Spline Parameterization for Continuous Normalizing Flows

by

Shuhui Zhu

A research paper
presented to the University of Waterloo
in fulfillment of the
research paper requirement for the degree of
Master of Mathematics
in
Computational Mathematics

Waterloo, Ontario, Canada, 2021

Author's Declaration

Abstract

Neural Ordinary Differential Equations (Neural ODEs) are deep learning neural networks with constraints specified by ordinary differential equations and initial values. Training Neural ODEs can be viewed as an optimal control problem. The weights and hidden states of Neural ODEs are equivalent to controls and states in an optimal control problem that aims to find an optimal set of controls for a dynamical system over a period of time. Continuous Normalizing Flows (CNFs), a family of probabilistic generative models, constructed by Neural ODEs. Based on the model description of continuous normalizing flows as optimal control problem, we introduce B-Spline basis functions to parameterize the Continuous Normalizing Flows controls across layers. This method produces time-dependent weights by combining a fixed number of trainable parameters with B-Spline basis functions, which makes it possible to learn complex time-dependent patterns by design. We execute experiments for various toy datasets and solve the neural network by the Discretize-Then-Optimize (DTO) approach and the Optimize-Then-Discrete (OTD) approach. Numerical results show that the Spline-based CNFs have good performance for density estimation and sampling. In this research paper, we also compare different right-hand side functions for of the ODE network including concatsquash layers and linear-spline layers, under DTO and OTD methods. Even though the OTD can guarantee the invertibility of the ODE network, the DTO is more efficient for training. Further, the Spline-based CNFs can achieve a dramatic reduction of parameters and number of evaluations while maintaining accuracy, thus saving computational cost.

Acknowledgements

I would like to express my deepest appreciation to my supervisors Hans De Sterck and Jun Liu for their supports and guidance. Thanks for helping me find my research direction and make me learn a lot during this journey.

I want to extend my sincere thanks to Giang Tran for reviewing my research paper.

Thanks to Derek Onken for providing useful suggestions.

Thanks to Erin Kelly for helping me access the laboratory.

Thanks to Pascal Poupart for helpful discussion on my research topic.

Thanks to my parents for supporting me with encouragement throughout the duration of my master's degree.

Minghao, thank you for always inspiring me and caring about me.

Dedication

To my love, Chengrong Wu, Xuecong Zhu, Yu Zhu and Minghao Ning

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 State of the Art

A crucial application of machine learning is to model an unknown posterior distribution given the data sampled from it. This is also called variational inference [Jordan et al., 1999] [Wainwright and Jordan, 2008a]. Neural networks have the ability to learn and build complex nonlinear models and enable the approximation of complex unknown distributions. This makes neural networks a popular choice in variational inference modeling. Many machine learning methods for density estimation are based on neural networks, including generative adversarial networks [Goodfellow et al., 2014], variational auto-encoders [Kingma and Welling, 2013] and normalizing flows [Rezende and Mohamed, 2015].

In this research paper, we focus on continuous normalizing flows, a continuous version of normalizing flows modeled by neural ordinary differential equations [Chen et al., 2018]. The goal of continuous normalizing flows has been to learn the approximate posterior probability to estimate the unknown target density, by training with sample data from the target density. The idea of this model is to assume that we can build invertible bijections to map the sample data from the target distribution to the standard normal distribution.

Suppose we have the sample data that generated from the unknown target distribution and we know the coordinates of the sample data denoted as $\vec{\mathbf{x}} \in \mathbb{R}^{d_0}$. By assumption that we can map the sample data from the target distribution to the standard normal distribution by a bijective function f, we can obtain the coordinates of the after-mapping sample data, denoted as $\vec{\mathbf{z}} \in \mathbb{R}^{d_0}$, by equation (1.1). Since $\vec{\mathbf{z}}$ is assumed to follow standard normal distribution, it is easy to compute the probability of $\vec{\mathbf{z}}$ by plugging it into the standard normal density function (1.3).

$$\vec{\mathbf{z}} = f(\vec{\mathbf{x}}) \tag{1.1}$$

$$\vec{\mathbf{x}} = f^{-1}(\vec{\mathbf{z}}) \tag{1.2}$$

$$p(\vec{\mathbf{z}}) = (2\pi)^{-\frac{d_0}{2}} \exp(-\frac{\vec{\mathbf{z}}^T \vec{\mathbf{z}}}{2}) \tag{1.3}$$

$$p(\vec{\mathbf{x}}) = p(\vec{\mathbf{z}}) - \Delta p \tag{1.4}$$

Afterwards, we can calculate the difference between the original sample data probability for $\vec{\mathbf{x}}$ and after-mapping data probability for $\vec{\mathbf{z}}$ by the instantaneous change of variables theorem [Chen et al., 2018]. Denoting the density difference as Δp, we can easily get the approximate posterior density of the original sample data $p(\vec{\mathbf{x}})$ by equation (1.4).

The intuitive idea of continuous normalizing flows is straightforward but how to construct proper mapping flows is a complicated problem. There are two points we need to think about when designing the model structure. The first point is to guarantee the invertibility. This is required when doing the sampling process, that is, to get samples from the approximate target density by sampling randomly from the normal distribution and then apply the inverse mapping to get the datapoints we want. The second point we need to pay attention to is the computational cost of the flow model. On the one hand, large models are time-consuming for training; on the other hand, small models may not provide accurate mappings. For example, we can never convert data from the standard normal distribution to a distribution of MNIST images by simple linear bijective mappings.

In this work, we introduce B-spline basis functions [Günther et al., 2021] to parameterize the neural network constructed in continuous normalizing flows. Instead of using the same parameters when discretizing the neural ordinary differential equations across multiple layers for the numerical solution, we choose time-dependent B-spline functions with a fixed number of trainable parameters to make up the neural network.

The spline-based networks are continuous in time by design, and we can establish linear or nonlinear mapping flows by adjusting the degree of B-spline basis functions and the number of spline knots or intervals. In particular, if we set the degree equal to one and the number of spline knots equal to number of discretization steps, the network would be equivalent to residual networks. B-spline basis functions have also been proved to be advantageous to handle numerical partial differential equations in isogemetric analysis

[Cottrell et al., 2009]. In this research paper, we want to investigate the capability of spline-based continuous normalizing flows for dealing with generative probability tasks. In experiments, we discover the numerical performance of the neural differential equation model.

Furthermore, the neural ordinary differential equations in continuous normalizing flows are solved by numerical methods for training and evaluation, that is, we make use of ODE solvers to discretize the ODEs from continuous states to multiple layers, and then solve the ODE forward and backward in time. All the time-dependent functions are evaluated at the discrete time point of each layer. In this research paper, we compare two different schemes for numerical solutions, which are named Discretize-Then-Optimize (DTO) and Optimize-Then-Discretize (OTD). Discretize-Then-Optimize (DTO) first discretizes the Neural ODE network, and then executes the backpropagation throughout all discrete layers. For each iteration, the hidden states are memorized in forward propagation, and used to calculate the gradients of the loss function with respect to the weights backward through the whole network.

The other approach, the Optimize-Then-Discretize (OTD) method, also called the adjoint-based method, first introduced for neural networks by [Chen et al., 2018], where the objective is to optimize the continuous ordinary differential equations. Instead of applying traditional backpropagation for training, the OTD method optimizes Neural ODEs by solving for the gradient by continuous formulas, making use of adjoint states. It then discretizes the ODE network and applies ODE solvers such as forward Euler to numerically solve the ODEs of the continuous formula during training and evaluation. Our goal of this research paper is to investigate the performance of spline-based Neural ODEs when applying to the CNF model, and to execute numerical experiments under DTO and OTD schemes, comparing the number of forward evaluations, testing loss and inverse error to check which is better regarding computational cost, accuracy and invertibility of each model. Furthermore, we use a multi-spline-layer neural network while training and modify the dimensions and number of intermediate layers to investigate the relationship between model complexity, performance, and computational cost.

1.2 Outline

In this paper, we first introduce related background of this research in Chapter 2, including details of Neural ODEs, CNFs and more basic materials like neural networks, B-spline functions and etc. In Chapter 3, we introduce our spline-based methodology to specify the right-hand side function of the Neural ODEs in CNFs model, the ODE solvers we pick, and the training schemes we use. In Chapter 4, we describe our numerical results with respect to different experiments. In this chapter, we display the density estimation results by figures and tables, so we can compare the models straightforwardly and quantifiably. We also discuss the observations from the numerical experiments in chapter 4. Finally, in Chapter 5, we draw conclusions.

Chapter 2

Background

This chapter reviews background knowledge related to our methodology and algorithm. Firstly, we introduce the basic materials about neural networks and functionals we use in the following algorithms. Then, we investigate the neural ordinary differential equations and numerical methods to solve ODEs with initial value conditions. We also introduce several ODE solvers including fixed-step ODE solvers and adaptive ODE solvers. Furthermore, we discuss the generative probability models for variational inference, especially focusing on normalizing flows that we apply in our experiments. We address the differences and relationship between traditional normalizing flows and continuous normalizing flows which are firstly introduced by [Chen et al., 2018]. Finally, we discuss the definition and intuition for spline functions used in neural networks. In this chapter, we give detailed intuition for continuous normalizing flows and B-spline functions, which contribute to understanding our methodology and further experiments analysis.

2.1 Neural Networks

Neural networks are computing systems based on interconnected artificial neurons. Each neuron unit can transform information via connection between neuron nodes. After processing the information transmitted from the former neuron, it can pass the information to neurons connected to it in the next layer. For multiple layers neural networks, which are generally used in deep learning, there can be different transformations on the input of in different layers. These transformations are typically specified by non-linear functions called activation functions. Therefore, neural networks have the ability of fitting complex models via multiple layers with various activation functions. This is one of the most important reasons that it is a popular choice to use neural networks in deep learning, especially when we have no idea what the real model is. In the next two subsections, we give a mathematical definition of multi-layer artificial neural network, and discuss the activation functions we use in our experiments.

2.1.1 Multi-layer Artificial Neural Networks

A fully connected multi-layer neural network is called a Multilayer Perceptron (MLP), which is generally used in deep learning problems. The MLP is composed of an input layer, an output layer, and several hidden layers with mappings between each layer. Next, we give a mathematical definition of MLP and display a simple example of a 4-layer MLP.

Multilayer Perceptron (MLP): Suppose the input data is a d-dimensional vector $\vec{x} \in \mathbb{R}^d$, the intermediate state $\vec{x_k}$ is d_k-dimensional vector, $\vec{x_k} \in \mathbb{R}^{d_k}$, f_k is the activation function for kth layer (start from 0, that is input layer), W_k is the weight matrix and $\vec{b_k}$ is the bias vector at kth layer. Then the L-layer feed-forward MLP can be denoted as:

$$\begin{aligned}
\vec{x}_1 =& f_1(W_1\vec{x} + \vec{b}_1), \\
\vec{x}_k =& f_k(W_k\vec{x}_{k-1} + \vec{b}_k), \quad k = 2, 3, 4, ..., L-1, \\
\vec{x}_L =& f_L(W_L\vec{x}_{L-1} + \vec{b}_L),
\end{aligned} \tag{2.1}$$

where $W_1 \in \mathbb{R}^{d_1 \times d}$, $b_1 \in \mathbb{R}^{d_1}$; $W_k \in \mathbb{R}^{d_k \times d_{k-1}}$, $b_k \in \mathbb{R}^{d_k}$ for $k = 2, 3, 4, ..., L$.

In Figure 2.1, we display a simple structure of a fully connected four-layer artificial neural network, where the circles represent the artificial neuron units and interconnected edges represent the mappings between layers. These mappings are done by a bunch of activation functions. Once we have the input data at the input layer, we can process it through forward propagation as displayed in equation (2.1) to get all hidden states values as well as the final state output. Each interconnected line in Figure 2.1 represents a mapping from the a previous layer to the next one in forward propagation process.

The training of the whole network in Neural ODEs is done by selected optimization algorithm via backpropagation or reverse-mode automatic differentiation, it depends on the discretization scheme we choose. We give a more detailed explanation for discretization schemes in Section 2.5. For each training iteration, we update the parameters to reduce the loss function value until we have the satisfying accuracy according to tolerance we set, or the number of total training iterations reaches the maximum. In our experiments, we choose the latter strategy, which allows 10,000 training iterations for each model.

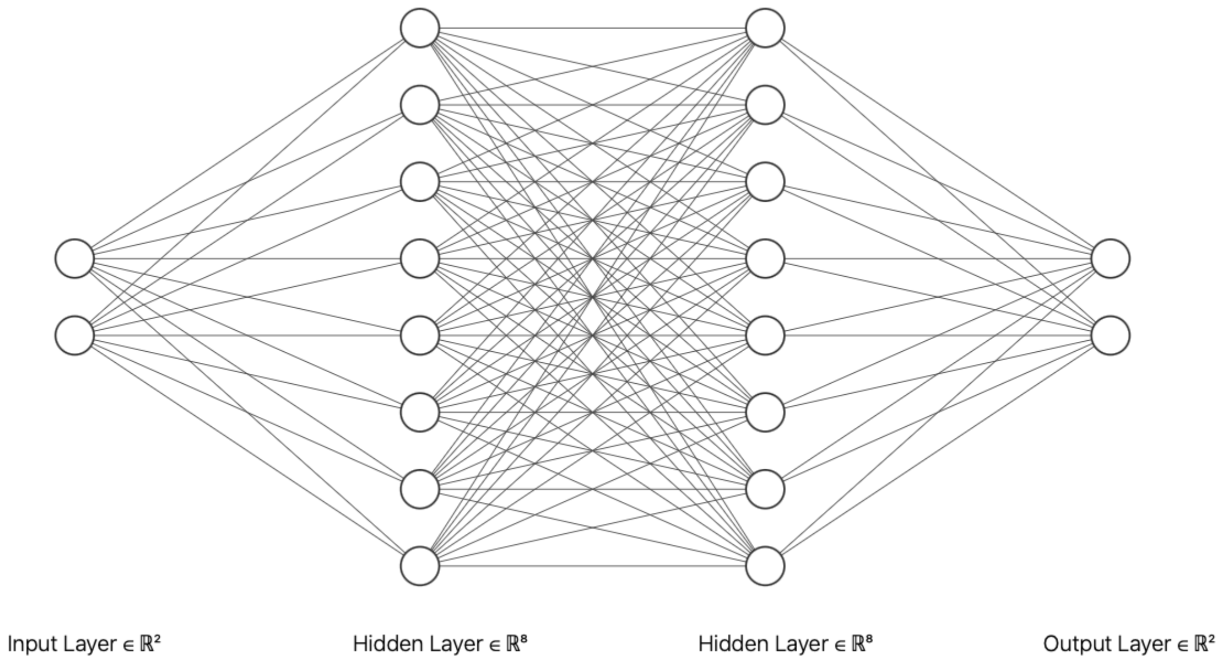


Figure 2.1: A 4-layer MLP: circles are artificial neurons, interconnected lines represent the mappings between neurons in forward propagation.

2.1.2 Activation Functions

In this section, we display several activation functions that will be applied in numerical experiments of continuous normalizing flows. Suppose the input of each function is x. Then the output of each formula can be given as follows. Plus, we display some figures via MATLAB to show the transformation and range of output of each function.

(1) RELU Activation:

$$f(x) = \max\{0, x\} \tag{2.2}$$

(2) Hyperbolic Tangent (Tanh) Activation:

$$f(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{2.3}$$

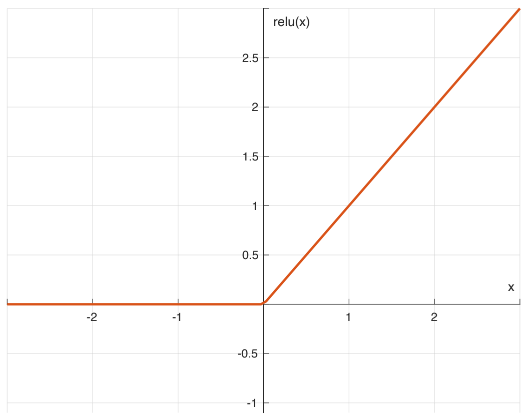


Figure 2.2: RELU

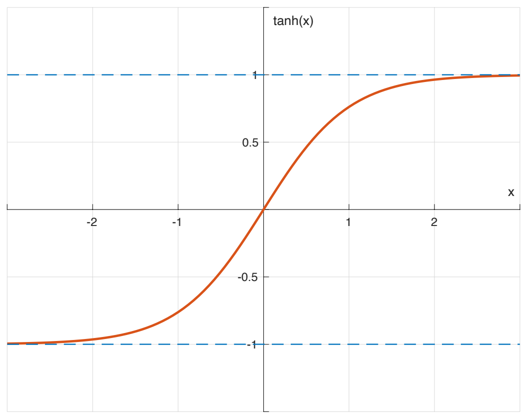


Figure 2.3: Tanh

(3) Logistic sigmoid Activation:

$$f(x) = \frac{1}{1 + e^{-x}} \tag{2.4}$$

(4) Softplus Activation:

$$f(x) = \ln(1 + e^{x}) \tag{2.5}$$

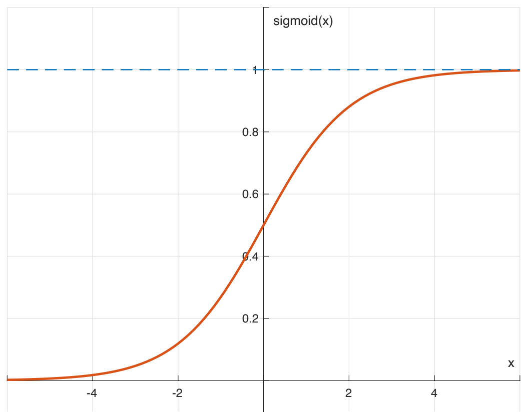


Figure 2.4: Logistic sigmoid

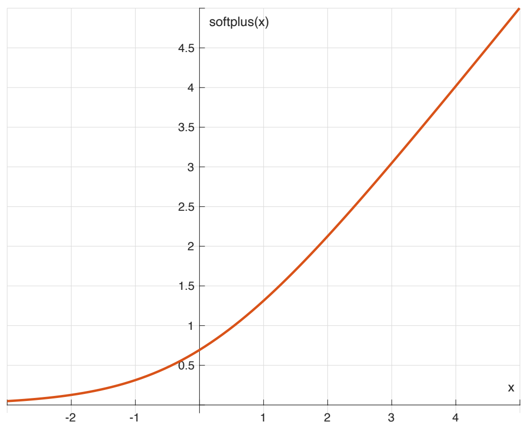


Figure 2.5: Softplus

2.2 Neural Ordinary Differential Equations

2.2.1 Model Description

Neural ordinary differential equations parameterize the continuous dynamics of hidden states using ordinary differential equations (ODEs) [Chen et al., 2018]. Instead of constructing the transformations between each layer, Neural ODEs directly model the gradient of dynamics of hidden units with respect to time.

Denote the d_0-dimensional initial value vector $\vec{x_0}$ as $\vec{x_0} \in \mathbb{R}^{d_0}$, the Neural ODEs can be expressed as follows:

$$\begin{aligned} \partial_t \vec{\mathbf{x}}(t) &= \boldsymbol{\ell}(\vec{\boldsymbol{\theta}}(t), \vec{\mathbf{x}}(t), t), \quad \text{for } t \in (0, T], \\ \vec{\mathbf{x}}(0) &= \vec{x}_0, \end{aligned} \tag{2.6}$$

where $\mathbf{x} : [0, T] \to \mathbb{R}^{d_0}$ is the continuous dynamic of hidden units with respect to time, $\boldsymbol{\theta} : [0, T] \to \mathbb{R}^p$ is the trainable parameter in this Neural ODE, and $\boldsymbol{\ell} : \mathbb{R}^p \times \mathbb{R}^{d_0} \times [0, T]$ is the model we build to approximate the derivative of the continuous dynamic hidden units with respect to time. We usually specify the function $\boldsymbol{\ell}$ by a multi-layer neural network (MLP) that we discuss in the previous section.

2.2.2 ODE Solvers

In Neural ODEs settings, the problem is restricted by an ODE equation and the initial value of the continuous dynamic. We refer to this kind of problem that finding a solution that satisfies the ODE conditions as initial value problem (IVP). The numerical techniques we apply to solve the the IVP built in Neural ODEs are called ODE solvers. Next, we introduce several ODE solvers that related to our methodology as well as numerical experiments.

Forward Euler's method: Given the IVP shown in equation (2.6), forward Euler method discretize the ODE to an N-layer network as follows:

$$\vec{\mathbf{x}}_n = \vec{\mathbf{x}}_{n-1} + h\boldsymbol{\ell}(\vec{\boldsymbol{\theta}}_{n-1}, \vec{\mathbf{x}}_{n-1}, t_{n-1}), \quad \text{for } n = 1, 2, 3, ..., N, \tag{2.7}$$

where $h = T/N$ is the step size, $t_0 = 0$, $t_N = T$, $t_{n+1} = t_n + h$ for $n = 0, 1, 2, ..., N-1$. The local truncation error of the forward Euler's method is $O(h^2)$.

Backward Euler's method: The backward Euler's discretization is defined as:

$$\vec{\mathbf{x}}_n = \vec{\mathbf{x}}_{n+1} + h\boldsymbol{\ell}(\vec{\boldsymbol{\theta}}_{n+1}, \vec{\mathbf{x}}_{n+1}, t_{n+1}), \quad \text{for } n = 0, 1, 2, ..., N-1, \tag{2.8}$$

where $h = T/N$ is the step size, $t_0 = 0$, $t_N = T$, $t_{n+1} = t_n + h$ for $n = 0, 1, 2, ..., N-1$. The local truncation error of backward Euler's method is $O(h^2)$.

For different discretization scheme of Neural ODEs, we discuss the forward Euler's method and backward Euler's method to show the difference between the schemes and training process with respect to them.

Runge-Kutta method: Runge-Kutta methods are a family of ODE solvers that can achieve higher accuracy than Euler's method. In this research paper, we use the most widely applied member in Runge-Kutta family, which we call RK4 method. The form of RK4 is given as below:

$$\begin{aligned}
&\vec{\mathbf{x}}_n = \vec{\mathbf{x}}_{n-1} + \frac{1}{6}h(k_1 + 2k_2 + 2k_3 + k_4), \quad \text{for } n = 1, 2, 3, .., N, \\
&k_1 = \boldsymbol{\ell}(\vec{\boldsymbol{\theta}}_{n-1}, \vec{\mathbf{x}}_{n-1}, t_{n-1}), \\
&k_2 = \boldsymbol{\ell}(\vec{\boldsymbol{\theta}}_{n-1}, \vec{\mathbf{x}}_{n-1} + h\frac{k_1}{2}, t_{n-1} + \frac{h}{2}), \\
&k_3 = \boldsymbol{\ell}(\vec{\boldsymbol{\theta}}_{n-1}, \vec{\mathbf{x}}_{n-1} + h\frac{k_2}{2}, t_{n-1} + \frac{h}{2}), \\
&k_4 = \boldsymbol{\ell}(\vec{\boldsymbol{\theta}}_{n-1}, \vec{\mathbf{x}}_{n-1} + hk_3, t_{n-1} + h),
\end{aligned} \tag{2.9}$$

where $h = T/N$ is the step size, $t_0 = 0$, $t_N = T$, $t_{n+1} = t_n + h$ for $n = 0, 1, 2, ..., N-1$. The local truncation error of the RK4 method is $O(h^5)$ which is higher than Euler's method. We apply the RK4 ODE solver in our numerical experiments using DTO approach for the training, which we discuss more in Chapter 4.

Dopri5: Dopri5 method is an adaptive step size ODE solver [Shampine, 1986]. This method is also a member of the Runge–Kutta family. In this method, we need to set the tolerance values for absolute error and relative error. Only if both absolute error and relative error are smaller than our tolerance values, can we accept the current step size and go to next step/layer. Otherwise, we need to adjust to set smaller step size and re-compute and re-evaluate the errors again until they are both accepted.

$$e_{absolute} = \left\| \vec{\mathbf{x}_n^{p}} - \vec{\mathbf{x}_n^{p-1}} \right\|, \tag{2.10}$$

$$e_{relative} = \frac{\left\| \vec{\mathbf{x}_n^{p}} - \vec{\mathbf{x}_n^{p-1}} \right\|}{max\left\{ \vec{\mathbf{x}_n^{p}}, \vec{\mathbf{x}_n^{p-1}} \right\}}, \tag{2.11}$$

where $e_{absolute}$ is the absolute error and $e_{relative}$ is referred to relative error. $\vec{\mathbf{x}}_n^p$ is the approximate hidden unit value for nth step which use pth-order Runge-Kutta method, $\vec{\mathbf{x}}_n^{p-1}$ is the approximate hidden unit value for nth step which use $(p-1)$th-order Runge-Kutta method.

In our numerical experiments, we apply the dopri5 in OTD scheme for training. And the combination of absolute error tolerance and relative error tolerance are shown as below:

$$\frac{e_{absolute}}{tol_{absolute} + tol_{relative} * e_{relative}} \leq 1, \tag{2.12}$$

where $tol_{absolute}$ and $tol_{relative}$ are absolute error tolerance and relative error tolerance respectively, aerr and rerr are absolute error and relative error given in equation (2.10) and (2.11). Once inequality (2.12) hold for the current step size we use, we can move to from n-step to the next and repeat the whole process until finishing all iterations for training.

2.3 Continuous Normalizing Flows

Continuous normalizing flows are generative probabilistic models specified by Neural ODEs that we discuss in last section. The goal of generative probabilistic modeling is to estimate a posterior probability distribution given samples drawn from the target distribution that we want to approximate. Before introducing the continuous normalizing flows, we firstly analyze the normalizing flows [Rezende and Mohamed, 2015] generally used for variational inference which is also called generative modeling.

2.3.1 Normalizing Flows

Normalizing flows [Rezende and Mohamed, 2015] are a family of generative models that are able to do density estimation and sampling. Density estimation is to approximate probability of an unknown target distribution, while sampling is to draw samples from the unknown target distribution.

A normalizing flow is modeled a bijection mapping between an unknown target distribution and a known simple base probability distribution that usually presented by a standard normal distribution. This kind of models make use of the discrete version of change of variables theorem to calculate the changes in probability after mapping through the bijective function. We denote the mapping function from target density to base density as ℓ, then the transformation of the data and the change of probability are given as below:

$$\begin{aligned} \vec{\mathbf{z}} &= \ell(\vec{\mathbf{x}}), \\ \log p(\vec{\mathbf{z}}) &= \log p(\vec{\mathbf{x}}) - \log \left| det \frac{\partial \ell}{\partial \vec{\mathbf{x}}} \right|, \end{aligned} \tag{2.13}$$

Through the change of variable theorem shown in equation (2.13), we can compute the change of probability forward and backward through the bijective function ℓ. Figure 2.6. gives an example to illustrate a normalizing flow model structure.

For density estimation, given the structure of mapping function ℓ and sample data $\vec{\mathbf{x}}$ from the unknown target distribution, we can compute the converted sample values $\vec{\mathbf{z}}$ after mapping via ℓ. Then we can easily get $p(\vec{\mathbf{z}})$ by plugging $\vec{\mathbf{z}}$ into a standard normal distribution. Next, through the change of variable theorem, we can get the $\log p(\vec{\mathbf{x}})$ by equation (2.13). Finally, we take an exponential with respect to $\log p(\vec{\mathbf{x}})$ to get the $p(\vec{\mathbf{x}})$. This is the approximate posterior probability derived by normalizing flows.

For sampling, we need to know the inverse function of ℓ, denoted as ℓ^{-1}. To do the sampling from the target density, we firstly sample some data points $\vec{\mathbf{z}}$ from standard normal distribution, subsequently we can compute the coordinates of samples $\vec{\mathbf{x}}$ in target distribution by mapping from $\vec{\mathbf{z}}$ via ℓ^{-1}, i.e. $\vec{\mathbf{x}} = \ell^{-1}(\vec{\mathbf{z}})$.

2.3.2 Continuous Normalizing Flows

Continuous normalizing flows are ODE-based normalizing flows, instead of using multiple layers of flow-based mappings, continuous normalizing flows directly model the derivative of hidden states with respect to time. The modeling for continuous dynamics is represented by a Neural ODE, which is continuous in time. Next, given the gradient function of hidden states with respect to time, we can compute the gradient of log likelihood of hidden units with respect to time by instantaneous change of variables theorem that is proved by [Chen et al., 2018]. The model structure of continuous normalizing flows is given below:

$$\begin{aligned} \partial_t \vec{\mathbf{x}}(t) &= \boldsymbol{\ell}(\vec{\boldsymbol{\theta}}(t), \vec{\mathbf{x}}(t), t), \quad for\ t \in (0, T], \\ \frac{\partial \log p(\vec{\mathbf{x}}(t))}{\partial t} &= -tr(\frac{\partial \boldsymbol{\ell}(\vec{\boldsymbol{\theta}}(t), \vec{\mathbf{x}}(t), t)}{\partial \vec{\mathbf{x}}(t)}), \quad for\ t \in (0, T], \\ \vec{\mathbf{x}}(0) &= \vec{\mathbf{x}}_0, \end{aligned} \tag{2.14}$$

where $\vec{\mathbf{x}} : [0, T] \to \mathbb{R}^{d_0}$ are the continuous dynamics of hidden units with respect to time, $\boldsymbol{\theta} : [0, T] \to \mathbb{R}^p$ are the trainable parameters in this Neural ODE, $\boldsymbol{\ell} : \mathbb{R}^p \times \mathbb{R}^{d_0} \times [0, T]$ is

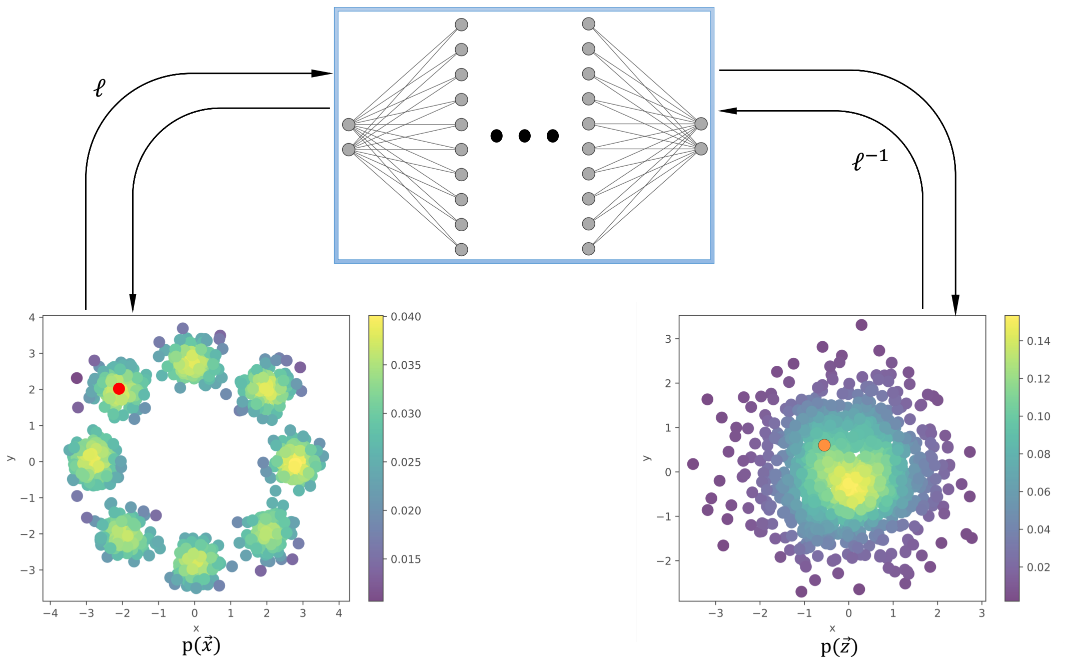


Figure 2.6: Normalizing Flows: We use a multi-layer neural network to represent the mapping function ℓ, the red point is $\vec{\mathbf{x}}$ from two dimensional eight-Gaussian distribution and the orange point is $\vec{\mathbf{z}}$ from the two dimensional standard normal distribution. These two points can transfer to each other by function ℓ or ℓ^{-1}.

the model we build to approximate the derivative of the continuous dynamic hidden units with respect to time, tr is the trace operation.

The invertibility of continuous normalzing flows is guaranteed by the existence and uniqueness of the solution of the ODE. Under assumptions that $\boldsymbol{\ell}$ is Lipschitz continuous w.r.t. $\vec{\mathbf{x}}(t)$ and $\vec{\boldsymbol{\theta}}(t)$ are continuous in t, the solution exists and is unique given the initial condition $\vec{\mathbf{x}}(0) = \vec{\mathbf{x}}_0$ [Arnol'd, 2013]. However, for discrete normalizing flows, we can not naturally get invertible bijections by design. This provides a motivation of using continuous normalizing flows rather than traditional multi-layer normalizing flows especially for sampling that requires to know ℓ^{-1}, the inverse function of forward mapping.

The training for continuous normalizing flows is also different from training a finite normalizing flows. We can choose different schemes to discretize the continuous dynamics and solve the Neural ODEs numerically. Then we use different backpropagation algorithms to train with the CNF network to minimize the loss function. These schemes are discussed in the next section.

2.3.3 Loss Function

We need to choose a metric to measure the training and evaluation loss for CNF model. For example, the loss functional usually used for time-series regression is the Euclidean distance between the approximate value of the objective computed by model and the true value of the objective. However, the goal of continuous normalizing flows is to estimate the unknown posterior distribution and we only have the samples from the unknown target distribution. This is so called unsupervised learning since we do not have the true value of the target distribution probability.

We hereby introduce the Kullback–Leibler divergence [Kullback and Leibler, 1951], a statistical distance, which is normally chosen to measure the difference between two probability distributions. Suppose we have two distributions, P is the obsevations or a known measured probability distribution, Q is the model we use to approximate the distribution P. Then for P and Q defined on the same probability space $\mathcal{X}$, the discrete Kullback–Leibler divergence is given by

$$D_{KL}(P||Q) = \sum_{\mathcal{X}} P(x) \log\left(\frac{P(x)}{Q(x)}\right). \tag{2.15}$$

The continuous Kullback–Leibler divergence is given by

$$D_{KL}(P||Q) = \int_{\mathcal{X}} p(x) \log\left(\frac{p(x)}{q(x)}\right) dx \tag{2.16}$$

Then, consider the continuous normalzing flow model, denote the unknown target density as ρ_0. The model we bulid to approximate ρ_0 is represented by $p_{\vec{\mathbf{x}_0}}(\mathbf{x}(\vec{0}))$, and $p_{\vec{\mathbf{x}_T}}(\mathbf{x}(\vec{T}))$ is the base distribution we mentioned before, i.e. standard normal distribution. The Kullback–Leibler divergence for the CNF is given by:

$$
\begin{aligned}
\mathcal{L}(\boldsymbol{\theta}, \vec{\mathbf{x}}) =& D_{KL}(\rho_0 || p_{\vec{\mathbf{x}}_0}), \\
=& \int \rho_0(\vec{\mathbf{x}}(0)) \log \frac{\rho_0(\vec{\mathbf{x}}(0))}{p_{\vec{x}_0}(\vec{\mathbf{x}}(0))} d\vec{\mathbf{x}}(0), \\
=& -\mathbb{E}_{\rho_0}[\log(p_{\vec{\mathbf{x}}_0}(\vec{\mathbf{x}}(0))] + const, \\
=& -\mathbb{E}_{\rho_0}\left[\log(p_{\vec{\mathbf{x}}_T}(\vec{\mathbf{x}}(T))) + \int_0^T tr(\frac{\partial \boldsymbol{\ell}(\vec{\boldsymbol{\theta}}(t), \vec{\mathbf{x}}(t), t)}{\partial \vec{\mathbf{x}}(t)})\right] + const, \\
=& -\mathbb{E}_{\rho_0}\left[-\frac{d_0}{2}\log(2\pi) - \frac{1}{2}\left\|\vec{\mathbf{x}}(T)\right\|^2 + \int_0^T tr(\frac{\partial \boldsymbol{\ell}(\vec{\boldsymbol{\theta}}(t), \vec{\mathbf{x}}(t), t)}{\partial \vec{\mathbf{x}}(t)})\right] + const, \qquad (2.17) \\
=& \mathbb{E}_{\rho_0}\left[const + \frac{1}{2}\left\|\vec{\mathbf{x}}(T)\right\|^2 - \int_0^T tr(\frac{\partial \boldsymbol{\ell}(\vec{\boldsymbol{\theta}}(t), \vec{\mathbf{x}}(t), t)}{\partial \vec{\mathbf{x}}(t)})\right] + const, \\
=& \mathbb{E}_{\rho_0}\left[const + \frac{1}{2}\left\|\vec{\mathbf{x}}_0 + \int_0^T \ell(\boldsymbol{\theta}(t), \vec{\mathbf{x}}(t), t)dt\right\|^2 - \int_0^T tr(\frac{\partial \boldsymbol{\ell}(\vec{\boldsymbol{\theta}}(t), \vec{\mathbf{x}}(t), t)}{\partial \vec{\mathbf{x}}(t)})\right] \\
& + const.
\end{aligned}
$$

Notice d_0 is the dimension of $\mathbf{x}(\vec{t})$, $\mathbf{x}(\vec{t}) \in \mathbb{R}^d, \ t = (0, T]$.

During training, we cannot necessarily evaluate the target density ρ_0, so we instead estimate the expectation over ρ_0 by Monte Carlo that gives an approximate value for the expectation.

Notice that minimizing $D_{KL}(\rho_0||p_{x_0})$ is equivalent to minimizing $-\mathbb{E}_{\rho_0}[log(p_{\vec{\mathbf{x_0}}}(\mathbf{x}(\vec{0}))]$, hence it is also equivalent to maximize $\mathbb{E}_{\rho_0}[p_{\vec{\mathbf{x_0}}}(\mathbf{x}(\vec{0})]$. This leads us to training the continuous normalizing flow for variational inference by maximum likelihood estimation, so we can view the training for a CNF as maximum likelihood training.

In the next section we give more details about numerical methods of training the CNF.

2.4 Discretization Schemes

Training a Neural ODE can viewed as an infinite-dimensional optimal control problem where the weights are controls and the hidden features are the states [Onken and Ruthotto, 2020]. Taking a standard neural ODE as an example, the goal of training is to minimize a loss function subject to the ODE constraint with initial value limit. Here, in order to distinguish trainable parameters represented by weight matrix and bias vector, we use a new notation $\boldsymbol{\theta}(t) = (\mathbf{W}(t), \vec{\mathbf{b}(t)})$ to describe the Neural ODE model:

$$\begin{aligned} &\min_{\boldsymbol{\theta},\vec{\mathbf{x}}} \ \mathcal{L}(\boldsymbol{\theta}, \vec{\mathbf{x}}), \\ &\partial_t \vec{\mathbf{x}}(t) = \ell(\boldsymbol{\theta}(t), \vec{\mathbf{x}}(t), t), \ \text{for } t \in (0, T], \\ &\vec{\mathbf{x}}(0) = \vec{\mathbf{x}}_0, \end{aligned} \tag{2.18}$$

where $\vec{\mathbf{x}}(t)$ satisfy the neural ODEs for initial values given by the training data, and $\mathcal{L}$ is the optimization objective of the whole model, i.e. the loss function.

We can not solve the continuous optimization problem directly through theoretical analysis. Hence numerical methods are applied to train and evaluate the model. Here are two schemes to do the discretization for Neural ODEs so we can solve the gradient of the loss function with respect to trainable parameters numerically.

2.4.1 Discretize-Then-Optimize

The first approach we discuss is the Discretize-Then-Optimize method (DTO). This method is to discretize the ODE network firstly by an explicit ODE solver scheme (e.g. forward Euler, RK4), then apply the backpropagation during training to solve the gradient of loss function with respect to trainable parameters. Afterwards, we can execute gradient-related optimization algorithm (e.g. gradient descent) to minimize the loss function.

We take the Euler method as an example to show how DTO works. Notice the ODE network is given by:

$$\partial_t \vec{\mathbf{x}}(t) = \ell(\boldsymbol{\theta}(t), \vec{\mathbf{x}}(t), t), \quad \text{for } t \in (0, T]. \tag{2.19}$$

Then we discretize the continuous ODE by applying Euler method to N layers with fixed step size h:

$$\vec{\mathbf{x}}_n = \vec{\mathbf{x}}_{n-1} + h\ell(\boldsymbol{\theta}_{n-1}, \vec{\mathbf{x}}_{n-1}, t_{n-1}), \quad \text{for } n = 1, 2, 3, ..., N, \tag{2.20}$$

where $h = \frac{T}{N}$. This is the same as the ResNet model structure.

By equation (2.20), training the Neural ODE model in CNF is transferred to a finite-dimensional optimization problem (Gunzburger, 2003). Given the initial condition $\vec{\mathbf{x}}(0) =$

$\vec{\mathbf{x}}_0$, we can solve $\vec{\mathbf{x}}_1$, $\vec{\mathbf{x}}_2$, $\vec{\mathbf{x}}_3$, ..., $\vec{\mathbf{x}}_N$ by forward propagation via equation (2.20). Afterwards, we can naturally compute $\frac{\partial \mathcal{L}}{\partial \vec{\mathbf{x}}_N}$ as well as $\frac{\partial \ell(\boldsymbol{\theta}_n, \vec{\mathbf{x}}_n, t_n)}{\partial \vec{\mathbf{x}}_n}$ for $n = 1, 2, 3, ..., N$.

Subsequently, we can apply backpropagation algorithm to solve the gradient of loss function with respect to trainable parameters.

According to backpropagation, suppose given $\frac{\partial \mathcal{L}}{\partial \vec{\mathbf{x}}_n}$, we can compute $\frac{\partial \mathcal{L}}{\partial \vec{\mathbf{x}}_{n-1}}$ by chain rule:

$$\begin{aligned} \frac{\partial \mathcal{L}}{\partial \vec{\mathbf{x}}_{n-1}} =& \frac{\partial \mathcal{L}}{\partial \vec{\mathbf{x}}_n} \frac{\partial \vec{\mathbf{x}}_n}{\partial \vec{\mathbf{x}}_{n-1}} \\ =& \frac{\partial \mathcal{L}}{\partial \vec{\mathbf{x}}_n} (I + h \frac{\partial \ell(\boldsymbol{\theta}_{n-1}, \vec{\mathbf{x}}_{n-1}, t_{n-1})}{\partial \vec{\mathbf{x}}_{n-1}}), \qquad \text{for } n = 1, 2, 3, ..., N. \end{aligned} \tag{2.21}$$

Denote the adjoint state at time t as $\vec{\mathbf{a}}(t) = \frac{\partial \mathcal{L}}{\partial \vec{\mathbf{x}}(t)}$. Then equation (2.21) can be replaced by:

$$\begin{aligned} \vec{\mathbf{a}}_{n-1} =& \vec{\mathbf{a}}_n (I + h \frac{\partial \ell(\boldsymbol{\theta}_{n-1}, \vec{\mathbf{x}}_{n-1}, t_{n-1})}{\partial \vec{\mathbf{x}}_{n-1}}), \\ =& \vec{\mathbf{a}}_n + h \frac{\partial \ell(\boldsymbol{\theta}_{n-1}, \vec{\mathbf{x}}_{n-1}, t_{n-1})}{\partial \vec{\mathbf{x}}_{n-1}} \vec{\mathbf{a}}_n, \qquad \text{for } n = 1, 2, 3, ..., N. \end{aligned} \tag{2.22}$$

We can then compute the gradient with respect to trainable parameters by the chain rule:

$$\begin{aligned} \frac{\partial \mathcal{L}}{\partial \boldsymbol{\theta}_{n-1}} =& h \frac{\partial \ell(\boldsymbol{\theta}_{n-1}, \vec{\mathbf{x}}_{n-1}, t_{n-1})}{\partial \boldsymbol{\theta}_{n-1}} \vec{\mathbf{a}}_{n-1}, \\ \vec{\mathbf{a}}(T) =& \frac{\partial \mathcal{L}}{\partial \vec{\mathbf{x}}(T)}, \\ \vec{\mathbf{a}}_{n-1} =& \vec{\mathbf{a}}_n + h \frac{\partial \ell(\boldsymbol{\theta}_{n-1}, \vec{\mathbf{x}}_{n-1}, t_{n-1})}{\partial \vec{\mathbf{x}}_{n-1}} \vec{\mathbf{a}}_n, \qquad \text{for } n = 1, 2, 3, ..., N. \end{aligned} \tag{2.23}$$

Then, we can apply a gradient-based optimization algorithm for the training.

2.4.2 Optimize-Then-Discretize

An alternative numerical method of solving the Neural ODE in the CNF model is Optimize-Then-Discretize approach, which is firstly introduced by [Chen et al., 2018]. This method is also named as adjoint-based approach. The idea is, we directly optimize the ODE network and then discretize the ODE after training. All the values of hidden dynamics are computed by a black-box differential equation solver.

Given the ODE with initial value condition shown in equation (2.18), we can solve $\vec{\mathbf{x}}(T)$ by an arbitrary ODE solver which is treated as a black-box solver. Then we can naturally compute $\vec{\mathbf{a}}(T) = \frac{\partial \mathcal{L}}{\partial \vec{\mathbf{x}}(T)}$ and solve the gradient of loss function with respect to trainable parameters via OTD approach. This is done by a reverse-mode derivative of an ODE initial value problem that we take the $\vec{\mathbf{x}}(T)$ as the initial condition and solve the ODE backward over the CNF network by an arbitrary ODE solver.

In OTD method, the gradients are computed by solving the adjoint equation:

$$\begin{aligned} \frac{\partial \mathcal{L}}{\partial \boldsymbol{\theta}(t)} &= \vec{\mathbf{a}}(t)\frac{\partial \ell(\boldsymbol{\theta}(t), \vec{\mathbf{x}}(t), t)}{\partial \boldsymbol{\theta}(t)}, \\ \vec{\mathbf{a}}(T) &= \frac{\partial \mathcal{L}}{\partial \vec{\mathbf{x}}(T)}, \\ \frac{\partial \vec{\mathbf{a}}(t)}{\partial t} &= -\vec{\mathbf{a}}(t)\frac{\partial \ell(\boldsymbol{\theta}(t), \vec{\mathbf{x}}(t), t)}{\partial \vec{\mathbf{x}}(t)}, \qquad \text{for } t \in (0, T]. \end{aligned} \tag{2.24}$$

According to Chen et al. (2018), the adjoint state is solved backward in time by an accurate ODE solver. Taking the Euler method as an example, the numerical adjoint method can be viewed as a discretization of equation (2.24):

$$\begin{aligned} \frac{\partial \mathcal{L}}{\partial \boldsymbol{\theta}_{n-1}} =& h\frac{\partial \ell(\boldsymbol{\theta}_{n-1}, \vec{\mathbf{x}}_{n-1}, t_{n-1})}{\partial \boldsymbol{\theta}_{n-1}}\vec{\mathbf{a}}_{n-1}, \\ \vec{\mathbf{a}}(T) =& \frac{\partial \mathcal{L}}{\partial \vec{\mathbf{x}}(T)}, \\ \vec{\mathbf{a}}_{n-1} =& \vec{\mathbf{a}}_n + h\frac{\partial \ell(\boldsymbol{\theta}_n, \vec{\mathbf{x}}_n, t_n)}{\partial \vec{\mathbf{x}}_n}\vec{\mathbf{a}}_n, \qquad \text{for } n = 1, 2, 3, ..., N. \end{aligned} \tag{2.25}$$

Comparing the OTD scheme (2.25) with DTO scheme (2.23), we notice the only difference between them is how they update the adjoint state value. The DTO method updates the adjoint state at layer $n-1$ via evaluating ℓ at layer $n-1$ while for OTD method updates the adjoint state at layer $n-1$ via evaluating ℓ at layer n.

For the reverse-mode automatic differentiation algorithm of ODE solutions for OTD method, we do not need to remember the intermediate hidden dynamics in forward propagation, which is different from backpropagation algorithm, thus saving computer memory.

However, we after need to solve the backward ODE with very small time steps to get sufficient accuracy for the gradient. It needs more computations than OTD method since we also need to recompute intermediate hidden dynamics via backward accumulative numerical ODE integration. This might lead to more training time for OTD approach.

Besides, according to [Gholami et al., 2019], the adjoint-based method is not consistent with forward propagation, and we incorectly replace the input of the neural network with the backward ODE solution from the its output. As a consequence, the quality of the gradient of loss function with respect to the trainable parameters can deteriorate. In our numerical experiments, we use adaptive step-size ODE solver to somehow guarantee the accuracy of OTD scheme, but this can also cause more computational cost while training.

Chapter 3

Methodology

In this chapter, we introduce the spline-based layers [Günther et al., 2021] to CNF with the structure of layers built in our neural network. We firstly introduce the definition of the B spline basis functions and the explain the intuitive of using splines in our network. We further discuss some properties of B-spline basis functions. Last, we define our model functions of our spline-based layers while comparing with concatsquash layers which is one of the well-behaved hyper network for reversible generative model according to [Grathwohl et al., 2018].

3.1 Spline Functions

In previous sections, we treat training a CNF as an optimal control problem. Instead of using the same bunch of trainable parameters $\boldsymbol{\theta}$ across all the layers, we make trainable parameters dependent on time denoted as $\boldsymbol{\theta}(t)$.

When we apply numerical methods to solve CNF, we need to discretize the whole network. According to Section 2.4, if we discretize the network to N layers, then we get a set of θ_i as trainable parameters, $i = 1, 2, 3, ..., N$. Typically, when we use a smaller step size to discretize the CNF network, we can get better accuracy, but the number of trainable parameters also increases that leading to a growing complexity of the optimal control problem. Meanwhile, if we use a larger step size for training, it can not guarantee the sufficient continuity of the CNF model, this may also bring about unstable forward propagation [Günther et al., 2021].

In this research paper, we make use of spline-based method to specify the $\theta(t)$ that can decouple the trainable parameters from the layers of the Neural ODE while maintaining the time dependency of $\boldsymbol{\theta}(t)$. More specifically, we choose B-spline basis functions to parameterize $\boldsymbol{\theta}(t)$ by a fixed number of trainable parameters. Thus, we can use sufficiently small step size when discretizing the Neural ODE in CNF while not increasing the complexity of the whole model.

B-Spline basis functions: Here, we display the definition of one-dimensional B-spline basis functions. Given $t \in [0, T]$, we divide $[0, T]$ into L equal-length intervals, therefore we have (L+1) equally spaced knots, i.e. $0 = \tau_0 < \tau_1 < ... < \tau_L = T$. Any spline function of given degree d can be expressed as a linear combination of B-spline functions of that degree. We denote a B-spline function as $B_{l,d}(t)$, where d is the chosen degree of the spline function, l is the subscript of knots, with $d, l \in \mathbb{N}$, $0 \leq l \leq L$:

$$B_{l,d}(t) = \frac{t - \tau_l}{\tau_{l+d} - \tau_l} B_{l,d-1}(t) + \frac{\tau_{l+d+1} - t}{\tau_{l+d+1} - \tau_{l+1}} B_{l+1,d-1}(t), \quad for\ d \geq 1 \tag{3.1}$$

$$B_{l,0}(t) = \begin{cases} 1 & for\ t \in [\tau_l, \tau_{l+1}), \\ 0 & otherwise \end{cases}. \tag{3.2}$$

Notice the B-spline function of degree d, $B_{l,d}(t)$, is a piecewise polynomial function of degree $d - 1$. Given all the knots are different, the first $(d - 2)$ derivatives of $B_{l,d}(t)$ are continuous across the knots, and $B_{l,d}(t) \geq 0$ for all d, l.

Each B-spline basis function $B_{l,d}(t)$ is computed by a combination of $B_{l,0}(t)$, $B_{l+1,0}(t)$, ... , $B_{l+d,0}(t)$, so each B-spline basis function $B_{l,d}(t)$ has a local support in $[\tau_l, \tau_{l+d+1})$. Besides, for any fixed d and $t \in [\tau_l, \tau_{l+1})$, only $B_{l-d,d}(t)$, $B_{l-d+1,d}(t)$, ... , $B_{l,d}(t)$ are non-zero. Important properties of B-spline basis functions are described by Figure 3.1 and Figure 3.2.

Therefore, we can compute the B-spline function values for any $t \in [0, T]$ given degree d and number of intervals L divided by $(L + 1)$ knots.

3.2 Parameterization via B-spline Basis Functions

The optimization problem for CNF is specified by:

$$\begin{aligned}
\min_{\boldsymbol{\theta},\vec{\mathbf{x}}} \quad & \mathcal{L}(\boldsymbol{\theta},\vec{\mathbf{x}}), \\
\partial_t \vec{\mathbf{x}}(t) &= \ell(\boldsymbol{\theta}(t),\vec{\mathbf{x}}(t),t), \;\; for\ t \in (0,T], \\
\frac{\partial \log p(\vec{\mathbf{x}}(t))}{\partial t} &= -tr\left(\frac{\partial \boldsymbol{\ell}(\boldsymbol{\theta}(t),\vec{\mathbf{x}}(t),t)}{\partial \vec{\mathbf{x}}(t)}\right), \quad for\ t \in (0,T], \\
\vec{\mathbf{x}}(0) &= \vec{\mathbf{x}}_0.
\end{aligned} \tag{3.3}$$

By multi-layer artificial neural network modeling, $\ell(\boldsymbol{\theta}(t),\vec{\mathbf{x}}(t),t)$ can be specified by a linear transformation plus activation function $\ell(\boldsymbol{\theta}(t),\vec{\mathbf{x}}(t),t) = \mathcal{G}(\mathbf{W}(t)\vec{\mathbf{x}}(t) + \vec{\mathbf{b}}(t))$, where $\boldsymbol{\theta}(t) = (\mathbf{W}(t),\vec{\mathbf{b}}(t))$ is the set of trainable parameters, $\mathbf{W}(t)$ is a linear transformation matrix, $\vec{\mathbf{b}}(t)$ is the bias vector, and $\mathcal{G}$ is the activation function in the neural network.

We apply the spline-based network for neural network proposed by [Günther et al., 2021], which define $\boldsymbol{\theta}(t) = (\mathbf{W}(t),\vec{\mathbf{b}}(t))$ using B-spline basis functions:

$$\begin{aligned}
\mathbf{W}(t) &= \sum_{l=-d}^{L-1} \omega_l B_{l,d}(t), \\
\vec{\mathbf{b}(t)} &= \sum_{l=-d}^{L-1} \vec{\beta}_l B_{l,d}(t),
\end{aligned} \tag{3.4}$$

where L is the number of time intervals divided by $(L+1)$ knots over $[0,T]$, d is the degree we choose for B-spline basis functions, and $\mathbf{W}(t), \omega_l \in \mathbb{R}^{d_{out} \times d_{in}}$, $\vec{\mathbf{b}}(t), \vec{\beta}_l \in \mathbb{R}^{d_{out}}$. Here, d_{out}, d_{in} are the output dimension and the input dimension determined by the neural network settings.

3.3 Spline-based layers in CNF

In the previous section, we introduced a general form of a neural network layer parameterized by spline functions. However, in practice, we usually try a more complex structure of multiple layers and various kinds of activation functions. Here, we propose a spline-based neural network to model the derivative of continuous dynamics $\vec{\mathbf{x}}(t)$ with respect to time, denoted as $\ell(\boldsymbol{\theta}(t),\vec{\mathbf{x}}(t),t)$ in Equation (3.3). We make use of multiple spline-based layers to construct $\ell(\boldsymbol{\theta}(t),\vec{\mathbf{x}}(t),t)$.

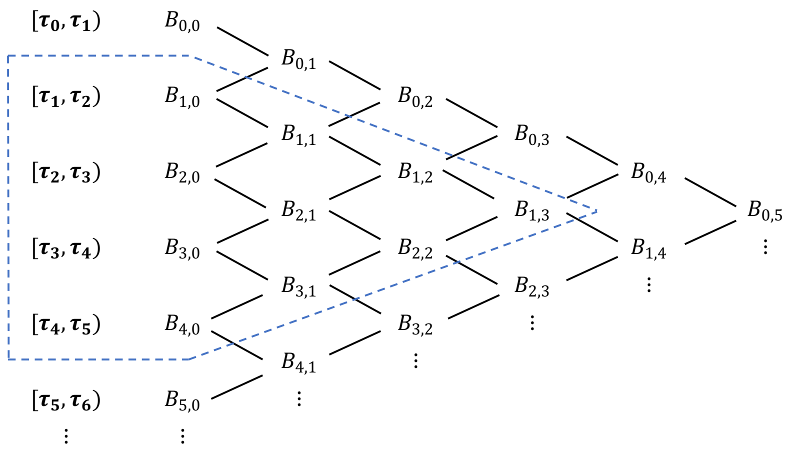


Figure 3.1: This blue area shows the non-zero interval with respect to $B_{1,3}(t)$. More generally, by definition of B-spline basis functions, $B_{l,0}$ is only non-zero when $t \in [\tau_l, \tau_{l+1})$, $B_{l+1,0}$ is only non-zero when $t \in [\tau_{l+1}, \tau_{l+2})$. Then, by simple calculation, $B_{l,1}$ must be non-zero when $t \in [\tau_l, \tau_{l+2})$, more specifically, $B_{l,1} > 0$ for all $t \in [\tau_l, \tau_{l+2})$. By accumulative calculation, $B_{l,d}$ is non-zero in $[\tau_l, \tau_{l+d+1})$.

Spline-based layer: Assume the continuous dynamic input $\vec{\mathbf{x}}(t) \in \mathbb{R}^{d_{in}}$, and the output dimension of the spline-based layer is d_{out}, then a spline based layer is defined as below:

$$S_{d_{out},d_{in}}(\vec{\mathbf{x}}(t), \boldsymbol{\theta}, t) = \mathcal{G}\left(\sigma(\vec{\mathbf{w}}(t)) \odot \mathbf{D}_{\boldsymbol{\theta}_f}(\vec{\mathbf{x}}(t)) + \vec{\mathbf{b}}(t)\right), \tag{3.5}$$

where $\mathbf{D}_{\boldsymbol{\theta}_f}(\vec{\mathbf{x}}(t))$ is an affine function modeled by a set of fixed trainable parameters $\boldsymbol{\theta}_f = (\mathbf{W}_f, \vec{\mathbf{b}}_f)$ with respect to the input $\vec{\mathbf{x}}(t)$:

$$\mathbf{D}_{\boldsymbol{\theta}_f}(\vec{\mathbf{x}}(t)) = \mathbf{W}_f \vec{\mathbf{x}}(t) + \vec{\mathbf{b}}_f, \tag{3.6}$$

where $\mathbf{W}_f \in \mathbb{R}^{d_{out} \times d_{in}}$, $\vec{\mathbf{b}}_f \in \mathbb{R}^{d_{out}}$.

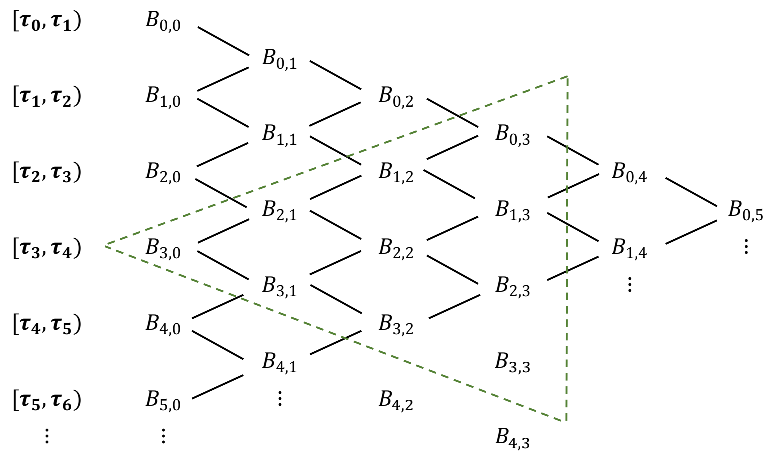


Figure 3.2: This green area shows the non-zero functions in interval $[\tau_3, \tau_4)$. More generally, given $t \in [\tau_l, \tau_{l+1})$, for any fixed d, only $B_{l-d,d}(t)$, $B_{l-d+1,d}(t)$, ... , $B_{l,d}(t)$ are non-zero, thus the total number of non-zero functions is $(d+1)$.

The time-dependent weight matrix and bias vector are given by a linear combination of B-spline basis functions:

$$\begin{aligned}\vec{\mathbf{w}}(t) &= \sum_{l=0}^{L+d-1} \vec{\omega}_l B_{l,d}(t) \\ \vec{\mathbf{b}}(t) &= \sum_{l=0}^{L+d-1} \vec{\beta}_l B_{l,d}(t)\end{aligned} \tag{3.7}$$

where $\vec{\omega}_l, \vec{\beta}_l, \vec{\mathbf{w}}(t), \vec{\mathbf{b}}(t) \in \mathbb{R}^{d_{out}}$.

Note that the trainable parameters for each layer are composed of $\mathbf{W}_f \in \mathbb{R}^{d_{out} \times d_{in}}$, $\vec{\mathbf{b}}_f \in \mathbb{R}^{d_{out}}$, $\vec{\omega}_l, \vec{\beta}_l \in \mathbb{R}^{d_{out}}$. For multi-layer spline-based neural network, we need to choose different input dimension, output dimension, and the activation type for each layer. In our numerical experiments, we compare four different structure of spline-based neural networks which can be discussed later in Chapter 4. Here we give an example to show how we model the gradient function by multiple spline-based layers.

Suppose we construct a right-hand side function using 4 spline-based layers, with intermediate dimensions d_1, d_2, d_3 where the input dimension and the output dimension of the neural network should be the same since the gradient of $\vec{\mathbf{x}}(t)$ with respect to t should be in the same dimension as $\vec{\mathbf{x}}(t)$. Denote the input dimension of $\vec{\mathbf{x}}(t)$ as d_0, then for any $t \in (0, T]$, $\ell(\boldsymbol{\theta}(t), \vec{\mathbf{x}}(t), t)$ is given by:

$$\begin{aligned} \vec{\mathbf{x}}_1(t) =& S_{d_1,d_0}(\vec{\mathbf{x}}(t), \boldsymbol{\theta}_1, t) \\ \vec{\mathbf{x}}_2(t) =& S_{d_2,d_1}(\vec{\mathbf{x}}_1(t), \boldsymbol{\theta}_2, t) \\ \vec{\mathbf{x}}_3(t) =& S_{d_3,d_2}(\vec{\mathbf{x}}_2(t), \boldsymbol{\theta}_3, t) \\ \ell(\boldsymbol{\theta}(t), \vec{\mathbf{x}}(t), t) =& S_{d_0,d_3}(\vec{\mathbf{x}}_3(t), \boldsymbol{\theta}_4, t) \end{aligned} \tag{3.8}$$

Here, $\boldsymbol{\theta}_i$ is the set of trainable parameters for layer i. For each layer, we use the logistic sigmoid function of (2.4) to specify the σ function. And for the first three layers, we use the hyperbolic tangent function of (2.3) to define the activation function $\mathcal{G}$. For the last layer, we do not add any activation function, i.e. $S_{d,d_3}(\vec{\mathbf{x}}_3(t), \boldsymbol{\theta}_4, t) = \sigma(\vec{\mathbf{w}}_4(t)) \odot \mathbf{D}_{\boldsymbol{\theta}_{f4}}(\vec{\mathbf{x}}_3(t)) + \vec{\mathbf{b}}_4(t)$, avoiding limiting the range of the output value.

To evaluate the performance of our model, we compare the spline-based layers with concatsquash-linear layers that perform quite well in image classification [Grathwohl et al., 2018], time-series regression and generative probabilistic modeling [Onken and Ruthotto, 2020].

Concatsquash-linear layer: Denote a concatsquash-linear layer as $C_{d_{out},d_{in}}$, with input dimension d_{in}, i.e. $\vec{\mathbf{x}}(t) \in \mathbb{R}^{d_{in}}$ and output dimension d_{out}. The structure of the concatsquash linear layers is as follows:

$$C_{d_{out},d_{in}}(\vec{\mathbf{x}}(t), \boldsymbol{\theta}, t) = \mathcal{G}\left(\sigma(\mathbf{D}_{\boldsymbol{\theta}_t}(t)) \odot \mathbf{D}_{\boldsymbol{\theta}_x}(\vec{\mathbf{x}}(t)) + \mathbf{D}_{\boldsymbol{\theta}_k}(t)\right) \tag{3.9}$$

Each $\mathbf{D}_{\boldsymbol{\theta}}$ is an affine function with respect to the function input:

$$\mathbf{D}_{\boldsymbol{\theta}_t}(t) = \vec{\mathbf{w}}_t t + \vec{\mathbf{b}}_t, \tag{3.10}$$

where $\vec{\mathbf{w}}_t, \vec{\mathbf{b}}_t \in \mathbb{R}^{d_{out}}$;

$$\mathbf{D}_{\boldsymbol{\theta}_x}(\vec{\mathbf{x}}(t)) = \mathbf{W}_x \vec{\mathbf{x}}(t) + \vec{\mathbf{b}}_x, \tag{3.11}$$

where $\mathbf{W}_x \in \mathbb{R}^{d_{out} \times d_{in}}$, $\vec{\mathbf{b}}_x \in \mathbb{R}^{d_{out}}$; and

$$\mathbf{D}_{\boldsymbol{\theta}_k}(t) = \vec{\mathbf{w}}_k t \tag{3.12}$$

where $\vec{\mathbf{w}}_k \in \mathbb{R}^{d_{out}}$. Note that there is no bias vector in (3.12).

3.4 Discretization for Spline-based CNF

Training for spline-based CNF under DTO and OTD schemes both require discretization when evaluating functionals over [0,T]. Considering t_i as the i_{th} grid point after discretization over time, we can evaluate the spline-based functions at t_i,

$$\begin{aligned} \vec{\mathbf{W}}(t_i) &= \sum_{l=-d}^{L-1} \vec{\omega}_l B_{l,d}(t_i) \\ \vec{\mathbf{b}}(t_i) &= \sum_{l=-d}^{L-1} \vec{\beta}_l B_{l,d}(t_i) \end{aligned} \tag{3.13}$$

Notice the number of time steps is independent of the number of trainable parameters, which means reducing the time step while training does not increase the number of trainable parameters, which is fixed by design.

Then we can additionally use the re-discretization trick when doing the evaluation. Since the trainable parameters are not dependent in time, we can use different step size as well as ODE solver for training and evaluation. Training with a tiny step size can be time-consuming, we can instead train the whole network by a larger step size (e.g. 0.05), but do the evaluation by a much smaller step size. It can increase the continuity of the network and model accuracy for evaluation while maintaining an efficient training process. Further details about numerical experiments are discussed in the next Chapter.

Chapter 4

Numerical Experiments

In this chapter, we execute numerical experiments on 2D toy data that we know the ground truth for density estimation. We investigate the performance as well as the invertibility of our spline-based CNF models, and do comparisons between DTO and OTD discretization schemes. Our goal is to compare the model performance under different discretization schemes and investigate the influence of some hyperparameyers such as degree d. Further, we compare our spline-based CNF with concatsquash-based CNF (Gholami, 2019) which is constructed by concatsquash linear layers and behaves pretty well on several density estimation tasks.

We display some tables and compare metrics among different experiments. The metrics we use include Number of Parameters (NOP), Number of Forward Evaluations (NFE), Negative Log Likelihood (NLL) and the inverse error. Notice the NLL is the loss function value for testing case, and inverse error is defined as below.

Suppose the mapping function from the target distribution to the base distribution is denoted by f, then the inverse error is given by:

$$\Delta_{inv}(\vec{\mathbf{x}_0}) = \left\| f^{-1}(\boldsymbol{\theta}, f(\boldsymbol{\theta}, \vec{\mathbf{x}_0})) - \vec{\mathbf{x}_0}) \right\| \tag{4.1}$$

Inverse error is used for measuring the invertibility of the CNF model. For sampling task, the f^{-1} is required for computing. However, for density estimation task, this metric is not as important as in sampling since the f^{-1} is not needed.

4.1 Density Estimation via Spline-based Layers

In this section, we train the model of four spline-based layers on checkerboard 2D toy data, with layers' dimensions $d_0 = 2$, $d_1 = 16$, $d_2 = 16$, $d_3 = 16$. We set number of knots intervals $L = 5$, degree of spline functions $d = 2$, learning rate $= 10^{-3}$, $T = 0.5$, and apply Adam as the optimizer of the neural network. Notice for DTO scheme, we use RK4 as ODE solver with fixed step size 0.05. For OTD scheme, we use dopri5 solver which is an adaptive ODE solver and set $atol = rtol = 10^{-5}$. We can obtain results shown in Table 4.1:

Table 4.1: Results of spline-based model on checkerboard

T	Scheme	Solver	NOP	NFE	NLL	Inverse Error
0.5	DTO	RK4	1327	40	3.7074	2.7579×10^{-5}
	OTD	dopri5	1327	320	3.7234	2.2769×10^{-6}

According to the results shown in Table 4.1, the testing loss (NLL) of DTO approach is a little bit smaller than that of OTD while the number of forward evaluations is drastically less than OTD approach. This shows the DTO approach can achieve a better accuracy with a smaller compuational cost for density estimation.

Subsequently, we compare the inverse errors and can easily observe that the inverse error of the OTD scheme is smaller than the DTO scheme. One of the reasons is that the number of evaluations of the OTD scheme is much more than that of the DTO scheme, which makes the step size for discretization smaller in OTD scheme, causing the neural network to be more continuous.

Figure 4.1 and Figure 4.2 show the evaluation results of DTO scheme and OTD scheme for checkerboard dataset respectively. Suppose we have mapping function f that can map $\mathbf{x}$ from the target distribution to $\mathbf{y}$ from the standard normal distribution, i.e. $\mathbf{y} = f(\vec{\mathbf{x}})$. The top left subfigure is the samples from the checkerboard distribution, denoted as $\mathbf{x}$; the top right subfigure shows $f(\mathbf{x})$, where f is constructed by our CNF model; the bottom left subfigure displays $f^{-1}(f(\mathbf{x}))$; the bottom right subfigure shows $f^{-1}(\mathbf{y})$, where $\mathbf{y}$ is sampled from standard normal distribution. The inverse error of each model can be expressed as the Euclidiean distance between $\mathbf{x}$ and $f^{-1}(f(\mathbf{x}))$, which reflects the invertibility of the model. For density estimation, we expect to have the $f(\mathbf{x})$ follows standard normal distribution and $f^{-1}(\mathbf{y})$ follows the target distribution. According to these two figures, we can find that under both OTD and DTO schemes, our spline-based model can map the sample data from target distribution to nearly standard normal distribution (from top right subfigure), and can map samples from standard normal distribution backward to approach to target

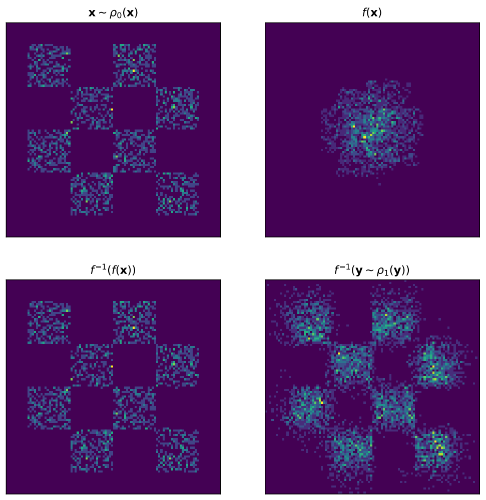


Figure 4.1: DTO for checkerborad

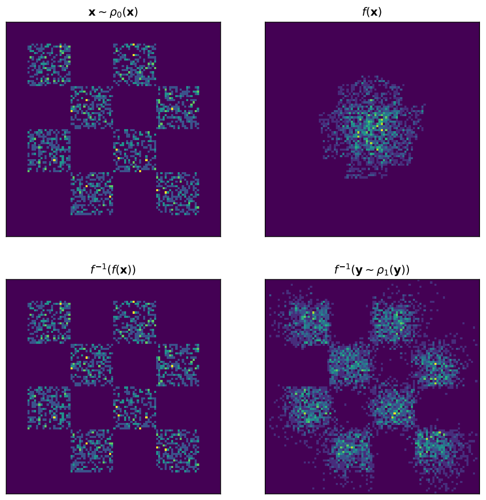


Figure 4.2: OTD for checkerborad

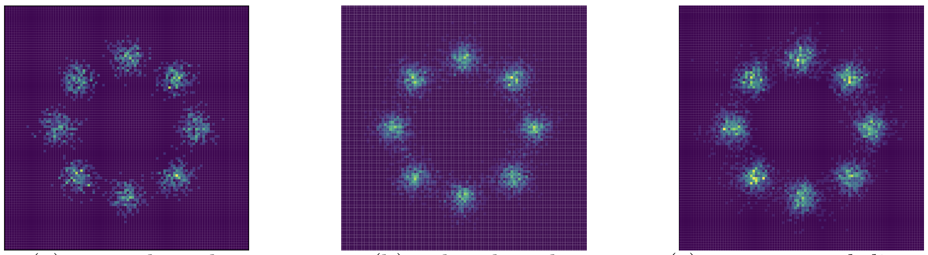

(a) ground truth (b) spline-based (c) concatsquash-linear

Figure 4.3: Results on 8gaussians

distribution (from bottom right subfigure). Besides, by comparison between top left and bottom left subfigures, we can observe the invertibility of our model is quite good because there is little difference between $\mathbf{x}$ and $\mathbf{x}$ and $f^{-1}(f(\mathbf{x}))$.

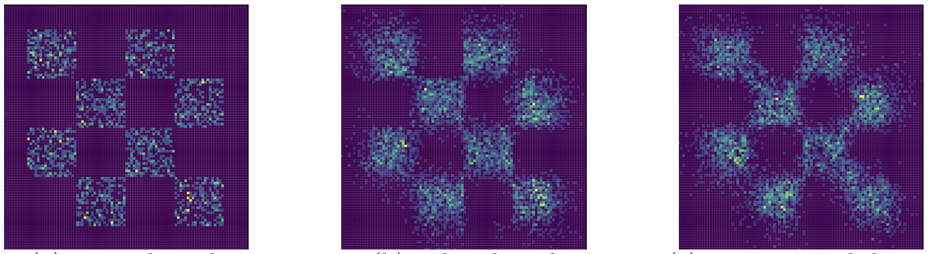
4.2 Compare with Concatsquash-Linear Layers

To evaluate the performance of spline-based layers, we compare our models with concatsquash-linear layers that behave well on other Neural ODE problems such as image classification [Gholami et al., 2019], time-series regression and continuous normalizing flows [Onken and Ruthotto, 202
We train the model of four spline-based layers and four concatsquash-linear layers on different datasets, with $d_0 = 2$, $d_1 = 16$, $d_2 = 16$, $d_3 = 16$.

Table 4.2: Results on 8gaussians

Layer Type	Scheme	Solver	NOP	NFE	NLL	Inverse Error
Spline-based	DTO	RK4	1327	40	2.9259	1.4907×10^{-6}
	OTD	dopri5	1327	74	2.8975	5.2907×10^{-6}
Concatsquash-linear	DTO	RK4	777	40	**2.8632**	5.4040×10^{-6}
	OTD	dopri5	777	116	**2.8943**	8.2833×10^{-6}

According to results shown in Table 4.2 and Figure 4.2, we can observe that both spline-based layers and concatsquash-linear layers have similar performance on 8gaussians dataset with respect to NLL and inverse error values. Compared to the ground truth, both models can have a good approximation to the 8gaussians distribution. However, it is easy to build bijective mapping function between 8gaussians to a standard normal distribuion.



(a) ground truth (b) spline-based (c) concatsquash-linear

Figure 4.4: Results on checkerboard

Further, we want to investigate whether spline-based layers can deal with more complex distributions than 8gaussians, so we also conduct numerical experiments on checkerboard, 2spirals, swissroll, to which it is hard to build the explicit mapping function from standard normal distribution.

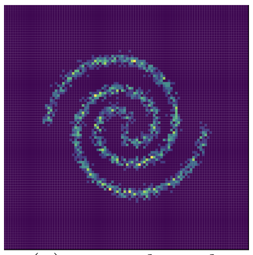
Table 4.3: Results on checkerboard

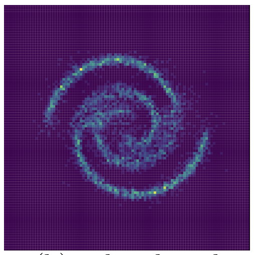
Layer Type	Scheme	Solver	NOP	NFE	NLL	Inverse Error
Spline-based	DTO	RK4	1327	40	**3.7074**	2.7579×10^{-5}
	OTD	dopri5	1327	320	**3.7234**	2.2769×10^{-6}
Concatsquash-linear	DTO	RK4	777	40	3.7429	3.1099×10^{-6}
	OTD	dopri5	777	110	3.7277	4.9029×10^{-6}

Table 4.4: Results on 2spirals

Layer Type	Scheme	Solver	NOP	NFE	NLL	Inverse Error
Spline-based	DTO	RK4	1327	40	**2.8074**	4.7927×10^{-5}
	OTD	dopri5	1327	212	**2.8620**	1.1200×10^{-5}
Concatsquash-linear	DTO	RK4	777	40	3.0648	1.7913×10^{-5}
	OTD	dopri5	777	116	2.9276	1.5375×10^{-5}

From the results shown in Table 4.3, Table 4.4, Table 4.5 as well as in Figure 4.2, Figure 4.2, Figure 4.5, we can observe smaller testing loss with spline-based model over all datasets while the spline-based layers can generate posterior distribution of better quality compare to the ground truth.


(a) ground truth


(b) spline-based

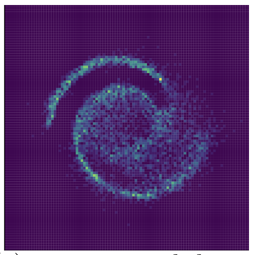
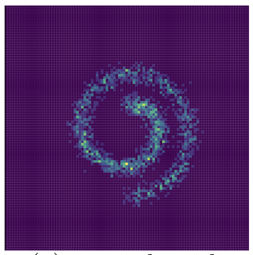
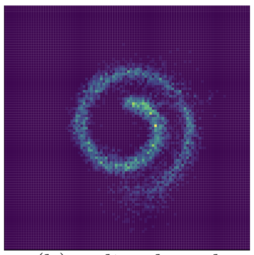

(c) concatsquash-linear

Figure 4.5: Results on 2spirals


(a) ground truth


(b) spline-based

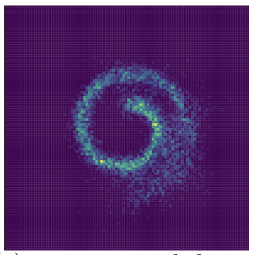

(c) concatsquash-linear

Figure 4.6: Results on swissroll

Table 4.5: Results on swissroll

Layer Type	Scheme	Solver	NOP	NFE	NLL	Inverse Error
Spline-based	DTO	RK4	1327	40	**2.7749**	1.1356×10^{-5}
	OTD	dopri5	1327	122	**2.6981**	1.8557×10^{-5}
Concatsquash-linear	DTO	RK4	777	40	2.8467	6.7051×10^{-6}
	OTD	dopri5	777	98	2.7322	1.1188×10^{-5}

By comparing the DTO and OTD schemes for each model, we find that spline-based model can achieve a lower NLL value under OTD discretization scheme for 8gaussian and swissroll datasets, while it have a lower NLL value under DTO discretization scheme for 2spirals and checkerboard datasets. However, the difference of the NLL values between DTO and OTD of each model is smaller than 0.1, which means they can have a similar performance on density estimation. Meanwhile, the NFE for OTD scheme is always larger than that of DTO, this means the computational cost of OTD scheme is larger than that of DTO. Therefore, DTO can be a perferable choice for discretization for spline-based CNF model since it can achive comparable performance while maintaining low computational cost.

Despite spline-based layers have a better performance than concatsquash-linear layers of same number of layers as well as the intermediate layers' dimensions, we should notice that the number of parameters of our spline-based model is always larger than concatsquash-linear model, and this might lead to a better performance. Therefore, we adjust the dimension for each layer of concatsquash-linear model so that the number of parameters can be close to spline-based model. Then we re-train the concatsquash-linear model and re-do the evaluation on test set. Table 4.6 displays the results of adjusted concatsquash-linear models and we can compare them with spline-based models of nearly equivalent number of parameters.

According to Table 4.6, we can observe a better performance of spline-based models with respect to NLL values on 2spirals, checkerboard, and swissroll datasets. For 8gaussians dataset, the model of concatsquash-linear layers behave better than spline-based layers under both DTO and OTD schemes. Therefore, it shows the spline-based model can achieve better accuracy than concatsquash-linear model when processing with complicate distributions, which indicates spline-based model may have a wider application than concatsquash-linear model in coping with realistic problems.

Table 4.6: Comparisons among all datasets

Data	Layer Type	Dims	Scheme	Solver	NOP	NFE	NLL	Inverse Error
2spirals	Spline	2-16-16-16-2	DTO	RK4	1327	40	**2.8074**	4.7927×10^{-5}
	Spline	2-16-16-16-2	OTD	dopri5	1327	212	**2.8620**	1.1200×10^{-5}
	CCS	2-22-22-22-2	DTO	RK4	1329	40	2.9965	1.8138×10^{-4}
	CCS	2-22-22-22-2	OTD	dopri5	1329	104	3.0918	1.8955×10^{-5}
8gaussians	Spline	2-16-16-16-2	DTO	RK4	1327	40	2.9259	1.4907×10^{-6}
	Spline	2-16-16-16-2	OTD	dopri5	1327	86	2.8975	5.2907×10^{-6}
	CCS	2-22-22-22-2	DTO	RK4	1329	40	**2.8547**	3.4301×10^{-6}
	CCS	2-22-22-22-2	OTD	dopri5	1329	74	**2.8472**	5.8758×10^{-6}
checkerboard	Spline	2-16-16-16-2	DTO	RK4	1327	40	**3.7074**	2.7579×10^{-5}
	Spline	2-16-16-16-2	OTD	dopri5	1327	320	**3.7234**	2.2769×10^{-6}
	CCS	2-22-22-22-2	DTO	RK4	1329	40	3.7077	3.5431×10^{-5}
	CCS	2-22-22-22-2	OTD	dopri5	1329	110	3.7289	6.1915×10^{-6}
swissroll	Spline	2-16-16-16-2	DTO	RK4	1327	40	**2.7749**	1.1356×10^{-5}
	Spline	2-16-16-16-2	OTD	dopri5	1327	122	**2.6981**	1.8557×10^{-5}
	CCS	2-22-22-22-2	DTO	RK4	1329	40	2.8013	1.2673×10^{-5}
	CCS	2-22-22-22-2	OTD	dopri5	1329	74	2.7646	1.0907×10^{-5}

4.3 Adjust Dimensions

From the results in previous sections, we observe that the negative log likelihood values decrease with the increment of dimensions. In order to investigate the influence of number of layers and hidden layers' dimensions on model performance, we execute experiments with spline-based and concatsquash-linear models of different layer type, layer dimension, as well as number of layers. Table 4.7 – Table 4.10 show the results of spline-based model; Table 4.11 – Table 4.14 show the results of concatsquash-linear model.

The results show that increasing the dimensions of hidden layers or the number of hidden layers can both improve the model performance with respect to NLL and inverse error values. One explanation to the results is that the model with hidden layers of lower dimensions or less hidden layer is inadequate for approximating complex target distribution. However, when we raise the number of hidden layers or dimensions of hidden layers, the number of parameters of the model also increases, thus increasing the complexity of the

Table 4.7: Spline-based model with different dimensions on 2spirals

Dims	Scheme	Solver	NOP	NFE	NLL	Inverse Error
2-16-2	DTO	RK4	335	40	3.2652	7.9447×10^{-6}
	OTD	dopri5	335	56	3.4427	1.6840×10^{-5}
2-16-16-16-2	DTO	RK4	1327	40	2.8074	4.7927×10^{-5}
	OTD	dopri5	1327	212	2.8620	1.1200×10^{-5}
2-64-2	DTO	RK4	1247	40	2.9750	6.1032×10^{-6}
	OTD	dopri5	1247	86	3.1016	2.0595×10^{-5}
2-64-64-64-2	DTO	RK4	11359	40	2.6835	2.5667×10^{-4}
	OTD	dopri5	11359	296	2.6841	4.7184×10^{-6}

Table 4.8: Spline-based model with different dimensions on 8gaussians

Dims	Scheme	Solver	NOP	NFE	NLL	Inverse Error
2-16-2	DTO	RK4	335	40	2.9057	4.5679×10^{-7}
	OTD	dopri5	335	62	2.8912	1.5983×10^{-5}
2-16-16-16-2	DTO	RK4	1327	40	2.9259	1.4907×10^{-6}
	OTD	dopri5	1327	86	2.8975	5.2907×10^{-6}
2-64-2	DTO	RK4	1247	40	2.8376	7.1755×10^{-8}
	OTD	dopri5	1247	74	2.8434	5.4040×10^{-6}
2-64-64-64-2	DTO	RK4	11359	40	2.8398	1.3483×10^{-6}
	OTD	dopri5	11359	104	2.8681	3.0110×10^{-6}

model. Adding more parameters into an insufficient model can improve the ability of dealing with complex distribution but it can also raise the computational cost. Therefore, there exists a trade-off between the number of layers, the dimension of each layer and the model accuracy.

Table 4.9: Spline-based model with different dimensions on checkboard

Dims	Scheme	Solver	NOP	NFE	NLL	Inverse Error
2-16-2	DTO	RK4	335	40	3.9290	4.3529×10^{-6}
	OTD	dopri5	335	56	3.9258	1.4311×10^{-5}
2-16-16-16-2	DTO	RK4	1327	40	3.7074	2.7579×10^{-5}
	OTD	dopri5	1327	320	3.7234	2.2769×10^{-6}
2-64-2	DTO	RK4	1247	40	3.8160	1.1955×10^{-6}
	OTD	dopri5	1247	68	3.7935	6.0533×10^{-6}
2-64-64-64-2	DTO	RK4	11359	40	3.5667	4.4777×10^{-6}
	OTD	dopri5	11359	152	3.6262	3.3561×10^{-6}

Table 4.10: Spline-based model with different dimensions on swissroll

Dims	Scheme	Solver	NOP	NFE	NLL	Inverse Error
2-16-2	DTO	RK4	335	40	2.9528	3.9884×10^{-6}
	OTD	dopri5	335	62	2.9837	8.6344×10^{-6}
2-16-16-16-2	DTO	RK4	1327	40	2.7749	1.1356×10^{-5}
	OTD	dopri5	1327	122	2.6981	1.8557×10^{-5}
2-64-2	DTO	RK4	1247	40	2.7410	2.3483×10^{-6}
	OTD	dopri5	1247	98	2.7195	7.4774×10^{-6}
2-64-64-64-2	DTO	RK4	11359	40	2.6787	1.8888×10^{-5}
	OTD	dopri5	11359	140	2.6980	4.5252×10^{-6}

Table 4.11: Concatsquash-linear model with different dimensions on 2spirals

Dims	Scheme	Solver	NOP	NFE	NLL	Inverse Error
2-16-2	DTO	RK4	137	40	3.5439	1.2409×10^{-6}
	OTD	dopri5	137	50	3.5123	5.3191×10^{-6}
2-16-16-16-2	DTO	RK4	777	40	3.0648	1.7913×10^{-5}
	OTD	dopri5	777	116	2.9276	1.5375×10^{-5}
2-64-2	DTO	RK4	521	40	3.1748	4.5803×10^{-6}
	OTD	dopri5	521	50	3.1936	9.5877×10^{-6}
2-64-64-64-2	DTO	RK4	9225	40	2.7220	2.8754×10^{-5}
	OTD	dopri5	9225	140	2.7471	8.6251×10^{-6}

Table 4.12: Concatsquash-linear model with different dimensions on 8gaussians

Dims	Scheme	Solver	NOP	NFE	NLL	Inverse Error
2-16-2	DTO	RK4	137	40	2.9604	1.4735×10^{-6}
	OTD	dopri5	137	50	2.8830	6.1683×10^{-6}
2-16-16-16-2	DTO	RK4	777	40	2.8632	2.8324×10^{-6}
	OTD	dopri5	777	116	2.8943	8.2833×10^{-6}
2-64-2	DTO	RK4	521	40	2.8821	1.1011×10^{-6}
	OTD	dopri5	521	50	2.8848	3.5526×10^{-6}
2-64-64-64-2	DTO	RK4	9225	40	2.8419	1.9244×10^{-6}
	OTD	dopri5	9225	140	2.8295	4.4106×10^{-6}

Table 4.13: Concatsquash-linear model with different dimensions on checkboard

Dims	Scheme	Solver	NOP	NFE	NLL	Inverse Error
2-16-2	DTO	RK4	137	40	3.9380	1.4143×10^{-6}
	OTD	dopri5	137	56	3.8926	4.7784×10^{-6}
2-16-16-16-2	DTO	RK4	777	40	3.7429	3.1099×10^{-6}
	OTD	dopri5	777	110	3.7277	4.9029×10^{-6}
2-64-2	DTO	RK4	521	40	3.8538	9.1984×10^{-7}
	OTD	dopri5	521	62	3.8324	5.8539×10^{-6}
2-64-64-64-2	DTO	RK4	9225	40	3.6018	4.8289×10^{-6}
	OTD	dopri5	9225	104	3.5967	2.7021×10^{-6}

Table 4.14: Concatsquash-linear model with different dimensions on swissroll

Dims	Scheme	Solver	NOP	NFE	NLL	Inverse Error
2-16-2	DTO	RK4	137	40	3.0006	3.4805×10^{-6}
	OTD	dopri5	137	44	3.0446	6.5326×10^{-6}
2-16-16-16-2	DTO	RK4	777	40	2.8467	6.7051×10^{-6}
	OTD	dopri5	777	98	2.7322	1.1188×10^{-5}
2-64-2	DTO	RK4	521	40	2.7454	2.6538×10^{-6}
	OTD	dopri5	521	50	2.7865	5.0067×10^{-6}
2-64-64-64-2	DTO	RK4	9225	40	2.7058	5.8436×10^{-6}
	OTD	dopri5	9225	74	2.7104	3.8043×10^{-6}

Chapter 5

Discussion

In this chapter, we provide conclusions with respect to comparisons among numerical experiments results. Besides, we discuss some future directions about spline parameterization for continuous normalizing flows model.

5.1 Conclusions

We compare the Discretize-Then-Optimize (DTO) and Optimize-Then-Discretize (OTD) methods for spline-based Continuous Normalizing Flows and concatsquash-linear Continuous Normalizing Flows using Neural ODEs. The DTO method can achieve comparable performance on density estimation as OTD method according to negative log likelihood and inverse error, while DTO method have much lower computational cost. Therefore, DTO method is the preferable choice for both spline-based Continuous Normalizing Flows and concatsquash-linear Continuous Normalizing Flows.

Besides, we notice there is a trade-off between the number of parameters and the accuracy of the CNF model. We can raise the number of parameters by increasing the number of hidden layers or increasing the dimension of each hidden layer. Both can contribute to improving model performance on density estimation, but raising the computational cost at the same time.

Last but not least, we find that the spline-based CNF have a better density estimation for complex target distributions than concatsquash-linear CNF. Each of the models produces a pretty small NLL as well as a small inverse error, which means the model is of high accuracy on density estimation and is quite invertible. With an approximately equal number of parameters, spline-based CNF can generate more accurate posterior density estimations on 2spirals, checkerboard, swissroll datasets than concatsquash-linear CNF, it indicates the spline-based CNF has the ability of dealing with more general cases, thus making it a more general choice as we want to approximate complex unknown distributions in reality.

5.2 Future Directions

In this research paper, we conduct several numerical experiments on 2D toy datasets. Even though we have tested on checkerboard, swissroll, 2spirals that can be more complex for density estimation than 8gaussians dataset, it is necessary to carry out experiments on real data or high-dimensional data, such as CIFAR 10, MNIST, and etc. Training with real data can be time-consuming, especially under OTD method which needs more computations for each iteration, we plan to investigate this for the next step.

Furthermore, we can adjust hyperparameters in the spline-based CNF network and try to find an optimal set of hyperparameters that can achieve good accuracy, invertibility while maintaining a low cost. This process is called hyperparameter tuning. However, it needs to do a lot of work since we can have a large number of scenarios with different combinations of hyperparameters. In our spline-based CNF model, we additionally have hyperparameters including the degree of B-spline functions d, number of knots invervals L, other than concatsquash-linear CNF model. It is important to find an optimal set of hyperparameters that can raise our model's performance. However, increasing the number of knots and degree of spline functions can raise the complexity of model, thus making it time-consuming. We plan to investigate how to find a proper set of hyperparameters in the future. Some of the hyperparameter tuning results are displayed in Table A.1 – Table A.3 in Appendix.

References

[Arnol'd, 2013] Arnol'd, V. I. (2013). *Mathematical methods of classical mechanics*, volume 60. Springer Science & Business Media.

[Benning et al., 2019] Benning, M., Celledoni, E., Ehrhardt, M. J., Owren, B., and Schönlieb, C.-B. (2019). Deep learning as optimal control problems: Models and numerical methods. *arXiv preprint arXiv:1904.05657*.

[Chen et al., 2018] Chen, R. T., Rubanova, Y., Bettencourt, J., and Duvenaud, D. (2018). Neural ordinary differential equations. *arXiv preprint arXiv:1806.07366*.

[Cottrell et al., 2009] Cottrell, J. A., Hughes, T. J., and Bazilevs, Y. (2009). *Isogeometric analysis: toward integration of CAD and FEA*. John Wiley & Sons.

[Dormand and Prince, 1980] 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.

[Gholami et al., 2019] Gholami, A., Keutzer, K., and Biros, G. (2019). Anode: Unconditionally accurate memory-efficient gradients for neural odes. *arXiv preprint arXiv:1902.10298*.

[Goodfellow et al., 2014] Goodfellow, I., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A., and Bengio, Y. (2014). Generative adversarial nets. *Advances in neural information processing systems*, 27.

[Grathwohl et al., 2018] Grathwohl, W., Chen, R. T., Bettencourt, J., Sutskever, I., and Duvenaud, D. (2018). Ffjord: Free-form continuous dynamics for scalable reversible generative models. *arXiv preprint arXiv:1810.01367*.

[Günther et al., 2021] Günther, S., Pazner, W., and Qi, D. (2021). Spline parameterization of neural network controls for deep learning. *arXiv preprint arXiv:2103.00301*.

[Gunzburger, 2002] Gunzburger, M. D. (2002). *Perspectives in flow control and optimization*. SIAM.

[Jordan et al., 1999] Jordan, M. I., Ghahramani, Z., Jaakkola, T. S., and Saul, L. K. (1999). An introduction to variational methods for graphical models. *Machine learning*, 37(2):183–233.

[Kincaid et al., 2009] Kincaid, D., Kincaid, D. R., and Cheney, E. W. (2009). *Numerical analysis: mathematics of scientific computing*, volume 2. American Mathematical Soc.

[Kingma and Welling, 2013] Kingma, D. P. and Welling, M. (2013). Auto-encoding variational bayes. *arXiv preprint arXiv:1312.6114*.

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

[Kullback, 1997] Kullback, S. (1997). *Information theory and statistics*. Courier Corporation.

[Kullback and Leibler, 1951] Kullback, S. and Leibler, R. A. (1951). On information and sufficiency. *The annals of mathematical statistics*, 22(1):79–86.

[Onken et al., 2020] Onken, D., Fung, S. W., Li, X., and Ruthotto, L. (2020). Ot-flow: Fast and accurate continuous normalizing flows via optimal transport. *arXiv preprint arXiv:2006.00104*.

[Onken and Ruthotto, 2020] Onken, D. and Ruthotto, L. (2020). Discretize-optimize vs. optimize-discretize for time-series regression and continuous normalizing flows. *arXiv preprint arXiv:2005.13420*.

[Papamakarios et al., 2019] Papamakarios, G., Nalisnick, E., Rezende, D. J., Mohamed, S., and Lakshminarayanan, B. (2019). Normalizing flows for probabilistic modeling and inference. *arXiv preprint arXiv:1912.02762*.

[Rezende and Mohamed, 2015] Rezende, D. and Mohamed, S. (2015). Variational inference with normalizing flows. In *International conference on machine learning*, pages 1530–1538. PMLR.

[Rodríguez and Lopez Fernandez, 2010] Rodríguez, O. H. and Lopez Fernandez, J. M. (2010). A semiotic reflection on the didactics of the chain rule. *The Mathematics Enthusiast*, 7(2):321–332.

[Shampine, 1986] Shampine, L. F. (1986). Some practical runge-kutta formulas. *Mathematics of computation*, 46(173):135–150.

[Wainwright and Jordan, 2008a] Wainwright, M. J. and Jordan, M. I. (2008a). *Graphical models, exponential families, and variational inference*. Now Publishers Inc.

[Wainwright and Jordan, 2008b] Wainwright, M. J. and Jordan, M. I. (2008b). Introduction to variational methods for graphical models. *Foundations and Trends in Machine Learning*, 1:1–103.

APPENDICES

Appendix A

Tables of Numerical Experiments

A.1 Adjust degree

For spline-based model, we can adjust the degree d and number of knots intervals L of B-spline functions. These can be viewed as model's hyperparameters. In this section, we show the adjustment on degree of B-spline basis functions and compare the model accuracy as well as invertibility among different datasets.

Table A.1: Spline-based model with different degrees on 2spirals

Degree	Scheme	Solver	NOP	NFE	NLL	Inverse Error
d=0	DTO	RK4	263	40	3.2535	3.5810×10^{-3}
	OTD	dopri5	263	452	3.1716	1.7021×10^{-4}
d=1	DTO	RK4	299	40	3.2055	1.2406×10^{-5}
	OTD	dopri5	299	128	3.2193	4.7123×10^{-5}
d=2	DTO	RK4	335	40	3.2652	7.9447×10^{-6}
	OTD	dopri5	335	56	3.4427	1.6840×10^{-5}

However, according to the results shown in Table A.1 – Table A.3, we can not find a consistent optimal set of hyperparameters among different datasets. For different datasets, the optimal choice of degree is different. Therefore, for further study, we should try more different combinations of hyperparameters to make the hyperparameter optimization.

Table A.2: Spline-based model with different degrees on checkboard

Degree	Scheme	Solver	NOP	NFE	NLL	Inverse Error
d=0	DTO	RK4	263	40	3.8879	2.9088×10^{-4}
	OTD	dopri5	263	446	3.8529	1.4228×10^{-4}
d=1	DTO	RK4	299	40	3.8399	1.4713×10^{-6}
	OTD	dopri5	299	86	3.8944	4.8372×10^{-5}
d=2	DTO	RK4	335	40	3.9290	4.3530×10^{-6}
	OTD	dopri5	335	56	3.9258	1.4311×10^{-5}

Table A.3: Spline-based model with different degrees on swissroll

Degree	Scheme	Solver	NOP	NFE	NLL	Inverse Error
d=0	DTO	RK4	263	40	2.8206	8.1672×10^{-4}
	OTD	dopri5	263	380	2.8302	1.1326×10^{-4}
d=1	DTO	RK4	299	40	2.8983	4.3935×10^{-6}
	OTD	dopri5	299	116	2.8881	2.8425×10^{-5}
d=2	DTO	RK4	335	40	2.9528	3.9884×10^{-6}
	OTD	dopri5	335	62	2.9837	8.6344×10^{-6}