Master of Science in Engineering
MSc Thesis in Mathematical Modelling and Computation

Universal Physics-Informed Neural Networks for Hidden Discovery

Jonas Søeborg Nielsen

DTU Compute
Department of Applied Mathematics and Computer Science

Title
Universal Physics-Informed Neural Networks for Hidden Discovery

Author
Jonas Søeborg Nielsen
s183529

Supervisor
Allan Peter Engsig-Karup
Associate Professor
Department of Applied Mathematics and Computer Science
Technical University of Denmark

ETCS
30

Hand-in date
March 16th, 2025
Postponed from February 2nd, 2025

Jonas Søeborg Nielsen

$$f(x+\Delta x)=\sum_{i=0}^{\infty}\frac{(\Delta x)^i}{i!}f^{(i)}(x)$$

DTU Compute
Department of Applied Mathematics and Computer Science
Technical University of Denmark

Richard Petersens Plads
Building 324
2800 Kongens Lyngby, Denmark
compute@compute.dtu.dk
www.compute.dtu.dk

Abstract

Physics-Informed Neural Networks (PINNs), introduced in 2017, is a powerful technique for solving known ordinary and partial differential equations. However, in many real-world applications, the underlying dynamics are only partially known, which makes PINNs less applicable. Universal Physics-Informed Neural Networks (UPINNs) are a recent (2022) advancement that allow solving hidden discovery problems by combining PINNs with Universal Differential Equations (UDE). While UPINNs have shown promising results, the framework is still in its infancy and research is sparse. This thesis aims to contribute by exploring the capabilities and limitations of UPINNs in a variety of settings, including the Lotka-Volterra system, SIR model, Burgers' equation, and Navier-Stokes equations for incompressible fluid flow. Further, a real-world example of using PINNs to model plasma transport in a tokamak reactor is considered.

The results demonstrate that UPINNs are capable of modelling hidden dynamics in partially known systems, often yielding solutions comparable to regular PINNs applied to the full system, even when data is sparse and/or noisy. Furthermore, the approximated hidden dynamics are accurate enough for symbolic regression frameworks like SINDy and PDE-find to recover the hidden equations.

The study also highlights a tendency for UPINNs to estimate known dynamics that are insufficiently constrained, or to introduce new "phantom" dynamics that does not exist in the true system but do not alter the solution either. Issues that were evident in inverse problems and systems with several governing equations. Moreover, while shown to be feasible, disregarding inductive biases can lead to less robust models and hinder extrapolation capabilities.

`Python` implementations written and used in this project can be accessed in the below linked `GitHub` repository. The repository also contains the model files for all results presented in this thesis.

GitHub Repository

https://github.com/MEGAjosni/Master-Thesis.git

Acknowledgements

I would like to thank my cat Newton for moral support and keeping me company during the long hours of writing this thesis.

Thanks to my supervisor, Allan Peter Engsig-Karup (DTU Compute), for suggesting the project and providing guidance throughout the process.

Further thanks to Volker Naulin (DTU Physics), Antti Salmi (VTT) and Tuomas Tala (VTT) for providing data and guidance on the plasma physics problem.

Finally, I would like to thank my friends and family for support during the process and help with proofreading in the final stages.

Abbreviations

Abbreviation	Description
ANN	Artificial Neural Network
BC	Boundary Condition
BVP	Boundary Value Problem
CAS	Computer Algebra System
CNN	Convolutional Neural Network
DE	Differential Equation
FDM	Finite Difference Method
FEM	Finite Element Method
FNN	Feedforward Neural Network
GPU	Graphics Processing Unit
IVP	Initial Value Problem
KAN	Kolmogorov-Arnold Network
KAT	Kolmogorov-Arnold (Representation) Theorem
LHS	Latin Hypercube Sampling
MLP	Multilayer Perceptron (FNN)
MSE	Mean Squared Error
NODE	Neural Ordinary Differential Equation
ODE	Ordinary Differential Equation
PDE	Partial Differential Equation
PINN	Physics-Informed Neural Network
PIML	Physics-Informed Machine Learning
ReLU	Rectified Linear Unit
ResNet	Residual Neural Network
RRMSE	Relative Root Mean Squared Error (L^2 relative error)
SciML	Scientific Machine Learning
SGD	Stochastic Gradient Descent
SiLU	Sigmoid Linear Unit (Swish function)
SINDy	Sparse Identification of Nonlinear Dynamics
SR	Symbolic Regression
TPU	Tensor Processing Unit
UAT	Universal Approximation Theorem
UDE	Universal Differential Equation
UPINN	Universal Physics-Informed Neural Network
XPINN	Extended Physics-Informed Neural Network

Terminology

Term	Description
Ansatz	A proposed solution to a problem, often based on intuition or prior knowledge.
Dirichlet condition	Boundary condition on the value of the solution at the boundary.
Hyperparameter	Parameter configuring the model or training process. E.g. the number of layers in a neural network or the learning rate of the optimizer.
Inductive prior/bias	Assumptions about the solution structure imposed on the model. E.g. passing partial derivatives to a NN learning a DE.
Low-descrepancy sequence	A sequence of points that are more evenly distributed than random points. E.g. a Sobol sequence.
Neumann condition	Boundary condition on the derivative of the solution at the boundary.
Pseudo	A prefix meaning (in math and computerscience) ”attempting to be” or ”behaving like”. E.g. pseudo-random numbers or pseudo-code.
Quasi	A prefix meaning ”resembling but not actually being”. E.g. sobol sequence being quasi-random or BFGS being a quasi-Newton method.
Runge-Kutta	A family of numerical methods for solving ordinary differential equations.
Spline	A piecewise smooth and continuous function defined by polynomials.
Squashing function	A (typically) monotonic function, mapping a real-valued input to a bounded output. E.g. the sigmoid or hyperbolic tangent functions.
Unispaced/Equispaced	A sequence of points distributed in the domain with equal spacing. E.g. a grid.

Notation

Symbol	Description
L	Number of layers in a neural network.
$\mathbf{W}^{(l)}$	Weight matrix of layer l.
$\mathbf{b}^{(l)}$	Bias vector of layer l.
$z^{(l)}$	Pre-activations of layer l.
$a^{(l)}$	Activations of layer l.
N	Number of samples in dataset.
B	Batchsize: Number of samples in batch.
$\mathcal{D}$	Dataset.
d_l	Size of network layer l. d_0 and d_L are the input and output sizes.
$[d_0, d_1, \ldots, d_L]$	Network architecture.
$\mathcal{L}$	Loss function.
ϵ	Noise level.
$\sigma(\cdot)$	Sigmoid activation function.
$[d] * k$	FNN (or KAN) with k hidden layers of size d.
$[d] * m * k$	ResNet with k residual blocks, each with m hidden layers of size d.
u_x	Partial derivative of u with respect to x, i.e. $\frac{\partial u}{\partial x}$. $u_{xy} = \frac{\partial^2 u}{\partial x \partial y}$.

Contents

Acknowledgements

Abbreviations

Terminology

Notation

1 Introduction **1**
1.1 Previous Work 2
1.2 Scope of the Thesis 3
1.3 Outline 4

Part I Theory **6**

2 Neural Networks **7**
2.1 The Artificial Neuron 7
2.2 Feedforward Neural Networks (FNN) 8
2.3 Activation functions 9
2.4 Universal Approximation Theorem (UAT) 11
2.5 Model Fitting - Training 12
2.5.1 Weight initialization 12
2.5.2 Loss Function 13
2.5.3 Gradient-based Optimization 14
2.5.4 Gradient computation - Backpropagation 15
2.6 Neural Network Architectures 18
2.6.1 Convolutional Neural Networks (CNN) 19
2.6.2 Residual Neural Networks (ResNet) 20
2.6.3 Kolmogorov Arnold Networks (KAN) 21

3 Universal Physics-Informed Neural Networks **25**
3.1 Physics Informed Neural Networks (PINNs) 25
3.1.1 Formulation 26
3.1.2 Activation functions in PINNs 29
3.1.3 Sampling 29

3.1.4 Adaptive loss weighting . . . 32
3.1.5 Regularization . . . 33
3.1.6 Learning rate scheduling . . . 33
3.2 Generalizing PINN (UPINN) . . . 34

4 Symbolic Regression **35**
4.1 SINDy and PDE find . . . 35

Part II Experiments and Results **37**

5 Experimental Preliminaries **38**
5.1 Computational framework . . . 38
5.2 Data . . . 38
5.3 Evaluation metric . . . 39
5.4 Model selection in symbolic regression . . . 39

6 Considered Problems and PINNs **41**
6.1 Lotka-Volterra predator-prey model (ODE system) . . . 41
6.2 Epidemiological compartment model (ODE system) . . . 43
6.3 Burgers' equation (PDE) . . . 46
6.4 Navier-Stokes equations (PDE system) . . . 48

7 Hidden discovery with UPINNs **51**
7.1 Lotka-Volterra with unknown interaction . . . 51
7.1.1 Baseline model . . . 52
7.1.2 Extrapolation with UPINN estimated dynamics . . . 53
7.1.3 Hidden discovery combined with inverse problem . . . 53
7.1.4 Noise and data sparsity . . . 56
7.1.5 Brief note on interpretability using KANs . . . 57
7.1.6 Interaction term discovery with SINDy . . . 58
7.2 Detecting hidden resusceptibility in SIR model . . . 59
7.3 Burgers' equation with unknown convection term . . . 63
7.3.1 Tuning hyperparameters . . . 64
7.3.2 Removing the inductive bias . . . 70
7.3.3 Solving Burgers' equation with PINN using residual network 71
7.3.4 Recovering the convection term with PDE-find . . . 73
7.4 Navier-Stokes with hidden viscosity . . . 75
7.4.1 Improve convergence rate . . . 75
7.4.2 Enhance residual modelling . . . 76

7.4.3 Recover viscosity terms . . . 81

8 Tokamak plasma experiment **83**
8.1 Problem description . . . 83
8.2 Data . . . 83
8.3 PDE residuals of given solution . . . 84
8.4 Model setup . . . 85
8.5 Results . . . 86

9 Discussion of results **90**
9.1 Undesired estimation . . . 90
9.1.1 Undesired estimation of known dynamics . . . 90
9.1.2 Phantom estimation . . . 91
9.1.3 Residual estimation by solution network . . . 92
9.2 Inductive bias . . . 92
9.3 Equation recovery . . . 93
9.4 Future work . . . 94

10 Conclusion **96**

References **98**

Appendix **102**
A Proofs and Derivations . . . 102
A.1 Deep neural networks require non-linear activation functions 102
A.2 Approximating interactions with KAN . . . 102
B Elaboration . . . 104
B.1 More on activation functions in PINNs . . . 104
C Additional Results . . . 107
C.1 Lotka-Volterra . . . 107
C.2 Burgers' . . . 109
C.3 Navier-Stokes . . . 109
C.4 Plasma Physics . . . 110
D Function Libraries for symbolic regression . . . 110
E PyTorch implementation . . . 111
E.1 UPINN framework documentation . . . 111
E.2 Example: UPINN for Burgers' equation . . . 115
E.3 Example: PDE-find for Burgers' equation . . . 119
E.4 Example: SINDy for Lotka-Volterra . . . 120

1 Introduction

Solving differential equations (DE) is central task in science and engineering, as Ordinary and Partial Differential Equations (ODEs and PDEs) are fundamental tools in modelling of dynamical systems. It is often more natural to formulate a system in terms of its dynamical behavior rather than its static properties, as many fundamental laws of physics describe or constrain how a system changes (e.g. conservation, interaction, motion etc.).

Traditional solution methods, such as analytical tools (e.g. Computer Algebra Systems (CAS)) or numerical schemes like Runge-Kutta for ODEs and finite difference/element (FDM/FEM) methods for PDEs, are well established and have proven effective in many applications. However, most of these methods require a fully known model, which might not always be available. If model parameters are unknown, additional assumptions and estimation techniques may be required. In addition to the forward problem of solving DEs, this adds the challenges of the inverse problem, where parameters of a model with known symbolic form are partially or fully unknown, and hidden discovery, where model dynamics are only partially or not at all described, simultaneously raising the issue of model interpretability.

Forward Problem: Solve differential equations with sufficient initial/boundary conditions to predict the system's behavior over the domain.

Inverse Problem: Use observed data from the system to identify unknown parameters.

Hidden Discovery: Detect and learn residuals of dynamical system through data-driven methods.

Interpretability: Uncover governing equations from data to provide model interpretability.

These 4 challenges have gained significant attention in the last decade within Physics Informed Machine Learning (PIML) [1, 2], a research area within the emerging subfield of Deep Learning known as Scientific Machine Learning (SciML). Traditional machine learning methods are often purely data-driven black-box techniques, usually requiring loads of data. Contrary, scientific computing is mostly driven by physical modelling and numerical discretization techniques with little to

no use of data. SciML is a hybrid approach that aims to leverage data in combination with physical knowledge to solve scientific problems.

Physics-Informed Neural Networks (PINNs) [3, 4, 5] were a key development within SciML. Using neural networks as a surrogate, PINNs embed differential equations into the training process, thereby imposing physical constraints while also leveraging data. Since their introduction in 2017, PINNs have been thoroughly studied in the context of solving forward and inverse problems.

Universal Physics-Informed Neural Networks (UPINNs) are a recent (2022) advancement in the field of PINNs. In an attempt to combine PINNs with Universal Differential Equations (UDE) [7], UPINNs add a trainable neural network to the diffential equations, allowing the model to learn functional representations of the residual dynamics while simoultaneously solving the DE. Thus UPINNs can potentially solve both forward, inverse and hidden discovery problems in a unified framework.

1.1 Previous Work

Universal Physics-Informed Neural Networks stand out in the landscape of PIML methods, as they combine the flexibility of neural networks with prior physical knowledge, while leveraging data to simultaneously solve partially known systems and discover their hidden dynamics.

UPINNs combine Universal Differential Equations (UDE) [7] and Physics-Informed Neural Networks. Aknowledging the concept of UDEs, it is relevant to establish why UPINNs are not UDEs. Both use a universal approximator (neural network) to model the residuals of differential equations. UDEs use traditional numerical solvers to simulate the observed data based on the current state of the system. The discrepancy between simulated data and true observations is then computed and used to train the residual network. By contrast, UPINNs leverage a solution surrogate neural network to solve the differential equation directly, and incorporate the residual network in the physics-informed loss computation of the surrogate. This allows information from the data to train the solution surrogate and indirectly train the residual network as well.

Podina et al. initially introduced UPINNs as “A PINN Approach to Symbolic Differential Operator Discovery with Sparse Data” [6]. In 2023 they released a revised paper, in which the term Universal Physics-Informed Neural Networks (UPINNs) first appeared [8]. The paper primarily serves as an introduction of the UPINN framework, highlighting its potential rather than providing a comprehensive anal-

ysis. The authors demonstrate examples of UPINNs for solving hidden discovery problems on the Lotka-Volterra system, Burgers' equation and a Cell Apoptosis model. However, while not comprehensive, only the Lotka Volterra model is explored in any significant detail. The authors aknowledge that their paper is not yet "camera-ready", thus it should be considered a work in progress, which leaves room for further exploration and study of the UPINN framework. This paper is the primary source of inspiration for this thesis.

In the context of hidden discovery and model interpretability, the SINDy and PDE find frameworks [9, 10] have been used to determine governing equations directly from data. Being purely data-driven, these methods rely solely on observed data and interpolation techniques to infer the governing equations. UPINNs do not find governing equations directly, but learns a functional representation of the differential equation residuals in the form of a neural network.

Another promising emergence in the context of model interpretability are Kolmogorov Arnold Networks (KAN) [11]. A recent (2024) neural network architecture, that has also made its appearance in the field of PINNs as PIKANs [12]. Using learnable activation functions, KANs have shown examples of learning highly interpretable computational chains in some applications.

With UPINNs being a functional extension of PINNs, much of the previous work and research done to enhance the performance and training of PINNs [5, 13, 14, 15, 16, 17, 18] might also be relevant for UPINNs. These works are referred throughout section 3.1. Note, despite its name, Extended PINNs (XPINNs) [19] are unrelated to the concept of UPINNs.

During the project period, Podina et al. released a new paper [20] discussing the use of UPINNs with dimensional analysis and the symbolic regression framework AI Feynman [21, 22] for recovering hidden equations. As of writing, March 2025, the AI Feynman framework is exclusive to macOS and Linux-based environments.

1.2 Scope of the Thesis

In simple terms, the focus of this project is to study the UPINN framework on a variety of ODE and PDE systems. Many physical systems involve processes that are not well understood or fully captured by the current models, and being able to use data-driven techniques to discover these hidden dynamics is a significant advancement.

This report serves as a study on the use of UPINNs to discover hidden dynamics on a selection of problems from the literature, and aims to provide insight into

UPINNs capabilities and limitations. The problems examined are systems commonly used to test capabilities of PINNs, and include the ODE systems of the Lotka-Volterra predator-prey model and the epidemic SIR model, as well as the PDE systems of Burgers' viscous equation and Navier-Stokes equations for incompressible fluid flow.

In addition to estimating hidden dynamics, the report also investigates the use of symbolic regression (SR) techniques to recover hidden equations of the, by UPINN, discovered dynamics using SINDy and PDE-find [9, 10].

All experiments are conducted using the Python implementation of the Torch framework [23]. A comprehensive highperformance machine learning library with a wide range of tools and utilities for neural network training and evaluation, also providing easy access and usage of the GPU accelerated CUDA toolkit[1].

1.3 Outline

The report is structured in two parts, a theoretical and an experimental part. This section outlines a brief overview of the contents in each chapter.

Chapter 2 provides a detailed introduction to neural networks, outlining the mathematical foundation of artificial neurons, activation functions, the training process and a selection of network architectures.

Chapter 3 introduces PINNs, covering the mathematical setup and outlining a variety of techniques that have been shown in literature to improve their performance. This ultimately leads to the formulation of UPINNs.

Chapter 4 is a short introduction to the symbolic regression frameworks of SINDy and PDE-find.

Chapter 5 introduces the experimental part of the thesis, outlining the computational framework, data generation, evaluation metrics and model selection.

Chapter 6 presents the considered problems and apply regular PINNs to solve forward and inverse problems.

Chapter 7 applies UPINNs to variations of the problems presented in chapter 6, focusing on residual modelling and hidden dynamics discovery as well as symbolic regression on the discovered dynamics to recover hidden equations.

Chapter 8 is a real-world application of PINNs to a plasma physics problem, at-

[1] https://developer.nvidia.com/cuda-zone

tempting to approximate transport coefficients in a 1.5D cylindrical plasma model.

Chapter 9 serves as a discussion of the results and findings from the UPINN experiments, highlighting the capabilities and limitations of the framework, while also providing suggestions for future work.

Chapter 10 concludes the thesis, briefly summarizing the main findings and contributions of the project.

Part I

Theory

2 Neural Networks

In recent years, the concept of Artificial Intelligence (AI) has gained significant attention by the general public, particularly with the emergence of generative- and large language models such as ChatGPT [24, 25], DALL-E [26, 27], Stable Diffusion [28] and several others. AI is commonly conceptualized as an abstract and highly complex computational system capable of independent thought and intelligent reasoning in response to its received input. However, fundamentally, these models are nothing more than large intricate mathematical functions known as neural networks.

This chapter provides a detailed overview of neural networks. It starts by introducing the concept of an artificial neuron, followed by outlaying the role of neural networks as universal function approximators and the model training pipeline, including the automatic differentiation-based backpropagation algorithm that enables efficient fitting of these models. Finally, a selection of common and relevant neural network architectures are presented.

2.1 The Artificial Neuron

The artificial neuron, visualized in figure 2.1, is the central building block in most neural network architectures and thus fundamental in modern Deep Learning.

Developed through the last century as a generalization of the initial concept of a **perceptron** [29, 30], an artificial neuron (or unit) is a function, $f : \mathbf{R}^d \to \mathbf{R}$, computing a shifted weighted sum of its inputs, $\mathbf{x} = \{x_i\}_{i=1}^d$. The weighted sum, given by equation (2.2), is composed with a (typically) non-linear activation function, $h : \mathbf{R} \to \mathbf{R}$ (see section 2.3), to obtain the output a given by equation (2.1) and denoted as the **activation** of the neuron.

$$a = f(\mathbf{x}) = (h \circ z)(\mathbf{x}) = h(\mathbf{w}^T\mathbf{x} + b), \quad \mathbf{w}, \mathbf{x} \in \mathbf{R}^d, \quad b \in \mathbf{R}, \tag{2.1}$$

$$z(\mathbf{x}) = \mathbf{w}^T\mathbf{x} + b = \sum_{i=1}^{d} w_i x_i + b. \tag{2.2}$$

The **weights**, $\mathbf{w} = \{w_i\}_{i=1}^d$, determine the influence of each input feature on the output, while the **bias**, b, serves as an additional parameter shifting the weighted sum before applying the activation function. To simplify notation, the bias can be implicitly incorporated into the weight vector $\mathbf{w}$ by introducing an additional

weight, $w_0 = b$, and a corresponding input feature, $x_0 = 1$, leading to the simplified expression (2.3),

$$f(\mathbf{x}) = h(\mathbf{w}^T\mathbf{x}). \tag{2.3}$$

This project considers only real-valued neurons, though this is not a strict requirement and the concept can generalized to complex numbers as well. However, this is beyond the scope of this report and will not be further discussed.

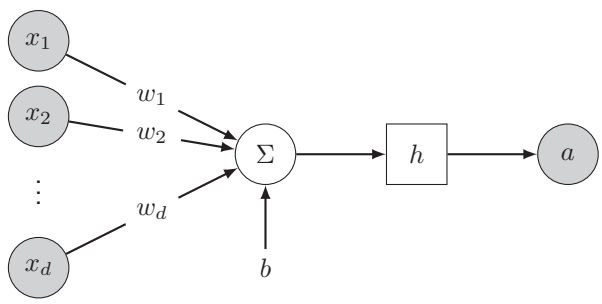

Figure 2.1: The Artificial Neuron.

The original perceptron model, proposed by Rosenblatt in 1958, is a special case of the artificial neuron, taking d binary inputs and using a Heaviside step function as activation function to yield a single binary output. This early neural model is a simple, and arguably limited [31], binary classification model.

2.2 Feedforward Neural Networks (FNN)

Feedforward Neural Networks, often referred to as Multilayer Perceptrons (MLPs) as a reference to the original perceptron model [30], extends the idea of the single artificial neuron to a connected system of multiple neurons.

As the name suggests, information flows forward through the network, passing a series of layers, each consisting of multiple neurons. The ordered layers, fully connects the neurons in adjacent layers, but not within the same layer (see figure 2.2). Put simply, a neuron computes a weighted sum of all neurons in the previous layer, applies an activation function, and passes the result to every neuron in the next layer.

Consider a collection of d artificial neurons, each taking d_{in} inputs. Associated with each neuron, i, is a weight vector $\mathbf{w}_i$ and a bias b_i. Then for each neuron, a transformation of the input $\mathbf{x} \in \mathbf{R}^{d_{\text{in}}}$ is computed by equation (2.2). As these are all vector operations, the transformation of all neurons may be written as the

matrix-vector product (2.4),

$$z(\mathbf{x}) = \mathbf{W}\mathbf{x} + \mathbf{b}, \quad \mathbf{W} = \begin{pmatrix} \mathbf{w}_1^T \\ \vdots \\ \mathbf{w}_d^T \end{pmatrix}, \quad \mathbf{b} = \begin{pmatrix} b_1 \\ \vdots \\ b_d \end{pmatrix}. \tag{2.4}$$

Composed with an activation function, h, this defines a single layer. Stacking multiple layers of neurons, where the output of each layer serves as input to the next, results in a feedforward neural network. Thus the compositional function (2.5),

$$f_\theta(\mathbf{x}) = h_L \circ z_L \circ h_{L-1} \circ z_{L-1} \circ \ldots \circ h_1 \circ z_1(\mathbf{x}), \tag{2.5}$$

defines a FNN with L layers parameterized by $\theta = \{\mathbf{W}^{(l)}, \mathbf{b}^{(l)}\}_{l=1}^{L}$. The architecture is illustrated in figure 2.2.

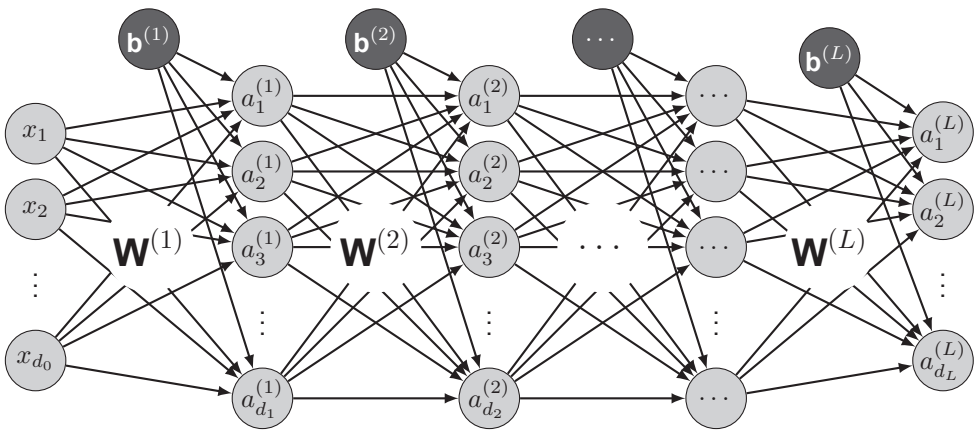

Figure 2.2: Architecture of the Feedforward Neural Network. Superscripts denote the layer index.

While not a strict requirement, all neurons in a layer typically use the same activation function. The number of layers and how many neurons to include in each layer are part of the model **hyperparameters**, which is a set of configurable parameters defining the model and training process. These are often chosen by intuition, experience, or ideally by systematic hyperparameter optimization using a validation scheme like cross-validation to obtain good hyperparameters.

2.3 Activation functions

Activation functions are fundamental to the flexibility of neural networks, as they introduce non-linearities, allowing networks to effectively incorporate multiple layers of neurons and thereby model intricate functions. Without them, a FNN is mathematically equivalent to a single-layer network (as shown in appendix A.1). The choice of activation function(s) should align with the problem being solved and the desired properties of the network. Some examples of activation functions

are shown in figure 2.3.

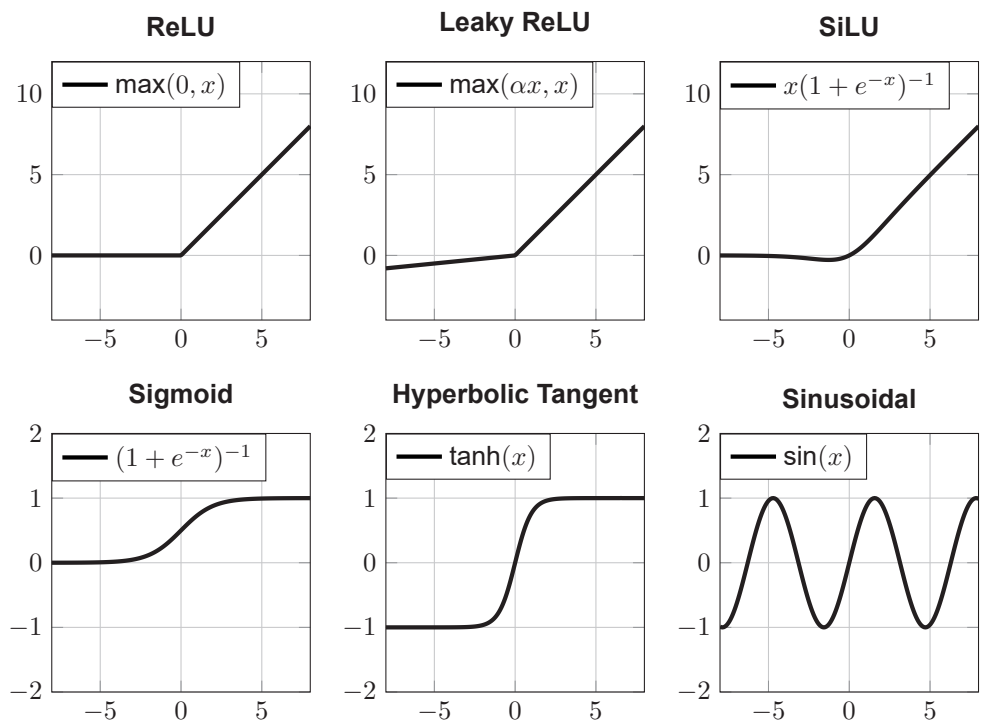

Figure 2.3: Selection of activation functions.

In addition to non-linearity, activation functions should be defined over all real numbers and ideally be smooth (i.e. differentiable), as it is crucial to run the gradient based backpropagation algorithm used for model training, which is outlayed in section 2.5.4. Although ReLU and Leaky ReLU activation functions are popular in modern deep neural networks, they are in fact not smooth at $x = 0$. Nevertheless, this is typically not an issue in practice, as their derivatives are defined at all other points, and $x = 0$ is an uncommon input value that can be handled seperatly if needed.

In classification tasks, the output layer often uses the softmax activation function, which normalizes the output to a probability distribution, ensuring the outputs sum to 1. Furthermore, it amplifies the relative difference between the largest and smallest output values. In this setting, the index of the largest output value would usually be interpreted as the predicted class. The softmax function is defined by equation (2.3),

$$\text{Softmax}(\mathbf{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{d} e^{z_j}}, \quad i = 1, 2, \ldots, d, \quad \mathbf{z} \in \mathbf{R}^d.$$

Choosing an appropriate activation function is quite an important part of designing

a neural network, as it can affect the capabilities and convergence of the model. Some considerations to make when choosing an activation function are (but not limited to):

- **Non-linearity:** The functions that a neural network aims to model are generally non-linear. As mentioned, modelling of complex functions requires non-linear activation functions, which are also crucial for the layered structure of neural networks.
- **Smoothness:** Differentiability is crucial for training neural networks using gradient based optimization algorithms.
- **Saturation/Vanishing gradients:** Some bounded or partially bounded activation functions, like Sigmoid and Tanh, can suffer from vanishing gradients for numerically large inputs as the output saturates to the bound, which can slow down and sometimes halt convergence of the training.
- **Zero-centered:** Having an activation function that is zero-centered can increase convergencerate, as it allows for update of both positive and negative weights simultaneously.

2.4 Universal Approximation Theorem (UAT)

Having established what a neural network is, it seems appropriate to motivate why they are useful at all and how their use is justified.

The holy grail is the **Universal Approximation Theorem**, stating;

"A feedforward neural network with a single hidden layer containing a finite number of neurons, is capable of approximating any continuous function defined on a compact domain, with arbitrary accuracy, provided enough neurons."

Formally, if $f : \mathbf{R}^d \to \mathbf{R}^o$ is a continuous function defined on a compact subset $\Omega \subset \mathbf{R}^d$ and $\epsilon > 0$, then there exist a single layer neural network f_θ with parameters θ such that,

$$\sup_{\mathbf{x}\in\Omega} |f(\mathbf{x}) - f_\theta(\mathbf{x})| < \epsilon. \tag{2.6}$$

It was first proven in 1989 by Cybenko [32] for sigmoidal activation functions, followed by an independent generalization to all squashing activation functions by Hornik et al. [33] later that year. Leshno et al. later showed that the theorem also holds for any non-polynomial activation function [34].

While UAT states that a single hidden layer is sufficient, in practice, deep multi-

layer neural networks are often used. Deeper networks can learn hierarchical representations of data, allowing them to model intricate functions more compactly, i.e. requiring fewer neurons, compared to shallow networks [35].

2.5 Model Fitting - Training

Motivated by the universal approximative capabilities of neural networks, the next question is how one gets the network to actually approximate the desired function. The process of tuning the networks parameters to make it fit a particular function, is commonly known as training. This section is a step by step outline of the general idea for the neural network training process. The structure is as follows:

- **Weight initialization:** Find a starting point for the optimization process.
- **Loss function:** Define the objective of the optimization process.
- **Gradient-based optimization:** Adjust the weights to minimize the loss function.
- **Gradient computation - Backpropagation:** Compute gradients efficiently for gradient-based optimization.

2.5.1 Weight initialization

Apart from the network architecture itself, the weights are what defines the neural network. These weights can be thought of as small dials that are tuned during the training process, such that the network approximates the desired function.

As will become apparent, training is an iterative process where weights are constantly adjusted based on their current value. Subsequently, the weights must be assigned initial values before training. The choice of initial weights can significantly impact convergence of the training process, as well as the networks ability to approximate the target function accurately.

The detailed theory of weight initialization goes beyond the scope of this project, but worth mentioning is that the choice of method often depends on the used activation function in the network. The most common weight initialization schemes originates from the work of Glorot et al. [36] and He et al. [37]. Both of these approaches use a random sampling strategy based on the number of neurons in a layer. The aim is to keep variance in activations and gradients stable throughout the network, while simultaneously addressing the issue of vanishing and exploding gradients. Some common weight initialization schemes are shown in table 2.1.

Initialization	Distribution	Suited for
Xavier/Glorot Uniform [36]	$w_i \sim \mathcal{U}\left(-\sqrt{\frac{6}{n_i+n_{i+1}}}, \sqrt{\frac{6}{n_i+n_{i+1}}}\right)$	Squashing activation functions
Xavier/Glorot Normal [36]	$w_i \sim \mathcal{N}\left(0, \sqrt{\frac{2}{n_i+n_{i+1}}}\right)$	Squashing activation functions
He Uniform [37]	$w_i \sim \mathcal{U}\left(-\sqrt{\frac{6}{n_i}}, \sqrt{\frac{6}{n_i}}\right)$	ReLU variants
He Normal [37]	$w_i \sim \mathcal{N}\left(0, \sqrt{\frac{2}{n_i}}\right)$	ReLU variants

Table 2.1: Common weight initialization schemes. n_i and n_{i+1} are the number of neurons in the current and next layer, respectively.

2.5.2 Loss Function

The goal of training is to adjust the weights of the network such that it approximates the desired function accurately. However, to achieve this, it is necessary to define what exactly it means to approximate a function accurately. Entering the **loss function**, often referred to simply as the **loss** or **cost** function.

The loss function quantifies how well the neural network, given its parameters, approximates the target function. Considering loss as an error measure, it becomes evident that training is an optimization problem with the goal of minimizing loss over the network weights. Formally, a network f_θ parameterized by θ, has a set of optimal parameters, with respect to the loss function $\mathcal{L}$, given by equation (2.7),

$$\theta^* = \arg\min_\theta \mathcal{L}(\theta). \tag{2.7}$$

The target function is generally unknown, hence the loss function serves merely as an approximation of the networks error across the domain D, computed on **training data**, i.e. discrete input-output pairs $(\mathbf{x}_i, \mathbf{y}_i)$ sampled from the target function. The assumption is that these samples are sufficiently representative of the overall structure of the target function, enabeling the network to learn an accurate approximation. A commonly used loss function in regression settings is the mean squared error (MSE) given by equation (2.8),

$$\mathcal{L}(\theta) = \frac{1}{n}\sum_{i=1}^{n} \|f_\theta(\mathbf{x}_i) - \mathbf{y}_i\|^2, \tag{2.8}$$

where $f_\theta(\mathbf{x}_i)$ is the network prediction for input $\mathbf{x}_i$, and $\mathbf{y}_i$ is the corresponding known target value.

The loss function should always be chosen as to reflect the desired behavior of the model. MSE and mean absolute error (MAE) are well suited for regression tasks, where cross-entropy is often used for classification. Additionally, regularization terms can be added to the loss function as to prevent overfitting. This is usually

done by adding a penalty term to the loss function, e.g. L^1 or L^2 regularization, which penalizes large weights according to equation (2.9),

$$\mathcal{L}_{\text{reg}}(\theta) = \|\mathbf{W}_i\|, \tag{2.9}$$

where $\|\cdot\|$ is a norm, usually the L^1 or L^2 norm.

Having multiple loss terms, it may happen that they do not align, hence optimization becomes a trade-off between the different loss components. Furthermore, the seperate loss terms may be weighted differently to emphasize the relative importance of each term in the optimization process.

2.5.3 Gradient-based Optimization

Having defined the goal of the training process, the next step is to devise a strategy for adjusting the weights of the network as to minimize the loss function. Generally, the loss is a complicated and non-linear function of the network weights, hence analytically obtaining the optimal set of weights is intractable.

Therefore, optimization is typically carried out using an iterative gradient based algorithm. The perhaps most simple of such algorithms is **Gradient Descent** (GD).

Gradient descent makes weight updates following the direction of steepest descent, i.e. the negative gradient of the loss function with respect to the network weights. Thus, the update rule for GD is given by (2.10),

$$\theta \leftarrow \theta - \eta \nabla_\theta \mathcal{L}(\theta), \tag{2.10}$$

where η is the **learning rate**, a hyperparameter controlling the step size of the updates, usually a small value around 10^{-3}.

If the computational cost gets too large or the entire dataset cannot fit into memory, weight updates can be done on subsets of the data, commonly referred to as **batches**. The N samples if the entire dataset are partitioned into batches $\{\mathcal{B}_i\}_{i=1}^{N/B}$ of size B, usually sampled without replacement. The loss and gradients are then computed for each batch, and weights are updated according to (2.10), either iteratively after processing each batch or by averaging the gradients over all batches in the dataset. Due to the stochastic nature of batching, this is known as **Stochastic Gradient Descent** (SGD). For $B = N$, SGD is equivalent to standard GD.

Each full pass through the entire dataset, i.e. all batches, is known as an **Epoch**.

The weights are iteratively updated according to (2.10) until some stopping criteria is met, typically that the loss has converged to a stable value or some predefined number of epochs has been reached. The full algorithm is outlined as algorithm 1.

Algorithm 1 Stochastic Gradient Descent (SGD)

Input: Learning rate η, Batch size B, Number of samples N, Stopping criterion
Initialize: Weights $\theta = \{\mathbf{W}_i, \mathbf{b}_i\}_{i=1}^{L}$ ▷ E.g. Xavier/Glorot or He
while stopping criterion not met **do** ▷ E.g. convergence or max epochs
    Randomly shuffle dataset $\mathcal{D}$
    Partition dataset into N/B batches $\{\mathcal{B}_i\}_{i=1}^{N/B}$
    for $i = 1$ to N/B **do**
        Compute gradients $\nabla_\theta \mathcal{L}(\theta; \mathcal{B}_i)$
        Update weights $\theta \leftarrow \theta - \eta \nabla_\theta \mathcal{L}(\theta; \mathcal{B}_i)$
    end for
end while

While the standard SGD algorithm has fallen out of favor, many of the most popular algorithms built upon it, improving its performance. Some of the most popular are SGD with momentum [38] and Adaptive Moment Estimation algorithms (Adam and AdamW) [39, 40]. Both of these algorithms incorporate momentum in the updates and adapt the learning rate throughout the training process.

Other optimization algorithms, such as second order quasi-Newton methods like BFGS [41] and especially its memory limited variant L-BFGS [42], have also proven effective for training neural networks. The terms ”quasi-Newton” and ”second order” refers to these algorithms using approximations of the loss function Hessian. Although these methods are computationally more expensive than first order methods, they often take more efficient steps towards the optimum, leading to faster convergence, allbeit, this does also make them prone to getting stuck in local minima.

2.5.4 Gradient computation - Backpropagation

Gradient-based optimization algorithms require gradient computation of the loss function with respect to the network parameters, i.e. weights and biases. As gradients are used for every weight update and directly influence the steps, thereby also convergence of the optimization process, it is crucial that they are computed efficiently and with high precision. Specifically, the gradient of the loss function with respect to the network weights is needed.

Symbolic differentiation, either manually or by CAS, provide exact symbolic representations of gradients. However, neural networks typically comprise long compositional chains of functions, which, due to the product rule for derivatives, makes the symbolic representation of gradients grow exponentially, making it computa-

tionally infeasible.

Numerical differentiation techniques, such as finite differences method, usually only require that the loss function can be evaluated. However, they are approximate and can be computationally expensive due to multiple evaluations of the loss function for each parameter. Furthermore, the limited precision of floating-point arithmetic can introduce numerical instability in these methods.

The method used in Deep Learning is known as automatic differentiation (AD), which provide exact gradients without relying on full symbolic representations. Through the chain rule, AD recursively unrolls the function into a computational graph of elementary operations with known symbolic derivatives. This overcomes the exploding representation of symbolic differentiation by computing gradients in a sequential manner and reusing intermediate results.

As neural networks are generally multivariate vector functions, the general multivariate **Chain rule** is used. Consider multivariate vector functions $h : \mathbf{R}^k \to \mathbf{R}^n$ and $z : \mathbf{R}^m \to \mathbf{R}^k$ and let $\mathbf{x} \in \mathbf{R}^m$ and $z(\mathbf{x})$ be inputs of h and z respectively. The chain rule states that the gradient of the composition $f = h \circ z$ with respect to $\mathbf{x}$ is given by equation (2.11),

$$\frac{\partial f}{\partial x} = \frac{\partial h}{\partial z}\frac{\partial z}{\partial x}. \tag{2.11}$$

Notationwise, $\frac{\partial f}{\partial x}$ denotes the Jacobian matrix comprising all first-order partial derivatives, formally defined in equation (2.12),

$$\frac{\partial f}{\partial x} = \left[\frac{\partial f_i}{\partial x_j}\right]_{1<i<n,\, 1<j<m}, \quad \frac{\partial}{\partial x} f(\mathbf{x}_0) = \frac{\partial f}{\partial x}\bigg|_{x=\mathbf{x}_0}. \tag{2.12}$$

In literature, the Jacobian is commonly denoted by $\mathbf{D}f$, $\mathbf{J}_f$ or ∇f, depending on the context. However, for the sake of clarity in terms of what variable gradients are taken with respect to, the notation $\frac{\partial f}{\partial x}$ is preferred here. In the context of Deep Learning, the chain rule is used to compute the gradient of the loss function with respect to the network weights. This is done in a layer by layer manner. Starting at the end, the Jacobian of the loss function with respect to the network output is given by equation (2.13),

$$\frac{\partial L}{\partial a^{(L)}} = \nabla_{a^{(L)}} \mathcal{L}. \tag{2.13}$$

Next consider the gradients of the output activations $a^{(l)}$ with respect to the inputs $a^{(l-1)}$ in a single layer l, which by the chain rule (2.11) can be written as (2.14),

$$\frac{\partial a^{(l)}}{\partial a^{(l-1)}} = \frac{\partial a^{(l)}}{\partial z^{(l)}}\frac{\partial z^{(l)}}{\partial a^{(l-1)}}. \tag{2.14}$$

Using the definition of z given in (2.4) and the rules of matrix calculus [43], the Jacobian of $z^{(l)}$ is simply the weight matrix $\mathbf{W}^{(l)}$. Note that the Jacobian $\frac{\partial a^{(l)}}{\partial z^{(l)}}$ is a diagonal matrix with the derivative of the activation function of layer l evaluated at each input along the diagonal. Thus the Jacobian in (2.14) can be written as,

$$\frac{\partial a^{(l)}}{\partial a^{(l-1)}} = \text{diag}\left(\frac{dh^{(l)}}{dz^{(l)}}\right)\mathbf{W}^{(l)}. \tag{2.15}$$

Now consider the Jacobian of the output activations with respect to the weights in a single layer l. Without loss of generality, focus on the weights $w_i^{(l)}$ of a single input neuron i in layer l, that is, the i'th column of the weight matrix $\mathbf{W}^{(l)}$. Then by the chain rule,

$$\frac{\partial a^{(l)}}{\partial w_i^{(l)}} = \frac{\partial a^{(l)}}{\partial z^{(l)}}\frac{\partial z^{(l)}}{\partial w_i^{(l)}}. \tag{2.16}$$

Writing out $z^{(l)}$ according to (2.4), the Jacobian $\frac{\partial z^{(l)}}{\partial w_i}$ becomes apparent,

$$z^{(l)} = \sum_{j=1}^{n_{l-1}} w_j^{(l)} a_j^{(l-1)} + b^{(l)} \quad \Rightarrow \quad \frac{\partial z^{(l)}}{\partial w_i^{(l)}} = \mathbf{I} a_i^{(l-1)},$$

thus the Jacobian (2.16) can be written as equation (2.17),

$$\frac{\partial a^{(l)}}{\partial w_i^{(l)}} = \text{diag}\left(\frac{dh^{(l)}}{dz^{(l)}}\right) a_i^{(l-1)}. \tag{2.17}$$

As explained in section 2.1, the bias can be incorporated into the weight matrix, thus gradients with respect to the bias can be computed the same way.

Putting everything together, the gradients of the loss function with respect to the weights of input i in layer l can be expressed as equation (2.18), obtained by repeatedly applying the chain rule,

$$\frac{\partial \mathcal{L}^{(l)}}{\partial w_i^{(l)}} = \frac{\partial \mathcal{L}}{\partial a^{(L)}}\frac{\partial a^{(L)}}{\partial a^{(L-1)}}\cdots\frac{\partial a^{(l+1)}}{\partial a^{(l)}}\frac{\partial a^{(l)}}{\partial w_i^{(l)}}, \tag{2.18}$$

which can also be written in recursive form as (2.19),

$$\frac{\partial \mathcal{L}^{(l)}}{\partial w_i^{(l)}} = \delta^{(l)}\frac{\partial a^{(l)}}{\partial w_i^{(l)}}, \quad \delta^{(l)} = \delta^{(l+1)}\frac{\partial a^{(l+1)}}{\partial a^{(l)}}, \quad \delta^{(L)} = \frac{\partial \mathcal{L}}{\partial a^{(L)}}. \tag{2.19}$$

At this point, equation (2.18) is still a symbolic form of the gradients. It also hints as to why storing the full symbolic representation of the gradients is infeasible. Multiplying out the chain of matrix products results in an enormous expression that contains many redundant terms. As a matter of fact, multiplying together k square

matrices of size $n \times n$ results in a matrix with $O(n^{k+1})$ symbolic terms. For some of the networks used in this report, which comprise 8 layers of 64 neurons each, the number of matrix multiplications (disregarding diagonal matrices as they do not increase the number of terms) amount to 7. This leads to more than $64^8 > 10^{14}$ terms in the symbolic representation of the gradients, just for the input layer.

This raises the natural question of what has then been achieved by setting up gradient computation in this way. The answer lies in the process of a **forward pass**, where all activations and intermediate results are computed, and stored. This allows evaluation of the Jacobians and multiplying them numerically rather than symbolic. Hence, Automatic Differentiation evaluates symbolic expressions for gradients of elementary operations, and multiplies them to get the full gradient, exploiting the computational graph through repeated use the chain rule.

One remaining consideration is whether to multiply the matrices from right to left, or left to right. It may seem irrelevant at first, but depending on the usecase it can significantly impact the computational cost. In general, when using equation (2.18), the left most Jacobian is a matrix, while the right most, the gradient of the loss function with respect to the network output, is a vector. Starting from the left results in repeated matrix-matrix products, while starting from the right means repeated matrix-vector products. Thus for neural networks, the latter is typically faster as it requires less operations. These two methods are known as **Forward-mode** and **Reverse-mode** AD, respectively.

Note how equation (2.19) indicates that when performing a **backward pass**, i.e. running the recursion ”bottom-up” starting with $\delta^{(L)}$, gradients propagate backward through the network. Thus gradients with respect to $\mathbf{W}^{(l)}$ can be obtained immediatly after computing $\delta^{(l)}$, thus avoiding redundant computation. In Deep Learning, this reverse-mode AD is known as **Backpropagation** [44]. Its introduction allowed efficient training of deep neural networks and remains state of the art for computing gradients in training deep learning models.

2.6 Neural Network Architectures

The feedforward neural network has proven useful in many applications, but it only represent a small part of the broad field of neural networks. FNNs is just one example of a neural network **architecture**. Architectures define the structure of a deep learning model, describing how neurons connect and information flows, representing different design philosophies of how a neural network may function.

The rise of deep learning has paved the way for development of many new archi-

tectures, some of which have become state of the art in specific tasks. This section introduces the popular architectures of Convolutional Neural Networks (CNN) [45] and Residual Neural Networks (ResNet) [37], as well as the recently introduced novel Kolmogorov Arnold Networks (KAN) [11].

2.6.1 Convolutional Neural Networks (CNN)

Convolutional Neural Networks are a class of models particularly well suited for tasks involving signal processing. They extract features from an input signal (e.g. an image) by convolution with a kernel (filter), typically small like 3×3 or 5×5 (see figure 2.4). Traditionally, such operations relied on handcrafted kernels (e.g. Sobol, Ridge, Gaussian, etc.) designed to capture specific features like edges, corners, textures etc. CNNs automate this process by learning kernels directly from data.

A convolutional layer typically comprise multiple kernels, each extracting different features from the input. The output, often referred to as a feature map, can be passed through activation functions and additional convolutional layers to extract more complicated features. Depending on the task, the final feature maps may be flattened and passed through a FNN to make a prediction, e.g. classifying the contents of an image.

An inherent advantage of CNNs is translation invariance, meaning features can be detected regardless of position in the signal. Contrary to FNNs, where each input has its own set of weights, CNNs share weights across the input using kernels, making them more efficient for large inputs like images. Consequently, convolutional layers can also handle inputs of arbitrary size.

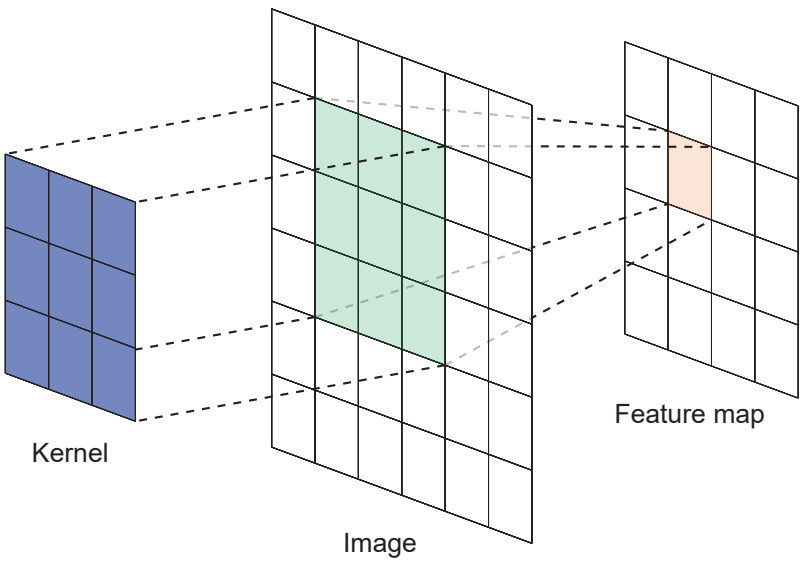

Figure 2.4: Discrete 2D convolution of an image with 3×3 kernel.

Although CNNs were introduced in the 1990s [46, 45], their widespread adoption first happened in the 2010s with the rise of computational power and popularization of deep learning [35, 47]. Today, CNN based architectures are widely used, i.e. U-net [48] (efficient for medical image segmentation and other data limited regimes) and YOLO [49] (YOLOv8 state of the art for real time object detection) etc.

2.6.2 Residual Neural Networks (ResNet)

Residual Neural Networks [37] (not to be confused with Recurrent Neural Networks (RNNs)) introduce the concept of skip connections, a direct connection between two layers bypassing one or more intermediate layers. Formally, if f_θ is a neural network (of arbitrary architecture), a **residual block** is defined as,

$$R(\mathbf{x}) = f_\theta(\mathbf{x}) + \mathbf{W}\mathbf{x}, \tag{2.20}$$

where $\mathbf{W}$ is a weight matrix representing a linear mapping projecting the input $\mathbf{x}$ to match the dimension of the output of f_θ. For equisized input and output, He et al. [37] found the identity mapping to be sufficient.

The idea behind ResNets is that the desired output of a layer is (potentially) only a small deviation from the input. Consequently, the network can learn the residual, i.e. the difference between the input and output. This approach means the network can focus on learning small adjustments rather than trying to reconstruct the entire output from scratch. This results in network layers with fewer parameters, allowing many ResNet blocks to be stacked while maintaining efficient training. Additionally, skip connections allow direct flow of gradients through the network, mitigating the vanishing gradient problem.

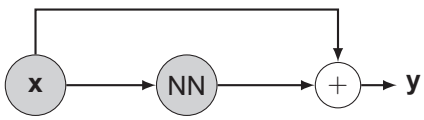

Figure 2.5: ResNet block with skip connection.

Following its introduction in 2015, ResNet established itself as state of the art for many tasks, particularly for image classification using ResNet blocks with CNNs. However, in some domains, like image processing tasks, more advanced architectures like Transformers [25] and Vision Transformers (ViT) [27] (not covered in this report) have begun to outperform them.

2.6.3 Kolmogorov Arnold Networks (KAN)

Kolmogorov Arnold Networks are a recent (2024) advancement in neural network architectures, introduced by Liu et al. [11]. While related, the idea behind KAN is fundamentally different from that of traditional neural networks, as it arises from the Kolmogorov-Arnold representation theorem [50, 51] (KAT) rather than the UAT.

The KAT states that any multivariate continuous function $f : \mathbf{R}^d \to \mathbf{R}$ defined on a bounded domain, can be represented as a finite composition of univariate continuous functions and the binary operation of addition,

$$f(\mathbf{x}) = \sum_{i=1}^{2n+1} \Phi_i \left(\sum_{j=1}^{n} \phi_{i,j}(x_j) \right), \tag{2.21}$$

where $\Phi_i : \mathbf{R} \to \mathbf{R}$ and $\phi_{i,j} : [0,1] \to \mathbf{R}$.

The following outlines how this can be used to design a neural network architecture, primarily following the work of Liu et al. [11] but with a slight modification of notation for clarity and consistency with section 2.2.

Letting $\phi_{i,j}(x_j) = w_{i,j} x_j$ and $\Phi_i(x) = a(x)$, (2.21) seems closely related to (2.1), a neural network type construction. In fact, this is equivalent to a FNN with a single hidden layer of $2n+1$ units, taking n inputs and yielding a single output.

The key difference from a FNN is that $\phi_{i,j}$ can represent any continuous function. To effectively exploit this, choosing an appropriate (e.g. flexible) parameterization of $\phi_{i,j}$ is crucial, as it allows the network to accuratly approximate these functions. The parameterization is free to choose, and could actually be a neural network itself. The authors of [11] suggest using a basis of **B-splines** (piecewise smooth and continuous functions defined by polynomials) alongside a SiLU (see figure 2.3) **base activation** function acting as a residual connection. Formally,

$$\begin{aligned} \phi_{i,j}^{(l)}(x) &= w_{b,i,j}^{(l)} b_{i,j}^{(l)}(x) + w_{s,i,j}^{(l)} \text{spline}_{i,j}^{(l)}(x), \\ b_{i,j}^{(l)}(x) &= \text{SiLU}(x) = \frac{x}{1+e^{-x}}, \\ \text{spline}_{i,j}^{(l)}(x) &= \sum_{k=1}^{n} w_{i,j,k}^{(l)} B_k(x), \end{aligned} \tag{2.22}$$

where $B_k(x)$ are B-spline basis functions, and $\theta = \{\mathbf{W}_b^{(l)}, \mathbf{W}_s^{(l)}, \mathbf{W}^{(l)}\}$ are learnable parameters. Using (2.22), a single **KAN layer** l is defined as,

$$\Phi^{(l)}(\mathbf{x}) = \left[\Phi_i^{(l)}(\mathbf{x}) \right]_{i=1}^{d_{l+1}}, \quad \Phi_i^{(l)}(\mathbf{x}) = \sum_{j=1}^{d_l} \phi_{i,j}^{(l)}(x_j). \tag{2.23}$$

Note that all operations in a KAN layer are smooth, thus it is fully differentiable. This means the trainable parameters can be optimized using backpropagation (see section 2.5.4).

The weights $\mathbf{W}_b^{(l)}$ are initialized using Xavier/Glorot (see section 2.5.1), while $\mathbf{W}_s^{(l)} = \mathbf{1}$ and $\mathbf{W}^{(l)}$ is initialized such that $\text{spline}_{i,j}^{(l)}(x) \approx 0$, i.e. the splines are initially deactivated. Note the total number of parameters in a KAN layer is $O(d_l d_{l+1} n)$, where an analogous FNN layer would have $O(d_l d_{l+1})$.

With this definition, the KAT can be viewed as a 2 layer KAN. However, the use of trainable parameterizations of the univariate functions turns it into an approximation rather than an exact representation. Thus, using such a KAN directly might not be desirable nor practical. The KAT only requires the univariate functions to be continuous, not necessarily smooth, which may limit the effectiveness of the B-spline basis. Stacking multiple KAN layers allows the network to capture more complex functions by composing multiple univariate functions. Consider the composite function,

$$f(x_1, x_2) = \exp(\sin(x_1^2 + x_2^2)).$$

While the KAT ensures existence of a 2-layer KAN representation of this function, it can trivially also be fully represented by a 3-layer KAN copying the composition exactly. In this example, the first layer being quadratics, the second a sine and the third an exponential function. This 3-layer representation may be easier to approximate by the network than the 2-layer representation, while also offering better interpretability of the learned intermediate functions.

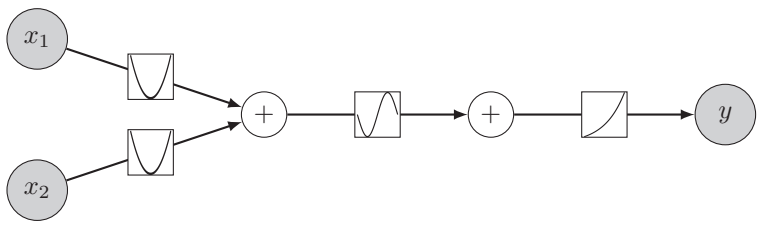

Figure 2.6: Illustration of a 3-layer KAN representing the function $f(x_1, x_2) = \exp(\sin(x_1^2 + x_2^2))$.

To allow this flexibility, KANs may consist of multiple stacked layers, making them deep as illustrated in figure 2.7. Additionally, each layer also has a width, referring to the number activation functions it approximates for each input (analogous to hidden units of a FNN). As mentioned, a KAN layer has n times more parameters than a FNN layer with the same number of input and output neurons. However, KAN layers have shown to be more expressive and can thus be significantly narrower than FNN layers, reducing the number of parameters in the hidden layers.

Liu et al. show examples of KANs outperforming FNNs in certain tasks while having fewer parameters. Although training KANs is slower compared to FNNs, they have shown faster convergence rates in terms of epochs.

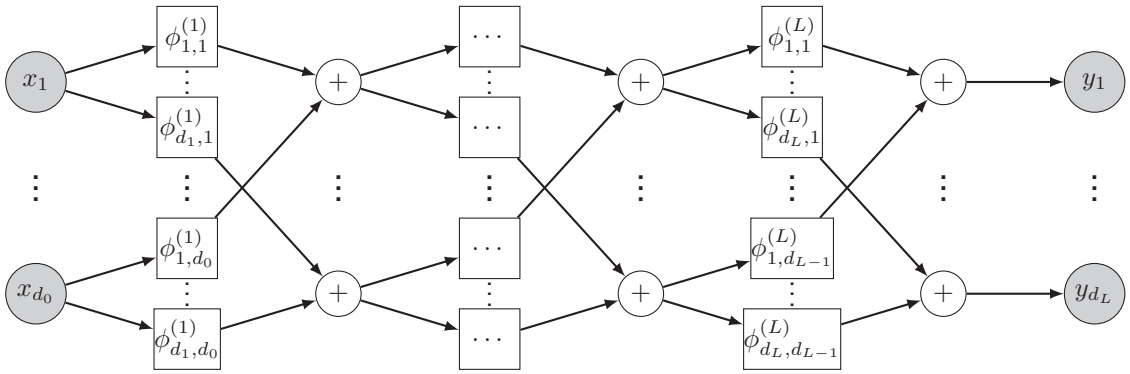

Figure 2.7: The Kolmogorov Arnold Network (KAN) architecture.

The grid of B-splines in KANs provide an inherent advantage over FNNs. Increase the resolution of a FNN, i.e. add more neurons, requires changing the architecture and retraining the network. In contrast, KANs can increase their resolution by simply adding more B-splines and resume training, a process known as **grid extension** (see figure 2.8).

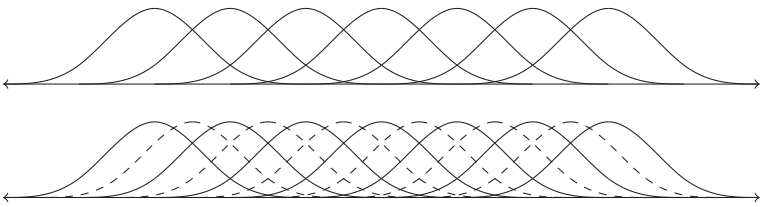

Figure 2.8: **Top:** Cubic B-spline basis for function approximation. **Bottom:** Grid extension of basis, dashed are added basis functions.

The number of B-splines, n, is a central part of what makes KANs computationally expensive. To allow gridextension while simoultaneously control the number of parameters, the authors of KAN suggest imposing a sparsity constraint on the weights and prune inactive neurons of the network prior to grid extension.

Kiyani et al. [18] recently did a study on several PDEs with a wide range of optimizers in the context of training KANs as Physics-Informed Neural Networks. They found that the quasi-Newton method SSBroyden with Wolfe linesearch is generally superior for training PIKANs [12].

KANs are still in their infancy, and the full potential is yet to be explored. However, the KAN architecture is a novel approach to neural network design, and its flexibility and interpretability properties may prove useful in a wide range of ap-

plications, such as machine learning, scientific computing, and physics-informed machine learning.

3 Universal Physics-Informed Neural Networks

Universal Physics-Informed Neural Networks (UPINNs) [8] is a recent (2023) advancement in the field of Physics-Informed Machine Learning (PIML), attempting to generalize the idea of Physics-Informed Neural Networks (PINNs). The PINN framework is a technique for training neural networks to solve differential equations with a surrogate by utilizing domain knowledge and data. While PINNs have shown great results in solving both forward and inverse problems, its formulation assumes that the underlying physics, i.e. the governing equations, are complete. UPINN relaxes this assumption by modelling residuals of the governing equations with a neural network.

This chapter initially outlines the formulation of the PINN framework in detail, along some important advancement and results made in recent years regarding training and optimization of PINNs. It then proceeds by extending to the UPINN framework.

3.1 Physics Informed Neural Networks (PINNs)

Since its introduction by Raissi et al. in 2017, Physics-Informed Neural Networks (PINNs) [3, 4, 5] have enjoyed great attention from the SciML community. Unlike many traditional numerical methods that solve differential equations on a set of discrete points in the domain, PINNs use a surrogate network to approximate the solution of a differential equation, thus the approximate solution exist continuously on the domain and can be evaluated at any point.

It is important to note that PINNs, as the name suggests, are Physics-Informed, not Physics-Enforced. This means the network is not (necessarily) strictly constrained to satisfy the governing equations but rather encouraged to do so.

A key insight is that, unlike traditional data-driven machine learning tasks that rely on labelled input-output pairs, PINNs do not require knowing the true solution at the training points. If derivatives of the surrogate can be accuratly computed, the residuals of the differential equation can be evaluated at any point on the domain. While real world data from the system might be sparse, noisy or even unavailable, sampling **collocation points** in the domain and evaluate the differential equation residuals comes ”only” at computational cost. With the constantly increasing

computational and efficiency of ultra parallelized hardware like GPUs and TPUs, this cost is rapidly declining. As an example, an entry-level gaming GPU like the NVIDIA RTX 4060, features 3072 CUDA cores and a theoretical performance of 15.11 TFLOPS at 32 bit floating point precision, that is, 15 trillion (10^{12}) floating point operations every second[1].

3.1.1 Formulation

Consider a general Boundary Value Problem (BVP) formulated as,

$$\begin{aligned} \text{PDE:} \quad & \mathcal{N}(u;\lambda,\mathbf{x}) = 0, \quad \mathbf{x} \in \Omega, \\ \text{BC:} \quad & \mathcal{B}(u;\lambda,\mathbf{x}) = 0, \quad \mathbf{x} \in \partial\Omega, \end{aligned} \tag{3.1}$$

where $\mathcal{N}$ is a (potentially non-linear) differential operator defining the PDE, $\mathcal{B}$ a (potentially differential and/or non-linear) operator expressing boundary (and initial) conditions. The BVP solution u takes input values $\mathbf{x}$ on a spaciotemporal domain Ω with boundary $\partial\Omega$. λ are the PDE parameters.

The goal of PINNs is to learn a surrogate solution $\hat{u}_\theta$, that satisfies (3.1) well. Ideally,

$$\sup_{\mathbf{x}\in\Omega} \|\mathcal{N}(\hat{u}_\theta;\lambda,\mathbf{x})\|_2^2 = \sup_{\mathbf{x}\in\partial\Omega} \|\mathcal{B}(\hat{u}_\theta;\lambda,\mathbf{x})\|_2^2 = 0, \tag{3.2}$$

implying that the surrogate is an exact solution to the BVP. In practice, this is rarely achievable, and the network can only aim to minimize these terms implying a weak enforcement. Directly minimizing (3.2) is not feasible either as NNs are trained using backpropagation which require the loss function to be differentiable. As a smooth approximation of (3.2), the loss function of PINNs is usually chosen as the mean squared residual error over a set of collocation points $[\mathbf{x}_c^{(i)}]_i$ sampled on the domain Ω and its boundary $\partial\Omega$. Formally, the loss terms are defined by (3.3),

$$\mathcal{L}_{\text{PDE}}(\theta;\lambda) = \frac{1}{N_c}\sum_{i=1}^{N_c} \left\|\mathcal{N}(\hat{u}_\theta;\lambda,\mathbf{x}_c^{(i)})\right\|_2^2, \quad \mathcal{L}_{\text{BC}}(\theta;\lambda) = \frac{1}{N_b}\sum_{i=1}^{N_b} \left\|\mathcal{B}(\hat{u}_\theta;\lambda,\mathbf{x}_b^{(i)})\right\|_2^2. \tag{3.3}$$

Imposing Dirichlét boundary conditions is straight forward, as it only requires evaluating $\hat{u}_\theta$ on a discrete set of input-output $(\mathbf{x}^{(i)}, u^{(i)})$ pairs at the boundary,

$$\mathcal{B}(u;\lambda,\mathbf{x}_b^{(i)}) = \hat{u}_\theta(\mathbf{x}_b^{(i)}) - u_b^{(i)}. \tag{3.4}$$

Imposing $\mathcal{N}$, and $\mathcal{B}$ if it is a differential operator, i.e. Neumann boundary conditions, requires justification that differential operators can meaningfully be applied

[1] https://www.techpowerup.com/gpu-specs/geforce-rtx-4060.c4107

to $\hat{u}_\theta$. The answer is backpropagation, as it allows exact and efficient computation of partial derivatives (including higher order derivatives [52]) of the network $\hat{u}_\theta$ with respect to its inputs, thus applying $\mathcal{N}$ and $\mathcal{B}$ to $\hat{u}_\theta$ is a well defined operation. More importantly, the partial derivatives are also functions of the network parameters θ, hence including them in the loss function will contribute to the training of the network. In this regard, one should be cautious with some activation functions, like ReLU, which have discontinuous first order derivatives and vanishing second order derivatives. These are generally not suitable for PINNs (see sections 3.1.2).

Note that differential conditions are imposed without requiring the true solution of the BVP to be known at the collocation points. As a result, the number of collocation points is primarily limited by computational resources. Section 3.1.3 elaborates on the topic of sampling collocation points.

In the general PINN formulation, boundary conditions are typically weakly enforced in the loss function. However, in some cases, it is possible to strongly enforce them, either fully or partially, through the construction of the surrogate. This is often referred to as an ansatz formulation. A demonstration of this is given in the 1D Burgers' equation example below.

So far, the formulation of PINNs is purely BVP-driven. If the problem is well constrained and the network is sufficiently expressive, PINNs may approximate the solution purely from the BVP, solving the forward problem. However, in many cases, BVPs are underdetermined, either by insufficient boundary conditions or unknown parameters. In such cases, data observed from the system can be used to further constrain the network. Thus an additional data loss (3.5) can be added,

$$\mathcal{L}_{\text{data}}(\theta) = \frac{1}{N_d} \sum_{i=1}^{N_d} \left\| \hat{u}_\theta(\mathbf{x}_d^{(i)}) - u_d^{(i)} \right\|_2^2, \tag{3.5}$$

where $(\mathbf{x}_d^{(i)}, u_d^{(i)})$ is a set of observed input-output pairs. The data loss term constrains the surrogate model and thereby the BVP during training. Consequently, the system parameters λ can be meaningfully tuned, provided the data sufficiently constrains the problem. This enables PINNs to solve inverse problems. Note that unknown model parameters are attached to the computational graph through the PDE loss, thus backpropagation will also compute gradients with respect to these parameters.

Collectively, PINNs are an optimization problem of tuning the parameters θ of a

neural network $\hat{u}_\theta$ as to minimize the total weighted loss defined by (3.6),

$$\mathcal{L}_{\text{PINN}}(\theta;\lambda) = \lambda_{\text{PDE}}\mathcal{L}_{\text{PDE}}(\theta;\lambda) + \lambda_{\text{BC}}\mathcal{L}_{\text{BC}}(\theta;\lambda) + \lambda_{\text{data}}\mathcal{L}_{\text{data}}(\theta), \tag{3.6}$$

where λ_{PDE}, λ_{BC} and λ_{data} are non-negative hyperparameters controlling the relative importance of the individual loss terms. A flow chart of the PINN training pipeline is given in figure 3.1.

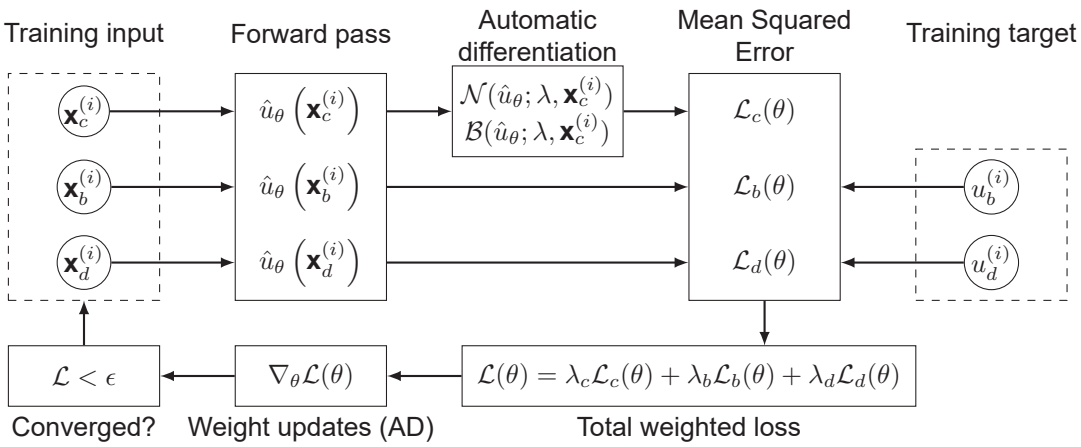

Figure 3.1: The PINN training pipeline. Starts at the training input and runs until convergence.

It is important to note that (3.6) can be extended to include additional loss terms. Apart from regularization terms, additional physical constraints such as conservation laws, symmetries, or other prior knowledge can be included. Utilizing all available information about the system may help guide the training process and improve the networks performance.

Example: 1D Burgers' Equation

To ellaborate on the notation introduced in section 3.1.1, consider a concrete example of the 1D viscous Burgers' equation,

$$\begin{aligned} \text{PDE:}\quad & u_t + uu_x - \nu u_{xx} = 0, && \mathbf{x} \in [0,1]\times[-1,1], \\ \text{BC:}\quad & u(t,-1) = u(t,1) = 0, && t \in [0,1], \\ & u(0,x) = -\sin(\pi x), && x \in [-1,1]. \end{aligned} \tag{3.7}$$

Where u is the velocity field, t is time, x is the spatial coordinate, and ν is the viscosity. Partial derivatives are indicated by subscripts, i.e. $u_t = \frac{\partial u}{\partial t}$ and $u_{xx} = \frac{\partial^2 u}{\partial x^2}$.

In terms of the formulation in (3.1), $\mathbf{x} = \{t, x\}$, $\lambda = \nu$ and the operators $\mathcal{N}(u;\lambda,\mathbf{x})$

and $\mathcal{B}(u;\lambda,\mathbf{x})$ are,

$$\mathcal{N}(u;\lambda,\mathbf{x}) = u_t + uu_x - \nu u_{xx},$$
$$\mathcal{B}(u;\lambda,\mathbf{x}) = \begin{cases} u(t,x), & \text{for } x \in \{-1,1\}, \\ u(t,x) + \sin(\pi x), & \text{for } t = 0. \end{cases}$$

The goal is to learn a surrogate solution $\hat{u}_\theta$ that satisfies the PDE and BCs in (3.7). For this problem, it is possible to construct an ansatz that strongly enforces the BCs. Let $\hat{v}_\theta$ be a neural network, then an ansatz trivially satisfying the BCs could be,

$$\hat{u}_\theta(t,x) = -\sin(\pi x) + (x^2-1)t\hat{v}_\theta(t,x). \tag{3.8}$$

It is important to emphasize that using an ansatz is not always appropriate, and in many cases not possible to construct at all. Embedding of strong enforcements in the surrogate can make it naturally overfit the boundary, restrict flexibility, and lead to poor generalization properties. Thus the use of an ansatz should be done with caution.

3.1.2 Activation functions in PINNs

As mentioned in section 2.3, choosing an appropriate activation function is crucial when designing a neural network. Considering the usecase of PINNs, the activation function should preferably be smooth and differentiable, as training requires higher order derivatives of the network.

Despite the popularity of ReLU and its variants in many deep learning applications, due to its cheap computational nature, its not widely used in physics-informed neural networks. PINNs usually require computation of higher order derivatives and the second derivative of ReLU vanishes, making it challenging to capture non-linear dynamics. Instead, infinitely smooth functions such as the sigmoid and hyperbolic tangent functions are often preferred in PINNs. A brief elaboration of selected activation functions in the context of PINNs can be found in appendix B.1.

3.1.3 Sampling

As mentioned in section 3.1.1, PINNs are constrained by the residuals of the differential operators $\mathcal{N}$ (and potentially $\mathcal{B}$) on sampled collocation points across the domain (and boundary). In a setup without observations, PINNs rely solely on the collocation points as training data. Even with observations available, these may be sparse or noisy, thus ensuring the quality of the collocation points is crucial for imposing the physical constraints.

Conceptually, it would make sense to have the distribution of collocation points reflect the relative importance of different domain regions, i.e. be dense around steep gradients or high curvature, e.g. around shocks and discontinuities. If such domain knowledge is present, it should be exploited. However, in a more general approach to sampling, one would not expect to have such information, thus sampling in an even manner is a more common approach.

For lower dimensional problems, the domain can feasibly be covered with a uniform grid of evenly spaced points. As the number of required points grows exponentially with the dimensionality of the problem, it is infeasible for higher dimensional problems. Alternativly one can sample uniform pseudo-random points. An inherent problem of this approach is that points are sampled independently and with equal probability, yielding a tendency to cluster some regions and leave gaps in others (see figure 3.2).

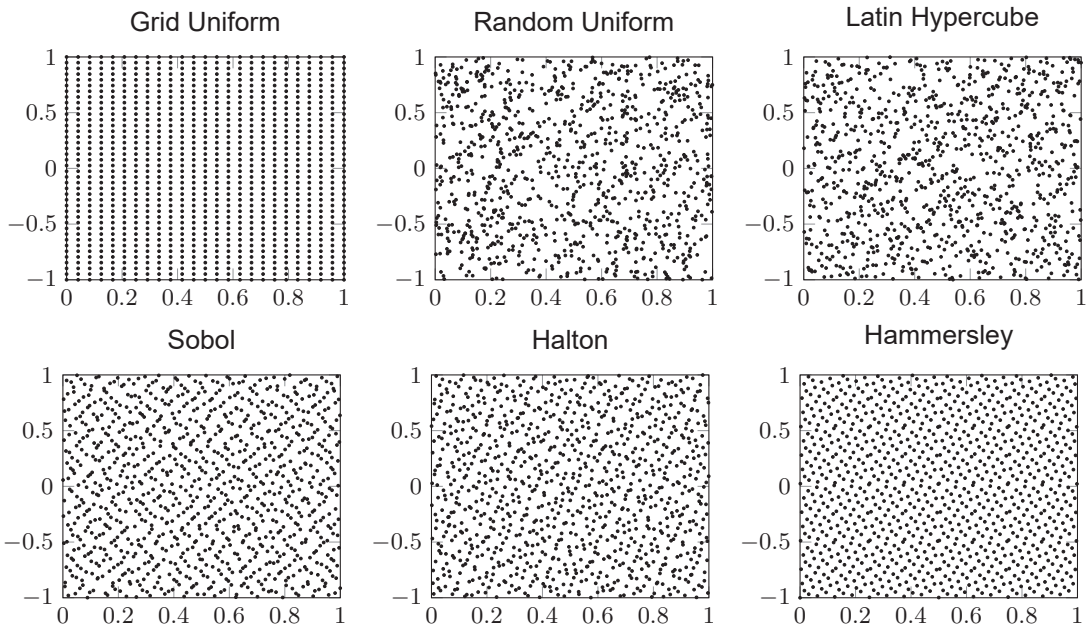

Figure 3.2: 1000 points sampled on the domain of the 1D Burgers' equation using different methods.

Quasi-random numbers are low descrepancy sequences that mitigate clustering by relaxing randomness for a more even coverage of the domain. Some common quasi-random number schemes are Sobol [53], Halton [54], and Hammerley sequences, all of which are shown in figure 3.2. The theoretical details behind these sequences and their generation are beyond the scope of this report. Fortunately, Wu et al. [16] has done a comprehensive study of the relative performance of these sampling methods in the context of PINNs, and found quasi-random sobol sequences to generally outperform grid, random uniform and latin hypercube sam-

pling methods.

Adaptive sampling

While collocation points are initially sampled in an even manner, resampling parts of or even the entire set of collocation points during training can be useful. This is due the spectral bias of the problem formulation. Spectral bias refers to the fact that neural networks tend to fit lower frequency functions better higher frequency ones. Sampling densly around high frequency regions can help inform the network to better capture those features. Generally, it is not known a priori where these regions are. An obvious solution is to simply increase the number of collocation points across the entire domain. However, this is a highly inefficient use of collocation points and comes at the cost of significantly increased computational cost, especially for high dimensional problems.

Adaptive sampling techniques aim to leverage information acquired during training to strategically refine the collocation points, by targeting regions in which some predefined criterion is met. Two (three) such methods are Residual-based Adaptive Refinement sampling (RAR) [55] and Residual-based Adaptive Distribution sampling (RAD and RAR-D) [16], both of which use the PDE residuals as the criterion for refinement.

RAR is a greedy algorithm that iteratively refines the set of collocation points by periodicly adding new points in regions with high PDE residuals. Practically, this is done by evenly sample a set of candidate points across the domain, then greedily add the N points with largest residuals to the set of collocation points.

RAD resamples all collocation points based on the distribution of the PDE residuals. The algorithm works by evenly sampling a set of candidate points across the domain, then compute the PDE residuals and sample a new set of collocation points without replacement from the distribution defined by $p(\mathbf{x}) \propto \frac{\varepsilon^k(\mathbf{x})}{\mathbb{E}[\varepsilon^k(\mathbf{x})]} + c$, where ε is the PDE residuals and $k, c \geq 0$ are tunable hyperparameters.

RAR-D is a hybrid of RAR and RAD, adding new points sampled according to the RAD scheme.

The authors of [16] found that RAD method using a quasi-random sampling scheme consistently outperformed non-adaptive sampling methods as well as RAR and RAR-D. They also discovered that $k = 2$ and $c = 0$ usually yields good results. Resampling all collocation points radically change the loss landscape, which can cause optimizers such as L-BFGS to crash or diverge. The authors propose a training strategy employing more stable optimizers such as Adam or AdamW for

a set amount of epochs following a resampling. This helps stabilize the training process before switching back to L-BFGS.

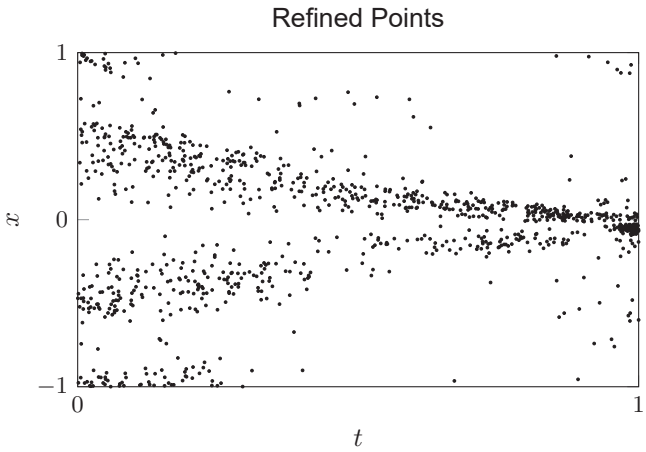

Figure 3.3: 1000 points sampled on the domain of the 1D Burgers' equation using the RAD adaptive sampling method with $k = 2$ and $c = 0$. The resampling was done after 1000 epochs of training a FNN with 8 hidden layers of 20 neurons using AdamW optimizer.

3.1.4 Adaptive loss weighting

Training a PINN invovles minimizing the total loss function (3.6), which contains several individual loss terms. Thus PINNs define a multiobjective optimization problem, in which the relative importance of the respective loss terms are determined by the hyperparameters λ_{PDE}, λ_{BC} and λ_{data}.

The optimal set of values for these hyperparameters is problem specific, and is likely to change during training. Rather than using a fixed loss weighting, i.e. manual tuning, weights can be adaptively adjusted through the epochs.

SoftAdapt [13] is a dynamic loss weighting scheme based on the softmax activation function (see equation (3.9)), that prioritize loss terms based on how much they decrease. Formally, the loss weights are defined as,

$$\lambda_i = \frac{e^{\beta(s_i - \max_k(s_k))}}{\sum_{j=1}^{m} e^{\beta(s_j - \max_k(s_k))} + \epsilon}, \tag{3.9}$$

where s_i is the signed change of loss term i in successive epochs, β is a non-negative hyperparameter controlling the relative scaling of weights, i.e. lower values yield more uniform weights with higher values prioritizing the most decreasing loss term. ϵ is a small positive constant for numerical stability. This scheme also has a variant including the current value of the loss term in (3.9) to normalize the weights with respect to the current loss values, known as loss weighted SoftAdapt

(3.10),

$$\lambda_i = \frac{\mathcal{L}_i e^{\beta(s_i - \max_k(s_k))}}{\sum_{j=1}^{m} \mathcal{L}_j e^{\beta(s_j - \max_k(s_k))} + \epsilon}, \quad i = 1, \ldots, m. \tag{3.10}$$

3.1.5 Regularization

Overfitting is a classic problem in machine learning, and neural networks are no exception. It often occurs if the network is too complex and begins to memorize the training data rather than learning underlying generalizable patterns. Weights of large magnitude can be an indication that overfitting is present in the model. Expression (2.15) shows that gradients through a NN layer are directly dependent on the weights, thus large weights can destabilize the training process, making gradients, and thereby weight updates, explode or vanish.

Regularization is a technique constraining the magnitude of the weights, directly imposed in the loss function. The most common regularization schemes are L_1 and L_2 regularization, penalizing the weights based on the respective norms. Using L_2 as an example, the regularization loss is,

$$\mathcal{L}_{\text{reg}}(\theta) = \lambda_{\text{reg}} \|\theta\|_2^2, \tag{3.11}$$

where λ_{reg} is a non-negative hyperparameter controlling the regularization strength.

L_2 regularization is also known as weight decay, as it effectively seek to decay the weights towards zero. Many machine learning frameworks, like PyTorch [23], incorporate weight decay in the optimizers and automatically applies it to the weights during training.

This issue can also be addressed with gradient clipping, i.e., thresholding the gradients during backpropagation.

3.1.6 Learning rate scheduling

The learning rate (lr) is a hyperparameter controlling the step size of the optimizer during weight updates. High lr accelerates convergence through larger steps, at the risk of overshooting the minimum. Contrary, low lr provide stability, but may require more epochs to converge or get stuck in local minima. Adaptive tuning between these tradeoffs can be employed during training to speed up convergence while retaining stability. This is the concept of learning rate scheduling. Common strategies include periodical reduction of the lr by a predetermined factor every set number of epochs, or when the loss stops decreasing. More sofisticated approaches adjust the lr based on the loss, gradients or other metrics, allowing more responsive tuning of the lr.

3.2 Generalizing PINN (UPINN)

Consider the BVP given by (3.1). Split the operators $\mathcal{N}$ and $\mathcal{B}$ into two parts, a known part $\mathcal{N}_K$ and $\mathcal{B}_K$ as well as an unknown part $\mathcal{F}$ and $\mathcal{G}$, such that,

$$\mathcal{N} = \mathcal{N}_K + \mathcal{F}, \quad \mathcal{B} = \mathcal{B}_K + \mathcal{G}. \tag{3.12}$$

The idea of UPINN [8] is to approximate the unknown parts of the PDE $\mathcal{F}$ and boundary $\mathcal{G}$ using neural networks $\hat{F}_{\theta_F}$ and $\hat{G}_{\theta_G}$ while simultaneously learning the solution u with a neural network $\hat{u}_{\theta_u}$. Apart from (3.12) nothing else change in the formulation of the BVP. Note that the weights θ_F and θ_G of the residual networks are included in the PDE loss through the collocation points.

θ_F and θ_G are trained to approximate unknown operators, that are potentially non-linear and differential. Podina et al. [8] suggest imposing an inductive bias on these networks by passing them inputs thought to be part of the unknown operators, such as partial derivatives of $\hat{u}_{\theta_u}$ evaluated at the collocation points, as these can be accuratly computed using backpropagation. Note that this methodology connects the computational graphs of the residual networks with the solution network, making weight updates interdependent. As data is only evaluated by the solution network, this connection is crucial if the residual networks are to learn directly from the data as well.

UPINNs generalize the idea of PINNs by extending the frameworks flexibility. This generalization does not only allow solving BVPs, but facilitates integration of data in combination with known physics to approximate a functional representation of the hidden parts of the system. This is particularly useful in cases where underlying physics are not fully understood and relevant data is available.

While not guaranteed or thouroughly investigated, some of the results and techniques outlined in section 3.1, developed and studied for PINNs, may also be relevant for UPINNs.

UPINNs have yet to be fully explored. This project aims to contribute to the understanding of this novel extension of the PINN framework, by investigating its abilities regarding approximation of the hidden parts of a system. While the formulation (3.12) assigns neural network representations to both the PDE and boundary conditions, this project will focus on problems related to the PDE part, i.e. $\mathcal{F}$.

4 Symbolic Regression

Approximating a functional representation of a system is a powerful tool. Translating it into a symbolic expressions is even more valuable. Symbolic regression is the data-driven process of obtaining governing symbolic expressions from observed data.

In the context of UPINNs, the unknown parts of the system are approximated by neural networks. If these approximations are sufficiently accurate, they can be used to densely sample the unknown dynamics within the domain. From this, symbolic regression methods may be used to find the underlying symbolic expressions.

This project utilize the symbolic regression frameworks of SINDy for ODEs and PDE find for PDEs. Thus, the following section provide a brief introduction to these methods.

4.1 SINDy and PDE find

Sparse Identification of Nonlinear Dynamics (SINDy) [9], is a data-driven technique for identifying sparse representations of nonlinear dynamical systems governed by ODEs. The philosophy behind SINDy is that the model should be as simple as to accurately describe the system, but no simpler.

Consider a time dependent system of ordinary differential equations (ODEs) of the form (4.1),

$$\frac{d\mathbf{x}}{dt} = \mathcal{F}(\mathbf{x}), \tag{4.1}$$

where the function $\mathcal{F}$ is unknown, but generally assumed to be sparse, i.e., has few non-zero terms. Suppose that the state vector $\mathbf{x}$ is observed at a set of discrete time points,

$$\mathbf{X} = \begin{bmatrix} \mathbf{x}^T(t_1) \\ \mathbf{x}^T(t_2) \\ \vdots \\ \mathbf{x}^T(t_N) \end{bmatrix}, \quad \dot{\mathbf{X}} = \begin{bmatrix} \dot{\mathbf{x}}^T(t_1) \\ \dot{\mathbf{x}}^T(t_2) \\ \vdots \\ \dot{\mathbf{x}}^T(t_N) \end{bmatrix}. \tag{4.2}$$

If $\Theta(\mathbf{X})$ is a library of candidate functions evaluated at the observed states, the goal is then to determine a sparse set of coefficients Ξ, such that,

$$\dot{\mathbf{X}} \approx \Theta(\mathbf{X})\Xi. \tag{4.3}$$

This approach allows directly discovering governing equations from data. In the context of UPINNs, $\mathbf{X}$ can be the outputs of the solution network evaluated on a dense set of points, and $\dot{\mathbf{X}}$ the corresponding outputs of the residual network, which approximates the unknown part of the ODE. Applying SINDy in this setup, symbolic representations of the unknown dynamics can be extracted.

Following the success of SINDy, PDE functional identification of nonlinear dynamics (PDE find) [10] was developed to generalize this methodology to partial diffential equations. Introduced by the authors behind SINDy, PDE find employs the same sparse regression framework, but applies it to spaciotemporal domains of PDEs. Analagous to SINDy, PDE find assumes a PDE of the form,

$$\frac{\partial u}{\partial t} = \mathcal{F}(u, u_x, u_{xx}, \ldots), \tag{4.4}$$

where $\mathcal{F}$ is a sparse function on the spaciotemporal derivatives of u. Using a library of candidate functions and their derivatives, PDE find identifies a sparse representation of $\mathcal{F}$.

PDE find can be applied to UPINNs in a similar manner as SINDy.

SINDy and PDE find were developed for discovery of governing equations exclusively from data, and in many applications, they rely on numeric differentiation (finite differences method) to compute the derivatives using only observed states. In the context of UPINNs, derivatives and partial derivatives are computed using backpropagation, providing exact and efficient computation of the derivatives, which also helps with stability and accuracy of the discovered equations.

PDE find is a natural extension of SINDy, and is also part of most implementations of SINDy, including the Python library PySINDy which is used in this project.

Part II

Experiments and Results

5 Experimental Preliminaries

Having established the theoretical foundation of neural networks, PINNs and UP-INNs, this next part of the project focus on applying these methods to a variety of problems, and study their performance and behavior. As mentioned in the introduction, this project considers the 4 PIML problems identified in [1]. Namely the forward problem of solving differential equations, the inverse problem of identifying unknown parameters of a known model, the hidden discovery problem of detecting and learning residuals of system dynamics, and finally the interpretability problem of uncovering governing equations.

This short preliminary chapter for the experimental part provides an overview of the computational framework, data, evaluation metrics, and model selection in symbolic regression.

5.1 Computational framework

All experiments conducted are implemented using the Python programming language. For training and evaluation of neural network models, the Python implementation of the Torch framework [23], PyTorch, is used. Specifically, PyTorch with CUDA, to leverage GPU accelerated training. For reproducibility, the random seed of the torch engine is set at the beginning of each experiment as 42.

For symbolic regression experiments, the Python implementations of SINDy and PDE-find [9, 10], contained in the PySINDy package, are used. This library builds on the scikit-learn and sympy frameworks, providing a range of tools for model selection and equation discovery. As all considered problems are time-dependent, time series cross validation is used for model selection (hyperparameter tuning), e.g. using TimeSeriesSplit from scikit-learn.

5.2 Data

The datasets used in the experiments come from various sources. The ODE systems of Lotka-Volterra and SIR models are generated using the odeint solver from the SciPy library. For the PDE systems of Burgers' and Navier-Stokes equations, the datasets from the original PINN paper [5] are used.

All of the above mentioned datasets are generated using simulation where the true models are known, which is necessary for evaluating their performance. To

emulate real-world scenarios, depending on the experiment, noise can be added to the data. Noise is sampled from a standard normal distribution and rescaled according to equation (5.1),

$$\text{noise} = \epsilon \cdot \max_{\mathbf{x}\in\Omega} |u(\mathbf{x})| \cdot \mathcal{N}(0,1), \tag{5.1}$$

where ϵ is the noise level, $\max_{\mathbf{x}\in\Omega} |u(\mathbf{x})|$ is the magnitude of the data and $\mathcal{N}(0,1)$ is the standard normal distribution. This is also the used noise scheme in the UPINN paper [8].

5.3 Evaluation metric

The L^2 relative error is chosen as the primary metric for evaluating model performance. Given a set of evaluation points $\{\mathbf{x}^{(i)}, u^{(i)}\}_{i=1}^{N}$ and a solution network $\hat{u}_{\theta_u}$, the L^2 relative error is defined by equation (5.2),

$$L^2 = \sqrt{\frac{\sum_i (\hat{u}_{\theta_u}(\mathbf{x}^{(i)}) - u^{(i)})^2}{\sum_i (u^{(i)})^2}}. \tag{5.2}$$

This metric is also known as the **relative root mean squared error** (RRMSE). Contrary to the mean squared error (MSE), which is an absolute measure of error, RRMSE is a relative measure, making it less sensitive to the scale of the data. This also makes it more suitable for comparing models across different datasets and problems.

For the UPINN residual surrogates in chapter 7, the RRMSE is calculated between expected and predicted residuals. The **expected residual** is defined by applying the true unknown operator $\mathcal{F}$ to the solution network $\hat{u}_{\theta_u}$, see equation (5.3). Generally, the goal of UPINNs is to model the unknown operator $\mathcal{F}$ and not the residuals themselves. The expected residuals are computed given the current state of the solution network, whereas the true residuals are computed from the true solution. Thus, the expected residuals is a more meaningful metric for performance of $\hat{F}_{\theta_F}$, much less dependent on the accuracy of $\hat{u}_{\theta_u}$.

$$\hat{F}_{\text{exp}}(\mathbf{x}) = \mathcal{F}(\hat{u}_{\theta_u}; \lambda, \mathbf{x}) \tag{5.3}$$

5.4 Model selection in symbolic regression

Regarding the SINDy and PDE-find frameworks, the equation discovery problem is formulated in terms of the notation used in section 4.1 [9] as equation (5.4),

$$\dot{\mathbf{X}} = \hat{F}_{\theta_F}, \quad \mathbf{X} = \hat{u}_{\theta_u}. \tag{5.4}$$

Symbolic regression is performed using the SR3 [56] optimizer with L^1 norm thresholding. The threshold value is determined automatically using 10-fold time series cross validation on a grid of candidate values.

In forecasting problems, it is crucial to split into a training and test set seperated at a specific time point, to prevent information leak and ensure the model generalizes well to unseen data. This is less important when training UPINNs for equation discovery with symbolic regression, as the goal is to accurately model the residuals within the training data rather than predicting future time steps. Nevertheless, it remains crucial when performing symbolic regression (i.e. time series cross validation), to ensure that the found equations fit the approximated residuals well.

6 Considered Problems and PINNs

Before exploring the capabilities of UPINNs, this chapter presents the 4 problems considered in this project, and demonstrates the use of regular PINNs to solve these problems. The problems considered are the Lotka-Volterra predator-prey model, an SIR epidemic model, Burgers' equation, and Navier-Stokes equations for incompressible fluid flow. These are common benchmark problems in the literature of PINNs and SciML [5, 8].

This simoultanously serves as an initial sanity check of the experimental part of this project, ensuring that the framework is correctly implemented and functioning as expected. As the problems are already well studied in the context of PINNs in the literature [5], in depth analysis is not provided in this thesis.

6.1 Lotka-Volterra predator-prey model (ODE system)

The Lotka-Volterra predator-prey model is a classic population dynamics model, describing population changes within an ecosystem of two species, e.g. rabbits and foxes. The model consists of two coupled first-order non-linear ODEs given by equation (6.1),

$$\begin{aligned} \frac{dx}{dt} &= \alpha x - \beta xy, \\ \frac{dy}{dt} &= -\delta y + \gamma xy, \\ x, y &\geq 0, \end{aligned} \tag{6.1}$$

where x and y are the dimensionless predator and prey populations, respectively, given relative to their initial sizes. The parameters α and δ represent the growth/decay rates of the prey and predator populations. β and γ describe the effects of interaction between species on the change in population size, i.e. prey is consumed by predators and predators are fed by prey.

The Lotka-Volterra system is known to exhibit oscillatory behavior, with rising prey populations facilitating rising predator populations, and vice versa. Solutions of the system are continuous, thus no shocks or discontinuities are present.

Provided parameters and initial population sizes, the model governs development of the populations over time. In the following, a specific problem using $\alpha = 2/3$, $\beta = 4/3$, $\delta = 1$, and $\gamma = 1$, with initial condition $x(0) = [1, 1]$, and domain $t \in [0, 20]$

is considered. This corrosponds to just over 2 cycles of the system.

PINN - forward problem

Solving the Lotka-Volterra system with a PINN can be challenging in a data-free setting. Constrained by only a single initial point, all information must be extracted through the system dynamics from this point. Directly sampling collocation points across the entire domain and then train is likely to get the network stuck in a local minima, that is the zero solution, which trivially satisfies the system 6.1 (except for the initial point).

To mitigate this, the domain can be partitioned into the 20 time units, beginning training at the first and then progressivly add collocation points from proceeding time intervals as the loss decreases. This way, information from the initial point is propagated through time in a more stable and controlled manner. The trained model is shown in figure 6.1. This serves merely as a demonstration that PINNs can solve the forward problem without data, and no optimization of the training process has been done.

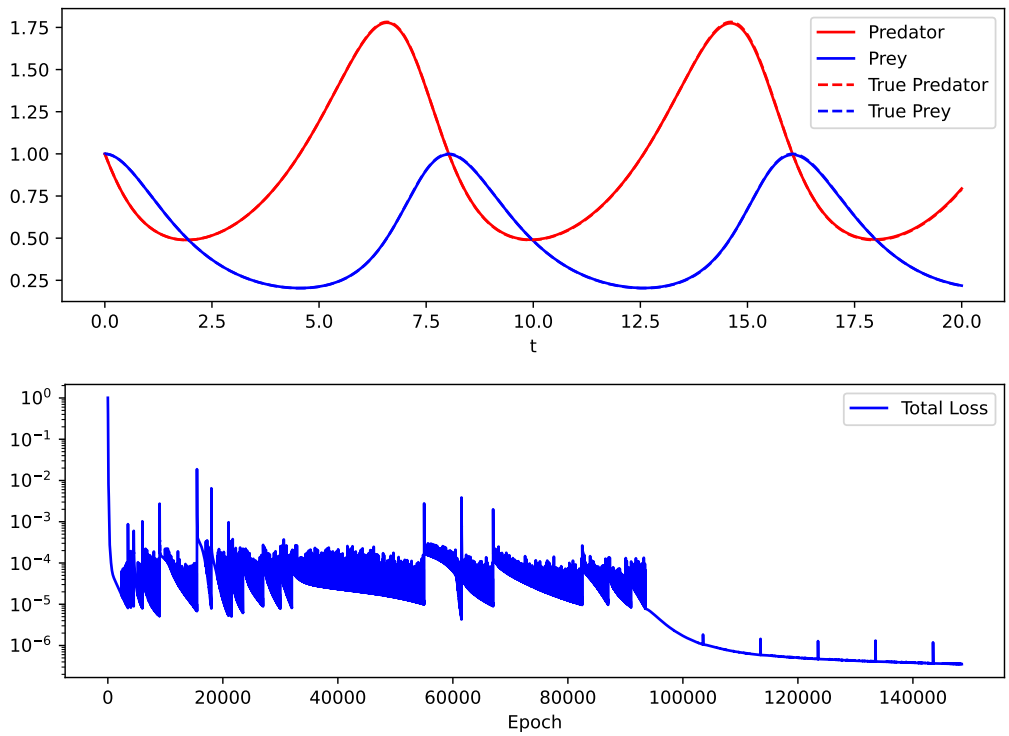

Figure 6.1: Lotka-Volterra: FNN with 4 hidden layers of 20 units with Tanh activation function and SiLU output activation function. Training is done using AdamW optimizer with learning rate 1e-3 and weight decay 1e-4 until all time units are included in the training set. The learning rate is then reduced to 1e-5 and training continues until convergence. Solution RRMSE is 2.85e-3.

PINN - inverse problem

To demonstrate solving the inverse problem, the same architecture is used as for the forward problem. However, all 4 parameters, α, β, δ, and γ are treated as trainable. The data consists of only 8 noise-free equispaced data points in the domain.Including data points simoultaneously mitigate the zero solution problem, as the loss function is now penalized for not fitting the data. As such, the model is trained on the entire domain from the beginning. Figure 6.2 shows that all 4 parameters converge to the true values, while the model accurately predicts the solution.

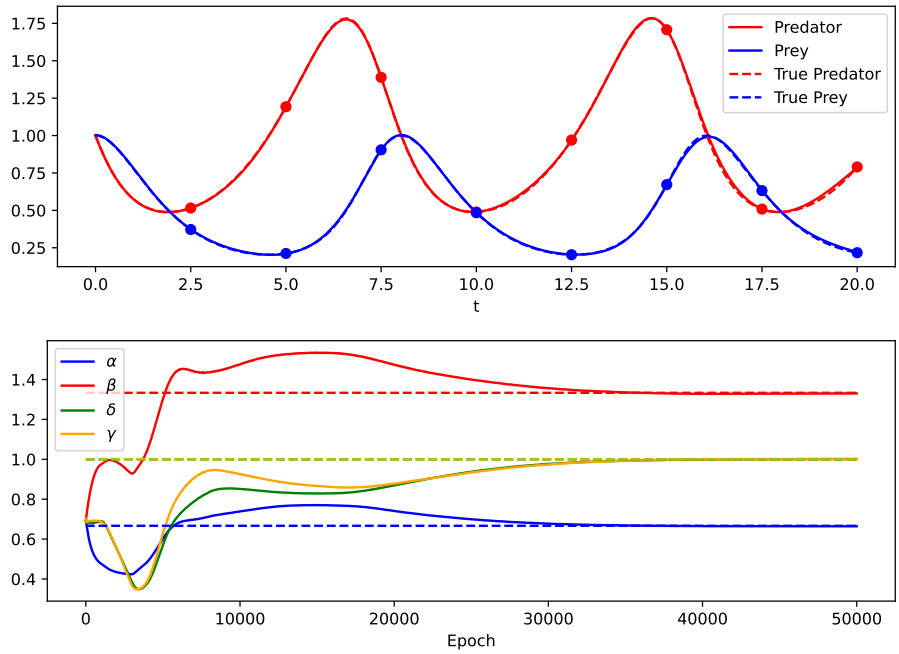

Figure 6.2: Lotka-Volterra: Trainable parameters. FNN with 4 hidden layers of 20 units with Tanh activation function and SiLU output activation function. Training is done using AdamW optimizer with learning rate 1e-3 and weight decay 1e-4 until convergencefor 50000 epochs. Solution RRMSE is 4.25e-3.

6.2 Epidemiological compartment model (ODE system)

The SIR model is a compartment model describing the spread of disease within a population. In its most basic form, it divides the population into 3 compartments: Susceptibles (S), Infected (I), and Recovered/Removed (R). The model describes the flow of individuals between these compartments governed by ordinary differential equations (ODEs). While the compartments may be given in units of absolute individuals, it is often normalized to fractions of the total population, which will also be the case in this project. Subsequently, the total population is 1.

This section considers the simplest form of the SIR model, illustrated by figure

6.3, where the population is constant, i.e. no births or deaths, and contracting the disease makes one immune. This system consists of 3 coupled first-order non-linear ODEs given by,

$$\begin{aligned} \frac{dS}{dt} &= -\beta SI, \\ \frac{dI}{dt} &= \beta SI - \gamma I, \\ \frac{dR}{dt} &= \gamma I. \\ S + I + R &= 1, \\ S, I, R &\in [0, 1]. \end{aligned} \tag{6.2}$$

The parameter β represents the rate of infection due to interaction between susceptible and infected individuals, while γ is the recovery rate of the infected individuals.

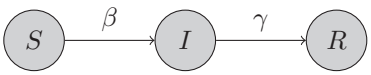

Figure 6.3: Flowchart of the basic SIR model.

Provided an initial state of each compartment, the model describes the progressive flow of individuals towards the removed state. An inherent assumption of this model is that individuals only move forward, thus contracting the disease makes one immune and reinfection is not possible. Section 7.2 considers a variation of the SIR model where reinfection is possible.

PINN - forward problem

Like the Lotka-Volterra system, the SIR model can be challenging to solve with a PINN when data is not present. Thus the approach of partitioning the domain into time units and progressivly extrapolate from the initial point is used. The specific problem considered is $\beta = 0.3$ and $\gamma = 1/21$ with initial conditions $S(0) = 0.99$, $I(0) = 0.01$, and $R(0) = 0$ in the domain $t \in [0, 60]$.

There are 2 constraints on the compartment populations; the total population is constant and the fraction of each compartment is within $[0, 1]$. The latter can be strongly enforced with a Sigmoid activation function on the output or weakly enforced with a penalty loss term. Both approaches require a loss penalty on the total population. Another approach is to construct an ansatz that trivially satisfies both constraints. This can be achieved either by normalizing the output layer with its sum or using a softmax activation function (see equation (2.3)). All of these approaches were tested, see table 6.1, and the ansatz formulations were found

to be the most stable and accurate with sum normalization being slightly better. The solution using sum normalization is visualized in figure 6.4.

Method	Softplus	Sigmoid	Ansatz norm	Ansatz softmax
RRMSE	3.11e-1	4.71e-3	2.06e-3	2.82e-3
Epochs	>150000	108500	25000	25000

Table 6.1: SIR: Comparison of different ways to enforce the compartment constraints. Using a FNN with 3 layers of 16 units with Tanh activation function. Training is done using AdamW optimizer with learning rate 1e-3 and weight decay 1e-4 in an incremental manner with 1000 grid collocation points.

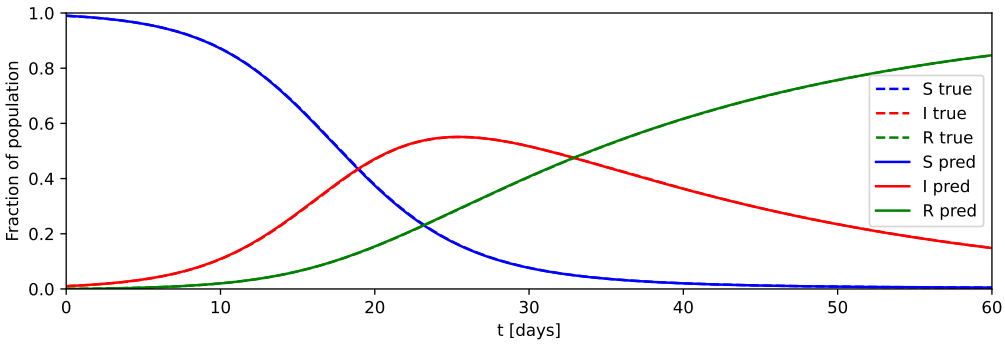

Figure 6.4: SIR: PINN predicted solution of basic SIR model using an ansatz with sum normalization. RRMSE is 2.06e-3.

PINN - inverse problem

Estimating the parameters β and γ of the SIR model in an early stage of the epidemic, enables forecasting of the spread of the disease. Thus, the inverse problem is solved using 10 datapoints sampled from the first 10 time units of the system depicted in figure 6.4 to represent the early stage. The same architecture is used as in the forward problem, with the parameters β and γ being trainable. The model is then trained for 30000 epochs with Adam optimizer and a learning rate of 1e-3, using 1000 grid collocation points on the domain $t \in [0, 10]$. The parameters β and γ are then fixed in place and the model is trained for another 20000 epochs with 1000 grid collocation points on the entire domain $t \in [0, 60]$.

As seen in figure 6.5, the model converges to the true values of $\beta = 0.3$ and $\gamma = 1/21 \approx 0.0476$. Further, it is seen that the model accurately predicts the solution beyond the training data, with a solution RRMSE that is lower than the forward problem.

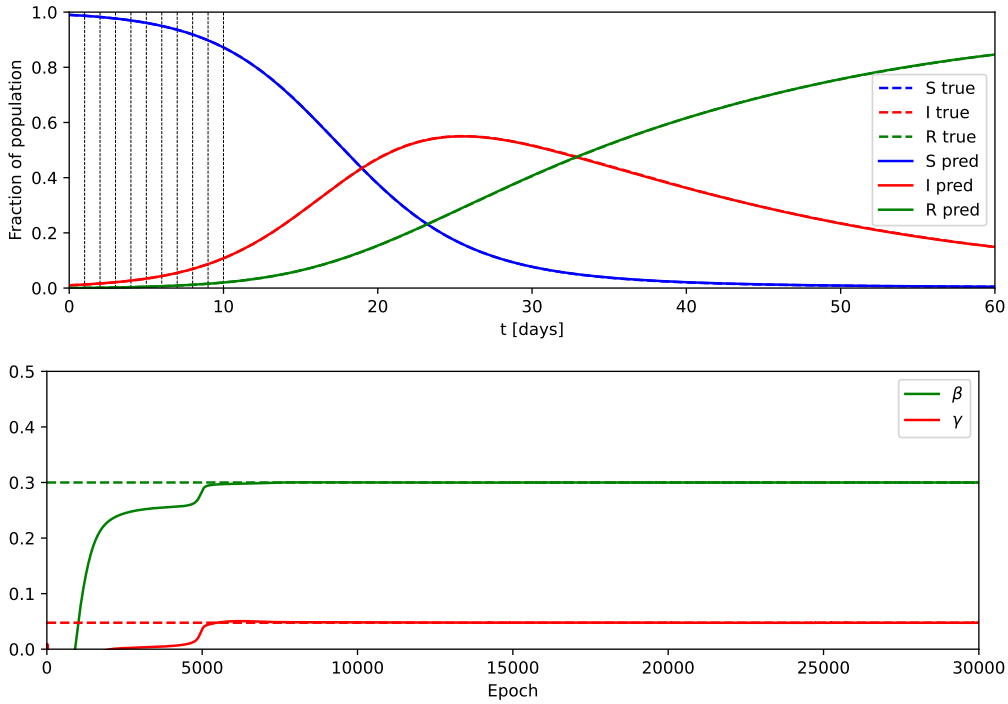

Figure 6.5: SIR: Solving the inverse problem of estimating β and γ in the SIR model. **Top**: Predicted solution of the SIR model. Vertical lines indicate training data points. **Bottom**: Convergence of the estimated parameters β and γ. Solution RRMSE is 1.52e-3.

6.3 Burgers' equation (PDE)

As introduced in section 3.1.1, Burgers' equation (or viscous Burgers' equation) is a fundamental convection-diffusion PDE occuring in many areas of physics, including fluid dynamics, acoustics and gas dynamics. Burgers' equation is particularly interesting as the right initial condition (wave profile) can lead to areas with steep gradients and, in the inviscid case, form shocks.

The specific instance of Burgers' equation considered is the viscous form, given in (3.7), with viscosity $\nu = (0.01\pi)^{-1}\frac{\mathrm{m}^2}{\mathrm{s}}$. The true solution is visualized in figure 6.6. The full dataset consists of 100 equally spaced time samples in the interval $t \in [0, 1]$, each with 256 equally spaced spatial samples on the interval $x \in [-1, 1]$, for a total of 25600 samples.

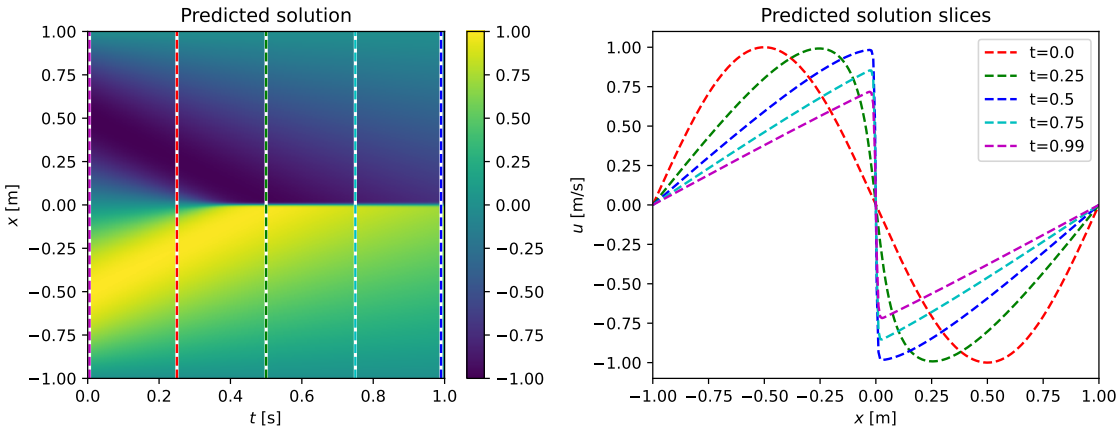

Figure 6.6: Burgers': True solution of Burgers' equation.

PINN - forward problem

Using a set of 100 equally spaced points on each of the 3 parts of the boundary, and 10000 sobol sampled collocation points in the domain (see figure 7.10). The architecture is a FNN with 8 hidden layers of 20 neurons each using Tanh activation functions. The model is trained for 2000 epochs with AdamW optimizer and a learning rate of 1e-3 followed by 2000 epochs with LBFGS optimizer with learning rate 1.

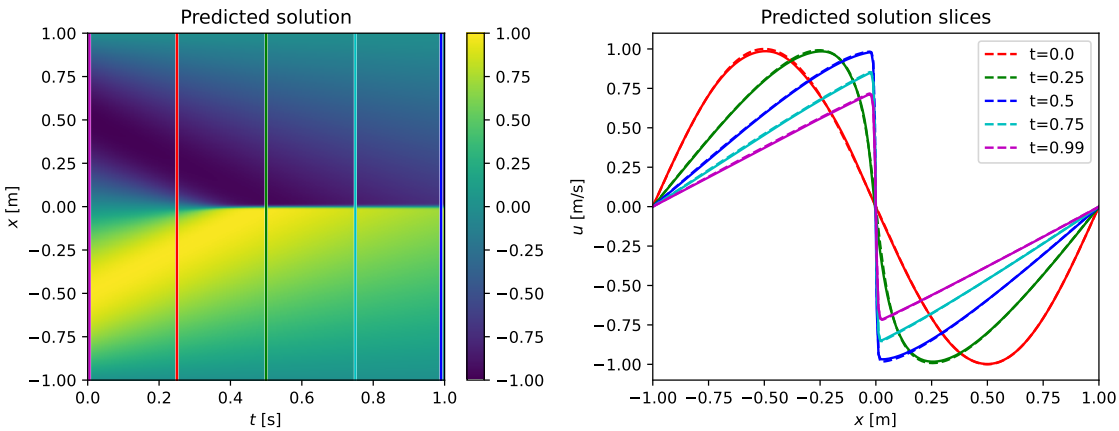

Figure 6.7: Burgers': Predicted solution of Burgers' equation. Dashed lines indicate the true solution. The solution is predicted with an RRMSE of 1.81e-2.

PINN - inverse problem

For solving the inverse problem, the same set of training points is used as in the forward problem, with the addition of 256 equally spaced data points (noise free) sampled from the true solution at $t = 0.5$ (a sparse dataset). Furthermore, the viscosity parameter ν is set as a trainable parameter. The model is trained using 30000 epochs with AdamW optimizer and a learning rate of 1e-4.

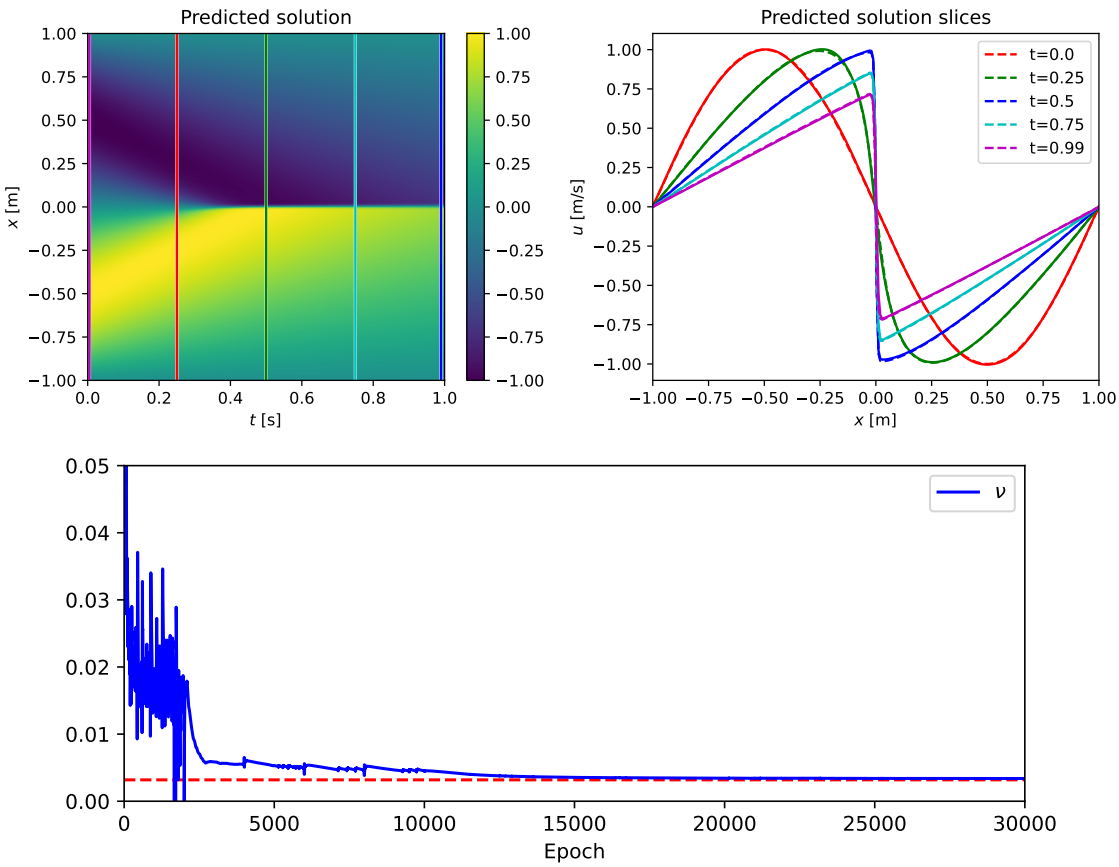

Figure 6.8: Burgers': **Top**: Predicted solution of Burgers' equation for the inverse problem. Black dashed lines indicate the true solution. The solution is predicted with an RRMSE of 1.88e-2. **Bottom**: Convergence of the estimated viscosity parameter ν.

The parameter ν is estimated as $\hat{\nu} = 0.0034\frac{\mathrm{m}^2}{\mathrm{s}}$ against the true value of $\nu = 0.0031\frac{\mathrm{m}^2}{\mathrm{s}}$. ν governs the diffusion term of the equation, which is responsible for smoothing out the solution, preventing the formation of shocks. Thus it is highly dependent on the resolution of the data at shocklike interface, where gradients are steep.

6.4 Navier-Stokes equations (PDE system)

The Navier-Stokes equations are a set of non-linear PDEs that describe the motion of fluid flow. This project considers the 2D incompressible fluid flow equations given by,

$$\begin{aligned} u_t + \lambda_1(uu_x + vu_y) + p_x - \lambda_2(u_{xx} + u_{yy}) &= 0, \\ v_t + \lambda_1(uv_x + vv_y) + p_y - \lambda_2(v_{xx} + v_{yy}) &= 0, \\ u_x + v_y &= 0, \end{aligned} \tag{6.3}$$

where (u, v) is the velocity field, p is the pressure, and the parameters λ_1 and λ_2 are the convective scaling factor and kinematic viscosity respectively. The last equation is the incompressibility constraint, expressing divergence-free flow. Note, for the purposes of this project, there are no boundary conditions, implying the system is exclusively constrained by the data and system dynamics.

The solution consists of the velocity field (u, v) and the pressure p. The pressure is only constrained by its gradients, thus p can only be determined up to an additive integration constant if not constrained by absolute measurements. The specific instance of (6.3) considered, is the example from [5], which describe the flow around a cylinder, specifically the turbulent wake flow behind it. The full dataset consists of 1000000 simulated measurements of u, v and p at steady state, that is, the vortex shedding has stabilized. The domain is $(x, y, t) \in [1, 8] \times [-2, 2] \times [0, 20]$ and the specific system is (6.3) with $\lambda_1 = 1$ and $\lambda_2 = 0.01$.

With inspiration from [5], the incompressibility constraint is strongly enforced with an ansatz, by letting the surrogate estimate (ψ, p) with $u = \psi_y$ and $v = -\psi_x$ rather than (u, v, p). This ensures that any approximation is trivially divergence-free as,

$$u_x + v_y = \psi_{yx} - \psi_{xy} = 0.$$

To simplify visualization, u and v are illustrated by the vorticity field $\omega = v_x - u_y$ rather than seperately.

Solving with PINN

The system (6.3) has 2 parameters, λ_1 and λ_2. The inverse problem is to use a subset of the full dataset to estimate these parameters with a PINN. A set of 10000 sobol sampled collocation points is used, along a random subsample of 2000 data points of the full dataset,

$$\{x^{(i)}, y^{(i)}, t^{(i)}, u^{(i)}, v^{(i)}\}_{i=1}^{2000}.$$

The model is trained using an FNN with 8 hidden layers of 20 units with Tanh activation function. AdamW optimizer with learning rate 1e-3 for 1000 epochs followed by L-BFGS with learning rate 1 until convergence.

The result is shown in figure 6.9. The estimated parameters are $\hat{\lambda}_1 = 0.998$ and $\hat{\lambda}_2 = 0.0107$ against the true values of $\lambda_1 = 1$ and $\lambda_2 = 0.01$. The data-free forward problem is not considered for this system.

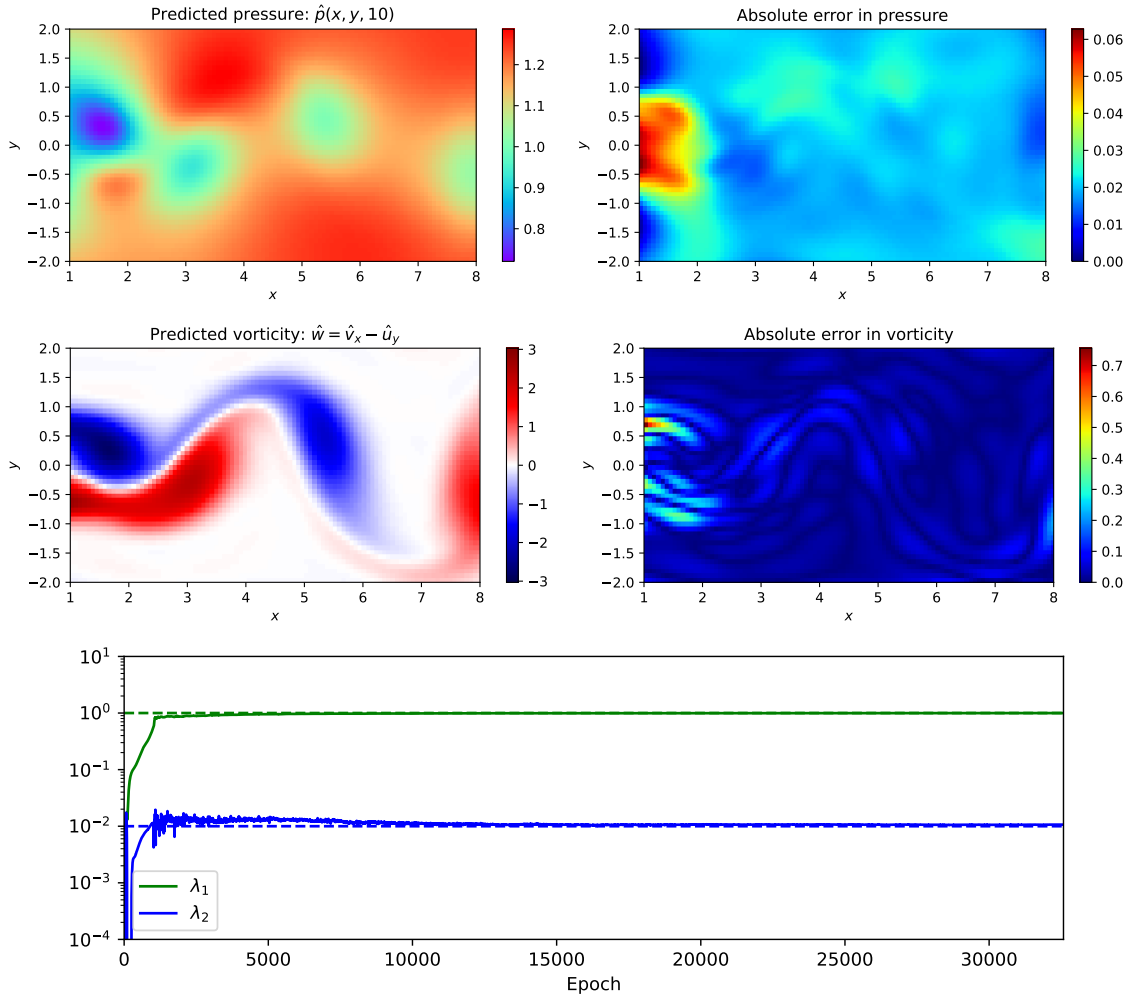

Figure 6.9: Navier Stokes: Trainable parameters. Solution at $t = 10$. **Top**: Predicted pressure. **Middel**: Predicted vorticity. **Bottom**: Convergence of parameters. Note: Pressure error is compensated for the additive constant. Solution RRMSE is 1.15e-2.

7 Hidden discovery with UPINNs

Having introduced the set of test problems, and verified that regular PINNs can solve forward and inverse problems on these systems, this chapter focus on using UPINNs to solve hidden discovery problems, i.e. detecting and learning residual dynamics from data and partially known systems.

These 4 systems are explored in the order presented in chapter 5. The aim is to explore the UPINN idea and its capabilities to discover and model hidden dynamics in a variety of settings. Further, symbolic regression, by means of SINDy and PDE-find, is applied to learn equations from the discovered dynamics.

Some of the following sections explore hyperparameter tuning. In these cases, intermediate results are not necessarily applied chronologically. Checking all combinations of hyperparameters and architectures is a combinatorial problem that is not feasible to solve exhaustively in general. Keeping a baseline model that is varied in each experiment, i.e one-factor-at-a-time search, makes a better foundation for evaluation and comparison. While this does not guarantee that the combination of the marginal best models will be the overall best, it serves as a good foundation for understanding how the model behaves when applied to the problem at hand. Additionally, the main goal is not to find hyperoptimal models, but rather to explore the capabilities of UPINNs.

7.1 Lotka-Volterra with unknown interaction

This section considers the Lotka-Volterra system presented in section 6.1, but with unknown interaction terms. Formally, the system is split into,

$$\mathcal{N}_K = \begin{cases} \alpha x \\ -\delta y \end{cases}, \quad \mathcal{F} = \begin{cases} -\beta xy \\ \gamma xy \end{cases}, \tag{7.1}$$

with $\alpha = 2/3$, $\beta = 4/3$, $\delta = 1$, and $\gamma = 1$. The primary focus in this section is to explore the capabilities of UPINNs in different settings of the Lotka-Volterra system. As such, a predetermined model, presented in section 7.1.1, is used and tuning of hyperparameters is not considered in this section. Except for section 7.1.3, the system defined by (7.1) is used throughout this problem.

7.1.1 Baseline model

As an initial sanity check, a UPINN model was trained on the fully known system, i.e. $\mathcal{F} = 0$. In this case, a UPINN should collapse into a regular PINN. If not, it is unreasonable to expect that the model can accurately learn hidden dynamics. This was confirmed and results can be seen in figure C.2 in Appendix C.

Regarding the formulation (7.1), a UPINN model using FNNs is employed. The model consists of 2 networks, a solution network $\hat{u}_{\theta_u}$ and a residual network $\hat{F}_{\theta_F}$, each containing 4 hidden layers of 16 neurons each with Tanh activation functions. Normal Xavier Glorot weight initialization. As the solution states only attain positive values, a SiLU activation is applied to the solution network output. Although SiLU can produce slightly negative values for inputs below 0, unlike Softplus, it is computationally faster and deemed sufficient for this problem. The residual network takes the solution network outputs, x and y, as input.

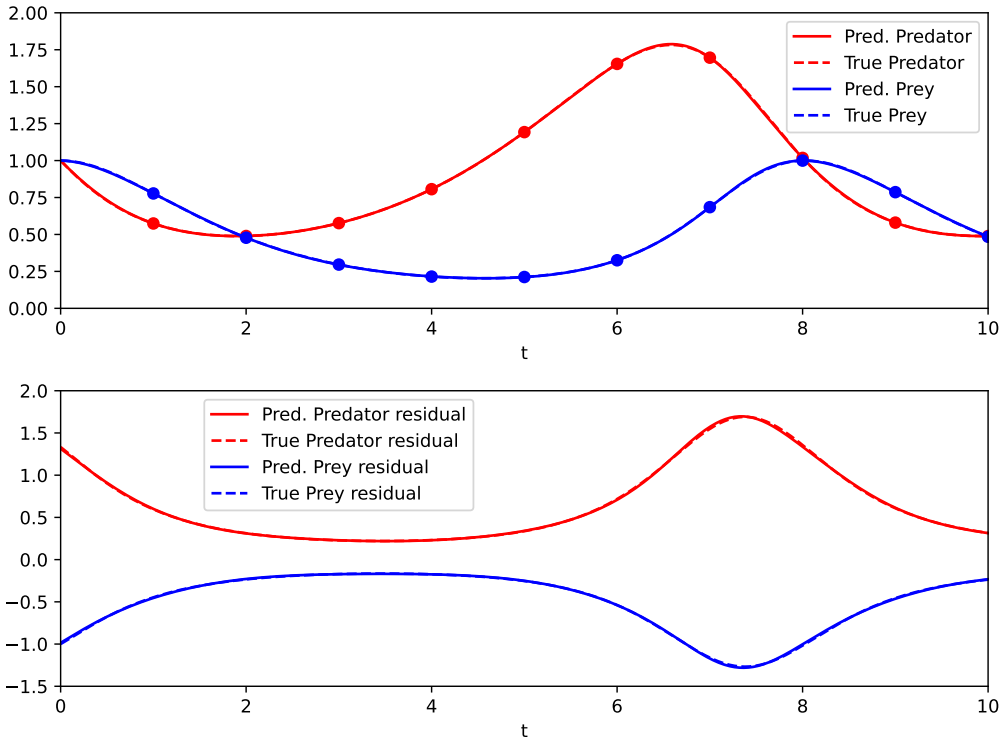

Figure 7.1: Lotka-Volterra: Unknown interaction terms trained on 10 datapoints. **Top**: Predicted solution with 5.31e-4 L^2 error. **Bottom**: Predicted residuals with 2.13e-3 L^2 error.

The model is trained on 10 uniformly spaced data points and 1000 grid collocation points over a domain spanning slighty more than a single cycle, $t \in [0, 10]$. Training is done for 30000 epochs using the Adam optimizer with a learning rate of 1e-3 and weight decay 1e-4. A learning rate scheduler is used to multiply the learning rate by 0.5 if the loss has not decreased for 500 epochs. The results are

presented in figure 7.1 and show that the model is able to predict the solution and residuals with a RRMSE of 5.31e-4 and 2.13e-3 respectively.

Unless otherwise specified, this model serves as the **baseline model** in the following sections.

7.1.2 Extrapolation with UPINN estimated dynamics

Having seen in figure 7.1 that a UPINN can model the hidden dynamics, a natural next step is to test if the residual network, $\hat{F}_{\theta_F}$, is sufficiently accurate to replace the true dynamics when performing extrapolation, like in figure 6.1. The goal is to predict the next cycle of the system, i.e. from $t = 10$ to $t = 20$. The model is kept as in section 7.1.1, except that the collocation points are extended to $t \in [0, 20]$. The weights of $\hat{F}_{\theta_F}$ are locked in place while the solution network $\hat{u}_{\theta_u}$ remains trainable. The results, presented in figure 7.2, show that the model is able to predict the next cycle with a RRMSE of 8.10e-3.

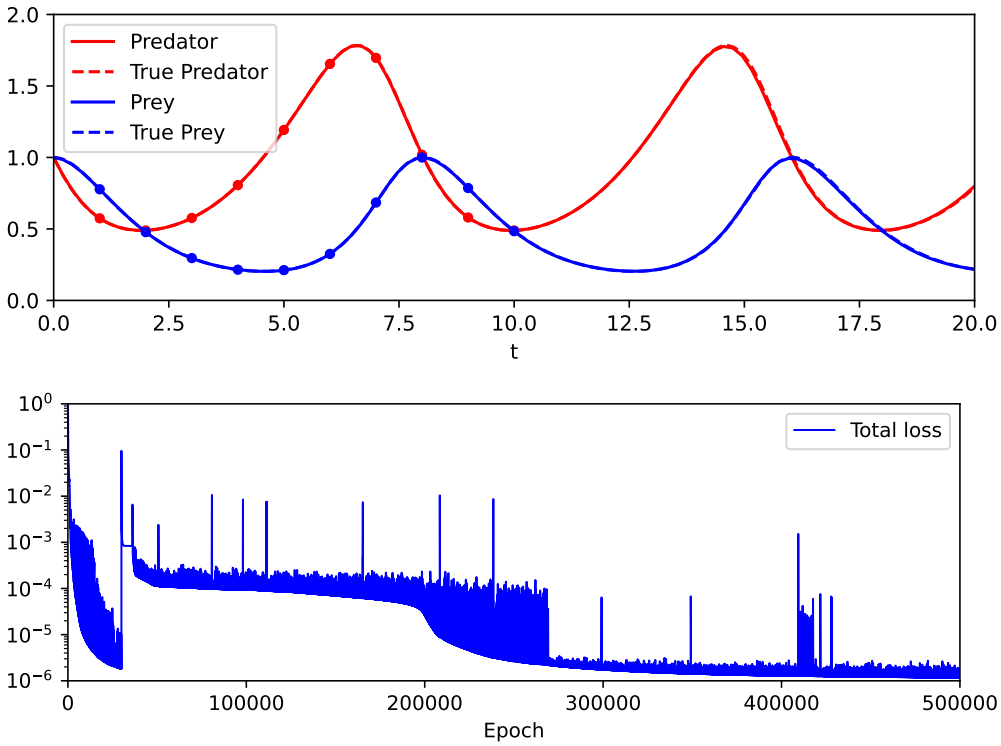

Figure 7.2: Lotka-Volterra: Using UPINN estimated dynamics for extrapolation. **Top**: Predicted solution. **Bottom**: Training loss, $\hat{F}_{\theta_F}$ weights locked at epoch 30000. Solution error: 1.58e-3, Residual error: 8.10e-3.

7.1.3 Hidden discovery combined with inverse problem

Having seen in section 6.1 that a regular PINN can solve the inverse problem, a pressing question is whether UPINN can do this while simultaneously solving the hidden discovery problem. Note that the forward problem is inherently solved in

both cases. This section shows results of using UPINNs to solve hidden discovery and inverse problems of the Lotka-Volterra system in 3 different settings.

The first case represent the most challenging scenario (apart from completely unknown equations), in which both interaction terms and the growth/decay rates, α and δ, are unknown. Training use an initial guess of $\alpha = 1$ and $\delta = 1$. Table 7.1 shows that the model solves the forward problem with an RRMSE of 1.49e-3 on the solution. However, the residual error is significantly higher at 2.35e-1. Additionally, the estimated values of α and δ deviate by a significant margin from the true values.

Repeating the experiment, without setting the random seed of the torch engine, would consistently result in similar errors for solution and residuals, with the estimated parameters varying between runs. The top of figure 7.3 illustrates the residual descrepancy.

In the second case, only α and $-\gamma xy$ are assumed unknown. Using an initial guess of $\alpha = 1$, the results are similar to the first case, yielding a solution error of 1.12e-3, with a high residual error of 5.59e-1. Repeating this experiment would also result in varying estimates of α. The upper middle panel of figure 7.3 show the residual descrepancy in this case. It is seen that even though the predator equation is fully known, except for α, the estimated residuals are non-zero.

An experiment was conducted to see if regularization of the residual network output could improve the results. That is, promote sparsity in the residuals by adding $\lambda_{res}\|\hat{F}\|_1$ to the loss function. This experiment used $\lambda_{res} = 10^{-3}$, and showed a lower residual error of 9.21e-3. Further, the estimated value of α is correct. The resulting residual network is shown in the lower middle of figure 7.3. Applying this technique to the first scenario made the results worse.

In the third case, α, βxy and $-\gamma xy$ are treated as unknowns. However, an inductive bias is imposed on the residual network, enforcing that residuals of the two equations are proportional, i.e. the hidden dynamics are equal to a scaling factor, $F_1 = \phi F_2$, as done in [8]. The results yield a solution error of 1.12e-3 and a low residual error of 2.68e-3, comparable to the baseline model. The bottom panel of figure 7.3 shows that the residual network captures the true hidden dynamics well. Letting α be trainable as well did not yield good results.

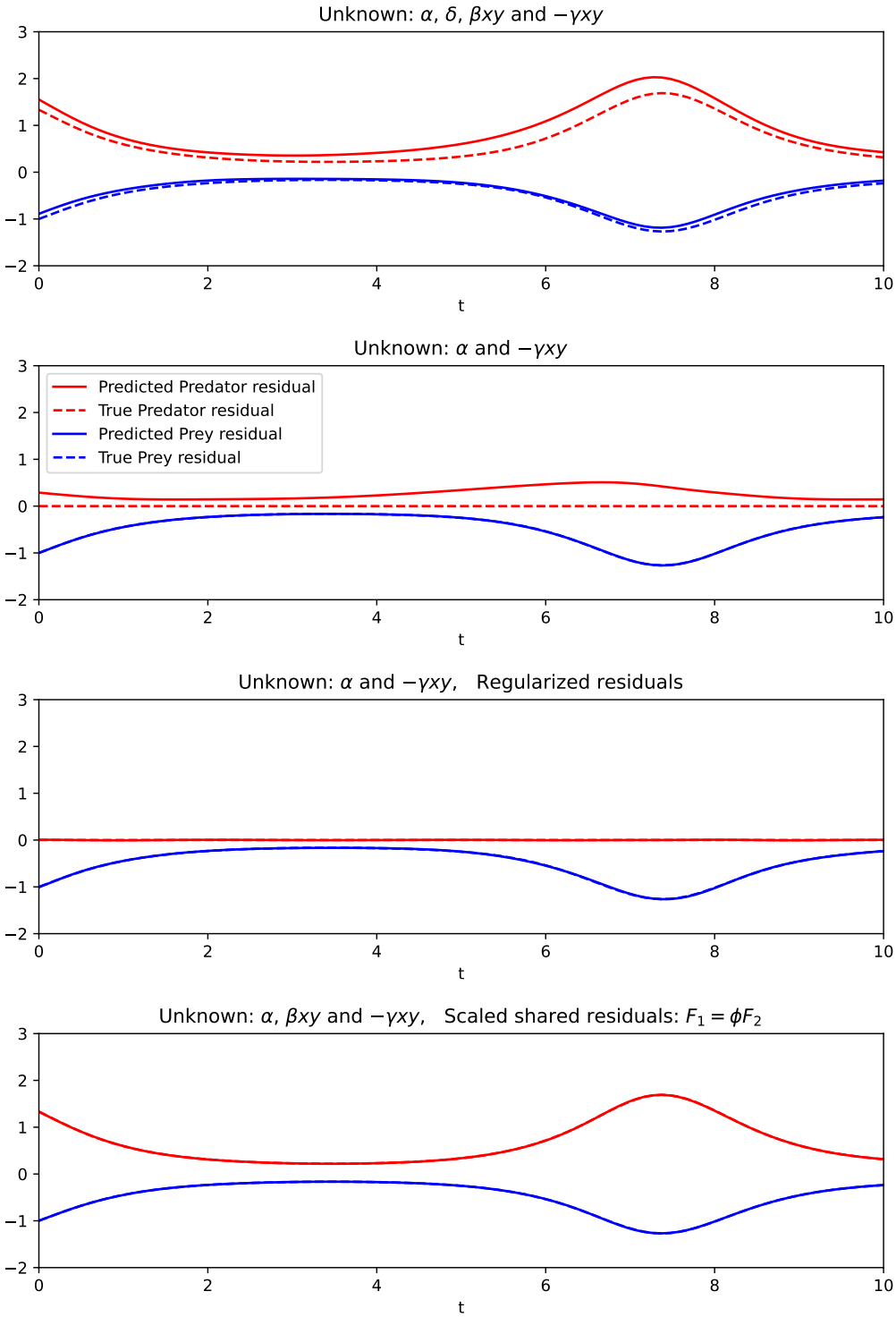

Figure 7.3: Lotka-Volterra: Output of residual network for different configurations of hidden dynamics. **Top**: Assuming α, δ, βxy and $-\gamma xy$ are unknown. **Upper middle**: Assuming α and $-\gamma xy$ are unknown. **Lower middel**: Assuming α and $-\gamma xy$ are unknown with regularized residual network output. **Bottom**: Assuming α, βxy and $-\gamma xy$ are unknown, using a shared residual network. The solution networks are visually indistinguishable from the baseline model in all cases and are omitted, see top of figure 7.1.

Unknowns	Solution error	Residual error	Notes
βxy, $-\gamma xy$	5.31e-4	2.13e-3	Figure 7.1. Baseline.
α, δ, βxy, $-\gamma xy$	1.49e-3	2.35e-1	$\hat{\alpha} = 0.813$, $\hat{\delta} = 0.900$
α, $-\gamma xy$	1.12e-3	5.59e-1	$\hat{\alpha} = 0.842$
α, $-\gamma xy$	1.84e-3	9.21e-3	$\hat{\alpha} = 0.666$, Regularized residual.
α, βxy, $-\gamma xy$	7.55e-4	2.68e-3	Using shared residual network. $\hat{F}_1 = \phi \hat{F}_2$, $\hat{\phi} = -0.750$, $\hat{\alpha} = 0.666$

Table 7.1: Lotka-Volterra: RRMSE and estimated parameters for different variations of hidden dynamics.

7.1.4 Noise and data sparsity

Having seen that UPINNs can model the hidden interaction terms of the Lotka-Volterra system using 10 exact points from the solution, it is now tested how noise and the number of datapoints used affect the performance. Section 7.1.3 showed that imposing an inductive bias of a shared residual network yields good results, thus the noise vs. data sparsity experiment is conducted on both the baseline model and the shared residual network model. The considered problem is the formulation 7.1 with known α and δ. The results are presented in tables 7.2 and 7.3.

It is seen that the shared residual model generally performs better than the baseline model, especially yielding lower solution and residual errors with only 5 data points. As the noiselevel increases, it is seen that increasing the number of datapoints helps reducing the error.

The largest descrepancy between the two models is seen for the case of 5 data points and 0.1 noise level, with 2 orders of magnitude difference in residual error in favor of the shared residual model. These 2 solutions are illustrated in figure C.3 of Appendix C.

Noise\Data	5	10	20	40
0.0	(5.47e-2 / 1.23e-1)	(5.31e-4 / 2.16e-3)	(3.86e-4 / 1.52e-3)	(3.97e-4 / 1.69e-3)
0.001	(4.58e-2 / 8.93e-2)	(7.79e-4 / 2.42e-3)	(6.76e-4 / 2.05e-3)	(4.27e-4 / 1.49e-3)
0.01	(3.27e-2 / 8.73e-2)	(8.04e-3 / 2.19e-2)	(6.55e-3 / 1.06e-2)	(4.09e-3 / 7.16e-3)
0.1	(1.02e-1 / 1.36e-1)	(6.16e-2 / 2.03e-1)	(7.27e-2 / 2.62e-1)	(2.93e-2 / 7.04e-2)

Table 7.2: Lotka-Volterra: RRMSE (Solution / Residual) for different combinations of datapoints and noise levels. Unknowns are βxy and $-\gamma xy$.

Noise\Data	5	10	20	40
0.0	(1.61e-3 / 3.74e-3)	(3.89e-4 / 1.18e-3)	(4.40e-4 / 1.36e-3)	(1.96e-4 / 9.30e-4)
0.001	(3.52e-3 / 7.00e-3)	(4.47e-4 / 8.93e-3)	(4.05e-4 / 8.61e-4)	(3.15e-4 / 1.06e-3)
0.01	(7.87e-3 / 1.32e-2)	(6.75e-3 / 8.00e-3)	(2.70e-3 / 4.79e-3)	(1.83e-3 / 3.59e-3)
0.1	(7.50e-2 / 8.73e-2)	(5.61e-2 / 5.04e-2)	(5.07e-2 / 9.78e-2)	(2.34e-2 / 4.87e-2)

Table 7.3: Lotka-Volterra: RRMSE (Solution / Residual) for different combinations of datapoints and noise levels. Unknowns are βxy and $-\gamma xy$. Using **inductive bias** $\hat{F}_1 = \phi \hat{F}_2$. Values of ϕ are omitted here but listed in table C.2 in Appendix C.

7.1.5 Brief note on interpretability using KANs

Neural networks are often criticized for their black-box nature. Although they can approximate any function arbitrarily well, they lack capabilities in providing insight into the underlying dynamics that they model. KANs are different in this context, as they approximate a computational chain of activation functions, potentially allowing for better interpretability. Figure 7.4 shows two instances of a [2,2,2] KAN with gridsize 5 trained as the residual network in the baseline setup, aiming to approximate the interactions βxy and $-\gamma xy$. Appendix A.2 provides a more detailed explanation of how KANs can approximate interactions.

While human interpretation of the learned activation functions in figure 7.4 is not a formal scientific measure, the input layer of the left KAN, though not perfect, could arguably resemble linear functions with the second layer showing quadratic like structure. Such a KAN is able to represent the product xy as shown by equation A.2 in Appendix A.2. Applying the build-in symbolic regression tool in the Python KAN library did not yield these results, but rather more complicated expressions.

Arguably, the learned activation functions in figure 7.4 are more interpretable for humans than the learned weights of a regular FNN. The right KAN in figure 7.4 could perhaps be interpreted as the case presented in equation A.1 in Appendix A.2, allbeit if one did not know the true terms, this seems like a stretch.

Note, this serves merely as an example of the interpretability that KANs could potentially provide, rather than a formal in depth analysis of the architecture as that is beyond the scope of this project.

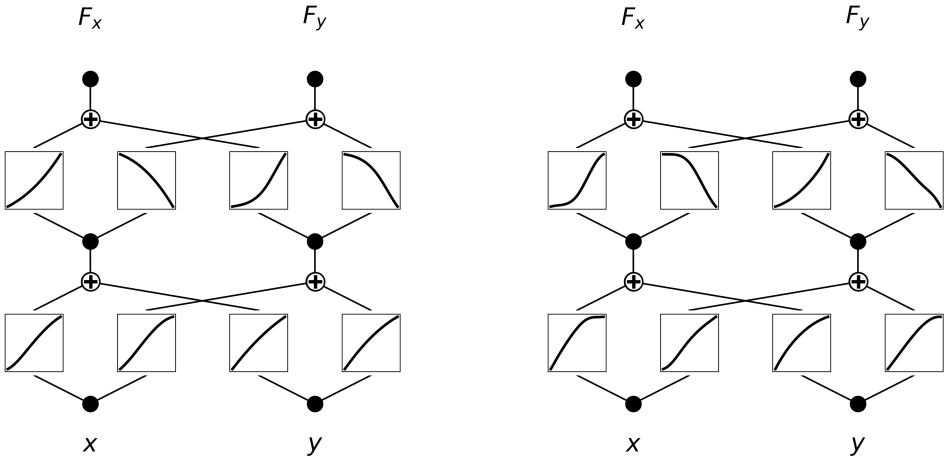

Figure 7.4: Lotka-Volterra: 2 runs of training a KAN as the residual network on the baseline setup.

7.1.6 Interaction term discovery with SINDy

So far, UPINNs have been used to model the hidden dynamics, being a surrogate for the unknown equations. While this has shown successful in various settings, the ultimate goal is to recover the true underlying equations. This section uses symbolic regression, specifically SINDy, to find equations for the residual networks trained in section 7.1.4.

A 2nd order polynomial library is used for the SINDy model, see table D.3 in Appendix D. The SR3 optimizer is supplied with data generated with the residual network, $\hat{F}_{\theta_F}$, at 100 equispaced points on the interval $t \in [0, 10]$. The optimizer threshold is determined using 10-fold time series cross validation. The results are presented in table 7.4. It was attempted to apply SINDy directly on the data used to train the UPINN models, but this did not yield correct equations.

Noise\Data	5	10	20	40
0.0	$1.286xy$ $-0.880xy - 0.110y^2$	$1.333xy$ $-1.000xy$	$1.333xy$ $-1.000xy$	$1.333xy$ $-1.000xy$
0.001	$0.069x^2 + 1.173xy$ $-1.012xy$	$1.333xy$ $-1.000xy$	$1.333xy$ $-1.000xy$	$1.334xy$ $-1.000xy$
0.01	- $-0.950xy$	$1.318xy$ $-0.991xy$	$1.332xy$ $-0.995xy$	$1.334xy$ $-0.999xy$
0.1	-	-	- $-0.964xy$	-

Table 7.4: Lotka-Volterra: SINDy discovered equations for the residual networks presented in table 7.2. (-) Found equations contained many wrong terms.

As seen in table 7.4, the SINDy model was able to recover the true interaction terms, including scalar values, in a variety of settings. It struggles with the highest

noise level (10%) and found less accurate equations when only using 5 datapoints.

To test if results could be improved, the experiment was repeated using a shared residual network, imposing the inductive bias $\hat{F}_1 = \phi\hat{F}_2$. As shown in table 7.5, this enabled the model to recover the correct equations using only 5 datapoints, in addition to the known dynamics. Additionally, it found the true interaction terms using 20 and 40 datapoints with 10% noise. For 5 and 10 datapoints with 10% noise, it found the correct term plus a small quadratic term.

Noise\Data	5	10	20	40
0.0	$1.332xy$ $-1.001xy$	$1.333xy$ $-1.000xy$	$1.333xy$ $-1.000xy$	$1.333xy$ $-1.000xy$
0.001	$1.336xy$ $-0.998xy$	$1.334xy$ $-1.000xy$	$1.334xy$ $-1.000xy$	$1.333xy$ $-1.000xy$
0.01	$1.328xy$ $-1.004xy$	$1.336xy$ $-0.995xy$	$1.333xy$ $-1.002xy$	$1.332xy$ $-0.998xy$
0.1	$1.264xy + 0.109y^2$ $-0.910xy - 0.079y^2$	$0.055x^2 + 1.231xy$ $-0.039x^2 - 0.883xy$	$1.324xy$ $-0.991xy$	$1.348xy$ $-0.994xy$

Table 7.5: Lotka-Volterra: SINDy discovered equations for the residual networks presented in table 7.3.

As a closing note of this section, SINDy was applied to the model with unknown interactions, α and δ from table 7.1. The resulting equations are shown below added to the known part of the system with the estimated values of α and δ,

$$\underbrace{\begin{bmatrix} -0.813x \\ 0.900y \end{bmatrix}}_{\text{Known estimate}} + \underbrace{\begin{bmatrix} 0.145x + 1.334xy \\ 0.099y - 1.000xy \end{bmatrix}}_{\text{SINDy on residual estimate}} = \begin{bmatrix} -0.668x + 1.334xy \\ 0.999y - 1.000xy \end{bmatrix},$$

which yield an accurate estimate of the true equations, indicating that the residual network captured part of the known dynamics. This is discussed further in section 9.1.

7.2 Detecting hidden resusceptibility in SIR model

SIR type models are highly relevant and frequently used to model the spread of infectious diseases, including the recent COVID-19 [57] pandemic. Epidemic systems is a prime real world example of dynamical systems in which underlying mechanisms are only partially known. They often contain hidden dynamics such as dark numbers regarding the reported number of infected individuals, competing mutations, temporary immunity, etc.

Note, there is generally no limit to how comprehensive SIR type models can be, as

they may be extended with additional compartments to model additional dynamics such as exposed, dead, vaccinated, hospitalized, ICU, etc. and various flows and interactions between these compartments.

This section focus on a simple extension of the basic SIR model that allow recovered individuals to gradually lose immunity and become susceptible again. It is assumed that this dynamic is present, but unknown, thus the disease is thought to cause permanent immunity when it in fact does not. The formulation of the resusceptibility model for UPINN is thus given by equation (7.2),

$$\mathcal{N}_K = \begin{cases} -\beta SI \\ \beta SI - \gamma I \\ \gamma I \end{cases}, \quad \mathcal{F} = \begin{cases} \theta R \\ 0 \\ -\theta R \end{cases}, \tag{7.2}$$

where θ is the rate at which recovered individuals become susceptible again. The specific problem considered is $\beta = 0.3$, $\gamma = 1/21$ and $\theta = 1/90$, corrosponding to a relatively high spread, an average infectious period of 21 days, and an average immunity period of 90 days. The initial conditions are $S(0) = 0.9999$, $I(0) = 0.0001$ and $R(0) = 0$, and the system is observed over a period of 60 days with datapoints measured weekly. At the end of the observed period, $t = 60$ days, the number of susceptibles has only decreased (see top of figure 7.8).

Having already experimented with noise, data sparsity and symbolic regression on a compartment model in section 7.1, this section focus on constraining the model to prevent undesired estimation.

Unless otherwise specified, a FNN with 3 layers of size 16 and Tanh activation function, using Normal Xavier Glorot initialization, is used for both solution and residual network throughout this section. Further, the solution network has softmax activation on the output layer to ensure that the total population is 1. The models are trained for 50000 epochs using the AdamW optimizer with a learning rate of 1e-3 and a learning rate scheduler multiplying the learning rate by 0.5 if the loss has not decreased for 500 epochs. The input t is normalized to the range $[0, 1]$.

Figure 7.5 shows the resulting residual predictions using a standard UPINN. It is seen that the model is able to detect a residual flow into susceptible and out of recovered, suggesting that some dynamic exist between these compartments. Small fluctuations are also present in the infected compartment, which should ideally be zero. It is also evident that the residual prediction exhibit oscillatory

behavior around the true dynamics. As the total population is constant, the total flow between compartments should also be zero, which is seen to have small deviations.

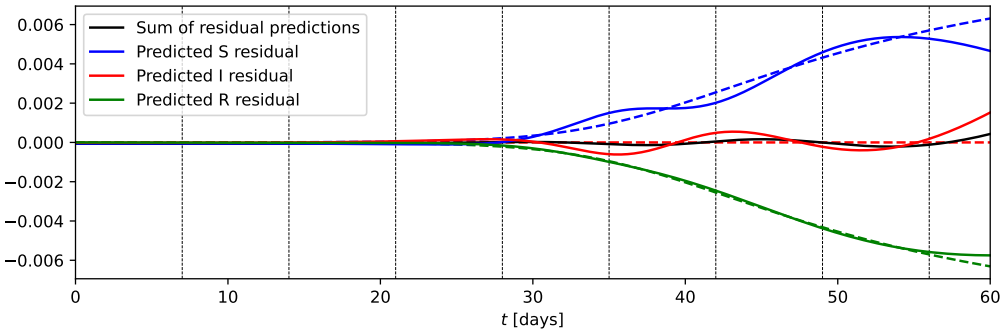

Figure 7.5: SIR: Standard UPINN residual network prediction of the hidden dynamics in the resusceptibility model. Vertical lines indicate the training data points. Dashed lines are the true dynamics. Solution RRMSE: 1.38e-3, Residual RRMSE: 1.32e-1.

Having tested the use of the standard UPINN formulation, the next step is to see if results can be improved with additional constraints and assumptions. The first experiment will strongly enforce zero total flow by letting the residual network model the flow between compartments rather than the total flow of the individual compartments. This is done by rearranging the output as in equation (7.3), which trivially sums to zero,

$$\hat{F} = \begin{cases} \hat{F}_1 - \hat{F}_2 \\ \hat{F}_2 - \hat{F}_3 \\ \hat{F}_3 - \hat{F}_1 \end{cases} . \tag{7.3}$$

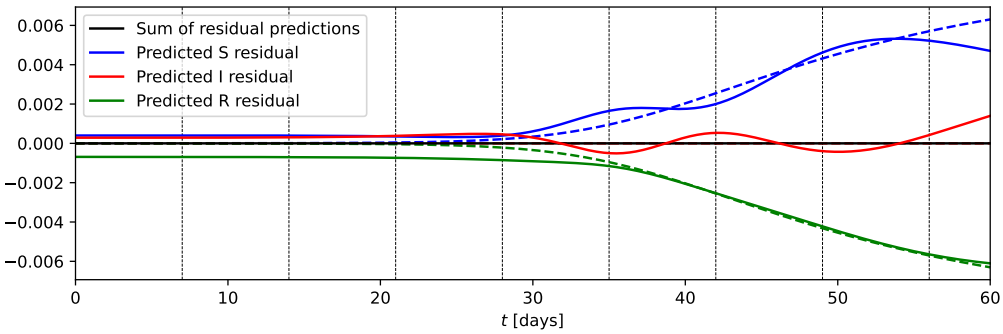

Figure 7.6: SIR: Prediction of residuals in the resusceptibility model using a zero total flow residual network. Vertical lines indicate the training data points. Dashed lines are the true dynamics. Solution RRMSE: 1.26e-3, Residual RRMSE: 2.08e-1.

Figure 7.6 shows the results of this experiment. It is seen that this trivial net zero flow gave rise to a new issue in the early parts of the epidemic, where the model

predicts significant residual flow between compartments when it is not actually present. Notably, it models flow from recovered into both susceptible and infected.

In an attempt to correct this behavior, it was tried to regularize the output of the residual network to promote smaller residual estimation. This was done by adding a term $\lambda_{res}\|\hat{F}\|_1$ to the loss function with $\lambda_{res} = 10^{-3}$. Figure 7.7 shows that this regularization did reduce the undesired flow in the early parts of the epidemic.

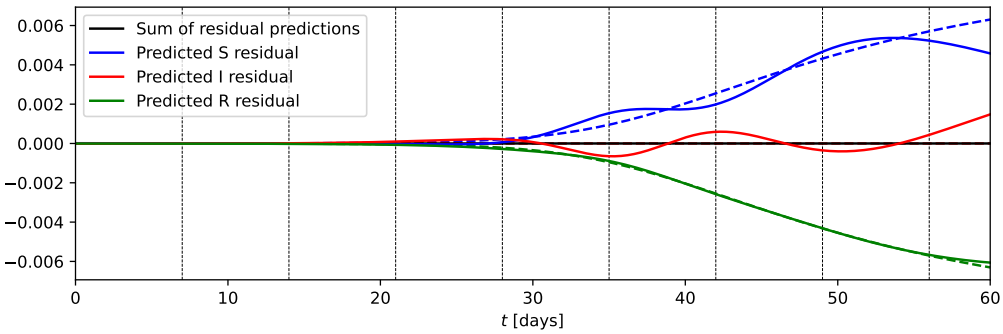

Figure 7.7: SIR: Prediction of residuals in the resusceptibility model using a zero total flow residual network with L^1 regularization on the output of $\hat{F}$. Vertical lines indicate the training data points. Dashed lines are the true dynamics. Solution RRMSE: 1.35e-3, Residual RRMSE: 1.40e-1.

The oscillatory behavior could potentially be caused by the residual network being too flexible. To test this, the model in figure 7.7 was trained using a FNN with only a single hidden layer of 16 neurons. The resulting model, presented in figure 7.8, shows that this indeed reduced the oscillatory behavior, subsequently reducing the solution and residual errors. The experiment was repeated using the standard UPINN residual network, which also showed to lower the error, but not as much as for the zero total flow model. RRMSE of all of these models are listed in table 7.6. Note that the solution networks are visually indistinguishable in all these models and are omitted except for the one in figure 7.8.

Model	**Solution RRMSE**	**Residual RRMSE**	**Figure**
Standard	1.38e-3	1.32e-1	7.5
Standard, regularized, shallow	8.13e-4	6.54e-2	-
Zero total flow (ZTF)	1.26e-3	2.08e-1	7.6
ZTF, regularized	1.35e-3	1.40e-1	7.7
ZTF, regularized, shallow	**5.93e-4**	**5.36e-2**	7.8

Table 7.6: SIR: Solution and residual RRMSE for resusceptibility models

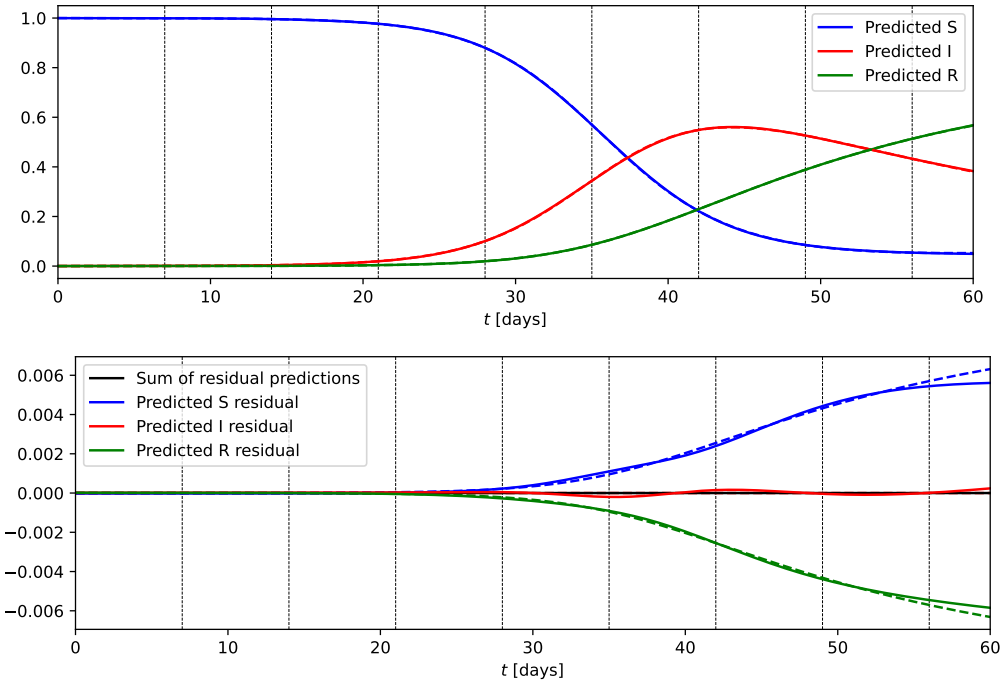

Figure 7.8: SIR: Resusceptibility model using a zero total flow residual network with L^1 regularization on the output of $\hat{F}$ and a shallow residual network. **Bottom:** Predicted residuals. **Top:** Predicted solution. Vertical lines indicate the training data points. Dashed lines are the true dynamics. Solution RRMSE: 5.93e-4, Residual RRMSE: 5.36e-2.

7.3 Burgers' equation with unknown convection term

This section considers Burgers' equation from section 6.3 with unknown convection term. Formally, the UPINN model should approximate u and $\mathcal{F}$ when the PDE is split into the following known and unknown parts,

$$\mathcal{N}_K = u_t - \nu u_{xx}, \quad \mathcal{F} = u u_x. \tag{7.4}$$

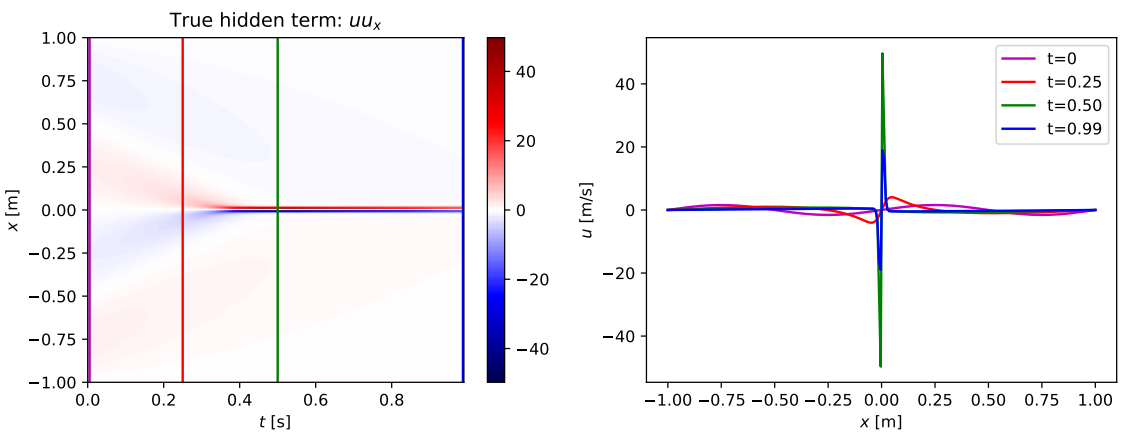

Figure 7.9: **Left:** True hidden dynamics of Burgers' equation. **Right:** Slices of vertical lines in left plot.

The true hidden dynamics are visualized in figure 7.9, with the true solution shown in figure 6.6. It is seen that the hidden dynamics are quite extreme around $x = 0$ when the wave collides with itself.

7.3.1 Tuning hyperparameters

This section explores different settings of hyperparameters for the model, when used on Burgers' equation.

Unless otherwise specified, a baseline model consisting of an FNN with 8 hidden layers of 20 neurons each, with Tanh activation function, is used for both solution and residual network. Training is done on 10000 sobol sampled collocation points, 100 grid points on each of the 3 boundaries and spatial data sampled at $t = [0.25, 0.5, 0.99]$ (see figure 7.9). Normal Xavier Glorot initialization is used for the weights.

In the following, "clean" refers to the noise-free data, while "noisy" refers to data with a noise level of 0.1, artificially added according to equation (5.1).

Training is done for 1000 epochs using the Adam optimizer with a learning rate of 2e-3 to obtain an initial solution, which is then followed by the LBFGS optimizer with learning rate 0.1 until convergence (LBFGS can no longer find optimization steps), usually 10000-20000 epochs.

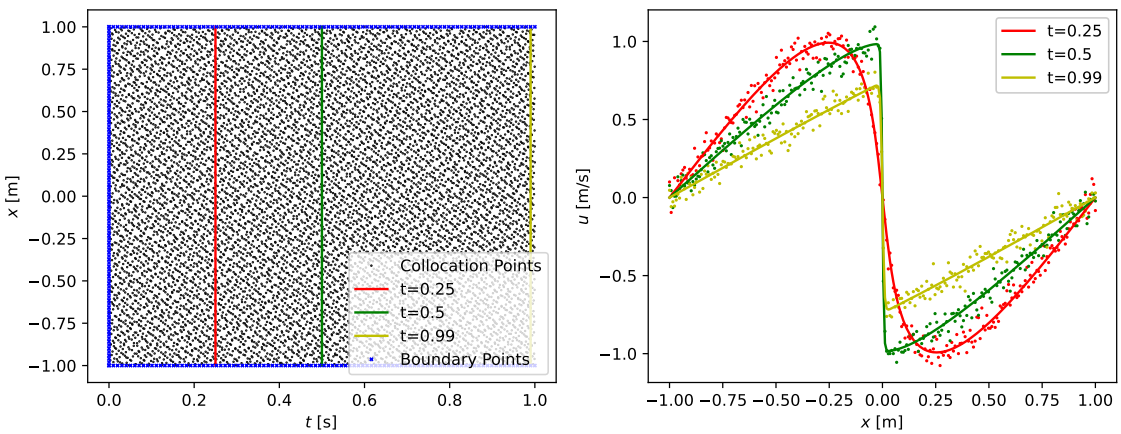

Figure 7.10: Burgers': Data used for training the UPINN model on Burgers' equation with unknown convection term. **Left:** Boundary points, collocation points and data points at selected time steps. **Right:** Data slices, lines are the noise-free signal and dots are 0.1 noiselevel data.

For comparison, the baseline model is first trained on clean data. Figure 7.11 shows the resulting predictions. This was also done using only data sampled at $t = 0.5$, which yielded similar results, see figure C.4 in Appendix C. Worth noting is that the residual error is quite high, even though it visually seems to capture the

true dynamics well.

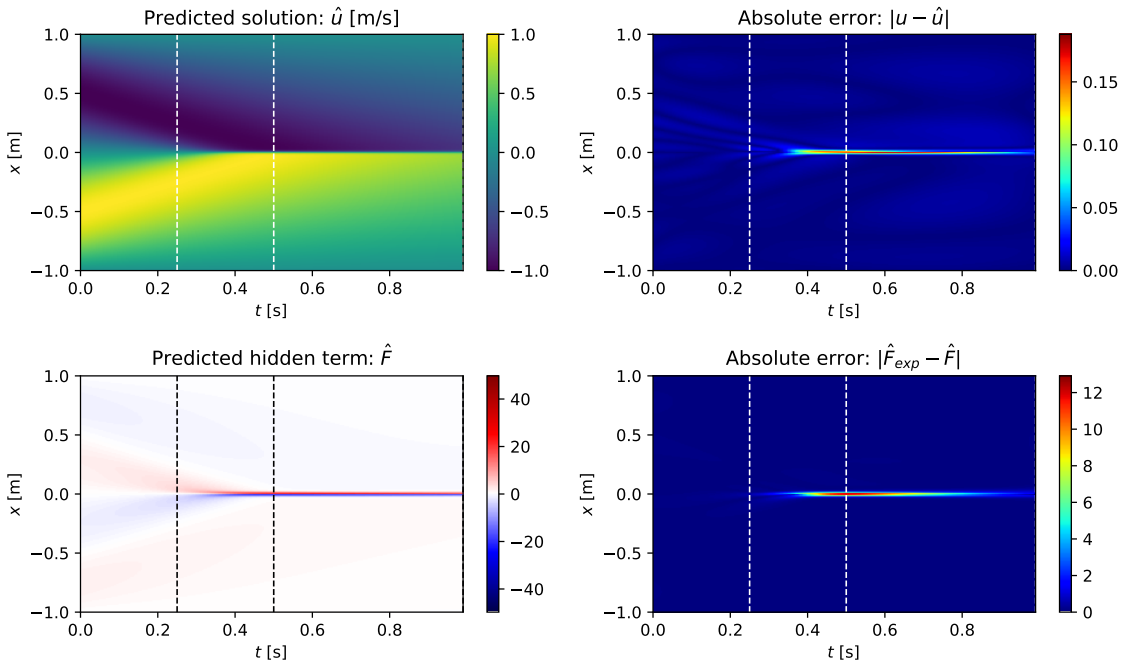

Figure 7.11: Burgers': UPINN solution using baseline of [20]*8 FNN with Tanh activation, training on clean data. Vertical lines indicate data. Solution error: 1.57e-2, Residual error: 2.50e-1.

Activation function

The first experiment tests different activation functions for the model. Table 7.7 shows the results of the experiments on noise-free data, indicating that the Tanh activation function is the best suited for this problem. Figure 7.13 shows that SiLU has faster convergence, but Tanh achieves lower loss.

The same experiment is repeated with a noise level of 0.1, with results also presented in table 7.7. Worth noting is that SiLU achieved lower residual error in a noisy setting, and SiLU and Tanh both showed lower solution errors, with Tanh being slightly better.

Activation function	Solution error	Residual error
Tanh	(**1.57e-2** / **1.82e-2**)	(**2.50e-1** / 2.31e-1)
SiLU*	(1.86e-2 / 1.92e-2)	(3.23e-1 / **1.11e-1**)
Sigmoid	(2.44e-2 / 3.94e-2)	(2.72e-1 / 5.91e-1)

Table 7.7: Burgers': RRMSE (Clean / Noisy) for different activation functions on clean and noisy data. (*) Using normal Kaiming He initialization.

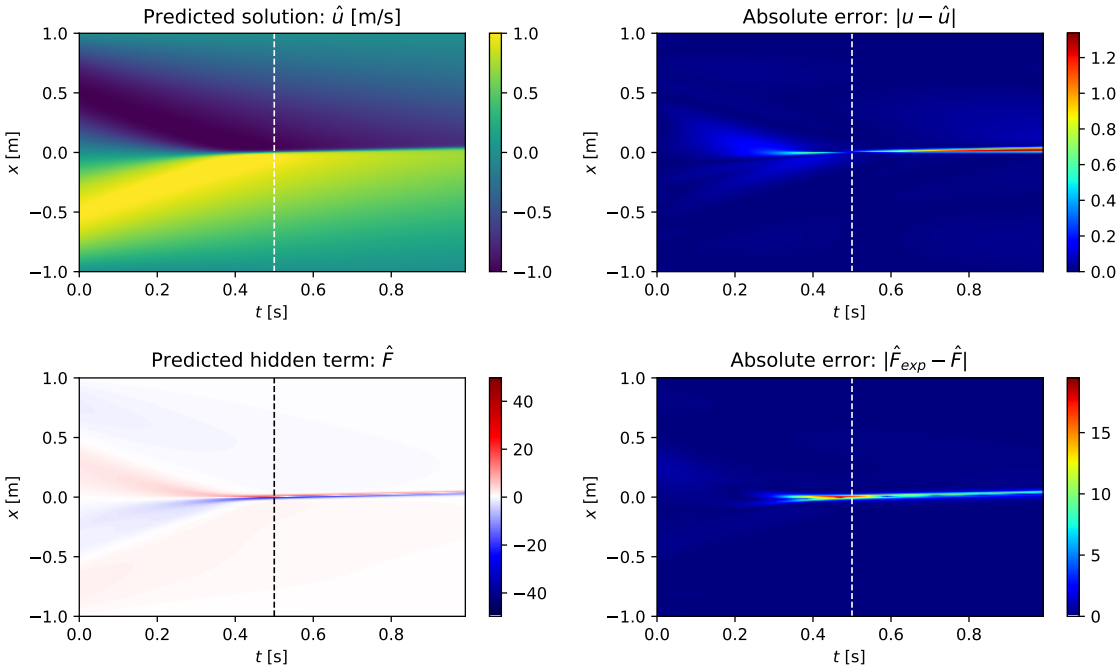

Figure 7.12: Burgers': UPINN solution using baseline of [20]*8 FNN with Tanh activation, training on sparse noisy data. Vertical line indicate data.

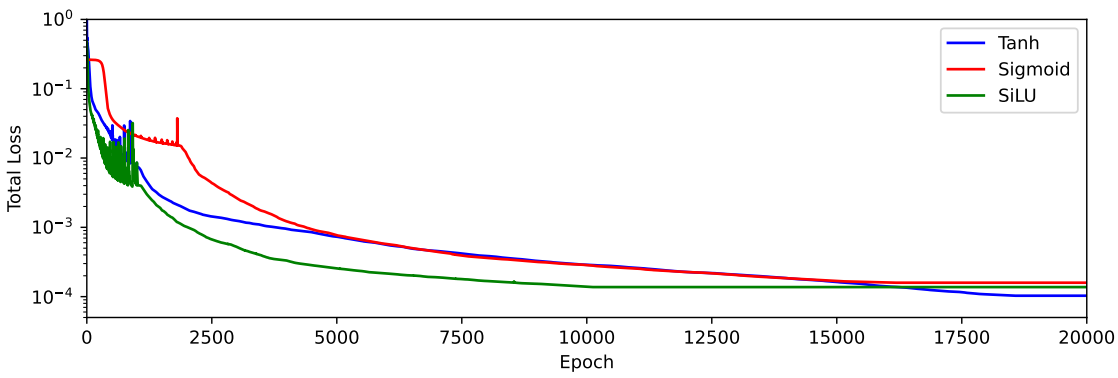

Figure 7.13: Burgers': Loss curves for different activation functions on Burgers' equation using clean data.

To round off experiments on the activation function, performance is tested in a very sparse setting with only $t = 0.5$ data with noiselevel 0.1. The results are shown in table 7.8. While Tanh has the worst solution error, it has the best residual error. Using a combination of SiLU and Tanh was tested, but yielded slightly worse results than just using SiLU. Figure 7.12 visualize the UPINN solution using Tanh activation function trained using noisy data sampled only at $t = 0.5$. It is seen that high noise levels makes the solution network prediction become skew beyond the training data. Worth noting is that switching to a sparse setting increased solution error by an order of magnitude, while the residual error remained relatively stable.

Activation function	Solution error	Residual error
Tanh (7.12)	2.23e-1	**2.72e-1**
SiLU	**1.99e-1**	3.04e-1
Sigmoid	2.14e-1	3.33e-1
SiLU + Tanh	2.09e-1	3.06e-1

Table 7.8: Burgers': RRMSE for different activation functions on Burgers' equation using only data at $t = 0.5$ with noiselevel 0.1.

Overall the experiments on activation functions indicate that SiLU and Tanh have a slight edge over sigmoid.

Loss weight balancing

The next experiment is to see if balancing the loss terms can improve the results. These experiments are conducted using Tanh activation function. The experiments goes on SoftAdapt with different β values, and manual weight balancing using educated guesses, i.e. in noisy settings it might be beneficial to give more weight to the residual loss and less to the data loss. The results are shown for clean and noisy data in table 7.9 below.

Weightloss scheme	Solution error	Residual error
Equal weights	(1.57e-2 / 1.82e-2)	(2.50e-1 / 2.31e-1)
$\lambda = [0.2, 0.4, 0.8]$	(2.52e-2 / 1.84e-2)	(3.35e-1 / 2.42e-1)
$\lambda = [0.4, 0.2, 0.8]$	(2.89e-2 / 2.31e-2)	(3.42e-1 / 2.93e-1)
SoftAdapt($\beta = 1$)	(2.10e-2 / 2.03e-2)	(2.81e-1 / 2.54e-1)
SoftAdapt($\beta = 0.5$)	(1.77e-2 / 2.02e-2)	(2.45e-1 / 2.41e-1)
SoftAdapt($\beta = 0.1$)	(**1.31e-2** / 2.10e-2)	(**2.14e-1** / 2.29e-1)
SoftAdapt($\beta = 0.1$)*	(2.85e-2 / 3.66e-2)	(3.21e-1 / 3.48e-1)
SoftAdapt($\beta = 0.05$)	(1.91e-2 / **1.79e-2**)	(2.66e-1 / **2.10e-1**)

Table 7.9: Burgers': RRMSE (Clean / Noisy) for different weightloss schemes with clean and noisy data. $\lambda = [\lambda_{bc}, \lambda_{data}, \lambda_{pde}]$. (*) Lossweighted SoftAdapt.

It is seen that some configurations of SoftAdapt yield slight improvements in solution error compared to having equal weights, with $\beta = 0.1$ almost cutting the solution error in half. Worth noting is that SoftAdapt sometimes made training with LBFGS unstable, especially for noisy data, leading to divergence of the training process. Using loss weighted SoftAdapt was found to make the results worse. Figure 7.14 present the loss curves for these experiments, indicating that SoftAdapt generally speed up convergence, but larger β values seem to get stuck in local minima, preventing further convergence. Using no SoftAdapt, i.e. $\beta = 0$, still yield the lowest loss, although it has slower convergence.

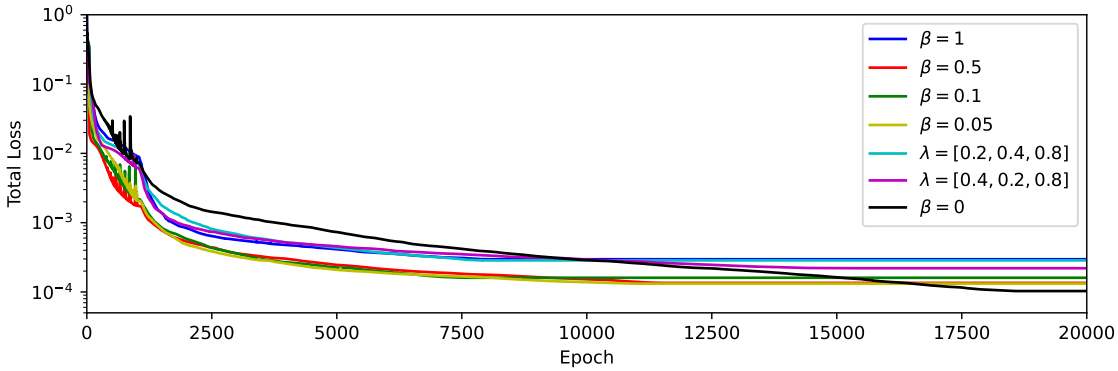

Figure 7.14: Burgers': Loss curves of different values of β in SoftAdapt for Burgers' equation using clean data.

Sampling strategy

All previous experiments have used 10000 sobol sampled collocation points. This experiment tests if other sampling strategies can improve the results. Apart from sobol, latin hypercube sampling (LHS), grid sampling and uniform random sampling are tested. Table 7.10 below shows the results for clean and noisy data.

Sampling	Solution error	Residual error
Sobol 10000	(1.57e-2 / 1.82e-2)	(**2.50e-1** / **2.31e-1**)
Lhs 10000	(1.76e-2 / 1.85e-2)	(2.65e-1 / 2.47e-1)
Grid 10000	(**1.53e-2** / **1.72e-2**)	(5.89e-1 / 6.32e-1)
Uniform 10000	(1.70e-2 / 2.24e-2)	(2.83e-1 / 3.09e-1)

Table 7.10: Burgers': RRMSE (Clean / Noisy) for different sampling methods with clean and noisy data.

While grid sampling has slighty lower solution errors compared to the other methods, it yield significantly worse residual errors. The quasi random methods had the lowest residual errors, with Sobol being slightly better than LHS. Thus, Sobol sampling is used to test the impact of sampling size. As the residual error for clean and noisy data was seen to be quite close, this experiment was only performed on noisy data. The results are presented in table 7.11, and show that increasing the sample size, from the baseline of 10000 points, does not improve results, while increasing the computational cost. Further, reducing the number of points to 5000 increased the errors.

Sampling	Solution error	Residual error
Sobol 5000	2.24e-2	3.09e-1
Sobol 10000	**1.82e-2**	**2.31e-1**
Sobol 50000	1.93e-2	2.34e-1
Sobol 100000	1.86e-2	2.34e-1

Table 7.11: Burgers': RRMSE for different sampling sizes using data with noiselevel 0.1.

Finally, the adaptive sampling strategies of RAR, RAD and RAR-D are tested. A sobol sampler is used to generate candidate points. The experiment is conducted on clean data at the 3 time points. Training is done by alternating between 1000 epochs of Adam with learning rate 1e-3 and 1000 epochs of LBFGS with learning rate 0.1 followed by resampling, as suggested in [16].

Table 7.12 shows the results of this experiment. It is seen that the adaptive sampling strategies perform similarly. However, the baseline still outperforms all adaptive sampling strategies in this setting.

Sampling	Solution error	Residual error
Baseline	1.57e-2	2.50e-1
RAR	3.96e-2	5.35e-1
RAD	2.66e-2	3.72e-1
RAR-D	2.87e-2	3.90e-1

Table 7.12: Burgers': RRMSE of applying adaptive sampling strategies using clean data.

Architecture

The FNN is often a good baseline model. However, different architectures might suit some problems better than others. This experiment investigates other sizes of FNNs, ResNets, KANs and the use of ansatz networks.

The unknown term $\mathcal{F}$ is a product of u and u_x, which (as shown in Appendix A.2) can be represented by a [3,2,1] KAN. Thus such a KAN might be a good candidate for the residual network, although it would take all 3 terms, u, u_t and u_x as input. The ansatz used is the one described in equation (3.8).

Architecture	Solution error	Residual error
FNN $[20] * 8$	1.57e-2	2.50e-1
FNN $[40] * 8$	**8.16e-3**	1.86e-1
FNN $[20] * 4$	1.99e-2	2.68e-1
FNN $[40] * 4$	1.04e-2	**1.65e-1**
FNN $[20] * 8$ Ansatz*	3.01e-0	8.92e-1
ResNet $[20] * 2 * 4$	1.44e-2	2.04e-1
ResNet $[40] * 2 * 4$	1.53e-2	2.33e-1
KAN $[3, 5, 5, 1]$	7.36e-2	6.19e-1
u: ResNet $[20] * 2 * 4$, F: FNN $[20] * 8$	3.04e-2	3.27e-1
u: ResNet $[20] * 2 * 4$, F: KAN $[3, 2, 1]$	1.96e-2	3.78e-0
u: ResNet $[20] * 2 * 4$, F: KAN $[3, 4, 4, 2, 1]$	1.88e-2	2.27e-1

Table 7.13: Burgers': RRMSE for different architectures with clean data. (*) Ansatz as defined by equation (3.8).

While KANs can theoretically represent the residuals perfectly, they did not perform well in this experiment. The best performing architectures were the wider FNNs with 40 neurons in each layer. While all architectures were not tested with noisy data, the best performing from table 7.13 were. The results are shown in table 7.14. The wider FNNs do not perform as well in a noisy setting, suggesting that the larger models are more prone to overfitting the noise.

Architecture	Solution error	Residual error
FNN $[20] * 8$	1.82e-2	2.31e-1
FNN $[40] * 8$	1.91e-2	2.67e-1
FNN $[40] * 4$	**1.80e-2**	2.27e-1
ResNet $[20] * 2 * 4$	2.12e-2	**2.28e-1**
ResNet $[40] * 2 * 4$	2.18e-2	2.32e-1

Table 7.14: Burgers': RRMSE for different architectures with noisy data.

7.3.2 Removing the inductive bias

In [8] they suggest imposing an inductive bias on the residual network, by directly supplying $\hat{u}$, $\hat{u}_x$ and $\hat{u}_t$ as inputs. This was also the assumption throughout section 7.3.1 above. This section investigates how the model performs without this inductive bias.

In this setting, the solution output $\hat{u}$ and spatial data x and t are passed as inputs to the residual network. This experiment was initially performed using the same architecture as the baseline model, i.e. a FNN with 8 hidden layers of 20 neurons

each and Tanh activation function. It was also tested using RAD sampling, a ResNet and a combination of an FNN and a KAN. The resulting model errors are presented in table 7.15 below, and figure 7.15 shows the results of the baseline model without inductive bias.

It is evident that the FNN models achieve errors slightly higher, but still comparable, to the inductive bias models.

Model	**Solution error**	**Residual error**
FNN $[20] * 8$, figure 7.15	2.19e-2	3.43e-1
FNN $[20] * 8$ RAD	2.17e-2	3.43e-1
u: FNN $[40] * 8$, F: KAN [3,5,5,1]	6.55e-2	4.92e-1
ResNet $[20] * 2 * 4$	3.95e-2	3.60e-1

Table 7.15: Burgers': RRMSE for different models trained without inductive bias on clean data.

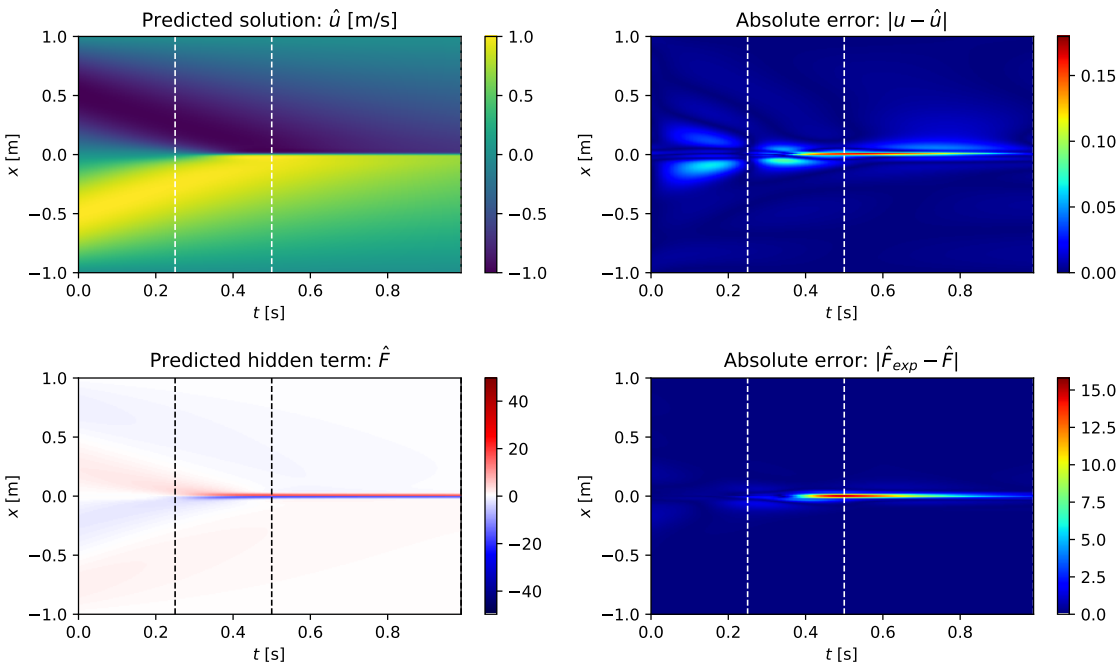

Figure 7.15: Burgers': UPINN solution without inductive bias. Vertical lines indicate data.

7.3.3 Solving Burgers' equation with PINN using residual network

In section 7.1.2 it was seen that a regular PINN can use the known dynamics in combination with the trained residual network to accuratly solve the differential equation. This section investigates if the same can be done for Burgers' equation. This is done with the residual network from the baseline model in figure 7.11 as well as the residual network without inductive bias in figure 7.15, where a new solution network is trained using no data and freezing the weights of the residual

network.

The results showed that the PINN model, using the residual network with inductive bias, obtained a reasonable solution compared to the baseline, yielding a solution error of 3.75e-2. The result is shown in figure 7.16. In contrast, the residual network without inductive bias could not converge to anything resembling a good solution, indicating that it does not function well as a standalone surrogate model for the differential operator. It was attempted to commence training with the solution network obtained when training the residual network without inductive bias, from figure 7.15, which immediatly diverged from the found solution.

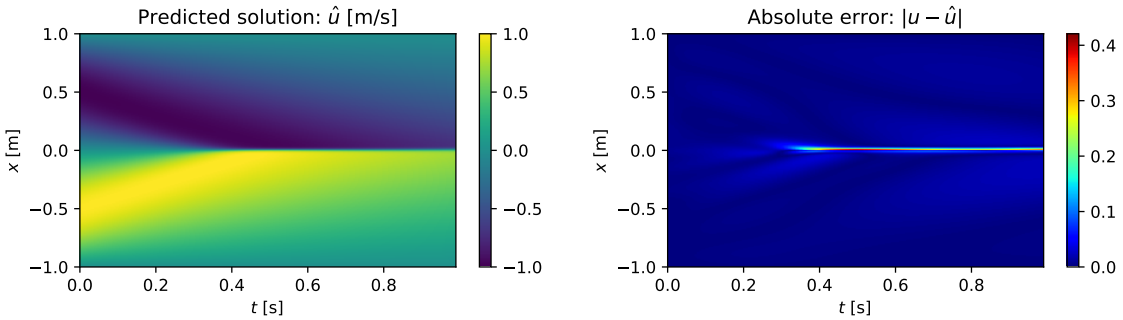

Figure 7.16: Burgers': PINN solution trained using no data and the residual network from figure 7.11. Solution error is 3.75e-2.

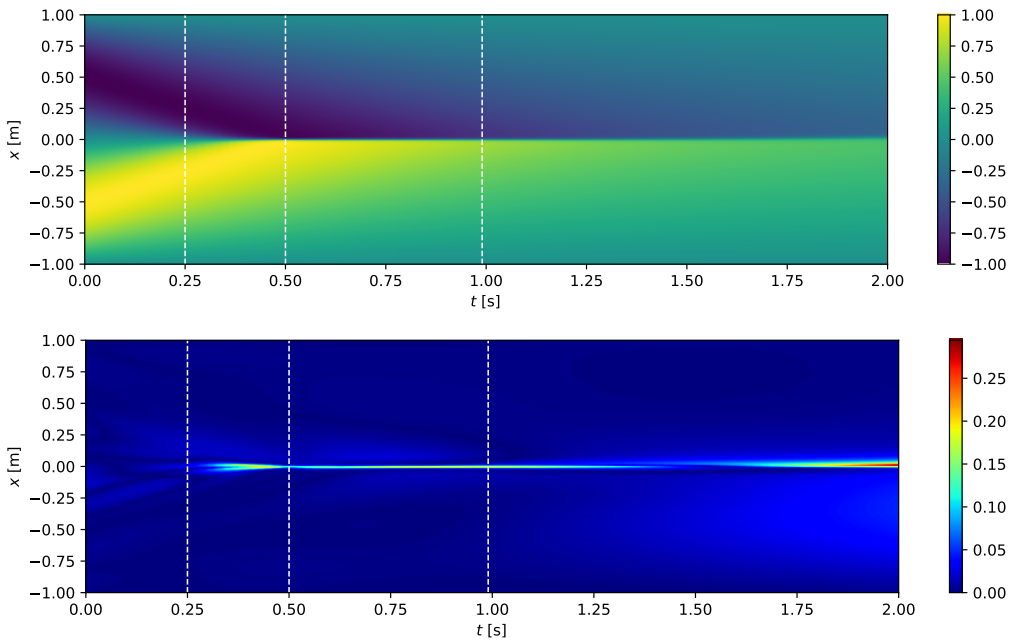

Figure 7.17: Burgers': PINN model trained on extended time domain with the residual network from figure 7.11. **Top**: Predicted solution. **Bottom**: Absolute solution error. Vertical lines indicate data. RRMSE of the solution is 4.48e-2.

As a final experiment, the PINN model was trained on an extended time domain,

from $t = 0$ to $t = 2$, using the residual network from figure 7.11 as well as the training data. The results shown in figure 7.17 illustrate that the residual network has learned the hidden dynamics well enough to generalize beyond the training data, although a small drift is observed at the end of the time domain. This was also tried without data, but resulted in much more severe drift of the interface.

7.3.4 Recovering the convection term with PDE-find

This section experiments with applying PDE-find to the residual network of UPINN models to construct equations of the unknown convection term in Burgers' equation.

As little difference in performance was observed on noisy data for different architectures, a FNN with 8 hidden layers of 20 neurons each will also be utilized here. Sobol sampling is used over lhs as they showed similar results in the previous experiments but sobol sampling generally is faster. The models are trained for 15000 epochs using LBFGS with learning rate 0.1. No adaptive sampling techniques are used as results did not show significant improvement in performance.

The PDE-find model uses a function library containing up to 2nd order derivatives and 2nd order polynomials, see table D.4 in Appendix D. The supplied data is the output from the surrogate and residual networks evaluated at a uniform grid of 100 points across the time dimension and 256 points across the spatial dimension, for a total of 25600 sampled points.

As a sanity check, to justify training a UPINN model, PDE-find was applied directly to the observed data. Note that PDE-find is purely data-driven, and does not utilize the known dynamics of the system, hence its task in this setting is to discover the entire PDE. The case of 3 dataslices yielded $-0.621u$ through manual thresholding. For a single data slice, PDE-find does not work. This confirms that directly applying PDE-find on the observed data is insufficient for discovering the convection term in Burgers' equation.

Noiselevel\Datapoints	[0.25, 0.50, 0.99]	[0.50]
0e-0	$0.638uu_x$	$0.623uu_x$
1e-2	$0.625uu_x$	$0.648uu_x$
1e-1	$0.643uu_x$	$0.569uu_x$
1e-0	$0.619uu_x$	$0.508uu_x$

Table 7.16: PDE-find identified equations for the residual network with inductive bias.

Table 7.16 show the results of applying PDE-find to the residual network with inductive bias. While none of the identified equations in have the correct scalar

value, they all have the correct algebraic form. This indicates that UPINN discovered the general form of the hidden dynamics accurately enough for PDE-find to recover the convection term. A suprising result is that the model is able to recover the convection term with 100% added noise.

In [8] they suggest imposing an inductive prior on the residual network, by directly supplying u, u_x and u_t as inputs, which was also the assumption throughout the experiments above. To test if this bias is important, the experiment is repeated without inductive bias. Thus, only u, x and t are provided as inputs to $\hat{F}_{\theta_F}$. In this setting the model must itself approximate the differential operator from the solution and spatial information. The results are shown in table 7.17.

Noise\Data	[0.25, 0.50, 0.99]	[0.50]
0e-0	$0.619uu_x$	$0.549uu_x$
1e-2	$0.631uu_x$	$0.002u_{xx} + 0.201uu_x$
1e-1	$0.578uu_x$	$0.068u_x + 0.003u_{xx}$
1e-0	$0.002u_{xx} + 0.137uu_x$	$0.141u + 0.143u_x + 0.003u_{xx} - 0.014uu_x$

Table 7.17: PDE-find identified equations for the residual network without inductive bias.

Without inductive bias, it is seen that the discovered equations are less accurate, particularly in sparse and/or noisy settings. However, many of the discovered equations still have the correct form.

Section 6.3 showed that a regular PINN can perform model discovery on the viscosity parameter ν from sparse data. Provided the found expression has the correct form, this technique may also be employed to estimate the correct scalar value. Table 7.18 shows the estimated scalar values for the discovered equations in table 7.16. With this additional step, the convection term has been accuratly recovered by utilizing the known dynamics and available data.

Noise\Data	[0.25, 0.50, 0.99]	[0.50]
0e-0	9.99e-1	9.95e-1
1e-2	9.99e-1	9.96e-1
1e-1	9.96e-1	9.91e-1
1e-0	9.45e-1	8.76e-1

Table 7.18: PINNs identified scalar values for the found terms in table 7.16. The PINN was trained as in section 6.3, but to estimate the convection scalar rather than ν.

7.4 Navier-Stokes with hidden viscosity

This section considers Navier-Stokes equations from section 6.4 with the assumption of unknown viscosity terms. Formally,

$$\mathcal{N}_K = \begin{cases} u_t + \lambda_1(uu_x + vu_y) + p_x \\ v_t + \lambda_1(uv_x + vv_y) + p_y \end{cases} , \quad \mathcal{F} = \begin{cases} -\lambda_2(u_{xx} + u_{yy}) \\ -\lambda_2(v_{xx} + v_{yy}) \end{cases} . \tag{7.5}$$

This problem is computationally more demanding than the previous ones, with a 3 dimensional input space (temporal and 2D spatial), 3 outputs and evaluation of 1st and 2nd order derivatives.

7.4.1 Improve convergence rate

Before assessing UPINNs ability to approximate PDE residuals for this problem, it is worth investigating if the convergence can be improved by means of reduced computational cost or achieve lower or comparable loss quicker.

The used baseline model is a FNN with 8 hidden layers of 20 neurons each, using Tanh activation function and Normal Xavier Glorot initialization. Training is performed on 10000 sobol collocation points and 2000 (pseudo) randomly sampled datapoints. LBFGS optimizer is used with learning rate of 1 until convergence, defined as LBFGS not being able to find an improving step.

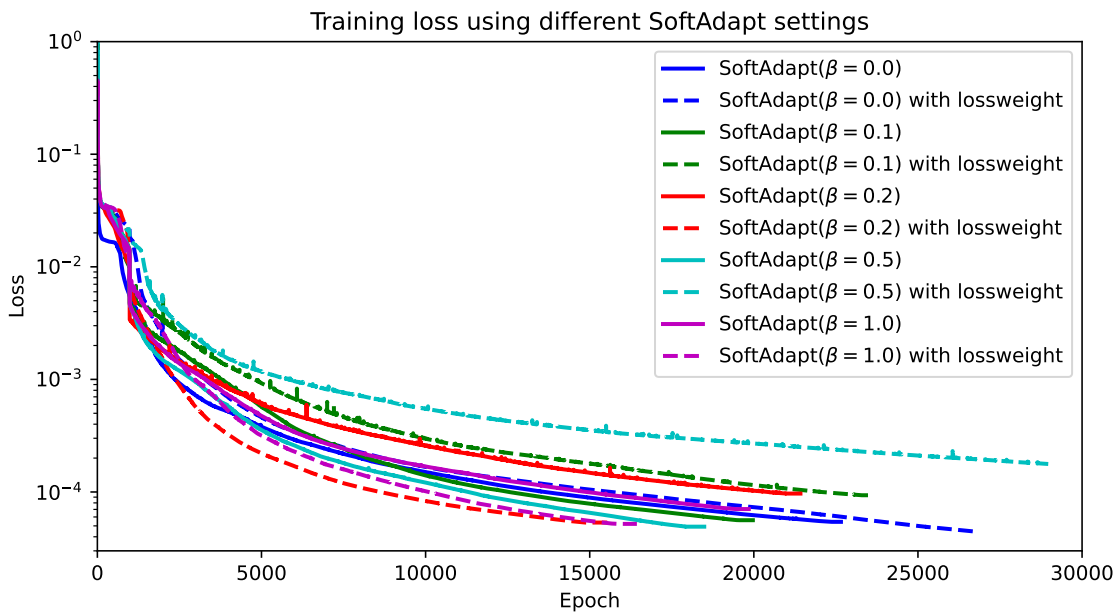

Figure 7.18: Navier Stokes: Loss evolution using different SoftAdapt settings. Note that $\beta = 0.0$ without loss weighting is equivalent to not using softadapt.

First, it is tested if loss weight balancing, in the form of SoftAdapt, can improve

convergence rate. The results are presented in figure 7.18, which shows that using SoftAdapt with $\beta = 0.2$ and loss weighting has the fastest convergence rate while also achieving low loss.

Next, it is tested if other model architectures have better convergence. In addition to a wider FNN, ResNet and blocked networks are tested. A blocked network refers to using seperate NNs for each output of u and/or F. KANs were also tested, but even a small one with 4 layers of width 4 took several times longer to converge than the others, and never reached a loss better than 1e-3. Subsequently, KANs were disregarded early in these experiments. The results are shown in figure 7.19. For commparison, the best performing model from figure 7.18 as well as the best performing ResNet with SoftAdapt are included. Although the wider networks with hidden size 40 took the longest to converge, they achieved the lowest loss. Note that the wider networks (FNN and ResNet) performed worse when using softadapt. The fastest converging architecture was the baseline model using softadapt with $\beta = 0.2$ and loss weighting.

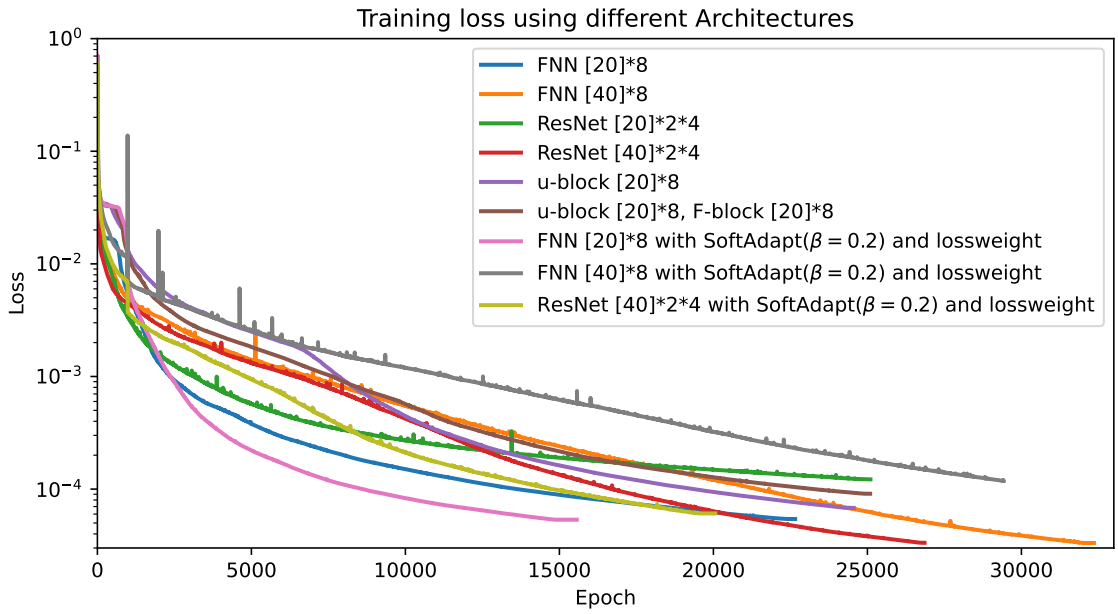

Figure 7.19: Navier Stokes: Loss evolution for different NN architectures.

7.4.2 Enhance residual modelling

Having established a model with good convergence, the next step is to investigate UPINNs ability to model the residuals (equation 7.5) of the Navier-Stokes equations. To keep the residual network input simple, no inductive prior is imposed on its input. That is, only the spatiotemporal information (x, y, t) and surrogate output (u, v, p) are passed to $\hat{F}_{\theta_F}$, thus no partial derivatives are supplied and must be learned by the network.

As a first experiment, the UPINN framework is applied using 2000 (pseudo) randomly sampled data points on the velocity field, corrosponding to 0.2% of the entire dataset, and 10000 sobol sampled collocation points. No boundary conditions. As in section 6.4, the solution is visualized by means of the preassure p and vorticity $\omega = v_x - u_y$ at $t = 10$, see figure 7.20. The results of the previous section showed that a FNN with 8 hidden layers of 20 neurons each and Tanh activation function is a fast converging model, hence this will also be the baseline model for this experiment.

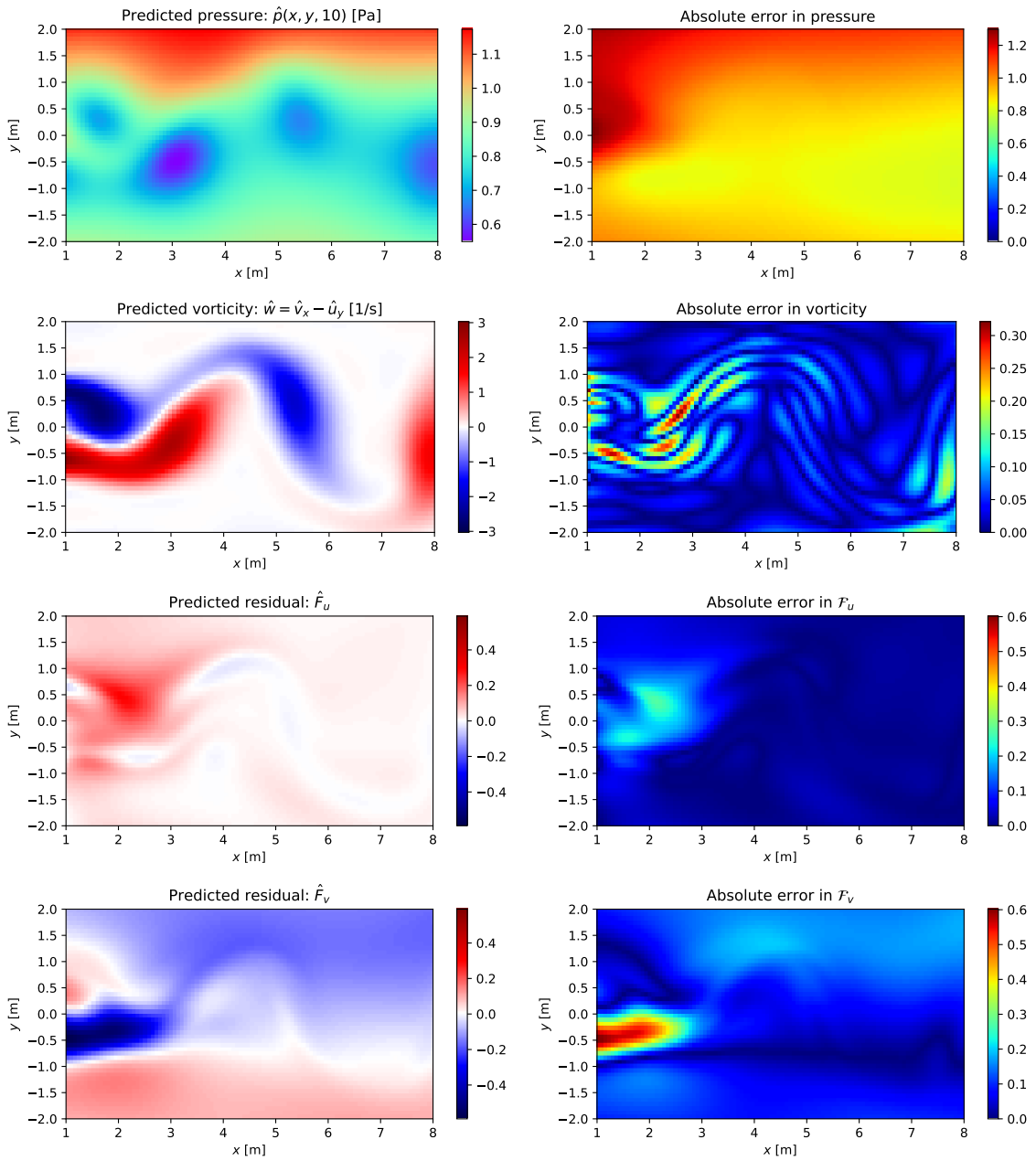

Figure 7.20: Navier Stokes: UPINN prediction of pressure, vorticity and PDE residuals at $t = 10$. Trained without inductive bias using 2000 datapoints from the velocity field. Solution RRMSE: 1.09e-2, Residual RRMSE: 6.64e-0.

Figure 7.20 shows that the model captures the vorticity well, but only the top part of the pressure field resembles the true pressure field (to an additive constant) and PDE residuals do not appear to be accurate.

Inspecting the UPINN formulation 7.5 provide a possible explaination for the descrepancy in the estimated pressure. The pressure gradients appear as isolated terms in the PDE system, just like the residual network, but are not constrained by data or additional conditions apart from the PDEs. Consequently, $\hat{u}_{\theta_u}$ is free to model the pressure such that it compensates for the PDE residuals. To check this, the sum of the pressure gradient and residual network output is visualized in figure 7.21. These quantities are well approximated, with RRMSE of 8.60e-3 and 6.44e-3 for the first and second equation respectively.

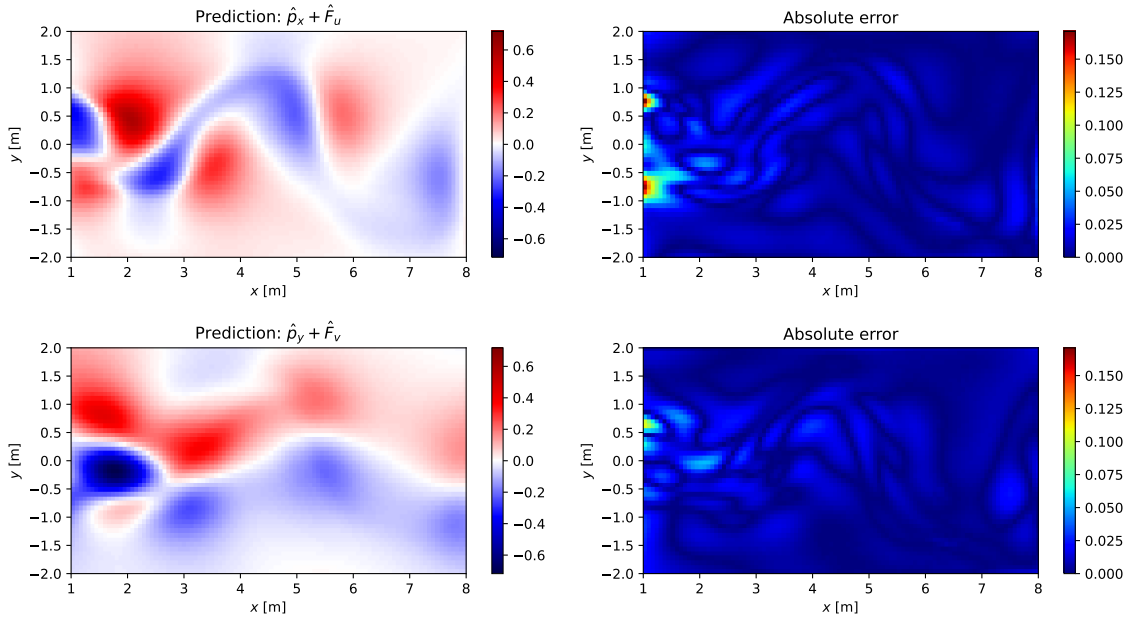

Figure 7.21: Navier Stokes: Sum of UPINN predicted pressure gradient and PDE residuals at $t = 10$, when not trained on pressure data.

If the pressure field is not of interest, figure 7.20 may be considered an acceptable result. However, these observations raise two questions worth investigating. First, does knowning the viscosity term of one of the equations sufficiently constrain the model to seperate the pressure field and PDE residuals of the other? Second, does including the pressure at the observed data points constrain it from being modelled by the residual network, and vice versa?

Regarding the first question, the issue persists, although to a smaller degree. Even with the first equation fully known, the pressure is still seen to compensate for residuals of the second equation. Consequently, the residual network models

the error of the incorrectly estimated pressure in the first equation. The results are shown in figure C.5 (see Appendix C).

As for the second question, figure 7.22 shows the resulting predicted pressure and residuals when including the pressure in the observed data points. The pressure is now well estimated by the model and the residual network approximate the hidden terms with significantly lower error.

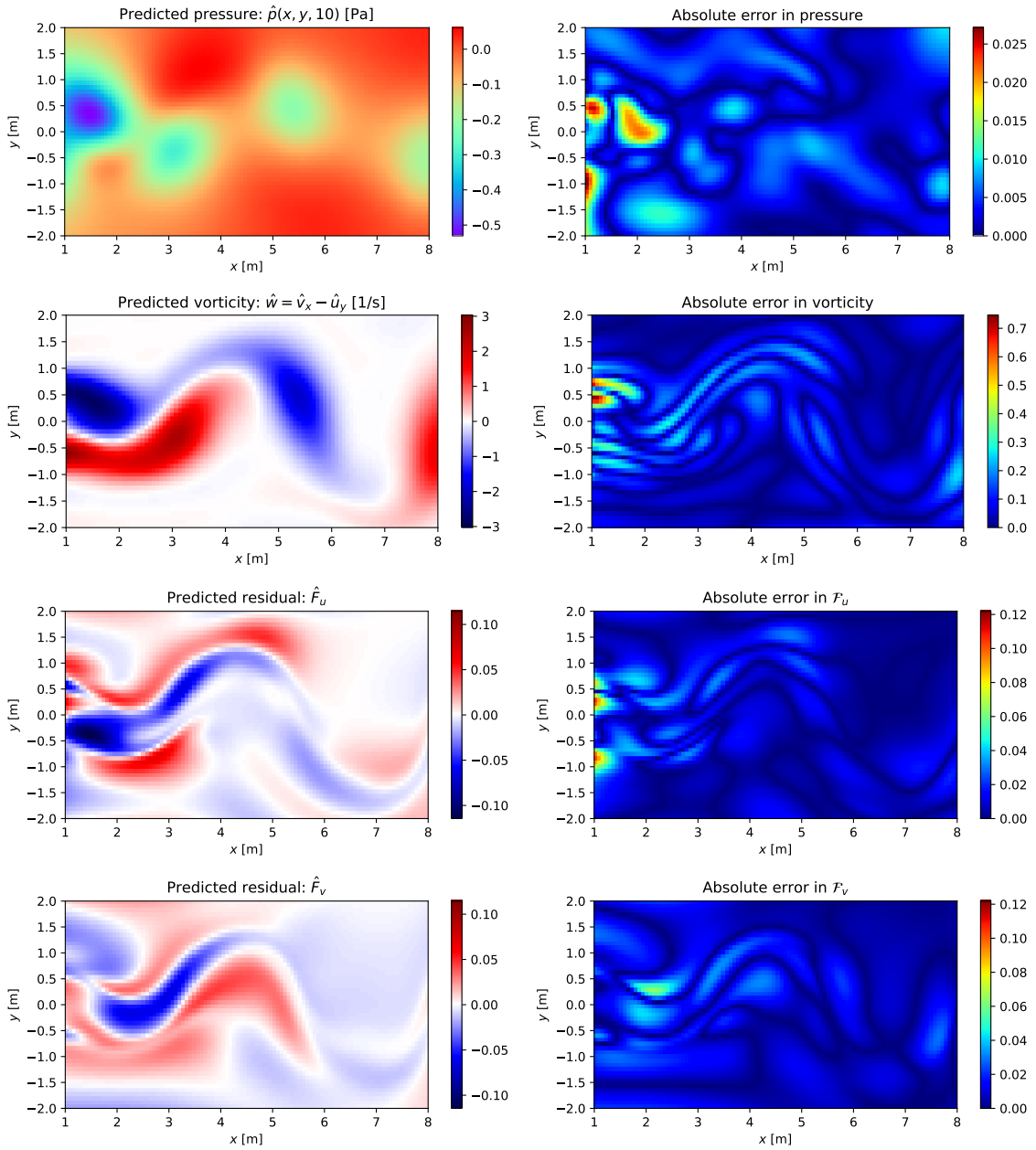

Figure 7.22: Navier Stokes: UPINN prediction at $t = 10$ using pressure at observed data points. Top 2: Pressure field and vorticity, Bottom 2: Predicted PDE residuals. Solution RRMSE: 1.49e-2, Residual RRMSE: 9.04e-1.

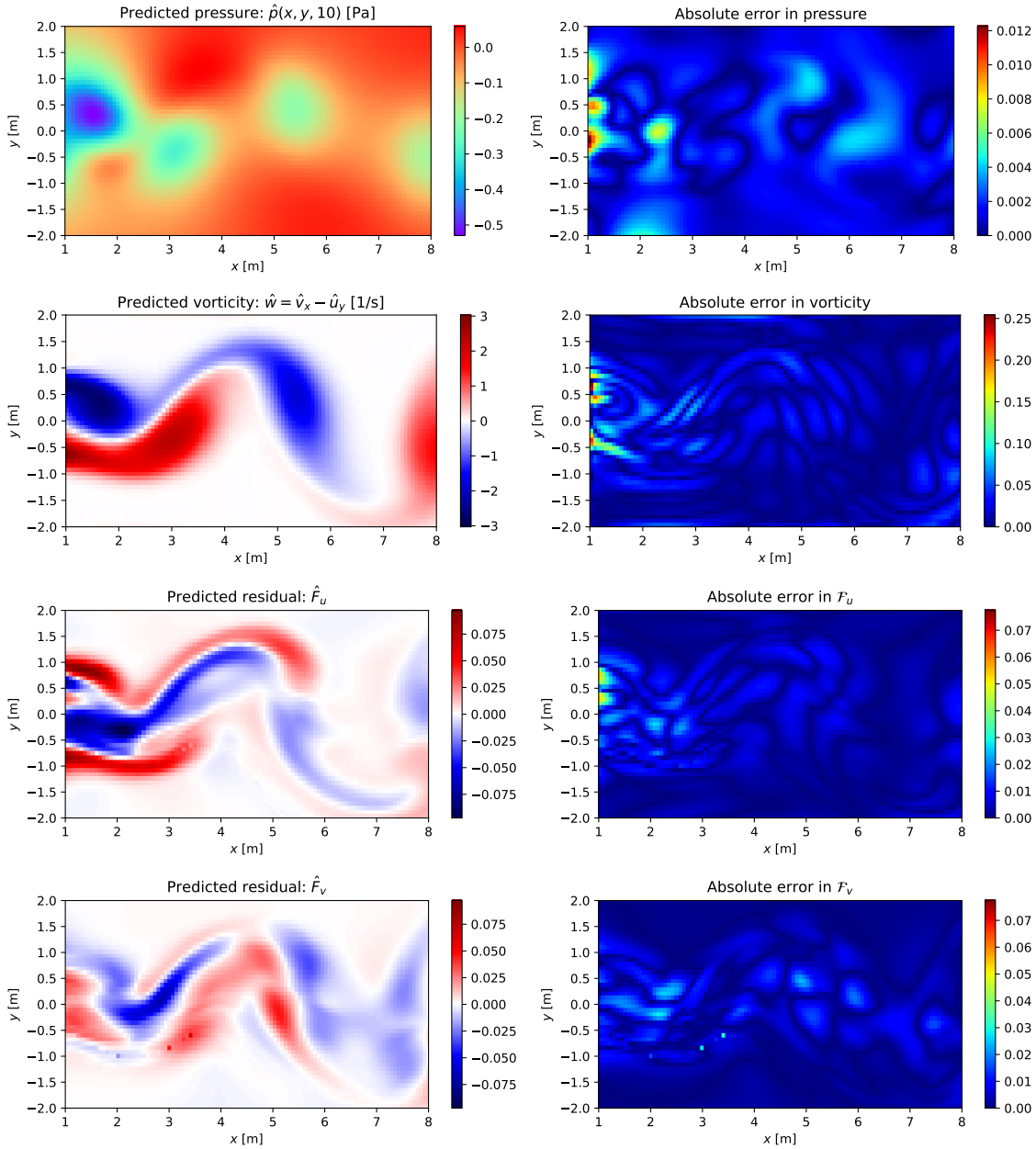

Figure 7.23: Navier Stokes: UPINN predictions when using RAD and SoftAdapt($\beta = 0.2$) with loss weighting on a [64]*8 FNN. Top 2: Pressure and vorticity field. Bottom 2: Predicted residuals of the first and second PDE. All at $t = 10$. Solution RRMSE: 5.68e-3, Residual RRMSE: 3.54e-1.

In the Kármán vortex street shedding pattern behind the cylinder modelled by (7.5), steep gradients are present. Thus, adaptive sampling might help increase collocation point resolution in these areas and improve model performance. The following training strategy is employed; 30 loops of 1000 Adam epochs with learning rate 1e-3 followed by 1000 LBFGS epochs with learning rate 0.1 and RAD resampling, ending with Adam for 40000 epochs with a learning rate scheduler scaling with 0.9 every 1000 epochs to stabilize Adam as the loss decreases. Sof-

tAdapt with $\beta = 0.2$ and loss weighting is still used. As figure 7.19 indicated that wider networks can obtain lower loss, an 8 layer FNN with 64 neurons in each layer is used with Tanh activation function. The results are presented in figure 7.23 and table 7.19. RAR and RAR-D was attempted, but as these methods keep increasing the number of collocation points and evaluating the PDE residuals requires 1st and 2nd order derivatives, the computational cost became too high. Starting with a smaller number of collocation points for these methods yielded poor results.

For formal comparison, the RRMSE of the predicted states and residuals with respect to the true solution are listed in table 7.19. By this metric, the wide FNN with 8 layers of 64 neurons each trained using RAD, is almost an order of magnitude more accurate than the other models. Thus, this model will be used for symbolic regression in section 7.4.3. Like figure 7.19, this shows that wider networks slighty increase performance. However, combining wider networks with RAD and SoftAdapt using loss weighting is seen to provide the most significant improvements.

Model	**Solution error**	**Residual error**	**Notes**
FNN [20]*8 (*)	1.09e-2	6.64e-0	Figure 7.20.
FNN [20]*8	1.49e-2	9.04e-1	Figure 7.22.
FNN [20]*8	1.43e-2	8.51e-1	RAD, SoftAdapt($\beta = 0.2$) with loss weighting. Figure 7.18.
FNN [40]*8	1.24e-2	8.03e-1	Figure 7.19.
FNN [40]*8	6.60e-3	6.17e-1	RAD, SoftAdapt($\beta = 0.2$) with loss weighting. 7.19.
FNN [64]*8	**5.68e-3**	**3.54e-1**	RAD, SoftAdapt($\beta = 0.2$) with loss weighting. Figure 7.23.
ResNet [20]*2*4	2.03e-2	9.73e-1	Figure 7.19.
ResNet [40]*2*4	1.29e-2	7.41e-1	Figure 7.19.
ResNet [40]*2*4	9.55e-3	6.98e-1	RAD, SoftAdapt($\beta = 0.2$) with loss weighting. 7.19.

Table 7.19: Navier Stokes: RRMSE of predicted solution and residuals for a selection of tested models. (*) No pressure data.

7.4.3 Recover viscosity terms

In this section, PDE-find is used to estimate equations for the, by UPINNs, estimated unknown dynamics in Navier Stokes equations. As the best performing model was a FNN with 8 hidden layers of 64 neurons each using RAD and SoftAdapt with $\beta = 0.2$ and loss weighting, this will be employed for PDE-find, using the same training process.

The function library contains up to 2nd order derivatives and 2nd order polynomials, see table D.5 in Appendix D. The residual network is sampled on a grid of 50 datapoints in each domain dimension, for a total of 125000 points. Fitting was done using the SR3 optimizer with L^1 thresholding.

The results are shown in table 7.20, along the RRMSE of the trained models in table 7.21. It is seen that PDE-find is able to correctly identify all 4 terms when trained on the clean data with 500 or more datapoints. While unable to identify all the terms in a 1% noise setting, it still captured 2 terms for 500 or more datapoints. Note, unlike the previous experiment in section 7.3.4, this experiment was only performed on models without the inductive bias of passing partial derivatives to the residual network, as the computational cost made it infeasible.

Data\Noise	0.0	0.01
100	$-0.037v - 0.010u_y - 0.014u_{yy} + 0.018v_{xy}$ $-0.005u_{yy} + 0.020u_{xy} - 0.005v_{xx}$	$-0.192uv_y$ $-0.117uv_y$
500	$-0.012u_{xx} - 0.010u_{yy}$ $-0.010v_{xx} - 0.009v_{yy}$	$-0.012u_{yy}$ $-0.017u_y - 0.026v_{yy}$
1000	$-0.013u_{xx} - 0.009u_{yy}$ $-0.011v_{xx} - 0.010v_{yy}$	$-0.011u_{yy}$ $-0.013v_{xx}$
2000	$-0.012u_{xx} - 0.010u_{yy}$ $-0.010v_{xx} - 0.009v_{yy}$	$-0.010u_{yy}$ $-0.013v_{xx}$

Table 7.20: Navier Stokes: PDE-find identified equations on the UPINN predicted residuals.

Datapoints	**Solution error**	**Residual error**
100	(5.12e-2 / 1.08e-1)	(2.53e-0 / 4.42e-0)
500	(1.17e-2 / 1.96e-2)	(6.52e-1 / 1.29e-0)
1000	(8.34e-3 / 1.05e-2)	(6.09e-1 / 8.51e-1)
2000	(3.68e-3 / 6.84e-3)	(3.54e-1 / 7.13e-1)

Table 7.21: Navier Stokes: RRMSE (Clean / 0.01 Noise) of models trained for symbolic regression experiment presented in table 7.20.

This experiment concludes the work on UPINNs for this thesis.

8 Tokamak plasma experiment

8.1 Problem description

This thesis is primarily an exploratory study on idealized toy problems. However, if some of these techniques can be applied to real-world problems, that would be of great interest.

In an attempt to do so, the real-world problem of modelling plasma transport in a tokamak reactor is considered. The concrete problem arise from the work of [58], which also provided the data used in this chapter. Formally, the problem is given as a continuity PDE,

$$\frac{\partial \mathcal{V}}{\partial \rho}\frac{\partial n_e}{\partial t} = \frac{\partial}{\partial \rho}\frac{\partial \mathcal{V}}{\partial \rho}\left(\langle(\nabla\rho)^2\rangle D\frac{\partial n_e}{\partial \rho} - \langle\nabla\rho\rangle V n_e\right) + \frac{\partial \mathcal{V}}{\partial \rho}\left(S - \frac{n_e}{\tau_{||}}\right), \tag{8.1}$$

where ρ is the radial coordinate, t is time, $\mathcal{V}$ is the plasma volume, n_e is the electron density, $\langle(\nabla\rho)^2\rangle$ and $\langle\nabla\rho\rangle$ are flux surface averaging terms, D is the electron diffusion coefficient, V is the convection velocity, S is the particle source term, and $\tau_{||}$ is the parrallel confinement time.

Additionally, 2 boundary conditions are given. The first is a Dirichlet condition at the plasma edge, $\rho = \rho_{\text{max}}$, where $n_e(t, \rho_{\text{max}}) = 0$. The second is a Neumann condition at the center of the plasma, $\rho = 0$, where the flux term is zero,

$$\frac{\partial \mathcal{V}}{\partial \rho}\left(\langle(\nabla\rho)^2\rangle D\frac{\partial n_e}{\partial \rho} - \langle\nabla\rho\rangle V n_e\right)\Bigg|_{\rho=0} = 0. \tag{8.2}$$

The purpose of this experiment is to apply PINNs to (8.1) and test if it can model the profiles of D, V and S using the available data. If this can be done accurately, it may be feasible to apply UPINNs to model hidden dynamics of (8.1) in the plasma.

8.2 Data

The dataset consists of discrete observations of all quantities in (8.1) at 105 radial coordinates and 50 time points. Table 8.1 provides an overview of the dataset.

Data	Shape	Notes
ρ	(105,)	Radial coordinate.
t	(50,)	Time.
$\frac{\partial \mathcal{V}}{\partial \rho}$	(105,)	Plasma volume expansion.
$\langle (\nabla \rho)^2 \rangle$	(105,)	Flux surface averaging term.
$\langle \nabla \rho \rangle$	(105,)	Flux surface averaging term.
D	(105, 50)	Electron diffusion coefficient.
V	(105, 50)	Convection velocity.
S	(105, 50)	Particle source term.
$\tau_{\|\|}^{-1}$	(105, 50)	Inverse parallel confinement time.
n_e	(105, 50)	Electron density.

Table 8.1: Tokamak plasma: Data used in the experiment.

8.3 PDE residuals of given solution

The core idea that makes PINNs a powerful tool, is the use of PDE residuals to guide the training process towards a solution satisfying the PDE. Thus it is crucial, in order to obtain a model resembling the true solution, that the residuals are of good and stable quality. Figure 8.1 shows a uniform set of time slices of the PDE terms in (8.1) for the given data, along the normalized residuals.

An immediate observation is that, for $\rho > 0.8$, the right hand side terms are several orders of magnitude larger than the left hand side term. This can potentially cause instability issues in the training process. The normalized residuals appear to be much more well behaved, but do still have large spikes. This indicates that training on the normalized PDE residuals may offer a better approach than training on the absolute PDE residuals.

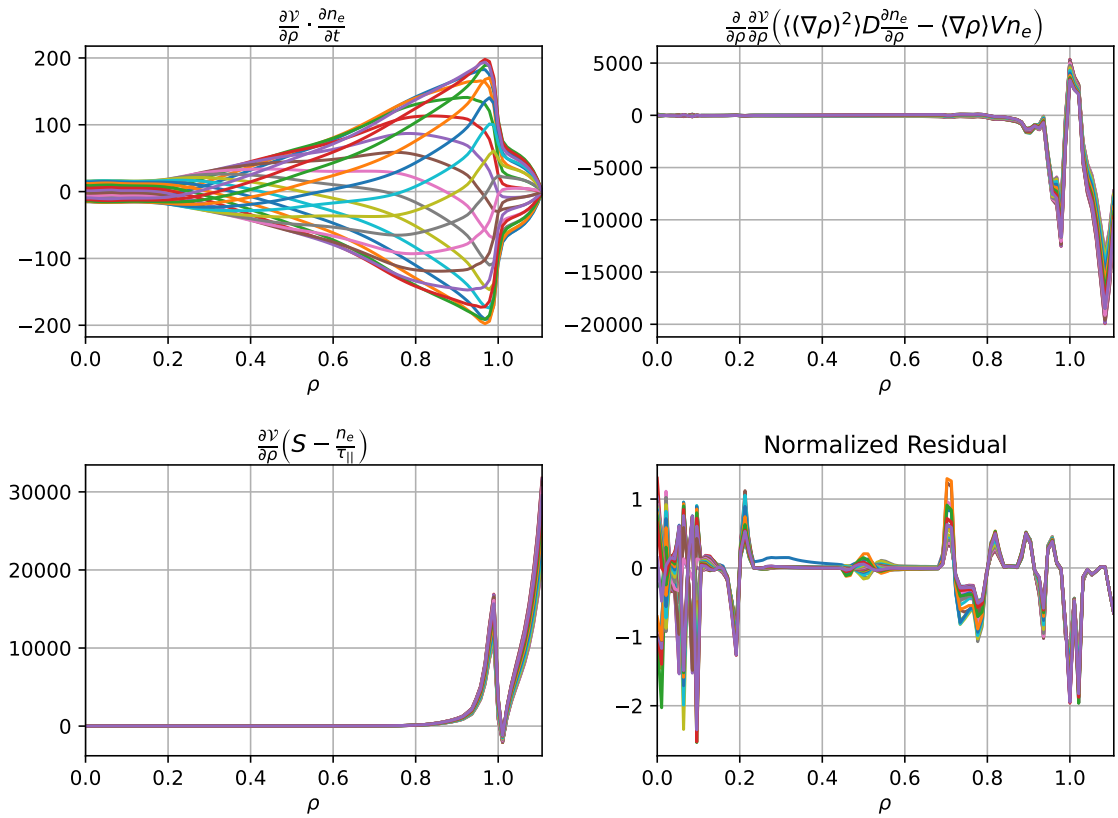

Figure 8.1: Tokamak plasma: PDE residuals of the given solution.

8.4 Model setup

A central advantage of PINNs is the ability to use densly sampled collocation points to evaluate the PDE residuals across the entire domain. However, the geometric terms of the PDE in (8.1) are only given on a discrete grid in the domain, which also make computing gradients with automatic differentiation challenging. To overcome this, these terms are approximated with FNNs with 2 layers of 20 neurons each and Tanh activation function. This allows for gradient computation and continuous evaluation of the PDE residuals. The neural network approximations can be seen in figure C.6 in Appendix C.

The loss in this model consists of 2 terms, the PDE loss and the zero flux loss at the center of the plasma, $\rho = 0$. The data loss remain constant as it is approximated already by the above mentioned FNNs.

The models will use FNNs with 2 hidden layers of 20 neurons each and Tanh activation function, for each of the profiles of D, V and S. Training is done for 100000 epochs using Adam optimizer with learning rate 1e-3.

8.5 Results

The initial attempt to train a PINN to model the profiles of D, V and S, was done by training directly using the absolute residuals of the PDE. As mentioned and indicated by the terms in figure 8.1, this approach did not yield good results. The top of figure 8.6 shows that the training process is dominated by the PDE loss, and quickly converges to a solution that does not resemble the true solution, except for a small dip in the V profile at the plasma edge as seen in figure 8.2.

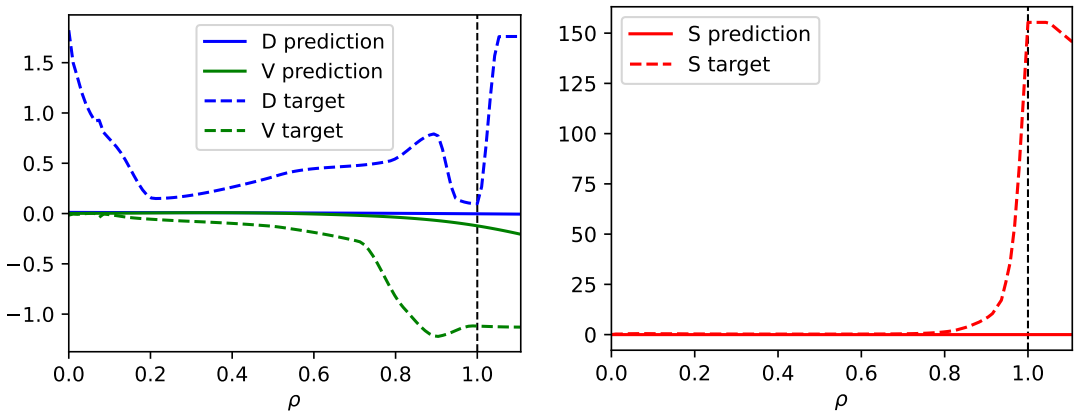

Figure 8.2: Tokamak plasma: Training on absolute residuals. **Left:** Predictions of D and V profiles. **Right:** Prediction of S profile. Black vertical line indicate the plasma edge.

To improve the training process, the PDE residuals were normalized and training was done on those instead. While this normalization trivially lowers the loss below 1, the middle panel of figure 8.6 shows that the training process is still dominated by the PDE loss which barely decreases. However, as seen in figure 8.3, the predicted profiles of D and V have some structure that resembles the true solution, whereas the predicted S profile remains flat at the plasma edge.

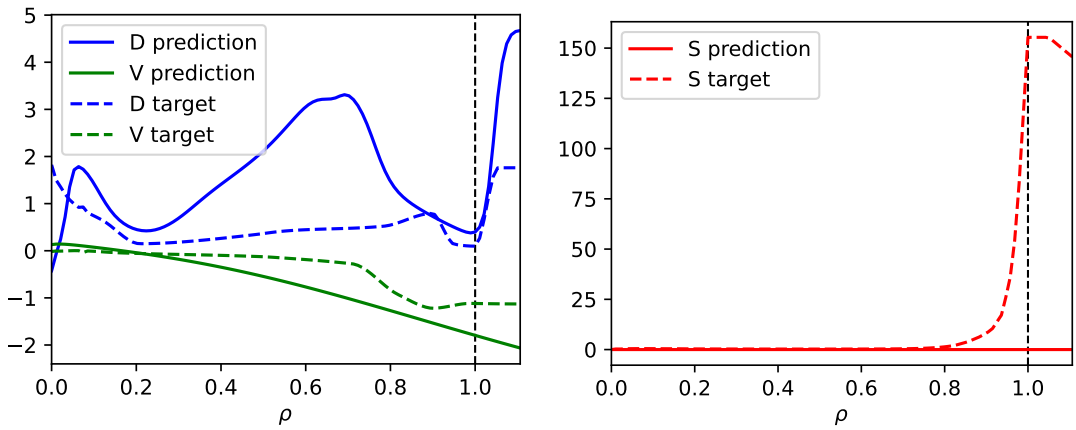

Figure 8.3: Tokamak plasma: Training on normalized residuals. **Left:** Predictions of D and V profiles. **Right:** Prediction of S profile. Black vertical line indicate the plasma edge.

To prevent the PDE loss from dominating the boundary flux loss, SoftAdapt with $\beta = 1$ was applied, both without loss weighting (figure 8.4) and with loss weighting (figure 8.5). As the bottom panel of figure 8.6 shows, this helped balance the loss terms and allowed the total loss to significantly decrease. It is also noticed that applying loss weighting makes the covergence less stable. Inspecting the resulting profile predictions in figures 8.4 and 8.5, it is seen that the profiles of D and V still retain some structure of the true solution, but notably, the S profile now shows promising structure as well.

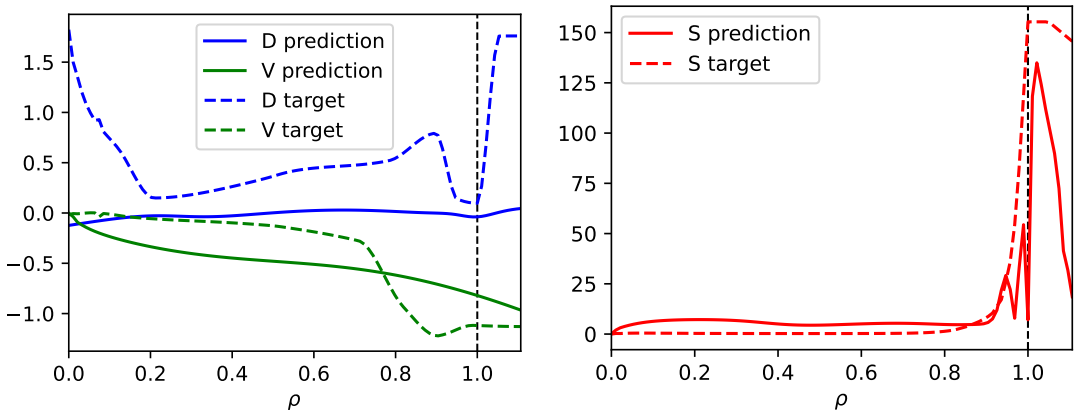

Figure 8.4: Tokamak plasma: Training on normalized residuals using SoftAdapt($\beta = 1$). **Left:** Predictions of D and V profiles. **Right:** Prediction of S profile. Black vertical line indicate the plasma edge.

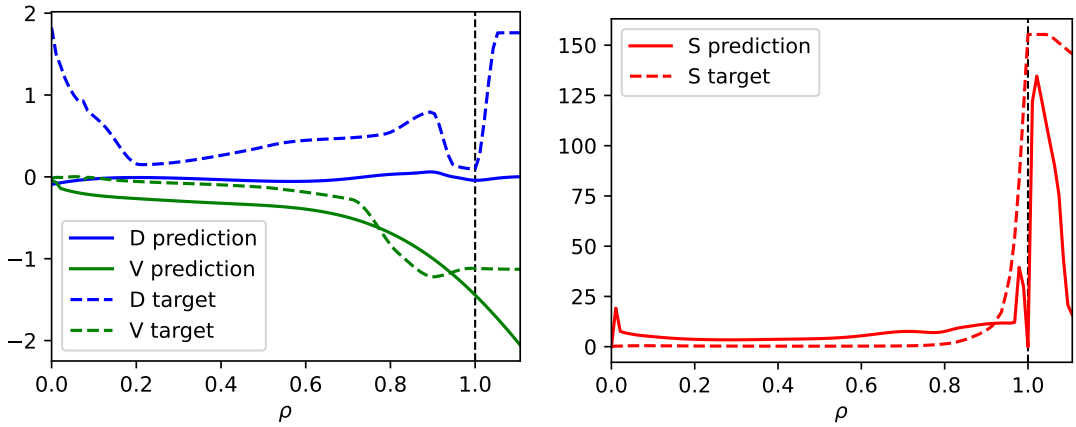

Figure 8.5: Tokamak plasma: Training on normalized residuals using SoftAdapt($\beta = 1$) with loss weighting. **Left:** Predictions of D and V profiles. **Right:** Prediction of S profile. Black vertical line indicate the plasma edge.

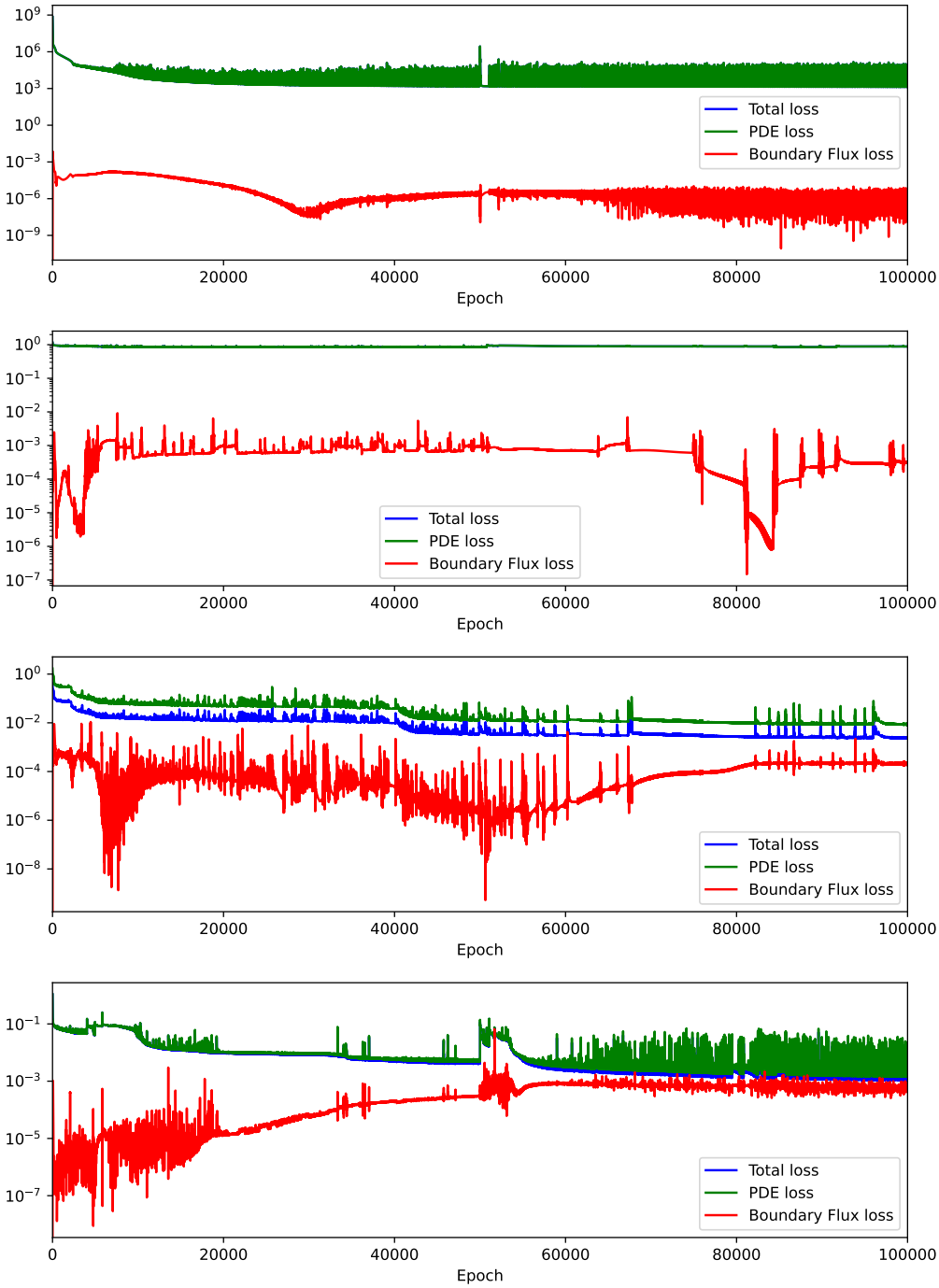

Figure 8.6: Tokamak plasma: Loss curves for different training strategies. **Top:** Training on the absolute residuals. **Upper middle:** Training on the normalized residuals. **Lower middle:** Training with SoftAdapt($\beta = 1$). **Bottom:** Training with SoftAdapt($\beta = 1$) and loss weighting. Note, the blue curves represent the total loss lies behind the green curves of the PDE loss.

The results presented above are a promising start to solving the problem. However, it seems appropriate to test if the current formulation of the problem is even feasible to be solved by a PINN. To do this, neural networks are trained to approximate the true solution of D, V and S using the available data. These are then

used as initial networks in the PINN training process. If the formulation allow a PINN to converge towards the true solution, this experiment is expected to yield good results as it starts at an assumed optimum. Training was done as for the best performing model of the previous experiments, using SoftAdapt($\beta = 1$) and loss weighting. The resulting profiles are shown in figure 8.7.

While the solution networks did not remain perfectly aligned with the true solution, they appear to have gotten stuck in a local minima that do still resemble the true solution. The reason for denoting it a local minima, is that the PDE loss is significantly larger than in the other experiments that did not use the true solution as an initial guess.

This indicates that there exist some structure around the true solution that the PINN is able to find, but the current formulation of the training process does not make it a global, or even reasonable local, minima that the model can converge to from a random initialization.

Nevertheless, with further investigation and tuning of the training process, it might be possible to augment the loss landscape and obtain better results for this problem with a PINN. Unfortunatly, due to time constraints, this is where the work on this problem ends for this thesis.

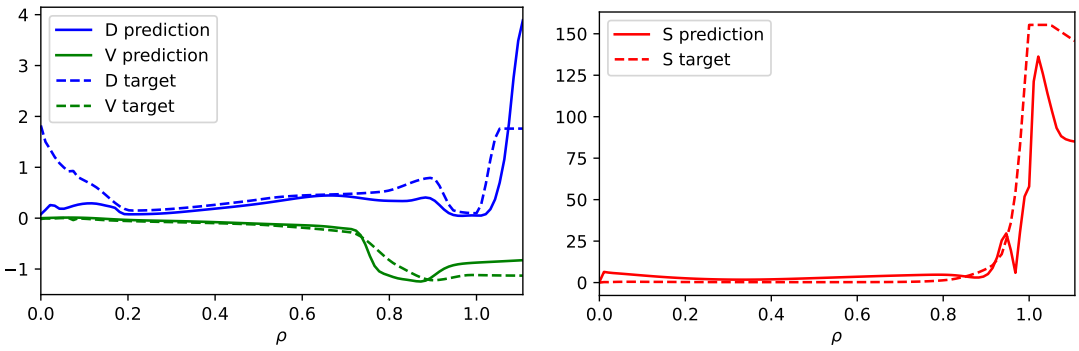

Figure 8.7: Tokamak plasma: Results of using the PINN framework on FNNs for D, V and S initialized as good approximations of the true solution.

9 Discussion of results

UPINNs are a novel deep learning approach to hidden discovery problems, combining the concepts of physics-informed neural networks and universal differential equations. By utilizing data, UPINNs solve incomplete differential equations while simultaneously estimating the missing terms.

Having conducted a range of experiments with UPINNs on a variety of problems, this chapter discuss the obtained results and their implications. While UPINNs showed promising results in a variety of settings, certain limitations emerged that may be addressed in future work.

9.1 Undesired estimation

Throughout chapter 7, it became evident that UPINNs have a tendency to make undesired estimation. This would often not affect the estimated solution, but could lead to inaccurate parameter estimates, unnecessary complexity in model residuals and hinder interpretability using symbolic regression frameworks.

UPINNs, in their general formulation, are highly flexible, which comes at the risk of estimating known dynamics, if not sufficiently constrained. Subsequently, UPINNs should be used with caution and careful consideration as to prevent these issues. As shown, incorporating inductive biases on the hidden dynamic structure and leveraging additional data can help mitigate these challenges. If left completely unconstrained, the residual network can overfit the PDE loss, artificially lowering it and halt training, rather than capturing the true hidden dynamics.

9.1.1 Undesired estimation of known dynamics

In section 7.1.3, it was seen that UPINNs ability to simultaneously solve hidden discovery and inverse problems simultaneously depends on the form of the unknown dynamics. The results indicated that insufficiently constrained systems allow UPINNs to estimate dynamics that are already known, while terms with unknown parameters may only be partially estimated. Consider splitting the first term of the first equation in the Lotka-Volterra system into two fractional terms,

$$\frac{dx}{dt} = \alpha_1 x + \alpha_2 x - \beta xy, \quad \alpha = \alpha_1 + \alpha_2.$$

As α is unknown and treated as a trainable parameter, the residual network is free to estimate $\alpha_2 x - \beta xy$ rather than just $-\beta xy$. Section 7.1.6 confirmed existence

of this behavior with results from SINDy, which was able to extract the part of the known dynamics estimated by the residual network. While this does not prevent obtaining a solution network with low error, it is undesired behavior, leading to inaccurate parameter estimates that are not unique, while the residual network becomes unnecessarily complicated and less interpretable. This also explains why fixing δ in the second equation and using a shared residual network allowed accurate estimation of α. By constraining the residual network to model γxy, the shared structure naturally enforce βxy in the first equation, ensuring that α can be accurately estimated.

It was tried adding a regularization term on the residual network output in the loss, which improved the results when no residuals were present but worsened them otherwise. Small residuals are not necessarily desirable, as they may arise naturally when modelling known terms with unknown parameters. The overall issue is that no constraints are in place to inform the network of existing terms with unknown parameters. No solution was found to this issue, but it is a clear limitation of the current UPINN formulation.

9.1.2 Phantom estimation

In the section 7.2, it was seen that a UPINN model which trivially enforced net zero flow in an SIR model with resusceptibility, estimated a flow that was not present in the data. However, the solution network still remained accurate, suggesting that the residual network estimated a ”phantom” flow - a flow which does not exist, but does not change the solution either.

This may be explained by considering how the dynamics of an SIR model would respond to an artificial flow from R to S and I. It will inevitably result in an increased number of susceptibles and infected, inducing natural flows from S to I and I to R by the known dynamics, cancelling out the artificial flow and leaving the overall dynamics unchanged. While regularizing the residual network output did not reduce estimation of known dynamics in the Lotka-Volterra system, it did reduce the phantom flow in the SIR model.

Although this behavior was only observed in the SIR model, it could potentially occur in other systems with several equations as well. This is a clear example of the residual network performing undesired estimations that do not affect the solution.

9.1.3 Residual estimation by solution network

In the Navier Stokes experiments of section 6.4, it was seen that the model struggled to estimate the pressure when not constrained by data. It became evident that the residual network was partially estimating the pressure term. This is perhaps not surprising, as the pressure was unconstrained by data and its partial derivatives appeared in the PDEs as free terms, which is also the case for the residual network. Subsequently, the residual network prediction and pressure part of the solution network got entangled, leading to wrongly estimated pressure and residuals, allbeit, they were collectively accurate which allowed the velocity field to be estimated accurately.

This further emphasizes the need for additional constraints on the known dynamics to prevent the residual network from estimating them.

9.2 Inductive bias

The UPINN results presented in [8] used inductive priors on the residual networks. That is, assumptions regarding the structure of the hidden dynamics were imposed on the model.

For the Lotka-Volterra system, the assumption is that the hidden dynamics are proportional between the two equations, directly coupling them through a trainable scalar parameter. Section 7.1.4 showed that this assumption consistently lowered the model error for a range of combinations of noise and sparsity levels. Further, section 7.1.3 showed that it also enabled the model to estimate one of the parameters in the known dynamics, while also accurately estimating the unknown dynamics, partially solving the issue discussed in section 9.1 by introducing a direct dependence between the hidden dynamics of the two equations.

For Burgers' equation, this included passing partial derivatives to the residual network. In section 7.3.2, it was demonstrated that a UPINN model can be trained without this inductive bias and still obtain accurate results. Subsequently, in section 7.3.3, it became apparent that such a model fails to be a surrogate of the true dynamics. This suggests that the neural networks were unable to learn the underlying differential operator, and merely served as an instance specific guide to the solution network during training. In contrast, using the suggested inductive bias allowed solving the problem with a data-free PINN and allowed extrapolation beyond the training data.

While the model without inductive bias did yield a residual network suitable for direct use in post training purposes, section 7.3.4 showed that it can be used to

recover the governing equations using PDE-find. More on this in section 9.3.

If the goal is solely to obtain the solution, inductive biases may not be necessary, and the model is still able to obtain good results. However, for extrapolation or tasks requiring knowledge of the differential operator, they can be crucial.

The motivation for testing the removal of inductive biases, is that prior knowledge of unknown dynamics can generally not be assumed available. While feasible in simpler problems, like Burgers' equation, to pass all possible first and/or second order partial derivatives to the residual network, it quickly becomes infeasible for larger problems like the Navier-Stokes equations considered in section 6.4. That system in particular, with 3 states and 3 domain dimensions, results in 9 partial derivatives of first order and 18 of second order. Computing every single one of these on all collocation points in every epoch quickly proved computationally infeasible. However, the experiments showed that good results can still be obtained by only passing the states and tempospatial positions to the residual network.

9.3 Equation recovery

Being able to model residuals using known dynamics and available data to obtain a surrogate solution is a powerful tool - recovering the governing equations, even more so. This was demonstrated for all of the tested systems, with varying degrees of success, using the SINDy and PDE-find frameworks.

In all cases, the correct equations were recovered in many of the tested scenarios. For the Lotka-Volterra system, the model using a shared residual network, i.e. inductive bias, could be used to correctly identify the equations in both sparse and noisy data. Without the inductive bias, it was still able to recover the equations but struggled with the highest noise levels and sparsest data.

A quite surprising results was recovering the correct form of the hidden dynamics in Burgers' equation using data from a single time point with 100% noise using the inductive bias model. Despite the extreme noiselevels, the structure of the sharp wave interface remained clear enough for the residual network to model it sufficiently accurate for PDE-find to determine the correct form of the equation. This was not possible without the inductive bias, highlighting its significance. Even so, without inductive bias, it was still possible to recover the equation with data from 3 time points with up to 10% noise, and also using a single noiseless time point.

Notably, the found equations of Burgers' were consistently estimated with similar scalar value, which deviated from the true value significantly. This suggests that

the data resolution at the sharp wave interface was insufficient for UPINN to accurately capture the extremity of the dynamics. However, it was accurate enough for PDE-find to recover the correct form of the equation. Furthermore, it was seen that a regular PINN can accurately estimate the true scalar value, ultimately leading to an accurate recovery of the equation.

Navier Stokes equations posed a more challenging problem - attempting the recover a total of 4 second order terms in the two equations, without inductive bias. Even so, the correct equations were recovered on noise-free data in all tested scenarios except the most sparse with 100 datapoints. Further, some of the terms could be recovered with 1% noise with 1000 and 2000 datapoints, but never all of them. With further refinements to the model, training process and tuning of PDE-find, it may be possible to recover the correct equations in all scenarios.

Collectively, these experiments showed that UPINNs have the potential to facilitate physics informed data-driven discovery of governing equations in sparse and noisy data settings, where fully data-driven methods may struggle.

9.4 Future work

The experiments conducted in this thesis have only scratched the surface of UPINNs potential, and the results have indicated multiple directions for future work.

The perhaps most immediate would be to investigate undesired estimation and the use of inductive biases in further detail, finding better ways to prevent the residual network from estimating known dynamics. Notably, identify methods that can effecttively penalize or completely constrain the residual network from estimating forms of known dynamics with unknown parameters.

Systems with multiple equations in generel, would be an interesting topic for further investigation, as results suggested that UPINNs can be too flexible compared to the constraints in these cases. Future work could investigate the use of additional inductive biases to address these issues.

Results also showed that combining UPINNs with symbolic regression can recover governing equations in many cases. Future work could further investigate the use of PySINDy or apply other symbolic regression frameworks, such as PySR [59], AI Feynman [22], gplearn or others. Improving performance on noisy and sparse data using models with fewer or no inductive biases, would be a valuable contribution.

Regarding the plasma physics problem of chapter 8, the experiments showed

promising results, but further work is needed to obtain a more accurate model. Future work could focus on a more thorough investigation of PINN optimization techniques and hyperparameter tuning. Unfortunatly, the limited time scope of this thesis did not allow for such an investigation to be conducted.

10 Conclusion

This thesis set out to investigate the use of Universal Physics-Informed Neural Networks (UPINNs) for discovering hidden physics in dynamical systems. The first part established the mathematical foundation of neural networks, covering artificial neurons, activation functions, the training process and various network architectures. This was followed by an introduction to regular PINNs, along with techniques that have shown to improve their performance, ultimatly leading to the formulation of UPINNs and the second part of the thesis, which focused on applying UPINNs to a variety of ODE and PDE systems.

Both regular PINNs and UPINNs have been applied to a selection of ODE and PDE systems, including the Lotka-Volterra predator-prey model, the SIR epidemic model, Burgers' equation and the Navier-Stokes equations for incompressible fluid flow. The results demonstrate that UPINNs are well capable of modelling the residuals of partially known systems, even when data is sparse and/or noisy. Additionally, the neural network approximations of the hidden dynamics were so precise that the symbolic regression frameworks of SINDy and PDE-find could, in many cases, accurately recover the hidden equations from a set of densely sampled residuals. Additionally, it was shown that UPINNs can perform well without the inductive bias suggested by the original UPINN paper [8]. Allbeit, it improves convergence, robustness to noise and generalization.

It was demonstrated that the high flexibility of the general UPINN formulation, allow the residual network to perform undesired estimation of insufficiently constrained known dynamics, or even introduce non-existing ”phantom” dynamics that do not alter the solution. This issue was particularly evident in inverse problems and systems with several equations. A fundamental challenge of this behavior is that the solution is often unaffected, making it difficult to detect. In these scenarios, imposing inductive biases on the model can be crucial for obtaining meaningful and consistent results.

Many of the well studied techniques for optimizing the training process and convergence rate of regular PINNs were found to have a less significant effect on UPINNs. Notably, the PDE residual modelling was enhanced in some cases when applying softadapt weight loss balancing and/or residual based adaptive distribution (RAD) refinement on the collocation points during training.

While the tested architectures, including FNNs, ResNets and KANs (and combinations of them), often performed similarly, the standard feed-forward neural network was found to be the most reliable and robust choice. The most significant improvements were achieved with wider networks, adaptive sampling and weight loss balancing, but did dependent on the specific problem.

References

[1] Rui Wang et al. “Physics-Guided Deep Learning for Dynamical Systems: A Survey”. In: (Feb. 2023). URL: https://arxiv.org/abs/2107.01272v6.

[2] George Em Karniadakis et al. “Physics-informed machine learning”. In: *Nature Reviews Physics 2021 3:6* 3 (6 May 2021), pp. 422–440. ISSN: 2522-5820. DOI: 10.1038/s42254-021-00314-5. URL: https://www.nature.com/articles/s42254-021-00314-5.

[3] Maziar Raissi et al. “Physics Informed Deep Learning (Part I): Data-driven Solutions of Nonlinear Partial Differential Equations”. In: (Nov. 2017). URL: https://arxiv.org/abs/1711.10561v1.

[4] Maziar Raissi et al. “Physics Informed Deep Learning (Part II): Data-driven Discovery of Nonlinear Partial Differential Equations”. In: (Nov. 2017). URL: https://arxiv.org/abs/1711.10566v1.

[5] M. Raissi et al. “Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations”. In: *Journal of Computational Physics* 378 (Feb. 2019). Original PINNs paper., pp. 686–707. ISSN: 0021-9991. DOI: 10.1016/J.JCP.2018.10.045.

[6] Lena Podina et al. “A PINN Approach to Symbolic Differential Operator Discovery with Sparse Data”. In: (Dec. 2022).

[7] Christopher Rackauckas et al. “Universal Differential Equations for Scientific Machine Learning”. In: (Jan. 2020). URL: https://arxiv.org/abs/2001.04385v4.

[8] Lena Podina et al. “Universal Physics-Informed Neural Networks: Symbolic Differential Operator Discovery with Sparse Data”. In: *Proceedings of Machine Learning Research*. Vol. 202. 2023.

[9] Steven L. Brunton et al. “Discovering governing equations from data by sparse identification of nonlinear dynamical systems”. In: *Proceedings of the National Academy of Sciences of the United States of America* 113 (15 2016). ISSN: 10916490. DOI: 10.1073/pnas.1517384113.

[10] Samuel H Rudy et al. “Data-driven discovery of partial differential equations”. In: (2016), p. 123.

[11] Ziming Liu et al. “KAN: Kolmogorov-Arnold Networks”. In: (Apr. 2024). URL: https://arxiv.org/abs/2404.19756v4.

[12] Subhajit Patra et al. “Physics Informed Kolmogorov-Arnold Neural Networks for Dynamical Analysis via Efficent-KAN and WAV-KAN”. In: (July 2024). URL: https://arxiv.org/abs/2407.18373v2.

[13] A. Ali Heydari et al. “SoftAdapt: Techniques for Adaptive Loss Weighting of Neural Networks with Multi-Part Loss Functions”. In: (Dec. 2019). URL: https://arxiv.org/abs/1912.12355v1.

[14] Shu-Mei Qin et al. “RAR-PINN algorithm for the data-driven vector-soliton solutions and parameter discovery of coupled nonlinear equations”. In: (2022).

[15] Siddhartha Mishra et al. “Estimates on the generalization error of physics-informed neural networks for approximating PDEs”. In: *IMA Journal of Numerical Analysis* 43 (1 2023). ISSN: 0272-4979. DOI: 10.1093/imanum/drab093.

[16] Chenxi Wu et al. “A comprehensive study of non-adaptive and residual-based adaptive sampling for physics-informed neural networks”. In: *Computer Methods in Applied Mechanics and Engineering* 403 (Jan. 2023). RAR-D, p. 115671. ISSN: 0045-7825. DOI: 10.1016/J.CMA.2022.115671.

[17] Viet Dung Duong et al. “On The Choice of Activation Functions in Physics-Informed Neural Network for Solving Incompressible Fluid Flows”. In: *AIAA SciTech Forum and Exposition, 2023* (2023). DOI: 10.2514/6.2023-1803. URL: https://arc.aiaa.org/doi/10.2514/6.2023-1803.

[18] Elham Kiyani et al. “Which Optimizer Works Best for Physics-Informed Neural Networks and Kolmogorov-Arnold Networks?” In: (Jan. 2025).

[19] Ameya D. Jagtap et al. “Extended Physics-Informed Neural Networks (XPINNs): A Generalized Space-Time Domain Decomposition Based Deep Learning Framework for Nonlinear Partial Differential Equations”. In: *Communications in Computational Physics* 28 (5 Jan. 2020), pp. 2002–2041. ISSN: 1815-2406. DOI: 10.4208/CICP.OA-2020-0164.

[20] Lena Podina et al. “Enhancing Symbolic Regression and Universal Physics-Informed Neural Networks with Dimensional Analysis”. In: (Nov. 2024). URL: https://arxiv.org/abs/2411.15919v1.

[21] Silviu Marian Udrescu et al. “AI Feynman: A physics-inspired method for symbolic regression”. In: *Science Advances* 6 (16 Apr. 2020). ISSN: 23752548. DOI: 10.1126/SCIADV.AAY2631.

[22] Silviu Marian Udrescu et al. “AI Feynman 2.0: Pareto-optimal symbolic regression exploiting graph modularity”. In: *Advances in Neural Information Processing Systems* 2020-December (June 2020). ISSN: 10495258. URL: https://arxiv.org/abs/2006.10782v2.

[23] Adam Paszke et al. “PyTorch: An Imperative Style, High-Performance Deep Learning Library”. In: *Advances in Neural Information Processing Systems* 32 (Dec. 2019). ISSN: 10495258. URL: https://arxiv.org/abs/1912.01703v1.

[24] OpenAI et al. “GPT-4 Technical Report”. In: (Mar. 2023). URL: https://arxiv.org/abs/2303.08774v6.

[25] Ashish Vaswani et al. “Attention Is All You Need”. In: *Advances in Neural Information Processing Systems* 2017-December (June 2017), pp. 5999–6009. ISSN: 10495258. URL: https://arxiv.org/abs/1706.03762v7.

[26] Aditya Ramesh et al. “Zero-Shot Text-to-Image Generation”. In: *Proceedings of Machine Learning Research*. Vol. 139. DALL-E. 2021.

[27] Alexey Dosovitskiy et al. “An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale”. In: *ICLR 2021 - 9th International Conference on Learning Representations* (Oct. 2020). URL: https://arxiv.org/abs/2010.11929v2.

[28] Robin Rombach et al. “High-Resolution Image Synthesis with Latent Diffusion Models”. In: *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition* 2022-June (Dec. 2021). Stable Diffusion, pp. 10674–10685. ISSN: 10636919. DOI: 10.1109/CVPR52688.2022.01042. URL: https://arxiv.org/abs/2112.10752v2.

[29] Warren S. McCulloch et al. “A logical calculus of the ideas immanent in nervous activity”. In: *The Bulletin of Mathematical Biophysics* 5 (4 1943). ISSN: 00074985. DOI: 10.1007/BF02478259.

[30] F. Rosenblatt. “The perceptron: A probabilistic model for information storage and organization in the brain”. In: *Psychological Review* 65 (6 1958). ISSN: 0033295X. DOI: 10.1037/h0042519.

[31] Marvin Minsky et al. *Perceptrons*. First book introducing the perceptron. 2017. DOI: 10.7551/mitpress/11301.001.0001.

[32] G. Cybenko. "Approximation by superpositions of a sigmoidal function". In: *Mathematics of Control, Signals, and Systems* 2 (4 Dec. 1989), pp. 303–314. ISSN: 09324194. DOI: 10.1007/BF02551274/METRICS. URL: https://link.springer.com/article/10.1007/BF02551274.

[33] Kurt Hornik et al. "Multilayer feedforward networks are universal approximators". In: *Neural Networks* 2 (5 Jan. 1989), pp. 359–366. ISSN: 0893-6080. DOI: 10.1016/0893-6080(89)90020-8.

[34] Moshe Leshno et al. "Multilayer feedforward networks with a nonpolynomial activation function can approximate any function". In: *Neural Networks* 6 (6 Jan. 1993), pp. 861–867. ISSN: 0893-6080. DOI: 10.1016/S0893-6080(05)80131-5.

[35] Yann Lecun et al. "Deep learning". In: *Nature* 521 (7553 May 2015), pp. 436–444. ISSN: 14764687. DOI: 10.1038/NATURE14539.

[36] Xavier Glorot et al. "Understanding the difficulty of training deep feedforward neural networks". In: *Journal of Machine Learning Research*. Vol. 9. 2010.

[37] Kaiming He et al. "Deep Residual Learning for Image Recognition". In: *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition* 2016-December (Dec. 2015), pp. 770–778. ISSN: 10636919. DOI: 10.1109/CVPR.2016.90. URL: https://arxiv.org/abs/1512.03385v1.

[38] B. T. Polyak. "Some methods of speeding up the convergence of iteration methods". In: *USSR Computational Mathematics and Mathematical Physics* 4 (5 Jan. 1964), pp. 1–17. ISSN: 0041-5553. DOI: 10.1016/0041-5553(64)90137-5.

[39] Diederik P. Kingma et al. "Adam: A Method for Stochastic Optimization". In: *3rd International Conference on Learning Representations, ICLR 2015 - Conference Track Proceedings* (Dec. 2014). URL: https://arxiv.org/abs/1412.6980v9.

[40] Ilya Loshchilov et al. "Decoupled Weight Decay Regularization". In: *7th International Conference on Learning Representations, ICLR 2019* (Nov. 2017). URL: https://arxiv.org/abs/1711.05101v3.

[41] Christoph Witzgall et al. "Practical Methods of Optimization." In: *Mathematics of Computation* 53 (188 1989). ISSN: 00255718. DOI: 10.2307/2008742.

[42] Dong C. Liu et al. "On the limited memory BFGS method for large scale optimization". In: *Mathematical Programming* 45 (1-3 Aug. 1989), pp. 503–528. ISSN: 00255610. DOI: 10.1007/BF01589116/METRICS. URL: https://link.springer.com/article/10.1007/BF01589116.

[43] K. B. Petersen et al. *The Matrix Cookbook*. Version 20121115. Nov. 2012. URL: http://www2.compute.dtu.dk/pubdb/pubs/3274-full.html.

[44] David E. Rumelhart et al. "Learning representations by back-propagating errors". In: *Nature* 323 (6088 1986). Possibly the original backprop paper. ISSN: 00280836. DOI: 10.1038/323533a0.

[45] Yann LeCun et al. "Gradient-based learning applied to document recognition". In: *Proceedings of the IEEE* 86 (11 1998), pp. 2278–2323. ISSN: 00189219. DOI: 10.1109/5.726791.

[46] Y. Le Cun et al. "Handwritten Digit Recognition with a Back-Propagation Network". In: (1990).

[47] Keiron O'Shea et al. "An Introduction to Convolutional Neural Networks". In: *International Journal for Research in Applied Science and Engineering Technology* 10 (12 Nov. 2015), pp. 943–947. DOI: 10.22214/ijraset.2022.47789. URL: https://arxiv.org/abs/1511.08458v2.

[48] Weihao Weng et al. “U-Net: Convolutional Networks for Biomedical Image Segmentation”. In: *IEEE Access* 9 (May 2015), pp. 16591–16603. ISSN: 21693536. DOI: 10.1109/ACCESS.2021.3053408. URL: https://arxiv.org/abs/1505.04597v1.

[49] Juan R Terven et al. “A COMPREHENSIVE REVIEW OF YOLO ARCHITECTURES IN COMPUTER VISION: FROM YOLOV1 TO YOLOV8 AND YOLO-NAS PUBLISHED AS A JOURNAL PAPER AT MACHINE LEARNING AND KNOWLEDGE EXTRACTION”. In: (Feb. 2024).

[50] A. K. Kolmogorov. “On the Representation of Continuous Functions of Several Variables by Superposition of Continuous Functions of One Variable and Addition”. In: *Doklady Akademii Nauk SSSR* 114 (1957), pp. 369–373.

[51] V. I. Arnold. “Geometrical Methods in the Theory of Ordinary Differential Equations”. In: 250 (1988). DOI: 10.1007/978-1-4612-1037-5. URL: http://link.springer.com/10.1007/978-1-4612-1037-5.

[52] Atılım Güneş Baydin et al. “Automatic differentiation in machine learning: a survey”. In: *Journal of Machine Learning Research* 18 (Feb. 2015), pp. 1–43. ISSN: 15337928. URL: https://arxiv.org/abs/1502.05767v4.

[53] I. M. Sobol’. “On the distribution of points in a cube and the approximate evaluation of integrals”. In: *USSR Computational Mathematics and Mathematical Physics* 7 (4 1967). ISSN: 00415553. DOI: 10.1016/0041-5553(67)90144-9.

[54] J H Halton et al. “Algorithm 247: Radical-inverse quasi-random point sequence”. In: *Communications of the ACM* 7 (12 Dec. 1964), pp. 701–702. ISSN: 15577317. DOI: 10.1145/355588.365104. URL: https://dl.acm.org/doi/10.1145/355588.365104.

[55] Lu Lu et al. “DeepXDE: A Deep Learning Library for Solving Differential Equations”. In: *https://doi.org/10.1137/19M1274067* 63 (1 Feb. 2021), pp. 208–228. ISSN: 00361445. DOI: 10.1137/19M1274067. URL: https://epubs.siam.org/doi/10.1137/19M1274067.

[56] Peng Zheng et al. “A Unified Framework for Sparse Relaxed Regularized Regression: SR3”. In: *IEEE Access* 7 (July 2018), pp. 1404–1423. ISSN: 21693536. DOI: 10.1109/ACCESS.2018.2886528. URL: https://arxiv.org/abs/1807.05411v4.

[57] Statens Serum Institut. “Teknisk baggrundsrapport om metode for tal bag genåbningsscenarier”. In: (Mar. 2021). URL: https://www.ssi.dk/-/media/arkiv/subsites/covid19/modelberegninger/notat-16-marts-2021-prognoser-for-.

[58] A. Salmi et al. “Electron density pedestal behaviour in strike-point sweeping experiment on jet”. In: *Plasma Physics and Controlled Fusion* 65 (5 2023), p. 055025. DOI: 10.1088/1361-6587/acc425.

[59] Miles Cranmer. “Interpretable Machine Learning for Science with PySR and SymbolicRegression.jl”. In: (May 2023). URL: https://arxiv.org/abs/2305.01582v3.

[60] Ashish Pal et al. “KAN/MultKAN with Physics-Informed Spline fitting (KAN-PISF) for ordinary/partial differential equation discovery of nonlinear dynamic systems”. In: (Nov. 2024). URL: https://arxiv.org/abs/2411.11801v1.

[61] Andrew L Maas et al. “Rectifier nonlinearities improve neural network acoustic models”. In: *in ICML Workshop on Deep Learning for Audio, Speech and Language Processing*. 2013.

Appendix

A Proofs and Derivations

A.1 Deep neural networks require non-linear activation functions

Consider a single artificial neuron with input vector $\mathbf{x}$, weight vector $\mathbf{w}$, bias b and a linear activation function, $h(x) = \alpha x + \beta$. The output of the neuron is then given by,

$$f(\mathbf{x}) = h(\mathbf{w}^T\mathbf{x} + b) = \alpha(\mathbf{w}^T\mathbf{x} + b) + \beta = \alpha\mathbf{w}^T\mathbf{x} + \alpha b + \beta.$$

Letting $\tilde{\mathbf{w}} = \alpha\mathbf{w}$ and $\tilde{b} = \alpha b + \beta$, the output can be rewritten as,

$$f(\mathbf{x}) = \tilde{\mathbf{w}}^T\mathbf{x} + \tilde{b},$$

thus linear activation functions are redundant and can be absorbed into the weights and bias.

Without loss of generality, let the bias be incorporated into the weights as mentioned in section 2.1. Now consider a FNN with L layers, each of arbitrary size with weight matrix $\mathbf{W}_l$. Without activation functions and the fact that matrix multiplication is associative, the network takes the form,

$$f_\theta(\mathbf{x}) = (z_L \circ \cdots \circ z_2 \circ z_1)(\mathbf{x}) = \left(\prod_{l=1}^{L} \mathbf{W}_l\right)\mathbf{x} = \mathbf{W}\mathbf{x}.$$

The absence of activation functions causes the layers to collapse into a chain of matrix products, which can be simplified into a single matrix, $\mathbf{W}$. This effectively reduces the network to a single-layer FNN without activation function, which is equivalent to a standard linear regression model.

Combined, these observations demonstrate that non-linear activation functions are a necessity for feed-forward neural networks to truly have multiple layers.

A.2 Approximating interactions with KAN

As mentioned in section 2.6.3, KANs approximate multivariate functions with sums of univariate functions. Initially, it may seem that this prevents the ability of capturing interactions xy between variables x and y. To see why this is not the case,

write the interaction using the sum of squares identity,

$$(x+y)^2 = x^2 + y^2 + 2xy \quad \Leftrightarrow \quad xy = \frac{(x+y)^2 - x^2 - y^2}{2}.$$

or equivalently,

$$xy = \frac{(x+y)^2 - (x-y)^2}{4}$$

Then it becomes apparent that the interaction can be represented by either of the 2 following [2, 2, 1] KANs with univariate activation functions,

$$\phi_{1,1}^{(1)}(x) = -x^2, \quad \phi_{2,1}^{(1)}(x) = x, \quad \phi_{1,2}^{(1)}(x) = -x^2, \quad \phi_{2,2}^{(1)}(x) = x,$$

$$\phi_{1,1}^{(2)}(x) = \frac{x}{2}, \quad \phi_{2,1}^{(2)}(x) = \frac{x^2}{2}, \tag{A.1}$$

or,

$$\phi_{1,1}^{(1)}(x) = x, \quad \phi_{2,1}^{(1)}(x) = x, \quad \phi_{1,2}^{(1)}(x) = x, \quad \phi_{2,2}^{(1)}(x) = -x,$$

$$\phi_{1,1}^{(2)}(x) = \frac{x^2}{4}, \quad \phi_{2,1}^{(2)}(x) = \frac{-x^2}{4}. \tag{A.2}$$

Figure A.1 below shows a KAN resembling (A.1). The approximated activation functions are seen to be consistent with the analytical form of the interaction.

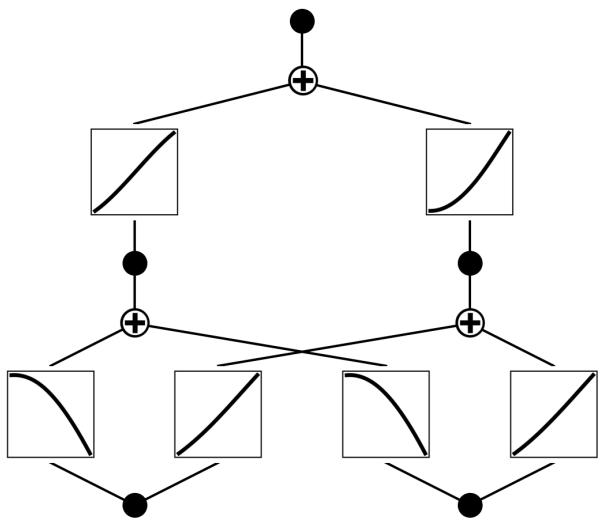

Figure A.1: [2, 2, 1] KAN approximating the interaction xy. Trained with 10000 sobol samples from $(x, y) \in [0, 1]^2$ using AdamW optimizer with learning rate 1e-4 for 10000 epochs. MSE Loss: 6.31e-9.

Although this example converged toward something resembling (A.1), it is important to note that these are not unique solutions. KANs are highly flexible and may converge to something in between (A.1) and (A.2) or something completely

different.

During the project period, Pal et al. introduced the concept of Multiplicative KANs (MultKAN) for ODE/PDE discovery (with inductive bias) [60]. They propose using multiplicative nodes in the KAN architecture along the standard additive nodes. A single layer [2, 1] MultKAN can thus approximate the interaction. MultKANs have not been investigated in this thesis but could be an interesting topic for future research.

B Elaboration

B.1 More on activation functions in PINNs

ReLU

The Rectified Linear Unit (ReLU) is simple and computationally efficient due to its piecewise linear nature. By outputting exactly 0 for all negative inputs, ReLU can introduce sparsity in the network, essentially deactivating neurons that may not contribute to the output, which in turn could indicate that a simpler model is sufficient.

As the first derivative of ReLU vanishes for all negative inputs, if a neuron is accidentally deactivated it cannot be reactivated, a phenomenon often reffered to as the *dead ReLU problem*. This can slow down or even halt training, as the weights of the affected neurons will not be updated.

Leaky ReLU

Leaky ReLU is a variant of ReLU that is designed to rectify the dead ReLU problem by introducing a small slope for negative inputs. This retains much of the computational effenciency and ensures non-zero gradients for all inputs, preventing neurons from deactivating permenantly. The slope is typically kept relatively small, e.g. 0.01 [61], but is a hyperparameter that can be tuned to the specific application.

SiLU (Swish)

The Sigmoid Linear Unit is a smooth approximation of ReLU, differentiable everywhere. It avoids the dead ReLU problem by having non-zero gradients for all inputs. This smoothness is appealing in the context of PINNs, which often require higher order derivatives.

SiLU is computationally more expensive than ReLU due to the exponential operation, but is used in practice due to its smooth properties.

Sigmoid

The sigmoid function is a classic activation function, squashing the input between 0 and 1 in a saturating manner. Despite its popularity in the early era of neural networks, the sigmoid function has fallen out of favor due to zero-centered activation functions converging faster.

The squashing nature also means that it suffers the vanishing gradient problem, as gradients approach 0 for numerically large inputs.

Tanh

The hyperbolic tangent function, is also a squashing function, but with a range of $]-1, 1[$. In contrast to the sigmoid function, tanh is zero-centered, making it suitable for problems where the solution is expected to be symmetric or oscillatory, which is quite common in dynamical systems.

Tanh is the most used activation function in PINNs. It has shown empirical success in many applications and was also the activation function used in the original paper on PINNs [5].

Sin

The sine function is periodic and infinitely smooth, thus potentially useful for problems with periodic solutions. Unlike sigmoid and tanh, the sine function does not suffer from vanishing gradients for large inputs due to its periodic nature, and does not saturate either.

However, due to lack of evidence regarding its effectiveness and poor ability to generalize, it is rarely used in practice.

Activation function	$h(x)$	$\frac{d}{dx}h(x)$	$\frac{d^2}{dx^2}h(x)$	**Advantages**	**Challenges**
Rectified Linear Unit (ReLU)	$\max(0, x)$	$\max(0, \mathrm{sgn}(x))$	0	Computationally fast Promotes sparsity	2nd order derivatives vanish Irreversible dead ReLU
Leaky ReLU	$\max(\alpha x, x)$	$\max(\alpha, \mathrm{sgn}(x))$	0	Improved gradient flow Solves dead ReLU	2nd order derivatives vanish Hyperparameter tuning
SiLU	$x\sigma(x)$	$\sigma(x)(1 + x - x\sigma(x))$	$\frac{d^2}{dx^2}x\sigma(x)$	Smooth ReLU Non-zero gradients	Computationally expensive
Sigmoid	$\sigma(x) = \frac{e^x}{1+e^x}$	$\sigma(x)(1 - \sigma(x))$	$-\frac{e^x(e^x-1)}{(1+e^x)^3}$	Infinitely smooth	Vanishing gradients Saturation
Hyperbolic Tangent	$\tanh(x) = \frac{e^x-e^{-x}}{e^x+e^{-x}}$	$1 - \tanh^2(x)$	$-\frac{2\tanh(x)}{(1-\tanh(x)^2)^{-1}}$	Centered outputs Infinitely smooth	Vanishing gradients Saturation
Sinusoidal	$\sin(x)$	$\cos(x)$	$-\sin(x)$	Periodic behavior No saturation	Unstable gradients Poor generalization Lacking evidence

Table B.1: Functional forms of commonly used activation functions, along their first and second order derivatives. Some known advantages and challenges of each function in the context of Physics-Informed Machine Learning are also listed.

C Additional Results

C.1 Lotka-Volterra

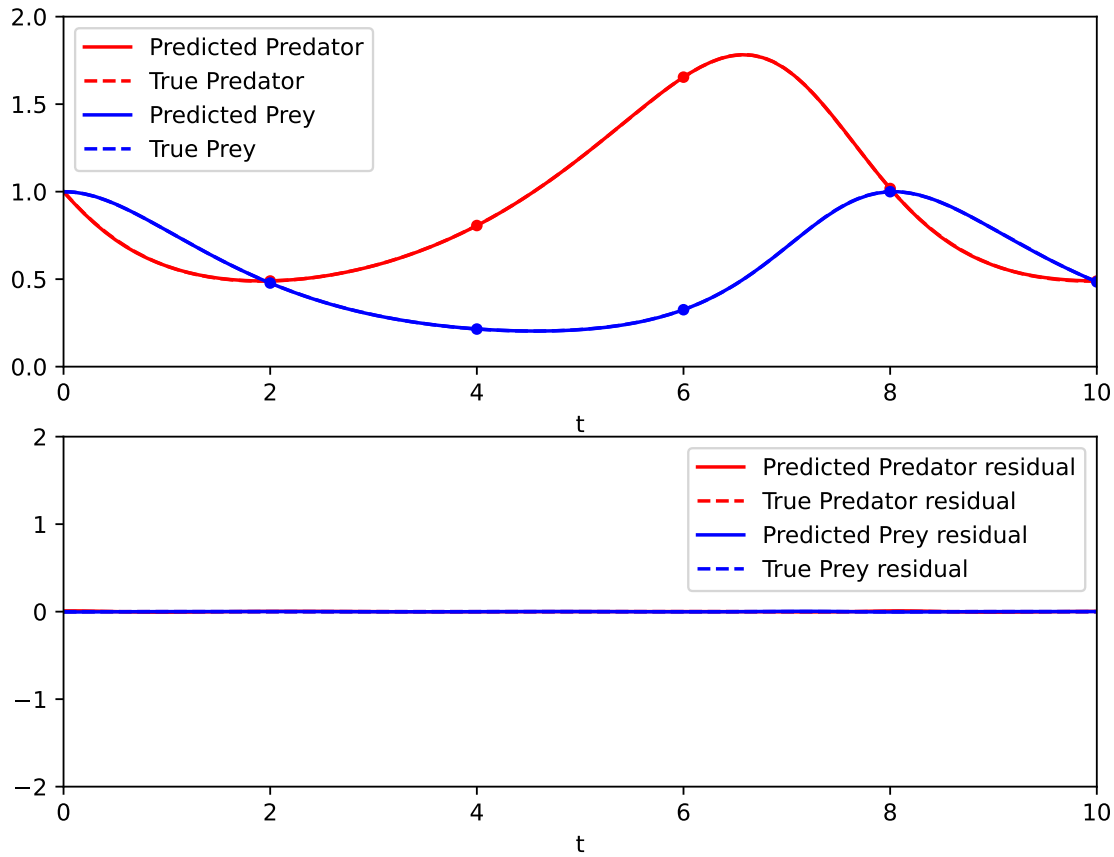

Figure C.2: Lotka Volterra: UPINN model trained on system without hidden dynamics. Dots indicate data.

Noise\Data	5	10	20	40
0.0	-0.752	-0.750	-0.750	-0.750
0.001	-0.747	-0.750	-0.750	-0.750
0.01	-0.756	-0.744	-0.751	-0.749
0.1	-0.720	-0.717	-0.749	-0.737

Table C.2: Lotka Volterra: Estimed values of ϕ for the inductive bias model $\hat{F}_1 = \phi \hat{F}_2$ in table 7.3.

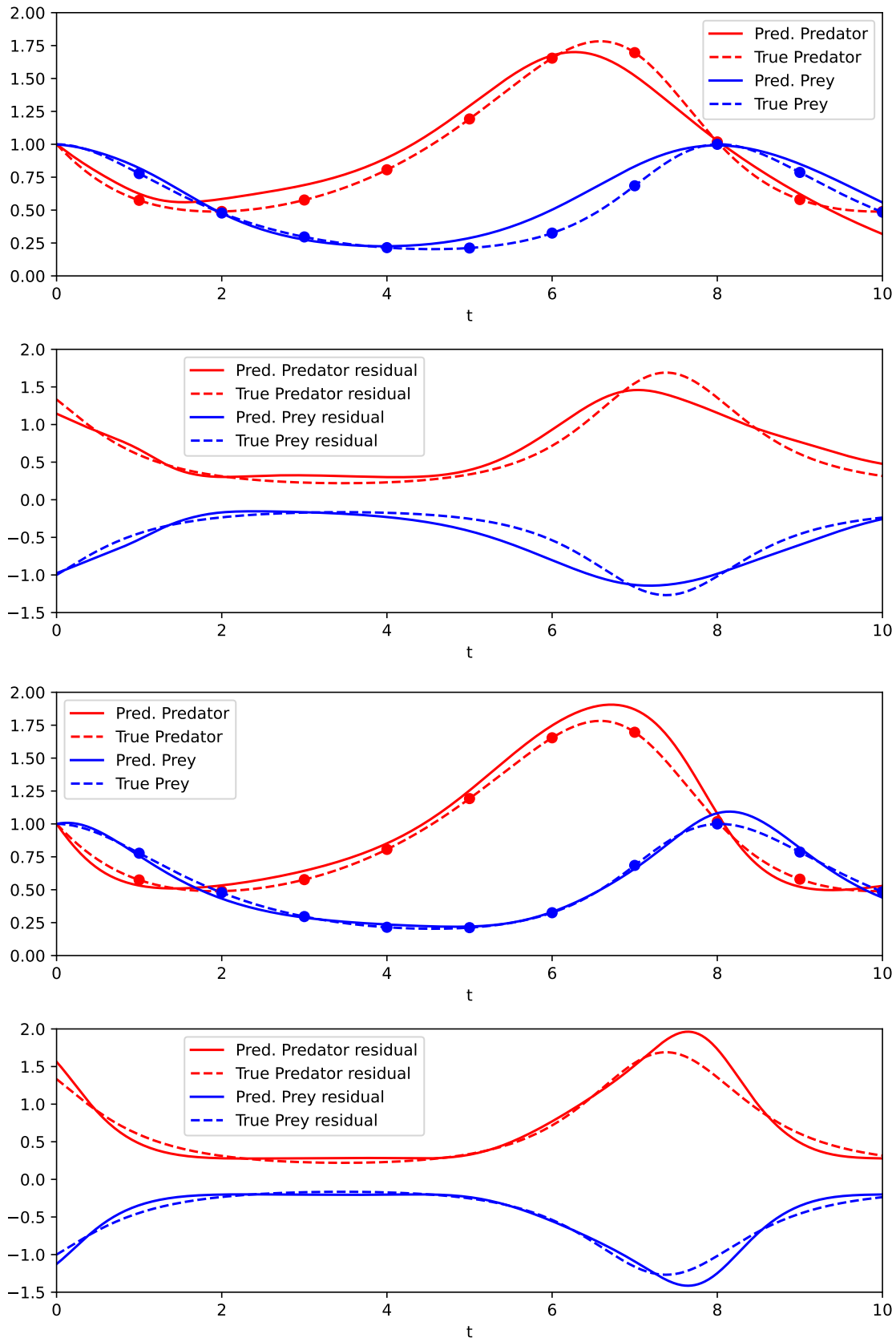

Figure C.3: Lotka Volterra: UPINN solutions trained on 5 data points with 0.1 noise level from section 7.1.4. **Top**: Baseline model. **Bottom**: Shared residual model.

C.2 Burgers'

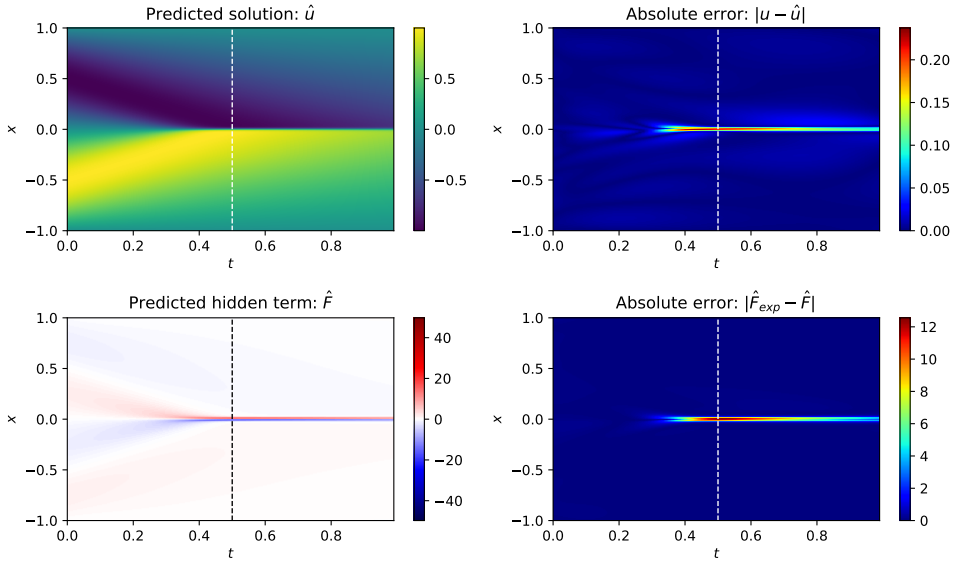

Figure C.4: Burgers': UPINN solution using baseline of [20]*8 FNN with tanh activation, training on sparse clean data. Vertical line indicate data.

C.3 Navier-Stokes

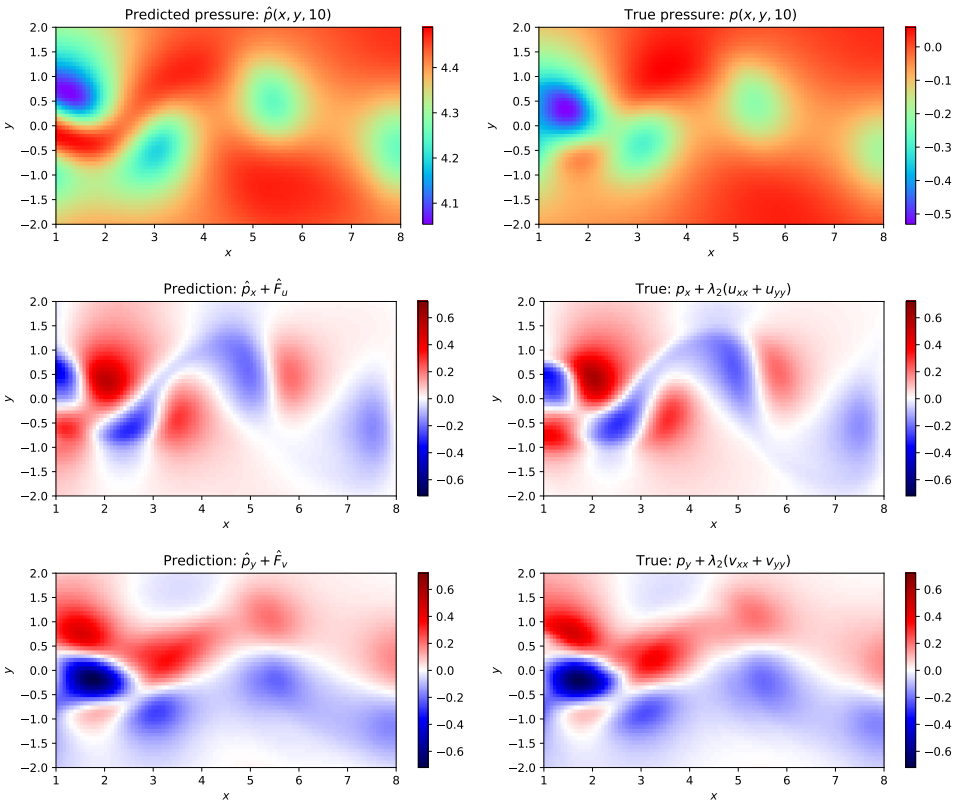

Figure C.5: Navier Stokes: UPINN prediction assuming $-\lambda_2(v_{xx} + v_{yy})$ is unknown without using pressure. Top: Predicted pressure. Bottom: Sum of predicted pressure gradient and PDE residuals.

C.4 Plasma Physics

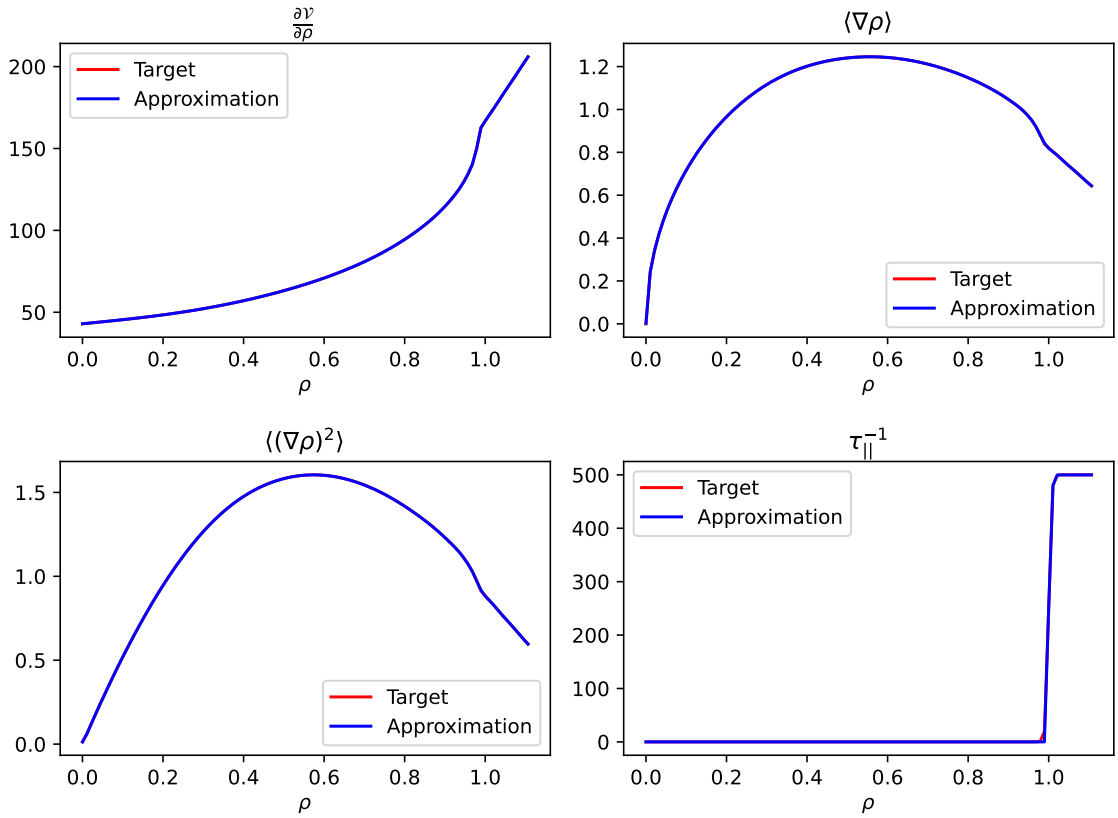

Figure C.6: Tokamak: Neural network approximations of geometric terms for use in PINN model.

D Function Libraries for symbolic regression

SINDy library for Lotka-Volterra

The SINDy library used for equation recovery in section 7.1.6 included up to 2nd order polynomial terms. A total of 6 terms all shown in table D.3.

	1	x	y
1	1	x	y
x	-	x^2	xy
y	-	-	y^2

Table D.3: Lotka-Volterra: SINDy function library.

PDE-find library for Burgers'

The PDE-find library used for equation recovery in section 7.3.4 included up to 2nd order derivatives and 2nd order polynomials. A total of 10 terms all shown in table D.4.

	1	u	u_x	u_{xx}
1	1	u	u_x	u_{xx}
u	-	u^2	uu_x	uu_{xx}
u_x	-	-	u_x^2	u_xu_{xx}
u_{xx}	-	-	-	u_{xx}^2

Table D.4: Burgers': PDE-find function library.

PDE-find library for Navier-Stokes

The PDE-find library used for equation recovery in section 7.4.3 included up to 2nd order derivatives and 2nd order polynomials. A total of 91 terms all shown in table D.5.

	1	u	v	u_x	u_y	u_{xx}	u_{yy}	u_{xy}	v_x	v_y	v_{xx}	v_{yy}	v_{xy}
1	1	u	v	u_x	u_y	u_{xx}	u_{yy}	u_{xy}	v_x	v_y	v_{xx}	v_{yy}	v_{xy}
u	-	u^2	uv	uu_x	uu_y	uu_{xx}	uu_{yy}	uu_{xy}	uv_x	uv_y	uv_{xx}	uv_{yy}	uv_{xy}
v	-	-	v^2	vu_x	vu_y	vu_{xx}	vu_{yy}	vu_{xy}	vv_x	vv_y	vv_{xx}	vv_{yy}	vv_{xy}
u_x	-	-	-	u_x^2	u_xu_y	u_xu_{xx}	u_xu_{yy}	u_xu_{xy}	u_xv_x	u_xv_y	u_xv_{xx}	u_xv_{yy}	u_xv_{xy}
u_y	-	-	-	-	u_y^2	u_yu_{xx}	u_yu_{yy}	u_yu_{xy}	u_yv_x	u_yv_y	u_yv_{xx}	u_yv_{yy}	u_yv_{xy}
u_{xx}	-	-	-	-	-	u_{xx}^2	$u_{xx}u_{yy}$	$u_{xx}u_{xy}$	$u_{xx}v_x$	$u_{xx}v_y$	$u_{xx}v_{xx}$	$u_{xx}v_{yy}$	$u_{xx}v_{xy}$
u_{yy}	-	-	-	-	-	-	u_{yy}^2	$u_{yy}u_{xy}$	$u_{yy}v_x$	$u_{yy}v_y$	$u_{yy}v_{xx}$	$u_{yy}v_{yy}$	$u_{yy}v_{xy}$
u_{xy}	-	-	-	-	-	-	-	u_{xy}^2	$u_{xy}v_x$	$u_{xy}v_y$	$u_{xy}v_{xx}$	$u_{xy}v_{yy}$	$u_{xy}v_{xy}$
v_x	-	-	-	-	-	-	-	-	v_x^2	v_xv_y	v_xv_{xx}	v_xv_{yy}	v_xv_{xy}
v_y	-	-	-	-	-	-	-	-	-	v_y^2	v_yv_{xx}	v_yv_{yy}	v_yv_{xy}
v_{xx}	-	-	-	-	-	-	-	-	-	-	v_{xx}^2	$v_{xx}v_{yy}$	$v_{xx}v_{xy}$
v_{yy}	-	-	-	-	-	-	-	-	-	-	-	v_{yy}^2	$v_{yy}v_{xy}$
v_{xy}	-	-	-	-	-	-	-	-	-	-	-	-	v_{xy}^2

Table D.5: Navier Stokes: PDE-find function library.

E PyTorch implementation

All examples and experiments in this thesis were implemented using the PyTorch library. A custom UPINN framework building on the torch.nn.Module class has been developed for easy setup and testing of UPINN models. The framework has been designed with a variety of options for training a UPINN model.

This section serves as a brief documentation of the UPINN class. The full implementation can be found in the provided GitHub repository.

E.1 UPINN framework documentation

The UPINN class is the main class in the framework.

UPINN

Attributes

`u` : **torch.nn.Module**
The solution network $\hat{u}$.

`N` : **torch.nn.Module**
The known dynamics $\mathcal{N}$.

`F` : **torch.nn.Module**
The residual network $\hat{F}$.

`boundary_points` : **tuple of torch.Tensor**
Tuple of boundary points and targets `(zb, Ub)`.

`data_points` : **tuple of torch.Tensor**
Tuple of data points and targets `(zd, Ud)`.

`collocation_points` : **torch.Tensor**
Collocation points `zc`.

`softadapt_kwargs` : **dict**
Keyword arguments for `softadapt` function. If set, must include `'beta'`, `'loss_weighted'` and `'softadapt_interval'`. Default is `None`.

`lambdas` : **list of float**
Weights for the loss terms. Overruled by `softadapt_kwargs`. Default is `[1.0, 1.0, 1.0]`.

`log` : **dict**
Dictionary for logging training progress.

`optimizer` : **torch.optim.Optimizer**
Default optimizer for `upinn.train_loop`. Default is `torch.optim.Adam` with lr `1e-3`.

`scheduler` : **torch.optim.Optimizer**
Default scheduler for `upinn.train_loop`. Default is `None`.

`UPINN(u, N, F, boundary_points, data_points, collocation_points) -> None`

UPINN class constructor.

`u` : **torch.nn.Module**
The solution network $\hat{u}$. Implements the forward method returning prediction of solution.

`N` : **torch.nn.Module**
The known dynamics $\mathcal{N}$. Implements the forward method returning residuals of the known dynamics. Default is `None`, corresponding to no known dynamics.

`F` : **torch.nn.Module**
The residual network $\hat{F}$. Implements the forward method returning prediction of unknown dynamics. Default is `None`, corresponding to a regular PINN.

`boundary_points` : **tuple of torch.Tensor**
Tuple of boundary points and targets `(zb, Ub)`. Default is `None`.

`data_points` : **tuple of torch.Tensor**
Tuple of data points and targets `(zd, Ud)`. Default is `None`.

`collocation_points` : **torch.Tensor**
Collocation points `zc`. Default is `None`.

`F_input(z, u) -> torch.Tensor`

Returns the input to the residual network $\hat{F}$. Should be implemented in the derived class. Default is to return `torch.cat((z, u), dim=1)`.

`z` : **torch.Tensor**
Input tensor.

`u` : **torch.Tensor**
Tensor of predictions.

`predict_solution(z) -> torch.Tensor`

Returns the solution network prediction $\hat{u}$ at points `z`.

`z` : **torch.Tensor**
Input tensor.

`predict_residual(z) -> torch.Tensor`

Returns the residual network prediction $\hat{F}$ at points `z`.

`z` : **torch.Tensor**
Input tensor.

`to(device) -> None`

Moves all modules, parameters and data to `device`.

`device` : **str**
Compute device. Options are `'cpu'` or `'cuda'`.

`save(name, path, overwrite, save_architecture, save_individual_models) -> None`

Saves the model.

`name` : **str**
Name of the model.

`path` : **str**
Path to save the model. Default is `''`.

`overwrite` : **bool**
Overwrite existing model. Default is `False`.

`save_architecture` : **bool**
Write the architecture to a .txt file. Default is `True`.

`save_individual_models` : **bool**
If true; saves the modules `u`, `N` and `F` individually. If false; saves the entire model in one file. Default is `False`.

`load(name, path, load_individual_models) -> None`

Load a saved model.

`name` : **str**
Name of the model.

`path` : **str**
Path to load the model. Default is `''`.

`load_individual_models` : **bool**
If true; loads the modules `u`, `N` and `F` individually. If false; loads the entire model from one file. Default is `False`.

`freeze(model) -> None`

Disables gradient computation for the specified model.

`model` : **str**
Options are `'u'`, `'N'`, `'F'` and `'all'`. Default is `'all'`.

`unfreeze(model) -> None`

Enables gradient computation for the specified `model`.

`model` : **str**
Options are `'u'`, `'N'`, `'F'` and `'all'`. Default is `'all'`.

`bc_loss() -> torch.Tensor`

Returns the boundary loss. Can be overridden in derived class.

`data_loss()`

Returns the data loss. Can be overridden in derived class.

`pde_loss()`

Returns the PDE loss. Can be overridden in derived class.

`get_loss() -> torch.Tensor`

Returns the total weighted loss.

`closure() -> torch.Tensor`

Computes the loss and updates the model. Appends total and marginal losses to `UPINN.log` and sends to `UPINN.epoch_iterator` or weights and biases depending on `UPINN.log_to_wandb`.

`train_loop(epochs, optimizer, optimizer_kwargs, scheduler, scheduler_kwargs, loss_tol, device, log_wandb_kwargs) -> None`

Trains the model.

`epochs` : **int**
Number of epochs. Default is `1000`.

`optimizer` : **torch.optim.Optimizer**
Optimizer. Default is `upinn.optimizer`.

`scheduler` : **torch.optim.Optimizer**
Learning rate scheduler. Default is `None`.

`loss_tol` : **float**
Loss tolerance for early stopping. Default is `-torch.inf`.

`device` : **str**
Compute device. Options are `'cpu'` or `'cuda'`. Default is `'cuda'` if available.

`log_wandb_kwargs` : **dict**
Keyword arguments for logging to weights and biases. Default is `dict()`.

`refine_collocation_points(method, lb, ub, N_new, N_candidates, N_max, method_kwargs, sample_method) -> None`

Uses adaptive refinement to update collocation points.

`method` : **str**
Refinement method. Options are `'RAD'` and `'RAR'`. Default is `'RAR-D'`.

`lb` : **list**
Lower bounds of the domain.

`ub` : **list**
Upper bounds of the domain.

`N_new` : **int**
Number of new collocation points. Default is `None`, using `UPINN.N_coll`.

`N_candidates` : **int**
Number of candidate points to sample from. Default is `None`, using `50*N_new`.

`N_max` : **int**
Maximum number of collocation points. Default is `None`, using `5*UPINN.N_coll`.

`method_kwargs` : **dict**
Keyword arguments for the refinement method. Default is `dict()`.

`sample_method` : **str**
Sampling method for candidate points. Options are `'grid'`, `'uniform'`, `'sobol'` and `'lhs'`. Default is `'sobol'`.

E.2 Example: UPINN for Burgers' equation

The following code snippet is an example of setting up and training a UPINN model using the framework. The example trains a UPINN model for Burgers' equation with unknown convection term using a FNN with 8 hidden layers of 20 neurons each with Tanh activation functions. It employs SoftAdapt with $\beta = 0.1$ and loss weighting every 100 epochs and alternate training with Adam and L-BFGS optimizers for 1000 epochs each followed by RAD refinement of collocation points. Further, it use the inductive bias of passing the solution network output and its partial derivatives as input to the residual network. Notice how this inductive bias is implemented by overwriting the `F_input` method.

Note that the class `BurgerData` is a custom class formatting the data for the Burgers' equation into the correct format. The class is not included in the code snippet but can be found in the provided GitHub repository. The `sample_collocation_points` function is listed below the code snippet. `FNN` is standard custom feedforward neural network class.

```
import torch
import numpy as np
from upinn import UPINN
from architectures import FNN
from utils import sample_collocation_points
```

```python
from BurgerData import BurgerData

# Define known dynamics N
class SparseBurger(torch.nn.Module):
    def __init__(self, nu):
        super(SparseBurger, self).__init__()
        self.nu = nu

    def forward(self, z, u):

        d2udxt = torch.autograd.grad(u, z, torch.ones_like(u), create_graph=True)[0]
        u_t = d2udxt[:,0:1]
        u_x = d2udxt[:,1:2]

        u_xx = torch.autograd.grad(d2udxt, z, torch.ones_like(d2udxt), create_graph=
            True)[0][:,1:2]

        return u_t - self.nu*u_xx

N = SparseBurger(nu=0.01/torch.pi)

# Define custom UPINN model
class BurgerUPINN(UPINN):
    def F_input(self, z, u):
        dudxt = torch.autograd.grad(u, z, torch.ones_like(u), create_graph=True)[0]
        u_t = dudxt[:,0:1]
        u_x = dudxt[:,1:2]
        return torch.cat((u, u_t, u_x), dim=-1)

# Define solution network u
u = FNN(
    dims=[2, 20, 20, 20, 20, 20, 20, 20, 20, 1],
    hidden_act=torch.nn.Tanh(),
    output_act=torch.nn.Identity(),
)

# Define residual network F
F = FNN(
    dims=[3, 20, 20, 20, 20, 20, 20, 20, 20, 1],
    hidden_act=torch.nn.Tanh(),
    output_act=torch.nn.Identity(),
)

# Initial condition
N_ic = N_bc = 100
x0 = torch.linspace(-1, 1, N_ic)
X0, U0 = torch.stack((torch.zeros_like(x0), x0), dim=-1), -torch.sin(torch.pi * x0).
    reshape(-1, 1)

# Boundary condition
t_bc = torch.linspace(0, 1, N_bc)
XL, XR = torch.stack((t_bc, -torch.ones_like(t_bc)), dim=-1), torch.stack((t_bc,
    torch.ones_like(t_bc)), dim=-1)
U_bc = torch.zeros_like(t_bc).reshape(-1, 1)

# All boundary conditions
```

```python
Xbc = torch.cat((X0, XL, XR), dim=0)
Ubc = torch.cat((U0, U_bc, U_bc), dim=0)


# Collocation points
N_coll = 10000
Xc = sample_collocation_points(N_coll, 2, lb=[0, -1], ub=[1, 1], method='sobol')

# Load data
noise_level = 0.0
time_slices = [25, 50, 99]
data = BurgerData(time_slices=time_slices, noise_level=noise_level)
Xd, Ud = data.data_points

# Initialize UPINN model
model = BurgerUPINN(u, N, F, (Xbc, Ubc), (Xd, Ud), Xc)

# Optimizers
adam = torch.optim.Adam(model.parameters(), lr=2e-3)
lbfgs = torch.optim.LBFGS(model.parameters(), lr=1e-1)

# --- Options ---
# Enable SoftAdapt with beta=0.1 and loss_weighting every 100 epochs
model.softadapt_kwargs = {'beta': 0.1, 'loss_weighted': True, 'softadapt_interval':
    100}

# Train model
for _ in range(10):
    model.train_loop(1000, optimizer=adam)
    model.train_loop(50, optimizer=lbfgs)
    model.refine_collocation_points(method='RAD', lb=[0, -1], ub=[1, 1])
```

```python
from scipy.stats import qmc

def sample_collocation_points(N, n_dims, lb, ub, method='sobol', verbose=False):
"""
Sample N collocation points from the domain [lb, ub]
"""

if method == 'sobol':
    sobol = torch.quasirandom.SobolEngine(dimension=n_dims)
    Zc = sobol.draw(n=N, dtype=torch.float32)

elif method == 'lhs':
    hypercube = qmc.LatinHypercube(d=n_dims)
    Zc = torch.tensor(hypercube.random(n=N), dtype=torch.float32)

elif method == 'uniform':
    Zc = torch.rand(N, n_dims)

elif method == 'grid':
    # Will not generate N points if N is not a perfect square
    root = N**(1/n_dims)
    m = int(np.ceil(N**(1/n_dims)))
```

```
        print(f"[Info]: The {n_dims}-root of {N} is not an integer, sample will contain {m**n_dims} points instead.") if root % 1 != 0 and verbose else None
        grid = torch.linspace(0, 1, m)
        Zc = torch.cartesian_prod(*[grid]*n_dims)
        if n_dims == 1:
            Zc = Zc.unsqueeze(1)

    else:
        raise ValueError(f"Invalid sample method: {method}\nChoose from ['sobol', 'lhs', 'uniform', 'grid']")
```

```
class SoftAdapt(torch.nn.Module):
    def __init__(self, beta=0.0, loss_weighted=False):
        super(SoftAdapt, self).__init__()
        self.beta = beta
        self.loss_weigthed = loss_weighted
        self.prev_losses = 0.0

    def forward(self, losses):
        self.prev_losses = losses
        s = losses - self.prev_losses
        t = torch.exp(self.beta*s)
        new_lambdas = losses*t / torch.sum(losses*t) if self.loss_weigthed else t / torch.sum(t)
        return new_lambdas
```

```
def refine_collocation_points(self, method, lb, ub, N_new=None, N_candidates=None, N_max=None, method_kwargs=dict(), sample_method='sobol'):
    # Method in UPINN class for adaptive refinement of collocation points
    if N_new is None: N_new = self.N_coll
    if N_candidates is None: N_candidates = 50*N_new
    if N_max is None: N_max = 5*self.N_coll # Maximum number of allowed collocation points

    if N_new > N_candidates: raise ValueError("N_new should be less than or equal to N_candidates.")

    # Sample candidate points
    Xc = sample_collocation_points(N_candidates, len(lb), lb=lb, ub=ub, method=sample_method).requires_grad_(True).float()

    # Compute the residual
    u = self.u(Xc)
    pde_loss = abs(self.N(Xc, u) + self.F(self.F_input(Xc, u)))
    residuals = torch.sum(pde_loss, dim=1)

    if method == 'RAD':
        # Replaces all collocation points with new points
        if method_kwargs == {}: method_kwargs = dict(k=0.5, c=0.1) # Use recommended default values from: Wu, C., Zhu, M., Tan, Q., Kartha, Y., & Lu, L. (2023). https://doi.org/10.1016/J.CMA.2022.115671
        self.collocation_points = RAD_sampler(Xc, residuals, N_new, **method_kwargs)

```

```
        elif method == 'RAR':
            # Appends new points to the existing collocation points
            if len(self.collocation_points) < N_max:
                self.collocation_points = torch.cat([self.collocation_points,
                    RAR_sampler(Xc, residuals, N_new, **method_kwargs)])

        elif method == 'RAR-D':
            # Appends new points to the existing collocation points
            if len(self.collocation_points) < N_max:
                self.collocation_points = torch.cat([self.collocation_points,
                    RAD_sampler(Xc, residuals, N_new, **method_kwargs)])

        else:
            raise ValueError(f"Invalid method: {method}. Choose from ['RAD', 'RAR', 'RAR
                -D']")
```

E.3 Example: PDE-find for Burgers' equation

The following code snippet is an example of using PDE-find with time series cross validation to estimate equations from a trained UPINN model (`upinn_model`) - could be the one trained in section E.2 above.

```
import torch
import numpy as np
import pysindy as ps
from sklearn.model_selection import TimeSeriesSplit

# Define grid for sampling solution and residual networks
train_shape = (256, 100)
t = np.linspace(0, 1, train_shape[1])
x = np.linspace(-1, 1, train_shape[0])
T, X = np.meshgrid(t, x)

# Construct input tensors
T_tensor = torch.tensor(T.flatten(), dtype=torch.float32).reshape(-1, 1)
X_tensor = torch.tensor(X.flatten(), dtype=torch.float32).reshape(-1, 1)
Z_tensor = torch.cat((T_tensor, X_tensor), dim=-1)
Z_tensor.requires_grad_(True)

# Get UPINN predictions for solution and residuals
U_tensor = upinn_model.predict_solution(Z_tensor)
U_dot_tensor = upinn_model.predict_residual(Z_tensor)
U_dot = U_dot_tensor.detach().numpy().reshape(*train_shape, 1)
U = U_tensor.detach().numpy().reshape(*train_shape, 1)

# Define function library
pde_lib = ps.PDELibrary(
    library_functions=[lambda x: x],
    function_names=[lambda x: x],
    derivative_order=2,
    include_interaction=True,
    spatial_grid=x,
    is_uniform=True,
)

```

```
# Find optimal model using 10-fold time series cross validation on the threshold parameter
cv_folds = TimeSeriesSplit(n_splits=10)
optimizer = ps.SR3(thresholder="L0", max_iter=10000, normalize_columns=True)
sindy_model = ps.SINDy(feature_library=pde_lib, optimizer=optimizer, feature_names=['u'])
scores = []
taus = np.logspace(0, 2, 50)
for t_ in taus:
    sindy_model.optimizer.threshold = t_
    cv_scores = []
    for train_idx, test_idx in cv_folds.split(t):
        sindy_model.fit(x=U[:, train_idx], t=t[train_idx], x_dot=-U_dot[:, train_idx])
        cv_scores.append(sindy_model.score(x=U[:, test_idx], t=t[test_idx]))
    scores.append(np.mean(cv_scores))

# Fit final model with best threshold
best_tau = taus[np.argmin(scores)]
sindy_model.optimizer.threshold = best_tau
sindy_model.fit(x=U, t=t, x_dot=U_dot)
sindy_model.print()
```

```
(u)' = 0.619 u u_1
```

E.4 Example: SINDy for Lotka-Volterra

The following code snippet is an example of using SINDy to recover the interaction terms in the Lotka-Volterra system.

```

import torch
import numpy as np
import pysindy as ps
from sklearn.model_selection import GridSearchCV

# Define SINDy class wrapper for sklearn compatibility
class SINDy_sklearn(ps.SINDy):
    # pysindy.SINDy class wrapper for better sklearn compatibility.
    # Variables are renamed to match sklearn's fit and score methods.
    #    Z = [t, X]
    #    y = X_dot

    def fit(self, Z, y, **fit_kwargs):
        return super(SINDy_sklearn, self).fit(x=Z[:, 1:], t=Z[:, 0], x_dot=y, **fit_kwargs)

    def score(self, Z, y, **score_kwargs):
        return super(SINDy_sklearn, self).score(x=Z[:, 1:], t=Z[:, 0], x_dot=y, **score_kwargs)

# Define grid for sampling residual network
t = torch.linspace(0, 10, 100).reshape(-1, 1)
```

```python
X = upinn_model.predict_solution(t)
X_dot = upinn_model.predict_residual(t)
t = t.detach().numpy()
X = X.detach().numpy()
X_dot = X_dot.detach().numpy()

# Find optimal model using 10-fold time series cross validation on the threshold parameter
sindy_model = SINDy_sklearn(feature_names=['x', 'y'], feature_library=ps.PolynomialLibrary(degree=2, include_bias=False), optimizer=ps.SR3(thresholder="L0", max_iter=10000, normalize_columns=True))
paramgrid = dict(optimizer__threshold=np.linspace(0.01, 10.0, 100))
cv_folds = TimeSeriesSplit(n_splits=10)
model_selector = GridSearchCV(sindy_model, param_grid=paramgrid, cv=cv_folds, n_jobs=-1)
wrapped_input = np.concatenate([t, X], axis=1)
model_selector.fit(wrapped_input, X_dot)
best_model = model_selector.best_estimator_
best_model.print()
```

```
(x)' = 1.334 x y
(y)' = -1.000 x y
```