Combining Deep Learning and Physics Models for Efficient and Robust Architectures

Filipe de Avila Belbute Peres

CMU-CS-22-148
September 2022

School of Computer Science
Computer Science Department
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
J. Zico Kolter, Chair
Zachary Manchester
Katerina Fragkiadaki
Venkat Viswanathan
Fei Sha (Google Research)

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

Copyright © 2022 **Filipe de Avila Belbute Peres**

This research was sponsored by Robert Bosch GMBH and the Defense Advanced Research Projects Agency under award number HR00112020006. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of any sponsoring institution, the U.S. government or any other entity.

Keywords: artificial intelligence, machine learning, deep learning, physics, neural networks, differential equations, rigid body dynamics, fluid dynamics

To my parents.

Abstract

Over the last decade, deep learning methods have achieved success in diverse domains, becoming one of the most widely employed approaches in artificial intelligence. These recent successes have also motivated their application in physics domains, such as solving differential equations, or predicting the motion of objects or the behavior of fluids.

Deep learning methods have as their strengths their extreme flexibility, allowing complex dynamics to be learned directly from data, and their proven track record working directly on unstructured, high-dimensional domains (such as image and video processing). However, these approaches also face some issues, such as difficulty in generalizing outside the training domain, large data requirements, and costly training. Traditional models of physics, on the other hand, have been developed to be universally valid within their domain of application (i.e., generalizable) and require little to no data for modelling.

In this proposal, we present methods for leveraging the strengths of both types of approaches, by combining deep learning and physics models. This allows for the development of deep learning architectures that are more data-efficient and robust to generalization than their standard, "physics-unaware" counterparts.

These methods fall under two broad categories: differentiable physics layers and physics-informed learning approaches. In the first group of methods, we embed full physics simulators as layers into deep learning models, fully constraining their outputs to match the underlying dynamics. By having these simulations be fully differentiable, we maintain the ability to train these systems end-to-end. We present the application of such methods to problems in rigid body and fluid dynamics.

Physics-informed learning methods provide information about the underlying physics in the form of physics-informed loss terms, which regularize the model's outputs to be physically consistent. We present a method to improve these approaches in order to learn parameterized systems of differential equations more efficiently. We also analyze theoretically and empirically the usage of sinusoidal neural networks to address known issues, such as spectral bias, in neural networks performing physics-informed learning.

Acknowledgments

The work presented in this thesis would obviously not have been possible without the support from my advisor, Zico Kolter. I want to first thank him for his research instincts, which were crucial in guiding this research process, and for his impressive knowledge of the wide array of topics spanned not only by the work presented here, but also by the many topics that did not make it into this thesis.

I am also greatly indebted to all my co-authors for the work discussed here, without whom none of this would have been possible. I am grateful to Kevin Smith, for being a great mentor even before I started my PhD, and for demonstrating what systematic thinking and careful research should look like. I am also thankful to Kevin, together with Joshua Tenenbaum and Kelsey Allen, for shaping my initial research interests and teaching me about a whole area of AI and cognitive psychology. In the same spirit, I also want to thank Tom Economon for being a welcoming and always available advisor, and most importantly also for guiding me in learning about everything CFD related. Moreover, I must also thank all the team at the Bosch Center for AI, not only for creating a positive work environment, but also for funding our group's research. I also want to thank Fei Sha for being a supportive advisor, always available and excited to discuss ideas and new directions. I am grateful to Yi-fan Chen too, for always taking time to make sure I had a positive experience while at Google. Fei, Yi-fan and the whole Google Research team truly build a great research environment, with their lively discussions and general excitement about the possibilities of their research.

I want to express my gratitude also to the members of the thesis committee, Zico Kolter, Fei Sha, Katerina Fragkiadaki, Zachary Manchester and Venkat Viswanathan, for their support throughout this process and their willingness to make accommodations to make everything possible.

During my years at CMU, I have also been fortunate to have had many discussions with my labmates. I am thankful to all the past and present members who were there along the journey: Brandon Amos, Vaishnavh Nagarajan, Eric Wong, Shaojie Bai, Powei Wang, Priya Donti, Alnur Ali, Gaurav Manek, Mel Roderick, Jeremy Cohen, Leslie Rice, Ezra Winston, Suvansh Sanjeev, Yash Savani, Samuel Sokota, Chun Kai Ling, Zhili Feng, Victor Akinwande, Christina Baek, Anna Bair, Sachin Goyal, Swaminathan Gurumurthy, Yiding Jiang, Pratyush Maini, Dylan Sam, Mingjie Sun, Ashwini Pokle, Asher Trockman, Josh Williams, Ezra Winston, Runtian Zhai and Huan Zhang.

Finally, I want to also thank my friends and family for the support throughout this journey. In particular I would like to thank my parents, to whom this work is dedicated. Beside their always present, unlimited support, they have always set the highest examples for me on every aspect of life, but most importantly of all they have always been my compass for what it means to be a good person.

Contents

1 Introduction **1**
1.1 Deep learning and traditional physics . . . 2
1.2 Contributions . . . 3
1.2.1 Part I: Learning with Differentiable Physics Layers . . . 3
1.2.2 Part II: Improving Physics-informed Learning . . . 3

2 Preliminaries & Background **5**
2.1 Combining Deep Learning and Physics Learning . . . 5
2.1.1 Differentiable physics layers . . . 5
2.1.2 Physics-informed losses . . . 6
2.2 Background . . . 7
2.2.1 Physics-informed learning . . . 7
2.2.2 Differentiable physics layers . . . 8
2.2.3 Rigid body dynamics . . . 8
2.2.4 Fluid dynamics . . . 9
2.2.5 Graph neural networks . . . 10
2.2.6 Sinusoidal networks . . . 10
2.2.7 Neural tangent kernel . . . 11

I Learning with Differentiable Physics Layers **13**

3 Learning and Control with Differentiable Rigid Body Dynamics **15**
3.1 Introduction . . . 15
3.2 Differentiable Physics Engine . . . 16
3.2.1 Formulating the LCP . . . 16
3.2.2 Solving the LCP . . . 17
3.2.3 Gradients . . . 17
3.2.4 Implementation . . . 18
3.3 Experiments . . . 18
3.3.1 Parameter learning . . . 18
3.3.2 Prediction on visual data . . . 20
3.3.3 Control . . . 22

4 Fluid Flow Prediction with Graph Neural Networks and Differentiable Fluid Dynamics **25**
4.1 Introduction 25
4.2 CFD-GCN 26
4.2.1 Architecture 26
4.2.2 Training 31
4.3 Experiments 32
4.3.1 Interpolation 33
4.3.2 Generalization 34
4.3.3 Runtime 36

II Improving Physics-Informed Learning 45

5 Solving Parameterized Differential Equations with Physics-Informed Hypernetworks **47**
5.1 Introduction 47
5.2 Preliminaries 48
5.2.1 Differential equations 48
5.2.2 Physics-Informed Neural Networks 49
5.2.3 Multistep Neural Networks 49
5.2.4 Hypernetworks 50
5.3 HyperPINN 50
5.4 Experiments 52
5.4.1 1D Burgers' equation 52
5.4.2 Lorenz system 54

6 Simple Sinusoidal Networks **59**
6.1 Introduction 59
6.2 Simple Sinusoidal Networks 60
6.2.1 Practical Implementation Details of SIRENs 61
6.2.2 Simplifying SIRENs 62
6.3 Experiments 67
6.3.1 Image 68
6.3.2 Poisson 69
6.3.3 Video 70
6.3.4 Audio 71
6.3.5 Helmholtz equation 72
6.3.6 Signed distance function (SDF) 73

7 Understanding and Applying Sinusoidal Networks **75**
7.1 Preliminaries 75
7.2 Shallow sinusoidal networks 78
7.2.1 SIREN 78

7.2.2 Simple sinusoidal network . . . 81
7.3 Deep sinusoidal networks . . . 85
7.3.1 Simple sinusoidal network . . . 85
7.3.2 SIREN . . . 88
7.4 Empirical Analysis . . . 90
7.5 Tuning ω . . . 93
7.5.1 Choosing ω from the Nyquist frequency . . . 95
7.5.2 Multi-dimensional ω . . . 96
7.5.3 Choosing ω from available information . . . 97
7.6 Experiments . . . 97
7.6.1 Evaluating generalization . . . 97
7.6.2 Solving differential equations . . . 98

III Conclusion **105**

8 Conclusion **107**

A Building a Differentiable Rigid Body Dynamics Engine **109**
A.1 Physics Engine . . . 109
A.1.1 Step Overview . . . 109
A.1.2 Bodies . . . 110
A.1.3 Global Parameters . . . 111
A.1.4 Contact Detection . . . 111
A.1.5 Constraints . . . 113
A.1.6 Dynamics LCP . . . 115
A.2 Solution and Derivatives . . . 117
A.2.1 Solution . . . 117
A.2.2 Derivatives . . . 118

Bibliography **121**

Chapter 1

Introduction

Over the last decade, deep learning methods have achieved success in diverse domains, becoming one of the most widely employed approaches in artificial intelligence . These recent successes have also motivated their application in physics domains, such as solving differential equations [Bar-Sinai et al., 2019, Long et al., 2018, Raissi et al., 2019a], or predicting the motion of objects [Battaglia et al., 2016, Chang et al., 2016, Ehrhardt et al., 2017] or the behavior of fluids [Afshar et al., 2019, Guo and Hesthaven, 2019, Kochkov et al., 2021].

The connection between these two disparate fields is greater than one might initially expect. When one thinks of the connection between deep learning and physics, what usually comes to mind are traditional tasks, such as finding solutions to differential equations, learning the dynamics of fluids, etc. Indeed, there has been a large amount of work in applying deep learning methods to these types of tasks.

However, this view is in fact too restrictive. In some sense, physics is at least implicitly a part of a large amount of artificial intelligence tasks. Any agent that interacts with the real world will need to have at least an implicit understanding of how the physics of the natural world work. For example an agent that manipulates objects needs to understand how these objects will move and interact. This is in fact borne out by the studies on infants that demonstrate that humans posses an innate knowledge of physics, which they leverage to interact with and learn efficiently about the world [Spelke and Kinzler, 2007].

Moreover, even agents that do not interact with the real world might have a need for understanding physics. For example, planning and controlling agents in a video game environment often involves understanding the dynamics of that environment. Given that such simulated environments are created by humans and for humans, it has been shown they are often similar in many important ways to the real world [Dubey et al., 2018], allowing humans to learn with great efficiency in these environments.

Therefore, having an understanding of physics can be useful not only for tasks seen as traditionally in the scope of the discipline of physics, but also to artificial intelligence in general.

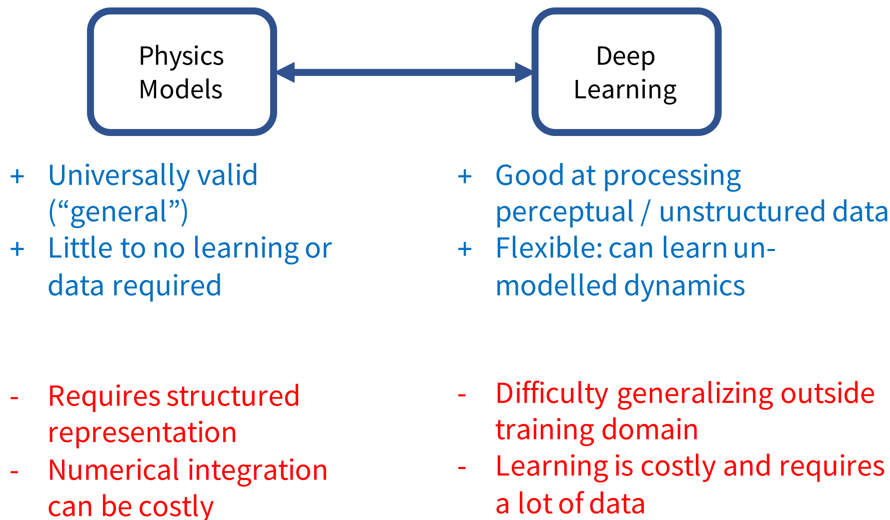


Figure 1.1: Comparison of general characteristics traditional physics models and conventional deep learning approaches.

1.1 Deep learning and traditional physics

Despite both the explicit and implicit connections to physics problems, the trend when applying deep learning methods to such tasks is to approach problems in a purely "data-driven" fashion. That is, to gather a large amount of data from the process of interest and to perform some sort of statistical learning procedure on that data, usually with minimal amount of prior knowledge provided to the deep learning model. This approach brings with it many advantages, demonstrated clearly in the many recent successes of deep learning. For instance, it is very clear that deep learning methods are very strong at processing unstructured, high-dimensional data – the type of data that usually arises from perceptual tasks, such as image, video and audio processing [Goodfellow et al., 2014, He et al., 2015, Schneider et al., 2019]. In comparison, traditional methods in physics commonly require data to be provided in a structured form, such as the explicit positions and velocities of particles or objects. Moreover, deep learning methods, given their parameterized learning-based nature and their universal approximation capabilities, are extremely flexible and able to learn to approximate even unknown, not modelled or partially measured dynamics [Eivazi et al., 2021, Kochkov et al., 2021].

Nevertheless, deep learning methods also have some drawbacks. It is known that these types of approaches can require large amounts of data in order to learn properly [Marcus, 2018]. In contrast, traditional physics models require little to no data for training, as most of the information regarding the underlying dynamics is hardcoded into the model by design. Moreover, it is known that deep learning methods also face issues when generalizing outside of the training domain [Belbute-Peres et al., 2020]. Combined with the high-cost associated with training a new model (compared to their relatively cheap evaluation), this imposes a severe restriction on their practical applicability. Physics models, conversely, are

generally designed to be universal, at least within their intended domain of application. A prototypical example of this are the Navier-Stokes equations, which are able to model fluid dynamics from everyday scales to the scale of jet engines.

Figure 1.1 presents a summarized representation of the general differences between these approaches.

In many domains, we indeed have a large body of knowledge acquired over the centuries from scientists who studied the natural world and developed accurate models of the underlying dynamics for diverse physical processes. In this thesis we propose methods for combining deep learning and physics models in order to develop architectures that leverage their aforementioned complementary strengths (outlined in blue and positive in Figure 1.1). By employing end-to-end trainable deep learning architectures, we are able to have models that are flexible and capable of learning from data. Additionally, the usage of physics models allows for models that are capable of achieving greater data efficiency and that are robust to data from outside the training domain.

1.2 Contributions

In the following chapter, we will summarize the relevant preliminary concepts and background to the rest of this thesis. After that, the subsequent chapters will describe in detail the relevant research contributions in this thesis.

1.2.1 Part I: Learning with Differentiable Physics Layers

In Part I, we present methods for developing and utilizing differentiable physics models as special neural network layers in order to embed structured physics knowledge into deep learning models.

- Chapter 3 presents a method for developing an analytically differentiable rigid-body physics engine, such that it can be employed as a layer in an end-to-end trainable deep learning system. We demonstrate that this method allows for improved data-efficiency in both prediction and control tasks.
- Chapter 4 presents a method for employing fast fluid simulations in conjunction with a graph neural network, such that we can have an architecture that is able to perform fluid flow predictions on unstructured meshes. We demonstrate that this method can perform efficient predictions and is able to generalize robustly outside of its training domain.

1.2.2 Part II: Improving Physics-informed Learning

In Part II, we present methods to address failure modes of physics-informed neural networks, such as their spectral bias and lack of generalization, by employing practical and theoretical techniques from modern deep learning learning.

- Chapter 5 presents a method for utilizing soft constraints, in the form physics-informed losses, combined with a weak inductive bias, in the form of a hierarchical architecture, the hypernetwork. We demonstrate that this method allows for more efficient learning of the solutions to parameterized differential equations.
- Chapter 6 introduces the concept of sinusoidal networks, which are of interest for physics-informed learning, and propose a simplified version of these architectures that still maintains their main benefits, such as the ability to avoid the spectral bias of traditional neural networks. We demonstrate that these simple sinusoidal networks have performance equivalent to more complex alternatives that have been proposed.
- Chapter 7 utilizes neural tangent kernel theory to analyze the sinusoidal networks proposed in the previous chapter, demonstrating their behavior is similar to that of low-pass filters and informing methods to tune their properties in order to achieve improved performance in physics-informed differential equation tasks.

Chapter 2

Preliminaries & Background

This thesis brings together many ideas from deep learning and physics. Though an extensive treatments of all these topics would be impossible, in this chapter, we provide a high-level background on the main topics employed. More specific concepts and lower level details are left to each individual chapter.

2.1 Combining Deep Learning and Physics Learning

The ideas we present in this thesis can be organized into two categories, which broadly match the main components of a deep learning model, the choice of architecture and the choice of objective.

In general, deep learning models can be seen as a composition of functions, *i.e.* a deep learning model f is in fact a series of neural network "layers"

$$f(x) = f_n \circ \cdots \circ f_1(x). \tag{2.1}$$

One common way of specifying the network architecture is by specifying the nature of these layers (*e.g.*, convolutional, fully connected, self-attention, etc).

Given a certain architecture, a neural network f_θ will be parameterized by some learnable parameters θ. These parameters are learned through an optimization procedure, which requires the choice of an objective (or loss) function. For example, for some ground truth training data $(x_i, y_i)_{i=1}^n$ and network predictions $f_\theta(x_i)$, a common objective is the mean squared error on the training data

$$\mathcal{L}(\theta) = \frac{1}{n} \sum_{i=1}^{n} \|y_i - f_\theta(x_i)\|_2^2. \tag{2.2}$$

These two components naturally suggest the two ways in which we combine our knowledge of physics with deep learning methods. We describe each in turn below.

2.1.1 Differentiable physics layers

As mentioned above, the layer functions f_i from Equation 2.1 come from a restricted set of commonly used architectural "building blocks", such as fully-connected layers,

convolutional layers, etc. Nevertheless, we need not be restricted to this set. From a deep learning optimization perspective, in which essentially all optimization methods used in practice are first-order, the main requirement we have is that the layer functions f_i have to be differentiable, so that the backpropagation algorithm [Rumelhart et al., 1986] can compute the derivatives with respect to the network's parameters and the preceding layers.

Therefore, one natural way of hard-constraining a neural network to conform to a given physics model ϕ is to set one of the layers $f_k(x) = \phi(x)$. That is

$$f(x) = f_n \circ \cdots \circ \phi \circ \cdots \circ f_1(x).$$

This hard-constraints the network f to have outputs that conform to the physics model ϕ at layer f_k. As a concrete example, which we explore in Chapter 3, ϕ could be a discrete rigid body dynamics model, that takes in as input rigid body positions and velocities, and returns these quantities at the next time step, *i.e.* $(x_{t+1}, v_{t+1}) = \phi(x_t, v_t)$. If we are able to frame the function $\phi(x)$ as a differentiable function, then it would still be possible to train the network f end-to-end using regular backpropagation.

In this way, we can still have a learnable neural network architecture, but at the same time incorporate prior physics knowledge into our model. This allows for deep learning models that are more efficient at learning and better at generalizing, due to the relevant information contained in the physics models. In Part I, we demonstrate examples of this approach, with networks that contain layers constrained to follow both rigid body dynamics (in Chapter 3) and fluid dynamics (in Chapter 4).

2.1.2 Physics-informed losses

As described above, deep learning models are trained by optimizing a choice of loss function, $\mathcal{L}(\theta)$. In the most common machine learning paradigm, supervised learning, this function is given by some error between the true labels y_i and the prediction from the neural network with parameters θ, as shown for example in Equation 2.2.

However, we need not be constrained only by this paradigm. Different choices of loss functions can be employed to penalize parameter configurations that do not conform to some desired behavior. Therefore, another way we can leverage our prior knowledge of physics is by designing loss functions that penalize neural networks that do not conform to our physics model for a given task.

For example, we can "soft-constrain" a neural network f_θ to conform to a certain differential equation model by adding loss terms that penalize deviations from those equations. If we know that our dynamics are specified by the differential equation

$$\frac{\partial f}{\partial t} = \frac{\partial f}{\partial x}, \tag{2.3}$$

then in order to have our deep learning model conform to these dynamics, we can add a loss term of the form

$$\mathcal{L}_{\text{phys}}(\theta) = \left\| \frac{\partial f_\theta(x)}{\partial t} - \frac{\partial f_\theta(x)}{\partial x} \right\|_2^2,$$

which will penalize parameterizations of f_θ that do not satisfy Equation 2.3, without having to hard-constraining the model to these dynamics, as we did in the previous section.

These types of loss terms are called physics-informed losses, and are the basis for the set of methods called physics-informed neural networks (PINNs) [Raissi et al., 2019a]. They are described in more detail in Part II. In Chapter 5 we present a method for efficiently using these types of soft-constrained deep learning models to learn parameterized systems of differential equations efficiently, and in Chapters 6 and 7 we propose a different architecture, utilizing sinusoidal activation functions, that enables more efficient learning when utilizing these physics-informed losses.

2.2 Background

2.2.1 Physics-informed learning

Differential equations. The differential equations we study in this thesis take the general form

$$\begin{aligned} &\mathcal{N}[t, x, u(t,x); \lambda] = 0, \\ &\text{with } t \in [0,T], x \in \Omega, \end{aligned} \tag{2.4}$$

where $\mathcal{N}[\cdot;\lambda]$ is an arbitrary (possibly non-linear) differential operator, which can contain time and space derivatives, and is parameterized by some list of parameters $\lambda \in \mathbb{R}^d$. Here, t is the time variable ranging up to time T, x is the D-dimensional spatial variable in some domain $\Omega \subseteq \mathbb{R}^D$, with boundary $\partial\Omega$, and $u(t,x)$ is the solution function to the differential equation. In order for a solution to be defined, initial and boundary conditions need to be provided. These can assume different forms, but in general initial conditions define $u(0,x)$ for $x \in \Omega$, and boundary conditions define $u(t,x)$ for $x \in \partial\Omega$ and $t \in [0,T]$.

As a concrete example of this formulation, we can take a look at a wave equation in one dimension, which is given by

$$\frac{\partial^2 u}{\partial t^2} = c^2 \frac{\partial^2 u}{\partial x^2} \tag{2.5}$$

Here $\mathcal{N}$ from Equation 2.4 is a non-linear operator containing second derivatives that defines the left-hand side of the equation, and $\lambda = c$.

Physics-Informed Neural Networks. Physics-informed neural networks [Raissi et al., 2019a] are a method for approximating the solution to differential equations, such as Equation 2.4, using neural networks (NNs). In this method, a neural network $\hat{u}(t,x;\theta)$, with learned parameters θ, is trained to approximate the actual solution function $u(t,x)$ to a given partial differential equation (PDE).

Importantly, PINNs employ not only a standard "supervised" data loss, but also a physics-informed loss, which consists of the differential equation residual defined by $\mathcal{N}$. Thus, for a given optimization hyper-parameter α, the training loss consists of

$$
\begin{aligned}
L(\theta) &= L_{\text{data}}(\theta) + \alpha L_{\text{physics}}(\theta), \\
L_{\text{data}}(\theta) &= \sum_{(t_i, x_i, u_i) \in \mathcal{D}} [\hat{u}(t_i, x_i; \theta) - u_i]^2, \\
L_{\text{physics}}(\theta) &= \sum_{(t_c, x_c) \in \mathcal{C}} \mathcal{N}[t_c, x_c, \hat{u}(t_c, x_c; \theta); \lambda]^2,
\end{aligned} \tag{2.6}
$$

where $\mathcal{D}$ is a dataset containing ground-truth values for u (e.g., from simulation data) at points (t_i, x_i), and $\mathcal{C}$ is a set of collocation points at which to evaluate the differential equation residual (which does not require ground-truth solution data). While the data in $\mathcal{D}$ can be used to enforce the initial and boundary conditions, the physics-informed loss term regularizes the search space, penalizing functions $\hat{u}$ that do conform to the differential equations, reducing the need for simulation data.

2.2.2 Differentiable physics layers

The work in Part I on differentiable physics layers relates methodologically to a recent trend of incorporating more structured layers within deep networks [Ramsundar et al., 2021]. Specifically, recent work has looked incorporating quadratic programs [Amos and Kolter, 2017], combinatorial optimization [Djolonga and Krause, 2017], computing equilibria in games [Ling et al., 2018], or dynamic programming [Mensch and Blondel, 2018].

The work in Chapter 3 on differentiable rigid body dynamics relates most closely to that of [Amos and Kolter, 2017]. Like this work, we use an interior point primal dual method to solve a nonlinear set of equations (in our case a general LCP, in their case a symmetric LCP resulting from the KKT conditions of QP). However, both the general nature of the LCP, and the application to physical simulation, specializes substantially from what has been considered previously. Full details for our approach are described in Appendix A.

The work in Chapter 4 uses a differentiable CFD solver, which allows us to embed a fluid simulation as a layers in a deep learning model. Differentiation of PDE solvers has been used for decades for shape optimization in aerodynamics Jameson [1988], with diverse formulations, such as the continuous adjoint Economon et al. [2015a] and the discrete adjoint Albring et al. [2015, 2016] being available. The adjoint method has also been applied in the graphics literature, but with different goals. McNamara et al. [2004] developed a differentiable fluid simulator for controlling smoke animations by optimizing forces acting on the smoke in order to have it match target shapes. In our work, we utilize SU2 [Economon et al., 2016], which is an open-source suite that provides both CFD simulations and adjoint-based differentiation.

2.2.3 Rigid body dynamics

Although they were not developed purely within the machine learning community, physical simulation tools such as MuJoCo [Todorov et al., 2012], Bullet [Coumans et al., 2013], and DART [Lee et al., 2018b], have become ubiquitous tools particularly in reinforcement learning. Despite their power, computing derivatives through these engines mostly involves

using finite difference methods, i.e. evaluating the forward simulation multiple times with small perturbations to the relevant parameters to approximate the relevant gradients. This strategy is often impractical due to (1) the high computational burden of finite differencing when computing the gradient with respect to a large number of model/policy parameters; and (2) the instability of numerical gradients over long time horizons, especially if contacts change over the course of a rollout.

In Chapter 3, we employ the strategy of directly differentiating the optimization problem that defines the dynamics. The analytic differentiation avoids the aforementioned issues, and can give gradients with respect to a large number of parameters essentially "for free" given a forward solution. Full details for our approach are described in Appendix A. The usage of analytical gradients in physics simulation has been previously investigated in spring-damper systems [Hermans et al., 2014]. However, due to its limitations, such as instability and unrealistic contact handling, most engines used in practice do not use spring-damper models. Degrave et al. [2016] also develop a differentiable physics engine, with similar motivations. However, in this case the engine is made differentiable by simply implementing it in its entirety in the Theano framework [Al-Rfou et al., 2016]. This severely limited the complexity of the allowable operations: for instance the engine only allowed for collision between balls and the ground plane.

In a related but orthogonal body of work, many studies have investigated the human ability to intuitively understand physics. Battaglia et al. [2013], Hamrick et al. [2015] and Smith and Vul [2013] suggested that people have an "intuitive physics engine" that they can use to simulate future or hypothetical states of the world for inference and planning. Recent work in machine learning has leveraged this idea by attempting to design networks that can learn physical dynamics in a differentiable system [Battaglia et al., 2016, Chang et al., 2016, Lerer et al., 2016], but because these dynamics must be learned, they require extensive training before they can be used as a layer in a larger network, and it is not clear how well they generalize across tasks. Conversely, by performing explicit simulation (similar to how people do), which is embedded as a "layer" in the system, our approach requires no pre-training and can generalize across scenarios that can be captured by a rigid-body engine.

2.2.4 Fluid dynamics

Many recent works have explored the interface between machine learning and CFD. In many cases, the approach has been model-free, aiming to directly learn to predict physical processes using solely deep learning methods [Afshar et al., 2019, Guo et al., 2016], with Afshar et al. [2019] applying an encoder-decoder convolutional architecture to the task of predicting flow fields around an airfoil. However, in many such works, generalization was not evaluated at a range of parameters that would generate behavior significantly different from the ones seen during training.

Others have looked at employing deep learning models as function approximators that substitute certain terms in equations of interest, such as the turbulence terms in turbulence modeling [Duraisamy et al., 2019, King et al., 2018, Singh et al., 2017]. In contrast to these approaches, which embed deep learning models as a component in larger physical models,

the approach we describe in Chapter 4 aims to embed a full physical simulation as a layer in a deep learning system.

Many papers have also explored applications of machine learning to fluid simulations for graphics. Unlike in CFD, fluid animations have the main goal of looking realistic, not necessarily aiming to model physical laws or conform perfectly to reality. Graphics applications are frequently more naturally suited to deep learning methods, as in many such applications the simulations are natively performed in structured grids. Some success has been achieved in generating realistic animations of smoke or water [Kim et al., 2018, Um et al., 2017, Wiewel et al., 2018].

Additionally, machine learning methods have also been applied to particle-based methods [Macklin and Müller, 2013] in order to develop a differentiable fluid simulator [Schenck and Fox, 2018]. Since these methods do not focus on accurately modeling the physical processes, having as their main goal generating realistic animations, they are unsuited for CFD tasks such as predicting aerodynamic flows for practical engineering applications.

2.2.5 Graph neural networks

In Chapter 4, as a consequence of working with unstructured meshes, our proposed method makes use of many recent advances in graph neural networks. These are neural networks that operate on general graph structures, instead of the regular grids required by convolutional networks. In the domain of physics, this allows for the usage of unstructured meshes, which are common due to their efficiency in allocating higher node density only where it is needed.

The graph convolution operation we use in our models was originally proposed by Kipf and Welling [2016], though many other approaches have also been proposed [Bronstein et al., 2017, Defferrard et al., 2016, Hamilton et al., 2017]. A detailed description of this architecture is provided in Chapter 4

Other works in the space of graph neural networks have worked on applying graph neural networks to meshes [Hanocka et al., 2019] or on graphs with positional information [Qi et al., 2017]. However, in their work, the modifications to the mesh do not attempt to preserve or improve its functionality, serving only the purpose of pooling for a classification task. Alet et al. [2019] employed graph neural networks with positional information to mesh a continuous space and model spatial processes. In this work, the dynamics were learned solely by the graph neural network, without the usage of any PDE solver. To the best of our knowledge, our work is the first to directly modify a mesh to optimize its functionality for a downstream task, through using it on a differentiable simulator.

2.2.6 Sinusoidal networks

A "sinusoidal network" is a neural network with a sine non-linearities, instead of traditional non-linearities such as the hyperbolic tangent (tanh) or the rectified linear unit (ReLU). A popular recent example of such type of neural network are sinusoidal representation networks (SIRENs) [Sitzmann et al., 2020]. Sinusoidal networks have also been evaluated in physics-informed learning settings, demonstrating promising results in a series of domains [Huang et al., 2021a,b, Raissi et al., 2019b, Song et al., 2021, Wong et al., 2022]. Sinusoidal

networks are discussed in detail in Chapter 6, with their behavior analysed through the lens of the neural tangent kernel in Chapter 7.

Among the benefits of such networks is the fact that the mapping of the inputs through an (initially) random linear layer followed by a sine function is mathematically equivalent to a transformation to a random Fourier basis, rendering them close to networks with Fourier feature transformations [Rahimi and Recht, 2007, Tancik et al., 2020]. Additionally, the periodic nature of such activation functions renders the network shift-invariant with respect to its inputs [Mildenhall et al., 2020, Tancik et al., 2020].

Moreover, Sitzmann et al. [2020] argue that SIRENs have the property of being closed under derivation, given that the derivative of its outputs with respect to its inputs is given by another sinusoidal network, due to the fact that

$$\frac{d}{dx}\sin(x) = \cos(x) = \sin\left(x + \frac{\pi}{2}\right). \tag{2.7}$$

This contrasts with the derivatives of networks with, for example, a traditional ReLU activation, which has vanishing higher-order derivatives.

2.2.7 Neural tangent kernel

An important prior result to the neural tangent kernel (NTK) is the neural network Gaussian process (NNGP). It has been shown that, at random initialization of the network parameters θ, the output function of a neural network of depth L with nonlinearity σ, converges to a Gaussian process, called the NNGP, as the width of its layers $n_1, \ldots, n_L \to \infty$ [Lee et al., 2018a, Neal, 1994]. This Gaussian process has covariance recursively defined by

$$\begin{aligned}\Sigma^{(1)}(x, \tilde{x}) &= \frac{1}{n_0} x^T \tilde{x} + \beta^2 \\ \Sigma^{(L+1)}(x, \tilde{x}) &= \mathbb{E}_{f \sim \mathcal{N}(0, \Sigma^{(L)})}\left[\sigma(f(x))\sigma(f(\tilde{x}))\right] + \beta^2,\end{aligned}$$

where β gives the variance of the bias terms in the neural network layers.

This result, though interesting, does not say much on its own about the behavior of *trained* neural networks. This role is left to the NTK, which is defined as the kernel given by

$$\Theta(x, \tilde{x}) = \left\langle \nabla_\theta f_\theta(x), \nabla_\theta f_\theta(\tilde{x}) \right\rangle.$$

It can be shown that this kernel can be written out as a recursive expression involving the NNGP (see Theorem 7.1).

Importantly, Jacot et al. [2018] demonstrated that, again as the network layer widths $n_1, \ldots, n_L \to \infty$, the NTK is (1) deterministic at initialization and (2) constant throughout training. Finally, it has also been demonstrated that, as the learning rate for the stochastic gradient descent (SGD) algorithm tends to 0, the output function of the trained neural network f_θ converges to the kernel regression solution using the NTK [Arora et al., 2019, Lee et al., 2020]. In other words, under the assumptions above, the behavior of a trained deep neural network can be modeled as kernel regression using the NTK.

In Chapter 7, we derive the NNGPs and NTKs for sinusoidal networks and use the findings, together with the NTK theory described above, to inform our understanding of how these networks behave.

Part I

Learning with Differentiable Physics Layers

Chapter 3

Learning and Control with Differentiable Rigid Body Dynamics

In this chapter, we present how to develop a differentiable rigid body physics engine and integrate it into a deep learning model as a layer that is hard-constrained to follow the underlying dynamics, which are fully specified. This allows for models that are significantly more data-efficient than their corresponding "physics-agnostic" deep learning counterparts.

3.1 Introduction

Physical simulation environments, such as MuJoCo [Todorov et al., 2012], Bullet [Coumans et al., 2013], and others, have played a fundamental role in developing intelligent reinforcement learning agents. Such environments are widely used, both as benchmark tasks for RL agents [Brockman et al., 2016], and as "cheap" simulation environments that can (ideally) allow for transfer to real domains. However, despite their ubiquity, these simulation environments are in some sense poorly suited for deep learning settings: the environments are not natively *differentiable*, and so gradients (e.g., policy gradients for control tasks, physical property gradients for modeling fitting, or dynamics Jacobians for model-based control) must all be evaluated via finite differencing, with some attendant issues of speed and numerical stability. Recent work has also proposed the development of a differentiable physical simulator [Degrave et al., 2016], but this was accomplished by simply writing the simulation engine entirely in an automatic differentiation framework; the limitations of this framework meant that the system only supported balls as objects, with limited extensibility.

In this chapter, we propose and present a differentiable two-dimensional physics simulator that addresses the main limitations of past work. Specifically, like many past simulation engines, our system simulates rigid body dynamics via a linear complementarity problem (LCP) [Cline, 2002, Cottle, 2008], which computes the equations of motion subject to contact and friction constraints. In addition to this, however, we also show how to differentiate, analytically, through the *optimal solution* to the LCP; this allows us to use general simulation methods for determining the non-differentiable parts of the dynamics (namely, the presence or absence of collisions between convex shapes), while still providing a

simulation environment that is end-to-end differentiable (given the observed set of collisions). The end result is that we can embed an entire physical simulation environment as a "layer" in a deep network, enabling agents to both learn the parameters of the environments to match observed behavior *and* improve control performance via traditional gradient-based learning. We highlight the utility of this system in a wide variety of different domains, each highlighting a different benefit that such differentiable physics can bring to deep learning systems: learning physical parameters from data; simulating observed (visual) behavior with minimal data requirements; and learning physical deep RL tasks, ranging from pure physical systems like Cartpole to "physics based" like Atari *Breakout*, via gradient planning methods. The environment itself is implemented as a function within popular the PyTorch library [Paszke et al., 2017]. Code for the engine and experiments is available at `https://github.com/locuslab/lcp-physics`.

3.2 Differentiable Physics Engine

A detailed description of the physics engine architecture is omitted here due to space constraints. This description, the LCP solution and the gradients are presnted in detail in Appendix A. Below we present a brief summary of the LCP formulation.

3.2.1 Formulating the LCP

Rigid body dynamics are commonly formulated as a linear complementarity problem, with the different constraints on the movement of bodies (such as joints, interpenetrations, friction, etc.) represented as equality and inequality constraints [Anitescu and Potra, 1997, Cline, 2002]. In this work, we follow closely the framework described in Cline [2002], in which at each time step an LCP is solved to find the constrained velocities of the objects.

To formulate such an LCP, we first find which contacts between bodies are present at the current time-step. Let t be the current time-step and $t + dt$ the following time-step, for a step of size dt. If the distance between possibly contacting objects is less than a predefined threshold, the interaction is considered a contact. From the equality constraints specified in the system, such as joints, we can build the matrix $\mathcal{J}_e$ such that $\mathcal{J}_e v_{t+dt} = 0$. From the contacts at each step, we can build a contact constraint matrix $\mathcal{J}_c$, such that $\mathcal{J}_c v_{t+dt} \geq 0$. Similarly, we have a friction constraint matrix $\mathcal{J}_f$ that introduces frictional interactions. From the definition of the simulated bodies we construct the inertia matrix $\mathcal{M}$. Finally, given the forces acting on the bodies at time t, f_t, and the collision coefficient c, the constrained dynamics can be formulated as the following mixed LCP

$$
\begin{bmatrix} 0 \\ 0 \\ a \\ \sigma \\ \zeta \end{bmatrix} - \begin{bmatrix} \mathcal{M} & -\mathcal{J}_e & -\mathcal{J}_c & -\mathcal{J}_f & 0 \\ \mathcal{J}_e & 0 & 0 & 0 & 0 \\ \mathcal{J}_c & 0 & 0 & 0 & 0 \\ \mathcal{J}_f & 0 & 0 & 0 & E \\ 0 & 0 & \mu & -E^T & 0 \end{bmatrix} \begin{bmatrix} v_{t+dt} \\ \lambda_e \\ \lambda_c \\ \lambda_f \\ \gamma \end{bmatrix} = \begin{bmatrix} \mathcal{M}v_t + dt f_t \\ 0 \\ c \\ 0 \\ 0 \end{bmatrix} \tag{3.1}
$$

$$
\text{subject to} \begin{bmatrix} a \\ \sigma \\ \zeta \end{bmatrix} \geq 0, \begin{bmatrix} \lambda_c \\ \lambda_f \\ \gamma \end{bmatrix} \geq 0, \begin{bmatrix} a \\ \sigma \\ \zeta \end{bmatrix}^T \begin{bmatrix} \lambda_c \\ \lambda_f \\ \gamma \end{bmatrix} = 0,
$$

where $[a, \sigma, \zeta]^T$ are slack variables for the inequality constraints, and $[v_{t+dt}, \lambda_e, \lambda_c, \lambda_f, \gamma]^T$ are the unknowns. By solving this LCP, we obtain the velocities for the next time-step v_{t+dt}, which are used to update the positions of the bodies.

3.2.2 Solving the LCP

Analogously to the differentiable optimizer in OptNet [Amos and Kolter, 2017], our LCP solver is adapted from the primal-dual interior point method described in Mattingley and Boyd [2012]. The advantage of using such a method is that it allows for efficient computation of the gradients, as we show in Section 3.2.3.

First, to simplify the notation from the LCP formulation of the dynamics in Equation 3.1, let us define

$$
\begin{aligned}
x &:= -v_{t+dt} & q &:= -\mathcal{M}v_t - dt f_t & s &:= \begin{bmatrix} a \\ \sigma \\ \zeta \end{bmatrix} & & \\
y &:= \lambda_e & A &:= \mathcal{J}_e & & & F &:= \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & E \\ \mu & -E^T & 0 \end{bmatrix}. \\
z &:= \begin{bmatrix} \lambda_c \\ \lambda_f \\ \gamma \end{bmatrix} & G &:= \begin{bmatrix} \mathcal{J}_c & 0 \\ \mathcal{J}_f & 0 \\ 0 & 0 \end{bmatrix} & m &:= \begin{bmatrix} c \\ 0 \\ 0 \end{bmatrix} & &
\end{aligned}
$$

Then we can rewrite the LCP above as the system below, which can be solved with only slight adaptations to the primal-dual interior point method by [Mattingley and Boyd, 2012].

$$
\begin{bmatrix} 0 \\ s \\ 0 \end{bmatrix} + \begin{bmatrix} \mathcal{M} & G^T & A^T \\ G & F & 0 \\ A & 0 & 0 \end{bmatrix} \begin{bmatrix} x \\ z \\ y \end{bmatrix} = \begin{bmatrix} -q \\ m \\ 0 \end{bmatrix} \tag{3.2}
$$

$$
\text{subject to } s \geq \mathbf{0},\ z \geq \mathbf{0},\ s^T z = 0.
$$

3.2.3 Gradients

All the work leading to the construction of the dynamics LCP in Equation 3.1 consists of differentiable operations on the simulations parameters and initial setting. Therefore, if we could differentiate through the solution for the LCP as well, the system would be differentiable end to end. To derive these gradients we apply the method described in

[Amos and Kolter, 2017] to the LCP in 3.2, which gives us the gradients of the solution of the LCP with respect to the input parameters from the previous time-step. By following this method we arrive at the partials that can then be used for the backward step

$$\begin{aligned} \frac{\partial \ell}{\partial q} &= -d_x & \frac{\partial \ell}{\partial \mathcal{M}} &= -\frac{1}{2}(d_x x^T + x d_x^T) \\ \frac{\partial \ell}{\partial m} &= D(z^\star) d_z & \frac{\partial \ell}{\partial G} &= -D(z^\star)(d_z x^T + z d_x^T) \\ \frac{\partial \ell}{\partial A} &= -d_y x^T - y d_x^T & \frac{\partial \ell}{\partial F} &= -D(z^\star) d_z z^T. \end{aligned} \tag{3.3}$$

3.2.4 Implementation

The physics engine is implemented in PyTorch [Paszke et al., 2017] in order to take advantage of the autograd automatic differentiation graph functionality. The LCP solver is implemented as an autograd Function, with the analytical gradients provided according to the definitions above. This allows the derivatives to be propagated across time-steps in the simulation. Furthermore, the autograd graph then allows the derivatives to be propagated backwards into the leaf parameters of the dynamics, such as the bodies' masses, positions, etc.

3.3 Experiments

To demonstrate the flexibility of the differentiable physics engine, we test its performance across three classes of experiments. First, we show that it can infer the mass of an object by observing the dynamics of a scene. Next, we demonstrate that embedding a differentiable physics engine within a deep autoencoder network can lead to high accuracy predictions and improved sample efficiency. Finally, we use the differentiable physics engine together with gradient-based control methods to show that we can learn to perform physics-based tasks with low sample complexity when compared to model-free methods.

3.3.1 Parameter learning

Task To evaluate the engine's capabilities for inference, we devised an experiment where one object has unknown mass which has to be inferred from its interactions with the other bodies. As depicted in Figure 3.1, a scene in which a ball of known mass hits a chain is observed and the resulting positions of the objects are recorded for $10s$. The goal is to infer the mass of the chain.

Learning and results Simulations are iteratively unrolled starting with an arbitrarily chosen mass of 1 for the chain. After each iteration, the mean squared error (MSE) between the observed positions and the simulated positions is observed, and then used to obtain its gradient with respect to the mass. Gradients are clipped to a maximum absolute value of 100 and then used to perform gradient descent on the value of the mass, with a learning

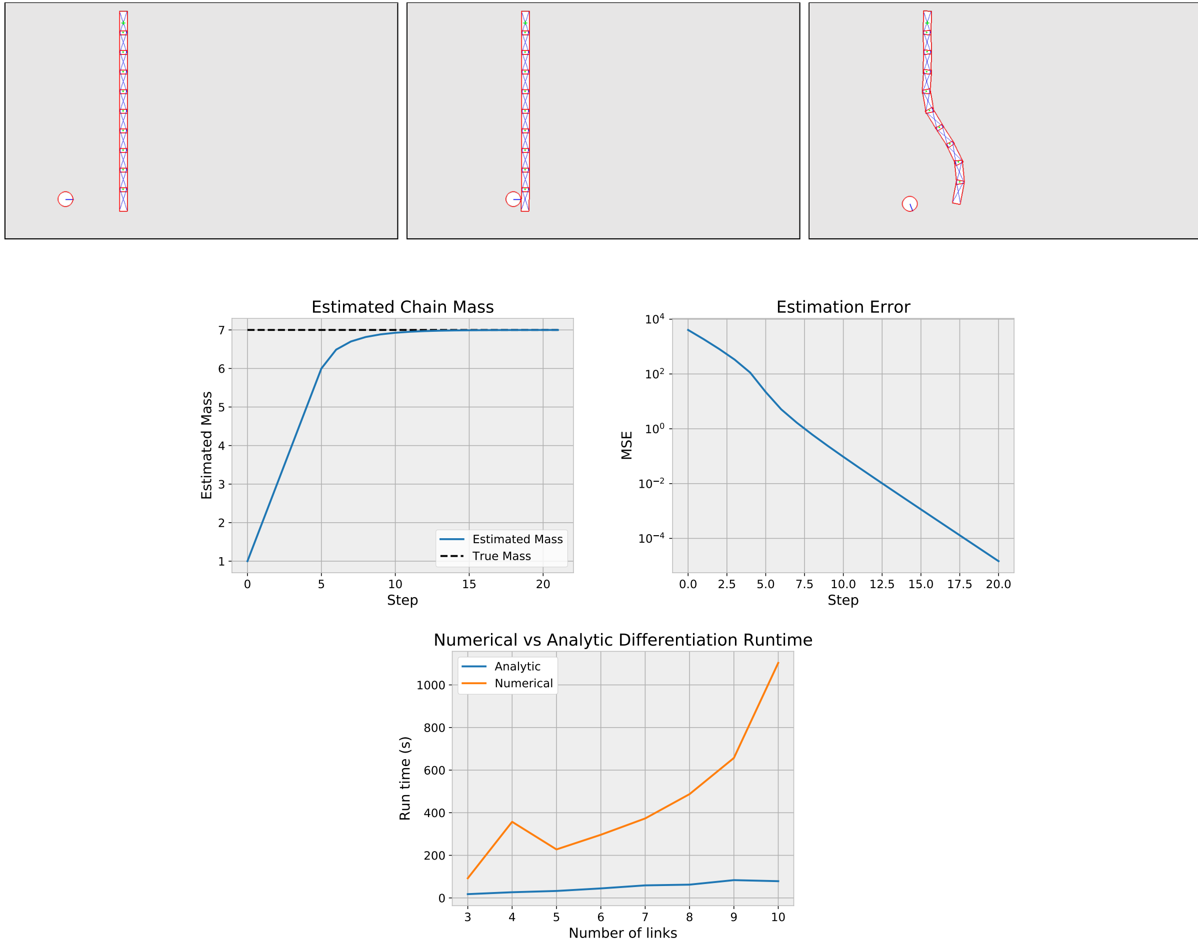


Figure 3.1: Inferring the mass of a chain. *Top*: Sequence of frames from the inference experiment. The goal is to infer the mass of the chain by unrolling simulations and using the gradient to minimize the loss from the predicted positions to the observed ones. *Bottom left*: The estimated mass quickly converges to the true value, $m = 7$, indicated by the dashed line. *Bottom center*: As a consequence of the improving mass estimation, the MSE (represented in log scale) between the true and simulated positions for the bodies decreases quickly. *Bottom right*: Run time comparison between using analytical gradients or finite differences for 30 updates, as a function of the number of links in the chain.

rate of 0.01. As shown in Figure 3.1 this process is able to quickly reduce the position MSE by converging to the true value of the mass.

Comparison to numerical derivatives We also compared using analytic and numerical gradients. In this experiment, the same optimization process described above was repeated for a varying number of links in the chain. The number of gradient updates was fixed to 30 and the run times were averaged over 5 runs for each condition. As can be seen in Figure 3.1, the run time with analytical gradients grows much more slowly with the increasing number of parameters.

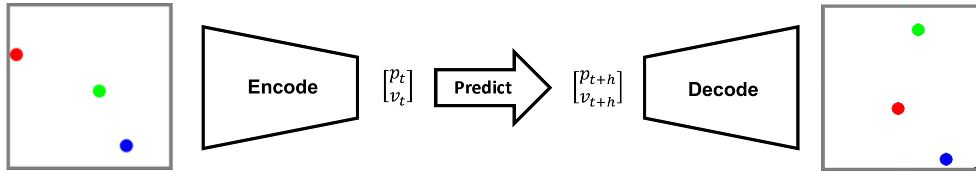


Figure 3.2: Diagram of autoencoder architecture. The encoder learns to map from input frames to the physical state of the objects (*i.e.*, position, velocity, etc.). The physics engine steps the world forward using the parameters from the encoder. The decoder takes the predicted physical parameters and generates a frame to match the true future frame. The system is trained end-to-end. Part of the labels have strong supervision, with ground truth values available for the output of the encoder and physics engine. Different proportions of strong and weak supervision (only the future frame is provided) in the data are evaluated. Using a large number of weakly labelled data improves sample efficiency for strongly labelled data.

3.3.2 Prediction on visual data

Task To test our approach on a benchmark for visual physical prediction, we generated a dataset of videos of billiard ball-like scenarios using the code from [Fragkiadaki et al., 2015]. Simulations lasting 10 seconds were generated, totalling 8,000 trials for training, 1,000 for validation and 1,000 for testing. Datasets with 1 and 3 balls were used, with all balls having the same mass. Frames from sample trials can be seen in Figure 3.3. In our task setup, balls bouncing in a box are observed for a period of time. The model is provided with 3 frames as input and has to learn to predict the state of the world at a future state, 10 frames later.

Architecture To make visual prediction given the visual input, we use an autoencoder architecture summarized in Figure 3.2. It consists of three parts: (1) the *encoder* maps input frames to the physical state of the objects (*i.e.*, position, velocity, etc.). Specifically, we take in a sequence of 3 RGB frames from the simulation. We then use a pretrained spatial pyramid network [Ranjan and Black, 2016] to obtain two optical flow frames (each consisting of two matrices, for x and y flow). Color filters are applied to the RGB images to segment the objects. The segmented region of each object is then used as a mask for the RGB and optical flow frames, such that at the end of this pipeline we have, for each object, a collection of 3 RGB frames and 2 optical flow frames (13 channels) with only a single segmented object. Then, each of these per-object processed inputs is passed to a VGG-11 network with its last layer modified to output size 4, in order to regress two position and two velocity parameters as outputs. (2) the *physics engine* steps the world forward using the physical parameters received from the encoder. The physics engine can be integrated into the autoencoder pipeline and allow for end-to-end training due to its differentiability, as described in Section 3.2. (3) the *decoder* takes the predicted physical parameters and generates a frame to match the true future frame. The architecture used is a mirror of the VGG encoder network, with transposed convolutions in the place of convolutions and

bilinear upsampling layers in the place of the maxpooling ones.

Learning In order to evaluate the sample efficiency of the model, the network was trained with varying amounts of labelled samples. The labels used consist of the ground truth physical parameters ϕ of the objects both at the present (ϕ_t) and the future time-step (ϕ_{t+dt}). When a label is available for a given sample, the model uses these ground truth physical parameters (instead of the estimated ones) to generate the predicted frame $\hat{y}$ from input frames x, such that

$$\hat{\phi}_t = encoder(x), \quad \hat{\phi}_{t+dt} = physics(\phi_t), \quad \hat{y} = decoder(\phi_{t+dt}). \tag{3.4}$$

Using the labels and the true future frame y, the model is then trained to minimize a loss consisting of the sum of three terms, the encoder, physics and decoder losses

$$\begin{gathered} \mathcal{L} = \mathcal{L}_{enc} + \mathcal{L}_{phys} + \mathcal{L}_{dec}, \\ \mathcal{L}_{enc} = \ell(\hat{\phi}_t, \phi_t), \;\; \mathcal{L}_{phys} = \ell(\hat{\phi}_{t+dt}, \phi_{t+dt}), \;\; \mathcal{L}_{dec} = \ell(\hat{y}, y), \end{gathered} \tag{3.5}$$

where $\ell(\cdot, \cdot)$ is the mean squared error loss.

When labels are not available for a given sample, the model uses its own estimated parameters to generate the predicted frame, that is

$$\hat{\phi}_t = encoder(x), \quad \hat{\phi}_{t+dt} = physics(\hat{\phi}_t), \quad \hat{y} = decoder(\hat{\phi}_{t+dt}). \tag{3.6}$$

Notice that here, unlike in Equation 3.5, the arguments to the function are estimated ($\hat{\phi}_t$, $\hat{\phi}_{t+dt}$). In this case, since there are no labels to use for the other losses, the loss consists only of $\mathcal{L} = \mathcal{L}_{dec}$. Notice that here the right hand side of the equations use the estimated $\hat{\phi}$. The gradients are thus being propagated end-to-end through the physics model back to the encoder. As shown in Figure 3.4, this signal from unlabeled examples allows the autoencoder to learn with greater sample efficiency. For all losses, the MSE is used. In our experiments, the squared loss performed better than the ℓ_1 loss, which was not able to produce meaningful decoder outputs.

Results As demonstrated in Figure 3.4, the model was able to learn to perform the task with high accuracy. Figure 3.3 contains sample predicted frames and their matching ground truth frame for a qualitative analysis of the results. As a comparison point, an MLP with two hidden-layers of size 100 and trained with only labeled data was used as a baseline, replacing the $physics(\cdot)$ function in Equation 3.4 above. In our experiments, using the baseline model in such a way, as a replacement for the physics function, provided better results than using it in an unstructured manner, relying solely on the decoder loss. It is clear from Figure 3.4 that the autoencoder with the physics model is able to learn more efficiently and with higher accuracy than the baseline model. To evaluate the sample efficiency of this model, we compare its performance on training regimens in which 100%, 25%, 10% and 2% of the available samples containing labels. Some supervision is still necessary, since when provided with no supervision at all (a 0% condition), relying solely on the decoder loss, the model was not able to learn to extract meaningful physical parameters. Still, as can be seen in Figure 3.4, the model is able to leverage the unlabeled data to quickly learn even from few labelled data points.

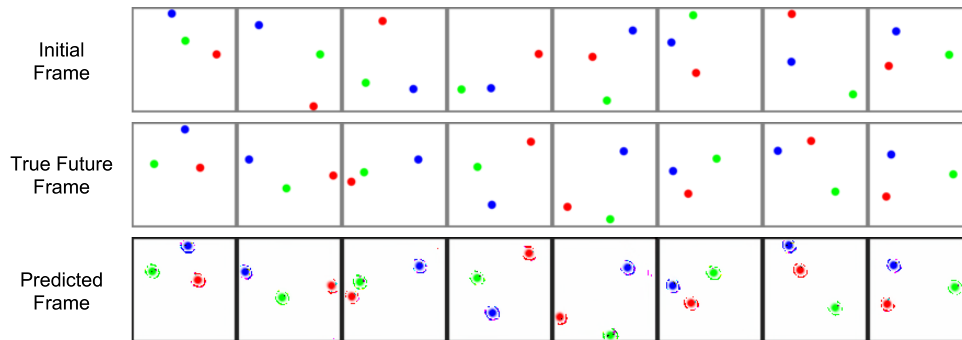


Figure 3.3: Qualitative results for prediction task comparing ground truth and predicted future frame. Only the initial frame and the two preceding frames are used as input, with physical parameters extracted, used to simulated the state forward and generate the predicted frame. In most cases the predicted frame is accurate. However, small differences can still be perceived in some cases, due to differences between the two engines.

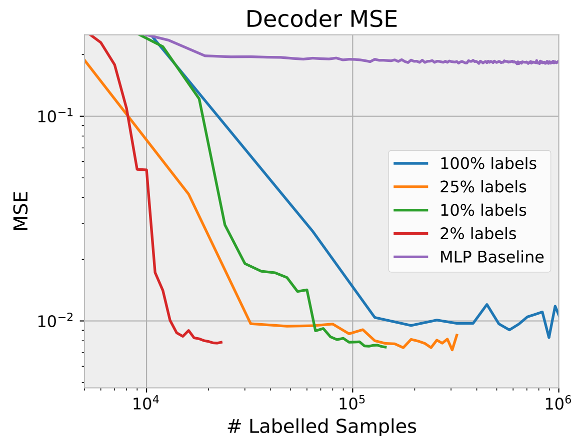


Figure 3.4: Sample efficiency for the prediction task measured by the validation loss per number of labelled samples used in training. The autoencoder is able to leverage unlabelled examples to improve its sample efficiency: training regimes that employ unlabeled data learn faster for a given amount of labeled data. The loss is the mean squared error of the predicted image to the ground truth. Each line represents a training regiment with a different proportion of labeled to unlabeled data. Note that the x-axis is already adjusted to the number of labeled samples used, to facilitate comparisons.

3.3.3 Control

Tasks Finally, in this section we demonstrate the physics engine ability to be readily used with gradient-based control methods. To this end, we evaluate its performance on physics based tasks from the OpenAI Gym's environment [Brockman et al., 2016], namely *Cartpole* and the Atari game *Breakout*.

Model and Controller For the *Cartpole* environment, a model is built using two articulated rectangles, whose dimensions and mass are learned from simulated trajectories using random actions. The physics engine-based model is compared to a baseline consisting of an MLP with two hidden layers of size 100 trained on the same data. A variation of the environment is used in which the actions to be taken by the cart are continuous, instead of discrete. Rewards are also limited to 1000, instead of the default 200 for which the task is considered done.

For *Breakout*, a model of the environments is built by applying color filters, segmenting the diverse objects (the paddle, the ball, etc.) and translating these positions into the physics engine. The ball's velocity is estimated by the difference in its position from the last two frames. The paddle velocity when moving at each step is learned by unrolling game episodes with randomly chosen actions, performing the same actions in the physics simulation and then fitting the simulation parameter via gradient descent to minimize the mean squared error to the observed trajectory, analogously to the process in Section 3.3.1. The physics engine model is compared to a Double Q Learning with prioritized replay [van Hasselt et al., 2015] baseline from OpenAI [Dhariwal et al., 2017].

Since the resulting physics models described above are differentiable, they are used in conjunction with iLQR [Li and Todorov, 2004] to control the agent in the tasks. The iLQR is set up with a time-horizon of 5 frames for both tasks. For *Cartpole* the cost consists of the square of pole's angular deviation from vertical. For the Ataro game the cost consists of the squared difference in the x position of the paddle and the ball when the ball is descending, and the squared distance to the center of the screen otherwise.

Results Results for the *Cartpole* task are shown in Figure 3.5. Even though the MLP baseline achieves a lower MSE faster in predicting the next state of the cartpole system, the physics engine is able to learn parameters for a model that allows for high reward on the task, even when error is higher.

In the Atari benchmark, the system is able to achieve high reward on the task with extremely low sample complexity. Specifically, the model is able to learn the paddle parameters quickly from random trajectories, improving the control precision, and leading to high reward, as shown in Figure 3.6 for *Breakout*. The model performs close to model-free reinforcement learning methods and is able to achieve a high level of reward with orders of magnitude fewer samples.

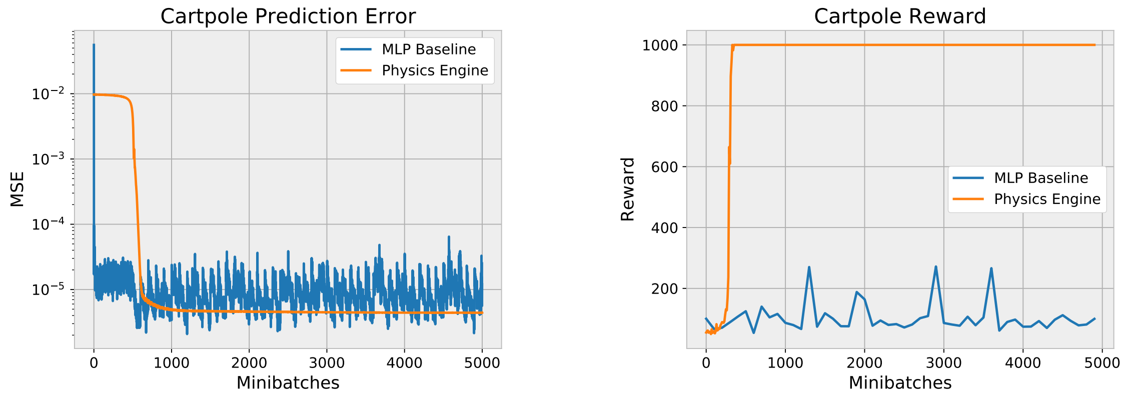


Figure 3.5: Even though the baseline is able to achieve lower MSE over one-step predictions of the dynamics of the *Cartpole* environment (*left*), the physics engine-based controller is able to achieve a higher reward very quickly (*right*).

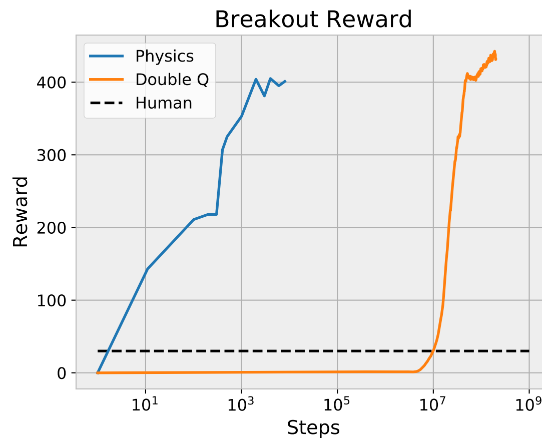


Figure 3.6: The physics based controller is able to quickly learn a good parameter values that lead to high reward. Even though the asymptotic performance is lower than the model-free method, it achieves a high level of reward with orders of magnitude data (the horizontal axis is log-scaled). Human level of 31 for a professional game tester was used, as per [Mnih et al., 2015].

Chapter 4

Fluid Flow Prediction with Graph Neural Networks and Differentiable Fluid Dynamics

In this chapter, we again present the usage of a differentiable physics simulation as a layer in a deep learning model, but here in the domain of fluid dynamics. Additionally, we also demonstrate the utilization of graph-based deep learning architectures, which allow the integration of additional inductive biases in the form of unstructured meshes, which are used to generate fluid flow fields. These allows the learned model to perform much stronger generalization to samples from outside the distribution seen during training.

4.1 Introduction

Several recent works have explored the application of deep models to approximate the solutions to partial differential equations (PDEs), particularly in the context of simulating fluid dynamics [Afshar et al., 2019, Guo et al., 2016, Um et al., 2017, Wiewel et al., 2018]. The behavior of fluids is a well-studied problem in the physical sciences, and predicting their dynamics involves solving the nonlinear Navier-Stokes PDEs. In order to perform computational fluid dynamics (CFD) simulations, these equations must be solved numerically. One of the primary bottlenecks in more accurate and advanced CFD simulation is specifically the time it takes to run these models. It is not uncommon for a single simulation to take many days to weeks on massive supercomputing infrastructure. Especially for the cases where fast iteration time is desired, for example when iterating over different aerodynamic design prototypes for a structure, then faster, learning-based surrogate models have spawned a great deal of interest. However, despite the recent enthusiasm for this area, most deep learning models cannot capture the full complexity of the underlying equations, and, as we demonstrate in this work, can quickly start to produce poor results when testing on settings well outside the domain of their training data.

In this chapter, we explore a hybrid approach that combines the benefits of (graph) neural networks for fast predictions, with the physical realism of an industry-grade CFD

simulator. Our system has two main components. First, we construct a graph convolution network Kipf and Welling [2016] (GCN), which operates directly upon the non-uniform mesh used in typical CFD simulation tasks. This use of GCNs is crucial because all realistic CFD solvers operate on these unstructured meshes rather than directly on the regular grid used by most prior work, which has typically used convolutional networks to approximate CFD simulations. Second, and more fundamentally, we embed a (differentiable) CFD solver, operating on a much coarser resolution, *directly* into the GCN itself. Although typically treated as black-boxes, modern CFD simulators are themselves perfectly well-suited to act as (costly) "layers" in a deep network. Using well-studied adjoint methods, modern solvers can compute gradients of the output quantities of a simulation with respect to the input mesh. This allows us to integrate a fast CFD simulation (made fast because it is operating on a much smaller mesh) into the network itself, and allows us to *jointly* train the GCN and the mesh input into the simulation engine, all in an end-to-end fashion.

We demonstrate that this combined approach performs substantially better than the coarse CFD simulation alone (i.e., the network is able to provide higher fidelity results than simply running a faster simulation to begin with), *and* generalizes to novel situations much better than a pure graph-network-based approach. Moreover, the approach is still substantially faster than running the CFD simulation on the original size mesh itself. We believe that in total this represents a substantial advance towards integrating deep learning and existing state-of-the-art simulation software.

4.2 CFD-GCN

Here we describe the general outline of our hybrid CFD simulation and graph neural network approach. Based on this hybrid nature, we refer to our model as CFD-GCN. We first describe its broad architecture and then its different components in detail. Finally, we describe the procedures used to train the network itself.

4.2.1 Architecture

The overall architecture of the CFD-GCN is shown in Figure 4.1. Intuitively, the network operates over two different graphs, a "fine" mesh over which to compute the CFD simulation, and a "coarse" mesh (initially a simple coarsened version of the fine mesh, but eventually tuned by our model) that acts as input to the CFD solver. As input, the network takes a small number of parameters that govern the simulation. For the case of the experiments in this work, in which we predict the flow fields around an airfoil, these parameters are the Angle of Attack (AoA) and the Mach number. These parameters are provided to the CFD simulation and are also appended to the initial GCN node features. Although this may seem to be a relatively low-dimensional task, even these two components can vary the output of the simulation drastically and are difficult for traditional models to learn when generalizing outside the precise range of values used to "train" the network. The network operates by first running a CFD simulation on the coarse mesh, while simultaneously processing the graph defined by the fine mesh with GCNs. It then upsamples the results of

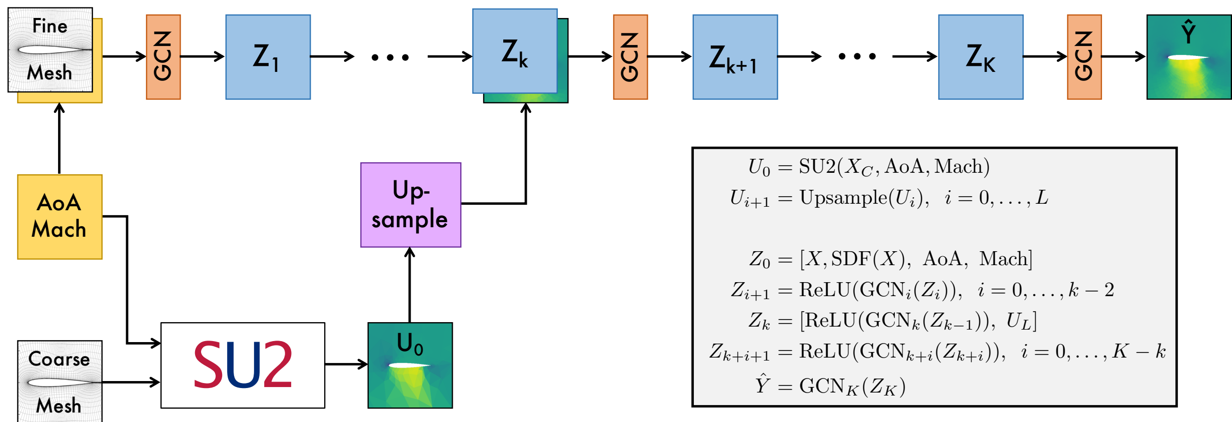


Figure 4.1: A diagram of the CFD-GCN model and its corresponding equations.

the simulation, and concatenates these with an intermediate output from a GCN. Finally, it applies additional GCN layers to these joint features, ultimately predicting the desired output values (in this case, the velocity and pressure fields at each node in the fine mesh). We now describe each of these components in detail.

Graph structure and network input. The graph structure we use for the CFD-GCN is directly derived from the mesh structure used by traditional CFD software to simulate the physical system. Specifically, we consider a two-dimensional, triangular mesh $M = (X, E, B)$. The first element, $X \in \mathbb{R}^{N \times 2}$, is a matrix containing the (x, y) coordinates of the N nodes that compose the mesh. The second,

$$E = \{(i_1, j_1, k_1), \dots, (i_M, j_M, k_M)\},$$

is a set of M triangular elements defined by the indices (i, j, k) of their component nodes. The third,

$$B = \{(i_1, b_1), \dots, (i_L, b_L)\},$$

is a set of L boundary points, defined as a pair consisting of the index of the node and a tag b that identifies which boundary the point belongs to (*e.g.* airfoil, farfield, etc.).

Such a mesh M clearly defines a graph $G_M = (X, E_G)$ whose nodes are the same X, and whose edges E_G can be directly inferred from the mesh elements E. Conversely, a graph can also be converted into a mesh if the structure of its edges is appropriate and a set of boundary points B is provided.

In addition to the fine mesh used to compute the CFD simulation, we also consider a coarse mesh, denoted M_C. This mesh has the same structure as the fine mesh M, with the number of nodes downsampled by almost 20x, which thus allows for much faster simulation. Although this mesh also technically defines a graph, we do not directly compute any GCN over this graph, but instead only use it as input to the simulation engine.

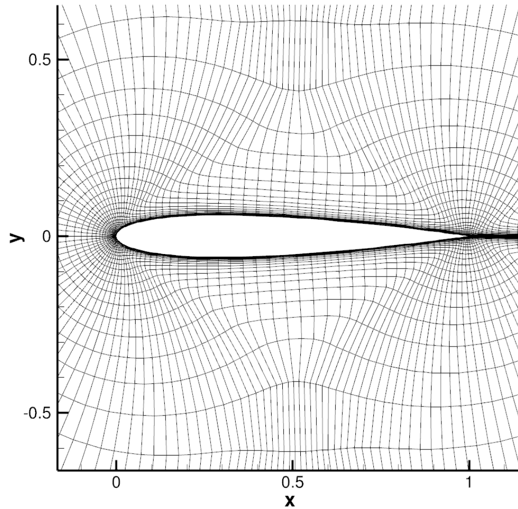


Figure 4.2: A NACA0012 mesh, zoomed in on the airfoil region.

In addition to the graphs themselves, the model also receives as input two physical parameters that define the behavior of the flow around an airfoil: the angle of attack (AoA) and the Mach number. These two parameters are both fed into the simulation and appended as initial node features for every node in the GCN. These two parameters ultimately are the quantities that vary from simulation-to-simulation, and thus the main task of the GCN is to learn how to predict the resulting flow field from these two parameters that define the simulation. Even thought this input space is low dimensional, it still defines a complex task, since varying these parameters gives rise to diverse behaviors of the fluid flows, as demonstrated, for example, in our generalization experiment (Section 4.3.2).

The SU2 Fluid Simulator. A central component of the CFD-CGN model is the integrated differentiable fluid dynamics simulator. As input, the fluid simulator takes coarse mesh M_C, plus the angle of attack and Mach number, and outputs predictions of the velocity and pressure at each node in the coarse graph. We specifically employ the SU2 fluid simulator [Economon et al., 2015b], an open source, industry-grade CFD simulation widely used by many researchers in aerospace and beyond. Briefly, SU2 uses a finite volume method (FVM) to solve the Navier-Stokes equations over its input mesh. Crucially for our purposes, the SU2 solver also support an adjoint method which lets us differentiate the outputs of the simulation with respect to its inputs and parameters (in this case, the coarse mesh M_C itself, plus the angle of attack and Mach number).

Intuitively, the SU2 solver should be thought of as an additional layer in our network, which takes the angle of attack and Mach number as input, and produces the output velocity and pressure fields. The equivalent of the "parameters" of a traditional layer is the coarse mesh itself: different configurations for the coarse mesh will be differently suited to integration within the remainder of the CFD-GCN. Thus, the main learning task for

the SU2 portion of our model is to *adjust* the coarse mesh in a manner than eventually maximizes accuracy of the resulting full CFD-GCN model. The adjoint method in SU2 uses reverse-mode differentiation, so gradients can be efficiently computed with respect to a scalar-valued loss such as the overall predictive error of the CFD-GCN.

Finally, although not strictly a research contribution, we want to mention that as part of this project we have developed an interface layer between the SU2 solver and the PyTorch library. This interface allows full SU2 simulations to be treated just as any other layer within a PyTorch module, and we hope it will find additional applications at the intersection of deep learning and (industrial-grade) CFD simulation. The code for the work presented in this chapter can be found at `https://github.com/locuslab/cfd-gcn`.

Upsampling. The output of the coarse simulation described above is a mesh with the predicted values for each field at every node. For this to be used towards generating the final prediction, we need to upsample it to the size of the fine mesh. We do this by performing successive applications of squared distance-weighted, k-nearest neighbors interpolation [Qi et al., 2017].

Let us call $U \in \mathbb{R}^{N_U \times 3}$ the upsampled version of some coarser graph $D \in \mathbb{R}^{N_D \times 3}$. For every row $U^{(i)}$, with corresponding node position $X_U^{(i)}$, we find the set $\{n_1, \ldots, n_k\}$ containing the indices of the k closest nodes to $X_U^{(i)}$ in the coarser graph X_D. Then, we define $U^{(i)}$ as

$$U^{(i)} = \frac{\sum_{j=1}^{k} w(n_j) D^{(n_j)}}{\sum_{j=1}^{k} w(n_j)},$$

where

$$w(c_j) = \frac{1}{\|X_U^{(i)} - X_D^{(n_j)}\|_2^2}.$$

As a default, we set $k = 3$.

Graph Convolutions. As depicted on Figure 4.1, the output of the coarse simulation is processed by a sequence of convolutional layers. In order to operate directly on the mesh output of the CFD simulation, we utilize the graph convolutional network (GCN) architecture from Kipf and Welling [2016]. This architecture defines a generalized convolutional layer for graphs.

A general graph consisting of N_Z nodes, each with F features, is defined by its feature matrix $Z_i \in \mathbb{R}^{N_Z \times F}$ and its adjacency matrix $A \in \mathbb{R}^{N_Z \times N_Z}$. We can then further define $\tilde{B} = \tilde{D}^{-\frac{1}{2}}(A + I)\tilde{D}^{-\frac{1}{2}}$, where I is the identity matrix and $\tilde{D}$ the diagonal degree matrix, with its diagonal given by $\tilde{D}_{ii} = 1 + \sum_{j=0}^{N_Z} A_{ij}$. Then, a GCN layer with F input channels and F' output channels, parameterized by the weight matrix $W \in \mathbb{R}^{F \times F'}$ and the bias term $b \in \mathbb{R}^{N_Z \times F'}$, followed by a ReLU non-linearity, will have as output

$$\begin{aligned} \tilde{Z}_{i+1} &= \tilde{B} Z_i W_i + b_i \equiv \mathrm{GCN}_i(Z_i). \\ Z_{i+1} &= \mathrm{ReLU}(\tilde{Z}_{i+1}) \end{aligned}$$

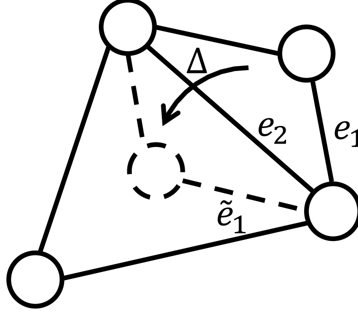


Figure 4.3: For certain updates to the mesh (Δ), a node might be pushed over the edge of its triangular element, generating overlap of elements. These non-physical situations harm convergence of the simulation. When this happens, the cross product between ordered edges changes. Before the update, $e_1 \times e_0 > 0$, while afterwards $\tilde{e}_1 \times e_2 < 0$.

CFD-GCN. With all the components of the CFD-GCN desribed above, we can now bring them all together to describe the full pipeline depicted in Figure 4.1.

First, an SU2 simulation is run with the coarse mesh and the physical parameters. The output of this coarse simulation is upsampled L times.

$$\begin{aligned} U_0 &= \mathrm{SU2}(X_C, \mathrm{AoA}, \mathrm{Mach}) \\ U_{i+1} &= \mathrm{Upsample}(U_i), \quad i = 0, \dots, L. \end{aligned} \tag{4.1}$$

Concurrently, the fine mesh has the physical parameters and the signed distance function (SDF) appended to each of its nodes' features. The resulting graph is then passed through a series of graph convolutions. At some specified convolutional layer k, the final upsampled value U_L is appended to the output Z_k of the k-th convolution. Another set of convolutions is performed in order to generate the final prediction $\hat{Y}$

$$\begin{aligned} Z_0 &= [X, \mathrm{SDF}(X),\ \mathrm{AoA},\ \mathrm{Mach}] \\ Z_{i+1} &= \mathrm{ReLU}(\mathrm{GCN}_i(Z_i)), \quad i = 0, \dots, k-2 \\ Z_k &= [\mathrm{ReLU}(\mathrm{GCN}_k(Z_{k-1})),\ U_L] \\ Z_{k+i+1} &= \mathrm{ReLU}(\mathrm{GCN}_{k+i}(Z_{k+i})), \quad i = 0, \dots, K-k \\ \hat{Y} &= \mathrm{GCN}_K(Z_K). \end{aligned} \tag{4.2}$$

Here, $[\cdot, \cdot]$ is the matrix concatenation operation over the column dimension.

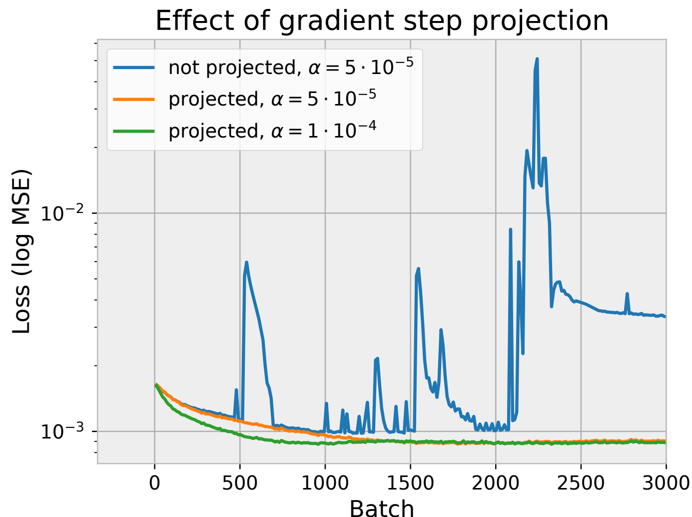


Figure 4.4: Optimizing a mesh without correcting the gradient update to prevent degeneration causes the training to diverge. Meshes trained using the projected gradient step (avoiding non-physical elements) learn smoothly, even with a higher learning rate α.

4.2.2 Training

Given that the entire CFD-GCN as formulated above can be treated as a single differentiable deep network (including the SU2 "layer" discussed above), the training process itself is largely straightforward. The model is trained to predict the output fields $Y \in \mathbb{R}^{N\times 3}$, consisting of the x and y components of the velocity and the pressure at each node in the fine mesh, by minimizing the mean squared error (MSE) loss ℓ between the prediction $\hat{Y}$ and ground truth

$$\ell(Y, \hat{Y}) = \frac{1}{3N} \|Y - \hat{Y}\|_2^2,$$

where the ground truth Y in this case is obtained by running the full SU2 solver to convergence on the original fine mesh.

The training procedure optimizes the weight matrices W_i and b_i of the GCNs, and the positions of the nodes in the coarse mesh X_C by backpropagating through the CFD simulation. The loss is minimized using the Adam optimizer [Kingma and Ba, 2014a] with a learning rate $\alpha = 5 \cdot 10^{-5}$.

Mesh degeneration. An issue arises when optimizing the input coarse mesh. Gradually, as the node positions are moved by the gradient descent updates, it is possible that, in a given triangular element, one of its nodes crosses over an edge (see Figure 4.3). This generates non-physical volumes, which harm the stability of the simulations, frequently impeding convergence. In other words, at each gradient update step, our optimizer updates

the mesh nodes by performing the update

$$X_C \leftarrow X_C + \Delta X_C,$$

with some small update matrix ΔX_C of the same shape as X_C. If left unmodified, this ΔX_C can cause the aforementioned issue.

In order to avoid this, we seek to generate a projected update $P(\Delta X_C)$ such that only non-degenerating updates are performed. We start with $P(\Delta X_C) = \Delta X_C$. Then, we check which elements in the mesh have a node pushed over an edge by ΔX_C. This can be done by computing the cross product of two edges in each triangular element in a consistent order. If the sign of the cross product flips with the update $X_C + \Delta X_C$, that means a node crossed over an edge (since this causes the ordering of the nodes to change). This is depicted in Figure 4.3, where the cross product of the edges e_1 and e_2 is positive before the update, but negative afterwards.

For every element $E = (i, j, k)$ which has flipped, we set the rows i, j and k of $P(\Delta X_C)$ to 0, thus performing no updates to those points in X_C. Since removing the updates to some nodes might cause new elements to flip, this procedure is repeated until no points are flipped. Once we reach this state, we perform the projected gradient update

$$X_C \leftarrow X_C + P(\Delta X_C).$$

In Figure 4.4 we see the results of optimizing the nodes of a mesh to improve a prediction loss both with and without the correction to the gradient update. Whereas the mesh optimized without the correction quickly degenerates and the loss diverges, the one with the projected gradient update learns smoothly, even for a higher learning rate α.

4.3 Experiments

For all experiments we use the NACA0012 airfoil, represented as a fine mesh with 6648 nodes Figure 4.2). The coarse mesh for the same airfoil has 354 nodes. Both meshes are mixed triangular and quadrilateral meshes, but for usage with the CFD-GCN model the coarse mesh is converted to purely triangular by dividing every quadrilateral element in half along a diagonal. All meshes were created using Pointwise Mesh Generation Software[1].

All CFD simulations are performed by solving the0.0 steady-state, compressible, inviscid case of the Navier-Stokes equations (the Euler equations) using SU2. Ground truth simulations on the fine mesh are run to convergence, while simulations on the coarse mesh are run for up to 200 iterations. Sample outputs of simulations with identical physical parameters in each mesh are presented in Figure 4.5.

For the CFD-GCN model, we set $k = 3$, $K = 6$, and $L = 1$. That is, we perform one upsampling step on the coarse simulation, appending it to the third GCN layer, and then perform 3 additional convolutions, for a total of 6 GCN layers. All GCNs are set to have 512 hidden channels. A batch size of 16 is used on all experiments.

[1]https://www.pointwise.com

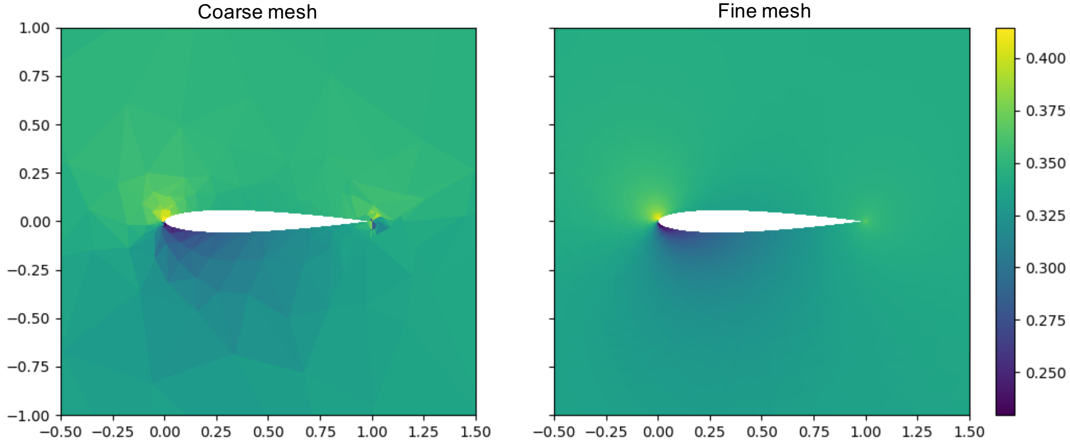


Figure 4.5: Two simulations with identical physical parameters, one with the coarse mesh (left) and one with the fine mesh (right). The pressure component of the output field is presented here. The coarse elements are easily noticeable on the left, whereas on the fine mesh on the right the elements are small enough to be barely visible at this size.

In the experiments, our model is compared to three baselines that can be interpreted as ablated versions of the full CFD-GCN model: the "upsampled coarse mesh" (UCM) baseline, a pure GCN baseline, and the "frozen mesh" version of the CFD-GCN. Each of these demonstrates the importance of each part of the full proposed model. The upsampled coarse mesh baseline consists simply of the part of the model described in Equations 4.1. That is, simply of running the simulation on the coarse mesh and interpolating the output up to the full mesh size. It does not have convolutional layers and it does not perform any learning. The GCN baseline, conversely, consists solely of the GCNs, without the simulation. That is, the part of the model described in Equations 4.2 (without the appended U_L). The GCNs are set to the same parameters as used for the CFD-GCN (6 layers with 512 hidden channels). Finally, the "frozen mesh" version of the CFD-GCN consists of the full CFD-GCN model, with both the GCNs and the coarse simulation, but the gradients through the fluid simulation are not computed, and thus the coarse mesh is not optimized (it is therefore "frozen" through training).

4.3.1 Interpolation

In order to test our model's ability to make accurate flow field predictions, we test its predictions across a range of different physical parameters. We construct training and test sets composed of values for the AoA and Mach number.

The training set is defined by

$$\begin{aligned}
\text{AoA}_{\text{train}} &= \{-10, -9, \ldots, 9, 10\}, \\
\text{Mach}_{\text{train}} &= \{0.2, 0.3, 0.35, 0.4, \\
&\quad\; 0.5, 0.55, 0.6, 0.7\}.
\end{aligned}$$

Similarly, test pairs are sampled from the sets

$$\begin{aligned}
\text{AoA}_{\text{test}} &= \{-10, -9, \ldots, 9, 10\}, \\
\text{Mach}_{\text{test}} &= \{0.25, 0.45, 0.65\}.
\end{aligned}$$

Training pairs are then sampled uniformly from $\text{AoA}_{\text{train}} \times \text{Mach}_{\text{train}}$, and test pairs from $\text{AoA}_{\text{test}} \times \text{Mach}_{\text{test}}$. Here we can see that even though the train and test set are different, the parameters come from similar ranges, and the two sets contain examples with a similar range of qualitative behaviors. This experiment therefore tests the ability of our model to interpolate from parameters seen in training to unseen, yet similar ones at test time. Even though this procedure does not present a strong test of the learning ability of the model, it is a common form of evaluation in many works that apply deep learning methods to CFD (e.g., Afshar et al. [2019], Guo et al. [2016]). We present a stronger test of generalization to new scenarios in our next experiment (Section 4.3.2).

The model takes in as input the pairs and, using the coarse mesh, predicts the three components of the output field, as described in Section 4.2. These predictions are compared against ground truth simulations performed on the fine mesh.

Results are summarized in Table 4.1. A sample prediction is presented in Figure 4.6. We can see from the results that our method outperforms the upsample coarse mesh baseline. This superiority to the upsample coarse mesh baseline demonstrates that the model is not simply upsampling the coarse prediction. The processing done by the GCNs is in fact improving its predictions. We can also observe that the CFD-GCN performs worse than the GCN baseline. The fact that the CFD-GCN underperforms the GCN baseline on the test set is a consequence of the similarity of the settings between training and testing, as we will see in the next experiment. The GCN is capable of overfitting the training set better (as we can see in Figure 4.7) therefore it also performs well on the very similar test set.

4.3.2 Generalization

Depending on the parameter configuration for a given simulation, a “shock” may or may not form around the airfoil. Figure 4.9 presents an example configuration in which we observe a shock. As can be noticed from the figure, these shocks present qualitatively different behavior from the smooth flow fields of “regular” simulations. In this experiment, we aimed to use such a difference in behavior in order to test the generalization capabilities of our model.

We thus constructed a training split such that there were no simulations with a shock present in the training set. To achieve this goal, we used the same data points consisting of

Model	Interpolation (RMSE)	Generalization (RMSE)	Batch Prediction Time (s)
CFD-GCN	$1.8 \cdot 10^{-2}$	$\mathbf{5.4 \cdot 10^{-2}}$	2.0
Frozen Mesh	$1.8 \cdot 10^{-2}$	$6.1 \cdot 10^{-2}$	2.0
Upsampled Coarse Mesh	$4.0 \cdot 10^{-2}$	$7.0 \cdot 10^{-2}$	1.9
GCN	$\mathbf{1.4 \cdot 10^{-2}}$	$9.5 \cdot 10^{-2}$	0.1
Ground Truth Simulation	–	–	137

Table 4.1: **Interpolation and generalization tasks.** Test root mean squared error (RMSE) for the interpolation and generalization tasks. The CFD-GCN model is compared to the frozen mesh, upsampled coarse mesh (UCM) and the pure GCN model baselines. The CFD-GCN and the GCN achieve similar performance in the interpolation task. The slightly better performance of the GCN is due to overfitting to the training distribution, as demonstrated by the superior performance of the CFD-GCN in the generalization task. **Runtime.** Runtimes for a batch of 16 predictions compared to ground truth simulations with the fine mesh. The CFD-GCN runs significantly faster than running a full simulation, while presenting better results than the GCN. Results are for evaluation mode, without the backwards pass. Tests performed on a 24-core, 2.2 GHz machine with an NVidia GTX 2080 GPU.

pairs of AoA and Mach parameters as in the last experiment. Here, however, the points were split into train and test set such that all points with a Mach number greater than 0.5 were placed in the test set. Shocks become very frequent as the Mach number increases. In order to ensure this qualitative split between training and test sets, the ground truth simulation for each pair of parameters was analyzed individually to guarantee no simulations with shocks put into the training set.

This particular training split generates a very strong test of generalization. Not only does the test set present behavior that is qualitatively different from what is observed in the training set, it also contains a significant quantitative difference, due to the wide range of Mach numbers that are never seen in training. Therefore, this experiment presents a good setting to evaluate the generalization capabilities of the proposed model and the baselines.

Table 4.1 summarizes the results for this experiment, and Figure 4.10 presents the training curves for the CFD-GCN and the baselines. As expected, we can see that the CFD-GCN model generalizes better to the test set containing unseen shock behavior. This is also demonstrated qualitatively in Figure 4.8, which presents a sample prediction from the GCN baseline. This baseline overfits the training set strongly, and is unable to consistently make predictions for simulations with shocks. Conversely, even though it was never trained on this type of flow, our method is able to generate predictions that are closer to the ground truth by using the available coarse simulation. This can also be observed in Figure 4.9. In many test cases containing shock, the CFD-GCN is able to approximate the characteristics even of the unseen behavior. Additionally, the performance of the upsampled coarse mesh baseline demonstrates that once again our method is not relying simply on upsampling the

simulation, but is also learning additional information to improve its predictions. Sample predictions for the upsampled coarse mesh baseline are presented in Figure 4.11.

Finally, we can also observe that the full CFD-GCN model also outperforms the frozen mesh baseline. This result demonstrates the optimizing the coarse mesh by using the gradients computed through the simulations allows the model to optimize the simulation outputs in order to achieve predictions that generalize better. Figure 4.12 demonstrates the transformation of the coarse mesh before and after the training procedure. The optimization performed is significant enough that the changes are easily perceptible visually. The changes are greater around the airfoil, where the gradient of the prediction loss is expected to be higher, demonstrating that the training procedure adjusts the mesh according to the training objective.

4.3.3 Runtime

Table 4.1 demonstrates the efficiency of our method compared to running a full simulation. By downsampling the mesh down almost 20x to 354 nodes, our method is able to make a prediction much faster than running the full ground-truth simulations.

In our experiments, and as can be noticed with the comparison to the upsampled coarse mesh and GCN baselines, we observed that the bulk of the time that the GCN takes to make a prediction is consumed by the CFD simulation. On average, approximately 85% of the time to make a batch of predictions was due to the CFD simulation, 10% to upsampling the mesh and 5% to processing the graph convolutions. Due to the additional complexity of performing the simulations, total training time for the pure GCN baseline was also faster. Whereas training the CFD-GCN took approximately 19 hours, training the GCN baseline took approximately 1.3 hours.

Even though the pure GCN baseline model is able to make predictions faster, it does not generalize as well across diverse physical behaviors, as demonstrated in our experiments. Therefore, we note that the CFD-GCN model, as its name suggests, provides a trade-off between the high cost and ability to generalize of a full CFD simulation, and the low cost and ability to generalize of GCN predictions.

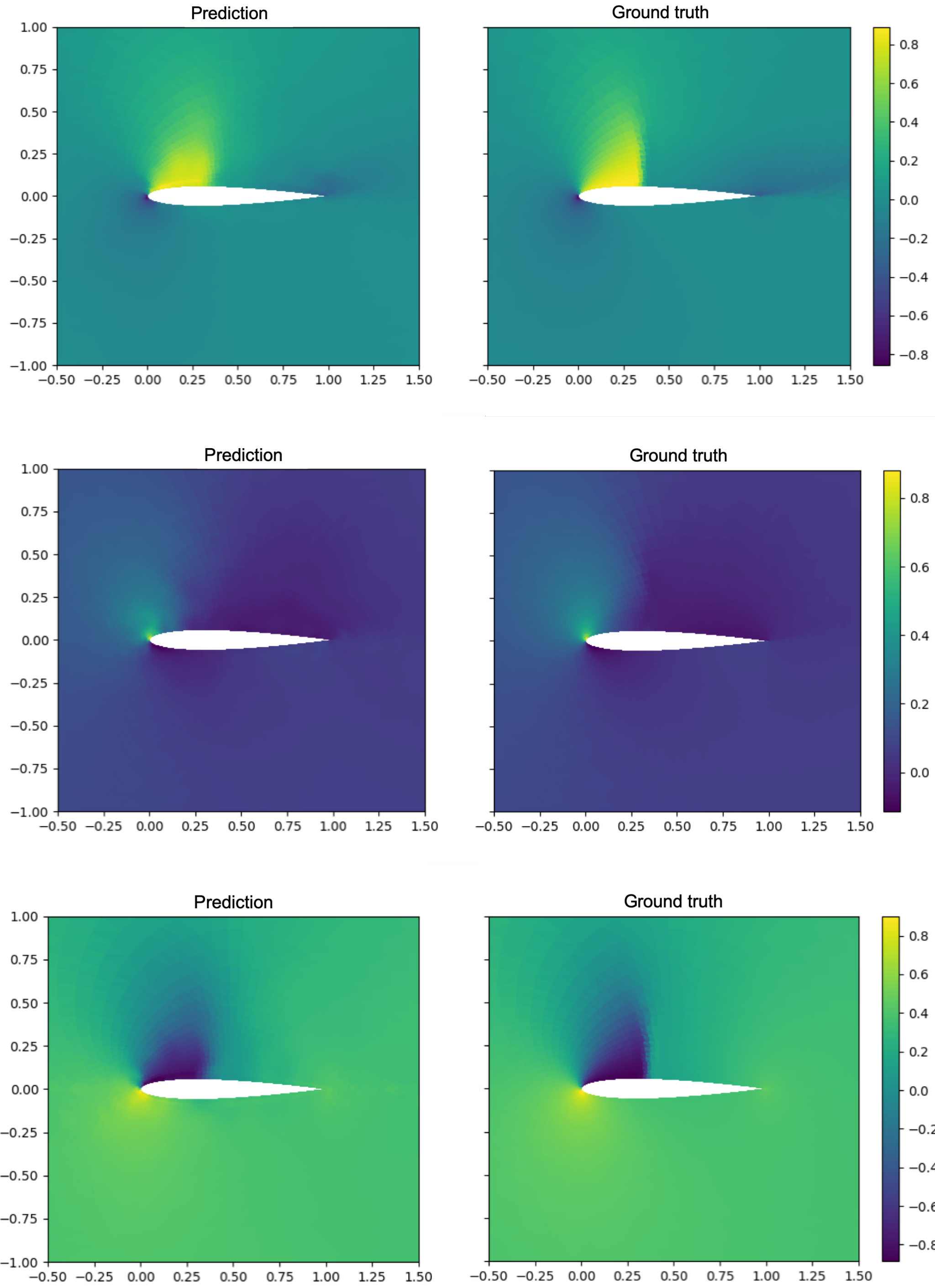


Figure 4.6: CFD-GCN model prediction and ground truth for a test sample in the interpolation task. The x and y components of the velocity and pressure output fields are presented here.

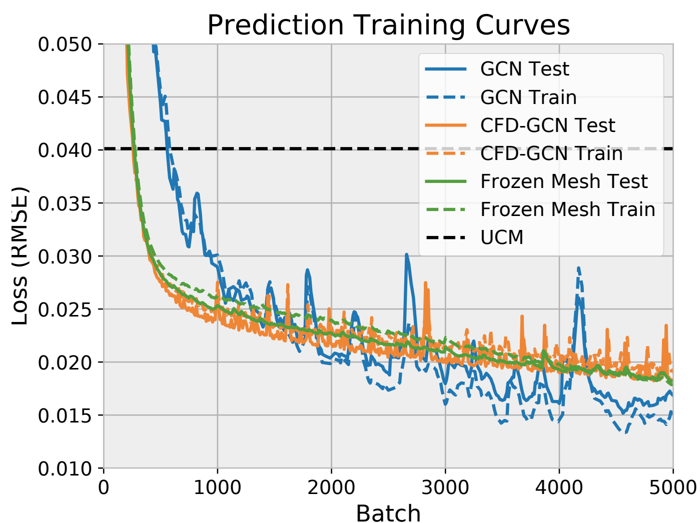


Figure 4.7: Training curves for the interpolation experiment. The vertical axis represented the root mean squared error (RMSE). The GCN baseline overfits more strongly to the training set, but since the test set is drawn from a similar distribution of parameters, this helps it outperform the CFD-GCN model.

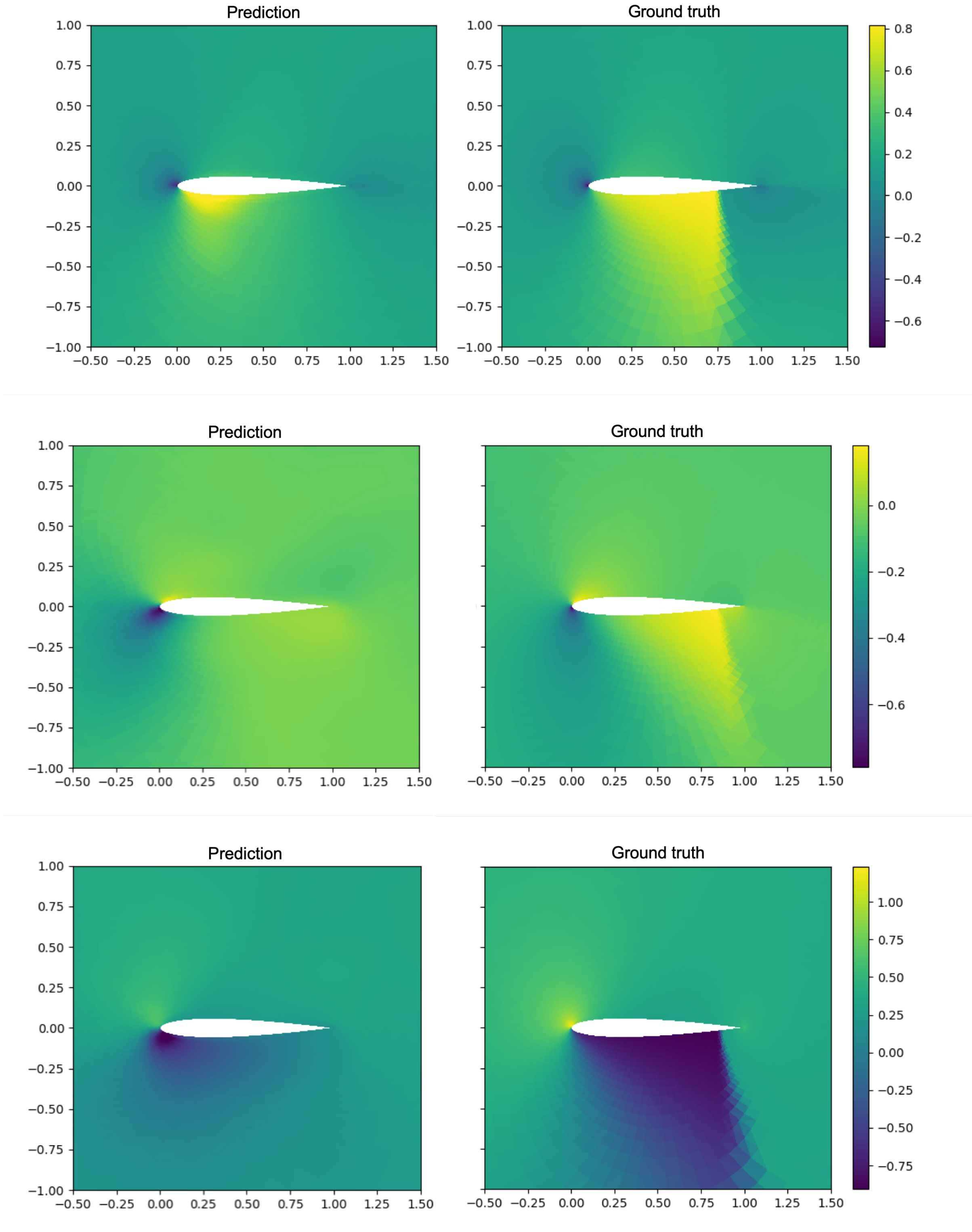


Figure 4.8: The GCN baseline prediction for a test sample with a large shock in the generalization task. In many cases with large shocks the GCN is unable to generalize to this previously unseen behavior. The x and y components of the velocity and pressure output fields are1 presented here.

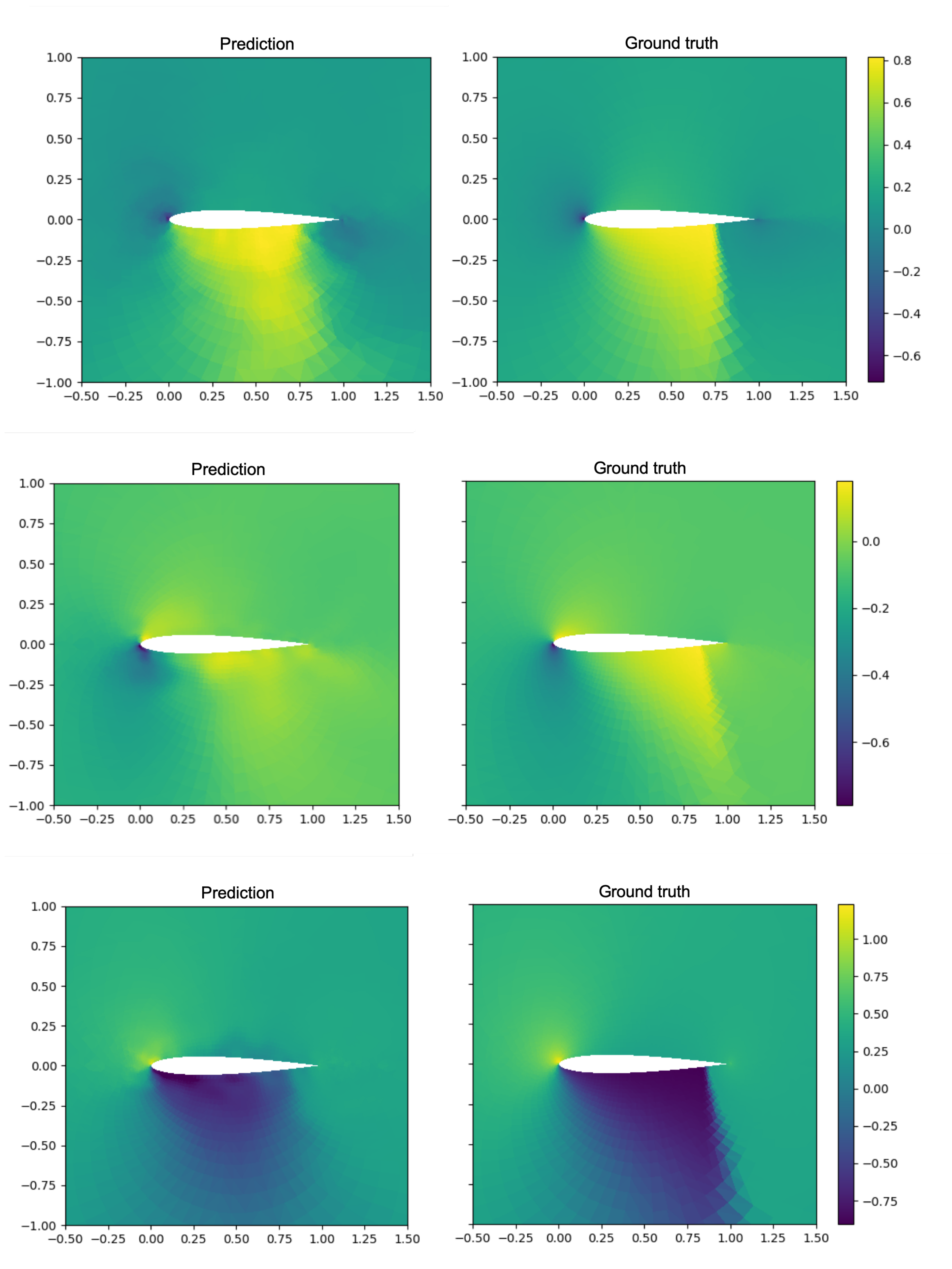


Figure 4.9: The CFD-GCN model prediction for a test sample with a large shock in the generalization task. It can generalize better than the pure GCN model to examples with large shocks, which were not seen in the training set. The x and y components of the velocity and pressure output fields are presented here.

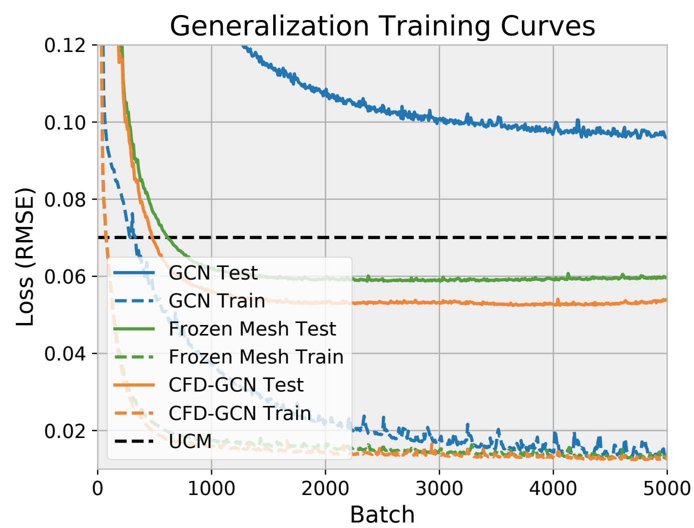


Figure 4.10: Training curves for the generalization experiment. The GCN baseline overfits more strongly to the training set, being unable to generalize well to the test set.

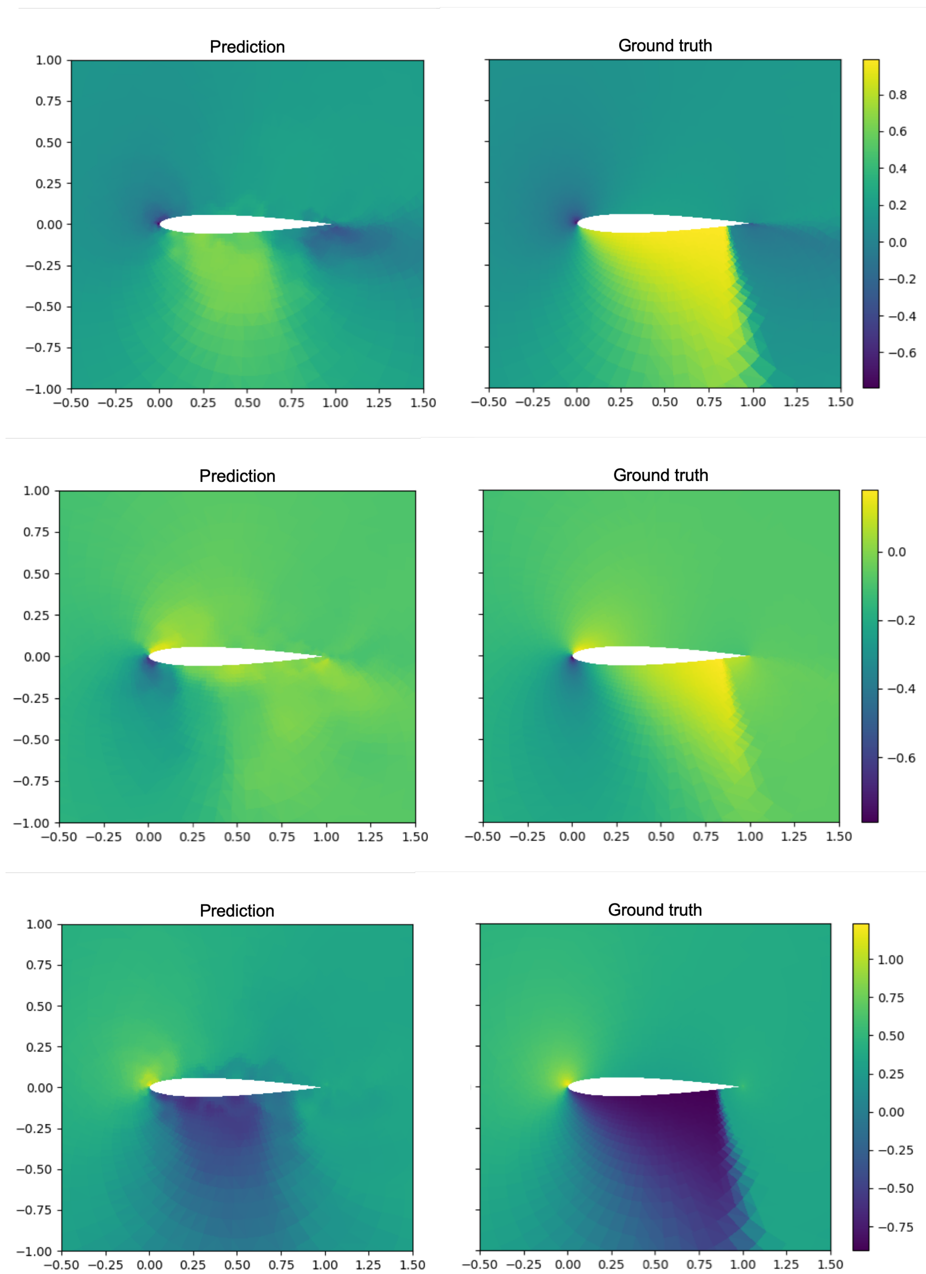
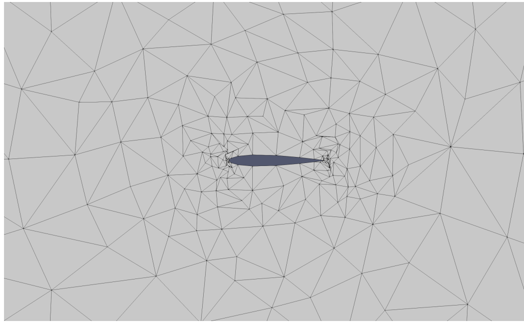
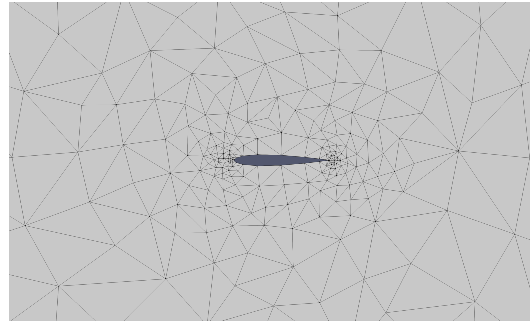


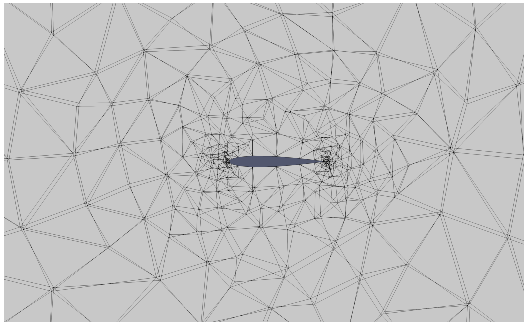
Figure 4.11: The upsampled coarse mesh baseline prediction for a test sample with a large shock in the generalization task. The x and y components of the velocity and the pressure output fields are presented here.



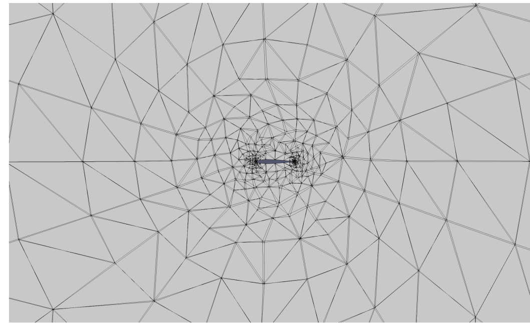
(a) The NACA0012 mesh after training.



(b) The original NACA0012 mesh before training.



(c) The meshes before and after training, superimposed.



(d) Farther view of before and after meshes, superimposed.

Figure 4.12: Comparison of the coarse mesh before and after being optimized during training. Changes are greater around the airfoil, where the gradients of the loss are large. Regions further away from the wing, which do not affect prediction strongly, are mostly unaltered.

Part II

Improving Physics-Informed Learning

Chapter 5

Solving Parameterized Differential Equations with Physics-Informed Hypernetworks

In this chapter, we present the usage of soft constraints, in the form physics-informed losses, combined with a weak inductive bias, in the form of a hierarchical architecture, the hypernetwork. This allows for more efficient learning of the solutions to parameterized differential equations.

5.1 Introduction

The recent successes of deep learning approaches in diverse domains have motivated many works exploring their application to physical systems, including approximating the solutions to differential equations [Berg and Nyström, 2018, Chen et al., 2018, He et al., 2000, Jin et al., 2021, Karniadakis et al., 2021, Lagaris et al., 1998, Lee and Kang, 1990, Long et al., 2018, Mai-Duy and Tran-Cong, 2003, Sirignano and Spiliopoulos, 2018]. Recent advancements in deep learning optimization methods and automatic differentiation tools have led to the development of deep learning-based methods that have been shown to be competitive with traditional solvers in certain conditions, such as in high-dimensional problems [Avrutskiy, 2020] or inverse problems [Raissi et al., 2020].

In many applications such as shape optimization, topology optimization or design prototyping, approximate solutions with fast iteration times might be preferred over ones that are guaranteed to be accurate, yet are more computationally complex. In these cases, machine learning models might offer an interesting alternative to traditional methods. Moreover, the usage of data-driven methods allows for the incorporation of data into the solution, which can be useful in domains where only noisy or partial measurements are available, or where the underlying physics are not fully known [Eivazi et al., 2021, Raissi, 2018, Raissi and Karniadakis, 2018, Tipireddy et al., 2019].

Physics-informed neural networks (PINNs) [Raissi et al., 2017, 2019a] have been recently proposed as a method for employing neural networks as function approximators for the

solution of differential equations, while allowing the incorporation of the underlying physical knowledge in the form of a physics-informed loss. In their standard formulation, PINNs fit the solution to a single parameterization of a differential equation. Therefore, when working in a domain that requires evaluating the solutions at multiple parameterizations, this requires either utilizing the naive approach of re-training the model multiple times to find the solution at each parameterization, or instead including the parameterization explicitly as an input to the neural network model [Arthurs and King, 2021, Gao et al., 2020, Sun et al., 2020]. Given that the training of the model is the most expensive part of the process, having to repeat the learning procedure for every parameterization can greatly increase the computational cost of the method. Conversely, augmenting the model to take into account the parameters requires increasing the capacity of the neural network, as it now has to both approximate the solution function *and* model the parameter space, thus also increasing the computational cost at inference time.

In this chapter, we propose looking at the problem of learning parameterized families of differential equations as a meta-learning problem, where the given parameters and initial/boundary conditions define a task, which can then be solved by a neural network. Under this framework, we propose the HyperPINN, which uses a hypernetwork [Ha et al., 2016] to learn to model the parameter space of a differential equation, taking as input a given parameterization and producing as output a main network that approximates the solution function at that specific parameterization. By separating this task into two parts, the complexity of modeling the parameter space is "offloaded" to the hypernetwork, which is only evaluated once for every parameterization. Importantly, this allows the main network, which is evaluated at every time-space point, to remain small. We demonstrate this approach with experiments on a PDE and an ODE, using both "standard" PINNs [Raissi et al., 2019a] and multistep neural networks [Raissi et al., 2018].

5.2 Preliminaries

5.2.1 Differential equations

Let us assume a differential equation in the general form

$$\begin{aligned} &\mathcal{N}[t, x, u(t,x); \lambda] = 0, \\ &\text{with } t \in [0, T], x \in \Omega, \end{aligned} \tag{5.1}$$

where $\mathcal{N}[\cdot; \lambda]$ is an arbitrary (possibly non-linear) differential operator, which can contain time and space derivatives, and is parameterized by some list of parameters $\lambda \in \mathbb{R}^d$. Here, t is the time variable ranging up to time T, x is the D-dimensional spatial variable in some domain $\Omega \subseteq \mathbb{R}^D$, with boundary $\partial\Omega$, and $u(t,x)$ is the solution function to the differential equation. In order for a solution to be defined, initial and boundary conditions need to be provided. These can assume different forms, but in general initial conditions define $u(0,x)$ for $x \in \Omega$, and boundary conditions define $u(t,x)$ for $x \in \partial\Omega$ and $t \in [0,T]$.

For a concrete example of this formulation, refer to Equation 5.8, in which $\mathcal{N}$ is a non-linear operator containing first and second derivatives that defines the left-hand side of the equation, and $\lambda = \nu$. The initial and boundary conditions are given in Equation 5.9.

5.2.2 Physics-Informed Neural Networks

Physics-informed neural networks [Raissi et al., 2019a] are a method for approximating the solution to differential equations using neural networks (NNs). In this method, a neural network $\hat{u}(t, x; \theta)$, with learned parameters θ, is trained to approximate the actual solution function $u(t, x)$ to a given partial differential equation (PDE).

Importantly, PINNs employ not only a standard "supervised" data loss, but also a physics-informed loss, which consists of the differential equation residual defined by $\mathcal{N}$. Thus, for a given optimization hyper-parameter α, the training loss consists of

$$\begin{aligned} L(\theta) &= L_{\text{data}}(\theta) + \alpha L_{\text{physics}}(\theta), \\ L_{\text{data}}(\theta) &= \sum_{(t_i, x_i, u_i) \in \mathcal{D}} [\hat{u}(t_i, x_i; \theta) - u_i]^2, \\ L_{\text{physics}}(\theta) &= \sum_{(t_c, x_c) \in \mathcal{C}} \mathcal{N}[t_c, x_c, \hat{u}(t_c, x_c; \theta); \lambda]^2, \end{aligned} \tag{5.2}$$

where $\mathcal{D}$ is a dataset containing ground-truth values for u (e.g., from simulation data) at points (t_i, x_i), and $\mathcal{C}$ is a set of collocation points at which to evaluate the differential equation residual (which does not require ground-truth solution data). While the data in $\mathcal{D}$ can be used to enforce the initial and boundary conditions, the physics-informed loss term regularizes the search space, penalizing functions $\hat{u}$ that do conform to the differential equations, reducing the need for simulation data.

5.2.3 Multistep Neural Networks

Multistep neural networks [Raissi et al., 2018] are a method related to PINNs in which a neural network model is used to approximate the time-derivative of a dynamical system (instead of the actual solution function, as in a traditional PINN). Instead of using the differential equation residuals, the loss for the multistep neural network is derived directly from the formulation for traditional multistep methods. That is, for a general dynamical system

$$\frac{d}{dt} x(t) = f(x(t)), \tag{5.3}$$

where f is some arbitrary (linear or non-linear) function of $x \in \mathbb{R}^D$, a two-step linear multistep method, with timestep Δt, gives us the relation

$$x_n = x_{n-1} + \frac{1}{2} \Delta t (f(x_n) + f(x_{n-1})). \tag{5.4}$$

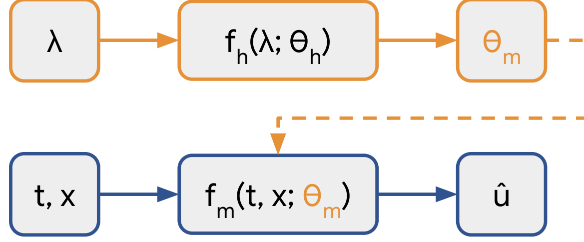


Figure 5.1: Diagram for the HyperPINN. The hypernetwork, represented in orange, takes as input the parameterization and outputs the parameters for the main network. The main network, represented in blue, takes as input a space-time coordinate and outputs its prediction using the parameters provided by the hypernetwork.

If we want to train a neural network $\hat{f}(\cdot;\theta)$ to approximate the time derivative f, this relation can then be used to define a residual loss over a dataset of points x_n sampled from a given trajectory

$$L(\theta) = \sum_{(x_n, x_{n-1}) \in \mathcal{D}} [x_n - x_{n-1} - \frac{1}{2}\Delta t(\hat{f}(x_n;\theta) + \hat{f}(x_{n-1};\theta))]^2. \tag{5.5}$$

5.2.4 Hypernetworks

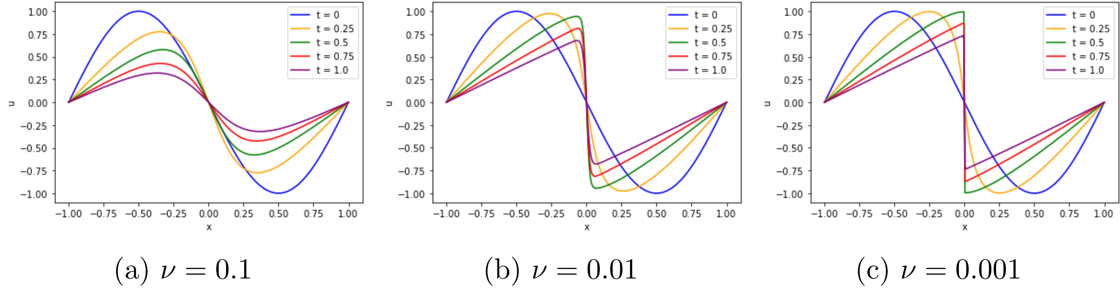
Hypernetworks [Ha et al., 2016] are a recently proposed meta-learning method in which learning is broken down into two separate networks: a main network and a hypernetwork. The main network performs the desired task, in the same way a neural network would normally be employed. The parameters for this network, however, are not learned directly at training time. Instead, the parameters for the main network are generated, at evaluation time, by the hypernetwork. That is, for a hypernetwork f_h that takes an input x_h, and a main network f_m that takes as input x_m, we have

$$\begin{aligned} \theta_m &= f_h(x_h;\theta_h), \\ \hat{y} &= f_m(x_m;\theta_m), \end{aligned} \tag{5.6}$$

where θ_h are the learnable parameters. This ability to generate neural networks allows the hypernet to meta-learn a space of tasks defined by x_h, outputting an appropriate main network for each task.

5.3 HyperPINN

In this chapter, we propose the HyperPINN, which combines the previously described physics-informed architectures with hypernetworks in order to learn parameterized families



(a) $\nu = 0.1$ (b) $\nu = 0.01$ (c) $\nu = 0.001$

Figure 5.2: Solutions from the 1D Burgers' PDE, displaying diverse behavior for different parameter values.

of differential equations. A HyperPINN has a hypernetwork that, given a certain parameterization, generates a main network that approximate the solution to the corresponding differential equation.

Following the general differential equation definition above, we want to have a hypernetwork that maps a parameterization λ into an approximation of the differential equation given by a neural network. Following the hypernetwork formulation in Equation 5.6, a HyperPINN could consist of

$$\begin{aligned} \theta_{\hat{u}} &= f_h(\lambda; \theta_h), \\ \hat{u} &= \hat{u}(t, x; \theta_{\hat{u}}). \end{aligned} \tag{5.7}$$

Here f_h is the hypernetwork with learneable parameters θ_h, and $\hat{u}$ is the approximate solution. These are trained using a loss on the predictions of the main network, which can then be optimized with gradient-based methods using conventional automatic differentiation packages, since all operations are differentiable. If employing a stochastic gradient descent method, batches of randomly sampled parameter-input pairs can be used. The loss can consist of a regular supervised loss, with a ground truth for the main network prediction for each given parameterization and inputs. Moreover, in order to make the training more data-efficient, physics-informed losses can also be used, such as the PINN loss or the multistep loss defined in Equations 5.2 and 5.5 above.

Figure 5.1 contains a schematic representation of the hyper and main networks that compose the HyperPINN. The hypernetwork, in orange, takes as input the parameterization and outputs the parameters for the main network. The main network, in blue, takes as input a space-time coordinate and outputs its prediction using the parameters provided by the hypernetwork.

To give concrete examples, for the case of a Burgers' PDE (described in more detail in Section 5.4.1 below), λ corresponds to the parameter ν in the PDE, and the main network outputs and approximation of the solution u. For the case of the Lorenz ODE (described in more detail in Section 5.4.2 below), λ corresponds to the parameters (σ, β, ρ). When using the multistep neural network approach, the main network takes as input the spatial coordinates (x, y, z) and outputs the time-derivative $(\dot{x}, \dot{y}, \dot{z})$.

5.4 Experiments

We evaluate the application of HyperPINNs to both an example PDE problem, the 1D Burgers' PDE, and an example ODE problem, the Lorenz system, using both a PINN and a multistep neural network.

5.4.1 1D Burgers' equation

The 1D Burgers' PDE, with solution $u(t, x)$, a function of time t and spatial coordinate x, is given by

$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} - \nu\frac{\partial^2 u}{\partial x^2} = 0. \tag{5.8}$$

We use as our time domain the range $[0, 1]$, and as our spatial domain the range $[-1, 1]$. As our initial condition, we utilized a simple sinusoidal function, coupled with a Dirichlet boundary condition given by

$$\begin{aligned} u(0, x) &= -\sin(\pi x), \\ u(t, -1) &= u(t, 1) = 0. \end{aligned} \tag{5.9}$$

This differential equation has a viscosity parameter ν, which can cause the underlying solution $u(t, x)$ to exhibit different behaviors depending on its value. For low values of ν, such as $\nu = 0.001$, the solution develops a characteristic shock. For higher values, such as $\nu = 0.1$, the solution is smooth. Example solutions for different parameter values are shown in Figure 5.2.

In order to test the ability of the PINN and the HyperPINN to learn parameterized systems across their full range, we performed this experiment by having the parameter ν vary in the range $[0.001, 0.1]$.

A dataset of size 100 was generated with 50 points sampled randomly uniformly at different positions from the initial timestep ($t = 0$) and 50 points sampled randomly uniformly at diverse timesteps from the boundary ($x = -1$ or $x = 1$). Since these values do not change with the parameterization, values of the parameter ν are sampled randomly uniformly from the range $[0.001, 0.1]$ at training time to form the training data points. The collocation points are sampled randomly uniformly from the time-space domain, with accompanying values for ν also sampled randomly uniformly.

We evaluate the HyperPINN and compare it to two PINN baselines. For the HyperPINN, the hypernetwork takes as input the parameter ν and outputs the parameters for a main neural network, which takes as input the space-time coordinates and approximates the solution function $u(t, x)$. For the standard PINN baselines, a single network takes as input both the parameter ν and the space time coordinates, and outputs the approximate solution. The summary of the results are shown in Table 5.1.

The HyperPINN consists of a main network with 6 fully-connected (FC) hidden layers of size 8 (393 parameters) and a hypernetwork with 3 FC hidden layers of size 32 followed by 1 hidden layer of size 16 (9385 parameters). The first baseline, the small PINN baseline,

consists of a FC network of approximately the same size as the main network in the HyperPINN, with 6 hidden layers of size 8 (401 parameters). This baseline serves as a comparison point to the main network, which is the one that is evaluated multiple times after the hypernetwork is evaluated once to generate the main network's weights.

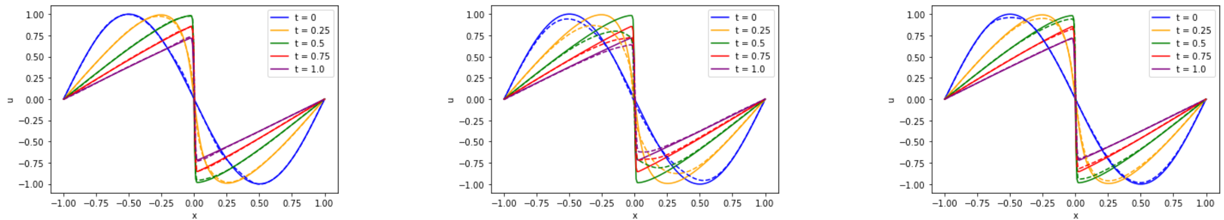
Qualitatively, we can see in Figure 5.3 that the HyperPINN learns to approximate the solution function even at a parameter value containing a shock. The small PINN baseline is not able to learn to properly approximate the solution, lacking capacity to fully learn the parameter space. Full solutions (over time and space) for the HyperPINN and the small PINN baseline are shown in Figure 5.4, where we can also observe that the small PINN solution is lacking.

The second baseline, the large PINN baseline, consists of a FC neural network of approximately the same size as the full hyper and main networks combined in the HyperPINN, with 10 hidden layers of size 32 (9665 parameters).

This large baseline serves as a comparison point with the same full capacity as the HyperPINN. It is able to achieve results close to the HyperPINN, but at the cost of having a larger size. As a consequence, the large PINN has a runtime of 158μs for performing a single prediction, whereas the main network in the HyperPINN has a runtime of 86μs for performing the same prediction, which is faster than even the small PINN. That is, for a similar number of parameters, the HyperPINN is able to surpass the performance of a large network while keeping the main network as efficient as a small one, by offloading the task of learning the parameter space to the hypernetwork. All times were measured on an NVIDIA Tesla V100 GPU.

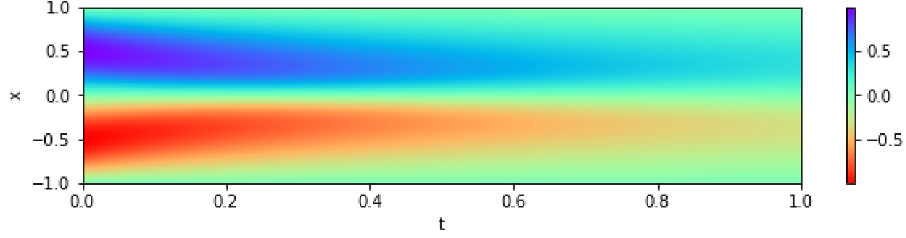
Table 5.1: Comparison of HyperPINN and baselines on the parameterized Burgers' PDE.

Model	**Mean squared error**	**Model size**	**Evaluation time**
Small PINN	$3.0 \cdot 10^{-4}$	401 parameters	92μs
Large PINN	$2.3 \cdot 10^{-5}$	9665 parameters	158μs
HyperPINN	$1.9 \cdot 10^{-5}$	Main: 393 parameters Hyper: 9385 parameters	Main: 86μs Hyper: 158μs

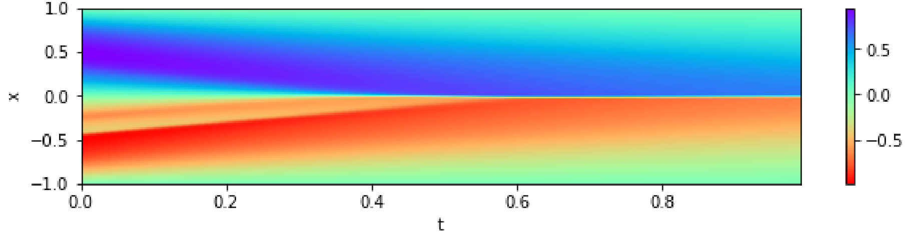


(a) HyperPINN (b) Small PINN baseline (c) Large PINN baseline

Figure 5.3: Results for the parameterized 1D Burgers' PDE at a series of time points for $\nu = 0.003$. Solid lines represent the ground truth simulation, and dashed lines represent the model predictions.



(a) HyperPINN



(b) Small PINN baseline

Figure 5.4: Comparison of the full domain solution for the HyperPINN and the Small PINN baseline for the 1D Burgers experiment.

5.4.2 Lorenz system

In this experiment we attempt to learn the solution to the Lorenz ODE, with solution (x, y, z), defined by

$$\begin{cases} \dfrac{dx}{dt} = \sigma(y - x) \\ \dfrac{dy}{dt} = x(\rho - z) - y \\ \dfrac{dz}{dt} = xy - \beta z.t \end{cases} \tag{5.10}$$

This system of differential equations has parameters (σ, β, ρ).

In our experiments we noticed that a traditional PINN approach had a hard time learning this function. This was less so due to a failure of the physics-informed approach than to a general difficulty of fully-connected neural networks to learn functions with high frequency components such as this one.

We demonstrate this with a simple experiment in which we attempt to directly fit the solution function, without any physics-informed loss term. Impressively, we observed that fully-connected neural networks, even of a fairly large size, had a hard time even fitting the function even under full supervision in the training data.

In Figure 5.5, we show a Lorenz system solution and its breakdown into individual components. We attempted to fit this solution (mapping time to space coordinates) using a fully-connected neural network with 5 hidden-layers of size 128 and tanh activations, trained for $10,000$ epochs on 250 equally spaced points from the ground truth solution. It is clear the network is not able to properly capture the high frequency behavior of the target

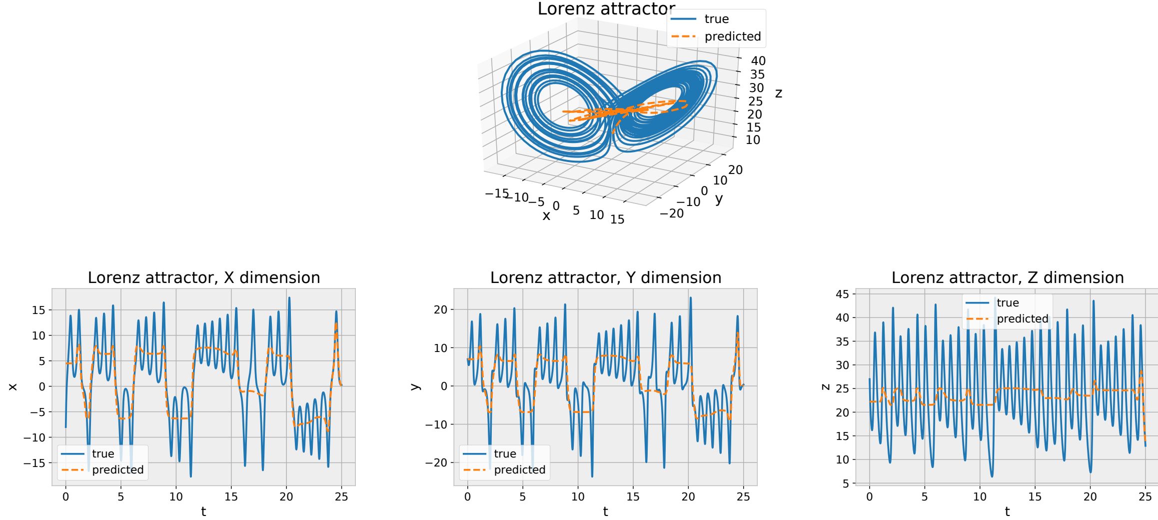


Figure 5.5: Fitting the Lorenz solution with a fully connected network with hyperbolic tangent activations. The network has 5 hidden layers of size 128 and is trained for 10,000 full batch steps on 250 equally spaced points from the solution. The solution to the Lorenz system is ran for $t \in [0, 25]$ with $(\sigma, \beta, \rho) = (10, 8/3, 28)$ and initial condition $(x_0, y_0, z_0) = (-8, 7, 27)$.

function, often averaging out the fast oscillations within each low frequency ("butterfly wing") mode. Similar results are shown for the same setting using ReLU activations in Figure 5.6. If we train for a very long time, with a very large network, we are eventually able to solve fit the solution, but this still very inefficient, and the task setting is the easiest possible. It would be infeasible to use this approach on any appropriate usage setting.

In Chapters 6 and 7, we will explore a solution to this problem, namely employing sinusoidal networks. (In Figure 6.1 we present the results for this same experiment on a smaller neural network with sin activations.) For now, in this chapter, we shift to employing multistep neural networks, a physics-informed approach that has been demonstrated to be able to learn the Lorenz system well [Raissi et al., 2018]. In this setting, instead of fitting the solution function directly like a traditional PINN, the multistep neural network learns the time-derivative of the Lorenz system (given by the right-hand side in Equation 5.10). This also has the added benefit of demonstrating that our proposed method works with diverse physics-informed approaches.

In order to test the ability of the PINN and the HyperPINN to learn parameterized systems, we performed this experiment with $\sigma = 10$, ρ varying in the range $[0, 28]$, β in $[2/3, 8/3]$. We sampled 100 different initial conditions randomly uniformly from $[-10, 10]^3$ for each of 30 different parameter value combinations. These values cause the underlying solution to exhibit diverse behavior, including the chaotic attractor.

Evaluation is performed using parameter and initial condition values not seen in training. We computed trajectories of 25s duration with $\Delta t = 0.01$ using the Runge-Kutta Fehlberg integrator. Evaluation is performed using a separate set of 100 trajectories, sampled from

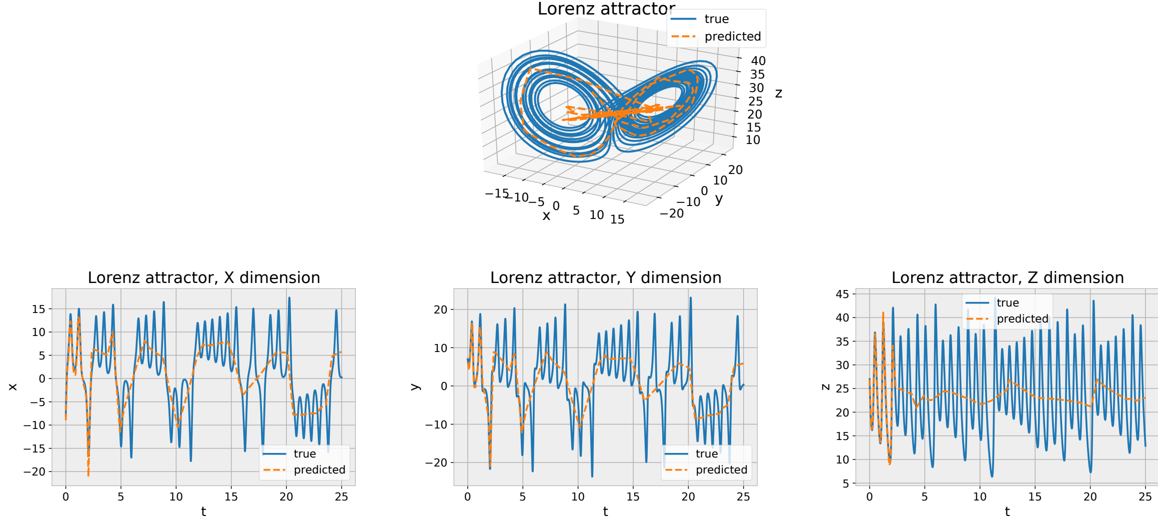


Figure 5.6: Fitting the Lorenz solution with a fully connected network with ReLU activations. The network has 5 hidden layers of size 128 and is trained for 10,000 full batch steps on 250 equally spaced points from the solution. The solution to the Lorenz system is ran for $t \in [0, 25]$ with $(\sigma, \beta, \rho) = (10, 8/3, 28)$ and initial condition $(x_0, y_0, z_0) = (-8, 7, 27)$.

the same range of parameters and initial conditions, but using different values from the training set.

Example trajectories for different initial conditions and parameter values are shown in Figure 5.7.

We compare the HyperPINN with two baselines, a small and a large multistep neural network. The HyperPINN consists of a main FC network with one hidden layer of size 16 (115 parameters) and a FC hypernetwork with 2 hidden layers of size 16 and 8 (1406 parameters). The small baseline consists of a FC neural network with one hidden layer of size 16 (214 parameters). The large baseline consists of a FC neural network with one hidden layer of size 256 (3334 parameters). In the HyperPINN, the hypernetwork takes as input the parameters (σ, β, ρ) and outputs the parameters for a main network, which takes as input the state (x, y, z) and outputs the time-derivative $(\dot{x}, \dot{y}, \dot{z})$. In the baselines, a single network takes as input both the parameters and the state, and outputs the time-derivative.

Note that when performing integration using the learned model, it is expected that the system will deviate from the ground truth. Given the chaotic nature of the system, even minuscule deviations from the true time-derivative compound over time when performing integration, causing large errors even when the correct qualitative behavior is captured. Nevertheless, when compared to the baselines, the HyperPINN achieves lower error in predicting the trajectories of the system. While the HyperPINN achieves an aggregate squared error over all test trajectories of 17.5, the small baseline has an error of 39.4, and the large baseline has an error of 20.6. Importantly, when analyzing the results qualitatively, we can see that the HyperPINN achieves this lower error by more accurately capturing the qualitative behavior of the system at different parameter values. In comparison, the

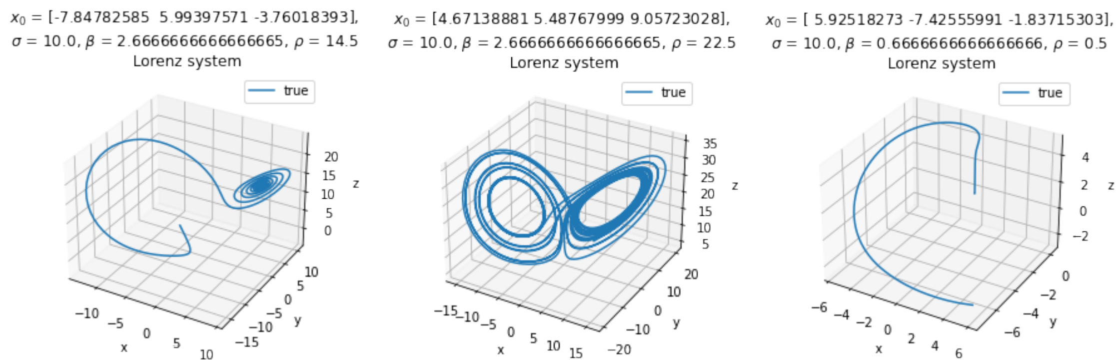
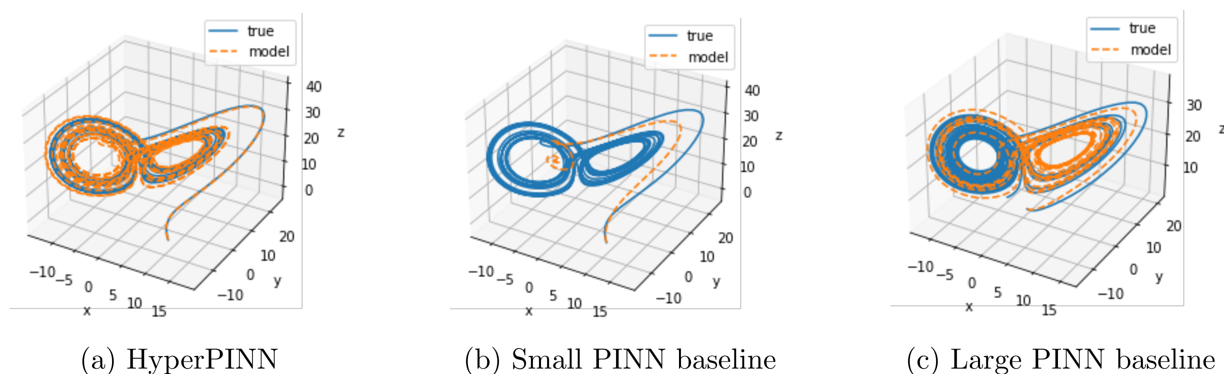


Figure 5.7: Trajectories from the Lorenz system, displaying diverse behavior for different initial conditions and parameter values.



(a) HyperPINN (b) Small PINN baseline (c) Large PINN baseline

Figure 5.8: Results for the Lorenz ODE with $\sigma = 10$, $\beta = 5/3$ and $\rho = 21.7$. Solid lines represent the ground truth trajectory, and dashed lines represent the trajectory integrated from model predictions.

baselines frequently mistake one type of behavior for another, as exemplified in Figure 5.8.

Particularly, when compared to the small PINN baseline, the HyperPINN achieves better performance with a similarly sized main network. In comparison to the large baseline, the HyperPINN is able to approximate the solution at diverse parameters and initial conditions while using a much smaller sized main network, which is repeatedly evaluated for integration at test time.

Chapter 6

Simple Sinusoidal Networks

6.1 Introduction

In the Lorenz experiment in Chapter 5 (Section 5.4.2), we observed that we had to use a multistep neural network [Raissi et al., 2018] (a neural network that fits the *derivatives* in a differential equation), instead of a traditional physics-informed neural network (a neural network that fits the *solution function* directly), due to the fact that regular PINNs (using hyperbolic tangent or ReLU activations) had a very hard time fitting the solution to the Lorenz system. In this chapter, we will use that failure mode as a starting point to motivate the development of an architecture better suited to the types of functions we frequently see in physics problems.

The phenomenon observed in Chapter 5 has been previously described in the literature. It has been observed that neural networks have a spectral bias [Basri et al., 2019, Cao et al., 2020, Rahaman et al., 2019]. They tend to have difficulty learning functions with high-frequency components, as seen in the Lorenz experiment. This has been described both in general and in the particular case of PINNs learning the solution to differential equations [Wang et al., 2020, 2021].

One proposed solution has been to employ Fourier feature mappings [Rahimi and Recht, 2007, Tancik et al., 2020], that is, mapping the input values to a set of sinusoidal bases. This has been shown to be effective in overcoming spectral bias in PINNs [Wang et al., 2021]. Moreover, it is also has the added benefit of rendering the input features shift-invariant, a useful property that has been leveraged, for example, in spatial and language domains [Mildenhall et al., 2020, Tancik et al., 2020].

Neural networks with sine non-linearities, which we call *sinusoidal networks* are mathematically analogous to regular networks with Fourier features (at least with respect to the first layer), and thus inherit many of their desirable properties, such as allowing for more efficient learning of higher frequency functions and shift-invariance.

As a motivating example, in Figure 6.1, we show the same Lorenz task mentioned above from Chapter 5, but employing a sinusoidal network. Not only does this simple change allow the network to easily fit the target function quickly, but also to do so much more efficiently – the neural network used to generate the output in the figure has only 3 hidden

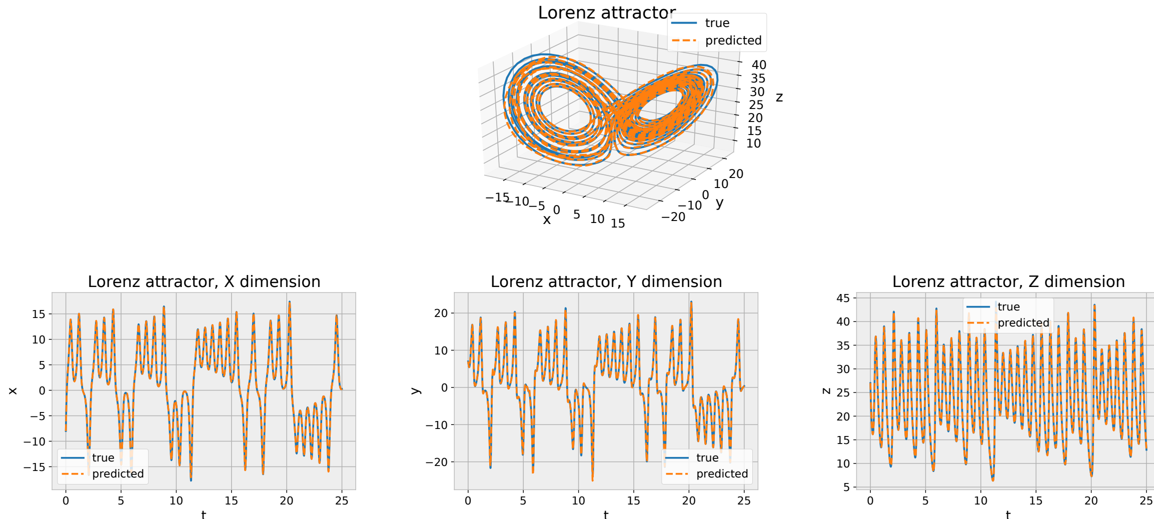


Figure 6.1: Fitting the Lorenz solution with a fully connected network with sine activations. The network has 3 hidden layers of size 64 and is trained for 10,000 full batch steps on 250 equally spaced points from the solution. The solution to the Lorenz system is ran for $t \in [0, 25]$ with $(\sigma, \beta, \rho) = (10, 8/3, 28)$ and initial condition $(x_0, y_0, z_0) = (-8, 7, 27)$.

layers of size 64.

Recently, sinusoidal representation networks (SIRENs) have been proposed [Sitzmann et al., 2020], employing sinusoidal activations together with particular design choices to improve learning. [Sitzmann et al., 2020] also argue that sinusoidal networks have the property of being closed under derivation, that the derivative of its outputs with respect to its inputs is given by another sinusoidal network, due to the nature of the derivatives of trigonometric functions. Intuitively, this property can be particularly useful for PINNs, in which training is often supervised based on PDE-based losses containing higher order derivatives. Indeed, some works have employed sinusoidal networks in the context of physics-informed learning, showing promising results in a variety of domains [Huang et al., 2021a,b, Raissi et al., 2019b, Song et al., 2021, Wong et al., 2022].

In this chapter and Chapter 7, we will propose a much simplified sinusoidal network architecture, and then analyze its behavior under the neural tangent kernel framework. This analysis will allow us to understand the functioning of these networks, enabling us to better tune their characteristics to the spectrum of each given problem. We then finally demonstrate that these "well-tuned" sinusoidal networks outperform their traditional counterparts in spatial and spatio-temporal tasks, including the learning of solutions to differential equations.

6.2 Simple Sinusoidal Networks

As we will see in the following section, there are many details that complicate the practical implementation of current sinusoidal networks. After describing these details, we will

propose a simplified version of sinusoidal networks. We then demonstrate that this simple sinusoidal network can nonetheless maintain or surpass the performance level of standard SIRENs.

Throughout the following chapters, we will use the term *sinusoidal network* to refer generally to any network with sinusoidal activations (including SIRENs). We will reserve the term *SIREN* particularly for the architecture proposed by Sitzmann et al. [2020]. That is, a sinusoidal network with the particular design choices we will describe below. Finally, we will call our simplified version of SIREN, described in section 6.2.2 below, the *simple* sinusoidal network (SSN).

6.2.1 Practical Implementation Details of SIRENs

On the surface, SIRENs appear to be simple. For a given layer l in an MLP, we can express its output as

$$f_l(x) = \phi(W_l x + b_l), \tag{6.1}$$

for some set of layer parameters (W_l, b_l) and some activation function ϕ. In a "traditional" MLP, ϕ would usually be a non-linear function such as the commonly employed hyperbolic tangent (*tanh*) or rectified linear unit (ReLU). One might think that "converting" such an MLP into a SIREN might amount to instead defining $\phi := \sin$. However, practical implementation of SIRENs, as originally suggested by Sitzmann et al. [2020], involves an additional number of details.

The ω parameter

The first such detail is a frequency hyperparameter, $\omega \in \mathbb{R}$, which inserts itself into the first layer of a sinusoidal MLP (notice $l = 1$) as

$$f_1(x) = \sin(\omega\,(W_1 x + b_1)). \tag{6.2}$$

This hyperparameter is chosen before training and kept fixed throughout. No method for choosing ω is provided, except for the fact that a value of $\omega = 30$ works well for most tasks because it allows the inputs to the sine function to span multiple periods over the range $[-1, 1]$. We perform a theoretical analysis of the function of ω in a sinusoidal network and how this can inform its choice later in this chapter.

When implementing SIRENs in practice, instead of applying ω only to the first layer, as specified in Equation 6.2, Sitzmann et al. [2020] in fact apply the scaling to all layers, but negate its effect by dividing the initial value of W_l for later layers ($l > 1$) by a factor of ω. That is, *at initialization*, we would have

$$W_l = \frac{W_l^{init}}{\omega}, \quad l > 1, \tag{6.3}$$

where W_l^{init} is the value of the usual initialization of W_l (discussed below). Obviously, the values of W_l can drift after training progresses, but at least intuitively it is clear that the effect of this adjustment is to cancel out the influence of ω on later layers, while keeping it as a scaling factor for the first one.

Initialization scheme

Besides this scaling factor, SIRENs also have a particular initialization scheme for its parameter matrices W_l. Each entry $W_l^{(i)}$ is sampled from a uniform distribution in the range $[-\sqrt{6/n}, \sqrt{6/n}]$, where n is the number of inputs to the layer l. This applies to all layers but the first. Sitzmann et al. [2020] prove that this initialization is such that the input to each sine activation will be normal distributed with a standard deviation of 1, and it is chosen for this reason.

The first layer is initialized separately by sampling $W_1^{(i)} \sim \mathcal{U}(-1/n, 1/n)$. This difference in scaling is mostly inconsequential, as the output of the first layer is scaled by the parameter ω described in the previous section. Nevertheless, it is worth mentioning as it both adds extra complexity to the activation scheme, and also changes our interpretation of the absolute values of ω (in comparisons to other approaches we will analyze and to NTK assumptions).

6.2.2 Simplifying SIRENs

After discussing the implementation details behind SIRENs, we will now propose a series of simplifications with the goal of formulating a sinusoidal network that (almost) amounts to simply substituting its activation functions by the sine function. We will however, keep a simplified implementation of the ω parameter, since (as we will in future analyses) it is in fact a useful tool for allowing the network to fit inputs of diverse frequencies.

The ω parameter

The first step is thus to remove all scaling parameters ω except for the first layer, and consequently also remove the "cancelling initialization" described in Equation 6.3. As a consequence, the layer activation equations of our simple sinusoidal network are simply

$$f_1(x) = \sin(\omega\,(W_1 x + b_1)), \tag{6.4}$$

$$f_l(x) = \sin(W_l x + b_l), \quad l > 1. \tag{6.5}$$

Though Sitzmann et al. [2020] claim that the scaling by ω of all layers (with further cancellation in layers after the first) helps accelerate training, in our experiments (in Section 6.3 below) we were able to at least match the performance of regular SIRENs even without this scaling factor – with the added benefit of simplifying the overall implementation and initialization of the sinusoidal layers. Though we have not fully tested this hypothesis, we believe it is likely that modern adaptive optimizers, such as Adam [Kingma and Ba, 2014b], are able to compensate for the different gradient magnitudes at individual layers, preserving the ability to learn efficiently even in the presence of different scaling factors.

Additionally, we can leverage this simplification to further enhance the capabilities of our sinusoidal networks. As we will show in Section 7.5.2, the fact that we are now only multiplying the network input layer by ω, allows us to know ω will always multiply a vector of a given dimension (which is not usually the case in SIRENs, since commonly input and hidden sizes are distinct in MLPs). Given this fact, we can in fact have ω be

multi-dimensional, instead of a scalar, which, as we will see, allows us to further tune our network to fit dimensions with different frequency spectra (as is commonly the case in problems with space and time dimensions, for example). We leave the discussion of this "enhancement" for after our further analysis of the sinusoidal networks in Chapter 7, which will inform our understanding of the role of ω, and for now we focus only on the simplifications.

Initialization scheme

Our final simplification to SIRENs is to reformulate their parameter initialization scheme. Instead of utilizing uniform initializations, with different bounds for the first and subsequent layers, we propose initializing all parameters using a standard Kaiming (*i.e.*, He) normal initialization scheme. This amounts to sampling all weights $W_l^{(i)}$ from a normal distribution with mean 0 and standard deviation $\sqrt{2/n}$. This choice not only greatly simplifies the initialization scheme of the network, but it also facilitates theoretical analysis of the behavior of the network under the NTK framework, as we will see later in Chapter 7.

Proof of the initialization scheme

We will now show that this particular choice of initialization distribution preserves the variance of the original proposed SIREN initialization distribution. As a consequence, the theoretical justification for the original initialization scheme still holds. Moreover, we also demonstrate empirically that the desired properties of the initialization (namely, having the input to each sine activation will be normal distributed with a standard deviation of 1) are maintained in practice

Lemma 6.1. *Given any c, for $X \sim \mathcal{N}\left(0, \frac{1}{3}c^2\right)$ and $Y \sim \mathcal{U}\left(-c, c\right)$, we have* $\mathrm{Var}[X] = \mathrm{Var}[Y] = \frac{1}{3}c^2$.

Proof. By definition, $\mathrm{Var}[X] = \sigma^2 = \frac{1}{3}c^2$. For Y, we know that the variance of a uniformly distributed random variable with bound $[a, b]$ is given by $\frac{1}{12}(b-a)^2$. Thus, $\mathrm{Var}[Y] = \frac{1}{12}(2c)^2 = \frac{1}{3}c^2$. □

This simple Lemma and its corollary relates to Lemma 1.7 in Sitzmann et al. [2020], showing that the initialization we propose here has the same variance as the one proposed for SIRENs. Using this result we can translate the result from the main Theorem 1.8 from Sitzmann et al. [2020], which shows that the SIREN initialization indeed has the desired properties, to our proposed initialization. The Theorem below show the same properties still hold. The reasoning for assuming the inputs are uniform in the range $[-1, 1]$ is that sinusoidal networks are usually applied to problems mapping from spatial (or time-space) coordinates (commonly normalized to $[-1, 1]$) to some output.

Theorem 6.2. *For a uniform input in $[-1, 1]$, the activations throughout a sinusoidal networks are approximately standard normal distributed before each sine non-linearity and*

arcsine-distributed after each sine non-linearity, irrespective of the depth of the network, if the weights are distributed normally, with mean 0 *and variance* $\frac{2}{n}$ *with* n *is the layer's fan-in.*

Proof. The proof follows exactly the proof for Theorem 1.8 in Sitzmann et al. [2020], only using Lemma 6.1 when necessary to show that the initialization proposed here has the same variance necessary for the proof to follow. □

Empirical evaluation of initialization scheme

To empirically demonstrate the proposed simple initialization scheme preserves the properties from the SIREN initialization scheme, we perform the same analysis performed by Sitzmann et al. [2020] (Figure 2 in their Appendix).

For this analysis we use a sinusoidal MLP with 6 hidden layers of 2048 units, and single-dimensional input and output. This MLP is initialized using the simplified scheme described above. For testing, 2^8 equally space inputs from the range $[-1, 1]$ are passed through the network. We then plot the histogram of activations after each linear operation (before the sine non-linearity) and after each sine non-linearity. To match the original plot, we also plot the 1D Fast Fourier Transform of all activations in a layer, and the gradient of this output with respect to each activation. These results are presented in Figure 6.2.

The main conclusion from this figure is that the distribution of activations matches the predicted normal (before the non-linearity) and arcsine (after the non-linearity) distributions, and that this behavior is stable across many layers. We also reproduced the same result up to 50 layers.

We then perform an additional experiment in which the exact same setup as above is employed, yet the 1D inputs are shifted by a large value (*i.e.*, $x \to x + 1000$). We the show the same plot as before in Figure 6.3. We can see that there is essentially no change from the previous plot, which demonstrates the sinusoidal networks shift-invariance in the input space, one of its important desirable properties, as discussed previously. This will also be further demonstrated and analyzed in Chapter 7, when we study the NTK of sinusoidal networks.

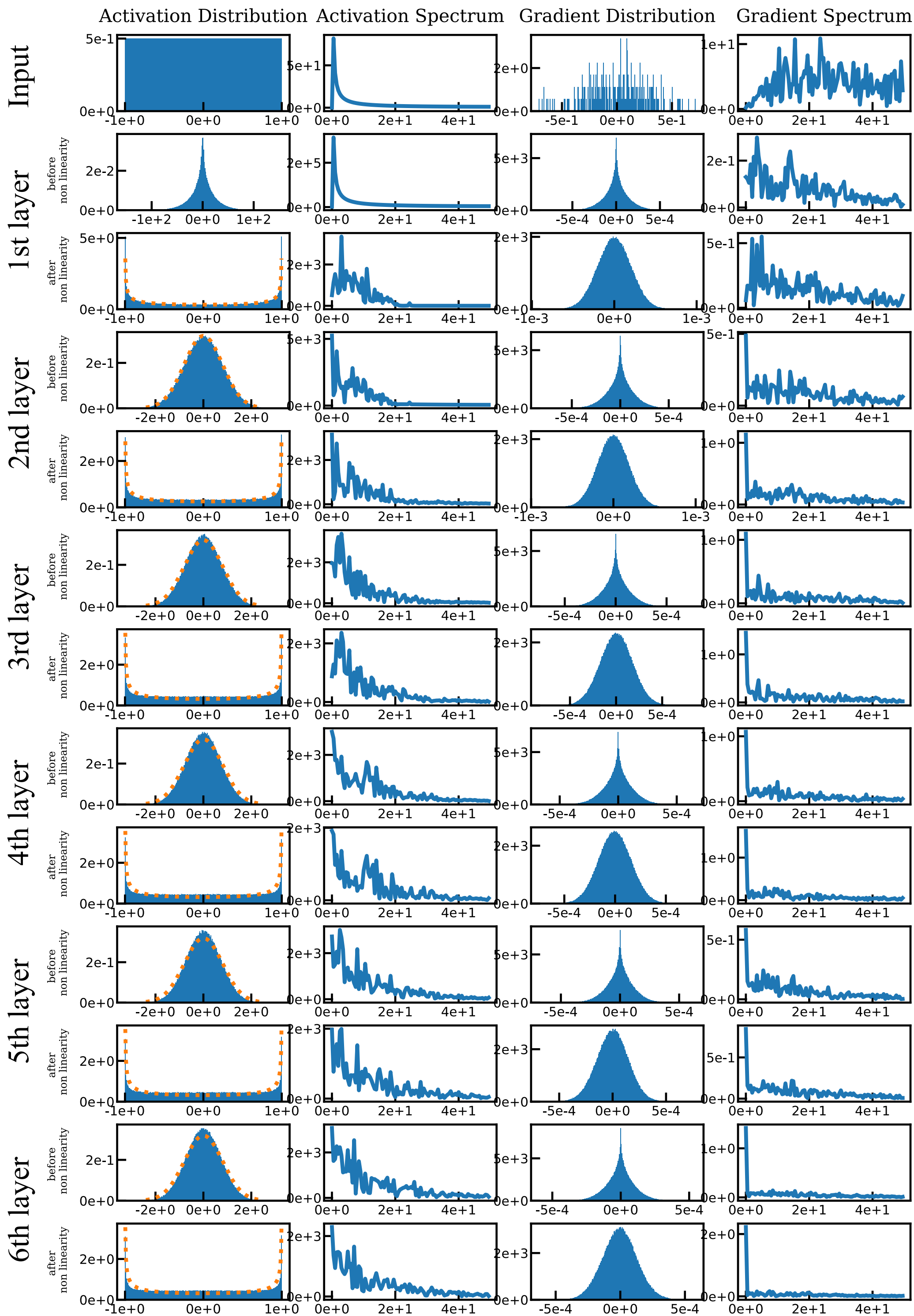


Figure 6.2: Activation statistics for 6 layers of a simplified sinusoidal network, demonstrating the observed distributions matched the theoretical expectation and preserves the properties from the SIREN initialization.

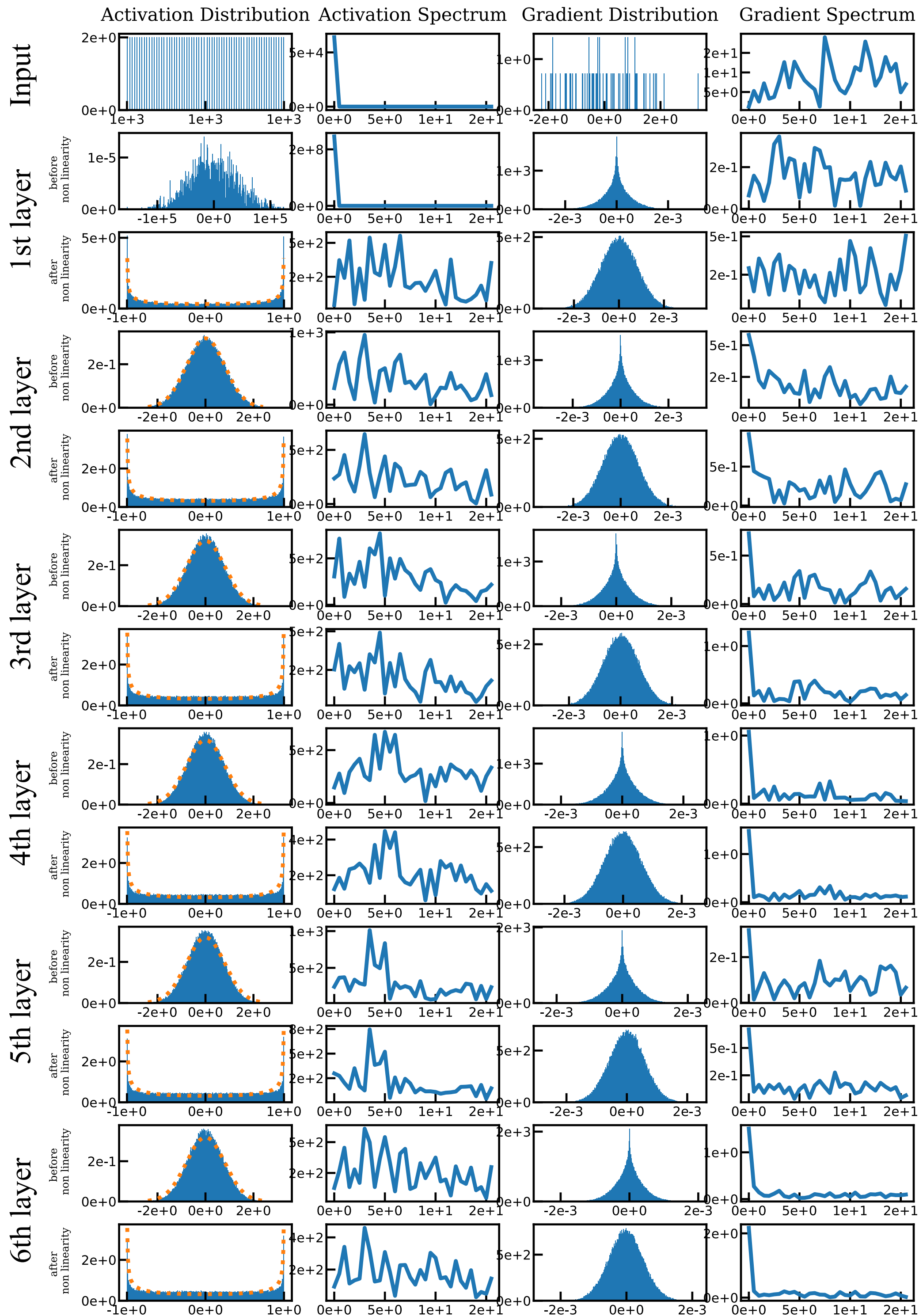


Figure 6.3: Activations for 6 layers of a simplified sinusoidal network in which the input has been shifted by a large value, *i.e.*, $x \to x + 1000$. The distribution characteristics are preserved, demonstrating the sinusoidal network's shift-invariance.

Table 6.1: Comparison of the simple sinusoidal network and SIREN results, both directly from Sitzmann et al. [2020] (when reported) and from our own reproduced experiments. Values above the horizontal center line are peak signal to noise ratio (PSNR), values below are mean squared error (MSE), except for SDF which uses a composite loss. †Audio experiments utilized a different learning rate for the first layer, see the full description below for details.

Experiment	Simple Sinusoidal Network	SIREN [paper]	SIREN [ours]
Image	**50.04**	49 (approx.)	49.0
Poisson (Gradient)	**39.66**	32.91	38.92
Poisson (Laplacian)	**20.97**	14.95	20.85
Video (cat)	**34.03**	29.90	32.09
Video (bikes)	**37.4**	32.88	33.75
Audio (Bach)†	$1.57 \cdot 10^{-5}$	$\mathbf{1.10 \cdot 10^{-5}}$	$3.28 \cdot 10^{-5}$
Audio (counting)†	$\mathbf{3.17 \cdot 10^{-4}}$	$3.82 \cdot 10^{-4}$	$4.38 \cdot 10^{-4}$
Helmholtz equation	$\mathbf{5.94 \cdot 10^{-2}}$	–	$5.97 \cdot 10^{-2}$
SDF (room)	**12.99**	–	14.32
SDF (statue)	6.22	–	**5.98**

6.3 Experiments

In order to demonstrate our simplified sinusoidal network does not suffer in performance compared to a standard SIREN, in this section we reproduce all the main results from Sitzmann et al. [2020]. In fact, we observe that in almost all cases our simplified sinusoidal network surpasses the performance of SIREN.

Table 6.1 compiles the results for all experiments. In order to be fair, we compare the simplified sinusoidal network proposed in this chapter with both the results directly reported in Sitzmann et al. [2020], and our own reproduction of the SIREN results (using the same parameters and settings as the original). We can see from the numbers reported in the table that the performance of the simple sinusoidal network proposed in this chapter matches or even surpasses the performance of the SIRENs in all cases.

It is important to note that this is not a favorable setting for the simplified sinusoidal network, given that the training durations were very short. The standard SIREN favors quickly converging to a solution, though it does not have as strong asymptotic behavior. This effect is likely due to the multiplicative factor applied to later layers described in Section 6.2.1.

As hypothesized in Section 6.2.2, we observe that indeed in almost all cases we can compensate this effect by simply lowering the learning rate with the Adam optimizer. The only exception was in the audio experiments, in which a very large omega is used (in the $15,000 - 30,000$ range, compared to the $10 - 30$ range for all other experiments). In these cases, we were still able to compensate by setting a separate, lower learning rate for the first layer compared to the rest of the network. We can see how this can be critiqued as

Table 6.2: Comparison of the simple sinusoidal network and SIREN on some experiments, with a longer training duration. The specific durations are described below in the details for each experiment. We can see that the simple sinusoidal network has stronger asymptotic performance. Values above the horizontal center line are peak signal to noise ratio (PSNR), values below are mean squared error (MSE). †Audio experiments utilized a different learning rate for the first layer, see the full description below for details.

Experiment	Simple Sinusoidal Network	SIREN [ours]
Image	**56.21**	52.95
Poisson (Gradient)	**39.51**	38.70
Poisson (Laplacian)	**22.09**	20.82
Video (cat)	**34.66**	32.19
Video (bikes)	**38.1**	34.07
Audio (Bach)†	$\mathbf{4.83 \cdot 10^{-7}}$	$3.02 \cdot 10^{-6}$
Audio (counting)†	$\mathbf{3.86 \cdot 10^{-5}}$	$1.46 \cdot 10^{-4}$

re-introducing complexity, counteracting the purpose the proposed simplification. However, we argue (1) that this is only limited to cases with extremely high omega, which is not present in any case except for fitting audio waves, and (2) that adjusting the learning rate for an individual layer is still an approach that is simpler and more in line with standard machine learning practice compared to multiplying all layers by a scaling factor and then dividing their initializations by the same amount, except for in the first layer.

Finally, we observe that besides being able to exceed the performance of SIREN in a shorter training regimen, the simple sinusoidal network performs even more strongly with longer training. To demonstrate this, we repeated some experiments from above, but with longer training durations. These results are shown in Table 6.2.

Below, we present qualitative results and describe experimental details for each experiment. As these are a reproduction of the experiments in Sitzmann et al. [2020], we refer to their details if more information is needed.

6.3.1 Image

In the image fitting experiment, we treat an image as a function from the spatial domain to color values $(x, y) \to (r, g, b)$. In the case of a monochromatic image, used here, this function maps instead to one-dimensional intensity values. We try to learn a function $f : \mathbb{R}^2 \to \mathbb{R}$, parameterized as a sinusoidal network, in order to fit such an image.

Figure 6.4 shows the image used in this experiment, and the reconstruction from the fitted sinusoidal network. The gradient and Laplacian for the learned function are also presented, demonstrating that higher order derivatives are also learned appropriately.

Training parameters. The input image used is 512×512, mapped to an input domain $[-1, 1]^2$. The sinusoidal network used is a 5-layer MLP with hidden size 256, following the

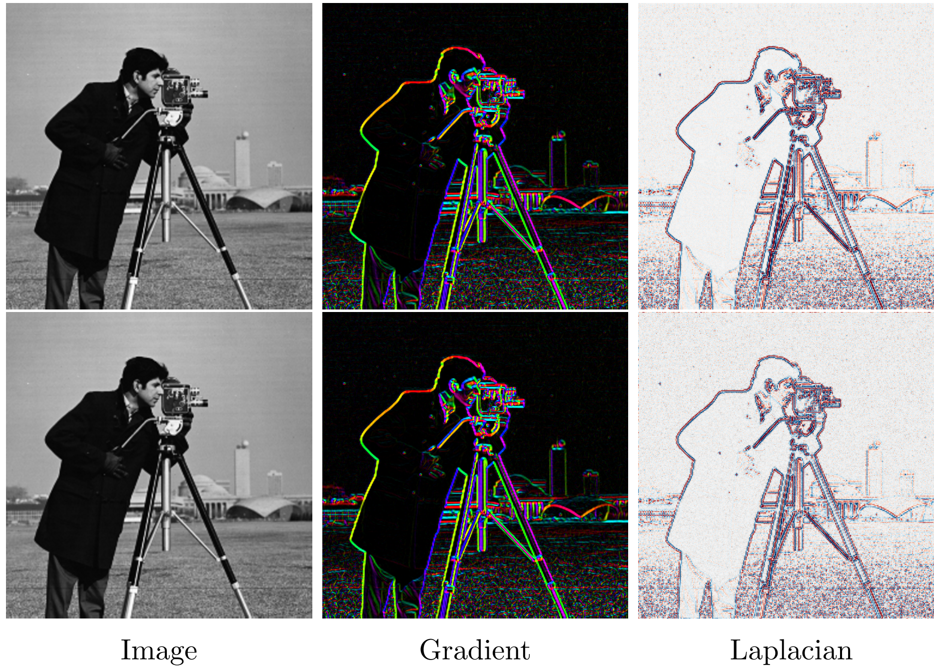


Figure 6.4: Top row: Ground truth image. Bottom: Reconstructed with sinusoidal network.

proposed initialization scheme above. The parameter ω is set to 32. The Adam optimizer is used with a learning rate of $3 \cdot 10^{-3}$, trained for $10,000$ steps in the short duration training results and for $20,000$ steps in the long duration training results.

6.3.2 Poisson

These tasks are similar to the image fitting experiment, but instead of supervising directly on the ground truth image, the learned fitted sinusoidal network is supervised on its derivatives, constituting a Poisson problem. We perform the experiment by supervising both on the input image's gradient and Laplacian, and report the reconstruction of the image and it's gradients in each case.

Figure 6.5 shows the image used in this experiment, and the reconstruction from the fitted sinusoidal networks. Since reconstruction from derivatives can only be correct up to a scaling factor, we scale the reconstructions for visualization. As in the original SIREN results, we can observe that the reconstruction from the gradient is of higher quality than the one from the Laplacian.

Training parameters. The input image used is of size 256×256, mapped from an input domain $[-1, 1]^2$. The sinusoidal network used is a 5-layer MLP with hidden size 256, following the proposed initialization scheme above. For both experiments, the parameter ω is set to 32 and the Adam optimizer is used. For the gradient experiments, in short and long training results, a learning rate of $1 \cdot 10^{-4}$ is used, trained for $10,000$ and $20,000$ steps respectively. For the Laplace experiments, in short and long training results, a learning

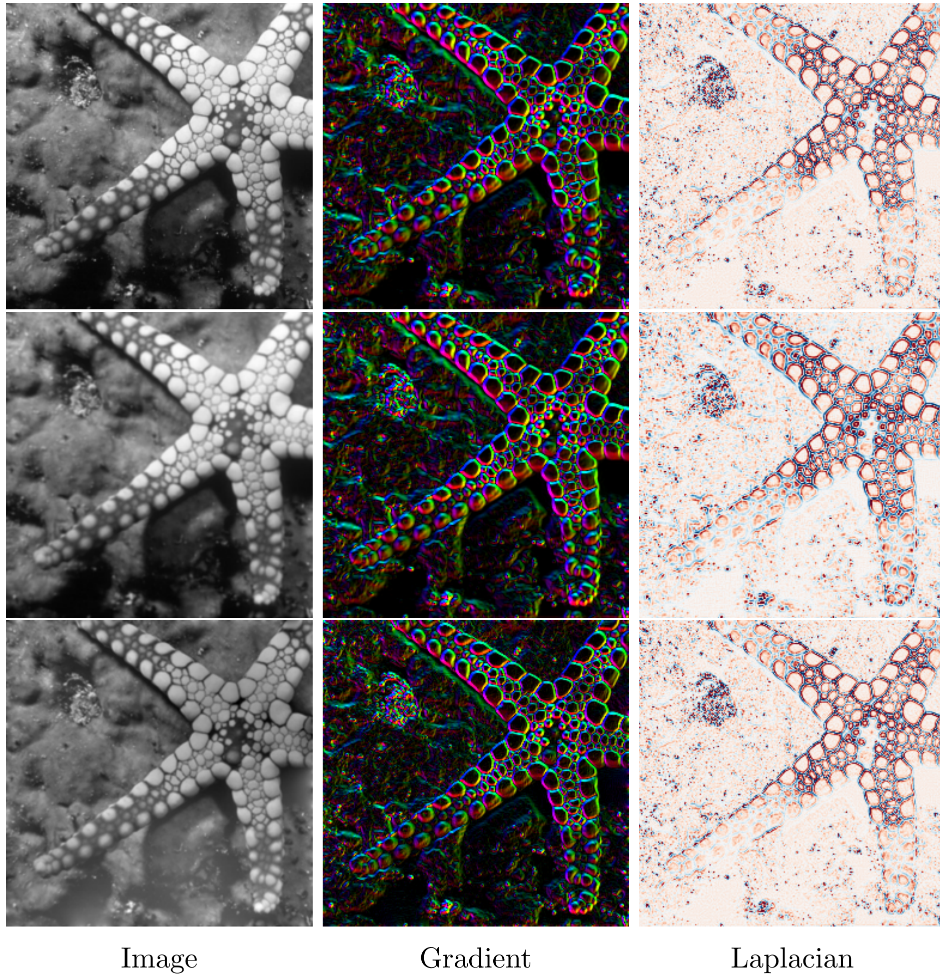


Figure 6.5: Top row: Ground truth image. Mid: Image reconstructed from sinusoidal network supervised on gradient. Bottom: Image reconstructed from sinusoidal network supervised on Laplacian.

rate of $1 \cdot 10^{-3}$ is used, trained for $10,000$ and $20,000$ steps respectively.

6.3.3 Video

These tasks are similar to the image fitting experiment, but we instead fit a video, which also has a temporal input dimension, $(t, x, y) \to (r, g, b)$. We learn a function $f : \mathbb{R}^3 \to \mathbb{R}^3$, parameterized as a sinusoidal network, in order to fit such a video.

Figures 6.6 and 6.7 show sampled frames from the videos used in this experiment, and their respective reconstructions from the fitted sinusoidal networks.

Training parameters. The cat video contains 300 frames of size 512×512. The bikes video contains 250 frames of size 272×640. These signals are fitted from the input domain $[-1, 1]^3$. The sinusoidal network used is a 5-layer MLP with hidden size 1024, following the

Figure 6.6: Top row: Frames from ground truth "cat" video. Bottom: Video reconstructed from sinusoidal network.

Figure 6.7: Top row: Frames from ground truth "bikes" video. Bottom: Video reconstructed from sinusoidal network.

proposed initialization scheme above. The parameter ω is set to 8. The Adam optimizer is used, with a learning rate of $3 \cdot 10^{-4}$ trained for $100,000$ steps in the short duration training results and for $200,000$ steps in the long duration training results.

6.3.4 Audio

In the audio experiments, we fit an audio signal in the temporal domain as a waveform $t \to w$. We to learn a function $f : \mathbb{R} \to \mathbb{R}$, parameterized as a sinusoidal network, in order to fit the audio.

Figure 6.8 shows the waveforms for the input audios and the reconstructed audios from the fitted sinusoidal network.

Training parameters. Both audios use a sampling rate of 44100Hz. The Bach audio is 7s long and the counting audio is approximately 12s long. These signals are fitted from the input domain $[-1, 1]$. The sinusoidal network used is a 5-layer MLP with hidden size 256, following the proposed initialization scheme above. For short and long training results, training is performed for $5,000$ and $50,000$ steps respectively. For the Bach experiment,

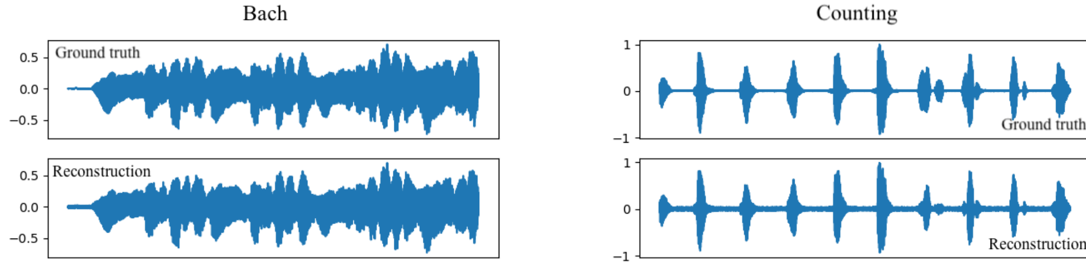


Figure 6.8: Ground truth and reconstructed waveforms for "Bach" and "counting" audios.

the parameter ω is set to $15,000$. The Adam optimizer is used, with a general learning rate of $3 \cdot 10^{-3}$. A separate learning rate of $1 \cdot 10^{-6}$ is used for the first layer to stabilize training due to the large ω value. For the counting experiment, the parameter ω is set to $32,000$. The Adam optimizer is used, with a general learning rate of $1 \cdot 10^{-3}$ and a first layer learning rate of $1 \cdot 10^{-6}$.

6.3.5 Helmholtz equation

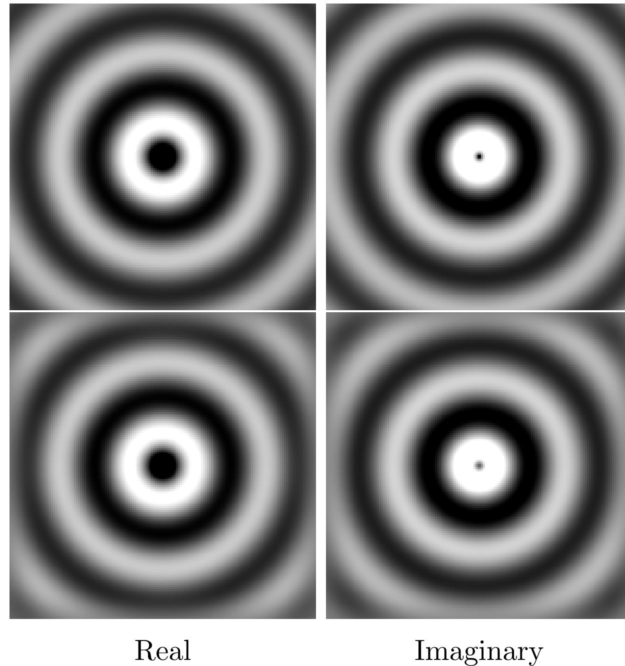


Figure 6.9: Top row: Ground truth real and imaginary fields. Bottom: Reconstructed with sinusoidal network.

In this experiment we solve for the unknown wavefield $\Phi : \mathbb{R}^2 \to \mathbb{R}^2$ in the Helmholtz equation

$$(\Delta + k^2)\Phi(x) = -f(x), \tag{6.6}$$

with known wavenumber k and source function f (a Gaussian with $\mu = 0$ and $\sigma^2 = 10^{-4}$). We solve this differential equation using a sinusoidal network supervised with the physics-informed loss $\int_\Omega \|(\Delta + k^2)\Phi(x) + f(x)\|_1 dx$, evaluated at random points sampled uniformly in the domain $\Omega = [-1, 1]^2$.

Figure 6.9 shows the real and imaginary components of the ground truth solution to the differential equation and the solution recovered by the fitted sinusoidal network.

Training parameters. The sinusoidal network used is a 5-layer MLP with hidden size 256, following the proposed initialization scheme above. The parameter ω is set to 16. The Adam optimizer is used, with a learning rate of $3 \cdot 10^{-4}$ trained for $50,000$ steps.

6.3.6 Signed distance function (SDF)

In these tasks we learn a 3D signed distance function. We learn a function $f : \mathbb{R}^3 \to \mathbb{R}$, parameterized as a sinusoidal network, to model a signed distance function representing a 3D scene. This function is supervised indirectly from point cloud data of the scene. Figures 6.11 and 6.10 show 3D renderings of the volumes inferred from the learned SDFs.

Training parameters. The statue point cloud contains $4,999,996$ points. The room point cloud contains $10,250,688$ points. These signals are fitted from the input domain $[-1, 1]^3$. The sinusoidal network used is a 5-layer MLP with hidden size 256 for the statue and 1024 for the room. The parameter ω is set to 4. The Adam optimizer is used, with a learning rate of $8 \cdot 10^{-4}$ and a batch size of 1400. All models are trained for $190,000$ steps for the statue experiment and for $410,000$ steps for the room experiment.

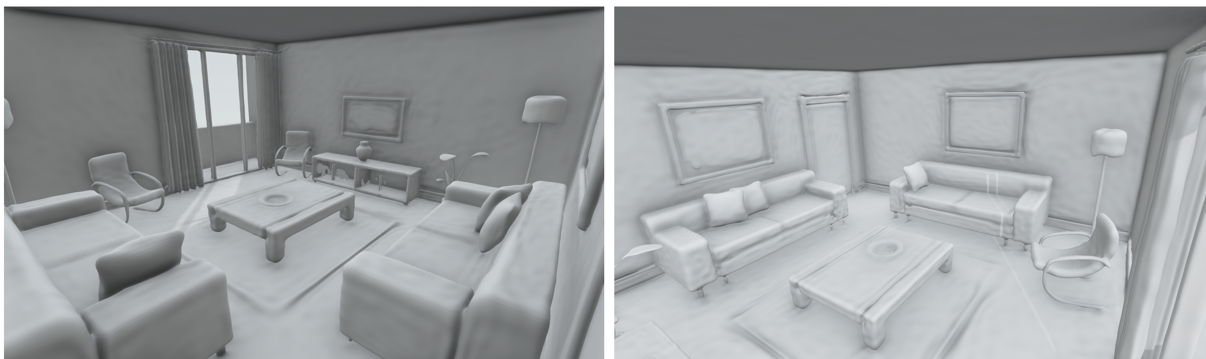


Figure 6.10: Rendering of the “room” 3D scene SDF learned by the sinusoidal network from a point cloud.

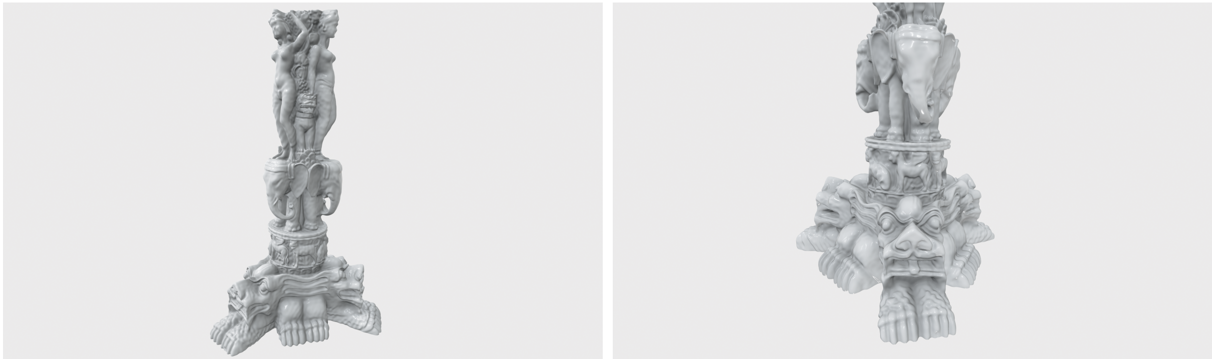


Figure 6.11: Rendering of the “statue” 3D scene SDF learned by the sinusoidal network from a point cloud.

Chapter 7

Understanding and Applying Sinusoidal Networks

In this chapter, we analyze the behavior of sinusoidal networks, including SIREN and our proposed simple sinusoidal network from Section 6.2, from a neural tangent kernel (NTK) perspective. Background on the NTK was previously discussed in Chapter 2.

In the following sections, we will derive the NTK for sinusoidal networks. This analysis provides many insights into their behavior. First, it supports their shift-invariance, observed empirically previously. Second, it shows that sinusoidal networks operate like low-pass filters, with their bandwidth directly defined by the ω parameter. We support these findings with empirical analysis as well. Finally, we demonstrate how these insights can be leveraged to properly "tune" sinusoidal networks to the frequency spectrum of the desired signal.

7.1 Preliminaries

In order to perform the subsequent NTK analysis, we first need to formalize definitions for simple sinusoidal networks and SIRENs. The definitions used here adhere to the common NTK analysis practices, and thus differ slightly from practical implementation.

Definition 7.1. *For the purposes of the following proofs, a (sinusoidal) fully-connected neural network with L hidden layers that takes as input $x \in \mathbb{R}^{n_0}$, is defined as the function $f^{(L)} : \mathbb{R}^{n_0} \to \mathbb{R}^{n_{L+1}}$, recursively given by*

$$f^{(0)}(x) = \omega \left(W^{(0)} x + b^{(0)} \right),$$
$$f^{(L)}(x) = W^{(L)} \frac{1}{\sqrt{n_L}} \sin\!\left(f^{(L-1)} \right) + b^{(L)},$$

where $\omega \in \mathbb{R}$. The parameters $\left\{ W^{(j)} \right\}_{j=0}^{L}$ have shape $n_{j+1} \times n_j$ and all have each element sampled independently either from $\mathcal{N}(0,1)$ (for simple sinusoidal networks) or from $\mathcal{U}(-c, c)$ with some bound $c \in \mathbb{R}$ (for SIRENs). The $\left\{ b^{(j)} \right\}_{j=0}^{L}$ are n_{j+1}-dimensional vectors sampled independently from $\mathcal{N}(0, I_{n_{j+1}})$.

With this definition, we now state the general formulation of the NTK, which applies in general to fully-connected networks with Lipschitz non-linearities, and consequently in particular to the sinusoidal networks studied here as well.

Theorem 7.1. *For a neural network with L hidden layers $f^{(L)} : \mathbb{R}^{n_0} \to \mathbb{R}^{n_{L+1}}$ following Definition 7.1, as the size of the hidden layers $n_1, \ldots, n_L \to \infty$ sequentially, the neural tangent kernel (NTK) of $f^{(L)}$ converges in probability to the deterministic kernel $\Theta^{(L)}$ defined recursively as*

$$\begin{aligned}\Theta^{(0)}(x, \tilde{x}) &= \Sigma^{(0)}(x, \tilde{x}) = \omega^2 \left(x^T \tilde{x} + 1\right), \\ \Theta^{(L)}(x, \tilde{x}) &= \Theta^{(L-1)}(x, \tilde{x}) \dot{\Sigma}^{(L)}(x, \tilde{x}) + \Sigma^{(L)}(x, \tilde{x}),\end{aligned}$$

where $\left\{\Sigma^{(l)}\right\}_{l=0}^{L}$ are the neural network Gaussian processes (NNGPs) corresponding to each $f^{(l)}$ and

$$\dot{\Sigma}^{(l)}(x, \tilde{x}) = \mathbb{E}_{(u,v)\sim\Sigma^{(l-1)}(x,\tilde{x})} \left[\cos(u)\cos(v)\right].$$

Proof. This is a standard general NTK theorem, showing that the limiting kernel recursively in terms of the network's NNGPs and the previous layer's NTK. For brevity we omit the proof here and refer the reader to, for example, Jacot et al. [2020].

The only difference is for the base case $\Sigma^{(0)}$, due to the fact that we have an additional ω parameter in the first layer. It is simple to see that the neural network with 0 hidden layers, *i.e.* the linear model $\omega \left(W^{(0)} x + b^{(0)}\right)$ will lead to the same Gaussian process covariance kernel as the original proof, $x^T \tilde{x} + 1$, only adjusted by the additional variance factor ω^2. □

This Theorem demonstrates that the NTK can be constructed as a recursive function of the NTK of previous layers and the network's NNGPs. In the following sections we will derive the NNGPs for the SIREN and the simple sinusoidal network directly. We will then use these NNGPs with Theorem 7.1 to derive their NTKs as well.

To finalize this preliminary section, we also provide two propositions that will be useful in following proofs in this section.

Proposition 7.2. *For any $\omega \in \mathbb{R}$, $x \in \mathbb{R}^d$,*

$$\mathbb{E}_{w\sim\mathcal{N}(0,I_d)} \left[e^{i\omega\left(w^T x\right)}\right] = e^{-\frac{\omega^2}{2}\|x\|_2^2}$$

Proof. Omitting $w \sim \mathcal{N}(0, I_d)$ from the expectation for brevity, we have

$$\mathbb{E}\left[e^{i\omega\left(w^T x\right)}\right] = \mathbb{E}\left[e^{i\omega\sum_{j=1}^d w_j x_j}\right].$$

By independence of the components of w and the definition of expectation,

$$\mathbb{E}\left[e^{i\omega\sum_{j=1}^d i w_j x_j}\right] = \prod_{j=1}^d \mathbb{E}\left[e^{i\omega\, w_j x_j}\right] = \prod_{j=1}^d \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} e^{i\omega\, w_j x_j} e^{-\frac{w_j^2}{2}} dw_j.$$

Completing the square, we get

$$\prod_{j=1}^{d} \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} e^{i\omega\, w_j x_j} e^{-\frac{1}{2}w_j^2} dw_j = \prod_{j=1}^{d} \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} e^{\frac{1}{2}\left(i^2\omega^2 x_j^2 - i^2\omega^2 x_j^2 + 2ix_j w_j - w_j^2\right)} dw_j$$

$$= \prod_{j=1}^{d} e^{\frac{1}{2}i^2\omega^2 x_j^2} \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} e^{-\frac{1}{2}\left(i^2\omega^2 x_j^2 - 2i\omega^2 x_j w_j + w_j^2\right)} dw_j$$

$$= \prod_{j=1}^{d} e^{-\frac{1}{2}\omega^2 x_j^2} \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} e^{-\frac{1}{2}(w_j - i\omega x_j)^2} dw_j.$$

Since the integral and its preceding factor constitute a Gaussian pdf, they integrate to 1, leaving the final result

$$\prod_{j=1}^{d} e^{-\frac{\omega^2}{2}x_j^2} = e^{-\frac{\omega^2}{2}\sum_{j=1}^{d} x_j^2} = e^{-\frac{\omega^2}{2}\|x_j\|_2^2}.$$

□

Proposition 7.3. *For any $c, \omega \in \mathbb{R}$, $x \in \mathbb{R}^d$,*

$$\mathbb{E}_{w\sim\mathcal{U}_d(-c,c)}\left[e^{i\omega\left(w^T x\right)}\right] = \prod_{j=1}^{d} \operatorname{sinc}(c\,\omega x_j).$$

Proof. Omitting $w \sim \mathcal{U}_d(-c, c)$ from the expectation for brevity, we have

$$\mathbb{E}\left[e^{i\omega\left(w^T x\right)}\right] = \mathbb{E}\left[e^{i\omega\sum_{j=1}^{d} w_j x_j}\right].$$

By independence of the components of w and the definition of expectation,

$$\mathbb{E}\left[e^{i\omega\sum_{j=1}^{d} w_j x_j}\right] = \prod_{j=1}^{d} \mathbb{E}\left[e^{i\omega\, w_j x_j}\right] = \prod_{j=1}^{d} \int_{-c}^{c} e^{i\omega\, w_j x_j} \frac{1}{2c} dw_j = \prod_{j=1}^{d} \frac{1}{2c} \int_{-c}^{c} e^{i\omega\, w_j x_j} dw_j.$$

Now, focusing on the integral above, we have

$$\int_{-c}^{c} e^{i\omega\, w_j x_j} dw_j = \int_{-c}^{c} \cos(\omega\, w_j x_j) dw_j + i \int_{-c}^{c} \sin(\omega\, w_j x_j) dw_j$$

$$= \left.\frac{\sin(\omega\, w_j x_j)}{\omega x_j}\right|_{-c}^{c} - i \left.\frac{\cos(\omega\, w_j x_j)}{\omega x_j}\right|_{-c}^{c}$$

$$= \frac{2\sin(c\,\omega x_j)}{\omega x_j}.$$

Finally, plugging this back into the product above, we get

$$\prod_{j=1}^{d} \frac{1}{2c} \int_{-c}^{c} e^{i\omega\, w_j x_j} dw_j = \prod_{j=1}^{d} \frac{1}{2c} \frac{2\sin(c\,\omega x_j)}{\omega x_j} = \prod_{j=1}^{d} \operatorname{sinc}(c\,\omega x_j).$$

□

7.2 Shallow sinusoidal networks

For the next few proofs, we will be focusing on neural networks with a single hidden layer, *i.e.* $L = 1$. Expanding the definition above, such a network is given by

$$f^{(1)}(x) = W^{(1)} \frac{1}{\sqrt{n_1}} \sin\bigl(\omega \, \bigl(W^{(0)}x + b^{(0)}\bigr)\bigr) + b^{(1)}. \tag{7.1}$$

The advantage of analysing such shallow networks is that their NNGPs and NTKs have formulations that are intuitively interpretable, providing insight into their characteristics. We later extend these derivations to networks of arbitrary depth.

7.2.1 SIREN

First, let us derive the NNGP for a SIREN with a single hidden layer.

Theorem 7.4. ***Shalow SIREN NNGP.*** *For a single hidden layer SIREN $f^{(1)} : \mathbb{R}^{n_0} \to \mathbb{R}^{n_2}$ following Definition 7.1, as the size of the hidden layer $n_1 \to \infty$, $f^{(1)}$ tends (by law of large numbers) to the neural network Gaussian Process (NNGP) with covariance*

$$\Sigma^{(1)}(x, \tilde{x}) = \frac{c^2}{6} \left[\prod_{j=1}^{n_0} \mathrm{sinc}(c\,\omega\,(x_j - \tilde{x}_j)) - e^{-2\omega^2} \prod_{j=1}^{n_0} \mathrm{sinc}(c\,\omega\,(x_j + \tilde{x}_j)) \right] + 1.$$

Proof. We first show that despite the usage of a uniform distribution for the weights, this initialization scheme still leads to an NNGP. In this initial part, we follow an approach similar to Lee et al. [2018a], with the modifications necessary for this conclusion to hold.

From our neural network definition, each element $f^{(1)}(x)_j$ in the output vector is a weighted combination of elements in $W^{(1)}$ and $b^{(1)}$. Conditioning on the outputs from the first layer ($L = 0$), since the sine function is bounded and each of the parameters is uniformly distributed with finite variance and zero mean, the $f^{(1)}(x)_j$ become normally distributed with mean zero as $n_1 \to \infty$ by the (Lyapunov) central limit theorem (CLT). Since any subset of elements in $f^{(1)}(x)$ is jointly Gaussian, we have that this outer layer is described by a Gaussian process.

Now that we have concluded that this initialization scheme still entails an NNGP, we have that its covariance is determined by $\sigma_W^2 \Sigma^{(1)} + \sigma_b^2 = \frac{c^2}{3} \Sigma^{(1)} + 1$, where

$$\begin{aligned}
\Sigma^{(1)}(x, \tilde{x}) &= \lim_{n_1 \to \infty} \left[\frac{1}{n_1} \left\langle \sin\bigl(f^{(0)}(x)\bigr), \sin\bigl(f^{(0)}(\tilde{x})\bigr) \right\rangle \right] \\
&= \lim_{n_1 \to \infty} \left[\frac{1}{n_1} \sum_{j=1}^{n_1} \sin\bigl(f^{(0)}(x)\bigr)_j \sin\bigl(f^{(0)}(\tilde{x})\bigr)_j \right] \\
&= \lim_{n_1 \to \infty} \left[\frac{1}{n_1} \sum_{j=1}^{n_1} \sin\Bigl(\omega \, \Bigl(W_j^{(0)} x + b_j^{(0)}\Bigr)\Bigr) \sin\Bigl(\omega \, \Bigl(W_j^{(0)} \tilde{x} + b_j^{(0)}\Bigr)\Bigr) \right].
\end{aligned}$$

Now by the law of large number (LLN) the limit above converges to

$$\mathbb{E}_{w\sim\mathcal{U}_{n_0}(-c,c),\, b\sim\mathcal{N}(0,1)}\left[\sin\big(\omega\left(w^T x + b\right)\big)\sin\big(\omega\left(w^T \tilde{x} + b\right)\big)\right],$$

where $w \in \mathbb{R}^{n_0}$ and $b \in \mathbb{R}$. Omitting the distributions from the expectation for brevity and expanding the exponential definition of sine, we have

$$\begin{aligned}
&\mathbb{E}\left[\frac{1}{2i}\left(e^{i\omega\left(w^T x+b\right)} - e^{-i\omega\left(w^T x+b\right)}\right)\frac{1}{2i}\left(e^{i\omega\left(w^T \tilde{x}+b\right)} - e^{-i\omega\left(w^T \tilde{x}+b\right)}\right)\right] \\
&= -\frac{1}{4}\mathbb{E}\left[e^{i\omega\left(w^T x+b\right)+i\omega\left(w^T \tilde{x}+b\right)} - e^{i\omega\left(w^T x+b\right)-i\omega\left(w^T \tilde{x}+b\right)} - e^{-i\omega\left(w^T x+b\right)+i\omega\left(w^T \tilde{x}+b\right)} + e^{-i\omega\left(w^T x+b\right)-i\omega\left(w^T \tilde{x}+b\right)}\right] \\
&= -\frac{1}{4}\left[\mathbb{E}\left[e^{i\omega\left(w^T (x+\tilde{x})\right)}\right]\mathbb{E}\left[e^{2i\omega b}\right] - \mathbb{E}\left[e^{i\omega\left(w^T (x-\tilde{x})\right)}\right] - \mathbb{E}\left[e^{i\omega\left(w^T (\tilde{x}-x)\right)}\right] + \mathbb{E}\left[e^{i\omega\left(w^T (-x-\tilde{x})\right)}\right]\mathbb{E}\left[e^{-2i\omega b}\right]\right]
\end{aligned}$$

Applying Propositions 7.2 and 7.3 to each expectation above and noting that the sinc function is even, we are left with

$$\begin{aligned}
&-\frac{1}{4}\left[2\prod_{j=1}^{n_0}\operatorname{sinc}(c\,\omega\,(x_j+\tilde{x}_j)) - 2e^{-2\omega^2}\prod_{j=1}^{n_0}\operatorname{sinc}(c\,\omega\,(x_j-\tilde{x}_j))\right] \\
&\qquad = \frac{1}{2}\left[\prod_{j=1}^{n_0}\operatorname{sinc}(c\,\omega\,(x_j-\tilde{x}_j)) - e^{-2\omega^2}\prod_{j=1}^{n_0}\operatorname{sinc}(c\,\omega\,(x_j+\tilde{x}_j))\right].
\end{aligned}$$

□

For simplicity, if we take the case of a one-dimensional output (*e.g.*, an audio signal or a monochromatic image) with the standard SIREN setting of $c = \sqrt{6}$, the NNGP reduces to

$$\Sigma^{(1)}(x,\tilde{x}) = \operatorname{sinc}\Big(\sqrt{6}\,\omega\,(x-\tilde{x})\Big) - e^{-2\omega^2}\operatorname{sinc}\Big(\sqrt{6}\,\omega\,(x+\tilde{x})\Big) + 1.$$

We can already notice that this kernel is composed of sinc functions. The sinc function is the ideal low-pass filter. For any value of $\omega > 1$, we can see the the first term in the expression above will completely dominate the expression, due to the exponential $e^{-2\omega^2}$ factor. In practice, ω is commonly set to values at least one order of magnitude above 1, if not multiple orders of magnitude above that in certain cases (*e.g.*, high frequency audio signals). This leaves us with simply

$$\Sigma^{(1)}(x,\tilde{x}) = \operatorname{sinc}\Big(\sqrt{6}\,\omega\,(x-\tilde{x})\Big) + 1.$$

Notice that not only does our kernel reduce to the sinc function, but it also reduces to a function solely of $\Delta x = x - \tilde{x}$. This agrees with the shift-invariant property we observe in SIRENs, since the NNGP is dependent only on Δx, but not on the particular values of x and $\tilde{x}$. Notice also that ω defines the bandwidth of the sinc function, thus determining the maximum frequencies it allows to pass.

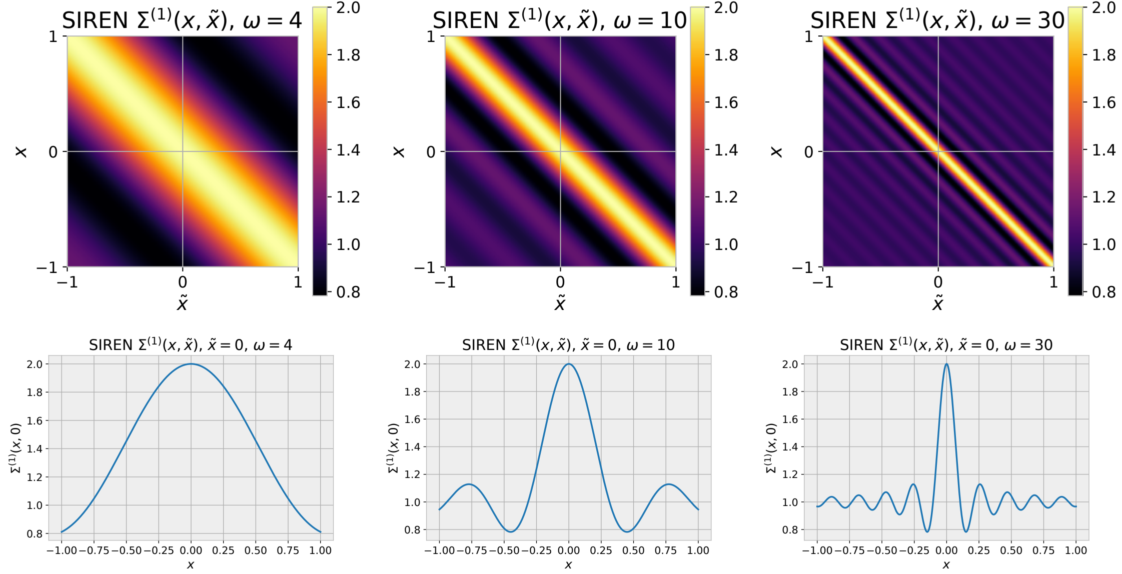


Figure 7.1: The NNGP for SIREN at different ω values. The top row shows the kernel values for pairs $(x, \tilde{x}) \in [-1, 1]^2$. Bottom row shows a slice at fixed $\tilde{x} = 0$.

The general sinc form and the shift-invariance of this kernel can be visualized in Figure 7.1, along with the effect of varying ω on the bandwidth of the NNGP kernel.

We analyzed the SIREN NNGP above due to its simplicity. However, the NNGP of a neural network does not paint a full picture of its behavior, since it models only its behavior at initialization, not after training. This is the role of the NTK which, under its limiting assumptions, models the behavior of the network after training. Nevertheless, the definition of the NTK, as can be seen from Theorem 7.1 is heavily dependent on the NNGP (and the closely related kernel $\dot{\Sigma}$). We can see that the NTK of the shallow SIREN, derived below, maintains the same relevant characteristics as the NNGP. We first derive $\dot{\Sigma}$ in the Lemma below.

Lemma 7.5. *For $\omega \in \mathbb{R}$, $\dot{\Sigma}^{(1)}(x, \tilde{x}) : \mathbb{R}^{n_0} \times \mathbb{R}^{n_0} \to \mathbb{R}$ is given by*

$$\Sigma^{(1)}(x, \tilde{x}) = \frac{c^2}{6} \left[\prod_{j=1}^{n_0} \operatorname{sinc}(c\,\omega\,(x_j - \tilde{x}_j)) + e^{-2\omega^2} \prod_{j=1}^{n_0} \operatorname{sinc}(c\,\omega\,(x_j + \tilde{x}_j)) \right] + 1.$$

Proof. The proof follows the same pattern as Theorem 7.4, with the only difference being a few sign changes after the exponential expansion of the trigonometric functions, due to the different identities for sine and cosine. □

Now we can derive the NTK for the shallow SIREN.

Corollary 7.6. *Shallow SIREN NTK.* *For a single hidden layer SIREN $f^{(1)} : \mathbb{R}^{n_0} \to \mathbb{R}^{n_2}$ following Definition 7.1, its neural tangent kernel (NTK), as defined in Theorem 7.1,*

is given by

$$\begin{aligned}\Theta^{(1)}(x,\tilde{x}) = \left(\omega^2\left(x^T\tilde{x}+1\right)\right)\left(\frac{c^2}{6}\left[\prod_{j=1}^{n_0}\operatorname{sinc}(c\,\omega\,(x_j-\tilde{x}_j)) - e^{-2\omega^2}\prod_{j=1}^{n_0}\operatorname{sinc}(c\,\omega\,(x_j+\tilde{x}_j))\right]+1\right)\\
+\frac{c^2}{6}\left[\prod_{j=1}^{n_0}\operatorname{sinc}(c\,\omega\,(x_j-\tilde{x}_j)) + e^{-2\omega^2}\prod_{j=1}^{n_0}\operatorname{sinc}(c\,\omega\,(x_j+\tilde{x}_j))\right]+1\\
=\frac{c^2}{6}\left(\omega^2\left(x^T\tilde{x}+1\right)+1\right)\prod_{j=1}^{n_0}\operatorname{sinc}(c\,\omega\,(x_j-\tilde{x}_j))\\
-\frac{c^2}{6}\left(\omega^2\left(x^T\tilde{x}+1\right)-1\right)e^{-2\omega^2}\prod_{j=1}^{n_0}\operatorname{sinc}(c\,\omega\,(x_j+\tilde{x}_j))+\omega^2\left(x^T\tilde{x}+1\right)+1.\end{aligned}$$

Proof. Follows trivially by applying Theorem 7.4 and Lemma 7.5 to Theorem 7.1. □

Though the expressions become more complex due to the formulation of the NTK, we can see that many of the same properties from the NNGP still apply. Again, for reasonable values of ω, the term with the exponential factor $e^{-2\omega^2}$ will be of negligible relative magnitude. With $c=\sqrt{6}$, this leaves us with

$$\left(\omega^2\left(x^T\tilde{x}+1\right)+1\right)\prod_{j=1}^{n_0}\operatorname{sinc}\left(\sqrt{6}\,\omega\,(x_j-\tilde{x}_j)\right)+\omega^2\left(x^T\tilde{x}+1\right)+1,$$

which is of the same form as the NNGP, with some additional linear terms $x^T\tilde{x}$. Though these linear terms break the pure shift-invariance, we still have a strong diagonal and the sinc form with bandwidth determined by ω, as can be seen in Figure 7.2.

Similarly to the NNGP, the SIREN NTK suggests that training a shallow SIREN is approximately equivalent to performing kernel regression with a sinc kernel, a low-pass filter, with its bandwidth defined by ω. This agrees intuitively with the experimental observation from Sitzmann et al. [2020] and Chapter 6 that in order to fit higher frequencies signals, a larger ω is required.

7.2.2 Simple sinusoidal network

Just as we did in the last section, we will now first derive the NNGP for a simple sinusoidal network, and then use that in order to obtain its NTK as well. As we will see, the Gaussian initialization employed in the SSN has the benefit of rendering the derivations cleaner, while retaining the relevant properties from the SIREN initialization.

Theorem 7.7. ***Shalow SSN NNGP.*** *For a single hidden layer simple sinusoidal network $f^{(1)}:\mathbb{R}^{n_0}\to\mathbb{R}^{n_2}$ following Definition 7.1, as the size of the hidden layer $n_1\to\infty$, $f^{(1)}$ tends (by law of large numbers) to the neural network Gaussian Process (NNGP) with covariance*

$$\Sigma^{(1)}(x,\tilde{x})=\frac{1}{2}\left(e^{-\frac{\omega^2}{2}\|x-\tilde{x}\|_2^2}-e^{-\frac{\omega^2}{2}\|x+\tilde{x}\|_2^2}e^{-2\omega^2}\right)+1.$$

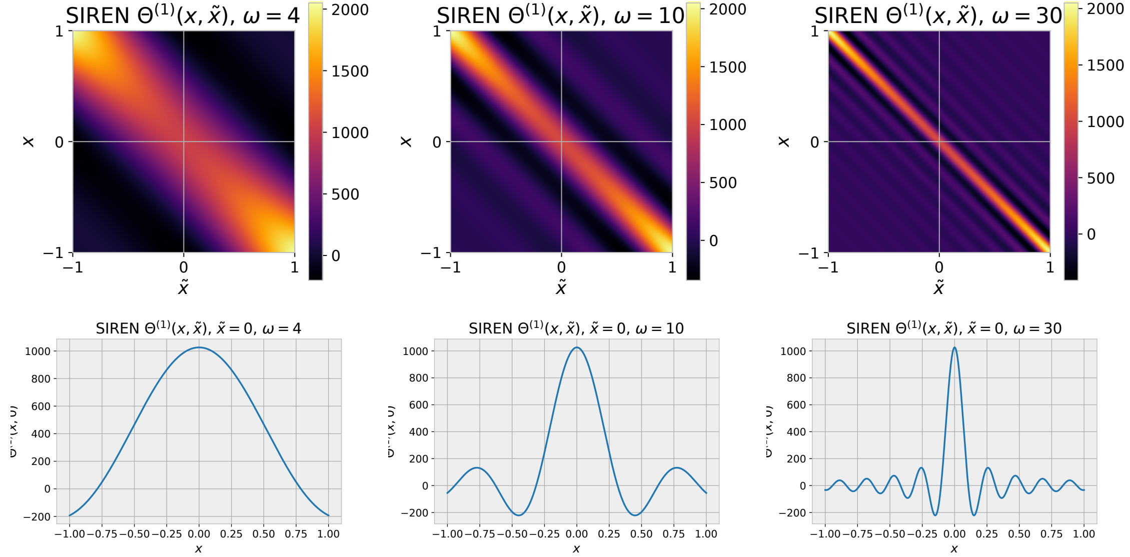


Figure 7.2: The NTK for SIREN at different ω values. The top row shows the kernel values for pairs $(x, \tilde{x}) \in [-1, 1]^2$. Bottom row shows a slice at fixed $\tilde{x} = 0$.

Proof. We again initially follow an approach similar to the one described in Lee et al. [2018a].

From our sinusoidal network definition, each element $f^{(1)}(x)_j$ in the output vector is a weighted combination of elements in $W^{(1)}$ and $b^{(1)}$. Conditioning on the outputs from the first layer ($L = 0$), since the sine function is bounded and each of the parameters is Gaussian with finite variance and zero mean, the $f^{(1)}(x)_j$ are also normally distributed with mean zero by the CLT. Since any subset of elements in $f^{(1)}(x)$ is jointly Gaussian, we have that this outer layer is described by a Gaussian process.

Therefore, its covariance is determined by $\sigma_W^2 \Sigma^{(1)} + \sigma_b^2 = \Sigma^{(1)} + 1$, where

$$\begin{aligned}
\Sigma^{(1)}(x, \tilde{x}) &= \lim_{n_1 \to \infty} \left[\frac{1}{n_1} \left\langle \sin\left(f^{(0)}(x)\right), \sin\left(f^{(0)}(\tilde{x})\right) \right\rangle \right] \\
&= \lim_{n_1 \to \infty} \left[\frac{1}{n_1} \sum_{j=1}^{n_1} \sin\left(f^{(0)}(x)\right)_j \sin\left(f^{(0)}(\tilde{x})\right)_j \right] \\
&= \lim_{n_1 \to \infty} \left[\frac{1}{n_1} \sum_{j=1}^{n_1} \sin\left(\omega \, \left(W_j^{(0)} x + b_j^{(0)}\right)\right) \sin\left(\omega \, \left(W_j^{(0)} \tilde{x} + b_j^{(0)}\right)\right) \right].
\end{aligned}$$

Now by the LLN the limit above converges to

$$\mathbb{E}_{w \sim \mathcal{N}(0, I_{n_0}), b \sim \mathcal{N}(0,1)} \left[\sin\left(\omega \, \left(w^T x + b\right)\right) \sin\left(\omega \, \left(w^T \tilde{x} + b\right)\right) \right],$$

where $w \in \mathbb{R}^{n_0}$ and $b \in \mathbb{R}$. Omitting the distributions from the expectation for brevity and

expanding the exponential definition of sine, we have

$$\mathbb{E}\left[\frac{1}{2i}\left(e^{i\omega\left(w^T x+b\right)} - e^{-i\omega\left(w^T x+b\right)}\right)\frac{1}{2i}\left(e^{i\omega\left(w^T \tilde{x}+b\right)} - e^{-i\omega\left(w^T \tilde{x}+b\right)}\right)\right]$$
$$= -\frac{1}{4}\mathbb{E}\left[e^{i\omega\left(w^T x+b\right)+i\omega\left(w^T \tilde{x}+b\right)} - e^{i\omega\left(w^T x+b\right)-i\omega\left(w^T \tilde{x}+b\right)} - e^{-i\omega\left(w^T x+b\right)+i\omega\left(w^T \tilde{x}+b\right)} + e^{-i\omega\left(w^T x+b\right)-i\omega\left(w^T \tilde{x}+b\right)}\right]$$
$$= -\frac{1}{4}\left[\mathbb{E}\left[e^{i\omega\left(w^T (x+\tilde{x})\right)}\right]\mathbb{E}\left[e^{2i\omega b}\right] - \mathbb{E}\left[e^{i\omega\left(w^T (x-\tilde{x})\right)}\right] - \mathbb{E}\left[e^{i\omega\left(w^T (\tilde{x}-x)\right)}\right] + \mathbb{E}\left[e^{i\omega\left(w^T (-x-\tilde{x})\right)}\right]\mathbb{E}\left[e^{-2i\omega b}\right]\right]$$

Applying Proposition 7.2 to each expectation above, it becomes

$$-\frac{1}{4}\left(e^{-\frac{\omega^2}{2}\|x+\tilde{x}\|_2^2}e^{-2\omega^2} - e^{-\frac{\omega^2}{2}\|x-\tilde{x}\|_2^2} - e^{-\frac{\omega^2}{2}\|x+\tilde{x}\|_2^2} + e^{-\frac{\omega^2}{2}\|x+\tilde{x}\|_2^2}e^{-2\omega^2}\right)$$
$$= \frac{1}{2}\left(e^{-\frac{\omega^2}{2}\|x-\tilde{x}\|_2^2} - e^{-\frac{\omega^2}{2}\|x+\tilde{x}\|_2^2}e^{-2\omega^2}\right).$$

□

We an once again observe that, for practical values of ω, the NNGP simplifies to

$$\frac{1}{2}e^{-\frac{\omega^2}{2}\|x-\tilde{x}\|_2^2} + 1.$$

This takes the form of a Gaussian kernel, which is also a low-pass filter, with its bandwidth determined by ω. We note that, similar to the $c = \sqrt{6}$ setting from SIRENs, in practice a scaling factor of $\sqrt{2}$ is applied to the normal activations, as described in Chapter 6, which cancels out the $1/2$ factors from the kernels, preserving the variance magnitude.

Moreover, we can also observe again that the kernel is a function solely of Δx, in agreement with the shift invariance that is also observed in simple sinusoidal networks. Visualizations of this NNGP are provided in Figure 7.3.

We will now proceed to derive the NTK, which requires first obtaining $\dot{\Sigma}$.

Lemma 7.8. *For $\omega \in \mathbb{R}$, $\dot{\Sigma}^{(1)}(x, \tilde{x}) : \mathbb{R}^{n_0} \times \mathbb{R}^{n_0} \to \mathbb{R}$ is given by*

$$\dot{\Sigma}^{(1)}(x, \tilde{x}) = \frac{1}{2}\left(e^{-\frac{\omega^2}{2}\|x-\tilde{x}\|_2^2} + e^{-\frac{\omega^2}{2}\|x+\tilde{x}\|_2^2}e^{-2\omega^2}\right) + 1.$$

Proof. The proof follows the same pattern as Theorem 7.7, with the only difference being a few sign changes after the exponential expansion of the trigonometric functions, due to the different identities for sine and cosine.

□

Corollary 7.9. ***Shallow SSN NTK.*** *For a simple sinusoidal network with a single hidden layer $f^{(1)} : \mathbb{R}^{n_0} \to \mathbb{R}^{n_2}$ following Definition 7.1, its neural tangent kernel (NTK), as defined*

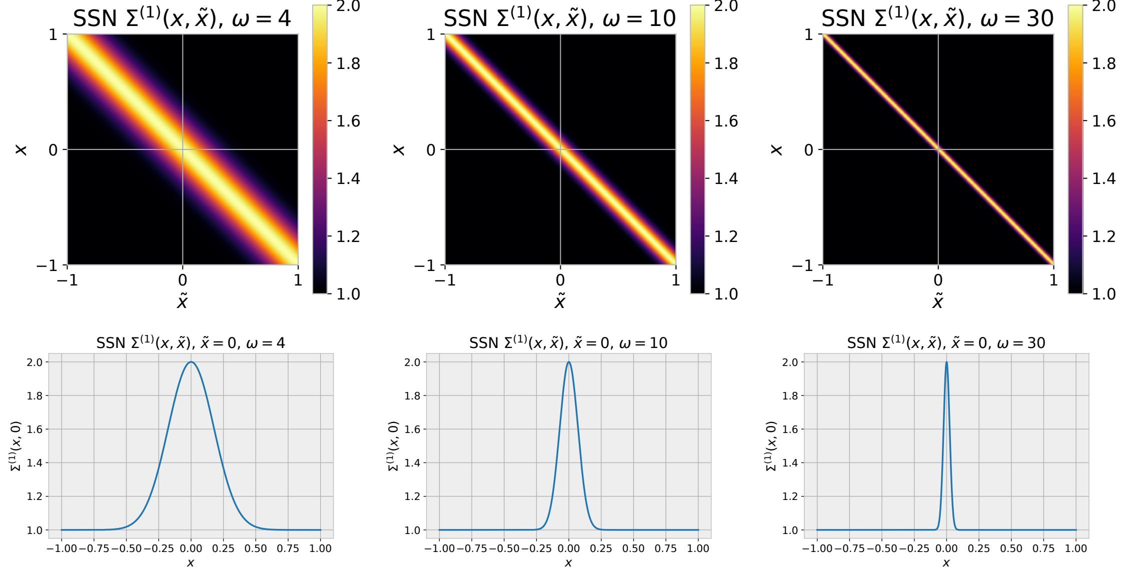


Figure 7.3: The NNGP for SSN at different ω values. The top row shows the kernel values for pairs $(x, \tilde{x}) \in [-1, 1]^2$. Bottom row shows a slice at fixed $\tilde{x} = 0$.

in Theorem 7.1, is given by

$$\begin{aligned}
\Theta^{(1)}(x, \tilde{x}) &= \left(\omega^2 \left(x^T \tilde{x} + 1\right)\right) \left[\frac{1}{2}\left(e^{-\frac{\omega^2}{2}\|x-\tilde{x}\|_2^2} + e^{-\frac{\omega^2}{2}\|x+\tilde{x}\|_2^2} e^{-2\omega^2}\right) + 1\right] \\
&\qquad + \frac{1}{2}\left(e^{-\frac{\omega^2}{2}\|x-\tilde{x}\|_2^2} - e^{-\frac{\omega^2}{2}\|x+\tilde{x}\|_2^2} e^{-2\omega^2}\right) + 1 \\
&= \frac{1}{2}\left(\omega^2\left(x^T \tilde{x} + 1\right) + 1\right) e^{-\frac{\omega^2}{2}\|x-\tilde{x}\|_2^2} \\
&\qquad - \frac{1}{2}\left(\omega^2\left(x^T \tilde{x} + 1\right) - 1\right) e^{-\frac{\omega^2}{2}\|x+\tilde{x}\|_2^2} e^{-2\omega^2} + \omega^2\left(x^T \tilde{x} + 1\right) + 1.
\end{aligned}$$

Proof. Follows trivially by applying Theorem 7.7 and Lemma 7.8 to Theorem 7.1. □

We again note the vanishing factor $e^{-2\omega^2}$, which leaves us with

$$\frac{1}{2}\left(\omega^2\left(x^T \tilde{x} + 1\right) + 1\right) e^{-\frac{\omega^2}{2}\|x-\tilde{x}\|_2^2} + \omega^2\left(x^T \tilde{x} + 1\right) + 1. \tag{7.2}$$

As with the SIREN before, this NTK is still of the same form as its corresponding NNGP. While again we have additional linear terms $x^T \tilde{x}$ in the NTK compared to the NNGP, in this case as well the kernel preserves its strong diagonal. It is still close to a Gaussian kernel, with its bandwidth determined directly by ω. We demonstrate this in Figure 7.4, where the NTK for different values of ω is shown. Additionally, we also plot a pure Gaussian kernel with variance ω^2, scaled to match the maximum and minimum values of the NTK. We can observe the NTK kernel closely matches the Gaussian. Moreover, we can also observe that, at $\tilde{x} = 0$ the maximum value is predicted by $k \approx \omega^2/2$, as expected from the scaling factors in the kernel in Equation 7.2.

This NTK suggests that training a simple sinusoidal network is approximately equivalent to performing kernel regression with a Gaussian kernel, a low-pass filter, with its bandwidth defined by ω. This also agrees intuitively with the observation from Chapter 6 that in order to fit higher frequencies signals, a larger ω is required.

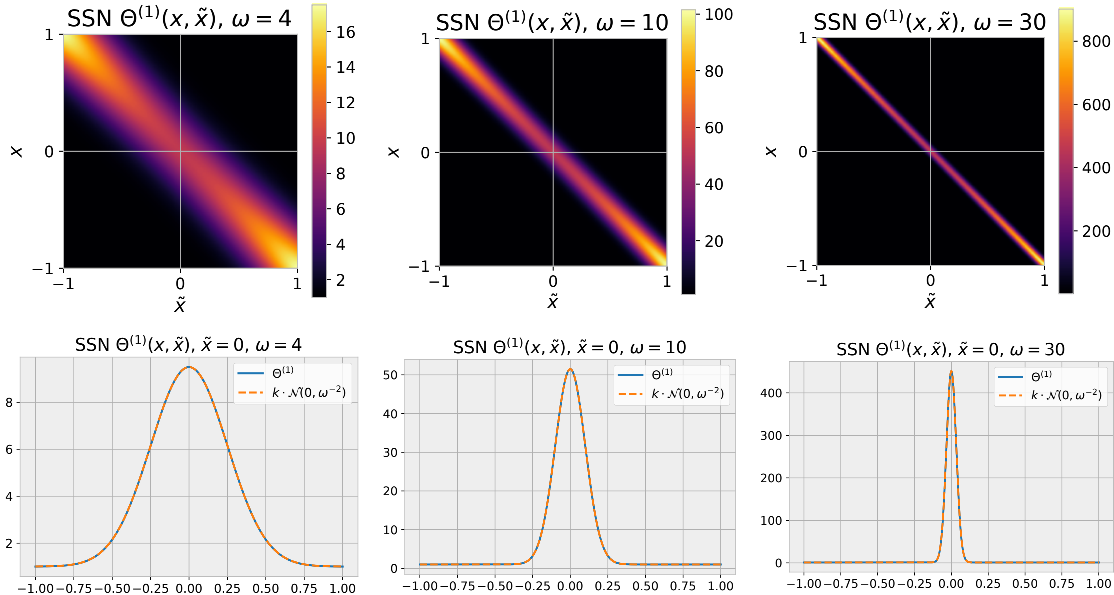


Figure 7.4: The NTK for SSN at different ω values. The top row shows the kernel values for pairs $(x, \tilde{x}) \in [-1, 1]^2$. Bottom row shows a slice at fixed $\tilde{x} = 0$, together with a Gaussian kernel scaled to match the maximum and minimum values of the NTK.

7.3 Deep sinusoidal networks

We will now look at the full NNGP and NTK for sinusoidal networks of arbitrary depth. As we will see, due to the recursive nature of these kernels, for networks deeper than the ones analyzed in the previous section, their full unrolled expressions quickly become intractable intuitively, especially for the NTK. Nevertheless, these kernels can still provide some insight, into the behavior of their corresponding networks. Moreover, despite their symbolic complexity, we will also demonstrate empirically that the resulting kernels can be approximated by simple Gaussian kernels, even for deep networks.

7.3.1 Simple sinusoidal network

As demonstrated in the previous section, simple sinusoidal networks produce simpler NNGP and NTK kernels due to their Gaussian initialization. We thus begin this section by now analyzing SSNs first, starting with their general NNGP.

Theorem 7.10. ***SSN NNGP.*** *For a simple sinusoidal network with L hidden layers $f^{(L)} : \mathbb{R}^{n_0} \to \mathbb{R}^{n_{L+1}}$ following Definition 7.1, as the size of the hidden layers $n_1, \ldots, n_L \to \infty$ sequentially, $f^{(L)}$ tends (by law of large numbers) to the neural network Gaussian Process (NNGP) with covariance $\Sigma^{(L)}(x, \tilde{x})$, recursively defined as*

$$\begin{aligned}
\Sigma^{(0)}(x, \tilde{x}) &= \omega^2 \left(x^T \tilde{x} + 1\right) \\
\Sigma^{(L)}(x, \tilde{x}) &= \frac{1}{2} e^{-\frac{1}{2}\left(\Sigma^{(L-1)}(x,x) + \Sigma^{(L-1)}(\tilde{x},\tilde{x})\right)} \left(e^{\Sigma^{(L-1)}(x,\tilde{x})} - e^{-\Sigma^{(L-1)}(x,\tilde{x})}\right) + 1.
\end{aligned}$$

Proof. We will proceed by induction on the depth L, demonstrating the NNGP for successive layers as $n_1, \ldots, n_L \to \infty$ sequentially. To demonstrate the base case $L = 1$, let us rearrange $\Sigma^{(1)}$ from Theorem 7.7 in order to express it in terms of inner products,

$$\begin{aligned}
\Sigma^{(1)}(x, \tilde{x}) &= \frac{1}{2} \left(e^{-\frac{\omega^2}{2}\|x-\tilde{x}\|_2^2} + e^{-\frac{\omega^2}{2}\|x+\tilde{x}\|_2^2} e^{-2\omega^2}\right) + 1 \\
&= \frac{1}{2} \left[e^{-\frac{\omega^2}{2}\left(x^T x - 2x^T \tilde{x} + \tilde{x}^T \tilde{x}\right)} - e^{-\frac{\omega^2}{2}\left(x^T x + 2x^T \tilde{x} + \tilde{x}^T \tilde{x}\right)} e^{-2\omega^2}\right] + 1 \\
&= \frac{1}{2} \left[e^{-\frac{1}{2}\left[\omega^2\left(x^T x + 1\right) + \omega^2\left(\tilde{x}^T \tilde{x} + 1\right)\right] + \omega^2\left(x^T \tilde{x} + 1\right)} - e^{-\frac{1}{2}\left[\omega^2\left(x^T x + 1\right) + \omega^2\left(\tilde{x}^T \tilde{x} + 1\right)\right] - \omega^2\left(x^T \tilde{x} + 1\right)}\right] + 1.
\end{aligned}$$

Given the definition of $\Sigma^{(0)}$, this is equivalent to

$$\frac{1}{2} e^{-\frac{1}{2}\left(\Sigma^{(0)}(x,x) + \Sigma^{(0)}(\tilde{x},\tilde{x})\right)} \left(e^{\Sigma^{(0)}(x,\tilde{x})} - e^{-\Sigma^{(0)}(x,\tilde{x})}\right) + 1,$$

which concludes this case.

Now given the inductive hypothesis, as $n_1, \ldots, n_{L-1} \to \infty$ we have that the first $L - 1$ layers define a network $f^{(L-1)}$ with NNGP given by $\Sigma^{(L-1)}(x, \tilde{x})$. Now it is left to show that as $n_L \to \infty$, we get the NNGP given by $\Sigma^{(L)}$. Following the same argument in Theorem 7.7, the network

$$f^{(L)}(x) = W^{(L)} \frac{1}{\sqrt{n_L}} \sin\left(f^{(L-1)}\right) + b^{(L)}$$

constitutes a Gaussian process given the outputs of the previous layer, due to the distributions of $W^{(L)}$ and $b^{(L)}$. Its covariance is given by $\sigma_W^2 \Sigma^{(L)} + \sigma_b^2 = \Sigma^{(L)} + 1$, where

$$\begin{aligned}
\Sigma^{(L)}(x, \tilde{x}) &= \lim_{n_L \to \infty} \left[\frac{1}{n_L} \left\langle \sin\left(f^{(L-1)}(x)\right), \sin\left(f^{(L-1)}(\tilde{x})\right)\right\rangle\right] \\
&= \lim_{n_L \to \infty} \left[\frac{1}{n_L} \sum_{j=1}^{n_L} \sin\left(f^{(L-1)}(x)\right)_j \sin\left(f^{(L-1)}(\tilde{x})\right)_j\right].
\end{aligned}$$

By inductive hypothesis, $f^{(L-1)}$ is a Gaussian process $\Sigma^{(L-1)}(x, \tilde{x})$. Thus by the LLN the limit above equals

$$\mathbb{E}_{(u,v)\sim\mathcal{N}\left(0, \Sigma^{(L-1)}(x,\tilde{x})\right)} \left[\sin(u) \sin(v)\right].$$

Omitting the distribution from the expectation for brevity and expanding the exponential definition of sine, we have

$$\mathbb{E}\left[\frac{1}{2i}\left(e^{iu}-e^{-iu}\right)\frac{1}{2i}\left(e^{iv}-e^{-iv}\right)\right]=-\frac{1}{4}\left[\mathbb{E}\left[e^{i(u+v)}\right]-\mathbb{E}\left[e^{i(u-v)}\right]-\mathbb{E}\left[e^{-i(u-v)}\right]+\mathbb{E}\left[e^{-i(u+v)}\right]\right].$$

Since u and v are jointly Gaussian, $p=u+v$ and $m=u-v$ are also Gaussian, with mean 0 and variance

$$\begin{aligned}\sigma_p^2&=\sigma_u^2+\sigma_v^2+2\operatorname{Cov}[u,v]=\Sigma^{(L-1)}(x,x)+\Sigma^{(L-1)}(\tilde{x},\tilde{x})+2\,\Sigma^{(L-1)}(x,\tilde{x}),\\ \sigma_m^2&=\sigma_u^2+\sigma_v^2-2\operatorname{Cov}[u,v]=\Sigma^{(L-1)}(x,x)+\Sigma^{(L-1)}(\tilde{x},\tilde{x})-2\,\Sigma^{(L-1)}(x,\tilde{x}).\end{aligned}$$

We can now rewriting the expectations in terms of normalized variables

$$-\frac{1}{4}\left[\mathbb{E}_{z\sim\mathcal{N}(0,1)}\left[e^{i\sigma_p z}\right]-\mathbb{E}_{z\sim\mathcal{N}(0,1)}\left[e^{i\sigma_m z}\right]-\mathbb{E}_{z\sim\mathcal{N}(0,1)}\left[e^{-i\sigma_m z}\right]+\mathbb{E}_{z\sim\mathcal{N}(0,1)}\left[e^{-i\sigma_p z}\right]\right].$$

Applying Proposition 7.2 to each expectation, we get

$$\begin{aligned}&\frac{1}{2}\left[e^{-\frac{1}{2}\sigma_m^2}-e^{-\frac{1}{2}\sigma_p^2}\right]\\ &\quad=\frac{1}{2}\left[e^{-\frac{1}{2}\left(\Sigma^{(L-1)}(x,x)+\Sigma^{(L-1)}(\tilde{x},\tilde{x})-2\,\Sigma^{(L-1)}(x,\tilde{x})\right)}-e^{-\frac{1}{2}\left(\Sigma^{(L-1)}(x,x)+\Sigma^{(L-1)}(\tilde{x},\tilde{x})+2\,\Sigma^{(L-1)}(x,\tilde{x})\right)}\right]\\ &\quad=\frac{1}{2}e^{-\frac{1}{2}\left(\Sigma^{(L-1)}(x,x)+\Sigma^{(L-1)}(\tilde{x},\tilde{x})\right)}\left(e^{\Sigma^{(L-1)}(x,\tilde{x})}-e^{-\Sigma^{(L-1)}(x,\tilde{x}))}\right)\end{aligned}$$

□

Unrolling the definition beyond $L=1$ leads to expressions that are difficult to parse. However, without unrolling, we can rearrange the terms in the NNGP above as

$$\begin{aligned}\Sigma^{(L)}(x,\tilde{x})&=\frac{1}{2}e^{-\frac{1}{2}\left(\Sigma^{(L-1)}(x,x)+\Sigma^{(L-1)}(\tilde{x},\tilde{x})\right)}\left(e^{\Sigma^{(L-1)}(x,\tilde{x})}-e^{-\Sigma^{(L-1)}(x,\tilde{x})}\right)+1\\ &=\frac{1}{2}\left[e^{-\frac{1}{2}\left(\Sigma^{(L-1)}(x,x)-2\Sigma^{(L-1)}(x,\tilde{x})+\Sigma^{(L-1)}(\tilde{x},\tilde{x})\right)}-e^{-\frac{1}{2}\left(\Sigma^{(L-1)}(x,x)+2\Sigma^{(L-1)}(x,\tilde{x})+\Sigma^{(L-1)}(\tilde{x},\tilde{x})\right)}\right]+1.\end{aligned}$$

Since the covariance *matrix* $\Sigma^{(L-1)}$ is positive semi-definite, we can observe that the exponent expressions can be reformulated into a quadratic forms analogous to the ones in Theorem 7.7. We can thus observe that the same structure is essentially preserved through the composition of layers, except for the ω factor present in the first layer. Moreover, given this recursive definition, since the NNGP at any given depth L is a function only of the preceding kernels, the resulting kernel will also be shift-invariant.

Let us now derive the $\dot{\Sigma}$ kernel, required for the NTK.

Lemma 7.11. *For $\omega\in\mathbb{R}$, $\dot{\Sigma}^{(L)}(x,\tilde{x}):\mathbb{R}^{n_0}\times\mathbb{R}^{n_0}\to\mathbb{R}$, is given by*

$$\dot{\Sigma}^{(L)}(x,\tilde{x})=\frac{1}{2}e^{-\frac{1}{2}\left(\Sigma^{(L-1)}(x,x)+\Sigma^{(L-1)}(\tilde{x},\tilde{x})\right)}\left(e^{\Sigma^{(L-1)}(x,\tilde{x})}+e^{-\Sigma^{(L-1)}(x,\tilde{x})}\right)+1.$$

Proof. The proof follows the same pattern as Theorem 7.10, with the only difference being a few sign changes after the exponential expansion of the trigonometric functions, due to the different identities for sine and cosine. □

As done in the previous section, it would be simple to now derive the full NTK for a simple sinusoidal network of arbitrary depth by applying Theorem 7.1 with the NNGP kernels from above. However, there is not much to be gained by writing the convoluted NTK expression explicitly, beyond what we have already gleaned from the NNGP above.

Nevertheless, some insight can be gained from the recursive expression of the NTK itself, as defined in Theorem 7.1. First, note that, as before, for practical values of ω, $\dot{\Sigma} \approx \Sigma$, both converging to simply a single Gaussian kernel. Thus, our NTK recursion becomes

$$\Theta^{(L)}(x, \tilde{x}) \approx \left(\Theta^{(L-1)}(x, \tilde{x}) + 1\right) \Sigma^{(L)}(x, \tilde{x}).$$

Now, note that when expanded, the form of this NTK recursion is essentially as a product of the Gaussian Σ kernels,

$$\begin{aligned}\Theta^{(L)}(x, \tilde{x}) &\approx \Big(\Big(\dots\Big(\Big(\Sigma^{(0)}(x, \tilde{x}) + 1\Big)\, \Sigma^{(1)}(x, \tilde{x}) + 1\Big)\dots\Big)\, \Sigma^{(L-1)}(x, \tilde{x}) + 1\Big)\, \Sigma^{(L)}(x, \tilde{x}) \\ &= \Big(\Big(\dots\Big(\Big(\omega^2 \left(x^T \tilde{x} + 1\right) + 1\Big)\, \Sigma^{(1)}(x, \tilde{x}) + 1\Big)\dots\Big)\, \Sigma^{(L-1)}(x, \tilde{x}) + 1\Big)\, \Sigma^{(L)}(x, \tilde{x}).\end{aligned} \tag{7.3}$$

We know that the product of two Gaussian kernels is Gaussian and thus the general form of the kernel should be approximately a sum of Gaussian kernels. As long as the magnitude of one of the terms dominates the sum, the overall resulting kernel will be approximately Gaussian. Empirically, we observe this to be the case, with the inner term containing ω^2 dominating the sum, for reasonable values (*e.g.*, $\omega > 1$ and $L < 10$). In Figure 7.5, we show the NTK for networks of varying depth and ω, together with a pure Gaussian kernel of variance ω^2, scaled to match the maximum and minimum values of the NTK. We can observe that the NTKs are still approximately Gaussian, with their maximum value approximated by $k \approx \frac{1}{2^L}\omega^2$, as expected from the product of ω^2 and L kernels above. We also observe that the width of the kernels is mainly defined by ω.

7.3.2 SIREN

For completeness, in this section we will derive the full SIREN NNGP and NTK. As discussed previously, both the SIREN and the simple sinusoidal network have kernels that approximate low-pass filters. Due to the SIREN initialization, its NNGP and NTK were previously shown to have more complex expressions. However, we will show in this section that the sinc kernel that arises from the shallow SIREN is gradually "dampened" as the depth of the network increases, gradually approximating a Gaussian kernel.

Theorem 7.12. ***SIREN NNGP.*** *For a SIREN with L hidden layers $f^{(L)} : \mathbb{R}^{n_0} \to \mathbb{R}^{n_{L+1}}$ following Definition 7.1, as the size of the hidden layers $n_1, \dots, n_L \to \infty$ sequentially, $f^{(L)}$ tends (by law of large numbers) to the neural network Gaussian Process (NNGP) with*

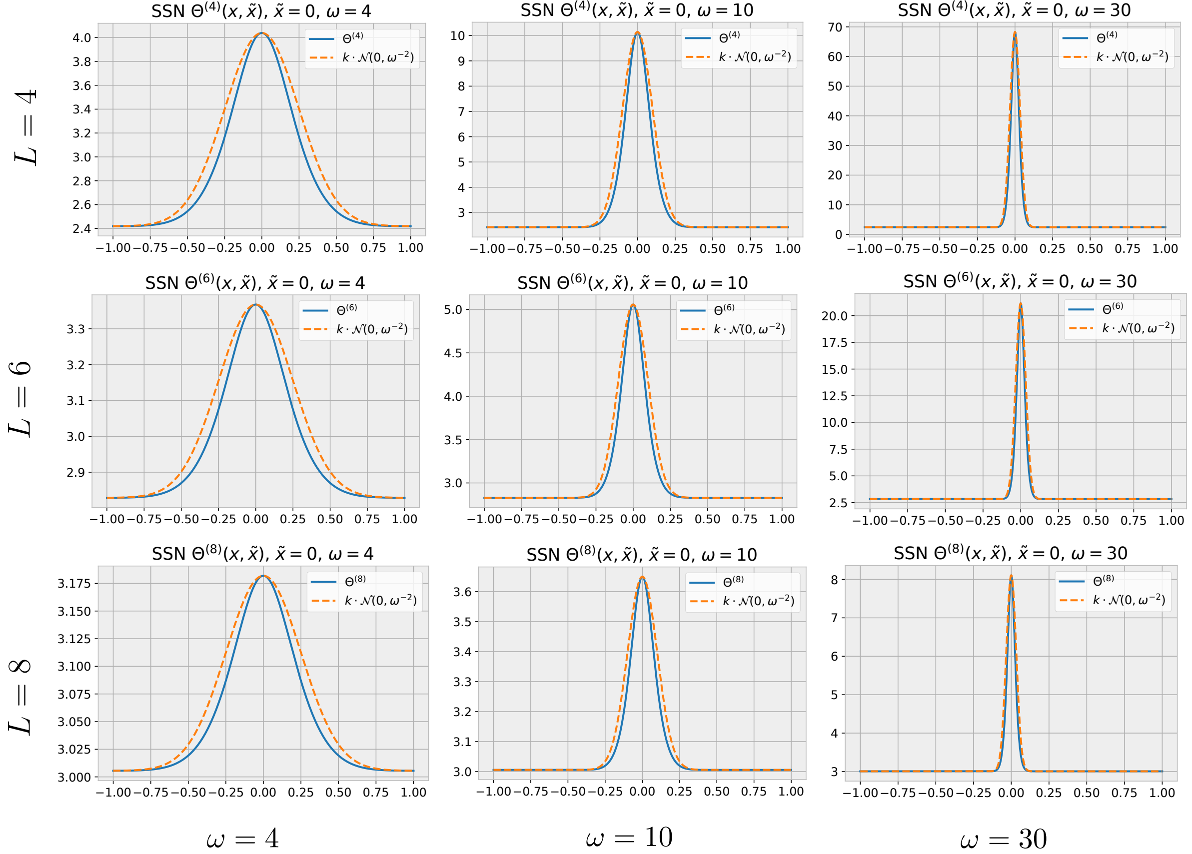


Figure 7.5: The NTK for SSN at different ω and network depth (L) values. Kernel values at a slice for fixed $\tilde{x} = 0$ are shown, together with a Gaussian kernel scaled to match the maximum and minimum values of the NTK.

covariance $\Sigma^{(L)}(x, \tilde{x})$*, recursively defined as*

$$\Sigma^{(1)}(x, \tilde{x}) = \frac{c^2}{6}\left[\prod_{j=1}^{n_0} \operatorname{sinc}(c\,\omega\,(x_j - \tilde{x}_j)) - e^{-2\omega^2} \prod_{j=1}^{n_0} \operatorname{sinc}(c\,\omega\,(x_j + \tilde{x}_j))\right] + 1$$

$$\Sigma^{(L)}(x, \tilde{x}) = \frac{1}{2} e^{-\frac{1}{2}\left(\Sigma^{(L-1)}(x,x) + \Sigma^{(L-1)}(\tilde{x},\tilde{x})\right)} \left(e^{\Sigma^{(L-1)}(x,\tilde{x})} - e^{-\Sigma^{(L-1)}(x,\tilde{x})}\right) + 1.$$

Proof. Intuitively, after the first hidden layer, the inputs to every subsequent hidden layer are of infinite width, due to the NNGP assumptions. Therefore, due to the CLT, the pre-activation values at every layer are Gaussian, and the NNGP is unaffected by the uniform weight initialization (compared to the Gaussian weight initialization case). The only layer for which this is not the case is the first layer, since the input size is fixed and finite. This gives rise to the different $\Sigma^{(1)}$.

Formally, this proof proceed by induction on the depth L, demonstrating the NNGP for successive layers as $n_1, \ldots, n_L \to \infty$ sequentially. The base case comes straight from Theorem 7.4. After the base case, the proof follows exactly the same as in Theorem 7.10. □

For the same reasons as in the proof above, the $\dot{\Sigma}$ kernels after the first layer are also equal to the ones for the simple sinusoidal network, given in Lemma 7.11.

Given the similarity of the kernels beyond the first layer, the interpretation of this NNGP is the same as discussed in the previous section for the simple sinusoidal network.

Analogously to the SSN case before, the SIREN NTK expansion can also be approximated as a product of Σ kernels, as in Equation 7.3. The product of a sinc function with $L-1$ subsequent Gaussians "dampens" the sinc, such that as the network depth increases the NTK approaches a Gaussian, as can be seen in Figure 7.6.

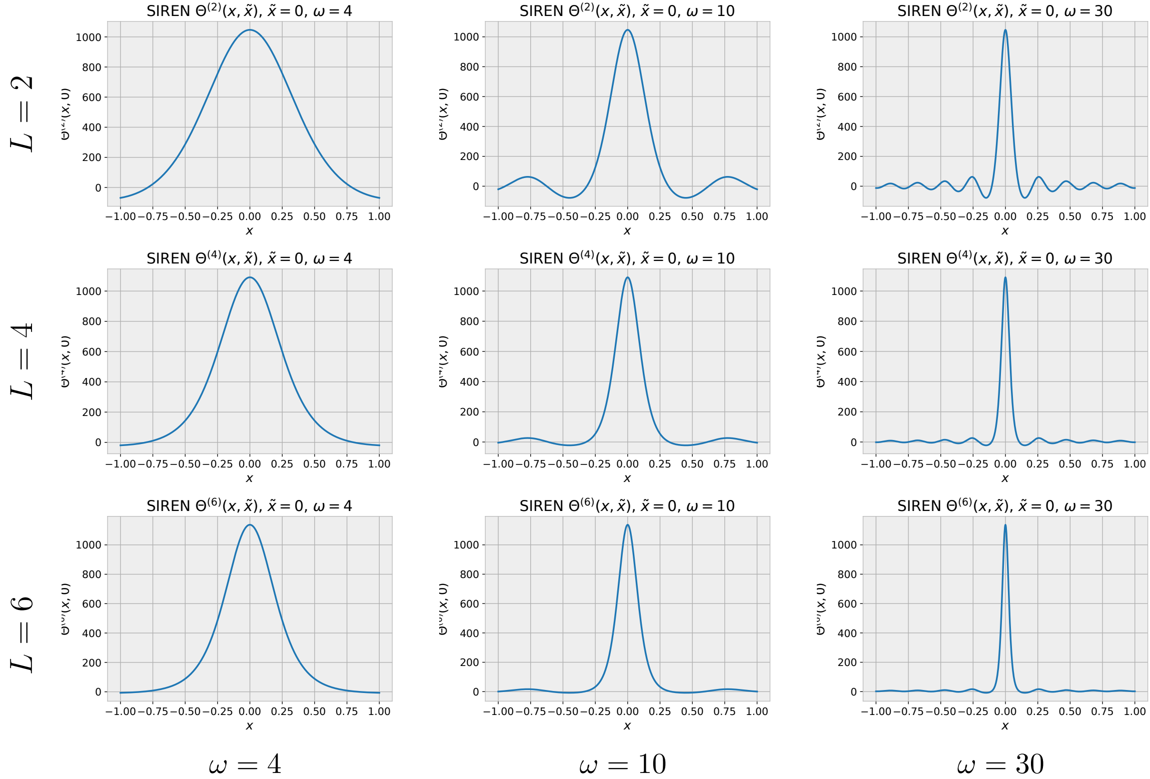


Figure 7.6: The NTK for SIREN at different ω and network depth (L) values. Kernel values at a slice for fixed $\tilde{x}=0$ are shown.

7.4 Empirical Analysis

As shown above, neural tangent kernel theory suggests that sinusoidal networks work as low-pass filters, with their bandwidth controlled by the parameter ω. However, the networks in NTK analysis are only theoretical extrapolations, for example assuming infinite width and infinitesimal learning rate. In this section, we demonstrate empirically that we can observe this predicted behavior even in real sinusoidal networks.

For this experiment, we generate a 512×512 monochromatic image by super-imposing

two orthogonal sinusoidal signals, each consisting of a single frequency. That is,

$$f(x, y) = \cos(128\pi x) + \cos(32\pi y). \tag{7.4}$$

This function is sampled in the domain $[-1, 1]^2$ to generate the image in Figure 7.7.

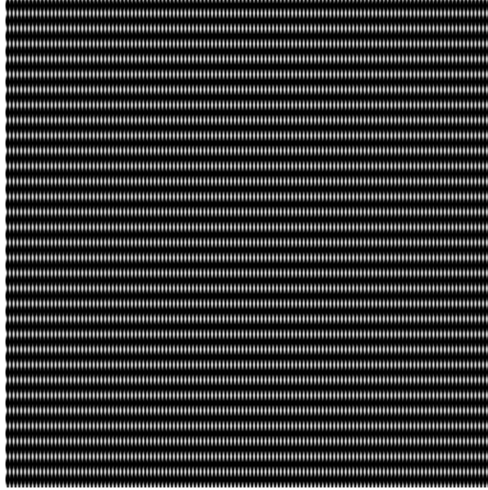


Figure 7.7: The test signal used to analyze the behavior of sinusoidal networks. It is created from two orthogonal, single-frequency signals, $f(x, y) = \cos(128\pi x) + \cos(32\pi y)$.

To demonstrate what we can expect from applying low-pass filters of different bandwidths to this signal, we take the Discrete Fourier Transform (DFT) of the image, cut off frequencies above a certain value, and then perform an inverse Fourier transform to recover the (now filtered) original image. We perform this same operation for various cutoff frequencies, using each value from 0 (all signal is lost) to 256 (signal fully recovered). The mean squared error (MSE) of the reconstruction, as a function of the cutoff frequency, is shown in Figure 7.8. We can see that due to the simple nature of the signal, containing only two frequencies, there are only three loss levels. When the cutoff frequency is smaller than the smallest frequency in Equation 7.4, our loss is at its highest – no signal is getting through. For cutoffs between the two base frequencies, we get an intermediate loss up until the point our low-pass cutoff is above the highest frequency. At that point the reconstruction is essentially perfect, with higher cutoffs not having a significant effect.

If indeed the NTK analysis is correct and sinusoidal networks act as low-pass filters, with their bandwidth controlled by ω, we should be able to observe the same behavior by the Fourier transform low-pass filter from above with sinusoidal networks of different bandwidth.

We reproduce this experiment by fitting the same image from Figure 7.7 using sinusoidal networks with different values of ω. We plot the final training loss for networks with different ω in Figure 7.9 and training curves in Figure 7.10.

We can observe, again, that there are three consistent loss levels following the magnitude of the ω parameter, confirming the intuition that, as in the DFT example above, the sinusoidal network is also working as a low-pass filter. This is also observable in Figure 7.11,

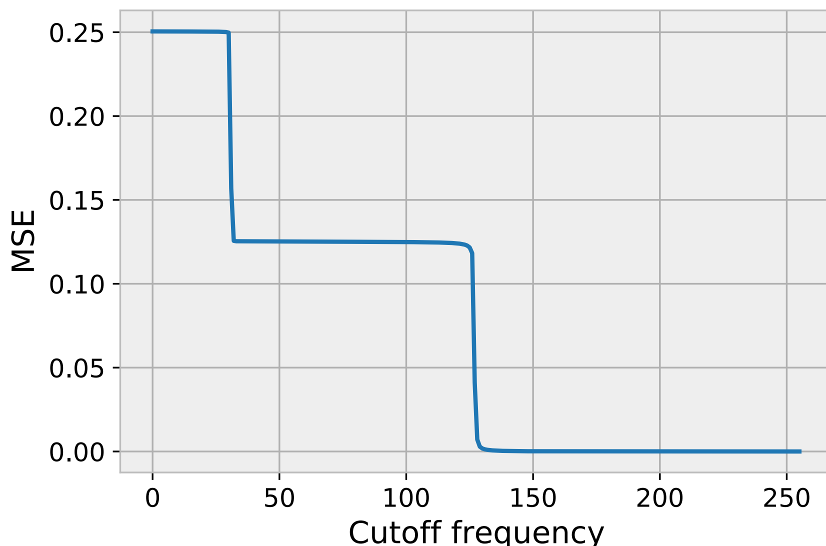


Figure 7.8: Reconstruction loss for different cutoff frequencies for a low-pass filter applied to Figure 7.7. The three loss levels reflect the 2 frequencies present in the simple signal: 8 and 32.

where we see example reconstructions for networks of various ω values after training. For small ω, none of the frequencies can pass, and the loss is high. For intermediate values, only the lower frequency is fitted, resulting in intermediate loss. Finally, for ω values above a certain high enough point, the image is perfectly reconstructed, resulting in loss close to 0.

However, unlike with the DFT low-pass filter (which does not involve any learning), we see in Figure 7.10 that during training some sinusoidal networks shift from one loss level to a lower one. This demonstrates the fact that sinusoidal networks differ from regular low-pass filters in that their weights can change, which implies that the bandwidth defined by ω can change to some extent with learning. We know the weights W_1 in the first layer of a sinusoidal network, given by

$$f_1(x) = \sin\left(\omega \cdot W_1^T x + b_1\right), \tag{7.5}$$

will change with training. Empirically, we observe that the spectral norm of W_1 increases throughout training for small ω values. We can interpret that as the overall magnitude of the term $\omega \cdot W_1^T x$ increasing, which is functionally equivalent to an increase in ω itself, and thus an increase in bandwidth. NTK analysis circumvents this issue by relying on an infinitesimal learning rate, such that the magnitude of W_1 does not change much during training. In practice, however, we should expect to see some drift in the bandwidth of the sinusoidal network as learning progresses. This is also likely to affect smaller ω values more strongly, as a smaller change in magnitude has a larger impact on the overall magnitude. We observe this in Figure 7.9, where for smaller ω values we achieve lower reconstruction loss faster than the expectation from the DFT would suggest.

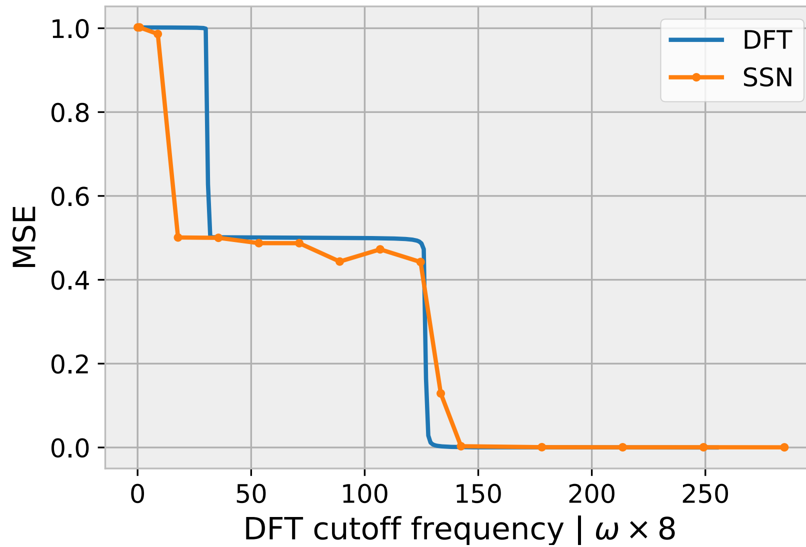


Figure 7.9: Final loss, after fitting the image in Figure 7.7 with sinusoidal networks, for different values of ω, superimposed on the DFT loss values from Figure 7.8. Values of ω are scaled to align with the DFT values. The three loss levels, analogous to the DFT, reflect the 2 frequencies present in the simple signal, and demonstrate that the sinusoidal network is indeed acting as a low-pass filter with bandwidth defined by ω.

In Figure 7.10, we observe that sinusoidal networks with smaller values of ω take a longer time to achieve a lower loss (if at all). Intuitively, this happens because, due to the effect described above, lower ω values require a larger increase in magnitude by the weights W_1. Given that all networks were trained with the same learning rate, the ones with a smaller ω require their weights to move a longer distance, and thus take more training steps to achieve a lower loss. In cases of very small ω, the "appropriate" weight magnitude to compensate might be so large that the network never reaches that point, and remains limited to a lower bandwidth, and incapable of learning the full signal, as we can observe for small ω values in Figures 7.10 and 7.11.

7.5 Tuning ω

According to the NTK analysis and empirical results above, a sinusoidal network acts as a low-pass filter, and its band can be tuned through its ω parameter. Though the bandwidth of the filter change throughout training, as observed in the previous section, the initial configuration of the network still influences how easily and quickly (if at all) it can learn a given signal.

It is therefore of great importance to be able to pick appropriate ω values for a given signal or task. Given the low-pass nature of the networks, a naive choice might be to simply pick a high enough value. However, it is often not the case that we simply want

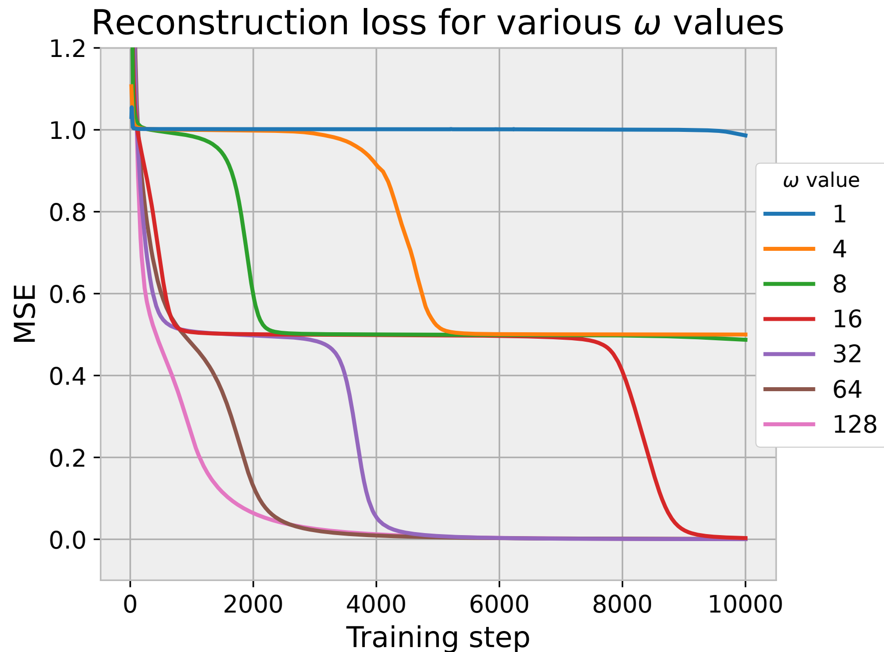
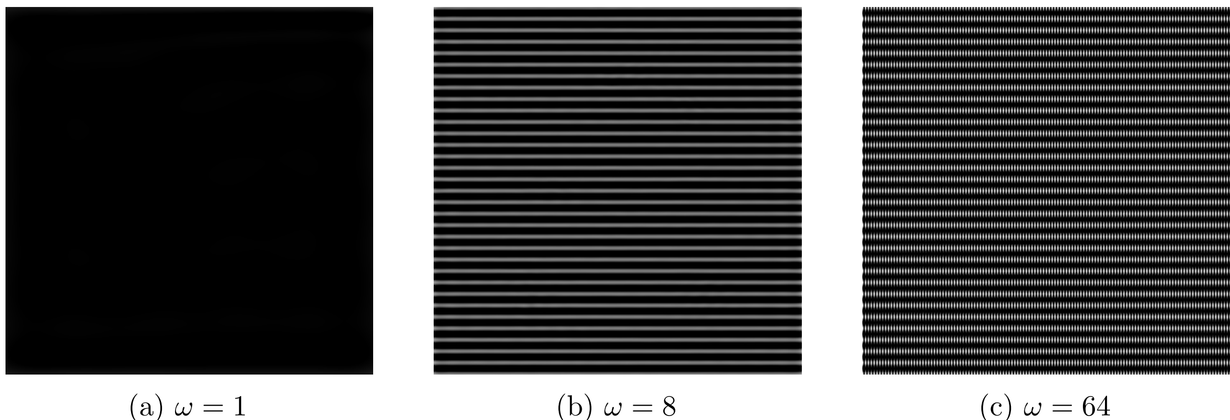


Figure 7.10: Fitting the image in Figure 7.7 with sinusoidal networks with different values of ω. The three loss levels reflect the 2 frequencies present in the simple signal, and demonstrate that the sinusoidal network is indeed acting as a low-pass filter with bandwidth defined by ω.

to fit only the exact training samples. Instead, it is most commonly the case that we care about finding a good interpolation (*i.e.*, generalizing well). Allowing the network to model too large frequencies, larger than the ones present in the actual signal, is likely to generate overfitting artifacts and poor generalization. This is demonstrated empirically in Figure 7.13.

Thus, the most appropriate choice is to tune the network to the highest frequency present in the signal. However, this poses a few challenges. First, we do not always have the knowledge of what is the value of the highest frequency in the true underlying signal of interest. Moreover, even when we have that information, it is not clear yet how to translate that into a choice of ω. As seen in the previous analyses, the value of ω does not correspond directly to the absolute numerical value of the network's cut off frequency. Moreover, we have also observed that, since the network learns and its weights change in magnitude, that value in fact changes with training.

Therefore, the most we can hope for is to have heuristics to guide the choice of ω based on the nature of the filter. Nevertheless, having a reasonable guess for ω is also likely sufficient for good performance, precisely due to the ability of the network to change during training.



(a) $\omega = 1$ (b) $\omega = 8$ (c) $\omega = 64$

Figure 7.11: Examples of the reconstructed signal for sinusoidal networks for different values of ω. Each is a qualitative for each of the loss levels in Figure 7.10.

7.5.1 Choosing ω from the Nyquist frequency

One source of empirical information on the relationship between ω and the sinusoidal network's "learnable frequencies", is the empirical analysis from the previous section. Taking into account the scaling, we can see from Figure 7.9 that around $\omega = 16$ the network starts being able to learn the full signal (frequency 128). As can be inferred by the figure scaling itself, this suggests a heuristic of setting ω to about 1/8 the maximum frequency in the signal. We can also see from Figure 7.10 that at about $\omega = 4$ the sinusoidal network starts to be able to efficiently learn a signal with frequency 32 – but not the signal with frequency 128.

For natural signals, such as pictures, it is common for frequencies close to the Nyquist frequency of the discrete sampling to be present. We provide an example for the "camera" image we have utilized so far in Figure 7.12, where we can see that the reconstruction loss through a low-pass filter (as done previously with the DFT) continues to decrease significantly up to the Nyquist frequency for the image resolution. In light of this information, analyzing the choices of ω above which results do not improve for the experiments in Chapter 6 again suggests that ω should be set around 1/8 of the Nyquist frequency of the signal. These values of ω are summarized in Table 7.1 in the "Fitting ω" column.

For example, the image fitting experiment shows that, for an image of shape 512×512 (and thus Nyquist frequency of 256 for each dimension), this heuristic suggests an ω value of $256/8 = 32$, which is the value found to work best empirically through search.

We find similar results for the audio fitting experiments. The audio signals used in the audio fitting experiment contained approximately $300,000$ and $500,000$ points. Notice that since we re-normalize all inputs to a common range, the actual audio sampling rate of $44,100$ Hz can be disregarded. This implies these signals have maximum frequencies of approximately $150,00$ and $250,000$. This would suggest reasonable values for ω of $18,750$ and $31,250$, which are close to the ones found empirically to work well.

In examples such as the video fitting experiments, in which each dimension has a different frequency, it is not completely clear how to pick a single ω to fit all dimensions.

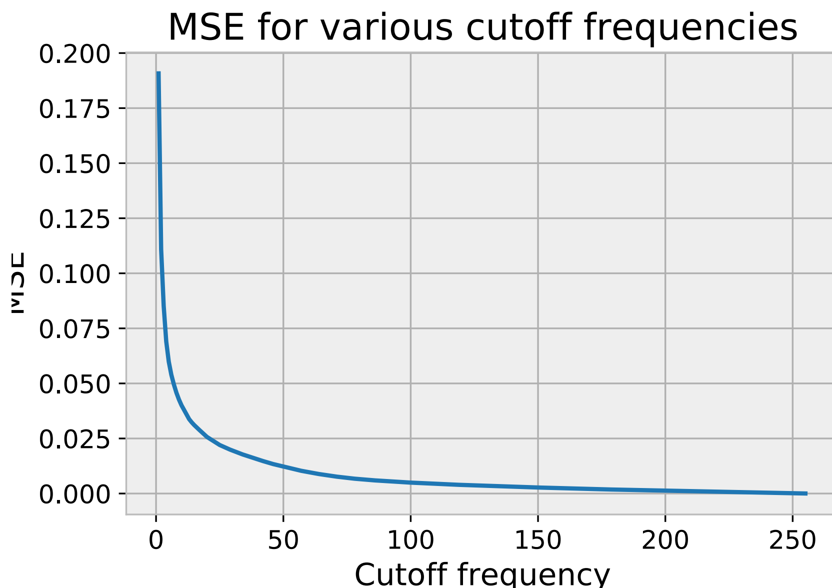


Figure 7.12: Reconstruction loss through a low-pass filter for the "camera" image. Notice the loss continues to go down up to the Nyquist frequency.

This suggests that having independent values of ω for each dimension might be useful for such cases. We analyze that possibility in the following section. We note that even in this setting, as will be show in the experiments below, applying the heuristic to each dimension independently gives an (multi-dimensional) ω with good performance. Moreover, eve if using scalar ω value, simply applying the heuristic to the minimum valued dimension yields good performance.

Finally, when performing the generalization experiments in Section 7.6, we show the best performing ω ended up being half the value of the best ω used in the fitting tasks from Chapter 6. This follows intuitively, since for the generalization task we set apart half the points for training and the other half for testing, thus dividing the maximum possible frequency in the training sample in half, providing further evidence of the relationship between ω and the maximum frequency in the input signal.

7.5.2 Multi-dimensional ω

In many problems, such as the video fitting and PDE problems we saw in Chapter 6, not only is the input space multi-dimensional, it also contains time and space dimensions (which are additionally possibly of different shape). Overall, the input signal might have a spectrum that is fundamentally different for different dimensions. Consequently, having a single scalar ω to tune the bandwidth of a sinusoidal network might be too crude an approach. This suggests that employing a multi-dimensional ω, specifying different frequencies for each dimension might be beneficial.

In practice, if we employ a scaling factor $\lambda = \begin{bmatrix}\lambda_1 & \lambda_2 & \dots & \lambda_d\end{bmatrix}^T$, we have the first layer

of the sinusoidal network given by

$$f_1(x) = \sin(\omega\,(W_1\,(\lambda \odot x) + b_1)) \tag{7.6}$$
$$= \sin(W_1\,(\Omega \odot x) + \omega b_1), \tag{7.7}$$

where $\odot$ is the elementwise (Hadamard) product, and thus $\Omega = \begin{bmatrix}\lambda_1\omega & \lambda_2\omega & \dots & \lambda_d\omega\end{bmatrix}^T$ works essentially as a multi-dimensional ω.

In the following experiments, we employ this approach to three-dimensional problems, in which we have time and differently shaped space domains, namely the video fitting and physics-informed neural network PDE experiments. For these experiments, we report the ω in the form of the (already scaled) Ω vector for simplicity.

7.5.3 Choosing ω from available information

Finally, in many problems we either have some knowledge of the underlying signal, or we have related data we can leverage. For example, if we are fitting the solution to a simple PDE, to which we might know the form of the solution, we can use that information to infer which choice of ω to employ.

Even though the particular setting in which we already know the solution to the problem is not completely realistic, in many realistic cases we are working with inverse problems. For example, let's say we have velocity fields for a fluid and we are trying to solve for the coupled pressure field and the Reynolds number using a physics-informed neural network (this experiment is performed in Section 7.6). In this case, we have access to two components of the solution field. Performing a Fourier transform on the training data we have can reveal the relevant spectrum and inform our choice of ω. If the maximum frequency in the signal is lower than the Nyquist frequency implied by the sampling for our training data, this can lead to a more appropriate choice of ω than suggested purely from the sampling. We perform this analysis on the identification experiments in Section 7.6.2, and use the heuristic suggested above in order to pick well performing ω values.

7.6 Experiments

In this section, we first perform experiments to demonstrate how the optimal value of ω, which is directly influenced by the sampling rate in the input signal, influences the generalization error of a sinusoidal network, following the discussion in Section 7.5. After that, we finally demonstrate that sinusoidal networks with properly tuned ω values outperform traditional neural networks in classic physics-informed learning tasks.

7.6.1 Evaluating generalization

Employing the simple sinusoidal network, we now perform experiments the ability of the fitted models to generalize to points outside the training set. For this purpose, in all experiments in this section, we segment the input signal into training and test sets using a

Table 7.1: Generalization results and the respective tuned ω value. Generalization values are mean squared error (MSE). We can observe the best performing ω for generalization with the simple sinusoidal network is half the ω used previously for fitting the full signal due to the fact that this task used half the sample points from previously.

Experiment	SIREN	SSN	ω	Fitting ω
Image	$2.76 \cdot 10^{-4}$	$\mathbf{1.16 \cdot 10^{-4}}$	16	32
Audio (Bach)	$4.55 \cdot 10^{-6}$	$\mathbf{3.87 \cdot 10^{-6}}$	8,000	15,000
Audio (counting)	$1.37 \cdot 10^{-4}$	$\mathbf{5.97 \cdot 10^{-5}}$	16,000	32,000
Video (cat)	$3.40 \cdot 10^{-3}$	$\mathbf{1.76 \cdot 10^{-3}}$	$\begin{bmatrix}4 & 8 & 8\end{bmatrix}$	8
Video (bikes)	$2.74 \cdot 10^{-3}$	$\mathbf{8.79 \cdot 10^{-4}}$	$\begin{bmatrix}4 & 4 & 8\end{bmatrix}$	8

checkerboard pattern – along all axis-aligned directions, points alternate between belonging to train and test set.

We perform audio, image and video fitting experiments, identical to the ones from Chapter 6, except for the different train/test split. When performing these fitting experiments, we search for the best performing ω value for generalization (defined as performance on the held-out points). We report the best values on Table 7.1. We observe that, as expected from the discussion in Section 7.5, the best performing ω values are half the minimum best performing value found in the fitting experiments from Chapter 6. This is confirms our expectation, since we are training on half the number of samples, and thus the bandwidth of our signal should also halve.

Trying to use a higher ω leads to overfitting, and poor generalization outside the training points. This is demonstrated in Figure 7.13, in which we can see that choosing an appropriate ω value from the heuristics described previously leads to a good fit and interpolation. Conversely, setting ω too high leads to interpolation artifacts, due to the learning of spurious high-frequency components.

Notice that for the 3D video signals, which have different dimensions along each axis, we employ a multi-dimensional ω. We scale each dimension of ω proportional to the size of the input signal along the corresponding axis, while picking a minimum value that is half the ω used for fitting.

7.6.2 Solving differential equations

Finally, we apply all the findings from the theoretical and empirical analysis so far to differential equation problems, using the physics-informed approach. We take the Navier-Stokes and Burgers identification problems and the Schrödinger inference problem from Raissi et al. [2019a], and the Helmholtz problem from Sitzmann et al. [2020], and compare the performance of the simple sinusoidal network to the standard tanh network that is commonly used for these tasks. Results are presented in Table 7.2.

(a) Ground truth image (b) Reconstruction with $\omega = 32$ (c) Reconstruction with $\omega = 128$

Figure 7.13: Examples of generalization from half the points in the original image using sinusoidal networks with different values of ω. Even though both networks achieve equivalent training loss, the rightmost one, with ω higher than what would be suggested from the Nyquist frequency of the input signal, overfits the data, causing high-frequency noise artifacts in the reconstruction (*e.g.*, notice the sky).

Table 7.2: Comparison of the sinusoidal network against MLP with hyperbolic tangent non-linearity on PINN experiments from Raissi et al. [2019a] and against SIREN for the Helmholtz experiment from Sitzmann et al. [2020]. Values are percent error relative to ground truth value for each parameter for identification problems and mean squared error (MSE) for inference problems.

Experiment	Baseline	SSN	ω
Burgers (Identification)	$[0.0521\%, 0.4522\%]$	$[\mathbf{0.0071\%}, \mathbf{0.0507\%}]$	10
Navier-Stokes (Identification)	$[0.0046\%, 2.093\%]$	$[\mathbf{0.0038\%}, \mathbf{1.782\%}]$	$\begin{bmatrix}0.6 & 0.3 & 1.2\end{bmatrix}$
Schrödinger (Inference)	$1.04 \cdot 10^{-3}$	$\mathbf{4.30 \cdot 10^{-4}}$	4
Helmholtz (Inference)	$5.97 \cdot 10^{-2}$	$\mathbf{5.94 \cdot 10^{-2}}$	16

Burgers equation (Identification)

This experiment reproduces the Burgers equation identification experiment from the appendix of Raissi et al. [2019a]. In this experiment, we are trying to identify, given a known solution, the parameters λ_1 and λ_2 of a 1D Burgers equation,

$$u_t + \lambda_1 u u_x - \lambda_2 u_{xx} = 0. \tag{7.8}$$

The ground truth value of the parameters are $\lambda_1 = 1.0$ and $\lambda_2 = 0.01/\pi$.

In order to find a good value for ω, we follow the method described in Section 7.5.3. We perform a procedure identical to the one in Section 7.4, in which we measure the loss for DFT reconstructions of the solution with truncated frequencies, for all possible cutoff values. From the results, presented in Figure 7.14, we observe that the solution does not have high bandwidth, with most of the loss being minimized at a cutoff frequency 70. Note

that the sampling performed to generate the training data (which will constitute the *de facto* input signal), described below in the training details, has a Nyquist frequency higher than this value, and is thus not limiting the underlying signal. This suggests an ω value of approximately 9.

Indeed, we observe that $\omega = 10$ gives the best identification of the desired parameters, with errors of 0.0071% and 0.0507% for λ_1 and λ_2 respectively, against errors of 0.0521% and 0.4522% of the baseline. This value of ω also achieve the lowest reconstruction loss against the known solution, with an MSE of $8.034 \cdot 10^{-4}$, which can further help identify the best performing ω value from the training data. Figure 7.15 shows the reconstructed solution using the identified parameters, together with the position of the sampled data points used for training.

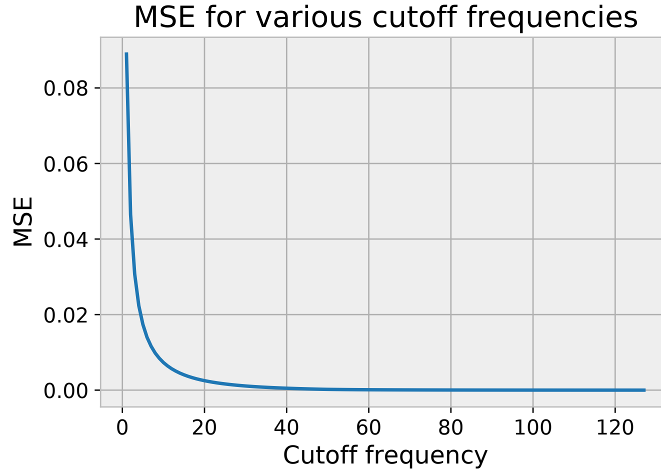


Figure 7.14: Reconstruction loss for different cutoff frequencies for a low-pass filter applied to the solution of the Burgers equation.

Training details. We follow the same training procedures as in Raissi et al. [2019a]. The training set is created by randomly sampling $2,000$ points from the available exact solution grid (shown in Figure 7.15). The neural networks used are 9-layer MLPs with 20 neurons per hidden layer. The network structure is the same for both the tanh and sinusoidal networks. As in the original work, the network is trained by using L-BFGS to minimize a mean square error loss composed of the sum of an MSE loss over the data points and a physics-informed MSE loss derived from Equation 7.8.

Navier-Stokes (Identification)

This experiment reproduces the Navier-Stokes identification experiment from Raissi et al. [2019a]. In this experiment, we are trying to identify, given known velocity fields u and v,

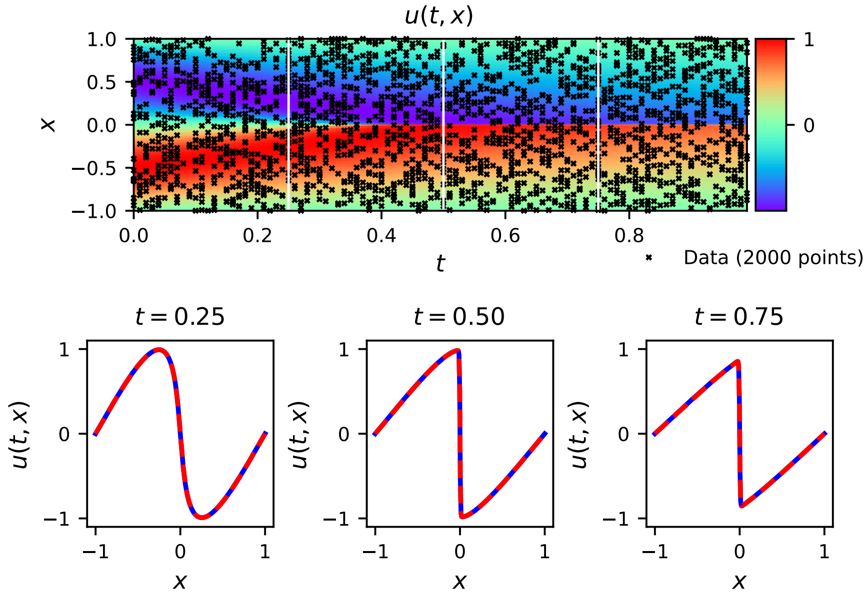


Figure 7.15: Reconstructed solution of the Burgers equation using the identified parameters with the sinusoidal network, together with the position of the sampled data points used for training.

the parameters λ_1, λ_2 and the pressure field p of the Navier-Stokes equations given by

$$u_t + \lambda_1(uu_x + vu_y) = -p_x + \lambda_2(u_{xx} + u_{yy}) \tag{7.9}$$

$$v_t + \lambda_1(uv_x + vv_y) = -p_y + \lambda_2(v_{xx} + v_{yy}). \tag{7.10}$$

The ground truth value of the parameters are $\lambda_1 = 1.0$ and $\lambda_2 = 0.01$.

Unlike the 1D Burgers case, in this case the amount of points sampled for the training set ($N = 5,000$, shown in Figure 7.17) is not high compared to the size of the full solution volume, and is thus the limiting factor for the bandwidth of the input signal. (Compare the approximate maximum frequency that can be inferred from Figure 7.16 to the one found for the sampling in the derivation below.)

Given the random sampling of points from the full solution, the generalized sampling theorem applies. Given the original solution dimensions of $100 \times 50 \times 200$, and the $5,000$ randomly sampled points, the average sampling rate per dimension is approximately 8.5, corresponding to a Nyquist frequency of approximately 4.25.

Furthermore, given the multi-dimensional nature of this problem, with both spatial and temporal axes, we employ an independent scaling to ω for each dimension. The analysis above suggests and average ω in the range $0.5 - 1$, with the dimensions of the problem suggesting scaling factors of $\begin{bmatrix} 0.5 & 1 & 2 \end{bmatrix}^T$.

Indeed, we observe that $\Omega = \begin{bmatrix} 0.3 & 0.6 & 1.2 \end{bmatrix}^T$ gives the best results. With with errors of 0.0038% and 1.782% for λ_1 and λ_2 respectively, against errors of 0.0046% and 2.093% of the baseline. Figure 7.18 shows the identified pressure field. Given the nature of the problem, this field can only be identified up to a constant.

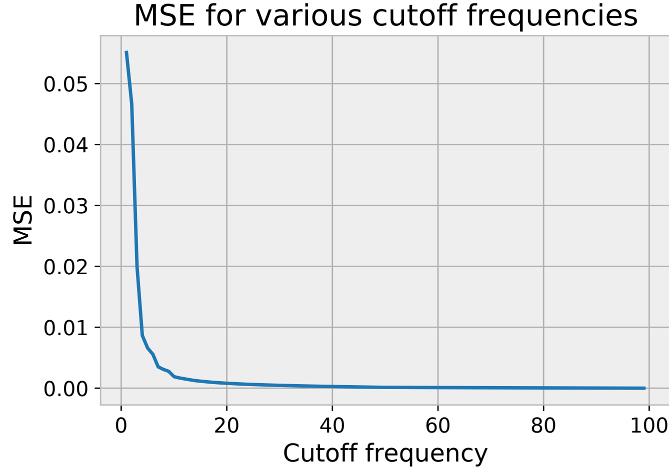


Figure 7.16: Reconstruction loss for different cutoff frequencies for a low-pass filter applied to the solution of the Navier-Stokes equations.

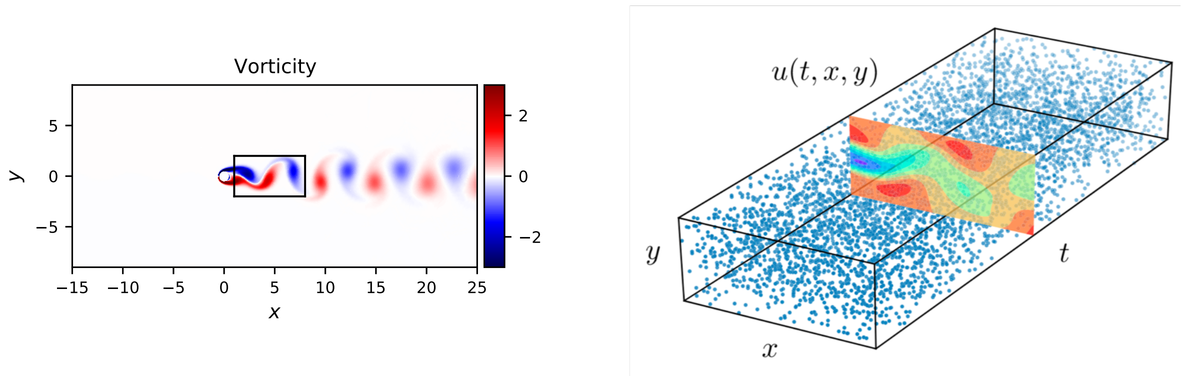


Figure 7.17: *Left:* One timestep of the ground truth Navier-Stokes solution. The black rectangle indicates the domain region used for the task. *Right:* The sampling of data points for the training set. Figures generated with code from Raissi et al. [2019a].

Training details. We follow the same training procedures as in Raissi et al. [2019a]. The training set is created by randomly sampling $5,000$ points from the available exact solution grid (one timestep is shown in Figure 7.17). The neural networks used are 9-layer MLPs with 20 neurons per hidden layer. The network structure is the same for both the tanh and sinusoidal networks. As in the original work, the network is trained by using the Adam optimizer to minimize a mean square error loss composed of the sum of an MSE loss over the data points and a physics-informed MSE loss derived from Equation 7.9.

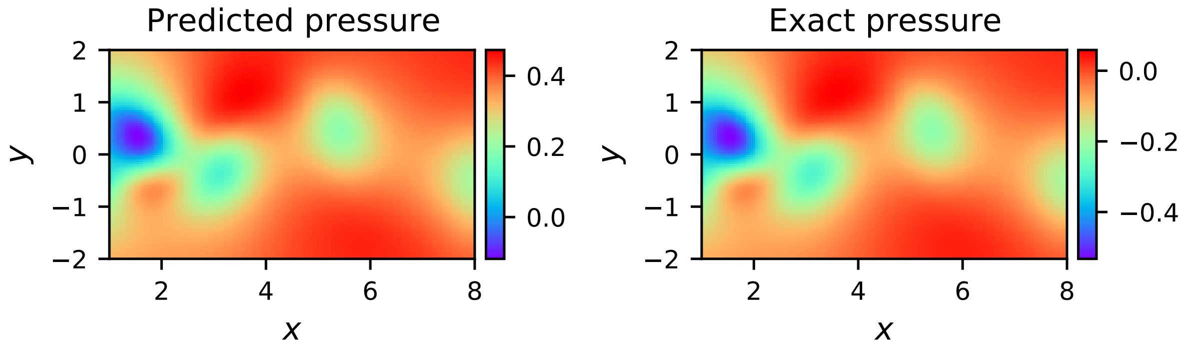


Figure 7.18: Identified pressure field for the Navier-Stokes equations using the sinusoidal network. Notice that the identification is only possible up to a constant.

Schrödinger (Inference)

This experiment reproduces the Schrödinger equation experiment from Raissi et al. [2019a]. In this experiment, we are trying to find the solution to the Schrödinger equation, given by

$$ih_t + 0.5h_{xx} + |h|^2 h = 0 \tag{7.11}$$

Since in this case we have a forward problem, we do not have any prior information to base our choice of ω on, besides a maximum limit given by the Nyquist frequency for the sampling generating the training data. We thus follow usual machine learning procedures and experiment with a number of small ω values, based on the previous experiments.

We find that $\omega = 4$ gives the best results, with a solution MSE of $4.30 \cdot 10^{-4}$, against an MSE of $1.04 \cdot 10^{-3}$ for the baseline. Figure 7.19 shows the solution from the sinusoidal network, together with the position of the sampled data points used for training.

Training details. We follow the same training procedures as in Raissi et al. [2019a]. The training set is created by randomly sampling 20,000 points from the domain ($x \in [-5, 5]$, $t \in [0, \pi/2]$) for evaluation for the physics-informed loss. Additionally, 50 points are sampled from each of the boundary and initial conditions for direct data supervision. The neural networks used are 5-layer MLPs with 100 neurons per hidden layer. The network structure is the same for both the tanh and sinusoidal networks. As in the original work, the network is trained first using the Adam optimizer by 50,000 steps and then by using L-BFGS until convergence. The loss is composed of the sum of an MSE loss over the data points and a physics-informed MSE loss derived from Equation 7.11.

Helmholtz equation

Details for the Helmholtz equation experiment are provided in Section 6.3.

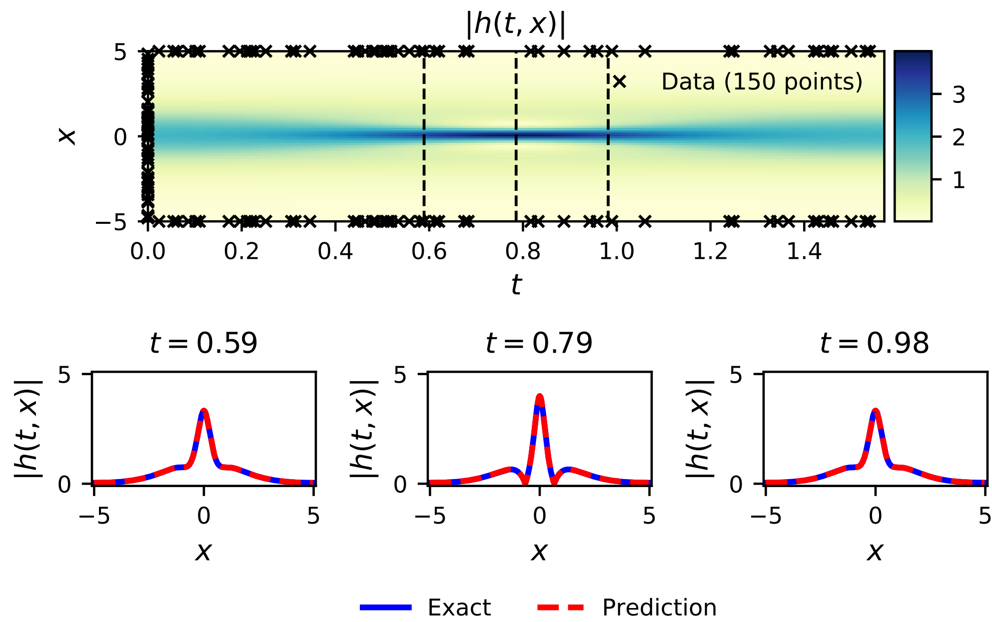


Figure 7.19: Solution to the Schrodinger equation with the sinusoidal network, together with the position of the sampled data points used for training.

Part III

Conclusion

Chapter 8

Conclusion

In this thesis, we have presented different approaches for combining prior knowledge of physics with deep learning models. As deep learning approaches have become successful in many different domains over the past decade, the intersection of machine learning and physics has also become a popular domain of research interest. In many domains in which it is applied, deep learning is presented as a disruptor that can displace traditional approaches by outperforming them using vasts amount of data. In part due to the limitations on the difficulty of acquiring data in physical domains, and in part due to the fact that they have been studied for far longer, traditional methods still have a significant advantage. In this work, we proposed instead methods that try and leverage the best of both worlds, combining the strengths of traditional methods and deep learning approaches.

In Part I we explained a particular approach to embedding a physics model into a deep learning architecture. We start by taking the view of neural network layers as general differentiable functions. We then replace one of those functions by a fully-capable, differentiable physics model of the relevant underlying dynamics. In Chapter 3, this amounts to developing a complete rigid body dynamics engine using a differentiable LCP solver. We are able to show that employing such an engine within a deep learning model can improve learning efficiency, by allowing the model to focus the learning in other aspects of the tasks, without having to learn the dynamics from scratch. This type of approach can have benefits not only limited to the domain of physics, but more broadly also to any task in which an agent interacts with the physical world.

In Chapter 4, we extended this approach naturally from rigid bodies to the domain of fluids. Here we employed an industrial grade CFD solver as a layer in a graph neural network model. This gives us twice the opportunity to guide the neural network, once through the physics model embedded as a layer and twice through the inductive biases provided by the mesh structure derived for the problem. We demonstrated in this study that providing such rich information to the neural network allowed it to be robust to changes from its training distribution, with the model being capable of generalizing to previously unseen behavior. This type of approach can be extremely valuable in fields such as aerodynamic design and prototyping, in which it is often desirable to have a fast and approximate estimate of certain aerodynamic properties, such as drag or lift.

In Part II, we presented methods for addressing and improving known shortcomings

in physics-informed learning. In Chapter 5, we address the issue of the high cost of re-training a physics-informed neural network when solving parameterized differential equations. By employing hypernetworks that learn the space of physics-informed neural network parameterizations for a given differential equation family, we are able to fit a range of parameters and perform faster inference for new parameterizations. This type of approach can be important for classes of problems in which it is required to repeatedly solve the same set of differential equations with slightly different parameterizations (including different initial or boundary conditions).

In Chapters 6 and 7, we addressed the issue of spectral bias, in which physics-informed neural networks have difficulty learning functions containing high-frequency components. In Chapter 6, we propose a simplified version of neural networks with sinusoidal activations and demonstrate they have performance similar to previously proposed analogous methods.Then, in Chapter 7, we perform a theoretical and empirical analysis of these simplified sinusoidal networks. Neural tangent kernel analysis suggests these networks behave similarly to low-pass filter kernels, with empirical analysis confirming these findings. Using these insights, we demonstrate how these networks can be properly tuned to match the appropriate spectrum in the learning signal. This allowed us to develop sinusoidal networks that outperform regular networks in physics-informed learning tasks. As sinusoidal networks become popular across a diverse set of problems, not only limited to physics, it is important to understand their behavior and how to tune their bandwidth to each given task.

Appendix A

Building a Differentiable Rigid Body Dynamics Engine

A.1 Physics Engine

In this section, we present a description of the structure of the physics engine. The formulation of the physics engine described here follows closely the one presented by Cline [2002], with some simplifications applied due to the engine presented here being two-dimensional. The description presented here is brief and intended only to be a sufficient guide to reproducing the work in the paper. For a more detailed introduction to physics engines, including comparisons to other architectural choices, see Garstenauer and Kurka [2006].

A.1.1 Step Overview

A physics simulation proceeds over time by iteratively taking small steps of size dt. In this section, we describe conceptually the sequence of sub-steps that compose a step in the simulation. In the following sections, we describe in greater depth each of these parts.

1. At the beginning of time step t we have the bodies at positions p_t with velocities v_t, as defined in Section A.1.3 (Equation A.3). Importantly, at this point (the beginning of the step), we assume current contacts are known and that constraints are satisfied (*i.e.*, there are no interpenetrations).
2. External forces acting on bodies are added up to form the force vector f_t, as defined in Section A.1.3 (Equation A.4).
3. Constraint matrices for the current step are formed, as defined in Section A.1.5.
4. We solve the dynamics LCP defined in Section A.1.6 to get the velocities v_{t+dt}.
5. Numerical integration is used with the velocities v_{t+dt} to get the positions p_{t+dt}. For example, simple explicit Euler integration gives $p_{t+dt} = p_t + dt \cdot v_{t+dt}$. The new positions p_{t+dt} have contacts detected and checked for interpenetrations (details in Section A.1.4). If interpenetrations do occur, we divide dt in half and repeat the

numerical integration from p_t to p_{t+dt} with the new dt. We repeat this process until a position free of interpenetrations is found. Since we know there are no interpenetrations at the beginning of the step (by the assumption in sub-step 1), we know there is $dt > 0$ for which there are no interpenetrations.

6. If post-stabilization is being employed, then constraint matrices are re-calculated from the new contacts at positions p_{t+dt}, and the post-stabilization correction to the positions is applied. Importantly, the post-stabilization update does not violate constraints, thus we still end the step with no interpenetrations.

A.1.2 Bodies

The basic unit in the engine are the rigid bodies. Bodies possess mass, position and velocity, and forces act on them. The position and velocity of a body are composed of three components: an angular (a) components, and two linear (x and y) components,

$$p_{body} = \begin{bmatrix} p_a \\ p_x \\ p_y \end{bmatrix} \qquad v_{body} = \begin{bmatrix} v_a \\ v_x \\ v_y \end{bmatrix}. \tag{A.1}$$

The mass of a body, m, defines the mass-inertia matrix of a body, $\mathcal{M}_{body}$, which is given by

$$\mathcal{M}_{body} = \begin{bmatrix} \mathcal{I} & 0 & 0 \\ 0 & m & 0 \\ 0 & 0 & m \end{bmatrix},$$

where $\mathcal{I}$ is the moment of inertia for the body, which is a scalar in 2D. The moment of inertia is a function of the mass and the shape of a body. For example, for a circle, we have $\mathcal{I} = \frac{1}{2}mr^2$, where r is the radius.[1]

Moreover, bodies also possess dimensionless scalar parameters that define their behavior when in contact with other bodies, namely the collision restitution coefficient and the friction coefficient. The restitution coefficient k specifies the elasticity of the collision, with $k = 1$ specifying a perfectly elastic collision, and $k = 0$ specifying a perfectly inelastic collision. The friction here is defined by a single coefficient μ, with no distinction for static and dynamic friction. When two bodies are in contact, the frictional force opposes a body's movement of sliding against the other. The friction coefficient coefficient defines the maximal frictional force as a proportion to the normal force between the bodies, that is

$$f_{fric} \leq \mu f_{normal}.$$

Finally, each body has a set of external forces that act on it. External forces are represented as acting on the center of mass of the body, and they are represented as a vector with three components: a torque (τ) component, and two linear (f_x and f_y) components

$$f_{external} = \begin{bmatrix} \tau \\ f_x \\ f_y \end{bmatrix}. \tag{A.2}$$

[1] Other examples available at `https://en.wikipedia.org/wiki/List_of_moments_of_inertia`

A.1.3 Global Parameters

To simplify the simulation equations specified below, parameters for all bodies are grouped into global structures. Assuming there are n bodies in a simulation, ordered consistently from 1 to n, the global position and velocity vectors are given by

$$p = \begin{bmatrix} p_1 \\ \vdots \\ p_n \end{bmatrix} \qquad v = \begin{bmatrix} v_1 \\ \vdots \\ v_n \end{bmatrix}. \tag{A.3}$$

where p_i and v_i are three dimensional position and velocity vectors (as defined in Equation A.1) for each body $i \in \{1, \ldots, n\}$. Forces acting on bodies are also concatenate

Similarly, the mass-inertia matrices for all n bodies are concatenated into a large global matrix $\mathcal{M}$, given by

$$\mathcal{M} = \begin{bmatrix} \mathcal{M}_1 & & 0 \\ & \ddots & \\ 0 & & \mathcal{M}_n \end{bmatrix}.$$

A global external force vector is constructed by summing all external forces acting on each body, and concatenating them into a single vector. We thus have

$$f = \begin{bmatrix} f_1 \\ \vdots \\ f_n \end{bmatrix}, \tag{A.4}$$

where each f_i is the sum of all external forces acting on body i.

A.1.4 Contact Detection

Let us define the distance between two bodies as the minimum length between two points, one in surface of each body. Two bodies are then considered to be in contact if the distance between them is less than some parameter $\epsilon > 0$. In other words, for simulation purposes, two bodies are in contact if they are either interpenetrating (*i.e.*, distance smaller than zero), or "touching" (*i.e.*, distance between 0 and ϵ).

The purpose of detecting contacts is to be able to enforce non-interpenetration constraints. From the definitions of the constraint matrices in Section A.1.5, we can see that for each contact between two bodies, our formulation requires a normal contact vector and a contact point in each body. The normal vector defines the direction in which the contact force will be applied, while the contact points define the points in which such force will be applied in each body.

The process of detecting contacts is divided into two phases: a "broadphase" that cheaply generates candidate contacts by finding bodies that are in each others vicinity, and a "narrowphase" that analyses candidate contacts carefully to determine if they are truly contacts and to generate the contact information. A naive broadphase approach is to simply list all possible pairs of objects, followed then by a narrowphase will have to verify

every possible contact. More efficient broadphase algorithms exist, but these will not be discussed here for the sake of brevity. Refer to Bergen [2004] for a more detailed exposition.

In the narrowphase, as described above, we want to not only verify that a contact between two bodies is present, but also to generate the required information related to that contact. We will rely on two algorithms for this purpose: the Gilbert–Johnson–Keerthi (GJK) and the Separating Axis Theorem (SAT). For a detailed exposition of these algorithms, refer to Catto [2010] and Gregorius [2013]. Suffice it to say here that the GJK algorithm can provide the closest points between two disjoint convex shapes and that the SAT algorithm can provide the axis of minimum penetration between two interpenetrating convex shapes.

The specifics of how contact detection is handled depends on the shapes involved. For the sake of brevity, we will cover here two examples: (1) circle against circle and (2) circle against convex hull. For a detailed exposition containing more collision types, including convex hulls against convex hulls, please refer to Gregorius [2015].

Circle against circle

For two circles, checking for contacts is simple. The distance between the two bodies, with positions p_1 and p_2 and radii r_1 and r_2, is given by

$$d = \|p_1 - p_2\| - r_1 - r_2.$$

We thus have a contact if $d < \epsilon$ and an interpenetration if $d < 0$. The normal vector is simply given by

$$n = \frac{p_1 - p_2}{\|p_1 - p_2\|}.$$

Finally, the contact points (given in each body's reference frame) are

$$c_1 = -n \cdot r_1 \quad \text{and} \quad c_2 = n \cdot r_2.$$

Circle against convex hull

In this case, we start by applying the GJK algorithm on the convex hull and the center of the circle. There are two possible cases. First, if the circle's center is outsde the hull, GJK will return the point in the convex hull closest to the circle's center. From this, we have that the distance between the two bodies is

$$d = \|p_c - p_{GJK}\| - r,$$

where p_c and r are the circle's center and radius, and p_{GJK} is the nearest point to p_c in the hull. As before, we have a contact if $d < \epsilon$ and an interpenetration if $d < 0$. The normal is then given by

$$n = \frac{p_c - p_{GJK}}{\|p_c - p_{GJK}\|},$$

and the contact points are

$$c_c = -n \cdot r \quad \text{and} \quad c_h = p_{GJK}.$$

The second case happens when the circle's center is inside the hull. In this case, we know we have an interpenetration, but GJK does not provide us with enough information to generate the contact. We thus employ the SAT algorithm to find the axis of minimum penetration. This is done by running the SAT on the axes defined by the normal to each the face of the hull (*i.e.*, the vector perpendicular to the face that points out of the hull). Once we find the face with the smallest distance to p_c, the collision normal n is the normal to that face, normalized to have length 1. In this case we know there is a penetration, thus the distance $d < 0$ between the two bodies is given by the distance from the circle's center to the closest face of the hull minus the radius of the circle, or equivalently

$$d = (p_c^{(h)} - p_{vert}) \cdot n - r,$$

where $(p_c - p_{vert}) \cdot n$ takes the vector from one of the vertices of the closest face in the hull (p_{vert}) to the center of the circle in the hull's reference frame $(p_c^{(h)})$, and projects it onto the normal (n). Finally, the contact points are given by the closest point to the circle in the hull's face, given in each body's reference frame

$$c_h = p_c^{(h)} - n \cdot (d + r) \quad \text{and} \quad c_c = c_h + p_c - p_c,$$

where p_h is the hull's position.

A.1.5 Constraints

In this section we describe in detail the equations that constrain the dynamics of the bodies. These constraints are divided into three categories: equality, contact and friction constraints.

Equality constraints

Equality constraints take the form $g(v) = 0$, where g is some function of the velocities. These constraints can be used to implement, for example, joints of many kinds.

In general, for two bodies (a and b), equality constraints are defined by two Jacobian matrices ($\mathcal{J}_e^{(body)}$), one for each body, such that

$$\mathcal{J}_e^{(a)} v^{(a)} + \mathcal{J}_e^{(b)} v^{(b)} = 0. \tag{A.5}$$

A hinge joint, for example, which in two dimensions gives the bodies only one degree of freedom to rotate about their connection point, would be defined by the following two Jacobians

$$\mathcal{J}_e^{(a)} = \begin{bmatrix} -r_{a_y} & 1 & 0 \\ r_{a_x} & 0 & 1 \end{bmatrix} \quad \text{and} \quad \mathcal{J}_e^{(b)} = \begin{bmatrix} r_{b_y} & -1 & 0 \\ -r_{b_x} & 0 & -1 \end{bmatrix},$$

where r_a and r_b are the vectors pointing to the connection point from the center of each body. Substituting these two matrices into Equation A.5 we can see that we get an equation that constraints the translation velocities of the two bodies at the connection point to be equal.

Finally, to more easily apply all constraint simultaneously in concert with the global parameters defined above, we define a global constraint Jacobian $\mathcal{J}_e$. For n bodies and m constraints, $\mathcal{J}_e$ is formed as a block matrix such that we have

$$\mathcal{J}_e v = \begin{bmatrix} \mathcal{J}_{11} & \cdots & \mathcal{J}_{1n} \\ \vdots & \ddots & \vdots \\ \mathcal{J}_{m1} & \cdots & \mathcal{J}_{mn} \end{bmatrix} \begin{bmatrix} v_1 \\ \vdots \\ v_n \end{bmatrix} = 0, \tag{A.6}$$

where in each block-row i of $\mathcal{J}_e$ all blocks $\mathcal{J}_{ij}$ are zero matrices, except for the two blocks $\mathcal{J}_{ia}$ and $\mathcal{J}_{ib}$ corresponding to the two bodies we want to constraint, which are then given by Equation A.5.

Contact constraints

Contact constraints are inequality constraints and thus take the form $g(v) \geq 0$. These constraints enforce that rigid bodies do not interpenetrate.

In general, for two bodies (a and b), contact constraints are defined by two Jacobian matrices ($\mathcal{J}_c^{(body)}$), one for each body, such that

$$\mathcal{J}_c^{(a)} v_{t+dt}^{(a)} + \mathcal{J}_c^{(b)} v_{t+dt}^{(b)} + c_{ab} \geq 0, \tag{A.7}$$

where the term c_{ab} depends on the velocities before the contact and on the combined restitution parameter for a and b, k_{ab},

$$c_{ab} = k_{ab} \left[\mathcal{J}_c^{(a)} v_t^{(a)} + \mathcal{J}_c^{(b)} v_t^{(b)} \right]. \tag{A.8}$$

The combined restitution parameter is usually defined as a simple function of the restitution parameter of each body, for example $k_{ab} = \frac{1}{2}(k_a + k_b)$.

At each time step, contact constraints as defined in Equation A.7 are constructed for each pair of bodies in contact. Using the normal vector n and the contact points p_a and p_b provided by the contact detection algorithms (see Section A.1.4), the Jacobians are 1×3 matrices defined as[2]

$$\mathcal{J}_c^{(a)} = \begin{bmatrix} (p_a \times n) & n^T \end{bmatrix} \quad \text{and} \quad \mathcal{J}_c^{(b)} = \begin{bmatrix} (p_b \times n) & n^T \end{bmatrix}, \tag{A.9}$$

As before, we define a global constraint Jacobian $\mathcal{J}_c$. For n bodies and m constraints, $\mathcal{J}_c$ is formed as a block matrix such that we have

$$\mathcal{J}_c v_{t+dt} = \begin{bmatrix} \mathcal{J}_{11} & \cdots & \mathcal{J}_{1n} \\ \vdots & \ddots & \vdots \\ \mathcal{J}_{m1} & \cdots & \mathcal{J}_{mn} \end{bmatrix} \begin{bmatrix} v_1 \\ \vdots \\ v_n \end{bmatrix} \geq -c, \tag{A.10}$$

where in each block-row i of $\mathcal{J}_c$ all blocks $\mathcal{J}_{ij}$ are zero matrices, except for the two blocks $\mathcal{J}_{ia}$ and $\mathcal{J}_{ib}$ corresponding to the two bodies in contact, which are then given by Equation A.9. The term c is then given by

$$c = \mathrm{diag}(k) \mathcal{J}_c v_t,$$

with $k = [k_1, \ldots, k_m]^T$, where $k_i = k_{ab}$ for the two contacting bodies in contact i.

[2]We define the two dimensional cross product as the scalar $x \times y = x_1 y_2 - x_2 y_1$.

Friction constraints

Friction constraints are also inequality constraints and thus take the form $g(v) \geq 0$. While contact forces act on the normal direction, friction constraints create forces that act tangentially to the plane of contact of two bodies.

For simplicity, in this section we will for now simply describe the structure of the friction Jacobian $\mathcal{J}_f$, noting its similarity to the contact Jacobian $\mathcal{J}_c$. In Section A.1.6, we will describe the structure of the inequalities and the complementarity constraints that cause the frictional forces to behave as expected.

As mentioned above, frictional constraints act on contacts, but in the tangential directions instead of the normal direction. In two dimensions there are two tangential directions to a contact, the two orthogonal directions to the contact normal vector. Intuitively, we can imagine that the friction Jacobians will have a structure analogous to the contact Jacobians in Equation A.9, with the normal vector substituted by the tangent vectors. Since there are two tangent directions, we will have two constraints for each contact. Let us call d the left orthogonal vector to the normal contact vector n, and p_a and p_b the contact points, as provided by the contact detection algorithms (see Section A.1.4). We then have the friction Jacobians for a contact between bodies a and b

$$\mathcal{J}_f^{(a)} = \begin{bmatrix} (p_a \times d) & d^T \\ (p_a \times -d) & -d^T \end{bmatrix} \quad \text{and} \quad \mathcal{J}_f^{(b)} = \begin{bmatrix} (p_b \times d) & d^T \\ (p_b \times -d) & -d^T \end{bmatrix}. \tag{A.11}$$

As before, we define a global constraint Jacobian $\mathcal{J}_f$. For n bodies and m constraints, $\mathcal{J}_c$ is formed as a block matrix given by

$$\mathcal{J}_f = \begin{bmatrix} \mathcal{J}_{11} & \cdots & \mathcal{J}_{1n} \\ \vdots & \ddots & \vdots \\ \mathcal{J}_{k1} & \cdots & \mathcal{J}_{kn} \end{bmatrix},$$

where in each block-row i of $\mathcal{J}_f$ all blocks $\mathcal{J}_{ij}$ are zero matrices, except for the two blocks $\mathcal{J}_{ia}$ and $\mathcal{J}_{ib}$ corresponding to the two bodies in contact, which are then given by Equation A.11.

Two other matrices will be important when dealing with friction constraints, E and μ. We will define them here for later reference. If there are m contacts for a given time step, we have

$$E = \begin{bmatrix} e_1 & \cdots & e_n \end{bmatrix} \quad \text{and} \quad \mu = \begin{bmatrix} \mu_1 & & \\ & \ddots & \\ & & \mu_m \end{bmatrix}.$$

Here, $\mu_i \in \mathbb{R}$ is the combined friction coefficient two bodies involved in contact i, which can be defined as the average of each body's friction coefficient, for example. Moreover, $e_i \in \mathbb{R}^{2m}$ is a column vector of zeroes except for the two entries $2i-1$ and $2i$ (*e.g.*, $e_2 = [0,0,1,1,\ldots,0]^T$ and $e_m = [0,\ldots,0,1,1]^T$).

A.1.6 Dynamics LCP

Let us call $\dot{v}$ the acceleration vector. From Newtonian dynamics, it generally holds that

$$\mathcal{M}\dot{v} = f^{(c)} + f,$$

where $f^{(c)}$ are constraint forces inherent to the dynamics, and f are external forces applied to the bodies. These two are here assumed to comprise the totality of forces acting on bodies. However, formulating the dynamics at the acceleration level can lead to systems with no solutions in the presence of friction [Anitescu and Potra, 1997]. Fortunately, by approximating the acceleration with a discrete step

$$\dot{v}_{t+dt} \approx \frac{v_{t+dt} - v_t}{dt},$$

we can rewrite the dynamics equation as

$$\mathcal{M}(v_{t+dt} - v_t) = dt f_t^{(c)} + dt f_t, \tag{A.12}$$

to get a velocity-based formulation, which is guaranteed to have a solution even with friction constraints. Since dt is a small time-step, $dt f_t^{(c)}$ can be seen as approximate constraint impulses. We omit the derivation here (see Garstenauer and Kurka [2006]), but these constraint impulses can be written as

$$dt f^{(c)} = \mathcal{J}\lambda, \tag{A.13}$$

where $\mathcal{J}$ is a constraint Jacobian such as the ones described in Section A.1.5, and λ is some vector of multipliers. By rearranging the terms in Equation A.12 and combining with Equation A.13 (broken down into the equality, contact and friction constraint matrices), we get the final dynamics equation

$$\mathcal{M}v_{t+dt} - \mathcal{J}_e\lambda_e - \mathcal{J}_c\lambda_c - \mathcal{J}_f\lambda_f = \mathcal{M}v_t + dt f_t. \tag{A.14}$$

Moreover, for a realistic rigid body simulation, we know that the impulses $\mathcal{J}_c\lambda_c$ can act to push bodies apart and avoid interpenetrations, but they cannot act to pull bodies together. Hence, we must have $\lambda_c \geq 0$. Additionally, if for each constraint i, $(\mathcal{J}_c v)_i + c_i = a_i$ for some $a_i > 0$ strictly greater than zero, then the bodies are moving apart and no separating forces should be applied, *i.e.* $(\lambda_c)_i = 0$. Conversely, if this condition is not satisfied, then a separating force is needed to counteract the penetration velocity, thus $(\lambda_c)_i = 0$. This gives rise to the following complementarity condition,

$$\lambda_c \geq 0, \quad a := \mathcal{J}_c v + c \geq 0 \quad \text{and} \quad a^T\lambda_c = 0. \tag{A.15}$$

The friction terms have similar complementarity constraints. However, due to the nature of Coulomb friction these constraints involve the contact terms (for example, to assert there is no frictional force when contact normal force is 0). We will omit the derivation here (see Cline [2002]), but these interactions give rise to the following constraints,

$$\zeta := \mu\lambda_c - E^T\lambda_f \geq 0, \quad \sigma := \mathcal{J}_f v + \gamma E \geq 0, \quad \sigma^T\lambda_f = 0 \quad \text{and} \quad \zeta^T\gamma = 0, \tag{A.16}$$

with $\lambda_f \geq 0$ and $\gamma \geq 0$.

Taking the constraints defined in Section A.1.5 (Equations A.6 and A.10), the dynamics equation (Equation A.14), and the complementarity conditions formulated above (Equations A.15 and A.16), the dynamics for a step in the simulation can be summarized as the following LCP

$$\begin{bmatrix} 0 \\ 0 \\ a \\ \sigma \\ \zeta \end{bmatrix} - \begin{bmatrix} \mathcal{M} & -\mathcal{J}_e & -\mathcal{J}_c & -\mathcal{J}_f & 0 \\ \mathcal{J}_e & 0 & 0 & 0 & 0 \\ \mathcal{J}_c & 0 & 0 & 0 & 0 \\ \mathcal{J}_f & 0 & 0 & 0 & E \\ 0 & 0 & \mu & -E^T & 0 \end{bmatrix} \begin{bmatrix} v_{t+dt} \\ \lambda_e \\ \lambda_c \\ \lambda_f \\ \gamma \end{bmatrix} = \begin{bmatrix} \mathcal{M}v_t + dt f_t \\ 0 \\ c \\ 0 \\ 0 \end{bmatrix}$$
$$\text{subject to} \begin{bmatrix} a \\ \sigma \\ \zeta \end{bmatrix} \geq 0, \begin{bmatrix} \lambda_c \\ \lambda_f \\ \gamma \end{bmatrix} \geq 0, \begin{bmatrix} a \\ \sigma \\ \zeta \end{bmatrix}^T \begin{bmatrix} \lambda_c \\ \lambda_f \\ \gamma \end{bmatrix} = 0. \tag{A.17}$$

Where the inequality constraints are written as equality constraints using the slack variables $[a, \sigma, \zeta]^T$ defined above, and $[v_{t+dt}, \lambda_e, \lambda_c, \lambda_f, \gamma]^T$ are the unknowns. From the solution, we obtain the velocities v_{t+dt} for the next step, which are used to update the positions of the bodies as described in Section A.1.1 (item 5).

A.2 Solution and Derivatives

A.2.1 Solution

The solution described here follows closely the method described in Mattingley and Boyd [2012], with small modifications for our LCP formulation above. The following is a small summary of that method, highlighting such differences.

In Equation 3.2, we formed the equivalent to the following system:

$$\begin{aligned} \mathcal{M}x + A^T y + G^T z + q &= 0 \\ Ax &= 0 \\ Gx + Fz + s &= m \\ s \geq 0,\ z \geq 0,\ s^T z &\geq 0. \end{aligned} \tag{A.18}$$

To solve such system, after an initialization step (described in Mattingley and Boyd [2012]), we iteratively minimize the residuals from the equations above over the variables x, s, z and y. At each iteration, if the stopping criteria (residual sizes and duality gap) are not met, we compute the affine scaling directions by solving the system

$$\begin{bmatrix} \mathcal{M} & 0 & G^T & A^T \\ 0 & D(z) & D(s) & 0 \\ G & I & F & 0 \\ A & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} \Delta x^{\text{aff}} \\ \Delta s^{\text{aff}} \\ \Delta z^{\text{aff}} \\ \Delta y^{\text{aff}} \end{bmatrix} = \begin{bmatrix} -(\mathcal{M}x + A^T y + G^T z + q) \\ -(D(s)z) \\ -(Gx + Fz + s - m) \\ -(Ax) \end{bmatrix}. \tag{A.19}$$

Then we compute the centering-plus-corrector directions

$$\begin{bmatrix} \mathcal{M} & 0 & G^T & A^T \\ 0 & D(z) & D(s) & 0 \\ G & I & F & 0 \\ A & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} \Delta x^{\text{cc}} \\ \Delta s^{\text{cc}} \\ \Delta z^{\text{cc}} \\ \Delta y^{\text{cc}} \end{bmatrix} = \begin{bmatrix} 0 \\ \sigma\mu\mathbf{1} - D(\Delta s^{\text{aff}})\Delta z^{\text{aff}} \\ 0 \\ 0 \end{bmatrix}. \tag{A.20}$$

where μ and σ is defined in [Mattingley and Boyd, 2012]. We then update the variables by applying the following combined updates

$$\begin{aligned} x &:= x + \alpha(\Delta x^{\text{aff}} + \Delta x^{\text{cc}}) \\ s &:= s + \alpha(\Delta s^{\text{aff}} + \Delta s^{\text{cc}}) \\ z &:= z + \alpha(\Delta z^{\text{aff}} + \Delta z^{\text{cc}}) \\ y &:= y + \alpha(\Delta y^{\text{aff}} + \Delta y^{\text{cc}}) \end{aligned} \tag{A.21}$$

according to the step-size α defined in Mattingley and Boyd [2012].

A.2.2 Derivatives

To obtain the derivatives, we use Equation A.18 in a slightly modified form, such that at a solution point we have

$$\begin{aligned} \mathcal{M}x^\star + A^T y^\star + G^T z^\star + q &= 0 \\ Ax^\star &= 0 \\ D(z^\star)(Gx^\star + Fz^\star - m) &= 0. \end{aligned}$$

We use matrix differential calculus [Magnus and Neudecker, 1988] to take the differentials of these equations:

$$\begin{aligned} \mathrm{d}\mathcal{M}x^\star + \mathcal{M}\mathrm{d}x + \mathrm{d}A^T y^\star + A^T\mathrm{d}y + \mathrm{d}G^T z^\star + G^T\mathrm{d}z + \mathrm{d}q &= 0 \\ \mathrm{d}Ax^\star + A\mathrm{d}x &= 0 \\ D(Gx^\star + Fz^\star - m)\mathrm{d}z + D(z^\star)(\mathrm{d}Gx^\star + G\mathrm{d}x + \mathrm{d}Fz^\star + F\mathrm{d}z - \mathrm{d}m) &= 0, \end{aligned} \tag{A.22}$$

which in matrix form is equivalent to

$$\begin{bmatrix} \mathcal{M} & G^T & A^T \\ D(z^\star)G & D(Gx^\star + Fz^\star - m) + F & 0 \\ A & 0 & 0 \end{bmatrix} \begin{bmatrix} \mathrm{d}x \\ \mathrm{d}z \\ \mathrm{d}y \end{bmatrix} = \begin{bmatrix} -\mathrm{d}\mathcal{M}x^\star - \mathrm{d}A^T y^\star - \mathrm{d}G^T z^\star - \mathrm{d}q \\ -D(z^\star)\mathrm{d}Gx^\star - D(z^\star)\mathrm{d}Fz^\star + D(z^\star)\mathrm{d}m \\ -\mathrm{d}Ax^\star \end{bmatrix}.$$

In this formulation, a given partial derivative, for example $\frac{\partial z^\star}{\partial q}$, can be found by substituting $\mathrm{d}q = I$, setting all other differential terms to zero, and solving for $\mathrm{d}z$. For the backpropagation algorithm, for a given backward pass vector with respect to the solution $x^\star$ to the LCP, say $\frac{\partial \ell}{\partial x^\star}$, we are interested in applying the chain rule to pass the derivatives further

backwards, for example to find $\frac{\partial \ell}{\partial q}$ by multiplying $\frac{\partial \ell}{\partial x^\star}\frac{\partial x^\star}{\partial q}$. To simplify this process, let us first define the vector

$$\begin{bmatrix} d_x \\ d_z \\ d_y \end{bmatrix} := \begin{bmatrix} \mathcal{M} & G^T & A^T \\ D(z^\star)G & D(Gx^\star + Fz^\star - m) + F & 0 \\ A & 0 & 0 \end{bmatrix}^{-T} \begin{bmatrix} \left(\frac{\partial \ell}{\partial x^\star}\right)^T \\ 0 \\ 0 \end{bmatrix}. \tag{A.23}$$

Then, we have that

$$\frac{\partial \ell}{\partial x^\star}\mathrm{d}x = \begin{bmatrix} d_x \\ d_z \\ d_y \end{bmatrix}^T \begin{bmatrix} -\mathrm{d}\mathcal{M}x^\star - \mathrm{d}A^T y^\star - \mathrm{d}G^T z^\star - \mathrm{d}q \\ -D(z^\star)\mathrm{d}Gx^\star - D(z^\star)\mathrm{d}Fz^\star + D(z^\star)\mathrm{d}m \\ -\mathrm{d}Ax^\star \end{bmatrix}. \tag{A.24}$$

Now, by applying properties from matrix differential calculus we can propagate back the derivatives via chain rule to obtain the derivatives of our given backwards pass vector ℓ with respect to the inputs. For example, to obtain $\frac{\partial \ell}{\partial q} = \frac{\partial \ell}{\partial x^\star}\frac{\partial x^\star}{\partial q}$, we can see that

$$\frac{\partial \ell}{\partial x^\star}\mathrm{d}x = \begin{bmatrix} d_x \\ d_z \\ d_y \end{bmatrix}^T \begin{bmatrix} -\mathrm{d}q \\ 0 \\ 0 \end{bmatrix} = -d_x^T \mathrm{d}q, \tag{A.25}$$

which implies $\frac{\partial \ell}{\partial x^\star}\frac{\partial x^\star}{\partial q} = -d_x$. The same procedure can be applied to the others quantities to arrive at the desired derivatives

$$\begin{aligned} \frac{\partial \ell}{\partial q} &= -d_x & \frac{\partial \ell}{\partial \mathcal{M}} &= -\frac{1}{2}(d_x x^T + x d_x^T) \\ \frac{\partial \ell}{\partial m} &= D(z^\star) d_z & \frac{\partial \ell}{\partial G} &= -D(z^\star)(d_z x^T + z d_x^T) \\ \frac{\partial \ell}{\partial A} &= -d_y x^T - y d_x^T & \frac{\partial \ell}{\partial F} &= -D(z^\star) d_z z^T. \end{aligned} \tag{A.26}$$

Bibliography

Yaser Afshar, Saakaar Bhatnagar, Shaowu Pan, Karthik Duraisamy, and Shailendra Kaushik. Prediction of Aerodynamic Flow Fields Using Convolutional Neural Networks. *Computational Mechanics*, 64(2), 2019. doi: 10.1007/s00466-019-01740-0. URL `http://arxiv.org/abs/1905.13166`. 1, 2.2.4, 4.1, 4.3.1

Rami Al-Rfou, Guillaume Alain, Amjad Almahairi, Christof Angermueller, Dzmitry Bahdanau, Nicolas Ballas, Frédéric Bastien, Justin Bayer, Anatoly Belikov, Alexander Belopolsky, Yoshua Bengio, Arnaud Bergeron, James Bergstra, Valentin Bisson, Josh Bleecher Snyder, Nicolas Bouchard, Nicolas Boulanger-Lewandowski, Xavier Bouthillier, Alexandre de Brébisson, Olivier Breuleux, Pierre-Luc Carrier, Kyunghyun Cho, Jan Chorowski, Paul Christiano, Tim Cooijmans, Marc-Alexandre Côté, Myriam Côté, Aaron Courville, Yann N. Dauphin, Olivier Delalleau, Julien Demouth, Guillaume Desjardins, Sander Dieleman, Laurent Dinh, Mélanie Ducoffe, Vincent Dumoulin, Samira Ebrahimi Kahou, Dumitru Erhan, Ziye Fan, Orhan Firat, Mathieu Germain, Xavier Glorot, Ian Goodfellow, Matt Graham, Caglar Gulcehre, Philippe Hamel, Iban Harlouchet, Jean-Philippe Heng, Balázs Hidasi, Sina Honari, Arjun Jain, Sébastien Jean, Kai Jia, Mikhail Korobov, Vivek Kulkarni, Alex Lamb, Pascal Lamblin, Eric Larsen, César Laurent, Sean Lee, Simon Lefrancois, Simon Lemieux, Nicholas Léonard, Zhouhan Lin, Jesse A. Livezey, Cory Lorenz, Jeremiah Lowin, Qianli Ma, Pierre-Antoine Manzagol, Olivier Mastropietro, Robert T. McGibbon, Roland Memisevic, Bart van Merriënboer, Vincent Michalski, Mehdi Mirza, Alberto Orlandi, Christopher Pal, Razvan Pascanu, Mohammad Pezeshki, Colin Raffel, Daniel Renshaw, Matthew Rocklin, Adriana Romero, Markus Roth, Peter Sadowski, John Salvatier, François Savard, Jan Schlüter, John Schulman, Gabriel Schwartz, Iulian Vlad Serban, Dmitriy Serdyuk, Samira Shabanian, Étienne Simon, Sigurd Spieckermann, S. Ramana Subramanyam, Jakub Sygnowski, Jérémie Tanguay, Gijs van Tulder, Joseph Turian, Sebastian Urban, Pascal Vincent, Francesco Visin, Harm de Vries, David Warde-Farley, Dustin J. Webb, Matthew Willson, Kelvin Xu, Lijun Xue, Li Yao, Saizheng Zhang, and Ying Zhang. Theano: A Python framework for fast computation of mathematical expressions. *arXiv e-prints*, abs/1605.02688, May 2016. URL `http://arxiv.org/abs/1605.02688`. 2.2.3

T. Albring, M. Sagebaum, and N. R. Gauger. Development of a consistent discrete adjoint solver in an evolving aerodynamic design framework. *AIAA Paper 2015-3240*, 2015. doi: 10.2514/6.2015-3240. 2.2.2

T. Albring, M. Sagebaum, and N.R. Gauger. Efficient aerodynamic design using the discrete

adjoint method in SU2. *AIAA Paper 2016-3518*, 2016. doi: 10.2514/6.2016-3518. 2.2.2

Ferran Alet, Adarsh K. Jeewajee, Maria Bauza, Alberto Rodriguez, Tomas Lozano-Perez, and Leslie Pack Kaelbling. Graph Element Networks: adaptive, structured computation and memory. 2019. URL `http://arxiv.org/abs/1904.09019`. 2.2.5

Brandon Amos and J. Zico Kolter. OptNet: Differentiable Optimization as a Layer in Neural Networks. 2017. URL `http://arxiv.org/abs/1703.00443`. 2.2.2, 3.2.2, 3.2.3

Mihai Anitescu and Florian A. Potra. Formulating dynamic multi-rigid-body contact problems with friction as solvable linear complementarity problems. *Nonlinear Dynamics*, 14 (3), 1997. URL `http://www.springerlink.com/index/J71678405QK31722.pdf`. 3.2.1, A.1.6

Sanjeev Arora, Simon S. Du, Wei Hu, Zhiyuan Li, and Ruosong Wang. Fine-Grained Analysis of Optimization and Generalization for Overparameterized Two-Layer Neural Networks. *arXiv:1901.08584 [cs, stat]*, May 2019. URL `http://arxiv.org/abs/1901.08584`. arXiv: 1901.08584. 2.2.7

Christopher J. Arthurs and Andrew P. King. Active Training of Physics-Informed Neural Networks to Aggregate and Interpolate Parametric Solutions to the Navier-Stokes Equations. *Journal of Computational Physics*, 438, 2021. doi: 10.1016/j.jcp.2021.110364. URL `http://arxiv.org/abs/2005.05092`. 5.1

Vsevolod I. Avrutskiy. Neural networks catching up with finite differences in solving partial differential equations in higher dimensions. *Neural Computing and Applications*, 32(17), 2020. doi: 10.1007/s00521-020-04743-8. URL `http://arxiv.org/abs/1712.05067`. 5.1

Yohai Bar-Sinai, Stephan Hoyer, Jason Hickey, and Michael P. Brenner. Learning data driven discretizations for partial differential equations. *Proceedings of the National Academy of Sciences*, 116(31), 2019. doi: 10.1073/pnas.1814058116. URL `http://arxiv.org/abs/1808.04930`. 1

Ronen Basri, David Jacobs, Yoni Kasten, and Shira Kritchman. The Convergence Rate of Neural Networks for Learned Functions of Different Frequencies. June 2019. URL `https://arxiv.org/abs/1906.00425v3`. 6.1

Peter W. Battaglia, Jessica B. Hamrick, and Joshua B. Tenenbaum. Simulation as an engine of physical scene understanding. *Proceedings of the National Academy of Sciences*, 110(45), 2013. doi: 10.1073/pnas.1306572110. URL `http://www.pnas.org/content/110/45/18327`. 2.2.3

Peter W. Battaglia, Razvan Pascanu, Matthew Lai, Danilo Rezende, and Koray Kavukcuoglu. Interaction Networks for Learning about Objects, Relations and Physics. 2016. URL `http://arxiv.org/abs/1612.00222`. 1, 2.2.3

Filipe de Avila Belbute-Peres, Thomas D. Economon, and J. Zico Kolter. Combining Differentiable PDE Solvers and Graph Neural Networks for Fluid Flow Prediction. July 2020. URL `http://arxiv.org/abs/2007.04439`. arXiv: 2007.04439. 1.1

Jens Berg and Kaj Nyström. A unified deep artificial neural network approach to partial differential equations in complex geometries. *Neurocomputing*, 317, 2018. doi: 10.1016/j.

neucom.2018.06.056. URL https://www.sciencedirect.com/science/article/pii/S092523121830794X. 5.1

Gino Johannes Apolonia van den Bergen. *Collison detection in interactive 3D environments*. The Morgan Kaufmann series in interactive 3D technology. Elsevier/Morgan Kaufman, Amsterdam ; Boston, 2004. ISBN 978-1-55860-801-6. A.1.4

Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym, 2016. 3.1, 3.3.3

Michael M Bronstein, Joan Bruna, Yann LeCun, Arthur Szlam, and Pierre Vandergheynst. Geometric deep learning: going beyond euclidean data. *IEEE Signal Processing Magazine*, 34(4), 2017. 2.2.5

Yuan Cao, Zhiying Fang, Yue Wu, Ding-Xuan Zhou, and Quanquan Gu. Towards Understanding the Spectral Bias of Deep Learning, October 2020. URL http://arxiv.org/abs/1912.01198. arXiv:1912.01198 [cs, stat]. 6.1

Erin Catto. Computing Distance Using GJK. In *GDC*, 2010. URL http://box2d.org/downloads/. A.1.4

Michael B. Chang, Tomer Ullman, Antonio Torralba, and Joshua B. Tenenbaum. A Compositional Object-Based Approach to Learning Physical Dynamics. 2016. URL http://arxiv.org/abs/1612.00341. 1, 2.2.3

Ricky T. Q. Chen, Yulia Rubanova, Jesse Bettencourt, and David Duvenaud. Neural Ordinary Differential Equations. 2018. URL http://arxiv.org/abs/1806.07366. 5.1

Michael Bradley Cline. *Rigid body simulation with contact and constraints*. PhD thesis, University of British Columbia, 2002. URL https://pdfs.semanticscholar.org/8567/e2467bb5ad67f3a3f11e7c3c4386d9ca8210.pdf. 3.1, 3.2.1, A.1, A.1.6

Richard W Cottle. Linear complementarity problem. In *Encyclopedia of Optimization*. Springer, 2008. 3.1

Erwin Coumans et al. Bullet physics library. *Open source: bulletphysics. org*, 15(49), 2013. 2.2.3, 3.1

Michaël Defferrard, Xavier Bresson, and Pierre Vandergheynst. Convolutional neural networks on graphs with fast localized spectral filtering. In *Advances in neural information processing systems*, 2016. 2.2.5

Jonas Degrave, Michiel Hermans, Joni Dambre, and Francis wyffels. A Differentiable Physics Engine for Deep Learning in Robotics. 2016. URL http://arxiv.org/abs/1611.01652. 2.2.3, 3.1

Prafulla Dhariwal, Christopher Hesse, Oleg Klimov, Alex Nichol, Matthias Plappert, Alec Radford, John Schulman, Szymon Sidor, and Yuhuai Wu. Openai baselines. https://github.com/openai/baselines, 2017. 3.3.3

Josip Djolonga and Andreas Krause. Differentiable learning of submodular models. In *Neural Information Processing Systems (NIPS)*, 2017. 2.2.2

Rachit Dubey, Pulkit Agrawal, Deepak Pathak, Thomas L. Griffiths, and Alexei A. Efros.

Investigating Human Priors for Playing Video Games. *arXiv:1802.10217 [cs]*, February 2018. URL http://arxiv.org/abs/1802.10217. arXiv: 1802.10217. 1

Karthik Duraisamy, Gianluca Iaccarino, and Heng Xiao. Turbulence Modeling in the Age of Data. *Annual Review of Fluid Mechanics*, 51(1), 2019. doi: 10.1146/annurev-fluid-010518-040547. URL http://arxiv.org/abs/1804.00183. 2.2.4

T. D. Economon, F. Palacios, and J. J. Alonso. Unsteady continuous adjoint approach for aerodynamic design on dynamic meshes. *AIAA Journal*, 53(9), 2015a. doi: 10.2514/1.J053763. 2.2.2

T. D. Economon, F. Palacios, S. R. Copeland, T. W. Lukaczyk, and J. J. Alonso. SU2: An open-source suite for multiphysics simulation and design. *AIAA Journal*, 54(3), 2016. doi: 10.2514/1.J053813. 2.2.2

Thomas D. Economon, Francisco Palacios, Sean R. Copeland, Trent W. Lukaczyk, and Juan J. Alonso. SU2: An Open-Source Suite for Multiphysics Simulation and Design. *AIAA Journal*, 54(3), 2015b. doi: 10.2514/1.J053813. URL https://arc.aiaa.org/doi/10.2514/1.J053813. 4.2.1

Sébastien Ehrhardt, Aron Monszpart, Andrea Vedaldi, and Niloy Mitra. Learning to Represent Mechanics via Long-term Extrapolation and Interpolation. 2017. URL http://arxiv.org/abs/1706.02179. 1

Hamidreza Eivazi, Mojtaba Tahani, Philipp Schlatter, and Ricardo Vinuesa. Physics-informed neural networks for solving Reynolds-averaged Navier$\unicode{x2013}$Stokes equations. 2021. URL http://arxiv.org/abs/2107.10711. 1.1, 5.1

Katerina Fragkiadaki, Pulkit Agrawal, Sergey Levine, and Jitendra Malik. Learning Visual Predictive Models of Physics for Playing Billiards. 2015. URL http://arxiv.org/abs/1511.07404. 3.3.2

Han Gao, Luning Sun, and Jian-Xun Wang. PhyGeoNet: Physics-Informed Geometry-Adaptive Convolutional Neural Networks for Solving Parameterized Steady-State PDEs on Irregular Domain. 2020. doi: 10.1016/j.jcp.2020.110079. URL https://arxiv.org/abs/2004.13145v2. 5.1

Helmut Garstenauer and Gerhard Kurka. *A unified framework for rigid body dynamics*. PhD thesis, 2006. A.1, A.1.6

Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial networks. 2014. 1.1

Dirk Gregorius. The Separating Axis Test. In *GDC*, 2013. URL http://box2d.org/downloads/. A.1.4

Dirk Gregorius. Robust Contact Creation for Physics Simulations. In *GDC*, 2015. URL http://box2d.org/downloads/. A.1.4

Mengwu Guo and Jan S. Hesthaven. Data-driven reduced order modeling for time-dependent problems. *Computer Methods in Applied Mechanics and Engineering*, 345, 2019. doi: 10.1016/j.cma.2018.10.029. URL https://www.sciencedirect.com/science/article/pii/S0045782518305334. 1

Xiaoxiao Guo, Wei Li, and Francesco Iorio. Convolutional neural networks for steady flow approximation. In *Proceedings of the 22nd ACM SIGKDD international conference on knowledge discovery and data mining*, 2016. 2.2.4, 4.1, 4.3.1

David Ha, Andrew Dai, and Quoc V. Le. HyperNetworks. 2016. URL `http://arxiv.org/abs/1609.09106`. 5.1, 5.2.4

Will Hamilton, Zhitao Ying, and Jure Leskovec. Inductive representation learning on large graphs. In *Advances in neural information processing systems*, 2017. 2.2.5

Jessica B. Hamrick, Kevin A. Smith, Thomas L. Griffiths, and Edward Vul. Think again? the amount of mental simulation tracks uncertainty in the outcome. In *Proceedings of the thirtyseventh annual conference of the cognitive science society*. Citeseer, 2015. URL `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.704.359&rep=rep1&type=pdf`. 2.2.3

Rana Hanocka, Amir Hertz, Noa Fish, Raja Giryes, Shachar Fleishman, and Daniel Cohen-Or. Meshcnn: a network with an edge. *ACM Transactions on Graphics (TOG)*, 38(4), 2019. 2.2.5

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. 2015. 1.1

S. He, K. Reif, and R. Unbehauen. Multilayer neural networks for solving a class of partial differential equations. *Neural Networks*, 13(3):385–396, April 2000. ISSN 08936080. doi: 10.1016/S0893-6080(00)00013-7. URL `https://linkinghub.elsevier.com/retrieve/pii/S0893608000000137`. 5.1

Michiel Hermans, Benjamin Schrauwen, Peter Bienstman, and Joni Dambre. Automated Design of Complex Dynamic Systems. *PLoS ONE*, 9(1), 2014. doi: 10.1371/journal.pone.0086696. URL `http://dx.plos.org/10.1371/journal.pone.0086696`. 2.2.3

Xiang Huang, Hongsheng Liu, Beiji Shi, Zidong Wang, Kang Yang, Yang Li, Bingya Weng, Min Wang, Haotian Chu, Jing Zhou, Fan Yu, Bei Hua, Lei Chen, and Bin Dong. Solving Partial Differential Equations with Point Source Based on Physics-Informed Neural Networks. *arXiv:2111.01394 [physics]*, November 2021a. URL `http://arxiv.org/abs/2111.01394`. arXiv: 2111.01394. 2.2.6, 6.1

Xinquan Huang, Tariq Alkhalifah, and Chao Song. A modified physics-informed neural network with positional encoding. page 2484, September 2021b. doi: 10.1190/segam2021-3584127.1. 2.2.6, 6.1

Arthur Jacot, Franck Gabriel, and Clement Hongler. Neural Tangent Kernel: Convergence and Generalization in Neural Networks. In *Advances in Neural Information Processing Systems*, volume 31. Curran Associates, Inc., 2018. URL `https://proceedings.neurips.cc/paper/2018/hash/5a4be1fa34e62bb8a6ec6b91d2462f5a-Abstract.html`. 2.2.7

Arthur Jacot, Franck Gabriel, and Clément Hongler. Neural tangent kernel: Convergence and generalization in neural networks. 2020. 7.1

A. Jameson. Aerodynamic design via control theory. *Journal of Scientific Computing*, 3, 1988. doi: 10.1007/BF01061285. 2.2.2

Xiaowei Jin, Shengze Cai, Hui Li, and George Em Karniadakis. NSFnets (Navier-Stokes Flow nets): Physics-informed neural networks for the incompressible Navier-Stokes equations. *Journal of Computational Physics*, 426, 2021. doi: 10.1016/j.jcp.2020.109951. URL `http://arxiv.org/abs/2003.06496`. 5.1

George Em Karniadakis, Ioannis G. Kevrekidis, Lu Lu, Paris Perdikaris, Sifan Wang, and Liu Yang. Physics-informed machine learning. *Nature Reviews Physics*, 3(6), 2021. doi: 10.1038/s42254-021-00314-5. URL `http://www.nature.com/articles/s42254-021-00314-5`. 5.1

Byungsoo Kim, Vinicius C. Azevedo, Nils Thuerey, Theodore Kim, Markus Gross, and Barbara Solenthaler. Deep Fluids: A Generative Network for Parameterized Fluid Simulations. 2018. URL `http://arxiv.org/abs/1806.02071`. 2.2.4

Ryan King, Oliver Hennigh, Arvind Mohan, and Michael Chertkov. From Deep to Physics-Informed Learning of Turbulence: Diagnostics. 2018. URL `http://arxiv.org/abs/1810.07785`. 2.2.4

Diederik P. Kingma and Jimmy Ba. Adam: A Method for Stochastic Optimization. 2014a. URL `http://arxiv.org/abs/1412.6980`. 4.2.2

Diederik P. Kingma and Jimmy Ba. Adam: A Method for Stochastic Optimization. 2014b. URL `http://arxiv.org/abs/1412.6980`. 6.2.2

Thomas N Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*, 2016. 2.2.5, 4.1, 4.2.1

Dmitrii Kochkov, Jamie A. Smith, Ayya Alieva, Qing Wang, Michael P. Brenner, and Stephan Hoyer. Machine learning accelerated computational fluid dynamics. 2021. URL `http://arxiv.org/abs/2102.01010`. arXiv: 2102.01010 version: 1. 1, 1.1

Isaac Elias Lagaris, Aristidis Likas, and Dimitrios I. Fotiadis. Artificial Neural Networks for Solving Ordinary and Partial Differential Equations. *IEEE Transactions on Neural Networks*, 9(5), 1998. doi: 10.1109/72.712178. URL `http://arxiv.org/abs/physics/9705023`. 5.1

Hyuk Lee and In Seok Kang. Neural algorithm for solving differential equations. *Journal of Computational Physics*, 91:110–131, November 1990. doi: 10.1016/0021-9991(90)90007-N. 5.1

Jaehoon Lee, Yasaman Bahri, Roman Novak, Samuel S. Schoenholz, Jeffrey Pennington, and Jascha Sohl-Dickstein. Deep Neural Networks as Gaussian Processes, March 2018a. URL `http://arxiv.org/abs/1711.00165`. arXiv:1711.00165 [cs, stat]. 2.2.7, 7.2.1, 7.2.2

Jaehoon Lee, Lechao Xiao, Samuel S. Schoenholz, Yasaman Bahri, Roman Novak, Jascha Sohl-Dickstein, and Jeffrey Pennington. Wide Neural Networks of Any Depth Evolve as Linear Models Under Gradient Descent. *Journal of Statistical Mechanics: Theory and Experiment*, 2019(12):124002, December 2020. ISSN 1742-5468. doi: 10.1088/1742-5468/abc62b. URL `http://arxiv.org/abs/1902.06720`. arXiv: 1902.06720. 2.2.7

Jeongseok Lee, Michael X. Grey, Sehoon Ha, Tobias Kunz, Sumit Jain, Yuting Ye, Siddhartha S. Srinivasa, Mike Stilman, and C. Karen Liu. DART: Dynamic Animation

and Robotics Toolkit. *The Journal of Open Source Software*, 3(22), 2018b. doi: 10.21105/joss.00500. URL http://joss.theoj.org/papers/10.21105/joss.00500. 2.2.3

Adam Lerer, Sam Gross, and Rob Fergus. Learning Physical Intuition of Block Towers by Example. 2016. URL http://arxiv.org/abs/1603.01312. 2.2.3

Weiwei Li and Emanuel Todorov. Iterative linear quadratic regulator design for nonlinear biological movement systems. In *ICINCO (1)*, 2004. 3.3.3

Chun Kai Ling, Fei Fang, and J Zico Kolter. What game are we playing? end-to-end learning in normal and extensive form games. *arXiv:1805.02777*, 2018. 2.2.2

Zichao Long, Yiping Lu, and Bin Dong. PDE-Net 2.0: Learning PDEs from Data with A Numeric-Symbolic Hybrid Deep Network. 2018. URL https://arxiv.org/abs/1812.04426v1. 1, 5.1

Miles Macklin and Matthias Müller. Position based fluids. *ACM Transactions on Graphics*, 32(4), 2013. doi: 10.1145/2461912.2461984. URL http://dl.acm.org/citation.cfm?doid=2461912.2461984. 2.2.4

X. Magnus and Heinz Neudecker. *Matrix differential calculus*. 1988. A.2.2

Nam Mai-Duy and Thanh Tran-Cong. Approximation of function and its derivatives using radial basis function networks. *Applied Mathematical Modelling*, 27(3):197–220, March 2003. ISSN 0307-904X. doi: 10.1016/S0307-904X(02)00101-4. URL https://www.sciencedirect.com/science/article/pii/S0307904X02001014. 5.1

Gary Marcus. Deep learning: A critical appraisal. 2018. 1.1

Jacob Mattingley and Stephen Boyd. CVXGEN: a code generator for embedded convex optimization. *Optimization and Engineering*, 13(1), 2012. doi: 10.1007/s11081-011-9176-9. URL http://link.springer.com/10.1007/s11081-011-9176-9. 3.2.2, A.2.1, A.2.1, A.2.1, A.2.1

Antoine McNamara, Adrien Treuille, Zoran Popovic, and Jos Stam. Fluid control using the adjoint method. *ACM Transactions On Graphics (TOG)*, 23(3), 2004. 2.2.2

Arthur Mensch and Mathieu Blondel. Differentiable dynamic programming for structured prediction and attention. *arXiv preprint arXiv:1802.03676*, 2018. 2.2.2

Ben Mildenhall, Pratul P. Srinivasan, Matthew Tancik, Jonathan T. Barron, Ravi Ramamoorthi, and Ren Ng. NeRF: Representing Scenes as Neural Radiance Fields for View Synthesis. *arXiv:2003.08934 [cs]*, August 2020. URL http://arxiv.org/abs/2003.08934. arXiv: 2003.08934. 2.2.6, 6.1

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540), 2015. doi: 10.1038/nature14236. URL http://www.nature.com/doifinder/10.1038/nature14236. 3.6

Radford M. Neal. Priors for infinite networks. Technical Report CRG-TR-94-1, University

of Toronto, 1994. URL https://www.cs.toronto.edu/~radford/ftp/pin.pdf. 2.2.7

Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. 2017. 3.1, 3.2.4

Charles Ruizhongtai Qi, Li Yi, Hao Su, and Leonidas J Guibas. Pointnet++: Deep hierarchical feature learning on point sets in a metric space. In *Advances in neural information processing systems*, 2017. 2.2.5, 4.2.1

Nasim Rahaman, Aristide Baratin, Devansh Arpit, Felix Draxler, Min Lin, Fred Hamprecht, Yoshua Bengio, and Aaron Courville. On the Spectral Bias of Neural Networks. In *Proceedings of the 36th International Conference on Machine Learning*, pages 5301–5310. PMLR, May 2019. URL https://proceedings.mlr.press/v97/rahaman19a.html. ISSN: 2640-3498. 6.1

Ali Rahimi and Benjamin Recht. Random Features for Large-Scale Kernel Machines. In *Advances in Neural Information Processing Systems*, volume 20. Curran Associates, Inc., 2007. URL https://papers.nips.cc/paper/2007/hash/013a006f03dbc5392effeb8f18fda755-Abstract.html. 2.2.6, 6.1

Maziar Raissi. Deep Hidden Physics Models: Deep Learning of Nonlinear Partial Differential Equations. 2018. URL http://arxiv.org/abs/1801.06637. 5.1

Maziar Raissi and George Em Karniadakis. Hidden Physics Models: Machine Learning of Nonlinear Partial Differential Equations. *Journal of Computational Physics*, 357, 2018. doi: 10.1016/j.jcp.2017.11.039. URL http://arxiv.org/abs/1708.00588. 5.1

Maziar Raissi, Paris Perdikaris, and George Em Karniadakis. Physics Informed Deep Learning (Part I): Data-driven Solutions of Nonlinear Partial Differential Equations. 2017. URL http://arxiv.org/abs/1711.10561. 5.1

Maziar Raissi, Paris Perdikaris, and George Em Karniadakis. Multistep Neural Networks for Data-driven Discovery of Nonlinear Dynamical Systems. 2018. URL http://arxiv.org/abs/1801.01236. 5.1, 5.2.3, 5.4.2, 6.1

Maziar Raissi, Paris Perdikaris, and George Em 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, 2019a. doi: 10.1016/j.jcp.2018.10.045. URL https://linkinghub.elsevier.com/retrieve/pii/S0021999118307125. 1, 2.1.2, 2.2.1, 5.1, 5.2.2, 7.6.2, 7.2, 7.6.2, 7.6.2, 7.6.2, 7.17, 7.6.2, 7.6.2, 7.6.2

Maziar Raissi, Zhicheng Wang, Michael S. Triantafyllou, and George Em Karniadakis. Deep Learning of Vortex Induced Vibrations. *Journal of Fluid Mechanics*, 861:119–137, February 2019b. ISSN 0022-1120, 1469-7645. doi: 10.1017/jfm.2018.872. URL http://arxiv.org/abs/1808.08952. arXiv: 1808.08952. 2.2.6, 6.1

Maziar Raissi, Alireza Yazdani, and George Em Karniadakis. Hidden fluid mechanics: Learning velocity and pressure fields from flow visualizations. *Science*, 367(6481), 2020. doi: 10.1126/science.aaw4741. URL https://www.sciencemag.org/lookup/doi/10.

1126/science.aaw4741. 5.1

Bharath Ramsundar, Dilip Krishnamurthy, and Venkatasubramanian Viswanathan. Differentiable Physics: A Position Piece, September 2021. URL http://arxiv.org/abs/2109.07573. arXiv:2109.07573 [physics]. 2.2.2

Anurag Ranjan and Michael J. Black. Optical Flow Estimation using a Spatial Pyramid Network. 2016. URL http://arxiv.org/abs/1611.00850. 3.3.2

David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *nature*, 323(6088):533–536, 1986. 2.1.1

Connor Schenck and Dieter Fox. SPNets: Differentiable Fluid Dynamics for Deep Neural Networks. 2018. URL https://arxiv.org/abs/1806.06094v2. 2.2.4

Steffen Schneider, Alexei Baevski, Ronan Collobert, and Michael Auli. wav2vec: Unsupervised pre-training for speech recognition. 2019. 1.1

Anand Pratap Singh, Karthikeyan Duraisamy, and Ze Jia Zhang. Augmentation of Turbulence Models Using Field Inversion and Machine Learning. In *55th AIAA Aerospace Sciences Meeting*. American Institute of Aeronautics and Astronautics, 2017. doi: 10.2514/6.2017-0993. URL https://arc.aiaa.org/doi/abs/10.2514/6.2017-0993. 2.2.4

Justin Sirignano and Konstantinos Spiliopoulos. DGM: A deep learning algorithm for solving partial differential equations. *Journal of Computational Physics*, 375, 2018. doi: 10.1016/j.jcp.2018.08.029. URL https://www.sciencedirect.com/science/article/pii/S0021999118305527. 5.1

Vincent Sitzmann, Julien N. P. Martel, Alexander W. Bergman, David B. Lindell, and Gordon Wetzstein. Implicit Neural Representations with Periodic Activation Functions. 2020. URL http://arxiv.org/abs/2006.09661. 2.2.6, 6.1, 6.2, 6.2.1, 6.2.1, 6.2.1, 6.2.2, 6.2.2, 6.2.2, 6.2.2, 6.1, 6.3, 6.3, 7.2.1, 7.6.2, 7.2

Kevin A. Smith and Edward Vul. Sources of Uncertainty in Intuitive Physics. *Topics in Cognitive Science*, 5(1), 2013. doi: 10.1111/tops.12009. URL http://onlinelibrary.wiley.com/doi/10.1111/tops.12009/abstract. 2.2.3

Chao Song, Tariq Alkhalifah, and Umair Bin Waheed. A versatile framework to solve the Helmholtz equation using physics-informed neural networks. *Geophysical Journal International*, 228(3):1750–1762, November 2021. ISSN 0956-540X, 1365-246X. doi: 10.1093/gji/ggab434. URL https://academic.oup.com/gji/article/228/3/1750/6409132. 2.2.6, 6.1

Elizabeth S. Spelke and Katherine D. Kinzler. Core knowledge. *Developmental science*, 10 (1):89–96, 2007. Publisher: Wiley Online Library. 1

Luning Sun, Han Gao, Shaowu Pan, and Jian-Xun Wang. Surrogate modeling for fluid flows based on physics-constrained deep learning without simulation data. *Computer Methods in Applied Mechanics and Engineering*, 361:112732, April 2020. ISSN 0045-7825. doi: 10.1016/j.cma.2019.112732. URL https://www.sciencedirect.com/science/article/pii/S004578251930622X. 5.1

Matthew Tancik, Pratul P. Srinivasan, Ben Mildenhall, Sara Fridovich-Keil, Nithin Raghavan, Utkarsh Singhal, Ravi Ramamoorthi, Jonathan T. Barron, and Ren Ng. Fourier features let networks learn high frequency functions in low dimensional domains. 2020. 2.2.6, 6.1

Ramakrishna Tipireddy, Paris Perdikaris, Panos Stinis, and Alexandre Tartakovsky. A comparative study of physics-informed neural network models for learning unknown dynamics and constitutive relations. 2019. URL `http://arxiv.org/abs/1904.04058`. 5.1

Emanuel Todorov, Tom Erez, and Yuval Tassa. MuJoCo: A physics engine for model-based control. IEEE, 2012. ISBN 978-1-4673-1736-8 978-1-4673-1737-5 978-1-4673-1735-1. doi: 10.1109/IROS.2012.6386109. URL `http://ieeexplore.ieee.org/document/6386109/`. 2.2.3, 3.1

Kiwon Um, Xiangyu Hu, and Nils Thuerey. Liquid Splash Modeling with Neural Networks. 2017. URL `http://arxiv.org/abs/1704.04456`. 2.2.4, 4.1

Hado van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. *CoRR*, abs/1509.06461, 2015. URL `http://arxiv.org/abs/1509.06461`. 3.3.3

Sifan Wang, Xinling Yu, and Paris Perdikaris. When and why pinns fail to train: A neural tangent kernel perspective. 2020. 6.1

Sifan Wang, Hanwen Wang, and Paris Perdikaris. On the eigenvector bias of fourier feature networks: From regression to solving multi-scale pdes with physics-informed neural networks. *Computer Methods in Applied Mechanics and Engineering*, 384:113938, 2021. 6.1

Steffen Wiewel, Moritz Becher, and Nils Thuerey. Latent-space Physics: Towards Learning the Temporal Evolution of Fluid Flow. 2018. URL `http://arxiv.org/abs/1802.10123`. 2.2.4, 4.1

Jian Cheng Wong, Chinchun Ooi, Abhishek Gupta, and Yew-Soon Ong. Learning in Sinusoidal Spaces with Physics-Informed Neural Networks. *arXiv:2109.09338 [physics]*, March 2022. URL `http://arxiv.org/abs/2109.09338`. arXiv: 2109.09338 version: 2. 2.2.6, 6.1