Machine Learning for Simulations

by

Karl Otness

A dissertation submitted in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

Department of Computer Science

New York University

May, 2025

Professor Joan Bruna

Professor Benjamin Peherstorfer

Acknowledgments

My first thanks go to my advisors Professors Joan Bruna and Benjamin Peherstorfer, who provided key insights and guidance on navigating the research process, and were of critical importance in developing the projects discussed here. I am deeply grateful for their help in identifying promising research directions and for their assistance in framing our results to best communicate the relevance of our work to the wider research community.

I am indebted to my collaborators, Professors Laure Zanna, Daniele Panozzo, and Denis Zorin, as well as Dr. Jiequn Han, Arvi Gjoka, and Dr. Teseo Schneier. These projects would not have been possible without their hard work and contributions.

I also benefitted significantly from discussions with many of my fellow students and other researchers. This was especially true of members of the M2LInES climate research collaboration, who helped me significantly as I was beginning work targeting climate modeling applications. Among many others, Pavel Perezhogin, Chris Pedersen, Adam Subel, Fabrizio Falasca, Dhruv Balwada, and Sara Shamekh pointed me to useful resources and helped me gain an understanding of this research area. I owe a great deal to their expertise.

I am also grateful to my defense committee—my advisors, as well as Professor Laure Zanna, Professor Carlos Fernandez-Granda, and Dr. Jiequn Han—for their willingness to serve on my committee and review this work.

Abstract

Computational modeling of physical systems is a core task of scientific computing. Machine learning methods can extend traditional approaches to modeling partial differential equations and hold the potential to simplify the modeling process and improve simulation accuracy and performance. In this thesis we explore the use of neural networks to learn the behavior of systems from data. We evaluate the performance-accuracy tradeoffs involved in their use as emulators, and use insights gained here to explore a specific application to learning subgrid parameterizations for climate models in particular. For this task we propose two novel techniques to improve the accuracy and stability of the learned parameterizations by tailoring architectures to incorporate favorable inductive biases, and by augmenting training data to encourage stability.

Table of Contents

List of Figures

List of Tables

List of Appendices

Chapter 1

Introduction

Modern machine learning and deep learning methods have had a profound impact across a wide range of application areas, leading to advancements in the ability of computer systems to derive insights about the world from data. Numerical simulations likewise distill real-world physics and dynamics into a mathematical form, enabling computational experiments that reflect the behavior of real-world systems. However, developing these simulations has traditionally required not only observations of the target system, but also human insight to identify the underlying patterns of behavior. Just as they have done in other areas, machine learning methods promise to enable new approaches to simulation problems by learning directly from data.

Specific applications making use of physical simulations are diverse and come with their own constraints and specific goals [13]. Even so, at a high level we distinguish two broad goals for the use of machine learning and neural networks in simulation tasks: (1) simplifying the modeling process and (2) improving simulation accuracy and performance. Many applications may target some combination of these goals.

Machine learning can simplify the modeling process by learning system dynamics directly from data. In this case, a machine learning method can be used in place of physics modeling to learn either how to evolve the state of a full system, or potentially only some subset of unmodeled

dynamics. With sufficient available data, this provides a shortcut through the model development process in cases where the target system is not fully understood or where the best theoretical approach may not be clear. In such applications, the evaluation may more heavily emphasize the accuracy of the learned model's behavior, and place less emphasis on the computational costs involved in its end use.

In many simulation applications, accuracy and performance are in tension. Improved accuracy can be achieved with more complex models, finer-resolution states, or smaller finer-grained time-stepping. If carefully applied, machine learning methods can alter the tradeoffs between these two goals, allowing greater accuracy to be achieved at a lower cost. However, these applications are challenging, particularly when compared against mature traditional non-learning approaches. Many such applications pursue a hybrid approach where a learned component is used to improve the accuracy of a simulation, after simplifying its dynamics to realize performance savings.

In this thesis we will present the results of three projects working in this area. The first, presented in Chapter 2, investigates the challenges involved in training neural networks to simulate entire dynamical systems and compares their performance and accuracy against standard numerical schemes. The insights gained from this work highlight the effects of architectural choices on the accuracy of the learned simulation, as well as a number of challenges in applying neural networks to simulation tasks. In the next two projects, detailed in Chapter 4 and Chapter 5, respectively, we explore ways to improve the application of neural networks to climate model subgrid parameterizations. Applying machine learning to this task is currently an active area of research, seeking to use neural networks to improve the accuracy of climate models by representing the dynamics of unresolved processes and augmenting existing manually-developed approaches. Further details on this application are included in Chapter 3.

Following an introduction to a few preliminary topics, further discussion of these projects and a summary of the contributions of this thesis are included below.

1.1 Preliminaries

Next, we introduce background information for applying machine learning to partial differential equations (PDEs). In the chapters that follow, we will explore several distinct target applications including learning to reproduce the dynamics of entire systems from observations of their behavior, and uses of neural networks to improve the accuracy of existing simulations by approximating the contribution of unresolved scales. Discussion of each of these particular applications will be accompanied by further background, but we provide a general overview in this section.

1.1.1 Discretizing PDEs

In this work we consider numerical simulation tasks, originally specified as differential equations. Generally these are specified as PDEs. As an illustration, consider:

$$\frac{\partial u}{\partial t} = \mathcal{L}u + \mathrm{f} \tag{1.1}$$

where $u(t, \omega) \in \mathbb{R}^{D_2}$ is a continuous function defined on a time interval and spatial domain $\Omega \subset \mathbb{R}^{D_1}$, f is a forcing function over the same domain, and $\mathcal{L}$ is a differential operator which may be nonlinear [37]. In the remainder of this thesis, we will be concerned with initial value problems where we are also given an initial state for u at time $t = 0$. In our work, we will consider *autonomous* systems, in which the updates $\mathcal{L}u$ and forcings f depend only on the current state u and have no time-dependence.

For our purposes, we require the ability to compute solutions to the PDEs. To accomplish this, we will apply the common scheme of discretizing the spatial domain Ω by representing our solutions on a discrete grid, or approximating u on this domain with respect to a chosen basis; handling any spatial derivatives; and then proceeding to update the states through time using a choice of numerical integration scheme [37, 96].

One common choice of representation is to discretize the system with respect to a regular spatial grid, producing fixed spatial coordinates $\omega_1, \ldots, \omega_n$ with regular spacing δ_x along each grid axis. Any spatial derivatives required by $\mathcal{L}$ can be handled by finite differences in the spatial grid, applying any standard stencil. This accomplished, we are left with an ordinary differential equation (ODE) system where we define a vector of system states $x(t) = [u(t, \omega_1), \ldots, u(t, \omega_n)] \in \mathbb{R}^n$. This leaves us with a need to time step our ODE, solely a function of continuous time t, with right hand side function f (including the forcing f):

$$x(t) \in \mathbb{R}^n \tag{1.2}$$

$$\dot{x}(t) \triangleq \frac{\partial x}{\partial t} = f(x(t)) \tag{1.3}$$

Alternatively, if u is periodic, we can take a spectral approach, representing it as a Fourier series. For example in one dimension, on a periodic domain $\Omega = [0, R]$, we can truncate the Fourier series and obtain:

$$u(t, \omega) \approx \sum_{k=-n/2+1}^{n/2} \hat{x}_k(t) e^{i2\pi \frac{k}{R}\omega} \tag{1.4}$$

which represents the state of the system as a vector $\hat{x} \in \mathbb{C}^n$. We can then process any spatial derivatives in $\mathcal{L}$ analytically using the Fourier series representation, and thus we again have an ODE system $\hat{x} \in \mathbb{C}^n$ over the Fourier coefficients:

$$\hat{x}(t) \in \mathbb{C}^n \tag{1.5}$$

$$\dot{\hat{x}}(t) \triangleq \frac{\partial \hat{x}}{\partial t} = \hat{f}(\hat{x}(t)) \tag{1.6}$$

In practice, we will take the discrete Fourier transform of an existing spatial grid representation of u to transform the system into spectral representation. It is also possible to mix calculations using spectral and real-space discretizations of a system for computational efficiency taking advantage of the Fast Fourier Transform and the convolution theorem (a "pseusospectral" approach) [104].

We will see an example of such a system in Chapter 3. Other methods to divide the domain Ω and process the PDE include finite element and volume methods which, while used internally in some test systems, will not be a major focus of the experiments and projects presented in this work [37, 104].

1.1.2 Time Integration

With the state representations discretized and spatial derivatives handled, in order to simulate this system, we select time steps t_i and apply a numerical integration scheme, starting from an initial condition $x(t_0)$. This procedure rolls out a trajectory of states approximating the evolution of the continuous time system, and we will generally denote these states with subscripts for the discrete time intervals: $x_i \approx x(t_i)$.

Numerical time integration can be carried out following many different schemes. Some particular dimensions of this decision are the selection of an explicit or implicit scheme, and the number of steps (single- or multi-step methods). Explicit schemes are those in which subsequent time steps are computed using only the state of previous time steps and which do not require solving an implicit equation. The simplest explicit scheme is the forward Euler method:

$$x_t = x_{t-1} + \delta_t f(x_{t-1}) \tag{1.7}$$

where δ_t is the chosen time step size $\delta_t = t_i - t_{i-1}$. The forward Euler method is a single-step method, and requires only the previous step x_{t-1} and a single evaluation of f, but other methods exist that compute multiple sub-steps or that make use of multiple steps of history. For example, with third-order Adams-Bashforth method:

$$x_t = x_{t-1} + \delta_t\Big(\frac{23}{12}f(x_{t-1}) - \frac{16}{12}f(x_{t-2}) + \frac{5}{12}f(x_{t-3})\Big). \tag{1.8}$$

a history of the two previous evaluations of f can be kept so that each new time step requires only one new evaluation for $f(x_{t-1})$.

These multi-step methods achieve a higher order of convergence, meaning the trajectory they generate converges to the continuous time flow of f at a faster rate. For example, the third-order Adams-Bashforth method converges at a rate $O(\delta_t^4)$ while the first-order Euler method achieves $O(\delta_t^2)$, requiring smaller time steps to achieve equivalent error [37, 96].

Some multistep methods take intermediate time steps of different lengths *between* times t_{i-1} and t_i and will require multiple evaluations of f at each time step which cannot be re-used for future steps. These methods avoid bootstrapping issues (i.e. the missing evaluations for "previous" steps at time t_0), but come at a higher computational cost.

Implicit schemes determine future time steps as the solutions to implicit equations. For example, the backward or implicit Euler method:

$$x_t - \delta_t f(x_t) = x_{t-1}. \tag{1.9}$$

For scenarios in which the right-hand side function f may be partially or entirely composed of a neural network, implicit schemes would require solving the thus nonlinear function f. In most of the experiments described in this work, we will make use of explicit schemes to avoid this difficulty. Combining f with a numerical integration scheme gives us a mapping T_f which advances the state of the resulting discrete dynamical system:

$$x_{t+1} = T_f(x_t). \tag{1.10}$$

Several other integration schemes will be introduced in Section 2.5.

1.1.3 Learning Simulations

For these types of problems, we distinguish several broad categories of tasks that a neural network could be trained to perform. First, a network might be trained to emulate the full system. A neural network f_θ might be trained to approximate the right-hand side function f:

$$f_\theta(x_t) \approx f(x_t) \tag{1.11}$$

which is then combined with a numerical integrator in place of f, to produce a new operator which substitutes for T_f when advancing the system states. We refer to this task as "derivative prediction." Similarly, we might also train f_θ to carry out the numerical integration process as well, replacing both the dynamics f *and* the numerical integration scheme. That is,

$$f_\theta(x_t) \approx x_{t+1}. \tag{1.12}$$

In this case, when rolling out a state using the trained networks, we no longer combine f_θ with a separate numerical integrator and we must necessarily fix a time step size δ_t which cannot be changed after training. We call this type of task "step prediction."

In both of the above tasks, the system's original dynamics, which follow f, are entirely replaced by the trained network f_θ. This network is sometimes referred to as an emulator of the original system. We also consider hybrid applications in which the original dynamics f and a trained network are combined. In these applications we typically have ground-truth dynamics which follow f, and modified dynamics following some function $\bar{f}$. These dynamics may be different as a result of some model reduction process, and often evolve states $\bar{x}_t$ with a coarser resolution on $\mathbb{R}^{\bar{n}}$. In these cases we may wish to patch dynamics to approximate as closely as possible those of

the original system f, training a network f_θ to provide this correction.

$$\bar{f}(\bar{x}_t) + f_\theta(\bar{x}_t) \approx \overline{f(x_t)} \tag{1.13}$$

Specific applications of this type will be further introduced in Chapter 3.

In training networks for these tasks we consider two types of training: online and offline. In offline training, the network is trained on separate *snapshots* of simulation states: that is, observations of a reference trajectory, produced as described in the previous sections. Any temporal connection between them is lost and the network is not exposed to accumulated errors during a recurrent rollout. In online training, by contrast, the network is combined, if necessary, with a numerical integration scheme and used recurrently to produce a short trajectory of steps, and backpropagation is then done through time. In online training the network does observe accumulated errors and is exposed to interactions with the time-stepping scheme, generally incurring a higher training cost. However, in this case all components of the dynamical system operator T_f beyond the neural network must also support automatic differentiation. In tasks which involve existing production models this is not always possible.

Training networks for such tasks usually involves a library of reference trajectories consisting of states x_t and any other necessary values such as references values for $f(x_t)$ in the case of derivative prediction, or errors to be corrected in hybrid applications $\overline{f(x_t)} - \bar{f}(\bar{x}_t)$. The usual set of training losses may be used for supervised training, including mean squared error (MSE) or ℓ_2. Hybrid model tasks may use these losses as well, but the quality of trajectories produced using a network is often evaluated using other statistics depending on the requirements of the particular application.

1.2 Summary of Contributions

In this thesis we present the results of three research projects pursued with collaborators. These projects explore several applications of deep learning methods to simulation tasks, including applications to real world problems arising in climate models.

In Chapter 2 we explore the use of machine learning to derive simulation dynamics directly from data. To do this we developed a set of PDE benchmark tasks which are usable by other researchers. We then test a variety of common neural network architectures and other machine learning methods on these problems. Our results here contribute not only the reusable set of benchmark problems and baseline results for comparisons, but our thorough analysis of our own test results and insights drawn from our analysis. In particular, we observed the benefit of selecting neural network architectures well suited to the target task, as well as issues with stability in some systems—a problem which recurs frequently in this area. This project also contributes one of the only careful measurements of the accuracy-performance tradeoff with learning methods and compares it against the same with traditional numerical integration methods. That is, we do not measure only the runtime cost of a particular integration scheme or network, but the runtime cost required to achieve a comparable level of error. This is an important consideration when evaluating neural networks and had not previously been explored in related work and remains uncommon in the literature.

Next, using some of the insights gained from this first project, we consider hybrid applications of neural networks to simulation problems. In particular we examine applications of machine learning to the subgrid parameterization problem which arises in climate modeling as well as in other applications. We contribute two novel methods for improving the quality of these learned parameterizations. The first, detailed in Chapter 4, tailors the neural network architecture to take advantage of the inherent multiscale structure in the subgrid parameterization problem. In Chapter 5 we describe a second approach which adapts the training process to increase the stability

of learned parameterizations. This method gives some of the benefits of online training—namely, some exposure to errors accumulated over time—but without imposing any requirements on the target simulation beyond those already required for evaluation. In particular, it does not require the ability to compute gradients through simulation time steps.

The results presented here provide insights that can guide future developments in this field and methods to improve the performance of learned subgrid parameterizations which are widely applicable.

Chapter 2

Learning Simulations from Data

In this chapter we describe a project developing a benchmark composed of physical systems with varying state dimensions, initial condition ranges, and dynamics. These systems are intended to be used to train neural networks to emulate a system: that is, to learn the dynamics of the system wholesale from observation snapshots. Learning PDE tasks is a rich area of application for neural network models with a wide range of potential applications [13]. However, this diversity of application tasks and of applicable machine learning methods makes comparing the performance of proposed methods difficult. In general, each work in this area tests on different systems, applies different methods, and measures performance with different metrics. In an attempt to unify some of this work, we propose a set of physical systems which is representative of some of the range of systems often studied. Using this set, we evaluate the performance of a set of baseline methods including several common neural network architectures, other non-neural network learning methods, and a range of standard numerical integration schemes.

In our analysis we pay particular attention to the computational efficiency of each approach, measuring the often significant accuracy and performance penalties that arise when using a neural network as a surrogate model, as well as models' abilities to generalize to unseen conditions. The full source code and raw results from each experiment are available online for further analysis

and testing. We also work to distill some insights from our efforts which may provide some guidance for future works in this area. The project described in this chapter is "An Extensible Benchmark Suite for Learning to Simulate Physical Systems" which was completed alongside several collaborators: Arvi Gjoka, Joan Bruna, Daniele Panozzo, Benjamin Peherstorfer, Teseo Schneider, and Denis Zorin [64].

Since the publication of this work, the application of deep learning to PDE problems has continued to be a very active area of research. Continuing in a similar vein to our work, several other benchmark sets have been developed and proposed for wider application. These include general application benchmarks with additional and alternative PDE systems and added network baselines such as Physics-Informed Neural Networks and Fourier Neural Operators [52, 74, 97] and others which specialize in particular applications such as airfoil design or climate model parameterization development and include, importantly, proposed metrics to evaluate performance [9, 109]. Recently, research attention has also turned to developing foundation models, capable of handling multiple physics and providing a starting point for further downstream specialization [59]. This has led to the proposal of larger benchmark sets incorporating a wider range of problem types [62]. In some respects this variety illustrates the challenges involved in targeting PDE tasks and the difficulty of scaling benchmark datasets of this type. For example, the largest benchmark set, compiled at significant cost and effort, includes approximately 20 types of systems. Even though these are drawn from problems studied across a range of research disciplines, there are still certainly additional tasks that could be included, and other variations that could be reflected (i.e. non-uniform grids, etc.), making it difficult to quantify or otherwise assess coverage of potential applications.

Despite such challenges, these datasets are a valuable resource and simplify the development and testing of new approaches to learning PDE dynamics. It is in this spirit that we proposed our own set of benchmark tasks. In this project, we attempt to cover a range of system dynamics, including varying simulation structures and complexity of spatial interactions within states, as

well as examining the impact of varying quantities of available training data, the range of initial conditions, generalization to out-of-distribution states, and the tradeoff between computational cost and accuracy of generated trajectories. This last comparison was conducted with particular care and attention. We evaluated not only the per-step cost of several traditional baseline integration schemes and machine learning methods, but also the time cost to achieve a particular level of error. To the best of our knowledge, this error-matched timing comparison remains unique.

2.1 Project Background

Computational modeling of physical systems is a core task of scientific computing. Standard methods rely on discretizations of explicit models typically given in the form of partial differential equations (PDEs). Machine learning techniques can extend these techniques in a number of ways. In some cases, a closed system of analytic equations relating all variables may not be available (e.g., a constitutive relation for a material may not be known). In other cases, while a full analytic description of a system is available, a traditional solution may be too costly (e.g., turbulence) or can be sped up substantially using data-driven reduced-order models. However, despite promising results, a successful adoption of these data-driven approaches into scientific computing pipelines requires a solid and exhaustive assessment of their performance—a challenging task given the diversity of physical systems, corresponding data-driven approaches, and the lack of standardized sets of problems, comparison protocols, and metrics.

We focus on the setting where the physical model is unavailable during training, mimicking situations in computational science and engineering with ample data and a lack of models. One can generally distinguish two different flavors of physical simulation with different associated computational cost: those that map a high-dimensional state space into another high-dimensional space (as in temporal integration schemes, mapping the state of the system at one time step to the next), or from a high-dimensional input space to a lower-dimensional output (as in surrogate

models, mapping the initial conditions to a functional of the solution). While this distinction also applies to data-driven approaches, another critical aspect emerges from the choice of input data distribution. We identify two extremes: the *narrow* data regime, where initial conditions are sampled from a low-dimensional manifold (even within a high-dimensional state space), and the *wide* regime, where initial conditions span a truly high-dimensional space. As could be expected, narrow data regimes define an easier prediction task where data-driven methods can potentially 'bypass the physics,' whereas wide regimes require models with enough encoded physical priors in order to beat the curse of dimensionality. Therefore, such choice of data distribution is a critical component of any data-driven physical simulation benchmark.

In this work, we introduce an extensible benchmark suite, including: **(1)** an extensible set of simple, yet representative, physical models with a range of training and evaluation (test) setups, as well as reference, high-accuracy numerical solutions to benchmark data-driven methods, **(2)** reference implementations of traditional time integration schemes, which are used as baselines for evaluation, and **(3)** implementations of widely used data-driven methods, including physics-agnostic multi-layer perceptrons (MLPs), convolutional neural networks (CNNs), kernel machines and non-parametric nearest neighbors. Our benchmark suite is modular, permitting extensions with limited code changes, and captures both 'narrow' and 'wide' regimes by appropriately parametrizing the set of initial conditions.

Our analysis reveals two important conclusions. First, even in the simplest physical models, current data-driven pipelines, while providing qualitatively acceptable solutions, are quantitatively far from directly numerically integrating physical models, and this performance gap appears unfeasible to close by merely scaling up the models and/or the dataset size. In other words, the cost of ignoring the physics is high, even for the simplest physics, and cannot in general be compensated by data, matching insights that have been obtained in other scientific computing settings [16, 106]. Next, and more importantly, our simple L^2-based nearest neighbor regressor is used to calibrate how 'narrow' the learning task is. Our finding is that even for seemingly complex

systems, such as the incompressible Navier-Stokes systems, such a naive predictor outperforms most deep-learning-based models in the narrow regime—thus providing a simple calibration of the true difficulty of the simulation task, that we advocate should be present in every future evaluation.

2.2 Related Work

Machine learning is used in physical simulation in a number of interrelated ways. Some important uses include reduced-order/surrogate modeling, learning constitutive models or more generally compact analytic representations from data. A unifying theme of these applications of machine learning is automatic construction of parametric models capable of reproducing the behavior of physical systems for a sufficiently broad range of initial data, boundary conditions and other system parameters. The purpose of these representations varies from acceleration (e.g., surrogate machine learning models are used to accelerate optimization), to automatic construction of multiscale models (learning macroscopic constitutive laws from microscopic simulation), to inferring compact descriptions of unknown representations from experimental data.

The purpose of our proposed benchmarks is to enable comparisons of different learning-based methods in terms of their accuracy and efficiency. We briefly review two streams of learning methods for physical systems. **(1)** One line of work aims to understand how neural networks can be structured and trained to reproduce known physical system behavior, with the goal of designing general methods applicable in a variety of settings [6, 7, 12, 15, 28, 31, 56, 74, 75, 80, 81, 98, 99, 105]. Our benchmark cases fit primarily into this category. **(2)** Another line of research aims to develop a variety of techniques to accelerate solving PDEs. Typically, these methods are developed for specific PDEs and a specific restricted range of problems: for example, fluid dynamics problems [41, 77, 107], with particular applications to cardiovascular modeling [44, 53] and aerodynamics [101]; or solid mechanics simulation tasks, including stresses [39, 50, 51,

54, 57, 61]. In cases where the governing equations are not given, the learning task becomes approximating them from data [2, 3, 14, 18, 30, 46, 60, 65, 72, 82, 83, 84, 85, 100].

2.3 Background and Problem Setup

PDEs, dynamical systems, and time integration Consider a PDE of the form $\frac{\partial u}{\partial t} = \mathcal{L}u$, where u is the unknown function and $\mathcal{L}$ is a possibly nonlinear operator that includes spatial derivatives of u. By discretizing in space, one obtains a dynamical system

$$\dot{x}(t) = f(x(t)) \tag{2.1}$$

with an N-dimensional state $x(t) \in \mathbb{R}^N$ at time $t \in [0, T]$. The function f is assumed to be Lipschitz to ensure solution uniqueness and the initial condition is denoted as $x_0 \in \mathbb{R}^N$. A PDE of a higher order in time can be reduced to the first-order form in the standard way, e.g., if we have a second-order system $\ddot{q}(t) = f(q(t))$, then we consider its formulation via position q and momentum p as a first-order system with $x = [q; p]$: $[\dot{q}(t); \dot{p}(t)] = [p(t); f(q(t))]$. To numerically integrate Equation 2.1, we choose time steps $0 = t_0 < t_1 < \cdots < t_K = T$. Then, a time integration scheme (e.g. [32, 33, 96]) gives an approximation $x_k \approx x(t_k)$ of the state $x(t_k)$ at each time step $k = 1, \ldots, K$. A list of the schemes we use along with details is given in Section 2.5.

Problem setup and learning problems Given M initial conditions $x_0^{(1)}, \ldots, x_0^{(M)} \in \mathbb{R}^N$ and the corresponding M trajectories $X^{(i)} = [x_0^{(i)}, \ldots, x_K^{(i)}] \in \mathbb{R}^{N \times (K+1)}, i = 1, \ldots, M$ obtained with a time integration scheme from dynamical system (2.1), we consider the following two learning problems, both of which aim to learn the physical model of the problem, viewed as unknown, from trajectory samples: **(1)** Learning an approximation f_θ of the right-hand side function f in Eq. (2.1). This gives an approximate $f_\theta(x(t)) \approx \dot{x}(t)$ that is then numerically integrated to produce a trajectory $\tilde{X}$ for an initial condition $\tilde{x}_0 = x_0$. The aim is that $\tilde{X}$ approximates well the true

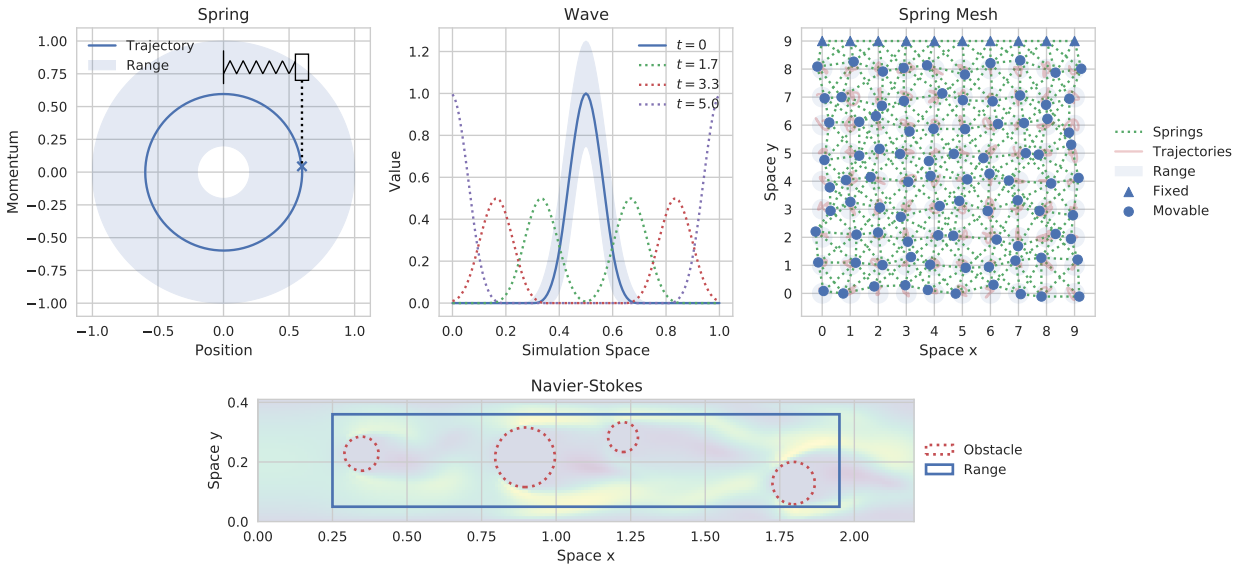


Figure 2.1: Representative visualizations of the four systems, depicting the results and ranges of initial condition sampling. Each has two state components: for the Navier-Stokes system, a flow velocity and a pressure field, and for the other three a position q and momentum p.

trajectory X obtained with f from (2.1) for the same initial condition. **(2)** Directly learning the next steps in the trajectory from the current one, i.e. predict $x_k^{(i)}$ given $x_{k-1}^{(i)}$.

To assess the learned models, we evaluate them on their ability to produce good approximate trajectories from randomly sampled initial conditions, by either integration or direct step prediction. During evaluation, we use initial conditions drawn independently from those used to produce training data, both from the same distribution as the training samples, as well as from a distribution with support outside the training range. We train networks on data sets of various sizes. For details, see Section 2.6.1.

2.4 Benchmark Systems

We consider four physical systems, illustrated in Figure 2.1: a single oscillating spring, a one-dimensional linear wave equation, a Navier-Stokes flow problem, and a mesh of damped springs. These systems represent a progression of complexity: the spring system is a linear system with a

low-dimensional space of initial conditions and low-dimensional state; the wave equation is a low-dimensional linear system with a (relatively) high-dimensional state space after discretization; the Navier-Stokes equations are nonlinear and we consider a setup with low-dimensional initial conditions and a high-dimensional state space; finally, the spring mesh system has both high-dimensional initial conditions as well as high-dimensional states. Additionally, the proposed spring system and Navier-Stokes problems represent diffusion-dominated and advection-dominated (for sufficiently low viscosity) PDE behaviors, as well as variability in initial conditions with fixed domain (spring system) and variable domain (Navier-Stokes). These varying complexities provide an opportunity to test methods on simpler systems and the ability to examine changing performance as system size increases, both in terms of the state dimension, and the initial condition distribution. The ground truth models for the spring, wave, and spring mesh systems with classical time integrators are implemented using NumPy [34], SciPy [102], and accelerated, where possible, with Numba [49]. The Navier-Stokes snapshots are generated using PolyFEM [86], a finite element library.

These systems were chosen in an effort to reflect the variety of systems used for testing in this area, while unifying choices of particular formulations. Past works have chosen systems of the types featured here: simple oscillators (both spring and pendulum [28]), particle systems with various interaction laws (gravity, spring forces, charges, cloth simulations, etc. [15, 17, 43, 71, 81]), and fluid-flow systems (with various sorts of obstacles, airfoils or cylinders [71, 99]). We make particular selections here in an effort to unify systems of interest and facilitate comparisons across experiments by providing a shared set of tasks which can be used for development and testing of machine learning methods.

Some examples of initial condition selection for each system are illustrated in Figure 2.1. The ground truth for the spring, wave, and spring mesh systems consists of the state variables (q, p) for position and momentum, and their associated derivatives $(\dot{q}, \dot{p})$. For the Navier-Stokes system the state consists of flow velocities, and a pressure field, along with approximated time derivatives

for each.

Table 2.1 lists the parameters used to generate trajectories for training and evaluation. Training sets of three sizes are generated, each containing the specified number of trajectories. The systems are integrated at the listed time step sizes, but the ground truth data is subsampled further by the factor shown after ÷ in the table: the integration schemes are run at a smaller time step and intermediate computations are discarded. Each larger training set is a strict superset of its predecessor to ensure that previous training samples are never removed.

2.4.1 Spring

We simulate a simple one-dimensional oscillating spring. In this system, the spring has zero rest length, and both the oscillating mass and spring constant are set to 1. The spring then exerts a force inversely proportional to the position of the mass q: $\dot{p}(t) = -q$ and $\dot{q}(t) = p$.

The energy of the system is proportional to $r = q^2 + p^2$ which is the radius of the circle in phase space. To sample initial conditions, we first sample a radius uniformly, then choose an angle theta uniformly. This produces a uniform distribution over spring system energy levels and starts at an arbitrary point in the cycle. The spring system has a closed-form solution: $(q(t), p(t)) = (r\sin(t + \theta_0), r\cos(t + \theta_0))$ where r is the radius of the circle traced in phase space (the energy of the spring) and θ_0 is the phase space angle at which the oscillation will start.

While this closed form solution is useful, for consistency with our other systems, we generate snapshots of the spring system by numerical integration. Simulations of the spring system always run through one period. For "in-distribution" training values, the radius is selected in the range $(0.2, 1)$ and "out-of-distribution" radii are chosen from $(1, 1.2)$.

2.4.2 Wave

This benchmark system is similar to the one used in [69]. Consider the wave equation with speed $c = 0.1$

$$\partial_{tt} u = c^2 \partial_{xx} u \,, \tag{2.2}$$

on a one-dimensional spatial domain $[0, 1)$ with periodic boundary conditions. We represent this second-order system as a first-order system and discretize in space to obtain

$$\begin{bmatrix} \dot{q}(t) \\ \dot{p}(t) \end{bmatrix} = \begin{bmatrix} 0 & I \\ c^2 D_{xx} & 0 \end{bmatrix} \begin{bmatrix} q(t) \\ p(t) \end{bmatrix}, \tag{2.3}$$

where $D_{xx} \in \mathbb{R}^{n \times n}$ corresponds to the three-point central difference approximation of the spatial derivative ∂_{xx} and the matrices I and 0 are the identity and zero matrix, respectively, of appropriate size. We discretize in space with $n = 125$ evenly spaced grid points and evolve the system following the dynamics described above.

Initial conditions are sampled with an initial pulse in the q component centered at 0.5. All initial conditions have zero momentum. The initial pulse is produced by a spline kernel as described in [69]:

$$s(x) = \frac{10}{p_w} \cdot |x - 0.5| \,, \qquad h(s) = p_h \cdot \begin{cases} 1 - \frac{3}{2}s^2 + \frac{3}{4}s^3 & \text{if } 0 \le s \le 1 \\ \frac{1}{4}(2 - s)^3 & \text{if } 1 < s \le 2 \\ 0 & \text{else} \end{cases}$$

where the width and height of the pulse are scaled by parameters p_w and p_h, respectively. The spline kernel pulse is then $h(s(x))$ for $x \in [0, 1)$, evaluated at the discretized grid points.

For "in-distribution" samples, parameters p_w, p_h are both chosen uniformly in the range $(0.75, 1.25)$ and "out-of-distribution" runs sample uniformly from $(0.5, 0.75) \cup (1.25, 1.5)$. All

trajectories are integrated until $t = 5$ when the wave has traveled through half a period.

2.4.3 Spring Mesh

This system manipulates a square grid of particles connected by springs, in a two dimensional space, and can be considered a simplified version of deformable surface and volume systems (cf. [71]). The particles all have mass 1, and are arranged into a unit grid. Springs are added along the axis-aligned edges and diagonally across each grid square, with rest lengths selected so that the regularly-spaced particles are in a rest position.

In this work we use a 10×10 grid where the top row of particles is fixed in place. Initial conditions are sampled by choosing a perturbation for the position of each non-fixed spring. These perturbations are chosen as uniform vectors inside a circle with radius 0.35. Out-of-distribution perturbations are chosen uniformly in a ring with inner radius 0.35 and outer radius 0.45. The sampled initial conditions all have zero momentum.

In this system, a spring between particles a and b exerts a force:

$$F_{ab} = -k \cdot \left(\|q_a - q_b\|_2 - \ell_{ab}\right) \frac{q_a - q_b}{\|q_a - q_b\|_2} - \gamma \dot{q}_a \tag{2.4}$$

where ℓ_{ab} is the rest length of the spring, $\gamma = 0.1$ is a parameter controlling the magnitude of an underdamped velocity-based decay, and $k = 1$ is the spring constant.

2.4.4 Navier-Stokes

We consider the standard Navier-Stokes equation over a domain Ω (cf. [71, 99]):

$$\left.\begin{aligned} \rho \frac{\partial u}{\partial t} + \rho(u \cdot \nabla)u - \nu \Delta u + \nabla p &= b \\ \nabla \cdot u &= 0 \\ u(0) &= u_0 \end{aligned}\right\} \text{on } \Omega \times (0, T) \quad \text{and} \quad \left.\begin{aligned} u &= d \\ \nu \frac{\partial u}{\partial n} + pn &= g \end{aligned}\right\} \text{on } \partial\Omega_D \times (0, T) \tag{2.5}$$

Table 2.1: Dataset sizes and simulation parameters

System	# Train Trajectories	# Eval Trajectories	Time Step Size		# Steps
Spring	10, 500, 1000	30	0.00781	÷128	805
Wave	10, 25, 50	6	0.00049	÷8	10204
Spring Mesh	25, 50, 100	15	0.00781	÷128	805
Navier-Stokes	25, 50, 100	5	0.08	÷1	65

where $u : \Omega \times (0, T) \rightarrow R^2$ is the velocity at time $t \in (0, T)$ of a fluid with kinematic viscosity ν and density ρ, $p : \Omega \times (0, T) \rightarrow R$ is the pressure and $\partial\Omega_D$ and $\partial\Omega_N$ are the Dirichlet and Neumann boundary conditions, respectively. In our setup we use the finite element method (FEM) to solve the PDE using mixed discretization: quadratic polynomial for the velocity and linear for pressure. In our experiment the domain Ω is a rectangle 0.22×0.41 with a randomly generated set of circular obstacles. We start with $u_0 = 0$ and specify a velocity on the left boundary of $u(0, y) = (6(1 - e^{-5t})(0.41 - y)y/0.1681, 0)$, zero on the top and bottom, and zero Neumann on the right side ($g = 0$). We solve the system using PolyFEM [86] using $dt = 0.08$ and backward differentiation formula (BDF) of order 3 for the time integration.

We sample obstacles into two configurations: a single obstacle, or a set of four. In each case, we sample the obstacles leaving a margin of 0.05 between each circle, and a margin of 0.25 from the left and right sides, and 0.05 between the top and bottom. Otherwise, each obstacle is determined by first sampling a radius, then sampling a center from the valid space, respecting the margins. If the sampled obstacle is too close to an existing circle, it is discarded and a new sample is drawn. In-distribution obstacles have radii in the range $(0.05, 0.1)$ and out-of-distribution radii are drawn from $(0.025, 0.05)$.

2.5 Numerical Integration Schemes

We briefly review the time integration schemes that we consider in this study: forward Euler (FE), leapfrog (LF), Runge-Kutta 4 (RK4), backward Euler (BE), and the second-order backward differen-

tiation formula (BDF2). Other sources also discuss these integration schemes, for example [32, 33, 96]. In what follows, f is the right-hand side function, $\delta_t > 0$ is the time step size of the integrator, and x_k, t_k are, respectively, discrete states and simulation times indexed by k.

2.5.1 Explicit Methods

Forward Euler (FE) Time integration with the explicit Euler method leads to

$$x_k = x_{k-1} + \delta_t f(x_{k-1}). \tag{2.6}$$

Runge-Kutta 4 (RK4) The explicit Runge-Kutta 4 scheme is

$$x_k = x_{k-1} + \frac{\delta_t}{6}\left(h_1 + 2h_2 + 2h_3 + h_4\right), \tag{2.7}$$

where

$$\begin{aligned} h_1 &= f(x_{k-1}) & h_2 &= f(x_{k-1} + \delta_t/2h_1) \\ h_3 &= f(x_{k-1} + \delta_t/2h_2) & h_4 &= f(x_{k-1} + \delta_t/2h_3) \end{aligned}$$

for $k = 1, \ldots, K$.

Leapfrog (LF) For leapfrog integration we separate the components of the state $x = (q, p)$ and $f(q_k, p_k) = (\dot{q}_k, \dot{p}_k)$ and compute:

$$\begin{aligned} p_{k+1/2} &= p_k + \frac{\delta_t}{2}\dot{p}_k \\ q_{k+1} &= q_k + \dot{q}(q_k, p_{k+1/2})\delta_t \\ p_{k+1} &= p_{k+1/2} + \frac{\delta_t}{2}\dot{p}(q_{k+1}, p_{k+1/2}) \end{aligned} \tag{2.8}$$

where the notation $\dot{q}(q_k, p_{k+1/2})$ denotes the $\dot{q}$ component of $f(q_k, p_{k+1/2})$ and analogously for $\dot{p}$.

2.5.2 Implicit Methods

Backward Euler (BE) We also consider the implicit Euler method, which is given by the potentially nonlinear equation

$$x_k - \delta_t f(x_k) = x_{k-1} \tag{2.9}$$

that is solved in each time step $k = 1, \ldots, K$.

Backward Differentiation Formula (BDF2) We tested another implicit method, BDF2. This is a second order multistep method with the formula given by:

$$x_k - \frac{4}{3}x_{k-1} + \frac{1}{3}x_{k-2} = \frac{2}{3}\delta_t f(t_k, x_k) \tag{2.10}$$

To kickstart this method, which requires two steps of history, we initially do one step of backward Euler. This maintains the stability and error properties of the method.

2.6 Numerical Experiments

We apply several basic learning methods to the datasets developed in this work: k-nearest neighbor regressors, a neural network kernel method, several sizes of feed-forward MLPs, and a variety of CNNs. Details of the architectures and the training protocol are provided in supplementary material, Section 2.6.1. Each of the neural networks we consider is implemented using PyTorch [66].

The learning methods considered in this work are each trained on one of the two target task formulations described in Section 2.3. For derivative-based prediction, the training is conducted supervised on ground truth snapshots gathered from the underlying models. For each system we randomly sample initial conditions and each of these is then numerically integrated to produce

a trajectory. Each trajectory includes state samples x as well as target derivatives $\dot{x}$ used for training. For direct prediction, we no longer require numerical integration; instead we directly predict the trajectory in a sequential fashion. In this setting, we approximate $f_\theta(x(t)) \approx x(t + \delta_t)$ for a discrete time step size δ_t. For the derivative prediction task we report results using the leapfrog integrator. Full results using other numerical integration schemes are available in the supplementary materials.

We pick the same set of learning methods and apply it to both tasks independently to judge performance in each. For many systems the state is divided into position and momentum components: $x \equiv (q, p)$. For the Navier-Stokes problem, the state x is made up of the flow velocity field, and the scalar field for pressure. After training, we produce rolled-out trajectories from held-out initial conditions, either by combining with a numerical integrator in the case of derivative prediction, or in a directly recurrent fashion in the case of step prediction. Each neural network is instantiated in three independent copies, each of which is trained and evaluated across all sampled trajectories. We compute a per-step MSE against a ground truth value, average these per-step MSEs to produce a per-trajectory error, and record these errors for analysis. Our experiments are designed to test several aspects of physical simulation. We highlight the most salient ones below in Section 2.6.2, and report more extensive results in Appendix A.

2.6.1 Learning Methods

Training

Training for both step and derivative problem formulations is done with the Adam [42] optimizer for all neural networks, except the neural network kernel which uses standard stochastic gradient descent with learning rate 0.001 and weight decay 0.0001. With the Adam optimizer, no weight decay is used, and most networks use a learning rate of 1×10^{-3}. Exceptions to this are: CNNs, MLPs and the u-net for Navier-Stokes, and CNNs and MLPs on the spring mesh. For both of these

systems the CNNs and MLPs use a learning rate of 1×10^{-4} and the u-net uses 4×10^{-4}.

On the Navier-Stokes system we also perturbed each batch of training data with normally-distributed noise with a variance of 1×10^{-3}. For step prediction the previous step was corrupted and the subsequent step left uncorrupted. For derivative prediction, the derivatives were updated to correct for the noise (i.e. $\tilde{x} = x + \mathcal{N} \implies \tilde{\dot{x}} = \dot{x} - \mathcal{N}$ where $\mathcal{N}$ is the sampled noise). This is inspired by the approach taken in [71] and we found it to improve stability for neural networks on the Navier-Stokes system.

The number of training epochs varies based on the target system. On spring, wave, and spring mesh the networks are trained for 400, 250, 800, and 800 epochs, respectively. When reporting evaluation errors below, we average errors over all time steps of each randomly-sampled trajectory in the held-out evaluation set.

We train three independent copies of each neural network. When evaluating these, each test trajectory is evaluated with each duplicate neural network and the performance results are collected and processed together. Variance in plots of these results is produced both by the differences in performance for the three duplicated neural networks, and differing performance across the sampled evaluation trajectories.

KNN Regressor

We use a k-nearest neighbors regressor to predict the value of the state derivatives, using $k = 1$. With this method $f_\theta(\tilde{x}_k^{(i)})$ finds the closest matching point in the training set, and uses that point's associated derivatives as its approximation, $\tilde{\dot{x}}_k^{(i)}$ in the case of derivative prediction. For direct step prediction, the KNN finds the closest point and returns the next time step from that point's trajectory in the training set. We use the KNN implemented in scikit-learn [68], along with its default Minkowski metric.

Kernel Methods

Kernel methods provide a nonparametric regression framework [87]. In this benchmark we consider dot-product kernels of the form $k(x, x') = \eta(\langle x, x' \rangle)$, which can be efficiently implemented in their primal formulation using random feature expansions [73] via the representation

$$k(x, x') = \mathbb{E}_{z \sim \nu}[\rho(\langle x, z \rangle)\rho(\langle x', z \rangle)] \approx \frac{1}{L} \sum_{l=1}^{L} \rho(\langle x, z_l \rangle)\rho(\langle x', z_l \rangle) \,,$$

where ν is a rotationally-invariant probability distribution over parameters and $z_l \sim \nu$ iid. The resulting maps $x \mapsto \rho(\langle x, z_l \rangle)$ are *random features*, associated with a shallow neural network with 'frozen' weights. While further choices of kernel may be considered in the future, dot-product kernels have flexible approximation properties and are easily scalable [79].

In our experiments, we use $\rho = \mathrm{ReLU}$ and $L = 32768$ random features and train using kernel ridge regression. We do not apply this approach to our Navier-Stokes system as its large state dimension makes achieving a sufficiently large set of random features infeasible.

Deep Networks

MLPs We apply simple multilayer perceptron (MLP) networks in a variety of sizes. The configuration of the MLPs used varies with the target system. In particular, we divide our two systems into two classes: those with smaller state dimension (the spring and wave systems), and those with a larger state dimension (the spring mesh, and the Navier-Stokes problem). We describe these architectures in terms of "depth" and "width." The depth denotes the number of fully-connected operations in the MLP, so that for a depth of d there are $d - 1$ hidden layers. The width is the size of each hidden layer; the input and output dimensions are fixed by the state dimension of the system. The MLPs use tanh activations.

For the small systems we use three MLP architectures: (1) a depth of 2 and a hidden dimension (width) of 2048, (2) a depth of 3 and width of 200, and (3) a depth of 5 and a hidden dimension of

2048. For the large systems, we use two architectures: (1) a depth of 4 and width of 4096, and (2) a depth of 5 and width of 2048. The 10×10 spring mesh merges both sets of MLP architectures.

For the Navier-Stokes and spring mesh systems, the MLP gets as input both the current network state, and a one-hot mask indicating which points in the discrete simulation space are "fixed," meaning either a boundary point, a point in an obstacle, or an immovable, fixed particle.

CNNs We also test several feed-forward convolutional neural networks. These use ReLU activations and we specify their architectures by a kernel size, and internal channel count. We use these simple CNNs only on the larger systems: the spring mesh and the Navier-Stokes. For both of these systems we test two CNN architectures: both have a kernel size of 9×9 and, respectively, 32 and 64 channels internally. The number of input channels is fixed by the system. Both systems have five: for the spring mesh, two channels each for position and momentum; and for the Navier-Stokes system two channels for velocity, one for pressure field, and two more for one-hot masks highlighing boundaries and the obstacles.

U-net Finally, we implement another convolutional network—only for the Navier-Stokes system—a u-net following the architecture tested in [99]. That work applied this architecture to another Navier-Stokes problem, predicting a single step of flow about an airfoil profile. Here we adjust the input and output channels of this architecture, and test on our Navier-Stokes problem, performing several recurrent steps of derivative or step prediction around circular obstacles.

The architecture itself consists of seven convolution operations on both the downsampling and upsampling side. The convolutions have a mix of 4×4 and 2×2 kernels, and have strides of two. The network includes skip connections common to u-net-style architectures. With each downsampling, the number of channels is doubled starting from an internal channel count of 64. Our Navier-Stokes system has a grid size of 221×42. To accommodate the amount of downsampling in this architecture we first upsample to 256×256 with bilinear interpolation.

Other Experimental Details

Our experiments were conducted on NYU's research HPC system, Greene. Neural networks were predominantly trained using NVIDIA RTX8000 GPUs, with a few runs on V100 GPUs. CPU-based runs used Intel Xeon Platinum 8268 CPUs. Our neural networks required, on average, approximately two hours to train and we consumed in total approximately 1785 hours of GPU time, across all our experiments, including some early experimental and exploratory runs not discussed here. Our dataset generation and non-neural network evaluation runs, which do not use GPUs, consumed approximately 2270 core-hours of CPU time, again including some exploratory runs. Datasets were generated using CPUs only. Neural network training and evaluation passes ran using GPUs through PyTorch. Evaluations and trainings of baseline numerical integrators and KNNs ran on CPU only.

2.6.2 Experiment Results

In this section we present a summary of the main results of our numerical experiments. An extended version of these results is included in Appendix A.

Training set size In general ML problems, one would expect additional training samples to systematically improve (in-distribution) evaluation performance. However, Figure 2.2 illustrates a clear saturation of performance on the simplest systems when using neural networks as function approximators, in contrast with non-parametric KNNs and the kernel method. We attribute this saturation to an inherent gap between the training and evaluation objectives. While data-driven methods are optimized to minimize next-step predictions, the final evaluation requires built-in stability to prediction errors. Including regularisation strategies to incorporate stability, such as noise injection [71], is shown to help, but not fully resolve this issue.

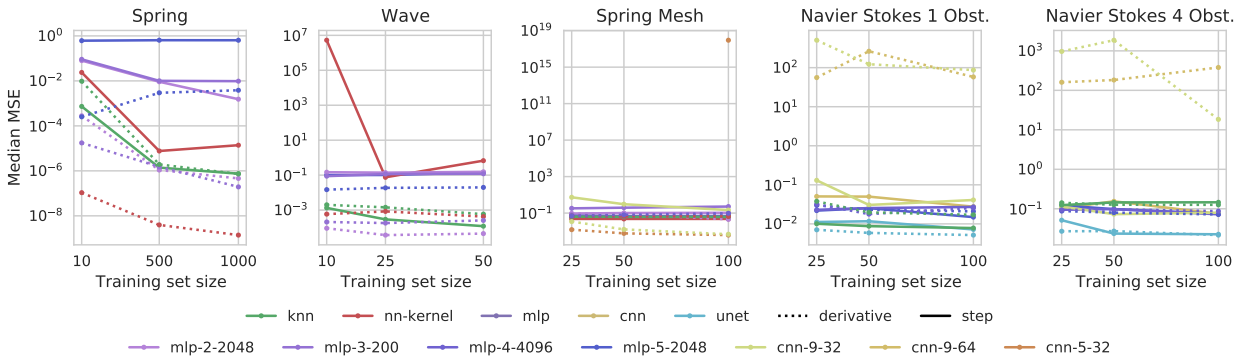


Figure 2.2: Median MSE error with respect to the training set size for each of our system configurations. We show varying architecture choices for each method.

Out-of-distribution evaluation For simplicity, we only examine the out-of-distribution error for networks trained on the largest training set size. The added challenge of out-of-distribution samples varies with the construction of each system. It is possible to get some idea of the difficulty increase by examining the accuracy penalty for the KNNs, and comparing it to how well the more advanced models are able to generalize.

Benefits of neural networks for generalization over KNN are visible across several systems in Figure 2.3, particularly in the spring system for small MLPs for derivative prediction and nn-kernel in both cases. The KNN suffers a significant increase in error while these methods produce only somewhat worse predictions. Benefits are still present, though less pronounced, for the wave system derivative prediction where neural networks increase in error, but the kernel method and small MLP maintain a lower absolute error than the KNN. On the Navier-Stokes systems none of the methods suffers an increase in error for out-of-distribution evaluation. The change in radius distribution for the obstacles did not pose an additional challenge sufficient to produce a measurable change in error distribution. We attribute this to low dimension of the initial condition space.

Step and derivative prediction The step and derivative prediction instances of each learning problem lead to different accuracy from the learning methods we test. While most physical systems

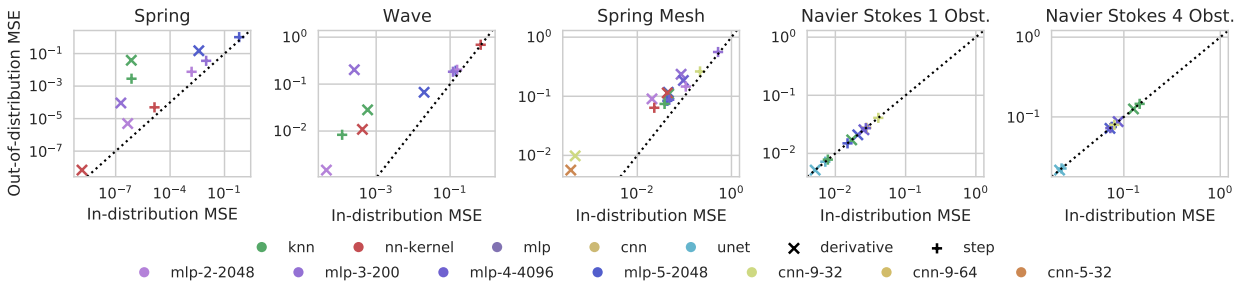


Figure 2.3: Median MSE for in-distribution evaluation sets vs. out-of-distribution evaluation sets for each system. Colors represent the same method and are varied for different architecture choices. Marker shapes distinguish step and derivative prediction, and the dotted line is the identity line. Outliers for the spring mesh and both Navier-Stokes configurations were removed. Values on both axes were approximately 10^{13} for the spring mesh and in the range 10^2–10^3 for Navier-Stokes.

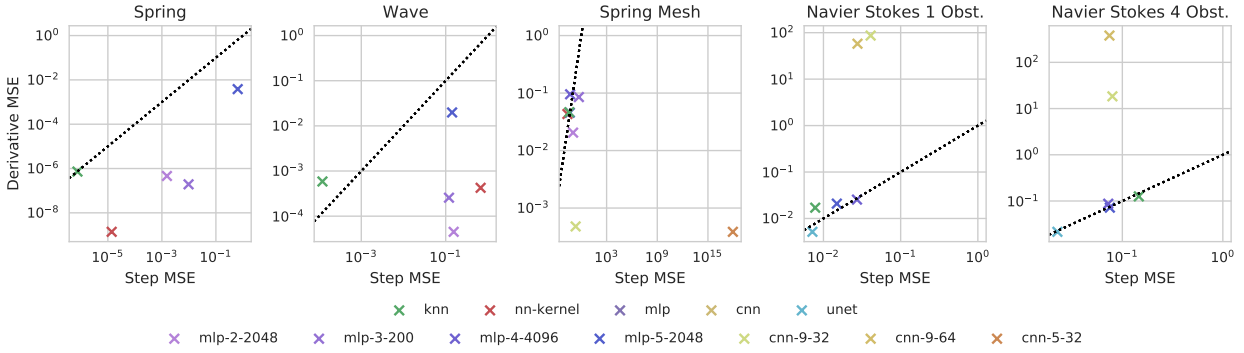


Figure 2.4: Median MSEs for derivative vs. step prediction on the same evaluation set. Results are displayed for each of our system configurations. The dotted line is the identity line.

are naturally described in terms of their derivatives through corresponding ODEs/PDEs, data-driven simulations also offer the alternative of bypassing this differential formulation and predict the next state directly. Such 'cavalier' approach avoids the compounding error amplification effects across integration steps, at the expense of sample efficiency. Figure 2.4 illustrates these tradeoffs across our systems.

An important example of this effect is the performance of CNNs on the spring mesh system (Figure A.3 in the Appendix). When working through a numerical integrator and performing derivative prediction they produce the lowest error of all methods tested, but following the same

training protocol for step prediction these architectures produce high errors, or are unstable. This case is likely an interaction of the architecture with the specific learning task. For the spring mesh, step prediction requires outputting the position of the particle which requires manipulating its global coordinates, while derivative predictions allow the network to more easily act locally and compute only a relative motion for the particle. The derivative prediction task better takes advantage of the spatial invariance of the CNNs. This difference in performance reflects the importance of tailoring architectures to the specific task, and some potential for neural network architectures to benefit from incorporating knowledge of a system's behavior.

System and dataset complexity Several trends we observe correlate with the difficulty of learning to simulate a system, and the variation in its behavior across the training and evaluation samples. This is generally a combination of the system's state dimension, and variation in its behavior, approximated by the dimension of the distribution from which initial conditions are sampled.

This is particularly visible in Figure 2.5 in the performance of the KNN methods, and, in many cases, the performance of simpler methods such as the small MLPs. On the simpler systems, such as the spring and wave, the KNNs generally perform well because even though the wave system has a relatively large state dimension of 125, like the spring its initial condition is sampled from only two parameters and its behavior can be readily predicted from these. The Navier-Stokes system with a single obstacle is another instance of this sort of behavior: the KNN is readily able to reproduce flows it has not seen because a sampling of 100 obstacle positions is such that an evaluation sample is close to a trajectory seen at training time. Therefore, small MLPs and the kernel method produce similar performance. When the difficulty is increased by sampling four obstacles, the KNN and MLP performances suffer, and larger networks such as the u-net are needed to maintain approximately the same performance.

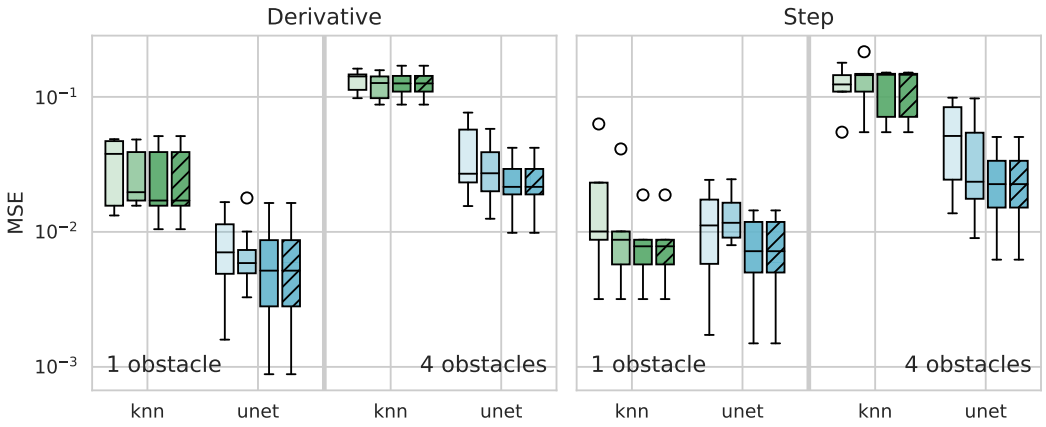


Figure 2.5: MSE distribution across trajectories in the evaluation set for KNNs and u-nets on our Navier-Stokes system, both the formulation with one obstacle, and the settings with four, as well as for derivative and step prediction. Box plot mid-lines show medians, box area is between first and third quantiles, whiskers extend 1.5× beyond the boxes, outliers are plotted past the whiskers. Darker colors denote increasing training set size. The final hatched box is the same network from the final un-hatched box, tested on an out-of-distribution evaluation set.

Choice of numerical integrator For our derivative prediction tasks we combine our trained methods with three explicit integrators with orders 1, 2, and 4. In most of our systems these produce at most a small increase in accuracy, holding all other training and evaluation parameters equal. However on the Navier-Stokes system the higher order integrators produce somewhat higher errors, particularly for the u-net and the MLPs. This appears related to the approximated derivatives used for training this system. The learned derivatives produce some small deviations which are compounded when combining multiple derivative samples.

Computational overheads Another important aspect to consider when applying learning methods to physical simulation problems is the time required to compute each step, and the computational overheads introduced by the lack of knowledge of the true system physics. With standard numerical integration methods, it is generally possible to improve the quality of generated trajectories by decreasing the size of the time step used during integration. We take advantage of this in order to estimate the time overheads of our learning methods relative to our baseline

Table 2.2: Time comparison for derivative prediction against baseline numerical integrators

System	Architecture	Euler		Leapfrog		RK4	
		Time Ratio	Scaling	Time Ratio	Scaling	Time Ratio	Scaling
Spring	knn	367.0	1×	405.8	16×	311.3	64×
	nn kernel	180.7	1×	198.6	1×	173.3	1×
	mlp-2-2048	185.8	1×	191.6	16×	177.7	64×
	mlp-3-200	237.6	1×	237.5	16×	227.2	64×
	mlp-5-2048	473.8	4×	369.6	64×	360.9	128×
Wave	knn	24,102.7	8×	16,945.1	256×	16,132.0	256×
	nn kernel	35.3	8×	22.3	256×	19.9	256×
	mlp-2-2048	25.4	8×	16.5	256×	14.2	256×
	mlp-3-200	31.0	16×	19.7	256×	17.8	256×
	mlp-5-2048	60.5	16×	38.0	256×	34.6	256×
Spring Mesh	knn	708.6	8×	626.1	128×	690.4	256×
	nn kernel	5.3	8×	4.4	128×	4.8	256×
	mlp-2-2048	3.0	8×	2.6	128×	2.8	256×
	mlp-3-200	3.4	8×	3.2	128×	3.4	256×
	mlp-4-4096	7.8	16×	7.1	128×	7.7	256×
	mlp-5-2048	7.1	8×	6.2	128×	5.3	256×
	cnn-9-32	11.0	2×	10.5	32×	11.3	64×
	cnn-5-32	7.4	1×	9.2	32×	7.3	64×

numerical integrators at approximately corresponding error levels.

We numerically integrate each system at time step sizes scaled by powers of two. For each trajectory in the derivative prediction setting, we find the smallest scaling factor at which the numerical integrator exceeds the learning method's error at their final shared time step, approximating the factor by which numerical integration can be made faster until it begins to underperform the learned method.

Table 2.2 reports the results of these experiments. For each numerical integrator, the "scaling" column reports the most common scaling factor found for each trajectory. The "time ratio" column represents the learned method's evaluation overhead (median times, counting only per-step network evaluation costs, not numerical integration or data transfers). Note that the numerical integrator makes fewer steps than the learned method so the overall trajectory time must be further adjusted by the scaling factor.

In general, the neural networks are slower per-step by one or two orders of magnitude. KNNs are slower by significantly larger factors, particularly for the wave system. This is likely partially due to the default scikit-learn KNN implementation used, and due to the large size of the wave system training sets (large state dimension and large number of training snapshots). Scaling factors increase with the order of the integrator as higher-order integrators are more tolerant of large step sizes and maintain low error.

It is likely that these overheads could be reduced with more optimized implementations of both the numerical integrators and learned methods. The derivative prediction task is also constrained by its need to interact with the numerical integrator. In this setting the learned methods cannot be expected to outperform the quality of the solutions generated by the true system derivatives. This reflects a penalty resulting from a lack of knowledge of the true underlying system, and a penalty for learning from observations in this case. Step prediction without involving the numerical integrator potentially avoids some of these constraints, if learning is successful.

2.7 Conclusions and Limitations

The results in this work illustrate the performance achievable by applying common machine learning methods to the simulation problems in our proposed benchmark task. We envision three ways in which the results of this work might be used: **(1)** the datasets developed here can be used for training and evaluating new machine learning techniques in this area, **(2)** the simulation software can be used to generate new datasets from these systems of different sizes, different initial condition dimensionality and distribution; and the training software could be used to assist in conducting further experiments, and **(3)** some of the trends seen in our results may help inform the design of future machine learning tasks for simulation.

For the first and second groups of downstream users, we have made available the pre-generated datasets used in this work, as well as the software used to produce them and carry out our

experiments. These components allow carrying out the measurements we have made here, and permit further adjustments to be made.

For the third group, we highlighted a few trends that suggest useful steps to take in developing new problems and datasets in this area. First, we advise including several simple baseline methods when designing new tasks. In particular the inclusion of standard numerical integrators (for derivative-type problems) and KNNs are useful to evaluate the difficulty of the proposed task. Specifically, KNNs are useful for examining the performance achievable by memorizing the training set, and are thus *witnessing* an appropriate design of data distribution that captures the true high-dimensionality of the prediction task. As an example, in the Navier-Stokes examples some task formulations may inadvertently be simple to memorize, even if the complexity of the system itself may not immediately suggest it. The numerical integrators are likewise useful as baselines both to ensure that the derivative learning is feasible even when achieving no error in predictions, and also to evaluate the penalty in accuracy which is incurred by operating without access to the true physics. We believe that in light of these observed trends, including baseline methods such as standard numerical integration schemes and simple learning methods such as the KNN is important in understanding tasks in this area. Including these assists in experiment design by helping to calibrate the difficulty of a target task.

Limitations While our benchmark provides actionable conclusions on a wide array of simulation domains, it is currently focused on temporal integration, and as such it does not cover important settings in scientific computing. For instance, we do not currently include an instance of a surrogate model, which could provide different tradeoffs benefiting ML models. Additionally, we have focused on two setups for data-driven simulation (differential snapshot prediction and direct snapshot prediction), but other alternatives exist that might mitigate some of the shortcomings we observed; for instance by considering larger temporal contexts (as in [15]), as well as enforcing certain conservation laws into the model [15, 28]. Finally, while we report some measurements of

timings and relative computational overheads, there are other dimensions to the time-accuracy tradeoff which remain to be explored and further software optimizations are most likely possible.

Chapter 3

Closure Modeling and Climate Applications

Having examined the task of modeling entire physical systems using neural networks (using either step or derivative prediction as defined in Section 1.1), we now turn our attention to a *hybrid* application targeting a real world need for improved modeling. In particular, we will examine the task of learning subgrid parameterizations or forcings for climate models. This is a particular instance of a closure problem, in which the evolution of a system's dynamics is not closed—that is, fully determined—by the state values tracked in the simulation. For example, consider a system tracking a right-hand side function with the state variable partitioned into two components:

$$x_{ab} \triangleq \big(x_a(t),\, x_b(t)\big) \in \mathbb{R}^{n+m} \tag{3.1}$$

$$\dot{x}(t) = f\big(x_a(t),\, x_b(t)\big) \tag{3.2}$$

In certain situations, we may be interested in projecting the evolution of this system onto only the portion of the state in x_a:

$$\dot{x}_a(t) = \bar{f}\big(x_a(t)\big) \tag{3.3}$$

However, in this simple approach we no longer resolve the portion of the state in x_b and the evolution of this system is non-Markovian. Closure modeling is the problem of extending a system to account for the contribution of these missing states.

Formally, we could account for these interactions and cast this system as a Langevin equation through Mori-Zwanzig theory [22, 112]

$$\dot{\bar{x}}_a(t) = \underbrace{i\Omega\bar{x}_a(t)}_{\text{frequency term}} \underbrace{- \int_0^t K(\tau)\bar{x}_a(t-\tau)\, d\tau}_{\text{memory term}} + \underbrace{F(t)}_{\text{random force}} \tag{3.4}$$

where the frequency term tracks the contribution of the resolved variables $\bar{x}_a$, and the random force term accounts for the impact of the unresolved (and therefore generally unpredictable) values from x_b. The memory term implements the non-Markovian evolution due to the unresolved state variables.

In practical scenarios the dependence on past history decays rapidly and so only a finite memory is necessary. This approach lends itself naturally to stochastic closures which can directly model the noisy impact of the random force term. However, in many cases modeling the closure as deterministic is sufficient, and it may be acceptable to treat it as Markovian and forgo the need to maintain any memory.

3.1 Climate Subgrid Forcing

Closure problems arise in a wide variety of settings. Here, however, we consider in particular those which arise as a result of the limited resolution of numerical simulation grids. Instead of partitioning a simulation state x into separate entries, one could instead imagine partitioning it into different *scales*. Specifically, taking a continuous state field and representing it on a discrete simulation grid will necessarily fail to resolve details finer than the grid's resolution. This scenario commonly occurs in climate models where, even with increases in computational resources, it is

infeasible to simulate the behavior of many parts of the climate system at a fine resolution, across the full globe, for the time spans required [23, 24].

In the research projects that are discussed in Chapter 4 and Chapter 5, we mainly target turbulent fluid simulations of the types that arise in models of the atmosphere or ocean. The subgrid parameterization task arises from the coarsening induced by the simulation's grid. For each target system, we consider its true dynamics to be fully defined on a grid with a very fine, "true" resolution x. However, carrying out a simulation at a scale this fine is not possible. Instead we run a simulation at a "high" resolution with states $\bar{x}$. The overline bar represents a filtered version of x. The specifics of this operation depend on the target system and will be discussed in Section 3.2. The coarsening operator, by discarding the finer scales at the "true" resolution, induces a commutation error in the dynamics:

$$S_x \triangleq \overline{\frac{\partial x}{\partial t}} - \frac{\partial \bar{x}}{\partial t}. \tag{3.5}$$

We will seek to model the quantity S_x (the subgrid closure or forcing) so that it can be added as a correction term to the "high" resolution models. Traditional approaches exist targeting this problem [20] which primarily increase dissipation normally provided by the smallest eddies. However, we will train neural networks f_θ to approximate these forcing values, leaving the possibility that the networks may learn more subtle relationships between the resolved scales and the required subgrid forcing terms. While many works produce stochastic models for the subgrid closure, our networks will be deterministic.

An alternative to this approach is to instead train a network f_θ to drive the entire simulation. That is, to perform either step or derivative prediction, in the terminology from Section 1.1.3:

$$f_\theta(\bar{x}_i) \approx \bar{x}_{i+1}$$
$$f_\theta(\bar{x}) \approx \overline{\frac{\partial x}{\partial t}}.$$

In this case the model f_θ is an emulator for the true resolution system, which operates at a coarsened resolution. Such approaches have the effect of implicitly solving the closure problem, since they learn all of the system dynamics from the true resolution behavior including the influence of the small scales. Emulators completely remove the underlying true dynamics. Producing accurate trajectories requires that the trained network reliably produce accurate outputs. However, as we have seen in Chapter 2, learning the behavior of simulations from observations is a challenging task. Such models often have high error, or have difficulty generalizing to states outside of the training distribution. Even so, recent works have begun to find methods to stabilize such models for climate applications [19, 67, 94] and emulators have found success when applied to weather forecasting [47, 48].

3.2 Test Systems

We develop our parameterizations, targeting two simplified models: (1) a quasi-geostrophic system (QG) and (2) a Kolmogorov flow (KF) system, a Navier-Stokes system configured with a periodic forcing.

3.2.1 Quasi-Geostrophic System

This system is a two-layer quasi-geostrophic model as implemented in the Python package PyQG. It provides a simplified approximation of fluid dynamics [1]. For the purposes of our experiments, we have developed a JAX port of this model [10, 63]. This port is distributed as a standalone Python package with documentation including worked examples, and has seen some use in projects by other researchers. This system tracks the evolution of a potential vorticity q on a regular grid, divided into two layers q_1, q_2 with periodic boundary conditions. Corrections S_q are applied to the time evolution of this field. Models receive the current potential vorticity state q as inputs and predict S_q across both layers directly.

This system is pseudospectral, and a large portion of the numerical calculations for time stepping are carried out in Fourier space (denoted below by a hat). The evolution of the quantities follows [78]:

$$\frac{\partial \hat{q}_1}{\partial t} = -\widehat{J(\psi_1, q_1)} - ik\beta_1\hat{\psi}_1 + \widehat{\text{ssd}} \tag{3.6}$$

$$\frac{\partial \hat{q}_2}{\partial t} = -\widehat{J(\psi_2, q_2)} - ik\beta_2\hat{\psi}_2 + r_{\text{ek}}\kappa^2\hat{\psi}_2 + \widehat{\text{ssd}} \tag{3.7}$$

The field $\hat{q}$ in Fourier space is indexed by k, l which are wavenumbers in the zonal and meridional directions (the axis-aligned directions in our regular grid). $\kappa \triangleq \sqrt{k^2 + l^2}$ is the radial wavenumber. J is the horizontal Jacobian

$$J(A, B) \triangleq \frac{\partial A}{\partial x}\frac{\partial B}{\partial y} - \frac{\partial A}{\partial y}\frac{\partial B}{\partial x}. \tag{3.8}$$

The Jacobian is computed in real space, following which the result is carried back into Fourier space, making this model pseudospectral. The quantity "ssd" is a small scale dissipation which is implemented as a spectral filter applied after each time step:

$$\mathcal{F}(\hat{\mathrm{x}}_{k,l}) = \begin{cases} \hat{\mathrm{x}}_{k,l} & \text{if } \kappa < \kappa_c \\ \hat{\mathrm{x}}_{k,l} \cdot e^{-23.6\alpha(\kappa-\kappa_c)^4\delta_x^4} & \text{if } \kappa \geq \kappa_c \end{cases} \tag{3.9}$$

In the above, κ is again the radial wavenumber, and δ_x is the grid spacing determined by L, W (described below) and the grid resolution. In our use of this system, all grids are square, so $\delta_x = \delta_y$. The filter cutoff κ_c is set to $0.65\pi/\delta_x$. The parameter α controls the sharpness of the filter. When generating ground truth data it is fixed as $\alpha = 1$.

The streamfunction ψ is determined by q through the system

$$\begin{bmatrix} -(\kappa^2 + F_1) & F_1 \\ F_2 & -(\kappa^2 + F_2) \end{bmatrix} \begin{bmatrix} \hat{\psi}_1 \\ \hat{\psi}_2 \end{bmatrix} = \begin{bmatrix} \hat{q}_1 \\ \hat{q}_2 \end{bmatrix} \tag{3.10}$$

which is solved at each step. Each block in the matrix above is diagonal. The values F_1, F_2 are configurable system parameters and κ is again the radial wavenumber.

In our experiments, we use the "eddy" configuration which has also been used in prior works using this model and implementation [78]. This configuration sets the following values for model constants:

$$r_{\text{ek}} = 5.787 \times 10^{-7} \qquad F_1 = \frac{1}{r_d^2(1+\delta)}$$
$$\delta = \frac{H_1}{H_2} = 0.25 \qquad F_2 = \delta F_1$$
$$\beta = 1.5 \times 10^{-11} \qquad W = 10^6$$
$$r_d = 15\,000 \qquad L = 10^6$$

where H_1, H_2 are the heights of each of the two layers of q; L, W are the length and width of the other edges of the grid; r_d is a deformation radius; r_{ek} controls drag on the lower layer; and β is the gradient of the Coriolis parameter. For more information on the model configuration, consult [78] and the documentation for the PyQG package [1].

When generating ground truth data, we use a "true" resolution grid with dimension 256×256, and use the default PyQG Adams-Bashforth method for time stepping (see Equation 1.8). We use a time step $\delta_t = 3600$ and generate 86400 steps. We skip half of these as a warmup phase, and from the remaining steps we keep every eighth leaving 5400 per trajectory. Our training set consists of 100 such trajectories, and our evaluation set contains 10.

Each step produces a ground truth potential vorticity q_{true} along with a spectral time derivative $\partial \hat{q}_{\text{true}}/\partial t$. From these we apply our coarsening operator as in Equation 3.5 to produce filtered and coarsened values $\bar{q}_{\text{true}}$ at resolutions of 128×128, 96×96, and 64×64.

When coarsening our samples to smaller scales, we apply an operator C to the "true" resolution outputs q_{true} and $\partial q_{\text{true}}/\partial t$. This operator implements the scaling denoted by the overline bar in Equation 3.5 and determines the target forcings S. This operation is built as a combination of a spectral filtering step, and a core spectral truncation operation and resampling operation, $\mathcal{D}$.

When coarsening from a higher resolution grid hr, the operation D passes the filtered data to a coarsened grid with lower resolution lr. For an input resolution hr and an output resolution lr, this operator truncates the 2D-Fourier spectrum to the wavenumbers which are resolved at the output resolution, then spatially resamples the resulting signal for the target size lr. These operators also apply a scalar multiplication to adjust the range of the coarsened values. We define a ratio $\rho \triangleq \text{hr}/\text{lr}$.

For the QG system, the filtering operator C is "Operator 1" as described in [78]. It is a combination of the truncation operator $\mathcal{D}$ with a spectral filter $\mathcal{F}$:

$$C \triangleq \rho^{-2} \cdot \mathcal{F} \circ \mathcal{D}. \tag{3.11}$$

The operator $\mathcal{F}$ is as described in Equation 3.9, except it operates on the lower resolution grid lr.

For each ground truth sample, we recompute spectral time derivatives in a coarsened PyQG model $\partial \hat{q}_{\text{lr}}/\partial t$, and we pass each time derivative to spatial variables and compute the target forcing for this scale:

$$S_{\text{lr}} = C_{\text{lr}}\Big(\frac{\partial q_{\text{true}}}{\partial t}\Big) - \frac{\partial q_{\text{lr}}}{\partial t}.$$

These forcings—at each of the three scales—along with the high resolution variables are stored in the training and evaluation sets for each step.

3.2.2 Kolmogorov Flow System

We also test on a Kolmogorov Flow system which is a single-layer incompressible Navier-Stokes flow with periodic boundary conditions and a periodic forcing.

$$\sin(4y)\hat{x} - 0.1u \tag{3.12}$$

We use an implementation from JAX-CFD configured to have Reynolds number 7000 [45]. Networks predict output quantities S_u and S_v, which are corrections applied to the velocity components.

We generate our ground truth data at a true resolution of 2048×2048 on a domain of size 4π on each side (double the default size of 2π). We configure the system to have a viscosity of $1/3500$ which, with the domain, produces a Reynolds number of 7 000. States are ported to lower resolutions using the `downsample_staggered_velocity` routine in the JAX-CFD package which computes means of the velocity values along a specific face of control volumes composed of groups of grid squares.

3.3 Training and Evaluation

In climate modeling applications, existing models generally do not support automatic differentiation, making online training of neural networks infeasible. Therefore, despite the fact that our testbed models would support it, we do not use online training. Instead we pre-generate training snapshots for each system and train our networks offline.

For both the quasi-geostrophic and Kolmogorov flow systems, we produce trajectories at the fully-resolving "true" resolution, and apply the coarsening operators described above to compute the subgrid forcing values directly as described in Equation 3.5. These snapshots are used for standard supervised training, providing the coarse-grained reference snapshots as inputs to the networks, and computing losses against the reference forcing fields. Our experiments train using mean squared error (MSE) as the target loss, but other standard losses could be used in its place.

After training each network, the networks are combined with the coarsened simulation dynamics and used to roll out a trajectory online, beginning from a saved reference starting time step. That is, we simulate a trajectory following:

$$\dot{\bar{x}} = \bar{f}(\bar{x}) + f_\theta(\bar{x}) \tag{3.13}$$

where f_θ is the neural network implementing the learned subgrid forcing. In our experiments, the learned parameterizations provide additive updates to the dynamics of the coarsened models, effectively performing hybrid derivative prediction as described in Section 1.1. Note that this approach reflects a few modeling assumptions, namely that our forcings are deterministic and Markovian. Other approaches are possible, implementing stochastic parameterizations in one of several schemes, or using a non-Markovian learned parameterization by adding a memory to the network. This could be done, for example, by adding a hidden recurrent memory state managed by the network, or by providing some number of previous steps as a reference.

We also make use of relatively compact, feedforward convolutional networks. Because these networks are integrated as part of an overall climate model, they will have to be evaluated as part of each time step. Larger networks may be more capable of learning complex dependencies between the subgrid forcing and the resolved model states, but are more expensive to deploy and increase the cost of time-stepping an overall system. In one of the following projects (discussed in Chapter 4), we explore an approach which appears to produce better-trained small parameterizations without increasing the cost of evaluation.

3.3.1 Evaluation Metrics

Our test systems model turbulent fluids, and because their evolution is chaotic we do not target reproducing the exact reference trajectory during online evaluation. Measuring performance at test time using a loss against the reference states would be overly punitive. Furthermore, for our climate modeling application we want in particular to correct more statistical properties of the system over long time horizons. To that end, in the works that follow we will mainly measure the quality of an evaluation trajectory using non-snapshot metrics. We provide an overview of these here.

Kinetic energy In both of our test systems, when generating a reference trajectory, we first run the system through a warmup phase until it reaches an equilibrium. As a result, even if an online test trajectory does not pass through the same states, it should generally have a roughly constant level of kinetic energy. We generally plot these as values across time, which allows spotting networks which cause the trajectories to be either under- or over-energized, or potentially unstable.

Decorrelation time We measure the correlation between a derived quantity of the simulation state and the same quantity from the reference trajectory. As we have noted, we do not expect our parameterized model to precisely track the reference trajectory. Even so, we expect the unparameterized model to deviate more quickly due to more significant differences in its behavior. We quantify this by tracking the correlation coefficient between the vorticity fields for evaluation and reference trajectories and report the simulation time at which the correlation drops below 0.5 [45, 78].

Spectral error One metric of particular interest in climate modeling is the distribution of energy across scales [78]. A coarse-grained simulation will have an equilibrium energy distribution which differs from that of the reference trajectory, even over the scales that the two simulations have in common. We want, in particular, for our subgrid forcing to correct this distribution, adjusting the spectral energy distribution to more closely match the reference. We evaluate the performance by first computing a kinetic energy field K for the reference and evaluation trajectories, computing their discrete Fourier transform $\hat{K}$, and then computing the root mean square error between them:

$$\sqrt{\frac{1}{|\mathcal{K}|}\sum_{k\in\mathcal{K}}\left(\hat{K}_{\text{eval}}(k)-\hat{K}_{\text{ref}}(k)\right)^2} \tag{3.14}$$

where $\mathcal{K}$ is the set of all nonzero wavenumbers in the discrete Fourier transform $\hat{K}$.

3.4 Project Overview

In the following two chapters, we present research projects investigating two approaches to improving the accuracy and stability of learned subgrid parameterizations.

In Chapter 4 we present a change to the neural network architecture that more directly incorporates the multiscale structure into the network's inference process. We find that this approach produces networks that are more stable and accurate, particularly for smaller architectures which are able to match the parameterization quality of larger, more expensive networks. The architectural change does meaningfully increase the cost of evaluating the network, and uses the same number of weights.

In Chapter 5 we instead look at the offline training process. Our learned parameterizations are trained offline on snapshots only, but evaluated online in a recurrent fashion. This leads to a significant mismatch between the training and evaluation tasks [23, 26]. While we cannot practically compute gradients through the simulation, we can try to make networks more robust to the sorts of errors they make during inference, improving stability. One method to accomplish this is to inject a small amount of noise during training, to ensure the networks are exposed to corrupted or perturbed data. We extend this approach and use the learned subgrid parameterization itself to perturb the training inputs, ensuring that the "noise" injected during training corresponds to the sorts of errors the networks will encounter during online inference.

Chapter 4

Multiscale Modeling of Climate Parameterizations

In this chapter we introduce a novel neural network architecture for learned subgrid parameterizations. Our architecture is designed to take advantage of the multiscale structure of the subgrid forcing problem, as well as self-similar structures common in turbulent fluids. Our approach first predicts the coarsest resolved scales of subgrid forcing and then, in a separate step, adds finer details. The architecture incorporates spectral coarsening operations, but otherwise incorporates no additional layers with significant cost. Evaluating a neural network modified with our approach should require no greater computational resources than a network that does not make predictions across scales.

In testing on two testbed fluid models, we observe that this approach enables significantly smaller models to attain the accuracy of larger networks, which could enable more efficient architectures in downstream tasks, reducing the cost of deploying a neural network as part of a production model. We also observe improvements in stability. We believe this is due to an increased emphasis on coarser-scale features, reducing reliance on fine-scale details which may not be fully intact after a few recurrent evaluation steps. We believe this approach can support

more efficient and more successful applications of neural networks to subgrid forcing prediction.

4.1 Project Background

Typical deep learning methods across a wide range of application areas make use of end-to-end learning. In such approaches, a neural network is trained such that it receives feedback which matches the requirements of the full task. However, in some applications this sort of training is impractical or even impossible. Applications of learning to scientific computing tasks—in particular to simulation problems—frequently involve real-world dynamics which may not be fully modeled or understood, or existing simulation software which is difficult to integrate with learned models and which does not admit backpropagation through simulation time steps. In these cases, training uses strictly offline data (either pre-computed or derived from real-world observations), while applying the network online after training. As a result, the network may learn behaviors which are successful on the offline training task, but which are unstable in online evaluation [26].

In this work we examine an approach to decomposing prediction tasks into separate prediction problems across scales, encouraging the network to more completely exploit available multiscale information. Networks making separate cross-scale predictions are trained separately and *not* end-to-end. We observe that this approach to offline training improves model stability, and improves accuracy in smaller architectures, allowing more efficient evaluation when integrated into end applications. While we believe that our approach may have wider applicability, we evaluate its performance on a set of subgrid forcing prediction tasks which arise in real-world climate modeling applications.

Climate models, which simulate the long-term evolution of the Earth's atmosphere, oceans, and terrestrial weather, are critical tools for projecting the impacts of climate change around the globe. Due to limits on available computational resources, these models must be run at a

coarsened spatial resolution which cannot resolve all physical processes relevant to the climate system [24]. To reflect the contribution of these subgrid-scale processes, closure models are added to climate models to provide the needed subgrid-scale forcing. These parameterizations model the contribution of these fine-scale dynamics and are critical to high quality and accurate long term predictions [25, 78]. A variety of approaches to designing these parameterizations have been tested, ranging from hand-designed formulations [91], to modern machine learning with genetic algorithms [78], or neural networks trained on collected snapshots [29, 58, 70, 110], or in an online fashion through the target simulation [26].

The problem of predicting these forcings is inherently multiscale; the subgrid dynamics which must be restored represent the impact of the subgrid and resolved scales on each other. Closure models for climate are designed to be resolution-aware [38], but even so existing deep learning subgrid models do not explicitly leverage the interactions between scales, leaving it to the neural networks to implicitly learn these relationships. Our approach makes these interactions explicit.

4.2 Approach

Recent methods for image generation have made use of diffusion modeling where an image is sampled from a learned distribution over several steps starting from noise, rather than training a network to produce the result in one shot [92, 108]. One can view this process as gradually filling in fine-scale details based on earlier coarse-scale features. This approach has found a wide variety of successful applications. Many simulation tasks, in which states must be evolved in time, have dynamics which might benefit from this approach—namely dynamics in which neighboring scales influence each other and in which coarser-scale features may be easier to predict and slower to evolve than finer-scale details.

While these approaches have been successful and provide a useful inductive bias, this sampling method is generally quite expensive. Many applications of machine learning to existing simulation

tasks insert a learned model as a *component* of an existing simulation and evaluate the learned model in each simulation step. In these applications, further adding multiple diffusion steps may be prohibitively costly. In this work, we try to keep many of the benefits of this generative approach while reducing the cost of evaluation by reducing the number of inference steps. More specifically, we divide the state into two scale ranges: a "high resolution" segment containing the fine scale details and a "low resolution" segment containing only the coarse scales. Our prediction process first produces a coarse, low resolution version of the target field, then a second step uses this initial prediction to fill in the fine scale details, yielding a high resolution output.

In particular, when predicting a field S_{x} we can try to predict the quantity directly, depending on the current simulation state x. These quantities are often uncertain, but for our purposes, we will use deterministic models to predict the expectation of our target quantities. However, our approach could be extended to stochastic models in several natural ways. A deterministic neural network f_θ can be trained to perform this task directly:

$$f_\theta(\mathsf{x}) \approx \mathrm{E}[S_{\mathsf{x}}|\mathsf{x}] \tag{4.1}$$

In our multiscale approach we first predict a low resolution version of the target field, $S_{\mathsf{x}\ \mathrm{lr}}$ and condition on this prediction while producing the full resolution output:

$$f_{\theta_1}^{\text{downscale}}(\mathsf{x}) \approx \mathrm{E}[S_{\mathsf{x}\ \mathrm{lr}}|\mathsf{x}] \tag{4.2}$$

$$f_{\theta_2}^{\text{buildup}}(\mathsf{x}, S_{\mathsf{x}\ \mathrm{lr}}) \approx \mathrm{E}[S_{\mathsf{x}}|\mathsf{x},\ S_{\mathsf{x}\ \mathrm{lr}}] \tag{4.3}$$

We use the estimated expectation from $f_{\theta_1}^{\text{downscale}}(\mathsf{x})$ in Equation 4.2 to provide a realization of $S_{\mathsf{x}\ \mathrm{lr}}$ in Equation 4.3. Further scale segments could be introduced if desirable for a particular task. For our application, the low resolution field is produced by low-pass filtering the full target, and resampling at a lower resolution, yielding a field with smaller dimensions. That is, $S_{\mathsf{x}\ \mathrm{lr}} \triangleq \mathcal{D} \circ \mathcal{F}(S_{\mathsf{x}})$

for a low-pass spectral filter $\mathcal{F}$ and a resampling operation $\mathcal{D}$.

To evaluate our multiscale approach, we consider the problem of learning subgrid forcings for fluid models, a problem which arises in climate modeling applications. In particular, we use two idealized simulations, as introduced in Section 3.2: (1) a two layer quasi-geostrophic model *QG*, and (2) Kolmogorov flow *KF*. Each target simulation autonomously evolves a set of state variables through time and can be evaluated with a configurable grid resolution. We refer to a general state variable x in the following introduction. In each system states may be ported to lower resolutions by coarse-graining and filtering.

For each system we generate ground truth data by running the model at a very high ("true") resolution. This produces trajectories $\mathrm{x}_{\text{true}}(t)$ and time derivatives $\partial \mathrm{x}_{\text{true}}(t)/\partial t$. Next we generate training data at a high resolution by applying a system-dependent coarsening and filtering operator C giving variables $\bar{\mathrm{x}} \triangleq C(\mathrm{x})$. Given nonlinearities in the target simulations, this coarsening does not commute with the dynamics of the models. To correct for this we must apply a subgrid forcing term S_{x} to the evolution of each state variable:

$$S_{\mathrm{x}} \triangleq \overline{\frac{\partial \mathrm{x}}{\partial t}} - \frac{\partial \bar{\mathrm{x}}}{\partial t}. \tag{4.4}$$

Note that formally the forcing S_{x} is a function of the state x_{true}. In a climate modeling application we do not have access to this variable and so we train a model $f_\theta(\bar{\mathrm{x}}) \approx S_{\mathrm{x}}$ which may be stochastic.

We continue this process, introducing another downscaling[1] operator D and upscaling D^+. See Equation 4.7 for the full definition. Taking $\mathrm{x}_{\text{hr}} \triangleq \bar{\mathrm{x}}$ as our high resolution samples, we produce low resolution samples $\mathrm{x}_{\text{lr}} \triangleq D(\mathrm{x}_{\text{hr}})$ and $S_{\mathrm{x}\,\text{lr}} \triangleq D(S_{\mathrm{x}})$. This allows a decomposition $\mathrm{x} = D^+ D\mathrm{x} + \mathrm{x}'$ where x' are the details removed by D. Our experiments thus involve three resolutions, from fine to coarse: a "true" resolution; a high resolution, hr; and a low resolution, lr. The closures S_{x} try to update hr to match the "true" resolution.

[1]We use "downscale" and "downscaling" to refer to *coarsening* a target variable, removing finer scales.

Just as predicting S_x from x_{true} is fully deterministic, while predicting it from x_{hr} involves uncertainty, we anticipate a similar trend to hold for $D(S_x)$. In other words, predicting $D(S_x)$ from x_{hr} should be easier than predicting $D(S_x)$ directly from x_{lr}. Then, using this coarse-grained prediction $D(S_x)$ as a foundation, we can learn to predict only the missing details and add them. This process splits the problem of predicting S_x into two phases: (1) a "downscale" prediction to form $D(S_x)$, and (2) a "buildup" prediction combining x_{hr} and $D(S_x)$ to predict S_x, adding the missing details. This decomposition takes advantage of self-similarity in the closure problem to pass information between the coarse and fine scales and improve predictions.

4.3 Experiments

To test this approach to predicting subgrid forcings we compare our *multiscale* approach against single-scale baselines. We select the "high" resolution size in each target system (QG or KF) so that the system requires closure (there are sufficient dynamics below the grid-scale cutoff), but does not diverge [78]. Further details on each system are provided in Section 3.2.

4.3.1 Test Systems

We carry out our experiments on two systems: a two-layer quasi-geostrophic system, and a single layer Kolmogorov Flow system which is a configuration of Navier-Stokes as described in Section 3.2. Using these systems we carry out two sets of experiments: a small set of "separated" experiments carried out on the QG model only (these neural networks are trained and evaluated offline both with and without access to the additional multiscale information); and "combined" experiments which run on both the QG and KF models which provide an implementable closure model trained end to end. In both cases we compare results against networks of equivalent architectures without the additional multiscale structure.

For each experiment, a set of feedforward convolutional neural networks is trained and

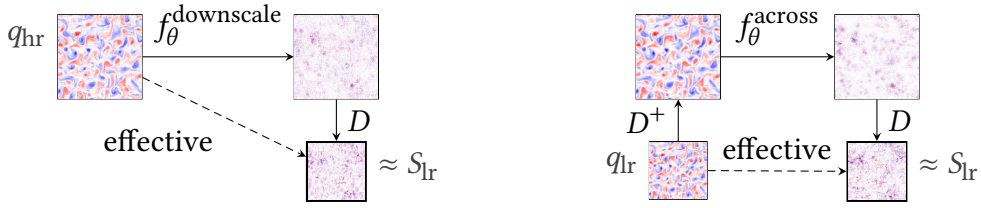
evaluated separately. We train several independently-initialized networks to capture the variance due to initialization. These experiments also compare the performance of two architectures, a "small" architecture and a "large" architecture which are modifications of networks used in past research [29]. The small architecture for the combined experiments was chosen as the result of an architecture search, discussed below. Results are included in Section 4.4, and information on the network architectures and training procedure is included in Appendix B.

4.3.2 Separated Experiments

We first carry out a set of preliminary tests on the quasi-geostrophic system only. In these experiments, we examine the accuracy of the learned forcings offline (that is, without rolling out trajectories) while separating the two steps in our multiscale prediction process. In particular, we examine the ability of networks to learn the downscale and buildup prediction tasks, and check for the advantages we intuitively expect from the additional multiscale information.

In these experiments, we train neural networks separately to predict quantities between different scales. In particular we train "downscale" networks which predict only the low-resolution components of the target forcing quantity while observing a high resolution state, and "buildup" networks which work in the opposite direction, predicting higher-resolution forcing details with access to the low-resolution forcing. These illustrate some of the advantage provided by the additional information and measure performance on snapshots only (offline testing) as these separated networks do not provide a fully-implementable closure model, since they require access to an oracle to provide the additional input features.

Because these experiments target the QG system the target quantity is the potential vorticity q. Each trained network receives a q input at the active (high resolution) simulation scale and predicts an S_q output at the target scale.



(a) Downscale prediction

(b) Across prediction

Figure 4.1: Downscale vs. across separated prediction tasks. The networks referenced in Equation 4.5 are combinations of an inner network f_θ with the fixed rescaling operators D, D^+. The overall prediction is indicated with a dashed line.

Downscale Prediction

We compare the task of predicting $S_{\text{lr}} \triangleq D(S_{\text{hr}})$ with access to high resolution information q_{hr} or restricted to low resolution q_{lr}. This provides an estimate of the advantage gained by predicting the target forcing with access to details at a scale finer than that of the network's output. We train two networks f_θ with the same architecture to perform one of two prediction tasks:

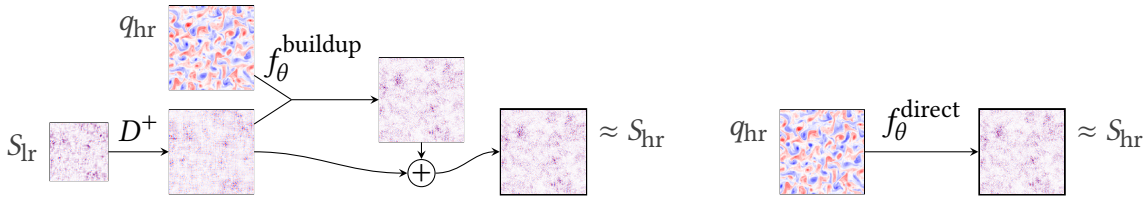
$$D \circ f_\theta^{\text{downscale}}(q_{\text{hr}}) \approx S_{\text{lr}} \qquad \text{and} \qquad D \circ f_\theta^{\text{across}} \circ D^+(q_{\text{lr}}) \approx S_{\text{lr}}. \tag{4.5}$$

To ensure that the convolution kernels process information at the same spatial size, and differ only in the spectral scales included, we first upsample all inputs to the same fixed size using a spectral upscaling operator D^+ described below. The full prediction process including the re-sampling operators is illustrated in Figure 4.1.

Buildup Prediction

We also test a prediction problem in the opposite direction, predicting finer-scale details with access to lower-resolution predictions, similar to a learned super-resolution process used in recent generative modeling works [35, 90]. We train neural networks:

$$f_\theta^{\text{buildup}}\big(q_{\text{hr}},\, D^+(S_{\text{lr}})\big) \approx S_{\text{hr}} - D^+(S_{\text{lr}}) \qquad \text{and} \qquad f_\theta^{\text{direct}}(q_{\text{hr}}) \approx S_{\text{hr}}\,, \tag{4.6}$$



(a) Buildup prediction (b) Direct prediction

Figure 4.2: Buildup vs. direct separated prediction. The networks in Equation 4.6 are combinations of the networks f_θ with the indicated fixed operations. In Figure 4.2a f_θ predicts the details which are combined with S_{lr} from an oracle.

where $S_{\text{hr}} - D^+(S_{\text{lr}})$ are the details of S_{hr} which are not reflected in S_{lr}. The additional input S_{lr} is given by an oracle using ground truth data in the training and evaluation sets.

This experiment estimates the value in having a high-quality, higher-confidence prediction S_{lr}, in addition to q_{hr}, when predicting the details of S_{hr}. That is, the experiment estimates the value in starting the prediction of S_{hr} by first locking in a coarse-grained version of the target, and separately enhancing it with finer-scale features. The two prediction tasks are illustrated in Figure 4.2.

Rescaling Operator

In each multiscale prediction task, we apply the same scaling operator D (with associated upscaling D^+). This operator is built around a core spectral truncation operation, $\mathcal{D}$. For an input resolution hr and an output resolution lr, this operator truncates the 2D-Fourier spectrum to the wavenumbers which are resolved at the output resolution, then spatially resamples the resulting signal for the target size lr. These operators also apply a scalar multiplication to adjust the range of the coarsened values. Below, ρ is the ratio of the scale dimensions $\rho \triangleq \text{hr}/\text{lr}$.

$$D \triangleq \rho^{-2}\mathcal{D} \qquad \text{and} \qquad D^+ \triangleq \rho^2\mathcal{D}^T. \tag{4.7}$$

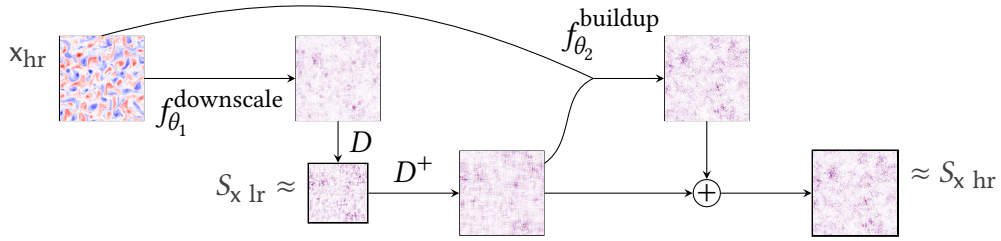
Note that D^+ is a right side inverse $DD^+ = I$, and that D^+ is the pseudoinverse $D^+ = D(DD^T)^{-1}$ because $\mathcal{D}D^T = I$. This operator omits the filtering $\mathcal{F}$ performed as part of coarsening operator C to avoid numerical issues when inverting the additional spectral filtering. This operator was used for the multiscale experiments with both QG and KF systems and is the operator referenced in Figure 4.1 and Figure 4.2.

4.3.3 Combined Experiments

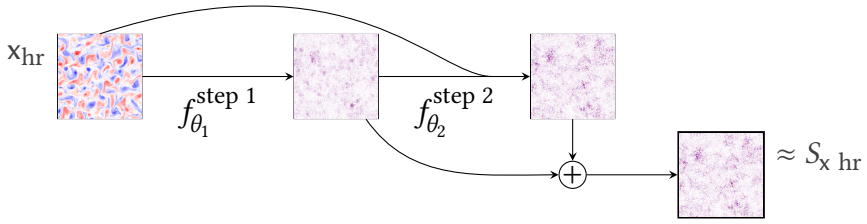
In these experiments, we combine the networks trained in the "downscale" and "buildup" experiments, passing the downscale prediction as an input to the buildup network. This removes the oracle providing lower resolution predictions used to train the separate networks. In each test, we choose two scale levels and first predict a coarsened version of the subgrid forcing at the lowest resolution, then gradually enhance it with missing scales using the buildup process discussed above. These tests are carried out for both the quasi-geostrophic and Kolmogorov flow systems. The overall flow of this combined approach is illustrated in Figure 4.3 along with the associated baseline architecture.

Because this configuration yields an implementable closure model, we perform our evaluations *online*. That is, while the networks are trained on snapshots, we evaluate the accuracy and stability of the forcing by rolling out multiple trajectories using the trained networks. As in the separated experiments we consider two network architectures, a "large" architecture based on other works and a "small" architecture with fewer layers and smaller convolution kernels. The small architecture was based on the results of an architecture search discussed in Appendix B. This allows us to compare how architectural capacity may affect the behavior and benefits of our multiscale approach.

For these experiments we retrain new neural networks building out the training pipeline sequentially. We first train the first, downscale, network, and then fix its weights and use its outputs to train the subsequent buildup network. In this way, later networks see realistic inputs



(a) Full combined prediction flow



(b) Structure of the baseline for the combined prediction task

Figure 4.3: Flow for the combined prediction experiments. For the full combined flow each network f_θ is trained sequentially, and earlier weights are frozen and used to train networks later in the pipeline. The overall flow no longer requires an oracle for any additional inputs. Note also that the networks in the baseline flow in Figure 4.3b have the same residual prediction structure. The baseline network is trained end-to-end and differs only in that it misses the additional supervision from the multiscale training. The variable x represents the differing scalar fields for the QG and KF systems.

during training rather than unrealistically clean data from a training set oracle.

For a combined prediction across two scales lr and hr, we predict $S_{\text{x hr}}$ from only x_{hr} following the procedure below:

$$\begin{aligned} \tilde{S}_{\text{x lr}} &\triangleq D \circ f_{\theta_1}^{\text{downscale}}(\text{x}_{\text{hr}}) \approx S_{\text{x lr}} \\ S_{\text{x hr}} &\approx f_{\theta_2}^{\text{buildup}}\big(\text{x}_{\text{hr}},\, D^{+}(\tilde{S}_{\text{x lr}})\big) + D^{+}(\tilde{S}_{\text{x lr}}). \end{aligned} \tag{4.8}$$

The quantity $\tilde{S}_{\text{x lr}}$ is an approximate neural network output used in subsequent predictions. Equation 4.8 composes the prediction tasks described in Equation 4.5 and Equation 4.6. See Figure 4.3a for an illustration of the above flow. Each network f_θ is trained separately against either $S_{\text{x lr}}$ or $S_{\text{x hr}}$ as appropriate.

The networks are evaluated on their ability to produce a stable trajectory when rolled out, and their ability to correct the energy spectrum—adding and removing energy across spectral scales as needed to correct issues from coarse-graining. For these experiments, we examine the trends in the system's total kinetic energy across time and distributions in the spectral error of the system, computed by adding errors in the average kinetic energy spectrum of a trajectory. These measurements are distinct from those used to compare the quality of snapshots used for the offline tests conducted as part of the separated experiments.

4.4 Results

In this section we describe the results of our experiments and measurements. Details of the approach taken in each experiment are provided in Section 4.3.

4.4.1 Separated Experiments

For both the downscale and buildup prediction tasks, we train three neural networks. Once trained, we evaluate their performance on a held out evaluation set measuring performance with three metrics: a mean squared error (MSE), a relative ℓ_2 loss, and a relative ℓ_2 of the spectra of the predictions.

The MSE is a standard mean squared error evaluated over each sample and averaged. The other two metrics are derived from previous work evaluating neural network parameterizations [70] (where they were called $\mathcal{L}_{\text{rmse}}$ and $\mathcal{L}_{\text{S}}$). The metrics in this previous work were originally designed to measure performance for stochastic subgrid forcings. Here we use the two metrics from that work which do not collapse to trivial results for deterministic models. These are defined as:

$$\text{Rel } \ell_2 \triangleq \frac{\|S - \tilde{S}\|_2}{\|S\|} \qquad \text{and} \qquad \text{Rel Spec } \ell_2 \triangleq \frac{\|\text{sp}(S) - \text{sp}(\tilde{S})\|_2}{\|\text{sp}(S)\|_2} \tag{4.9}$$

where S is the true target forcing, $\tilde{S}$ is a neural network prediction being evaluated, and sp is the isotropic power spectrum. See `calc_ispec` in PyQG for calculation details [1]. Each of these three metrics is averaged across the same batch of 1024 samples selected at random from the set of held out trajectories in the evaluation set.

Table 4.1 shows the results for the downscale experiments, comparing against "across" prediction which accesses only coarse-scale information. In these results we observe an advantage to performing the predictions with access to higher-resolution data (the "downscale" columns), suggesting potential advantages and a decrease in uncertainty in such predictions.

Results for experiments examining prediction in the opposite direction—predicting a high-resolution forcing with access to a low-resolution copy of the target from an oracle—are included in Table 4.2. We also observe an advantage in this task from having access to the additional information. The low resolution input in the buildup experiments yields lower errors on average at evaluation. This advantage is greater when the additional input is closer in scale to the target

NN Size	Metric	128 → 96		128 → 64		96 → 64	
		Downscale	Across	Downscale	Across	Downscale	Across
Small	MSE	0.054	0.072	0.002	0.006	0.032	0.058
	Rel ℓ_2	0.317	0.364	0.394	0.629	0.348	0.469
	Rel Spec ℓ_2	0.133	0.145	0.154	0.471	0.154	0.254
Large	MSE	0.038	0.057	0.002	0.006	0.024	0.052
	Rel ℓ_2	0.259	0.316	0.335	0.595	0.297	0.436
	Rel Spec ℓ_2	0.092	0.129	0.125	0.443	0.103	0.212

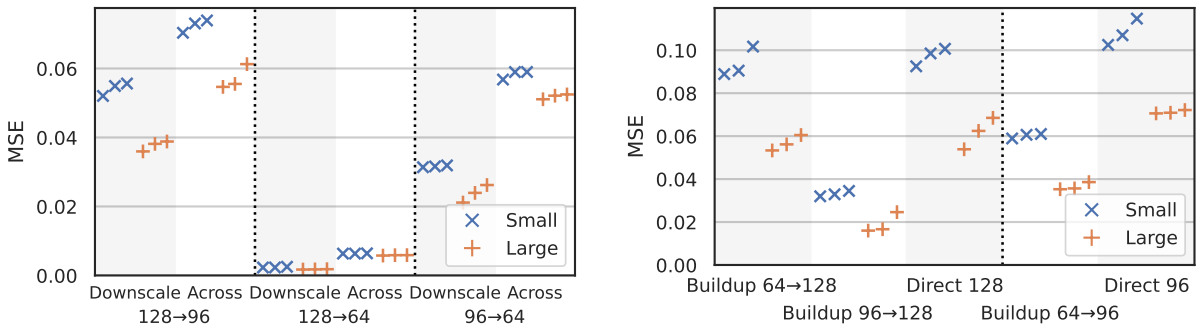
Table 4.1: Evaluation results for downscale vs. across generation. In all metrics, lower is better. The numbers in the first row of the table heading show the different scales involved in both prediction tasks. The results contributing to the MSE averages in this table are illustrated in Figure 4.4a.

NN Size	Metric	Buildup 64 → 128	Buildup 96 → 128	Direct 128	Buildup 64 → 96	Direct 96
Small	MSE	0.094	0.033	0.097	0.060	0.108
	Rel ℓ_2	0.314	0.187	0.319	0.251	0.333
	Rel Spec ℓ_2	0.138	0.054	0.139	0.095	0.162
Large	MSE	0.057	0.019	0.062	0.037	0.071
	Rel ℓ_2	0.242	0.141	0.251	0.195	0.268
	Rel Spec ℓ_2	0.074	0.029	0.084	0.041	0.091

Table 4.2: Evaluation results from buildup vs. direct experiments. In all metrics, lower is better. The numbers in the second row of the table heading show the different scales involved in both prediction tasks. The results contributing to the MSE averages in this table are illustrated in Figure 4.4b.

output. The predictions building up from 96×96 to 128×128 have lower errors than those which access an additional 64×64 input. This is not unexpected given that the input with nearer resolution resolves more of the target value, leaving fewer details which need to be predicted by the network.

The results for both separated experiments (those reported in Table 4.1 and Table 4.2) for the MSE metric are illustrated in Figure 4.4.



(a) Downscale vs. across MSE

(b) Buildup vs. direct MSE

Figure 4.4: Evaluation results from both of the separated experiments for the MSE metric. These are the same numbers which are reported as averages in Table 4.1 and Table 4.2. The plot here shows the three samples—one from each trained network—used to compute the means.

4.4.2 Combined Experiments

We also carry out tests comparing the performance of networks using our multiscale prediction approach to a network predicting only at one scale throughout. Figure 4.3 illustrates the overall flows of these networks. In the combined task, each independent cross-scale network is trained separately in phases. Earlier networks in the pipeline are trained and have their weights frozen and these are used to produce inputs during the training of networks later in the pipeline. The baseline for this configuration is trained end-to-end and has the same residual structure and number of weights as the multiscale network, but without any enforced predictions across scales.

This setting produces a trained parameterization network which can be applied to a real simulation and tested online. As a result, we evaluate these networks by rolling out trajectories from a held out test set. These are compared against a reference trajectory which was originally produced at a "true" resolution. Because these systems are chaotic we do not expect to reproduce exact states in these trajectories, particularly over the long time horizons involved in climate modeling. Consequently, we do not examine snapshot errors over long time horizons. Instead we compare statistical properties of these trajectories including their total kinetic energy (which gives a sense of stability), errors in the trajectory's energy spectrum (providing a sense of the

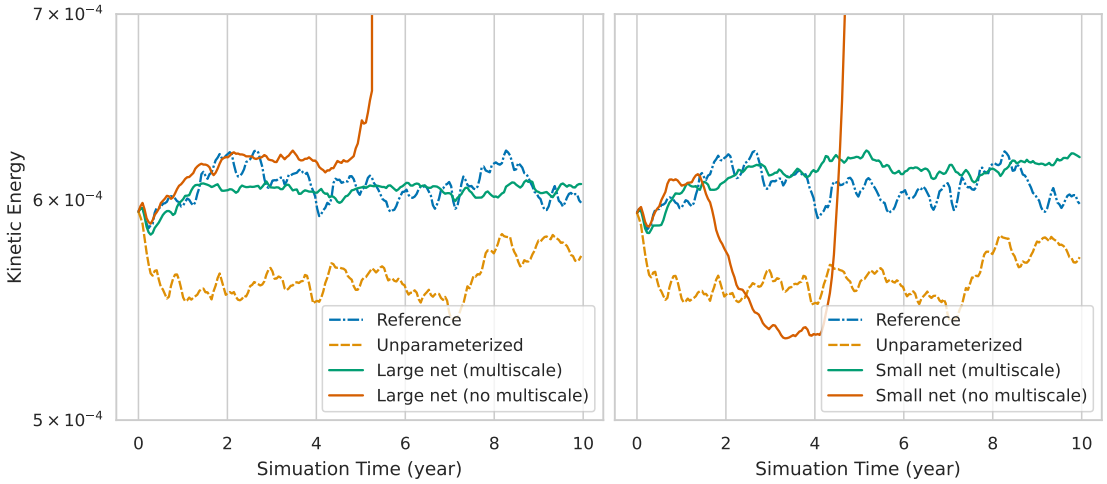
quality of the parameterization), and for the KF system, the vorticity decorrelation time. Overall, we find that the multiscale approach improves the stability of the learned parameterizations while also permitting smaller neural network architectures to be used for these tasks. Results of our experiments for the QG and KF systems are described below.

Quasi-Geostrophic Results

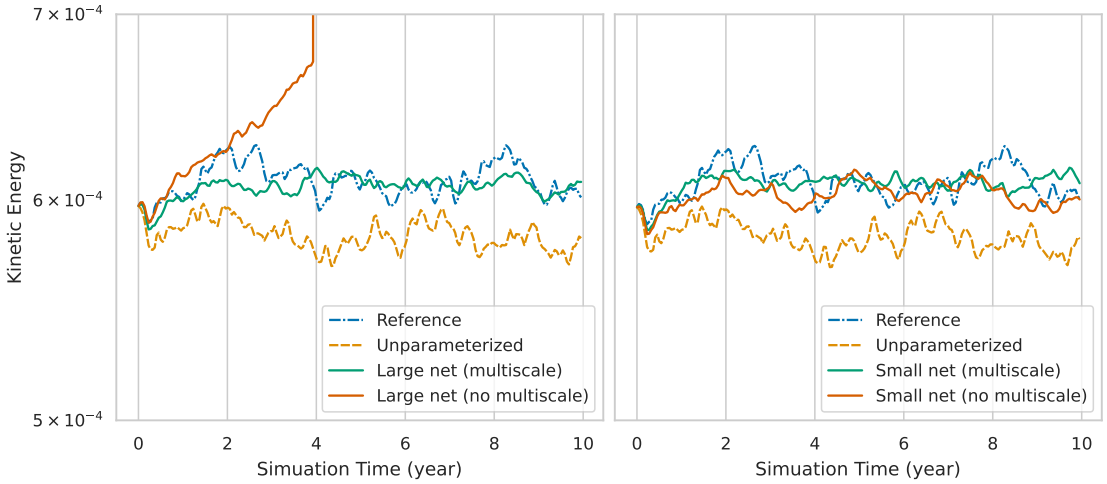
For the QG system we simulate the evolution of a held out trajectory and conduct tests over four independently trained neural networks and 16 held out trajectories for each network. For these tests we are concerned with two aspects: the stability of the system running with the neural network closure, and the quality of the resulting parameterization—in particular the extent to which it improves the energy spectrum of the trajectory.

Figure 4.5 shows variation in the mean kinetic energy over time. We intend this as a rough measure of the stability of the parameterized system providing a rough idea of whether the system is under- or over-energized which can lead to instability and eventual collapse in the system. The QG system is simulated at a grid size of 96×96 and 128×128 with separate sets of neural networks trained for each case. The simulations at a size of 96 have a greater range of unrealized dynamics leading to a more challenging closure problem. We note in particular that the trajectories which display instability are those using networks without the multiscale component. The smaller networks for the system at size 128 all display general kinetic energy stability; however, with only kinetic energy statistics this is difficult to distinguish from a network which applies no parameterization at all. The QG system is stable without a parameterization due to a fixed spectral filter which attenuates high frequencies, and the target parameterization values have zero mean which can in some cases lead a network to learn to apply no correction which still yields a stable trajectory.

To distinguish these cases we also examine the spectral error of these trajectories, distributions of which are plotted in Figure 4.6. We also run trajectories with varying values for α which



(a) Scale 96—multiscale networks predict between 96 ← 64



(b) Scale 128—multiscale networks predict between 128 ← 96

Figure 4.5: Time evolution of mean kinetic energy for the QG system simulated at different scales. Trajectories are simulated for 16 held out trajectories using 4 independently-trained networks.

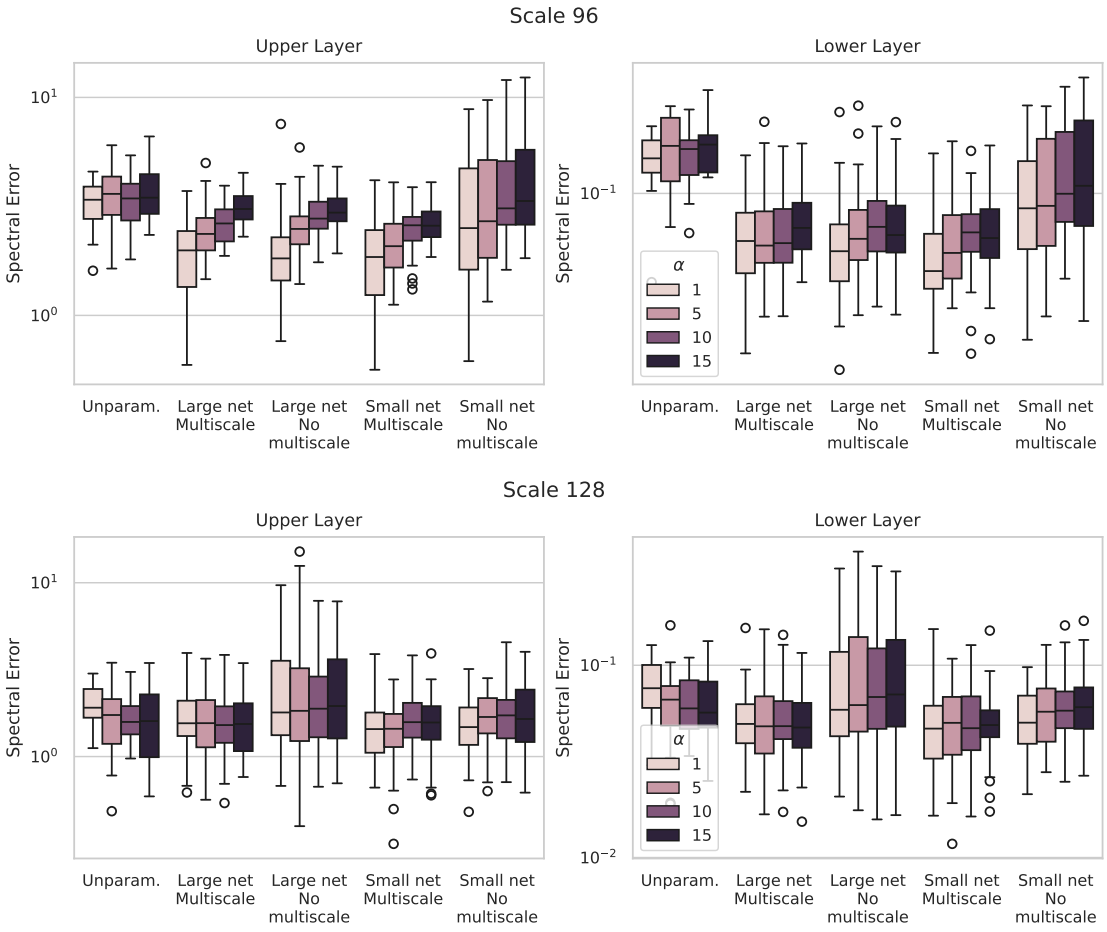


Figure 4.6: Spectral errors for trained architectures averaged over 5 400 steps. Increasing α values reduce the sharpness of the QG model's filter, which in turn reduces built-in model stability. In each row, the separate panels show values for the two layers of the QG system.

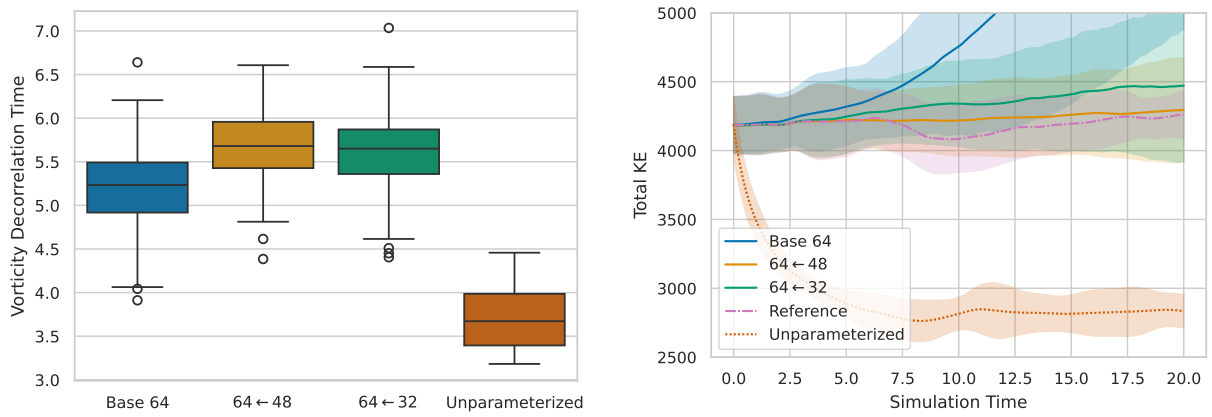
controls the sharpness of the QG model's internal spectral filter (see Equation 3.9). Higher values of α reduce the attenuation of higher frequencies, making the QG model less stable and making errors in the parameterization more evident over time. The results in this figure suggest that the multiscale training generally improves model performance. For large architectures on the system at scale 128, multiscale training reduces instabilities. For the results on scale 96—where more parameterization is required—adding multiscale training allows the small architecture to achieve the same results as the large architecture, allowing for a smaller more efficient network to yield a more accurate and stable parameterization.

Kolmogorov Flow Results

In addition to our experiments on the QG model we also test our method on a Kolmogorov Flow (KF) system. Unlike the QG model, the KF system does not include an internal filter. This results in a simulation which can be highly unstable, and many of the learned parameterizations we trained and tested did not successfully complete a stable trajectory. To reduce this problem we modify our training procedure to inject Gaussian noise to inputs in order to encourage the learned parameterizations to be stable to corruptions in the system states. This noise has mean zero and a configurable scale parameter β. Further discussion of our experiments adjusting this parameter are included later in this section.

Using the best calibrated noise scales, we train a series of small architecture networks both with multiscale training and with the single scale baseline architecture. The large architectures displayed very high instability irrespective of the noise level. As a result further results on the KF in this section are produced for the small architecture only. An illustration of this noise calibration issue is included in Appendix C. For each of these networks we roll out a series of trajectories online. Figure 4.7 shows the results of this online testing for the KF experiments. Results are averaged across 10 trained networks of each type and 16 held out test trajectories for each.

Figure 4.7a reports the observed decorrelation time in the vorticity channel ω of the KF system.



(a) Vorticity decorrelation times from online experiments for the KF system. A longer decorrelation time generally reflects a better parameterization.

(b) Kinetic energy evolution. Solid lines are means over all test trajectories and trained networks and shaded area is a 1σ region.

Figure 4.7: Results of online experiments for Kolmogorov Flow (KF) system. Values are computed over a collection of 10 trained networks evaluated on 16 reference trajectories each.

Results are in simulation time (out of a full trajectory length of 70.0). Longer decorrelation times reflect a parameterization which better reproduced the statistical properties of the reference trajectory even though, due to the chaos in these systems, the trajectories will eventually decorrelate. We note that the multiscale runs have a longer decorrelation time than both the baseline network and the unparameterized system with the multiscale network using two scales closer in size having a slightly better performance.

Trends in the evolution of kinetic energy over time are plotted in Figure 4.7b. The lines in this figure are the mean kinetic energy across the test trajectories and networks and the shaded regions are a 1σ range. In these results, the multiscale networks remain more stable over time and show smaller variance at later time steps. All networks prevent the significant energy loss of the unparameterized trajectories even though the baseline network displays significant instability.

The results above were produced using separately chosen values of the training noise level β, one noise level for each network architecture and set of predictino scales. The values of β were set as a fraction of the empirical standard deviation of input values from the training set. Higher noise levels injected during training reduce instability but decrease parameterization accuracy,

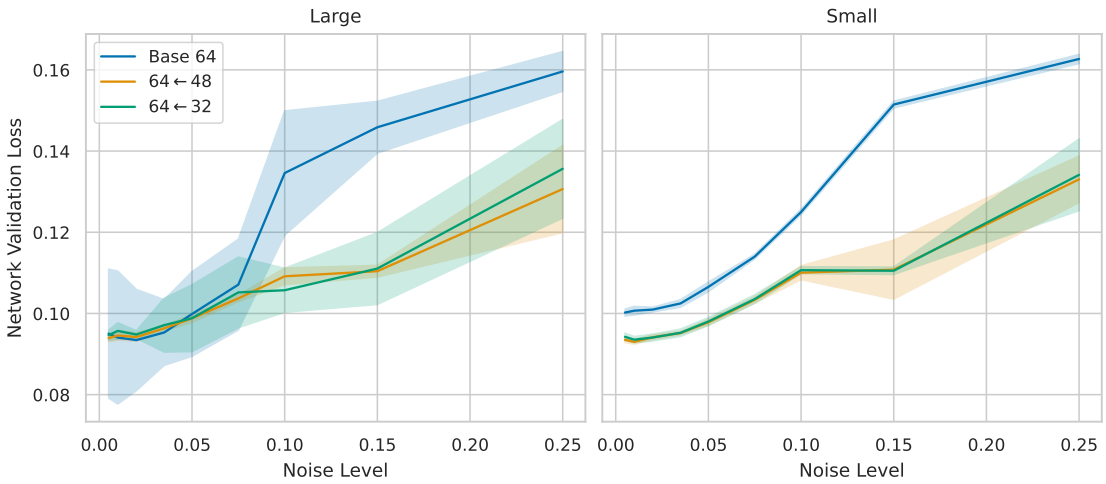


Figure 4.8: Offline noise calibration for the Kolmogorov Flow (KF) system. Solid lines are a mean validation loss observed during network training and shaded regions show a 1σ range. The large architecture shows significantly greater variance than the small architecture. Increasing noise scale during training increases the validation loss but can improve robustness against noise during online rollouts.

while lower noise levels fail to correct the problem of instability during online rollouts.

Figure 4.8 and Figure 4.9 show measurements used in this calibration. We select our target noise level based in part on the offline network validation losses and online kinetic energy errors observed after training. We also show results for a system scale of size 64 as larger state sizes did not have sufficient unresolved dynamics to close on the KF system. Each network is provided the two velocity components u, v as well as a vorticity ω computed from these two.

Based on the results in Figure 4.8 we see that higher noise levels reduce variance in offline validation quality, but can increase instabilities in online testing as evidenced by the kinetic energy errors in Figure 4.9. As a result we selected a noise level of 0.035 for baseline runs, 0.075 for multiscale runs between scales $64 \leftarrow 48$ and 0.1 for $64 \leftarrow 32$ runs (the lowest kinetic energy error point in each curve). These values were used to train the networks used in the further KF experiments discussed earlier in this section.

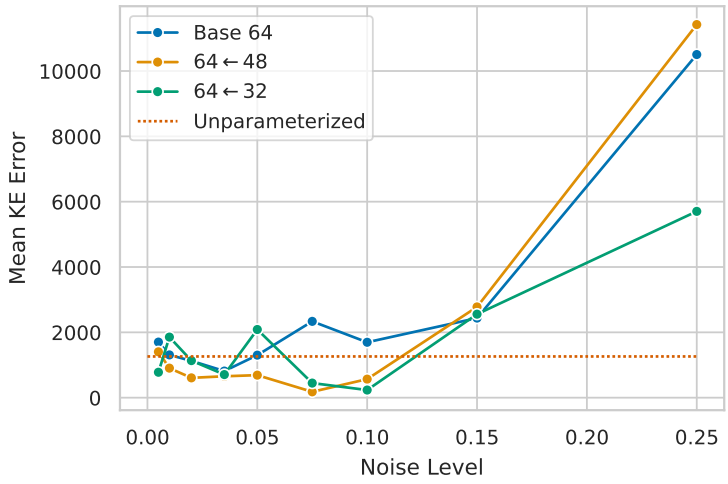


Figure 4.9: Online noise calibration for the KF system (small architecture only). Increasing noise scale eventually reduces the networks' ability to correct kinetic energy. We select the lowest online kinetic energy error point on each curve for each network as the noise scale used during training.

4.5 Conclusion

Our experiments in this work illustrate the potential advantages resulting from decomposing the subgrid forcing problem into one across scales. In particular, our results show that this decomposition improves stability and accuracy, especially for smaller network architectures. Such improvements could support the deployment of machine learning methods to tasks which are constrained by available computational resources, as is common in climate applications. Our results show improvements made possible by structuring prediction tasks to expose important structures of the task. For the fluid problems considered here, and in other tasks, a multiscale decomposition is natural and makes use of links between scales in the model dynamics, and better handles underlying uncertainty in the parameterization task.

In addition to using a multiscale decomposition in future learned parameterizations, future work could explore other applications of this approach. In particular, this decomposition could be advantageous for stochastic parameterizations; perhaps using a deterministic downscale prediction as a foundation for later stochastic or generative outputs or further integrating the multiscale

prediction into the network architecture to realize greater efficiency.

Chapter 5

Live Offline Resampling for Stability

Despite recent projects to develop new models in frameworks supporting automatic differentiation [88, 103] most production climate and ocean models do not support computing gradients through simulation steps [27]. As a result, neural network models trained to interact with these simulations in online rollouts must nevertheless be trained offline on ground truth snapshots [76]. This results in a fundamental mismatch between the task at training time, and the task at evaluation. In particular, networks trained offline are often only exposed to clean ground truth data and never to errors which will certainly accumulate over multiple time steps when rolling out a trajectory. For many tasks, this can lead to instability as the neural networks may produce lower accuracy results when exposed to these out-of-distribution samples, compounding the problem by pushing the states further out of distribution [26, 55].

One approach which has long been used is to regularize training by adding noise to training inputs [8, 21, 89]. This approach has been used in a variety of learned parameterization works and has been applied to simulation problems more broadly [5, 71, 93]. In this approach a dataset

D_{base} of input and target pairs (x, y) is augmented to include additional samples

$$D_{\text{base}} = \left\{(x_i,\ y_i)\right\}_{i=1}^{N} \tag{5.1}$$

$$D_{\text{noise}} = \left\{(x_{b(i)} + \epsilon_i,\ y_{b(i)})\right\}_{i=1}^{N'} \tag{5.2}$$

where the datasets have N and N' samples, respectively, and each ϵ_i is an independently sampled noise mask. Often a single data point in D_{base} may be used multiple times with separate masks ϵ. The mapping b selects the base sample from D_{base}. Depending on the task, different distributions for ϵ may be chosen; however, a common choice is an isotropic Gaussian $\epsilon \sim \mathcal{N}(0, \sigma^2 I)$ with the variance tuned for the specific problem. The augmentation detailed in Equation 5.2 is the approach used in Section 4.4.2 to stabilize the learned parameterizations for the Kolmogorov Flow problem. In numerical simulation problems, if a network has been trained to apply additive updates to the right-hand side function of a model (that is $y_i = \dot{x}_i$, see Section 2.3) we might also want to model outputs to *correct* for these perturbations and drive the simulation to remove them:

$$D_{\text{correcting}} = \left\{(x_{b(i)} + \epsilon_i,\ y_{b(i)} - \alpha\epsilon_i)\right\}_{i=1}^{N'} \tag{5.3}$$

where α is a scaling parameter affecting how aggressively the noise is removed. It is often chosen as a fraction of the time step of the system's numerical integrator, so that the network is trained to remove these errors over a chosen number of time steps [71].

The structure of errors produced by rolling out a simulation with a neural network parameterization does not follow a Gaussian distribution. Furthermore, these errors may also be dependent on the simulation states themselves, with certain simulated behaviors leading to higher errors than others. One method we consider here to capture this effect is to perturb the training inputs using the neural network itself. That is, given a neural network f_θ trained to parameterize a simulation, we can combine it with the base model and a time-stepping scheme to produce a

combined operator T_θ to advance the resulting dynamical system through discrete time steps:

$$\tilde{x}_{i,t+L} \triangleq T_\theta^L(x_{i,t}) \approx x_{i,t+L} \tag{5.4}$$

where $x_{i,t}$ is a simulation state from a reference trajectory i and discrete time step t, and the superscript L represents L successive applications of the operator T_θ. Using this we can produce a dataset augmented with neural network perturbations:

$$D_{\text{net}} = \left\{\left(T_\theta^L(x_{b(i),b'(i)-L}),\ y_{b(i),b'(i)}\right)\right\}_{i=1}^{N'} \tag{5.5}$$

where b and b' are mappings which select the base trajectory index and ending time step in this trajectory, respectively. The augmentations here can be computed starting from any trajectory and step such that necessary reference data exists at both the start and end of the length L rollout.

We propose a method of constructing datasets of this type during neural network training, and keeping them updated as the network itself is trained. We find in experiments that using the network's own errors to perturb training data results in improvements in online evaluation stability over the more common technique of adding Gaussian noise. This method provides some of the benefits of online training, even in cases where the target simulation makes actual online training infeasible.

Our approach of using the network's own errors to augment training is similar in spirit to the "pushforward trick" [11]. In that approach, stability is encouraged by taking short, two step online rollouts during training. A stability loss term is added consisting of:

$$\mathcal{L}(T_\theta(x_t + \epsilon),\ x_{t+1}) \tag{5.6}$$

where ϵ is chosen such that

$$x_t + \epsilon = T_\theta(x_{t-1}). \tag{5.7}$$

That is, the first time step is used to compute a perturbation of the state x_t and the second is used to compute the training sample. Gradient updates are propagated only through the second time step.

Our technique is also related to previous experiments with coupled learning [76]. In previous work a pre-trained learned subgrid closure is updated in a separate training phase making reference to a ground truth simulation (running at the "true" resolution as described in Section 3.1). The forcing added by the network is supervised so that it more closely matches the evolution of the true-resolution model, which is also stepped forward. In addition to different treatment of the updates during training, this approach requires time-stepping the ground truth model, which significantly increases the cost in many applications. By contrast, the method proposed here uses only the coarsened, parameterized model making it possible to gather a larger number of samples more efficiently, and without requiring a separate non-augmented pre-training phase.

The approach we consider here permits the use of significantly longer rollouts, and does not require the ability to propagate gradients through the model time steps. This latter advantage makes our approach usable with existing climate models and other simulations which are not automatically differentiable.

5.1 Approach

In implementing this network perturbation and constructing a training set D_{net} there are a few concerns which must be addressed. In particular: (1) the cost of computing a large, diverse range of perturbed samples usable for training, and (2) the "freshness" of these samples. The freshness concern arises from the fact that the behavior of the neural network evolves during training. As a result, errors made by the network early on in learning will not be as relevant toward the end. Therefore very old samples may no longer help improve—and could even harm—stability and generalization. In this section, we detail our approach to implementing this perturbation scheme

and how we attempt to address these concerns.

In order to ensure good coverage of a representative range of inputs, typical neural network training tasks compile relatively large datasets D_{base}. This process is typically completed once, often at significant upfront cost. Attempting to construct a network-perturbed set D_{net} of a similar size during training is likely to be prohibitively expensive, even with a relatively short rollout length L. To reduce this cost we do not fill the set D_{net} at once, but instead add samples to it gradually, computing a few rollouts after each training epoch and drawing training samples uniformly from the union of the base dataset and perturbed dataset: $D = D_{\text{base}} \cup D_{\text{net}}$.

Even so, if D_{base} is very large, or if the computational cost of adding a sample to D_{net} is high, it may still be impractical to achieve a proportion of network-perturbed samples in each training batch sufficient to have a meaningful impact on training. We reduce this cost by treating both D_{base} and D_{net} as *multisets* and adding samples to D_{net} with a multiplicity such that a batch sampled uniformly from its multiset union with D_{base} features a significant number of elements drawn from D_{net}.

Adding samples to D_{net} gradually over the course of training already ensures the presence of at least some samples drawn after more recent training epochs. However, we further enhance this ratio by periodically removing old samples from D_{net}. To simplify the implementation and avoid the need to track the age of each sample independently, we implement this by splitting D_{net} into two multiset pools $D_{\text{net}} = D_{\text{net1}} \cup D_{\text{net2}}$. We begin by first inserting samples into one of these subsets and periodically we switch the active subset receiving new samples. When we switch to a new active set we first clear it, deleting all samples it contains. In this way we periodically clear the subset containing the oldest samples and gradually replace them with fresher data. This also produces, effectively, an annealing schedule, altering the proportion of clean and perturbed samples in each batch as training progresses.

In order to evaluate the impact of the network perturbations, we compare the evaluation performance of networks trained using our new approach against that of networks trained with

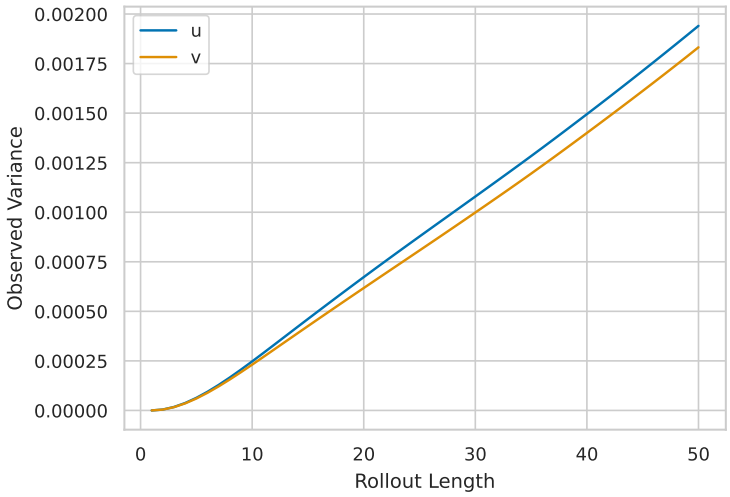


Figure 5.1: Observed error variances for each input channel as a function of rollout length. Values were computed using a set of calibration networks. Variances are computed separately for u and v, the two directional components of the velocity state.

Gaussian noise and with no perturbations. We inject Gaussian noise as in Equation 5.2, but we choose the variance σ^2 such that it matches the empirical variance of errors produced by fully-trained networks after a rollout of length L. In this way we attempt to compare the impact of the structure of the noise and not its scale. With this calibration, we can view the rollout length L as determining a noise level. Longer rollouts are expected to produce larger errors both for network and Gaussian perturbations.

5.2 Experiments

We evaluate performance on the Kolmogorov flow dataset used in Chapter 4 and described in Section 3.2.2, using the small network architecture described in that section and detailed as "psm4pmd8" in Section B.2 without the internal multiscale prediction. We follow the same training procedure described in Section B.2, namely each network is trained for 150 epochs, each consisting of 374 training steps of 32 samples each. We use the Adam optimizer with $\epsilon = 0.001$ and follow a cosine annealing schedule for the learning rate of 7.5×10^{-4}.

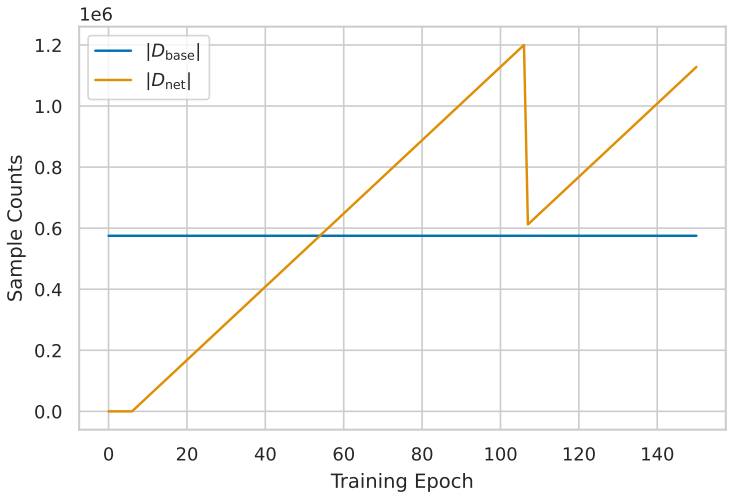


Figure 5.2: Sample counts in each portion of the training set, both base and network-perturbed as a function of the training epoch. At the peak of the network sample counts each training batch is expected to be two-thirds perturbed samples.

Using this procedure we train a set of 10 networks, with Gaussian noise of variance 0.035 injected into 75% of training samples. This corresponds to the best-performing non-multiscale networks selected in Section 4.4.2. From our validation set we select 75 time steps at random, and compute rollouts from these steps using each network for 50 steps. Because each observed simulation step is computed using 15 substeps at a smaller time step for purposes of numerical integration, computing this trajectory involves 750 neural network evaluations. We gather all perturbed states at each of these rollout time steps and compute empirical variances of their errors against the reference trajectory. We use these calibrated variances as the corresponding noise scale for a given network-perturbation rollout length. The results of these measurements are plotted in Figure 5.1.

During training with network perturbations we add 375 perturbed steps after each training epoch to D_{net} with a multiplicity of 32. We begin adding perturbed samples after 6 warmup epochs and every 50 epochs thereafter we switch underlying noise sample pools, emptying the pool with the oldest samples. The number of samples in D_{base} (575,064 samples) and the changing size of D_{net} are plotted in Figure 5.2. This schedule leads to a sawtooth annealing schedule where at its

peak two-thirds of samples in each training batch are expected to be perturbed. We follow this same sample count schedule when training using Gaussian noise perturbation. That is, in place of D_{net} we use a Gaussian-perturbed set D_{noise} where each sample drawn from D_{noise} is perturbed with an independently-sampled Gaussian mask. Finally, we also train a family of networks with no perturbations added. These networks follow the same training procedure, except samples for each training batch are drawn only from D_{base}.

Following these training procedures we train networks using both network and Gaussian noise perturbations for rollout lengths and noise levels corresponding to 1, 5, 10, 25, and 50 steps. All networks are provided the two directional components of velocity u, v and produce parameterization corrections to be applied at each time step. For each noise level we train 5 networks from independent initialization. Once each network is trained it is evaluated on a held-out set of 19 trajectories of 1141 steps each, corresponding to a range of 7.5 simulated seconds. The results of these experiments are reported below.

5.3 Results

For each network we measure its performance on the set of evaluation trajectories by examining the distribution of vorticity decorrelation times, and the error in its energy spectrum vs. the spectrum of the reference trajectory.

The distribution of vorticity decorrelation times is plotted in Figure 5.3. At each evaluation trajectory time step computed online with each trained network, we compute the correlation coefficient of the vorticity field with that of the reference trajectory. The decorrelation time is the number of simulated seconds after which this correlation drops below 0.5. A longer decorrelation time indicates a parameterization which has succeeded to correct the dynamics to more closely track those of the original ground truth simulation through a longer length of time. The set of results for networks trained with no perturbations added is repeated in both the network

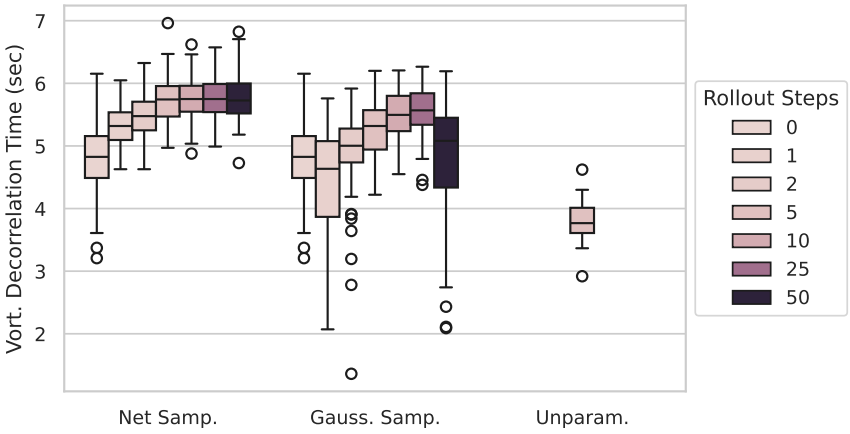


Figure 5.3: Distribution of vorticity decorrelation times for networks trained with different noise types and levels, as well as results for an unparameterized simulation. Higher values are better.

and Gaussian perturbation groups as an entry with a rollout length of 0. The unparameterized simulation is an evaluation trajectory computed with no neural network parameterization of any kind (or equivalently a parameterization which is zero at each step).

From the results in Figure 5.3 we observe that—regardless of perturbation type and noise level—all parameterizations increase the decorrelation time. We also observe that the best-performing noise levels for the network perturbations slightly outperform the best performance of the Gaussian noise networks. The network perturbation's maximum benefit appears to appear to have fully-accrued at a relatively short rollout length of 5 steps, and even the longest rollout lengths do not appear to harm performance in these measurements. For the Gaussian noise training, however, achieving the peak benefit requires a relatively high noise level with values past that point harming performance.

We also measure the error in the energy spectrum of each evaluation trajectory. For each trajectory we average energy spectra across the first 1000 time steps. We then compute the mean error between these and the analogous spectra of the reference trajectories and record the distributions of these errors. These measurements reflect the errors both in total energy of the trajectory and its misallocation across spectral scales. Errors in this measurement can be due to

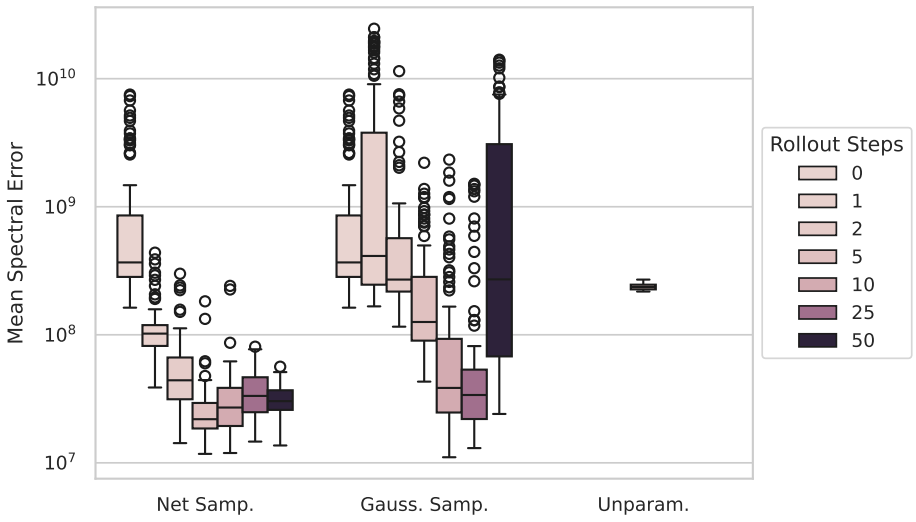


Figure 5.4: Mean spectral error distributions for networks trained with varying noise levels (rollout lengths) and noise types, as well as an unparameterized simulation. Spectra are averaged across a trajectory of 1000 steps and errors are computed against the reference spectra. Lower values are better.

a trajectory having too much or too little energy, or accumulating it too heavily at a particular wavenumber.

The distributions of these errors are plotted in Figure 5.4 divided by perturbation type and noise scale. First we note spectral error of the unparameterized trajectories. These errors are largely due to the uncorrected simulation becoming under-energized as illustrated in Figure 5.5. We also note that parameterization networks trained without noise often harm the spectral error, with large outliers. Some of these trajectories become unstable by the end of the rollout.

For the Gaussian noise training, we require a relatively high noise level before obtaining improved errors, with the best performance being reached only at a level corresponding to 25 steps and with the highest noise level significantly increasing errors. We also note that the Gaussian-trained networks persistently have some trajectories with large, outlier spectral errors regardless of noise level.

By contrast, the network perturbation training reaches its peak error reduction for relatively short rollouts of 5 steps, and even at the lowest nonzero noise level outliers are much better

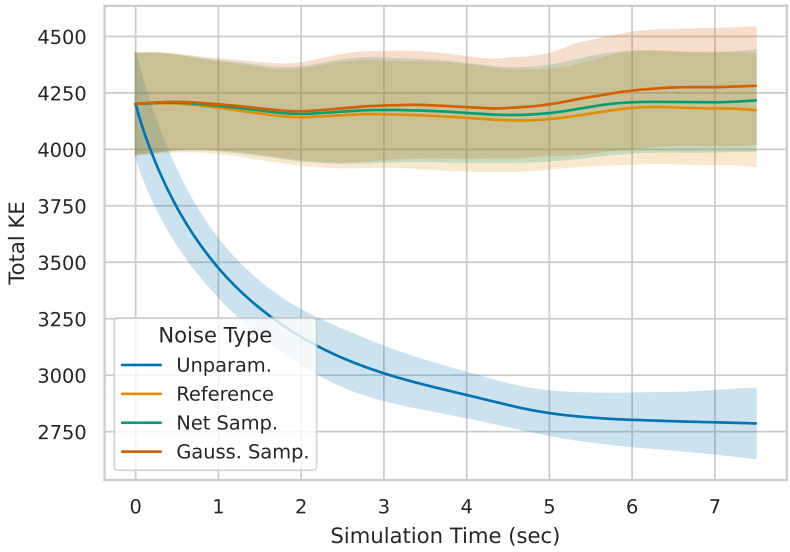


Figure 5.5: Kinetic energy time evolution for each noise type with the noise level chosen as that yielding the best mean spectral error (see Figure 5.4). Gaussian noise uses a level of 25 steps and the network sampling uses 5 steps. Solid lines are means over test trajectories and networks while shaded regions represent 1σ bounds.

controlled, with none observed to significantly exceed the median spectral error of the unparameterized system. This suggests that the network perturbation training does in fact improve network stability and the quality of the parameterization at evaluation time. The best spectral error results observed in our experiments come from network perturbation training.

Figure 5.5 illustrates the evolution of total kinetic energy across evaluation time steps. The values for network and Gaussian perturbations are chosen at the optimal noise level based on results shown in Figure 5.3 and Figure 5.4. From these results we see that networks trained with either type of perturbation are relatively stable, but the network perturbation results track the reference energy level more closely. The Gaussian noise-trained networks more significantly add excess energy to the evaluation trajectories.

Finally, we examine the costs of these two perturbation approaches. In the Gaussian noise injection method, the necessary noise masks are inexpensive to apply, and the computational costs are invariant with respect to the noise level. However, this is not the case for the network noise

Noise Type	Rollout Length					
	1	2	5	10	25	50
Gaussian Sampling	61	61	66	60	58	63
Network Sampling	113	131	141	169	241	378
None	60					

Table 5.1: Training costs in minutes for each combination of noise type and rollout length determining the noise level. Note that the "none" noise type is an exception as it does not follow the noise level and adds no noise. These values are illustrated in Figure 5.6.

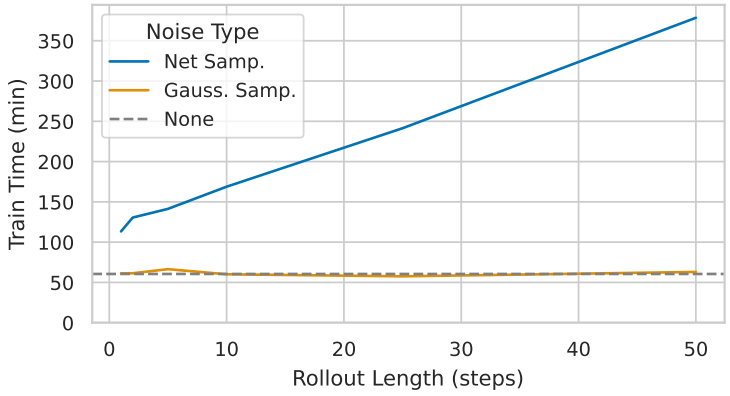


Figure 5.6: Training time costs by noise type and rollout length. This illustrates the dependence of the network sampling type on the rollout length as well as the fixed costs for each noise sampling method. The values plotted are those reported in Table 5.1.

perturbation. In this method there are significant costs involved in performing the rollouts in between each training epoch, and these rise with length of the rollout. To examine these impacts, we record the median time for the training runs of each type, for each possible noise level. These measurements were collected for training runs using NVIDIA RTX 8000 GPUs and are recorded in Table 5.1 and plotted in Figure 5.6. Because our approach affects only the training process, it has no impact on the cost of evaluating the network and rolling out test trajectories.

From these measurements we observe that there are significant fixed costs to enabling the network noise perturbation even for a rollout of a single step. One significant component of these costs is the added overhead of loading samples from two different datasets. In particular, there are significant optimizations which could be applied to the simpler data loader used to sample from D_{net} which is not as well optimized in our current implementation as the data loader used to sample from D_{base}. We believe that optimizations here could significantly reduce these fixed costs. Even so, there also remain significant variable costs which depend on the length of the rollout. However, given our experimental results above it appears that the benefit of network perturbation is already achieved even with very short rollouts with a lower impact on training time.

5.4 Conclusion

In this chapter we have introduced an approach to inject perturbations to training data which closely reflect the real errors made by a neural network parameterization as well as techniques to reduce the cost of this data generation. Our results suggest that this method produces neural network parameterizations that are more stable, and more accurate than those trained using Gaussian noise. Our method is flexible and has no more requirements for the target simulation, than is required to evaluate a neural network with it. In particular, we do not require the ability to backpropagate gradient information through any simulation time steps. This makes our approach generally applicable to a wide variety of target simulations and tasks.

There remain, however, a many extensions which could be explored to further distinguish which aspects of this method contribute the most to improved performance. In particular, one might consider injecting Gaussian noise with a *dynamic* noise level tuned based on a smaller population of rollouts after each training epoch. This would help to further distinguish how much of the improved results is due to state-dependent structures from the network rollouts and how much is due to the adaptation of the perturbations as network training progresses. Other extensions for suitable tasks could include applying the network perturbations to the output values as well, in a style similar to $D_{\text{correcting}}$ in Equation 5.3. This might produce further stability improvements by training the parameterizations to correct small errors in the trajectories but may require additional tuning for the scaling parameter α.

Chapter 6

Conclusion

In this thesis we have presented several projects exploring different ways that machine learning methods can support simulation tasks. We have evaluated their usefulness both in simplifying the modeling process and in improving model accuracy and performance. These works include a first project in which we presented a large suite of benchmark simulation tasks and evaluated the performance of a set of neural network architectures for their ability to learn these problems. We tested multiple configurations of each target task and reported the effects on the accuracy and stability of the resulting models. This project also included an evaluation of the accuracy and performance tradeoffs involved with neural networks and compared these against the capabilities of traditional numerical integration approaches. To the best of our knowledge, prior to this project this type of evaluation had not been systematically conducted.

Using several insights derived from this first project—such as results indicating potential benefits from tailoring neural network architectures to the target problem—we presented in this thesis two further projects involving novel techniques for improving the stability, accuracy, and efficiency of neural networks for the subgrid parameterization problem which arises in climate models. Our first proposed technique was a neural network architecture that directly takes advantage of the inherent multiscale structure involved in the subgrid closure task, making it possible

to produce smaller, more efficient networks that nevertheless preserve higher accuracy. Second, we also presented a novel adjustment to the training process, performing data augmentation such that a network being trained offline is exposed to the types of errors that will be accumulated during an online rollout. Our approach imposes few requirements on the target simulation, only requiring the ability to carry out online evaluation, with no need to backpropagate gradients through time steps.

Applications of machine learning to PDE problems remains an active area of research, and there will undoubtedly be further innovations in this area which will contribute to real world modeling improvements across a wide range of disciplines. Continuing in the same vein as our benchmark project, there is ongoing work to expand the capabilities of learned simulations and to improve the training process. Among these are efforts to develop large training sets useful for general pretraining, with the goal of using these to yield foundation models which can, thanks to a facility with a variety of simulation types, be more cheaply fine-tuned to adapt them to particular problems [62, 95]. The overall success of these methods remains somewhat unclear, but efforts in this direction have already contributed larger and more diverse training sets which also streamline the work of testing machine learning methods on a range of target systems.

Applications to climate models are also progressing. In addition to general improvements in learned subgrid parameterizations such as those we have explored in previous chapters, efforts to deploy these parameterizations to production climate models are ongoing [111]. Efforts to develop new full Earth system models using modern numerical environments may also streamline these deployments and enable computational improvements by allowing both the learned and non-learned components to be developed using the same numerical software toolkit [103]. Deep learning has also found applications to improving weather forecasting, using a neural network system to produce forecasts at a lower computational cost over medium-term time horizons [47, 48]. Further developments here may improve the accuracy of these models over longer time spans and support their use as core parts of real world forecasting. Efforts are also ongoing to

explore similar emulation methods for climate modeling applications, using neural networks to predict climatological quantities over long time spans [19, 94]. Improvements to these methods may come from a variety of techniques to stabilize these networks (such as those explored in Chapter 5) or by taking advantage of more recent deep learning methods such as diffusion modeling [67]. The project discussed in Chapter 4 takes some inspiration from diffusion modeling. Better understanding of these methods will support more extensive production deployments, and techniques such as those explored in this thesis can help improve the efficiency of these architectures and provide possible methods to improve learned model stability.

It is our hope that the results presented in this thesis will be useful for other researchers working in this area. The results from our benchmark experiments in the first project presented here provide general guidance for applications of machine learning to simulation tasks. In the second and third projects discussed in this thesis, our techniques for improving neural network parameterizations have applications to closure models in other settings.

Appendices

A Extended Results for Chapter 2

To illustrate the error distribution for each neural network over the evaluation sampling distribution, we plot the errors as a box plot. Figures A.1, A.2, A.3, A.4, and A.5 show these error distributions, one plot for each system configuration.

Each plot is divided into two panes: one for derivative, and the other for step prediction. The datasets and training protocols followed are identical between the two task formulations. In each, the boxes are grouped first according to learning method, labeled at the bottom on the x-axis. For derivative prediction, the boxes are assembled into sub-groups according to the integrator applied (forward Euler/FE, leapfrog/LF, RK4, backward Euler/BE, or BDF2). These integrators are also indicated by the color of the box. In each group, from left to right the boxes become darker; this indicates the increasing training set size (see Table 2.1). The final box is hatched; this shows the evaluation results on the out-of-distribution set for the network exposed to the largest training set.

The boxes illustrate the distribution over per-trajectory average errors. For each system configuration (a system, derivative/step prediction, learning method, integrator, and particular training set size) we compute the per-step MSE against a ground truth result; these per-step errors are averaged to produce an error estimate for the trajectory. We also train three independent instantiations of each neural network architecture and evaluate each of these on all trajectories

independently. These three repetitions of each trajectory for each network are included as part of the distribution in the box plot. The KNNs and numerical integrators are run a single time each. The errors of these different sampled trajectories form the distribution summarized by the box plot. The variance in the results is produced by a combination of the training results for the three copies of each network, and by the varying performance on each of the sampled evaluation trajectories. These plots were generated using Matplotlib's [36] box plot routines. The box itself ends at the first and third quartiles of the data and the line in the middle is placed at the median of the data. The whiskers extend past the box by 1.5 times the size of the box. Circles are plotted for outlier points which lie outside the range of the whiskers. The plots here have a logarithmic y-axis to accommodate the wide range of error values, thus the boxes do not appear symmetric.

A.1 Weighted errors

In most cases, due to accumulated errors, per-step errors increase as numerical integration proceeds away from the initial condition. To compensate for this trend and in an effort to explore the impact of early vs. late step errors, we include several plots of error distributions for which each time step's MSE has been weighted. To produce these weights, each step's MSE is scaled by a value $1/\exp(\ln(10^2)\cdot p_t)$ where $p_t \in [0,1]$ is a scalar representing the proportional time of the step (zero at start of the trajectory, and one at the end). This produces an exponential decay from the initial steps to the end and reduces the contribution of the final steps by two orders of magnitude. These scaled MSEs are then averaged for each trajectory and each neural network retraining as in the plots above.

The results of these distributions for the Navier-Stokes system—both single- and multi-obstacle forms—are included in Figure A.6 and Figure A.7 below. A change in the relative behavior of the learned methods is most visible in the step prediction results in Figure A.7. Without the weighting, many of the learning methods perform comparably to the KNN; however when emphasizing early steps, these methods demonstrate improved errors relative to the re-weighted KNN errors. This

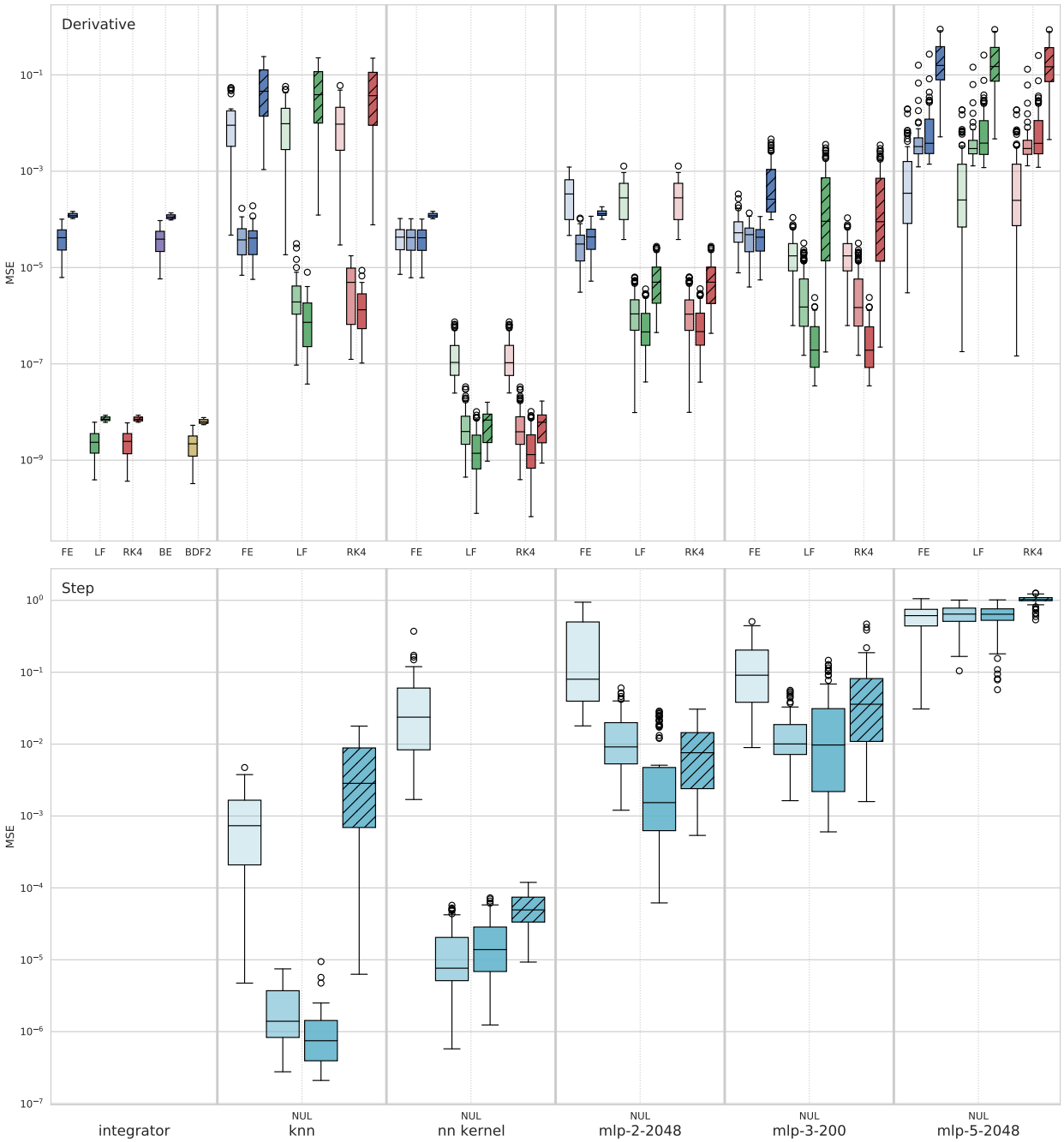


Figure A.1: Error distribution for spring system for multiple training set sizes as well as out-of-distribution results.

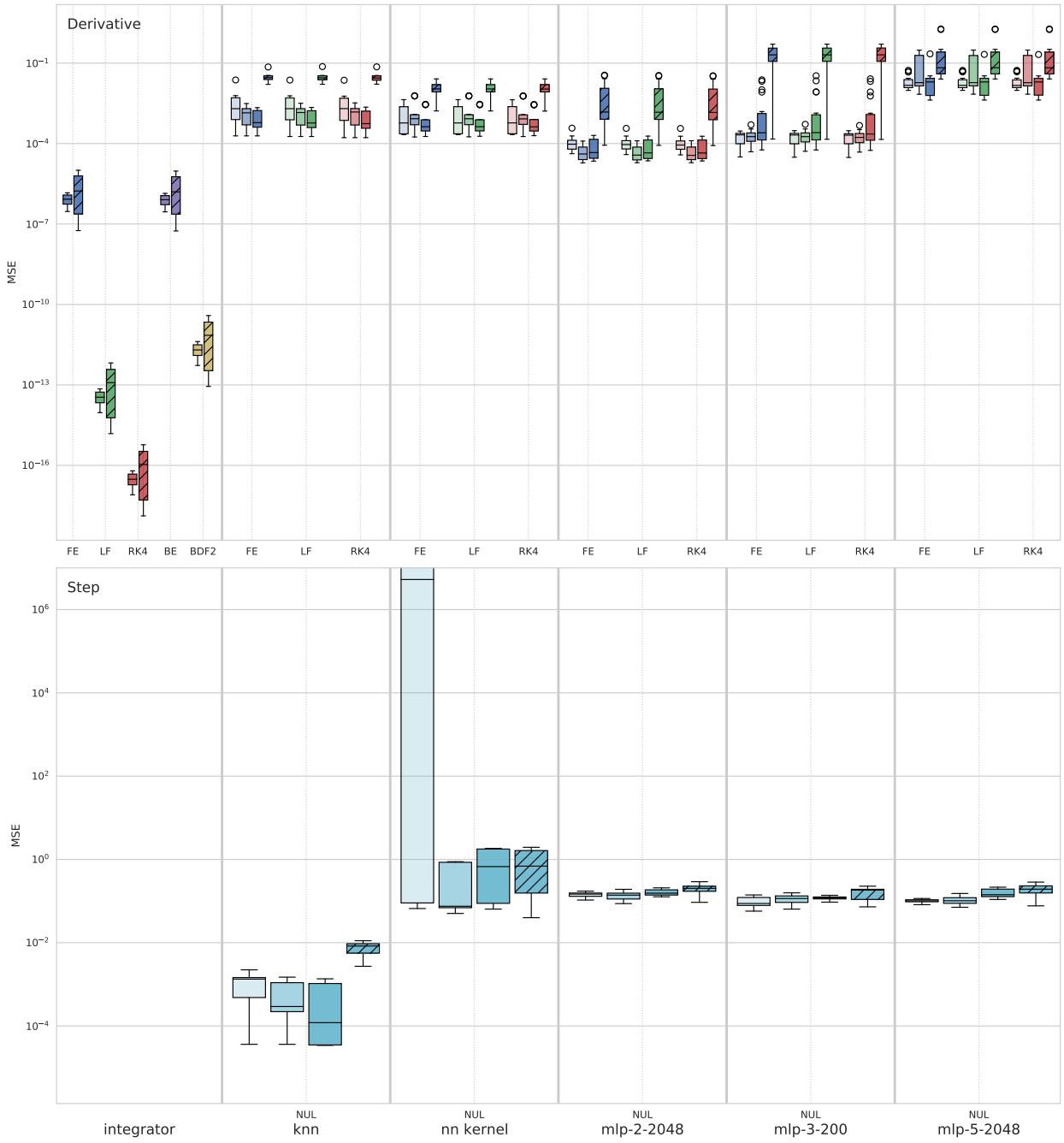


Figure A.2: Error distribution for wave system for multiple training set sizes as well as out-of-distribution results.

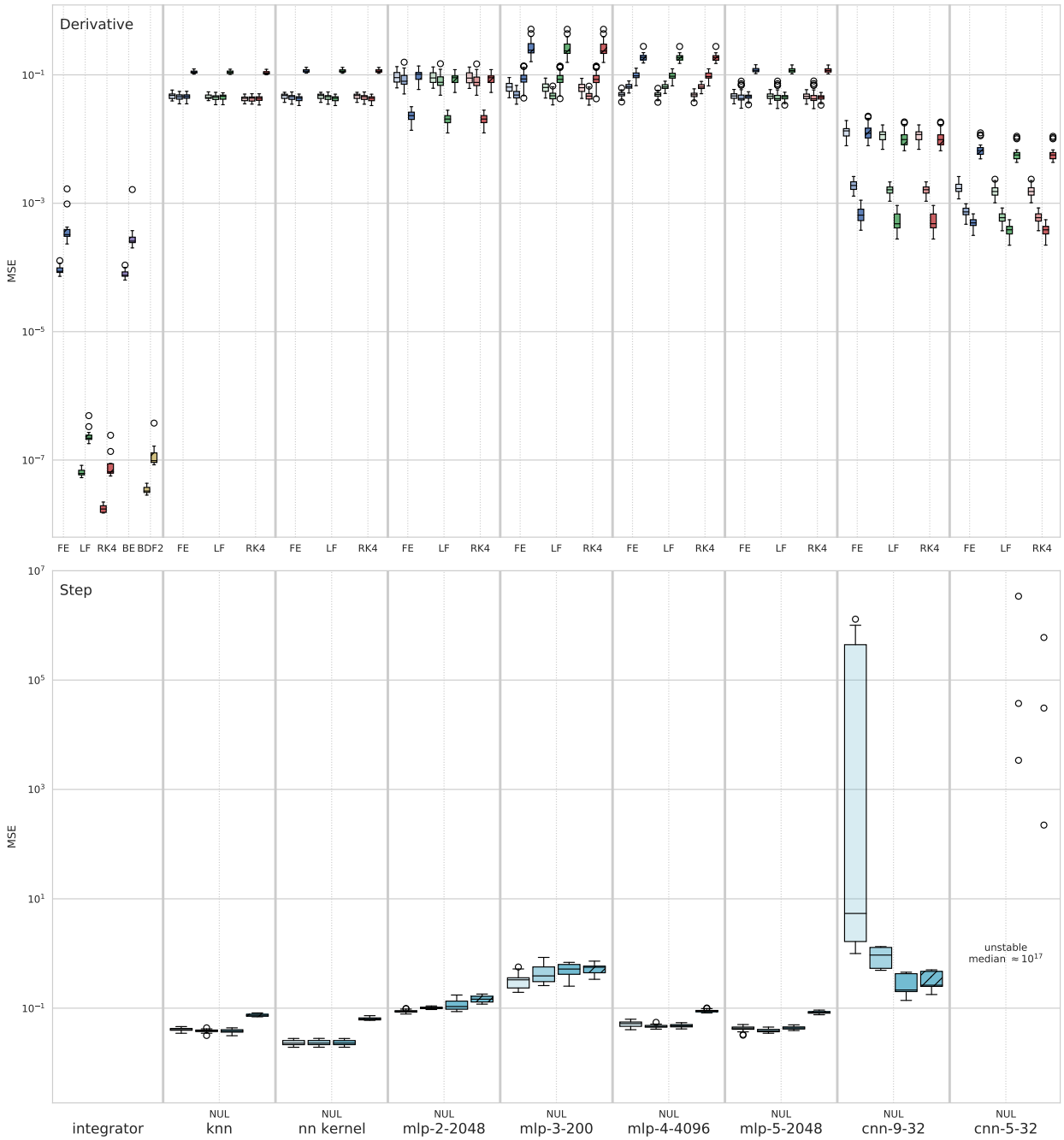


Figure A.3: Error distribution for 10×10 spring mesh system for multiple training set sizes as well as out-of-distribution results.

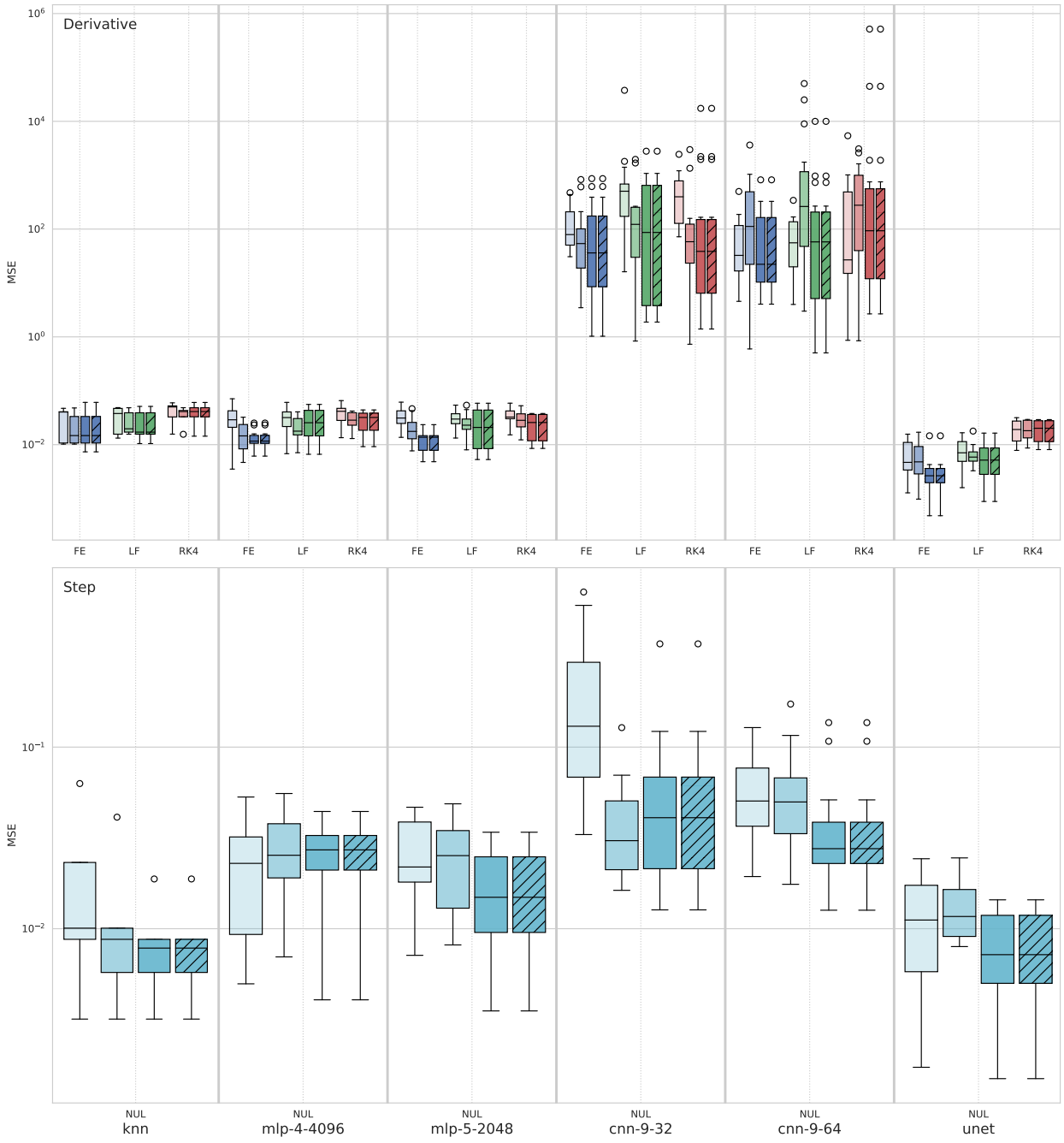


Figure A.4: Error distribution for Navier-Stokes system for multiple training set sizes, and out-of-distribution results. Each trajectory has a single randomly-positioned obstacle. Note that this system does not have results for plain numerical integration.

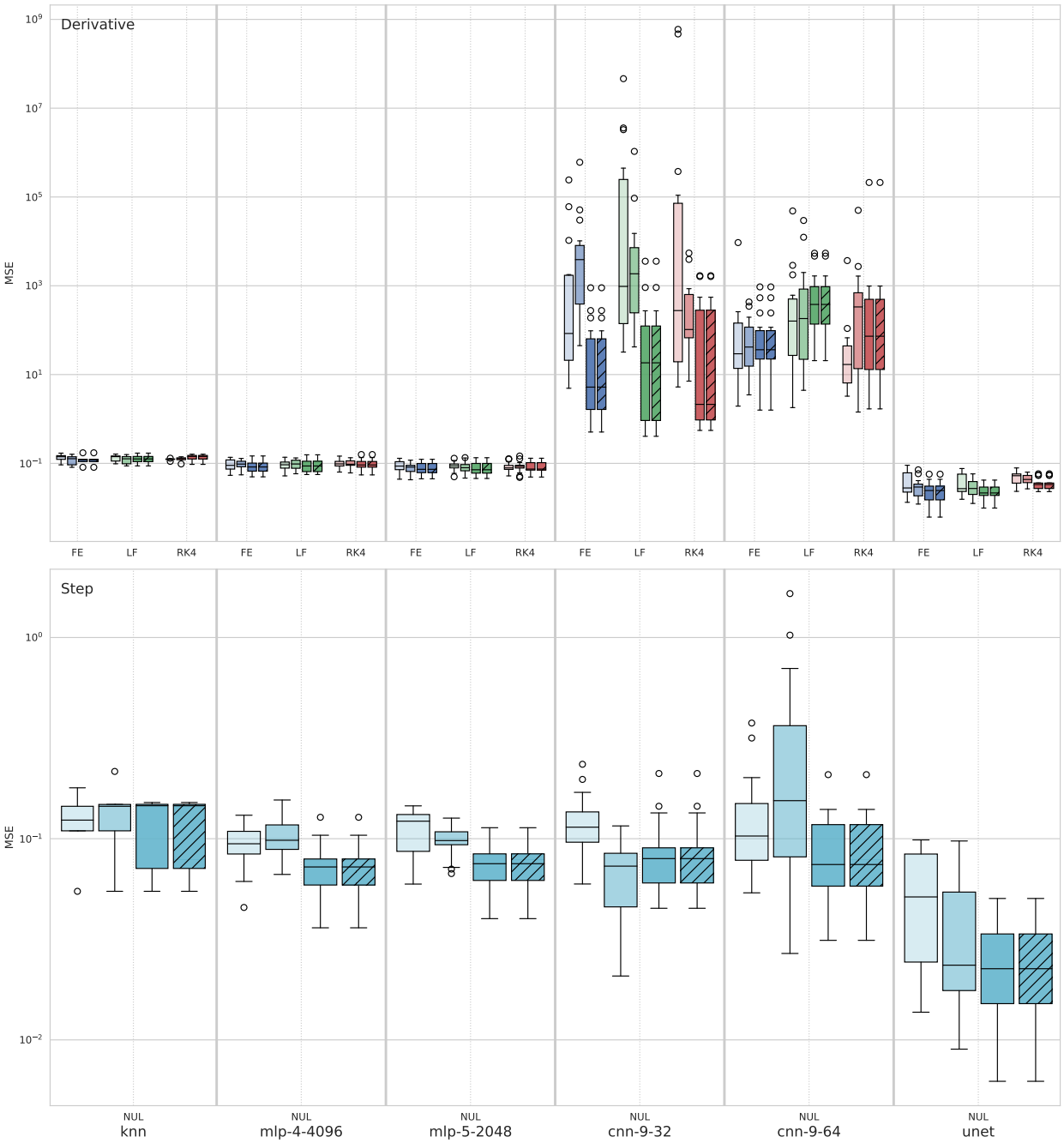


Figure A.5: Error distribution for Navier-Stokes system for multiple training set sizes, and out-of-distribution results. Each trajectory has four randomly-positioned obstacles.

indicates that the learned methods outperform the accuracy of the KNN on the early steps, but are somewhat unstable as the simulation progresses.

For other systems, we did not observe significant changes in relative performance of the learned methods. MSE distributions shifted, but roughly in proportion to each other. This represents a greater general stability in the learned methods on other systems, likely reflecting the more predictable long-term behavior of the other systems. The spring system is periodic, the wave system is stable over time, and the spring mesh system has an energy decay term which simplifies and stabilizes its long-term evolution. As a result, in most cases, successfully learning the target task permits the learned methods to maintain some stability over time, which decreases the relative effect of the per-step weighting.

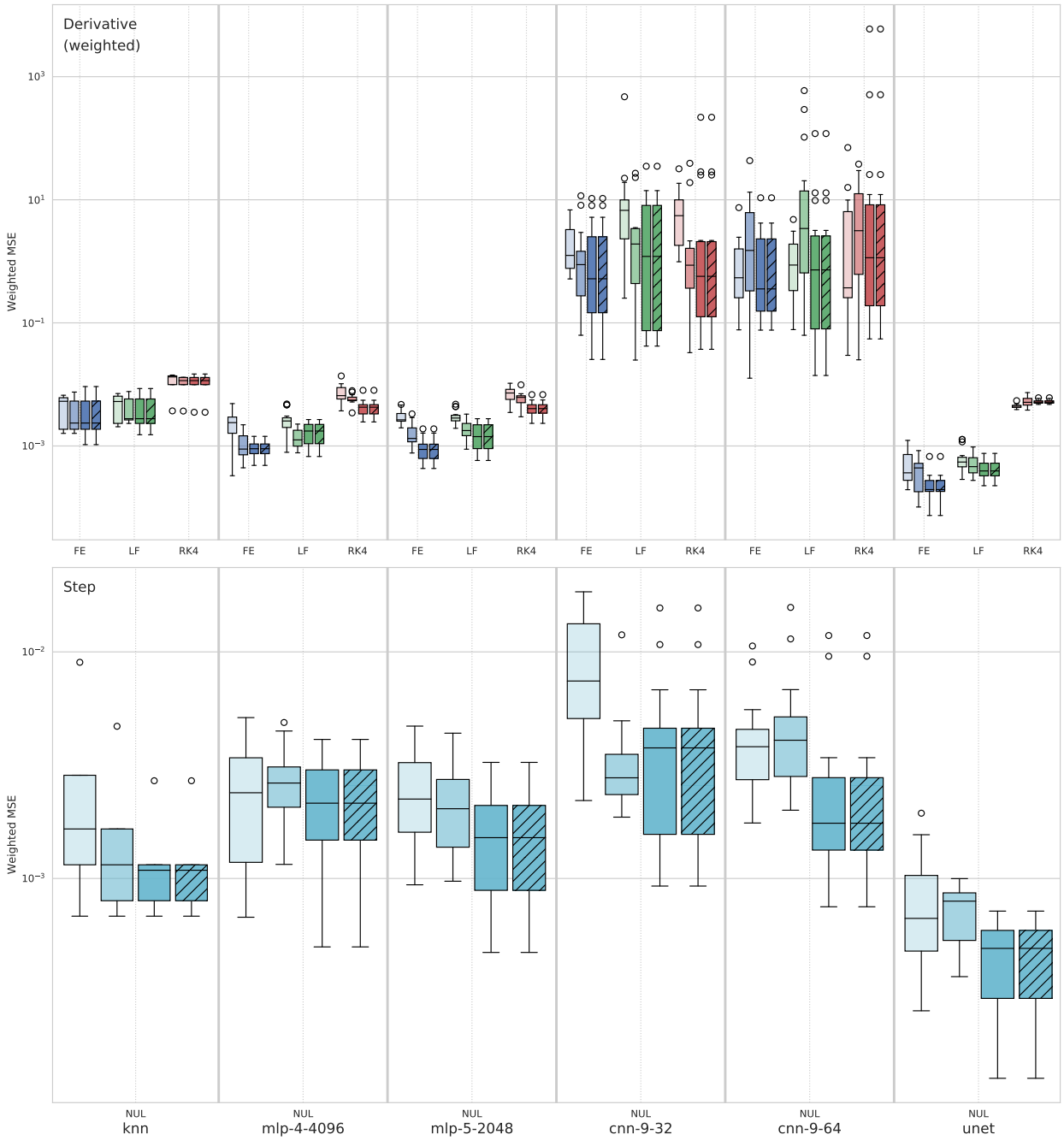


Figure A.6: Error distribution for Navier-Stokes system for multiple training set sizes, and out-of-distribution results. Each trajectory has a single randomly-positioned obstacle. Per-step errors are weighted to decrease the contribution of later time steps with higher errors.

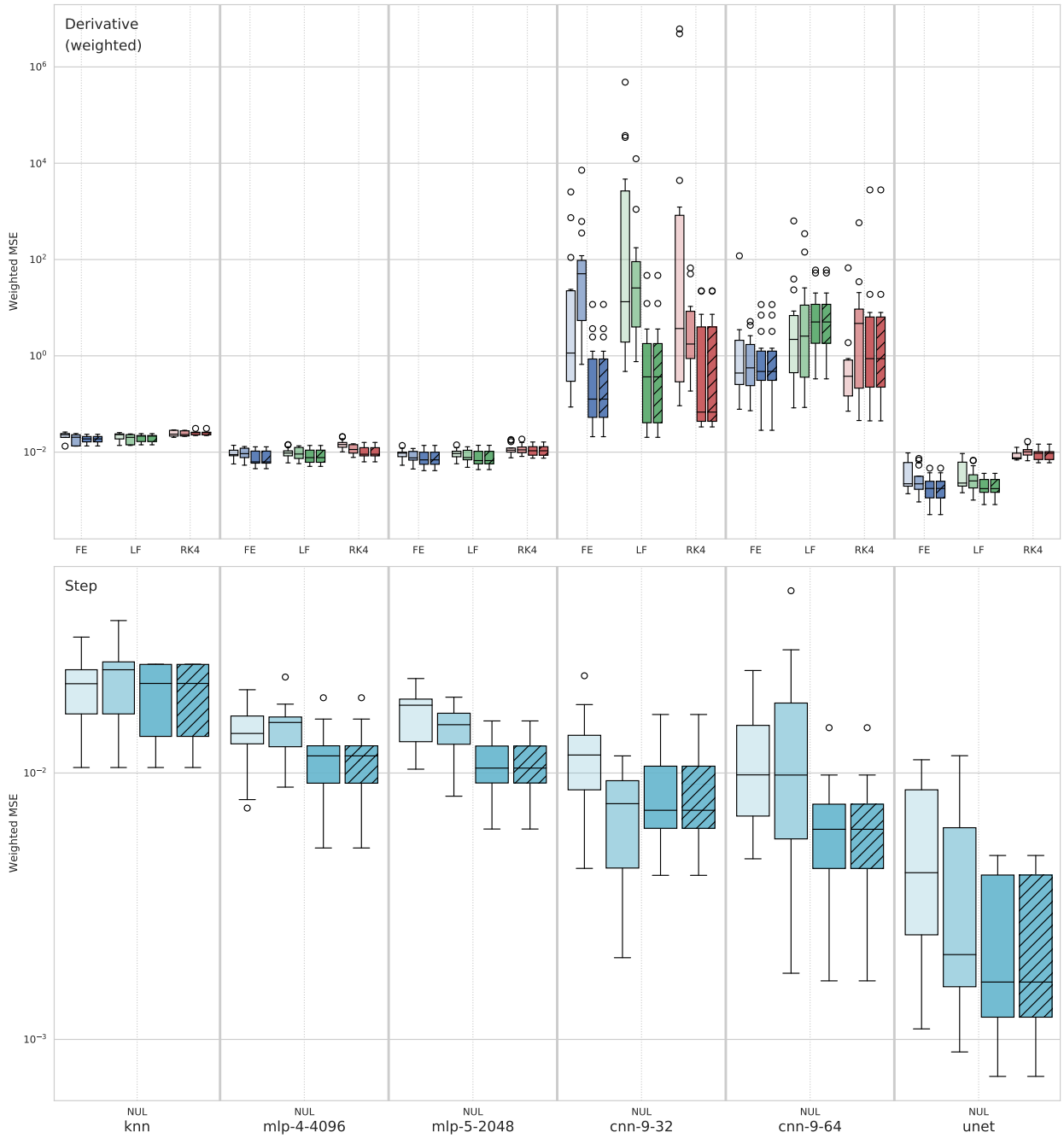


Figure A.7: Error distribution for Navier-Stokes system for multiple training set sizes, and out-of-distribution results. Each trajectory has four randomly-positioned obstacles. Per-step errors are weighted to decrease the contribution of later time steps with higher errors.

B Closure Modeling Network Architecture and Training

The network architectures used in this paper are all feedforward convolutional neural networks. The structure and training parameters of each network vary with the target system and were adjusted for the online combined experiments using the results of the offline separated experiments.

B.1 Separated Experiment Architectures

For our *separated* experiments on the QG system we use two different feedforward CNN architectures from previous work without batch norm [29]. We take the architectural parameters from this work as our default "small" architecture, while the "large" architecture for these experiments roughly doubles the size of each convolution kernel. This produces the architectures listed in Table B.1. We use ReLU activations between each convolution. Each convolution is performed with periodic padding, matching the boundary conditions of the system. All convolutions are with bias. The input and output channel counts are determined by the inputs of the network. For the QG system each input has two layers, each of which is handled as a separate channel. Quantities for the KF system have only a single layer each. These parameters are adjusted for each task to accommodate the inputs and make the required predictions. We implement our networks with Equinox [40].

We train each network with the Adam optimizer [42] as implemented in Optax [4]. The learning rate is set to a constant depending on architecture size: the small networks use 5×10^{-4}, while the large networks use 2×10^{-4}. The networks are trained to minimize MSE loss. Large chunks of 10 850 steps are sampled with replacement from the dataset which is pre-shuffled uniformly. Then each of these chunks is shuffled again and divided into batches of size 256 without replacement. One epoch consists of 333 such batches. We train the small networks for 132 epochs, and the large networks for 96 epochs. We store the network weights which produced the lowest training set loss and use these for evaluation.

Conv. Layer	Chans. Out	Small Kernel Size	Large Kernel Size
1	128	$(5, 5)$	$(9, 9)$
2	64	$(5, 5)$	$(9, 9)$
3	32	$(3, 3)$	$(5, 5)$
4	32	$(3, 3)$	$(5, 5)$
5	32	$(3, 3)$	$(5, 5)$
6	32	$(3, 3)$	$(5, 5)$
7	32	$(3, 3)$	$(5, 5)$
8	out layers	$(3, 3)$	$(5, 5)$

Table B.1: Architecture specifications for each neural network used in the separated experiments. Convolution kernel sizes vary between the architecture sizes. The channel counts are adjusted to accommodate the inputs and outputs of each task.

For all input and target data, we compute empirical means and standard deviations and standardize the overall distributions by these values before passing them to the network. The means and standard deviations from the training set are used in evaluation as well.

B.2 Combined Experiment Architectures

For our combined experiments, which carry out online tests on both the QG and KF systems, we made some modifications to the tested architectures in order to explore the efficiency improvements which could be realized by our multiscale approach. We kept the same "large" architecture as was used in the separated experiments, but carried out an architecture search to select a "small" architecture.

Because each overall parameterization combines two sub-networks (see Figure 4.3) we describe each network by giving a description of the architectures of each component network. These are described by the sizes of the convolution kernels ("pure small (psm)," "small (sm)," "pure medium (pmd)," and "medium (md)," respectively) and the number of convolution layers, either 4 or 8. The "large" architecture described in Table B.1 would be described as "md8" and the combination of two of these is "md8md8."

Using these options, four networks of each architecture were trained on the QG system and

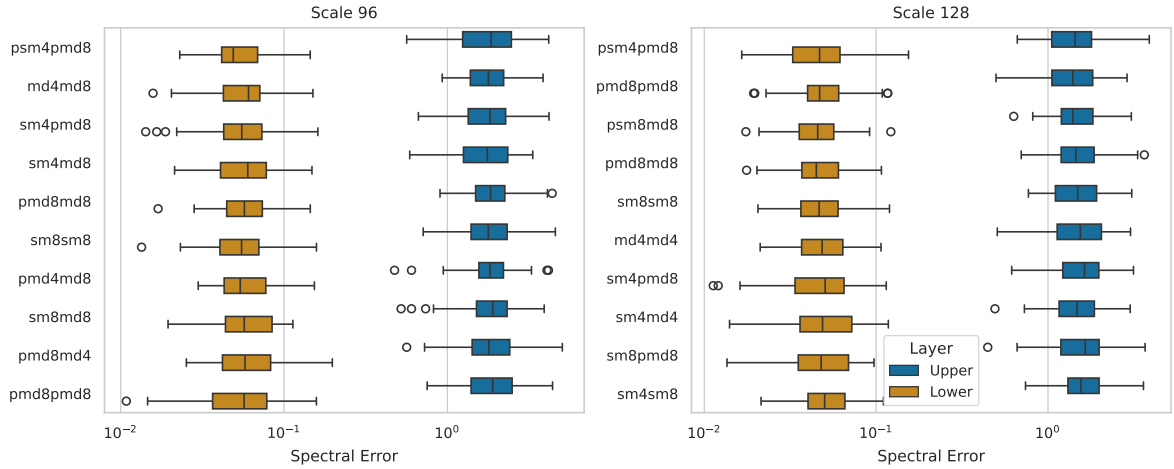


Figure B.1: Architecture selection by scale size on QG system for the combined experiments.

tested online for the resulting spectral errors. The results of each of the two network scales are reported in Figure B.1. These were ranked by increased spectral errors separately for each layer, then each architecture choice was ranked by the worst of these two layer ranks.

Conv. Layer	md8		psm4		pmd8	
	Chans. out	Kernel	Chans. out	Kernel	Chans. out	Kernel
1	128	(9, 9)	128	(3, 3)	128	(5, 5)
2	64	(9, 9)	64	(3, 3)	64	(5, 5)
3	32	(5, 5)	32	(3, 3)	32	(5, 5)
4	32	(5, 5)	out layers	(3, 3)	32	(5, 5)
5	32	(5, 5)			32	(5, 5)
6	32	(5, 5)			32	(5, 5)
7	32	(5, 5)			32	(5, 5)
8	out layers	(5, 5)			out layers	(5, 5)

Table B.2: Architecture specifications for each neural network used in the combined experiments including those selected through the architecture search.

The winning "small" architecture for both QG scales was "psm4pmd8." Details of these parameters are provided in Table B.2.

The training parameters also varied from the separated experiments due to the new end-to-end training configuration. For the QG system training was conducted using the Adam optimizer

following a cosine annealing schedule with one epoch of linear learning rate warmup. The "md" and "pmd" networks were trained for 50 epochs at a learning rate of 0.0004 while the "psm" network was trained for 100 epochs (with 374 batches per epoch) with a learning rate of 0.001. All batches had size 64.

For the KF system, training was carried out using the Adam optimizer with $\epsilon = 0.001$ (modified from the default). These networks followed the same cosine annealing with warmup schedule as the QG systems but had an ending learning rate of 0.0001 and a peak learning rate of 7.5×10^{-4} for 150 epochs with batches of size 32. Each epoch consisted of 374 batches. Each training run selected the network with the best validation loss.

C Extended Online Results for Chapter 4

Extended KF calibration results for large architectures are presented in Figure C.1 and Figure C.2. The large architectures had persistent stability problems on the KF system that could not be resolved with added noise during training.

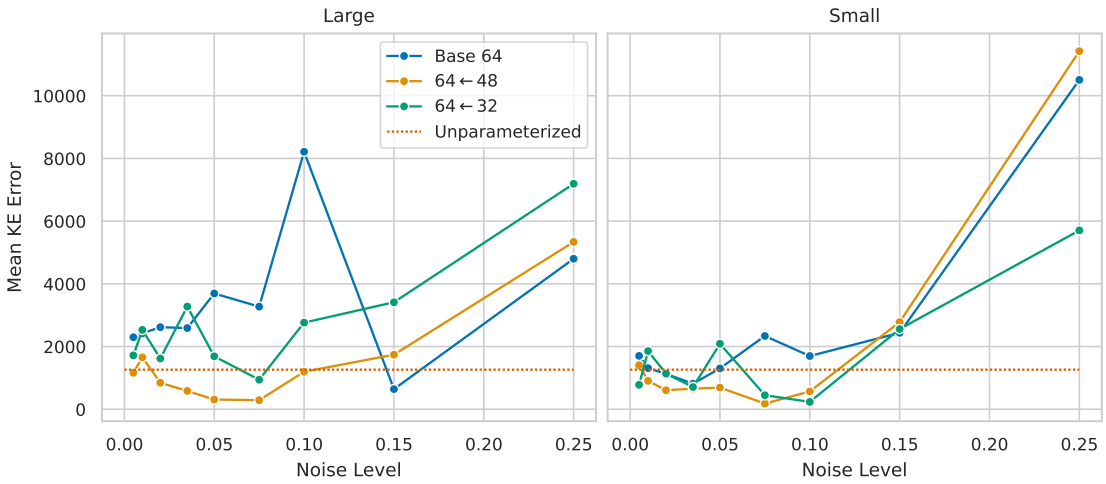


Figure C.1: An extended version of Figure 4.9 including results for the large network architecture. These architectures show significant instability, particularly for the baselines, even with significant added noise.

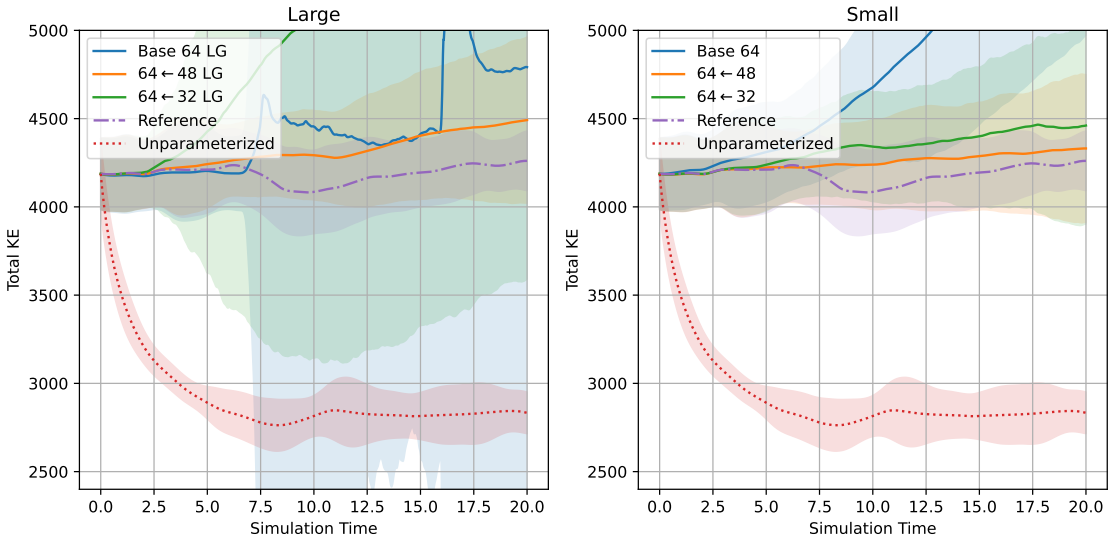


Figure C.2: An extended version of Figure 4.7b including results for the large network architecture which produced generally unstable results even with attempts to improve stability. However even here it appears that the added information from a scale of size 48 may help improve network stability.

Bibliography

[1] Ryan Abernathey et al. *PyQG*. Version 0.7.2. May 2022. DOI: 10.5281/zenodo.6563667.

[2] A. C. Antoulas and B. D. Q. Anderson. "On the Scalar Rational Interpolation Problem". In: *IMA Journal of Mathematical Control & Information* 3.2-3 (1986), pp. 61–88. DOI: 10.1093/imamci/3.2-3.61.

[3] Athanasios C. Antoulas, Christopher Beattie, and Serkan Gugercin. *Interpolatory Methods for Model Reduction*. SIAM, 2020.

[4] Igor Babuschkin et al. *The DeepMind JAX Ecosystem*. 2020. URL: http://github.com/deepmind.

[5] Andrea Beck and Marius Kurz. "A perspective on machine learning methods in turbulence modeling". In: *GAMM-Mitteilungen* 44.1 (2021). DOI: 10.1002/gamm.202100002.

[6] Jules Berman and Benjamin Peherstorfer. "Randomized Sparse Neural Galerkin Schemes for Solving Evolution Equations with Deep Networks". In: *Advances in Neural Information Processing Systems*. Ed. by A. Oh et al. Vol. 36. Curran Associates, Inc., 2023, pp. 4097–4114. DOI: 10.48550/arXiv.2310.04867. URL: https://proceedings.neurips.cc/paper_files/paper/2023/file/0cb310ed8121549488fea8e8c2056096-Paper-Conference.pdf.

[7] Jules Berman, Paul Schwerdtner, and Benjamin Peherstorfer. "Chapter 8 - Neural Galerkin schemes for sequential-in-time solving of partial differential equations with deep net-

works". In: *Numerical Analysis Meets Machine Learning*. Ed. by Siddhartha Mishra and Alex Townsend. Vol. 25. Handbook of Numerical Analysis. Elsevier, 2024, pp. 389–418. DOI: 10.1016/bs.hna.2024.05.006.

[8] Chris M. Bishop. "Training with Noise is Equivalent to Tikhonov Regularization". In: *Neural Computation* 7.1 (Jan. 1995), pp. 108–116. DOI: 10.1162/neco.1995.7.1.108.

[9] Florent Bonnet et al. "AirfRANS: High Fidelity Computational Fluid Dynamics Dataset for Approximating Reynolds-Averaged Navier-Stokes Solutions". In: *Proceedings of the 35th Conference on Neural Information Processing Systems Track on Datasets and Benchmarks*. 2022. DOI: 10.48550/arXiv.2212.07564. URL: https://proceedings.neurips.cc/paper_files/paper/2022/file/94ab7b23a345f93333eac8748a66c763-Paper-Datasets_and_Benchmarks.pdf.

[10] James Bradbury et al. *JAX: composable transformations of Python+NumPy programs*. Version 0.4.11. 2023. URL: https://github.com/google/jax.

[11] Johannes Brandstetter, Daniel Worrall, and Max Welling. "Message Passing Neural PDE Solvers". In: *ICLR* (2022). URL: https://arxiv.org/abs/2202.03376.

[12] Joan Bruna, Benjamin Peherstorfer, and Eric Vanden-Eijnden. "Neural Galerkin schemes with active learning for high-dimensional evolution equations". In: *Journal of Computational Physics* 496 (2024). DOI: 10.1016/j.jcp.2023.112588.

[13] Steven L. Brunton and J. Nathan Kutz. "Promising directions of machine learning for partial differential equations". In: *Nature Computational Science* 4 (2024). DOI: 10.1038/s43588-024-00643-2.

[14] Steven L. Brunton, Joshua L. Proctor, and J. Nathan Kutz. "Discovering governing equations from data by sparse identification of nonlinear dynamical systems". In: *Proceedings of the National Academy of Sciences* 113.15 (2016), pp. 3932–3937. DOI: 10.1073/pnas.1517384113.

[15] Zhengdao Chen et al. “Symplectic Recurrent Neural Networks”. In: *8th International Conference on Learning Representations*. 2020. DOI: 10.48550/arXiv.1909.13334.

[16] Peter V. Coveney, Edward R. Dougherty, and Roger R. Highfield. “Big data need big theory too”. In: *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences* 374.2080 (2016), p. 20160153. DOI: 10.1098/rsta.2016.0153. eprint: https://royalsocietypublishing.org/doi/pdf/10.1098/rsta.2016.0153. URL: https://royalsocietypublishing.org/doi/abs/10.1098/rsta.2016.0153.

[17] Miles Cranmer et al. “Discovering Symbolic Models from Deep Learning with Inductive Biases”. In: *Proceedings of the 34th International Conference on Neural Information Processing Systems*. NIPS ’20. 2020. URL: https://proceedings.neurips.cc/paper/2020/file/c9f2f917078bd2db12f23c3b413d9cba-Paper.pdf.

[18] James P. Crutchfield and Bruce S. Mcnamara. “Equations of motion from a data series”. In: *Complex Systems* (1987), p. 452.

[19] Surya Dheeshjith et al. “Samudra: An AI Global Ocean Emulator for Climate”. In: *arXiv preprint* (2024). DOI: 10.48550/arXiv.2412.03795.

[20] P. A. Durbin and B. A. Pettersson Reif. *Statistical Theory and Modeling for Turbulent Flows*. 2nd ed. Wiley and Sons, Ltd., 2011. DOI: 10.1002/9780470972076.

[21] Jeffrey L. Elman and David Zipser. “Learning the hidden structure of speech”. In: *The Journal of the Acoustical Society of America* 83.4 (Apr. 1988), pp. 1615–1626. DOI: 10.1121/1.395916.

[22] Denis J. Evans and Gary Morriss. *Statistical Mechanics of Nonequilibrium Liquids*. Cambridge University Press, 2008. DOI: 10.1017/CBO9780511535307.

[23] Veronika Eyring et al. “Pushing the frontiers in climate modelling and analysis with machine learning”. In: *Nature Climate Change* (2024). DOI: 10.1038/s41558-024-02095-y.

[24] Baylor Fox-Kemper et al. “Principles and advances in subgrid modelling for eddy-rich simulations”. In: *CLIVAR Exchanges (WGOMD Workshop on High Resolution Ocean Climate Modeling)*. Vol. 19. 2014, pp. 42–46.

[25] Baylor Fox-Kemper et al. “Challenges and Prospects in Ocean Circulation Models”. In: *Frontiers in Marine Science* 6 (2019). ISSN: 2296-7745. DOI: 10.3389/fmars.2019.00065. URL: https://www.frontiersin.org/articles/10.3389/fmars.2019.00065.

[26] Hugo Frezat et al. “A Posteriori Learning for Quasi-Geostrophic Turbulence Parametrization”. In: *Journal of Advances in Modeling Earth Systems* 14.11 (2022). e2022MS003124 2022MS003124, e2022MS003124. DOI: 10.1029/2022MS003124. eprint: https://agupubs.onlinelibrary.wiley.com/doi/pdf/10.1029/2022MS003124. URL: https://agupubs.onlinelibrary.wiley.com/doi/abs/10.1029/2022MS003124.

[27] Hugo Frezat et al. “Gradient-free online learning of subgrid-scale dynamics with neural emulators”. In: *arXiv preprint* (2023). DOI: 10.48550/arXiv.2310.19385. eprint: 2310.19385.

[28] Samuel Greydanus, Misko Dzamba, and Jason Yosinski. “Hamiltonian Neural Networks”. In: *Advances in Neural Information Processing Systems*. Ed. by H. Wallach et al. Vol. 32. Curran Associates, Inc., 2019, pp. 15379–15389. URL: https://proceedings.neurips.cc/paper/2019/file/26cd8ecadce0d4efd6cc8a8725cbd1f8-Paper.pdf.

[29] Arthur P. Guillaumin and Laure Zanna. “Stochastic-Deep Learning Parameterization of Ocean Momentum Forcing”. In: *Journal of Advances in Modeling Earth Systems* 13.9 (2021). e2021MS002534 2021MS002534, e2021MS002534. DOI: 10.1029/2021MS002534. eprint: https://agupubs.onlinelibrary.wiley.com/doi/pdf/10.1029/2021MS002534. URL: https://agupubs.onlinelibrary.wiley.com/doi/abs/10.1029/2021MS002534.

[30] B. Gustavsen and A. Semlyen. "Rational approximation of frequency domain responses by vector fitting". In: *IEEE Transactions on Power Delivery* 14.3 (July 1999), pp. 1052–1061. DOI: 10.1109/61.772353.

[31] Ehsan Haghighat et al. *A deep learning framework for solution and discovery in solid mechanics*. 2020. DOI: 10.48550/arXiv.2003.02751.

[32] Ernst Hairer, Syvert P. Nørsett, and Gerhard Wanner. *Solving Ordinary Differential Equations I: Nonstiff Problems*. Springer, 2009.

[33] Ernst Hairer and Gerhard Wanner. *Solving Ordinary Differential Equations II: Stiff and Differential-Algebraic Problems*. Springer, 2009.

[34] Charles R. Harris et al. "Array programming with NumPy". In: *Nature* 585.7825 (2020), pp. 357–362. URL: https://doi.org/10.1038/s41586-020-2649-2.

[35] Jonathan Ho et al. "Imagen Video: High Definition Video Generation with Diffusion Models". In: *arXiv Preprint* (2022). DOI: 10.48550/arXiv.2210.02303.

[36] 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.

[37] Arieh Iserles. *A First Course in the Numerical Analysis of Differential Equations*. 2nd ed. Cambridge University Press, 2008. DOI: 10.1017/CBO9780511995569.

[38] Malte F. Jansen et al. "Toward an Energetically Consistent, Resolution Aware Parameterization of Ocean Mesoscale Eddies". In: *Journal of Advances in Modeling Earth Systems* 11.8 (2019), pp. 2844–2860. DOI: https://doi.org/10.1029/2019MS001750. URL: https://agupubs.onlinelibrary.wiley.com/doi/abs/10.1029/2019MS001750.

[39] Aditya Khadilkar, Jun Wang, and Rahul Rai. "Deep learning-based stress prediction for bottom-up SLA 3D printing process". In: *The International Journal of Advanced Manufactur-*

ing Technology 102.5 (June 2019), pp. 2555–2569. ISSN: 1433-3015. DOI: 10.1007/s00170-019-03363-4. URL: https://doi.org/10.1007/s00170-019-03363-4.

[40] Patrick Kidger and Cristian Garcia. "Equinox: neural networks in JAX via callable PyTrees and filtered transformations". In: *Differentiable Programming workshop at Neural Information Processing Systems 2021* (2021). DOI: 10.48550/arXiv.2111.00254.

[41] Byungsoo Kim et al. "Deep fluids: A generative network for parameterized fluid simulations". In: *Computer Graphics Forum* 38.2 (2019), pp. 59–70. DOI: 10.1111/cgf.13619.

[42] Diederik P Kingma and Jimmy Ba. "Adam: A method for stochastic optimization". In: *International Conference on Learning Representations*. 2015. DOI: 10.48550/arXiv.1412.6980.

[43] Thomas Kipf et al. "Neural Relational Inference for Interacting Systems". In: *International Conference on Machine Learning*. PMLR. 2018, pp. 2688–2697. DOI: 10.48550/arXiv.1802.04687.

[44] Georgios Kissas et al. "Machine learning in cardiovascular flows modeling: Predicting arterial blood pressure from non-invasive 4D flow MRI data using physics-informed neural networks". In: *Computer Methods in Applied Mechanics and Engineering* 358 (2020), p. 112623. DOI: 10.1016/j.cma.2019.112623.

[45] Dmitrii Kochkov et al. "Machine learning–accelerated computational fluid dynamics". In: *Proceedings of the National Academy of Sciences* 118.21 (2021). ISSN: 0027-8424. DOI: 10.1073/pnas.2101784118. eprint: https://www.pnas.org/content/118/21/e2101784118.full.pdf. URL: https://www.pnas.org/content/118/21/e2101784118.

[46] Boris Kramer, Benjamin Peherstorfer, and Karen E. Willcox. "Learning Nonlinear Reduced Models from Data with Operator Inference". In: *Annual Review of Fluid Mechanics* 56 (2024). DOI: 10.1146/annurev-fluid-121021-025220.

[47] Thorsten Kurth et al. “FourCastNet: Accelerating Global High-Resolution Weather Forecasting Using Adaptive Fourier Neural Operators”. In: *Proceedings of the Platform for Advanced Scientific Computing Conference*. New York, NY, USA: Association for Computing Machinery, 2023. DOI: 10.1145/3592979.3593412.

[48] Remi Lam et al. “GraphCast: Learning skillful medium-range global weather forecasting”. In: *Science* 382.6677 (2023). DOI: 10.1126/science.adi2336.

[49] Siu Kwan Lam, Antoine Pitrou, and Stanley Seibert. “Numba: A LLVM-Based Python JIT Compiler”. In: *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*. LLVM ’15. Austin, Texas: Association for Computing Machinery, 2015. ISBN: 9781450340052. DOI: 10.1145/2833157.2833162. URL: https://doi.org/10.1145/2833157.2833162.

[50] Yu Li et al. “Reconstruction of Simulation-Based Physical Field by Reconstruction Neural Network Method”. In: *arXiv Preprint* (2018). DOI: 10.48550/arXiv.1805.00528.

[51] Yu Li et al. “Image-based reconstruction for the impact problems by using DPNNs”. In: *arXiv Preprint* (2019). DOI: 10.48550/arXiv.1905.03229.

[52] Zongyi Li et al. “Fourier Neural Operator for Parametric Partial Differential Equations”. In: *International Conference on Learning Representations*. 2021. DOI: 10.48550/arXiv.2010.08895.

[53] Liang Liang, Wenbin Mao, and Wei Sun. “A feasibility study of deep learning for predicting hemodynamics of human thoracic aorta”. In: *Journal of Biomechanics* 99 (2020), p. 109544. DOI: 10.1016/j.jbiomech.2019.109544.

[54] Liang Liang et al. “A deep learning approach to estimate stress distribution: a fast and accurate surrogate of finite-element analysis”. In: *Journal of The Royal Society Interface* 15.138 (2018), p. 20170844. DOI: 10.1098/rsif.2017.0844. eprint: https://roy

alsocietypublishing.org/doi/pdf/10.1098/rsif.2017.0844. URL: https://royalsocietypublishing.org/doi/abs/10.1098/rsif.2017.0844.

[55] Jerry Lin et al. "Navigating the Noise: Bringing Clarity to ML Parameterization Design with $\mathcal{O}(100)$ Ensembles". In: *arXiv preprint* (2023). DOI: 10.48550/arXiv.2309.16177.

[56] Lu Lu et al. "DeepXDE: A deep learning library for solving differential equations". In: *SIAM Review* 63.1 (2021), pp. 208–228. DOI: 10.1137/19M1274067.

[57] Gonzalo D Maso Talou et al. "Deep Learning Over Reduced Intrinsic Domains for Efficient Mechanics of the Left Ventricle". In: *Frontiers in Physics* 8 (2020), p. 30. DOI: 10.3389/fphy.2020.00030.

[58] R. Maulik et al. "Subgrid modelling for two-dimensional turbulence using neural networks". In: *Journal of Fluid Mechanics* 858 (2019), pp. 122–144. DOI: 10.1017/jfm.2018.770.

[59] Michael McCabe et al. "Multiple Physics Pretraining for Spatiotemporal Surrogate Models". In: *Advances in Neural Information Processing Systems*. Ed. by A. Globerson et al. Vol. 37. Curran Associates, Inc., 2024. DOI: 10.48550/arXiv.2310.02994. URL: https://proceedings.neurips.cc/paper_files/paper/2024/file/d7cb9db5ade2db7814fbd01ee59f4c7b-Paper-Conference.pdf.

[60] Yuji Nakatsukasa, Olivier Sète, and Lloyd N. Trefethen. "The AAA Algorithm for Rational Approximation". In: *SIAM Journal on Scientific Computing* 40.3 (2018), A1494–A1522. DOI: 10.1137/16M1106122.

[61] Zhenguo Nie, Haoliang Jiang, and Levent Burak Kara. "Stress Field Prediction in Cantilevered Structures Using Convolutional Neural Networks". In: *Journal of Computing and Information Science in Engineering* 20.1 (2020). DOI: 10.1115/1.4044097.

[62] Ruben Ohana et al. "The Well: a Large-Scale Collection of Diverse Physics Simulations for Machine Learning". In: *Proceedings of the 38th Conference on Neural Information Processing*

Systems Track on Datasets and Benchmarks. 2024. DOI: 10.48550/arXiv.2412.00568. URL: https://proceedings.neurips.cc/paper_files/paper/2024/file/4f9a5acd91ac76569f2fe291b1f4772b-Paper-Datasets_and_Benchmarks_Track.pdf.

[63] Karl Otness. *PyQG-JAX*. Version 0.8.1. Feb. 2024. DOI: 10.5281/zenodo.10719906.

[64] Karl Otness et al. "An Extensible Benchmark Suite for Learning to Simulate Physical Systems". In: *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks*. Ed. by J. Vanschoren and S. Yeung. Vol. 1. 2021. URL: https://datasets-benchmarks-proceedings.neurips.cc/paper/2021/file/3def184ad8f4755ff269862ea77393dd-Paper-round1.pdf.

[65] Norman H. Packard et al. "Geometry from a Time Series". In: *Physical Review Letters* 45 (1980), pp. 712–716. DOI: 10.1103/PhysRevLett.45.712.

[66] Adam Paszke et al. "PyTorch: An Imperative Style, High-Performance Deep Learning Library". In: *Advances in Neural Information Processing Systems 32*. Ed. by H. Wallach et al. Curran Associates, Inc., 2019, pp. 8024–8035. URL: https://papers.nips.cc/paper_files/paper/2019/file/bdbca288fee7f92f2bfa9f7012727740-Paper.pdf.

[67] Chris Pedersen, Laure Zanna, and Joan Bruna. "Thermalizer: Stable autoregressive neural emulation of spatiotemporal chaos". In: *arXiv preprint* (2025). DOI: 10.48550/arXiv.2503.18731.

[68] F. Pedregosa et al. "Scikit-learn: Machine Learning in Python". In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830. URL: https://dl.acm.org/doi/10.5555/1953048.2078195.

[69] Liqian Peng and Kamran Mohseni. "Symplectic model reduction of Hamiltonian systems". In: *SIAM Journal on Scientific Computing* 38.1 (2016), A1–A27. URL: https://doi.org/10.1137/140978922.

[70] Pavel Perezhogin, Laure Zanna, and Carlos Fernandez-Granda. “Generative Data-Driven Approaches for Stochastic Subgrid Parameterizations in an Idealized Ocean Model”. In: *Journal of Advances in Modeling Earth Systems* 15.10 (2023). DOI: 10.1029/2023MS003681.

[71] Tobias Pfaff et al. “Learning Mesh-Based Simulation with Graph Networks”. In: *ICLR*. 2021. DOI: 10.48550/arXiv.2010.03409.

[72] E. Qian et al. “Lift & Learn: Physics-informed machine learning for large-scale nonlinear dynamical systems”. In: *Physica D: Nonlinear Phenomena* 406 (2020). DOI: 10.1016/j.physd.2020.132401.

[73] Ali Rahimi and Benjamin Recht. “Random Features for Large-Scale Kernel Machines”. In: *Advances in Neural Information Processing Systems*. Ed. by J. Platt et al. Vol. 20. Curran Associates, Inc., 2008. URL: https://proceedings.neurips.cc/paper/2007/file/013a006f03dbc5392effeb8f18fda755-Paper.pdf.

[74] Maziar Raissi, Paris Perdikaris, and George E Karniadakis. “Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations”. In: *Journal of Computational Physics* 378 (2019), pp. 686–707. DOI: 10.1016/j.jcp.2018.10.045.

[75] Maziar Raissi, Paris Perdikaris, and George Em Karniadakis. “Physics informed deep learning (part II): Data-driven discovery of nonlinear partial differential equations. arXiv”. In: *arXiv Preprint* (2017). DOI: 10.48550/arXiv.1711.10566.

[76] Stephan Rasp. “Coupled online learning as a way to tackle instabilities and biases in neural network parameterizations: general algorithms and Lorenz 96 case study (v1.0)”. In: *Geoscientific Model Development* 13.5 (2020). DOI: 10.5194/gmd-13-2185-2020.

[77] Mateus Dias Ribeiro et al. “DeepCFD: Efficient Steady-State Laminar Flow Approximation with Deep Convolutional Neural Networks”. In: *arXiv Preprint* (2020). DOI: 10.48550/arXiv.2004.08826.

[78] Andrew Ross et al. “Benchmarking of Machine Learning Ocean Subgrid Parameterizations in an Idealized Model”. In: *Journal of Advances in Modeling Earth Systems* 15.1 (2023). e2022MS003258 2022MS003258, e2022MS003258. DOI: 10.1029/2022MS003258. eprint: https://agupubs.onlinelibrary.wiley.com/doi/pdf/10.1029/2022MS003258. URL: https://agupubs.onlinelibrary.wiley.com/doi/abs/10.1029/2022MS003258.

[79] Alessandro Rudi, Luigi Carratino, and Lorenzo Rosasco. “FALKON: an Optimal Large Scale Kernel Method”. In: *Advances in Neural Information Processing Systems*. Ed. by I. Guyon et al. Vol. 30. Curran Associates, Inc., 2017. DOI: 10.48550/arXiv.1705.10958. URL: https://papers.neurips.cc/paper_files/paper/2017/hash/05546b0e38ab9175cd905eebcc6ebb76-Abstract.html.

[80] Alvaro Sanchez-Gonzalez et al. “Graph Networks as Learnable Physics Engines for Inference and Control”. In: *International Conference on Machine Learning*. PMLR. 2018, pp. 4470–4479. DOI: 10.48550/arXiv.1806.01242.

[81] Alvaro Sanchez-Gonzalez et al. “Hamiltonian Graph Networks with ODE Integrators”. In: *Second Workshop on Machine Learning and the Physical Sciences (NeurIPS 2019)*. 2019. DOI: 10.48550/arXiv.1909.12790.

[82] H. Schaeffer, G. Tran, and R. Ward. “Extracting Sparse High-Dimensional Dynamics from Limited Data”. In: *SIAM Journal on Applied Mathematics* 78.6 (2018), pp. 3279–3295. DOI: 10.1137/18M116798X.

[83] Hayden Schaeffer. “Learning partial differential equations via data discovery and sparse optimization”. In: *Proceedings of the Royal Society A: Mathematical, Physical and Engineering Sciences* 473.2197 (2017), p. 20160446. DOI: 10.1098/rspa.2016.0446.

[84] P. Schmid. “Dynamic mode decomposition of numerical and experimental data”. In: *Journal of Fluid Mechanics* 656 (Aug. 2010), pp. 5–28. ISSN: 1469-7645. DOI: 10.1017/S0022112010001217.

[85] P. Schmid and J. Sesterhenn. “Dynamic mode decomposition of numerical and experimental data”. In: *Bull. Amer. Phys. Soc., 61st APS meeting*. American Physical Society, 2008, p. 208.

[86] Teseo Schneider et al. *PolyFEM*. `https://polyfem.github.io/`. 2019.

[87] Bernhard Schölkopf and Alexander J. Smola. *Learning with Kernels*. MIT Press, 1998.

[88] Chaopeng Shen et al. “Differentiable modelling to unify machine learning and physical models for geosciences”. In: *Nature Reviews Earth and Environment* 4 (2023). DOI: 10.1038/s43017-023-00450-9.

[89] Jocelyn Sietsma and Robert J. F. Dow. “Creating Artificial Neural Networks that Generalize”. In: *Neural Networks* 4.1 (1991), pp. 67–79. DOI: 10.1016/0893-6080(91)90033-2.

[90] Uriel Singer et al. “Make-A-Video: Text-to-Video Generation without Text-Video Data”. In: *ICLR*. 2023. DOI: 10.48550/arXiv.2209.14792.

[91] J. Smagorinsky. “General Circulation Experiments with the Primitive Equations: I. The Basic Experiment”. In: *Monthly Weather Review* 91.3 (1963), pp. 99–164. DOI: 10.1175/1520-0493(1963)091<0099:GCEWTP>2.3.CO;2.

[92] Yang Song et al. “Score-Based Generative Modeling through Stochastic Differential Equations”. In: *ICLR*. 2021. DOI: 10.48550/arXiv.2011.13456.

[93] Kimberly Stachenfeld et al. “Learned Coarse Models for Efficient Turbulence Simulation”. In: *International Conference on Learning Representations*. 2022. DOI: 10.48550/arXiv.2112.15275.

[94] Adam Subel and Laure Zanna. “Building Ocean Climate Emulators”. In: *ICLR 2024 workshop on Tackling Climate Change with Machine Learning*. 2024. DOI: 10.48550/arXiv.2402.04342.

[95] Shashank Subramanian et al. “Towards Foundation Models for Scientific Machine Learning: Characterizing Scaling and Transfer Behavior”. In: *Advances in Neural Information Processing Systems*. Curran Associates, Inc., 2023. DOI: 10.48550/arXiv.2306.00258. URL: https://proceedings.neurips.cc/paper_files/paper/2023/file/e15790966a4a9d85d688635c88ee6d8a-Paper-Conference.pdf.

[96] Endre Süli and David F. Mayers. “An Introduction to Numerical Analysis”. In: Cambridge University Press, 2003. Chap. Initial Value Problems for ODEs, pp. 349–353. DOI: 10.1017/CBO9780511801181.

[97] Makoto Takamoto et al. “PDEBench: An Extensive Benchmark for Scientific Machine Learning”. In: *Advances in Neural Information Processing Systems*. Ed. by S. Koyejo et al. Vol. 35. Curran Associates, Inc., 2022, pp. 1596–1611. URL: https://proceedings.neurips.cc/paper_files/paper/2022/file/0a9747136d411fb83f0cf81820d44afb-Paper-Datasets_and_Benchmarks.pdf.

[98] Alexandre M Tartakovsky et al. “Learning parameters and constitutive relationships with physics informed deep neural networks”. In: *arXiv Preprint* (2018). DOI: 10.48550/arXiv.1808.03398.

[99] Nils Thuerey et al. “Deep learning methods for Reynolds-averaged Navier-Stokes simulations of airfoil flows”. In: *AIAA Journal* 58.1 (2020), pp. 25–36. DOI: 10.2514/1.J058291.

[100] Jonathan H. Tu et al. “On dynamic mode decomposition: Theory and applications”. In: *Journal of Computational Dynamics* 1.2 (2014), pp. 391–421. DOI: 10.3934/jcd.2014.1.391.

[101] Nobuyuki Umetani and Bernd Bickel. “Learning three-dimensional flow for interactive aerodynamic design”. In: *ACM Transactions on Graphics (TOG)* 37.4 (2018), pp. 1–10. DOI: 10.1145/3197517.3201325.

[102] Pauli Virtanen et al. “SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python”. In: *Nature Methods* 17 (2020), pp. 261–272. URL: https://doi.org/10.1038/s41592-019-0686-2.

[103] G. L. Wagner et al. “High-level, high-resolution ocean modeling at all scales with Oceananigans”. In: *arXiv preprint* (2025). DOI: 10.48550/arXiv.2502.14148. arXiv: 2502.14148.

[104] Thomas Tomkins Warner. *Numerical Weather and Climate Prediction*. Cambridge University Press, 2011. DOI: 10.1017/CBO9780511763243.

[105] Yuxiao Wen, Eric Vanden-Eijnden, and Benjamin Peherstorfer. “Coupling parameter and particle dynamics for adaptive sampling in Neural Galerkin schemes”. In: *Physica D: Nonlinear Phenomena* 462 (2024). DOI: 10.1016/j.physd.2024.134129.

[106] Karen E. Willcox, Omar Ghattas, and Patrick Heimbach. “The imperative of physics-based modeling and inverse theory in computational science”. In: *Nature Computational Science* 1.3 (Mar. 2021), pp. 166–168. ISSN: 2662-8457. DOI: 10.1038/s43588-021-00040-z. URL: https://doi.org/10.1038/s43588-021-00040-z.

[107] 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. DOI: 10.1145/3197517.3201304.

[108] Ling Yang et al. “Diffusion Models: A Comprehensive Survey of Methods and Applications”. In: *ACM Computing Surveys* 56.4 (Nov. 2023). DOI: 10.1145/3626235.

[109] Sungduk Yu et al. “ClimSim: A large multi-scale dataset for hybrid physics-ML climate emulation”. In: *Advances in Neural Information Processing Systems*. Vol. 36. Curran Associates, Inc., 2023. DOI: 10.48550/arXiv.2306.08754. URL: https://proceedings.neurips.cc/paper_files/paper/2023/file/45fbcc01349292f5e059a0b8b02c8c3f-Paper-Datasets_and_Benchmarks.pdf.

[110] Laure Zanna and Thomas Bolton. "Data-Driven Equation Discovery of Ocean Mesoscale Closures". In: *Geophysical Research Letters* 47.17 (2020). DOI: 10.1029/2020GL088376.

[111] Cheng Zhang et al. "Implementation and Evaluation of a Machine Learned Mesoscale Eddy Parameterization Into a Numerical Ocean Circulation Model". In: *Journal of Advances in Modeling Earth Systems* 15.10 (2023). DOI: 10.1029/2023MS003697.

[112] Robert Zwanzig. *Nonequilibrium Statistical Mechanics*. Oxford University Press, 2001. DOI: 10.1093/oso/9780195140187.001.0001.