Technische Universität München
TUM School of Computation, Information and Technology

Solving Forward and Inverse Problems with Differentiable Physics and Deep Learning

Philipp Marcel Holl

Vollständiger Abdruck der von der TUM School of Computation, Information and Technology der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Die Dissertation wurde am 12.04.2024 bei der Technischen Universität München eingereicht und durch die TUM School of Computation, Information and Technology am 16.09.2024 angenommen.

Abstract

This dissertation encapsulates the research papers I authored during my tenure as a Ph.D. candidate at the Technical University of Munich, providing a comprehensive contextualization of their significance. My primary contributions revolve around the application of machine learning methodologies, particularly neural networks, to tackle both forward and inverse problems.

The focal point of my work lies within the realm of addressing numerically intricate scenarios, particularly those involving simulations of partial differential equations, such as the Navier-Stokes equations governing incompressible fluids.

In order to conduct the numerical experiments essential to these studies, I devised a software library tailored for differentiable simulations, known as Φ_{Flow}, which will be elaborated upon herein. The utilization of Φ_{Flow} is ubiquitous across all the publications enumerated in the subsequent pages, and it has garnered attention from independent researchers who have subsequently employed it in their own diverse publications.

The primary publications are reproduced at the end of this document.

Zusammenfassung

Diese Dissertation fasst die Forschungsarbeiten zusammen, die ich während meiner Zeit als Doktorand an der Technischen Universität München verfasst habe, und stellt eine umfassende Kontextualisierung ihrer Bedeutung dar. Meine primären Beiträge drehen sich um die Anwendung von Methoden des maschinellen Lernens, insbesondere von neuronalen Netzen, um sowohl Vorwärts als auch inverse Probleme zu lösen.

Der Schwerpunkt meiner Arbeit liegt im Bereich der Behandlung numerisch schwieriger Szenarien, insbesondere der Simulation partieller Differentialgleichungen, wie der Navier-Stokes-Gleichungen für inkompressible Fluide.

Um die für diese Studien erforderlichen numerischen Experimente durchzuführen, habe ich eine Software-Bibliothek Φ_{Flow} entwickelt, die auf differenzierbare Simulationen spezialisiert ist. Die Verwendung von Φ_{Flow} ist in allen auf den folgenden Seiten aufgezählten Publikationen allgegenwärtig und hat die Aufmerksamkeit unabhängiger Forscher auf sich gezogen, die es anschließend in ihren eigenen Publikationen eingesetzt haben.

Die wichtigsten Veröffentlichungen sind am Ende dieses Dokuments abgedruckt.

List of Publications

The following table lists all works I contributed to during my Ph.D. Contributions and copyright notices in chapter 6. Reprints of selected papers at the end of this document. Prior publications are not included in this list.

Core publications, peer-reviewed

JOSS, 2024 Journal Paper	**Φ_{ML}: A Science-oriented Math and Neural Network Library for Jax, PyTorch, TensorFlow & NumPy.** Philipp Holl, Nils Thuerey
NeurIPS 2022 Conference paper	**Scale-invariant Learning by Physics Inversion.** Philipp Holl, Vladlen Koltun, Nils Thuerey.
ICLR 2020 (spotlight) Conference paper	**Learning to Control PDEs with Differentiable Physics.** Philipp Holl, Vladlen Koltun, Nils Thuerey.

Other publications, peer-reviewed

ICLR 2022 (spotlight) Conference paper	**Half-Inverse Gradients for Physical Deep Learning.** Patrick Schnell, Philipp Holl, Nils Thuerey.
NeurIPS 2022 Conference paper	**Solver-in-the-Loop: Learning from Differentiable Physics to Interact with Iterative PDE-Solvers.** K. Um, R. Brand, Y. Fei, P. Holl, N. Thuerey.
NeurIPS 2020 Workshop paper	***phiflow*: A differentiable pde solving framework for deep learning via physical simulations.** P. Holl, V. Koltun, K. Um, N. Thuerey
ICML 2024 Conference paper	**Φ_{Flow}: Differentiable Simulations for PyTorch, TensorFlow and Jax.** Philipp Holl, Nils Thuerey.

Others, not peer-reviewed

Manuscript Submitted to ICML 2024	**Solving Inverse Physics Problems Jointly with Neural Networks** Philipp Holl, Nils Thuerey.
Book, 2021 Online book	**Physics-based Deep Learning.** N. Thuerey, Philipp Holl, M. Mueller, P. Schnell, F. Trost, K. Um
Tech report, 2022 ArXiv preprint	**Simulating Liquids with Graph Networks.** Jonathan Klimesch, Philipp Holl, Nils Thuerey

Contents

1 Introduction

This dissertation summarizes and contextualizes the various publications and contributions I have made during my work as a doctoral student at the Technical University of Munich. These broadly revolve around employing deep learning and differentiable physics for forward and inverse problems in science. This chapter introduces the general topic and outlines my contributions to the field.

1.1 Machine Learning in Science and Engineering

In recent years, the integration of machine learning techniques into scientific and engineering disciplines has revolutionized the way we approach complex problems, driving innovation and discovery across a spectrum of fields [LBH15]. From deciphering intricate biological mechanisms to optimizing manufacturing processes, the application of machine learning algorithms has become increasingly pervasive, offering unprecedented insights and efficiencies [JM15]. This dissertation delves into the profound impact of machine learning methodologies for problem solving in science and engineering, highlighting my own contributions along the way and examining existing challenges and opportunities.

The convergence of machine learning and traditional scientific and engineering practices has ushered in a new era of exploration and problem-solving. By leveraging vast datasets and powerful computational tools, researchers have been able to unravel intricate patterns, predict behaviors, and uncover hidden relationships within complex systems [GHV17]. Whether it be in the realms of materials science, environmental engineering, or biomedical research, machine learning algorithms have demonstrated remarkable capabilities in accelerating the pace of discovery and innovation.

However, despite the remarkable progress achieved, significant challenges persist. The interpretability of machine learning models and the robustness of algorithms in the face of uncertainty are but a few of the pressing issues that demand careful consideration [Lip18]. Moreover, the interdisciplinary nature of many scientific and engineering problems necessitates a holistic approach that bridges the gap between domain expertise and computational proficiency [Car+15].

This dissertation seeks to address these challenges and provide a comprehensive overview of the current state-of-the-art in machine learning for science and engineering. Through a synthesis of theoretical frameworks, practical case studies, and critical analysis, it aims to shed light on both the promise and the pitfalls of integrating machine learning techniques into traditional research methodologies. By examining real-world applications across diverse domains, it endeavors to uncover best practices, identify areas for improvement, and chart a course towards the responsible and effective utilization of machine learning in pursuit of scientific and engineering advancements.

As we stand on the cusp of a new era defined by unprecedented technological capabilities, the exploration of machine learning for science and engineering represents not only a scientific endeavor but also a philosophical and societal one. By embracing the potential of these transformative technologies while remaining vigilant to their limitations and implications, we can forge a path towards a future where innovation is empowered by intelligence, and progress is driven by knowledge.

1.2 My contributions

I will begin by reviewing the necessary background required to understand my research. Chapter 2 includes an overview of current machine learning methods as well as partial differential equations (PDEs) which govern the evolution of physical systems.

A large part of my time as a doctoral student was spent developing tools for simulating complex PDEs in a differentiable way to enable advanced machine learning methods. This effort has resulted in the software libraries Φ_{Flow} and Φ_{ML} which I will introduce in chapter 3.

With access to differentiable simulations, my colleagues and I have trained neural networks on various tasks which can broadly be categorized into two types, forward problems and inverse problems. In this context, forward problems refer to solving PDEs given the initial and boundary conditions. Deep learning can be used for forward problems in various ways, and we have made three separate contributions to this field, which are explained in chapter 4. First, we have trained neural networks to act as surrogates for a particle-based liquid simulator (4.2). Second, we have trained correctors to reproduce the behavior of higher-resolution solvers by learning the evolution differences over many time steps (4.3). Third, we have explored using neural networks to approximate a smoothed version of physics-based objective functions to improve optimization (4.4).

Applying machine learning techniques to *inverse* problems poses a different set of challenges, and I have made multiple contributions to this area as well. Chapter 5.1 presents my work on the advantages of optimizing inverse problems with neural networks compared to traditional optimizers. Then in 5.2, I present my work on controlling dynamical systems as a practical application of this approach. In performing these studies, we have identified some fundamental limitations in traditional machine learning methods, which have led to two improvements for neural network training involving the principle of update inversion. Chapter 5.4 explains my work on inverting the physics, and chapter 5.5 summarizes related work I have contributed to, which inverts the gradients.

2 Background and Preliminaries

This chapter introduces machine learning as a whole as well as partial differential equations (PDEs) and traditional methods of solving them. Chapter 3 then investigates the interface between these two, which naturally leads to differentiable physics and the Φ_{Flow} library I implemented.

2.1 Deep learning and its applications

Deep learning, a subfield of artificial intelligence (AI), has emerged as a transformative force in modern science and technology [Goo16]. It represents a class of machine learning algorithms that enable computers to learn complex representations of data through the use of deep neural networks, inspired by the structure and function of the human brain [LBH15]. The evolution of deep learning has revolutionized various scientific disciplines, offering unprecedented capabilities in data analysis, pattern recognition, and decision-making processes [Sch15].

Historically, the roots of deep learning can be traced back to the development of artificial neural networks (ANNs) in the 1940s and 1950s [MP43; Ros58]. However, it wasn't until the 1980s and 1990s that significant advancements were made in training deep neural networks with the introduction of backpropagation and other optimization techniques [RHW86]. Despite these advancements, deep learning faced significant challenges due to limited computational power and insufficient amounts of labeled data [HOT06].

The breakthroughs in deep learning can be largely attributed to the convergence of several factors, including the exponential growth of computational resources, the proliferation of big data, and innovative algorithmic developments. The resurgence of deep learning began in the mid-2000s, with the introduction of deep convolutional neural networks (CNNs) by Hinton et al., which demonstrated remarkable performance in image classification tasks. Subsequently, deep learning techniques have been extended and applied to a wide range of scientific domains, including but not limited to:

1. Biomedical Imaging: Deep learning has shown exceptional promise in medical image analysis, facilitating tasks such as disease diagnosis, tumor detection, and organ segmentation. For instance, CNNs have been employed to interpret radiological images with high accuracy, aiding clinicians in making more informed decisions [KSH12].
2. Genomics and Proteomics: In genomics, deep learning techniques have been utilized to analyze DNA sequences, predict gene functions, and identify genetic variations associated with diseases. Similarly, in proteomics, deep learning models have been applied to protein structure prediction and drug discovery [Ang+16].
3. Drug Discovery and Development: Deep learning has the potential to revolutionize drug discovery pipelines by accelerating the process of drug screening, lead optimization, and target identification. Deep learning models can learn complex relationships between chemical structures and biological activities, leading to the design of novel therapeutics [GHS16].
4. Climate Science: Deep learning techniques have been employed to analyze climate data, predict extreme weather events, and model climate patterns. By leveraging deep neural networks, researchers can extract valuable insights from large-scale climate datasets, contributing to our understanding of climate dynamics and informing mitigation strategies [RPG18].
5. Neuroscience: Deep learning plays a crucial role in analyzing neuroimaging data, deciphering brain connectivity networks, and understanding neural activity patterns. These insights are instrumental in advancing our understanding of brain function and neurological disorders [Sch+17].

Furthermore, deep learning finds application in engineering [Alz+21; And+23], computer vision [Vas+17], speech recognition [Hin+12], medical imaging [Lit+17], autonomous vehicles [Boj+16], recommender systems [CAS16], financial services [BYR17], drug discovery [GHV17], gaming [Mni+13a], robotics [Lev+16], cybersecurity [SB15], agriculture [KP18], retail [Tan+18], energy [Hos+17; Li+24], among others.

Overall, deep learning represents a paradigm shift in scientific research, empowering scientists and researchers to tackle complex problems across various disciplines. With ongoing advancements in hardware capabilities, algorithmic innovations, and interdisciplinary collaborations, the potential applications of deep learning in science continue to expand, promising transformative breakthroughs in the years to come.

2.2 Partial differential equations (PDEs)

Partial Differential Equations (PDEs) are fundamental mathematical tools used to describe various physical phenomena and processes occurring in fields such as physics, engineering, biology, and economics [CH62; SSS85].

Unlike ordinary differential equations (ODEs), which involve functions of a single variable, PDEs involve functions of several variables and their partial derivatives [Eva22; Str07; Hab04].

The study of PDEs dates back to the 18th century, with seminal contributions from mathematicians such as Leonhard Euler and Joseph-Louis Lagrange. However, their significance in modeling real-world phenomena became more pronounced with the advent of modern science and technology. PDEs provide a powerful framework for describing continuous systems and have applications ranging from fluid dynamics and heat transfer to electromagnetism and quantum mechanics.

A generic form of a partial differential equation can be expressed as:

$$F\left(x_1, x_2, \ldots, x_n, u, \frac{\partial u}{\partial x_1}, \frac{\partial u}{\partial x_2}, \ldots, \frac{\partial^m u}{\partial x_n^m}\right) = 0 \tag{2.1}$$

where u is the unknown function of variables $x_1, x_2, ..., x_n$, and F is a given function involving u and its partial derivatives up to order m.

Most PDEs describing real systems have time t as one of their variables and these terms can usually be separated. Higher-order temporal derivatives can further be reduced to first-order by increasing the dimensionality of the dynamical system, e.g. describing the evolution of phase space. This results in the following form:

$$\frac{\partial u}{\partial t} = F\left(t, x_1, x_2, \ldots, x_n, u, \frac{\partial u}{\partial x_1}, \frac{\partial u}{\partial x_2}, \ldots, \frac{\partial^m u}{\partial x_n^m}\right) \tag{2.2}$$

PDEs are classified based on their order, linearity, and the number of independent variables. Common types include elliptic, parabolic, and hyperbolic equations, each with distinct characteristics and solution methods.

Elliptic equations, such as Laplace's equation, describe steady-state phenomena and are characterized by smooth solutions. Parabolic equations, exemplified by the heat equation (Fig. 2.1), govern processes involving diffusion and exhibit solutions that evolve over time. Hyperbolic equations, like the wave equation, model wave propagation phenomena with solutions that propagate along characteristic curves.

The study of PDEs encompasses a wide range of mathematical techniques, including separation of variables, Fourier and Laplace transforms, numerical methods, and variational principles. These tools enable the analysis and solution of complex problems arising in diverse fields of science and engineering. In the following section, we will explore numerical schemes for solving them, before moving on to machine learning methods in later chapters.

2.3 Traditional PDE solving methods

Solving PDEs analytically is often intractable for complex systems, necessitating the development of numerical methods. Here we present an overview of traditional numerical techniques for solving PDEs.

Finite Difference Methods Finite Difference Methods (FDM) discretize PDEs by approximating derivatives using finite difference approximations. The domain is discretized into a grid, and the PDE is solved

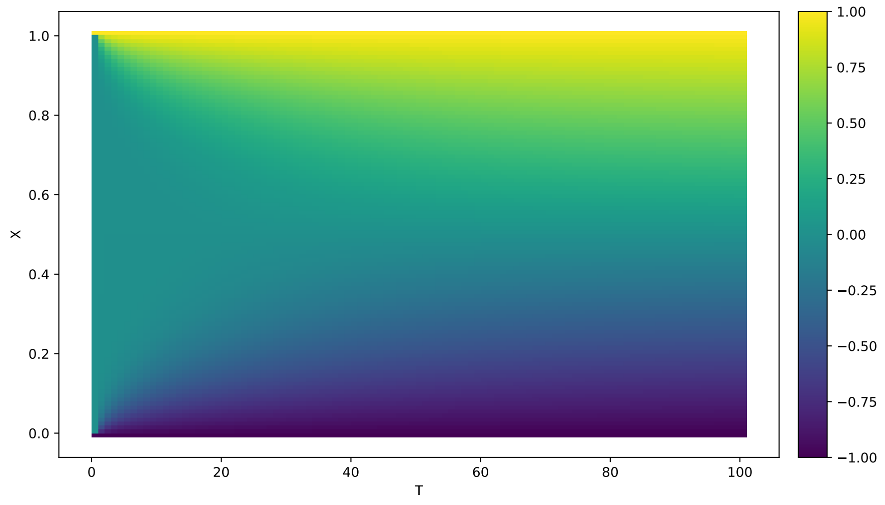


Figure 2.1: 1D heat convection. Initially (left edge) all values at constant temperature $u = 0$. The boundaries are constrained to $u = \pm 1$. The PDE $\frac{\partial u}{\partial t} = 10^{-3} \cdot \nabla^2 u$ determines the evolution. Simulated with Φ_{Flow}.

iteratively at grid points. Common schemes include explicit, implicit, and Crank-Nicolson methods, each offering trade-offs between stability, accuracy, and computational efficiency [LeV07]. FDM is conceptually simple and applicable to a wide range of PDEs, making it a foundational technique in computational physics and engineering.

Finite Element Methods Finite Element Methods (FEM) discretize the domain into finite elements, where the PDE is approximated by piecewise polynomial functions defined over these elements. FEM provides flexibility in handling irregular geometries and complex boundary conditions [ZT05]. It achieves higher accuracy compared to FDM for problems with smooth solutions but requires solving a system of algebraic equations, making it computationally intensive.

Finite Volume Methods Finite Volume Methods (FVM) discretize PDEs by dividing the domain into control volumes and approximating integrals over these volumes. FVM emphasizes conservation laws and fluxes across control volume interfaces. It is widely used in fluid dynamics and heat transfer simulations due to its robustness and conservation properties [FP02]. FVM is particularly suitable for problems with discontinuous solutions or shocks.

Spectral Methods Spectral Methods represent solutions using basis functions such as Fourier, Chebyshev, or Legendre polynomials. These methods provide exponential convergence rates for smooth solutions and are well-suited for problems with periodic boundary conditions [Boy01]. However, they are less flexible for handling complex geometries and boundary conditions compared to finite element methods.

Traditional PDE solving techniques offer a diverse toolkit for approximating solutions to PDEs across various disciplines. Each method has its strengths and weaknesses, and the choice depends on factors such as problem characteristics, computational resources, and desired accuracy. This begs the question as to whether machine learning approaches can improve upon the existing algorithms. As we will see, a key ingredient to enable advanced machine learning techniques is being able to differentiate through these numerical methods in order to obtain feedback for optimizing the learned model. In the next chapter, I will introduce Φ_{Flow}, a software library I developed to tightly integrate differentiable simulations with existing machine learning libraries.

3 Differentiable Physics and Φ_{Flow}

Differentiable physics refers to simulated processes for which we can compute derivatives. Imagine we can simulate a process $\mathcal{P}(x\,|\,\Phi)$, such as a weather forecast, to obtain a result y, such as the weather some days after the initial state x. This simulation is influenced by various parameters Φ, like modelling the influence of microphysics or turbulence [Cou+12]. Then to improve the simulation, we must find the optimal parameters Φ to reduce the difference between forecast and recorded weather data $L(x)$ on a set of historical weather data $\mathbb{D}$. And to find the optimal parameters, we would like to know how L is affected by each parameter, i.e. $\frac{\partial L}{\partial \Phi}$. For a squared error metric, this gradient is $\frac{\partial L}{\partial y}\frac{\partial \mathcal{P}}{\partial \Phi}$. Here, the second term denotes the derivative of the simulation output w.r.t. the parameters Φ. We call a simulation *differentiable*, if this quantity and/or $\frac{\partial \mathcal{P}}{\partial x}$ (and optionally higher-order derivatives) can be computed efficiently, i.e. the computational cost scales sub-linearly with the number of parameters Φ. This is typically achieved by calculating the analytical derivative via the adjoint method, i.e. backpropagation [Goo16].

3.1 Applications of Differentiable Physics

Numerical applications involving processes can broadly be categorized into *forward* and *inverse* problems. For a known process $\mathcal{P}(x\,|\,\Phi) \to y$, forward problems involve solving for y given (x, Φ), while inverse problems generally search for parameters from either x or Φ. In the context of PDEs, forward problems amount to solving the initial value problem given by the specified initial state x and boundary conditions Φ. The classification into forward and backward problems is somewhat ambiguous, since it depends on the definition of $\mathcal{P}$. Furthermore there are applications that require solving both forward and inverse problems.

In recent years, researchers have applied machine learning techniques to a wide variety of both forward and inverse problems. Many of the corresponding applications are made possible by differentiable physics or can strongly benefit from its usage. Fig. 3.1 categorizes a broad range of methods by the type of problem they are designed to solve as well as their usage of machine learning and differentiable physics.

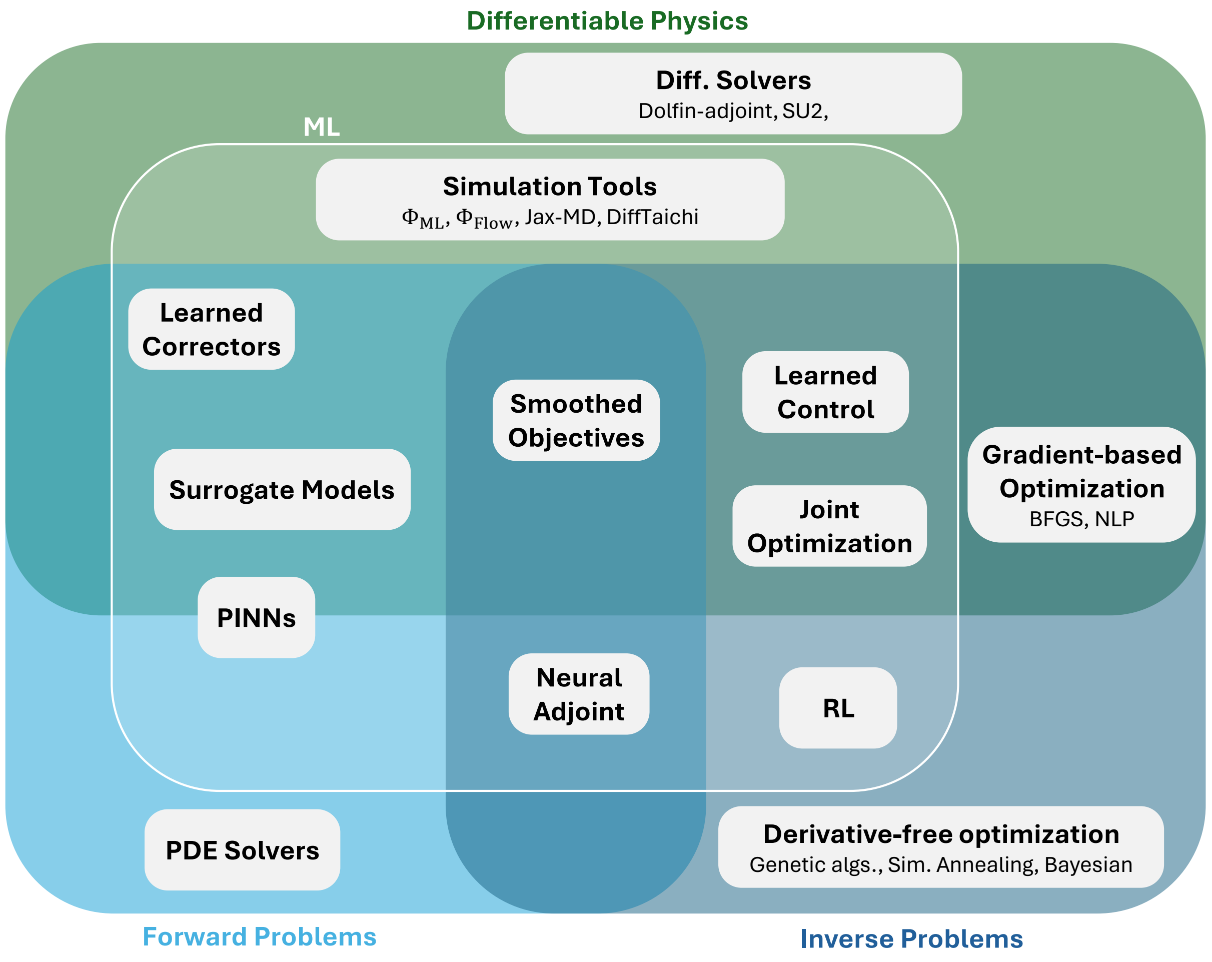


Figure 3.1: Overview of methods and tools used to solve forward as well as inverse problems.

The combination of deep learning and physics simulations has sparked a multitude of promising lines of research. For applications where PDE solvers are traditionally employed, surrogate ML models have been shown to provide significant performance improvements for certain types of problems [San+20; Tom+17] or improve simulation accuracy for fixed resolutions [Koc+21b; Um+20]. This can come in the form of completely replacing PDE solvers or augmenting them by providing correction terms to model unresolved dynamics. These models can be trained without differentiable physics on single time steps, but this training scheme is prone to instability at inference, when the models are autoregressively executed on long sequences. Here, differentiable physics enables rolling out multiple time steps during network training. We have shown that this reduces the risk of numerical instabilities at inference time and benefits accuracy [Um+20].

PINNs are an alternative way to of solving PDEs using neural networks. PINNs directly use the partial derivatives of the governing PDE as a training objective but do not require a differentiable simulator. They will be introduced in more detail in chapter 4.

In addition to these applications for forward problems, ML has also seen success for inverse problems. Most classical inverse problems involve estimating parameters of a known model $\mathcal{P}$ from observations, but learning the optimal behavior of agents also falls into this category. There are numerous classical algorithms to solve these problems, which can be broadly classified by whether they require the derivative of the objective function w.r.t. the unknown parameters. Derivative-free optimization can be applied to discontinuous or integer problems as well, but for continuous and differentiable problems, making use of the gradient and possibly higher-order derivatives via differentiable physics can significantly speed up convergencen [JOB91; Jam03].

In recent years, neural networks have been applied to inverse problems from science, engineering, robotics, health care, video and board games, and many more. Notable breakthroughs have been achieved using reinforcement learning (RL), such as learning to play Atari games [Mni+13b], Go [Sil+17] and Dota [Ber+19]. RL relies on a non-differentiable model of the environment, which allows it to be applied to a vast range of problems. On continuous problems, RL research has been working on integrating differentiable models into the training. This branch, known as *model-based* RL [Moe+23], is converging with the work from the science and engineering community on ML methods for inverse problems.

There, ML methods have been applied to inverse problems, such as controlling complex physical systems [Bie+20; HKT20], encoding physical states and sequences [RPK19], and finding conservation laws [GDY19]. These approaches are based on the assumption that there are exploitable patterns in the optimal solutions to similar inverse problems. Chapter 5 provides more information on ML methods for scientific inverse problems, including my contributions to the field.

3.2 Backpropagation and the adjoint method

Differentiable simulations have long been used in classical optimization where the adjoint method is typically employed to compute the required gradients [Ple06]. The adjoint method is also used in machine learning, where it is known as reverse-mode differentiation or simply backpropagation. Consequently, it can also be used to backpropagate through joint systems comprising of both neural networks and physical simulations as long as all parts are differentiable.

Despite this deep connection between the two fields, most software frameworks focus on only one of them. There are frameworks for differentiable simulations [TET12; MFD19] and separate frameworks for neural network optimization, such as PyTorch [Pas+19], TensorFlow [Aba+16], and Jax [Bra+18; Bab+20; Hen+20]. Combining these frameworks is hard to achieve in practice and many researchers have instead chosen to implement custom differentiable simulations compatible with one specific machine learning framework [Tom+17; Koc+21b; Bie+20]. However, this approach results in highly specialized and low-level simulation code, preventing adoption to different projects.

A number of libraries target this problem [SC20; Hu+20] but they are either very specialized or use different programming paradigms than the popular machine learning frameworks, making seamless integration difficult. The programming language Julia [Bez+17] offers language-level differentiation but is not compatible with most established machine learning frameworks. The difficulty in training neural networks with differentiable physics has led many authors to fall back to supervised learning [San+20; RT21; TLY20; Sta+21].

3.3 Φ_{ML} and Φ_{Flow}

Φ_{Flow} (PhiFlow) is an open-source framework for differentiable simulations I developed during my Ph.D. Originally developed as one project, Φ_{Flow} is now distinct from Φ_{ML} which provides a science-oriented tensor API and fully integrates with PyTorch, TensorFlow, Jax and NumPy [Har+20]. Φ_{Flow} builds on Φ_{ML} to provide access to geometry and physics functions and and is intended to be used for a wide variety of simulations. It includes high-level data structures for grid-based (Eulerian) as well as particle-based (Lagrangian) simulations. Φ_{Flow} is designed to make simulation code as reusable as possible without sacrificing readability or performance. Additionally Φ_{Flow} aims to accelerate development iterations by promoting interactivity and clean code.

3.4 Design Principles

Here, we lay out our goals in developing Φ_{ML} and Φ_{Flow}. These serve as the foundation for the design as well as the metrics for their evaluation.

Reusability Simulation code based on Φ_{Flow} should be able to run in many different settings without modification. The dynamics of a system, e.g. governed by a partial differential equations, are often formulated in a dimension-agnostic manner. Simulation code implementing these dynamics should also exhibit that property. Most simulations use some form of discretization, such as particles or grids. Simulation code written for one such discretization should be easy to port to another appropritate one.

Compatibility There are many toolkits and libraries extending machine learning frameworks with specialized functionality. These are generally only available for a certain framework, be it TensorFlow, PyTorch or Jax. Φ_{Flow} users should be free to choose whatever framework they desire without modifying their simulation code. Additionally, simulations should be able to run on GPUs and CPUs and be vectorizable without modification. Φ_{Flow} should support Linux, Windows and Mac.

Interactivity To develop code, tune parameters or find issues quickly, it is vital to get immediate visual feedback on what the code is doing and influence it if necessary. This requires a user interface that can visualize physical quantities as well as statistics and provide user-specified controls to adjust parameters. It should work for remote processes as well as locally executed scripts. Users should be able to halt execution at any point using a debugger and visualize variable values.

Usability Φ_{Flow} should be easy to learn and use. To achieve this, the API should be intuitive with expressively named functions matching existing frameworks where possible. User code as well as built-in simulation functionality should be easy to read, i.e. concise and expressive.

Maintainability Users should be able to read and understand all high-level source code of Φ_{Flow}. All relevant framework functions should undergo continuous testing to ensure patches do not break existing code. When installing Φ_{Flow}, users should be able to check the installation status and get hints as to how to solve potential issues.

Performance Simulations using Φ_{Flow} should make use of hardware accelerators (GPUs, TPUs) where possible. During development, we prioritize rapid code iterations over execution speed but the completed code should run as fast as if written directly against the chosen framework.

3.5 Design

We choose Python [VD95] 3.6 and newer as the main programming language for Φ_{ML}'s and Φ_{Flow}'s API due to its simplicity and compatibility with machine learning frameworks and operating systems. We also implement all core functionality in Python to enable users to easily locate and understand the implementation of all library functions. In the following, we list the major design decisions, which affect both libraries.

Layered Architecture Φ_{Flow} consists of five sub-modules forming a software stack (Fig. 3.2), each designed to minimally wrap functionality with as little intersection to each other as possible.. The `math` module, now part of Φ_{ML}, manages tensor calculus and includes BLAS, differentiation, jit-compilation and optimization functionality. The `geometry` package provides classes to represent n-dimensional shapes and collections of shapes. The module `field` contains various data structures to discretize continuous physical fields, such as centered and staggered grids and point clouds. The functions in the `physics` module operate on these and serve as building blocks for physics simulations. Finally, the `vis` module includes user interface and plotting functionality for fields and tensors.

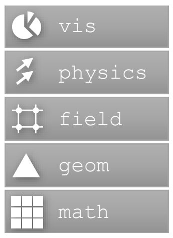


Figure 3.2: Modules of Φ_{Flow}. The `math` API is part of Φ_{ML}.

Uniform API for PyTorch, TensorFlow, Jax and NumPy Φ_{ML}'s math API is similar to NumPy, containing a `Tensor` class and functions to operate on tensors. However, we extend the base functionality that is present in all popular machine learning frameworks by new features to make writing reusable code easier. Φ_{ML} tensors can be created from all supported backends – PyTorch, TensorFlow, Jax and NumPy – and act as a wrappers for the underlying arrays which we refer to as *native tensors*. Since all native tensors are represented by the same class in Φ_{ML}, code written against Φ_{ML}'s math API is backend-agnostic. Data can also be passed between backends, internally using the tensor sharing functionality of DLPack [al17] when possible. This way, an easy-to-use PyTorch network can interact with a Jax simulation for performance but also with an identical PyTorch simulation to facilitate debugging.

Named dimensions In Φ_{ML} and Φ_{Flow}, dimensions are not referenced by their index but are given a name instead, similar to pandas [McK10]. This allows code to only interact with certain dimensions while being agnostic to the number and order of other dimensions. This is especially relevant considering PyTorch networks require dimensions to be laid out in the order `batch, channel, spatial` while TensorFlow needs `batch, spatial, channel`. It also saves users from reshaping tensors because non-matching tensors can internally be reshaped to match, adding missing dimensions as needed.

Element names along dimensions Like with the dimension orders described above, there are also multiple conventions for component orders, such as storing vectors as (x, y, z) or (z, y, x). Φ_{ML} supports component-order-agnostic code by allowing users to specify the component order explicitly. This enables extracting the x component of a vector-valued tensor using the syntax `tensor.vector['x']` instead of the traditional `tensor[:, 0, ...]` or `tensor[:, -1, ...]`, assuming PyTorch dimension order.

Non-uniform tensors With some data structures, such as staggered grids, the number of elements along one or multiple dimensions can be variable. We will refer to tensors holding such data as *non-uniform* tensors but they are also known as *ragged* or *nested* tensors. Users will often pad the missing elements with zeros to make the data easier to handle but this can lead to problems down the line. Instead, Φ_{ML} automatically creates non-uniform tensors when stacking tensors with non-matching shapes. The `shape` attribute of a non-uniform tensor stores its exact layout, allowing users to operate on non-uniform shapes like on regular shapes, e.g. allocating new memory with `zeros(non_uniform_shape)`.

Unified functional math For differentiation, just-in-time compilation and iterative solves, we adopt a function-based approach similar to Jax. This is different from TensorFlow, where gradients are tracked via Python context managers, and PyTorch, where gradients are attached to tensors. Φ_{ML} unifies these different paradigms, providing unified function operations that run with all backends. For example, `math.functional_gradient(f)` returns a function that computes the gradient of `f` and, to solve a sparse system of linear equations, users simply supply a Python function and the desired output of that function.

Built-in simulation and learning functionality To make it as easy as possible for users to get started, Φ_{ML} ships with many common building blocks for simulations, such as classes to represent grids and point clouds. Built-in functions like finite-difference operators, diffusion, advection or pressure computation for fluids allow users to quickly assemble readable and flexible physics simulations. Additionally, they facilitate altering the physics later, e.g. switching resolution, domain size or boundary conditions. All provided classes and functions are implemented in Python and interface only with Φ_{ML}'s math API, not any particular backend.

This makes it easy for users to navigate and read their source code. Φ_{ML} also provides generic convenience functions for setting up common neural network types, such as fully-connected networks or U-Nets [RFB15].

Dimension types Each tensor dimension in Φ_{ML} is assigned one of four types: *batch*, *spatial*, *instance* or *channel*. *Batch* dimensions have no physical interpretation but serve to run operations in parallel. All math functions treat slices along batch dimensions as independent, i.e. all operations are performed element-wise along batch dimensions. Since all functions in Φ_{ML}'s math module adhere to this rule, user code will also exhibit that property unless batch dimensions are referenced explicitly. This enables seamless parallelization for efficient GPU utilization without additional functions like `vmap`. *Spatial* dimensions list data sampled at regular intervals. They define the physical space in which spatial operations like grid sampling or the Fourier transform operate in. Thus, a simulation written for 2D can be scaled to 3D simply by adding an additional spatial dimension to the initial state, assuming the simulation code does not explicitly reference a specific spatial dimension. This feature mimics the operator notation in mathematics and abstracts out the error-prone process of porting code to new shapes. In contrast to spatial dimensions, *instance* dimensions list data sampled at *irregular* intervals. They can be used to represent collections of points, particles, finite volumes or elements as well as graph nodes. Lastly, *channel* dimensions enumerate properties of single instances or sample points, such as the x, y and z components of a vector field.

Floating-point precision by context Φ_{ML}'s tensor operations determine the desired floating point precision from the operation context rather than the data types of its inputs. The precision can be set globally or specified locally via context managers and operations will automatically convert tensors of non-matching data types if necessary. This avoids data-type-related problems and errors, as well as making user code more concise and cohesive.

Lazy stacking Simulations often perform component-wise operations separately if there is no function achieving the desired effect with a single call, like computing the x, y and z-component of a velocity field in three lines. This often leads users to declare separate variables for the components to avoid repeated tensor stacking and slicing. However, this clutters the code and prevents it from being dimension-agnostic. Instead, Φ_{ML} performs lazy stacking by default, i.e. memory is only allocated once the stacked data is required as a block. Consequently, functions can unstack the components, operate on them individually, and restack them, without worrying about unnecessary memory allocations.

Just-in-time compilation While the previous features allow for concise, expressive and flexible code, the added abstraction layer and shape tracking induces additional overhead. To avoid this in production, Φ_{ML} supports just-in-time (JIT) compilation for PyTorch, TensorFlow and Jax. Once compiled, only the tensor operations are executed, eliminating all Python-based overhead.

Sparse matrices from linear functions Solving linear systems of equations is a key requirement in both particle and grid-based simulations. Since the physical influence is typically limited to neighboring sample points or particles, the resulting linear systems are often sparse. Constructing such sparse matrices by hand yields code that is hard to understand and debug as well as limited to specific boundary conditions. Instead, Φ_{ML} lets users specify linear systems with a linear Python function, like with matrix-free solvers. However, these functions often consist of many individual operations, which makes it inefficient to call them at each solver iteration. To avoid this overhead, Φ_{ML} can convert most linear and affine functions to sparse matrices so that solvers can perform the matrix multiplication in a single operation. When JIT-compiling a simulation that includes a linear solve, the matrix generation will be performed during the initial tracing of the function, assuming the sparsity pattern is constant.

Custom CUDA Operatorions Φ_{ML} provides custom CUDA kernels for specific operations that could bottleneck simulations, such as grid sampling for TensorFlow or linear solves. If available, these will be used automatically in place of the fallback Python implementation.

Plotting recipes Φ_{Flow} provides convenience plotting recipes for tensors and all built-in data structures, such as grids and point clouds. These plots can be created with a single function call and can be used during debugging, e.g. to visualize a variable. Φ_{Flow} supports the popular plotting libraries Matplotlib [Hun07] and Plotly [Inc15] as well as basic visualization for the command line.

Web-based user interface To provide instant feedback and enable interactivity, Φ_{Flow} includes user interfaces for both Jupyter notebooks [Klu+16] and stand-alone Python scripts. The user interfaces employ the above-mentioned plotting recipes to show selected module or notebook variables in real time. They can also display graphs to track scalar quantities, such as learning curves. By annotating numeric, Boolean or textual variables or functions, control components can be added to the interface, allowing users to interact with the code, e.g. to adjust the learning rate or to tweak physical parameters in real time without restarting the program. Φ_{Flow}'s interface uses Jupyter interactive widgets [com15] for notebooks and Dash [Inc15] for the web-based interface, enabling supervision and interaction with remote scripts.

3.6 Evaluation

We now test whether Φ_{Flow}'s design fulfills the design principles. Specifically, we examine the design principles in order and measure the degree to which they apply to Φ_{Flow} as objectively as possible. We consider examples implementations for three important applications of Φ_{Flow}: running simulations, optimizing physical parameters, and training neural networks.

Running simulations without computing gradients is a common task in learning as well as science, e.g. to generate synthetic data sets for training or validation. For this task, we set up a galaxy-like simulation,

Table 3.1: Number of lines changed by porting the experiment code to a different setting. We break the changes down by category: 1. executing script, 2. visualization, 3. simulation. Asterisks indicate that there are multiple options depending on the desired outcome; the minimum is shown.

Task	2D/3D	Backend	Device	Parallelize	Boundary	Precision	Production
Galaxy sim.	1, 0, 0	1, 0, 0	1, 0, 0	1, 0*, 0	n/a	1, 0, 0	1, 0, 0
Bubble opt.	1*, 0, 0	1, 0, 0	1, 0, 0	1, 0*, 0	1, 1, 0	1, 0, 0*	1, 0, 0
Fluid learn.	3, 0, 0	1, 0, 0	1, 0, 0	1, 0*, 0	1, 0, 0	1, 0, 0*	1, 0, 0

Table 3.2: Φ_{Flow} support matrix as of PyTorch 1.11.0, TensorFlow 2.8.0, Jax 0.3.1.

Operating system	PyTorch	TensorFlow 2	Jax	NumPy (CPU)
Linux	✓	Ubuntu 16.04+	Ubuntu 16.04+	✓
Windows	Vista+	7+	10+ (WSL2)	✓
macOS	✓(CPU*)	10.12.6+ (CPU)	12.12+	✓

i.e. a many-body system evolving under Newtonian gravity. In the next experiment, inspired by related work [SC20], we consider a collection of non-overlapping spherical particles or bubbles. We define an energy function based on the pair-wise distances between them and minimize it using the popular L-BFGS-B [LN89] optimizer to obtain a valid configuration of the system. For the third task, we consider the movement of test particles in an incompressible fluid and train a U-Net [RFB15] to exert control forces upon the fluid to move the particles towards the center.

Reusability To measure reusability, we port our experiments to different settings and measure how many lines of code are changed in the process. In particular, we port all problems to two and three-dimensional physical settings, run them with all backends, both on the CPU and GPU and with 32 and 64 bit floating point precision, extend them to batched settings, modify the physical boundary conditions, and switch between production and debug mode (see Tab. 3.1).

We observe that the simulation code is compatible with all tested settings and does not require any modification. The built-in visualization adapts to the dimensionality of the data and only requires manual tweaking for the case of plotting particles on a periodic domain or plotting large batches of data. Remarkably, switching between PyTorch, TensorFlow and Jax only requires a single change to the executing script in all examples despite all tensor operations being different. Specifically, the backend is selected via the import statement, e.g. `phi.torch.flow` for PyTorch and `phi.tf.flow` for TensorFlow. This even holds for the case of training a 3D U-Net interacting with an incompressible fluid simulation because identical U-Net implementations are available for all backends.

Compatibility Φ_{Flow} itself runs on Linux, Windows and macOS. However not all backends are available on all operating systems, e.g. Jax has no official support for Windows. Table 3.2 shows which operating systems are supported by which backends.

Interactivity Φ_{Flow}'s built-in plotting recipes include curve plots for 1D data, heatmap, vector and scatter plots for 2D grids and points, as well as 3D density, scatter and vector plots. Composite plots can be created by overlaying multiple plots or laying out multiple fields in a grid-like structure. Fig. 3.3 shows a selection of basic plots, each visualizing a single field. Additionally, scalar quantities can be graphed over time.

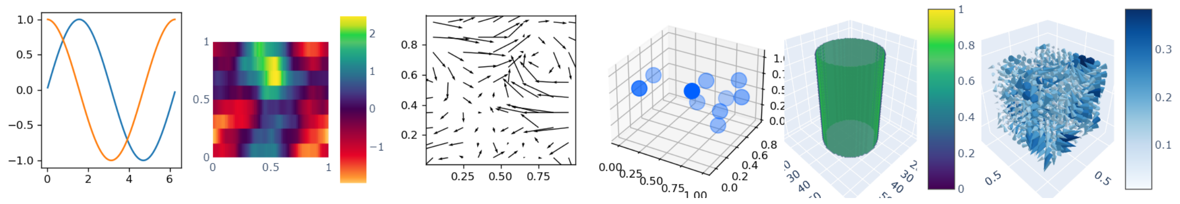


Figure 3.3: Example plots created through plotting recipes included in Φ_{Flow}. Each plot was set up in one line of code.

The interactive user interfaces enable fast iterations without restarting the program each time a change is made. Instead, program and visualization code can be combined into a single interactive application, see Fig. 3.4. This is similar in spirit to Jupyter notebooks which also let users make changes to specific pieces of code without restarting the whole program and Φ_{Flow}'s visualization and interaction tools are fully compatible with Jupyter notebooks. For controlling running applications, Φ_{Flow} currently includes check boxes, sliders, drop-down menus, text fields and buttons. Making a variable or function accessible to the user interface typically requires changing one line of code. Interactive controls make it possible to adjust the time increment of a simulation in real time based on the CFL number or to switch optimizer and change the learning rate for a neural network during training.

Usability Fig. 3.5 compares our Φ_{Flow} implementation of the galaxy simulation against an equivalent Jax version. Implementations for PyTorch, TensorFlow and NumPy strongly resemble the Jax version. While most Jax operations require dimensions being passed as integers, Φ_{Flow} uses human-readable names, such as `stars`, to improve clarity and reduce the chance of mistakes. The implementations of the other two experiments are shown in Figs. 3.6 and 3.7. They are both incredibly short and expressive considering the complex tasks being solved. The loss function for the fluid learning experiment (Fig. 3.7), for example, consists of only twelve lines of Python code, including function declaration and return statement. Within the ten remaining

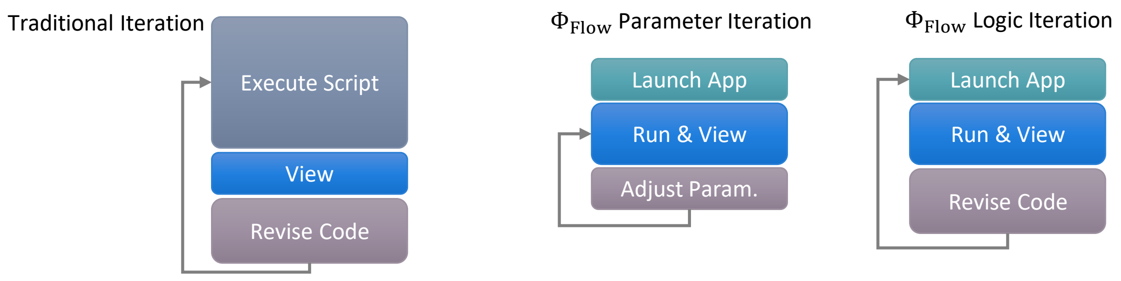


Figure 3.4: A classical workflow requires restarting the program for a parameter change while the interactive workflow in Φ_{Flow} enables parameter and logic changes at runtime.

Jax

```
def galaxy_step(x: jnp.ndarray, v: jnp.ndarray, dt: float):
  dx = jnp.expand_dims(x, -3) - jnp.expand_dims(x, -2)
  denominator = jnp.sum(dx ** 2, -1, keepdims=True) ** 1.5
  individual_a = jnp.expand_dims(masses_others, -1) * dx / denominator
  a = - G * jnp.sum(jnp.where(denominator != 0, individual_a, 0), -2)
  return x + v * dt, v + a * dt
```

Φ_{Flow}

```
def galaxy_step(x: math.Tensor, v: math.Tensor, dt: float):
  dx = x - math.rename_dims(x, 'stars', 'others')
  a = - G * math.sum(math.divide_no_nan(masses_others * dx, math.vec_squared(dx) ** 1.5), 'others')
  return x + v * dt, v + a * dt
```

Figure 3.5: Galaxy simulation function, implemented against Jax and Φ_{Flow}.

lines, it executes the neural network to predict the control force, simulates an incompressible 3D fluid flow for n time steps as well as the passive advection of the marker particles and computes the two loss terms for training.

Φ_{Flow}

```
def energy(x: math.Tensor, boundary=PERIODIC):
  dx = boundary.shortest_distance(x, math.rename_dims(x, 'spheres', 'others'), DOMAIN)
  dr = math.vec_length(dx, eps=1e-8) / (RADII + math.rename_dims(RADII, 'spheres', 'others'))
  return math.l2_loss(math.where((dr < 2e-4) | (dr > 1), 0, 1 - dr))
```

Jax M.D.

Too many lines to list here.

- space.periodic()
  - space.periodic().displacement_fn()
    - space.raw_transform() → space._get_free_indices()
  - space.periodic().shift_fn()
    - space.periodic_shift()
- energy.soft_sphere_pair() → maybe_downcast()
  - energy.soft_sphere() → util.safe_mask()
  - space.canonicalize_displacement_or_metric()
    - space.metric()
    - space.metric().lambda
      - space.distance() → squared_distance(), safe_mask()
  - smap.pair() → 3 nested functions, 8 distinct Jax M.D. function dependencies

Figure 3.6: Top: Energy function for the bubble relaxation experiment, our implementation (Φ_{Flow}), **Bottom**: Jax M.D. function dependency graph for the same task.

The collection of built-in functions further improves readability by replacing common series of operations by their proper name. Among others, Φ_{Flow} includes functions for vector calculus (length, rotation, dot and cross product, normalization, etc.), common training losses (L^1, L^2, frequency-based losses, etc.), finite difference derivatives (gradient, Laplace, curl, divergence) as well as exact operations for periodic domains, grid sampling (up-/down-sampling by factors of 2, arbitrary resampling), diffusion, and advection (Euler, MacCormack, RK4 for both grids and particles). These functions serve as building blocks for users to define complex physical simulations.

```python
def loss_function(v: StaggeredGrid, markers: PointCloud):
  markers_grid = CenteredGrid(markers.grid_scatter(v.bounds, v.resolution, 'clamp'), v.extrapolation)
  force = field.native_call(NET, field.stack([*v.at_centers().vector, markers_grid], channel('inputs')))
  pressure = CenteredGrid(0, 0, force.bounds, force.resolution)
  for t in range(STEPS):
    v += force.time[t]
    v, pressure = fluid.make_incompressible(v, (), Solve('auto', 1e-5, 1e-5, x0=pressure))
    markers = advect.points(markers, v, dt=1, integrator=advect.rk4)
    v = advect.mac_cormack(v, v, dt=1)
  match_loss = math.l2_loss((markers.points - markers.bounds.center))
  force_loss = math.l2_loss(force) * .1
  return match_loss + force_loss, match_loss, force_loss, v, markers
```

Figure 3.7: Implementation of the loss function and simulation for the fluid control training experiment. We set `STEPS=1` for the performance measurements.

Maintainability Φ_{Flow} includes a diagnosis function that can be used to verify the installation and help in finding both software and hardware-related problems. Most notably, it tests both availability and functionality of all relevant dependencies as well as checking for available GPUs by backend.

Φ_{Flow} has been used in production for multiple publications [HKT20; Um+20; HKT21] and we have validated our solvers on a variety of standard fluid flow cases, such as the lid-driven cavity, laminar pipe flow, Taylor-Green vortex decay, and the Backward-facing step problem (see supplemental material). Additionally, Φ_{Flow}'s code base undergoes automated unit tests and coverage reports upon each push. Fig. 3.8 shows the coverage by module as measured by `codecov.io`. Nearly all `math`, `field` and `physics` functions are covered by unit tests, with most untested lines amounting to error handling code. The `vis` package displays the lowest coverage at 64%. While the plotting functionality is tested extensively, there are no automated tests for the user interfaces as of yet.

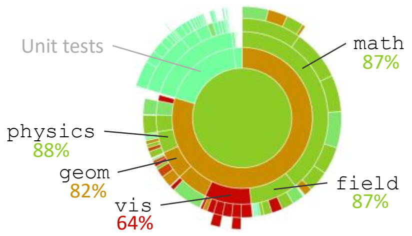


Figure 3.8: Test coverage of Φ_{Flow} by module.

Performance Table 3.3 shows performance measurements for the experiments described above, run with all backends in both production and debug mode. In production mode, the JIT compilation optimizes out all Python-based convenience functionality, such as tracking dimension names, automatic reshaping or data type checking. Debug mode is slower across all backends but we deem this acceptable since typically only a couple of simulation steps need to be performed to find numerical problems or bugs. We observe that Jax yields the best performance in production across all tasks due to its advanced code optimization features. PyTorch is faster than TensorFlow in our particle-based tests but is beaten by TensorFlow in our three-dimensional fluid experiment. In all experiments and with all backends, production mode can be enabled by decorating the simulation or loss function with the `jit_compile` decorator. This simple and unified approach is only

Table 3.3: Mean iteration time (ms) for various tasks, standard deviations captured within one process.

		GPU (GeForce RTX 3090)			CPU (Intel Core i7-7700K)			
Task	Mode	PyTorch	TensorFlow	Jax	PyTorch	TensorFlow	Jax	NumPy
Galaxy sim.	Prod.	18.0 ± 6.0	51.5 ± 15.0	3.6 ± 0.7	$3,853 \pm 90$	$2,204 \pm 44$	657 ± 6	$5,669 \pm 10$
	Debug	27.6 ± 6.3	53.7 ± 14.2	29.1 ± 0.9	$5,418 \pm 38$	$3,103 \pm 9$	$3,146 \pm 30$	
Bubble opt.	Prod.	52.1 ± 11.5	736.0 ± 3.8	28.7 ± 5.9	$3,370 \pm 25$	$3,159 \pm 74$	$1,813 \pm 19$	n/a
	Debug	158.2 ± 0.9	$1,535 \pm 8$	212 ± 6	$3,611 \pm 19$	$3,252 \pm 4$	$5,298 \pm 27$	
Fluid learn.	Prod.	$3,927 \pm 609$	545 ± 21	158.9 ± 3.2	$26,921 \pm 496$	$15,556 \pm 184$	$15,899 \pm 112$	n/a
	Debug	$6,264 \pm 88$	$3,118 \pm 396$	$9,953 \pm 3,558$	$31,940 \pm 629$	$21,202 \pm 336$	$29,249 \pm 1,889$	

possible because Φ_{Flow} includes workarounds for the various limitations in the underlying machine learning libraries.

To check for other sources of computational overhead, we also implement the Galaxy simulation against the native Jax API (Fig. 3.5) and measure (3.8 ± 0.7) ms in production and (19.2 ± 0.8) ms in debug mode. While the variation in measured time between consecutive iterations is relatively large, the overall time of the experiment varies only by about 0.01 ms per step, making the Φ_{Flow} implementation with (3.6 ± 0.7) consistently faster than the Jax version in production, if only slightly. This surprising result is not generally true likely due to Jax being able to better optimize the Φ_{Flow} code in this instance. Comparing the low-level Jax representations of both versions shows that the two versions are indeed different. Despite this, we can conclude that, in production, any additional overhead induced by Φ_{Flow} must be negligible.

3.7 Summary and Outlook

Φ_{Flow} improves upon existing frameworks in three major ways. First, it unifies the APIs of NumPy, PyTorch, TensorFlow and Jax, enabling the development of framework-agnostic code bases. This allows members of a project to collaborate on a shared code base even though they might prefer different machine learning frameworks, as is typical outside large corporations hiring for specific frameworks. Second, Φ_{Flow}'s live visualization makes the framework easy to work with as users can get an intuitive understanding of what their code is doing in real time. This greatly facilitates finding bugs or numerical problems with simulation, optimization and training code. Monitoring a physical quantity over time requires no more than one additional line of code. Third, the live visualization combined with the functionality to make parameters controllable through the user interface at runtime enables much quicker iterations than with traditional frameworks. The instant feedback makes it possible to quickly test different configurations without restarting a program, which simplifies finding optimal settings for simulations and neural network training.

While the performance of Φ_{Flow} already matches backend-specific implementations in production, we aim to further improve performance, e.g. by adding specialized CUDA operators and by supporting preconditioners for linear solves.

We hope that widespread adoption of our framework will lead to more code sharing across simulation as well as machine learning projects. This will facilitate collaborations between different groups and give researchers and developers a broader code base to work with. Since libraries built on top of Φ_{Flow} are automatically compatible with all major Python-based machine learning frameworks, it could also lead developers working on similar libraries for different ecosystems to combine their efforts.

4 Learning Solutions to PDEs

Machine learning techniques for solving partial differential equations (PDEs) encompass a variety of approaches tailored to address the challenges of computational cost, accuracy, and scalability inherent in traditional numerical methods. Applications that rely on precise and fast simulations range from weather and climate modeling to aerodynamics and biomedical simulations [TSM12; Sto+13; RC83; Joh+04]. Despite the effectiveness of numerical methods, computational costs remain a significant challenge due to the high resolutions required for accurate real-world simulations. Deep learning methods have garnered attention for their potential to address components of solutions that are challenging to resolve or not well-captured by traditional physical models [Mor+18; Bar+19; GDY19].

In this chapter, I will first give an overview of various ML-based techniques before delving deeper into three contributions my colleagues and I made to this field.

4.1 Overview of ML approaches to solving PDEs

The property of neural networks to approximate any function has led to a range of approaches to leveraging their flexibility to solve PDEs. These approaches include learning simulation surrogates, where ML models approximate PDE solutions to accelerate computations; learning correctors, which refine numerical solutions by training ML models to correct discrepancies between approximate and true solutions; physics-informed networks (PINNs), which integrate known physical principles into neural network architectures to enforce governing equations and boundary conditions during training; and Lagrangian/Hamiltonian networks, which leverage principles from Lagrangian and Hamiltonian mechanics to model PDE dynamics in a higher-dimensional space using neural networks, as well as various other approaches [Son+22]. These techniques combine the flexibility of ML with the interpretability of physics-based models, offering promising avenues for enhancing the efficiency, accuracy, and scalability of PDE solvers across diverse scientific and engineering domains.

Physics-informed neural networks Physics-informed neural networks (PINNs) integrate principles from both neural networks and physics-based modeling to solve partial differential equations (PDEs) and inverse problems. Traditional methods for solving PDEs often require discretization of the domain, which can be computationally expensive, especially for complex geometries or high-dimensional problems. PINNs offer an alternative approach by leveraging the expressive power of neural networks to approximate the solution of the PDE directly, without discretizing the domain. This is achieved by training the neural network to satisfy both the governing PDE and the boundary/initial conditions, effectively incorporating physical knowledge into the learning process.

PINNs typically consist of two components: a neural network architecture and a loss function that enforces the PDE and boundary/initial conditions. The neural network takes inputs corresponding to spatial or temporal coordinates and outputs the solution of the PDE at those points. The loss function penalizes the deviation of the neural network's predictions from the PDE and boundary/initial conditions. Through iterative optimization techniques such as stochastic gradient descent, the neural network learns to minimize this loss, thereby approximating the solution of the PDE.

Several notable publications have contributed to the development and application of PINNs. Raissi et al. introduced the concept of PINNs in their seminal work [RPK19], demonstrating their effectiveness in solving a variety of PDEs, including Burgers' equation and the Navier-Stokes equations. Subsequent research has extended PINNs to tackle inverse problems, where the PDE is used to infer parameters or initial/boundary conditions from observed data [Rac+20]. Furthermore, efforts have been made to improve the robustness and efficiency of PINNs, such as incorporating adaptive refinement strategies [LMK21] and developing hybrid approaches combining physics-based and data-driven modeling [Zhu+19]. These advancements continue to broaden the scope of applications for PINNs across various scientific and engineering disciplines.

Lagrangian and Hamiltonian networks Lagrangian and Hamiltonian neural networks represent another paradigm in physics-informed machine learning, focusing on conservational laws and symplectic structures. These networks are tailored to emulate the dynamics of physical systems governed by Lagrangian or Hamiltonian mechanics, which are foundational frameworks in classical physics for describing the motion of particles and systems.

Lagrangian neural networks are designed to learn the underlying Lagrangian function of a system directly from data, often in the form of trajectories or observations. The Lagrangian of a dynamical system is defined as $L = T - V$ where T is the total kinetic energy and V is the total potential energy. Using the Euler-Lagrange equations and techniques from variational calculus, one can then derive the evolution of the physical state from the Lagrangian. Consequently, the neural networks allow inference of the equations of motion governing the system's behavior. This approach is particularly useful when explicit equations of motion are unknown or difficult to derive analytically, as it allows for data-driven discovery of the underlying dynamics. Notable work in this area includes methods for learning Lagrangian systems from observed trajectories [Cra+20b].

On the other hand, Hamiltonian neural networks aim to preserve the symplectic structure of Hamiltonian systems, which ensures energy conservation and long-term stability in numerical simulations. These networks are trained to approximate the Hamiltonian function of a system, which describes its total energy $H = T + V$, as well as the corresponding Hamiltonian dynamics. By enforcing symplecticity constraints during training, Hamiltonian neural networks maintain the key properties of Hamiltonian systems, such as conservation of energy and phase-space volume. Research in this field has demonstrated the effectiveness of Hamiltonian neural networks in simulating complex physical systems while preserving important conservation laws [GDY19].

Both Lagrangian and Hamiltonian neural networks offer promising avenues for physics-informed machine learning, providing powerful tools for modeling and simulating a wide range of physical phenomena. By incorporating principles from classical mechanics into neural network architectures, these approaches enable more accurate and interpretable predictions, while also capturing the fundamental conservation laws that govern the behavior of dynamical systems. Ongoing research in this area continues to advance the capabilities of Lagrangian and Hamiltonian neural networks, with applications spanning from fundamental physics to engineering and beyond.

4.2 Learning simulation surrogates

Learning simulation surrogates involve training ML models to approximate the solutions of PDEs based on given input parameters and boundary conditions. This approach aims to accelerate the solution process by replacing computationally expensive simulations with fast predictions from trained ML models. Techniques such as neural networks, Gaussian processes, and kernel methods have been employed for building simulation surrogates.

One such application aims to learn the behavior of liquids using a particle-based (Lagrangian) representation. Under my supervision, Jonathan Klimesch implemented a Lagrangian simulator and used it to train a surrogate neural network with the aim of significantly reducing computation time compared to the simulator. This section summarizes our findings which are available in full on the ArXiv [KHT22].

Previous work suggested that deep learning models can be an efficient and accurate alternative to traditional, hand-crafted simulation approaches [San+20; He+19; Thu+18]. Graph networks in particular have become popular in collider physics [SBV20], astrophysics [Cra+20a] or chemistry [Gil+17] and have been used to build various data-driven simulators [Bat+16; Gil+17; Bat+18; San+18; Li+18].

Sanchez-Gonzales *et al.* proposed the *Graph Network-based Simulators* (GNS) framework, where they approximate fluid dynamics by learned message-passing on particle graphs [San+20]. We replicate their results on data from our custom fluid-Implicit-Particle (FLIP) [BR86; ZB05] simulator written in Φ_{Flow}, and extend the GNS framework with new training variants, investigating the impact of the random noise and the generalization of the trained model to different domains.

Using Φ_{Flow} allows for rolling out multiple simulation steps during training, enabling the formulation of an alternative training scheme where we feed the predicted next simulation state back to the network as an input. The network can thus learn under conditions matching inference time, which makes the noise term superfluous. This methodology requires the use of differentiable physics in order to backpropagate the gradients to previous time steps and would not been feasible without Φ_{Flow}.

To apply the GNS architecture to the FLIP simulation, we first use our simulator to generate training, validation and test data from randomly placed liquid blocks and elongated obstacles.

To quantitatively compare the tested model variants, four metrics are employed. The first metric, MSE-acc 1, measures particle-wise mean-squared error for one-step acceleration predictions. The second metric, MSE 20, averages MSE across time, particles, and spatial dimensions for 20 frames, sampled every 20 steps of 400-step rollouts. This metric provides insights into model performance across different stages of FLIP trajectories. The third metric, MSE 400, is the average MSE across time, particles, and spatial dimensions for full 400-step rollouts.

While MSE-acc 1 offers valuable insights, it can be misleading due to particle permutation sensitivity. Thus, the fourth metric, Earth Mover's Distance (EMD), computed using optimal transport (OT) principles, provides a distributional metric invariant to particle permutations. OT seeks to minimize the cost of transporting one particle distribution to another, employing a cost matrix (M) based on particle-wise distances. The resulting distance, also known as EMD, measures dissimilarity between predicted and ground truth particle distributions. Figure 4.1 depicts error trajectories for MSE 400 and EMD.

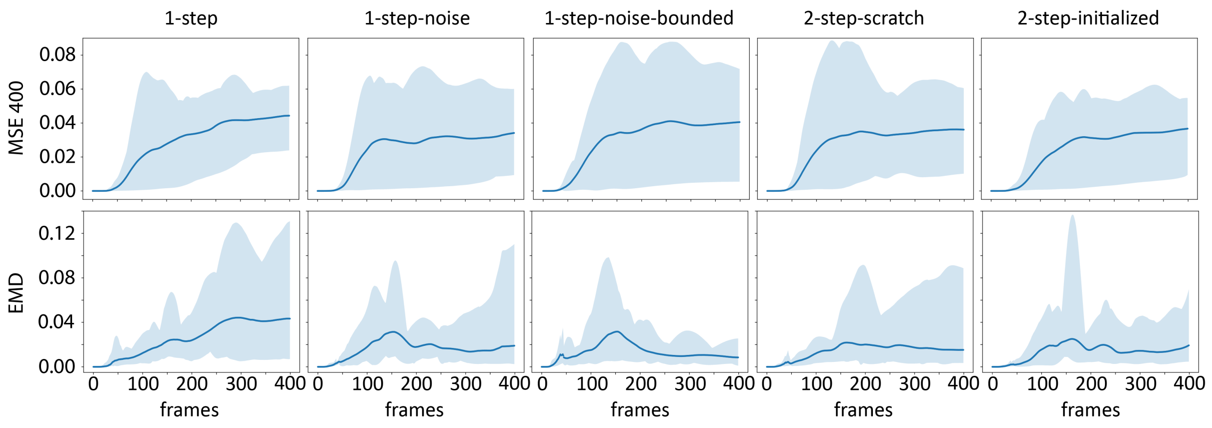


Figure 4.1: Error trajectories of the MSE 400 (MSE averaged over full rollouts) and the EMD metrics for all five model variants. The blue line represents the mean (over the entire data set) and the shaded area indicates the range of possible values. Source: [KHT22]

The 1-step model exhibits superior performance in MSE-acc 1 due to its specialized training for one-step predictions. However, it struggles with EMD, potentially indicating a tendency to maintain particle positions rather than simulating dynamic fluid behavior. Conversely, the 1-step-noise model, challenged by artificial noise during training, achieves better EMD results but lags in MSE-acc 1. The 1-step-noise-bounded model, removing boundary features, excels in EMD while maintaining competitiveness in MSE-acc 1 but suffers in MSE 20 and MSE 400, possibly due to the loss of boundary information.

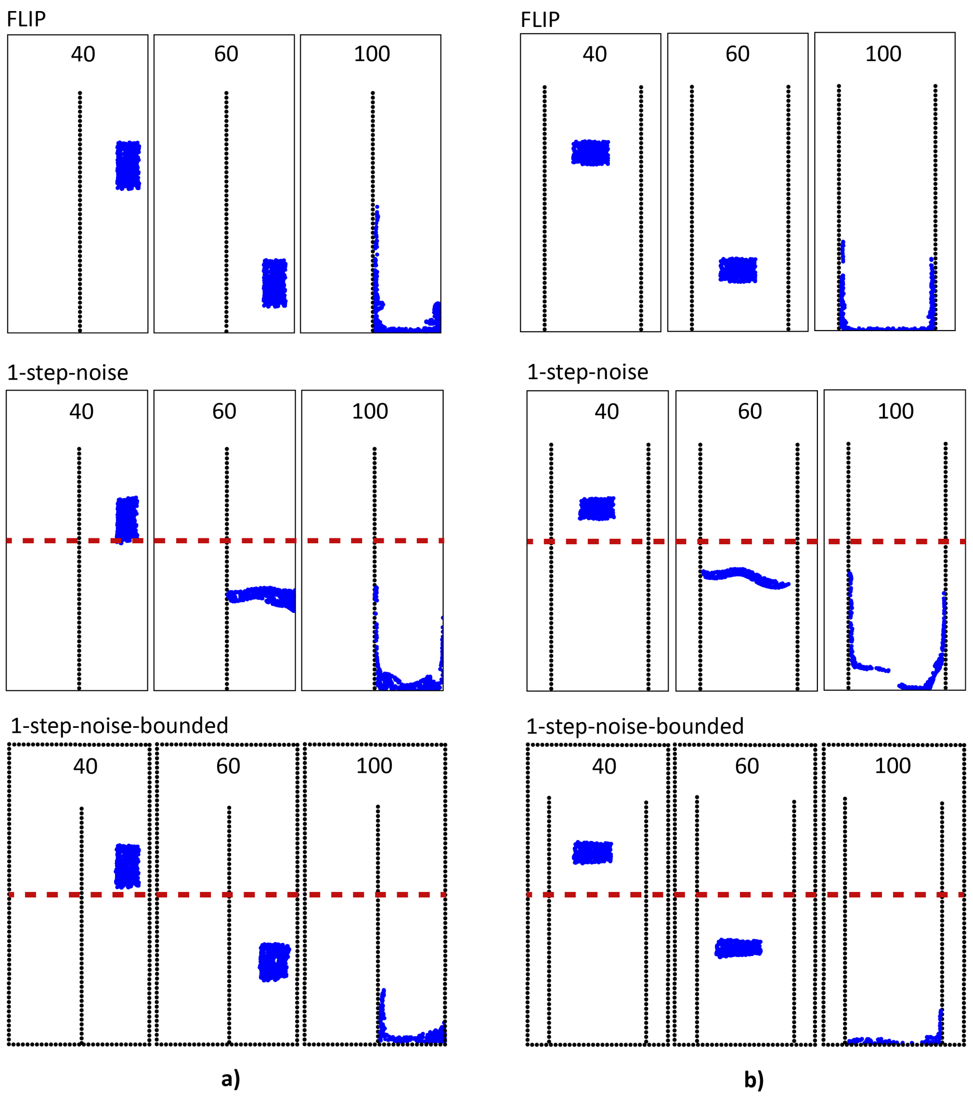


Figure 4.2: Domain generalization experiment. Both left and right examples show ground truth (FLIP), predictions from 1-step-noise and predictions from 1-step-noise-bounded at the indicated frames. The red dashed lines indicate the original domain size. Videos available at `https://git.io/JOOQn`. Source: [KHT22]

.

Among the 2-step variants, the 2-step-initialized model, initialized with weights from the 1-step model, outperforms the 2-step-scratch model across all metrics, showcasing the benefits of initialization. However, both inherit compression tendencies, affecting fluid density during rollouts.

Importantly, training the GNS models with our multi-step loss enables the models to mitigate accumulating errors in simulation rollouts and yields competitive results compared to models trained with the artificial noise proposed by Sanches-Gonzales *et al.* [San+20].

Furthermore, a domain generalization experiment, shown in Figure 4.2, illustrates how the models respond to larger domains. The 1-step-noise-bounded model demonstrates improved behavior compared to the 1-step-noise model, but still exhibits deformations in fluid blocks, suggesting the model's sensitivity to velocity thresholds and boundary distances. Replacing the domain-specific boundary distance features with obstacle boundaries increases the generalization to larger domains. This indicates that the GNS does not learn the true underlying physics but rather problem-specific correlations between input velocities and output accelerations. This is supported by the fact that the architecture takes the previous five velocities as inputs and does not only rely on just the previous velocity as one would expect from physical dynamics. Furthermore, most models are unable to retain the original density of the fluid and compress the fluid particles much more than the FLIP simulator.

Extending the GNS architecture with strong inductive biases towards physical laws and symmetries could improve its physical understanding and force it to learn actual dynamics instead of problem-specific correlations. Future work should also concentrate on examining the physical reasoning of learned simulators in more detail. Transforming parts of learned simulators into symbolic models [Cra+20a; Cra+19] and extending tools like the recently proposed GNNExplainer from Ying *et al.* [Yin+19] could provide further insights into the reasoning of Graph Networks and could yield new ideas on how to improve their physical understanding.

4.3 Learning correctors with differentiable physics

Learning correctors focus on improving the accuracy of existing numerical solvers by training ML models to correct their solutions. These correctors learn the discrepancies between approximate numerical solutions and true solutions of PDEs, thereby enhancing the overall accuracy of the solver. Reinforcement learning, adversarial training, and variational methods are among the techniques utilized for learning correctors.

In this context, Kiwon Um, Robert Brand, Yun Fei, Nils Thuerey and I contributed Solver-in-the-loop [Um+20], which showed that differentiable physics can be used to unroll time sequences in order to improve correction accuracy, similar to the purely surrogate approach discussed above.

Our study targets the identification and mitigation of numerical errors arising from PDE discretization. Our research demonstrates that although closed-form descriptions of discretization errors are lacking, these errors exhibit regular and repeating structures that can be learned by neural networks [Gho96; Arn12]. By training a neural network to recognize and correct these errors, it becomes possible to improve the accuracy of PDE solvers by reducing their numerical error.

Specifically, the study focuses on iterative PDE solving algorithms, highlighting the importance of neural networks interacting with the solver during training to achieve optimal performance [GV12]. Leveraging differentiable simulations enables the trained model to explore and experience the physical environment autonomously, receiving feedback throughout solver iterations [AK17; Tou+18].

Fig. 4.3 shows the effects of correction in a 3D incompressible fluid setting. The low-resolution solver is not accurate enough to resolve the turbulence that develops behind an obstacle. However, in combination with

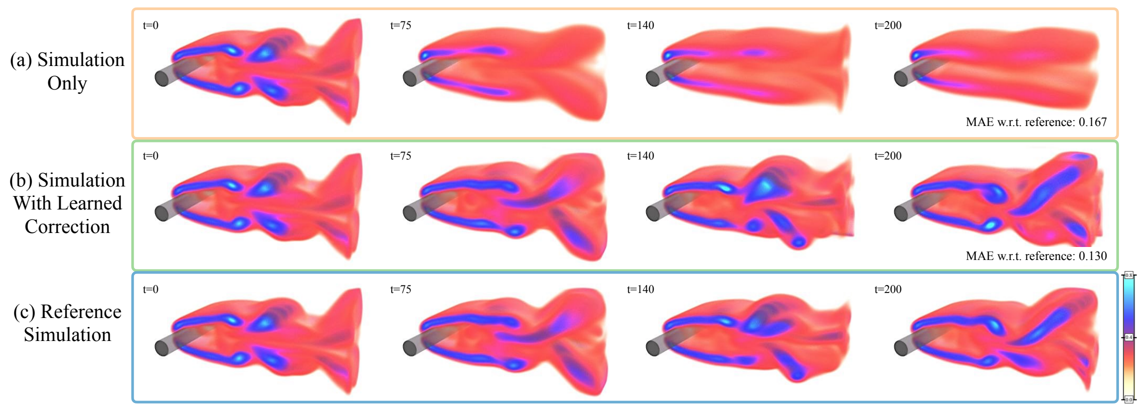

(a) Simulation Only
(b) Simulation With Learned Correction
(c) Reference Simulation
t=0
t=75
t=140
t=200
MAE w.r.t. reference: 0.167
MAE w.r.t. reference: 0.130

Figure 4.3: A 3D fluid problem (shown in terms of vorticity) for which the regular simulation introduces numerical errors that deteriorate the resolved dynamics (a). Combining the same solver with a learned corrector trained via differentiable physics (b) significantly reduces errors w.r.t. the reference (c). Source: [Um+20].

the corrector network, trained on the difference between this low-resolution solver and a high-resolution ground truth solver, the turbulent dynamics is restored at a much lower cost than the high-resolution solver.

Our approach facilitates improved generalization capabilities of the trained models and enhances their accuracy in handling varying physical behaviors encountered during solution processes [Wei17]. Through empirical studies across a range of canonical PDEs and solver types, including explicit, semi-implicit, and implicit solvers, our research demonstrates the effectiveness of the proposed method in improving solution accuracy while handling complex real-world scenarios [MW70].

Our study positions itself within the broader context of machine learning as differentiable programming, emphasizing the recurrent interactions between highly nonlinear PDEs and deep neural networks as a promising avenue for future research [AK17; Tou+18; SC19; Hu+20; Inn+19; HKT20].

4.4 Learning smoothed objective functions

The previous sections considered the problem of finding solutions to PDEs given initial and boundary conditions, i.e. forward problems. However, *inverse* problems also play a pivotal role across scientific and engineering domains, with applications ranging from optimizing fluid dynamics [OBA22] and materials design [Fun+21] to enhancing structural health monitoring [MJU17], manufacturing optimization [Wur+23], and weather prediction [Hua+05].

These problems pose significant challenges due to the inherent complexity of reconstructing inputs from outputs. Traditional methods like Bayesian inference [Sha+15] and quasi-Newton techniques [Rud17; Bro70] often struggle with convergence issues in non-linear landscapes. Recent approaches employing deep neural networks (DNNs) as surrogates show promise but face hurdles such as data limitations, overfitting risks, and generalization difficulties [CDP21; Cao+22; Moh21; Ant+23; GBC16]. My own contributions in the field of inverse problems are given in the next chapter.

However, I supervised Girnar Goyal on a project training proxy networks to predict an *inverse loss*, quantifying deviations from known trajectories induced by sampled parameters [Dhe+22]. Through implicit and explicit regularization techniques, inspired by prior work, our work aims to simplify the complexity of inverse loss landscapes during training. While the ultimate goal of this project is finding better solutions to inverse problems, its machine learning aspect involves surrogate networks for the forward dynamics, similar to the previous projects.

We employ a surrogate neural network, implemented as a deep proxy network, that is trained to approximate the configuration loss function, which quantifies the influence of control parameters on the spatio-temporal evolution of physical states. This network is designed to map pairs of inputs—consisting of the true trajectory of system states, Y^*, and randomly sampled control parameters, X_s—to the corresponding target configuration loss values, $\mathcal{L}(Y^*, X_s)$.

The training process involves minimizing a network training loss, denoted as $\mathcal{L}_N$, using the Adam optimizer. The network's parameters (θ) are adjusted iteratively to minimize the difference between the predicted configuration loss values ($f_\theta(Y^*, X_s)$) and the ground truth configuration loss values ($\mathcal{L}(Y^*, X_s)$). This training loss is calculated as the squared Euclidean distance between the predicted and true configuration loss values. Mathematically, this can be expressed as:

$$\mathcal{L}_N = ||f_\theta(Y^*, X_s) - \mathcal{L}(Y^*, X_s)||^2 \tag{4.1}$$

Here, $f_\theta(Y^*, X_s)$ represents the output of the proxy network given inputs Y^* and X_s, while $\mathcal{L}(Y^*, X_s)$ is the ground truth configuration loss value for the given inputs.

To enhance the network's generalization capability and control its complexity during training, we employ various regularization techniques. These include noise regularization, loss penalty-based regularization, and flatness regularization. Noise regularization involves training the network with noise-labeled inputs, which helps mitigate the impact of high-frequency signals within the target configuration loss function. Loss penalty-based regularization favors low-lying regions of the target inverse landscape, encouraging improved learning of geometric features. Flatness regularization penalizes learning trajectories that traverse steep slopes across the configuration loss surface, promoting smoother mappings and enhanced generalization.

These regularization techniques, with their respective hyperparameters, are incorporated into the training process to achieve reasonable control over the complexity of the proxy network while ensuring effective approximation of the configuration loss function. Fig. 4.4 shows the effects of regularization on a billiards optimization problem, where the task consists of finding the optimal cue ball velocity in order to push another ball towards the target. Due to the chaotic nature of the combined collisions, the loss landscape plotted by angle (blue curve in Fig. 4.4b,d,f) shows discontinuous and chaotic behavior. The proxy network learns to smooth this to yield a differentiable model of the loss function, better suited to optimization tasks.

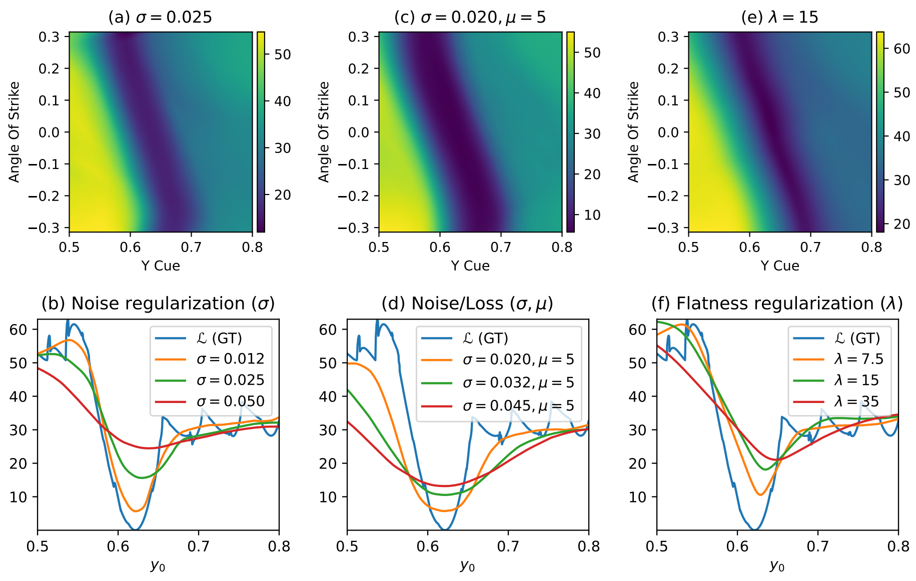


Figure 4.4: Effect of complexity control on proxy network predictions for *configuration loss* $\mathcal{L}$ using (a,b) noise regularization; (c,d) combination of noise and loss-penalty regularization; and (e,f) flatness (gradient-based) regularization in the billiards setup.

Our results demonstrate that complexity-controlled proxy networks enhance convergence in solving complex inverse problems, outperforming traditional methods across scenarios including fluid dynamics, rigid body simulations, and chaotic systems [Pfr+18; RPM20]. This research contributes to a deeper understanding of utilizing DNNs for inverse problems and presents a novel approach to address the challenges associated with their complexity.

The next chapter will dive deeper into the world of inverse problems and how machine learning can be used to solve them.

5 Learning to Solve Inverse Problems

Model parameter estimation through solving inverse problems is a fundamental endeavor in various scientific domains, ranging from the detection of gravitational waves [Tar05; GH18] to the manipulation of plasma flows [Mai+19], and even in the pursuit of detecting neutrinoless double-beta decay [Ago+13; Aal+18]. Iterative optimization techniques, exemplified by methods like limited-memory BFGS [LN89] or Gauss-Newton [GM78], are commonly leveraged to address unconstrained parameter estimation challenges [Pre+07]. These algorithms offer notable advantages, including simplicity, versatility, rapid convergence, and high precision, typically bounded solely by noise inherent in observations and floating-point arithmetic limitations.

5.1 Solving Inverse Physics Problems Jointly

Classical optimizers face several fundamental problems that are rooted in the fact that these algorithms rely on local information, i.e., objective values $L(x_k)$ and derivatives close to the current solution estimate x_k, such as the gradient $\frac{\partial L}{\partial x}|_{x_k}$ and the Hessian matrix $\frac{\partial^2 L}{\partial x^2}|_{x_k}$. Acquiring non-local information can be done in low-dimensional solution spaces, but the curse of dimensionality prevents this approach for high-dimensional problems. These limitations lead to poor performance or failure in settings involving (i) local optima which attract the optimizer in the absence of a counter-acting force. Although using a large step size or adding momentum to the optimizer can help to traverse small local minima, local optimizers are fundamentally unable to avoid this issue. (ii) Flat regions can cause optimizers to become trapped along one or multiple directions. Higher-order solvers can overcome this issue when the Hessian only vanishes proportionally with the gradient, but all local optimizers struggle in zero-gradient regions. (iii) Chaotic regions, characterized by rapidly changing gradients, are extremely hard to optimize. Iterative optimizers typically decrease their step size to compensate, which prevents the optimization from progressing on larger scales.

In many practical cases, a *set* of observations is available, comprising many individual parameter estimation problems, e.g., when repeating experiments multiple times or collecting data over a time frame [Car+19; Del+18; GH18; Ago+13; MAL13] and, even in the absence of many recorded samples, synthetic data can be generated to supplement the data set. In these cases, information from multiple examples can be pooled by jointly optimizing a data set of inverse problems.

Neural networks have become popular tools to model physical processes, either completely replacing physics solvers [Kim+19; San+20; Sta+21; Pat+22] or improving them [Tom+17; Um+20; Koc+21a]. This can improve performance since network evaluations and solvers may be run at lower resolution while maintaining stability and accuracy. Additionally, it automatically yields a differentiable forward process which can then be used to solve inverse problems [SF18; RPM20; All+22], similar to how style transfer optimizes images [GEB16].

Neural networks have also been used as regularizers to solve inverse tomography problems [Muk+21; Li+20] and employed recurrently for image denoising and super-resolution [PW17]. Recent works have also explored them for predicting solutions to inverse problems [Shm+23; Luc+18; HKT22; SHT21] or aiding in finding solutions [Kha+17; Dai+21; AMJ18]. In this context, the inductive bias of convolutional networks has been shown to benefit the optimization of individual inverse problems with image solution spaces [UVL18; HSG19].

Underlying many of these approaches are differentiable simulations, required to obtain gradients of the inverse problem. These can be used in iterative optimization or to train neural networks. Many recent software packages have demonstrated this use of differentiable simulations, with general frameworks [Hu+20; SC20; HKT20] and specialized simulators [Tak+21; LLK19].

Physics-informed neural networks [RPK19] encode solutions to optimization problems in the network weights themselves. They model a continuous solution to an ODE or PDE and are trained by formulating a loss function based on the differential equation, and have been explored for a variety of directions [Yan+19; Lu+21; Kri+21]. However, as these approaches rely on loss terms formulated with neural network derivatives, they do not apply to general inverse problems.

The training process of neural networks themselves can also be framed as an inverse problem, and employing learning models to aid this optimization is referred to as *meta-learning* [VD02]. However, due to the large differences, meta-learning algorithms strongly differ from methods employed for inverse problems in physics.

In this context, I investigated the question *Can better solutions x_i to general inverse problems be found by optimizing them jointly instead of individually, without requiring additional information about the problems?*

To answer this question, we employ neural networks to formulate a joint optimization problem. Neural networks as general function approximators are a natural way to enable joint optimization of multiple a priori independent examples. They have been extensively used in the field of machine learning [GBC16], and a large number of network architectures have been developed, from multilayer perceptrons (MLPs) [Hay94] to convolutional networks (CNNs) [KSH12] to transformers [Vas+17]. Overparameterized neural network architectures typically smoothly interpolate the training data [BHM18; BPL21], allowing them to generalize, i.e., make predictions about data the network was not trained on. We aim to leverage this *inductive bias* to improve the convergence on general inverse problems. In particular, we propose using the training process of a neural network as a drop-in component for traditional optimizers like BFGS without requiring additional data, configuration, or tuning. Instead of making predictions about new data after training, our objective is to solve a set of given inverse problems. The so-found solutions can also be combined with an iterative optimizers to improve accuracy. Unlike related machine learning applications [Kim+19; San+20; Sta+21; SHT21; HKT22; SF18; RPM20; All+22], where a significant goal is accelerating time-intensive computations, we accept a higher computational demand if the resulting solutions are more accurate.

5.2 Predictor-corrector-based PDE Control

Intelligent systems operating in the physical domain necessitate capabilities to perceive, predict, and interact with physical phenomena [BHT13]. Historically, optimal control problems concerning PDEs have been tackled through iterative optimization techniques. Differentiable solvers coupled with the adjoint method have facilitated efficient optimization for high-dimensional systems [Tou+18; Avi+18; SF18]. However, direct optimization, particularly through gradient descent (single shooting) during test time, poses resource-intensive challenges and may not be readily deployable in interactive settings. Although more sophisticated methods like multiple shooting and collocation exist, they often rely on restrictive modeling assumptions, limiting their generalizability and necessitating computationally intensive iterative optimization during testing.

The computational expense of iterative optimization methods stems from the need to iteratively refine solutions from scratch, often requiring a significant number of iterations to converge to an optimal solution. Real-world control problems frequently demand rapid decision-making in specialized environments, with reaction times constrained to fractions of a second. Thus, there is a compelling rationale for leveraging data-driven models, such as deep neural networks, which offer swift inference times while possessing the capacity to construct internal representations of the environment.

Previous research has explored various approaches to enhance the solution of PDE problems or utilize PDE formulations for unsupervised optimization [Lon+17; Bar+19; Hsi+19; RYK18]. Techniques such as regression forests, graph neural networks, continuous convolutions, and MLPs have been employed to tackle challenges in Lagrangian fluid simulation and turbulence modeling [Lad+15; Mro+18; Li+18; LKT16; Tom+17; Xie+18; Mor+18]. Additionally, deep learning methodologies have been applied to predict chemical properties and outcomes of chemical reactions [Gil+17; Bra+19].

Differentiable solvers have demonstrated utility across various domains, including rigid body mechanics [Deg+19; Avi+18], manipulation planning [Tou+18], protein structure inference [Ing+19], liquid interaction [SF18], soft robot control [Hu+19], and inverse problems involving cloth [LLK19]. These works leverage the automatic differentiation capabilities of deep learning frameworks. Notably, while prior efforts have predominantly focused on Lagrangian solvers, this study addresses grid-based solvers, which are well-suited for modeling dense, volumetric phenomena.

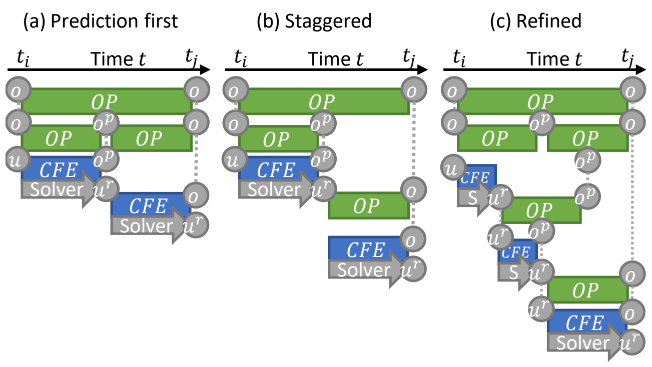


Figure 5.1: Overview of three different predictor-corrector execution schemes. OP denotes observation predictor and CFE denotes control force estimator, i.e. the corrector. Source: [HKT20].

In collaboration with Vladlen Koltun from Intel, I introduced a novel deep learning approach that significantly outperforms iterative optimization techniques in terms of speed and efficiency. Our method revolves around a hierarchical predictor-corrector scheme, which breaks down the problem temporally into manageable sub-problems.

Fig. 5.1 shows three types of predictor-corrector execution schemes we considered. The first (a) initially runs the predictor networks to generate a full trajectory, i.e. a plan. Then, the corrector network is used at each time step to nudge the simulation towards the predicted plan. The second version (b) delays later predictions and performs simulation steps immediately as soon as possible. This allows later predictions to take the actual simulated trajectory into account, which leads to improved performance. The third version (c) goes one step further by performing additional predictions after each simulated step. This allows predictions to always be as close to the simulation as possible, which lead to the best accuracy of the three.

By harnessing models tailored to different time scales, we can effectively control prolonged sequences of intricate, high-dimensional systems. Training our models involves leveraging a differentiable PDE solver, providing the agent with real-time feedback on the consequences of its interactions at any given moment. Consequently, our models acquire the ability to navigate solution manifolds containing myriad solutions, thus evading the local minima pitfalls inherent in classical optimization methods.

We validated our approach through extensive evaluations across a spectrum of control tasks within systems governed by advection-diffusion PDEs, such as the Navier-Stokes equations. Our quantitative assessments focus on the fidelity of the resultant sequences in approximating the target state and the exerted force on the physical system. Notably, our method demonstrates stable control over significantly extended time spans compared to existing alternatives.

This work was the first to use Φ_{Flow} (chapter 3.3) and critically influenced its early development.

Our methodology demonstrated the feasibility of acquiring representations of solution manifolds for optimal control trajectories through data-driven means. Rapid inference remains crucial in time-sensitive applications and can complement classical solvers to expedite convergence, thereby enhancing solution quality.

5.3 The problem with Gradient Descent

In the previously introduced work, the reliance on machine learning techniques, particularly deep neural networks, for parameter estimation from observations has become prevalent [Car+19; Del+18; GH18; Ago+13]. Despite machine learning methods typically not achieving solutions up to machine precision, they offer several advantages over iterative solvers. Firstly, their computational overhead for inferring solutions tends to be substantially lower compared to iterative methods, a critical factor in time-sensitive applications like the quest for rare events within vast datasets typical in collider physics. Secondly, learning-based approaches eliminate the need for an initial guess to commence problem-solving, mitigating the risk of poor initial guesses hindering convergence to a global optimum or inducing divergence. Thirdly, these methods exhibit reduced susceptibility to getting trapped in local optima compared to iterative techniques, owing to parameter sharing across diverse problem instances [HKT20]. The stochastic nature of the training process further aids in escaping local optima, as gradients from other problems can guide predictions away from the basin of attraction of such local optima.

Despite the significant strides in deep learning, the fundamental limitations of gradient descent endure, especially pronounced when optimizing non-linear functions such as those governing physical systems. Notably, state-of-the-art neural networks typically rely solely on first-order information due to the computational challenges associated with evaluating the Hessian with respect to network parameters. In scenarios involving non-linear functions, gradient magnitudes often exhibit considerable variation across examples and parameters, exacerbating the limitations of gradient descent methods.

This is especially true when the problems being solved are ill-conditioned, as this leads to gradients which do not properly align with a stable descent direction but only with the short-term steepest descent. Fig. 5.2 shows an example of this for a simple function. In the initial gradient descent updates, the x_2 direction overshadows the x_1 direction, leading to sub-optimal optimization steps. Newton's method performs much better in these situations, as is uses higher-order information in the form of the inverse Hessian matrix H^{-1}, not just first-order information like gradient descent.

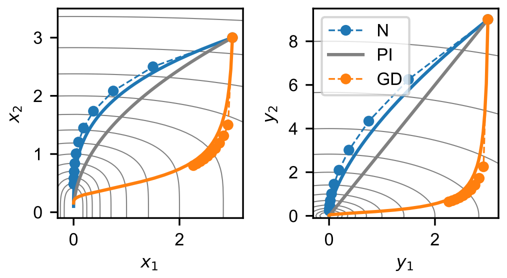


Figure 5.2: Minimization with $y \equiv \mathcal{P}(x) = (x_1, x_2^2)$. L contours in gray. Trajectories of gradient descent (GD), Newton's method (N), and perfect physics inversion (PI) shown as lines (infinitesimal η) and circles (10 iterations with constant η). Source: [HKT22].

To mitigate the issue with first-order updates, momentum has been introduced into first-order optimizers like stochastic gradient descent (SGD) and Adam [KB14], but this only moderates the issue and does not eliminate it. In highly ill-conditioned settings, none of the traditional neural network optimizers perform well.

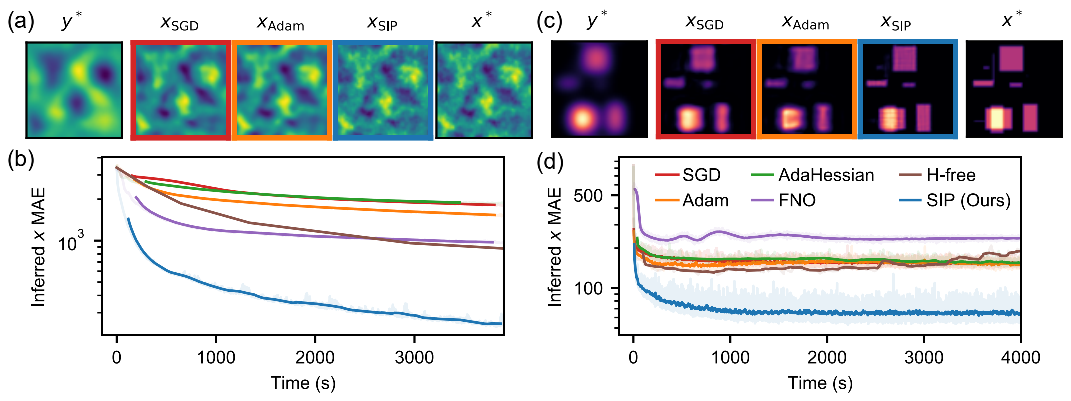


Figure 5.3: Poisson's equation (left) and the heat equation (right). (a,c) Example from the data set: observed distribution (y^*), inferred solutions, and ground truth solution (x^*). (b,d) Learning curves, running average over 64 mini-batches (except for H-free). Source: [HKT22].

5.4 Learning with Physics inversion

In order to overcome the problems caused by first-order updates, I introduced a novel hybrid training scheme that integrates inverse physics solvers into the traditional deep learning pipeline. This approach aims to combine the rapid convergence of higher-order solvers like BFGS [Fle00] with the computational efficiency of backpropagation for network training [Goo16].

Instead of employing the adjoint method for backpropagation through the physical process, we replace the regular gradient with updates computed from higher-order solvers, capturing the local nonlinear behavior of the physics. This can be done e.g. by using to Hessian matrix H in addition to the Jacobian J to compute Newton updates $-\eta H^{-1} J$ [GST07], where η is the step size.

These physics updates are then integrated into a traditional neural network optimizer, facilitating weight updates via backpropagation. Our method ensures compatibility with existing acceleration and stabilization techniques developed for training deep learning models. We present theoretical insights and conduct an extensive empirical evaluation across various inverse problems, including the challenging Navier-Stokes equations.

Fig. 5.3 shows the network optimization curves, i.e. the error evolution over training time, for two very common scientific problems. The first is the Poisson problem (a,b) which is essential to electrostatics, Newtonian gravity, and fluid dynamics [Ame14]. The second (c,d) is a diffusion problem which finds application in almost every field of science, such as physics, chemistry, biology, medicine, environmental science, and engineering [Bir02].

Our results demonstrate that leveraging higher-order or domain-specific solvers significantly enhances convergence speed and solution quality without necessitating the evaluation of the Hessian with respect to model parameters. We introduce Scale-Invariant Physics (SIP) training, which utilizes physics inversion to compute scale-invariant updates in the solution space.

SIP training exhibits provable convergence under conditions of sufficient network updates per solver evaluation and demonstrates convergence with a single update for a broad spectrum of physics experiments. The scale-invariance property enables exponential acceleration in finding solutions for numerous physics problems while maintaining relatively low computational costs.

The idea of inversion to combat ill-condition problems spawned a similar avenue of research, which my colleague Patrick Schnell and I investigated.

5.5 Learning with Gradient inversion

From a mathematical perspective, training a neural network using a physical loss function poses challenges inherent to both network training and physics optimization. Effective handling of flat regions within optimization landscapes is crucial for obtaining satisfactory results. In conventional learning tasks, complex loss landscapes are navigated using gradient-based optimizers with data-driven normalization techniques like Adam [KB15]. Conversely, in physics optimization, higher-order methods such as Newton's method [GM78] are favored, leveraging inversion processes. However, [HKT21] discovered that these approaches struggle to effectively manage joint optimization of network and physics. Gradient-descent-based optimizers encounter issues like vanishing or exploding gradients, hindering convergence, while higher-order methods often struggle to scale to the high-dimensional parameter spaces typical in deep learning [GBC16].

Motivated by the pivotal role of inversion in physics problems highlighted by [HKT21], we adopt an inversion-based strategy. We propose a novel approach for simultaneous optimization of physics and network parameters, termed *half-inverse gradients*. At its core, this method relies on partial matrix inversion derived from the interplay between network and physics, both formally and geometrically. A key advantage of our approach is its linear scalability with the number of network parameters. To showcase its broad applicability and practical utility, we demonstrate significant enhancements in convergence speed and final loss values compared to existing methods. These improvements are assessed in terms of both absolute accuracy and computational efficiency. We conduct evaluations across diverse physical systems, including the Schrödinger equation, a nonlinear chain system, and the Poisson problem.

In conclusion, our study challenges the prevailing approach of utilizing standard gradient-based network optimizers for training neural networks coupled with physical solvers. Through an in-depth analysis of the transition between gradient descent and Gauss-Newton's method, our novel approach, coined Half-Inverse Gradients (HIGs), efficiently learns physics modes without imposing undue strain on the network via large weight updates. This results in expedited and more precise minimization of the learning objective, as demonstrated across a diverse array of experiments. We envision our work as a catalyst for further investigations into enhanced learning methodologies tailored for addressing physical problems. Promising directions for future research include devising efficient techniques for half-inverting the Jacobian matrix and extending the application of HIGs to physical systems characterized by chaotic behavior or more intricate training setups [BHT13; Umm+20; Pfa+20].

As the inclusion of a physics solver significantly alters the gradient flow, our study underscores the necessity of reevaluating conventional practices in network optimization for such scenarios. By devising a method that seamlessly transitions from gradient descent to the Gauss-Newton algorithm, we offer a pathway to better tackle the joint optimization problem. Compared to existing optimizers, our approach enables swifter and more accurate convergence of learning objectives [KB15; GBC16]. We anticipate that our findings will serve as a springboard for further exploration of advanced learning techniques tailored to address the nuances of physical problems. Particularly intriguing avenues for future investigation involve developing efficient strategies for inverting the Jacobian matrix within the context of HIGs and integrating unstructured solvers into the framework [Zie+77]. The amalgamation of finite elements with graph-neural networks offers a promising foundation for future applications of HIGs, especially in complex and dynamic physical systems [BHT13; Umm+20; Pfa+20].

6 Conclusion and Future Directions

In this concluding chapter, I reflect on the progress made in leveraging machine learning (ML) techniques for addressing forward and inverse problems through the paradigm of differentiable simulations. This innovative approach has garnered significant attention across various domains, offering promising avenues for enhancing computational efficiency, accuracy, and flexibility in modeling complex systems.

State-of-the-Art Summary The integration of differentiable simulations with ML methodologies has led to notable breakthroughs in several scientific and engineering disciplines. Researchers have successfully employed neural networks, particularly deep learning architectures, to learn and optimize simulation models directly from observational data. This coupling allows for the seamless incorporation of domain knowledge into ML models while circumventing the need for explicit analytical expressions, which are often elusive or computationally expensive in complex systems.

In the realm of forward problems, where the objective is to predict system behavior given input parameters, differentiable simulations coupled with ML have demonstrated remarkable predictive capabilities. By training neural networks on simulated data generated from underlying physical models, researchers have achieved high-fidelity predictions across diverse scenarios, ranging from fluid dynamics and materials science to climate modeling and biological systems.

Similarly, in inverse problems, where the goal is to infer model parameters or underlying system properties from observed data, the fusion of differentiable simulations with ML techniques has revolutionized traditional approaches. By formulating the inverse problem as an optimization task within a differentiable framework, researchers have devised novel algorithms capable of efficiently estimating model parameters while accommodating uncertainties and constraints, thereby enabling robust parameter inference and model calibration.

Moreover, the synergy between differentiable simulations and ML has facilitated the exploration of novel research avenues, such as model discovery, uncertainty quantification, and decision-making under uncertainty. By harnessing the power of data-driven approaches, scientists can uncover hidden patterns, discover emergent phenomena, and enhance our understanding of complex systems, ultimately accelerating scientific discovery and technological innovation.

Outlook and Future Directions Looking ahead, the field of machine learning for solving forward and inverse problems is poised for continued growth and innovation. Several promising directions merit attention:

1. **Hybrid Modeling Approaches:** Future research will likely focus on integrating traditional physics-based models with data-driven techniques, leveraging the strengths of both approaches to enhance predictive accuracy and generalization capabilities. Our Solver-in-the-loop publication [Um+20] proves this point, and various other research is following in this vein [Koc+21b; Hei+21].
2. **Uncertainty Quantification and Robustness:** Addressing uncertainties inherent in simulation models and observational data remains a critical challenge, with current focusing on Bayesian analyses [HA15; And+23]. Future efforts will aim to develop robust uncertainty quantification techniques to improve the reliability and trustworthiness of ML-based predictions and inferences.
3. **Scaling with Increasing Compute Capability:** With hardware performance increasing exponentially, methods that are infeasible today due to performance constraints will become more relevant in the future. This includes compute-intensive architectures like diffusion models [Yan+23] and the attention mechanism [Vas+17] whose $\mathcal{O}(N^2)$ compute scaling limits possible sequence lengths today.
4. **Interdisciplinary Applications:** The applicability of differentiable simulations and ML spans numerous domains, including physics, engineering, biology, finance, and beyond. Interdisciplinary collaborations will be pivotal in tackling complex real-world problems and driving innovation across diverse fields.
5. **Ethical and Societal Implications:** As ML-powered simulations become increasingly prevalent in decision-making processes, such as in finance, attention to ethical considerations, transparency, and accountability will be paramount. Research efforts will need to address potential biases, fairness concerns, and ethical implications associated with the deployment of ML models in critical domains.
6. **Continual Learning and Adaptive Systems:** Building dynamic, adaptive systems capable of continual learning and adaptation to evolving environments represents a frontier for research. By incorporating mechanisms for self-improvement and autonomous decision-making, such systems can exhibit enhanced resilience and performance in real-world settings.

In conclusion, the integration of machine learning with differentiable simulations holds tremendous promise for advancing our understanding of complex systems, solving challenging forward and inverse problems, and driving innovation across various domains. By embracing interdisciplinary collaboration, fostering ethical practices, and pursuing novel research directions, we can unlock the full potential of this synergistic approach and pave the way for transformative advancements in science, engineering, and beyond.

Published Research Papers and My Contributions

Φ-ML: Intuitive Scientific Computing with Dimension Types for Jax, PyTorch, TensorFlow & NumPy

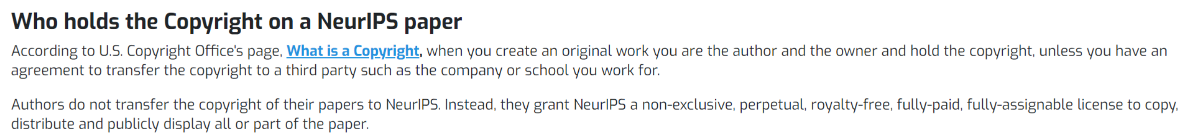
Abstract of Paper Φ_{ML} is a math and neural network library designed for science applications. It enables users to quickly evaluate many network architectures on their data sets, perform (sparse) linear and non-linear optimization, and write differentiable simulations that scale to n dimensions. Φ_{ML} is compatible with Jax, PyTorch, TensorFlow and NumPy, and user code can be executed on all of these backends. The project is hosted at `https://github.com/tum-pbs/PhiML` under the MIT license.

Author Contributions I developed and published the software and wrote the manuscript. Various students, fellow Ph.D. candidates, and external researchers contributed minor features and bug fixes to the project and helped designing the software architecture. Nils Thuerey proofread the manuscript and helped testing the code.

Scale-invariant Learning by Physics Inversion

Abstract of Paper Solving inverse problems, such as parameter estimation and optimal control, is a vital part of science. Many experiments repeatedly collect data and rely on machine learning algorithms to quickly infer solutions to the associated inverse problems. We find that state-of-the-art training techniques are not well-suited to many problems that involve physical processes. The highly nonlinear behavior, common in physical processes, results in strongly varying gradients that lead first-order optimizers like SGD or Adam to compute suboptimal optimization directions. We propose a novel hybrid training approach that combines higher-order optimization methods with machine learning techniques. We take updates from a scale-invariant inverse problem solver and embed them into the gradient-descent-based learning pipeline, replacing the regular gradient of the physical process. We demonstrate the capabilities of our method on a variety of canonical physical systems, showing that it yields significant improvements on a wide range of optimization and learning problems.

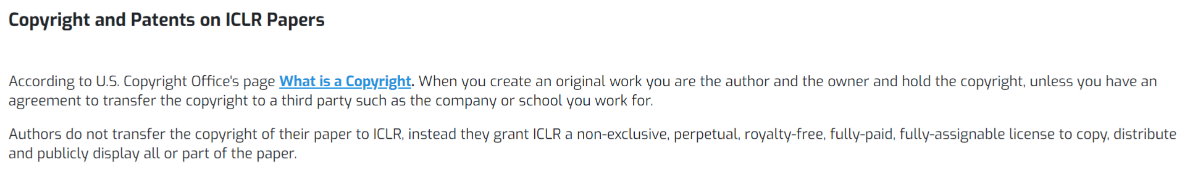
Author Contributions I performed the experiments, generated the data, trained the neural networks, analyzed the results, plotted all diagrams and drafted the manuscript. Nils Thuerey and Vladlen Koltun took on a supervisory role in this project, giving me continuous feedback on the ongoing work and proofreading the manuscript.



Learning to Control PDEs with Differentiable Physics

Abstract of Paper Predicting outcomes and planning interactions with the physical world are long-standing goals for machine learning. A variety of such tasks involves continuous physical systems, which can be described by partial differential equations (PDEs) with many degrees of freedom. Existing methods that aim to control the dynamics of such systems are typically limited to relatively short time frames or a small number of interaction parameters. We present a novel hierarchical predictor-corrector scheme which enables neural networks to learn to understand and control complex nonlinear physical systems over long time frames. We propose to split the problem into two distinct tasks: planning and control. To this end, we introduce a predictor network that plans optimal trajectories and a control network that infers the corresponding control parameters. Both stages are trained end-to-end using a differentiable PDE solver. We demonstrate that our method successfully develops an understanding of complex physical systems and learns to control them for tasks involving PDEs such as the incompressible Navier-Stokes equations.

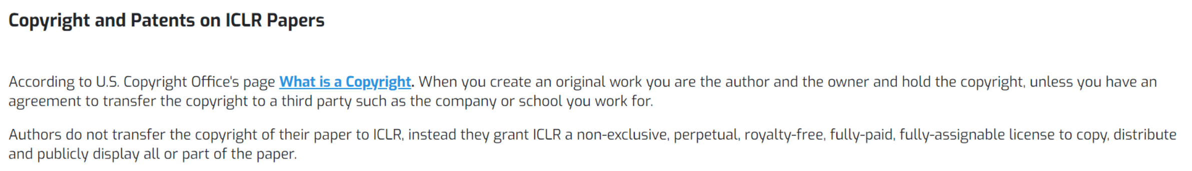
Author Contributions I performed the experiments, generated the data, trained the neural networks, analyzed the results, plotted all diagrams and drafted the manuscript. Nils Thuerey and Vladlen Koltun took on a supervisory role in this project, giving me continuous feedback on the ongoing work and proofreading the manuscript.



Half-Inverse Gradients for Physical Deep Learning

Abstract of Paper Recent works in deep learning have shown that integrating differentiable physics simulators into the training process can greatly improve the quality of results. Although this combination represents a more complex optimization task than supervised neural network training, the same gradient-based optimizers are typically employed to minimize the loss function. However, the integrated physics solvers have a profound effect on the gradient flow as manipulating scales in magnitude and direction is an inherent property of many physical processes. Consequently, the gradient flow is often highly unbalanced and creates an environment in which existing gradient-based optimizers perform poorly. In this work, we analyze the characteristics of both physical and neural network optimizations to derive a new method that does not suffer from this phenomenon. Our method is based on a half-inversion of the Jacobian and combines principles of both classical network and physics optimizers to solve the combined optimization task. Compared to state-of-the-art neural network optimizers, our method converges more quickly and yields better solutions, which we demonstrate on three complex learning problems involving nonlinear oscillators, the Schrödinger equation and the Poisson problem.

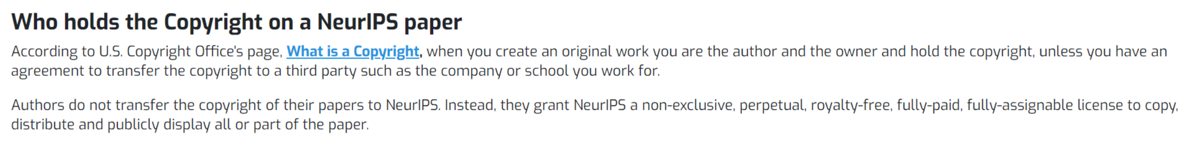
Author Contributions I performed one of the experiments, including data generation, network training, analysis and visualization. Additionally, I proof-read the manuscript, contributing significantly to the final text. Patrick Schnell performed the other experiments and drafted the main text. Nils Thuerey supervised the project and helped with coordination.



Solver-in-the-Loop: Learning from Differentiable Physics to Interact with Iterative PDE-Solvers

Abstract of Paper Finding accurate solutions to partial differential equations (PDEs) is a crucial task in all scientific and engineering disciplines. It has recently been shown that machine learning methods can improve the solution accuracy by correcting for effects not captured by the discretized PDE. We target the problem of reducing numerical errors of iterative PDE solvers and compare different learning approaches for finding complex correction functions. We find that previously used learning approaches are significantly outperformed by methods that integrate the solver into the training loop and thereby allow the model to interact with the PDE during training. This provides the model with realistic input distributions that take previous corrections into account, yielding improvements in accuracy with stable rollouts of several hundred recurrent evaluation steps and surpassing even tailored supervised variants. We highlight the performance of the differentiable physics networks for a wide variety of PDEs, from non-linear advection-diffusion systems to three-dimensional Navier-Stokes flows.

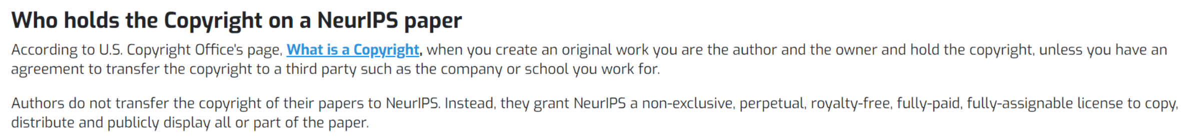
Author Contributions Kiwon Um performed most of the experiments and drafted the manuscript. I helped performing the experiments by providing the differentiable physics software and supervised Robert Brand, a master student who contributed experimental data. Additionally, I helped in creating the figures and proof-read the manuscript. Nils Thuerey supervised the project and provided experimental data.



phiflow: A differentiable pde solving framework for deep learning via physical simulations

Summary This workshop paper introduces differentiable physics and the first public version of Φ_{Flow}. It shows how differentiable physics can be used to train networks to interact with physics solvers and summarizes experimental results from our previous control and correction projects.

Author Contributions I developed the software and contributed the control results. Kiwon contributed the correction results. Nils Thuerey drafted the manuscript.

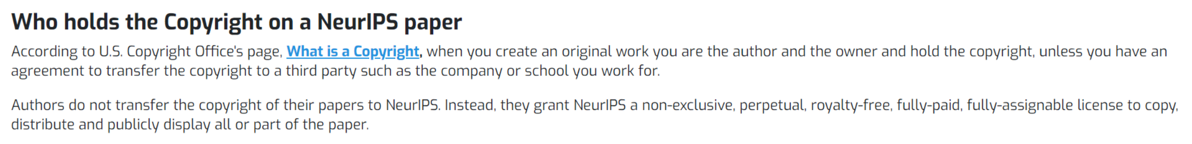


Copyright of Included Papers

This chapter is included for formal reasons.

The publication *Φ-ML: Intuitive Scientific Computing with Dimension Types for Jax, PyTorch, TensorFlow & NumPy* is available from the Journal of Open Source Software (JOSS) under the Creative Commons Attribution 4.0 International License, see next page.

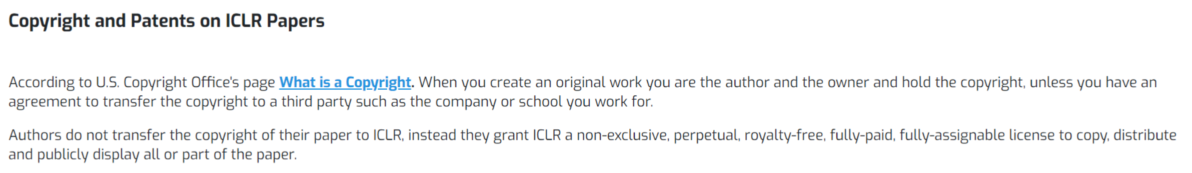
All other papers included in this document were published at machine learning conferences which are granted a non-exclusive, perpetual, royalty-free, fully-paid, fully-assignable license to copy, distribute and publicly display all or part of the paper. The following copyright notices are available on their respective websites, `https://neurips.cc/FAQ/Copyright` and `https://iclr.cc/FAQ/Copyright`.



Who holds the Copyright on a NeurIPS paper

According to U.S. Copyright Office's page, **What is a Copyright**, when you create an original work you are the author and the owner and hold the copyright, unless you have an agreement to transfer the copyright to a third party such as the company or school you work for.

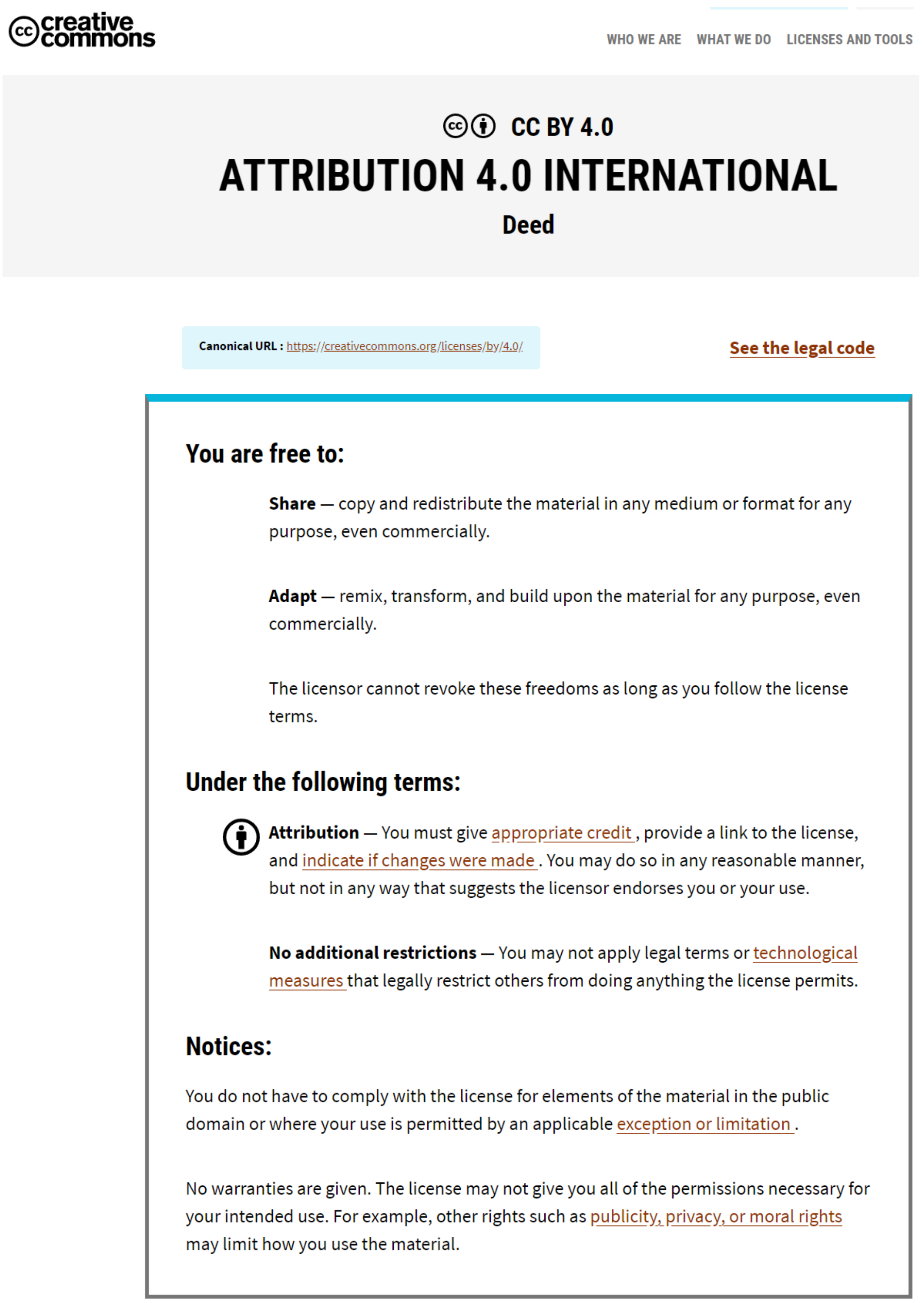
Authors do not transfer the copyright of their papers to NeurIPS. Instead, they grant NeurIPS a non-exclusive, perpetual, royalty-free, fully-paid, fully-assignable license to copy, distribute and publicly display all or part of the paper.



Copyright and Patents on ICLR Papers

According to U.S. Copyright Office's page **What is a Copyright**. When you create an original work you are the author and the owner and hold the copyright, unless you have an agreement to transfer the copyright to a third party such as the company or school you work for.

Authors do not transfer the copyright of their paper to ICLR, instead they grant ICLR a non-exclusive, perpetual, royalty-free, fully-paid, fully-assignable license to copy, distribute and publicly display all or part of the paper.



CC BY 4.0

ATTRIBUTION 4.0 INTERNATIONAL

Deed

Canonical URL : https://creativecommons.org/licenses/by/4.0/

See the legal code

You are free to:

Share — copy and redistribute the material in any medium or format for any purpose, even commercially.

Adapt — remix, transform, and build upon the material for any purpose, even commercially.

The licensor cannot revoke these freedoms as long as you follow the license terms.

Under the following terms:

Attribution — You must give appropriate credit , provide a link to the license, and indicate if changes were made . You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use.

No additional restrictions — You may not apply legal terms or technological measures that legally restrict others from doing anything the license permits.

Notices:

You do not have to comply with the license for elements of the material in the public domain or where your use is permitted by an applicable exception or limitation .

No warranties are given. The license may not give you all of the permissions necessary for your intended use. For example, other rights such as publicity, privacy, or moral rights may limit how you use the material.

Acknowledgments

I am profoundly grateful to my advisor, Professor Nils Thürey, whose guidance, encouragement, and unwavering support have been instrumental throughout this journey. I am indebted to him for providing me with the freedom to explore my own ideas while offering invaluable aid and direction. His dedication, patience, and commitment to my academic growth were truly exemplary. I am also grateful for his availability before deadlines and for our insightful weekly meetings, which have enriched my understanding immeasurably.

I extend my heartfelt thanks to my esteemed colleagues, whose contributions have significantly enriched this work. Robin Greif's expertise in software architecture has been invaluable, and I am deeply appreciative of Kiwon Um's assistance with software and role as second examiner. Lukas Prantl's camaraderie and assistance with the thesis, along with the enjoyable conference experiences we shared, are greatly appreciated. I am thankful to Felix Köhler, Björn List, Georg Kohl, Patrick Schnell, Liwei Chen, Stefan Rasp, Simon Niedermayr, Kevin Höhlein, Brener Ramos, Justus Thies, Sebastian Wohner, Prof. Matthias Nießner, and others for their insightful discussions, cheerful camaraderie, and various forms of support throughout this endeavor. Special mention goes to those who joined me for lunch at IPP / Mensa, creating an atmosphere of collaboration and camaraderie. I would also like to express my appreciation to external collaborators Afshawn Lotfi and Nico De Nitti for their invaluable assistance and exchange of knowledge.

Furthermore, I would like to extend my sincere gratitude to Prof. Dr. Christian Mendl, Prof. Dr. Phaedon-Stelios Koutsourelakis, and Prof. Dr. Matthew Piggott for their invaluable expertise and commitment as members of the Ph.D. committee.

To my family and friends, I owe a debt of gratitude beyond words. To my father, Matthias, for his role as a sounding board and meticulous reading of paper drafts. To my mother, Renate, and Ralph Harbort for their unwavering emotional support and genuine interest in my endeavors. To my uncle Alois and my cousin Miguel for engaging in thought-provoking scientific discussions.

Lastly, I would like to acknowledge the financial support provided by TUM, without which this research would not have been possible.

Thank you all for being a part of this incredible journey.

Bibliography

[Aal+18] Craig E Aalseth et al. “Search for neutrinoless double-β decay in Ge 76 with the majorana demonstrator”. In: *Physical review letters* 120.13 (2018), p. 132502.

[Aba+16] Martin Abadi et al. “Tensorflow: A system for large-scale machine learning”. In: *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*. 2016, pp. 265–283.

[Ago+13] M Agostini et al. “Pulse shape discrimination for Gerda Phase I data”. In: *The European Physical Journal C* 73.10 (2013), p. 2583.

[AK17] Brandon Amos and J Zico Kolter. “OptNet: Differentiable optimization as a layer in neural networks”. In: *International Conference on Machine Learning*. 2017.

[al17] Tianqi Chen et al. *DLPack: Open In Memory Tensor Structure*. `https://github.com/dmlc/dlpack`. 2017.

[All+22] Kelsey R Allen et al. “Physical design using differentiable learned simulators”. In: *arXiv preprint arXiv:2202.00728* (2022).

[Alz+21] Laith Alzubaidi et al. “Review of deep learning: concepts, CNN architectures, challenges, applications, future directions”. In: *Journal of big Data* 8 (2021), pp. 1–74.

[Ame14] William F Ames. *Numerical methods for partial differential equations*. Academic press, 2014.

[AMJ18] Hemant K Aggarwal, Merry P Mani, and Mathews Jacob. “MoDL: Model-based deep learning architecture for inverse problems”. In: *IEEE transactions on medical imaging* 38.2 (2018), pp. 394–405.

[And+23] Daniel Andrés Arcones et al. “Model bias identification for Bayesian calibration of stochastic digital twins of bridges”. In: *arXiv e-prints* (2023), arXiv–2312.

[Ang+16] Christof Angermueller et al. “Deep learning for computational biology”. In: *Molecular systems biology* 12.7 (2016), p. 878.

[Ant+23] Harbir Antil et al. “A deep neural network approach for parameterized PDEs and Bayesian inverse problems”. In: *Machine Learning: Science and Technology* 4.3 (Aug. 2023), p. 035015. DOI: 10.1088/2632-2153/ace67c.

[Arn12] Vladimir Igorevich Arnold. *Geometrical methods in the theory of ordinary differential equations*. Vol. 250. Springer Science & Business Media, 2012.

[Avi+18] Filipe de Avila Belbute-Peres et al. “End-to-end differentiable physics for learning and control”. In: *Advances in Neural Information Processing Systems*. 2018.

[Bab+20] Igor Babuschkin et al. *The DeepMind JAX Ecosystem*. 2020.

[Bar+19] Yohai Bar-Sinai et al. “Learning data-driven discretizations for partial differential equations”. In: *Proceedings of the National Academy of Sciences* 116.31 (2019), pp. 15344–15349.

[Bat+16] Peter Battaglia et al. “Interaction Networks for Learning about Objects, Relations and Physics”. In: *Advances in Neural Information Processing Systems*. 2016, pp. 4502–4510.

[Bat+18] P. W. Battaglia et al. “Relational inductive biases, deep learning, and graph networks”. In: *arXiv preprint arXiv:1806.01261* (2018).

[Ber+19] Christopher Berner et al. “Dota 2 with large scale deep reinforcement learning”. In: *arXiv preprint arXiv:1912.06680* (2019).

[Bez+17] Jeff Bezanson et al. “Julia: A fresh approach to numerical computing”. In: *SIAM review* 59.1 (2017), pp. 65–98.

[BHM18] Mikhail Belkin, Daniel J Hsu, and Partha Mitra. “Overfitting or perfect fitting? risk bounds for classification and regression rules that interpolate”. In: *Advances in neural information processing systems* 31 (2018).

[BHT13] Peter W Battaglia, Jessica B Hamrick, and Joshua B Tenenbaum. “Simulation as an engine of physical scene understanding”. In: *Proceedings of the National Academy of Sciences* 110.45 (2013).

[Bie+20] Katharina Bieker et al. “Deep model predictive flow control with limited sensor data and online learning”. In: *Theoretical and computational fluid dynamics* 34.4 (2020), pp. 577–591.

[Bir02] R Byron Bird. “Transport phenomena”. In: *Appl. Mech. Rev.* 55.1 (2002), R1–R4.

[Boj+16] Mariusz Bojarski et al. “End to end learning for self-driving cars”. In: *arXiv preprint arXiv:1604.07316* (2016).

[Boy01] John P Boyd. *Chebyshev and Fourier spectral methods*. Courier Corporation, 2001.

[BPL21] Randall Balestriero, Jerome Pesenti, and Yann LeCun. “Learning in high dimension always amounts to extrapolation”. In: *arXiv preprint arXiv:2110.09485* (2021).

[BR86] J.U. Brackbill and H. M. Ruppel. “FLIP: A method for adaptively zoned, particle-in-cell calculations of fluid flows in two dimensions”. In: *Journal of Computational Physics* 65.2 (1986), pp. 314–343.

[Bra+18] James Bradbury et al. "JAX: Composable transformations of Python+NumPy programs". In: *GitHub* (2018).

[Bra+19] John Bradshaw et al. "A Generative Model For Electron Paths". In: *ICLR*. 2019.

[Bro70] Charles George Broyden. "The convergence of a class of double-rank minimization algorithms 1. general considerations". In: *IMA Journal of Applied Mathematics* 6.1 (1970), pp. 76–90.

[BYR17] Wei Bao, Jun Yue, and Yulei Rao. "A deep learning framework for financial time series using stacked autoencoders and long-short term memory". In: *PloS one* 12.7 (2017), e0180944.

[Cao+22] Ningping Cao et al. "Neural networks for quantum inverse problems". In: *New Journal of Physics* 24.6, 063002 (June 2022), p. 063002. DOI: 10.1088/1367-2630/ac706c.

[Car+15] Rich Caruana et al. "Intelligible models for healthcare: Predicting pneumonia risk and hospital 30-day readmission". In: *Proceedings of the 21th ACM SIGKDD international conference on knowledge discovery and data mining*. 2015, pp. 1721–1730.

[Car+19] Giuseppe Carleo et al. "Machine learning and the physical sciences". In: *Reviews of Modern Physics* 91.4 (2019), p. 045002.

[CAS16] Paul Covington, Jay Adams, and Emre Sargin. "Deep neural networks for youtube recommendations". In: *Proceedings of the 10th ACM conference on recommender systems*. 2016, pp. 191–198.

[CDP21] Emilie Chouzenoux, Cecile Della Valle, and Jean-Christophe Pesquet. "Inversion of Integral Models: a Neural Network Approach". In: *arXiv e-prints*, arXiv:2105.15044 (May 2021), arXiv:2105.15044. DOI: 10.48550/arXiv.2105.15044. arXiv: 2105.15044 [math.OC].

[CH62] Richard Courant and David Hilbert. *Methods of Mathematical Physics*. Vol. II: Partial Differential Equations. Interscience Publishers, 1962.

[com15] Jupyter widgets community. "ipywidgets, a GitHub repository". Retrieved from https://github.com/jupyter-widgets/ipywidgets. 2015.

[Cou+12] National Research Council et al. *A national strategy for advancing climate modeling*. 2012.

[Cra+19] M. Cranmer et al. "Learning Symbolic Physics with Graph Networks". In: *arXiv preprint arXiv:1909.05862* (2019).

[Cra+20a] M. Cranmer et al. "Discovering Symbolic Models from Deep Learning with Inductive Biases". In: *arXiv preprint arXiv:2006.11287* (2020).

[Cra+20b] Miles Cranmer et al. "Lagrangian Neural Networks". In: *arXiv:2003.04630* (2020).

[Dai+21] Hanjun Dai et al. "Neural stochastic dual dynamic programming". In: *arXiv preprint arXiv:2112.00874* (2021).

[Deg+19] Jonas Degrave et al. "A differentiable physics engine for deep learning in robotics". In: *Frontiers in Neurorobotics* 13 (2019).

[Del+18] S Delaquis et al. "Deep neural networks for energy and position reconstruction in EXO-200". In: *Journal of Instrumentation* 13.08 (2018), P08023.

[Dhe+22] Benoit Dherin et al. *Why neural networks find simple solutions: the many regularizers of geometric complexity*. 2022. arXiv: 2209.13083 [cs.LG].

[Eva22] Lawrence C Evans. *Partial differential equations*. Vol. 19. American Mathematical Society, 2022.

[Fle00] Roger Fletcher. *Practical methods of optimization*. John Wiley & Sons, 2000.

[FP02] Joel H Ferziger and Milovan PeriC. *Computational methods for fluid dynamics*. 2002.

[Fun+21] Victor Fung et al. "Inverse design of two-dimensional materials with invertible neural networks". In: *npj Computational Materials* 7.1 (Dec. 2021), p. 200. ISSN: 2057-3960. DOI: 10.1038/s41524-021-00670-x.

[GBC16] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016.

[GDY19] Samuel Greydanus, Misko Dzamba, and Jason Yosinski. "Hamiltonian Neural Networks". In: *Advances in Neural Information Processing Systems*. 2019, pp. 15353–15363.

[GEB16] Leon A Gatys, Alexander S Ecker, and Matthias Bethge. "Image style transfer using convolutional neural networks". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 2414–2423.

[GH18] Daniel George and EA Huerta. "Deep Learning for real-time gravitational wave detection and parameter estimation: Results with Advanced LIGO data". In: *Physics Letters B* 778 (2018), pp. 64–70.

[Gho96] Sandip Ghosal. "An analysis of numerical errors in large-eddy simulations of turbulence". In: *Journal of Computational Physics* 125.1 (1996), pp. 187–206.

[GHS16] Erik Gawehn, Jan A Hiss, and Gisbert Schneider. "Deep learning in drug discovery". In: *Molecular informatics* 35.1 (2016), pp. 3–14.

[GHV17] Garrett B Goh, Nathan O Hodas, and Abhinav Vishnu. "Deep learning for computational chemistry". In: *Journal of computational chemistry* 38.16 (2017), pp. 1291–1307.

[Gil+17] Justin Gilmer et al. "Neural Message Passing for Quantum Chemistry". In: 2017.

[GM78] Philip E Gill and Walter Murray. "Algorithms for the solution of the nonlinear least-squares problem". In: *SIAM Journal on Numerical Analysis* 15.5 (1978), pp. 977–992.

[Goo16] Ian Goodfellow. "NIPS 2016 Tutorial: Generative Adversarial Networks". In: *arXiv:1701.00160* (Dec. 2016). eprint: 1701.00160 (cs).

[GST07] Amparo Gil, Javier Segura, and Nico M Temme. *Numerical methods for special functions*. SIAM, 2007.

[GV12] Gene H Golub and Charles F Van Loan. *Matrix computations*. Vol. 3. JHU press, 2012.

[HA15] José Miguel Hernández-Lobato and Ryan Adams. "Probabilistic backpropagation for scalable learning of bayesian neural networks". In: *International conference on machine learning*. PMLR. 2015, pp. 1861–1869.

[Hab04] Richard Haberman. "Applied partial differential equations with Fourier series and boundary value problems". In: *(No Title)* (2004).

[Har+20] C. R. Harris et al. “Array programming with NumPy”. In: *Nature* 585.7825 (2020), pp. 357–362.

[Hay94] Simon Haykin. *Neural networks: a comprehensive foundation*. Prentice Hall PTR, 1994.

[He+19] S. He et al. “Learning to predict the cosmological structure formation”. In: *Proceedings of the National Academy of Sciences of the United States of America* 116 (2019), pp. 13825–13832.

[Hei+21] Eric Heiden et al. “NeuralSim: Augmenting differentiable simulators with neural networks”. In: *2021 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE. 2021, pp. 9474–9481.

[Hen+20] Tom Hennigan et al. *Haiku: Sonnet for JAX*. Version 0.0.3. 2020.

[Hin+12] Geoffrey Hinton et al. “Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups”. In: *IEEE Signal processing magazine* 29.6 (2012), pp. 82–97.

[HKT20] Philipp Holl, Vladlen Koltun, and Nils Thuerey. “Learning to Control PDEs with Differentiable Physics”. In: *International Conference on Learning Representations (ICLR)*. 2020.

[HKT21] Philipp Holl, Vladlen Koltun, and Nils Thuerey. “Physical Gradients for Deep Learning”. In: *arXiv* 2109.15048 (2021).

[HKT22] Philipp Holl, Vladlen Koltun, and Nils Thuerey. “Scale-invariant Learning by Physics Inversion”. In: *Advances in Neural Information Processing Systems* 35 (2022), pp. 5390–5403.

[Hos+17] Tareq Hossen et al. “Short-term load forecasting using deep neural networks (DNN)”. In: *2017 North American Power Symposium (NAPS)*. IEEE. 2017, pp. 1–6.

[HOT06] Geoffrey E. Hinton, Simon Osindero, and Yee Whye Teh. “A Fast Learning Algorithm for Deep Belief Nets”. In: *Neural Computation* 18 (2006), pp. 1527–1554.

[HSG19] Stephan Hoyer, Jascha Sohl-Dickstein, and Sam Greydanus. “Neural reparameterization improves structural optimization”. In: *arXiv preprint arXiv:1909.04240* (2019).

[Hsi+19] Jun-Ting Hsieh et al. “Learning neural PDE solvers with convergence guarantees”. In: *arXiv:1906.01200* (2019).

[Hu+19] Yuanming Hu et al. “ChainQueen: A Real-Time Differentiable Physical Simulator for Soft Robotics”. In: *ICRA*. 2019.

[Hu+20] Yuanming Hu et al. “DiffTaichi: Differentiable Programming for Physical Simulation”. In: *International Conference on Learning Representations (ICLR)* (2020).

[Hua+05] Sixun Huang et al. “Inverse problems in atmospheric science and their application”. In: *Journal of Physics: Conference Series* 12.1 (Jan. 2005), p. 45. DOI: `10.1088/1742-6596/12/1/005`.

[Hun07] J. D. Hunter. “Matplotlib: A 2D graphics environment”. In: *Computing in Science & Engineering* 9.3 (2007), pp. 90–95. DOI: `10.1109/MCSE.2007.55`.

[Inc15] Plotly Technologies Inc. *Collaborative data science*. 2015. URL: `https://plot.ly`.

[Ing+19] John Ingraham et al. “Learning Protein Structure with a Differentiable Simulator”. In: *ICLR*. 2019.

[Inn+19] Mike Innes et al. “A differentiable programming system to bridge machine learning and scientific computing”. In: *arXiv 1907.07587* (2019).

[Jam03] Antony Jameson. “Aerodynamic shape optimization using the adjoint method”. In: *Lectures at the Von Karman Institute, Brussels* (2003).

[JM15] Michael I Jordan and Tom M Mitchell. “Machine learning: Trends, perspectives, and prospects”. In: *Science* 349.6245 (2015), pp. 255–260.

[JOB91] Y Jarny, MN Ozisik, and JP Bardon. “A general optimization method using adjoint equation for solving multidimensional inverse heat conduction”. In: *International journal of heat and mass transfer* 34.11 (1991), pp. 2911–2919.

[Joh+04] Barbara M Johnston et al. “Non-Newtonian blood flow in human right coronary arteries: steady state simulations”. In: *Journal of biomechanics* 37.5 (2004), pp. 709–720.

[KB14] Diederik Kingma and Jimmy Ba. “Adam: A method for stochastic optimization”. In: *arXiv:1412.6980* (2014).

[KB15] Diederik Kingma and Jimmy Ba. “Adam: A Method for Stochastic Optimization”. In: *International Conference on Learning Representations (ICLR)*. 2015.

[Kha+17] Elias Khalil et al. “Learning combinatorial optimization algorithms over graphs”. In: *Advances in neural information processing systems* 30 (2017).

[KHT22] Jonathan Klimesch, Philipp Holl, and Nils Thuerey. “Simulating liquids with graph networks”. In: *arXiv preprint arXiv:2203.07895* (2022).

[Kim+19] Byungsoo Kim et al. “Deep Fluids: A Generative Network for Parameterized Fluid Simulations”. In: *Computer Graphics Forum* (2019). ISSN: 1467-8659. DOI: `10.1111/cgf.13619`.

[Klu+16] Thomas Kluyver et al. “Jupyter Notebooks – a publishing format for reproducible computational workflows”. In: *Positioning and Power in Academic Publishing: Players, Agents and Agendas*. Ed. by F. Loizides and B. Schmidt. IOS Press. 2016, pp. 87–90.

[Koc+21a] Dmitrii Kochkov et al. “Machine Learning Accelerated Computational Fluid Dynamics”. In: *arXiv:2102.01010 [physics]* (2021). arXiv: `2102.01010`.

[Koc+21b] Dmitrii Kochkov et al. “Machine learning–accelerated computational fluid dynamics”. In: *Proceedings of the National Academy of Sciences* 118.21 (2021).

[KP18] Andreas Kamilaris and Francesc X Prenafeta-Boldú. “Deep learning in agriculture: A survey”. In: *Computers and electronics in agriculture* 147 (2018), pp. 70–90.

[Kri+21] Aditi Krishnapriyan et al. “Characterizing possible failure modes in physics-informed neural networks”. In: *Advances in Neural Information Processing Systems* 34 (2021), pp. 26548–26560.

[KSH12] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. “ImageNet Classification with Deep Convolutional Neural Networks”. In: *Advances in Neural Information Processing Systems*. 2012.

[Lad+15] Lubor Ladický et al. “Data-Driven Fluid Simulations Using Regression Forests”. In: *ACM Trans. Graph.* 34.6 (Oct. 2015), 199:1–199:9. ISSN: 0730-0301. DOI: 10.1145/2816795.2818129.

[LBH15] Y. LeCun, Y. Bengio, and G. Hinton. “Deep Learning”. In: *Nature* 521 (May 2015), pp. 436–44.

[Lev+16] Sergey Levine et al. “End-to-end training of deep visuomotor policies”. In: *Journal of Machine Learning Research* 17.39 (2016), pp. 1–40.

[LeV07] Randall J LeVeque. *Finite difference methods for ordinary and partial differential equations: steady-state and time-dependent problems*. SIAM, 2007.

[Li+18] Yunzhu Li et al. “Learning particle dynamics for manipulating rigid bodies, deformable objects, and fluids”. In: *arXiv:1810.01566* (2018).

[Li+20] Housen Li et al. “NETT: Solving inverse problems with deep neural networks”. In: *Inverse Problems* 36.6 (2020), p. 065005.

[Li+24] Siyi Li et al. “Learning to optimise wind farms with graph transformers”. In: *Applied Energy* 359 (2024), p. 122758.

[Lip18] Zachary C Lipton. “The mythos of model interpretability: In machine learning, the concept of interpretability is both important and slippery.” In: *Queue* 16.3 (2018), pp. 31–57.

[Lit+17] Geert Litjens et al. “A survey on deep learning in medical image analysis”. In: *Medical image analysis* 42 (2017), pp. 60–88.

[LKT16] Julia Ling, Andrew Kurzawski, and Jeremy Templeton. “Reynolds averaged turbulence modelling using deep neural networks with embedded invariance”. In: *Journal of Fluid Mechanics* 807 (2016).

[LLK19] Junbang Liang, Ming Lin, and Vladlen Koltun. “Differentiable Cloth Simulation for Inverse Problems”. In: *Advances in Neural Information Processing Systems*. 2019, pp. 771–780.

[LMK21] Qin Lou, Xuhui Meng, and George Em Karniadakis. “Physics-informed neural networks for solving forward and inverse flow problems via the Boltzmann-BGK formulation”. In: *Journal of Computational Physics* 447 (Dec. 2021), p. 110676. ISSN: 0021-9991. DOI: 10.1016/j.jcp.2021.110676.

[LN89] Dong C Liu and Jorge Nocedal. “On the limited memory BFGS method for large scale optimization”. In: *Mathematical programming* 45.1-3 (1989), pp. 503–528.

[Lon+17] Zichao Long et al. “PDE-Net: Learning PDEs from Data”. In: *arXiv:1710.09668* (2017).

[Lu+21] Lu Lu et al. “Physics-informed neural networks with hard constraints for inverse design”. In: *SIAM Journal on Scientific Computing* 43.6 (2021), B1105–B1132.

[Luc+18] Alice Lucas et al. “Using deep neural networks for inverse problems in imaging: beyond analytical methods”. In: *IEEE Signal Processing Magazine* 35.1 (2018), pp. 20–36.

[Mai+19] Rajesh Maingi et al. “Summary of the FESAC transformative enabling capabilities panel report”. In: *Fusion Science and Technology* 75.3 (2019), pp. 167–177.

[MAL13] Kohta Murase, Markus Ahlers, and Brian C Lacki. "Testing the hadronuclear origin of PeV neutrinos observed with IceCube". In: *Physical Review D* 88.12 (2013), p. 121301.

[McK10] Wes McKinney. "Data Structures for Statistical Computing in Python". In: *Proceedings of the 9th Python in Science Conference*. Ed. by Stéfan van der Walt and Jarrod Millman. 2010, pp. 56–61. DOI: `10.25080/Majora-92bf1922-00a`.

[MFD19] Sebastian K Mitusch, Simon W Funke, and Jørgen S Dokken. "dolfin-adjoint 2018.1: automated adjoints for FEniCS and Firedrake". In: *Journal of Open Source Software* 4.38 (2019), p. 1292.

[MJU17] Michael T. McCann, Kyong Hwan Jin, and Michael Unser. "Convolutional Neural Networks for Inverse Problems in Imaging: A Review". In: *IEEE Signal Processing Magazine* 34.6 (2017), pp. 85–95. DOI: `10.1109/MSP.2017.2739299`.

[Mni+13a] Volodymyr Mnih et al. "Playing atari with deep reinforcement learning". In: *arXiv:1312.5602* (2013).

[Mni+13b] Volodymyr Mnih et al. "Playing atari with deep reinforcement learning". In: *arXiv preprint arXiv:1312.5602* (2013).

[Moe+23] Thomas M Moerland et al. "Model-based reinforcement learning: A survey". In: *Foundations and Trends® in Machine Learning* 16.1 (2023), pp. 1–118.

[Moh21] Ali Mohammad-Djafari. "Regularization, Bayesian inference, and machine learning methods for inverse problems". In: *Entropy* 23.12 (2021), p. 1673.

[Mor+18] Jeremy Morton et al. "Deep dynamical modeling and control of unsteady fluid flows". In: *Advances in Neural Information Processing Systems*. 2018.

[MP43] Warren S McCulloch and Walter Pitts. "A logical calculus of the ideas immanent in nervous activity". In: *The bulletin of mathematical biophysics* 5 (1943), pp. 115–133.

[Mro+18] Damian Mrowca et al. "Flexible neural representation for physics prediction". In: 2018.

[Muk+21] Subhadip Mukherjee et al. "End-to-end reconstruction meets data-driven regularization for inverse problems". In: *Advances in Neural Information Processing Systems* 34 (2021), pp. 21413–21425.

[MW70] Jon Mathews and Robert Lee Walker. *Mathematical methods of physics*. Vol. 501. WA Benjamin New York, 1970.

[OBA22] M. Oulghelou, C. Beghein, and C. Allery. "A surrogate optimization approach for inverse problems: Application to turbulent mixed-convection flows". In: *Computers & Fluids* 241 (2022), p. 105490. ISSN: 0045-7930. DOI: `https://doi.org/10.1016/j.compfluid.2022.105490`.

[Pas+19] Adam Paszke et al. "PyTorch: An Imperative Style, High-Performance Deep Learning Library". In: 2019.

[Pat+22] Jaideep Pathak et al. "FourCastNet: A global data-driven high-resolution weather model using adaptive fourier neural operators". In: *arXiv:2202.11214* (2022).

[Pfa+20] Tobias Pfaff et al. "Learning Mesh-Based Simulation with Graph Networks". In: *arXiv preprint arXiv:2010.03409* (2020).

[Pfr+18] Julius Pfrommer et al. "Optimisation of manufacturing process parameters using deep neural networks as surrogate models". In: *Procedia CIRP* 72 (2018). 51st CIRP Conference on Manufacturing Systems, pp. 426–431. ISSN: 2212-8271. DOI: `https://doi.org/10.1016/j.procir.2018.03.046`.

[Ple06] R-E Plessix. "A review of the adjoint-state method for computing the gradient of a functional with geophysical applications". In: *Geophysical Journal International* 167.2 (2006), pp. 495–503.

[Pre+07] William H. Press et al. *Numerical Recipes*. 3rd ed. Cambridge University Press, 2007. ISBN: 9780521880688.

[PW17] Patrick Putzky and Max Welling. "Recurrent inference machines for solving inverse problems". In: *arXiv preprint arXiv:1706.04008* (2017).

[Rac+20] Christopher Rackauckas et al. "Universal differential equations for scientific machine learning". In: *arXiv:2001.04385*. 2020.

[RC83] CM Rhie and W Li Chow. "Numerical study of the turbulent flow past an airfoil with trailing edge separation". In: *AIAA journal* 21.11 (1983), pp. 1525–1532.

[RFB15] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. "U-net: Convolutional networks for biomedical image segmentation". In: *Medical Image Computing and Computer-Assisted Intervention*. 2015.

[RHW86] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. "Learning representations by back-propagating errors". In: *nature* 323.6088 (1986), pp. 533–536.

[Ros58] Frank Rosenblatt. "The perceptron: a probabilistic model for information storage and organization in the brain." In: *Psychological review* 65.6 (1958), p. 386.

[RPG18] S. Rasp, M. Pritchard, and P. Gentine. "Deep learning to represent subgrid processes in climate models". In: *Proceedings of the National Academy of Sciences of the United States of America* 115 (2018), pp. 9684–9689.

[RPK19] Maziar Raissi, Paris Perdikaris, and George Karniadakis. "Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations". In: *Journal of Computational Physics* 378 (2019), pp. 686–707.

[RPM20] Simiao Ren, Willie Padilla, and Jordan Malof. "Benchmarking deep inverse models over time, and the neural-adjoint method". In: *Advances in Neural Information Processing Systems* 33 (2020), pp. 38–48.

[RT21] Stephan Rasp and Nils Thuerey. "Data-driven medium-range weather prediction with a resnet pretrained on climate simulations: A new model for weatherbench". In: *Journal of Advances in Modeling Earth Systems* 13.2 (2021), e2020MS002405.

[Rud17] Sebastian Ruder. *An overview of gradient descent optimization algorithms*. 2017. arXiv: `1609.04747 [cs.LG]`.

[RYK18] Maziar Raissi, Alireza Yazdani, and George Em Karniadakis. "Hidden fluid mechanics: A Navier-Stokes informed deep learning framework for assimilating flow visualization data". In: *arXiv:1808.04327* (2018).

[San+18] Alvaro Sanchez-Gonzalez et al. "Graph networks as learnable physics engines for inference and control". In: *arXiv:1806.01242* (2018).

[San+20] Alvaro Sanchez-Gonzalez et al. "Learning to Simulate Complex Physics with Graph Networks". In: *arXiv:2002.09405* (2020).

[SB15] Joshua Saxe and Konstantin Berlin. "Deep neural network based malware detection using two dimensional binary program features". In: *2015 10th international conference on malicious and unwanted software (MALWARE)*. IEEE. 2015, pp. 11–20.

[SBV20] J. Shlomi, P. W. Battaglia, and J. Vlimant. "Graph Neural Networks in Particle Physics". In: *arXiv preprint arXiv:2007.13681* (2020).

[SC19] Samuel S Schoenholz and Ekin D Cubuk. "JAX, MD: End-to-End Differentiable, Hardware Accelerated, Molecular Dynamics in Pure Python". In: *arXiv:1912.04232* (2019).

[SC20] Samuel Schoenholz and Ekin Dogus Cubuk. "Jax md: a framework for differentiable physics". In: *Advances in Neural Information Processing Systems* 33 (2020), pp. 11428–11441.

[Sch+17] Robin Tibor Schirrmeister et al. "Deep learning with convolutional neural networks for EEG decoding and visualization". In: *Human brain mapping* 38.11 (2017), pp. 5391–5420.

[Sch15] J. Schmidhuber. "Deep Learning in Neural Networks: An Overview". In: *Neural Networks* 61 (2015). Published online 2014; based on TR arXiv:1404.7828 [cs.NE]. DOI: `10.1016/j.neunet.2014.09.003`.

[SF18] Connor Schenck and Dieter Fox. "SPNets: Differentiable Fluid Dynamics for Deep Neural Networks". In: *Conference on Robot Learning*. 2018, pp. 317–335.

[Sha+15] Bobak Shahriari et al. "Taking the human out of the loop: A review of Bayesian optimization". In: *Proceedings of the IEEE* 104.1 (2015), pp. 148–175.

[Shm+23] Alexander Shmakov et al. "End-To-End Latent Variational Diffusion Models for Inverse Problems in High Energy Physics". In: *arXiv preprint arXiv:2305.10399* (2023).

[SHT21] Patrick Schnell, Philipp Holl, and Nils Thuerey. "Half-Inverse Gradients for Physical Deep Learning". In: *International Conference on Learning Representations*. 2021.

[Sil+17] David Silver et al. "Mastering the game of Go without human knowledge". In: *Nature* 550.7676 (2017).

[Son+22] Wenbin Song et al. "M2N: mesh movement networks for PDE solvers". In: *Advances in Neural Information Processing Systems* 35 (2022), pp. 7199–7210.

[SSS85] Gordon D Smith, Gordon D Smith, and Gordon Dennis Smith Smith. *Numerical solution of partial differential equations: finite difference methods*. Oxford university press, 1985.

[Sta+21] Kimberly Stachenfeld et al. “Learned Coarse Models for Efficient Turbulence Simulation”. In: *arXiv preprint arXiv:2112.15275* (2021).

[Sto+13] Thomas F Stocker et al. “Climate change 2013: The physical science basis”. In: *Contribution of working group I to the fifth assessment report of the intergovernmental panel on climate change* 1535 (2013).

[Str07] Walter A Strauss. *Partial differential equations: An introduction*. John Wiley & Sons, 2007.

[Tak+21] Tetsuya Takahashi et al. “Differentiable fluids with solid coupling for learning and control”. In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 35(7). 2021, pp. 6138–6146.

[Tan+18] Chuanqi Tan et al. “A survey on deep transfer learning”. In: *Artificial Neural Networks and Machine Learning–ICANN 2018: 27th International Conference on Artificial Neural Networks, Rhodes, Greece, October 4-7, 2018, Proceedings, Part III 27*. Springer. 2018, pp. 270–279.

[Tar05] Albert Tarantola. *Inverse problem theory and methods for model parameter estimation*. SIAM, 2005.

[TET12] Emanuel Todorov, Tom Erez, and Yuval Tassa. “Mujoco: A physics engine for model-based control”. In: *IEEE/RSJ International Conference on Intelligent Robots and Systems*. 2012.

[Thu+18] Nils Thuerey et al. *Deep Learning Methods for Reynolds-Averaged Navier-Stokes Simulations of Airfoil Flows*. 2018. arXiv: `1810.08217 [cs.LG]`.

[TLY20] Simron Thapa, Nianyi Li, and Jinwei Ye. “Dynamic fluid surface reconstruction using deep neural network”. In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2020, pp. 21–30.

[Tom+17] Jonathan Tompson et al. “Accelerating Eulerian Fluid Simulation With Convolutional Networks”. In: *Proceedings of Machine Learning Research*. 2017, pp. 3424–3433.

[Tou+18] Marc Toussaint et al. “Differentiable physics and stable modes for tool-use and manipulation planning”. In: *Robotics: Science and Systems*. 2018.

[TSM12] Karl E Taylor, Ronald J Stouffer, and Gerald A Meehl. “An overview of CMIP5 and the experiment design”. In: *Bulletin of the American Meteorological Society* 93.4 (2012), pp. 485–498.

[Um+20] Kiwon Um et al. “Solver-in-the-Loop: Learning from Differentiable Physics to Interact with Iterative PDE-Solvers”. In: *Advances in Neural Information Processing Systems* (2020).

[Umm+20] B. Ummenhofer et al. “Lagrangian Fluid Simulation with Continuous Convolutions”. In: *International Conference on Learning Representations (ICLR)* (2020).

[UVL18] Dmitry Ulyanov, Andrea Vedaldi, and Victor Lempitsky. “Deep image prior”. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2018, pp. 9446–9454.

[Vas+17] Ashish Vaswani et al. “Attention is all you need”. In: *Advances in Neural Information Processing Systems*. 2017, pp. 5998–6008.

[VD02] Ricardo Vilalta and Youssef Drissi. “A perspective view and survey of meta-learning”. In: *Artificial intelligence review* 18 (2002), pp. 77–95.

[VD95] Guido Van Rossum and Fred L Drake Jr. *Python reference manual*. Centrum voor Wiskunde en Informatica Amsterdam, 1995.

[Wei17] E Weinan. “A proposal on machine learning via dynamical systems”. In: *Communications in Mathematics and Statistics* 5.1 (2017), pp. 1–11.

[Wur+23] Tobias Wurth et al. “Physics-informed neural networks for data-free surrogate modelling and engineering optimization An example from composite manufacturing”. In: *Materials & Design* 231 (2023), p. 112034. ISSN: 0264-1275. DOI: `https://doi.org/10.1016/j.matdes.2023.112034`.

[Xie+18] You Xie et al. “tempogan: A temporally coherent, volumetric gan for super-resolution fluid flow”. In: *ACM Transactions on Graphics (TOG)* 37.4 (2018), pp. 1–15.

[Yan+19] XIA Yang et al. “Predictive large-eddy-simulation wall modeling via physics-informed neural networks”. In: *Physical Review Fluids* 4.3 (2019), p. 034602.

[Yan+23] Ling Yang et al. “Diffusion models: A comprehensive survey of methods and applications”. In: *ACM Computing Surveys* 56.4 (2023), pp. 1–39.

[Yin+19] R. Ying et al. “GNNExplainer: Generating Explanations for Graph Neural Networks”. In: *Advances in neural information processing systems* 32 (2019), pp. 9240–9251.

[ZB05] Yongning Zhu and Robert Bridson. “Animating Sand As a Fluid”. In: *ACM Trans. Graph.* 24.3 (July 2005), pp. 965–972. ISSN: 0730-0301. DOI: `10.1145/1073204.1073298`.

[Zhu+19] Yinhao Zhu et al. “Physics-constrained deep learning for high-dimensional surrogate modeling and uncertainty quantification without labeled data”. In: *Journal of Computational Physics* 394 (2019), pp. 56–81.

[Zie+77] Olgierd Cecil Zienkiewicz et al. *The finite element method*. Vol. 3. McGraw-hill London, 1977.

[ZT05] Olek C Zienkiewicz and Robert L Taylor. *The finite element method for solid and structural mechanics*. Elsevier, 2005.

Φ_{ML}: Intuitive Scientific Computing with Dimension Types for Jax, PyTorch, TensorFlow & NumPy

Philipp Holl[1] and **Nils Thuerey**[1]

1 School of Computation, Information and Technology, Technical University of Munich, Germany

DOI: 10.21105/joss.06171

Software

- Review
- Repository
- Archive

Editor: Marcel Stimberg

Reviewers:

- @wandeln
- @chaoming0625
- @gauravbokil8

Submitted: 11 August 2023
Published: 01 March 2024

Summary

Φ_{ML} is a math and neural network library designed for science applications. It enables users to quickly evaluate many network architectures on their data sets, perform (sparse) linear and non-linear optimization, and write differentiable simulations that scale to *n* dimensions. Φ_{ML} is compatible with Jax, PyTorch, TensorFlow and NumPy, and user code can be executed on all of these backends. The project is hosted at https://github.com/tum-pbs/PhiML under the MIT license.

Statement of need

Machine learning (ML) has become an essential tool for scientific research. In recent years, ML has been used to make significant advances in a wide range of scientific fields, including chemistry (Butler et al., 2018), materials science (Wei et al., 2019), weather and climate prediction (Bochenek & Ustrnul, 2022; Rolnick et al., 2022), computational fluid dynamics (Brunton et al., 2020), drug discovery (Jumper et al., 2021; Vamathevan et al., 2019), astrophysics (De La Calleja & Fuentes, 2004; Ntampaka et al., 2015; Petroff et al., 2020), geology (Rodriguez-Galiano et al., 2015), and many more. The use of ML for scientific applications is still in its early stages, but it has the potential to revolutionize the way that science is done. ML can help researchers to make new discoveries and insights that were previously impossible.

The availability of domain knowledge sets science applications apart from other ML fields like computer vision or language modelling. Domain knowledge often allows for explicit modelling of known dynamics by simulating them with handwritten algorithms, which has been shown to improve results when training ML models (Raissi et al., 2019; Um et al., 2020). Implementing differentiable simulations into ML frameworks requires different functions and concepts than classical ML tasks. The major differences are:

- Data typically represent objects or signals that exist in space and time. Data dimensions are interpretable, e.g. vector components, time series, *n*-dimensional lattices.
- Information transfer is usually local, resulting in sparsity in the dependency matrix between objects (particles, elements or cells).
- A high numerical accuracy is desirable for some operations, often requiring 64-bit and 32-bit floating point calculations.

However, current machine learning frameworks have been designed for the core ML tasks which reflects in their priorities and design choices. This can result in overly verbose code when implementing scientific applications and may require implementing custom operators, since many common functions like sparse-sparse matrix multiplication, periodic padding or sparse linear solvers are not available in all libraries.

Φ_{ML} is a scientific computing library based on Python 3 (Van Rossum & Drake, 2009) targeting scientific applications that use machine learning methods. Its main goals are:

- **Reusability.** Code based on Φ_{ML} should be able to run in many settings without modification. It should be agnostic towards the dimensionality of simulated systems and the employed discretization. All code should be trivially vectorizable.
- **Compatibility.** Users should be free to choose whatever ML or third-party library they desire without modifying their simulation code. Φ_{ML} should support Linux, Windows and Mac.
- **Usability.** Φ_{ML} should be easy to learn and use, matching existing APIs where possible. It should encourage users to write concise and expressive code.
- **Maintainability.** All high-level source code of Φ_{ML} should be easy to understand. Continuous testing should be used to ensure that future updates do not break existing code.
- **Performance.** Φ_{ML} should be able to make use of hardware accelerators, such as GPUs and TPUs, where possible. During development, we prioritize rapid code iterations over execution speed but the completed code should run as fast as if written directly against the chosen ML library.

In the following, we explain the architecture and major features that help Φ_{Flow} reach these goals. Φ_{ML} consists of a high-level NumPy-like API geared towards writing easy-to-read and scalable simulation code, as well as a neural network API designed to allow users to quickly iterate over many network architectures and hyperparameter settings. Similar to eagerpy (Rauber et al., 2020), Φ_{ML} integrates with Jax (Bradbury et al., 2018), PyTorch (Paszke et al., 2019), TensorFlow (Abadi et al., 2016) and NumPy (Harris et al., 2020) and provides a custom `Tensor` class. However, Φ_{ML} adds additional functionality.

- **Dimension names.** Tensor dimensions are always referenced by their user-defined name, not their index. We support the syntax `tensor.dim` for operations like indexing or unstacking to make using dimension names as simple as possible.
- **Automatic reshaping.** Φ_{ML} automatically transposes tensors and inserts singleton dimensions to match arguments. Consequently, user code is agnostic to the dimension order by default.
- **Element names.** Slices or *items* along dimensions can be named as well, e.g. allowing users to specify that a dimension lists the values `(x,y,z)` or `(r,g,b)`. These names can be used in slicing, gathering and scattering operations.
- **Dimension types.** Tensor dimensions are grouped into five different types: *batch*, *spatial*, *instance*, *channel*, and *dual*. This allows tensor-related functions to automatically select dimensions to operate on, without requiring the user to specify individual dimensions.
- **Non-uniform tensors.** Stacking tensors with different dimension sizes yields non-uniform tensors. Φ_{ML} keeps track of the resulting shape, allowing users to operate on non-uniform tensors the same way as uniform ones.
- **Floating-point precision by context.** All tensor operations determine the desired floating point precision from the operation context, not the data types of its inputs. This is much simpler and more predictable than the systems used by other libraries.
- **Lazy stacking.** New memory is only allocated once stacked data is required as a block. Consequently, functions can unstack the components, operate on them individually, and restack them, without worrying about unnecessary memory allocations.
- **Sparse matrices from linear functions.** Φ_{ML} can transform linear functions into their corresponding sparse matrix representation. This makes solving linear systems of equations more performant and enables computation of preconditioners.
- **Compute device from Inputs.** Tensor operations execute on the device on which the tensors reside. This prevents unintentional copies and transfers, as users have to explicitly declare them.
- **Custom CUDA Operatorions.** Φ_{ML} provides custom CUDA kernels for specific operations that could bottleneck simulations, such as grid sampling for TensorFlow or linear solves.

Research Projects

Φ_{ML} has been in development since 2019 as part of the PhiFlow (Φ_{Flow}) project where it originated as a unified API for TensorFlow and NumPy, used to run differentiable fluid simulations. Φ_{Flow} includes geometry, physics, and visualization modules, all of which use the `math` API of Φ_{ML} to benefit from its reusability, compatibility, and performance.

It was first used to show that differentiable PDE simulations can be used to train neural networks that steer the dynamics towards desired outcomes (Holl et al., 2019). Differentiable PDEs, implemented against Φ_{ML}'s API, were later shown to benefit learning corrections for low-resolution or incomplete physics models (Um et al., 2020). These findings were summarized and formalized in Thuerey et al. (2022), along with many additional examples.

The library was also used in network optimization publications, such as showing that inverted simulations can be used to train networks (Holl et al., 2022) and that gradient inversion benefits learning the solutions to inverse problems (Schnell et al., 2021).

Simulations powered by Φ_{ML} have since been used in open data sets (Gupta & Brandstetter, 2022; Takamoto et al., 2022) and in publications from various research groups (Brandstetter et al., 2021, 2023; Li et al., 2023; Parekh et al., 1993; Ramos et al., 2022; Sengar et al., 2021; Wandel et al., 2020, 2021; P. Wang, 2023; R. Wang et al., 2022a, 2022b; Wu et al., 2022).

Acknowledgements

We would like to thank Robin Greif, Kartik Bali, Elias Djossou and Brener Ramos for their contributions, as well as everyone who contributed to the project on GitHub.

References

Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., & others. (2016). Tensorflow: A system for large-scale machine learning. *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, 265–283.

Bochenek, B., & Ustrnul, Z. (2022). Machine learning in weather prediction and climate analyses—applications and perspectives. *Atmosphere*, *13*(2), 180. https://doi.org/10.3390/atmos13020180

Bradbury, J., Frostig, R., Hawkins, P., Johnson, M. J., Leary, C., Maclaurin, D., Necula, G., Paszke, A., VanderPlas, J., Wanderman-Milne, S., & Zhang, Q. (2018). *JAX: Composable transformations of Python+NumPy programs* (Version 0.2.5). http://github.com/google/jax

Brandstetter, J., Berg, R. van den, Welling, M., & Gupta, J. K. (2023). *Clifford neural layers for PDE modeling*. arXiv. https://arxiv.org/abs/2209.04934

Brandstetter, J., Worrall, D. E., & Welling, M. (2021). Message passing neural PDE solvers. *International Conference on Learning Representations*.

Brunton, S. L., Noack, B. R., & Koumoutsakos, P. (2020). Machine learning for fluid mechanics. *Annual Review of Fluid Mechanics*, *52*, 477–508. https://doi.org/10.1146/annurev-fluid-010719-060214

Butler, K. T., Davies, D. W., Cartwright, H., Isayev, O., & Walsh, A. (2018). Machine learning for molecular and materials science. *Nature*, *559*(7715), 547–555. https://doi.org/10.1038/s41586-018-0337-2

De La Calleja, J., & Fuentes, O. (2004). Machine learning and image analysis for morphological galaxy classification. *Monthly Notices of the Royal Astronomical Society*, *349*(1), 87–93. https://doi.org/10.1111/j.1365-2966.2004.07442.x

Gupta, J. K., & Brandstetter, J. (2022). *Towards multi-spatiotemporal-scale generalized PDE modeling*. arXiv. https://arxiv.org/abs/2209.15616

Harris, C. R., Millman, K. J., Walt, S. J. van der, Gommers, R., Virtanen, P., Cournapeau, D., Wieser, E., Taylor, J., Berg, S., Smith, N. J., Kern, R., Picus, M., Hoyer, S., Kerkwijk, M. H. van, Brett, M., Haldane, A., Río, J. F. del, Wiebe, M., Peterson, P., … Oliphant, T. E. (2020). Array programming with NumPy. *Nature*, *585*(7825), 357–362. https://doi.org/10.1038/s41586-020-2649-2

Holl, P., Koltun, V., & Thuerey, N. (2022). Scale-invariant learning by physics inversion. *Advances in Neural Information Processing Systems*, *35*, 5390–5403.

Holl, P., Thuerey, N., & Koltun, V. (2019). Learning to control PDEs with differentiable physics. *International Conference on Learning Representations*.

Jumper, J., Evans, R., Pritzel, A., Green, T., Figurnov, M., Ronneberger, O., Tunyasuvunakool, K., Bates, R., Žídek, A., Potapenko, A., & others. (2021). Highly accurate protein structure prediction with AlphaFold. *Nature*, *596*(7873), 583–589. https://doi.org/10.1038/s41586-021-03819-2

Li, Z., Patil, S., Shu, D., & Farimani, A. B. (2023). Latent neural PDE solver for time-dependent systems. *NeurIPS 2023 AI for Science Workshop*.

Ntampaka, M., Trac, H., Sutherland, D. J., Battaglia, N., Póczos, B., & Schneider, J. (2015). A machine learning approach for dynamical mass measurements of galaxy clusters. *The Astrophysical Journal*, *803*(2), 50. https://doi.org/10.1088/0004-637X/803/2/50

Parekh, N., Zou, A., Jungling, I., Endlich, K., Sadowski, J., & Steinhausen, M. (1993). Sex differences in control of renal outer medullary circulation in rats: Role of prostaglandins. *American Journal of Physiology-Renal Physiology*, *264*(4), F629–F636. https://doi.org/10.1152/ajprenal.1993.264.4.F629

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., & others. (2019). Pytorch: An imperative style, high-performance deep learning library. *Advances in Neural Information Processing Systems*, *32*.

Petroff, M. A., Addison, G. E., Bennett, C. L., & Weiland, J. L. (2020). Full-sky cosmic microwave background foreground cleaning using machine learning. *The Astrophysical Journal*, *903*(2), 104. https://doi.org/10.3847/1538-4357/abb9a7

Raissi, M., Perdikaris, P., & Karniadakis, G. E. (2019). Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations. *Journal of Computational Physics*, *378*, 686–707. https://doi.org/10.1016/j.jcp.2018.10.045

Ramos, B., Trost, F., & Thuerey, N. (2022). Control of two-way coupled fluid systems with differentiable solvers. *ICLR 2022 Workshop on Generalizable Policy Learning in Physical World*. https://doi.org/10.48550/arXiv.2206.00342

Rauber, J., Bethge, M., & Brendel, W. (2020). *EagerPy: Writing code that works natively with PyTorch, TensorFlow, JAX, and NumPy*. arXiv. https://arxiv.org/abs/2008.04175

Rodriguez-Galiano, V., Sanchez-Castillo, M., Chica-Olmo, M., & Chica-Rivas, M. (2015). Machine learning predictive models for mineral prospectivity: An evaluation of neural networks, random forest, regression trees and support vector machines. *Ore Geology Reviews*, *71*, 804–818. https://doi.org/10.1016/j.oregeorev.2015.01.001

Rolnick, D., Donti, P. L., Kaack, L. H., Kochanski, K., Lacoste, A., Sankaran, K., Ross, A. S., Milojevic-Dupont, N., Jaques, N., Waldman-Brown, A., & others. (2022). Tackling climate change with machine learning. *ACM Computing Surveys (CSUR)*, *55*(2), 1–96.

Schnell, P., Holl, P., & Thuerey, N. (2021). Half-inverse gradients for physical deep learning. *International Conference on Learning Representations*.

Sengar, V., Seemakurthy, K., Gubbi, J., & P, B. (2021). Multi-task learning based approach for surgical video desmoking. *Proceedings of the Twelfth Indian Conference on Computer Vision, Graphics and Image Processing*, 1–9. https://doi.org/10.1145/3490035.3490283

Takamoto, M., Praditia, T., Leiteritz, R., MacKinlay, D., Alesiani, F., Pflüger, D., & Niepert, M. (2022). PDEBench: An extensive benchmark for scientific machine learning. *Advances in Neural Information Processing Systems*, *35*, 1596–1611.

Thuerey, N., Holl, P., Mueller, M., Schnell, P., Trost, F., & Um, K. (2022). *Physics-based deep learning*. arXiv. https://arxiv.org/abs/2109.05237

Um, K., Brand, R., Fei, Y. R., Holl, P., & Thuerey, N. (2020). Solver-in-the-loop: Learning from differentiable physics to interact with iterative PDE-solvers. *Advances in Neural Information Processing Systems*, *33*, 6111–6122.

Vamathevan, J., Clark, D., Czodrowski, P., Dunham, I., Ferran, E., Lee, G., Li, B., Madabhushi, A., Shah, P., Spitzer, M., & others. (2019). Applications of machine learning in drug discovery and development. *Nature Reviews Drug Discovery*, *18*(6), 463–477. https://doi.org/10.1038/s41573-019-0024-5

Van Rossum, G., & Drake, F. L. (2009). *Python 3 reference manual*. CreateSpace. ISBN: 1441412697

Wandel, N., Weinmann, M., & Klein, R. (2020). Learning incompressible fluid dynamics from scratch-towards fast, differentiable fluid models that generalize. *International Conference on Learning Representations*.

Wandel, N., Weinmann, M., & Klein, R. (2021). Teaching the incompressible Navier–Stokes equations to fast neural surrogate models in three dimensions. *Physics of Fluids*, *33*(4). https://doi.org/10.1063/5.0047428

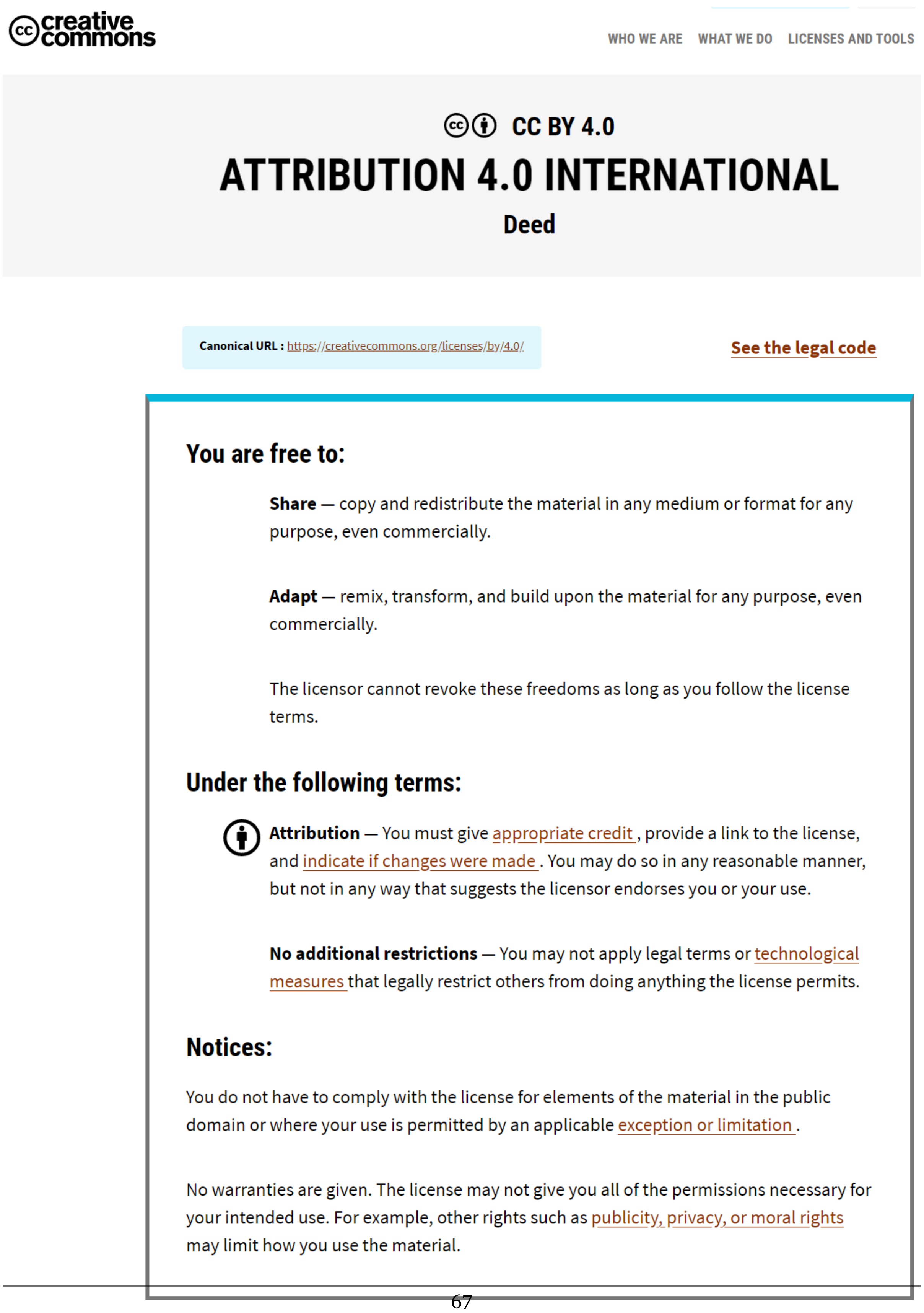
Wang, P. (2023). The applications of generative adversarial network in surgical videos. *Third International Conference on Intelligent Computing and Human-Computer Interaction (ICHCI 2022)*, *12509*, 300–305. https://doi.org/10.1117/12.2656026

Wang, R., Walters, R., & Yu, R. (2022a). Approximately equivariant networks for imperfectly symmetric dynamics. *International Conference on Machine Learning*, 23078–23091.

Wang, R., Walters, R., & Yu, R. (2022b). Meta-learning dynamics forecasting using task inference. *Advances in Neural Information Processing Systems*, *35*, 21640–21653.

Wei, J., Chu, X., Sun, X.-Y., Xu, K., Deng, H.-X., Chen, J., Wei, Z., & Lei, M. (2019). Machine learning in materials science. *InfoMat*, *1*(3), 338–358. https://doi.org/10.1002/inf2.12028

Wu, T., Maruyama, T., & Leskovec, J. (2022). Learning to accelerate partial differential equations via latent global evolution. *Advances in Neural Information Processing Systems*, *35*, 2240–2253.



WHO WE ARE WHAT WE DO LICENSES AND TOOLS

CC BY 4.0

ATTRIBUTION 4.0 INTERNATIONAL

Deed

Canonical URL : https://creativecommons.org/licenses/by/4.0/

See the legal code

You are free to:

Share — copy and redistribute the material in any medium or format for any purpose, even commercially.

Adapt — remix, transform, and build upon the material for any purpose, even commercially.

The licensor cannot revoke these freedoms as long as you follow the license terms.

Under the following terms:

Attribution — You must give appropriate credit , provide a link to the license, and indicate if changes were made . You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use.

No additional restrictions — You may not apply legal terms or technological measures that legally restrict others from doing anything the license permits.

Notices:

You do not have to comply with the license for elements of the material in the public domain or where your use is permitted by an applicable exception or limitation .

No warranties are given. The license may not give you all of the permissions necessary for your intended use. For example, other rights such as publicity, privacy, or moral rights may limit how you use the material.

Scale-invariant Learning by Physics Inversion

Philipp Holl [*]
Technical University of Munich

Vladlen Koltun
Apple

Nils Thuerey
Technical University of Munich

Abstract

Solving inverse problems, such as parameter estimation and optimal control, is a vital part of science. Many experiments repeatedly collect data and rely on machine learning algorithms to quickly infer solutions to the associated inverse problems. We find that state-of-the-art training techniques are not well-suited to many problems that involve physical processes. The highly nonlinear behavior, common in physical processes, results in strongly varying gradients that lead first-order optimizers like SGD or Adam to compute suboptimal optimization directions. We propose a novel hybrid training approach that combines higher-order optimization methods with machine learning techniques. We take updates from a scale-invariant inverse problem solver and embed them into the gradient-descent-based learning pipeline, replacing the regular gradient of the physical process. We demonstrate the capabilities of our method on a variety of canonical physical systems, showing that it yields significant improvements on a wide range of optimization and learning problems.

1 Introduction

Inverse problems that involve physical systems play a central role in computational science. This class of problems includes parameter estimation [54] and optimal control [57]. Among others, solving inverse problems is integral in detecting gravitational waves [22], controlling plasma flows [38], searching for neutrinoless double-beta decay [3, 1], and testing general relativity [19, 35].

Decades of research in optimization have produced a wide range of iterative methods for solving inverse problems [47]. Higher-order methods such as limited-memory BFGS [36] have been especially successful. Such methods compute or approximate the Hessian of the optimization function in addition to the gradient, allowing them to locally invert the function and find stable optimization directions. Gradient descent, in contrast, only requires the first derivative but converges more slowly, especially in ill-conditioned settings [49].

Despite the success of iterative solvers, many of today's experiments rely on machine learning methods, and especially deep neural networks, to find unknown parameters given the observations [11, 15, 22, 3]. While learning methods typically cannot recover a solution up to machine precision, they have a number of advantages over iterative solvers. First, their computational cost for inferring a solution is usually much lower than with iterative methods. This is especially important in time-critical applications, such as the search for rare events in data sets comprising of billions of individual recordings for collider physics. Second, learning-based methods do not require an initial guess to solve a problem. With iterative solvers, a poor initial guess can prevent convergence to a global optimum or lead to divergence (see Appendix C.1). Third, learning-based solutions can be less prone to finding local optima than iterative methods because the parameters are shared across a large collection of problems [30]. Combined with the stochastic nature of the training process, this allows gradients from other problems to push a prediction out of the basin of attraction of a local optimum.

[*]Corresponding author, `philipp.holl@tum.de`

Practically all state-of-the-art neural networks are trained using only first order information, mostly due to the computational cost of evaluating the Hessian w.r.t. the network parameters. Despite the many breakthroughs in the field of deep learning, the fundamental shortcomings of gradient descent persist and are especially pronounced when optimizing non-linear functions such as physical systems. In such situations, the gradient magnitudes often vary strongly from example to example and parameter to parameter.

In this paper, we show that inverse physics solvers can be embedded into the traditional deep learning pipeline, resulting in a hybrid training scheme that aims to combine the fast convergence of higher-order solvers with the low computational cost of backpropagation for network training. Instead of using the adjoint method to backpropagate through the physical process, we replace that gradient by the update computed from a higher-order solver which can encode the local nonlinear behavior of the physics. These physics updates are then passed on to a traditional neural network optimizer which computes the updates to the network weights using backpropagation. Thereby our approach maintains compatibility with acceleration schemes [18, 33] and stabilization techniques [32, 31, 7] developed for training deep learning models. The replacement of the physics gradient yields a simple mathematical formulation that lends itself to straightforward integration into existing machine learning frameworks.

In addition to a theoretical discussion, we perform an extensive empirical evaluation on a wide variety of inverse problems including the highly challenging Navier-Stokes equations. We find that using higher-order or domain-specific solvers can drastically improve convergence speed and solution quality compared to traditional training without requiring the evaluation of the Hessian w.r.t. the model parameters.

2 Scale-invariance in Optimization

We consider unconstrained inverse problems that involve a differentiable physical process $\mathcal{P} : X \subset \mathbb{R}^{d_x} \to Y \subset \mathbb{R}^{d_y}$ which can be simulated. Here X denotes the physical parameter space and Y the space of possible observed outcomes. Given an observed or desired output $y^* \in Y$, the inverse problem consists of finding optimal parameters

$$x^* = \arg\min_x L(x) \quad \text{with} \quad L(x) = \frac{1}{2}\|\mathcal{P}(x) - y^*\|_2^2. \tag{1}$$

Such problems are classically solved by starting with an initial guess x_0 and iteratively applying updates $x_{k+1} = x_k + \Delta x_k$. Newton's method [5] and many related methods [10, 36, 23, 41, 46, 8, 12, 6] approximate L around x_k as a parabola $\tilde{L}(x) = L(x_k) + \frac{\partial L(x_k)}{\partial x_k}(x - x_k) + \frac{1}{2}H_k(x - x_k)^2$ where H_k denotes the Hessian or an approximation thereof. Inverting $\tilde{L}$ and walking towards its minimum with step size η yields

$$\Delta x_k = -\eta \cdot H_k^{-1} \left(\frac{\partial L(x_k)}{\partial x_k}\right)^T. \tag{2}$$

The inversion of H results in scale-invariant updates, i.e. when rescaling x or any component of x, the optimization will behave the same way, leaving $L(x_k)$ unchanged. An important consequence of scale-invariance is that minima will be approached equally quickly in terms of L no matter how wide or deep they are.

Newton-type methods have one major downside, however. The inversion depends on the Hessian H which is expensive to compute exactly, and hard to approximate in typical machine learning settings with high-dimensional parameter spaces [25] and mini-batches [51].

Instead, practically all state-of-the-art deep learning relies on first-order information only. Setting H to the identity in Eq. 2 yields gradient descent updates $\Delta x = -\eta \cdot \left(\frac{\partial L}{\partial x}\right)^T$ which are not scale-invariant. Rescaling x by λ also scales Δx by λ, inducing a factor of λ^2 in the first-order-accurate loss change $L(x) - L(x + \Delta x) = -\eta \cdot (\frac{\partial L}{\partial x})^2 + \mathcal{O}(\Delta x^2)$. Gradient descent prescribes small updates to parameters that require a large change to decrease L and vice-versa, typically resulting in slower convergence than Newton updates [56]. This behavior is the root cause of exploding or vanishing gradients in deep neural networks. The step size η alone cannot remedy this behavior whenever $\frac{\partial L}{\partial x}$

varies along x. Figure 1 shows the optimization trajectories for the simple problem $\mathcal{P}(x) = (x_1, x_2^2)$ to illustrate this problem.

When training a neural network, the effect of scaling-variance can be reduced through normalization in intermediate layers [32, 7] and regularization [37, 53] but this level of control is not present in most other optimization tasks, such as inverse problems (Eq. 1). More advanced first-order optimizers try to solve the scaling issue by approximating higher-order information [33, 29, 18, 39, 40, 55, 45, 50], such as Adam where $H \approx \operatorname{diag}\left(|\frac{\partial L}{\partial x}|\right)$, decreasing the loss scaling factor from λ^2 to λ. However, these methods lack the exact higher-order information which limits the accuracy of the resulting update steps when optimizing nonlinear functions.

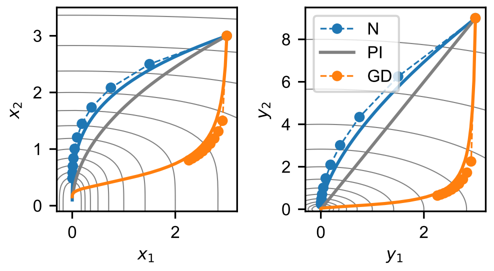


Figure 1: Minimization (Eq. 1) with $y \equiv \mathcal{P}(x) = (x_1, x_2^2)$. L contours in gray. Trajectories of gradient descent (GD), Newton's method (N), and perfect physics inversion (PI) shown as lines (infinitesimal η) and circles (10 iterations with constant η).

3 Scale-invariant Physics and Deep Learning

We are interested in finding solutions to Eq. 1 using a neural network, $x^* = \mathrm{NN}(y^* \mid \theta)$, parameterized by θ. Let $\mathcal{Y}^* = \{y_i^* \mid i = 1, ..., N\}$ denote a set of N inverse problems involving $\mathcal{P}$. Then training the network means finding

$$\theta_* = \arg\min_\theta \sum_{i=1}^{N} \frac{1}{2} \|\mathcal{P}\big(\mathrm{NN}(y_i^* \mid \theta)\big) - y_i^*\|_2^2 \,. \tag{3}$$

Assuming a large parameter space θ and the use of mini-batches, higher-order methods are difficult to apply to this problem, as described above. Additionally, the scale-variance issue of gradient descent is especially pronounced in this setting because only the network part of the joint problem $\mathrm{NN} \circ \mathcal{P}$ can be properly normalized while the physical process $\mathcal{P}$ is fixed. Therefore, the traditional approach of computing the gradient $\frac{\partial L}{\partial \theta}$ using the adjoint method (backpropagation) can lead to undesired behavior.

Consider the problem $\mathcal{P}(x) = e^x$ with observed data $y^* \in (0, 1]$ (Appendix C.2). Due to the exponential form of $\mathcal{P}$, the curvature around the solutions $x^* = \log(y^*)$ strongly depends on y^*. This causes first-order network optimizers such as SGD or Adam to fail in approximating the correct solutions for small y^* because their gradients are overshadowed by larger y^* (see Fig. 2). Scaling the gradients to unit length in x drastically improves the prediction accuracy, which hints at a possible solution: If we could employ a scale-invariant physics solver, we would be able to optimize all examples, independent of the curvature around their respective minima.

3.1 Derivation

If $\mathcal{P}$ has a unique inverse and is sufficiently well-behaved, we can split the joint optimization problem (Eq. 3) into two stages. First, solve all inverse problems individually, constructing a set of unique solutions $\mathcal{X}^{\mathrm{sv}} = \{x_i^{\mathrm{sv}} = \mathcal{P}^{-1}(y_i^*)\}$ where $\mathcal{P}^{-1}$ denotes the inverse problem solver. Second, use $\mathcal{X}^{\mathrm{sv}}$ as labels for supervised training

$$\theta_* = \arg\min_\theta \sum_{i=1}^{N} \frac{1}{2} \|\mathrm{NN}(y_i^* \mid \theta) - x_i^{\mathrm{sv}}\|_2^2. \tag{4}$$

This enables scale-invariant physics (SIP) inversion while a fast first-order method can be used to train the network which can be constructed to be normalized using state-of-the-art procedures [32, 7, 53].

Unfortunately, this two-stage approach is not applicable in multimodal settings, where x^{sv} depends on the initial guess x_0 used in the first stage. This would cause the network to interpolate between possible solutions, leading to subpar convergence and generalization performance. To avoid these problems, we alter the training procedure from Eq. 4 in two ways. First, we reintroduce $\mathcal{P}^{-1}$ into the training loop, yielding

$$\theta_* = \arg\min_\theta \sum_{i=1}^{N} \frac{1}{2} \| \mathrm{NN}(y_i^* \,|\, \theta) - \mathcal{P}^{-1}(y_i^*) \|_2^2 \,. \tag{5}$$

Next, we condition $\mathcal{P}^{-1}$ on the neural network prediction by using it as an initial guess, $\mathcal{P}^{-1}(y^*) \to \mathcal{P}^{-1}(y^* \,|\, \mathrm{NN}(y^* \,|\, \theta))$. This makes training in multimodal settings possible because the embedded solver $\mathcal{P}^{-1}$ searches for minima close to the prediction of the NN. Therefore θ can exit the basin of attraction of other minima and does not need to interpolate between possible solutions. Also, since all inverse problems from $\mathcal{Y}^*$ are optimized jointly, this reduces the likelihood of any individual solution getting stuck in a local minimum, as discussed earlier.

The obvious downside to this approach is that $\mathcal{P}^{-1}$ must be run for each training step. When $\mathcal{P}^{-1}$ is an iterative solver, we can write it as $P^{-1}(y^* \,|\, \mathrm{NN}(y^* \,|\, \theta)) = \mathrm{NN}(y^* \,|\, \theta) + \Delta x_0 + ... + \Delta x_n$. We denote the first update $\Delta x_0 \equiv U(y^* \,|\, \theta)$.

Instead of computing all updates Δx, we approximate P^{-1} with its first update U. Inserting this into Eq. 5 with $(\circ) \equiv (y_i^* \,|\, \theta)$ yields

$$\theta_* = \arg\min_\theta \sum_{i=1}^{N} \frac{1}{2} \| \mathrm{NN}(y_i^* \,|\, \theta) - (\mathrm{NN}(\circ) + U(\circ)) \|_2^2 . \tag{6}$$

This can be further simplified to $\sum_{i=1}^{N} \frac{1}{2} \| U(y_i^* \,|\, \theta) \|_2^2$ but this form is hard to optimize directly as is requires backpropagation through U. In addition, its minimization is not sufficient, because all fixed points of U, such as maxima or saddle points of L, can act as attractors.

Instead, we make the assumption $\frac{\partial \mathcal{P}^{-1}}{\partial y} = 0$ to remove the $(\circ)$ dependencies in Eq. 6, treating them as constant. This results in a simple L^2 loss for the network output. As we will show, this avoids both issues. It also allows us to factor the optimization into a network and a physics graph (see Fig. 3), so that all necessary derivative computations only require data from one of them.

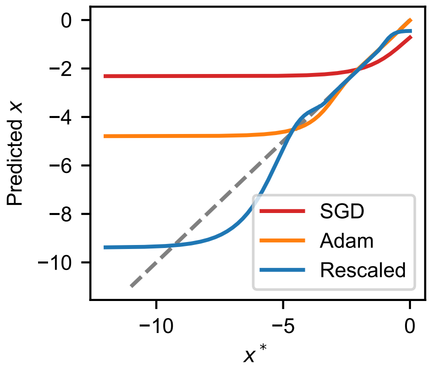


Figure 2: Networks trained according to Eq. 3 with $\mathcal{P}(x) = e^x$. Stochastic gradient descent (SGD) and Adam fail to approximate solutions for small values due to scale variance. Normalizing the gradient in x space (Rescaled) improves solution accuracy by decreasing scale variance.

3.2 Update Rule

The factorization described above results in the following update rule for training with SIP updates, shown in Fig. 3:

1. Pass the network prediction x_0 to the physics graph and compute $\Delta x_0 \equiv U(y_i^* \,|\, x_0)$ for all examples in the mini-batch.
2. Send $\tilde{x} \equiv x_0 + \Delta x_0$ back and compute the network loss $\tilde{L} = \frac{1}{2}||x_0 - \tilde{x}||_2^2$.
3. Update θ using any neural network optimizer, such as SGD or Adam, treating $\tilde{x}$ as a constant.

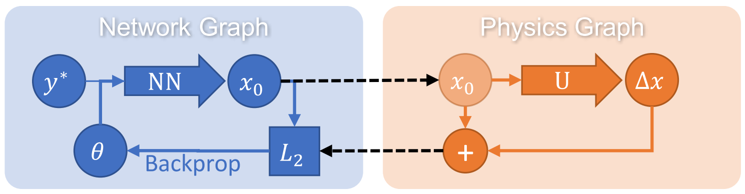


Figure 3: Neural network (NN) training procedure with embedded inverse physics solver (U).

To see what updates $\Delta\theta$ result from this algorithm, we compute the gradient w.r.t. θ,

$$\frac{\partial \tilde{L}}{\partial \theta} = \sum_{i=1}^{N} U(y^* \,|\, x_0) \cdot \frac{\partial \mathrm{NN}}{\partial \theta} \tag{7}$$

Comparing this to the gradient resulting from optimizing the joint problem $\mathrm{NN} \circ \mathcal{P}$ with a single optimizer (Eq. 3),

$$\frac{\partial L}{\partial \theta} = \Delta y_i \cdot \frac{\partial \mathcal{P}}{\partial x} \frac{\partial \mathrm{NN}}{\partial \theta} \tag{8}$$

where $\Delta y_i = \mathcal{P}\big(\mathrm{NN}(y_i^* \,|\, \theta)\big) - y_i^*$, we see that U takes the place of $\Delta y_i \cdot \frac{\partial \mathcal{P}}{\partial x}$, the adjoint vector that would otherwise be computed by backpropagation. Unlike the adjoint vector, however, $\tilde{x} = x_0 + U$ encodes an actual physical configuration. Since $\tilde{x}$ can stem from a higher-order solver, the resulting updates $\Delta\theta$ can also encode non-linear information about the physics without computing the Hessian w.r.t. θ.

3.3 Convergence

It is a priori not clear whether SIP training will converge, given that $\frac{\partial \mathcal{P}^{-1}}{\partial y}$ is not computed. We start by proving convergence for two special choices of the inverse solver $\mathcal{P}^{-1}$ before considering the general case. We assume that NN is expressive enough to fit our problem and that it is able to converge to every point x for all examples using gradient descent:

$$\exists \eta > 0 \,:\, \forall i \;\forall x \;\forall \epsilon > 0 \;\exists n \in \mathbb{N} : ||\mathrm{NN}_{\theta_n} - x||_2 \leq \epsilon \tag{9}$$

where θ_n is the sequence of gradient descent steps

$$\theta_{n+1} = \theta_n - \eta \left(\frac{\partial \mathrm{NN}}{\partial \theta} \right)^T (\mathrm{NN}_{\theta_n} - x) \tag{10}$$

with $\eta > 0$. Large enough networks fulfill this property under certain assumptions [17] and the universal approximation theorem guarantees that such a configuration exists [13].

In the first special case, $U(x) \equiv U(y^* \,|\, x)$ points directly towards a solution x^*. This models the case of a known ground truth solution in a unimodal setting. For brevity, we will drop the example indices and the dependencies on y^*.

Theorem 1. *If* $\forall x \;\exists \lambda \in (0, 1] \;:\; U(x) = \lambda(x^* - x)$, *then the gradient descent sequence* NN_{θ_n} *with* $\theta_{n+1} = \theta_n + \eta \left(\frac{\partial \mathrm{NN}}{\partial \theta} \right)^T U(x)$ *converges to* x^*.

Proof. Rewriting $U(x) = -\frac{\partial}{\partial x} \left(\frac{\lambda}{2} ||x - x^*||_2^2 \right)$ yields the update $\theta_{n+1} - \theta_n = -\eta \left(\frac{\partial \hat{L}}{\partial x} \frac{\partial \mathrm{NN}}{\partial \theta} \right)^T$ where $\hat{L} = \frac{\lambda}{2} ||x - x^*||_2^2$. This describes gradient descent towards x^* with the gradients scaled by λ. Since $\lambda \in (0, 1]$, the convergence proof of gradient descent applies. □

The second special case has $U(x)$ pointing in the direction of steepest gradient descent in x space.

Theorem 2. *If* $\forall x \;\exists \lambda \in (0, 1] \;:\; U(x) = -\lambda \left(\frac{\partial L}{\partial x} \right)^T$, *then the gradient descent sequence* NN_{θ_n} *with* $\theta_{n+1} = \theta_n + \eta \left(\frac{\partial \mathrm{NN}}{\partial \theta} \right)^T U(x)$ *converges to minima of L.*

Proof. This is equivalent to gradient descent in $L(\theta) \equiv (\mathrm{NN} \circ L)(\theta)$. Rewriting the update yields $\theta_{n+1} - \theta_n = -\eta\lambda \left(\frac{\partial L}{\partial x} \frac{\partial \mathrm{NN}}{\partial \theta} \right)^T$ which is the gradient descent update scaled by λ. □

Next, we consider the general case of an arbitrary $\mathcal{P}^{-1}$ and U. We require that U decreases L by a minimum relative amount specified by τ,

$$\exists \tau > 0 \;:\; \forall x \;:\; L(x) - L(x + U(x)) \geq \tau \left(L(x) - L(x^*) \right). \tag{11}$$

To guarantee convergence to a region, we also require

$$\exists K > 0 \;:\; \forall x \;:\; ||U(x)|| \leq K(L(x) - L(x^*)). \tag{12}$$

Theorem 3. *There exists an update strategy $\theta_{n+1} = S(\theta_n)$ based on a single evaluation of U for which $L(\mathrm{NN}_{\theta_n}(y))$ converges to a minimum x^* or minimum region of L for all examples.*

Proof. We denote $\tilde{x}_n \equiv x_n + U(x_n)$ and $\Delta L^* = L(x_n) - L(x^*)$. Let I_2 denote the open set of all x for which $L(x) - L(x_n) > \frac{\tau}{2}\Delta L^*$. Eq. 11 provides that $\tilde{x}_n \in I_2$. Since I_2 is open, $\exists \epsilon > 0 : \forall x \in B_\epsilon(\tilde{x}_n) : L(x_n) - L(x) > \frac{\tau}{2}\Delta L^*$ where $B_\epsilon(x)$ denotes the open set containing all $x' \in \mathbb{R}^d$ for which $||x' - x||_2 < \epsilon$, i.e. there exists a small ball around $\tilde{x}_n$ which is fully contained in I_2 (see sketch in Fig. 4).

Using Eq. 9, we can find a finite $n \in \mathbb{N}$ for which $\mathrm{NN}_{\theta_n} \in B_\epsilon(\tilde{x}_n)$ and therefore $L(x_n) - L(\mathrm{NN}_{\theta_n}) > \frac{\tau}{2}\Delta L^*$. We can thus use the following strategy S for minimizing L: First, compute $\tilde{x}_n = x_n + U(x_n)$. Then perform gradient descent steps in θ with the effective objective function $\frac{1}{2}||\mathrm{NN}_\theta - \tilde{x}_n||_2^2$ until $L(x_n) - L(\mathrm{NN}_\theta) \geq \frac{\tau}{2}\Delta L^*$. Each application of S reduces the loss to $\Delta L^*_{n+1} \leq (1 - \frac{\tau}{2})\Delta L^*_n$ so any value of $L > L(x^*)$ can be reached within a finite number of steps. Eq. 12 ensures that $||U(x)|| \to 0$ as the optimization progresses which guarantees that the optimization converges to a minimum region. □

While this theorem guarantees convergence, it requires potentially many gradient descent steps in θ for each physics update U. This can be advantageous in special circumstances, e.g. when U is more expensive to compute than an update in θ, or when θ is far away from a solution. However, in many cases, we want to re-evaluate U after each update to θ. Without additional assumptions about U and NN_θ, there is no guarantee that L will decrease every iteration, even for infinitesimally small step sizes. Despite this, there is good reason to assume that the optimization will decrease L over time. This can be seen visually in Fig. 4 where the next prediction x_{n+1} is guaranteed to lie within the blue circle. The region of increasing loss is shaded orange and always fills less than half of the volume of the circle, assuming we choose a sufficiently small step size. We formalize this argument in appendix A.2.

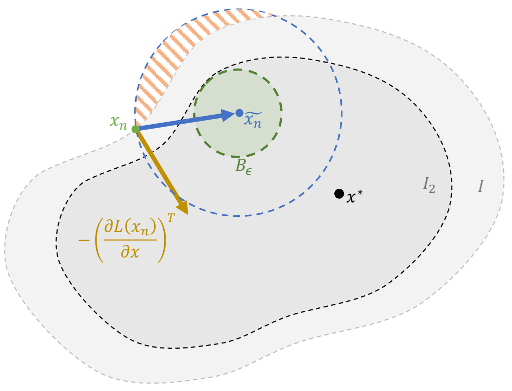


Figure 4: Convergence visualization for the proof of theorem 3. All shown objects are visualized in x space for one example i.

Remarks and lemmas We considered the case that $U \equiv \Delta x_0$. This can be trivially extended to the case that $U \equiv \Delta x_0 + ... + \Delta x_m$ for any $m \in \mathbb{N}$. When we let the solver run to convergence, i.e. m large enough, theorem 1 guarantees convergence if $\mathcal{P}^{-1}$ consistently converges to the same x_i^*. Also note that any minimum θ^* found with SIP training fulfills $U = 0$ for all examples, i.e. we are implicitly minimizing $\sum_{i=1}^N \frac{1}{2}||U(y_i^* \mid \theta)||_2^2$.

3.4 Experimental Characterization

We first investigate the convergence behavior of SIP training depending on characteristics of $\mathcal{P}$. We construct the synthetic two-dimensional inverse process

$$\mathcal{P}(x) = (\sin(\hat{x}_1)/\xi,\ \hat{x}_2 \cdot \xi) \quad \text{with} \quad \hat{x} = \gamma \cdot R_\phi \cdot x,$$

where $R_\phi \in \mathrm{SO}(2)$ denotes a rotation matrix and $\gamma > 0$. The parameters ξ and ϕ allow us to continuously change the characteristics of the system. The value of ξ determines the conditioning of $\mathcal{P}$ with large ξ representing ill-conditioned problems while ϕ describes the coupling of x_1 and x_2. When $\phi = 0$, the off-diagonal elements of the Hessian vanish and the problem factors into two independent problems. Fig. 5a shows one example loss landscape.

We train a fully-connected neural network to invert this problem (Eq. 3), comparing SIP training using a saddle-free Newton solver [14] to various state-of-the-art network optimizers. We select the best learning rate for each optimizer independently. For $\xi = 0$, when the problem is perfectly conditioned, all network optimizers converge, with Adam converging the quickest (Fig. 5b). Note

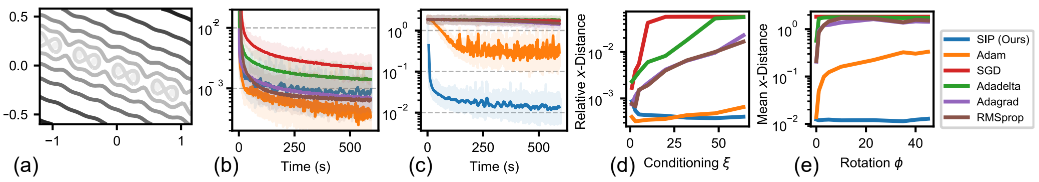


Figure 5: (a) Example loss landscape with $y^* = (0.3, -0.5)$, $\xi = 1$, $\phi = 15°$. (b,c) Learning curves with $\phi = \frac{\pi}{4}$, averaged over 256 min-batches. For (b) $\xi = 1$, and (c) $\xi = 32$. (d) Dependence on problem conditioning ξ (with $\phi = 0$). (e) Dependence on parameter coupling ϕ (with $\xi = 32$).

that the relatively slow convergence of SIP mostly stems from it taking significantly more time per iteration than the other methods, on average 3 times as long as Adam. As we have spent little time in eliminating computational overheads, SIP performance could likely be significantly improved to near-Adam performance.

When increasing ξ with $\phi = 0$ fixed (Fig. 5d), the accuracy of all traditional network optimizers decreases because the gradients scale with $(1/\xi, \xi)$ in x, elongating in x_2, the direction that requires more precise values. SIP training uses the Hessian to invert the scaling behavior, producing updates that align with the flat direction in x to avoid this issue. This allows SIP training to retain its relative accuracy over a wide range of ξ. At $\xi = 32$, only SIP and Adam succeed in optimizing the network to a significant degree (Fig. 5c).

Varying ϕ with $\xi = 32$ fixed (Fig. 5e) sheds light on how Adam manages to learn in ill-conditioned settings. Its diagonal approximation of the Hessian reduces the scaling effect when x_1 and x_2 lie on different scales, but when the parameters are coupled, the lack of off-diagonal terms prevents this. SIP training has no problem with coupled parameters since its updates are based on the full-rank Hessian $\frac{\partial^2 L}{\partial x^2}$.

3.5 Application to High-dimensional Problems

Explicitly evaluating the Hessian is not feasible for high-dimensional problems. However, scale-invariant updates can still be computed, e.g. by inverting the gradient or via domain knowledge. We test SIP training on three high-dimensional physical systems described by partial differential equations: Poisson's equation, the heat equation, and the Navier-Stokes equations. This selection covers ubiquitous physical processes with diffusion, transport, and strongly non-local effects, featuring both explicit and implicit solvers. All code required to reproduce our results is available at `https://github.com/tum-pbs/SIP`. A detailed description of all experiments along with additional visualizations and performance measurements can be found in appendix B.

Poisson's equation Poisson's equation, $\mathcal{P}(x) = \nabla^{-2}x$, plays an important role in electrostatics, Newtonian gravity, and fluid dynamics [4]. It has the property that local changes in x can affect $\mathcal{P}(x)$ globally. Here we consider a two-dimensional system and train a U-net [48] to solve inverse problems (Eq. 1) on pseudo-randomly generated y^*. We compare SIP training to SGD with momentum, Adam, AdaHessian [55], Fourier neural operators (FNO) [34] and Hessian-free optimization (H-free) [39]. Fig. 6b shows the learning curves. The training with SGD, Adam and AdaHessian drastically slows within the first 300 iterations. FNO and H-free both improve upon this behavior, reaching twice the accuracy before slowing. For SIP, we construct scale-invariant Δx based on the analytic inverse of Poisson's equation and use Adam to compute $\Delta\theta$. The curve closely resembles an exponential curve, which indicates linear convergence, the ideal case for first-order methods optimizing an L_2 objective. During all of training, the SIP variant converges exponentially faster than the traditional optimizers, its relative performance difference compared to Adam continually increasing from a factor of 3 at iteration 60 to a full order of magnitude after 5k iterations. This difference can be seen in the inferred solutions (Fig. 6a) which are noticeably more detailed.

Heat equation Next, we consider a system with fundamentally non-invertible dynamics. The heat equation, $\frac{\partial u}{\partial t} = \nu \cdot \nabla^2 u$, models heat flow in solids but also plays a part in many diffusive systems [16]. It gradually destroys information as the temperature equilibrium is approached [26], causing $\nabla\mathcal{P}$ to become near-singular. Inspired by heat conduction in microprocessors, we generate

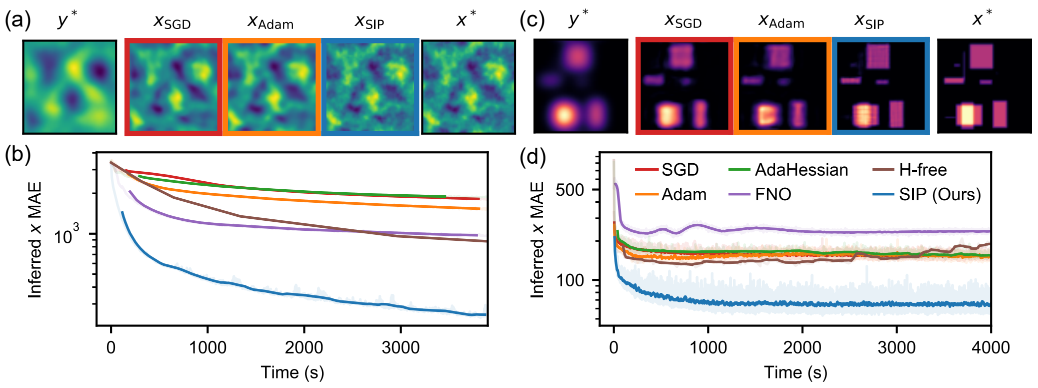


Figure 6: Poisson's equation (left) and the heat equation (right). (a,c) Example from the data set: observed distribution (y^*), inferred solutions, and ground truth solution (x^*). (b,d) Learning curves, running average over 64 mini-batches (except for H-free).

examples x_{GT} by randomly scattering four to ten heat generating rectangular regions on a plane and simulating the heat profile $y^* = \mathcal{P}(x_{\mathrm{GT}})$ as observed from outside a heat-conducting casing. The learning curves for the corresponding inverse problem are shown in Fig. 6d.

When training with SGD, Adam or AdaHessian, we observe that the distance to the solution starts rapidly decreasing before decelerating between iterations 30 and 40 to a slow but mostly stable convergence. The sudden deceleration is rooted in the adjoint problem, which is also a diffusion problem. Backpropagation through $\mathcal{P}$ removes detail from the gradients, which makes it hard for first-order methods to recover the solution. H-free initially finds better solutions but then stagnates with the solution quality slowly deteriorating. FNO performs poorly on this task, likely due to the sharp edges in x^*.

The information loss in $\mathcal{P}$ prevents direct numerical inversion of the gradient or Hessian. Instead, we add a dampening term to derive a numerically stable scale-invariant solver which we use for SIP training. Unlike SGD and Adam, the convergence of SIP training does not decelerate as early as the other methods, resulting in an exponentially faster convergence. At iteration 100, the predictions are around 34% more accurate compared to Adam, and the difference increases to 130% after 10k iterations, making the reconstructions noticeably sharper than with traditional training methods (Fig. 6c). To test the dependence of SIP on hyperparameters like batch size or learning rate, we perform this experiment with a range of values (see appendix B.3). Our results indicate that SIP training and Adam are impacted the same way by non-optimal hyperparameter configurations.

Navier-Stokes equations Fluids and turbulence are among the most challenging and least understood areas of physics due to their highly nonlinear behavior and chaotic nature [21]. We consider a two-dimensional system governed by the incompressible Navier-Stokes equations: $\frac{\partial v}{\partial t} = \nu\nabla^2 v - v \cdot \nabla v - \nabla p$, $\nabla \cdot v = 0$, where p denotes pressure and ν the viscosity. At $t = 0$, a region of the fluid is randomly marked with a massless colorant m_0 that passively moves with the fluid, $\frac{\partial m}{\partial t} = -v \cdot \nabla m$. After time t, the marker is observed again to obtain m_t. The fluid motion is initialized as a superposition of linear motion, a large vortex and small-scale perturbations. An example observation pair $y^* = \{m_0, m_t\}$ is shown in Fig. 7a. The task is to find an initial fluid velocity $x \equiv v_0$ such that the fluid simulation $\mathcal{P}$ matches m_t at time t. Since $\mathcal{P}$ is deterministic, x encodes the complete fluid flow from 0 to t. We define the objective in frequency space with lower frequencies being weighted more strongly. This definition considers the match of the marker distribution on all scales, from the coarse global match to fine details, and is compatible with the definition in Eq. 1. We train a U-net [48] to solve these inverse problems; the learning curves are shown in Fig. 7b.

When training with Adam, the error decreases for the first 100 iterations while the network learns to infer velocities that lead to an approximate match. The error then proceeds to decline at a much lower rate, nearly coming to a standstill. This is caused by an overshoot in terms of vorticity, as visible in Fig. 7a right. While the resulting dynamics can roughly approximate the shape of the observed m_t,

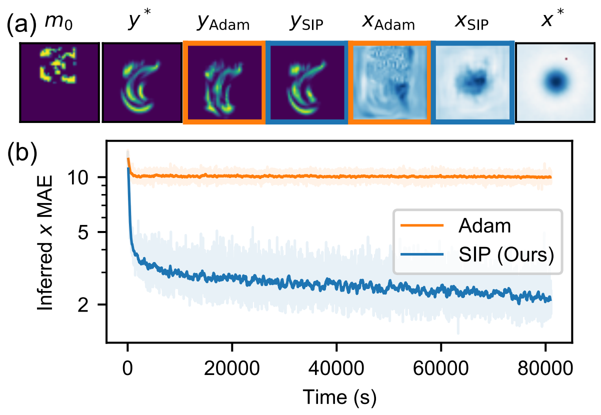


Figure 7: Incompressible fluid flow reconstruction. (a) Example from the data set: initial marker distribution (m_0); simulated marker distribution after time t using ground-truth velocity (y^*) and network predictions ($y_\circ$); predicted initial velocities ($x_\circ$); ground truth velocity (x^*). (b) Learning curves, running average over 64 mini-batches.

they fail to match its detailed structure. Moving from this local optimum to the global optimum is very hard for the network as the distance in x space is large and the gradients become very noisy due to the highly non-linear physics. A similar behavior can also be seen when optimizing single inverse problems with gradient descent where it takes more than 20k iterations for GD to converge on single problems.

For SIP training, we run the simulation in reverse, starting with m_t, to estimate the translation and vortex of x in a scale-invariant manner. When used for network training, we observe vastly improved convergence behavior compared to first-order training. The error rapidly decreases during the first 200 iterations, at which point the inferred solutions are more accurate than pure Adam training by a factor of 2.3. The error then continues to improve at a slower but still exponentially faster rate than first-order training, reaching a relative accuracy advantage of 5x after 20k iterations. To match the network trained with pure Adam for 20k iterations, the SIP variant only requires 55 iterations. This improvement is possible because the inverse-physics solver and associated SIP updates do not suffer from the strongly-varying gradient magnitudes and directions, which drown the first-order signal in noise. Instead, the SIP updates behave much more smoothly, both in magnitude and direction.

Comparing the inferred solutions from the network to an iterative approach shows a large difference in inference time (table 1). To reach the same solution quality as the neural network prediction, $\mathcal{P}_{\text{NS}}^{-1}$ needs 7 iterations on average, which takes more than 10,000 times as long, and gradient descent (GD) does not reach the same quality even after 20k iterations. This difference is caused by $\mathcal{P}_{\text{NS}}^{-1}$ having to run the full forward and backward simulation for each iteration. This cost is also required for each training iteration of the network but once converged, its inference is extremely fast, solving around 9000 problems per second in batch mode. For both iterative solver and network, we used a batch size of 64 and divide the total time by the batch size.

Table 1: Time to reach equal solution quality in the fluid experiment, measured as MAE in x space. The inference time is given per example in batch mode, followed by the number of iterations in parentheses.

Method	Training time	Inference time
NN	17.6 h (15.6 k)	0.11 ms
$\mathcal{P}_{\text{NS}}^{-1}$	n/a	2.2 s (7)
GD	n/a	$>$ 4h (20k)

3.6 Limitations and Discussion

While SIP training manages to find vastly more accurate solutions for the examples above, there are some caveats to consider. First, an approximately scale-invariant physics solver is required. While in low-dimensional x spaces Newton's method is a good candidate, high-dimensional spaces require another form of inversion. Some equations can locally be inverted analytically but for complex problems, domain-specific knowledge may be required. However, this is a widely studied field and many specialized solvers have been developed [9].

Second, SIP uses traditional first-order optimizers to determine $\Delta\theta$. As discussed, these solvers behave poorly in ill-conditioned settings which can also affect SIP performance when the network outputs lie on different scales. Some recent works address this issue and have proposed network optimization based on inversion [39, 40, 20].

Third, while SIP training generally leads to more accurate solutions, measured in x space, the same is not always true for the loss $L = \sum_i L_i$. SIP training weighs all examples equally, independent of the curvature $|\frac{\partial^2 L}{\partial x^2}|$ near a chosen solution. This can cause small errors in examples with large curvatures to dominate L. In these cases, or when the accuracy in x space is not important, like in some control tasks, traditional training methods may perform better than SIP training.

4 Conclusions

We have introduced scale-invariant physics (SIP) training, a novel neural network training scheme for learning solutions to nonlinear inverse problems. SIP training leverages physics inversion to compute scale-invariant updates in the solution space. It provably converges assuming enough network updates $\Delta\theta$ are performed per solver evaluation and we have shown that it converges with a single $\Delta\theta$ update for a wide range of physics experiments. The scale-invariance allows it to find solutions exponentially faster than traditional learning methods for many physics problems while keeping the computational cost relatively low. While this work targets physical processes, SIP training could also be applied to other coupled nonlinear optimization problems, such as differentiable rendering or training invertible neural networks.

Scale-invariant optimizers, such as Newton's method, avoid many of the problems that plague deep learning at the moment. While their application to high-dimensional parameter spaces is currently limited, we hope that our method will help establish them as commonplace tools for training neural networks in the future.

References

[1] Craig E Aalseth, N Abgrall, Estanislao Aguayo, SI Alvis, M Amman, Isaac J Arnquist, FT Avignone III, Henning O Back, Alexander S Barabash, PS Barbeau, et al. Search for neutrinoless double-β decay in ge 76 with the majorana demonstrator. *Physical review letters*, 120(13):132502, 2018.

[2] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*, pages 265–283, 2016.

[3] M Agostini, M Allardt, E Andreotti, AM Bakalyarov, M Balata, I Barabanov, M Barnabé Heider, N Barros, L Baudis, C Bauer, et al. Pulse shape discrimination for gerda phase i data. *The European Physical Journal C*, 73(10):2583, 2013.

[4] William F Ames. *Numerical methods for partial differential equations*. Academic press, 2014.

[5] Kendall E Atkinson. *An introduction to numerical analysis*. John wiley & sons, 2008.

[6] Mordecai Avriel. *Nonlinear programming: analysis and methods*. Courier Corporation, 2003.

[7] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E Hinton. Layer normalization. In *International Conference on Learning Representations (ICLR)*, 2016.

[8] Ernst R Berndt, Bronwyn H Hall, Robert E Hall, and Jerry A Hausman. Estimation and inference in nonlinear structural models. In *Annals of Economic and Social Measurement, Volume 3, number 4*, pages 653–665. NBER, 1974.

[9] A. Borzì, G. Ciaramella, and M. Sprengel. *Formulation and Numerical Solution of Quantum Control Problems*. Society for Industrial and Applied Mathematics, Philadelphia, PA, 2017.

[10] Charles George Broyden. The convergence of a class of double-rank minimization algorithms 1. general considerations. *IMA Journal of Applied Mathematics*, 6(1):76–90, 1970.

[11] Giuseppe Carleo, Ignacio Cirac, Kyle Cranmer, Laurent Daudet, Maria Schuld, Naftali Tishby, Leslie Vogt-Maranto, and Lenka Zdeborová. Machine learning and the physical sciences. *Reviews of Modern Physics*, 91(4):045002, 2019.

[12] Andrew R Conn, Nicholas IM Gould, and Ph L Toint. Convergence of quasi-newton matrices generated by the symmetric rank one update. *Mathematical programming*, 50(1-3):177–195, 1991.

[13] George Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of control, signals and systems*, 2(4):303–314, 1989.

[14] Yann Dauphin, Razvan Pascanu, Caglar Gulcehre, Kyunghyun Cho, Surya Ganguli, and Yoshua Bengio. Identifying and attacking the saddle point problem in high-dimensional non-convex optimization. *arXiv preprint arXiv:1406.2572*, 2014.

[15] S Delaquis, MJ Jewell, I Ostrovskiy, M Weber, T Ziegler, J Dalmasson, LJ Kaufman, T Richards, JB Albert, G Anton, et al. Deep neural networks for energy and position reconstruction in exo-200. *Journal of Instrumentation*, 13(08):P08023, 2018.

[16] Jerome Droniou. Finite volume schemes for diffusion equations: introduction to and review of modern methods. *Mathematical Models and Methods in Applied Sciences*, 24(08):1575–1619, 2014.

[17] Simon S Du, Xiyu Zhai, Barnabas Poczos, and Aarti Singh. Gradient descent provably optimizes over-parameterized neural networks. *arXiv preprint arXiv:1810.02054*, 2018.

[18] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of machine learning research*, 12(7), 2011.

[19] F. W. Dyson, A. S. Eddington, and C. Davidson. A Determination of the Deflection of Light by the Sun's Gravitational Field, from Observations Made at the Total Eclipse of May 29, 1919, January 1920.

[20] Israel Elias, José de Jesús Rubio, David Ricardo Cruz, Genaro Ochoa, Juan Francisco Novoa, Dany Ivan Martinez, Samantha Muñiz, Ricardo Balcazar, Enrique Garcia, and Cesar Felipe Juarez. Hessian with mini-batches for electrical demand prediction. *Applied Sciences*, 10(6):2036, 2020.

[21] Giovanni Galdi. *An introduction to the mathematical theory of the Navier-Stokes equations: Steady-state problems*. Springer Science & Business Media, 2011.

[22] Daniel George and EA Huerta. Deep learning for real-time gravitational wave detection and parameter estimation: Results with advanced ligo data. *Physics Letters B*, 778:64–70, 2018.

[23] Philip E Gill and Walter Murray. Algorithms for the solution of the nonlinear least-squares problem. *SIAM Journal on Numerical Analysis*, 15(5):977–992, 1978.

[24] Alexander Gluhovsky and Christopher Tong. The structure of energy conserving low-order models. *Physics of Fluids*, 11(2):334–343, 1999.

[25] Ian Goodfellow, Yoshua Bengio, Aaron Courville, and Yoshua Bengio. *Deep learning*. MIT press Cambridge, 2016.

[26] Matthew A Grayson. The heat equation shrinks embedded plane curves to round points. *Journal of Differential geometry*, 26(2):285–314, 1987.

[27] FH Harlow. The marker-and-cell method. *Fluid Dyn. Numerical Methods*, 38, 1972.

[28] Francis H Harlow and J Eddie Welch. Numerical calculation of time-dependent viscous incompressible flow of fluid with free surface. *The physics of fluids*, 8(12):2182–2189, 1965.

[29] Magnus R Hestenes, Eduard Stiefel, et al. Methods of conjugate gradients for solving linear systems. *Journal of research of the National Bureau of Standards*, 49(6):409–436, 1952.

[30] Philipp Holl, Vladlen Koltun, and Nils Thuerey. Learning to control pdes with differentiable physics. In *International Conference on Learning Representations (ICLR)*, 2020.

[31] Gao Huang, Yu Sun, Zhuang Liu, Daniel Sedra, and Kilian Q Weinberger. Deep networks with stochastic depth. In *European conference on computer vision*, pages 646–661. Springer, 2016.

[32] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *arXiv:1502.03167*, February 2015.

[33] Diederik Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In *International Conference on Learning Representations (ICLR)*, 2015.

[34] Nikola Kovachki, Zongyi Li, Burigede Liu, Kamyar Azizzadenesheli, Kaushik Bhattacharya, Andrew Stuart, and Anima Anandkumar. Neural operator: Learning maps between function spaces. *arXiv preprint arXiv:2108.08481*, 2021.

[35] GV Kraniotis and SB Whitehouse. Compact calculation of the perihelion precession of mercury in general relativity, the cosmological constant and jacobi's inversion problem. *Classical and Quantum Gravity*, 20(22):4817, 2003.

[36] Dong C Liu and Jorge Nocedal. On the limited memory bfgs method for large scale optimization. *Mathematical programming*, 45(1-3):503–528, 1989.

[37] Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization. In *International Conference on Learning Representations (ICLR)*, 2019.

[38] Rajesh Maingi, Arnold Lumsdaine, Jean Paul Allain, Luis Chacon, SA Gourlay, CM Greenfield, JW Hughes, D Humphreys, V Izzo, H McLean, et al. Summary of the fesac transformative enabling capabilities panel report. *Fusion Science and Technology*, 75(3):167–177, 2019.

[39] James Martens et al. Deep learning via hessian-free optimization. In *ICML*, volume 27, pages 735–742, 2010.

[40] James Martens and Roger Grosse. Optimizing neural networks with kronecker-factored approximate curvature. In *International conference on machine learning*, pages 2408–2417. PMLR, 2015.

[41] Jorge J Moré. The levenberg-marquardt algorithm: implementation and theory. In *Numerical analysis*, pages 105–116. Springer, 1978.

[42] Patrick Mullen, Keenan Crane, Dmitry Pavlov, Yiying Tong, and Mathieu Desbrun. Energy-preserving integrators for fluid animation. *ACM Transactions on Graphics (TOG)*, 28(3):1–8, 2009.

[43] Mykola Novik. torch-optimizer – collection of optimization algorithms for PyTorch., 1 2020.

[44] 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. In *Advances in Neural Information Processing Systems*, pages 1–8, 2017.

[45] J Gregory Pauloski, Qi Huang, Lei Huang, Shivaram Venkataraman, Kyle Chard, Ian Foster, and Zhao Zhang. Kaisa: an adaptive second-order optimizer framework for deep neural networks. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–14, 2021.

[46] Michael Powell. A hybrid method for nonlinear equations. In *Numerical methods for nonlinear algebraic equations*. Gordon and Breach, 1970.

[47] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes*. Cambridge University Press, 3 edition, 2007.

[48] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation. In *International Conference on Medical image computing and computer-assisted intervention*, pages 234–241. Springer, 2015.

[49] Sirpa Saarinen, Randall Bramley, and George Cybenko. Ill-conditioning in neural network training problems. *SIAM Journal on Scientific Computing*, 14(3):693–714, 1993.

[50] Patrick Schnell, Philipp Holl, and Nils Thuerey. Half-inverse gradients for physical deep learning. *arXiv preprint arXiv:2203.10131*, 2022.

[51] Nicol N Schraudolph, Jin Yu, and Simon Günter. A stochastic quasi-newton method for online convex optimization. In *Artificial intelligence and statistics*, pages 436–443, 2007.

[52] Andrew Selle, Ronald Fedkiw, ByungMoon Kim, Yingjie Liu, and Jarek Rossignac. An unconditionally stable maccormack method. *Journal of Scientific Computing*, 35(2-3):350–371, June 2008.

[53] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research*, 15(1):1929–1958, 2014.

[54] Albert Tarantola. *Inverse problem theory and methods for model parameter estimation*. SIAM, 2005.

[55] Zhewei Yao, Amir Gholami, Sheng Shen, Mustafa Mustafa, Kurt Keutzer, and Michael W Mahoney. Adahessian: An adaptive second order optimizer for machine learning. *arXiv preprint arXiv:2006.00719*, 2020.

[56] Nan Ye, Farbod Roosta-Khorasani, and Tiangang Cui. Optimization methods for inverse problems. In *2017 MATRIX Annals*, pages 121–140. Springer, 2019.

[57] Kemin Zhou, John Comstock Doyle, Keith Glover, et al. *Robust and optimal control*, volume 40. Prentice hall New Jersey, 1996.

Checklist

1. For all authors...
   (a) Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope? [Yes]
   (b) Did you describe the limitations of your work? [Yes]
   (c) Did you discuss any potential negative societal impacts of your work? [N/A] We do not expect any negative impacts.
   (d) Have you read the ethics review guidelines and ensured that your paper conforms to them? [Yes]
2. If you are including theoretical results...
   (a) Did you state the full set of assumptions of all theoretical results? [Yes] We declare all assumptions the main text and provide an additional compact list in the appendix.
   (b) Did you include complete proofs of all theoretical results? [Yes]
3. If you ran experiments...
   (a) Did you include the code, data, and instructions needed to reproduce the main experimental results (either in the supplemental material or as a URL)? [Yes] We provide the full source code in the supplemental material.
   (b) Did you specify all the training details (e.g., data splits, hyperparameters, how they were chosen)? [Yes] We list learning rates and our training procedure in the appendix.
   (c) Did you report error bars (e.g., with respect to the random seed after running experiments multiple times)? [Yes] We show learning curves with different seeds in the appendix.

   (d) Did you include the total amount of compute and the type of resources used (e.g., type of GPUs, internal cluster, or cloud provider)? [Yes] Our hardware is listed in the appendix.

4. If you are using existing assets (e.g., code, data, models) or curating/releasing new assets...
   (a) If your work uses existing assets, did you cite the creators? [Yes] All used software libraries are given in the appendix.
   (b) Did you mention the license of the assets? [N/A] No proprietary licenses are involved.
   (c) Did you include any new assets either in the supplemental material or as a URL? [Yes] We include our source code.
   (d) Did you discuss whether and how consent was obtained from people whose data you're using/curating? [N/A]
   (e) Did you discuss whether the data you are using/curating contains personally identifiable information or offensive content? [N/A]

5. If you used crowdsourcing or conducted research with human subjects...
   (a) Did you include the full text of instructions given to participants and screenshots, if applicable? [N/A]
   (b) Did you describe any potential participant risks, with links to Institutional Review Board (IRB) approvals, if applicable? [N/A]
   (c) Did you include the estimated hourly wage paid to participants and the total amount spent on participant compensation? [N/A]

Who holds the Copyright on a NeurIPS paper

According to U.S. Copyright Office's page, **What is a Copyright**, when you create an original work you are the author and the owner and hold the copyright, unless you have an agreement to transfer the copyright to a third party such as the company or school you work for.

Authors do not transfer the copyright of their papers to NeurIPS. Instead, they grant NeurIPS a non-exclusive, perpetual, royalty-free, fully-paid, fully-assignable license to copy, distribute and publicly display all or part of the paper.

LEARNING TO CONTROL PDES WITH DIFFERENTIABLE PHYSICS

Philipp Holl
Technical University of Munich

Vladlen Koltun
Intel Labs

Nils Thuerey
Technical University of Munich

ABSTRACT

Predicting outcomes and planning interactions with the physical world are long-standing goals for machine learning. A variety of such tasks involves continuous physical systems, which can be described by partial differential equations (PDEs) with many degrees of freedom. Existing methods that aim to control the dynamics of such systems are typically limited to relatively short time frames or a small number of interaction parameters. We present a novel hierarchical predictor-corrector scheme which enables neural networks to learn to understand and control complex nonlinear physical systems over long time frames. We propose to split the problem into two distinct tasks: planning and control. To this end, we introduce a predictor network that plans optimal trajectories and a control network that infers the corresponding control parameters. Both stages are trained end-to-end using a differentiable PDE solver. We demonstrate that our method successfully develops an understanding of complex physical systems and learns to control them for tasks involving PDEs such as the incompressible Navier-Stokes equations.

1 INTRODUCTION

Intelligent systems that operate in the physical world must be able to perceive, predict, and interact with physical phenomena (Battaglia et al., 2013). In this work, we consider physical systems that can be characterized by partial differential equations (PDEs). PDEs constitute the most fundamental description of evolving systems and are used to describe every physical theory, from quantum mechanics and general relativity to turbulent flows (Courant & Hilbert, 1962; Smith, 1985). We aim to endow artificial intelligent agents with the ability to direct the evolution of such systems via continuous controls.

Such optimal control problems have typically been addressed via iterative optimization. Differentiable solvers and the adjoint method enable efficient optimization of high-dimensional systems (Toussaint et al., 2018; de Avila Belbute-Peres et al., 2018; Schenck & Fox, 2018). However, direct optimization through gradient descent (single shooting) at test time is resource-intensive and may be difficult to deploy in interactive settings. More advanced methods exist, such as multiple shooting and collocation, but they commonly rely on modeling assumptions that limit their applicability, and still require computationally intensive iterative optimization at test time.

Iterative optimization methods are expensive because they have to start optimizing from scratch and typically require a large number of iterations to reach an optimum. In many real-world control problems, however, agents have to repeatedly make decisions in specialized environments, and reaction times are limited to a fraction of a second. This motivates the use of data-driven models such as deep neural networks, which combine short inference times with the capacity to build an internal representation of the environment.

We present a novel deep learning approach that can learn to represent solution manifolds for a given physical environment, and is orders of magnitude faster than iterative optimization techniques. The core of our method is a hierarchical predictor-corrector scheme that temporally divides the problem into easier subproblems. This enables us to combine models specialized to different time scales in order to control long sequences of complex high-dimensional systems. We train our models using a differentiable PDE solver that can provide the agent with feedback of how interactions at any point in time affect the outcome. Our models learn to represent manifolds containing a large number of solutions, and can thereby avoid local minima that can trap classic optimization techniques.

We evaluate our method on a variety of control tasks in systems governed by advection-diffusion PDEs such as the Navier-Stokes equations. We quantitatively evaluate the resulting sequences on how well they approximate the target state and how much force was exerted on the physical system. Our method yields stable control for significantly longer time spans than alternative approaches.

2 Background

Physical problems commonly involve nonlinear PDEs, often with many degrees of freedom. In this context, several works have proposed methods for improving the solution of PDE problems (Long et al., 2018; Bar-Sinai et al., 2019; Hsieh et al., 2019) or used PDE formulations for unsupervised optimization (Raissi et al., 2018). Lagrangian fluid simulation has been tackled with regression forests (Ladicky et al., 2015), graph neural networks (Mrowca et al., 2018; Li et al., 2019), and continuous convolutions (Ummenhofer et al., 2020). Data-driven turbulence models were trained with MLPs (Ling et al., 2016). Fully-convolutional networks were trained for pressure inference (Tompson et al., 2017) and advection components were used in adversarial settings (Xie et al., 2018). Temporal updates in reduced spaces were learned via the Koopman operator (Morton et al., 2018). In a related area, deep networks have been used to predict chemical properties and the outcome of chemical reactions (Gilmer et al., 2017; Bradshaw et al., 2019).

Differentiable solvers have been shown to be useful in a variety of settings. Degrave et al. (2019) and de Avila Belbute-Peres et al. (2018) developed differentiable simulators for rigid body mechanics. (See Popovic et al. (2000) for earlier work in computer graphics.) Toussaint et al. (2018) applied related techniques to manipulation planning. Specialized solvers were developed to infer protein structures (Ingraham et al., 2019), interact with liquids (Schenck & Fox, 2018), control soft robots (Hu et al., 2019), and solve inverse problems that involve cloth (Liang et al., 2019). Like ours, these works typically leverage the automatic differentiation of deep learning pipelines (Griewank & Walther, 2008; Maclaurin et al., 2015; Amos & Kolter, 2017; Mensch & Blondel, 2018; van Merriënboer et al., 2018; Chen et al., 2018; Bradbury et al., 2018; Paszke et al., 2019; Tokui et al., 2019). However, while the works above target Lagrangian solvers, i.e. reference frames moving with the simulated material, we address grid-based solvers, which are particularly appropriate for dense, volumetric phenomena.

The adjoint method (Lions, 1971; Pironneau, 1974; Jameson, 1988; Giles & Pierce, 2000; Bewley, 2001; McNamara et al., 2004) is used by most machine learning frameworks, where it is commonly known as reverse mode differentiation (Werbos, 2006; Chen et al., 2018). While a variety of specialized adjoint solvers exist (Griewank et al., 1996; Fournier et al., 2012; Farrell et al., 2013), these packages do not interface with production machine learning frameworks. A supporting contribution of our work is a differentiable PDE solver called Φ_{Flow} that integrates with TensorFlow (Abadi et al., 2016) and PyTorch (Paszke et al., 2019). It is publicly available at https://github.com/tum-pbs/PhiFlow.

3 Problem

Consider a physical system $\boldsymbol{u}(\boldsymbol{x}, t)$ whose natural evolution is described by the PDE

$$\frac{\partial \boldsymbol{u}}{\partial t} = \mathcal{P}\left(\boldsymbol{u}, \frac{\partial \boldsymbol{u}}{\partial \boldsymbol{x}}, \frac{\partial^2 \boldsymbol{u}}{\partial \boldsymbol{x}^2}, ..., \boldsymbol{y}(t)\right), \tag{1}$$

where $\mathcal{P}$ models the physical behavior of the system and $\boldsymbol{y}(t)$ denotes external factors that can influence the system. We now introduce an agent that can interact with the system by controlling certain parameters of the dynamics. This could be the rotation of a motor or fine-grained control over a field. We factor out this influence into a force term $\boldsymbol{F}$, yielding

$$\frac{\partial \boldsymbol{u}}{\partial t} = \mathcal{P}\left(\boldsymbol{u}, \frac{\partial \boldsymbol{u}}{\partial \boldsymbol{x}}, \frac{\partial^2 \boldsymbol{u}}{\partial \boldsymbol{x}^2}, ...\right) + \boldsymbol{F}(t). \tag{2}$$

The agent can now be modelled as a function that computes $\boldsymbol{F}(t)$. As solutions of nonlinear PDEs were shown to yield low-dimensional manifolds (Foias et al., 1988; Titi, 1990), we target solution manifolds of $\boldsymbol{F}(t)$ for a given choice of $\mathcal{P}$ with suitable boundary conditions. This motivates our choice to employ deep networks for our agents.

In most real-world scenarios, it is not possible to observe the full state of a physical system. When considering a cloud of smoke, for example, the smoke density may be observable while the velocity field may not be seen directly. We model the imperfect information by defining the observable state of $\boldsymbol{u}$ as $\boldsymbol{o}(\boldsymbol{u})$. The observable state is problem dependent, and our agent is conditioned only on these observations, i.e. it does not have access to the full state $\boldsymbol{u}$.

Using the above notation, we define the control task as follows. An initial observable state $\boldsymbol{o}_0$ of the PDE as well as a target state $\boldsymbol{o}^*$ are given (Figure 1a). We are interested in a reconstructed trajectory $\boldsymbol{u}(t)$ that matches these states at t_0 and t_*, i.e. $\boldsymbol{o}_0 = \boldsymbol{o}(\boldsymbol{u}(t_0)), \boldsymbol{o}^* = \boldsymbol{o}(\boldsymbol{u}(t_*))$, and minimizes the amount of force applied within the simulation domain $\mathcal{D}$ (Figure 1b):

$$L_{\boldsymbol{F}}[\boldsymbol{u}(t)] = \int_{t_0}^{t_*} \int_{\mathcal{D}} |\boldsymbol{F}_{\boldsymbol{u}}(t)|^2 \, dx \, dt. \tag{3}$$

Taking discrete time steps Δt, the reconstructed trajectory $\boldsymbol{u}$ is a sequence of $n = (t_* - t_0)/\Delta t$ states.

When an observable dimension cannot be controlled directly, there may not exist any trajectory $\boldsymbol{u}(t)$ that matches both $\boldsymbol{o}_0$ and $\boldsymbol{o}^*$. This can stem from either physical constraints or numerical limitations. In these cases, we settle for an approximation of $\boldsymbol{o}^*$. To measure the quality of the approximation of the target, we define an observation loss $L_{\boldsymbol{o}}^*$. The form of this loss can be chosen to fit the problem. We combine Eq. 3 and the observation loss into the objective function

$$L[\boldsymbol{u}(t)] = \alpha \cdot L_{\boldsymbol{F}}[\boldsymbol{u}(t)] + L_{\boldsymbol{o}}^*(\boldsymbol{u}(t_*)), \tag{4}$$

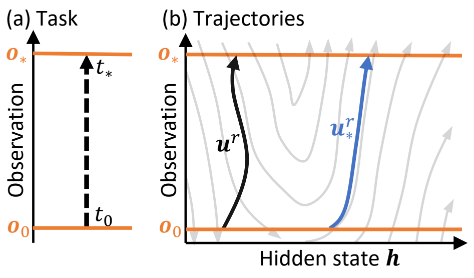


Figure 1: Illustration of possible trajectories. The grey lines represent the unperturbed evolution of the physical system. The amount of applied force corresponds to how far the trajectory deviates from the natural evolution.

with $\alpha > 0$. We use square brackets to denote functionals, i.e. functions depending on fields or series rather than single values.

4 Preliminaries

Differentiable solvers. Let $\boldsymbol{u}(\boldsymbol{x}, t)$ be described by a PDE as in Eq. 1. A regular solver can move the system forward in time via Euler steps:

$$\boldsymbol{u}(t_{i+1}) = \text{Solver}[\boldsymbol{u}(t_i), \boldsymbol{y}(t_i)] = \boldsymbol{u}(t_i) + \Delta t \cdot \mathcal{P}\left(\boldsymbol{u}(t_i), ..., \boldsymbol{y}(t_i)\right). \tag{5}$$

Each step moves the system forward by a time increment Δt. Repeated execution produces a trajectory $u(t)$ that approximates a solution to the PDE. This functionality for time advancement by itself is not well-suited to solve optimization problems, since gradients can only be approximated by finite differencing. For high-dimensional or continuous systems, this method becomes computationally expensive because a full trajectory needs to be computed for each optimizable parameter.

Differentiable solvers resolve this issue by solving the adjoint problem (Pontryagin, 1962) via analytic derivatives. The adjoint problem computes the same mathematical expressions while working with lower-dimensional vectors. A differentiable solver can efficiently compute the derivatives with respect to any of its inputs, i.e. $\partial \boldsymbol{u}(t_{i+1})/\partial \boldsymbol{u}(t_i)$ and $\partial \boldsymbol{u}(t_{i+1})/\partial \boldsymbol{y}(t_i)$. This allows for gradient-based optimization of inputs or control parameters over an arbitrary number of time steps.

Iterative trajectory optimization. Many techniques exist that try to find optimal trajectories by starting with an initial guess for $\boldsymbol{F}(t)$ and slightly changing it until reaching an optimum. The simplest of these is known as single shooting. In one optimization step, it simulates the full dynamics, then backpropagates the loss through the whole sequence to optimize the controls (Kraft, 1985; Leineweber et al., 2003). Replacing $\boldsymbol{F}(t)$ with an agent $\boldsymbol{F}(t|\boldsymbol{o}_t, o^*)$, which can be parameterized by a deep network, yields a simple training method. For a sequence of n frames, this setup contains n linked copies of the agent and is depicted in Figure 2. We refer to such an agent as a control force estimator (CFE).

Optimizing such a chain of CFEs is both computationally expensive and causes gradients to pass through a potentially long sequence of highly nonlinear simulation steps. When the reconstruction u is close to an optimal trajectory, this is not a problem because the gradients $\Delta \boldsymbol{u}$ are small and the operations executed by the solver are differentiable by construction. The solver can therefore be locally approximated by a first-order polynomial and the gradients can be safely backpropagated. For large $\Delta \boldsymbol{u}$, e.g. at the beginning of an optimization, this approximation breaks down, causing the gradients to become unstable while passing through the chain. This instability in the training process can prevent single-shooting approaches from converging and deep networks from learning unless they are initialized near an optimum.

Alternatives to single shooting exist, promising better and more efficient convergence. Multiple shooting (Bock & Plitt, 1984) splits the trajectory into segments with additional defect constraints. Depending on the physical system, this method may have to be adjusted for specific problems (Treuille et al., 2003). Collocation schemes (Hargraves & Paris, 1987) model trajectories with splines. While this works well for particle trajectories, it is poorly suited for Eulerian solvers where the evolution of individual points does not reflect the overall motion. Model reduction can be used to reduce the dimensionality or nonlinearity of the problem, but generally requires domain-specific knowledge. When applicable, these methods can converge faster or in a more stable manner than single shooting. However, as we are focusing on a general optimization scheme in this work, we will use single shooting and its variants as baseline comparisons.

Supervised and differentiable physics losses. One of the key ingredients in training a machine learning model is the choice of loss function. For many tasks, supervised losses are used, i.e. losses that directly compare the output of the model for a specific input with the desired ground truth. While supervised losses can be employed for trajectory optimization, far better loss functions are possible when a differentiable solver is available. We will refer to these as *differentiable physics* loss functions. In this work, we employ a combination of supervised and differentiable physics losses, as both come with advantages and disadvantages.

One key limitation of supervised losses is that they can only measure the error of a single time step. Therefore, an agent cannot get any measure of how its output would influence future time steps. Another problem arises from the form of supervised training data which comprises input-output pairs, which may be obtained directly from data generation or through iterative optimization. Since optimal control problems are generally not unimodal, there can exist multiple possible outputs for one input. This ambiguity in the supervised training process will lead to suboptimal predictions as the network will try to find a compromise between all possible outputs instead of picking one of them.

Differentiable physics losses solve these problems by allowing the agent to be directly optimized for the desired objective (Eq. 4). Unlike supervised losses, differentiable physics losses require a differentiable solver to backpropagate the gradients through the simulation. Multiple time steps can be chained together, which is a key requirement since the objective (Eq. 4) explicitly depends on all

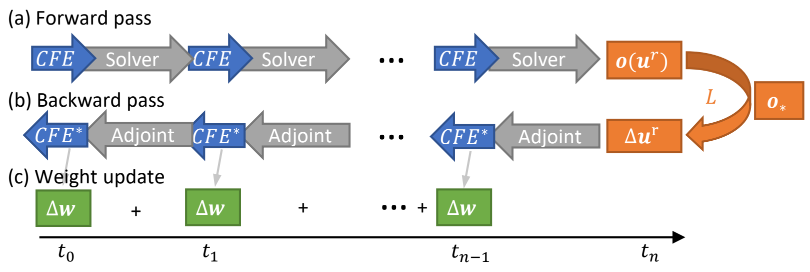


Figure 2: Single-shooting optimization with a control force estimator (CFE). (a) The forward pass simulates the full sequence. (b) Backpropagation computes the adjoint problem. (c) Weight updates are accumulated and applied to the CFE.

time steps through $L_{\boldsymbol{F}}[\boldsymbol{u}(t)]$ (Eq. 3). As with iterative solvers, one optimization step for a sequence of n frames then invokes the agent n times before computing the loss, each invocation followed by a solver step. The employed differentiable solver backpropagates the gradients through the whole sequence, which gives the model feedback on (i) how its decisions change the future trajectory and (ii) how to handle states as input that were reached because of its previous decisions. Since no ground truth needs to be provided, multi-modal problems naturally converge towards one solution.

5 Method

In order to optimally interact with a physical system, an agent has to (i) build an internal representation of an optimal observable trajectory $\boldsymbol{o}(\boldsymbol{u}(t))$ and (ii) learn what actions to take to move the system along the desired trajectory. These two steps strongly resemble the predictor-corrector method (Press et al., 2007). Given $\boldsymbol{o}(t)$, a predictor-corrector method computes $\boldsymbol{o}(t + \Delta t)$ in two steps. First, a prediction step approximates the next state, yielding $\boldsymbol{o}^p(t + \Delta t)$. Then, the correction uses $\boldsymbol{o}^p(t + \Delta t)$ to refine the initial approximation and obtain $\boldsymbol{o}(t + \Delta t)$. Each step can, to some degree, be learned independently.

This motivates splitting the agent into two neural networks: an observation predictor (OP) network that infers intermediate states $\boldsymbol{o}^p(t_i)$, $i \in \{1, 2, ...n-1\}$, planning out a trajectory, and a corrector network (CFE) that estimates the control force $\boldsymbol{F}(t_i|\boldsymbol{o}(\boldsymbol{u}_i), \boldsymbol{o}^p_{i+1})$ to follow that trajectory as closely as possible. This splitting has the added benefit of exposing the planned trajectory, which would otherwise be inaccessible. As we will demonstrate, it is crucial for the prediction stage to incorporate knowledge about longer time spans. We address this by modelling the prediction as a temporally hierarchical process, recursively dividing the problem into smaller subproblems.

To achieve this, we let the OP not directly infer $\boldsymbol{o}^p(t_{i+1} \,|\, \boldsymbol{o}(\boldsymbol{u}_i), \boldsymbol{o}^*)$ but instead model it to predict the optimal center point between two states at times t_i, t_j, with $i, j \in \{1, 2, ...n-1\}, j > i$, i.e. $\boldsymbol{o}^p((t_i + t_j)/2 \,|\, \boldsymbol{o}_i, \boldsymbol{o}_j)$. This function is much more general than predicting the state of the next time step since two arbitrary states can be passed as arguments. Recursive OP evaluations can then partition the sequence until a prediction $\boldsymbol{o}^p(t_i)$ for every time step t_i has been made.

This scheme naturally enables scaling to arbitrary time frames or arbitrary temporal resolutions, assuming that the OP can correctly anticipate the physical behavior. Since physical systems often exhibit different behaviors on different time scales and the OP can be called with states separated by arbitrary time spans, we condition the OP on the time scale it is evaluated on by instantiating and training a unique version of the model for every time scale. This simplifies training and does not significantly increase the model complexity as we use factors of two for the time scales, and hence the number of required models scales with $\mathcal{O}(\log_2 n)$. We will refer to one instance of an OP$_n$ by the time span between its input states, measured in the number of frames $n = (t_j - t_i)/\Delta t$.

Execution order. With the CFE and OP$_n$ as building blocks, many algorithms for solving the control problem, i.e. for computing $\boldsymbol{F}(t)$, can be assembled and trained. We compared a variety of algorithms and found that a scheme we will refer to as *prediction refinement* produces the best results. It is based on the following principles: (i) always use the finest scale OP possible to make a prediction, (ii) execute the CFE followed by a solver step as soon as possible, (iii) refine predictions after the solver has computed the next state. The algorithm that realizes these goals is shown in Appendix B with an example for $n = 8$. To understand the algorithm and resulting execution orders, it is helpful to consider simpler algorithms first.

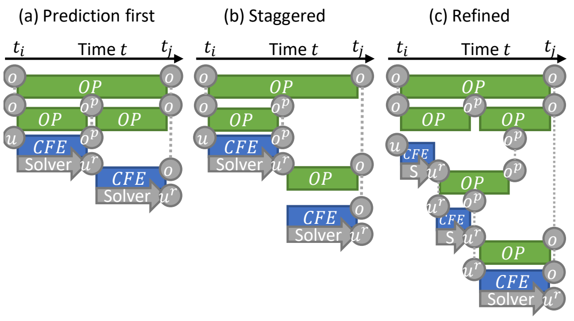


Figure 3: Overview of the different execution schemes.

The simplest combination of CFE and OP$_n$ invocations that solves the full trajectory, shown in Figure 3a, can be described as follows. Initially, all intermediate states are predicted hierarchically. The first prediction is the half-way point $\boldsymbol{o}^p(t_{n/2} \,|\, \boldsymbol{o}_0, \boldsymbol{o}^*)$, generated by the OP$_n$. Using that as input to an OP$_{n/2}$ results in new predictions at $t_{n/4}, t_{3n/4}$. Continuing with this scheme, a prediction can be made for each $t_i, i \in 1, ..., n-1$. Next, the actual trajectory is evaluated step by step. For each step t_i, the CFE computes the control force $\boldsymbol{F}(t_i)$ conditioned on the state at t_i and the prediction $\boldsymbol{o}^p(t_{i+1})$. Once $\boldsymbol{F}(t_i)$ is known, the solver can step the simulation to the next state at t_{i+1}. This algorithm finds a trajectory in time $\mathcal{O}(n)$ since n CFE calls and $n-1$ OP calls are required in total (see Appendix B). However, there are inherent problems with this algorithm. The physical constraints of the PDE and potential approximation errors of the CFE can result in observations that are only matched partially. This can result in the reconstructed trajectory exhibiting undesirable oscillations, often visible as jittering. When subsequent predictions do not line up perfectly, large forces may be applied by the CFE or the reconstructed trajectory might stop following the predictions altogether.

This problem can be alleviated by changing the execution order of the two-stage algorithm described above. The resulting algorithm is shown in Figure 3b and will be referred to as *staggered execution*. In this setup, the simulation is advanced as soon as a prediction for the next observable state exists and OPs are only executed when their state at time t_i is available. This staggered execution scheme allows future predictions to take deviations from the predicted trajectory into account, preventing a divergence of the actual evolution $\boldsymbol{o}(\boldsymbol{u}(t))$ from the prediction $\boldsymbol{o}^p(t)$.

While the staggered execution allows most predictions to correct for deviations from the predicted trajectory $\boldsymbol{o}^p$, this scheme leaves several predictions unmodified. Most notably, the prediction $\boldsymbol{o}^p(t_{n/2})$, which is inferred from just the initial state and the desired target, remains unchanged. This prediction must therefore be able to guide the reconstruction in the right direction without knowing about deviations in the system that occurred up to $t_{n/2-1}$. As a practical consequence, a network trained with this scheme typically learns to average over the deviations, resulting in blurred predictions (see Appendix D.2).

Algorithm 1: Recursive algorithm computing the prediction refinement. The algorithm is called via $\mathrm{Reconstruct}[\boldsymbol{o}_0, \boldsymbol{o}_*, absent]$ to reconstruct a full trajectory from $\boldsymbol{o}_0$ to $\boldsymbol{o}_*$.

```
function $\mathrm{Reconstruct}[\boldsymbol{o}(\boldsymbol{u}_0), \boldsymbol{o}_n, \boldsymbol{o}_{2n}]$;
Input  : Initial observation $\boldsymbol{o}(\boldsymbol{u}_0)$, observation $\boldsymbol{o}_n$, optional observation $\boldsymbol{o}_{2n}$
Output: Observation of the reconstructed state $\boldsymbol{o}(\boldsymbol{u}_n)$
if $n = 1$ then
    $\boldsymbol{F} \leftarrow \mathrm{CFE}[\boldsymbol{o}(\boldsymbol{u}_0), \boldsymbol{o}_1]$
    $\boldsymbol{u}_1 \leftarrow \mathrm{Solver}[\boldsymbol{u}_0, \boldsymbol{F}]$
    return $\boldsymbol{o}(\boldsymbol{u}_1)$
else
    $\boldsymbol{o}_{n/2} \leftarrow \mathrm{OP}[\boldsymbol{o}(\boldsymbol{u}_0), \boldsymbol{o}_n]$
    $\boldsymbol{o}(\boldsymbol{u}_{n/2}) \leftarrow \mathrm{Reconstruct}[\boldsymbol{o}(u_0), \boldsymbol{o}_{n/2}, \boldsymbol{o}_n]$
    if $\boldsymbol{o}_{2n}$ present then
        $\boldsymbol{o}_{3n/2} \leftarrow \mathrm{OP}[\boldsymbol{o}_n, \boldsymbol{o}_{2n}]$
        $\boldsymbol{o}_n \leftarrow \mathrm{OP}[\boldsymbol{o}(\boldsymbol{u}_{n/2}), \boldsymbol{o}_{3n/2}]$
    else
        $\boldsymbol{o}_{3n/2} \leftarrow absent$
    end
    $\boldsymbol{o}(\boldsymbol{u}_n) \leftarrow \mathrm{Reconstruct}[\boldsymbol{o}(\boldsymbol{u}_{n/2}), \boldsymbol{o}_n, \boldsymbol{o}_{3n/2}]$
    return $\boldsymbol{o}(\boldsymbol{u}_n)$
end
```

The prediction refinement scheme, listed in Algorithm 1 and illustrated in Figure 3c, solves this problem by re-evaluating existing predictions whenever the simulation progesses in time. Not all predictions need to be updated, though, and an update to a prediction at a finer time scale can depend on a sequence of other predictions. The *prediction refinement* algorithm that achieves this in an optimal form is listed in Appendix B. While the resulting execution order is difficult to follow for longer sequences with more than $n = 8$ frames, we give an overview of the algorithm by considering the prediction for time $t_{n/2}$. After the first center-frame prediction $\boldsymbol{o}^p(t_{n/2})$ of the n-frame sequence is made by OP$_n$, the prediction refinement algorithm calls itself recursively until all frames up to frame $n/4$ are reconstructed from the CFE and the solver. The center prediction is then updated using OP$_{n/2}$ for the next smaller time scale compared to the previous prediction. The call of OP$_{n/2}$

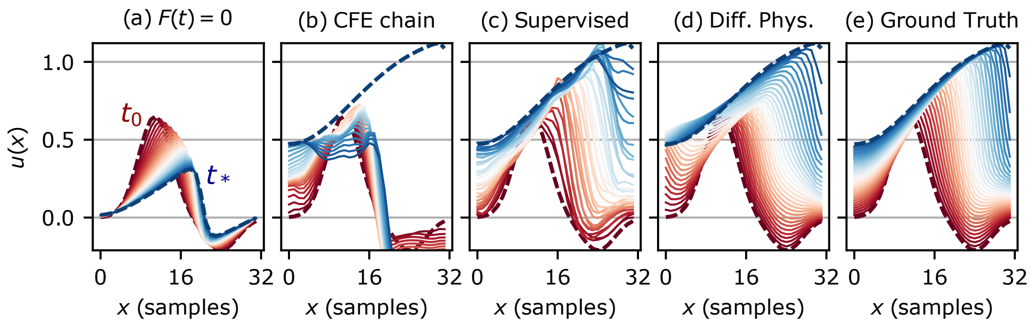


Figure 4: Trajectories for an example control task using Burger's equation. Initial and target states are plotted with thick dashed lines in red and blue, respectively. Inferred states are shown as solid lines. (a) Natural evolution. (b) Reconstruction using a CFE chain. (c,d) Reconstructions using our hierarchical predictor-corrector scheme. (e) Ground-truth trajectory generated with constant force.

also depends on $\boldsymbol{o}^p(t_{3n/4})$, which was predicted using $\mathrm{OP}_{n/2}$. After half of the remaining distance to the center is reconstructed by the solver, the center prediction at $t_{n/2}$ is updated again, this time by the $\mathrm{OP}_{n/4}$, including all prediction dependencies. Hence, the center prediction is continually refined every time the temporal distance between the latest reconstruction and the prediction halves, until the reconstruction reaches that frame. This way, all final predictions $\boldsymbol{o}^p(t_i)$ are conditioned on the reconstruction of the previous state $\boldsymbol{u}(t_{i-1})$ and can therefore account for all previous deviations.

The prediction refinement scheme requires the same number of force inferences but an increased number of OP evaluations compared to the simpler algorithms. With a total of $3n - 2\log_2(n) - 3$ OP evaluations (see Appendix B), it is of the same complexity, $\mathcal{O}(n)$. In practice, this refinement scheme incurs only a small overhead in terms of computation, which is outweighed by the significant gains in quality of the learned control function.

6 Results

We evaluate the capabilities of our method to learn to control physical PDEs in three different test environments of increasing complexity. We first target a simple but nonlinear 1D equation, for which we present an ablation study to quantify accuracy. We then study two-dimensional problems: an incompressible fluid and a fluid with complex boundaries and indirect control. Full details are given in Appendix D. Supplemental material containing additional sequences for all of the tests can be downloaded from https://ge.in.tum.de/publications/2020-iclr-holl.

Burger's equation. Burger's equation is a nonlinear PDE that describes the time evolution of a single field, u (LeVeque, 1992). Using Eq. 1, it can be written as

$$\mathcal{P}\left(u, \frac{\partial u}{\partial x}, \frac{\partial^2 u}{\partial x^2}\right) = -u \cdot \frac{\partial u}{\partial x} + \nu \frac{\partial^2 u}{\partial x^2}. \tag{6}$$

Examples of the unperturbed evolution are shown in Figure 4a. We let the whole state be observable and controllable, i.e. $o(t) = u(t)$, which implies that $\boldsymbol{o}^*$ can always be reached exactly.

The results of our ablation study with this equation are shown in Table 1. The table compares the resulting forces applied by differently trained models when reconstructing a ground-truth sequence (Figure 4e). The variant denoted by *CFE chain* uses a neural network to infer the force without any intermediate predictions. With a supervised loss, this method learns to approximate a single step well. However, for longer sequences, results quickly deviate from an ideal trajectory and diverge because the network never learned to account for errors made in previous steps (Figure 4b). Training the network with the objective loss (Eq. 4) using the differentiable solver greatly increases the quality of the reconstructions. On average, it applies only 34% of the force used by the supervised model as it learns to correct the temporal evolution of the PDE model.

Next, we evaluate variants of our predictor-corrector approach, which hierarchically predicts intermediate states. Here, the CFE is implemented as $F(t_i) = (o^p(t_{i+1}) - u(t_i))/\Delta t$. Unlike the

Table 1: Quantitative reconstruction evaluation using Burger's equation, avg. for 100 examples.

Execution scheme	Training loss	Force $\int \lvert \boldsymbol{F} \rvert \, dt$	Inference time (ms)
CFE chain	Supervised	83.4 ± 2.0	0.024 ± 0.013
CFE chain	Diff. Physics	28.8 ± 0.8	0.024 ± 0.013
Staggered	Supervised	34.3 ± 1.1	1.15 ± 0.19
Staggered	Diff. Physics	15.3 ± 0.7	1.15 ± 0.19
Refined	Diff. Physics	14.2 ± 0.7	3.05 ± 0.37
Iterative optim. (60 iter.)	Diff. Physics	15.3 ± 1.6	52.7 ± 2.1
Iterative optim. (300 iter.)	Diff. Physics	10.2 ± 1.9	264.0 ± 3.0

Table 2: A comparison of methods in terms of final cost for (a) the natural flow setup and (b) the shape transitions. The initial distribution is sampled randomly and evolved to the target state.

Execution	Loss	a) Force $L_{\boldsymbol{F}}$	a) Obs. $L_{\boldsymbol{o}}^*$	b) Force $L_{\boldsymbol{F}}$	b) Obs. $L_{\boldsymbol{o}}^*$
Staggered	Supervised	243 ± 11	1.53 ± 0.23	n/a	n/a
Staggered	Diff. Physics	22.6 ± 1.1	0.64 ± 0.08	89 ± 6	0.331 ± 0.134
Refined	Diff. Physics	11.7 ± 0.6	0.88 ± 0.11	75 ± 4	0.126 ± 0.010

simple CFE chain above, training with the supervised loss and staggered execution produces stable (albeit jittering) trajectories that successfully converge to the target state (Figure 4c). Surprisingly, this supervised method reaches almost the same accuracy as the differentiable CFE, despite not having access to physics-based gradients. However, employing the differentiable physics loss greatly improves the reconstruction quality, producing solutions that are hard to distinguish from ideal trajectories (Figure 4d). The prediction refinement scheme further improves the accuracy, but the differences to the staggered execution are relatively small as the predictions of the latter are already very accurate.

Table 1 also lists the results of classic shooting-based optimization applied to this problem. To match the quality of the staggered execution scheme, the shooting method requires around 60 optimization steps. These steps are significantly more expensive to compute, despite the fast convergence. After around 300 iterations, the classic optimization reaches an optimal value of 10.2 and the loss stops decreasing. Starting the iterative optimization with our method as an initial guess pushes the optimum slightly lower to 10.1. Thus, even this relatively simple problem shows the advantages of our learned approach.

Incompressible fluid flow. Next, we apply our algorithm to two-dimensional fluid dynamics problems, which are challenging due to the complexities of the governing Navier-Stokes equations (Batchelor, 1967). For a velocity field $\boldsymbol{v}$, these can be written as

$$\mathcal{P}(\boldsymbol{v}, \nabla \boldsymbol{v}) = -\boldsymbol{v} \cdot \nabla \boldsymbol{v} + \nu \nabla^2 \boldsymbol{v} - \nabla p, \tag{7}$$

subject to the hard constraints $\nabla \cdot \boldsymbol{v} = 0$ and $\nabla \times p = 0$, where p denotes pressure and ν the viscosity. In addition, we consider a passive density ρ that moves with the fluid via $\partial \rho / \partial t = -\boldsymbol{v} \cdot \nabla \rho$. We set $\boldsymbol{v}$ to be hidden and ρ to be observable, and allow forces to be applied to all of $\boldsymbol{v}$.

We run our tests on a 128^2 grid, resulting in more than 16,000 effective continuous control parameters. We train the OP and CFE networks for two different tasks: reconstruction of natural fluid flows and controlled shape transitions. Example sequences are shown in Figure 5 and a quantitative evaluation, averaged over 100 examples, is given in Table 2. While all methods manage to approximate the target state well, there are considerable differences in the amount of force applied. The supervised technique exerts significantly more force than the methods based on the differentiable solver, resulting in jittering reconstructions. The prediction refinement scheme produces the smoothest transitions, converging to about half the loss of the staggered, non-refined variant.

We compare our method to classic shooting algorithms for this incompressible flow problem. While a direct shooting method fails to converge, a more advanced multi-scale shooting approach still re-

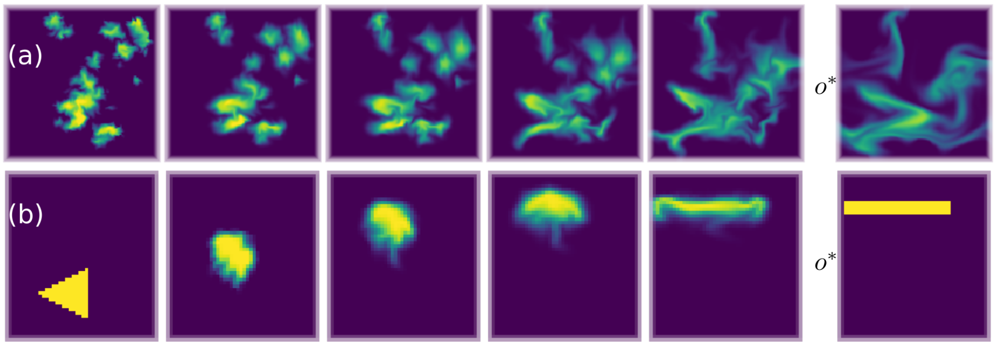


Figure 5: Example reconstructed trajectory from (a) the natural flow test set and (b) the shape test set. The target state $\boldsymbol{o}^*$ is shown on the right.

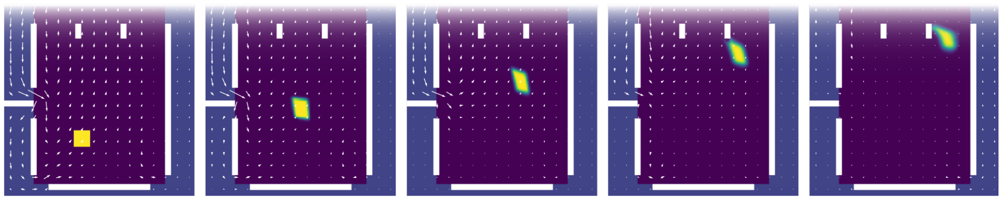


Figure 6: Indirect control sequence. Obstacles are marked white, control regions light blue. The white arrows indicate the velocity field. The domain is enclosed in a solid box with an open top.

quires 1500 iterations to obtain a level of accuracy that our model achieves almost instantly. In addition, our model successfully learns a solution manifold, while iterative optimization techniques essentially *start from scratch* every time. This global view leads our model to more intuitive solutions and decreases the likelihood of convergence to undesirable local minima. The solutions of our method can also be used as initial guesses for iterative solvers, as illustrated in Appendix D.4. We find that the iterative optimizer with an initial guess converges to solutions that require only 57.4% of the force achieved by the iterative optimizer with default initialization. This illustrates how the more global view of the learned solution manifold can improve the solutions of regular optimization runs.

Splitting the task into prediction and correction ensures that intermediate predicted states are physically plausible and allows us to generalize to new tasks. For example, we can infer transitions involving multiple shapes, despite training only on individual shapes. This is demonstrated in Appendix D.2.

Incompressible fluid with indirect control. The next experiment increases the complexity of the fluid control problem by adding obstacles to the simulated domain and limiting the area that can be controlled by the network. An example sequence in this setting is shown in Figure 6. As before, only the density ρ is observable. Here, the goal is to move the smoke from its initial position near the center into one of the three "buckets" at the top. Control forces can only be applied in the peripheral regions, which are outside the visible smoke distribution. Only by synchronizing the 5000 continuous control parameters can a directed velocity field be constructed in the central region.

We first infer trajectories using a trained CFE network and predictions that move the smoke into the desired bucket in a straight line. This baseline manages to transfer $89\% \pm 2.6\%$ of the smoke into the target bucket. Next we enable the hierarchical predictions and train the OPs. This version manages to maneuver $99.22\% \pm 0.15\%$ of the smoke into the desired buckets while requiring $19.1\% \pm 1.0\%$ less force.

For comparison, Table 3 also lists success rate and execution time for a direct optimization. Despite only obtaining a low success rate of 82%, the shooting method requires several orders of magnitude longer than evaluating our trained model. Since all optimizations are independent of each other, some find better solutions than others, reflected in the higher standard deviation. The increased number of free parameters and complexity of the fluid dynamics to be controlled make this problem

Table 3: Comparison of different methods on the task of moving a distribution of smoke into the target region by applying forces outside the region.

Method	Optimized quantity	Inside target (%)	Inference time (ms)
Straight trajectory	CFE	89.5 ± 2.6	31.46 ± 0.20
Staggered predictions	CFE, OP$_n$	99.22 ± 0.15	67.40 ± 0.20
Iterative optim.	Control velocity	82.1 ± 7.3	$266.5 \cdot 10^3$

intractable for the shooting method, while our model can leverage the learned representation to infer a solution very quickly. Further details are given in Appendix D.3.

7 Conclusions

We have demonstrated that deep learning models in conjunction with a differentiable physics solver can successfully predict the behavior of complex physical systems and learn to control them. The introduction of a hierarchical predictor-corrector architecture allowed the model to learn to reconstruct long sequences by treating the physical behavior on different time scales separately.

We have shown that using a differentiable solver greatly benefits the quality of solutions since the networks can learn how their decisions will affect the future. In our experiments, hierarchical inference schemes outperform traditional sequential agents because they can easily learn to plan ahead.

To model realistic environments, we have introduced observations to our pipeline which restrict the information available to the learning agent. While the PDE solver still requires full state information to run the simulation, this restriction does not apply when the agent is deployed.

While we do not believe that learning approaches will replace iterative optimization, our method shows that it is possible to learn representations of solution manifolds for optimal control trajectories using data-driven approaches. Fast inference is vital in time-critical applications and can also be used in conjunction with classical solvers to speed up convergence and ultimately produce better solutions.

8 Acknowledgements

This work was supported in part by the ERC Starting Grant *realFlow* (ERC-2015-StG-637014).

References

Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. TensorFlow: A system for large-scale machine learning. In *Symposium on Operating Systems Design and Implementation*, 2016.

Brandon Amos and J Zico Kolter. OptNet: Differentiable optimization as a layer in neural networks. In *ICML*, 2017.

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.

George K. Batchelor. *An Introduction to Fluid Dynamics*. Cambridge University Press, 1967.

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.

Thomas R. Bewley. Flow control: new challenges for a new renaissance. *Progress in Aerospace Sciences*, 37, 2001.

Hans Georg Bock and Karl-Josef Plitt. A multiple shooting algorithm for direct solution of optimal control problems. *IFAC Proceedings Volumes*, 17(2), 1984.

James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, and Skye Wanderman-Milne. JAX: Composable transformations of Python+NumPy programs. *GitHub*, 2018.

John Bradshaw, Matt J Kusner, Brooks Paige, Marwin HS Segler, and José Miguel Hernández-Lobato. A generative model for electron paths. In *ICLR*, 2019.

Tian Qi Chen, Yulia Rubanova, Jesse Bettencourt, and David K Duvenaud. Neural ordinary differential equations. In *Advances in Neural Information Processing Systems*, 2018.

Alexandre Joel Chorin. The numerical solution of the Navier-Stokes equations for an incompressible fluid. *Bulletin of the American Mathematical Society*, 73(6), 1967.

Richard Courant and David Hilbert. *Methods of Mathematical Physics*, volume II: Partial Differential Equations. Interscience Publishers, 1962.

Filipe de Avila Belbute-Peres, Kevin Smith, Kelsey Allen, Josh Tenenbaum, and J Zico Kolter. End-to-end differentiable physics for learning and control. In *Advances in Neural Information Processing Systems*, 2018.

Jonas Degrave, Michiel Hermans, Joni Dambre, and Francis wyffels. A differentiable physics engine for deep learning in robotics. *Frontiers in Neurorobotics*, 13, 2019.

Patrick E Farrell, David A Ham, Simon W Funke, and Marie E Rognes. Automated derivation of the adjoint of high-level transient finite element programs. *SIAM Journal on Scientific Computing*, 35(4), 2013.

Ciprian Foias, George R Sell, and Roger Temam. Inertial manifolds for nonlinear evolutionary equations. *Journal of Differential Equations*, 73(2), 1988.

David A Fournier, Hans J Skaug, Johnoel Ancheta, James Ianelli, Arni Magnusson, Mark N Maunder, Anders Nielsen, and John Sibert. Ad model builder: Using automatic differentiation for statistical inference of highly parameterized complex nonlinear models. *Optimization Methods and Software*, 27(2), 2012.

Michael B Giles and Niles A Pierce. An introduction to the adjoint approach to design. *Flow, Turbulence and Combustion*, 65, 2000.

Justin Gilmer, Samuel S. Schoenholz, Patrick F. Riley, Oriol Vinyals, and George E. Dahl. Neural message passing for quantum chemistry. In *ICML*, 2017.

Gene H Golub and Charles F Van Loan. *Matrix Computations*. JHU press, 2012.

Andreas Griewank and Andrea Walther. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*. SIAM, 2nd edition, 2008.

Andreas Griewank, David Juedes, and Jean Utke. Algorithm 755: ADOL-C: A package for the automatic differentiation of algorithms written in C/C++. *ACM Transactions on Mathematical Software (TOMS)*, 22(2), 1996.

Charles R Hargraves and Stephen W Paris. Direct trajectory optimization using nonlinear programming and collocation. *Journal of Guidance, Control, and Dynamics*, 10(4), 1987.

Francis Harlow and Eddie Welch. Numerical calculation of time-dependent viscous incompressible flow of fluid with free surface. *Physics of Fluids*, 8(12), 1965.

Carsten Hartmann, Juan C Latorre, Wei Zhang, and Grigorios A Pavliotis. Optimal control of multiscale systems using reduced-order models. *Journal of Computational Dynamics*, 1(2), 2014.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *CVPR*, 2016.

Jun-Ting Hsieh, Shengjia Zhao, Stephan Eismann, Lucia Mirabella, and Stefano Ermon. Learning neural PDE solvers with convergence guarantees. In *ICLR*, 2019.

Yuanming Hu, Jiancheng Liu, Andrew Spielberg, Joshua B Tenenbaum, William T Freeman, Jiajun Wu, Daniela Rus, and Wojciech Matusik. ChainQueen: A real-time differentiable physical simulator for soft robotics. In *ICRA*, 2019.

John Ingraham, Adam Riesselman, Chris Sander, and Debora Marks. Learning protein structure with a differentiable simulator. In *ICLR*, 2019.

Antony Jameson. Aerodynamic design via control theory. *Journal of Scientific Computing*, 3(3), 1988.

Dieter Kraft. On converting optimal control problems into nonlinear programming problems. In *Computational Mathematical Programming*. 1985.

Lubor Ladicky, SoHyeon Jeong, Barbara Solenthaler, Marc Pollefeys, and Markus Gross. Data-driven fluid simulations using regression forests. *ACM Transactions on Graphics (TOG)*, 34(6), 2015.

Horace Lamb. *Hydrodynamics*. Cambridge University Press, 1932.

Daniel B Leineweber, Irene Bauer, Hans Georg Bock, and Johannes P Schlöder. An efficient multiple shooting based reduced SQP strategy for large-scale dynamic process optimization. Part 1: Theoretical aspects. *Computers & Chemical Engineering*, 27(2), 2003.

Randall J. LeVeque. *Numerical Methods for Conservation Laws*. Springer, 1992.

Yunzhu Li, Jiajun Wu, Russ Tedrake, Joshua B Tenenbaum, and Antonio Torralba. Learning particle dynamics for manipulating rigid bodies, deformable objects, and fluids. In *ICLR*, 2019.

Junbang Liang, Ming C. Lin, and Vladlen Koltun. Differentiable cloth simulation for inverse problems. In *Advances in Neural Information Processing Systems*, 2019.

Julia Ling, Andrew Kurzawski, and Jeremy Templeton. Reynolds averaged turbulence modelling using deep neural networks with embedded invariance. *Journal of Fluid Mechanics*, 807, 2016.

Jacques Louis Lions. *Optimal Control of Systems Governed by Partial Differential Equations*. Springer-Verlag, 1971.

Zichao Long, Yiping Lu, Xianzhong Ma, and Bin Dong. PDE-net: Learning PDEs from data. In *ICML*, 2018.

Dougal Maclaurin, David Duvenaud, and Ryan P. Adams. Autograd: Effortless gradients in Numpy. In *ICML Workshops*, 2015.

Antoine McNamara, Adrien Treuille, Zoran Popović, and Jos Stam. Fluid control using the adjoint method. *ACM Transactions on Graphics (TOG)*, 23(3), 2004.

Arthur Mensch and Mathieu Blondel. Differentiable dynamic programming for structured prediction and attention. In *ICML*, 2018.

Jeremy Morton, Antony Jameson, Mykel J Kochenderfer, and Freddie Witherden. Deep dynamical modeling and control of unsteady fluid flows. In *Advances in Neural Information Processing Systems*, 2018.

Damian Mrowca, Chengxu Zhuang, Elias Wang, Nick Haber, Fei-Fei Li, Josh Tenenbaum, and Daniel L. Yamins. Flexible neural representation for physics prediction. In *Advances in Neural Information Processing Systems*, 2018.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. PyTorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems*, 2019.

Olivier Pironneau. On optimum design in fluid mechanics. *Journal of Fluid Mechanics*, 64(1), 1974.

Lev Semenovich Pontryagin. *Mathematical Theory of Optimal Processes*. John Wiley, 1962.

Jovan Popovic, Steven M. Seitz, Michael A. Erdmann, Zoran Popovic, and Andrew P. Witkin. Interactive manipulation of rigid body simulations. In *SIGGRAPH*, 2000.

William H Press, Saul A Teukolsky, William T Vetterling, and Brian P Flannery. *Numerical Recipes*. Cambridge University Press, 3rd edition, 2007.

Maziar Raissi, Alireza Yazdani, and George Karniadakis. Hidden fluid mechanics: A Navier-Stokes informed deep learning framework for assimilating flow visualization data. *arXiv:1808.04327*, 2018.

Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation. In *Medical Image Computing and Computer-Assisted Intervention*, 2015.

Connor Schenck and Dieter Fox. SPNets: Differentiable fluid dynamics for deep neural networks. In *Conference on Robot Learning*, 2018.

Gordon D Smith. *Numerical Solution of Partial Differential Equations: Finite Difference Methods*. Oxford University Press, 1985.

Jos Stam. Stable Fluids. In *SIGGRAPH*, 1999.

Edriss S Titi. On approximate inertial manifolds to the Navier-Stokes equations. *Journal of Mathematical Analysis and Applications*, 149(2), 1990.

Seiya Tokui, Ryosuke Okuta, Takuya Akiba, Yusuke Niitani, Toru Ogawa, Shunta Saito, Shuji Suzuki, Kota Uenishi, Brian Vogel, and Hiroyuki Yamazaki Vincent. Chainer: A deep learning framework for accelerating the research cycle. In *Conference on Knowledge Discovery and Data Mining (KDD)*, 2019.

Jonathan Tompson, Kristofer Schlachter, Pablo Sprechmann, and Ken Perlin. Accelerating Eulerian fluid simulation with convolutional networks. In *ICML*, 2017.

Marc Toussaint, Kelsey Allen, Kevin Smith, and Joshua B Tenenbaum. Differentiable physics and stable modes for tool-use and manipulation planning. In *Robotics: Science and Systems*, 2018.

Adrien Treuille, Antoine McNamara, Zoran Popović, and Jos Stam. Keyframe control of smoke simulations. *ACM Transactions on Graphics (TOG)*, 22(3), 2003.

Adrien Treuille, Andrew Lewis, and Zoran Popović. Model reduction for real-time fluids. *ACM Transactions on Graphics (TOG)*, 25(3), 2006.

Benjamin Ummenhofer, Lukas Prantl, Nils Thürey, and Vladlen Koltun. Lagrangian fluid simulation with continuous convolutions. In *ICLR*, 2020.

Bart van Merriënboer, Dan Moldovan, and Alexander Wiltschko. Tangent: Automatic differentiation using source-code transformation for dynamically typed array programming. In *Advances in Neural Information Processing Systems*, 2018.

Hermann von Helmholtz. Über integrale der hydrodynamischen gleichungen, welcher der wirbelbewegungen entsprechen. *Journal für Mathematik Bd. LV. Heft*, 1:4, 1858.

Paul J Werbos. Backwards differentiation in AD and neural nets: Past links and new opportunities. In *Automatic Differentiation: Applications, Theory, and Implementations*. Springer, 2006.

You Xie, Erik Franz, Mengyu Chu, and Nils Thuerey. tempoGAN: A temporally coherent, volumetric GAN for super-resolution fluid flow. *ACM Transactions on Graphics (TOG)*, 37(4), 2018.

Kemin Zhou, John C. Doyle, and Keith Glover. *Robust and Optimal Control*. Prentice Hall, 1996.

A Implementation details of the differentiable PDE solver

Our solver is publicly available at https://github.com/tum-pbs/PhiFlow, licensed as MIT. It is implemented via existing machine learning frameworks to benefit from the built-in automatic differentiation and to enable tight integration with neural networks.

For the experiments shown here we used the popular machine learning framework TensorFlow (Abadi et al., 2016). However, our solver is written in a framework-independent way and also supports PyTorch (Paszke et al., 2019). Both frameworks allow for a low-level NumPy-like implementation which is well suited for basic PDE building blocks. The following paragraphs outline how we implemented these building blocks and how they can be put together to solve the PDEs shown in Section 6.

Staggered grids. Many of the experiments presented in Section 6 use PDEs which track velocities. We adopt the marker-and-cell method (Harlow & Welch, 1965), storing densities in a regular grid and velocity in a staggered grid. Unlike regular grids, where all components are sampled at the centers of grid cells, staggered grids sample vector fields in a staggered form. Each vector component is sampled in the center of the cell face perpendicular to that direction. This sampling allows for an exact formulation of the divergence of a staggered vector field, decreasing discretization errors in many cases. On the other hand, it complicates operations that combine vector fields with regular fields such as transport or density-dependent forces.

We use staggered grids for the velocities in all of our experiments. The buoyancy operation, which applies an upward force proportional to the smoke density, interpolates the density to the staggered grid. For the transport, also called *advection*, of regular or staggered fields, we interpolate the staggered field to grid cell centers or face centers, respectively. These interpolations are implemented in TensorFlow using basic tensor operations, similar to the differential operators. We implemented all differential operators that act on vector fields to support staggered grids as well.

Differential operators. For the experiments outlined in this paper, we have implemented the following differential operators:

- Gradient of scalar fields in any number of dimensions, ∇x
- Divergence of regular and staggered vector fields in any number of dimensions, $\nabla \cdot \boldsymbol{x}$
- Curl of staggered vector fields in 2D and 3D, $\nabla \times \boldsymbol{x}$
- Laplace of scalar fields in any number of dimensions, $\nabla^2 x$

All differential operators are local operations, i.e. they only act on a small neighbourhood of grid points. In the context of machine learning, this suggests implementing them as convolution operations with a fixed kernel. Indeed, all differential operators can be expressed this way and we have implemented some low-dimensional versions of them using this method.

This method does, however, scale poorly with the dimensionality of the physical system as the convolutional kernels pick up a large number of zeros, thus wasting computations. Therefore, we express n-dimensional differential operators using basic mathematical tensor operations.

Consider the gradient computation in 1D, which results in a staggered grid. Each resulting value is

$$(\nabla x)_i = x_i - x_{i-1},$$

assuming the result is staggered at the lower faces of each grid cell. This operation can be implemented as a 1D convolution with kernel $(-1, 1)$ or as a vector operation which subtracts the array from itself, shifted by one element.

In a low-dimensional setting, the convolution operation will be faster as it is highly optimized and can be executed on GPUs with one call. In higher dimensions, however, the vector-based version is faster and more practical because it avoids unnecessary computations and can be coded in a dimension-independent fashion. Both convolutions and basic mathematical operations are supported by all common machine learning frameworks, eliminating the need to implement custom gradient functions.

Advection. PDEs containing material derivatives can be solved using an advection step

$$f \leftarrow \mathrm{Advect}[f, \boldsymbol{v}]$$

which moves each value of a field f in the direction specified by a vector field $\boldsymbol{v}$. We implement the advection with semi-Lagrangian step (Stam, 1999) that looks back in time and supports regular and staggered vector fields.

To determine the advected value of a grid cell or face x_{target}, first $\boldsymbol{v}$ is interpolated to that point. Then the origin location is determined by following the vector backwards in time,

$$x_{\text{origin}} = x_{\text{target}} - \Delta t \cdot \boldsymbol{v}(x_{\text{target}}).$$

The final value is determined by linearly interpolating between the neighbouring grid cells around x_{origin}. All of these operations are, again, implemented using basic mathematical operations. Hence, gradients can be provided by the framework.

Poisson problems. Incompressible fluids, governed by the Navier-Stokes equations, are subject to the hard constraints $\nabla \cdot \boldsymbol{v} = 0$ and $\nabla \times p = 0$ where p denotes the pressure. A numerical solver can achieve this by finding a p such that these constraints are satisfied. This step is often called *Chorin Projection*, or *Helmholtz decomposition*, and is closely related to the fundamental theorem of vector calculus (von Helmholtz, 1858; Chorin, 1967). On a grid, solving for p is equal to solving a Poisson problem, i.e. a system of N linear equations, $\boldsymbol{A}p = \nabla \cdot u$ where N is the total number of grid cells. The $(N \cdot N)$ matrix A is sparse and its entries are located at predictable indices.

We numerically solve this Poisson problem with a conjugate gradient (CG) algorithm (Golub & Van Loan, 2012) that iteratively approximates p. Since hundreds of CG steps typically need to be performed for each Poisson solve, it is unfeasible to unroll this chain of iterations, and store all intermediate results in memory.

To ensure that the automatic differentiation chain is not broken, we instead solve the adjoint problem. For the pressure solve operation, the matrix A is symmetric and positive-definite. This causes the adjoint problem to have the same mathematical form (McNamara et al., 2004) as the original problem. Therefore we implement the gradient for the pressure solve by performing a pressure solve on the gradient. We believe that this is a good example of leveraging the methodology of adjoint method optimizations (Giles & Pierce, 2000; Treuille et al., 2006; Pontryagin, 1962) within deep learning. With this formalism we arrive at a differentiable solver framework that closely integrates numerical methods for forward problems with support for inverse problems such as deep learning via the adjoint method.

Solving Burger's equation and the Navier-Stokes equations. Using the basic building blocks outlined above, solving the PDEs becomes relatively simple. Burger's equation involves an advection and a diffusion term which we evaluate independently.

$$\text{Solver}[u] = \text{Diffuse}[\text{Advect}[u, u]]$$

where we explicitly compute $\text{Diffuse}[u] = u + \nu \nabla^2 u$ with viscosity ν. The advection is semi-Lagrangian with back-tracing as described above.

Solving the Navier-Stokes equations, typically comprises of the following steps:

- Transporting the density, $\rho \leftarrow \text{Advect}[\rho, \boldsymbol{v}]$
- Transporting the velocity, $\boldsymbol{v} \leftarrow \text{Advect}[\boldsymbol{v}, \boldsymbol{v}]$
- Applying diffusion if the viscosity is $\nu > 0$.
- Applying buoyancy force, $\boldsymbol{v} \leftarrow \boldsymbol{v} - \boldsymbol{\beta} \cdot \rho$ with buoyancy direction $\boldsymbol{\beta}$
- Enforcing incompressibility by solving for the pressure, $p \leftarrow \text{Solve}[Ap = \nabla \cdot \boldsymbol{v}]$, then $\boldsymbol{v} \leftarrow \boldsymbol{v} - \nabla p$

These steps are executed in this order to advance the simulation forward in time.

B Complexity of Execution Schemes

The staggered execution scheme recursively splits a sequence of length n into smaller sequences, as depicted in Fig. 3b and Fig. 7a for $n = 8$. With each level of recursion depth, the sequence length

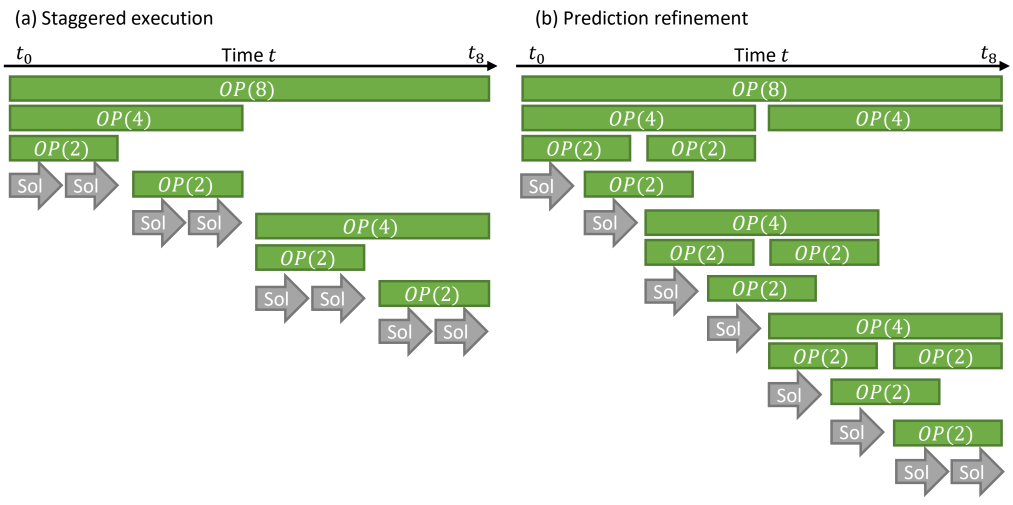


Figure 7: OP and CFE+Solver (Sol) executions for a sequence of length 8 performed by (a) the staggered execution scheme, (b) the prediction refinement scheme. The execution order is from top to bottom.

is cut in half and twice as many predictions need to be performed. The maximum depth depends on the sequence length $t_n - t_0$ and the time steps Δt performed by the solver,

$$d_{\text{max}} = \log_2 \left(\frac{t_n - t_0}{\Delta t} \right) - 1.$$

Therefore, the total number of predictions, equal to the number of OP evaluations, is

$$N_{\text{OP}} = 1 + 2 + 4 + ... + n/2 = \sum_{k=0}^{d_{\text{max}}} 2^k = n - 1.$$

The prediction refinement scheme performs more predictions, as can be seen in Fig. 7b. To understand the number of OP evaluations, we need to consider the recursive algorithm $\text{Reconstruct}[\boldsymbol{u}_0, \boldsymbol{o}_n, \boldsymbol{o}_{2n}]$, listed in Alg 1, that reconstructs a sequence or partial sequence of n frames. For the first invocation, the last parameter $\boldsymbol{o}_{2n}$ is absent, but for subsequences, that is not necessarily the case. Each invocation performs one OP evaluation if $\boldsymbol{o}_{2n}$ is absent, otherwise three. By counting the sequences for which this condition is fulfilled, we can compute the total number of network evaluations to be

$$N_{\text{OP}} = 3 \sum_{k=0}^{d_{\text{max}}} 2^k - 2\log_2(n) = 3n - 2\log_2(n) - 3.$$

C Network Architectures and Training

All neural networks used in this work are based on a modified U-net architecture (Ronneberger et al., 2015). The U-net represents a typical multi-level convolutional network architecture with skip connections, which we modify by using residual blocks (He et al., 2016) instead of regular convolutions for each level. We slightly modify this basic layout for some experiments.

The network used for predicting observations for the fluid example is detailed in Tab. 4. The input to the network are two feature maps containing the current state and the target state. Zero-padding is applied to the input, so that all strided convolutions do not require padding. Next, five residual blocks are executed in order, each decreasing the resolution (1/2, 1/4, 1/8, 1/16, 1/32) while increasing the number of feature maps (4, 8, 16, 16, 16). Each block performs a convolution with kernel size 2 and stride 2, followed by two residual blocks with kernel size 3 and symmetric padding. Inside each

Table 4: Layers comprising the observation predictor network used in the direct fluid control experiment.

Layer	Resolution	Feature Maps
Input	128	2
Pad	159	2
Strided convolution + 2x Residual block	79	4
Strided convolution + 2x Residual block	39	8
Strided convolution + 2x Residual block	19	16
Strided convolution + 2x Residual block	9	16
Strided convolution + 2x Residual block	4	16
3x Residual block	4	16
Upsample + Concatenate	8	32
Convolution + 2x Residual block	8	16
Upsample + Concatenate	16	32
Convolution + 2x Residual block	16	16
Upsample + Concatenate	32	24
Convolution + 2x Residual block	32	16
Upsample + Concatenate	64	20
Convolution + 2x Residual block	64	16
Upsample + Concatenate	128	18
Convolution	128	1

block, the number of feature maps stays constant. Three more residual blocks are executed on the lowest resolution of the bowtie structure, after which the decoder part of the network commences, translating features into spatial content.

The decoder works as follows: Starting with the lowest resolution, the feature maps are upsampled with linear interpolation. The upsampled maps and the output of the previous block of same resolution are then concatenated. Next, a convolution with 16 filters, a kernel size of 2 and symmetric padding, followed by two more residual blocks, is executed. When the original resolution is reached, only one feature map is produced instead of 16, forming the output of the network.

Depending on the dimensionality of the problem, either 1D or 2D convolutions are used. The network used for the indirect control task is modified in the following ways: (i) It produces two output feature maps, representing the velocity (v_x, v_y). (ii) Four feature maps of the lowest resolution (4x4) are fed into a dense layer producing four output feature maps. These and the other feature maps are concatenated before moving to the upsampling stage. This modification ensures that the receptive field of the network is the whole domain.

All networks were implemented in TensorFlow (Abadi et al., 2016) and trained using the ADAM optimizer on an Nvidia GTX 1080 Ti. We use batch sizes ranging from 4 to 16. Supervised training of all networks converges within a few minutes, for which we iteratively decrease the learning rate from 10^{-3} to 10^{-5}. We stop supervised training after a few epochs, comprising between 2000 and 10.000 iterations, as the networks usually converge within a fraction of the first epoch.

For training with the differentiable solver, we start with a decreased learning rate of 10^{-4} since the backpropagation through long chains is more challenging than training with a supervised loss. Optimization steps are also considerably more expensive since the whole chain needs to be executed, which includes a forward and backward simulation pass. For the fluid examples, an optimization step takes 1-2 seconds to complete for the 2D fluid problems. We let the networks run about 100.000 iterations, which takes between one and two days for the shown examples.

D Detailed description and analysis of the experiments

In the following paragraphs, we give further details on the experiments of Section 6.

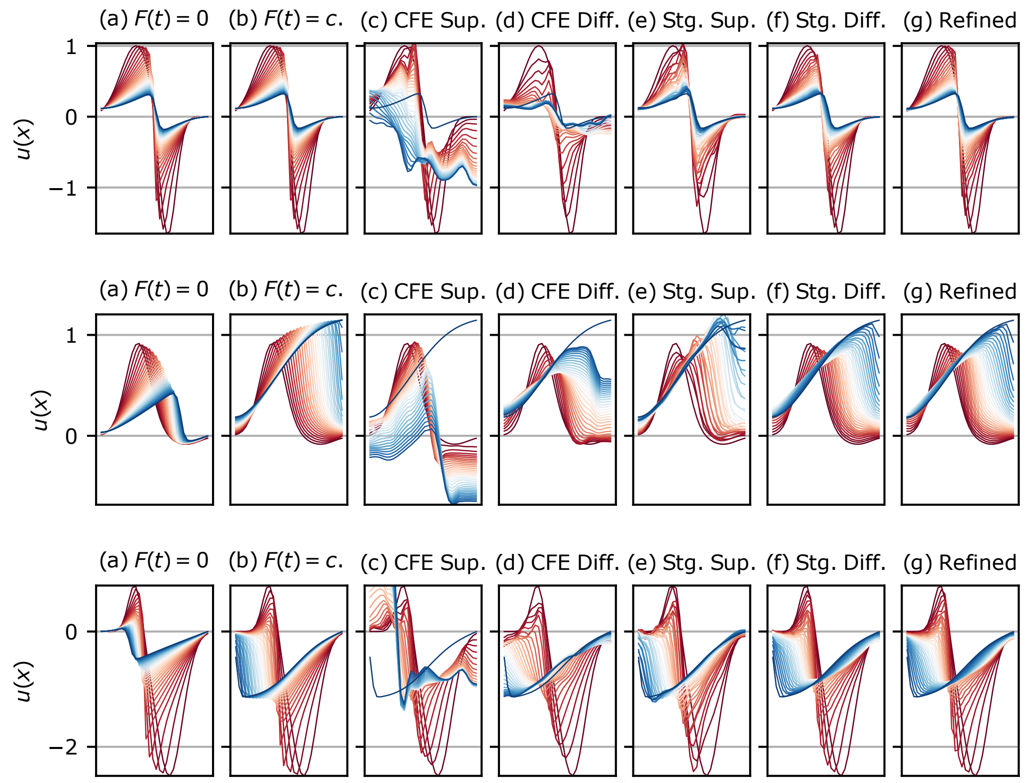


Figure 8: Trajectories for example control tasks using Burger's equation. The initial state is plotted in red, the target state in blue. (a) Natural evolution, (b) Ground truth trajectory generated with constant force, (c) CFE with supervised loss, (d) CFE with differentiable physics loss, (e) Reconstruction with supervised loss, (f) Reconstruction with differentiable physics loss and staggered execution, (g) Reconstruction with differentiable physics loss and prediction refinement.

D.1 BURGER'S EQUATION

For this experiment, we simulate Burger's equation (Eq. 6) on a one-dimensional grid with 32 samples over a course of 32 time steps. The typical behavior of Burger's equation in 1D exhibits shock waves that move in $+x$ or $-x$ direction for $u(x) > 0$ or $u(x) < 0$, respectively. When opposing waves clash, they both weaken until only the stronger wave survives and keeps moving. Examples are shown in Figs. 4a and 8a.

All 32 samples are observable and controllable, i.e. $o(t) = u(t)$. Thus, we can enforce that all trajectories reach the target state exactly by choosing the force for the last step to be

$$F(t_{n-1}) = \frac{o^* - u(t_{n-1})}{\Delta t}.$$

To measure the quality of a solution, it is therefore sufficient to consider the applied force $\int_{t_0}^{t_*} |F(t)| \, dt$ which is detailed for the tested methods in Table 1.

Network training. Both for the CFE chains as well as for the observation prediction models, we use the same network architecture, described in Appendix C. We train the networks on 3600 randomly generated scenes with constant driving forces, $F(t) = \text{const}$. The examples are initialized with two Gaussian waves of random amplitude, size and position, set to clash in the center. In each time step, a constant Gaussian force with the same randomized parameters is applied to the system to steer it away from its natural evolution. Constant forces have a larger impact on the evolution than temporally varying forces since the effects of temporally varying forces can partly cancel out over time. The ground truth sequence can therefore be regarded as a near-perfect but not necessarily optimal trajectory. Figs. 4d and 8b display such examples. The same trajectories, without any forces applied, are shown in sub-figures (a) for comparison.

We pretrain all networks (OPs or CFE, depending on the method) with a supervised observation loss,

$$L_o^{\text{sup}} = \left| \text{OP}[o(t_i), o(t_j)] - u^{\text{GT}}\left(\frac{t_i + t_j}{2}\right) \right|^2 . \tag{8}$$

The resulting trajectory after supervised training for the CFE chain is shown in Figure 4b and Figure 8c. For the observation prediction models, the trajectories are shown in Figure 4c and Figure 8e.

After pretraining, we train all OP networks end-to-end with our objective loss function (see Eq. 4), making use of the differentiable solver. For this experiment, we choose the mean squared difference for the observation loss function:

$$L_o^* = |o(u(t_*)) - o^*|^2 . \tag{9}$$

We test both the staggered execution scheme and the prediction refinement scheme, shown in Figure 8f and Figure 8g.

Results. Table 1 compares the resulting forces inferred by different methods. The results are averaged over a set of 100 examples from the test set which is sampled from the same distribution as the training set. The CFE chains both fail to converge to o^*. While the differentiable physics version manages to produce a u_{n-1} that resembles o^*, the supervised version completely deviates from an optimal trajectory. This shows that learning to infer the control force $F(t_i)$ only from $u(t_i)$, o^* and t is very difficult as the model needs to learn to anticipate the physical behavior over any length of time.

Compared to the CFE chains, the hierarchical models require much less force and learn to converge towards o^*. Still, the supervised training applies much more force to the system than required, the reasons for which become obvious when inspecting Figure 4b and Fig. 8e. While each state seems close to the ground truth individually, the control oscillates undesirably, requiring counter-actions later in time.

The methods using the differentiable solver significantly outperform their supervised counterparts and exhibit an excellent performance that is very close the ground truth solutions in terms of required forces. On many examples, they even reach the target state with less force than was applied by the ground truth simulation. This would not be possible with the supervised loss alone, but by having access to the gradient-based feedback from the differentiable solver, they can learn to find more efficient trajectories with respect to the objective loss. This allows the networks to learn applying forces in different locations that make the system approach the target state with less force.

Figure 4e and Fig.8f,g show examples of this. The ground truth applies the same force in each step, thereby continuously increasing the first sample $u(x = 0)$, and the supervised method tries to imitate this behavior. The governing equation then slowly propagates $u(x = 0)$ in positive x direction since $u(x = 0) > 0$. The learning methods that use a differentiable solver make use of this fact by applying much more force $F(x = 0) > 0$ at this point than the ground truth, even overshooting the target state. Later, when this value had time to propagate to the right, the model corrects this overshoot by applying a negative force $F(x = 0) < 0$. Using this trick, these models reach the target state with up to 13% less force than the ground truth on the sequence shown in Figure 4.

Figure 9 analyzes the variance of inferred forces. The supervised methods often fail to properly converge to the target state, resulting in large forces in the last step, visible as a second peak in the supervised CFE chain. The formulation of the loss (Eq. 3) suppresses force spikes. In the solutions inferred by our method, the likelihood of large forces falls off multi-exponentially as a consequence. This means that large forces are exponentially rare, which is the expected behavior given the L2 regularizer from Eq. 3.

We also compare our results to a single-shooting baseline which is able to find near-optimal solutions at the cost of higher computation times. The classic optimization uses the ADAM optimizer with a learning rate of 0.01 and converges after around 300 iterations. To reach the quality of the staggered prediction scheme, it requires only around 60 iterations. This quick convergence can be explained by the relatively simple setup that is dominated by linear effects. Therefore, the gradients are stable, even when propagated through many frames.

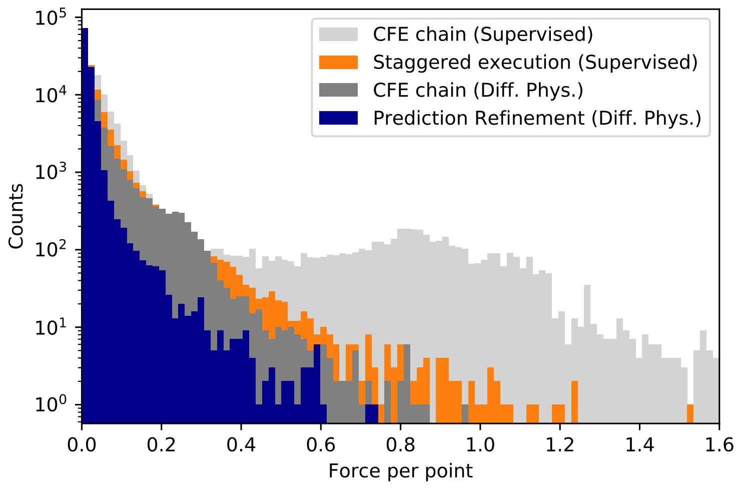


Figure 9: Histogram comparing the frequency of force strengths of different methods, summed over 100 examples from the Burger's experiment.

The computation times, shown in Tab. 1, were recorded on a single GTX 1080 Ti. We run 100 examples in parallel to reduce the relative overhead caused by GPU instruction queuing. For the network-based methods, we average the inference time over 100 runs. We perform 10 runs for the optimization methods.

D.2 Incompressible fluid flow

The incompressible Navier-Stokes equations model dynamics of fluids such as water or air, which can develop highly complex and chaotic behavior. The phenomenon of turbulence is generally seen as one of the few remaining fundamental and unsolved problems of classical physics. The challenging nature of the equations indicates that typically a very significant computational effort and a large number of degrees of freedom are required to numerically compute solutions. Here, we target an incompressible two-dimensional gas with viscosity ν, described by the Navier-Stokes equations for the velocity field $\boldsymbol{v}$. We assume a constant fluid density throughout the simulation, setting $\rho_f = \mathrm{const.} \equiv 1$. The gas velocity is controllable and, according to Eq. 1, we set

$$\mathcal{P}(\boldsymbol{v}, \nabla \boldsymbol{v}) = -(\boldsymbol{v} \cdot \nabla)\boldsymbol{v} + \nu \nabla^2 \boldsymbol{v} - \frac{\nabla p}{\rho_f}$$

subject to the hard constraints $\nabla \cdot \boldsymbol{v} = 0$ and $\nabla \times p = 0$. For our experiments, we target fluids with low viscosities, such as air, and set $\nu = 0$ in the equation above as the transport steps implicitly apply numerical diffusion that is on average higher than the targeted one. For fluids with a larger viscosity, the Poisson solver outlined above for computing p could be used to implicitly solve a vector-valued diffusion equation for $\boldsymbol{v}$.

However, incorporating a significant amount of viscosity would make the control problem easier to solve for most cases, as viscosity suppresses small scale structures in the motion. Hence, in order to create a challenging environment for training our networks, we have but a minimal amount of diffusion in the physical model.

In addition to the velocity field $\boldsymbol{v}$, we consider a smoke density distribution ρ which moves passively with the fluid. The evolution of ρ is described by the equation $\partial \rho / \partial t = -v \cdot \nabla \rho$. We treat the velocity field as hidden from observation, letting only the smoke density be observed, i.e. $o(t) = \rho(t)$. We stack the two fields as $u = (v, \rho)$ to write the system as one PDE, compatible with Eq. 1.

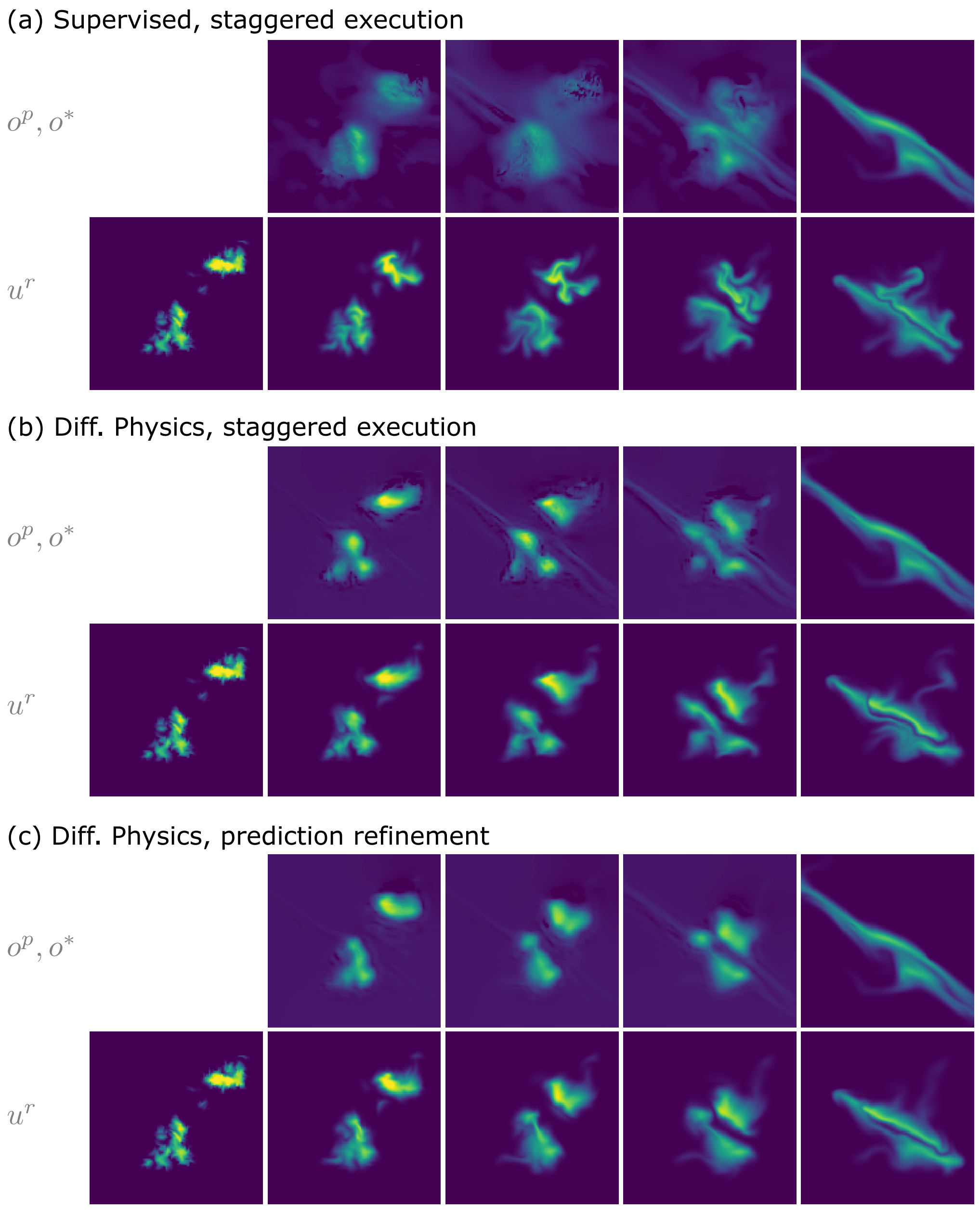


Figure 10: Reconstruction of an example natural flow sequence from the test set. The predictions $\boldsymbol{o}^p$ are plotted above the reconstructions u. The target state o^* is shown in the last column of the predictions.

For the OP and CFE networks, we use the 2D network architecture described in Appendix C. Instead of directly generating the velocity update in the CFE network for this problem setup, we make use of stream functions (Lamb, 1932). Hence, the CFE network outputs a vector potential Φ of which the curl $\nabla \times \Phi$ is used as a velocity update. This setup numerically simplifies the incompressibility condition of the Navier-Stokes equations but retains the same number of effective control parameters.

Datasets. We generate training and test datasets for two distinct tasks: flow reconstruction and shape transition. Both datasets have a resolution of 128×128 with the velocity fields being sampled in staggered form (see Appendix A). This results in over 16.000 effective continuous control parameters that make up the control force $\boldsymbol{F}(t_i)$ for each step i.

The flow reconstruction dataset is comprised of ground-truth sequences where the initial states $(\rho_0, \boldsymbol{v}_0)$ are randomly sampled and then simulated for 64 time steps. The resulting smoke density is then taken to be the target state, $\boldsymbol{o}^* \equiv \rho^* = \rho^{\text{sim}}(t_{64})$. Since we use fully convolutional networks for both CFE and OPs, the open domain boundary must be handled carefully. If smoke was lost from the simulation, because it crossed the outer boundary, a neural network would see the smoke simply vanish unless it was explicitly given the domain size as input. To avoid these problems, we run the simulation backwards in time and remove all smoke from ρ_0 that left the simulation domain.

For the shape transition dataset, we sample initial and target states ρ_0 and ρ_* by randomly choosing a shape from a library containing ten basic geometric shapes and placing it at a random location inside the domain. These can then be used for reconstructing sequences of any length n. For the results on shape transition presented in section 6, we choose $n = 16$ because all interesting behavior can be seen within that time frame. Due to the linear interpolation used in the advection step (see Appendix A), both ρ and $\boldsymbol{v}$ smear out over time. This numerical limitation makes it impossible to match target states exactly in this task as the density will become blurry over time. While we could generate ground-truth sequences using a classical optimizer, we refrain from doing so because (i) these trajectories are not guaranteed to be optimal and (ii) we want to see how well the model can learn from scratch, without initialization.

Training. We pretrain the CFE on the natural flow dataset with a supervised loss,

$$L_{\text{sup}}^{\text{CFE}}(\boldsymbol{u}(t)) = |\boldsymbol{v}_{\boldsymbol{u}(t)} + \boldsymbol{F}(t) - \boldsymbol{v}^*(t)|^2$$

where $\boldsymbol{v}^*(t)$ denotes the velocity from ground truth sequences. This supervised training alone constitutes a good loss for the CFE as it only needs to consider single-step intervals Δt while the OPs handle longer sequences. Nevertheless, we found that using the differentiable solver with an observation loss,

$$L_{\boldsymbol{o}}^{\text{CFE}} = |B_r(\boldsymbol{o}^*) - B_r\left(\text{Solver}[\boldsymbol{u} + \text{CFE}[\boldsymbol{u}, \boldsymbol{o}^*]]\right)|^2,$$

further improves the accuracy of the inferred force without sacrificing the ground truth match. Here $B_r(x)$ denotes a blur function with a kernel of the form $\frac{1}{1+x/r}$. The blur helps make the gradients smoother and creates non-zero gradients in places where prediction and target do not overlap. During training, we start with a large radius of $r = 16\,\Delta x$ for B_r and successively decrease it to $r = 2\,\Delta x$. We choose α such that $L_{\boldsymbol{F}}$ and $L_{\boldsymbol{o}}^*$ are of the same magnitude when the force loss spikes (see Fig. 15).

After the CFE is trained, we successively train the OPs starting with the smallest time scale. For the OPs, we train different models for natural flow reconstruction and shape transition, both based on the same CFE model. We pre-train all OPs independently with a supervised observation loss before jointly training them end-to-end with objective loss function (Eq. 4) and the differentiable solver to find the optimal trajectory. We use the OPs trained with the staggered execution scheme as initialization for the prediction refinement scheme. The complexity of solving the Navier-Stokes equations over many time steps in this example requires such a fully supervised initialization step. Without it, this setting is so non-linear that the learning process does not converge to a good solution. Hence, it illustrates the importance of combining supervised and unsupervised (requiring differentiable physics) training for challenging learning objectives.

A comparison of the different losses is shown in Fig. 10. The predictions, shown in the top rows of each subfigure, illustrate the differences between the three methods. The supervised predictions, especially the long-term predictions (central images), are blurry because the network learns to average over all ground truth sequences that match the given initial and target state. The differentiable

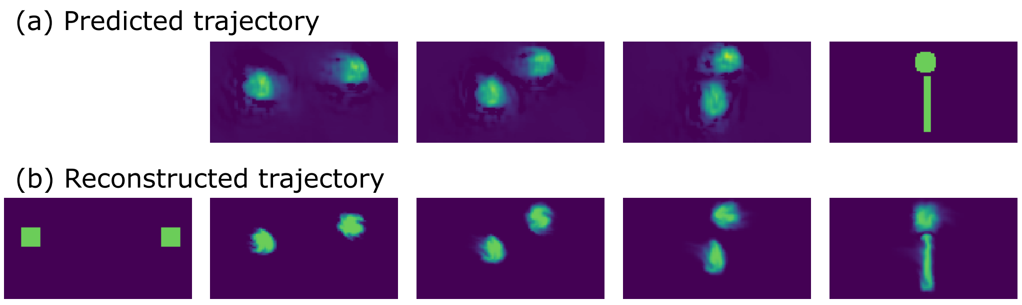


Figure 11: Reconstruction of multiple shapes using prediction refinement. The CFE is trained on the natural flow dataset and OPs are trained on single-shape transitions. The predictions of all shapes are added and passed to the CFE as one prediction.

physics solver largely resolves this issue. The predictions are much sharper but the long-term predictions still do not account for short-term deviations. This can be seen in the central prediction of Fig. 10b which shows hints of the target state $\boldsymbol{o}^*$, despite the fact that the actual reconstruction $\boldsymbol{u}$ cannot reach that state at that time. The refined prediction, shown in subfigure (c), is closer to $\boldsymbol{u}$ since it is conditioned on the previous reconstructed state.

In the training data, we let the network transform one shape into another at a random location. The differentiable solver and the long-term intuition provided by our execution scheme make it possible to train networks that can infer accurate sequences of control forces. In most cases, the target shapes are closely matched. As our networks infer sequences over time, we refer readers to the supplemental material (https://ge.in.tum.de/publications/2020-iclr-holl), which contains animations of additional sequences.

Generalization to multiple shapes. Splitting the reconstruction task into prediction and correction has the additional benefit of having full access to the intermediate predictions $\boldsymbol{o}^p$. These model real states of the system so classical processing or filter operations can be applied to them as well. We demonstrate this by generalizing our method to $m > 1$ shapes that evolve within the same domain. Figure 11 shows an example of two weakly-interacting shape transitions. We implement this by executing the OPs independently for each transition $k \in \{1, 2, ...m\}$ while inferring the control force $\boldsymbol{F}(t)$ on the joint system. This is achieved by adding the predictions of the smoke density ρ before passing it to the CFE network, $\tilde{\boldsymbol{o}^p} = \sum_{k=1}^{m} \boldsymbol{o}_k^p$. The resulting force is then applied to all sequences individually so that smoke from one transition does not end up in another target state. Using this scheme, we can define start and end positions for arbitrarily many shapes and let them evolve together.

Evaluation of force strengths The average force strengths are detailed in Tab. 2 while Figure 12 gives a more detailed analysis of the force strengths. As expected from using a L2 regularizer on the force, large values are exponentially rare in the solutions inferred from our test set. None of the hierarchical execution schemes exhibit large outliers. The prediction refinement requires the least amount of force to match the target, slightly ahead of the staggered execution trained with the same loss. The supervised training produces trajectories with reduced continuity that result in larger forces being applied.

D.3 Incompressible fluid with indirect control

As a fourth test environment, we target a case with increased complexity, where the network does not have the means anymore to directly control the full fluid volume. Instead, the network can only apply forces in the peripheral regions, with a total of more than 5000 control parameters per step. The obstacles prevent fluid from passing through them and the domain is enclosed with solid boundaries from the left, right and bottom. This leads to additional hard constraints and interplays between constraints in the physical model, and as such provides an interesting and challenging test case for our method. The domain has three target regions (*buckets*) separated by walls at the top of the domain, into which a volume of smoke should be transported from any position in the center

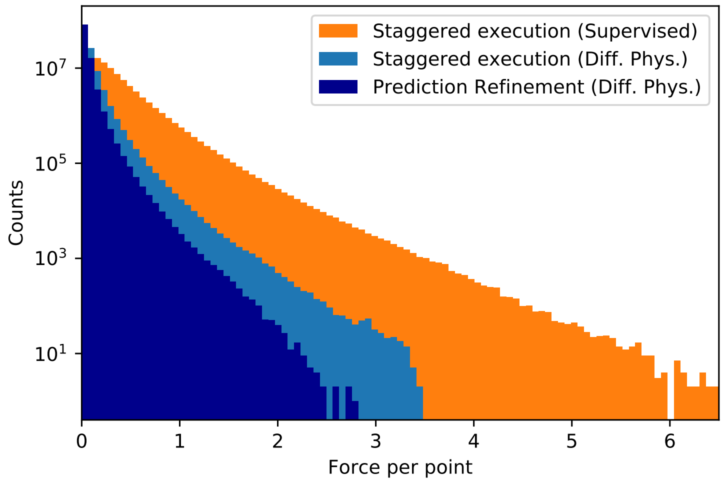


Figure 12: Histogram comparing the frequency of force strengths applied in the direct fluid control experiment on the natural flow dataset, summed over 100 examples.

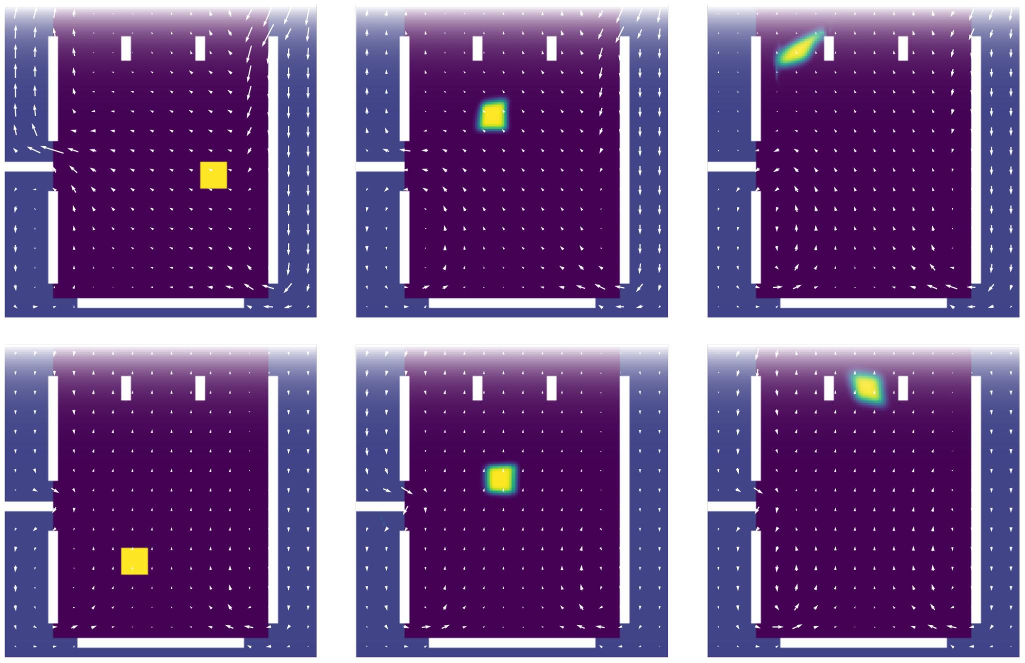


Figure 13: Two reconstructed trajectories from the test set of the indirect smoke control problem.

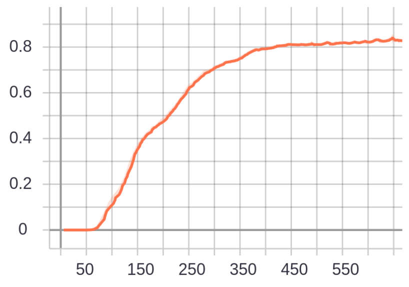


Figure 14: Iterative optimization of the indirect incompressible fluid control problem. The graph shows the fraction of smoke that ends up in the correct bucket vs number of optimization steps, averaged over 10 examples.

part. Both initial position and the target bucket are randomized for our training set of 3600 examples and test set of 100 examples. Each sequence consists of 16 time steps.

In this case the control is indirect since the smoke density lies outside the controlled area at all times. Only the incompressibility condition allows the network to influence the velocity outside the controlled area. This forces the model to consider the global context and synchronize a large number of parameters to create a desired flow field. The requirement of complex synchronized force fields makes generating reliable training data difficult, as manual or random sampling is unlikely to produce a directed velocity field in the center. We therefore skip the pretraining process and directly train the CFE using the differentiable solver, while the OP networks are trained as before with $r = 2\,\Delta x$.

To evaluate how well the learning method performs, we measure how much of the smoke density ends up inside the buckets and how much force was applied in total. For reference, we replace the observation predictions with an algorithm that moves the smoke towards the bucket in a straight line. Averaged over 100 examples from the test set, the resulting model manages to put $89\% \pm 2.6\%$ of the smoke into the target bucket. In contrast, the model trained with our full algorithm moves $99.22\% \pm 0.15\%$ of the smoke into the target buckets while requiring $19.1\% \pm 1.0\%$ less force.

We also compare our method to an iterative optimization which directly optimizes the control velocities. We use the ADAM optimizer with a learning rate of 0.1. Despite the highly non-linear setup, the gradients are stable enough to quickly let the smoke flow in the right direction. Fig. 14 shows how the trajectories improve during optimization. After around 60 optimization steps, the smoke distribution starts reaching the target bucket in some examples. Over the next 600 iterations, it converges to a a configuration in which 82.1 ± 7.3 of the smoke ends up in the correct bucket.

D.4 Comparison to Shooting Methods

We compare the sequences inferred by our trained models to classical shooting optimizations using our differentiable physics solver to directly optimize $F(t)$ with the objective loss L (Eq. 4) for a single input. We make use of stream functions (Lamb, 1932), as in the second experiment, to ensure the incompressibility condition is fulfilled. For this comparison, the velocities of all steps are initialized with a normal distribution with $\mu = 0$ and $\sigma = 0.01$ so that the initial trajectory does not significantly alter the initial state, $u(t) \approx u(t_0)$.

We first show how a simple single-shooting algorithm (Zhou et al., 1996) fares with our Navier-Stokes setup. When solving the resulting optimization problem using single-shooting, strong artifacts in the reconstructions can be observed, as shown in Figure 17a. This undesirable behavior stems from the nonlinearity of the Navier-Stokes equations, which causes the gradients $\boldsymbol{\Delta u} \gg 0$ to become noisy and unreliable when they are recurrently backpropagated through many time steps. Unsurprisingly, the single-shooting optimizer converges to a undesirable local minimum.

As single-shooting is well known to have problems with non-trivial problem settings, we employ a multi-scale shooting (MS) method (Hartmann et al., 2014). This solver first computes the trajectory on a coarsely discretized version of the problem before iteratively refining the discretization. For

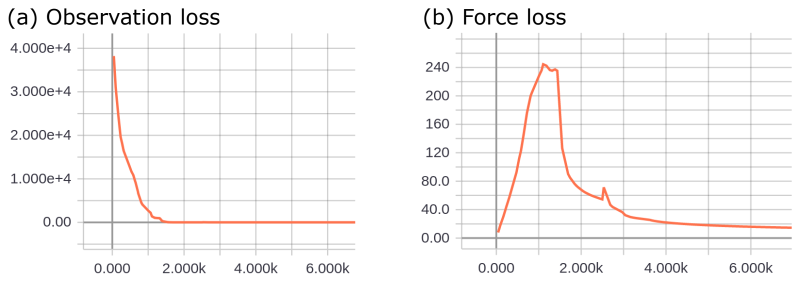


Figure 15: Mean convergence curves of the adjoint method optimization for 100 shape transitions.

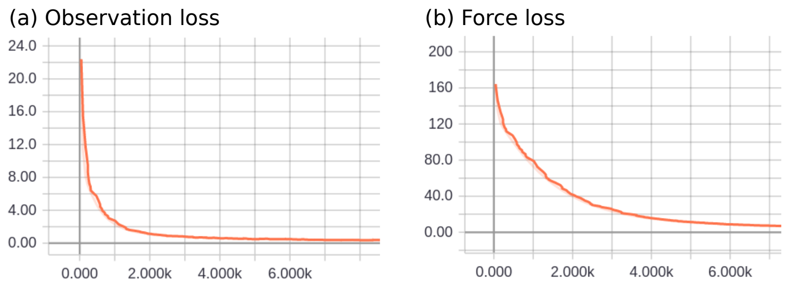


Figure 16: Mean convergence curves of the adjoint method optimization for 100 shape transitions, taking the reconstruction from the refinement scheme as initial guess.

the first resolution, we use 1/16 of the original width and height which both reduces the number of control parameters and reduces nonlinear effects from the physics model. By employing an exponential learning rate decay, this multi-scale optimization converges reliably for all examples. We use the ADAM optimizer to compute the control variable updates from the gradients of the differentiable Navier-Stokes solver.

An averaged set of representative convergence curves for this setup is shown in Figure 15. The objective loss (Eq. 4) is shown in its decomposed state as the sum of the observation loss $L_{\boldsymbol{o}}^*$, shown in Figure 15a, and the force loss $L_{\boldsymbol{F}}$, shown in Figure 15b. Due to the initialization of all velocities with small values, the force loss starts out small. For the first 1000 iteration steps, $L_{\boldsymbol{o}}^*$ dominates which causes the system to move towards the target state $\boldsymbol{o}^*$. This trajectory is not ideal, however, as more force than necessary is applied. Once observation loss and force loss are of the same magnitude, the optimization refines the trajectory to use less force.

We found that the trajectories predicted by our neural network based method correspond to performing about 1500 steps with the MS optimization while requiring less tuning. Reconstructions of the same example are compared in Figure 17. Performing the MS optimization up to this point took 131 seconds on a GTX 1080 Ti graphics card for a single 16-frame sequence while the network inference ran for 0.5 seconds. For longer sequences, this gap grows further because the network inference time scales with $\mathcal{O}(n)$. This could only be matched if the number of iterations for the MS optimization scaled with $O(1)$, which is not the case for most problems. These tests indicate that our model has successfully internalized the behavior of a large class of physical behavior, and can exert the right amount of force to reach the intended goal. The large number of iterations required for the single-case shooting optimization highlights the complexity of the individual solutions.

Interestingly, the network also benefits from the much more difficult task to learn a whole manifold of solutions: comparing solutions with similar observation loss for the MS algorithm and our network, the former often finds solutions that are unintuitive and contain noticeable detours, e.g., not taking a straight path for the density matching examples of Fig. 5. In such situations, our network benefits from having to represent the solution manifold, instead of aiming for single task optimizations. As the solutions are changing relatively smoothly, the complex task effectively regularizes the inference of new solutions and gives the network a more global view. Instead, the shooting optimiza-

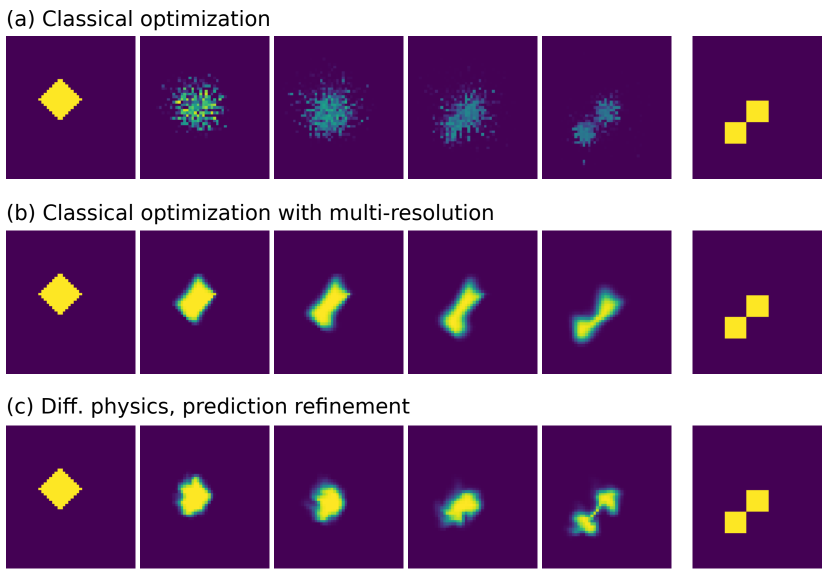


Figure 17: Example reconstruction of a shape transition. (a) Direct shooting optimization, 2300 iterations, (b) multi-scale shooting optimization, 1500 iterations, (c) output of our neural network based method with prediction refinement. Our model infers the shown solution in a single pass, and generalizes to a large class of inputs.

tions have to purely rely on local gradients for single-shooting or manually crafted multi-resolution schemes for MS.

Our method can also be employed to support the MS optimization by initializing it with the velocities inferred by the networks. In this case, shown in Figure 16, both L_o^* and L_F decrease right from the beginning, similar to the behavior in Figure 15 from iteration 1500 on. The reconstructed trajectory from the neural-network-based method is so close to the optimum that the multi-resolution approach described above is not necessary.

D.5 Additional Results

In Fig. 18, we provide a visual overview of a sub-set of the sequences that can be found in the supplemental materials. It contains 16 randomly selected reconstructions for each of the natural flow, the shape transitions, and the indirect control examples. In addition, the supplemental material, available at https://ge.in.tum.de/publications/2020-iclr-holl, highlights the differences between unsupervised, staggered, and refined versions of our approach.

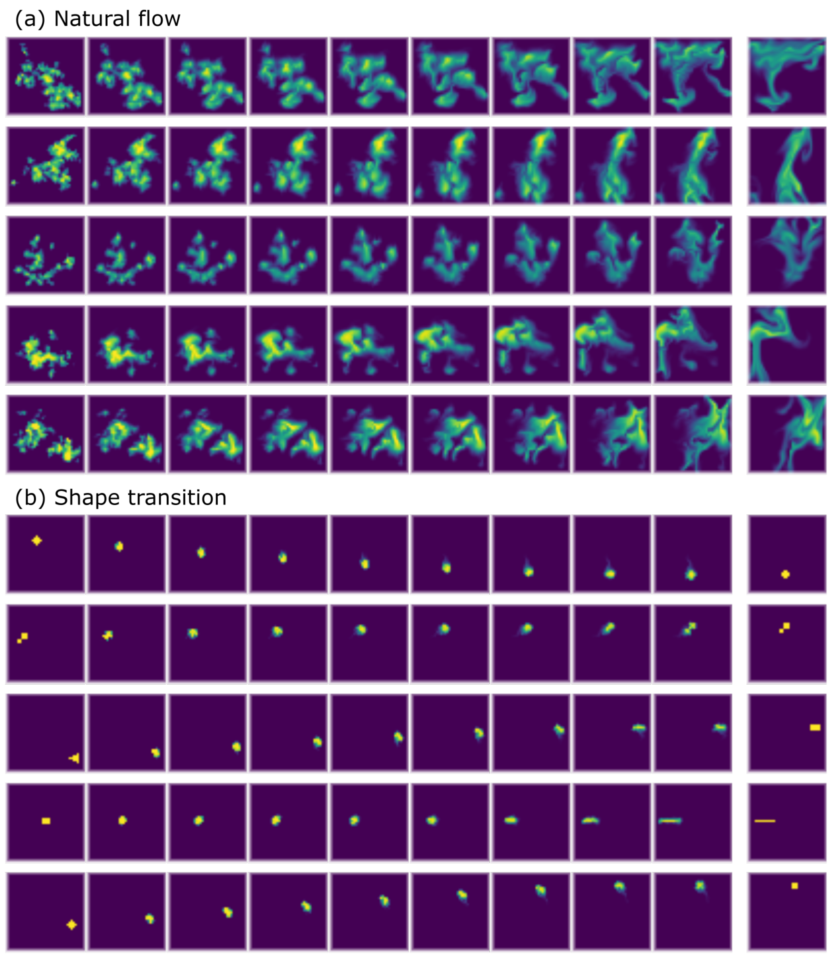


Figure 18: Five additional sequences from the test sets of the natural flow and shape transition setups. The first nine frames contain frames from our reconstruction. The far right image shows the target.

HALF-INVERSE GRADIENTS FOR PHYSICAL DEEP LEARNING

Patrick Schnell, Philipp Holl and Nils Thuerey
Department of Informatics
Technical University of Munich
Boltzmannstr. 3, 85748 Garching, Germany
{patrick.schnell,philipp.holl,nils.thuerey}@tum.de

ABSTRACT

Recent works in deep learning have shown that integrating differentiable physics simulators into the training process can greatly improve the quality of results. Although this combination represents a more complex optimization task than supervised neural network training, the same gradient-based optimizers are typically employed to minimize the loss function. However, the integrated physics solvers have a profound effect on the gradient flow as manipulating scales in magnitude and direction is an inherent property of many physical processes. Consequently, the gradient flow is often highly unbalanced and creates an environment in which existing gradient-based optimizers perform poorly. In this work, we analyze the characteristics of both physical and neural network optimizations to derive a new method that does not suffer from this phenomenon. Our method is based on a half-inversion of the Jacobian and combines principles of both classical network and physics optimizers to solve the combined optimization task. Compared to state-of-the-art neural network optimizers, our method converges more quickly and yields better solutions, which we demonstrate on three complex learning problems involving nonlinear oscillators, the Schrödinger equation and the Poisson problem.

1 INTRODUCTION

The groundbreaking successes of deep learning (Krizhevsky et al., 2012; Sutskever et al., 2014; Silver et al., 2017) have led to ongoing efforts to study the capabilities of neural networks across all scientific disciplines. In the area of physical simulation, neural networks have been used in various ways, such as creating accurate reduced-order models (Morton et al., 2018), inferring improved discretization stencils (Bar-Sinai et al., 2019), or suppressing numerical errors (Um et al., 2020). The long-term goal of these methods is to exceed classical simulations in terms of accuracy and speed, which has been achieved, e.g., for rigid bodies (de Avila Belbute-Peres et al., 2018), physical inverse problems (Holl et al., 2020), and two-dimensional turbulence (Kochkov et al., 2021).

The successful application of deep learning to physical systems naturally hinges on the training setup. In recent years, the use of physical loss functions has proven beneficial for the training procedure, yielding substantial improvements over purely supervised training approaches (Tompson et al., 2017; Wu & Tegmark, 2019; Greydanus et al., 2019). These improvements were shown to stem from three aspects (Battaglia et al., 2016; Holl et al., 2020): (i) Incorporating prior knowledge from physical principles facilitates the learning process , (ii) the ambiguities of multimodal cases are resolved naturally, and (iii) simulating the physics at training time can provide more realistic data distributions than pre-computed data sets. Approaches for training with physical losses can be divided into two categories. On the one hand, equation-focused approaches that introduce physical residuals (Tompson et al., 2017; Raissi et al., 2019), and on the other hand, solver-focused approaches that additionally integrate well-established numerical procedures into training (Um et al., 2020; Kochkov et al., 2021).

From a mathematical point of view, training a neural network with a physical loss function bears the difficulties of both network training and physics optimization. In order to obtain satisfying results, it is vital to treat flat regions of the optimization landscapes effectively. In learning, the challenging loss landscapes are addressed using gradient-based optimizers with data-based normalizing schemes, such as Adam (Kingma & Ba, 2015), whereas in physics, the optimizers of choice are higher-order techniques, such as Newton's method (Gill & Murray, 1978), which inherently make use of inversion processes. However, Holl et al. (2021) found that these approaches can not effectively handle the joint optimization of network and physics. Gradient-descent-based optimizers suffer from vanishing or exploding gradients, preventing effective convergence, while higher-order methods do not generally scale to the high-dimensional parameter spaces required by deep learning (Goodfellow et al., 2016).

Inspired by the insight that inversion is crucial for physics problems in learning from Holl et al. (2021), we focus on an inversion-based approach but propose a new method for joint physics and network optimization which we refer to as *half-inverse gradients*. At its core lies a partial matrix inversion, which we derive from the interaction between network and physics both formally and geometrically. An important property of our method is that its runtime scales linearly with the number of network parameters. To demonstrate the wide-ranging and practical applicability of our method, we show that it yields significant improvements in terms of convergence speed and final loss values over existing methods. These improvements are measured both in terms of absolute accuracy as well as wall-clock time. We evaluate a diverse set of physical systems, such as the Schrödinger equation, a nonlinear chain system and the Poisson problem.

2 Gradients based on Half-Inverse Jacobians

Optimization on continuous spaces can be effectively performed with derivative-based methods, the simplest of which is gradient descent. For a target function $L(\boldsymbol{\theta})$ to be minimized of several variables $\boldsymbol{\theta}$, using bold symbols for vector-valued quantities in this section, and learning rate η, gradient descent proceeds by repeatedly applying updates

$$\Delta\boldsymbol{\theta}_{\mathrm{GD}}(\eta) = -\eta \cdot \left(\frac{\partial L}{\partial \boldsymbol{\theta}}\right)^{\top}. \tag{1}$$

For quadratic objectives, this algorithm convergences linearly with the rate of convergence depending on the condition number λ of the Hessian matrix (Lax, 2014). In the ill-conditioned case $\lambda \gg 1$, flat regions in the optimization landscape can significantly slow down the optimization progress. This is a ubiquitous problem in non-convex optimization tasks of the generic form:

$$L(\boldsymbol{\theta}) = \sum_i l\big(\boldsymbol{y}_i(\boldsymbol{\theta}), \hat{\boldsymbol{y}}_i\big) = \sum_i l\big(\boldsymbol{f}(\boldsymbol{x}_i;\boldsymbol{\theta}), \hat{\boldsymbol{y}}_i\big) \tag{2}$$

Here $(\boldsymbol{x}_i, \hat{\boldsymbol{y}}_i)$ denotes the ith data points from a chosen set of measurements, $\boldsymbol{f}$ is a function parametrized by $\boldsymbol{\theta}$ to be optimized to model the relationship between the data points $\boldsymbol{y}_i(\boldsymbol{\theta}) = \boldsymbol{f}(\boldsymbol{x}_i;\boldsymbol{\theta})$, and l denotes a loss function measuring the optimization progress. In the following, we assume the most common case of $l(\boldsymbol{y}_i, \hat{\boldsymbol{y}}_i) = \frac{1}{2}||\boldsymbol{y}_i - \hat{\boldsymbol{y}}_i||_2^2$ being the squared L2-loss.

Physics Optimization. Simulating a physical system consists of two steps: (i) mathematically modeling the system by a differential equation, and (ii) discretizing its differential operators to obtain a solver for a computer. Optimization tasks occur for instance when manipulating a physical system through an external force to reach a given configuration, for which we have to solve an inverse problem of form 2. In such a control task, the sum reduces to a single data point $(\boldsymbol{x}, \hat{\boldsymbol{y}})$ with $\boldsymbol{x}$ being the initial state, $\hat{\boldsymbol{y}}$ the target state and $\boldsymbol{\theta}$ the external force we want to find. The physical solver corresponds to the function $\boldsymbol{f}$ representing time evolution $\boldsymbol{y}(\boldsymbol{\theta}) = \boldsymbol{f}(\boldsymbol{x};\boldsymbol{\theta})$. This single data point sum still includes summation over vector components of $\boldsymbol{y} - \hat{\boldsymbol{y}}$ in the L2-loss. Sensitive behavior of the physical system arising from its high-frequency modes is present in the physical solver $\boldsymbol{f}$, and produces small singular values in its Jacobian. This leads to an ill-conditioned Jacobian and flat regions in the optimization landscape when minimizing 2. This is addressed by using methods that incorporate more information than only the gradient. Prominent examples are Newton's method or the Gauss-Newton's algorithm (Gill & Murray, 1978); the latter one is based on the Jacobian of $\boldsymbol{f}$ and the loss gradient:

$$\Delta\boldsymbol{\theta}_{\mathrm{GN}} = -\left(\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{\theta}}\right)^{-1} \cdot \left(\frac{\partial L}{\partial \boldsymbol{y}}\right)^{\top} \tag{3}$$

Here the inversion of the Jacobian is calculated with the pseudoinverse. The Gauss-Newton update maps the steepest descent direction in $\boldsymbol{y}$-space to the parameter space $\boldsymbol{\theta}$. Therefore, to first order, the resulting update approximates gradient descent steps in $\boldsymbol{y}$-space, further details are given in appendix A.2. An advantage of such higher-order methods is that the update steps in $\boldsymbol{y}$-space are invariant under arbitrary rescaling of the parameters $\boldsymbol{\theta}$, which cancels inherent scales in $\boldsymbol{f}$ and ensures quick progress in the optimization landscape.

Neural Network Training. For $\boldsymbol{f}$ representing a neural network in equation 2, the optimization matches the typical supervised learning task. In this context, the problem of flat regions in the optimization landscape is also referred to as pathological curvature (Martens, 2010). Solving this problem with higher-order methods is considered to be too expensive given the large number of parameters $\boldsymbol{\theta}$. For learning tasks, popular optimizers, such as Adam, instead use gradient information from earlier update steps, for instance in the form of momentum or adaptive learning rate terms, thereby improving convergence speed at little additional computational cost. Furthermore, the updates are computed on mini-batches instead of the full data set, which saves computational resources and benefits generalization (Goodfellow et al., 2016).

Neural Network Training with Physics Objectives. For the remainder of the paper, we consider joint optimization problems, where $\boldsymbol{f}$ denotes a composition of a neural network parameterized by $\boldsymbol{\theta}$ and a physics solver. Using classical network optimizers for minimizing equation 2 is inefficient in this case since data normalization in the network output space is not possible and the classical initialization schemes cannot normalize the effects of the physics solver. As such, they are unsuited to capture the strong coupling between optimization parameters typically encountered in physics applications. While Gauss-Newton seems promising for these cases, the involved Jacobian inversion tends to result in large overshoots in the updates when the involved physics solver is ill-conditioned. As we will demonstrate, this leads to oversaturation of neurons, hampering the learning capability of the neural network.

2.1 An Ill-Conditioned Toy Example

To illustrate the argumentation so far, we consider a data set sampled from $\hat{\boldsymbol{y}}(x) = (\sin(6x), \cos(9x))$ for $x \in [-1, 1]$: We train a neural network to describe this data set by using the loss function:

$$l(\boldsymbol{y}, \hat{\boldsymbol{y}}; \gamma) = \frac{1}{2}\left(y^1 - \hat{y}^1\right)^2 + \frac{1}{2}\left(\gamma \cdot y^2 - \hat{y}^2\right)^2 \tag{4}$$

Here, we denote vector components by superscripts. For a scale factor of $\gamma = 1$, we receive the well-conditioned mean squared error loss. However, l becomes increasingly ill-conditioned as γ is decreased, imitating the effects of a physics solver. For real-world physics solvers, the situation would be even more complex since these scales usually vary strongly in direction and magnitude across different data points and optimization steps. We use a small neural network with a single hidden layer with 7 neurons and a $\tanh$ activation. We then compare training with the well-conditioned $\gamma = 1$ loss against an ill-conditioned $\gamma = 0.01$ loss. In both cases, we train the network using both Adam and Gauss-Newton as representatives of gradient-based and higher-order optimizers, respectively. The results are shown in figure 1.

In the well-conditioned case, Adam and Gauss-Newton behave similarly, decreasing the loss by about three orders of magnitude. However, in the ill-conditioned case, both optimizers fail to minimize the objective beyond a certain point. To explain this observation, we first illustrate the behavior from the physics viewpoint by considering the trajectory of the network output $\boldsymbol{f}(x)$ for a single value x during training (figure 1, right). For $\gamma = 1$, Adam optimizes the network to accurately predict $\hat{\boldsymbol{y}}(x)$ while for $\gamma = 0.01$, the updates neglect the second component preventing Adam to move efficiently along the small-scale coordinate (blue curve in figure 1b, right). To illustrate the situation from the viewpoint of the network, we consider the variance in the outputs of specific neurons over different x (figure 1, middle). When $\gamma = 1$, all neurons process information by producing different outcomes for different x. However, for $\gamma = 0.01$, Gauss-Newton's inversion of the small-scale component y^2 results in large updates, leading to an oversaturation of neurons (red curve in figure 1b, middle). These neurons stop processing information, reducing the effective capacity of the network and preventing the network from accurately fitting $\hat{\boldsymbol{y}}$. Facing these problems, a natural questions arises: ***Is it possible to construct an algorithm that can successfully process the inherently different scales of a physics solver while training a neural network at the same time?***

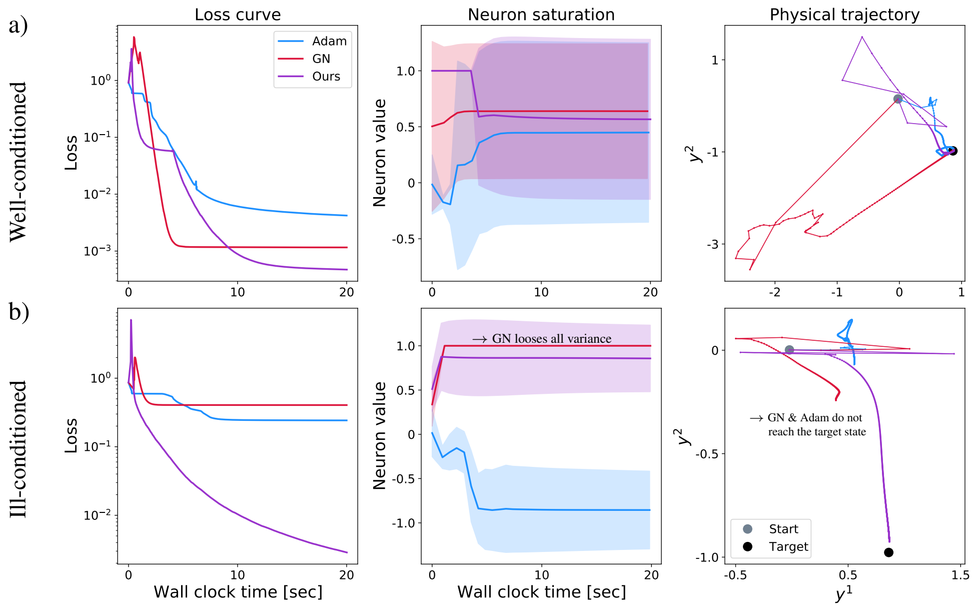


Figure 1: Results of the learning problem of section 2.1. Optimization is performed with a) a well-conditioned loss and b) an ill-conditioned loss. Plots show loss curves over training time (left), data set mean and standard deviation of the output of a neuron output over training time (middle), and the training trajectory of a data point (right).

2.2 Updates based on Half-Inverse Jacobians

We propose a novel method for optimizing neural networks with physics objectives. Since pure physics or neural network optimization can be thought of as special cases of the joint optimization, we analogously look for a potential method in the continuum of optimization methods between gradient descent and Gauss-Newton. We consider both of them to be the most elementary algorithms representing network and physics optimizers, respectively. The following equation describes updates that lie between the two.

$$\Delta\boldsymbol{\theta}(\eta,\kappa) = -\eta \cdot \left(\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{\theta}}\right)^{\kappa} \cdot \left(\frac{\partial L}{\partial \boldsymbol{y}}\right)^{\top} \tag{5}$$

Here, the exponent κ of the Jacobian denotes the following procedure defined with the aid of the singular value decomposition $J = U\Lambda V^{\top}$:

$$J^{\kappa} := V\Lambda^{\kappa}U^{\top} \tag{6}$$

When $\kappa = 1$, equation 5 reduces to the well-known form of gradient descent. Likewise, the case $\kappa = -1$ yields Gauss-Newton since the result of the Jacobian exponentiation then gives the pseudoinverse of the Jacobian. Unlike other possible interpolations between gradient descent and Gauss-Newton, exponentiation by κ as in equation 5 significantly affects the scales inherent in the Jacobian. This is highly important to appropriately influence physics and neural network scales.

To determine κ, we recall our goal to perform update steps which are optimal in both $\boldsymbol{\theta}$- and $\boldsymbol{y}$-space. However, since any update $\Delta\boldsymbol{\theta}$ and its corresponding effect on the solver output $\Delta\boldsymbol{y}$ are connected by the inherent scales encoded in the Jacobian, no single κ exists that normalizes both at the same time. Instead, we distribute the burden equally between network and physics by choosing $\kappa = -1/2$. From a geometric viewpoint, the resulting update can be regarded as a steepest descent step when the norm to measure distance is chosen accordingly. This alternative way to approach our method is explained in the appendix (A.2) and summarized in table 1.

For batch size b and learning rate η, we define the following update step for our method by stacking network-solver Jacobians $\frac{\partial \boldsymbol{y}_i}{\partial \boldsymbol{\theta}}\big|_{\boldsymbol{x}_i}$ and loss gradients $\frac{\partial L}{\partial \boldsymbol{y}_i}\big|_{\boldsymbol{x}_i,\hat{\boldsymbol{y}}_i}$ of different data points $(\boldsymbol{x}_i, \hat{\boldsymbol{y}}_i)$:

Table 1: Optimization algorithms viewed as steepest descent algorithm w.r.t. the given L2-norms.

Optimization Method	performs Steepest Descent:	Norm ($\boldsymbol{\theta}$-space)		Norm ($\boldsymbol{y}$-space)
Gradient Descent	in Parameter Space	$\|\|\cdot\|\|_{\boldsymbol{\theta}}$	$=$	$\|\|\cdot\|\|_{J^{-1}\boldsymbol{y}}$
Gauss-Newton	in Physics Space	$\|\|\cdot\|\|_{J\boldsymbol{\theta}}$	$=$	$\|\|\cdot\|\|_{\boldsymbol{y}}$
Ours	in Intermediate Space	$\|\|\cdot\|\|_{J^{3/4}\boldsymbol{\theta}}$	$=$	$\|\|\cdot\|\|_{J^{-1/4}\boldsymbol{y}}$

$$\Delta\boldsymbol{\theta}_{\mathrm{HIG}} = -\eta \cdot \begin{pmatrix} \frac{\partial \boldsymbol{y}_1}{\partial \boldsymbol{\theta}}\Big|_{\boldsymbol{x}_1} \\ \frac{\partial \boldsymbol{y}_2}{\partial \boldsymbol{\theta}}\Big|_{\boldsymbol{x}_2} \\ \vdots \\ \frac{\partial \boldsymbol{y}_b}{\partial \boldsymbol{\theta}}\Big|_{\boldsymbol{x}_b} \end{pmatrix}^{-1/2} \cdot \begin{pmatrix} \frac{\partial L}{\partial \boldsymbol{y}_1}\Big|_{\boldsymbol{x}_1,\hat{\boldsymbol{y}}_1}^{\top} \\ \frac{\partial L}{\partial \boldsymbol{y}_2}\Big|_{\boldsymbol{x}_2,\hat{\boldsymbol{y}}_2}^{\top} \\ \vdots \\ \frac{\partial L}{\partial \boldsymbol{y}_b}\Big|_{\boldsymbol{x}_b,\hat{\boldsymbol{y}}_b}^{\top} \end{pmatrix} \tag{7}$$

Besides batch size b and learning rate η, we specify a truncation parameter τ as an additional hyperparameter enabling us to suppress numerical noise during the half-inversion process in equation 6. As with the computation of the pseudoinverse via SVD, we set the result of the $-\frac{1}{2}$-exponentiation of every singular value smaller than τ to 0.

The use of a *half-inversion* – instead of a full inversion – helps to prevent exploding updates of network parameters while still guaranteeing substantial progress in directions of low curvature. With the procedure outlined above, we arrived at a balanced method that combines the advantages of optimization methods from deep learning and physics. As our method uses half-inverse Jacobians multiplied with gradients we refer to them in short as *half-inverse gradients* (HIGs).

Half-inverse Gradients in the Toy Example. With the definition of HIGs, we optimize the toy example introduced in section 2.1. The results in figure 1 show that for $\gamma = 1$, HIGs minimize the objective as well as Adam and Gauss-Newton's method. More interestingly, HIGs achieve a better result than the other two methods for $\gamma = 0.01$. On the one hand, the physics trajectory (figure 1b, right) highlights that HIGs can process information along the small-scale component y^2 well and successfully progress along this direction. On the other hand, by checking neuron saturation (figure 1b, middle), we see that HIGs – in contrast to Gauss Newton – avoid oversaturating neurons.

2.3 Practical Considerations

Computational Cost. A HIG update step consists of constructing the stacked Jacobian and computing the half-inversion. The first step can be efficiently parallelized on modern GPUs, and therefore induces a runtime cost comparable to regular backpropagation at the expense of higher memory requirements. In situations where the computational cost of the HIG step is dominated by the half-inversion, memory requirements can be further reduced by parallelizing the Jacobian computation only partially. At the heart of the half-inversion lies a divide and conquer algorithm for the singular value decomposition (Trefethen & Bau, 1997). Hence, the cost of a HIG step scales as $\mathcal{O}(|\boldsymbol{\theta}|\cdot b^2\cdot|\boldsymbol{y}|^2)$, i.e. is linear in the number of network parameters $|\boldsymbol{\theta}|$, and quadratic in the batch size b and the dimension of the physical state $|\boldsymbol{y}|$. Concrete numbers for memory requirements and duration of a HIG step are listed in the appendix.

Hyperparameters. Our method depends on several hyperparameters. First, we need a suitable choice of the learning rate. The normalizing effects of HIGs allow for larger learning rates than commonly used gradient descent variants. We are able to use $\eta = 1$ for many of our experiments. Second, the batch size b affects the number of data points included in the half-inversion process. It should be noted that the way the feedback of individual data points is processed is fundamentally different from the standard gradient optimizers: Instead of the averaging procedure of individual gradients of a mini batch, our approach constructs an update that is optimal for the complete batch. Consequently, the quality of updates increases with higher batch size. However, overly large batch sizes can cause the Jacobian to become increasingly ill-conditioned and destabilize the learning progress. In appendix C, we discuss the remaining parameters τ and κ with several ablation experiments to illustrate their effects in detail.

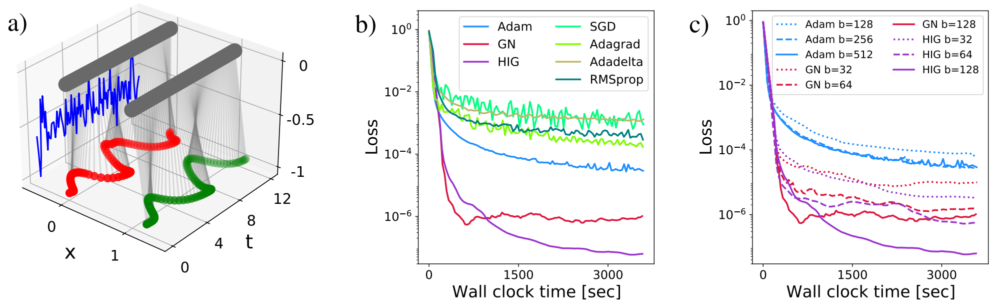


Figure 2: Nonlinear oscillator system: a) Time evolution controlled by a HIG-trained neural network. Its inferred output is shown in blue in the background. b) Loss curves for different optimization methods. c) Loss curves for Adam, GN, and HIG with different batch sizes b.

3 Experiments

We evaluate our method on three physical systems: controlling nonlinear oscillators, the Poisson problem, and the quantum dipole problem. Details of the numerical setups are given in the appendix along with results for a broad range of hyperparameters. For a fair comparison, we show results with the best set of hyperparameters for each of the methods below and plot the loss against wall clock time measured in seconds. All learning curves are recorded on a previously unseen data set.

3.1 Control of Nonlinear Oscillators

First, we consider a control task for a system of coupled oscillators with a nonlinear interaction term. This system is of practical importance in many areas of physics, such as solid state physics (Ibach & Lüth, 2003). Its equations of motions are governed by the Hamiltonian

$$\mathcal{H}(x_i, p_i, t) = \sum_i \left(\frac{x_i^2}{2} + \frac{p_i^2}{2} + \alpha \cdot (x_i - x_{i+1})^4 + u(t) \cdot x_i \cdot c_i \right), \tag{8}$$

where x_i and p_i denote the Hamiltonian conjugate variables of oscillator i, α the interaction strength, and the vector c specifies how to scalar-valued control function $u(t)$ is applied. In our setup, we train a neural network to learn the control signal $u(t)$ that transforms a given initial state into a given target state with 96 time steps integrated by a 4th order Runge-Kutta scheme. We use a dense neural network with three hidden layers totalling 2956 trainable parameters and ReLU activations. The Mean-Squared-Error loss is used to quantify differences between predicted and target state. A visualization of this control task is shown in figure 2a.

Optimizer comparison. The goal of our first experiments is to give a broad comparison of the proposed HIGs with commonly used optimizers. This includes stochastic gradient descent (SGD), Adagrad (Duchi et al., 2011), Adadelta (Zeiler, 2012), RMSprop (Hinton et al., 2012), Adam (Kingma & Ba, 2015), and Gauss-Newton (GN) applied to mini batches. The results are shown in figure 2b where all curves show the best runs for each optimizer with suitable hyperparameters independently selected, as explained in the appendix. We find that the state-of-the-art optimizers stagnate early, with Adam achieving the best result with a final loss value of 10^{-4}. In comparison, our method and GN converge faster, exceeding Adam's accuracy after about three minutes. While GN exhibits stability problems, the best stable run from our hyperparameter search reaches a loss value of 10^{-6}. HIGs, on the other hand, yield the best result with a loss value of 10^{-7}. These results clearly show the potential of our method to process different scales of the physics solver more accurately and robustly. They also make clear that the poor result of the widely-used network optimizers cannot be attributed to simple numerical issues as HIG converges to better levels of accuracy with an otherwise identical setup.

Role of the batch size. We conduct multiple experiments using different values for the batch size b as a central parameter of our method. The results are shown in figure 2c. We observe that for

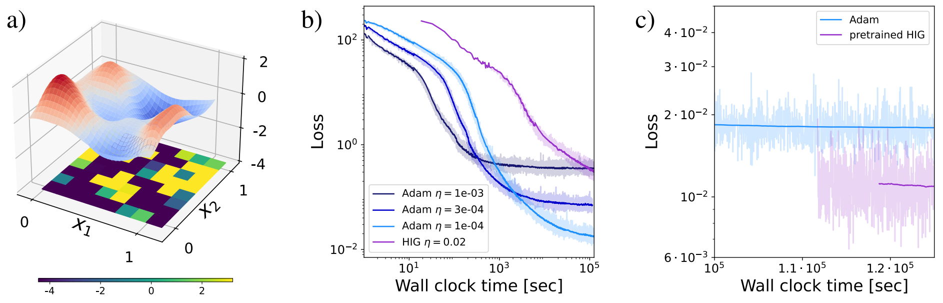


Figure 3: Poisson problem: a) Example of a source distribution ρ (bottom) and inferred potential field (top). b) Loss curves of Adam and HIG training for different learning rates η. c) Loss curves of Adam ($\eta = 0.0001$), and HIG ($\eta = 0.02$) pretrained with Adam.

Adam, all runs converge about equally quickly while HIGs and GN show improvements from larger batch sizes. This illustrates an important difference between Adam and HIG: Adam uses an average of gradients of data points in the mini batch, which approaches its expectation for large b. Further increasing the batch size has little influence on the updates. In contrast, our method includes the individual data point gradients without averaging. As shown in equation 7, we construct updates that are optimized for the whole batch by solving a linear system. This gives our method the ability to hit target states very accurately with increasing batch size. To provide further insights into the workings of HIGs, we focus on detailed comparisons with Adam as the most popular gradient descent variant.

3.2 Poisson Problem

Next we consider Poisson's equation to illustrate advantages and current limitations of HIGs. Poisson problems play an important role in electrostatics, Newtonian gravity, and fluid dynamics (Ames, 2014). For a source distribution $\rho(x)$, the goal is to find the corresponding potential field $\phi(x)$ fulfilling the following differential equation:

$$\Delta\phi = \rho \tag{9}$$

Classically, Poisson problems are solved by solving the corresponding system of linear equations on the chosen grid resolution. Instead, we train a dense neural network with three hidden layers and 41408 trainable parameters to solve the Poisson problem for a given right hand side ρ. We consider a two-dimensional system with a spatial discretization of 8×8 degrees of freedom. An example distribution and solution for the potential field are shown in figure 3a.

Convergence and Runtime. Figure 3b shows learning curves for different learning rates when training the network with Adam and HIGs. As we consider a two-dimensional system, this optimization task is challenging for both methods and requires longer training runs. We find that both Adam and HIGs are able to minimize the loss by up to three orders of magnitude. The performance of Adam varies, and its two runs with larger η quickly slow down. In terms of absolute convergence per time, the Adam curve with the smallest η shows advantages in this scenario. However, choosing a log-scale for the time axis reveals that both methods have not fully converged. In particular, while the Adam curve begins to flatten at the end, the slope of the HIG curve remains constant and decreases with a steeper slope than Adam. The performance of Adam can be explained by two reasons. First, the time to compute a single Adam update is much smaller than for HIGs, which requires the SVD solve from equation 6. While these could potentially be sped up with appropriate methods (Foster et al., 2011; Allen-Zhu & Li, 2016), the absolute convergence per iteration, shown in the appendix in figure 7, shows how much each HIG update improves over Adam. Second, compared to the other examples, the Poisson problem is relatively simple, requiring only a single matrix inversion. This represents a level of difficulty which Adam is still able to handle relatively well.

HIGs with Adam Pretraining. To further investigate the potential of HIGs, we repeat the training, this time using the best Adam model from figure 3b for network initialization. While Adam progresses slowly, HIGs are able to quickly improve the state of the neural network, resulting in

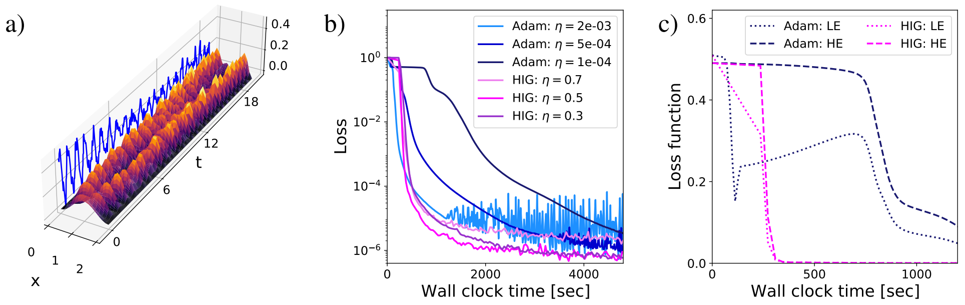


Figure 4: Quantum dipole: a) A transition of two quantum states in terms of probability amplitude $|\Psi(t,x)|^2$, controlled by a HIG-trained neural network. Its inferred output is shown in blue in the background. b) Loss curves with Adam and HIGs for different η. c) Low-energy (LE) and high-energy (HE) loss with Adam ($\eta = 0.0001$) and HIG ($\eta = 0.5$).

a significant drop of the loss values, followed by a faster descent than Adam. Interestingly, this experiment indicates that the HIG updates are able to improve aspects of the solution which Adam is agnostic to. Despite outlining the potential gains from faster SVD calculations, this example also highlights the quality of the HIG updates for simpler PDEs.

3.3 Quantum Dipole

As a final example, we target the quantum dipole problem, a standard control task formulated on the Schrödinger equation and highly relevant in quantum physics (Von Neumann, 2018). Given an initial and a target state, we train a neural network to compute the temporal transition function $u(t)$ in an infinite-well potential V according the evolution equation of the physical state Ψ:

$$i\partial_t \Psi = \big(-\Delta + V + u(t)\cdot \hat{x} \big)\Psi \tag{10}$$

We employ a modified Crank-Nicolson scheme (Winckel et al., 2009) for the discretization of spatial and temporal derivatives. Thus, each training iteration consists of multiple implicit time integration steps – 384 in our setup – for the forward as well as the backward pass of each mini-batch. The control task consists of inferring a signal that converts the ground state to a given randomized linear combination of the first and the second excited state. We use a dense neural network with three hidden layers, 9484 trainable parameters and $\tanh$ activations. Similarity in quantum theories is quantified with inner products; therefore, our loss function is given by $L(\Psi_a, \Psi_b) = 1 - |\langle \Psi_a, \Psi_b \rangle|^2$. A visualization of this control task is shown in figure 4a.

Speed and Accuracy. We observe that HIGs minimize the loss faster and reach a better final level of accuracy than Adam (figure 4b). While the Adam run with the largest learning rate drops faster initially, its final performance is worse than all other runs. In this example, the difference between the final loss values is not as large as for the previous experiments. This is due to the numerical accuracy achievable by a pure physics optimization, which for our choice of parameters is around 10^{-6}. Hence, we can not expect to improve beyond this lower bound for derived learning problems. Our results indicate that the partial inversion of the Jacobian successfully leads to the observed improvements in convergence speed and accuracy.

Low and High Energy Components. The quantum control problem also serves to highlight the weakness of gradient-based optimizers in appropriately processing different scales of the solutions. In the initial training stage, the Adam curves stagnate at a loss value of 0.5. This is most pronounced for $\eta = 10^{-4}$ in dark blue. To explain this effect, we recall that our learning objective targets transitions to combinations of the 1st and 2nd excited quantum states, and both states appear on average with equal weight in the training data. Transitions to the energetically higher states are more difficult and connected to smaller scales in the physics solver, causing Adam to fit the lower-energetic component first. In contrast, our method is constructed to process small scales in the Jacobian via the half-inversion more efficiently. As a consequence, the loss curves decrease faster below 0.5. We support this explanation by explicitly plotting separate loss curves in figure 4c quantifying how well the low and high energy component of the target state was learned. Not only does Adam prefer to minimize the low-energy loss, it also increases the same loss again before it is able to minimize the high-energy loss. In contrast, we observe that HIGs minimize both losses uniformly. This is another indication for the correctness of the theory outlined above of an more even processing of different scales in joint physics and neural network objectives through our method.

4 Related Work

Optimization algorithms. Optimization on continuous spaces is a huge field that offers a vast range of techniques (Ye et al., 2019). Famous examples are gradient descent (Curry, 1944), Gauss-Newton's method (Gill & Murray, 1978), Conjugate Gradient (Hestenes et al., 1952), or the limited-memory BFGS algorithm (Liu & Nocedal, 1989). In deep learning, the preferred methods instead rely on first order information in the form of the gradient, such as SGD (Bottou, 2010) and RMSProp (Hinton et al., 2012). Several methods approximate the diagonal of the Hessian to improve scaling behavior, such as Adagrad (Duchi et al., 2011), Adadelta (Zeiler, 2012), and most prominently, Adam (Kingma & Ba, 2015). However, due to neglecting inter-dependencies of parameters, these methods are limited in their capabilities to handle physical learning objectives. Despite the computational cost, higher-order methods have also been studied in deep learning (Pascanu & Bengio, 2013) . Practical methods have been suggested by using a Kroenecker-factorization of the Fisher matrix (Martens & Grosse, 2015), iterative linear solvers (Martens, 2010), or by recursive approximations of the Hessian (Botev et al., 2017). To the best of our knowledge, the only other technique specifically targeting optimization of neural networks with physics objectives is the inversion approach from Holl et al. (2021). However, their updates are based on inverse physics solvers, while we address the problem by treating network and solver as an entity and half-inverting its Jacobian. Thus, we work on the level of linear approximations while updates based on physics inversion are able to harness higher-order information provided that an higher-order inverse solver exists. Additionally, they compute their update by averaging gradients over different data points, in line with typical gradient-based neural network optimizers. HIGs instead process the feedback of different data points via collective inversion.

Incorporating physics. Many works involve differentiable formulations of physical models, e.g., for robotics (Toussaint et al., 2018), to enable deep architectures (Chen et al., 2018), as a means for scene understanding (Battaglia et al., 2013; Santoro et al., 2017), or the control of rigid body environments de Avila Belbute-Peres et al. (2018). Additional works have shown the advantages of physical loss formulations (Greydanus et al., 2019; Cranmer et al., 2020). Differentiable simulation methods were proposed for a variety of phenomena, e.g. for fluids (Schenck & Fox, 2018), PDE discretizations (Bar-Sinai et al., 2019), molecular dynamics (Wang et al., 2020), reducing numerical errors (Um et al., 2020), and cloth (Liang et al., 2019; Rasheed et al., 2020). It is worth noting that none of these works question the use of standard deep learning optimizers, such as Adam. In addition, by now a variety of specialized software frameworks are available to realize efficient implementations (Hu et al., 2020; Schoenholz & Cubuk, 2019; Holl et al., 2020).

5 Discussion and Outlook

We have considered optimization problems of neural networks in combination with physical solvers and questioned the current practice of using the standard gradient-based network optimizers for training. Derived from an analysis of smooth transitions between gradient descent and Gauss-Newton's method, our novel method learns physics modes more efficiently without overly straining the network through large weight updates, leading to a faster and more accurate minimization of the learning objective. This was demonstrated with a range of experiments.

We believe that our work provides a starting point for further research into improved learning methods for physical problems. Highly interesting avenues for future work are efficient methods for the half-inversion of the Jacobian matrix, or applying HIGs to physical systems exhibiting chaotic behavior or to more sophisticated training setups (Battaglia et al., 2013; Ummenhofer et al., 2020; Pfaff et al., 2020).

ACKNOWLEDGEMENTS

This work was supported by the ERC Consolidator Grant CoG-2019-863850 *SpaTe*, and by the DFG SFB-Transregio 109 *DGD*. We would also like to express our gratitude to the reviewers and the area chair for their helpful feedback.

REPRODUCIBILITY STATEMENT

Our code for the experiments presented in this paper is publicly available at `https://github.com/tum-pbs/half-inverse-gradients`. Additionally, the chosen hyperparameters are listed in the appendix along with the hardware used to run our simulations.

REFERENCES

Peter Adby. *Introduction to optimization methods*. Springer Science and Business Media, 2013.

Zeyuan Allen-Zhu and Yuanzhi Li. Lazysvd: Even faster svd decomposition yet without agonizing pain. *Advances in Neural Information Processing Systems*, 29:974–982, 2016.

William Ames. *Numerical methods for partial differential equations*. Academic press, 2014.

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):15344–15349, 2019.

Peter Battaglia, Razvan Pascanu, Matthew Lai, Danilo Jimenez Rezende, et al. Interaction networks for learning about objects, relations and physics. In *Advances in Neural Information Processing Systems*, pp. 4502–4510, 2016.

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.

Aleksandar Botev, Hippolyt Ritter, and David Barber. Practical gauss-newton optimisation for deep learning. In *International Conference on Machine Learning*, pp. 557–565. PMLR, 2017.

Léon Bottou. Large-scale machine learning with stochastic gradient descent. In *Proceedings of COMPSTAT'2010*, pp. 177–186. Springer, 2010.

Tian Qi Chen, Yulia Rubanova, Jesse Bettencourt, and David K Duvenaud. Neural ordinary differential equations. In *Advances in Neural Information Processing Systems*, pp. 6571–6583, 2018.

Miles Cranmer, Sam Greydanus, Stephan Hoyer, Peter Battaglia, David Spergel, and Shirley Ho. Lagrangian neural networks. *arXiv:2003.04630*, 2020.

Haskell B Curry. The method of steepest descent for non-linear minimization problems. *Quarterly of Applied Mathematics*, 2(3):258–261, 1944.

Filipe de Avila Belbute-Peres, Kevin Smith, Kelsey Allen, Josh Tenenbaum, and J Zico Kolter. End-to-end differentiable physics for learning and control. In *Advances in Neural Information Processing Systems*, 2018.

John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of machine learning research*, 12(7), 2011.

Blake Foster, Sridhar Mahadevan, and Rui Wang. A gpu-based approximate svd algorithm. In *International Conference on Parallel Processing and Applied Mathematics*, pp. 569–578. Springer, 2011.

Philip E Gill and Walter Murray. Algorithms for the solution of the nonlinear least-squares problem. *SIAM Journal on Numerical Analysis*, 15(5):977–992, 1978.

Ian Goodfellow, Yoshua Bengio, Aaron Courville, and Yoshua Bengio. *Deep learning*, volume 1. MIT Press, 2016.

Samuel Greydanus, Misko Dzamba, and Jason Yosinski. Hamiltonian neural networks. In *Advances in Neural Information Processing Systems*, pp. 15353–15363, 2019.

Magnus R Hestenes, Eduard Stiefel, et al. Methods of conjugate gradients for solving linear systems. *Journal of research of the National Bureau of Standards*, 49(6):409–436, 1952.

Geoffrey Hinton, Nitish Srivastava, and Kevin Swersky. Lecture 6a, overview of mini-batch gradient descent. *Neural networks for machine learning*, 14(8):2, 2012.

Philipp Holl, Vladlen Koltun, and Nils Thuerey. Learning to control pdes with differentiable physics. In *International Conference on Learning Representations (ICLR)*, 2020.

Philipp Holl, Vladlen Koltun, and Nils Thuerey. Physical gradients for deep learning. *arXiv*, 2109.15048, 2021.

Yuanming Hu, Luke Anderson, Tzu-Mao Li, Qi Sun, Nathan Carr, Jonathan Ragan-Kelley, and Frédo Durand. Difftaichi: Differentiable programming for physical simulation. *International Conference on Learning Representations (ICLR)*, 2020.

H. Ibach and H. Lüth. *Solid-State Physics*. Advanced texts in physics. Springer, 2003. ISBN 9783540438700.

Diederik Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In *International Conference on Learning Representations (ICLR)*, 2015.

Dmitrii Kochkov, Jamie A Smith, Ayya Alieva, Qing Wang, Michael P Brenner, and Stephan Hoyer. Machine learning–accelerated computational fluid dynamics. *Proceedings of the National Academy of Sciences*, 118(21), 2021.

Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems*, 2012.

Peter Lax. *Linear algebra and its applications*, volume 2. Hoboken : Wiley, 2014.

Junbang Liang, Ming Lin, and Vladlen Koltun. Differentiable cloth simulation for inverse problems. In *Advances in Neural Information Processing Systems*, pp. 771–780, 2019.

Dong Liu and Jorge Nocedal. On the limited memory bfgs method for large scale optimization. *Mathematical programming*, 45(1-3):503–528, 1989.

James Martens. Deep learning via hessian-free optimization. In *International conference on machine learning*, pp. 735–742. PMLR, 08 2010.

James Martens and Roger Grosse. Optimizing neural networks with kronecker-factored approximate curvature. In *International conference on machine learning*, pp. 2408–2417. PMLR, 2015.

Jeremy Morton, Antony Jameson, Mykel J Kochenderfer, and Freddie Witherden. Deep dynamical modeling and control of unsteady fluid flows. In *Advances in Neural Information Processing Systems*, 2018.

Razvan Pascanu and Yoshua Bengio. Revisiting natural gradient for deep networks. *arXiv:1301.3584*, 2013.

Tobias Pfaff, Meire Fortunato, Alvaro Sanchez-Gonzalez, and Peter W Battaglia. Learning mesh-based simulation with graph networks. *arXiv preprint:2010.03409*, 2020.

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

Abdullah-Haroon Rasheed, Victor Romero, Florence Bertails-Descoubes, Stefanie Wuhrer, Jean-Sébastien Franco, and Arnaud Lazarus. Learning to measure the static friction coefficient in cloth contact. In *The Conference on Computer Vision and Pattern Recognition*, 2020.

Adam Santoro, David Raposo, David GT Barrett, Mateusz Malinowski, Razvan Pascanu, Peter Battaglia, and Timothy Lillicrap. A simple neural network module for relational reasoning. *arXiv:1706.01427*, 2017.

Connor Schenck and Dieter Fox. Spnets: Differentiable fluid dynamics for deep neural networks. In *Conference on Robot Learning*, pp. 317–335, 2018.

Samuel S Schoenholz and Ekin D Cubuk. Jax, md: End-to-end differentiable, hardware accelerated, molecular dynamics in pure python. *arXiv:1912.04232*, 2019.

David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, et al. Mastering the game of Go without human knowledge. *Nature*, 550(7676), 2017.

Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to sequence learning with neural networks. In *Advances in Neural Information Processing Systems*, pp. 3104–3112, 2014.

Jonathan Tompson, Kristofer Schlachter, Pablo Sprechmann, and Ken Perlin. Accelerating eulerian fluid simulation with convolutional networks. In *Proceedings of Machine Learning Research*, pp. 3424–3433, 2017.

Marc Toussaint, Kelsey Allen, Kevin Smith, and Joshua B Tenenbaum. Differentiable physics and stable modes for tool-use and manipulation planning. In *Robotics: Science and Systems*, 2018.

L. Trefethen and D. Bau. *Numerical Linear Algebra*. EngineeringPro collection. Society for Industrial and Applied Mathematics, 1997. ISBN 9780898714876.

Kiwon Um, Robert Brand, Yun Raymond Fei, Philipp Holl, and Nils Thuerey. Solver-in-the-loop: Learning from differentiable physics to interact with iterative pde-solvers. *Advances in Neural Information Processing Systems*, 2020.

Benjamin Ummenhofer, Lukas Prantl, Nils Thuerey, and Vladlen Koltun. Lagrangian fluid simulation with continuous convolutions. In *International Conference on Learning Representations*, 2020.

John Von Neumann. *Mathematical foundations of quantum mechanics*. Princeton university press, 2018.

Wujie Wang, Simon Axelrod, and Rafael Gómez-Bombarelli. Differentiable molecular simulations for control and learning. *arXiv:2003.00868*, 2020.

Greg von Winckel, Alfio Borzì, and Stefan Volkwein. A globalized newton method for the accurate solution of a dipole quantum control problem. *SIAM Journal on Scientific Computing*, 31(6): 4176–4203, 2009. doi: 10.1137/09074961X.

Tailin Wu and Max Tegmark. Toward an artificial intelligence physicist for unsupervised learning. *Physical Review E*, 100(3):033311, 2019.

Nan Ye, Farbod Roosta-Khorasani, and Tiangang Cui. Optimization methods for inverse problems. In *2017 MATRIX Annals*, pp. 121–140. Springer, 2019.

Matthew Zeiler. Adadelta: an adaptive learning rate method. *arXiv:1212.5701*, 2012.

Olgierd Cecil Zienkiewicz, Robert Leroy Taylor, Perumal Nithiarasu, and JZ Zhu. *The finite element method*, volume 3. McGraw-hill London, 1977.

APPENDIX

A FURTHER DETAILS ON OPTIMIZATION ALGORITHMS

Our work considers optimization algorithms for functions of the form $\boldsymbol{f}(\boldsymbol{x};\boldsymbol{\theta}) = \boldsymbol{y}$ with $\boldsymbol{\theta}, \Delta\boldsymbol{\theta} \in \mathbb{R}^t$, denoting weight vector and weight update vector, respectively, while $\boldsymbol{x} \in \mathbb{R}^n$ and $\boldsymbol{y} \in \mathbb{R}^m$ denote input and output. The learning process solves the minimization problem $\mathrm{argmin}_{\boldsymbol{\theta}} L(\boldsymbol{f}(\boldsymbol{x};\boldsymbol{\theta}), \hat{\boldsymbol{y}})$ via a sequence $\boldsymbol{\theta}^{k+1} = \boldsymbol{\theta}^k + \eta\Delta\boldsymbol{\theta}$. Here, $\hat{\boldsymbol{y}}$ are the reference solutions, and we target losses of the form $L(\boldsymbol{x}, \hat{\boldsymbol{y}}; \boldsymbol{\theta}) = \sum_i l\big(\boldsymbol{f}(\boldsymbol{x}_i;\boldsymbol{\theta}), \hat{\boldsymbol{y}}_i\big)$ with i being an index for multiple data points (i.e., observations). l denotes the L2-loss $\sum_j ||\boldsymbol{x}_j - \hat{\boldsymbol{y}}_j||^2$ with j referencing the entries of a mini batch of size b.

A.1 UPDATE STEP OF THE GAUSS-NEWTON ALGORITHM

Using this notation, the update step of the Gauss-Newton algorithm (Adby, 2013) for $\eta = 1$ is given by:

$$\Delta\boldsymbol{\theta}_{\mathrm{GN}} = -\left(\left(\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{\theta}}\right)^T \cdot \left(\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{\theta}}\right)\right)^{-1} \cdot \left(\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{\theta}}\right)^T \cdot \left(\frac{\partial L}{\partial \boldsymbol{y}}\right)^\top \tag{11}$$

The size of the Jacobian matrix is given by the dimensions of $\boldsymbol{y}$- and $\boldsymbol{\theta}$-space. For a full-rank Jacobian corresponding to non-constrained optimization, the Gauss-Newton update is equivalent to:

$$\Delta\boldsymbol{\theta}_{\mathrm{GN}} = -\left(\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{\theta}}\right)^{-1} \cdot \left(\frac{\partial L}{\partial \boldsymbol{y}}\right)^\top \tag{12}$$

Even in a constrained setting, we can reparametrize the coordinates to obtain an unconstrained optimization problem on the accessible manifold and rewrite $\Delta\boldsymbol{\theta}_{\mathrm{GN}}$ similarly. This shortened form of the update step is given in equation 3, and is the basis for our discussion in the main text.

A.2 GEOMETRIC INTERPRETATION AS STEEPEST DESCENT ALGORITHMS

It is well-known that the negative gradient of a function $L(\boldsymbol{\theta})$ points in the direction of steepest descent leading to the interpretation of gradient descent as a steepest descent algorithm. However, the notion of steepest descent requires defining a measure of distance, which is in this case the usual L2-norm in $\boldsymbol{\theta}$. By using different metrics, we can regard Gauss-Newton and HIG steps as steepest descent algorithms as well.

Gauss-Newton updates. The updates $\Delta\boldsymbol{\theta}_{\mathrm{GN}}$ can be regarded as gradient descent in $\boldsymbol{y}$ up to first order in the update step. This can be seen with a simple equation by considering how these updates change $\boldsymbol{y}$.

$$\Delta\boldsymbol{y} = \left(\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{\theta}}\right) \cdot \Delta\boldsymbol{\theta}_{\mathrm{GN}} + o(\Delta\boldsymbol{\theta}_{\mathrm{GN}}) = -\left(\frac{\partial L}{\partial \boldsymbol{y}}\right)^\top + o(\Delta\boldsymbol{\theta}_{\mathrm{GN}}) \tag{13}$$

In figure 1 of the main paper, this property is visible in the physics trajectories for the well-conditioned case, where $L(\boldsymbol{y})$ is a uniform L2-loss and hence, gradient descent in $\boldsymbol{y}$ produces a straight line to the target point. The Gauss-Newton curve first shows several steps in varying directions as the higher-order terms from the neural network cannot be neglected yet. However, after this initial phase the curve exhibits the expected linear motion.

The behavior of GN to perform steepest descent on the $\boldsymbol{y}$-manifold stands in contrast to gradient descent methods, which instead perform steepest descent on the $\boldsymbol{\theta}$-manifold. This geometric view is the basis for an alternative way to derive our method that is presented below.

HIG updates. HIG updates can be regarded as a steepest descent algorithm, again up to first order in the update step, when measuring distances of $\boldsymbol{\theta}$-vectors with the following semi-norm:

$$||\boldsymbol{\theta}||_{HIG} := ||J^{3/4}\boldsymbol{\theta}|| \tag{14}$$

Here $||\cdot||$ denotes the usual L2-norm and $J = \frac{\partial \boldsymbol{y}}{\partial \boldsymbol{\theta}}$ the Jacobian of network and solver. The exponentiation is performed as explained in the main text, with $J = U\Lambda V^\top$ being the SVD, and $J^{3/4}$ given by $V\Lambda^{3/4}U^\top$. Additionally, we will use the natural map between dual vector and vector $\langle\cdot,\cdot\rangle$ and the loss gradient $g = \frac{\partial L}{\partial \boldsymbol{y}}$.

To prove the claim above, we expand the loss around an arbitrary starting point $\boldsymbol{\theta}_0$:

$$L(y(\boldsymbol{\theta}_0 + \Delta\boldsymbol{\theta})) = L(y(\boldsymbol{\theta}_0)) + \langle g \cdot J, \Delta\boldsymbol{\theta}\rangle + o(\Delta\boldsymbol{\theta}) \tag{15}$$

The first term on the right-hand side is constant and the third term is neglected according to the assumptions of the claim. Hence, we investigate for which fixed-length $\Delta\boldsymbol{\theta}$ the second term decreases the most:

$$\begin{aligned}
\underset{||\boldsymbol{\Delta\theta}||_{HIG}=const.}{\arg\min} \big(\langle g\cdot J, \Delta\boldsymbol{\theta}\rangle\big) &= \underset{||\boldsymbol{\theta}||_{HIG}=const.}{\arg\min} \big(\langle g \cdot J^{1/4}, J^{3/4}\Delta\boldsymbol{\theta}\rangle\big) \\
&= \underset{\gamma}{\arg\min} \big(\cos\gamma \cdot \underbrace{||g\cdot J^{1/4}||}_{const.} \cdot \underbrace{||J^{3/4}\Delta\boldsymbol{\theta}||}_{=const.} \big) \\
&= \underset{\gamma}{\arg\min} \big(\cos\gamma \big)
\end{aligned} \tag{16}$$

In the first step above, we split the Jacobian $J^\top = V\Lambda U^\top = (V\Lambda^{1/4}V^\top)(V\Lambda^{3/4}U^\top) = J^{1/4}J^{3/4}$. γ denotes the angle between $J^{1/4}g^\top$ and $J^{3/4}\Delta\boldsymbol{\theta}$. This expression is minimized for $\gamma = -\pi$, meaning the two vectors have to be antiparallel:

$$J^{3/4}\Delta\boldsymbol{\theta} = -J^{1/4}g^\top \tag{17}$$

This requirement is fulfilled by the HIG update $\Delta\boldsymbol{\theta}_{HIG} = -J^{1/2}g^\top$, and is therefore a steepest descent method, which concludes our proof.

This presents another approach to view HIGs as an interpolation between gradient descent and Gauss-Newton's method. More precisely, gradient descent performs steepest descent in the usual L2-norm in $\boldsymbol{\theta}$-space ($||\boldsymbol{\theta}||$). Considering only terms up to linear order, Gauss-Newton performs steepest descent in the L2-norm in $\boldsymbol{y}$-space ($||J\boldsymbol{\theta}||$). The HIG update ($||J^{3/4}\boldsymbol{\theta}||$) lies between these two methods. The quarter factors in the exponents result from the additional factor of 2 that has to be compensated for when considering L2-norms.

A.3 Stability of Inversions in the Context of Physical Deep Learning.

In the following, we illustrate how the full inversion of GN can lead to instabilities at training time. Interestingly, physical solvers are not the only cause of small singular values in the Jacobian. They can also occur when applying equation 12 to a mini batch to train a neural network and are not caused by numerical issues. Consider the simple case of two data points $(\boldsymbol{x}_1, \hat{\boldsymbol{y}}_1)$ and $(\boldsymbol{x}_2, \hat{\boldsymbol{y}}_2)$ and a one-dimensional output. Let f be the neural network and J the Jacobian, which is in this case the gradient of the network output. Then equation 12 yields:

$$\begin{pmatrix} J_{\boldsymbol{f}}(\boldsymbol{x}_1) \\ J_{\boldsymbol{f}}(\boldsymbol{x}_2) \end{pmatrix} \cdot \Delta\boldsymbol{\theta}_{\text{GN}} = \begin{pmatrix} \boldsymbol{f}(\boldsymbol{x}_1) - \hat{\boldsymbol{y}}_1 \\ \boldsymbol{f}(\boldsymbol{x}_2) - \hat{\boldsymbol{y}}_2 \end{pmatrix} \tag{18}$$

Next, we linearly approximate the second row by using the Hessian H by assuming the function to be learned is $\hat{\boldsymbol{f}}$, i.e. $\hat{\boldsymbol{f}}(\boldsymbol{x}_1) = \boldsymbol{y}_1$ and $\hat{\boldsymbol{f}}(\boldsymbol{x}_2) = \boldsymbol{y}_2$. Neglecting terms beyond the linear approximation, we receive:

$$\begin{pmatrix} J_{\boldsymbol{f}}(\boldsymbol{x}_1) \\ J_{\boldsymbol{f}}(\boldsymbol{x}_1) + H_{\boldsymbol{f}}(\boldsymbol{x}_1) \cdot (\boldsymbol{x}_2 - \boldsymbol{x}_1) \end{pmatrix} \cdot \Delta\boldsymbol{\theta}_{\mathrm{GN}} = \begin{pmatrix} \boldsymbol{f}(\boldsymbol{x}_1) - \boldsymbol{y}_1 \\ \boldsymbol{f}(\boldsymbol{x}_1) - \boldsymbol{y}_1 + (J_{\boldsymbol{f}}(\boldsymbol{x}_1) - J_{\hat{\boldsymbol{f}}}(\boldsymbol{x}_1)) \cdot (\boldsymbol{x}_2 - \boldsymbol{x}_1) \end{pmatrix} \tag{19}$$

Considering the case of two nearby data points, i.e. $\boldsymbol{x}_2 - \boldsymbol{x}_1$ being small, the two row vectors in the stacked Jacobian on the left-hand side are similar, i.e. the angle between them is small. This leads to a small singular value of the stacked Jacobian. In the limit of $\boldsymbol{x}_2 = \boldsymbol{x}_1$ both row vectors are linearly dependant and hence, one singular value becomes zero.

Moreover, even if $\boldsymbol{x}_2$ is not close to $\boldsymbol{x}_1$, small singular values can occur if the batch size increases: for a growing number of row vectors it becomes more and more likely that the Jacobian contains similar or linearly dependent vectors.

After inversion, a small singular value becomes large. This leads to a large update $\Delta\boldsymbol{\theta}_{\mathrm{GN}}$ when the right-hand side of equation 19 overlaps with the corresponding singular vector.

This can easily happen if the linear approximation of the right-hand side is poor, for instance when $\hat{\boldsymbol{f}}$ is a solution to an inverse physics problem. Then $\hat{\boldsymbol{f}}$ can have multiple modes and can, even within a mode, exhibit highly sensitive or even singular behavior.

In turn, applying large updates to the network weights naturally can lead to the oversaturation of neurons, as illustrated above, and diverging training runs in general.

As illustrated in the main paper, these inherent problems of GN are alleviated by the partial inversion of the HIG. It yields a fundamentally different order of scaling via its square-root inversion, which likewise does not guarantee that small singular values lead to overshoots (hence the truncation), but in general strongly stabilizes the training process.

B EXPERIMENTAL DETAILS

In the following, we provide details of the physical simulations used for our experiments in section 3 of the main paper. For the different methods, we use the following abbreviations: half-inverse gradients (HIG), Gauss-Newton's method (GN), and stochastic gradient descent (GD). Learning rates are denoted by η, batch sizes by b, and truncation parameters for HIG and GN by τ. All loss results are given for the average loss over a test set with samples distinct from the training data set.

For each method, we run a hyperparameter search for every experiment, varying the learning rate by several orders of magnitude, and the batch size in factors of two. Unless noted otherwise, the best runs in terms of final test loss were selected and shown in the main text. The following sections contain several examples from the hyperparameter search to illustrate how the different methods react to the changed settings.

Runtime Measurements Runtimes for the non-linear chain and quantum dipole were measured on a machine with Intel Xeon 6240 CPUs and NVIDIA GeForce RTX 2080 Ti GPUs. The Poisson experiments used an Intel Xeon W-2235 CPU with NVIDIA Quadro RTX 8000 GPU. We experimentally verified that these platforms yield an on-par performance for our implementation. As deep learning API we used TensorFlow version 2.5. If not stated otherwise, each experiment retained the default settings.

All runtime graphs in the main paper and appendix contain wall-clock measurements that include all steps of a learning run, such as initialization, in addition to the evaluation time of each epoch. However, the evaluations of the test sets to determine the performance in terms of loss are not included. As optimizers such as Adam typically performs a larger number of update steps including these evaluations would have put these optimizers at an unnecessary disadvantage.

B.1 TOY EXAMPLE (SECTION 2.1)

For the toy example, the target function is given by $\hat{f}(x) = (\sin(6x), \cos(9x))$. We used a dense neural network consisting of one hidden layer with 7 neurons and $\tanh$ activation, and an output layer with 2 neurons and linear activation. For training, we use 1024 data points uniformly sampled

Table 2: Hyperparameters for different optimization algorithms in figure 2b

Method	Adadelta	Adagrad	Adam	GN	HIG	RMSprop	SGD
b	512	512	512	128	128	512	512
η	0.1	0.1	$3 \cdot 10^{-4}$	-	1	10^{-4}	0.1
τ	-	-	-	10^{-6}	10^{-6}	-	-

Table 3: Nonlinear oscillators: memory requirements, update duration and duration of the Jacobian computation for Adam and HIG

Optimizer	Adam	Adam	Adam	HIG	HIG	HIG
Batch size	256	512	1024	32	64	128
Memory (MB)	11.1	22.2	44.5	169	676	2640
Update duration (sec)	0.081	0.081	0.081	0.087	0.097	0.146
Jacobian duration (sec)	0.070	0.070	0.070	0.070	0.070	0.070

from the $[-1, 1]$ interval, and a batch size of 256. For the optimizers, the following hyperparameters were used for both the well-conditioned loss and the ill-conditioned loss: Adam $\eta = 0.3$; GN has no learning rate (equivalent to $\eta = 1$), $\tau = 10^{-4}$; HIG $\eta = 1.0$, $\tau = 10^{-6}$.

B.2 Control of Nonlinear Oscillators (section 3.1)

The Hamiltonian function given in equation 8 leads to the following equations of motions:

$$\ddot{x}_i = -x_i + 4\alpha(x_i - x_{i-1})^3 - 4\alpha(x_i - x_{i+1})^3 - u(t) \cdot c_i \tag{20}$$

The simulations of the nonlinear oscillators were performed for two mass points and a time interval of 12 units with a time step $\Delta t = 0.125$. This results in 96 time steps via 4th order Runge-Kutta per learning iteration. We generated 4096 data points for a control vector $c = (0.0, 3.0)$, and an interaction strength $\alpha = 1.0$ with randomized conjugate variables x and p. The test set consists of 4096 new data points. For the neural network, we set up a fully-connected network with ReLU activations passing inputs through three hidden layers with 20 neurons in each layer before being mapped to a 96 output layer with linear activation.

For the comparison with other optimizers (figure 2b) we performed a broad hyperparameter search for each method, as outlined above, to determine suitable settings. The parameters for Adagrad (Duchi et al., 2011), Adadelta (Zeiler, 2012), Adam (Kingma & Ba, 2015), RMSprop (Hinton et al., 2012), Gauss-Newton (Gill & Murray, 1978), HIGs, and stochastic gradient descent (Curry, 1944) are summarized in table 2. For figure 2c the following hyperparameters were used: $\eta = 3 \cdot 10^{-4}$ for Adam, and $\eta = 1.0$, $\tau = 10^{-6}$ for HIG.

Further Experiments. Figure 5 and figure 6 contain additional runs with different hyperparameters for the method comparison of figure 2b in the main paper. The graphs illustrate that all five method do not change their behavior significantly for the different batch sizes in each plot, but become noticeably unstable for larger learning rates η (plots on the right sides of each section).

Details on the memory footprint and update durations can be found in table 3. Since our simulations were not limited by memory, we used an implementation for the Jacobian computation of HIGs, which scales quadratically in the batch size. Should this become a bottleneck, this scaling could potentially be made linear by exploiting that the Jacobian of the physical solver for multiple data points is blockdiagonal.

B.3 Poisson Problem (section 3.2)

We discretize Poisson's equation on a regular grid for a two-dimensional domain $\Omega = [0, 8] \times [0, 8]$ with a grid spacing of $\Delta x = 1$. Dirichlet boundary conditions of $\phi = 0$ are imposed on all four sides of Ω. The Laplace operator is discretized with a finite difference stencil (Ames, 2014).

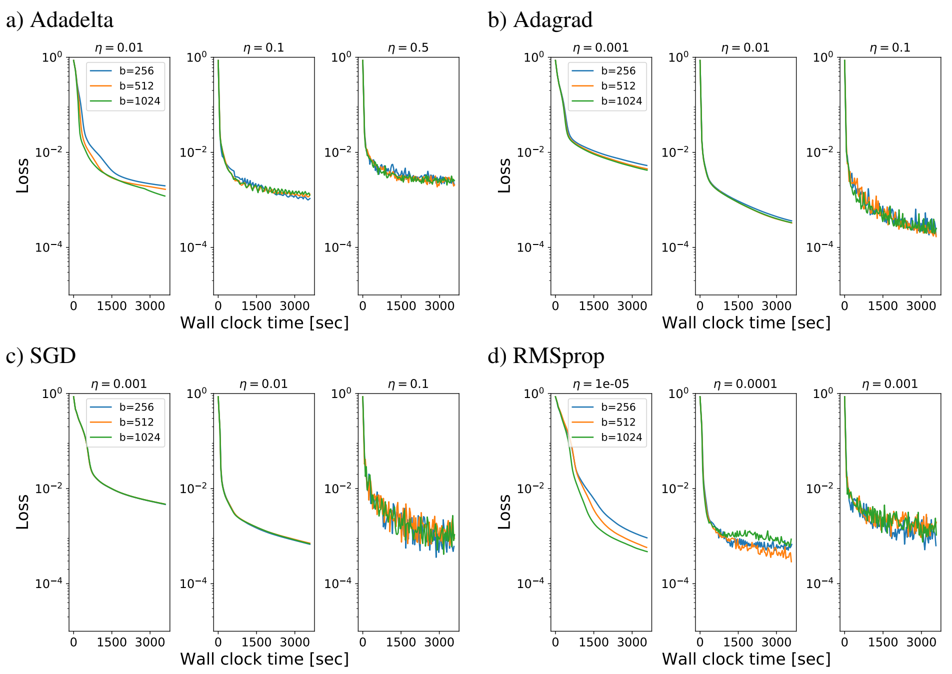


Figure 5: Control of nonlinear oscillators: Additional experiments with (a) Adadelta, (b) Adagrad, (c) stochastic gradient descent , and (d) RMSprop. Each showing different learning rates η and batch sizes b.

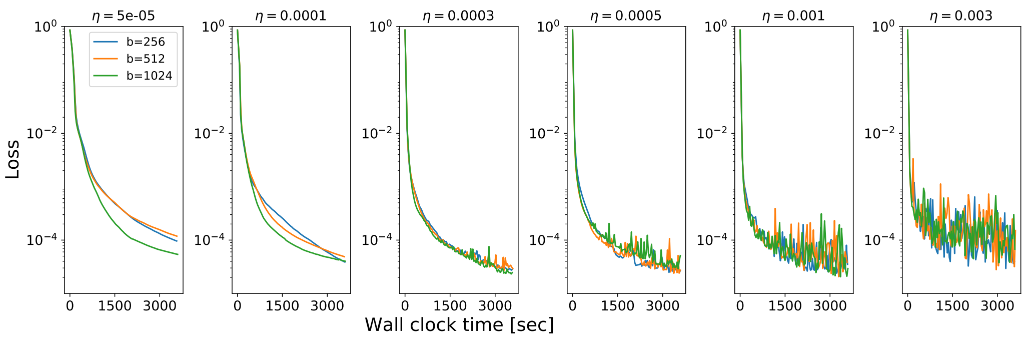


Figure 6: Control of nonlinear oscillators: Additional experiments with Adam for different learning rates η and batch sizes b.

For the neural network, we set up a fully-connected network with $\tanh$ activation functions. The 8x8 inputs pass through three hidden layers with 64, 256 and 64 neurons, respectively, before being mapped to 8x8 in the output layer. For training, source distributions ρ are sampled from random frequencies in Fourier space, and transformed to real space via the inverse Fourier transform. The mean value is normalized to zero. We sample data on-the-fly, resulting in an effectively infinite data set. This makes a separate test set redundant as all training data is previously unseen.

Further Experiments. Figure 7a shows Adam and HIG runs from figure 3b over epochs. The HIG runs converge faster per iteration, which indicates that HIGs perform qualitatively better updates.

Additionally, we use the pretrained HIG run from figure 3c as a starting point for further Adam training. The results are shown in 7b. We observe that the network quickly looses the progress the HIGs have made, and continues with a loss value similar to the orginal Adam run. This again

Table 4: Poisson problem: memory requirements, update duration and duration of the Jacobian computation for Adam and HIG

Optimizer	Adam	HIG
Batch size	64	64
Memory (MB)	1.3	3560
Update duration (sec)	0.011	13.8
Jacobian duration (sec)	0.010	0.0035

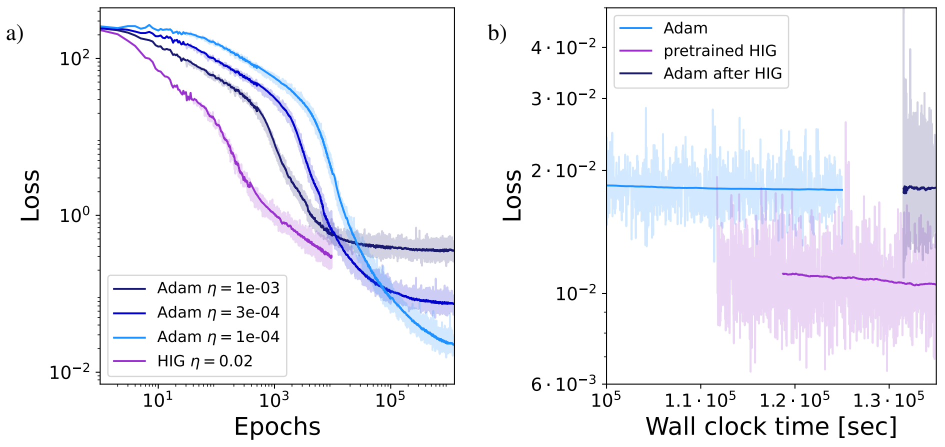


Figure 7: Poisson problem: a) Loss curves for Adam and HIG per epoch for different learning rates, b) Loss curves of Adam (η =1e-04), of HIG ($\eta = 0.02$) pretrained with Adam, and of Adam (η =1e-04) pretrained with the HIGs.

supports our intuition that Adam, in contrast to HIGs, cannot harness the full potential of the physics solver.

Details on the memory footprint and update durations can be found in table 4

B.4 Quantum Dipole (section 3.3)

For the quantum dipole problem, we discretize the Schrödinger equation on a spatial domain $\Omega = [0, 2]$ with a spacing of $\Delta x = 0.133$ resulting in 16 discretization points. We simulate up to a time of 19.2 with a time step of $\Delta t = 0.05$, which yields 384 time steps. Spatial and temporal discretization use a modified Crank-Nicolson scheme (Winckel et al., 2009) which is tailored to quantum simulations. The training data set consists of 1024 randomized superpositions of the first and second excited state, while the test set contains a new set of 1024 randomized superpositions. For the neural network, we set up a fully-connected network with $\tanh$ activations passing the inputs through three hidden layers with 20 neurons in each layer before being mapped to a 384 neuron output layer with linear activation. Overall, the network contains 9484 trainable parameters.

Experimental details. For the training runs in figure 4b, Adam used $b = 256$, while for HIG $b = 16$, and $\tau = 10^{-5}$ were used. For the training runs in figure 4c, Adam used $b = 256$, $\eta = 0.0001$, while HIGs used $b = 16$, $\tau = 10^{-5}$, and $\eta = 0.5$. Details on the memory footprint and update durations can be found in table 5

Figure 8 and figure 9 show the performance of both methods for a broader range of τ settings for HIGs, and η for Adam. For Adam, a trade-off between slow convergence and oscillating updates exists. The HIGs yield high accuracy in training across a wide range of values for τ, ranging from 10^{-5} to 10^{-3}. This supports the argumentation in the main text that the truncation is not

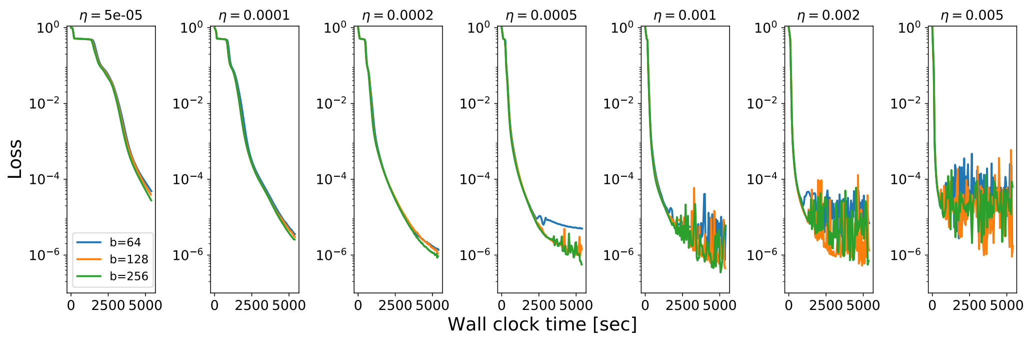


Figure 8: Quantum dipole: Additional experiments with Adam for different learning rates η and batch sizes b.

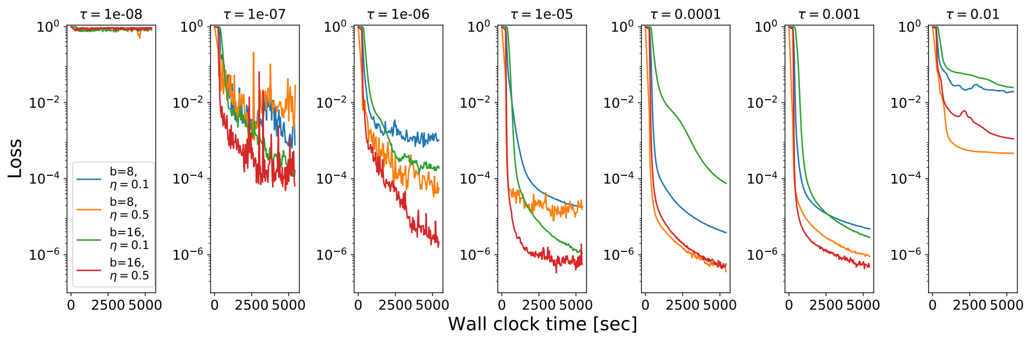


Figure 9: Quantum dipole: Additional experiments with HIGs for different learning rates η, batch sizes b, and truncation parameters τ

overly critical for HIGs. As long as numerical noise is suppressed with $\tau > 10^{-6}$, and the actual information about scaling of network parameters and physical variables is not cut off. The latter case is visible for an overly large $\tau = 0.01$ in the last graph on the right.

Note that many graphs in figure 9 contain a small plateau at the start of each training run. These regions with relatively small progress per wall clock time are caused by the initialization overhead of the underlying deep learning framework (TensorFlow in our case). As all graphs measure wall clock time, we include the initialization overhead of TensorFlow, which causes a noticeable slow down of the first iteration. Hence, the relatively slow convergence of the very first steps in figure 9 are not caused by conceptual issues with the HIGs themselves. Rather, they are a result of the software frameworks and could, e.g., be alleviated with a pre-compilation of the training graphs. In contrast, the initial convergence plateaus of Adam with smaller η in Figure 8 are of a fundamentally different nature: they are caused by an inherent problem of non-inverting optimizers: their inability to appropriately handle the combination of large and small scale components in the physics of the quantum dipole setup (as outlined in section 3.3).

Table 5: Quantum dipole: memory requirements, update duration and duration of the Jacobian computation for Adam and HIG

Optimizer	Adam	Adam	Adam	HIG	HIG
Batch size	256	512	1024	8	16
Memory (MB)	460	947	2007	1064	5039
Update duration (sec)	0.40	0.50	1.33	0.42	0.60
Jacobian duration (sec)	0.39	0.49	1.32	0.40	0.53

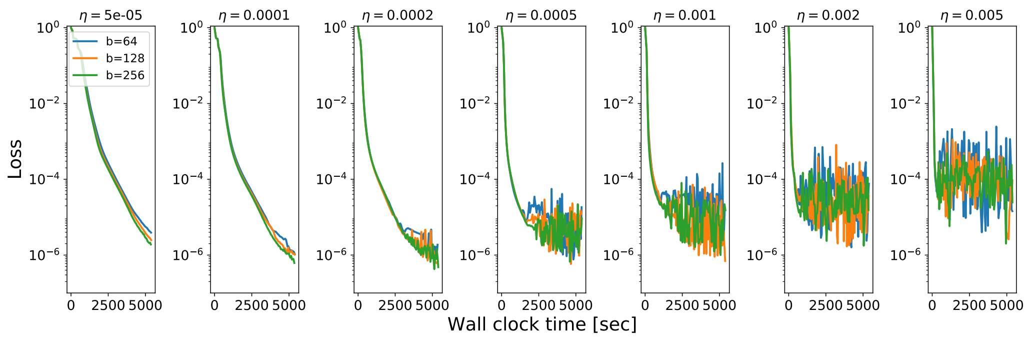


Figure 10: Quantum dipole with Convolutional Neural Network: Experiments with Adam for different learning rates η and batch sizes b.

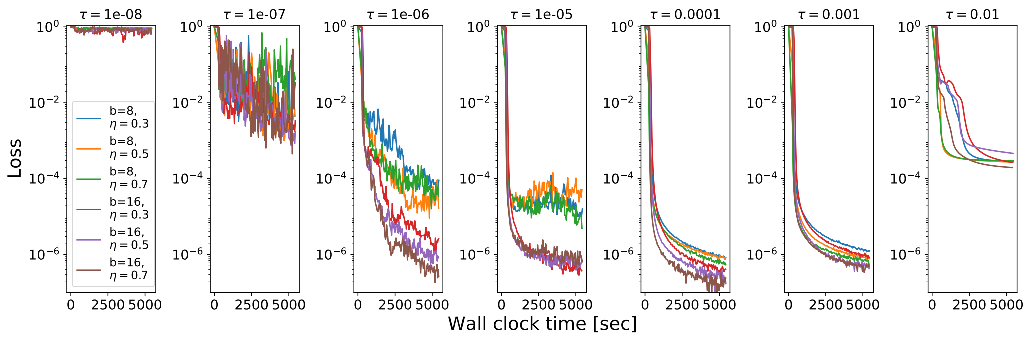


Figure 11: Quantum dipole with Convolutional Neural Network: Experiments with HIGs for different learning rates η, batch sizes b, and truncation parameters τ

Loss Functions. While training is evaluated in terms of the regular inner product as loss function: $L(\Psi_a, \Psi_b) = 1 - |\langle\Psi_a, \Psi_b\rangle|^2$, we use the following modified losses to evaluate low- and high-energy states for figure 4c. Let Ψ_1 be the first excited state, then we define the low-energy loss as:

$$L(\Psi_a, \Psi_b) = (|\langle\Psi_a, \Psi_1\rangle| - |\langle\Psi_1, \Psi_b\rangle|)^2$$

Correspondingly, we define the high-energy loss with the second excited state Ψ_2:

$$L(\Psi_a, \Psi_b) = (|\langle\Psi_a, \Psi_2\rangle| - |\langle\Psi_2, \Psi_b\rangle|)^2$$

Additional Experiments with a Convolutional Neural Network. Our method is agnostic to specific network architectures. To illustrate this, we conduct additional experiments with a convolutional neural network. The setup is the same as before, only the fully-connected neural network is replaced by a network with 6 hidden convolutional layers each with kernel size 3, 20 features and $\tanh$ activation, followed by an 384 neuron dense output layer with linear activation giving the network a total of 21984 trainable parameters.

The results of these experiments are plotted in figure 10 and 11. We find that HIGs behave in line with the fully-connected network case (figure 9). There exists a range τ-values from around 10^{-5} to 10^{-3} for which stable training is possible. Regarding optimization with Adam, we likewise observe a faster and more accurate minimization of the loss function for the best HIG run ($\eta = 0.7$, $b = 16$, $\tau = 10^{-4}$) compared to the best Adam run ($\eta = 0.0002$, $b = 256$).

C ABLATION STUDY

In this last section, we investigate how the HIG-hyperparameters affect the outcome. This includes ablation experiments with respect to κ and τ defined in section 2.2. We use the nonlinear oscillator example as the basis for these comparisons and consider the following HIG update step:

$$\Delta\boldsymbol{\theta}(\eta,\beta,\kappa) = -\eta \cdot \left(\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{\theta}}\right)^{<\beta,\kappa>} \cdot \left(\frac{\partial L}{\partial \boldsymbol{y}}\right)^{\top} \tag{21}$$

Here, the exponent $<\beta,\kappa>$ of the Jacobian denotes the following procedure defined with the aid of the singular value decomposition $J = U\Lambda V^{\top}$ as:

$$J^{<\beta,\kappa>} := \max\{\text{diag}(\Lambda)\}^{\beta} \cdot V\Lambda^{\kappa}U^{\top}, \tag{22}$$

Compared to the HIG update 5 in the main text, update 21 has an additional scalar prefactor with an parameter β resulting from earlier experiments with our method. Setting $\beta = -1-\kappa$ yields algorithms that rescale the largest singular value to 1, which ensures that the resulting updates cannot produce arbitrarily large updates in y-space. This can be thought of as a weaker form of scale invariance. Just as 5, equation 21 defines an interpolation between gradient descent ($\beta = 0$, $\kappa = 1$) and the Gauss-Newton method ($\beta = 0$, $\kappa = -1$) as well.

Scalar prefactor term β: We test β-values between 0, no scale correction, and -0.5, which fully normalizes the effect of the largest singular value for $\kappa = -0.5$. The results are shown in figure 12a. Compared to the other hyperparameters, we observe that β has only little influence on the outcome, which is why we decided to present the method without this parameter in the main text.

Exponent of the diagonal singular value matrix κ: We test κ for various values between 1.0, stochastic gradient descent, and -1, Gauss-Newton. The results are shown in figure 12b. For positive values, curves stagnate early, while for negative κ, the final loss values are several orders of magnitude better. The HIG curve corresponding to $\beta = -0.5$ achieves the best result. This supports our argumentation that a strong dependence on this parameter exists, and that a choice of $\kappa = -0.5$ is indeed a good compromise for scale-correcting updates of reasonable size. The strong improvement as soon as κ becomes negative indicates that the collective inversion of the feedback of different data points of the mini-batch is an important ingredient in our method.

Truncation parameter τ: To understand the effect of this parameter, we consider the singular value decomposition (SVD) of the network-solver Jacobian, which is determined by the SVDs of the network Jacobian and the solver Jacobian. The singular values of a matrix product AB depend non-trivially on the singular values of the matrices A and B. In the simplest case, the singular values of the matrix product are received by multiplication of the individual singular values of both matrix factors. In the general case, this depends on how the singular vectors of A and B overlap with each other. However, it is likely that singular vectors with a small singular value of A or B overlap significantly with singular vectors with a small singular value of AB. For this reason, it is important not to truncate too much as this might remove the small-scale physics modes that we are ultimately trying to preserve in order to achieve accurate results. On the other hand, less truncation leads to large updates of network weights on a scale beyond the validation of the linear approximation by first-order derivatives. These uncontrolled network modifications can lead to over-saturated neurons and prevent further training progress.

From a practical point of view, we choose τ according to the accuracy of the pure physics optimization problem without a neural network. For the quantum dipole training, this value was set to 10^{-5}. Trying to solve the pure physics optimization with far smaller values leads to a worse result or no convergence at all. The network training behaves in line with this: Figure 9 shows that the network does not learn to control the quantum system with τ-values far smaller than 10^{-5} . For the nonlinear oscillator system, the pure physics optimization is stable over a large range of τ-values with similarly good results. For the network training, we chose τ to be 10^{-6}. We conducted further experiments for the network training with different τ from 10^{-5} to 10^{-10} presented in figure 13,

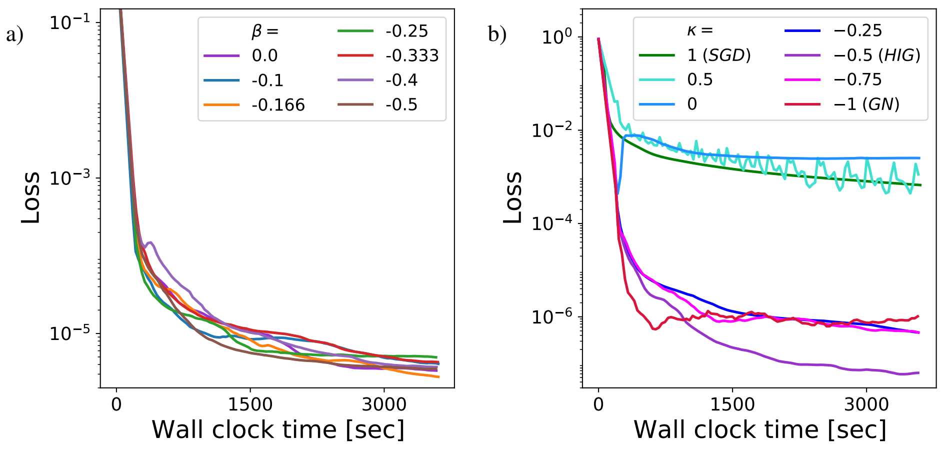


Figure 12: a) Ablation experiments with the β-hyperparameter, and b) with the κ-hyperparameter.

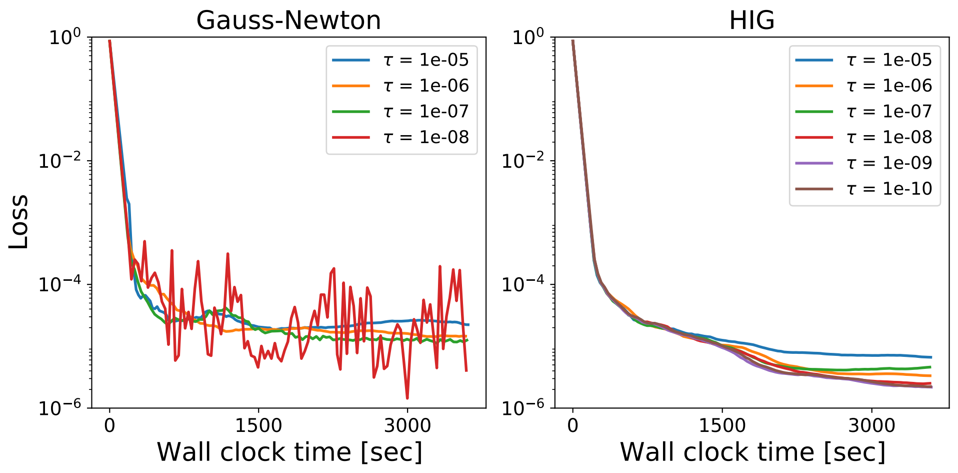


Figure 13: Ablation experiments with the τ-hyperparameter.

which show that HIGs have a similar tolerance in τ. For a comparison, we also plotted Gauss-Newton curves for different τ. We observe that GN curves become more unstable for smaller truncation values τ and diverge in the case 10^{-9} and 10^{-10} while HIG curves achieve overall better loss values and start to converge in this parameter.

Copyright and Patents on ICLR Papers

According to U.S. Copyright Office's page **What is a Copyright.** When you create an original work you are the author and the owner and hold the copyright, unless you have an agreement to transfer the copyright to a third party such as the company or school you work for.

Authors do not transfer the copyright of their paper to ICLR, instead they grant ICLR a non-exclusive, perpetual, royalty-free, fully-paid, fully-assignable license to copy, distribute and publicly display all or part of the paper.

Solver-in-the-Loop: Learning from Differentiable Physics to Interact with Iterative PDE-Solvers

Kiwon Um[1,2] **Robert Brand**[1] **Yun (Raymond) Fei**[3] **Philipp Holl**[1] **Nils Thuerey**[1]

[1]Technical University of Munich, [2]LTCI, Telecom Paris, IP Paris, [3]Columbia University

`kiwon.um@telecom-paris.fr, robert.brand@tum.de`
`yf2320@columbia.edu, {philipp.holl, nils.thuerey}@tum.de`

Abstract

Finding accurate solutions to partial differential equations (PDEs) is a crucial task in all scientific and engineering disciplines. It has recently been shown that machine learning methods can improve the solution accuracy by correcting for effects not captured by the discretized PDE. We target the problem of reducing numerical errors of iterative PDE solvers and compare different learning approaches for finding complex correction functions. We find that previously used learning approaches are significantly outperformed by methods that integrate the solver into the training loop and thereby allow the model to interact with the PDE during training. This provides the model with realistic input distributions that take previous corrections into account, yielding improvements in accuracy with stable rollouts of several hundred recurrent evaluation steps and surpassing even tailored supervised variants. We highlight the performance of the differentiable physics networks for a wide variety of PDEs, from non-linear advection-diffusion systems to three-dimensional Navier-Stokes flows.

1 Introduction

Numerical methods are prevalent in science to improve the understanding of our world, with applications ranging from climate modeling [55, 53] over simulating the efficiency of airplane wings [47] to analyzing blood flow in a human body [27]. These applications are extremely costly to compute due

34th Conference on Neural Information Processing Systems (NeurIPS 2020), Vancouver, Canada.

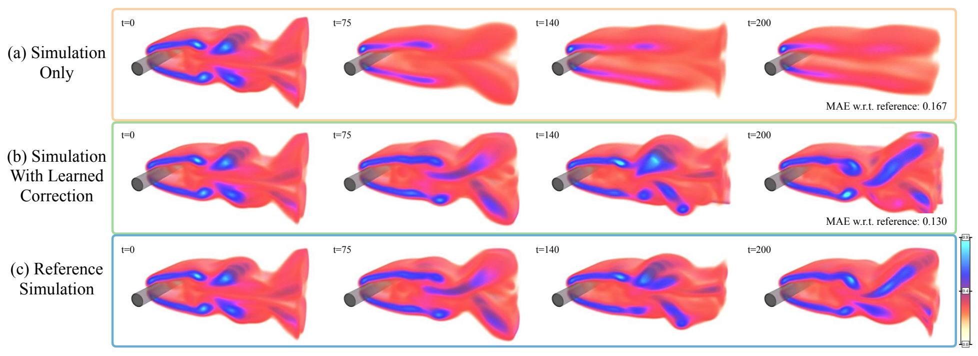

(a) Simulation Only
(b) Simulation With Learned Correction
(c) Reference Simulation
t=0
t=75
t=140
t=200
MAE w.r.t. reference: 0.167
MAE w.r.t. reference: 0.130
0.8
0.4
0.0

Figure 1: A 3D fluid problem (shown in terms of vorticity) for which the regular simulation introduces numerical errors that deteriorate the resolved dynamics (a). Combining the same solver with a learned corrector trained via differentiable physics (b) significantly reduces errors w.r.t. the reference (c).

to the fine spatial and temporal resolutions required in real-world scenarios. In this context, deep learning methods are receiving strongly growing attention [40, 4, 18] and show promise to account for those components of the solutions that are difficult to resolve or are not well captured by our physical models. Physical models typically come in the form of PDEs and are discretized in order to be processed by computers. This step inevitably introduces numerical errors. Despite a vast amount of work [15, 2] and experimental evaluations [7, 41], analytic descriptions of these errors remain elusive for most real-world applications of simulations.

In our work, we specifically target the numerical errors that arise in the discretization of PDEs. We show that, despite the lack of closed-form descriptions, discretization errors can be seen as functions with regular and repeating structures and, thus, can be learned by neural networks. Once trained, such a network can be evaluated locally to improve the solution of a PDE-solver, i.e., to reduce its numerical error.

The core of most numerical methods contains some form of iterative process – either in the form of repeated updates over time for explicit solvers or even within a single update step for implicit solvers. Hence, we focus on iterative PDE solving algorithms [17]. We show that neural networks can achieve excellent performance if they take the reaction of the solver into account. This interaction is not possible with supervised learning on pre-computed data alone. Even small inference errors of a supervised model can quickly accumulate over time [57, 29], leading to a data distribution that differs from the distribution of the pre-computed data. For supervised learning methods, this causes deteriorated inference at best and solver explosions at worst.

We demonstrate that neural networks can be successfully trained if they can *interact* with the respective PDE solver during training. To achieve this, we leverage differentiable simulations [1, 58]. Differentiable simulations allow a trained model to autonomously explore and experience the physical environment and receive directed feedback regarding its interactions throughout the solver iterations. Hence, our work fits into the broader context of machine learning as differentiable programming, and we specifically target recurrent interactions of highly non-linear PDEs with deep neural networks. This combination bears particular promise: it improves generalizing capabilities of the trained models by letting the PDE-solver handle large-scale changes to the data distribution such that the learned model can focus on localized structures not captured by the discretization. While physical models generalize very well, learned models often specialize in data distributions seen at training time. However, we will show that, by combining PDE-based solvers with a learned model, we can arrive at hybrid methods that yield improved accuracy while handling solution manifolds with significant amounts of varying physical behavior.

We show the advantages of training via differentiable physics for explicit and implicit solvers applied to a broad class of canonical PDEs. For explicit and semi-implicit solvers, we consider advection-diffusion systems as well as different types of Navier-Stokes variants. We showcase models trained with up to 128 steps of a differentiable simulator and apply our model to complex three-dimensional (3D) flows, as shown in Fig. 1. Additionally, we present a detailed empirical study of different approaches for training neural networks in conjunction with iterative PDE-solvers for recurrent rollouts of several hundred time steps. On the side of implicit solvers, we consider the Poisson problem [37], which is an essential component of many PDE models. Here, our method outperforms existing techniques on predicting initial guesses for a conjugate gradient (CG) solver by receiving feedback from the solver at training time. The source code for this project is available at `https://github.com/tum-pbs/Solver-in-the-Loop`.

Previous Work Combining machine learning techniques with PDE models has a long history in machine learning [13, 28, 8]. More recently, deep-learning-based methods were successfully applied to infer stencils of advection-diffusion problems [4], to discover PDE formulations [35, 42, 52], and to analyze families of Poisson equations [36]. While identifying governing equations represents an interesting and challenging task, we instead focus on a general method to improve the solutions of chosen spaces of solutions.

Other studies have investigated the similarities of dynamical systems and deep learning methods [65] and employed conservation laws to learn systems described by Hamiltonian mechanics [18, 12]. Existing studies have also identified discontinuities in finite-difference solutions with deep learning [46] and focused on improving the iterative behavior of linear solvers [24]. So-called Koopman operators likewise represent an interesting opportunity for deep learning algorithms [40, 32]. While these methods replace the PDE-based time integration with a learned version, our models rely on and interact with a PDE-solver that provides a coarse approximation to the problem. Hence, our models always alternate between inference via an artificial neural network (ANN) and a solver step. This distinguishes our work from studies of recurrent ANN architectures [11, 54, 62] as the PDE-solver can introduce significant non-linearities in-between evaluations of the ANN.

We focus on chaotic systems for which fluid flow represents an exciting and challenging problem domain that is highly relevant for industrial applications. Deep learning methods have received significant amounts of attention in this area [31]. For example, both steady [19] and unsteady [40], as well as multi-phase flows [16] have been investigated with deep learning based approaches. Turbulence closure modeling has been an area of particular focus [59, 38, 6]. Additionally, convolutional neural networks (CNNs) were studied for stochastic sub-grid modeling [60], airfoil flow problems [56, 67], and as part of generative networks to leverage the fast inference of pre-trained models [10, 66, 29]. Other studies have targeted the unsupervised learning of divergence-free corrections [57] or incorporated PDE-based loss functions to represent individual flow solutions via ANNs [43, 52]. In addition to temporal predictions of turbulent flows [39], similar algorithms were more recently also employed for classification problems [20]. However, to the best of our knowledge, the existing methods do not let ANNs interact with solver in a recurrent manner. As we will demonstrate below, this combination yields significant improvements in terms of inference accuracy.

While we focus on Eulerian, i.e., grid-based discretizations, the Lagrangian viewpoint is a popular alternative. While a variety of studies has investigated graph-based simulators, e.g., for rigid-body physics in the context of human reasoning [5, 64, 3] or weather predictions [51], particles are also a popular basis for fluid flow problems [33, 61, 48]. Despite our Eulerian focus, Lagrangian methods could likewise benefit from incorporating differentiable solvers into the training process.

Our work shares the motivation of previous work to use differentiable components at training time [1, 14, 58, 9] and frameworks for differentiable programming [50, 25, 26, 23]. Differentiable physics solvers were proposed for inverse problems in the context of liquids [49], cloth [34], soft robots [25], and molecular dynamics [63]. While these studies typically focus on optimization problems or replace solvers with learned components, we focus on the interaction between the two. Hence, in contrast to previous work, we always rely on a PDE-solver to yield a coarse approximate solution and improve its performance via a trained ANN.

2 Learning to Reduce Numerical Errors

Numerical methods yield approximations of a smooth function $\boldsymbol{u}$ in a discrete setting and invariably introduce errors. These errors can be measured in terms of the deviation from the exact analytical solution. For discrete simulations of PDEs, they are typically expressed as a function of the truncation, $O(\Delta t^k)$. Higher-order methods, with large k, are preferable but difficult to arrive at in practice. For practical schemes, no closed-form expression exists for truncation errors, and the errors often grow exponentially as solutions are integrated over time. We investigate methods that solve a discretized PDE $\mathcal{P}$ by performing discrete time steps Δt. Each subsequent step can depend on any number of previous steps, $\boldsymbol{u}(\boldsymbol{x}, t+\Delta t) = \mathcal{P}(\boldsymbol{u}(\boldsymbol{x}, t), \boldsymbol{u}(\boldsymbol{x}, t-\Delta t), ...)$, where $\boldsymbol{x} \in \Omega \subseteq \mathbb{R}^d$ for the domain Ω in d dimensions, and $t \in \mathbb{R}^+$.

Problem Statement: We consider two different discrete versions of the same PDE $\mathcal{P}$, with $\mathcal{P}_R$ denoting a more accurate discretization with solutions $\mathbf{r} \in \mathscr{R}$ from the *reference manifold*, and an approximate version $\mathcal{P}_s$ with solutions $\boldsymbol{s} \in \mathscr{S}$ from the *source manifold*. We consider $\mathbf{r}$ and $\boldsymbol{s}$ to be states at a certain instance in time, i.e., they represent phase space points, and evolutions over time are given by a trajectory in each solution manifold. As we focus on the discrete setting, a solution over time consists of a *reference sequence* $\{\mathbf{r}_t, \mathbf{r}_{t+\Delta t}, \cdots, \mathbf{r}_{t+k\Delta t}\}$ in the solution manifold $\mathscr{R}$, and correspondingly, a more coarsely approximated *source sequence* $\{\boldsymbol{s}_t, \boldsymbol{s}_{t+\Delta t}, \cdots, \boldsymbol{s}_{t+k\Delta t}\}$ exists in the solution manifold $\mathscr{S}$. We also employ a mapping operator $\mathcal{T}$ that transforms a phase space point from one solution manifold to a suitable point in the other manifold, e.g., for the initial conditions of the sequences above, we typically choose $\boldsymbol{s}_t = \mathcal{T}\mathbf{r}_t$. We discuss the choice of $\mathcal{T}$ in more detail in the appendix, but in the simplest case, it can be obtained via filtering and re-sampling operations.

By evaluating $\mathcal{P}_R$ for $\mathscr{R}$, we can compute the points of the phase space sequences, e.g., $\mathbf{r}_{t+\Delta t} = \mathcal{P}_R(\mathbf{r}_t)$ for an update scheme that only depends on time t. Without loss of generality, we assume a fixed Δt and denote a state $\mathbf{r}_{t+k\Delta t}$ after k steps of size Δt with $\mathbf{r}_{t+k}$. Due to the inherently different numerical approximations, $\mathcal{P}_s(\mathcal{T}\mathbf{r}_t) \neq \mathcal{T}\mathbf{r}_{t+1}$ for the vast majority of states. In chaotic systems, such differences typically grow exponentially over time until they saturate at the level of mean difference between solutions in the two manifolds. We use an L^2-norm in the following to quantify the deviations, i.e., $\mathcal{L}(\boldsymbol{s}_t, \mathcal{T}\mathbf{r}_t) = \|\boldsymbol{s}_t - \mathcal{T}\mathbf{r}_t\|_2$. Our learning goal is to arrive at a correction operator $\mathcal{C}(\boldsymbol{s})$ such that a solution to which the correction is applied has a lower error than an unmodified solution: $\mathcal{L}(\mathcal{P}_s(\mathcal{C}(\mathcal{T}\mathbf{r}_{t_0})), \mathcal{T}\mathbf{r}_{t_1}) < \mathcal{L}(\mathcal{P}_s(\mathcal{T}\mathbf{r}_{t_0}), \mathcal{T}\mathbf{r}_{t_1})$. The correction function $\mathcal{C}(\boldsymbol{s}|\theta)$ is represented as a deep neural network with weights θ and receives the state $\boldsymbol{s}$ to infer an additive correction field with the same dimension. To distinguish the original phase states $\boldsymbol{s}$ from corrected ones, we denote the latter with $\tilde{\boldsymbol{s}}$, and we use an exponential notation to indicate a recursive application of a function, i.e.,

$$\boldsymbol{s}_{t+n} = \mathcal{P}_s(\mathcal{P}_s(\cdots \mathcal{P}_s(\mathcal{T}\mathbf{r}_t)\cdots)) = \mathcal{P}_s^n(\mathcal{T}\mathbf{r}_t)\ . \tag{1}$$

Within this setting, any type of learning method naturally needs to compare states from the source domain with the reference domain in order to bridge the gap between the two solution manifolds. How the evolution in the source manifold at training time is computed, i.e., if and how the corrector interacts with the PDE, has a profound impact on the learning process and the achievable final accuracy. We distinguish three cases: no interaction, a pre-computed form of interaction, and a tight coupling via a differentiable solver in the training loop.

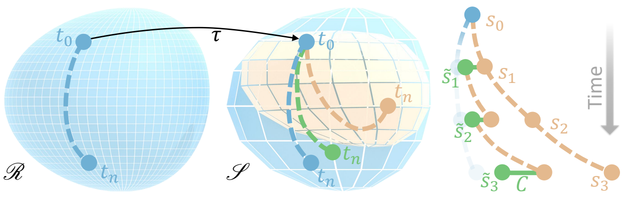


Figure 2: Transformed solutions of the reference sequence computed on $\mathscr{R}$ (blue) differ from solutions computed on the source manifold $\mathscr{S}$ (orange). A correction function $\mathcal{C}$ (green) updates the state after each iteration to more closely match the projected reference trajectory on $\mathscr{S}$.

- **Non-interacting (NON)**: The learning task purely uses the unaltered PDE trajectories, i.e., $\boldsymbol{s}_{t+n} = \mathcal{P}_s^n(\mathcal{T}\mathbf{r}_t)$ with n evaluations of $\mathcal{P}_s$. These trajectories are fully contained in the source manifold $\mathscr{S}$. Learning from these states means that a model will not see any states that deviate from the original solutions. As a consequence, models trained in this way can exhibit undesirably strong error accumulations over time. This corresponds to learning from the difference between the orange and blue trajectories in Fig. 2, and most commonly applied supervised approaches use this variant.
- **Pre-computed interaction (PRE)**: To let an algorithm learn from states that are closer to those targeted by the correction, i.e., the reference states, a pre-computed or analytic correction is applied. Hence, the training process can make use of phase space states that deviate from those in $\mathscr{S}$, as shown in green in Fig. 2, to improve inference accuracy and stability. This approach can be formulated as $\boldsymbol{s}_{t+n} = (\mathcal{P}_s\mathcal{C}_{\text{pre}})^n(\mathcal{T}\mathbf{r}_t)$ with a pre-computed correction function $\mathcal{C}_{\text{pre}}$. In this setting, the states $\boldsymbol{s}$ are corrected without employing a neural network, but they should ideally resemble the states achievable via the learned correction later on. As the modified states $\boldsymbol{s}$ are not influenced by the learning process, the training data can be pre-computed. A correction model $\mathcal{C}(\boldsymbol{s}|\theta)$ is trained via $\tilde{\boldsymbol{s}}$ that replaces $\mathcal{C}_{\text{pre}}$ at inference time.
- **Solver-in-the-loop (SOL)**: By integrating the learned function into a differentiable physics pipeline, the corrections can interact with the physical system, alter the states, and receive gradients about the future performance of these modifications. The learned function $\mathcal{C}$ now depends on states that are modified and evolved through $\mathcal{P}$ for one or more iterations. A trajectory for n evaluations of $\mathcal{P}_s$ is given by $\tilde{\boldsymbol{s}}_{t+n} = (\mathcal{P}_s\mathcal{C})^n(\mathcal{T}\mathbf{r}_t)$, with $\mathcal{C}(\tilde{\boldsymbol{s}}|\theta)$. The key difference with this approach is that $\mathcal{C}$ is trained via $\tilde{\boldsymbol{s}}$, i.e., states that were affected by previous evaluations of $\mathcal{C}$, and it affects $\tilde{\boldsymbol{s}}$ in the following iterations. As for (PRE), this learning setup results in a trajectory like the green one shown in Fig. 2, however, in contrast to before, the learned correction itself influences the evolution of the trajectory, preventing a gap for the data distribution of the inputs.

In addition to these three types of interaction, a second central parameter is the look-ahead trajectory per iteration and mini-batch of the optimizer during learning. A subscript n denotes the number of steps over which the future evolution is recursively evaluated, e.g., SOL$_n$. The objective function, and hence the quality of the correction, is evaluated with the training goal to minimize $\sum_{i=t}^{t+n} \mathcal{L}(\boldsymbol{s}_i, \mathbf{r}_i)$. Below, we will analyze a variety of learning methodologies that are categorized via learning methodology (NON, PRE or SOL) and look-ahead horizon n.

3 Experiments

We now provide a summary and discussion of our experiments with the different types of PDE interactions for a selection of physical models. Full details of boundary conditions, parameters, and discretizations of all five PDE scenarios are given in App. B.

3.1 Model Equations and Data Generation

We investigate a diverse set of constrained advection-diffusion models of which the general form is

$$\partial \boldsymbol{u}/\partial t = -\boldsymbol{u} \cdot \nabla \boldsymbol{u} + \nu \nabla \cdot \nabla \boldsymbol{u} + \mathbf{g} \quad \text{subject to} \quad \boldsymbol{M}\boldsymbol{u} = 0, \tag{2}$$

where $\boldsymbol{u}$ is the velocity, ν denotes the diffusion coefficient (i.e., viscosity), and $\mathbf{g}$ denotes external forces. The constraint matrix $\boldsymbol{M}$ contains an additional set of equality constraints imposed on $\boldsymbol{u}$. In total, we target four scenarios: pure non-linear advection-diffusion (Burger's equation), two-dimensional Navier-Stokes flow, Navier-Stokes coupled with a second advection-diffusion equation for a buoyancy-driven flow, and a 3D Navier-Stokes case. Also, we discuss CG solvers in the context of differentiable operators below.

For each of the five scenarios, we implement the non-interacting evaluation (NON) by pre-computing a large-scale data set that captures a representative and non-trivial space of solutions in $\mathscr{S}$. The reference solutions from $\mathscr{R}$ are typically computed with the same numerical method using a finer discretization (4x in our setting, with effective resolutions of 128^2 and higher). The PDEs are parametrized such that the change of discretization leads to substantial differences when integrated over time. For several of the 2D scenarios, we additionally train models with data sets of trajectories that have been corrected with other pre-computated correction functions. For these PRE variants, we use a time-regularized, constrained least-squares corrector [21] to obtain corrected phase state points. For the SOL variants, we employ a differentiable PDE-solver that runs mini-batches of simulations and provides gradients for all operations of the solving process within the deep learning framework. This allows gradients to freely propagate through the PDE-solver and coupled neural networks via automatic differentiation. For $n > 1$, i.e., PDE-based look-ahead at training time, the gradients are back-propagated through the solver $n - 1$ times, and the difference w.r.t. a pre-computed reference solution is evaluated for all intermediate results.

3.2 Training Procedure

The neural network component $F(\boldsymbol{s} \mid \theta)$ of the correction function is realized with a fully convolutional architecture. As our focus lies on the methodology for incorporating PDE models into the training, the architectures are intentionally kept simple. However, they were chosen to yield high accuracy across all variants. Our networks typically consist of 10 convolutional layers with 16 features each, interspresed with ReLU activation functions using kernel sizes of 3^d and 5^d. The networks parameters θ are optimized with a fixed number of steps with an ADAM optimizer [30] and a learning rate of 10^{-4}. For validation, we use data sets generated from the same parameter distribution as the training sets. All results presented in the following use test data sets whose parameter distributions differ from the ones of the training data set.

We quantify the performance of the trained models by computing the mean absolute error between a computed solution and the corresponding projected reference for n consecutive steps of a simulation. We report absolute error values for different models in comparison to an unmodified source trajectory from $\mathscr{S}$. Additionally, relative improvements are given w.r.t. the difference between unmodified source and reference solutions. An improvement by 100% would mean that the projected reference is reproduced perfectly, while negative values indicate that the modified solution deviates more from the reference than the original source trajectory.

4 Results

Our experiments show that learned correction functions can achieve substantial gains in accuracy over a regular simulation. When training the correction functions with differentiable physics, this additionally yields further improvements of more than 70% over supervised and pre-computed approaches from previous work. A visual overview of the different tests is given in Fig. 3, and a summary of the full evaluation from the appendix is provided in Fig. 4 and Table 1. In the appendix, we also provide error measurements w.r.t. physical quantities such as kinetic energy and frequency content. The source code of our experiments and analysis will be published upon acceptance.

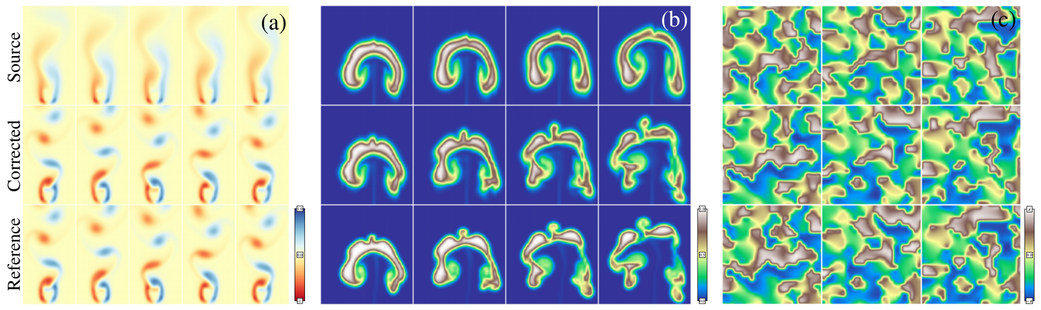


Figure 3: Our PDE scenarios cover a wide range of behavior including (a) vortex shedding, (b) complex buoyancy effects, and (c) advection-diffusion systems. Shown are different time steps (l.t.r.) in terms of vorticity for (a), transported density for (b), and angle of velocity direction for (c).

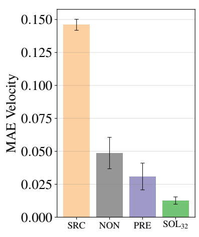
Unsteady Wake Flow The PDE scenario for unsteady wake flows represents a standard benchmark case for fluids [44, 40] and involves a continuous inflow with a fixed, circular obstacle, which induces downstream vortex shedding with distinct frequencies depending on the Reynolds number. For coarse discretizations, the approximation errors distort the flow leading to deteriorated motions or suppressed vortex shedding altogether. An example flow configuration is shown in Fig. 3a. In this scenario, the simplest method (NON) yields stable training and a model that already reduces the mean absolute error (MAE) from 0.146 for a regular simulation without correction (SRC) to an MAE of 0.049 when applying the learned correction. The pre-computed correction (PRE) improves on this behavior via its time regularization with an error of 0.031. A SOL_{32} model trained with a differentiable physics solver for 32 time steps in each iteration of ADAM yields a significantly lower error of 0.013. This means, the numerical errors of the source simulation w.r.t. the reference were reduced by more than a factor of 10. Despite the same architecture and weight count for all three models, the overall performance varies strongly, with the SOL_{32} version outperforming the simpler variants by 73% and more. An example of the further evaluations provided in the appendix is given in Fig. 4h.

Buoyancy-driven Flow We evaluate buoyancy-driven flows as a scenario with increased complexity. In addition to an incompressible fluid, a second, non-uniform marker quantity is advected with the flow that exerts a buoyancy force. This coupled system of equations leads to interesting and complex swirling behavior over time. We additionally use this setup to highlight that the reference solutions can be obtained with different discretization schemes. We use a higher-order advection scheme in addition to a $4\times$ finer spatial discretization to compute the reference data.

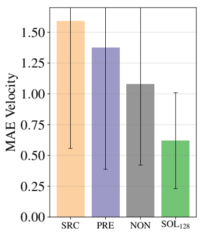
Interestingly, the correction functions benefit from particularly long rollouts at training time in this scenario. Models with simple pre-computed or unaltered trajectories yield mean errors of 1.37 and 1.07 compared to an error of 1.59 for the source simulation, respectively. Instead, a model trained with differentiable physics with 128 steps (SOL_{128}) successfully reduces the error to 0.62, an improvement of more than 59% compared to the unmodified simulation.

Forced Advection-Diffusion A third scenario employs Burger's equation as a physical model. We mimic the setup from previous work [4] to inject energy into the system via a forcing term with a spectrum of sine waves. This forcing prevents the system from dissipating to relatively static and slowly moving configurations. While the PRE and NON versions yield clear improvements, the SOL versions do not significantly outperform the simpler baselines. This illustrates a limitation of long rollouts via differentiable physics: Learned correction functions need to be able to anticipate future behavior to make high-quality corrections. The randomized forcing in this example severely limits the number of future steps that can accurately be predicted given one state. This behavior contrasts with other physical systems without external disturbances, where a single state uniquely determines its evolution. We show in the appendix that the SOL models with an increased number of interaction steps pay off when the external disturbances are absent.

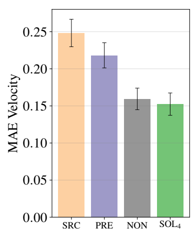
Conjugate Gradient Solver We turn to iterative solvers for linear systems of equations to illustrate another aspect of learning from differentiable physics: its importance for the propagation of boundary condition effects. As our learning objective, we target the inference of initial guesses for CG solvers [22]. Following previous work [57], we target Poisson problems of the form $\nabla \cdot \nabla p = \nabla \cdot \boldsymbol{u}$, which arise for projections of a velocity $\boldsymbol{u}$ to a divergence-free state. Instead of fully relying on an ANN



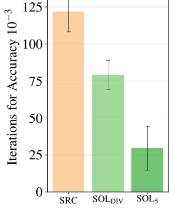
(a) Unsteady wake



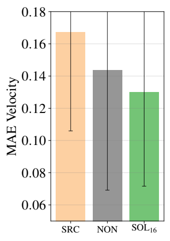
(b) Buoyancy-driven



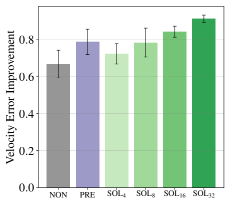
(c) Advection-diffusion



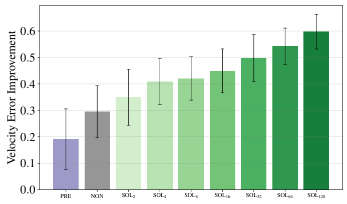
(d) CG solver



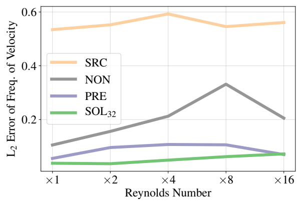
(e) 3D wake



(f) Unsteady wake, look-ahead



(g) Buoyancy-driven, look-ahead



(h) Unsteady wake, frequency error

Figure 4: (a)-(e) Numerical approximation error w.r.t. reference solution for unaltered simulations (SRC) and with learned corrections. The models trained with differentiable physics and look-ahead achieve significant gains over the other models. (f,g) Relative improvement over varying look-ahead horizons. (h) A frequency-based evaluation for the unsteady wake flow scenario.

to produce the pressure field p, we instead target the learning objective to produce an initial guess, which is improved by a regular CG solver until a given accuracy threshold is reached.

This goal can be reached by directly minimizing the right-hand side term $\nabla \cdot \boldsymbol{u}$, similar to physics-based loss terms proposed in a variety of studies [43, 52]. Alternatively, we can employ a differentiable CG solver and formulate the learning goal as minimizing the same residual after n steps of the CG solver (similar to the SOL$_n$ models above). While the physics-based loss version reduces the initial divergence more successfully, it fares badly when interacting with the CG solver: compared to the SOL version, it requires 63% more steps to reach a desired accuracy. Inspecting the inferred solutions reveals that the former model leads to comparatively large errors near boundaries, which are small for each grid cell but significantly influence the solution on a large scale. The SOL version immediately receives feedback about this behavior via the differentiable solver iterations. I.e., the differentiable solver provides a look-ahead of how different parts of the solution affect future states. In this way, it can anticipate problems such as those in the vicinity of boundary conditions.

Three-dimensional Fluid Flow Lastly, we investigate a 3D case of incompressible flow. The overall setup is similar to the unsteady wake flow in two dimensions outlined above, but the third dimension extends the axes of rotation in the fluid from one to three, yielding a very significant increase in complexity. As a result, the flow behind the cylindrical obstacle quickly becomes chaotic and forms partially turbulent eddies, as shown in Fig. 1. This scenario requires significantly larger models to learn a correction function, and the NON version does not manage to stabilize the flow consistently. Instead, the SOL$_{16}$ version achieves stable rollouts for several hundred time steps and successfully corrects the numerical inaccuracies of the coarse discretization, improving the numerical accuracy of the source (SRC) simulation by more than 22% across a wide range of configurations.

5 Ablations and Discussion

We performed an analysis of the proposed training via differentiable physics to highlight which hyperparameters most strongly influence results. Specifically, we evaluate varying look-ahead horizons, different model architectures, training via perturbations, and pre-computed variants.

Future Look-Ahead For systems with deterministic behavior, long rollouts via differentiable physics at training time yield significant improvements, as shown in Fig. 4f and 4g. While training with a few (1 to 4) steps yields improvements of up to 40% for the buoyancy-driven flow scenario, this number can be raised significantly by increasing the look-ahead at training time. A performance of more than 54% can be achieved by 64 recurrent solver iterations, while raising the look-ahead to

Table 1: A summary of the quantitative evaluation for the five PDE scenarios. SOL_s denotes a variant with shorter look-ahead compared to SOL. (* For the CG solver scenario, iterations to reach an accuracy of 0.001 are given. Here, SOL_s denotes the physics-based loss version.)

Exp.	**Mean absolute error of velocity**					**Rel. improvement**			
	SRC	PRE	NON	SOL_s	SOL	PRE	NON	SOL_s	SOL
Wake Flow	0.146±0.004	0.031±0.010	0.049±0.012	0.041±0.009	0.013±0.003	79%	67%	72%	91%
Buoyancy	1.590±1.033	1.373±0.985	1.080±0.658	0.944±0.614	0.620±0.390	19%	29%	41%	60%
Adv.-diff.	0.248±0.019	0.218±0.017	0.159±0.015	0.152±0.015	0.158±0.017	12%	36%	39%	36%
*CG Solver	121.6±13.44	-	-	79.03±10.02	29.59±14.83	-	-	35%	76%
3D Wake	0.167±0.061	-	0.144±0.074	-	0.130±0.058	-	14%	-	22%

128 yields average improvements of 60%. Our tests consistently show that, without changing the number of weights or the architecture of a network, the gradients provided by the longer rollout times allow the network to anticipate the behavior of the physical system better and react to it. Throughout our tests, similar performances could not be obtained by other means.

Generalization The buoyancy scenario also highlights the very good generalizing capabilities of the resulting models. All test simulations were generated with an out-of-distribution parametrization of the initial conditions, leading to substantially different structures, and velocity ranges over time.

Training with Noise An interesting variant to stabilize physical predictions in the context of Graph Network-based Simulators was proposed by Sanchez et al. [48]. They report that perturbations of input features with noise lead to more stable long-term rollouts. We mimic this setup in our Eulerian setting by perturbing the inputs to the neural networks with $\mathcal{N}(0, \sigma)$ for varying strengths σ. While a sweet spot with improvements of 34.5% seems to exist around $\sigma = 10^{-4}$, the increase in performance is small compared to a model with less perturbations (30.6%), as training with an increased look-ahead for the SOL models gives improvements up to 60.0%.

Training Stability The physical models we employ introduce a large amount of complexity into the training loop. Especially during the early stages of training, an inferred correction can overly distort the physical state. Performing time integration via the PDE then typically leads to exponential increases of existing oscillations and a diverging calculation. Hence, we found it important to pre-train networks with small look-aheads (we usually use SOL_2 models), and then continue training with longer recurrent iterations for the look-ahead. While this scheme can be applied hierarchically, we saw no specific gains from, e.g., starting a SOL_{32} training with a SOL_2 model versus a SOL_{16} model.

Runtime Performance The training via differentiable physics incurs an increased computational cost at training time, as the PDE model has to be evaluated for n steps for each learning iteration, and the calculation of the gradients is typically of similar complexity as the evaluation of the PDE itself. However, this incurs only moderate costs in our tests. For example, for the buoyancy-driven flow, the training time increases from 0.21 seconds per iteration on average for SOL_2 to 0.42s for SOL_4, and 1.25s for SOL_{16}. The look-ahead additionally provides n times more gradients at training time, and the inference time of the resulting models is not affected. Hence, the training cost can quickly pay off in practical scenarios by yielding more accurate results without any increase in cost at inference time.

Computing solutions with the resulting hybrid method which alternates PDE evaluations and ANN inference also provides benefits in terms of evaluation performance: A pre-trained, fully convolutional CNN has an $\mathcal{O}(n)$ cost for n degrees of freedom, in contrast to many PDE-solvers with a super-linear complexity. For example, a simulation as shown in Fig. 1 involving the trained model took 13.3s on average for 100 time steps, whereas a CPU-based reference simulation required 913.2s. A speed-up of more than 68×.

6 Conclusions

We have demonstrated how to achieve significant reductions of numerical errors in PDE-solvers by training ANNs with long look-ahead rollouts and differentiable physics solvers. The resulting models yield substantially lower errors than models trained with pre-computed data. We have additionally provided a first thorough evaluation of different methodologies for letting PDE-solvers interact with recurrent ANN evaluations.

Identical networks yield significantly better results purely by having the solver in the learning loop. This indicates that the numerical errors have regular structures that can be learned and corrected via learned representations. The resulting networks likewise improve generalization for out-of-distribution samples and provide stable, long-term recurrent predictions. Our results have the potential to enhance learning physical priors for a variety of deep learning tasks. Beyond engineering applications and medical simulations, a particularly interesting application of our approach is weather prediction [45], where a simple differentiable solver could be augmented with a learned correction function to recover the costly predictions of operational forecasting systems.

Overall, we hope that the demonstrated gains in accuracy will help to establish trained neural networks as components in the numerical toolbox of computational science.

Broader Impact

PDE-based models are very commonly used and can be applied to a wide range of applications, including weather and climate, epidemics, civil engineering, manufacturing processes, and medical applications. Our work has the potential to improve how these PDEs are solved. As PDE-solvers have a long history, there is a wide range of established tools, some of which still use COBOL and FORTRAN. Hence, it will not be easy to integrate deep learning methods into the existing solving pipelines, but in the long run, our method could yield solvers that compute more accurate solutions with a given amount of computational resources.

Due to the wide range of applications of PDEs, our methods could also be used in the development of military equipment (machines and weapons) or other harmful systems. However, our method shares this danger with all numerical methods. For the discipline of computational science as a whole, we see more positive aspects when computer simulations become more powerful. Nonetheless, we will encourage users of our method likewise to consider ethical implications when employing PDE-solvers with learning via differentiable physics.

Acknowledgments and Disclosure of Funding

This work is supported by the ERC Starting Grant *realFlow* (StG-2015-637014).

References

[1] B. Amos and J. Z. Kolter. OptNet: Differentiable optimization as a layer in neural networks. In *International Conference on Machine Learning*, 2017.

[2] V. I. Arnold. *Geometrical methods in the theory of ordinary differential equations*, volume 250. Springer Science & Business Media, 2012.

[3] V. Bapst, A. Sanchez-Gonzalez, C. Doersch, K. Stachenfeld, P. Kohli, P. Battaglia, and J. Hamrick. Structured agents for physical construction. In *International Conference on Machine Learning*, pages 464–474, 2019.

[4] Y. Bar-Sinai, S. Hoyer, J. Hickey, and M. P. Brenner. Learning data-driven discretizations for partial differential equations. *Proceedings of the National Academy of Sciences*, 116(31):15344–15349, 2019.

[5] P. W. Battaglia, J. B. Hamrick, and J. B. Tenenbaum. Simulation as an engine of physical scene understanding. *Proceedings of the National Academy of Sciences*, 110(45), 2013.

[6] A. D. Beck, D. G. Flad, and C. Munz. Deep neural networks for data-driven turbulence models. *CoRR*, abs/1806.04482, 2018.

[7] M. E. Brachet, D. I. Meiron, S. A. Orszag, B. Nickel, R. H. Morf, and U. Frisch. Small-scale structure of the taylor–green vortex. *Journal of Fluid Mechanics*, 130:411–452, 1983.

[8] S. L. Brunton, J. L. Proctor, and J. N. Kutz. Discovering governing equations from data by sparse identification of nonlinear dynamical systems. *Proceedings of the National Academy of Sciences*, 113(15):3932–3937, 2016.

[9] T. Q. Chen, Y. Rubanova, J. Bettencourt, and D. K. Duvenaud. Neural ordinary differential equations. In *Advances in neural information processing systems*, pages 6571–6583, 2018.

[10] M. Chu and N. Thuerey. Data-driven synthesis of smoke flows with CNN-based feature descriptors. *ACM Trans. Graph.*, 36(4):69:1–69:14, July 2017.

[11] J. T. Connor, R. D. Martin, and L. E. Atlas. Recurrent neural networks and robust time series prediction. *IEEE transactions on neural networks*, 5(2):240–254, 1994.

[12] M. Cranmer, S. Greydanus, S. Hoyer, P. Battaglia, D. Spergel, and S. Ho. Lagrangian neural networks. *arXiv:2003.04630*, 2020.

[13] J. P. Crutchfield and B. S. McNamara. Equations of motion from a data series. *Complex systems*, 1(417-452):121, 1987.

[14] F. de Avila Belbute-Peres, K. Smith, K. Allen, J. Tenenbaum, and J. Z. Kolter. End-to-end differentiable physics for learning and control. In *Advances in neural information processing systems*, 2018.

[15] S. Ghosal. An analysis of numerical errors in large-eddy simulations of turbulence. *Journal of Computational Physics*, 125(1):187–206, 1996.

[16] F. Gibou, D. Hyde, and R. Fedkiw. Sharp interface approaches and deep learning techniques for multiphase flows. *Journal of Computational Physics*, May 2018.

[17] G. H. Golub and C. F. Van Loan. *Matrix computations*, volume 3. JHU press, 2012.

[18] S. Greydanus, M. Dzamba, and J. Yosinski. Hamiltonian neural networks. In *Advances in Neural Information Processing Systems*, pages 15353–15363, 2019.

[19] X. Guo, W. Li, and F. Iorio. Convolutional neural networks for steady flow approximation. In *SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, pages 481–490. ACM, 2016.

[20] X. He, H. L. Cao, and B. Zhu. Advectivenet: An eulerian-lagrangian fluidic reservoir for point cloud processing. *International Conference on Learning Representations (ICLR)*, 2020.

[21] C. R. Henderson. Best linear unbiased estimation and prediction under a selection model. *Biometrics*, pages 423–447, 1975.

[22] M. R. Hestenes, E. Stiefel, et al. Methods of conjugate gradients for solving linear systems. *Journal of research of the National Bureau of Standards*, 49(6):409–436, 1952.

[23] P. Holl, V. Koltun, and N. Thuerey. Learning to control pdes with differentiable physics. *International Conference on Learning Representations (ICLR)*, 2020.

[24] J.-T. Hsieh, S. Zhao, S. Eismann, L. Mirabella, and S. Ermon. Learning neural pde solvers with convergence guarantees. *arXiv:1906.01200*, 2019.

[25] Y. Hu, L. Anderson, T.-M. Li, Q. Sun, N. Carr, J. Ragan-Kelley, and F. Durand. Difftaichi: Differentiable programming for physical simulation. *International Conference on Learning Representations (ICLR)*, 2020.

[26] M. Innes, A. Edelman, K. Fischer, C. Rackauckas, E. Saba, V. B. Shah, and W. Tebbutt. A differentiable programming system to bridge machine learning and scientific computing. *arXiv 1907.07587*, 2019.

[27] B. M. Johnston, P. R. Johnston, S. Corney, and D. Kilpatrick. Non-newtonian blood flow in human right coronary arteries: steady state simulations. *Journal of biomechanics*, 37(5):709–720, 2004.

[28] I. G. Kevrekidis, C. W. Gear, J. M. Hyman, P. G. Kevrekidid, O. Runborg, C. Theodoropoulos, et al. Equation-free, coarse-grained multiscale computation: Enabling mocroscopic simulators to perform system-level analysis. *Communications in Mathematical Sciences*, 1(4):715–762, 2003.

[29] B. Kim, V. C. Azevedo, N. Thuerey, T. Kim, M. Gross, and B. Solenthaler. Deep fluids: A generative network for parameterized fluid simulations. *Computer Graphics Forum*, 2019.

[30] D. Kingma and J. Ba. Adam: A method for stochastic optimization. *arXiv:1412.6980 [cs]*, Dec. 2014.

[31] J. N. Kutz. Deep learning in fluid dynamics. *Journal of Fluid Mechanics*, 814:1–4, 2017.

[32] Y. Li, H. He, J. Wu, D. Katabi, and A. Torralba. Learning compositional koopman operators for model-based control. *arXiv:1910.08264*, 2019.

[33] Y. Li, J. Wu, R. Tedrake, J. B. Tenenbaum, and A. Torralba. Learning particle dynamics for manipulating rigid bodies, deformable objects, and fluids. *arXiv:1810.01566*, 2018.

[34] J. Liang, M. Lin, and V. Koltun. Differentiable cloth simulation for inverse problems. In *Advances in Neural Information Processing Systems*, pages 771–780, 2019.

[35] Z. Long, Y. Lu, X. Ma, and B. Dong. PDE-Net: Learning PDEs from data. *arXiv:1710.09668*, 2017.

[36] M. Magill, F. Qureshi, and H. de Haan. Neural networks trained to solve differential equations learn general representations. In *Advances in Neural Information Processing Systems*, pages 4071–4081, 2018.

[37] J. Mathews and R. L. Walker. *Mathematical methods of physics*, volume 501. WA Benjamin New York, 1970.

[38] R. Maulik and O. San. A neural network approach for the blind deconvolution of turbulent flows. *Journal of Fluid Mechanics*, 831:151–181, Oct 2017.

[39] A. Mohan, D. Daniel, M. Chertkov, and D. Livescu. Compressed convolutional LSTM: An efficient deep learning framework to model high fidelity 3d turbulence. *arXiv:1903.00033*, 2019.

[40] J. Morton, A. Jameson, M. J. Kochenderfer, and F. Witherden. Deep dynamical modeling and control of unsteady fluid flows. In *Advances in Neural Information Processing Systems*, 2018.

[41] J.-S. Pang. Error bounds in mathematical programming. *Mathematical Programming*, 79(1-3):299–332, 1997.

[42] M. Raissi and G. E. Karniadakis. Hidden physics models: Machine learning of nonlinear partial differential equations. *Journal of Computational Physics*, 357:125–141, 2018.

[43] M. Raissi, A. Yazdani, and G. E. Karniadakis. Hidden fluid mechanics: A navier-stokes informed deep learning framework for assimilating flow visualization data. *arXiv:1808.04327*, 2018.

[44] B. Rajani, A. Kandasamy, and S. Majumdar. Numerical simulation of laminar flow past a circular cylinder. *Applied Mathematical Modelling*, 33(3):1228–1247, 2009.

[45] S. Rasp, P. D. Dueben, S. Scher, J. A. Weyn, S. Mouatadid, and N. Thuerey. Weatherbench: A benchmark dataset for data-driven weather forecasting. *arXiv:2002.00469*, 2020.

[46] D. Ray and J. S. Hesthaven. An artificial neural network as a troubled-cell indicator. *Journal of Computational Physics*, 367:166–191, Aug 2018.

[47] C. Rhie and W. L. Chow. Numerical study of the turbulent flow past an airfoil with trailing edge separation. *AIAA journal*, 21(11):1525–1532, 1983.

[48] A. Sanchez-Gonzalez, J. Godwin, T. Pfaff, R. Ying, J. Leskovec, and P. W. Battaglia. Learning to simulate complex physics with graph networks. *arXiv:2002.09405*, 2020.

[49] C. Schenck and D. Fox. Spnets: Differentiable fluid dynamics for deep neural networks. In *Conference on Robot Learning*, pages 317–335, 2018.

[50] S. S. Schoenholz and E. D. Cubuk. Jax, md: End-to-end differentiable, hardware accelerated, molecular dynamics in pure python. *arXiv:1912.04232*, 2019.

[51] S. Seo, C. Meng, and Y. Liu. Physics-aware difference graph networks for sparsely-observed dynamics. *International Conference on Learning Representations (ICLR)*, 2020.

[52] J. Sirignano and K. Spiliopoulos. Dgm: A deep learning algorithm for solving partial differential equations. *Journal of Computational Physics*, 375:1339–1364, 2018.

[53] T. F. Stocker, D. Qin, G.-K. Plattner, M. Tignor, S. K. Allen, J. Boschung, A. Nauels, Y. Xia, V. Bex, P. M. Midgley, et al. Climate change 2013: The physical science basis. *Contribution of working group I to the fifth assessment report of the intergovernmental panel on climate change*, 1535, 2013.

[54] I. Sutskever, O. Vinyals, and Q. V. Le. Sequence to sequence learning with neural networks. In *Advances in neural information processing systems*, pages 3104–3112, 2014.

[55] K. E. Taylor, R. J. Stouffer, and G. A. Meehl. An overview of cmip5 and the experiment design. *Bulletin of the American Meteorological Society*, 93(4):485–498, 2012.

[56] N. Thuerey, K. Weißenow, L. Prantl, and X. Hu. Deep learning methods for reynolds-averaged navier–stokes simulations of airfoil flows. *AIAA Journal*, 58(1):25–36, 2020.

[57] J. Tompson, K. Schlachter, P. Sprechmann, and K. Perlin. Accelerating eulerian fluid simulation with convolutional networks. In *Proceedings of Machine Learning Research*, pages 3424–3433, 2017.

[58] M. Toussaint, K. Allen, K. Smith, and J. B. Tenenbaum. Differentiable physics and stable modes for tool-use and manipulation planning. In *Robotics: Science and Systems*, 2018.

[59] B. D. Tracey, K. Duraisamy, and J. J. Alonso. A machine learning strategy to assist turbulence model development. In *AIAA aerospace sciences meeting*, page 1287, 2015.

[60] K. Um, X. Hu, and N. Thuerey. Liquid splash modeling with neural networks. *Computer Graphics Forum*, 37(8):171–182, Dec. 2018.

[61] B. Ummenhofer, L. Prantl, N. Thuerey, and V. Koltun. Lagrangian fluid simulation with continuous convolutions. In *International Conference on Learning Representations*, 2020.

[62] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin. Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008, 2017.

[63] W. Wang, S. Axelrod, and R. Gómez-Bombarelli. Differentiable molecular simulations for control and learning. *arXiv:2003.00868*, 2020.

[64] N. Watters, D. Zoran, T. Weber, P. Battaglia, R. Pascanu, and A. Tacchetti. Visual interaction networks: Learning a physics simulator from video. In *Advances in neural information processing systems*, 2017.

[65] E. Weinan. A proposal on machine learning via dynamical systems. *Communications in Mathematics and Statistics*, 5(1):1–11, 2017.

[66] Y. Xie, E. Franz, M. Chu, and N. Thuerey. tempogan: A temporally coherent, volumetric gan for super-resolution fluid flow. *ACM Transactions on Graphics (TOG)*, 37(4):1–15, 2018.

[67] Y. Zhang, W. J. Sung, and D. N. Mavris. Application of convolutional neural network to predict airfoil lift coefficient. *Structures, Structural Dynamics, and Materials Conference*, 2018.

phiflow: A Differentiable PDE Solving Framework for Deep Learning via Physical Simulations

Philipp Holl
Technical University of Munich
philipp.holl@tum.de

Vladlen Koltun
Intel Labs
vladlen.koltun@intel.com

Kiwon Um
LTCI, Telecom Paris, IP Paris
kiwon.um@telecom-paris.fr

Nils Thuerey
Technical University of Munich
nils.thuerey@tum.de

1 Introduction

Understanding physical environments is a key requirement for machine learning applications such as autonomous agents and robots [8, 1]. It is typically of vital importance to not only understand the unperturbed physical behavior but also anticipate how the environment reacts to an agent interacting with it [15, 6]. We consider partial differential equations (PDEs) as the most fundamental description of physical systems. The language of PDEs is general enough to describe every physical theory, from quantum mechanics and general relativity to turbulent flows [14]. Existing machine learning methods that deal with agents learning to interact with their environments have often focused on reinforcement learning [11, 5], but for high-dimensional environments, the computational cost of exploring the state space puts severe limits on the number of interaction parameters with which the agent can influence the physical system [9].

Meanwhile, progress has been made in utilizing differentiable solvers to find solutions to high-dimensional optimization problems [15, 4, 13]. Yet existing methods are still computationally expensive and thus limited to short time frames. We combine differentiable physics with deep learning to represent solution manifolds rather than computing single solutions via optimization. In this way, trained models can interact with a physical environment using a large number of interaction parameters, and inference times are orders of magnitude faster than with classic optimization algorithms. Here the use of differentiable physics is key for a robust learning of the complex spaces of behavior encoded by the model PDEs.

In this context, we present *phiflow* (`https://github.com/tum-pbs/PhiFlow`), a fully differentiable Eulerian PDE framework that provides operators and solvers for a large class of PDEs with analytic gradients. By fully integrating the numerical solver into the training process, neural networks (NNs) can, e.g., learn to reduce numerical errors of PDE solvers, and to optimally control a physical system given an initial state and a target state. We show the capabilities of *phiflow* with a wide range of correction and control tasks for various advection-diffusion type PDEs, and demonstrate that long time frames can be handled via a specialized architecture and evaluation scheme that separates the learning of physical behavior for different time scales.

2 Differentiable PDE solvers

Let $\boldsymbol{u}(\boldsymbol{x}, t)$ be described by a PDE that can be explicitly solved forward in time, i.e. time and space derivatives do not mix. The PDE can then be written as

$$\frac{\partial \boldsymbol{u}}{\partial t} = \mathcal{P}\left(\boldsymbol{u}, \frac{\partial \boldsymbol{u}}{\partial \boldsymbol{x}}, \frac{\partial^2 \boldsymbol{u}}{\partial \boldsymbol{x}^2}, ..., \boldsymbol{y}(t)\right) \tag{1}$$

where $\mathcal{P}$ models the physical behavior of the system and $\boldsymbol{y}(t)$ denotes any external factors that can influence the system. A classic solver can move the system forward in time via Euler steps:

$$\boldsymbol{u}(t_{i+1}) = \text{Solver}[\boldsymbol{u}(t_i), \boldsymbol{y}(t_i)] = \boldsymbol{u}(t_i) + \Delta t \cdot \mathcal{P}\left(\boldsymbol{u}(t_i), ..., \boldsymbol{y}(t_i)\right) \tag{2}$$

The square brackets indicate that Solver is a functional rather than a function, i.e. it takes full fields as input. Each step moves the system forward by a time increment Δt. Repeated execution produces a trajectory $u(t)$ that is a solution to the PDE.

When discretizing this formulation for time advancement directly it is not well-suited to solve optimization problems, since gradients can only be approximated by finite differencing in a regular forward solver. For high-dimensional or continuous systems, this method becomes computationally expensive because a full trajectory needs to be computed for each optimizable parameter. Differentiable solvers resolve this issue by solving the adjoint problem [12, 10] via analytic derivatives. The adjoint problem computes the same mathematical expressions while working with lower-dimensional vectors. A differentiable solver can efficiently compute the derivatives with respect to any of its inputs, i.e. $\partial \boldsymbol{u}(t_{i+1})/\partial \boldsymbol{u}(t_i)$ and $\partial \boldsymbol{u}(t_{i+1})/\partial \boldsymbol{y}(t_i)$. This allows for gradient-based optimization of inputs or control parameters of the simulation over an arbitrary number of time steps. The adjoint method is also used by most machine learning frameworks, where it is more commonly known as reverse mode differentiation [16, 3].

We make use of this analogy to realize *phiflow*, a differentiable PDE solver as a set of mathematical operations within a deep learning framework. We focus on Eulerian rather than Lagrangian methods since they are widely used for a large class of PDEs [14]. All solver operations are implemented in a differentiable manner, i.e. the automatic differentiation tools can chain the derivatives of these operations with built-in machine learning operations to build analytic derivatives for any combination of operations, thus enabling end-to-end training. This toolkit of operations enables the solver to handle a large class of PDEs, including the incompressible Navier-Stokes equations.

3 Learning solver interactions

Assuming the physical behavior $\mathcal{P}$ is described by a PDE as in Eq. (1), we add a force term $\boldsymbol{F}(t)$, which can be seen as a "correction" or "control" that allows the model to interact with the system:

$$\frac{\partial \boldsymbol{u}}{\partial t} = \mathcal{P}\left(\boldsymbol{u}, \frac{\partial \boldsymbol{u}}{\partial \boldsymbol{x}}, \frac{\partial^2 \boldsymbol{u}}{\partial \boldsymbol{x}^2}, ...\right) + \boldsymbol{F}(t) \tag{3}$$

While the evolution of the complete state $\boldsymbol{u}$ is determined by the above equation, we allow some parts of $\boldsymbol{u}$ to be hidden for the forcing. This restriction reflects the fact that it is often not possible to observe the full state of a physical system. When considering a cloud of smoke, for example, the smoke density might be observable while the velocity field cannot be seen directly. Mathematically, we model this restriction by decomposing $\boldsymbol{u}$ into an observable part $\boldsymbol{o}$ and a hidden part $\boldsymbol{h}$ with $\boldsymbol{u} = \boldsymbol{o}(\boldsymbol{u}) \otimes \boldsymbol{h}(\boldsymbol{u})$. Here, $\otimes$ denotes the tensor product, adding all components of the states. The hidden part can include spatial regions of some fields as well as entire fields.

Using the above notation, we define the control task as follows. An initial observable state $\boldsymbol{o}_0$ of the PDE as well as a target state $\boldsymbol{o}^*$ are given. We are interested in a reconstructed trajectory $\boldsymbol{u}^r(t)$ that matches these states at t_0 and t_*, i.e. $\boldsymbol{o}_0 = \boldsymbol{o}(\boldsymbol{u}^r(t_0)), \boldsymbol{o}^* = \boldsymbol{o}(\boldsymbol{u}^r(t_*))$, and requires the least amount of effort over the whole time span. I.e., we aim for minimizing the forces to be applied in terms of their magnitude with:

$$L_{\boldsymbol{F}}[\boldsymbol{u}(t)] = \int_{t_0}^{t_*} |\boldsymbol{F}_{\boldsymbol{u}}(t)|^2 \, dt \tag{4}$$

Taking discrete time steps Δt, the reconstructed trajectory $\boldsymbol{u}^r$ is a sequence of $n = (t_* - t_0)/\Delta t$ states. This problem definition is portrayed in Fig. 1. An initial observation $\boldsymbol{o}_0$ and target observation $\boldsymbol{o}_*$ are given (a). The goal is to reconstruct a trajectory $\boldsymbol{u}^r$ that moves from $\boldsymbol{o}_0$ to $\boldsymbol{o}_*$ in the state space and requires as little force as possible, as shown in (b). The grey lines represent the unperturbed evolution of the physical system. The amount of applied force corresponds to how far the trajectory deviates from the natural evolution in this picture.

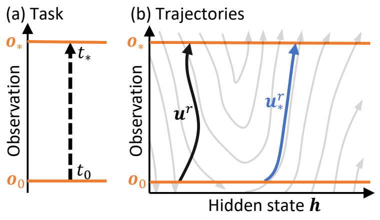


Figure 1: Possible trajectories.

When an observable dimension cannot be controlled directly, there may not exist any trajectory $\boldsymbol{u}(t)$ that matches both $\boldsymbol{o}_0$ and $\boldsymbol{o}^*$. This can stem from either physical constraints or numerical limitations.

In these cases, we settle for an approximation of $\boldsymbol{o}^*$. To measure the quality of the approximation of the target, we define an observation loss $L^*_{\boldsymbol{o}}$. The form of this loss can be chosen to fit the problem. For our experiments we use the filtered L_2 distance between target and reconstruction:

$$L^*_{\boldsymbol{o}}(\boldsymbol{u}(t_*)) = |B_r(\boldsymbol{o}^*) - B_r\left(\boldsymbol{o}(\boldsymbol{u}(t_*))\right)|^2 \tag{5}$$

where B_r denotes a spatial blur function with a fixed, problem-dependent radius $r \geq 0$. We combine Eqs. 4 and 5 into the objective loss function

$$L[u(t)] = \alpha \cdot L_{\boldsymbol{F}}[\boldsymbol{u}(t)] + \beta \cdot L^*_{\boldsymbol{o}}(u(t_*)), \tag{6}$$

with $\alpha, \beta > 0$. Since our solver is differentiable, L can be used directly to optimize a machine learning model such as a neural network that models $\boldsymbol{u}^r(t), \boldsymbol{o}_*, t \to \boldsymbol{F}(t)$ with weights $\boldsymbol{w}$. We call this network the control force estimator (CFE).

For a sequence of n frames, $L[\boldsymbol{u}(t)]$ depends on all n states of the trajectory $\boldsymbol{u}(t)$. Thus, for recurrent end-to-end training, n linked copies of the network need to be chained together. When inferring the force, this results in a CFE chain, shown in Fig. 2, that alternates between network and solver execution. When using a CFE chain, the complete sequence needs to be run forward and backward for each optimization step of the model. This is not only slow, it also means that gradients are passed through a potentially long chain of highly non-linear simulation steps. When the reconstruction u^r is close to an optimal trajectory, this is not a problem since the gradients $\Delta\boldsymbol{u}^r$ are small and the operations executed by the solver are differentiable by construction. The solver can therefore be locally approximated by a first-order polynomial and the gradients can be safely backpropagated. For large $\Delta\boldsymbol{u}^r$, such as at the beginning of training, this approximation breaks down, causing the gradients to become highly unstable while passing through the chain. In some cases below, we employ a second model, which predicts the observable state $\boldsymbol{o}^p\left((t_i + t_j)/2\right)$ given two observations. We refer to this model as the observation predictor (OP) [7].

Note that while some existing approaches rely on a continuous time formulation, e.g. for incorporating ODEs [3], we instead make use of a given time discretziation with a chosen temporal step size. While this requires storing the intermediate states of the simulated system, it allows for using numerical methods that are suitable to handle the specifics of a PDE under consideration. E.g., tailored time stepping schemes or specialized and efficient solvers can be integrated into the learning process in this way. E.g., we make use of these capabilities for the pressure calculation within a Navier-Stokes solver.

As this workshop paper can only provide a very brief summary of the different *phiflow* applications, the de-anonmyized version is this paper will refer to the *phiflow* source code and corresponding full papers.

4 Results

Here, we focus on *phiflow* applications in terms of two-dimensional fluid dynamic problems, which are highly challenging due to the complexities on the governing Navier-Stokes equations [2] for the velocity field $\boldsymbol{v}$,

$$\mathcal{P}(\boldsymbol{v}, \nabla\boldsymbol{v}) = -\boldsymbol{v} \cdot \nabla\boldsymbol{v} + \nu\nabla^2\boldsymbol{v} + \nabla p, \tag{7}$$

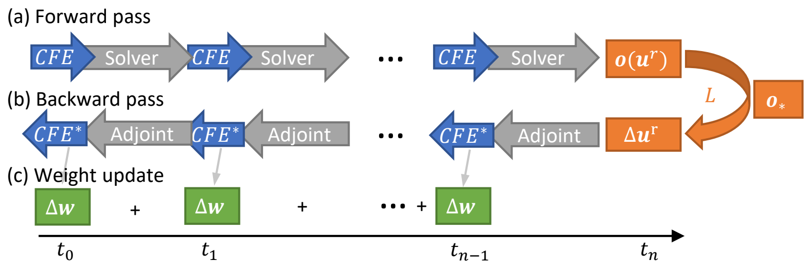


Figure 2: A chained force prediction network: (a) The forward pass reconstructs a trajectory by alternating between force estimation and solver execution. (b) For backpropagation, the adjoint problem is computed. (c) Weight updates are accumulated and applied to the model.

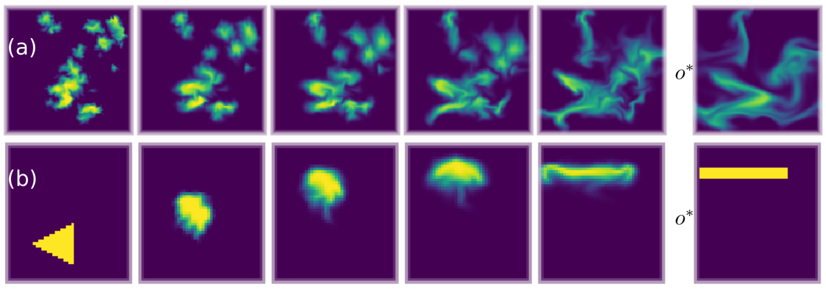


Figure 3: Example reconstructed trajectory from (a) the natural flow test set and (b) the shape test set. The initial state is shown on the far left, the target state $\boldsymbol{o}^*$ is shown on the right. The optimization goal for the NN is to reach the target state given the constraints of the physical model with forces for a wide range of randomized source and target states.

Table 1: A comparison of methods in terms of final cost for (a) a natural flow setup and (b) shape transitions from Fig. 3. The initial distribution is sampled randomly and evolved to the target state.

Execution	Loss	a) Force $L_{\boldsymbol{F}}$	a) Obs. $L_{\boldsymbol{o}}^*$	b) Force $L_{\boldsymbol{F}}$	b) Obs. $L_{\boldsymbol{o}}^*$
Regular	Supervised	243 ± 11	1.53 ± 0.23	n/a	n/a
Regular	*phiflow*	22.6 ± 1.1	0.64 ± 0.08	89 ± 6	0.331 ± 0.134
Refined	*phiflow*	11.7 ± 0.6	0.88 ± 0.11	75 ± 4	0.126 ± 0.010

subject to the hard constraints $\nabla \cdot \boldsymbol{v} = 0$ and $\nabla \times p = 0$, where p denotes pressure and ν the viscosity. In addition, we consider a passive density ρ which moves with the fluid via $\partial\rho/\partial t = -\boldsymbol{v} \cdot \nabla\rho$. We set $\boldsymbol{v}$ to be hidden and ρ to be observable and allow forces to be applied to all of $\boldsymbol{v}$.

Example sequences for the control task on 128×128 domains are shown in Fig. 3 and a quantitative evaluation, averaged over 100 examples, is given in Tab. 1. While all divide-and-conquer methods manage to approximate the target state well, there are considerable differences in the amount of force applied. The supervised technique, denoted as *regular*, exerts significantly more force than the differentiable solver based methods, resulting in jittering reconstructions. A prediction refinement scheme (denoted as *refined*) re-evaluates predictions over the course of a sequence. This version produces the smoothest transitions, converging to about half the loss of the regular, non-refined variant. For comparison, we run a classic optimization with hierarchical shooting that computes solutions for single cases, and find that it requires 1500 iterations to compute a control function that our trained model infers almost instantly. In the accompanying publications, we also demonstrate that more indirect forms of control of systems such as a Navier-Stokes environments are possible.

Additionally, combining differentiable PDE solvers and deep learning can be leveraged to reduce numerical errors, by omitting the OP network, and prediciton a correction for each step of a sequence via a CFE network. This is demonstrated for a 3D case of incompressible unsteady wake flow in 4. While a traditional, supervised version fares poorly and becomes unstable (not shown), the SOL_{16} version (trained with 16 steps of differentiable physics) achieves stable rollouts for several hundred time steps and successfully corrects the numerical inaccuracies of the coarse discretization. It improves the numerical accuracy of the source (SRC) simulation by more than 22% across a wide range of configurations. This case also highlights the gains in performance that can be achieved with our method: while the deep learning-based hybrid solver with SOL_{16} took 13.3s on average for 100 time steps, a CPU-based reference simulation required 913.2s. A speed-up of more than 68×.

5 Conclusions

We have introduced the *phiflow* framework with a summary of selected results. They show that deep learning models in conjunction with a differentiable physics solver can successfully predict the behavior of complex physical models and learn to control and correct them. We believe that learning differentiable physics has significant potential to provide physical intuition for a wide range of systems that understand and interact with the real world.

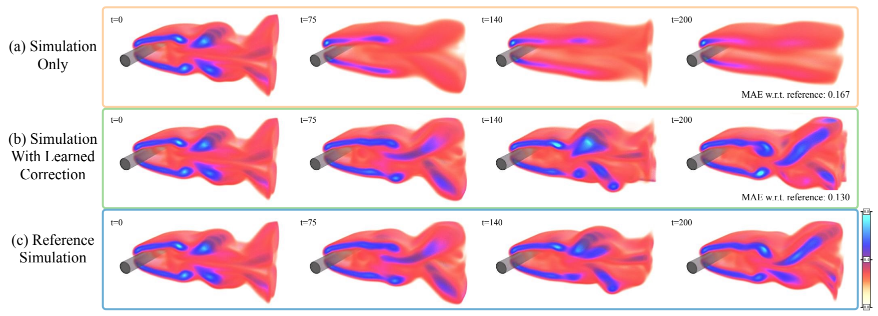


Figure 4: A 3D fluid problem, shown in terms of vorticity. From top to bottom: a) regular simulation, b) reference, c) regular simulation with learned corrector.

References

[1] Pulkit Agrawal, Ashvin V Nair, Pieter Abbeel, Jitendra Malik, and Sergey Levine. Learning to poke by poking: Experiential learning of intuitive physics. In *Advances in Neural Information Processing Systems*, 2016.

[2] G. K. Batchelor. *An Introduction to Fluid Dynamics*. Cambridge University Press, 1967.

[3] Tian Qi Chen, Yulia Rubanova, Jesse Bettencourt, and David K Duvenaud. Neural ordinary differential equations. In *Advances in Neural Information Processing Systems*, 2018.

[4] Filipe de Avila Belbute-Peres, Kevin Smith, Kelsey Allen, Josh Tenenbaum, and J Zico Kolter. End-to-end differentiable physics for learning and control. In *Advances in Neural Information Processing Systems*, 2018.

[5] Chelsea Finn, Ian Goodfellow, and Sergey Levine. Unsupervised learning for physical interaction through video prediction. In *Advances in Neural Information Processing Systems*, 2016.

[6] Nick Haber, Damian Mrowca, Li Fei-Fei, and Daniel LK Yamins. Learning to play with intrinsically-motivated self-aware agents. *arXiv:1802.07442*, 2018.

[7] Philipp Holl, Vladlen Koltun, and Nils Thuerey. Learning to control pdes with differentiable physics. *ICLR*, 2020.

[8] Leslie Pack Kaelbling, Michael L Littman, and Andrew W Moore. Reinforcement learning: A survey. *Journal of Artificial Intelligence Research*, 4:237–285, 1996.

[9] Timothy P Lillicrap, Jonathan J Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. *arXiv:1509.02971*, 2015.

[10] Antoine McNamara, Adrien Treuille, Zoran Popović, and Jos Stam. Fluid control using the adjoint method. *ACM Trans. Graph.*, 23(3), 2004.

[11] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *ICML*, 2016.

[12] Lev Semenovich Pontryagin. *Mathematical Theory of Optimal Processes*. John Wiley, 1962.

[13] Connor Schenck and Dieter Fox. SPNets: Differentiable fluid dynamics for deep neural networks. In *Conference on Robot Learning*, 2018.

[14] Gordon D Smith. *Numerical Solution of Partial Differential Equations: Finite Difference Methods*. Oxford University Press, 1985.

[15] Marc Toussaint, Kelsey Allen, Kevin Smith, and Joshua B Tenenbaum. Differentiable physics and stable modes for tool-use and manipulation planning. In *Robotics: Science and Systems*, 2018.

[16] Paul J Werbos. Backwards differentiation in AD and neural nets: Past links and new opportunities. In *Automatic Differentiation: Applications, Theory, and Implementations*, pages 15–34. Springer, 2006.