

UC Santa Barbara

UC Santa Barbara Electronic Theses and Dissertations

Title

Hybrid Machine Learning Algorithms for Solving Forward and Inverse Problems in Physical Sciences

Permalink

<https://escholarship.org/uc/item/8cm0s7kz>

Author

Pakravan, Samira

Publication Date

2024

Peer reviewed|Thesis/dissertation

University of California
Santa Barbara

Hybrid Machine Learning Algorithms for Solving Forward and Inverse Problems in Physical Sciences

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

Doctor of Philosophy
in
Mechanical Engineering

by

Samira Pakravan

Committee in charge:

Professor Frederic G. Gibou, Chair
Professor Tresa M. Pollock
Professor Enoch Yeung
Professor Paolo Luzzatto-Fegiz

June 2024

The Dissertation of Samira Pakravan is approved.

Professor Tresa M. Pollock

Professor Enoch Yeung

Professor Paolo Luzzatto-Fegiz

Professor Frederic G. Gibou, Committee Chair

June 2024

Hybrid Machine Learning Algorithms for Solving Forward and Inverse Problems in
Physical Sciences

Copyright © 2024

by

Samira Pakravan

To My Parents & Pouria

Acknowledgements

I am immensely grateful to the individuals who have supported me throughout my doctoral journey. First and foremost, my sincere gratitude goes to Professor Frederic G. Gibou, my advisor, for his guidance, encouragement, and patience. His belief in my abilities has been a constant source of motivation.

I am also grateful to my dissertation committee members, Professor Tresa M. Pollock, Professor Paolo Luzzatto-Fegiz, and Professor Enoch Yeung, for their valuable input and support. Their expertise has been invaluable in shaping my work.

To my loving husband, Pouria, words cannot express the depth of my gratitude for your support and boundless patience. Your counsel and love have illuminated my path, offering perspective and solace both in my research and in life. I am endlessly grateful for your companionship and the countless sacrifices you've made to see me succeed.

I am deeply appreciative of the sacrifices made by my parents, Zahra and Mohammadtaghi, both of whom dedicated their lives to education. Their commitment to fostering learning has deeply inspired my own passion for teaching. As a teaching assistant, I have been driven by their example to work diligently and passionately. Additionally, I am thankful for the steadfast support of my sister, Parisa, especially during the past decade when her presence was sorely missed, and my brother, Alireza, for his unwavering presence and encouragement.

I would like to express my heartfelt thanks to my dear friends Anees and Mitra for their incredible kindness, their belief in me and words of encouragement that carried me through the toughest of times. Ghazaleh and Behzad, I am grateful for the late-night conversations, and the shared moments of celebration that have enriched this journey.

To each and every person who has touched my life and contributed to my journey, I am forever grateful for your presence in my life.

Curriculum Vitæ

Samira Pakravan

Education

2024	Ph.D. in Mechanical Engineering (Expected), University of California, Santa Barbara, USA.
2015	M.S. in Computer Science, New Mexico State University, USA.
2012	B.S. in Computer Science, Sadjad Institute of Higher Education, Iran.

Publications

(†: equal contribution)

• Journal papers:

1. P Mistani[†], **S Pakravan**[†], R Ilango, F Gibou. JAX-DIPS: neural bootstrapping of finite discretization methods and application to elliptic problems with discontinuities, *Journal of Computational Physics*, 2023 [1]
2. **S Pakravan**[†], P Mistani[†], MA Aragon-Calvo, F Gibou. Solving inverse-PDE problems with physics-aware neural networks, *Journal of Computational Physics*, 2021 [2]
3. P Mistani, **S Pakravan**, F Gibou. A fractional stochastic theory for interfacial polarization of cell aggregates, arXiv preprint arXiv:2008.11819, 2020 (under review) [3].

• Conference workshops on AI:

1. **S Pakravan**, N Evangelou, M Usdin, L Brooks and J Lu, From noise to signal: unveiling treatment effects from digital health data through pharmacology-informed neural-SDE, *The International Conference on Learning Representations (ICLR)*, *Learning from Time-Series for Health Workshop*, 2024 [4]
2. P Mistani[†], **S Pakravan**[†], R Ilango, S Choudhary, F Gibou, Neuro-symbolic partial differential equation solver, *Neural Information Processing Systems (NeurIPS)*, *Machine Learning and the Physical Sciences Workshop*, 2022 [5]

• Book chapters:

1. P Mistani, **S Pakravan**, F Gibou. Towards a tensor network representation of complex systems, *Sustainable Interdependent Networks II*, Springer International Publishing 2019 [6].
2. P Mistani, **S Pakravan**, F Gibou. Tensor network renormalization as an ultracalculus for complex system dynamics, *Sustainable Interdependent Networks II*, Springer International Publishing 2019 [7].

Abstract

Hybrid Machine Learning Algorithms for Solving Forward and Inverse Problems in
Physical Sciences

by

Samira Pakravan

Machine learning (ML) techniques have emerged as powerful tools for solving differential equations, particularly in the context of partial differential equations (PDEs), enabling accelerated forward simulations and parameter discovery from limited data. However, challenges persist in maintaining numerical accuracy, especially in scenarios requiring real-time inference and inverse problem-solving. This dissertation investigates innovative hybrid strategies that blend classical finite discretization methods with modern ML techniques to enhance accuracy while maintaining computational efficiency. Our research focuses on addressing key challenges at the intersection of scientific computing, machine learning, and applied mathematics, including performance, accuracy, and data efficiency.

Contents

Curriculum Vitae	vi
Abstract	vii
1 Introduction	1
1.1 Permissions and Attributions	6
2 Solving Inverse-PDE Problems with Physics-Aware Neural Networks	7
2.1 abstract	7
2.2 Introduction	8
2.3 Blended inverse-PDE network (BiPDE-Net)	16
2.4 Mesh-based BiPDE: Finite Differences	21
2.5 Mesh-less BiPDE: Multi-Quadratic Radial Basis Functions	45
2.6 Conclusion	62
3 JAX-DIPS:	63
3.1 abstract	63
3.2 Introduction	64
3.3 Neural Bootstrapping Method (NBM)	70
3.4 JAX-DIPS: Differentiable Interfacial PDE Solver	75
3.5 Numerical Results	89
3.6 Discussion and future work	102
3.7 Conclusion	106
4 Pharmacology-Informed Neural-SDE	109
4.1 abstract	109
4.2 Introduction	110
4.3 Methods	111
4.4 Results	116
4.5 Conclusion	120
Bibliography	121

Chapter 1

Introduction

Machine learning (ML) techniques have shown promising potential in solving differential equations, a fundamental tool in modeling and understanding various phenomena in science and engineering. Particularly, incorporating machine learning approaches in solving partial differential equations (PDEs), opens new frontiers for accelerating forward simulations and discovering hidden parameters that characterize physical systems from limited data. The standard procedure for simulating systems of PDEs involves iteratively solving linear systems that arise from finite discretization methods. Beyond forward simulations, solving inverse PDE problems requires the extra step of solving the adjoint equations which adds to the computational costs and challenges the limit of traditional methods. Furthermore, in many areas of physical sciences long-time processes and large-scale interactions are the determining phenomena that need to be computationally resolved, these extended spatiotemporal scales often exacerbate their limitations on currently available computational resources, therefore leaving many problems unfeasible to address. In recent years an important example is mitigating instabilities of high-concentration biotherapeutics (such as developing antibody drugs) in the biotechnology industry where direct numerical simulations of macromolecules over sufficiently

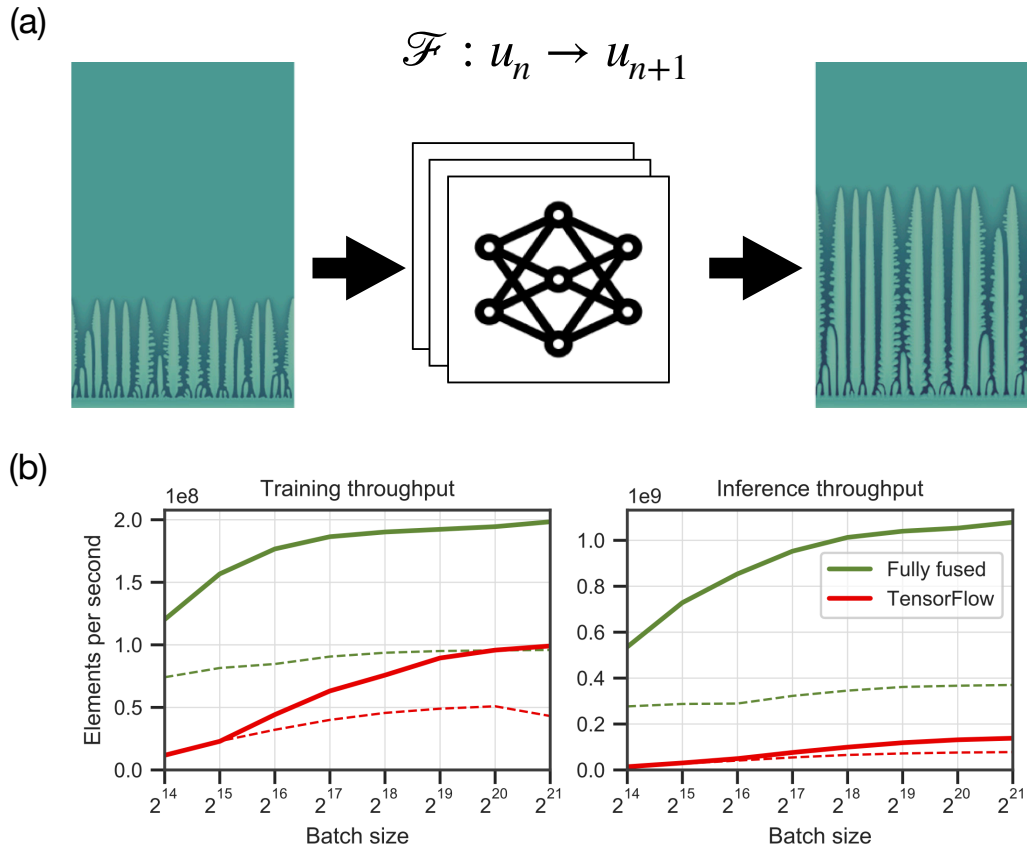
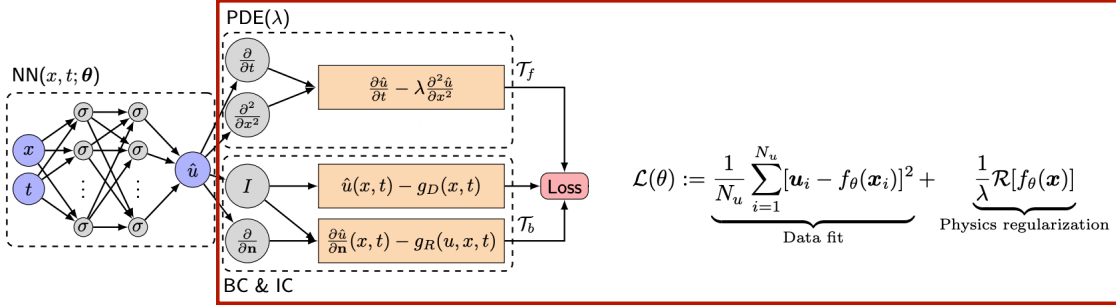


Figure 1.1: Real-time digital twins can be achieved through neural operators, \mathcal{F} , that update the simulation state at unprecedented efficiency. The top panel displays snapshots adapted from [8], while the bottom panel presents throughput measurements from [9], illustrating neural network inference performance on contemporary accelerators.

large spatiotemporal scales have been impossible. Another example is the direct numerical simulation of the Earth’s atmosphere for real-time weather prediction¹. Therefore, algorithmic innovations for accelerating forward and inverse PDE solvers is a significant area of research with immediate impact across many industries. Figure 1.1 illustrates the potential of “neural operators”, *i.e.* a class of machine learning techniques for solving PDEs, on achieving real-time digital twins of complex systems.

¹See the NVIDIA Earth-2 initiative.

- (a)
 - *All derivatives using automatic differentiation (AD)*
 - *Trained models lack numerical accuracy.*
 - *Leads to high order AD, hence not scalable to higher order PDEs.*



- (b)
 - *Spatial derivatives using finite discretization (FD), AD for model parameters.*
 - *Trained models benefit numerical accuracy & inherent solution properties (symmetries, jump, etc).*
 - *Leads to first order AD, hence it is scalable to high order PDEs.*

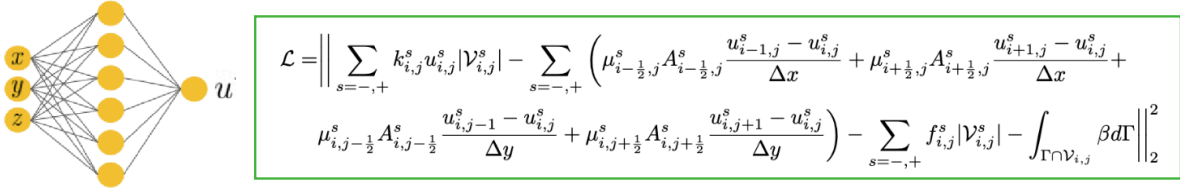


Figure 1.2: Two main strategies for constrained optimization of neural networks to represent PDE solutions.

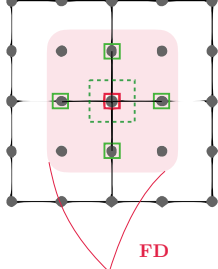
Since 2017, a class of physics-informed machine learning algorithms emerged with the potential to address these computational challenges, although at the expense of degraded numerical accuracies which is paramount to simulating the physical processes of interest; top panel of figure 1.2. Therefore, in this dissertation, we explored innovative avenues to improve the accuracy problem by inventing hybrid optimization strategies of finite discretizations and deep learning methods; see bottom panel of figure 1.2. As illustrated in figure 1.2, the constrained optimization framework for solving PDEs with neural networks comes in two flavors: (a) computing all derivatives with automatic differentiation, or (b) computing spatiotemporal gradients with numerical derivatives and model param-

*FD adds extra backpropagation paths
to the local neighborhoods established by the FD scheme*

$$\begin{aligned} \nabla \cdot (D(\mathbf{x})\nabla u) &= -f(\mathbf{x}), & \mathbf{x} \in \Omega \\ u(\mathbf{x}) &= u_0(\mathbf{x}), & \mathbf{x} \in \partial\Omega \end{aligned}$$

$$\frac{D_{i-1/2,j}u_{i-1,j} - (D_{i-1/2,j} + D_{i+1/2,j})u_{i,j} + D_{i+1/2,j}u_{i+1,j}}{\Delta x^2} +$$

$$\frac{D_{i,j-1/2}u_{i,j-1} - (D_{i,j-1/2} + D_{i,j+1/2})u_{i,j} + D_{i,j+1/2}u_{i,j+1}}{\Delta y^2} + f_{i,j} = 0,$$



$$U(t, \mathbf{x}; \theta) = \text{NN}_\theta(\mathbf{x}) \rightarrow \mathcal{L}(t, \mathbf{x}; \theta) = \left\| \mathbf{D}_{t,\mathbf{x}}[U(\mathbf{x}; \theta)] \right\|$$

$$\theta^{k+1} = \theta^k - \alpha \underbrace{\nabla_\theta}_{\text{AD}} \mathcal{L}(t, \mathbf{x}; \theta^k)$$

Figure 1.3: FD loss acts as a regularization mechanism that enforces underlying symmetries and conservation laws.

eter gradients with automatic differentiation. We demonstrated that finite discretization loss improves accuracy and performance of training. Moreover, at a given accuracy level, the trained neural network model has significantly less number of parameters which leads to higher inference performance.

We emphasize that this line of research is based on our hypothesis that the origin of such inaccuracies lies, primarily, in the evaluation of the loss function rather than a consequence of insufficient expressibility of the neural network architecture that is used to represent the solution fields. Figure 1.3 demonstrates the mechanisms by which finite difference (FD) methods enhance the accuracy of trained neural network models. The FD loss function introduces additional backpropagation pathways during the training process, which regularizes the encoded solutions. Furthermore, FD enforces the conservation laws and symmetries intrinsic to the partial differential equation (PDE) system in local neighborhoods around the training points. These analytical principles have historically underpinned the development of various finite discretization schemes and should be similarly applied to the training of neural network encodings.

Although many innovations by the broader community focused on building larger and more complex neural network architectures, we demonstrate comparable levels of accuracy can be achieved by hybrid discretizations while maintaining a surprisingly simple and shallow neural network model. Arguably, larger and more complex models are counter-productive at enabling real-time digital twins due to their huge computational burden, which is one of the main motivations for developing machine learning simulation algorithms in the first place. Therefore, we believe our contributions in the form of developing innovative hybridization algorithms as well as releasing an open source and high-performance implementation of these algorithms have been significant contributions that complement the mostly model-centric hypothesis pursued in the research community, and pave the way for additional research to further realize the potential of machine learning for physical simulations.

In chapter 2 we present “BiPDE”, that is one of the first frameworks for blending numerical solvers with deep neural networks in the context of solving inverse PDE problems. Notably, we developed one of the earliest frameworks for training neural operators (*i.e.*, mappings between function spaces) of PDE problems, and we illustrated the high accuracy and data efficiency of our framework. In chapter 3 we generalized this hybridization strategy to encompass any finite discretization scheme and addressed several algorithmic and software-centric computational efficiency challenges that arise in three spatial dimensions over irregular interfaces with jump conditions. We presented the “neural bootstrapping method (NBM)” as a generic method to hybridize neural network-based PDE solvers. Moreover, in collaboration with NVIDIA engineers, we implemented and open sourced JAX-DIPS which is a high-performance multi-GPU/TPU/CPU and end-to-end differentiable software that leverages NBM to train complex neural network models of solutions to 3D PDE problems with jump conditions across irregular interfaces. Notably, JAX-DIPS is the first, and to the best of our knowledge the only, high-performance

open-source ML framework for solving this important class of PDE problems. Finally, in chapter 4, analogous to the BiPDE framework, we developed a hybrid inverse stochastic differential equation solver for the discovery of *effective* pharmacokinetics (PK, concentration with time) and pharmacodynamics (PD, effect with time) models (*i.e.*, widely known as PK/PD modeling) from real-world, limited, and noisy patients data. We developed the “pharmacology-informed neural networks” in collaboration with scientists at the clinical pharmacology division at the Genentech Research and Early Development (gRED). This work exemplifies the impact of hybridization algorithms at the frontiers of biotechnology research in industry.

1.1 Permissions and Attributions

1. The content of chapter 2 is the result of a collaboration with Pouria A. Mistani, Miguel A. Aragon-Calvo, and Frederic Gibou, and has previously appeared in the Journal of Computational Physics [2]. It is reproduced here with the permission of Elsevier: <https://www.sciencedirect.com/science/article/abs/pii/S0021999121003090>.
2. The content of chapter 3 is the result of a collaboration with Pouria A. Mistani, Rajesh Ilango, and Frederic Gibou, and has previously appeared in the Journal of Computational Physics [1]. It is reproduced here with the permission of Elsevier: <https://www.sciencedirect.com/science/article/abs/pii/S0021999123005752>.
3. The content of chapter 4 is the result of a collaboration with Nikolaos Evangelou, Maxime Usdin, Logan Brooks, and James Lu, and has previously appeared on *arXiv*. It is reproduced here with the permission of *arXiv* [4]: <https://arxiv.org/pdf/2403.03274>.

Chapter 2

Solving Inverse-PDE Problems with Physics-Aware Neural Networks

2.1 abstract

We propose a novel composite framework to find unknown fields in the context of inverse problems for partial differential equations (PDEs). We blend the high expressibility of deep neural networks as universal function estimators with the accuracy and reliability of existing numerical algorithms for partial differential equations as custom layers in semantic autoencoders. Our design brings together techniques of computational mathematics, machine learning and pattern recognition under one umbrella to incorporate domain-specific knowledge and physical constraints to discover the underlying hidden fields. The network is explicitly aware of the governing physics through a hard-coded PDE solver layer in contrast to most existing methods that incorporate the governing equations in the loss function or rely on trainable convolutional layers to discover proper discretizations from data. This subsequently focuses the computational load to only the discovery of the hidden fields and therefore is more data efficient. We call this architec-

ture Blended inverse-PDE networks (hereby dubbed BiPDE networks) and demonstrate its applicability for recovering the variable diffusion coefficient in Poisson problems in one and two spatial dimensions, as well as the diffusion coefficient in the time-dependent and nonlinear Burgers' equation in one dimension. We also show that this approach is robust to noise.

2.2 Introduction

Inverse differential problems, where given a set of measurements one seeks a set of optimal parameters in a governing differential equation, arise in numerous scientific and technological domains. Some well-known applications include X-ray tomography [10, 11], ultrasound [12], MRI imaging [13], and transport in porous media [14]. Moreover, modeling and control of dynamic complex systems is a common problem in a broad range of scientific and engineering domains, with examples ranging from understanding the motion of bacteria colonies in low Reynolds number flows [15], to the control of spinning rotorcrafts in high speed flights [16, 17]. Other applications in medicine, navigation, manufacturing, *etc.* need estimation of the unknown parameters in *real-time*, *e.g.* in electroporation [18, 19] the pulse optimizer has to be informed about tissue parameters in microsecond time. On the other hand, high resolution data-sets describing spatiotemporal evolution of complex systems are becoming increasingly available by advanced multi-scale numerical simulations (see *e.g.* [19, 20]). These advances have become possible partly due to recent developments in discretization techniques for nonlinear partial differential equations with sharp boundaries (see *e.g.* the reviews [21, 22]). However, solving these inverse problems poses substantial computational and mathematical challenges that makes it difficult to infer reliable parameters from limited data and in real-time.

The problem can be mathematically formulated as follows. Let the values of $u =$

$u(t, x_1, \dots, x_n)$ be given by a set of measurements, which may include noise. Knowing that u satisfies the partial differential equation:

$$\frac{\partial u}{\partial t} = f \left(t, x_1, \dots, x_n; u, \frac{\partial u}{\partial x_1}, \dots, \frac{\partial u}{\partial x_n}; \frac{\partial^2 u}{\partial x_1 \partial x_1}, \dots, \frac{\partial^2 u}{\partial x_1 \partial x_n}; \dots; \mathbf{c} \right),$$

find the hidden fields stored in \mathbf{c} , where the hidden fields can be constant or variable coefficients (scalars, vectors or tensors).

Deep neural networks have, rather recently, attracted considerable attention for data modeling in a vast range of scientific domains, in part due to freely available modern deep learning libraries (in particular `TensorFlow` [23]). For example, deep neural networks have shown astonishing success in emulating sophisticated simulations [24, 25, 26, 27, 28], discovering governing differential equations from data [29, 30, 31, 32], as well as potential applications to study and improve simulations of multiphase flows [21]. We refer the reader to [33, 34] for a comprehensive survey of interplays between numerical approximation, statistical inference and learning. However, these architectures require massive datasets and extensive computations to train numerous hidden weights and biases. Therefore, reducing complexity of deep neural network architectures for inverse problems poses a significant practical challenge for many applications in physical sciences, especially when the collection of large datasets is a prohibitive task [35]. One remedy to reduce the network size is to embed the knowledge from existing mathematical models [36] or known physical laws within a neural network architecture [37, 38]. Along these lines, semantic autoencoders were recently proposed by Aragon-Calvo [39], where they replaced the decoder stage of an autoencoder architecture with a given physical law that can reproduce the provided input data given a physically meaningful set of parameters. The encoder is then constrained to discover optimal values for these parameters, which can be extracted from the bottleneck of the network after training. We shall emphasize

that this approach reduces the size of the unknown model parameters, and that the encoder can be used independently to infer hidden parameters in real time, while adding interpretability to deep learning frameworks. Inspired by their work, we propose to blend traditional numerical solver algorithms with custom deep neural network architectures to solve inverse PDE problems more efficiently, and with higher accuracy.

2.2.1 Existing works

Recently, the most widely used approach for solving forward and inverse partial differential equations using neural networks has been the constrained optimization technique. These algorithms augment the cost function with terms that describe the PDE, its boundary and its initial conditions, while the neural network acts as a surrogate for the solution field. Depending on how the derivatives in the PDEs are computed, there may be two general classes of methods that we review in the next paragraph.

In the first class, spatial differentiations in the PDE are performed exclusively using automatic differentiation, while temporal differentiation may be handled using the traditional Runge-Kutta schemes (called *discrete time models*) or using automatic differentiations (called *continuous time models*) [40]. In these methods, automatic differentiation computes gradients of the output of a neural network with respect to its input variables. Hence, the input must always be the independent variables, *i.e.* the input coordinates \mathbf{x} , time and the free parameters. In this regard, network optimization aims to calibrate the weights and biases such that the neural network outputs the closest approximation of the solution of a PDE; this is enforced through a regularized loss function. An old idea that was first proposed by Lagaris *et al.* (1998) [41]. In 2015, the general framework of solving differential equations as a learning problem was proposed by Owhadi [42, 43, 44] which revived interest in using neural networks for solving differential equations in recent

years. Raissi *et al.* (2017) [40, 45] presented the regularized loss function framework under the name *physics informed neural networks* or PINNs and applied it to time-dependent PDEs. Ever since, other authors have mostly adopted PINNs, see *e.g.* [46, 47]. The second class of constrained optimization methods was proposed by Xu and Darve [48] who examined the possibility of directly using pre-existing finite discretization schemes within the loss function.

An alternative approach for solving PDE systems is through explicit embedding of the governing equations inside the architecture of deep neural networks via convolutional layers, activation functions or augmented neural networks. Below we review some of these methods:

- A famous approach is PDE-Net [31, 49] which relies on the idea of numerical approximation of differential operators by convolutions. Therefore, PDE-Nets use convolution layers with trainable and constrained kernels that mimic differential operators (such as U_x, U_y, U_{xx}, \dots) whose outputs are fed to a (symbolic) multilayer neural network that models the nonlinear response function in the PDE system, *i.e.* the right hand side in $U_t = F(U, U_x, U_y, U_{xx}, \dots)$. Importantly, PDE-Nets can only support *explicit* time integration methods, such as the forward Euler method [31]. Moreover, because the differential operators are being learned from data samples, these methods have hundreds of thousands of trainable parameters that demand hundreds of data samples; *e.g.* see section 3.1 in [31] that uses 20 δt -blocks with 17,000 parameters in each block, and use 560 data samples for training.
- Berg and Nyström [50] (hereby BN17) proposed an augmented design by using neural networks to estimate PDE parameters whose output is fed into a forward finite element PDE solver, while the adjoint PDE problem is employed to compute gradients of the loss function with respect to weights and biases of the network

using automatic differentiation. Even though their loss function is a simple L_2 -norm functional, the physics is not localized in the structure of the neural network as the adjoint PDE problem is also employed for the optimization process. It is important to recognize that in their approach the numerical solver is a separate computational object than the neural network, therefore computing gradients of error functional with respect to the network parameters has to be done explicitly through the adjoint PDE problem. Moreover, their design can not naturally handle trainable parameters in the numerical discretization itself, a feature that is useful for some meshless numerical schemes. In contrast, *in BiPDEs the numerical solver is a computational layer added in the neural network architecture and naturally supports trainable parameters in the numerical scheme*. For example in the meshless method developed in section 2.5 we leverage this unique feature of BiPDEs to also train for shape parameters and interpolation seed locations of the numerical scheme besides the unknown diffusion coefficient.

- Dal Santos *et al.* [51] proposed an embedding of a reduced basis solver as *activation function* in the last layer of a neural network. Their architecture resembles an autoencoder in which the decoder is the reduced basis solver and the parameters at the bottleneck “are the values of the physical parameters themselves or the affine decomposition coefficients of the differential operators” [51].
- Lu *et al.* [52] proposed an unsupervised learning technique using variational autoencoders to extract physical parameters (not inhomogeneous spatial fields) from noisy spatiotemporal data. Again the encoder extracts physical parameters and the decoder propagates an initial condition forward in time given the extracted parameters. These authors use convolutional layers both in the encoder to extract features as well as in the decoder with recurrent loops to propagate solutions in

time; *i.e.* the decoder leverages the idea of estimating differential operators with convolutions. Similar to PDE-Nets, this architecture is also a “PDE-integrator with explicit time stepping”, and also they need as few as 10 samples in the case of Kuramoto-Sivashinsky problem.

In these methods, a recurring idea is treating latent space variables of autoencoders as physical parameters passed to a physical model decoder. This basic idea pre-dates the literature on solving PDE problems and has been used in many different domains. Examples include Aragon-Calvo [39] who developed a galaxy model fitting algorithm using *semantic autoencoders*, or Google Tensorflow Graphics [53] which is a well-known application of this idea for scene reconstruction.

2.2.2 Present work

Basic criteria of developing numerical schemes for solving partial differential equations are *consistency* and *convergence* of the method, *i.e.* increasing resolution of data should yield better results. Not only there is no guarantee that approximating differential operators through learning convolution kernels or performing automatic differentiations provide a consistent or even stable numerical method, but also the learning of convolution kernels to approximate differential operators requires more data and therefore yield less data-efficient methods. Therefore it seems reasonable to explore the idea of blending classic numerical discretization methods in neural network architectures, hence informing the neural network about proper discretization methods. This is the focus of the present manuscript.

In the present work, we discard the framework of constrained optimization altogether and instead choose to explicitly blend fully traditional finite discretization schemes as the decoder layer in semantic autoencoder architectures. In our approach, the loss function is

only composed of the difference between the actual data and the predictions of the solver layer, but contrary to BN17 [50] we do not consider the adjoint PDE problem to compute gradients of the error functional with respect to network parameters. This is due to the fact that in our design the numerical solver is a custom layer inside the neural network through which backpropagation occurs naturally. This is also in contrast to PINNs where the entire PDE, its boundary and its initial conditions are reproduced by the output of a neural network by adding them to the loss function. Importantly, the encoder learns an approximation of the inverse transform in a *self-supervised* fashion that can be used to evaluate the hidden fields underlying unseen data without any further optimization. Moreover, the proposed framework is versatile as it allows for straightforward consideration of other domain-specific knowledge such as symmetries or constraints on the hidden field. In this work, we develop this idea for stationary and time-dependent PDEs on structured and unstructured grids and on noisy data using mesh-based and mesh-less numerical discretization methods.

2.2.3 Novelties and features of BiPDEs

A full PDE solver is implemented as a *custom layer inside the architecture of semantic autoencoders* to solve inverse-PDE problems in a self-supervised fashion. Technically this is different than other works that implement a propagator decoder by manipulating activation functions or kernels/biases of convolutional layers, or those that feed the output of a neural network to a separate numerical solver such as in BN17 which requires the burden of considering the adjoint problem in order to compute partial differentiations. The novelties and features of this framework are summarized below:

1. **General discretizations.** We do not limit numerical discretization of differential equations to only finite differences that are emulated by convolution operations,

our approach is more general and permits employing more sophisticated numerical schemes such as meshless discretizations. It is a more general framework that admits any existing discretization method directly in a decoder stage.

2. **Introducing solver layers.** All the information about the PDE system is *only* localized in a solver layer; *i.e.* we do not inform the optimizer or the loss function with the adjoint PDE problem, or engineer regularizers or impose extra constraints on the kernels of convolutions, or define exotic activation functions as reviewed above. In other words, PDE solvers are treated as custom layers similar to convolution operations that are implemented in convolutional layers. An important aspect is the ability to employ any of the usual loss functions used in deep learning, for example we arbitrarily used mean absolute error or mean squared error in our examples.
3. **Blending meshless methods with trainable parameters.** Another unique proposal made in this work is the use of Radial Basis Function (RBF) based PDE solver layers as a natural choice to blend with deep neural networks. Contrary to other works, the neural network is not only used as an estimator for the unknown field but also it is tasked to optimize the shape parameters and interpolation points of the RBF scheme. In fact, our meshless decoder is not free of trainable parameters similar to reviewed works, instead shape parameters and seed locations are trainable parameters that define the RBF discretization, this is analogous to convolutional layers with trainable weights/biases that are used in machine learning domain. In fact this presents an example of neural networks complementing numerical discretization schemes. Choosing optimal shape parameters or seed locations is an open question in the field of RBF-based PDE solvers and here we show neural networks can be used to optimally define these discretization parameters.

4. **Explicit/implicit schemes.** Most of the existing frameworks only accept explicit numerical discretizations in time, however our design naturally admits implicit methods as well. Using implicit methods allows taking bigger timesteps for stiff problems such as the diffusion problem, hence not only providing faster inverse-PDE solvers, but also present more robust/stable inverse PDE solvers.
5. **Data efficient.** Our design lowers the computational cost as a result of reusing classical numerical algorithms for PDEs during the learning process, which focuses provided data to infer the actual unknowns in the problem, *i.e.* reduces the load of learning a discretization scheme from scratch.
6. **Physics informed.** Domain-specific knowledge about the unknown fields, such as symmetries or specialized basis functions, can be directly employed within our design.
7. **Inverse transform.** After training, the encoder can be used independently as a real-time estimator for unknown fields, *i.e.* without further optimization. In other words, the network can be pre-trained and then used to infer unknown fields in real-time applications.

2.3 Blended inverse-PDE network (BiPDE-Net)

The basic idea is to embed a numerical solver into a deep learning architecture to recover unknown functions in inverse-PDE problems, and all the information about the governing PDE system is only encoded inside the DNN architecture as a solver layer. In this section we describe our proposed architectures for inverse problems in one and two spatial dimensions.

2.3.1 Deep neural networks (DNN)

The simplest neural network is a single layer of perceptron that mathematically performs a linear operation followed by a nonlinear composition applied to its input space,

$$\mathcal{N} = \sigma(\mathbf{W}\mathbf{x} + \mathbf{b}), \quad (2.1)$$

where σ is called the *activation function*. Deep neural networks are multiple layers stacked together within some architecture. The simplest example is a set of layers connected in series without any recurrent loops, known as feedforward neural networks (FNN). In a densely connected FNN, the action of the network is simply the successive compositions of previous layer outputs with the next layers, *i.e.*,

$$\mathcal{N}_l = \sigma(\mathbf{W}_l \mathcal{N}_{l-1}(\mathbf{x}) + \mathbf{b}_l), \quad (2.2)$$

where l indicates the index of a layer. This compositional nature of NNs is the basis of their vast potential as universal function estimators of any arbitrary function on the input space \mathbf{x} , see e.g. [54, 55, 56]. Another important feature of NNs is that they can effectively express certain high dimensional problems with only a few layers, for example Darbon *et al.* [57] have used NNs to overcome the curse of dimensionality for some Hamilton-Jacobi PDE problems (also see [58, 46]).

Most machine learning models are reducible to composition of simpler layers which allows for more abstract operations at a higher level. Common layers include dense layers as described above, convolutional layers in convolutional neural networks (CNNs) [59, 60], Long-short term memory networks (LSTM) [61], Dropout layers [62] and many more. In the present work, we pay particular attention to CNNs owing to their ability to extract complicated spatial features from high dimensional input datasets. Furthermore,

we define custom PDE solver layers as new member of the family of pre-existing layers by directly implementing numerical discretization schemes inside the architecture of deep neural networks.

2.3.2 Custom solver layers

A *layer* is a high level abstraction that plays a central role in existing deep learning frameworks such as TensorFlow¹ [23], Keras API [63], PyTorch [64], *etc.* Each Layer encapsulates a state, *i.e.* trainable parameters such as weights/biases, and a transformation of inputs to outputs. States in a layer could also be non-trainable parameters in which case they will be excluded from backpropagation during training.

We implement different explicit or implicit numerical discretization methods as custom layers that transform an unknown field, initial data and boundary conditions to outputs in the solution space. Solver layers encapsulate numerical discretization schemes with trainable (*e.g.* shape parameters and seeds in meshless methods) or non-trainable (*e.g.* the finite difference methods) state parameters. Interestingly, solver layers with trainable parameters are new computational objects analogous to pre-existing convolutional layers with trainable kernel parameters.

An important aspect of layer objects is that they can be composed with other layers in any order. Particularly, this offers an interesting approach for solving inverse problems given by systems of partial differential equations with several unknown fields that can be modeled with neural layers. We will explore this avenue in future work. In the remainder of this manuscript we will only focus on different inverse-PDE examples given by a single PDE equation and one unknown field.

¹For example see TensorFlow manual page at https://www.tensorflow.org/guide/keras/custom_layers_and_models

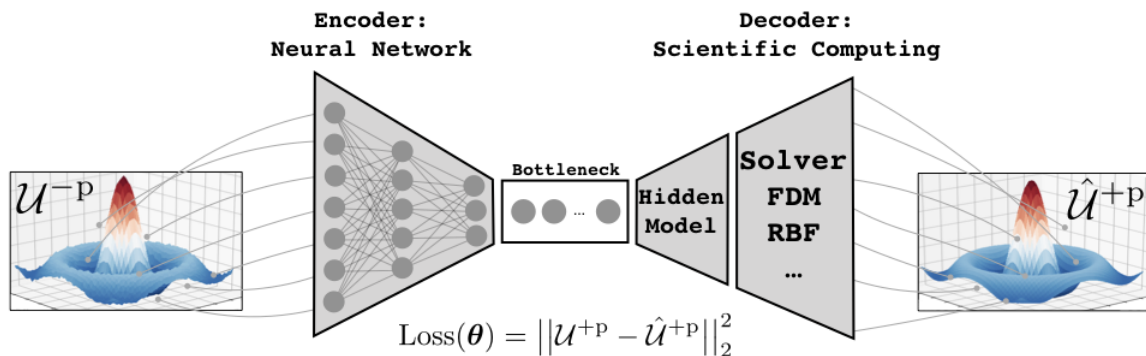


Figure 2.1: Architecture of the BiPDE to infer unknown parameters of hidden fields. Here the loss function is the mean squared error between data and output of the autoencoder, however other choices for loss function may be used depending on the nature of data.

2.3.3 Blended neural network architectures

BiPDE is a two-stage architecture, with the first stage responsible for learning the unknown coefficients and the second stage performing numerical operations as in traditional numerical solvers (see figure 2.1). To achieve higher performance, it is essential to use GPU-parallelism. We leverage the capability provided by the publicly available library TensorFlow [23] by implementing our PDE-solver as a *custom layer* into our network using the Keras API [63]. Details of this includes vectorized operations to build the linear system associated by the PDE discretization.

We propose a semantic autoencoder architecture as proposed by Aragon-Calvo (2019) [39] with hidden parameters being represented at the bottleneck of the autoencoder. Figure 2.1 illustrates the architecture for the proposed semantic autoencoder. Depending on static or time dependent nature of the governing PDE, one may train this network over pairs of input-output solutions that are shifted p steps in time, such that for a static PDE we have $p = 0$ while dynamic PDEs correspond to $p \geq 1$. We call this parameter the *shift parameter*, which will control the accuracy of the method (*cf.* see section 2.5).

An important aspect is that the input to BiPDE is the solution data itself. In other words the neural network in a BiPDE is learning the *inverse transform*,

$$\mathcal{NN} : \{u\} \rightarrow \text{hidden field}, \quad (2.3)$$

where $\{u\}$ indicates an ensemble of solutions, *e.g.* solutions obtained with different boundary conditions or with different hidden fields. Note that in other competing methods such as PINNs the input is sanctioned to be the coordinates in order for automatic differentiation to compute spatial and temporal derivatives; as a consequence PINNs can only be viewed as *surrogates* for the solution of the differential problem defined on the space of coordinates. However, we emphasize that semantic autoencoders are capable to approximate the inverse transformation from the space of solutions to the space of hidden fields, a feature that we exploit in section 2.4.1.

Essentially different numerical schemes can be implemented in the decoder stage. We will blend examples of both mesh-based and mesh-less numerical discretizations and present numerical results and comparisons with PINNs. We will show how BiPDEs can handle data on unstructured grids and data with added noise. In section 2.4, we demonstrate performance of mesh-based BiPDEs on inverse problems in two spatial dimensions by using a finite difference discretization and Zernike expansion of the non-homogeneous hidden field, we will consider both stationary and dynamic PDE problems in this section. Then in section 2.5, we develop a mesh-less BiPDE and consider a dynamic nonlinear inverse partial differential problem.

2.4 Mesh-based BiPDE: Finite Differences

We consider a variable coefficient Poisson problem in one and two spatial dimensions as well as the one dimensional nonlinear Burger’s equation as an example of a nonlinear dynamic PDE problem with a scalar unknown parameter.

2.4.1 Stationary Poisson problem

We consider the governing equation for diffusion dominated processes in heterogeneous media:

$$\nabla \cdot (D(\mathbf{x})\nabla u) = -f(\mathbf{x}), \quad \mathbf{x} \in \Omega \quad (2.4)$$

$$u(\mathbf{x}) = u_0(\mathbf{x}), \quad \mathbf{x} \in \partial\Omega \quad (2.5)$$

Here we consider a rectangular domain with Dirichlet boundary conditions.

Discretization. In our architecture, we use the standard 5-point stencil finite difference discretization of the Poisson equation in the solver layer, *i.e.*

$$\frac{D_{i-1/2,j}u_{i-1,j} - (D_{i-1/2,j} + D_{i+1/2,j})u_{i,j} + D_{i+1/2,j}u_{i+1,j}}{\Delta x^2} + \frac{D_{i,j-1/2}u_{i,j-1} - (D_{i,j-1/2} + D_{i,j+1/2})u_{i,j} + D_{i,j+1/2}u_{i,j+1}}{\Delta y^2} + f_{i,j} = 0,$$

and we use the linear algebra solver implemented in `TensorFlow` to solve for the solution field, *i.e.* we used `tf.linalg.solve` method that is a dense linear system solver. Of course, this can be improved by implementing a sparse linear solver.

Hidden Model. We decompose the hidden field into a finite number of eigenfunctions and search for their optimal coefficients. This is also advantageous from a physics point of view, because domain’s knowledge of hidden fields can be naturally formulated

in terms of basis functions into this framework. One such family of series expansions are the moment-based methods that have been largely exploited in image reconstruction [65, 66, 67, 68]. In particular, Zernike moments [69] provide a linearly independent set of polynomials defined on the unit circle/sphere in two/three spatial dimensions. Zernike moments are well-suited for such a task and are commonly used for representing optical aberration in astronomy and atmospheric sciences [70], for image reconstruction and for enhanced ultrasound focusing in biomedical imaging [71, 72, 73].

Zernike moments are advantageous over regular moments in that they intrinsically provide rotational invariance, higher accuracy for irregular patterns, and are orthogonal, which reduces information redundancy in the different coefficients. Zernike polynomials capture deviations from zero mean as a function of radius and azimuthal angle. Furthermore, the complete set of orthogonal bases provided by Zernike moments can be obtained with lower computational precision from input data, which enhances the robustness of the reconstruction procedure.

Odd and even Zernike polynomials are given as a function of the azimuthal angle θ and the radial distance ρ between 0 and 1 measured from the center of image,

$$\begin{bmatrix} Z_{nm}^o(\rho, \theta) \\ Z_{nm}^e(\rho, \theta) \end{bmatrix} = R_{nm}(\rho) \begin{bmatrix} \sin(m\theta) \\ \cos(m\theta) \end{bmatrix},$$

with

$$R_{nm}(\rho) = \begin{cases} \sum_{l=0}^{(n-|m|)/2} \frac{(-1)^l (n-l)!}{l! [(n+|m|)/2-l]! [(n-|m|)/2-l]!} \rho^{n-2l} & \text{for } n - m \text{ even,} \\ 0 & \text{for } n - m \text{ odd,} \end{cases}$$

where n and m are integers with $n \geq |m|$. A list of radial components is given in table 2.1 (from [74]). For an extensive list of Zernike polynomials in both two and three spatial

n	 m 	R_{nm}	Z_{nm}^o	Z_{nm}^e	Aberration/Pattern
0	0	1	0	1	Piston
1	1	ρ	$\rho \sin(\theta)$	$\rho \cos(\theta)$	Tilt
2	0	$2\rho^2 - 1$	0	$2\rho^2 - 1$	Defocus
	2	ρ^2	$\rho^2 \sin(2\theta)$	$\rho^2 \cos(2\theta)$	Oblique/Vertical Astigmatism
3	1	$3\rho^3 - 2\rho$	$(3\rho^3 - 2\rho) \sin(\theta)$	$(3\rho^3 - 2\rho) \cos(\theta)$	Vertical/Horizontal Coma
	3	ρ^3	$\rho^3 \sin(3\theta)$	$\rho^3 \cos(3\theta)$	Vertical/Oblique Trefoil
4	0	$6\rho^4 - 6\rho^2 + 1$	0	$6\rho^4 - 6\rho^2 + 1$	Primary Spherical
	2	$4\rho^4 - 3\rho^2$	$(4\rho^4 - 3\rho^2) \sin(2\theta)$	$(4\rho^4 - 3\rho^2) \cos(2\theta)$	Oblique/Vertical Secondary Astigmatism
	4	ρ^4	$\rho^4 \sin(4\theta)$	$\rho^4 \cos(4\theta)$	Oblique/Vertical Quadrafoil

Table 2.1: First 15 odd and even Zernike polynomials according to Noll’s nomenclature. Here, the ordering is determined by ordering polynomial with lower radial order first, cf. [76].

dimensions, we refer the interested reader to [75].

Furthermore, each Zernike moment is defined by projection of the hidden field $f(x, y)$ on the orthogonal basis,

$$\begin{bmatrix} A_{nm} \\ B_{nm} \end{bmatrix} = \frac{n+1}{\epsilon_{mn}^2 \pi} \int_x \int_y f(x, y) \begin{bmatrix} Z_{nm}^o(x, y) \\ Z_{nm}^e(x, y) \end{bmatrix} dx dy, \quad x^2 + y^2 \leq 1,$$

where for $m = 0$, $n \neq 0$ we defined $\epsilon_{0n} = 1/\sqrt{2}$ and $\epsilon_{mn} = 1$ otherwise. Finally, superposition of these moments expands the hidden field in terms of Zernike moments:

$$\hat{f}(x, y) = \sum_{n=0}^{N_{max}} \sum_{|m|=0}^n [A_{nm} Z_{nm}^o(r, \theta) + B_{nm} Z_{nm}^e(r, \theta)]. \quad (2.6)$$

In order to identify the coefficients in the Zernike expansion (2.6) for hidden fields, we use a semantic autoencoder architecture with Zernike moments being represented by the code at the bottleneck of the autoencoder. Figure 2.2 illustrates the architecture for the proposed semantic autoencoder.

Architecture. Even though a shallow neural network with as few neurons as the number of considered Zernike terms suffices to estimate values of the unknown Zernike

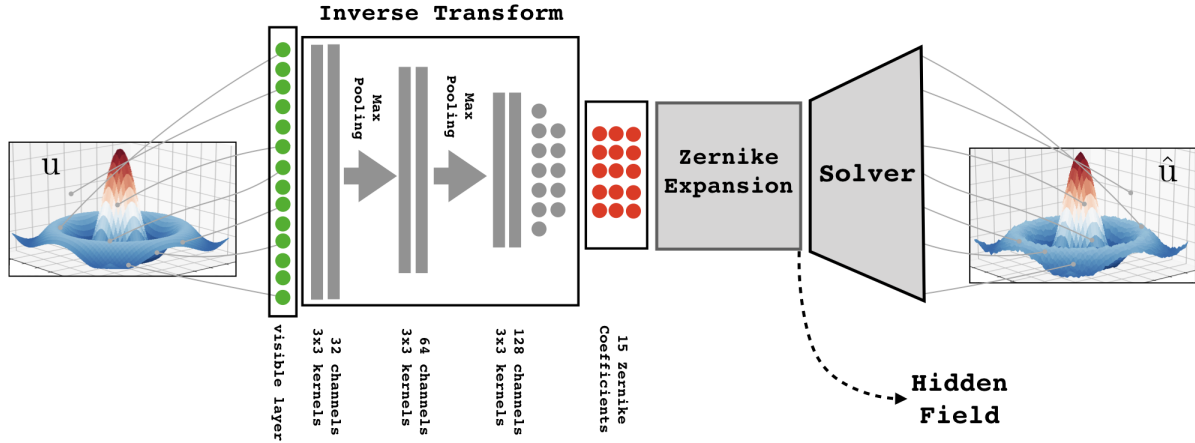


Figure 2.2: Architecture of the semantic autoencoder to infer hidden fields. Zernike moments are discovered at the bottleneck of the architecture.

moments in each of the problems considered in this section, however we will use a deep convolutional neural network (detailed below) in order to achieve our ultimate goal of approximating the inverse transform for the Poisson problem in a broad range of diffusion coefficient fields. Therefore we design one deep neural network and uniformly apply it to several problems in this section.

In the training process of a CNN, the kernels are trained at each layer such that several feature maps are extracted at each layer from input data. The CNN is composed of 3 convolutional blocks with 32, 64, 128 channels respectively and kernel size 3×3 . Moreover, we use the **MaxPooling** filter with kernel size $(2, 2)$ after each convolutional block to downsample the feature maps by calculating the maximum values of each patch within these maps. We use the **ReLU** activation function [77], *i.e.* a piecewise linear function that only outputs positive values: $\text{ReLU}(x) = \max(0, x)$, in the convolutional layers followed by a **Sigmoid** activation in dense layers and a scaled **Sigmoid** activation

at the final layer,

$$\tilde{\sigma}(x) = D_{\min} + (D_{\max} - D_{\min})\sigma(x), \quad (2.7)$$

such that the actual values of the diffusion coefficient are within the range (D_{\min}, D_{\max}) , known from domain specific knowledge. After each dense layer, we apply `Dropout` layers with a rate of 0.2 to prevent overfitting [78, 62] (a feature that is most useful in estimating the inverse transform operator) and avoid low quality local minima during training.

Test cases.

Case I. A tilted plane. In the first example we consider a linear diffusion model given by

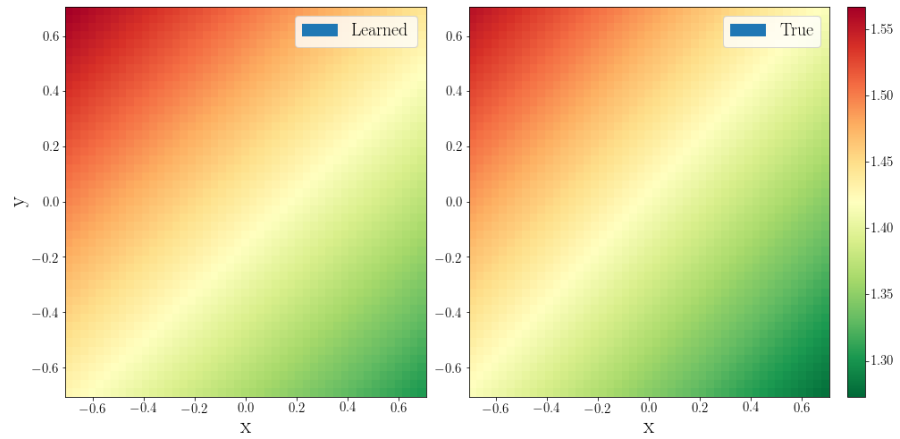
$$D(x, y) = \sqrt{2} + 0.1(y - x)$$

where the boundary condition function u_{BC} and the source field f are given by

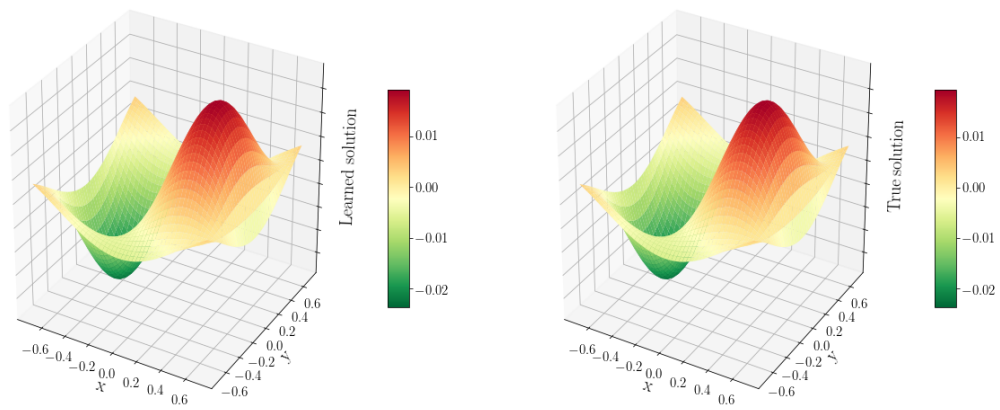
$$u_{BC}(x, y) = 0.01 \cos(\pi x) \cos(\pi y) \quad \text{and} \quad f(x, y) = \sin(\pi x) \cos(\pi y)$$

In this experiment we only use a *single solution field* for training. Even though in our experiments the method succeeded to approximate the hidden field even with a *single grid point* to compute the loss function, here we consider all the grid points in the domain to obtain improved accuracy in the results. We trained the network for 30 epochs using an `Adam` optimizer [79] that takes 170 seconds on a Tesla T4 GPU available on a free Google Colaboratory account². Figure 2.3 depicts the results obtained by the proposed scheme. The diffusion map is discovered with a maximum relative error of only 2%, while

²<https://colab.research.google.com/>

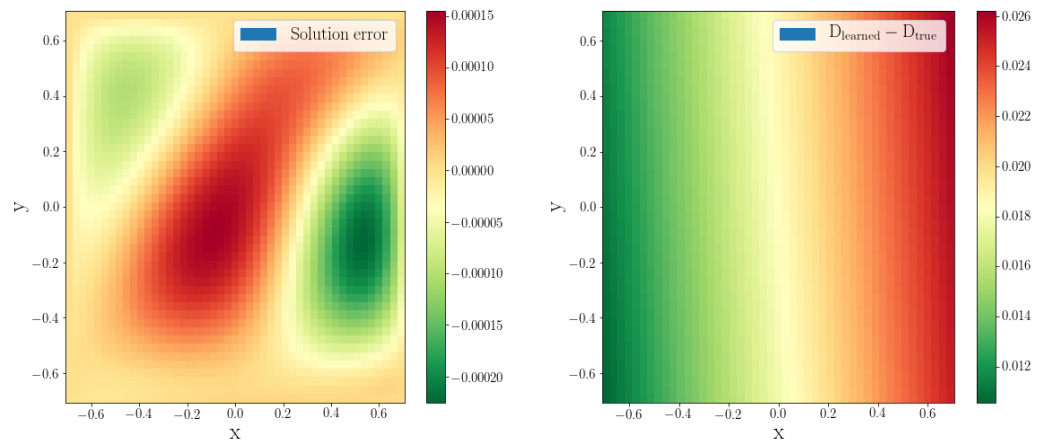


(a) Comparison of learned (left) versus true diffusion coefficient (right).



(b) Learned solution.

(c) True solution.

(d) Error in learned solution $u - \hat{u}$.

(e) Error in learned diffusion coefficient.

Figure 2.3: Results for the two dimensional tilted plane (case I).

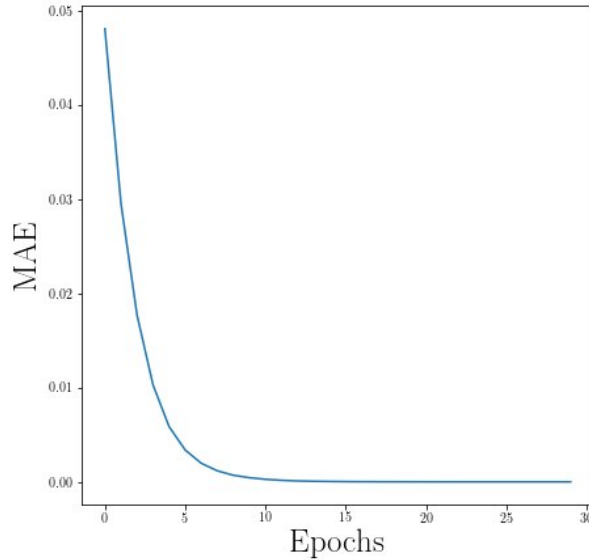


Figure 2.4: Mean absolute error in solution vs. epochs for the two dimensional tilted plane (case I).

the error in the solution field is 1%. It is noteworthy to mention that the accuracy of the results in this architecture are influenced by the accuracy of the discretizations used in the solver layer. While we used a second-order accurate finite difference discretization, it is possible to improve these results by using higher order discretizations instead. We leave such optimizations to future work.

Influence of architecture. Table 2.2 tabulates the mean absolute error in the discovered tilted plane diffusion coefficient for different architectures of the encoder stage. No significant improvement is observed for deeper or shallower encoder network for the example considered here.

Case II. superimposed Zernike polynomials. We consider a more complicated hidden diffusion field given by

$$D(x, y) = 4 + a_0 + 2a_1x + 2a_2y + \sqrt{3}a_3(2x^2 + 2y^2 - 1).$$

T	# params	C(32)	C(32)	C(64)	C(64)	C(128)	C(128)	D(64)	D(32)	MAE _D	L _D [∞]
1	1,468,323	Y	Y	Y	Y	Y	Y	Y	Y	0.0144207	0.0294252
2	1,459,075	Y	-	Y	Y	Y	Y	Y	Y	0.0193128	0.0267854
3	1,422,147	Y	-	Y	-	Y	Y	Y	Y	0.0226252	0.0527432
4	1,274,563	Y	-	Y	-	Y	-	Y	Y	0.0199361	0.0272122
5	682,627	Y	-	Y	-	Y	-	-	Y	0.0141946	0.0243868
6	313,859	Y	-	Y	-	-	-	-	Y	0.0301841	0.0544990
7	46,467	Y	-	Y	-	-	-	-	-	0.0190432	0.0264254
8	6,915	-	-	-	-	-	-	-	-	0.0183808	0.0267156

Table 2.2: Influence of architecture of the decoder stage on mean absolute error $\text{MAE}_D \equiv \sum |\mathbf{D}(\mathbf{x}) - \hat{\mathbf{D}}(\mathbf{x})|/N$ and maximum error L_D^∞ in the discovered hidden field in case I. Double vertical lines correspond to `MaxPooling2D()` layers and triple vertical lines correspond to `Flatten()` layer. `C(o)` and `D(o)` stand for `conv2D(filters)` and `Dense(neurons)` layers respectively. There are 3 neurons at the bottleneck not shown in the table.

The boundary condition function u_{BC} and the source field f are given by

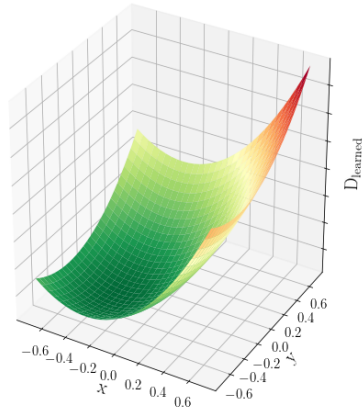
$$u_{BC}(x, y) = \cos(\pi x) \cos(\pi y) \quad \text{and} \quad f(x, y) = x + y.$$

Figure 2.5 illustrates the performance of the proposed Zernike-based network using a mean absolute error measure for the loss function. We trained the network for 100 epochs using an Adam optimizer [79].

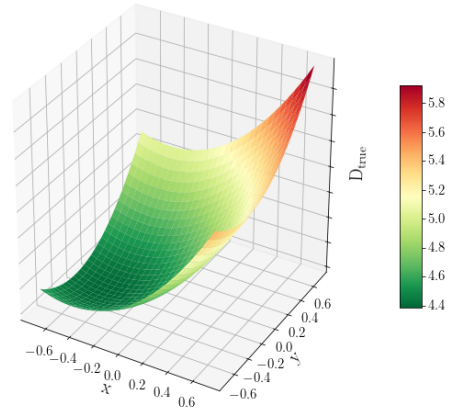
Resilience to noise. We also assess the performance of our scheme on noisy datasets. We consider a zero-mean Gaussian noise with standard deviation 0.025 superimposed on the solution field. Figure 2.6 depicts the solution learned from a noisy input image. The network succeeds in discovering the diffusion field with comparable accuracy as in the noise-free case. Note that this architecture naturally removes the added noise from the learned solution, a feature that is similar to applying a low-pass filter on noisy images.

Learning the inverse transform

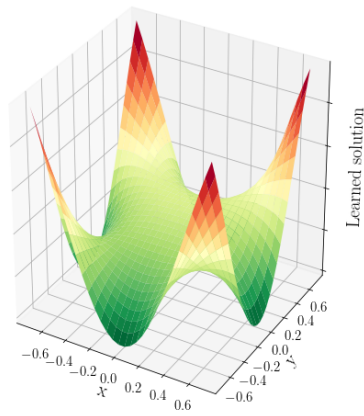
In the previous sections, we have applied BiPDE to find the variable diffusion coefficient from a single input image. Another interesting feature of the proposed semantic



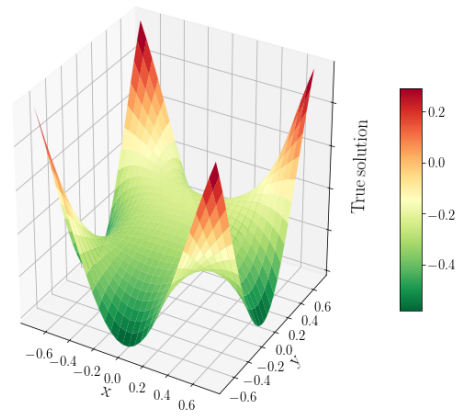
(a) Learned diffusion.



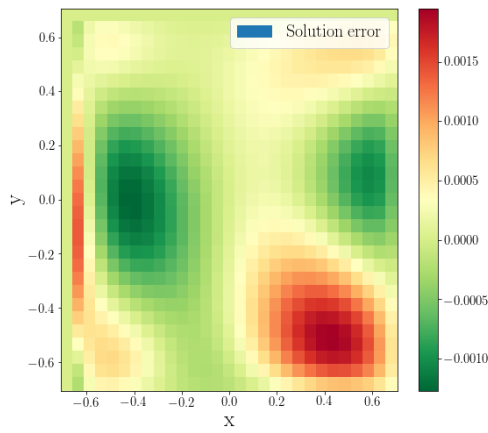
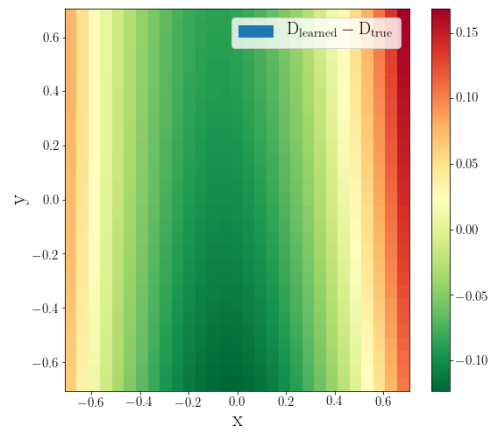
(b) True diffusion.



(c) Learned solution.



(d) True solution.

(e) Error in learned solution $u - \hat{u}$.

(f) Error in learned diffusion coefficient.

Figure 2.5: Results in the two dimensional parabolic case.

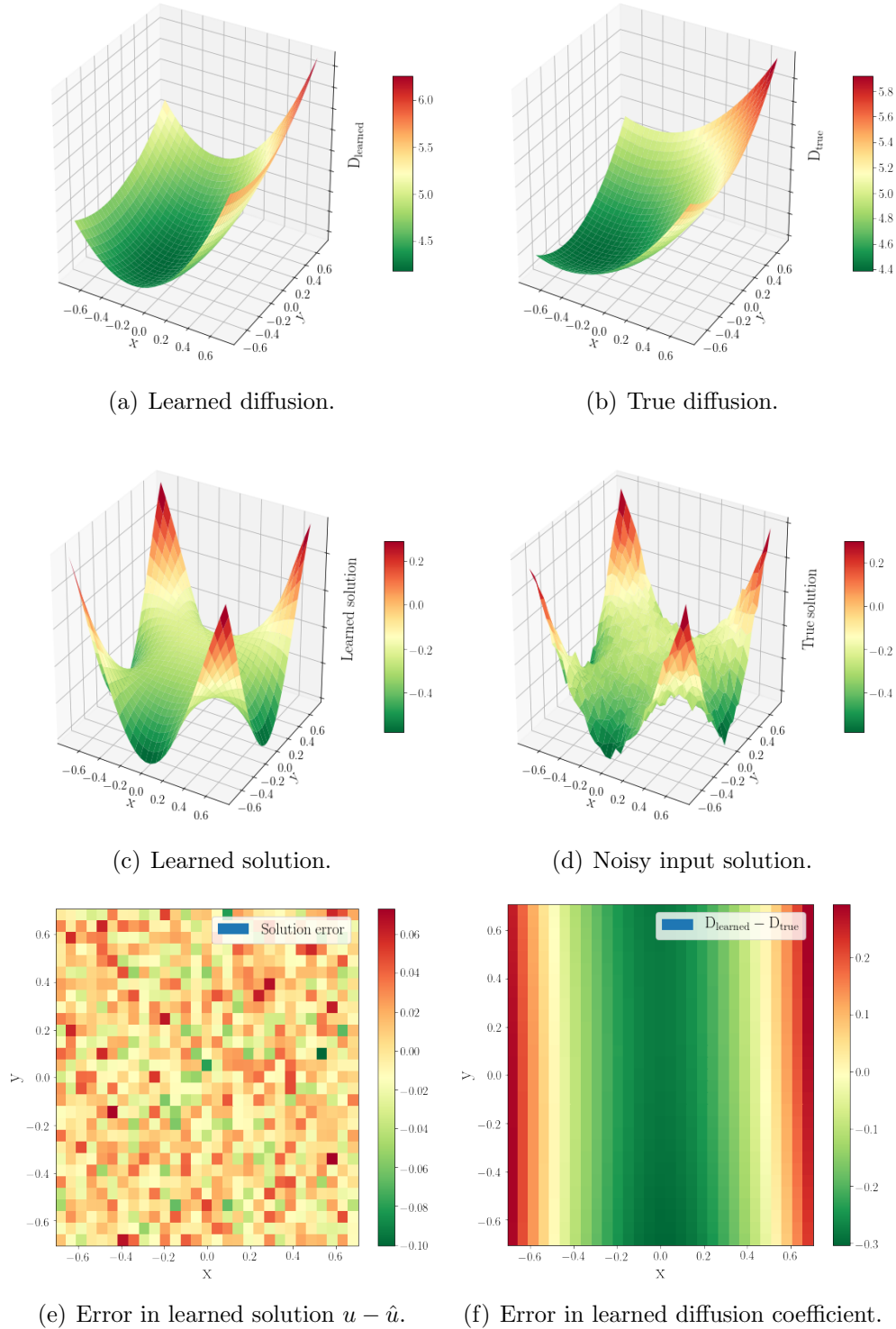


Figure 2.6: Results in the two dimensional case with added noise. After 300 epochs the network discovers the hidden diffusion field with a maximum relative error of 5%. Interestingly the learned solution is resilient to added noise and the network approximates a noise-free solution.

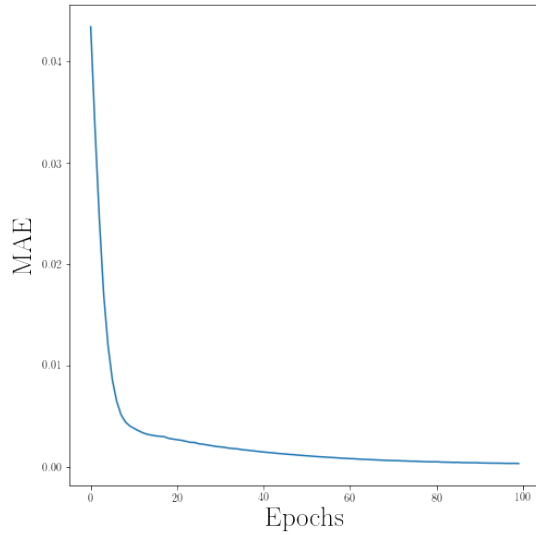
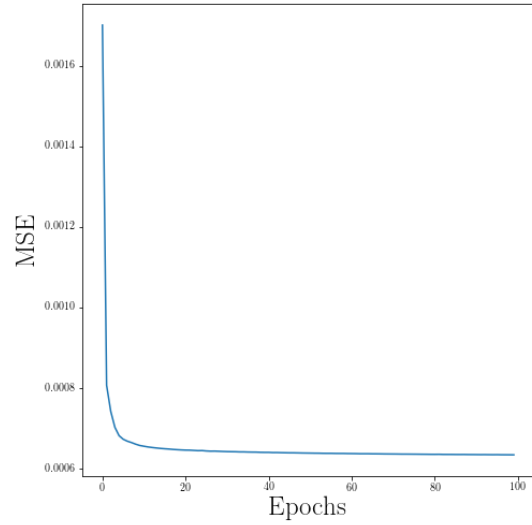
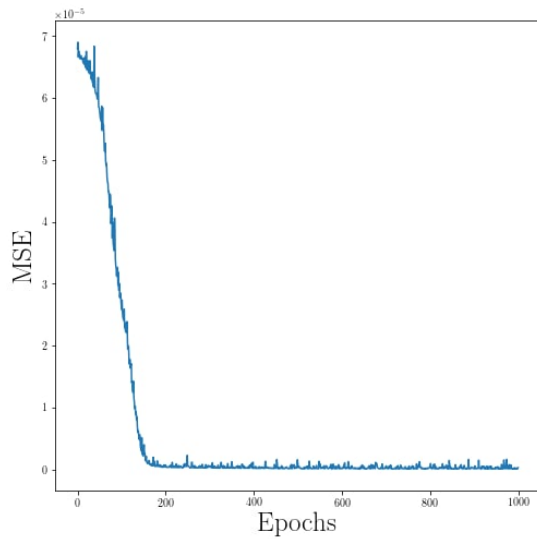
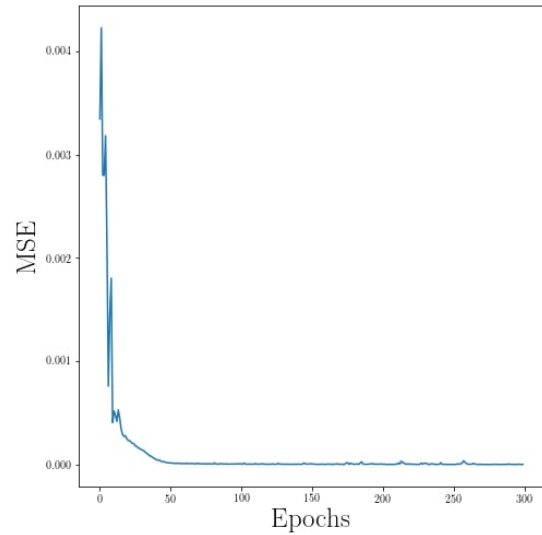
(a) L_1 loss vs. epoch for case II without added noise.(b) L_2 loss vs. epoch for case II with added noise.(c) L_2 loss vs. epoch for 1D inverse transform.(d) L_2 loss vs. epoch for 2D inverse transform.

Figure 2.7: Mean absolute/square error vs. epochs for (top panel) the two dimensional parabolic experiment (case II) with and without added Gaussian noise of section 2.4.1, and (bottom panel) the inverse transform for 1D and 2D experiments of section 2.4.1.

autoencoder architecture is its ability to train neural networks in order to discover the inverse transform for the underlying hidden fields *in a self-supervised fashion*. In this scenario, the trained encoder learns the inverse transform function that approximates the hidden parameters given a solution field to its input. Note that even though the same task could be accomplished by supervised learning of the hidden fields, *i.e.* by explicitly defining loss on the hidden fields without considering the governing equations, BiPDEs substitute the data labels with a governing PDE and offer comparable prediction accuracy. In this section we train BiPDEs over ensembles of solution fields to estimate hidden Zernike moments of diffusion coefficients underlying unseen data.

One dimensional inverse transform

We build a one dimensional semantic autoencoder using 3 layers with 100, 40, and 2 neurons respectively. We used the **ReLU** activation function for the first two layers and a **Sigmoid** activation function for the last layer representing the hidden parameters. A linear solver is then stacked with this encoder that uses the second order accurate finite difference discretization, *i.e.*

$$\frac{D_{i-1/2}u_{i-1} - (D_{i-1/2} + D_{i+1/2})u_i + D_{i+1/2}u_{i+1}}{\Delta x^2} + f_i = 0, \quad D_{i+1/2} = \frac{D_i + D_{i+1}}{2}$$

However, the diffusion map is internally reconstructed using the hidden parameters before feeding the output of the encoder to the solver. As a test problem, we consider the one dimensional Poisson problem with a generic linear form for the diffusion coefficient,

$$D(x) = 1 + a_0 + a_1x.$$

We consider identical left and right Dirichlet boundary conditions of 0.2 for all images and let the source term be $f(x) = \sin(\pi x)$. We consider random diffusion coefficients

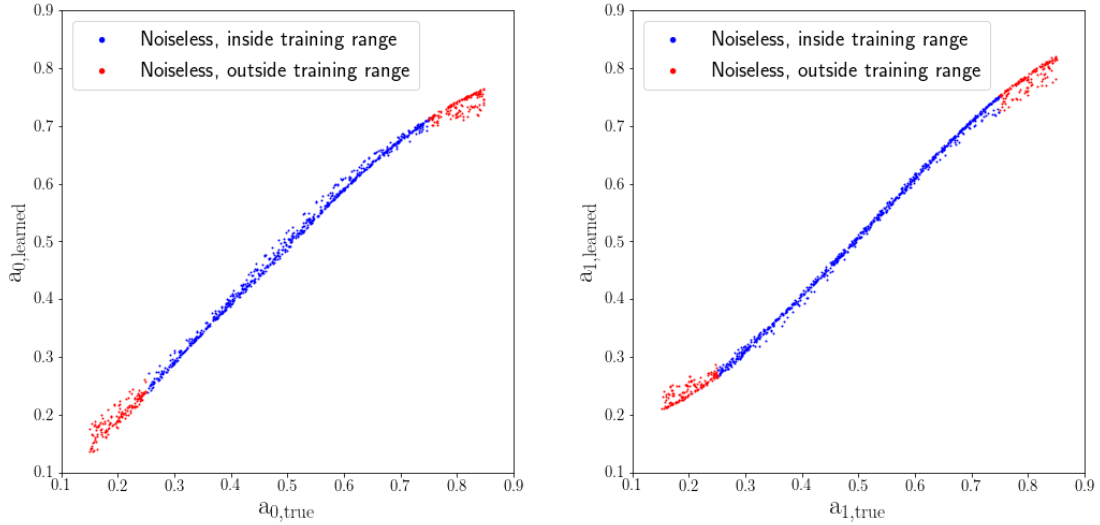
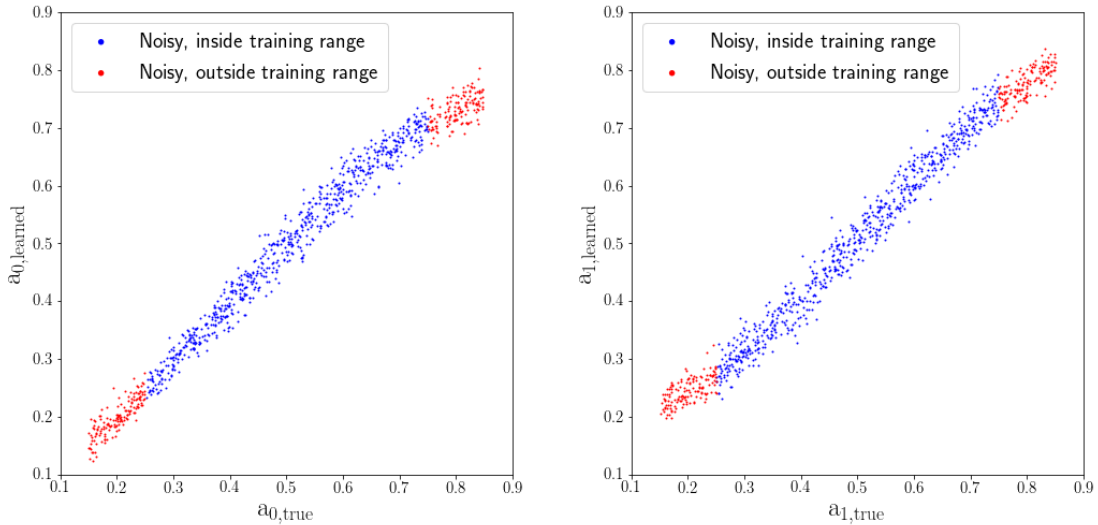
(a) Regression quality is $R^2 = 0.9906891$.(b) Regression quality is $R^2 = 0.9953392$.(c) Regression quality is $R^2 = 0.9796781$.(d) Regression quality is $R^2 = 0.9834912$.

Figure 2.8: (Top, bottom) panel shows performance of BiPDE over 1000 randomly chosen one-dimensional images with $N_x = 160$ grid points after 1000 epochs (with,without) added zero-mean Gaussian noise with standard deviation 0.025 to the test sample. The hidden diffusion coefficient is $D(x) = 1 + a_0 + a_1x$. In each case the R^2 coefficient is reported for the blue data points, where unknown parameters fall within the training range $[0.25, 0.75]$. Red data points show predictions outside of training range. Network has 20,222 trainable parameters, and training takes ~ 2 seconds per epoch on a Tesla T4 GPU available on a free Google Colaboratory account.

a_0 and a_1 with a uniform distribution in $[0.25, 0.75]$ and we generate 1000 solutions over the domain $x \in [-1, 1]$. We train BiPDE over 900 images from this dataset and validate its performance over the remaining 100 images using a mean squared error loss function for 1000 epochs. Each image is generated on a uniform grid with $N_x = 160$ grid points. We used a batch size of 100 in these experiments using the Adam optimizer. Figure 2.7(c) shows loss versus epochs in this experiment. Figure 2.8 compares learned and true coefficients over two independent test samples containing 1000 solutions, with and without a zero-mean Gaussian noise with standard deviation 0.025, *i.e.* amounting to $\sim 13\%$ added noise over the images.

In figure 2.8, we expanded the range of unknown parameters $a_0, a_1 \in [0.15, 0.85]$ in our test sample to assess performance of trained encoder over unseen data that are outside the range of training data (as a measure of generalizability of BiPDEs). In this figure blue points correspond to new images whose true unknown parameters fall inside the training range, and red data points correspond to those outside the training range. We observe that the encoder is able to predict the unknown parameters even outside of its training range, although its accuracy gradually diminishes far away from the training range. Note that the predicted values for a_0 and a_1 exhibit a systematic error towards the lower and upper bounds of the **Sigmoid** activation function, indicative of the influence of the **Sigmoid** activation function used in the last layer. This emphasizes the significance of properly designing activation functions at the bottleneck.

Using the R^2 statistical coefficient as a measure of accuracy for the trained encoder, we assess effects of sample size and grid points on the performance of BiPDEs and report the results in table 2.3.

1. *Effect of sample size:* First, we fix number of grid points and vary sample size. We find that increasing sample size improves accuracy of the predictions in the case of

1D inverse transform	Noiseless		Noisy (13% relative noise)	
Sample Size, $N_x = 100$	a_0	a_1	a_0	a_1
$N_{\text{data}} = 250$	0.9953634	0.9977753	0.9609166	0.9570264
$N_{\text{data}} = 500$	0.9979478	0.9988417	0.9644154	0.9640230
$N_{\text{data}} = 1000$	0.9990417	0.9992921	0.9600430	0.9586783
$N_{\text{data}} = 2000$	0.9995410	0.9997107	0.9599427	0.9652383
$N_{\text{data}} = 4000$	0.9994279	0.9994974	0.9603496	0.9661519
$N_{\text{data}} = 8000$	0.9998054	0.9998115	0.9614795	0.9619859
Grid Points, $N_{\text{data}} = 1000$	a_0	a_1	a_0	a_1
$N_x = 20$	0.9900532	0.9987560	0.8623348	0.8822680
$N_x = 40$	0.9932568	0.9975166	0.9161125	0.9081806
$N_x = 80$	0.9986574	0.9993274	0.9509870	0.9511483
$N_x = 160$	0.9991550	0.9990234	0.9747287	0.9762977
$N_x = 320$	0.9985649	0.9987451	0.9861375	0.9860783
$N_x = 640$	0.9991842	0.9991606	0.9920950	0.9922520

Table 2.3: R^2 coefficient for predicted Zernike coefficients of the one dimensional Poisson problem at different training sample size and number of grid points.

clean data, however in the case of noisy data the accuracy does not show significant improvement by enlarging sample size.

2. *Effect of grid points:* Second, we fix the sample size and gradually increase number of grid points. We find that accuracy of predictions on noisy data is strongly correlated with number of grid points, however this dependence is weaker for clean data.

Two dimensional inverse transform

We consider an example of variable diffusion coefficients parameterized as $D(x, y) = 4 + 2a_2y + \sqrt{3}a_3(2x^2 + 2y^2 - 1)$, with unknown coefficients randomly chosen in range $a_2, a_3 \in [0.25, 0.75]$. The equations are solved on a square domain $\Omega \in [-\frac{1}{\sqrt{2}}, \frac{1}{\sqrt{2}}]^2$

governed by the Poisson equation:

$$\begin{aligned}\nabla \cdot ([4 + 2a_2y + \sqrt{3}a_3(2x^2 + 2y^2 - 1)]\nabla u) + x + y &= 0, & (x, y) \in \Omega, \\ u_{BC} &= \cos(\pi x) \cos(\pi y), & (x, y) \in \partial\Omega.\end{aligned}$$

The encoder is composed of two convolutional layers with 32 and 64 channels followed by a 2×2 average pooling layer and a dense layer with 128 neurons, at the bottleneck there are 2 neurons representing the two unknowns. All activation functions are **ReLU** except at the bottleneck that has **Sigmoid** functions. An **Adam** optimizer is used on a mean squared error loss function.

We trained BiPDE over 900 generated solutions with randomly chosen parameters a_2, a_3 and validated its performance on 100 independent solution fields for 300 epochs, evolution of loss function is shown in figure 2.7(d). Then we tested the trained model over another set of 1000 images with randomly generated diffusion maps independent of the training dataset. Furthermore, we repeated this exercise over 1000 images with added zero-mean Gaussian noise with standard deviation 0.025. In each case, the learned parameters are in good agreement with the true values, as illustrated in figure 2.9. Moreover, we performed a sensitivity analysis on the accuracy of the predictions with respect to sample size. We measured quality of fit by the R^2 statistical coefficient. Results are tabulated in table 2.4 and indicate training over more samples leads to more accurate predictions when applied to clean data, while noisy data do not show a strong improvement by increasing sample size.

2.4.2 Dynamic Burger's problem

In this section, we demonstrate the applicability of BiPDEs on time-dependent non-linear partial differential equations, and we use those results to illustrate the consistency

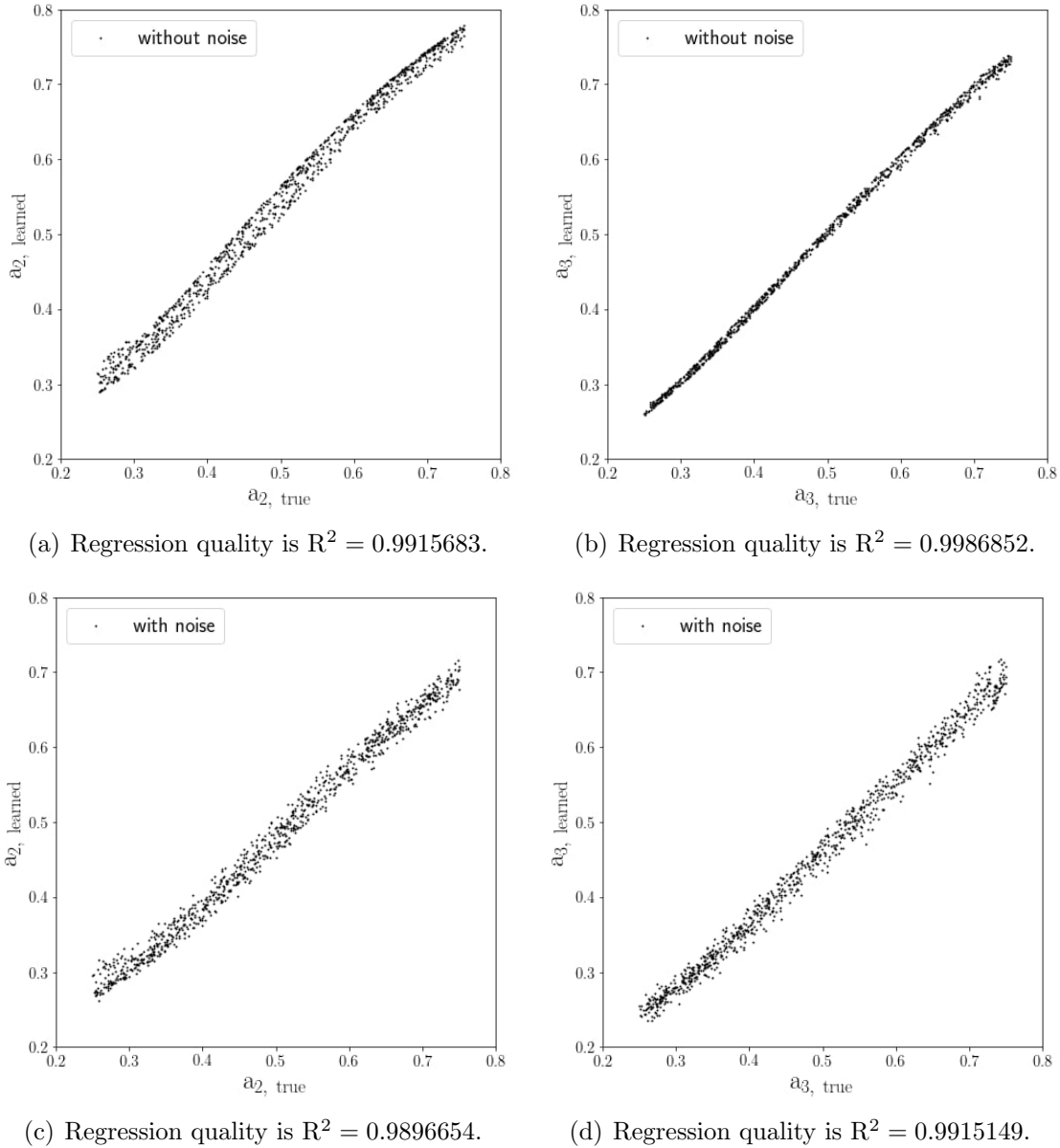


Figure 2.9: Top row shows performance of BiPDE over 1000 randomly chosen clean 2D images after 1000 epochs, and the bottom panel shows performance of the same network on noisy images given a zero-mean Gaussian noise with standard deviation 0.025. Network has 1,852,000 trainable parameters and training takes ~ 11 seconds on a Tesla T4 GPU available on a free Google Colaboratory account.

2D inverse transform	Noiseless		Noisy (13% relative noise)	
Sample Size	a_2	a_3	a_2	a_3
$N_{\text{data}} = 250$	0.9897018	0.9958963	0.9872887	0.9892064
$N_{\text{data}} = 500$	0.9917211	0.9977917	0.9910183	0.9900091
$N_{\text{data}} = 1000$	0.9915683	0.9986852	0.9896654	0.9915149
$N_{\text{data}} = 2000$	0.9940470	0.9993891	0.9909640	0.9883151
$N_{\text{data}} = 4000$	0.9938268	0.9997119	0.9919061	0.9898697

Table 2.4: R^2 coefficient for predicted Zernike coefficients of the two dimensional Poisson problem by increasing training sample size. Number of grid points are fixed at 30×30 .

and accuracy of the proposed framework. Similar to previous works [45], we consider the nonlinear Burgers' equation in one spatial dimension,

$$\frac{\partial u}{\partial t} + u \frac{\partial u}{\partial x} = \nu \frac{\partial^2 u}{\partial x^2} \quad x \in [-1, 1], \quad t \in [0, 1) \quad (2.8)$$

$$u(-1, t) = u(1, t) = 0 \quad u(x, 0) = -\sin(\pi x) \quad (2.9)$$

where $\nu = 1/Re$ with Re being the Reynolds number. Notably, Burgers' equation is of great practical significance for understanding evolution equations as it is nonlinear. Burgers' equation has been used as a model equation for the Navier-Stokes equation and by itself can be used to describe shallow water waves [80], turbulence [81], traffic flow [82], and many more.

- **Discretization.** In our design we adopted the 6th-order compact finite difference scheme proposed by Sari and Gurarslan (2009) [83] for its simplicity of implementation, its high accuracy and because it leads to a linear system with narrow band and subsequently ensures computational efficiency. This scheme combines a tridiagonal³ sixth-order compact finite difference scheme (CFD6) in space with a low-storage third-order accurate total variation diminishing Runge-Kutta scheme

³Tridiagonal systems of equations can be obtained in $\mathcal{O}(N)$ operations.

(TVD-RK3) for its time evolution ([84]). In particular, the high-order accuracy associated with this discretization provides highly accurate results on coarse grids. This is an important aspect of BiPDEs as a data-efficient inverse solver, which stems from their capacity to seamlessly blend highly accurate and sophisticated discretization methods with deep neural networks.

The first-order spatial derivatives are given at intermediate points by

$$\alpha u'_{i-1} + u'_i + \alpha u'_{i+1} = b \frac{u_{i+2} - u_{i-2}}{4h} + a \frac{u_{i+1} - u_{i-1}}{2h}, \quad i = 3, \dots, N-2 \quad (2.10)$$

where

$$a = \frac{2}{3}(\alpha + 2), \quad b = \frac{1}{3}(4\alpha - 1), \quad (2.11)$$

and $h = x_{i+1} - x_i$ is the mesh size, with grid points identified by the index $i = 1, 2, \dots, N$. For $\alpha = 1/3$ we obtain a sixth order accurate tridiagonal scheme. The boundary points (for non-periodic boundaries) are treated by using the formulas [85, 83],

$$\begin{aligned} u'_1 + 5u'_2 &= \frac{1}{h} \left[-\frac{197}{60}u_1 - \frac{5}{12}u_2 + 5u_3 - \frac{5}{3}u_4 + \frac{5}{12}u_5 - \frac{1}{20}u_6 \right] \\ \frac{2}{11}u'_1 + u'_2 + \frac{2}{11}u'_3 &= \frac{1}{h} \left[-\frac{20}{33}u_1 - \frac{35}{132}u_2 + \frac{34}{33}u_3 - \frac{7}{33}u_4 + \frac{2}{33}u_5 - \frac{1}{132}u_6 \right] \\ \frac{2}{11}u'_{N-2} + u'_{N-1} + \frac{2}{11}u'_N &= \frac{1}{h} \left[\frac{20}{33}u_N + \frac{35}{132}u_{N-1} - \frac{34}{33}u_{N-2} + \frac{7}{33}u_{N-3} - \frac{2}{33}u_{N-4} + \frac{1}{132}u_{N-5} \right] \\ 5u'_{N-1} + u'_N &= \frac{1}{h} \left[\frac{197}{60}u_N + \frac{5}{12}u_{N-1} - 5u_{N-2} + \frac{5}{3}u_{N-3} - \frac{5}{12}u_{N-4} + \frac{1}{20}u_{N-5} \right] \end{aligned}$$

This can be easily cast in the matrix form

$$BU' = AU \tag{2.12}$$

where $U = [u_1, u_2, \dots, u_N]^T$ is the vector of solution values at grid points. Furthermore, second order derivatives are computed by applying the first-order derivatives twice⁴,

$$BU'' = AU' \tag{2.13}$$

Burgers' equation are thus discretized as:

$$\frac{dU}{dt} = \mathcal{L}U, \quad \mathcal{L}U = \nu U'' - U \otimes U', \tag{2.14}$$

where \otimes is the element-wise multiplication operator and \mathcal{L} is a *nonlinear* operator.

We use a low storage TVD-RK3 method to update the solution field from time-step k to $k + 1$,

$$U^{(1)} = U_k + \Delta t \mathcal{L}U_k \tag{2.15}$$

$$U^{(2)} = \frac{3}{4}U_k + \frac{1}{4}U^{(1)} + \frac{1}{4}\Delta t \mathcal{L}U^{(1)} \tag{2.16}$$

$$U_{k+1} = \frac{1}{3}U_k + \frac{2}{3}U^{(2)} + \frac{2}{3}\Delta t \mathcal{L}U^{(2)} \tag{2.17}$$

with a CFL coefficient of 1. Note that this method only requires two storage units per grid point, which is useful for large scale scientific simulations in higher

⁴From implementation point of view this is a very useful feature of this scheme, because A and B are constant matrices that do not change through training it is possible to pre-compute them using `numpy`'s [86] basic data structures, and then simply import the derivative operators into `TensorFlow`'s custom solver layer using `tf.convert_to_tensor()` command.

dimensions.

- **Training protocol.** For training, we first solve Burgers' equation for M time-steps, then we construct two shifted solution matrices that are separated by a single time-step, *i.e.*,

$$\mathcal{U}^{-1} = \begin{bmatrix} U^1 & U^2 & \dots & U^{M-1} \end{bmatrix} \quad \mathcal{U}^{+1} = \begin{bmatrix} U^2 & U^3 & \dots & U^M \end{bmatrix} \quad (2.18)$$

Basically, one step of TVD-RK3 maps a column of \mathcal{U}^{-1} to its corresponding column in \mathcal{U}^{+1} given an accurate prediction for the hidden parameter. Hence, a semantic BiPDE is trained with \mathcal{U}^{-1} and \mathcal{U}^{+1} as its input and output respectively. The unknown diffusion coefficient is discovered by the code at the bottleneck of the architecture.

- **Numerical setup.** To enable direct comparison with PINNs, we also consider a second parameter γ in Burger's equation. In these current experiments we train for 2 unknown parameters (ν, γ) in the Burger's equation given by

$$u_t + \gamma u u_x - \nu u_{xx} = 0, \quad t \in [0, 1], \quad x \in [-1, 1]$$

Similar to Raissi *et al.* [45] we consider $\nu = 0.01/\pi$ and $\gamma = 1.0$. For completeness we also recall the loss function used in PINN that encodes Burger's equation as a regularization,

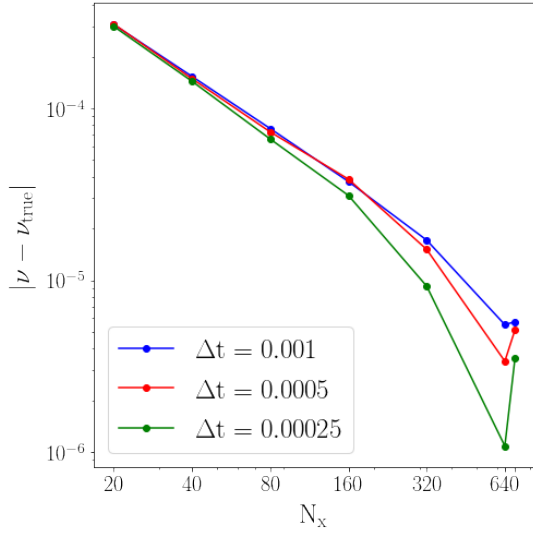
$$MSE = \frac{1}{N} \sum_{i=1}^N \left| u(t_u^i, x_u^i) - u^i \right|^2 + \frac{1}{N} \sum_{i=1}^N \left| u_t(t_u^i, x_u^i) + \gamma u(t_u^i, x_u^i) u_x(t_u^i, x_u^i) - \nu u_{xx}(t_u^i, x_u^i) \right|^2$$

where (t_u^i, x_u^i, u^i) constitute training data with $N = 2000$ observation points in the spatio-temporal domain. In this experiment PINN is composed of 9 layers with

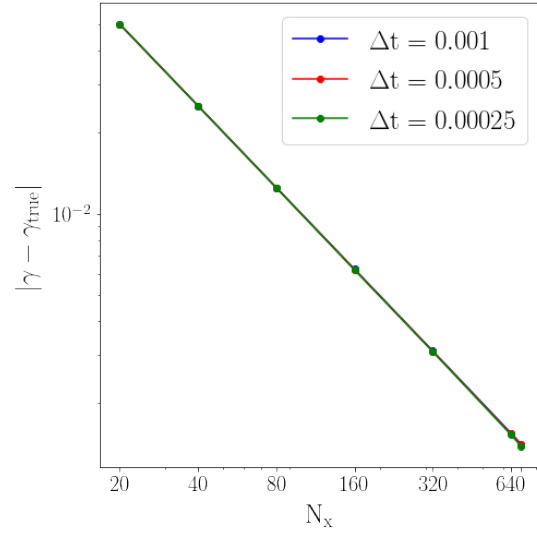
20 neurons per hidden layer. It is worth mentioning that we are reporting BiPDE results by considering solutions in a narrower time span $t \in [0, 0.2]$.

- **Architecture.** Obviously, one can choose a single neuron to represent an unknown parameter ν or γ and in a few iterations an approximate value can be achieved. However, our goal is to train a general purpose encoder that is capable of predicting the unknown value from an input solution pair with arbitrary values of ν and γ and without training on new observations (*cf.* see part 2.5.1). Therefore, we consider a BiPDE that is composed of a `conv1D` layer with 128 filters and a kernel size of 10 with `tanh` activation function, followed by `AveragePooling1D` with a pool size of 2 that after flattening is fed to two `Dense` layers with 20 and 2 neurons respectively that apply `Sigmoid` activation function. We used the `Adam` optimizer to minimize the mean absolute error measure for the loss function.
- **Accuracy test.** First, we train for two unknowns in Burger’s equation, namely ν and γ . We perform a sensitivity analysis for 200 epochs with different numbers of spatial grid points, as well as different time-steps. In each case, we measure the error between the learned values of ν and γ with their true value $\nu_{\text{true}} = 0.01/\pi$ and $\gamma_{\text{true}} = 1.0$. Convergence results of this experiment are tabulated in table 2.5 and shown in figure 2.10. We find that increasing the number of grid points (*i.e.* the resolution) improves the accuracy up to almost 700 grid points before accuracy in ν (but not γ) starts to deteriorate. Note the decrease in time-step size does not have a significant effect on accuracy unless large number of grid points $N_x > 160$ are considered where decreasing time-step clearly improves results.

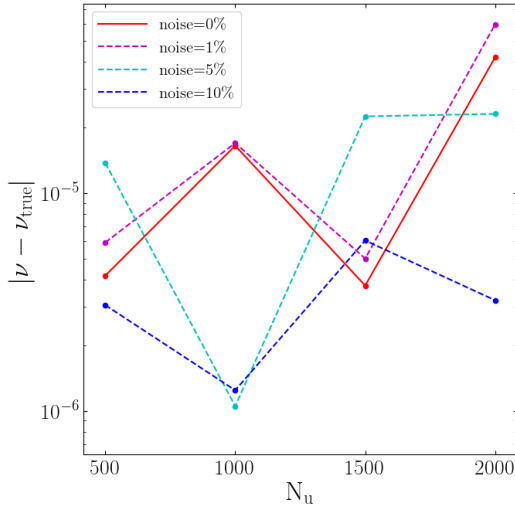
For comparison purposes we report numerical results from table 1 of Raissi *et al.* (2017) [45] in our figures 2.10(c)–2.10(d). Here we only presented noise-less results of BiPDE, therefore only the 0% added noise case of PINN is comparable, *i.e.*



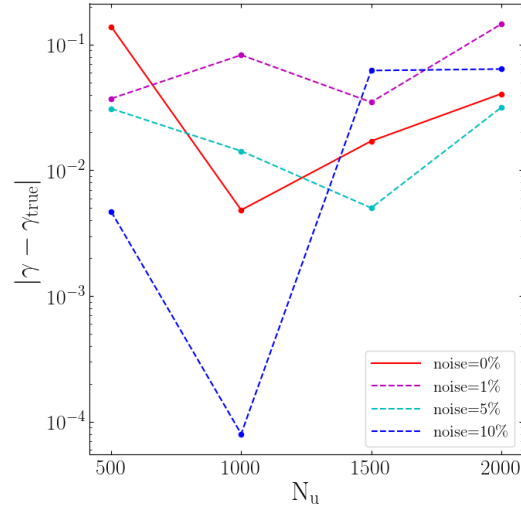
(a) Error in ν - BiPDE with finite difference method.



(b) Error in γ - BiPDE with finite difference method.



(c) Error in ν - PINN.



(d) Error in γ - PINN.

Figure 2.10: Sensitivity analysis in training both parameters γ and ν with BiPDE (a,b), also results from table 1 of Raissi *et al.* (2017) [45] are shown for comparison (c,d) - note only the solid red line may be compared to BiPDE results where no noise is considered on the solution data. True values are $\nu_{\text{true}} = 0.01/\pi$ and $\gamma = 1.0$. In figure (a) the data points at the right end of N_x axis correspond to $N_x = 700$ grid points where the accuracy in the discovered ν value deteriorates.

# epochs = 200	$\Delta t = 0.001$		$\Delta t = 0.0005$		$\Delta t = 0.00025$	
grid size	ν	γ	ν	γ	ν	γ
$N_x = 20$	0.0028751	0.9500087	0.0028731	0.9500685	0.0028828	0.9499334
$N_x = 40$	0.0030294	0.9750050	0.0030341	0.9750042	0.0030391	0.9750047
$N_x = 80$	0.0031067	0.9875077	0.0031101	0.9875285	0.0031167	0.9875455
$N_x = 160$	0.0031455	0.9937580	0.0031443	0.9937674	0.0031519	0.9937985
$N_x = 320$	0.0031659	0.9968843	0.0031679	0.9968919	0.0031738	0.9969027
$N_x = 640$	0.0031775	0.9984500	0.0031797	0.9984597	0.0031841	0.9984711
$N_x = 700$	0.0031773	0.9985866	0.0031779	0.9985945	0.0031865	0.9986123

Table 2.5: Discovering two unknown values of ν and γ in Burger’s equation. These values are plotted in figure 2.10.

$\langle \hat{\nu} \rangle$	$\Delta t = 0.001$	$\Delta t = 0.0005$	$\Delta t = 0.00025$
$N_x = 20$	0.0064739	0.0065189	0.0065514
$N_x = 40$	0.0048452	0.0048200	0.0047086
$N_x = 80$	0.0040260	0.0040324	0.0039963
$N_x = 160$	0.0036042	0.0036011	0.0036310
$N_x = 320$	0.0033958	0.0034144	0.0033827
$N_x = 640$	0.0032919	0.0032895	0.0032916
$N_x = 700$	0.0032829	0.0032816	0.0032906

Table 2.6: Discovering one unknown parameter in Burger’s equation, values for ν .

the solid red line in figures 2.10(c)–2.10(d). Even though the two test cases have significant differences and much care should be taken to directly compare the two methods, however BiPDEs have a clear advantage by exhibiting *convergence* in the unknown values, *i.e.* more data means better results.

In a second experiment, we fix the value of $\gamma = 1.0$ and only train for the unknown diffusion coefficient ν . Similar to previous test we trained the network for 200 epochs and figure 2.11 shows the error in the discovered value of ν at different levels of resolution. In this case decreasing time-step size does not seem to have a significant effect on accuracy. A curious observation is the absolute value of error for ν is two orders of magnitude more precise when the network is trained for both parameters ν and γ than when only tuning for ν . Again, convergence in unknown

parameter is retained in this experiment.

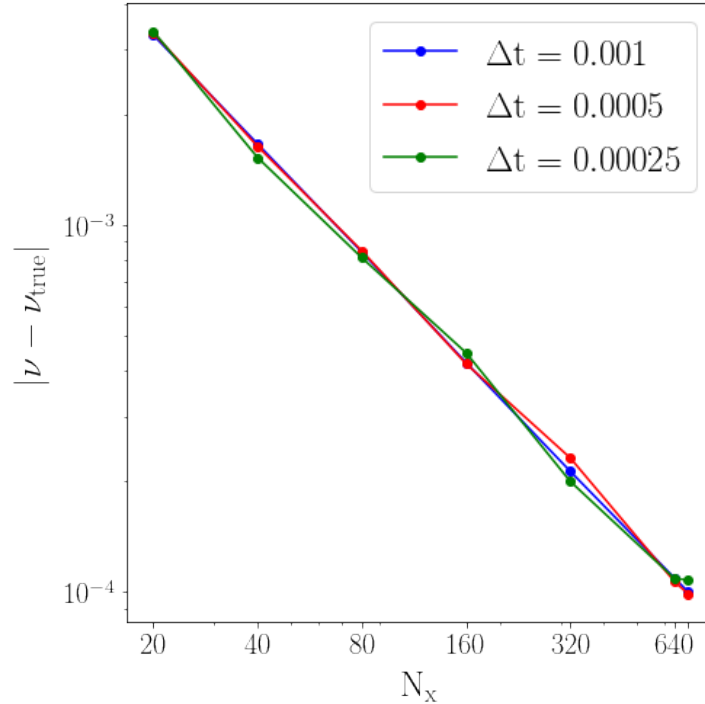


Figure 2.11: Sensitivity analysis in training only one parameter ν . True value of $\nu_{\text{true}} = 0.01/\pi$ is sought in Burgers' equation at different levels of resolution. Right-most data points correspond to $N_x = 700$ grid points.

2.5 Mesh-less BiPDE: Multi-Quadratic Radial Basis Functions

Not only are direct computations of partial derivatives from noisy data extremely challenging, in many real world applications, measurements can only be made on scattered point clouds. Tikhonov regularization type approaches have been devised to avoid difficulties arising from high sensitivity of differencing operations on noisy data [87, 88, 89]; for neural network based approaches, see [90, 91]. Recently, Trask *et al.* [92] have proposed an efficient framework for learning from unstructured data that is based on the General-

ized Moving Least Squares (GMLS) technique. They show performance of GMLS-Nets to identify differential operators and to regress quantities of interest from unstructured data fields. Another interesting approach had been put forth in the late 80s by [93, 94] that designed neural networks based on Radial Basis Functions (RBF) to perform functional interpolation tasks. In these networks, the activation function is defined as the radial basis function of interest and the training aims to discover the weights of this network, which interestingly coincide with the coefficients in the corresponding radial basis expansion.

Since the early 70s, RBFs have been used for highly accurate interpolation from scattered data. In particular, Hardy [95] introduced a special kind of RBF called the *multiquadratic* series expansion, that provides superior performance in terms of accuracy, stability, efficiency, simplicity and memory usage [96]. Kansa (1990) [97, 98] pioneered the use of radial basis functions to solve time dependent partial differential equations through deriving a modified multi-quadratic scheme. In 1998, Hon and Mao [99] applied multiquadratics as a spatial discretization method for the nonlinear Burgers' equation and solved it for a wide range of Reynolds numbers (from 0.1 to 10,000). Their scheme was later enhanced to second-order accuracy in time by Xie and Li (2013) [100] via introducing a compact second-order accurate time discretization. Interestingly, the accuracy of these mesh-free methods can be improved by fine-tuning distributions of collocation points or their *shape parameters*. For example, Hon and Mao devised an adaptive point to chase the peak of shock waves, which improved their results. Fortunately, such fine-tuning of parameters can be automated using BiPDE networks; we demonstrate this in this section.

- **Discretization.** We chose to blend the second-order accurate method of Xie and Li, briefly described next and we leave further details to their original paper [100]. Initially, one can represent a distribution $u(\mathbf{x})$ in terms of a linear combination of

radial basis functions,

$$u(\mathbf{x}) \approx \sum_{j=0}^{N_s} \lambda_j \phi_j(\mathbf{x}) + \psi(\mathbf{x}), \quad \mathbf{x} \in \Omega \subset \mathcal{R}^{dim}, \quad (2.19)$$

where $\phi(\mathbf{x})$ is the radial basis function that we adopt,

$$\phi_j(\mathbf{x}) = \sqrt{r_j^2 + c_j^2}, \quad r_j^2 = \|\mathbf{x} - \mathbf{x}_j\|_2^2, \quad (2.20)$$

and c_j is the *shape parameter* that has been experimentally shown to follow $c_j = Mj + b$ with $j = 0, 1, \dots, N_s$ (N_s is number of seed points). Moreover M and b are tuning parameters. In equation (2.19), $\psi(\mathbf{x})$ is a polynomial to ensure solvability of the resulting system when ϕ_j is only conditionally positive definite. To solve PDEs, one only needs to represent the solution field with an appropriate form of equation (2.19). In the case of Burgers' equation the solution at any time-step n can be represented by

$$u^n(x) \approx \sum_{j=0}^{N_s} \lambda_j^n \phi_j(x) + \lambda_{N_s+1}^n x + \lambda_{N_s+2}^n \quad (2.21)$$

over a set of reference points for the basis functions that are given by $x_j = j/N_s$, $j = 0, 1, \dots, N_s$. Xie and Li derived the following compact second-order accurate system of equations

$$\left[1 + \frac{\Delta t}{2} u_x^n(\hat{x}_j)\right] u^{n+1}(\hat{x}_j) + \frac{\Delta t}{2} u^n(\hat{x}_j) u_x^{n+1}(\hat{x}_j) - \frac{\nu \Delta t}{2} u_{xx}^{n+1}(\hat{x}_j) = u^n(\hat{x}_j) + \frac{\nu \Delta t}{2} u_{xx}^n(\hat{x}_j) \quad (2.22)$$

over a set of $N_d + 1$ distinct collocation points $\hat{x}_j = (1 + j)/(N_d + 2)$ with $j = 0, 1, \dots, N_d$. Two more equations are obtained by considering the left and right

boundary conditions $u^{n+1}(x_L) = u^{n+1}(x_R) = 0$. Note that spatial derivatives are directly computed by applying derivative operator over equation (2.21). At every time-step, one solves for the $N + 3$ coefficients $\lambda_0^n, \dots, \lambda_{N+2}^n$, while the spatial components of the equations remain intact (as long as points are not moving). The solution is obtained over the initial conditions given by $u^0(\hat{x}_j)$.

For implementation purposes, we represent the system of equations (2.22) in a matrix notation that is suitable for tensorial operations in `TensorFlow`. To this end, we first write equation (2.21) as

$$U^n(\hat{x}) = A(\hat{x})\Lambda^n, \quad (2.23)$$

where

$$U_{(N_d+1) \times 1}^n = \begin{bmatrix} u^n(\hat{x}_0) \\ u^n(\hat{x}_1) \\ \vdots \\ u^n(\hat{x}_{N_d}) \end{bmatrix}, \quad \Lambda_{(N_s+1) \times 1}^n = \begin{bmatrix} \lambda_0^n \\ \lambda_1^n \\ \vdots \\ \lambda_{N_s}^n \end{bmatrix}, \quad (2.24)$$

$$\left[A_{ij}(\hat{\mathbf{x}}) \right]_{(N_d+1) \times (N_s+1)} = \left[\phi_j(\hat{x}_i) - \phi_j(x_L) - \frac{\phi_j(x_R) - \phi_j(x_L)}{x_R - x_L} (\hat{x}_i - x_L) \right], \quad (2.25)$$

with $i = 0, 1, \dots, N_d$ and $j = 0, 1, \dots, N_s$. Note that we already injected the homogeneous boundary conditions into equation (2.23). Therefore, equation (2.22) can be written as,

$$\left[A + (\mathbf{g}_x \mathbf{1}^T) \otimes A + (\mathbf{g} \mathbf{1}^T) \otimes A_x - \frac{\nu \Delta t}{2} A_{xx} \right] \Lambda^{n+1} = \left[A + \frac{\nu \Delta t}{2} A_{xx} \right] \Lambda^n, \quad (2.26)$$

where $\mathbf{1}^T = [1, 1, \dots, 1]_{1 \times (N_s+1)}$, \otimes is component-wise multiplication, and

$$\mathbf{g} = \frac{\Delta t}{2} A \Lambda^n, \quad \mathbf{g}_x = \frac{\Delta t}{2} A_x \Lambda^n, \quad (2.27)$$

$$(A_x)_{ij} = \phi_j'(\hat{x}_i) - \frac{\phi_j(x_R) - \phi_j(x_L)}{x_R - x_L}, \quad (A_{xx})_{ij} = \phi_j''(\hat{x}_i). \quad (2.28)$$

Note that in case of training for two parameters (ν, γ) , expression for \mathbf{g} in equation 2.26 needs to be modified by letting $\mathbf{g} = \frac{\gamma \Delta t}{2} A \Lambda^n$.

- Architecture.** Note that both the collocation points and the interpolation seed points can be any random set of points within the domain and not necessarily a uniform set of points as we chose above. In fact, *during training we allow BiPDE to find a suitable set of interpolation points as well as the shape parameters on its own.* The input data is calculated using aforementioned finite difference method over uniform grids and later interpolated on a random point cloud to produce another sample of solutions on unstructured grids for training. Thus, in our architecture the last layer of the encoder has $2N_s + 1$ neurons with `sigmoid` activation functions representing the $2N_s$ shape parameters and seed points, as well as another neuron for the unknown diffusion coefficient. Note that for points to the left of origin, in the range $x \in [-1, 0]$, we simply multiplied the output of N_s activation functions by “ -1 ” within the solver layer (because output of `Sigmoid` function is always positive). We use the mean squared error between data and predicted solution at time-step $n + p$ as the loss function. We used the `Adam` optimizer to minimize the loss function.
- Training protocol.** As in the previous case, we apply successive steps of MQ-RBF scheme to march the input data forward to a future time-step. Not surprisingly, we observed that taking higher number of steps improves the results because erroneous

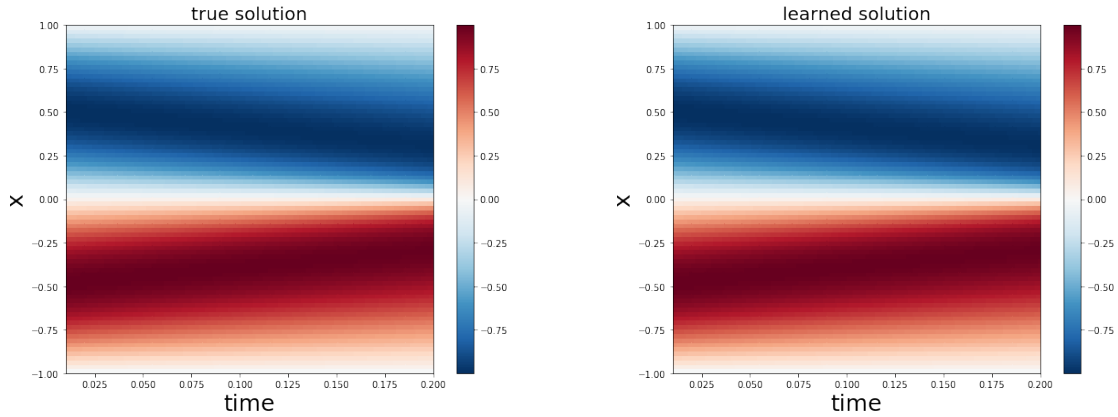
guess of the diffusion coefficient leads to more pronounced discrepancy after longer periods of time. Hence, we map the data \mathcal{U}^{-p} to \mathcal{U}^{+p} , which is p time-steps shifted in time,

$$\mathcal{U}^{-p} = \left[U^1, U^2, \dots, U^{M-p} \right] \quad \mathcal{U}^{+p} = \left[U^{1+p}, U^{2+p}, \dots, U^M \right] \quad (2.29)$$

In our experiments a value of $p = 10$ was sufficient to get satisfactory results at the absence of noise. However, at the presence of Gaussian noise and for smaller values of the diffusion coefficient (such as for $\nu_{\text{true}} = 0.01/\pi$) we had to increase the shift parameter to $p = 100$.

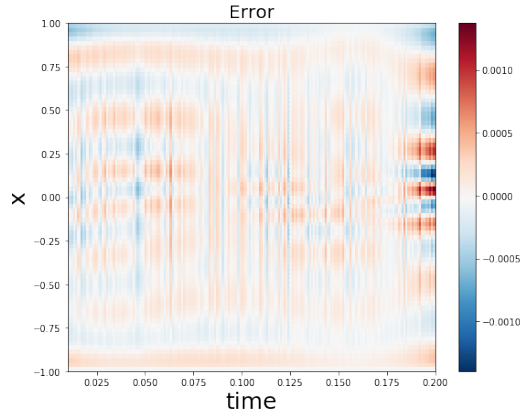
- **Numerical setup.** Once again, we let $\nu_{\text{true}} = 0.01/\pi \approx 0.00318$ and integrate Burgers' equation up to $t_f = 0.2$ with a fixed time-step of $\Delta t = 0.001$. We use the finite difference method of the previous section to generate the datasets. We then interpolate the solution on 80 data points, uniformly distributed in the range $(-1, 1)$ with 20 interpolation seed points. For this experiment, we set the batch size to 1. We trained the network using Adam optimizer. The results after 50 epochs are given in figure 2.12.

Interestingly, for every pair of input-output, the network discovers a distinct value for the diffusion coefficient that provides a measure of uncertainty for the unknown value. We report the average value of all diffusion coefficients as well as the probability density of these values. We observe that for all pairs of solutions, the predicted value for the diffusion coefficient is distributed in the range $0.00305 \leq \hat{\nu} \leq 0.00340$ with an average value of $\langle \hat{\nu} \rangle = 0.00320$, which is in great agreement with the true value, indeed with 0.6% relative error. Interestingly, we observe that the BiPDE network has learned to concentrate its interpolation seed points around the origin

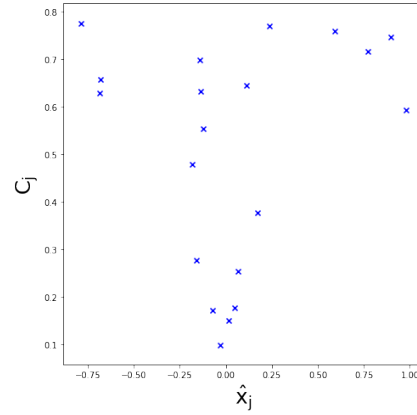


(a) True solution generated by finite differences (input data).

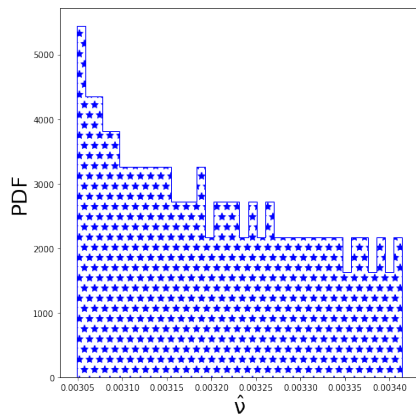
(b) Learned solution generated by MQ-RBF BiPDE (output data).



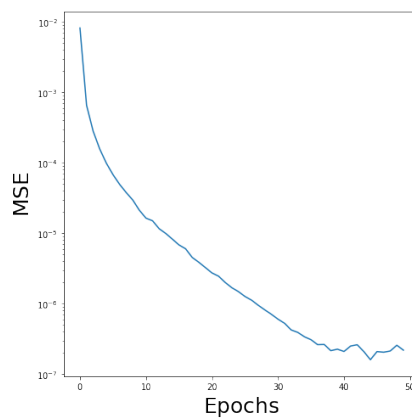
(c) Error in solution.



(d) Discovered seeds and shape parameters.



(e) Distribution of diffusion coefficients.



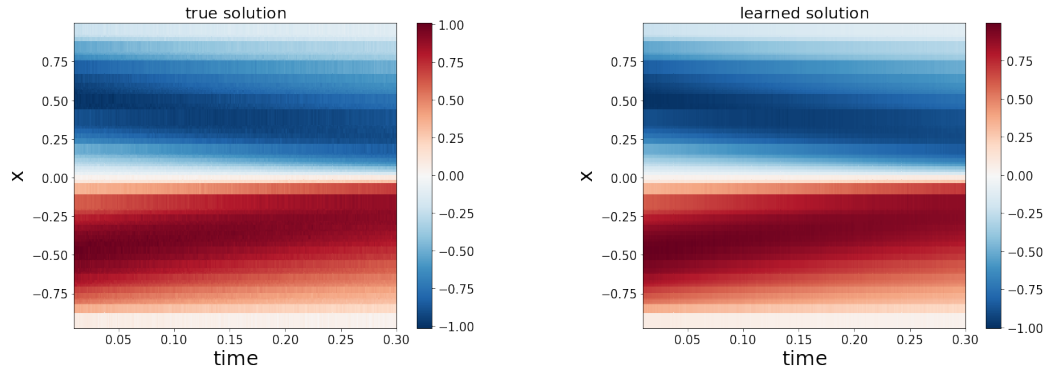
(f) Evolution of mean squared error during training.

Figure 2.12: Results of applying the RBF-BiPDE to Burgers' equation with a true diffusion coefficient of $\nu_{\text{true}} = 0.003183$. The average value of the predicted diffusion coefficients is $\hat{\nu} = 0.00320$.

where the solution field varies more rapidly. Furthermore, around $x = \pm 0.5$, the interpolation points are more sparse, which is in agreement with the smooth behavior of the solution field at these coordinates. Therefore, this network may be used as an automatic shock tracing method to improve numerical solutions of hyperbolic problems with shocks and discontinuities as was shown by Hon and Mao.

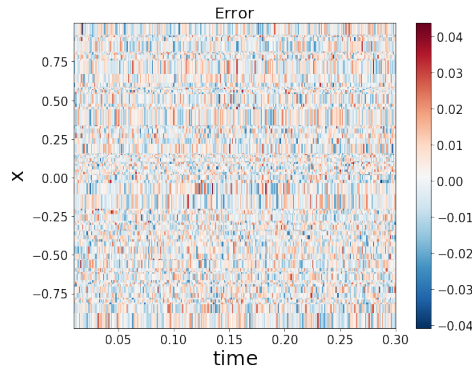
- **Resilience to noise on unstructured grids.** We consider several cases to assess robustness to noise. In each case, we pick 80 *randomly* distributed points along the x -axis and linearly interpolate the solution field on this set of points. Then, we add a Gaussian noise with a given standard deviation. This noisy and unstructured data field is then fed into the MQ-RBF based BiPDE of this section. We use a batch size of 10, with 10% of each sample for validation during training. A summary of our results follows:

1. Let $\nu_{\text{true}} = 0.1/\pi$, $p = 10$, $N_d = 80$, $N_s = 20$, $\Delta t = 0.001$, and consider a Gaussian noise with a standard deviation of 1%. After 100 epochs, we obtain the results in figure 2.13.
2. Let $\nu_{\text{true}} = 0.1/\pi$, $p = 100$, $N_d = 200$, $N_s = 20$, $\Delta t = 0.001$, and consider a Gaussian noise with a standard deviation of 5%. After 150 epochs, we obtain the results in figure 2.14.
3. Let $\nu_{\text{true}} = 0.01/\pi$, $p = 100$, $N_d = 80$, $N_s = 20$, $\Delta t = 0.001$, and consider a Gaussian noise with a standard deviation of 1%. After 200 epochs, we obtain the results in figure 2.15.
4. Let $\nu_{\text{true}} = 0.01/\pi$, $p = 100$, $N_d = 200$, $N_s = 20$, $\Delta t = 0.001$, and consider a Gaussian noise with a standard deviation of 5%. After 150 epochs, we obtain the results in figure 2.16.

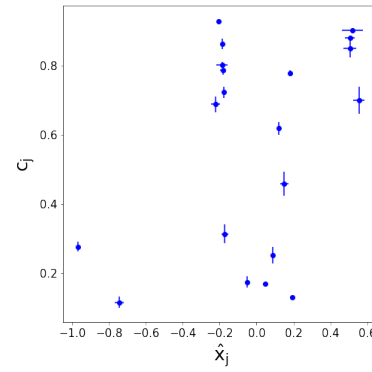


(a) True solution generated by finite differences and with added noise. Solution is interpolated on a random grid.

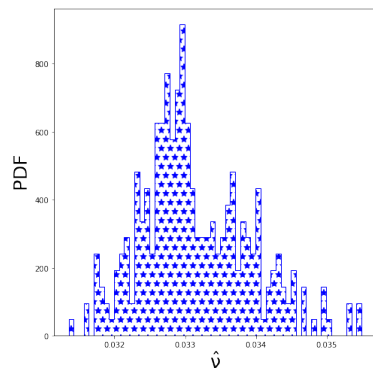
(b) Learned solution generated by MQ-RBF BiPDE (output data).



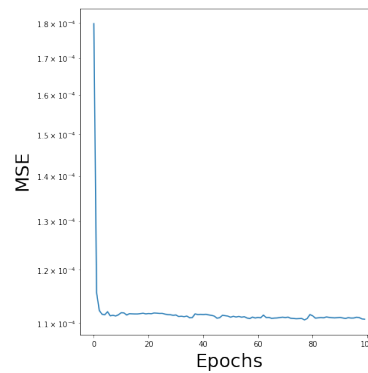
(c) Error in solution.



(d) Discovered seeds and shape parameters. Error bars indicate one standard deviation.

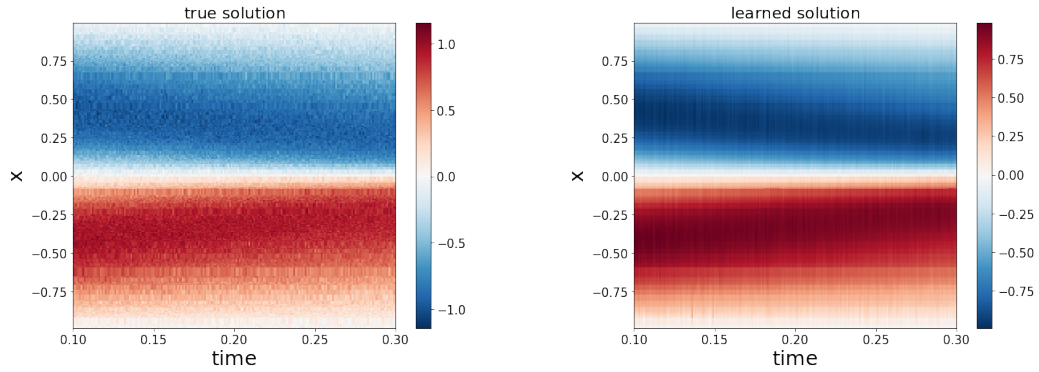


(e) Probability density of diffusion coefficients.



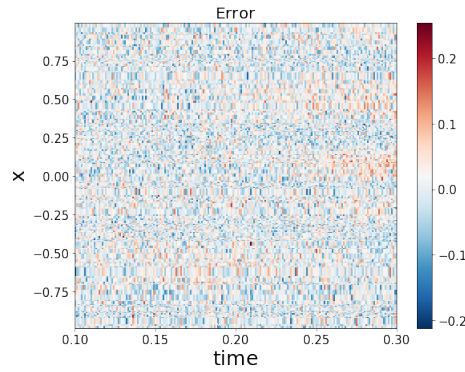
(f) Evolution of mean squared error versus number of epochs.

Figure 2.13: Results of applying the RBF-BiPDE to Burgers' equation with a true diffusion coefficient of $\nu_{\text{true}} = 0.03183$. The average value of the predicted diffusion coefficients is $\hat{\nu} = 0.0331$. The data is provided on a scattered point cloud with added Gaussian noise with 1% standard deviation.

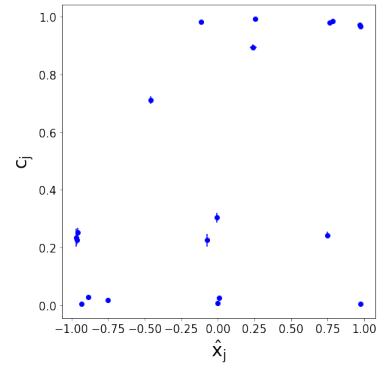


(a) True solution generated by finite differences and with added noise. Solution is interpolated on a random grid.

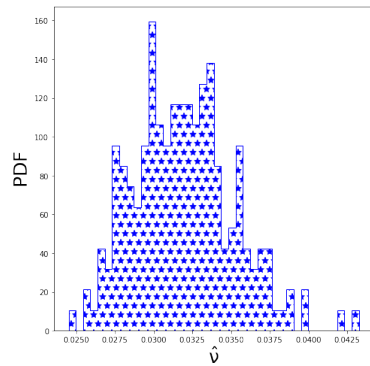
(b) Learned solution generated by MQ-RBF BiPDE (output data).



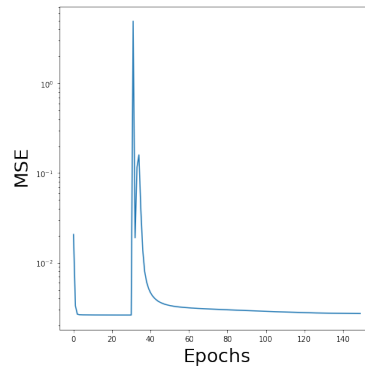
(c) Error in solution.



(d) Discovered seeds and shape parameters. Error bars indicate one standard deviation.

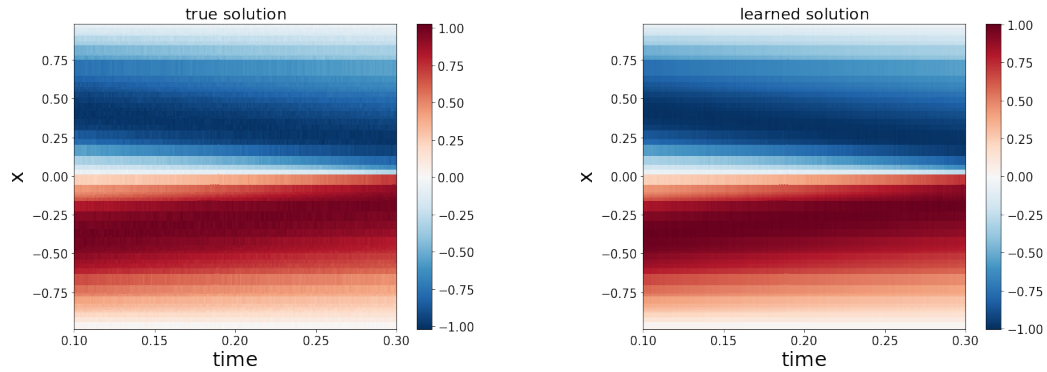


(e) Probability density of diffusion coefficients.



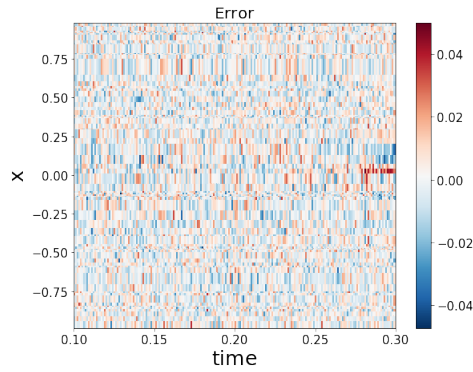
(f) Evolution of mean squared error versus number of epochs.

Figure 2.14: Results of applying the RBF-BiPDE to Burgers' equation with a true diffusion coefficient of $\nu_{\text{true}} = 0.03183$. The average value of the predicted diffusion coefficients is $\hat{\nu} = 0.03160$. The data is provided on a scattered point cloud with added Gaussian noise with 5% standard deviation.

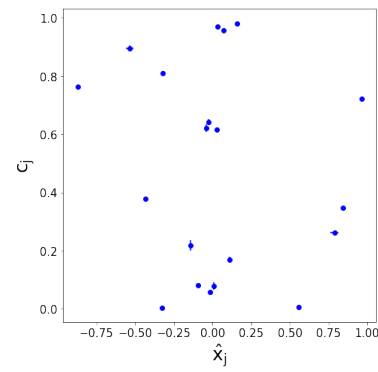


(a) True solution generated by finite differences and with added noise. Solution is interpolated on a random grid.

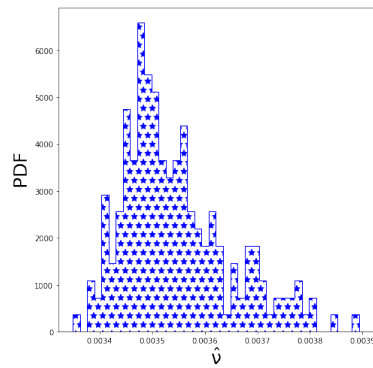
(b) Learned solution generated by MQ-RBF BiPDE (output data).



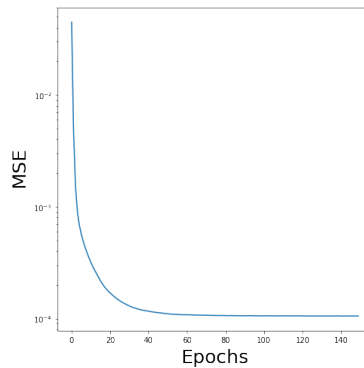
(c) Error in solution.



(d) Discovered seeds and shape parameters. Error bars indicate one standard deviation.

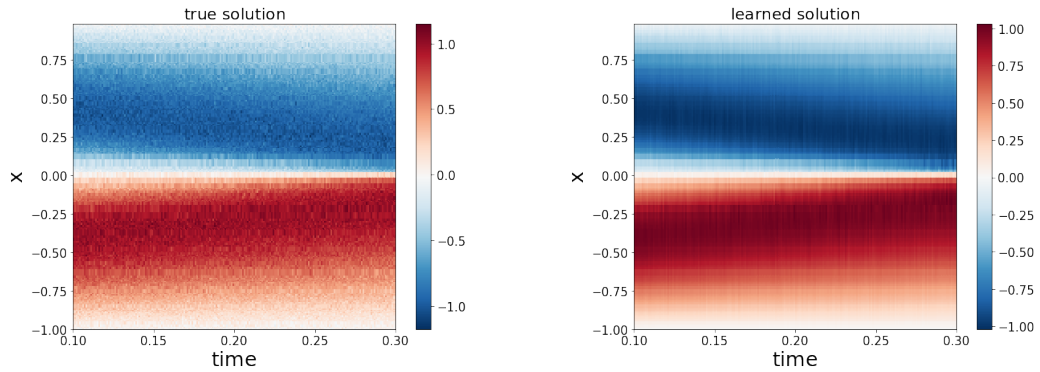


(e) Probability density of diffusion coefficients.



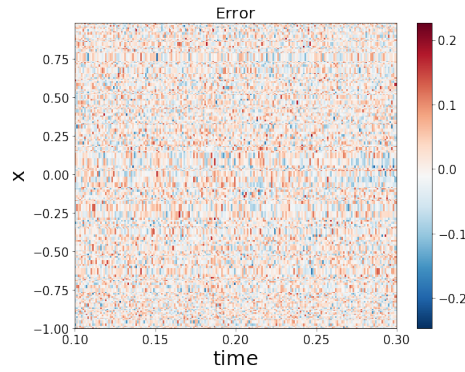
(f) Probability density of diffusion coefficients.

Figure 2.15: Results of applying the RBF-BiPDE to Burgers' equation with a true diffusion coefficient of $\nu_{\text{true}} = 0.003183$. The average value of the predicted diffusion coefficients is $\hat{\nu} = 0.00352$. The data is provided on a scattered point cloud with added Gaussian noise with 1% standard deviation.

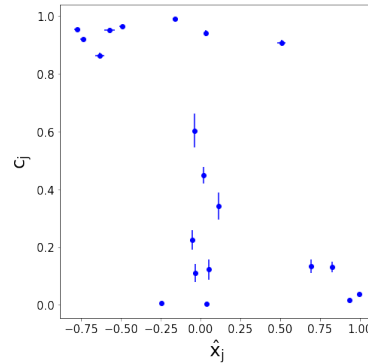


(a) True solution generated by finite differences and with added noise. Solution is interpolated on a random grid.

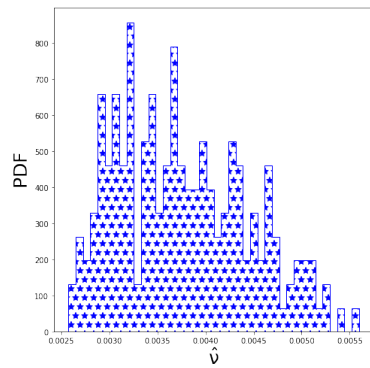
(b) Learned solution generated by MQ-RBF BiPDE (output data).



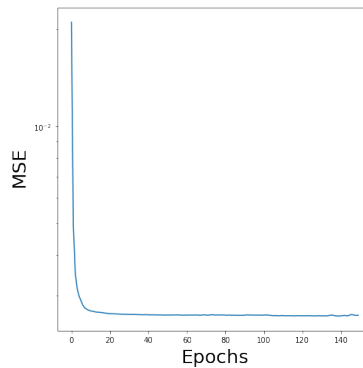
(c) Error in solution.



(d) Discovered seeds and shape parameters. Error bars indicate one standard deviation.



(e) Probability density of diffusion coefficients.



(f) Probability density of diffusion coefficients.

Figure 2.16: Results of applying the RBF-BiPDE to Burgers' equation with a true diffusion coefficient of $\nu_{\text{true}} = 0.003183$. The average value of the predicted diffusion coefficients is $\hat{\nu} = 0.003677$. The data is provided on a scattered point cloud with added Gaussian noise with 5% standard deviation.

We observe that this architecture is generally robust to noise. However, at higher noise values require more tuning of hyperparameters, as well as longer training.

- **Accuracy tests.** We report the values of the discovered diffusion coefficients in the Burgers' equation for different grid sizes and different time-steps. We use the same setting as that detailed in the numerical setup part in this section. Particularly, the interpolation seeds are determined by the network and the training data is on a uniformly distributed set of points computed by the finite difference method of the previous section. We consider three different time-steps, $\Delta t = 0.001, 0.0005, 0.00025$, and two diffusion coefficients of $\nu_{\text{true}} = 0.01/\pi, 0.1/\pi$ over grids of size $N_x = 80, 160$. At each time-step, for all experiments with different grid sizes, we stop the training when the mean squared error in the solution field converges to a constant value and does not improve by more epochs; this roughly corresponds to 50, 25, 12 epochs for each of the training time-steps, respectively. This indicates that by choosing smaller time steps less number of epochs are needed to obtain the same level of accuracy in the unknown parameter. Furthermore, we use an **Adam** optimizer with a learning rate of 0.001.

The results of the accuracy tests are tabulated in tables 2.7–2.8. We observe, for all experiments, that the discovered coefficient is in great agreement with the true values. Due to adaptivity of the interpolation seed points and their shape parameters for different experiments, the observed error values do not seem to follow the trend of traditional finite difference methods, as depicted in previous sections. This could also be due to lower order of accuracy of the MQ-RBF method, *i.e.* being a second-order accurate method, compared to the higher-order accurate finite difference method used in the previous section.

$\langle \hat{\nu} \rangle$	$\Delta t = 0.001$	$\Delta t = 0.0005$	$\Delta t = 0.00025$
# <i>epochs</i>	50	25	12
$N_x = 80$	$0.03173 \pm 3.4 \times 10^{-4}$	$0.03196 \pm 4.2 \times 10^{-4}$	$0.03188 \pm 2.8 \times 10^{-4}$
$N_x = 160$	$0.03186 \pm 5.8 \times 10^{-5}$	$0.03191 \pm 3.6 \times 10^{-4}$	$0.03137 \pm 1.2 \times 10^{-4}$

Table 2.7: Discovered values of the diffusion coefficient for $\nu_{\text{true}} = 0.03183$ at different time-steps and grid sizes.

$\langle \hat{\nu} \rangle$	$\Delta t = 0.001$	$\Delta t = 0.0005$	$\Delta t = 0.00025$
# <i>epochs</i>	50	25	12
$N_x = 80$	$0.003326 \pm 5.1 \times 10^{-5}$	$0.003162 \pm 2.2 \times 10^{-4}$	$0.003155 \pm 1.2 \times 10^{-4}$
$N_x = 160$	$0.003264 \pm 1.0 \times 10^{-4}$	$0.003151 \pm 1.3 \times 10^{-4}$	$0.003192 \pm 1.2 \times 10^{-4}$

Table 2.8: Discovered values of the diffusion coefficient for $\nu_{\text{true}} = 0.003183$ obtained with different time-steps and grid sizes.

2.5.1 Learning the inverse transform

As we emphasized before, a feature of BiPDE is to produce self-supervised pre-trained encoder models for inverse differential problems that are applicable in numerous applications where hidden values should be estimated in real-time. We train an encoder over a range of values $\nu \in [0.1/\pi, 1/\pi]$ and assess the performance of the trained model on new data with arbitrarily chosen ν values. We choose 50 diffusion coefficients that are distributed uniformly in this range, then integrate the corresponding Burgers' equation up to $t_f = 0.2$ with a constant time-step of $\Delta t = 0.0005$ on a grid with $N_x = 100$ grid points using the aforementioned finite difference method. There are 4000 time-steps in each of the 50 different realizations of Burgers' equation. For a fixed value of $p = 20$, we draw 10 solution pairs for each value of ν at uniformly distributed time instances and discard the first two instances to improve convergence of the network. Hence, the training data uniformly samples the space of solutions over a 8×50 grid of (t, ν) , as illustrated in figure 2.17. We use the resulting 400 pairs in training of a semantic BiPDE, with 320 pairs used for training and 80 pairs for validation.

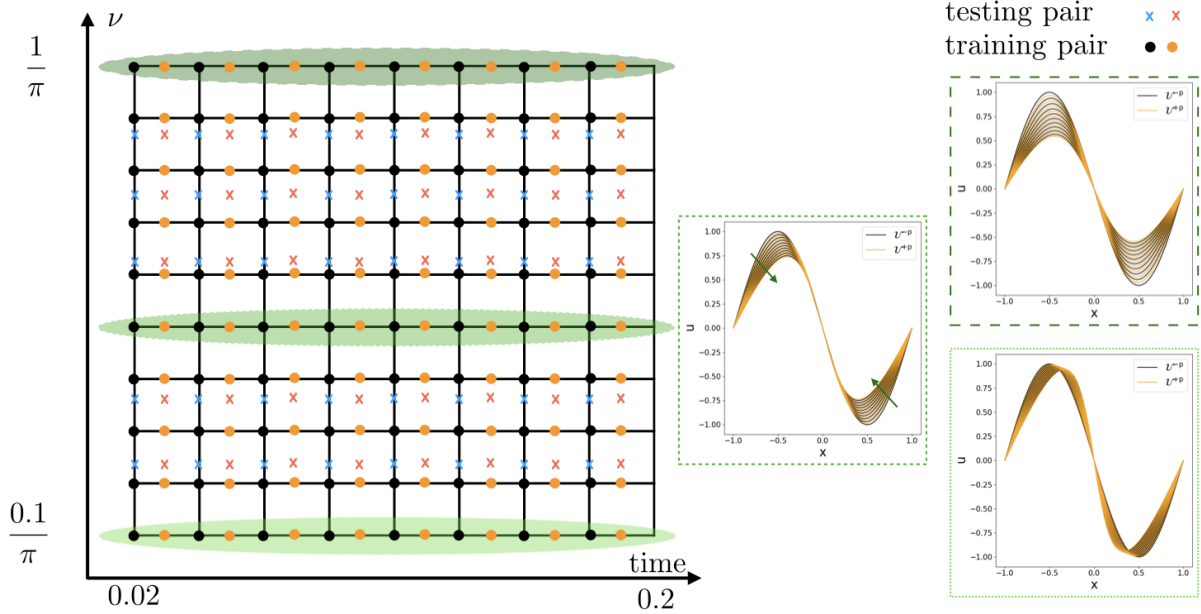


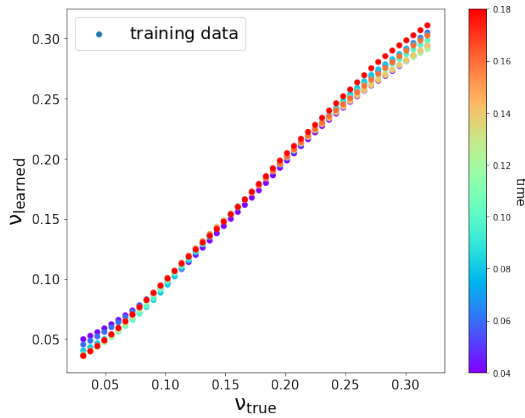
Figure 2.17: Topology of data points for training and testing of the semantic BiPDE. Along the ν dimension, we depict 10 (out of 50) of the selected data points, while along the time dimension we illustrate the actual 8 data points. Training pairs of U^{-p} and U^{+p} are color coded by black and orange dots, respectively; testing pairs are depicted by blue and red crosses. On the right panel, we illustrate the training data for three nominal values of the diffusion coefficient, highlighted by green shades. Green arrows indicate the direction of time.

Architecture. Given an arbitrary input, the signature of the hidden physical parameters will be imprinted on the data in terms of complex patterns spread in space and time. We use a CNN layer as a front end unit to transform the input pixels to internal image representations. The CNN unit has 32 filters with kernel size of 5. The CNN is followed by max pooling with pool size of 2, which is then stacked with another CNN layer of 16 filters and kernel size of 5 along with another max pooling layer. The CNN block is stacked with two dense layers with 100 and 41 neurons, respectively. CNN and dense layers have **ReLU** and **Sigmoid** activation functions, respectively. Overall, there are 42,209 trainable parameters in the network. Conceptually, the CNN extracts features on every snapshot that characterizes the evolution of the solution field through time-steps with a proper physical parameter. This parameter is enforced to be the diffusion coefficient through the PDE solver decoder stage. We train this network for 500 epochs using an **Adam** optimizer.

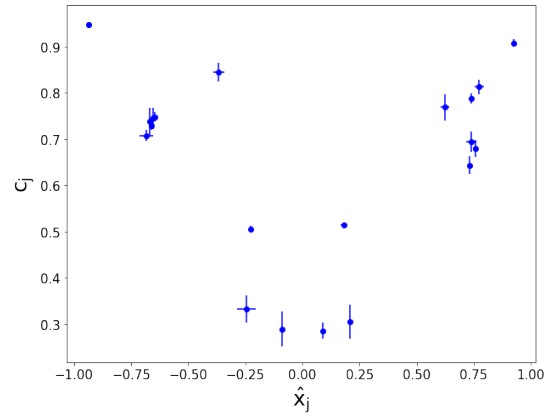
Resilience to noise. Even though the encoder is trained on ideal datasets, we demonstrate a semantic BiPDE still provides accurate results on noisy datasets. In contrast to other methods, we pre-train the network in a self-supervised fashion on clean data and later we apply the trained encoder on unseen noisy data⁵.

In figure 2.18, we provide the performance of this network on training as well as on unseen clean/noisy data-sets. Furthermore, the network determines optimal parameters of the MQ-RBF method by evaluating interpolation seed points as well as their corresponding shape parameters to obtain the best approximation over *all* input data.

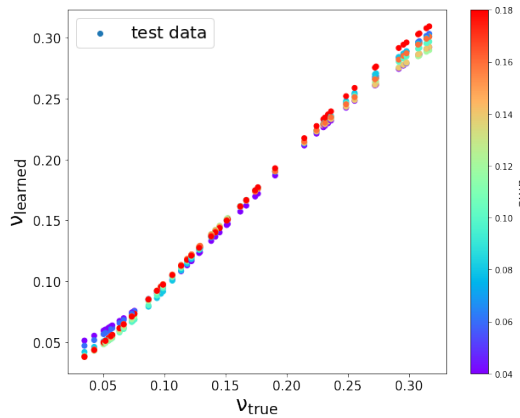
⁵Note that the network could also be trained on noisy data as we showed before; however training would take longer in that case.



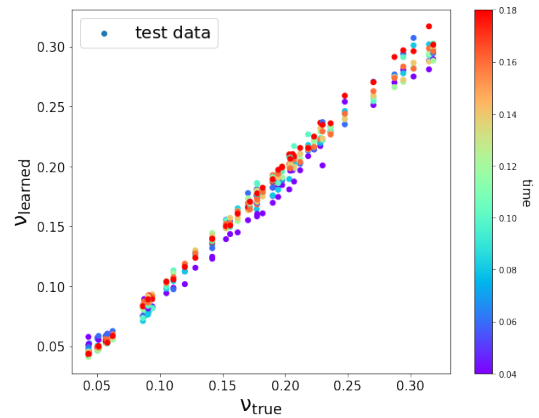
(a) Performance of encoder on training data set.



(b) Distribution of interpolation points and shape parameters discovered by the network.



(c) Performance of the encoder on unseen data.



(d) Performance of the encoder on unseen data with Gaussian noise with standard deviation 0.01.

Figure 2.18: Semantic autoencoder learns how to discover hidden variables from pairs of solutions. These results are obtained after 500 epochs on 50 data points along the ν -axis.

2.6 Conclusion

We introduced BiPDE networks, a natural architecture to infer hidden parameters in partial differential equations given a limited number of observations. We showed that this approach is versatile as it can be easily applied to arbitrary static or nonlinear time-dependent inverse-PDE problems. We showed the performance of this design on multiple inverse Poisson problems in one and two spatial dimensions as well as on the non-linear time-dependent Burgers' equation in one spatial dimension. Moreover, our results indicate BiPDEs are robust to noise and can be adapted for data collected on unstructured grids by resorting to traditional mesh-free numerical methods for solving partial differential equations. We also showed the applicability of this framework to the discovery of inverse transforms for different inverse-PDE problems.

There are many areas of research that could be further investigated, such as considering diffusion maps with discontinuities across subdomains, using more sophisticated neural network architectures for more complex problems, using higher-order numerical solvers and finally tackle more complicated governing PDE problems with a larger number of unknown fields or in higher dimensions.

Acknowledgment

This research was supported by ARO W911NF-16-1-0136 and ONR N00014-17-1-2676.

Chapter 3

JAX-DIPS:

3.1 abstract

We present a scalable strategy for development of mesh-free hybrid neuro-symbolic partial differential equation solvers based on existing mesh-based numerical discretization methods. Particularly, this strategy can be used to efficiently train neural network surrogate models of partial differential equations by (i) leveraging the accuracy and convergence properties of advanced numerical methods, solvers, and preconditioners, as well as (ii) better scalability to higher order PDEs by strictly limiting optimization to first order automatic differentiation. The presented neural bootstrapping method (hereby dubbed NBM) is based on evaluation of the finite discretization residuals of the PDE system obtained on implicit Cartesian cells centered on a set of random collocation points with respect to trainable parameters of the neural network. Importantly, the conservation laws and symmetries present in the bootstrapped finite discretization equations inform the neural network about solution regularities within local neighborhoods of training points. We apply NBM to the important class of elliptic problems with jump conditions across irregular interfaces in three spatial dimensions. We show the method is convergent such

that model accuracy improves by increasing number of collocation points in the domain and preconditioning the residuals. We show NBM is competitive in terms of memory and training speed with other PINN-type frameworks. The algorithms presented here are implemented using JAX in a software package named JAX-DIPS (<https://github.com/JAX-DIPS/JAX-DIPS>), standing for differentiable interfacial PDE solver. We open sourced JAX-DIPS to facilitate research into use of differentiable algorithms for developing hybrid PDE solvers.

3.2 Introduction

3.2.1 Problem statement

Consider a closed irregular interface (Γ) that partitions the computational domain (Ω) into interior (Ω^-) and exterior (Ω^+) subdomains; *i.e.*, $\Omega = \Omega^- \cup \Gamma \cup \Omega^+$. We are interested in the solutions $u^\pm \in \Omega^\pm$ to the following class of linear elliptic problems in $\mathbf{x} \in \Omega^\pm$:

$$\begin{aligned} k^\pm u^\pm - \nabla \cdot (\mu^\pm \nabla u^\pm) &= f^\pm, & \mathbf{x} \in \Omega^\pm \\ [u] &= \alpha, & \mathbf{x} \in \Gamma \\ [\mu \partial_{\mathbf{n}} u] &= \beta, & \mathbf{x} \in \Gamma \end{aligned}$$

Here $f^\pm = f(\mathbf{x} \in \Omega^\pm)$ is the spatially varying source term, $\mu^\pm = \mu(\mathbf{x} \in \Omega^\pm)$ are the diffusion coefficients, and k^\pm are the reaction coefficients in the two domains. We consider Dirichlet boundary conditions in a cubic domain $\Omega = [-L/2, L/2]^3$.

This class of problems arise ubiquitously in describing diffusion dominated processes in physical systems and life sciences where sharp and irregular interfaces regulate trans-

port across regions with different properties. Examples include Poisson-Boltzmann equation for describing electrostatic properties of membranes, colloids and solvated biomolecules with jump in dielectric permittivities [101, 102], electroporation of cell aggregates with nonlinear membrane jump conditions [19], epitaxial growth in fabrication of opto-electronic devices where atomic islands grow by surface diffusion of adatoms across freely moving interfaces [103], solidification of multicomponent alloys used for manufacturing processes with free interfaces separating different phases of matter [104, 105], directed self-assembly of diblock copolymers for next generation lithography [106, 107, 108], multiphase flows with and without phase change, and Poisson-Nernst-Planck equations for electrokinetics. Much of these processes are multiscale and the changes across interfaces must be mathematically modeled and numerically solved as sharp surfaces. Smoothing strategies introduce unphysical characteristics in the solution and lead to systemic errors.

3.2.2 Literature on relevant finite discretization methods

Several numerical methods have been proposed for accurate solution of this class of problems based on explicit or implicit representation of the interface. Finite element methods rely on explicit meshing of the surface that poses severe challenges [109, 110]. Implicit methods include the Immersed Interface Method (IIM) [111] and its variants [112, 113, 114, 115] that rely on Taylor expansions of the solution on both sides of the interface and modifying the local stencils to impose the jump conditions. The main challenge is evaluating high order jump conditions and surface derivatives along interface. Another method is the Ghost Fluid Method (GFM) [116] that was originally introduced to approximate two-phase compressible flows and later applied to the Poisson problem with jump conditions [117]. The basic idea is to define fictitious fluid regions across the discontinuities by adding jump conditions to the true fluid. While GFM captures the

normal jump in solution accurately, the tangential jump is smeared. This was solved by the Voronoi Interface Method (VIM) [118] by applying the GFM treatment on a local Voronoi mesh by adapting a local Cartesian mesh which introduces numerical challenges. Several other approaches include the cut-cell method [119], discontinuous Galerkin and eXtended Finite Element Method (XFEM) [120, 121, 122] among others.

In this work we bootstrap the level-set based finite volume method on Cartesian grids proposed by Bochkov & Gibou (2020) [123]. This method is based on the idea of Taylor expansions in the normal direction and employing one-sided least-square interpolations for imposing jump conditions. In particular, this method offers second order accurate numerical solutions with first order accurate gradients in the L^∞ -norm.

3.2.3 Literature on solving PDEs with neural networks

Since early 1990s, artificial neural networks have been used for solving partial differential equations by (i) mapping the algebraic operations of the discretized PDE systems onto specialized neural network architectures and minimizing the network energy, or (ii) treating the whole neural network as the basic approximation unit whose parameters are adjusted to minimize a specialized error function that includes the differential equation itself with its boundary/initial conditions.

In the first category, neurons output the discretized solution values over a set number of grid points and minimizing the network energy drives the neuronal values towards the solution of the linear system at the mesh points. In this case, the neural network energy is the residual of the finite discretization method summed over all neurons of the network [124]. Although the convergence properties of the finite discretization methods guarantee and control quality of the obtained solutions, the computational costs grow by increasing resolution and dimensionality. Interestingly, due to regular and sparse

structure of the finite discretizations, such locally connected neural network PDE solvers have been implemented on VLSI analog CMOS circuits [125, 126, 127].

The second strategy proposed by Lagaris *et al.* [41] relies on the function approximation capabilities of the neural networks. Encoding the solution everywhere in the domain within a neural network offers a mesh-free, compact, and memory efficient surrogate model for the solution function that can be utilized in subsequent inference tasks. This method has recently re-emerged as the physics-informed neural networks (PINNs) [128] and is widely used.

Despite their implementation simplicity and offering fast inference on accelerated hardware, these methods suffer from several shortcomings:

1. lack controllable accuracy and convergence properties of finite discretization methods [129],
2. computing the loss and optimizing it involves evaluation of second order (and higher order) gradients using automatic differentiation (AD) through deep neural networks which leads to evaluating exponentially large computational graphs that is extremely memory-intensive, slow, and impractical to scale,
3. the basic assumption that automatic differentiation capabilities of current machine learning frameworks can evaluate “exact” derivatives across complex surrogate models is fundamentally flawed [130],
4. automatic differentiation of an *un-optimized* neural network during training to compute the spatial gradients does not offer exact gradients for the PDE. Although these derivatives are “exact” (see [130] for a discussion) within the parameters of the neural network, it is important to note that these derivatives do not represent the true spatial derivatives of the solution but *exact derivatives of an approximate*

function.

These shortcomings motivate pursuit of hybrid solvers to combine the performance advantages of neural network inference on modern accelerated hardware with the accuracy of finite discretization methods while reducing the computational costs and errors associated with excessive use of AD.

Hybridization efforts are algorithmic or architectural. One important algorithmic method is the deep Galerkin method (DGM) [131] that is a neural network extension of the mesh-free Galerkin method where the solution is represented as a deep neural network rather than a linear combination of basis functions. The mesh-free nature of DGM, that stems from the underlying mesh-free Galerkin method, enables solving problems in higher dimensions by training the neural network model to satisfy the PDE operator and its initial and boundary conditions on a randomly sampled set of points rather than on an exponentially large grid. Although the number of points is huge in higher dimensions, the algorithm can process training on smaller batches of data points sequentially. Besides, second order derivatives in PDEs are calculated by a Monte Carlo method that retain scaling to higher dimensions. Another important algorithmic method is the deep Ritz method for solving variational problems [132] that implements a deep neural network approximation of the trial function that is constrained by numerical quadrature rule for the variational functional, followed by stochastic gradient descent.

Architectural hybridization methods are based on differentiable numerical linear algebra. One emerging class involves implementing differentiable finite discretization solvers and embedding them in the neural network architectures that enable application of end-to-end differentiable gradient based optimization methods. Recently, differentiable solvers have been developed in JAX [133] for fluid dynamic problems, such as `Phi-Flow` [134], `JAX-CFD` [135], and `JAX-FLUIDS` [136]. These methods are suitable for inverse prob-

lems where an unknown field is modeled by the neural network, while the model influence is propagated by the differentiable solver into a measurable residual [2, 51, 52]. We also note the classic strategy for solving inverse problems is the adjoint method to obtain the gradient of the loss without differentiation across the solver [50]; however, deriving analytic expression for the adjoint equations is tedious, should be repeated after modification of the problem or its loss function, and can become impractical for multiphysics problems. Other important utilities of differentiable solvers are to model and correct for the solution errors of finite discretization methods [137], learning and controlling PDE systems [138, 139].

Neural networks are not only universal approximators of continuous functions, but also of nonlinear operators [140]. Although this fact has been leveraged using data-driven strategies for learning differential operators by many authors [141, 142, 143, 144], current authors have demonstrated utility of differentiable solvers to effectively train nonlinear operators without any data in a completely physics-driven fashion, see section on learning the inverse transforms in [2]. In subsequent work we will demonstrate how NBM can be used to train neural operators in a purely physics-driven fashion.

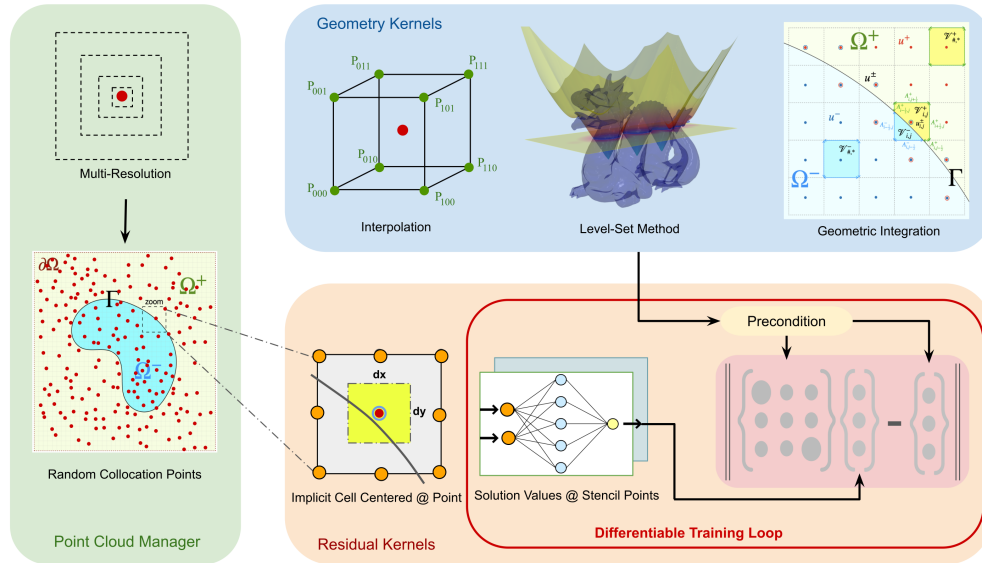
In this work we propose a novel algorithm for solving PDEs based on deep neural networks by lifting any existing mesh-based finite discretization method off of its underlying grid and extend it into a mesh-free method that can be applied to high dimensional problems on unstructured random points in an embarrassingly parallel fashion. In section 3.3 we present the neural bootstrapping method, next we apply it to an advanced finite volume discretization scheme for elliptic problems with jump conditions across irregular geometries in section 3.4. We show numerical results of the proposed framework on interfacial PDE problems in section 3.5 and conclude with section 3.7.

3.3 Neural Bootstrapping Method (NBM)

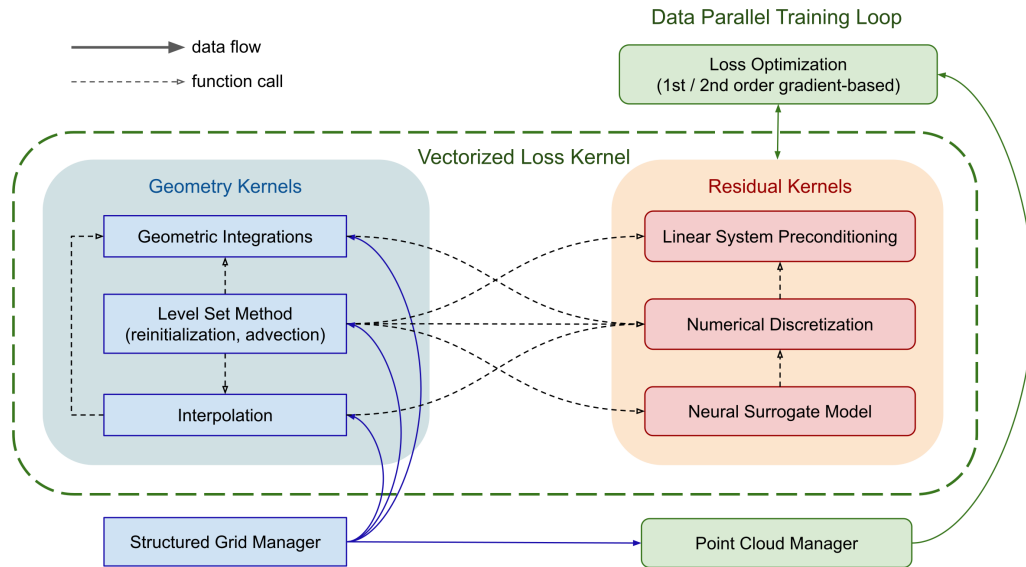
3.3.1 Algorithm

Figure 3.1 illustrates schematic of the proposed algorithm. Neural networks are used as surrogates for the solution function that are iteratively adjusted to minimize discretization residuals at a set of randomly sampled points and at arbitrary resolutions. The key idea is that neural networks can be evaluated over vertices of any discretization stencil centered at any point in the domain to emulate the effect of an structured mesh without ever materializing the mesh. Therefore, we use neural networks to bootstrap mesh based finite discretization (FD) methods to compile mesh free numerical methods. Operations in differentiable NBM kernels are:

1. A compute cell is implicitly constructed at any input coordinate and at a specified resolution. At the presence of discontinuities a coarse mesh encapsulates an interpolant for the level-set function whose intersection with the implicit cell is calculated to obtain necessary geometric information for the FD kernel and preconditioner.
2. FD kernel is applied on the compute cell where the solution values are evaluated by the neural network. Each kernel contributes a local linear system L^2 -norm residual $r_p = \|Au_p - b\|$ at a point p .
3. Residuals are preconditioned using common preconditioners to balance relative magnitude of contributions from different points and set them on equal level of importance before summing to produce a global loss value.
4. Gradient based optimization methods used in machine learning are applied to adjust neural network parameters. The automatic differentiation loop passes across the



(a) NBM kernels compute residual contribution by each collocation point per thread. Kernel operations involve considering implicit cells at different resolutions according to the bootstrapped finite discretization method. The point-wise evaluations at each implicit cell is locally preconditioned based on the geometry of the interfaces crossing the implicit cells.



(b) JAX-DIPS software architecture layout. Geometric information is managed by a mesh oracle that is a structured mesh at much lower resolution that stores the level-set function. The training loop involves automatic differentiation across the assembly of the linear system performed by the NBM kernels. Data distribution is achieved by composing the point-wise loss kernel with `jax.pmap` and `jax.vmap`.

Figure 3.1: Neural Bootstrapping Method (NBM) and the JAX-DIPS software architecture.

NBM kernels, see figure 3.3.1.

NBM training of neural network surrogate models for PDEs offers several benefits:

- FD methods offer guaranteed accuracy and controllable convergence properties for the training of neural network surrogate models. These are critical features for solving real-world complex physical systems using neural networks.
- NBM offers a straightforward path for applying mesh-based FD methods on unstructured random points. This is an important ability for augmenting observational data in the training pipelines.
- The algorithm is highly parallelizable and is ideally suited for GPU-accelerated computing paradigm.
- Multi-GPU parallel solution of PDE systems is reduced to the much simpler problem of data-parallel training using existing machine learning frameworks. Data parallelism involves distributing collocation points across multiple processors to compute gradient updates and then aggregating these locally computed updates [145].
- Only first order automatic differentiation is required for training PDE systems. This dramatically reduces memory requirements and computational costs associated with higher order AD computations across neural network models in other methods.
- Use of first order optimizers, enabled by differentiable finite discretizations, could improve scaling of traditional PDE solvers that use second order optimization techniques.

3.3.2 Neural network approximators for the solution

In 1987, Hecht and Nielson [146] applied an improved version of Kolmogorov’s 1957 superposition theorem [147], due to Sprecher [148], to the field of neurocomputing and demonstrated that a 3-layer feedforward neural network (one input layer with n inputs, one hidden layer with $2n + 1$ neurons, one output layer) are universal approximators for all *continuous* functions from the n -dimensional cube to a finite m -dimensional real vector space; *i.e.*, $f : [0, 1]^n \rightarrow \mathbb{R}^m$. Recently, Ismailov (2022) [149] demonstrated existence of neural networks implementing *discontinuous* functions, however efficient learning algorithms for such networks are not still available.

The solutions of interfacial PDE problems are discontinuous, with jumps appearing not only in the solution but also in the solution gradient. In light of above considerations, we define two separate neural networks to represent solution in Ω^- and Ω^+ regions:

$$u^+ = \mathcal{N}^+(\mathbf{x}) : \mathbb{R}^3 \cap \Omega^+ \rightarrow \mathbb{R} \quad u^- = \mathcal{N}^-(\mathbf{x}) : \mathbb{R}^3 \cap \Omega^- \rightarrow \mathbb{R}$$

We use SIREN neural networks, where we implement fully connected feedforward architecture with `sin` activation function and the output layer is a single linear neuron. Note that piecewise differentiable nonlinearities such as the `ReLU` function are inappropriate choices for representing solutions to differential equations. Weights and biases are initialized from a truncated normal distribution with zero mean and unit variance.

Solution networks are evaluated on sampled points in the domain while the parameters of these networks are optimized using the loss function. We define the loss function by the mean-squared-error (MSE) of the residual of the discretized partial differential equation

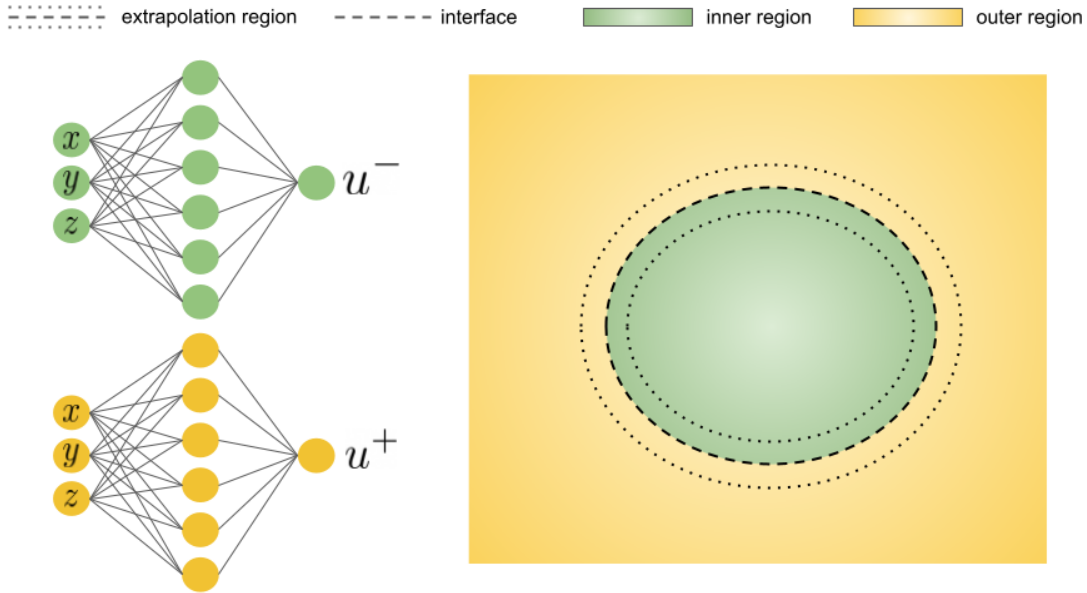


Figure 3.2: Two neural networks are defined for the two regions of the computational domain.

with jump conditions derived in section 3.4.1 that is evaluated on the grid points:

$$\mathcal{L}(u) = \|A\hat{u}_\theta(\mathbf{x}_{ijk} \in \Omega) - b\|_2^2$$

However, other choices such as Huber [150] or log-cosh [151] loss functions may improve results by automatically suppressing L_1 norm for larger residual values while minimizing L_2 norm for smaller values of the residual. JAX-DIPS allows for computation of the gradient of the loss function using automatic differentiation, *i.e.* $\nabla_\theta \mathcal{L}(u)$ where θ 's are network parameters. Therefore, our strategy is to leverage this capability and use first order optimizers developed in the deep learning community (such as Adam [79], *etc*) to minimize the aforementioned loss function. We emphasize the main benefit of using first order gradient based optimization algorithms is better memory efficiency that is suitable for large scale optimization problems with very large number of parameters in the neural

network model.

In the remainder of this manuscript we present details of applying NBM to solving elliptic problems with discontinuities across irregular interfaces.

3.4 JAX-DIPS: Differentiable Interfacial PDE Solver

We developed an end-to-end differentiable library for solving the elliptic problems with discontinuities in solution and solution gradient across irregular geometries. In JAX-DIPS, we bootstrap a sophisticated and modern finite volume discretization method [123]. The geometries are represented implicitly using the level-set function on a coarse mesh. We have implemented a uniform grid that supports operations such as interpolations, interface advection, integrations over interfaces as well as in domains. We describe the numerical algorithms for the level-set module and the elliptic solver in this section. Figure 3.3.1 illustrates a high-level overview of the JAX-DIPS software architecture.

Below we present and compare two possible approaches for treating the jump conditions in the interfacial PDE solver: (i) regression-based extrapolation, and (ii) neural extrapolation. The main difference between the two approaches is that approach (i) only requires first order AD for optimizing the loss, while (ii) effectively requires second order AD computations due to first evaluation of the loss and a second AD during optimization of the loss. Approach (i) offers better computational properties thanks to a complete bootstrapping of the underlying finite volume discretization method.

3.4.1 Approach I. Finite discretization method fused with regression-based extrapolation

For spatial discretizations at the presence of jump conditions we employ the numerical algorithm proposed by Bochkov and Gibou (2020) [152]. This method produces second-order accurate solutions and first-order accurate gradients in the L^∞ -norm, while having a compact stencil that makes it a good candidate for parallelization. Moreover, treatment of the interface jump conditions do not introduce any augmented variables, this preserves the homogeneous structure of the linear system. Most importantly, jump conditions only appear on the right-hand-side of the discretization and do not pollute the matrix term, this is beneficial for accelerating the solver. Here we use a background uniform 2D grid only for presentation of the finite volume discretization equations; we will not use this grid in the actual implementation but instead assume a local 3D cell around random points spanning in the domain during optimization.

At points where the finite volumes are crossed by Γ we have

$$\sum_{s=-,+} \int_{\Omega^s \cap \mathcal{V}_{i,j}} k^s u^s d\Omega - \sum_{s=-,+} \int_{\Omega^s \cap \partial \mathcal{V}_{i,j}} \mu^s \partial_{\mathbf{n}} u^s d\Gamma = \sum_{s=-,+} \int_{\Omega^s \cap \mathcal{V}_{i,j}} f^s d\Omega + \int_{\Gamma \cap \mathcal{V}_{i,j}} [\mu \partial_{\mathbf{n}} u] d\Gamma$$

following standard treatment of volumetric integrals and using central differencing for derivatives we obtain in 2D (with trivial 3D extension)

$$\begin{aligned} & \sum_{s=-,+} k_{i,j}^s u_{i,j}^s |\mathcal{V}_{i,j}^s| - \sum_{s=-,+} \left(\mu_{i-\frac{1}{2},j}^s A_{i-\frac{1}{2},j}^s \frac{u_{i-1,j}^s - u_{i,j}^s}{\Delta x} + \mu_{i+\frac{1}{2},j}^s A_{i+\frac{1}{2},j}^s \frac{u_{i+1,j}^s - u_{i,j}^s}{\Delta x} + \right. \\ & \left. \mu_{i,j-\frac{1}{2}}^s A_{i,j-\frac{1}{2}}^s \frac{u_{i,j-1}^s - u_{i,j}^s}{\Delta y} + \mu_{i,j+\frac{1}{2}}^s A_{i,j+\frac{1}{2}}^s \frac{u_{i,j+1}^s - u_{i,j}^s}{\Delta y} \right) \\ & = \sum_{s=-,+} f_{i,j}^s |\mathcal{V}_{i,j}^s| + \int_{\Gamma \cap \mathcal{V}_{i,j}} \beta d\Gamma + \mathcal{O}(\max(\Delta x, \Delta y)^D) \end{aligned}$$

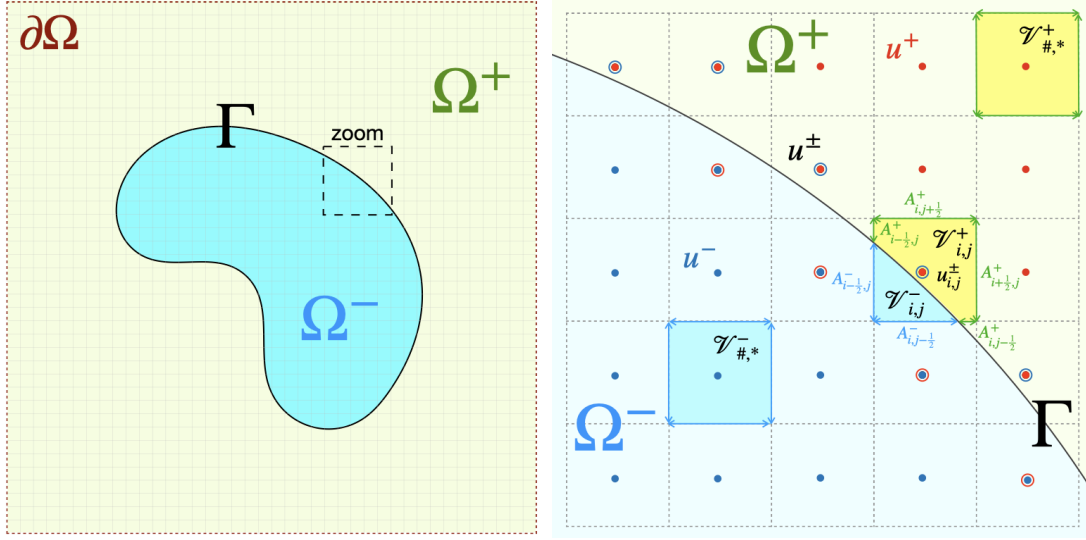


Figure 3.3: Notation used in this paper. Close to the interface where finite volumes are crossed by the interface, there are extra degrees of freedom (open circles) that are extrapolations of solutions from each domain to the opposite domain. Jump conditions are implicitly encoded in these extrapolated values.

where \mathcal{D} is the problem dimensionality. Note that far from interface either $s = -$ (for $\mathbf{x} \in \Omega^-$) or $s = +$ (for $\mathbf{x} \in \Omega^+$) is retained. This is automatically considered through zero values for sub-volumes $|\mathcal{V}_{i,j}^+|$ and $|\mathcal{V}_{i,j}^-|$ as well as their face areas. Note that $\mu_{i-1/2,j}^-$ (or $\mu_{i-1/2,j}^+$) corresponds to the value of diffusion coefficient at the middle of segment $A_{i-1/2,j}^-$ (or $A_{i-1/2,j}^+$) respectively, same is true for other edges as well. However, there are extra degrees of freedom on grid points whose finite volumes are crossed by the interface; *i.e.*, see double circles in figure 3.3. [152] derived analytical expressions for the extra degrees of freedom (u^+ in Ω^- and u^- in Ω^+) in terms of the original degrees of freedom (u^- in Ω^- and u^+ in Ω^+) as well as the jump conditions, this preserves the original $N_x \times N_y$ system size. The basic idea is to extrapolate the jump at grid point from jump condition at the projected point onto the interface using a Taylor expansion: $u_{i,j}^+ - u_{i,j}^- = [u]_{\mathbf{r}_{i,j}^{pr}} + \delta_{i,j}(\partial_{\mathbf{n}} u^+(\mathbf{r}_{i,j}^{pr}) - \partial_{\mathbf{n}} u^-(\mathbf{r}_{i,j}^{pr}))$. The unknown value ($u_{i,j}^-$ or $u_{i,j}^+$) is obtained based on approximation of the normal derivatives (*i.e.* $\partial_{\mathbf{n}} u^\pm(\mathbf{r}_{i,j}^{pr})$) which are computed using a least squares calculation on neighboring grid points that are in the

fast-diffusion region (referred to as “Bias Fast”) or in the slow diffusion region (referred to as “Bias Slow”). This makes two sets of rules for unknown values $u_{i,j}^\pm$.

In two dimensions and on uniform grids, the gradient operator at the grid cell (i, j) that is crossed by an interface is estimated by a least squares solution given by

$$(\nabla u^\pm)_{i,j} = \mathbf{D}_{i,j}^\pm \begin{bmatrix} u_{i-1,j-1} - u_{i,j}^\pm \\ u_{i,j-1} - u_{i,j}^\pm \\ \vdots \\ u_{i+1,j+1} - u_{i,j}^\pm \end{bmatrix} \quad \mathbf{D}_{i,j}^\pm = (X_{i,j}^T W_{i,j}^\pm X_{i,j})^{-1} (W_{i,j}^\pm X_{i,j})^T$$

and

$$W_{i,j}^\pm = \begin{bmatrix} \omega_{i,j}^\pm(-1, -1) & & & & & & & & \\ & \omega_{i,j}^\pm(0, -1) & & & & & & & \\ & & \ddots & & & & & & \\ & & & \omega_{i,j}^\pm(1, 1) & & & & & \end{bmatrix} \quad X_{i,j} = \begin{bmatrix} -h_x & -h_y \\ 0 & -h_y \\ h_x & -h_y \\ -h_x & 0 \\ 0 & 0 \\ h_x & 0 \\ -h_x & h_y \\ 0 & h_y \\ h_x & h_y \end{bmatrix}$$

and

$$\omega_{i,j}^\pm(p, q) = \begin{cases} 1 & (p, q) \in N_{i,j}^\pm \\ 0 & \text{else} \end{cases} \quad (3.1)$$

In this case, $D_{i,j}^\pm$ is a 2×9 matrix and we denote each of its 2×1 columns with $d_{i,j,p,q}^\pm$

$$\mathbf{D}_{i,j}^\pm = \begin{bmatrix} d_{i,j,-1,-1}^\pm & d_{i,j,0,-1}^\pm & d_{i,j,1,-1}^\pm & d_{i,j,-1,0}^\pm & d_{i,j,0,0}^\pm & d_{i,j,1,0}^\pm & d_{i,j,-1,1}^\pm & d_{i,j,0,1}^\pm & d_{i,j,1,1}^\pm \end{bmatrix}$$

The least square coefficients are then obtained by dot product of normal vector with these columns

$$c_{i,j,p,q}^\pm = \mathbf{n}_{i,j}^T d_{i,j,p,q}^\pm$$

and normal derivative can be computed (noting that $c_{i,j}^\pm = -\sum_{(p,q) \in N_{i,j}^\pm} c_{i,j,p,q}^\pm$)

$$\partial_n u^\pm(\mathbf{r}_{i,j}^{proj}) = c_{i,j}^\pm u_{i,j}^\pm + \sum_{(p,q) \in N_{i,j}^\pm} c_{i,j,p,q}^\pm u_{i+p,j+q}^\pm + \mathcal{O}(h)$$

At this point we can define a few intermediate variables at each grid point to simplify the presentation of the method,

$$\begin{aligned} \zeta_{i,j,p,q}^\pm &:= \delta_{i,j} \frac{[\mu]}{\mu^\mp} c_{i,j,p,q}^\pm & \zeta_{i,j}^\pm &:= - \sum_{(p,q) \in N_{i,j}^\pm} \zeta_{i,j,p,q}^\pm \\ \gamma_{i,j,p,q}^\pm &:= \frac{\zeta_{i,j,p,q}^\pm}{1 \pm \zeta_{i,j}^\pm} & \gamma_{i,j}^\pm &:= - \sum_{(p,q) \in N_{i,j}^\pm} \gamma_{i,j,p,q}^\pm \end{aligned}$$

where the set of neighboring grid points are

$$N_{i,j}^\pm = \{(p,q) : p = -1, 0, 1, \quad q = -1, 0, 1, \quad (p,q) \neq (0,0), \quad \mathbf{x}_{i+p,j+q} \in \Omega^\pm\}$$

and $\delta_{i,j}$ is the signed distance from $\mathbf{x}_{i,j}$ that is computed from the level-set function $\phi(\mathbf{x})$

$$\delta_{i,j} = \frac{\phi(\mathbf{x}_{i,j})}{|\nabla \phi(\mathbf{x}_{i,j})|}$$

- Rules based on approximating $\partial_{\mathbf{n}} u^+(\mathbf{r}_{i,j}^{pr})$:

$$u_{i,j}^- = \begin{cases} u_{i,j} & \mathbf{x}_{i,j} \in \Omega^- \\ u_{i,j}(1 - \gamma_{i,j}^-) - \sum_{(p,q) \in N_{i,j}^-} \gamma_{i,j,p,q}^- u_{i+p,j+q} - (\alpha + \frac{\delta_{i,j}\beta}{\mu^+})(1 - \gamma_{i,j}^-) & \mathbf{x}_{i,j} \in \Omega^+ \end{cases} \quad (3.2)$$

$$u_{i,j}^+ = \begin{cases} u_{i,j}(1 - \zeta_{i,j}^-) - \sum_{(p,q) \in N_{i,j}^-} \zeta_{i,j,p,q}^- u_{i+p,j+q} + \alpha + \delta_{i,j} \frac{\beta}{\mu^+} & \mathbf{x}_{i,j} \in \Omega^- \\ u_{i,j} & \mathbf{x}_{i,j} \in \Omega^+ \end{cases} \quad (3.3)$$

It is useful to cast this in the form of matrix kernel operations through defining intermediate tensors:

$$\mathbf{\Gamma}_{i,j} := \begin{bmatrix} \gamma_{i-1,j+1}^- & \gamma_{i,j+1}^- & \gamma_{i+1,j+1}^- \\ \gamma_{i-1,j}^- & \gamma_{i,j}^- & \gamma_{i+1,j}^- \\ \gamma_{i-1,j-1}^- & \gamma_{i,j-1}^- & \gamma_{i+1,j-1}^- \end{bmatrix}, \quad \mathbf{\zeta}_{i,j} := \begin{bmatrix} \zeta_{i-1,j+1}^- & \zeta_{i,j+1}^- & \zeta_{i+1,j+1}^- \\ \zeta_{i-1,j}^- & \zeta_{i,j}^- & \zeta_{i+1,j}^- \\ \zeta_{i-1,j-1}^- & \zeta_{i,j-1}^- & \zeta_{i+1,j-1}^- \end{bmatrix}$$

$$\mathbf{U}_{i,j} := \begin{bmatrix} u_{i-1,j+1} & u_{i,j+1} & u_{i+1,j+1} \\ u_{i-1,j} & u_{i,j} & u_{i+1,j} \\ u_{i-1,j-1} & u_{i,j-1} & u_{i+1,j-1} \end{bmatrix}, \quad \mathbf{N}_{i,j}^\pm := \begin{bmatrix} \omega_{i,j}^\pm(-1,1) & \omega_{i,j}^\pm(0,1) & \omega_{i,j}^\pm(1,1) \\ \omega_{i,j}^\pm(-1,0) & 0 & \omega_{i,j}^\pm(1,0) \\ \omega_{i,j}^\pm(-1,-1) & \omega_{i,j}^\pm(0,-1) & \omega_{i,j}^\pm(1,-1) \end{bmatrix}$$

where \mathbf{N}^- is a masking filter that passes the values in the negative neighborhood of node (i, j) .

We also introduce the Hadamard product \odot between two identical matrices that creates another identical matrix with each entry being elementwise products. Moreover, double contraction of two tensors A and B is defined by $A : B = \sum A \odot B$ which is a scalar value and equals the sum of all entries of the Hadamard product of the tensors; *i.e.*, note $A : A$ is square of Frobenius norm of A . Using these notations, the substitution

rules read

$$u_{i,j}^- = \begin{cases} u_{i,j} & \mathbf{x}_{i,j} \in \Omega^- \\ (1 + \mathbf{\Gamma}_{i,j}^- : \mathbf{N}_{i,j}^-)u_{i,j} - (\mathbf{\Gamma}_{i,j}^- \odot \mathbf{N}_{i,j}^-) : \mathbf{U}_{i,j} - (\alpha + \delta_{i,j} \frac{\beta}{\mu^+})(1 + \mathbf{\Gamma}_{i,j}^- : \mathbf{N}_{i,j}^-) & \mathbf{x}_{i,j} \in \Omega^+ \end{cases} \quad (3.4)$$

$$u_{i,j}^+ = \begin{cases} (1 + \mathbf{\zeta}_{i,j}^- : \mathbf{N}_{i,j}^-)u_{i,j} - (\mathbf{\zeta}_{i,j}^- \odot \mathbf{N}_{i,j}^-) : \mathbf{U}_{i,j} + \alpha + \delta_{i,j} \frac{\beta}{\mu^+} & \mathbf{x}_{i,j} \in \Omega^- \\ u_{i,j} & \mathbf{x}_{i,j} \in \Omega^+ \end{cases} \quad (3.5)$$

- Rules based on approximating $\partial_{\mathbf{n}} u^-(\mathbf{r}_{i,j}^{pr})$:

$$u_{i,j}^- = \begin{cases} u_{i,j} & \mathbf{x}_{i,j} \in \Omega^- \\ u_{i,j}(1 - \zeta_{i,j}^+) - \sum_{(p,q) \in N_{i,j}^+} \zeta_{i,j,p,q}^+ u_{i+p,j+q} - \alpha - \delta_{i,j} \frac{\beta}{\mu^-} & \mathbf{x}_{i,j} \in \Omega^+ \end{cases} \quad (3.6)$$

$$u_{i,j}^+ = \begin{cases} u_{i,j}(1 - \gamma_{i,j}^+) - \sum_{(p,q) \in N_{i,j}^+} \gamma_{i,j,p,q}^+ u_{i+p,j+q} + (\alpha + \delta_{i,j} \frac{\beta}{\mu^-})(1 - \gamma_{i,j}^+) & \mathbf{x}_{i,j} \in \Omega^- \\ u_{i,j} & \mathbf{x}_{i,j} \in \Omega^+ \end{cases} \quad (3.7)$$

in matrix notation we have

$$u_{i,j}^- = \begin{cases} u_{i,j} & \mathbf{x}_{i,j} \in \Omega^- \\ (1 + \mathbf{\zeta}_{i,j}^+ : \mathbf{N}_{i,j}^+)u_{i,j} - (\mathbf{\zeta}_{i,j}^+ \odot \mathbf{N}_{i,j}^+) : \mathbf{U}_{i,j} - \alpha - \delta_{i,j} \frac{\beta}{\mu^-} & \mathbf{x}_{i,j} \in \Omega^+ \end{cases} \quad (3.8)$$

$$u_{i,j}^+ = \begin{cases} (1 + \mathbf{\Gamma}_{i,j}^+ : \mathbf{N}_{i,j}^+)u_{i,j} - (\mathbf{\Gamma}_{i,j}^+ \odot \mathbf{N}_{i,j}^+) : \mathbf{U}_{i,j} + (\alpha + \delta_{i,j} \frac{\beta}{\mu^-})(1 + \mathbf{\Gamma}_{i,j}^+ : \mathbf{N}_{i,j}^+) & \mathbf{x}_{i,j} \in \Omega^- \\ u_{i,j} & \mathbf{x}_{i,j} \in \Omega^+ \end{cases} \quad (3.9)$$

The overall algorithm is summarized in Algorithm 1.

Algorithm 1 Bias Slow approximation of the non-existing solution value on a grid point based on existing solution values in its neighborhood. The notation is used for $u_{i,j}^\pm = B_{i,j}^\pm : \mathbf{U}_{i,j} + r_{i,j}^\pm$.

```

1: procedure BIAS SLOW
2:   if  $\Gamma \cap \mathcal{C}_{i,j} = \emptyset$  then
3:      $B_{i,j}^\pm = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$ ;  $r_{i,j}^\pm = 0$ 
4:   else
5:     if  $\mu_{i,j}^- > \mu_{i,j}^+$  then
6:       if  $\phi_{i,j} \geq 0$  then
7:          $B_{i,j}^+ = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$ ;  $r_{i,j}^+ = 0$ 
8:          $B_{i,j}^- = \begin{bmatrix} -\gamma_{i,j,-1,1}^- & -\gamma_{i,j,0,1}^- & -\gamma_{i,j,1,1}^- \\ -\gamma_{i,j,-1,0}^- & 1 - \gamma_{i,j}^- & -\gamma_{i,j,1,0}^- \\ -\gamma_{i,j,-1,-1}^- & -\gamma_{i,j,0,-1}^- & -\gamma_{i,j,1,-1}^- \end{bmatrix}$ ;  $r_{i,j}^- = -(\alpha_{i,j}^{proj} +$ 
9:            $\delta_{i,j} \frac{\beta_{i,j}^{proj}}{\mu_{proj}^+})(1 - \gamma_{i,j}^-)$ 
10:        else
11:           $B_{i,j}^+ = \begin{bmatrix} -\zeta_{i,j,-1,1}^- & -\zeta_{i,j,0,1}^- & -\zeta_{i,j,1,1}^- \\ -\zeta_{i,j,-1,0}^- & 1 - \zeta_{i,j}^- & -\zeta_{i,j,1,0}^- \\ -\zeta_{i,j,-1,-1}^- & -\zeta_{i,j,0,-1}^- & -\zeta_{i,j,1,-1}^- \end{bmatrix}$ ;  $r_{i,j}^+ = \alpha_{i,j}^{proj} + \delta_{i,j} \frac{\beta_{i,j}^{proj}}{\mu_{proj}^+}$ 
12:           $B_{i,j}^- = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$ ;  $r_{i,j}^- = 0$ 
13:        else
14:          if  $\phi_{i,j} \geq 0$  then
15:             $B_{i,j}^+ = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$ ;  $r_{i,j}^+ = 0$ 
16:             $B_{i,j}^- = \begin{bmatrix} -\zeta_{i,j,-1,1}^+ & -\zeta_{i,j,0,1}^+ & -\zeta_{i,j,1,1}^+ \\ -\zeta_{i,j,-1,0}^+ & 1 - \zeta_{i,j}^+ & -\zeta_{i,j,1,0}^+ \\ -\zeta_{i,j,-1,-1}^+ & -\zeta_{i,j,0,-1}^+ & -\zeta_{i,j,1,-1}^+ \end{bmatrix}$ ;  $r_{i,j}^- = \alpha_{i,j}^{proj} + \delta_{i,j} \frac{\beta_{i,j}^{proj}}{\mu_{proj}^-}$ 
17:          else
18:             $B_{i,j}^+ = \begin{bmatrix} -\gamma_{i,j,-1,1}^+ & -\gamma_{i,j,0,1}^+ & -\gamma_{i,j,1,1}^+ \\ -\gamma_{i,j,-1,0}^+ & 1 - \gamma_{i,j}^+ & -\gamma_{i,j,1,0}^+ \\ -\gamma_{i,j,-1,-1}^+ & -\gamma_{i,j,0,-1}^+ & -\gamma_{i,j,1,-1}^+ \end{bmatrix}$ ;  $r_{i,j}^+ = (\alpha_{i,j}^{proj} + \delta_{i,j} \frac{\beta_{i,j}^{proj}}{\mu_{proj}^-})(1 -$ 
19:               $\gamma_{i,j}^+)$ 
20:             $B_{i,j}^- = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$ ;  $r_{i,j}^- = 0$ 

```

3.4.2 Approach II. Finite discretization method fused with neural extrapolation

We point out that although in approach I we used a regression-based method to impose the jump conditions on the grid points around the interface, it is possible to evaluate the neural network models as interpolation and extrapolation functions within the finite discretization scheme. Using the neural network models for solutions, we are able to evaluate extrapolations of the solution functions in a banded region around the interface as illustrated in figure 3.2. Starting from the jump conditions, for points on the interface, $\mathbf{x} \in \Gamma$, we have

$$\begin{aligned} u^+ - u^- &= \alpha \\ \mu^+ \partial_n u^+ - \mu^- \partial_n u^- &= \beta \end{aligned}$$

and after Taylor expansion in the normal direction we obtain on the adjacent grid points (i, j)

$$u_{i,j}^+ - u_{i,j}^- = [u]_{\mathbf{r}_{i,j}^{pr}} + \delta_{i,j} (\partial_n u^+(\mathbf{r}_{i,j}^{pr}) - \partial_n u^-(\mathbf{r}_{i,j}^{pr})) \quad (3.10)$$

which explicitly incorporates the jump condition in the solutions. To incorporate the jump condition in fluxes we can rewrite either of the normal gradients in terms of the other

$$\begin{aligned} \partial_n u^+(\mathbf{r}_{i,j}^{pr}) &= \frac{\mu^-}{\mu^+} \partial_n u^-(\mathbf{r}_{i,j}^{pr}) + \frac{\beta}{\mu^+} \\ \partial_n u^-(\mathbf{r}_{i,j}^{pr}) &= \frac{\mu^+}{\mu^-} \partial_n u^+(\mathbf{r}_{i,j}^{pr}) - \frac{\beta}{\mu^-} \end{aligned}$$

which leads to two relationships among predictions of the two neural networks at each grid point in the banded extrapolation region

$$u_{i,j}^+ - u_{i,j}^- = \alpha(\mathbf{r}_{i,j}^{pr}) + \delta_{i,j} \left(\left(\frac{\mu^-}{\mu^+} - 1 \right) \partial_{\mathbf{n}} u^-(\mathbf{r}_{i,j}^{pr}) + \frac{\beta(\mathbf{r}_{i,j}^{pr})}{\mu^+} \right) \quad (3.11)$$

$$u_{i,j}^+ - u_{i,j}^- = \alpha(\mathbf{r}_{i,j}^{pr}) + \delta_{i,j} \left(\left(1 - \frac{\mu^+}{\mu^-} \right) \partial_{\mathbf{n}} u^+(\mathbf{r}_{i,j}^{pr}) + \frac{\beta(\mathbf{r}_{i,j}^{pr})}{\mu^-} \right) \quad (3.12)$$

Note that we are representing solution functions, $\hat{u}^\pm(\mathbf{r})$, with neural networks where computing the normal derivatives is trivial using automatic differentiation of the network along the normal directions. In contrast to finite discretization methods, solutions at off-grid points is readily available by simply evaluating the neural network function at any desired points. Note that we can compute the projected location on the interface starting from each grid point (i, j) using the level-set function:

$$\mathbf{r}_{ij}^{proj} = \mathbf{r}_{ij} - \delta_{i,j} \mathbf{n}_{i,j}$$

In the second approach, the loss function remains as before, except the unknown u^\pm values are derived using equations 3.11–3.12, instead of computing a regression-based extrapolation function based on the points in the neighborhood of interface cells:

$$\mathcal{L} = \left\| \sum_{s=-,+} k_{i,j}^s u_{i,j}^s |\mathcal{V}_{i,j}^s| - \sum_{s=-,+} \left(\mu_{i-\frac{1}{2},j}^s A_{i-\frac{1}{2},j}^s \frac{u_{i-1,j}^s - u_{i,j}^s}{\Delta x} + \mu_{i+\frac{1}{2},j}^s A_{i+\frac{1}{2},j}^s \frac{u_{i+1,j}^s - u_{i,j}^s}{\Delta x} + \mu_{i,j-\frac{1}{2}}^s A_{i,j-\frac{1}{2}}^s \frac{u_{i,j-1}^s - u_{i,j}^s}{\Delta y} + \mu_{i,j+\frac{1}{2}}^s A_{i,j+\frac{1}{2}}^s \frac{u_{i,j+1}^s - u_{i,j}^s}{\Delta y} \right) - \sum_{s=-,+} f_{i,j}^s |\mathcal{V}_{i,j}^s| - \int_{\Gamma \cap \mathcal{V}_{i,j}} \beta d\Gamma \right\|_2^2$$

However, there is a major downside with this approach for training because the automatic differentiation has to be applied on the network once more that effectively amounts to compute second-order derivatives of the network. This slows down convergence, and

the time-to-solution increases with square of depth of the neural network while in the regression-based method the cost grows linearly in the network depth by restricting to only first order automatic differentiation.

3.4.3 Optimization scheme

One of the main benefits of NBM is enabling the application of techniques from the vast literature on preconditioning linear systems to accelerate training of neural network models for the solution of PDEs. We note that in NBM these preconditioners do not need to be differentiable as long as their operations only depend on the geometry and physical properties of the domains, and not explicitly on the solution values of the PDE. Therefore existing software libraries for preconditioning could be used in JAX-DIPS. In this section we introduce the optimization techniques for training neural network models in JAX-DIPS.

Preconditioners are ideal network regularizers for solving PDEs

Finite discretization methods lead to solving a linear algebraic system with guarantees on convergence and accuracy. The geometric irregularities and fine-grain details of the system around interfaces often lead to bad condition number for the linear system, which can be remedied by applying preconditioners. Intuitively, condition number is caused by a separation of scales for geometric lengthscales or material properties that underly the solution patterns. One of the strengths of the presented approach is to readily enable usage of preconditioners for training neural network surrogate models.

Preconditioners are a powerful technique to accelerate convergence of traditional numerical linear algebraic solvers. Given a poorly conditioned linear system $Ax = b$ one can obtain an equivalent system $\hat{A}\hat{x} = \hat{b}$ with accelerated convergence rate when using iter-

ative gradient based methods. For the conjugate gradient method convergence iteration is proportional to $\sqrt{\kappa(A)}$ where $\kappa(A)$ is the condition number of matrix A . Preconditioning is achieved by mapping the linear system with a nonsingular matrix M into a new space $M^{-1}Ax = M^{-1}b$ where $M^{-1}A$ has more regular spread of eigenvalues, hence a better condition number. The precondition matrix M should approximate A^{-1} such that $|I - M^{-1}A| < 1$. The simplest choice is the Jacobi preconditioner which amounts to using the diagonal part of A as the preconditioner, $M = \text{diag}(A)$. Note that the diagonal term is locally available at each point and it is straightforward to parallelize.

In this work we use the Jacobi pre-conditioner. Basically, every element of the left-hand-side (Au) and right-hand-side (b) vectors are divided by the coefficient of the diagonal term of the matrix given by:

$$a_{ii} = \sum_{s=-,+} \left(k_{i,i}^s |\mathcal{V}_{i,i}^s| + (\mu_{i-\frac{1}{2},i}^s A_{i-\frac{1}{2},i}^s + \mu_{i+\frac{1}{2},i}^s A_{i+\frac{1}{2},i}^s) / \Delta x + (\mu_{i,i-\frac{1}{2}}^s A_{i,i-\frac{1}{2}}^s + \mu_{i,i+\frac{1}{2}}^s A_{i,i+\frac{1}{2}}^s) / \Delta y \right)$$

Note that for memory efficiency we never explicitly compute the matrix, instead we compute the effect of matrix product of Au .

Learning rate scheduling

First order methods have longer time-to-solution but require less memory, while second order methods are faster to converge but require massive memory footprint. In JAX-DIPS we primarily utilize first order optimization methods such as Adam [79]. Second order methods such as Newton or BFGS certainly offer convergence in less iterations but require much more memory. Traditionally used GMRES or Conjugate Gradient methods for sparse linear systems are somewhere between first order and second order optimization methods that are based on building basis vectors by computing gradients that are conjugate to each other $\mathbf{p}_j^T \mathbf{A} \mathbf{p}_i = 0$ and will converge to the solution in at most

n steps; *i.e.*, at most the solution vector is spanned in the full basis. We found that starting from a zero guess for the solution it is important to start from a large learning rate and gradually decay the learning rate in a process of exponential annealing. For this purpose, we use the exponential decay scheduler provided by `Optax` [153] to control the learning rate in the Adam optimizer:

$$r_k = r_0 \alpha^{k/T}$$

where r_k is the learning rate at step k of optimization, $\alpha < 1$ is the decay rate, and T is the decay count-scale. By default, we set $T = 100$, $\alpha = 0.975$, starting from an initial value of $r_0 = 10^{-2}$ and clip gradients by maximum global gradient norm (to a value 1) [154] before applying the Adam updates in each step. We note a larger decay rate, *e.g.* $\alpha = 0.98$, leads to small oscillations after 10000 steps and although similar levels of accuracy can be achieved at much less iterations, here we report results with the more robust decay rate.

Domain switching optimization scheme

The linear system suffers from worse condition number in the domain with more variability in diffusion coefficient, or where diffusion coefficient is larger; *i.e.*, the fast region. This leads to regionally unbalanced solution error where the overall error is systematically lopsided by the faster diffusion region. We found this problem can be improved by interleaving region-specific optimization epochs in the training pipeline, where only one of the networks is updated based on the loss computed in its region. See Algorithm 2 for details of the algorithm. We note that an alternative strategy for addressing this issue is scaling the gradients for the fast region with respect to the slow region during training.

Algorithm 2 Domain switching method. Switching interval is τ .

```

1: procedure DOMAIN SWITCHING OPTIMIZATION
2:   for epoch in  $0 \cdots N$  do
3:     region = Region(epoch)
4:     if region > 0 then
5:       if  $\mu^- > \mu^+$  then
6:         optimize  $u_{NN}^-$  in  $\Omega^-$  given fixed  $u_{NN}^+$ 
7:       else
8:         optimize  $u_{NN}^+$  in  $\Omega^+$  given fixed  $u_{NN}^-$ 
9:
10:    if region == 0 then
11:      optimize both networks in  $\Omega^- \cup \Omega^+$ 
12:
13:    if region < 0 then
14:      if  $\mu^- < \mu^+$  then
15:        optimize  $u_{NN}^-$  in  $\Omega^-$  given fixed  $u_{NN}^+$ 
16:      else
17:        optimize  $u_{NN}^+$  in  $\Omega^+$  given fixed  $u_{NN}^-$ 
18:
19:  procedure REGION(epoch)
20:    if mode == whole region  $\rightarrow$  fast region then
21:      region = epoch %  $\tau$ 
22:    if mode == fast region  $\rightarrow$  whole region  $\rightarrow$  slow region then
23:      region =  $\tau/2 - \text{epoch} \% \tau$ 

```

Multi-GPU parallelization with data parallel training

The NBM is embarrassingly parallel and residual evaluation at each point is independent from other points. Therefore, multi-GPU parallelization does not involve inter-GPU communication for evaluating the residuals per point. We partition the training points and distribute them along with copies of neural network parameters among multiple GPUs to compute gradient updates per batch. Then we aggregate these updates by averaging the values on different GPUs. The updates are then broadcasted and model parameters are updated on each device [145].

The ability to batch over grid points is one of the key enabling factors for reaching higher resolutions and higher dimensions, as it allows to set a limit on the required GPU memory. With NBM it is straightforward to scale finite discretization methods on GPU clusters.

3.5 Numerical Results

We consider examples for solution to elliptic problems of the form

$$\begin{aligned} k^\pm u^\pm - \nabla \cdot (\mu^\pm \nabla u^\pm) &= f^\pm, & \mathbf{x} \in \Omega^\pm \\ [u] &= \alpha, & \mathbf{x} \in \Gamma \\ [\mu \partial_{\mathbf{n}} u] &= \beta, & \mathbf{x} \in \Gamma \end{aligned}$$

Using different features of JAX-DIPS one can compose solvers with different training configurations; *i.e.*, single/multi-resolution, single/multi-batch, and single/multi-GPU, and domain alternating training. Moreover, the neural extrapolation method discussed in section 3.4.2 provides an alternative solver. Below we implement and compare numerical accuracy and performance of these strategies.

For each accuracy metric we report *order of convergence*. Order of convergence, denoted by p , is computed by doubling the number of grid points in every dimension and measuring the L^∞ error of solution and its gradient over all the grid points in the domain:

$$\frac{\text{err}(2h)}{\text{err}(h)} = 2^p \rightarrow p = \log_2 \left(\frac{\text{err}(2h)}{\text{err}(h)} \right) \quad h = \min(h_x, h_y, h_z)$$

3.5.1 Accuracy in the bulk: no interface

As baseline we consider the solution in the bulk at the absence of interfaces. Computational domain is $\Omega \in [-1, 1]^3$ with the exact solution given by $u(x, y, z) = \cos(x) \sin(y) \cos(z)$, coefficients $\mu = 1$ and $k = 0$ and the source term is $f(x, y, z) = 3 \cos(x) \sin(y) \cos(z)$. The accuracy results are reported in Table 3.1. The neural network consists of 5 hidden layers with 10 neurons with **CeLU** activation function, consisting of 491 total trainable parameters. In each case the batch size is equal to the number of training grid points in order to ensure only a single batch training. We note this setting identifies the maximum memory usage and the minimum time per epoch. Decreasing the batch size reduces the amount of GPU memory that is required by folding the training data into several passes of the optimizer at the expense of increasing the time needed per epoch. This degree of freedom accomodates for adjusting to the available hardware specifications.

To quantify the solution error we construct a high resolution *evaluation grid* with $256 \times 256 \times 256$ grid points to account for spatial regions outside of the training grid. After training the network with specified training resolution we evaluate the network over the evaluation grid and compute RMSE, L^∞ , and relative L^2 errors. We also report the GPU utilization during our experiments. It is also important to note the number of epochs determines the level of accuracy in our experiments, here we wait 10,000 epochs

before measuring the errors.

Table 3.1: Convergence test of example 3.5.1. Timings include the initial compilation time. Measurements are on a single NVIDIA A6000 GPU. The model has 5 hidden layers with 10 neurons each, overall 491 trainable parameters.

$N_{x,y,z}$	RMSE		L^∞		Rel. L^2		GPU Statistics	
	Solution	Order	Solution	Order	Solution	Order	t (sec/epoch)	VRAM (GB)
2^3	1.07×10^{-1}	—	2.13×10^{-2}	—	2.42×10^{-2}	—	0.012	0.81
2^4	8.04×10^{-3}	3.73	1.67×10^{-2}	0.35	1.80×10^{-2}	0.43	0.024	2.22
2^5	1.83×10^{-3}	2.13	5.24×10^{-3}	1.67	4.09×10^{-3}	2.13	0.091	5.77

3.5.2 Accuracy on spherical interface: single-resolution, single batch, single GPU

We use a single uniform grid and train on all the points in a single batch. We consider a sphere $\phi(\mathbf{x}) = \sqrt{x^2 + y^2 + z^2} - 0.5$ centered in a box $\Omega : [-1, 1]^3$ with the exact solution

$$\begin{aligned} u^-(x, y, z) &= e^z, & \phi(\mathbf{x}) < 0 \\ u^+(x, y, z) &= \cos(x) \sin(y), & \phi(\mathbf{x}) \geq 0 \end{aligned}$$

and variable diffusion coefficients

$$\begin{aligned} \mu^-(x, y, z) &= y^2 \ln(x + 2) + 4 & \phi(\mathbf{x}) < 0 \\ \mu^+(x, y, z) &= e^{-z} & \phi(\mathbf{x}) \geq 0 \end{aligned}$$

that imply variable source terms

$$\begin{aligned} f^-(x, y, z) &= -[y^2 \ln(x + 2) + 4]e^z & \phi(\mathbf{x}) < 0 \\ f^+(x, y, z) &= 2 \cos(x) \sin(y)e^{-z} & \phi(\mathbf{x}) \geq 0 \end{aligned}$$

The network has 5 hidden layers with 10 neurons in each layer using sine activation functions. Table 3.2 reports convergence results for the solution in the L^∞ -norm and root-mean-squared-error of the solution.

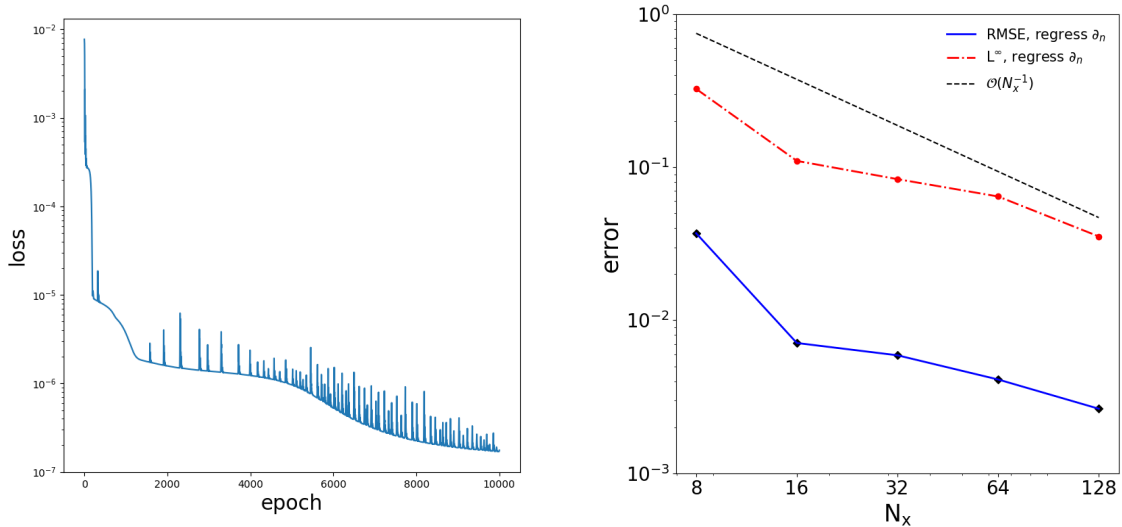
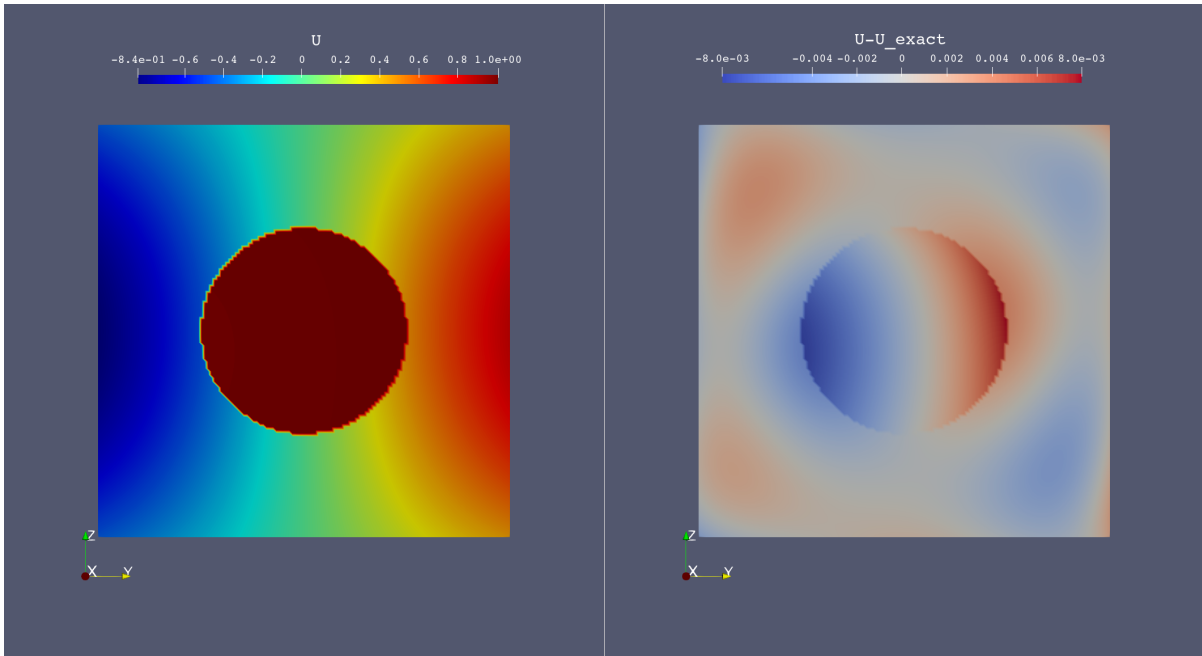


Figure 3.4: Loss evolution with epochs for the sphere of $16 \times 16 \times 16$ grid (left) and different accuracy measures, RMSE and L^∞ , at 5 different resolutions (right).

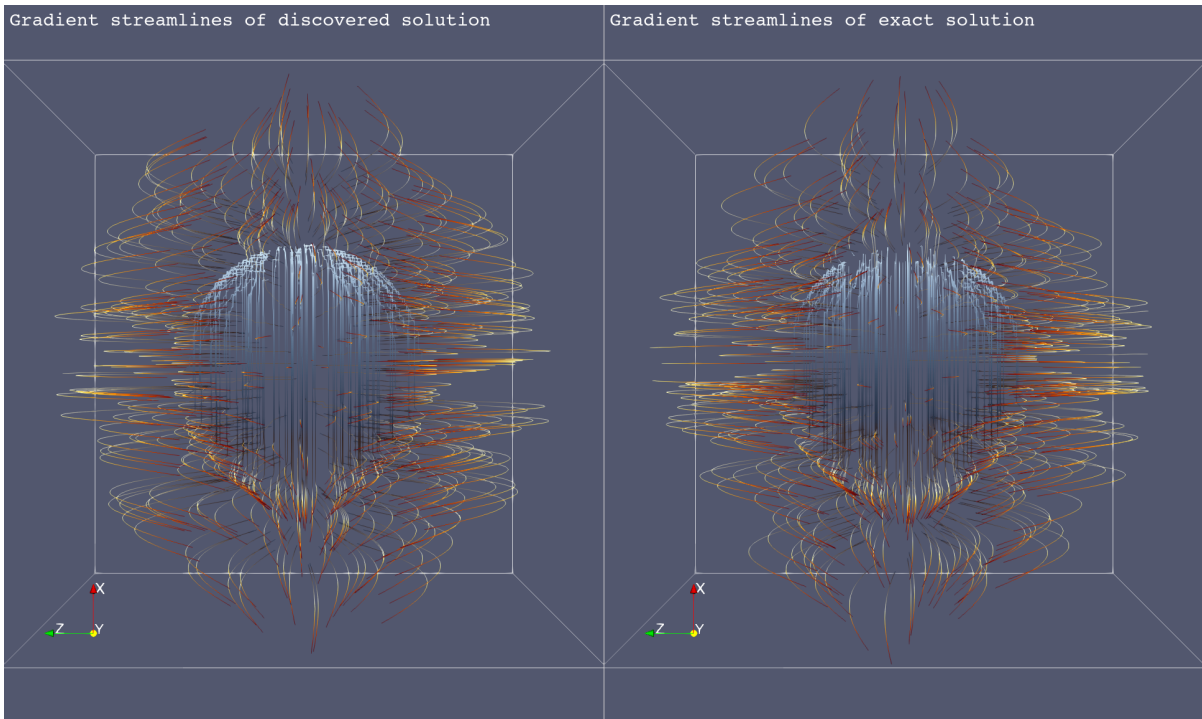
3.5.3 Accuracy on star interface: single GPU, domain switching, neural extrapolation, and batching

We consider a more irregular interface in this section where the less favorable condition number of the discretized linear system is known to degrade the accuracy of results in traditional solvers. Figures 5 and 6 in [152] characterize degradation of solution errors due to worse condition number in the method bootstrapped in our work. We note this is a main challenge facing all finite discretization schemes for interfacial PDE problems.

For the star example we use a pair of fully connected feedforward neural networks, each composed of 1 hidden layer and 100 neurons with `sine` activation function, followed by an output layer with 1 linear neuron. There are a total of 1,002 trainable parameters



(a) Illustration of numerical solution and absolute error on a cross section of the domain.



(b) Streamlines of solution gradient for (left) the surrogate neural model colored by model solution value, (right) exact streamlines colored by exact solution values.

Figure 3.5: The neural network surrogate model trained on a 128^3 grid using a single NVIDIA A6000 GPU.

Table 3.2: Convergence and timings for the sphere example averaged over 10,000 epochs. Timings include the initial compilation time. Measurements are on a single NVIDIA A6000 GPU. The regression-based method has 5 hidden layers with 10 neurons each, overall 982 trainable parameters.

$N_{x,y,z}$	RMSE		L^∞		GPU Statistics	
	Solution	Order	Solution	Order	t (sec/epoch)	VRAM (GB)
2^3	3.7×10^{-2}	-	3.25×10^{-1}	-	0.0306	1.05
2^4	7.1×10^{-3}	2.38	1.10×10^{-1}	1.56	0.056	1.72
2^5	5.9×10^{-3}	0.27	8.36×10^{-2}	0.4	0.053	2.15
2^6	4.1×10^{-3}	0.53	6.44×10^{-2}	0.38	0.287	5.57
2^7	2.64×10^{-3}	0.64	3.53×10^{-2}	0.87	2.125	32.1

in the model. We consider a star-shaped interface with inner and outer radii $r_i = 0.151$ and $r_e = 0.911$ that is immersed in a box $\Omega : [-1, 1]^3$ described by the level-set function

$$\phi(\mathbf{x}) = \sqrt{x^2 + y^2 + z^2} - r_0 \left(1 + \left(\frac{x^2 + y^2}{x^2 + y^2 + z^2} \right)^2 \sum_{k=1}^3 \beta_k \cos(n_k (\arctan(\frac{y}{x}) - \theta_k)) \right)$$

with the parameters

$$r_0 = 0.483, \quad \begin{pmatrix} n_1 \\ \beta_1 \\ \theta_1 \end{pmatrix} = \begin{pmatrix} 3 \\ 0.1 \\ 0.5 \end{pmatrix}, \quad \begin{pmatrix} n_2 \\ \beta_2 \\ \theta_2 \end{pmatrix} = \begin{pmatrix} 4 \\ -0.1 \\ 1.8 \end{pmatrix}, \quad \begin{pmatrix} n_3 \\ \beta_3 \\ \theta_3 \end{pmatrix} = \begin{pmatrix} 7 \\ 0.15 \\ 0 \end{pmatrix}$$

Considering an exact solution

$$u^-(x, y, z) = \sin(2x) \cos(2y) e^z, \quad \phi(\mathbf{x}) < 0$$

$$u^+(x, y, z) = \left[16 \left(\frac{y-x}{3} \right)^5 - 20 \left(\frac{y-x}{3} \right)^3 + 5 \left(\frac{y-x}{3} \right) \right] \ln(x+y+3) \cos(z), \quad \phi(\mathbf{x}) \geq 0$$

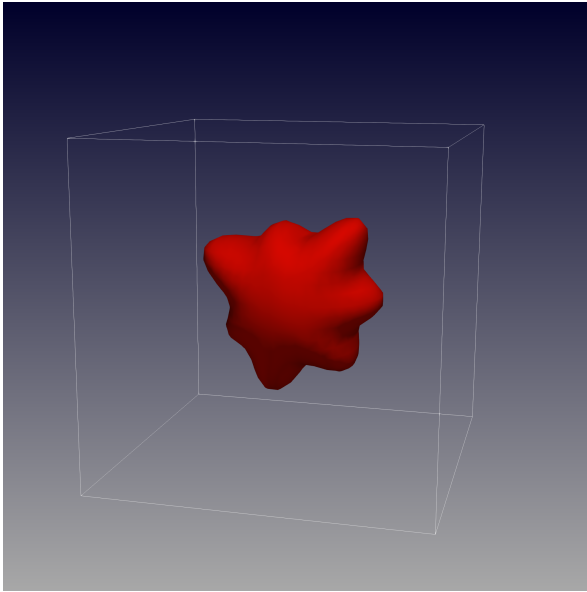
and the diffusion coefficient

$$\begin{aligned}\mu^-(x, y, z) &= 10 \left[1 + 0.2 \cos(2\pi(x + y)) \sin(2\pi(x - y)) \cos(z) \right] & \phi(\mathbf{x}) < 0 \\ \mu^+(x, y, z) &= 1 & \phi(\mathbf{x}) \geq 0\end{aligned}$$

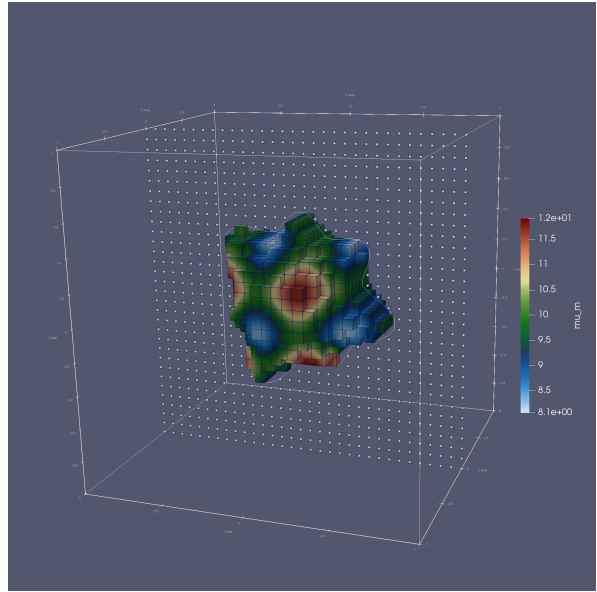
Table 3.3 compares approaches I and II for treating the jump conditions using domain switching optimization strategy, these results are illustrated in figure 3.6. Table 3.4 demonstrates the effect of batching and a multi-resolution training policy on the convergence. We observe convergence is still retained although at a lower rate, and batching can degrade the accuracy at finest resolution (demonstrated by the last row of table 3.4 highlighted in red). Although the results generally demonstrate convergence, they are clearly less accurate than what is currently possible by using traditional numerical solvers. In section 3.6.3 we describe strategies that may improve these results, importantly using better preconditioners on par with algebraic multigrid methods used in traditional solvers. However, we emphasize that in the case of neural network models considered here the number of degrees of freedom (~ 1000 trainable parameters) is significantly less than the number of degrees of freedom in traditional solvers (for example a grid of 128^3 has over 2 million unknown values), besides it is challenging to separate the effect of expressivity of the network (architecture) from the loss composition and the optimization method.

3.5.4 Time complexity and parallel scaling on GPU clusters

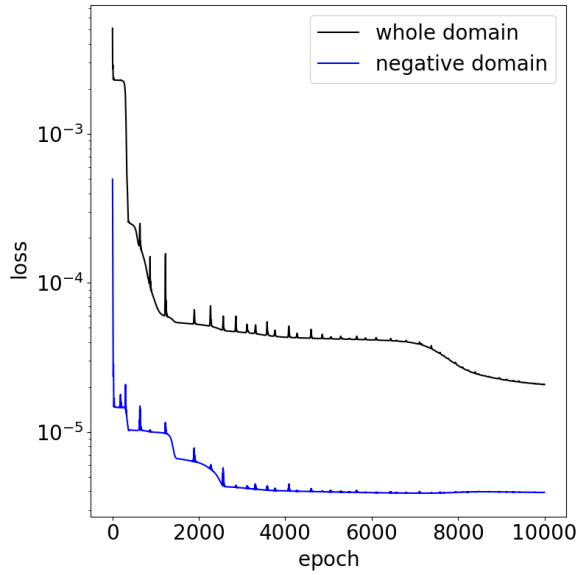
We adopt the problem setup presented in 3.5.3, however with a considerably more challenging geometry of the Dragon presented in [155]. In this case we used the signed-



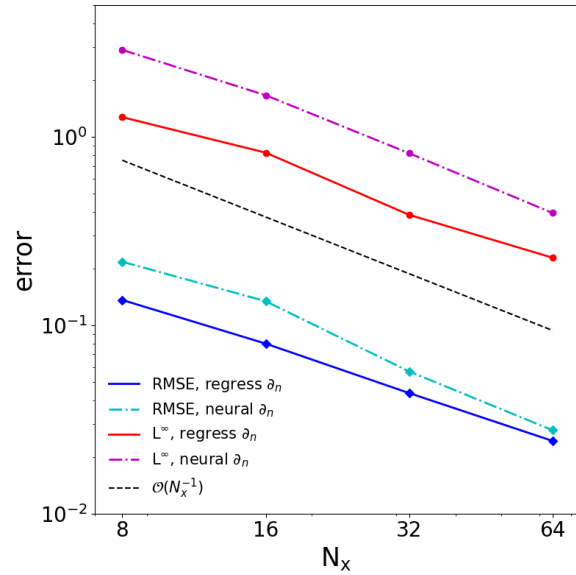
(a) Illustration of three dimensional interface used



(b) μ^\pm on the $32 \times 32 \times 32$ grid



(c) Loss evolution with epochs for the star of $64 \times 64 \times 64$ grid using domain switching training



(d) decrease in error by increasing resolutions

Figure 3.6: The neural network model trained with different configurations and resolutions.

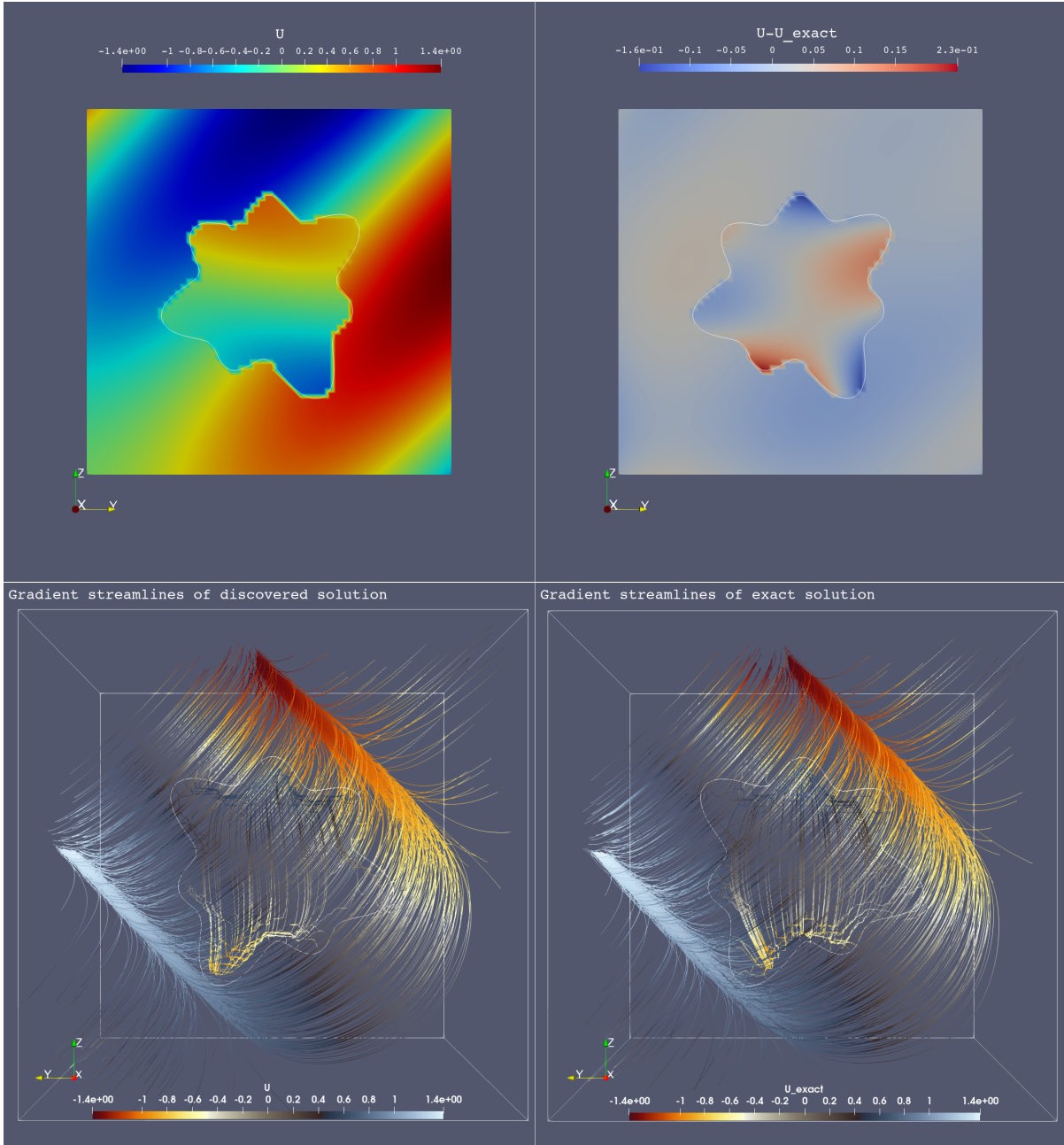


Figure 3.7: Illustration of exact and numerical solutions (top row) and gradient streamlines (bottom row) on a $64 \times 64 \times 64$ grid.

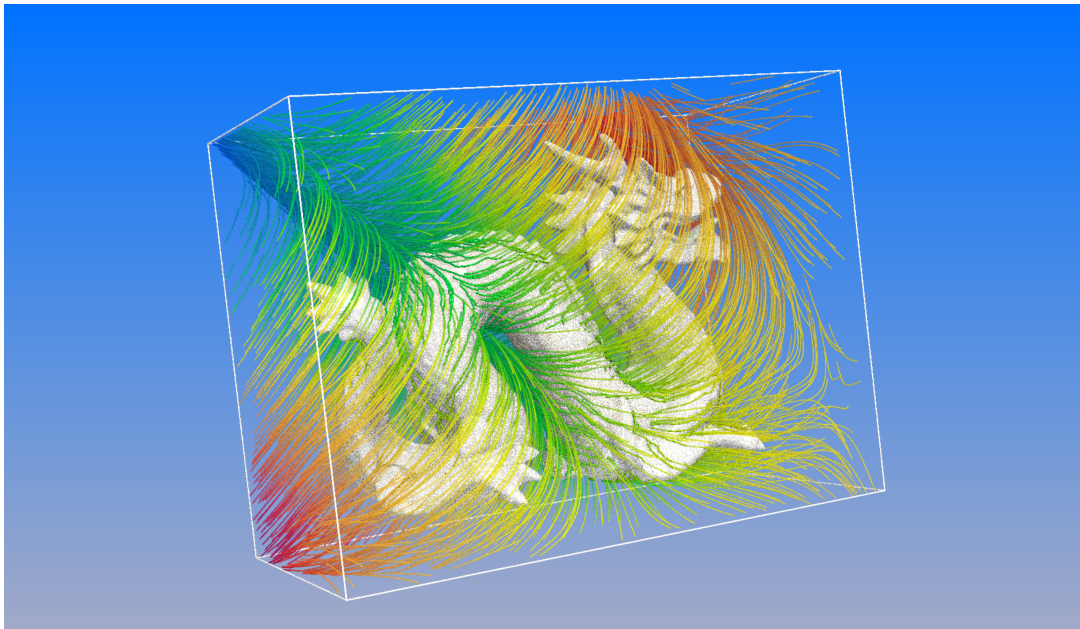
distance function produced by SDFGen, and initiated an interpolant based on its values.

The results are shown in figure 3.8, with a L^∞ -error of 0.5 and RMSE of 0.06 after 1000 epochs on a base resolution of 64^3 and implicitly refined onto multi-resolutions $128^3, 256^3, 512^3$. The neural network pair have only 1 hidden layer with 100 sine-activated neurons, although investigating more complex networks (transformers, symmetry preserving, *etc.*) would likely improve accuracy.

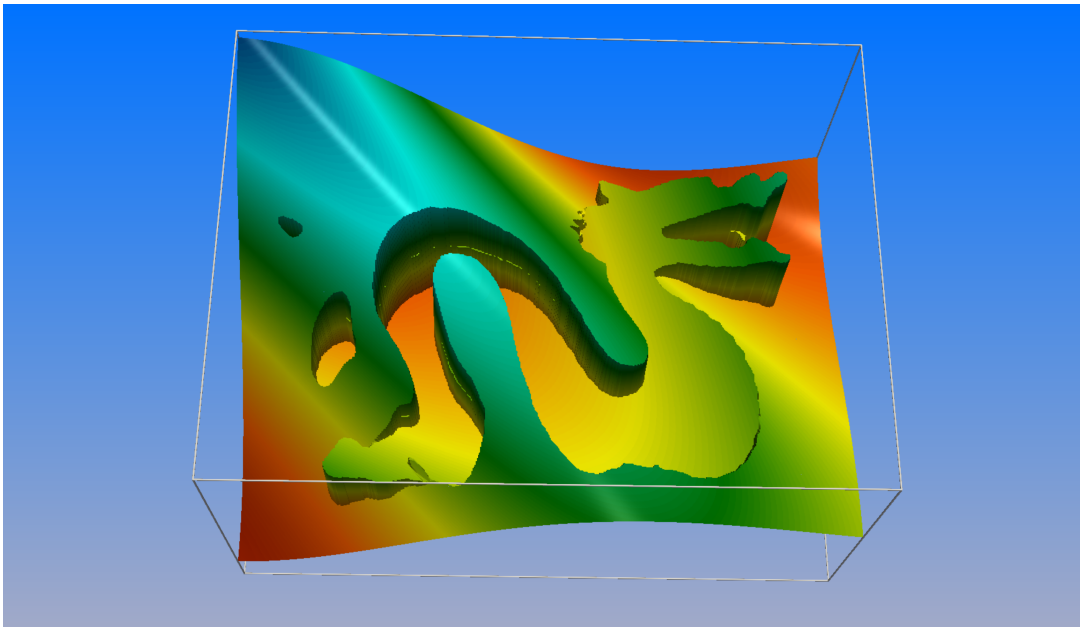
In table 3.5 we report scaling results on NVIDIA A100 GPUs at four base resolutions with three levels of implicit refinement. We used a batchsize of $32 \times 32 \times 16$ in all cases. At fixed number of GPUs, training time scales linearly (*i.e.*, optimal scaling) with the number of grid points. At a fixed resolution, increasing the number of GPUs accelerates training roughly with epoch time $\sim 1/\sqrt{\# \text{ GPUs}}$, although the advantage is more effective at higher resolutions. Compile time increases with resolution and decreases with number of GPUs. A maximum grid size of 1024^3 at multi-resolutions $1024^3, 2048^3, 4096^3, 8192^3$ was simulated on one NVIDIA DGX with 8 A100 GPUs. The results are shown in figure 3.8.

3.5.5 Comparison with other methods

Recently other methods have been proposed in the framework of PINNs for solving interfacial PDEs with jump conditions in two [156] and three spatial dimensions [157]. In this section we compare our method with the recently proposed interfaced neural networks (INN) algorithm of Wu *et al.* (2022) [157]. INN uses AD for computing gradients similar to other PINN-like algorithms. INN uses two neural networks for the two computational subdomains and applies an extended multiple-gradient descent (MGD) method in the training phase. This method utilizes information from multiple gradients to adjust and optimize the balance between different terms in the loss function. Intuitively, balancing



(a) Geometry of the dragon and gradient streamlines, colored by solution values.



(b) Jump in solution and its gradient are accurately captured by the surrogate neural network model.

Figure 3.8: The NBM approach enables a 1024^3 effective resolution on a single NVIDIA A6000 GPU. Once trained, it takes sub-milliseconds for the network to evaluate such a simulation that enables near-real-time digital twins for physical systems [9].

the weights of different loss terms from different subdomains in the INN method achieves a similar objective as the application of preconditioners in NBM, although NBM preconditions residual contributions from each grid point. Below we compare with example problem 4 of [157].

Linear Poisson-Boltzmann equation (LPBE) over a sphere

LPBE describes electrostatic potential around biomolecules solvated in low concentration ionic solvents. Here the computational domain is $\Omega \in [-2.5, 2.5]^3$ in dimensionless units. The solution contains a regular component and a singular component due to existence of a unit positive charge, $+e_c$, centered at $\mathbf{x}_c = (0, 0, 0)$. To solve LPBE decomposition techniques are usually employed, here we are only interested in the regular component that constitutes an interfacial problem given by

$$\begin{aligned} -\nabla \cdot (\mu^\pm \nabla u) + k^\pm u &= 0, & \mathbf{x} \in \Omega^\pm, \\ [u] &= g(\mathbf{x}), & \mathbf{x} \in \Gamma, \\ [\mu \partial_n u] &= \mu^- \partial_n g(\mathbf{x}), & \mathbf{x} \in \Gamma, \\ u &= \frac{\omega}{4\pi\mu^+} \frac{\exp(\kappa(\sigma - r))}{(1 + \kappa\sigma)r}, & \mathbf{x} \in \partial\Omega, \\ g(\mathbf{x}) &= \frac{\omega}{4\pi\mu^- \|\mathbf{x} - \mathbf{x}_c\|_2} \end{aligned}$$

where $\sigma = 1$ is the radius of the sphere, $\mu^- = 2$ and $\mu^+ = 80$ while $k^- = 0$ and $k^+ = \mu^+ \kappa^2$ with $\kappa = 1.0299 \times 10^{-3}$ and $\omega = 7.0465 \times 10^3$. The exact solution is then given by

$$u^- = \frac{\omega}{4\pi\sigma} \left(\frac{1}{\mu^+(1 + \kappa\sigma)} - \frac{1}{\mu^-} \right), \quad u^+ = \frac{\omega}{4\pi\mu^+} \frac{\exp(\kappa(\sigma - r))}{(1 + \kappa\sigma)r}$$

Noting that all equations are scaled by ω , we normalize the solution values to the range

$[-1.0, 1.0]$ by scaling ω by $\hat{\omega} = 293.6$, this is equivalent to setting $\omega = 24.0$ in the solver and solving for $\hat{u}^\pm = u^\pm/\hat{\omega}$ instead. This is to avoid the issues with scaling activation functions and having plausible numerical values close to unit value during training. Note that this trick does not impact the *relative* L_2 errors for comparison with results reported for INN. For fair comparison with INN we use exactly the same grid point generation code as in INN to generate $N_+ = 2000$, $N_- = 100$, $N_b = 1000$, and $N_\Gamma = 200$. We build small voxels centered on these grid points with side length of 0.00244 corresponding to 2048 resolution. We report error estimates in table 3.6 using mostly shallow multilayer perceptron (MLP) architectures as well as the ResNet architecture.

Analysis

Based on table 4 of [157], the smallest network architecture for INN is a pair of one-block ResNet networks with 20 hidden neurons per layer in Ω^- and 40 neurons in Ω^+ , this model produces a relative L_2 error of 4.1×10^{-2} which is on par with the shallow MLP in our experiments, yet NBM trains to this level of accuracy at a much smaller computational cost (see table 3.6 for size of our network parameters, timings, and memory requirement).

On the other hand, tables 4 and 8 of [157] report the best relative L_2 value achieved by INN being 3.675×10^{-4} using a pair of ResNets with 3 blocks each (*i.e.*, each ResNet block contains 2 hidden layers) with 40 neurons per hidden layer to learn the solution in Ω^- as well as 80 neurons per hidden layer for the solution in Ω^+ , with a total of 41,202 trainable parameters. Regarding validity of these values, we note that the errors reported by [157] are estimated on a relatively coarse grid of size $41 \times 41 \times 41$ that sparsely samples the error in the domain, in contrast our results are measured over a fine resolution of $256 \times 256 \times 256$ grid points to reliably quantify model accuracy in the space between the points that were used for training. For comparison, we implemented the exact settings

described here for the ResNet architecture and used the L-BFGS-B algorithm from the excellent `jaxopt` package [158], we also removed the $\hat{\omega}$ scaling, JAX-DIPS only takes 89 seconds to achieve a relative error of 2.22×10^{-2} using 3.69 GB of memory. Although the error is still lower than the ones achieved by INN, we emphasize this can improve by using state-of-the-art preconditioners, and also timings prove the efficiency of NBM that is on-par with and even superior to automatic differentiation based algorithms.

3.6 Discussion and future work

3.6.1 Spatial gradient calculation using finite discretization improves neural network regularity

NBM minimizes the sum of discretization residuals calculated over a set of grid voxels centered on unstructured grid points using any arbitrary finite discretization scheme. Essentially, the finite discretization scheme imposes specific spatial regularities to the neural network predictions inside the spatial volume enclosed by each grid voxel centered at each grid point. This adds an extra *spatial regularization* on the neural network predictions. In contrast to PINNs, computing loss using finite discretizations imposes a spatial regularity on the neural network predictions inside each finite volume; *i.e.*, neural network predictions at vertices and faces of the grid voxel are related to the voxel center according to finite discretization equations. This adds more spatial regularity to the trained neural network compared to automatic differentiation that computes gradients in completely local manner.

One important advantage of this extended *spatial support* for computing gradients is realized when using conservative finite discretizations such as the one in section 3.4.1. These explicit conservation laws help to constrain the neural network within each finite

volume. We believe this spatial regularization is an important mechanism to encode physical and mathematical priors into the training of neural networks. During training on a set of grid points the neural network not only learns the solution at each point, but also is informed about local symmetries within its neighborhood. As a result of spreading the residual probe according to finite discretization equations, the regularity of the network in the regions between training points follows mathematical regularities and symmetries that are encoded in the bootstrapped finite discretization scheme. In the case of interfacial PDEs with jump conditions considered here, these conservation laws are explicitly considered in section 3.4.1 that govern the solution flux across the interface given the jump conditions. We believe this *mathematically-informed spatial regularization* is responsible for explaining the improved regularity in errors and convergence rates observed in our numerical experiments of section 3.5.

Interestingly, the numerical results in section 3.5.5 demonstrate that even shallow multilayer perceptrons with a few hundreds of trainable parameters can in practice reach levels of accuracy that otherwise require complex architectures such as ResNets with tens of thousands of parameters using PINNs. A network’s ability to converge more accurately depends on the optimization strategy, because a simple MLP is a universal function approximator and therefore can in theory represent any function to any level of accuracy. Here we have proposed a new way for optimizing neural networks for PDEs that encodes solutions more efficiently. This is a result of more constrained training that underlies NBM residuals.

3.6.2 Training with only first order automatic differentiation improves training performance

Additionally, NBM distinguishes between spatial gradients of the neural network with respect to its inputs (needed for the PDE evaluation) from gradients of the neural network with respect to its internal parameters (needed for optimization). NBM computes spatial gradients using finite discretization schemes, while optimization of its internal parameters are handled by the AD-based first order optimizers. This removes the need for higher order AD computations over the computational graph of the neural network during training, therefore increasing scalability and decreasing memory requirements for training deeper and more complex neural network models with higher number of trainable parameters.

The numerical results presented in section 3.5 and comparisons of section 3.5.5 demonstrate the superior computational efficiency of NBM implemented in JAX-DIPS compared to PINN-like frameworks such as INN. Our results demonstrate that by removing the expensive higher order AD calculations we gain $\sim 10x$ speedup while maintaining minimal memory requirements (1 – 2 GB) for training neural network models.

3.6.3 Current shortcomings and future improvements

There are several algorithmic improvements and applications for the current work that we did not explore in this manuscript. The most important algorithmic improvement is application of more advanced preconditioners that can effectively improve the condition number of linear systems arising in interfacial PDE problems such as the ones considered here. Preconditioning the discretization residuals before applying optimization step is an essential requirement for enhancing accuracy of interfacial PDEs in finite discretization methods. The irregular geometries of interfaces as well as the jump conditions and dis-

continuous parameters of the environment lead to ill-conditioned linear systems. In the current work we only considered the Jacobi preconditioner which is the most basic preconditioner due to ease of implementation. We believe the most important improvement to the current work is application of more advanced preconditioners such as the algebraic multigrid preconditioner [159, 160]. The numerical methods considered in section 3.4.1 can in principle reach several orders of magnitudes better numerical accuracies when preconditioned properly, see [152, 118] for results of using algebraic preconditioners. These reported results also indicate the obtained convergence results in section 3.5 are not simply reflective of the accuracy limitation of the bootstrapped finite discretization method, but they are limited by the improper condition number of the residuals and suboptimal neural network models.

Adaptive mesh refinement is another important algorithmic upgrade to the current work. Adaptive grids with enhanced resolutions closer to the interface and coarser grid cells in the bulk play a significant role in both reducing the computational load as well as improving accuracy of fluxes across interfaces by increasing contributions to the total loss from points closer to the interface. Additionally, more expressive neural network architectures should be considered in JAX-DIPS by adding to the model class of the library. We only considered MLPs, however in recent years there have been a plethora of deep neural network models that have shown great promise such as FNO [144], DeepONet [141], and their numerous variants.

In terms of applications, our ultimate goal is training neural operators that can map from different geometries for discontinuities to the solution field. This is critical for developing near real-time simulations of time-evolving systems in digital twins for physical systems. NBM is applicable for physics-driven training of neural operators for elliptic problems with freely moving boundaries, we will present this work in a future time. Additionally, benefiting from differentiability of the solver we will explore utility of NBM

for solving inverse-PDE problems as well as parameterized PDEs.

3.7 Conclusion

We developed a differentiable multi-GPU framework for solving partial differential equations with jump conditions across irregular interfaces in three spatial dimensions. We developed the neural bootstrapping method to leverage existing finite discretization methods for optimization of neural network internal parameters, while explicitly calculating the spatial gradients using advanced finite discretization methods that encode symmetries and conservation laws governing the PDE solution and its flux across interfaces. Importantly, our framework only uses first order automatic differentiation for optimizing internal state of the neural networks, this technique provides an efficient alternative for training higher order PDE systems by avoiding computational challenges posed by higher order AD over deep neural networks. Moreover, NBM paves the path for obtaining more accurate neural network models of PDEs by leveraging numerical preconditioners by, intuitively, regularizing residuals computed on individual grid points, thus improving optimization gradients.

Acknowledgement

This work has been partially funded by ONR N00014-11-1-0027. We extend our sincere gratitude to the reviewers for their invaluable contributions and insightful feedback, which greatly enhanced the quality and rigor of our work.

Table 3.3: Convergence in solution of the star geometry using the single-resolution regression-based solver with domain switching. We report L^∞ -norm error as well as root-mean-squared-error (RMSE) of the solution field evaluated everywhere in the domain. Timings are averaged over 10,000 epochs in each case and include the initial compilation time for jaxpressions. The neural network pair have 1 hidden layer each with 100 neurons, overall 1,002 trainable parameters. Domain switching scheme follows the whole region \rightarrow fast region \rightarrow fast region sequence.

$N_{x,y,z}$	RMSE		L^∞		GPU Statistics	
	Solution	Order	Solution	Order	t (sec/epoch)	VRAM (GB)
regress ∂_n						
2^3	1.36×10^{-1}	-	1.27	-	0.019	0.98
2^4	7.98×10^{-2}	0.77	8.23×10^{-1}	0.63	0.022	1.01
2^5	4.36×10^{-2}	0.87	3.85×10^{-1}	1.10	0.032	1.30
2^6	2.43×10^{-2}	0.84	2.28×10^{-1}	0.76	0.200	3.7
neural ∂_n						
2^3	2.17×10^{-1}	-	2.89	-	0.0259	0.93
2^4	1.34×10^{-1}	0.70	1.66	0.80	0.0408	1.19
2^5	5.68×10^{-2}	1.24	8.17×10^{-1}	1.02	0.0712	2.96
2^6	2.77×10^{-2}	1.03	3.94×10^{-1}	1.05	0.334	13.6

Table 3.4: Convergence in solution of the star geometry using the multi-resolution regression-based solver with batching. We use a multi-resolution training protocol that refines to 4 levels at each collocation point, this slightly improves accuracies although in the current version of JAX-DIPS (v0.0.1) the memory requirement increases. Batch size is the minimum of $64 \times 64 \times 32$ and number of collocation points, which ensures memory saturation at 30 GB.

regress ∂_n	RMSE		L^∞		GPU Statistics	
$N_{x,y,z}$	Solution	Order	Solution	Order	t (sec/epoch)	VRAM (GB)
2^3	1.05×10^{-1}	-	1.29	-	0.0225	1.27
2^4	5.52×10^{-2}	0.93	6.22×10^{-1}	1.05	0.0411	1.27
2^5	2.44×10^{-2}	1.18	2.66×10^{-1}	1.23	0.1814	8.3
2^6	2.33×10^{-2}	0.07	2.24×10^{-1}	0.25	1.889	29.6
2^7	8.62×10^{-2}	-1.88	3.80×10^{-1}	-0.76	9.649	29.7

Table 3.5: Scaling test. Time per epoch (sec) and JAX compile time for different configurations.

base resolution:	64 ³		128 ³		256 ³		512 ³	
A100 GPUs	epoch	compile	epoch	compile	epoch	compile	epoch	compile
1	0.908	9.027	6.960	9.288	55.287	12.164	438.45	49.020
2	0.657	7.575	5.893	7.823	47.360	10.045	378.98	39.815
4	0.405	7.480	3.629	7.863	28.261	9.129	226.73	27.405
8	0.384	7.983	3.340	7.901	26.799	9.154	204.88	20.632

Table 3.6: Convergence test of example 3.5.5, positive region uses `CeLU` activations while negative region uses `tanh` activations. Note that in all cases there exists an output linear layer. Measurements are on a single NVIDIA A6000 GPU. Errors are measured on a $256 \times 256 \times 256$ grid to sample the space far from training points. These results are obtained using the Jacobi preconditioner, Adam and L-BFGS-B optimizers, and the discretization of 3.4.1.

type	model - +	#hidden layer:#hidden unit	#params	Accuracy			Training Statistics			
				RMSE	L^∞	Rel. L^2	epochs	time (sec)	Mem. (GB)	optimizer
using INN grid with $N_+ = 2000, N_- = 100, N_b = 1000, N_T = 200$										
MLP	1 : 1 2 : 10	167	3.27×10^{-3}	1.22×10^{-2}	1.92×10^{-2}	50,000	693	1.7	Adam@optax	
MLP	1 : 1 2 : 10	167	3.50×10^{-3}	1.28×10^{-2}	2.15×10^{-2}	50,000	95	1.1	L-BFGS-B@jaxopt	
MLP	1 : 10 2 : 10	212	3.98×10^{-3}	2.13×10^{-2}	2.34×10^{-2}	50,000	738	1.8	Adam@optax	
MLP	1 : 10 2 : 10	212	3.43×10^{-3}	1.29×10^{-2}	2.10×10^{-2}	50,000	120	1.17	L-BFGS-B@jaxopt	
MLP	2 : 10 2 : 10	322	3.83×10^{-3}	1.55×10^{-2}	2.25×10^{-2}	50,000	784	1.9	Adam@optax	
MLP	2 : 10 2 : 10	322	3.52×10^{-3}	1.31×10^{-2}	2.16×10^{-2}	50,000	96	1.2	L-BFGS-B@jaxopt	
using INN grid with $N_+ = 2000, N_- = 100, N_b = 1000, N_T = 1000$										
MLP	0 : 1 2 : 10	165	4.65×10^{-3}	1.53×10^{-2}	2.73×10^{-2}	50,000	923	1.78	Adam@optax	
MLP	0 : 1 2 : 10	165	3.52×10^{-3}	1.29×10^{-2}	2.16×10^{-2}	50,000	92	1.15	L-BFGS-B@jaxopt	
MLP	1 : 1 2 : 10	167	1.63×10^{-3}	4.20×10^{-3}	9.62×10^{-3}	50,000	946	1.80	Adam@optax	
MLP	1 : 1 2 : 10	167	3.53×10^{-3}	1.28×10^{-3}	2.17×10^{-2}	50,000	99	1.18	L-BFGS-B@jaxopt	
MLP	1 : 10 2 : 10	212	3.80×10^{-3}	1.47×10^{-2}	2.22×10^{-2}	50,000	977	1.85	Adam@optax	
MLP	1 : 10 2 : 10	212	3.51×10^{-3}	1.31×10^{-2}	2.14×10^{-2}	50,000	111	1.18	L-BFGS-B@jaxopt	
MLP	2 : 10 2 : 10	322	3.72×10^{-3}	1.37×10^{-2}	2.19×10^{-2}	50,000	1031	1.93	Adam@optax	
MLP	2 : 10 2 : 10	322	3.49×10^{-3}	1.29×10^{-2}	2.14×10^{-2}	50,000	126	1.22	L-BFGS-B@jaxopt	
MLP	5 : 40 5 : 40	13,522	3.54×10^{-3}	1.30×10^{-2}	2.17×10^{-2}	50,000	80	3.34	L-BFGS-B@jaxopt	
ResNet	3 : 80 3 : 40	41,202	—	—	2.22×10^{-2}	50,000	89*	3.69	L-BFGS-B@jaxopt	
uniform $32 \times 32 \times 32$ grid										
MLP	1 : 1 2 : 10	167	3.15×10^{-3}	1.3×10^{-2}	1.85×10^{-2}	10,000	794	3.15	Adam@optax	
MLP	2 : 10 2 : 10	322	2.99×10^{-3}	1.25×10^{-2}	1.83×10^{-2}	50,000	160	2.96	L-BFGS-B@jaxopt	

Chapter 4

Pharmacology-Informed Neural-SDE

4.1 abstract

Digital health technologies (DHT), such as wearable devices, provide personalized, continuous, and real-time monitoring of patient. These technologies are contributing to the development of novel therapies and personalized medicine. Gaining insight from these technologies requires appropriate modeling techniques to capture clinically-relevant changes in disease state. The data generated from these devices is characterized by being stochastic in nature, may have missing elements, and exhibits considerable inter-individual variability - thereby making it difficult to analyze using traditional longitudinal modeling techniques. We present a novel pharmacology-informed neural stochastic differential equation (SDE) model capable of addressing these challenges. Using synthetic data, we demonstrate that our approach is effective in identifying treatment effects and learning causal relationships from stochastic data, thereby enabling counterfactual simulation.

4.2 Introduction

The rise of digital health technologies (DHT) including wearable devices such as smart watch and patch based physiological sensors has opened new possibilities for continuous patient monitoring [161] and enables generation of time-series data at an unprecedented temporal resolution and duration, thereby offering the potential to generate new clinical measures and insights [162]. Furthermore, recent examples have shown the clinical value in modeling both the longitudinal trends as well as the stochasticity in digital health (DH) data [163].

Stochastic differential equations (SDEs) have been developed to describe various phenomena that exhibits random fluctuations [164], including in biological and biomedical applications [165, 166]. In the context of DH, the interplay between physiology and the measurement device is likely far too complex for one to theoretically derive the equations underlying the link between disease status and DH data from first principles. Instead, we propose to learn the underlying dynamical system directly from data, with the help of neural-SDE [167, 168].

Here, we develop a *pharmacology-informed* [169, 170] neural-SDE that:

- learns the underlying dynamical system from a patient population, while introducing patient-dependent parameters that enables the characterization of patient-to-patient variability;
- incorporates the causality between pharmacokinetics (PK) and pharmacodynamics (PD);
- enables counterfactual simulations to describe drug effects at the individual patient level.

We demonstrate the effectiveness of the proposed model using synthetic data.

4.3 Methods

4.3.1 Neural-SDE Model

We assume that the longitudinal data are modelled by a system of equations of the form,

$$dc_t = f(c_t)dt \quad (4.1)$$

$$dx_t = \nu(x_t, c_t, \mathbf{p})dt + \sigma(x_t, c_t, \mathbf{p})dW_t \quad (4.2)$$

where Equation 4.1 represents a known Ordinary Differential Equation (ODE) model with $f(\cdot)$ being the vector field for PK that governs the drug concentration, $c_t \in \mathbb{R}$, and where the drift and diffusion terms (i.e., $\nu(x_t, c_t, \mathbf{p})$ and $\sigma(x_t, c_t, \mathbf{p})$ respectively) are described by neural networks. We work under the hypothesis that the drift and diffusivity terms of the *effective* SDE, are dependent on the state ($x_t \in \mathbb{R}$) as well as the drug concentration c_t . Additionally, while the underlying equations are the same for all patients, the model includes a *latent* patient-dependent parameter vector \mathbf{p} that describes the patient-to-patient variability. This *latent* parameter \mathbf{p} is discovered in a data-driven way based on the work of [169], which we elaborate below.

While the available data are in the form of trajectories, we transform them to snapshots \mathcal{D} in a manner analogous to that done in [168]. In particular, each snapshot \mathcal{D}^i , uniquely identified by the index i , takes the form $\mathcal{D}^i = \{x_1^i, x_0^i, \Delta t, c_1^i, \mathbf{p}^{i,j}\}$, where x_1^i is the evolution of the state variable x_t after a time step Δt given the initial condition x_0^i ; $\mathbf{p}^{i,j}$ is the *latent* parameter for the j th patient. Note that we utilize the concentration at c_1 and not at c_0 following the (symplectic) Euler-Maryama scheme discussed in [168]. The concentration c_t and the patient dependent parameter \mathbf{p} enter into the overall architecture as inputs based on [168, 167].

The construction of the loss function (based on [168]) is derived from the numerical integration scheme (symplectic) Euler-Maruyama. The numerical approximation of Equations 4.1 and 4.2 results in:

$$c_1^i = c_0^i + f(c_0^i)\Delta t \quad (4.3)$$

$$x_1^i = x_0^i + \nu(x_0^i, c_1^i, \mathbf{p}^{i,j})\Delta t + \sigma(x_0^i, c_1^i, \mathbf{p}^{i,j})\delta W_0, \quad (4.4)$$

where δW_0 is normally distributed around zero and Δt is a variable timestep. The drift and diffusivity terms are approximated by two networks ν_θ and σ_θ , under the assumption that x_1 is drawn from a normal distribution of the form,

$$x_1^i \sim \mathcal{N}(x_0^i + \nu_\theta(x_0^i, c_1^i, \mathbf{p}^{i,j})\Delta t, \sigma_\theta(x_0^i, c_1^i, \mathbf{p}^{i,j})^2\Delta t). \quad (4.5)$$

With the assumed mean and variance in Equation 4.5 for the drift and diffusivity, we can compute the logarithm of the resulting normal distribution and derive the following loss function that maximizes the likelihood:

$$\mathcal{L}(\theta|x_0^i, x_1^i, \Delta t) := \frac{(x_1^i - x_0^i - \nu_\theta(x_0^i, c_1^i, \mathbf{p}^{i,j}))^2}{\Delta t \sigma_\theta(x_0^i, c_1^i, \mathbf{p}^{i,j})^2} + \log|\Delta t \sigma_\theta(x_0^i, c_1^i, \mathbf{p}^{i,j})^2|. \quad (4.6)$$

It should be noted that the Neural-SDE framework by [168] is also capable of handling varying time steps Δt .

The Neural-SDE architecture consists of two network components for the drift and diffusion models. In our work, the drift network consists of 4 layers where each layer has 64 neurons each followed by ELU activation function. The diffusion network consists of 3 layers with 32 neurons, the first two layers are followed by ELU activation function and the output layer is followed by `softplus` activation function. A schematic of the Neural-SDE architecture is shown in Figure 4.1.

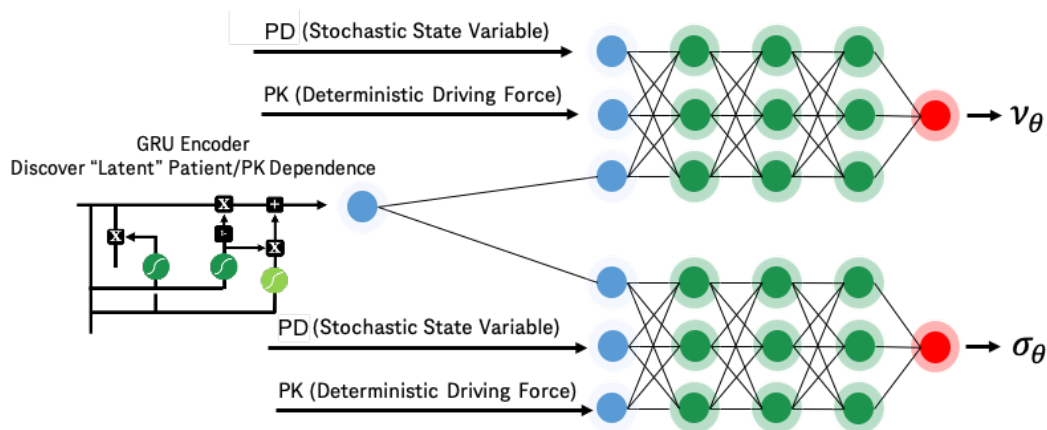


Figure 4.1: The Neural-SDE architecture including the GRU encoder.

4.3.2 Latent Patient Descriptors - GRU Encoder

Our approach to learning the Neural-SDE from data across the patient population is to identify a set of dynamical equations that holds across all patients, as well as patient-specific descriptors (or embedding) that characterize patient-to-patient variability [170]. In our approach, those patient-specific descriptors are discovered in a data-driven manner, based on the work of [169]: a Gated Recurrent Unit (GRU) encoder was used to discover the latent parameter \mathbf{p} , with longitudinal data provided in a tabular form as an input. More specifically, the input data entering the encoder consist of variable number of rows for each patient and the following four columns: (1) the absolute time; (2) the time after dose; (3) the stochastic PD data (4) the deterministic PK data.

Each tabular input was *padded* and *masking* was applied in order to handle the variable time points. The GRU encoder has 128 hidden states and is connected to a Multilayer Perceptron (MLP) consisting of 2 layers, each with 128 neurons, both followed by ELU activation function. The output of MLP is the *latent* parameter \mathbf{p} that enters the Neural-SDE architecture. An end-to-end training was implemented by using the loss function given by Equation 4.6.

4.3.3 Dataset

To mimic clinical digital health measurements, synthetic data was simulated in which the PK serves as a deterministic driving input that causally influences a stochastically evolving PD. Patient specific parameters were sampled from a log-normal distribution: 50 individual patient trajectories were sampled across 3 different dose levels (50 mg, 100 mg, 400 mg) for a total of 150 patient trajectories and 70:30 train-test split was used.

Synthetic training data was generated to represent a indirect response PK-PD model [171] by which PK acts causally to change the PD, with the additional modification that the observable PD variable is stochastic in nature. This system follows the general form of Equations 4.1 and 4.2,

with the following system of ODEs being specified for the term $f(c_t, \mathbf{p})dt$:

$$\frac{du_1}{dt} = -KA \times u_1(t) \quad (4.7)$$

$$\frac{du_2}{dt} = KA \times u_1(t) - u_2(t) \times (KE + K12) + u_3(t) \times K21 \quad (4.8)$$

$$\frac{du_3}{dt} = K12 \times u_2(t) - K21 \times u_3(t) \quad (4.9)$$

where $c_t = u_2(t)/V2$ with $V2$ representing the volume of distribution for drug in plasma circulation. The drift term in the relationship between c_t and PD is represented by the following:

$$\frac{du_4}{dt} = KIN - (KOUT * (1 - (Imax \times c_t / IC50 + c_t))) \times u_4(t). \quad (4.10)$$

Example trajectories of this system are shown in Figure 4.2. The diffusion term in Equation 4.2 is described by the following $\beta u_4 dW_t$, where β was sampled from a log-normal distribution. Examples of stochastic trajectories for c_t are shown in Figure 4.3.

In the current set of experiments, an equal number of patients were simulation for

a range of doses (50, 100, 400 mg). Dosing was set to begin at day 5 for all synthetic subjects with daily dosing; the PD sampling frequency is once per hour, over a period of 30 days.

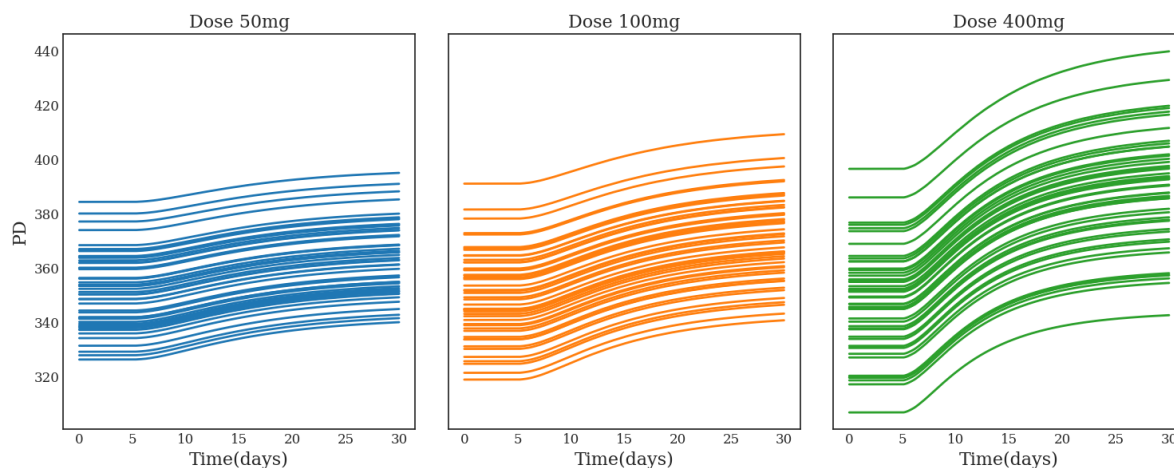


Figure 4.2: Synthetic data trajectories without the diffusivity component under different simulated doses.

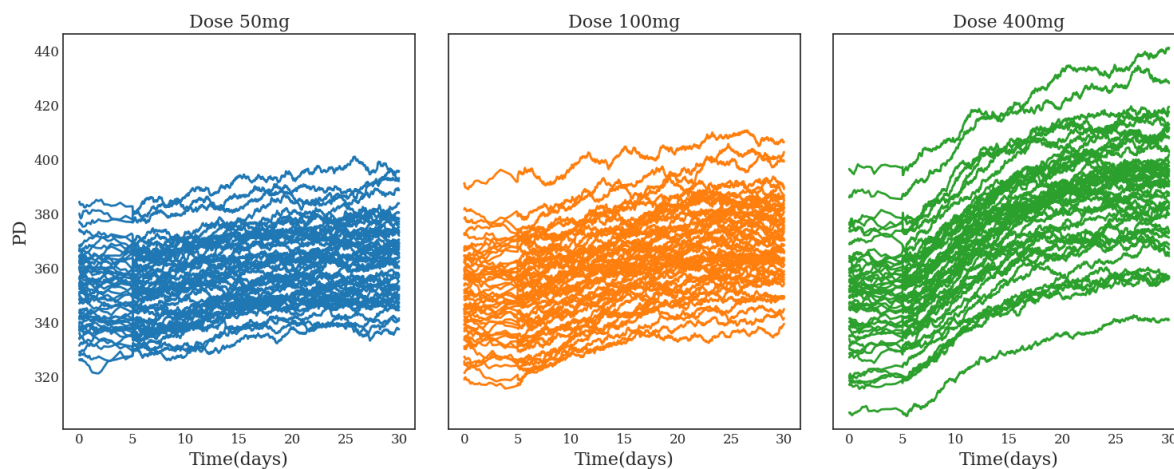


Figure 4.3: Synthetic data trajectories under different doses.

4.3.4 Training methodology and optimization strategy

The current model, including the numerical integration scheme which employs a Euler-Maruyama solver, have been implemented in PyTorch. While a higher-order meth-

ods were not used in this current work, it remains open for future development based on specific needs.

In model training, we leveraged vectorization rather than operating on a single value at a time whereby the model processes each time-step for each patient sequentially. In this way, the model operates at a patient level, concurrently processing all data points associated with a specific patient. This is feasible based on the observation that evaluating the loss function given in Equation 4.6 at each time-step is independent from other time instances. The vectorization strategy significantly enhances the training and inference performance.

We trained the network for 100 epochs using the ADAM optimizer with learning rate 0.001 and batch size of 1. The overall training process takes around 140 seconds using one NVIDIA V100 GPU.

4.4 Results

Figure 4.4 demonstrates the model’s ability to learn the underlying system’s dynamics by comparing “true” (i.e., the underlying ground truth) SDE trajectories from the test dataset against the model predicted trajectories. For each patient in the test set, we sampled 250 trajectories to provide a robust representation of the predictive variability associated with the model. This result demonstrates the model’s ability in replicating the complex dynamics of PD trajectories at the population level.

4.4.1 Dosing regimen analysis

To analyze the impact of different dosing regimens on PD, we consider three distinct simulated doses at 50 mg, 100 mg, and 400 mg. For each patient from the test dataset, we sampled 250 SDE trajectories. Figure 4.5 shows the the model is qualitatively able

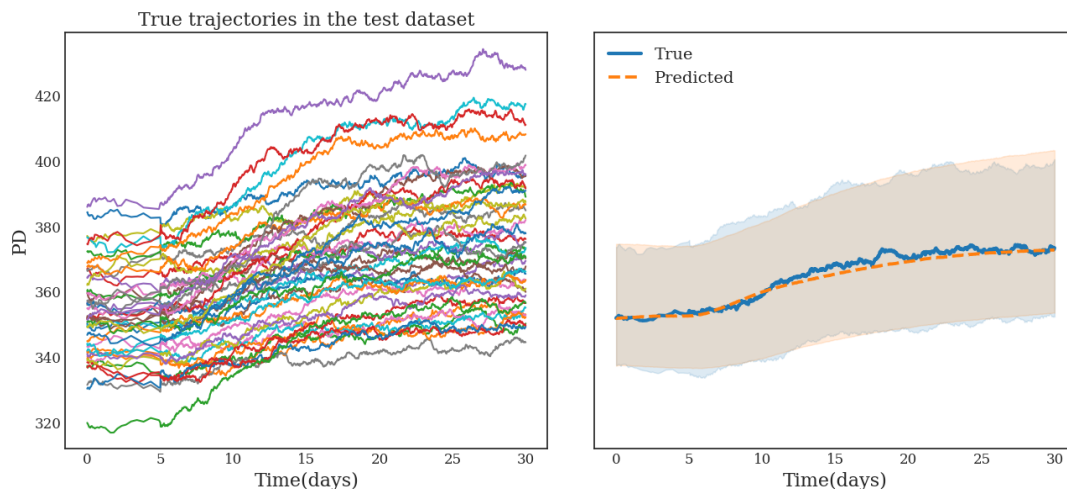


Figure 4.4: Comparison of the true and predicted SDE trajectories in the test dataset. Left panel: the colored lines represent the observed stochastic trajectories in the test data. Right panel: blue line and shaded region represent the median and the 10th to 90th percentile respectively of the ground truth trajectories; similarly, the orange lines and shade region represent those from the model.

to capture the true underlying dose response relationship.

4.4.2 Patient-specific responses and counterfactual analysis

Figure 4.6 demonstrates the proposed methodology’s ability to perform counterfactual analysis and identify individual treatment effects. To accomplish this, for each patient the drift and diffusivity terms were inferred from the trained model and 250 SDE trajectories were generated. The results demonstrate the model’s ability to capture the underlying dynamics of the stochastic process for individual patients. This suggests that the GRU encoding strategy not only captures the population behaviors, but also successfully learns to differentiate amongst patients. Moreover, we demonstrate a what-if scenario: in the absence of PK, the model correctly predicts a lack of dynamical change in the modeled PD endpoint. This suggests our model is able to correctly identify the causal relationship between PK and PD.

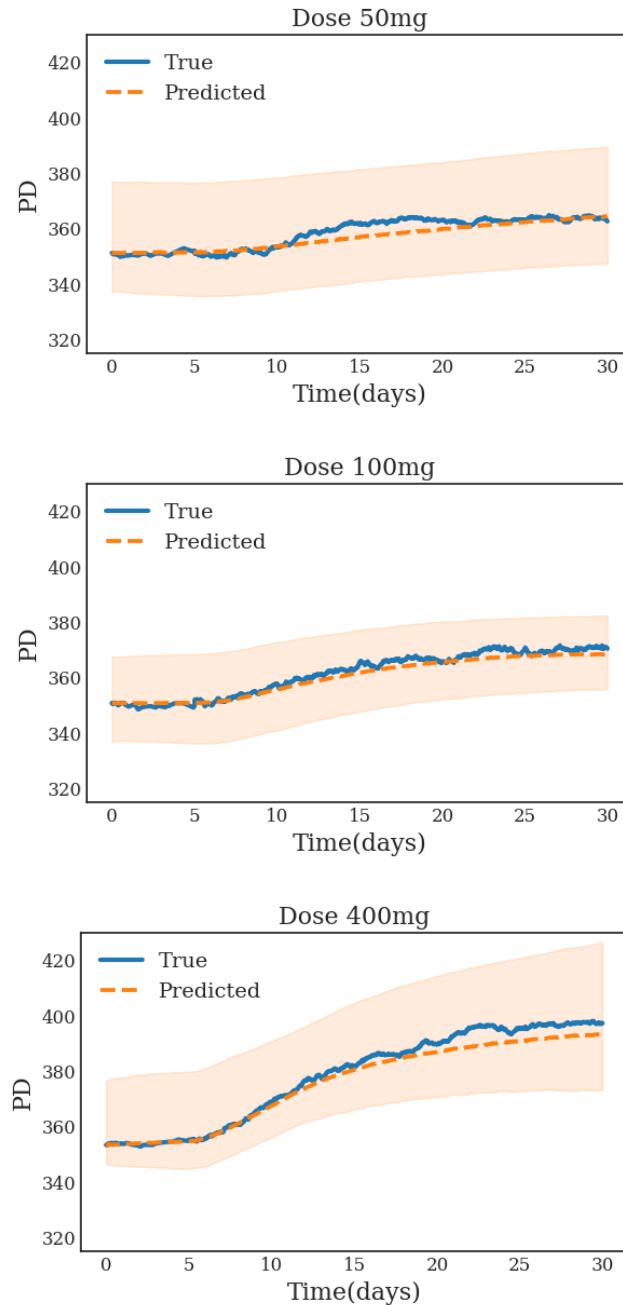


Figure 4.5: Comparison of the true and predicted SDE trajectories in the test dataset for 50, 100 and 400 mg doses. Blue lines represent the median of the ground truth trajectories; orange dashed lines and shaded regions represent median and the 10th to 90th percentile of trajectories from the model.

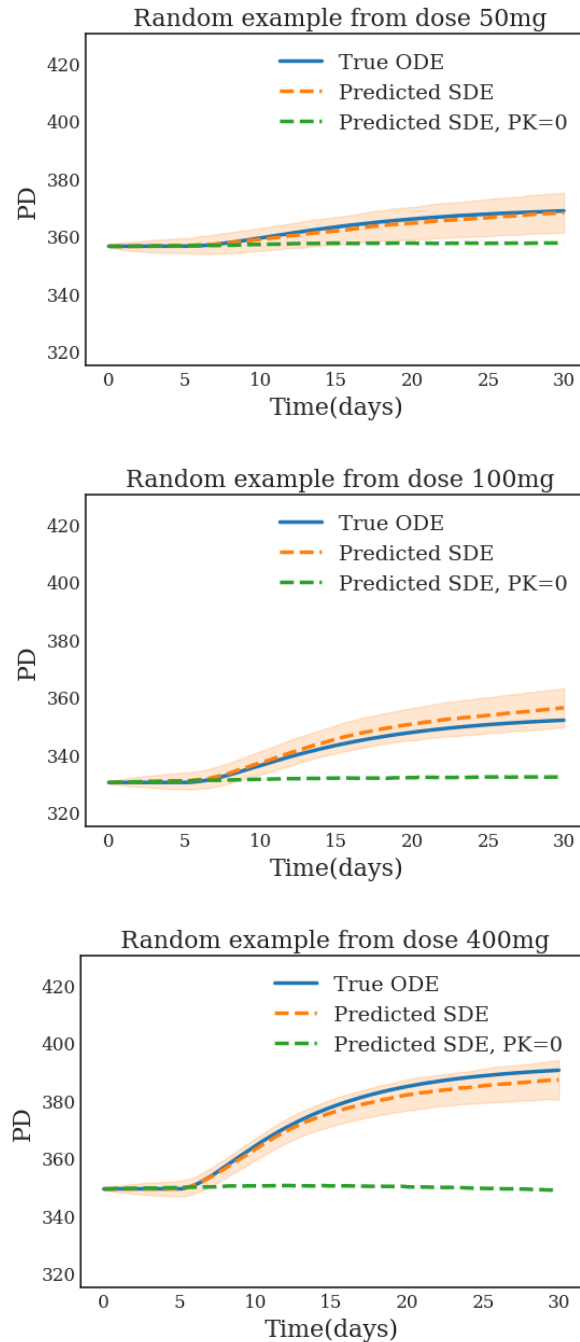


Figure 4.6: Patient-specific trajectories and counterfactual simulations. Each subplot represents a random patient from the respective dosages. The solid blue line represents the true drift; the orange dashed line and shaded region represent the mean and mean \pm standard deviation (std) of 250 posterior samples; the green dashed lines represent counterfactual simulations assuming no dosing (*i.e.*, $PK = 0$).

4.5 Conclusion

We proposed a pharmacology-informed neural-SDE architecture that is able learn the relationship between a deterministic PK and stochastic PD. Using synthetic data, the model correctly reproduces the underlying PK-PD relationship at the population level. Furthermore, the model enables the counterfactual simulation of PD in the absence of the hypothetical drug - and in doing so, quantify the individual treatment effect.

Bibliography

- [1] P. A. Mistani, S. Pakravan, R. Ilango, and F. Gibou, *Jax-dips: neural bootstrapping of finite discretization methods and application to elliptic problems with discontinuities*, *Journal of Computational Physics* **493** (2023) 112480.
- [2] S. Pakravan, P. A. Mistani, M. A. Aragon-Calvo, and F. Gibou, *Solving inverse-pde problems with physics-aware neural networks*, *Journal of Computational Physics* **440** (2021) 110414.
- [3] P. A. Mistani, S. Pakravan, and F. G. Gibou, *A fractional stochastic theory for interfacial polarization of cell aggregates*, 2020.
- [4] S. Pakravan, N. Evangelou, M. Usdin, L. Brooks, and J. Lu, *From noise to signal: Unveiling treatment effects from digital health data through pharmacology-informed neural-sde*, 2024.
- [5] P. Mistani, S. Pakravan, R. Ilango, S. Choudhry, and F. Gibou, *Neuro-symbolic partial differential equation solver*, *arXiv preprint arXiv:2210.14907* (2022).
- [6] P. Mistani, S. Pakravan, and F. Gibou, *Towards a tensor network representation of complex systems*, *Sustainable Interdependent Networks II: From Smart Power Grids to Intelligent Transportation Networks* (2019) 69–85.
- [7] P. Mistani, S. Pakravan, and F. Gibou, *Tensor network renormalization as an ultra-calculus for complex system dynamics*, *Sustainable Interdependent Networks II: From Smart Power Grids to Intelligent Transportation Networks* (2019) 87–106.
- [8] D. Bochkov, T. Pollock, and F. Gibou, *A numerical method for sharp-interface simulations of multicomponent alloy solidification*, *Journal of Computational Physics* **494** (2023) 112494.
- [9] T. Müller, *tiny-cuda-nn*, 4, 2021.
- [10] C. L. Epstein, *Introduction to the mathematics of medical imaging*. SIAM, 2007.
- [11] F. Natterer, *The mathematics of computerized tomography*. SIAM, 2001.

- [12] R. J. van Sloun, R. Cohen, and Y. C. Eldar, *Deep learning in ultrasound imaging*, *Proceedings of the IEEE* (2019).
- [13] K. H. Jin, M. T. McCann, E. Froustey, and M. Unser, *Deep convolutional neural network for inverse problems in imaging*, *IEEE Transactions on Image Processing* **26** (2017), no. 9 4509–4522.
- [14] G. P. J.-P. Fouque, J. Garnier and K. Solna, *Wave Propagation and Time Reversal in Randomly Layered Media*. Springer, 2007.
- [15] K. Samsami, S. A. Mirbagheri, F. Meshkati, and H. C. Fu, *Stability of soft magnetic helical microrobots*, *Fluids* **5** (2020), no. 1 19.
- [16] M. Hedayatpour, M. Mehrandezh, and F. Janabi-Sharifi, *A unified approach to configuration-based dynamic analysis of quadcopters for optimal stability*, in *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pp. 5116–5121, IEEE, 2017.
- [17] M. Hedayatpour, M. Mehrandezh, and F. Janabi-Sharifi, *Precision modeling and optimally-safe design of quadcopters for controlled crash landing in case of rotor failure*, in *2019 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pp. 5206–5211, IEEE, 2019.
- [18] A. Zupanic, B. Kos, and D. Miklavcic, *Treatment planning of electroporation-based medical interventions: electrochemotherapy, gene electrotransfer and irreversible electroporation*, *Physics in Medicine & Biology* **57** (2012), no. 17 5425.
- [19] P. Mistani, A. Guittet, C. Poinard, and F. Gibou, *A parallel voronoi-based approach for mesoscale simulations of cell aggregate electroporation*, *Journal of Computational Physics* **380** (2019) 48–64.
- [20] P. Mistani, A. Guittet, D. Bochkov, J. Schneider, D. Margetis, C. Ratsch, and F. Gibou, *The island dynamics model on parallel quadtree grids*, *Journal of Computational Physics* **361** (2018) 150–166.
- [21] F. Gibou, D. Hyde, and R. Fedkiw, *Sharp interface approaches and deep learning techniques for multiphase flows*, *Journal of Computational Physics* **380** (2019) 442 – 463.
- [22] F. Gibou, R. Fedkiw, and S. Osher, *A review of level-set methods and some recent applications*, *Journal of Computational Physics* **353** (2018) 82–109.
- [23] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, *et. al.*, *Tensorflow: Large-scale machine learning on heterogeneous distributed systems*, *arXiv preprint arXiv:1603.04467* (2016).

- [24] S. He, Y. Li, Y. Feng, S. Ho, S. Ravanbakhsh, W. Chen, and B. Póczos, *Learning to predict the cosmological structure formation*, *Proceedings of the National Academy of Sciences* (2019) 201821458.
- [25] X. Zhang, Y. Wang, W. Zhang, Y. Sun, S. He, G. Contardo, F. Villaescusa-Navarro, and S. Ho, *From dark matter to galaxies with convolutional networks*, *arXiv preprint arXiv:1902.05965* (2019).
- [26] J. Zamudio-Fernandez, A. Okan, F. Villaescusa-Navarro, S. Bilaloglu, A. D. Cengiz, S. He, L. P. Levasseur, and S. Ho, *HIGAN: Cosmic neutral hydrogen with generative adversarial networks*, *arXiv preprint arXiv:1904.12846* (2019).
- [27] A. Chandrasekaran, D. Kamal, R. Batra, C. Kim, L. Chen, and R. Ramprasad, *Solving the electronic structure problem with machine learning*, *npj Computational Materials* **5** (2019), no. 1 22.
- [28] A. V. Sinititskiy and V. S. Pande, *Deep neural network computes electron densities and energies of a large set of organic molecules faster than density functional theory (DFT)*, *arXiv preprint arXiv:1809.02723* (2018).
- [29] M. Raissi, P. Perdikaris, and G. E. Karniadakis, *Machine learning of linear differential equations using gaussian processes*, *Journal of Computational Physics* **348** (2017) 683 – 693.
- [30] J. Berg and K. Nyström, *Data-driven discovery of PDEs in complex datasets*, *Journal of Computational Physics* **384** (2019) 239–252.
- [31] Z. Long, Y. Lu, X. Ma, and B. Dong, *Pde-net: Learning pdes from data*, in *International Conference on Machine Learning*, pp. 3208–3216, PMLR, 2018.
- [32] H. Schaeffer, *Learning partial differential equations via data discovery and sparse optimization*, *Proceedings of the Royal Society A: Mathematical, Physical and Engineering Sciences* **473** (2017), no. 2197 20160446.
- [33] H. Owhadi and C. Scovel, *Operator-Adapted Wavelets, Fast Solvers, and Numerical Homogenization: From a Game Theoretic Approach to Numerical Approximation and Algorithm Design*, vol. 35. Cambridge University Press, 2019.
- [34] H. Owhadi, C. Scovel, and F. Schäfer, *Statistical numerical approximation*, .
- [35] M. Raissi and G. E. Karniadakis, *Hidden physics models: Machine learning of nonlinear partial differential equations*, *Journal of Computational Physics* **357** (2018) 125 – 141.
- [36] P. Stinis, T. Hagge, A. M. Tartakovsky, and E. Yeung, *Enforcing constraints for interpolation and extrapolation in generative adversarial networks*, *Journal of Computational Physics* **397** (2019) 108844.

- [37] J. Ling, R. Jones, and J. Templeton, *Machine learning strategies for systems with invariance properties*, *Journal of Computational Physics* **318** (2016) 22 – 35.
- [38] Z. Geng, D. Johnson, and R. Fedkiw, *Coercing machine learning to output physically accurate results*, *Journal of Computational Physics* (2019) 109099.
- [39] M. A. Aragon-Calvo, *Self-supervised learning with physics-aware neural networks i: Galaxy model fitting*, *arXiv preprint arXiv:1907.03957* (2019).
- [40] M. Raissi, P. Perdikaris, and G. E. Karniadakis, *Physics informed deep learning (part i): Data-driven solutions of nonlinear partial differential equations*, *arXiv preprint arXiv:1711.10561* (2017).
- [41] I. E. Lagaris, A. Likas, and D. I. Fotiadis, *Artificial neural networks for solving ordinary and partial differential equations*, *IEEE transactions on neural networks* **9** (1998), no. 5 987–1000.
- [42] H. Owhadi, *Bayesian numerical homogenization*, *Multiscale Modeling & Simulation* **13** (2015), no. 3 812–828.
- [43] H. Owhadi, *Calculus for the optimal quantification of uncertainties*, Presented at the 2015 SIAM conference in Computational Science and Engineering, Salt Lake City, UT, 2015.
- [44] H. Owhadi, *Multigrid with rough coefficients and multiresolution operator decomposition from hierarchical information games*, *SIAM Review* **59** (2017), no. 1 99–149.
- [45] M. Raissi, P. Perdikaris, and G. E. Karniadakis, *Physics informed deep learning (part ii): Data-driven discovery of nonlinear partial differential equations*, *ArXiv abs/1711.10566* (2017).
- [46] J. Sirignano and K. Spiliopoulos, *DGM: A deep learning algorithm for solving partial differential equations*, *Journal of Computational Physics* **375** (2018) 1339 – 1364.
- [47] L. Bar and N. Sochen, *Unsupervised deep learning algorithm for PDE-based forward and inverse problems*, *arXiv preprint arXiv:1904.05417* (2019).
- [48] K. Xu and E. Darve, *The neural network approach to inverse problems in differential equations*, *arXiv preprint arXiv:1901.07758* (2019).
- [49] Z. Long, Y. Lu, and B. Dong, *PDE-Net 2.0: Learning PDEs from data with a numeric-symbolic hybrid deep network*, *Journal of Computational Physics* **399** (2019) 108925.

- [50] J. Berg and K. Nyström, *Neural network augmented inverse problems for PDEs*, *arXiv preprint arXiv:1712.09685* (2017).
- [51] N. Dal Santo, S. DeParis, and L. Pegolotti, *Data driven approximation of parametrized PDEs by reduced basis and neural networks*, *Journal of Computational Physics* (2020) 109550.
- [52] P. Y. Lu, S. Kim, and M. Soljačić, *Extracting interpretable physical parameters from spatiotemporal systems using unsupervised learning*, *Physical Review X* **10** (2020), no. 3 031056.
- [53] J. Valentin, C. Keskin, P. Pidlypenskyi, A. Makadia, A. Sud, and S. Bouaziz, “Tensorflow graphics.” Available at: <https://github.com/tensorflow/graphics>, 2019.
- [54] V. Tikhomirov, *On the representation of continuous functions of several variables as superpositions of continuous functions of one variable and addition*, in *Selected Works of AN Kolmogorov*, pp. 383–387. Springer, 1991.
- [55] G. Cybenko, *Approximation by superpositions of a sigmoidal function*, *Mathematics of control, signals and systems* **2** (1989), no. 4 303–314.
- [56] B. C. Csáji *et. al.*, *Approximation with artificial neural networks*, *Faculty of Sciences, Eötvös Lornd University, Hungary* **24** (2001), no. 48 7.
- [57] J. Darbon, G. P. Langlois, and T. Meng, *Overcoming the curse of dimensionality for some Hamilton–Jacobi partial differential equations via neural network architectures*, *Research in the Mathematical Sciences* **7** (2020), no. 3 1–50.
- [58] J. Han, A. Jentzen, and E. Weinan, *Solving high-dimensional partial differential equations using deep learning*, *Proceedings of the National Academy of Sciences* **115** (2018), no. 34 8505–8510.
- [59] Y. LeCun, L. Bottou, Y. Bengio, P. Haffner, *et. al.*, *Gradient-based learning applied to document recognition*, *Proceedings of the IEEE* **86** (1998), no. 11 2278–2324.
- [60] A. Krizhevsky, I. Sutskever, and G. E. Hinton, *ImageNet classification with deep convolutional neural networks*, in *Advances in neural information processing systems*, pp. 1097–1105, 2012.
- [61] S. Hochreiter and J. Schmidhuber, *Long short-term memory*, *Neural computation* **9** (1997), no. 8 1735–1780.
- [62] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, *Dropout: a simple way to prevent neural networks from overfitting*, *The journal of machine learning research* **15** (2014), no. 1 1929–1958.

- [63] F. Chollet *et. al.*, *Keras*, 2015.
- [64] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer, *Automatic differentiation in PyTorch*, .
- [65] A. Khotanzad and Y. H. Hong, *Invariant image recognition by zernike moments*, *IEEE Transactions on pattern analysis and machine intelligence* **12** (1990), no. 5 489–497.
- [66] S. O. Belkasim, M. Shridhar, and M. Ahmadi, *Pattern recognition with moment invariants: a comparative study and new results*, *Pattern recognition* **24** (1991), no. 12 1117–1138.
- [67] R. J. Prokop and A. P. Reeves, *A survey of moment-based techniques for unoccluded object representation and recognition*, *CVGIP: Graphical Models and Image Processing* **54** (1992), no. 5 438–460.
- [68] R. R. Bailey and M. Srinath, *Orthogonal moment features for use with parametric and non-parametric classifiers*, *IEEE Transactions on Pattern Analysis and Machine Intelligence* **18** (1996), no. 4 389–399.
- [69] Z. von F, *Beugungstheorie des schneidenver-fahrens und seiner verbesserten form, der phasenkontrastmethode*, *physica* **1** (1934), no. 7-12 689–704.
- [70] R. Ragazzoni, E. Marchetti, and G. Valente, *Adaptive-optics corrections available for the whole sky*, *Nature* **403** (2000), no. 6765 54.
- [71] S. Dong, J. Kettenbach, I. Hinterleitner, H. Bergmann, and W. Birkfellner, *The Zernike expansion—an example of a merit function for 2D/3D registration based on orthogonal functions*, in *International Conference on Medical Image Computing and Computer-Assisted Intervention*, pp. 964–971, Springer, 2008.
- [72] P. Markelj, D. Tomažević, B. Likar, and F. Pernuš, *A review of 3d/2d registration methods for image-guided interventions*, *Medical image analysis* **16** (2012), no. 3 642–661.
- [73] E. A. Kaye, Y. Hertzberg, M. Marx, B. Werner, G. Navon, M. Levoy, and K. B. Pauly, *Application of zernike polynomials towards accelerated adaptive focusing of transcranial high intensity focused ultrasound*, *Medical physics* **39** (2012), no. 10 6254–6263.
- [74] E. W. Weisstein, *Zernike polynomial*, .
- [75] R. J. Mathar, *Zernike basis to Cartesian transformations*, *arXiv preprint arXiv:0809.2368* (2008).

- [76] J. C. Wyant and K. Creath, *Basic wavefront aberration theory for optical metrology*, *Applied optics and optical engineering* **11** (1992), no. part 2 28–39.
- [77] R. H. Hahnloser, R. Sarpeshkar, M. A. Mahowald, R. J. Douglas, and H. S. Seung, *Digital selection and analogue amplification coexist in a cortex-inspired silicon circuit*, *Nature* **405** (2000), no. 6789 947.
- [78] G. E. Hinton, N. Srivastava, A. Krizhevsky, I. Sutskever, and R. R. Salakhutdinov, *Improving neural networks by preventing co-adaptation of feature detectors*, *arXiv preprint arXiv:1207.0580* (2012).
- [79] D. P. Kingma and J. Ba, *Adam: A method for stochastic optimization*, *arXiv preprint arXiv:1412.6980* (2014).
- [80] L. Debnath, *Nonlinear partial differential equations for scientists and engineers*. Springer Science & Business Media, 2011.
- [81] J. M. Burgers, *A mathematical model illustrating the theory of turbulence*, in *Advances in applied mechanics*, vol. 1, pp. 171–199. Elsevier, 1948.
- [82] T. Nagatani, *Density waves in traffic flow*, *Physical Review E* **61** (2000), no. 4 3564.
- [83] M. Sari and G. Gürarşlan, *A sixth-order compact finite difference scheme to the numerical solutions of Burgers’ equation*, *Applied Mathematics and Computation* **208** (2009), no. 2 475–483.
- [84] C.-W. Shu and S. Osher, *Efficient implementation of essentially non-oscillatory shock capturing schemes, 2*, .
- [85] D. V. Gaitonde and M. R. Visbal, *High-order schemes for Navier-Stokes equations: algorithm and implementation into FDL3DI*, tech. rep., Air Force Research Lab Wright-Patterson AFB OH Air Vehicles Directorate, 1998.
- [86] T. Oliphant, “NumPy: A guide to NumPy.” USA: Trelgol Publishing, 2006–.
- [87] J. Cullum, *Numerical differentiation and regularization*, *SIAM Journal on numerical analysis* **8** (1971), no. 2 254–265.
- [88] R. Chartrand, *Numerical differentiation of noisy, nonsmooth data*, *ISRN Applied Mathematics* **2011** (2011).
- [89] J. J. Stickel, *Data smoothing and numerical differentiation by a regularization method*, *Computers & chemical engineering* **34** (2010), no. 4 467–475.
- [90] A. Maas, Q. V. Le, T. M. O’neil, O. Vinyals, P. Nguyen, and A. Y. Ng, *Recurrent neural networks for noise reduction in robust ASR*, .

- [91] H. Shen, D. George, E. Huerta, and Z. Zhao, *Denoising gravitational waves using deep learning with recurrent denoising autoencoders*, *arXiv preprint arXiv:1711.09919* (2017).
- [92] N. Trask, R. G. Patel, B. J. Gross, and P. J. Atzberger, *GMLS-Nets: A framework for learning from unstructured data*, *arXiv preprint arXiv:1909.05371* (2019).
- [93] D. S. Broomhead and D. Lowe, *Radial basis functions, multi-variable functional interpolation and adaptive networks*, tech. rep., Royal Signals and Radar Establishment Malvern (United Kingdom), 1988.
- [94] T. Poggio and F. Girosi, *Networks for approximation and learning*, *Proceedings of the IEEE* **78** (1990), no. 9 1481–1497.
- [95] R. L. Hardy, *Multiquadric equations of topography and other irregular surfaces*, *Journal of geophysical research* **76** (1971), no. 8 1905–1915.
- [96] R. Franke, *Scattered data interpolation: tests of some methods*, *Mathematics of computation* **38** (1982), no. 157 181–200.
- [97] E. J. Kansa, *Multiquadrics—a scattered data approximation scheme with applications to computational fluid-dynamics—i surface approximations and partial derivative estimates*, *Computers & Mathematics with applications* **19** (1990), no. 8-9 127–145.
- [98] E. J. Kansa, *Multiquadrics—a scattered data approximation scheme with applications to computational fluid-dynamics—ii solutions to parabolic, hyperbolic and elliptic partial differential equations*, *Computers & mathematics with applications* **19** (1990), no. 8-9 147–161.
- [99] Y.-C. Hon and X. Mao, *An efficient numerical scheme for Burgers’ equation*, *Applied Mathematics and Computation* **95** (1998), no. 1 37–50.
- [100] H. Xie and D. Li, *A meshless method for Burgers’ equation using MQ-RBF and high-order temporal approximation*, *Applied Mathematical Modelling* **37** (2013), no. 22 9215–9222.
- [101] K. A. Sharp and B. Honig, *Calculating total electrostatic energies with the nonlinear poisson-boltzmann equation*, *Journal of Physical Chemistry* **94** (1990), no. 19 7684–7692.
- [102] M. Mirzadeh, M. Theillard, and F. Gibou, *A second-order discretization of the nonlinear Poisson-Boltzmann equation over irregular geometries using non-graded adaptive Cartesian grids*, *Journal of Computational Physics* **230** (Mar., 2011) 2125–2140.

- [103] P. Mistani, A. Guittet, D. Bochkov, J. Schneider, D. Margetis, C. Ratsch, and F. Gibou, *The island dynamics model on parallel quadtree grids*, *Journal of Computational Physics* **361** (2018) 150–166.
- [104] M. Theillard, F. Gibou, and T. Pollock, *A sharp computational method for the simulation of the solidification of binary alloys*, *Journal of scientific computing* **63** (2015), no. 2 330–354.
- [105] D. Bochkov, T. Pollock, and F. Gibou, *Sharp-interface simulations of multicomponent alloy solidification*, *arXiv preprint arXiv:2112.08650* (2021).
- [106] K. Galatsis, K. L. Wang, M. Ozkan, C. S. Ozkan, Y. Huang, J. P. Chang, H. G. Monbouquette, Y. Chen, P. Nealey, and Y. Botros, *Patterning and templating for nanoelectronics*, *Advanced Materials* **22** (2010), no. 6 769–778.
- [107] G. Y. Ouaknin, N. Laachi, K. Delaney, G. H. Fredrickson, and F. Gibou, *Level-set strategy for inverse dsa-lithography*, *Journal of Computational Physics* **375** (2018) 1159–1178.
- [108] D. Bochkov and F. Gibou, *A non-parametric shape optimization approach for solving inverse problems in directed self-assembly of block copolymers*, *arXiv preprint arXiv:2112.09615* (2021).
- [109] I. Babuška, *The finite element method for elliptic equations with discontinuous coefficients*, *Computing* **5** (1970), no. 3 207–213.
- [110] J. H. Bramble and J. T. King, *A finite element method for interface problems in domains with smooth boundaries and interfaces*, *Advances in Computational Mathematics* **6** (1996), no. 1 109–138.
- [111] R. J. LeVeque and Z. Li, *The immersed interface method for elliptic equations with discontinuous coefficients and singular sources*, *SIAM Journal on Numerical Analysis* **31** (1994), no. 4 1019–1044.
- [112] L. Adams and Z. Li, *The immersed interface/multigrid methods for interface problems*, *SIAM Journal on Scientific Computing* **24** (2002), no. 2 463–479.
- [113] Z. Li, T. Lin, and X. Wu, *New cartesian grid methods for interface problems using the finite element formulation*, *Numerische Mathematik* **96** (2003), no. 1 61–98.
- [114] R. E. Ewing, Z. Li, T. Lin, and Y. Lin, *The immersed finite volume element methods for the elliptic interface problems*, *Mathematics and Computers in Simulation* **50** (1999), no. 1-4 63–76.
- [115] Y. Gong, B. Li, and Z. Li, *Immersed-interface finite-element methods for elliptic interface problems with nonhomogeneous jump conditions*, *SIAM Journal on Numerical Analysis* **46** (2008), no. 1 472–495.

- [116] R. P. Fedkiw, T. Aslam, B. Merriman, and S. Osher, *A non-oscillatory eulerian approach to interfaces in multimaterial flows (the ghost fluid method)*, *Journal of computational physics* **152** (1999), no. 2 457–492.
- [117] X.-D. Liu, R. P. Fedkiw, and M. Kang, *A boundary condition capturing method for poisson’s equation on irregular domains*, *Journal of computational Physics* **160** (2000), no. 1 151–178.
- [118] A. Guittet, M. Lepilliez, S. Tanguy, and F. Gibou, *Solving elliptic problems with discontinuities on irregular domains—the voronoi interface method*, *Journal of Computational Physics* **298** (2015) 747–765.
- [119] R. Crockett, P. Colella, and D. T. Graves, *A cartesian grid embedded boundary method for solving the poisson and heat equations with discontinuous coefficients in three dimensions*, *Journal of Computational Physics* **230** (2011), no. 7 2451–2469.
- [120] A. J. Lew and G. C. Buscaglia, *A discontinuous-galerkin-based immersed boundary method*, *International Journal for Numerical Methods in Engineering* **76** (2008), no. 4 427–454.
- [121] N. Moës, J. Dolbow, and T. Belytschko, *A finite element method for crack growth without remeshing*, *International journal for numerical methods in engineering* **46** (1999), no. 1 131–150.
- [122] T. Belytschko, N. Moës, S. Usui, and C. Parimi, *Arbitrary discontinuities in finite elements*, *International Journal for Numerical Methods in Engineering* **50** (2001), no. 4 993–1013.
- [123] D. Bochkov and F. Gibou, *Solving elliptic interface problems with jump conditions on cartesian grids*, *Journal of Computational Physics* **407** (2020) 109269.
- [124] H. Lee and I. S. Kang, *Neural algorithm for solving differential equations*, *Journal of Computational Physics* **91** (1990), no. 1 110–131.
- [125] D. Gobovic and M. Zaghoul, *Design of locally connected cmos neural cells to solve the steady-state heat flow problem*, in *Proceedings of 36th Midwest Symposium on Circuits and Systems*, pp. 755–757, IEEE, 1993.
- [126] L. O. Chua and L. Yang, *Cellular neural networks: Theory*, *IEEE Transactions on circuits and systems* **35** (1988), no. 10 1257–1272.
- [127] L. O. Chua and L. Yang, *Cellular neural networks: Applications*, *IEEE Transactions on circuits and systems* **35** (1988), no. 10 1273–1290.

- [128] M. Raissi, P. Perdikaris, and G. Karniadakis, *Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations*, *Journal of Computational Physics* **378** (2019) 686–707.
- [129] A. Krishnapriyan, A. Gholami, S. Zhe, R. Kirby, and M. W. Mahoney, *Characterizing possible failure modes in physics-informed neural networks*, *Advances in Neural Information Processing Systems* **34** (2021) 26548–26560.
- [130] D. Johnson, T. Maxfield, Y. Jin, and R. Fedkiw, *Software-based automatic differentiation is flawed*, *arXiv preprint arXiv:2305.03863* (2023).
- [131] J. Sirignano and K. Spiliopoulos, *Dgm: A deep learning algorithm for solving partial differential equations*, *Journal of Computational Physics* **375** (2018) 1339–1364.
- [132] B. Yu *et. al.*, *The deep ritz method: a deep learning-based numerical algorithm for solving variational problems*, *Communications in Mathematics and Statistics* **6** (2018), no. 1 1–12.
- [133] J. Bradbury, R. Frostig, P. Hawkins, M. J. Johnson, C. Leary, D. Maclaurin, G. Necula, A. Paszke, J. VanderPlas, S. Wanderman-Milne, and Q. Zhang, *JAX: composable transformations of Python+NumPy programs*, 2018.
- [134] P. Holl, V. Koltun, K. Um, and N. Thuerey, *phiflow: A differentiable pde solving framework for deep learning via physical simulations*, in *NeurIPS Workshop*, vol. 2, 2020.
- [135] D. Kochkov, J. A. Smith, A. Alieva, Q. Wang, M. P. Brenner, and S. Hoyer, *Machine learning–accelerated computational fluid dynamics*, *Proceedings of the National Academy of Sciences* **118** (2021), no. 21 [<https://www.pnas.org/content/118/21/e2101784118.full.pdf>].
- [136] D. A. Bezgin, A. B. Buhendwa, and N. A. Adams, *Jax-fluids: A fully-differentiable high-order computational fluid dynamics solver for compressible two-phase flows*, *arXiv preprint arXiv:2203.13760* (2022).
- [137] K. Um, R. Brand, Y. R. Fei, P. Holl, and N. Thuerey, *Solver-in-the-loop: Learning from differentiable physics to interact with iterative pde-solvers*, *Advances in Neural Information Processing Systems* **33** (2020) 6111–6122.
- [138] F. de Avila Belbute-Peres, K. Smith, K. Allen, J. Tenenbaum, and J. Z. Kolter, *End-to-end differentiable physics for learning and control*, *Advances in neural information processing systems* **31** (2018).

- [139] P. Holl, V. Koltun, and N. Thuerey, *Learning to control pdes with differentiable physics*, *arXiv preprint arXiv:2001.07457* (2020).
- [140] T. Chen and H. Chen, *Universal approximation to nonlinear operators by neural networks with arbitrary activation functions and its application to dynamical systems*, *IEEE Transactions on Neural Networks* **6** (1995), no. 4 911–917.
- [141] L. Lu, P. Jin, and G. E. Karniadakis, *Deeponet: Learning nonlinear operators for identifying differential equations based on the universal approximation theorem of operators*, *arXiv preprint arXiv:1910.03193* (2019).
- [142] K. Bhattacharya, B. Hosseini, N. B. Kovachki, and A. M. Stuart, *Model reduction and neural networks for parametric pdes*, *arXiv preprint arXiv:2005.03180* (2020).
- [143] Z. Li, N. Kovachki, K. Azizzadenesheli, B. Liu, K. Bhattacharya, A. Stuart, and A. Anandkumar, *Neural operator: Graph kernel network for partial differential equations*, *arXiv preprint arXiv:2003.03485* (2020).
- [144] Z. Li, N. Kovachki, K. Azizzadenesheli, B. Liu, K. Bhattacharya, A. Stuart, and A. Anandkumar, *Fourier neural operator for parametric partial differential equations*, *arXiv preprint arXiv:2010.08895* (2020).
- [145] C. J. Shallue, J. Lee, J. Antognini, J. Sohl-Dickstein, R. Frostig, and G. E. Dahl, *Measuring the effects of data parallelism on neural network training*, *arXiv preprint arXiv:1811.03600* (2018).
- [146] R. Hecht-Nielsen, *Kolmogorov’s mapping neural network existence theorem*, in *Proceedings of the international conference on Neural Networks*, vol. 3, pp. 11–14, IEEE Press New York, NY, USA, 1987.
- [147] A. N. Kolmogorov, *On the representation of continuous functions of many variables by superposition of continuous functions of one variable and addition*, in *Doklady Akademii Nauk*, vol. 114, pp. 953–956, Russian Academy of Sciences, 1957.
- [148] D. A. Sprecher, *On the structure of continuous functions of several variables*, *Transactions of the American Mathematical Society* **115** (1965) 340–355.
- [149] V. Ismailov, *A three layer neural network can represent any multivariate function*, 2020.
- [150] P. J. Huber, *Robust estimation of a location parameter*, in *Breakthroughs in statistics: Methodology and distribution*, pp. 492–518. Springer, 1992.
- [151] R. A. Saleh and A. Saleh, *Statistical properties of the log-cosh loss function used in machine learning*, *arXiv preprint arXiv:2208.04564* (2022).

- [152] D. Bochkov and F. Gibou, *Solving elliptic interface problems with jump conditions on cartesian grids*, *Journal of Computational Physics* **407** (2020) 109269.
- [153] M. Hessel, D. Budden, F. Viola, M. Rosca, E. Sezener, and T. Hennigan, *Optax: composable gradient transformation and optimisation*, in *jax!*, 2020.
- [154] R. Pascanu, T. Mikolov, and Y. Bengio, *On the difficulty of training recurrent neural networks*, 2012.
- [155] B. Curless and M. Levoy, *A volumetric method for building complex models from range images*, in *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, pp. 303–312, 1996.
- [156] W.-F. Hu, T.-S. Lin, and M.-C. Lai, *A discontinuity capturing shallow neural network for elliptic interface problems*, *Journal of Computational Physics* **469** (2022) 111576.
- [157] S. Wu and B. Lu, *Inn: Interfaced neural networks as an accessible meshless approach for solving interface pde problems*, *Journal of Computational Physics* **470** (2022) 111588.
- [158] M. Blondel, Q. Berthet, M. Cuturi, R. Frostig, S. Hoyer, F. Llinares-López, F. Pedregosa, and J.-P. Vert, *Efficient and modular implicit differentiation*, *arXiv preprint arXiv:2105.15183* (2021).
- [159] K. Stüben *et. al.*, *An introduction to algebraic multigrid*, *Multigrid* (2001) 413–532.
- [160] R. D. Falgout, *An introduction to algebraic multigrid*, tech. rep., Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States), 2006.
- [161] S. H. Friend, G. S. Ginsburg, and R. W. Picard, *Wearable digital health technology*, 2023.
- [162] V. Berisha, C. Krantsevich, P. R. Hahn, S. Hahn, G. Dasarathy, P. Turaga, and J. Liss, *Digital medicine and the curse of dimensionality*, *NPJ digital medicine* **4** (2021), no. 1 153.
- [163] J. Leander, M. Jirstrand, U. G. Eriksson, and R. Palmér, *A stochastic mixed effects model to assess treatment effects and fluctuations in home-measured peak expiratory flow and the association with exacerbation risk in asthma*, *CPT: Pharmacometrics & Systems Pharmacology* **11** (2022), no. 2 212–224.
- [164] J. Fagin, J. W. Park, H. Best, K. S. Ford, M. J. Graham, V. A. Villar, S. Ho, J. H.-H. Chan, and M. O’Dowd, *Latent stochastic differential equations for modeling quasar variability and inferring black hole properties*, *arXiv preprint arXiv:2304.04277* (2023).

- [165] Y. Mei, A. Carbo, R. Hontecillas, and J. Bassaganya-Riera, *Enisi sde: a novel web-based stochastic modeling tool for computational biology*, in *2013 IEEE International Conference on Bioinformatics and Biomedicine*, pp. 392–397, IEEE, 2013.
- [166] M. Tajmiriahi and Z. Amini, *Modeling of seizure and seizure-free eeg signals based on stochastic differential equations*, *Chaos, Solitons & Fractals* **150** (2021) 111104.
- [167] N. Evangelou, F. Dietrich, J. M. Bello-Rivas, A. J. Yeh, R. S. Hendley, M. A. Bevan, and I. G. Kevrekidis, *Learning effective sdes from brownian dynamic simulations of colloidal particles*, *Molecular Systems Design & Engineering* (2023).
- [168] F. Dietrich, A. Makeev, G. Kevrekidis, N. Evangelou, T. Bertalan, S. Reich, and I. G. Kevrekidis, *Learning effective stochastic differential equations from microscopic simulations: Linking stochastic numerics to deep learning*, *Chaos: An Interdisciplinary Journal of Nonlinear Science* **33** (2023), no. 2.
- [169] J. Lu, B. Bender, J. Y. Jin, and Y. Guan, *Deep learning prediction of patient response time course from early data via neural-pharmacokinetic/pharmacodynamic modelling*, *Nature machine intelligence* **3** (2021) 696–704.
- [170] M. Laurie and J. Lu, *Explainable deep learning for tumor dynamic modeling and overall survival prediction using neural-ode*, *npj Systems Biology and Applications* **9** (2023), no. 1 58.
- [171] N. L. Dayneka, V. Garg, and W. J. Jusko, *Comparison of four basic models of indirect pharmacodynamic responses*, *Journal of pharmacokinetics and biopharmaceutics* **21** (1993), no. 4 457–478.