Accelerating Continuous Normalizing Flow with Trajectory Polynomial Regularization

Han-Hsien Huang [1,2], **Mi-Yen Yeh** [1]
[1] Institute of Information Science, Academia Sinica
[2] Department of Computer Science and Engineering, Texas A&M University
hanhsien.huang@tamu.edu, miyen@iis.sinica.edu.tw

Abstract

In this paper, we propose an approach to effectively accelerating the computation of continuous normalizing flow (CNF), which has been proven to be a powerful tool for the tasks such as variational inference and density estimation. The training time cost of CNF can be extremely high because the required number of function evaluations (NFE) for solving corresponding ordinary differential equations (ODE) is very large. We think that the high NFE results from large truncation errors of solving ODEs. To address the problem, we propose to add a regularization. The regularization penalizes the difference between the trajectory of the ODE and its fitted polynomial regression. The trajectory of ODE will approximate a polynomial function, and thus the truncation error will be smaller. Furthermore, we provide two proofs and claim that the additional regularization does not harm training quality. Experimental results show that our proposed method can result in 42.3% to 71.3% reduction of NFE on the task of density estimation, and 19.3% to 32.1% reduction of NFE on variational auto-encoder, while the testing losses are not affected at all.

1 Introduction

Normalizing flows (Rezende and Mohamed 2015) are a kind of invertible neural networks that have an efficient calculation of Jacobian determinant. They can be used as generative models, density estimation or posterior distribution of variational auto-encoders (VAE). However, their two requirements, invertibility and easy Jacobian determinant calculation, put a great restriction on the design of corresponding neural network architecture.

Recently, continuous normalizing flow (CNF) (Chen et al. 2018; Grathwohl et al. 2019) is proposed that can avoid such design restriction. CNF describes the transformation of hidden states with ordinary differential equations (ODE), instead of layer-wise function mappings. In this way, the invertibility becomes trivial and the determinant of Jacobian becomes simply a trace. With the freedom of network architecture design, CNF has been shown to outperform normalizing flow with discrete layers in terms of lower testing loss.

However, running a CNF model, which is equal to solving a series of ODEs, may take a lot of time. A single training iteration can need up to hundreds of function evaluations, which is equivalent to running a neural network with hundreds of layers. Moreover, the number of function evaluations required per training iteration can gradually grow up throughout the training process. The reason is that CNF uses ODE solvers with adaptive steps. The step sizes of solving ODEs are determined based on the truncation errors. Such error usually grows as the training goes and thus the number of steps becomes larger.

To make training of CNF faster, we propose *Trajectory Polynomial Regularization* (TPR). TPR is an additional loss function which regularizes the trajectories of the ODE solutions. It penalizes the difference between the trajectory of solution and its fitted polynomial regression. Therefore, TPR enforces the solutions to approximate polynomial curves, which can reduce the truncation error of solving the ODEs of CNF. Despite adding the regularization, we argue that the method does not harm the testing loss much. We prove that the optimal solutions of the modified loss are still optimal solutions for the original task. The detail of the proofs and argument are in Section 3. In experiments, we find that for both density estimation and variational inference tasks, our model can save as much as 71% of *number of function evaluations* (NFE), which is the number of evaluating the ODEs. Furthermore, our method barely affect the testing errors. Our code is published in our Github repository[1].

The remainder of the paper is organized as follows. We first introduce the background and related work of CNF in Section 2. In Section 3, we describe the intuition and mathematics of TPR and provide two proofs to argue the power of its quality. In Section 4, we conduct experiments on two tasks, density estimation and VAE. We visualize the effect of our model with simple 2D toy data. And then we evaluate the efficiency and quality of our model on real data.

2 Preliminaries

2.1 Background

Normalizing flows (Rezende and Mohamed 2015) are models that can be used to represent a wide range of distributions. They consist of an invertible function. Let $f : \mathbb{R}^d \to \mathbb{R}^d$ be an invertible function, i.e., $\mathbf{z} = f(\mathbf{x})$ and $\mathbf{x} = f^{-1}(\mathbf{z})$.

[1]https://github.com/hanhsienhuang/CNF-TPR

Due to the invertibility, the log probabilities of the input $\mathbf{x}$ and the output $\mathbf{z} = f(\mathbf{x})$ have the following relation.

$$\log p(\mathbf{z}) = \log p(\mathbf{x}) - \log \left| \det \frac{\partial f(\mathbf{x})}{\partial \mathbf{x}} \right|. \tag{1}$$

Once $p(\mathbf{x})$ is set to be a known distribution, the distribution $p(\mathbf{z})$ is represented by $p(\mathbf{x})$ and the function f. The function f can be parametrized by deep neural networks. However, since f should be invertible and the Jacobian $\det \frac{\partial f}{\partial \mathbf{x}}$ in equation (1) should be easily calculated, the possible architecture of f is highly restricted. As a result, the capacity of representing $p(\mathbf{x})$ is also limited.

Different from normalizing flows, continuous normalizing flow (CNF) (Chen et al. 2018) makes the transformation from $\mathbf{x}$ to $\mathbf{z}$ a continuous evolution. That is, instead of a direct function mapping from $\mathbf{x}$ to $\mathbf{z}$, i.e. $\mathbf{z} = f(\mathbf{x})$, the transformation can be represented by a continuous function $\mathbf{y}(t)$, with $\mathbf{x} = \mathbf{y}(t_0)$ and $\mathbf{z} = \mathbf{y}(t_1)$. The evolution of $\mathbf{y}(t)$ in CNF is defined by an ordinary differential equation (ODE),

$$\frac{d\mathbf{y}}{dt} = \mathbf{v}(\mathbf{y}, t). \tag{2}$$

With this continuous transformation, the invertibility of $\mathbf{x}$ and $\mathbf{z}$ is inherently satisfied. Therefore, there isn't any other restriction on the form of function $\mathbf{v}(\mathbf{y}, t)$.

The dynamics of the log likelihood $\log p(\mathbf{y}(t))$ is derived in the instantaneous change of variables theorem by Chen et al. (2018), which is

$$\frac{d}{dt} \log p = -\operatorname{Tr}\left(\frac{\partial \mathbf{v}}{\partial \mathbf{y}}\right). \tag{3}$$

The computation complexity for obtaining the exact value of $\operatorname{Tr}\left(\frac{\partial \mathbf{v}}{\partial \mathbf{y}}\right)$ is $\mathcal{O}(D^2)$, where D is the dimension of data $\mathbf{x}$. It can be reduced to $\mathcal{O}(D)$ (Grathwohl et al. 2019) with the Hutchinson's trace estimator (Hutchinson 1989).

$$\operatorname{Tr}\left(\frac{\partial \mathbf{v}}{\partial \mathbf{y}}\right) = \mathop{\mathbb{E}}_{p(\boldsymbol{\epsilon})} \left[\boldsymbol{\epsilon}^T \frac{\partial \mathbf{v}}{\partial \mathbf{y}} \boldsymbol{\epsilon} \right], \tag{4}$$

where $\mathbb{E}[\boldsymbol{\epsilon}] = 0$ and $\operatorname{Cov}(\boldsymbol{\epsilon}) = I$. With this method, computing the estimation of $\operatorname{Tr}\left(\frac{\partial \mathbf{v}}{\partial \mathbf{y}}\right)$ costs exactly the same as computing $\mathbf{v}$.

Given the initial values of $\mathbf{y}(t_0)$ and $\log p(t_0)$, CNF outputs the final values $\mathbf{y}(t_1)$ and $\log p(t_1)$, which can be written as

$$\mathbf{y}(t_1) = \mathbf{y}(t_0) + \int_{t_0}^{t_1} \mathbf{v}\, dt\,, \text{and} \tag{5}$$

$$\log p(\mathbf{y}(t_1)) = \log p(\mathbf{y}(t_0)) - \int_{t_0}^{t_1} \operatorname{Tr}\left(\frac{\partial \mathbf{v}}{\partial \mathbf{y}}\right) dt\,. \tag{6}$$

These equations are solved by numerical ODE solvers. Usually, adaptive stepsize solvers, such as Dopri5 (Dormand and Prince 1980), are used because they guarantee a desired precision of output. Adaptive solvers dynamically determine the step sizes of solving ODE by estimating the magnitude of truncation errors. They shrink the step sizes when the truncation errors exceed a tolerable value and increase the step sizes when the errors are too small.

Unlike the restriction of f's architecture in normalizing flows, the $\mathbf{v}(\mathbf{y}, t)$ in CNF can be represented by arbitrary neural networks. As a result, CNF can have greater capacity to represent the distributions $p(\mathbf{z})$ than other normalizing flow models.

2.2 Related work

The design of normalizing flows is restricted by the requirements of invertibility and efficient Jacobian determinant computation. Rezende and Mohamed (2015) proposed planar flow and radial flow for variational inference. The layers in the proposed model have to be in a simple form. Real NVP (Dinh, Sohl-Dickstein, and Bengio 2017) is a structure of network such that its Jacobian matrix are triangular, so the determinant are simply the product of diagonal values. Kingma and Dhariwal (2018) proposed Glow, which has 1×1 convolution layers that pass partitioned dimensions into invertible affine transformations. Auto-regressive flow models (Kingma et al. 2016; Papamakarios, Pavlakou, and Murray 2017; Oliva et al. 2018; Huang et al. 2018) also have triangular Jacobian matrix in each layer. Although they have high expressiveness, they require D sequential computations in each layer, which can be costly when D is large. Huang, Dinh, and Courville (2020) proposed Augmented Normalizing Flow (ANF), which augments the data by adding more dimensions in transformation. Although reaching high performance, it requires large number of samples to compute accurate importance sampling.

Continuous normalizing flow (CNF) (Chen et al. 2018) is a continuous version of normalizing flows, which replaces the layer-wise transformations of normalizing flows with ODEs. The model does not restrict the structure of its transformation function, but computing Jacobian determinant costs computing gradient D times. Grathwohl et al. (2019) improved the efficiency of computing Jacobian determinant by using Hutchinson's trace estimator. Chen and Duvenaud (2019) proposed a new architecture of neural network which can efficiently compute trace of Jacobian.

As we mentioned in Section 2.1, training CNF takes a lot of time because solving ODEs can be very slow. To enhance the training stability and reduce the training time, some researchers have proposed to adopt regularization methods. Grathwohl et al. (2019) proposed to use weight decay and spectral normalization (Miyato et al. 2018). Salman et al. (2018) proposed geodesic regularization and inverse consistency regularization. Finlay et al. (2020) and Onken et al. (2020) proposed to regularize the dynamics of CNF with the notion of optimal transport. Kelly et al. (2020) proposed to regularize the higher-order derivatives of the velocity field and applied Taylor-mode automatic differentiation to compute the derivatives efficiently. Our work is different from these works in two ways. First, we provide proofs for the existence of infinite optimal solutions with our regularization. Second, our proposed TPR only requires a few samples point in time. On the contrary, those methods need to solve more than one additional ODEs, which increases computational consumption.

3 Method

The inefficiency of CNF comes from the large number of steps of solving ODEs. With adaptive ODE solvers, the number of steps is dynamic. The solvers decrease the step sizes when the truncation errors of solving OEDs are too large, and vice versa. Therefore, to reduce the number of steps, we propose a method to reduce the truncation errors when solving ODEs. To do so, we focus on reducing the truncation error of solving $\mathbf{y}(t)$ but not $\log p$. Although we actually have two ODEs to solve (equations of (2) and (3)), both their right-hand-sides depend only on $\mathbf{y}$.

Theoretically, for a k-th order ODE solver to solve the equation $\frac{d\mathbf{y}}{dt} = \mathbf{v}(\mathbf{y}, t)$, the local truncation error of $\mathbf{y}$ is $\mathcal{O}\left(\frac{d^{k+1}\mathbf{y}}{dt^{k+1}} \Delta t^{k+1}\right)$, where Δt is the size of step. To increase the step size Δt while maintaining the magnitude of truncation error, $\left\| \frac{d^{k+1}\mathbf{y}}{dt^{k+1}} \right\|$ should be as small as possible. Obviously, the optimal situation is that $\mathbf{y}(t)$ is a polynomial function of degree $d \leq k$, because $\frac{d^{k+1}\mathbf{y}}{dt^{k+1}} = 0$.

To force the trajectory $\mathbf{y}(t)$ to approximate a polynomial function, we add a loss term L_p called *Trajectory Polynomial Regularization* (TPR) into the total loss.

$$L = L_0 + \alpha L_p, \tag{7}$$

where L_0 is the original loss function of the corresponding task and α is a constant coefficient. L_p is derived as follows. First, a number of time steps $\{\tau_0, \ldots, \tau_{n-1}\}$ are randomly sampled. Next, a polynomial function $\mathbf{f}(t)$ is fitted to the points $\{\mathbf{y}(\tau_0), \ldots, \mathbf{y}(\tau_{n-1})\}$. And finally, L_p is calculated as the mean squared error (MSE) between $\mathbf{f}(\tau_i)$ and $\mathbf{y}(\tau_i)$. When minimizing L_p, $\mathbf{y}(t)$ will approach the polynomial function, $\mathbf{f}(t)$.

Figure 1 illustrates the mechanism and effect of our method. Figure 1(a) demonstrates how TPR regularizes $\mathbf{y}(t)$ to approximate a polynomial function. The loss L_p is defined as the squared difference between the two curves. So it act as a force to pull the four points from the trajectory to the fitted polynomial regression. Figure 1(b) shows a possible result without TPR. The trajectories of the solutions $\mathbf{y}(t)$ can be very winding, so solving $\mathbf{y}(t)$ can be more difficult. Because the truncation errors are higher when solving $\mathbf{y}(t)$, ODE solvers then have to shrink step sizes in order to guarantee the precision of the solution, which leads to more computation in terms of higher number of function evaluations. On the other hand, Figure 1(c) shows the result using TPR with degree 1. The trajectories of $\mathbf{y}(t)$ are more like a straight line. As a result, $\mathbf{y}(t)$ is easier to solve, and thus the required number of function evaluations decreases.

To derive L_p, the math expression of L_p can be written as

$$L_p = \frac{1}{n} \sum_{i=0}^{n-1} \|\mathbf{y}(\tau_i) - \mathbf{f}(\tau_i)\|^2 = \frac{1}{n} \|Y - TC\|^2, \tag{8}$$

where $\|\cdot\|$ is Frobenius norm. $\mathbf{f}(t) = \mathbf{c}_0 P_0(t) + \cdots + \mathbf{c}_d P_d(t)$ is the polynomial function we want to fit. d is the degree of the polynomial function, $\{\mathbf{c}_0, \ldots, \mathbf{c}_d\}$ is the coefficients to be fitted and $\{P_0(t), \ldots, P_d(t)\}$ is a polynomial basis. The matrices Y, T and C are written as follows,

$$Y = \begin{pmatrix} \mathbf{y}(\tau_0) & \cdots & \mathbf{y}(\tau_{n-1}) \end{pmatrix}^\top, \tag{9}$$

$$T = \begin{pmatrix} P_0(\tau_0) & \cdots & P_d(\tau_0) \\ \vdots & \ddots & \vdots \\ P_0(\tau_{n-1}) & \cdots & P_d(\tau_{n-1}) \end{pmatrix}, \tag{10}$$

$$C = \begin{pmatrix} \mathbf{c}_0 & \cdots & \mathbf{c}_d \end{pmatrix}^\top. \tag{11}$$

Solving C to minimize $\|Y - TC\|^2$ yields to the equation, $C = (T^\top T)^{-1} T^\top Y$. After substituting it back to equation (8), we can get

$$L_p = \frac{1}{n} \left\| \left(I - T(T^\top T)^{-1} T^\top\right) Y \right\|^2. \tag{12}$$

To let the matrix computation more numerically stable, we use singular value decomposition (SVD) on T, which gives us $T = U \Sigma V^\top$. And thus, the final form of calculating the TPR loss is written as follows,

$$L_p = \frac{1}{n} \left\| \left(I - U_{1:d+1} (U_{1:d+1})^\top\right) Y \right\|^2, \tag{13}$$

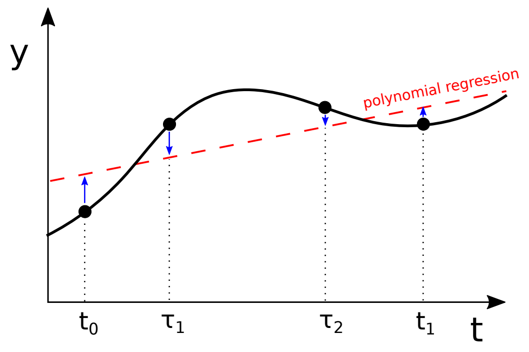
where $U_{1:d+1}$ denotes the leftmost $d+1$ columns of matrix U.

Although we introduce a new loss term into the total loss, we prove that there are always optimal solutions minimizing both L and L_0. In other words, when minimizing the total loss L to find an optimal solution, the solution is also an optimal solution of L_0. In the following two theorems, we prove that, even with the hard constraint $\frac{d^2\mathbf{y}}{dt^2} = \frac{d\mathbf{v}}{dt} = 0$ (so $L_p = 0$), there still exist infinitely many functions $\mathbf{v}(\mathbf{y}, t)$ that can transform any distribution to any other distribution. Because L_0 is a function of $\log p(\mathbf{y}(t_1))$, from the theorems, there are infinitely many $\mathbf{v}(\mathbf{y}, t)$ that can simultaneously minimize L_0 and L_p. Therefore, these $\mathbf{v}$ can minimize L and L_0 simultaneously. We present the two theorems as follows.

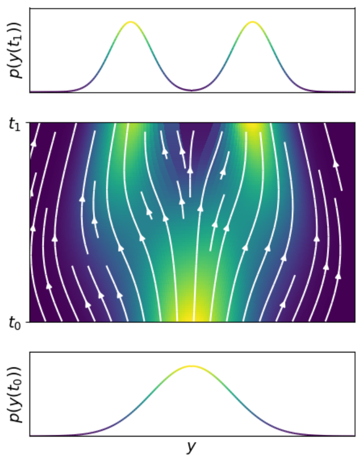
Theorem 1 *Assume that $\mathbf{y}(t) \in \mathbb{R}^D$ and $\log p(\mathbf{y}(t))$ are governed by the differential equations* (2) *and* (3)*, respectively. Given any distributions $p_0(\mathbf{x})$ and $p_1(\mathbf{x})$, where $p_0(\mathbf{x})$ and $p_1(\mathbf{x}) > 0$ for all $\mathbf{x} \in \mathbb{R}^D$, there exists a vector field $\mathbf{v}(\mathbf{y}, t)$ with the constraint $\frac{d\mathbf{v}}{dt} = 0$ everywhere, such that if the initial value of $\log p(\mathbf{y}(t))$ is $\log p_0(\mathbf{y}(t_0))$, then its final value is $\log p_1(\mathbf{y}(t_1))$.*

Theorem 2 *For $D > 1$, there are infinitely many such vector fields $\mathbf{v}(\mathbf{y}, t)$.*

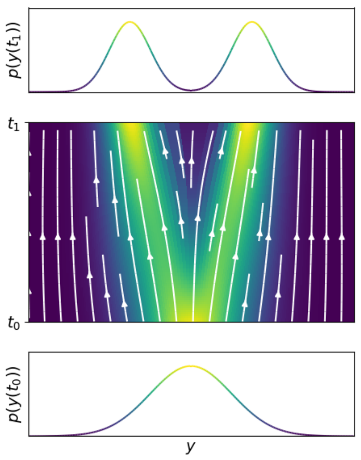
The next question is whether we can approach the optimal solutions with a function approximator, which is usually neural networks. We can't prove or guarantee anything about that, but we provide two arguments. First, the number of optimal solutions is infinite. Compared to finite number, the neural networks are more likely to approach one of the infinitely many optimal solutions. Second, in experiments, we will show that our method doesn't affect testing loss much. The model with TPR has approximately the same testing loss as the model without TPR.



(a) Illustration of the mechanism of TPR



(b) Without TPR.



(c) With TPR.

Figure 1: Illustration of our Trajectory Polynomial Regularization (TPR). (a): The illustrated mechanism of TPR. The solid curve represents trajectory of $\mathbf{y}(t)$. The dashed line is a fitted polynomial regression on the four randomly sampled points. Polynomial regression of degree 1 is demonstrated here. The TPR loss L_p is designed to pull the points of $\mathbf{y}$ to the fitted polynomial. (b) and (c): Two transformations of 1D distributions by CNF either with or without our TPR. The input is a Gaussian distribution and the output is a mixture of two Gaussian distributions. The white streamlines represent the trajectories of $\mathbf{y}(t)$ and the color represents the density $p(\mathbf{y}(t))$.

The details of the proofs are in Appendix A (see supplementary files). The basic idea to prove Theorem 1 is to actually construct a vector field $\mathbf{v}(\mathbf{y}, t)$ and the solution $\mathbf{y}(t)$ that satisfy all constraints. The two main constraints in the theorem should be carefully considered. First, the solution should satisfy the initial and final values. Second, $\mathbf{v}(\mathbf{y}, t)$ must be a well-defined function at any points $\mathbf{y}$ and t. For Theorem 2, the idea is to find an infinitesimal variation of $\mathbf{v}$ such that all the conditions are still satisfied. It should be careful that the value of the variation should be bounded in all space.

4 Experiments

We evaluate our proposed trajectory polynomial regularization (TPR) on two tasks, density estimation and variational autoencoder (VAE) (Kingma and Welling 2018). We compare our results with FFJORD (Grathwohl et al. 2019), the current state of the art of CNF, and the corresponding state-of-the-art normalizing flow models.

Two metrics are evaluated, testing loss and number of function evaluations (NFE). We want to see whether our model leads to lower computational cost compared to FFJORD in training, while keeping comparable training quality. To compare the computational cost, we count the average NFE per training iteration. NFE is defined as the number of evaluating the right-hand-side of the ODEs (2) and (3) when solving them. The lower NFE, the less computational cost.

In all experiments, We use exactly the same architectures and hyper-parameters of neural network as in FFJORD. Also the same as FFJORD, the package of ODE solvers Torchdiffeq[2] (Chen et al. 2018) is used. The only thing we do is adding our TPR loss to training loss. The hyper-parameters of TPR are described below. We choose the number of sampled point as $n = 4$, with $\tau_0 = t_0$ and $\tau_{n-1} = t_1$ as the start and end of time steps. The other τs are randomly sampled from a uniform distribution. The degree of the polynomial regression is set to be $d = 1$. We adopted the same ODE solver as FFJORD (Grathwohl et al. 2019), which is Dopri5 (Dormand and Prince 1980). Dopri5 is a 4th order ODE solver, so we can use any degree with $d \leq 4$. In this work, we only choose $n = 1$ in experiments to prove our concept. Despite the seemingly small degree, we have showed in Section 3 that it is powerful enough because the optimal solutions of L_0 still exist. The coefficient of the polynomial regularization is $\alpha = 5$. The tolerance coefficient of Dopri5 is set to be $atol = rtol = 10^{-4}$ in training, and $atol = rtol = 10^{-5}$ in testing to ensure precision of solving ODEs. These hyper-parameters above are shared in all experiments.

4.1 Density estimation

Given a data distribution $p(\mathbf{x})$, the task of density estimation aims at approximating this distribution. Let $q_\theta(\mathbf{x})$ be the approximated distribution. The loss of this task is the negative log-likelihood shown below.

$$L = - \mathop{\mathbb{E}}_{p(\mathbf{x})} \log q_\theta(\mathbf{x}). \tag{14}$$

We experiment density estimation on three 2D toy data and real data sets, which are five tabular data sets from (Papamakarios, Pavlakou, and Murray 2017) and two image data sets, MNIST and CIFAR10.

[2]https://github.com/rtqichen/torchdiffeq

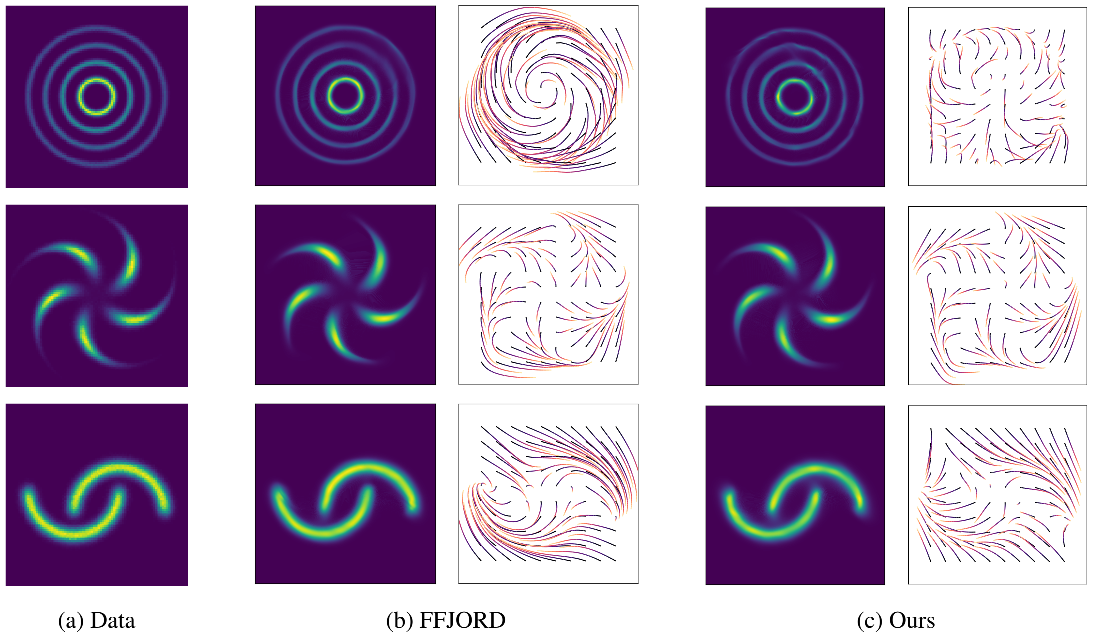


Figure 2: (a): Distribution of three 2D data sets. (b) and (c): The reconstructed distributions and the trajectories of $\mathbf{y}(t)$ by FFJORD and our model. To generate the trajectories, we select some grid points to be the initial values $\mathbf{y}(t_0)$ and solve the ODE in Equation (2) to obtain $\mathbf{y}(t)$. Each curve in the figures represents a single trajectory of $\mathbf{y}(t)$, and the color represents the value of time t.

Results on 2D toy data In this experiment, we test with three simple 2D toy data. The distributions of the three 2D data used for the experiment are shown in Figure 2(a). The approximated distributions using FFJORD and our model are in the upper halves of Figures 2(b) and 2(c), respectively. Both FFJORD and our model successfully recover the data distributions in decent quality, which implies that our new regularization loss doesn't harm approximation capability, and the model is able to find a good solution.

We visualize the trajectories of $\mathbf{y}(t)$ in the lower halves of Figures 2(b) and 2(c). We want to know whether our model actually performs how we expected as in Figure 1(c), which is that the trajectories of $\mathbf{y}(t)$ are approximately straight lines. It can be seen that the trajectories of FFJORD are very winding, while our trajectories are less winding and most are straight. Since we are using polynomial regularization of degree 1, the result is what we expected. Because of this, our ODEs are simpler to solve and the required step sizes for solving ODEs can be larger, so the number of steps can be smaller in training. In reality, the average NFEs of FFJORD on the three data are 90.42, 60.48 and 86.7 respectly, while the NFE of our model are just 44.17, 47.39 and 33.66. That is, our TPR leads to respectively 51.15%, 21.64% and 60.90% drop of computational cost.

Results on real data The real data we used here are tabular data, which are five data sets processed in (Papamakarios, Pavlakou, and Murray 2017), and the image data including the MNIST and CIFAR10 data sets. Besides FFJORD, we compare our model with two discrete normalizing flows: Real NVP (Dinh, Sohl-Dickstein, and Bengio 2017) and Glow (Kingma and Dhariwal 2018), and four autoregressive flows: MADE (Kingma et al. 2016), MAF (Papamakarios, Pavlakou, and Murray 2017), TAN (Oliva et al. 2018) , and MAF-DDSF (Huang et al. 2018). But the main purpose of our work is improving continuous normalizing flow, so we mainly compare our result with FFJORD.

For tabular data, we use the same neural network architectures as those in FFJORD, and for image data, the same multiscale (Dinh, Sohl-Dickstein, and Bengio 2017) architectures as in FFJORD are used. For image data, due to our hardware limitation, we used a smaller batch size (200) and smaller number of total training epoch (200). Despite these adjustments, we find the training quality of FFJORD is not affected. For tabular data the setup is the same as those in FFJORD.

The results of average NFE and negative log-likelihood on testing data are shown in Table 1. For average NFE, our model significantly outperforms FFJORD. On all data sets, our model uses fewer NFE in training. The reduction of NFE ranges from 42.3% to 71.3%. The largest reduction, 71.3%, occurs on POWER data set. Note that the actually training time is approximately proportional to NFE. Therefore, our model takes significantly less time in training.

In terms of testing loss, our model is comparable to the original FFJORD. It can be seen that our testing loss is very similar to that of FFJORD*, which is the FFJORD model run by us. Our model even produces lower testing loss on four out of seven data sets than FFJORD*. This result matches our expectation that the additional TPR loss does not affect the performance much, as discussed in Section 3. After all,

Table 1: The average NFE and testing negative log-likelihood on tabular data (in nats) and image data (in bits/dim). FFJORD represents the results reported in the original paper, while FFJORD* is the results run by us. The value of NFE is not reported in the original paper of FFJORD.

	POWER	GAS	HEPMASS	MINIBOONE	BSDS300	MNIST	CIFAR10
				Average NFE			
FFJORD*	885.19	488.00	628.41	107.29	284.51	399.01	530.41
Ours	**253.88**	**178.08**	**260.14**	**32.76**	**87.05**	**221.09**	**306.12**
				Testing Loss			
Real NVP	-0.17	-8.33	18.71	13.55	-153.28	1.06	3.49
Glow	-0.17	-8.15	18.92	11.35	-155.07	1.05	**3.35**
FFJORD	-0.46	-8.59	**14.92**	**10.43**	-157.40	0.99	3.40
FFJORD*	-0.46	**-11.56**	15.79	11.37	-157.46	0.96	**3.35**
Ours	**-0.50**	-11.36	15.85	11.34	**-157.94**	**0.95**	3.36
MADE	3.03	-3.56	20.98	15.59	-148.85	2.04	5.67
MAF	-0.24	-10.08	17.70	11.75	-155.69	1.89	4.31
TAN	-0.48	-11.19	15.12	11.01	-157.03	1.19	3.98
MAF-DDSF	-0.62	-11.96	15.09	8.86	-157.73	-	-

our purpose is improving training efficiency but not testing loss.

For image data sets, MNIST and CIFAR10, our model also works great. The two data sets have the highest dimension of data. Therefore, they are the most difficult to train among all these data sets. The fact that our model yields good results on the two data sets indicates that our model can work on difficult data with high dimensions. For the two image data sets, the generated images sampled from our trained model are shown in Figure 3.

4.2 Variational autoencoder (VAE)

We perform VAE experiments on four data sets obtained from (Berg et al. 2018). The loss of this task is the negative evidence lower bound (ELBO).

$$L(\mathbf{x}) = \mathrm{D}_{\mathrm{KL}}(q_\phi(\mathbf{z}|\mathbf{x})\|p(\mathbf{z})) - \mathop{\mathbb{E}}_{q_\phi(\mathbf{z}|\mathbf{x})}[\log p_\theta(\mathbf{x}|\mathbf{z})]. \quad (15)$$

In this experiment, we further compare the state-of-the-art discrete models, including Planar Flow (Rezende and Mohamed 2015), Inverse Autoregressive Flow (IAF) (Kingma et al. 2016) and Sylvester normalizing flow (Berg et al. 2018).

The hyper-parameters of neural network architecture and training of our model are the same as those in FFJORD. In short, the neural network is composed of the low-rank parameter encoding layers, to encode the input data into CNF. The learning rate is set to be 5×10^{-4} and is divided by 10 when the validation error does not decrease for 35 epochs. Each model is run three times in experiment.

The result of average training NFE and testing negative ELBO is shown in Table 2. For average NFE, our model still significantly outperforms FFJORD. Our model has reduced NFE ranging from 19.3% to 32.1%. The largest reduction is on the Caltech Silhouettes data set. For the negative ELBO, our model performs very similarly to FFJORD. This exactly meets our expectation that simply adding TPR reduces NFE but keeps testing loss the same.

5 Conclusion

We have improved the computational efficiency of continuous normalizing flow models. High computational cost was the largest limitation of CNF. We proposed to add a loss function based on polynomial regression to regularize the trajectory shape of the ODE solution. Our method is proposed to reduce the truncation errors of solving ODE and can result in fewer NFE. Furthermore, we proved that with this regularization, there are always optimal solutions of the vector field for the original loss function, and we argued that our new regularization doesn't harm testing loss. Empirically, our model reduces a great amount of computation cost, while reaching a comparable testing to FFJORD.

Acknowledgments

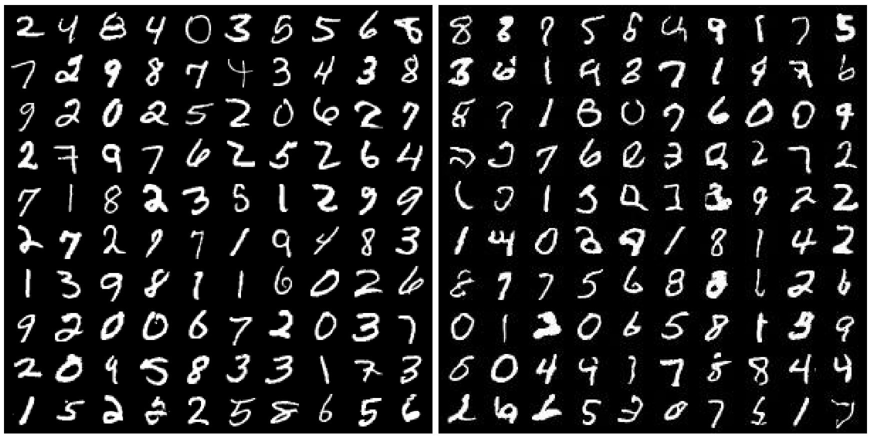
This study was supported in part by the Ministry of Science and Technology (MOST) of Taiwan, R.O.C., under Contracts 107-2221-E-001-009-MY3 and 106-3114-E-002-008.

References

Berg, R. v. d.; Hasenclever, L.; Tomczak, J. M.; and Welling, M. 2018. Sylvester normalizing flows for variational inference. In *Conference on Uncertainty in Artificial Intelligence*, 393–402.

Chen, R. T.; and Duvenaud, D. K. 2019. Neural networks with cheap differential operators. In *Advances in Neural Information Processing Systems*, 9961–9971.

Chen, T. Q.; Rubanova, Y.; Bettencourt, J.; and Duvenaud, D. K. 2018. Neural ordinary differential equations. In *Advances in Neural Information Processing Systems*, 6571–6583.


(a) Left: Ground truth images of MNIST data set. Right: Generated images by our model trained on MNIST data set.

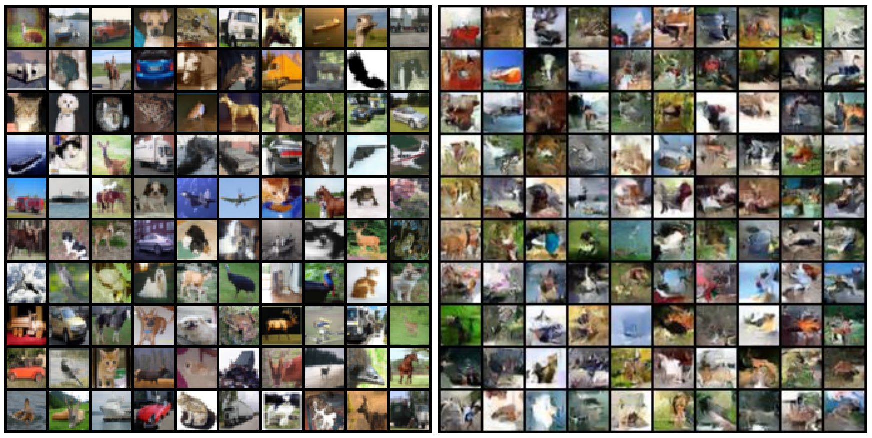

(b) Left: Ground truth images of CIFAR10 data set. Right: Generated images by our model trained on CIFAR10 data set..

Figure 3: The true images and the generated images by our model on two image data sets, MNIST and CIFAR10. The images are generated by running the CNF model in reverse with Gaussian noise as input.

Table 2: The average NFE and testing negative ELBO, with mean and stdev, for VAE models on four data sets. FFJORD represents the results reported in the original paper, while FFJORD* is the results run by us. The value of NFE is not reported in the original paper of FFJORD

	MNIST	Omniglot	Frey Faces	Caltech Silhouettes
		Average NFE		
FFJORD*	58.72 ± 1.32	108.73 ± 5.15	72.19 ± 6.96	39.83 ± 2.56
Ours	**40.45** ± **0.37**	**87.73** ± **2.59**	**52.71** ± **0.87**	**27.06** ± **0.51**
		Testing Loss		
No Flow	86.55 ± .06	104.28 ± .39	4.53 ± .02	110.80 ± 0.46
Planar	86.06 ± .31	102.65 ± .42	4.40 ± .06	109.66 ± 0.42
IAF	84.20 ± .17	102.41 ± .04	4.47 ± .05	111.58 ± 0.38
Sylvester	83.32 ± .06	99.00 ± .04	4.45 ± .04	104.62 ± 0.29
FFJORD	82.82 ± .01	98.33 ± .09	**4.39** ± **.01**	104.03 ± 0.43
FFJORD*	**81.90** ± **.08**	**97.40** ± **.21**	4.49 ± .04	102.84 ± 1.31
Ours	81.94 ± .06	97.52 ± .02	4.51 ± .08	**102.12** ± **0.56**

Dinh, L.; Sohl-Dickstein, J.; and Bengio, S. 2017. Density estimation using Real NVP. In *International Conference on Learning Representations*.

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.

Finlay, C.; Jacobsen, J.-H.; Nurbekyan, L.; and Oberman, A. M. 2020. How to train your neural ode. *arXiv preprint arXiv:2002.02798* .

Grathwohl, W.; Chen, R. T.; Betterncourt, J.; Sutskever, I.; and Duvenaud, D. 2019. FFJORD: free-form continuous dynamics for scalable reversible generative models. In *International Conference on Learning Representations*.

Huang, C.-W.; Dinh, L.; and Courville, A. 2020. Augmented normalizing flows: Bridging the gap between generative flows and latent variable models. *arXiv preprint arXiv:2002.07101* .

Huang, C.-W.; Krueger, D.; Lacoste, A.; and Courville, A. 2018. Neural autoregressive flows. *arXiv preprint arXiv:1804.00779* .

Hutchinson, M. 1989. A stochastic estimator of the trace of the influence matrix for Laplacian smoothing splines. *Communications in Statistics-Simulation and Computation* 18(3): 1059–1076.

Kelly, J.; Bettencourt, J.; Johnson, M. J.; and Duvenaud, D. K. 2020. Learning differential equations that are easy to solve. *Advances in Neural Information Processing Systems* 33.

Kingma, D. P.; and Dhariwal, P. 2018. Glow: Generative flow with invertible 1x1 convolutions. In *Advances in Neural Information Processing Systems*, 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 *Advances in neural information processing systems*, 4743–4751.

Kingma, D. P.; and Welling, M. 2018. Auto-encoding variational bayes. In *Advances in Neural Information Processing Systems*, 6117–6128.

Miyato, T.; Kataoka, T.; Koyama, M.; and Yoshida, Y. 2018. Spectral normalization for generative adversarial networks. In *International Conference on Learning Representations*.

Oliva, J. B.; Dubey, A.; Zaheer, M.; Poczos, B.; Salakhutdinov, R.; Xing, E. P.; and Schneider, J. 2018. Transformation autoregressive networks. *arXiv preprint arXiv:1801.09819* .

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* .

Papamakarios, G.; Pavlakou, T.; and Murray, I. 2017. Masked autoregressive flow for density estimation. In *Advances in Neural Information Processing Systems*, 2338–2347.

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

Salman, H.; Yadollahpour, P.; Fletcher, T.; and Batmanghelich, K. 2018. Deep Diffeomorphic Normalizing Flows. *arXiv preprint arXiv:1810.03256* .

A Proofs

A.1 Proof of Theorem 1

To prove Theorem 1, we construct a vector field $\mathbf{v}(\mathbf{y},t)$ to satisfy the two conditions. The first one is $\frac{d\mathbf{v}}{dt}=0$ for all $\mathbf{y}$ and t. And the second one is the initial and final values of $\log p(t)$.

Let $\mathbf{s}(\mathbf{x},t)$ be the solution of the differential equation (2) in Section 2.1, with $\mathbf{x}$ being its initial value, i.e. $\mathbf{s}(\mathbf{x},t_0)=\mathbf{x}$. Define $\mathbf{z}(\mathbf{x})=\mathbf{s}(\mathbf{x},t_1)$ be the end value of the solution. Because $\frac{d\mathbf{v}}{dt}=0$, $\mathbf{s}$ is moving in space at a constant velocity from $\mathbf{x}$ to $\mathbf{z}$, so we can deduce that

$$\mathbf{s}(\mathbf{x},t)=\mathbf{x}+\frac{t-t_0}{t_1-t_0}\left(\mathbf{z}(\mathbf{x})-\mathbf{x}\right). \tag{16}$$

Once the trajectories $\mathbf{s}(\mathbf{x},t)$ are all defined, the vector field $\mathbf{v}(\mathbf{y},t)$ is also determined. Consequently, the goal to find the vector field $\mathbf{v}(\mathbf{y},t)$ becomes finding the mapping $\mathbf{z}(\mathbf{x})$.

Note that $\mathbf{v}(\mathbf{y},t)$ should be well-defined at any point. For example, the function $\mathbf{z}(\mathbf{x})=-\mathbf{x}$ will lead to an ill-defined $\mathbf{v}$, because at $t=\frac{t_0+t_1}{2}$, $\mathbf{s}$ is at the origin no matter what its initial value $\mathbf{x}$ is, and thus $\mathbf{v}(0,\frac{t_0+t_1}{2})$ can not be defined.

To make $\mathbf{v}(\mathbf{y},t)$ well-defined, we must find the requirement. Because $\mathbf{v}$ is constant on the trajectory $\mathbf{s}(\mathbf{x},t)$, we have the equation $\mathbf{v}(\mathbf{s}(\mathbf{x},t),t)=\mathbf{v}(\mathbf{x},t_0)$. It can be seen that $\mathbf{v}(\mathbf{s}(\mathbf{x},t),t)$ is well-defined if there is only one unique $\mathbf{x}$ that traverses to $\mathbf{s}(\mathbf{x},t)$ at time t. In other words, the inverse function $\mathbf{x}=\mathbf{s}^{-1}(\mathbf{y},t)$ needs to be well-defined. From the inverse function theorem, $\mathbf{s}(\mathbf{x},t)$ is invertible if the determinant of Jacobian is not zero, i.e. $\det\frac{\partial\mathbf{s}(\mathbf{x},t)}{\partial\mathbf{x}}\neq 0$ for all $\mathbf{x}$ and t. Substituting $\mathbf{s}$ with equation (16), the requirement of determinant of Jacobian becomes

$$\det\left((1-\xi)I+\xi\frac{\partial\mathbf{z}}{\partial\mathbf{x}}\right)\neq 0,\quad \text{where } 0\leq\xi\equiv\frac{t-t_0}{t_1-t_0}\leq 1. \tag{17}$$

The second condition to prove the theorem is to satisfy the initial and final conditions of $\log p$. Given the transformation $\mathbf{z}=\mathbf{z}(\mathbf{x})$, we have the equation for probabilities, $p(\mathbf{x})=p(\mathbf{z})\left|\det\frac{\partial\mathbf{z}}{\partial\mathbf{x}}\right|$. Since the initial value condition is $p(\mathbf{x})=p_0(\mathbf{x})$ and the final value condition is $p(\mathbf{z})=p_1(\mathbf{z})$, we must have the equation below for $\mathbf{z}(\mathbf{x})$,

$$p_0(\mathbf{x})=p_1(\mathbf{z})\left|\det\frac{\partial\mathbf{z}}{\partial\mathbf{x}}\right|. \tag{18}$$

To prove Theorem 1, we have to find at least one $\mathbf{z}(\mathbf{x})$ such that both conditions (17) and (18) are satisfied. Our trick is to assume that the function $\mathbf{z}(\mathbf{x})$ satisfies the equations

$$\begin{aligned} z_1 &= z_1(x_1),\\ z_2 &= z_2(x_1,x_2),\\ &\vdots\\ z_d &= z_d(x_1,x_2,\ldots,x_d), \end{aligned} \tag{19}$$

where x_i and z_i are the i-th dimensions of $\mathbf{x}$ and $\mathbf{z}$. With these equations, the Jacobian matrix $\frac{\partial\mathbf{z}}{\partial\mathbf{x}}$ is simply a lower triangular matrix, and then its determinant is simply $\prod_i\frac{\partial z_i}{\partial x_i}$.

Satisfaction of equation (18) Since p_0 and p_1 can be written as the product of conditional probabilities,

$$\begin{aligned} p_0(\mathbf{x}) &= p_0(x_1)p_0(x_2|x_1)\ldots p_0(x_d|x_1,\ldots,x_{d-1}),\\ p_1(\mathbf{z}) &= p_1(z_1)p_1(z_2|z_1)\ldots p_1(z_d|z_1,\ldots,z_{d-1}), \end{aligned} \tag{20}$$

we can construct $\mathbf{z}(\mathbf{x})$ by assuming that it follows the differential equations

$$\begin{aligned} \frac{\partial z_1}{\partial x_1} &= \frac{p_0(x_1)}{p_1(z_1)},\\ \frac{\partial z_2}{\partial x_2} &= \frac{p_0(x_2|x_1)}{p_1(z_2|z_1)},\\ &\vdots\\ \frac{\partial z_d}{\partial x_d} &= \frac{p_0(x_d|x_{1:d-1})}{p_1(z_d|z_{1:d-1})}, \end{aligned} \tag{21}$$

where $x_{1:d-1}$ denotes the vector $(x_1,x_2,\ldots,x_{d-1})$. After these equation are solved, $\mathbf{z}(\mathbf{x})$ can be determined. And due to these equations, the determinant of Jacobian becomes

$$\det\frac{\partial\mathbf{z}}{\partial\mathbf{x}}=\prod_{i=1}^{d}\frac{p_0(x_i|x_{1:i-1})}{p_1(z_i|x_{1:i-1})}=\frac{p_0(\mathbf{x})}{p_1(\mathbf{z})}. \tag{22}$$

Thus, condition (18) is satisfied.

Satisfaction of inequality (17) Since $\frac{\partial\mathbf{z}}{\partial\mathbf{x}}$ is a triangular matrix, the left hand side of equation (17) becomes

$$\begin{aligned} \det\left((1-\xi)I+\xi\frac{\partial\mathbf{z}}{\partial\mathbf{x}}\right) &= \prod_{i=1}^{d}\left((1-\xi)+\xi\frac{\partial z_i}{\partial x_i}\right)\\ &= \prod_{i=1}^{d}\left((1-\xi)+\xi\frac{p_0(x_i|x_{1:i-1})}{p_1(z_i|z_{1:i-1})}\right)\\ &> 0. \end{aligned} \tag{23}$$

The determinant is always greater than zero, so condition (17) is satisfied.

Because we can construct at least one solution $\mathbf{v}(\mathbf{y},t)$ to satisfy all the conditions, Theorem 1 is proved.

A.2 Proof of Theorem 2

If we can find any infinitesimal function $\delta\mathbf{z}(\mathbf{x})$ such that $\mathbf{z}(\mathbf{x})+\delta\mathbf{z}(\mathbf{x})$ still satisfies the conditions (17) and (18), then we can arbitrarily construct another solution of $\mathbf{v}$ based on the solutions found. And thus there are infinitely many solutions of $\mathbf{v}$.

Assume that

$$\mathbf{z}'(\mathbf{x},\epsilon)=\mathbf{z}(\mathbf{x})+\epsilon\mathbf{v}'(\mathbf{z}(\mathbf{x})), \tag{24}$$

where $\mathbf{z}(\mathbf{x})$ is a solution that satisfies (17) and (18), and ϵ is an infinitesimal variable. To make $\mathbf{z}'(\mathbf{x},\epsilon)$ still a solution, we need to find $\mathbf{v}'$ such that $\mathbf{z}'$ also satisfies (17) and (18).

First, the left hand side of condition (17) now becomes

$$\det\left((1-\xi)I+\xi\frac{\partial\mathbf{z}}{\partial\mathbf{x}}\left(I+\epsilon\frac{\partial\mathbf{v}'}{\partial\mathbf{z}}\right)\right). \tag{25}$$

To satisfy condition (17), we assume that every element of the Jacobian matrix $\frac{\partial \mathbf{v}'}{\partial \mathbf{z}}$ is bounded. If they are bounded, we can always find an ϵ sufficiently small such that the overall determinant still stays unequal to zero.

For the condition (18), we don't directly use equation (18) to prove. Instead, we assume that $\log p(\mathbf{z}, \epsilon)$ is the log probability with the points in space following the trajectory in equation (24). So $\log p(\mathbf{z}') = \log p(\mathbf{z}'(\mathbf{x}, \epsilon), \epsilon)$. Because we have $\log p(\mathbf{z}, 0) = \log p_1(\mathbf{z})$ and we need that $\log p(\mathbf{z}') = \log p_1(\mathbf{z}')$, we can obtain the equation below,

$$\frac{\partial}{\partial \epsilon} \log p(\mathbf{z}, \epsilon)\bigg|_{\epsilon=0} = 0. \tag{26}$$

On the other hand, from the dynamics of $\log p$ in equation (3), we also know that

$$\frac{d}{d\epsilon} \log p(\mathbf{z}')\bigg|_{\epsilon=0} = -\nabla_{\mathbf{z}} \cdot \mathbf{v}', \tag{27}$$

where $\nabla_{\mathbf{z}} \cdot \mathbf{v}' \equiv \mathrm{Tr}\left(\frac{\partial \mathbf{v}'}{\partial \mathbf{z}}\right)$ is called the divergence of $\mathbf{v}'$. Below, we will omit the subscript $\mathbf{z}$ in $\nabla_{\mathbf{z}}$ and just use ∇. From the chain rule, $\frac{d \log p}{d\epsilon}\Big|_{\epsilon=0} = \frac{\partial \log p}{\partial \epsilon}\Big|_{\epsilon=0} + \mathbf{v}' \cdot \boldsymbol{\nabla} \log p_1$, we can derive the following equation

$$\mathbf{v}' \cdot \boldsymbol{\nabla} \log p_1 + \boldsymbol{\nabla} \cdot \mathbf{v}' = 0. \tag{28}$$

To find a $\mathbf{v}'(\mathbf{z})$ for equation (28), let us first assume that $\mathbf{v}'$ has the form

$$\mathbf{v}'(\mathbf{z}) = e^{g(\mathbf{z})} \mathbf{u}(\mathbf{z}). \tag{29}$$

Substituting it into equation (28), we get

$$\mathbf{u} \cdot \boldsymbol{\nabla} \log p_1 + \boldsymbol{\nabla} \cdot \mathbf{u} + \mathbf{u} \cdot \boldsymbol{\nabla} g = 0. \tag{30}$$

Because $g(\mathbf{z})$ can be any function, we can set $g = -\log p_1$, and obtain the equation

$$\boldsymbol{\nabla} \cdot \mathbf{u} = 0. \tag{31}$$

And thus

$$\mathbf{v}' = \frac{\mathbf{u}}{p_1}. \tag{32}$$

It may seem that we have completed the proof because there are infinite solutions for the equation $\boldsymbol{\nabla} \cdot \mathbf{u} = 0$. However, most of the solutions of $\mathbf{u}$ do not guarantee that the Jacobian matrix $\frac{\partial \mathbf{v}'}{\partial \mathbf{z}}$ is bounded. Since p_1 asymptotically tends to 0 at infinity, if $\mathbf{u}$ does not decay faster than p_1, $\mathbf{v}'$ and its Jacobian will explode when approaching infinity. This is why this proof doesn't work for $d = 1$, because $\boldsymbol{\nabla} \cdot \mathbf{u} = 0$ implies that $\mathbf{u}$ is a constant and then should be 0.

If we can find a $\mathbf{u}(\mathbf{z})$ that is nonzero only in a bounded area, the problem can be solved. We can see that it is possible by imagining an incompressible fluid confined in a box. Because of incompressibility, the velocity field of the fluid has zero divergence everywhere inside the box. In addition, the velocity field outside the box is always zero. Therefore, the velocity field is nonzero only in the box and has zero divergence everywhere.

Mathematically, we can set $\mathbf{u}(\mathbf{z})$ as

$$u_i = -c_i \frac{L_i}{\pi} \left(1 + \cos\left(\frac{z_i \pi}{L_i}\right)\right) \prod_{j \neq i} \sin\left(\frac{z_j \pi}{L_j}\right) \tag{33}$$

inside the box $-L_i \leq z_i \leq L_i$, for $i = 1, 2, \ldots, d$, and $\mathbf{u} = 0$ outside the box. It can be seen that $\mathbf{u}$ is continuous everywhere, and it has the divergence

$$\boldsymbol{\nabla} \cdot \mathbf{u} = \left(\sum_i c_i\right) \prod_i \sin\left(\frac{z_i \pi}{L_i}\right) \tag{34}$$

inside the box. For $d > 1$, we can always find at least one set of c_i so that $\sum_i c_i = 0$ and thus $\boldsymbol{\nabla} \cdot \mathbf{u} = 0$ everywhere. Consequently, because of the existence of $\mathbf{u}$ and $\mathbf{v}'$, the proof is completed.