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* in modern machine learning practices through a review of contemporary literature on the subject. We will assume the reader has some familiarity with basic neural network architectures and ordinary differential equations but we will not assume an in depth knowledge of either. We will discuss, in detail, the theory and motivation behind two popular variants: *convolutional neural networks* (CNNs), and *residual neural networks* (ResNETs), a type of CNN. Understanding these architectures is particularly important because one application for Neural ODEs is replacing the hidden layers of ResNET implementation.

We 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).

The *convolution* of two functions f and g is often defined as:

$$(f * g)(t) := \int_{-\infty}^{\infty} f(\tau)g(t-\tau)d\tau$$

Intuitively, we think of this integral expression as describing "the amount of overlap of one function as it is shifted over another function". However, we choose to focus on the discrete definition, as this is what we employ for convolutional neural networks [She].

We define the *discrete convolution of f and g* as

$$(f * g)[n] := \sum_{m=-\infty}^{\infty} f[m]g[n-m]$$

which is the convolution operation we reference in the following sections. The interested reader is encouraged to find the rigorous development of this definition in [DM11].

2.2 Convolutional Neural Networks

Now we discuss the motivation for, and design of, a convolutional architecture. At times we 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" (loosely defined as a neural network with multiple hidden layers) with reasonable speed.

In general, neural networks comprised only of fully-connected layers (layers of neurons who receive 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, since CNNs exhibit high levels of accuracy in this area and are 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) lends itself to the development of practical applications. 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 output layer of fully-connected neurons

4. Train the network using some learning algorithm A (such as stochastic gradient descent) by backpropagation

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 $C(\alpha_{i\times j}, \beta_{k\times l}, f)$

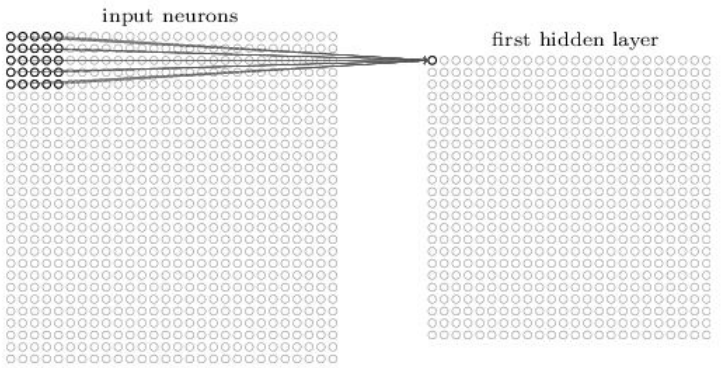
Let's examine these parameters further. Our first parameter, α, is a 2-tuple specifying the dimensions of the input into the layer. If we assume our input layer is the image we are trying to classify itself, then α simply corresponds to the dimensions of the image.

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 of determining hidden neuron activation is 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 comprise 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 cetera). 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 n of feature maps, all of dimension $i \times j$).

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 performing (often non-overlapping) operations on adjacent neurons. 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).

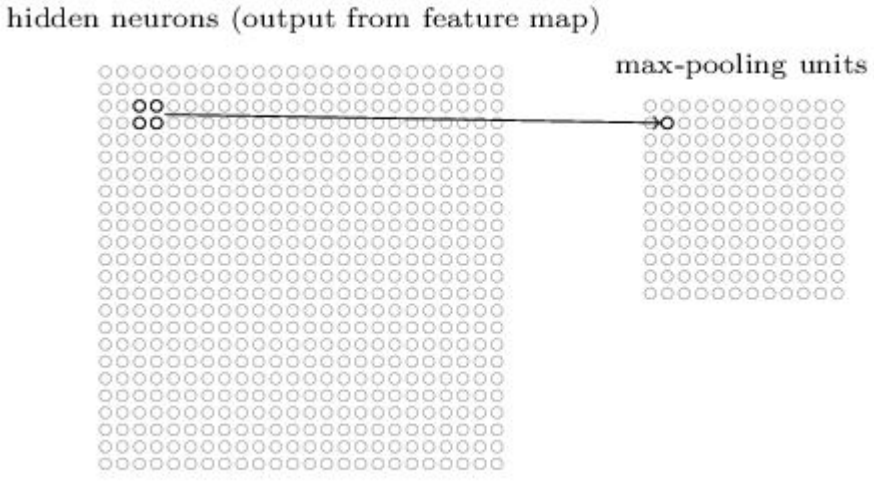


Figure 2: Example of developing a pooling layer from the output of a convolutional layer [Nie15]

2.5 Putting it All Together

A CNN is constructed out of convolutional layers and pooling layers in some order, which are then fed into fully connected layers. There may also be insertions of activation functions such as a ReLU or sigmoid to induce nonlinearity. We do not develop the backpropagation algorithm of a CNN in-depth. We encourage the motivated reader to pursue [Sol] for further development. It is sufficient for our paper to say that traditional backpropagaion of an MLP is not substantially changed for convolutional layers.

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 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 can be found in [HSW89].

3.2 Motivation for Residual Neural Networks

Let's assume we have a CNN, and let it be X. Now, let's define a new, deeper CNN, Y. We define Y to be $Y \equiv X + D$, where D is a collection of additional deep layers, the first of which takes the last layer of X as input.

We would expect, intuitively, this network should perform at least as good as X. This is expected as if we let $D = I$, that is, a collection of networks that are simply identity mappings of the input to the output. This would imply that there exists a network $X + D \approx X$, telling us that, at worse, the deep network performs as well as X.

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 frame the image classification problem differently:

Given a set of images I and a finite collection of categories J, find a mapping $G : I \to J$ 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 the use of identity mapping which traditional deep CNNs struggle with. So, instead, we let our network approximate each $\mathcal{F}_i(x)$ as $\mathcal{F}_i(x) \equiv \mathcal{H}_i(x) - x$, 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, by way 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 *shortcut 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 parameterized CNNs, and can be fairly compared with them [He+15].

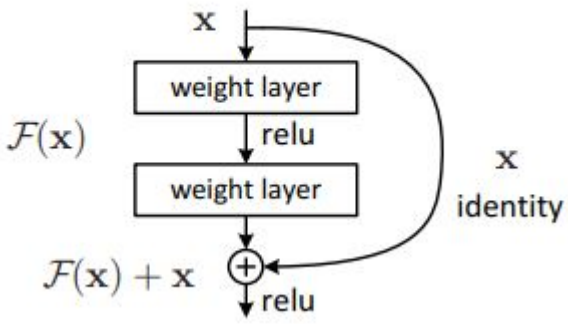


Figure 3: A hypothetical construction of a residual block for an input x [He+15]

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]):

$$\mathbf{y} = \mathcal{F}_i(x, \{W_j\}) + W_s x \tag{2}$$

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:

$$\mathbf{h_{t+1}} = \mathbf{h_t} + f(\mathbf{h_t}, \theta_t) \tag{3}$$

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

In fact, we consider the following derivation:

$$\begin{aligned} \mathbf{h}_{t+1} &= f(\mathbf{h}_t) + \mathbf{h}_t && (4)\\ &= \frac{\Delta t}{\Delta t} f(\mathbf{h}_t) + \mathbf{h}_t && (5)\\ &= \Delta t G(\mathbf{h}_t) + \mathbf{h_t} && (6) \end{aligned}$$

Where $G(\mathbf{h}_t) = f(\mathbf{h}_t)/\Delta t$ ([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

$$\frac{d\mathbf{h}(t)}{dt} = G(\mathbf{h}_t, t, \theta) \tag{7}$$

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.

In essence, we will be defining some dynamics function $G(\mathbf{h}, t, \theta)$ such that the "learning" occurs by way of *variation of parameters* of this function with each forward/backwards pass. We will need to define how to perform both

forwards and backwards passes (which we do below), but, assuming we know how to do so, learning the parameters that best optimize G when we perform an ODE Solve is the primary computational goal.

So, practically speaking, our "network" learns by optimizing the parameters of a dynamical system that is a continuous-depth residual block (which can be loosely thought of as being a block with *infinite* layers).

Before going further in the details of Neural ODEs we would like to make an important distinction between Physics Informed Neural Networks (PINN) and Neural ODEs. PINNs work on obtaining numerical approximations to the solution y of an ODE $\frac{dy}{dt} = f(t, y(t))$ by representing the solution as some neural network $y = y_\theta$. f is known and the model y_θ is minimising a loss function of the form:

$$\min_\theta \frac{1}{N} \sum_{i=1}^{N} \left\| \frac{dy_\theta}{dt}(t_i) - f(t_i, y_\theta(t_i)) \right\|$$

This is a distinct notion. Neural ODEs use neural networks to specify differential equations and PINNs use neural networks to obtain solutions to prespecified differential equations. This can be a common point of confusion.

4.1.1 Forward and Backwards Passes

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

$$\mathbf{h}(t_1) = \text{ODESolve}(\mathbf{h}(t_0), G, t_0, t_1, \theta) \tag{8}$$

$$= \mathbf{h}(t_0) + \int_{t_0}^{t_1} G(\mathbf{h}(t), t, \theta) dt \tag{9}$$

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):

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

This isn't a necessarily easier problem, but considering the case where B and C are instead matrices,

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

is equivalent to

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

The utility of this method, computing the dual rather than the original matrix/vector, is clear when B is a matrix. 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 $Q = [q_1^T, \dots q_n^T]^T$ and $F(Q, u) = [f_0(q_0, u)^T, \dots f_{n-1}(q_{n-1}, u)^T]^T$. 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 $q_1, \dots, q_n$, we can view R as a sequence of *adjoint* states $r_1, \dots r_n$. 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:

$$L(\mathbf{z}(t_1)) = L\left(\mathbf{z}(t_0) + \int_{t_0}^{t_1} f(\mathbf{z}(t), t, \theta)dt\right) = L(\text{ODESolve}(\mathbf{z}(t_0), f, t_0, t_1, \theta)) \tag{10}$$

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

$$\frac{d\mathbf{a}(t)}{dt} = -\mathbf{a}(t)^T \frac{\partial f(\mathbf{z}(t), t, \theta)}{\partial \mathbf{z}} \tag{11}$$

We then need to compute:

$$\frac{dL}{d\theta} = \int_{t_1}^{t_0} \mathbf{a}(t)^T \frac{\partial f(\mathbf{z}(t), t, \theta)}{\partial \theta} dt \tag{12}$$

This development finally lets us define the reverse-mode differentiation (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 $[f(\mathbf{z}(t), t, \theta), -\mathbf{a}(t)^\intercal \frac{\partial f}{\partial \mathbf{z}}, -\mathbf{a}(t)^\intercal \frac{\partial f}{\partial \theta}]$ ▷ 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])

4.3 Applications

4.3.1 Hand Written Digit Recognition

As a proof of concept that Neural ODEs can be an improvement over Residual Neural Networks we attempt to implement a Neural ODE that can recognize hand written digits. In a network that uses Neural 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].

Implementation:
Our own implementation of such a network is inspired by and modeled closely after Mikhail Surtsukov's implementation. [Sur]

We include a flowchart describing the network below:

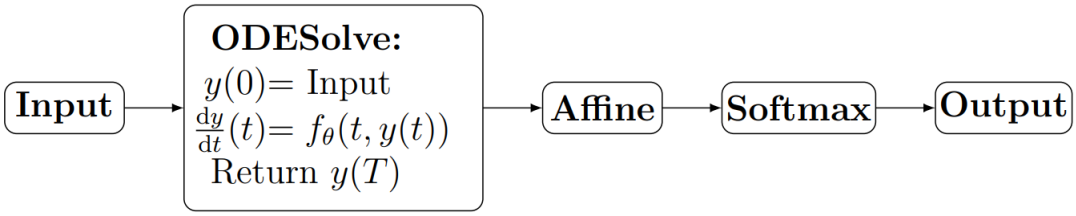


Figure 5: Flow chart for image classification Neural ODE ([Kid22])

Suppose we observe some picture $y_0 \in \mathbb{R}^{3\times 28\times 28}$ and wish to classify it as a written digit. We proceed by taking $y(0) = y_0$ as the initial condition of the

neural ODE and evolve the ODE until some time T. An affine transformation $l_\theta : \mathbb{R}^{3\times 28\times 28} \to \mathbb{R}^2$ is then applied followed by a softmax. This allows us to interpret the output as length 10 tuple ($\mathbb{P}$(picture is of 0)..$\mathbb{P}$(picture is of 9))

Model Details:
1) Dynamics function: We use a traditional CNN as our dynamics function whose weights and biases are our learned parameters, θ. We use two convolutional layers and use a ReLU activation function.

2) Training: The dynamics function was trained for 5 training epochs with 1875 inputs per epoch.The training data is the popular MNIST handwritten digit data set. The loss function was defined as the cross entropy loss between the true class distribution vs the predicted class distribution.

$$L = -\sum_i y_i \log(\hat{y}_i)$$

Loss function defined above was optimized using the Adam Optimizer from Pytorch.

3) ODE Solver: A simple python function that uses Euler's method was implemented as an ODE solver. ODE solvers in general are not computationally negligible hence using more complicated ODE solvers (such as adaptive step solvers) would cause a significant increase in training time without much decrease in the loss function. We demonstrate this principal in the application to dynamical systems.

Results:
$\to$ Initial: accuracy = 10.280%, loss = 0.16338
$\to$ Final: accuracy = 99.230%, loss = 0.03573

Every epoch takes approximately 11.52 minutes to complete thus total training time of approximately 1 hour. This is much slower than most Residual Neural Network. However, 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

Consider 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:

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

and can be thought of as the infinite limit to the layers of a traditional neural network.

Implementation:
As a proof of concept we attempt our own implementation of such a network in Python. The Neural ODE learns the the trajectory of a particle whose dynamics are given by 2 coupled first order linear ODEs:

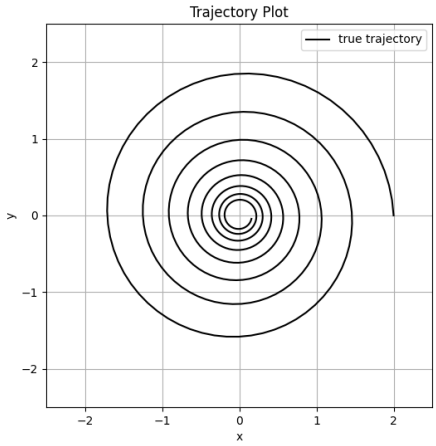
$$\frac{dx(t)}{dt} = -0.1y(t) + 2x(t)$$
$$-\frac{dy(t)}{dt} = 2y(t) + 0.1x(t)$$
$$(x_0, y_0) = (2, 0)$$

In matrix form this is given by:

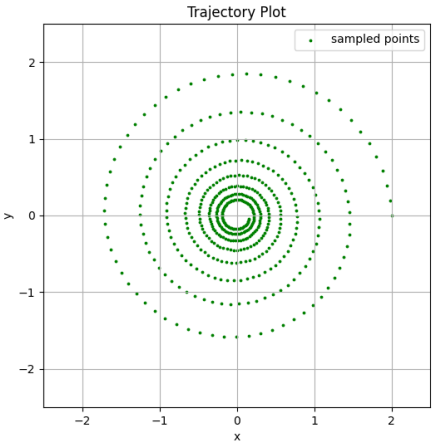
$$\boldsymbol{y'} = A\boldsymbol{y}$$

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 a spiral trajectory along which we sample 500 points as our training dataset:



(a) Spiral Trajectory



(b) Sampled Points

Model Details:
1) Dynamics Function: We use Pytorch to define a Multi Layered Perceptron with one hidden layer (50 nodes) and a tanh activation function. This MLP serves as our dynamics function.

2) Training: The dynamics function was trained for 150 epochs with the RMSprop Optimizer from Pytorch. The loss function was defined as the mean

squared error between the true trajectory and predicted trajectory.

$$L = \sum_i (y_i - \hat{y}_i)^2$$

Loss function defined above was optimized using RMSprop from Pytorch.

3) ODE Solver: Solvers were implemented using the odeint function from torchdiffeq. We use both fixed step size solvers (Euler, Runge Kutta Order 4) and adaptive step size solvers (Dopri5).

Results:
→ Euler: time = 13.43 sec, error = 0.0183
→ Runge Kutta Order 4: time = 12.95 sec, error = 0.0173
→ Dopri5: time = 39.06 sec, error = 0.0167
Space complexity for all methods remains the same as we make no changes to the dynamics function. However, the adaptive step size solver takes more than twice as long to train while providing minimal decrease in the mean squared error. We conclude using Runge Kutta might be the best for real world applications.

Below we include plots for the model with the Runge-Kutta solver:

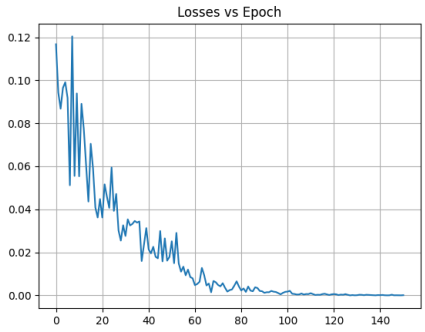
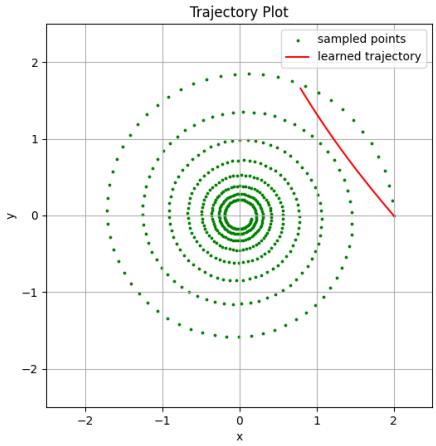
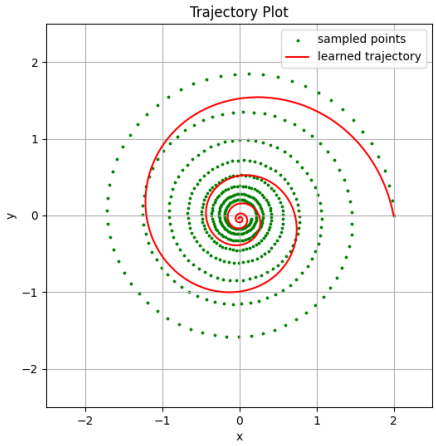


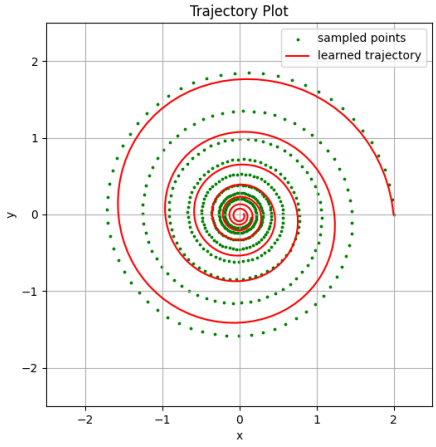
Figure 7: Losses vs Epochs



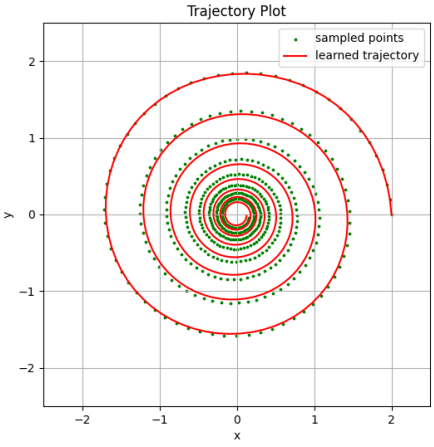
(a) Initial guess



(b) Epoch 50



(c) Epoch 100



(d) Epoch 150

The GitHub repository containing our implementations can be found at `https://github.com/owenc21/Neural-ODE`.

4.4 Conclusion

The authors conclude with discussions about some of the shortcomings of using Neural ODEs.

One major advantage of using Neural ODEs for supervised learning is space efficiency. By completing a forward pass with the use of an ODE Solver and backwards pass with the differentiation of that solver, the quantity of intermediate values that need to be maintained is cut drastically (and this quantity is typically nontrivial with respect to deep learning architectures [Che+19]). Additionally, Neural ODEs are said to be *computationally adaptive*[Che+19], in that the user has a choice in which ODE Solver they employ (Euler, Runge-Kutta, et cetera).

The major drawbacks of using Neural ODEs is that, in exchange for removing the need to define the hyperparameter of *network depth*, the user must define an error tolerance for the choice of ODE Solver. Additionally, ODE Solvers are not computationally negligible, or is the differentiation of them (ODE Solvers will often include substantial control flow), meaning that training and evaluation using a Neural ODE will be, at best, as fast as a typical deep network [Che+19].

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].

[DM11] Steven B. Damelin and Willard Miller Jr. *The Mathematics of Signal Processing*. Cambridge Texts in Applied Mathematics. Cambridge University Press, 2011. DOI: 10.1017/CBO9781139003896.

[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.

[Kid22] Patrick Kidger. "On Neural Differential Equations". In: (2022). arXiv: 2202.02435.

[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.

[She] Anirudh Shenoy. *How Are Convolutions Actually Performed Under the Hood?* URL: https://towardsdatascience.com/how-are-convolutions-actually-performed-under-the-hood-226523ce7fbf.

[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/.