Neural Networks with Cheap Differential Operators

Ricky T. Q. Chen [1] **David Duvenaud** [1]

Abstract

Gradients of neural networks can be computed efficiently for any architecture, but some applications require computing differential operators with higher time complexity. We describe a family of neural network architectures that allow easy access to a family of differential operators involving dimension-wise derivatives, and we show how to modify the backward computation graph to compute them efficiently. We demonstrate the use of these operators in implicit ODE solvers, exact density evaluation for continuous normalizing flows, and parameter estimation for stochastic differential equations by matching the Fokker-Planck equation.

1. Introduction

Artificial neural networks are ubiquitous tools as function approximators in a large number of fields. However, its use for applications involving differential equations is still in its infancy. While many focus on the training of black-box neural nets to approximately represent differential equations, less have focused on designing neural networks that work well in the context of differential operators. In this work, we propose to construct the computation graph representing a neural net in a manner that allows a family of differential operators to be efficiently computed. The proposed method works by removing and splicing connections within the graph to allow different forward and backward connections.

Common Differential Operators. In differential equations, it is common to see differential operators such as the divergence (denoted $\nabla\cdot$) or Laplace (denoted ∇^2) operators.

$$\nabla \cdot f := \sum_{i=1}^{d} \frac{\partial f_i(x)}{\partial x_i}, \qquad \nabla^2 g := \sum_{i=1}^{d} \frac{\partial^2 g_i(x)}{\partial x_i^2} \tag{1}$$

[1]University of Toronto, Vector Institute. Correspondence to: Ricky T. Q. Chen <rtqichen@cs.toronto.edu>.

First workshop on *Invertible Neural Networks and Normalizing Flows* (ICML 2019), Long Beach, CA, USA

Automatic differentiation software excel at computing gradients, but cannot efficiently compute the divergence operator. Oftentimes we want to evaluate these types of operators in order to find a solution that satisfies a differential equation, or as a downstream application. For instance, once we have learned a model for a stochastic differential equation (SDE), we may want to apply Fokker-Planck to compute the probability of our samples. This would require computing the divergence and Laplace operators on the drift and diffusion terms on the learned SDE.

Limitation of Automatic Differentiation. The forward evaluation of a neural network can be viewed as traversing a computation graph, while reverse-mode automatic differentiation (AD)–a.k.a backpropagation–traverses the same set of nodes in the reverse direction. However, this is limited to computing vector-Jacobian products. In general, the time complexity for constructing the full Jacobian using naïve AD grows linearly with the dimensionality of the input or output. Unfortunately, this is true as well for extracting the diagonal elements of the Jacobian, since a vector-Jacobian product can only obtain one element of the diagonal per evaluation.

2. DiffOpNet: One-Pass Dimension-wise Derivatives

Given a function $f : \mathbb{R}^d \to \mathbb{R}^d$, we seek to obtain a vector containing its *dimension-wise* k-th order derivatives,

$$\mathcal{D}^k_{\text{dim}} f := \begin{bmatrix} \frac{\partial^k f_1(x)}{\partial x_1^k} & \cdots & \frac{\partial^k f_d(x)}{\partial x_d^k} \end{bmatrix}^T \in \mathbb{R}^d \tag{2}$$

using only k evaluations of automatic differentiation regardless of the dimension d. For notational simplicity, we denote $\mathcal{D}_{\text{dim}} := \mathcal{D}^1_{\text{dim}}$. The divergence operator is obtained by simply summing over the dimensions $\nabla \cdot f = \sum_i^d [\mathcal{D}_{\text{dim}} f]_i$.

The dimension-wise derivatives can be computed efficiently by introducing a bottleneck in the computation graph where we can prune connections to remove paths that connect x_i and $f_j(x)$ when $i \neq j$. During reverse-mode AD, this sets the corresponding partial derivatives to zero and standard AD will produce the vector $\mathcal{D}^k_{\text{dim}} f$. When we want to backpropagate through $\mathcal{D}^k_{\text{dim}} f$, we simply reconnect the missing connections to ensure correct gradients.

2.1. Building the Computation Graph

We build the computation graph for $f(x)$ by first constructing hidden vectors $h_i \in \mathbb{R}^{d_h}$ that don't depend on x_i, and then concatenating them with x_i to be fed into an arbitrary neural network. We can describe this approach as two main steps:

1. $h_i = r_i(x_{-i})$ where r is a neural network and x_{-i} denotes the vector of length $\mathbb{R}^{d-1}$ where x_i is taken out. All elements $\{h_i\}_{i=1}^d$ can be computed in parallel by using networks with masked weights, which exist for both fully connected (Germain et al., 2015) and convolutional architectures (Oord et al., 2016).

2. $f_i(x) = g_i(x_i, h_i)$ where $g_i : \mathbb{R}^{d+d_h} \to \mathbb{R}$ is a neural network that takes as input the concatenated vector $[x_i, h_i]$. All dimensions of $f_i(x)$ can be computed in parallel if the g_i's are composed of matrix-vector and element-wise operations, as is standard in deep learning.

Generalization of existing works. This computation graph contains many models as special cases (Kingma et al., 2016; Papamakarios et al., 2017; Huang et al., 2018; Dinh et al., 2014; 2016). Whereas existing works focus on structuring the Jacobian to be triangular, the $f(x)$ by our construction has a fully filled-in Jacobian. Instead, we use the special structure in the construction of f to manipulate the graph for easier access to dimension-wise derivatives.

Universality and expressiveness. This computation graph introduces a bottleneck in terms of expressiveness. If $d_h \geq d - 1$, then this graph is at least as expressive as a standard neural network, since we can simply set $h_i = x_{-i}$ to recover a standard neural net. Due to this, any $f(x)$ constructed in this manner is still a "universal approximator" if d_h and the width of the layers in g are sufficiently large (Cybenko, 1989; Hornik, 1991). On the other hand, we would like to have $d_h \ll d$ to reduce the amount of compute in practice. In our experiments, we find that values in $\{8, 16, 32\}$ are sufficiently expressive while keeping wall-clock time low. In comparison, other works that make use of masking to parallelize computation only use $d_h \leq 2$ while use simple affine transformations as g_i (Kingma et al., 2016; Papamakarios et al., 2017; Dinh et al., 2014; 2016).

2.2. Splicing the Computation Graph

A single call to naïve reverse-mode automatic differentiation (AD) computes vector-Jacobian products.

$$v^T \frac{\partial f(x)}{\partial x} = \sum_i v_i \frac{\partial f_i(x)}{\partial x} \tag{3}$$

By choosing v such that $v_i = 1$ and $v_j = 0 \; \forall j \neq i$, then we obtain a single row of the Jacobian $\frac{\partial f_i(x)}{\partial x}$ which contains the dimension-wise derivative of the i-th dimension. To obtain the full Jacobian or even $\mathcal{D}_{\text{dim}} f$ would require d AD calls.

Now suppose f is constructed in the manner described in 2.1. Let $\hat{h}$ be h but with the backward connection removed, so that AD would return $\frac{\partial \hat{h}}{\partial x_j} = 0$ for any index j. This kind of computation graph manipulation is equivalent to the use of `stop_gradient` in Tensorflow (Abadi et al., 2016) or `detach` in PyTorch (Paszke et al., 2017). Let $\hat{f}(x) = g(x, \hat{h})$, then the Jacobian of $\hat{f}(x)$ contains only zeros on the off-diagonal elements.

$$\frac{\partial \hat{f}_i(x)}{\partial x_j} = \frac{\partial g_i(x_i, \hat{h}_i)}{\partial x_j} = \begin{cases} \frac{\partial f_i(x)}{\partial x_i} & \text{if } i = j \\ 0 & \text{if } i \neq j \end{cases} \tag{4}$$

As the Jacobian of $\hat{f}(x)$ is a diagonal matrix, we can recover the diagonals by multiplying by $\mathbf{1}$, denoting a vector with all elements equal to one.

$$\mathbf{1}^T \frac{\partial \hat{f}(x)}{\partial x} = \mathcal{D}_{\text{dim}} \hat{f} = \begin{bmatrix} \frac{\partial f_1(x)}{\partial x_1} & \cdots & \frac{\partial f_d(x)}{\partial x_d} \end{bmatrix}^T = \mathcal{D}_{\text{dim}} f \tag{5}$$

The higher orders $\mathcal{D}^k_{\text{dim}}$ can be obtained by k AD calls, as $\hat{f}_i(x)$ is only connected to the i-th dimension of x in the computation graph, so any differential operator on $\hat{f}_i$ only contains the derivatives $\frac{\partial \hat{f}_i(x)}{\partial x_i}$. This is illustrated in the following recursion:

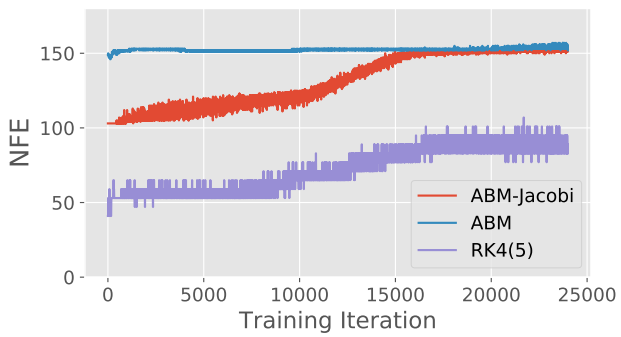
$$\mathbf{1}^T \frac{\partial \mathcal{D}^{k-1}_{\text{dim}} \hat{f}(x)}{\partial x} = \mathcal{D}^k_{\text{dim}} \hat{f}(x) = \mathcal{D}^k_{\text{dim}} f(x) \tag{6}$$

As connections have been removed from the computation graph, backpropagating through $\mathcal{D}^k_{\text{dim}} \hat{f}$ would give erroneous gradients as the connection between $f_i(x)$ and x_j for $j \neq i$ was severed. To ensure correct gradients, we must reconnect $\hat{h}$ and h in the backward pass,

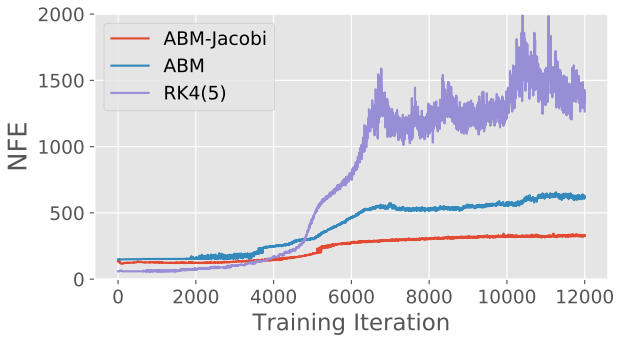
$$\frac{\partial \mathcal{D}^k_{\text{dim}} \hat{f}}{\partial w} + \frac{\partial \mathcal{D}^k_{\text{dim}} \hat{f}}{\partial \hat{h}} \frac{\partial h}{\partial w} = \frac{\partial \mathcal{D}^k_{\text{dim}} f}{\partial w} \tag{7}$$

where w is any node in the computation graph. Through accessing only $\hat{f}$ using the left-hand-sides of equations (5), (6), and (7), we can efficiently compute $\mathcal{D}^k_{\text{dim}} f$ and its gradients, shown on the right-hand-sides of (5), (6), and (7). The number of AD calls is dependent on k but independent of d.

In the following sections, we show how access to $\mathcal{D}_{\text{dim}}$ provides improvements in a number of example applications including (i) more efficient ODE solvers for stiff dynamics, (ii) solving for the density of continuous normalizing flows, and (iii) learning stochastic differential equation models by Fokker-Planck matching. Each of the following sections can be read individually and in any order.



(a) Explicit solver is sufficient for nonstiff dynamics.



(b) Training may result in stiff dynamics.

Figure 1: (a) Explicit methods such as RK4(5) are generally more efficient when the system isn't too stiff. (b) However, if a trained dynamics model becomes stiffer, predictor-corrector methods (ABM & ABM-Jacobi) are much more efficient. In difficult cases, the Jacobi-Newton iteration (ABM-Jacobi) uses significantly less evaluations than functional iteration (ABM).

3. Efficient Jacobi Iterations in Implicit Linear Multistep Methods

Ordinary differential equations (ODEs) parameterized by neural networks are typically solved using explicit methods such as Runge-Kutta 4(5) (Hairer and Peters, 1987). However, the learned ODE can often become "stiff", requiring a large number of evaluations to accurately solve using explicit methods. Instead, implicit methods can achieve better accuracy at the cost of solving an inner-loop optimization problem at every step. When the initial guess is given by an explicit method of the same order, this is referred to as a predictor-corrector method (Moulton, 1926; Radhakrishnan and Hindmarsh, 1993). Implicit formulations also show up as inverse problems of explicit linear multistep methods. Invertible ResNets (Behrmann et al., 2018) compute forward Euler steps in the forward pass, but the inverse requires solving backward Euler for every residual block.

The class of linear multistep methods includes forward and backward Euler, explicit and implicit Adams, and backward differentiation formulas.

$$\begin{aligned} &y_{n+s} + a_{s-1}y_{n+s-1} + a_{s-2}y_{n+s-2} + \cdots + a_0 y_n \\ &\quad = h(b_s f(t_{n+s}, y_{n+s}) + \cdots + b_0 f(t_n, y_n)) \end{aligned} \tag{8}$$

where the values of y_i and $f(t_i, y_i)$ from the previous s steps are used to solve for y_{n+s}. When $b_s \neq 0$, this requires solving a non-linear optimization problem as both y_{n+s} and $f(t_{n+s}, y_{n+s})$ appears in the equation, resulting in an implicit method.

Simplifying notation with $y = y_{n+s}$, we can write (8) as a root finding problem.

$$F(y) := y - hb_s f(y) - \delta = 0 \tag{9}$$

where δ is a constant representing the rest of the terms in (8) from previous steps. Newton-Raphson can be used to solve this problem, resulting in an iterative algorithm

$$y^{(k+1)} = y^{(k)} - \left[\frac{\partial F(y^{(k)})}{\partial y^{(k)}}\right]^{-1} F(y^{(k)}) \tag{10}$$

When the full Jacobian inverse is unknown or inefficient to compute, one can approximate using the diagonal entries. This approximation results in the Jacobi-Newton iteration (Radhakrishnan and Hindmarsh, 1993).

$$\begin{aligned} y^{(k+1)} &= y^{(k)} - [\mathcal{D}_{\text{dim}} F(y)]^{-1} \odot F(y^{(k)}) \\ &= y^{(k)} - [\mathbf{1} - hb_s \mathcal{D}_{\text{dim}} f(y)]^{-1} \odot (y - hb_s f(y) - \delta) \end{aligned} \tag{11}$$

where $\odot$ denotes the Hadamard product, $\mathbf{1}$ is a vector with all elements equal to one, and the inverse is taken element-wise. Each iteration requires evaluating f once. In our implementation, the fixed point iteration is repeated until $||y^{(k-1)} - y^{(k)}||/\sqrt{d} \leq \tau_a + \tau_r ||y^{(0)}||_\infty$ for some user-provided tolerance parameters τ_a, τ_r.

Alternatively, when $\left[\frac{\partial F(y)}{\partial y}\right]^{-1}$ is approximated by the identity matrix, the resulting algorithm is referred to as functional iteration (Radhakrishnan and Hindmarsh, 1993). Using our efficient computation of the $\mathcal{D}_{\text{dim}}$ operator, we can apply Jacobi-Newton and obtain faster convergence than functional iteration while maintaining the same computational cost.

We compare a standard Runge-Kutta (RK) solver with adaptive stepping (Shampine, 1986) and a predictor-corrector Adams-Bashforth-Moulton (ABM) method in Figure 1. The learned dynamics is part of a continuous normalizing flow

Model	MNIST		Omniglot	
	ELBO ↑	NLL ↓	ELBO ↑	NLL ↓
VAE	-86.55	82.14	-104.28	97.25
Planar	-86.06	81.91	-102.65	96.04
IAF	-84.20	80.79	-102.41	96.08
Sylvester	-83.32	80.22	-99.00	93.77
FFJORD	-82.82	–	-98.33	–
DiffOp-CNF	**-82.38**	**80.18**	**-97.35**	**93.72**

Table 1: Evidence lower bound and negative log-likelihood for static MNIST and Omniglot.

(discussed in Section 4). The number of function evaluations (NFE) includes both evaluations made in the forward pass and evaluations for solving the adjoint state in the backward pass for parameter updates (Chen et al., 2018). The efficient $\mathcal{D}_{\text{dim}}$ operator is used whenever possible in both the forward and backward directions. As expected, when the learned dynamics model becomes too stiff, RK results in using very small step sizes and uses almost 10 times the number of evaluations as ABM with Jacobi-Newton iterations. When implicit methods are used, Jacobi-Newton can help significantly reduce the number of evaluations required if the dynamics are difficult to solve.

4. Continuous Normalizing Flows with Exact Trace Computation

Continuous normalizing flows (CNF) (Chen et al., 2018) transform particles from a base distribution $p(x_0)$ at time t_0 to another time t_1 according to an ordinary differential equation $\frac{dh}{dt} = f(t, h(t))$.

$$x := x(t_1) = x_0 + \int_{t_0}^{t_1} f(t, h(t))dt \tag{12}$$

The change in distribution as a result of this transformation is described by an instantaneous change of variables equation (Chen et al., 2018)

$$\frac{\partial \log p(t, h(t))}{\partial t} = -\text{Tr}\left(\frac{\partial f(t, h(t))}{\partial h(t)}\right) = -\sum_{i=1}^{d}[\mathcal{D}_{\text{dim}} f]_i \tag{13}$$

If (13) is solved along with (12) as a combined ODE system, we can obtain the density of transformed particles at any desired time t_1.

Due to requiring d AD calls to compute $\mathcal{D}_{\text{dim}} f$ for a black-box neural network f, Grathwohl et al. (2019) adopted the stochastic trace estimator (Skilling, 1989; Hutchinson, 1990) to provide unbiased estimates for $\log p(t, h(t))$. Behrmann et al. (2018) used the same estimator and showed that the per-example standard deviation is roughly 10% of the true $\log p$ value. Furthermore, an unbiased estimator of the log-density has limited uses. For instance, the IWAE objective (Burda et al., 2015) for estimating the negative log-likelihood (NLL) $\log p(x) = \log \mathbb{E}_{z\sim p(z)}[p(x|z)]$ of latent variable models has the following form:

$$\mathcal{L}_{\text{IWAE-}k} = \mathbb{E}_{z_1,\ldots,z_k\sim q(z|x)}\left[\log \frac{1}{k}\sum_{i=1}^{k}\frac{p(x, z_i)}{q(z_i|x)}\right] \tag{14}$$

Flow-based models have been used as the distribution $q(z|x)$, but an unbiased estimator of $\log q$ would not translate into an unbiased estimate of this importance weighted objective, resulting in biased evaluations and biased gradients if used for training. For this reason, FFJORD (Grathwohl et al., 2019) was unable to report NLLs for evaluation which are standardly estimated using (14) with $k = 5000$.

4.1. Exact Trace Computation

By constructing f in the manner described in Section 2.1, we can efficiently compute $\mathcal{D}_{\text{dim}} f$. This allows us to exactly compute (13) using a single AD call, which is the same cost as the stochastic trace estimator. We believe that using exact trace should reduce gradient variance during training, allowing models to converge to better local optima. Furthermore, it should help reduce the complexity of solving (13) as stochastic estimates can lead to more difficult dynamics.

4.2. Latent Variable Model Experiments

We train variational autoencoders (Kingma and Welling, 2013) using the same setup as Berg et al. (2018). This corresponds to training using (14) with $k = 1$, also known as the evidence lower bound (ELBO). Figure 1 shows that even in this simple setting, training CNFs with exact trace using the DiffOpNet architecture can lead to improvements on standard benchmark datasets, static MNIST and Omniglot. Furthermore, we can estimate the NLL of our models using $k = 5000$ for evaluating the quality of the generative model. Interestingly, although the NLLs have not improved significantly, CNFs can achieve much better ELBO values. This may suggest that while strong flow-based models such as CNFs perform better posterior inference, this perhaps comes at the expense of anchoring the generative model to this learned posterior.

4.3. Exact vs. Stochastic CNFs

We take a closer look at the effects of using an exact trace. We use DiffOpNet computation graphs where exact trace can be efficiently computed, and compare against the same architecture but with the Skilling-Hutchinson trace estimator. Figure 2 contains comparisons of models trained using

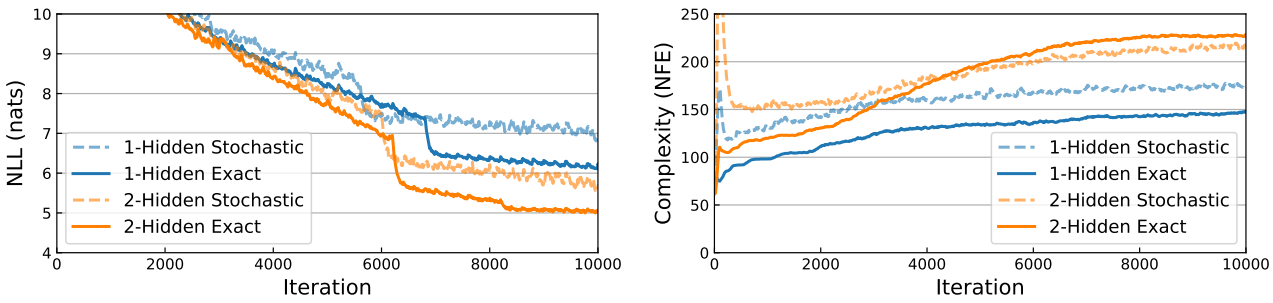


Figure 2: Comparison of exact trace versus stochastically estimated trace on learning continuous normalizing flows with identical initialization. Continuous normalizing flows with exact trace converge faster and can be easier to solve.

maximum likelihood on the MINIBOONE dataset preprocessed by Papamakarios et al. (2017). The comparisons between exact and stochastic trace are carried out by using the same initial weights. We find that not only can exact trace CNFs achieve better training NLLs, they converge faster. With a tolerance of 10^{-5}, we find that using exact trace allows the ODE to be solved with comparable or fewer number of evaluations.

5. Learning Stochastic Differential Equations by Fokker-Planck Matching

Let $\mathbf{x}(t) \in \mathbb{R}^d$ follow a stochastic differential equation (SDE) following a drift function $\mu(\mathbf{x}(t), t)$ and diagonal diffusion matrix $\sigma(\mathbf{x}(t), t)$ in the Itô sense.

$$d\mathbf{x}(t) = \mu(\mathbf{x}(t), t)dt + \sigma(\mathbf{x}(t), t)d\mathbf{W} \tag{15}$$

The Fokker-Planck equation describes how the density of this SDE changes through time.

$$\begin{aligned}\frac{\partial p(t,x)}{\partial t} = \sum_{i=1}^{d} \Big[&- (\mathcal{D}_{\text{dim}}\mu)p - (\nabla p) \odot \mu + (\mathcal{D}^2_{\text{dim}}\text{diag}(\sigma))p \\ &+ 2(\mathcal{D}_{\text{dim}}\text{diag}(\sigma)) \odot (\nabla p) \\ &+ {}^1\!/_2 \,\text{diag}(\sigma)^2 \odot (\mathcal{D}_{\text{dim}}\nabla p) \Big]_i\end{aligned} \tag{16}$$

Written in this form, it's clear where we can take advantage of efficient dimension-wise derivatives.

For simplicity, we make the assumption that the density at any time t can be modeled using a mixture of m Gaussians, which can approximate any distribution if m is large enough.

$$p(t,x) = \sum_{c=1}^{m} \pi_c(t)\mathcal{N}(x; \nu_c(t), \Sigma_c(t)) \tag{17}$$

Then differential operators applied to p can be computed exactly.

$$\begin{aligned}\nabla_x p(t,x) &= -\frac{1}{p(t,x)} \sum_{c=1}^{m} \pi_c(t)\Sigma_c(t)^{-1}(x - \nu_c(t)) \\ \mathcal{D}_{\text{dim}}\nabla_x p(t,x) &= -\sum_{c=1}^{m} \pi_c(t) \left(\nabla p \odot \text{diag}\left(\Sigma_c(t)^{-1}\right)\right)\end{aligned} \tag{18}$$

We parameterize $\mu(t,x)$ and $\sigma(t,x)$ as neural networks with parameters θ, and parameterize $\nu_c(t)$ and $\Sigma_c(t)$ using neural networks with parameters ϕ. We seek to perform maximum-likelihood on the density model p while simultaneously learning an SDE model that satisfies the Fokker-Planck equation (16) applied to this density. Our objective function for training is

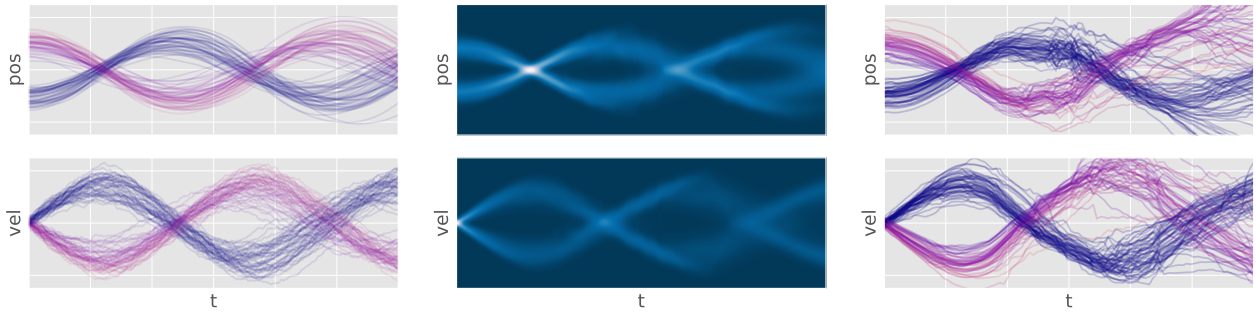
$$\begin{aligned}\max_{\theta,\phi} &\mathbb{E}_{t,x_t \sim p_{\text{data}}}\left[\log p(t, x_t)\right] \\ &+ \lambda \mathbb{E}_{t,x_t \sim p_{\text{data}}}\left[\left|\frac{\partial p(t,x_t)}{\partial t} - \text{FP}(t,x_t)\right|\right]\end{aligned} \tag{19}$$

where $\text{FP}(t, x_t)$ refers to the right-hand-side of (16), and λ is a non-negative weight that is annealed to zero by the end of training. Having a positive λ value regularizes the density model to be closer to the SDE model, which can help guide the SDE parameters at the beginning of training.

Compared to parameter estimation approaches that rely on finite-difference such as pseudo-maximum likelihood (Yoshida, 1992; Aït-Sahalia, 2002) or simulation, this purely functional approach has multiple benefits:

1. No reliance on finite-difference approximations. All partial derivatives are evaluated exactly.
2. No need for sequential evaluations. All observations t, x_t can be trained in parallel.

Importantly, due to the efficient parallel computation of the $\mathcal{D}_{\text{dim}}$ operator, we can scale this training algorithm to higher dimensions without introducing any sequential loops.



Data Learned Density Learned SDE

Figure 3: Result of Fokker-Planck matching for learning a multimodal SDE.

5.1. Feasibility of Fokker-Planck Matching

We examine the feasibility of learning a stochastic differential equation model by optimizing (19). We construct a synthetic experiment where a pendulum is initialized randomly at one of two modes. The pendulum's velocity changes with gravity and is randomly perturbed by a diffusion process. This results in two states, a position and velocity following the stochastic differential equation

$$d\begin{bmatrix}p\\v\end{bmatrix} = \begin{bmatrix}v\\-2\sin(p)\end{bmatrix}dt + \begin{bmatrix}0 & 0\\0 & 0.2\end{bmatrix}dW. \qquad (20)$$

We use black-box neural networks to parameterize the SDE and density models, and use $m = 5$ Gaussian mixtures. The result after training for 30000 iterations is shown in Figure 3. The density model correctly recovers the multimodality of the marginal distributions, including at the initial time, and the SDE model correctly recovers the sinusoidal behavior of the data. The behavior is more erratic where the density model has slight imperfections, but the overall dynamics was recovered successfully.

6. Conclusion

We propose a neural network construction procedure along with a computation graph manipulation method that allows us to obtain dimension-wise derivatives with only one evaluation through reverse-mode AD, whereas naïve AD would require d evaluations. Dimension-wise derivatives are very useful when modeling various differential equations as differential operators frequently appear in such formulations. We show that parameterizing differential equations using this approach allows more efficient solving when the dynamics are stiff, provides a way to scale up continuous normalizing flows without resorting to stochastic evaluations, and gives rise to a completely parallel way of training SDE models through matching the Fokker-Planck equation.

References

Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, pages 265–283, 2016.

Yacine Aït-Sahalia. Maximum likelihood estimation of discretely sampled diffusions: a closed-form approximation approach. *Econometrica*, 70(1):223–262, 2002.

Jens Behrmann, David Duvenaud, and Jörn-Henrik Jacobsen. Invertible residual networks. *arXiv preprint arXiv:1811.00995*, 2018.

Rianne van den Berg, Leonard Hasenclever, Jakub M Tomczak, and Max Welling. Sylvester normalizing flows for variational inference. *arXiv preprint arXiv:1803.05649*, 2018.

Yuri Burda, Roger Grosse, and Ruslan Salakhutdinov. Importance weighted autoencoders. *arXiv preprint arXiv:1509.00519*, 2015.

Ricky T. Q. Chen, Yulia Rubanova, Jesse Bettencourt, and David Duvenaud. Neural ordinary differential equations. *Advances in Neural Information Processing Systems*, 2018.

George Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of control, signals and systems*, 2(4):303–314, 1989.

Laurent Dinh, David Krueger, and Yoshua Bengio. Nice: Non-linear independent components estimation. *arXiv preprint arXiv:1410.8516*, 2014.

Laurent Dinh, Jascha Sohl-Dickstein, and Samy Bengio. Density estimation using real nvp. *arXiv preprint arXiv:1605.08803*, 2016.

Mathieu Germain, Karol Gregor, Iain Murray, and Hugo Larochelle. Made: Masked autoencoder for distribution estimation. In *International Conference on Machine Learning*, pages 881–889, 2015.

Will Grathwohl, Ricky T. Q. Chen, Jesse Bettencourt, Ilya Sutskever, and David Duvenaud. Ffjord: Free-form continuous dynamics for scalable reversible generative models. *International Conference on Learning Representations*, 2019.

Hairer and Peters. *Solving ordinary differential equations I*. Springer Berlin Heidelberg, 1987.

Kurt Hornik. Approximation capabilities of multilayer feedforward networks. *Neural networks*, 4(2):251–257, 1991.

Chin-Wei Huang, David Krueger, Alexandre Lacoste, and Aaron Courville. Neural autoregressive flows. *arXiv preprint arXiv:1804.00779*, 2018.

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

Diederik P Kingma and Max Welling. Auto-encoding variational bayes. *arXiv preprint arXiv:1312.6114*, 2013.

Durk P Kingma, Tim Salimans, Rafal Jozefowicz, Xi Chen, Ilya Sutskever, and Max Welling. Improved variational inference with inverse autoregressive flow. In *Advances in neural information processing systems*, pages 4743–4751, 2016.

Forest Ray Moulton. *New methods in exterior ballistics*. 1926.

Aaron van den Oord, Nal Kalchbrenner, and Koray Kavukcuoglu. Pixel recurrent neural networks. *arXiv preprint arXiv:1601.06759*, 2016.

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

Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. In *NIPS-W*, 2017.

Krishnan Radhakrishnan and Alan C Hindmarsh. Description and use of lsode, the livermore solver for ordinary differential equations. 1993.

Lawrence F Shampine. Some practical runge-kutta formulas. *Mathematics of Computation*, 46(173):135–150, 1986.

John Skilling. The eigenvalues of mega-dimensional matrices. In *Maximum Entropy and Bayesian Methods*, pages 455–466. Springer, 1989.

Nakahiro Yoshida. Estimation for diffusion processes from discrete observation. *Journal of Multivariate Analysis*, 41 (2):220–242, 1992.