Exploring Task-Specific Structure in Neural Networks: Advancing Numerical Interpretation and Uncertainty Quantification

Dissertation

der Mathematisch-Naturwissenschaftlichen Fakultät
der Eberhard Karls Universität Tübingen
zur Erlangung des Grades eines
Doktors der Naturwissenschaften
(Dr. rer. nat.)

vorgelegt von
Katharina Ott, M. Sc.
aus Ludwigsburg

Tübingen
2023

Gedruckt mit Genehmigung der Mathematisch-Naturwissenschaftlichen Fakultät der Eberhard Karls Universität Tübingen.

Tag der mündlichen Qualifikation:	17. November 2023
Dekan:	Prof. Dr. Thilo Stehle
1. Berichterstatter:	Prof. Dr. Philipp Hennig
2. Berichterstatter:	Prof. Dr. Jakob Macke

Acknowledgments

I extend my heartfelt gratitude to my supervisors Prof. Dr. Philipp Hennig and Dr. Michael Tiemann for granting me the opportunity to pursue this PhD. Your unwavering support and engaging discussions have played a pivotal role in shaping the trajectory of this thesis. Working alongside both of you has been a delight.

I am thankful to Prof. Dr. Jakob Macke for his commitment as examiner, dedicating time and effort to assess this thesis.

A special acknowledgment goes to Prof. Dr. François-Xavier Briol for the insightful collaboration, it provided a great learning opportunity.

My sincere appreciation extends to the Bosch R1 group and BCAI, especially the PhDs, for their support, academic exchanges, software engineering assistance, and the enjoyable bouldering sessions.

I am grateful to the MoML group at the University of Tübingen for the enriching academic dialogues and moments of respite from academic pressures. Thank you for fostering a conducive work environment, with a particular nod to the running group and the bouldering gang.

Lastly, to my partner Chris, your unwavering support and attentive ear during challenging times have been invaluable.

Thank you!

Katharina Ott
Tübingen, November 2023

Abstract

The choice of network architecture is one of the key ingredients that has led to significant advances in machine learning. Architectures often encode available knowledge about the world or the learning process. A prominent example is convolutional neural networks, whose structure is motivated by the fact that image classification should remain invariant under small translations.

In many real-world applications, detailed knowledge of the underlying physics or mathematical description is available and should be incorporated into the network architecture. For instance, dynamical systems are often characterized by differential equations. This information can be encoded in neural ODEs. These architectures are often used for scientific applications, so understanding the output of the neural network is crucial. Uncertainty quantification can help to build confidence in the network's outputs and indicate cases of failure. Therefore, the aim is to develop methods for uncertainty quantification alongside the network architecture.

A core idea of the neural ODE framework is to provide a continuous perspective on deep learning. However, if a too coarse numerical ODE solver is used for training, the resulting network will no longer fit this paradigm. Specifically, for a continuous interpretation, a numerically more accurate ODE solver would achieve a similar performance during test time as the solver used for training. However, there are cases where we observe a significant drop in performance when testing with a numerically more accurate solver. We investigate this issue and develop an algorithm to maintain the continuous interpretation of the architecture throughout training.

In scientific applications, additional information about a task is often available, e.g., in the form of conservation laws or partially known dynamics. Recent work has incorporated such information into the design of neural ODEs. These scientific applications require a high degree of confidence in the networks' outputs, hence we propose to equip neural ODEs with uncertainty estimates. We find that even small changes in the setup can lead to drastically different results, a fact that is reflected in the uncertainty estimates.

An often difficult task in both science and machine learning is the numerical computation of integrals. A plethora of methods exit, but Bayesian quadrature stands out as an uncertainty-aware approach. Bayesian quadrature provides estimates of numerical uncertainty using Gaussian process regression. However, Bayesian quadrature does not scale well to higher dimensional settings and large amounts of data, a task at which neural networks often excel. This motivates the development of a Bayesian numerical integration method using neural networks. The architecture uses the Langevin-Stein operator and uncertainty estimates are obtained using the Laplace approximation.

This thesis highlights the understanding and development of more task-specific neural network architectures, and shows how uncertainty estimates help to understand the outputs of neural networks.

Zusammenfassung

Die Wahl der Netzarchitektur ist eine der wichtigsten Komponenten, die zu bedeutenden Fortschritten im Bereich maschinellen Lernen geführt haben. Architekturen liegt oft vorhandenes Wissen über die Welt oder den Lernprozess zugrunde. Ein bekanntes Beispiel sind faltende neuronale Netze, deren Struktur durch die Tatsache motiviert ist, dass die Bildklassifizierung bei kleinen Verschiebungen unverändert bleiben sollte.

In vielen realen Anwendungen ist detailliertes Wissen über die zugrundeliegende Physik oder mathematische Beschreibung des Problems vorhanden und sollte in die Netzarchitektur einbezogen werden. Dynamische Systeme werden beispielsweise häufig durch Differentialgleichungen charakterisiert, was durch neuronale gewöhnliche Differentialgleichungen (englisch ordinary differential equations, ODEs) abgebildet werden kann. Diese Architekturen werden häufig für wissenschaftliche Anwendungen verwendet, wobei ein besseres Verständnis der Ausgabe des neuronalen Netzes ein wichtiger Gesichtspunkt ist. Die Quantifizierung der Unsicherheit kann dazu beitragen, das Vertrauen in die Ergebnisse des Netzes zu stärken und Fehler zu erkennen. Ziel ist es daher, Methoden zur Quantifizierung von Unsicherheiten in Verbindung mit der Netzarchitektur zu entwickeln.

Ein Kerngedanke der neuronalen ODE Architekturen besteht darin, eine kontinuierliche Perspektive für Deep Learning zu bieten. Wenn jedoch ein zu grober numerischer ODE-Löser für das Training verwendet wird, passt das resultierende Netz nicht mehr zu diesem Paradigma. Insbesondere gegeben einer kontinuierlichen Interpretation müsste ein numerisch genauerer ODE-Löser beim Testen ein ähnliches Ergebnis erzielen wie der für das Training verwendete Löser. Es gibt jedoch Fälle, in denen ein signifikanter Leistungsabfall beobachtet wird, wenn man mit einem numerisch genaueren Löser testet. Wir untersuchen dieses Problem und entwickeln einen Algorithmus, der die kontinuierliche Interpretation der Architektur während des Trainings beibehält.

In wissenschaftlichen Anwendungen sind oft zusätzliche Informationen über das zu lösende Problem verfügbar, z.B. in Form von Erhaltungsgesetzen oder teilweise bekannter Beschreibung der Dynamik. In neueren Arbeiten wurden solche Informationen in den Entwurf von neuronalen ODEs einbezogen. Diese wissenschaftlichen Anwendungen erfordern ein hohes Maß an Vertrauen in die Ergebnisse der Netze, daher schlagen wir vor, neuronale ODEs mit Unsicherheitsschätzungen auszustatten. Wir stellen fest, dass selbst kleine Änderungen in der Problemstellung zu drastisch unterschiedlichen Ergebnissen führen können, was sich in den Unsicherheitsschätzungen widerspiegelt.

Eine oft schwierige Aufgabe sowohl in der Wissenschaft als auch beim maschinellen Lernen ist die numerische Berechnung von Integralen. Es gibt eine Fülle von Methoden, aber die bayessche Quadratur zeichnet sich als ein Ansatz aus, der die Unsicherheit der Methode selbst berücksichtigt. Die bayessche Quadratur liefert Schätzungen der numerischen Unsicherheit unter Verwendung der Gaußprozess-Regression. Die bayessche Quadratur lässt sich jedoch nicht gut auf höherdimensionale Probleme und große Datenmengen übertragen, im Gegensatz zu neuronale Netze, die in diesen Bereich oft herausstechen. Die Architektur verwendet den Langevin-Stein-Operator, und die Unsicherheitsschätzungen werden mit Hilfe der Laplace-Approximation ermittelt.

Diese Arbeit beleuchtet das Verständnis und die Entwicklung aufgabenspezifischer Architekturen für neuronaler Netze und zeigt, wie Unsicherheitsschätzungen helfen können, die Ergebnisse neuronaler Netze besser zu verstehen.

Table of Contents

Notation

Acronyms & Abbreviations

E.g. or *e.g.*	For example (*exempli gratia*)
I.e. or *i.e.*	That is (*id est*)
BNN	Bayesian neural network
BQ	Bayesian quadrature
BSN	Bayesian Stein network
CF	Control functional
CPU	Central processing unit
EOM	Equations of motion
ERM	Empirical risk minimization
GGN	Generalized Gauss-Newton (matrix)
GP	Gaussian process
GPU	Graphics processing unit
HMC	Hamiltonian Monte Carlo
IVP	Initial value problem
L-BFGS	Limited-memory Broyden-Fletcher-Goldfarb-Shanno algorithm
MALA	Metropolis adjusted Langevin algorithm
MAP	Maximum a posteriori (estimation)
MC	Monte Carlo
MCMC	Markov chain Monte Carlo
MLE	Maximum likelihood estimation
mRNA	Messenger ribonucleic acid
MSE	Mean squared error
NFE	Number of function evaluations
NUTS	No U-turn sampler
ODE	Ordinary differential equation
PDE	Partial differential equation
PINN	Physics informed neural network
PNM	Probabilistic numerical method
PSD	Positive semi-definite
QMC	Quasi-Monte Carlo
RELU	Rectified linear unit
ResNet	Residual (neural) network
SDE	Stochastic differential equations

Overview 1.

1.1 Introduction

Modern neural network architectures excel at a long list of tasks, including image classification and generation [67, 117], text translation and completion [45], and time series analysis (e.g., for weather and climate modeling [65]), to name just a few. We might think of neural networks as this all-encompassing black box which does not require adjusting the network structure to a specific task. And while some modern neural network models seem to be getting closer to this idea [47, 151], the excellent performance of many models is only made possible by careful design of their architecture. Historically, the structure of neural networks has been inspired by the biological structure of neural networks in the human brain [66]. While some architectures still draw inspiration from biological neural networks (e.g., spiking neural networks [131]), the influences of modern architectures are diverse. Improvements in network architecture are motivated by various goals, such as better performance [117], faster training [86], better extrapolation [231], increased data and/or memory efficiency [221], obeying mathematical properties [155, 171], or obeying physical laws [227]. To improve some of these aspects, researchers incorporate knowledge about the data generating process [29], symmetries in the data [59, 117, 187], or the mathematical representation of the task into the structure of the neural network [1, 155, 171].

A key motivation for making changes to the network architecture is that training neural networks can be difficult, especially with increasing depth. However, a small change in the architecture makes it possible to train much deeper networks. The resulting architecture, known as a residual neural network [86], has led to major advances in image classification. A residual network consists of neural network blocks $f^i_{\theta^i}$ that transform an input x^i by the following transformation:

$$x^{i+1} = x^i + f^i_{\theta^i}(x^i).$$

This transformation not only allows for easier training of the network but also has some interesting numerical properties. A residual neural network is similar to the Euler method used to solve ordinary differential equations. Viewing residual networks as the numerical solution of an ordinary differential equation provides an entirely new, continuous view of the architecture. This continuous perspective suggests combining several blocks into an ordinary differential equation, called a neural ODE

$$x' = f_\theta(t, x), \quad x(t = t_0) = x_0.$$

The two descriptions do not overlap completely, but the similarity is still surprising. Since in reality we still rely on numerical methods to solve neural ODEs, it is crucial to understand the impact of these numerical solvers on the continuous perspective and the model in general. Chen et al. [28] has developed this continuous view into a new perspective for existing deep learning applications such as image classification, normalizing flow, and time series analysis. Their goal is to provide a more memory efficient approach and new insights into existing architectures. Neural ODEs also lend themselves directly to the modeling of dynamical systems, which are often already modeled by differential equations (e.g., equations of motion, reaction-diffusion equations in chemistry, or fluid dynamics). In such scientific settings, the goal is to model unknown dynamics in a data-driven way. Here, we make use of knowledge of the data generating process and incorporate it into the network structure, with the aim of obtaining models that extrapolate better and align with our understanding of the underlying physics.

The structure of a neural network can also be motivated by considering novel tasks. Integration is a task rarely tackled by deep learning methods, but it is important throughout machine learning, e.g. for computing normalization constants or posterior means in Bayesian inference tasks. Despite a number of existing numerical methods, there are still cases where a neural network approach is useful. In higher-dimensional integration settings, Monte Carlo methods are the default integration method. These methods are often numerically cheap if function evaluations are cheap, but since they make no assumptions about the functional form of the integrand, they often require a large number of function evaluations. On the other hand, Bayesian quadrature [92, 150], a probabilistic numerical method, encodes assumptions about the functional class of the integrand, leading to good results in the low-data regime. However, Bayesian quadrature does not scale well to large numbers of function evaluations or to higher dimensional settings. Therefore, higher dimensional settings, where a lot of data is available, but sampling is not extremely cheap, require the development of novel algorithms. This thesis introduces a novel neural network architecture where the structure of the integration problem itself is directly encoded in the network architecture.

In the previous paragraphs, we motivated two network architectures, one encoding the structure of ordinary differential equations and the other encoding the structure of the integration problem. Although these architectures seem similar to physical models or the mathematical expressions we want to compute, they are still data-driven approaches whose neural network output can be difficult to understand. Neural networks, if chosen large enough, can fit arbitrary data, but this also means that the neural network can converge to an arbitrarily bad or complex solution, resulting in inaccurate predictions. This requires some form of uncertainty quantification on the network outputs.

This thesis aims to develop, apply, and understand neural network architectures for dynamics modeling and integration. The goal is to highlight the strengths but also the limitations of the imposed structural knowledge.

1.2 Outline

The first part of this thesis introduces the essential ingredients used in this thesis, providing background on numerical integration, numerical approximation of ODE solutions and deep learning.

Chapter 2 lays the foundation for later chapters, giving a short introduction to numerical integration. The chapter starts with an overview of classic numerical quadrature in Section 2.1. These classic algorithm become relevant in Chapter 3 for the development of numerical methods for ODEs. Section 2.2 introduces the Laplace approximation, a simple and extremely cheap technique to obtain approximations of integrals. We revisit this technique in the context of developing a Bayesian framework for neural networks in Section 4.3. The remainder of this chapter, Section 2.3 and Section 2.4.1, describes how to perform numerical integration via Monte Carlo methods and Bayesian quadrature.

Chapter 3 treats ordinary differential equations and numerical methods to approximate ODE solutions. Section 3.1 establishes the formal concept and provides theoretical background on the existence of ODE solutions. Since it is often impossible to find analytical solutions, the main objective of this chapter is to numerically approximate the ODE solution, building on the foundations from numerical integration. Section 3.2 provides an overview of numerical ODE solvers, i.e., fixed step and adaptive step size Runge-Kutta methods. These numerical solvers play an important role in the construction of neural ODEs in Chapter 4, and severely affect the behavior of such models, as discussed in Chapter 5.

Chapter 4 presents the basic ingredients to build (Section 4.1) and train (Section 4.2) neural networks. Since neural network prediction can be highly inaccurate outside the data domain, Section 4.3 establishes a framework to equip neural networks with uncertainty estimates. Section 4.4 motivates extensions to the basic architectures developed in the previous part of this chapter. These architectures are inspired by the structure in the data, the data generating process or by aiming to solve a specific numerical task. We then take a closer look a one specific neural network structure—neural ODEs—and its applications.

The second part of this thesis provides novel contributions to understanding and developing neural networks with specialized structural components. First, we take a detailed look at neural ODEs in Chapter 5 and Chapter 6. Chapter 7 develops a novel neural network architecture for Bayesian numerical integration.

A key appeal of neural ODEs is that they provide a continuous framework for deep learning. In Chapter 5 we take a closer look at this viewpoint, and find that it actually breaks down if a too coarse numerical solver is used for training the neural ODE. In such cases, a numerically more accurate solver no longer recovers the same result as the solver used for training, but leads to a significant drop in performance. Chapter 5 investigates this issue in detail, to then develop an algorithm which automatically adapts the solver precision during training. The aim of this algorithm is to maintain the continuous interpretation of the model throughout training.

Disclaimer 1.1 Chapter 5 is based on the peer-reviewed conference publication with the following co-author contributions:

K. Ott, P. Katiyar, P. Hennig, and M. Tiemann. "ResNet After All: Neural {ODE}s and Their Numerical Solution". *International Conference on Learning Representations*. 2021 [153]

	Ideas	Experiments	Analysis	Writing
K. Ott	30 %	60 %	30 %	35 %
P. Katiyar	15 %	40 %	15 %	20 %
P. Hennig	25 %	0 %	25 %	25 %
M. Tiemann	30 %	0 %	30 %	20 %

The neural ODE framework can be extended using mechanistic knowledge—via conservation laws implemented by the Hamiltonian equations of motion and by augmenting physical models with neural ODEs. Chapter 6 investigates these extensions by developing uncertainty estimates for neural ODEs. Uncertainty estimates are particularly interesting for these specialized architectures, as their extrapolation and interpolation capabilities significantly differ from standard neural ODEs. For example, additional structure can improve the extrapolation capabilities of a neural ODE, which can best be assessed using structured uncertainty estimates. To compute the uncertainty estimates, Chapter 6 proposes to use the Laplace approximation and shows how to adapt this approach to neural ODEs.

Disclaimer 1.2 Chapter 6 is based on a prepreint with the following co-author contributions:

K. Ott, M. Tiemann, and P. Hennig. "Uncertainty and Structure in Neural Ordinary Differential Equations". *arXiv:2305.13290* (2023) [154]

	Ideas	Experiments	Analysis	Writing
K. Ott	45 %	85 %	60 %	70 %
M. Tiemann	15 %	5 %	10 %	10 %
P. Hennig	45 %	10 %	30 %	20 %

Chapter 7 considers the task of numerical integration and how to leverage the interpolation power of neural networks in this context. Bayesian quadrature tends to struggle in higher dimensions and for large amounts of data, whereas Monte Carlo methods often require large amounts of data as they make few assumptions on the integrand. To bridge the gap between the two methods, when plenty of data is available but not yet sufficient to obtain good results with Monte Carlo, Chapter 7 proposes a Bayesian neural network for numerical integration. The network architecture itself is based on the Langevin Stein operator, and uncertainty estimates are obtained via the Laplace approximation. This network requires special care in the choice of activation function, architecture and optimization procedure. The resulting architecture utilizes the networks capabilities to handle large amounts of data and scales well to higher dimensional problems.

Disclaimer 1.3 Chapter 7 is based on a preprint to be published as peer-reviewed conference publication at UAI 2023 with the following co-author contributions:

K. Ott, M. Tiemann, P. Hennig, and F.-X. Briol. “Bayesian Numerical Integration with Neural Networks”. *arXiv:2305.13248* (2023) [155]

	Ideas	**Experiments**	**Analysis**	**Writing**
K. Ott	35 %	80 %	30 %	35 %
M. Tiemann	10 %	5 %	15 %	10 %
P. Hennig	20 %	10 %	25 %	20 %
F.-X. Briol	35 %	10 %	30 %	35 %

A brief summary is provided in Chapter 8. This chapter also contains pointers to obstacles when adding complex structure to the architecture of neural networks and derives ideas for future work.

Part I.

Background & Motivation

Numerical Integration 2.

Integration problems occur throughout science, but is especially prominent in Bayesian probability theory, e.g., for computing normalization constants or posterior expectations. While it might be possible to analytically solve the integration task for simple problems, in practice one often has to resort to some numerical method. In Section 2.1 we introduce classic numerical integration methods, which essentially work by fitting integrable functions through evaluations of the integrand. To avoid the curse of dimensionality of classic methods in high dimensions, one can instead stochastically approximate the integral via Monte Carlo methods which are introduced in Section 2.3. Bayesian Quadrature (Section 2.4) leverages knowledge about the functional form of the integrand and obtains uncertainty estimates for the integral evaluation. The last section of this chapter discusses how to choose the appropriate integration method for a given task. Throughout this chapter, we use the terms quadrature and numerical integration interchangeably.

2.1 Classic Quadrature

We consider the methods presented in this section not only for their relevance to numerical integration, but later in Section 3.2.1 we will see that they played an important role in the development of numerical ODE solvers. For a more exhaustive discussion of classic numerical methods we refer to [38]. In this section we consider one dimensional integration problems of a function $f : \mathcal{X} \subset \mathbb{R} \to \mathbb{R}$ over the domain $[a, b]$:

$$\Pi[f] = \int_a^b f(x)dx.$$

One simple way to approximate the integral $\Pi[f]$ is via a piece-wise constant function, known as the *Midpoint method* (illustrated in Figure 2.1).

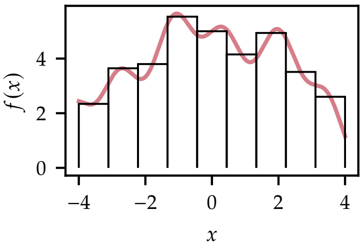


Figure 2.1: Illustration of the Midpoint method.

Definition 2.1 Midpoint method Consider a grid of size N, $x_0 = a, x_N = b, x_n = n\frac{a-b}{N}$. We then approximate the integral via a piece wise constant

$$\Pi[f] \approx \sum_{n=0}^{N-1} f\left(\frac{x_{n+1} - x_n}{2}\right)(x_{n+1} - x_n).$$

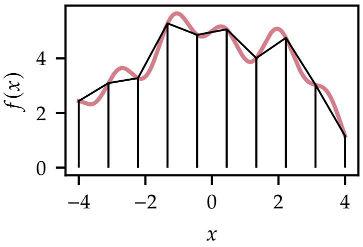


Figure 2.2: Illustration of the Trapezoidal method.

Instead of considering just constant approximations, one can instead use affine functions known as the *Trapezoidal rule* (see Figure 2.2) or even higher order polynomials. *Simpson's rule* locally approximates the function f with second order polynomials (shown in Figure 2.3).

Definition 2.2 **Simpson's rule** Consider the grid of size N, $x_0 = a, x_N = b, x_n = n\frac{a-b}{N}$. We then approximate the integral via second order polynomials

$$\Pi[f] \approx \sum_{n=0}^{N-1} \left(f(x_n) + 4\left(\frac{x_{n+1}-x_n}{2}\right) - f(x_{n+1})\right) \frac{x_{n+1}-x_n}{6}.$$

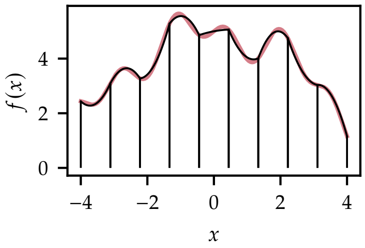


Figure 2.3: Illustration of the Simpson's method.

2.1.1 Gaussian quadrature

Instead of considering a regular grid, the evaluation points x_n can be chosen based on orthogonal polynomials. These methods are known as Gaussian quadrature (see [38]). We can write $f(x) = \tilde{f}(x)w(x)$, where $w : \mathcal{X} \subseteq \mathbb{R} \rightarrow \mathbb{R}$ is some weighting function. By approximating $\tilde{f}$ with a polynomial of degree N, we can make use of the fact that the integral of the polynomial can be computed in closed form

$$\int_{\mathcal{X}} f(x)dx = \int_{\mathcal{X}} \tilde{f}(x)w(x)dx \approx \sum_{n=1}^{N} \tilde{f}(x_n)\omega_n. \tag{2.1}$$

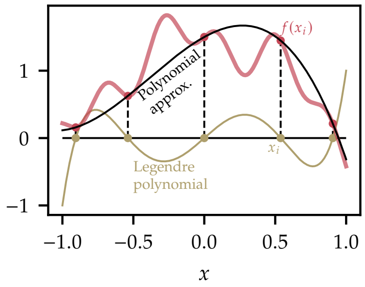


Figure 2.4: Illustration of Gauss-Legendre quadrature of degree $n = 5$. A Legendre polynomial (yellow) is used to compute the evaluation points x_i. The function f (red) is then approximated with a polynomial (black solid line).

The x_n correspond to the roots for the orthogonal polynomial of degree N with the same weighting function w, e.g., the Legendre polynomials for $w(x) = 1$ (illustrated in Figure 2.4). ω_n are the weights, obtained by integration. The method fits polynomial $\tilde{f}$ up to degree $2N - 1$ exactly [38]. For many orthogonal polynomials, precomputed tables for the weights ω_n and the evaluation points x_n exist.

The methods described in this section can be extended to higher dimensions using Fubini's theorem (evaluating one integral after the other), which leads to an exponential increase in function evaluations known as the *curse of dimensionality*. Higher dimensional extensions for classic quadrature methods exist [196], but the next sections discuss alternative methods which overcome this issue.

2.2 Laplace Approximation for Integration

We consider the task of integrating a function $f : \mathcal{X} \subseteq \mathbb{R}^d \rightarrow \mathbb{R}^+$,

$$\Pi[f] = \int_{\mathcal{X}} f(x)dx.$$

The Laplace approximation works by approximating f with a Gaussian (Figure 2.5). We consider an integrable function f, where $f(x) = e^{Cg(x)}$ and $C \in \mathbb{R}^+$. $g : \mathcal{X} \subseteq \mathbb{R}^d \rightarrow \mathbb{R}$ is a twice differentiable function. Let $x^* \in \mathcal{X}$ be the maximum of both $f(x)$ and $g(x)$. We then expand g around x^* up to second order

$$\begin{aligned} g(x) \approx & g(x^*) + \nabla_x g|_{x=x^*}(x - x^*) \\ & + \frac{1}{2}(x - x^*)^\top \nabla_x^2 g|_{x=x^*}(x - x^*). \end{aligned}$$

The second term $\nabla_x g|_{x=x^*}$ is zero by the choice of x^*. The function f can then be approximated via a Gaussian function

$$f(x) \approx \exp(Cg(x^*) + \frac{1}{2}(x - x^*)^\top C\Sigma^{-1}(x - x^*)) =: q(x),$$

where $\Sigma^{-1} = \nabla_x^2 g|_{x=x*}$ is a negative definite matrix. The integral over q can then be computed in closed form by making use of the fact that the integral of a Gaussian is known (and precomputed tables for finite domains exist). One can show that $\Pi[q] \to \Pi[f]$ for $C \to \infty$ [180]. An advantage of this approach is that it provides a fast and simple approximation of the integral, but the accuracy is limited especially for multimodal distributions.

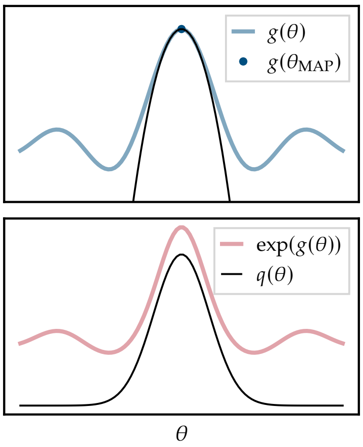


Figure 2.5: (*Top*) quadratic approximation for g. (*Bottom*) Approximation of the function f with a Gaussian.

2.3 Monte Carlo Methods

This section considers the task of computing the integral over a function $f : \mathcal{X} \subseteq \mathbb{R}^d \to \mathbb{R}$ and a probability density function $\pi : \mathcal{X} \to \mathbb{R}_0^+$, $\int_\mathcal{X} \pi(x)dx = 1$,

$$\Pi_\pi[f] = \int_\mathcal{X} f(x)\pi(x)dx.$$

The previously introduced quadrature rules do not take into account the probability density function in the choice of evaluation points. Additionally, these methods often do not scale well to higher dimensions. Monte Carlo (MC) methods overcome both of these issues. Instead of deterministically approximating the function, Monte Carlo methods randomly sample evaluation points from π. This section briefly introduces the main concepts, focusing on methods that remain relevant to later chapters.

2.3.1 Monte Carlo Sampling

We consider the case when it is possible to sample form π, then

$$\mathbb{E}_\pi[f] := \Pi_\pi[f] \approx \frac{1}{N}\sum_{n=1}^{N} f(x_n) =: \hat{\mu}, \quad x_n \sim \pi. \tag{2.2}$$

$\hat{\mu}$ provides an *unbiased* estimator for $\mathbb{E}_\pi[f]$, i.e.,

$$\mathbb{E}_\pi[\hat{\mu}] = \mathbb{E}_\pi\left[\frac{1}{N}\sum_{n=1}^{N} f(x_n)\right] = \frac{1}{N}\sum_{n=1}^{N} \mathbb{E}_\pi[f] = \mathbb{E}_\pi[f].$$

To analyze the convergence, we compute the variance of $\hat{\mu}$

$$\mathbb{V}_\pi[\hat{\mu}] := \Pi_\pi\left[(\hat{\mu} - \Pi_\pi[f])^2\right] = \frac{\sigma_f^2}{N}, \tag{2.3}$$

where σ_f^2 is the variance of f

$$\sigma_f^2 = \Pi_\pi\left[(f - \Pi_\pi[f])^2\right].$$

From Equation 2.3 we obtain a convergence rate of $1/\sqrt{N}$, and note that this convergence rate is independent of the dimensionality. The variance of f might, however, be arbitrarily small or large. The use of control variates can lead to a variance reduction, and works by adding a term with expectation value of zero [180]. When sampling from π directly is not possible, the expectation value in Equation 2.2 can still be computed through importance sampling.

Definition 2.3 Importance Sampling We would like to approximate $\Pi_\pi[f]$ using MC sampling, however sampling from π is not possible/infeasible. Instead, we rewrite the integral as

$$\Pi_{pi}[f] = \int_{\mathcal{X}} f(x)\pi(x)dx = \int_{\mathcal{X}} f(x)\frac{\pi(x)}{\rho(x)}\rho(x)dx,$$

where the probability density function ρ is chosen such that it has greater or equal support than π and sampling from ρ is possible. Then $\Pi_\pi[f]$ can be approximated via *importance sampling*

$$\Pi_\pi[f] \approx \frac{1}{N}\sum_{n=1}^{N} f(x_n)\frac{\pi(x_n)}{\rho(x_n)}, \quad x_n \sim \rho.$$

Quasi-Monte Carlo

In practice sampling from π requires an algorithm which produces these samples. These algorithms work by deterministically computing a sequence of samples given a seed where the samples are distributed according to π. Such sequences are also known as pseudorandom, as the underlying algorithm is deterministic.

Instead of using pseudorandom numbers to approximate the integral, quasi-Monte Carlo proposes to choose the evaluation points based on low discrepancy sequences, i.e., sequences that cover the integration domain evenly. Where random sampling can lead to large patches without samples, this issue is alleviated by a low discrepancy sequence (see Figure 2.6 for a comparison). Quasi-Monte Carlo methods can lead to a better convergence rate than standard Monte Carlo sampling.

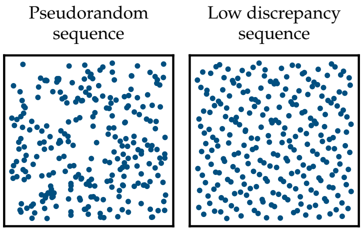


Figure 2.6: Sampling $n = 256$ evaluation points using a pseudorandom sequence (*left*) and a low discrepancy sequence (*right*) (here the Sobol sequence [193]).

2.3.2 Markov Chain Monte Carlo

Often, sampling from π is not possible, and π might only be known up to a normalization constant. An example is the computation of posterior expectations, where likelihood and prior are known, but the normalization constant of the posterior is not. In such cases Markov Chain Monte Carlo methods (MCMC) are useful. While importance sampling can deal with the case when sampling from π is not possible, it still requires π to be available in normalized form. This is an advantage of MCMC methods, which do not require knowledge of the normalization constant of π.

Definition 2.4 **Markov chain** A Markov chain is a set of random variables $\{x_n\}_{n=0}^N \in \mathcal{X}$, with prior $x_0 \sim p(x_0)$ and transition probability $p : \mathcal{X} \times \mathcal{B}(\mathcal{X}) \longrightarrow [0,1]$ which fulfill the *Markov property*

$$p(x_{n+1} \in A \mid x_0 \ldots x_n) = p(x_{n+1} \in A \mid x_n). \tag{2.4}$$

for any $A \in \mathcal{B}(\mathcal{X})$, where $\mathcal{B}(\mathcal{X})$ denotes the σ-algebra [183] on $\mathcal{X}$.

To use Markov chains for the integration tasks, they need to be *stationary*, i.e., they need to sample from π, and they need to be *ergodic*, i.e., exhaustively sample from all of $\mathcal{X}$. To be more precise

Definition 2.5 **Stationary distribution** π is a *stationary distribution* of a Markov chain if for all $A \in \mathcal{B}(\mathcal{X})$

$$\pi(A) = \int_{\mathcal{X}} p(A \mid x)\pi(x)dx,$$

i.e., $x_n \sim \pi$ if $x_0 \sim \pi$.

Definition 2.6 **Irreducible Markov chain** A Markov chain is called *irreducible* if for any $x_0 \in \mathcal{X}, A \in \mathcal{B}(\mathcal{X})$ there exists some n such that

$$p(x_n \in A \mid x_0) > 0.$$

Loosely speaking, for an irreducible Markov chain, we can reach any "state" with finite probability. A Markov chain is *ergodic*, if it is aperiodic and irreducible. For such chains the expectation under samples from the chain is equal to sampling from π directly

$$\lim_{n\to\infty} \frac{1}{n} \sum_{n=1}^{N} f(x_n) = \Pi_\pi[f],$$

where x_n are samples from an ergodic Markov chain with stationary π. The goal of Markov chain Monte Carlo is to find such Markov chains.

Metropolis Hastings Algorithm

This well known algorithm was already developed in the 50s [138] (and generalized by Hastings [85]) but became popular in the 90s [61] when computers where more readily available. To obtain samples from π the algorithm requires that a distribution $q \propto \pi$ is available and that q can be evaluated. Additionally, we need to define a transition probability distribution $g(x \mid y)$, which allows sampling, e.g., a Gaussian distribution. The algorithm then takes a tentative step $\hat{x}_{n+1} \sim g(x_{n+1} \mid x_n)$. The step is accepted or rejected based on the following acceptance ratio

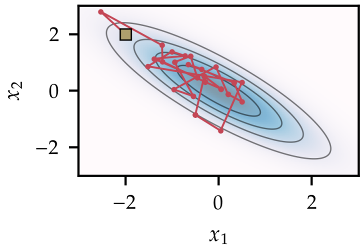


Figure 2.7: Illustration of the Metropolis Hastings Algorithm. Realization of a Markov chain, using a Gaussian transition kernel and $N = 50$ steps.

$$\alpha = \min\left[1, \frac{q(\hat{x}_{n+1})g(x_n \mid \hat{x}_{n+1})}{q(x_n)g(\hat{x}_{n+1} \mid x_t)}\right].$$

A step is accepted if a random number $u \sim [0,1]$ is smaller than α. The accept-reject step encourages moving to regions of high probability with respect to π. In case of rejection, the old step is reused. The choice of g can be tricky especially in high dimensions where different jump lengths for different directions might be needed. A sample path of random walk Metropolis Hastings using a Gaussian distribution for $g(x \mid y)$ is depicted in Figure 2.7.

Metropolis Adjusted Langevin Dynamics

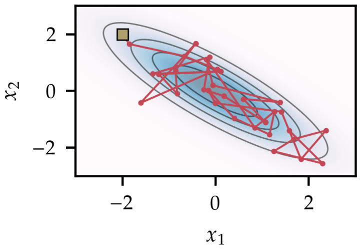


Figure 2.8: Illustration of the MALA Algorithm. Realization of a Markov chain using $N = 50$ steps.

The Metropolis Adjusted Langevin algorithm (MALA) [73, 163, 181] computes proposal steps based on the Langevin equations.

Here we consider the Langevin diffusion process

$$x' = \nabla \log \pi(x) + \sqrt{2}\eta \tag{2.5}$$

where η is a Gaussian-distributed noise term with correlation $\langle \eta(t), \eta(t') \rangle \propto \delta(t - t')$. From the Fokker Planck equation we know that π is an invariant distribution of the process x_t. To find sample paths based on Equation 2.5, numerical stochastic differential equation solvers such as Euler Maruyama are used. To get a proposal step, MALA works by taking a single Euler Maruyama step, and then doing a Metropolis-Hastings accept-reject step to correct for solver errors (a corresponding sample path is shown in Figure 2.8). Chapter 7 applies this algorithm to finding posterior expectations for the parameters of an ODE.

Hamiltonian Monte Carlo

The Hamiltonian Monte Carlo (HMC) algorithm is a Metropolis Hastings algorithm, which computes steps via the Hamiltonian equations of motion.

Given a system with energy $E(x)$ and states x, then the probability density function over the states is given via the *Boltzmann distribution*

$$p(x) = \frac{1}{Z} \exp\left(-\frac{E(x)}{\tau}\right), \tag{2.6}$$

where τ is a temperature parameter. In the following we consider the case where we describe a probability distribution in terms of energy, i.e., $E(x) = -\log p(x) - \log(Z)$, and hence $\tau = 1$ describes the distribution of interest.

The Hamiltonian is a conserved quantity of a physical system (which can often be interpreted as the energy of the system), which describes the *equations of motion (EOM)* in terms of the (generalized)* space and momentum coordinates

$$q' = \frac{\partial H}{\partial p}, \quad p' = -\frac{\partial H}{\partial q}. \tag{2.7}$$

In practice, one often considers separable systems where the Hamiltonian H can be expressed as a sum of potential energy $U(q)$ and kinetic energy

* Generalized coordinates are an important concept in theoretical mechanics and allow a simpler description of the system than using standard Cartesian coordinates.

$T(p) = \frac{p^2}{2m}$, where m corresponds to the mass. To solve the Hamiltonian EOM one requires special solvers, see Section 3.2.1.

To obtain samples from π, we start by expressing the potential U via the probability density π as $U(q) = -\nabla_q \log \pi(q)$. The Boltzmann distribution Equation 2.6 provides the joint distribution of q and p, i.e., $p(q, p) \propto \exp -H(q, p)$. To compute proposals the EOM Equation 2.7 are solved for L steps with step size h (h and L are hyperparameters of the problem), shown in Figure 2.9. The proposal is then accepted or rejected as in the Metropolis Hastings algorithm. To marginalize out p, the momentum is resampled at each step.

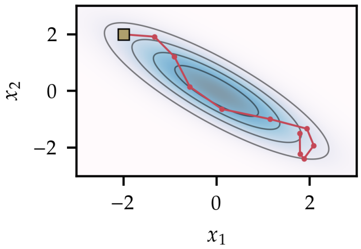


Figure 2.9: Illustration of the HMC Algorithm. The red line depicts a solution to the Hamiltonian equations of motions to find the next proposal step.

The HMC algorithm requires that the derivatives of $\log \pi$ can be computed (we note that this means we only need to know π up to a normalization constant). HMC samples tend to be less correlated than random walk Metropolis Hastings. However, for certain settings and sets of hyperparameters, HMC chains can potentially be non-ergodic or extremely slow to sample [140].

No-U-Turn Sampler If L is chosen too large, the trajectory can start oscillating back and forth, effectively wasting computation time and potentially leading to non-ergodic Markov chains. To avoid this issue, Hoffman, Gelman, et al. [95] suggest running the dynamics forward and backwards in time. If the trajectories start turning around, a random point from the trajectory is sampled and the run is continued from there. This algorithm is nowadays often the default algorithm, and we will later see in Chapter 6 that it can be used to compute posterior distributions for small Bayesian neural networks.

2.4 Bayesian Quadrature

Bayesian quadrature works by placing a Gaussian process (GP) [174] prior on the integrand, which not only allows the approximation of the integral but also provides an uncertainty estimate for the computation. First, we define kernel functions and Gaussian processes. We then show how to use GPs for regression tasks, and apply GPs to integration.

Definition 2.7 Kernel A *kernel* is a symmetric function $k : \mathcal{X} \subseteq \mathbb{R}^d \times \mathcal{X} \subseteq \mathbb{R}^d \to \mathbb{R}$, where $k(x_n, x_m) = k(x_m, x_n)$ for any $x_n, x_m \in \mathcal{X}$. In the following we consider *positive definite* kernel functions, i.e., for any sequence $\{x_n\}_{n=1}^N \subset \mathcal{X}$ the $N \times N$ matrix K, with $K_{n,m} = k(x_n, x_m)$, is positive definite.

Definition 2.8 Gaussian Process Given a mean function $m : \mathcal{X} \subseteq \mathbb{R}^d \to \mathbb{R}$ and a kernel $k : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$, $g \sim \mathcal{GP}(m, k)$ is a *Gaussian process*, if for any choice of $X = \{x_n\}_{n=1}^N \subset \mathcal{X}$ the vector $g_X = [g(x_1), \ldots g(x_N)]$ is distributed according to a multivariate normal

$$g_X \sim \mathcal{N}(m_X, k_{XX}),$$

where $[m_X]_i = m(x_n)$ and $[k_{XX}]_{nn'} = k(x_n, x_{n'})$, $n, n' \in 1 \ldots N$.

We consider the regression problem, where noisy data y_n given some x_n is available, but the data generation process is unknown. In such cases a GP might be a good prior choice for a Bayesian regression approach. By conditioning the GP on the data $\mathcal{D} = \{x_n, y_n\}_{n=1}^N$, we can obtain a predictive posterior $g \mid \mathcal{D} \sim \mathcal{GP}(m,k)$ for any new location $x \in \mathcal{X}$:

$$\begin{aligned} m_p(x) &= m(x) + k_X(x)^\top (k_{XX} + \sigma^2 I)^{-1}(Y - m_X) \\ k_p(x,x') &= k(x,x') - k_X(x)^\top (k_{XX} + \sigma^2 I)^{-1} k_X(x), \end{aligned} \tag{2.8}$$

where $[k_X(x)]_n = k(x,x_n), [k_{XX}]_{nn'} = k(x_n,x_{n'}), Y_n = y_n, [m_X] = m(x_n)$ for $n,n' \in 1 \ldots N$. σ models the data noise, i.e., $y_n = f(x_n) + \epsilon, \epsilon \sim \mathcal{N}(0,\sigma^2)$, where f is the (unknown) data generating process.

2.4.1 Introduction to Bayesian Quadrature

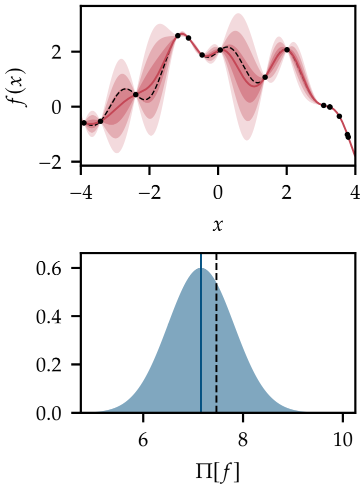


Figure 2.10: Illustration of Bayesian quadrature. (*Top*) Gaussian process regression (red) on evaluations of f (black dots). Gaussian approximation for the integral (mean indicated by the blue line, true value indicated by the dashed black line).

We are interested in approximating the integral

$$\Pi_\pi[f] = \int_{\mathcal{X}} f(x)\pi(x)dx.$$

Bayesian quadrature [22, 46, 150, 173] works by placing a prior $g \sim \mathcal{GP}(m,k)$ on f. Hence, we define

$$\Lambda := \int_{\mathcal{X}} g(x)\pi(x)dx,$$

where Λ is a random variable. GPs are closed under linear transformations (such as integrals) and, therefore, Λ is distributed according to a normal distribution, i.e.,

$$\Lambda \sim \mathcal{N}(\mu,\sigma)$$

with

$$\begin{aligned} \mu &= \mathbb{E}_g[\Lambda] = \int_{\mathcal{X}} m(x)\pi(x)dx \\ \sigma &= \mathbb{V}_g[\Lambda] = \int_{\mathcal{X}}\int_{\mathcal{X}} k(x,x')\pi(x)\pi(x')dxdx'. \end{aligned}$$

Conditioning the GP on observations $\mathcal{D} = \{x_n, f(x_n)\}_{n=0}^N$, we find

$$\Lambda \mid \mathcal{D} \sim \mathcal{N}(\mu_{\mathcal{D}}, \sigma_{\mathcal{D}}),$$

where (using Equation 2.8)

$$\begin{aligned} \mu_{\mathcal{D}} = \mathbb{E}_{g \mid \mathcal{D}}[\Lambda] &= \int_{\mathcal{X}} \left(m(x) + k_X(x)^\top k_{XX}^{-1}(f_X - m_X)\right)\pi(x)dx \\ &= \Pi_\pi[m] + \Pi_\pi[k_X]k_{XX}^{-1}(f_X - m_X), \end{aligned}$$

and

$$\begin{aligned} \sigma_{\mathcal{D}} = \mathbb{V}_{g \mid \mathcal{D}}[\Lambda] &= \int_{\mathcal{X}}\int_{\mathcal{X}} \left(k(x,x') - k_X(x)^\top k_{XX}^{-1} k_X(x')\right)\pi(x)\pi(x')dxdx' \\ &= \Pi\bar{\Pi}_\pi[k] + \Pi_\pi[k_X^\top]k_{XX}^{-1}\Pi_\pi[k_X]. \end{aligned}$$

Here $[f_X]_n = f(x_n)$, and $\Pi\bar{\Pi}_\pi[k] = \int_{\mathcal{X}}\int_{\mathcal{X}} k(x,x')\pi(x)\pi(x')dxdx'$. Above we assume that we can obtain noiseless samples from, i.e., $y_n = f(x_n)$, and $\sigma \longrightarrow 0$. Clearly, the expressions above can only be used if $\Pi_{\pi}[k_X]$, called the kernel mean embedding, and $\Pi\bar{\Pi}_{\pi}[k]$, called the initial error, are known in closed-form. This is only possible for some combinations of distribution π and covariance function k. For example, if $\mathcal{X} = \mathbb{R}^d$, π is a Gaussian and k is the RBF-kernel, then the expressions above can be computed analytically. Alternatively, if the kernel mean and variance with respect π is not available, one can compute the integral with respect to a distribution ρ where kernel mean and variance are known. This requires replacing the function f with $\tilde{f}(x) = f(x)\pi(x)/\rho(x)$, similar to importance sampling (Definition 2.3). So far we have assumed that we are given a data set $\mathcal{D}$, but BQ can also be extended to adaptively sample points [75, 152].

Summary

So with all these integration methods available, which is the right one to use? The truth is there is no "one-size-fits-all" method, but the method has to be chosen on a per-task basis. Classic methods based on polynomials encode the fact that f is continuous. Similarly, for BQ we can encode knowledge about the functional form of f via the choice of GP prior. Actually, a lot of classic algorithms can be encoded as a BQ method [93]. Hence, both of these methods are attractive if function evaluations are expensive as they encode additional information about f, but they only work well for lower dimensional problems. BQ comes with the additional advantage of providing uncertainty estimates for the computation allowing for a fully Bayesian approach, which is especially useful if evaluations are expensive and only limited data points are available. The advantage of MC methods is that their scaling is independent of dimension. Although that makes them often scale much worse than BQ and classic quadrature methods for low dimensional problems, MC methods are often the only choice for higher dimensional problems. Additionally, MC methods put few restrictions on f and π, i.e., f only needs to be integrable and MCMC methods even work if the distribution is only available in unnormalized form. The above points are only rough guidelines, but in Chapter 7 we discuss some of these methods in the context of experiments, and—to make choice even harder—propose a novel numerical integration method using neural networks.

Ordinary Differential Equations | 3.

The description of dynamical systems plays an important role throughout science. From describing the motions of planets or understanding chemical reactions to simulating fluids, all rely on the mathematical concept of solving differential equations. This chapter introduces a subclass of differential equations, i.e., ordinary differential equations. To find solutions to these equations, the use of numerical methods is often necessary, and we provide an overview of the most essential methods.

3.1 Ordinary Differential Equations

A first-order ordinary differential equation (ODE) describes an unknown function via its derivative.

Definition 3.1 Ordinary differential equation Let $f : \mathbb{R} \times \mathbb{R}^d \longrightarrow \mathbb{R}^d$ be a continuous function, then

$$x' = f(t, x) \tag{3.1}$$

is called an *ordinary differential equation*, where x' denotes the "time" derivative of x: $x' = \frac{dx}{dt}$. If an initial condition $x(t_0) = x_0$ is provided, the problem is called an *initial value problem (IVP)*.

The function f is also called the *vector field* of the ODE. If

$$x'(t) = f(t, x(t)) \tag{3.2}$$

then the function $x(t)$ is called a solution to the differential equation. Equation 3.1 can be extended to higher order problems, to describe an ordinary differential equation of order k

$$x^{(k)} = f(t, x, x', \dots, x^{(k-1)}).$$

To turn this into an IVP the following initial conditions are required $x(t_1) = x_0, x'(t_0) = x_1, \dots, x^{(k-1)}(t_0) = x_{k-1}$.

3.1.1 Existence of Solution

We just introduced the concept of a solution to an ODE, but the question remains whether such a solution exists. There are several theoretical statements about the existence (and uniqueness) of solutions. Here we state the well known Picard-Lindelöf theorem [164] which covers both existence and uniqueness of a solution, and is therefore more restrictive than theorems that just proof existence (e.g., the Peano existence theorem [80, Theorem 7.6]).

Theorem 3.1 Picard-Lindelöf Let $\mathcal{D} \subset \mathbb{R} \times \mathbb{R}^d$, be a closed set, containing the point (t_0, x_0) in its interior. Let $f : \mathcal{D} \longrightarrow \mathbb{R}^d$ be a continuous function which is Lipschitz-continuous in its second variable on $\mathcal{D}$, i.e., there exists an $L \geq 0$ such that for any $(t, x_1), (t, x_2) \in \mathcal{D}$

$$||f(t, x_1) - f(t, x_2)|| \leq L||x_1 - x_2||.$$

Then there exists some $\epsilon > 0$ such that the initial value problem

$$x' = f(t, x), \quad x(t_0) = x_0$$

has a unique solution on $[t_0 - \epsilon, t_0 + \epsilon]$.

For a proof see [80, Theorem 7.3]. An extension of this local result is available for continuously differentiable functions.

Theorem 3.2 Picard-Lindelöf - Extension Let f be continuously differentiable on an open set $\mathcal{E} \subseteq \mathbb{R} \times \mathbb{R}^d$, then for all $(t_0, x_0) \in \mathcal{E}$, there exists a unique solution to the IVP that continues to the boundary of $\mathcal{E}$ in both directions.

3.2 Numerical Approximation of ODE Solutions

ODEs are often hard or impossible to solve analytically. But there exist certain classes of functions and some analytical tricks to find analytical solutions (for an overview of these methods see, e.g., [23]). In this thesis, though, the focus is not on finding analytical solutions to ODEs but on approximating the solutions numerically.

3.2.1 Fixed Step Runge-Kutta Methods

For now, we consider scalar first order ordinary differential equations. The solution to the ODE in Equation 3.1 on the interval at time $t + h$, where $x(t)$ is known, can be written as

$$x(t + h) = x(t) + \int_t^{t+h} f(s, x(s))ds. \tag{3.3}$$

Doing a zeroth-order Taylor series expansion of $f(s, x(s))$ around t, gives us

$$f(s, x(s)) \approx f(t, x(t)).$$

Plugging this expression into the integral in Equation 3.3, we arrive at what is known as *(explicit) Euler method*.

Definition 3.2 Euler method We consider an initial value problem with $x(t_0) = x_0$. Then, given a step size $h > 0$ and $t_n = t_0 + nh$ where $n = 0, 1, 2, \ldots$ the solution to the ODE can be approximated via

$$x_{n+1} = x_n + hf(t_n, x_n). \tag{3.4}$$

Alternatively, doing a zeroth-order expansion of $f(s, x(s))$ around $t + h$, leads to the *implicit Euler method*.

Definition 3.3 **Implicit Euler method** The solution to an ODE can be approximated via the implicit Euler method, i.e.,

$$x_{n+1} = x_n + hf(t_{n+1}, x_{n+1}). \tag{3.5}$$

We note that Equation 3.5 requires solving a non-linear equation. Generally, methods where the update step not only contains the current state of the system but also future states, are known as *implicit*. If implicit solvers are significantly more efficient than explicit solvers on an ODE system, this system is known as *stiff*.

Instead of approximating the integral with a constant, one can use more advanced integration schemes. In the previous chapter we introduced the Midpoint rule 2.1 for numerical integration. Applying this rule to Equation 3.3 we arrive at

$$x(t + h) \approx x(t) + hf(t + \frac{h}{2}, x(t + \frac{h}{2})).$$

The only term left to compute is $x(t + \frac{h}{2})$, which again is computed with another Euler step. We arrive at the *(explicit) Midpoint method* for ODEs.

Definition 3.4 **Midpoint method** The solution to an ODE can be approximated via the Midpoint method, which is given by

$$\begin{aligned} k_1 &= f(t_n, x_n), \\ k_2 &= f(t_n + \frac{h}{2}, x_n + \frac{h}{2}k_1), \\ x_{n+1} &= x_n + hk_2. \end{aligned}$$

To describe the convergence of numerical ODE integration methods, we introduce the *order* of a Runge-Kutta method.

Definition 3.5 A Runge-Kutta method is of order p if for sufficiently smooth solutions x

$$\|x(t_0 + h) - x_1\| \leq Kh^{p+1}$$

for some problem specific constant K.

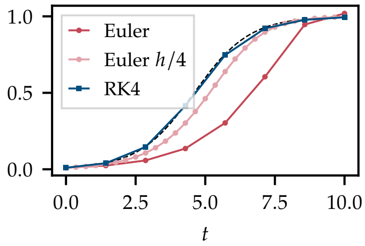


Figure 3.1: Numerical solutions (in red and blue) to the logistic ODE (details in Appendix A.1.1).

Additionally, for methods of order p the global truncation error (the error accumulated over several steps $\|x(t_n) - x_n\|$) is $\mathcal{O}(h^p)$. The Euler method, following from the Taylor series expansion, is a first order method. The midpoint method is a second order method (see [80, Ch. II.3] for exact statements and proofs). One can use even higher order quadrature methods to derive higher order (ODE) integration methods. Runge [185] derives a 4th order scheme based on Simpson's rule (Definition 2.2). To illustrate the better convergence of higher order methods, we compute solutions to an ODE using the Euler method and a 4th order Runge-Kutta method (RK4) (see Figure 3.1).

In general all these schemes can be summarized quite conveniently using the following notation for a general Runge-Kutta scheme.

Definition 3.6 Explicit Runge-Kutta method An *s-stage explicit Runge-Kutta method* is given by

$$x_{n+1} = x_n + h \sum_{i=1}^{s} b_i k_i$$

where

$$\begin{aligned}
k_1 &= f(t_n, x_n),\\
k_2 &= f(t_n + c_2 h, y_n + (a_{21}k_1)h)\\
k_3 &= f(t_n + c_3 h, y_n + (a_{31}k_2 + a_{32}k_2)h),\\
&\vdots\\
k_s &= f(t_n + c_s h, y_n + (a_{s1}k_1 + a_{s2}k_2 + \dots a_{s,s-1}k_{s-1})h).
\end{aligned}$$

The coefficients are given by a *Butcher tableau*

$$\begin{array}{c|cccc}
0 & & & & \\
c_2 & a_{21} & & & \\
\vdots & \vdots & \ddots & & \\
c_s & a_{s1} & \dots & a_{s,s-1} & \\
\hline
 & b_1 & \dots & b_{s-1} & b_s
\end{array}.$$

We can clearly see that the methods (Euler and Midpoint) we developed above fit in the definition of an explicit Runge-Kutta method. By Taylor expansion and comparing coefficients, certain rules for finding Butcher tableaux can be established and many Runge-Kutta schemes have been proposed over the years. The Runge-Kutta coefficients for a method of stage s and order p are not necessarily unique, numerical implementation and stability can play a role in the choice. A prominent example were at least two common implementations exist are 4-stage 4th order Runge-Kutta methods:

$$\begin{array}{c|cccc}
0 & & & & \\
1/2 & 1/2 & & & \\
1/2 & 0 & 1/2 & & \\
1 & 0 & 0 & 1 & \\
\hline
 & 1/6 & 1/3 & 1/3 & 1/6
\end{array}.$$

An alternative implementation is the 3/8-rule which leads to smaller computational error but is computationally slightly more expensive

$$\begin{array}{c|cccc}
0 & & & & \\
1/3 & 1/3 & & & \\
2/3 & -1/3 & 1 & & \\
1 & 1 & -1 & 1 & \\
\hline
 & 1/8 & 3/8 & 3/8 & 1/8
\end{array}.$$

Only certain combinations of s and p exist, e.g., to represent a method of order p it requires at least $s \geq p$ stages but $p = 5$, for example, requires $s \geq 6$ [80, Ch. II.4, 5].

Solving Higher Order ODEs

The algorithms derived above only apply to first order ODEs. They can be extended to second order ODEs (and respectively any higher order) by writing $x_2 = x'$ and $x_1 = x$. Then we can write the one-dimensional second order system

$$\begin{aligned} x_1' &= x_2 \\ x_2' &= f(t, x_1, x_2) \end{aligned}$$

as a first order two-dimensional system. To this we can directly apply any of the solvers derived above. For some of those systems even more specialized solvers exist [80, Ch. II.14].

Symplectic Runge-Kutta Methods

A special ODE system are the Hamiltonian equations of motion (Equation 2.7). An important property of such systems is that the corresponding flow is volume preserving. Hence, numerical solvers designed specifically for Hamiltonian systems are designed to maintain volume conservation for each step. Solvers which fulfill this property are called *symplectic*. Two well-known and rather simple symplectic solvers are the symplectic Euler method and the Leapfrog method.

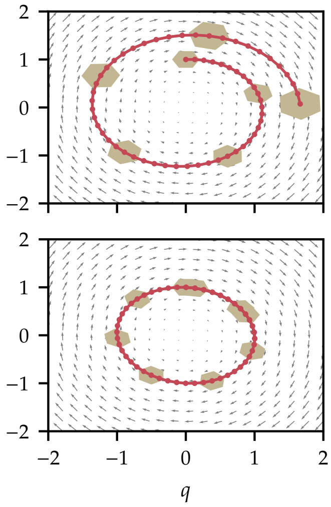


Figure 3.2: Numerical solutions (in red) to the harmonic oscillator (details in Appendix A.1.2) using the explicit Euler method (*top*) and the symplectic Euler method (*bottom*). A set of initial conditions which describe a volume (brown area) are used for each solver. The area is evolved with the same step size as the red trajectory, but only plotted for every 7th step.

Definition 3.7 Symplectic Euler method The solution to the Hamiltonian equations of motion can be approximated via the implicit Euler method, i.e.,

$$\begin{aligned} p_{n+1} &= p_n - h\partial_q H(q_n, p_{n+1}) \\ q_{n+1} &= q_n + h\partial_p H(q_n, p_{n+1}) \end{aligned}$$

or

$$\begin{aligned} q_{n+1} &= q_n + h\partial_p H(q_{n+1}, p_n) \\ p_{n+1} &= p_n - h\partial_q H(q_{n+1}, p_n). \end{aligned}$$

The method is an implicit method. However, for separable Hamiltonians ($H(q, p) = V(q) + T(p)$) the symplectic Euler method becomes explicit as $\partial_q H = \partial_q V$ and $\partial_p H = \partial_p T$. We will not proof that these methods indeed fulfill the symplectic property, but Figure 3.2 illustrates the volume preservation (we refer to [81, Ch. VI] for details).

3.2.2 Adaptive Step Size Solvers

What, if the difficulty of the problem varies along the integration path, i.e., different regions require different step sizes to reach similar numerical accuracy? So far we are forced to choose the smallest necessary step size along the entire integration path to reach our desired numerical accuracy. Adaptive step size Runge-Kutta methods overcome this issue

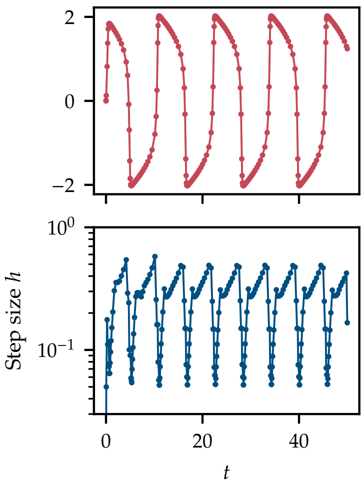


Figure 3.3: (*Top*) Numerical solution to the Van der Pol oscillator (for details Appendix A.1.3) using the Dopri5(4) method. (*Bottom*) Step size chosen by the adaptive step size solver.

by changing the step size along the integration path. These methods work by comparing the output of a higher and a lower order method. If the difference is above or below a certain tolerance level, the step size is decreased or increased [80, Ch. II.4]. As the choice of coefficients of a Runge-Kutta method is not unique, it is possible to find Runge-Kutta schemes where the method of order p and the method of order $p+1$ share the computation of the k_i. An example is the Felberg2(1) method [55], where the first row provides the coefficients for a second order method and the second row the coefficients for a first order method, as shown in the following Butter tableau

$$\begin{array}{c|ccc} 0 & & & \\ 1/2 & 1/2 & & \\ 1 & 1/256 & 255/256 & \\ \hline & 1/512 & 255/256 & 1/512 \\ & 1/256 & 255/256 & 0 \end{array}.$$

A well-known method is the Dopri5(4) scheme [168], using a method of order 5 to compute the steps and a method of order 4 to compute an error estimate (see Figure 3.3).

Illustrative Example

To illustrate the effect of the numerical error different Runge-Kutta methods make, we consider an example based on the Arenstorf-orbits [4] (the example is adapted from [80, Ch. II]). One considers the orbit of a spaceship around the earth and the moon. We assume the mass of the spaceship to be negligible relative to the other bodies and that the moon orbits the earth on a plane.

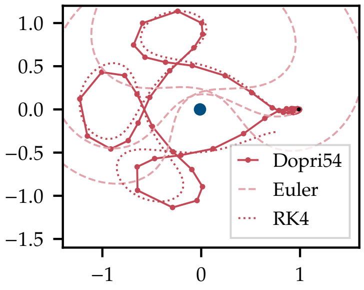


Figure 3.4: Solutions to the Arenstorf-orbit differential equations. Example adapted from [80] (details are found in Appendix A.1.4).

To show the behavior of different solvers, we consider the Euler method with $h = T/24000$, RK4 with $h = T/6000$ and Dopri5 with $tol = 10^{-3}$. Figure 3.4 shows the resulting trajectories, and one can observe that the Euler method, despite the small step size, results in an inaccurate approximation of the solution. This is due to the error accumulation over time, the approximation gets worse and worse. By using higher order solvers (here the RK4 method) this issues becomes already less pronounced, but we obtain the best result by using adaptive step size methods.

Differential equations like the Arenstorf-orbits can be used to describe swing-bys used in aerospace engineering to change the direction and speed of, e.g., a spaceship by flying past a larger object such as the moon. The Apollo 13 mission used such a maneuver* to rescue its astronauts, after a failure in an oxygen tank.

* possibly based on rescue orbits mapped out by Arenstorf

Neural Networks 4.

The goal of machine learning is to make predictions from data and prior information. One way to formulate this is as empirical risk minimization problem which is expressed mathematically as minimizing a cost function given a data set $\mathcal{D} = \{x_n, y_n\}_{n=1}^N$, by adjusting the parameters θ of a function f_θ. In the context of deep learning the function f_θ is a neural network. At the basis of any complex neural network architecture are fundamental operations which are often by themselves quite simple. Section 4.1 introduces the basic building blocks of a neural network, and Section 4.2 then explains how to fit a neural network to data. As neural networks are data driven models, insufficient data can lead to bad results. To detect inaccurate predictions good uncertainty quantification is necessary. Section 4.3 introduces uncertainty quantification via the Laplace approximation, and provides a short general overview of Bayesian neural networks. Finally, Section 4.4 concludes with an overview of how modern architectures use structure to improve performance. For an in-depth introduction to deep learning and neural network architectures we refer to Goodfellow et al. [66].

4.1 Neural Network Components

In this thesis we consider regression and classification tasks. Regression tasks aim to estimate the relationships between two (or more) variables based on data, where one variable is assumed to depend on the other. In classification, the goal is to estimate the category of a given observation, given previous data. Neural networks are a good choice for such tasks when the data set under consideration is high-dimensional and large (e.g. for image classification). Their good performance is achieved by their special structure and a way to find appropriate parameters. In this section we will focus on the structure of neural networks and in the next section we will discuss how to train neural networks.

Neural networks are functions $f_\theta : \mathcal{X} \subseteq \mathbb{R}^{d_x} \rightarrow \mathbb{R}^{d_y}$, where d_x is dimensionality of some input data $x_n \in \mathcal{X}$. We consider the case where $\mathcal{X} = \mathbb{R}^{d_x}$, as this holds true for most architectures and allows for a less cluttered notation. Here $\theta \in \Theta \subseteq \mathbb{R}^p$ denotes the *weights* of the neural network, which are essentially parameters of the function f_θ. Neural networks are a composition of often relatively simple building blocks $f^i_{\theta_i} : \mathbb{R}^{d_{i-1}} \rightarrow \mathbb{R}^{d_i}, i \in \{1, \dots, L\}, d_0 = d_x, d_L = d_y$, commonly called *layers*

$$f_\theta = f^L_{\theta_L} \circ \cdots \circ f^1_{\theta_1}. \tag{4.1}$$

Here, L is the number of layers, and θ_i the parameters of layer i. Each neural network layer is composed of an affine transformation $z^i_{\theta_i} : \mathbb{R}^{d_{i-1}} \rightarrow \mathbb{R}^{d_i}$ applying the weights of the neural network and an *activation function* $\sigma^i : \mathbb{R}^{d_i} \rightarrow \mathbb{R}^{d_i}$.

$$f^i_{\theta_i} = \sigma^i \circ z^i_{\theta_i}. \tag{4.2}$$

If we choose $\sigma^i = I_{d_i}$ for $i = 1, \dots, L$, then the neural network f_θ corresponds to an affine transformation. By choosing σ^i to be a non-linear function, f_θ can be used to model more complex, non-affine functions. Indeed, with the right choice of activation function and sufficiently many parameters, the neural network f_θ can theoretically approximate any continuous function $g : \mathcal{D} \subseteq \mathbb{R}^{d_x} \longrightarrow \mathbb{R}^{d_y}$, $\mathcal{D}$ compact, arbitrarily well [96, 97].

We note that we include the identity transformation in our definition of σ^i to allow the same notation *input* ($i = 1$), *hidden* ($i = 2, \dots, L-1$) and *output* ($i = L$) layers. For example for the output layer, $\sigma^L = I_{d_y}$ is a common choice for regression tasks.

The description of a neural network introduced in Equation 4.1 and Equation 4.2 is not sufficient to capture all modern architectures. Yet, those architectures still follow the same overall structure.

4.1.1 Linear Layers

A simple structure for the layers are so-called *linear layers* or *fully connected layers*.

Definition 4.1 Linear layer We can write a general linear layer as a function $z^i_{\theta_i} : \mathbb{R}^{d_{i-1}} \longrightarrow \mathbb{R}^{d_i}$ as

$$z^i_{\theta_i}(x^{i-1}) = A^i x^{i-1} + b^i \tag{4.3}$$

where $\theta_i = \{A^i, b^i\}$, $A^i \in \mathbb{R}^{d_i \times d_{i-1}}$ is a matrix and $b^i \in \mathbb{R}^{d_i}$.

4.1.2 Convolutional Layers

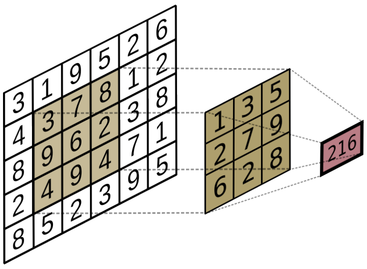


Figure 4.1: Sketch of a convolutional layer. A kernel (brown) slides over an image (white). The weighted sum of the pixels covered by the kernel (light brown) is the output of the layer (red).

Convolutional layers [59, 119] are commonly used for tasks dealing with image data. The structure of convolutional layers can be motivated by the assumption that the network output should remain invariant under small data shifts, i.e., we can still recognize the image even if it is moved a few pixels to the left or right. Convolutions consist of a *kernel* A, a $k \times k$ matrix sliding over an image. At each pixel a sum over the pixels covered by the kernel is performed, weighted with the entries of A, (this procedure is illustrated in Figure 4.1). This description is oversimplified and does not introduce concepts like padding, stride and channels (see Goodfellow et al. [66, Ch. 9] for details).

4.1.3 Activation Functions

Both, linear layers and convolutional layers, can only model affine transformations. To model functions other than affine transformations, e.g., the XOR function, non-linear features are necessary. Common choices for these *non-linearities* (also called *activation functions*) include rectified linear units (RELU) [58], tanh, and the softmax function.

Definition 4.2 **RELU** A *RELU activation* is a function $\sigma : \mathbb{R} \to \mathbb{R}_0^+$:

$$\sigma(x) = \max(0, x).$$

To use the RELU activation for higher dimensional data, the activation is applied element-wise. RELU activations often achieve better training results than tanh (both faster training and a lower loss), and are nowadays the most frequently used activation function [48]. Tanh activations can suffer from the vanishing gradient problem [15], due to very small gradients in the tails. Nonetheless, tanh activations are still commonly used for certain neural network architectures. Figure 4.2 shows a visualization of the tanh and the RELU activation, and their derivatives.

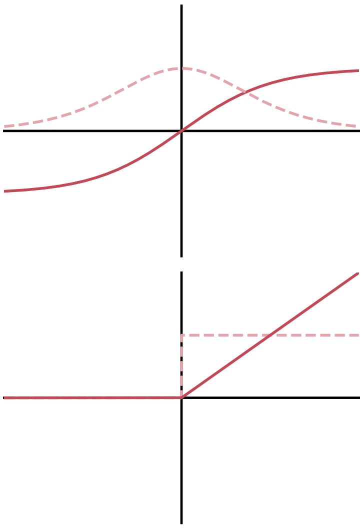


Figure 4.2: Tanh (*top*) and RELU (*bottom*) activation functions and their respective derivatives.

The softmax function is commonly employed in the last layer of classification problems.

Definition 4.3 **Softmax** The *softmax activation* is a function $\sigma : \mathbb{R}^d \to \mathbb{R}^d$

$$\sigma(x)_k = \frac{e^{x_k}}{\sum_{l=1}^{d} e^{x_l}}.$$

The list of activation functions presented here is only limited to the most used ones. Modern software libraries like `PyTorch` [159] implement a long list of activations functions, including several extensions of RELU.

With the presented building blocks it is possible to describe simple neural networks. However, many modern architectures have a more complex structure and include additional operations like maximum and average pooling, dropout or batch normalization, which are outside the scope of this thesis (see [66, Ch. 9, Ch. 11] for more details).

4.1.4 Basic Neural Network Architectures

One objective in deep learning is to construct network architectures f_θ, which are not only *theoretically* able to represent a function of interest, but where it is practically possible to find a set of weights that works well for a given task. Below we outline some basic neural network architectures, consisting of the basic building blocks described previously. Even those simple architectures contain structures which make finding a good set of weights easier.

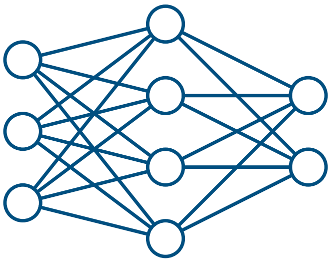


Figure 4.3: Sketch of a fully connected linear neural network.

Fully Connected Linear Neural Network This simple architecture consists of all linear layers with some activation function (see Figure 4.3). For the output layer, a common choice for the activation function is either the identity function or the softmax function.

Convolutional Neural Networks Due to the special structure of convolutional layers, these architectures are often used for image classification tasks. Here, we present a very simple form of a convolutional neural network. Layers $f^1 \dots f^i$ are convolutional layers, and make up the biggest part of the network. The last few layers $f^i \dots f^L$ are linear layers with the output activation function usually being the softmax function for classification tasks. The architecture presented here ignores some additional tricks used in modern architectures like down-sampling layers, max-pooling, dropout, batch-norm [117, 191]. Commonly RELU activation functions are used for this type of architecture.

4.2 Neural Network Training

The previous sections provided the ingredients to build a neural network, but a key ingredient, finding suitable weights, is still missing. This section establishes how to locate a good set of weights via minimizing a cost function. The optimization problem is non-trivial, and we discuss how deep learning tackles this problem.

4.2.1 Empirical Risk Minimization

We consider the task where we have a data set $\mathcal{D} = \{x_n, y_n\}_{n=1}^N, x_n \in \mathcal{X} \subseteq \mathbb{R}^{d_x}, y_n \in \mathcal{Y} \subseteq \mathbb{R}^{d_y}$, with some data generating process which can be described through a probability distribution such that $x_n, y_n \sim p(x, y)$. We want to evaluate the performance of a model $f_\theta : \mathcal{X} \subseteq \mathbb{R}^{d_x} \longrightarrow \mathbb{R}^{d_y}$, and therefore consider the loss function $c : \mathcal{Y} \times \mathcal{Y} \subseteq \mathbb{R}^{d_y \times d_y} \longrightarrow \mathbb{R}_0^+$. The expected loss under the data distribution is therefore given by:

$$\mathbb{E}_{p(x,y)}\left[c\right] = \iint c(y, f_\theta(x))p(x, y)dxdy. \tag{4.4}$$

The goal is then to find a set of parameters θ^* which minimizes the expected cost, i.e.,

$$\theta^* = \arg\min_\theta \mathbb{E}_{p(x,y)}\left[c\right].$$

The equations above cannot be evaluated in practice, as the true data generating process $p(x, y)$ is unknown. But one can approximate the expression in Equation 4.4 using the data set $\mathcal{D}$

$$l(\theta) := \mathbb{E}_{\mathcal{D}}\left[c\right] = \sum_{n=1}^N c\left(y_n, f_\theta(x_n)\right).$$

Hence, the objective for finding the network weights changes to

$$\theta^* = \arg\min_\theta l(\theta).$$

This task is called *empirical risk minimization*.

4.2.2 Optimization

Finding the minimum of the empirical risk minimization task is, in general, hard. Given that optimization problems are omnipresent in classic machine learning and many other fields, a plethora of methods on how to solve these problems exist, especially in the case of convex $l(\theta)$ [143]. But in the context of deep learning, where θ are the parameters of a neural network, classic optimization methods (like the ones discussed in,

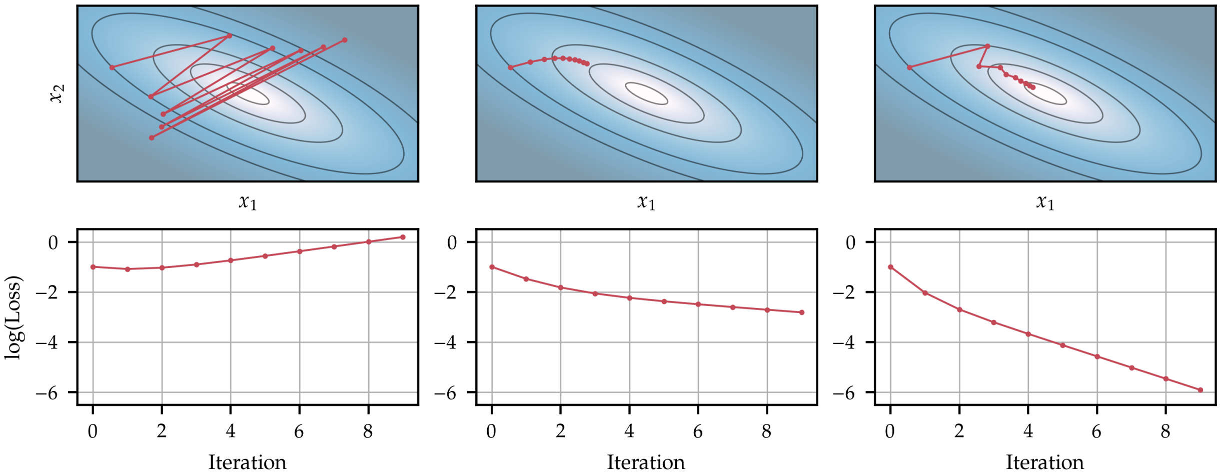


Figure 4.4: Effect of the learning rate on a quadratic optimization task using gradient descent. Top row shows the first 10 steps of the optimizer in the loss landscape, bottom row shows the corresponding loss. (*Left*) Optimization using a too large learning rate, leading to divergent behavior. (*Center*) The learning rate is chosen too small, so convergence is much slower than when training with a more appropriate learning rate (*right*).

e.g., [143]) usually do not work out of the box. The optimization landscape for deep learning models is non-convex [120] and high-dimensional due to the large parameter space. Therefore, classic methods (e.g., Newtons method), do not work.

Dep learning models can be incredibly large (e.g., $\sim 1e6$ parameters for a classic image classification network [86], $3.5e8$ for a common language model [45], $3.5e9$ for the decoder of a high-quality image generation network [172]), which limits the choice of optimization procedure. Hence, finding the global minima is not feasible due to the large parameter space, but it is still essential to find a "good" local minimum, where "good" refers to the model achieving a low loss and good generalization.

The optimization method commonly used in deep learning is based on *gradient descent*.

Definition 4.4 Gradient descent We assume a differentiable cost function c and model f_θ (note that not even this is true in practice - the derivative of RELU activation is only defined on $\mathbb{R}\backslash\{0\}$). Then, given a set of parameters θ_0, these are updated based on the following rule

$$\theta_{t+1} = \theta_t - \eta \nabla_{\theta_t} l(\theta_t) \tag{4.5}$$

where

$$\nabla_\theta l(\theta) = \sum_{n=1}^{N} \nabla_\theta c\left(y_n, f_\theta(x_n)\right).$$

Here η is the *learning rate*, i.e., a parameter that determines how large steps taken by the algorithm are.

If the learning rate is chosen too large, it can lead to divergent behavior [66, Ch. 7], if chosen too small, convergence can be extremely slow (illustrated in Figure 4.4). Choosing the learning rate can be tedious and classic convex optimization algorithms use more refined schemes like line-search and trust-region methods [143]. We observe that Equation 4.5 contains the gradient of the loss function. This gradient can be computed through an algorithm known as backpropagation implemented in modern software libraries like `PyTorch` [159]. Backpropagation essentially works via the repeated application of the chain rule.

The update step in Equation 4.5 requires the computation of the average gradient over the entire data set. But, given the size of deep learning data sets, this is often infeasible in practice. So instead of computing the gradient over the entire data set, a subset of the data set (the *batch*) is used to compute an approximation. The resulting algorithm is called *stochastic gradient descent* (SGD) [179] (we note that in the literature the term SGD is sometimes only used for a batch size of $B = 1$, but deep learning commonly refers to batch gradient descent as SGD).

Definition 4.5 **SGD** We consider a subset of the data set $\mathcal{B} \subseteq \mathcal{D}$ where $B = |\mathcal{B}| \leq N$ and use this information for the parameter update

$$\theta_{t+1} = \theta_t - \eta \nabla_{\theta_t} \hat{l}(\theta_t)$$

where

$$\nabla_\theta \hat{l}(\theta) = \sum_{n=1}^{B} \nabla_\theta c\left(y_n, f_\theta(x_n)\right).$$

The batch is resampled from the data set after each step.

SGD might seem like a crude approximation to the original objective, but it works well in the context of deep learning, possibly not despite but due to the approximations made. We will not discuss why SGD and its variants find good global minima (as a priori this sounds rather perplexing), but several papers have made progress on understanding this behavior [82, 113, 192]. Finding optimizers that work well is an area of active research in modern deep learning, and several extensions of SGD exist, e.g., SGD with Momentum [184], Adam [110], or Adagrad [49] to name a few commonly used choices.

4.2.3 Loss Functions

We now specify common loss function used in machine learning. These loss functions provide the objective we want to optimize.

Definition 4.6 The *mean squared error* (MSE) $l_{\text{MSE}} : \mathbb{R}^p \to \mathbb{R}^+$ describes the average squared distance between model outputs and the data set

$$l_{\text{MSE}}(\theta) = \frac{1}{B} \sum_{n=1}^{B} ||y_n - f_\theta(x_n)||_2^2.$$

The MSE loss is often used for regression tasks.

Maximum Likelihood Estimation

Optimizing the MSE loss can also be interpreted as *maximum (log)likelihood estimation* (MLE). $p(y \mid x, \theta)$ describes the probability distribution of the model outputting y given some inputs x and parameters θ, also known as the *likelihood*. We can then try to optimize the expectation of the likelihood under the data generating distribution $p(x, y)$

$$\theta^*_{\text{MLE}} = \arg\max_\theta \mathbb{E}_{p(x,y)}\left[p(y \mid x, \theta)\right].$$

Again, the expectation cannot be computed and has to be approximated using data. Considering the negative log-likelihood given the data set, we obtain

$$l_{\text{MLE}}(\theta) = -\log p(Y \mid X, \theta) = -\sum_{n=1}^{N} \log p(y_n \mid f_\theta(x_n)),$$

where $X = \{x_n\}_{n=1}^N$ and $Y = \{y_n\}_{n=1}^N$. We assume $p(y_n|f_\theta(x_n)) = \mathcal{N}(y_n \mid f_\theta(x_n), \sigma^2 I)$. Here σ is the data set noise (assumed to be constant across all dimensions). Plugging the expression for $p(y_n|f_\theta(x_n))$ into the expression for the loss,

$$l_{\text{MLE}}(\theta) = -N \log \sigma - \frac{N}{2}\log(2\pi) - \frac{1}{\sigma^2}\sum_{n=1}^{N} ||y_n - f_\theta(x_n)||_2^2.$$

Since the constant factors do not affect the minima, we find that maximum log-likelihood estimation is indeed equivalent to minimizing the mean squared error: $\theta^* = \arg\min_\theta l_{\text{MLE}}(\theta) = \arg\min_\theta l_{\text{MSE}}(\theta)$.

Cross Entropy Loss We consider the task of assigning each data point x_n to a class $c_n \in \{1, \ldots, C\}$. We assume we have a model f_θ, which outputs a probability distribution over all possible classes $p(c \mid x, \theta)$. This is commonly achieved by using the softmax layer as the last layer. Therefore, the model f_θ outputs a C-dimensional vector encoding the probability for each class. Hence, the objective we want to optimize is

$$\theta^*_{\text{MLE}} = \arg\max_\theta \mathbb{E}_{p(c,x)}\left[p(c \mid x, \theta)\right].$$

Again, we approximate this expression using samples from the data set. The true class distribution is given by $p(c \mid x_n) = \delta_{c,c_n}$ for $c \in \{1 \ldots C\}$ and c_n is the true class label. We assume a constant prior across all classes.

Definition 4.7 The *cross entropy loss* $l_{\text{Cross entropy}} : \mathbb{R}^p \longrightarrow \mathbb{R}^+$ is given by

$$l_{\text{Cross entropy}}(\theta) = \frac{1}{N}\sum_{n=1}^{N}\sum_{c=1}^{C} \log\left([f_\theta(x_n)]_c\right)\delta_{c,c_n}.$$

Regularization

Neural networks are capable of modelling a large class of functions. Hence, a model with sufficiently many parameters might be able to achieve zero loss on a given data set but then perform badly on previously unseen data. This issue, i.e., the model not being able to capture the structure of the data, is known as *overfitting*. To detect overfitting a common setup

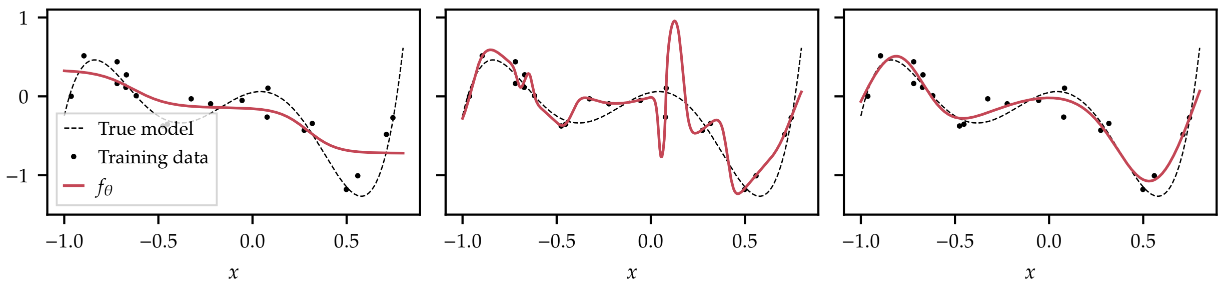


Figure 4.5: Effect of under- and over-parametrizing a model. (*Left*) The neural network is under-parametrized and hence underfitting the data. (*Center*) The model is overfitting the data, as it is too large. However, overfitting can be avoided using a regularizer (here weight decay) (*right*).

is to train the model on a subset of the available data set, and then test the model on the held out data. If a large drop in performance occurs on the test data, this is likely due to overfitting. In contrast, if the model is not able to achieve low loss on the training data, i.e., it is underfitting, the network's modelling capacity needs to be increased, e.g., by using a larger network. Figure 4.5 illustrates the effect of over- and underfitting.

One way to improve the generalization of a model is to penalize unfavorable properties, for example by using a regularizer $r : \mathbb{R}^p \to \mathbb{R}$ on the model. This regularizer is then added to the loss

$$l_{tot} = l(\theta) + \lambda r(\theta). \tag{4.6}$$

λ is a parameter which determines the strength of the regularization. If λ is chosen to large it might limit the expressiveness of the model. One possible choice of regularizer is the Euclidean norm of the weights known as *weight decay* $r(\theta) = ||\theta||_2^2$. This method works by penalizing too large weights which avoids rapid changes in the function values (illustrated by Figure 4.5).

4.3 Bayesian Deep Learning

Neural networks achieve remarkable results on a range of tasks. However, if simply not enough data is available during training, model predictions might be arbitrarily bad. Additionally, outside the data domain, the model's extrapolation is dominated by the functional form of its activation function [214], potentially leading to bad predictions. On classification tasks this even leads to an overconfidence of the model outside the data domain [90]. Therefore, it is absolutely crucial to get uncertainty estimates for the model outputs, and we argue for a Bayesian approach.

4.3.1 Bayesian Neural Networks

To take a Bayesian perspective, we would like to obtain a posterior distribution over model parameters θ given some data and a prior. *Bayes rule* can be used to compute a posterior, provided some prior knowledge about the distribution of model parameters $p(\theta)$, and the likelihood given some data.

Definition 4.8 Bayes theorem Consider the prior $p(B)$ and the marginal probability $p(A) = \int p(A, B)dB$ on B, A respectively, and some likelihood $p(A \mid B)$. Then the distribution over B given A, the *posterior*, is given by

$$p(B \mid A) = \frac{p(A \mid B)p(B)}{p(A)}.$$

This means we would like to compute

$$p(\theta \mid \mathcal{D}) = \frac{p(Y \mid X, \theta)p(\theta)}{\int p(Y \mid X, \theta)p(\theta)d\theta}. \tag{4.7}$$

In the previous section we have already identified that common loss functions are likelihoods $p(Y \mid \theta, X)$ up to a constant.

Usually it is not possible to compute the normalization constant in Bayes rule Equation 4.7, and thus BNNs require some approximation strategy. One option is to use MC-methods (introduced in Section 2.3) for approximating $p(\theta|\mathcal{D})$, but this comes with a high computational cost [100]. Other common approximate inference schemes are variational inference [18, 72, 94], and ensemble methods [118].

4.3.2 The Laplace Approximation

A way to equip neural networks with lightweight, post-training uncertainty is the *Laplace approximation* for neural networks [133]. We first establish a connection between the regularized loss and the posterior, and then show how to use the Laplace approximation to compute the posterior.

Loss as Maximum-a-posteriori Estimate

We assume a prior $p(\theta)$ on the weights, (often assumed to be an isotropic Gaussian). Then, the posterior of the parameters given data can be computed via

$$p(\theta \mid X, Y) = \frac{p(Y \mid \theta, X)p(\theta)}{p(X, Y)}.$$

As discussed, $p(X, Y)$ is usually unknown, and computing it via Monte Carlo sampling is computationally infeasible.

Given the expression for the posterior of the weights, the *maximum-a-posteriori* estimate, i.e., the mode of the posterior, is defined via

$$\theta_{\text{MAP}} = \arg\max_{\theta} p(\theta \mid X, Y).$$

We then apply the logarithm to this expression and drop terms independent of θ

$$\theta_{\text{MAP}} = \arg\max_{\theta} \left(\log p(Y \mid \theta, X) + \log p(\theta)\right).$$

By identifying the regularizer as prior $\log p(\theta) \propto \lambda||\theta||^2$ - which corresponds to an isotropic Gaussian for weight decay - we can interpret the minimum of l_{tot} in Equation 4.6 as the MAP-estimate.

Applying the Laplace Approximation

We now consider the task of computing a posterior distribution, given an isotropic Gaussian prior on the weights $p(\theta) = \mathcal{N}(\theta \,|\, 0, \sigma_0^{-2} I_p)$. Given this interpretation of the loss, the challenge of computing the normalization constant of the posterior remains. But the mode of the posterior—the MAP—can be computed by doing gradient descent on the loss. To obtain a normalized expression for the posterior, we approximate the posterior with a Gaussian using the Laplace approximation (see Section 2.2), i.e.,

$$p(\theta \mid \mathcal{D}) \approx \mathcal{N}(\theta \mid \theta_{\text{MAP}}, \Sigma) =: q(\theta).$$

The variance is given by the inverse Hessian of the negative log posterior

$$\Sigma = (-\nabla^2_\theta \log p(\theta \mid \mathcal{D})|_{\theta=\theta_{\text{MAP}}})^{-1} = (\nabla^2_\theta l_{\text{MLE}}(\theta)|_{\theta=\theta_{\text{MAP}}} + \sigma_0^2 I_p)^{-1}.$$

We note that the posterior expectation of the weights under the true posterior might actually differ from the MAP estimate, hence this can only been seen as an approximation to the fully-Bayesian approach. But the MAP estimate is currently the quantity of interest in deep learning and much effort goes towards tuning this quantity.

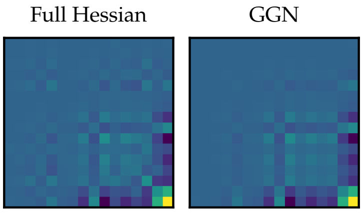


Figure 4.6: Full Hessian (*left*) and GGN approximation (*right*) of a small neural network.

General Gauss Newton Approximation Computing the full Hessian Σ is computationally expensive. Additionally, the Laplace approximation requires a positive semi-definite Hessian, which is not always guaranteed for neural networks. For large network, some weights might play little to no role in obtaining a low loss. These weights, due to the lack of gradient information, might not reach a minimum in the loss landscape, leading to negative eigenvalues of the Hessian. Hence, it is common to consider positive semi-definite approximations of the Hessian, one of them is the general Gauss Newton approximation (GGN; [188]) (see Figure 4.6).

Since the Hessian of the prior is trivial to compute, we only consider an approximation of the Hessian of the likelihood

$$\begin{aligned}
H &= \sum_{n=1}^{N} \nabla^2_\theta \log p(y_n \mid f_\theta(x_n))|_{\theta=\theta_{\text{MAP}}} \\
&= \sum_{n=1}^{N} J(x_n)^\top \nabla^2_f \log p(y_n \mid f)|_{f=f_{\theta_{\text{MAP}}}(x_n)} J(x_n) \\
&\quad + \sum_{n=1}^{N} \nabla^2_\theta \log p(y_n \mid f_\theta(x_n))|_{\theta=\theta_{\text{MAP}}} (\nabla_f \log p(y_n \mid f))^2|_{f=f_{\theta_{\text{MAP}}}(x_n)} \\
&= H_{\text{GGN}} + R,
\end{aligned}$$

where $J(x_n) = \nabla_\theta f_\theta(x_n)|_{\theta=\theta_{\text{MAP}}}$. The GGN approximation is given by

$$\begin{aligned} H_{\text{GGN}} &= \sum_{n=1}^{N} J(x_n)^\top \nabla_f^2 \log p(y_n \mid f)|_{f=f_{\theta_{\text{MAP}}}(x_n)} J(x_n), \\ \Sigma_{\text{GGN}}^{-1} &= H_{\text{GGN}} + \sigma_0^2 I_p. \end{aligned}$$

Instead of considering the full Hessian, further approximations only use the diagonal [44] or a per-layer block-diagonal structure [135]. Additionally, rather than considering the full network, we can consider only a subnetwork [40] or even only the last layer [116]. These approximations are especially useful if the network considered is large.

Computing Predictive Distributions

With an approximation of the weight-posterior available in closed form, we would like to compute the expected prediction y given some input x, the data set $\mathcal{D}$ and the model f_θ:

$$\begin{aligned} p(y \mid f_\theta(x), \mathcal{D}) &= \int p(y \mid f_\theta) p(\theta \mid \mathcal{D}) d\theta \\ &\approx \int p(y \mid f_\theta) q(\theta) d\theta. \end{aligned}$$

We assume observation noise on the data, i.e., $p(y \mid f_\theta) = \mathcal{N}(y \mid f_\theta, \sigma^2 I_d)$.

Sampling One option to compute this integral is Monte Carlo sampling (see Section 2.3)

$$p(y \mid f_\theta(x), \mathcal{D}) \approx \frac{1}{S} \sum_{i=1}^{S} p(y \mid f_{\theta_i}), \quad \theta_i \sim q(\theta).$$

Linearization Instead of sampling, we can linearize the neural network with respect to the weights [54, 107]. This approximation allows for a closed form computation of predictive distribution. Additionally, the assumption of a linear f in the parameters corresponds to the GGN matching the exact Hessian.

In detail, we start by linearizing the neural network in the parameters

$$f_\theta \approx f_{\theta_{\text{MAP}}} + J(x)(\theta - \theta_{\text{MAP}}).$$

We then obtain a predictive distribution over the outputs of the network

$$p(y \mid f_\theta(x), \mathcal{D}) \approx \mathcal{N}(y \mid f_{\theta_{\text{MAP}}}(x), J(x)^T \Sigma_{\text{GGN}} J(x) + \sigma^2 I),$$

where σ^2 is the variance of the observation noise.

In case the GGN is used to approximate the Hessian, MC sampling often does not lead to good results. This is possibly due to the fact that using the GGN implicitly assumes a linear model, and hence requires the use of the linearization approach [99]. To illustrate this issue we conduct a small experiment running full HMC to compute the predictive distribution and compare it to the Laplace approach using linearization and sampling.

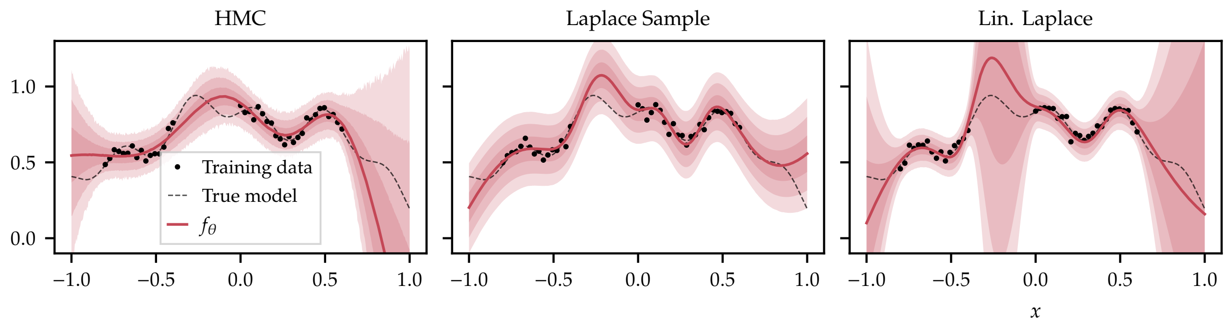


Figure 4.7: Training a Bayesian neural network on a regression data set. (*Left*) Hamiltonian Monte Carlo is used to compute the predictive distribution. (*Center*) and (*right*) employ the Laplace approximation, using sampling and linearization respectively to compute the predictive posterior. Shaded areas indicate one to three standard deviations.

Both HMC and the linearization approach give high uncertainty estimates outside the data regime in contrast to Laplace approach using sampling (see Figure 4.7).

The Laplace approximation introduces two parameters, σ_0^2 as the prior variance, and σ^2 as the observations noise. Although σ_0^2 can be computed from the weight decay factor, it is often tuned in practice. Optimization of these two parameters works via log-likelihood maximization [39].

4.4 Structure of Neural Networks

Modern neural network architectures are commonly significantly more complex than a simple multi-layer linear network. Architectures are designed to improve network performance, where performance here refers to several aspects, e.g., to decrease training time, to decrease model size, to generalize better, or to fulfill certain mathematical constraints. This chapter provides examples of certain structural choices with a particular focus on architectures motivated by scientific modelling assumptions.

First, we motivate architectures by the goal of improving training efficiency. Second, we consider how modern architectures incorporate the structure of the training data. And third, we show how the data generating process can be incorporated into the structure of neural networks. Since neural ODEs recur throughout this section, we take a closer look at this continuous perspective of deep learning.

4.4.1 Improving Training Efficiency

We start off with the observation that training deeper and deeper networks becomes increasingly difficult. The special structure of *Residual neural networks* (ResNets) [86] reduces this problem. ResNets consist of *residual blocks* containing *skip-connections*, i.e.,

$$x^{i+1} = x^i + f^i_{\theta^i}(x^i)$$

x^i is the output of the previous layer/block and x^{i+1} the input to the next layer. f^i_θ commonly corresponds to a two-layer neural network. The argument for this specific structure is that each f^i only needs to learn deviations from the identity, making it easier to train the network.

ResNets allow training of networks with up to a thousand layers [87]. ResNets are just one example of a long list of ingredients that enable the training of modern architectures.

Haber and Ruthotto [79] and He et al. [86] observe that ResNets can be interpreted as numerical solutions of ODEs using the Euler method (Definition 3.2) with step size $h = 1$. Since the Euler method is known to make a relatively large numerical error per step, Lu et al. [130] suggest replacing Euler steps with higher order Runge-Kutta methods. In this framework, we can think of the network $f^i_{\theta^i}$ as the vector field of an ODE. Instead of using only fixed step methods, Chen et al. [29] suggest taking a fully continuous view by using adaptive step-size solvers. They compare the resulting architecture, termed *neural ODE*, to a ResNet that automatically adjusts its depth. We will return to the discussion of if and when such architectures can be considered continuous in Chapter 5. In a similar vein, *Deep Equilibrium Networks* [7] aim to replace the functionality of repeatedly applying the same layer with a single layer.

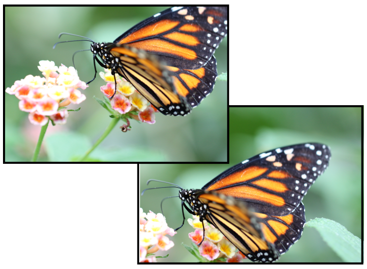


Figure 4.8: Example of a translational shift of an image. Image classification networks should remain invariant to this transformation.

4.4.2 Incorporating Data Structure

Much of the data used in deep learning has some structure that can be incorporated into the learning process to improve model performance. A prime example is image classification - the class label should not change when the position of an object is slightly shifted (see Figure 4.8). Therefore, transformations that are invariant under translations are desirable for such tasks. And indeed, convolutional neural networks implement translation invariance. Since their success on the ImageNet benchmark [43, 117], they have become the architecture of choice for image classification.

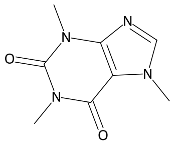


Figure 4.9: Representation of a caffeine molecule.

Another example is data with a graph structure, e.g., the structure of molecules [208] (see Figure 4.9), representations of code [228], or social networks [53]. Graph neural networks [187] are specifically designed to handle this kind of data and have become an active area of research. A prime example is AlphaFold [102], which predicts the structure of a protein given its sequence. Incorporating the graph structure of the input and output into the model architecture is a key component, along with incorporating additional biological and chemical knowledge.

4.4.3 Incorporating the Data Generating Process

For some data sets, we may have a scientific model that describes the data generating process. Incorporating this additional knowledge could make the model more data efficient or could lead to better interpolation and extrapolation. As an example, we consider dynamical systems where we might assume, that the underlying dynamics can be modelled with an ODE. However, we may not know the exact functional form of the ODE, so it may be useful to derive it in a data driven way. To maintain the ODE description, we can use *neural ODEs* [29, 182, 219], i.e., ODEs where a neural network $f_\theta : \mathbb{R} \times \mathbb{R}^n \to \mathbb{R}^n$ with trainable parameters θ describes the vector field:

$$x' = f_\theta(t, x), \quad x(t_0) = x_0.$$

We note that using standard activation functions and linear or convolutional layers, neural network are locally Lipschitz continuous on any bounded domain, thus guaranteeing existence and uniqueness of a solution (via the Picard-Lindelöf theorem 3.1). This approach can be extended to systems where a model for dynamics is partially known [220] or to energy conserving systems via Hamiltonian neural networks [74, 227]. To model noisy processes, neural ODEs can be extended to neural stochastic differential equations (SDEs) [108, 114, 122, 126, 200].

4.4.4 Incorporating the Numerical Task

Scientific models often involve solving a numerical task, which can be computationally quite expensive. The question is if and how machine learning can be used to improve computational speed. The idea of network architectures in this area is to embed the numerical problem directly into the network architecture and the optimization task, to then make use of the interpolation capabilities of the neural network. We will illustrate this using the example of *Physics informed neural networks* (PINNs) [171]. PINNs consider the task of finding the solution to a non-linear *partial differential equation* (PDE):

$$u' + \mathcal{T}[u] = 0,$$

where $\mathcal{T}$ is a nonlinear differential operator, and $u : \mathbb{R} \times \mathcal{X} \subset \mathbb{R}^d \to \mathbb{R}^d$. The problem setup also includes initial and boundary conditions for some $\{t_n^b, x_n^b\}_{n=1}^{N_b}, t_n^b \in \mathbb{R}, x_n^b \in \mathcal{X}$ such that $\mathcal{D}_n[u(t_n^b, x_n^b)] = b_n$. Here $\mathcal{D}_n$ is a differential operator necessary to define suitable boundary conditions for the PDE. Based on this definition we can then define a network architecture f_θ using a sufficiently differentiable neural network $u_\theta : \mathbb{R} \times \mathcal{X} \subset \mathbb{R}^d \to \mathbb{R}^d$:

$$f_\theta := \frac{du_\theta}{dt} + \mathcal{T}[u_\theta].$$

If f_θ is zero everywhere and u_θ satisfies the boundary conditions then u_θ is a solution to the PDE problem. This statement can be encoded in the loss

$$l(\theta) = \sum_{n=1}^{N_b} \left(\mathcal{D}_n[u_\theta(t_n^b, x_n^b)]\right)^2 + \sum_{n=1}^{N_c} f_\theta(t_n^c, x_n^c)^2$$

where t_n^c, x_n^c are some collocation points. By minimizing this loss, the neural network u_θ approximates the solution to the initial boundary value problem. This approach can also be extended to the discrete setting by considering a specific numerical method for solving the task [171]. We observe that this neural network obtains valid results by incorporating the problem formulation directly into the model architecture. Similarly, neural operators attempt to find solutions for linear PDEs [1].

In Chapter 7 we introduce a neural network architecture for integration, which essentially works by cleverly building the integration task into the network architecture.

4.4.5 A Continuous Viewpoint: Neural Ordinary Differential Equations

In the previous sections, we have motivated the structure of different network architectures. Neural ODEs have occurred twice in this context: first with the aim of making a classic architecture more compact and second with the aim of modelling dynamical systems. Based on this motivation we take a closer look at the neural ODE architecture and the wide range of applications of neural ODEs. A neural ODE is no different from a standard ODE except that the vector field is a neural network.

Definition 4.9 Neural ODE We call an ODE a *neural ODE* if the vector field is described by a neural network $f_\theta : \mathbb{R} \times \mathcal{X} \subseteq \mathbb{R}^d \to \mathcal{X}$, with weights $\theta \in \Theta \subseteq \mathbb{R}^p$. We then consider the initial value problem

$$x' = f_\theta(t, x), \quad x(t_0) = x_0.$$

To train the parameters of a neural ODE it is necessary to compute gradients of the flow (or its numerical approximation). One option is to simply use the auto differentiation capabilities of modern deep learning frameworks, and backpropagate through the numerical solver. Alternatively, Chen et al. [29] proposed using a continuous adjoint method based on Pontryagin [166], which essentially works by solving the ODE backwards. This method requires less memory than backpropagation, but can run into severe numerical issues for some problems [64]. Finding good methods for computing gradients of neural ODEs has remained an issue, and several works attempt to address this [37, 230]. Backpropagation through the solver and the adjoint method are implemented in the software library `torchdiffeq` [29], and `DiffEqFlux` [170] implements different backpropagation schemes. Generally, the choice of backpropagation method is not straightforward, as one has to take memory requirements (storing the entire trajectory) vs runtime (solving the ODE backwards) into account.

Neural ODEs are often hard to train, learning unnecessarily complex dynamics which require additional ODE solver steps. Therefore, Xia et al. [212] suggest adding a damping term to the ODE equations. Another option to improve ODE training is via an appropriate regularization of the Jacobian [56]. To improve the expressive power of the neural ODE model, Dupont et al. [50] suggest adding additional input dimensions. Other extensions to the neural ODE framework are possible, e.g., special implementations for time varying parameters and the use of higher order ODEs [137].

Applications of Neural ODEs

We now provide an overview of neural ODE applications which range from classification tasks, to modelling a dynamical system, to sampling from probability distributions.

Classification Tasks Due to their structural similarity to ResNets, neural ODEs are often used for classification tasks, although it remains difficult to achieve state-of-the-art performance with these models. Several methods aim to encode time varying weights for neural ODEs, making the resulting architecture more similar to standard ResNets [31, 169, 225]. The structure of neural ODEs or neural SDEs has shown to make architectures more robust [126, 216]. Additionally, neural SDEs can provide uncertainty estimated for classification tasks [114].

Dynamics Modelling The obvious application for neural ODEs is time series data. Yin et al. [220] propose to extend already known parametric ODE systems with a neural ODE. To incorporate knowledge like energy conservation, Zhong et al. [227] embed the Hamiltonian equations of motion into the neural network architecture. Since most neural ODE models only use the initial conditions as input to their model, Kidger et al. [109] propose a method to include all observations as inputs for the model. Neural ODEs can also be used, to extend recurrent neural networks in a continuous fashion, making the use of irregularly sampled data possible [42, 182]. Since the input is often a priori too complex to be modelled directly by a neural ODE, one option is to use the neural ODE in the latent space of a variational autoencoder [182, 219]. Neural ODEs can also be used to extend graph neural networks [165].

Generative Modelling Learning general distributions and then being able to sample from them, enables learning the structure of any data set, such as images. Normalizing flow describes the idea of using learnable transformations to get from a standard normal distribution to a distribution learned from data [175]. Chen et al. [29] and Grathwohl et al. [71] propose a continuous extension of normalizing flows using neural ODEs. The idea exploits the fact that neural ODEs can be solved both forward and backward in time, making it possible to evaluate prior and posterior in closed form. Song et al. [194] propose to use a neural SDE to learn a score function from data, leading to high quality image generation.

Part II.

Structure in Neural Networks

ResNet After All? Neural ODEs and Their Numerical Solution

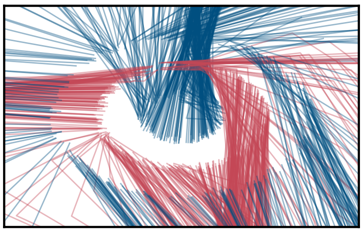
5.

Abstract

A key appeal of the Neural Ordinary Differential Equation (ODE) framework is that it seems to provide a continuous-time extension of discrete residual neural networks. As we show herein, though, trained neural ODE models actually depend on the specific numerical method used during training. If the trained model is supposed to be a flow generated from an ODE, it should be possible to choose another numerical solver with equal or smaller numerical error without loss of performance. We observe that if training relies on a solver with overly coarse discretization, then testing with another solver of equal or smaller numerical error results in a sharp drop in accuracy. In such cases, the combination of vector field and numerical method cannot be interpreted as a flow generated from an ODE, which arguably poses a fatal breakdown of the neural ODE concept. We observe, however, that there exists a critical step size beyond which the training yields a valid ODE vector field. We propose a method that monitors the behavior of the ODE solver during training to adapt its step size, aiming to ensure a valid ODE without unnecessarily increasing computational cost. We verify this adaptation algorithm on a common benchmark data set as well as a synthetic data set.

Code and experiments available at the Github repository `numerics_independent_neural_odes`

5.1 Introduction



The choice of neural network architecture is an important consideration in the deep learning community. Among a plethora of options, ResNets [86] have emerged as an important subclass of models, as they mitigate the gradient issues [8] arising with training deep neural networks by adding skip connections between the successive layers. Besides the architectural advancements inspired from the original scheme [213, 222], recently neural ODE models [29, 51, 79, 130] have been proposed as an analog of continuous-depth ResNets. While neural ODEs do not necessarily improve upon the sheer predictive performance of ResNets, they offer the vast knowledge of ODE theory to be applied to deep learning research. For instance, the authors in Yan et al. [216] discovered that for specific perturbations, neural ODEs are more robust than convolutional neural networks. Moreover, inspired by the theoretical properties of the solution curves, they propose a regularizer which improved the robustness of neural ODE models even further. However, if neural ODEs are chosen for their theoretical advantages, it is essential that the effective model—the combination of ODE problem and its solution via a particular numerical method—is a close approximation of the true analytical, but practically inaccessible ODE solution.

In this work, we study the empirical risk minimization (ERM) problem

$$l = \frac{1}{N}\sum_{n=1}^{N} c(f_\theta(x_n), y_n) \tag{5.1}$$

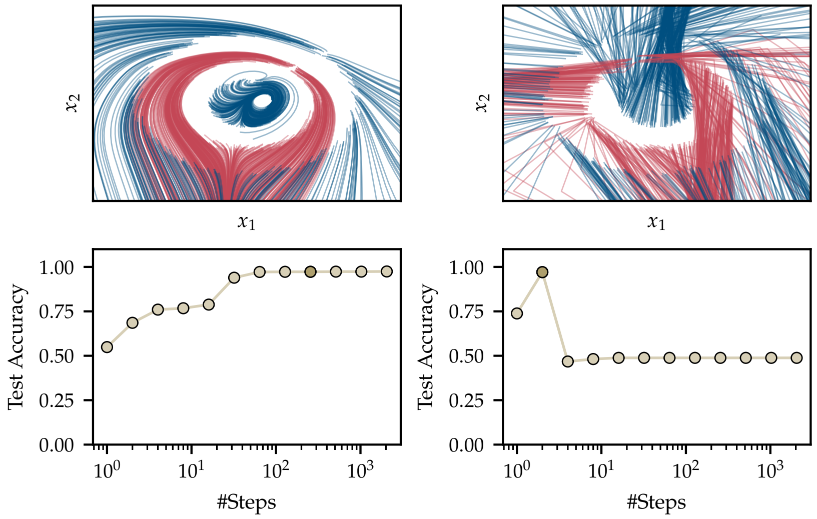


Figure 5.1: The neural ODE was trained on a classification task with a small (*left*) and large (*right*) step size. (*Top-left*) and (*top-right*) show the trajectories for the two different solvers. The colors of the trajectories indicate the label for each IVP. Panels (*bottom-left*) and (*bottom-right*) show the test accuracy of the neural ODE solver using different step sizes for testing. (◉) indicates the number of steps used for testing are the same as the number of steps used for training. (○) - the number of steps used for testing are different from the number of steps used for training.

where $\mathcal{D} = \{(x_n, y_n) \mid x_n \in \mathbb{R}^{d_x}, y_n \in \mathbb{R}^{d_y}, n = 1, \ldots, N\}$ is a set of training data, $c : \mathbb{R}^{d_x} \times \mathbb{R}^{d_y} \longrightarrow \mathbb{R}$ is a (non-negative) loss function and f is a Neural ODE model with weights θ, i.e.,

$$f = f_d \circ \varphi_T^{f_v} \circ f_u \tag{5.2}$$

where $f_x, x \in \{d, v, u\}$ are neural networks and u and d denote the upstream and downstream layers respectively. φ is defined to be the (analytical) flow of the dynamical system

$$z' = f_v(z; \theta_v), \ z(t) = \varphi_t^{f_v}(z(0)). \tag{5.3}$$

As the vector field f_v of the dynamical system is itself defined by a neural network, evaluating $\varphi_T^{f_v}$ is intractable, and we have to resort to a numerical scheme Ψ_t to compute φ_t. Ψ belongs either to a class of fixed step methods (Section 3.2.1) or is an adaptive step size solver (Section 3.2.2) as proposed in Chen et al. [29]. For fixed step solvers with step size h one can directly compute the number of steps taken by the solver #steps = Th^{-1}. We set the final time $T = 1$ for all our experiments. The global numerical error e_{train} of the model is the difference between the true, (unknown), analytical solution of the model and the numerical solution $e_{\text{train}} = ||\varphi_T(z(0)) - \Psi_T(z(0))||$ at time T. The global numerical error for a given problem can be controlled by adjusting either the step size or the local error tolerance.

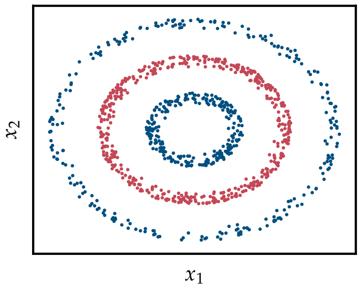


Figure 5.2: Synthetic data set used for the neural ODE experiments. The color indicates the class labels.

Since the numerical solvers play an essential role in the approximation of the solutions of an ODE, it is intuitive to ask: how does the choice of the numerical method affect the training of a neural ODE model? Specifically, does the discretization of the numerical solver impact the resulting flow of the ODE? To test the effect of the numerical solver on a neural ODE model, we first train a neural ODE on a synthetic classification task consisting of three concentric spheres, where the outer and inner sphere correspond to the same class (see Figure 5.2 and for more information see Section 5.2.4). For this problem there are no true underlying dynamics and therefore, the model only has to find some dynamics which solve the problem. We train the neural ODE model using a fixed step solver with a small step size and a solver with a large step size (see Figure 5.1

(*left*) and (*right*) respectively). If the model is trained with a large step size, then the numerically computed trajectories for the individual IVPs cross in phase space (see Figure 5.1 (*right*)). Specifically, we observe that trajectories of IVPs belonging to different classes cross. This crossing behavior contradicts the expected behavior of ODE solutions, as according to the Picard-Lindelöf theorem we expect unique solutions to the IVPs (Theorem 3.1). We observe crossing trajectories because the discretization error of the solver is so large that the resulting numerical solutions no longer maintain the properties of ODE solutions.

We observe that both, the model trained with the small step size and the model trained with the large step size, achieve very high accuracy. This leads us to the conclusion that the step size parameter is not like any other hyperparameter, as its chosen value often does not affect the performance of the model. Instead, the step size affects whether the trained model has a valid ODE interpretation. Crossing trajectories are not bad per se if the performance is all we are interested. If, however, we are interested in applying algorithms whose success is motivated from ODE theory to, for example, increase model robustness [216], then the trajectories must not cross.

We argue that if *any* discretization with similar or lesser discretization error yields the same prediction, the trained model corresponds to an ODE that is qualitatively well approximated by the applied discretization. Therefore, in our experiments we evaluate each neural ODE model with smaller and larger step sizes than the training step size. We notice that the model trained with the small step size achieves the same level of performance when using a solver with smaller discretization error for testing (Figure 5.1 (*bottom-left*)). For the model trained with the large step size, we observe a significant drop in performance if the model is evaluated using a solver with a smaller discretization error (see Figure 5.1 (*bottom-right*)). The reason for the drop in model performance is that the decision boundary of the classifier has adapted to the global numerical error e_{train} in the computed solution. For this specific example, correct classification relies on crossing trajectories as a feature. Therefore, the solutions of solvers with a smaller discretization error are no longer assigned the right class by the classifier and the neural ODE model is a ResNet model without ODE interpretation.

If we are interested in extending ODE theory to neural ODE models, we have to ensure that the trained neural ODE model indeed maintains the properties of ODE solutions. In this work we show that the training process of a neural ODE yields a discrete ResNet without valid ODE interpretation if the discretization is chosen too coarse. With our rigorous neural ODE experiments on a synthetic data set as well as CIFAR10 using both fixed step and adaptive step size methods, we show that if the precision of the solver used for training is high enough, the model does not depend on the solver used for testing as long as the test solver has a small enough discretization error. Therefore, such a model allows for a valid ODE interpretation. Based on this observation we propose an algorithm to find the coarsest discretization for which the model is independent of the solver.

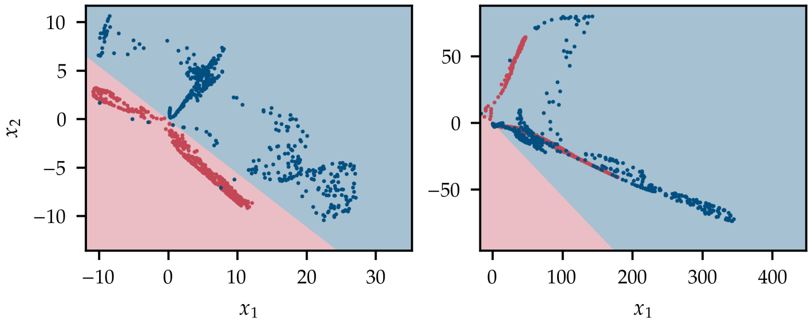


Figure 5.3: Output of the neural ODE block for different step sizes The model is the same as in Figure 5.1, trained with a step size of $h_{\text{train}} = 1/2$. The points indicate the output of the neural ODE block, the color of the points shows their true label, and the light color in the background indicates the label assigned by the classifier to this region. The model was tested with a step size of $h_{\text{test}} = 1/2$ in (*left*) and $h_{\text{test}} = 1/4$ in (*right*). Axis were scaled such that the final locations of all test data points after passing through the ODE block are shown.

5.2 Interaction of neural ODE and ODE Solver can Lead to Discrete Dynamics

We want to study how the neural ODE is affected by the specific solver configuration used for training. To this end, in our experiments we first train each model with a specific step size h_{train} (or a specific tolerance $\text{tol}_{\text{train}}$ in the case of adaptive step size methods). For the remainder of this section we will only consider fixed step solvers, but all points made equally hold for adaptive step methods, as shown by our experiments. Post-training, we evaluate the trained models using different step sizes h_{test} and note how using smaller steps sizes $h_{\text{test}} < h_{\text{train}}$ affects the model performance. We expect that if the model yields good performance in the limiting behavior using smaller and smaller step sizes $h_{\text{test}} \rightarrow 0$ for testing, then model should correspond to a valid ODE. For a model trained with a *small* step size, we find that the numerical solutions do not change drastically if the testing step size h_{test} is decreased (see Figure 5.1 (c)). But if the step size h_{train} is beyond a critical value, the model accumulates a large global numerical error e_{train}. The decision layer may use these drastically altered solutions as a signal/feature in the downstream computations. In this case, the model is tied to a *specific*, *discrete* flow and the model remains no longer valid in the limit of using smaller and smaller step sizes $h_{\text{test}} \rightarrow 0$.

5.2.1 The Trajectory Crossing Problem

In this subsection, we examine the trajectory crossing effect which causes the ODE interpretation to break down. First, we look at the numerically computed trajectories in phase space of a neural ODE model trained with a very large step size of $h_{\text{train}} = 1/2$ (see Figure 5.1 (*right*)). A key observation is that the trajectories cross in phase space. This crossing happens because the step size h_{train} is much bigger than the length scale at which the vector field changes, thus missing “the curvature” of the true solution. Specifically, we observe that the model exploits this trajectory crossing behavior as a feature to separate observations from different classes. This is a clear indication that these trajectories do not approximate the true analytical solution of an ODE, as according to the Picard-Lindelöf theorem (Theorem 3.1), solutions of ODEs do not cross in phase space. Since the numerical solutions using smaller step sizes $h_{\text{test}} < h_{\text{train}}$ no longer maintain the crossing trajectory feature,

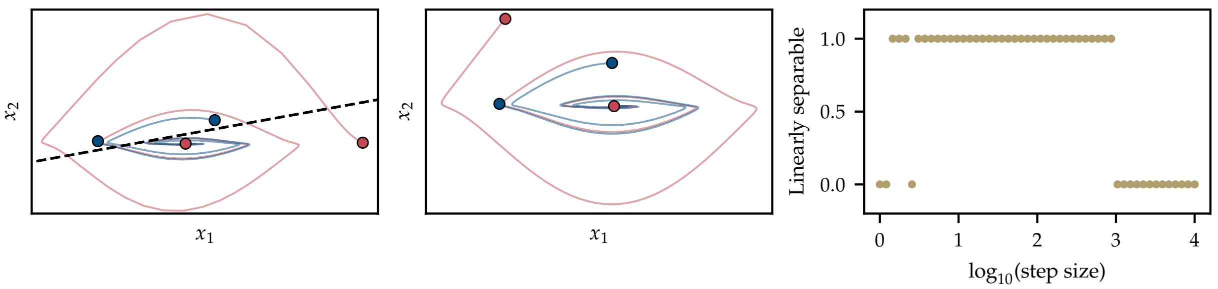


Figure 5.4: Solutions to Equation 5.4 using the Euler method with step size $h = 10^{-2.5}$ (*left*) and $h = 10^{-3.5}$ (*center*). The points indicate where each IVP ends up in phase space and the color indicates the class the solution belongs to. The trajectories taken by the numerical solver are shown in the color indicating the respective class. In (*left*) the black dotted line indicates one possibility how to separate the data linearly. (*Right*) shows for which step sizes the solution is linearly separable (1) or not (0).

the classifier cannot separate the data with the learned vector field (see Figure 5.3).

5.2.2 Lady Windermere's Fan

In cases where trajectory crossings do *not* occur, other, more subtle effects can also lead to a drop in performance in the limit of using smaller and smaller test step sizes $h_{\text{test}} \longrightarrow 0$. The compound effect of local numerical error leads to a biased global error which is sometimes exploited as a feature in downstream blocks. This effect how the local error gets accumulated into global error was coined as Lady Windermere's Fan in Hairer et al. [80, Ch. I.7].

To understand these effects we introduce an example based on the XOR problem $D = \{((0,0) \mapsto 0), ((1,1) \mapsto 0), ((0,1) \mapsto 1), ((1,0) \mapsto 1)\}$. This data set cannot be classified correctly in two dimensions with a linear decision boundary [66, Ch. 1.2]. Therefore, we consider the ODE

$$z'(t) = \begin{pmatrix} \alpha & 1 \\ -\gamma ||z||^{\delta} & \beta \end{pmatrix} z. \tag{5.4}$$

The qualitative behavior of the analytical flow are increasing ellipsoids with ever-increasing rotational speed. We chose this problem as an example based on the knowledge that the precision of a solver influences how the rotational speed of the ellipsoids is resolved. Therefore, this problem is useful in illustrating how the numerical accuracy of the solver can affect the final solution.

Figure 5.4 depicts the numerical solution of this flow with one set of fixed parameters and different step sizes $h = 10^{-2.5}, 10^{-3.5}$. For both step sizes we do not observe crossing trajectories, but the final solutions differ greatly. For $h = 10^{-2.5}$ the numerical flow produces a transformation in which the data points can be separated linearly. But for the smaller step size of $h = 10^{-3.5}$, the numerical solution is no longer linearly separable. The problem here is that the numerical solution using the larger step size is not accurate enough to resolve the rotational velocity. For each step the local error gets accumulated into the global error. In Figure 5.4 (*left*), the accumulation of error in the numerical solution results in a valid feature for a linear decision (classification) layer. The reason for this is that the global numerical errors e_{train} are biased. We define as the *fingerprint* of the method the structure in the global numerical error. The decision layer then adapts to this method specific fingerprint. How this fingerprint affects the performance of the model when using smaller step size h_{test} is dependent on two aspects. First, does the data remain separable when using smaller step sizes $h_{\text{test}} < h_{\text{train}}$? If not, we will observe a significant drop in performance. Second, how sensitive is the decision layer to changes in the solutions, and how much do the numerical solutions change when decreasing the test step size $h_{\text{test}} \to 0$? Essentially, the input sensitivity of the downstream layer should be less than the output sensitivity of $h_{\text{test}} < h_{\text{train}}$. For the decision layer, there should exist a sensitivity threshold d such that $f_d(z(T) + \delta) = f_d(z(T)) \; \forall ||\delta|| < d$. Thus, if two solvers compute the same solution up to δ, the classifier identifies these solutions as the same class and the result of the model is not affected by the interchanging these solvers.

5.2.3 Discussion

We have just described two effects, trajectory crossing and Lady Windermere's fan, which can lead to a drop in performance when the model is tested with a different solver. The trajectory crossing effect is a clear indication that the model violates ODE-semantics, and it is not valid to apply ODE-theory to this model. But even if we do not observe trajectory crossing for some step size $\tilde{h}$ we are not guaranteed to not observe trajectory crossings for all $h < \tilde{h}$ (see Appendix Section B.1 for an example). On the other hand, note that the occurrence of Lady Windermere's fan as a downstream feature *does not* violate an ODE interpretation of the numerical flow. However, we argue that a pronounced Lady Windermere's fan should still be avoided as a downstream feature because the feature vanishes in the continuous-depth limit leading the neural ODE model ad absurdum. Additionally, distinguishing Lady Windermere's fan from the trajectory crossing problem is hard without visualization as both effects lead to a drop in performance if a solver with higher numerical accuracy is used for testing. However, there exists a critical step size/tolerance θ_{crit} where convergence is close to floating point accuracy and both effects vanish in the limit. As a first step we propose to check how robust the model is with respect to the step size/tolerance to ensure that the resulting model is in a regime where ODE-ness is guaranteed and therefore one can apply reasoning from ODE theory to the model.

The current implementation of neural ODEs does not ensure that the model is driven towards continuous semantics as there are no checks in the gradient update ensuring that the model remains a valid ODE nor are there penalties in the loss function if the neural ODE model becomes tied to a specific numerical configuration. An interesting direction would also be to regularize the neural ODE block towards continuous semantics. One idea is to restrict the Lipschitz constant to below 1, as proposed by Behrmann et al. [13] for ResNets which avoids crossing trajectories.

5.2.4 Experiments

For our experiments, we introduce a classification task based on the concentric sphere data set proposed by Dupont et al. [50] (see Figure 5.2).

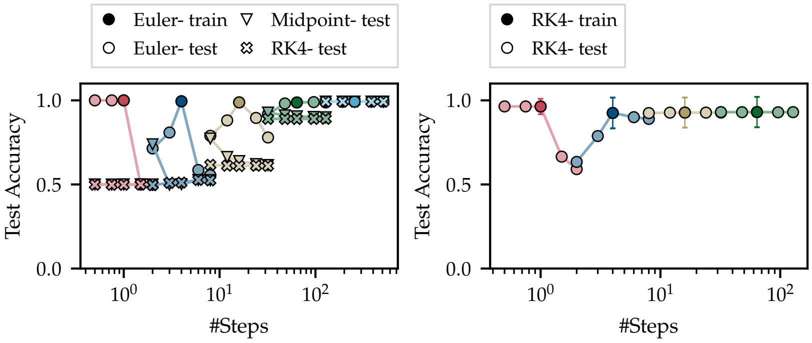


Figure 5.5: A neural ODE trained on the Sphere2 data set. The model is tested with different solvers and different step sizes. Dark circles indicate that the same solver is used for training and testing. Light data indicates a solver with different settings is used for testing. The Euler method and a 4th order Runge-Kutta was used for training (*left* and *right* respectively).

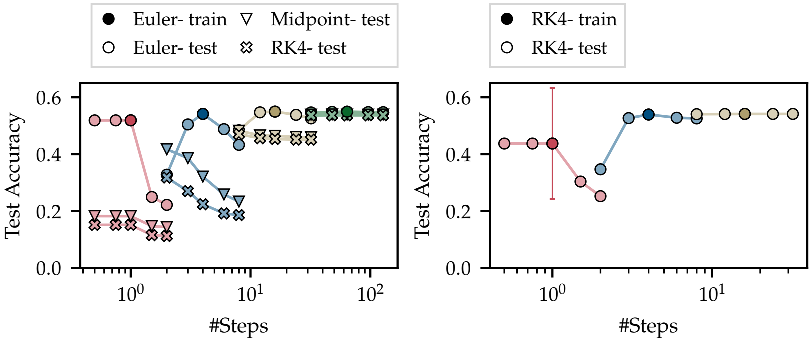


Figure 5.6: A neural ODE trained on the CIFAR10 data set. The model is tested with different solvers and different step sizes. Dark circles indicate that the same solver is used for training and testing. Light data indicates a solver with different settings is used for testing. The Euler method and the RK4 method were used for training (*left* and *right* respectively).

Whether this data set can be fully described by an autonomous ODE, is dependent on the degrees of freedom introduced by combining the neural ODE with additional downstream (and upstream) layers.

In this subsection, we present results from the experiments performed on Sphere2 and CIFAR10 data sets using fixed step and adaptive step solvers. For additional results on MNIST we refer to the Appendix Section B.2. The aim of these experiments is to analyze the dynamics of neural ODEs and show its dependence on the specific solver used during training by testing the model with a different solver configuration. In the main experiments presented in the paper, we choose to back-propagate through the numerical solver. The results pertaining to the adjoint method [29] are provided in the Appendix Section B.2. For all our experiments, we do not use an upstream block f_u similar to the architectures proposed in Dupont et al. [50]. Additionally, we decided to only use a *single* ODE block and a simple architecture for the classifier. We chose such an architectural scheme to maximize the modeling contributions of the ODE block. We do not expect to achieve state-of-the-art results with this simple architecture, but we expect our results to remain valid for more complicated architectures.

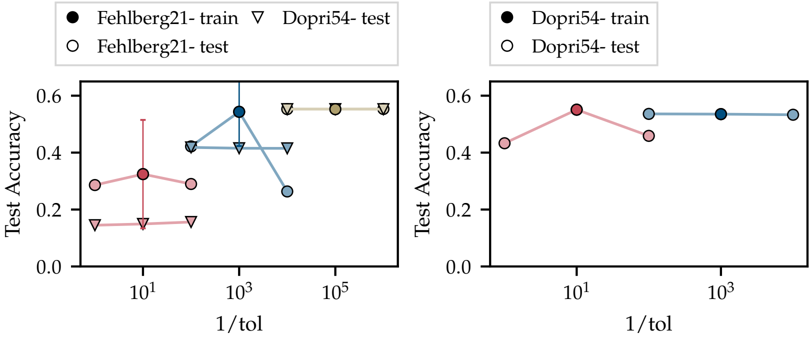


Figure 5.7: A neural ODE trained on the CIFAR10 data set. The model is tested with different solvers and different tolerances. Dark circles indicate that the same solver is used for training and testing. Light data indicates a solver with different settings is used for testing. The Fehlberg21 method and the Dopri54 method were used for training (*left* and *right* respectively).

For training the neural ODE with fixed step solvers, the Euler method and a 4th order Runge-Kutta (RK4) method were used (descriptions of these methods can be found in Section 3.2.1, (for RK4 we used the 3/8-rule)). The trained neural ODE was then tested with different step sizes and solvers. For a neural ODE trained with the Euler method, the model was tested with the Euler method, the midpoint method and the RK4 method. The testing step size was chosen as a factor of 0.5, 0.75, 1, 1.5, and 2 of the original step size used for training. For RK4, we only tested using the RK4 method with different step sizes. Likewise, the adaptive step solver experiments were performed using Fehlberg21 and Dopri54 (see Section 3.2.2). The models were trained and tested using different tolerances and solvers. The models trained using Fehlberg21 were tested using Fehlberg21 and Dopri54, whereas the models trained using Dopri54 were only tested using Dopri54. The testing tolerance was chosen as a factor of 0.1, 1, and 10 of the original tolerance used for training. Here we do not show the results for all training step sizes and tolerances, but reduced the data to maintain readability of the plots (for plots showing all the data see the Appendix Section B.2). We report an average over five runs, where we used an aggregation of seeds for which the neural ODE model trained successfully. We did not tune all hyperparameters to reach the best performance for each solver configuration. Rather, we focused on hyperparameters that worked well across the entire range of step sizes and tolerances used for training (see Appendix for the choice of hyperparameters and the architecture of the neural ODE).

As shown in Figure 5.5, Figure 5.6, and Figure 5.7, when training and testing the model with the same solver configuration, the test accuracy does not show any clear dependence on the step size or tolerance. Since we did not tune the learning rate for each step size/tolerance, any visible trends could be due to this choice. Indeed, many solver configurations work well in practice, but only for small enough step sizes/tolerances the model represents a valid ODE. On both data sets, we observe similar behavior for dependence of the test accuracy on the test solver: when using large step sizes/tolerances for training, the neural ODE shows dependence on the solver used for testing. But there exists some critical step size/tolerance below which the model shows no clear dependence on the test solver as long as this test solver has equal or smaller numerical error than the solver used for training. For additional experimental results we refer the reader to the Appendix Section B.2.1.

The aforementioned dynamics of neural ODE were also verifiable in the adaptive step solver experiments (see Figure 5.7). In this case, the trained model's test accuracy was dependent on the configuration of the test solver below a critical tolerance value. For additional results on the Sphere2 data set we refer the reader to the Appendix Section B.2.2.

5.3 Algorithm for Step Size Adaptation

Although neural ODEs achieve good accuracy for a large variety of solver configurations, if theoretic results of ODEs are to be applicable to neural ODEs, it is paramount to find a solution corresponding to an ODE flow. To ensure this, we propose an algorithm that checks whether the neural ODE remains independent of the specific train solver configuration and

Algorithm 1: Step and tolerance adaptation algorithm

```
Inputs $\epsilon$, train_solver, test_solver, model;
while Training do
    batch = draw_batch(data);
    logits = model.forward_pass(batch, train_solver($\epsilon$));
    loss = model.calculate_loss(logits);
    train_solver_acc = model.calculate_acc(logits);
    if Iteration % 50 == 0 then
        logits = model.do_forward_pass(batch, test_solver($\epsilon$));
        test_solver_acc = model.calculate_acc(logits);
        if |train_solver_acc-test_solver_acc| > 0.1 then
            $\epsilon = 0.5\,\epsilon$;
        else
            $\epsilon = 1.1\,\epsilon$;
    model.update(loss);
```

adapts the step size for fixed step solver and the tolerance for adaptive solvers if necessary. The proposed algorithm tries to find solver settings throughout training which keep the number of function evaluations small, while maintaining continuous semantics.

It is important to note that adaptive step size methods with one fixed tolerance parameter do not solve this issue, as embedded methods can severely underestimate the local numerical error if the vector field is not sufficiently smooth [80]. In contrast to the common application of such methods, in the case of neural ODEs we cannot choose the appropriate solver and tolerance for a given problem as the vector field of the neural ODE block is changing throughout training. While there always exists a low enough tolerance such that the adaptation issue does not occur, this low enough tolerance may be prohibitively small in practice and is certainly leaving runtime efficiency on the table.

So far, there does not exist any other algorithm that we are aware of which solves the issue. The aim of the proposed algorithms is not to achieve state-of-the-art results, but rather be a first step towards ensuring that trained neural ODE models can be viewed independently of the solver configuration used for training. Here we will describe the algorithm for the fixed step solvers, which shows promising results for an equivalent algorithm for adaptive methods (see in the Appendix Section B.3). Pseudocode for both settings is presented in Algorithm 1.

First the algorithm has to initialize the accuracy parameter ϵ, which corresponds to the step size h for fixed step solvers and the tolerance for adaptive step size solvers. The initial step size is chosen according to an algorithm described by Hairer et al. [80][p. 169] which ensures that the neural ODE chooses an appropriate step size for all neural networks and initializations. In our experiments we found that the initial step size suggested by the algorithm is not too small, which makes the algorithm useful in practice. The neural ODE starts training with the proposed accuracy parameter ϵ. After a predefined number of iterations (we chose $k = 50$), the algorithm checks whether the model can still be interpreted as a valid ODE: the accuracy is calculated over one batch with the train solver and with a test solver, where the test solver has

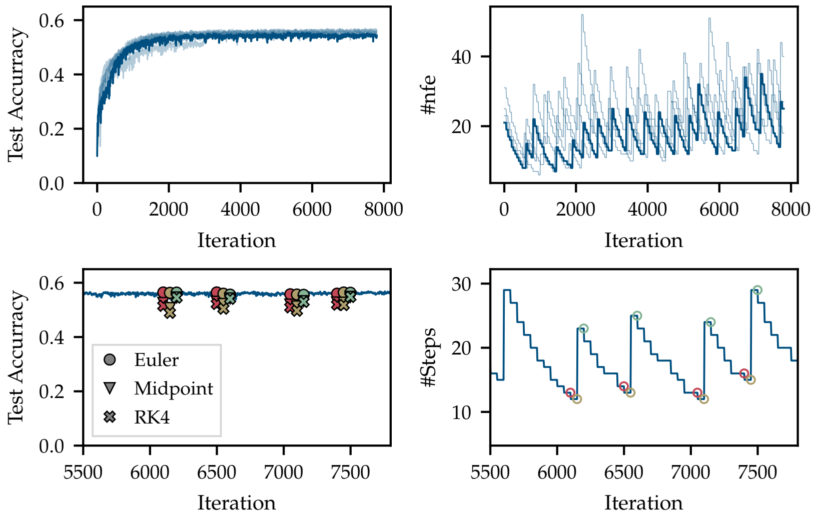


Figure 5.8: Using the step adaptation algorithm for training on CIFAR10. (*Top-left*) shows the test accuracy over the course of training for five different seeds. (*top-right*) shows the number of steps chosen by the algorithm over the course of training. (*Bottom-left*) shows the test accuracy. At certain points in time (also marked in (*bottom-right*)), the model is evaluated with solvers of smaller discretization error. (*Bottom-right*) shows the number of steps chosen by the algorithm.

a smaller discretization error than the train solver. To ensure that the test solver has a smaller discretization error than the train solver, the accuracy parameter ϵ is adjusted for testing if necessary (see Appendix Section B.3 for details). If test and train solver show a significant difference in performance, we decrease the accuracy parameter and let the model train with this accuracy parameter to regain valid ODE dynamics. If performance of test solver and train solver agree up to a threshold, we cautiously increase the accuracy parameter.

Unlike in ODE solvers, the difference between train and test accuracy does not tell by how much the step size needs to be adapted, so we choose some constant multiplicative factor that works well in practice (see Algorithm 1 for a simplified version and the Appendix B.3 for details). The algorithm was robust against small changes to the constants in the algorithm.

5.3.1 Experiments

We test the step adaptation algorithm on two different data sets: the synthetic data set and on CIFAR10 (see Appendix B.2.2 for additional results). We use the Euler method as the train solver and the midpoint method as the test solver (additional configurations are found in the Appendix B.2.2). On all data sets, we observe that the number of steps taken by the solver fluctuate over the course of training (see Figure 5.8). The reason for such a behavior is that the algorithm increases the step size until the step size is too large and training with this step size leads to an adaptation of the vector field to this particular step size. Upon continuing training with a smaller step size, this behavior is corrected, and the algorithm starts increasing the step size again. To compare the results of the step adaptation algorithm to the results of the grid search, we detail accuracy as well as number of average function evaluations (NFE) per iteration. For the grid search, we determine the critical number of steps using the same method as in the step adaptation algorithm. We report the two step sizes above and below the critical step size which were part of the grid search. For the step adaptation algorithm we calculate the NFE per iteration by including all function evaluations over course of training (see Table 5.1). The achieved accuracy and step size found

Table 5.1: Results for the accuracy and the number of function evaluations to achieve time continuous dynamics using a grid search and the proposed step adaptation algorithm. For the grid search, we report the accuracy of the run with the smallest step size above the critical threshold.

	Grid search		Step adaptation algorithm	
Data set	NFE	Accuracy	NFE	Accuracy
Concentric spheres 2d	65-129	98.7 ± 1.0%	100.5	98.9 ± 0.6%
Cifar10	17-33	54.7 ± 0.3%	21.9	55.0 ± 0.8%

by our algorithm is on par with the smallest step size above the critical threshold thereby eliminating the need for a grid search.

5.4 Related Work

The connections between ResNets and ODEs have been discussed in E [51], Haber and Ruthotto [79], Lu et al. [130], and Sonoda and Murata [195]. The authors in Behrmann et al. [13] use similar ideas to build an invertible ResNet. Likewise, additional knowledge about the ODE solvers can be exploited to create more stable and robust architectures with a ResNet backend [16, 26, 32, 34, 78, 79, 186].

Continuous-depth deep learning was first proposed in Chen et al. [29] and E [51]. Although ResNets are universal function approximators [123], neural ODEs require specific architectural choices to be as expressive as their discrete counterparts [50, 121, 223]. In this direction, one common approach is to introduce a time-dependence for the weights of the neural network [6, 31, 169, 225]. Other solutions include novel neural ODE models [129, 136] with improved training behavior, and variants based on kernels [157] and Gaussian processes [88]. Adaptive ResNet architectures have been proposed in Chang et al. [27] and Veit and Belongie [202]. The dynamical systems view of ResNets has lead to the development of methods using time step control as a part of the ResNet architecture [218, 224]. Thorpe and Gennip [198] show that in the deep limit the neural ODE block and its weights converge. This supports our argument for the existence of a critical step size. Weinan et al. [206] and Bo et al. [19] show the theoretical implications and advantages a continuous formulation ResNet models has.

Gusak et al. [77] and Zhuang et al. [230] observe a drop in performance when changing to numerically more precise solvers. In a similar vein as our work, Queiruga et al. [169] study how the solver influences the neural ODE model, showing that a model trained with the Euler method can have significantly lower performance when tested with a higher order solver. To avoid this issue, they propose to use higher order solvers for training neural ODEs. Krishnapriyan et al. [115] discuss how to achieve a continuous neural ODE model in context of modelling dynamics of physical systems. Gusak et al. [76] look at the role the exact parametrization of the Runge-Kutta method has.

5.5 Conclusion

We have shown that the step size of fixed step solvers and the tolerance for adaptive methods used for training neural ODEs impacts whether the resulting model maintains properties of ODE solutions. As a simple test that works well in practice, we conclude that the model only corresponds to a continuous ODE flow, if the performance does not depend on the exact solver configuration. We illustrated that the reasons for the model to become dependent on a specific train solver configuration are the use of the bias in the numerical global errors as a feature by the classifier, and the sensitivity of the classifier to changes in the numerical solution. We have verified this behavior on CIFAR10 as well as a synthetic data set using fixed step and adaptive methods. Based on these observations, we developed step size and tolerance adaptation algorithms, which maintain a continuous ODE interpretation throughout training. For minimal loss in accuracy and computational efficiency, our step adaptation algorithm eliminates a massive grid search. In future work, we plan to eliminate the oscillatory behavior of the adaptation algorithm and improve the tolerance adaptation algorithm to guarantee robust training on many data sets.

Uncertainty and Structure in Neural Ordinary Differential Equations 6.

Abstract

Neural ordinary differential equations are an emerging class of deep learning models for dynamical systems. They are particularly useful for learning an ODE vector field from observed trajectories (i.e., inverse problems). We here consider aspects of these models relevant for their application in science and engineering. Scientific predictions generally require structured uncertainty estimates. As a first contribution, we show that basic and lightweight Bayesian deep learning techniques like the Laplace approximation can be applied to neural ODEs to yield structured and meaningful uncertainty quantification. But, in the scientific domain, available information often goes beyond raw trajectories, and also includes mechanistic knowledge, e.g., in the form of conservation laws. We explore how mechanistic knowledge and uncertainty quantification interact on two recently proposed neural ODE frameworks—symplectic neural ODEs and physical models augmented with neural ODEs. In particular, uncertainty reflects the effect of mechanistic information more directly than the predictive power of the trained model could. And vice versa, structure can improve the extrapolation abilities of neural ODEs, a fact that can be best assessed in practice through uncertainty estimates. Our experimental analysis demonstrates the effectiveness of the Laplace approach on both low dimensional ODE problems and a high dimensional partial differential equation.

6.1 Introduction

Ordinary differential equations are a powerful tool for modelling dynamical systems. If the dynamics of the underlying system are partially unknown and only sampled trajectories are available, modelling the vector field poses a learning problem. One option is to parametrize the right-hand side of an ODE with a neural network, commonly known as a neural ODE [29]. Yet, even if the exact parametric form of the underlying dynamics is unknown, we often have some structural information available. Examples include partial knowledge of the parametric form, or knowledge of symmetries or conservation laws observed by the system. This structural knowledge can be incorporated in the neural ODE architecture. For example, Zhong et al. [226, 227] encode Hamiltonian dynamics and dissipative Hamiltonian dynamics into the structure of the neural ODE using Hamiltonian neural networks [74]. Yin et al. [220] exploit knowledge about the underlying physical model by augmenting a known parametric model with a neural ODE. Both approaches provide a more informative prior on the network architecture giving the models superior extrapolation behavior in comparison to plain neural ODEs. This kind of structure helps, but does not completely remove the need for training data. When there is just not enough data available to identify the system, meaningful predictive uncertainties are crucial. Structured

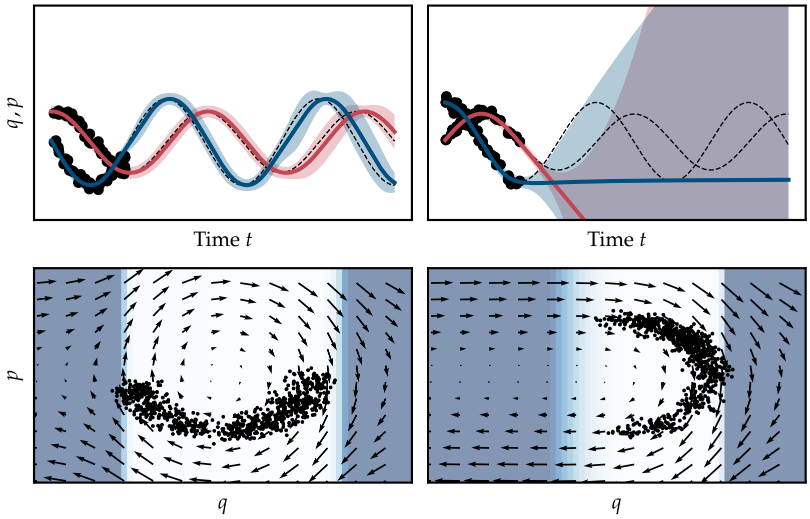


Figure 6.1: *Structure of training data impacts model uncertainty.* Training of a Hamiltonian neural ODE on two different data sets (*bottom*), with Laplace-approximated uncertainty. q, p describe position and momentum of the particle. (*Top*) show trajectories for each data set, solid lines correspond to the MAP output, (🟥) corresponds to q, (🟦) corresponds to p. (*Bottom*) Vector field recovered by the model. Background color indicates the uncertainty estimates (bright means certain, dark means uncertain).

uncertainty can help quantify the benefit arising from structural prior knowledge. Bayesian inference provides the framework to construct and quantify this uncertainty. Generally, a fully Bayesian approach can be slow or infeasible in the context of deep learning but the Laplace approximation [39, 133, 178] enables the use of approximate Bayesian methods in deep learning. The advantage of the Laplace approximation is that it is applied post-training, which avoids adding additional complexity to the model during training, and the model maintains the predictive performance of the *maximum a posteriori* (MAP) trained model.

In this work, we apply the Laplace approximation to neural ODEs to obtain uncertainty estimates for ODE solutions and the vector field. Doing so is not a straightforward application of previous works on Laplace approximations, because of the nonlinear nature of the ODE solution. We then demonstrate that the Laplace approximated neural ODEs provide meaningful, structured uncertainty, which in turn provides novel insight into the information provided by mechanistic knowledge. Specifically, the uncertainty estimates inform us how confident we can be in the model's extrapolation.

As an example for intuition, we use a Hamiltonian neural ODE (for details see Section 6.4.1), trained on data generated from the harmonic oscillator. We apply the Laplace approach to find uncertainty estimates for the trained model. The harmonic oscillator (without friction) is the textbook case of an energy-conserving system, and Hamiltonian neural ODEs capture precisely this conservation property. We use two slightly different data sets, the only difference being they are shifted by a quarter period (corresponding to a rotation by 90 degrees in phase space see Figure 6.1 (*bottom*)). For the first data set the solution in the extrapolation regime follows the true solution closely (see Figure 6.1 (*top-left*)). This behavior is reflected in the low uncertainties around the solution and the large region of high confidence in the vector field (Figure 6.1 (*top-left*) and (*top-right*)). On the other hand, for the second data set the extrapolation diverges quickly from the true solution, which is reflected in the high uncertainty in the extrapolation region. The reason for this difference in model precision is that the architecture captures the dependence on p explicitly, which can be exploited in one case, but not in the other. The same raw number of data points can thus be more or less informative, depending on where they lie in phase space.

In dynamical systems, the trajectory may leave the data domain eventually. Even if the combination of structural prior and data set is sufficient to provide good extrapolations close to the training conditions, small changes in the initial conditions can eradicate this ability. Without uncertainty estimates (or knowledge of the true dynamics), it is then difficult to judge the validity of the extrapolation.

6.2 Technical Background

This briefly discusses neural ODEs. Section 6.3 then combines neural ODEs and the Laplace approximation (see Section 4.3) to find uncertainty estimates for neural ODEs.

6.2.1 Neural ODEs

Neural ODEs are differential equations where the right-hand side, the vector field, is parametrized by a neural network $f_\theta(z)$ with weights θ

$$z' = f_\theta(z), \quad t \in [t_0, t_N], \quad z(t_0) = z_0.$$

In general, neural ODEs cannot be solved analytically, and a numerical scheme has to be employed, here denoted by `ODESolve` (Runge-Kutta methods [80] are common concrete choices):

$$z(t_n) = \texttt{ODESolve}(f_\theta, z_0, [t_0, t_n]).$$

t_n denotes the time point of a specific output.

We consider regression tasks $\mathcal{D} = (z_0, t_0, \{t_n, y_n\}_{n=1}^N)$, where y_n defines the outputs and t_n the corresponding points in time. This translates to an empirical risk minimization task of the form

$$l(\theta) = \sum_{(x_n, y_n) \in \mathcal{D}} c(\texttt{ODESolve}(f_\theta, x_n), y_n),$$

where c is a standard loss function (i.e., square loss for regression tasks). We use $x_n = \{z_0, [t_0, t_n]\}$ to denote the inputs for the `ODESolve`.

6.3 Laplace Approximation for Neural ODEs

Where neural ODEs are used in the scientific domain, the model output should include quantified uncertainties, assessing the reliability of predictions. In this section we extend the Laplace approach to neural ODEs and introduce how to compute uncertainty estimates for neural ODEs. For our implementation we extend `laplace-torch` [39] to neural ODEs.

Given the approximate Hessian, we can calculate the predictive distribution for new data via

$$p(y \mid x, \mathcal{D}) = \int p(y \mid \texttt{ODESolve}(f_\theta, x)) q(\theta) d\theta.$$

Since this integral is analytically intractable, some form of approximation has to be applied. Thus, applying the Laplace approximation to the neural ODE architecture poses a few technical challenges. In particular, the predictive distribution of `ODESolve` can be computed by sampling and linearization. Below, we discuss both options, and argue that linearization is favorable. Finally, we show how to find the predictive distribution of the vector field.

Sampling the Network Weights A first way to approximate Equation 6.3 is to sample the weights of the neural net from the posterior distribution (see also Section 4.3)

$$p(y \mid x, \mathcal{D}) = \frac{1}{N} \sum_{i=1}^{N} p(y \mid \texttt{ODESolve}(f_{\theta_i}, x)),$$

for $\theta_i \sim q(\theta)$. The neural ODE is then solved for each of these weight configurations. This requires solving a neural ODE repeatedly, for each perturbed vector field.

Linearizing the ODESolve Farquhar et al. [54] and Khan et al. [107] suggest linearizing the neural network with respect to the weights. In this case sampling is no longer necessary since the predictive distribution can be obtained in closed form. For neural ODEs, this corresponds to linearizing the entire `ODESolve` around the MAP with respect to the parameters

$$\texttt{ODESolve}(f_\theta, x) \approx \texttt{ODESolve}(f_{\theta_{\text{MAP}}}, x) + J_{\theta_{\text{MAP}}}(x)(\theta - \theta_{\text{MAP}}),$$

where $J_{\theta_{\text{MAP}}}$ is the Jacobian of `ODESolve` with respect to the parameters

$$[J_{\theta_{\text{MAP}}}]_{i,j} = \frac{\partial \texttt{ODESolve}_i}{\partial \theta_j}(\theta_{\text{MAP}}, x).$$

The Jacobian of `ODESolve` is computed using automatic differentiation functionalities. The predictive distribution can now be obtained in closed form

$$p(y \mid x, \mathcal{D}) \approx \mathcal{N}(y \mid \texttt{ODESolve}(f_{\theta_{\text{MAP}}}, x), J_{\theta_{\text{MAP}}}(x)^T \Sigma J_{\theta_{\text{MAP}}}(x) + \sigma^2 I),$$

where σ^2 is the variance of the observation noise (for more details we refer to [39]).

Given the two approaches to find the predictive distribution of the `ODESolve`, which one is preferable? Sampling, in combination with GGN approximation of the Hessian, does not provide useful uncertainties, possibly due to the mismatch between the approximation and true Hessian. Additionally, Immer et al. [99] show that the GGN implicitly assumes linearization, so sampling may not provide additional benefits. For a comparison of the sampling and the linearization approach we

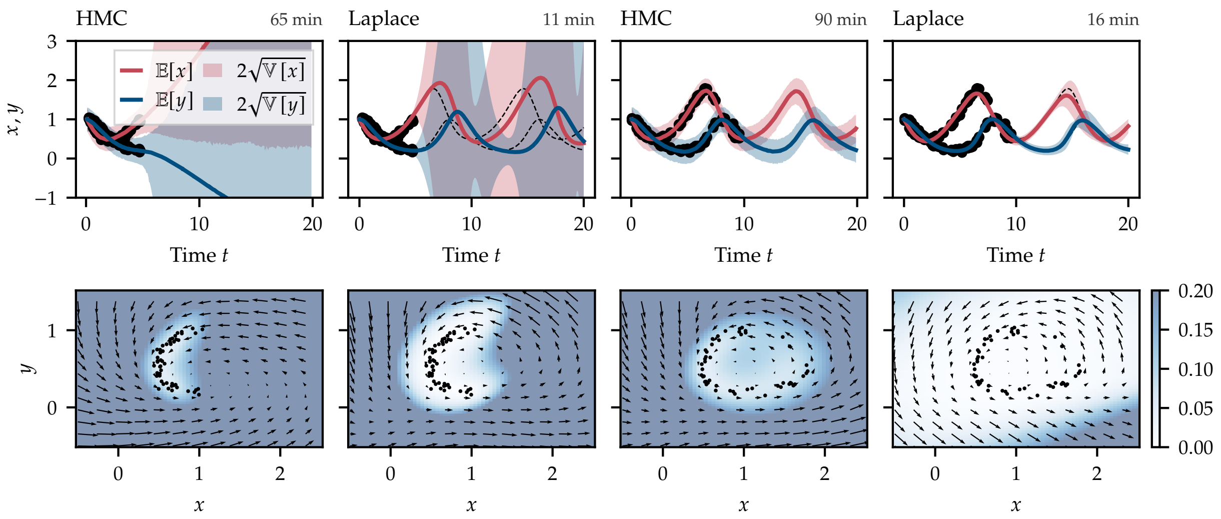


Figure 6.2: *Comparison of HMC and Laplace approximation on the Lotka-Volterra data set.* (*Top*) Trajectories (lines) with uncertainty estimates. (*Bottom*) Vector field with uncertainty estimates. The background color indicates the norm of the uncertainty estimates.

refer to Figure C.1 in the Appendix. Thus, we use a linearization of the `ODESolve` to approximate the predictive distribution.

Linearizing the Vector Field The key to understanding the interplay between data, model structure and extrapolation quality and uncertainty lies in understanding which parts of the vectorfield have been identified through the available data and the model structure. Although linearization provides closed-form predictive distributions for the outputs of the `ODESolve`, it does not provide uncertainties for the vector field f_θ. Instead of linearizing the `ODESolve`, another option is to just linearize the vector field with respect to the parameters. However, in this case, the GGN approximation is no longer equal to the Hessian of the linearized model. Linearizing the vector field allows to calculate the predictive distribution for the vector field in closed form

$$p(z' \mid z, \mathcal{D}) \approx \mathcal{N}\left(z' \mid f_{\theta_{\text{MAP}}}(z), J(z)^T \Sigma J(z)\right),$$

where J is in this case the Jacobian of the vector field f with respect to the parameters.

6.3.1 Comparison to HMC

For an empirical evaluation, we here compare the quality of uncertainty estimates provided by the Laplace approximation to Monte Carlo sampling. Specifically, we use HMC with no U-turn sampling (NUTS) (see Section 2.3.2, and implementation details in Section C.2.2 in the Appendix). To compare HMC and the Laplace approach, we introduce two data sets based the Lotka-Volterra equations (see Equation C.1 in the Appendix, for additional results on the pendulum data set we refer to Section C.3.2 in the Appendix). *Data-set-half-cycle* only provides data for half a cycle whereas *data-set-full-cycle* provides data about the entire cycle. For *data-set-half-cycle*, both HMC and Laplace are uncertain about the extrapolation outside data domain (see Figure 6.2 (*top-left*) and (*bottom-left*)), in contrast to *data-set-full-cycle* where both models extrapolate accurately with high confidence. This is also reflected in the uncertainty estimates for the vector field (see Figure 6.2). Notably, the mean extrapolation for HMC and Laplace behave differently as the Laplace extrapolation corresponds to the MAP whereas the HMC extrapolation corresponds to the mean of the posterior. Furthermore, the uncertainty estimates of the Laplace approximation decrease whenever the MAP solution returns to data domain in phase space. This is likely a shortcoming of the linearization approach, which neglects that the uncertainties should add up over the course of the `ODESolve`. As indicated by high uncertainty estimates, the similarity between the MAP extrapolation and the true solution is just coincidence as there are other weight configurations which do not extrapolate well.

One of the biggest advantages of the Laplace approximation is that is computationally much cheaper than HMC (see Figure 6.2 for the runtime of each experiment). In our experiment we choose a sufficiently small number of weights to enable HMC inference, however already doubling the network weights proved to be infeasibly slow for HMC (surpassing the allocated runtime of 24 h) whereas training and inference time for the Laplace approximation barely increased. We find that the Laplace approximation is slightly inaccurate in its uncertainty estimates due to the linearization approach, but the superior runtime performance justifies the output quality.

6.4 Structure Interacts with Uncertainty

Recent approaches [220, 227] in using neural ODEs for dynamics model learning aim to improve the modelling capabilities of neural ODEs by including structural knowledge about the dynamics. We will see that, for such models, even small shifts in the data set can disproportionally change the prediction of the neural ODE (see Figure 6.1), because the structural information causes complicated interactions with the identifiability, and hence, uncertainty of the phase space. Experiments below show that a small change in data set, and structural information can lead to a wide range of results, but that the Laplace approximation serves as a reliable tool to characterize and visualize this issue through the notion of uncertainty.

6.4.1 Hamiltonian Neural ODEs

Dynamical systems of practical concern often come with information about the underlying physics. One option to introduce knowledge about the underlying dynamics of a system is via conservation laws, e.g., the Hamiltonian equation of motion. For energy-conserving systems, the dynamics obey Hamiltonian equations of motion Equation 2.7. The Hamiltonian H is conserved over time ($dH/dt = 0$). We give a short introduction of how to construct a Hamiltonian neural ODE similar to SymODE introduced in Zhong et al. [226, 227]. For a lot of real systems it is sufficient to consider a separable Hamiltonian of the form $H(q, p) = T(p) + V(q)$. To learn such a separable Hamiltonian from data, we use neural networks to represent V and T (we term this model

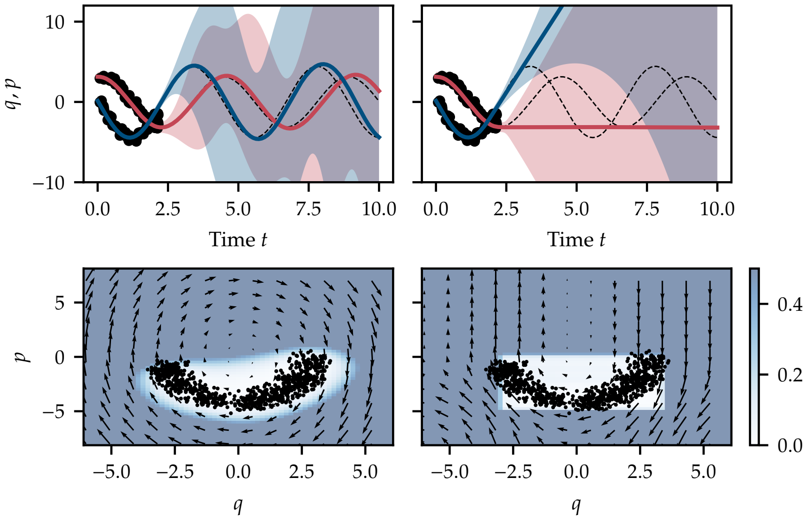


Figure 6.3: *Structural information in neural ODEs affects both point and uncertainty estimates.* Two different architectures used for training: naive (*left*) architecture and separable (*right*) Hamiltonian network. (*Top*) Solutions to the initial value problem ((▬) corresponds to q, (▬) corresponds to p) with uncertainty estimates provided by the Laplace approximation. (*Bottom*) Vector field of the neural ODE model.

separable). To further refine the prior knowledge in the architecture, we can use that the kinetic energy T is given by $T(p) = p^T M^{-1} p/2$ where M is a positive definite mass matrix, a **constrained** model. We add the parameters of the matrix M to the trainable parameters of our model. In summary, there are three neural ODE models we consider in this work: The **naive** approach of Equation 6.2.1, where the right-hand side, f, is represented by a neural network, a **separable** model and a **constrained** model.

For the following experiments, we train neural ODEs on different data sets generated from the harmonic oscillator (for experimental details and the network architecture see Appendix Section C.2.3).

Uncertainty in Hamiltonian Neural ODEs The model is only provided with data in the lower plane (i.e., *data set-lower-half*), corresponding to half a period of the particle's motion (see for example Figure 6.3). To compute uncertainty estimates for the trajectories and the vector field f_θ of the neural ODE we use the Laplace approximation with the linearization schemes introduced in Section 6.3. In the region where data is available the naive model is able to fit the data. The extrapolation for the trajectories and the vector field is close to the true dynamics, but the uncertainty estimates indicate that there is not enough information available to guarantee correct extrapolation outside the data domain (cf. Section C.2.2).

How does additional structure (i.e., a separable and a constrained neural ODE) change the extrapolation behavior on *data set-lower-half*? Similar to the naive model, the separable model becomes uncertain outside the data domain (see Figure 6.3). Although the approximation power of this model should be identical to the naive structure, and hence, an identical extrapolation quality should be achievable, there is a qualitative difference in the extrapolation of the solution and the vector field (we attribute this to the different structure in the architecture see Appendix Section C.2.3). The distinctive rectangular shape in the uncertainty estimates of the vector field reflects the separable structure of this model.

On the other hand, if we train the constrained model on *data-set-lower-half*, the uncertainty estimates of the vector field exhibit a vertical band of low uncertainty around the data (Figure **??**). This shape is directly linked to the architecture of the model: The only trainable parameters (apart from M which is a scalar), are the parameters of the potential $V_\theta(q)$ where learning V_θ depends only on the availability of data for q. So wherever data identifies V_θ the value of the Hamiltonian H is known since we use the concrete parametric form of $T(p)$. If there is data available for some (q, p) the vector field is determined in the region $(q, \mathbb{R})$, leading to uncertainty bands in the structure of the vector field. Therefore, the constrained model is able to extrapolate with relatively low uncertainty, since the extrapolation remains within the domain where training data was available for q.

Conversely, if we rotate the data set by 90 degrees (*data set-left-half*), the domain where data is available for q changes significantly. This time, however, sufficient data for q is not available and therefore, the solution shows high uncertainty in the extrapolation region (Figure 6.1). For results of the other models on *data-set-left-half* see Appendix Section C.3.3.

These results highlight the intricate effect of mechanistic knowledge on extrapolation, and its reflection in model uncertainty: While a particular data set might not be sufficient for the naive model to extrapolate accurately, mechanistic knowledge in the form of conservation laws might change this. However, seemingly benign changes in the data set (here: a shift in phase) can substantially affect the quality of the predictions. These effects are not always intuitive and hard to see in point predictions, but they become immediately evident when looking at the uncertainty estimates of the vector field and the solution provided by the Laplace approximation. Given the distinct algebraic structure of the model considered in this section and the low dimensionality of the problem we can assess which data fully identifies the vector field to allow for extrapolation. But also in more complex, high dimensional problems, where predicting the impact of the encoded structural knowledge becomes increasingly hard, the temporal evolution of the Laplace uncertainty estimates reveals the extrapolation capabilities of a model consistently. The probabilistic notion thus recommends itself when designing structural priors.

6.4.2 Augmenting Parametric Models with Neural ODEs

Instead of including knowledge of conserved quantities in the architecture of the neural ODE, another option is to consider knowledge of the parametric form of the underlying physics. But in many cases we do not have knowledge of the full dynamics, hence Yin et al. [220] suggest augmenting a known parametric model f_p with a neural network f_θ. The resulting dynamics of the model are given by

$$z' = f_\theta(z) + f_p(z), \quad t \in [t_0, t_N], \quad z(t_0) = z_0. \tag{6.1}$$

To ensure that the dynamics are not dominated by the neural ODE, Yin et al. [220] suggest regularizing f_θ (for more details we refer to their paper). We apply the Laplace approach to models trained on two data sets proposed by Yin et al. [220] – a damped pendulum and damped wave equations.

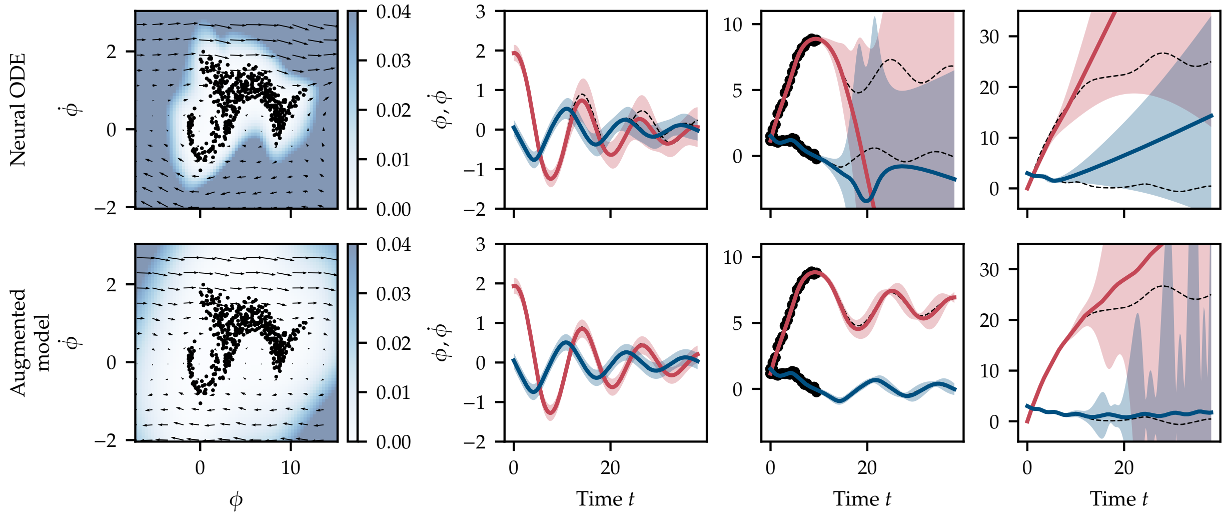


Figure 6.4: *Uncertainty estimates for augmented parametric models.* Neural ODE and augmented parametric model trained on the damped pendulum data set. (*top-left*) and (*bottom-left*) vector field with uncertainty estimates. (*Center-left, center-right, right*) trajectories for different initial conditions with uncertainty estimates, (—) corresponds to ϕ, (—) corresponds to $\dot{\phi}$.

Pendulum Dataset To show the effectiveness of the Laplace approach, we train an augmented parametric model and a plain neural ODE on a data set describing a damped pendulum $\phi'' + \omega^2 \sin\phi + \alpha\phi' = 0$, where ϕ is the angle, ω the frequency and α a damping coefficient. For the parametric model we use frictionless pendulum dynamics where we add the frequency ω as a trainable parameter to our model.

The difference in the architecture between the two models is already evident in the uncertainty estimates of the vector field (shown in Figure 6.4). For the augmented model, the region where the model has low uncertainty is larger than for the plain neural ODE model. However, far away from the data both models become uncertain. To highlight the differences between the two models we evaluate them with different initial conditions. On the first set of initial conditions (Figure 6.4 (*center-left*)) both models are able to extrapolate accurately, which is captured by the low uncertainty estimates. For the initial conditions in Figure 6.4 *center-right*, both models are able to fit the data, yet the plain neural ODE model is unable to extrapolate, reflected by the large uncertainty estimates in the extrapolation regime. There also exist initial conditions for which both models fail to extrapolate, but our experiments reveal that the Laplace approximation is able to detect these regions (see Figure 6.4 *right*). Running HMC inference on a reduced version of this task, we observe the same behavior. For the additional HMC results we refer to the Appendix Section C.3.2.

Given the complicated structure of the data set and model architecture, it is *a priori* unclear for which initial conditions the model extrapolates well. Hence, it is absolutely crucial to use uncertainty estimates to assess the models' outputs.

High Dimensional Wave Dataset We show that the Laplace approximation is also applicable to high dimensional data. In this case we use the damped wave equation as a data set, where the wave is described by a scalar function u and the wave equation is given by

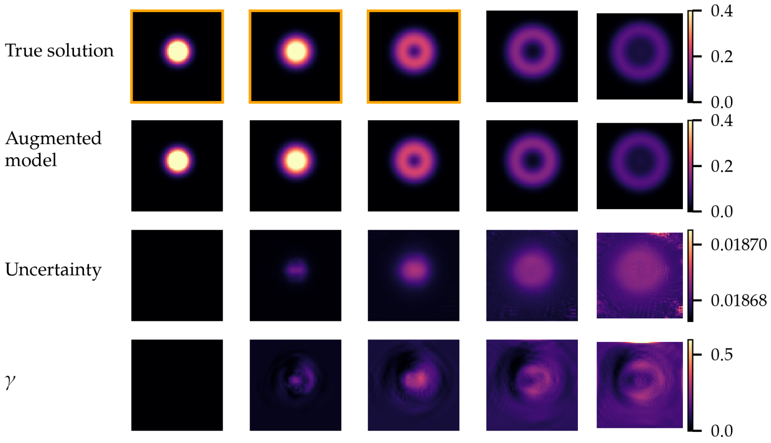


Figure 6.5: *Laplace approximation applied to wave PDE.* Training an augmented parametric model on the damped wave equation data set (Figure shows result for u — for the time derivative and additional results we refer to Appendix.). First three images (marked by orange frame) are part of the training data.

$\partial^2 u/\partial t^2 - c^2 \Delta u + k \partial u/\partial t = 0$. k is a damping coefficient. The data set consists of a 64x64 spatial discretization for u and du/dt over multiple time points. Our model consists of a parametric model for the wave equation without the damping term augmented with a neural ODE model.

The uncertainty estimates reproduce the overall structure of the wave expansion (see Section C.3.4 in the Appendix for different initial conditions). Specifically, in the extrapolation regime behind the wave front the uncertainty estimates increase (see Figure 6.5). To check if the uncertainty estimates are well calibrated, we compute the ratio of error and standard deviation, where the error is given by the difference between the true solution and the model's output. We denote this ratio by γ. The uncertainty estimates are well calibrated if γ is close to one. Overall, we find that the uncertainty estimates provided by the Laplace approximation are underconfident ($\gamma < 1$). The imperfections in the uncertainty estimates are compensated by the fact the Laplace approach facilitates the computation of uncertainty estimates on such a high dimensional data set in reasonable time—other approaches like HMC would be computationally infeasible.

6.4.3 Discussion And Outlook

While our experiments suggest that the Laplace approximation produces high quality uncertainty quantification for a variety of tasks, and for various quantities, there are some numerical and computational issues to carefully consider, which we briefly discuss here.

The computationally most expensive part of the Laplace approximation is the calculation of the GGN, and especially its inverse. But once calculated and stored, it does not have to be reevaluated for future predictions. Since neural ODEs commonly use a relatively small network size, compared to other deep learning applications, storing the Hessian need not be an issue. If it is, there are a few options available, like diagonalizing, only using the last layer or only considering a subnetwork to reduce the memory cost [39]. How well these methods apply to neural ODEs is left as future work. Another issue we face is that the GGN sometimes loses its positive-semi-definiteness, due to numerical issues (i.e., large variance in the eigenvalues). This can be alleviated by adding a small constant to the diagonal elements of the Hessian. We also found that this effect is enhanced by the structure of Hamiltonian neural networks (possibly due to the derivative structure of the activation functions). For inference, taking the Jacobian over the whole trajectory can be costly (especially for large data sets like the wave data set). However, in practice we are often only interested in the final output and dense sampling is not necessary.

6.5 Related Work

Neural ODEs [29] have been applied to a wide range of problems such as image classification [29, 31, 225], normalizing flows [29, 70], learning dynamical systems via residual neural networks [42, 109] or variational autoencoders [29, 182, 219].

Neural ODEs With Structure Greydanus et al. [74] introduce the idea of adding a Hamiltonian structure to a neural network. Zhong et al. [227] extend this idea to neural ODEs and Zhong et al. [226] add a term to Hamiltonian neural ODEs to model dissipative systems. Yin et al. [220] propose to augment parametric models with neural ODEs by regularizing the neural ODE term.

Neural ODEs With Uncertainty To model the latent space of a variational autoencoder Yildiz et al. [219] use a Bayesian neural network to describe the vector field. Similarly, Dandekar et al. [35] train a neural ODE with a Bayesian neural network as the vector field on regression and classification tasks using Monte Carlo sampling to do inference. Yang et al. [217] apply HMC and variational inference to physics-informed neural networks. Norcliffe et al. [144] propose to use neural processes to equip neural ODEs with uncertainty estimates. Relative to these works, ours is the first to construct and assess uncertainty quantification for neural ODEs with structured architectural priors.

Stochastic differential equations (SDEs) can be used to model the stochasticity of real-world processes. This approach has been transferred to neural ODEs for example for training a recurrent neural network [42] or to do variational inference using a latent stochastic differential equation [122]. To improve uncertainty quantification in image classification, Kong et al. [114] propose to use a neural SDEs and Anumasa and Srijith [3] combine a GP with a neural ODE.

GPs for Modelling ODE Dynamics Another approach to free-form dynamics modelling are Gaussian processes (GPs) [89, 91]. Hegde et al. [89] learn a posterior distribution over the vector field of the ODE. Ensinger et al. [52] encode a Hamiltonian structure in the GP and use symplectic ODE solvers to train the model. Wang et al. [205] augment incomplete physical knowledge with a GP. Ridderbusch et al. [177] propose to use GPs to learn a vector field from data by using prior structural knowledge.

6.6 Conclusion

Uncertainty is always relevant in machine learning, but particularly so for the highly structured and often unintuitive prediction of dynamical systems with neural ODEs. At its most basic, uncertainty estimates tell us if the model has seen enough data to learn the dynamics. But the position in state space, and number, of data points required for this to happen depend intricately on additional mechanistic knowledge potentially encoded in the model. Small changes in the data set can have a fatal effect on the extraploratory abilities of the neural ODE model. These aspects can be hard or impossible to spot from point predictions alone, yet may become obvious when uncertainty estimates are available.

To make neural ODEs a useful tool for scientific or engineering applications, it is thus crucial to make uncertainty estimates available at train- and test-time. The experiments presented in this work suggest that Laplace approximations, with the necessary technical adaptations for this model class, can provide such uncertainties for neural ODEs at simultaneously high fidelity and at low cost. Moreover, the uncertainty estimates are able to reflect key structural effects of mechanistic knowledge, and they thus help make neural ODEs more reliable as a tool for scientific inference.

Bayesian Numerical Integration with Neural Networks 7.

Abstract

Bayesian quadrature is a probabilistic numerical method for computing integrals. Using a Gaussian process model of the integrand to encode prior information, BQ returns a posterior distribution on the value of the integral which can be used for uncertainty quantification. Despite these advantages, BQ suffers from high computational cost, at least in its basic form. We propose an alternative to BQ based on Bayesian neural network priors, which greatly improves scalability. This is achieved through neural network architectures based on Stein operators, and an approximation of the Bayesian posterior based on the Laplace approximation. We call the method *Bayesian Stein networks*, and demonstrate its advantages on the Genz functions benchmark, posterior expectations arising in the Bayesian analysis of dynamical systems, and the expected energy production for large-scale wind farms.

7.1 Introduction

Integration is a core task in probabilistic machine learning. It is required to perform operations such as marginalizing out random variables, or computing normalization constants, predictive distributions, and posterior expectations. Here, we consider the computation of the integral of some function $f : \mathcal{X} \to \mathbb{R}$, where $\mathcal{X} \subseteq \mathbb{R}^d$, against some distribution Π with (Lebesgue) density $\pi : \mathcal{X} \to \mathbb{R}$:

$$\Pi[f] = \int_{\mathcal{X}} f(x)\pi(x)dx, \tag{7.1}$$

where we assume we have access to evaluations $\{f(x_n)\}_{n=1}^N$ at a set of points $\{x_n\}_{n=1}^N \subseteq \mathcal{X}$. A plethora of methods exist for tackling this task; the most common are MC methods, which are sampling-based methods that have been studied extensively in theory and practice [156, 180] (see Section 2.3). This subsumes naive Monte Carlo, Markov chain Monte Carlo and quasi-Monte Carlo (QMC). Sampling is (at least asymptotically, for MCMC) unbiased and thus a gold standard, but precisely for this reason, it can only converge with stochastic rate, and thus requires a large number of samples N, both for accuracy and uncertainty quantification.

This is a challenge if evaluations of f or samples from π are expensive. The former ("expensive f") emerges regularly in climate simulations or other large physical models. Section 7.5.3 provides an example with a wind farm model – a field where state-of-the-art models require hundreds of hours of CPU for a single evaluation [111, 112]. The latter ("expensive sampling") features where π is a posterior distribution for a complex model conditioned on a large amount of data. Section 7.5.2 illustrates this through an example of Bayesian inference in dynamical system.

In such scenarios, probabilistic numerical methods (PNMs) [33, 92, 93, 149, 207], and in particular Bayesian approaches, perform particularly well. For numerical integration, the principle behind Bayesian PNMs is to encode prior information about the integrand f, then condition on evaluations of f to obtain a posterior distribution over $\Pi[f]$. These methods are well suited for computationally expensive problems since informative priors can be used to encode properties of the problem and to reduce the number of evaluations needed. In addition, the posterior quantifies uncertainty for any finite value of N.

The most popular Bayesian PNM for integration is Bayesian Quadrature [22, 46, 150, 173], a method that places a Gaussian Process [174] prior on f (see Section 2.4.1). With this convenient choice of prior, the posterior on $\Pi[f]$ is a univariate Gaussian, whose mean and variance can be computed in closed form for certain combinations of prior covariance and distribution. However, for high-dimensional problems where large amounts of data are necessary, the computational cost of GPs, cubic in N, can render BQ too computationally expensive. Fast BQ methods have been proposed to resolve this issue [101, 106], but these usually work for a limited range of π or $\{x_n\}_{n=1}^N$, and therefore do not provide a widely applicable solution.

This raises the question of whether an alternative probabilistic model could be used in place of a GP within probabilistic integration. Bayesian neural networks (BNNs) are an obvious candidate, as they are known to work well in high dimensions and with large N. Unfortunately, their application to integration tasks is not straightforward since, in contrast to the GP case, analytical integration of the posterior mean of a BNN is usually intractable. This is a significant challenge which has so far prevented their use for probabilistic numerics. We resolve this challenge by proposing the concept of *Bayesian Stein (neural) networks* (BSNs), a novel BNN architecture based on a final layer constructed through a Stein operator [2]. Such choice of architecture is designed specifically so that the resulting BNN is analytically integrable (see Section 7.3.1), and hence at our disposal for numerical integration.

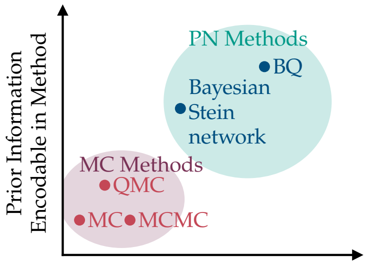


Figure 7.1: Integration methods can be compared at a high-level in terms of their computational cost and ability to include prior information. In both respects, BSNs provide a compromise in-between MC and BQ.

Given these approaches—MC, BQ, BSNs—a natural question remains: “How should we select a method for a given integration task?”. We provide an empirical answer to this question in Section 7.5, where we consider a popular benchmark data set, compute posterior expectations arising in the Bayesian treatment of dynamical systems, and estimate the expected power output of a wind farm.

Our conclusions are summarized in Figure 7.1 and presented below. If sampling π and evaluating f is computationally cheap, so one can obtain a very large number of data points relative to the complexity of the problem, then MC methods are likely the best choice. But if N is very limited due to our computational budget, then BQ is likely a better option. BSNs excel in the intermediate region where N is such that BQ becomes prohibitively expensive, but MC is not accurate enough. The architecture of neural networks, plus sophisticated deep learning software libraries, make training of (small) neural networks memory efficient and fast. However, achieving good accuracy at low training cost requires special care during training for the Stein architecture. Finding a good training setup is a main contribution of this work, outlined in Section 7.4.

For all integration methods, estimates from scarce data are imperfect, so uncertainty estimates are crucial. Bayesian deep learning provides this functionality. Full Bayesian inference is costly even for small neural networks, but we show that a lightweight Laplace approximation [133, 178] can provide good approximate uncertainty for the Stein network.

7.2 Related Work

BQ is the method most closely related to our proposed approach, and BQ is fully detailed in Section 2.4.1. Bayesian PN methods based on alternative priors have also been proposed. These include Bayesian additive regression tree priors [229], multi-output Gaussian process priors [63, 211], and Dirichlet process priors [147]. These priors each provide different advantages, such as the ability to model highly discontinuous functions, vector-valued functions, or modelling probability distributions respectively. Unfortunately none of these approaches significantly improve scalability, the main goal of this work.

The use of (non-Bayesian) neural networks for integration was previously proposed by Lloyd et al. [127]. However, their method is only applicable for uniform π and shallow networks. Si et al. [190] and Wan et al. [204] propose to use a Langevin Stein operator applied to a neural network to find good control variates for variance reduction in Monte Carlo approximations (based on an earlier construction by [146]). In contrast to their work, we use the neural network to directly compute $\Pi[f]$, and our neural network follows Bayesian semantics and can be used to quantify uncertainty. This requires a different network architecture and an efficient posterior inference algorithm.

7.3 Bayesian Stein Networks

We now describe BSNs. This requires discussing Stein operators, BNNs, and Laplace approximations.

7.3.1 Stein Neural Networks

Stein operators are a technical construction originating in probability theory, but have recently been used as a computational tool [2]. Building on this line of work, we will use Stein operators to construct the final layer of our BNNs. The reason for this is simple: given some function u (with possibly unknown mean) and a distribution π, a Stein operator can map u to a mean zero function under π. This final layer therefore allows us to construct flexible BNNs with the powerful property that *any draw from the posterior will have a known mean under π*. We now highlight this procedure in detail.

We call $\mathcal{S}$ a *Stein Operator* if for any suitably regular continuously differentiable $u : \mathbb{R}^d \rightarrow \mathbb{R}^d$, the following holds

$$\Pi\left[\mathcal{S}[u]\right] = 0. \tag{7.2}$$

Suppose $\mathcal{X} = \mathbb{R}^d$, π is continuously differentiable on $\mathcal{X}$, such that $\nabla_x \log \pi$ is well-defined ($[\nabla_x \log \pi(x)]_i = \partial \log \pi(x)/\partial x_i$ for all $i \in \{1, \dots, d\}$). One example of an operator fulfilling Equation 7.2 is the diffusion Stein operator [9, 69]:

$$\begin{aligned} \mathcal{S}_m[u](x) := &\left(m(x)^\top \nabla_x \log \pi(x)\right)^\top u(x) \\ &+ \nabla_x \cdot (m(x)u(x)) , \end{aligned} \tag{7.3}$$

where $\nabla_x \cdot u(x) = \sum_{i=1}^d \partial u_i(x)/\partial x_i$, and $m : \mathbb{R}^d \to \mathbb{R}^{d\times d}$ is an invertible matrix-valued function. This operator only requires access to $\nabla_x \log \pi(x)$, and can thus be used even if the normalization constant of π in unknown. This is an advantage if π is itself a posterior distribution. In such settings, samples form π can be obtained via MCMC, but the distribution π itself cannot be evaluated directly.

To construct BSNs, we use an architecture based on a continuously differentiable deep neural network $u_{\theta_u} : \mathcal{X} \to \mathbb{R}^d$, where $\theta_u \in \Theta_u \subseteq \mathbb{R}^p$, combined with a final layer taking the form of a Stein operator (that we call *Stein layer*). More precisely, we consider an architecture $g_\theta : \mathcal{X} \to \mathbb{R}$ where:

$$g_\theta(x) := \mathcal{S}_m \left[u_{\theta_u}\right](x) + \theta_0. \tag{7.4}$$

We call this neural network a *Stein neural network* following [190, 204], but note that we use the more general diffusion Stein operators $\mathcal{S}_m$ [9, 69]. Previous cases can be recovered with $m(x) = I_d$, where I_d is a d-dimensional identity matrix, however we will demonstrate in Section 7.5.2 that alternative choices for m can significantly improve the performance of our method.

The parameter $\theta = \{\theta_0, \theta_u\} \in \Theta \subseteq \mathbb{R}^{p+1}$ denotes the weights of the neural network g_θ. Thanks to our choice of architecture, Equation 7.2 holds, and we have:

$$\Pi\left[g_\theta\right] = \theta_0. \tag{7.5}$$

The last layer of g_θ directly tracks the integral of the network, which is the key property for our purpose: by training such a network g_θ on data from f so that $g_\theta \approx f$, we are simultaneously constructing a good approximation of the integral $\Pi[g_\theta] \approx \Pi[f]$ (see Figure 7.2 for a summary).

7.3.2 Uncertainty Estimates for Stein Neural Networks

In the context of Bayesian PNM, proposing a BNN architecture is not enough: we are also interested in *tractable uncertainty estimates over* $\Pi[f]$. We obtain this through the Laplace approximation introduced in Section 4.3.

The specific architecture of the BSN model means that all the uncertainty on $\Pi[f]$ is represented by the Bayesian posterior on θ_0. The posterior on θ_0 is then the marginal of the weight posterior $p(\theta|\mathcal{D})$. Bayesian inference for deep networks provides uncertainty estimates [132, 139] through $p(\theta|\mathcal{D})$, but this posterior is intractable in general. MCMC is a prominent tool for approximating $p(\theta|\mathcal{D})$, but using it within an

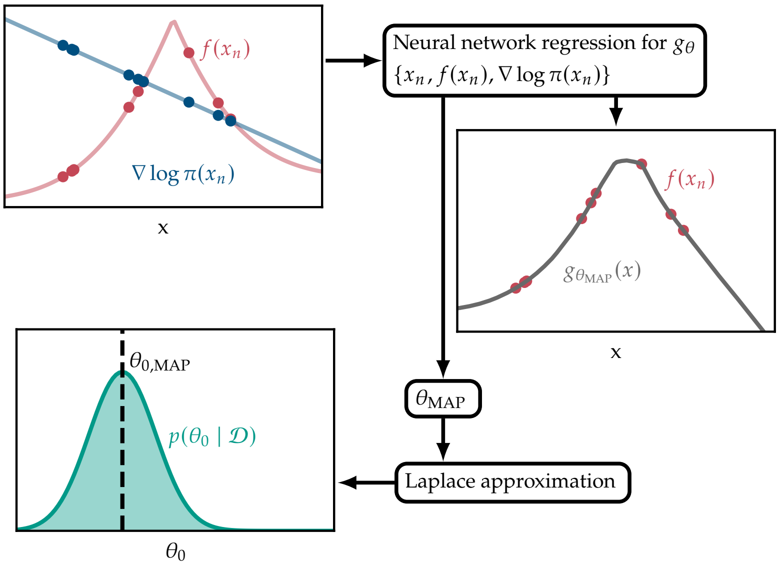

$f(x_n)$
$\nabla \log \pi(x_n)$
x
Neural network regression for g_θ
$\{x_n, f(x_n), \nabla \log \pi(x_n)\}$
$f(x_n)$
$g_{\theta_{\text{MAP}}}(x)$
x
$\theta_{0,\text{MAP}}$
$p(\theta_0 \mid \mathcal{D})$
θ_0
θ_{MAP}
Laplace approximation

Figure 7.2: *Visualization of BSNs.* The BSN prior is conditioned on $\{x_n, f(x_n), \nabla \log \pi(x_n)\}_{n=1}^N$ to obtain a Bayesian posterior on θ_0. This posterior quantifies our uncertainty about $\Pi[f]$. For computational reasons, this posterior is approximated the Laplace approximation around the MAP estimate $\theta_{0,\text{MAP}}$.

integration method would be circular and re-introduce the specter of high computational cost [100]. Other popular approximate inference schemes include variational inference [18, 72, 94] and ensemble methods [118]. Although cheaper, the cost associated with this can still be significant.

We instead opt for the arguably most lightweight approach available for BNNs: the Laplace approximation [133, 178]. It is a simple and computationally cheap method, but yet provides competitive uncertainty estimates [39]. The Laplace approximation constructs a second-order Taylor approximation around the mode of the posterior, which amounts to a Gaussian approximate of the posterior around the MAP (maximum-a-posteriori) estimate. This can be criticized from a Bayesian standpoint as the MAP estimate and the posterior mean of the weights do not necessarily coincide. However, the MAP estimate is the quantity that is usually tuned in deep learning and is also cheap as it only has to be computed once.

Of course, *any* Bayesian treatment of neural networks requires a prior $p(\theta)$. The choice is important since the prior encodes the model class, but there is currently no widely accepted choice. Our choice is motivated by the fact that for the Laplace approximation, only isotropic Gaussian priors are currently feasible [39, 133, 178]. Fortuin et al. [57] suggest that such priors are undesirable, but Wilson and Izmailov [209] argue to the contrary: despite their simplicity, such priors still induce sufficiently complex distributions over functions.

To be more precise, our approximation of the posterior is implemented in two steps: a Laplace approximation, and an approximation of the corresponding Hessian. For the first step, we consider a BSN g_θ trained to minimize the mean squared error loss with weight decay regularizer. Hence, the minimum of the loss is indeed a MAP estimate, and we can apply the Laplace approximation as in Section 4.3. Our second step consists of a positive definite approximation of the Hessian which we obtain via the GGN approximation [188]. Hence, we can extract an approximation of the posterior on the network's prediction of the integral $\Pi[f]$:

$$q_{\text{GGN-Laplace}}(\theta_0) = \mathcal{N}\left(\theta_0 | \theta_{0,\text{MAP}}, (\Sigma_{\text{GGN}})_{0,0}\right).$$

7.4 Architectural Considerations

Due to their specific architecture, naive attempts to train BSNs can lead to unsatisfactory results. Below, as a key contribution, we provide architectural considerations that we have found to significantly improve the conditioning of the loss and lead to better training.

7.4.1 Choice of Activation Function

We require u_{θ_u} to be continuously differentiable on $\mathcal{X}$, which imposes restrictions on the activation functions of the BSN. A sufficient condition is for these activation functions to be themselves continuously differentiable. This excludes the popular RELU activation functions, but includes the CELU ('Continuously Differentiable Exponential Linear Units' [11]; $\text{CELU}(x) = \max(0, x) + \min(0, \exp(x) - 1)$), its continuous extension. It also includes the tanh ($\tanh(x) = (\exp(x) - \exp(-x))/(\exp(x) + \exp(-x))$), Gaussian ($\text{Gauss}(x) = \exp(-x^2)$), and sigmoid ($\text{sigm}(x) = 1/(1 + \exp(-x))$), TanhShrink ($\text{TanhShrink}(x) = x - \tanh(x)$) activations. We compared activation functions (see Figure 7.4 below) and found the CELU to give marginally superior performance on test problems. Based on its good performance, we use CELU activations for all experiments.

7.4.2 Choice of Optimization Procedure

Optimization for BSNs is challenging due to the unique network architecture. For one, the architecture contains gradients of the Stein layer, which are harder to train than standard activation functions. This is because $\nabla_x \log \pi$ can be arbitrarily complicated depending on π. We find that the training of g_θ with Adam [110] is considerably slower compared with training u_{θ_u} (see Section D.1.1 in the Appendix). We suspect that this is due to the loss landscape of the BSN being more narrow (i.e., having a larger spread in curvature eigenvalue spectrum) than that of u_{θ_u}. A second order method should alleviate this issue. Hence, we train the BSN with L-BFGS (an approximate second order method) and the *Hessian-free* optimizer [134] (a conjugate gradient based second order method). And indeed, (approximate) second order optimization reaches much better performance (for an extended discussion see Appendix Section D.1.1).

We therefore used L-BFGS throughout all subsequent experiments. Such quasi-Newton methods have fallen out of fashion in deep learning because they are not stable to noise. In our experiments, we train on the full data set, so noise is not an issue. We accomplish better (i.e., lower loss) and faster convergence (both in iterations and compute time) with this method compared to gradient descent and its variants. Note that this approach is only feasible for relatively small (in number of weights p) network architectures, as it requires storing the gradient history for the approximate Hessian in memory. When training on the entire data set (i.e. no mini-batching), we observe significant speed-up from using GPUs when n is large ($\approx 10^4$).

7.4.3 Choice of $m(x)$

For most of the experiments we set $m(x) = I_d$, but in general other choices for m are possible. We test a set of different choices ($m(x) = I_d/(||x||_2^2+1)$, $m(x) = I_d/\sqrt{||x||_2^2+1}$, $m(x) = I_d\pi(x)$, $m(x) = \text{diag}(x)$), but find that none of these perform significantly better than $m(x) = I_d$ (see Section D.1.1 for more details).

7.4.4 Choice of Point Set

BSNs can be implemented regardless of the choice of $\{x_n\}_{n=1}^N$, but we expect better performance when $\{x_n\}_{n=1}^N$ cover regions of high probability under π. A simple solution is to use independent samples from π; this will be our default choice. When independent sampling is not possible, we can use MCMC instead, so long as π can be evaluated up to some normalization constant. Alternatives also include grid of points or QMC point sets (see Section D.1.1 in the Appendix for a comparison of different point sets), but these are usually only helpful when $\mathcal{X}$ is a hypercube and Π is uniform. Alternatively, one could also use active learning (see [21, 75] for corresponding approaches for BQ) based on the Laplace approximation of the uncertainty, but this may not perform well for larger d, and we did not explore the idea further.

7.4.5 Stein Architecture for Bounded Domains

The architecture outlined in Section 7.3.1 is only valid on the open integration domain $\mathcal{X} = \mathbb{R}^d$. For bounded $\mathcal{X} \subset \mathbb{R}^d$, it is incorrect because $\Pi[\mathcal{S}_m[u]] = 0$ is not necessarily true. This can be guaranteed by adding a layer before the Stein layer. For example, let $\tilde{u}_{\theta_u}(x) = u_{\theta_u}(x)\delta(x)$, where $\delta(x)$ is a smooth function (so that $\tilde{u}_{\theta_u}$ is continuously differentiable) going to zero on the boundary of $\mathcal{X}$. Then, $\pi(\cdot)\tilde{u}_{\theta_u}(\cdot)$ is zero on the boundary of $\mathcal{X}$, and as a result $\Pi[\mathcal{S}[\tilde{u}_{\theta_u}]] = 0$. When $\mathcal{X} = (a, b) \subset \mathbb{R}$, one such function is given by $\delta(x) = (x-a)(b-x)$, and we will use this example where necessary in our experiments. Beyond bounded $\mathcal{X}$, the architecture can also be adapted to manifold or discrete $\mathcal{X}$; see [10] and [189] respectively.

7.5 Experiments

We consider three main experiments: the Genz functions benchmark, a parameter inference problem for a dynamical system called Goodwin Oscillator, and an example describing the energy output of a wind farm. We compare BSNs to the following approaches:

- Monte Carlo methods. When independent sampling from π can be used (i.e. for the Genz benchmark and the wind farm experiments) we use MC. When this is not possible, we use instead an MCMC method called Metropolis-Adjusted Langevin algorithm [MALA; 181] (see Section 2.3.2).

Table 7.1: *Performance on Genz integral family in $d = 2$.* Mean relative integration error and standard deviation (based on 5 repetitions) using $N = 5120$.

	Mean Absolute Error		
Integrand	**MC**	**BQ**	**BSN**
Continuous	1.59e-03 ± 0.90e-03	1.40e-03 ± 0.09e-03	**1.11e-05 ± 0.55e-05**
Discontinuous	2.69e-02 ± 2.64e-02	1.12e-02 ± 0.50e-02	**2.56e-03 ± 1.94e-03**
Gaussian	1.52e-02 ± 8.85e-03	**1.17e-06 ± 1.11e-06**	1.83e-04 ± 1.35e-04
Corner	1.85e-02 ± 1.85e-02	**2.49e-04 ± 1.53e-04**	6.00e-04 ± 5.39e-04
Oscillatory	2.88e-01 ± 1.75e-01	4.13e-03 ± 0.89e-03	**1.34e-03 ± 0.97e-03**
Product	7.59e-03 ± 4.11e-03	1.82e-04 ± 0.42e-04	**1.42e-04 ± 0.76e-04**

- A BQ implementation based on `emukit` [158], with an RBF covariance function $k(x, y) = \lambda \exp(-\|x - y\|_2^2/l^2)$ for some $l, \lambda > 0$. We use log-likelihood maximization to choose l and set the GP prior mean to 0, as we do not have any prior knowledge about the value of the integral. In Section D.1.1 we conduct an additional experiment using the Matern 1/2 Kernel. However, for this kernel, the posterior mean is only available in $d = 1$.
- A control functional estimator based on Stein's method (Stein-CF) as described in [145] for the experiments on the Genz data set and the Goodwin oscillator. The approach can be thought of as a kernel interpolant alternative to our neural network approach. We use $m(x) = I_d$ and an RBF kernel. We use log-likelihood maximization to set the kernel hyperparameters.

To implement the Laplace approximation, we use `laplace-torch` library [39]. Across all experiments we employ the same fully connected architecture for u_{θ_u}, where each hidden layer has 32 units, and we use 2 hidden layers (see Section D.1.1 in the Appendix for more details).

7.5.1 Genz Benchmark

We first consider the Genz family of integrands [62], available in the `ProbNum` package [207], as a test ground (see Appendix Section D.1.2 for detailed definitions). This benchmark, consisting of six integrands with known integrals, was proposed to highlight the performance of numerical integration methods on challenging tasks including discontinuities, peaks and oscillations. Each integrand has a parameter which can be used to increase the dimensionality d of the domain. We follow the implementation of Si et al. [190], where the test functions are transformed to be supported on $\mathcal{X} = \mathbb{R}^d$ and integrated against a multivariate standard Gaussian distribution Π. Since these functions are very cheap, we do not expect BSN or BQ to be competitive with MC methods in terms of runtime, but we use this experiment to showcase the performance of BSNs for challenging integrands and compare methods for fixed N.

In Table 7.1, we first consider the case $d = 2$ and $N = 5120$. BSN and BQ both outperform MC by several orders of magnitude in terms of mean relative integration error. Notably, BSN is significantly better than BQ for the discontinuous Genz function, indicating that the neural network is able to adapt to rapidly changing functions. For the Gaussian Genz function, BQ outperforms the BSN due to the fact that the prior is more informative. Both methods lead to a significant improvement over MC,

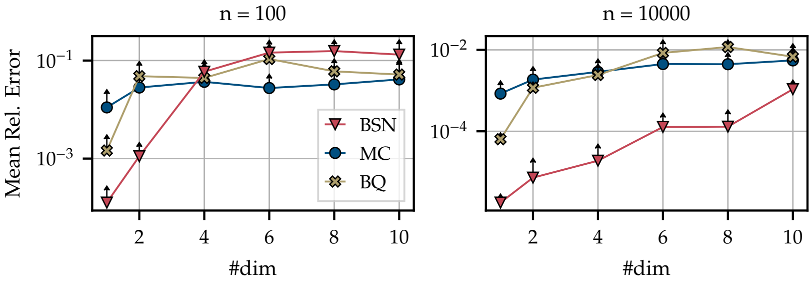


Figure 7.3: *Continuous Genz function*. We compare methods as a function of d for $N = 100$ (*left*) and $N = 10000$ (*right*)(based on 5 repetitions).

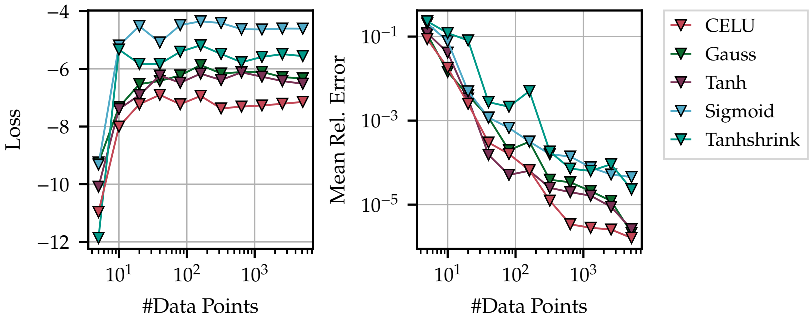


Figure 7.4: *Impact of the choice of activation function for the Continuous Genz function*. Loss l (*left*) and mean relative integration error (*right*) (mean based on 5 repetitions) as a function of N.

but we can run the BSN at higher number of data points N than BQ. See Section D.1.2 in the Appendix for detailed figures.

We then considered the impact of dimensionality on MC, BQ, and BSN in Figure 7.3. We focus on the Continuous Genz function for simplicity. If too few evaluations N are available, the Stein network cannot approximate f well, but with a sufficiently large N (i.e. $N \approx 10^2$ in $d = 1$ and $N \approx 10^4$ in $d = 10$), BSN significantly outperforms MC and BQ.

We also considered the impact of the choice of activation functions for u_{θ_u} in Figure 7.4. Again, we focus on the Continuous Genz integrand, but limit ourselves to $d = 1$. We consider a diverse set of activation functions (described in Section 7.4), all continuously differentiable as required for the final Stein layer. We find that the CELU activation leads to the best results on the Continuous Genz dataset, but other activation functions like the tanh and Gaussian activations also perform well.

Finally, we have a deeper look at the Continuous Genz function in $d = 20$ in Figure 7.5. We observe that a large enough N ($N \approx 10^4$) is necessary for the interpolation capabilities of the model to significantly improve performance. In those cases, the BSN achieves significantly better performance than MC-sampling. We note that MC sampling is cheap on the Genz benchmark dataset, and this benchmark is only used as a test bed to vary the complexity of our integrands, so we only compare the MC method to the other methods in terms of sample efficiency. Both BQ and Stein-CF do not achieve good performance and are too expensive (in runtime and in memory) to run for large N. The BSN can perform well even for much larger data sets (we ran it up to $N \approx 10^6$).

To evaluate the uncertainty estimates provided by the GGN-Laplace approach, we calculate their calibration γ. The calibration is given by the ratio between relative integration error e_{abs}, and the standard deviation σ_{θ_0} of the GGN-Laplace approximation of the posterior on θ_0: $\gamma = e_{\text{abs}}/\sigma_{\theta_0}$. Similarly, for BQ, σ_{θ_0} is the posterior standard deviation on $\Pi[f]$. A calibration fluctuating around one indicates a well calibrated model,

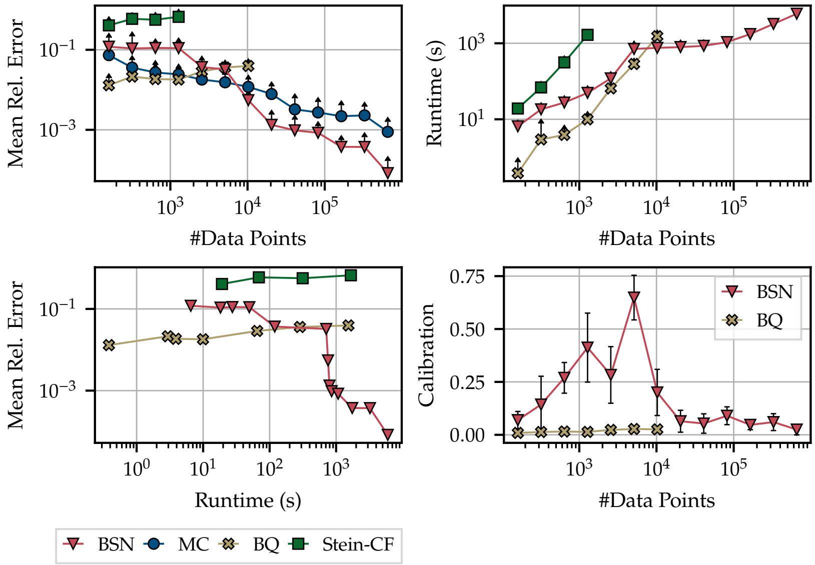


Figure 7.5: *Continuous Genz function in* $d = 20$. Mean relative integration error (*top-left*), run time (*top-right*), and calibration (*bottom-right*) (mean and standard deviation based on 5 repetitions) as a function of N. *Bottom-left:* Mean relative integration error as a function of run time in seconds.

and a large calibration suggests a model that is overconfident, rendering its uncertainty estimates unreliable. The GGN-Laplace approach as well as BQ lead to uncertainty estimates which are underconfident (although less so for the BSN), especially in the high data regime (see Figure 7.5). Underconfident predictions are still useful in that they provide a prudent assessment of our uncertainty.

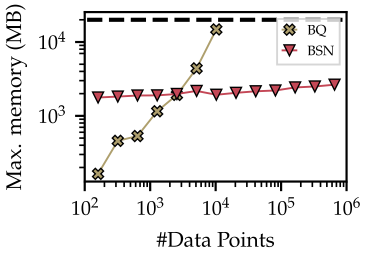


Figure 7.6: Memory requirements of BQ and BSN on the continuous Genz data set in $d = 20$ (based on 5 repetitions).

We can compare the BSN and BQ not only in runtime but also in terms of memory requirements. However, computing accurate memory requirements in python can be difficult as common python libraries use for example `C++` backends. The memory requirements of these non-python backends is commonly not taken into account using the built-in memory profiler. So instead, we use the profiler of our cluster, which outputs the maximum memory required by the program. Figure 7.6 shows that the BSN memory requirements increase more slowly than for BQ. The kernel based methods (BQ and CF) both surpass our allotted memory limit of 20 GB (see Figure 7.6).

7.5.2 Bayesian Inference for the Goodwin Oscillator

A challenging computational task in Bayesian inference is posterior inference for parameters of dynamical systems (see for example [25]). The challenge is due to the large computational cost of posterior sampling, which is incurred due to the need to solve systems of differential equations numerically at a high-level of accuracy. In addition, large data sets can further increase the computational cost, making the task a prime candidate for BSNs. For this experiment, we consider parameter inference in a dynamical system called the Goodwin oscillator [68]. This model describes how the feedback loop between mRNA transcription and protein expression can lead to oscillatory dynamics in a cell. It is a common benchmark for MC methods [24, 148, 176].

We analyze the setting with no intermediate protein species, leading to a system with $d = 4$ parameters: $x = (a_1, a_2, k, \alpha) \in \mathbb{R}^4_+$. Given a posterior distribution π, we want to compute the posterior mean $\Pi[f]$ of each of the ODE parameters, i.e., $f(x) = x$. For this experiment, the posterior

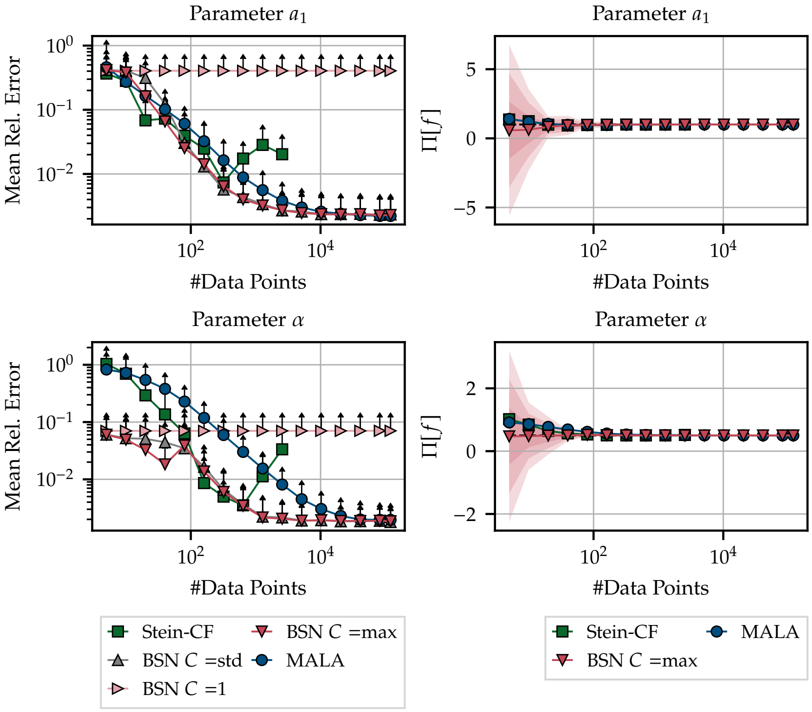


Figure 7.7: *Posterior expectations for the parameters of a Goodwin ODE.* Mean absolute integration error (*top-left* and *bottom-left*), and uncertainty estimates (*top-right* and *bottom-right*) (based on 5 repetitions) as a function of N.

distribution is conditioned on a synthetic data set of 2400 observations generated for some known parameter values. Our exact experimental setup is based on [30], and we refer to the Section D.1.3 in the Appendix for more details.

The posterior density π is only available in unnormalized form, and we therefore use MALA for sampling. This is relatively expensive: sampling $n = 1000$ realizations takes around 30 seconds, which is on the same timescale as network inference ($\sim$ 1 min). For ODE problems requiring more complex solvers or settings with a large data set, the sampling time might increase even further.

In this setting, $\nabla_x \log \pi(x)$ can take very large values, which makes training the BSN harder. We find that $m(x) = I_d/C$ for $C \in \mathbb{R}$ can considerably improve the performance. We considered two choices for the constant C:

- Using the standard deviation of $\{\nabla_x \log \pi(x_n)\}_{n=1}^N$ (called $C =$ std in Figure 7.7).
- Using the largest value of $\nabla_x \log \pi$ across the data set: $C = \max_{n=1,\ldots,N} \nabla_x \log \pi(x_n)$ (called $C =$ max in Figure 7.7).

Figure 7.7 compares the performance of the proposed regularizations. Both choices work well, in contrast to using no regularization at all (i.e. $C = 1$). We find that the BSN either matches the performance of MALA (for parameter α) or surpasses it (parameter a_1). The Stein-CF performs well but struggles in the high data regime due to unstable hyperparameter optimization. The results for a_2 and k are presented in Section D.1.3 in the Appendix. The saturation in reached accuracy for both the BSN and MALA can be attributed to the noisy likelihood evaluations. Before concluding, we emphasize that BSN is the only available Bayesian PNM here. This is because π is unnormalized and BQ is therefore not possible to implement.

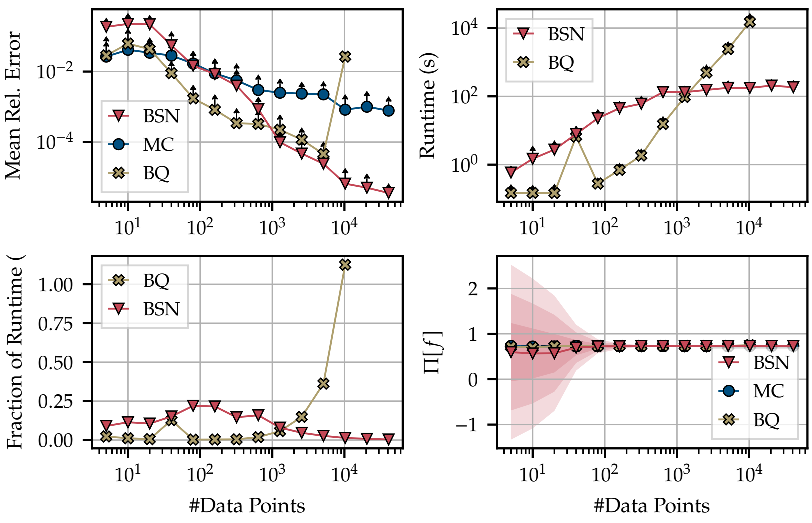


Figure 7.8: *Wind farm model*. Mean relative integration error (*top left*), and run time (*top-right*) (mean and standard deviation based on 5 repetitions) as a function of N. *Bottom-left:* Fraction of runtime BSN and BQ contribute to the total runtime which includes the runtime of the wind farm simulation. *Bottom-right:* Uncertainty estimates provided by the Laplace approximation.

7.5.3 Expected Local Turbine Thrust Coefficient for Wind Farm Layout Design

The energy produced by a wind farm depends on factors including the distance between turbines, the direction of the wind, and the wake produced by individual turbines. To understand this phenomenon, fluid dynamic simulations can be used to estimate a local turbine thrust coefficient (which we denote f), which largely determines energy production [142]. Since a number of these factors are unknown, it is common practice to represent uncertainty through a distribution (denoted π), and calculate the *expected* local turbine thrust coefficient $\Pi[f]$.

A particular challenge here is the cost of evaluating f. For the model we are using (a low-order wake model from [141]), each evaluation of f takes approximately 130 seconds, but more accurate models [112] can take up to tens of hours per evaluation. However, it is well known that f is a smooth function of the inputs, which makes Bayesian PNMs, such as BSNs, prime candidates for the task.

The input to our model f are the wind direction, the turbulence intensity, as well as a number of parameters representing the design of the wind farm (including parameters impacting the distance between turbines, and turbine-specific parameters such as the turbine resistance coefficient, the turbine hub height and diameter, and parameters describe the turbine wake). The distribution π consists of independent distributions (either mixtures of Gaussians, or a truncated Gaussian) on each input to the wake model. Section D.1.4 in the Appendix provides full details on the wind farm data set.

The results are presented in Figure 7.8. Since the ground truth is unknown for this problem, we ran BSN on a data set which is 5 times larger than what is plotted in order to get a benchmark value. We compared the runtime of all methods including sampling, where we assume that all the points were sampled sequentially (corresponding to running the experiment on a single CPU). The additional runtime of both BQ and the BSN is negligible compared to the initial sampling time. Both methods achieve a much lower mean relative integration error compared to sampling, clearly demonstrating the power of Bayesian PNM methods for problems involving expensive integrands.

On this data set BQ cannot be used to compute uncertainty estimates, because we cannot integrate the kernel twice in closed form for truncated Gaussians. However, the uncertainty estimates computed with the Laplace approximation for the BSN accurately capture deviations from the ground truth value (shown in Figure 7.8).

7.6 Limitations and Discussion

The primary advantage of BSNs is in terms of scalability, but they also suffer from some limitations, discussed below.

Firstly, in contrast to GPs where prior knowledge (such as periodicity or smoothness) about f can be encoded via a kernel, selecting good functional priors for BNNs can be challenging. Our experiments show that simple prior choices are often sufficient to achieve good results for moderately hard problems. More advanced options [160, 197] could be considered, but this would require novel Laplace approximations.

Secondly, our experiments suggest convergence with large N. Although we did not analyse this convergence from a theoretical viewpoint, we note that Si et al. [190, Proposition 1 and 2] can be used to prove consistency of the BSN posterior mean to the true value of the integral. Currently, we do not have any results for the convergence rates, but this could be an interesting direction for future research (for example, Belomestny et al. [14] provides a rate for a related approach). This is in contrast with the GP case where convergence results are highly developed [103–105, 210].

Thirdly, computational cost is highly dependent on the complexity of the deep network u_{θ_u}. Across all our experiments we used the same architecture for u_{θ_u} independent of N. We expect that the complexity of the network will need to increase significantly when high accuracy is required and large N is available. In such cases, we expect that minibatching and first order optimization could improve *scalability*, but would likely incur new issues with *stability*.

7.7 Conclusion

We have introduced a way to leverage the function approximation abilities of deep BNNs specifically for integration through the application of a Stein operator. Employing a Laplace approximation provides uncertainty quantification of good quality in this architecture. We have noted that significant work is required to stabilize the training process to this end: both the architecture and the training method must be adapted to the non-standard form of the loss.

BSNs perform consistently well across experiments, both in accuracy and in runtime, and are thus an interesting alternative to BQ, especially for the intermediate regime between very small sample size (where traditional BQ works well), and very large sample numbers (where classic MC methods continue to be the preferred solution). Our experiments on a variety of applications also highlight some functional strengths of the BSN approach. In particular, it can deal flexibly with a wide range of integration densities, including cases in which the density is known in unnormalized form.

Part III.

Conclusion & Future Directions

Conclusion & Future Directions 8.

The first part of this chapter summarizes the scientific results of the previous chapters. In the second part then takes a look at remaining challenges and potential ideas how to solve them.

8.1 Summary & Impact

Although the basic building blocks are quite simple, neural network architectures are often complex. Applications require knowledge of the data, the data generating process or the mathematical problem to be encoded in the model. This adds complexity, but also brings clear benefits: Depending on the modelling choices made, the model may be more data efficient, extrapolate better, or fulfil certain mathematical or physical properties. In this thesis we have considered two important tasks, the use of ODEs to model data and the numerical computation of integrals. Throughout this thesis our aim has been to develop and improve these models and to understand them in more detail.

Understanding Neural ODEs Chapter 5 and Chapter 6 took a closer look at neural ODEs. The mathematical description of ODEs may suggest that the resulting neural ODE describes a time-continuous dynamical system. In reality, numerical solvers are required to implement such models, which always leads to a discrete representation. During training, the neural network may strongly overfit to the numerical solver, especially if the chosen discretization is coarse. When training with very large step sizes, the resulting trajectories may cross in phase space, leading to a breakdown of the continuous view. To check whether the model adapts to a specific solver, one should always test the model with a numerically more accurate solver. Chapter 5 proposes a novel algorithm for training neural ODEs to maintain the properties of continuous ODE solutions. This algorithm involves automatically adjusting the step size or tolerance of the numerical method during training.

Neural ODE models can be augmented with additional structure, e.g. by incorporating known physical models or by directly implementing the Hamiltonian equations of motion. Adherence to conservation laws (as in the case of Hamiltonian neural ODEs) or partial knowledge of the true physics might suggest that these models have superior interpolation and extrapolation capabilities. However, this is not always true in practice, and in order to have a good understanding of model failure, Chapter 6 introduced a method for adding uncertainty estimates to the model. These uncertainty estimates are computed using the Laplace approximation. The resulting uncertainty estimates improve the understanding of the model and inform practitioners in case of model failure.

Neural Networks for Numerical Integration Chapter 7 proposed a novel neural network architecture for numerical integration. The use of neural networks has two advantages: neural networks tend to scale well to large amounts of data, and they scale well to higher dimensions. However, in order to use neural networks for integration, a closed-form solution of the integral of the neural network must be available. Chapter 7 ensures the availability of a closed-form solution by making the Langevin Stein operator part of the network architecture. Since the approximation of the integral can be very inaccurate for few evaluation points, the proposed method includes uncertainty estimates via the Laplace approximation. The choice of activation function, the exact model architecture and the choice of optimizer are essential for the model to work well. The resulting model is particularly useful for higher dimensional problems (~ 10 dimensional) where the data set is too large for BQ to be used effectively, but data acquisition is too computationally expensive for MC-sampling.

8.2 Current Challenges & Future Directions

This section identifies several challenges in the research of task-specific architectures: Improving the training process and the implementation of the models, better understanding of the network's outputs, and the application to real world problems.

Improve Training All models described in this thesis, the neural ODE models and Hamiltonian ODEs in Chapter 5 and Chapter 6 or the Bayesian Stein Network in Chapter 7, are hard to train, especially compared to training a standard fully connected neural network with RELU activations. This is not an issue of their large size, as all architectures considered are relatively small (several 1000 parameters).

Optimization process: Training neural ODEs on data from a long trajectory can lead to collapse of the model to the mean function. To avoid this problem, two approaches have been proposed in the literature. One is to sequentially increase the length of the data set, first feeding only the first few points into the model and then increasing the data set after a certain number of iterations. This approach introduces additional hyperparameters and requires the choice of how to partition the data set and the choice of iterations at which to increase the data set further. Another approach is to use multiple shooting, i.e. to start the model from multiple points in time and possibly for shorter trajectories. However, this requires (almost) noiseless data. Other approaches include gradient matching (which may introduce scaling issues when fitting the GP/splines) or using controlled neural ODEs [109]. Other options might include using a different loss function or considering probabilistic numerical solvers as in Tronarp et al. [199] for parameter inference of ODEs.

Choice of optimizer: The experiments with the BSN showed that this specific architecture benefits significantly more from training with approximate second order optimization than a standard RELU architecture. In fact, it seems necessary to avoid (stochastic) first-order descent to achieve good results. A similar choice is made for PINNs [171], as they often operate on the entire data set as well. To make the BSN more applicable in practice,

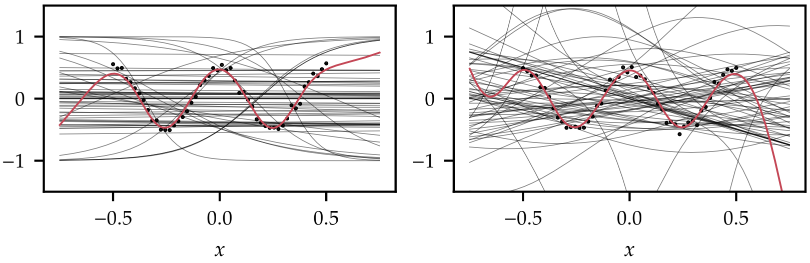


Figure 8.1: Effective activation functions for a standard tanh network (*left*) and a BSN (*right*). Red line indicates the output of the network, black lines the individual activation functions.

especially for large scale problems where training on the full data set is no longer feasible, it might be important to understand the issues related to first order optimization. A possible research direction could be the development of good preconditioners for such architectures.

Effective activation functions: One avenue towards understanding why training such architectures is hard is by visualizing the effective activation functions. Here, we introduce effective activation functions as a concept to potentially understand why training is hard. Effective activation function visualize the functions that are the output of the second to last layer after applying the task specific transformations. For BSNs we can write the network architecture as $f_\theta = \mathcal{T}[u_\theta]$, where $\mathcal{T}$ is an affine operator corresponding to the Stein operator. The network u_θ is a composition of layers, i.e., $u_\theta = u^L_{\theta_L} \circ \cdots \circ u^1_{\theta_1}$. To visualize the effective activations we consider the outputs of $\mathcal{T}[u^{L-1}_{\theta_{L-1}} \circ \cdots \circ u^1_{\theta_1}]$. For a two layer tanh network, the outputs correspond to shifted tanh functions. Figure 8.1 shows the effective activation functions for a linear three layer tanh-network, a BSN with tanh activations. The effective activations for the BSN are more complex than for the simple network, due to their special structure. In certain cases, these functions can have (multiple) minima and maxima, making optimization harder. This issue was for example also observed for periodic activation functions [231].

Being hard to train is not only an issue of the more niche architectures considered herein, but also of standard large scale models like generative adversarial networks [67] or transformers [201], e.g., see Arjovsky et al. [5], Bau et al. [12], and Liu et al. [125] for a discussion of model training challenges. Hence, improving understanding of why models are hard to train is not only important for the models discussed in this thesis, but also for modern deep learning models in general.

Better Implementation Both BSNs and Hamiltonian neural ODEs include derivatives as part of the network architectures. These derivatives must be computed for each forward pass, making the network much slower than standard architectures. In addition, computing the Jacobian of the network with respect to the weights, necessary for the Laplace approximation, is computationally very expensive, since convenient tools such as `backpack` [36] do not work for these non-standard architectures. One way to mitigate this problem is to consider a software stack other than `PyTorch` [159], i.e., `JAX` [20] or `Julia` [17]. The latter may be interesting for neural ODEs, as it provides a large library of numerical solvers.

Opening the Black-box Neural networks and their predictions can be difficult to understand - adding structure seems to improve this problem, but also adds complexity to the model.

Uncertainty estimates Good uncertainty estimates are critical for detecting model failure in a timely manner. In this thesis we have used the Laplace approximation to obtain uncertainty estimates. Although the results were already good, the speed could be further improved by using a better implementation and possibly considering only a substructure of the model [40, 116].

Choice of prior Both the choice of prior on the weights and the choice of network architecture define the functional prior of Bayesian neural networks. The current default choice for the prior on the network weights is an isotropic Gaussian. This prior seems sufficient to model a large class of functions [209] and is therefore a good choice when little information is available. However, defining a more task-specific prior remains a challenge for BNNs. In addition, it is unclear how to apply non-Gaussian priors in a computationally efficient way, since the Laplace approximation no longer works out of the box.

Choice of architecture On the architecture side, the choice of activation function impacts the interpolation and extrapolation behavior of the network. RELU networks model piece-wise linear functions and extrapolate linearly far away from the data. In cases where no additional information about the data is known, this may be sufficient, but in cases where some knowledge about the data set exists, it may be necessary to encode this knowledge in the prior. GPs allow knowledge about the function class to be encoded by the choice of kernel, e.g. smoothness of the function via the Matern kernel or periodicity via a periodic kernel. Similar settings might also be interesting for neural networks, potentially leading to more accurate and more interpretable results.

Application to Real World Problems Neural ODEs and BSNs are motivated by real world applications. Neural ODEs are motivated by the fact that many real world processes can be described by differential equations. If a description of the full dynamics is not available, it only makes sense to model unknowns in a data driven way, e.g., using neural networks. Neural ODEs have already been used successfully for various applications, e.g., fluid dynamics [167], molecular dynamics [215], health care [41, 128], and quantum chromodynamics [84]. Integration problems are ubiquitous to science, and BSNs are directly targeting this problem. BSNs are particularly interesting for problem setups where relatively large amounts of data are available, but sampling is not for free. One example might be integrating out uncertainties over parameters in simulations that roughly take on the order of minutes, as illustrated by the wind farm experiment described in Chapter 7. Extending this list of applications is important to test and improve the applicability of both neural ODEs and BSNs. This work extends both of these models with uncertainty estimates provided by the Laplace approximation. Given this (approximate) Bayesian treatment, it might be interesting to include uncertainty estimates from upstream tasks or use the uncertainty estimates obtained via the Laplace approximation for downstream applications. One way to utilize the uncertainty estimates provided by the Laplace approach might be to construct an algorithm for active acquisition of data points.

Part IV.

Appendix

Additional Material for Chapter 3 A.

A.1 List of Differential Equations

In this section we provide a list of all the differential equations used in Chapter 3.

A.1.1 Logistic ODE

The logistic ODE is described by the following differential equation

$$x' = x\,(1-x)\,, \quad x(0) = x_0,$$

with the following analytical solution

$$x(t) = \frac{x_0 e^t}{x_0 e^t + 1 - x_0}.$$

The example in Figure 3.1 uses the following settings:

$$\begin{aligned} [t_0, T] &= [0, 10], \\ x(0) = x_0 &= 0.01, \\ h &= \frac{10}{7}. \end{aligned}$$

A.1.2 Harmonic Oscillator

The harmonic oscillator describes the motion of a particle in a quadratic potential. It can also be used to approximate the motion of a pendulum with a small amplitude. The harmonic oscillator is described by the following second order differential equation:

$$x'' = \omega x$$

where ω is a problem specific constant. For Figure 3.2, the following settings were used:

$$\begin{aligned} \omega &= 1 \\ [t_0, T] &= [0, 3\pi] \\ x(0) = 0, &\quad x'(0) = 1 \\ h &= \frac{\pi}{20} \end{aligned}$$

A.1.3 Van der Pol Equation

The Van der Pol oscillator is a non-conservative oscillator described by the following equation:

$$x'' = \omega(1 - x^2)x' - x,$$

where ω is a problem specific constant. The following parameters where used to create Figure 3.3:

$$
\begin{aligned}
\omega &= 5 \\
[t_0, T] &= [0, 50] \\
x(0) &= 0, \quad x'(0) = 2 \\
rtol &= 1e-3, \quad atol = 1e-6
\end{aligned}
$$

A.1.4 Arenstorf Orbits

The Arenstorf orbits are described by the following set of differential equations

$$
\begin{aligned}
x_1'' &= x_1 + 2x_2' - \mu_2 \frac{x_1 + \mu}{D_1} - \mu_1 \frac{x_1 - \mu_2}{D_2} \\
x_2'' &= x_2 - 2x_1' - \mu_2 \frac{x_2}{D_1} - \mu_1 \frac{x_2}{D_2} \\
D_1 &= \left((x_1 + \mu_1)^2 + x_2^2\right)^{3/2} \\
D_2 &= \left((x_1 - \mu_2)^2 + x_2^2\right)^{3/2}
\end{aligned}
$$

where $\mu_1 = 0.012277471$ and $\mu_2 = 1 - \mu_1$ are the masses of the moon and the earth. x_1, x_2 corresponds to the position of the spaceship. For the right initial conditions there exist periodic orbits, e.g.,

$$
\begin{aligned}
x_1(0) &= 0.994, x_1'(0) = 0, x_2(0) = 0, \\
x_2'(0) &= -2.00158510637908252240537862224, \\
T &= 7.06521656015796255889172062 49
\end{aligned}
$$

where T is the period length.

Additional Material for Chapter 5 B.

B.1 Crossing Trajectories

In the main text we describe how trajectory crossings are an indication that the vector field has adapted to a specific solver and such a vector field then has no meaningful interpretation in the continuous setting. In this section we want to answer whether the following statement is true: If for a step size $\tilde{h}$ we do not observe crossing trajectories then for all $h < \tilde{h}$ we will not observe crossing trajectories. To study this problem in more detail we introduce a vector field $f : [0.0, \infty) \times [0.1, \infty) \rightarrow [0, \infty) \times [0, \infty)$

$$\frac{\mathrm{d}}{\mathrm{d}t}\begin{bmatrix} x & y \end{bmatrix} = f(x, y),$$

$$f(x,y) = \begin{cases} \begin{bmatrix} 1 - 2(x-1) & \frac{2(x-1)}{y} \end{bmatrix}, & \text{if } x \in (1, 1.25] \\ \begin{bmatrix} 0.5 & \frac{0.5}{y} \end{bmatrix}, & \text{if } x \in (1.25, 1.75] \\ \begin{bmatrix} 0.5 + 2(x - 1.75) & \frac{0.5 - 2(x-1.75)}{y} \end{bmatrix}, & \text{if } x \in (1.75, 2] \\ \begin{bmatrix} 1 & 0 \end{bmatrix}, & \text{else.} \end{cases} \tag{B.1}$$

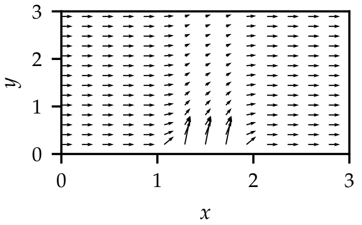


Figure B.1: Visualization of the vector field defined in Equation B.1

We chose this vector field because only for the region $x \in (1, 2]$ the curvature of the vector field changes (see Figure B.1 for a visualization of the vector field). The described vector field is Lipschitz continuous on $[0.0, \infty) \times [0.1, \infty)$, therefore the Picard-Lindelöf theorem applies and the true solutions to the ODE do not cross in phase space.

We use the Euler method with different step sizes to solve the ODE. First we only look at the IVPs where $x(t = 0) = 0$ (shown in blue in Figure B.2). Each IVP can be thought of as a data point in a data set. For the large step size $h = 1$ we do not observe crossing trajectories (see Figure B.2). The numerical solver fails to resolve the change in curvature in the vector field. If we decrease the step size to $h = 1/2$ we observe crossing trajectories (see Figure B.2). This clearly shows that not observing crossing trajectories for some $\tilde{h}$ does not give us any information about what will happen for $h < \tilde{h}$. But since we know that the numerical solvers converge to the true solution, we can make the following statement: For a given set of IVPs there exists a step size h^* such that for all $h < h^*$ we do not observe crossing trajectories. And indeed if we choose an even smaller step size we no longer observe crossing trajectories (see Figure B.2 (c)).

We also want to emphasize that observed behavior for the different step sizes is dependent on the set of IVPs. To illustrate this point we add additional IVPs to our original data set, where for the new IVP $x(t = 0) = 1.4$ (the additional IVPs are shown in orange in Figure B.2). Now we also observe crossing trajectories for the large step size $h = 1$. This shows that whether we observe crossing trajectories or not, is not only dependent on the step size but also on the set of IVPs we choose.

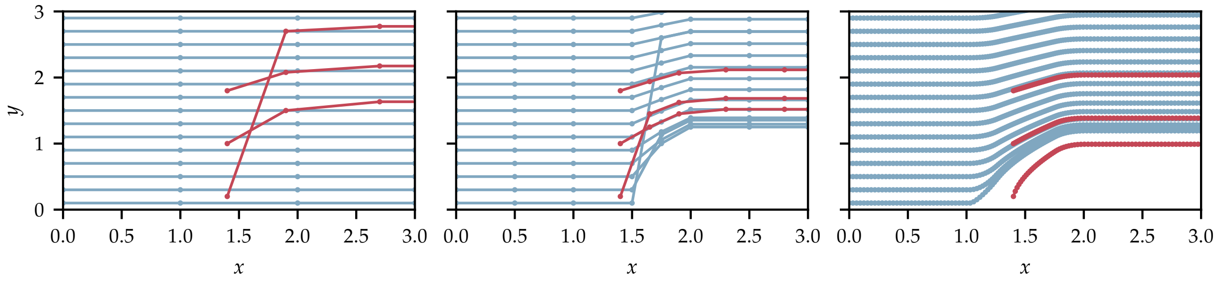


Figure B.2: Numerical solutions to Equation B.1 using the Euler method. In blue is the set of IVPs for which $x(0) = 0$ and in orange is an additional set of IVPs for which $x(t = 0) = 1.5$. The ODE is solved using different step size, $h = 1$ in (*left*), $h = 2^{-1}$ in (*center*), and $h = 2^{-5}$ in (*right*).

B.2 Experimental Results

In the main text we do not plot the results for all the training step sizes/ tolerances but only for every second training step size/tolerance to improve the clarity of the plots. Here we now include the plots showing all training runs, and we include additional results for all data sets.

B.2.1 Results for Fixed Step Solvers

In this section we present the results for fixed step solvers. The model is trained with the Euler method or a 4th order Runge-Kutta method (RK4) with different step sizes. If the Euler method is used for training then the model is tested with the Euler method, the Midpoint method and RK4. If RK4 is used for training then the model is only tested with RK4.

We train neural ODE models on CIFAR10 (see Figure B.3), on MNIST (see Figure B.4), on the 2-dimensional concentric sphere data set (see Figure B.5).

For all data sets we observe that if the model is trained with a large step size then there is a drop in performance if a solver with smaller numerical error is used for testing. But there exists a training step size above which using a solver with a higher numerical accuracy does not lead to a drop in performance. The observations support our claims made in the main text.

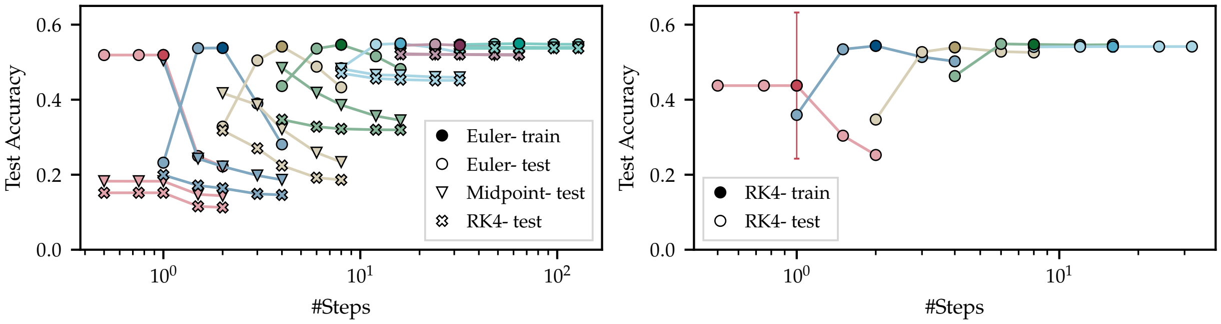


Figure B.3: A neural ODE was trained with different step sizes (plotted in different colors) on CIFAR10. The model was tested with different solvers and different step sizes. In (*left*) the model was trained using the Euler method. Results obtained by using the same solver for training and testing are marked by dark circles. Light data indicated different step sizes used for testing. Circles correspond to the Euler method, cross to the midpoint method and triangles to a 4th order Runge-Kutta method. In (*right*) a 4th order Runge-Kutta methods was used for training (dark circles) and testing (light circles).

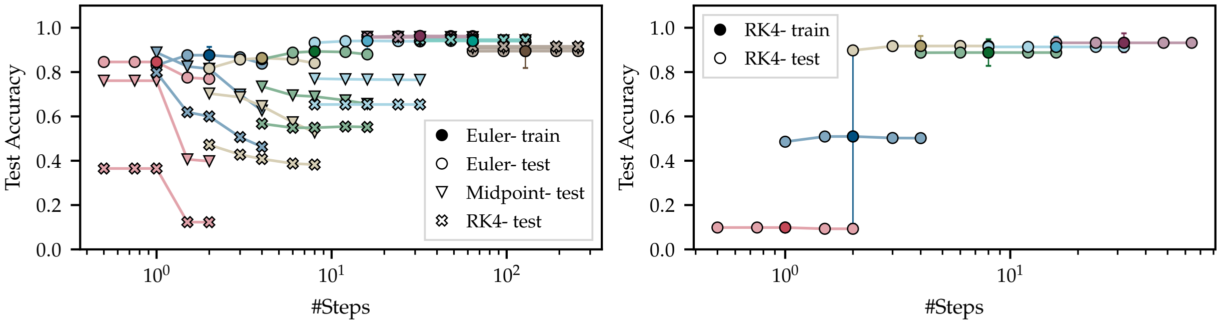


Figure B.4: A neural ODE was trained with different step sizes (plotted in different colors) on MNIST. The model was tested with different solvers and different step sizes. In (*left*) the model was trained using the Euler method. Results obtained by using the same solver for training and testing are marked by dark circles. Light data indicated different step sizes used for testing. Circles correspond to the Euler method, cross to the midpoint method and triangles to a 4th order Runge-Kutta method. In (*right*) a 4th order Runge-Kutta methods was used for training (dark circles) and testing (light circles).

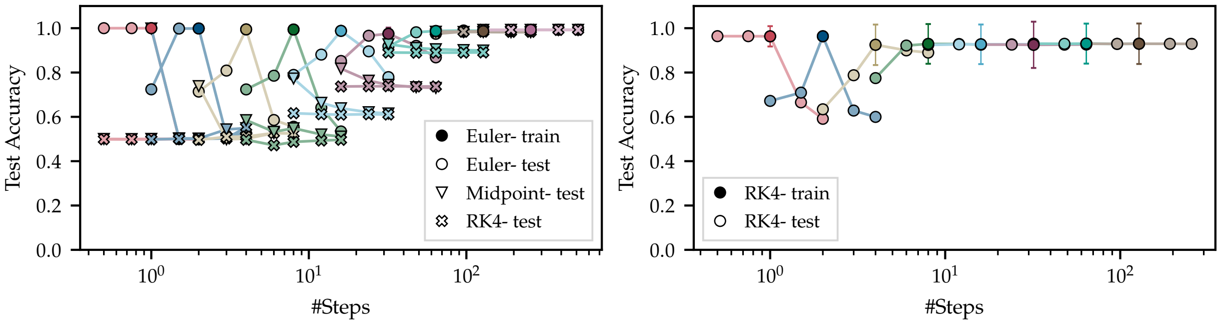


Figure B.5: A neural ODE was trained with different step sizes (plotted in different colors) on the 2-dimensional concentric sphere data set. The model was tested with different solvers and different step sizes. In (*left*) the model was trained using the Euler method. Results obtained by using the same solver for training and testing are marked by dark circles. Light data indicated different step sizes used for testing. Circles correspond to the Euler method, cross to the midpoint method and triangles to a 4th order Runge-Kutta method. In (*right*) a 4th order Runge-Kutta methods was used for training (dark circles) and testing (light circles).

B.2.2 Results for Adaptive Solvers

In this section we present the results for adaptive step size solvers. The model is trained with Fehlberg21 or Dopri54 with different tolerances. If Fehlberg21 is used for training then the model is tested with Fehlberg21 and Dopri54. If Dopri54 is used for training then the model is only tested with Dopri54.

We train the neural ODE models on CIFAR10 (see Figure B.6) on MNIST (see Figure B.7) and on the Sphere2 data set (see Figure B.8). For CIFAR10 we use backpropagation through the numerical solver to calculate the gradients. For Sphere2 we use backpropagation through the numerical solver as well as the adjoint method described in [29] to calculate the gradients.

We observe that the models trained with a relatively large tolerance show a drop in performance if the models are tested with a solver with lower numerical error. We observe this behavior for all data sets and also for the adjoint method (see Figure B.8), the only exception is MNIST trained with Dopri54 which we attribute to the high order of the solver and the low critical step size on MNIST. If the model is trained with a small tolerance then there is no drop in performance if the model is tested with a solver with lower numerical error. We note that the Sphere2 data set trained with Dopri54 (Figure B.8) shows decreasing performance for smaller tolerances. This might be due to the model taking a lot of steps

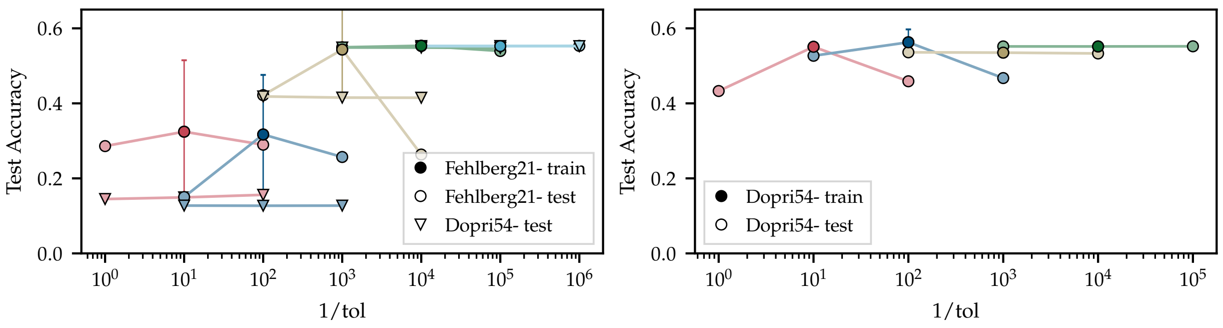


Figure B.6: A neural ODE was trained with different tolerances (plotted in different colors)on CIFAR10. The model was tested with different solvers and different tolerances. In (*left*) the model was trained using the Fehlberg21 method. Results obtained by using the same solver for training and testing are marked by dark circles. Light data indicated different step sizes used for testing. Circles correspond to Fehlberg21 method, cross to the Dopri54 method. In (*right*) Dopri54 was used for training (dark circles) and testing (light circles).

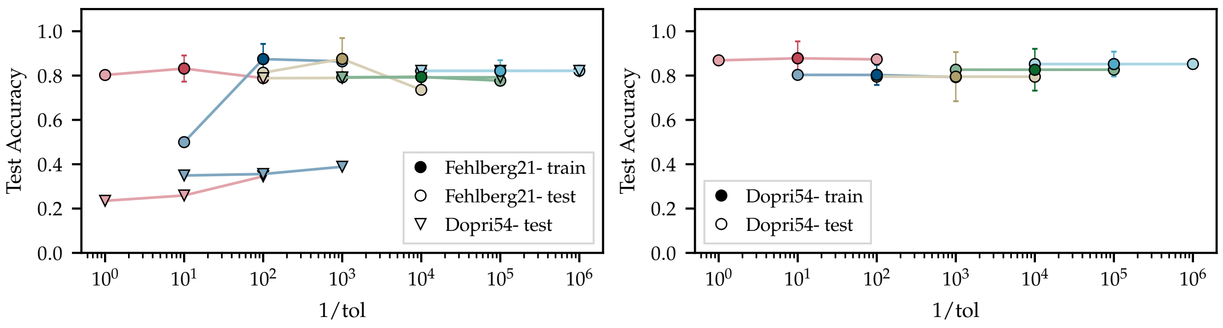


Figure B.7: A neural ODE was trained with different tolerances (plotted in different colors)on MNIST. The model was tested with different solvers and different tolerances. In (*left*) the model was trained using the Fehlberg21 method. Results obtained by using the same solver for training and testing are marked by dark circles. Light data indicated different step sizes used for testing. Circles correspond to Fehlberg21 method, cross to the Dopri54 method. In (*right*) Dopri54 was used for training (dark circles) and testing (light circles).

and therefore, the gradient provided by backpropagation might not be accurate enough. Additionally, tuning the hyperparameters for specific tolerances might improve the performance. For the model trained on Sphere2 using the adjoint method we observe that for large tolerances the model achieves relatively low test accuracy. At large tolerances the model takes relatively large and inaccurate steps. Therefore, the backward solutions of the ODE differs from the forward solve and no valid gradient information is provided to the optimizer. Overall the performance on MNIST is lower than with fixed step solvers, even though we specifically tuned the learning rate and optimizer.

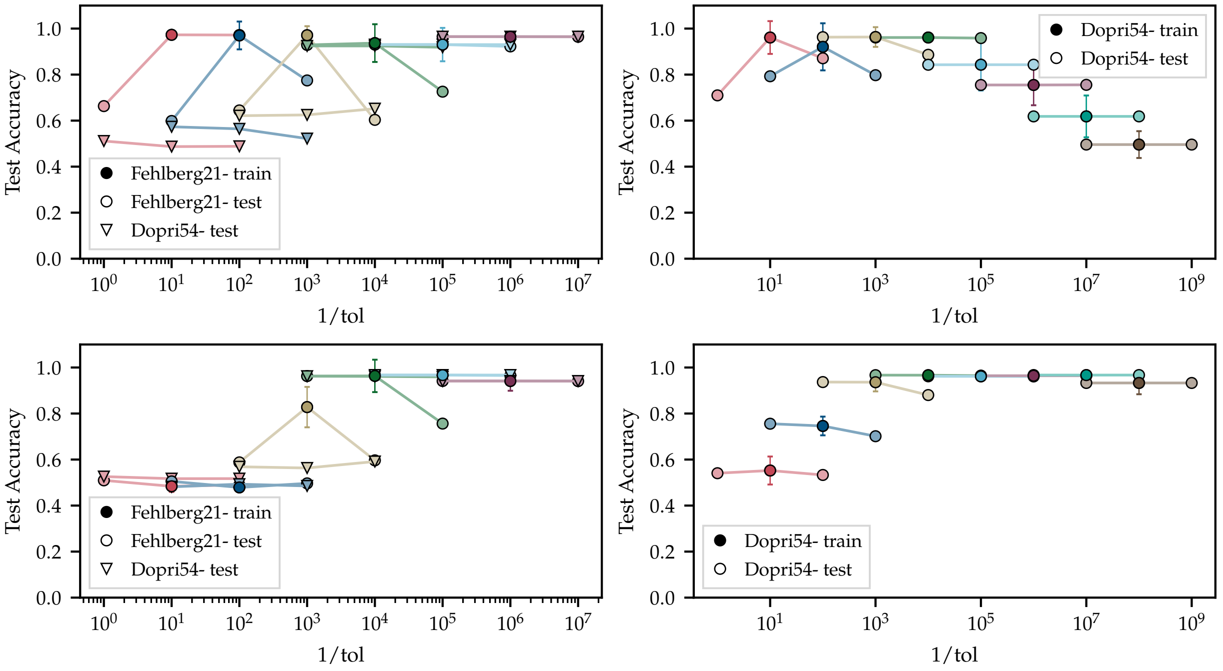


Figure B.8: A neural ODE was trained with different tolerances (plotted in different colors)on Sphere2 (*top*) and on Sphere2 using the adjoint method (*bottom*). The model was tested with different solvers and different tolerances. In (*left-column*) the model was trained using the Fehlberg21 method. Results obtained by using the same solver for training and testing are marked by dark circles. Light data indicated different step sizes used for testing. Circles correspond to Fehlberg21 method, cross to the Dopri54 method. In (*right-column*) Dopri54 was used for training (dark circles) and testing (light circles).

B.3 Step Size and Tolerance Adaptation Algorithm

B.3.1 Step Adaptation Algorithm

Here we describe the full step adaptation algorithm with additional details omitted in the main text for clarity. The algorithm queries whether the suggested step size has surpassed the length of the time interval over which the ODE is integrated. If this is the case, the step size is reduced to the size of the time interval. This check is necessary to avoid infinitely increasing the step size. The algorithm does not increase the step size if accuracy of the model tested with the train solver and the proposed step size is past the threshold. We do not want the model to continue training with a too large step size leading to discrete dynamics.

Additionally, it is important to ensure that the test solver is more accurate than the train solver. For our experiments we use as a combination of train and test solver the following pairs (Euler, Midpoint), (Euler, RK4), (Midpoint, RK4). To ensure that the test solver achieves a smaller numerical error, the test solver reduces the step size for testing if the training step size is too large. In our algorithm we require that the order of the global numerical error of the test solver is a factor 50 smaller than the order of the numerical error of the train solver.

We also tested the proposed step adaptation algorithm on the Sphere2 data set (see Figure B.11), on MNIST (see Figure B.10) and on CIFAR10 (see Figure B.9). The behavior of the algorithm on different data sets as well as with different combinations of train and test solver supports the claims in the main text. It is interesting to note that if Midpoint is used as a train solver, the algorithm fluctuates around a much lower step size than if Euler is used as a train solver. A possible explanation for this behavior is that Midpoint is a second order Runge-Kutta method which has therefore a lower numerical error than Euler which is a first order Runge-Kutta method.

Algorithm 2: Step adaptation algorithm

```
initialize starting step_size $h$ according to [80, p. 169];
while Training do
    batch = draw_batch(data);
    logits = model.do_forward_pass(batch, train_solver($h$));
    loss = model.calculate_loss(logits);
    train_solver_acc = model.calculate_acc(logits);
    if Iteration % 50 == 0 then
        logits = model.do_forward_pass(batch, test_solver($h$));
        test_solver_acc = model.calculate_acc(logits);
        if |train_solver_acc-test_solver_acc| > 0.1 then
            $h\_new = 0.5\ h$;
        else
            $h\_new = 1.1\ h$;
            if $h\_new > T$ then
                // Avoid increasing the step size
                   indefinitely
                $h\_new$ = T;
            end
            logits = model.do_forward_pass(batch,
              train_solver($h\_new$));
            $h\_new$_acc = model.calculate_acc(logits);
            if |test_solver_acc-$h\_new$_acc| > 0.1 then
                $h\_new = h$;
            end
            $h = h\_new$
        end
    end
end
```

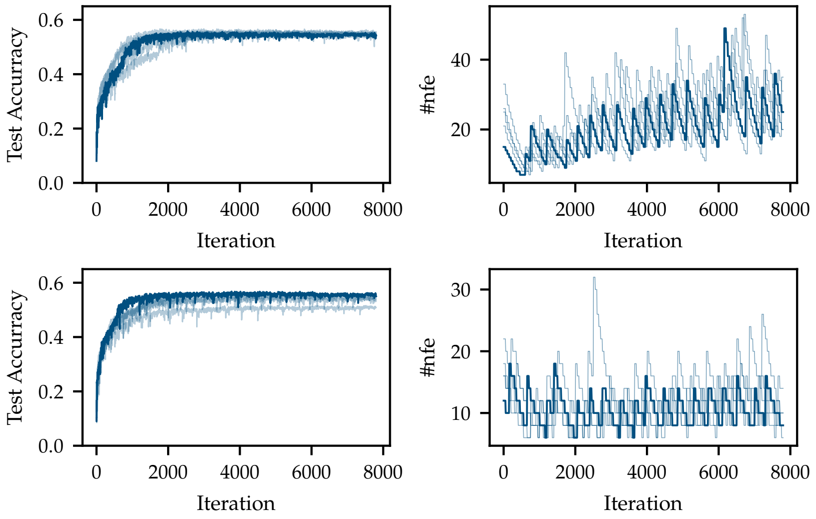


Figure B.9: Using the step adaptation algorithm for training on CIFAR10. (*top*) using Euler as train solver and RK4 as the test solver and (*bottom*) using Midpoint as train solver and RK4 as the test solver. (*Left-column*) show the test accuracy over the course of training for five different seeds. (*Right-column*) show the number of steps chosen by the algorithm over the course of training.

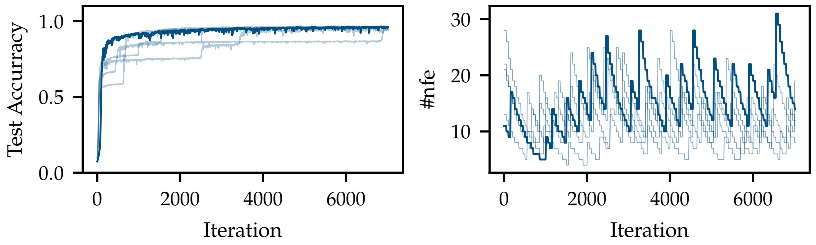


Figure B.10: Using the step adaptation algorithm for training on MNIST. Using Euler as the train solver and Midpoint as the test solver. (*Left*) show the test accuracy over the course of training for five different seeds. (*Righ*) show the number of steps chosen by the algorithm over the course of training.

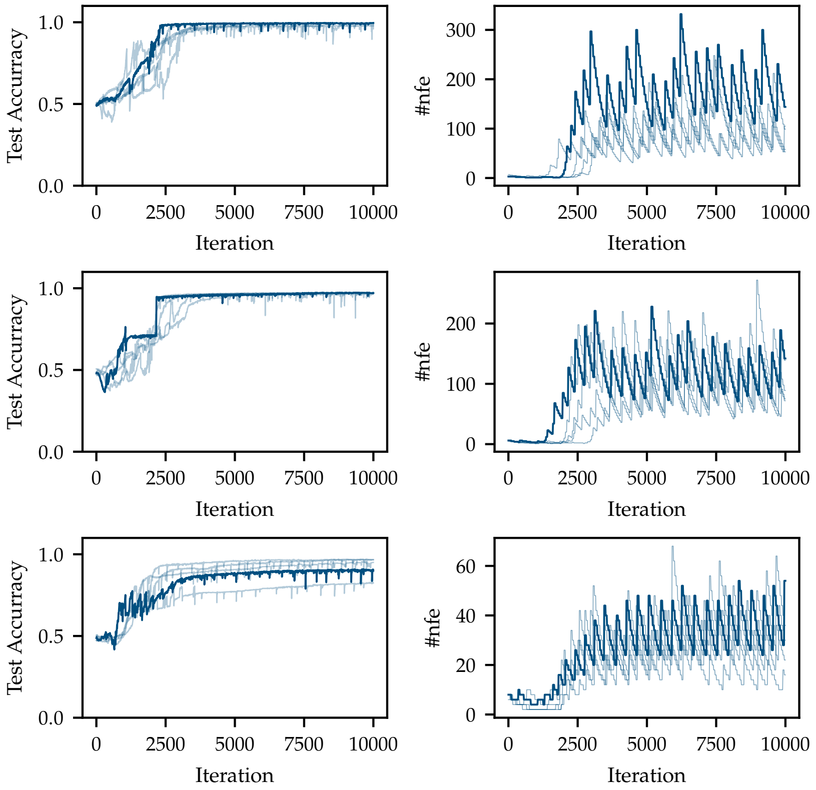


Figure B.11: Using the step adaptation algorithm for training on Sphere2. (*Top*) using Euler as the train solver and Midpoint as the test solver (*center*) using Euler as train solver and RK4 as the test solver and (*bottom*) using Midpoint as train solver and RK4 as the test solver. (*Left-column*) show the test accuracy over the course of training for five different seeds. (*Righ-column*) show the number of steps chosen by the algorithm over the course of training.

B.3.2 Tolerance Adaptation Algorithm

The tolerance adaptation algorithm is similar to the step adaptation algorithm. In order to avoid infinitely increasing the tolerance, the algorithm checks whether the solver uses at least min_{steps} which we set to 2 for our experiments. As for the fixed step solvers, the algorithm checks whether to accept the tolerance if the tolerance was increased.

For our experiments we used Fehlberg21 and Dopri54 as train solver and Dopri54 as test solver. To ensure that the test solver has a smaller numerical error than the train solver the test solver uses a smaller tolerance than the train solver. If Fehlberg21 is used as a train solver, then the tolerance for the test solver Dopri54 is set to 1/5 of the train solver tolerance. If Dopri54 is used as a train solver, then the tolerance for the test solver Dopri54 is set to 1/10 of the train solver tolerance.

The results for the tolerance adaptation algorithm are shown in Figure B.12. After an initial adjustment phase, the number of function evaluations (nfe) and the tolerance fluctuate for the rest of training similar to the number of steps for the step adaptation algorithm. For different seeds, the tolerance fluctuates around different final values.

Algorithm 3: Tolerance adaptation algorithm

```
Inputs train_solver, test_solver, model;
initialize tolerance tol= $10^{-6}$;
while Training do
    batch = draw_batch(data);
    logits = model.do_forward_pass(batch, train_solver(tol));
    loss = model.calculate_loss(logits);
    train_solver_acc = model.calculate_acc(logits);
    if Iteration % 50 == 0 then
        logits = model.do_forward_pass(batch, test_solver(tol));
        test_solver_acc = model.calculate_acc(logits);
        if |train_solver_acc-test_solver_acc| > 0.1 then
            tol_new= 0.5 tol;
        else
            tol_new = 1.1 tol;
            if steps <= min_steps then
                // Avoid increasing the tolerance
                   indefinitely
                tol_new = tol;
            else
                logits = model.do_forward_pass(batch,
                  train_solver(tol_new));
                tol_new_acc = model.calculate_acc(logits);
                if |test_solver_acc-tol_new_acc| > 0.1 then
                    tol_new = tol;
                end
            end
        end
        tol = tol_new
    end
    model.update(loss);
end
```

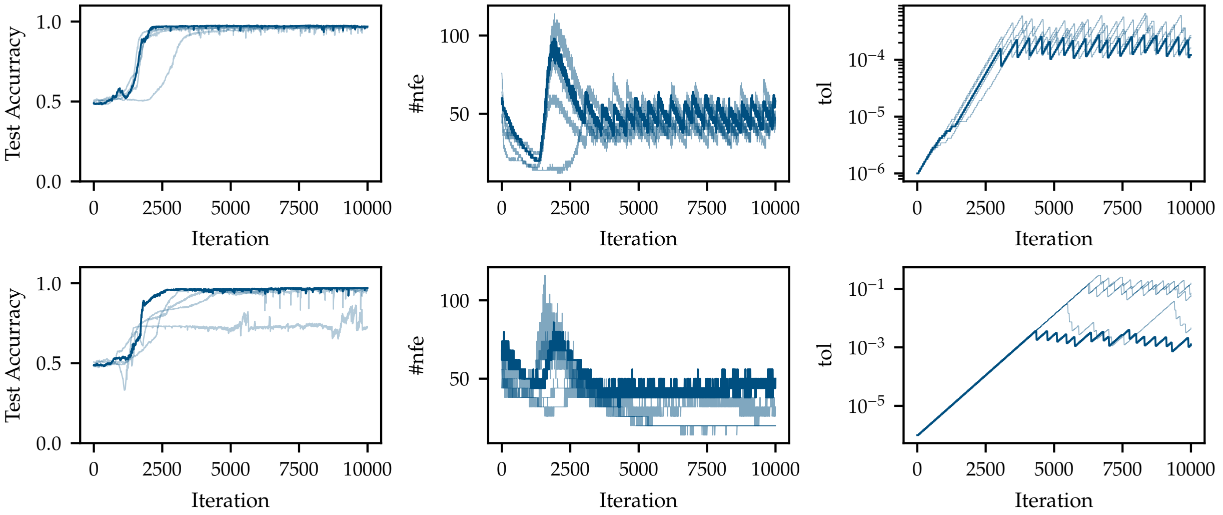


Figure B.12: Using the tolerance adaptation algorithm for training on Sphere2 for Fehlberg21 as train solver, and Dopri54 as test solver (*top*). (*Bottom*) Dopri54 was used as train and test solver where the test solver uses a reduced tolerance. (*Left-column*) shows the test accuracy over the course of training for five different seeds. (*Center-column*) shows the number of function evaluations the course of training. (*Right-column*) show the tolerance chosen by the algorithm over the course of training

B.4 Architecture and Hyper-parameters

We chose the architecture for our network similar to the architecture proposed by [50]. We tried to find hyperparameters which worked well for all step sizes. The same hyperparameters were used for the grid search and for training with the step adaptation algorithm:

B.4.1 Architecture and Hyper-parameters Used for MNIST

Neural ODE Block

- Conv2D(1, 96, Kernel 1x1, padding 0) + RELU
- Conv2D(96, 96, Kernel 3x3, padding 1) + RELU
- Conv2D(96, 1, Kernel 1x1, padding 0)

Classifier

- Flatten + LinearLayer(784,10) + SoftMax

Hyper-parameters

- Batch size: 256
- Optimizer: SGD (fixed step solvers), Adam (adaptive step size solvers)
- Learning rate: 1e-2 (fixed step solvers), 1e-4 (adaptive step size solvers)
- Iterations used for training: 7020

B.4.2 Architecture and Hyper-parameters Used for CIFAR10

Neural ODE Block

- Conv2D(3, 128, Kernel 1x1, padding 0) + RELU
- Conv2D(128, 128, Kernel 3x3, padding 1) + RELU
- Conv2D(128, 3, Kernel 1x1, padding 0)

Classifier

- Flatten + LinearLayer(3072,10) + SoftMax

Hyper-parameters

- Batch size: 256
- Optimizer: Adam
- Learning rate: 1e-3
- Iterations used for training: 7800

B.4.3 Architecture Used for Concentric Sphere 2D Dataset

Neural ODE Block

- Conv1D(1, 32, Kernel 1x1, padding 0) + RELU
- Conv1D(32, 32, Kernel 3x3, padding 1) + RELU
- Conv1D(32, 1, Kernel 1x1, padding 0)

Classifier

- Flatten + LinearLayer(2,2) + SoftMax

Hyper-parameters

- Batch size: 128
- Optimizer: Adam
- Learning rate: 1e-4
- Iterations used for training: 10000

B.4.4 Python Packages

In our code we make use of the following packages: Matplotlib [98], Numpy [83], Pytorch [159] and Torchdiffeq [29].

Additional Material for Chapter 6 C.

C.1 Datasets

C.1.1 Lotka-Volterra

As one of the standard data sets to discuss ODEs, we use the Lotka-Volterra system to compare the Laplace approximation to other models. The Lotka-Volterra system models the interaction between a species of predator y and prey x. The system is described by the following differential equations

$$\frac{dx}{dt} = \alpha x - \beta xy, \frac{dy}{dt} = \delta xy - \gamma y, \tag{C.1}$$

where $\alpha, \beta, \gamma, \delta$ are positive real parameters describing the interactions between predators and prey. For our experiment we chose $\alpha = 2/3, \beta = 4/3, \delta = 1, \gamma = 1$. We add zero mean Gaussian noise with variance $\sigma^2 = 0.03$.

For *data-set-half-cycle* training data is generated on the interval $t \in [0, 5]$. For *data-set-full-cycle* training data is generated on the interval $t \in [0, 10]$. For both data sets we set $x_0 = (1, 1)$.

C.1.2 Harmonic Oscillator

The equations of motion for a harmonic oscillator, for a particle with position q and momentum p, are given by

$$\dot{q} := \frac{dq}{dt} = \frac{p}{m}, \quad \dot{p} := \frac{dp}{dt} = -kx. \tag{C.2}$$

k is the spring constant of the system, m is the mass of the particle. For the experiments we use three different data sets generated from the harmonic oscillator. All data sets consist of 16 trajectories with different initial conditions and each trajectory consists of 50 samples. We add zero mean Gaussian noise with variance $\sigma^2 = 0.3$ to each data set.

Data Set Lower Half We set $m = 1$, $k = 2$ in Equation C.2. The initial conditions q_0 and p_0 are each sampled from a Gaussian distribution with mean $\mu_q = 3$, $\mu_p = 0$ and variance $\sigma_q^2 = \sigma_p^2 = 0.2$. Training data is generated on the interval $t \in [0, \pi/\sqrt{k}]$.

Data Set Left Half We set $m = 1$, $k = 2$ in Equation C.2. The initial conditions q_0 and p_0 are each sampled from a Gaussian distribution with mean $\mu_q = 0$, $\mu_p = 3\sqrt{k}$ and variance $\sigma_q^2 = \sigma_p^2 = 0.2$. Training data is generated on the interval $t \in [0, \pi/\sqrt{k}]$.

C.1.3 Pendulum

The damped pendulum is described by

$$\frac{d^2\phi}{dt^2} + \omega^2 \sin\phi + \alpha\frac{d\phi}{dt} = 0 \tag{C.3}$$

where ϕ is the angle, ω frequency and α a damping constant. For our data set we set $\omega = 2\pi/12$ and $\alpha = 0.1$. We choose $t \in [0, 10]$ and add zero mean Gaussian noise with variance $\sigma^2 = 0.1$ to each data set. The initial conditions are the same as in Yin et al. [220], i.e., $\phi_0 \leftarrow r y_0/||y_0||_2$ where $y_0 \sim U_{[0,1]\times[0,1]}$ and $r = 1.3 + \epsilon, \quad \epsilon \sim U_{[0,1]}$. U denotes the uniform distribution over the specified set. The data set consists of 25 trajectories.

C.1.4 Wave

The damped wave equation is given by

$$\frac{\partial^2 u}{\partial t^2} - c^2\Delta u + k\frac{\partial u}{\partial t} = 0 \tag{C.4}$$

For the data set we set $c = 330$ and $k = 50$ and $t \in [0, 0.0024]$. The initial conditions are the same as in Yin et al. [220], i.e., $u_0 \leftarrow \exp\left[\frac{-(x-m_0)^2-(y-m_1)^2}{\sigma}\right]$ where $\sigma \sim U_{[10,100]}$, $m_0, m_1 \sim U_d\{20, 40\}$. U_d denotes the discrete uniform distribution over the denoted interval. x, y are 64 dimensional square matrices with $x_{ij} = i$ and $y_{ij} = j$. The data set consists of 200 trajectories.

C.2 Implementation Details

C.2.1 Python Packages

In our code we make use of the following packages: Matplotlib [98], NumPy [83], JAX [20], Numpyro [162], Laplace-torch [39], SciPy [203], PyTorch [159] and Torchdiffeq [29].

C.2.2 Comparison to HMC

Details on HMC Settings

For HMC we use an implementation in JAX [20] as we found the sampling to be very fast for our particular model and easy to integrate in our existing setup. For our implementation we use the NUTS sampler supplied by JAX. We use 2000 samples for warm-up and 2000 samples. We used a normal distribution with zero mean and variance one as a prior for the weights. Our implementation is based on [35]. For this experiment we try to keep the network architecture as simple and small as possible, to keep sampling times reasonably fast.

Network Architecture

We used the same architecture for the HMC model and the Laplace model:

- $\text{Linear}(16, 2) \circ \tanh \circ \text{Linear}(2, 16)$.

As an ODE solver we use for both models Dopri5(4) with $rtol = atol = 1.4e^{-8}$. For the MAP trained model we use 10000 training iterations. **Computational requirements:** We trained the model on CPU requiring a computation time up to 1.5 h.

C.2.3 Hamiltonian Neural ODEs

The fact that a system is energy conserving cannot only be encoded into the architecture of the vector field but also in the algorithm of the numerical ODE solver. We solve all required ODEs using the Euler method for the naive model, and symplectic Euler for the Hamiltonian neural ODEs (see Section 3.2.1 for details on the numerical solvers). We set the batch size equal to the data set size of 16, and use 5000 iterations for training (we trained on CPU). Each data set consists of 16 trajectories with slightly different initial conditions. For our implementation we use code provided by [227] and base our architecture on the architecture proposed in this work. We reduced the network size slightly as this facilitated faster training and using the Laplace approximation on the entire network.

Neural Network Architectures

For the experiments on the harmonic oscillator data sets the following architectures were used:

- *naive*: $\text{Linear}(256, 2) \circ \tanh \circ \text{Linear}(2, 256)$
- *separable*: $V = T = \text{Linear}(128, 1, bias = false) \circ \tanh \circ \text{Linear}(1, 128)$
- *constrained*: $V = \text{Linear}(128, 1, bias = false) \circ \tanh \circ \text{Linear}(1, 128)$

Training of Hamiltonian Neural ODEs

Training of Hamiltonian neural ODEs can be unstable, especially if sufficient structure is not provided (similar observations have been made by Zhong et al. [227]). Potentially part of the issue is that Hamiltonian neural networks contain derivatives of neural networks. These derivatives also change the architecture of the neural network—e.g., a network with two linear layers and a tanh activation has a $1/\cosh^2$-activation after taking the derivative. Furthermore, different activation functions lead to different extrapolation properties of neural networks [214]. That is why the extrapolations of the vector field far away from the data look different for the naive model and the Hamiltonian models. **Computational requirements:** We trained the model on CPU requiring a computation time up to 8 h per run.

C.2.4 Augmented Parametric Models

Pendulum

We use the same neural network architecture for the parametric model and the plain neural ODE model:

- $\text{Linear}(32, 2) \circ \tanh \circ \text{Linear}(32, 32) \circ \tanh \circ \text{Linear}(2, 32)$.

We use 10000 epochs for training. For our implementation we use the code base provided by Yin et al. [220] and the architecture suggested in this work. **Computational requirements:** We trained the model on CPU requiring a computation time up to 5 h per run.

Wave

We use the following neural network architecture for the parametric model:

- $\text{Conv2d}(16, 2, \text{kernel\_size} = 3, \text{padding} = 1) \circ \text{ReLU} \circ \text{Conv2d}(16, 16, \text{kernel\_size} = 3, \text{padding} = 1, \text{bias} = false) \circ \text{ReLU} \circ \text{Conv2d}(16, 2, \text{kernel\_size} = 3, \text{padding} = 1, \text{bias} = false)$.

We use 1000 epochs for training. For our implementation we use the code base provided by Yin et al. [220]. We use the same architecture as in the paper, but removed the batch norm as this provided better results for us. **Computational requirements:** We trained the model on GPU (NVIDIA A100 Tensor Core GPU) requiring a computation time up to 12 h per run.

C.3 Additional Experimental Results

C.3.1 Sampling from the Laplace Posterior

As described in the main text, one option is to sample from the weight posterior. We compare a sampling based approach, the linearization approach and HMC on the Lotka-Volterra data set (see Figure **??**).

C.3.2 HMC on Pendulum Dataset

We also run a Bayesian neural network using HMC on the damped pendulum data set from Section C.1.3. However, we use a significantly smaller architecture as for the experiments in the main text, as the HMC sampling becomes too slow if the model has too many weights. Therefore, we use the same architecture as for the Lotka-Volterra experiment (see Section C.2.2). Since the smaller network size is not as expressive as the large network, we reduced the data set size from 25 samples to 4. This smaller data set also leads to a speedup of the HMC approach. The results for both the plain model and the augmented model are shown in Figure **??**. We find that as for the experiments in the main text, the vector field of the augmented model has a larger area of low uncertainty than the plain neural ODE model. Similarly, we observe that the extrapolation behavior augmented model is better (see Figure **??**). Increasing the data

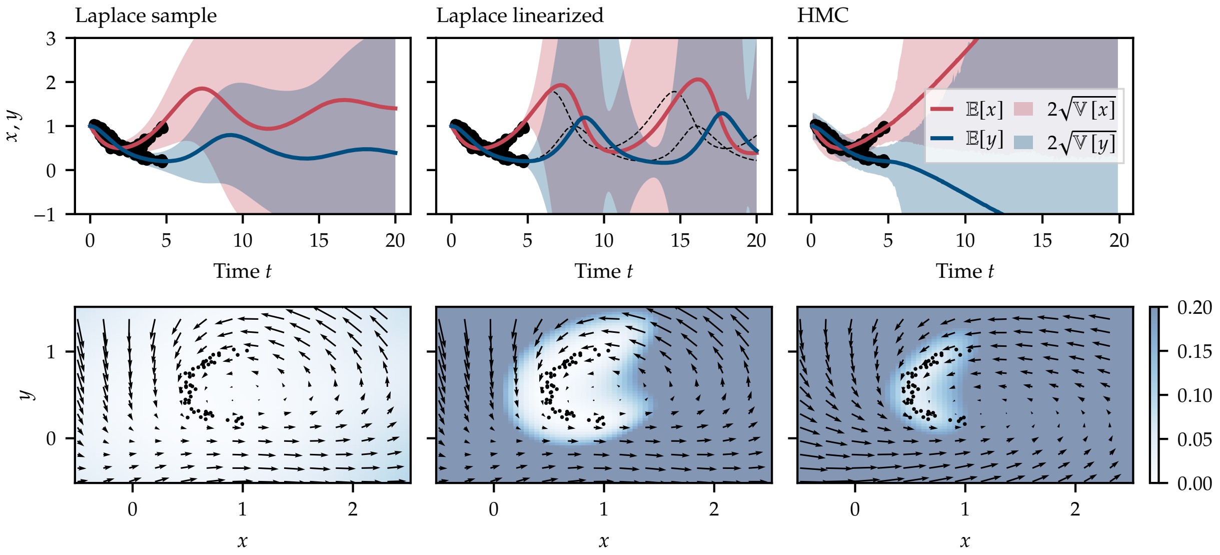


Figure C.1: (*Left*) Laplace approach using sampling, (*center*) Laplace approach using linearization, (*right*) HMC.

set would again improve extrapolation behavior even further, but also requires a network large enough to be expressive enough to capture all the data.

C.3.3 Hamiltonian Neural ODEs

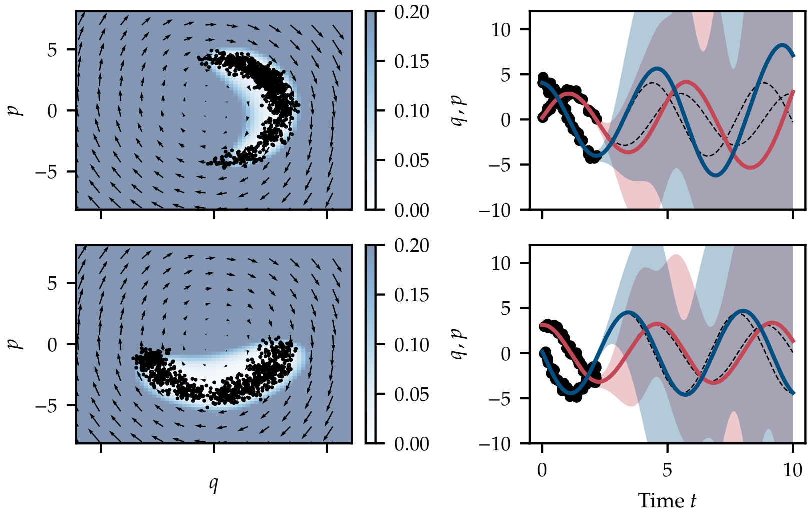


Figure C.3: Naive network trained on harmonic oscillator using the Euler method. Training on *data-set-lower-half* (*top*), *data-set-left-half* (*bottom*).

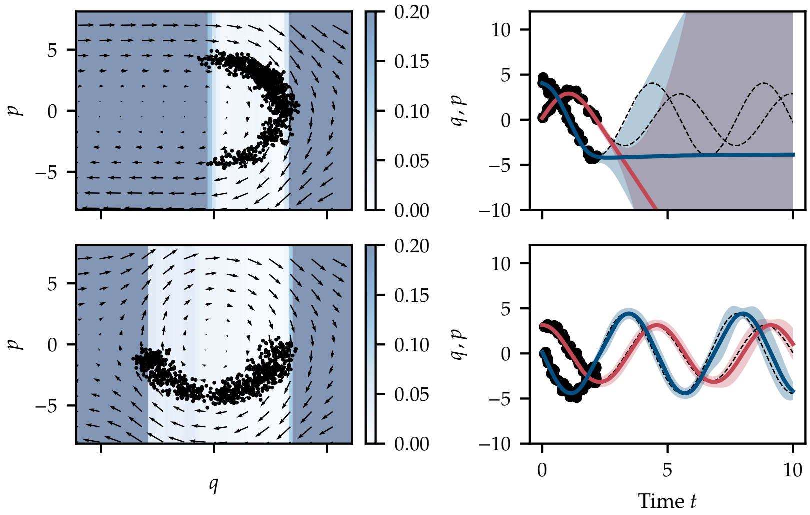


Figure C.4: Constrained network trained on harmonic oscillator using symplectic Euler. Training on *data-set-lower-half* (*top*), *data-set-left-half* (*bottom*).

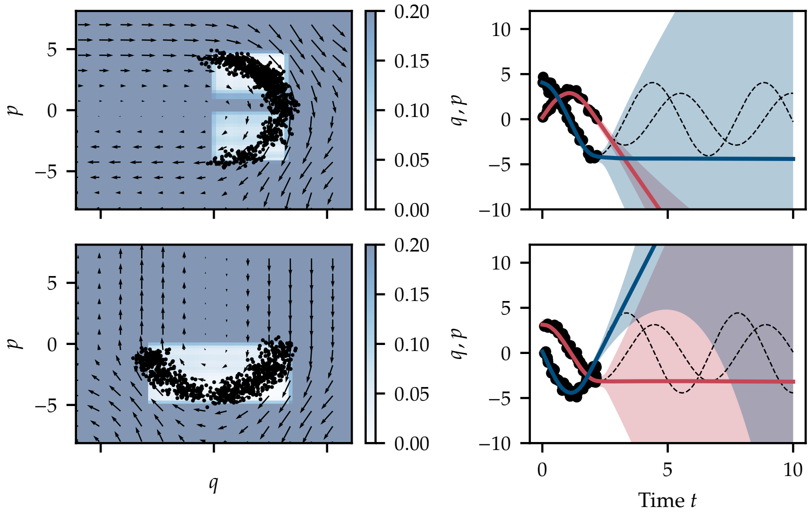


Figure C.5: Separable network trained on harmonic oscillator using symplectic Euler. Training on *data-set-lower-half* (*top*), *data-set-left-half* (*bottom*).

C.3.4 Additional Wave Results

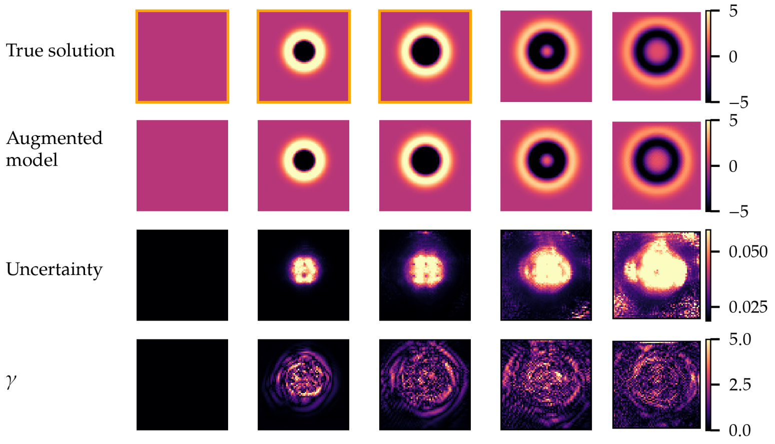


Figure C.6: *Laplace approximation applied to wave PDE.* Training an augmented parametric model on the damped wave equation data set (Figure shows result for du/dt). First three images (marked by orange frame) are part of the training data.

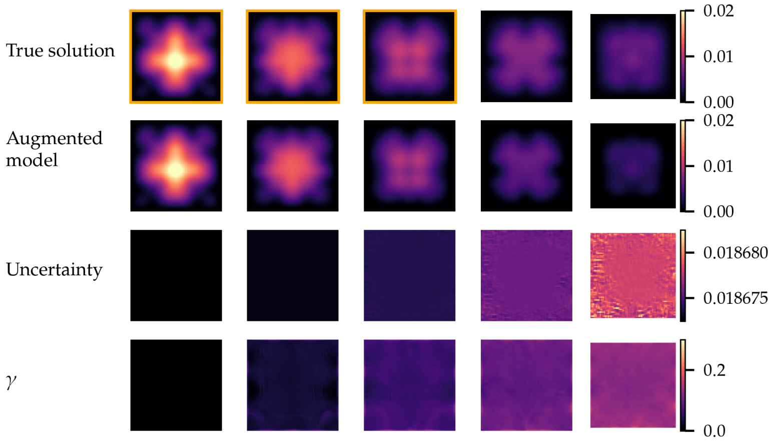


Figure C.7: *Laplace approximation applied to wave PDE.* Training an augmented parametric model on the damped wave equation data set with a different initial condition (Figure shows result for u). First three images (marked by orange frame) are part of the training data.

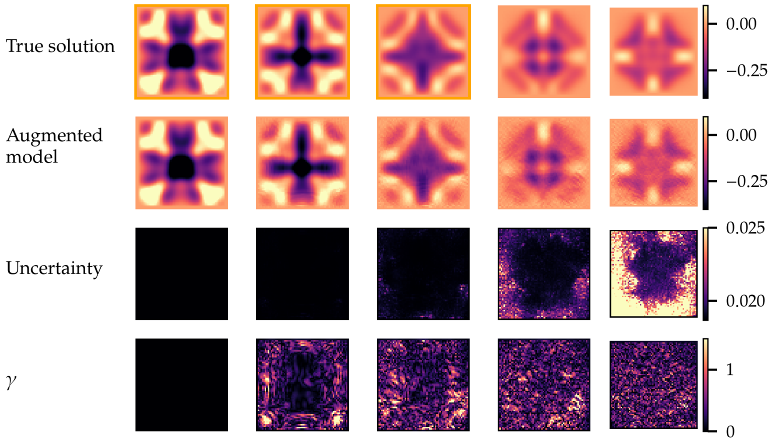


Figure C.8: *Laplace approximation applied to wave PDE.* Training an augmented parametric model on the damped wave equation data set with a different initial condition (Figure shows result for du/dt). First three images (marked by orange frame) are part of the training data.

Additional Material for Chapter 7 D.

D.1 Experimental Details

Below we give implementation details for all the data sets used and provide additional experimental results.

For our implementation of the BSN and the experiments described in the main text we make use of the following packages: `PyTorch` [159], `emukit` [158], `GPyTorch` [60], `laplace-torch` [39], `PyWake`[161], and `Matplotlib` [98].

D.1.1 Impact of Architecture Design

We provide additional discussion concerning the choice of activation function, choice of sampling strategy and choice of optimizer.

Choice of Optimizer

We compare different optimizers for the BSN and a standard neural network, where for the standard neural network we use the same architecture as for u_{θ_u}. We use the 1-dimensional wind farm data set with $N = 320$ data points. We choose this data set, due to the complicated structure of the score function of a mixture of Gaussians. For the experiment we consider three optimizers, i.e., Adam [110], L-BFGS [124] and the Hessian-free optimizer [134]. For Adam, we use mini-batching with a batch size of 32 and full-batch training. For the Hessian-free optimizer and L-BFGS we only consider full-batch training. For Adam, we use 10000 iterations, for the Hessian-free optimizer 1000 iterations, and for L-BFGS we use automatic stopping based on the strong Wolfe conditions. We compare the loss for all training methods. We also use CELU and RELU activation functions, where RELU is included as it is the standard activation function for neural networks.

Training a standard neural network with RELUs work significantly better, than using CELUs both in terms of the loss reached at the end of training and in terms of runtime (see Figure D.1 and Figure D.2). Using RELUs does not work for the BSN, as the gradients of u_{θ_u} lead to discontinuities.

Training of the BSN using Adam is considerably slower than the training progress of the standard neural network. We find that for CELU activation function, using (approximate) second order methods leads to a large improvement both in terms of speed and loss. The success of the second order methods might be due to a narrow loss landscape, i.e., a larger spread in the eigenvalue spectrum of the curvature. Therefore, we also examine the condition number of the Hessian, and we find that the BSN has a slightly higher condition number than the standard neural network (we do not report the condition number for RELUs as it cannot be

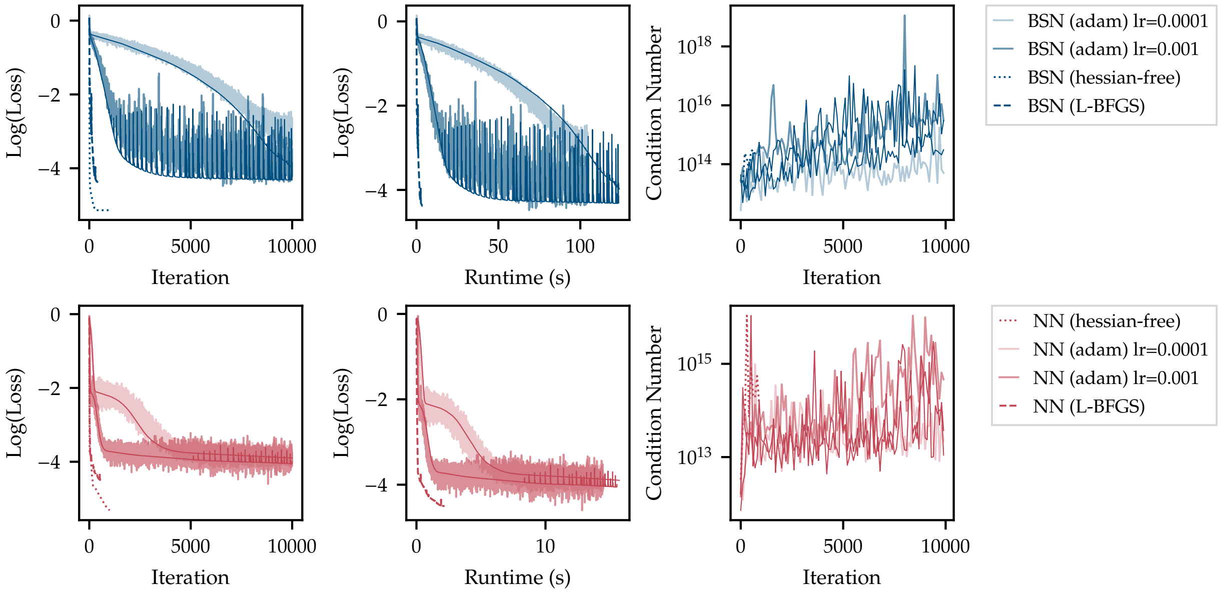


Figure D.1: Regression performance of a plain neural network (red) and a BSN (blue) using CELU activations. Loss (*left*), and condition number (*left*) as a function of the iteration. *Center:* loss as a function of the runtime. Thin dark lines correspond to training with full-batch Adam. Runtime of the Hessian-free optimizer not plotted, due to its long runtime.

computed numerically). Given its short runtime and good optimization results, we choose L-BFGS for all our experiments.

Sampling Strategies

For our experiments in the main text, we choose the data points by sampling from π, i.e., $x_n \sim \pi$. Here we consider two additional sampling strategies:

- Using a quasi-Monte Carlo (QMC) sequence. We use `SciPy`'s [203] implementation of QMC based on the Sobol sequence [193].
- Linearly spaced points in a hypercube (called *grid* in Figure D.3). Here we consider the hypercube $[-5\sigma_\pi, 5\sigma_\pi]^d$, where $\pi(x) = \mathcal{N}(x|0, \sigma_\pi)$.

Figure D.3 shows the result of the different sampling strategies in $d = 1$. The BSN performs better using MC samples then using QMC samples and grid points. The low performance of the latter is expected, since too few points a placed in regions with a high probability mass.

Choice of Architecture

We consider a basic architecture of the following form:

$$u_{\theta_u} = \text{Linear}(d, h) \circ CELU(\circ\text{Linear}(h, h) \circ CELU)^l \circ \text{Linear}(h, d),$$

where h are the number of hidden units and l are the number of hidden layers. Figure D.4 shows the performance of different architectures on the 1-dimensional continuous Genz data set. All architectures perform similar but the architecture with $l = 2$ and $h = 32$ reaches the lowest error the fastest for large N. Hence, we use this architecture for our experiments.

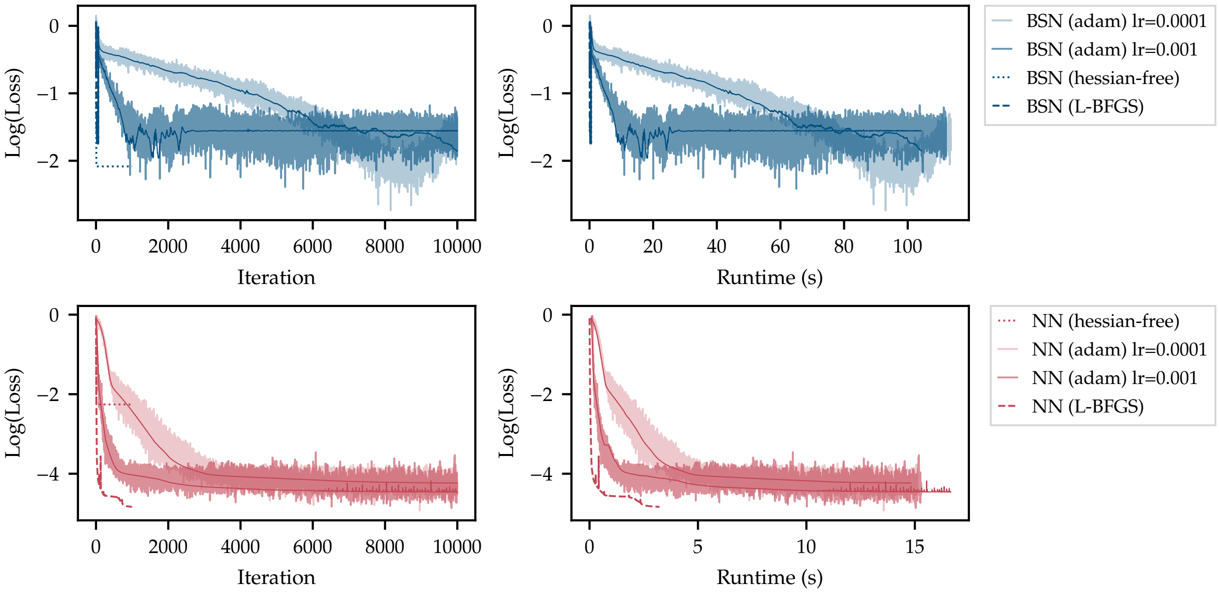


Figure D.2: Regression performance of a plain neural network (red) and a BSN (blue) using RELU activations. Loss (*left*) as a function of the iteration. *Center:* loss as a function of the runtime. Thin dark lines correspond to training with full-batch Adam. Runtime of the Hessian-free optimizer not plotted, due to its long runtime.

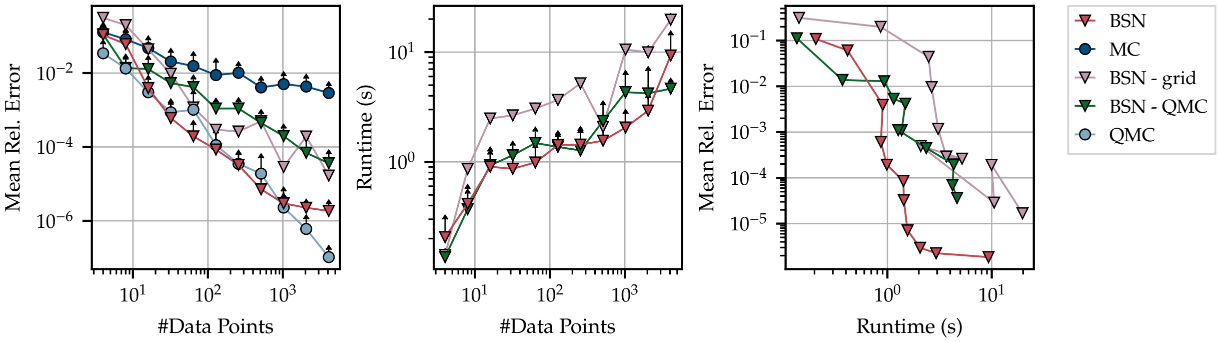


Figure D.3: Comparing different sampling schemes on the continuous Genz data set in $d = 1$. The BSN is trained on MC-sampled points, QMC-sampled points and on a regular grid. Mean relative integration error (*left*), and runtime (*center*), (based on 5 repetitions) as a function of N. *Right:* Mean relative integration error as a function of runtime in seconds.

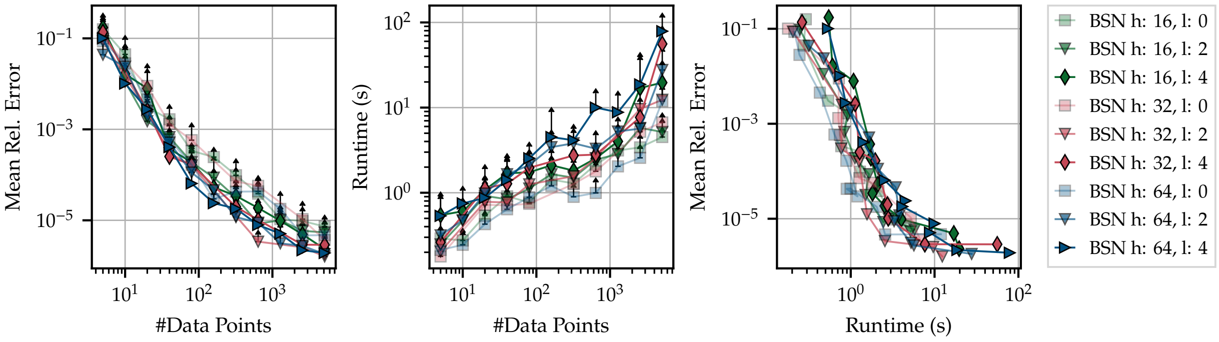


Figure D.4: Testing different architectures on the 1-dimensional continuous Genz data set. Mean relative integration error (*left*), and run time (*center*), (based on 5 repetitions) as a function of N. *Right:* Mean relative integration error as a function of run time in seconds.

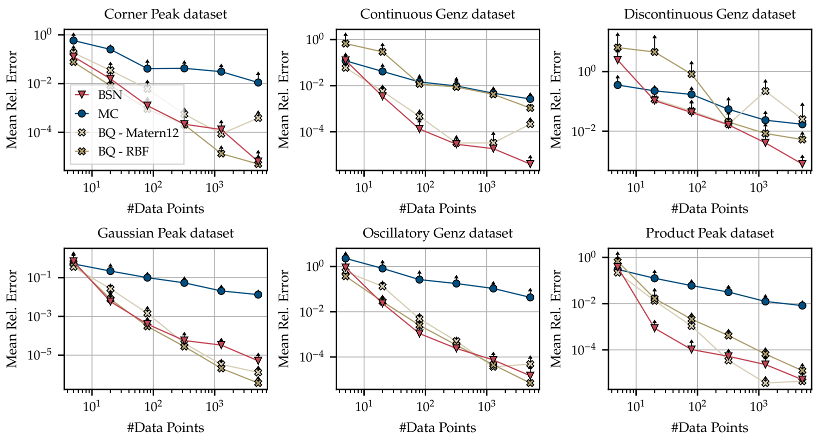


Figure D.5: *BQ with Matern 1/2 kernel on the Genz family in $d = 1$.* Mean relative integration error (based on 5 repetitions) as a function of n.

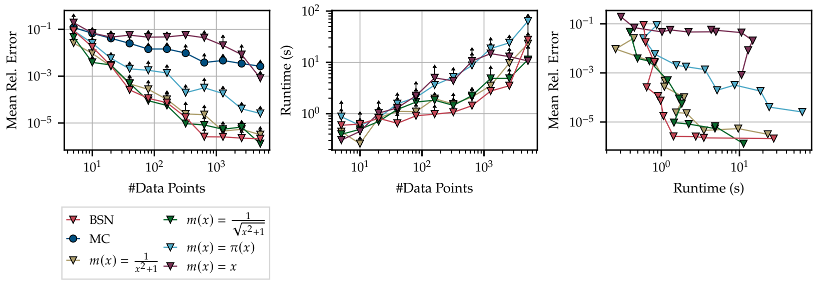


Figure D.6: *Continuous Genz dataset in $d = 1$ with different $m(x)$.* Mean relative integration error (*left*), run time (*center*) (based on 5 repetitions) as a function of n. *Right:* Mean relative integration error as a function of run time in seconds.

Choice of $m(x)$

For most of our experiments we set $m(x) = I_d$ in Equation 7.3. This might not necessarily be the best choice for a given task, but finding a function m that works well is hard. We test different m on the 1-dimensional Continuous Genz function:

- $m(x) = \frac{I_d}{||x||_2^2+1}$ - $m(x)$ goes to zero for $x \longrightarrow \pm\infty$
- $m(x) = \frac{I_d}{\sqrt{||x||_2^2+1}}$ - $m(x)$ goes to zero for $x \longrightarrow \pm\infty$ and cancels the $\nabla_x \log \pi(x)$ term for large x.
- $m(x) = I_d \pi(x)$ - in cases where π is a normal distribution, this function also goes to zero for $x \longrightarrow \pm\infty$.
- $m(x) = \text{diag} x$ - example of a function having negative effect.

The results of comparing these different m are shown in Figure D.6. On this test problem, none of the proposed m significantly outperforms the choice $m(x) = I_d$, with some performing significantly worse.

Choice of GP Kernel

As a benchmark we use BQ with an RBF kernel for all our experiments. The reason for this choice of kernel is the closed form availability of posterior mean and covariance when π is a normal distribution. Here we add an experiment using a Matern 1/2 kernel. For this choice of kernel the posterior mean is only available in $d = 1$, hence we conduct the experiment on the 1-dimensional Genz dataset (see Section D.2.1 for the expression of the kernel mean embedding). The corresponding results a found in Figure D.5. Once again, we do not observe a significant difference in performance, except for the continuous Genz dataset.

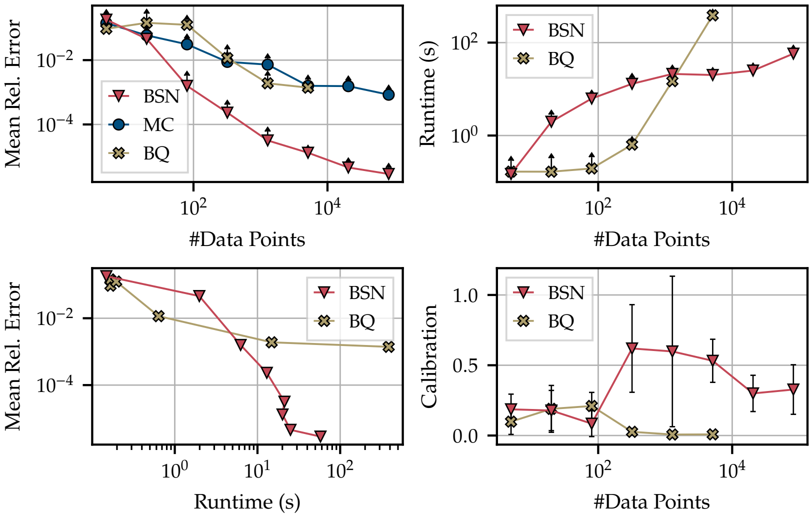


Figure D.7: *Continuous Genz data set in* $d = 2$. Mean relative integration error (*top-left*), run time (*top-right*), and calibration (*bottom-right*) (based on 5 repetitions) as a function of N. *Bottom-left:* Mean relative integration error as a function of run time in seconds.

D.1.2 Genz Benchmark

In our experiments we use the Genz integrand family data set. Here we include a short description of each data sett, plus additional experiments on the 2-dimensional version of each data set. In our experiments we integrate the Genz function against a standard normal $\pi(x) = \mathcal{N}(x|0,1)$. This requires the transformation of the inputs to the original Genz functions f, which are to be integrated against $[0,1]^d$. Therefore, we compute $\Pi_\pi[f \circ c]$ where $c(x) = \frac{1}{2}\left(1 + \text{erf}\left(\frac{x}{\sqrt{2}}\right)\right)$ is the cumulative density function of the standard normal. We give the form of f below.

Continuous Genz data set The integrand is given by

$$f(x) = \exp\left(-\sum_{i=1}^{d} a_i |x_i - u_i|\right)$$

with parameters $a_i = 1.3$ and $u_i = 0.55$. See Figure D.7 for results on a 2-dimensional version of this data set.

Corner Peak Data Set The integrand is given by

$$f(x) = \left(1 + \sum_{i=1}^{d} a_i x_i\right)^{-(d+1)}$$

with parameters $a_i = 5$. See Figure D.8 for results on a 2-dimensional version of this data set.

Discontinuous Genz Data Set The integrand is given by

$$f(x) = \begin{cases} 0, & \text{if } x_i > u_i \text{ for any i} \\ \exp\left(\sum_{i=1}^{d} a_i x_i\right) \end{cases}$$

with parameters $a_i = 5$ and $u_i = 0.5$. See Figure D.9 for results on a 2-dimensional version of this data set.

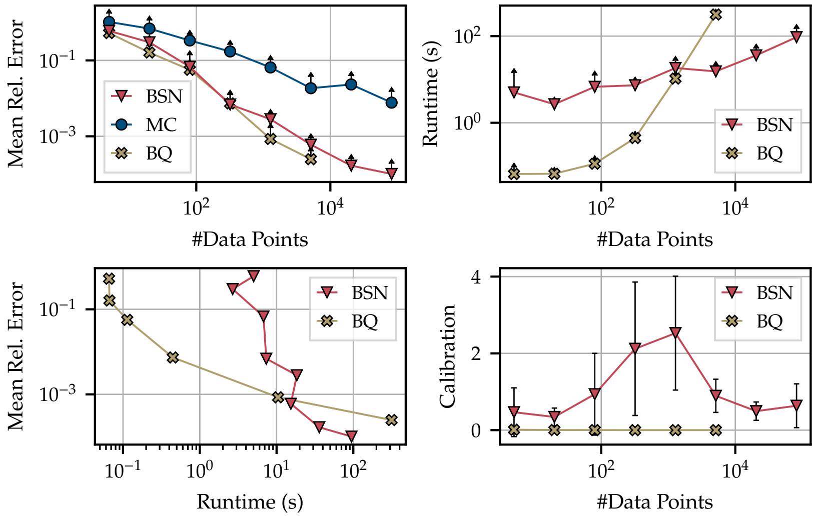


Figure D.8: *Corner Peak data set in d =* 2. Mean relative integration error (*top-left*), run time (*top-right*), and calibration (*bottom-right*) (based on 5 repetitions) as a function of N. *Bottom-left:* Mean relative integration error as a function of run time in seconds.

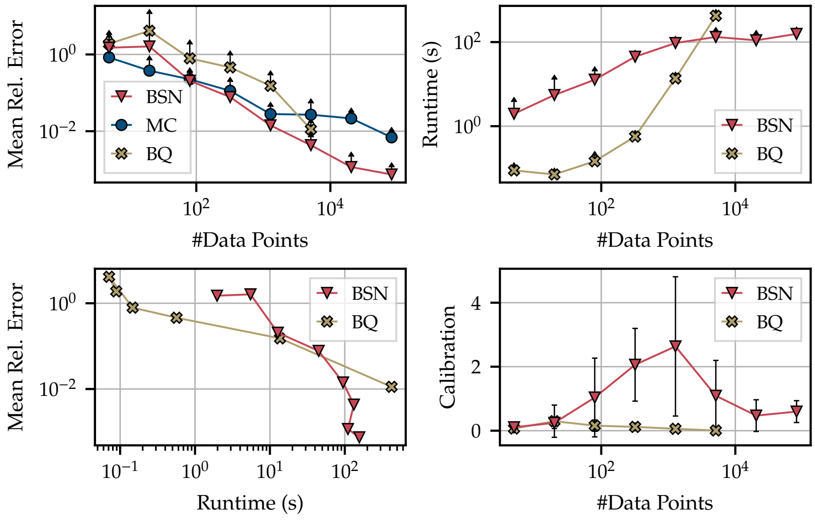


Figure D.9: *Discontinuous Genz data set in d* = 2. Mean relative integration error (*top-left*), run time (*top-right*), and calibration (*bottom-right*) (based on 5 repetitions) as a function of N. *Bottom-left:* Mean relative integration error as a function of run time in seconds.

Gaussian Peak Data Set The integrand is given by

$$f(x) = \exp\left(-\sum_{i=1}^{d} a_i^2 (x_i - u_i)^2\right)$$

with parameters $a_i = 5$ and $u_i = 0.5$ See Figure D.10 for results on a 2-dimensional version of this data set.

Product Peak Data Set The integrand is given by

$$f(x) = \prod_{i=1}^{d} \frac{1}{(a_i^{-2} + (x_i - u_i)^2)}$$

with parameters $a_i = 5$ and $u_i = 0.5$ See Figure D.11 for results on a 2-dimensional version of this data set.

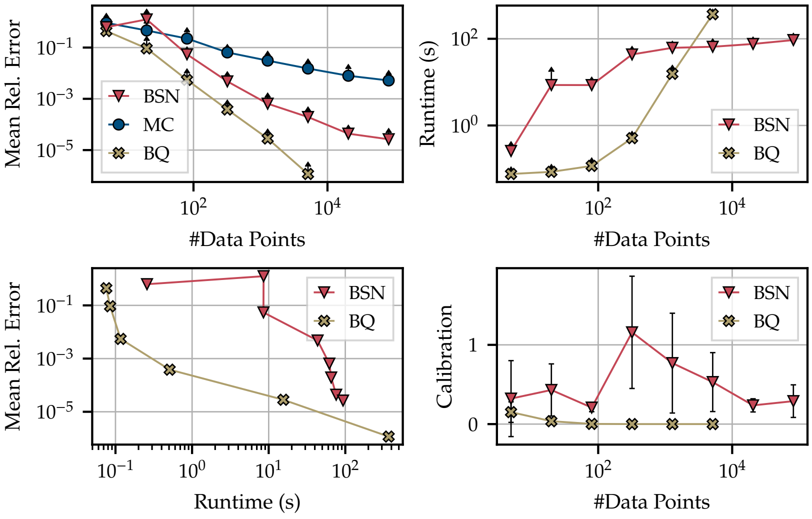


Figure D.10: *Gaussian peak data set in* $d = 2$. Mean relative integration error (*top-left*), run time (*top-right*), and calibration (*bottom-right*) (based on 5 repetitions) as a function of N. *Bottom-left:* Mean relative integration error as a function of run time in seconds.

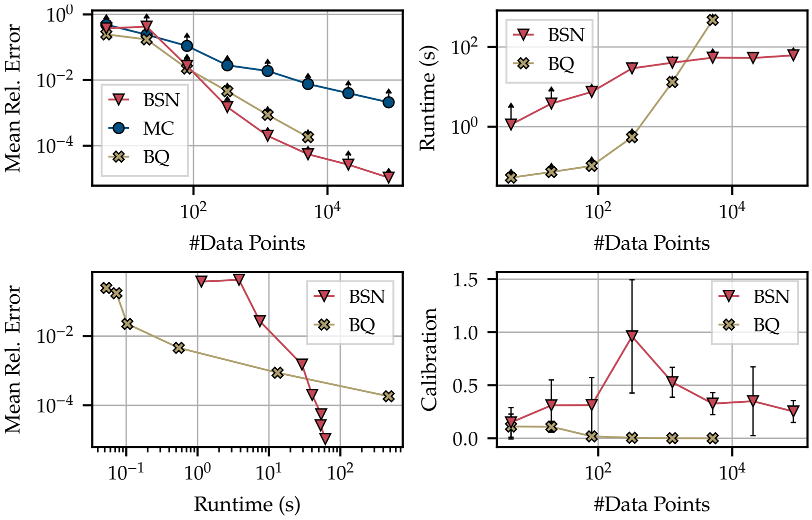


Figure D.11: *Product peak data set in* $d = 2$. Mean relative integration error (*top-left*), run time (*top-right*), and calibration (*bottom-right*) (based on 5 repetitions) as a function of N. *Bottom-left:* Mean relative integration error as a function of run time in seconds.

Oscillatory Genz Data Set The integrand is given by

$$f(x) = \cos\left(2\pi u + \sum_{i+1}^{d} a_i x_i\right)$$

with parameters $a_i = 5$ and $u = 0.5$ See Figure D.12 for results on a 2-dimensional version of this data set.

D.1.3 Goodwin Oscillator

Goodwin oscillator [68] describes how the feedback loop between mRNA transcription and protein expression can lead to oscillatory dynamics in a cell. We here consider the case with no intermediate protein species. The experimental setup is based on earlier work by [24, 30, 148, 176].

The Goodwin oscillator with no intermediate protein species is given

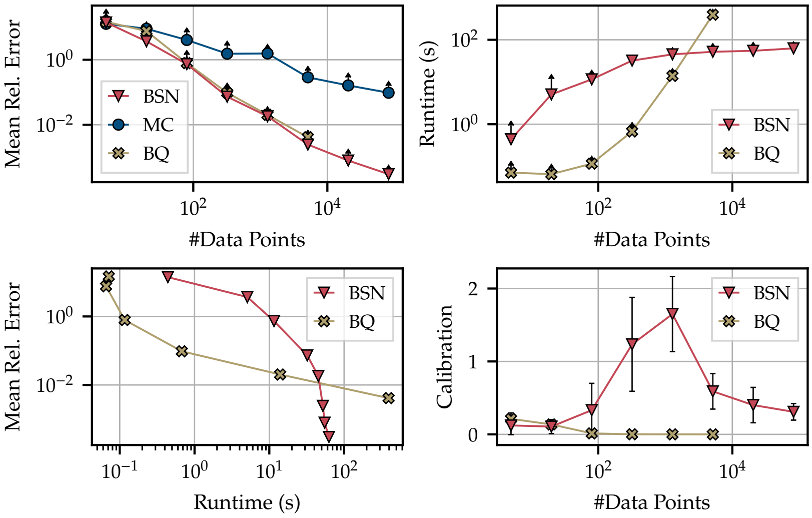


Figure D.12: *Oscillatory Genz data set in* $d = 2$. Mean relative integration error (*top-left*), run time (*top-right*), and calibration (*bottom-right*) (based on 5 repetitions) as a function of N. *Bottom-left:* Mean relative integration error as a function of run time in seconds.

by:

$$\frac{du_1}{dt} = \frac{a_1}{1 + a_2 u_2^{\rho}} - \alpha u_1$$

$$\frac{du_2}{dt} = k_1 u_1 - \alpha u_2,$$

where u_1 corresponds to the concentration of mRNA and u_2 to the concentration of the corresponding protein product. We set $\rho = 10$.

As initial conditions we set $u_0 = (0, 0)$. To generate the ground truth data set, we set $a_1 = 1, a_2 = 3, k_1 = 1$ and $\alpha = 0.5$. We use a measurement noise of $\sigma = (0.1, 0.05)$. Data was collected for 2400 time points in $t \in [1, 25]$, leading to the following expression for the likelihood:

$$p(y|x) \propto \exp\left(-\frac{1}{2\sigma_1^2}\sum_{k=1}^{2400} ||y_{1,k} - u_1(t_k)||_2^2 - \frac{1}{2\sigma_2^2}\sum_{k=1}^{2400} ||y_{2,k} - u_2(t_k)||_2^2\right)$$

We use an JAX's implementation of Dopri5(4) to solve the ODE. We use automatic differentiation implemented in JAX to compute derivatives of the likelihood with respect to the parameters. To avoid parameters becoming negative, we use log-transformed parameters $w = \log(x)$ for the parameter inference via MCMC. We place a standard normal prior on the log-transformed parameters w. For each data set we run five chains, where the initial conditions for each chain are sampled from the prior.

Figure D.13 shows the results for the remaining two parameters not shown in the main text.

D.1.4 Wind Farm Modelling

For the wind farm model in our experiments, we assume we have a large-scale wind farm with equally spaced turbines on a two-dimensional grid and an ambient turbulence intensity. For each turbine, we use a wake deficit model by Niayifar and Porté-Agel [141]. We put the following distributions on parameters for the wind farm simulation

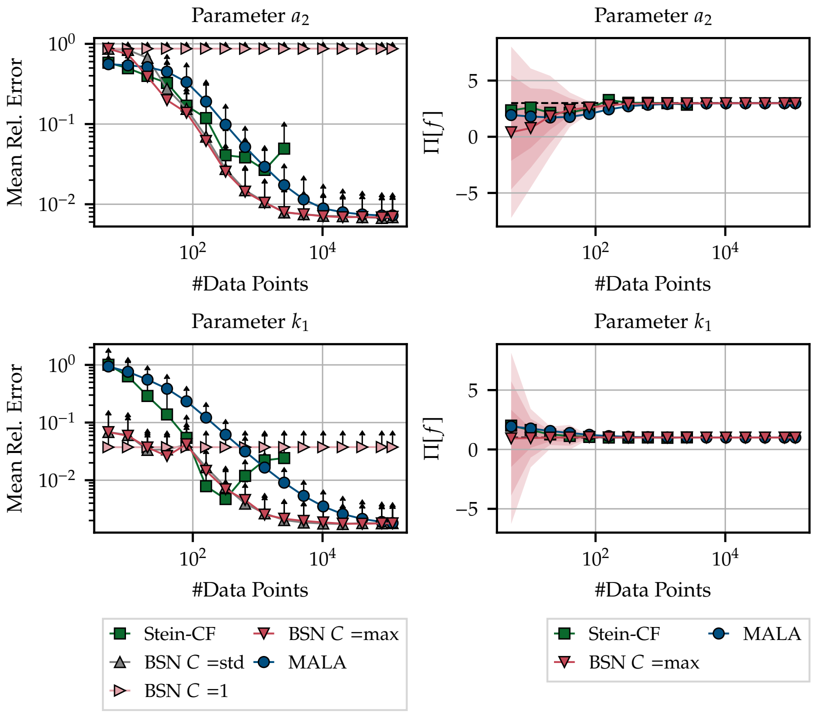


Figure D.13: *Posterior expectations for the parameters of a Goodwin ODE.* Mean relative integration error (*top-left* and *bottom-left*), and uncertainty estimates (*top-right* and *bottom-right*) (based on 5 repetitions) as a function of N.

- **Turbine resistance coefficient:** Gaussian distribution with mean $\mu = 1.33$ and variance $\sigma^2 = 0.1$.
- **Coefficient describing the wake expansion:** Gaussian distribution left-truncated at 0 with mean $\mu = 0.38$ and variance $\sigma = 0.001$.
- **Second coefficient describing the wake expansion:** Gaussian distribution left-truncated at 0 with mean $\mu = 4e - 3$ and variance $\sigma^2 = 1e - 8$.
- **Turbulence intensity:** Gaussian distribution left-truncated at 0 with mean $\mu = 0.1$ and variance $\sigma^2 = 0.003$
- **Wind direction:** Mixture of Gaussian distributions truncated to have support on $[0, 45]$ with means $\mu_1 = 0, \mu_2 = 22.5, \mu_3 = 33.75$ and variances $\sigma_1^2 = 50, \sigma_2^2 = 40, \sigma_3^2 = 8$.
- **Hub heights:** Gaussian distribution left-truncated at 0 with mean $\mu = 100$ and variance $\sigma^2 = 0.5$.
- **Hub diameter:** Gaussian distribution left-truncated at 0 with mean $\mu = 100$ and variance $\sigma^2 = 0.1$.

These distributions were chosen to have scales which might realistically represent uncertainty for their input, but if applying our method in practice these would have to be elicited from wind-farm experts. Note that the BSNs could be applied to much more complex distributions so-long as the density of Π can be evaluated point-wise up to some normalization constant.

Our code is based on the code estimating the local turbine thrust coefficient Kirby et al. [112] using a low-order wake model provided here: `https://github.com/AndrewKirby2/ctstar_statistical_model`. This code is based on the `PyWake` package [161].

D.2 Kernel Mean Embedding for Truncated Gaussians

We now provide the derivation of the kernel mean embedding for truncated Gaussians. For truncated Gaussian distributions and the RBF kernel, we can compute the posterior mean but not the posterior variance. Here we consider the 1-dimensional case with $\mathcal{X} = [a, b]$ which can be extended to the d-dimensinal case for isotropic Gaussians. We provide the expression for the kernel mean embedding: $\Pi[k(\cdot, x)] = \int_{\mathcal{X}} k(x', x)\pi(x')dx'$. We consider the case when π is a truncated Gaussian and introduce the following notation:

$$\pi(x) = \frac{\phi(x, \mu, \sigma)}{\Phi(\frac{b-\mu}{\sigma}) - \Phi(\frac{a-\mu}{\sigma})}$$

where $\phi(x) = (\sqrt{2\pi}\sigma)^{-1} \exp(-(x-\mu)^2/2\sigma^2)$ and $\Phi(x) = \frac{1}{2}(1+\text{erf}(x/\sqrt{2}))$.

We use Z to denote the normalization constant

$$Z(a, b, \mu, \sigma) = \Phi\left(\frac{b-\mu}{\sigma}\right) - \Phi\left(\frac{a-\mu}{\sigma}\right)$$

We rewrite the RBF kernel using the above identities $k(x, x') = \exp\left(-(x - x')^2/2l^2\right) = l\sqrt{2\pi}\phi(x, x', l)$. We can now express the kernel mean embedding as:

$$\begin{aligned}\Pi[k(\cdot, x)] &= \int_a^b l\sqrt{2\pi}\phi(x, x', l)\frac{\phi(x', \mu, \sigma)}{Z(a, b, \mu, \sigma)}dx' \\ &= Cl\sqrt{2\pi}\int_a^b \frac{\phi(x', \tilde{\mu}, \tilde{\sigma})}{Z(a, b, \mu, \sigma)}dx' \\ &= l\sqrt{2\pi}C\frac{Z(a, b, \tilde{\mu}, \tilde{\sigma})}{Z(a, b, \mu, \sigma)},\end{aligned}$$

where

$$\tilde{\mu} = \frac{\mu l^2 + x\sigma^2}{\sigma^2 + l^2}, \qquad \tilde{\sigma} = \sqrt{\frac{\sigma^2 l^2}{\sigma^2 + l^2}}, \qquad C = \frac{1}{\sqrt{2\pi(\sigma^2 + l^2)}}\exp\left(\frac{(\mu - x)^2}{2(\sigma^2 + l^2)}\right).$$

D.2.1 Kernel Mean Embedding for Matern 1/2 Kernel

For Gaussian distributions and the Matern 1/2 kernel, we can compute the posterior mean but only in $d = 1$. We provide the expression for the kernel mean embedding: $\Pi[k(\cdot, x)] = \int_{\mathbb{R}} k(x', x)\pi(x')dx'$, where $\pi(x) = \mathcal{N}(0, 1)$ is a standard normal and $k(x', x) = \exp(|x - x'|/l)$ is the Matern 1/2 kernel.

$$\Pi[k(\cdot, x)] = \frac{1}{2}\exp\left(\frac{2xl + 1}{2l^2}\right)\text{erfc}\left(\frac{x + \frac{1}{l}}{\sqrt{2}}\right) + \frac{1}{2}\exp\left(\frac{1 - 2xl}{2l^2}\right)\left(\text{erf}\left(\frac{x - \frac{1}{l}}{\sqrt{2}}\right) + 1\right).$$

Bibliography

[1] A. Anandkumar, K. Azizzadenesheli, K. Bhattacharya, N. Kovachki, Z. Li, B. Liu, and A. Stuart. "Neural Operator: Graph Kernel Network for Partial Differential Equations". *ICLR 2020 Workshop on Integration of Deep Neural Models and Differential Equations*. 2019.

[2] A. Anastasiou, A. Barp, F.-X. Briol, B. Ebner, R. E. Gaunt, F. Ghaderinezhad, J. Gorham, A. Gretton, C. Ley, Q. Liu, L. Mackey, C. J. Oates, G. Reinert, and Y. Swan. "Stein's method meets statistics: A review of some recent developments". *arXiv:2105.03481* (2021).

[3] S. Anumasa and P. Srijith. "Improving Robustness and Uncertainty Modelling in Neural Ordinary Differential Equations". *2021 IEEE Winter Conference on Applications of Computer Vision (WACV)*. 2021.

[4] R. F. Arenstorf. "Periodic Solutions of the Restricted Three Body Problem Representing Analytic Continuations of Keplerian Elliptic Motions". *American Journal of Mathematics* 85.1 (1963), pages 27–35. (Visited on 01/26/2023).

[5] M. Arjovsky, S. Chintala, and L. Bottou. "Wasserstein generative adversarial networks". *International conference on machine learning*. PMLR. 2017, pages 214–223.

[6] B. Avelin and K. Nyström. "Neural ODEs as the deep limit of ResNets with constant weights". *Analysis and Applications* (2020).

[7] S. Bai, J. Z. Kolter, and V. Koltun. "Deep equilibrium models". *Advances in Neural Information Processing Systems* 32 (2019).

[8] D. Balduzzi, M. Frean, L. Leary, J. Lewis, K. W.-D. Ma, and B. McWilliams. "The shattered gradients problem: If resnets are the answer, then what is the question?" *Proceedings of the 34th International Conference on Machine Learning*. 2017, pages 342–350.

[9] A. Barp, F.-X. Briol, A. B. Duncan, M. Girolami, and L. Mackey. "Minimum Stein discrepancy estimators". *Neural Information Processing Systems*. 2019, pages 12964–12976.

[10] A. Barp, C. J. Oates, E. Porcu, and M. Girolami. "A Riemannian–Stein kernel method". *Bernoulli, (to appear), arXiv:1810.04946* (2022).

[11] J. T. Barron. "Continuously differentiable exponential linear units". *arXiv preprint arXiv:1704.07483* (2017).

[12] D. Bau, J.-Y. Zhu, J. Wulff, W. Peebles, H. Strobelt, B. Zhou, and A. Torralba. "Seeing what a gan cannot generate". *Proceedings of the IEEE/CVF International Conference on Computer Vision*. 2019, pages 4502–4511.

[13] J. Behrmann, W. Grathwohl, R. T. Chen, D. Duvenaud, and J.-H. Jacobsen. "Invertible residual networks". *arXiv preprint arXiv:1811.00995* (2018).

[14] D. Belomestny, A. Goldman, A. Naumov, and S. Samsonov. "Theoretical guarantees for neural control variates in MCMC". *arXiv:2304.01111* (2023).

[15] Y. Bengio, P. Simard, and P. Frasconi. "Learning long-term dependencies with gradient descent is difficult". *IEEE Transactions on Neural Networks* 5.2 (1994), pages 157–166.

[16] M. Benning, E. Celledoni, M. J. Ehrhardt, B. Owren, and C.-B. Schönlieb. "Deep learning as optimal control problems: Models and numerical methods". *Journal of Computational Dynamics* 6 (2019), page 171.

[17] J. Bezanson, A. Edelman, S. Karpinski, and V. B. Shah. "Julia: A fresh approach to numerical computing". *SIAM review* 59.1 (2017), pages 65–98.

[18] C. Blundell, J. Cornebise, K. Kavukcuoglu, and D. Wierstra. "Weight Uncertainty in Neural Network". *International Conference on Machine Learning*. 2015.

[19] L. Bo, A. Capponi, and H. Liao. "Relaxed Control and Gamma-Convergence of Stochastic Optimization Problems with Mean Field". *arXiv preprint arXiv:1906.08894* (2019).

[20] J. Bradbury, R. Frostig, P. Hawkins, M. J. Johnson, C. Leary, D. Maclaurin, G. Necula, A. Paszke, J. VanderPlas, S. Wanderman-Milne, and Q. Zhang. *JAX: composable transformations of Python+NumPy programs*. Version 0.2.5. 2018.

[21] F.-X. Briol, C. J. Oates, M. Girolami, and M. A. Osborne. "Frank-Wolfe Bayesian quadrature: Probabilistic integration with theoretical guarantees". *Neural Information Processing Systems*. 2015, pages 1162–1170.

[22] F.-X. Briol, C. J. Oates, M. Girolami, M. A. Osborne, and D. Sejdinovic. "Probabilistic integration: a role in statistical computation?" *Statistical Science* 34.1 (2019), pages 1–22.

[23] J. C. Burkill. "The theory of ordinary differential equations". *Longman Publishing Group*, 1975.

[24] B. Calderhead and M. Girolami. "Estimating Bayes factors via thermodynamic integration and population MCMC". *Computational Statistics and Data Analysis* 53.12 (2009), pages 4028–4045.

[25] B. Calderhead and M. Girolami. "Statistical analysis of nonlinear dynamical systems using differential geometric sampling methods". *Journal of Royal Society: Interface Focus* 1 (2011), pages 821–835.

[26] B. Chang, L. Meng, E. Haber, L. Ruthotto, D. Begert, and E. Holtham. "Reversible architectures for arbitrarily deep residual neural networks". *Thirty-Second AAAI Conference on Artificial Intelligence*. 2018.

[27] B. Chang, L. Meng, E. Haber, F. Tung, and D. Begert. "Multi-level residual networks from dynamical systems view". *arXiv preprint arXiv:1710.10348* (2017).

[28] C. Chen, C. Li, L. Chen, W. Wang, Y. Pu, and L. C. Duke. "Continuous-Time Flows for Efficient Inference and Density Estimation". *Proceedings of the 35th International Conference on Machine Learning*. Volume 80. 2018, pages 824–833.

[29] T. Q. Chen, Y. Rubanova, J. Bettencourt, and D. K. Duvenaud. "Neural ordinary differential equations". *Advances in Neural Information Processing Systems*. 2018, pages 6571–6583.

[30] W. Y. Chen, A. Barp, F.-X. Briol, J. Gorham, M. Girolami, L. Mackey, and C. J. Oates. "Stein Point Markov Chain Monte Carlo". *International Conference on Machine Learning*. 2019, pages 1011–1021.

[31] K. Choromanski, J. Q. Davis, V. Likhosherstov, X. Song, J.-J. Slotine, J. Varley, H. Lee, A. Weller, and V. Sindhwani. "An Ode to an ODE". *arXiv preprint arXiv:2006.11421* 33 (2020).

[32] M. Ciccone, M. Gallieri, J. Masci, C. Osendorfer, and F. Gomez. "NAIS-Net: Stable Deep Networks from Non-Autonomous Differential Equations". *Advances in Neural Information Processing Systems 31*. Edited by S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett. 2018, pages 3025–3035.

[33] J. Cockayne, C. Oates, T. Sullivan, and M. Girolami. "Bayesian probabilistic numerical methods". *SIAM Review* 61.4 (2019), pages 756–789.

[34] M. Cranmer, S. Greydanus, S. Hoyer, P. Battaglia, D. Spergel, and S. Ho. "Lagrangian Neural Networks". *ICLR 2020 Workshop on Integration of Deep Neural Models and Differential Equations*. 2020.

[35] R. Dandekar, K. Chung, V. Dixit, M. Tarek, A. Garcia-Valadez, K. V. Vemula, and C. Rackauckas. "Bayesian neural ordinary differential equations". *arXiv preprint arXiv:2012.07244* (2020).

[36] F. Dangel, F. Kunstner, and P. Hennig. "BackPACK: Packing more into Backprop". *International Conference on Learning Representations (ICLR)*. 2020.

[37] T. Daulbaev, A. Katrutsa, L. Markeeva, J. Gusak, A. Cichocki, and I. Oseledets. "Interpolation technique to speed up gradients propagation in neural odes". *Advances in Neural Information Processing Systems* 33 (2020), pages 16689–16700.

[38] P. J. Davis and P. Rabinowitz. "Methods of numerical integration". *Academic Press*, 1975.

[39] E. Daxberger, A. Kristiadi, A. Immer, R. Eschenhagen, M. Bauer, and P. Hennig. "Laplace Redux - Effortless Bayesian Deep Learning". *Advances in Neural Information Processing Systems*. 2021, pages 20089–20103.

[40] E. Daxberger, E. Nalisnick, J. U. Allingham, J. Antoran, and J. M. Hernandez-Lobato. "Bayesian Deep Learning via Subnetwork Inference". *Proceedings of the 38th International Conference on Machine Learning*. Volume 139. Proceedings of Machine Learning Research. 2021.

[41] E. De Brouwer, J. Gonzalez, and S. Hyland. "Predicting the impact of treatments over time with uncertainty aware neural differential equations." *International Conference on Artificial Intelligence and Statistics*. PMLR. 2022, pages 4705–4722.

[42] E. De Brouwer, J. Simm, A. Arany, and Y. Moreau. "GRU-ODE-Bayes: Continuous Modeling of Sporadically-Observed Time Series". *Advances in Neural Information Processing Systems*. Volume 32. 2019.

[43] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. "Imagenet: A large-scale hierarchical image database". *2009 IEEE conference on computer vision and pattern recognition*. Ieee. 2009, pages 248–255.

[44] J. Denker and Y. LeCun. "Transforming neural-net output levels to probability distributions". *Advances in neural information processing systems* 3 (1990).

[45] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova. "Bert: Pre-training of deep bidirectional transformers for language understanding". *arXiv preprint arXiv:1810.04805* (2018).

[46] P. Diaconis. "Bayesian numerical analysis". *Statistical Decision Theory and Related Topics IV* (1988), pages 163–175.

[47] D. Driess, F. Xia, M. S. Sajjadi, C. Lynch, A. Chowdhery, B. Ichter, A. Wahid, J. Tompson, Q. Vuong, T. Yu, et al. "Palm-e: An embodied multimodal language model". *arXiv preprint arXiv:2303.03378* (2023).

[48] S. R. Dubey, S. K. Singh, and B. B. Chaudhuri. "Activation functions in deep learning: A comprehensive survey and benchmark". *Neurocomputing* 503 (2022), pages 92–108.

[49] J. Duchi, E. Hazan, and Y. Singer. "Adaptive subgradient methods for online learning and stochastic optimization." *Journal of machine learning research* 12.7 (2011).

[50] E. Dupont, A. Doucet, and Y. W. Teh. "Augmented neural odes". *Advances in Neural Information Processing Systems*. 2019, pages 3134–3144.

[51] W. E. "A Proposal on Machine Learning via Dynamical Systems". *Communications in Mathematics and Statistics* 5.1 (2017), pages 1–11.

[52] K. Ensinger, F. Solowjow, M. Tiemann, and S. Trimpe. "Symplectic Gaussian Process Dynamics". *arXiv preprint arXiv:2102.01606* (2021).

[53] W. Fan, Y. Ma, Q. Li, Y. He, E. Zhao, J. Tang, and D. Yin. "Graph neural networks for social recommendation". *The world wide web conference*. 2019, pages 417–426.

[54] S. Farquhar, L. Smith, and Y. Gal. "Liberty or Depth: Deep Bayesian Neural Nets Do Not Need Complex Weight Posterior Approximations". *Advances in Neural Information Processing Systems*. Volume 33. 2020.

[55] E. Fehlberg. "Low-Order Classical Runge-Kutta Formulas with Stepsize Control and Their Application to Some Heat Transfer Problems". *National Aeronautics and Space Administration* (1969).

[56] C. Finlay, J.-H. Jacobsen, L. Nurbekyan, and A. M. Oberman. "How to train your neural ODE". *arXiv preprint arXiv:2002.02798* (2020).

[57] V. Fortuin, A. Garriga-Alonso, S. W. Ober, F. Wenzel, G. Ratsch, R. E. Turner, M. van der Wilk, and L. Aitchison. "Bayesian Neural Network Priors Revisited". *International Conference on Learning Representations*. 2022.

[58] K. Fukushima. "Visual Feature Extraction by a Multilayered Network of Analog Threshold Elements". *IEEE Transactions on Systems Science and Cybernetics* 5.4 (1969), pages 322–333.

[59] K. Fukushima. "Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in potion". *Biological cybernetics* 36.4 (1980), pages 193–202.

[60] J. R. Gardner, G. Pleiss, D. Bindel, K. Q. Weinberger, and A. G. Wilson. "gardner2018gpytorch: Blackbox Matrix-Matrix Gaussian Process Inference with GPU Acceleration". *Advances in Neural Information Processing Systems*. 2018.

[61] A. E. Gelfand and A. F. Smith. "Sampling-based approaches to calculating marginal densities". *Journal of the American statistical association* 85.410 (1990), pages 398–409.

[62] A. Genz. “Testing multidimensional integration routines”. *Proc. of international conference on Tools, methods and languages for scientific and engineering computation*. 1984, pages 81–94.

[63] A. Gessner, J. Gonzalez, and M. Mahsereci. “Active multi-information source Bayesian quadrature”. *Uncertainty in Artificial Intelligence*. 2019.

[64] A. Gholaminejad, K. Keutzer, and G. Biros. “ANODE: Unconditionally Accurate Memory-Efficient Gradients for Neural ODEs”. *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI-19. International Joint Conferences on Artificial Intelligence Organization*, 2019, pages 730–736.

[65] P. B. Gibson, W. E. Chapman, A. Altinok, L. Delle Monache, M. J. DeFlorio, and D. E. Waliser. “Training machine learning models on climate model output yields skillful interpretable seasonal precipitation forecasts”. *Communications Earth & Environment* 2.1 (2021), page 159.

[66] I. Goodfellow, Y. Bengio, A. Courville, and Y. Bengio. “Deep learning”. Volume 1. *MIT press Cambridge*, 2016.

[67] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio. “Generative Adversarial Nets”. *Advances in Neural Information Processing Systems*. Volume 27. *Curran Associates, Inc.*, 2014.

[68] B. C. Goodwin. “Oscillatory behavior in enzymatic control processes”. *Advances in Enzyme Regulation* (1965).

[69] J. Gorham, A. Duncan, L. Mackey, and S. Vollmer. “Measuring sample quality with diffusions”. *Annals of Applied Probability* 29.5 (2019), pages 2884–2928.

[70] W. Grathwohl, R. T. Q. Chen, J. Bettencourt, and D. Duvenaud. “Scalable Reversible Generative Models with Free-form Continuous Dynamics”. *International Conference on Learning Representations*. 2019.

[71] W. Grathwohl, R. T. Q. Chen, J. Bettencourt, I. Sutskever, and D. Duvenaud. “FFJORD: Free-form Continuous Dynamics for Scalable Reversible Generative Models”. *International Conference on Learning Representations* (2019).

[72] A. Graves. “Practical Variational Inference for Neural Networks”. *Neural Information Processing Systems*. 2011.

[73] U. Grenander and M. I. Miller. “Representations of knowledge in complex systems”. *Journal of the Royal Statistical Society: Series B (Methodological)* 56.4 (1994), pages 549–581.

[74] S. Greydanus, M. Dzamba, and J. Yosinski. “Hamiltonian Neural Networks”. *Advances in Neural Information Processing Systems*. Volume 32. 2019.

[75] T. Gunter, R. Garnett, M. Osborne, P. Hennig, and S. Roberts. “Sampling for inference in probabilistic models with fast Bayesian quadrature”. *Neural Information Processing Systems*. 2014, pages 2789–2797.

[76] J. Gusak, A. Katrutsa, T. Daulbaev, A. Cichocki, and I. Oseledets. “Meta-solver for neural ordinary differential equations”. *arXiv preprint arXiv:2103.08561* (2021).

[77] J. Gusak, L. Markeeva, T. Daulbaev, A. Katrutsa, A. Cichocki, and I. Oseledets. “Towards Understanding Normalization in Neural {ODE}s”. *ICLR 2020 Workshop on Integration of Deep Neural Models and Differential Equations*. 2020.

[78] E. Haber, K. Lensink, E. Treister, and L. Ruthotto. “IMEXnet A Forward Stable Deep Neural Network”. *Proceedings of the 36th International Conference on Machine Learning*. Volume 97. 2019, pages 2525–2534.

[79] E. Haber and L. Ruthotto. “Stable architectures for deep neural networks”. *Inverse Problems* 34.1 (2017), page 014004.

[80] E. Hairer, S. Nørsett, and G. Wanner. “Solving Ordinary Differential Equations I – Nonstiff Problems”. 2nd edition. *Springer*, 1993.

[81] E. Hairer, C. Lubich, and G. Wanner. “Geometric numerical integration”. Second. Volume 31. Springer Series in Computational Mathematics. *Springer-Verlag, Berlin*, 2006, pages xviii+644.

[82] M. Hardt, B. Recht, and Y. Singer. "Train faster, generalize better: Stability of stochastic gradient descent". *International conference on machine learning*. PMLR. 2016, pages 1225–1234.

[83] C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, R. Kern, M. Picus, S. Hoyer, M. H. van Kerkwijk, M. Brett, A. Haldane, J. F. del Río, M. Wiebe, P. Peterson, P. Gérard-Marchant, K. Sheppard, T. Reddy, W. Weckesser, H. Abbasi, C. Gohlke, and T. E. Oliphant. "Array programming with NumPy". *Nature* (2020).

[84] K. Hashimoto, H.-Y. Hu, and Y.-Z. You. "Neural ordinary differential equation and holographic quantum chromodynamics". *Machine Learning: Science and Technology* 2.3 (2021), page 035011.

[85] W. K. Hastings. "Monte Carlo sampling methods using Markov chains and their applications" (1970).

[86] K. He, X. Zhang, S. Ren, and J. Sun. "Deep residual learning for image recognition". *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pages 770–778.

[87] K. He, X. Zhang, S. Ren, and J. Sun. "Identity Mappings in Deep Residual Networks". *Computer Vision – ECCV 2016*. Cham: *Springer International Publishing*, 2016, pages 630–645.

[88] P. Hegde, M. Heinonen, H. Lähdesmäki, and S. Kaski. "Deep learning with differential Gaussian process flows". *The 22nd International Conference on Artificial Intelligence and Statistics*. 2019, pages 1812–1821.

[89] P. Hegde, Ç. Yildiz, H. Lähdesmäki, S. Kaski, and M. Heinonen. "Bayesian inference of ODEs with Gaussian processes". *arXiv preprint arXiv:2106.10905* (2021).

[90] M. Hein, M. Andriushchenko, and J. Bitterwolf. "Why ReLU Networks Yield High-Confidence Predictions Far Away From the Training Data and How to Mitigate the Problem". *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. 2019.

[91] M. Heinonen, C. Yildiz, H. Mannerström, J. Intosalmi, and H. Lähdesmäki. "Learning unknown ODE models with Gaussian processes". *Proceedings of the 35th International Conference on Machine Learning*. Volume 80. Proceedings of Machine Learning Research. 2018.

[92] P. Hennig, M. A. Osborne, and M. Girolami. "Probabilistic numerics and uncertainty in computations". *Journal of the Royal Society A* 471.2179 (2015).

[93] P. Hennig, M. A. Osborne, and H. Kersting. "Probabilistic Numerics: Computation as Machine Learning". *Cambridge University Press*, 2022.

[94] G. E. Hinton and D. van Camp. "Keeping the Neural Networks Simple by Minimizing the Description Length of the Weights". *Conference on Computational Learning Theory*. 1993, 5–13.

[95] M. D. Hoffman, A. Gelman, et al. "The No-U-Turn sampler: adaptively setting path lengths in Hamiltonian Monte Carlo." *J. Mach. Learn. Res.* 15.1 (2014), pages 1593–1623.

[96] K. Hornik. "Approximation capabilities of multilayer feedforward networks". *Neural networks* 4.2 (1991), pages 251–257.

[97] K. Hornik, M. Stinchcombe, H. White, et al. "Multilayer feedforward networks are universal approximators." *Neural networks* 2.5 (1989), pages 359–366.

[98] J. D. Hunter. "Matplotlib: A 2D graphics environment". *Computing in Science & Engineering* 9.3 (2007), pages 90–95.

[99] A. Immer, M. Korzepa, and M. Bauer. "Improving predictions of Bayesian neural nets via local linearization". *AISTATS*. 2021.

[100] P. Izmailov, S. Vikram, M. D. Hoffman, and A. G. Wilson. "What Are Bayesian Neural Network Posteriors Really Like?" *International Conference on Machine Learning*. 2021.

[101] R. Jagadeeswaran and F. J. Hickernell. "Fast automatic Bayesian cubature using lattice sampling". *Statistics and Computing* 29.6 (2019), pages 1215–1229.

[102] J. Jumper, R. Evans, A. Pritzel, T. Green, M. Figurnov, O. Ronneberger, K. Tunyasuvunakool, R. Bates, A. Žídek, A. Potapenko, et al. "Highly accurate protein structure prediction with AlphaFold". *Nature* 596.7873 (2021), pages 583–589.

[103] M. Kanagawa and P. Hennig. “Convergence guarantees for adaptive Bayesian quadrature methods”. *Neural Information Processing Systems*. 2019, pages 6237–6248. eprint: 1905.10271.

[104] M. Kanagawa, B. K. Sriperumbudur, and K. Fukumizu. “Convergence analysis of deterministic kernel-based quadrature rules in misspecified settings”. *Foundations of Computational Mathematics* 20 (2020), pages 155–194.

[105] T. Karvonen, C. J. Oates, and M. Girolami. “Integration in reproducing kernel Hilbert spaces of Gaussian kernels”. *Mathematics of Computation* 90.331 (2020), pages 2209–2233.

[106] T. Karvonen and S. Särkkä. “Fully symmetric kernel quadrature”. *SIAM Journal on Scientific Computing* 40.2 (2018), pages 697–720.

[107] M. E. E. Khan, A. Immer, E. Abedi, and M. Korzepa. “Approximate Inference Turns Deep Networks into Gaussian Processes”. *Advances in Neural Information Processing Systems*. Volume 32. 2019.

[108] P. Kidger, J. Foster, X. C. Li, and T. Lyons. “Efficient and Accurate Gradients for Neural SDEs”. *Advances in Neural Information Processing Systems*. Volume 34. *Curran Associates, Inc.*, 2021, pages 18747–18761.

[109] P. Kidger, J. Morrill, J. Foster, and T. Lyons. “Neural Controlled Differential Equations for Irregular Time Series”. *Advances in Neural Information Processing Systems*. Volume 33. 2020.

[110] D. P. Kingma and J. L. Ba. “Adam: A method for stochastic optimization”. *International Conference on Learning Representations*. 2015.

[111] A. Kirby, F.-X. Briol, T. D. Dunstan, and T. Nishino. “Data-driven modelling of turbine wake interactions and flow resistance in large wind farms”. *arXiv:2301.01699* (2023).

[112] A. Kirby, T. Nishino, and T. D. Dunstan. “Two-scale interaction of wake and blockage effects in large wind farms”. *arXiv:2207.03148* (2022).

[113] B. Kleinberg, Y. Li, and Y. Yuan. “An alternative view: When does SGD escape local minima?” *International conference on machine learning*. PMLR. 2018, pages 2698–2707.

[114] L. Kong, J. Sun, and C. Zhang. “SDE-Net: Equipping Deep Neural Networks with Uncertainty Estimates”. *Proceedings of the 37th International Conference on Machine Learning*. *PMLR*, 2020.

[115] A. S. Krishnapriyan, A. F. Queiruga, N. B. Erichson, and M. W. Mahoney. “Learning continuous models for continuous physics”. *arXiv preprint arXiv:2202.08494* (2022).

[116] A. Kristiadi, M. Hein, and P. Hennig. “Being Bayesian, Even Just a Bit, Fixes Overconfidence in ReLU Networks”. *Proceedings of the 37th International Conference on Machine Learning*. Volume 119. Proceedings of Machine Learning Research. 2020.

[117] A. Krizhevsky, I. Sutskever, and G. E. Hinton. “Imagenet classification with deep convolutional neural networks”. *Advances in neural information processing systems*. 2012, pages 1097–1105.

[118] B. Lakshminarayanan, A. Pritzel, and C. Blundell. “Simple and Scalable Predictive Uncertainty Estimation using Deep Ensembles”. *Neural Information Processing Systems*. 2017.

[119] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. “Gradient-based learning applied to document recognition”. *Proceedings of the IEEE* 86.11 (1998), pages 2278–2324.

[120] H. Li, Z. Xu, G. Taylor, C. Studer, and T. Goldstein. “Visualizing the Loss Landscape of Neural Nets”. *Advances in Neural Information Processing Systems*. Volume 31. 2018.

[121] Q. Li, T. Lin, and Z. Shen. “Deep Learning via Dynamical Systems: An Approximation Perspective”. *arXiv preprint arXiv:1912.10382* (2019).

[122] X. Li, T.-K. L. Wong, R. T. Q. Chen, and D. Duvenaud. “Scalable Gradients for Stochastic Differential Equations”. *Proceedings of the Twenty Third International Conference on Artificial Intelligence and Statistics*. Volume 108. Proceedings of Machine Learning Research. 2020.

[123] H. Lin and S. Jegelka. “ResNet with one-neuron hidden layers is a Universal Approximator”. *Advances in Neural Information Processing Systems 31*. 2018, pages 6169–6178.

[124] D. C. Liu and J. Nocedal. “On the limited memory BFGS method for large scale optimization”. *Mathematical programming* 45.1 (1989), pages 503–528.

[125] L. Liu, X. Liu, J. Gao, W. Chen, and J. Han. “Understanding the difficulty of training transformers”. *arXiv preprint arXiv:2004.08249* (2020).

[126] X. Liu, T. Xiao, S. Si, Q. Cao, S. Kumar, and C.-J. Hsieh. “Neural sde: Stabilizing neural ode networks with stochastic noise”. *arXiv preprint arXiv:1906.02355* (2019).

[127] S. Lloyd, R. A. Irani, and M. Ahmadi. “Using Neural Networks for Fast Numerical Integration and Optimization”. *IEEE Access* 8 (2020), pages 84519–84531.

[128] J. Lu, K. Deng, X. Zhang, G. Liu, and Y. Guan. “Neural-ODE for pharmacokinetics modeling and its advantage to alternative machine learning models in predicting new dosing regimens”. *Iscience* 24.7 (2021), page 102804.

[129] Y. Lu, C. Ma, Y. Lu, J. Lu, and L. Ying. “A mean-field analysis of deep resnet and beyond: Towards provable optimization via overparameterization from depth”. *arXiv preprint arXiv:2003.05508* (2020).

[130] Y. Lu, A. Zhong, Q. Li, and B. Dong. “Beyond Finite Layer Neural Networks: Bridging Deep Architectures and Numerical Differential Equations”. *Proceedings of the 35th International Conference on Machine Learning*. Volume 80. 2018, pages 3276–3285.

[131] W. Maass. “Networks of spiking neurons: the third generation of neural network models”. *Neural networks* 10.9 (1997), pages 1659–1671.

[132] D. J. C. Mackay. “Probable networks and plausible predictions — a review of practical Bayesian methods for supervised neural networks”. *Network: Computation in Neural Systems* (1995).

[133] D. J. MacKay. “A practical Bayesian framework for backpropagation networks”. *Neural computation* 4.3 (1992), pages 448–472.

[134] J. Martens. “Deep learning via Hessian-free optimization”. *International Conference on Machine Learning*. 2010, pages 735–742.

[135] J. Martens and R. Grosse. “Optimizing neural networks with kronecker-factored approximate curvature”. *International conference on machine learning*. PMLR. 2015, pages 2408–2417.

[136] S. Massaroli, M. Poli, M. Bin, J. Park, A. Yamashita, and H. Asama. “Stable Neural Flows”. *arXiv preprint arXiv:2003.08063* (2020).

[137] S. Massaroli, M. Poli, J. Park, A. Yamashita, and H. Asama. “Dissecting Neural ODEs”. *Advances in Neural Information Processing Systems*. Volume 33. *Curran Associates, Inc.*, 2020, pages 3952–3963.

[138] N. Metropolis, A. W. Rosenbluth, M. N. Rosenbluth, A. H. Teller, and E. Teller. “Equation of state calculations by fast computing machines”. *The journal of chemical physics* 21.6 (1953), pages 1087–1092.

[139] R. M. Neal. “Bayesian learning for neural networks”. *Springer Verlag*, 1996.

[140] R. M. Neal et al. “MCMC using Hamiltonian dynamics”. *Handbook of Markov Chain Monte Carlo* 2.11 (2011), page 2.

[141] A. Niayifar and F. Porté-Agel. “Analytical modeling of wind farms: A new approach for power prediction”. *Energies* 9.9 (2016), pages 1–13.

[142] T. Nishino. “Two-scale momentum theory for very large wind farms”. *Journal of Physics: Conference Series* 753.3 (2016).

[143] J. Nocedal and S. J. Wright. “Numerical Optimization”. 2e. New York, NY, USA: *Springer*, 2006.

[144] A. Norcliffe, C. Bodnar, B. Day, J. Moss, and P. Liò. “Neural ODE Processes”. *International Conference on Learning Representations*. 2021.

[145] C. J. Oates, J. Cockayne, F.-X. Briol, and M. Girolami. “Convergence rates for a class of estimators based on Stein’s method”. *Bernoulli* 25.2 (2019), pages 1141–1159.

[146] C. J. Oates, M. Girolami, and N. Chopin. “Control functionals for Monte Carlo integration”. *Journal of the Royal Statistical Society B: Statistical Methodology* 79.3 (2017), pages 695–718.

[147] C. J. Oates, S. Niederer, A. Lee, F.-X. Briol, and M. Girolami. “Probabilistic models for integration error in the assessment of functional cardiac models”. *Neural Information Processing Systems*. 2017, pages 110–118.

[148] C. J. Oates, T. Papamarkou, and M. Girolami. “The controlled thermodynamic integral for Bayesian model comparison”. *Journal of the American Statistical Association* 111.514 (2016), pages 634–645.

[149] C. J. Oates and T. J. Sullivan. “A modern retrospective on probabilistic numerics”. *Statistics and Computing* 29 (2019), pages 1335–1351.

[150] A. O’Hagan. “Bayes–Hermite quadrature”. *Journal of Statistical Planning and Inference* 29 (1991).

[151] OpenAI. “GPT-4 Technical Report”. *arXiv:2303.08774* (2023).

[152] M. Osborne, R. Garnett, Z. Ghahramani, D. K. Duvenaud, S. J. Roberts, and C. Rasmussen. “Active Learning of Model Evidence Using Bayesian Quadrature”. *Advances in Neural Information Processing Systems*. Volume 25. 2012.

[153] K. Ott, P. Katiyar, P. Hennig, and M. Tiemann. “ResNet After All: Neural {ODE}s and Their Numerical Solution”. *International Conference on Learning Representations*. 2021.

[154] K. Ott, M. Tiemann, and P. Hennig. “Uncertainty and Structure in Neural Ordinary Differential Equations”. *arXiv:2305.13290* (2023).

[155] K. Ott, M. Tiemann, P. Hennig, and F.-X. Briol. “Bayesian Numerical Integration with Neural Networks”. *arXiv:2305.13248* (2023).

[156] A. B. Owen. “Monte Carlo Theory, Methods and Examples”. 2013.

[157] H. Owhadi and G. R. Yoo. “Kernel Flows: From learning kernels from data into the abyss”. *Journal of Computational Physics* 389 (2019), pages 22–47.

[158] A. Paleyes, M. Pullin, M. Mahsereci, N. Lawrence, and J. González. “Emulation of physical processes with Emukit”. *Second Workshop on Machine Learning and the Physical Sciences, NeurIPS*. 2019.

[159] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala. “PyTorch: An Imperative Style, High-Performance Deep Learning Library”. *Advances in Neural Information Processing Systems 32*. 2019, pages 8024–8035.

[160] T. Pearce, R. Tsuchida, M. Zaki, A. Brintrup, and A. Neely. “Expressive priors in Bayesian neural networks: Kernel combinations and periodic functions”. *Conference on Uncertainty in Artificial Intelligence*. 2019.

[161] M. M. Pedersen, P. van der Laan, M. Friis-Møller, J. Rinker, and P.-E. Réthoré. “DTUWindEnergy/PyWake: PyWake” (2019).

[162] D. Phan, N. Pradhan, and M. Jankowiak. “Composable Effects for Flexible and Accelerated Probabilistic Programming in NumPyro”. *arXiv preprint arXiv:1912.11554* (2019).

[163] D. B. Phillips and A. F. Smith. “Bayesian model comparison via jump diffusions”. *Markov chain Monte Carlo in practice* (1995), page 215.

[164] E. Picard. “Sur l’application des méthodes d’approximations successives à l’étude de certaines équations différentielles ordinaires”. fre. *Journal de Mathématiques Pures et Appliquées* 9 (1893), pages 217–272.

[165] M. Poli, S. Massaroli, J. Park, A. Yamashita, H. Asama, and J. Park. “Graph neural ordinary differential equations”. *arXiv preprint arXiv:1911.07532* (2019).

[166] L. S. Pontryagin. “Mathematical theory of optimal processes”. *CRC press*, 1987.

[167] G. D. Portwood, P. P. Mitra, M. D. Ribeiro, T. M. Nguyen, B. T. Nadiga, J. A. Saenz, M. Chertkov, A. Garg, A. Anandkumar, A. Dengel, et al. “Turbulence forecasting via neural ode”. *arXiv preprint arXiv:1911.05180* (2019).

[168] P. Prince and J. Dormand. “High order embedded Runge-Kutta formulae”. *Journal of Computational and Applied Mathematics* 7.1 (1981), pages 67–75.

[169] A. F. Queiruga, N. B. Erichson, D. Taylor, and M. W. Mahoney. “Continuous-in-Depth Neural Networks”. *arXiv preprint arXiv:2008.02389* (2020).

[170] C. Rackauckas, M. Innes, Y. Ma, J. Bettencourt, L. White, and V. Dixit. "DiffEqFlux.jl - A Julia Library for Neural Differential Equations". *CoRR* abs/1902.02376 (2019).

[171] M. Raissi, P. Perdikaris, and G. E. Karniadakis. "Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations". *Journal of Computational physics* 378 (2019), pages 686–707.

[172] A. Ramesh, P. Dhariwal, A. Nichol, C. Chu, and M. Chen. "Hierarchical text-conditional image generation with CLIP latents". *arXiv preprint arXiv:2204.06125* (2022).

[173] C Rasmussen and Z Ghahramani. "Bayesian Monte Carlo". *Neural Information Processing Systems*. 2002, pages 489–496.

[174] C. Rasmussen and C. Williams. "Gaussian Processes for Machine Learning". *MIT Press*, 2006.

[175] D. Rezende and S. Mohamed. "Variational inference with normalizing flows". *International conference on machine learning*. PMLR. 2015, pages 1530–1538.

[176] M. Riabiz, W. Chen, J. Cockayne, P. Swietach, S. A. Niederer, L. Mackey, and C. J. Oates. "Optimal thinning of MCMC output". *Journal of the Royal Statistical Society Series B (Statistical Methodology), to appear.* (2022).

[177] S. Ridderbusch, C. Offen, S. Ober-Blöbaum, and P. Goulart. "Learning ODE Models with Qualitative Structure Using Gaussian Processes". *arXiv preprint arXiv:2011.05364* (2021).

[178] H. Ritter, A. Botev, and D. Barber. "A Scalable Laplace Approximation for Neural Networks". *International Conference on Learning Representations*. 2018.

[179] H. Robbins and S. Monro. "A stochastic approximation method". *The annals of mathematical statistics* (1951), pages 400–407.

[180] C. P. Robert and G. Casella. "Monte Carlo Statistical Methods". *Springer*, 2000.

[181] G. O. Roberts and R. L. Tweedie. "Exponential convergence of Langevin distributions and their discrete approximations". *Bernoulli* (1996), pages 341–363.

[182] Y. Rubanova, R. T. Q. Chen, and D. K. Duvenaud. "Latent Ordinary Differential Equations for Irregularly-Sampled Time Series". *Advances in Neural Information Processing Systems*. Volume 32. 2019.

[183] W. Rudin. "Real and complex analysis (mcgraw-hill international editions: Mathematics series)" (1987).

[184] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. "Learning representations by back-propagating errors". *nature* 323.6088 (1986), pages 533–536.

[185] C. Runge. "Ueber die numerische Auflösung von Differentialgleichungen." *Mathematische Annalen* 46 (1895), pages 167–178.

[186] L. Ruthotto and E. Haber. "Deep neural networks motivated by partial differential equations". *Journal of Mathematical Imaging and Vision* (2019), pages 1–13.

[187] F. Scarselli, M. Gori, A. C. Tsoi, M. Hagenbuchner, and G. Monfardini. "The Graph Neural Network Model". *IEEE Transactions on Neural Networks* 20.1 (2009), pages 61–80.

[188] N. N. Schraudolph. "Fast Curvature Matrix-Vector Products for Second-Order Gradient Descent". *Neural Computation* 14.7 (2002), pages 1723–1738.

[189] J. Shi, Y. Zhou, J. Hwang, M. K. Titsias, and L. Mackey. "Gradient estimation with discrete Stein operators". *Neural Information Processing Systems*. 2022.

[190] S. Si, C. J. Oates, A. B. Duncan, L. Carin, and F.-X. Briol. "Scalable control variates for Monte Carlo methods via stochastic optimization". *Proceedings of the 14th Conference on Monte Carlo and Quasi-Monte Carlo Methods. arXiv:2006.07487* (2021).

[191] K. Simonyan and A. Zisserman. "Very deep convolutional networks for large-scale image recognition". *arXiv preprint arXiv:1409.1556* (2014).

[192] S. Smith, E. Elsen, and S. De. "On the generalization benefit of noise in stochastic gradient descent". *International Conference on Machine Learning*. PMLR. 2020, pages 9058–9067.

[193] I. Sobol'. "On the distribution of points in a cube and the approximate evaluation of integrals". *USSR Computational Mathematics and Mathematical Physics* 7.4 (1967), pages 86–112.

[194] Y. Song, J. Sohl-Dickstein, D. P. Kingma, A. Kumar, S. Ermon, and B. Poole. "Score-Based Generative Modeling through Stochastic Differential Equations". *International Conference on Learning Representations*. 2021.

[195] S. Sonoda and N. Murata. "Transport Analysis of Infinitely Deep Neural Network". *Journal of Machine Learning Research* 20.2 (2019), pages 1–52.

[196] A. H. Stroud. "Approximate calculation of multiple integrals". Prentice-Hall series in automatic computation. Englewood Cliffs, NJ: *Prentice-Hall*, 1971.

[197] S. Sun, G. Zhang, J. Shi, and R. Grosse. "Functional variational Bayesian neural networks". *International Conference on Learning Representations*. 2019.

[198] M. Thorpe and Y. van Gennip. "Deep limits of residual neural networks". *arXiv preprint arXiv:1810.11741* (2018).

[199] F. Tronarp, N. Bosch, and P. Hennig. "Fenrir: Physics-Enhanced Regression for Initial Value Problems". *Proceedings of the 39th International Conference on Machine Learning*. Edited by K. Chaudhuri, S. Jegelka, L. Song, C. Szepesvari, G. Niu, and S. Sabato. Volume 162. Proceedings of Machine Learning Research. *PMLR*, 2022, pages 21776–21794.

[200] B. Tzen and M. Raginsky. "Neural stochastic differential equations: Deep latent gaussian models in the diffusion limit". *arXiv preprint arXiv:1905.09883* (2019).

[201] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin. "Attention is all you need". *Advances in neural information processing systems* 30 (2017).

[202] A. Veit and S. Belongie. "Convolutional Networks with Adaptive Inference Graphs". *The European Conference on Computer Vision (ECCV)*. 2018.

[203] P. Virtanen, R. Gommers, T. E. Oliphant, M. Haberland, T. Reddy, D. Cournapeau, E. Burovski, P. Peterson, W. Weckesser, J. Bright, S. J. van der Walt, M. Brett, J. Wilson, K. J. Millman, N. Mayorov, A. R. J. Nelson, E. Jones, R. Kern, E. Larson, C. J. Carey, İ. Polat, Y. Feng, E. W. Moore, J. VanderPlas, D. Laxalde, J. Perktold, R. Cimrman, I. Henriksen, E. A. Quintero, C. R. Harris, A. M. Archibald, A. H. Ribeiro, F. Pedregosa, P. van Mulbregt, and SciPy 1.0 Contributors. "SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python". *Nature Methods* 17 (2020), pages 261–272.

[204] R. Wan, M. Zhong, H. Xiong, and Z. Zhu. "Neural Control Variates for Monte Carlo Variance Reduction". *Machine Learning and Knowledge Discovery in Databases*. *Springer International Publishing*, 2020.

[205] Z. Wang, W. Xing, R. Kirby, and S. Zhe. "Physics Regularized Gaussian Processes". *arXiv preprint arXiv:2006.04976* (2020).

[206] E Weinan, J. Han, and Q. Li. "A mean-field optimal control formulation of deep learning". *Research in the Mathematical Sciences* 6.1 (2019), page 10.

[207] J. Wenger, N. Krämer, M. Pförtner, J. Schmidt, N. Bosch, N. Effenberger, J. Zenn, A. Gessner, T. Karvonen, F.-X. Briol, M. Mahsereci, and P. Hennig. "ProbNum: Probabilistic numerics in Python". *arXiv:2112.02100* (2021). eprint: `2112.02100`.

[208] O. Wieder, S. Kohlbacher, M. Kuenemann, A. Garon, P. Ducrot, T. Seidel, and T. Langer. "A compact review of molecular property prediction with graph neural networks". *Drug Discovery Today: Technologies* 37 (2020), pages 1–12.

[209] A. G. Wilson and P. Izmailov. "Bayesian Deep Learning and a Probabilistic Perspective of Generalization". *Neural Information Processing Systems*. 2020.

[210] G. Wynne, F.-X. Briol, and M. Girolami. "Convergence guarantees for Gaussian process means with misspecified likelihoods and smoothness". *Journal of Machine Learning Research* 22.123 (2021), pages 1–40.

[211] X. Xi, F.-X. Briol, and M. Girolami. "Bayesian quadrature for multiple related integrals". *35th International Conference on Machine Learning*. Volume 12. 2018, pages 8533–8564.

[212] H. Xia, V. Suliafu, H. Ji, T. Nguyen, A. Bertozzi, S. Osher, and B. Wang. "Heavy Ball Neural Ordinary Differential Equations". *Advances in Neural Information Processing Systems*. Volume 34. *Curran Associates, Inc.*, 2021, pages 18646–18659.

[213] S. Xie, R. Girshick, P. Dollár, Z. Tu, and K. He. "Aggregated residual transformations for deep neural networks". *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2017, pages 1492–1500.

[214] K. Xu, M. Zhang, J. Li, S. S. Du, K.-I. Kawarabayashi, and S. Jegelka. "How Neural Networks Extrapolate: From Feedforward to Graph Neural Networks". *International Conference on Learning Representations*. 2021.

[215] M. Xu, S. Luo, Y. Bengio, J. Peng, and J. Tang. "Learning neural generative dynamics for molecular conformation generation". *arXiv preprint arXiv:2102.10240* (2021).

[216] H. Yan, J. Du, V. Tan, and J. Feng. "On Robustness of Neural Ordinary Differential Equations". *International Conference on Learning Representations*. 2020.

[217] L. Yang, X. Meng, and G. E. Karniadakis. "B-PINNs: Bayesian physics-informed neural networks for forward and inverse PDE problems with noisy data". *Journal of Computational Physics* (2021).

[218] Y. Yang, J. Wu, H. Li, X. Li, T. Shen, and Z. Lin. "Dynamical System Inspired Adaptive Time Stepping Controller for Residual Network Families". *Thirty-Fourht AAAI Conference on Artificial Intelligence*. 2020.

[219] C. Yildiz, M. Heinonen, and H. Lahdesmaki. "ODE2VAE: Deep generative second order ODEs with Bayesian neural networks". *Advances in Neural Information Processing Systems*. Volume 32. 2019.

[220] Y. Yin, V. L. Guen, J. Dona, E. de Bezénac, I. Ayed, N. Thome, and P. Gallinari. "Augmenting Physical Models with Deep Networks for Complex Dynamics Forecasting". *International Conference on Learning Representations*. 2021.

[221] M. Yousef, K. F. Hussain, and U. S. Mohammed. "Accurate, data-efficient, unconstrained text recognition with convolutional neural networks". *Pattern Recognition* 108 (2020), page 107482.

[222] S. Zagoruyko and N. Komodakis. "Wide residual networks". *arXiv preprint arXiv:1605.07146* (2016).

[223] H. Zhang, X. Gao, J. Unterman, and T. Arodz. "Approximation capabilities of neural ordinary differential equations". *arXiv preprint arXiv:1907.12998* (2019).

[224] J. Zhang, B. Han, L. Wynter, B. K. H. Low, and M. Kankanhalli. "Towards Robust ResNet: A Small Step but a Giant Leap". *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI-19*. 2019, pages 4285–4291.

[225] T. Zhang, Z. Yao, A. Gholami, J. E. Gonzalez, K. Keutzer, M. W. Mahoney, and G. Biros. "ANODEV2: A Coupled Neural ODE Framework". *Advances in Neural Information Processing Systems 32*. 2019, pages 5151–5161.

[226] Y. D. Zhong, B. Dey, and A. Chakraborty. "Dissipative SymODEN: Encoding Hamiltonian Dynamics with Dissipation and Control into Deep Learning". *ICLR 2020 Workshop on Integration of Deep Neural Models and Differential Equations*. 2020.

[227] Y. D. Zhong, B. Dey, and A. Chakraborty. "Symplectic ODE-Net: Learning Hamiltonian Dynamics with Control". *International Conference on Learning Representations*. 2020.

[228] Y. Zhou, S. Liu, J. Siow, X. Du, and Y. Liu. "Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks". *Advances in neural information processing systems* 32 (2019).

[229] H. Zhu, X. Liu, R. Kang, Z. Shen, S. Flaxman, and F.-X. Briol. "Bayesian probabilistic numerical integration with tree-based models". *Neural Information Processing Systems*. 2020, pages 5837–5849.

[230] J. Zhuang, N. Dvornek, X. Li, S. Tatikonda, X. Papademetris, and J. Duncan. "Adaptive Checkpoint Adjoint Method for Gradient Estimation in Neural ODE". *arXiv preprint arXiv:2006.02493* (2020).

[231] L. Ziyin, T. Hartwig, and M. Ueda. "Neural Networks Fail to Learn Periodic Functions and How to Fix It". *Neural Information Processing Systems*. Edited by H. Larochelle, M. Ranzato, R. Hadsell, M. Balcan, and H. Lin. Volume 33. *Curran Associates, Inc.*, 2020, pages 1583–1594.