Neural Ordinary Differential Equations: Review of Literature

Owen Capell, Evan Hyzer, and Malhar Vaishampayan

University of Pittsburgh

April 2023

1 Introduction

In this paper, we discuss the state of *neural ordinary differential equations* (hereby referred to as *Neural ODEs*) in modern machine learning practices through a review of contemporary literature on the subject. We will make the assumption the reader has some familiarity with basic neural network architectures. However, will discuss the theory and motivation behind two popular variants (*convolutional neural networks* (CNNs) and *residual neural networks* (ResNET), a subset of CNNs), due to the fact (as will be discussed) that one application for Neural ODEs is replacing the hidden layers of ResNET implementation.

We attempt to thoroughly develop the theory presented in the literature, supplying the proofs where necessary. It may be beneficial to the reader to consider this paper an overview of the prerequisites to understand Neural ODEs, as well as an overview of the contemporary state of Neural ODEs.

2 Convolutional Neural Networks (CNNs)

2.1 Convolutions

We begin our discussion on convolutional neural networks with a brief overview of convolutions, both of the continuous and discrete variety (the latter of which is the basis of CNNs)

2.2 Convolutional Neural Networks

Now, we will discuss the motivation for and design of a convolutional architecture. At times we will include figures from Michael Nielsen's "Neural Networks and Deep Learning" [Nie15].

One of the primary motivations for the construction of a convolutional neural network is the desire to achieve "deep learning" (a loosely defined term that we will take to be any neural network with multiple hidden layers) with reasonable speed.

In general, neural networks comprised only of fully-connected layers (layers of neurons who receieve inputs from all neurons of the left-adjacent layer and provide inputs to all neurons of right-adjacent layers) will suffer from the *vanishing/exploding gradient problem*. (The reader should recall that *reverse-mode differentiation* or *backpropogation* is performed on the chosen cost function C to determine the partial derivatives with respect to all parameters θ of the network, in an attempt to minimize said function). This problem causes significant slowdowns in the speed of learning for some layers due to under/over-adjustment of weights/biases. It is due to back propagation either sending the gradient to near-zero in the early layers, or the gradient ballooning to artificially high values. In general, the gradient is *unstable* [BSF94].

An effective method to circumvent the problem of vanishing or exploding gradients is to maintain the potential efficacy of a deep neural network while

eliminating some parameters.

We choose to examine CNNs through the lens of this computational problem because of the fact that CNNs exhibit high levels of accuracy in this area and are the industry-standard for image classification [CMS12].

2.2.1 The Image Classification Problem

Discussing the motivation behind machine learning models through the lens of some computation problem (as opposed to in the abstract) tends to lend itself to the development of practical applications. As such, we approach the remainder of the development of CNNs through the lens of the image classification problem:

Given a set of images I and a finite collection of image categories J, map each image $i \in I$ to its appropriate classification $j \in J$.

Constructing a CNN to approach this problem follows the basic outline:

1. Develop *feature maps* using convolutional layers and pooling layers
2. Where advantageous, insert fully-connecting layers into the network
3. Define the dim J-dimensional output layer of fully-connected neurons
4. Train network using some learning algorithm A (such as stochastic gradient descent) using backpropogation

We will now discuss the construction of convolutional layers and pooling layers. A brief overview of backpropogation is given, however given the assumed prerequisite knowledge of neural network construction (and the similarity that backpropagation of a CNN has to any other traditional architecture, such as a *multi-layer perceptron* or *MLP*), we do not develop the algorithm thoroughly.

2.3 Convolutional Layers

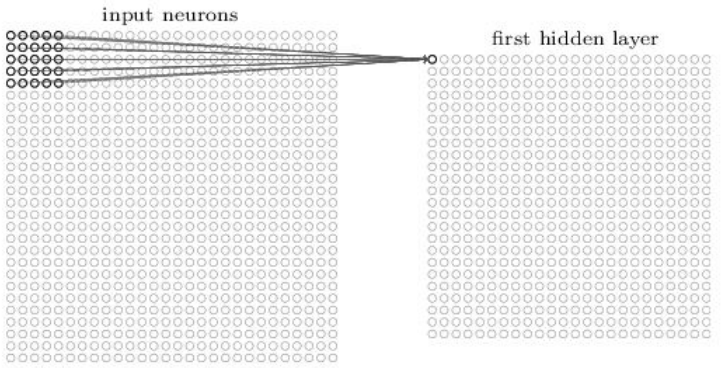
We define a *convolutional layer* of a CNN as $C(\alpha_{i\times j}, \beta_{k\times l}, f)$

Let's examine these parameters further. Our first parameter, α, is a 2-tuple specifying the dimensions of the input into the layer. If we assume our input layer is the image we are trying to classify itself, then α simply corresponds to the dimensions of the image.

The following parameter, β represents the dimensions of the *local receptive field* for the layer. This field will "scan" the input image with a given *stride length* (for simplicity's sake, we assume a stride length of 1 throughout this paper), and perform a convolution operation on the pixels within the field. This convolution operation defines the activation of a single hidden neuron in the convolutional layer.

The implication of this method to determine hidden neuron activation tell us that each hidden neuron in a convolutional layer has $\beta = k \times l$ weights w, and a bias b. The resulting layer of neurons from performing these convolution operations across the entire image defines a *feature map*, where the idea is the learned parameters (weights and biases) will be trained to examine a feature of the image, such as edges or curves.

Figure 1: An example of the convolutional operation performed to produce the activation of the first hidden neuron in the convolutional layer [Nie15]



The weights of the convolution (which are sometimes referred to as comprising the *kernel*) as well as the bias are all shared among the feature map. This significantly reduces the number of parameters the network needs to learn, and ensures the feature map successfully captures a specific aspect of the image. As such, we can define the activation of the hidden neuron (here we assume a $5x5$ local receptive field) in row j, column k by

$$b + \sum_{d=0}^{4} \sum_{m=0}^{4} w_{d,m} a_{j+d,k+m} \tag{1}$$

where we omit an activation function, let b be the shared bias, $w_{d,m}$ be the 5×5 matrix of shared learned weights, and $a_{x,y}$ be the input activation at position x, y [Nie15].

Our last parameter, f defines the number of *disjoint* feature maps that the convolutional layer should have. The idea behind this is that we want a single convolutional layer to detect many features of an image (say, top edges, side edges, et cetra). These maps are disjoint, that is, they all contain different (still shared within the maps themselves) weights and biases to learn. All feature maps are then mapped into the next layer of neurons, which we now discuss.

2.4 Pooling Layers

We define a pooling layer to immediately succeed and take a convolutional layer C as input as $P(\alpha_{n \times i \times j}, \beta_{k \times l}, p)$.

Similar to that of convolutional layers, our first parameter, α is the dimensions of the input (this time though, our input is from a convolutional layer, which, as stated above, will produce some number m of feature maps, all of dimension $i \times j$. As such, we include this number in our calculation of dimension as n, for the number of feature maps).

Our next parameter, β is similar to that of a convolution layer, where we're

defining the dimensions of the local receptive field to scan the output of the convolutional layer. This speaks towards the goal of a pooling layer: eliminating noise from the convolutional layer. It does this by, similar to a convolutional layer, performing a convolution operation on the output of a convolutional layer to simplify the output.

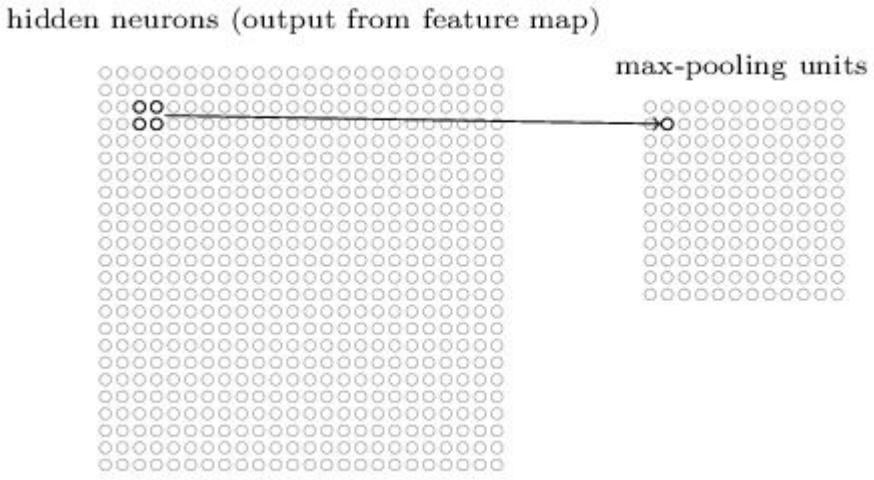


Figure 2: Example of developing a pooling layer from the output of a convolutional layer [Nie15]

This gives us our last parameter: p, which specifies the pooling function. A common choice is *max-pool*, which selects the maximum of the activations of the input neurons, given the local receptive field (as seen in the figure).

2.5 Putting it All Together

Finally, as prescribed above, a CNN is constructed by feeding the input image to a convolutional layer, which itself feeds to a pooling layer, followed by a choice of variable insertions of fully-connected layers or convolutional-pooling layer combinations, and concluding with an output layer of neurons corresponding to the network's classification of the image.

2.5.1 Backpropagation with CNNs

In order to successfully train a CNN, it must be possible to compute the gradient of the cost function (which is left as a black-box in this paper, as the choice doesn't provide relevance) with respect to each weight and bias, so as to update them in a manner that reduces the cost of training.

To motivate the development of backpropagation with CNNs, we reference and model the outline given in "Convolutions and Backpropagations" [Sol].

Suppose we have a simple function, f (we will later treat this as a convolution operation in a convolutional layer), which takes two inputs x and y, and outputs a single value z.

We let our loss function be L and recognize our goal is to compute $\frac{\partial L}{\partial x}$ and $\frac{\partial L}{\partial y}$, by backpropagating $\frac{\partial L}{\partial z}$ through f. (We assume we have the latter quantity).

We note, by the chain rule, that

$$\frac{\partial L}{\partial x} = \frac{\partial L}{\partial z}\frac{\partial z}{\partial x} \tag{2}$$

$$\frac{\partial L}{\partial y} = \frac{\partial L}{\partial z}\frac{\partial z}{\partial y} \tag{3}$$

Now we let f be a convolution of size 2×2 defined by the matrix F and let its input $x = X$ be a 3×3 matrix. The application of f on X is a 2×2 matrix O. As such, our local gradient for the input $x = X$ goes from $\frac{\partial L}{\partial x}$ to $\frac{\partial L}{\partial O}$.

We can then derive the partial derivative of L with respect to each element F_i with:

$$\frac{\partial L}{\partial F_i} = \sum_{k=(1,1)}^{(2,2)} \frac{\partial L}{\partial O_k} \cdot \frac{\partial O_k}{\partial F_i} \tag{4}$$

By noting F is a matrix operator depending on X to get output O, we

recognize that we can compute $\frac{\partial O}{\partial F}$:

$$\frac{\partial O_{i,j}}{\partial F_{i,j}} = X_{i,j} \tag{5}$$

and recognizing we can compute the partial derivatives for the rest of the entries of O in analogous fashion.

Now by by substituting (4) in (5), we see that $\frac{\partial L}{\partial F}$ can be defined as a convolution between the input matrix X and the loss gradient from the next layer, $\frac{\partial L}{\partial O}$.

We do not develop $\frac{\partial L}{\partial X}$ with the same rigour, and instead reiterate our reference [Sol] and note that it can be computed in near analogous fashion, through the definition of a convolution.

We do note, importantly, that both the forwards and backwards pass of an input through a convolutional layer is given by convolutional operations.

3 Residual Neural Networks (ResNET)

CNNs were a breakthrough in deep learning for image classification. But, there were limitations. Our intuition that the deeper (more layers) a network is, the more accurate it becomes, fails. Instead, traditional CNNs reach a depth level in which they perform worse [He+15]. This led to the development of *residual neural networks* or *ResNET*. The basis of ResNET is the adoption of *residual blocks*.

3.1 Universality of Neural Networks

Before we discuss the construction of residual blocks, we make an aside to an incredible important result that was proven in 1998 about neural networks.

Theorem 3.1. *All multilayered feedforward neural networks are universal*

approximators. That is, any feedforward network N can approximate a vector-valued function f to any desired degree of approximation.

The proof of this theorem involves high-level applications of analysis that go beyond the scope of this paper. Instead, we reference the paper in which this theorem was proved and accept the theorem as a fact henceforth [HSW89].

3.2 Motivation for Residual Neural Networks

Let's assume we have a CNN, and let it be X. Let's assume the average error (failed classifications) of X is e_X. Now, let's define a new, deeper CNN, Y. We define Y to be $X + I_n$, that is, all of the layers of X, followed by n successive identity mappings I. By definition, $Y = X$, and we should have the case that $e_Y \leq e_X$. However, in practice when we deepen a CNN to some arbitrary threshold, the average error *increases*, which is troublesome. We know a solution exists that is at least as good as our original network, so why doesn't our learning find that solution? It was hypothesized that a deeper network has trouble approximating such identity mappings [He+15] As such, a new construction of network architecture was presented to bring the previous layers closer to identity mapping as a precondition.

3.3 The Residual Block

Let us recall the image classification problem, and present it in a different light:

Given a set of images I and a finite collection of categories J, find a mapping $G : I \to H$ that maps the most $i \in I$ to the proper designation $j \in J$.

We abstract this further, by defining our ideal neural network to be (by reference to Theorem 3.2) an *approximator* to G. Furthermore, let this network be $\mathcal{F}$ and, assuming we have a finite collection of layers L comprising $\mathcal{F}$, let $\mathcal{F}_i$ be the mapping of the ith layer in $\mathcal{F}$.

Clearly, we want $\mathcal{F} \approx G$. That must mean that for each $\mathcal{F}_i$, $\exists\ \mathcal{H}_i$ that is the best mapping for $\mathcal{F}_i$ (that is, there is an ideal mapping we want $\mathcal{F}_i$ to be).

Typically, we let our network simply approximate each $\mathcal{H}_i$. However, as stated previously, this may rely on use of identity mapping which traditional deep CNNs strugle with. So, instead, we let our network approximate each $\mathcal{F}_i(x)$ as $\mathcal{F}_i(x) \equiv \mathcal{H}_i - x$, where $\mathcal{H}_i(x)$ is our ideal mapping.

As such, we now have a goal of computing the residual function $\mathcal{F}_i(x) + x$. We do this by defining a block (specifically, a *residual block*) of network layers designed to approximate $\mathcal{F}_i$, for each i, As such, by approximating that mapping and adding the input x as a residual, byway of an identity mapping, we approximate the exact same mapping $\mathcal{H}_i$ as previous. The hypothesis is that despite being the same approximation, providing the residual input x to a block creates a more conducive learning environment for the network.

Moreover, since the same mapping is being approximated, and identity mappings of an input x to the output of stacked layers can be done with the use of *shorcut connections* (connections mapping from one layer to another layer, skipping some number of existing layers between the connections), constructing a network with residual blocks adds no computational complexity nor parameters, meaning that it operates with the same run-time as similarly parametrized CNNs, and can be fairly compared with them [He+15].

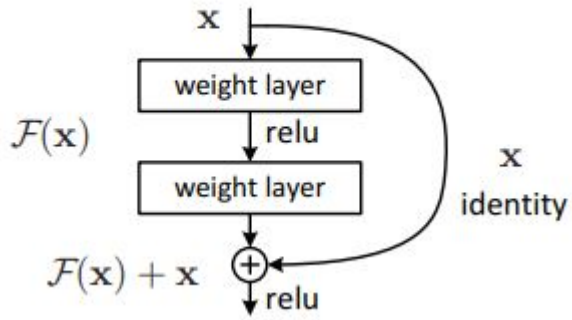


Figure 3: A hypothetical construction of a residual block for an input x [He+15]

For the sake of brevity, we do not further entertain the architecture of residual neural networks nor the precisce figures of their accuracy. Instead, we conclude by summarizing the results of the paper.

Firstly, it should be addressed that, since residual neural networks are designed to be an enhancement of CNNs, the layers comprising a residual block are typically convolutional layers. Given that convolutional layers change the dimensions of the input, it's very possible that x and $\mathcal{F}_i(x)$ have different dimensions. If this is the case, a computationally negligible linear projection W_s can be applied to the shortcut connection. This gives the following output function $\mathbf{y}$ for the residual block ([He+15]):

$$\mathbf{y} = \mathcal{F}_i(x, \{W_j\}) + W_s x \tag{6}$$

When networks were constructed with residual blocks, deeper networks exhibited an average error that was less than networks with less blocks (shallower). Constructed residual neural networks performed at or above the level of accuracy of other well-known networks on certain sets of image data [He+15].

In all, the development of residual neural networks further enhanced the deep learning capabilities of CNNs without adding additional computational complexity nor additional parameters. This makes such an architecture an attractive choice when constructing deep CNNs.

As a final note, we generalize to say that a *residual neural network* is any neural network (that is, not necessarily a CNN) that involves the chaining together of smaller neural networks and adding one network's input to its output [ZJ].

4 Neural Ordinary Differential Equations (Neural ODEs)

4.1 Introduction to Neural ODEs

Let's begin with a simple abstraction of residual neural networks. We view the network and being composed of a sequence of transformations to a hidden state:

$$\mathbf{h_{t+1}} = \mathbf{h_t} + f(\mathbf{h_t}, \theta_t) \tag{7}$$

where $t \in \{0, \dots T\}$ and $\mathbf{h}_t \in \mathbb{R}^D$ [Che+19]. It should be noted these updates to hidden states can be seen as an Euler discretization of a continuous function [Ee17].

In fact, we consider the following derivation:

$$\begin{aligned} \mathbf{h}_{t+1} &= f(\mathbf{h}_t) + \mathbf{h}_t && (8)\\ &= \frac{\Delta t}{\Delta t} f(\mathbf{h}_t) + \mathbf{h}_t && (9)\\ &= \Delta t G(\mathbf{h}_t) + \mathbf{h_t} && (10) \end{aligned}$$

where we now learn $G(\mathbf{h}_t) = f(\mathbf{h}_t)/\Delta t + \mathbf{h}_t$ ([Gib]).

As we see, this is now a single step of Euler's method for solving ordinary differential equations. As such, we now consider our neural network to, on a per-layer basis, perform a single-step of Euler's method. Subsequently, we can model our network with

$$\frac{d\mathbf{h}(t)}{dt} = G(\mathbf{h}_t, t, \theta) \tag{11}$$

which allows us to describe the output of the network as some state at time t_1, namely, $\mathbf{h}(t_1)$, with our initial state (input) being $\mathbf{h}(t_0)$. We have some number of parameters, θ, that will be our learned parameters.

We can determine the output of our network, $\mathbf{h}(t_1)$ in the following manner:

$$\mathbf{h}(t_1) = \text{ODESolve}(\mathbf{h}(t_0), G, t_0, t_1, \theta) \tag{12}$$

$$= \mathbf{h}(t_0) + \int_{t_0}^{t_1} G(\mathbf{h}(t), t, \theta)dt \tag{13}$$

by definition of an *initial-value problem (IVP)*, where we use any ODE Solver [Che+19][Gib].

We will now look at optimizing some cost function $L(\mathbf{h}(t_1)) = L(\text{ODESolve}(\mathbf{h}(t_0), G, t_0, t_1, \theta)$ by calculating its gradient, specifically with respect to the learned parameters θ, before discussing applications of Neural ODEs as well as potential limitations.

4.2 Optimizing Cost Function with Neural ODEs

4.2.1 The Adjoint Method

Here, we recount how "Fluid Control Using the Adjoint Method" ([McN+04]) calculates the gradient of many controlled parameters efficiently (almost verbatim), as the same method is used to optimize the loss function for our Neural ODE.

Suppose that a matrix A and vectors c and g are known. If we want to compute the vector product (in terms of an unknown vector, b)

$$g^T b \text{ such that } Ab = c$$

We can solve for b and compute the vector product, or introduce a new vector s and compute:

$$s^T c \text{ such that } A^T s = g$$

which is known as the *dual*. Then, we continue (by substitution):

$$s^T c = s^T Ab = (A^T s)^T b = g^T b$$

where we know consider the case where b and c are actually matrices, B and C, respectively, so that the vector-matrix product

$$g^T B \text{ such that } AB = C$$

is equivalent to

$$s^T C \text{ such that } A^T s = g$$

Now, we will begin gradient calculation.

Assume we have an initial state (fixed) q_0, which is evolved repeatedly into n subsequent states $q_1, \ldots, q_n$ according to:

$$q_{i+1} = f_i(q_i, u)$$

where each f_i is an arbitrary differentiable function parameterized by u. Now, define $Q = [q_1^T, \ldots q_n^T]^T$ and $F(Q, u) = [f_0(q_0, u)^T, \ldots f_{n-1}(q_{n-1}, u)^T]^T$. Now, we rewrite the above equation as

$$Q = F(Q, u)$$

Now, assume there is a differentiable function $\varphi(Q, u)$ for which we want to compute its derivative with respect to the vector u:

$$\frac{d\varphi}{du} = \frac{\partial \varphi}{\partial Q}\frac{dQ}{du} + \frac{\partial \varphi}{\partial u}$$

We now differentiate our equation for Q above, meaning the calculation for $d\varphi/du$ relies on

$$\frac{\partial \varphi}{\partial Q}\frac{dQ}{du} \text{ such that } (I - \frac{\partial F}{\partial Q})\frac{dQ}{du} = \frac{\partial F}{\partial u}$$

We now define a vector R (similar to s as above) and instead compute

$$R^T \frac{\partial F}{\partial u} \text{ such that } (I - \frac{\partial F}{\partial Q})^T R = \frac{\partial \varphi^T}{\partial Q}$$

We call R the adjoint vector, and calculating R allows us to compute:

$$\frac{d\varphi}{du} = R^T \frac{\partial F}{\partial q} + \frac{\partial \varphi}{\partial u}$$

which only leaves us to calculating R. First, let's rewrite the second-to-last equation above as

$$R = \left(\frac{\partial F}{\partial Q}\right) R + \frac{\partial \varphi^T}{\partial Q}$$

In the same respects that Q is viewed as a sequence of states $q_1, \ldots, q_n$, we can view R as a sequence of *adjoint* states $r_1, \ldots r_n$. The above implies that $r_n = (\partial\varphi/\partial q_n)^T$ and

$$r_i = \left(\frac{\partial f_i}{\partial q_i}\right)^T r_{i+1} + \left(\frac{\partial \varphi}{\partial q_i}\right)^T$$

where each adjoint state depends on the subsequent state.

Having established this method, we now explore calculating the gradient of a cost function L with respect to some learned parameters θ of a Neural ODE based off this method.

4.2.2 Optimizing a Cost Function L of the Neural ODE

We now review the method to optimize any scalar-valued cost function L whose input is the result of an ODE solver, as discussed in "Neural Ordinary Differential Equations" ([Che+19]).

Consider such a function L:

$$L(\mathbf{z}(t_1))L\left(\mathbf{z}(t_0) + \int_{t_0}^{t_1} f(\mathbf{z}(t), t, \theta)dt\right) = L(\text{ODESolve}(\mathbf{z}(t_0), f, t_0, t_1, \theta)) \tag{14}$$

We need to calculate the gradient of L with respect to the hidden stat $\mathbf{z}(t)$. We do this by defining the adjoint $\mathbf{a}(t) = \partial L/\partial \mathbf{z}(t)$, whose dynamics are given by

$$\frac{d\mathbf{a}(t)}{dt} = -\mathbf{a}(t)^T \frac{\partial f(\mathbf{z}(t), t, theta)}{\partial \mathbf{z}} \tag{15}$$

We then need to compute:

$$\frac{dL}{d\theta} = \int_{t_1}^{t_0} \mathbf{a}(t)^T \frac{\partial f(\mathbf{z}(t), t, \theta}{\partial \theta} dt \tag{16}$$

This development finally lets us define the reverse-mode differntiation (back-propagation) algorithm for an ODE initial value problem:

Algorithm 1 Reverse-mode derivative of an ODE initial value problem

Input: dynamics parameters θ, start time t_0, stop time t_1, final state $\mathbf{z}(t_1)$, loss gradient $\partial L / \partial \mathbf{z}(t_1)$

$s_0 = [\mathbf{z}(t_1), \frac{\partial L}{\partial \mathbf{z}(t_1)}, \mathbf{0}_{|\theta|}]$ ▷ Define initial augmented state

def aug_dynamics($[\mathbf{z}(t), \mathbf{a}(t), \cdot], t, \theta$): ▷ Define dynamics on augmented state

  return $[f(\mathbf{z}(t), t, \theta), -\mathbf{a}(t)^\mathsf{T} \frac{\partial f}{\partial \mathbf{z}}, -\mathbf{a}(t)^\mathsf{T} \frac{\partial f}{\partial \theta}]$ ▷ Compute vector-Jacobian products

$[\mathbf{z}(t_0), \frac{\partial L}{\partial \mathbf{z}(t_0)}, \frac{\partial L}{\partial \theta}] = \text{ODESolve}(s_0, \text{aug\_dynamics}, t_1, t_0, \theta)$ ▷ Solve reverse-time ODE

return $\frac{\partial L}{\partial \mathbf{z}(t_0)}, \frac{\partial L}{\partial \theta}$ ▷ Return gradients

Figure 4: Algorithm for Reverse-Mode Derivative of an ODE initial value problem ([Che+19])

We now begin our concluding discussion on Neural ODEs which will center around applications.

4.3 Applications

4.3.1 Replace Residual Neural Networks

In implementing a neural network that uses Nerual ODEs, the residual blocks of a residual network are replaced with ODE Solvers of Neural ODEs. This is representative of letting the residual layer count go to infinity (continuous layer depth).

Because the forward and backwards pass of inputs is done through an (chosen here to be) implicit ODE solver, intermediate values do not need to be stored, so we have a rough outline as to say this network takes $O(1)$ space, as a function of the number of layers [Che+19].

We attempt our own implementation of such a network, inspired by Mikhail Surtsukov's implementation (which we model closely, almost verbatim). [Sur]

Using PyTorch to define a dynamics function class as well as methods to perfom backwards and forwards passes, we trained (using 5 training epochs and 1875 training inputs per epoch, similar to Surtsukov) our network on the popular MNIST handwritten digit data set. We were able to achieve similar results achieving as high as a 99.110% accuracy on classification.

The authors of "Neural Ordinary Differential Equations" concluded that, with a training task of MNIST handwritten-digit classification, such a network could achieve similar accuracy with better space efficiency, and less than half the parameters [Che+19].

4.3.2 Modeling Dynamical Systems

Problem:
There is a process following an unknown ODE:

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

and we have made some observations along its trajectory:

$$(y_0, t_0, (y_1, t_1), ...(y_n, t_n))$$

Can we find an approximation to $f(y(t), t)$ using a neural network $\hat{f}_\theta(y(t), t)$? Traditionally this is done using a network which is a sequence of transformations to the initial state:

$$y(t_{i+h}) = y(t_i) + f_\theta(y(t_i), t_i)$$

Neural ODEs attempt to learn a function:

$$f_\theta(y(t_i), t_i) = \lim_{h \to 0} y(t_{i+h}) - y(t_i)$$

Implementation:
As a proof of concept we implement a Neural ODE attempting to learn the

the trajectory of a particle whose dynamics are given by:

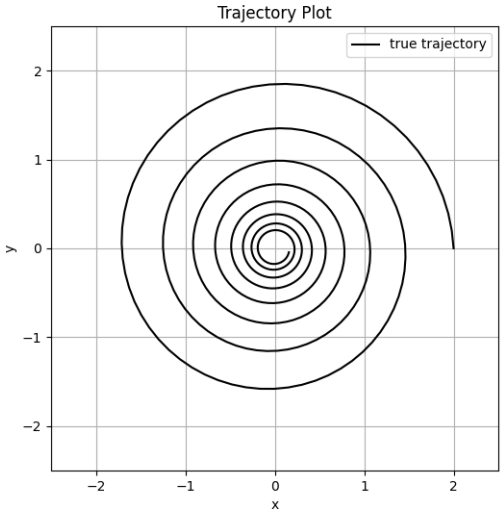
$$\frac{dx(t)}{dt} = -0.1y(t) + 2x(t)$$
$$-\frac{dy(t)}{dt} = 2y(t) + 0.1x(t)$$
$$(x_0, y_0) = (2, 0)$$

In matrix form this is given by:

$$\boldsymbol{y'} = A\boldsymbol{y}$$

where $A = \begin{pmatrix} -0.1 & 2 \\ -2 & -0.1 \end{pmatrix}$ and $\boldsymbol{y} = \begin{pmatrix} x(t) \\ y(t) \end{pmatrix}$

This gives the following spiral trajectory:



4.4 Concluding Thoughts

The authors conclude with discussions about some of the shortcomings of using Neural ODEs. Without much elaboration, these include a messy ap-

proach to mini-batching training data, as well as forcing the user to set error tolerance on forward and reverse training passes.

In all, Neural ODEs are a memory-efficient and computational adaptive [Che+19] implementation of a neural network that attempt to enhance the construction of a residual neural network. Given the prevalence of irregularly shaped time series in the real world, it seems Neural ODEs could become a very important tool in modeling such time series. Though the neural networks implementing Neural ODEs were seen to be accurate and memory-efficient, the complexity of training might reduce their relevance.

References

[BSF94] Y. Bengio, P. Simard, and P. Frasconi. "Learning long-term dependencies with gradient descent is difficult". In: *IEEE Transactions on Neural Networks* 5.2 (1994), pp. 157–166. DOI: 10.1109/72.279181.

[Che+19] Ricky T. Q. Chen et al. *Neural Ordinary Differential Equations*. 2019. arXiv: 1806.07366 [cs.LG].

[CMS12] Dan Cireşan, Ueli Meier, and Juergen Schmidhuber. "Multi-column Deep Neural Networks for Image Classification". In: (2012). arXiv: 1202.2745 [cs.CV].

[Ee17] Weinan Ee. "A Proposal on Machine Learning via Dynamical Systems". In: *Communications in Mathematics and Statistics* 5 (Feb. 2017), pp. 1–11. DOI: 10.1007/s40304-017-0103-z.

[Gib] Kevin Gibson. *Neural networks as Ordinary Differential Equations*. URL: https://rkevingibson.github.io/blog/neural-networks-as-ordinary-differential-equations/.

[He+15] Kaiming He et al. "Deep Residual Learning for Image Recognition". In: (2015). arXiv: 1512.03385 [cs.CV].

[HSW89] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. "Multilayer feedforward networks are universal approximators". In: *Neural Networks* 2.5 (1989), pp. 359–366. ISSN: 0893-6080. DOI: https://doi.org/10.1016/0893-6080(89)90020-8. URL: https://www.sciencedirect.com/science/article/pii/0893608089900208.

[McN+04] Antoine McNamara et al. "Fluid Control Using the Adjoint Method". In: *ACM Trans. Graph.* 23.3 (Aug. 2004), pp. 449–456. ISSN: 0730-0301. DOI: 10.1145/1015706.1015744. URL: https://doi.org/10.1145/1015706.1015744.

[Nie15] Michael Nielsen. *Neural Networks and Deep Learning*. Determination Press, 2015.

[Sol] Pavithra Solai. *Convolutions and Backpropagations*. URL: https://pavisj.medium.com/convolutions-and-backpropagations-46026a8f5d2c.

[Sur] Mikhail Surtsukov. *Neural Ordinary Differential Equations*. URL: hhttps://github.com/msurtsukov/neural-ode/blob/master/Neural%20ODEs.ipynb.

[ZJ] David Duvenaud Zico Kolter and Matt Johnson. *Deep Implicit Layers - Neural ODEs, Deep Equilibrium Models, and Beyond*. URL: http://implicit-layers-tutorial.org/.