Generative Modeling with Neural Ordinary Differential Equations

by

Tim Dockhorn

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Applied Mathematics

Waterloo, Ontario, Canada, 2019

Author's Declaration

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Neural ordinary differential equations (NODEs) (Chen et al., 2018) are ordinary differential equations (ODEs) with their dynamics modeled by neural networks. Continuous normalizing flows (CNFs) (Chen et al., 2018; Grathwohl et al., 2018), a class of reversible generative models which builds on NODEs and uses an instantaneous counterpart of the change of variables formula (CVF), have recently proven to achieve state-of-the-art results on density estimation and variational inference tasks. In this thesis, we review key concepts that are important to understand NODEs and CNFs, ranging from numerical ODE solvers to generative models. We derive an explicit formulation of the adjoint sensitivity method for both NODEs and CNFs using a constrained optimization framework. Furthermore, we review several classes of NODEs and prove that a particular class of hypernetwork NODEs is a universal function approximator in the discretized state. Our numerical results suggest that the ODEs arising in CNFs do not need to be solved to high precision for training and we show that training of CNFs can be made more efficient by using a tolerance scheduler that exponentially reduces the ODE solver tolerances. Moreover, we quantify the discrepancy of the CVF and the discretized instantaneous CVF for two ODE solvers. Our hope in writing this thesis is to give a comprehensive and self-contained introduction to generative modeling (with neural ordinary differential equations) and to stimulate both theoretical as well as computational future work on NODEs and CNFs.

Acknowledgements

First and foremost, I want to thank my family for their everlasting support from afar. Without you, all of this would have never been possible.

I want to thank my supervisors Sander Rhebergen and Hans De Sterck for their guidance and for letting me explore my research interests. It has been a long journey from preconditioning for finite element methods to generative modeling with differential equations.

Thank you to Giang Tran and Jun Liu for reviewing this work.

Thanks to MFCF and Steve Weber for letting me test the teaching GPU servers.

I would like to thank Marko Ilievski, Priyank Jaini, and Yaoliang Yu for useful discussions.

Thanks to David Duvenaud for spotting some inaccuracies in the introduction.

To my office mates Jesse Legaspi, Christian Barna, Rishi Chakraborty, Josh Thompson, and Stan Zonov, thank you for making MC6407 the best office on campus.

Thank you to Ashwin Krishnan and the whole UWaterloo pickup group for kicking the ball with me.

Maddy, thank you for being the best study buddy and for always having my back.

Dedication

To my parents, Birgitt and Uli.

Table of Contents

List of Figures

List of Tables

Abbreviations

CNF Continuouos normalizing flow

CVF Change of variables formula

ELBO Evidence lower bound

FOBM (Fully observable) Boltzmann machine

GAN Generative adversarial network

IVP Initial value problem

KL Kullback–Leibler

MAP Maximum a posteriori

MLE Maximum likelihood estimation

NBE Number of backward function evaluations

NFE Number of forward function evaluations

NFEG Negative free energy

NLL Negative log-likelihood

NN Neural network

NODE Neural ordinary differential equation

NTE Number of total function evaluations

ODE Ordinary differential equation

RBM Restricted Boltzmann machine

VAE Variatonal autoencoder

VI Variational inference

List of Operators

$\circ$ Function composition

$\det$ Determinant

$D_{\mathrm{KL}}(\cdot \parallel \cdot)$ Kullback–Leibler divergence

$\mathbb{E}[\cdot]$ Expected value

∇ Gradient operator

$\odot$ Hadamard product

$\frac{\partial}{\partial t}$ Partial derivative (with respect to time t)

$\frac{d}{dt}$ Total derivative (with respect to time t)

tr Trace

List of Functions and Function Classes

C Cost function

Cov Covariance

exp Exponential function

log Natural logarithm

ReLU Rectifier function

σ Sigmoid function

SP Softplus function

tanh Hyperbolic tangent

$\mathcal{F}_2$ Class of two-layer neural networks

List of Probability Distributions

p_{data} Data generating distribution

p_{model} Model distribution

$\mathcal{N}(\cdot, \cdot)$ Normal distribution

$\mathcal{U}(\cdot)$ Uniform distribution

Chapter 1

Introduction

According to Arthur Samuel, machine learning is the field of study that gives computers the ability to learn without being explicitly programmed[1]. Machine learning can be divided into three main subareas: supervised learning, unsupervised learning, and reinforcement learning. The goal of supervised learning is to build a model that can predict the label of a data point given its features. In reinforcement learning, models learn to take suitable actions in order to maximize a cumulative reward. Given an unlabeled dataset, the goal of unsupervised learning is to learn a hidden underlying structure of the data.

In this thesis, we consider an important application of unsupervised learning called density estimation. Based on observations from a dataset, density estimation algorithms aim to construct an approximation p_{model} to the underlying distribution p_{data} that generated the data. One of the simplest approaches to approximate p_{data} is choosing the model distribution to be a mixture of normal distributions. The weights and parameters of the mixture model can then, for example, be learned using the expectation-maximization algorithm. Once the parameters are learned, samples of the model distribution p_{model} can be drawn by simply drawing samples from the individual mixture components. Models that learn a density function and have the ability to generate samples are generally referred to as generative models.

One of the drawbacks of mixture models is the assumption that each data point is explained by exactly one component. This assumption is especially restrictive for higher-dimensional

[1] This is rather the generally accepted gist of Samuel (1967) than an exact quote.

data, e.g., a mixture model cannot (easily)[2] incorporate the observation that neighboring pixels in images are likely to take similar values. This drawback has led to the development of models that learn distributed representations, e.g. the Boltzmann machine (Hinton and Sejnowski, 1983).

Boltzmann machines introduce unobserved variables in order to explain hidden causes of the data behavior and approximate p_{data} using a Boltzmann distribution. Salakhutdinov and Hinton (2009) showed that deep Boltzmann machines, Boltzmann machines that stack many layers of unobserved variables on top of each other, are particularly well-suited for learning an approximation to a high-dimensional data distribution. However, the necessity of Markov chains to generate samples of the Boltzmann distribution remains a major drawback of deep Boltzmann machines. Nevertheless, the idea of "depth" in generative models persisted.

The enormous amount of available data, the development of more advanced algorithms, and the increased processing power by graphical processing units are some of the many reasons that have led to the rise of deep learning[3] in the second decade of this century. It is not surprising that modern generative models also incorporate neural networks in their frameworks. Two of the most well-known classes of generative models that incorporate deep neural networks are generative adversarial networks (GANs) (Goodfellow et al., 2014) and variational autoencoders (VAEs) (Kingma and Welling, 2013; Rezende et al., 2014). Compared to Boltzmann machines, GANs and VAEs achieve "depth" without relying on Markov chains to generate samples.

Neither GANs nor VAEs provide an expression for the model density p_{model}, however, they have proven to be able to generate realistic samples that are similar to samples from the data generating distribution. This ability can be exploited in many areas of life, e.g., by assisting humans in creating artwork (Zhu et al., 2016; Brock et al., 2017) and building super-resolution images (Ledig et al., 2017) that can, for example, be used in movie production. Despite the success of GANs and VAEs, there is still a strong interest in generative models that have an explicit expression of their model density.

The two most prominent classes of modern generative models that have an explicit density function are autoregressive models (Germain et al., 2015; Van Oord et al., 2016) and

[2] A color image of size 64×64 would lead to a covariance matrix with around 7.5×10^7 parameters per mixture component.

[3] In this work, we refer to deep learning as using neural networks with many hidden layers as function approximators.

reversible generative models (Dinh et al., 2014, 2017; Papamakarios et al., 2017; Oliva et al., 2018; Kingma et al., 2016; Grathwohl et al., 2018; Kingma and Dhariwal, 2018). The former models the density function as a product of conditional distributions and the latter learns a transformation from a simple density into a complex one. One disadvantage of autoregressive models, when compared to reversible generative models, is that their computational cost to generate samples scales poorly with the dimensionality of the data.

In this thesis, we study continuous normalizing flows (CNFs) (Chen et al., 2018; Grathwohl et al., 2018), a reversible generative model that incorporates neural ordinary differential equations (NODEs) (Chen et al., 2018). NODEs are ordinary differential equations (ODEs) with their dynamics modeled by neural networks. Instead of using backpropagation, which is the standard algorithm to learn the parameters of a neural network, the parameters of the NODE dynamics are generally learned using the adjoint sensitivity method (Pontryagin et al., 1962). Formulating generative models with ODEs, one can take advantage of more than a century of development in numerical ODE solvers.

In this work, we discuss theoretical aspects about NODEs and CNFs. Furthermore, we investigate the effect of the NODE dynamics and the numerical ODE solver on the performance of CNFs. Numerical tests are carried out on (high-dimensional) density estimation, (low-dimensional) sample generation, and variational inference tasks.

1.1 Main Contributions

The main contribution of this thesis is to give a comprehensive and self-contained introduction to generative modeling (with neural ordinary differential equations) which is especially well-suited for applied mathematicians. Novel contributions of this thesis are the following:

1. We present a derivation of the adjoint sensitivity method for NODEs using a constrained optimization framework. We further extend this algorithm to CNFs.

2. We explain why linear and autonomous dynamics are not-well suited for NODEs and we prove that modeling the dynamics with a particular hypernetwork leads to universal function approximation in the discretized case.

3. Numerical results on density estimation and variational inference tasks suggest that the ODEs arising in CNFs do not need to be solved to high precision for training. Furthermore, we show that training of CNFs can be made more efficient by using a tolerance scheduler that exponentially reduces the ODE solver tolerances.

1.2 Outline

In Chapter 2, we provide a detailed overview on background that is relevant to this thesis. We review ordinary differential equations with a focus on numerical methods to solve them. We further lay out important concepts of probability theory and neural networks. We explain the difference between fully observable and latent variable generative models and discuss state-of-the-art generative models. We briefly review gradient based optimization.

Neural ordinary differential equations are the main subject of Chapter 3. We discuss choices for the dynamics of NODEs and for the algorithm to learn the parameters of NODEs. This chapter also contains a derivation of the adjoint sensitivity method for NODEs based on a constrained optimization approach.

In Chapter 4, we review reversible generative models. We summarize how to use the NODE framework for generative modeling resulting in continuous normalizing flows. Furthermore, this chapter contains a derivation of the adjoint sensitivity method for CNFs. Moreover, we quantify the discrepancy of the CVF and the discretized instantaneous CVF for two ODE solvers.

In Chapter 5, we investigate the effect of the NODE dynamics and the numerical ODE solver on the performance of CNFs. Numerical tests are carried out on density estimation for two tabular datasets, sample generation for a two-dimensional toy problem, and variational inference for an image dataset.

Chapter 2

Background

This chapter provides background knowledge for the subsequent chapters. It reviews some theoretical and numerical aspects of ordinary differential equations. Furthermore, we briefly discuss some important concepts from probability theory. We then discuss the basics of neural networks as well as their role in the success of deep learning. Subsequently, we review several classes of modern generative models and attempt to show their differences as well as their similarities. Lastly, we address gradient based optimization, the predominant class of optimization algorithms in modern machine learning.

2.1 Ordinary Differential Equations

Ordinary differential equations (ODEs) of order n are equations of the form

$$\boldsymbol{z}^{(n)} = f\left(\boldsymbol{z}^{(n-1)}, \dots, \boldsymbol{z}', \boldsymbol{z}, t\right), \tag{2.1}$$

where $\boldsymbol{z}^{(i)} = \frac{d^i \boldsymbol{z}}{dt^i}$ with $\boldsymbol{z}^{(i)} \in \mathbb{R}^d$ for all $i = 0, 1, \dots, n$. The dynamics f of Equation (2.1) is a function

$$f\colon \underbrace{\mathbb{R}^d \times \cdots \times \mathbb{R}^d}_{n \text{ times}} \times [t_0, T] \to \mathbb{R}^d,$$

with $t_0 < T$. In case that $d = 1$ and f is linear in $z^{(i)}$, i.e.,

$$f\left(z^{(n-1)}, \dots, z^{(1)} z, t\right) = a_{n-1}(t) z^{(n-1)} + \cdots + a_1(t) z^{(1)} + a_0(t) z + g(t),$$

we can write Equation (2.1) as a system of first order ODEs

$$\frac{d}{dt}\begin{bmatrix} z \\ z^{(1)} \\ \vdots \\ z^{(n-2)} \\ z^{(n-1)} \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 & 0 & \dots & 0 & 0 \\ 0 & 0 & 1 & 0 & \dots & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \dots & 0 & 1 \\ a_0(t) & a_1(t) & a_2(t) & a_3(t) & \dots & a_{n-2}(t) & a_{n-1}(t) \end{bmatrix} \begin{bmatrix} z \\ z^{(1)} \\ \vdots \\ z^{(n-2)} \\ z^{(n-1)} \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \\ g(t) \end{bmatrix}.$$

If we are given an ODE together with an initial condition $\boldsymbol{z}_0$, we refer to the problem of finding a solution that satisfies the differential equation with the particular initial condition as an initial value problem (IVP). An IVP for $n = 1$ is given as

$$\begin{aligned} \frac{d\boldsymbol{z}(t)}{dt} &= f(\boldsymbol{z}, t), \quad t \in [t_0, T], \\ \boldsymbol{z}(t_0) &= \boldsymbol{z}_0. \end{aligned} \tag{2.2}$$

We address the existence and uniqueness of IVP solutions briefly in the next section.

2.1.1 Existence and Uniqueness of Solutions to Initial Value Problems

For this work, we do not need much theory of ODEs, however, we need to understand the requirements on the dynamics f such that a unique solution to an IVP exists. The following theorem is sufficient for our needs:

Theorem 2.1.1 (Khalil (2002)) *Suppose that $f(\boldsymbol{z}, t)$ is piecewise continuous in t and*

$$\|f(\boldsymbol{z}_1, t) - f(\boldsymbol{z}_2, t)\| \leq L\|\boldsymbol{z}_1 - \boldsymbol{z}_2\|,$$

for some finite L, for all $\boldsymbol{z}_1, \boldsymbol{z}_2 \in \mathbb{R}^d$ and for all $t \in [t_0, T]$. Then, the IVP (2.2) has a unique solution for $t = [t_0, T]$.

We refer the reader to Khalil (2002) for a proof of Theorem 2.1.1 and more theory on ODEs.

2.1.2 Numerical Methods for Ordinary Differential Equations

This section is intended to give the reader an idea on how to solve ODEs numerically. For simplicity, we restrict ourselves to ODEs of first order, however, all ideas are easily transferable to n-th order ODEs.

Euler's method: Given the IVP

$$\begin{aligned} \frac{d\boldsymbol{z}(t)}{dt} &= f(\boldsymbol{z}, t), \quad t \in [t_0, T], \\ \boldsymbol{z}(t_0) &= \boldsymbol{z}_0, \end{aligned}$$

Euler's method approximates the solution of the IVP at time $t_0 + nh$ as

$$\boldsymbol{z}_n = \boldsymbol{z}_{n-1} + hf(\boldsymbol{z}_{n-1}, t_{n-1}) \quad \forall n = 1, \ldots, N,$$

for a pre-defined step size $h = (T - t_0)/N$. The local truncation error, the error made in a single step of the method, is $\mathcal{O}(h^2)$. The global truncation error, the accumulation of the local truncation error for multiple steps, is $\mathcal{O}(h)$, which is considered a slow convergence rate. Besides slow convergence, another potential problem of Euler's method is that it can be numerically unstable. Consider solving the IVP

$$\begin{aligned} \frac{dz(t)}{dt} &= -3z(t), \quad t \in [t_0, T], \\ z(t_0) &= 1, \end{aligned}$$

with step size $h = 1$, then $z_n = (-2)^n$ whereas the exact solution is $z(t) = \exp(-3(t - t_0))$. For this example, Euler's method is diverging[1]. This numerical instability does not occur for the backward Euler method

$$\boldsymbol{z}_n = \boldsymbol{z}_{n-1} + hf(\boldsymbol{z}_n, t_n) \quad \forall n = 1, \ldots, N,$$

which is unconditionally stable and has the same convergence rates as Euler's method. This stability comes at the cost of solving a possibly non-linear equation for every time step. For "simple" problems, however, a few iterations of a fixed point method might suffice to solve the equation sufficiently well. Even though the backward Euler method fixes the potential stability issues of Euler's method, we still need to deal with the problem of low-order convergence, which is addressed in the next paragraph.

Explicit Runge–Kutta methods: The family of explicit Runge–Kutta methods is of the form

$$\boldsymbol{z}_{n+1} = \boldsymbol{z}_n + h \sum_{i=1}^{s} b_i \boldsymbol{k}_i,$$

[1]Time steps $h < 2/3$ would lead to convergence.

p	1	2	3	4	5	6	7	8
min s	1	2	3	4	6	7	9	11

Table 2.1: Minimum number of stages s for explicit Runge–Kutta methods to achieve convergence rate p (Butcher, 2016).

where

$$\begin{aligned}
\boldsymbol{k}_1 &= f(\boldsymbol{z}_n, t_n), \\
\boldsymbol{k}_2 &= f(\boldsymbol{z}_n + h(a_{2,1}\boldsymbol{k}_1), t_n + c_2 h), \\
&\vdots \\
\boldsymbol{k}_s &= f(\boldsymbol{z}_n + h(a_{s,1}\boldsymbol{k}_1 + \cdots + a_{s,s-1}\boldsymbol{k}_{s-1}), t_n + c_s h).
\end{aligned}$$

The parameters of a Runge–Kutta method are generally chosen to achieve the highest convergence rate possible. It is still an open problem to find an expression for the minimum number of stages s needed for a Runge–Kutta method to achieve converge rates of arbitrary order p; some combinations that are known are listed in Table 2.1.

Runge–Kutta methods are well-suited for adaptive step size control. The idea is to choose the parameters to achieve a certain convergence rate p and modify the parameters $\boldsymbol{b}$, resulting in new parameters $\boldsymbol{b}^*$, such that we get another method with convergence rate $p-1$. Assuming that the step $\boldsymbol{z}_n$ is accepted, an approximation to the local truncation error of the latter method, otherwise referred to as the absolute error of the (adaptive) method, can then be computed as

$$e = \|\boldsymbol{y} - \boldsymbol{y}^*\| = h\left\|\sum_{i=1}^{s}(\boldsymbol{b}_i - \boldsymbol{b}_i^*)\boldsymbol{k}_i\right\|, \tag{2.3}$$

where $\boldsymbol{y}$ and $\boldsymbol{y}^*$ are the approximate solutions of the methods of order p and $p-1$, respectively. If e is smaller than a certain tolerance, we accept the step from the p-th order method and set $\boldsymbol{z}_{n+1} = \boldsymbol{y}$, otherwise the calculation is redone with a smaller time step.

Probably the simplest adaptive Runge–Kutta method can be obtained by combining Heun's method, a method of order two, with the explicit Euler method. Heun's method is a two-stage Runge–Kutta method with parameters $\boldsymbol{b}_{\text{Heun}} = [1/2, 1/2]^T$ and

$$\boldsymbol{k}_2^{\text{Heun}} = f\left(\boldsymbol{z}_n + h\boldsymbol{k}_1, t_n + h\right).$$

The explicit Euler method, on the other hand, is a one-stage Runge–Kutta method with $b_{\text{Euler}} = 1$. An adaptive version of Heun's method is then given by setting $\boldsymbol{b} = \boldsymbol{b}_{\text{Heun}}$ and $\boldsymbol{b}^* = [b_{\text{Euler}}, 0]^T$; we refer to this method as `ah2`. The absolute error of this method can then be computed as

$$e^{\texttt{ah2}} = \frac{h}{2} \left\| \boldsymbol{k}_2^{\text{Heun}} - \boldsymbol{k}_1 \right\|.$$

Note that `ah2` uses only two function evaluations per step.

One of the most widely used adaptive Runge–Kutta methods is the Dormand–Prince method (Dormand and Prince, 1980). The method is of order five (fourth-order method for error computation) and needs six function evaluations per step making use of the "first same as last" property (Hairer et al., 1991). The Dormand–Prince method is one of the most popular adaptive ODE solvers in MATLAB's ODE suite (Shampine and Reichelt, 1997). In MATLAB, the method is known as `ode45`. A modified version of the original method has been proposed by Shampine (1986); we refer to this method as `dopri5`.

Absolute and relative error: One of the problems of the absolute error (2.3) is that it is not scale-invariant. Consider a one-dimensional example where the absolute error tolerance is set to 10^{-5}. A step with $y = 10^{-7}$ and $y^* = 10^{-6}$ is accepted, whereas a step with $y = 1.0$ and $y^* = 1.0 + 10^{-5}$ would not be accepted. However, the ratios $(y^* - y)/y^*$ are 0.9 and 1/100001 for the former and the latter cases, respectively. To prohibit this behavior, we additionally track the relative error

$$\frac{\|\boldsymbol{y} - \boldsymbol{y}^*\|}{\max\left(\|\boldsymbol{y}\|, \|\boldsymbol{y}^*\|\right)}.$$

In practice, we use both the relative and the absolute error to determine whether or not we accept a step. For all our numerical experiments in Chapter 5, we accept a step if

$$\frac{e}{\texttt{abs\_tol} + \max\left(\|\boldsymbol{y}_n\|, \|\boldsymbol{y}_n^*\|\right) \texttt{rel\_tol}} \leq 1,$$

where `abs_tol` as well as `rel_tol` are user-specified tolerances and e is computed using Equation (2.3).

For more information on convergence and stability issues as well as adaptive step size control of numerical solvers, we refer the reader to the excellent textbook Butcher (2016).

2.2 Probability Theory

A (univariate) probability density function is an absolutely continuous non-negative function $p\colon \mathbb{R} \to \mathbb{R}$ that integrates to 1. A probability density function is associated with a random variable as it specifies with what probability the random variable takes values in a particular range. Let $\mathrm{x} \in \mathbb{R}$ be a continuous random variable and p_{x} its associated probability density function, then

$$\Pr[a \leq \mathrm{x} \leq b] = \int_a^b p_{\mathrm{x}}(\mathrm{x})\, d\mathrm{x},$$

where $-\infty \leq a \leq b \leq \infty$. The expected value of x can be computed as

$$\mathbb{E}_{p_{\mathrm{x}}}[\mathrm{x}] = \int_{-\infty}^{\infty} \mathrm{x} p_{\mathrm{x}}(\mathrm{x})\, d\mathrm{x}.$$

A probability distribution over two random variables $p_{\mathrm{x,y}}(\mathrm{x}, \mathrm{y})$ is called a joint probability distribution. The joint distribution can also be expressed as the product of the conditional distribution $p_{\mathrm{x|y}}(\mathrm{x} \mid \mathrm{y})$ and the marginal distribution $p_{\mathrm{y}}(\mathrm{y})$, i.e.,

$$p_{\mathrm{x,y}}(\mathrm{x}, \mathrm{y}) = p_{\mathrm{x|y}}(\mathrm{x} \mid \mathrm{y}) p_{\mathrm{y}}(\mathrm{y}),$$

or similarly

$$p_{\mathrm{x,y}}(\mathrm{x}, \mathrm{y}) = p_{\mathrm{y|x}}(\mathrm{y} \mid \mathrm{x}) p_{\mathrm{x}}(\mathrm{x}).$$

For ease of notation, we generally omit the subscript of a probability density function as its argument uniquely defines the function, e.g.,

$$p_{\mathrm{x,y}}(\mathrm{x}, \mathrm{y}) = p(\mathrm{x}, \mathrm{y}).$$

2.2.1 Bayesian Inference

Bayesian inference is a method for systemically updating one's belief for a hypothesis as more information becomes available. The famous Bayes' theorem

$$\Pr[A \mid B] = \frac{\Pr[B \mid A] \Pr[A]}{\Pr[B]},$$

where A and B are events with $\Pr[B] \neq 0$, is at the heart of Bayesian inference. In this thesis, we use Bayes' theorem to infer the parameters $\boldsymbol{\theta}$ of a (model) probability density function $p(\mathbf{x} \mid \boldsymbol{\theta})$. Let us define a prior distribution $p(\boldsymbol{\theta} \mid \alpha)$ of $\boldsymbol{\theta}$ parameterized by the hyperparameter α. Given a dataset $\mathbf{X} = \left\{\mathbf{x}^{(i)}\right\}_{i=1}^{m} \sim p_{\text{data}}$ from the data generating distribution, one can compute the posterior distribution of $\boldsymbol{\theta}$ (given α and $\mathbf{X}$) as

$$\begin{aligned} p(\boldsymbol{\theta} \mid \mathbf{X}, \alpha) &= \frac{p(\mathbf{X}, \alpha \mid \boldsymbol{\theta}) p(\boldsymbol{\theta})}{p(\mathbf{X}, \alpha)} \quad \text{(by Bayes' theorem)} \\ &= \frac{p(\mathbf{X} \mid \boldsymbol{\theta}, \alpha) p(\alpha \mid \boldsymbol{\theta}) p(\boldsymbol{\theta})}{p(\mathbf{X} \mid \alpha) p(\alpha)} \\ &= \frac{p(\mathbf{X} \mid \boldsymbol{\theta}, \alpha) p(\boldsymbol{\theta} \mid \alpha)}{p(\mathbf{X} \mid \alpha)} \quad \text{(by Bayes' theorem).} \end{aligned} \tag{2.4}$$

Assuming that $\mathbf{X}$ is independent and identically distributed[2], the likelihood of $\mathbf{X}$ can be computed as

$$p(\mathbf{X} \mid \boldsymbol{\theta}, \alpha) = \prod_{i=1}^{m} p\left(\mathbf{x}^{(i)} \mid \boldsymbol{\theta}, \alpha\right).$$

To obtain the marginal distribution, one has to compute the integral

$$p(\mathbf{X} \mid \alpha) = \int p(\mathbf{X} \mid \boldsymbol{\theta}, \alpha) p(\boldsymbol{\theta} \mid \alpha) \, d\boldsymbol{\theta},$$

which is intractable already for a few parameters. We will discuss remedies to this issue in Section 2.2.3. The distribution of a new datapoint $\tilde{\mathbf{x}}$, called the posterior predictive distribution, can be calculated as

$$p(\tilde{\mathbf{x}} \mid \mathbf{X}, \alpha) = \int p(\tilde{\mathbf{x}} \mid \boldsymbol{\theta}) p(\boldsymbol{\theta} \mid \mathbf{X}, \alpha) \, d\boldsymbol{\theta}. \tag{2.5}$$

An alternative to this fully Bayesian approach is to compute the point estimate $\boldsymbol{\theta}_{\text{MAP}}$ that maximizes the posterior distribution, i.e.,

$$\boldsymbol{\theta}_{\text{MAP}} = \arg\max_{\boldsymbol{\theta}} p(\boldsymbol{\theta} \mid \mathbf{X}, \alpha) = \arg\max_{\boldsymbol{\theta}} p(\mathbf{X} \mid \boldsymbol{\theta}, \alpha) p(\boldsymbol{\theta} \mid \alpha).$$

The point estimate $\boldsymbol{\theta}_{\text{MAP}}$ is called the maximum a posteriori (MAP) estimate. If one chooses the prior distribution $p(\boldsymbol{\theta} \mid \alpha)$ to be uniform, the MAP estimate simplifies to the maximum likelihood estimate (MLE)

$$\boldsymbol{\theta}_{\text{MLE}} = \arg\max_{\boldsymbol{\theta}} p(\mathbf{X} \mid \boldsymbol{\theta}, \alpha).$$

[2]In this work, we assume that all datasets are independent and identically distributed.

An approximation of Equation (2.5) can then be computed as

$$p(\tilde{\mathbf{x}} \mid \boldsymbol{\theta}_{\mathrm{MAP}}, \alpha),$$

or

$$p(\tilde{\mathbf{x}} \mid \boldsymbol{\theta}_{\mathrm{MLE}}, \alpha).$$

2.2.2 Divergence Measures

The field of information theory was originally proposed in Shannon (1948) to study the effect of noise on a general communication system. Throughout this work, we use divergence measures from information theory to measure the similarity of two probability density functions. A divergence on a space of probability density functions S is a function

$$D(\cdot \parallel \cdot)\colon S \times S \to \mathbb{R}_0^+$$

satisfying

1. $D(p \parallel q) \geq 0$,
2. $D(p \parallel q) = 0 \iff p = q$,

for all $p, q \in S$. An important class of divergences is the f-divergences

$$D_{\mathrm{f}}(p \parallel q) = \int p(\mathrm{x}) f\left(\frac{q(\mathrm{x})}{p(\mathrm{x})}\right) d\mathrm{x},$$

where $f(u)$ is convex on $u > 0$ and $f(1) = 0$. The choice $f(u) = -\log(u)$ leads to the Kullback–Leibler (KL) divergence

$$D_{\mathrm{KL}}(p \parallel q) = \int p(\mathrm{x}) \log\left(\frac{p(\mathrm{x})}{q(\mathrm{x})}\right) d\mathrm{x}.$$

We make use of the KL divergence in the next section and see it in many more places of this thesis.

2.2.3 Variational Inference

A problem of modern statistics is the approximation of complicated probability density functions (Blei et al., 2017). Variational inference (VI) is a method to approximate these problematic functions through optimization. We examine this procedure for the posterior distribution $p(\boldsymbol{\theta} \mid \mathbf{X})$ from Equation (2.4) (note that we omitted the condition on α for ease of notation). In VI, one specifies a family of candidate densities $\mathcal{Q}$ and tries to find $q^*(\boldsymbol{\phi}) \in \mathcal{Q}$ that is closest to a target density using some appropriate measure. Letting $p(\boldsymbol{\theta} \mid \mathbf{X})$ be the target density and measuring the densities using the KL divergence, we have

$$q^*(\boldsymbol{\phi}) = \underset{q(\boldsymbol{\phi}) \in \mathcal{Q}}{\arg\min}\, D_{\mathrm{KL}}(q(\boldsymbol{\phi}) \parallel p(\boldsymbol{\theta} \mid \mathbf{X})), \tag{2.6}$$

where we omitted the dependence of q on $\mathbf{X}$. The objective $D_{\mathrm{KL}}(q(\boldsymbol{\phi}) \parallel p(\boldsymbol{\theta} \mid \mathbf{X}))$ in Equation (2.6) is generally intractable as it involves the computation of the marginal $p(\mathbf{X})$. To see this, let us recall the definition of the KL divergence

$$\begin{aligned} D_{\mathrm{KL}}(q(\boldsymbol{\phi}) \parallel p(\boldsymbol{\theta} \mid \mathbf{X})) &= \mathbb{E}_q[\log q(\boldsymbol{\phi})] - \mathbb{E}_q[\log p(\boldsymbol{\theta} \mid \mathbf{X})] \\ &= \mathbb{E}_q[\log q(\boldsymbol{\phi})] - \mathbb{E}_q[\log p(\boldsymbol{\theta}, \mathbf{X})] + \mathbb{E}_q[\log p(\mathbf{X})] \\ &= \mathbb{E}_q[\log q(\boldsymbol{\phi})] - \mathbb{E}_q[\log p(\boldsymbol{\theta}, \mathbf{X})] + \log p(\mathbf{X}). \end{aligned}$$

The last equality reveals the dependence on the marginal.

The evidence lower bound: Since the KL divergence is intractable, one often maximizes the evidence lower bound (ELBO) instead of minimizing the KL divergence. The ELBO is the negative KL divergence plus $\log p(\mathbf{X})$, i.e.,

$$\mathrm{ELBO}(q) \coloneqq \log p(\mathbf{X}) - D_{\mathrm{KL}}(q(\boldsymbol{\phi}) \parallel p(\boldsymbol{\theta} \mid \mathbf{X})) = \mathbb{E}_q[\log p(\boldsymbol{\theta}, \mathbf{X})] - \mathbb{E}_q[\log q(\boldsymbol{\phi})]. \tag{2.7}$$

Rewriting Equation (2.7) helps us in understanding the optimal variational density

$$\begin{aligned} \mathrm{ELBO}(q) &= \mathbb{E}_q[\log p(\boldsymbol{\theta}, \mathbf{X})] - \mathbb{E}_q[\log q(\boldsymbol{\phi})] \\ &= \mathbb{E}_q[\log p(\mathbf{X} \mid \boldsymbol{\theta})] + \mathbb{E}_q[\log p(\boldsymbol{\theta})] - \mathbb{E}_q[\log q(\boldsymbol{\phi})] \\ &= \mathbb{E}_q[\log p(\mathbf{X} \mid \boldsymbol{\theta})] - D_{\mathrm{KL}}(q(\boldsymbol{\phi}) \parallel p(\boldsymbol{\theta})). \end{aligned}$$

From the last equation, we can see that q places its "mass" on parameters $\boldsymbol{\phi}$ that explain the data $\mathbf{X}$ well while simultaneously trying to be close to the prior $p(\boldsymbol{\theta})$. Variational inference will play an important role in Section 4.1.4. We refer the reader to Blei et al. (2017) for a thorough review on VI.

Name	$g(\boldsymbol{x})$ (acting elementwise)
Rectifier function	$\mathrm{ReLU}(\boldsymbol{x}) = \max(\boldsymbol{x}, 0)$
Sigmoid function	$\sigma(\boldsymbol{x}) = (1 + \exp(-\boldsymbol{x}))^{-1}$
Hyperbolic tangent	$\tanh(\boldsymbol{x})$
Softplus function [3]	$\mathrm{SP}(\boldsymbol{x}) = \log(1 + \exp \boldsymbol{x})$

Table 2.2: Common activation functions for neural networks.

2.3 Neural Networks

Neural networks (NNs) are a class of mathematical functions inspired by neural circuits in the brains of humans and animals. Neural networks are widely incorporated in machine learning algorithms as they are able to approximate a wide range of functions well; this result is further discussed in Section 2.3.3.

2.3.1 Feed-Forward Neural Networks

Feed-forward neural networks are an important subclass of neural networks. In the next two paragraphs, we examine the building blocks of (deep) feed-forward neural networks.

Two-layer neural networks: We first restrict ourselves to two-layer neural networks (without biases), mainly for ease of presentation. The class of two-layer feed-forward neural networks with d-dimensional input is defined as

$$\mathcal{F}_2(d_1, d_2, g) \coloneqq \left\{ f \colon \mathbb{R}^d \to \mathbb{R}^{d_2} \mid \boldsymbol{x} \mapsto \boldsymbol{W}_2 g(\boldsymbol{W}_1 \boldsymbol{x}), \boldsymbol{W}_1 \in \mathbb{R}^{d_1 \times d}, \boldsymbol{W}_2 \in \mathbb{R}^{d_2 \times d_1} \right\},$$

where the activation function g is applied elementwise; an example is visualized in Figure 2.1. Popular activation functions are listed in Table 2.2.

We can use a two-layer neural network for binary classification, with labels $l = \{+, -\}$, by computing the label of an instance $\boldsymbol{x}$ as

$$l(\boldsymbol{x}) = \begin{cases} + & \text{if } \sigma\left(f(\boldsymbol{x})\right) \geq 0 \\ - & \text{otherwise} \end{cases},$$

[3] The softplus function is a smooth approximation to the rectifier function.

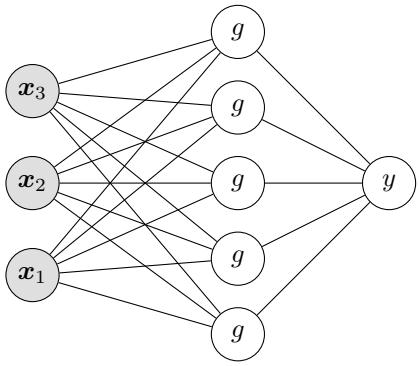


Figure 2.1: A two-layer neural network with $d_1 = 5$ and $d_2 = 1$; inputs to round nodes are summed up and the activation g is applied to the output of round circles with the label g.

with $f \in \mathcal{F}_2(d_1, d_2 = 1, g)$. A two-layer feed-forward neural network with biases is given as

$$\left\{ f \colon \mathbb{R}^d \to \mathbb{R}^{d_2} \mid \boldsymbol{x} \mapsto \boldsymbol{W}_2 g(\boldsymbol{W}_1 \boldsymbol{x} + \boldsymbol{b}_1) + \boldsymbol{b}_2, \boldsymbol{W}_1 \in \mathbb{R}^{d_1 \times d}, \boldsymbol{W}_2 \in \mathbb{R}^{d_2 \times d_1}, \boldsymbol{b}_1 \in \mathbb{R}^{d_1}, \boldsymbol{b}_2 \in \mathbb{R}^{d_2} \right\}.$$

Multilayer (feed-forward) neural networks We can easily extend two-layer feed-forward neural networks to L layers using the recursion

$$\begin{aligned} \boldsymbol{h}_1 &= g(\boldsymbol{W}_1 \boldsymbol{x} + \boldsymbol{b}_1), \\ \boldsymbol{h}_l &= g(\boldsymbol{W}_l \boldsymbol{h}_{l-1} + \boldsymbol{b}_l), \quad \forall l = 2, 3, \ldots, L-1, \\ \boldsymbol{h}_L &= \boldsymbol{W}_L \boldsymbol{h}_{L-1} + \boldsymbol{b}_L. \end{aligned}$$

2.3.2 The Backpropagation Algorithm

In this section, we review how to efficiently learn the parameters of a neural network given a dataset and a cost function C that is to be minimized. Backpropagation is arguably the predominant method in solving the aforementioned problem. The origins of this algorithm go back to control theory, and in particular to Kelley (1960) and Bryson (1961) who worked on flight performance optimization and multistage allocation processes, respectively. Dreyfus (1962) worked out a derivation of the backpropagation algorithm using the chain rule. To the best of our knowledge, the algorithm was first applied to learning the parameters of neural networks by Rumelhart (Rumelhart et al., 1988, 1985).

Backpropagation applied to neural networks: We want to find the parameters of a neural network with L layers that minimize the cost function for a dataset. Generally, the cost function is designed such that it can be written as a mean, i.e., given a dataset $\mathbf{X} = \left\{\mathbf{x}^{(i)}\right\}_{i=1}^{m}$ the cost can be computed as

$$C(\mathbf{X}) = \frac{1}{m}\sum_{i=1}^{m} \hat{C}\left(\boldsymbol{h}_L\left(\mathbf{x}^{(i)} \mid \boldsymbol{\theta}\right)\right),$$

where $\hat{C}\colon \mathbb{R}^{d_L} \to \mathbb{R}$ and $\boldsymbol{h}_L\left(\mathbf{x}^{(i)} \mid \boldsymbol{\theta}\right)$ is the output of the neural network given the input $\mathbf{x}^{(i)}$ and parameters $\boldsymbol{\theta}$. We can then learn the parameters of a neural network by gradient descent, i.e.,

$$\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k - \frac{\eta_k}{m}\sum_{i=1}^{m} \nabla_{\boldsymbol{\theta}} \hat{C}\left(\boldsymbol{h}_L\left(\mathbf{x}^{(i)} \mid \boldsymbol{\theta}_k\right)\right).$$

We refer the reader to Section 2.5 for more details on gradient descent and its computationally cheaper approximations. The backpropagation algorithm is concerned with efficiently computing $\nabla_{\boldsymbol{\theta}} C(\mathbf{X})$. For ease of notation, we only consider the case $\mathbf{X} = \mathbf{x}$ and abbreviate, with a slight abuse of notation, $\hat{C}(\boldsymbol{h}_L(\mathbf{x} \mid \boldsymbol{\theta}_k))$ as C.

Let us denote the values of layer l before activation given $\mathbf{x}$ by $\boldsymbol{a}^l$, i.e., $\boldsymbol{a}^l = \boldsymbol{W}_l \boldsymbol{h}_{l-1} + \boldsymbol{b}_l$. Furthermore, we denote the error in layer l, the partial derivative of C with respect to $\boldsymbol{a}^l$, as $\boldsymbol{\delta}^l$, i.e,

$$\boldsymbol{\delta}^l = \frac{\partial C}{\partial \boldsymbol{a}^l},$$

where the partial derivatives are computed elementwise. Note that for $l = L$ we have $\boldsymbol{a}^L = \boldsymbol{h}^L$ since there is no activation in the last layer. The error in layer l can be related to the error in layer $l+1$ by

$$\boldsymbol{\delta}^l = \left(\boldsymbol{W}_{l+1}^T \boldsymbol{\delta}^{l+1}\right) \odot g'(\boldsymbol{a}_l) \quad \forall l = L-1, \ldots, 1,$$

where g' is the derivative of g and the operator $\odot$ is the Hadamard product[4], defined as

$$(\boldsymbol{a} \odot \boldsymbol{b})_i = \boldsymbol{a}_i \boldsymbol{b}_i.$$

[4]This should not be confused with the Einstein summation convention.

The error in layer l can be related to the biases and weight matrices of layer l by

$$\frac{\partial C}{\partial \boldsymbol{b}_l} = \boldsymbol{\delta}^l,$$

and

$$\frac{\partial C}{\partial \boldsymbol{W}_l} = \boldsymbol{\delta}^l \left(\boldsymbol{h}_{l-1}\right)^T,$$

where $\boldsymbol{h}_0 = \mathbf{x}$. A derivation of these formulas can be found in Nielsen (2015). A concise summary of the backpropagation algorithm is given in Algorithm 1.

Algorithm 1 The backpropagation algorithm

for $l = 1, \ldots, L$ **do** ▷ Feed-forward part of algorithm
  $\boldsymbol{a}^l = \boldsymbol{W}_l \boldsymbol{h}_{l-1} + \boldsymbol{b}_l$ ▷ $\boldsymbol{h}_0 = \mathbf{x}$
  $\boldsymbol{h}_l = g\left(\boldsymbol{a}^l\right)$
end for
Compute $\boldsymbol{\delta}^L = \frac{\partial C}{\partial \boldsymbol{h}^L}$, $\frac{\partial C}{\partial \boldsymbol{b}_L} = \boldsymbol{\delta}^L$, and $\frac{\partial C}{\partial \boldsymbol{W}_L} = \boldsymbol{\delta}^L \left(\boldsymbol{h}_{L-1}\right)^T$.
for $l = L-1, \ldots, 1$ **do** ▷ Backpropagation of error
  $\boldsymbol{\delta}^l = \left(\boldsymbol{W}_{l+1}^T \boldsymbol{\delta}^{l+1}\right) \odot g'(\boldsymbol{a}_l)$
  $\frac{\partial C}{\partial \boldsymbol{b}_l} = \boldsymbol{\delta}^l$
  $\frac{\partial C}{\partial \boldsymbol{W}_l} = \boldsymbol{\delta}^l \left(\boldsymbol{h}_{l-1}\right)^T$
end for

2.3.3 Universal Approximation and Deep Learning

Much work has been done on proving that neural networks are universal function approximators, i.e., that they are dense in some function space. Cybenko (1989) proved that two-layer feed-forward neural networks $\mathcal{F}_2(d_1, d_2, g)$ can approximate any continuous function on compact subsets of $\mathbb{R}^d$ as $d_1 \to \infty$ when g is the sigmoid function (defined in Table 2.2). Hornik et al. (1989) showed that the restriction of the sigmoid function can be relaxed to squashing functions, non-decreasing functions ψ with the properties $\lim_{x \to -\infty} \psi(x) = 0$ and $\lim_{x \to \infty} \psi(x) = 1$. In recent years, the research community has focused on exploring the effectiveness of depth in neural networks. It has been shown that the function space of deep networks is larger than the one of shallow networks for the same number of parameters; we refer the reader to the following pool of references: Lin and Jegelka (2018); Eldan and Shamir (2016); Cohen et al. (2016); Telgarsky (2016); Liang and Srikant (2016); Rolnick and Tegmark (2017); Arora et al. (2018).

Deep residual networks: One neural network architecture that seems to profit substantially from depth is the residual neural network (He et al., 2016b), especially when applied to image classification problems (He et al., 2016a). Deep residual networks are functions of many stacked residual units. An example of a residual network is

$$\boldsymbol{x}_{l+1} = \boldsymbol{x}_l + r(\boldsymbol{x}_l, \boldsymbol{\theta}_l),$$

where a possible choice of r is $r \in \mathcal{F}_2(d_1, d_2, g)$ with $d_2 = d$. Stacking an infinite number of layers in a residual network with $\|r\| \to 0$ has led to the idea of neural ordinary differential equations, which we will explore in Chapter 3.

2.4 Generative Modeling

Given an unlabeled dataset $\mathbf{X} = \left\{\mathbf{x}^{(i)}\right\}_{i=1}^{m} \sim p_{\text{data}}$, the goal of unsupervised learning is to learn a hidden underlying structure of the data. We could for example be interested in clustering the data into groups or to reduce the dimensionality of the data. We refer the reader to the well-known textbook Bishop (2006) for these applications.

In this work, we focus on a class of unsupervised learning algorithms called generative modeling. In generative modeling, we want to learn the underlying probability distribution of the data and to be able to sample from the learned model distribution. Before we discuss some important generative models, we address the difference between fully observable and latent variable models.

2.4.1 Fully Observable and Latent Variable Models

Fully observable models learn the distribution of data by learning soft constraints, e.g. how the value of one dimension affects another dimension. Examples of fully observable models are Markov random fields (Kindermann, 1980), fully visible sigmoid belief networks (Frey et al., 1998), (fully observable) Boltzmann machines (Ackley et al., 1985), etc.

The (fully observable) Boltzmann machine (FOBM) can model multi-dimensional binary data $\mathbf{x} \in \{-1, 1\}^d$. The model follows, as could be guessed from the name, a Boltzmann distribution, i.e.,

$$p_{\text{FOBM}}(\mathbf{x}) = \frac{\exp(-E_{\text{FOBM}}(\mathbf{x}))}{Z},$$

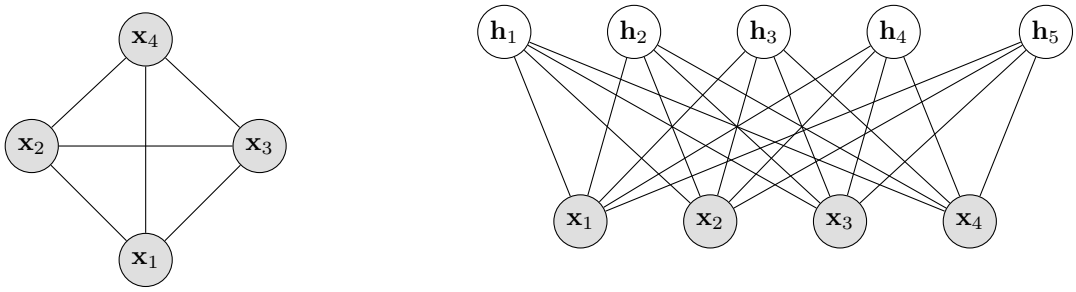


Figure 2.2: Fully observable Boltzmann machine (left) and restricted Boltzmann machine with five latent variables (right); observed variable nodes are grey and latent variable nodes are white.

where the energy function is defined as

$$E_{\text{FOBM}}(\mathbf{x}) = -\sum_j \boldsymbol{a}_j \mathbf{x}_j - \sum_{j \neq k} \boldsymbol{W}_{jk} \mathbf{x}_j \mathbf{x}_k,$$

and Z is a normalization constant. Given a dataset, the parameters of the FOBM, $\boldsymbol{\theta} = \{\boldsymbol{W}, \boldsymbol{a}\}$, can be learned by maximum likelihood estimation, i.e.,

$$\begin{aligned}\boldsymbol{\theta} &= \arg\max_{\boldsymbol{\theta}} \sum_{i=1}^{m} \log p_{\text{FOBM}}\left(\mathbf{x}^{(i)}\right) \\ &= \arg\max_{\boldsymbol{\theta}} -\sum_{i=1}^{m} E_{\text{FOBM}}\left(\mathbf{x}^{(i)}\right) - m \log Z.\end{aligned}$$

Solving this optimization problem is intractable since finding the normalization constant involves a sum of 2^d parameters, where d is the dimensionality of the data. As a remedy, one can approximate $\sum_{i=1}^{m} \log p_{\text{FOBM}}\left(\boldsymbol{x}^{(i)}\right)$ using Markov chains and learn the parameters by approximate maximum likelihood estimation.

Latent variable models, on the other hand, introduce unobserved random variables to explain hidden causes. Presumably the most well-known latent variable model for generative modeling is the restricted Boltzmann machine (RBM) (Smolensky, 1986). Introducing a set of unobserved binary variables $\mathbf{h}$, the RBM follows the distribution

$$p_{\text{RBM}}(\mathbf{x}, \mathbf{h}) = \frac{\exp(-E_{\text{RBM}}(\mathbf{x}, \mathbf{h}))}{Z},$$

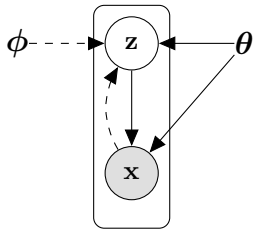


Figure 2.3: Generative model $p(\mathbf{x} \mid \mathbf{z}, \boldsymbol{\theta})p(\mathbf{z} \mid \boldsymbol{\theta})$ in solid lines and approximate posterior $q(\mathbf{z} \mid \mathbf{x}, \boldsymbol{\phi})$ is dashed lines (Kingma and Welling, 2013).

where the energy function is defined as

$$E_{\text{RBM}}(\mathbf{x}, \mathbf{h}) = -\sum_j \boldsymbol{a}_j \mathbf{x}_j - \sum_k \boldsymbol{b}_k \mathbf{h}_k - \sum_{j,k} \boldsymbol{W}_{jk} \mathbf{x}_j \mathbf{h}_k,$$

and Z is a normalization constant. In order to do inference, we need to compute the marginal distribution over the observable variables, i.e., sum the joint distribution over all configurations of the hidden variables

$$p_{\text{RBM}}(\boldsymbol{x}) = \frac{1}{Z} \sum_{\mathbf{h}} p_{\text{RBM}}(\mathbf{x}, \mathbf{h}).$$

We can again use approximate maximum likelihood estimation to learn the parameters of the RBM; thorough explanations of learning algorithms for restricted Boltzmann machines are given in Hinton et al. (2006) and Hinton (2012). Both the FOBM and the RBM are visualized in Figure 2.2.

2.4.2 Variational Autoencoders

Variational autoencoders (VAEs) (Kingma and Welling, 2013; Rezende et al., 2014) are state-of-the-art in generative modeling; they have, for example, been effectively applied to image, label and caption generation (Pu et al., 2016). Compared to other generative models, neural networks are easily incorporated in the VAE framework, letting VAEs take advantage of the recent breakthroughs of deep learning. Amortized variational inference (Rezende et al., 2014) is at the heart of VAE, and hence we spend some time on explaining this concept which is based on variational inference (see Section 2.2.3).

Amortized variational inference: In this paragraph, we consider a directed probabilistic model with latent variables $\mathbf{z}$, observable variables $\mathbf{x}$ and model parameters $\boldsymbol{\theta}$. We let $q(\mathbf{z} \mid \mathbf{x}, \boldsymbol{\phi})$ be a parameterized approximation to the posterior distribution of the latent variables $p(\mathbf{z} \mid \mathbf{x}, \boldsymbol{\theta})$. Both the graphical model as well as the approximate posterior are visualized in Figure 2.3. To perform inference on $\boldsymbol{\theta}$, we need to compute the marginal likelihood $p(\mathbf{x} \mid \boldsymbol{\theta})$, however, the marginalization over the latent variables $\mathbf{z}$ is generally intractable. We use the principles of VI to find a lower bound on the marginal log-likelihood (Rezende and Mohamed, 2015)

$$\begin{aligned}
\log p(\mathbf{x} \mid \boldsymbol{\theta}) &= \log \int p(\mathbf{x} \mid \mathbf{z}, \boldsymbol{\theta}) p(\mathbf{z} \mid \boldsymbol{\theta}) \, d\mathbf{z} \\
&= \log \int \frac{q(\mathbf{z} \mid \mathbf{x}, \boldsymbol{\phi})}{q(\mathbf{z} \mid \mathbf{x}, \boldsymbol{\phi})} p(\mathbf{x} \mid \mathbf{z}, \boldsymbol{\theta}) p(\mathbf{z} \mid \boldsymbol{\theta}) \, d\mathbf{z} \\
&\geq \int q(\mathbf{z} \mid \mathbf{x}, \boldsymbol{\phi}) \log \left(\frac{p(\mathbf{x} \mid \mathbf{z}, \boldsymbol{\theta}) p(\mathbf{z} \mid \boldsymbol{\theta})}{q(\mathbf{z} \mid \mathbf{x}, \boldsymbol{\phi})} \right) d\mathbf{z} \quad \text{(by Jensen's inequality)} \\
&= -D_{\mathrm{KL}}(q(\mathbf{z} \mid \mathbf{x}, \boldsymbol{\phi}) \parallel p(\mathbf{z} \mid \boldsymbol{\theta})) + \mathbb{E}_q[\log p(\mathbf{x} \mid \mathbf{z}, \boldsymbol{\theta})] = -\mathcal{F}(x),
\end{aligned}$$

where $-\mathcal{F}(x)$ is referred to as the negative free energy (NFEG). Note that equality is achieved if and only if $q(\mathbf{z} \mid \mathbf{x}, \boldsymbol{\phi}) = p(\mathbf{z} \mid \mathbf{x}, \boldsymbol{\theta})$. We then maximize the negative free energy with respect to both $\boldsymbol{\theta}$ and $\boldsymbol{\phi}$, i.e., given a dataset $\mathbf{X} = \left\{\mathbf{x}^{(i)}\right\}_{i=1}^{m}$, we compute the parameters as

$$\boldsymbol{\varphi}^* = \arg\max_{\boldsymbol{\varphi}} -\sum_{i=1}^{m} \mathcal{F}\left(\mathbf{x}^{(i)}\right),$$

where $\boldsymbol{\varphi} = \{\boldsymbol{\theta}, \boldsymbol{\phi}\}$. This optimization problem is generally solved using gradient based optimization. There are, however, two issues connected with this optimization problem (Rezende and Mohamed, 2015):

1. efficient computation of $\nabla_{\boldsymbol{\phi}} \mathbb{E}_q[\log p(\mathbf{x} \mid \mathbf{z}; \boldsymbol{\theta})]$ (naive Monte Carlo gradient estimation has very high variance (Paisley et al., 2012)) and

2. choosing an approximate posterior distribution q that is rich and yet makes the approximate maximum likelihood estimation computationally feasible.

When the fist problem is addressed with Monte Carlo gradient estimation (non-naive) and q is represented with an inference network, the above problem is referred to as amortized variational inference (Rezende and Mohamed, 2015).

Inference networks: An inference network learns to map from observations to latent variables using a global set of parameters. As an example, we could choose

$$q(\mathbf{z} \mid \mathbf{x}, \boldsymbol{\phi}) = \mathcal{N}\left(\mathbf{z} \mid \boldsymbol{\mu}, \operatorname{diag}\left(\boldsymbol{\sigma}^2\right)\right).$$

where $\boldsymbol{\mu}$ and $\boldsymbol{\sigma}$ are generally outputs of a feed-forward neural network, i.e., $\boldsymbol{\mu} = \boldsymbol{\mu}(\mathbf{x})$ and $\boldsymbol{\sigma} = \boldsymbol{\sigma}(\mathbf{x})$, and $\mathcal{N}$ is the normal distribution. Note that, in order to make inference possible, we would sample points from $\mathbf{z} \sim q$ as

$$\mathbf{z} = \boldsymbol{\mu} + \boldsymbol{\sigma} \odot \boldsymbol{\epsilon}, \quad \boldsymbol{\epsilon} \sim \mathcal{N}(\mathbf{0}, \boldsymbol{I}),$$

instead of sampling from $\mathcal{N}\left(\mathbf{z} \mid \boldsymbol{\mu}, \operatorname{diag}\left(\boldsymbol{\sigma}^2\right)\right)$ directly; this is referred to as the reparameterization trick (Kingma and Welling, 2013). In a variational autoencoder, one generally also computes the distribution parameters of the decoder $p(\mathbf{x} \mid \mathbf{z}, \boldsymbol{\theta})$ using a neural network. The prior distribution $p(\mathbf{z} \mid \boldsymbol{\theta})$ is chosen to be the standard normal distribution.

Generating samples: Samples can be generated from a VAE by first sampling $\mathbf{z}^* \sim \mathcal{N}(\mathbf{0}, \boldsymbol{I})$ and then sampling $\mathbf{x}^* \sim p(\mathbf{x} \mid \mathbf{z}^*, \boldsymbol{\theta})$.

We refer the reader to Doersch (2016) for a tutorial on VAEs.

2.4.3 Generative Adversarial Networks

Generative adversarial networks (GANs) (Goodfellow et al., 2014) are based on the idea of training two models simultaneously: a generative model G that maps input noise, $\mathbf{z} \sim p(\mathbf{z})$, from the latent space to the data space and a neural network D that distinguishes if a data point comes from the data generating distribution p_{data} or from G. The models can be trained by solving the following objective

$$\min_G \max_D V(D, G),$$

where

$$V(D, G) \coloneqq \mathbb{E}_{\mathbf{x} \sim p_{\text{data}}}[\log D(\mathbf{x})] + \mathbb{E}_{\mathbf{z} \sim p(\mathbf{z})}[\log\left(1 - D(G(\mathbf{z}))\right)],$$

which can be thought of as a minimax two-player game between the generator G and the neural network D. GANs lack an explicit expression for the density, however, they have proven to produce excellent results for synthesizing realistic images (Karras et al., 2017; Brock et al., 2018). Generative adversarial networks have gained much attention in the past few years, and we want to refer the reader to some interesting work on GANs: Arjovsky et al. (2017); Gulrajani et al. (2017); Salimans et al. (2016); Miyato et al. (2018).

2.4.4 Autoregressive Models

Autoregressive models (e.g. Hochreiter and Schmidhuber, 1997; Graves, 2013; Van Oord et al., 2016; Germain et al., 2015) are based on the observation that any d-dimensional joint distribution $p(\mathbf{x})$ can be factored into a product of conditional distributions (Uria et al., 2016), i.e.,

$$p(\mathbf{x}) = \prod_{i=1}^{d} p(\mathbf{x}_{o_i} \mid \mathbf{x}_{o_{<i}}),$$

using a permutation o of the integers $1, \ldots, d$. The vector $\mathbf{x}_{o_{<i}}$ contains the first $i-1$ dimensions in the ordering of o. An autoregressive model is then specified by modeling the d conditional distributions. One simple approach would be to model the conditionals as normal distributions, i.e.,

$$p(\mathbf{x}_{o_i} \mid \mathbf{x}_{o_{<i}}) = \mathcal{N}\left(\mathbf{x}_{o_i} \mid \mu_{o_i}, (\exp \alpha_{o_i})^2\right),$$

where $\mu_i = f_{\mu_i}(\mathbf{x}_{o_{<i}})$ and $\alpha_i = f_{\alpha_i}(\mathbf{x}_{o_{<i}})$; f_{μ_i} and f_{α_i} could for example be neural networks. One problem of autoregressive models is that their cost for generating samples depends on the dimensionality of the data d; this is especially problematic for large and high-quality images. Compared to VAEs, Boltzmann machines and GANs, autoregressive models have an explicit expression for the density function.

2.4.5 Taxonomy of Generative Models

We have discussed several generative models in the previous sections. To gain a better understanding of the variety of models, we classify them according to the expression of their probability distribution. As discussed in Section 2.4.3, GANs learn a probability distribution implicitly. Both VAEs and Boltzmann machines learn an approximation to the true probability distribution making use of variational inference and Markov chains (Brémaud, 2013), respectively. Autoregressive models are the only discussed models that have a (tractable) explicit expression for the density function. Another class of models that fits in this section is the class of reversible generative models, which we will discuss in Section 4.1. The taxonomy of the discussed generative models is visualized in Figure 2.4.

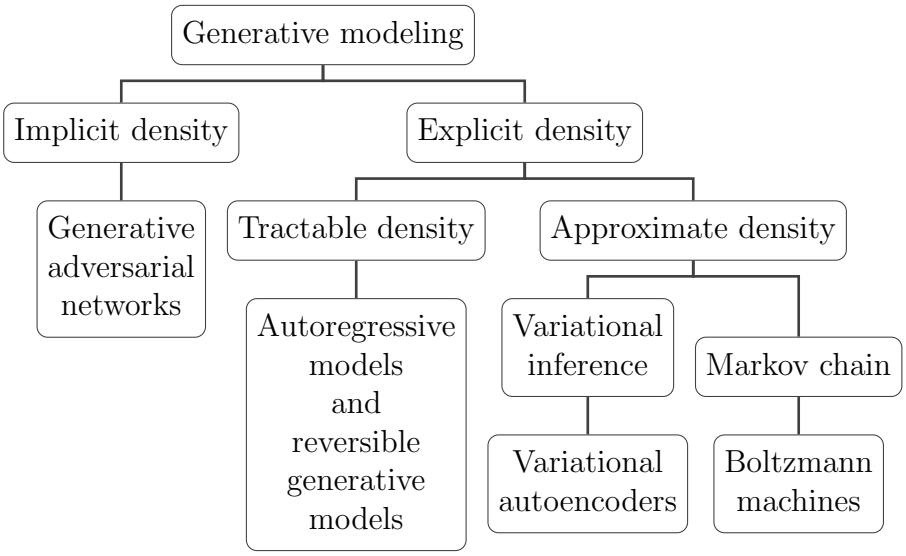


Figure 2.4: Classification of some generative models based on the expression of their probability distributions (Goodfellow, 2016).

2.5 Gradient Based Optimization

Gradient descent is an effective method to minimize a parameterized objective function $J(\boldsymbol{\theta})$. The parameters are updated in an iterative procedure of the form

$$\boldsymbol{\theta}_{i+1} = \boldsymbol{\theta}_i - \eta_i \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_i), \tag{2.8}$$

where $i = 0, 1, \ldots, K$ and η_i is called the learning rate. In generative modeling, the objective function is often of the form

$$J(\boldsymbol{\theta}) = \sum_{i=1}^{m} \log p\left(\mathbf{x}^{(i)} \mid \boldsymbol{\theta}\right),$$

where $\mathbf{X} = \left\{\mathbf{x}^{(i)}\right\}_{i=1}^{m} \sim p_{\text{data}}$. For large m, computing $\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$ is expensive. An unbiased estimator[5] of the former is

$$\nabla_{\boldsymbol{\theta}} J_{\text{stochastic}}(\boldsymbol{\theta}),$$

[5]An estimator is called unbiased if its expectation is equal to the quantity that is estimated.

where

$$J_{\text{stochastic}}(\boldsymbol{\theta}) = \log p\left(x^{(\mathrm{j})} \mid \boldsymbol{\theta}\right),$$

and $\mathrm{j} \sim \mathcal{U}(\{1, \ldots, m\})$ with $\mathcal{U}$ being the uniform distribution. When we use this unbiased estimator in the iterative procedure (2.8) instead of $\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$, we refer to it as stochastic gradient descent. While being a lot cheaper to compute, the estimator injects a lot of noise in the updates of $\boldsymbol{\theta}$. Hence, nowadays, it is common to use mini-batch gradient descent instead. The idea of minibatch gradient descent is to separate the datasets of m points in b batches of size $k = m/b$ and to use

$$\nabla_{\boldsymbol{\theta}} J_{\text{minibatch}}(\boldsymbol{\theta}) = \sum_{l=1}^{k} \nabla_{\boldsymbol{\theta}} \log p\left(x^{\left(o_l^{(r)}\right)} \mid \boldsymbol{\theta}\right), \tag{2.9}$$

as an unbiased estimator of $\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$. In Equation (2.9), o is an ordering of the m examples with $r \in \{1, \ldots, b\}$ such that

$$\left\{ \left\{ o_l^{(r)} \right\}_{l=1}^{k} \right\}_{r=1}^{b} = \{1, \ldots, m\}.$$

Note that the iterative procedure (2.8) does not converge if the learning rate is constant, and therefore we seek methods that automatically decrease the learning rate. One such method is the Adam optimizer (Kingma and Ba, 2014).

2.5.1 Adam

The Adam method estimates the first and second moment (mean and variance) of the gradient $\boldsymbol{g}_t$ as

$$\boldsymbol{m}_t = \beta_1 \boldsymbol{m}_{t-1} + (1 - \beta_1)\boldsymbol{g}_t, \tag{2.10}$$

and

$$\boldsymbol{v}_t = \beta_2 \boldsymbol{v}_{t-1} + (1 - \beta_2)\boldsymbol{g}_t \odot \boldsymbol{g}_t, \tag{2.11}$$

respectively. Here, $\boldsymbol{g}_t = \nabla_{\boldsymbol{\theta}} J_{\text{mini-batch}}(\boldsymbol{\theta})|_{\boldsymbol{\theta}=\boldsymbol{\theta}_t}$, β_1 as well as β_2 are pre-defined constants, and the moments are initialized as $\boldsymbol{m}_0 = \boldsymbol{v}_0 = \mathbf{0}$. Note that, due to the initialization, the

moments are biased towards $\mathbf{0}$. In order to correct this bias, Kingma and Ba (2014) propose to use the bias-corrected estimates

$$\hat{\boldsymbol{m}}_t = \frac{\boldsymbol{m}_t}{1-\beta_1^t},$$
$$\hat{\boldsymbol{v}}_t = \frac{\boldsymbol{v}_t}{1-\beta_2^t},$$

instead. The updates of the parameters in the Adam method are then computed as

$$\boldsymbol{\theta}_{t+1}^{\text{Adam}} = \boldsymbol{\theta}_t^{\text{Adam}} - \frac{\eta}{\sqrt{\hat{\boldsymbol{v}}_t} + \boldsymbol{\epsilon}} \odot \hat{\boldsymbol{m}}_t, \tag{2.12}$$

where the square root and the division are componentwise. Commonly used values for the constants are $\beta_1 = 0.9$, $\beta_2 = 0.999$, and $\boldsymbol{\epsilon} = \mathbf{10^{-8}}$ (Kingma and Ba, 2014).

Weight decay:

Weight decay can be seen as a regularization technique, sharing many similarities with ridge regression (Loshchilov and Hutter, 2017). For Adam, weight decay means adding a term $c_{\text{wd}}\boldsymbol{\theta}_t$ to $\boldsymbol{g}_t$ resulting in

$$\hat{\boldsymbol{g}}_t = \boldsymbol{g}_t + c_{\text{wd}}\boldsymbol{\theta}_t.$$

If we use Adam with weight decay, we replace $\boldsymbol{g}_t$ with $\hat{\boldsymbol{g}}_t$ in Equation (2.10) and Equation (2.11). We will use Adam (with weight decay) for all numerical experiments in Chapter 5.

Chapter 3

Neural Ordinary Differential Equations

In this chapter, we review neural ordinary differential equations (NODEs) and we show how they can be used for binary classification. We demonstrate how one can infer model parameters in the NODE framework from data and compare the adjoint sensitivity method to the backpropagation algorithm. Lastly, we examine several different classes of NODEs.

3.1 Introduction

As explained in Section 2.3.3, the hidden state in a residual network is iteratively modified as

$$\boldsymbol{z}_{l+1} = \boldsymbol{z}_l + r(\boldsymbol{z}_l, \boldsymbol{\theta}_l). \tag{3.1}$$

In the limit $\|r\| \to 0$, Equation (3.1) can be expressed as an ordinary differential equation

$$\begin{aligned} \frac{d\boldsymbol{z}(t)}{dt} &= r(\boldsymbol{z}(t), \boldsymbol{\theta}(t)), \quad t \in [0, T], \\ \boldsymbol{z}(0) &= \boldsymbol{z}_0, \end{aligned} \tag{3.2}$$

for some final time T. In a residual network r is generally just a two-layer neural network. The idea of NODEs (Chen et al., 2018) is to replace r in the IVP (3.2) with an arbitrary

neural network of the form $f(\boldsymbol{z}(t), t, \boldsymbol{\theta})$ resulting in the IVP

$$\begin{aligned} \frac{d\boldsymbol{z}(t)}{dt} &= f(\boldsymbol{z}(t), t, \boldsymbol{\theta}), \quad t \in [0, T], \\ \boldsymbol{z}(0) &= \boldsymbol{z}_0. \end{aligned} \tag{3.3}$$

Note that in Equation (3.3) all parameters $\boldsymbol{\theta}$ are independent of t and f depends explicitly on t. We will discuss commonly used structures of the dynamics f in Section 3.3. The connection between residual neural networks and ODEs has been widely discussed in recent years (e.g. Haber and Ruthotto, 2017; Lu et al., 2018; E, 2017).

Binary classification: We can use NODEs for binary classification by letting f be an arbitrary neural network of the form $f(\boldsymbol{z}(t), t, \boldsymbol{\theta})$ and computing the label of an observation $\boldsymbol{x}$ as

$$l(\boldsymbol{x}) = \begin{cases} + & \text{if } \hat{f}(f(\boldsymbol{z}(T))) \geq 0 \\ - & \text{otherwise} \end{cases},$$

where $\boldsymbol{z}(T)$ is the solution of the IVP (3.3) with $\boldsymbol{z}_0 = \boldsymbol{x}$ and $\hat{f} \in \mathcal{F}_2(d_1 = 1, d_2 = 1, g = \sigma)$; the class of two-layer neural networks $\mathcal{F}_2$ was defined in Section 2.3.1. Note that we can generally not compute $\boldsymbol{z}(T)$ analytically and instead need to use an approximation $\boldsymbol{z}_N$ computed by a numerical ODE solver. We encourage the interested reader to compare this approach to the standard neural network approach for binary classification in Section 2.3.1.

3.2 Learning of Neural Ordinary Differential Equations

Learning of NODEs means learning the parameters $\boldsymbol{\theta}$ of the dynamics $f(\boldsymbol{z}(t), t, \boldsymbol{\theta})$ in the IVP (3.3). This can be done by either using the adjoint sensitivity method (Pontryagin et al., 1962) and treating the ODE solvers as a black-box or by backpropagating through the discretized version of the ODE.

3.2.1 The Adjoint Sensitivity Method

In this section, we show how to compute the derivatives of a loss function with respect to the parameters $\boldsymbol{\theta}$ of the dynamics f using the adjoint sensitivity method (Pontryagin et al.,

1962). We derive this algorithm by considering a constrained optimization problem; we present this as an alternative to the derivation in Chen et al. (2018). We want to point out that LeCun et al. (1988) used a similar approach to derive the standard backpropagation algorithm as well as a generalization of the backpropagation algorithm to recurrent neural networks that are governed by a differential equation.

Consider the constrained optimization problem

$$\min_{\boldsymbol{\theta}} C(\boldsymbol{z}(T)) = \min_{\boldsymbol{\theta}} C\left(\boldsymbol{z}(0) + \int_0^T f(\boldsymbol{z}(t), t, \boldsymbol{\theta})\, dt\right),$$

subject to

$$\begin{aligned} h(\boldsymbol{z}(t), \dot{\boldsymbol{z}}(t), \boldsymbol{\theta}, t) = \frac{d\boldsymbol{z}(t)}{dt} - f(\boldsymbol{z}(t), t, \boldsymbol{\theta}) = \mathbf{0}, \\ g(\boldsymbol{z}(0)) = \boldsymbol{z}(0) - \boldsymbol{x} = \mathbf{0}, \end{aligned} \tag{3.4}$$

with C being a cost function. For ease of notation, we assume that $\boldsymbol{z} = z \in \mathbb{R}$, however, the general Algorithm 2 can be used for $d > 1$ as well. The constraints (3.4) are enforced by introducing the Lagrangian multipliers $\lambda = \lambda(t)$ and μ and consider the Lagrangian

$$\mathcal{L} = C(z(T)) + \int_0^T \lambda(t) h(t)\, dt + \mu g.$$

Note that since we compute $z(T)$ as $z(0) + \int_0^T f(z(t), t, \boldsymbol{\theta})\, dt$, the constraints (3.4) are actually already satisfied by construction. The reason to introduce them is to derive a tractable expression of $d_{\boldsymbol{\theta}} C(z(T))$ using $d_{\boldsymbol{\theta}} \mathcal{L}$.

The derivative of $\mathcal{L}$ with respect to $\boldsymbol{\theta}$ can be computed as

$$\begin{aligned} \frac{d}{d\boldsymbol{\theta}} \mathcal{L} &= \partial_{z(T)} C \frac{d}{d\boldsymbol{\theta}} \int_0^T f\, dt + \frac{d}{d\boldsymbol{\theta}} \int_0^T \lambda h\, dt \\ &= \partial_{z(T)} C \int_0^T \left[\partial_z f d_{\boldsymbol{\theta}} z + \partial_{\boldsymbol{\theta}} f\right] dt + \int_0^T \lambda \left[\partial_z h d_{\boldsymbol{\theta}} z + \partial_{\dot{z}} h d_{\boldsymbol{\theta}} \dot{z} + \partial_{\boldsymbol{\theta}} h\right] dt, \end{aligned}$$

where $\partial_r f$ and $d_r f$ are short-hand notations for $\partial f / \partial r$ and df/dr, respectively. To eliminate $d_{\boldsymbol{\theta}} \dot{z}$, we integrate by parts

$$\begin{aligned} \int_0^T \lambda \partial_{\dot{z}} h d_{\boldsymbol{\theta}} \dot{z}\, dt &= \lambda \partial_{\dot{z}} h d_{\boldsymbol{\theta}} z \Big|_0^T - \int_0^T \left[\dot{\lambda} \partial_{\dot{z}} h + \lambda d_t \partial_{\dot{z}} h\right] d_{\boldsymbol{\theta}} z\, dt \\ &= -\int_0^T \dot{\lambda} d_{\boldsymbol{\theta}} z\, dt, \end{aligned}$$

where we used that $\partial_{\dot{z}} h = 1$ as well as $d_{\boldsymbol{\theta}} z(0) = d_{\boldsymbol{\theta}} x = 0$ and where we chose $\lambda(T) = 0$. The derivative of the Lagrangian then simplifies to

$$\frac{d}{d\boldsymbol{\theta}}\mathcal{L} = \int_0^T \left[\left(\partial_{z(T)} C \partial_z f - \dot{\lambda} + \partial_z h \lambda \right) d_{\boldsymbol{\theta}} z + \partial_{z(T)} C \partial_{\boldsymbol{\theta}} f + \lambda \partial_{\boldsymbol{\theta}} h \right] dt.$$

To avoid having to compute $d_{\boldsymbol{\theta}} z$, we choose λ such that

$$\partial_{z(T)} C \partial_z f - \dot{\lambda} + \partial_z h \lambda = 0. \tag{3.5}$$

for all $0 < t < T$. By definition of h, Equation (3.5) is equivalent to

$$\partial_{z(T)} C \partial_z f - \dot{\lambda} - \partial_z f \lambda = 0. \tag{3.6}$$

Letting $\lambda^* = \partial_{z(T)} C - \lambda$, we can rewrite Equation (3.6) as

$$-\dot{\lambda}^* = \lambda^* \partial_z f,$$

with initial condition $\lambda^*(T) = \partial_{z(T)} C - \lambda(T) = \partial_{z(T)} C$. Using Equation (3.6), the derivative of the Lagrangian with respect to $\boldsymbol{\theta}$ further simplifies to

$$\begin{aligned}
\frac{d}{d\boldsymbol{\theta}}\mathcal{L} &= \int_0^T \left[\partial_{z(T)} C \partial_{\boldsymbol{\theta}} f + \lambda \partial_{\boldsymbol{\theta}} h \right] dt \\
&= \int_0^T \lambda^* \partial_{\boldsymbol{\theta}} f \, dt \\
&= -\int_T^0 \lambda^* \partial_{\boldsymbol{\theta}} f \, dt.
\end{aligned}$$

Hence, we can compute the derivative of the Lagrangian with respect to $\boldsymbol{\theta}$ by solving

$$\begin{bmatrix} z(0) - z(T) \\ \lambda^*(0) - \lambda^*(T) \\ d_{\boldsymbol{\theta}}\mathcal{L} \end{bmatrix} = \int_T^0 \begin{bmatrix} f \\ -\lambda^* \partial_z f \\ -\lambda^* \partial_{\boldsymbol{\theta}} f \end{bmatrix} dt, \tag{3.7}$$

with initial condition $\lambda^*(T) = \partial_{z(T)} C$ and $z(T) = z(0) + \int_0^T f \, dt$. Note that since the constraints are satisfied by construction, we have that $d_{\boldsymbol{\theta}}\mathcal{L} = d_{\boldsymbol{\theta}} C(z(T))$. In practice, we solve Equation (3.7) using a numerical ODE solver. The full adjoint sensitivity method for NODEs is summarized in Algorithm 2 where we use the abbreviation `odesolve` $\left(f, \boldsymbol{x}, \left[t^{(a)}, t^{(b)}\right]\right)$ to describe the approximate solution of an IVP at time $t^{(b)}$ with dynamics f and solution $\boldsymbol{x}$ at time $t^{(a)}$.

Algorithm 2 The adjoint sensitivity method for NODEs

1: $\boldsymbol{z}(T) = \texttt{odesolve}(f, \boldsymbol{x}, [0, T])$

2: Compute $\boldsymbol{\lambda}^*(T) = \partial_{\boldsymbol{z}(T)} C$

3: $$\begin{bmatrix} \boldsymbol{z}(0) \\ \boldsymbol{\lambda}^*(0) \\ d_{\boldsymbol{\theta}} C(\boldsymbol{z}(T)) \end{bmatrix} = \texttt{odesolve}\left(\begin{bmatrix} f \\ -(\partial_{\boldsymbol{z}} f)^T \boldsymbol{\lambda}^* \\ -(\partial_{\boldsymbol{\theta}} f)^T \boldsymbol{\lambda}^* \end{bmatrix}, \begin{bmatrix} \boldsymbol{z}(T) \\ \boldsymbol{\lambda}^*(T) \\ \mathbf{0} \end{bmatrix}, [T, 0] \right)$$

3.2.2 Backpropagation Through Ordinary Differential Equation Solvers

Instead of using the adjoint sensitivity method of Section 3.2.1, one can also learn the parameters $\boldsymbol{\theta}$ of the dynamics $f(\boldsymbol{z}(t), t, \boldsymbol{\theta})$ using standard backpropagation through the discretized version of the NODE. This approach is based on the observation that a discretized NODE is itself a neural network. We demonstrate this procedure for the Euler method and the midpoint method.

Euler's network: Applying Euler's method with N steps to the IVP (3.3), i.e.,

$$\boldsymbol{z}_n = \boldsymbol{z}_{n-1} + h f(\boldsymbol{z}_{n-1}, t_{n-1}, \boldsymbol{\theta}), \tag{3.8}$$

for all $n = 1, \ldots, N$ and $h = T/N$, results in a "residual-like" neural network of the form

$$\mathcal{F}_{\text{Euler}}(N, f) \coloneqq \left\{ f_{\text{Euler}} \colon \mathbb{R}^d \to \mathbb{R}^d \mid \boldsymbol{x} \mapsto \left(\boldsymbol{I} + h \hat{f}_{t_{N-1}, \boldsymbol{\theta}} \right) \circ \cdots \circ \left(\boldsymbol{I} + h \hat{f}_{t_0, \boldsymbol{\theta}} \right) (\boldsymbol{z}_0) \right\}.$$

Here we used short-hand notation $\hat{f}_{t_n, \boldsymbol{\theta}}(\boldsymbol{x}) \coloneqq f(\boldsymbol{x}, t_n, \boldsymbol{\theta})$ and $(f \circ g)(\boldsymbol{x}) \coloneqq f(g(\boldsymbol{x}))$. The relation between Euler's network and a residual network becomes clearer when we consider, for example, a residual network with two layers and Euler's network with two steps, i.e., $N = 2$:

- Residual network: $\boldsymbol{z}_2 = \boldsymbol{z}_1 + r(\boldsymbol{z}_1, \boldsymbol{\theta}_1) = \boldsymbol{z}_0 + r(\boldsymbol{z}_0, \boldsymbol{\theta}_0) + r(\boldsymbol{z}_0 + r(\boldsymbol{z}_0, \boldsymbol{\theta}_0), \boldsymbol{\theta}_1)$,
- Euler's network: $\boldsymbol{z}_2 = \boldsymbol{z}_1 + h f(\boldsymbol{z}_1, t_1, \boldsymbol{\theta}) = \boldsymbol{z}_0 + h f(\boldsymbol{z}_0, t_0, \boldsymbol{\theta}) + h f(\boldsymbol{z}_0 + h f(\boldsymbol{z}_0, t_0, \boldsymbol{\theta}), t_1, \boldsymbol{\theta})$.

Note that all dynamics in Euler's network share the global set of parameters $\boldsymbol{\theta}$, while the residual network has a different set of parameters for each layer. We now derive a

backpropagation algorithm for Euler's network. Let us denote the error at time step $\boldsymbol{z}_n$, the partial derivative of a cost function C with respect to $\boldsymbol{z}_n$, as $\boldsymbol{\delta}^n$, i.e.,

$$\boldsymbol{\delta}^n = \frac{\partial C}{\partial \boldsymbol{z}_n}$$

We first compute the approximate solution $\boldsymbol{z}_n$ at every time step t_n and store it in memory. Next, we compute the error at time step N as $\boldsymbol{\delta}^N = \partial C / \partial \boldsymbol{z}_N$. The error at time step t_n is related to the error at time step t_{n+1} by

$$\boldsymbol{\delta}^n = \left(\boldsymbol{I} + h \left(\frac{\partial f(\boldsymbol{z}_n, t_n, \boldsymbol{\theta})}{\partial \boldsymbol{z}_n} \right)^T \right) \boldsymbol{\delta}^{n+1}. \tag{3.9}$$

The derivative of the cost function C with respect to the parameters of the dynamics f can then be computed as

$$\frac{\partial C}{\partial \boldsymbol{\theta}} = \sum_{n=0}^{N-1} h \left(\frac{\partial f(\boldsymbol{z}_n, t_n, \boldsymbol{\theta})}{\partial \boldsymbol{\theta}} \right)^T \boldsymbol{\delta}^{n+1}. \tag{3.10}$$

Proofs of Equation (3.9) and Equation (3.10) can be found in Appendix A. The Euler backpropagation algorithm is summarized in Algorithm 3.

Algorithm 3 The Euler backpropagation algorithm

1: **for** $n = 1, \ldots, N$ **do**
2: $\quad \boldsymbol{z}_n = \boldsymbol{z}_{n-1} + h f(\boldsymbol{z}_{n-1}, t_{n-1}, \boldsymbol{\theta})$
3: **end for**
4: Compute $\boldsymbol{\delta}^N = \frac{\partial C}{\partial \boldsymbol{z}_N}$
5: **for** $n = N-1, \ldots, 1$ **do**
6: $\quad \boldsymbol{\delta}^n = \left(I + h \left(\frac{\partial f(\boldsymbol{z}_n, t_n, \boldsymbol{\theta})}{\partial \boldsymbol{z}_n} \right)^T \right) \boldsymbol{\delta}^{n+1}$
7: **end for**
8: $\frac{\partial C}{\partial \boldsymbol{\theta}} = \sum_{n=0}^{N-1} h \left(\frac{\partial f(\boldsymbol{z}_n, t_n, \boldsymbol{\theta})}{\partial \boldsymbol{\theta}} \right)^T \boldsymbol{\delta}^{n+1}$

Midpoint network: The explicit midpoint method is a two-stage Runge–Kutta method which approximates the solution to the IVP (3.3) by

$$\boldsymbol{z}_n = \boldsymbol{z}_{n-1} + h f \left(\boldsymbol{z}_{n-1} + \frac{h}{2} f\left(\boldsymbol{z}_{n-1}, t_{n-1}, \boldsymbol{\theta}\right), t_{n-1} + \frac{h}{2}, \boldsymbol{\theta} \right), \tag{3.11}$$

for all $n = 1, \ldots, N$ and $h = T/N$, with N being the total number of steps. For notational convenience, we can rewrite Equation (3.11) as

$$\boldsymbol{z}_n = \boldsymbol{z}_{n-1} + hf\left(\tilde{\boldsymbol{z}}_{n-1}, t_{(n-1)^*}, \boldsymbol{\theta}\right), \tag{3.12}$$

where

$$\tilde{\boldsymbol{z}}_{n-1} = \boldsymbol{z}_{n-1} + \frac{h}{2} f\left(\boldsymbol{z}_{n-1}, t_{n-1}, \boldsymbol{\theta}\right), \tag{3.13}$$

and $t_{n^*} = t_n + h/2$. The error at time time step t_n is related to the error at time step t_{n+1} by

$$\boldsymbol{\delta}^n = \left(\boldsymbol{I} + h\left(\frac{\partial f(\tilde{\boldsymbol{z}}_n, t_{n^*}, \boldsymbol{\theta})}{\partial \tilde{\boldsymbol{z}}_n}\left(\boldsymbol{I} + \frac{h}{2}\frac{f(\boldsymbol{z}_n, t_n, \boldsymbol{\theta})}{\partial \boldsymbol{z}_n}\right)\right)^T\right)\boldsymbol{\delta}^{n+1}, \tag{3.14}$$

and the derivative of the cost function C with respect to $\boldsymbol{\theta}$ can be computed as

$$\frac{\partial C}{\partial \boldsymbol{\theta}} = \sum_{n=0}^{N-1} h\left(\frac{\partial f(\tilde{\boldsymbol{z}}_n, t_n^*, \boldsymbol{\theta})}{\partial \boldsymbol{\theta}}\right)^T \boldsymbol{\delta}^{n+1}. \tag{3.15}$$

Proofs for Equation (3.14) and Equation (3.15) can be found in Appendix A. The backpropagation algorithm for the midpoint method is given in Algorithm 4. Note that this approach can be extended to any explicit Runge–Kutta method, however, the notation can become complicated.

Algorithm 4 The midpoint backpropagation algorithm

for $n = 1, \ldots, N$ **do**
    $\tilde{\boldsymbol{z}}_{n-1} = \boldsymbol{z}_{n-1} + \frac{h}{2} f(\boldsymbol{z}_{n-1}, t_{n-1}, \boldsymbol{\theta})$
    $\boldsymbol{z}_n = \boldsymbol{z}_{n-1} + hf(\tilde{\boldsymbol{z}}_{n-1}, t_{(n-1)^*}, \boldsymbol{\theta})$
end for
Compute $\boldsymbol{\delta}^n = \frac{\partial C}{\partial \boldsymbol{z}_n}$
for $n = N-1, \ldots, 1$ **do**
    $\boldsymbol{\delta}^n = \left(\boldsymbol{I} + h\left(\frac{\partial f(\tilde{\boldsymbol{z}}_n, t_{n^*}, \boldsymbol{\theta})}{\partial \tilde{\boldsymbol{z}}_n}\left(\boldsymbol{I} + \frac{h}{2}\frac{f(\boldsymbol{z}_n, t_n, \boldsymbol{\theta})}{\partial \boldsymbol{z}_n}\right)\right)^T\right)\boldsymbol{\delta}^{n+1}$
end for
$\frac{\partial C}{\partial \boldsymbol{\theta}} = \sum_{n=0}^{N-1} h\left(\frac{\partial f(\tilde{\boldsymbol{z}}_n, t_{n^*}, \boldsymbol{\theta})}{\partial \boldsymbol{\theta}}\right)^T \boldsymbol{\delta}^{n+1}$

3.2.3 Comparison of the Adjoint Sensitivity Method and Backpropagation

We now compare the adjoint sensitivity method for NODEs (see Algorithm 2), where we use Euler's method to solve all IVPs, to the Euler backpropagation algorithm (see Algorithm 3). For this example, we choose $t = 0$ and $T = h$ and let $\boldsymbol{z}_0 = \boldsymbol{x}$.

The adjoint sensitivity method with Euler discretization: After one Euler step, we have $\boldsymbol{z}_1 = \boldsymbol{x} + f(\boldsymbol{x}, 0, \boldsymbol{\theta})$ and $\boldsymbol{\lambda}_1^* = \partial_{\boldsymbol{z}_1} C$. Using Equation (3.7), the derivative of the cost function C with respect to the parameters $\boldsymbol{\theta}$ can then be computed as

$$\begin{aligned} \frac{dC}{d\boldsymbol{\theta}} &= h \left(\frac{\partial f(\boldsymbol{z}_1, h, \boldsymbol{\theta})}{\partial \boldsymbol{\theta}} \right)^T \boldsymbol{\lambda}_1^* \\ &= h \left(\frac{\partial f(\boldsymbol{z}_1, h, \boldsymbol{\theta})}{\partial \boldsymbol{\theta}} \right)^T \frac{\partial C}{\partial \boldsymbol{z}_1}. \end{aligned} \tag{A}$$

Euler backpropagation: The forward propagation of the Euler backpropagation algorithm is equivalent to the above, i.e., $\boldsymbol{z}_1 = \boldsymbol{x} + f(\boldsymbol{x}, 0, \boldsymbol{\theta})$ and $\boldsymbol{\delta}^1 = \boldsymbol{\lambda}_1^* = \partial_{\boldsymbol{z}_1} C$. Using Equation (3.10), we have

$$\begin{aligned} \frac{dC}{d\boldsymbol{\theta}} &= h \left(\frac{\partial f(\boldsymbol{x}, 0, \boldsymbol{\theta})}{\partial \boldsymbol{\theta}} \right)^T \boldsymbol{\delta}^1 \\ &= h \left(\frac{\partial f(\boldsymbol{x}, 0, \boldsymbol{\theta})}{\partial \boldsymbol{\theta}} \right)^T \frac{\partial C}{\partial \boldsymbol{z}_1}. \end{aligned} \tag{B}$$

The absolute difference in $d_{\boldsymbol{\theta}} C$ found by the two methods (A) and (B) for one particular parameter $\tilde{\theta}$ is then given by

$$h \left| (\partial_{\tilde{\theta}} f(\boldsymbol{z}_1, h, \boldsymbol{\theta}) - \partial_{\tilde{\theta}} f(\boldsymbol{x}, 0, \boldsymbol{\theta}))^T \frac{\partial C}{\partial \boldsymbol{z}_1} \right|.$$

This problem is well-known in optimal control theory (e.g. Betts and Campbell, 2005; Hinze and Rösch, 2012; Kindermann, 1980). The two approaches are known as first-optimize-then-discretize and first-discretize-then-optimize, respectively. In this work, we focus on the adjoint sensitivity method as the implementation works as a black box for any ODE solver. Nonetheless, we want to bring to the reader's attention, that it is not clear which approach works better in practice.

3.3 Choosing the Dynamics

In the NODE framework, one is free to choose any dynamics f of the form $f(\boldsymbol{z}(t), t, \boldsymbol{\theta})$ for the IVP (3.3). In this section, we review function classes that are implemented in (Grathwohl et al., 2018). We further show why one should neither use autonomous NODEs nor linear NODEs in $\boldsymbol{z}$, i.e., neither $f(\boldsymbol{z}(t), t, \boldsymbol{\theta}) = f(\boldsymbol{z}(t), \boldsymbol{\theta})$ nor $f(\boldsymbol{z}(t), t, \boldsymbol{\theta}) = \boldsymbol{W}(t)\boldsymbol{z}(t) + \boldsymbol{b}(t)$, respectively. Lastly, we prove that picking f from a particular function class leads to universal function approximation; this proof is heavily based on the results of Lin and Jegelka (2018).

For the convenience of this section, let us define the Euler approximation to the IVP

$$\begin{aligned} \frac{d\boldsymbol{z}(t)}{dt} &= f(\boldsymbol{z}(t), t, \boldsymbol{\theta}), \quad t \in [0, T], \\ \boldsymbol{z}(0) &= \boldsymbol{x}, \end{aligned}$$

at time step n with respect to the variable initial value $\boldsymbol{x}$ as $\boldsymbol{z}_n(\boldsymbol{x})$. Hence,

$$\boldsymbol{z}_n(\boldsymbol{x}) = \boldsymbol{z}_{n-1}(\boldsymbol{x}) + hf(\boldsymbol{z}_{n-1}(\boldsymbol{x}), t_{n-1}, \boldsymbol{\theta}), \tag{3.16}$$

with

$$\boldsymbol{z}_0(\boldsymbol{x}) = \boldsymbol{x}. \tag{3.17}$$

3.3.1 Problematic Neural Ordinary Differential Equations

Linear Neural Ordinary Differential Equations: Theorem 3.3.1 shows why linear NODEs are problematic.

Theorem 3.3.1 *Let $\boldsymbol{z}_N$ be the Euler approximation to the IVP*

$$\begin{aligned} \frac{d\boldsymbol{z}(t)}{dt} &= \boldsymbol{W}(t)\boldsymbol{z}(t) + \boldsymbol{b}(t), \quad t \in [0, T], \\ \boldsymbol{z}(0) &= \boldsymbol{z}_0, \end{aligned}$$

at time T. Then we have that

$$\boldsymbol{z}_N = \widetilde{\boldsymbol{W}}\boldsymbol{z}_0 + \widetilde{\boldsymbol{b}},$$

for some $\widetilde{\boldsymbol{W}} \in \mathbb{R}^{d \times d}$ and $\widetilde{\boldsymbol{b}} \in \mathbb{R}^d$.

Theorem 3.3.1 implies that $\boldsymbol{z}_N$ is a linear transformation of the initial value $\boldsymbol{z}_0$, and therefore one cannot encode any non-linear relations in the "output" of linear NODEs using Euler's method. A proof of Theorem 3.3.1 can be found in Appendix B.

Autonomous Neural Ordinary Differential Equations: An autonomous NODE is given by

$$\begin{aligned} \frac{d\boldsymbol{z}(t)}{dt} &= f(\boldsymbol{z}(t), \boldsymbol{\theta}), \quad t \in [0, T], \\ \boldsymbol{z}(0) &= \boldsymbol{z}_0. \end{aligned} \tag{3.18}$$

In Theorem 3.3.2 we show that no matter how expressive f is, and as we take infinitesimally small time steps, the Euler approximation to the solution of the above IVP can never converge to the tent function

$$h(x) = \begin{cases} 1 + x & \text{if } x \in [-1, 0] \\ 1 - x & \text{if } x \in (0, 1] \\ 0 & \text{otherwise} \end{cases}.$$

An example for dynamics of a non-autonomous NODE whose Euler approximation converges to h pointwise is the following:

$$f(z(t), t) = \begin{cases} -Nz(t) & \text{if } t = 0 \text{ and } z(t) \in (-\infty, -1] \cup [1, \infty) \\ N & \text{if } t = 0 \text{ and } z(t) \in (-1, 0] \\ N - 2Nz(t) & \text{if } t = 0 \text{ and } z(t) \in (0, 1) \\ 0 & \text{otherwise} \end{cases},$$

where N is the number of time steps.

Theorem 3.3.2 *Let $z_n(x)$ be the Euler approximation to the solution of the autonomous IVP* (3.18) *at time t_n with respect to the variable initial value $x \in \mathbb{R}$. Then, there is no dynamics f such that*

$$\lim_{N \to \infty} |z_N(x) - h(x)| = 0 \quad \forall x \in \mathbb{R},$$

where

$$h(x) = \begin{cases} 1 + x & \textit{if } x \in [-1, 0] \\ 1 - x & \textit{if } x \in (0, 1] \\ 0 & \textit{otherwise} \end{cases}.$$

A proof of Theorem 3.3.2 can be found in Appendix B. This result renders autonomous NODEs impractical, and therefore from this point onward we only consider non-autonomous NODEs.

3.3.2 Standard Dynamics

The difficulty with choosing the dynamics of the IVP (3.3) is to find an effective way to incorporate the time variable t. The discussed neural network dynamics in this section have time in some form as an input to every layer. The simplest approach is to concatenate time to the input of every layer, resulting in the function class

$$\mathcal{F}_{\text{concat}}(g, \boldsymbol{d}) = \left\{ f_{\text{concat}} \mid \mathbf{x} \mapsto \hat{\boldsymbol{A}}^{L+1} \begin{bmatrix} g\left(\hat{\boldsymbol{A}}^{L} \begin{bmatrix} \mathbf{x}^{L} \\ t \end{bmatrix} + \boldsymbol{b}^{L} \right), t \end{bmatrix}^{T} + \boldsymbol{b}^{L+1} \right\},$$

with $f_{\text{concat}} \colon \mathbb{R}^d \to \mathbb{R}^d$ and

$$\begin{aligned} \mathbf{x}^{l} &= g\left(\hat{\boldsymbol{A}}^{l-1} \begin{bmatrix} \mathbf{x}^{l-1} \\ t \end{bmatrix} + \boldsymbol{b}^{l-1} \right) \\ &= g\left(\hat{\boldsymbol{A}}_1^{l-1} \mathbf{x}^{l-1} + \hat{\boldsymbol{A}}_2^{l-1} t + \boldsymbol{b}^{l-1} \right), \end{aligned} \tag{3.19}$$

for all $l = 2, 3, \ldots, L$ and $\mathbf{x}^1 = \mathbf{x}$. Furthermore, $\boldsymbol{d} = \{\boldsymbol{d}_1, \ldots, \boldsymbol{d}_L\}$ and g is a non-linear function. The dimensions of the matrices and vectors are:

- $\hat{\boldsymbol{A}}^1 \in \mathbb{R}^{\boldsymbol{d}_1 \times (d+1)}$ and $\boldsymbol{b}^1 \in \mathbb{R}^{\boldsymbol{d}_1}$,
- $\hat{\boldsymbol{A}}^l \in \mathbb{R}^{\boldsymbol{d}_l \times (\boldsymbol{d}_{l-1}+1)}$ and $\boldsymbol{b}^l \in \mathbb{R}^{\boldsymbol{d}_l}$ for all $l = 2, \ldots, L$,
- $\hat{\boldsymbol{A}}^{L+1} \in \mathbb{R}^{d \times (\boldsymbol{d}_L+1)}$ and $\boldsymbol{b}^{L+1} \in \mathbb{R}^d$.

The submatrices $\hat{\boldsymbol{A}}_2^l$ in Equation (3.19) are of size $\mathbb{R}^{\boldsymbol{d}_l \times 1}$ for all $l = 1, \ldots, L$ and $\mathbb{R}^{d \times 1}$ for $l = L + 1$, and can therefore be regarded as vectors.

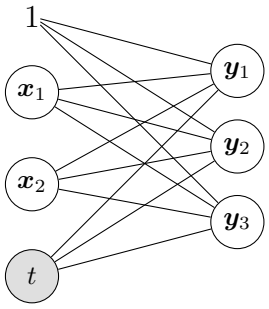
From now on, we refer to NODEs with dynamics $f_{\text{concat}} \in \mathcal{F}_{\text{concat}}(g, \boldsymbol{d})$ as `concat` networks. As can be seen in Equation (3.19), the building block of the `concat` network is

$$\hat{\boldsymbol{A}} \begin{bmatrix} \mathbf{x} \\ t \end{bmatrix} + \boldsymbol{b} = \hat{\boldsymbol{A}}_1 \mathbf{x} + \hat{\boldsymbol{A}}_2 t + \boldsymbol{b}.$$

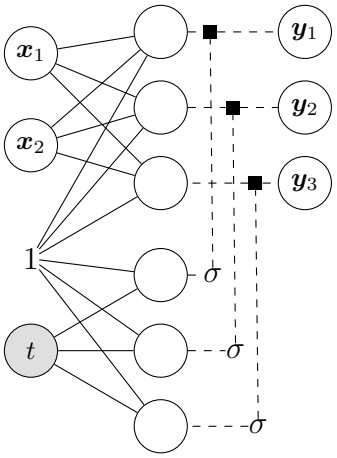
Architecture	$f_{\text{block}}(\boldsymbol{x}, t)$	# of parameters
`concat`	$\boldsymbol{A}\boldsymbol{x} + \boldsymbol{b} + t\boldsymbol{c}$	$d_{\text{out}}(d_{\text{in}} + 2)$
`squash`	$\sigma(t\boldsymbol{c} + \boldsymbol{d}) \odot (\boldsymbol{A}\boldsymbol{x} + \boldsymbol{b})$	$d_{\text{out}}(d_{\text{in}} + 3)$
`concatsquash`	$\sigma(t\boldsymbol{c} + \boldsymbol{d}) \odot (\boldsymbol{A}\boldsymbol{x} + \boldsymbol{b}) + t\boldsymbol{e}$	$d_{\text{out}}(d_{\text{in}} + 4)$
`blend`	$(\boldsymbol{A}\mathbf{x} + \boldsymbol{b})(1 - t) + (\boldsymbol{B}\boldsymbol{x} + \boldsymbol{d})$	$2d_{\text{out}}(d_{\text{in}} + 1)$

Table 3.1: Function f_{block} and number of learnable parameters for the building blocks of NODE dynamics. The matrix $\boldsymbol{A}$ and vector $\boldsymbol{b}$ are elements of $\mathbb{R}^{d_{\text{out}} \times d_{\text{in}}}$ and $\mathbb{R}^{d_{\text{out}}}$, respectively. The sigmoid function is defined in Table 2.2.

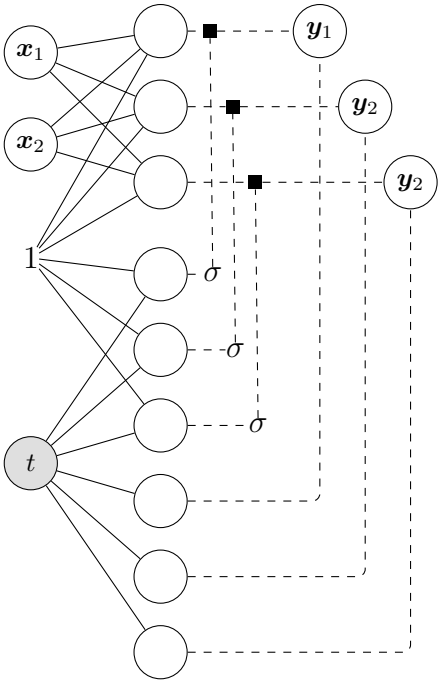
The functions of the building blocks for the `concat` network and three other networks, as well as the number of parameters for each block, are presented in Table 3.1. For a better understanding, the four building blocks are visualized in Figure 3.1 for two input and three output variables each.



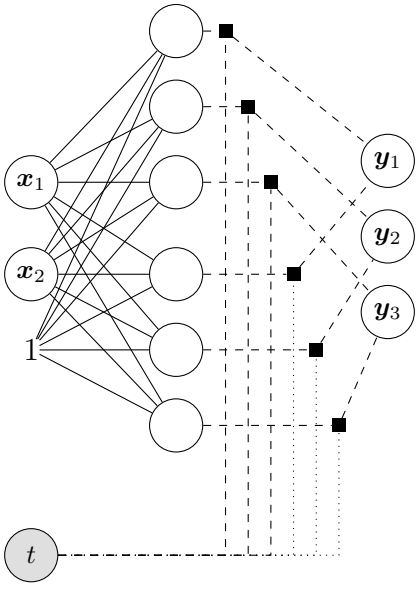
(a) `concat`



(b) `squash`



(c) `concatsquash`



(d) `blend`

Figure 3.1: Example building blocks for the dynamics of NODEs with $d_{\text{in}} = 2$ and $d_{\text{out}} = 3$. Solid and dashed lines represent a multiplication with weights and 1, respectively. Dotted lines represent the function $1 - t$. All inputs to round nodes are added up. Black square nodes indicate a multiplication of the inputs (from left and below) and the intersection of σ with a path means applying the sigmoid function (see Table 2.2).

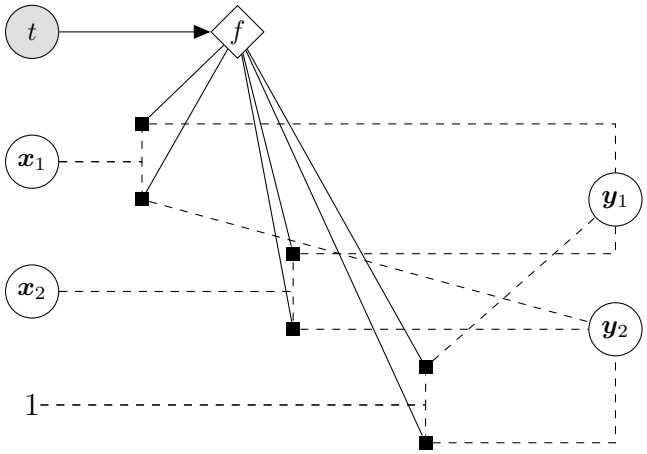


Figure 3.2: Hypernetwork NODE building block for $d_{\text{in}} = d_{\text{out}} = 2$; nodes and lines behave as described in Figure 3.1 and f represents a hypernetwork.

3.3.3 Hypernetworks and Universal Function Approximation

In this section, we describe a fundamentally different class of NODEs and how its discretization leads to universal function approximaton. This class, dubbed hypernetwork NODEs, is implemented in Grathwohl et al. (2018). The building block for the dynamics of hypernetwork NODEs is

$$\boldsymbol{A}(t) + \boldsymbol{b}(t),$$

with the elements of $\boldsymbol{A}(t)$ and $\boldsymbol{b}(t)$ being the output of a hypernetwork with input t. To the best of our knowledge, the idea of hypernetworks to learn the weights of a neural network was introduced in Ha et al. (2016). A hypernetwork building block is visualized in Figure 3.2. Theorem 3.3.3 proves that choosing f from the function class

$$\left\{ f_{\text{universal}} \mid \mathbf{x} \mapsto \boldsymbol{V}(t)\text{ReLU}(\boldsymbol{U}(t)\mathbf{x} + u(t)), \boldsymbol{V} \in \mathbb{R}^{d\times 1}, \boldsymbol{U} \in \mathbb{R}^{1\times d}, u \in \mathbb{R} \right\}, \quad (3.20)$$

with the elements of $\boldsymbol{V}(t)$, $\boldsymbol{U}(t)$ and $u(t)$ being the output of a hypernetwork leads to universal function approximaton when the hypernetwork is expressive enough and an appropriate discretization is chosen.

Theorem 3.3.3 *For any $d \in \mathbb{N}$, Lebesgue-integrable function $g\colon \mathbb{R}^d \to \mathbb{R}$ and $\epsilon > 0$, there is a linear operator $\mathcal{L} : \mathbb{R}^d \to \mathbb{R}$, a two-layer hypernetwork f_{hyper} and a finite N such that*

$$\int_{\mathbb{R}^d} \|g(\boldsymbol{x}) - \mathcal{L} \circ \boldsymbol{z}_N(\boldsymbol{x})\|\, d\boldsymbol{x} \leq \epsilon,$$

where $\boldsymbol{z}_N(\boldsymbol{x})$ *is the Euler approximation to the IVP* (3.3)*, with dynamics* $f_{\text{universal}}$ *and initial value* $\boldsymbol{x}$*, at time* $t = T$*. The width of the hypernetwork is exactly* $2dN + N$*.*

A proof of Theorem 3.3.3 can be found in Appendix B.

Chapter 4

Neural Ordinary Differential Equations for Generative Modeling

In this chapter, we review reversible generative models, a class of generative models that makes use of the change of variables formula. We show how one can infer model parameters from data and how reversible generative models can improve variational autoencoders (see Section 2.4.2). We then give a detailed summary of how NODEs can be used for generative modeling (Chen et al., 2018; Grathwohl et al., 2018).

4.1 Reversible Generative Models

Reversible generative models are a class of generative models that are based on the change of variables formula (CVF). The basic idea is to build a complex distribution implicitly by transforming a simple base distribution.

4.1.1 The Change of Variables Formula

The change of variables formula (4.1) is formalized in Theorem 4.1.1.

Theorem 4.1.1 (Rudin (2006)) *Given a random variable* $\mathbf{z}_0 \sim p_{\mathbf{z}_0}(\mathbf{z}_0)$*, the density of* $\mathbf{x} = f(\mathbf{z}_0)$ *is given by*

$$p_{\mathbf{x}}(\mathbf{x}) = \frac{p_{\mathbf{z}_0}(\mathbf{z}_0)}{\left|\det \frac{\partial f}{\partial \mathbf{z}_0}\right|}, \tag{4.1}$$

when $f\colon \mathbb{R}^d \to \mathbb{R}^d$ is bijective and differentiable.

Note that the CVF (4.1) involves the costly computation of the determinant of the Jacobian $\partial f/\partial \mathbf{z}_0$, which generally has time cost $\mathcal{O}(d^3)$ (Grathwohl et al., 2018). Reversible generative models generally choose f from a class of functions such that the computation of the determinant becomes tractable.

4.1.2 Normalizing Flows

Normalizing flows are based on the work by Tabak et al. (2010) and Tabak and Turner (2013). Given K transformations $f_1, \ldots, f_K$, a sample $\mathbf{z}_0 \sim p_{\mathbf{z}_0}(\mathbf{z}_0)$ can be transformed as

$$\mathbf{z}_K = f_K \circ \cdots \circ f_2 \circ f_1(\mathbf{z}_0).$$

Applying the CVF (4.1) iteratively, the density of $\mathbf{z}_K$ is given by

$$\begin{aligned} p_{\mathbf{z}_K}(\mathbf{z}_K) &= \frac{p_{\mathbf{z}_{K-1}}(\mathbf{z}_{K-1})}{\left|\det \frac{\partial f_K}{\partial \mathbf{z}_{K-1}}\right|} \\ &= \frac{p_{\mathbf{z}_{K-2}}(\mathbf{z}_{K-2})}{\left|\det \frac{\partial f_K}{\partial \mathbf{z}_{K-1}}\right| \left|\det \frac{\partial f_{K-1}}{\partial \mathbf{z}_{K-2}}\right|} \\ &= \frac{p_{\mathbf{z}_0}(\mathbf{z}_0)}{\prod_{i=1}^{K} \left|\det \frac{\partial f_i}{\partial \mathbf{z}_{i-1}}\right|}, \end{aligned}$$

and the log-density is

$$\log p_{\mathbf{z}_K}(\mathbf{z}_K) = \log p_{\mathbf{z}_0}(\mathbf{z}_0) - \sum_{i=1}^{K} \log \left|\det \frac{\partial f_i}{\partial \mathbf{z}_{i-1}}\right|. \tag{4.2}$$

We now review popular choices of transformations that make the computation of the determinant of the Jacobian tractable.

Planar flows (Rezende and Mohamed, 2015): Planar flows are based on transformations f_{planar} from the class

$$\mathcal{F}_{\text{planar}}(g) \coloneqq \{f_{\text{planar}} : \mathbb{R}^d \to \mathbb{R}^d | \boldsymbol{x} \mapsto \boldsymbol{x} + \boldsymbol{u} g(\boldsymbol{w} \cdot \boldsymbol{x} + b), \boldsymbol{u} \in \mathbb{R}^d, \boldsymbol{w} \in \mathbb{R}^d, b \in \mathbb{R}\},$$

where g is generally a smooth non-linear function. The Jacobian of $f_{\text{planar}} \in \mathcal{F}_{\text{planar}}(g)$ can be computed as

$$\frac{\partial f_{\text{planar}}}{\partial \boldsymbol{x}} = \boldsymbol{I} + \boldsymbol{u}\boldsymbol{w}^T g'(\boldsymbol{w} \cdot \boldsymbol{x} + b).$$

Using the matrix determinant lemma (e.g. Harville, 1998)

$$\det\left(\boldsymbol{A} + \boldsymbol{v}_1 \boldsymbol{v}_2^T\right) = \left(1 + \boldsymbol{v}_2^T \boldsymbol{A}^{-1} \boldsymbol{v}_1\right) \det \boldsymbol{A},$$

for an invertible matrix $\boldsymbol{A}$ and column vectors $\boldsymbol{v}_1$ and $\boldsymbol{v}_2$, we compute the determinant of the Jacobian as

$$\det \frac{\partial f_{\text{planar}}}{\partial \boldsymbol{x}} = 1 + \boldsymbol{u} \cdot \boldsymbol{w} g'(\boldsymbol{w} \cdot \boldsymbol{x} + b).$$

Hence, the log-density of $\mathbf{z}_k$ is

$$\log p_{\mathbf{z}_K}(\mathbf{z}_K) = \log p_{\mathbf{z}_0}(\mathbf{z}_0) - \sum_{i=1}^{K} \log \left|1 + \boldsymbol{u}_i \cdot \boldsymbol{w}_i g'(\boldsymbol{w}_i \cdot \mathbf{z}_{i-1} + b_i)\right|. \tag{4.3}$$

The flow defined by Equation (4.3) modifies $p_{\mathbf{z}_0}$ by applying contractions and expansions in the direction perpendicular to $\boldsymbol{w}_i \cdot \mathbf{z}_{i-1} + b_i = 0$ for all $i = 1, \ldots, K$ (Rezende and Mohamed, 2015). Note that the elements of $\mathcal{F}_{\text{planar}}(g)$ are not necessarily invertible. However, Rezende and Mohamed (2015) showed that $\boldsymbol{w} \cdot \boldsymbol{u} \geq -1$ is a sufficient condition for invertibility if $g(x) = \tanh(x)$. The constraint $\boldsymbol{w} \cdot \boldsymbol{u} \geq -1$ can be enforced by replacing $\boldsymbol{u}$ with $\hat{\boldsymbol{u}}$ after every update of the weights, where

$$\hat{\boldsymbol{u}}(u, w) = \boldsymbol{u} + \left[m(\boldsymbol{w} \cdot \boldsymbol{u}) - \boldsymbol{w} \cdot \boldsymbol{u}\right] \frac{\boldsymbol{w}}{\|\boldsymbol{w}\|_2^2},$$

with $m(x) = -1 + \log(1 + \exp x)$. Even though the constraint $\boldsymbol{w} \cdot \boldsymbol{u} \geq -1$ ensures that an inverse of f exists, not having an analytical formulation of f^{-1} renders the planar flow impractical for direct training on data, as we will see in Section 4.1.3.

NICE (Dinh et al., 2014): The NICE framework keeps the determinant of the Jacobian tractable by partitioning dimensions. The basic idea is to split $\boldsymbol{x} = (\boldsymbol{x}_{I_1}, \boldsymbol{x}_{I_2}) \in \mathbb{R}^d$ and apply the transformation

$$f(\boldsymbol{x}) = \begin{bmatrix} \boldsymbol{x}_{I_1} \\ \boldsymbol{x}_{I_2} + m(\boldsymbol{x}_{I_1}) \end{bmatrix},$$

where m is a neural network. The index sets are generally chosen to be

$$I_1 = \{1, 2, \cdots, |I_1| - 1, |I_1|\}$$

and

$$I_2 = \{|I_1| + 1, \cdots, d - 1, d\}.$$

This building block has unit Jacobian-determinant for any m and the inverse is given by

$$f^{-1}(f(\mathbf{x})) = \begin{bmatrix} f(\mathbf{x})_{I_1} \\ f(\mathbf{x})_{I_2} - m(f(\mathbf{x})_{I_1}) \end{bmatrix}.$$

In order to allow all dimensions to influence each other and make the model non volume-preserving[1], Dinh et al. (2014) stack multiple building blocks and apply a scaling factor in the last term, e.g.,

$$\begin{aligned}
f^{(1)}_{I_1} &= \boldsymbol{x}_{I_1}, \\
f^{(1)}_{I_2} &= \boldsymbol{x}_{I_2} + m^{(1)}(\boldsymbol{x}_{I_1}), \\
f^{(2)}_{I_1} &= f^{(1)}_{I_1} + m^{(2)}\left(f^{(1)}_{I_2}\right), \\
f^{(2)}_{I_2} &= f^{(1)}_{I_2}, \\
f^{(3)}_{I_1} &= f^{(2)}_{I_1}, \\
f^{(3)}_{I_2} &= f^{(2)}_{I_2} + m^{(3)}\left(f^{(2)}_{I_1}\right), \\
f(\mathbf{x}) &= \exp(\boldsymbol{s}) \odot \left(f^{(3)} \circ f^{(2)} \circ f^{(1)}\right)(\mathbf{x}),
\end{aligned}$$

[1] A volume-preserving density transformation has unit Jacobian determinant.

with neural networks $m^{(1)}$, $m^{(2)}$, and $m^{(3)}$ and the exponential function is applied element-wise, i.e., $(\exp \boldsymbol{s})_i = \exp \boldsymbol{s}_i$. The log-density of the above is

$$\begin{aligned}
\log p_{\mathbf{z}_1}(\mathbf{z}_1) &= \log p_{\mathbf{z}_0}(\mathbf{z}_0) - \log \left| \det \frac{\partial f}{\partial \mathbf{z}_0} \right| \\
&= \log p_{\mathbf{z}_0}(\mathbf{z}_0) - \log \left| \det \frac{\partial \exp(\boldsymbol{s}) \odot \left(f^{(3)} \circ f^{(2)} \circ f^{(1)}\right)(\mathbf{z}_0)}{\partial \mathbf{z}_0} \right| \\
&= \log p_{\mathbf{z}_0}(\mathbf{z}_0) - \log \left| \det \frac{\partial \left(\exp(\boldsymbol{s}) \odot f^{(3)}\right)}{\partial f^{(2)}} \frac{\partial f^{(2)}}{\partial f^{(1)}} \frac{\partial f^{(1)}}{\partial \mathbf{z}_0} \right| \\
&= \log p_{\mathbf{z}_0}(\mathbf{z}_0) - \log \left| \det \frac{\partial \left(\exp(\boldsymbol{s}) \odot f^{(3)}\right)}{\partial f^{(2)}} \right| \quad \left(\text{since } \partial_{f^{(1)}} f^{(2)} = \boldsymbol{I} \text{ and } \partial_{\mathbf{z}_0} f^{(1)} = \boldsymbol{I}\right) \\
&= \log p_{\mathbf{z}_0}(\mathbf{z}_0) - \log \left| \det \operatorname{diag}(\exp(\boldsymbol{s})) \right| \quad \left(\text{since } \partial_{f^{(2)}} f^{(3)} = \boldsymbol{I}\right) \\
&= \log p_{\mathbf{z}_0}(\mathbf{z}_0) - \sum_{i=1}^{d} \boldsymbol{s}_i,
\end{aligned}$$

where

$$(\operatorname{diag} \boldsymbol{a})_{ij} = \begin{cases} \boldsymbol{a}_i & \text{if } i = j, \\ 0 & \text{otherwise} \end{cases}.$$

Real NVP (Dinh et al., 2017): Real NVP builds on the work of Dinh et al. (2014) and modifies the building block as

$$f(\boldsymbol{x}) = \begin{bmatrix} \boldsymbol{x}_{I_1} \\ \boldsymbol{x}_{I_2} \odot \exp(l(\boldsymbol{x}_{I_1})) + m(\boldsymbol{x}_{I_1}) \end{bmatrix},$$

where both l and m are neural networks. The determinant of the Jacobian is then given as

$$\det \frac{\partial f}{\partial \boldsymbol{x}} = \exp \left(\sum_{i=1}^{|I_2|} l(\boldsymbol{x}_{I_1})_i \right).$$

Autoregressive transformations: In this paragraph, we summarize how to use autoregressive models as normalizing flows; this idea was, to the best of our knowledge, introduced in Papamakarios et al. (2017). Consider an autoregressive model where the conditionals are modeled as

$$p(\mathbf{x}_{o_i} \mid \mathbf{x}_{o_{<i}}) = \mathcal{N}\left(\mathbf{x}_{o_i} \mid \mu_{o_i}, (\exp \alpha_{o_i})^2\right),$$

and the permutation o is in order, i.e., $o = 1, \ldots, d$. This ordering implies

$$p(\mathbf{x}_i \mid \mathbf{x}_{1:(i-1)}) = \mathcal{N}\left(\mathbf{x}_i \mid \mu_i, (\exp \alpha_i)^2\right),$$

where $\mu_i = f_{\mu_i}(\mathbf{x}_{1:(i-1)})$, $\alpha_i = f_{\alpha_i}(\mathbf{x}_{1:(i-1)})$ and $\mathbf{x} = 1 : (i-1) = [\mathbf{x}_1, \ldots, \mathbf{x}_{i-1}]^T$. One can generate a sample $\mathbf{x}_{\text{auto}}$ by sequentially sampling each dimension, i.e.,

$$(\mathbf{x}_{\text{auto}})_i = \mathrm{u} \exp \alpha_i + \mu_i, \tag{4.4}$$

where $\mathrm{u} \sim \mathcal{N}(0, 1)$. Equation (4.4) shows that we can express the model as $\mathbf{x} = f(\mathbf{u})$, where $\mathbf{u} \sim \mathcal{N}(\mathbf{0}, \boldsymbol{I})$. A drawback of autoregressive transformations are that they require d evaluations to compute their inverse:

$$\mathbf{u}_i = (\mathbf{x}_i - \mu_i) \exp(-\alpha_i).$$

The determinant of the Jacobian is given as

$$\det \partial_{\mathbf{u}} f = \exp\left(\sum_{i=1}^{d} \alpha_i\right).$$

Autoregressive transformations achieve state-of-the-art results for density estimation on tabular datasets (Oliva et al., 2018; Papamakarios et al., 2017).

4.1.3 Training on Data

Given a dataset $\mathbf{X} = \left\{\mathbf{x}^{(i)}\right\}_{i=1}^{m} \sim p_{\text{data}}$, we want to infer the parameters $\boldsymbol{\lambda} = \{\boldsymbol{\lambda}_1, \ldots, \boldsymbol{\lambda}_K\}$, such that the resulting density $p^{\boldsymbol{\lambda}}_{\mathbf{z}_K}(\mathbf{z})$ is close to p_{data}. It is a well-known fact that the MLE of the parameters using $\mathbf{X}$ is a Monte Carlo approximation to finding the parameters that

minimize the KL divergence

$$\begin{aligned} D_{\mathrm{KL}}(p_{\mathrm{data}} \parallel p^{\boldsymbol{\lambda}}_{\mathbf{z}_K}) &= \int_{\mathbf{z}} p_{\mathrm{data}} \left[\log p_{\mathrm{data}} - \log p^{\boldsymbol{\lambda}}_{\mathbf{z}_K}\right] d\mathbf{x} \\ &= -\int_{\mathbf{z}} p_{\mathrm{data}} \log p^{\boldsymbol{\lambda}}_{\mathbf{z}_K} \, d\mathbf{x} + c \\ &= -\mathbb{E}_{p_{\mathrm{data}}}[\log p^{\lambda}_{\mathbf{z}_K}] + c \\ &\approx -\frac{1}{m} \sum_{i=1}^{m} \log p^{\boldsymbol{\lambda}}_{\mathbf{z}_K} \left(\mathbf{x}^{(i)}\right) + c, \quad \mathbf{x}^{(i)} \sim p_{\mathrm{data}}, \end{aligned}$$

where the constant c is the negative entropy of the data distribution, i.e.,

$$c = \int_{\mathbf{z}} p_{\mathrm{data}} \log p_{\mathrm{data}} \, d\mathbf{x}.$$

Hence,

$$\arg\min_{\boldsymbol{\lambda}} D_{\mathrm{KL}}(p_{\mathrm{data}} \parallel p^{\boldsymbol{\lambda}}_{\mathbf{z}_K}) \approx \arg\max_{\boldsymbol{\lambda}} \sum_{i=1}^{m} \log p^{\boldsymbol{\lambda}}_{\mathbf{z}_K} \left(\mathbf{x}^{(i)}\right). \tag{4.5}$$

As we have an implicit model for $q^{\boldsymbol{\lambda}}_{\mathbf{z}_K}$, we need to address how $q^{\boldsymbol{\lambda}}_{\mathbf{z}_K}(\mathbf{x})$ is actually computed. Given a reversible generative model, we can compute the latent variables using the recursive formula

$$\mathbf{z}_{i-1}(\boldsymbol{x}) = f_i^{-1}(\mathbf{z}_i(\mathbf{x})), \tag{4.6}$$

for all $i = K, \ldots, 1$ with $\mathbf{z}_K(\mathbf{x}) = \mathbf{x}$. Given, $\mathbf{z}_0, \ldots, \mathbf{z}_K$, the quantity $q^{\boldsymbol{\lambda}}_{\mathbf{z}_K}(\mathbf{x})$ can be computed using Equation (4.2). One can see from Equation (4.6), why having an analytical inverse is important, and therefore why planar flows are impractical for inference from data.

4.1.4 Improving Variational Inference

In this section, we address how reversible generative models can be used to improve variational autoencoders (see Section 2.4.2). The inference network of VAEs is generally of the form

$$q(\mathbf{z} \mid \mathbf{x}, \boldsymbol{\phi}) = \mathcal{N}\left(\mathbf{z} \mid \boldsymbol{\mu}, \mathrm{diag}\left(\boldsymbol{\sigma}^2\right)\right),$$

where $\boldsymbol{\mu}$ and $\boldsymbol{\sigma}$ are generally outputs of a neural network, i.e., $\boldsymbol{\mu} = \boldsymbol{\mu}(\mathbf{x})$ and $\boldsymbol{\sigma} = \boldsymbol{\sigma}(\mathbf{x})$. In order to make the inference network more expressive, it is possible to use a normalizing flow to transform samples from $q(\mathbf{z} \mid \mathbf{x}, \boldsymbol{\phi})$, i.e.,

$$\mathbf{z}_K^{\mathrm{NF}} = f_K \circ \cdots \circ f_1(\mathbf{z}),$$

where $\mathbf{z} \sim q(\mathbf{z} \mid \mathbf{x}, \boldsymbol{\phi})$. The resulting inference network is then given by

$$q_K^{\mathrm{NF}}(\mathbf{z} \mid \mathbf{x}, \boldsymbol{\phi}, \boldsymbol{\lambda})(\mathbf{z}_K^{\mathrm{NF}}) = \frac{q(\mathbf{z} \mid \mathbf{x}, \boldsymbol{\phi})(\mathbf{z})}{\prod_{i=1}^{K} \left| \det \frac{\partial f_i}{\partial \mathbf{z}_{i-1}^{\mathrm{NF}}} \right|},$$

where $\mathbf{z}_0 = \mathbf{z}$. A lower bound to the marginal log-likelihood $p(\mathbf{x} \mid \boldsymbol{\theta})$ (see Section 2.4.2) can be computed as

$$-\mathcal{F}^{\mathrm{NF}}(\mathbf{x}) = -D_{\mathrm{KL}}(q_K^{\mathrm{NF}}(\mathbf{z} \mid \mathbf{x}, \boldsymbol{\phi}, \boldsymbol{\lambda}) \parallel p(\mathbf{z} \mid \boldsymbol{\theta})) + \mathbb{E}_{q_K^{\mathrm{NF}}}[\log p(\mathbf{x} \mid \mathbf{z}, \boldsymbol{\theta})]. \tag{4.7}$$

Given a dataset $\mathbf{X} = \left\{\mathbf{x}^{(i)}\right\}_{i=1}^{m}$, all three sets of parameters can then be simultaneously learned by

$$\boldsymbol{\varphi}_{\mathrm{NF}}^* = \arg\max_{\boldsymbol{\varphi}_{\mathrm{NF}}} -\sum_{i=1}^{m} \mathcal{F}\left(x^{(i)}\right),$$

where $\boldsymbol{\varphi}_{\mathrm{NF}} = \{\boldsymbol{\theta}, \boldsymbol{\phi}, \boldsymbol{\lambda}\}$.

Samples from the VAE can be generated by first sampling $\mathbf{z} \sim \mathcal{N}(\mathbf{0}, \boldsymbol{I})$, then transforming $\mathbf{z}$ to $\mathbf{z}_K^{\mathrm{NF}} = f_K \circ \cdots \circ f_1(\mathbf{z})$, and lastly sampling $\mathbf{x}^* \sim p\left(\mathbf{x} \mid \mathbf{z}_K^{\mathrm{NF}}, \boldsymbol{\theta}\right)$. Note that we could choose the flow parameters $\boldsymbol{\lambda}$ to be data-dependent, i.e., $\boldsymbol{\lambda} = \boldsymbol{\lambda}(\mathbf{x})$. This modification might help in maximizing the negative free energy (4.7), however, it makes the VAE lose the ability to draw (data-independent) samples.

4.2 Generative Modeling with Neural Ordinary Differential Equations

In this section, we review how NODEs can be used for generative modeling. This idea was introduced in Chen et al. (2018) and Grathwohl et al. (2018).

4.2.1 The Instantaneous Change of Variables Formula

Density estimation with NODEs is based on a continuous formulation of the change of variables formula (4.1). Consider the IVP where the initial condition is drawn from some probability distribution $p_{\mathbf{z}_0}$, i.e.,

$$\begin{aligned} \frac{d\mathbf{z}(t)}{dt} &= f(\mathbf{z}(t), t, \boldsymbol{\theta}), \quad t \in [0, T], \\ \mathbf{z}(0) &\sim p_{\mathbf{z}_0}(\mathbf{z}(0)). \end{aligned}$$

Chen et al. (2018) showed that when $\mathbf{z}(t)$ is transformed by the dynamics $f(\mathbf{z}(t), t, \boldsymbol{\theta})$, the log-density also follows a differential equation

$$\frac{d \log p(\mathbf{z}(t), t)}{dt} = -\operatorname{tr} \frac{\partial f}{\partial \mathbf{z}}, \quad t \in [0, T],$$

with initial condition $\log p(\mathbf{z}(0), 0) = \log p_{\mathbf{z}_0}(\mathbf{z}(0))$; the proof of this is based on the definition of the derivative and on Jacobi's formula (Chen et al., 2018). The complete IVP is then given as

$$\begin{aligned} \frac{d}{dt} \begin{bmatrix} \mathbf{z}(t) \\ \log p(\mathbf{z}(t), t) \end{bmatrix} &= \begin{bmatrix} f(\mathbf{z}(t), t, \boldsymbol{\theta}) \\ -\operatorname{tr} \partial_{\mathbf{z}} f \end{bmatrix}, \quad t \in [0, T], \\ \mathbf{z}(0) &\sim p_{\mathbf{z}_0}(\mathbf{z}(0)), \\ \log p(\mathbf{z}(0), 0) &= \log p_{\mathbf{z}_0}(\mathbf{z}(0)). \end{aligned} \tag{4.8}$$

If we use $p(\mathbf{z}(T), T)$ as the model distribution for a generative model, we refer to the resulting model as a continuous normalizing flow (CNF) (Grathwohl et al., 2018). Continuous normalizing flows have two advantages over the reversible generative models discussed in Section 4.1.2:

1. instead of the log determinant, one just needs to compute the trace of the Jacobian which is generally computationally cheaper;
2. the dynamics f of the differential equation do not need to be bijective; the IVP (4.8) has a unique solution if $\partial_{\mathbf{z}} f$ and f are uniformly Lipschitz continuous in $\mathbf{z}(t)$ and continuous in t according to Theorem 2.1.1.

We can learn the parameters $\boldsymbol{\theta}$ of f, which inherently determine the model distribution $p(\mathbf{z}(T), T)$, by maximum likelihood estimation (similar to Equation (4.5))

$$\arg\max_{\boldsymbol{\theta}} \sum_{i=1}^{m} \log p\left(\mathbf{x}^{(i)}, T\right), \quad \mathbf{x}^{(i)} \sim p_{\text{data}}.$$

In order to compute $p\left(\mathbf{x}^{(i)}, T\right)$ we first need to compute the initial value $\mathbf{z}^{(i)}(0)$ that generated $\mathbf{x}^{(i)}$ by computing

$$\mathbf{z}^{(i)}(0) = \mathbf{z}^{(i)}(T) + \int_T^0 f\left(\mathbf{z}^{(i)}(t), t, \theta\right)\, dt,$$

with $\mathbf{z}^{(i)}(T) = \mathbf{x}^{(i)}$. We can then compute $\log p\left(\mathbf{x}^{(i)}, T\right)$ as

$$\log p\left(\mathbf{x}^{(i)}, T\right) = \log p_{\mathbf{z}_0}\left(\mathbf{z}_0^{(i)}\right) - \int_0^T \operatorname{tr} \partial_{\mathbf{z}} f\, dt. \tag{4.9}$$

The adjoint method for CNFs to compute the gradients of a cost function C with respect to $\boldsymbol{\theta}$, given a dataset $\mathbf{X}$, is summarized in Algorithm 5; a derivation, based on constrained optimization, can be found in Appendix C.

Algorithm 5 The adjoint sensitivity method for CNFs

1: $\begin{bmatrix} \mathbf{z}(0) \\ \log p(\mathbf{z}(T), T) - \log p(\mathbf{z}(0), 0) \end{bmatrix} = \texttt{odesolve}\left(\begin{bmatrix} f \\ \operatorname{tr} \partial_{\mathbf{z}} f \end{bmatrix}, \begin{bmatrix} \mathbf{x} \\ \mathbf{0} \end{bmatrix}, [T, 0]\right)$

2: Compute $\partial C \coloneqq \partial_{\log p(\mathbf{z}(T),T)} C\left(\left(\log p(\mathbf{z}(T), T) - \log p(\mathbf{z}(0), 0)\right) + \log p_{\mathbf{z}_0}(\mathbf{z}(0))\right)$

3: $\begin{bmatrix} \mathbf{z}(T) \\ \boldsymbol{\lambda}^*(T) \\ d_{\boldsymbol{\theta}} C \end{bmatrix} = \texttt{odesolve}\left(\begin{bmatrix} f \\ -\partial C \partial_{\mathbf{z}} \operatorname{tr} \partial_{\mathbf{z}} f - \left(\partial_{\mathbf{z}} f\right)^T \boldsymbol{\lambda}^* \\ -\left(\partial_{\boldsymbol{\theta}} f\right)^T \boldsymbol{\lambda}^* \end{bmatrix}, \begin{bmatrix} \mathbf{z}(0) \\ \partial C \frac{\partial \log p_{\mathbf{z}_0}}{\mathbf{z}(0)} \\ -\partial C \int_T^0 \operatorname{tr} \partial_{\mathbf{z}} f\, dt \end{bmatrix}, [0, T]\right)$

4.2.2 Stacking Continuous Normalizing Flows

As shown in Section 4.1.2 for the NICE architecture (Dinh et al., 2014), generative models based on the normalizing flow idea can, and often do, compose many flows to improve performance. In this section, we show how to stack multiple CNFs.

In Section 4.2.1, we assumed that the probability distribution $p_{\mathbf{z}_0}$ is pre-defined. In order to stack two CNFs, we now assume that $p_{\mathbf{z}_0}$ is the output from another CNF. Hence, instead of computing $\log p\left(\mathbf{x}^{(i)}, T\right)$ from Equation (4.9) directly, we first need to compute

$$\begin{aligned} \log p_{\mathbf{z}_0}\left(\mathbf{z}^{(i)}(0)\right) &= \log \hat{p}\left(\mathbf{z}^{(i)}(0), \hat{T}\right) \\ &= \log \hat{p}_{\hat{\mathbf{z}}_0}\left(\hat{\mathbf{z}}^{(i)}(0)\right) - \int_0^{\hat{T}} \operatorname{tr} \partial_{\hat{\mathbf{z}}} \hat{f}\, dt. \end{aligned}$$

where $p_{\hat{\mathbf{z}}_0}$ is a known distribution and $\hat{\mathbf{z}}^{(i)}(0)$ can be computed as

$$\hat{\mathbf{z}}^{(i)}(0) = \hat{\mathbf{z}}^{(i)}(\hat{T}) + \int_{\hat{T}}^{0} \hat{f}\left(\hat{\mathbf{z}}^{(i)}(t), \hat{t}, \hat{\boldsymbol{\theta}}\right) d\hat{t},$$

with $\hat{\mathbf{z}}^{(i)}(T) = \mathbf{z}^{(i)}(0)$. Finally, we can compute $\log p\left(\mathbf{x}^{(i)}, T\right)$ as

$$\begin{aligned}\log p\left(\mathbf{x}^{(i)}, T\right) &= \log p_{\mathbf{z}_0}\left(\mathbf{z}_0^{(i)}\right) - \int_0^T \operatorname{tr} \partial_{\mathbf{z}} f \, dt \\ &= \log \hat{p}_{\hat{\mathbf{z}}_0}\left(\hat{\mathbf{z}}^{(i)}(0)\right) - \int_0^{\hat{T}} \operatorname{tr} \partial_{\hat{\mathbf{z}}} \hat{f} \, d\hat{t} - \int_0^T \operatorname{tr} \partial_{\mathbf{z}} f \, dt.\end{aligned}$$

Doing this recursively, we can stack an unrestricted amount of CNFs. In Chapter 5, we will make use of this by stacking up to five CNFs.

4.2.3 Trace Estimation

To compute the trace of the Jacobian, one needs to compute the derivatives $\frac{\partial f_i}{\partial \mathbf{z}_i}$ for all $i = 1, \ldots, d$, which is approximately as expensive as d evaluations of f (Grathwohl et al., 2018). On the other hand, using reverse-mode automatic differentiation (Linnainmaa, 1976), vector-Jacobian products of the form $\boldsymbol{v}^T \frac{\partial f}{\partial \mathbf{z}}$ can be computed for approximately the cost of one evaluation of f (Grathwohl et al., 2018).

Hutchinson's trace estimator (Hutchinson, 1990; Skilling, 1989) gives an unbiased estimate of the trace as

$$\operatorname{tr} \boldsymbol{A} = \mathbb{E}_{\boldsymbol{\epsilon} \sim p_{\boldsymbol{\epsilon}}}\left[\boldsymbol{\epsilon}^T \boldsymbol{A} \boldsymbol{\epsilon}\right],$$

for any square matrix $\boldsymbol{A}$, with the requirements $\mathbb{E}[\boldsymbol{\epsilon}] = \mathbf{0}$ and $\operatorname{Cov}[\boldsymbol{\epsilon}] = \boldsymbol{I}$. Hence, instead of computing the trace of the Jacobian for approximately the cost of d evaluations of f, we can approximate it by $\boldsymbol{\epsilon}^T \partial_{\mathbf{z}} f \boldsymbol{\epsilon}$ for approximately the cost of one evaluation of f and one vector-vector product (negligible cost). The distribution $p_{\boldsymbol{\epsilon}}$ is typically chosen to be $\mathcal{N}(\mathbf{0}, \boldsymbol{I})$ or the Rademacher distribution (Grathwohl et al., 2018). Approximating the trace of the Jacobian is especially useful when the dimensionality of the data is large, e.g., for high-quality images.

4.3 Problems of the Discretized Instantaneous Change of Variables Formula

In this section, we investigate some issues of the discretized instantaneous CVF by comparing it to the standard CVF.

4.3.1 Euler Discretization

We start by comparing the standard CVF to the Euler-discretized instantaneous CVF. Let $\mathbf{s} = [\mathbf{z}(\mathrm{t}), \mathrm{t}]^T \sim p(\mathbf{s})$ with

$$p(\mathbf{s}) = p_{\mathbf{z}(\mathrm{t})}(\mathbf{z}(\mathrm{t}))\delta(\mathrm{t} = t)$$

where δ is the Dirac delta distribution. Furthermore, define the map $f_{\text{Euler}} \colon \mathbb{R}^{d+1} \to \mathbb{R}^{d+1}$ as

$$f_{\text{Euler}}(\mathbf{s}) = \begin{bmatrix} \mathbf{z}(\mathrm{t}) + hf(\mathbf{s}) \\ \mathrm{t} + h \end{bmatrix}.$$

The random variable $\mathbf{y} = f_{\text{Euler}}(\mathbf{s})$ has a density function defined by the CVF (4.1)

$$p(\mathbf{y}) = \frac{p(\mathbf{s})}{|\det \partial_{\mathbf{s}} f_{\text{Euler}}|},$$

where the Jacobian is

$$(\partial_{\mathbf{s}} f_{\text{Euler}})_{ij} = \begin{cases} 1 + h\partial_{\mathbf{z}_j} f_i & \text{if } i = j \text{ and } i < d+1 \\ h\partial_{\mathbf{z}_j} f_i & \text{if } i \neq j \text{ and } i < d+1 \\ 0 & \text{if } i < d+1 \text{ and } j = d+1 \\ 0 & \text{if } i = d+1 \text{ and } j < d+1 \\ 1 & \text{if } i = j = d+1 \end{cases}.$$

The Jacobian is a block matrix with zero lower-left and upper-right blocks, i.e.,

$$\partial_{\mathbf{s}} f_{\text{Euler}} = \begin{bmatrix} \boldsymbol{J}_{1,1} & \mathbf{0} \\ \mathbf{0}^T & 1 \end{bmatrix},$$

where $\boldsymbol{J}_{1,1} \in \mathbb{R}^{d \times d}$. By the block determinant formula (e.g. Trefethen and Bau III, 1997), we can compute the determinant of the Jacobian as

$$\det \partial_{\mathbf{s}} f_{\text{Euler}} = \det(\boldsymbol{J}_{1,1}) \det 1 = \det \boldsymbol{J}_{1,1}.$$

Using that $\boldsymbol{J}_{1,1} = \boldsymbol{I} + h\hat{\boldsymbol{J}}_{1,1}$, with

$$\left(\hat{\boldsymbol{J}}_{1,1}\right)_{ij} = \partial_{\mathbf{z}_j} f_i,$$

we can rewrite the determinant of the Jacobian as (Rezende et al., 2019)

$$\det \partial_{\mathbf{s}} f_{\text{Euler}} = \det \boldsymbol{J}_{1,1} = \det\left(\boldsymbol{I} + h\hat{\boldsymbol{J}}_{1,1}\right) = 1 + h \operatorname{tr} \hat{\boldsymbol{J}}_{1,1} + \mathcal{O}\left(h^2\right).$$

Hence, the density of $\mathbf{y}$ is given by

$$p(\mathbf{y}) = \frac{p(\mathbf{s})}{\left|1 + h \operatorname{tr} \hat{\boldsymbol{J}}_{1,1} + \mathcal{O}\left(h^2\right)\right|}. \tag{4.10}$$

Note that for $d = 1$ the determinant of the Jacobian is exactly $1 + h\partial_{\mathrm{z}} f$, and therefore

$$p(\mathbf{y}) = \frac{p(\mathbf{s})}{|1 + h\partial_{\mathrm{z}} f|}.$$

Let us now consider one Euler step of the instantaneous change of variables formula, i.e.,

$$\log p(\mathbf{z}(h), h) = \log p(\mathbf{z}(0), 0) - h \operatorname{tr} \partial_{\mathbf{z}} f(\mathbf{z}(0), 0),$$

where $\mathbf{z}(h) = \mathbf{z}(0) + hf(\mathbf{z}(0), 0)$. Applying the exponential function yields

$$\begin{aligned} p(\mathbf{z}(h), h) &= p(\mathbf{z}(0), 0) \exp(-h \operatorname{tr} \partial_{\mathbf{z}} f(\mathbf{z}(0), 0)) \\ &= \frac{p(\mathbf{z}(0), 0)}{\exp(h \operatorname{tr} \partial_{\mathbf{z}} f(\mathbf{z}(0), 0))} \\ &= \frac{p(\mathbf{z}(0), 0)}{\sum_{k=0}^{\infty} \frac{h^k (\operatorname{tr} \partial_{\mathbf{z}} f(\mathbf{z}(0), 0))^k}{k!}}. \end{aligned}$$

Letting $\mathrm{t} = 0$ for the CVF, we can see that the difference of the CVF and the discretized instantaneous change of variables formula is

$$\left|\frac{p(\boldsymbol{z}(h), h) - p(\mathbf{y})}{p(\mathbf{s})}\right| = \left|\frac{1}{|1 + h \operatorname{tr} \partial_{\mathbf{z}} f(\mathbf{z}(0), 0) + \mathcal{O}\left(h^2\right)|} - \frac{1}{\sum_{k=0}^{\infty} \frac{h^k (\operatorname{tr} \partial_{\mathbf{z}} f(\mathbf{z}(0), 0))^k}{k!}}\right|,$$

using $p(\mathbf{s}) = p(\mathbf{z}(0), 0)$. For sufficiently small h^2, the above is $\mathcal{O}(h^2)$. Note that when we use trace estimation, as described in Section 4.2.3, the error is $\mathcal{O}(h)$ even for sufficiently small h. Furthermore, for too large h, the error is $\mathcal{O}(1)$. We make this difference clear by considering an example.

Transforming a one-dimensional normal distribution: Let $f(\mathrm{z}, t) = \mu(t) + \mathrm{z}(t)\sigma(t)$ with $\sigma(t) > 0$ for all $t \in [0, T]$ and let $p(\mathrm{z}(0), 0) = \mathcal{N}(\mathrm{z}(0) \mid 0, 1)$. The CVF with the Euler map f_{Euler} transforms $p(\mathrm{z}(0), 0)$ as

$$p(\mathbf{y}) = \frac{p(\mathrm{z}(0), 0)}{1 + h\sigma(0)},$$

with

$$\begin{aligned}\mathbf{y}_1 &= \mathrm{z}(0) + h\mu(0) + h\mathrm{z}(0)\sigma(0) \\ &= h\mu(0) + (1 + h\sigma(0))\mathrm{z}(0),\end{aligned}$$

and therefore

$$\begin{aligned}p(\mathbf{y}_1, t + h) &= \frac{\mathcal{N}\left(\mathrm{z}(0) \mid 0, 1\right)}{(1 + h\sigma(0))} \\ &= \frac{\mathcal{N}\left(\frac{\mathbf{y}_1 - h\mu(0)}{1 + h\sigma(0)} \mid 0, 1\right)}{(1 + h\sigma(0))} \\ &= \mathcal{N}\left(\mathbf{y}_1 \mid h\mu(0), (1 + h\sigma(0))\right).\end{aligned}$$

On the other hand, one Euler step of the instantaneous change of variables formula gives

$$\begin{aligned}p(\mathrm{z}(h), t + h) &= \frac{p(\mathrm{z}(0), 0)}{\sum_{k=0}^{\infty} \frac{h^k (\sigma(0))^k}{k!}} \\ &= \frac{p(\mathrm{z}(0), 0)}{1 + h\sigma(0) + \sum_{k=2}^{\infty} \frac{h^k (\sigma(0))^k}{k!}},\end{aligned}$$

which is not even a density function since

$$\int_{\mathbb{R}} p(\mathrm{z}(h), t + h)\, d\mathrm{z}(h) < \int_{\mathbb{R}} p(\mathbf{y}_1, t + h)\, d\mathbf{y}_1 = 1.$$

Hence, $p(\mathrm{z}(h), t + h)$ is equal to $s\mathcal{N}\left(\mathbf{y}_1 \mid h\mu(0), (1 + h\sigma(0))\right)$ for some $0 < s < 1$.

[2] In particular h small enough such that $1 + h \operatorname{tr} \hat{\boldsymbol{J}}_{1,1} + \mathcal{O}\left(h^2\right) > 0$ and we can drop the absolute value in Equation (4.10).

4.3.2 One-Dimensional Midpoint Discretization

We now examine how the discrepancy of the CVF and the discretized instantaneous CVF behaves for a higher-order ODE solver. It is not trivial to do this for an arbitrary method, however, we can explore how the discrepancy behaves for the one-dimensional midpoint discretization.

We define the map $f_{\mathrm{MP}} \colon \mathbb{R}^2 \to \mathbb{R}^2$ as

$$f_{\mathrm{MP}}(\mathbf{s}) = \begin{bmatrix} \mathrm{z}(t) + hf\left(\mathrm{z}(t) + \frac{h}{2} f(\mathbf{s}), \mathrm{t} + \frac{h}{2}\right) \\ \mathrm{t} + h \end{bmatrix},$$

where $\mathbf{s} = [\mathrm{z}(\mathrm{t}), \mathrm{t}]^T$. The determinant of the Jacobian is

$$\begin{aligned} \det \partial_{\mathbf{s}} f_{\mathrm{MP}} &= 1 + h\partial_{\mathrm{z(t)}} f\left(\mathrm{z(t)} + \frac{h}{2} f(\mathbf{s}), \mathrm{t} + \frac{h}{2}\right) \\ &= 1 + h\partial_{\mathrm{z(t)} + \frac{h}{2} f(\mathbf{s})} f\left(\mathrm{z(t)} + \frac{h}{2} f(\mathbf{s}), \mathrm{t} + \frac{h}{2}\right) \partial_{\mathrm{z(t)}} \left(\mathrm{z(t)} + \frac{h}{2} f(\mathbf{s})\right) \\ &= 1 + h\partial_{\mathrm{z}} f\left(\mathrm{z(t)} + \frac{h}{2} f(\mathbf{s}), \mathrm{t} + \frac{h}{2}\right) \left(1 + \frac{h}{2} \partial_{\mathrm{z}} f\left(\mathrm{z(t)}, \mathrm{t}\right)\right) \\ &= 1 + h\partial_{\mathrm{z}} f\left(\mathrm{z(t)} + \frac{h}{2} f(\mathbf{s}), \mathrm{t} + \frac{h}{2}\right) + \frac{h^2}{2} \partial_{\mathrm{z}} f\left(\mathrm{z(t)} + \frac{h}{2} f(\mathbf{s}), \mathrm{t} + \frac{h}{2}\right) \partial_{\mathrm{z}} f\left(\mathrm{z(t)}, \mathrm{t}\right). \end{aligned}$$

On the other hand, one midpoint step of the instantaneous change of variables formula gives

$$\log p(\mathrm{z}(h), h) = \log p(\mathrm{z}(0), 0) - h\partial_{\mathrm{z}} f\left(\widetilde{\mathrm{z}}, t + \frac{h}{2}\right)$$

where $\widetilde{\mathrm{z}} = \mathrm{z}(0) + \frac{h}{2} f(\mathrm{z}(0), 0)$. Applying the exponential yields

$$p(\mathrm{z}(h), h) = \frac{p(\mathrm{z}(0), 0)}{1 + \partial_{\mathrm{z}} f\left(\widetilde{\mathrm{z}}, t + \frac{h}{2}\right) + \sum_{k=2}^{\infty} \frac{h^k \left(\partial_{\mathrm{z}} f\left(\widetilde{\mathrm{z}}, t + \frac{h}{2}\right)\right)^k}{k!}},$$

and therefore we have the same error behavior with respect to h as in Section 4.3.1. This implies that the "local density truncation error" of the midpoint-discretized instantaneous change of variables formula is generally $\mathcal{O}(h^2)$. However, the local truncation error of the midpoint method for a standard ODE is $\mathcal{O}(h^3)$, and therefore it is questionable if higher-order ODE solvers remain higher-order accurate when used in Algorithm 5. This result motivates the use of lower-order accurate adaptive ODE solvers as they use less function evaluations per step than higher-order accurate adaptive ODE solvers.

Chapter 5

Experiments

In this chapter, we investigate the influence of the dynamics f from the IVP (4.8) and the numerical ODE solver on numerical experiments for CNFs. We compare the performance of models using the dynamics architectures `concat`, `squash`, and `concatsquash` (see Figure 3.1). We train models using both `dopri5` and `ah2` (see Section 2.1.2).

In Section 5.1, we train models for density estimation on a two-dimensional toy dataset and generate samples using the trained models. In Section 5.2, we train models for density estimation on two higher-dimensional tabular datasets. In Section 5.3, we test the ability of continuous normalizing flows to improve variational autoencoders on an image dataset.

Software setup:

We use PyTorch (Paszke et al., 2017), an open source machine learning library, for all experiments. On top of PyTorch, we use the `torchdiffeq`[1] library (Chen et al., 2018) which provides ODE solvers and an implementation of the adjoint method. Our numerical experiments are based on code[2] from Grathwohl et al. (2018). Within this work, we contributed to `torchdiffeq` by implementing the adaptive Heun method `ah2`[3]. The code and scripts to reproduce our results can be found at `https://github.com/timudk/Generative-Modeling-with-NODEs`.

[1] The library can be found at `https://github.com/rtqichen/torchdiffeq`.
[2] The code can be found at `https://github.com/rtqichen/ffjord`.
[3] The merged pull request can be found at `https://github.com/rtqichen/torchdiffeq/pull/80`.

Hardware setup:

Numerical experiments on two-dimensional toy datasets are executed on two Intel Xeon E5-2630 v2 CPUs with six cores each. All other experiments are executed on one GeForce GTX 1080 Ti GPU.

5.1 Density Estimation on 2D Toy Data

As a first numerical test, we train models on the two-dimensional toy dataset `8gaussians`. A description of this dataset can be found in Appendix D. We set the learning rate to 10^{-3} and train the model for 10^4 iterations with batch size 10^2. As new training data is generated in every iteration, we basically train the models for one epoch with a training set of size 10^6. We use the Adam optimizer with a weight decay constant $c_{\text{wd}} = 10^{-5}$ (see Section 2.5.1).

Dynamics structure:

We choose the matrix and vector dimensions to be $\boldsymbol{d} = \{64, 64, 64\}$ (see Section 3.3.2) and let the hyperbolic tangent be the non-linearity of the neural network. We compute the trace explicitly, i.e., we do not make use of the trace estimation discussed in Section 4.2.3.

ODE solver setup 1:

We set the final time to $T = 0.5$. To save function evaluations in the early stage of training, we apply a tolerance scheduler

$$\texttt{tol}(n) = \max\left(\texttt{final\_tol}, \texttt{start\_tol} \exp\left(-n/\texttt{decay\_factor}\right)\right),$$

where the decay factor is computed as

$$\texttt{decay\_factor} = \frac{\texttt{warmup\_steps}}{\log \texttt{start\_tol} - \log \texttt{final\_tol}},$$

and n is the number of iterations. We experiment by using 0, 1000, and 10000 warmup steps. An example for a tolerance scheduler is visualized in Figure 5.1. For training with `dopri5`, we set the absolute and relative final tolerances to 10^{-5} and for `ah2` we set both final tolerances to 10^{-3}. These tolerances were chosen such that `dopri5` and `ah2` have

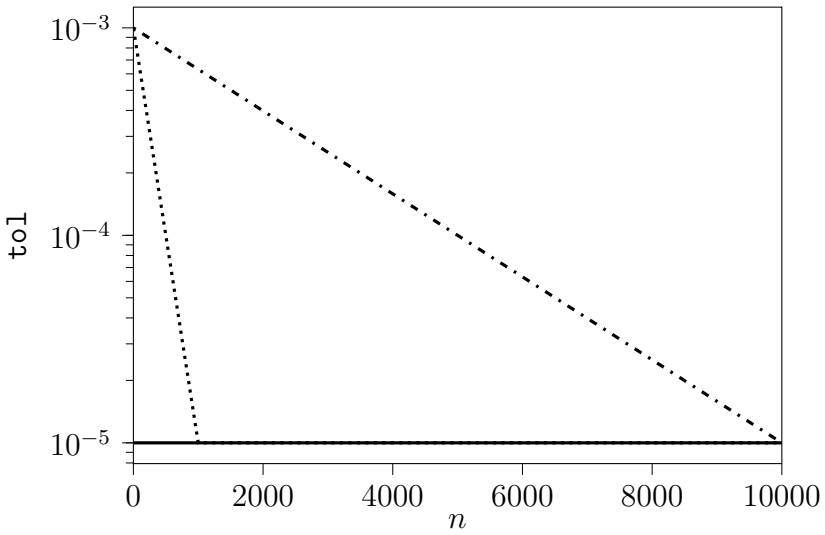


Figure 5.1: An example of a tolerance scheduler. The solid, dotted, and dash-dotted lines represent the tolerance at epoch n for 0, 1000, and 10000 warmup steps, respectively.

roughly the same number of function evaluations. The starting tolerances are chosen to be two orders of magnitude larger than the final tolerances, i.e., 10^{-3} and 10^{-1} for `dopri5` and `ah2`, respectively. For testing, we solve the arising ODEs to high precision using `dopri5` with absolute and final tolerances of 10^{-5} for all models.

Results 1:

We compare the number of forward function evaluations per iteration (NFE), number of backward function evaluations per iteration (NBE), and the negative log-likelihood (NLL) on a test set with 1000 examples; note that a minimal NLL is desirable. For each combination of dynamics architecture, solver, and number of warmup steps we train five models with different random seeds and compute the sample mean and the uncorrected sample standard deviation of the aforementioned quantities. The results for `dopri5` and `ah2` are given in Table 5.1 and Table 5.2, respectively.

For the same dynamics architecture, with varying solver and number of warmup steps, the NLL does not seem to vary much. This observation suggests that for both choices of ODE solvers and all three choices of warmup steps the models converged to their optimal solution.

Architecture	NFE	NBE	NLL
`concat`	59.1 ± 0.52	39.4 ± 0.09	$2.8577 \pm 1.14 \times 10^{-3}$
`squash`	67.0 ± 1.80	41.2 ± 0.96	$2.8524 \pm 9.70 \times 10^{-4}$
`concatsquash`	65.8 ± 3.34	43.2 ± 1.52	$2.8540 \pm 2.86 \times 10^{-3}$
`concat`	58.1 ± 0.49	38.6 ± 0.07	$2.8577 \pm 1.16 \times 10^{-3}$
`squash`	66.2 ± 1.77	40.4 ± 0.97	$\mathbf{2.8522} \pm 1.18 \times 10^{-3}$
`concatsquash`	64.8 ± 3.34	42.2 ± 1.55	$2.8540 \pm 2.84 \times 10^{-3}$
`concat`	46.4 ± 0.24	29.9 ± 0.10	$2.8575 \pm 1.17 \times 10^{-3}$
`squash`	53.8 ± 1.85	30.6 ± 0.37	$\mathbf{2.8522} \pm 1.41 \times 10^{-3}$
`concatsquash`	51.7 ± 2.63	31.6 ± 0.88	$2.8539 \pm 2.85 \times 10^{-3}$

Table 5.1: Mean and one standard deviation on `8gaussian` (test set) using `dopri5` with 0, 1000, and 10000 warmup setps (from top to bottom in this order) and final absolute/relative training tolerance 10^{-5}.

Architecture	NFE	NBE	NLL
`concat`	54.5 ± 0.26	57.4 ± 0.22	$2.8577 \pm 1.11 \times 10^{-3}$
`squash`	54.3 ± 0.57	55.2 ± 0.47	$\mathbf{2.8523} \pm 1.01 \times 10^{-3}$
`concatsquash`	55.0 ± 1.08	56.0 ± 0.87	$2.8540 \pm 2.77 \times 10^{-3}$
`concat`	52.2 ± 0.10	55.0 ± 0.13	$2.8576 \pm 9.32 \times 10^{-4}$
`squash`	52.5 ± 0.53	53.3 ± 0.40	$2.8529 \pm 6.70 \times 10^{-4}$
`concatsquash`	53.0 ± 1.05	54.0 ± 0.82	$2.8541 \pm 2.23 \times 10^{-3}$
`concat`	26.9 ± 0.05	27.2 ± 0.07	$2.8578 \pm 9.95 \times 10^{-4}$
`squash`	27.0 ± 0.42	26.6 ± 0.26	$2.8524 \pm 9.25 \times 10^{-4}$
`concatsquash`	26.8 ± 0.63	26.7 ± 0.45	$2.8526 \pm 2.64 \times 10^{-3}$

Table 5.2: Mean and one standard deviation on `8gaussian` (test set) using `ah2` with 0, 1000, and 10000 warmup steps (from top to bottom in this order) and final absolute/relative training tolerance 10^{-3}.

Models with the `squash` dynamics architecture have the lowest NLL for all combinations of ODE solvers and number of warmup steps. However, the relative difference between the largest and lowest NLL of all models is only

$$\frac{2.8578 - 2.8522}{2.8578} \approx 0.00196,$$

suggesting that the dynamics architecture does not have a significant influence on the model performance for this test.

We can see, respectively, a slight and a significant decrease in the number of function evaluations when we use 1000 and 10000 warmup steps. Using `ah2`, the number of function evaluations is less than half when using 10000 warmup steps compared to no warmup steps.

ODE solver setup 2:

We are interested if the models still converge and how the number of function evaluations behaves as we increase the final tolerances of the ODE solvers. Therefore, in this setup, we change the final tolerances of `dopri5` and `ah2` to 10^{-4} and 10^{-2}, respectively. The initial tolerances remain unchanged.

Results 2:

The results for `dopri5` and `ah2` can be found in Table 5.3 and Table 5.4, respectively. The NLL is lower for all but two combinations of dynamics architecture, number of warmup steps, and ODE solver compared to the first ODE solver setup (see Paragraph **ODE solver setup 1**).

As one might expect, the NFE and NBFE are lower for all combinations of dynamics architectures, numbers of warmup steps, and ODE solvers compared to the first ODE solver setup. The best NLL, for all models in this section, is achieved with only 16.6 NFE and 15.9 NBE (see gray cell in Table 5.4).

Architecture	NFE	NBE	NLL
`concat`	43.9 ± 0.24	27.6 ± 0.09	$2.8575 \pm 1.14 \times 10^{-3}$
`squash`	53.3 ± 2.36	29.8 ± 0.83	$2.8524 \pm 9.48 \times 10^{-4}$
`concatsquash`	51.3 ± 3.19	31.2 ± 1.31	$2.8539 \pm 2.79 \times 10^{-3}$
`concat`	43.4 ± 0.25	27.2 ± 0.06	$2.8569 \pm 1.16 \times 10^{-3}$
`squash`	52.9 ± 2.32	29.5 ± 0.87	$\mathbf{2.8521} \pm 1.36 \times 10^{-3}$
`concatsquash`	51.0 ± 2.82	30.9 ± 1.37	$2.8539 \pm 2.80 \times 10^{-3}$
`concat`	41.9 ± 0.24	25.0 ± 0.07	$2.8569 \pm 1.09 \times 10^{-3}$
`squash`	47.0 ± 2.09	26.1 ± 0.24	$2.8522 \pm 1.45 \times 10^{-3}$
`concatsquash`	44.7 ± 1.69	26.8 ± 0.60	$2.8538 \pm 2.86 \times 10^{-3}$

Table 5.3: Mean and one standard deviation on `8gaussian` (test set) using `dopri5` with 0, 1000, and 10000 `warmup_steps` (from top to bottom in this order) and final absolute/relative training tolerance 10^{-4}.

Architecture	NFE	NBE	NLL
`concat`	24.4 ± 0.08	23.2 ± 0.12	$2.8573 \pm 8.60 \times 10^{-4}$
`squash`	25.0 ± 0.32	23.9 ± 0.29	$2.8520 \pm 7.74 \times 10^{-4}$
`concatsquash`	24.6 ± 0.55	23.6 ± 0.51	$2.8536 \pm 2.61 \times 10^{-3}$
`concat`	23.7 ± 0.05	22.5 ± 0.12	$2.8577 \pm 9.35 \times 10^{-4}$
`squash`	24.4 ± 0.34	23.3 ± 0.27	$2.8524 \pm 6.39 \times 10^{-4}$
`concatsquash`	23.9 ± 0.52	23.0 ± 0.49	$2.8534 \pm 2.54 \times 10^{-3}$
`concat`	16.7 ± 0.10	16.3 ± 0.06	$2.8572 \pm 1.28 \times 10^{-3}$
`squash`	16.6 ± 0.29	15.9 ± 0.18	$\mathbf{2.8514} \pm 1.35 \times 10^{-3}$
`concatsquash`	16.1 ± 0.45	15.7 ± 0.33	$2.8520 \pm 2.20 \times 10^{-3}$

Table 5.4: Mean and one standard deviation on `8gaussian` (test set) using `ah2` with 0, 1000, and 10000 `warmup_steps` (from top to bottom in this order) and final absolute/relative training tolerance 10^{-2}.

5.1.1 Conclusion

In this section, we compared models with different dynamics and ODE solver setups on the two-dimensional toy problem `8gaussians`.

For a given dynamics architecture, the NLL was about the same for all ODE solver setups (varying tolerances and numbers of warmup steps). We showed that using higher tolerances and a larger number of warmup steps was sufficient for training CNFs on this dataset. We believe that this is due to the error introduced by the ODE solver being smaller than other errors introduced during training, e.g. the noisy gradient estimation.

In the next section, we will investigate if we can see a similar behavior for density estimation on higher-dimensional tabular datasets. In future work, we plan to explore how large the ODE solver tolerances can be set before the NLL starts to suffer.

5.2 Density Estimation on Tabular Data

In this section, we train models for density estimation on two tabular datasets: `miniboone` and `power`; a description of these datasets can be found in Papamakarios et al. (2017). Instead of using the original versions of these datasets from the UCI machine learning repository (Lichman et al., 2013), we use the pre-processed versions from Papamakarios et al. (2017)[4]. Results for density estimation on these datasets for state-of-the-art generative models are listed in Table 5.5; the results are taken from Grathwohl et al. (2018). The dimensionality of the data as well as the number of training, validation and test points for `miniboone` and `power` are summarized in Table 5.6. The number of learnable parameters for all combinations of dataset and dynamics architectures can be found in Table 5.7. For all dynamics architectures the number of parameters are roughly around 8.2×10^5 and 4.2×10^4 for `miniboone` and `power`, respectively.

For training on both datasets, we use the Adam optimizer with learning rate 10^{-3} and set the weight decay constant to $c_{\text{wd}} = 10^{-6}$. We stop training once the NLL on the validation set hat not improved for 30 consecutive validations; validation is performed after every 200-th iteration. Furthermore, we change the learning rate from 10^{-3} to 10^{-4} if the NLL on the validation set has not improved for 10 consecutive validations; the learning rate is

[4]The pre-processed datasets can be found at `https://zenodo.org/record/1161203#.XcsRnlFKg5m`

Model	miniboone	power
Dinh et al. (2017)	13.55 ± 0.49	-0.17 ± 0.01
Kingma and Dhariwal (2018)	11.35 ± 0.07	-0.17 ± 0.01
Papamakarios et al. (2017)	11.75 ± 0.44	-0.24 ± 0.01
Oliva et al. (2018)	11.01 ± 0.48	-0.48 ± 0.01
CNF (Grathwohl et al., 2018)	10.43 ± 0.04	-0.46 ± 0.01

Table 5.5: Mean and one standard deviation of NLL for state-of-the-art generative models on `miniboone` and `power` (test sets). The quantities are estimated over three runs. The results are taken from Grathwohl et al. (2018).

		Number of		
Dataset	Dimensionality	training points	validation points	testing points
`power`	6	1,659,917	184,435	204,928
`miniboone`	43	29,556	3,284	3,648

Table 5.6: Dimensionality as well as the number of training, validation, and testing points for the datasets `miniboone` and `power`.

set back to 10^{-3} once the NLL has improved on the validation set. We use batch sizes of 10^3 and 10^4 for `miniboone` and `power`, respectively.

	# Parameters	
Architecture	`miniboone`	`power`
`concat`	817087	41465
`squash`	818850	42395
`concatsquash`	820613	43325

Table 5.7: Number of parameters for density estimation models on tabular datasets.

5.2.1 Density Estimation on `miniboone`

Dynamics structure:

We choose the matrix and vector dimensions $\boldsymbol{d} = \{860, 860\}$. We use the softplus function as the non-linearity in the neural network, i.e., $g(\boldsymbol{x}) = \log(1 + \exp(\boldsymbol{x}))$. We also approximate the trace using one sample from the standard normal distribution, i.e.,

$$\operatorname{tr} \partial_{\mathbf{z}} f \approx \boldsymbol{\epsilon}^T \partial_{\mathbf{z}} f \boldsymbol{\epsilon},$$

where $\boldsymbol{\epsilon} \sim \mathcal{N}(\mathbf{0}, \boldsymbol{I})$ (see Section 4.2.3).

ODE solver setup 1:

We set the final time to $T = 1.0$. For all combinations of absolute and relative tolerances, the absolute tolerance is chosen to be two orders of magnitude smaller than the relative tolerance. Hence, for brevity, we only report the relative tolerances.

For training with `dopri5`, we train models with final relative tolerances of 10^{-6} (`dopri5_1`) and 10^{-5} (`dopri5_2`). For `ah2`, we train models with final relative tolerances of 10^{-4} (`ah2_1`) and 10^{-3} (`ah2_2`). The initial relative tolerances are set to 10^{-4} and 10^{-2} for `dopri5` and `ah2`, respectively. For testing and validation, we use `dopri5` with relative tolerance 10^{-6} and absolute tolerance 10^{-8}.

Results 1:

We compare the number of forward function evaluations per iteration (NFE), number of backward function evaluations per iteration (NBE), the number of iterations until convergence (NI), the number of total function evaluations (NTE) and the negative log-likelihood (NLL) on the test set. The sample mean and the uncorrected sample standard deviation of these quantities are computed over three runs. The results for `dopri5` can be found in Table 5.8 (`dopri5_1`) and Table 5.9 (`dopri5_2`). The results for `ah2` can be found in Table 5.10 (`ah_1`) and Table 5.11 (`ah_2`).

To make the evaluation of the results more clear, we sort the models according to their NLL in three groups. The performance classes I, II and III contain models with NLL of < 10.6, 10.6 to 11.0, and > 11.0, respectively. Groupings of the models with respect to

Architecture	NFE	NBE	NLL	NI	NTE
`concat`	104 ± 1.68	92.4 ± 0.54	$10.977 \pm 2.20 \times 10^{-1}$	$1.4 \times 10^4 \pm 1.1 \times 10^3$	2.7×10^6
`squash`	115 ± 2.52	88.3 ± 0.46	$10.583 \pm 6.05 \times 10^{-2}$	$1.9 \times 10^4 \pm 1.6 \times 10^3$	3.9×10^6
`concatsquash`	119 ± 4.96	89.5 ± 0.97	$10.566 \pm 9.90 \times 10^{-2}$	$2.2 \times 10^4 \pm 1.8 \times 10^3$	4.5×10^6
`concat`	84.0 ± 3.33	72.9 ± 2.85	$11.346 \pm 9.18 \times 10^{-2}$	$1.4 \times 10^4 \pm 1.2 \times 10^3$	2.2×10^6
`squash`	99.8 ± 3.43	74.6 ± 1.72	$10.746 \pm 1.16 \times 10^{-1}$	$1.8 \times 10^4 \pm 1.5 \times 10^3$	3.2×10^6
`concatsquash`	103 ± 6.01	76.2 ± 2.20	$\mathbf{10.475} \pm 7.08 \times 10^{-2}$	$1.9 \times 10^4 \pm 1.7 \times 10^3$	3.4×10^6

Table 5.8: Mean and one standard deviation on `miniboone` (test set) using `dopri5` with 0 and 10000 warmup steps (from top to bottom in this order). The final absolute and relative training tolerances are chosen to be 10^{-8} and 10^{-6}, respectively.

different sorting criteria can be found in Table 5.12.

As can be seen in Table 5.12, models with the `concatsquash` architecture clearly outperform models with `squash` and `concat` architectures. Models with the `squash` architecture perform still significantly better than models with `concat` architecture. The difference in performance based on the number of warmup steps does not follow a clear trend. Training with either number of warmup steps results in three models of the worst performance class (III). However, training with 0 warmup steps produces three models in performance class I, whereas training with 10000 warmup steps only results in one model of the first performance class. For `dopri5`, the ODE solver setup with lower final tolerances (`dopri5_1`) performs significantly better than the ODE solver setup with higher final tolerances (`dopri5_2`). For `ah2`, there is almost no difference in the two setups.

We now compare the models on their number of total function evaluations. Using a tolerance scheduler results in a lower NTE for ten out of the twelve combinations of ODE solvers and dynamics architectures. Increasing the final total tolerances for both `dopri5` and `ah2` results in a lower NTE for all models, however, the significance is higher for `ah2` than `dopri5`. For all six combinations of dynamics architectures and numbers of warmup steps for `ah2`, the NTE decreases by more than half when using the higher final tolerances.

Architecture	NFE	NBE	NLL	NI	NTE
`concat`	70.3 ± 1.89	61.6 ± 1.14	$11.128 \pm 3.55 \times 10^{-1}$	$1.4 \times 10^4 \pm 8.1 \times 10^2$	1.8×10^6
`squash`	82.2 ± 1.51	60.1 ± 0.21	$\mathbf{10.642} \pm 1.32 \times 10^{-2}$	$1.9 \times 10^4 \pm 1.6 \times 10^3$	3.4×10^6
`concatsquash`	85.7 ± 1.85	61.0 ± 0.95	$10.751 \pm 1.54 \times 10^{-1}$	$2.4 \times 10^4 \pm 5.0 \times 10^3$	3.5×10^6
`concat`	65.4 ± 1.08	55.4 ± 0.54	$11.229 \pm 1.06 \times 10^{-1}$	$1.6 \times 10^4 \pm 9.4 \times 10^2$	1.9×10^6
`squash`	76.5 ± 2.35	55.2 ± 0.82	$10.789 \pm 1.26 \times 10^{-1}$	$1.8 \times 10^4 \pm 1.5 \times 10^3$	2.7×10^6
`concatsquash`	79.3 ± 2.29	56.6 ± 0.73	$10.710 \pm 3.00 \times 10^{-1}$	$2.2 \times 10^4 \pm 1.8 \times 10^3$	2.9×10^6

Table 5.9: Mean and one standard deviation on `miniboone` (test set) using `dopri5` with 0 and 10000 warmup steps (from top to bottom in this order). The final absolute and relative training tolerances are chosen to be 10^{-7} and 10^{-5}, respectively.

Architecture	NFE	NBE	NLL	NI	NTE
`concat`	369 ± 6.52	402 ± 6.45	$11.220 \pm 3.55 \times 10^{-1}$	$1.5 \times 10^4 \pm 1.2 \times 10^3$	1.2×10^7
`squash`	360 ± 1.72	407 ± 1.44	$10.735 \pm 1.15 \times 10^{-1}$	$1.9 \times 10^4 \pm 1.6 \times 10^3$	1.5×10^7
`concatsquash`	403 ± 5.68	453 ± 6.78	$10.695 \pm 2.28 \times 10^{-1}$	$2.1 \times 10^4 \pm 1.9 \times 10^3$	1.8×10^7
`concat`	203 ± 14.8	225.3 ± 15.6	$10.932 \pm 3.46 \times 10^{-1}$	$1.4 \times 10^4 \pm 1.1 \times 10^3$	5.8×10^6
`squash`	252 ± 9.55	287 ± 11.2	$10.719 \pm 1.19 \times 10^{-1}$	$1.9 \times 10^4 \pm 1.6 \times 10^3$	1.0×10^7
`concatsquash`	277 ± 13.1	313 ± 15.1	$\mathbf{10.521} \pm 3.48 \times 10^{-2}$	$1.9 \times 10^4 \pm 1.4 \times 10^3$	1.1×10^7

Table 5.10: Mean and one standard deviation on `miniboone` (test set) using `ah2` with 0 and 10000 warmup steps (from top to bottom in this order). The final absolute and relative training tolerances are chosen to be 10^{-6} and 10^{-4}, respectively.

Architecture	NFE	NBE	NLL	NI	NTE
`concat`	117 ± 1.82	128 ± 1.92	$11.423 \pm 2.85 \times 10^{-1}$	$1.3 \times 10^4 \pm 1.2 \times 10^3$	3.2×10^6
`squash`	116 ± 0.81	129 ± 0.87	$10.613 \pm 2.89 \times 10^{-2}$	$1.9 \times 10^4 \pm 1.6 \times 10^3$	3.8×10^6
`concatsquash`	130 ± 2.50	143 ± 3.63	$\mathbf{10.543} \pm 9.44 \times 10^{-2}$	$1.9 \times 10^4 \pm 1.4 \times 10^3$	5.1×10^6
`concat`	84.0 ± 1.99	93.8 ± 2.05	$11.203 \pm 7.08 \times 10^{-2}$	$1.5 \times 10^4 \pm 6.6 \times 10^1$	2.6×10^6
`squash`	92.3 ± 1.91	104 ± 2.10	$10.618 \pm 1.52 \times 10^{-2}$	$1.9 \times 10^4 \pm 1.6 \times 10^3$	4.7×10^6
`concatsquash`	104 ± 3.97	116 ± 4.82	$10.569 \pm 8.24 \times 10^{-2}$	$1.9 \times 10^4 \pm 1.7 \times 10^3$	4.2×10^6

Table 5.11: Mean and one standard deviation on `miniboone` (test set) using `ah2` with 0 and 10000 warmup steps (from top to bottom in this order). The final absolute and relative training tolerances are chosen to be 10^{-5} and 10^{-3}, respectively.

	Class		
Sorting criteria	I	II	III
`dopri5_1`	3	2	1
`dopir5_2`	0	4	2
`ah2_1`	1	4	1
`ah2_2`	1	3	2
`concat`	0	2	6
`squash`	1	7	0
`concatsquash`	4	4	0
0 warmup steps	3	6	3
10^4 warmup steps	1	8	3

Table 5.12: Grouping of trained models (**ODE solver setup 1**) on `miniboone` in three performance classes (based on NLL) according to several sorting criteria.

ODE solver setup 2:

As described in Section 5.1.1, we are interested in how far we can push the ODE solver tolerances while still producing competitive results. In this setup, for `dopri5` we train models with initial and final relative tolerances of 10^{-3} and 10^{-4}, respectively. For `ah2`, we train models with initial and final relative tolerances of 10^{-2} and 10^{-1}. The absolute tolerances are again two orders of magnitude lower than the relative tolerances. We only train models using 10000 warmup steps in this setup. The testing and validation ODE solver remains unchanged.

Results 2:

The results for `dopri5` and `ah2` can be found in Table 5.13 and Table 5.14, respectively.

Similar to the first ODE solver setup, we group the models again according to their NLL. As before, the performance classes I, II, and III correspond to a NLL of < 10.6, 10.6 to 11.0, and > 11.0, respectively. The results can be found in Table 5.15.

The behavior of `dopri5` in this setup is similar to the behavior for `dopri5_2` in **ODE solver setup 1**, both producing two models of class II and one model of class I. Moreover,

Architecture	NFE	NBE	NMLL	NI	NTE
`concat`	25.5 ± 0.62	29.9 ± 0.83	$11.067 \pm 2.05 \times 10^{-1}$	$1.4 \times 10^4 \pm 7.5 \times 10^2$	8.0×10^5
`squash`	27.8 ± 0.82	32.9 ± 1.35	$10.870 \pm 1.67 \times 10^{-1}$	$1.8 \times 10^4 \pm 1.5 \times 10^3$	1.1×10^6
`concatsquash`	32.5 ± 1.64	38.9 ± 1.67	$\mathbf{10.629} \pm 1.20 \times 10^{-1}$	$1.9 \times 10^4 \pm 1.7 \times 10^3$	1.4×10^6

Table 5.13: Mean and one standard deviation on `miniboone` (test set) using `dopri5` with 10000 warmup steps. The ODE solver setup can be found in Paragraph **ODE solver setup 2**.

Architecture	NFE	NBE	NMLL	NI	NTE
`concat`	42.8 ± 1.21	38.0 ± 0.73	$11.338 \pm 5.27 \times 10^{-2}$	$1.3 \times 10^4 \pm 1.2 \times 10^3$	1.1×10^6
`squash`	49.4 ± 0.64	40.9 ± 0.46	$10.855 \pm 2.32 \times 10^{-1}$	$2.2 \times 10^4 \pm 1.8 \times 10^3$	2.0×10^6
`concatsquash`	49.5 ± 1.27	40.2 ± 0.61	$\mathbf{10.560} \pm 4.06 \times 10^{-2}$	$1.9 \times 10^4 \pm 1.7 \times 10^3$	1.7×10^6

Table 5.14: Mean and one standard deviation on `miniboone` (test set) using `ah2` with 10000 warmup steps. The ODE solver setup can be found in Paragraph **ODE solver setup 2**.

the behavior of `ah2` in this setup is similar to the behavior for `ah2_2` in the first ODE solver setup, both producing one model of each performance class. As we can see in Table 5.13 and Table 5.14, training with this ODE solver setup still results in competitive models, especially when `concatsquash` is used for the dynamics architecture.

5.2.2 Density Estimation on `power`

Dynamics structure:

We set $\boldsymbol{d} = \{60, 60, 60\}$ and use the hyperbolic tangent as the non-linearity of the neural network. Furthermore, instead of using just a single flow, we stack five flows as described in Section 4.2.2.

ODE solver setup 1:

The ODE solver setup exactly mirrors the setup from Paragraph **ODE solver setup 1** in Section 5.2.1.

	Class		
Sorting criteria	I	II	III
`dopri5`	0	2	1
`ah`	1	1	1
`concat`	0	0	2
`squash`	0	2	0
`concatsquash`	1	1	0

Table 5.15: Grouping of trained models (**ODE solver setup 2**) on `miniboone` in three performance classes (based on NLL) according to several sorting criteria.

Results 1:

We compare the same quantities as in Section 5.2.1, however, due to our restricted computational resources, we are only able to to train one model per setup. This decision can be partially justified by the small standard deviation of all state-of-the-art generative models on `power` (see Table 5.5).

We found that the number of function evaluations for training with `ah2_1` becomes prohibitively large; training on one GeForce GTX 1080 Ti GPU was not completed within seven days for any combination of dynamics architecture and number of warmup steps. Hence, we do not present any results for this ODE solver setup. The results for `ah2_2` can be found in Table 5.18. Results for `dopri5` can be found in Table 5.16 (`dopri5_1`) and Table 5.17 (`dopri5_2`).

Similar to Section 5.2.1, we sort the trained models according to their NLL in three groups. The performance classes I, II and III contain models with NLL of < -0.4, -0.4 to -0.25, and > -0.25, respectively. Groupings of the models with respect to different sorting criteria can be found in Table 5.19.

As can be seen in Table 5.19, models with the `concat` architecture clearly outperform models with `squash` and `concatsquash` architectures. There is almost no difference in performance based on the number of warmup steps. For `dopri5`, there is essentially no difference in performance for the ODE solver setups with lower (`dopri5_1`) and higher (`dopri5_2`) final tolerances.

Architecture	NFE	NBE	NLL	NI	NTE
`concat`	610	630	-4.0948×10^{-1}	3.6×10^4	4.4×10^7
`squash`	856	761	-3.1826×10^{-1}	3.6×10^4	5.8×10^7
`concatsquash`	501	488	-3.6114×10^{-1}	3.2×10^4	3.1×10^7
`concat`	671	690	$\mathbf{-4.4386 \times 10^{-1}}$	4.2×10^4	5.8×10^7
`squash`	830	736	-3.3582×10^{-1}	3.6×10^4	5.6×10^7
`concatsquash`	475	466	-3.5764×10^{-1}	3.2×10^4	3.0×10^7

Table 5.16: Results on `power` (test set) using `dopri5` with 0 and 10000 warmup steps (from top to bottom in this order). The final absolute and relative training tolerances are chosen to be 10^{-8} and 10^{-6}, respectively.

Architecture	NFE	NBE	NLL	NI	NTE
`concat`	403	405	$\mathbf{-4.1391 \times 10^{-1}}$	3.6×10^4	2.9×10^7
`squash`	644	564	-3.5614×10^{-1}	4.8×10^4	5.8×10^7
`concatsquash`	339	340	-3.4317×10^{-1}	2.9×10^4	2.0×10^7
`concat`	387	390	-4.0798×10^{-1}	3.6×10^4	2.8×10^7
`squash`	589	516	-3.4531×10^{-1}	4.2×10^4	4.7×10^7
`concatsquash`	347	334	-3.5671×10^{-1}	3.2×10^4	2.2×10^7

Table 5.17: Results on `power` (test set) using `dopri5` with 0 and 10000 warmup steps (from top to bottom in this order). The final absolute and relative training tolerances are chosen to be 10^{-7} and 10^{-5}, respectively.

We now compare the models on their number of total function evaluations. Using a tolerance scheduler results in a lower NTE for all of the nine combinations of ODE solvers and dynamics architectures. As one might expect, increasing the final tolerances for `dopri5` results in a lower or equal NTE.

ODE solver setup 2: We are interested to see how the higher-order accurate solver `dopri5` behaves as we set its tolerances to the tolerances of `ah2_2` from **ODE solver setup 1**, i.e., initial relative tolerance of 10^{-2} and final relative tolerance of 10^{-3}; the absolute tolerances are two orders of magnitude lower than the relative tolerances.

Architecture	NFE	NBE	NLL	NI	NTE
`concat`	543	535	$\mathbf{-4.3692 \times 10^{-1}}$	4.2×10^4	4.6×10^7
`squash`	731	740	-3.2749×10^{-1}	3.6×10^4	5.3×10^7
`concatsquash`	556	549	-4.0507×10^{-1}	4.2×10^4	4.7×10^7
`concat`	489	485	-4.0627×10^{-1}	3.4×10^4	3.3×10^7
`squash`	865	878	-3.3443×10^{-1}	4.2×10^4	7.4×10^7
`concatsquash`	518	508	-3.2908×10^{-1}	2.9×10^4	3.0×10^7

Table 5.18: Results on `power` (test set) using `ah2` with 0 and 10000 warmup steps (from top to bottom in this order). The final absolute and relative training tolerances are chosen to be 10^{-5} and 10^{-3}, respectively.

	Class		
Sorting criteria	I	II	III
`dopri5_1`	2	4	0
`dopir5_2`	2	4	0
`ah2_2`	3	3	0
`concat`	6	0	0
`squash`	0	6	0
`concatsquash`	1	5	0
0 warmup steps	4	5	0
10^4 warmup steps	3	6	0

Table 5.19: Grouping of trained models (**ODE solver setup 1**) on `power` in three performance classes (based on NLL) according to several sorting criteria.

Architecture	NFE	NBE	NLL	NI	NTE
`concat`	180	166	-2.6194×10^{-1}	1.6×10^4	5.7×10^6
`squash`	224	179	-1.7900×10^{-1}	2.9×10^4	1.2×10^7
`concatsquash`	178	185	$\mathbf{-3.4621 \times 10^{-1}}$	2.9×10^4	1.1×10^7
`concat`	203	207	-1.2462×10^{-1}	2.1×10^4	8.4×10^6
`squash`	186	169	-3.8044×10^{-2}	2.7×10^4	9.5×10^6
`concatsquash`	226	243	-2.3796×10^{-1}	2.4×10^4	1.1×10^7

Table 5.20: Results on `power` (test set) using `dopri5` with 0 and 10000 warmup steps (from top to bottom in this order). The final absolute and relative training tolerances are chosen to be 10^{-5} and 10^{-3}, respectively.

Results 2: Results for `dopri5` in this setup can be found in Table 5.20. For all six combinations of dynamics architectures and numbers of warmup steps, models trained with `dopri5` perform worse compared to when trained with `ah2` using the same tolerance setup. We can see that the NTE is significantly lower for `dopri5` compared to `ah2` (see Table 5.18). As described in Section 4.3, the discrepancy of the discretized instantaneous CVF and the CVF is generally $\mathcal{O}(h^2)$. We believe that `dopri5` with this tolerance setup ends up taking time steps that are too large, introducing an error that might be connected to the aforementioned discrepancy.

5.2.3 Conclusion

In this section, we compared models with different dynamics and ODE solver setups on two tabular datasets, namely `miniboone` and `power`.

We found that there was no best dynamics architecture overall: models with `concatsquash` performed best for `miniboone` and models with `concat` performed best for `power`. We showed that, as long as the high initial tolerance did not lead to divergence, training of CNFs could be made more efficient with a tolerance scheduler. We found that `dopri5` generally produced better results with lower tolerances.

On `miniboone`, using `ah2` even with very high tolerances led to CNFs that outperformed all other four state-of-the-art generative models in Table 5.5. On `power`, using `dopri5` with a low enough tolerance led to models that outperformed three out of the four state-of-the-art generative models in Table 5.5. Using `ah2` on this dataset led to CNFs that

outperformed the same three state-of-the-art generative models even for very high tolerances.

We found that `ah2` performed significantly better for higher tolerances than `dopri5`, however, training CNFs with `dopri5` using lower tolerances often still required less total number of function evaluations than training with `ah2` using higher tolerances.

5.3 Variational Inference on an Image Dataset

In Section 4.1.4, we reviewed how to use generative models to improve variational autoencoders. In this section, we investigate the influence of the dynamics architecture and the numerical ODE solver for improving VAEs with continuous normalizing flows on the image dataset `frey_faces`.

The `frey_faces` dataset contains 1965 grayscale images of size 20 by 28. The images are sequential frames of a short video showing Brendan Frey's face[5]. The dataset is split into 1565 training, 200 validation, and 200 test images. Results for VAEs with state-of-the-art generative models as well as for a VAE with planar flow (described in Section 4.1.2) and for a VAE without any flow are shown in Table 5.21.

For all experiments we use the same encoder $q(\mathbf{z} \mid \mathbf{x}, \boldsymbol{\phi})$ and decoder $p(\mathbf{x} \mid \mathbf{z}, \boldsymbol{\theta})$ architectures as Berg et al. (2018). As suggested by Bowman et al. (2015) and Sønderby et al. (2016), we maximize the modified negative free energy

$$-\mathcal{F}^{\mathrm{NF}}_{\mathrm{mod}}(\mathbf{x}) = -\min\left(1, \frac{\texttt{epoch}}{100}\right) D_{\mathrm{KL}}(q^{\mathrm{NF}}_K(\mathbf{z} \mid \mathbf{x}, \boldsymbol{\phi}, \boldsymbol{\lambda}) \parallel p(\mathbf{z} \mid \boldsymbol{\theta})) + \mathbb{E}_q[\log p(\mathbf{x} \mid \mathbf{z}, \boldsymbol{\theta})],$$

where `epoch` is the number of epochs, instead of the negative free energy (4.7) in training. The dimension of the latent variable space is $d_{\mathbf{z}} = 64$, i.e., $\mathbf{z} \in \mathbb{R}^{d_{\mathbf{z}}}$. Training is stopped after 2000 epochs or if the negative free energy on the validation set has not improved in 100 consecutive epochs.

As discussed in Section 4.1.4, the flow parameters $\boldsymbol{\lambda}$ can be modeled as a function of the input $\mathbf{x}$. Grathwohl et al. (2018) reported that letting all flow parameters be dependent on $\mathbf{x}$ can lead to differential equations that are too difficult to numerically integrate and propose to only let a subset of $\boldsymbol{\lambda}$ be data-dependent. In practice, this can, for example,

[5]The dataset can be downloaded at `https://github.com/riannevdberg/sylvester-flows/blob/master/data/Freyfaces/freyfaces.pkl`.

Model	`frey_faces`
No flow	$1.76 \times 10^3 \pm 7.8$
Planar flow	$1.71 \times 10^3 \pm 23$
Kingma et al. (2016)	$1.74 \times 10^3 \pm 19$
Berg et al. (2018)	$1.73 \times 10^3 \pm 16$
CNF (Grathwohl et al., 2018)	$1.70 \times 10^3 \pm 3.9$

Table 5.21: Mean and one standard deviation (estimated over three runs) of negative free energy (4.7) on `frey_faces`. The results were taken form Grathwohl et al. (2018).

be done by replacing $\boldsymbol{A}$ and $\boldsymbol{b}$ in the building blocks of the dynamics architectures (see Table 3.1) with

$$\boldsymbol{A}(\mathbf{x}) = \boldsymbol{A}_{\text{fixed}} + \hat{\boldsymbol{U}}(\mathbf{x})\hat{\boldsymbol{V}}^T(\mathbf{x}),$$

and

$$\boldsymbol{b}(\mathbf{x}) = \boldsymbol{b}_{\text{fixed}} + \hat{\boldsymbol{b}}(\mathbf{x}),$$

respectively. The matrices $\hat{\boldsymbol{U}}$ and $\hat{\boldsymbol{V}}$ are of size $d_{\text{out}} \times k$ and $d_{\text{in}} \times k$, respectively. The product $\hat{\boldsymbol{U}}(\mathbf{x})\hat{\boldsymbol{V}}^T(\mathbf{x})$ can be interpreted as a data-dependent low-rank update of the global weight matrix $\boldsymbol{A}$.

Similar to Grathwohl et al. (2018), we use a modification of Adam, called AdaMax (Kingma and Ba, 2014), as the optimizer for the experiments in this section. AdaMax modifies the update rule of Adam (see Equation (2.12)) as

$$\boldsymbol{\theta}_{t+1}^{\text{AdaMax}} = \boldsymbol{\theta}_t^{\text{AdaMax}} - \frac{\eta}{\boldsymbol{u}_t} \odot \hat{\boldsymbol{m}}_t,$$

where $\boldsymbol{u}_t = \max\left(\beta_2 \boldsymbol{u}_{t-1}, |\boldsymbol{g}_t| + \boldsymbol{\epsilon}^{\text{AdaMax}}\right)$ and the maximum is applied elementwise. AdaMax initializes $\boldsymbol{u}_0 = \mathbf{0}$ and we choose $\boldsymbol{\epsilon}^{\text{AdaMax}} = \mathbf{10^{-7}}$. The learning rate is set to 0.0005 and we do not use any weight decay. The batch size is set to 100.

Dynamics structure:

We follow the recommended dynamics structure from Grathwohl et al. (2018); the matrix and vector dimensions are chosen as $\boldsymbol{d} = \{512, 512\}$. We use the softplus function as the

non-linearity in the neural network. The rank of the low-rank update is set to $k = 20$. For the experiments in this section, we stack two CNFs (see Section 4.2.2). The trace is approximated using one sample from a standard normal distribution, i.e.,

$$\operatorname{tr} \partial_{\mathbf{z}} f \approx \boldsymbol{\epsilon}^T \partial_{\mathbf{z}} f \boldsymbol{\epsilon},$$

where $\boldsymbol{\epsilon} \sim \mathcal{N}(\mathbf{0}, \boldsymbol{I})$ (see Section 4.2.3).

ODE solver setup:

For both ODE solvers we train models with 0 and 1000 warmup steps. The initial tolerances are set to 10^{-3} and 10^{-1} for `dopri5` and `ah2`, respectively. We train models for two different sets of final tolerances for each ODE solver: 10^{-5} (`dopri5_1`)/10^{-4} (`dopri5_2`) for `dopri5` and 10^{-3}(`ah2_1`)/10^{-2}(`ah2_2`) for `ah2`. The final time for both solvers is set to $T = 0.5$. Testing and validation is done using `dopri5` with tolerances of 10^{-5}.

Results:

We compare the number of forward function evaluations per iteration (NFE), number of backward function evaluations per iteration (NBE), the number of iterations until convergence (NI), the number of total function evaluations (NTE) and the negative free energy (NFEG) on the test set. The sample mean and the uncorrected sample standard deviation of the aforementioned quantities are estimated over three runs. The results for `dopri5` can be found in Table 5.22 (`dopri5_1`) and Table 5.23 (`dopri5_2`). The results for `ah2` can be found in Table 5.24 (`ah_1`) and Table 5.25[6] (`ah_2`).

We sort the models according to their performance in two groups. The performance classes I and II contain models with NFEG of $< 1.740 \times 10^1$ and $\geq 1.740 \times 10^1$, respectively. Groupings of the models with respect to different sorting criteria can be found in Table 5.26.

As can be seen in Table 5.26, models with `squash` and `concatsquash` dynamics architectures perform better than models with `concat` dynamics architecture. On the other hand, the best result overall is achieved with a model having the `concat` dynamics architecture. The difference in performance based on the number of warmup steps is only apparent for `ah2`; except for one model, models trained with `ah2` perform worse when we

[6]Mean and standard deviation of the model with † is estimated only over two runs as one run led to a step size close to zero.

Architecture	NFE	NBE	NFEG	NI	NTE
`concat`	69.4 ± 1.73	81.7 ± 2.75	$1.741 \times 10^3 \pm 2.7 \times 10^1$	$1.3 \times 10^3 \pm 1.4 \times 10^2$	2.0×10^5
`squash`	77.4 ± 5.27	94.1 ± 7.70	$1.727 \times 10^3 \pm 1.4 \times 10^1$	$1.3 \times 10^3 \pm 1.0 \times 10^2$	2.3×10^5
`concatsquash`	77.8 ± 2.67	95.3 ± 7.80	$1.734 \times 10^3 \pm 4.0 \times 10^1$	$1.3 \times 10^3 \pm 1.3 \times 10^2$	2.2×10^5
`concat`	60.0 ± 3.47	69.9 ± 3.75	$1.742 \times 10^3 \pm 2.7 \times 10^1$	$1.4 \times 10^3 \pm 1.8 \times 10^2$	1.8×10^5
`squash`	65.7 ± 2.21	76.5 ± 3.10	$\mathbf{1.724 \times 10^3} \pm 1.4 \times 10^1$	$1.4 \times 10^3 \pm 2.1 \times 10^2$	1.9×10^5
`concatsquash`	64.4 ± 3.03	79.7 ± 8.56	$1.736 \times 10^3 \pm 4.3 \times 10^1$	$1.2 \times 10^3 \pm 1.3 \times 10^2$	1.7×10^5

Table 5.22: Mean and one standard deviation on `frey_faces` (test set) using `dopri5` with 0 and 1000 warmup steps (from top to bottom in this order). The final absolute and relative training tolerances are chosen to be 10^{-5}.

Architecture	NFE	NBE	NFEG	NI	NTE
`concat`	52.8 ± 1.36	57.6 ± 1.02	$1.738 \times 10^3 \pm 2.8 \times 10^1$	$1.3 \times 10^3 \pm 1.4 \times 10^2$	1.4×10^5
`squash`	57.0 ± 2.69	65.4 ± 3.05	$1.723 \times 10^3 \pm 1.9 \times 10^1$	$1.3 \times 10^3 \pm 1.3 \times 10^2$	1.6×10^5
`concatsquash`	57.5 ± 1.96	70.2 ± 4.97	$1.731 \times 10^3 \pm 4.0 \times 10^1$	$1.2 \times 10^3 \pm 9.9 \times 10^1$	1.6×10^5
`concat`	49.1 ± 1.89	55.1 ± 1.22	$1.741 \times 10^3 \pm 2.5 \times 10^1$	$1.3 \times 10^3 \pm 1.2 \times 10^2$	1.3×10^5
`squash`	53.1 ± 1.30	58.1 ± 1.60	$\mathbf{1.724 \times 10^3} \pm 1.3 \times 10^1$	$1.3 \times 10^3 \pm 2.1 \times 10^2$	1.4×10^5
`concatsquash`	53.5 ± 2.04	63.3 ± 6.19	$1.735 \times 10^3 \pm 3.7 \times 10^1$	$1.2 \times 10^3 \pm 1.7 \times 10^2$	1.4×10^5

Table 5.23: Mean and one standard deviation on `frey_faces` (test set) using `dopri5` with 0 and 1000 warmup steps (from top to bottom in this order). The final absolute and relative training tolerances are chosen to be 10^{-4}.

use warmup steps. For `dopri5`, there is almost no difference for the two setups of tolerances. The ODE solver `ah2_1` produces models with a similar performance to the models produced by `dopri5`, however, `ah_2` only produces models of the second performance class.

We now compare the models on their number of total function evaluations. Using a tolerance scheduler results in a lower NTE for eleven out of the twelve combinations of ODE solvers and dynamics architectures. Increasing the final total tolerances for both `dopri5` and `ah2` results in a lower NTE for all models. Similar to Section 5.2.1, the decrease of NTE is more significant for `ah2` than for `dopri5`.

Architecture	NFE	NBE	NFEG	NI	NTE
`concat`	57.4 ± 4.21	71.7 ± 4.63	$1.728 \times 10^3 \pm 1.8 \times 10^1$	$1.3 \times 10^3 \pm 1.7 \times 10^2$	1.7×10^5
`squash`	64.3 ± 1.67	74.7 ± 3.71	$1.723 \times 10^3 \pm 1.9 \times 10^1$	$1.2 \times 10^3 \pm 1.2 \times 10^2$	1.7×10^5
`concatsquash`	67.6 ± 2.87	77.3 ± 3.07	$1.736 \times 10^3 \pm 3.9 \times 10^1$	$1.2 \times 10^3 \pm 1.2 \times 10^2$	1.8×10^5
`concat`	33.6 ± 1.75	42.3 ± 1.68	$\mathbf{1.718 \times 10^3} \pm 1.2 \times 10^1$	$1.2 \times 10^3 \pm 1.9 \times 10^1$	8.7×10^4
`squash`	52.1 ± 8.57	59.9 ± 6.88	$1.812 \times 10^3 \pm 7.2 \times 10^1$	$1.3 \times 10^3 \pm 1.5 \times 10^2$	1.5×10^5
`concatsquash`	48.3 ± 3.80	56.0 ± 3.39	$1.839 \times 10^3 \pm 1.2 \times 10^2$	$1.2 \times 10^3 \pm 1.7 \times 10^2$	1.3×10^5

Table 5.24: Mean and one standard deviation on `frey_faces` (test set) using `ah2` with 0 and 1000 warmup steps (from top to bottom in this order). The final absolute and relative training tolerances are chosen to be 10^{-3}.

Architecture	NFE	NBE	NFEG	NI	NTE
`concat`	25.4 ± 1.63	32.5 ± 2.40	$1.742 \times 10^3 \pm 2.8 \times 10^1$	$1.3 \times 10^3 \pm 1.5 \times 10^2$	7.3×10^4
`squash`	25.5 ± 1.68	32.6 ± 2.34	$\mathbf{1.741 \times 10^3} \pm 2.4 \times 10^1$	$1.3 \times 10^3 \pm 1.3 \times 10^2$	7.6×10^4
`concatsquash`	33.5 ± 0.26	59.3 ± 23.2	$1.782 \times 10^3 \pm 3.4 \times 10^1$	$1.1 \times 10^3 \pm 1.7 \times 10^2$	1.0×10^5
`concat`	24.7 ± 3.73	32.1 ± 5.94	$1.773 \times 10^3 \pm 1.3 \times 10^1$	$1.1 \times 10^3 \pm 2.8 \times 10^2$	6.2×10^4
`squash`†	34.1 ± 5.45	44.8 ± 0.10	$1.827 \times 10^3 \pm 1.0 \times 10^1$	$1.3 \times 10^3 \pm 1.5 \times 10^2$	1.1×10^5
`concatsquash`	33.7 ± 1.82	42.0 ± 4.09	$1.798 \times 10^3 \pm 2.5 \times 10^1$	$1.3 \times 10^3 \pm 1.1 \times 10^2$	9.8×10^4

Table 5.25: Mean and one standard deviation (model with † is estimated only over two runs as one run led to a step size close to zero) on `frey_faces` (test set) using `ah2` with 0 and 1000 warmup steps (from top to bottom in this order). The final absolute and relative training tolerances are chosen to be 10^{-2}.

	Class	
Sorting criteria	I	II
`dopri5_1`	4	2
`dopir5_2`	5	1
`ah2_1`	4	2
`ah2_2`	0	6
`concat`	3	5
`squash`	5	3
`concatsquash`	5	3
0 warmup steps	7	5
10^4 warmup steps	6	6

Table 5.26: Grouping of trained models on `frey_faces` in three performance classes (based on NFEG) according to several sorting criteria.

5.3.1 Conclusion

In this section, we compared models with different dynamics and ODE solver setups for variational inference on the `frey_faces` dataset.

For a given dynamics architecture, models trained with `dopri5` generally performed better than models trained with `ah2`. There was no clear trend in the difference of performance based on the dynamics architecture. When using `dopri5`, training could be made more efficient using a tolerance scheduler. For `ah2`, using a tolerance scheduler generally worsened model performance.

The `frey_faces` dataset with 1565 training examples is rather small, leading to a high influence of the hyperparameters on the model performance. In future work, we plan to explore how the dynamics and the ODE solver affect variational inference for larger datasets.

Chapter 6

Conclusion and Outlook

This thesis considers neural ordinary differential equations and in particular their application to generative modeling. One of our main contributions is the derivation of the adjoint sensitivity method for NODEs and CNFs within a constrained optimization framework. Furthermore, we provide a comprehensive and self-contained introduction to generative modeling (with NODEs). In this chapter, we summarize our theoretical and numerical results. Moreover, we give an outlook on potential future work.

6.1 Theoretical Results

We presented theoretical results for both NODEs and CNFs. We showed that NODEs with linear dynamics result in a linear transformation of the initial value[1], and therefore no non-linear relations can be encoded in the "output" of linear NODEs. We further showed that we can construct a tent function such that there is no autonomous NODE that can converge to it; on the other hand, there is always a non-autonomous NODE that can converge to this tent function. Moreover, we proved that modeling the dynamics of a NODE using a particular hypernetwork leads to universal function approximation in the discretized case[1].

For CNFs, we quantified the discrepancy between the CVF and the discretized instantaneous CVF for two ODE solvers. We illustrated this problem for a one-dimensional normal distribution using Euler's method. We further showed that using trace estimation, initially used with the purpose of saving computational cost, can increase the aforementioned discrepancy.

[1] This was shown for Euler's method but could be extended to other explicit discretization methods.

6.2 Numerical Results

We performed numerical tests for CNFs on density estimation and variational inference tasks. The experiments focused on the performance behavior of CNFs with varying ODE solver setup and dynamics architecture.

We found that using a tolerance scheduler that exponentially reduces the ODE solver tolerances generally resulted in more efficient training of CNFs; less function evaluations were needed in the training stage to achieve roughly the same model performance. This result held for both the higher-order accurate ODE solver `dopri5` and the lower-order accurate ODE solver `ah2`. Furthermore, we showed that `ah2` could produce CNFs with competitive performance even for very high solver tolerances. The higher-order accurate ODE solver `dopri5` generally could not produce competitive results with high solver tolerances, however, training with `dopri5` using lower tolerances often still required less function evaluations than training with `ah2` using higher tolerances.

For variational inference, we found that tolerance schedulers still resulted in more efficient training, however, the results were not as clear as for the density estimation tasks. We are not able to make any other precise statement of the behavior of CNFs with varying ODE solver setup and dynamics architecture for variational inference. As a result, we plan to explore CNFs for this task on larger datasets, where the choice of hyperparameters has less influence on model performance.

6.3 Future Work

In future work, we plan to numerically investigate the class of NODEs that leads to universal function approximation. To gain a better understanding, we will first explore this class of NODEs for supervised learning tasks. Furthermore, we plan to study the affect of the aforementioned discrepancy between the discretized instantaneous CVF and the CVF on training of CNFs. In our numerical experiments, we saw that the choice of ODE solver tolerances significantly influenced the cost for training CNFs. In future work, we plan to explore methods that adaptively change the ODE solver tolerances. In doing so, we hope to make training of CNFs more efficient. Since both ODE solvers, `dopri5` and `ah2`, needed many function evaluations per iteration, we also plan to explore training of CNFs using implicit ODE solvers. Lastly, we plan to investigate trace free CNFs, a natural way to avoid the high costs of full trace computation and the noise accompanied with trace estimation.

References

Ackley, D. H., Hinton, G. E., and Sejnowski, T. J. (1985). A Learning Algorithm for Boltzmann Machines. *Cognitive science*, 9(1):147–169.

Arjovsky, M., Chintala, S., and Bottou, L. (2017). Wasserstein Generative Adversarial Networks. In *International conference on machine learning*, pages 214–223.

Arora, R., Basu, A., Mianjy, P., and Mukherjee, A. (2018). Understanding Deep Neural Networks with Rectified Linear Units. In *International Conference on Learning Representations*.

Berg, R. v. d., Hasenclever, L., Tomczak, J. M., and Welling, M. (2018). Sylvester Normalizing Flows for Variational Inference. *arXiv preprint arXiv:1803.05649*.

Betts, J. T. and Campbell, S. L. (2005). Discretize Then Optimize. *Mathematics for industry: challenges and frontiers*, pages 140–157.

Bishop, C. M. (2006). *Pattern Recognition and Machine Learning*. Springer.

Blei, D. M., Kucukelbir, A., and McAuliffe, J. D. (2017). Variational Inference: A Review for Statisticians. *Journal of the American Statistical Association*, 112(518):859–877.

Bowman, S. R., Vilnis, L., Vinyals, O., Dai, A. M., Jozefowicz, R., and Bengio, S. (2015). Generating Sentences From a Continuous Space. *arXiv preprint arXiv:1511.06349*.

Brémaud, P. (2013). *Markov Chains: Gibbs Fields, Monte Carlo Simulation, and Queues*, volume 31. Springer Science & Business Media.

Brock, A., Donahue, J., and Simonyan, K. (2018). Large Scale GAN Training for High Fidelity Natural Image Synthesis. *arXiv preprint arXiv:1809.11096*.

Brock, A., Lim, T., Ritchie, J. M., and Weston, N. J. (2017). Neural Photo Editing with Introspective Adversarial Networks. In *5th International Conference on Learning Representations 2017*.

Bryson, A. E. (1961). A Gradient Method for Optimizing Multi-Stage Allocation Processes. In *Proc. Harvard Univ. Symposium on digital computers and their applications*, volume 72.

Butcher, J. C. (2016). *Numerical Methods for Ordinary Differential Equations*. John Wiley & Sons.

Chen, T. Q., Rubanova, Y., Bettencourt, J., and Duvenaud, D. K. (2018). Neural Ordinary Differential Equations. In *Advances in neural information processing systems*, pages 6571–6583.

Cohen, N., Sharir, O., and Shashua, A. (2016). On the Expressive Power of Deep Learning: A Tensor Analysis. In *Conference on Learning Theory*, pages 698–728.

Cybenko, G. (1989). Approximation by Superpositions of a Sigmoidal Function. *Mathematics of control, signals and systems*, 2(4):303–314.

Dinh, L., Krueger, D., and Bengio, Y. (2014). NICE: Non-Linear Independent Components Estimation. *arXiv preprint arXiv:1410.8516*.

Dinh, L., Sohl-Dickstein, J., and Bengio, S. (2017). Density Estimation Using Real NVP. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*.

Doersch, C. (2016). Tutorial on Variational Autoencoders. *arXiv preprint arXiv:1606.05908*.

Dormand, J. R. and Prince, P. J. (1980). A Family of Embedded Runge–Kutta Formulae. *Journal of computational and applied mathematics*, 6(1):19–26.

Dreyfus, S. (1962). The Numerical Solution of Variational Problems. *Journal of Mathematical Analysis and Applications*, 5(1):30–45.

E, W. (2017). A Proposal on Machine Learning via Dynamical Systems. *Communications in Mathematics and Statistics*, 5(1):1–11.

Eldan, R. and Shamir, O. (2016). The Power of Depth for Feedforward Neural Networks. In *Conference on learning theory*, pages 907–940.

Frey, B. J., Brendan, J. F., and Frey, B. J. (1998). *Graphical Models for Machine Learning and Digital Communication*. MIT press.

Germain, M., Gregor, K., Murray, I., and Larochelle, H. (2015). MADE: Masked Autoencoder for Distribution Estimation. In *International Conference on Machine Learning*, pages 881–889.

Goodfellow, I. (2016). NIPS 2016 Tutorial: Generative Adversarial Networks. *arXiv preprint arXiv:1701.00160*.

Goodfellow, I., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A., and Bengio, Y. (2014). Generative Adversarial Nets. In *Advances in neural information processing systems*, pages 2672–2680.

Grathwohl, W., Chen, R. T., Betterncourt, J., Sutskever, I., and Duvenaud, D. (2018). FFJORD: Free-Form Continuous Dynamics for Scalable Reversible Generative Models. *arXiv preprint arXiv:1810.01367*.

Graves, A. (2013). Generating Sequences with Recurrent Neural Networks. *arXiv preprint arXiv:1308.0850*.

Gulrajani, I., Ahmed, F., Arjovsky, M., Dumoulin, V., and Courville, A. C. (2017). Improved Training of Wasserstein GANs. In *Advances in neural information processing systems*, pages 5767–5777.

Ha, D., Dai, A., and Le, Q. V. (2016). Hypernetworks. *arXiv preprint arXiv:1609.09106*.

Haber, E. and Ruthotto, L. (2017). Stable Architectures for Deep Neural Networks. *Inverse Problems*, 34(1):014004.

Hairer, E., Nørsett, S. P., and Wanner, G. (1991). *Solving Ordinary Differential Equations 1, Nonstiff problems*. Springer-Vlg.

Harville, D. A. (1998). Matrix Algebra from a Statistician's Perspective.

He, K., Zhang, X., Ren, S., and Sun, J. (2016a). Deep Residual Learning for Image Recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778.

He, K., Zhang, X., Ren, S., and Sun, J. (2016b). Identity Mappings in Deep Residual Networks. In *European conference on computer vision*, pages 630–645. Springer.

Hinton, G. E. (2012). A Practical Guide to Training Restricted Boltzmann Machines. In *Neural networks: Tricks of the trade*, pages 599–619. Springer.

Hinton, G. E., Osindero, S., and Teh, Y.-W. (2006). A Fast Learning Algorithm for Deep Belief Nets. *Neural computation*, 18(7):1527–1554.

Hinton, G. E. and Sejnowski, T. J. (1983). Optimal Perceptual Inference. In *Proceedings of the IEEE conference on Computer Vision and Pattern Recognition*, pages 448–453. Citeseer.

Hinze, M. and Rösch, A. (2012). Discretization of Optimal Control Problems. In *Constrained Optimization and Optimal Control for Partial Differential Equations*, pages 391–430. Springer.

Hochreiter, S. and Schmidhuber, J. (1997). Long Short-Term Memory. *Neural computation*, 9(8):1735–1780.

Hornik, K., Stinchcombe, M., and White, H. (1989). Multilayer Feedforward Networks are Universal Approximators. *Neural networks*, 2(5):359–366.

Hutchinson, M. F. (1990). A Stochastic Estimator of the Trace of the Influence Matrix for Laplacian Smoothing Splines. *Communications in Statistics-Simulation and Computation*, 19(2):433–450.

Karras, T., Aila, T., Laine, S., and Lehtinen, J. (2017). Progressive Growing of GANs for Improved Quality, Stability, and Variation. *arXiv preprint arXiv:1710.10196*.

Kelley, H. J. (1960). Gradient Theory of Optimal Flight Paths. *Ars Journal*, 30(10):947–954.

Khalil, H. K. (2002). Nonlinear Systems. *Upper Saddle River*.

Kindermann, R. (1980). Markov Random Fields and their Applications. *American mathematical society*.

Kingma, D. P. and Ba, J. (2014). Adam: A Method for Stochastic Optimization. *arXiv preprint arXiv:1412.6980*.

Kingma, D. P. and Dhariwal, P. (2018). GLOW: Generative Flow with Invertible 1x1 Convolutions. In *Advances in Neural Information Processing Systems*, pages 10215–10224.

Kingma, D. P., Salimans, T., Jozefowicz, R., Chen, X., Sutskever, I., and Welling, M. (2016). Improved Variational Inference with Inverse Autoregressive Flow. In Lee, D. D., Sugiyama, M., Luxburg, U. V., Guyon, I., and Garnett, R., editors, *Advances in Neural Information Processing Systems 29*, pages 4743–4751. Curran Associates, Inc.

Kingma, D. P. and Welling, M. (2013). Auto-Encoding Variational Bayes. *arXiv preprint arXiv:1312.6114*.

LeCun, Y., Touresky, D., Hinton, G., and Sejnowski, T. (1988). A Theoretical Framework for Back-Propagation. In *Proceedings of the 1988 connectionist models summer school*, volume 1, pages 21–28. CMU, Pittsburgh, Pa: Morgan Kaufmann.

Ledig, C., Theis, L., Huszár, F., Caballero, J., Cunningham, A., Acosta, A., Aitken, A., Tejani, A., Totz, J., Wang, Z., et al. (2017). Photo-Realistic Single Image Super-Resolution Using a Generative Adversarial Network. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4681–4690.

Liang, S. and Srikant, R. (2016). Why Deep Neural Networks for Function Approximation? *arXiv preprint arXiv:1610.04161*.

Lichman, M. et al. (2013). UCI Machine Learning Repository.

Lin, H. and Jegelka, S. (2018). ResNet with One-Neuron Hidden Layers is a Universal Approximator. In *Advances in Neural Information Processing Systems*, pages 6169–6178.

Linnainmaa, S. (1976). Taylor Expansion of the Accumulated Rounding Error. *BIT Numerical Mathematics*, 16(2):146–160.

Loshchilov, I. and Hutter, F. (2017). Fixing Weight Decay Regularization in Adam. *arXiv preprint arXiv:1711.05101*.

Lu, Y., Zhong, A., Li, Q., and Dong, B. (2018). Beyond Finite Layer Neural Networks: Bridging Deep Architectures and Numerical Differential Equations.

Miyato, T., Kataoka, T., Koyama, M., and Yoshida, Y. (2018). Spectral Normalization for Generative Adversarial Networks. *arXiv preprint arXiv:1802.05957*.

Nielsen, M. A. (2015). *Neural Networks and Deep Learning*, volume 25. Determination press San Francisco, CA, USA:.

Oliva, J., Dubey, A., Zaheer, M., Poczos, B., Salakhutdinov, R., Xing, E., and Schneider, J. (2018). Transformation Autoregressive Networks. In *International Conference on Machine Learning*, pages 3895–3904.

Paisley, J., Blei, D. M., and Jordan, M. I. (2012). Variational Bayesian Inference with Stochastic Search.

Papamakarios, G., Pavlakou, T., and Murray, I. (2017). Masked Autoregressive Flow for Density Estimation. In *Advances in Neural Information Processing Systems*, pages 2338–2347.

Paszke, A., Gross, S., Chintala, S., Chanan, G., Yang, E., DeVito, Z., Lin, Z., Desmaison, A., Antiga, L., and Lerer, A. (2017). Automatic Differentiation in PyTorch.

Pontryagin, L. S., Mishchenko, E., Boltyanskii, V., and Gamkrelidze, R. (1962). The Mathematical Theory of Optimal Processes.

Pu, Y., Gan, Z., Henao, R., Yuan, X., Li, C., Stevens, A., and Carin, L. (2016). Variational Autoencoder for Deep Learning of Emages, Labels and Captions. In *Advances in neural information processing systems*, pages 2352–2360.

Rezende, D. and Mohamed, S. (2015). Variational Inference with Normalizing Flows. In *International Conference on Machine Learning*, pages 1530–1538.

Rezende, D. J., Mohamed, S., and Wierstra, D. (2014). Stochastic Backpropagation and Approximate Inference in Deep Generative Models. In *International Conference on Machine Learning*, pages 1278–1286.

Rezende, D. J., Racanière, S., Higgins, I., and Toth, P. (2019). Equivariant Hamiltonian Flows. *arXiv preprint arXiv:1909.13739*.

Rolnick, D. and Tegmark, M. (2017). The Power of Deeper Networks for Expressing Natural Functions. *arXiv preprint arXiv:1705.05502*.

Rudin, W. (2006). *Real and Complex Analysis*. Tata McGraw-hill education.

Rumelhart, D. E., Hinton, G. E., and Williams, R. J. (1985). Learning Internal Representations by Error Propagation. Technical report, California Univ San Diego La Jolla Inst for Cognitive Science.

Rumelhart, D. E., Hinton, G. E., Williams, R. J., et al. (1988). Learning Representations by Back-Propagating Errors. *Cognitive modeling*, 5(3):1.

Salakhutdinov, R. and Hinton, G. (2009). Deep Boltzmann Machines. In *Artificial intelligence and statistics*, pages 448–455.

Salimans, T., Goodfellow, I., Zaremba, W., Cheung, V., Radford, A., and Chen, X. (2016). Improved Techniques for Training GANs. In *Advances in neural information processing systems*, pages 2234–2242.

Samuel, A. L. (1967). Some Studies in Machine Learning Using the Game of Checkers. IIRecent progress. *IBM Journal of research and development*, 11(6):601–617.

Shampine, L. F. (1986). Some Practical Runge–Kutta Formulas. *Mathematics of Computation*, 46(173):135–150.

Shampine, L. F. and Reichelt, M. W. (1997). The Matlab ODE Suite. *SIAM journal on scientific computing*, 18(1):1–22.

Shannon, C. E. (1948). A Mathematical Theory of Communication. *Bell system technical journal*, 27(3):379–423.

Skilling, J. (1989). The Eigenvalues of Mega-Dimensional Matrices. In *Maximum Entropy and Bayesian Methods*, pages 455–466. Springer.

Smolensky, P. (1986). Information Processing in Dynamical Systems: Foundations of Harmony Theory. Technical report, Colorado Univ at Boulder Dept of Computer Science.

Sønderby, C. K., Raiko, T., Maaløe, L., Sønderby, S. K., and Winther, O. (2016). Ladder Variational Autoencoders. In *Advances in neural information processing systems*, pages 3738–3746.

Tabak, E. G. and Turner, C. V. (2013). A Family of Nonparametric Density Estimation Algorithms. *Communications on Pure and Applied Mathematics*, 66(2):145–164.

Tabak, E. G., Vanden-Eijnden, E., et al. (2010). Density Estimation by Dual Ascent of the Log-Likelihood. *Communications in Mathematical Sciences*, 8(1):217–233.

Telgarsky, M. (2016). Benefits of Depth in Neural Networks. In *Conference on Learning Theory*, pages 1517–1539.

Trefethen, L. N. and Bau III, D. (1997). *Numerical Linear Algebra*, volume 50. Siam.

Uria, B., Côté, M.-A., Gregor, K., Murray, I., and Larochelle, H. (2016). Neural Autoregressive Distribution Estimation. *The Journal of Machine Learning Research*, 17(1):7184–7220.

Van Oord, A., Kalchbrenner, N., and Kavukcuoglu, K. (2016). Pixel Recurrent Neural Networks. In *International Conference on Machine Learning*, pages 1747–1756.

Zhu, J.-Y., Krähenbühl, P., Shechtman, E., and Efros, A. A. (2016). Generative Visual Manipulation on the Natural Image Manifold. *arXiv preprint arXiv:1609.03552*.

APPENDICES

Appendix A

Backpropagation Through Ordinary Differential Equation Solvers

A.1 Euler's Network

A.1.1 A Proof of Equation (3.9)

For notational convenience, we write $\boldsymbol{z}^n \coloneqq \boldsymbol{z}_n$, and denote the j-th element of $\boldsymbol{z}^n$ as $\boldsymbol{z}_j^n$. We have

$$
\begin{aligned}
\boldsymbol{\delta}_j^n &= \frac{\partial C}{\partial \boldsymbol{z}_j^n} \quad \text{(by definition of } \boldsymbol{\delta}^n\text{)} \\
&= \sum_k \frac{\partial C}{\partial \boldsymbol{z}_k^{n+1}} \frac{\partial \boldsymbol{z}_k^{n+1}}{\partial \boldsymbol{z}_j^n} \quad \text{(by chain rule)} \\
&= \sum_k \boldsymbol{\delta}_k^{n+1} \frac{\partial \boldsymbol{z}_k^{n+1}}{\partial \boldsymbol{z}_j^n} \quad \text{(by definition of } \boldsymbol{\delta}^{n+1}\text{)} \\
&= \sum_k \boldsymbol{\delta}_k^{n+1} \frac{\partial \left(\boldsymbol{z}_k^n + h f_k(\boldsymbol{z}^n, t_n, \boldsymbol{\theta}) \right)}{\partial \boldsymbol{z}_j^n} \quad \text{(by Equation (3.8))} \\
&= \sum_k \boldsymbol{\delta}_k^{n+1} \left(\boldsymbol{I}_{kj} + h \frac{\partial f_k(\boldsymbol{z}_n, t_n, \boldsymbol{\theta})}{\partial \boldsymbol{z}_j^n} \right) \\
&= \sum_k \left(\boldsymbol{I}_{jk} + h \frac{\partial f_k(\boldsymbol{z}_n, t_n, \boldsymbol{\theta})}{\partial \boldsymbol{z}_j^n} \right) \boldsymbol{\delta}_k^{n+1}.
\end{aligned}
\tag{A.1}
$$

A.1.2 A Proof of Equation (3.10)

For notational convenience, we write $\boldsymbol{z}^n \coloneqq \boldsymbol{z}_n$, and denote the j-th element of $\boldsymbol{z}^n$ as $\boldsymbol{z}_j^n$. By chain rule, we have

$$\frac{dC}{d\boldsymbol{\theta}_i} = \sum_k \frac{\partial C}{\partial \boldsymbol{z}_k^N} \frac{d\boldsymbol{z}_k^N}{d\boldsymbol{\theta}_i}.$$

Since $\boldsymbol{z}_k^n = \boldsymbol{z}_k^{n-1} + h f_k\left(\boldsymbol{z}^{n-1}, t_{n-1}, \boldsymbol{\theta}\right)$ (see Equation (3.8)), we note that

$$\frac{d\boldsymbol{z}_k^n}{d\boldsymbol{\theta}_i} = \frac{\partial \boldsymbol{z}_k^n}{\partial \boldsymbol{\theta}_i} + \sum_l \frac{\partial \boldsymbol{z}_k^n}{\partial \boldsymbol{z}_l^{n-1}} \frac{d\boldsymbol{z}_l^{n-1}}{d\boldsymbol{\theta}_i}.$$

It then follows that

$$\begin{aligned}
\frac{dC}{d\boldsymbol{\theta}_i} &= \sum_k \frac{\partial C}{\partial \boldsymbol{z}_k^N} \frac{\partial \boldsymbol{z}_k^N}{\partial \boldsymbol{\theta}_i} + \sum_{k,l} \frac{\partial C}{\partial \boldsymbol{z}_k^N} \frac{\partial \boldsymbol{z}_k^N}{\partial \boldsymbol{z}_l^{N-1}} \frac{d\boldsymbol{z}_l^{N-1}}{d\boldsymbol{\theta}_i} \quad \text{(by chain rule)} \\
&= \sum_k \boldsymbol{\delta}_k^N \frac{\partial \boldsymbol{z}_k^N}{\partial \boldsymbol{\theta}_i} + \sum_l \frac{\partial C}{\partial \boldsymbol{z}_l^{N-1}} \frac{d\boldsymbol{z}_l^{N-1}}{d\boldsymbol{\theta}_i} \quad \text{(by definition of } \boldsymbol{\delta}^N\text{)} \quad \text{(by chain rule)} \\
&= \sum_k \boldsymbol{\delta}_k^N \frac{\partial \boldsymbol{z}_k^N}{\partial \boldsymbol{\theta}_i} + \sum_l \frac{\partial C}{\partial \boldsymbol{z}_l^{N-1}} \frac{\partial \boldsymbol{z}_l^{N-1}}{\partial \boldsymbol{\theta}_i} + \sum_{l,m} \frac{\partial C}{\partial \boldsymbol{z}_l^{N-1}} \frac{\partial \boldsymbol{z}_l^{N-1}}{\partial \boldsymbol{z}_m^{N-2}} \frac{d\boldsymbol{z}_m^{N-2}}{d\boldsymbol{\theta}_i} \quad \text{(by chain rule)} \\
&= \sum_k \boldsymbol{\delta}_k^N \frac{\partial \boldsymbol{z}_k^N}{\partial \boldsymbol{\theta}_i} + \sum_l \boldsymbol{\delta}_l^{N-1} \frac{\partial \boldsymbol{z}_l^{N-1}}{\partial \boldsymbol{\theta}_i} + \sum_m \boldsymbol{\delta}_m^{N-2} \frac{d\boldsymbol{z}_m^{N-2}}{d\boldsymbol{\theta}_i} \quad \text{(by chain rule)} \\
&= \sum_{n=1}^{N} \left(\sum_k \boldsymbol{\delta}_k^n \frac{\partial \boldsymbol{z}_k^n}{\partial \boldsymbol{\theta}_i} \right) + \sum_k \boldsymbol{\delta}_k^0 \frac{d\boldsymbol{z}_k^0}{d\boldsymbol{\theta}_i} \quad \text{(by induction)} \\
&= \sum_{n=1}^{N} h \left(\sum_k \boldsymbol{\delta}_k^n \frac{\partial f_k\left(\boldsymbol{z}^{n-1}, t_{n-1}, \boldsymbol{\theta}\right)}{\partial \boldsymbol{\theta}_i} \right) \quad \text{(by Equation (3.8) and since } \boldsymbol{z}_0 \text{ is independent of } \boldsymbol{\theta}\text{)} \\
&= \sum_{n=0}^{N-1} h \left(\sum_k \boldsymbol{\delta}_k^{n+1} \frac{\partial f_k\left(\boldsymbol{z}^n, t_n, \boldsymbol{\theta}\right)}{\partial \boldsymbol{\theta}_i} \right)
\end{aligned}$$

For reading purposes, we also give a version of this proof in matrix notation:

$$
\begin{aligned}
\frac{dC}{d\boldsymbol{\theta}} &= \left(\frac{d\boldsymbol{z}_N}{d\boldsymbol{\theta}}\right)^T \frac{\partial C}{\partial \boldsymbol{z}_N} \quad \text{(by chain rule)} \\
&= \left(\frac{\partial \boldsymbol{z}_N}{\partial \boldsymbol{\theta}}\right)^T \frac{\partial C}{\partial \boldsymbol{z}_N} + \left(\frac{\partial \boldsymbol{z}_N}{\partial \boldsymbol{z}_{N-1}} \frac{d\boldsymbol{z}_{N-1}}{d\boldsymbol{\theta}}\right)^T \frac{\partial C}{\partial \boldsymbol{z}_N} \quad \text{(by chain rule)} \\
&= \left(\frac{\partial \boldsymbol{z}_N}{\partial \boldsymbol{\theta}}\right)^T \boldsymbol{\delta}^N + \left(\frac{d\boldsymbol{z}_{N-1}}{d\boldsymbol{\theta}}\right)^T \left(\frac{\partial \boldsymbol{z}_N}{\partial \boldsymbol{z}_{N-1}}\right)^T \frac{\partial C}{\partial \boldsymbol{z}_N} \quad \text{(by definition of } \boldsymbol{\delta}^N\text{)} \\
&= \left(\frac{\partial \boldsymbol{z}_N}{\partial \boldsymbol{\theta}}\right)^T \boldsymbol{\delta}^N + \left(\frac{\partial \boldsymbol{z}_{N-1}}{\partial \boldsymbol{\theta}}\right)^T \boldsymbol{\delta}^{N-1} + \left(\frac{d\boldsymbol{z}_{N-2}}{d\boldsymbol{\theta}}\right)^T \left(\frac{\partial \boldsymbol{z}_{N-1}}{\partial \boldsymbol{z}_{N-2}}\right)^T \boldsymbol{\delta}^{N-1} \quad \text{(by chain rule)} \\
&= \left(\frac{\partial \boldsymbol{z}_N}{\partial \boldsymbol{\theta}}\right)^T \boldsymbol{\delta}^N + \left(\frac{\partial \boldsymbol{z}_{N-1}}{\partial \boldsymbol{\theta}}\right)^T \boldsymbol{\delta}^{N-1} + \left(\frac{d\boldsymbol{z}_{N-2}}{d\boldsymbol{\theta}}\right)^T \boldsymbol{\delta}^{N-2} \quad \text{(by chain rule)} \\
&= \sum_{n=1}^{N} \left(\frac{\partial \boldsymbol{z}_n}{\partial \boldsymbol{\theta}}\right)^T \boldsymbol{\delta}^n + \left(\frac{d\boldsymbol{z}_0}{d\boldsymbol{\theta}}\right)^T \boldsymbol{\delta}^0 \quad \text{(by induction)} \\
&= \sum_{n=1}^{N} h \left(\frac{\partial f(\boldsymbol{z}_{n-1}, t_{n-1}, \boldsymbol{\theta})}{\partial \boldsymbol{\theta}}\right)^T \boldsymbol{\delta}^n \quad \text{(by Equation (3.8) and since } \boldsymbol{z}_0 \text{ is independent of } \boldsymbol{\theta}\text{)} \\
&= \sum_{n=0}^{N-1} h \left(\frac{\partial f(\boldsymbol{z}_n, t_n, \boldsymbol{\theta})}{\partial \boldsymbol{\theta}}\right)^T \boldsymbol{\delta}^{n+1}.
\end{aligned}
\tag{A.2}
$$

A.2 Midpoint Network

A.2.1 A Proof of Equation (3.14)

For notational convenience, we write $\boldsymbol{z}^n \coloneqq \boldsymbol{z}_n$, and denote the j-th element of $\boldsymbol{z}^n$ as $\boldsymbol{z}_j^n$. We have

$$
\begin{aligned}
\boldsymbol{\delta}_j^n &= \sum_k \boldsymbol{\delta}_k^{n+1} \frac{\partial \boldsymbol{z}_k^{n+1}}{\partial \boldsymbol{z}_j^n} \quad \text{(by the third line of Equation (A.1))} \\
&= \sum_k \boldsymbol{\delta}_k^{n+1} \frac{\partial \left(\boldsymbol{z}_k^n + h f_k(\tilde{\boldsymbol{z}}^n, t_{n^*}, \boldsymbol{\theta}) \right)}{\partial \boldsymbol{z}_j^n} \quad \text{(by Equation (3.12))} \\
&= \sum_k \boldsymbol{\delta}_k^{n+1} \left(\boldsymbol{I}_{jk} + h \frac{\partial f_k(\tilde{\boldsymbol{z}}^n, t_{n^*}, \boldsymbol{\theta})}{\partial \boldsymbol{z}_j^n} \right) \\
&= \sum_k \boldsymbol{\delta}_k^{n+1} \left(\boldsymbol{I}_{kj} + h \sum_l \frac{\partial f_k(\tilde{\boldsymbol{z}}^n, t_{n^*}, \boldsymbol{\theta})}{\partial \tilde{\boldsymbol{z}}_l^n} \frac{\partial \tilde{\boldsymbol{z}}_l^n}{\partial \boldsymbol{z}_j^n} \right) \quad \text{(by chain rule)} \\
&= \sum_k \boldsymbol{\delta}_k^{n+1} \left(\boldsymbol{I}_{kj} + h \sum_l \frac{\partial f_k(\tilde{\boldsymbol{z}}^n, t_{n^*}, \boldsymbol{\theta})}{\partial \tilde{\boldsymbol{z}}_l^n} \frac{\partial \left(\boldsymbol{z}_l^n + (h/2) f_l(\boldsymbol{z}^n, t_n, \boldsymbol{\theta}) \right)}{\partial \boldsymbol{z}_j^n} \right) \quad \text{(by Equation (3.13)} \\
&= \sum_k \boldsymbol{\delta}_k^{n+1} \left(\boldsymbol{I}_{kj} + h \sum_l \frac{\partial f_k(\tilde{\boldsymbol{z}}^n, t_{n^*}, \boldsymbol{\theta})}{\partial \tilde{\boldsymbol{z}}_l^n} \left(\boldsymbol{I}_{lj} + \frac{h}{2} \frac{\partial f_l(\boldsymbol{z}^n, t_n, \boldsymbol{\theta})}{\partial \boldsymbol{z}_j^n} \right) \right) \\
&= \sum_k \left(\boldsymbol{I}_{jk} + h \sum_l \left(\boldsymbol{I}_{lj} + \frac{h}{2} \frac{\partial f_l(\boldsymbol{z}^n, t_n, \boldsymbol{\theta})}{\partial \boldsymbol{z}_j^n} \right) \frac{\partial f_k(\tilde{\boldsymbol{z}}^n, t_{n^*}, \boldsymbol{\theta})}{\partial \tilde{\boldsymbol{z}}_l^n} \right) \boldsymbol{\delta}_k^{n+1}.
\end{aligned}
$$

A.2.2 A Proof of Equation (3.15)

We have

$$
\begin{aligned}
\frac{\partial C}{\partial \boldsymbol{\theta}} &= \sum_{n=0}^{N-1} \left(\frac{\partial \boldsymbol{z}_{n+1}}{\partial \boldsymbol{\theta}} \right)^T \boldsymbol{\delta}^{n+1} \quad \text{(by the sixth line of Equation (A.2) and since } \boldsymbol{z}_0 \text{ is independent of } \boldsymbol{\theta}) \\
&= \sum_{n=0}^{N-1} h \left(\frac{\partial f(\tilde{\boldsymbol{z}}_n, t_{n^*}, \boldsymbol{\theta})}{\partial \boldsymbol{\theta}} \right)^T \boldsymbol{\delta}^{n+1} \quad \text{(by Equation (3.12)).}
\end{aligned}
$$

Appendix B

Neural Ordinary Differential Equations

B.1 Linear Neural Ordinary Differential Equations

We give a proof of Theorem 3.3.1:

Proof: Let $\boldsymbol{z}_N$ be the Euler approximation to the IVP

$$\begin{aligned} \frac{d\boldsymbol{z}(t)}{dt} &= \boldsymbol{W}(t)\boldsymbol{z}(t) + \boldsymbol{b}(t), \quad t \in [0, T], \\ \boldsymbol{z}(0) &= \boldsymbol{z}_0, \end{aligned}$$

at time T. Then, $\boldsymbol{z}_N$ can be computed as

$$\begin{aligned} \boldsymbol{z}_N &= \boldsymbol{z}_{N-1} + h\boldsymbol{W}(t_{N-1})\boldsymbol{z}_{N-1} + h\boldsymbol{b}(t_{N-1}) \\ &= \boldsymbol{z}_{N-2} + h\boldsymbol{W}(t_{N-2})\boldsymbol{z}_{N-2} + h\boldsymbol{b}(t_{N-2}) + h\boldsymbol{W}(t_{N-1})\left(\boldsymbol{z}_{N-2} + h\boldsymbol{W}(t_{N-2})\boldsymbol{z}_{N-2} + h\boldsymbol{b}(t_{N-2})\right) + h\boldsymbol{b}(t_{N-1}) \\ &= \left(\boldsymbol{I} + h\boldsymbol{W}(t_{N-1})\right)\left(\boldsymbol{I} + h\boldsymbol{W}(t_{N-2})\right)\boldsymbol{z}_{N-2} + h\left(\boldsymbol{I} + h\boldsymbol{W}(t_{N-1})\right)\boldsymbol{b}(t_{N-2}) + h\boldsymbol{b}(t_{N-1}) \\ &= \left[\prod_{i=0}^{N-1}(\boldsymbol{I} + h\boldsymbol{W}(t_{N-1-i}))\right]\boldsymbol{z}_0 + h\sum_{j=0}^{N-1}\left[\prod_{i=j+1}^{N-1}(\boldsymbol{I} + h\boldsymbol{W}(t_i))\right]\boldsymbol{b}(t_j). \quad \text{(by induction)} \end{aligned}$$

Hence,

$$\boldsymbol{z}_n = \widetilde{\boldsymbol{W}}\boldsymbol{z}_0 + \widetilde{\boldsymbol{b}},$$

where

$$\widetilde{\boldsymbol{W}} = \left[\prod_{i=0}^{N-1} (\boldsymbol{I} + h\boldsymbol{W}(t_{N-1-i}))\right],$$

and

$$\widetilde{\boldsymbol{b}} = h \sum_{j=0}^{N-1} \left[\prod_{i=j+1}^{N-1} (\boldsymbol{I} + h\boldsymbol{W}(t_i))\right] \boldsymbol{b}(t_j).$$

□

B.2 Autonomous Neural Ordinary Differential Equations

We give a proof of Theorem 3.3.2:

Proof: Assume by contradiction that z_N converges to h pointwise, i.e.,

$$\lim_{N\to\infty} |z_N(x) - h(x)| = 0 \quad \forall x \in \mathbb{R}$$

This implies that for any $0 < \delta < 1$ there exits an $n^* \in \mathbb{N}$ such that

$$|z_n(x)| < \delta \quad \forall x \in [-\infty, -1] \cup [1, \infty], \tag{B.1}$$

for all $n > n^*$. Let us now fix an $\hat{x} \in [-\infty, -1] \cup [1, \infty]$ and denote $z_{n^*+1+i}(\hat{x}) = r_i$ for all $i \in \mathbb{N}$. By Equation (B.1), we have

$$|r_i| < \delta \quad \forall i \in \mathbb{N}. \tag{B.2}$$

Furthermore, note that by the autonomous IVP (3.18) we have

$$\begin{aligned} r_i &= z_{n^*+1+i}(\hat{x}) \\ &= z_{n^*+i}(\hat{x}) + hf(z_{n^*+i}(\hat{x}), \boldsymbol{\theta}) \\ &= r_{i-1} + hf(r_{i-1}, \boldsymbol{\theta}) \\ &= z_1(r_{i-1}) \quad \text{(by Equation (3.16))} \\ &= z_i(r_0) \quad \text{(by induction)}, \end{aligned}$$

for all $i > 1$ and trivially $r_0 = z_0(r_0)$. This implies, by Equation (B.2), that the absolute value of the Euler approximation at time step t_i with initial value r_0 is bounded by δ, i.e.,

$$|z_i(r_0)| = |r_i| < \delta \quad \forall i \in \mathbb{N}.$$

Let us now assume without loss of generality that $0 \leq r_0 < 1$[1], then $h(r_0) = 1 - r_0$. Hence, if we choose $\delta < 1 - r_0$, then

$$\lim_{N \to \infty} |z_N(r_0) - h(r_0)| \geq 1 - r_0 - \delta > 0,$$

which completes the proof. □

B.3 Universal Approximation with Hypernetworks

We give a proof of Theorem 3.3.3:

Proof: Lin and Jegelka (2018) showed that a residual network of the form

$$R(\boldsymbol{x}, \boldsymbol{\theta}) = (\boldsymbol{I} + \mathcal{T}_{N-1}) \circ \cdots \circ (\boldsymbol{I} + \mathcal{T}_0)(\boldsymbol{x}),$$

is a universal function approximator, i.e., for any Lebesgue-integrable function g and any $\epsilon > 0$ there is a finite N such that

$$\int_{\mathbb{R}^d} \|\mathcal{L} \circ R(\boldsymbol{x}) - g(\boldsymbol{x})\| \, d\boldsymbol{x} < \epsilon.$$

Here, $\mathcal{T}_i(\boldsymbol{x}) = \boldsymbol{V}_i \mathrm{ReLU}(\boldsymbol{U}_i \boldsymbol{x} + u_i)$ with $\boldsymbol{V}_i \in \mathbb{R}^{d \times 1}$, $\boldsymbol{U}_i \in \mathbb{R}^{1 \times d}$ and $u_i \in \mathbb{R}$ for all $i = 0, \ldots, N-1$. The Euler approximation at time $t = T$ to the IVP with dynamics $f_{\text{universal}}$, with $f_{\text{universal}}$ defined in Equation (3.20), and variable initial value $\boldsymbol{x}$ is given by

$$\begin{aligned} \boldsymbol{z}_N(\boldsymbol{x}) &= \boldsymbol{z}_{N-1}(\boldsymbol{x}) + h\boldsymbol{V}(t_{N-1})\mathrm{ReLU}(\boldsymbol{U}(t_{N-1})\boldsymbol{z}_{N-1}(\boldsymbol{x}) + u(t_{N-1})) \\ &= (\boldsymbol{I} + \mathcal{T}(t_{N-1}))\left(\boldsymbol{z}_{N-1}(\boldsymbol{x})\right), \end{aligned}$$

where we define

$$\mathcal{T}(t_i)(\boldsymbol{x}) = h\boldsymbol{V}(t_i)\mathrm{ReLU}(\boldsymbol{U}(t_i)\boldsymbol{x} + u(t_i)).$$

[1] We know that $|r_0| < 1$ by Equation (B.2)

By induction, it follows that

$$\boldsymbol{z}_N(\boldsymbol{x}) = (\boldsymbol{I} + \mathcal{T}(t_{N-1})) \circ \cdots \circ (\boldsymbol{I} + \mathcal{T}(t_0))(\boldsymbol{x}),$$

since $\boldsymbol{z}_0(\boldsymbol{x}) = \boldsymbol{x}$ by Equation (3.17). Comparing the definitions of $\mathcal{T}_i(\boldsymbol{x})$ and $\mathcal{T}(t_i)(\boldsymbol{x})$, we note that $R(\boldsymbol{x})$ and $\boldsymbol{z}_N(\boldsymbol{x})$ are equal if

$$\begin{aligned} \boldsymbol{V}(t_i) &= \boldsymbol{V}_i/h, \\ \boldsymbol{U}(t_i) &= \boldsymbol{U}_i, \\ u(t_i) &= u_i, \end{aligned}$$

for all $i = 0, \ldots, N-1$. The existence of a two-layer neural network with ReLU activation that can exactly map s input values to s output values was shown, for example, by Arora et al. (2018). Hence, there exists a two-layer hypernetwork of width $2dN + N$ that can realize the above conditions. By equality of $R(\boldsymbol{x})$ and $\boldsymbol{z}_N(\boldsymbol{x})$ we obtain

$$\int_{\mathbb{R}^d} \|\mathcal{L} \circ \boldsymbol{z}_N(\boldsymbol{x}) - g(\boldsymbol{x})\| \, d\boldsymbol{x} < \epsilon,$$

which completes the proof. □

Appendix C

Adjoint Sensitivity Method for Continuous Normalizing Flows

In this chapter, we give a derivation of Algorithm (5) for $d = 1$. Consider the constrained optimization problem

$$\min_{\boldsymbol{\theta}} C(\tilde{p}_T) = \min_{\boldsymbol{\theta}} C\left(\tilde{p}_0 + \int_0^T g(\mathrm{z}(t), t, \boldsymbol{\theta})\, dt\right),$$

subject to

$$\begin{aligned} h(\mathrm{z}(t), \dot{\mathrm{z}}(t), \boldsymbol{\theta}, t) &= \frac{d\mathrm{z}(t)}{dt} - f(\mathrm{z}(t), t, \boldsymbol{\theta}) = 0, \\ g(\mathrm{z}(T)) &= \mathrm{z}(T) - \mathrm{x} = 0, \\ l\left(\dot{\tilde{p}}_t, \mathrm{z}(t), \boldsymbol{\theta}, t\right) &= \frac{d\tilde{p}_t}{dt} - g(\mathrm{z}(t), t, \boldsymbol{\theta}) = 0, \\ m(\tilde{p}_0, \mathrm{z}(0)) &= \tilde{p}_0 - \log p_{\mathrm{z}_0}(\mathrm{z}(0)) = 0, \end{aligned}$$

where $\tilde{p}_t = \log p(\mathrm{z}(t), t)$ and $g(\mathrm{z}(t), t, \boldsymbol{\theta}) = -\operatorname{tr} \partial_{\mathrm{z}} f(\mathrm{z}(t), t, \boldsymbol{\theta})$. The Lagrangian is given as

$$\mathcal{L} = C(\tilde{p}_T) + \int_0^T \left[\lambda h + \eta l\right] dt + \mu g + \delta m,$$

where $\lambda = \lambda(t)$, $\eta = \eta(t)$, μ and δ are Lagrangian multipliers. Similar to the derivation of the adjoint sensitivity method for NODEs in Section 3.2.1, all constraints are satisfied

by construction and we choose the Lagrangian multipliers in order to find a tractable expression of $d_{\boldsymbol{\theta}} C(\tilde{p}_T)$. The derivatives of $\mathcal{L}$ with respect to $\boldsymbol{\theta}$ can be computed as

$$\begin{aligned}\frac{d}{d\boldsymbol{\theta}}\mathcal{L} =& \partial_{\tilde{p}_T} C \left(\frac{d\tilde{p}_0}{d\boldsymbol{\theta}} + \int_0^T \left[\partial_{\mathrm{z}} g d_{\boldsymbol{\theta}} \mathrm{z} + \partial_{\boldsymbol{\theta}} g \right] dt \right) + \int_0^T \lambda \left[\partial_{\mathrm{z}} h d_{\boldsymbol{\theta}} \mathrm{z} + \partial_{\dot{\mathrm{z}}} h d_{\boldsymbol{\theta}} \dot{\mathrm{z}} + \partial_{\boldsymbol{\theta}} h \right] dt + \\ & \int_0^T \eta \left[\partial_{\mathrm{z}} l d_{\boldsymbol{\theta}} \mathrm{z} + \partial_{\dot{\tilde{p}}} l d_{\boldsymbol{\theta}} \dot{\tilde{p}} + \partial_{\boldsymbol{\theta}} l \right] dt + \delta \left(\frac{d\tilde{p}_0}{d\boldsymbol{\theta}} + \frac{\partial \log p_{\mathrm{z}_0}(\mathrm{z}(0))}{\mathrm{z}(0)} \frac{d\mathrm{z}(0)}{d\boldsymbol{\theta}} \right).\end{aligned} \tag{A}$$

We eliminate $d_{\boldsymbol{\theta}} \dot{\mathrm{z}}$ and $d_{\boldsymbol{\theta}} \dot{\tilde{p}}$ using integration by parts:

$$\begin{aligned}\int_0^T \lambda \partial_{\dot{\mathrm{z}}} h d_{\boldsymbol{\theta}} \dot{\mathrm{z}} \, dt &= \lambda \partial_{\dot{\mathrm{z}}} h d_{\boldsymbol{\theta}} \mathrm{z} \big|_0^T - \int_0^T \left[\dot{\lambda} \partial_{\dot{\mathrm{z}}} h + \lambda d_t \partial_{\dot{\mathrm{z}}} h \right] d_{\boldsymbol{\theta}} \mathrm{z} \, dt \\ &= - \int_0^T \dot{\lambda} d_{\boldsymbol{\theta}} \mathrm{z} \, dt,\end{aligned} \tag{B}$$

where we used that $\partial_{\dot{\mathrm{z}}} h = 1$ and $d_{\boldsymbol{\theta}} \mathrm{z}(T) = d_{\boldsymbol{\theta}} \mathrm{x} = 0$; we further chose $\lambda(0) = 0$. Using that $\partial_{\dot{\tilde{p}}} l = 1$ and choosing $\eta(T) = 0$, we get

$$\begin{aligned}\int_0^T \eta \partial_{\dot{\tilde{p}}} l d_{\boldsymbol{\theta}} \dot{\tilde{p}} \, dt &= \eta \partial_{\dot{\tilde{p}}} l d_{\boldsymbol{\theta}} \tilde{p} \big|_0^T - \int_0^T \left[\dot{\eta} \partial_{\dot{\tilde{p}}} l + \eta d_t \partial_{\dot{\tilde{p}}} l \right] d_{\boldsymbol{\theta}} \tilde{p} \, dt \\ &= -\eta(0) d_{\boldsymbol{\theta}} \tilde{p}_0 - \int_0^T \dot{\eta} d_{\boldsymbol{\theta}} \tilde{p} \, dt.\end{aligned} \tag{C}$$

Combining (A), (B), and (C) we find

$$\begin{aligned}\frac{d}{d\boldsymbol{\theta}}\mathcal{L} = & \int_0^T \left[\partial_{\tilde{p}_T} C \partial_{\boldsymbol{\theta}} g + \lambda \partial_{\boldsymbol{\theta}} h + \eta \partial_{\boldsymbol{\theta}} l \right] dt - \int_0^T \dot{\eta} d_{\boldsymbol{\theta}} \tilde{p} \, dt + (\partial_{\tilde{p}_T} C + \delta - \eta(0)) \frac{d\tilde{p}_0}{d\boldsymbol{\theta}} \\ & + \int_0^T \left[\partial_{\tilde{p}_T} C \partial_{\mathrm{z}} g - \dot{\lambda} + \lambda \partial_{\mathrm{z}} h + \eta \partial_{\mathrm{z}} l \right] d_{\boldsymbol{\theta}} \mathrm{z} \, dt + \delta \frac{\partial \log p_{\mathrm{z}_0}(\mathrm{z}(0))}{\mathrm{z}(0)} \frac{d\mathrm{z}(0)}{d\boldsymbol{\theta}}.\end{aligned} \tag{C.1}$$

To avoid having to compute $d_{\boldsymbol{\theta}} \tilde{p}$, we choose η such that

$$\dot{\eta} = 0,$$

for all $0 < t < T$. Using the initial condition $\eta(T) = 0$ from above implies that $\eta(t) = 0$. Furthermore, to avoid having to compute $d_{\boldsymbol{\theta}} \tilde{p}_0$, we choose $\delta = \eta(0) - \partial_{\tilde{p}_T} C = -\partial_{\tilde{p}_T} C$. Equation (C.1) then simplifies to

$$\begin{aligned}\frac{d}{d\boldsymbol{\theta}}\mathcal{L} = & \int_0^T \left[\partial_{\tilde{p}_T} C \partial_{\boldsymbol{\theta}} g + \lambda \partial_{\boldsymbol{\theta}} h \right] dt + \int_0^T \left[\partial_{\tilde{p}_T} C \partial_{\mathrm{z}} g - \dot{\lambda} + \lambda \partial_{\mathrm{z}} h \right] d_{\boldsymbol{\theta}} \mathrm{z} \, dt \\ & - \partial_{\tilde{p}_T} C \frac{\partial \log p_{\mathrm{z}_0}(\mathrm{z}(0))}{\mathrm{z}(0)} \frac{d\mathrm{z}(0)}{d\boldsymbol{\theta}}.\end{aligned} \tag{C.2}$$

The term $d_{\boldsymbol{\theta}}\mathrm{z}(0)$ can be rewritten as

$$\begin{aligned}\frac{d}{d\boldsymbol{\theta}}\mathrm{z}(0) &= \frac{d}{d\boldsymbol{\theta}}\left(\mathrm{z}(T) + \int_T^0 f(\mathrm{z}(t), t, \boldsymbol{\theta})\, dt\right) \\ &= \int_T^0 \left[\partial_{\mathrm{z}} f d_{\boldsymbol{\theta}}\mathrm{z} + \partial_\theta f\right]\, dt.\end{aligned} \tag{C.3}$$

Using Equation (C.3), we can rewrite Equation (C.2) as

$$\begin{aligned}\frac{d}{d\boldsymbol{\theta}}\mathcal{L} = &\int_0^T \left[\partial_{\tilde{p}_T} C \partial_{\boldsymbol{\theta}} g + \lambda \partial_{\boldsymbol{\theta}} h - \partial_{\tilde{p}_T} C \frac{\partial \log p_{\mathrm{z}_0}(\mathrm{z}(0))}{\mathrm{z}(0)} \partial_{\boldsymbol{\theta}} f\right]\, dt + \\ &\int_0^T \left[\partial_{\tilde{p}_T} C \partial_{\mathrm{z}} g - \dot{\lambda} + \lambda \partial_{\mathrm{z}} h - \partial_{\tilde{p}_T} C \frac{\partial \log p_{\mathrm{z}_0}(\mathrm{z}(0))}{\mathrm{z}(0)} \partial_{\mathrm{z}} f\right] d_{\boldsymbol{\theta}}\mathrm{z}\, dt.\end{aligned}$$

To avoid having to compute $d_{\boldsymbol{\theta}}\mathrm{z}$, we choose λ such that

$$\partial_{\tilde{p}_T} C \partial_{\mathrm{z}} g - \dot{\lambda} - \lambda \partial_{\mathrm{z}} f - \partial_{\tilde{p}_T} C \frac{\partial \log p_{\mathrm{z}_0}(\mathrm{z}(0))}{\mathrm{z}(0)} \partial_{\mathrm{z}} f = 0, \tag{C.4}$$

for all $0 < t < T$, where we used that $\partial_{\mathrm{z}} h = -\partial_{\mathrm{z}} f$. Letting $\lambda^* = \lambda + \partial_{\tilde{p}_T} C \frac{\partial \log p(\mathrm{z}(0))}{\mathrm{z}(0)}$, we can rewrite Equation (C.4) as

$$\dot{\lambda}^* = \partial_{\tilde{p}_T} C \partial_{\mathrm{z}} g - \lambda^* \partial_{\mathrm{z}} f, \tag{C.5}$$

with initial condition $\lambda^*(0) = \lambda(0) + \partial_{\tilde{p}_T} C \frac{\partial \log p_{\mathrm{z}_0}(\mathrm{z}(0))}{\mathrm{z}(0)} = \partial_{\tilde{p}_T} C \frac{\partial \log p_{\mathrm{z}_0}(\mathrm{z}(0))}{\mathrm{z}(0)}$. Using Equation (C.5), the derivative of the Lagrangian with respect to $\boldsymbol{\theta}$ further simplifies to

$$\begin{aligned}\frac{d}{d\boldsymbol{\theta}}\mathcal{L} &= \int_0^T \left[\partial_{\tilde{p}_T} C \partial_{\boldsymbol{\theta}} g + \lambda \partial_{\boldsymbol{\theta}} h - \partial_{\tilde{p}_T} C \frac{\partial \log p_{\mathrm{z}_0}(\mathrm{z}(0))}{\mathrm{z}(0)} \partial_{\boldsymbol{\theta}} f\right]\, dt \\ &= -\int_0^T \lambda^* \partial_{\boldsymbol{\theta}} f\, dt + \int_0^T \partial_{\tilde{p}_T} C \partial_{\boldsymbol{\theta}} g\, dt \\ &= -\int_0^T \lambda^* \partial_{\boldsymbol{\theta}} f\, dt - \int_0^T \partial_{\tilde{p}_T} C \partial_{\boldsymbol{\theta}} \operatorname{tr} \partial_{\mathrm{z}} f\, dt.\end{aligned} \tag{C.6}$$

This derivation suggests the following algorithm to compute $\partial_{\boldsymbol{\theta}} C(\tilde{p}_t)$: Given $\mathrm{z}(T) = \mathrm{x}$, we first compute $\mathrm{z}(0) = \mathrm{z}(T) + \int_T^0 f\, dt$ and $\int_T^0 \operatorname{tr} \partial_{\mathrm{z}} f\, dt$. Next, we compute $\partial_{\mathrm{z}(0)} \log p_{\mathrm{z}_0}(\mathrm{z}(0))$ and

$$\partial_{\tilde{p}_T} C = \partial_{\tilde{p}_T} C \left(\log p_{\mathrm{z}_0}(\mathrm{z}(0)) + \int_T^0 \operatorname{tr} \partial_{\mathrm{z}} f\, dt\right).$$

Using these quantities, we can consider Equation (C.6), as the solution of an IVP at time T with dynamics

$$-\lambda^* \partial_{\boldsymbol{\theta}} f,$$

and initial condition

$$-\int_0^T \partial_{\tilde{p}_T} C \partial_{\boldsymbol{\theta}} \operatorname{tr} \partial_{\mathrm{z}} f \, dt.$$

By Equation (C.6), Equation (C.5), and initial condition $\lambda^*(0) = \partial_{\tilde{p}_T} C \frac{\partial \log p_{\mathrm{z}_0}(\mathrm{z}(0))}{\mathrm{z}(0)}$, we can then obtain $d_{\boldsymbol{\theta}} C(\tilde{p}_T) = d_{\boldsymbol{\theta}} \mathcal{L}$ by computing

$$\begin{bmatrix} \mathrm{z}(T) \\ \lambda^*(T) \\ d_{\boldsymbol{\theta}} C(\tilde{p}_T) \end{bmatrix} = \texttt{odesolve} \left(\begin{bmatrix} f \\ -\partial_{\tilde{p}_T} C \partial_{\mathrm{z}} \operatorname{tr} \partial_{\mathrm{z}} f - \lambda^* \partial_{\mathrm{z}} f \\ -\lambda^* \partial_{\boldsymbol{\theta}} f \end{bmatrix}, \begin{bmatrix} \mathrm{z}(0) \\ \partial_{\tilde{p}_T} C \frac{\partial \log p_{\mathrm{z}_0}(\mathrm{z}(0))}{\mathrm{z}(0)} \\ -\partial_{\tilde{p}_T} C \int_T^0 \operatorname{tr} \partial_{\mathrm{z}} f \, dt \end{bmatrix}, [0, T] \right).$$

Appendix D

Experiments

D.1 A Description of `8gaussians`

In this section, we explain how data points of the `8gaussians` dataset (Grathwohl et al., 2018) are generated. First, we define an array of eight circle centers

$$C_{\texttt{8gaussians}} = \Bigg[(4,0), (-4,0), (0,4) (0,-4), \left(4/\sqrt{2}, 4/\sqrt{4}\right), \\ \left(4/\sqrt{2}, -4/\sqrt{2}\right), \left(-4/\sqrt{2}, 4/\sqrt{2}\right), \left(-4/\sqrt{2}, -4/\sqrt{2}\right) \Bigg].$$

Given a desired dataset size m, we first draw m random integers $\{\mathcal{I}_i\}_{i=1}^m$ uniformly from the index set $\{1, \ldots, 8\}$. The i-th point $\mathbf{x}^{(i)}$ of the `8gaussians` dataset is then computed as

$$\mathbf{x}^{(i)} = C_{\texttt{8gaussians}}[i] + \frac{1}{2}\boldsymbol{\epsilon},$$

where $\boldsymbol{\epsilon} \sim \mathcal{N}(\mathbf{0}, \boldsymbol{I})$. The dataset can be kept deterministic by manually setting the random seed of the random number generator.