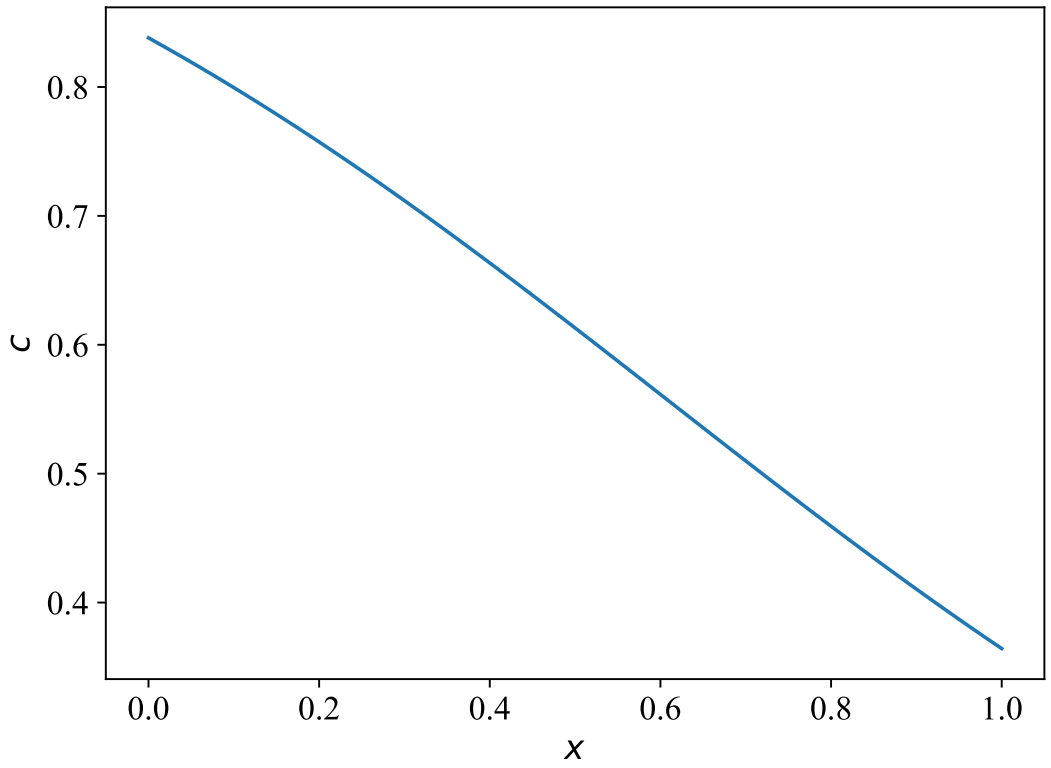


Figure 4.18: Learned control policy to the optimal control problem on Laplace's equation.

As both networks are trained together, they will have to learn to satisfy boundary and physics loss in addition to minimizing the objective function. By looking at the training loss values, they should all go to zero to perfectly represent the true solution to the optimal control problem. The individual training losses are shown in Figure 4.19.

It can be seen that the cost value corresponding to the objective function is prioritized during training, which makes sense due to its higher relative weighting. It also shows that the boundary loss and physics loss are much higher relatively. This can partially be explained because the periodic boundary condition is not possible to satisfy with the learned control policy in Figure 4.18, as that is also not periodic. Can also see that the losses converge after around 4000 epochs to a local minima, which is because the L-BFGS method doesn't use any momentum.

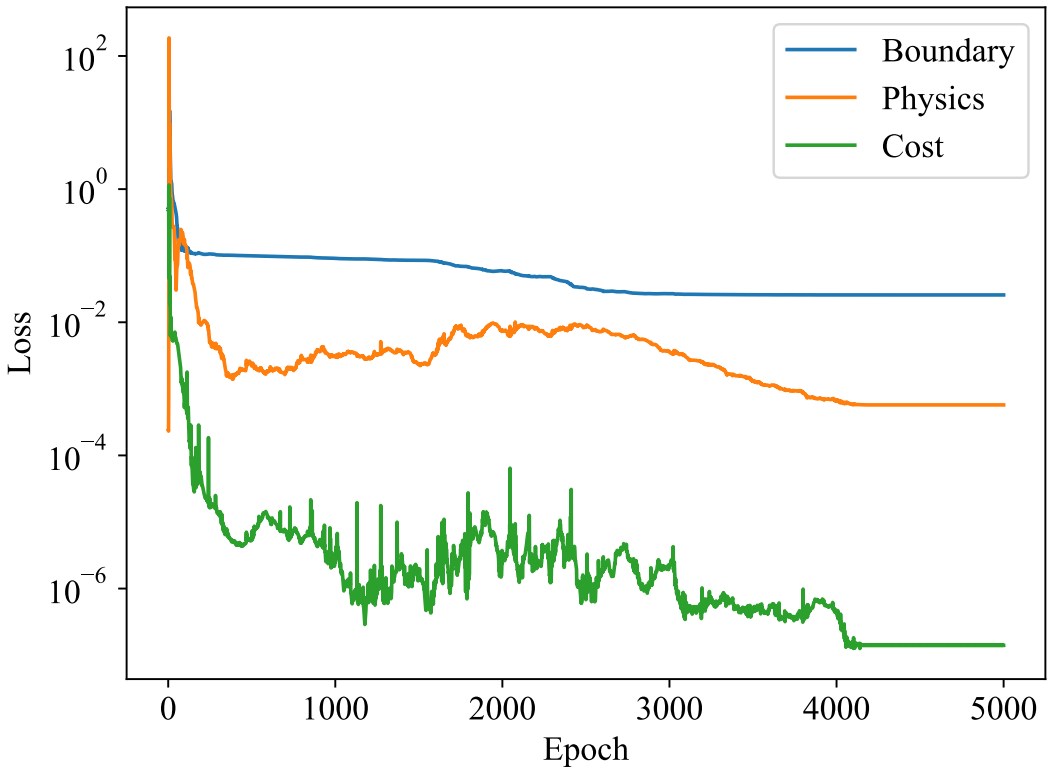


Figure 4.19: Training losses when training for the optimal control problem on Laplace's equation.

Another way to validate the learned control policy is to train a new PINN where the control policy is kept constant during training. For this experiment, this is equivalent to solving a standard boundary value problem. The objective function is not used for training, but is instead measured and used as a performance metric during training.

Doing this results in the output solution plot in Figure 4.20 and losses in Figure 4.21.

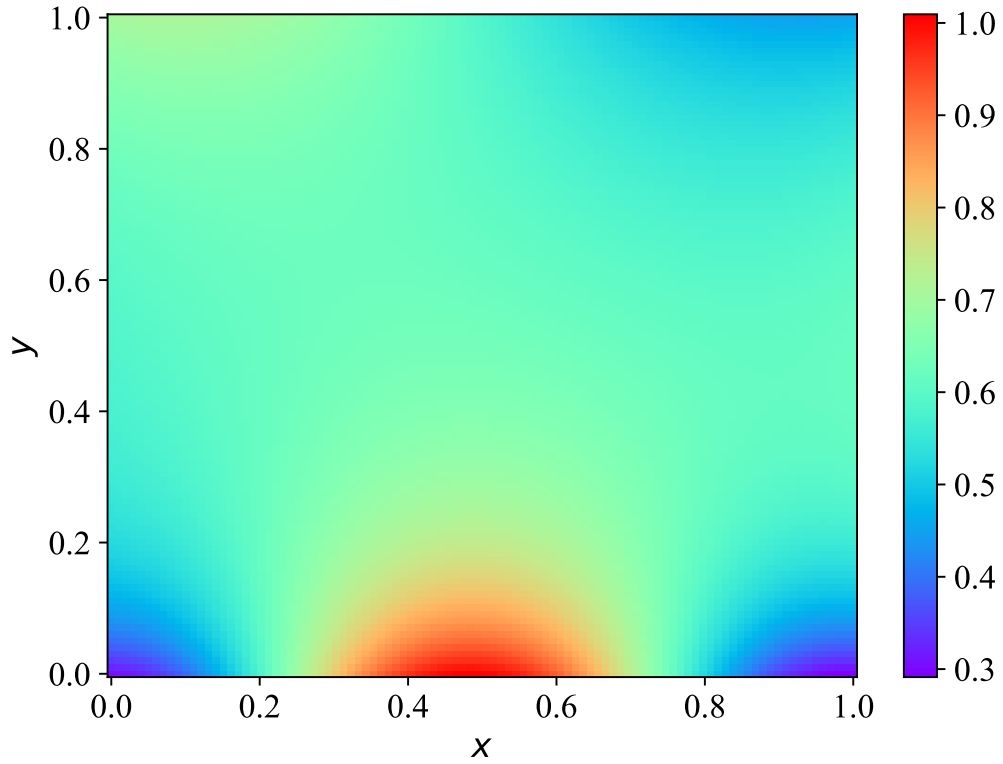


Figure 4.20: Output solution to Laplace's equation with the control policy fixed as a boundary condition.

The output plot looks relatively similar to the previous output plot, with the biggest differences around the boundary. However, by looking at the individual losses, the cost of the objective function is now significantly worse, which is an indication that the learned control policy would not work very well on a true system. The same problem remains with the fact that the learned control policy is not periodic, which can be why the physics loss is much lower than the boundary loss. But it is likely that it is not possible to find a solution that perfectly satisfies both the boundary conditions and the desired flux. A possible solution to the non-periodic control policy could be to satisfy the periodic boundary with a Fourier embedding in the y-direction, which could be worth investigating further.

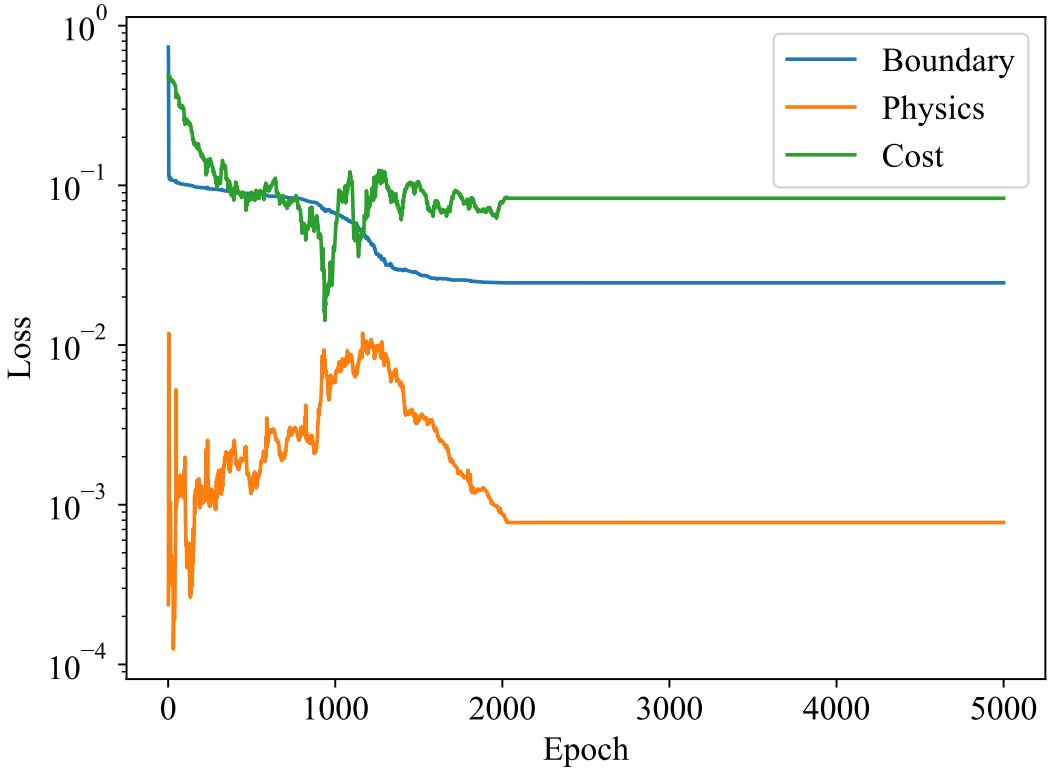


Figure 4.21: Training losses when training on Laplace's equation with the control policy fixed as a boundary condition. The cost is not used when training, and is instead used as a performance metric.

Dirichlet Boundary Control

Training a PINN on the output solution of the heat equation results in the plot shown below in Figure 4.22, which looks relatively similar to the previous 1-dimensional heat equation plots. As the desired temperature distribution is a constant 0.5, the system will attempt to move towards this distribution while also subject to the constraint of the dynamics. Figure 4.23 shows the same plot viewed at some slices in time.

As seen from the time slices, the temperature distribution becomes flatter over time, but the heat equation is not really capable of staying constant over a bounded domain. It is possible that the temperature distribution would oscillate at the boundaries and remain close to 0.5 in the center, which overall averages out to 0.5 over a longer time horizon.

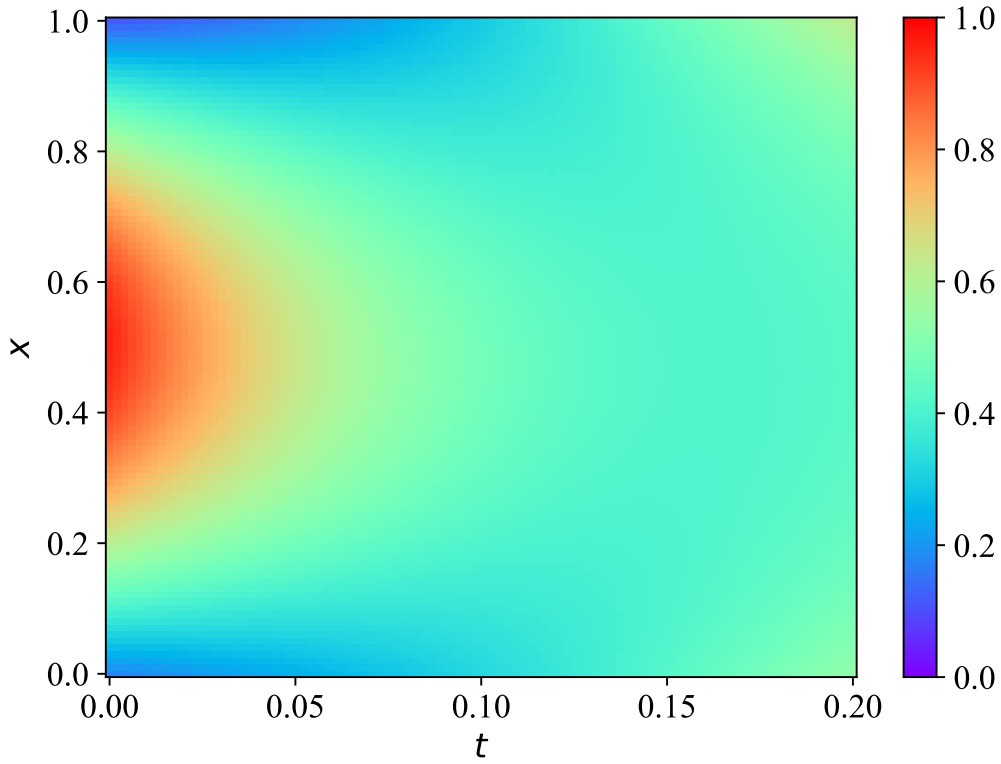
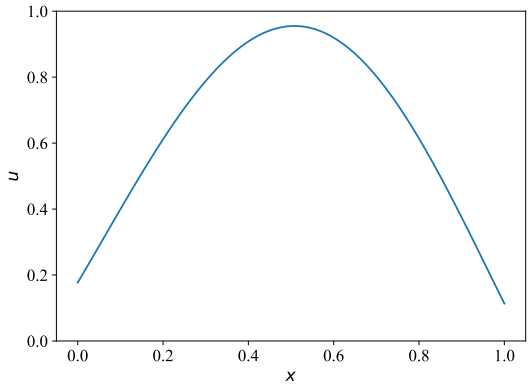
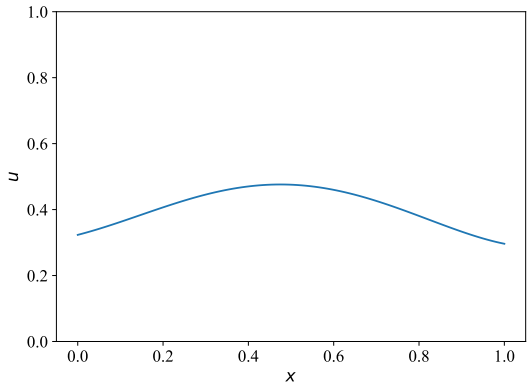


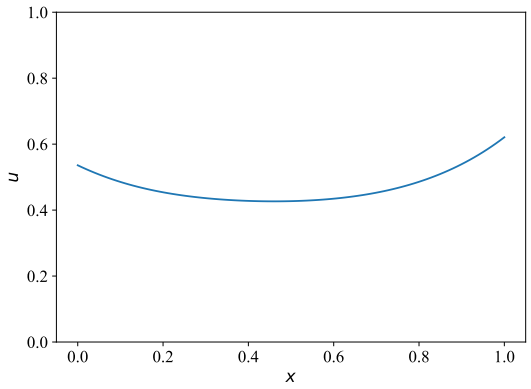
Figure 4.22: PINN output of the solution to the optimal control problem on the heat equation with Dirichlet boundary control.



(a) $t = 0$



(b) $t = 0.1$



(c) $t = 0.2$

Figure 4.23: Visualization of time-slices from the PINN output of the solution to the optimal control problem on the heat equation with Dirichlet boundary control.

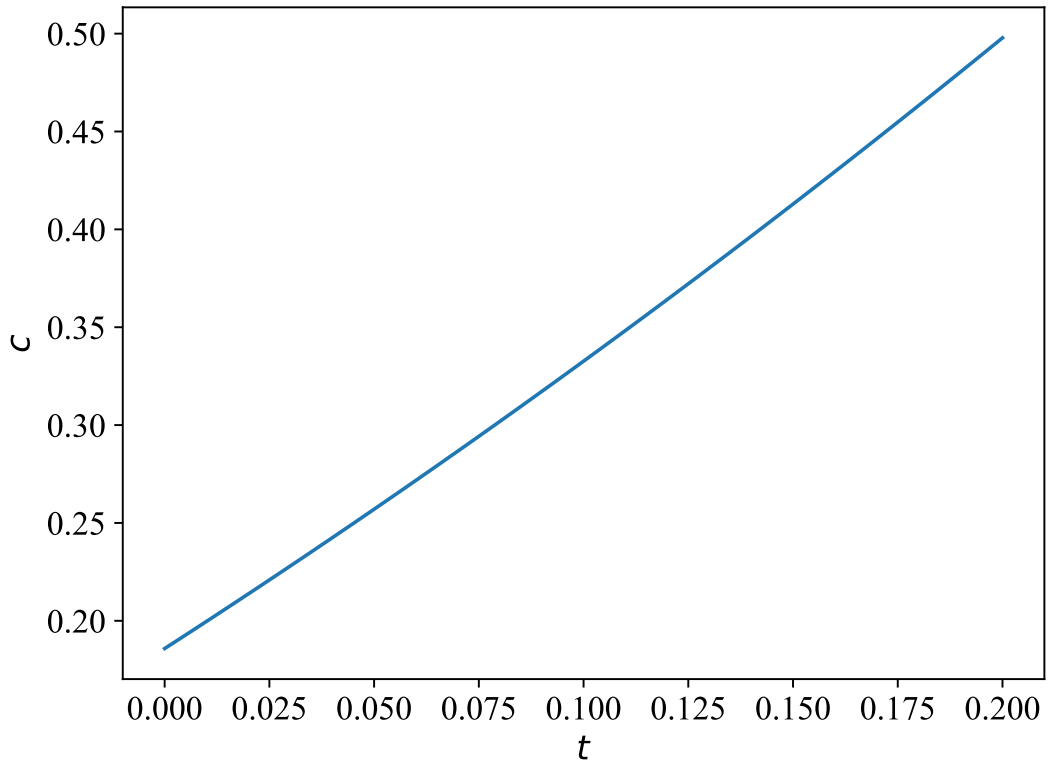


Figure 4.24: Learned control policy to the optimal control problem on the heat equation with Dirichlet boundary control.

The learned control policy is shown in Figure 4.24, which also serves as a boundary condition along the bottom side of the domain. It appears as a linearly increasing function, which can make sense by considering that there is an initial heat distribution that is spread out over the domain over time. As the distribution becomes more spread out, there is also less incoming heat to the edges of the domain, which is compensated for by the control policy. This can also be justified by thinking of the heat equation as containing a first order partial derivative of time, which when integrated over time becomes a straight line.

As there is no boundary condition on the top, the trained PINN appears to impose a symmetric boundary condition on itself. This can be thought of as equivalent to applying the control boundary on both sides of the domain. For real systems governed by the heat equation on a bounded domain, there will always be some boundary conditions on both sides of the domain. This also means that real systems will typically get a more asymmetric distribution when supplied with heat from only one of the boundaries.

Neumann Boundary Control

Replacing the Dirichlet boundary with the Neumann boundary and training a PINN on the optimal control problem results in the output solution in Figure 4.25 along with time-slices in Figure 4.26. The desired temperature distribution is still set to be a constant 0.5.

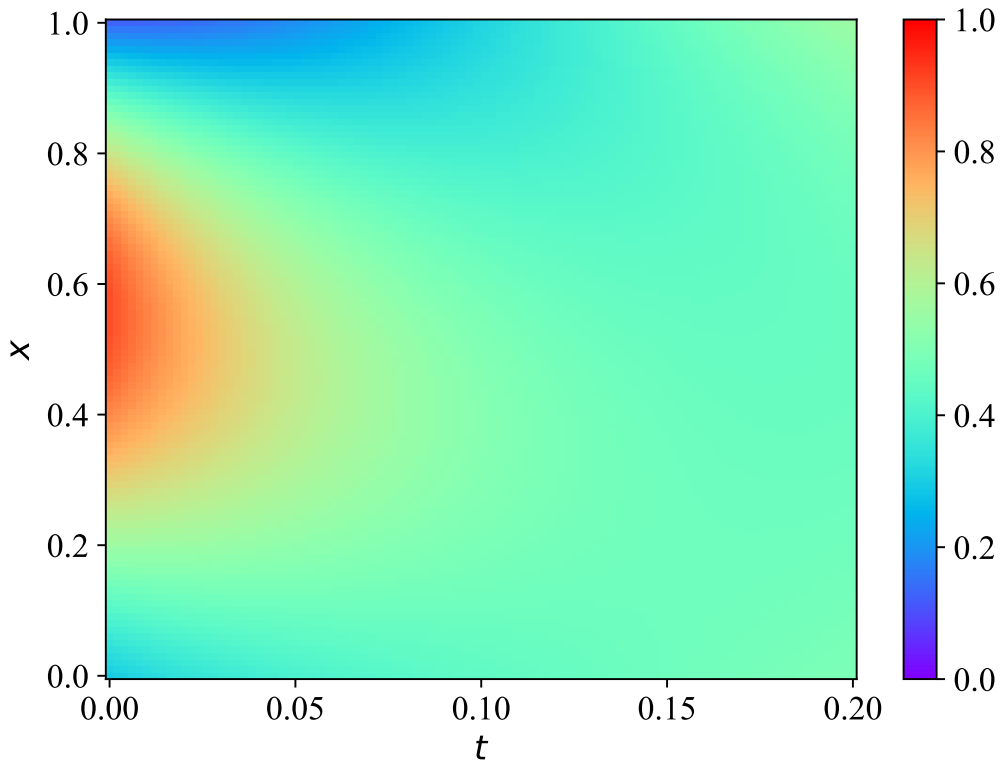
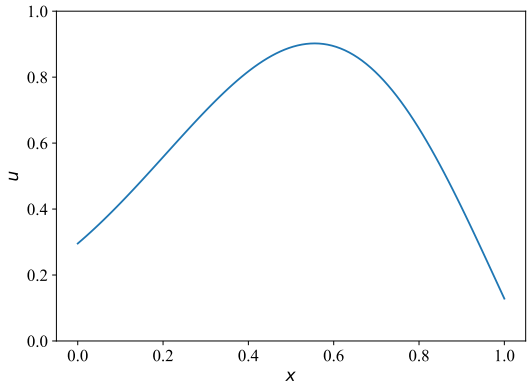
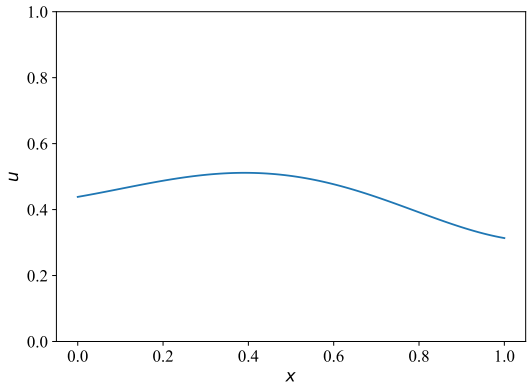


Figure 4.25: PINN output of the solution to the optimal control problem on the heat equation with Neumann boundary control.

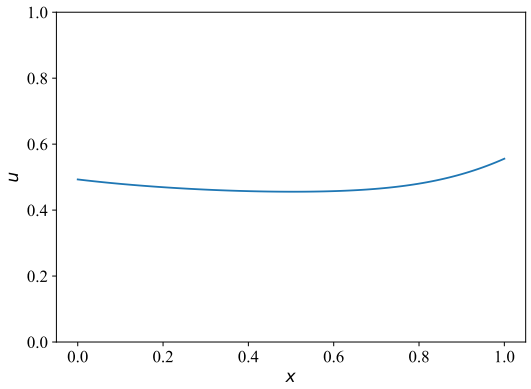
One noticeable difference from the previous experiment is that the overall temperature distribution is much less symmetric. This also seems like a more realistic result, and might be related to how Neumann control can be interpreted as more realistic than Dirichlet control. Now, heat is sent in with some inertia, as opposed to instantly setting the heat on the boundary. Otherwise, the same heat distribution appears to flatten out relatively well, and the same argument related to averaging out over a longer time span is still valid.



(a) $t = 0$



(b) $t = 0.1$



(c) $t = 0.2$

Figure 4.26: Visualization of time-slices from the PINN output of the solution to the optimal control problem on the heat equation with Neumann boundary control.

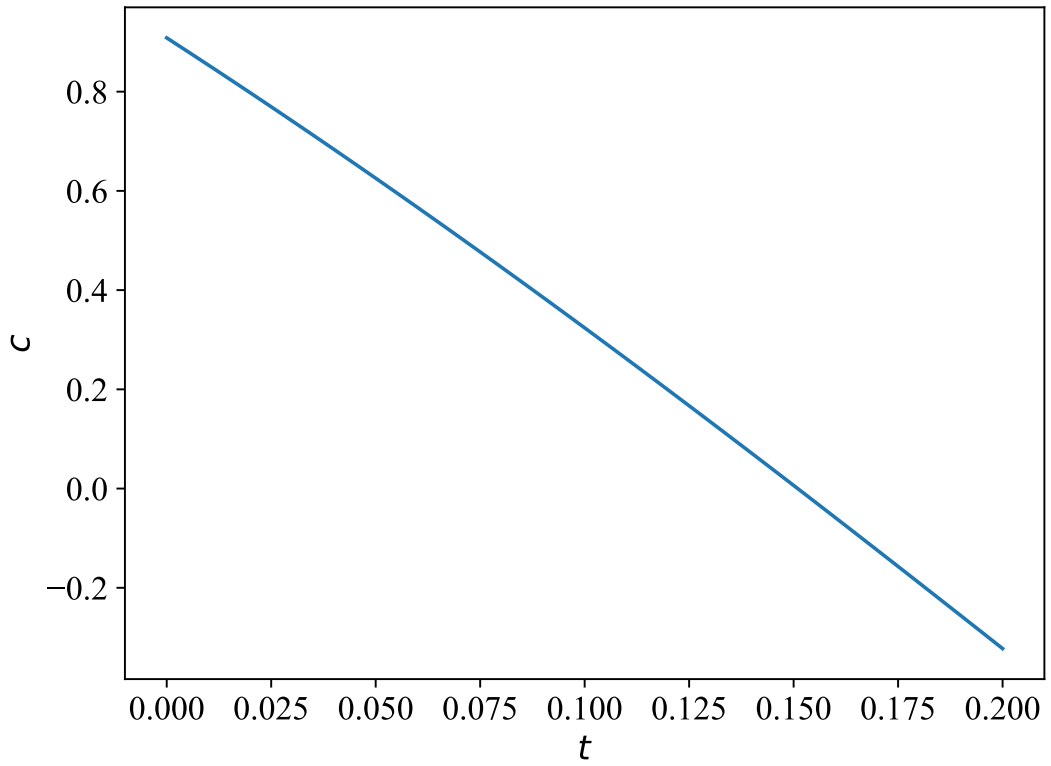


Figure 4.27: Learned control policy to the optimal control problem on the heat equation with Neumann boundary control.

The learned control policy in Figure 4.27 still appears as a straight line, although this time with a downward slope. As the value of $c(t)$ refers to the rate of change in the x-direction on the boundary, this is related to how much heat to introduce to the system. At the beginning the boundary is close to zero which means a high rate of heat must be introduced. After a while, as the heat distributes evenly, it is less necessary to add more heat. Eventually due to inertia, the heat on the boundary exceeds the desired heat of 0.5, so the rate of change becomes negative to lower it.

This overshoot might lead to some oscillations which could resemble a step response for an underdamped second order linear system, where it eventually approaches a stationary value. A reason for this could be because the objective function is set to target a heat distribution of a constant 0.5 without considering the inertia in the heat distribution itself, causing it to overshoot on the boundaries. For the step response mentioned previously, the reference can be set to a constant value for the state, and a value of zero to the derivative of the state to account for this.

To test this hypothesis further, the same optimal control problem was solved again with a longer time horizon. The final time now goes from 0.2 to 1 second. The number of data points along the initial condition was also increased from 100 to 1000 to increase the general accuracy.

The resulting solution from this is plotted in Figure 4.28 along with time slices in Figure 4.29. The overall solution is now much more symmetric again, which could indicate that the previous lack of symmetry was more because of too few data points along the initial condition, thus making it asymmetric from the start. Because there still isn't any boundary condition on the top, it might be that the PINN imposes its own

symmetry again.

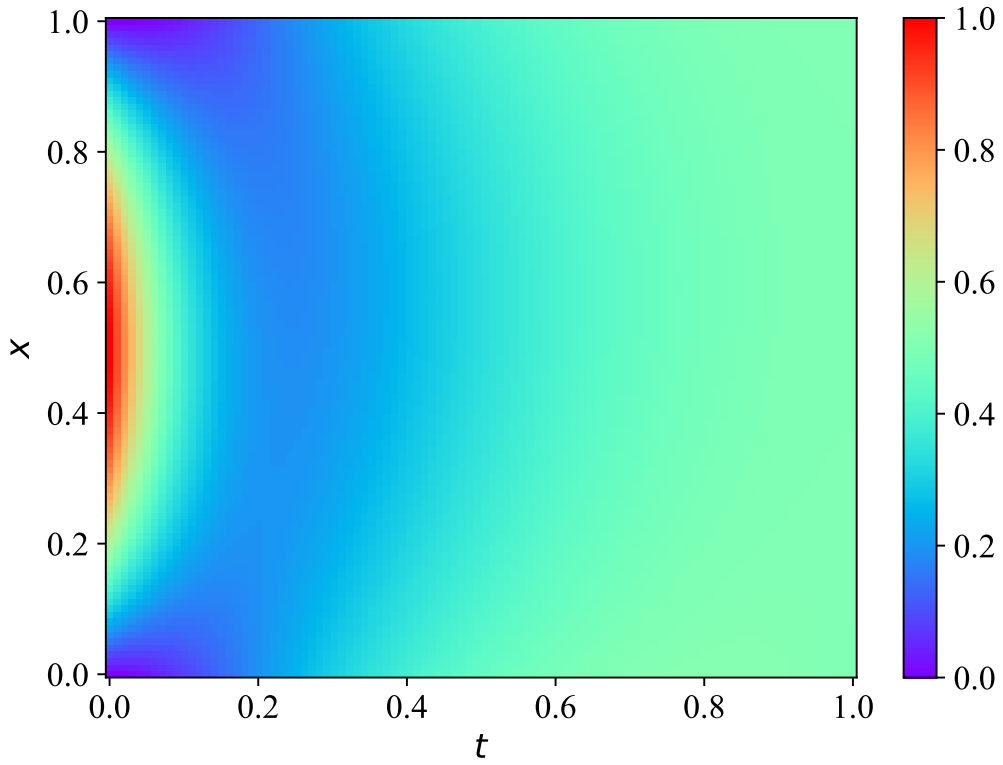
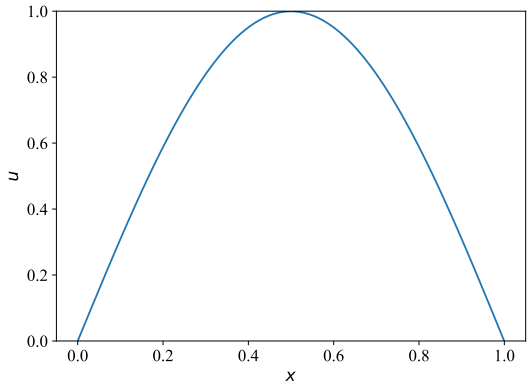
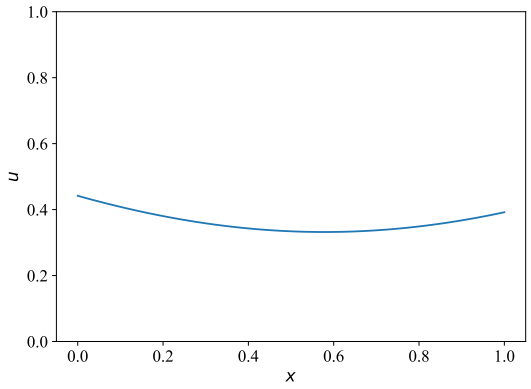


Figure 4.28: PINN output of the solution to the optimal control problem on the heat equation with Neumann boundary control and a longer time horizon.

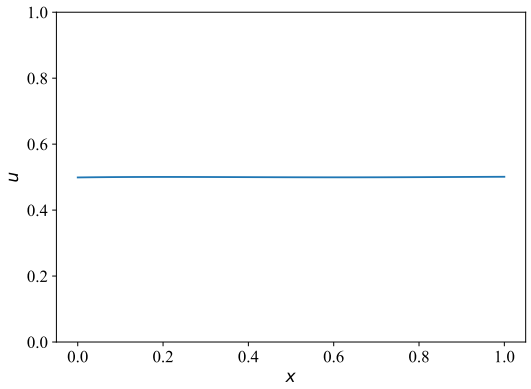
It is also interesting to note that the new plot in Figure 4.28 appears to almost vanish completely in an arc a little after 0.2 seconds. As the desired temperature distribution is set to be at $t = 1$, it does not have to resemble the plot from the previous run. It is then brought back by the control input that gradually shapes the distribution.



(a) $t = 0$



(b) $t = 0.1$



(c) $t = 0.2$

Figure 4.29: Visualization of time-slices from the PINN output of the solution to the optimal control problem on the heat equation with Neumann boundary control and a longer time horizon.

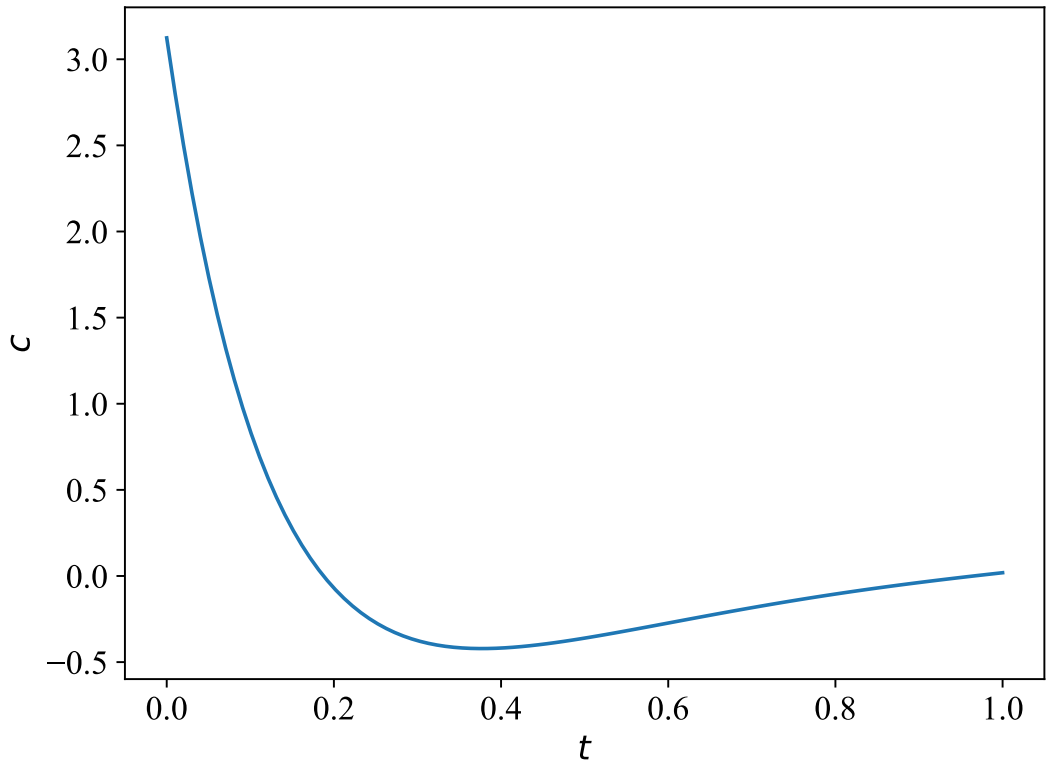


Figure 4.30: Learned control policy to the optimal control problem on the heat equation with Neumann boundary control and a longer time horizon.

The control input visualized in Figure 4.30 looks similar to the previous input in the sense that it looks like a relatively straight line at the start. It also overshoots zero, before slowly converging back to it from below, which is very reminiscent of a second order step response that is slightly underdamped. At the end with a time derivative of zero, it means that the rate of change in the x direction is zero. Which shows that the heat distribution has converged to a constant value.

Initial Control

For the final optimal control experiment, the trained PINN output is visualized in Figure 4.31. Comparing the plot visually to the analytical plot, it is immediately clear that it is not a good representation of the true solution. Looking at specific slices in time shown in Figure 4.32, the differences become more obvious. The learned solution at the initial time $t = 0$ also corresponds to the learned control policy, and does match the analytical solution very well. However, the PINN appears to still converge to the desired analytical solution at $t = 5$ to a high degree, which also corresponds to satisfying the objective function well.

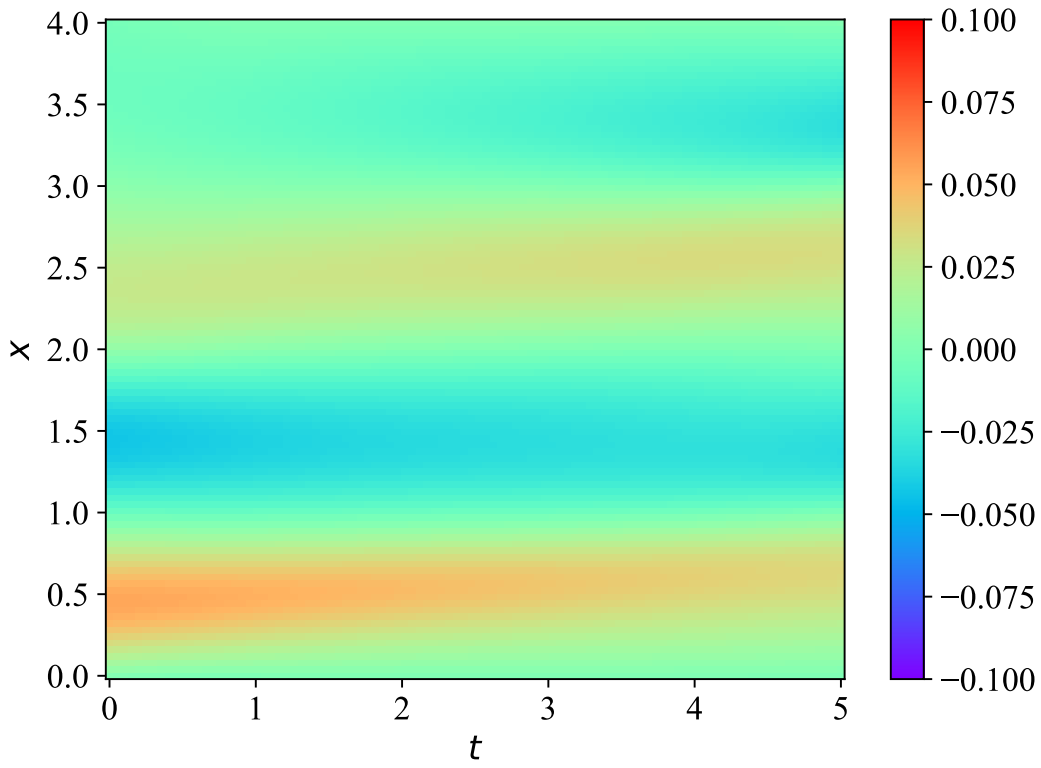
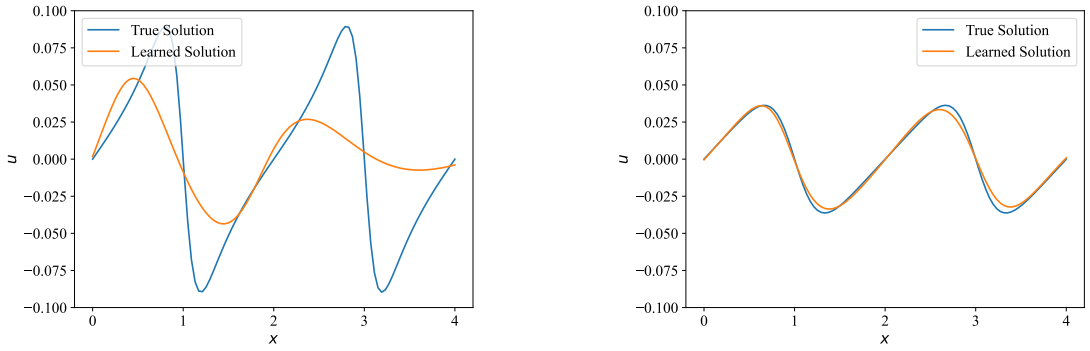


Figure 4.31: PINN output of the solution to the optimal control problem with Burgers' equation.



(a) $t = 0$ (b) $t = 5$

Figure 4.32: PINN output at specific slices in time of the solution to the optimal control problem with Burgers' equation.

Looking at the individual training losses in Figure 4.33, it can be seen that the boundary loss is the lowest, which means that the network is not obstructed in the same way as previous experiments. The initial loss comes next, and refers to the difference between the control policy and initial value. So even with a low initial loss, the large discrepancy comes because the control policy itself is inaccurate. The cost of the objective function comes next, leaving the physics informed loss as the worst performing. So this can indicate that even though the learned control policy worked well to lower the objective

function here, it would not work well when applied to a real system, as the physics cheat to get there.

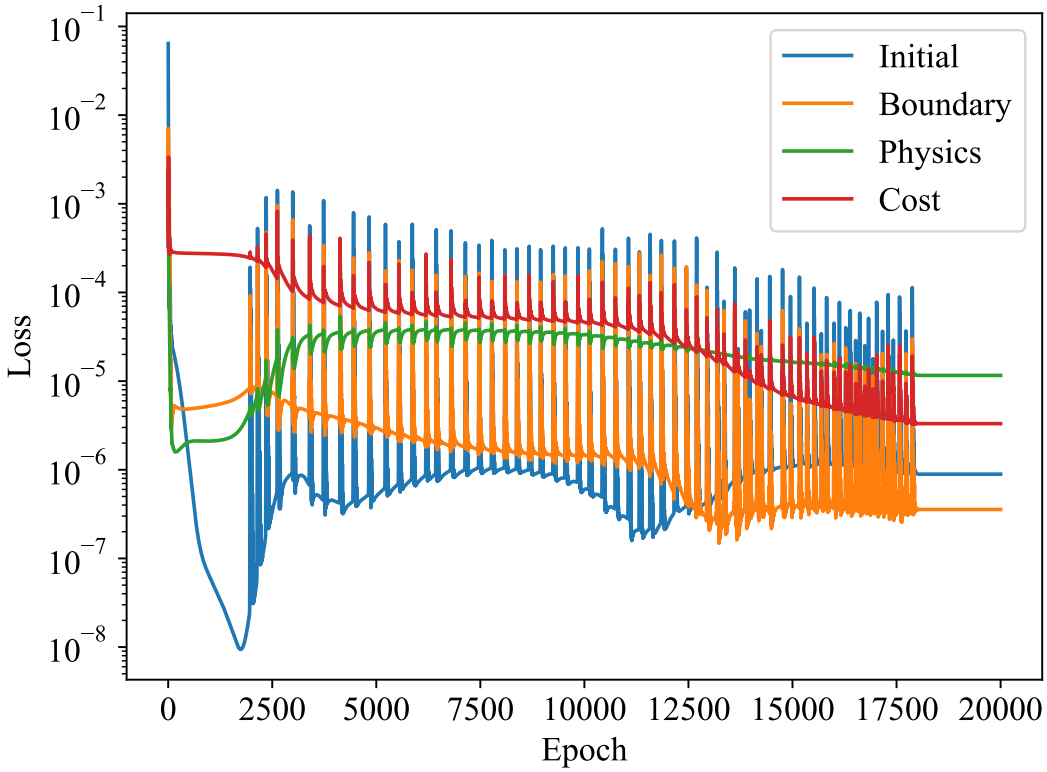


Figure 4.33: Training losses when training for the optimal control problem with Burgers' equation.

The plots in Figure 4.33 are generally noisy with many frequent spikes. The spikes happen at the same times when the Adam optimizer is stuck in a local minima but is carried outside with momentum. Then the physics loss increases somewhat, while the other three increase. This essentially means that it is a difficult problem to solve with this simple formulation. The plots also appear to converge at the end. This happens because of a change to the L-BFGS optimizer at 18000 epochs. But it also looks like the L-BFGS optimizer simply converges to the same local minima the models already were bouncing around in with Adam, and might not be necessary to use.

4.1.6 Regularization with the Maximum Principle

Elliptic PDE

Regularizing the training by using the maximum principle for Laplace's equation results in the PINN output in Figure 4.34, which overall looks very similar to the analytical solution in Figure 3.5. Calculating the MSE from the analytical solution results in $1.21 \cdot 10^{-4}$. For comparison, training another PINN from scratch with the exact same setup except the regularization results in an MSE from the analytical solution of $1.91 \cdot 10^{-4}$. So the regularization yields a very small improvement.

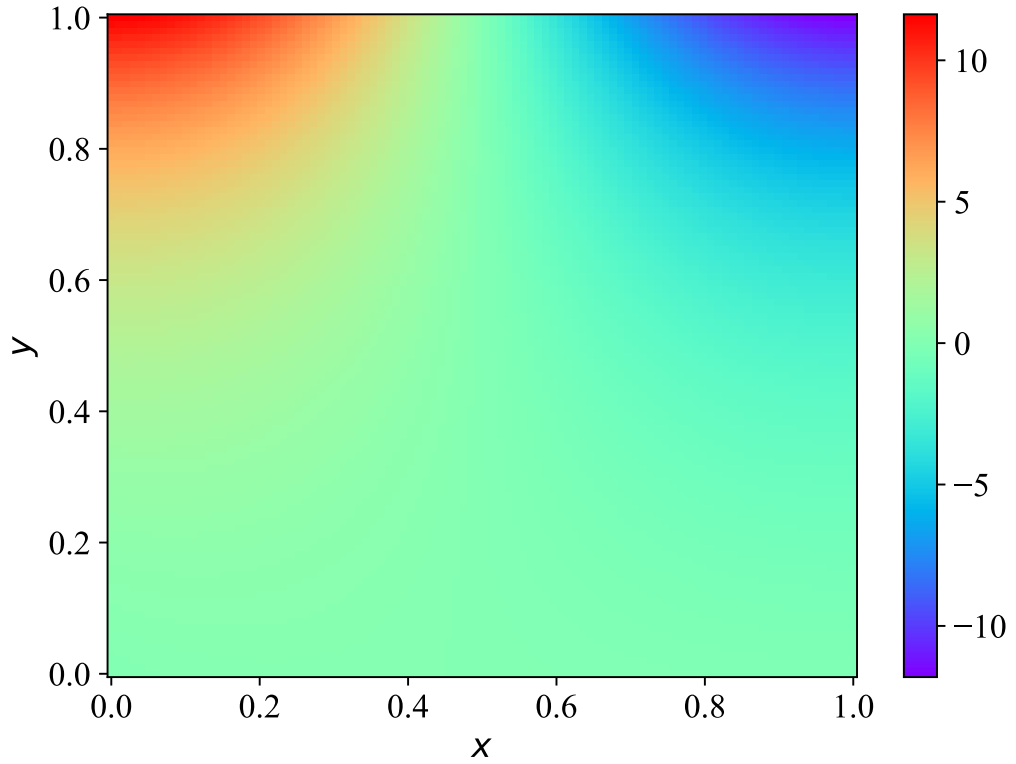


Figure 4.34: PINN output after training with maximum principle regularization on Laplace's equation.

Plotting this MSE value as a validation loss for each epoch when training results in the plots in Figure 4.35. The regularized PINN appears to learn faster during the first 200 epochs before flattening out. It then eventually surpasses the standard PINN again after around 700 epochs.

It is possible that the regularization is something that works best at the start of training, and can then be gradually or completely removed from the loss function. But it does seem to give slightly faster training along with a little better final performance / accuracy. And as the computational cost is insignificant in comparison to the rest of the training, adding this regularization term when training on elliptic PDEs does not seem to have any obvious disadvantages. However, while the performance gains are very minor for this specific experiment, it could be more useful for elliptic PDEs in higher dimensions, and might be interesting to investigate further.

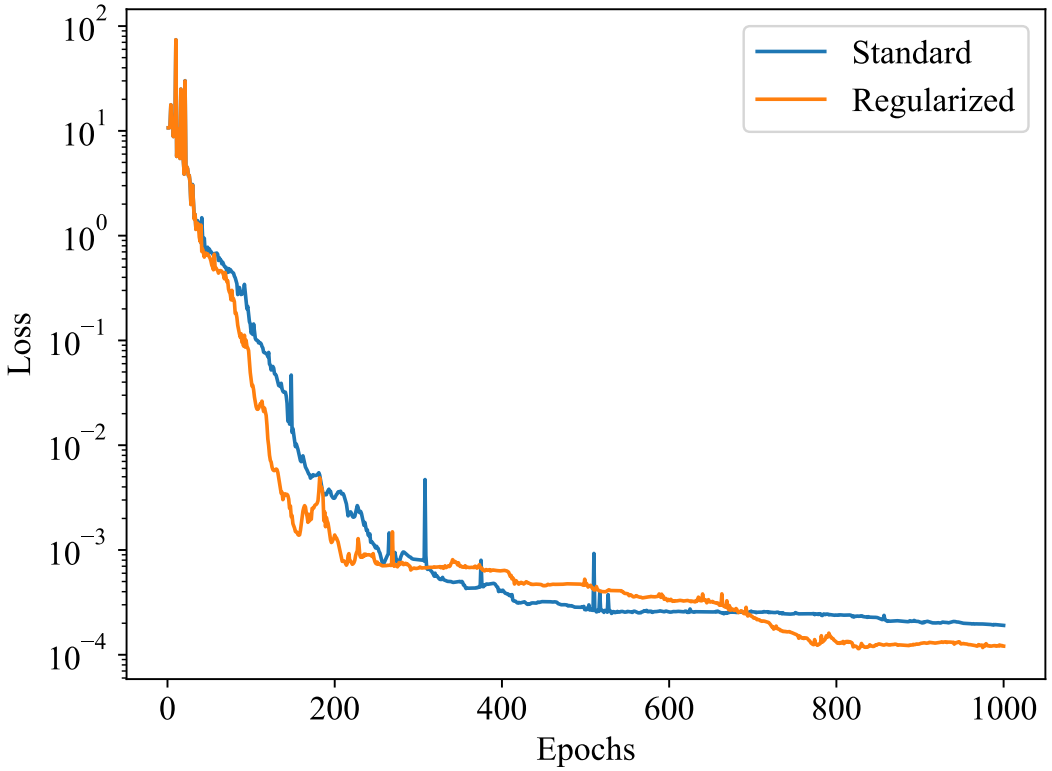


Figure 4.35: Validation losses after training with maximum principle regularization on Laplace's equation.

4.1.7 Causal Optimal Control

Initial Control

Training a PINN with causal training and the other mentioned improvements results in the plots shown below in Figure 4.36 and Figure 4.37.

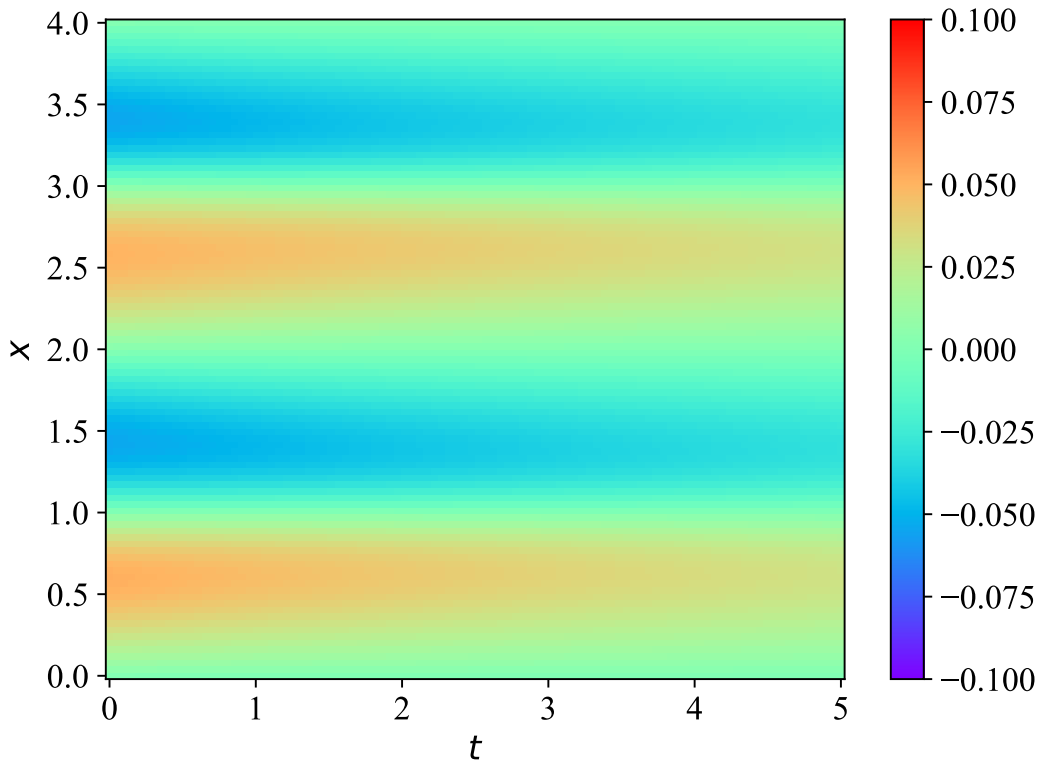
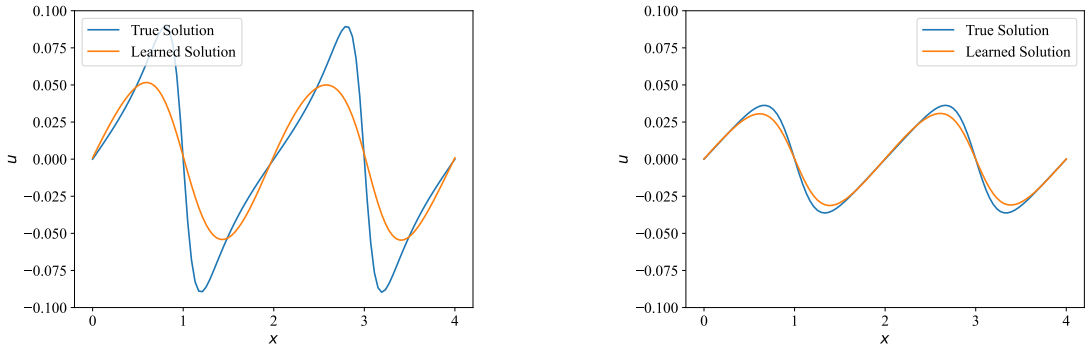


Figure 4.36: PINN output of the solution to the optimal control problem with Burgers' equation after training with causality and other improvements.



(a) $t = 0$ (b) $t = 5$

Figure 4.37: PINN output at specific slices in time of the solution to the optimal control problem with Burgers' equation after training with causality and other improvements.

Comparing these results visually to the previous attempt in Figure 4.31 and Figure 4.32 it is overall a better result, which is easier to see when $t = 0$. It is also possible that further training would give even better results, as the causality makes the overall training take much longer for a problem like this. Another consideration is that the relative weighting of the physics loss is set to a really large value, which is necessary to not be overrun by the causal loss and initial loss.

However, the initial condition is still not very good when compared with the analytical solution. It is possible that the causality makes the problem too difficult to learn. Running the same experiment again with the same setup with all the improved techniques except for the causal loss results in the Figures shown below in Figure 4.38 and Figure 4.39.

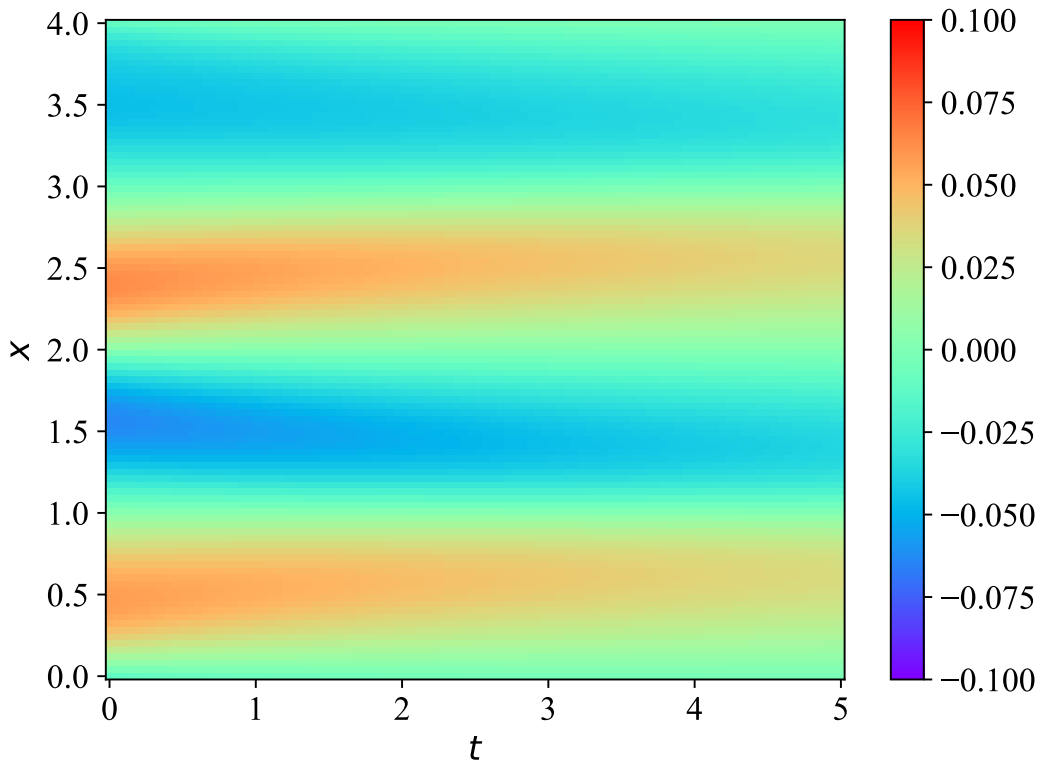
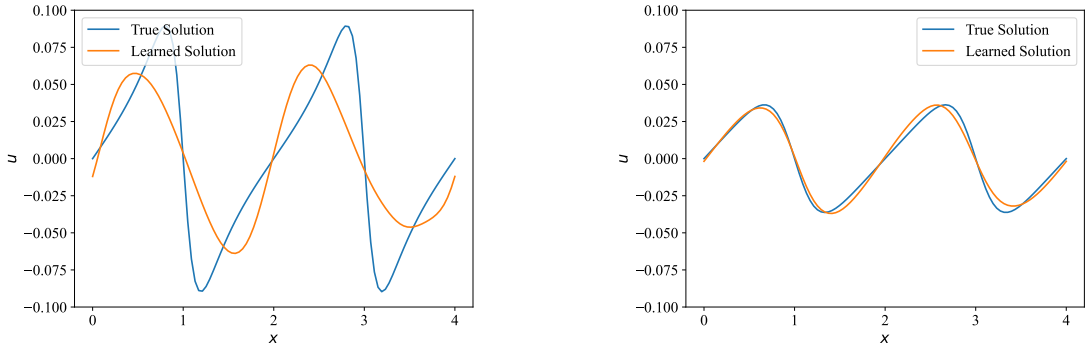


Figure 4.38: PINN output of the solution to the optimal control problem with Burgers' equation after training with many improvements, but not causality.



(a) $t = 0$ (b) $t = 5$

Figure 4.39: PINN output at specific slices in time of the solution to the optimal control problem with Burgers' equation after training with causality and other improvements.

The solution looks relatively similar to the previous attempt that used causality. The initial condition is the biggest change, where the amplitude is closer to the true solution but some of the skewness is lost. It is hard to say which attempt resulted in the best solution, and that might depend on what the application of the problem would be.

Reversed Initial Control

Using the time-reversed PDE and training a PINN on these dynamics while also using causality and the other training techniques described before results in Figure 4.40 and Figure 4.41.

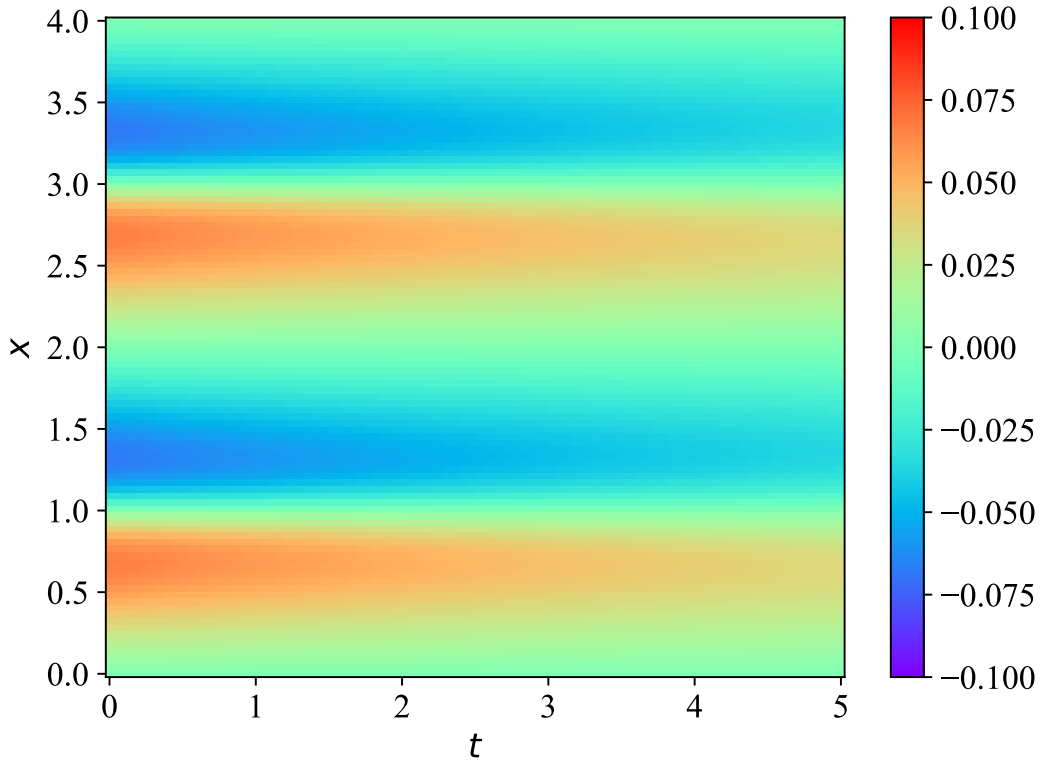
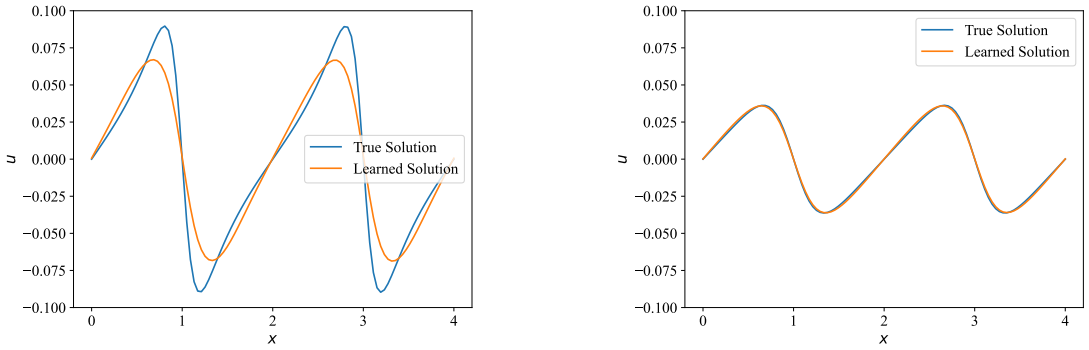


Figure 4.40: PINN output of the solution to the optimal control problem with Burgers' equation after training on the time-reversed system.



(a) $t = 0$ (b) $t = 5$

Figure 4.41: PINN output at specific slices in time of the solution to the optimal control problem with Burgers' equation after training on the time-reversed system.

These final results from training on the time-reversed dynamics can be seen to improve upon the previous attempts by looking at the initial learned solution. While still not perfect, it is a noticeable improvement from the ones in Figure 4.36 and Figure 4.38. This plot was here generated by sampling at the final time for the trained model to get the initial distribution to use. Using this problem formulation, training with causality also makes more sense as it is working in the same way as the dependence between final state and initial condition.

This experiment shows the importance of adapting the method to the problem. Not all problems become easier by time-reversing the dynamics, but in cases like this where it is possible it resulted in a solution that was easier to learn.

4.2 Discussion

Modeling dynamical systems with standard neural networks is often difficult due to a lack of training data, and the trained networks struggle with generalization outside the training domain. Adding prior physics information about the dynamics, which can often be obtained by modeling systems from first principles, can regularize the neural networks in a way that both allows them to learn from much less data and also generalize much better outside the domain of the training data.

A non-intuitive result was that complicated dynamics turned out to be easier to learn compared to more difficult dynamics, the exact opposite of what usually happens when working with dynamical systems. As complicated dynamical systems are more strict in how the system evolves it is possible that they also contain more information when training PINNs, which allows for easier training. In general, it appears that PDEs are easier to learn than ODEs, time-varying is easier than time-invariant, and nonlinear is easier than linear.

How the training is done can also greatly influence the final result. Using the correct optimization algorithm turned out to be important for some problems. Having enough collocation points is necessary for better generalization, and is also necessary for increasing the number of training steps. The collocation points should also be placed at the correct locations. A linearly spaced grid makes learning very difficult, while a random uniform sampling often works well enough.

Discovering unknown system parameters is one of the main advantages of PINNs compared to traditional numerical methods, and seems to work well for simple problems.

This does however require significantly more training data, and is also much more computationally expensive. Learning terms symbolically has the problem where the input variables must be explicitly set. This could mean that to find the best overall solution it requires that the PINN is trained from scratch from every possible input combination if nothing prior is known. And without access to an analytical solution or data it is not possible to verify the solution, which is always the case when using the method for new problems.

Using PINNs for optimal control problems are possible to setup, and the framework has shown to be very flexible and easy to setup for solving many different types of problems, requiring a relatively low complexity for the implementation. The results are however very difficult to accurately verify, and the method does not have any convergence guarantees. This stands in contrast to traditional numerical optimization methods, where a convex optimization problem has algorithms that provably converge to the global optimum. Of course not all problems are convex, but many of practical interest are possible to formulate like this.

5 Conclusion and Further Work

5.1 Conclusion

Physics informed neural networks seem to work well for learning output trajectories from both ODEs and PDEs, provided that there is enough data and enough computational effort spent during the training. Data-driven discovery of unknown parameters also seemed to work well, as long as enough data was provided. PINNs are overall a natural choice for incorporating prior knowledge about system dynamics into a machine learning framework. Discovering unknown terms of an equation from data is also another problem where the PINN approach can work relatively well.

Applying PINNs for more advanced problems can also work well, especially by also incorporating the various enhancements to the training process. This can also lead to an increased computational complexity, but can give much better results. This might not necessarily be worth using for simply solving PDEs numerically, as traditional numerical methods will outperform this both computationally and accurately. However, adding causality to solving optimal control problems gave much better results than not using it. Some of the other improvements like the modified network structure doesn't seem to have any disadvantages when used, and can always be applied. Fourier embeddings are very useful when the problem has a periodic boundary, not so useful if not.

5.2 Future Work

The current method of validating the trained PINNs is not very robust. A better way than visually comparing can be to implement numerical PDE solvers to compare against. An alternative is to compare against a dataset sampled from either a real system or another numerical solver. The current data being used should also be augmented with some random noise to better simulate sensors and make the overall experiments more realistic. However, when learning symbolic representations for unknown systems or new control policies it can be much harder to verify the results even numerically.

An alternative PINN framework based on discretizing the dynamics in a similar way to the finite difference numerical PDE method was also described by the original authors [33]. The actual discretization is based on Runge-Kutta numerical methods for ODEs, which are much more accurate compared to a simple finite difference. The purpose of this framework was to avoid the curse of dimensionality by reducing the number of collocation points, which allows for faster training. Discretized PINN models could be worth exploring further.

Using PINNs for data-driven discovery works well for systems where the expression for the dynamics is known, but with unknown parameter values. Modeling real systems from first principles relies on making assumptions, which can lead to model inaccuracies. This is also true when using this symbolic approach as the input variables must be set explicitly, either based on prior information or a guess. In the ideal case, redundant input variables would not be used by the trained network, but neural networks can be unpredictable and possibly overfit on these variables in a way that is not physically

accurate but still causes the training loss to decrease. An interesting experiment can be to treat the set of input variables as a hyperparameter and then perform hyperparameter optimization on this set. By choosing from this optimization, there is a higher chance of finding the most accurate true expression.

Using PINNs for optimal control has shown to be a very flexible framework for different types of control problems. This master thesis has shown experiments on some types, but many other problems can be formulated and experimented on with PINNs. Another thing that should be done is verify the learned control policies by solving the same problem numerically using the learned and freezed control policy. Validating this on some simpler problems gives some evidence that the method works, which can then be extrapolated to more complicated problems.

Some more future work could be to combine these two methods in order to do optimal control of systems with partially unknown dynamics. It could be possible to either do it in two stages, where first the unknown term is learned symbolically based on some dataset of the true system. This symbolic term can then be added to the optimal control PINN when learning the control policy. It might also be possible to do both of these in a single step by adding the symbolic network directly to the optimal control PINN problem, but this will most likely be too difficult to learn with neural networks directly and still learn something accurate. Using PINNs to learn unknown terms could also be combined with model predictive control in the way described by [52]. All of these problem setups can be improved by adding causality and other training improvements.

Bibliography

[1] Arthur L. Samuel. Some studies in machine learning using the game of checkers. *IBM Journal of Research and Development*, 3(3):210–226, 1959.

[2] Frank Rosenblatt. The perceptron - a perceiving and recognizing automaton. *Cornell Aeronautical Laboratory*, (85-460-1), 1957.

[3] W.S. McCulloch and W. Pitts. A logical calculus of the ideas immanent in nervous activity. *Bulletin of Mathematical Biophysics*, 5(4):115–133, 1943.

[4] Marvin Minsky and Seymour A. Papert. *Perceptrons: An Introduction to Computational Geometry*. MIT Press, Cambridge, MA, USA, 1969.

[5] D. Rumelhart, G. Hinton, and R. Williams. Learning representations by back-propagating errors. *Nature*, 323:533–536, 1986.

[6] T. Mitchell. *Machine Learning*. McGraw-Hil, 1997.

[7] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. *CoRR*, abs/2005.14165, 2020.

[8] Robin Rombach, Andreas Blattmann, Dominik Lorenz, Patrick Esser, and Björn Ommer. High-resolution image synthesis with latent diffusion models. *CoRR*, abs/2112.10752, 2021.

[9] R.S. Sutton, A.G. Barto, and R.J. Williams. Reinforcement learning is direct adaptive optimal control. *IEEE Control Systems Magazine*, 12(2):19–22, 1992.

[10] Gerald Tesauro. Temporal difference learning and td-gammon. *Commun. ACM*, 38(3):58–68, mar 1995.

[11] Julian Schrittwieser, Ioannis Antonoglou, Thomas Hubert, Karen Simonyan, Laurent Sifre, Simon Schmitt, Arthur Guez, Edward Lockhart, Demis Hassabis, Thore Graepel, Timothy P. Lillicrap, and David Silver. Mastering atari, go, chess and shogi by planning with a learned model. *CoRR*, abs/1911.08265, 2019.

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

[13] Kurt Hornik, Maxwell B. Stinchcombe, and Halbert L. White. Multilayer feedforward networks are universal approximators. *Neural Networks*, 2:359–366, 1989.

[14] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2017.

[15] Jorge Nocedal and Stephen J. Wright. *Numerical Optimization*. Springer, 2006.

[16] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521(7553):436–444, 2015.

[17] Rafael C. Gonzalez and Richard E. Woods. *Digital Image Processing*. Pearson Prentice Hall, 2008.

[18] Y. Bengio, P. Simard, and P. Frasconi. Learning long-term dependencies with gradient descent is difficult. *IEEE Transactions on Neural Networks*, 5(2):157–166, 1994.

[19] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. Sequence to sequence learning with neural networks. *CoRR*, abs/1409.3215, 2014.

[20] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. *CoRR*, abs/1706.03762, 2017.

[21] David G. Schaeffer and John W. Cain. *Ordinary Differential Equations: Basics and Beyond*. Springer, 2016.

[22] E. Kreyszig. *Introductory Functional Analysis with Applications*. 1978.

[23] Lawrence C. Evans. *Partial Differential Equations*. American Mathematical Society, 2010.

[24] Peter J. Olver. *Introduction to Partial Differential Equations*. Springer, 2014.

[25] Erwin Kreyszig. *Advanced Engineering Mathematics*. Wiley, 2011.

[26] David Borthwick. *Introduction to Partial Differential Equations*. Springer, 2016.

[27] Ricky T. Q. Chen, Yulia Rubanova, Jesse Bettencourt, and David K Duvenaud. Neural ordinary differential equations. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 31. Curran Associates, Inc., 2018.

[28] Emilien Dupont, Arnaud Doucet, and Yee Whye Teh. Augmented neural odes, 2019.

[29] Stefano Massaroli, Michael Poli, Jinkyoo Park, Atsushi Yamashita, and Hajime Asama. Dissecting neural odes, 2021.

[30] Samuel Greydanus, Misko Dzamba, and Jason Yosinski. Hamiltonian neural networks. In *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019.

[31] Miles Cranmer, Sam Greydanus, Stephan Hoyer, Peter Battaglia, David Spergel, and Shirley Ho. Lagrangian neural networks, 2020.

[32] Alexander Norcliffe, Cristian Bodnar, Ben Day, Nikola Simidjievski, and Pietro Liò. On second order behaviour in augmented neural odes, 2020.

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

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

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

[36] Maziar Raissi, Alireza Yazdani, and George Em Karniadakis. Hidden fluid mechanics: A navier-stokes informed deep learning framework for assimilating flow visualization data. 2018.

[37] George Em Karniadakis, Ioannis G. Kevrekidis, Lu Lu, Paris Perdikaris, Sifan Wang, and Liu Yang. Physics-informed machine learning. *Nature*, 2021.

[38] Zhongkai Hao, Songming Liu, Yichi Zhang, Chengyang Ying, Yao Feng, Hang Su, and Jun Zhu. Physics-informed machine learning: A survey on problems, methods and applications, 2022.

[39] Yeonjong Shin. On the convergence of physics informed neural networks for linear second-order elliptic and parabolic type pdes. *Communications in Computational Physics*, 28(5):2042–2074, jun 2020.

[40] Siddhartha Mishra and Roberto Molinaro. Estimates on the generalization error of physics informed neural networks (pinns) for approximating a class of inverse problems for pdes, 2020.

[41] Aditi Krishnapriyan, Amir Gholami, Shandian Zhe, Robert Kirby, and Michael W Mahoney. Characterizing possible failure modes in physics-informed neural networks. In M. Ranzato, A. Beygelzimer, Y. Dauphin, P.S. Liang, and J. Wortman Vaughan, editors, *Advances in Neural Information Processing Systems*, volume 34, pages 26548–26560. Curran Associates, Inc., 2021.

[42] Stefano Markidis. The old and the new: Can physics-informed deep-learning replace traditional linear solvers? 2021.

[43] Sifan Wang, Xinling Yu, and Paris Perdikaris. When and why pinns fail to train: A neural tangent kernel perspective. *Journal of Computational Physics*, 449:110768, 2022.

[44] Junwoo Cho, Seungtae Nam, Hyunmo Yang, Seok-Bae Yun, Youngjoon Hong, and Eunbyung Park. Separable pinn: Mitigating the curse of dimensionality in physics-informed neural networks, 2022.

[45] Suchuan Dong and Naxian Ni. A method for representing periodic functions and enforcing exactly periodic boundary conditions with deep neural networks. *Journal of Computational Physics*, 435:110242, jun 2021.

[46] Sifan Wang, Yujun Teng, and Paris Perdikaris. Understanding and mitigating gradient pathologies in physics-informed neural networks, 2020.

[47] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778, 2016.

[48] Revanth Mattey and Susanta Ghosh. A novel sequential method to train physics informed neural networks for allen cahn and cahn hilliard equations. *Computer Methods in Applied Mechanics and Engineering*, 390:114474, 2022.

[49] Sifan Wang, Shyam Sankaran, and Paris Perdikaris. Respecting causality is all you need for training physics-informed neural networks, 2022.

[50] Ritam Majumdar, Vishal Jadhav, Anirudh Deodhar, Shirish Karande, Lovekesh Vig, and Venkataramana Runkana. Physics informed symbolic networks, 2022.

[51] Lena Podina, Brydon Eastman, and Mohammad Kohandel. A pinn approach to symbolic differential operator discovery with sparse data, 2022.

[52] Eric Aislan Antonelo, Eduardo Camponogara, Laio Oriel Seman, Eduardo Rehbein de Souza, Jean P. Jordanou, and Jomi F. Hubner. Physics-informed neural nets for control of dynamical systems, 2021.

[53] Saviz Mowlavi and Saleh Nabi. Optimal control of pdes using physics-informed neural networks. *Journal of Computational Physics*, 473:111731, 2023.

[54] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019.

[55] Richard Zou Horace He. functorch: Jax-like composable function transforms for pytorch. `https://github.com/pytorch/functorch`, 2021.

[56] Ricky T. Q. Chen. torchdiffeq, 2018.

[57] J. R. Dormand and P. J. Prince. A family of embedded runge-kutta formulae. *Journal of Computational and Applied Mathematics*, 6(1):19–26, 1980.

[58] Respecting causality is all you need for training physics-informed neural networks.

[59] Physics informed neural networks.

[60] L. S. Pontryagin, E. F. Mishchenko, V. G. Boltyanskii, and R. V. Gamkrelidze. *The mathematical theory of optimal processes*. 1962.

[61] Hassan K. Khalil. *Nonlinear Systems*. Pearson, 2014.

NTNU

Norwegian University of
Science and Technology