Neural Ordinary Differential Equations

Pramook Khungurn

April 28, 2022

This is a note on the paper "Neural Ordinary Differential Equations" by Chen et al.[CRBD18].

1 Introduction

- Many existing neural networks models creates a sequence of hidden states $\mathbf{h}_0$, $\mathbf{h}_1$, $\mathbf{h}_2$, $\ldots$ $\mathbf{h}_T$ by adding something to the previous state:

  $$\mathbf{h}_{t+1} = \mathbf{h}_t + \mathbf{f}(\mathbf{h}_t, t, \boldsymbol{\theta})$$

  Such models include such as residual networks [HZRS15], recurrent neural networks, and normalizing flows [RM15, DKB14].

- What if we take the limit as the number of time step goes to infinity? We will have a differential equation:

  $$\frac{\mathrm{d}\mathbf{h}(t)}{\mathrm{d}t} = \mathbf{f}(\mathbf{h}(t), t, \boldsymbol{\theta}).$$

- To use the network, we simply say that $\mathbf{h}(0)$ is the input layer, and the output is $\mathbf{h}(T)$ at some time T. The output can be found by solving the initial value problem, and this can be done by any black-box differential equation solver.

2 How to train a neural ODE model

- The problem with the above approach is that it is unclear how to train such a neural ODE model.
  - The computation of the solution can require a lot of time steps. Differentiating through these time steps to compute the gradient would requires saving a lot of information in memory.
- The good news is that there is a method to compute the gradient using constant memory (i.e., does not depend on the number of time steps). This is called the **adjoint sensitivity method**. It requires, however, an ODE solve, which can be done, again, by any ODE solver.

2.1 Problem Setup

- Let the hidden state be a vector in $\mathbb{R}^n$. We typically denote it by $\mathbf{z}$.
- Let the neural network's parameters be a vector in $\mathbb{R}^m$, and we typically denote it by $\boldsymbol{\theta}$.
- We will work on a state space vector $\mathbf{r} = (\mathbf{z}, t, \boldsymbol{\theta}) \in \mathbb{R}^{n+1+m}$.
- We will want to see how $\mathbf{r}$ evolves through time. We denote the $\mathbf{r}$ at time t with $\mathbf{r}_t = (\mathbf{z}_t, t, \boldsymbol{\theta})$. Note that $\boldsymbol{\theta}$ does not vary with t.

- It also makes sense to talk about the function that sends t to $\mathbf{r}_t$. We denote this by $\mathbf{R} : \mathbb{R} \to \mathbb{R}^{n+1+m}$, and we can write

$$\mathbf{r}_t = \mathbf{R}(t) = (\mathbf{Z}(t), T(t), \mathbf{\Theta}(t)) = (\mathbf{z}_t, t, \boldsymbol{\theta}).$$

Note that T is the identity function, and $\mathbf{\Theta}$ is a constant function.

- The act of solving the neural ODE is a function that maps $\mathbf{r}_t$ to some $\mathbf{r}_{t+\Delta t}$ for some $\Delta t \geq 0$. Let us denote this function by $\mathbf{s}^+_{\Delta t} : \mathbb{R}^{n+1+m} \to \mathbb{R}^{n+1+m}$. (The letter $\mathbf{s}$ stands for "solve.") We have that

$$\mathbf{s}^+_{\Delta t}(\mathbf{z}_t, t, \boldsymbol{\theta}) = (\mathbf{z}_{t+\Delta}, t, \boldsymbol{\theta}) = \begin{bmatrix} \mathbf{z}_{t+\Delta t} \\ t + \Delta t \\ \boldsymbol{\theta} \end{bmatrix} = \begin{bmatrix} \mathbf{z}_t + \int_t^{t+\Delta t} \mathbf{f}(\mathbf{z}_u, u, \boldsymbol{\theta}) \, \mathrm{d}u \\ t + \Delta t \\ \boldsymbol{\theta} \end{bmatrix}.$$

- The above function runs the ODE for a fixed time internal Δt. However, we can also talk about running the ODE until a fixed time t_1. We denote this by

$$\mathbf{s}^+_{\to t_1}(\mathbf{z}_t, t, \boldsymbol{\theta}) = \mathbf{s}^+_{t_1 - t}(\mathbf{z}_t, t, \boldsymbol{\theta}) = \begin{bmatrix} \mathbf{z}_t + \int_t^{t_1} \mathbf{f}(\mathbf{z}_u, u, \boldsymbol{\theta}) \, \mathrm{d}u \\ t + \Delta t \\ \boldsymbol{\theta} \end{bmatrix}.$$

- When optimizing a neural network, we need a loss function. In our case, the loss function is given by $L : \mathbb{R}^{n+1+m} \to \mathbb{R}$ that maps a state vector to a real number. When we write $L(\mathbf{r}) = L(\mathbf{z}, t, \boldsymbol{\theta})$, it is typical to say that the function only depends on $\mathbf{z}$, the produced hidden state. So,

$$L(\mathbf{r}) = L(\mathbf{z}, t, \boldsymbol{\theta}) = L(\mathbf{z}).$$

- When training a neural ODE, we start with the input state vector $\mathbf{r}_t$. We then solve the ODE to get the state $\mathbf{r}_{t_1}$. We then evaluate $L(\mathbf{r}_{t_1})$ to compute the loss. Let $\mathcal{L} : \mathbb{R}^{n+1+m} \to \mathbb{R}$ be the function that maps the input state to the final loss. This function is thus given by

$$\mathcal{L}(\mathbf{z}_t, t, \boldsymbol{\theta}) = L(\mathbf{s}^+_{\to t_1}(\mathbf{z}_t, t, \boldsymbol{\theta})).$$

- To train the neural network, we need the gradient

$$\nabla_{\S 3} \mathcal{L}(\mathbf{z}_{t_0}, t_0, \boldsymbol{\theta})$$

where t_0 is the time we designate for the input, typically 0. Here, we use the notations for multivariable derivatives from [Khu22] to avoid confusion. $\nabla_{\S 3}\mathcal{L}$ denotes the gradient with respect to the third block of arguments of $\mathcal{L}$, which is the network parameters $\boldsymbol{\theta}$.

2.2 Adjoint Sensitivity Method

- Define the **adjoint** to be the function $\mathbf{a} : \mathbb{R} \to \mathbb{R}^{1 \times (n+1+m)}$ such that

$$\mathbf{a} : t \mapsto \nabla \mathcal{L}(\mathbf{z}_t, t, \boldsymbol{\theta}).$$

In other words,

$$\mathbf{a}(t) = \mathcal{L}(\mathbf{R}(t)) = L(\mathbf{s}^+_{\to t_1}(\mathbf{R}(t)))$$

or $\mathbf{a} = \mathcal{L} \circ \mathbf{R} = L \circ s^+_{\to t_1} \circ \mathbf{R}$.

- With the adjoint function, our end goal is to evaluate

$$\mathbf{a}_{\S 3}(t_0) = \mathbf{a}(t_0)[:, \S 3] = \nabla \mathcal{L}(\mathbf{z}_{t_0}, t_0, \boldsymbol{\theta})[:, \S 3] = \nabla_{\S 3} \mathcal{L}(\mathbf{z}_{t_0}, t_0, \boldsymbol{\theta}).$$

- The adjoint sensivity method relies on the fact that we can express $\mathrm{d}\mathbf{a}/\mathrm{d}t$ in terms for $\mathbf{a}$ and $\mathbf{f}$.

Theorem 1. *We have that*

$$\frac{\mathrm{d}\mathbf{a}(t)}{\mathrm{d}t} = -\mathbf{a}(t)\begin{bmatrix} \nabla_{\S1}\mathbf{f}(\mathbf{z}_t,t,\boldsymbol{\theta}) & \nabla_{\S2}\mathbf{f}(\mathbf{z}_t,t,\boldsymbol{\theta}) & \nabla_{\S3}\mathbf{f}(\mathbf{z}_t,t,\boldsymbol{\theta}) \\ \mathbf{0} & 0 & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} \end{bmatrix}$$

In particular,

$$\begin{aligned} \frac{\mathrm{d}\mathbf{a}_{\S1}(t)}{\mathrm{d}t} &= -\mathbf{a}_{\S1}(t)\nabla_{\S1}\mathbf{f}(\mathbf{z}_t,t,\boldsymbol{\theta}), \\ \frac{\mathrm{d}\mathbf{a}_{\S3}(t)}{\mathrm{d}t} &= -\mathbf{a}_{\S1}(t)\nabla_{\S3}\mathbf{f}(\mathbf{z}_t,t,\boldsymbol{\theta}). \end{aligned}$$

Proof. We have that

$$\frac{\mathrm{d}\mathbf{a}(t)}{\mathrm{d}t} = \lim_{\varepsilon\to 0}\frac{\mathbf{a}(t+\varepsilon)+\mathbf{a}(t)}{\varepsilon}.$$

To prove the theorem, we shall write $\mathbf{a}(t)$ in terms of $\mathbf{a}(t+\varepsilon)$.

Consider the function $\mathcal{L}$. We have that, for any $\varepsilon > 0$ such that $t+\varepsilon < t_1$,

$$\mathcal{L}(\mathbf{z}_t,t,\boldsymbol{\theta}) = \mathcal{L}(\mathbf{z}_{t+\varepsilon},t+\varepsilon,\boldsymbol{\theta}).$$

This is because both $(\mathbf{z}_t,t,\boldsymbol{\theta})$ and $(\mathbf{z}_{t+\varepsilon},t+\varepsilon,\boldsymbol{\theta})$ are on the trajectory to the final state vector $(\mathbf{z}_{t_1},t_1,\boldsymbol{\theta})$. So, starting running the ODE from either points would lead to the same result. As a result, we may say that

$$\mathcal{L} = \mathcal{L}\circ\mathbf{s}_\varepsilon^+$$

if ε is small enough. Applying the chain rule, we have that

$$\begin{aligned} \nabla\mathcal{L}(\mathbf{z}_t,t,\boldsymbol{\theta}) &= \nabla\mathcal{L}(\mathbf{s}_\varepsilon^+(\mathbf{z}_t,t,\boldsymbol{\theta}))\nabla\mathbf{s}_\varepsilon^+(\mathbf{z}_t,t,\boldsymbol{\theta}) \\ \nabla\mathcal{L}(\mathbf{z}_t,t,\boldsymbol{\theta}) &= \nabla\mathcal{L}(\mathbf{z}_{t+\varepsilon},t+\varepsilon,\boldsymbol{\theta})\nabla\mathbf{s}_\varepsilon^+(\mathbf{z}_t,t,\boldsymbol{\theta}) \\ \mathbf{a}(t) &= \mathbf{a}(t+\varepsilon)\nabla\mathbf{s}_\varepsilon^+(\mathbf{z}_t,t,\boldsymbol{\theta}). \end{aligned}$$

Now,

$$\begin{aligned} \mathbf{s}_\varepsilon^+(\mathbf{z}_t,t,\boldsymbol{\theta}) &= \begin{bmatrix} \mathbf{z}_t + \int_t^{t+\varepsilon}\mathbf{f}(\mathbf{z}_u,u,\boldsymbol{\theta})\,\mathrm{d}u \\ t+\varepsilon \\ \boldsymbol{\theta} \end{bmatrix} = \begin{bmatrix} \mathbf{z}_t + \varepsilon\mathbf{f}(\mathbf{z}_t,t,\boldsymbol{\theta}) + O(\varepsilon^2) \\ t+\varepsilon \\ \boldsymbol{\theta} \end{bmatrix} \\ &= \begin{bmatrix} \mathbf{z}_t \\ t \\ \boldsymbol{\theta} \end{bmatrix} + \varepsilon\begin{bmatrix} \mathbf{f}(\mathbf{z}_t,t,\boldsymbol{\theta}) \\ 1 \\ \mathbf{0} \end{bmatrix} + O(\varepsilon^2). \end{aligned}$$

So,

$$\nabla\mathbf{s}_\varepsilon^+(\mathbf{z}_t,t,\boldsymbol{\theta}) = I + \varepsilon\begin{bmatrix} \nabla_{\S1}\mathbf{f}(\mathbf{z}_t,t,\boldsymbol{\theta}) & \nabla_{\S2}\mathbf{f}(\mathbf{z}_t,t,\boldsymbol{\theta}) & \nabla_{\S3}\mathbf{f}(\mathbf{z}_t,t,\boldsymbol{\theta}) \\ \mathbf{0} & 0 & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} \end{bmatrix} + O(\varepsilon^2).$$

This gives

$$\mathbf{a}(t) = \mathbf{a}(t+\varepsilon) + \varepsilon\mathbf{a}(t+\varepsilon)\begin{bmatrix} \nabla_{\S1}\mathbf{f}(\mathbf{z}_t,t,\boldsymbol{\theta}) & \nabla_{\S2}\mathbf{f}(\mathbf{z}_t,t,\boldsymbol{\theta}) & \nabla_{\S3}\mathbf{f}(\mathbf{z}_t,t,\boldsymbol{\theta}) \\ \mathbf{0} & 0 & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} \end{bmatrix} + O(\varepsilon^2),$$

and so

$$\frac{\mathbf{a}(t+\varepsilon)-\mathbf{a}(t)}{\varepsilon} = -\mathbf{a}(t+\varepsilon)\begin{bmatrix} \nabla_{\S 1}\mathbf{f}(\mathbf{z}_t, t, \boldsymbol{\theta}) & \nabla_{\S 2}\mathbf{f}(\mathbf{z}_t, t, \boldsymbol{\theta}) & \nabla_{\S 3}\mathbf{f}(\mathbf{z}_t, t, \boldsymbol{\theta}) \\ \mathbf{0} & 0 & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} \end{bmatrix} + O(\varepsilon).$$

Taking the limit as $\varepsilon \to 0$, we have that

$$\frac{\mathrm{d}\mathbf{a}(t)}{\mathrm{d}t} = -\mathbf{a}(t)\begin{bmatrix} \nabla_{\S 1}\mathbf{f}(\mathbf{z}_t, t, \boldsymbol{\theta}) & \nabla_{\S 2}\mathbf{f}(\mathbf{z}_t, t, \boldsymbol{\theta}) & \nabla_{\S 3}\mathbf{f}(\mathbf{z}_t, t, \boldsymbol{\theta}) \\ \mathbf{0} & 0 & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} \end{bmatrix}$$

as required. □

¡¡¡¡¡¡¡ HEAD

- In a typical traning process, we start from $\mathbf{r}_{t_0} = (\mathbf{z}_{t_0}, t_0, \boldsymbol{\theta})$, and we solve the neural SDE forward in time to obtain $\mathbf{r}_{t_1} = (\mathbf{z}_{t_1}, t_1, \boldsymbol{\theta})$. We assume that we do not save any intermediate information in the forward solving process. Now, we need to compute the gradient $\mathbf{a}_{\S 3}(t_0) = \nabla_{\S 3}\mathcal{L}(\mathbf{z}_{t_0}, t_0, \boldsymbol{\theta})$.

=======

- In a typical traning process, we start from $\mathbf{r}_{t_0} = (\mathbf{z}_{t_0}, t_0, \boldsymbol{\theta})$, and we solve the neural SDE forward in time to obtain $\mathbf{r}_{t_1} = (\mathbf{z}_{t_1}, t_1, \boldsymbol{\theta})$. We assume that we do not save any intermediate information in the forward solving process. Now, we need to compute the gradient $\mathbf{a}_{\S 3}(t_0) = \nabla_{\S 3}\mathcal{L}(\mathbf{z}_{t_0}, t_0, \boldsymbol{\theta})$.

¿¿¿¿¿¿¿ 59f7aea97bc71b38541aa8ae8fede48971e0efd5

- The idea is then to start at time t_1 and jointly solve the following differential equations backward in time to t_0:

$$\begin{aligned} \frac{\mathrm{d}\mathbf{z}_t}{\mathrm{d}t} &= \mathbf{f}(\mathbf{z}_t, t, \boldsymbol{\theta}), \\ \frac{\mathrm{d}\mathbf{a}_{\S 1}(t)}{\mathrm{d}t} &= -\mathbf{a}_{\S 1}(t)\nabla_{\S 1}\mathbf{f}(\mathbf{z}_t, t, \boldsymbol{\theta}), \\ \frac{\mathrm{d}\mathbf{a}_{\S 3}(t)}{\mathrm{d}t} &= -\mathbf{a}_{\S 1}(t)\nabla_{\S 3}\mathbf{f}(\mathbf{z}_t, t, \boldsymbol{\theta}). \end{aligned}$$

In other words, we would like to compute the following integrals:

$$\begin{aligned} \mathbf{z}_{t_0} &= \mathbf{z}_{t_1} + \int_{t_1}^{t_0} \mathbf{f}(\mathbf{z}_t, t, \boldsymbol{\theta})\,\mathrm{d}t, \\ \mathbf{a}_{\S 1}(t_0) &= \mathbf{a}_{\S 1}(t_1) - \int_{t_1}^{t_0} \mathbf{a}_{\S 1}(\mathbf{z}_t, t, \boldsymbol{\theta})\nabla_{\S 1}\mathbf{f}(\mathbf{z}_t, t, \boldsymbol{\theta})\,\mathrm{d}t, \\ \mathbf{a}_{\S 3}(t_0) &= \mathbf{a}_{\S 3}(t_1) - \int_{t_1}^{t_0} \mathbf{a}_{\S 1}(\mathbf{z}_t, t, \boldsymbol{\theta})\nabla_{\S 3}\mathbf{f}(\mathbf{z}_t, t, \boldsymbol{\theta})\,\mathrm{d}t. \end{aligned}$$

The initial conditions include $\mathbf{z}_{t_1}$, which we just computed using the forward process. The other initial conditions are:

$$\begin{aligned} a_{\S 1}(t_1) &= \nabla_{\S 1}\mathcal{L}(\mathbf{z}_{t_1}, t_1, \boldsymbol{\theta}) = \nabla_{\S 1}L(\mathbf{z}_{t_1}, t_1, \boldsymbol{\theta}) = \nabla L(\mathbf{z}_{t_1}), \\ a_{\S 3}(t_1) &= \nabla_{\S 3}\mathcal{L}(\mathbf{z}_{t_1}, t_1, \boldsymbol{\theta}) = \nabla_{\S 3}L(\mathbf{z}_{t_1}, t_1, \boldsymbol{\theta}) = \mathbf{0}. \end{aligned}$$

The last line follows from the fact that we assumed that L does not depend on $\boldsymbol{\theta}$. All of these values are easy to compute.

- To solve the ODEs, we can use any black-box ODE solver. The interface for such a solver requires us to provide (1) an initial state vector, and (2) a function that computes the time derivative of the state vector given the time and the state vector. ¡¡¡¡¡¡¡ HEAD

  =======

  ¿¿¿¿¿¿¿ 59f7aea97bc71b38541aa8ae8fede48971e0efd5 Here, our state vector would be $\mathbf{q}^{(t)} \in \mathbb{R}^{n+n+m}$. It would be divided into three blocks $\mathbf{q}^{(t)} = (\mathbf{q}^{(t)}_{\S 1}, \mathbf{q}^{(t)}_{\S 2}, \mathbf{q}^{(t)}_{\S 3})$, and the blocks would correspond to $\mathbf{z}_t$, $\mathbf{a}_{\S 1}(t)^T$, and $\mathbf{a}_{\S 3}(t)^T$, respectively. The initial state vector would be

$$\mathbf{q}^{(t_1)} = \begin{bmatrix} \mathbf{z}_{t_1} \\ \nabla\big(L(\mathbf{z}_{t_1})\big)^T \\ \mathbf{0} \end{bmatrix}.$$

  The derivative would be given by

$$\frac{\mathrm{d}\mathbf{q}^{(t)}}{\mathrm{d}t} = \begin{bmatrix} \mathbf{f}(\mathbf{q}^{(t)}_{\S 1}, t, \boldsymbol{\theta}) \\ -\big(\mathbf{q}^{(t)}_{\S 2}\big)^T \nabla_{\S 1}\mathbf{f}(\mathbf{q}^{(t)}_{\S 1}, t, \boldsymbol{\theta}) \\ -\big(\mathbf{q}^{(t)}_{\S 2}\big)^T \nabla_{\S 3}\mathbf{f}(\mathbf{q}^{(t)}_{\S 1}, t, \boldsymbol{\theta}) \end{bmatrix}.$$

  Note that both $\big(\mathbf{q}^{(t)}_{\S 2}\big)^T \nabla_{\S 1}\mathbf{f}(\mathbf{q}^{(t)}_{\S 1}, t, \boldsymbol{\theta})$ and $\big(\mathbf{q}^{(t)}_{\S 2}\big)^T \nabla_{\S 3}\mathbf{f}(\mathbf{q}^{(t)}_{\S 1}, t, \boldsymbol{\theta})$ are both vector-Jacobian products (i.e., they are directional derivatives). They can thus be evaluated efficiently using automatic differentiation at the cost proportational to the evaluation of $\mathbf{f}(\mathbf{q}^{(t)}_{\S 1}, t, \boldsymbol{\theta})$.

- All in all, the adjoint sensitivity method allows us to compute the gradient without backpropagating through the operations of the forward solver. If we use forward-mode automatic differentiation, then the required memory is proportional to the size of the intermediate tensor vectors. There's no dependence on the network's depth at all. Hence, neural ODE is a very memory efficient architecture.

3 Continuous Normalizing Flows

- **Normalizing flows** refer to a body of techniques for modeling probability distributions that work by transforming a simple probability distribution (such as an isotropic Gaussian) to a more complicated one by compositing multiple simple transformations [KPB21].

- More concretely, we may start with $\mathbf{z}_0 \sim p(\mathbf{z}_0)$ where $p(\mathbf{z}_0)$ is simple. We can now make the probability distribution more complex by applying a bijective function $\mathbf{g}_1$ to get

$$\mathbf{z}_1 = \mathbf{g}_1(\mathbf{z}_0).$$

  We have that

$$p(\mathbf{z}_1) = p(\mathbf{z}_0)|\det \nabla \mathbf{g}_1(\mathbf{z}_0)|^{-1}$$

  or

$$\log p(\mathbf{z}_1) = \log p(\mathbf{z}_0) - \log|\det \nabla \mathbf{g}_1(\mathbf{z}_0)|.$$

- In most normalizing flow techniques, multiple transformations are used:

$$\mathbf{z}_k = (\mathbf{g}_k \circ \mathbf{g}_{k-1} \circ \dots \circ \mathbf{g}_2 \circ \mathbf{g}_1)(\mathbf{z}_0) = \mathbf{g}_k(\mathbf{g}_{k-1}(\cdots \mathbf{g}_2(\mathbf{g}_1(\mathbf{z}_0)))),$$

which implies

$$\log p(\mathbf{z}_k) = \log p(\mathbf{z_0}) - \sum_{i=1}^{k} |\det \nabla \mathbf{g}_i(\mathbf{z}_{i-1})|. \tag{1}$$

This above expression allows us to (1) compute the probability, and (2) train the normalizing flow model with maximum likelihood.

- Normalizing flows can be casted into the neural ODE framework if we require that all transformations have the same form

  $$\mathbf{z}_{t+1} = \mathbf{g}_{t+1}(\mathbf{z}_t) = \mathbf{z}_t + \mathbf{f}(\mathbf{z}_t, t, \boldsymbol{\theta}).$$

  As usual, we take the limit as $t \leftarrow \infty$ to obtain

  $$\frac{\mathrm{d}\mathbf{z}(t)}{\mathrm{d}t} = \mathbf{f}(\mathbf{z}, t, \boldsymbol{\theta}),$$

  which gives us a continuous normalizing flow.

- To compute probability and to train our neural ODE model, we need an expression like (1). This is given by the following theorem.

  Theorem 2 (Instantataneous change of variables). *Let $\mathbf{z}(t)$ be a finite continuous random variable with probability $p(\mathbf{z}(t))$ dependent on time. Let $\mathrm{d}\mathbf{z}/\mathrm{d}t = \mathbf{f}(\mathbf{z}(t), t)$ be a differential equation governing the value of $\mathbf{z}$. Assuming that $\mathbf{f}$ is uninformly Lipschitz continuous in $\mathbf{z}$ and continuous in t. Then,*

  $$\frac{\mathrm{d}\log p(\mathbf{z}(t))}{\mathrm{d}t} = -\mathrm{tr}(\nabla_{\S 1}\mathbf{f}(\mathbf{z}(t), t)).$$

References

[CRBD18] Ricky T. Q. Chen, Yulia Rubanova, Jesse Bettencourt, and David K Duvenaud. Neural ordinary differential equations. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 31. Curran Associates, Inc., 2018.

[DKB14] Laurent Dinh, David Krueger, and Yoshua Bengio. Nice: Non-linear independent components estimation, 2014.

[HZRS15] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *CoRR*, abs/1512.03385, 2015.

[Khu22] Pramook Khungurn. Notations for multivariable derivatives. `https://pkhungurn.github.io/notes/notes/math/multivar-deriv-notations/multivar-deriv-notations.pdf`, 2022. Accessed: 2022-04-24.

[KPB21] Ivan Kobyzev, Simon J.D. Prince, and Marcus A. Brubaker. Normalizing flows: An introduction and review of current methods. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 43(11):3964–3979, nov 2021.

[RM15] Danilo Jimenez Rezende and Shakir Mohamed. Variational inference with normalizing flows, 2015.