Distribution Agreement

In presenting this thesis or dissertation as a partial fulfillment of the requirements for an advanced degree from Emory University, I hereby grant to Emory University and its agents the non-exclusive license to archive, make accessible, and display my thesis or dissertation in whole or in part in all forms of media, now or hereafter known, including display on the world wide web. I understand that I may select some access restrictions as part of the online submission of this thesis or dissertation. I retain all ownership rights to the copyright of the thesis or dissertation. I also retain the right to use in future works (such as articles or books) all or part of this thesis or dissertation.

Signature:

Derek Onken Date

Optimal Control Approaches for Designing Neural Ordinary Differential Equations

By

Derek Onken
Doctor of Philosophy

Computer Science and Informatics

Lars Ruthotto, Ph.D.
Advisor

Rachel Jennings, Ph.D.
Committee Member

James G. Nagy, Ph.D.
Committee Member

Yuanzhe Xi, Ph.D.
Committee Member

Accepted:

Lisa A. Tedesco, Ph.D.
Dean of the James T. Laney School of Graduate Studies

Date

Optimal Control Approaches for Designing Neural Ordinary Differential Equations

By

Derek Onken
M.Sc., Emory University, GA, 2019
B.Sc., The University of Georgia, GA, 2015

Advisor: Lars Ruthotto, Ph.D.

An abstract of
A dissertation submitted to the Faculty of the
James T. Laney School of Graduate Studies of Emory University
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy
in Computer Science and Informatics
2021

Abstract

Optimal Control Approaches for Designing Neural Ordinary Differential Equations
By Derek Onken

Neural network design encompasses both model formulation and numerical treatment for optimization and parameter tuning. Recent research in formulation focuses on interpreting architectures as discretizations of ordinary differential equations (ODEs). These neural ODEs, in which the ODE dynamics are defined by neural network components, benefit from reduced parameterization and smoother hidden states than traditional discrete neural networks but come at high computational costs. Training a neural ODE can be phrased as an ODE-constrained optimization problem, which allows for the application of mathematical optimal control (OC). The application of OC theory leads to design choices that differ from popular high-cost implementations. We improve neural ODE numerical treatment and formulation for models used in time-series regression, image classification, continuous normalizing flows, and path-finding problems.

Optimal Control Approaches for Designing Neural Ordinary Differential Equations

By

Derek Onken
M.Sc., Emory University, GA, 2019
B.Sc., The University of Georgia, GA, 2015

Advisor: Lars Ruthotto, Ph.D.

A dissertation submitted to the Faculty of the
James T. Laney School of Graduate Studies of Emory University
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy
in Computer Science and Informatics
2021

Acknowledgments

I am forever grateful to the people who have helped me in life to arrive at this point. Most directly, I would like to thank my advisor Lars Ruthotto for guiding me through this work and broader research methodologies. His enthusiasm and mentorship have helped me grow as a critical scientist, and his direction has shaped my technical skills in a unique and interdisciplinary space. I would also like to thank my committee members: Jim Nagy, Yuanzhe Xi, and Rachel Jennings. Their insights and encouragement have helped shape this dissertation and my growth during my graduate school experience. Furthermore, I would like to thank Rachel for her guidance in my internships and the opportunities she afforded me.

My journey has been positively influenced by research experiences with numerous individuals: Levon Nurbekyan, Xingjian Li, Elizabeth Newman, Juan B. Gutiérrez, Ted Shifrin, Stephen Garth, Cameron Peterson, Daniel Cowan, Heather Pangburn, and Thorgeir Karlsson. Without their help, my successes would not have been so.

I am also fortunate to have shared these years with incredible friends and peers: Thanks to Reza Karimi, Pranav Bhandari, Sapoon Dutta, Kelvin Kan, Samy Wu Fung, Zelalem Gero, Harshita Sahijwani, Gary Lai, Blair Rosetti, Abbey Julian, Malvern Madondo, Radhika Sharma, Misael Romero-Reyes, and Jeffrey Davis. Without them, my time in graduate school would not have been as enjoyable. I would also like to thank my past coaches: Yit Aun Lim, Anthony Byrne, Katrin Koch, Harvey Humphries, and Jack Bauerle. Their unwavering support and guidance have helped me develop the skills necessary to undertake graduate school.

Most importantly, I would like to thank my family. Thank you to my brothers, Drew and Decker, for their support and for listening to me talk about my research ad nauseum. This dissertation is dedicated to my parents, Rich and Pat, for their immense sacrifices, love, and support throughout. Their work ethic and insightfulness have made me the person I am today.

Contents

1 Introduction **1**

1.1 Contribution 2

1.1.1 Efficient Numerical Treatment 2

1.1.2 Formulation 4

1.2 Overview 5

2 Mathematical Background **6**

2.1 Neural Networks 6

2.1.1 Architecture 6

2.1.2 Training 8

2.1.3 Concepts 8

2.2 Neural ODEs 11

2.3 Optimal Control 12

2.3.1 Pontryagin Maximum Principle 14

2.3.2 Hamilton-Jacobi-Bellman PDE 14

2.4 Learning and Optimal Control 16

2.5 Neural ODEs as Reinforcement Learning 17

3 Time-Series Regression **19**

3.1 Problem 19

3.2 Discretize-Optimize vs. Optimize-Discretize 20

3.3 Numerical Experiments 24

3.3.1 Extrapolation and Different Initial Conditions 27

4 Image Classification **29**

4.1 Problem 29

4.1.1 Focal Loss 31

4.2 Convolutional Neural Networks 32

4.2.1 Convolutional Layer 32

4.2.2 Normalization Layer 34

4.2.3 Double Symmetric Layer 35

4.3 Decoupling the Weights and Layers 35

4.4 Image Classification for Lung Cancer Detection 37

4.4.1 Motivation 37

4.4.2 Model 39

4.4.3 National Lung Screening Trial Experiment 39

5 Continuous Normalizing Flows for Density Estimation **42**

5.1 Problem 42

5.2 Related Works 46

5.2.1 Finite Flows 47

5.2.2 Infinitesimal Flows 47

5.2.3 Flows Influenced by Optimal Control 49

5.3 Discretize-Optimize Flows 49

5.3.1 Numerical Experiments 51

5.4 OT-Flow 60

5.4.1 Model Formulation 61

5.4.2 Implementation 66

5.4.3 Numerical Experiments 71

6 Path-Finding **77**

6.1 Problem 77

6.2 Related Works 79

6.2.1 High-Dimensional Deterministic Optimal Control 79

6.2.2 High-Dimensional Stochastic Optimal Control 80

6.2.3 Multi-Agent Path-Finding 81

6.3 Neural ODE Formulation 82

6.3.1 Main Formulation 82

6.3.2 Adding Hamilton-Jacobi-Bellman Penalizers 83

6.3.3 Robustness to Shocks 84

6.4 Numerics 85

6.4.1 Hyperparameter Tuning 86

6.5 Numerical Experiments 88

6.5.1 Baseline: Discrete Optimization for a Single Initial State . . . 89

6.5.2 Two-Agent Corridor Experiment 90

6.5.3 Effect of the Hamilton-Jacobi-Bellman Penalizers 92

6.5.4 Multi-Agent Swap Experiments 98

6.5.5 Swarm Problem 101

6.5.6 Quadcopter Experiment 102

7 Summary **107**

Appendix A Derivation of Adjoint Equations **110**

A.1 Continuous Adjoint 110

A.2 Discrete Adjoints 112

Appendix B Bootstrapping **114**

Bibliography **115**

Mathematical Symbols **130**

Index **134**

List of Figures

3.1 DO and OD training iterations 23
3.2 DO and OD convergence 24
3.3 Time-series regression derivative check 25
3.4 Time-series regression extrapolation. 26
3.5 Different initial parameterizations 27

4.1 Pooling examples 30
4.2 Image classification model example 36
4.3 Lung cancer image classification. 37
4.4 Lung cancer model applied to individual cubes. 38
4.5 Model interpretability for deployment. 41

5.1 Normalizing flow example for the Gaussian mixture problem. 43
5.2 Performance of our exact trace and Hutchinson's estimator using AD 49
5.3 Illustration of the DO approach for the Gaussian mixture problem . . 51
5.4 Problems when choosing a coarse discretization. 55
5.5 Training CNFs on Miniboone with and without multilevel. 60
5.6 OT-Flow enforces straight trajectories. 61
5.7 Effect of adding an HJB penalizer during training 65
5.8 Comparison of OT-Flow (exact trace) and an identical model using the Hutchinson's estimator. 70

5.9 Power density estimation using a discrete flow 72
5.10 OT-Flow density estimation on 2-D toy problems. 73
5.11 Miniboone density estimation 2-D slices. 74
5.12 MNIST generation conditioned by class. 76
5.13 MNIST interpolation in the latent space. 76

6.1 Two-agent corridor problem. 90
6.2 Effect of the terminal HJB penalizers. 93
6.3 Shock-Robustness 94
6.4 Comparable neural network trained with no penalizers. 95
6.5 Comparison of one NN model with 10,001 baseline models. 97
6.6 Swap experiment with hard-boundary corridor. 98
6.7 Swap experiment for 12 agents in $\mathbb{R}^2$. 98
6.8 Overcoming CoD 100
6.9 Swarm problem for 50 agents in $\mathbb{R}^3$. 101
6.10 Quadcopter diagram 102
6.11 Quadcopter problem comparison with baseline. 105
6.12 Quadcopter comparison of states. 106

List of Tables

4.1 Model performance 2×2 tables. 40
4.2 Model performance metrics. 40

5.1 CNF methods comparison 50
5.2 Number of parameters comparison with discrete normalizing flows. . . 50
5.3 DO results for the Gaussian mixture problem 54
5.4 DO and OD hyperparameters. 56
5.5 Density estimation using DO and OD. 57
5.6 Testing loss C comparison with other methods. 58
5.7 Re-discretization of the trained CNF in the inference phase. 59
5.8 OT-Flow density estimation on real data sets 75

6.1 Neural network training statistics. 86
6.2 Problem variables and neual network hyperparameters. 87
6.3 Corridor problem comparison of baseline and NN. 94

List of Algorithms

1 Neural network training 9

Chapter 1

Introduction

Neural networks (NNs), compositions of many simple nonlinear and parameterized linear functions, have become a state-of-the-art machine learning tool [90, 91, 116]. Simultaneously, neural networks have become over-parameterized, complicated discrete objects that empirically produce good results on specific intended tasks. Based on the popular, successful residual neural network (ResNet) [45], neural ordinary differential equations (ODEs) [19, 25, 41] present a continuous interpretation of neural networks. The continuous approach benefits from a smooth parameterized governing equation which is appealing for fluid mechanics [16], Boltzmann machines [73], density estimation [13, 39], particle physics [14], and solving partial differential equations (PDEs) [85].

Although appealing, neural ODEs are young. The field lacks maturity, and many paramount works in the space demonstrate the exciting capabilities of neural ODEs that may benefit from polishing and improvements. Many applications can benefit from their use, but their implementation can come at high computational cost, among other concerns. High costs stem from the high-dimensional state ($d \geq 4$) of the ODEs present in the applications and the thousands to millions of parameters in a single neural ODE. When tuning these parameters, NN approaches often use a momentum-based stochastic gradient descent method, e.g., ADAM [52], to search this high-dimensional space. Ultimately, the many parameters and the stochasticity of the optimization scheme result in a time-consuming tuning regime that is not easily parallelizable.

We strive to leverage optimal control (OC) to improve neural ODEs, focusing on the design of neural ODEs in several applications. In neural ODEs, the ODE dynamics are defined by neural network components, and training the network presents an optimization problem over the parameters and constrained by the ODE. This form fits nicely into an OC interpretation in which the parameters are the *controls* and the neural network features (or ODE solutions) are the *states*.

From the OC lens, we address the two aspects of neural network design: numerical treatment (or learning rules) and formulation (or architecture) [42]. We apply mathematically sound methods for efficient optimization and novel architecture.

1.1 Contribution

We focus on the applications of neural ODEs for time-series regression, image classification, continuous normalizing flows (CNFs), and path-finding problems. Our two main thrusts address improvements for neural ODE design by: incorporating efficient numerical treatment—which improves training speed and increases the model's high-dimensional scalability—and using the OC value function to improve the neural ODE formulation.

1.1.1 Efficient Numerical Treatment

Training the neural ODE parameters is a constrained stochastic optimization problem. Popular existing neural ODE implementations [19, 39, 84] choose the optimize-discretize (OD) approach for training. However, for image classification, the discretize-optimize (DO) approach has been shown to converge faster due to accurate gradient computation [36]. We extend this analysis to time-series regression (Chapter 3) and CNFs (Section 5.3). These problems contain additional constraints on the hidden states which can require specific treatment not relevant to the image classification task. We empirically observe 6x–20x speedup in training time from training with the DO approach.

Whereas the DO approach has already been applied to image classification [41, 45, 114], we investigate an interpolation scheme for the neural ODE parameters, which is an alternative to the more common neural ODE design where neural network components operate on time itself [19]. We apply our resultant model on three-dimensional medical images (Chapter 4). Medical image classification tasks must be able to handle limited quantities of data where each data point is high-dimensional (a three-dimensional image). Neural ODEs benefit from reduced parameterization relative to existing discrete approaches and often require less training data. We compare our neural ODE approach to the performance of radiologists and existing methods [3, 62].

CNFs are bottlenecked by a trace computation (Section 5.1). State-of-the-art methods reduce the cost by estimating the trace with Hutchinson's estimator [49] during training [32, 39, 119]. This design choice leads to slow convergence with respect to number of optimization steps. We instead develop a method for an efficient exact trace computation made possible by our model formulation (Section 5.4). We demonstrate the competitive speed and scalability of our exact trace computation with the Hutchinson's estimator implemented with automatic differentiation (AD).

While our scalable trace computation helps train CNFs efficiently, other problems such as multi-agent path-finding do not reap these benefits but still suffer from the curse of dimensionality (CoD) [7]. The path-finding problems require solving a high-dimensional PDE know as the Hamilton-Jacobi-Bellman (HJB) equations (Section 2.3). Traditional numerical methods for HJB equations, such as ENO/WENO [79], employ grid-discretization and interpolation for off-grid points. These grid-based methods scale poorly due to CoD, which we mitigate using Lagrangian coordinates [74, 93] (Chapter 6).

1.1.2 Formulation

For the CNF and path-finding tasks, we use OC theory to formulate and regularize the neural ODEs. For CNFs, adding the arclength of the trajectories straightens the dynamics and reduces training costs [32]. The inclusion of the arclength also formulates the problem as an optimal transport problem [67], which fits inside the OC framework. Thus, we are guaranteed the existence of a scalar value function (potential function) [29, 33]. Similar to particles in a physical system minimizing their potential energy, samples (or agents) move in a manner to minimize this value function. The solution to the HJB equations [7, 29, 33], the value function can be used to compute directions of optimal trajectory [118, 119]. We convert one of these equations into a penalizer to help train the CNF, which introduces no bias but allows for fewer time steps and reduced CNF training cost.

For path-finding problems (Chapter 6), we extend on the CNF formulation. In CNFs, we only used the intermediate-time HJB equation for the penalizer and were unable to employ the final-time condition since the problem formulation does not allow it. However, in solving multi-agent path-finding, we develop two more penalizers out of the final-time HJB condition and empirically show their effectiveness in training neural ODEs. From using the value function and the HJB penalizers, we train neural ODEs to model obstacle avoidance and swarm problems. Furthermore, including the value function results in the model's ability to extrapolate beyond the training space and to handle shocks to the system.

1.2 Overview

This dissertation is organized as follows. In Chapter 2, we present the mathematical background relevant to neural ODEs and OC. We review the original formulation of the continuous form of the ResNet. Most importantly, we present the connection between training a neural ODE and OC. In Chapter 3, we give a first glimpse of a simple neural ODE for the time-series use case. We compare the DO and OD approaches for training neural ODEs. We present the mathematical differences and demonstrate where OD leads to inaccurate gradients and slow training. In Chapter 4, we combine many of these neural ODEs at various resolutions for solving image classification problems. We leverage the reduced parameterization of neural ODEs to develop a model capable of achieving state-of-the-art performance in lung cancer detection. In Chapter 5, we address neural ODE design for solving CNFs. We use our knowledge of the DO and OD comparison to reduce the CNF costs of state-of-the-art approaches. Recognizing space for further improvement, we apply theoretical and numerical design principles to produce our model OT-Flow, which we rigorously compare to the state-of-the-art. We recognize that OT-Flow can solve other tasks, and in Chapter 6, we adapt it for solving high-dimensional path-finding problems. We conclude with a summary and discussion of future work in Chapter 7.

Chapter 2

Mathematical Background

We present the mathematical background relevant for this dissertation. First, we briefly describe the basics of discrete neural networks. Then, we define neural ODEs, review OC background, and establish their connections.

2.1 Neural Networks

The hallmark tool in the machine learning toolbox, a neural network (NN) is merely a composition of simple parameterized functions. Typically, this composition strings together matrix multiplications and element-wise nonlinear representations known as *activation functions*.

2.1.1 Architecture

For example, consider a single layer NN known as a *dense layer*

$$\boldsymbol{D}(\boldsymbol{x};\boldsymbol{\theta}) = \sigma(\boldsymbol{K}\boldsymbol{x} + \boldsymbol{b}), \tag{2.1}$$

for input data $\boldsymbol{x} \in \mathbb{R}^d$. This layer has parameters (weights): dense matrix $\boldsymbol{K} \in \mathbb{R}^{m\times d}$ and bias vector $\boldsymbol{b} \in \mathbb{R}^m$. The vector $\boldsymbol{\theta}$ holds all the parameters; thus, in this example, we denote $\boldsymbol{\theta} = (\boldsymbol{K}, \boldsymbol{b})$ to represent vector $\boldsymbol{\theta} \in \mathbb{R}^{m(d+1)}$. Activation functions $\sigma\colon \mathbb{R}^i \to \mathbb{R}^i$, for $i \in \mathbb{Z}^+$, are the simple element-wise pieces that add nonlinearity to the model. By element-wise, we mean that we define $\sigma\colon \mathbb{R} \to \mathbb{R}$ for the $i = 1$ case and apply it to each element of the i-dimensional input vector. Activation functions often take the form of hyperbolic tangent, sigmoid, or rectified linear units (ReLU), i.e., $\sigma(z) = \tanh(z)$, $\sigma(z) = 1/(1 + \exp(-z))$, or $\sigma(z) = \max(z, 0)$, respectively.

Composition of layers similar to (2.1) increases NN complexity where composition refers to the operation in the classic functional sense. Consider a two-layer NN, or *double layer*,

$$\begin{aligned} f(\boldsymbol{x}; \boldsymbol{\theta}) &= \boldsymbol{D}_2 \circ \boldsymbol{D}_1(\boldsymbol{x}) \\ &= \sigma_2\big(\boldsymbol{K}_2\, \sigma_1(\boldsymbol{K}_1 \boldsymbol{x} + \boldsymbol{b}_1) + \boldsymbol{b}_2\big), \end{aligned} \tag{2.2}$$

where $\boldsymbol{D}_1, \boldsymbol{D}_2$ are both dense layers. The parameters $\boldsymbol{\theta} = (\boldsymbol{K}_1, \boldsymbol{K}_2, \boldsymbol{b}_1, \boldsymbol{b}_2)$ are the vectorized forms of $\boldsymbol{K}_1 \in \mathbb{R}^{m_1 \times d}$, $\boldsymbol{K}_2 \in \mathbb{R}^{m_2 \times m_1}$, $\boldsymbol{b}_1 \in \mathbb{R}^{m_1}$, $\boldsymbol{b}_2 \in \mathbb{R}^{m_2}$.

Even more complicated NNs f can be created using additional layers and/or various orders of the building blocks presented in (2.2). In NN terminology, the number of layers defines the depth of the network, and the sizes of the hidden dimensions m_1, m_2 determine the width of the network. Naturally, when deciding a network's architecture, more layers tends to lead to more parameters and complexity. As a result, so-called deep NNs—networks with many layers—are hand-waved as incomprehensible magic blackboxes due to the incredible amount of complexity.

Ultimately, NNs provide a data-driven methodology that transform data to different dimensional spaces where the desired task may be easier to perform. This series of transformations can be comprehended as a series of projections onto manifolds [99] in different dimensional spaces.

2.1.2 Training

Given an NN f with specified architecture, one must train the network for it to be useful. Training an NN equates to tuning the values of $\boldsymbol{\theta}$ by solving the optimization problem

$$\min_{\boldsymbol{\theta}} \quad \mathcal{J}\left[\, f(\boldsymbol{x};\boldsymbol{\theta})\,\right], \tag{2.3}$$

for general NN f and scalar loss function $\mathcal{J}$—also called the objective function or cost function or error function. Solving (2.3) generally follows Algorithm 1, an iterative optimization process.

Backpropagation and preferred optimization schemes present the most interesting portions of NN training. Backpropagation [91], the fancy term for chain rule, involves the complicated process of computing the gradient of the loss function with respect to the parameters $\nabla_{\boldsymbol{\theta}}\mathcal{J}\left[\, f(\boldsymbol{x};\boldsymbol{\theta})\,\right]$. Since the forward propagation of the network passes through the network layers to compute $f(\boldsymbol{x})$, backpropagation passes through the layers in the opposite direction to compute $\nabla_{\boldsymbol{\theta}}\mathcal{J}\left[\, f(\boldsymbol{x};\boldsymbol{\theta})\,\right]$. Many machine learning packages include automatic differentiation (AD) [72], which computes the gradient by applying chain rule via a computational graph. The optimization schemes used to solve (2.3) and train the NN often require the gradient output from backpropagation. Variants of gradient descent that incorporate some understanding of batching have risen in popularity to become the preferred optimization schemes. For most of our experiments, we use ADAM [52], a subgradient method [12] with adaptive momentum, because it empirically performs well in nonconvex, noisy, high-dimensional search spaces.

2.1.3 Concepts

Taking a step back and using simpler linear least squares regression, we discuss some broader machine learning terms used in this dissertation.

Algorithm 1: Neural network training

```
   Data: inputs $\boldsymbol{x}$
   Result: $f(\boldsymbol{x}\,;\boldsymbol{\theta})$ where $f$ is a neural network with parameters $\boldsymbol{\theta}$
1  initialize parameters $\boldsymbol{\theta}$ ;
2  while iteration $i <$ max number of iterations do
3      compute $f(\boldsymbol{x}\,;\boldsymbol{\theta})$ ;
4      compute loss $\mathcal{J}[\,f(\boldsymbol{x}\,;\boldsymbol{\theta})\,]$ ;
5      compute gradient $\nabla_{\boldsymbol{\theta}}\mathcal{J}[\,f(\boldsymbol{x}\,;\boldsymbol{\theta})\,]$ via chain rule (called backpropagation) ;
6      update parameters $\boldsymbol{\theta}$ via gradient-based optimization ;
7      $i \leftarrow i+1$ ;
8  end
```

Linear Least Squares Regression

The simplest form of machine learning involves fitting a line $y = mx + b$ to noisy data $\boldsymbol{X} = \{(x_i, y_i)\}_{i=1}^{n}$. Specifically, we solve

$$\begin{aligned} \min_{\boldsymbol{\theta}} \|\boldsymbol{y} - \boldsymbol{A}\boldsymbol{\theta}\|^2 &= \min_{m,b} \left\| \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix} - \begin{bmatrix} x_1 & 1 \\ x_2 & 1 \\ \vdots & \vdots \\ x_n & 1 \end{bmatrix} \begin{bmatrix} m \\ b \end{bmatrix} \right\|^2 \\ &= \min_{m,b} \mathop{\mathbb{E}}_{(x_i,y_i)\in\boldsymbol{X}} \|y_i - (mx_i + b)\|^2, \end{aligned} \tag{2.4}$$

where $\mathbb{E}$ denotes the expectation value. This model has two parameters $m, b \in \mathbb{R}$, and we refer to each y_i as the *ground truth* for corresponding input x_i.

Training, Validation, and Testing

Machine learning approaches strive to provide tools that perform well when provided an input x_j that the model has not seen before. As such, we simulate running a model on unseen data by creating and running a model on a hold-out set $\boldsymbol{X}_{\text{val}}$, not used in training. Therefore, we really solve

$$\min_{m,b} \mathop{\mathbb{E}}_{(x_j,y_j)\in \boldsymbol{X}_{\text{val}}} \|y_j - (mx_j + b)\|^2, \tag{2.5}$$

but only by using the training set $\boldsymbol{X}$ to tune the parameters

$$\operatorname*{arg\,min}_{m,b} \mathop{\mathbb{E}}_{(x_i,y_i)\in \boldsymbol{X}} \left\{\|y_i - (mx_i + b)\|^2 + \mathcal{R}(x_i, m, b)\right\}, \tag{2.6}$$

for some optional regularization term $\mathcal{R}$. Regularization terms are often used to make numerical problems strongly convex or to reduce variance [4].

When using an iterative solver for training (Algorithm 1), we often check the performance of the model on the validation set throughout, storing or overwriting the parameters that yield minimum validation loss as defined in (2.5). The validation set ensures that the model does not *overfit* to the set $\boldsymbol{X}$ during the training process. Overfitting occurs when the model fits to the unimportant noise of the training data $\boldsymbol{X}$ while its performance on the validation set degrades [46]. A model that does not overfit is called *generalizable* because the solution generalizes from the training set $\boldsymbol{X}$ to the validation set $\boldsymbol{X}_{\text{val}}$.

After the entire training process, we select the parameters that produce the minimum validation loss (2.5) as the tuned model parameters. Since $\boldsymbol{X}_{\text{val}}$ is used multiple times during the training process, it no longer acts as an actual hold-out test. Thus, we use a *testing set*, a third set disjoint from the training and validation sets. The testing set is an actual hold-out set that is used only once. In practice, for given data, we randomly assign each input-output pair (x_i, y_i) to exactly one of training, validation, and testing sets.

While on the topic of noisy data, we should address the volume of data required to train machine learning models. A typical rule of thumb is that one needs more independent data points than ten times the number of model parameters. For the least squares problem, this equates to needing 20 data points to fit a line. This arises from the idea that fewer points will not present enough observations to overpower the influence of the noise; the few parameters will thus overfit to the noise because the underlying signal from the data is not pronounced enough. Thus, machine learning models with fewer parameters can be preferred. While aspirational, obtaining large amounts of data may not always be possible. Furthermore, the role of regularization and conditioning can assist highly parameterized models to fit limited data.

2.2 Neural ODEs

Historically, NNs were limited in depth because as NNs grew deeper and deeper, accuracy saturated. Motivated to fix this degradation issue, He et al. [45] incorporated skip connections (direct inputs into a layer come from the previous two layers instead of solely the previous layer) into their model. The resultant residual neural network (ResNet) outperformed the state-of-the-art on multiple tasks—most notably, in image classification.

An M-layer discrete ResNet for initial state $\boldsymbol{z}^{(0)} = \boldsymbol{x} \in \mathbb{R}^d$ can be written as

$$\boldsymbol{z}^{(j+1)} = \boldsymbol{z}^{(j)} + h\,\mathbf{v}\left(\boldsymbol{z}^{(j)}, t_j\,; \boldsymbol{\theta}^{(j)}\right), \quad \text{where} \quad j = 1, \dots, M, \tag{2.7}$$

for function $\mathbf{v}$ parameterized by $\boldsymbol{\theta}^{(j)} \in \mathbb{R}^p$ and step size h. In machine learning language, $\mathbf{v}$ is an NN layer where $\boldsymbol{z}^{(j)}$ are the features and $\boldsymbol{\theta}^{(j)}$ are the layer weights. Time $t \in [0, T]$ for $T < \infty$ is artificial in this context as most NNs do not depend on time, but merely an ordering of the layers.

The discrete ResNet (2.7) is exactly the forward Euler method applied to an ODE [25, 41], as we show here. Interpreting the weights $\boldsymbol{\theta}^{(j)}$ as evaluations of $\boldsymbol{\theta}\colon [0, T] \to \mathbb{R}^p$ at the time points $t_j = j \cdot h$ in time-horizon $[0, T]$ and taking the limit as $M \to \infty$ and $h \to 0$, we see that the M-layer ResNet converges to the ODE

$$\begin{aligned} \partial_t \boldsymbol{z}_{\boldsymbol{x}}(t) &= \mathbf{v}\big(\boldsymbol{z}_{\boldsymbol{x}}(t), t\,;\, \boldsymbol{\theta}(t)\big)\,, \quad \text{for} \quad 0 \le t \le T, \\ \boldsymbol{z}_{\boldsymbol{x}}(0) &= \boldsymbol{x}. \end{aligned} \tag{2.8}$$

The notation $\boldsymbol{z}_{\boldsymbol{x}}(t)$ denotes the *state* (or *features*) $\boldsymbol{z}$ at a time t associated with initial point $\boldsymbol{x}$; $\boldsymbol{z}$ is a function of $\boldsymbol{x}$ and t, hence the use of the partial derivative in (2.8). Viewing the ResNet from a continuous viewpoint allows for the analysis of the convergence properties and a more intuitive interpretation of the weights $\boldsymbol{\theta}$. Instead of solving (2.8) with forward Euler, using any blackbox solver can then define an NN. This approach gained popularization, and (2.8) is dubbed a *neural ODE* [19].

2.3 Optimal Control

Optimal control (OC) problems present a similar and more general form of the neural ODE defined in (2.8). We present OC theory relevant for the rest of the dissertation.

We are interested in deterministic finite time-horizon OC problems. Construct a family of these problems by varying the starting times and starting points [29]. Fix a time-horizon $[0, T]$ and consider system dynamics governed by

$$\partial_s \boldsymbol{z}_{\boldsymbol{x},t}(s) = F\big(\boldsymbol{z}_{\boldsymbol{x},t}(s), \boldsymbol{u}_{\boldsymbol{x},t}(s), s\big), \quad \boldsymbol{z}_{\boldsymbol{x},t}(t) = \boldsymbol{x}, \tag{2.9}$$

for $t \le s \le T$. Here, $\boldsymbol{x} \in \mathbb{R}^d$ denotes the initial state, and $t \in [0, T]$ is the initial time of the system. Next, $\boldsymbol{z}_{\boldsymbol{x},t}(s) \in \mathbb{R}^d$ represents the state of the system at time $s \in [t, T]$ with initial data $(\boldsymbol{x}, t)$, and $\boldsymbol{u}_{\boldsymbol{x},t}(s) \in U \subset \mathbb{R}^a$ gives the control applied at time s. Hence, $F\colon \mathbb{R}^d \times U \times [0, T] \to \mathbb{R}^d$ models the evolution of the state $\boldsymbol{z}_{\boldsymbol{x},t}\colon [t, T] \to \mathbb{R}^d$ in response to the control $\boldsymbol{u}_{\boldsymbol{x},t}\colon [t, T] \to U$.

Next, suppose that a control $\boldsymbol{u}_{\boldsymbol{x},t}\colon [t,T] \to U$ yields a cost $\boldsymbol{\mathcal{L}}$ such that

$$\boldsymbol{\mathcal{L}}_{\boldsymbol{x},t}[\boldsymbol{z}_{\boldsymbol{x},t}, \boldsymbol{u}_{\boldsymbol{x},t}] = \int_t^T L\big(\boldsymbol{z}_{\boldsymbol{x},t}(s), \boldsymbol{u}_{\boldsymbol{x},t}(s), s\big)\, \mathrm{d}s \,+\, G\big(\boldsymbol{z}_{\boldsymbol{x},t}(T)\big), \tag{2.10}$$

where $L\colon \mathbb{R}^d \times U \times [0,T] \to \mathbb{R}$ is the *running cost* or the *Lagrangian*, and $G\colon \mathbb{R}^d \to \mathbb{R}$ is the *terminal cost*. We assume that F, L, G, U are sufficiently regular [33]. The objective of the OC problem is to find the control that incurs the minimal cost, i.e.,

$$\Phi(\boldsymbol{x}, t) = \inf_{\boldsymbol{u}_{\boldsymbol{x},t}} \, \boldsymbol{\mathcal{L}}_{\boldsymbol{x},t}[\boldsymbol{z}_{\boldsymbol{x},t}, \boldsymbol{u}_{\boldsymbol{x},t}], \tag{2.11}$$

where Φ is called the *value function*. A solution $\boldsymbol{u}^*_{\boldsymbol{x},t}$ of (2.11) is called an *optimal control*. Accordingly, the $\boldsymbol{z}^*_{\boldsymbol{x},t}$ which corresponds to $\boldsymbol{u}^*_{\boldsymbol{x},t}$ is an *optimal trajectory*.

Two closely related approaches to solve (2.11) exist: applying the Pontryagin Maximum Principle [83] (Section 2.3.1) and solving the Hamilton-Jacobi-Bellman PDE [7] (Section 2.3.2). Both utilize the Hamiltonian H of the system which is defined as

$$\begin{aligned} H(\boldsymbol{x}, \boldsymbol{p}, t) &= \sup_{\boldsymbol{u} \in U} \, \{-\boldsymbol{p} \cdot F(\boldsymbol{x}, \boldsymbol{u}, t) - L(\boldsymbol{x}, \boldsymbol{u}, t)\} \\ &= \sup_{\boldsymbol{u} \in U} \, \mathcal{H}(\boldsymbol{x}, \boldsymbol{p}, \boldsymbol{u}, t), \end{aligned} \tag{2.12}$$

where variable $\boldsymbol{p}$ is called the *adjoint state*.

For a wide variety of OC problems, (2.12) admits a closed-form solution. Thus, we make the following assumption throughout this work.

Assumption 2.3.1. *Suppose that* (2.12) *admits a unique continuous closed-form solution* $\boldsymbol{u}^*(\boldsymbol{x}, \boldsymbol{p}, t)$ *to the problem* (2.11)*.*

This assumption provides several advantages. First, finding the analytic expression of H is convenient when working with various approaches. Second, once state $\boldsymbol{z}$ and adjoint $\boldsymbol{p}$ variables are available, no optimization is necessary to find the optimal control—a key advantage for real-time applications.

2.3.1 Pontryagin Maximum Principle

The Pontryagin Maximum Principle (PMP) is a set of necessary first-order optimality conditions for the optimal control and trajectory [29, 33, 83]. Ultimately, the PMP conditions reduce to a system of forward-backward ODEs

$$\begin{cases} \partial_s \boldsymbol{z}^*_{\boldsymbol{x},t}(s) = -\nabla_{\boldsymbol{p}} H\big(\boldsymbol{z}^*_{\boldsymbol{x},t}(s), \boldsymbol{p}_{\boldsymbol{x},t}(s), s\big), \\ \partial_s \boldsymbol{p}_{\boldsymbol{x},t}(s) = \nabla_{\boldsymbol{x}} H\big(\boldsymbol{z}^*_{\boldsymbol{x},t}(s), \boldsymbol{p}_{\boldsymbol{x},t}(s), s\big), \\ \boldsymbol{z}^*_{\boldsymbol{x},t}(t) = \boldsymbol{x}, \quad \boldsymbol{p}_{\boldsymbol{x},t}(T) = \nabla_{\boldsymbol{x}} G\big(\boldsymbol{z}^*_{\boldsymbol{x},t}(T)\big), \end{cases} \tag{2.13}$$

for $t \leq s \leq T$. Using Assumption 2.3.1, once the state and adjoint are found, the optimal control is recovered via $\boldsymbol{u}^*_{\boldsymbol{x},t}(s) = \boldsymbol{u}^*\left(\boldsymbol{z}^*_{\boldsymbol{x},t}(s), \boldsymbol{p}_{\boldsymbol{x},t}(s), s\right)$.

The PMP approach is a *local* solution method that can be challenging to solve. First, by local solution method, we mean that (2.13) are necessary conditions for *fixed* initial data $(\boldsymbol{x}, t)$. Hence, if the initial data are changed or external shocks push the system off an optimized trajectory, a new system must be solved with the new initial data. Second, the forward-backward structure of (2.13)—the combination of the initial value problem for $\boldsymbol{z}$ and the terminal value problem for $\boldsymbol{p}$—makes the system difficult to solve numerically and depends on the initialization [51]. For example, one possible approach is a shooting method with some initial guess for $\boldsymbol{p}_{\boldsymbol{x},t}(0)$. Additional considerations regarding the PMP exist in Fleming and Soner [33].

2.3.2 Hamilton-Jacobi-Bellman PDE

The value function Φ contains complete information about the optimal control; that is, $\boldsymbol{u}^*$ and $\boldsymbol{p}$ can be recovered from Φ. For an OC problem, a unique value function Φ exists [33]. For simplicity, we assume that Φ is differentiable. In general, value functions are not differentiable, but the HJB theory still holds in the framework of viscosity solutions [21, 33]. The next theorem connects Φ and $(\boldsymbol{u}^*, \boldsymbol{p})$.

Theorem 2.3.2 (Theorem I.6.2 [33])**.** *Assume that* $\boldsymbol{u}^*_{\boldsymbol{x},t}$ *is a right-continuous optimal control and* Φ *is differentiable at* $(\boldsymbol{z}^*_{\boldsymbol{x},t}(s), s)$ *for* $t \leq s < T$*. Then*

$$\boldsymbol{p}_{\boldsymbol{x},t}(s) = \nabla_{\boldsymbol{x}} \Phi\big(\boldsymbol{z}^*_{\boldsymbol{x},t}(s), s\big) \tag{2.14}$$

satisfies (2.13)*. Thus, under Assumption 2.3.1,*

$$\boldsymbol{u}^*_{\boldsymbol{x},t}(s) = \boldsymbol{u}^*\Big(\boldsymbol{z}^*_{\boldsymbol{x},t}(s)\,,\, \nabla_{\boldsymbol{x}} \Phi\big(\boldsymbol{z}^*_{\boldsymbol{x},t}(s), s\big)\,,\, s\Big) \tag{2.15}$$

for almost all $s \in [t, T]$*.*

Also, the value function Φ is the unique (viscosity) solution of the HJB equations (also known as the *dynamic programming* equations)

$$\begin{cases} -\partial_t \Phi(\boldsymbol{x}, t) = -H\big(\boldsymbol{x}, \nabla_{\boldsymbol{x}} \Phi(\boldsymbol{x}, t), t\big) \\ \quad \Phi(\boldsymbol{x}, T) = G(\boldsymbol{x}) \end{cases} \tag{2.16}$$

for $(\boldsymbol{x}, t) \in \mathbb{R}^d \times [0, T]$. Thus, solving (2.16) is equivalent to solving the OC problem (2.11) [33].

The HJB approach comes with its own benefits and difficulties. Since Φ has complete information about the optimal controls for all initial data (Theorem 2.3.2), any solution method that seeks Φ is a *global* solution method. Although superior to PMP in terms of the completeness of the information about optimal controls, the HJB is challenging to solve in high dimensions by traditional numerical methods, e.g., ENO/WENO [79], that employ grids and are prohibitively expensive in high dimensions due to the *curse of dimensionality* (CoD)—costs increase exponentially when adding dimensions [7].

Assumption 2.3.1 yields optimal controls in a *feedback form* (2.15), and no further optimization is necessary to find the optimal controls. Feedback form representations are valuable in real-world applications. If Φ can be efficiently approximated and $\nabla_{\boldsymbol{x}}\Phi$ quickly calculated, optimal controls are readily available at any point in space and time. As such, the feedback form avoids recomputation at new points in scenarios when sudden changes to the initial data or the system's state occur.

2.4 Learning and Optimal Control

For the remainder of this dissertation, we use a specific form of the OC problems presented in Section 2.3. We consider problems with initial value $\boldsymbol{x}$ at time 0, and use $\boldsymbol{z}_{\boldsymbol{x}}(t)$ to denote the state of the system at time $t \in [0,T]$.

Training the neural ODE (2.8) can be phrased as an infinite-dimensional OC problem

$$\begin{aligned} &\min_{\boldsymbol{\theta}} \left\{ \mathcal{J}[\boldsymbol{\theta}] := \mathop{\mathbb{E}}_{\boldsymbol{x}\in\boldsymbol{X}} \mathcal{L}[\boldsymbol{z}_{\boldsymbol{x}},\boldsymbol{\theta}] + \mathcal{R}[\boldsymbol{\theta}] \right\} \\ &\text{s.t. } \boldsymbol{z}_{\boldsymbol{x}}(t) \text{ solves (2.8)}, \end{aligned} \tag{2.17}$$

where the model parameters $\boldsymbol{\theta}\colon [0,T] \to \mathbb{R}^p$ are the controls, $\boldsymbol{z}_{\boldsymbol{x}}$ satisfies the neural ODE for initial values given by the training data $\boldsymbol{x} \in \boldsymbol{X}$, and the loss functional $\mathcal{L}$ and regularization functional $\mathcal{R}$ are chosen to model a given learning task. For $\mathcal{R}$, we consider Tikhonov regularization (often called weight decay [56])

$$\mathcal{R}[\boldsymbol{\theta}] = \frac{\alpha}{2}\int_0^T \|\boldsymbol{\theta}(t)\|^2 \,\mathrm{d}t, \tag{2.18}$$

where we assume that parameter $\alpha > 0$ is chosen and kept fixed.

We will consider several types of loss functionals. As an example, here, we present the regression loss used in the time-series problem (Chapter 3). When training a neural ODE to approximate a given function $\boldsymbol{y}\colon [0,T] \to \mathbb{R}^d$, we choose

$$\mathcal{L}[\boldsymbol{z_x}, \boldsymbol{\theta}] = \int_0^T L\big(\boldsymbol{z_x}(t), \boldsymbol{y}(t)\big)\, \mathrm{d}t, \quad L(\boldsymbol{z_x}, \boldsymbol{y}) = \frac{1}{2}\|\boldsymbol{z_x} - \boldsymbol{y}\|^2. \tag{2.19}$$

When the ground truth $\boldsymbol{y}$ is known only at some time points, a discretized version of this functional $\mathcal{L}$ is used (Chapter 3).

2.5 Neural ODEs as Reinforcement Learning

Machine learning methods generally fall into three categories: supervised, unsupervised, and reinforcement learning. Supervised learning tasks require an input with corresponding ground truth, e.g., the least squares problem (2.4). Unsupervised learning is "typically about finding structure" hidden in data with no ground truth, and often involves clustering-type algorithms [103]. Reinforcement learning focuses on minimizing some error function instead of trying to match some ground truth or discover hidden structure in data.

Reinforcement learning problems focus on an agent interacting with its dynamic environment to learn an optimal policy, often through trial and error. Such problems contain four main elements: a policy, reward signal, value function, and an environment model [103].

The *policy* dictates an agent's action at a given time depending on the current state of the environment. At each time step, the environment sends a *reward signal* to the agent, like points in a video game. The *value function* specifies the expected return/reward an agent can obtain in a given state by following a policy. This long-run view provides a farsighted judgment whereas the reward signal provides immediate incentive. Agents can either learn from a predictive *model* of the environment to decide what actions to take, or agents bypass this altogether in favor of learning the policy directly through trial and error. The former is commonly known as model-based reinforcement learning and the latter as model-free reinforcement learning.

Reinforcement learning problems specifically seek an agent's policy that optimizes the value function or the accumulated reward, and modern implementations leverage the advances in NNs [61]. OC and reinforcement learning share significant overlap in theory and design [10, 86].

In both reinforcement learning and OC, the agent or system begins with some initial state, which we have been calling $\boldsymbol{x}$. In OC, we seek a control that minimizes the value function Φ, whereas in reinforcement learning, the optimal policy maximizes the value function. The reward signal is applied at every time step in reinforcement learning, similar to the running cost L in OC. Lastly, the OC deterministic dynamics F equate to the model in reinforcement learning.

The main difference between the OC background (Section 2.3) and reinforcement learning centers around continuity versus discreteness. The reinforcement learning model often takes the form of a transition matrix with discrete probabilities. The reward signal provides values based on the discrete state of the agent, and thus the discrete value function appears like a table to be traversed. Undergraduate computer science students agonize over setting up and traversing this table in a process called *dynamic programming*, which ultimately is the approach to solving the discrete Bellman equation, a form of the HJB equations (2.16).

The time-series and image classification problems addressed in Chapters 3 and 4, respectively, are typical examples of supervised learning. Meanwhile, the OT-Flow formulation of the CNF (Section 5.4) and the path-finding problems (Chapter 6) are instances of model-based reinforcement learning.

Chapter 3

Time-Series Regression

We use a simple time-series regression problem to demonstrate important aspects of neural ODEs and compare the discretize-optimize and optimize-discretize approaches. This chapter heavily incorporates portions from Onken and Ruthotto [76].

3.1 Problem

In time-series regression, we aim to model an unknown ODE from sampled data. Given time-series data $\boldsymbol{y}^{(1)}{=}\boldsymbol{y}(t_1), \ldots, \boldsymbol{y}^{(n)}{=}\boldsymbol{y}(t_n)$ (e.g., the data in Figure 3.1) obtained from some unknown function $\boldsymbol{y}$, the goal is to tune the neural ODE weights $\boldsymbol{\theta}$ in (2.8) such that $\boldsymbol{z}(t_k) \approx \boldsymbol{y}^{(k)}$ for $k = 0, \ldots, n$. For this chapter, we use a single initial value $\boldsymbol{x}$ at initial time 0. As such, we avoid the subscripts on $\boldsymbol{z}$ and ground truth $\boldsymbol{y}$ and write $\boldsymbol{z}(0) = \boldsymbol{z}^{(0)} = \boldsymbol{y}^{(0)}$.

For this task, we use the ODE [15, 19, 84]

$$\begin{cases} \partial_t \boldsymbol{y} = \boldsymbol{B} \boldsymbol{y}^{\circ 3} \\ \boldsymbol{y}^{(0)} = \boldsymbol{x} \end{cases}, \quad \text{where } \boldsymbol{B} = \begin{bmatrix} -0.1 & 2 \\ -2 & -0.1 \end{bmatrix}, \ \boldsymbol{x} = \begin{bmatrix} 2 \\ 0 \end{bmatrix} \tag{3.1}$$

where $\boldsymbol{y}^{\circ 3}$ denotes the element-wise cubic and $t \in [0, 1.5]$.

To transform the variational problem (2.17) into a finite-dimensional problem, it is most common to discretize the control on a *control grid*, i.e., $\boldsymbol{\theta}^{(j)} = \boldsymbol{\theta}(t_j)$ for $t_j{=}h\cdot j$ with stepsize h, and numerically approximate the integrals in $\mathcal{J}$ from (2.18) and (2.19) using quadrature rules. Discrete NNs traditionally also discretize the states on the same time grid (the same t_j). For time-series regression, we consider the discrete objective function

$$J(\boldsymbol{\theta}) = h \sum_{i=1}^{M} L\big(\boldsymbol{z}^{(i)}, \boldsymbol{y}^{(i)}\big), \tag{3.2}$$

for L defined in (2.19). Here, the notation $\boldsymbol{z}^{(i)}$ denotes the features of the ith layer as in (2.7). The continuous neural ODE (2.8) allows for the control and state discretizations to differ, which we address in the next chapter. We use this discrete optimization strategy and in Section 3.2 focus on handling the state equation (2.8).

3.2 Discretize-Optimize vs. Optimize-Discretize

For solving PDE-constrained optimization problems, such as (2.17), two prominent approaches exist: discretize-optimize (DO) and optimize-discretize (OD). In DO, one first discretizes the constraining PDE at several time points, then optimizes on this discretization. In OD, one first optimizes the problem in the continuous space (using the adjoint equations from the Karush-Kuhn-Tucker optimality conditions), then discretizes the PDE to provide a result. Most neural ODE literature currently being published promotes the OD approach [19, 32, 39]. We compare the two approaches, address relevant related works, and demonstrate the computational and convergence benefits that make DO an often preferred choice [76].

The DO approach to performing time integration discretizes the ODE (2.8) in time and then optimizes $\mathcal{J}$ using that discretization. This approach is common in NNs and easy to implement, especially when AD can be used for the backpropagation. For example, a discrete ResNet with input features (or initial conditions) $\boldsymbol{x}$ follows forward propagation (2.7) as the explicit Euler method on a uniform time discretization with step size h. The choice of fixed step size h trades off the accuracy of the solution with the amount of computation and overall training time.

In applications that require the neural ODE for prediction or inference, such as time-series regression, the step size h must be chosen judiciously to ensure the trained discrete model captures the properties of the continuous model. Choosing a too large value of h, may, for example, yield a discrete flow model with an inaccurate or suboptimal inverse. Choosing a too small value of h leads to greater computational costs. These tradeoffs influence the tuning of the step size hyperparameter. In contrast to selection of other hyperparameters, the step size choice provides the advantage of monitoring the accuracy of the ODE solver. While we use a fixed step size h for simplicity, we can obtain even more efficient approaches by combining adaptive discretization schemes with the backpropagation used in DO.

Following similar steps used in [36], we expose a crucial difference in the gradient computation between the DO and OD approaches. Recall from Section 2.1 that accurate gradients are critical in ensuring the efficiency of gradient-based optimization algorithms, including Stochastic Gradient Descent (SGD) [89] and ADAM [52]. The updates for $\boldsymbol{\theta}$ in such methods depend on the gradient—really, the subgradient [12]—of the objective function in (2.17). For ease of presentation, we use and assume that the forward propagation is the forward Euler scheme (2.7) and that the objective function consists of the time-series regression loss defined by (2.19) with no regularization.

In DO, the backpropagation of the discrete ResNet (2.7) computes the gradients. In practice, AD traverses the computational graph[1] backward in time to perform this computation. The discretization of the forward propagation completely determines

[1]The computational graph is constructed during the forward propagation and is a directed acyclical graph that tracks which inputs go through which functions in the order that the composition of functions is applied.

this process. For discrete objective function (3.2) and using auxiliary variable $\boldsymbol{a}$, the backpropagation through the forward Euler discretization in (2.7) is

$$\begin{aligned} \nabla_{\boldsymbol{\theta}^{(j)}} J(\boldsymbol{\theta}) &= h\, \nabla_{\boldsymbol{\theta}}\, \mathbf{v}\big(\boldsymbol{z}^{(j)}, t_j\,;\, \boldsymbol{\theta}^{(j)}\big)\; \boldsymbol{a}_j, \quad \text{where} \\ \boldsymbol{a}_M &= h\, \nabla_{\boldsymbol{z}} L\left(\boldsymbol{z}^{(M)}, \boldsymbol{y}^{(M)}\right) \quad \text{and} \\ \boldsymbol{a}_j &= \boldsymbol{a}_{j+1} + h \left(\nabla_{\boldsymbol{z}} \mathbf{v}\big(\boldsymbol{z}^{(j)}, t_j\,;\, \boldsymbol{\theta}^{(j)}\big)\, \boldsymbol{a}_{j+1} + \nabla_{\boldsymbol{z}} L\big(\boldsymbol{z}^{(j)}, \boldsymbol{y}^{(j)}\big) \right), \end{aligned} \tag{3.3}$$

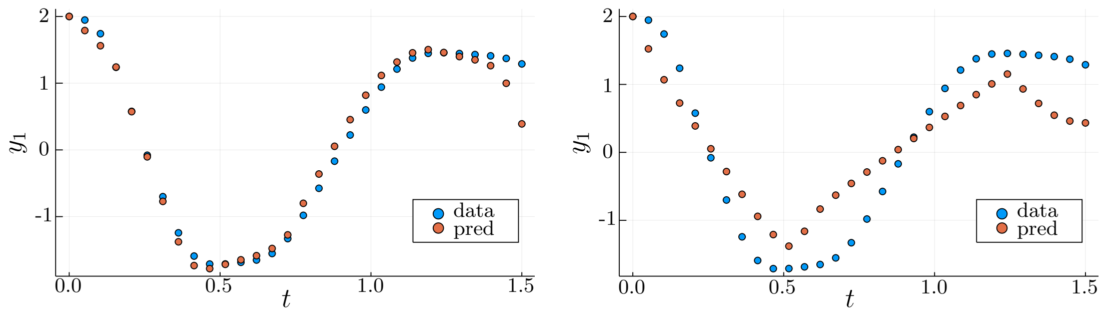
and the computations are backward through the layers, i.e., $j = M{-}1, M{-}2, \ldots, 1$.

In OD the gradients are computed by numerically solving the adjoint equation

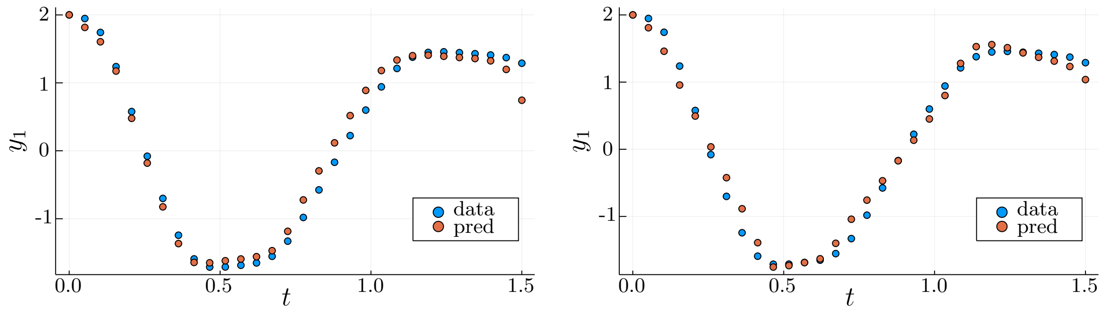
$$\begin{aligned} \nabla_{\boldsymbol{\theta}(t)} \mathcal{J}[\boldsymbol{\theta}] &= \boldsymbol{a}(t) \nabla_{\boldsymbol{\theta}} \mathbf{v}\big(\boldsymbol{z}(t), t\,;\, \boldsymbol{\theta}(t)\big), \quad \text{where} \\ \boldsymbol{a}(T) &= \nabla_{\boldsymbol{z}} L\big(\boldsymbol{z}(T), \boldsymbol{y}(T)\big) \quad \text{and} \\ -\partial_t \boldsymbol{a} &= \nabla_{\boldsymbol{z}} \mathbf{v}\big(\boldsymbol{z}(t), t\,;\, \boldsymbol{\theta}(t)\big)\, \boldsymbol{a}(t) + \nabla_{\boldsymbol{z}} L\big(\boldsymbol{z}(t), \boldsymbol{y}(t)\big). \end{aligned} \tag{3.4}$$

As indicated by the notation $-\partial_t$, this final value problem is solved backward in time. This equation can be derived from the Karush-Kuhn-Tucker (KKT) optimality conditions of the continuous learning problem (Appendix A).

Comparison of the backpropagation (3.3) and the adjoint computation (3.4) shows that both depend on the intermediate states, which need to be stored or recomputed. However, the two differ because flexibility exists for choosing the numerical scheme to discretize the adjoint computation in (3.4), whereas the computation in (3.3) is determined by the discrete forward propagation. In fact, the backpropagation shown in (3.3) can be seen as a discretization of the adjoint equation; however, the standard



(a) DO iter 100, time: 1.22 ms, loss: 1.165 (b) OD iter 100, time: 88.6 ms, loss: 13.484



(c) DO iter 300, time: 1.68 ms, loss: 0.334 (d) OD iter 300, time: 83.5 ms, loss: 1.405

Figure 3.1: Time-series regression training iterations 100 and 300 comparing the DO and OD approaches with the ground truth (3.1). Discrepancies between convergence behavior of the approaches vary with initial parameterization (Section 3.3.1).

backward Euler scheme reads

$$\begin{aligned}\nabla_{\boldsymbol{\theta}^{(j)}} J(\boldsymbol{\theta}) &= h \nabla_{\boldsymbol{\theta}} \mathbf{v}\big(\boldsymbol{z}(t_j), t_j\,;\,\boldsymbol{\theta}(t_j)\big)\,\boldsymbol{a}_j, \quad \text{where}\\ \boldsymbol{a}_M &= h \nabla_{\boldsymbol{z}} L\big(\boldsymbol{z}(t_M), \boldsymbol{y}(t_M)\big) \quad \text{and}\\ \boldsymbol{a}_j &= \boldsymbol{a}_{j+1} + h\big[\nabla_{\boldsymbol{z}} \mathbf{v}\big(\boldsymbol{z}(t_{j+1}), t_{j+1}\,;\,\boldsymbol{\theta}(t_{j+1})\big)\boldsymbol{a}_{j+1} + \nabla_{\boldsymbol{z}} L\big(\boldsymbol{z}(t_{j+1}), \boldsymbol{y}(t_{j+1})\big)\big].\end{aligned} \tag{3.5}$$

The only difference between (3.3) and (3.5) is a shift of the indices of the intermediate weights and features. Thus, the gradients obtained using both methods differ unless both equations are solved accurately, such as when h converges to zero. Hence, the OD approach in Chen et al. [19], which uses adaptive time integrators for the forward and adjoint equations, may provide inaccurate gradients when the time steps differ between both solvers and the tolerance of those solvers is not sufficiently small [36, 59].

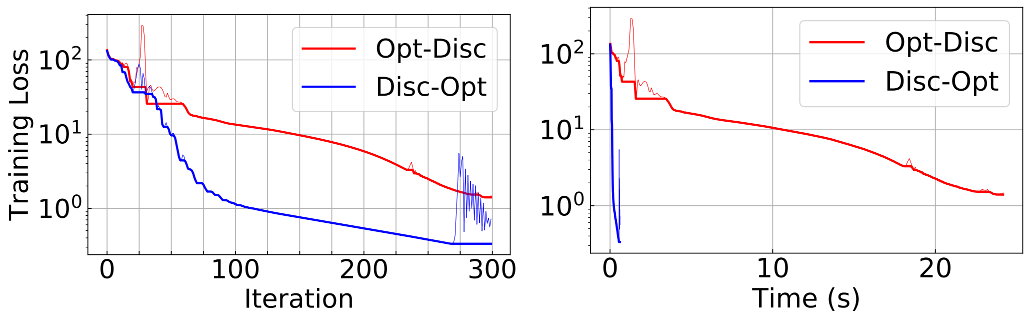


Figure 3.2: For time-series regression, the DO method converges in fewer iterations, and each of its iterations requires less time. The mean iteration clocktimes are 2.0 ms for DO and 80.5 ms for OD. Since ADAM is not a descent method, the iteration loss (thin line) can be higher than the best loss thus far (thick line).

This comparison is fairly standard and is also performed in Gholaminejad et al. [36].

The differences of the gradients computed using DO and OD affect the convergence of neural ODEs for image classification [36]. We compare the training convergence of neural ODEs in time-series regression, where the discretization trained in DO must capture the relevant properties of the continuous model.

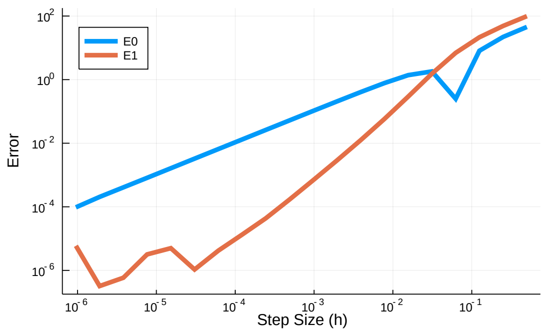
3.3 Numerical Experiments

Given 30 time-series data points $\boldsymbol{x}, \boldsymbol{y}^{(1)}, \dots, \boldsymbol{y}^{(29)}$ $(n = 29)$, we train a neural ODE (2.8) so that $\boldsymbol{z}(t_k) \approx \boldsymbol{y}^{(k)}$ for $k = 0, \dots, 29$.

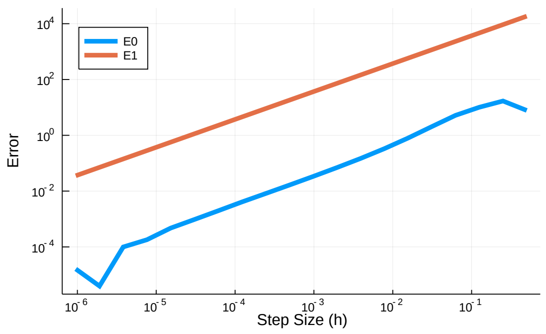
We copy the neural ODE in Rackauckas et al. [84] where the NN layer $\mathbf{v}$ in (2.8) resembles a double dense layer given by

$$\begin{aligned} \mathbf{v}(\boldsymbol{z}, t\,; \boldsymbol{\theta}) &:= \boldsymbol{D}_2 \circ \boldsymbol{D}_1(\boldsymbol{z}^{\circ 3}) \\ &= \boldsymbol{K}_2 \tanh(\boldsymbol{K}_1 \boldsymbol{z}^{\circ 3} + \boldsymbol{b}_1) + \boldsymbol{b}_2 \end{aligned} \tag{3.6}$$

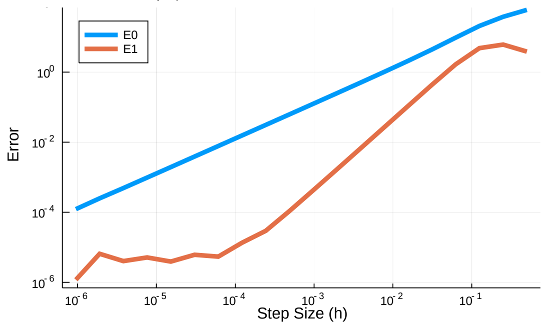
where $\boldsymbol{D}_1, \boldsymbol{D}_2$ represent dense layers as defined in (2.1). We explicitly write out the layers and remark that $\boldsymbol{\theta} = (\boldsymbol{K}_1, \boldsymbol{K}_2, \boldsymbol{b}_1, \boldsymbol{b}_2)$. The neural ODE is trained using 300 steps of the ADAM [52] optimizer with a learning rate (the size of the update step)



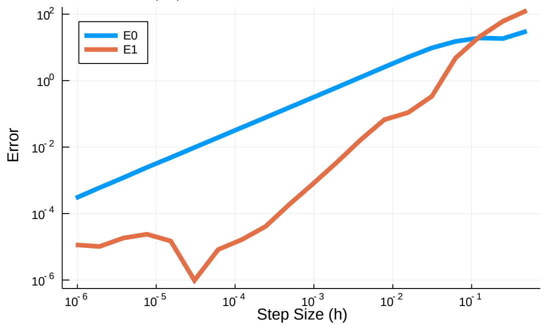
(a) OD Iteration 14



(b) OD Iteration 24



(c) DO Iteration 14



(d) DO Iteration 24

Figure 3.3: The derivative check (3.7) for iterations 14 and 24 shown on log-log plot. As expected, the gradients of the DO approach (bottom row) are correct; as h decays, $E_1(h)$ (red line) decays faster than $E_0(h)$ (blue line) for definitions in (3.7). The gradient in the OD approach (a,b) is correct for iteration 14, but not for iteration 24. In this case (b), the function E_1 (red line) is greater than E_0 (blue line).

of 0.1, starting with a Glorot initialization [38].[2] The OD and DO approaches use the same initializations (i.e., same random seed).

The OD approach employs an adaptive explicit Runge-Kutta 4(5) [84]. We compare this with a DO approach that uses Runge-Kutta 4 (RK4) with a fixed step size h for training the neural ODE. For the step size h, we select the spacing of the data points, i.e., $h = T/29 = 1.5/29$. With this choice, the evaluation of the loss function does not require any interpolation.

The DO approach substantially reduces the cost of training. We attribute these savings to the following two reasons. First, the runtime per iteration is 97% lower than the OD approach's(Figure 3.2). This speedup mainly comes from the fewer func-

[2]Code is available at `https://github.com/EmoryMLIP/DOvsOD_NeuralODEs`.

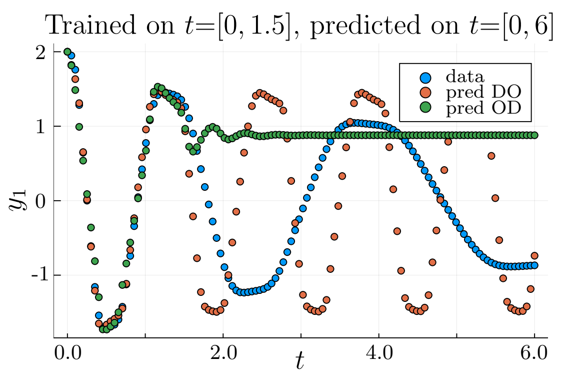
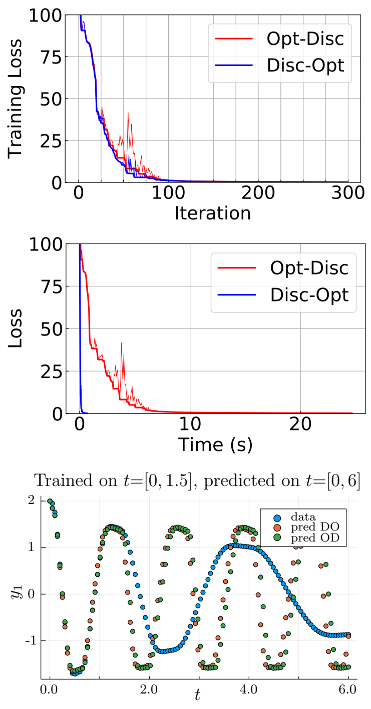


Figure 3.4: Extrapolation of time-series regression models. After training each model on the time interval $t \in [0, 1.5]$, we visualize how the trained NNs extrapolate up to $t = 6$. Each neural ODE models behaves as a smooth ODE, by construction, though the learned models oscillate much more quickly than the ground truth ODE.

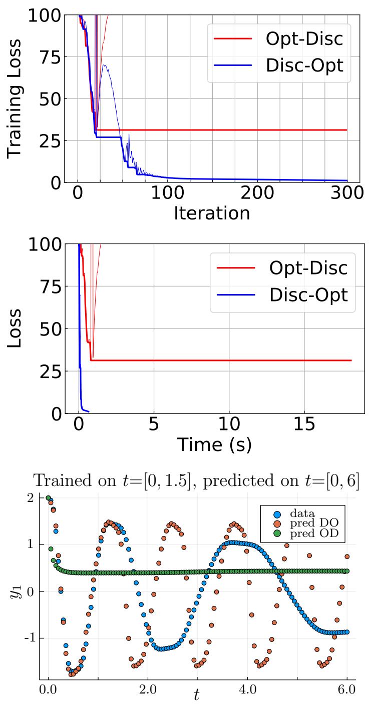
tion evaluations, pre-determined by the step size choice, used in the DO approach. Since the OD method uses adaptive time-stepping, the number of function evaluations (NFE), and thus, cost per iteration varies during the optimization. The mean iteration clocktimes are 2.0 ms for DO and 80.5 ms for OD. Second, the DO approach also converges in roughly one third of the total iterations (Figure 3.2) with predictions at iteration 100 drastically closer to the ground truth than the OD approach (Figure 3.1a,3.1b). Discrepancies between convergence behavior of the approaches vary with initial parameterization (Section 3.3.1). Around iteration 24, the ODE solvers struggle to improve the training (Figure 3.3), but the OD approach appears to suffer more.

To explain the fewer iterations needed by DO, we numerically test the quality of the gradients provided by both approaches through use of Taylor's theorem. Let $\boldsymbol{g} \in \mathbb{R}^n$ denote the gradient of the objective function $\boldsymbol{F} \colon \mathbb{R}^n \to \mathbb{R}$ at a point $\boldsymbol{\theta}$ and let $\boldsymbol{\omega} \in \mathbb{R}^n$ be a randomly chosen direction. Then, by Taylor's theorem, we have that

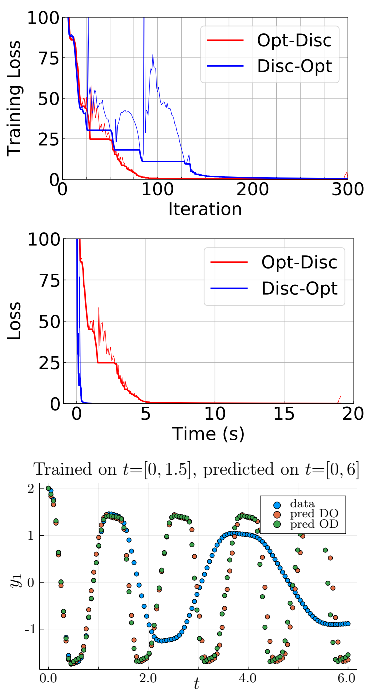
$$\begin{aligned} E_0(h) &:= \|\boldsymbol{F}(\boldsymbol{\theta} + h\boldsymbol{\omega}) - \boldsymbol{F}(\boldsymbol{\theta})\| = \mathcal{O}(h\|\boldsymbol{\omega}\|) \quad \text{and} \\ E_1(h) &:= \|\boldsymbol{F}(\boldsymbol{\theta} + h\boldsymbol{\omega}) - \boldsymbol{F}(\boldsymbol{\theta}) - h\boldsymbol{g}^\top\boldsymbol{\omega}\| = \mathcal{O}(h^2\|\boldsymbol{\omega}\|). \end{aligned} \tag{3.7}$$



(a) Different seed 1



(b) Different seed 2



(c) Different seed 3

Figure 3.5: Time-series regression convergence for different initial parameterizations (cf. Figure 3.2, 3.4).

We observe the decay of $E_0(h)$ and $E_1(h)$ as $h \to 0$ for a fixed (randomly chosen) direction $\boldsymbol{p}$ around the current network weights for two different iterations of the training (Figure 3.3). We scale both axes logarithmically. As expected, E_0 decays almost perfectly linearly, and the decay of E_1 using the DO method is approximately twice as steep for large h, ultimately leveling off due to rounding errors and conditioning (Figure 3.3c, 3.3d). While the derivative obtained in the OD approach is correct at iteration 14 (Figure 3.3a), it fails to provide an accurate gradient at iteration 24 (Figure 3.3b), where the error E_1 is greater than E_0 for all h that we tested.

3.3.1 Extrapolation and Different Initial Conditions

For time-series regression on $t \in [0, 1.5]$, the NNs appear to have modeled the ground truth ODE quite well (Figure 3.1). We ask if the models extrapolate correctly to time t outside the training period $[0, 1.5]$. We extend the prediction of the trained models to the time period $t \in [0, 6]$ (Figure 3.4), where we see that both the DO and OD approaches learn behaviors that represent smooth ODEs that are different from the ground truth used to create the data. This presents an example where neural ODEs extrapolate poorly.

We repeat the time-series regression for several initial conditions to demonstrate that the results span a broad set of convergent behaviors. We plot three other random seeds (Figure 3.5). Sometimes the DO and OD approaches converge at a similar rate per iteration and can have similar extrapolations (Figure 3.5a). Some cases exist where the loss skyrockets, and the models fail to fully converge in the 300 iterations (Figure 3.5b). Still other cases appear where OD converges to a lower loss than DO after 300 iterations, but extrapolation shows that the two models learn similar ODE dynamics (Figure 3.5c). In such cases, we observe that the DO model still trains more efficiently in terms of time. In fact, out of ten random initial conditions selected, the DO approach used fewer iterations to converge than the OD approach in nine of the comparisons. In the one model where the OD converges in fewer iterations than DO (Figure 3.5c), the DO model converges to the same loss in 466 iterations and still reduced training time by 94%. Training cost reductions due to the DO model across these ten initial conditions ranged from 94% to 99% with an average reduction of 97% (a 20x speedup).

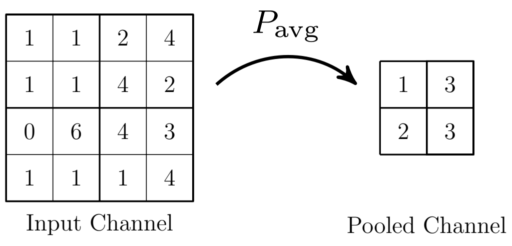
Chapter 4

Image Classification

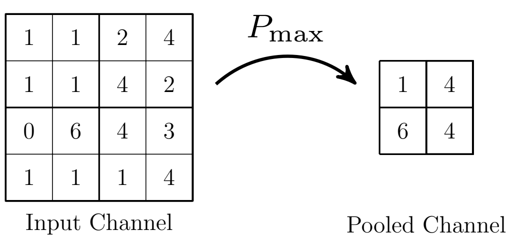
We use neural ODEs for image classification. The construction of the method included contributions by Simion Novikov, Eran Treister, and Lars Ruthotto. The medical application of this method took place at UnitedHealth Group Research & Development with contributions from Renn Caday, Wesley Carter, Seth Colbert-Pollack, Stephen Garth, Jessica Gronski, Rachel Jennings, Hunter McCawley, Prajakta Patil, and Jonathan Rolfs. We thank the National Cancer Institute for access to NCI's data collected by the National Lung Screening Trial (NLST). The statements contained herein are solely those of the authors and do not represent or imply concurrence or endorsement by NCI.

4.1 Problem

For image classification tasks, we expect a model $f\colon \mathbb{R}^d \to \mathbb{R}^c$ to accept a vectorized image $\boldsymbol{x}$ with d features and return a one-hot encoded vector $\boldsymbol{y} \in \mathbb{R}^c$, a vector of magnitude 1 consisting of $c-1$ zeros. Each of the c slots in $\boldsymbol{y}$ correspond to one of the c possible classes. The d features are often the product of the number of pixels and the number of channels in the image.



(a) Average Pooling



(b) Max Pooling

Figure 4.1: Pooling examples

Often, image classification models will take on the form

$$f(\boldsymbol{x};\boldsymbol{\theta}) = S \circ \boldsymbol{D}_2 \circ f_M \circ P_{M-1} \circ f_{M-1} \circ \cdots \circ P_2 \circ f_2 \circ P_1 \circ f_1 \circ \boldsymbol{D}_1(\boldsymbol{x}) \tag{4.1}$$

where S is the softmax operator, each f_i component is a neural ODE, each P_i is a pooling operator, each $\boldsymbol{D}_i$ is a dense layer (2.1), and $\boldsymbol{\theta}$ denotes all the parameters for the model f.

Pooling operators edit the input image by combining pixels. For instance, consider $\boldsymbol{z} \in \mathbb{R}^{48}$, a two-dimensional image with 4×4 pixels and 3 channels. If P_1 is a max pooling operator with a 2×2 kernel, then each of the four non-overlapping sets of 2×2 pixel squares in the image are "pooled" into one pixel. The value assigned to the pooled pixel is the maximum of the four pixels used to make it (Figure 4.1). After pooling, $P_1(\boldsymbol{z}) \in \mathbb{R}^{12}$ is a two-dimensional image with 2×2 pixels and 3 channels. Average pooling works similarly, except the pooled pixel is assigned the average value of the pixels pooled together to make it. Pooling operators apply separately to each image channel; Figure 4.1 contains simple examples when pooling is applied to one channel of $\boldsymbol{z}$.

The softmax operator S converts input features into percentages for each class. The design of (4.1) uses neural ODEs f_i that do not change dimensionality, while the dense layers and pooling operators do. Specifically, $\boldsymbol{D}_2$ is structured to return a hidden state $\boldsymbol{z} \in \mathbb{R}^c$. The softmax is an element-wise activation function that converts each element z_i in $\boldsymbol{z}$ to a percentage likelihood for the respective class via

$$S(z_i) = \frac{\exp(z_i)}{\sum_{j=1}^{c} \exp(z_j)}, \quad \text{for} \quad i = 1, \ldots, c. \tag{4.2}$$

4.1.1 Focal Loss

For a loss function, we use an α-weighted focal loss [65]

$$G\big(f(\boldsymbol{x}), \boldsymbol{y}\big) = -(\boldsymbol{\alpha} \odot \boldsymbol{y}) \Big((\mathbf{1}^\top - f(\boldsymbol{x})^\top)^{\circ\gamma} \odot \log\big(f(\boldsymbol{x})^\top\big) \Big), \tag{4.3}$$

where $\mathbf{1} \in \mathbb{R}^c$ is a vector of all ones. Here, the logarithm is applied element-wise, and $\circ\gamma$ denotes the element-wise power for a constant hyperparameter γ. The symbol $\odot$ represents the element-wise product of equally sized vectors or matrices (the Hadamard Product). The vector $\boldsymbol{\alpha} \in \mathbb{R}^c$ provides separate weighting for each class.

The α-weighted focal loss (4.3) is based on the cross entropy loss: $-\boldsymbol{y} \log\big(f(\boldsymbol{x})^\top\big)$, and in fact, the two are the same when $\boldsymbol{\alpha} = \mathbf{1}$ and $\gamma = 0$. Loss (4.3) adds two components. First, the focal loss portion $(\mathbf{1} - f(\boldsymbol{x})^\top)^{\circ\gamma}$ incentivizes the model to "focus" on samples that the model gets incorrect. Meanwhile, the samples that the model easily classifies receive less importance. Second, we use the $\boldsymbol{\alpha}$ to counteract imbalance arising from the classes. The training data $\boldsymbol{X}$ contains samples from c classes, but the number of samples per class need not be the same for all classes. To counteract this imbalance, we give more weight to the classes with fewer samples. Consider the vector $\boldsymbol{\eta} \in \mathbb{R}^c$ where element η_i is the number of samples in $\boldsymbol{X}$ for class i. For determining the $\boldsymbol{\alpha}$, we set

$$\boldsymbol{\alpha} = \frac{\sum_{j=1}^{c} \eta_j}{\boldsymbol{\eta}}. \tag{4.4}$$

The optimization problem (2.17) for image classification then is

$$\min_{\boldsymbol{\theta}} \underset{(\boldsymbol{x},\boldsymbol{y})\in\boldsymbol{X}}{\mathbb{E}} \left\{ G(f(\boldsymbol{x};\boldsymbol{\theta}),\boldsymbol{y}) \right\} + \mathcal{R}[\boldsymbol{\theta}] \tag{4.5}$$

where f is parameterized by $\boldsymbol{\theta}$ and the set of training data $\boldsymbol{X}$ holds image-label pairs $(\boldsymbol{x},\boldsymbol{y})$. We use weight decay (2.18) for regularizer $\mathcal{R}$.

4.2 Convolutional Neural Networks

Images present a structured form of input data since the pixels form orderly grid where the spatial positioning between two pixels is important. To exploit the spatial relationship between data features, convolution is a preferred approach for solving image classification tasks.

4.2.1 Convolutional Layer

Convolutional neural networks (CNNs) follow the same formulation as dense NNs. For example, a single convolutional layer is formulated the same as a dense layer (2.1), except that the linear operator $\boldsymbol{K}$ represents a convolutional operator instead of a dense matrix. We vectorize the input image $\boldsymbol{x}$ by vectorizing and stacking each of the image channels. For instance, a standard color image is comprised of the three channels for red, green, and blue

$$\boldsymbol{x} = \left[\begin{array}{c} \boldsymbol{x}_R \\ \hline \boldsymbol{x}_G \\ \hline \boldsymbol{x}_B \end{array} \right]. \tag{4.6}$$

The dimension d of this RGB image $\boldsymbol{x}$ is three times the number of pixels of the image.

Next, we formalize the convolutional operator as matrix. Convolutional operator $\boldsymbol{K}$ is parameterized by so-called kernels, small matrices convolved with the entire image channel. Convolutional operator $\boldsymbol{K}$ then can be written as an i-by-j block matrix for transforming an input image with j channels to i channels. Each of the $i \cdot j$ blocks in $\boldsymbol{K}$ is parameterized by a single kernel. For convolving our example image (4.6) from three to two channels, we can write $\boldsymbol{K}\boldsymbol{x}$ as

$$\boldsymbol{K}\boldsymbol{x} = \left[\begin{array}{c|c|c} K_{R2} & K_{G2} & K_{B2} \\ \hline K_{R1} & K_{G1} & K_{B1} \end{array}\right] \left[\begin{array}{c} \boldsymbol{x}_R \\ \hline \boldsymbol{x}_G \\ \hline \boldsymbol{x}_B \end{array}\right] = \left[\begin{array}{c|c} K_{R1}\boldsymbol{x}_R & K_{R2}\boldsymbol{x}_R \\ \hline K_{G1}\boldsymbol{x}_G & K_{G2}\boldsymbol{x}_G \\ \hline K_{B1}\boldsymbol{x}_B & K_{B2}\boldsymbol{x}_B \end{array}\right]. \tag{4.7}$$

Each block in $\boldsymbol{K}$ is a block-circulant matrix formed by the parameterizing kernel. For example, consider a 4×4 pixel image $\boldsymbol{x}$ and a standard sharpening kernel

$$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}. \tag{4.8}$$

When this kernel parameterizes K_{R1}, then $K_{R1}\boldsymbol{x}_R$ is

$$\begin{bmatrix} 0 & -1 & 0 & & -1 & 5 & -1 & & 0 & -1 & 0 & & & & & \\ & 0 & -1 & 0 & & -1 & 5 & -1 & & 0 & -1 & 0 & & & & \\ & & & & 0 & -1 & 0 & & -1 & 5 & -1 & & 0 & -1 & 0 & \\ & & & & & 0 & -1 & 0 & & -1 & 5 & -1 & & 0 & -1 & 0 \end{bmatrix} \begin{bmatrix} x_{R1} \\ \vdots \\ x_{R16} \end{bmatrix} \tag{4.9}$$

which is convolving the sharpening kernel with the red channel of the 4×4 image $\boldsymbol{x}$. In this example, $K_{R1}\boldsymbol{x}_R \in \mathbb{R}^4$ which equates to a channel of a 2×2 image. To stop convolutional operators from shrinking image dimensions, we can pad the input image with zeros on all sides prior to vectorization. Although the K_{R1} in (4.9) appears large, it depends entirely on a kernel with nine parameters. We present the convolution as matrix-vector multiplication; however, in practice, the convolutional operators are stored and implemented as tensors of the underlying kernels.

4.2.2 Normalization Layer

Training NNs (Algorithm 1) can often be slow and unstable. To assist in training speed and stability, normalization layers $\mathcal{N}$ are incorporated in the architecture. A normalization layer adds stability to the network because it re-normalizes hidden features by subtracting the mean of the features, then dividing by the standard deviation, then scaling the features.

We apply the normalization layer to input features via

$$\mathcal{N}(\boldsymbol{x};\boldsymbol{\theta}) = \zeta \frac{\boldsymbol{x} - \mu_{\mathcal{N}}}{\sqrt{\sigma_{\mathcal{N}}^2 + \varepsilon}} + \beta \tag{4.10}$$

where the parameters $\boldsymbol{\theta} = (\zeta, \beta)$ are for each channel and constant $\varepsilon > 0$ is a small floating point value that prevents division by zero. Here, $\mu_{\mathcal{N}}$ and $\sigma_{\mathcal{N}}$ denote the mean and standard deviation of the features, respectively. Many choices exist for what data should be used to compute $\mu_{\mathcal{N}}, \sigma_{\mathcal{N}}$ [117]. Instance normalization computes across all the features (pixels) in a single channel of single input object. Layer normalization uses all features of all channels in a single input object. Batch normalization computes across all features of a single channel across all input objects in a batch. Due to the large variance that can occur across different batches, batch normalization involves further parameterization terms—running mean and running variance—so that the model can be run on a single example at inference, where batching is not present.

4.2.3 Double Symmetric Layer

Recall that each f_i in (4.1) uses the neural ODE (2.8). For the neural layer $\mathbf{v}$ in the neural ODE, we choose a *double symmetric layer* [92]

$$\mathbf{v}(\boldsymbol{z}, t\,; \boldsymbol{\theta}) := -\boldsymbol{K}^\top \circ \sigma \circ \mathcal{N} \circ \boldsymbol{K}\boldsymbol{z} \tag{4.11}$$

for instance normalization layer $\mathcal{N}$ (4.10), parameters $\boldsymbol{\theta} = (\boldsymbol{K}, \zeta, \beta)$, and ReLU activation function—i.e., $\sigma(z) = \max(0, z)$. Most notably, the double symmetric layer uses the same parameters (kernels) in $\boldsymbol{K}$ and $\boldsymbol{K}^\top$, differing it from the double layer (2.2) which uses two convolutional operators with different parameters. This double symmetric layer leads to a negative semi-definite Jacobian and promotes NN stability [92].

4.3 Decoupling the Weights and Layers

Inspired by and comparing against the existing ResNet (2.7), the neural ODEs f_i in (4.1) result in reduced parameterization from decoupling the weights and layers.

In discrete networks, e.g., ResNet, each layer $\mathbf{v}^{(j)}$ has an associated set of weights $\boldsymbol{\theta}^{(j)}$ that belong solely to that layer. Thus, increasing the number of layers in a model similarly increases its number of weights. Deep networks often yield excellent performance as more parameters and layers gives the model greater expressibility. Simultaneously, highly parameterized models tend to require more data to avoid overfitting to the training data (Section 2.1.3). Thus, we seek models with comparable performance but fewer parameters than a many-layered discrete network. Whereas fewer parameters tends to result in a simpler model and needing less training data, networks with few layers tends to lead to poorer results.

We use the ResNet to present the associations of number of layers and model

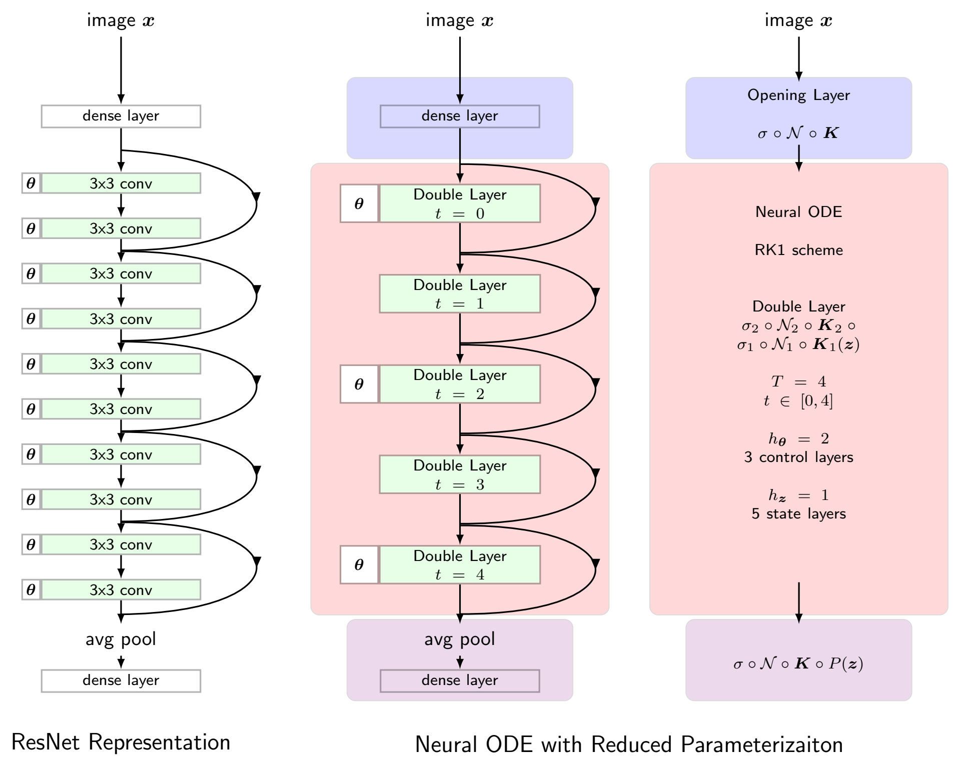


Figure 4.2: Image classification model example

performance. Recall that the ResNet is the forward Euler discretization of the neural ODE (2.8) [25, 41]. For example, imagine solving the neural ODE over time horizon $t \in [0, 1]$ with a step size h=0.5. This results in a two-layer network, i.e., two evaluations of $\mathbf{v}$ at $t = 0, 0.5$. With forward Euler, a smaller step size h results in more functional evaluations (more layers) and a more accurate approximation. Using this insight, we motivate how neural ODEs decouple the weights and layers so the model can have many layers but few parameters. We refer to the the discretization of the function evaluations as the *state layers* because they correspond to discretizing the state $\boldsymbol{z}$. We can discretize the controls (parameters) $\boldsymbol{\theta}$ with a different fineness generating what we call the *control layers*. When evaluating $\boldsymbol{\theta}$ between control layers, we linearly interpolate. In Figure 4.2, we demonstrate how an example neural ODE approach, $f(\boldsymbol{x}; \boldsymbol{\theta}) = S \circ \boldsymbol{D}_2 \circ P_1 \circ f_1 \circ \boldsymbol{D}_1(\boldsymbol{x})$, can reduce the parameterization of a

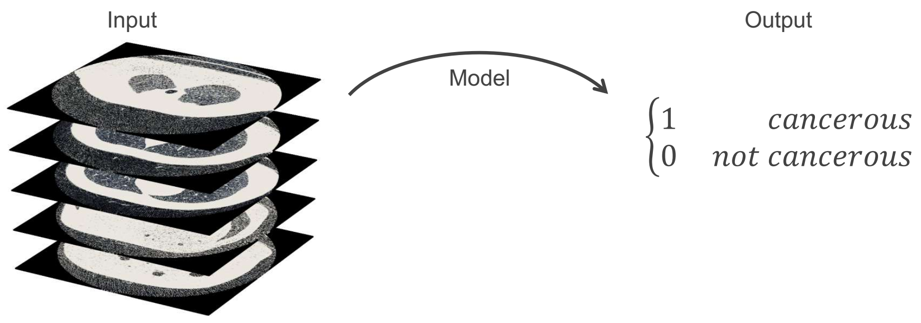


Figure 4.3: Lung cancer image classification.

ResNet [45] without changing the number of function evaluations.[1]

4.4 Image Classification for Lung Cancer Detection

We apply our image classification model to low-dose computed tomography (LDCT) images for lung cancer detection.

4.4.1 Motivation

Cancer claims millions of lives annually with more than 600,000 of those in the United States. Individually, lung cancer (135,000 U.S. deaths) is the most lethal [70]. However, when detected in early stages while still localized to the lungs, this lethal cancer can be treated. Early detection provides this silver lining and a hope for saving and/or extending lives. Observably, lung cancer five-year survival rates range from 59% when localized to 5.8% when distant [70].

The National Comprehensive Cancer Network (NCCN) recommends that asymptomatic high-risk lung cancer patients (30 pack-years,[2] ages 55-74, quit smoking within past 15 years) be screened annually using low-dose computed tomography (LDCT) imaging. LDCT scans work by shooting x-ray photons through the pa-

[1]Code for 2-D images is available at `https://github.com/EmoryMLIP/DynamicBlocks`.

[2]A patient's pack-years are the product of the number of packs smoked daily and the number of years smoked.

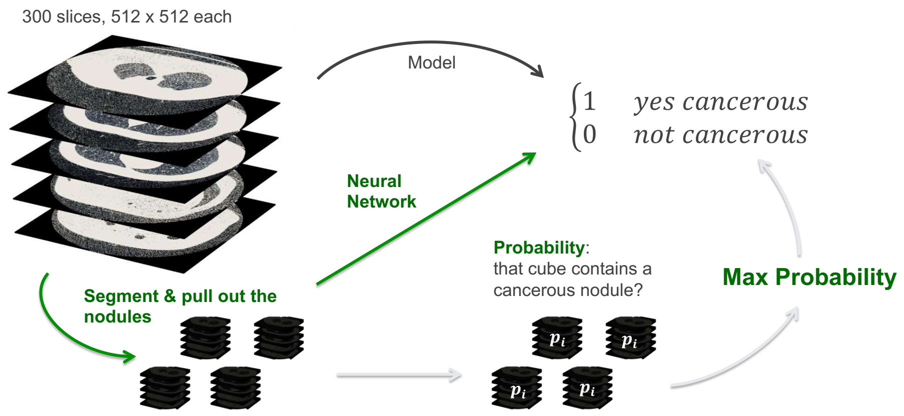

300 slices, 512 x 512 each
Model
$\begin{cases}1 & yes\ cancerous \\ 0 & not\ cancerous\end{cases}$
Neural Network
Segment & pull out the nodules
Probability: that cube contains a cancerous nodule?
Max Probability
p_i

Figure 4.4: Lung cancer model applied to individual cubes.

tient's chest and collecting the attenuation values (the scattering patterns on the other side). From the attenuation values, the CT scanner solves an inverse problem using Radon's transform to generate a CT slice (a 2-D image read by the radiologist). The CT scanner produces multiple slices, which, when stacked together, form a 3-D image representation of the patient's lungs. Radiologists read these slices, looking for nodules that may be cancerous. Cancerous nodules often appear large, oddly shaped, or fast-growing. The NCCN recommends annual low-dose CT screening because it grants higher sensitivity for detection than chest x-rays and lacks the higher radiation of a full-dose CT, which can be detrimental when used annually.

After an LDCT scan and a suspicious determination by the radiologist, the patient undergoes follow-up screening (PET scan or full CT) potentially followed by biopsy. Currently, the LDCT step is marred with false positives (positive predictive value, PPV, of 3.8%) [107]. Thus, 96% of patients marked as likely to have cancer are incorrectly marked and undergo some form of costly follow-up screening and/or biopsy. These additional costs are a mix of patient out-of-pocket costs and insurance company costs, which contribute to the skyrocketing medical insurance rates. The current pipeline suffers from other issues: some doctors are uncomfortable with LDCT and will use chests x-rays instead, patients do not always return for follow-up testing (many high-risk patients have mobility issues or comorbidities), and the entire pipeline can take six months from LDCT scan to biopsy results [37].

We set out to build an NN that reads a patient's three-dimensional LDCT scan and predicts whether that patient has a cancerous nodule or is cancer free (Figure 4.3). Our model should have a PPV greater than 4% while maintaining the same sensitivity of radiologists (94%).

4.4.2 Model

We adjust the image classification approach presented in Section 4.2 to apply to three-dimensional images. Namely, we now have three-dimensional convolutional kernels whereas the rest of the formulation remains the same. We empirically find that an effective lung cancer model requires at least 32 channels for the hidden features of the CNN. As such, the hidden states require much memory when training in the GPU. Since the RAM of the available GPU was 16 GB, we realized the necessity for separating the initial LDCT image into smaller cubes (Figure 4.4). The NN (4.1) predicts a cancer probability for each cube. Then, the model combines all the cubes of a patient, assigning the maximum probability of a patient's cubes for the patient as a whole. Maximum probability performed competitively with other combination methods implemented in Liao et al. [62].

For our model (4.1), we use three neural ODEs f_1, f_2, f_3 with average pooling operators P_1, P_2, P_3 in (4.1). For each neural ODE, we use a double symmetric layer with $T = 4$, three control layers, and five state layers arranged as in Figure 4.2. We choose the α-weighted focal loss for G (4.3), where α is decided dynamically to counter the class imbalance of the provided training data.

4.4.3 National Lung Screening Trial Experiment

We obtained a subset of 15,000 patients from the 26,000 patients who partook in the National Lung Screening Trial (NLST). Although the NLST data set is public,

Table 4.1: Model performance 2 × 2 tables.

	Radiologists [107]		**Google** [3]		**Ours (Validation)**	
	Actual		Actual		Actual	
	True	False	True	False	True	False
Predicted True	270	6,911	82	1,260	40	398
Predicted False	18	19,043	4	5,370	2	1,560
class imbalance	1.1%		1.3%		2.1%	

Table 4.2: Model performance metrics.

Metric	**Radiologists** [107]	**Google** [3]	**Ours (Validation)**
Sensitivity	0.94	0.95	0.95
PPV	0.04	0.06	0.09

only subsets of the data (with a maximum of 15,000 patients) are provided due to privacy concerns. All patients were high-risk and nonsymptomatic at the time of LDCT screening. For each patient, we possess the LDCT scan and a biopsy-confirmed ground truth of cancerous or non-cancerous. We split the data set of patients into separate training, validation, and testing sets (Section 2.1.3). While splitting, we control for the distribution of nodule count and size to aid in generalizability.

Despite long training times, our model achieves competitive performance on the lung cancer detection task for NLST data (Table 4.1, Table 4.2). The radiologists of the NLST [107] used all 26,000 patients. The Google attempt [3] similarly obtained a subset of the NLST data and trained an NN to detect lung cancer. The Google approach predicted more than two classes, matching the Lung CT Screening Reporting & Data System (Lung-RADS) diagnostic classes.

The neural ODE PPV beats both radiologists and Google, while achieving competitive sensitivity (Table 4.2). The class imbalance and size of the data present the two most difficult obstacles. Hence, we acknowledge the differences between all methods' number of patients and the associated class imbalance (Table 4.1).

In deployment, an NN detection model can assist radiologists in their diagnosis of

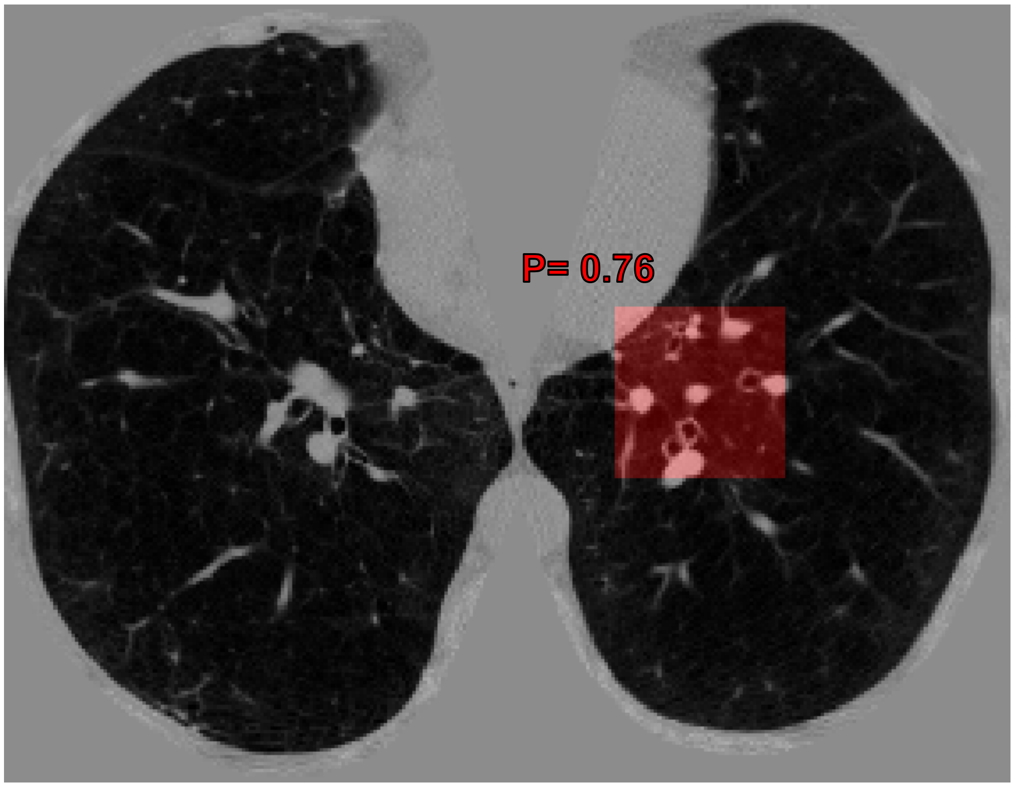


Figure 4.5: Model interpretability for deployment.

lung cancer. In addition to poor PPV, machine learning methods for healthcare fail to be implemented due to a lack of interpretability. Radiologists have a responsibility to their patients and thus need to know when to reject or accept the model's prediction; therefore, they view uninterpretable black box machine learning models negatively. Since we have the predictions for each small LDCT cube (Figure 4.4), we can show though predictions to the radiologist during diagnosis (Figure 4.5). Therefore, the radiologist can somewhat determine why the NN made its prediction, and optionally can double-check a reduced search space.

Chapter 5

Continuous Normalizing Flows for Density Estimation

Existing continuous normalizing flow (CNF) approaches suffer from high costs. We leverage optimal control theory to reduce CNF costs. Similar to Chapter 3, we apply the DO approach and compare training times and performance with the state-of-the-art methods on many real data sets. Next, we develop OT-Flow by incorporating running costs and the value function (Section 2.3). This chapter incorporates portions from Onken and Ruthotto [76] and Onken et al. [77].

5.1 Problem

A normalizing flow [87] is an invertible mapping $f\colon \mathbb{R}^d \to \mathbb{R}^d$ between an arbitrary probability distribution P_0 and a known distribution P_1 whose densities we denote by ρ_0 and ρ_1, respectively. For simplicity, we specify that P_1 is the standard normal distribution with Gaussian density ρ_1. Normalizing flows possess two tiers of reference: a microscopic view and a macroscopic view (Figure 5.1). The microscopic view considers a single sample $\boldsymbol{x} \in \mathbb{R}^d$ drawn from P_0 and how it follows f, flowing to $f(\boldsymbol{x})$. The outputs $f(\boldsymbol{x})$ follow distribution P_1. Alternatively, one can view the normalizing

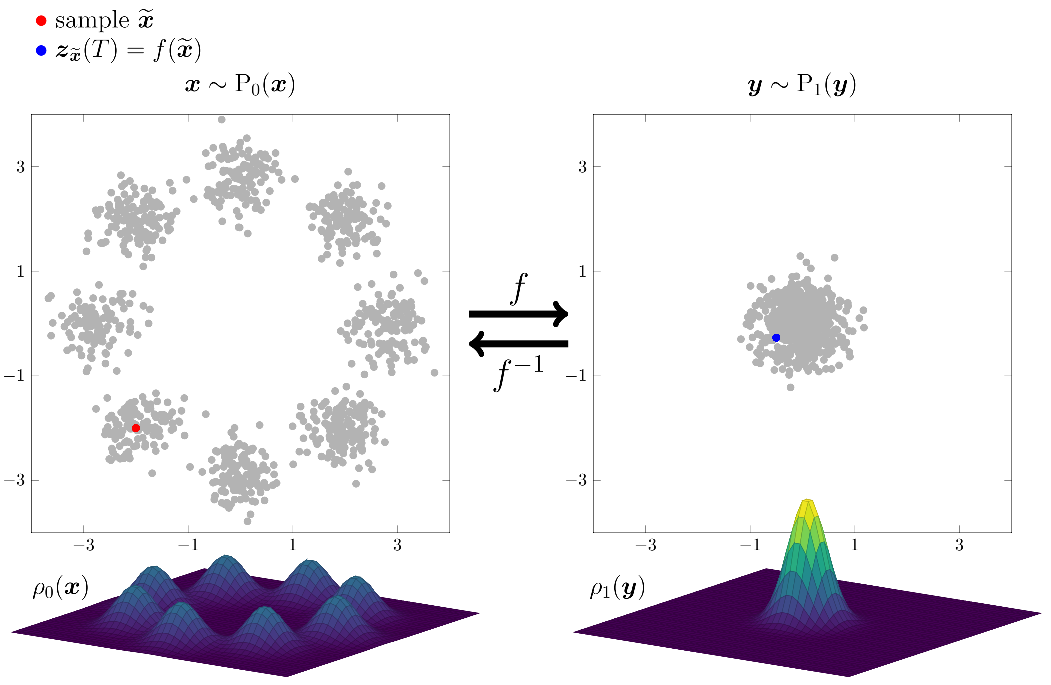


Figure 5.1: Normalizing flow example for the Gaussian mixture problem.

flow from the macroscopic, or density view. By the change of variables formula, for all $\boldsymbol{x} \in \mathbb{R}^d$, the flow must satisfy [81, 87]

$$\log \rho_0(\boldsymbol{x}) = \log \rho_1(f(\boldsymbol{x})) + \log |\det \nabla f(\boldsymbol{x})| \, . \tag{5.1}$$

Effectively, we want f to approximate a diffeomorphism (a bijective function that is differentiable and has a differentiable inverse).

A normalizing flow is constructed by composing invertible neural network layers f_i, where $i = 1, \ldots, k$, i.e.,

$$f = f_k \circ \cdots \circ f_2 \circ f_1. \tag{5.2}$$

Since computing the log-determinant in general requires $\mathcal{O}(d^3)$ floating point operations (FLOPS), effective finite normalizing flows (5.1) consist of layers f_i whose gradients have exploitable structure (e.g., diagonal, triangular, low-rank) [22, 23, 80].

We next convert the normalizing flows into a continuous formulation, where f is modeled by a neural ODE (2.8), i.e., $f(\boldsymbol{x}) = \boldsymbol{z}_{\boldsymbol{x}}(T)$ where $\boldsymbol{z}_{\boldsymbol{x}}(t) \in \mathbb{R}^d$ is the state at time t for initial condition $\boldsymbol{x}$. The change of variables formula (5.1) becomes

$$\log \rho_0(\boldsymbol{x}) = \log \rho\big(\boldsymbol{z}_{\boldsymbol{x}}(t), t\big) + \log\det\big(\nabla \boldsymbol{z}_{\boldsymbol{x}}(t)\big), \tag{5.3}$$

which is the log transform of the Jacobi identity

$$\rho_0(\boldsymbol{x}) = \rho\big(\boldsymbol{z}_{\boldsymbol{x}}(t), t\big)\det\big(\nabla \boldsymbol{z}_{\boldsymbol{x}}(t)\big), \tag{5.4}$$

where $\rho\big(\boldsymbol{z}_{\boldsymbol{x}}(t), t\big)$ is the predicted density at time t. Using Jacobi's formula (different from the Jacobi identity) and (5.3), Chen et al. [19] derive the instantaneous change of variables theorem

$$\frac{\partial \log \rho\big(\boldsymbol{z}_{\boldsymbol{x}}(t), t\big)}{\partial t} = -\operatorname{tr}\big(\nabla \mathbf{v}(\boldsymbol{z}_{\boldsymbol{x}}(t), t\,; \boldsymbol{\theta})\big), \tag{5.5}$$

for layer $\mathbf{v}$ in (2.8). Thus, for a time $t \in [0, T]$,

$$\log \rho\big(\boldsymbol{z}_{\boldsymbol{x}}(t), t\big) - \log \rho\big(\boldsymbol{z}_{\boldsymbol{x}}(0), 0\big) = -\int_0^t \operatorname{tr}\big(\nabla \mathbf{v}(\boldsymbol{z}_{\boldsymbol{x}}(s), s; \boldsymbol{\theta})\big)\,\mathrm{d}s. \tag{5.6}$$

Recall that $\rho\big(\boldsymbol{z}_{\boldsymbol{x}}(0), 0\big) = \rho_0(\boldsymbol{x})$. Furthermore, we define $\ell_{\boldsymbol{x}}(t) = \log \rho\big(\boldsymbol{z}_{\boldsymbol{x}}(t), t\big) - \log \rho_0(\boldsymbol{x})$ so that the final state satisfies $\ell_{\boldsymbol{x}}(T) = \log\det \nabla f(\boldsymbol{x})$. Therefore, we can incorporate the time integration (5.6) into the neural ODE, creating a continuous normalizing flow (CNF) [19, 39]

$$\partial_t \begin{bmatrix} \boldsymbol{z}_{\boldsymbol{x}}(t) \\ \ell_{\boldsymbol{x}}(t) \end{bmatrix} = \begin{bmatrix} \mathbf{v}\big(\boldsymbol{z}_{\boldsymbol{x}}(t), t\,; \boldsymbol{\theta}\big) \\ \operatorname{tr}\big(\nabla \mathbf{v}(\boldsymbol{z}_{\boldsymbol{x}}(t), t\,; \boldsymbol{\theta})\big) \end{bmatrix}, \quad \begin{bmatrix} \boldsymbol{z}_{\boldsymbol{x}}(0) \\ \ell_{\boldsymbol{x}}(0) \end{bmatrix} = \begin{bmatrix} \boldsymbol{x} \\ 0 \end{bmatrix}, \tag{5.7}$$

for artificial time $t \in [0, T]$ and $\boldsymbol{x} \in \mathbb{R}^d$. Addressing the microscopic view, the first component maps a point $\boldsymbol{x}$ to $f(\boldsymbol{x}) = \boldsymbol{z}_{\boldsymbol{x}}(T)$ by following the trajectory $\boldsymbol{z}\colon \mathbb{R}^d \times [0, T] \to$

$\mathbb{R}^d$ (Figure 5.1). This mapping is invertible and orientation-preserving under mild assumptions on the dynamics $\mathbf{v}\colon \mathbb{R}^d \times [0,T] \to \mathbb{R}^d$. Replacing the log determinant (5.1) with a trace (5.7) reduces the FLOPS to $\mathcal{O}(d^2)$ for exact computation or $\mathcal{O}(d)$ for an unbiased estimate [32, 39, 119] (Section 5.2.2).

We train the dynamics by matching the final-time predicted density $\rho\big(\boldsymbol{x},T\big)$ to the desired density ρ_1. The error between the two densities is measured using the Kullback-Leibler (KL) divergence

$$\mathbb{D}_{\mathrm{KL}}\big[\,\rho(\boldsymbol{x},T)\,||\,\rho_1(\boldsymbol{x})\,\big] = \int_{\mathbb{R}^d} \log\left(\frac{\rho(\boldsymbol{x},T)}{\rho_1(\boldsymbol{x})}\right) \rho(\boldsymbol{x},T)\,\mathrm{d}\boldsymbol{x}. \tag{5.8}$$

Changing variables, and using (5.4), we can rewrite (5.8) as

$$\begin{aligned}
&\mathbb{D}_{\mathrm{KL}}\big[\,\rho\big(\boldsymbol{z}_{\boldsymbol{x}}(T),T\big)\,||\,\rho_1\big(\boldsymbol{z}_{\boldsymbol{x}}(T)\big)\,\big]\\
&= \int_{\mathbb{R}^d} \log\left(\frac{\rho\big(\boldsymbol{z}_{\boldsymbol{x}}(T),T\big)}{\rho_1\big(\boldsymbol{z}_{\boldsymbol{x}}(T)\big)}\right) \rho\big(\boldsymbol{z}_{\boldsymbol{x}}(T),T\big)\det\big(\nabla \boldsymbol{z}_{\boldsymbol{x}}(T)\big)\,\mathrm{d}\boldsymbol{x},\\
&= \int_{\mathbb{R}^d} \log\left(\frac{\rho_0(\boldsymbol{x})}{\rho_1\big(\boldsymbol{z}_{\boldsymbol{x}}(T)\big)\,\det\big(\nabla \boldsymbol{z}_{\boldsymbol{x}}(T)\big)}\right)\rho_0(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x},\\
&= \int_{\mathbb{R}^d} \Big[\log\big(\rho_0(\boldsymbol{x})\big) - \log\big(\rho_1(\boldsymbol{z}_{\boldsymbol{x}}(T))\big) - \log\det\big(\nabla \boldsymbol{z}_{\boldsymbol{x}}(T)\big)\Big]\rho_0(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x}.
\end{aligned} \tag{5.9}$$

For normalizing flows, we assume ρ_1 is the density associated with the standard normal

$$\rho_1(\boldsymbol{x}) = \frac{1}{\sqrt{(2\pi)^d}}\exp\left(\frac{-\|\boldsymbol{x}\|^2}{2}\right), \tag{5.10}$$

which implies

$$\log\big(\rho_1(\boldsymbol{z}_{\boldsymbol{x}}(T))\big) = -\frac{1}{2}\|\boldsymbol{z}_{\boldsymbol{x}}(T)\|^2 - \frac{d}{2}\log(2\pi). \tag{5.11}$$

Substituting (5.11) into (5.9), we obtain

$$\begin{aligned}
&\mathbb{D}_{\mathrm{KL}}\left[\,\rho\big(\boldsymbol{z}_{\boldsymbol{x}}(T),T\big)\,||\,\rho_1(\boldsymbol{z}_{\boldsymbol{x}}(T))\,\right] \\
&= \int_{\mathbb{R}^d}\Big[\log\big(\rho_0(\boldsymbol{x})\big) - \log\det\big(\nabla\boldsymbol{z}_{\boldsymbol{x}}(T)\big) + \frac{1}{2}\|\boldsymbol{z}_{\boldsymbol{x}}(T)\|^2 + \frac{d}{2}\log(2\pi)\Big]\rho_0(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} \\
&= \int_{\mathbb{R}^d}\Big[\log\big(\rho_0(\boldsymbol{x})\big) + C_{\boldsymbol{x}}(T)\Big]\rho_0(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} \qquad (5.12)\\
&= \mathop{\mathbb{E}}_{\boldsymbol{x}\sim\mathrm{P}_0(\boldsymbol{x})}\big\{\log\big(\rho_0(\boldsymbol{x})\big) + C_{\boldsymbol{x}}(T)\big\}\,, \\
&\qquad\text{where}\quad C_{\boldsymbol{x}}(T) \coloneqq \frac{1}{2}\|\boldsymbol{z}_{\boldsymbol{x}}(T)\|^2 - \ell_{\boldsymbol{x}}(T) + \frac{d}{2}\log(2\pi).
\end{aligned}$$

Density $\rho_0(\boldsymbol{x})$ is unknown in density estimation Thus, the term $\log(\rho_0(\boldsymbol{x}))$ is dropped, and normalizing flows minimize C alone. Subtracting this constant does not affect the minimizer. Thus, CNFs solve the optimization problem [39, 80, 81, 87]

$$\min_{\boldsymbol{\theta}} \quad \mathop{\mathbb{E}}_{\boldsymbol{x}\sim\mathrm{P}_0(\boldsymbol{x})}\{C_{\boldsymbol{x}}(T)\} \quad \text{subject to} \quad (5.7). \qquad (5.13)$$

Solving (5.13) often comes with high costs for two predominant reasons: the large number of function evaluations (NFE) needed to solve the ODE (5.7) and the expensive trace computation that occurs for every function evaluation. Next, we present existing approaches to mitigate these costs (Section 5.2). Then we present our two models to mitigate such costs: the DO approach (Section 5.3) and OT-Flow (Section 5.4).

5.2 Related Works

We present a non-exhaustive summary of existing approaches split into three categories: finite flows (or discrete flows) that solve the normalizing flow problem (5.1) without a continuity requirement, approaches to the CNF problem with emphasis on the use of trace estimation for training efficiency, and CNF approaches that incorporate OC theory.

5.2.1 Finite Flows

Normalizing flows [55, 81, 87, 105] use a composition of discrete transformations as described in (5.2), where specific architectures are chosen for f_i to allow for efficient inverse and Jacobian determinant computations. NICE [22], RealNVP [23], IAF [54], and MAF [80] use either autoregressive or coupling flows where the gradient is triangular, so the gradient determinant can be tractably computed. GLOW [53] expands upon RealNVP by introducing an additional invertible convolution step. These flows are based on either coupling layers or autoregressive transformations, whose tractable invertibility allows for density evaluation and generative sampling. Neural Spline Flows [24] use splines instead of the coupling layers used in GLOW and RealNVP. Using monotonic neural networks, NAF [48] requires positivity of the weights. UMNN [113] circumvents this requirement by parameterizing the Jacobian and then integrating numerically.

5.2.2 Infinitesimal Flows

Modeling flows with differential equations is natural and common [71, 94, 104, 115]. In particular, CNFs [18, 19, 39] model their flow via (5.7). The continuous nature of CNFs tend to result in fewer parameters and more accurate inverses than finite flows.

Trace Estimation

The computation of the trace term in (5.7) occurs at every time step of every training iteration. When computed exactly using existing $\mathcal{O}(d^2)$ approaches, the trace computation becomes a computational bottleneck in CNF training—and leads to high costs during inference. Typically, the exact trace is computed by

$$\operatorname{tr}(\nabla \mathbf{v}) = \sum_{i=1}^{d} \left\{ \boldsymbol{e}_i^\top \, \nabla \mathbf{v} \; \boldsymbol{e}_i \right\} \tag{5.14}$$

where $\boldsymbol{e}_i$ is the ith standard basis vector. In implementation, AD computes the Jacobian-vector product $\nabla \mathbf{v} \; \boldsymbol{e}_i$. The multiplication with $\boldsymbol{e}_i^\top$ is implemented via indexing operations. Each of the d Jacobian-vector products is $\mathcal{O}(d)$. Since most AD toolboxes do not perform multiple Jacobian-vector products simultaneously, a loop is required in implementation and computing (5.14) costs $\mathcal{O}(d^2)$ FLOPS.

To alleviate the expensive training costs of CNFs, FFJORD [39] sacrifices the exact but slow trace computation for a Hutchinson's trace estimator with reduced time complexity [49]. The Hutchinson's trace estimator is computed via

$$\operatorname{tr}(\nabla \mathbf{v}) = \mathop{\mathbb{E}}_{\nu(\boldsymbol{\epsilon})} \left\{ \boldsymbol{\epsilon}^\top \, \nabla \mathbf{v} \; \boldsymbol{\epsilon} \right\} \tag{5.15}$$

for noise vector $\boldsymbol{\epsilon}$ with density $\nu(\boldsymbol{\epsilon})$, $\mathbb{E}\{\boldsymbol{\epsilon}\} = 0$, $\operatorname{Cov}(\boldsymbol{\epsilon}) = \boldsymbol{I}$. Leveraging AD's efficiency with Jacobian-vector products, FFJORD uses AD to compute (5.15) with a single noise vector $\boldsymbol{\epsilon}$ drawn from the Rademacher distribution for complexity $\mathcal{O}(d)$ FLOPS [39]. The Hutchinson's estimator has improved accuracy when replacing $\boldsymbol{\epsilon}$ with a matrix of noise vectors (Figure 5.2d). The Hutchinson's estimator only makes sense to be used for fewer than d noise vectors since we can compute the exact trace with d standard basis vectors at similar cost using (5.14).

The Hutchinson's estimator helps FFJORD achieve training tractability by reducing the trace cost from $\mathcal{O}(d^2)$ to $\mathcal{O}(d)$ per time step. However, during inference, FFJORD has $\mathcal{O}(d^2)$ trace computation cost since accurate CNF inference requires the exact trace. FFJORD also uses the OD approach and an adjoint-based backpropagation like in the Rackauckas et al. [84] approach for time-series (Section 3.2).

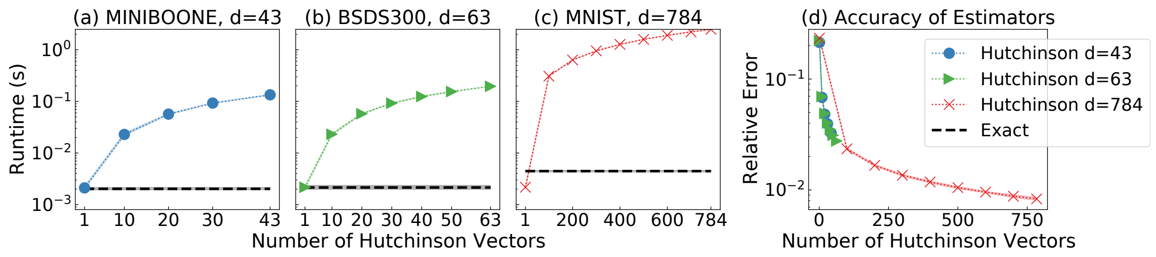


Figure 5.2: Performance comparison of trace computation using exact approach (Section 5.4.2) and Hutchinson's trace estimator using AD. (a-c): runtimes (in seconds) over dimensions 43, 63, and 784, corresponding to the MINIBOONE, BSDS300, and MNIST data sets, respectively. (d): relative errors vs. number of Hutchinson vectors for different dimensions. We present means with shaded 99% error bounds computed from twenty runs via bootstrapping [27] (Appendix B).

5.2.3 Flows Influenced by Optimal Control

To encourage straight trajectories and reduce costs, RNODE [32] regularizes FFJORD with a running cost L. RNODE also includes the Frobenius norm of the Jacobian $\|\nabla \mathbf{v}\|_F^2$ to stabilize training. They estimate the trace and the Frobenius norm using a stochastic estimator and report 2.8x speedup. Numerically, RNODE, FFJORD, and our methods differ. Clearly, DO favors the DO approach in contrast to the OD approach used in FFJORD and RNODE. Furthermore, OT-Flow's exact trace allows for stable training without $\|\nabla \mathbf{v}\|_F^2$ (Figure 5.8). In formulation, OT-Flow shares the L_2 cost with RNODE but follows a potential flow approach (Table 5.1).

Monge-Ampère Flows [119] and Potential Flow Generators [118] draw from OC theory and parameterize the value function (Table 5.1). However, OT-Flow's numerics differ substantially due to our scalable exact trace computation (Section 5.4.2). Optimal transport is also used in other generative models [5, 58, 95, 96, 106].

5.3 Discretize-Optimize Flows

We augment the FFJORD model with the DO approach addressed in Section 3.2 to alleviate the expensive training costs stemming from high NFE. Drawing on the

Table 5.1: The CNF methods we address share the underlying neural ODEs but differ in use of value function Φ, penalizers (L, R, $\|\nabla \mathbf{v}\|_F^2$), ODE solver, approach (DO or OD), and trace computation (exact using AD, Hutchinson's estimator with a single vector sampled from a Rademacher or Gaussian distribution).

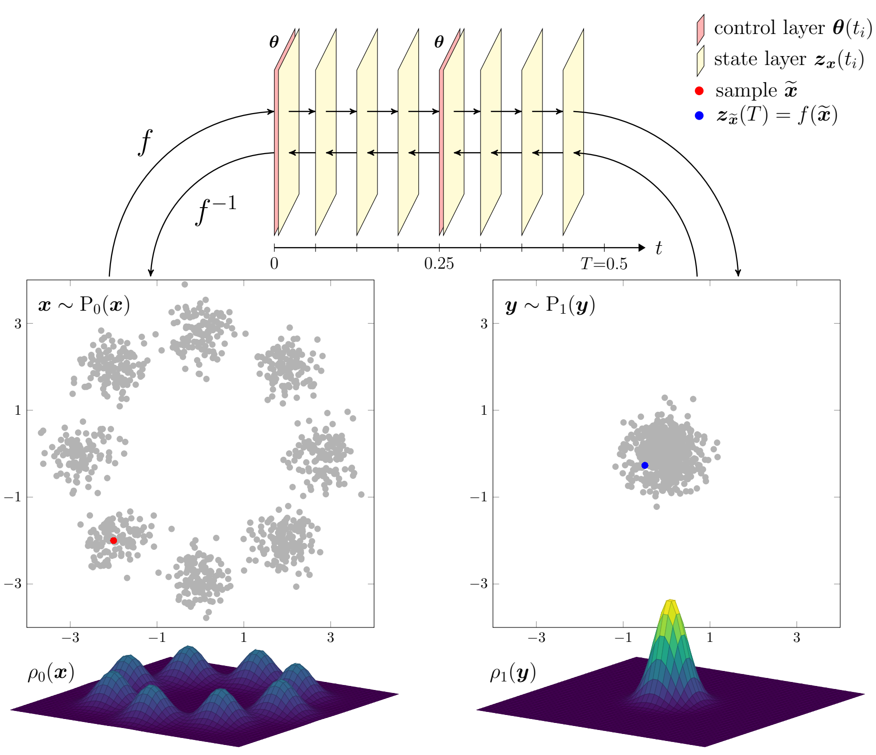
Model	Formulation				Training Implementation			Inference
	Φ	L	R	$\|\nabla \mathbf{v}\|_F^2$	Solver	Approach	Trace	Trace
FFJORD [39]	✗	✗	✗	✗	RK(4)5	OD	Hutch Rad	AD exact
DO	✗	✗	✗	✗	RK4	DO	Hutch Rad	AD exact
RNODE [32]	✗	✓	✗	✓	RK4	OD	Hutch Rad	AD exact
M-A Flows [119]	✓	✗	✗	✗	RK4	DO	Hutch Gauss	
PFG [118]	✓	✗	✓	✗	RK1	DO	AD exact	
OT-Flow	✓	✓	✓	✗	RK4	DO	efficient exact (Sec. 5.4.2)	

Table 5.2: Number of parameters comparison with discrete normalizing flows.

Data Set Dimension d	Power 6	Gas 8	Hepmass 21	Miniboone 43	Bsds300 63
OT-Flow	18K	127K	72K	78K	297K
FFJORD, RNODE, DO	43K	279K	547K	821K	6.7M
NAF[48]	414K	402K	9.27M	7.49M	36.8M
UMNN [113]	509K	815K	3.62M	3.46M	15.6M

stepsize discussion in Section 3.2, we know that a too coarse stepsize results in a finite flow that is a discretization of the CNF. The DO model then can suffer from some of the same problems as finite flows, specifically poor invertibility. However, CNFs tend to have fewer parameters than finite flows (Table 5.2).

A meaningful CNF must be invertible, i.e., the reverse mode must also push-forward ρ_1 to ρ_0, which is not enforced in training. For discrete normalizing flows, computing f^{-1} involves computing the inverse of (5.2) by using the inverse of each piece f_i, which are invertible by design. Computing the inverse of a CNF is performed by integrating the neural ODE backwards in time, i.e., from T to 0. We stress the importance of checking model invertibility. To gauge the accuracy of the trained flow



control layer $\boldsymbol{\theta}(t_i)$
state layer $\boldsymbol{z}_{\boldsymbol{x}}(t_i)$
sample $\widetilde{\boldsymbol{x}}$
$\boldsymbol{z}_{\widetilde{\boldsymbol{x}}}(T) = f(\widetilde{\boldsymbol{x}})$
$\boldsymbol{\theta}$
f
f^{-1}
t
0
0.25
T=0.5
$\boldsymbol{x} \sim \mathrm{P}_0(\boldsymbol{x})$
$\boldsymbol{y} \sim \mathrm{P}_1(\boldsymbol{y})$
$\rho_0(\boldsymbol{x})$
$\rho_1(\boldsymbol{y})$

Figure 5.3: DO approach for the Gaussian mixture problem in Section 5.3.1. As common in OC, we use different discretization points for the weights and the states. We refer to the discretizations as control and state layers, respectively. We discretize the state equation using a Runge-Kutta 4 scheme with constant stepsize $h = 1$.

model, we compute the inverse error

$$\mathop{\mathbb{E}}_{\boldsymbol{x}\sim\mathrm{P}_0(\boldsymbol{x})} \left\| f^{-1}\left(f(\boldsymbol{x})\right) - \boldsymbol{x} \right\|_2 \tag{5.16}$$

where $\boldsymbol{x}$ is sampled from P_0. This calculates the Euclidean distance between an initial point $\boldsymbol{x}$ and the result from mapping $\boldsymbol{x}$ forward then back.

5.3.1 Numerical Experiments

We compare DO and OD on a toy Gaussian mixture problem and five high-dimensional real data sets.

Toy Gaussian Mixture Problem

To help visualize CNFs, consider the synthetic test problem where ρ_0 is the Gaussian mixture obtained by averaging eight bivariate Gaussians situated in a circular pattern about the origin (Figure 5.3). For the DO approach, we use a Runge-Kutta 4 (RK4) solver with a constant stepsize h. We illustrate an example in Figure 5.3, where for final-time $T = 0.5$, the control is discretized at $\boldsymbol{\theta}(0)$ and $\boldsymbol{\theta}(0.25)$. The state layers are discretized at one-quarter intervals as determined by the RK4 scheme. Overall, the forward pass requires eight evaluations of $\mathbf{v}$ and the weights at intermediate time points are obtained by interpolating the two control layers (illustration in Figure 5.3).

We use the neural ODE proposed in Grathwohl et al. [39], where $\mathbf{v}$ is given by

$$\begin{aligned}\mathbf{v}\big(\boldsymbol{z}(t), t; \boldsymbol{\theta}(t)\big) =& \kappa_{64,2}\big(\cdot, t; \boldsymbol{\theta}_4(t)\big) \circ \kappa_{64,64}\big(\cdot, t; \boldsymbol{\theta}_3(t)\big) \\ & \circ \kappa_{64,64}\big(\cdot, t; \boldsymbol{\theta}_2(t)\big) \circ \kappa_{2,64}\big(\boldsymbol{z}(t), t; \boldsymbol{\theta}_1(t)\big).\end{aligned}$$

The so-called concatsquash layer $\kappa_{i,j}(\boldsymbol{x}, t;\, \boldsymbol{\theta})$ maps features $\boldsymbol{x} \in \mathbb{R}^i$ to outputs in $\mathbb{R}^j$ and is defined as

$$\begin{aligned}\kappa_{i,j}(\boldsymbol{x}, t;\, \boldsymbol{\theta}) &= (\boldsymbol{D}_2\boldsymbol{x}) \odot (\sigma \circ \boldsymbol{D}_1 t) + (\boldsymbol{D}_0 t) \\ &= (\boldsymbol{K}_2\boldsymbol{x} + \boldsymbol{b}_2) \odot \sigma(\boldsymbol{K}_1 t + \boldsymbol{b}_1) + (\boldsymbol{K}_0 t),\end{aligned} \tag{5.17}$$

where the parameters $\boldsymbol{\theta}$ are: $\boldsymbol{K}_0, \boldsymbol{K}_1, \boldsymbol{b}_1, \boldsymbol{b}_2 \in \mathbb{R}^j$ and $\boldsymbol{K}_2 \in \mathbb{R}^{j\times i}$. The nonlinear activation function σ is the element-wise sigmoid function $\sigma(x) = 1/(1 + \exp(-x))$, and $\boldsymbol{D}_0$ uses no bias. This layer accepts and operates on the space features and the time separately.

To solve the problem, the CNF needs to be trained to push the Gaussian mixture P_0 to the center to form P_1 (Figure 5.1). To discretize the loss function, we use samples drawn from the Gaussian mixture P_0. The reverse mode of the model will separate the large Gaussian back into the eight terms of the Gaussian mixture. What makes the problem challenging is that the true inverse is discontinuous at the origin.

A relatively small network model consisting of one control layer accurately solves the problem (Table 5.3). For a fixed 10,000 iterations, the DO approach with $h = 0.05$ demonstrates a slight speedup per iteration and achieves a similar testing loss as the OD approach; also, the DO solution has a comparable inverse error.

The DO approach with $h = 0.25$ has the lowest training time and achieves a spuriously low loss value; however, because the stepsize is too large, the discrete model loses its invertibility (Section 5.3.1). Decreasing the time step to $h = 0.05$ still leads to a substantial reduction in the number of function evaluations, comparable loss, and comparable inverse error than the OD approach.

Too Coarse Time Stepping

Using a too large stepsize, h=0.25, the DO model may achieve a low testing loss while losing the invertibility. In fact, this flow fails to accurately align the densities.

A sufficiently small stepsize is needed to ensure the discrete forward propagation adequately represents the neural ODE [4]; however, a too small stepsize leads to excessive and unnecessary computation and training time. Hence, there is a trade-off between accuracy and speed. For the Gaussian mixture problem, we trained two DO networks with stepsizes h=0.05 and h=0.25 (Table 5.3). The h=0.25 stepsize showed an inverse error much higher than the other two because the larger stepsize defines a coarser state grid, which leads to a less-accurate approximation for the ODE (2.8).

The coarse model (h=0.25) achieves a loss on par with the fine grid models and an inverse error that is accurate for a couple decimal points (Table 5.3). However, visually, the forward propagation leaves artifacts and does not create a smooth Gaussian (Figure 5.4 middle row). Also, the inverse flow does not fully match the initial distribution. Therefore, we observe issues with a coarse discrete model despite a competitive loss. Re-discretizing the coarse model after training, we can switch to a finer

Table 5.3: DO and OD Gaussian mixture results. The DO approach with $h = 0.25$ has the lowest training time and achieves a spuriously low loss; however, because the stepsize is too large, the discrete model loses its invertibility (Section 5.3.1). Decreasing the time step to $h = 0.05$ still leads to a substantial reduction in the number of function evaluations (NFE), comparable loss, and comparable inverse error to the OD approach.

Data Set	Model		Training		Testing	
	Title	Solver	$\frac{\text{Time}}{\text{Epoch}}$ (s)	NFE	Loss C	Inv Err
Gaussian Mixture	FFJORD	OD RK(4)5	0.79	65	2.83	8.88e-7
	DO	DO RK4 $h = 0.05$	0.48	40	2.79	1.82e-8
	DO	DO RK4 $h = 0.25$	0.10	8	2.83	1.47e-2

solver for testing (Figure 5.4 bottom row), which smooths some of the artifacts in the central Gaussian and the inverse plot. The coarse model without changes (Figure 5.4 middle row) may be sufficient for some application because the inverse error could be sufficiently small.

High-Dimensional Density Estimation

We now consider some high-dimensional instances of the CNF problem arising in density estimation; the public data sets we use are listed in Table 5.5. A detailed description of the data sets is given in Papamakarios et al. [80]. In short, Miniboone, Power, Hepmass, and Gas are data sets compiled for various tasks and housed by the University of California, Irvine (UCI) Machine Learning Repository. The contexts include neutrinos, electric power consumption of houses, particle-producing collisions, and gas sensors, respectively. The Berkeley Segmentation Data Set (Bsds300) contains image patches for image segmentation and boundary detection. Papamakarios et al. [80] preprocessed these data sets for use in density estimation. These data sets have also been used to validate the OD approach in FFJORD [39].

For each data set, we define a neural layer $\mathbf{v}$ via concatenations of concatsquash layer $\kappa_{i,j}(\boldsymbol{z}, t; \boldsymbol{\theta})$ (5.17) that are more complicated than for the Gaussian mixture

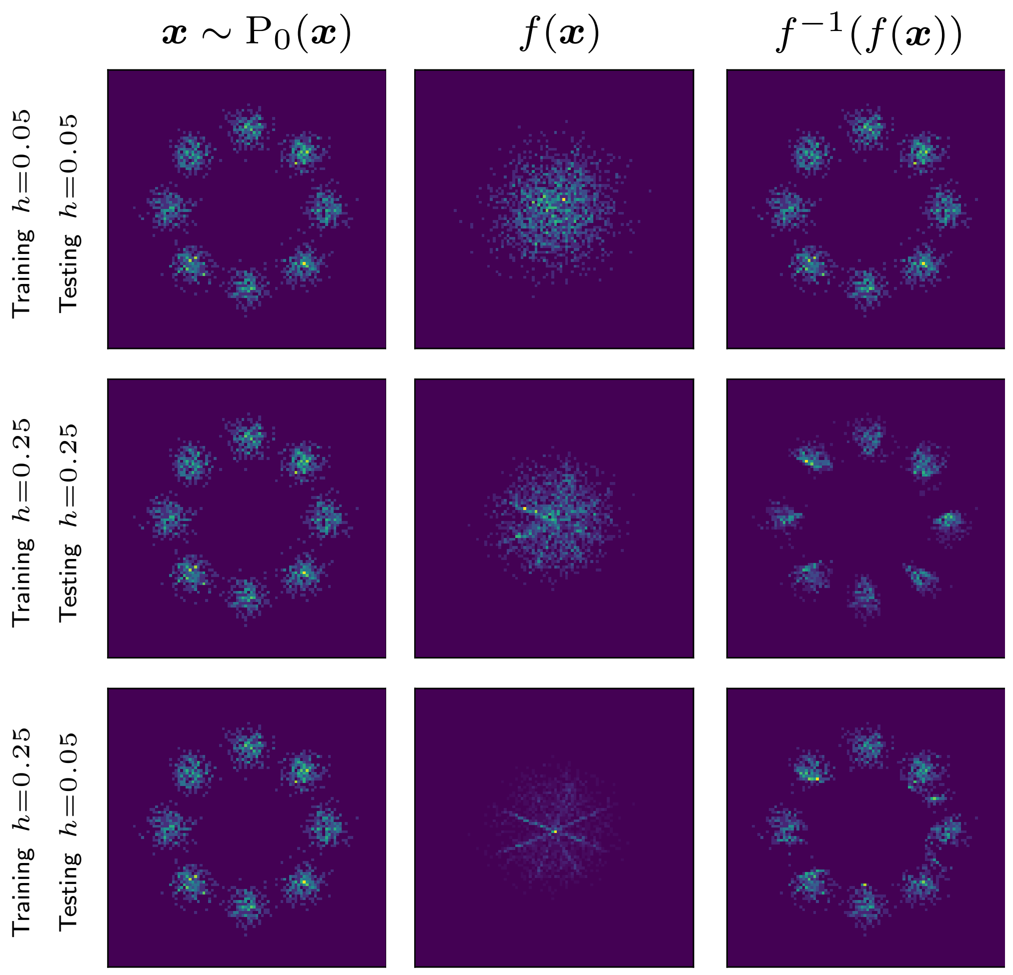


Figure 5.4: Using a stepsize not sufficiently small enough in the DO method can result in lack of invertibility.

problem. For hyperparameters, we can select an activation function σ, number of hidden layers n_h, number of flow steps χ, and the number of hidden dimensions, which we choose based on some multiplier o of the number of input features (Table 5.4). Our general layer $\mathbf{v}$ then is

$$\mathbf{v}(\boldsymbol{z}, t; \boldsymbol{\theta}) = \Big[\kappa_{od,d}(\cdot, t; \boldsymbol{\theta}) \circ \kappa_{od,od}(\cdot, t; \boldsymbol{\theta})^{n_h - 1} \circ \kappa_{d,od}(\boldsymbol{z}, t; \boldsymbol{\theta})\Big]^{\chi}. \tag{5.18}$$

Each of the χ blocks starts with one concatsquash layer that maps from $\mathbb{R}^d$ to $\mathbb{R}^{od}$, then $n_h{-}1$ more concatsquash layers that maintain the dimensionality, then a final

Table 5.4: CNF hyperparameters, including stepsize h, activation function σ, number of hidden layers n_h, dimension multiplier o, and the number of flow steps χ for (5.18). The OD hyperparameters were used for both FFJORD [39] and RNODE [32]. All used initial learning rate 1e-3. DO often used smaller batch sizes so that the batch would fit into the Titan X GPU with 12GB RAM.

Data Set	d	Model	h	batch size	σ	n_h	o	χ
Power	6	OD	-	30000	tanh	3	10	5
		DO	0.10	10000				
Gas	8	OD	-	5000	tanh	3	20	5
		DO	0.10	5000				
Hepmass	21	OD	-	10000	softplus	2	10	10
		DO	0.10	5000				
Miniboone	43	OD	-	5000	softplus	2	20	1
		DO	0.25	5000				
Bsds300	63	OD	-	10000	softplus	3	20	2
		DO	0.05	500				

concatsquash layer that maps back to $\mathbb{R}^d$. Depending on the difficulty of the data set, different hyperparameters are selected (Table 5.4); we mostly followed the hyperparameters tuned by FFJORD [39].

The performance on the testing sets of all data sets demonstrates that the OD and DO approaches result in similar negative log-likelihood loss (Table 5.5). However, DO does so more quickly on most of the data sets but with greater inverse error. Although the AD used in DO is faster than the adjoint-based recalculation used in OD for batches of the same size, the memory requirements of storing intermediates in AD restrict the DO batch sizes to be much smaller than the OD batch sizes. Checkpointing can also handle the memory constraints. Nonetheless, the OD method for solving the Hepmass density estimation trains faster than DO because of the larger batches. Since we use the same models, code, and hyperparameters (Table 5.4) as FFJORD [39], we witness similar testing losses (Table 5.6).

We observe DO converge in less time for most data sets, but no timing payoff for Hepmass (Table 5.5). Even so, on average for the five high-dimensional data sets,

Table 5.5: DO reduces the training times. Negative log-likelihood loss (lower is better) in nats (natural unit of information). The number of functions evaluations for the forward propagation (NFE) are averaged over all the batches and epochs. The OD approach for Bsds300 was terminated prematurely after nearly seven days of training. While the discrete model in DO can have a large inverse error with the stepsize used in training, re-discretizing in the inference phase can reduce this error substantially. In summary, DO reduces the training times and yields invertible models with comparable performance to OD.

		Training	Inference	Training		Testing		
		Solver	Solver	Time (s)	NFE	NFE	Loss	Inv Err
Power	OD	RK(4)5	RK(4)5	248K	583	649	-0.37	7.60e-6
	DO	RK4 h=0.10	RK(4)5	56.0K	200	2066	-0.25	1.58e-5
	DO	RK4 h=0.10	RK4 h=0.10	56.0K	200	200	-0.33	4.23e-3
Gas	OD	RK(4)5	RK(4)5	271K	475	527	-10.69	1.78e-5
	DO	RK4 h=0.10	RK(4)5	121K	200	437	-10.27	4.70e-6
	DO	RK4 h=0.10	RK4 h=0.10	121K	200	200	-10.27	6.63e-3
Hepmass	OD	RK(4)5	RK(4)5	357K	770	866	16.13	5.65e-6
	DO	RK4 h=0.10	RK(4)5	281K	400	765	16.60	1.09e-6
	DO	RK4 h=0.10	RK4 h=0.10	281K	400	400	16.60	2.25e-4
Miniboone	OD	RK(4)5	RK(4)5	32.4K	115	132	10.57	4.80e-6
	DO	RK4 h=0.25	RK(4)5	3.77K	16	118	10.50	2.17e-6
	DO	RK4 h=0.25	RK4 h=0.25	3.77K	16	16	10.45	1.45e-3
Bsds300	OD	RK(4)5	RK(4)5	598K	345	544	-133.94	3.41e-6
	DO	RK4 h=0.05	RK(4)5	49.7K	160	511	-146.23	1.82e-6
	DO	RK4 h=0.05	RK4 h=0.05	49.7K	160	160	-146.14	3.75e-4

DO offers a 6x speedup in training over the OD method. All OD and DO models are competitive in performance with other density estimation models (Table 5.6).

Re-Discretizing the Trained Flows

Re-discretization in the inference phase presents a straightforward idea for reducing the inversion error of DO trained models.

Using the Miniboone data set, we consider training a CNF with the DO approach with RK4 and h=0.25. Training takes less than 3,800 seconds, roughly one-tenth of the time required by the OD (Table 5.5). After training (e.g., during inference or

Table 5.6: Testing loss C comparison with other methods.

	Power	Gas	Hepmass	Miniboone	Bsds300
FFJORD [39] (OD)	-0.37	-10.69	16.13	10.57	-133.94†
RNODE [32]	-0.39	-11.10	16.37	10.65	-129.75†
DO	-0.25	-10.27	16.60	10.50	-146.23
OT-Flow	-0.30	-9.20	17.32	10.55	-154.20
MADE [35]	3.08	-3.56	20.98	15.59	-148.85
RealNVP [23]	-0.17	-8.33	18.71	13.55	-153.28
Glow [53]	-0.17	-8.15	18.92	11.35	-155.07
MAF [80]	-0.24	-10.08	17.70	11.75	-155.69
NAF [48]	-0.62	-11.96	15.09	8.86	-157.73
UMNN [113]	-0.63	-10.89	13.99	9.67	-157.98

†Training terminated before convergence.

model deployment) we re-discretize the model using stepsize h=0.05 (Table 5.7). This maintains a similar testing loss of 10.50 while substantially reducing the inverse error. Demonstrating the flexible choice of solver, we also consider the adaptive RK(4)5 solver in the evaluation (which has 118 function evaluations).

Since the DO model was trained using a fixed stepsize, the generalization with respect to re-discretization is remarkable; note the low loss observed for all time integrators. As to be expected, the inverse error is reduced for more accurate time integrators. Less accurate time integrators can use their error to their advantage and result in lower loss values while sacrificing invertibility. Therefore, both loss and invertibility should be used to evaluate CNF performance. This observation motivates the use of a different metric in Section 5.4.

Multilevel Training

Models trained with finer grids tend to have better convergence than models trained with coarser grids; however, these fine grid models consume much more training time from the large NFE. Taking a *multilevel* approach, we train the initial epochs with a coarse grid (few state layers), then add state layers to make the grid finer for the

Table 5.7: Model trained using the solver associated with **bold** values. All other results in the column result from changing the solver settings (and thus the state discretization) for the forward propagation when running on the test data. Since the DO problem is solved with a fixed stepsize and not using adaptive integration used in OD, the generalization to other discretizations is notable.

	Trained via DO			**Trained via OD**		
	Testing Loss C	NFE	Inverse Error	Testing Loss C	NFE	Inverse Error
Miniboone						
RK(4)5 rtol 1e-1 atol 1e-3	10.075	28	4.25e-2	10.108	29	3.16e-2
RK(4)5 rtol 1e-2 atol 1e-4	10.529	38	3.55e-3	10.669	38	1.61e-2
RK(4)5 rtol 1e-6 atol 1e-8	10.502	118	2.17e-7	**10.636**	**132**	**1.85e-7**
RK4 h=0.05	10.502	80	3.40e-7	10.636	80	5.22e-7
RK4 h=0.25	**10.454**	**16**	**1.45e-3**	10.590	16	2.57e-3
RK4 h=0.50	10.122	8	2.97e-2	10.239	8	4.79e-2
RK4 h=1.0	8.916	4	2.77e-1	9.082	4	3.70e-1

final training epochs.

We test the multilevel strategy for the DO and OD approaches on the Miniboone data set (Figure 5.5). For both approaches, we train the first 500 epochs with a coarse grid and initial learning rate 1e-3 then the next 1500 epochs with a finer grid (using the ODE solver in Table 5.5) while lowering the learning rate to 5e-4. For DO, we start with an RK 4 scheme using h=0.50 then switch the stepsize to h=0.25. For OD, we start with RK(4)5 with relative tolerance 1e-3 and absolute tolerance 1e-5, then switch to the default RK(4)5 (dividing each tolerance by 10^3). For both approaches, when we switch grids at epoch 500, we witness a large uptick in the loss, which quickly decays. Overall, the performance of the multilevel schemes is comparable in both cases, and a similar reduction of the training time is observed in Figure 5.5. The convergence by epoch remains similar to non-multilevel approaches, but the overall time is reduced in the multilevel approach.

Model	Training Time (s)	Testing Loss	Inverse Accuracy
OD	23.8K	10.64	1.85e-7
OD Multilevel	20.2K	10.43	1.22e-7
DO	2.73K	10.45	1.45e-3
DO Multilevel	2.38K	10.38	1.32e-3

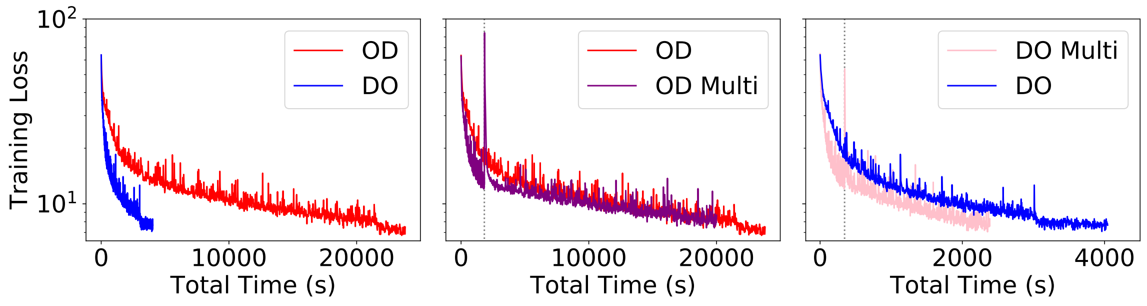


Figure 5.5: Training CNFs on MINIBOONE with and without multilevel. For multilevel, train first 500 epochs with a quicker coarse grid, then switch to a fine grid for 1500 epochs. The switch to the finer grid results in immediate uptick in loss, which quickly recovers. The convergence by epoch remains similar to non-multilevel approaches, but the overall time is reduced in the multilevel approach.

5.4 OT-Flow

Whereas the improvements in numerical treatment in Section 5.3 result in notable reductions in training costs, CNFs still have high costs. We alleviated some cost by moderately reducing the NFE for solving the CNF (5.7); however, the trace is estimated with Hutchinson's (5.15) which we further investigate. Our model OT-Flow further reduces NFE while also providing an efficient exact trace computation to replace the Hutchinson's estimator while maintaining competitive speed.

In its original presentation [77], OT-Flow draws from OT theory. Rather than using the Benamou-Brenier formulation [9] (true OT formulation with hard constraints), OT-Flow stems from the "relaxed Benamou-Brenier formulation." This relaxed formulation turns the hard constraints into a soft terminal constraint as part of the loss—like G in (2.10). Similarly, transport costs equate the OC running costs, and therefore the OT-Flow formulation is an OC problem. Leveraging OC and the value function, we formulate OT-Flow and demonstrate its fast training and inference.

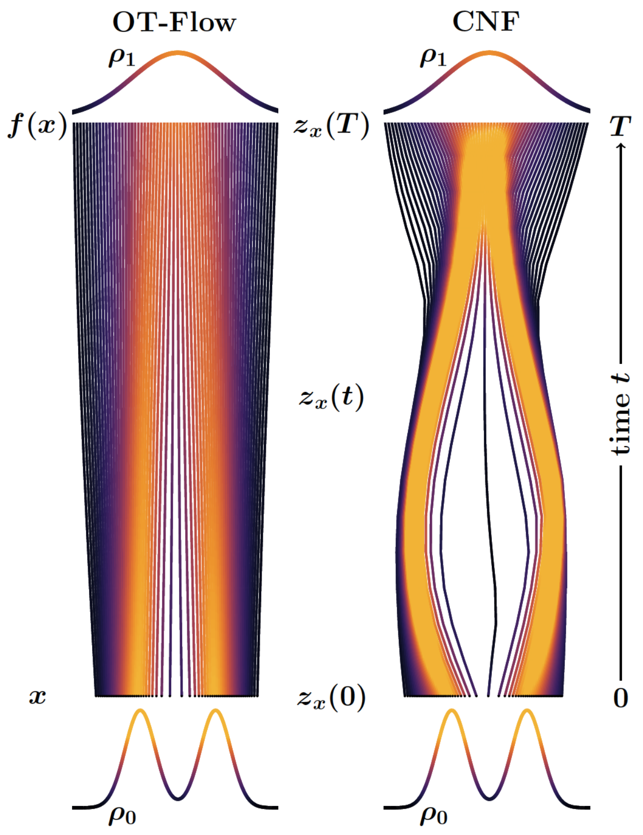


Figure 5.6: Two flows with approximately equal loss (modification of Figure 1 in Grathwohl et al. [39], Finlay et al. [32]). While OT-Flow enforces straight trajectories, a generic CNF can have curved trajectories.

5.4.1 Model Formulation

Motivated by the similarities between training CNFs and solving OT problems [9, 82], we regularize the minimization problem (5.13) as follows. First, we formulate the CNF problem as an OT problem by adding a transport cost. Second, from OT theory, we leverage the fact that the optimal dynamics $\mathbf{v}$ are the negative gradient of a value function Φ, which satisfies the HJB equations. Finally, we add an extra term to the learning problem that penalizes violations of the HJB equations. This reformulation encourages straight trajectories (Figure 5.6).

Running Cost Consider the CNF problem (5.13). We add the running cost

$$L_{\boldsymbol{x}}(t) = \frac{1}{2} \|\mathbf{v}\big(\boldsymbol{z}_{\boldsymbol{x}}(t), t\big)\|^2, \tag{5.19}$$

which we accumulate along the trajectories via

$$c_{\mathrm{L},\boldsymbol{x}}(T) = \int_0^T L_{\boldsymbol{x}}(t)\,\mathrm{d}t.$$

This incorporation results in the regularized problem

$$\min_{\boldsymbol{\theta}} \quad \mathop{\mathbb{E}}_{\boldsymbol{x} \sim \mathrm{P}_0(\boldsymbol{x})} \Big\{ C_{\boldsymbol{x}}(T) + c_{\mathrm{L},\boldsymbol{x}}(T) \Big\} \quad \text{s.t.} \quad (5.7). \tag{5.20}$$

The running cost L penalizes the squared arc-length of the trajectories. In practice, this integral can be computed in the ODE solver, similar to the trace accumulation in (5.7). The optimization problem (5.20) is an OC problem (2.10) with terminal cost C. Recall that C in (5.12) is not the full KL divergence because density ρ_0 is unknown. This formulation has mathematical properties that we exploit to reduce CNF computational costs [30, 32, 110]. In particular, (5.20) is now equivalent to a *convex* optimization problem (prior to the NN parameterization), and the trajectories matching the two densities ρ_0 and ρ_1 are straight and non-intersecting [34] (Figure 5.6). This reduces the number of time steps required to solve (5.7). The OC formulation also guarantees a solution flow that is smooth, invertible, and orientation-preserving [1].

Value Function and HJB Penalizer We further capitalize on OC theory by incorporating additional structure to guide our modeling. Recall the existence of the value function (also called the potential function) $\Phi\colon \mathbb{R}^d \times [0,T] \to \mathbb{R}$. Using (2.12) and (2.14), we can construct the Hamiltonian of the CNF

$$H(\boldsymbol{x}, \nabla\Phi(\boldsymbol{x},t), t) = \sup_{\mathbf{v}} \left\{ -\nabla\Phi(\boldsymbol{x},t) \cdot \mathbf{v}(\boldsymbol{x},t) - \frac{1}{2}\|\mathbf{v}(\boldsymbol{x},t)\|^2 \right\} \tag{5.21}$$

where the control is the velocity of the samples $\mathbf{v}$. The first-order necessary conditions

$$0 = -\nabla\Phi(\boldsymbol{x},t) - \mathbf{v}(\boldsymbol{x},t). \tag{5.22}$$

lead to

$$\mathbf{v}(\boldsymbol{x},t;\boldsymbol{\theta}) = -\nabla\Phi(\boldsymbol{x},t;\boldsymbol{\theta}), \tag{5.23}$$

which means we can parameterize Φ with an NN instead of $\mathbf{v}$ directly. Analogous to classical physics, samples move in a manner to minimize their potential function. Moreover, Φ satisfies the HJB equations (2.16) [7]; for our problem, these become

$$\begin{aligned} -\partial_t\Phi(\boldsymbol{x},t) &= -\frac{1}{2}\|\nabla\Phi(\boldsymbol{x},t)\|^2, \\ \Phi(\boldsymbol{x},T) &= G(\boldsymbol{x}), \end{aligned} \tag{5.24}$$

with terminal condition G. To derive an equation for G, consider the KL divergence in (5.9) after the change of variables is performed. The HJB terminal condition is given by [8, 110]

$$\begin{aligned} G(\boldsymbol{z}_{\boldsymbol{x}}(T)) &\coloneqq \frac{\delta}{\delta\rho_0}\mathbb{D}_{\mathrm{KL}}\left[\,\rho\big(\boldsymbol{z}_{\boldsymbol{x}}(T),T\big)\,||\,\rho_1\big(\boldsymbol{z}_{\boldsymbol{x}}(T)\big)\,\right] \\ &= \frac{\delta}{\delta\rho_0}\int_{\mathbb{R}^d}\Big[\log\big(\rho_0(\boldsymbol{x})\big) - \log\big(\rho_1(\boldsymbol{z}_{\boldsymbol{x}}(T))\big) - \log\det\big(\nabla\boldsymbol{z}_{\boldsymbol{x}}(T)\big)\Big]\rho_0(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} \\ &= 1 + \log\big(\rho_0(\boldsymbol{x})\big) - \log\big(\rho_1(\boldsymbol{z}_{\boldsymbol{x}}(T))\big) - \log\det\big(\nabla\boldsymbol{z}_{\boldsymbol{x}}(T)\big), \end{aligned} \tag{5.25}$$

where $\frac{\delta}{\delta\rho_0}$ is the variational derivative with respect to ρ_0.

The value function allows us to reformulate the CNF in terms of Φ instead of $\mathbf{v}$

and add an additional regularization term which penalizes the violations of (5.24) along the trajectories by

$$c_{\mathrm{HJt},\boldsymbol{x}}(T) = \int_0^T R_{\boldsymbol{x}}(t)\,\mathrm{d}t, \quad R_{\boldsymbol{x}}(t) = \left| \partial_t \Phi\big(\boldsymbol{z}_{\boldsymbol{x}}(t), t\big) - \frac{1}{2}\|\nabla \Phi\big(\boldsymbol{z}_{\boldsymbol{x}}(t), t\big)\|^2 \right|. \qquad (5.26)$$

This HJB penalizer R favors plausible Φ without affecting the solution of the optimization problem (5.20). While solving (5.24) in high-dimensional spaces is notoriously difficult, penalizing its violations along the trajectories is inexpensive. Therefore, we include the value $R_{\boldsymbol{x}}(T)$ in the objective function. The density ρ_0, which is required to evaluate G, is unknown in our problems. Similar to Yang and Karniadakis [118], we do not enforce the HJB terminal condition but do enforce the HJB equations for $t \in (0, T)$ via penalizer R.

Typically, regularizers (such as weight decay) introduce bias to the problem while reducing variance. Since the R term does not mathematically alter the solution, we call it a *penalizer* instead. As we will show, penalizers help reduce costs or find the solution. As a thought experiment, consider the traversal of the noisy high-dimensional space in which $\boldsymbol{\theta}$ lives. Imagine penalizers as smoothing this space to help facilitate the traversal without actually altering the location of the optimum.

Effect of HJB Penalizer In Figure 5.7, we show the effect of training the toy Gaussian mixture problem with and without the HJB penalizer R. For this demonstration, we train the model using two RK4 steps. As a result, the L cost is penalized at too few time steps. Therefore, without an HJB penalizer, the model achieves poor performance and unstraight characteristics (Figure 5.7). This issue can be remedied by adding more RK4 time steps or the HJB penalizer—see examples in Yang and Karniadakis [118], Ruthotto et al. [93], and Lin et al. [64]. The additional RK4 time steps would add significant memory and computational overhead. The HJB penalizer, however, adds little memory and computation. We thus can train the model

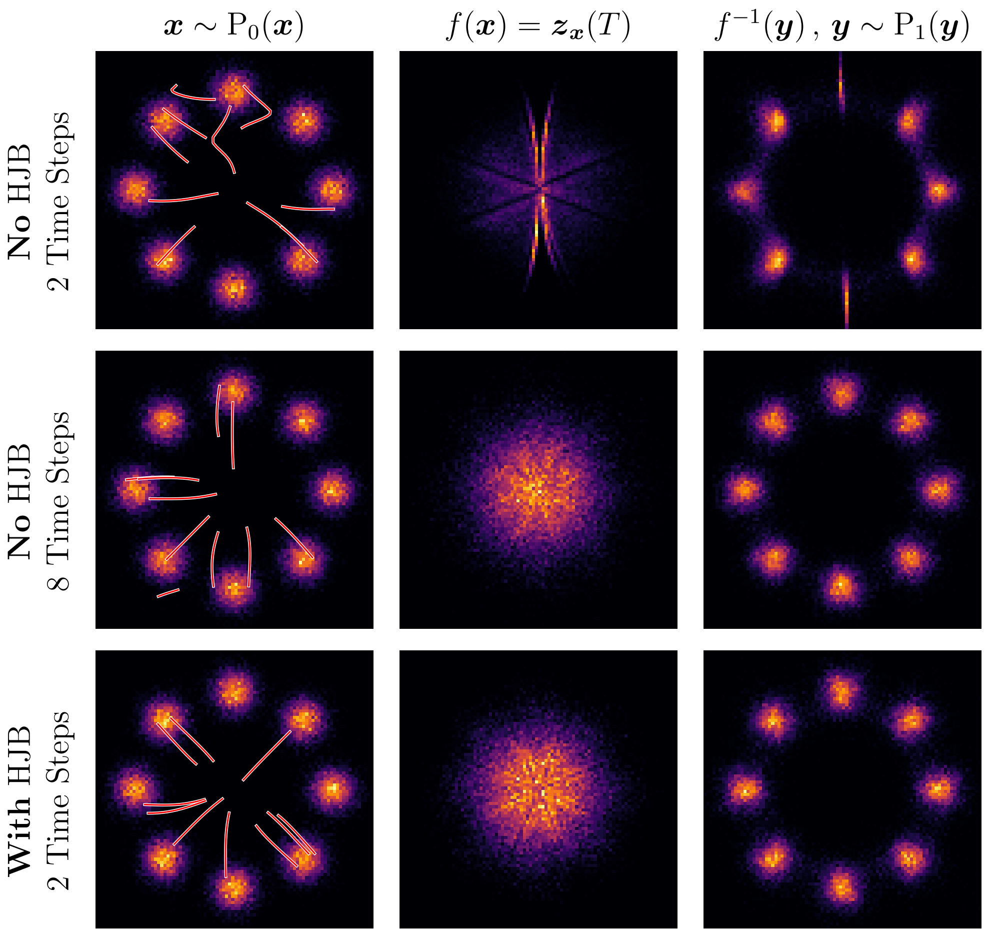


Figure 5.7: Effect of adding an HJB regularizer during training. For each flow, we show initial, forward mapping, and generation. The HJB regularizer allows for training a flow with one-fourth the number of time steps, leading to a drastic reduction in computational and memory costs. White trajectories display the forward flow f for several random samples; red trajectories display the inverse flow f^{-1}.

with two RK4 time steps and an HJB penalizer with efficient computational cost *and* good performance.

OT-Flow Problem In summary, the regularized problem solved in OT-Flow combines aspects from Zhang et al. [119], Grathwohl et al. [39], Yang and Karniadakis [118], and Finlay et al. [32] (Table 5.1). The L_2 and HJB terms add regularity and are accumulated along the trajectories. Thus, the full optimization problem is

$$\min_{\boldsymbol{\theta}} \quad \mathop{\mathbb{E}}_{\boldsymbol{x}\sim \mathrm{P}_0(\boldsymbol{x})} \left\{ \alpha_1 C_{\boldsymbol{x}}(T) + c_{\mathrm{L},\boldsymbol{x}}(T) + \alpha_2\, c_{\mathrm{HJt},\boldsymbol{x}}(T) \right\}, \tag{5.27}$$

subject to

$$\partial_t \begin{pmatrix} \boldsymbol{z}_{\boldsymbol{x}}(t) \\ \ell_{\boldsymbol{x}}(t) \\ c_{\mathrm{L},\boldsymbol{x}}(t) \\ c_{\mathrm{HJt},\boldsymbol{x}}(t) \end{pmatrix} = \begin{pmatrix} -\nabla\Phi(\boldsymbol{z}_{\boldsymbol{x}}(t),t;\boldsymbol{\theta}) \\ -\operatorname{tr}\left(\nabla^2\Phi(\boldsymbol{z}_{\boldsymbol{x}}(t),t;\boldsymbol{\theta})\right) \\ \frac{1}{2}\|\nabla\Phi(\boldsymbol{z}_{\boldsymbol{x}}(t),t;\boldsymbol{\theta})\|^2 \\ \left|\, \partial_t\Phi(\boldsymbol{z}_{\boldsymbol{x}}(t),t;\boldsymbol{\theta}) - \frac{1}{2}\|\nabla\Phi(\boldsymbol{z}_{\boldsymbol{x}}(t),t;\boldsymbol{\theta})\|^2 \,\right| \end{pmatrix}, \quad \begin{pmatrix} \boldsymbol{z}_{\boldsymbol{x}}(0) \\ \ell_{\boldsymbol{x}}(0) \\ c_{\mathrm{L},\boldsymbol{x}}(0) \\ c_{\mathrm{HJt},\boldsymbol{x}}(0) \end{pmatrix} = \begin{pmatrix} \boldsymbol{x} \\ 0 \\ 0 \\ 0 \end{pmatrix},$$

where we optimize the weights $\boldsymbol{\theta}$, defined in (5.28), that parameterize Φ. We include two hyperparameters α_1, α_2 to assist the optimization; these hyperparameters weight the importance of the corresponding terms relative to the Lagrangian term. Specially selected hyperparameters can improve the convergence and performance of the model.[1] By making use of the ODE solver and the computed $\nabla\Phi$, we compute the terms L, R with negligible cost.

5.4.2 Implementation

We define our model, derive analytic formulas for fast and exact trace computation, and describe our efficient ODE solver.

Network We parameterize the potential as

$$\begin{gathered} \Phi(\boldsymbol{s};\boldsymbol{\theta}) = \boldsymbol{w}^\top N(\boldsymbol{s};\boldsymbol{\theta}_N) + \frac{1}{2}\boldsymbol{s}^\top(\boldsymbol{A}^\top\boldsymbol{A})\boldsymbol{s} + \boldsymbol{b}^\top\boldsymbol{s} + c, \\ \text{where} \quad \boldsymbol{\theta} = (\boldsymbol{w}, \boldsymbol{\theta}_N, \boldsymbol{A}, \boldsymbol{b}, c). \end{gathered} \tag{5.28}$$

[1]All hyperparameters are included in the public code repository available at `https://github.com/EmoryMLIP/OT-Flow`.

Here, $\boldsymbol{s} = (\boldsymbol{x}, t) \in \mathbb{R}^{d+1}$ are the input features corresponding to space-time, $N(\boldsymbol{s}; \boldsymbol{\theta}_N) \colon \mathbb{R}^{d+1} \to \mathbb{R}^m$ is a neural network chosen to be a residual neural network (ResNet) [45] in our experiments, and $\boldsymbol{\theta}$ consists of all the trainable weights: $\boldsymbol{w} \in \mathbb{R}^m$, $\boldsymbol{\theta}_N \in \mathbb{R}^{p_N}$, $\boldsymbol{A} \in \mathbb{R}^{\iota \times (d+1)}$, $\boldsymbol{b} \in \mathbb{R}^{d+1}$, $c \in \mathbb{R}$. We set a rank $\iota = \min(10, d)$ to limit the number of parameters of the symmetric matrix $\boldsymbol{A}^\top \boldsymbol{A}$. Here, $\boldsymbol{A}$, $\boldsymbol{b}$, and c model quadratic potentials, i.e., linear dynamics; N models the nonlinear dynamics. This formulation was found to be effective in Ruthotto et al. [93].

ResNet The $(M+1)$-layer ResNet uses an opening layer to convert the $\mathbb{R}^{d+1}$ inputs to the $\mathbb{R}^m$ space, then M layers operating on the features in hidden space $\mathbb{R}^m$. We obtain $N(\boldsymbol{s}; \boldsymbol{\theta}_N)$ by forward propagation

$$\begin{aligned} \boldsymbol{a}_0 &= \sigma(\boldsymbol{K}_0 \boldsymbol{s} + \boldsymbol{b}_0) \\ \boldsymbol{a}_1 &= \boldsymbol{a}_0 + h\, \sigma(\boldsymbol{K}_1 \boldsymbol{a}_0 + \boldsymbol{b}_1) \\ \vdots \quad & \quad\quad \vdots \\ N(\boldsymbol{s}; \boldsymbol{\theta}_N) = \boldsymbol{a}_M &= \boldsymbol{a}_{M-1} + h\, \sigma(\boldsymbol{K}_M \boldsymbol{a}_{M-1} + \boldsymbol{b}_M), \end{aligned} \tag{5.29}$$

where $h > 0$ is a fixed stepsize, and the network's weights are $\boldsymbol{K}_0 \in \mathbb{R}^{m \times (d+1)}$, $\boldsymbol{K}_1, \ldots, \boldsymbol{K}_M \in \mathbb{R}^{m \times m}$, and $\boldsymbol{b}_0, \ldots, \boldsymbol{b}_M \in \mathbb{R}^m$. We select the element-wise activation function $\sigma(\boldsymbol{x}) = \log(\exp(\boldsymbol{x}) + \exp(-\boldsymbol{x}))$, which is the antiderivative of the hyperbolic tangent, i.e., $\sigma'(\boldsymbol{x}) = \tanh(\boldsymbol{x})$. Therefore, hyperbolic tangent is the activation function of the flow $\nabla \Phi$.

Gradient Computation The gradient of the potential is

$$\nabla_{\boldsymbol{s}} \Phi(\boldsymbol{s}; \boldsymbol{\theta}) = \nabla_{\boldsymbol{s}} N(\boldsymbol{s}; \boldsymbol{\theta}_N) \boldsymbol{w} + (\boldsymbol{A}^\top \boldsymbol{A}) \boldsymbol{s} + \boldsymbol{b}, \tag{5.30}$$

where we simply take the first d components of $\nabla_{\boldsymbol{s}}\Phi$ to obtain the space derivative $\nabla\Phi$. The first term is computed using chain rule (backpropagation)

$$\begin{aligned} \boldsymbol{v}_{M+1} &= \boldsymbol{w} \\ \boldsymbol{v}_M &= \boldsymbol{v}_{M+1} + h\,\boldsymbol{K}_M^\top \operatorname{diag}\big(\sigma'(\boldsymbol{K}_M\boldsymbol{a}_{M-1} + \boldsymbol{b}_M)\big)\boldsymbol{v}_{M+1}, \\ &\vdots \qquad \vdots \\ \boldsymbol{v}_1 &= \boldsymbol{v}_2 + h\,\boldsymbol{K}_1^\top \operatorname{diag}\big(\sigma'(\boldsymbol{K}_1\boldsymbol{a}_0 + \boldsymbol{b}_1)\big)\boldsymbol{v}_2, \\ \nabla_{\boldsymbol{s}} N(\boldsymbol{s};\boldsymbol{\theta}_N)\boldsymbol{w} = \boldsymbol{v}_0 &= \boldsymbol{K}_0^\top \operatorname{diag}\big(\sigma'(\boldsymbol{K}_0\boldsymbol{s} + \boldsymbol{b}_0)\big)\boldsymbol{v}_1. \end{aligned} \tag{5.31}$$

Here, $\operatorname{diag}(\boldsymbol{q}) \in \mathbb{R}^{m\times m}$ denotes a diagonal matrix with diagonal elements given by $\boldsymbol{q} \in \mathbb{R}^m$. Multiplication by diagonal matrix is implemented as an element-wise product.

Trace Computation We compute the trace of the Hessian of the potential model. We first note that

$$\operatorname{tr}\big(\nabla_{\boldsymbol{s}}^2\Phi(\boldsymbol{s};\boldsymbol{\theta})\big) = \operatorname{tr}\Big(\boldsymbol{E}^\top\,\nabla_{\boldsymbol{s}}^2\big(N(\boldsymbol{s};\boldsymbol{\theta}_N)\boldsymbol{w}\big)\,\boldsymbol{E}\Big) + \operatorname{tr}\Big(\boldsymbol{E}^\top(\boldsymbol{A}^\top\boldsymbol{A})\,\boldsymbol{E}\Big), \tag{5.32}$$

where the columns of $\boldsymbol{E} \in \mathbb{R}^{(d+1)\times d}$ are the first d standard basis vectors in $\mathbb{R}^{d+1}$. All matrix multiplications with $\boldsymbol{E}$ can be implemented as constant-time indexing operations. The trace of the $\boldsymbol{A}^\top\boldsymbol{A}$ term is trivial. We compute the trace in one forward pass through the layers. The trace of the first ResNet layer is

$$\begin{aligned} t_0 &= \operatorname{tr}\Big(\boldsymbol{E}^\top\,\nabla_{\boldsymbol{s}}\big(\boldsymbol{K}_0^\top \operatorname{diag}(\sigma'(\boldsymbol{K}_0\boldsymbol{s} + \boldsymbol{b}_0))\boldsymbol{v}_1\big)\,\boldsymbol{E}\Big) \\ &= \operatorname{tr}\big(\boldsymbol{E}^\top\boldsymbol{K}_0^\top \operatorname{diag}\big(\sigma''(\boldsymbol{K}_0\boldsymbol{s} + \boldsymbol{b}_0) \odot \boldsymbol{v}_1\big)\,\boldsymbol{K}_0\boldsymbol{E}\big) \\ &= \big(\sigma''(\boldsymbol{K}_0\boldsymbol{s} + \boldsymbol{b}_0) \odot \boldsymbol{v}_1\big)^\top\big((\boldsymbol{K}_0\boldsymbol{E}) \odot (\boldsymbol{K}_0\boldsymbol{E})\big)\mathbf{1}, \end{aligned} \tag{5.33}$$

where $\mathbf{1} \in \mathbb{R}^d$ is a vector of all ones. Computing t_0 requires $\mathcal{O}(m \cdot d)$ FLOPS when first squaring the elements in the first d columns of $\boldsymbol{K}_0$, then summing those columns, and finally one inner product. To compute the trace of the entire ResNet, we continue with the remaining rows in (5.31) to obtain

$$\operatorname{tr}\big(\boldsymbol{E}^\top \nabla_{\boldsymbol{s}}^2 (N(\boldsymbol{s};\boldsymbol{\theta}_N)\boldsymbol{w})\,\boldsymbol{E}\big) = t_0 + h\sum_{i=1}^{M} t_i, \tag{5.34}$$

where t_i is computed as

$$\begin{aligned} t_i &= \operatorname{tr}\big(\boldsymbol{J}_{i-1}^\top \nabla_{\boldsymbol{s}}\big(\boldsymbol{K}_i^\top \operatorname{diag}(\sigma'(\boldsymbol{K}_i\boldsymbol{a}_{i-1}(\boldsymbol{s})+\boldsymbol{b}_i))\boldsymbol{v}_{i+1}\big)\boldsymbol{J}_{i-1}\big) \\ &= \operatorname{tr}\big(\boldsymbol{J}_{i-1}^\top \boldsymbol{K}_i^\top \operatorname{diag}\big(\sigma''(\boldsymbol{K}_i\boldsymbol{a}_{i-1}+\boldsymbol{b}_i)\odot \boldsymbol{v}_{i+1}\big)\,\boldsymbol{K}_i\boldsymbol{J}_{i-1}\big) \\ &= \big(\sigma''(\boldsymbol{K}_i\boldsymbol{a}_{i-1}+\boldsymbol{b}_i)\odot\boldsymbol{v}_{i+1}\big)^\top \big((\boldsymbol{K}_i\boldsymbol{J}_{i-1})\odot(\boldsymbol{K}_i\boldsymbol{J}_{i-1})\big)\mathbf{1}. \end{aligned}$$

Here, $\boldsymbol{J}_{i-1} = \nabla\boldsymbol{a}_{i-1}^\top \in \mathbb{R}^{m\times d}$ is a Jacobian matrix, which can be updated and over-written in the forward pass at a computational cost of $\mathcal{O}(m^2\cdot d)$ FLOPS. The $\boldsymbol{J}$ update is initialized with $\boldsymbol{J} = \nabla\boldsymbol{a}_0^\top = \operatorname{diag}\big(\sigma'(\boldsymbol{K}_0\boldsymbol{s}+\boldsymbol{b}_0)\big)(\boldsymbol{K}_0\boldsymbol{E})$ and follows:

$$\begin{aligned} \nabla\boldsymbol{a}_i^\top &= \nabla\boldsymbol{a}_{i-1}^\top + \operatorname{diag}\big(h\,\sigma'(\boldsymbol{K}_i\boldsymbol{a}_{i-1}+\boldsymbol{b}_i)\big)\boldsymbol{K}_i\nabla\boldsymbol{a}_{i-1}^\top \\ \boldsymbol{J} &\leftarrow \boldsymbol{J} + \operatorname{diag}\big(h\,\sigma'(\boldsymbol{K}_i\boldsymbol{a}_{i-1}+\boldsymbol{b}_i)\big)\,\boldsymbol{K}_i\boldsymbol{J}. \end{aligned} \tag{5.35}$$

Our experiments use a simple two-layer ResNet (M=2). When tuning the number of layers as a hyperparameter, we found that wide networks promoted expressibility but deep networks offered no noticeable improvement. We choose h=1 for simplicity.

The total computational cost of our model is $\mathcal{O}(m^2\cdot d)$ FLOPS. Thus, our exact trace computation has similar computational complexity as FFJORD's and RNODE's trace estimation. In clocktime, the analytic exact trace computation is competitive with the Hutchinson's estimator using AD, while introducing no estimation error (Figure 5.2). We time our approach and compare against the Hutchinson's estimator using AD with different numbers of vectors and present 99% error bounds via boot-strapping [27] (Appendix B). Our efficiency in trace computation (5.34) stems from

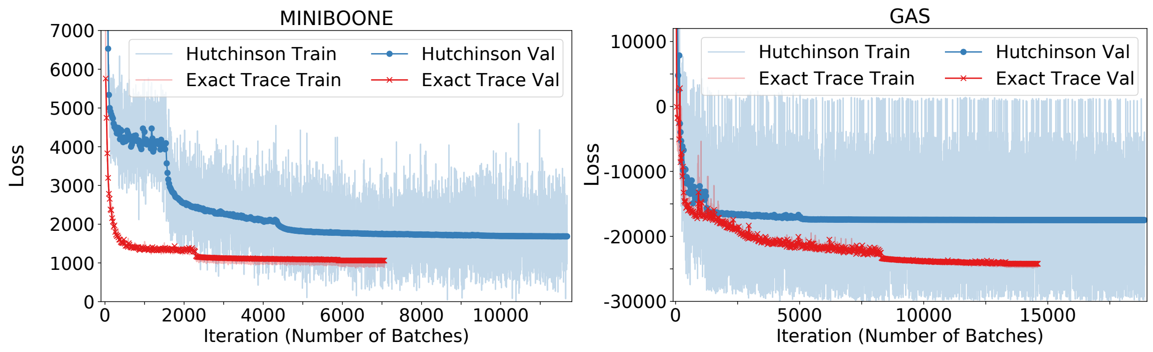


Figure 5.8: We compare the training and validation losses for OT-Flow (exact trace) and an identical model where we instead use the Hutchinson's estimator using a single vector. The exact trace allows OT-Flow to converge in fewer iterations to a lower validation loss with smaller variance in training loss than the Hutchinson's estimator.

exploiting the identity structure of matrix $\boldsymbol{E}$ and not building the full Hessian.

Since we parameterize the value function Φ instead of the $\mathbf{v}$, the Jacobian of the dynamics $\nabla\mathbf{v}$ is given by the Hessian of Φ in (5.7). We note that Hessians are *symmetric* matrices. When trace estimates seem appealing, we note that a plethora of estimators perform better in accuracy and speed on symmetric matrices than on nonsymmetric matrices [6, 49, 109]. We, however, stick with the exact trace for our use case.

We find that using the exact trace instead of a trace estimator improves convergence (Figure 5.8). Specifically, we train an OT-Flow model and a replicate model in which we only change the trace computation, i.e., we replace the exact trace computation with Hutchinson's estimator using a single random vector. The model using the exact trace (OT-Flow) converges more quickly and to a lower validation loss, while its training loss has less variance (Figure 5.8).

Using Hutchinson's estimator without sufficiently many time steps fails to converge [76] because such an approach poorly approximates the time integration *and* the trace in the second component of (5.7). Whereas FFJORD and RNODE estimate the trace but solve the time integral well, OT-Flow trains with the exact trace and notably fewer time steps (Table 5.8). At inference, all three solve the trace and integration well.

ODE Solver For the forward propagation, we use RK4 with equidistant time steps to solve the constraints of (5.27). The number of time steps is a hyperparameter denoted n_t. For validation and testing, we use more time steps than for training, which allows for higher precision and a check that our discrete OT-Flow still approximates the continuous object. A large number of training time steps prevents overfitting to a particular discretization of the continuous solution and lowers inverse error; too few time steps results in high inverse error but low computational cost. We tune the number of training time steps so that validation and training loss are similar with low computational cost. Other hyperparameters include the hidden space size m, the number of ResNet layers for which we use 2 for all experiments, and various settings for the ADAM optimizer.

For the backpropagation, we use AD. This technique corresponds to the DO approach. Our implementation exploits the benefits of our proposed exact trace computation combined with the efficiency of DO.

5.4.3 Numerical Experiments

We perform density estimation on seven two-dimensional toy problems and five high-dimensional problems from real data sets. We also show OT-Flow's generative abilities on MNIST.

Metrics In density estimation, the goal is to approximate ρ_0 using observed samples $\boldsymbol{X} = \{\boldsymbol{x}_i\}$, where $\boldsymbol{x}_i$ are drawn from the distribution P_0. In real applications, we lack a ground-truth ρ_0, rendering proper evaluation of the density itself untenable. However, we can follow evaluation techniques applied to generative models. Drawing random points $\{\boldsymbol{y}_i\}$ from P_1, we invert the flow to generate synthetic samples $\boldsymbol{Q} = \{\boldsymbol{q}_i\}$, where $\boldsymbol{q}_i = f^{-1}(\boldsymbol{y}_i)$. We compare the known samples to the generated samples via

Discrete normalizing flow trained on Power

d	**Testing Loss**	Inv Error	MMD
6	**−0.64**	2.34e-3	1.94e-2

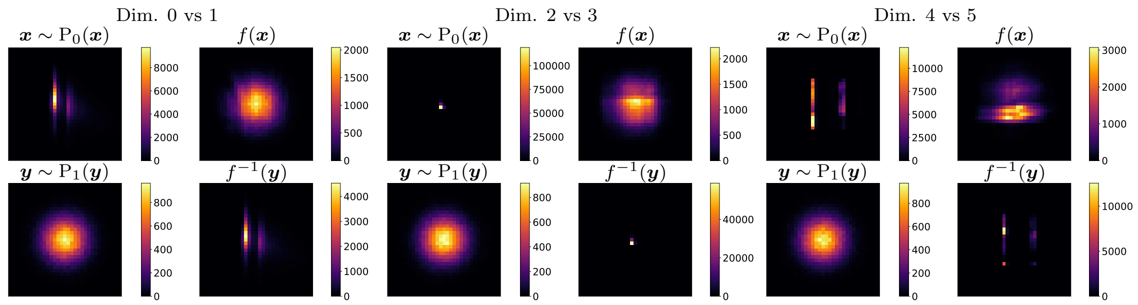


Figure 5.9: Power density estimation using a discrete normalizing flow. By the testing loss metric, this model is considered very competitive. However, the model itself performs poorly, as clear in the visualization of the last two dimensions. The MMD shows that the generation is poor. The inverse error shows that the testing loss uses an integration scheme that is too coarse, as addressed in Section 5.3.1 and Wehenkel and Louppe [113].

maximum mean discrepancy (MMD) [40, 60, 82, 108]

$$\begin{aligned}\mathrm{MMD}(\boldsymbol{X},\boldsymbol{Q}) &= \frac{1}{|\boldsymbol{X}|^2}\sum_{i=1}^{|\boldsymbol{X}|}\sum_{j=1}^{|\boldsymbol{X}|} k(\boldsymbol{x}_i,\boldsymbol{x}_j) \\ +\frac{1}{|\boldsymbol{Q}|^2}\sum_{i=1}^{|\boldsymbol{Q}|}\sum_{j=1}^{|\boldsymbol{Q}|} k(\boldsymbol{q}_i,\boldsymbol{q}_j) &- \frac{2}{|\boldsymbol{X}|\,|\boldsymbol{Q}|}\sum_{i=1}^{|\boldsymbol{X}|}\sum_{j=1}^{|\boldsymbol{Q}|} k(\boldsymbol{x}_i,\boldsymbol{q}_j),\end{aligned} \tag{5.36}$$

for Gaussian kernel $k(\boldsymbol{x}_i,\boldsymbol{q}_j)=\exp(-\frac{1}{2}\|\boldsymbol{x}_i-\boldsymbol{q}_j\|^2)$. MMD tests the difference between two densities (ρ_0 and our estimate of ρ_0) on the basis of samples drawn from the corresponding distributions. A low MMD value means that the two sets of samples are likely to have been drawn from the same distribution [40]. Since MMD is not used in the training, it provides an external, impartial metric to evaluate our model on the hold-out test set (Table 5.8).

Normalizing flows use C for evaluation. Recall that loss C is used to train the forward flow to match ρ_1. Testing loss, i.e., C evaluated on the testing set, should provide the same quantification on a hold-out set. However, in some cases, the testing

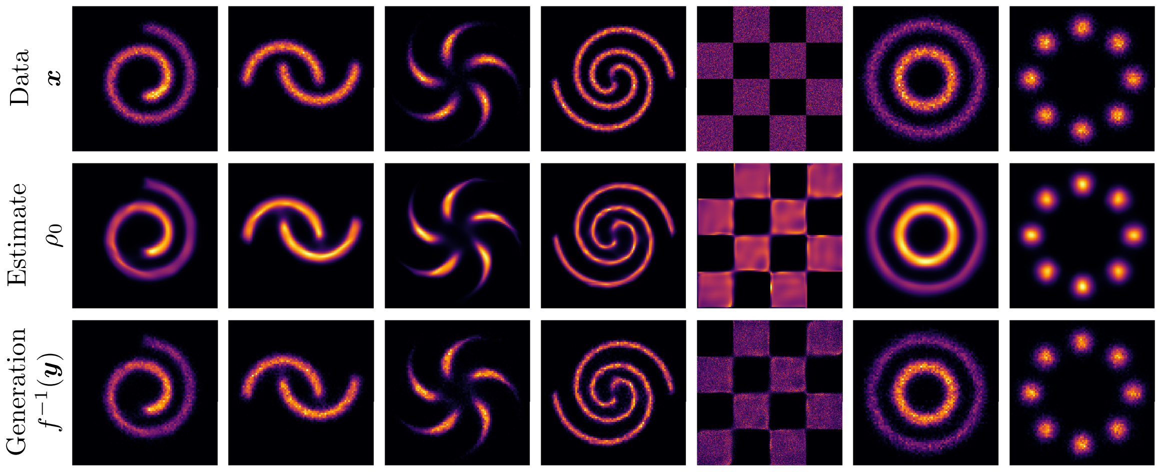
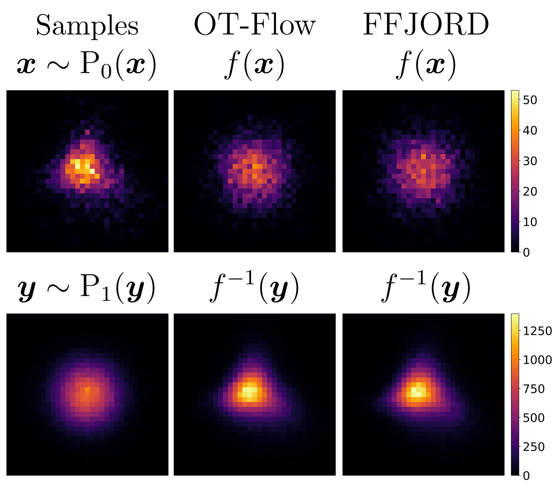


Figure 5.10: OT-Flow density estimation on 2-D toy problems. **Top:** samples from the unknown distribution. **Middle:** density estimate for unknown ρ_0 computed by inverse flowing from ρ_1 via (5.7).

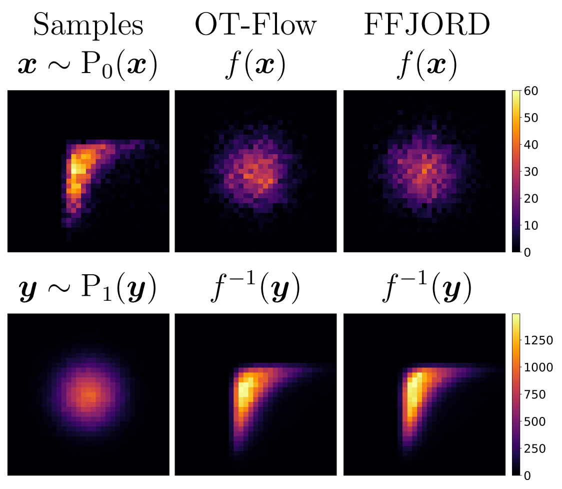
loss can be low even when $f(\boldsymbol{x})$ is poor and does not match ρ_1 (Figure 5.9). Furthermore, because the model's inverse contains error, accurately mapping to ρ_1 with the forward flow does not necessarily mean the inverse flow accurately maps to ρ_0.

Testing loss varies drastically with the integration computation [76, 108, 113]. It depends on ℓ, which is computed along the characteristics via time integration of the trace. Too few discretization points leads to an inaccurate integration computation and greater inverse error. Thus, a low inverse error implies an accurate integration computation because the flow closely models the ODE. An adaptive ODE solver alleviates this concern when provided a sufficiently small tolerance [39]. Similarly, we check that the flow models the continuous solution of the ODE by computing the inverse error (5.16) on the testing set using a finer time discretization than used in training. We evaluate the expectation values in (5.27) and (5.16) using the discrete samples $\boldsymbol{X}$, which we assume are randomly drawn from and representative of the initial distribution ρ_0.

Toy Problems We train OT-Flow on several toy distributions that serve as standard benchmarks [39, 113]. Given random samples, we train OT-Flow then use it to



(a) Miniboone dimension 16 vs 17



(b) Miniboone dimension 28 vs 29

Figure 5.11: Miniboone density estimation. Two-dimensional slices using the 3,648 43-dimensional testing samples $\boldsymbol{x} \sim \mathrm{P}_0(\boldsymbol{x})$ and 10^5 samples $\boldsymbol{y}$ from distribution P_1; more visuals presented the Appendix of Onken et al. [77].

estimate the density ρ_0 and generate samples (Figure 5.10).

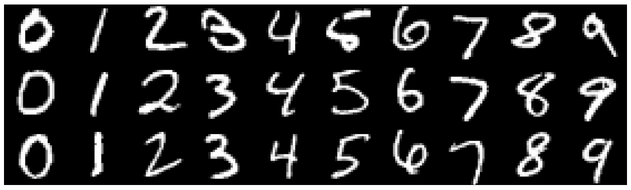
Density Estimation on Real Data Sets We compare our model's performance on the real data sets Power, Gas, Hepmass, Miniboone, and Bsds300 [80]. The data sets are commonly used in normalizing flows [23, 39, 48, 113] and vary in dimensionality (Table 5.4).

For each data set, we compare OT-Flow with FFJORD [39] and RNODE [32] (current state-of-the-art) in speed and performance. We compare speed both in training the models and when running the model on the testing set. To compare performance, we compute the MMD between the data set and $M{=}10^5$ generated samples $f^{-1}(\boldsymbol{y})$ for each model; for a fair comparison, we use the same $\boldsymbol{y}$ for FFJORD and OT-Flow (Table 5.8). We show visuals of the samples $\boldsymbol{x} \sim \mathrm{P}_0(\boldsymbol{x})$, $\boldsymbol{y} \sim \mathrm{P}_1(\boldsymbol{y})$, $f(\boldsymbol{x})$, and $f^{-1}(\boldsymbol{y})$ generated by OT-Flow and FFJORD (Figure 5.11). These two-dimensional slices use the 3,648 43-dimensional testing samples $\boldsymbol{x} \sim \mathrm{P}_0(\boldsymbol{x})$ and 10^5 samples $\boldsymbol{y}$ from distribution P_1. We report the loss C values (Table 5.8) to be comparable to other literature but reiterate the inherent flaws in using C to compare models.

Table 5.8: OT-Flow density estimation on real data sets. We present the number of training iterations, the number of function evaluations for the forward ODE solve (NFE), and the time per iteration. For Bsds300 training, FFJORD and RNODE were terminated when validation loss C hit -140. All values are the average across three runs on a single NVIDIA TITAN X GPU with 12GB RAM.

	Model	Training				Testing		
		Time (h)	# Iter	$\frac{\text{Time}}{\text{Iter}}$(s)	NFE	Time (s)	Inv Err	MMD
Power	OT-Flow	3.1	22K	0.56	40	10.6	4.10e-6	4.68e-5
	RNODE	25.0	32K	2.78	200	88.2	5.95e-6	5.64e-5
	FFJORD	68.9	29K	8.63	583	72.4	7.60e-6	4.34e-5
Gas	OT-Flow	6.1	52K	0.42	40	30.9	1.79e-4	2.47e-4
	RNODE	36.3	59K	2.23	200	763.7	2.53e-5	8.03e-5
	FFJORD	75.4	49K	5.54	475	892.4	1.78e-5	1.02e-4
Hepmass	OT-Flow	5.2	35K	0.53	48	47.9	2.98e-6	1.58e-5
	RNODE	46.5	40K	4.16	400	446.7	1.91e-5	1.58e-5
	FFJORD	99.4	47K	7.56	770	450.4	2.98e-5	1.58e-5
Miniboone	OT-Flow	0.8	7K	0.44	24	0.8	5.65e-6	2.84e-4
	RNODE	1.4	15K	0.33	16	33.0	4.42e-6	2.84e-4
	FFJORD	9.0	16K	2.01	115	31.5	4.80e-6	2.84e-4
Bsds300	OT-Flow	7.1	37K	0.70	56	432.7	5.54e-5	4.24e-4
	RNODE	106.6	16K	23.4	200	15 253.3	2.66e-6	1.64e-2
	FFJORD	166.1	18K	33.6	345	20 061.2	3.41e-6	6.52e-3

The results demonstrate the computational efficiency of OT-Flow relative to the state-of-the-art (Table 5.8). With the exception of the Gas data set, OT-Flow achieves comparable MMD to the state-of-the-art with drastically reduced training time. We attribute most of the training speedup to the efficiency from using our exact trace instead of the Hutchinson's trace estimation (Figure 5.2, Figure 5.8). On the testing set, our exact trace leads to faster testing time than the state-of-the-art's exact trace computation via AD (Table 5.1, Table 5.8). To evaluate the testing data, we use more time steps than for training, effectively re-discretizing the ODE at different points. The inverse error shows that OT-Flow is numerically invertible and suggests that it approximates the true solution of the ODE. Ultimately, OT-Flow's combination of OT-influenced regularization, reduced parameterization, DO approach, and


(a) Originals

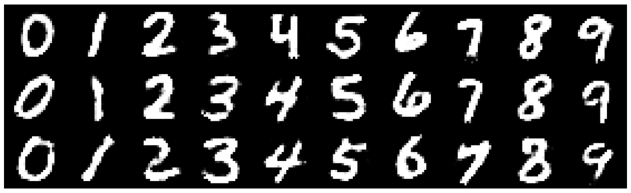

(b) Generated

Figure 5.12: MNIST generation conditioned by class. The encoder and decoder are trained prior and cause the slight thickness of the generations.

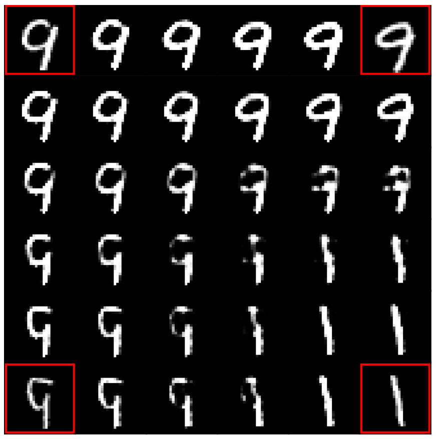


Figure 5.13: MNIST interpolation in the latent space. Original images are boxed in red.

efficient exact trace computation results in fast and accurate training and testing.

MNIST We demonstrate the generation quality of OT-Flow on the MNIST data set using an encoder-decoder structure. The MNIST data set contains 28×28-pixel images of handwritten numerical digits (Figure 5.12a).

Consider encoder $B \colon \mathbb{R}^{784} \to \mathbb{R}^d$ and decoder $D \colon \mathbb{R}^d \to \mathbb{R}^{784}$ such that $D(B(\boldsymbol{x})) \approx \boldsymbol{x}$. We train d-dimensional flows that map distribution $\rho_0(B(\boldsymbol{x}))$ to ρ_1. The encoder and decoder each use a single dense layer and activation function (ReLU for B and sigmoid for D). We train the encoder-decoder separate from and prior to training the flows. The trained encoder-decoder, due to its simplicity, renders digits $D(B(\boldsymbol{x}))$ that are a couple pixels thicker than the supplied digit $\boldsymbol{x}$.

We generate new images via two methods. First, using d=64 and a flow conditioned on class, we sample a point $\boldsymbol{y} \sim \rho_1(\boldsymbol{y})$ and map it back to the pixel space to create image $D(f^{-1}(\boldsymbol{y}))$ (Figure 5.12b). Second, using d=128 and an unconditioned flow, we interpolate between the latent representations $f(B(\boldsymbol{x}_1)), f(B(\boldsymbol{x}_2))$ of original images $\boldsymbol{x}_1, \boldsymbol{x}_2$. For interpolated latent vector $\boldsymbol{y} \in \mathbb{R}^d$, we invert the flow and decode back to the pixel space to create image $D(f^{-1}(\boldsymbol{y}))$ (Figure 5.13).

Chapter 6

Path-Finding

Path-finding problems are modeled by an ODE. Fitting within the OC framework, path-finding problems are often solved with the PMP or HJB (Section 2.3). The PMP approach works well in high dimensions for a single initial condition. The HJB approach works well for many initial conditions but scales poorly to high dimensions. We design and apply a neural ODE to blend the benefits of the PMP and HJB approaches to solve several path-finding problems. This chapter heavily incorporates portions from Onken et al. [78][1] and ongoing work with the same authors.

6.1 Problem

Consider n centrally controlled, homogeneous agents at initial locations $x_1, x_2, \dots, x_n \in \mathbb{R}^q$. Viewing all agents as part of a single OC system (Section 2.3), we represent the initial state of the system as $\boldsymbol{x} = (x_1, x_2, \dots, x_n) \in \mathbb{R}^d$. We use the agents have individual controls u_i which similarly are formulated as the system control $\boldsymbol{u_x} = (u_1, u_2, \dots, u_n) \in \mathbb{R}^a$. The dynamics F of the system follow

$$\partial_t \boldsymbol{z_x}(t) = F(\boldsymbol{z_x}(t), \boldsymbol{u_x}(t), t), \quad 0 \le t \le T, \quad \boldsymbol{z_x}(0) = \boldsymbol{x}. \tag{6.1}$$

[1]This work is under ©2021 EUCA.

similar to the general form in (2.9). We want the agents to have a particular target state $\boldsymbol{y} \in \mathbb{R}^d$ at final-time. Specifically, we minimize

$$G(\boldsymbol{z}(T)) = \frac{\alpha_0}{2} \|\boldsymbol{z}(T) - \boldsymbol{y}\|^2, \tag{6.2}$$

for some multiplier (weighting term) α_0. We also want to encode some optimality of the agent's paths and avoidance with each other in the L term.

In our setting, the Lagrangian L consists of three terms: an energy term $E \colon \mathbb{R}^d \times U \to \mathbb{R}$ which penalizes how much the agents travel, an obstacle term $Q \colon \mathbb{R}^d \times U \to \mathbb{R}$, which penalizes agents from spatial locations (i.e., a terrain function), and an interaction term $W \colon \mathbb{R}^d \times U \to \mathbb{R}$, which penalizes the proximity among agents (i.e, collision avoidance). As a result, we define the Lagrangian as

$$\begin{aligned} L\big(\boldsymbol{z_x}(t), \boldsymbol{u_x}(t), t\big) &= E\big(\boldsymbol{z_x}(t), \boldsymbol{u_x}(t)\big) + \alpha_1 Q\big(\boldsymbol{z_x}(t), \boldsymbol{u_x}(t)\big) + \alpha_2 W\big(\boldsymbol{z_x}(t), \boldsymbol{u_x}(t)\big) \\ &= \sum_{i=1}^{n} E_i\big(z_i(t), u_i(t)\big) + \alpha_1 \sum_{i=1}^{n} Q_i\big(z_i(t), u_i(t)\big) + \alpha_2 \sum_{j \neq i} W_{ij}\big(z_i(t), z_j(t)\big). \end{aligned} \tag{6.3}$$

Here, W_{ij} implicitly depends on the controls since the agents z_i depend on u_i. We remove this for brevity. In this framework, the dimensionality is proportional to the number of agents. The scalar parameters, α_1 and α_2, control the magnitude of the penalization of the obstacle and interactions, respectively. In general, these interactions may not be symmetric, and agents may be heterogeneous. However, for our problems, we assume the agents are identical. In particular, to model the interaction, we choose

$$W_{ij}(z_i, z_j) = \begin{cases} \exp\left(-\frac{\|z_i - z_j\|_2^2}{2r^2}\right), & \|z_i - z_j\|_2 < 2r, \\ 0, & \text{otherwise,} \end{cases} \tag{6.4}$$

where the radius r applies to an agent's space bubble or boundary. In this choice of W_{ij} encoded using piecewise Gaussian repulsion, agents avoid overlapping their space bubbles.

The energy term E and obstacles Q are problem-dependent (Section 6.5). The energy depends on the control formulation. The obstacles can vary in dimensionality and shape. We investigate using smooth obstacles, such as hills, and obstacles with a hard boundary such as rectangular prisms.

6.2 Related Works

Path-finding problems for many agents fit into the high-dimensional OC realm. We address the existing literature for solving deterministic and stochastic problems with NNs and approaches to path-finding problems.

6.2.1 High-Dimensional Deterministic Optimal Control

A common difficulty in solving high-dimensional OC problems is CoD, where the computational costs grow exponentially with spatial dimension. Lin et al. [63] address this by using the Hopf-Lax formulas to create a splitting technique that allows for quick trajectory generation without the need to discretize over a grid.

Kang and Wilcox [50] alleviate the CoD by introducing a sparse grid in the state space and use the method of characteristics to solve boundary value problems over each sparse grid point. To approximate the feedback control at arbitrary points, they interpolate the solutions of the grid using high-order polynomials. The authors solve up to six-dimensional control problems. Nakamura-Zimmerer et al. [69] also attempt to alleviate CoD by learning a closed-form value function. First, trajectories are generated in a similar manner as in Kang and Wilcox [50]. Using a supervised learning approach, the NN is trained to match the generated trajectories. The trajectories (training data) are generated adaptively using information about the costate and by combining progressive batching with an efficient adaptive sampling technique.

Our work stems from the same framework as Kunisch and Walter [57], which approximates the feedback control with an NN then optimizes the control cost on a distribution of initial states. The authors also provide a theoretical analysis of OC solutions via NN approximations. We extend the framework to finite horizon problems with non-quadratic costs and parameterize the value function instead of the feedback function. This extension enables enforcing HJB conditions via penalizer terms; such terms empirically improve numerical performance for solving high-dimensional mean-field games and mean-field control [64, 93], similar to the advantages shown in Section 5.4.

Our work also bears close resemblance to existing methods [64, 77, 93], which make similar use of NNs to parameterize the value function.

6.2.2 High-Dimensional Stochastic Optimal Control

In the seminal works [26, 44], the authors solve high-dimensional semilinear parabolic PDE problems by the method of (stochastic) characteristics. To overcome CoD, they approximate the gradient of the solution at different times by NNs and introduce a loss function that measures the deviation from the correct terminal condition in the characteristic equations. In particular, they solve high-dimensional stochastic OC problems by solving the corresponding viscous HJB equation. This method recovers the gradient of the solution as a function of $(\boldsymbol{x}, t)$ and can be considered a global method. Nevertheless, loss functions employed in E et al. [26] and Han et al. [44] consider only one initial point at a time, and the generalization depends on how well the generated random trajectories fill the space. The variance of the trajectories increases as time grows. Finally, in the deterministic limit the method becomes local as there is no diffusion to enforce the trajectories to explore the whole space. Similar techniques are applied in Nüsken and Richter [75] based on different loss functions.

In Han and E [43], the authors solve stochastic OC problems by directly approximating controls and using the control objective as a loss function. As in E et al. [26] and Han et al. [44], the loss function considers a single initial point.

6.2.3 Multi-Agent Path-Finding

Multi-Agent Path-Finding (MAPF) [102] methods are methods tailored for multi-agent control problems. These methods tend to focus on collision avoidance rather than optimality. Among these are Conflict-Based Search (CBS) methods [97, 111], which are two-level algorithms. At the low level, optimal paths are found for individual agents, while at the high-level, a search is performed in a constraint tree whose nodes include constraints on time and location for a single agent. Decoupled optimization approaches [28, 47] first compute independent paths and then try to avoid collision afterwards. These methods are often combined with graph-based methods [101], sub-dimensional expansions [112], and CBS approaches [11, 20]. Another approach phrases the MAPF problem as a differential game [68]. Provided certain assumptions, this differential game strategy guarantees that the agents reach their targets while avoiding collisions. Machine learning approaches for multi-agent control have also been successfully applied in where supervised learning is used to imitate non machine learning solutions generated [88]. Our approach differs from these methods in that we do not have a data generation and fitting/imitation phases; instead, we directly solve for the control objective. Additionally, localization and interaction modeling techniques [98] can be incorporated in our model in a straightforward manner.

6.3 Neural ODE Formulation

We provide a detailed presentation of our formulation, leveraging the advantages of both the PMP and HJB approaches. In particular, we directly optimize the cost (2.11) subject to (6.1). However, rather than solving for the controls, we use (2.13) and (2.14) to represent the dynamics and cost in terms of the value function parameterized by an NN. Moreover, since we know the value function solves the HJB, we add terms that penalize deviations from the HJB equations similar to Section 5.4.

6.3.1 Main Formulation

Our goal is to eliminate $\boldsymbol{u}_{\boldsymbol{x}}, F, L$ from the optimization problem (2.11) and obtain concise expressions for the dynamics and control cost in terms of the state and adjoint variables. Under this Assumption 2.3.1 and equations (2.12) and (2.13), the envelope formula [31, Section 3.1, Theorem 1] yields

$$\begin{aligned} F\big(\boldsymbol{x}, \boldsymbol{u}^*(\boldsymbol{x}, \boldsymbol{p}, t),\, t\big) &= -\nabla_{\boldsymbol{p}} H(\boldsymbol{x}, \boldsymbol{p}, t) \\ L\big(\boldsymbol{x}, \boldsymbol{u}^*(\boldsymbol{x}, \boldsymbol{p}, t),\, t\big) &= \boldsymbol{p} \cdot \nabla_{\boldsymbol{p}} H(\boldsymbol{x}, \boldsymbol{p}, t) - H(\boldsymbol{x}, \boldsymbol{p}, t) \end{aligned} \tag{6.5}$$

Also, by (2.15),

$$\boldsymbol{u}_{\boldsymbol{x}}(t) = \boldsymbol{u}^*\big(\boldsymbol{z}_{\boldsymbol{x}}(t), \boldsymbol{p}_{\boldsymbol{x}}(t), t\big), \quad 0 \le t \le T, \tag{6.6}$$

where $\boldsymbol{p}_{\boldsymbol{x}}(t)$ is the adjoint state. Therefore, for controls of the form (6.6), we obtain

$$\begin{aligned} \partial_t \boldsymbol{z}_{\boldsymbol{x}}(t) &= -\nabla_{\boldsymbol{p}} H\big(\boldsymbol{z}_{\boldsymbol{x}}(t), \boldsymbol{p}_{\boldsymbol{x}}(t), t\big), \\ L\big(\boldsymbol{z}_{\boldsymbol{x}}(t), \boldsymbol{u}_{\boldsymbol{x}}(t), t\big) &= \boldsymbol{p}_{\boldsymbol{x}}(t) \cdot \nabla_{\boldsymbol{p}} H\big(\boldsymbol{z}_{\boldsymbol{x}}(t), \boldsymbol{p}_{\boldsymbol{x}}(t), t\big) - H\big(\boldsymbol{z}_{\boldsymbol{x}}(t), \boldsymbol{p}_{\boldsymbol{x}}(t), t\big), \text{ and} \end{aligned} \tag{6.7}$$

From Theorem 2.3.2, we recognize that the adjoint variable must have the form $\boldsymbol{p}_{\boldsymbol{x}}(t) = \nabla \Phi(\boldsymbol{z}_{\boldsymbol{x}}(t), t)$. Consequently, (2.11) can be equivalently written as the optimization problem

$$\begin{aligned}
&\inf_{\Phi} \mathop{\mathbb{E}}_{\boldsymbol{x}\sim \mathrm{P}_0(\boldsymbol{x})} \left\{ \int_0^T L\big(\boldsymbol{z}_{\boldsymbol{x}}(t), \boldsymbol{u}_{\boldsymbol{x}}(t), t\big)\,\mathrm{d}t \,+\, G\big(\boldsymbol{z}_{\boldsymbol{x}}(T)\big) \right\} \\
&\text{s.t.} \begin{cases} \boldsymbol{p}_{\boldsymbol{x}}(t) = \nabla \Phi(\boldsymbol{z}_{\boldsymbol{x}}(t), t) \\ \partial_t \boldsymbol{z}_{\boldsymbol{x}}(t) = -\nabla_{\boldsymbol{p}} H(\boldsymbol{z}_{\boldsymbol{x}}(t), \boldsymbol{p}_{\boldsymbol{x}}(t), t),\ \ \boldsymbol{z}_{\boldsymbol{x}}(0) = \boldsymbol{x}. \end{cases}
\end{aligned} \tag{6.8}$$

The initial conditions $\boldsymbol{x} \in \mathbb{R}^d$ are drawn from some initial distribution P_0. In each problem, we set up and solve (6.8), where the dynamics F, the Lagrangian L, and thus the Hamiltonian H vary with the problem.

As in Section 5.4, we approximate Φ by an NN, denoted $\Phi(\cdot;\boldsymbol{\theta})$, which turns (6.8) into a finite-dimensional optimization over the weights $\boldsymbol{\theta}$. For brevity, we often omit the explicit dependence of Φ on $\boldsymbol{\theta}$. We use the same model for Φ (5.28) as in OT-Flow.. This approach does not require first solving for sample trajectories to generate training data and thus differs from the supervised training approaches presented in [69]. Instead, our approach aligns more with model-based reinforcement learning (Section 2.5).

6.3.2 Adding Hamilton-Jacobi-Bellman Penalizers

We introduce three penalty terms $c_{\mathrm{HJt},\boldsymbol{x}}$, $c_{\mathrm{HJfin},\boldsymbol{x}}$, and $c_{\mathrm{HJgrad},\boldsymbol{x}}$ derived from the HJB PDE (2.16) as follows:

$$\begin{aligned}
c_{\mathrm{HJt},\boldsymbol{x}}(t) &= \int_0^t \left| \partial_s \Phi\big(\boldsymbol{z}_{\boldsymbol{x}}(s), s\big) - H\Big(\boldsymbol{z}_{\boldsymbol{x}}(s), \nabla\Phi\big(\boldsymbol{z}_{\boldsymbol{x}}(s), s\big), s\Big) \right| \mathrm{d}s \\
c_{\mathrm{HJfin},\boldsymbol{x}} &= |\, \Phi(\boldsymbol{z}_{\boldsymbol{x}}(T), T) - G(\boldsymbol{z}_{\boldsymbol{x}}(T)) \,| \\
c_{\mathrm{HJgrad},\boldsymbol{x}} &= |\, \nabla\Phi(\boldsymbol{z}_{\boldsymbol{x}}(T), T) - \nabla G(\boldsymbol{z}_{\boldsymbol{x}}(T)) \,|.
\end{aligned} \tag{6.9}$$

Penalizers prove helpful in similar problems without introducing bias (Section 5.4.1). Adding $c_{\mathrm{HJt},\boldsymbol{x}}, c_{\mathrm{HJfin},\boldsymbol{x}}, c_{\mathrm{HJgrad},\boldsymbol{x}}$ to (6.8) and rewriting the time-integral in terms of

ODE constraints, we obtain

$$\min_{\Phi} \mathop{\mathbb{E}}_{\boldsymbol{x}\sim \mathrm{P}_0(\boldsymbol{x})} \left\{ c_{\mathrm{L},\boldsymbol{x}}(T) + G(\boldsymbol{z}_{\boldsymbol{x}}(T)) + \beta_1 c_{\mathrm{HJt},\boldsymbol{x}}(T) + \beta_2 c_{\mathrm{HJfin},\boldsymbol{x}} + \beta_3 c_{\mathrm{HJgrad},\boldsymbol{x}} \right\}, \tag{6.10}$$

subject to

$$\partial_t \begin{pmatrix} \boldsymbol{z}_{\boldsymbol{x}}(t) \\ c_{\mathrm{L},\boldsymbol{x}}(t) \\ c_{\mathrm{HJt},\boldsymbol{x}}(t) \end{pmatrix} = \begin{pmatrix} -\nabla_{\boldsymbol{p}} H\Big(\boldsymbol{z}_{\boldsymbol{x}}(t), \nabla\Phi(\boldsymbol{z}_{\boldsymbol{x}}(t),t)\,,t\Big) \\ L_{\boldsymbol{x}}(t) \\ R_{\boldsymbol{x}}(t) \end{pmatrix}, \tag{6.11}$$

initialized with $\boldsymbol{z}_{\boldsymbol{x}}(0) = \boldsymbol{x}$ and $c_{\mathrm{L},\boldsymbol{x}}(0) = c_{\mathrm{HJt},\boldsymbol{x}}(0) = 0$, and

$$\begin{aligned} L_{\boldsymbol{x}}(t) =& \nabla\Phi(\boldsymbol{z}_{\boldsymbol{x}}(t),t)\cdot \nabla_{\boldsymbol{p}} H\Big(\boldsymbol{z}_{\boldsymbol{x}}(t), \nabla\Phi(\boldsymbol{z}_{\boldsymbol{x}}(t),t)\,,t\Big) - H\Big(\boldsymbol{z}_{\boldsymbol{x}}(t), \nabla\Phi(\boldsymbol{z}_{\boldsymbol{x}}(t),t)\,,t\Big) \\ R_{\boldsymbol{x}}(t) =& \Big|\partial_t \Phi\big(\boldsymbol{z}_{\boldsymbol{x}}(t),t\big) - H\Big(\boldsymbol{z}_{\boldsymbol{x}}(t), \nabla\Phi\big(\boldsymbol{z}_{\boldsymbol{x}}(t),t\big)\,,t\Big)\Big|. \end{aligned}$$

We note that reformulating the Lagrangian $L_{\boldsymbol{x}}$ uses (6.5) and that $L_{\boldsymbol{x}}$ and $R_{\boldsymbol{x}}$ are generalized forms of their use in Section 5.4.1.

The objective function thus contains the accumulated running cost $c_{\mathrm{L},\boldsymbol{x}}(T)$, the HJB penalty along the trajectories $c_{\mathrm{HJt},\boldsymbol{x}}(T)$, the final-time HJB penalty $c_{\mathrm{HJfin},\boldsymbol{x}}$, and the transversality penalty $c_{\mathrm{HJgrad},\boldsymbol{x}}$. The penalty multipliers $\beta_1, \beta_2, \beta_3 > 0$ are hyperparameters of the model (Section 6.4). We address the empirical effectiveness of the penalizers in Section 6.5.2.

6.3.3 Robustness to Shocks

Instead of approximating individual trajectories give initial data $\boldsymbol{x}$, we approximate the dynamics

$$\partial_t \boldsymbol{z} = -\nabla_{\boldsymbol{p}} H\big(\boldsymbol{z}, \nabla\Phi(\boldsymbol{z},t),t\big) \tag{6.12}$$

that generate optimal paths. To accomplish this, we parameterize the value function Φ with an NN [57, 64, 69, 93, 118]. Because the value function is global, obtained models are robust to various disturbances, such as sudden shocks to the system, that alter the initial data.

In Section 6.5.3, we experimentally verify the model's robustness to shocks. More specifically, we observe that the NN controller successfully drives the system to the target after abrupt changes in the initial data as a result of minor and major shocks (Figure 6.3). Minor shocks result in initial data within the training distribution defined by P_0 in (6.8). As a result, the performance of the NN controller does not degrade (Figure 6.3a, Table 6.3). Major shocks result in initial data significantly far from the training distribution; they potentially can significantly degrade the performance of the NN controller in terms of the cost (Figure 6.3c, Table 6.3). Remarkably, however, the NN controller manages to safely drive the system towards the desired location (Figure 6.3b).

We attribute the shock robustness to the NN parameterization of the global value function. Experimentally, we find that penalizers help in training convergence (Figure 6.2) but not necessarily in shock robustness (cf. Figures 6.3,6.4).

6.4 Numerics

We solve the ODE-constrained optimization problem (6.10) using the DO approach (Section 3.2), in which we define a discretization of the ODE, then optimize on that discretization. The forward pass of the model uses an RK4 integrator with n_t time steps to approximate the constraints (6.11). The objective function is then computed; AD [72] calculates the gradient of the objective function with respect to $\boldsymbol{\theta}$; ADAM [52] updates the parameters $\boldsymbol{\theta}$. We iterate this process a selected maximum number of times. For the learning rate (step size) provided to ADAM, we follow a piece-wise

Table 6.1: NN training statistics. All timings are approximate from training on a shared NVIDIA Quadro RTX 8000 GPU.

	# Params	# Iters	$\frac{\text{Time}}{\text{Iter}}$ (s)	Training Time (min)
Corridor	1,311	1800	0.32	10
Swap 2 [68]	415	4000	0.56	37
Swap 12 [68]	2,196	4000	0.26	17
Swarm [47]	342,654	6000	0.57	57
Quadcopter [63]	18,576	6000	0.72	72

constant decay schedule. For instance, in the experiment in Figure 6.2, we divide the learning rate by 10 every 800 iterations.

To produce an NN that generalizes to the state-space, we must define initial points in a manner to promote model generalizability. We assume the initial points are drawn from a distribution P_0. In training the NN, we train on a batch of samples at a time, where the batch is one sampled set from the distribution. After training many iterations on that batch, we resample the distribution to define a new batch and train many iterations on that batch. We repeat this process until we hit the maximum number of iterations. As an example, we commonly choose batches of 1024 or 2048 samples which were re-sampled every 25-100 iterations. We found no noticeable empirical difference in solution quality across those ranges. Through this process, the model uses few data points at each iteration, but does not overfit to a specific set of data points.

6.4.1 Hyperparameter Tuning

The number of time steps n_t is selected *a priori* as a hyperparameter of the model. Large n_t leads to high computation and training time while reducing error; meanwhile, too small n_t leads to overfitting to a refinement of the time discretization of the trajectories. To check for overfitting, we use more time steps for the hold-out validation set as addressed in Chapters 3 and 5. We similarly check for overfitting for

Table 6.2: Variables and hyperparameters inherent to the problem itself (shared for NN and baseline) and the hyperparameters tuned for the NN approach. All α_i values are determined relative to the α-less E term in the problem definition. The β_i hyperparameters are tuned relative to the α values.

	Problem Definition				
	n # agents	d dim.	α_1 on G	α_2 on Q	α_3 on W
Corridor	2	4	100	10^4	300
Swap 2 [68]	2	4	300	10^6	10^5
Swap 12 [68]	12	24	300	-	10^5
Swarm [47]	50	150	900	10^7	25000
Quadcopter [63]	1	12	5000	-	-

	NN-specific Hyperparameters					
	m width	β_1 on HJt	β_2 on HJfin	β_3 on HJgrad	n_t training	n_t validation
Corridor	32	0.02	0.02	0.02	20	50
Swap 2 [68]	16	1	1	3	20	50
Swap 12 [68]	32	5	2	5	20	50
Swarm [47]	512	2	1	3	26	80
Quadcopter [63]	128	0.1	0	0	26	50

the baseline approach and observe stable results for different n_t. Training on a single NVIDIA Quadro RTX 8000 GPU requires between ten minutes and a little over one hour for the considered OC problems (Table 6.1).

Other hyperparameters include the number of ResNet layers (tuned to equal 2), width of the ResNet m, and the multipliers $\beta_1, \beta_2, \beta_3$. In contrast, each OC problem has particular $\alpha_1, \alpha_2, \alpha_3$ defined which we use for both the baseline and NN; changing these values alters the problem (Table 6.2).[2]

ResNet Hyperparameters

For our ResNet N (5.29), each network layer is a dense layer (2.1). Varying network depths in solving path-finding problems revealed no noticeable benefits, so we stick

[2]For reproducibility, our Python implementation and all hyperparameters are available at `https://github.com/donken/NeuralOC`.

to a two-layer ResNet for simplicity and speed.

The ResNet width m describes the dimension of the hidden space onto which the model projects. Specifically, multiplying the model inputs by $\boldsymbol{K}_0$, returns an m-dimensional result for subsequent operators. The manifold [99] onto which the model projects the data exists in this hidden space. If the precise dimensionality of the manifold and the projection are known, then one could set m as the manifold dimension and define the network to be the projection. Machine learning approaches rely on the model learning the manifold and projection. Therefore, in practice, one tunes the architecture width m to be greater than or equal to the manifold dimensionality. In theory, if m exceeds the manifold dimensionality, then since manifolds exist in higher dimensional spaces, the model could learn to ignore the additional unnecessary dimensions.

We tune hyperparameter m so that we obtain the smallest model without sacrificing performance. We prefer smaller models as a model with few parameters is more interpretable and easier to train as the parameter space in which we optimize $\boldsymbol{\theta}$ is reduced. Often, smaller models also present quicker computation with fewer FLOPs. However, we train models on a GPU which parallelizes the matrix-vector computations so the different model widths in our experiments have negligible influence on time per training iteration.

6.5 Numerical Experiments

We provide a baseline local solution method and solve several multi-agent path-finding problems and one high-dimensional quadcopter problem with complicated dynamics.

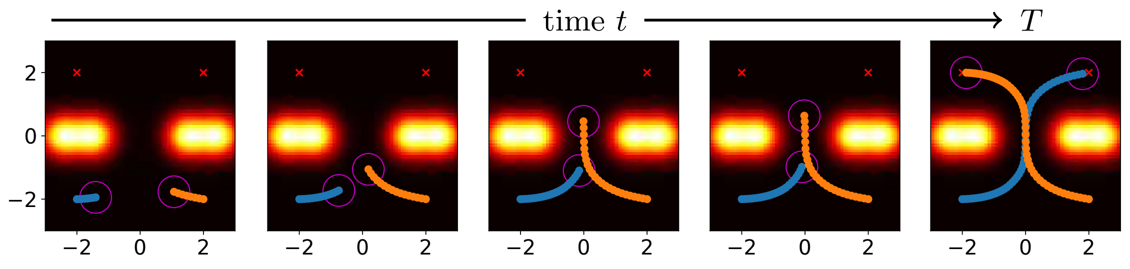
6.5.1 Baseline: Discrete Optimization for a Single Initial State

For comparison with the NN approach, we provide a local solution method that solves the OC problem for a fixed initial state $\boldsymbol{z}^{(0)} = \boldsymbol{x}_0$. To this end, we obtain a discrete optimization problem by applying forward Euler to the state equation and a midpoint rule to the integrals, which leads to

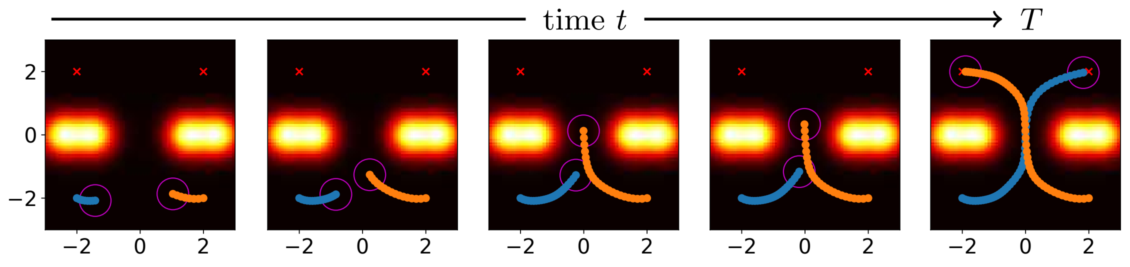
$$\begin{aligned} \min_{\{\boldsymbol{u}^{(k)}\}} \quad & G\left(\boldsymbol{z}^{(n_t)}\right) + h \sum_{k=0}^{n_t-1} L\left(\boldsymbol{z}^{(k)}, \boldsymbol{u}^{(k)}, t_k\right) \\ \text{s.t.} \quad & \boldsymbol{z}^{(k+1)} = \boldsymbol{z}^{(k)} + h\, F\big(\boldsymbol{z}^{(k)}, \boldsymbol{u}^{(k)}, t_k\big), \end{aligned} \tag{6.13}$$

where $h{=}T/n_t$. We use $T{=}1$ and $n_t{=}50$ and solve (6.13) using ADAM with initialization of the controls set as straight paths from $\boldsymbol{z}^{(0)}$ to $\boldsymbol{y}$ with small added Gaussian noise.

We arrived at these training decisions empirically. First, when solving (6.13) in our experiments, ADAM finds slightly more optimal solutions ($1-2\%$ more optimal) in practice than L-BFGS. Second, the initialization of the controls substantially influences the solution. As a particular example, the baseline solution depicted in Figure 6.3c learns to send agent 1 around the left side of the obstacle; the baseline struggles to learn this optimal trajectory if initialized with controls that point towards passing through the right of that obstacle or through the corridor. As a response, we initialize with controls of equal magnitude pointing in a straight trajectory from initial point to target. Third, we find examples where initializing with these completely straight trajectories can lead to poor optima that can even model collisions, e.g., when solving the problem in Figure 6.1a. We find that adding small Gaussian noise to the straight trajectories fixes this issue, and thus we initialize with controls that dictate a noisy straight trajectory.



(a) Baseline.



(b) NN model. Solved for multiple pairs; only one shown.

Figure 6.1: Two-agent corridor problem.

6.5.2 Two-Agent Corridor Experiment

We design a four-dimensional problem in which two agents attempt to reach fixed targets on the other side of two hills. We design the hills in such a manner that one agent must pass through the corridor between the two hills while the other agent waits. For this example, the hills use a smooth terrain, and we assess the shock-resilient nature of the NN.

Set-up

Suppose the two homogeneous agents with radius $r{=}0.5$ start at $x_1{=}[-2,-2]^\top$ and $x_2{=}[2,-2]^\top$ with respective targets $y_1{=}[2,2]^\top$ and $y_2{=}[-2,2]^\top$. Thus, the initial and target joint-states are $\boldsymbol{x}_0{=}[-2,-2,2,-2]^\top$ and $\boldsymbol{y}{=}[2,2,-2,2]^\top$. We sample from P_0, which is a Gaussian centered at $\boldsymbol{x}_0$ with an identity covariance. These sampled initial positions form the training set $\boldsymbol{X}$.

The running costs depend on the spatio-temporal cost function Q_i. Throughout, our obstacles use the Gaussian density function with mean $\boldsymbol{\mu}$ and covariance $\boldsymbol{\Sigma}$

$$\nu(z_i\,;\,\boldsymbol{\mu},\boldsymbol{\Sigma}) = \frac{\exp\left(-\frac{1}{2}(z_i-\boldsymbol{\mu})\boldsymbol{\Sigma}^{-1}(z_i-\boldsymbol{\mu})\right)}{\sqrt{(2\pi)^d \det \boldsymbol{\Sigma}}}.$$

In this experiment, we define the corridor between obstacles

$$\begin{aligned} Q_i(z_i) = \nu\left(z_i\,;\begin{bmatrix}-2.5\\0\end{bmatrix}, 0.2\boldsymbol{I}\right) + \nu\left(z_i\,;\begin{bmatrix}2.5\\0\end{bmatrix}, 0.2\boldsymbol{I}\right)\\ +\nu\left(z_i\,;\begin{bmatrix}-1.5\\0\end{bmatrix}, 0.2\boldsymbol{I}\right) + \nu\left(z_i\,;\begin{bmatrix}1.5\\0.0\end{bmatrix}, 0.2\boldsymbol{I}\right). \end{aligned}$$

The energy terms are given by

$$E_i\big(z_i(t), u_i(t)\big) = \frac{1}{2}\|u_i(t)\|^2, \tag{6.14}$$

and the dynamics are given by $F(\boldsymbol{z},\boldsymbol{u},t) = \boldsymbol{u}$.

We note that the Q and W terms do not directly depend on the controls $\boldsymbol{u_x}$, and we therefore write the Hamiltonian (2.12) as

$$\begin{aligned} H(\boldsymbol{z_x},\boldsymbol{p_x},t) &= \sup_{\boldsymbol{u_x}\in U}\left\{-\boldsymbol{p_x}^\top\boldsymbol{u_x} - L\big(\boldsymbol{z_x},\boldsymbol{u_x},t\big)\right\}\\ &= \sup_{\boldsymbol{u_x}\in U}\left\{-\boldsymbol{p_x}^\top\boldsymbol{u_x} - E\big(\boldsymbol{z_x},\boldsymbol{u_x}\big) - \alpha_2 Q\big(\boldsymbol{z_x}\big) - \alpha_3 W\big(\boldsymbol{z_x}\big)\right\} \end{aligned} \tag{6.15}$$

We then can solve for the first-order necessary condition

$$\begin{aligned} 0 &= -\boldsymbol{p_x} - \nabla_{\boldsymbol{u}} E\big(\boldsymbol{z_x},\boldsymbol{u_x}\big)\\ \Rightarrow\quad \boldsymbol{p_x} &= -\nabla_{\boldsymbol{u}}\left(\sum_{i=1}^{n}\frac{1}{2}\|u_i\|^2\right) = -\boldsymbol{u_x} \end{aligned} \tag{6.16}$$

Using the closed-form solution for the controls (6.16), we can rewrite the Hamiltonian as

$$\begin{aligned} H(\boldsymbol{z_x}, \boldsymbol{p_x}, t) &= \|\boldsymbol{p}\|^2 - \frac{1}{2}\|\boldsymbol{p_x}\|^2 - \alpha_2 Q\big(\boldsymbol{z_x}\big) - \alpha_3 W\big(\boldsymbol{z_x}\big) \\ &= \frac{1}{2}\|\boldsymbol{p_x}\|^2 - \alpha_2 Q\big(\boldsymbol{z_x}\big) - \alpha_3 W\big(\boldsymbol{z_x}\big) \end{aligned} \tag{6.17}$$

where the characteristics are given by

$$\partial_t \boldsymbol{z_x}(t) = -\nabla_{\boldsymbol{p}} H\big(\boldsymbol{z_x}(t), \boldsymbol{p_x}(t), t\big) = -\boldsymbol{p_x}(t). \tag{6.18}$$

Results

The baseline and the NN learn to wait for one agent to pass through the corridor first, followed by the second agent (Figure 6.1). The NN performs marginally worse in running cost (Table 6.3), which can be seen in the early stages of the trajectories of x_1 (Figure 6.1b). The NN achieves a slightly better G value than the baseline. The closeness of the values between the approaches is exciting; although we solve the NN by optimizing the expectation value of a set of points in the region, the NN achieves a near-optimal solution for $\boldsymbol{x}_0$.

6.5.3 Effect of the Hamilton-Jacobi-Bellman Penalizers

We experimentally assess the effectiveness of the penalizers c_{HJfin} and c_{HJgrad} when used alone or with each other. the c_{HJt} penalizer reduces the necessary number of time steps (Section 5.4.1). We define six models (combinations of the HJB penalizers and one using only weight decay) and train each on the corridor problem. The HJB penalizers results in quickest convergence on a hold-out validation set (Figure 6.2).

$\mathbf{HJ_t}$: We enforce the PDE (2.16) describing the time derivative of Φ along the trajectories. Including this penalizer improves regularity and reduces the necessary number of time steps when solving the dynamics [64, 77, 93, 118].

$\mathbf{HJ_{fin}}$: We enforce the final-time condition of the PDE (2.16). The inclusion of this

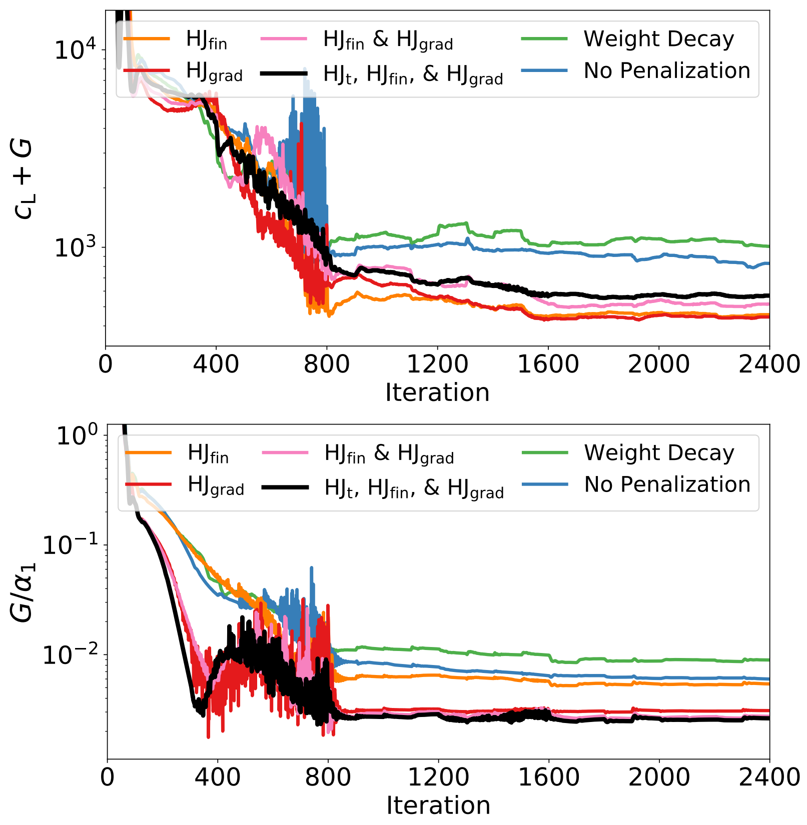
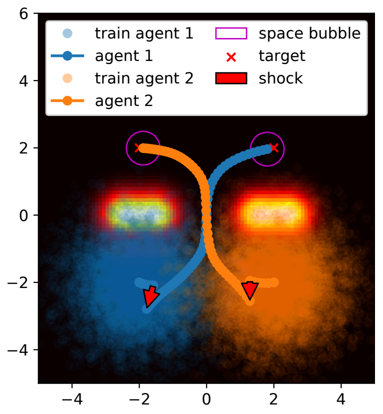


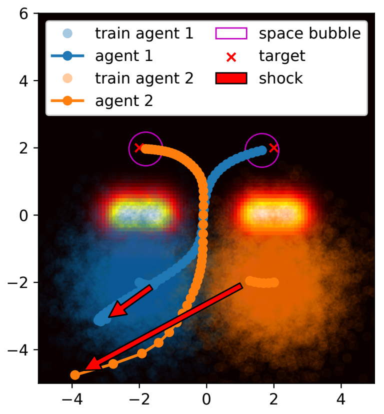
Figure 6.2: We train several models that only differed in the penalizers used in training. We compare against an unpenalized model and one that uses a typical weight decay (Tikohonov) regularizer. The terminal HJB penalizers lead to improved convergence and lower G value. Each curve is the average of three training instances.

penalizer helps the network achieve the target [93]. We observe this experimentally, where the inclusion of HJ$_{\text{fin}}$ correlates with a lower G value (Figure 6.2). HJ$_{\text{fin}}$ is also the sole aspect of our formulation that directly tunes the Φ, while all other aspects use $\nabla\Phi$.

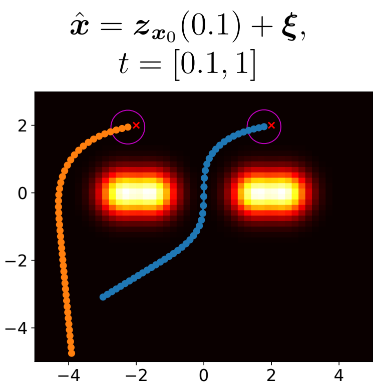
HJ$_{\text{grad}}$: We enforce the transversality condition $\nabla\Phi(\boldsymbol{z_x}(T),T)=\nabla G(\boldsymbol{z_x}(T))\ \forall \boldsymbol{z_x}$, a consequence of the final-time HJB condition (2.16). Numerically, all conditions are enforced on a finite sample set. Therefore, higher-order penalization may help the generalization; i.e., achieving a better match of $\Phi(\cdot,T)$ and G for samples not used during training (the hold-out validation set). We observe the latter experimentally; using HJ$_{\text{grad}}$ instead of HJ$_{\text{fin}}$ results in better validation convergence (Figure 6.2). The importance of enforcing the values of $\nabla\Phi$ was addressed in Nakamura-Zimmerer



(a) Minor shock $\|\boldsymbol{\xi}\|$=0.94 within the training space.



(b) Major shock $\|\boldsymbol{\xi}\|$=6.2 outside the training space.



(c) Baseline solution for $(\hat{\boldsymbol{x}}$,0.1) after major shock.

Figure 6.3: Shock-Robustness. The NN handles a shock $\boldsymbol{\xi}$ at time t=0.1.

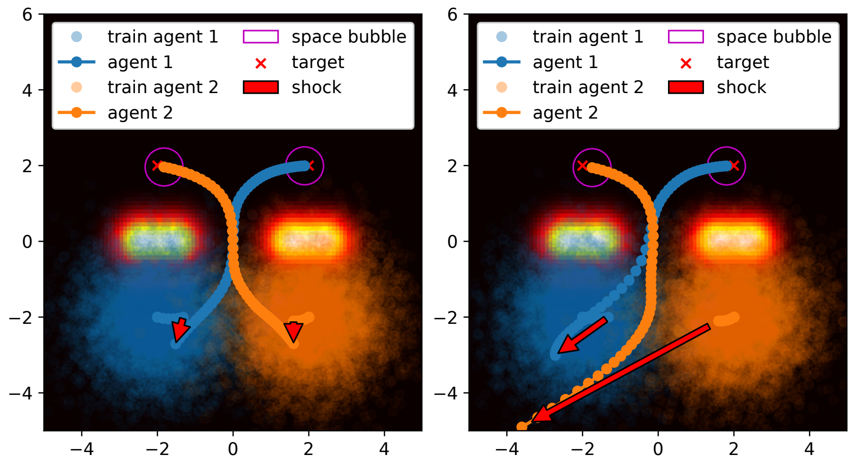
Table 6.3: Multiple Agent Corridor Problem. Comparison of values for single instance $\boldsymbol{x}_0$ (cf. Figures 6.1, 6.3).

Scenario	Method	$c_{\mathrm{L}} + G$	c_{L}	G
no shocks	Baseline	61.33	61.02	0.31
$t \in [0, 1]$	NN	62.19	61.98	0.21
following shock $\|\boldsymbol{\xi}\| = 0.94$	Baseline	59.79	59.46	0.33
$t \in [0.1, 1]$	NN	60.54	60.34	0.20
following shock $\|\boldsymbol{\xi}\| = 6.2$	Baseline	71.77	71.22	0.55
$t \in [0.1, 1]$	NN	151.67	150.63	1.03

et al. [69].

Shocks

We use this experiment to demonstrate how our approach is robust to shocks (Figure 6.3). Consider solving the control problem for $t \in [0, T]$ as always. Then for $T = 1$, we consider a shock $\boldsymbol{\xi}$ (implemented as a random shift) to the system at time $t = 0.1$. Our method is designed to handle minor shocks that stay within the space of trajectories of the initial distribution about $\boldsymbol{x}_0$. Our model computes a trajectory to $\boldsymbol{y}$ for many initial points. Therefore, for point $\widetilde{\boldsymbol{x}} \in \boldsymbol{X}$, the model provides dynamics $F(\boldsymbol{z}_{\widetilde{\boldsymbol{x}}}(t), \boldsymbol{u}_{\widetilde{\boldsymbol{x}}}(t), t)$ before the shock. After the shock, the state picks up the trajectory



(a) Minor shock $\|\boldsymbol{\xi}\| = 0.94$ within the training space.

(b) Major shock $\|\boldsymbol{\xi}\| = 6.2$ outside of the training space.

Figure 6.4: Comparable NN trained with no penalizers.

of some other point $\hat{\boldsymbol{x}} \in \boldsymbol{X}$ and follows that trajectory to $\boldsymbol{y}$ (Figure 6.3a). In this scenario, the total trajectory has two portions

$$\boldsymbol{z}_{\widetilde{\boldsymbol{x}}}(0.1) = \int_0^{0.1} F\big(\boldsymbol{z}_{\widetilde{\boldsymbol{x}}}(t), \boldsymbol{u}_{\widetilde{\boldsymbol{x}}}(t), t\big)\,\mathrm{d}t, \quad \boldsymbol{z}_{\widetilde{\boldsymbol{x}}}(0) = \widetilde{\boldsymbol{x}}, \quad \text{and}$$
$$\boldsymbol{z}_{\hat{\boldsymbol{x}}}(1) = \int_{0.1}^{1} F\big(\boldsymbol{z}_{\hat{\boldsymbol{x}}}(t), \boldsymbol{u}_{\hat{\boldsymbol{x}}}(t), t\big)\,\mathrm{d}t,\ \boldsymbol{z}_{\hat{\boldsymbol{x}}}(0.1) = \boldsymbol{z}_{\widetilde{\boldsymbol{x}}}(0.1) + \boldsymbol{\xi},$$

before and after the shock, respectively. We imagine a minor shock then as moving from one trajectory to another (Figure 6.3a). The results of the control problem along t=[0.1, 1] for the NN and baseline solutions are similar (Table 6.3).

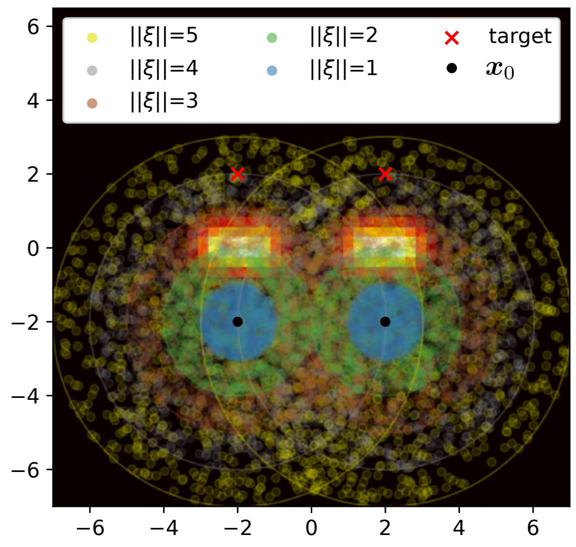
Interestingly, our model extends outside the training region (Figure 6.3b). Although the vast majority of NNs cannot extrapolate, our NN still solves the control problem after a major shock, demonstrating some extrapolation capabilities. We note that the NN solves the original problem for $\boldsymbol{x}_0$ to near optimality. However, after a large shock, the NN solves the control problem, but with little hope for optimality. In our example, we compare the NN's solution (Figure 6.3b) with the baseline solution for t=[0.1, 1] (Figure 6.3c). The NN learned a solution in which agent 2 passes through the corridor followed by agent 1. After the major shock, the NN still applies these dynamics (Figure 6.3b) while the baseline finds a more optimal solution (Figure 6.3c). The NN is roughly 100% less optimal in this single shock example (Table 6.3).

We attribute the shock robustness to the NN parameterization of the global value function. Experimentally, the shock robustness of our model (Figure 6.3) does not noticeably differ from a model trained without penalization. Since the NN is trained offline prior to deployment, it handles shocks in real-time. In contrast, methods that solve for a single trajectory—e.g., the baseline—must pause to recompute following a shock.

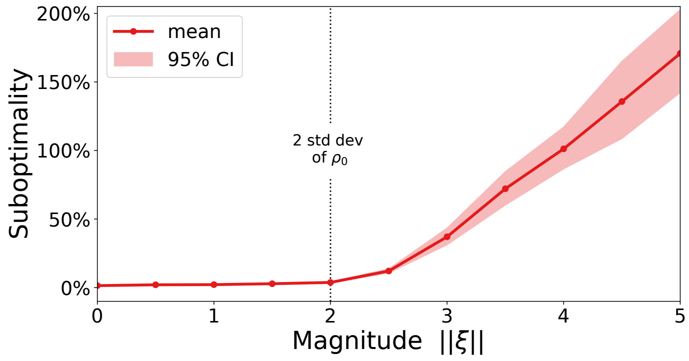
Global Capabilities of NN Model

To provide a more thorough analysis of our model's global optimality, we assess one NN's performance for many different initial conditions $\boldsymbol{x}_0 + \boldsymbol{\xi}$. We sample random shifts $\boldsymbol{\xi}$ of varying magnitudes. We sample 1000 random $\boldsymbol{\xi}$ for each magnitude $\|\boldsymbol{\xi}\| = 0.5, 1.0, \ldots, 5.0$. For each $\boldsymbol{x}_0 + \boldsymbol{\xi}$, we evaluate the trained NN model and train a baseline model, computing the suboptimality of the NN (Figure 6.5). We note that this equivalently can be described as comparing the NN model and the baseline on many samples from concentric hyperspheres. Since a shock can be phrased as picking up a trajectory from an initial condition, testing the NN's global capabilities and shock-robustness are synonymous.

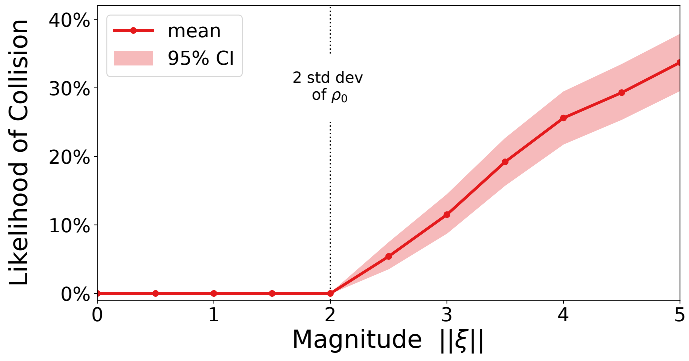
We observe that the NN optimality slowly decays as $\|\boldsymbol{\xi}\|$ increases (Figure 6.5). Specifically, for the corridor experiment, the NN performs near optimality within $\|\boldsymbol{\xi}\| \leq 2$. Since the NN was trained on P_0 which was a Gaussian about $\boldsymbol{x}_0$ with covariance $\boldsymbol{I}$. The bound $\|\boldsymbol{\xi}\| \leq 2$ then equates to being within two standard deviations of $\boldsymbol{x}_0$.



(a) The initial points $\boldsymbol{x}_0 + \boldsymbol{\xi}$ for the corridor problem sampled from hyperspheres of radius $\|\boldsymbol{\xi}\|$.



(b) The mean suboptimality of the NN's solution $c_{\mathrm{L}} + G$, where the baseline solution for each initial point is considered optimal.



(c) For initial points at each magnitude, we present the percentage of those resulting in a collision of any severity when run with the NN.

Figure 6.5: Comparison of one NN model with 10,001 baseline models for 1000 initial points $\boldsymbol{x}_0 + \boldsymbol{\xi}$ at each magnitude $\|\boldsymbol{\xi}\|$. Confidence intervals are computed via bootstrapping 10,000 sub-samplings of size 500 from each set of 1000 points (Appendix B).

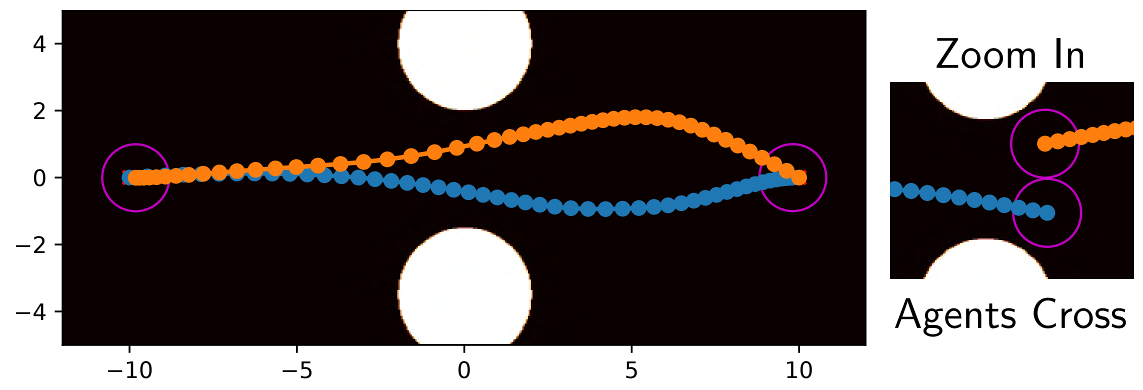


Figure 6.6: Swap experiment with hard-boundary corridor.

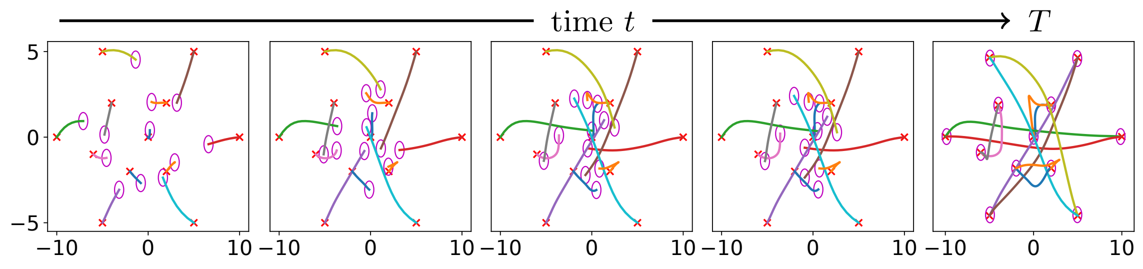


Figure 6.7: Swap experiment for 12 agents in $\mathbb{R}^2$.

6.5.4 Multi-Agent Swap Experiments

We replicate the experiments found in Mylvaganam et al. [68] where agents swap positions while avoiding each other. All agents are two-dimensional, and the formulation mostly matches that presented in the corridor example (Section 6.5.2). For the swap experiments, we alter $\boldsymbol{x}_0$, $\boldsymbol{y}$, and Q.

Setup

We begin with two agents that swap positions with each other while passing through a corridor with hard edges. To enforce these hard edges, we enforce a space bubble around obstacles similar to how we implement multi-agent interactions (6.4). Therefore, we train with this space bubble but evaluate and plot the results without it. The actual obstacles (two circles with radius 2) are formulated as follows. Let

$\Omega_{\text{obs}} = \{z \mid \|z - \boldsymbol{\mu}_1\| < 2 \quad \text{or} \quad \|z - \boldsymbol{\mu}_2\| < 2\}$, then

$$Q_i(z_i) = \begin{cases} 1, & \text{if } z_i \in \Omega_{\text{obs}}, \\ 0, & \text{otherwise,} \end{cases}$$

where $\boldsymbol{\mu}_1 = \begin{bmatrix} 0 \\ 4 \end{bmatrix}$ and $\boldsymbol{\mu}_2 = \begin{bmatrix} 0 \\ -3.5 \end{bmatrix}$. However, for training, we encode this as

$$Q_{i,\text{trn}}(z_i) = \begin{cases} \nu\left(z_i\,;\,\boldsymbol{\mu}_1, \boldsymbol{I}\right) + \nu\left(z_i\,;\,\boldsymbol{\mu}_2, \boldsymbol{I}\right), & \text{if } z_i \in \Omega_{\text{obs,trn}}, \\ 0, & \text{otherwise,} \end{cases}$$

where $\Omega_{\text{obs,trn}} = \{z \mid \|z - \boldsymbol{\mu}_1\| < 2.2 \quad \text{or} \quad \|z - \boldsymbol{\mu}_2\| < 2.2\}$. By training with Gaussian repulsion—which has gradient information within the obstacles—we incentivize the optimizer to learn trajectories avoiding the obstacles. Additionally, we increase the obstacle radial bound by ten percent to give the network some wiggle room. We use the same obstacle definitions for the baseline and NN approaches.

For initial and target states, we choose $\boldsymbol{x}_0 = [10, 0, -10, 0]^\top$ and $\boldsymbol{y} = [-10, 0, 10, 0]^\top$. These values are a scaled down version of those in [68] and are easier to visualize. For the two-agent problem, the agents successfully switch positions while avoiding each other (Figure 6.6). Qualitatively, our method learns trajectories with shorter arclength than those in Mylvaganam et al. [68].

We also replicate the 12-agent case [68]. For this experiment, six pairs of agents swap positions across the same space. Since there are no obstacles, $Q=0$. In our setup, the problem is slightly adjusted as our global problem solves for a fixed $\boldsymbol{y}$ but with initial conditions in P_0, instead of just $\boldsymbol{x}_0$. We display the solution for the single initial case $\boldsymbol{x}_0$ (Figure 6.7).

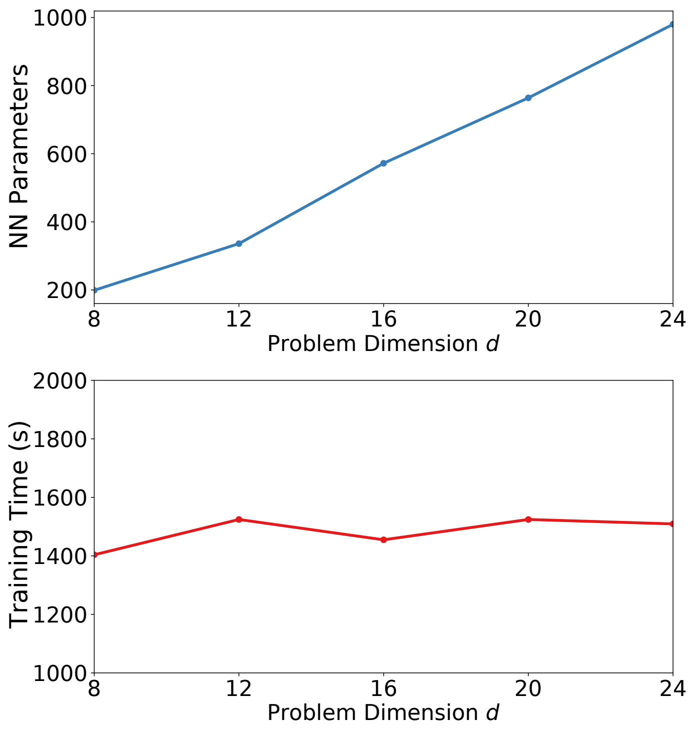


Figure 6.8: The NN's number of parameters scales linearly with the problem dimension as the computational cost remains mostly constant, overcoming CoD. For each problem (subproblem of the 12-agent swap experiment), we train the smallest NN that achieves at least 10% suboptimality.

Overcoming CoD

Expanding on the 12-agent swap experiment, we demonstrate how the NN model overcomes CoD (Figure 6.8). We design four additional similar problems by dropping out agents from the 12-agent version. Thus, we arrive at problems containing 2, 3, 4, 5, and 6 pairs of agents that swap positions. We select a fixed suboptimality of 10% and tune the smallest NN we can to achieve this suboptimality or better. The size of the NN is quantized by the NN width m, which we tune to make the NN small. The resulting NNs follow a linear growth of number of parameters relative to the problem dimension d (Figure 6.8). Due to the parallelization of the GPU, the training time of these models remains constant.

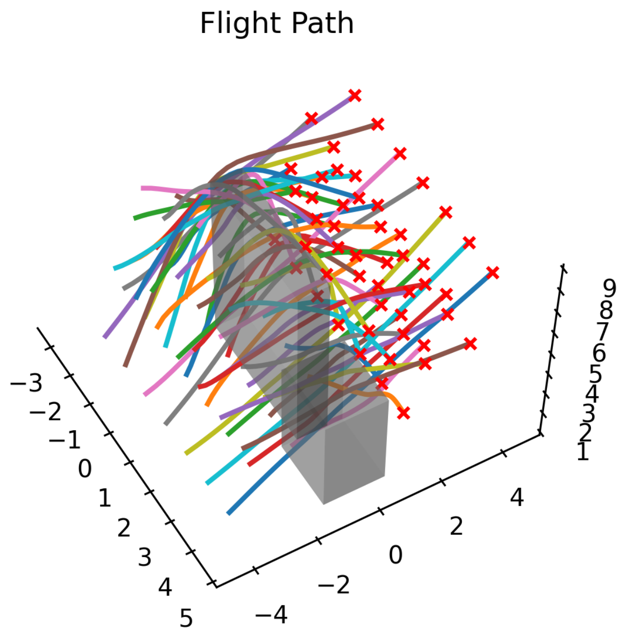


Figure 6.9: Swarm problem for 50 agents in $\mathbb{R}^3$.

6.5.5 Swarm Problem

We demonstrate the high-dimensional capabilities of our model by solving a swarm problem in the spirit of Hönig et al. [47]. The swarm problem contains 50 three-dimensional agents that fly from initial to target positions while avoiding each other and obstacles. We construct Q_i to model two rectangular prism obstacles $[-2,2] \times [-0.5,0.5] \times [0,7]$ and $[2,4] \times [-1,1] \times [0,4]$. We train with Gaussian repulsion inside the obstacles similar to the swap experiment (Section 6.5.4) and use the same dynamics (6.18). Due to the complexity of the collision avoidance, we find it beneficial to switch the scalar weights on the HJB penalizers during training—recall that the penalizers do not alter the solution (Section 6.5.3). For the first portion of training, we choose $\beta_1{=}2$, $\beta_2{=}1$, and $\beta_3{=}3$ (Table 6.2); for the rest of training, we use $\beta_1{=}\beta_2{=}\beta_3{=}0$. This set-up focuses the model on solving the control problem in the first portion of training as the final-time penalizers help the agents reach their destinations. We then reduce the weights of the penalizers for optimal fine-tuning.

The NN learns to guide all agents around the obstacles (Figure 6.9). Naturally, if

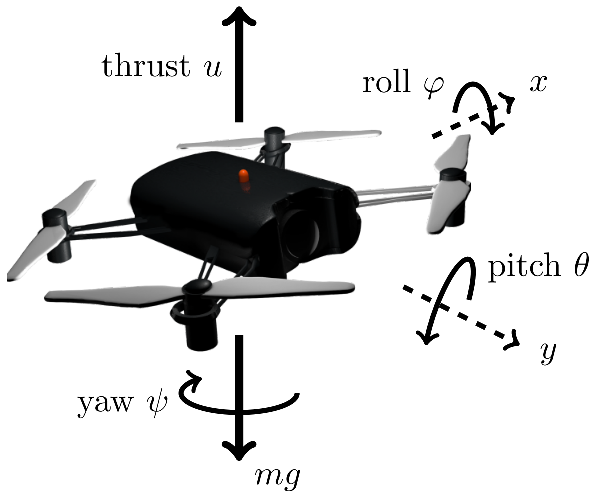


Figure 6.10: Quadcopter diagram [100].

the time discretization is too coarse (small n_t), then the model may simulate collisions solely due to inaccurate time integration. In validation, we use a fine time discretization and observe that the agents avoid colliding the obstacles and each other by observing that values for Q and W are exactly 0.

6.5.6 Quadcopter Experiment

In this experiment, a quadcopter, i.e., a multirotor helicopter, utilizes four rotors to propel itself across space from an initial state in the vicinity of $\boldsymbol{x}_0$ to target state $\boldsymbol{y}$. We choose values $\boldsymbol{x}_0=[-1.5, -1.5, -1.5, 0, \dots, 0]^\top \in \mathbb{R}^{12}$ and $\boldsymbol{y}=[2, 2, 2, 0, \dots, 0]^\top \in \mathbb{R}^{12}$. Denoting gravity as g, the acceleration of a quadcopter with mass m is given by

$$\begin{cases} \ddot{x} = \frac{u}{m}\big(\sin(\psi)\sin(\varphi) + \cos(\psi)\sin(\theta)\cos(\varphi)\big) \\ \ddot{y} = \frac{u}{m}\big(-\cos(\psi)\sin(\varphi) + \sin(\psi)\sin(\theta)\cos(\varphi)\big) \\ \ddot{z} = \frac{u}{m}\cos(\theta)\cos(\varphi) - g \end{cases} \tag{6.19}$$

where (x, y, z) is the spatial position of the quadcopter, and (ψ, θ, φ) is the angular orientation [17].

The dynamics can be written as the following first-order system

$$\dot{\boldsymbol{z}} = F(\boldsymbol{z}, \boldsymbol{u}, t) \implies \begin{cases} \dot{x} = v_x \\ \dot{y} = v_y \\ \dot{z} = v_z \\ \dot{\psi} = v_\psi \\ \dot{\theta} = v_\theta \\ \dot{\varphi} = v_\varphi \\ \dot{v}_x = \frac{u}{m} F_7(\psi, \theta, \varphi) \\ \dot{v}_y = \frac{u}{m} F_8(\psi, \theta, \varphi) \\ \dot{v}_z = \frac{u}{m} F_9(\theta, \varphi) - g \\ \dot{v}_\psi = \tau_\psi \\ \dot{v}_\theta = \tau_\theta \\ \dot{v}_\varphi = \tau_\varphi \end{cases} \tag{6.20}$$

where

$$\begin{cases} F_7(\psi, \theta, \varphi) & = \sin(\psi)\sin(\varphi) + \cos(\psi)\sin(\theta)\cos(\varphi), \\ F_8(\psi, \theta, \varphi) & = -\cos(\psi)\sin(\varphi) + \sin(\psi)\sin(\theta)\cos(\varphi), \\ F_9(\theta, \varphi) & = \cos(\theta)\cos(\varphi). \end{cases}$$

Here, $\boldsymbol{z} = [x\ y\ z\ \psi\ \theta\ \varphi\ v_x\ v_y\ v_z\ v_\psi\ v_\theta\ v_\varphi]^\top \in \mathbb{R}^{12}$ is our state and $\boldsymbol{u} = [u\ \tau_\psi\ \tau_\theta\ \tau_\varphi]^\top \in \mathbb{R}^4$ is our control, where u is the main thrust directed out of the bottom of the aircraft and the τ are the torques corresponding to the yaw φ, pitch ψ, and roll θ (Figure 6.10). For the energy term, we consider

$$E(\boldsymbol{z}, \boldsymbol{u}) = 2 + \|\boldsymbol{u}\|^2 = 2 + u^2 + \tau_\psi^2 + \tau_\theta^2 + \tau_\varphi^2. \tag{6.21}$$

For this problem, we have no obstacles nor other agents, so $L(\boldsymbol{z}, \boldsymbol{u}, t) = E(\boldsymbol{z}, \boldsymbol{u})$. We consider the Hamiltonian in (2.12) where $\boldsymbol{p} = [p_1\ p_2\ \dots\ p_{12}]^\top \in \mathbb{R}^{12}$. Noting the optimality conditions of (2.12) for the quadcopter problem are obtained by

$$
\begin{aligned}
-\nabla_{\boldsymbol{u}} E - \boldsymbol{p}^\top \nabla_{\boldsymbol{u}} F &= \mathbf{0} \\
\Rightarrow -2 \begin{bmatrix} u \\ \tau_\psi \\ \tau_\theta \\ \tau_\varphi \end{bmatrix} - \begin{bmatrix} p_7 \\ p_8 \\ p_9 \\ p_{10} \\ p_{11} \\ p_{12} \end{bmatrix}^\top \begin{bmatrix} F_7/m & 0 & 0 & 0 \\ F_8/m & 0 & 0 & 0 \\ F_9/m & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} &= \mathbf{0} \\
\Rightarrow -2 \begin{bmatrix} u \\ \tau_\psi \\ \tau_\theta \\ \tau_\varphi \end{bmatrix} - \begin{bmatrix} \frac{1}{m}(F_7 p_7 + F_8 p_8 + F_9 p_9) \\ p_{10} \\ p_{11} \\ p_{12} \end{bmatrix} &= \mathbf{0},
\end{aligned}
\tag{6.22}
$$

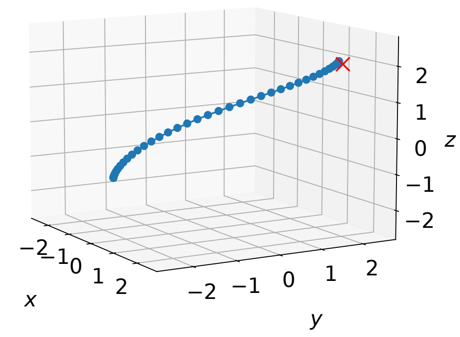
we can derive an expression for the controls as

$$
u = \frac{-1}{2m}(F_7 p_7 + F_8 p_8 + F_9 p_9), \quad \tau_\psi = \frac{-p_{10}}{2}, \quad \tau_\theta = \frac{-p_{11}}{2}, \quad \tau_\varphi = \frac{-p_{12}}{2}. \tag{6.23}
$$

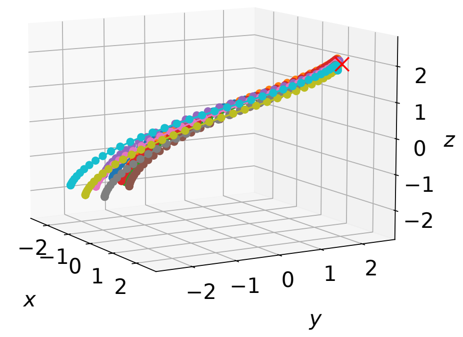
We therefore can compute the Hamiltonian

$$
\begin{aligned}
H(\boldsymbol{x}, \boldsymbol{p}, t) = -L(\boldsymbol{z}, \boldsymbol{u}, t) - [v_x \; v_y \; v_z] \begin{bmatrix} p_1 \\ p_2 \\ p_3 \end{bmatrix} - [v_\psi \; v_\theta \; v_\varphi] \begin{bmatrix} p_4 \\ p_5 \\ p_6 \end{bmatrix} \\
+ \frac{1}{2m^2}\big(p_7 F_7 + p_8 F_8 + p_9 F_9\big)^2 + p_9 g + \frac{1}{2}(p_{10}^2 + p_{11}^2 + p_{12}^2).
\end{aligned}
\tag{6.24}
$$

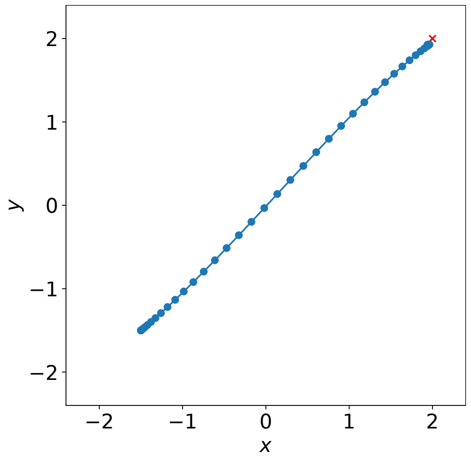
Finally, using (2.14) and (6.23), we compute the controls $\boldsymbol{u}$ using the NN (Fig-



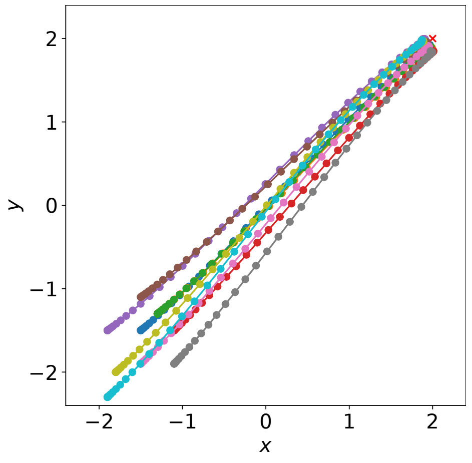
(a) Baseline trajectory of discrete optimization for the four controls on $n_t = 50$.



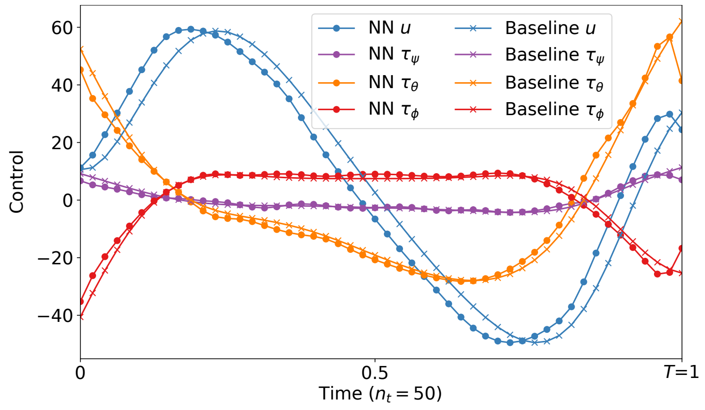
(b) NN trajectories, demonstrating the NN's usability for many initial conditions.



(c) Baseline trajectory.



(d) NN trajectories.

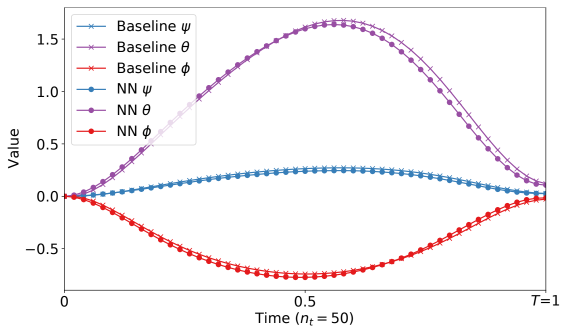


(e) Comparison of controls.

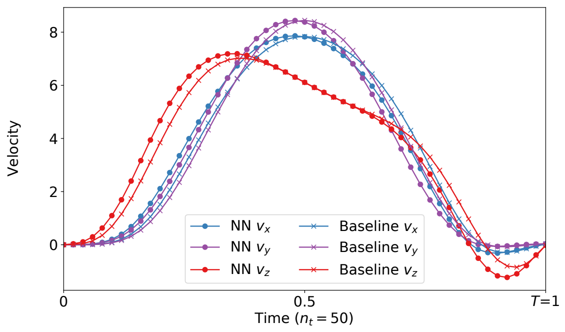
	$c_L + G$	c_L	G
Baseline	2,182.7	2,111.2	71.47
NN	2,184.9	2,122.0	62.90

(f) Comparison of loss values for single initial point $\boldsymbol{x}_0$.

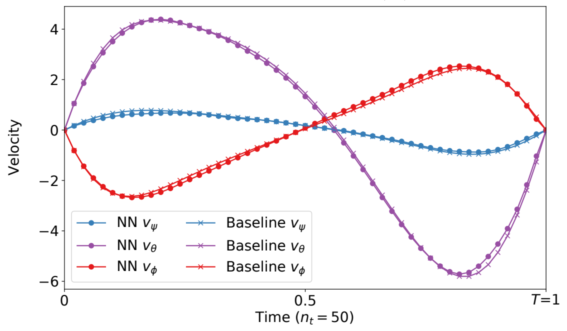
Figure 6.11: Quadcopter problem comparison with baseline.



(a) Angular values.



(b) Spatial position velocities.



(c) Angular velocities.

Figure 6.12: Quadcopter comparison of some states, supplementing Figure 6.11.

ure 6.11e) with

$$u = \frac{-1}{2m}\left(F_7 \frac{\partial \Phi}{\partial v_x} + F_8 \frac{\partial \Phi}{\partial v_y} + F_9 \frac{\partial \Phi}{\partial v_z}\right), \quad \tau_\psi = -\frac{1}{2}\frac{\partial \Phi}{\partial v_\psi}, \quad \tau_\theta = -\frac{1}{2}\frac{\partial \Phi}{\partial v_\theta}, \quad \tau_\varphi = -\frac{1}{2}\frac{\partial \Phi}{\partial v_\varphi}. \tag{6.25}$$

The quadcopter contains highly coupled 12-dimensional dynamics, which can lead to time-consuming model training despite its dimension and lack of obstacles and interactions (Table 6.1). We find that the HJB terminal conditions offered little impact as no obstacle or interaction costs interfered with the terminal cost.

The NN approach learns similar controls (Figure 6.11e) and states (Figure 6.12) as the baseline method. Both methods learn a similar flight path though the NN approach learns for many initial conditions (Figure 6.11). As with the corridor problem, the NN learned a solution with better terminal cost, but less optimal c_L than the baseline (Figure 6.11f).

Chapter 7

Summary

Neural ODEs draw from both recently developed data-driven machine learning and more theory-driven applied mathematics. Acknowledging that the development and hype of neural ODEs stems mostly from the machine learning viewpoint, we leverage optimal control theory to improve the design of neural ODEs. While hardware advances and the rise of GPUs have certainly assisted in NN development, we see design improvements as a similarly important avenue for cost-reduction.

In this dissertation, we presented methods for designing neural ODEs used in four tasks. These design choices fall under two categories: numerical treatment and formulation. In Chapter 2, we presented the mathematical background relevant for this work. We briefly described general neural networks, introduced neural ODEs, and reviewed relevant optimal control theory.

In Chapter 3, we motivate a time-series task where a continuous neural ODEs appear applicable. We use this simple task to present the numerical treatment benefits that arise from using the discretize-optimize approach instead of the optimize-discretize approach. These numerical benefits result in reduced training costs.

In Chapter 4, we shift to focusing on image classification tasks. Neural ODEs present little computational benefits over a more standard NN approach, e.g., the discrete ResNet. However, we design a neural ODE that is formulated to decouple the weights and layers of the neural network, resulting in a model with fewer parameters than comparable methods. We demonstrate the effectiveness of our low-parameterized neural ODE on a problem where training data is sparse—low-dose computed tomography for lung cancer detection.

For Chapter 5, we address the continuous normalizing flows problem. We first assess the numerical benefits of using the discretize-optimize approach introduced in Chapter 3. Using the value function, we furthermore formulate and regularize a neural ODE for the task. Numerically, in addition to using the discretize-optimize approach, we develop an efficient exact trace computation, derived from the analytical Laplacian of our neural network approximation of the value function. We observe drastic reductions in training and inference costs relative to the state-of-the-art.

We lastly turn to path-finding problems in Chapter 6. Using a similar formulation as in Chapter 5, we formulate a neural ODE that yields shock-robust and collision-avoidant paths for high-dimensional and centrally-controlled multi-agent problems. Numerically, we use Lagrangian coordinates and penalizers fashioned from the Hamilton-Jacobi-Bellman equations for efficient implementation.

This dissertation paves the way for future work in neural ODEs. Because neural ODEs present a single tool in the overlapping space of data-driven machine learning and traditional mathematical modeling, we feel this work is merely part of the beginning of a much larger movement[2, 36, 59, 66]. We encourage the further leveraging of optimal control and, more widely, theoretically sound mathematical approaches for the improvement of expensive neural networks.

Although we present some tasks (image classification and normalizing flows) on real-world data sets and applications, we leave the expansion of other tasks as future work. We used a simple time-series problem merely to introduce concepts; neural ODEs can necessarily apply to time-series data in practical applications. Furthermore, we simulate solutions for path-finding, but have interest in joining our work with engineering methods for real-world applications. Specifically, we have interest in deploying our model on physical quadcopters; we want to combine our work with approaches that allow the agents to be autonomous or respond in real-time to sensory information; we want to see our work in action.

Appendix A

Derivation of Adjoint Equations

We provide derivations behind the continuous backpropagation (3.4) and discrete backpropagations (3.3) and (3.5).

A.1 Continuous Adjoint

Consider the continuous ResNet optimization problem (2.19) subject to (2.8),

$$\min_{\boldsymbol{\theta}} \int_0^T L\big(\boldsymbol{z}(t), \boldsymbol{y}(t)\big)\mathrm{d}t \quad \text{s.t.} \quad \partial_t \boldsymbol{z}(t) = \mathbf{v}\big(\boldsymbol{\theta}(t), \boldsymbol{z}(t), t\big), \quad \boldsymbol{z}(0) = \boldsymbol{x} \tag{A.1}$$

where $\boldsymbol{y}$ is the ground truth and $\boldsymbol{z}$ depends on $\boldsymbol{\theta}$.

From here, we calculate the Lagrangian $\mathcal{G}$, with adjoint variable (and Lagrangian multiplier) $\boldsymbol{a}$

$$\begin{aligned}\mathcal{G}[\boldsymbol{\theta}, \boldsymbol{z}, \boldsymbol{a}] &= \int_0^T L\big(\boldsymbol{z}(t), \boldsymbol{y}(t)\big)\,\mathrm{d}t + \int_0^T \boldsymbol{a}(t)^\top \left[\mathbf{v}(\boldsymbol{z}(t), t; \boldsymbol{\theta}(t)) - \partial_t \boldsymbol{z}(t)\right]\mathrm{d}t. \\ &= \int_0^T L\big(\boldsymbol{z}(t), \boldsymbol{y}(t)\big) + \boldsymbol{a}(t)^\top \mathbf{v}\big(\boldsymbol{z}(t), t; \boldsymbol{\theta}(t)\big) - \boldsymbol{a}(t)^\top \partial_t \boldsymbol{z}(t)\,\mathrm{d}t.\end{aligned} \tag{A.2}$$

From optimization theory, we know that all the variations of $\mathcal{G}$ have to vanish at an optimal point. Here, we derive the strong form of the optimality system. Before doing so, we simplify the third term using integration by parts and simplify using $\boldsymbol{a}(0) = 0$ (due to the initial condition of the neural ODE):

$$\begin{aligned}\int_0^T \boldsymbol{a}(t)^\top \partial_t \boldsymbol{z}(t)\,\mathrm{d}t &= \boldsymbol{a}(t)^\top \boldsymbol{z}(t)\Big|_0^T - \int_0^T \partial_t \boldsymbol{a}(t)^\top \boldsymbol{z}(t)\,\mathrm{d}t \\ &= \boldsymbol{a}(T)^\top \boldsymbol{z}(T) - \int_0^T \partial_t \boldsymbol{a}(t)^\top \boldsymbol{z}(t)\,\mathrm{d}t.\end{aligned} \tag{A.3}$$

After substituting (A.3) into (A.2), the Lagrangian now becomes

$$\begin{aligned}\mathcal{G}[\boldsymbol{\theta}, \boldsymbol{z}, \boldsymbol{a}] = &\int_0^T L(\boldsymbol{z}(t), \boldsymbol{y}(t)) \;+\; \boldsymbol{a}(t)^\top \mathbf{v}\big(\boldsymbol{z}(t), t; \boldsymbol{\theta}(t)\big) + \partial_t \boldsymbol{a}(t)^\top \boldsymbol{z}(t)\,\mathrm{d}t \\ &- \boldsymbol{a}(T)^\top \boldsymbol{z}(T).\end{aligned} \tag{A.4}$$

For some time $t \in [0, T)$, the variational derivative of $\mathcal{G}$ with respect to $\boldsymbol{z}(t)$ is

$$\partial_{\boldsymbol{z}(t)} \mathcal{G}[\boldsymbol{\theta}, \boldsymbol{z}, \boldsymbol{a}] = \nabla_{\boldsymbol{z}} L(\boldsymbol{z}(t), \boldsymbol{y}(t)) \;+\; \nabla_{\boldsymbol{z}} \mathbf{v}\big(\boldsymbol{z}(t), t; \boldsymbol{\theta}(t)\big)\,\boldsymbol{a}(t) + \,\partial_t \boldsymbol{a}(t). \tag{A.5}$$

Setting this equal to zero gives the backward in time ODE

$$-\,\partial_t \boldsymbol{a}(t) = \nabla_{\boldsymbol{z}} L\big(\boldsymbol{z}(t), \boldsymbol{y}(t)\big) \;+\; \nabla_{\boldsymbol{z}} \mathbf{v}\big(\boldsymbol{z}(t), t; \boldsymbol{\theta}(t)\big)\,\boldsymbol{a}(t). \tag{A.6}$$

Using that this equation holds for all $t \in [0, T)$, we see that the variational derivative of $\mathcal{G}$ with respect to $\boldsymbol{z}(T)$ is

$$\nabla_{\boldsymbol{z}(T)} \mathcal{G}(\boldsymbol{\theta}, \boldsymbol{z}, \boldsymbol{a}) = \nabla_{\boldsymbol{z}} L\big(\boldsymbol{z}(T), \boldsymbol{y}(T)\big) - \boldsymbol{a}(T). \tag{A.7}$$

Setting this equal to zero gives the final time condition

$$\boldsymbol{a}(T) = \nabla_{\boldsymbol{z}} L\big(\boldsymbol{z}(T), \boldsymbol{y}(T)\big) \tag{A.8}$$

Finally, we note that, via chain rule, the variation of $\mathcal{J}$ with respect to $\boldsymbol{\theta}$ is

$$\nabla_{\boldsymbol{\theta}(t)}\mathcal{J}[\boldsymbol{\theta}] = \nabla_{\boldsymbol{\theta}}\mathbf{v}\big(\boldsymbol{z}(t), t; \boldsymbol{\theta}(t)\big)\, \boldsymbol{a}(t). \tag{A.9}$$

A.2 Discrete Adjoints

The backpropagations presented in (3.3) and (3.5) come from discretizing the continuous adjoint (3.4).

We consider a deep discrete ResNet (forward Euler scheme) and calculate the DO discretization for the backpropagation (3.3). The forward mode (2.7) follows

$$\begin{aligned}
\boldsymbol{a}_1 &= \boldsymbol{a}_0 + h\,\mathbf{v}\big(\boldsymbol{a}_0, t_0; \boldsymbol{\theta}^{(0)}\big)\\
\boldsymbol{a}_2 &= \boldsymbol{a}_1 + h\,\mathbf{v}\big(\boldsymbol{a}_1, t_1; \boldsymbol{\theta}^{(1)}\big)\\
&\vdots \qquad \vdots\\
\boldsymbol{a}_M &= \boldsymbol{a}_{M-1} + h\,\mathbf{v}\big(\boldsymbol{a}_{M-1}, t_{M-1}; \boldsymbol{\theta}^{(M-1)}\big).
\end{aligned}$$

From outputs $\boldsymbol{a}_1, \ldots, \boldsymbol{a}_M$, we calculate a discrete sum as the loss

$$J(\boldsymbol{\theta}) = \sum_{i=1}^{M} h\, L\big(\boldsymbol{z}^{(i)}, \boldsymbol{y}^{(i)}\big). \tag{A.10}$$

We then backpropagate to calculate the gradients to update model parameters. Using auxiliary term $\boldsymbol{a}$ to accumulate the gradient of J with respect to the states $\boldsymbol{z}$

as we step backwards in time,

$$\begin{aligned}
\boldsymbol{a}_M &= h\,\nabla_{\boldsymbol{z}^{(M)}} L\big(\boldsymbol{z}^{(M)}, \boldsymbol{y}^{(M)}\big) \\
\boldsymbol{a}_j &= h\,\nabla_{\boldsymbol{z}^{(j)}} L\big(\boldsymbol{z}^{(j)}, \boldsymbol{y}^{(j)}\big) + h\,\nabla_{\boldsymbol{z}^{(j)}} L\big(\boldsymbol{z}^{(j+1)}, \boldsymbol{y}^{(j+1)}\big) \\
&= h\,\nabla_{\boldsymbol{z}^{(j)}} L\big(\boldsymbol{z}^{(j)}, \boldsymbol{y}^{(j)}\big) + \nabla_{\boldsymbol{z}^{(j)}} \boldsymbol{z}^{(j+1)}\ \ h\,\nabla_{\boldsymbol{z}^{(j+1)}} L\big(\boldsymbol{z}^{(j+1)}, \boldsymbol{y}^{(j+1)}\big) \\
&= h\,\nabla_{\boldsymbol{z}^{(j)}} L\big(\boldsymbol{z}^{(j)}, \boldsymbol{y}^{(j)}\big) + \Big(I + h\nabla_{\boldsymbol{z}^{(j)}} \mathbf{v}\big(\boldsymbol{z}^{(j)}, t_j; \boldsymbol{\theta}^{(j)}\big)\Big)\, \boldsymbol{a}_{j+1} \\
&= \boldsymbol{a}_{j+1} + h\Big(\nabla_{\boldsymbol{z}} \mathbf{v}\big(\boldsymbol{z}^{(j)}, t_j; \boldsymbol{\theta}^{(j)}\big)\boldsymbol{a}_{j+1} + \nabla_{\boldsymbol{z}} L\big(\boldsymbol{z}^{(j)}, \boldsymbol{y}^{(j)}\big)\Big)
\end{aligned}$$

Using chain rule, we can now calculate the gradient with respect to the parameters

$$\begin{aligned}
\nabla_{\boldsymbol{\theta}^{(j)}} J(\boldsymbol{\theta}) &= \nabla_{\boldsymbol{\theta}}\, \boldsymbol{z}^{(j)}\ \nabla_{\boldsymbol{z}^{(j)}} J(\boldsymbol{\theta}) \\
&= h\nabla_{\boldsymbol{\theta}}\, \mathbf{v}\big(\boldsymbol{z}^{(j)}, t_j; \boldsymbol{\theta}^{(j)}\big)\ \boldsymbol{a}_j.
\end{aligned} \tag{A.11}$$

The OD discretization (3.5) follows the backward Euler scheme. It calculates the gradient at time t_{j+1} instead of at t_j as DO does.

Appendix B

Bootstrapping

Bootstrapping is a resampling method used when dealing with limited data [27]. We only use bootstrapping for generating confidence intervals (CIs) in plots. A CI is an error bound on the central tendency of a distribution of data. When bound by number of observations (applicable to us because each observation is expensive), bootstrapping follows the following process: draw fewer than n samples from the n observations you have, compute their central tendency, replace samples, repeat many times. Ultimately, one has a distribution of observations for the central tendency and can use that distribution to determine the percentage bounds on the error of the central tendency.

For the trace comparison between our exact trace and the Hutchinson's estimator (Figure 5.2), we run 20 replications and compute error bounds via bootstrapping. From the 20 runs, we sample with replacement 4,000 times with size 16. We compute the means for each of these 4,000 samplings and compute the 0.5 and 99.5 percentiles which we include in Figure 5.2 as a shaded area.

For Figure 6.5, we compare 1000 initial points $\boldsymbol{x}_0 + \xi$ at each magnitude $\|\xi\|$. Confidence intervals from 2.5 to 97.5% are computed via bootstrapping 10,000 subsamplings of size 500 from each set of 1000 points.

Bibliography

[1] Luigi Ambrosio, Nicola Gigli, and Giuseppe Savaré. *Gradient flows: in metric spaces and in the space of probability measures.* Springer Science & Business Media, 2008.

[2] Harbir Antil, Enrique Otarola, and Abner J Salgado. A space-time fractional optimal control problem: Analysis and discretization. *SIAM Journal on Control and Optimization*, 54(3):1295–1328, 2016.

[3] Diego Ardila, Atilla P Kiraly, Sujeeth Bharadwaj, Bokyung Choi, Joshua J Reicher, Lily Peng, Daniel Tse, Mozziyar Etemadi, Wenxing Ye, Greg Corrado, David P Naidich, and Shravya Shetty. End-to-end lung cancer screening with three-dimensional deep learning on low-dose chest computed tomography. *Nature Medicine*, 25(6):954–961, 2019.

[4] Uri M Ascher and Chen Greif. *A First Course in Numerical Methods.* SIAM, 2011.

[5] G. Avraham, Y. Zuo, and T. Drummond. Parallel optimal transport GAN. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 4406–4415, 2019.

[6] Haim Avron and Sivan Toledo. Randomized algorithms for estimating the trace of an implicit symmetric positive semi-definite matrix. *Journal of the ACM (JACM)*, 58(2):1–34, 2011.

[7] Richard Bellman. *Dynamic Programming.* Princeton University Press, Princeton, N. J., 1957.

[8] J.-D. Benamou, G. Carlier, and F. Santambrogio. Variational mean field games. In *Active Particles. Vol. 1. Advances in Theory, Models, and Applications*, Model. Simul. Sci. Eng. Technol., pages 141–171. Birkhäuser/Springer, Cham, 2017.

[9] Jean-David Benamou and Yann Brenier. A computational fluid mechanics solution to the Monge-Kantorovich mass transfer problem. *Numerische Mathematik*, 84(3):375–393, 2000.

[10] Dimitri P Bertsekas. *Reinforcement Learning and Optimal Control.* Athena Scientific Belmont, MA, 2019.

[11] Eli Boyarski, Ariel Felner, Roni Stern, Guni Sharon, Oded Betzalel, David Tolpin, and Eyal Shimony. ICBS: The improved conflict-based search algorithm for multi-agent pathfinding. In *Eighth Annual Symposium on Combinatorial Search.* Citeseer, 2015.

[12] Stephen Boyd, Stephen P Boyd, and Lieven Vandenberghe. *Convex Optimization.* Cambridge University Press, 2004.

[13] Johann Brehmer and Kyle Cranmer. Flows for simultaneous manifold learning and density estimation. *arXiv:2003.13913*, 2020.

[14] Johann Brehmer, Felix Kling, Irina Espejo, and Kyle Cranmer. Madminer: Machine learning-based inference for particle physics. *Computing and Software for Big Science*, 4(1):1–25, 2020.

[15] Steven L Brunton, Joshua L Proctor, and J Nathan Kutz. Discovering governing

equations from data by sparse identification of nonlinear dynamical systems. *PNAS*, 113(15):3932–3937, 2016.

[16] Steven L Brunton, Bernd R Noack, and Petros Koumoutsakos. Machine learning for fluid mechanics. *Annual Review of Fluid Mechanics*, 52, 2019.

[17] Luis Rodolfo García Carrillo, Alejandro Enrique Dzul López, Rogelio Lozano, and Claude Pégard. Modeling the quad-rotor mini-rotorcraft. In *Quad Rotorcraft Control*, pages 23–34. Springer, 2013.

[18] Changyou Chen, Chunyuan Li, Liqun Chen, Wenlin Wang, Yunchen Pu, and Lawrence Carin Duke. Continuous-time flows for efficient inference and density estimation. In *International Conference on Machine Learning (ICML)*, volume 80, pages 824–833, 2018.

[19] Tian Qi Chen, Yulia Rubanova, Jesse Bettencourt, and David K Duvenaud. Neural ordinary differential equations. In *Advances in Neural Information Processing Systems (NeurIPS)*, pages 6571–6583, 2018.

[20] Liron Cohen, Tansel Uras, T. K. Satish Kumar, Hong Xu, Nora Ayanian, and Sven Koenig. Improved solvers for bounded-suboptimal multi-agent path finding. In *International Joint Conference on Artificial Intelligence (IJCAI)*, pages 3067–3074, 2016.

[21] Michael G. Crandall and Pierre-Louis Lions. Viscosity solutions of Hamilton-Jacobi equations. *Transactions of the American Mathematical Society*, 277(1): 1–42, 1983. ISSN 0002-9947.

[22] Laurent Dinh, David Krueger, and Yoshua Bengio. NICE: non-linear independent components estimation. In Yoshua Bengio and Yann LeCun, editors, *International Conference on Learning Representations (ICLR)*, 2015.

[23] Laurent Dinh, Jascha Sohl-Dickstein, and Samy Bengio. Density estimation using real NVP. In *International Conference on Learning Representations (ICLR)*, 2017.

[24] Conor Durkan, Artur Bekasov, Iain Murray, and George Papamakarios. Neural spline flows. In *Advances in Neural Information Processing Systems (NeurIPS)*, pages 7509–7520, 2019.

[25] Weinan E. A proposal on machine learning via dynamical systems. *Communications in Mathematics and Statistics*, 5(1):1–11, 2017.

[26] Weinan E, Jiequn Han, and Arnulf Jentzen. Deep learning-based numerical methods for high-dimensional parabolic partial differential equations and backward stochastic differential equations. *Communications in Mathematics and Statistics*, 5(4):349–380, Nov 2017. ISSN 2194-671X.

[27] Bradley Efron and Robert J Tibshirani. *An Introduction to the Bootstrap*. CRC press, 1994.

[28] Michael Erdmann and Tomas Lozano-Perez. On multiple moving objects. *Algorithmica*, 2(1-4):477, 1987.

[29] Lawrence Evans. An introduction to mathematical optimal control theory version 0.2, 2013.

[30] Lawrence C Evans. Partial differential equations and Monge-Kantorovich mass transfer. *Current Developments in Mathematics*, 1997(1):65–126, 1997.

[31] Lawrence C Evans. *Partial Differential Equations*, volume 19. American Mathematical Society, 2010.

[32] Chris Finlay, Jörn-Henrik Jacobsen, Levon Nurbekyan, and Adam M Oberman.

How to train your neural ODE: the world of Jacobian and kinetic regularization. In *International Conference on Machine Learning (ICML)*, 2020.

[33] Wendell H. Fleming and H. Mete Soner. *Controlled Markov Processes and Viscosity Solutions*, volume 25 of *Stochastic Modelling and Applied Probability*. Springer, New York, second edition, 2006. ISBN 978-0387-260457; 0-387-26045-5.

[34] Wilfrid Gangbo and Robert J McCann. The geometry of optimal transportation. *Acta Mathematica*, 177(2):113–161, 1996.

[35] Mathieu Germain, Karol Gregor, Iain Murray, and Hugo Larochelle. MADE: Masked autoencoder for distribution estimation. In *International Conference on Machine Learning (ICML)*, pages 881–889, 2015.

[36] Amir Gholaminejad, Kurt Keutzer, and George Biros. ANODE: Unconditionally accurate memory-efficient gradients for neural ODEs. In *International Joint Conference on Artificial Intelligence (IJCAI)*, pages 730–736, 2019.

[37] Thomas R Gildea, Stacey DaCosta Byfield, D Kyle Hogarth, David S Wilson, and Curtis C Quinn. A retrospective analysis of delays in the diagnosis of lung cancer and associated costs. *ClinicoEconomics and Outcomes Research: CEOR*, 9:261, 2017.

[38] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *International Conference on Artificial Intelligence and Statistics*, pages 249–256, 2010.

[39] Will Grathwohl, Ricky TQ Chen, Jesse Betterncourt, Ilya Sutskever, and David Duvenaud. FFJORD: Free-form continuous dynamics for scalable reversible generative models. *International Conference on Learning Representations (ICLR)*, 2019.

[40] Arthur Gretton, Karsten M. Borgwardt, Malte J. Rasch, Bernhard Schölkopf, and Alexander Smola. A kernel two-sample test. *Journal of Machine Learning Research (JMLR)*, 13(25):723–773, 2012.

[41] Eldad Haber and Lars Ruthotto. Stable architectures for deep neural networks. *Inverse Problems*, 34(1):1–22, 2017.

[42] M.T. Hagan, H.B. Demuth, M.H. Beale, and O. De Jesús. *Neural Network Design*. Martin Hagan, 2014. ISBN 9780971732117.

[43] Jiequn Han and Weinan E. Deep learning approximation for stochastic control problems. *arXiv:1611.07422*, 2016.

[44] Jiequn Han, Arnulf Jentzen, and Weinan E. Solving high-dimensional partial differential equations using deep learning. *Proceedings of the National Academy of Sciences*, 115(34):8505–8510, Aug 2018. ISSN 1091-6490.

[45] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778, 2016.

[46] Catherine F Higham and Desmond J Higham. Deep learning: An introduction for applied mathematicians. *SIAM Review*, 61(4):860–891, 2019.

[47] Wolfgang Hönig, James A Preiss, TK Satish Kumar, Gaurav S Sukhatme, and Nora Ayanian. Trajectory planning for quadrotor swarms. *IEEE Transactions on Robotics*, 34(4):856–869, 2018.

[48] Chin-Wei Huang, David Krueger, Alexandre Lacoste, and Aaron Courville. Neural autoregressive flows. In *International Conference on Machine Learning (ICML)*, pages 2078–2087, 2018.

[49] Michael F Hutchinson. A stochastic estimator of the trace of the influence matrix for Laplacian smoothing splines. *Communications in Statistics-Simulation and Computation*, 19(2):433–450, 1990.

[50] Wei Kang and Lucas C Wilcox. Mitigating the curse of dimensionality: Sparse grid characteristics method for optimal feedback control and HJB equations. *Computational Optimization and Applications*, 68(2):289–315, 2017.

[51] Wei Kang, Qi Gong, and Tenavi Nakamura-Zimmerer. Algorithms of data development for deep learning and feedback design. *arXiv:1912.00492*, 2019.

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

[53] Durk P Kingma and Prafulla Dhariwal. Glow: Generative flow with invertible 1x1 convolutions. In *Advances in Neural Information Processing Systems (NeurIPS)*, pages 10215–10224, 2018.

[54] Durk P Kingma, Tim Salimans, Rafal Jozefowicz, Xi Chen, Ilya Sutskever, and Max Welling. Improved variational inference with inverse autoregressive flow. In *Advances in Neural Information Processing Systems (NeurIPS)*, pages 4743–4751, 2016.

[55] I. Kobyzev, S. Prince, and M. Brubaker. Normalizing flows: An introduction and review of current methods. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2020.

[56] Anders Krogh and John A Hertz. A simple weight decay can improve generalization. In *Advances in Neural Information Processing Systems (NeurIPS)*, pages 950–957, 1992.

[57] Karl Kunisch and Daniel Walter. Semiglobal optimal feedback stabilization of autonomous systems via deep neural network approximation. *arXiv:2002.08625*, 2020.

[58] Na Lei, Kehua Su, Li Cui, Shing-Tung Yau, and Xianfeng David Gu. A geometric view of optimal transportation and generative model. *Computer Aided Geometric Design*, 68:1–21, 2019.

[59] Qianxiao Li, Long Chen, Cheng Tai, and E Weinan. Maximum principle based algorithms for deep learning. *The Journal of Machine Learning Research (JMLR)*, 18(1):5998–6026, 2017.

[60] Yujia Li, Kevin Swersky, and Rich Zemel. Generative moment matching networks. In *International Conference on Machine Learning (ICML)*, pages 1718–1727, 2015.

[61] Yuxi Li. Deep reinforcement learning. *arXiv preprint arXiv:1810.06339*, 2018.

[62] Fangzhou Liao, Ming Liang, Zhe Li, Xiaolin Hu, and Sen Song. Evaluate the malignancy of pulmonary nodules using the 3-D deep leaky noisy-or network. *IEEE Transactions on Neural Networks and Learning Systems*, 30(11):3484–3495, 2019.

[63] Alex Tong Lin, Yat Tin Chow, and Stanley J Osher. A splitting method for overcoming the curse of dimensionality in Hamilton–Jacobi equations arising from nonlinear optimal control and differential games with applications to trajectory generation. *Communications in Mathematical Sciences*, 16(7):1933–1973, 2018.

[64] Alex Tong Lin, Samy Wu Fung, Wuchen Li, Levon Nurbekyan, and Stanley J Osher. APAC-Net: Alternating the population and agent control via two neural networks to solve high-dimensional stochastic mean field games. *arXiv:2002.10113*, 2020.

[65] T. Lin, P. Goyal, R. Girshick, K. He, and P. Dollr. Focal loss for dense object detection. In *2017 IEEE International Conference on Computer Vision (ICCV)*, pages 2999–3007, 2017. doi: 10.1109/ICCV.2017.324.

[66] Yiping Lu, Aoxiao Zhong, Quanzheng Li, and Bin Dong. Beyond finite layer neural networks: Bridging deep architectures and numerical differential equations. In *International Conference on Machine Learning (ICML)*, pages 3276–3285, 2018.

[67] Gaspard Monge. Mémoire sur la théorie des déblais et des remblais (Memoir on the theory of cutting and filling). *Histoire de l'Académie Royale des Sciences de Paris*, 1781.

[68] Thulasi Mylvaganam, Mario Sassano, and Alessandro Astolfi. A differential game approach to multi-agent collision avoidance. *IEEE Transactions on Automatic Control*, 62(8):4229–4235, 2017.

[69] Tenavi Nakamura-Zimmerer, Qi Gong, and Wei Kang. Adaptive deep learning for high dimensional Hamilton-Jacobi-Bellman equations. *arXiv:1907.05317*, 2019.

[70] National Cancer Institute (NCI). Surveillance, Epidemiology, and End Results Program (SEER), 2020. URL `https://seer.cancer.gov/statfacts/`.

[71] Radford M Neal. MCMC using Hamiltonian dynamics. *Handbook of Markov chain Monte Carlo*, 2(11):2, 2011.

[72] Jorge Nocedal and Stephen Wright. *Numerical Optimization*. Springer Science & Business Media, 2006.

[73] Frank Noé, Simon Olsson, Jonas Köhler, and Hao Wu. Boltzmann generators:

Sampling equilibrium states of many-body systems with deep learning. *Science*, 365, 2019.

[74] Levon Nurbekyan and Joao Saude. Fourier approximation methods for first-order nonlocal mean-field games. *arXiv:1811.01156*, 2018.

[75] Nikolas Nüsken and Lorenz Richter. Solving high-dimensional Hamilton-Jacobi-Bellman PDEs using neural networks: Perspectives from the theory of controlled diffusions and measures on path space. *arXiv:2005.05409*, 2020.

[76] Derek Onken and Lars Ruthotto. Discretize-optimize vs. optimize-discretize for time-series regression and continuous normalizing flows. *arXiv:2005.13420*, 2020.

[77] Derek Onken, Samy Wu Fung, Xingjian Li, and Lars Ruthotto. OT-Flow: Fast and accurate continuous normalizing flows via optimal transport. *arXiv:2006.00104*, 2020.

[78] Derek Onken, Levon Nurbekyan, Xingjian Li, Samy Wu Fung, Stanley Osher, and Lars Ruthotto. A neural network approach applied to multi-agent optimal control. *arXiv:2011.04757*, 2020.

[79] Stanley Osher and Chi-Wang Shu. High-order essentially nonoscillatory schemes for Hamilton–Jacobi equations. *SIAM Journal on Numerical Analysis*, 28(4): 907–922, 1991.

[80] George Papamakarios, Theo Pavlakou, and Iain Murray. Masked autoregressive flow for density estimation. In *Advances in Neural Information Processing Systems (NeurIPS)*, pages 2338–2347, 2017.

[81] George Papamakarios, Eric Nalisnick, Danilo Jimenez Rezende, Shakir Mo-

hamed, and Balaji Lakshminarayanan. Normalizing flows for probabilistic modeling and inference. *arXiv:1912.02762*, 2019.

[82] Gabriel Peyr and Marco Cuturi. Computational optimal transport. *Foundations and Trends in Machine Learning*, 11(5-6):355–607, 2019.

[83] L. S. Pontryagin, V. G. Boltyanskii, R. V. Gamkrelidze, and E. F. Mishchenko. *The Mathematical Theory of Optimal Processes.* Translated by K. N. Trirogoff; edited by L. W. Neustadt. Interscience Publishers John Wiley & Sons, Inc. New York-London, 1962.

[84] Christopher Rackauckas, Mike Innes, Yingbo Ma, Jesse Bettencourt, Lyndon White, and Vaibhav Dixit. DiffEqFlux.jl - A julia library for neural differential equations. *arXiv:1902.02376*, 2019.

[85] Maziar Raissi. Deep hidden physics models: Deep learning of nonlinear partial differential equations. *The Journal of Machine Learning Research (JMLR)*, 19 (1):932–955, 2018.

[86] Benjamin Recht. A tour of reinforcement learning: The view from continuous control. *Annual Review of Control, Robotics, and Autonomous Systems*, 2:253–279, 2019.

[87] Danilo Jimenez Rezende and Shakir Mohamed. Variational inference with normalizing flows. In *International Conference on Machine Learning (ICML)*, pages 1530–1538, 2015.

[88] Benjamin Rivière, Wolfgang Hönig, Yisong Yue, and Soon-Jo Chung. GLAS: Global-to-local safe autonomy synthesis for multi-robot motion planning with end-to-end learning. *IEEE Robotics and Automation Letters*, 5(3):4249–4256, 2020.

[89] Herbert Robbins and Sutton Monro. A stochastic approximation method. *The Annals of Mathematical Statistics*, 22(3):400–407, 09 1951. doi: 10.1214/aoms/1177729586.

[90] Frank Rosenblatt. The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, 65(6):386, 1958.

[91] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning representations by back-propagating errors. *Nature*, 1986.

[92] Lars Ruthotto and Eldad Haber. Deep neural networks motivated by partial differential equations. *Journal of Mathematical Imaging and Vision*, Sep 2019. ISSN 1573-7683. doi: 10.1007/s10851-019-00903-1.

[93] Lars Ruthotto, Stanley J. Osher, Wuchen Li, Levon Nurbekyan, and Samy Wu Fung. A machine learning framework for solving high-dimensional mean field game and mean field control problems. *Proceedings of the National Academy of Sciences (PNAS)*, 117(17):9183–9193, 2020.

[94] Tim Salimans, Diederik Kingma, and Max Welling. Markov chain Monte Carlo and variational inference: Bridging the gap. In *International Conference on Machine Learning (ICML)*, pages 1218–1226, 2015.

[95] Tim Salimans, Han Zhang, Alec Radford, and Dimitris N. Metaxas. Improving GANs using optimal transport. In *International Conference on Learning Representations (ICLR)*, 2018.

[96] Maziar Sanjabi, Jimmy Ba, Meisam Razaviyayn, and Jason D Lee. On the convergence and robustness of training GANs with regularized optimal transport. In *Advances in Neural Information Processing Systems (NeurIPS)*, pages 7091–7101, 2018.

[97] Guni Sharon, Roni Stern, Ariel Felner, and Nathan R Sturtevant. Conflict-based search for optimal multi-agent pathfinding. *Artificial Intelligence*, 219: 40–66, 2015.

[98] Guanya Shi, Wolfgang Hönig, Yisong Yue, and Soon-Jo Chung. Neural-swarm: Decentralized close-proximity multirotor control using learned interactions. *arXiv:2003.02992*, 2020.

[99] Theodore Shifrin. *Multivariable Mathematics: Linear Algebra, Multivariable Calculus, and Manifolds*. John Wiley & Sons, 2005.

[100] Kai Stachowiak. Drone image. URL `https://www.publicdomainpictures.net/en/view-image.php?image=288508&picture=drone`. Accessed March 23, 2021. CC0 Public Domain License.

[101] Trevor Standley and Richard Korf. Complete algorithms for cooperative pathfinding problems. In *International Joint Conference on Artificial Intelligence (IJCAI)*, pages 668–673, 2011.

[102] Roni Stern, Nathan Sturtevant, Ariel Felner, Sven Koenig, Hang Ma, Thayne Walker, Jiaoyang Li, Dor Atzmon, Liron Cohen, TK Kumar, et al. Multi-agent pathfinding: Definitions, variants, and benchmarks. *arXiv:1906.08291*, 2019.

[103] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, second edition, 2018.

[104] Johan Suykens, Herman Verrelst, and Joos Vandewalle. On-line learning Fokker-Planck machine. *Neural Processing Letters*, 7:81–89, 04 1998.

[105] Esteban G Tabak and Cristina V Turner. A family of nonparametric density estimation algorithms. *Communications on Pure and Applied Mathematics*, 66 (2):145–164, 2013.

[106] Akinori Tanaka. Discriminator optimal transport. In *Advances in Neural Information Processing Systems (NeurIPS)*, pages 6816–6826, 2019.

[107] National Lung Screening Trial Research Team. Results of initial low-dose computed tomographic screening for lung cancer. *New England Journal of Medicine (NEJM)*, 368(21):1980–1991, 2013.

[108] Lucas Theis, Aron van den Oord, and Matthias Bethge. A note on the evaluation of generative models. In *International Conference on Learning Representations (ICLR)*, 2016.

[109] Shashanka Ubaru, Jie Chen, and Yousef Saad. Fast estimation of tr(f(a)) via stochastic Lanczos quadrature. *SIAM Journal on Matrix Analysis and Applications*, 38(4):1075–1099, 2017.

[110] Cédric Villani. *Optimal Transport: Old and New*, volume 338. Springer Science & Business Media, 2008.

[111] Glenn Wagner and Howie Choset. M*: A complete multirobot path planning algorithm with performance bounds. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 3260–3267, 2011.

[112] Glenn Wagner and Howie Choset. Subdimensional expansion for multirobot path planning. *Artificial Intelligence*, 219:1–24, 2015.

[113] Antoine Wehenkel and Gilles Louppe. Unconstrained monotonic neural networks. In *Advances in Neural Information Processing Systems (NeurIPS)*, pages 1543–1553, 2019.

[114] E Weinan, Jiequn Han, and Qianxiao Li. A mean-field optimal control formulation of deep learning. *Research in the Mathematical Sciences*, 6(1):10, 2019.

[115] Max Welling and Yee W Teh. Bayesian learning via stochastic gradient Langevin dynamics. In *International Conference on Machine Learning (ICML)*, pages 681–688, 2011.

[116] Paul Werbos. Beyond regression: New tools for prediction and analysis in the behavior science. *Doctoral Dissertation, Harvard University*, 1974.

[117] Yuxin Wu and Kaiming He. Group normalization. In *European Conference on Computer Vision (ECCV)*, pages 3–19, 2018.

[118] Liu Yang and George Em Karniadakis. Potential flow generator with L_2 optimal transport regularity for generative models. *IEEE Transactions on Neural Networks and Learning Systems*, 2020.

[119] Linfeng Zhang, Lei Wang, et al. Monge-Ampère flow for generative modeling. *arXiv:1809.10188*, 2018.

Mathematical Symbols

a	dimension of the control
$\boldsymbol{a}$	auxillary variable
B	encoder
$\boldsymbol{b}$	bias component of neural network
C	normalizing flows loss function
d	dimension of state
D	decoder
$\boldsymbol{D}$	dense layer
$\mathbb{D}_{\mathrm{KL}}$	Kullback-Leibler divergence
E	energy term
$\boldsymbol{e}_i$	ith standard basis vector
$\boldsymbol{E}$	rectangular identity matrix
$\mathbb{E}$	expectation value
f	neural network/neural ODE
F	dynamics
g	gravity
G	terminal cost
h	stepsize
H	Hamiltonian
$\boldsymbol{I}$	identity matrix

J	discrete loss
$\boldsymbol{J}$	ResNet Jacobian
$\mathcal{J}$	loss functional
$\boldsymbol{K}$	neural network component
ℓ	log-determinant
L	Lagrangian
$\mathcal{L}$	loss functional
M	neural network depth (number of layers)
N	ResNet
$\mathcal{N}$	normalization layer
n_h	number of hidden layers
n_t	number of time steps
o	hidden dimension multiplier
$\mathcal{O}$	big O notation for time complexity
p	number of network parameters
P	pooling operator
$\boldsymbol{p}$	adjoint
q	dimension of a single agent's state
Q	obstacle terrain
$\boldsymbol{q}_i$	synthetic sample
$\boldsymbol{Q}$	set of synthetic samples
r	agent radius
R	HJB penalty term
$\mathcal{R}$	regularization functional
S	softmax function
$\boldsymbol{s}$	space-time vector in $\mathbb{R}^{d+1}$
T	final time

$\boldsymbol{u}$	controls
U	set of admissible controls
u	thrust
$\mathbf{v}$	neural network layer
$\boldsymbol{v}$	hidden state
v	velocity
$\boldsymbol{w}$	neural network component
W	interaction costs
$\boldsymbol{x}$	initial state
$\boldsymbol{X}$	training set
$\boldsymbol{y}$	ground truth, desired output
$\boldsymbol{z}$	state / hidden features
$\mathbb{Z}^+$	set of positive integers
α_i	weighting parameter
β	normalization scaling bias
β_i	weighting hyperparameter
γ	focal loss hyperparameter
$\boldsymbol{\epsilon}$	noise vector
ε	machine epsilon
ζ	normalization layer scaling weight
$\boldsymbol{\eta}$	number of training samples for each class
θ	quadcopter roll angle
$\boldsymbol{\theta}$	neural network parameters
ι	rank
κ	concatsquash layer
$\mu_{\mathcal{N}}$	normalization mean
ν	Gaussian density function

$\boldsymbol{\xi}$	shock
ρ	predicted density of the model
ρ_0	unknown, initial-time density
P_0	initial-time distribution
ρ_1	known, final-time density
P_1	final-time distribution (standard normal)
σ	activation function
$\sigma_{\mathcal{N}}$	normalization standard deviation
τ	torque
φ	quadcopter yaw angle
Φ	value function
χ	number of flow steps
ψ	quadcopter pitch angle
$\boldsymbol{\omega}$	random direction for derivative check
$\odot$	Hadamard Product

Index

activation function, 6, 7, 52, 67
AD, 8, 20, 21, 48, 85
ADAM, 2, 21, 24
automatic differentiation, *see* AD

backpropagation, 8
bootstrapping, 49, 114

change of variables, 44
CNF, 2, 3, 18, 44
CoD, 3, 15, 79, 80
continuous normalizing flow, *see* CNF
convolution, 32
curse of dimensionality, *see* CoD

dense layer, 6, 24, 30, 87
depth, 7, 87
derivative check, 25, 26
discretize-optimize, *see* DO
DO, 3, 20, 85
dynamic programming, 15

feedback form, 16
focal loss, 31, 39

ground truth, 9

Hadamard Product, 31, 52
Hamilton-Jacobi-Bellman, *see* HJB
HJB, 3, 4, 13, 18
Hutchinson's trace estimator, 49, 50, 69, 70
hyperparameter, 21

Jacobi identity, 44
Jacobi's formula, 44

kernel, 30, 33, 39, 72
KL, 45
Kullback-Leibler divergence, *see* KL

LDCT, 37, 39
low-dose computed tomography, *see* LDCT

National Lung Screening Trial, *see* NLST neural ODE, 4, 12

NLST, 39, 40

OD, 2, 20

optimal cost-to-go, *see* value function

optimize-discretize, *see* OD

overfit, 10, 71

penalizer, 4, 64, 92, 93

PMP, 13, 14

Pontryagin Maximum Principle, *see* PMP

pooling, 30

positive predictive value, *see* PPV

potential function, *see* value function

PPV, 38–41

quadcopter, 102

residual neural network, *see* ResNet

ResNet, 1, 11, 20, 35, 67, 88, 108

RK, 52

Runge-Kutta, *see* RK

SGD, 21

subgradient, 8

Tikhonov regularizaton, *see* weight decay

time-series, 18

value function, 2, 4, 14, 17

weight decay, 16, 64

width, 7