What are Neural ODEs?

Theo Carr

15 April 2020

1 Introduction

“Neural Ordinary Differential Equations”, by Chen et al., won the best paper award at NeurIPs in 2018 [1]. The paper introduces “a new family of neural networks”, similar to residual networks, which use an ordinary differential equation (ODE) solver in the forward pass. While other researchers have previously proposed this idea, Chen et al. demonstrate how to efficiently backpropagate through the network without actually backpropagating through the operations of the ODE solver.

The purpose of this article is to explain the paper to a broader audience. The content is based on the original paper by Chen et al., and in many places (especially the proofs for deriving the adjoint) this article closely follows the paper.

To supplement this explanation, I have implemented a basic ODE-net classifier for MNIST (without the adjoint method), and attempted to reproduce the results shown from the first three rows of Table I in the Neural ODEs paper. I have also implemented a version of the adjoint method, and demonstrate how to compute the gradient of a scalar valued loss with respect to the parameters of an ODE without backpropagating through the ODE solver. All code is available at **https://github.com/ktcarr/neural-ode**.

To understand Neural ODEs (and the motivation for using them), it is helpful to have some knowledge of ODE solvers (Section 2) and residual networks (Section 3). To skip to ODE networks, please go to Section 4.

2 Ordinary Differential Equation (ODE) solvers

For the purpose of this blog post, ordinary differential equations are equations of the form:

$$\frac{dy}{dt} = f(t, y)$$

That is, the derivative of a variable y with respect to t (representing time in this case) is a function of both t and y.

In many cases, we would like to find $y(t)$, which is the variable y as a function of time. In some cases it is possible to solve for $y(t)$ analytically, but in many cases it is not. Mathematicians have developed numerous effective numerical ODE solvers which approximate the solution to a differential equation. The simplest ODE solver, called Euler’s method, works by evaluating $\frac{dy}{dt} = f(t, y)$ at the initial condition to obtain the “slope” of $y(t)$ at t_0: $m = f(t_0, y_0)$. Then, the next value of $y(t)$ is estimated as $y(t_0 + h) = y(t_0) + h \times m$, where h is the desired step size. This function evaluation and update step is repeated until reaching a desired time, T. This approach is formalized in Algorithm 1, and the update process is visualized in Figure 1.

Algorithm 1: Euler method

```
Result: $y(T)$
Input: differential equation $\frac{dy}{dt} = f(t, y)$, initial condition $y(t_0)$, step size $h$, end time $T$;
Set $t = t_0$, $y = y_0$;
while $t \neq T$ do
    $m = f(t, y)$;
    $y = y + (m \times h)$;
    $t = t + h$;
end
return $y$
```

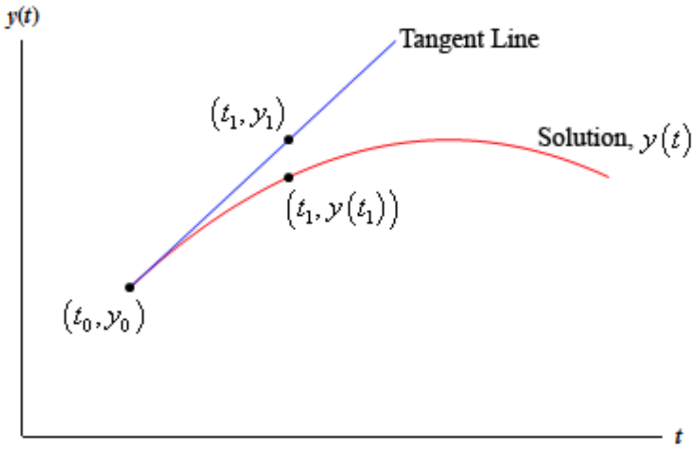


Figure 1: Euler's method. From Paul's Online Math Notes [2].

As an example of Euler's method, consider the following ODE:

$$\frac{dy}{dt} = y - \frac{1}{2}e^{\frac{t}{2}}\sin(5t) + 5e^{\frac{t}{2}}\cos(5t),\ \ y(0) = 0$$

It is possible to solve for the exact solution $y(t)$, which we will use as a reference to evaluate Euler's method. The solution is:

$$y(t) = e^{\frac{t}{2}}\sin(5t)$$

The results of applying Euler's method and another ODE solver, Runge-Kutta, are shown in Figure 2. As shown in the figure, the Runge-Kutta solver performs much better than Euler's method, even with a larger step size.

3 Residual Networks

Residual networks are a class of neural networks proposed in "Deep Residual Learning for Image Recognition", by Kaiming He et al., in 2015 [3]. Residual networks, or "ResNets", sought to address an apparent paradox with standard feed-forward and convolution neural networks: while adding more layers to the network initially appears to increase performance, at a certain point adding more layers starts to *decrease* performance. In theory, this shouldn't happen: a model with $n + 1$ layers can be

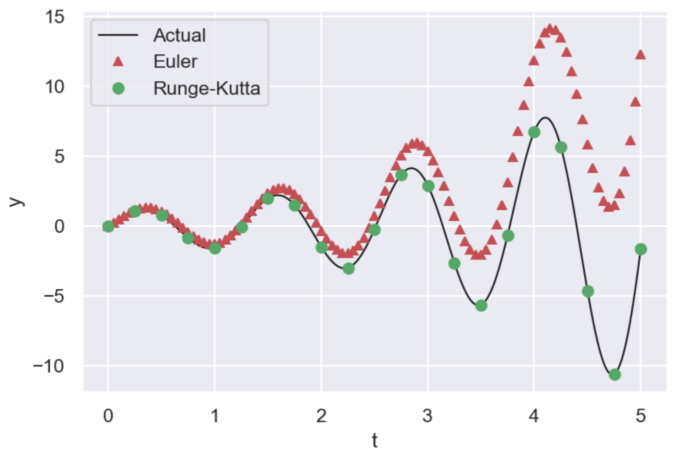


Figure 2: Numerical solutions to $\frac{dy}{dt} = y - \frac{1}{2}e^{\frac{t}{2}}\sin(5t) + 5e^{\frac{t}{2}}\cos(5t),\ y(0) = 0$.

set equivalent to a layer with n layers by setting one of the layers in the larger network to an identity transformation. However, in practice, there appears to be a maximum number of layers before performance starts to degrade.

Residual networks are built from "residual blocks" which closely resemble standard feed-forward networks: an input $\mathbf{x}$ is passed through one or multiple weight layers with corresponding non-linear activations. However, after the final layer in the residual block, the output $\mathcal{F}(x)$ is added back to the original input. An example of a residual block is shown in Figure 3.

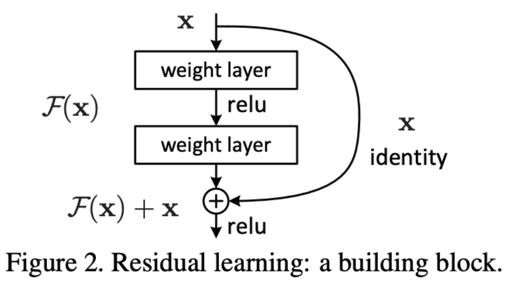

$\mathbf{x}$
weight layer
$\mathcal{F}(\mathbf{x})$
relu
weight layer
$\mathbf{x}$
identity
$\mathcal{F}(\mathbf{x}) + \mathbf{x}$
relu
Figure 2. Residual learning: a building block.

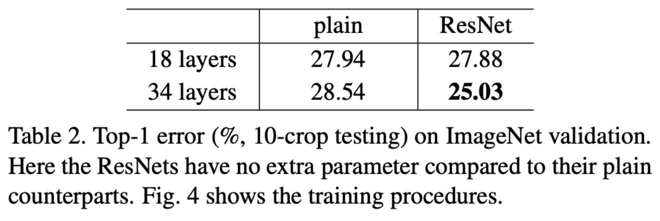
Figure 3: Example of a residual block, from [3].

ResNets are constructed by stacking these residual blocks together. He et al. showed that using these building blocks allows for deeper neural models (on the order of 100-1000 layers). Figure 4 demonstrates the improvement over plain neural networks, from [3].

4 ResNet to ODE-Net

A residual block looks very similar to a single step in Euler's method. Given an initial hidden state x_i, the residual block computes the next hidden state as $x_{i+1} = x_i + \mathcal{F}(x_i)$. If $\mathcal{F}(x) = \frac{dx}{dt}$, then this is a single iteration of Euler's method with a step size of 1.

For a more concrete example, consider the stack of three residual blocks shown in Figure 5. Let the input to the first residual block be h_t. Then we can think of this stack as three steps of the Euler



	plain	ResNet
18 layers	27.94	27.88
34 layers	28.54	**25.03**

Table 2. Top-1 error (%, 10-crop testing) on ImageNet validation. Here the ResNets have no extra parameter compared to their plain counterparts. Fig. 4 shows the training procedures.

Figure 4: ResNet performance gains for deeper networks (from [3]).

method, where each step is given by $h_{t+1} = h_t + f(h_t, \theta_t)$, and $f(h_t, \theta_t)$, representing $\frac{dh}{dt}$, is a pass through two convolutional layers.

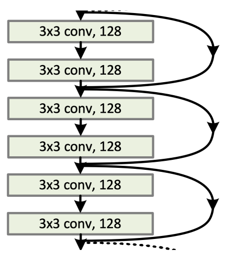


Figure 5: Stack of three residual blocks.

This is where the ODE Network comes in. Rather than specifying a set of discrete transformations (by choosing the number and arrangement of residual blocks), we can just define a function representing $\frac{dh}{dt}$, as in Figure 6. Then we can use an ODE solver to do a forward pass through our "network". We could then recreate the stack of residual blocks in Figure 5 by specifying $\frac{dh}{dt}$ as a pass through two convolutional layers, and using Euler's method as our ODE solver, with a step size of 1 and three steps. However, we could also generalize this approach by using a more advanced ODE solver with a smaller step size, or with adaptive step sizes.

$$\frac{d\mathbf{h}(t)}{dt} = f(\mathbf{h}(t), t, \theta) \quad =$$

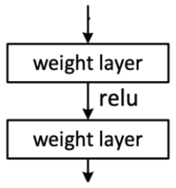


Figure 6: Parameterization of $\frac{dh}{dt}$

5 The adjoint (and parameter updates)

The forward pass through an ODE-network is straightforward. For the backward pass, we could just use standard autodiff libraries such as Pytorch. However, backpropagating through the network means backpropagating through the operations of the ODE-solver. For a solver such as the Euler method, this is similar to backpropagating through a ResNet, but we would like to use more complex ODE solvers, which may not allow for efficient backpropagation. To overcome this potential bottleneck, Chen et al. propose using the "adjoint method".

The adjoint state at time t is defined as $\mathbf{a}(t) = \frac{dL}{d\mathbf{z}(t)}$, where L is the loss computed after the forward pass, and $\mathbf{z}(t)$ is the hidden state of the network at time t. Knowing the adjoint state at every time step will allow us to compute the gradient of the loss with respect to the parameters.

The authors show that the derivative of the adjoint with respect to time is:

$$\frac{d\mathbf{a}(t)}{dt} = -\mathbf{a}(t)\frac{\partial f(\mathbf{z}(t), \theta, t)}{\partial \mathbf{z}(t)} \tag{1}$$

Where $f(\mathbf{z}(t), \theta, t)$ represents $\frac{d\mathbf{z}(t)}{dt}$, the derivative of the hidden state with respect to time (an example of such a function is shown in Figure 6). On the right side of the equation, $\mathbf{a}(t)$ is the adjoint state at the current time, and $\frac{\partial f(\mathbf{z}(t),\theta,t)}{\partial \mathbf{z}(t)}$ is the partial derivative of f with respect to the hidden state, $\mathbf{z}(t)$.

To evaluate $\frac{d\mathbf{a}(t)}{dt}$, both the adjoint state $\mathbf{a}(t)$ and hidden state $\mathbf{z}(t)$ are required. However, if these states are known, the expression can be efficiently evaluated using standard autodiff libraries. In practice, this means evaluating $f(\mathbf{z}(t), \theta, t)$, and computing the gradient with respect to $\mathbf{z}(t)$ (the result is called the Jacobian). Then multiply $\mathbf{a}(t)$ by the Jacobian (this operation is called the vector-Jacobian product).

The punchline of this mathematical setup is that we can use the same ODE solver from the forward pass to solve for the adjoint state at all time steps. To understand this, consider the result of a forward pass: we have the final hidden state $\mathbf{z}(T)$ and the loss L. Then we can compute $\mathbf{a}(T)$ as $\frac{dL}{d\mathbf{z}(T)}$, without backpropagating through the solver. Now we have both $\mathbf{a}(T)$ and $\mathbf{z}(T)$, and can compute $\frac{d\mathbf{a}(T)}{dt}$. Using our ODE solver, we can now proceed to the next time step, as we have an initial condition (i.e. $\mathbf{a}(t)$ and $\mathbf{z}(t)$), and an ODE ($\frac{d\mathbf{a}(t)}{dt}$).

What about updating the parameters? It turns out that the gradient of the loss with respect to the parameters can be computed as:

$$\frac{dL}{d\theta} = -\int_{T}^{t_0} \mathbf{a}(t)\frac{\partial f(\mathbf{z}(t), \theta, t)}{\partial \theta}dt \tag{2}$$

6 Explaining the adjoint

While implementing equations 1 and 2 is relatively straightforward (more on that later), you may be wondering how they are derived. The authors of the paper provide a nice explanation in Appendix B, which I will attempt to relay here.

First, we'll start with the derivative of the adjoint with respect to time (equation 1), then cover the gradient of the loss with respect to the parameters (equation 2).

6.1 Deriving $\frac{d\mathbf{a}(t)}{dt}$

Like backpropagation, the adjoint method is based on the chain rule. For backpropagation, if we would like to compute the gradient of the loss with respect to a hidden state $\mathbf{h}_t$, we apply the chain rule:

$$\frac{dL}{d\mathbf{h}_t} = \frac{dL}{d\mathbf{h}_{t+1}}\frac{d\mathbf{h}_{t+1}}{d\mathbf{h}_t}$$

For a continuous hidden state $\mathbf{z}(t)$ (as we treat the hidden state in Neural ODEs), we can write the chain rule as:

$$\frac{dL}{d\mathbf{z}(t)} = \frac{dL}{d\mathbf{z}(t+\epsilon)}\frac{d\mathbf{z}(t+\epsilon)}{d\mathbf{z}(t)} \tag{3}$$

Recalling that the adjoint is defined as $\mathbf{a}(t)$ is defined as $\mathbf{a}(t) = \frac{dL}{d\mathbf{z}(t)}$, we can rewrite equation 3 as:

$$\mathbf{a}(t) = \mathbf{a}(t+\epsilon)\frac{d\mathbf{z}(t+\epsilon)}{d\mathbf{z}(t)} \tag{4}$$

Now, consider the definition of the derivative:

$$\frac{df(t)}{dt} = \lim_{\epsilon\to 0^+}\frac{f(t+\epsilon)-f(t)}{\epsilon}$$

Substituting $\mathbf{a}(t)$, we have:

$$\frac{d\mathbf{a}(t)}{dt} = \lim_{\epsilon\to 0^+}\frac{\mathbf{a}(t+\epsilon)-\mathbf{a}(t)}{\epsilon}$$

Substituting equation 4 for $\mathbf{a}(t)$, we have:

$$\frac{d\mathbf{a}(t)}{dt} = \lim_{\epsilon\to 0^+}\frac{\mathbf{a}(t+\epsilon)-\mathbf{a}(t+\epsilon)\frac{d\mathbf{z}(t+\epsilon)}{d\mathbf{z}(t)}}{\epsilon} \tag{5}$$

Now consider $\mathbf{z}(t+\epsilon)$. Applying Taylor's theorem, we can write $\mathbf{z}(t+\epsilon)$ as:

$$\begin{aligned}\mathbf{z}(t+\epsilon) &= \mathbf{z}(t) + \epsilon\mathbf{z}'(t) + \epsilon^2\frac{\mathbf{z}''(t)}{2} + \cdots + \epsilon^n\frac{\mathbf{z}^{(n)}(t)}{n!} + \dots && (6)\\ &= \mathbf{z}(t) + \epsilon\mathbf{z}'(t) + \mathcal{O}(\epsilon^2) && (7)\end{aligned}$$

$\mathbf{z}'(t) = \frac{d\mathbf{z}(t)}{dt}$, and $\frac{d\mathbf{z}(t)}{dt}$ is defined as $f(\mathbf{z}(t),\theta,t)$, so:

$$\mathbf{z}(t+\epsilon) = \mathbf{z}(t) + \epsilon f(\mathbf{z}(t),\theta,t) + \mathcal{O}(\epsilon^2) \tag{8}$$

To derive $\frac{d\mathbf{a}(t)}{dt}$, the rest is just algebra. Substituting equation 8 into equation 5, we have:

$$\begin{aligned}\frac{d\mathbf{a}(t)}{dt} &= \lim_{\epsilon\to 0^+}\frac{\mathbf{a}(t+\epsilon)-\mathbf{a}(t+\epsilon)\frac{\partial}{\partial\mathbf{z}(t)}\Bigg(\mathbf{z}(t)+\epsilon f(\mathbf{z}(t),\theta,t)+\mathcal{O}(\epsilon^2)\Bigg)}{\epsilon} && (9)\\ &= \lim_{\epsilon\to 0^+}\frac{\mathbf{a}(t+\epsilon)-\mathbf{a}(t+\epsilon)(I+\epsilon\frac{\partial f}{\partial\mathbf{z}(t)}+\mathcal{O}(\epsilon^2))}{\epsilon} && (10)\\ &= \lim_{\epsilon\to 0^+}\frac{\mathbf{a}(t+\epsilon)-\mathbf{a}(t+\epsilon)-\epsilon\mathbf{a}(t+\epsilon)\frac{\partial f}{\partial\mathbf{z}(t)}-\mathcal{O}(\epsilon^2)\mathbf{a}(t+\epsilon)}{\epsilon} && (11)\\ &= \lim_{\epsilon\to 0^+}\frac{-\epsilon\mathbf{a}(t+\epsilon)\frac{\partial f}{\partial\mathbf{z}(t)}-\mathcal{O}(\epsilon^2)\mathbf{a}(t+\epsilon)}{\epsilon} && (12)\\ &= \lim_{\epsilon\to 0^+}\frac{-\epsilon\Bigg(\mathbf{a}(t+\epsilon)\frac{\partial f}{\partial\mathbf{z}(t)}-\mathcal{O}(\epsilon)\mathbf{a}(t+\epsilon)\Bigg)}{\epsilon} && (13)\\ &= \lim_{\epsilon\to 0^+}\Bigg(-\mathbf{a}(t+\epsilon)\frac{\partial f}{\partial\mathbf{z}(t)}+\mathcal{O}(\epsilon)\mathbf{a}(t+\epsilon)\Bigg) && (14)\\ &= -\mathbf{a}(t)\frac{\partial f}{\partial\mathbf{z}(t)} && (15)\end{aligned}$$

Note that this proof is taken from Appendix B of the Neural ODEs paper; this version just has slightly more of the intermediate (algebraic) steps shown.

6.2 Deriving $\frac{dL}{d\theta}$

Previously, we have just considered the hidden state $\mathbf{z}(t)$, and the adjoint state $\mathbf{a}(t) = \frac{dL}{d\mathbf{z}(t)}$. We can generalize the hidden state to form an augmented state which includes θ and t: $[\mathbf{z}, \theta, t](t)$. The augmented state is a function of time, but we will abbreviate it as $[\mathbf{z}, \theta, t]$. The derivative of the augmented state with respect to time is:

$$\begin{aligned} f_{\text{aug}}([\mathbf{z}, \theta, t]) &= \frac{d}{dt}[\mathbf{z}, \theta, t] && (16) \\ &= [\frac{d\mathbf{z}(t)}{dt}, \frac{d\theta(t)}{dt}, \frac{dt(t)}{dt}] && (17) \\ &= [f(\mathbf{z}, \theta, t), \mathbf{0}, 1] && (18) \end{aligned}$$

Note that the parameters θ are constant with respect to time, so $\frac{d\theta}{dt} = \mathbf{0}$. The derivative of time with respect to itself is $\frac{dt}{dt} = 1$.

Then we can compute the gradient of f_{aug} with respect to the augmented state, $[\mathbf{z}, \theta, t]$ to form the following Jacobian:

$$\frac{\partial f_{\text{aug}}}{d([\mathbf{z}, \theta, t])} = \begin{bmatrix} \frac{\partial f}{\partial z} & \frac{\partial f}{\partial \theta} & \frac{\partial f}{\partial t} \\ \frac{\partial \mathbf{0}}{\partial z} & \frac{\partial \mathbf{0}}{\partial \theta} & \frac{\partial \mathbf{0}}{\partial t} \\ \frac{\partial 1}{\partial z} & \frac{\partial 1}{\partial \theta} & \frac{\partial 1}{\partial t} \end{bmatrix} = \begin{bmatrix} \frac{\partial f}{\partial z} & \frac{\partial f}{\partial \theta} & \frac{\partial f}{\partial t} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} \end{bmatrix} \tag{19}$$

The adjoint, defined as $\mathbf{a}(t) = \frac{dL}{d\mathbf{z}(t)}$, represents the sensitivity of the loss with respect to the hidden state. We can also define adjoints for the sensitivity of the loss with respect to the parameters and to time:

$$\mathbf{a}_\theta(t) = \frac{dL}{d\theta(t)}, \ \mathbf{a}_t(t) = \frac{dL}{dt(t)}$$

Then the augmented adjoint state is:

$$[\mathbf{a}(t), \mathbf{a}_\theta(t), \mathbf{a}_t(t)] \tag{20}$$

Substituting equations 19 and 20 into equation 15, we have:

$$\begin{aligned} \frac{d\mathbf{a}_{\text{aug}}(t)}{dt} &= -[\mathbf{a}(t), \mathbf{a}_\theta(t), \mathbf{a}_t(t)] \begin{bmatrix} \frac{\partial f}{\partial z} & \frac{\partial f}{\partial \theta} & \frac{\partial f}{\partial t} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} \end{bmatrix} && (21) \\ &= -[\mathbf{a}(t)\frac{\partial f}{\partial z}, \mathbf{a}_\theta(t)\frac{\partial f}{\partial \theta}, \mathbf{a}_t(t)\frac{\partial f}{\partial t}] && (22) \end{aligned}$$

Unpacking equation 22, we have:

$$\begin{aligned} \frac{d\mathbf{a}(t)}{dt} &= -\mathbf{a}(t)\frac{\partial f}{\partial \mathbf{z}} && (23) \\ \frac{d\mathbf{a}_\theta(t)}{dt} &= -\mathbf{a}(t)\frac{\partial f}{\partial \theta} && (24) \\ \frac{d\mathbf{a}_t(t)}{dt} &= -\mathbf{a}(t)\frac{\partial f}{\partial t} && (25) \end{aligned}$$

We are interested in $\mathbf{a}_\theta = \frac{dL}{d\theta(t)} = \frac{dL}{d\theta}$. We have $\frac{d\mathbf{a}_\theta(t)}{dt} = -\mathbf{a}(t)\frac{\partial f}{\partial \theta}$, so:

$$\mathbf{a}_\theta(t) = \int_T^{t_0} -\mathbf{a}(t)\frac{\partial f}{\partial \theta}dt = \int_T^{t_0} -\mathbf{a}(t)\frac{\partial f(\mathbf{z}(t), \theta, t)}{\partial \theta}dt$$

Therefore,

$$\frac{dL}{d\theta} = \int_T^{t_0} -\mathbf{a}(t)\frac{\partial f(\mathbf{z}(t),\theta,t)}{\partial\theta}dt \tag{26}$$

7 Implementation

The authors of Neural ODEs released *torchdiffeq* [4], a "Pytorch implementation of ODE solvers" (available at https://github.com/rtqichen/torchdiffeq). For a pared down example of how to train an ODE-net on MNIST, see https://github.com/ktcarr/neural-ode/blob/master/**mnist_results.ipynb**. Note that for this example, I do not implement the adjoint method, but backpropagate directly through the ODE-net (this corresponds to the "RK-Net" in Table I of the Neural ODEs paper). To compare direct backpropagation to the adjoint method, I have also the adjoint method from the *torchdiffeq* package.

My results on MNIST are shown in Table 1. Note that the ODE-Net, which uses the adjoint method, takes about twice as long as the RK-Net, which backpropagates directly through the ODE solver. Both RK-Net and the ODE-Net use a Runge-Kutta solver with fixed step size of .5 on the time interval $[0, 2]$, corresponding to 4 "steps" for the solver for the forward pass. In Chen et al., the authors do not provide the time statistics, but say that training time for RK-Net and ODE-Net should be on the order of $\mathcal{O}(\tilde{L})$, where $\tilde{L}$ is the number of function evaluations.

Model	Error (%)		Time (s)	
	μ	σ	μ	σ
1-layer MLP	2.21	0.07	176	1.5
ResNet	0.51	0.05	725	14.5
RK-Net	0.54	0.03	1121	9.3
ODE-Net*	0.56	0.03	2081	66.7

Table 1: Results on MNIST, averaged over 3 trials. "Time" is time to train 20 epochs. "Error" is minimum error achieved on validation set during training. Compare to Table I in Chen et al. 2018. *ODE-Net uses adjoint method from *torchdiffeq* package.

For a pared down implementation of the adjoint method, see https://github.com/ktcarr/neural-ode/blob/master/**adjoint.ipynb**. In this notebook, I show how to compute the gradient for an ODE solver (in this case, Runge-Kutta), and update parameters, without backpropagating through the solver. The gradients from this custom adjoint method are compared to those obtained from direct backpropagation, and to those obtained using the *torchdiffeq* package.

Going forwards, I would like to integrate my custom adjoint method with the ODE-Net used for MNIST classification. For some context, the ODE-Net contains several downsampling layers, then the output is passed through an ODE solver, before being passed through several fully-connected layers. My current implementation of the adjoint method cannot be "dropped in" to a network with other layers. Currently, it computes the adjoint $a(t)$ as the gradient of a scalar-valued loss with respect to the final hidden state (i.e. the output of the ODE solver). To make it compatible with fully connected layers coming after the solver, I think the adjoint will become a Jacobian, representing the gradient of the z_{T+1} layer with respect to the z_T layer (i.e. $\frac{\partial z_{T+1}}{\partial z_T}$), where z_T is the final hidden layer for the ODE solver. To compute the gradient of the loss with respect to the parameters of earlier layers, I will have to pass the adjoint of the first hidden state, $a(t_0) = \frac{\partial L}{\partial z_0}$, to the earlier layers.

References

[1] Ricky Chen, Yulia Rubanova, Jesse Bettencourt, and David Duvenaud. Neural ordinary differential equations. *Conference on Neural Information Processing Systems*, 2018.

[2] Paul Dawkins. Paul's online math notes.

[3] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *Conference on Computer Vision and Pattern Recognition*, 2016.

[4] Ricky Chen et al. torchdiffeq: Differentiable ode solvers with full gpu support and o(1)-memory backpropagation.