Simon Fraser University
Faculty of Statistics & Actuarial Science
Burnaby, British Columbia

An Introduction to Neural Differential Equations

PRESENTED BY:

Barinder Thind

Contents

1 Introduction

The ever-expanding umbrella that encompasses deep-learning methodologies welcomed another member earlier this year with the advent of Neural Ordinary Differential Equations (NeuralODEs) [4]. This approach expands on Residual Neural Networks [16] which circumvented the vanishing gradient problem [29] that traditional deep neural networks confronted with an increasing number of hidden layers. NeuralODEs transformed the above approach from a discrete to a continuous domain allowing for more efficient memory allocation, flexibility with respect to time-evaluation, and parameter efficiency.

In Section II, I introduce neural networks along with an example demonstrating their use. In Section III, I present residual neural networks and go into some *depth* as to why they were an improvement; the results of a coded example are provided. Then, Section V unifies together the previous sections by introducing neural differential equations as a gestalt of the aforementioned approaches and differential equations. All code will be provided in the index and in a separate *.rmd* file. The Appendix contains additional information on definitions, differential equations, and the connection between DEs and neural networks.

2 Neural Networks

Neural networks have excelled at prediction problems over the last decade. In this section, I highlight the underlying methodology and present a couple of coding examples.

2.1 Methodology

2.1.1 Forward Pass

For simplicity sake, we will first look at a network with just a ”singe hidden layer”. Consider Figure 1 - the blue *”neurons”* contain values of the input data. For example, if the input was an image, then each neuron would hold a value corresponding to the gray scale value of each pixel of the image. This is known as the *activation* value. The red neurons contained within the second row are referred to as neurons from the *hidden layer*. Each single layer is a non-linear transformation of a linear combination of each activation in the first layer. For example, z_1 would be defined as:

$$z^{(1)} = \sigma(\vec{\alpha_{0_1}} + \vec{x}\boldsymbol{\alpha}^{(\mathbf{1})}) \tag{1}$$

Where $\boldsymbol{\alpha}^{(\mathbf{1})}$ and $\vec{\alpha_{0_1}}$ is the set of weights and biases[1] and $\sigma()$ is some *activation function* that transforms the resulting linear combination so that it can provide us with

[1]These are initialized randomly in this simple case

useful numbers (for example, we might want probabilities for a binary response so we could use the sigmoid function[2] when that is the case). Note that the vector $\vec{x}$ corresponds to a single "row" of our data set (or, a single image if that was the data type we were working with) and is therefore a p-dimensional vector where p is the number of covariates.

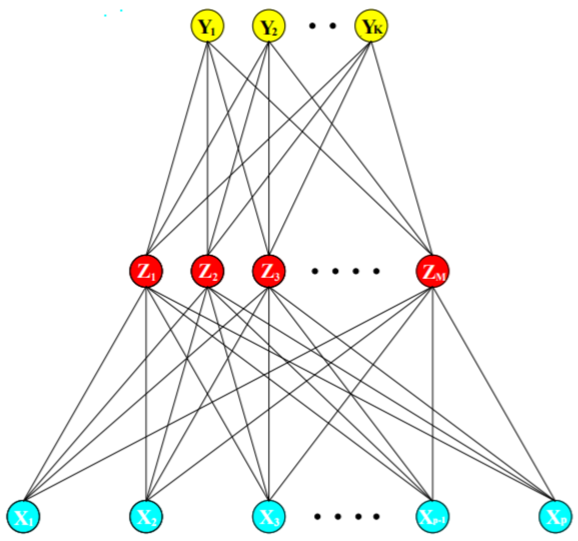

Figure 1: An overview of a single hidden layer network [28]

Once we have the values for the m[3] neurons in the hidden layer, we have another set of activations! Using these, we can move onto the final layer. In the case of image classification, say for the purposes of number recognition, an image might correspond to a single number $y \in \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$. In this case, $K = 10$ and is the number of possibilities for classification. The last transformation assigns a probability to each of the K classes effectively providing a likelihood that your observation (in our example, the single image) belongs to each of them, respectively. Often, the softmax function:

$$g_k(T) = \frac{e^{T_k}}{\sum_{i=1}^{K} e^{T_i}} \tag{2}$$

is used as it allows probability specification for a number of classes[4]. Here, there are k sets of T values and these are all linear combinations moving from the hidden layer to the output layer. You can imagine that this could be generalized to n number of hidden layers with some choice of m neurons in each of them resulting in a myriad of parameters to be estimated!

[2] $\frac{1}{1+e^{-x}}$
[3] The choice of m is left to the user
[4] In fact, letting $k \in \{0, 1\}$ returns the famous logistic transformation

2.1.2 Backpropogation

The ”learning” of this approach takes place in a process called *backpropogation* - this is just jargon for gradient descent. In order to do so, we must first define a loss function. This function is some measure of the aggregated residual between our prediction and the true value. One approach is to use squared-error:

$$R(\theta) = \sum_{k=1}^{K} \sum_{i=1}^{N} (y_{ik} - f_k(x_i))^2 \tag{3}$$

This function is a sum of the difference between the prediction for every observation for every output. For example, the outer sum would be of the difference between the predicted probability of an image belonging to a particular class, over all classes and the inner sum would aggregate over every image (or observation) you have[5].

Now that we have a measure of error, we can look to minimize it! This function takes in $(p+1)\cdot M \cdot (M+1)\cdot K$ parameters[6] so we will have a very high dimensional gradient. This results in an inordinate amount of peaks and valleys on the optimization landscape. It is also very likely that the global minimizer of $R(\theta)$ will overfit the data so any local minimizer may serve us better; in fact, we will take small steps towards the optimum specified by a ”learning rate”, γ.

For simplicity sake, we consider a network with a single neuron in its 2 hidden layers and only look at a 1-dimensional observation [1].

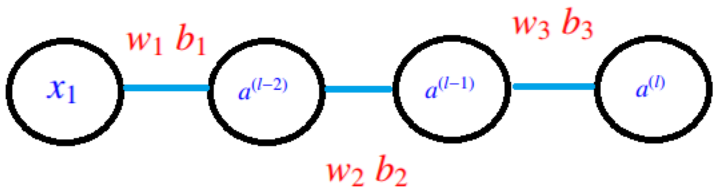

Figure 2: Schematic representing toy example

Then, the cost function, R will have 6 parameters outlined in red in Figure 2. Using (3), we see that in this simple case, the cost function reduces to $R_1 = (z^{(l)} - y)^2$ where $a^{(l)}$ is the activation in the final layer[7] defined as: $a^{(l)} = \sigma(w^{(l)}a^{(l-1)} + b^{(l)})$. For convenience, define $w^{(l)}a^{(l-1)} + b^{(l)}$ as $u^{(l)}$. Remember, we want to minimize the cost function; we can note that a change in the weight $w^{(l)}$ causes some change to the cost function, R_1 - we want to know this change: $\dfrac{\partial R_1}{\partial w^{(l)}}$ as our goal is to minimize

[5]There are a plethora of loss functions to pick from; for example, cross-entropy and log-loss
[6]The set θ encompasses these parameters
[7]Let l be indicative of the last layer i.e. $w_3 = w^{(1)}$ and $w_2 = w^{(l-1)}$

this. Using chain rule, we can observe that this derivative can be broken down to a number of sub-derivatives:

$$\frac{\partial R_1}{\partial w^{(l)}} = \frac{\partial u^{(l)}}{\partial w^{(l)}} \cdot \frac{\partial a^{(l)}}{\partial z^{(l)}} \cdot \frac{\partial R_1}{\partial a^{(l)}} \tag{4}$$

$$= a^{(l-1)} \cdot \sigma^{'}(z^{(l)}) \cdot 2 \cdot (a^{(l)} - y) \tag{5}$$

We want to find the roots of this derivative but, not ONLY this derivative. First, we note that (3) is defined for every observation:

$$\frac{\partial R}{\partial w^{(l)}} = \frac{1}{n} \sum_{i=0}^{n-1} \frac{\partial R_i}{\partial w^{(l)}} \tag{6}$$

And note that (6) is only one of the 6 derivatives making up the gradient of the cost function:

$$\nabla R = \left(\frac{\partial R}{\partial w^{(l)}} \quad \frac{\partial R}{\partial w^{(l-1)}} \quad \frac{\partial R}{\partial w^{(l-2)}} \quad \frac{\partial R}{\partial b^{(l)}} \quad \frac{\partial R}{\partial b^{(l-1)}} \quad \frac{\partial R}{\partial b^{(l-2)}} \right)^T = \vec{0}$$

The parameter values that satisfy the above equation are the changes we need to make to the current weights. The change is done proportional to the aforementioned learning rate, γ. This approach is taken for computational efficiency - finding the full gradients is nearly impossible so the optimal values are found in *mini batches*[8]; these are subsets of observations for which the optimization takes place as opposed to the entire data set. This approach is also known as *stochastic gradient descent*. The process repeats for some number of *epochs*[9].

In summary, you begin with a set of weights, train the model, and get predictions. You run these predictions through a loss function and attempt to minimize it by updating the parameters of the function according to the gradient. You do this at some learning rate and the evaluations are done on subsets of data (mini batches) for some number of iterations.

2.2 Results

2.2.1 Data Description

The data set is taken from a 2017 Kaggle competition [27] in which participants were asked to classify satellite images as either icebergs or ships. There are two variables corresponding to the pixel values of the images (x, y coordinates) and a unique *ID*

[8]Definitions for some of this jargon are provided in the Appendix

[9]A single epoch is one full forward and backward pass for every observation in your data set

variable which corresponds to the *i* index in the theory above (observation number). There's a final binary (output) variable which classifies the image as an iceberg (or not). In Figure 3, some of the images are visualized.

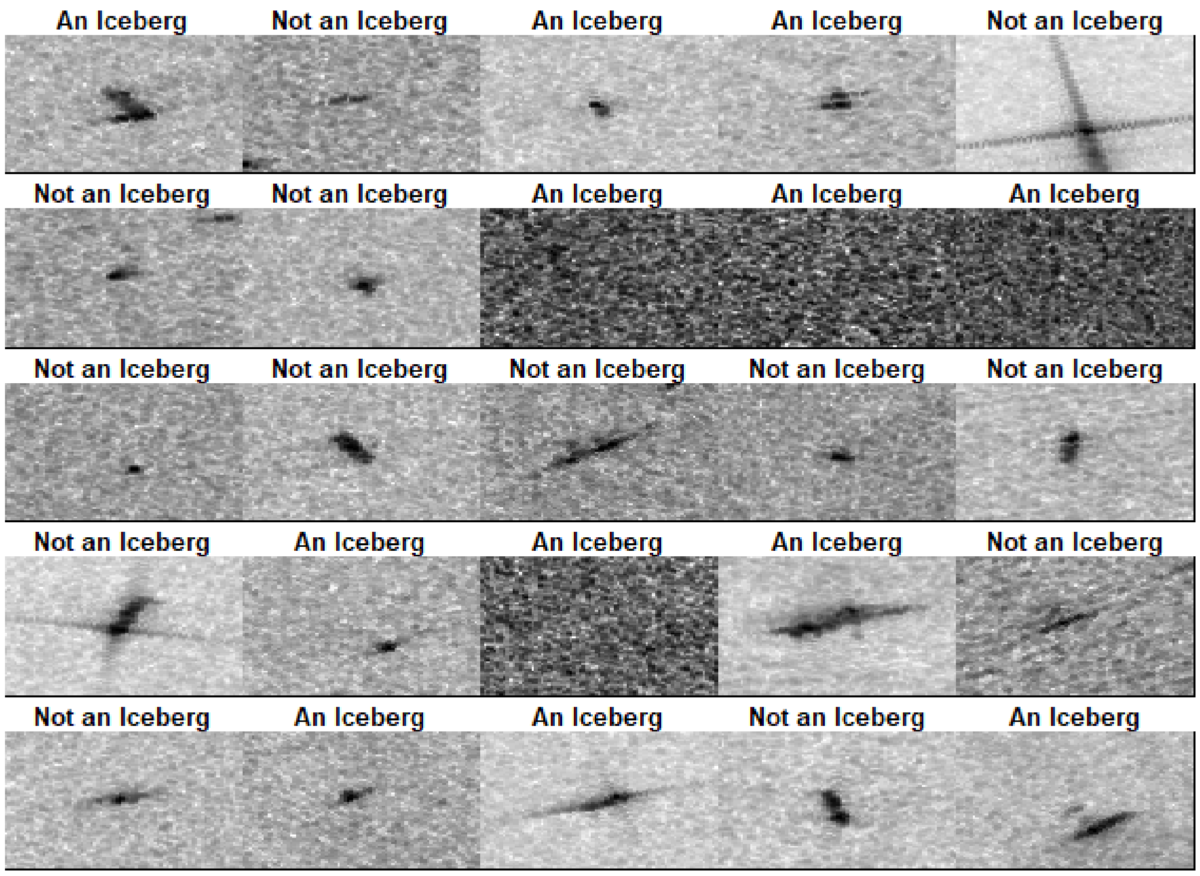

Figure 3: A snapshot of the grayscale iceberg/not iceberg images

2.2.2 Model Specifics

The model was trained on 1300 images and used to predict 304 [3]. The activation function used in the 4 *dense* hidden layers was *relu*[10] and the sigmoid function was used for the output. The optimization landscape was explored by *stochastic gradient descent* (sgd) and loss was characterized by *binary cross-entropy*.

2.2.3 Performance

The model performed with exceptional mediocrity after being run through 150 epochs. Figure 4 provides the loss results for the model as it worked its way through the epochs. The final accuracy on the test images was: 54%.

2.2.4 An Example from Scratch

A model was trained in R which was used to predict a binary response from normally generated data. The response, y, was 1 if the randomly generated Gaussian data point

[10] $\sigma(z) = max\{0, z\}$

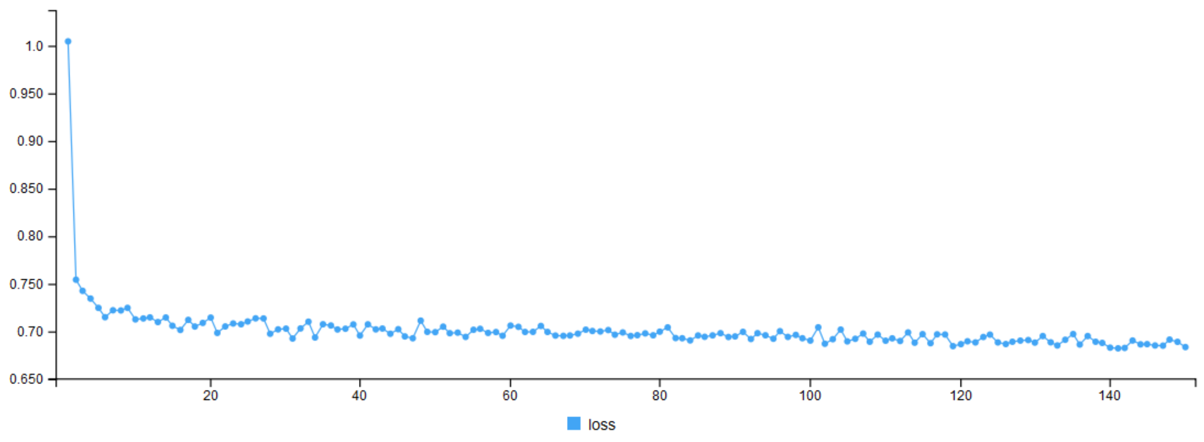

Figure 4: Accuracy results for neural network model

was between -0.5 and 0.5 and 0 otherwise. The model was set to predict all 0's in the beginning and had an accuracy of 0.64. After training the model for 50 epochs, the model had an accuracy of 1. The MSE loss plot is given in Figure 5:

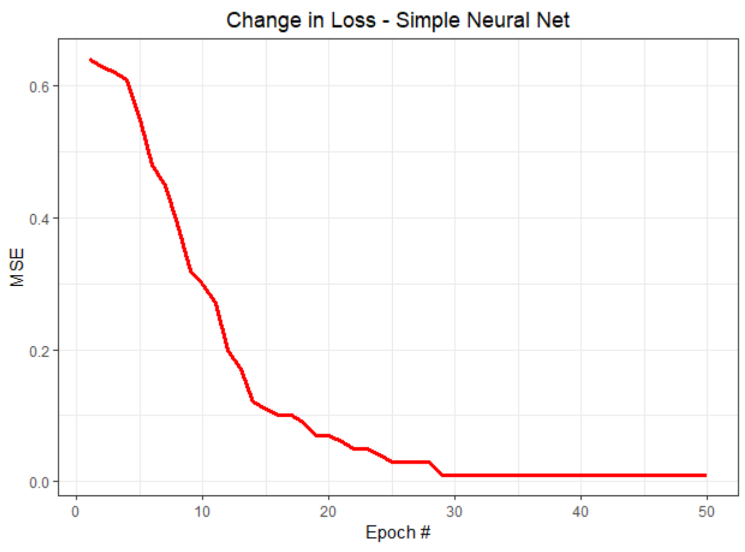

Figure 5: Loss results for hand-coded neural network

3 Residual Neural Networks

In this section, I introduce an extension to deep neural networks developed by researchers at Microsoft [16].

3.1 Methodology

3.1.1 Residual Blocks

A common problem with recurrent (plain) neural networks is their inability to be trained on a large number of hidden layers. This problem arises due to vanishing (and exploding) gradients. A vanishing gradient occurs for weights and biases earlier in the network. Recall that, during backpropagation, we use chain rule to find gradient values and that, the further back we are, the more terms there are that are used to compute the gradient. Since there are more terms, their exists a higher probability that some of those terms will be small and hence, due to the multiplicative nature of the chain rule, there becomes a tendency for those earlier weights to hardly even move during the update portion of the iteration[11] [7].

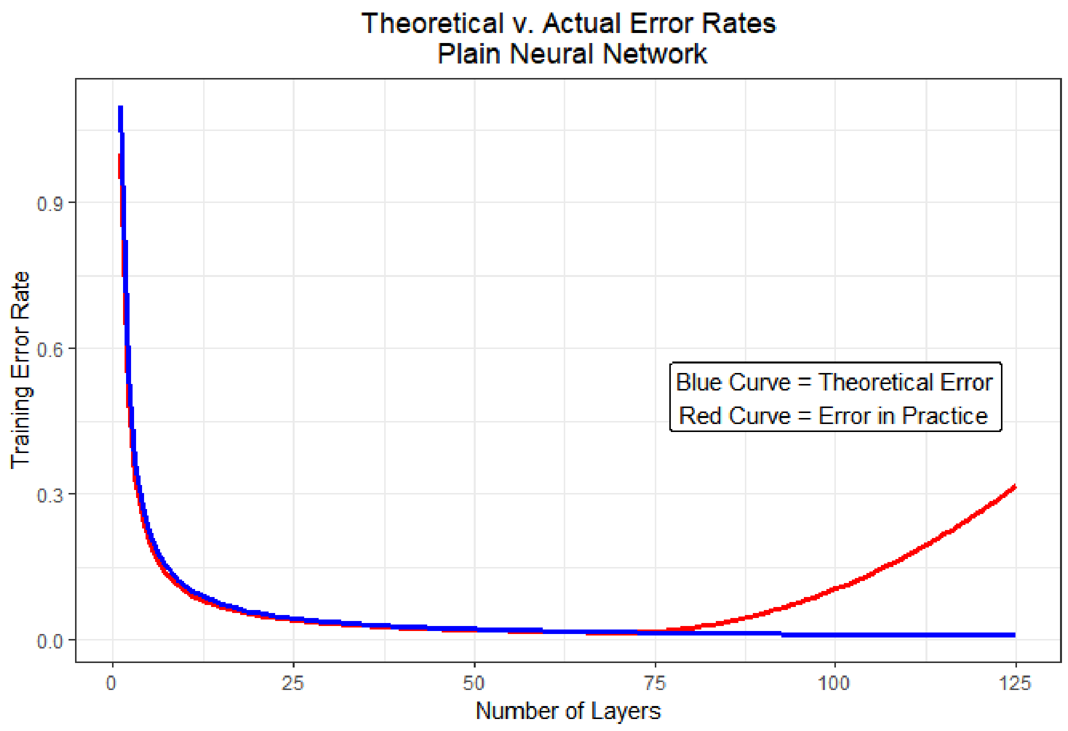

Figure 6: An overview of the training set error rates for recurrent (plain) neural networks. The vanishing gradient problem is theorized to be responsible for the blue curve

A solution to this problem comes in the form of *residual blocks*. These are modifications to the linear part of the neural network in between layers. Consider an activation in classic neural networks:

$$z^{(1)} = \sigma(\vec{\alpha_{0_1}} + \vec{x}\boldsymbol{\alpha}^{(\mathbf{1})}) \tag{7}$$

Letting $z^{(2)}$ and $z^{(3)}$ be the activations in the second and third layer[12]. Then, normally, we would have the following:

$$z^{(1)} = \sigma(\vec{\alpha_{0_1}} + \vec{x}\boldsymbol{\alpha}^{(\mathbf{1})}) \tag{8}$$
$$z^{(2)} = \sigma(\vec{\alpha_{0_2}} + z^{(1)}\boldsymbol{\alpha}^{(\mathbf{2})}) \tag{9}$$
$$z^{(3)} = \sigma(\vec{\alpha_{0_3}} + z^{(2)}\boldsymbol{\alpha}^{(\mathbf{3})}) \tag{10}$$

[11]The update is: $w_i = w_{i-1} - \gamma \cdot \frac{\delta R}{\delta w_{i-1}}$. That is to say, the second term in this equation can become very small

[12]These are single dimensional i.e. only a single neuron in each layer. This generalizes easily an m-dimensional case where these would be vectors instead

However, in a residual block we adjust say, z_3 so that we get:

$$z^{(3)} = \sigma(\vec{\alpha_{0_3}} + z^{(2)}\boldsymbol{\alpha}^{(3)}) + z^{(2)} \tag{11}$$

The key insight here is that as the weights $\boldsymbol{\alpha}^{(3)}$ and the bias $\vec{\alpha_{0_3}}$ vanish, the input into the activation function tends toward the identity transformation rather than 0. This means that, instead of having a degradation in learning as we increase the number of layers, the neural network will instead have, at *worst*, an identity transformation layer to layer (that is, the activation function will just take you back to the activation value of the $((i-2)+1)^{th}$ layer and allow the optimization to flourish in other elements of the gradient that are not (yet) experiencing the problem.

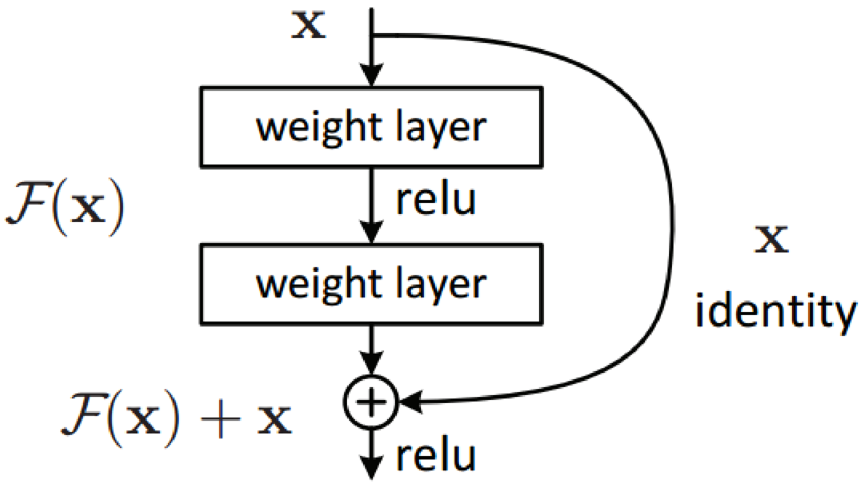
x
weight layer
$\mathcal{F}(\mathbf{x})$
relu
weight layer
x
identity
$\mathcal{F}(\mathbf{x}) + \mathbf{x}$
relu

Figure 7: A residual block. The $F(x)$ here is analogous to the $z^{(3)}$ in the notation used here. [13]

The algorithm for backpropagation remains the same. The additional derivative is computed with respect to the added term but the overall process follows the same logic.

3.2 Results

The titanic data set was used once again here for the implementation. The residual blocks were used in conjunction with a convolutional neural network (CNN)[13] (as opposed to an addition to recurrent neural networks) [11]. The relevant code is found in section 7.2.2 of the Appendix.

The model used the same number of epochs as the previous neural network and was trained on the same number of images (1300). Batch normalization was applied along with a number of other sub-layers relevant to a convolutional neural network[14]. In Figure 8, we can see the relative superiority of this approach:

[13]This choice was made due to the nature of the data

[14]Definitions are provided in Section 7.2.1

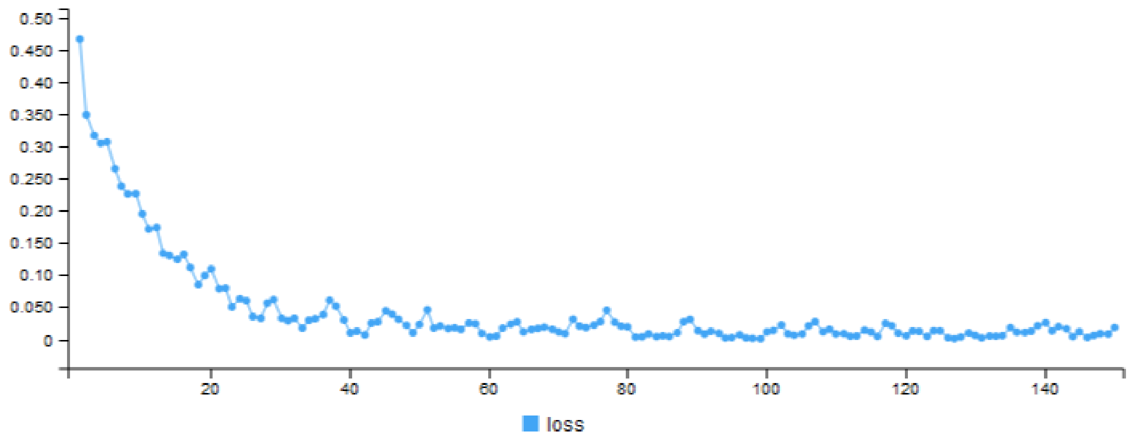

Figure 8: Accuracy results for the CNN with residual blocks.

The prediction accuracy for this model on the same set of images was over 87% using mean squared error as the measure.

4 Neural Differential Equations

4.1 Methodology

4.1.1 An Overview

The main idea underlying the use of differential equations is that they require a fewer number of parameters which contributes to efficiency. To see why, consider a simple linear regression problem where the goal is to estimate optimal values of a and b for $f(x) = ax + b$. Observe that we make an implicit assumption here - the function $f(x)$ is differentiable and so, we can find $f(x)$ directly or we can estimate its derivative, $f'(x)$. The derivative of $f(x)$ is a; in the differential equation approach, we only have one parameter to estimate! And, in fact, differential-equation solver approaches don't provide analytic forms of $f(x)$ but rather, numerical values that are dependent on the initial inputs (the data) thus, eliminating the need to ever find b explicitly.

Remember that a neural network, more than anything else, is a high dimensional function, $f(\vec{x}; \theta)$ where θ is the set of weights and biases. Instead of estimating this function, we can model the derivative instead - i.e. the change in the function from layer to layer. Consider some vector [14] of hidden activations[15] [16]:

$$z_{t+1} = f(z_t, \theta_t) \tag{12}$$

More importantly, in the case of residual networks, the functional form becomes:

[15]Moving forward, we will consider the depth (or the hidden layer we are at) by t

[16]Here, I am going to let the subscript represent where in the network the activations are at

$$z_{t+1} = z_t + f(z_t, \theta_t) \tag{13}$$

An important insight is the striking resemblance of (13) to Euler's method [17] and, recall that Euler's method is a discretization of a continuous relationship between x and y (inputs and outputs). A neural network then, similarly, is also a discretization characterized by the hidden layers. ResNets, while discrete, effectively work as ODE solvers by measuring the rate of differnce in their hidden layers. Let t, the depth, go to infinite - then the entire set of layers of a neural network can be written as a differential equation:

$$\frac{\partial z}{\partial t} = f(z(t), t; \theta) \tag{14}$$

Intuitively, we have taken a step back in the ODE solving process to where we now have an option on which direction to go to solve the problem. In ResNets, Euler's method is the specified direction however, we aren't limited to that approach here and could use more sophisticated and efficient estimators. The authors use a "black-box differential equation solver".

The trajectory of Euler's method attempts to model the dynamic of the output over the continuum, x; analogously, the hidden layers in a neural network represent the dynamics of the hidden activations with respect to the depth of the network. The limit allows us to smooth out this trajectory so that we can evaluate a hidden activation at any depth $\in \mathbf{R}$. Note that the differential equation trajectories will differ depending on the inputs (think of these as initial conditions). In Figure 9, I present one such trajectory[18].

One advantage of such an approach is that there is a constant memory cost with respect to depth. Recall that derivatives in earlier hidden layers would require more operations in the backpropagation process but this is not the case here. This model also has much less parameters than networks with residual blocks and can be computed efficiently by ODE solvers. There is also an advantage associated with irregular time-series model that classic neural networks had trouble dealing with.

The hidden state is evaluated by the following integral:

$$z(t) = \int f(t, h(t), \theta_t) dt \tag{15}$$

[17]The appendix provides more details
[18]It's the plot on the right hand side

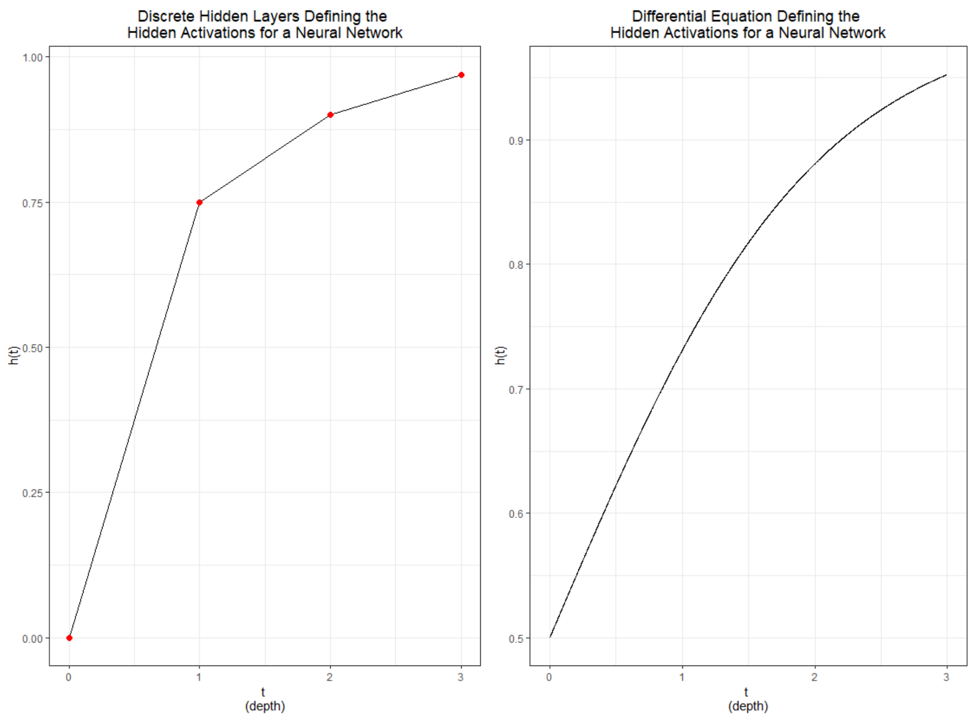

Figure 9: Trajectory comparisons of the two hidden state approaches. Note that the red dots in the left plot are the only evaluations we can do with classic neural networks whereas the dynamics are modeled at any depth in the NeuralODE approach

Where θ_t is the set of parameters at some layer, t. Lastly, note that the initial conditions (that is, at t_0) are given by the observations, $\vec{x}$ with the output being evaluated at some t_j where j is the + 1 iteration from the last hidden state. Deciding on t_0 and t_j is a problem left best to the optimization process; therefore, the final predictions can be summarized as [25]:

$$\hat{y} = z(t_j) = ODESolve(z(t_0, t_1, \theta, f)) \tag{16}$$

4.1.2 Backpropagation

Now that we have a functional form of the hidden states, we can begin to formulate the backpropagation process. As before, we begin with some (general) loss function:

$$R(t_0, t_1, \theta_t) = R(ODESolve(z(t_0, t_0, t_1, \theta, f))) \tag{17}$$

Beginning with the final hidden state, we can compute the gradient: $\dfrac{\partial R}{\partial z(t)}$. We implement the chain rule here because the hidden states themselves are dependent on t - essentially, we are working backwards along the path taken to get to the output, $z(t_j)$. In the paper, they use the adjoint method [15]. This is a numerical technique used to compute derivatives. An adjoint state is defined as:

$$a(t) = -\frac{\partial R}{\partial z(t)} \tag{18}$$

This is the change in the loss at any point t in the hidden state interval. Note that the loss function and the neural network are differentiable. We observe then that:

$$\frac{\partial a(t)}{\partial t} = -a(t)\frac{\partial f(t, z(t), \theta_t)}{\partial z(t)} \tag{19}$$

Which we note is also a differential equation. Using the Fundamental Theorem of Calculus, we can integrate both sides to find a solution for $a(t)$ and, recalling (18), we derive:

$$\frac{\partial R}{\partial h(t)} = -a(T) = \int a(t)^T \frac{\partial f(t, z(t), \theta_t)}{\delta z(t)} dt \tag{20}$$

And finally, we can solve this integral with the black-box ODE solve that was alluded to earlier. Computing this integral from $t1$ to t_0[19], we can get the gradient at t_0. Lastly, the θ gradient is computed by:

$$\frac{\partial R}{\partial z(t)} = \int_{t_1}^{t_0} a(t)^T \frac{\partial f(t, z(t), \theta_t)}{\partial \theta} dt \tag{21}$$

All of these derivatives can be computed simultaneously as the results do not depend on one another; this parallelization leads to computational efficiency.

4.2 Results

For the purpose of this paper, tests were limited to the MNIST[20] data set [21] [22]. There was a total of 6 epochs with each mini batch being of size 32 (this means that it took over 1500 iterations to complete each epoch [22]). The Neural ODE block was embedded in a convolutional neural network and effectively replaced 6 residual blocks. After just a SINGLE epoch, the ODE block fell to an error rate $< 2\%$. The results can be seen in Figure 10.

The code to produce these results is provided in the appendix[23].

[19] Remember, this is a reverse traversal of the hidden states

[20] This is image data for number classification

[21] Ideally, I would have used the results on the iceberg/ship data but due to some technical difficulties, I wasn't able to complete it on time; I will continue to work on this for the purpose of my thesis and hope to have it done in the next couple of months.

[22] Epoch = # of iterations x batchSize

[23] NOTE: This is more or less source code. I have cited the author. I do however go through it, function by function. I have also begun my own implementation in R. More details can be found in the .rmd file

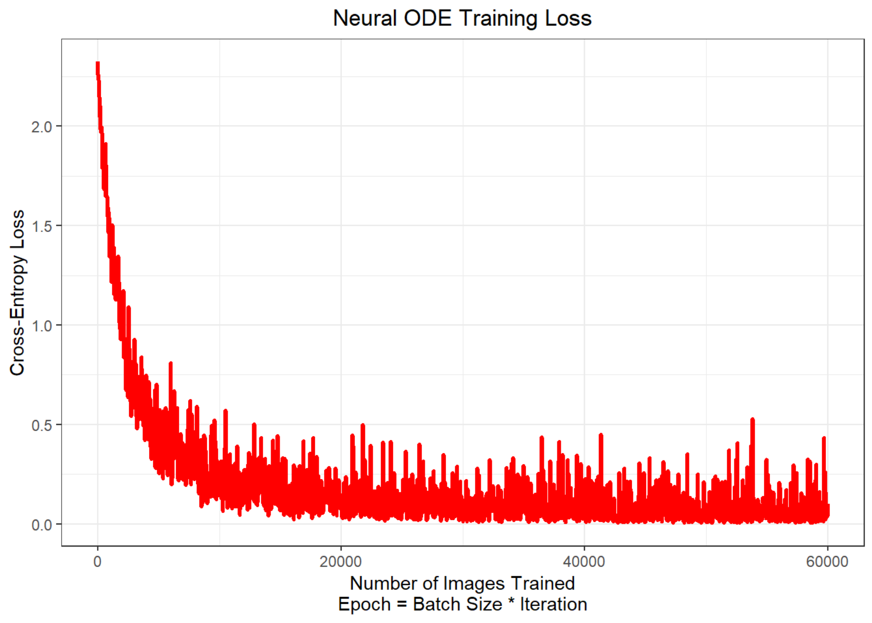

Figure 10: Loss plot for Neural ODE's using one epoch on the MNIST data

5 Conclusions & Future Considerations

In this report, I detailed a through a number of machine learning techniques that have been significant with respect to AI and prediction. Recurrent neural networks were revolutionary in their ability to model non-linear relationships but suffered from problems arising from computational inefficiency. Residual neural networks provided a reasonable solution to the vanishing gradient problem and allowed the training of over 150 layers resulting in exceptional accuracy results.

Neural ordinary differential equations recognized the similarity between the ResNet algorithm and Euler's method and took a step back in terms of the algorithmic process; the methodology proposed allowed for the training of an infinite number of hidden layers and the flexibility of modelling using a differential equation. That is, there was great parameter efficiency that wasn't present in ResNets. More importantly, it is the key insight that neural networks can effectively modelled as differential equations that should be the takeaway.

It seems that the examples given in the paper were limited to an equal number of dimensions between layers - this can be expanded upon. A different dimensionality may contribute to the need of more sophisticated models that are defined for some different numbers of neurons, layer to layer. Expansions could also be made to the realm of functional data analysis where the inputs of the neural network would be sets of functions rather than scalar values. This is an open area of research with plenty of room for creative contributions!

6 References

[1] 3Blue1Brown. *Backpropagation calculus — Deep learning, chapter 4*. Nov. 2017. URL: https://www.youtube.com/watch?v=tIeHLnjs5U8&index=4&list=PLZHQObOWTQDNU6R1_67000Dx_ZCJB-3pi.

[2] A.I. Socratic Circles AISC. *Neural Ordinary Differential Equations - part 1 (algorithm review) AISC*. Feb. 2019. URL: https://www.youtube.com/watch?v=BzTyEJvnyd8.

[3] J.J. Allaire. *TensorFlow for R*. URL: https://tensorflow.rstudio.com/keras/articles/tutorial_basic_classification.html.

[4] Tian Qi Chen et al. «Neural Ordinary Differential Equations». In: *CoRR* abs/1806.07366 (2018). arXiv: 1806.07366. URL: http://arxiv.org/abs/1806.07366.

[5] colesbury. *Batch Normalization Momentum? Issue #695 torchnn*. URL: https://github.com/torch/nn/issues/695.

[6] Angus CS.ai. *Neural Ordinary Differential Equations - Best Paper Awards NeurIPS 2018*. Jan. 2019. URL: https://www.youtube.com/watch?v=V6nGT0Gakyg.

[7] deeplizard. *Vanishing and Exploding Gradient explained*. Mar. 2018. URL: https://www.youtube.com/watch?v=qO_NLVjD6zE.

[8] Firdaouss Doukkali and Firdaouss Doukkali. *Batch normalization in Neural Networks*. Oct. 2017. URL: https://towardsdatascience.com/batch-normalization-in-neural-networks-1ac91516821c.

[9] DrChainsaw. *DrChainsaw/neuralODE4j*. Mar. 2019. URL: https://github.com/DrChainsaw/neuralODE4j.

[10] William E Boyce and Richard C DiPrima. «Elementary differential equations and boundary value problems / William E. Boyce, Richard C. DiPrima». In: *SERBIULA (sistema Librum 2.0)* (Mar. 2019).

[11] Dimitri F. *keras with data augmentation (LB: 0.1826)*. URL: https://www.kaggle.com/dimitrif/keras-with-data-augmentation-lb-0-1826.

[12] Lima Fonseca and Lima Fonseca. *What's happening inside the Convolutional Neural Network? The answer is Convolution.* Nov. 2017. URL: https://buzzrobot.com/whats-happening-inside-the-convolutional-neural-network-the-answer-is-convolution-2c22075dc68d.

[13] Vincent Fung. *An Overview of ResNet and its Variants*. July 2017. URL: https://towardsdatascience.com/an-overview-of-resnet-and-its-variants-5281e2f56035.

[14] Kevin Gibson. *Neural networks as Ordinary Differential Equations*. Dec. 2018. URL: https://rkevingibson.github.io/blog/neural-networks-as-ordinary-differential-equations/.

[15] Pontryagin Mishchenko Boltyanskii Gramkrelidize. *The mathematical theory of optimal processes*.

[16] Kaiming He et al. «Deep Residual Learning for Image Recognition». In: *CoRR* abs/1512.03385 (2015). arXiv: 1512.03385. URL: http://arxiv.org/abs/1512.03385.

[17] *How to build your own Neural Network from scratch in R*. Oct. 2018. URL: https://www.r-bloggers.com/how-to-build-your-own-neural-network-from-scratch-in-r/.

[18] *Interface to 'Python'*. URL: https://rstudio.github.io/reticulate/.

[19] JSeam2. *JSeam2/Neural-Ordinary-Differential-Equations*. Jan. 2019. URL: https://github.com/JSeam2/Neural-Ordinary-Differential-Equations.

[20] kaustav1987. *kaustav1987/Cuda-Error-in-Pytorch*. URL: https://github.com/kaustav1987/Cuda-Error-in-Pytorch/blob/master/dog%20Breed%20Classifier%20-Cuda%20Error.ipynb.

[21] Mandubian. *mandubian/neural-ode*. URL: https://github.com/mandubian/neural-ode/blob/master/tf-neural-ode-v1.0.ipynb.

[22] msurtsukov. *Notebook on nbviewer*. URL: https://nbviewer.jupyter.org/github/urtrial/neural_ode/blob/master/Neural%20ODEs.ipynb.

[23] Ayeshmantha Perera and Ayeshmantha Perera. *What is Padding in Convolutional Neural Network's(CNN's) padding*. Sept. 2018. URL: https://medium.com/@ayeshmanthaperera/what-is-padding-in-cnns-71b21fb0dd7.

[24] Rajat. *Neural Ordinary Differential Equations and Adversarial Attacks*. URL: https://rajatvd.github.io/Neural-ODE-Adversarial/.

[25] Jonty Sinai. *Understanding Neural ODE's*. Jan. 2019. URL: https://jontysinai.github.io/jekyll/update/2019/01/18/understanding-neural-odes.html.

[26] Nitish Srivastava et al. «Dropout: A Simple Way to Prevent Neural Networks from Overfitting». In: *Journal of Machine Learning Research* 15 (2014), pp. 1929–1958. URL: http://jmlr.org/papers/v15/srivastava14a.html.

[27] *Statoil/C-CORE Iceberg Classifier Challenge*. URL: https://www.kaggle.com/c/statoil-iceberg-classifier-challenge/data.

[28] Jerome Friedman Trevor Hastie Robert Tibshirani. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. Springer, 2009.

[29] *Vanishing gradient problem*. Feb. 2019. URL: https://en.wikipedia.org/wiki/Vanishing_gradient_problem.

[30] Wikipedia contributors. *Euler method — Wikipedia, The Free Encyclopedia*. [Online; accessed 26-March-2019]. 2004. URL: https://en.wikipedia.org/w/index.php?title=Plagiarism&oldid=5139350.

7 Appendix

7.1 Neural Network Information

7.1.1 Definitions

Definition 7.1. *Batch Noramlization: is the process of normalizing activations layer to layer in an effort to increase stability and avoid covariate shift.*

Definition 7.2. *Covariate Shift: is a significant change in distribution of new input data. For example, consider an animal classifier trained on black and white images and used on colored images. This color difference is the cause of a covariate shift in the example.* [8]

Definition 7.3. *Mini Batching: An initial step in stochastic gradient descent where the roots of the network are found for only a subset of the data for computational efficiency (and feasibility).*

Definition 7.4. *Momentum: A smoothing factor for the moving mean and variance of the batch normalization process.* [5]

Definition 7.5. *Dense Layer: A dense layer is one that takes a linear combination of all activations from the previous layer for each neuron in its layer.*

Definition 7.6. *Units: The number of neurons present in some layer i of the neural network. The input layer will have units == p where p is the number of input variables.*

7.1.2 Neural Network Code

This is the implementation from scratch:

```r
### Setting seed
set.seed(25)

## Defining data frame
x <- rnorm(100)
y <- ifelse(x >= -0.5 & x <= 0.5, 1, 0)
gaussian_df <- sample(data.frame(rn = x, resp = y))

## Poisoning the data
#gaussian_df[sample(which(gaussian_df$resp == 1), 25),] = 0

## Looking at data set
head(gaussian_df)

## Activation Function
sigmoid <- function(x) {
  return(1.0/(1.0 + exp(-x)))
}

## Derivative of the activation
sigmoid_deriv <- function(x) {
  return(x*(1.0 - x))
```

```
}

## Loss Function
MSE <- function(neural_net) {
  return(mean((neural_net$y - round(neural_net$output))^2))
}

## Initializing
layer_weights_1 <- c(runif(length(gaussian_df$rn)))
layer_weights_2 <- c(runif(length(gaussian_df$rn)))
layer_bias_1 <- c(runif(length(gaussian_df$rn)))
layer_bias_2 <- c(runif(length(gaussian_df$rn)))

## Setting up neural network list
neuralnet_info <- list(
  input = gaussian_df$rn,
  layer_weights_1 = layer_weights_1,
  layer_bias_1 = layer_bias_1,
  layer_weights_2 = layer_weights_2,
  layer_bias_2 = layer_bias_2,
  y = gaussian_df$resp,
  output = matrix(rep(0, 1000), ncol = 1)
)

## Forward pass
forward_pass <- function(neural_net) {

  # Layer 1 activations
  neural_net$layer1 <- c(sigmoid(neural_net$input * neural_net$layer_weights_1 +
                                  layer_bias_1))

  # Output activations
  neural_net$output <- c(sigmoid(neural_net$layer1 * neural_net$layer_weights_2 +
                                  layer_bias_2))

  return(neural_net)
}

## Backpropagation
grad_descent <- function(neural_net){

  ## Easier derivative first
  # weights closer to the output layer
  deriv_weights2 <- (
    neural_net$layer1*(2*(neural_net$y - neural_net$output)*sigmoid_deriv(neural_net$
        output))
  )

  ## Backpropagating to first layer
  # Applied chain rule here
  deriv_weights1 <- (2*(neural_net$y - neural_net$output)*sigmoid_deriv(neural_net$output
      ))*neural_net$layer_weights_2
  deriv_weights1 <- deriv_weights1*sigmoid_deriv(neural_net$layer1)
  deriv_weights1 <- neural_net$input*deriv_weights1

  ## Now need to do bias derivatives
  deriv_bias2 <- 2*(neural_net$y - neural_net$output)*sigmoid_deriv(neural_net$output)
  deriv_bias1 <- 2*(neural_net$y - neural_net$output)*sigmoid_deriv(neural_net$output)*
      layer_weights_2*sigmoid_deriv(neural_net$layer1)

  # Weight update using derivative
  learn_rate = 1
  neural_net$layer_weights_1 <- neural_net$layer_weights_1 + learn_rate*deriv_weights1
  neural_net$layer_weights_2 <- neural_net$layer_weights_2 + learn_rate*deriv_weights2
  neural_net$layer_bias_1 <- neural_net$layer_bias_1 + learn_rate*deriv_bias1
  neural_net$layer_bias_2 <- neural_net$layer_bias_2 + learn_rate*deriv_bias2

  # Returning updated information
  return(neural_net)

}

## Error Rate after no iterations
mean(round(neuralnet_info$output) == gaussian_df$resp)

## Epochs
epoch_num <- 50

## Initializing loss vector
lossData <- data.frame(epoch = 1:epoch_num, MSE = rep(0, epoch_num))

## Training Neural Net
for (i in 1:epoch_num) {

  # Foreward iteration
  neuralnet_info <- forward_pass(neuralnet_info)

  # Backward iteration
  neuralnet_info <- grad_descent(neuralnet_info)

  # Storing loss
  lossData$MSE[i] <- MSE(neuralnet_info)

}

## Error Rate after 50 iterations
mean(round(neuralnet_info$output) == gaussian_df$resp)

## Plotting Loss
lossData %>%
  ggplot(aes(x = epoch, y = MSE)) +
  geom_line(size = 1.25, color = "red") +
  theme_bw() +
  labs(x = "Epoch #", y = "MSE") +
  ggtitle("Change in Loss - Simple Neural Net") +
  theme(plot.title = element_text(hjust = 0.5))
```

This is the **keras** implementations:

```
### Final NN Code Using Iceberg Dataset

## Libraries
library(RJSONIO)
library(keras)
library(abind)
library(kohonen)
library(tidyr)
library(ggplot2)

## Setting seed
set.seed(1)

## Reading in dataset

# Iceberg data
train = fromJSON("train.json")

# Getting relevant information
x <- train %>%
  lapply(function(x){c(x$band_1, x$band_2)}) %>%
  unlist %>%
  array(dim=c(75,75,1604)) %>%
  aperm(c(3,1,2))

# Values for Output
y <- classvec2classmat(unlist(lapply (train, function(x) {x$is_iceberg})))

# Training Set list
nums <- sample(1:1604, 1300)

# Organizing
train_iceberg <- x[nums, , ]
train_truth <- y[nums, 2]
test_iceberg <- x[-nums, , ]
test_truth <- y[-nums, 2]

# Class Names
iceberg_name <- c("Not an Iceberg", "An Iceberg")

## Need to scale data
train_iceberg <- train_iceberg/max(abs(train_iceberg))
test_iceberg <- test_iceberg/max(abs(train_iceberg))

## Looking at the first 25 images
par(mfcol=c(5,5))
par(mar=c(0, 0, 1.5, 0), xaxs='i', yaxs='i')
for (i in 1:25) {
  img <- train_iceberg[i, , ]
  img <- t(apply(img, 2, rev))
  image(1:75, 1:75, img, col = gray((-44:0)/-44), xaxt = 'n', yaxt = 'n',
        main = paste(iceberg_name[train_truth[i] + 1]))
}

#### Creating model

# Initialization
iceberg_nn <- keras_model_sequential()

# Adding Layers
iceberg_nn %>%
  layer_flatten(input_shape = c(75, 75)) %>% # Turning image into 784 input variables
  layer_dense(units = 128, activation = 'relu') %>% # 128 neurons with relu activation,
      HL1
  layer_dense(units = 128, activation = 'relu') %>% # 128 neurons with relu activation,
      HL2
  layer_dense(units = 128, activation = 'relu') %>% # 128 neurons with relu activation,
      HL3
  layer_dense(units = 128, activation = 'relu') %>% # 128 neurons with relu activation,
      HL4
  layer_dense(units = 1, activation = 'sigmoid') # Output layer: 1 of 10 things with
      softmax
# activation function

## Densely connected means FULLY-CONNECTED (EACH NEURON IS INVOLVED IN THE CALCULATION OF
# EVERY SINGLE NEURON IN THE NEXT LAYER)

## Adding loss function and optimizer
iceberg_nn %>% compile(
  optimizer = 'sgd', # Using stochastic gradient descent as backprop method
  loss = 'binary_crossentropy', # Using cross-entropy as loss evaluator
  metrics = c('accuracy') # Looking at accuracy
)

## Fitting the model
iceberg_nn %>% fit(train_iceberg, train_truth, epochs = 150)

## Seeing the accuracy
```

```
score <- iceberg_nn %>% evaluate(test_iceberg, test_truth)

cat('Test loss:', score$loss, "\n")
cat('Test accuracy:', score$acc, "\n")
```

7.2 Residual Neural Network Information

7.2.1 Definitions

Definition 7.7. *Pooling: reduces the resolution of the feature map but retains particularities of the map required for classification through translational and rotational invariants.*

Definition 7.8. *Dropout: is a regularization technique developed by google to prevent overfitting. The process involves the prevention of "learning" complex patterns within training data.* [26]

Definition 7.9. *Activation-Elu: An exponential linear unit: $f(x) = \alpha \cdot (exp(x) - 1.0)$.*

Definition 7.10. *Padding: is an additional layer added to act on the border of an image suppressing pixels with less information.* [23]

Definition 7.11. *Kernels: are a matrix transformation that change the input image to some variatn of it (blur, blacken, sharpen, etc.)* [12]

7.2.2 Residual Neural Network Code

```
### RESNET Final Code

## Libraries
library(RJSONIO)
library(keras)
library(abind)
library(kohonen)
library(tidyr)
library(ggplot2)

## Loading data
train = fromJSON("train.json")

# Getting relevant information
x = train %>% lapply(function(x){
  c(x$band_1,
    x$band_2,
    apply(cbind(x$band_1,x$band_2), 1, mean))}) %>%
  unlist %>%
  array(dim=c(75,75,3,1604)) %>%
  aperm(c(4,1,2,3))

# Values for Output
y <- classvec2classmat(unlist(lapply (train, function(x) {x$is_iceberg})))

# Training Set list
nums <- sample(1:1604, 1300)

# Organizing
train_iceberg <- x[nums, , , ]
```

```r
train_truth <- y[nums, ]
test_iceberg <- x[-nums, , , ]
test_truth <- y[-nums, ]

## Prepare model
kernel_size = c(5,5)
input_img = layer_input(shape = c(75, 75, 3), name="img")

## Normalizing data
input_img_norm = input_img %>%
  layer_batch_normalization(momentum = 0.99)

## input CNN
input_CNN = input_img_norm %>%
  layer_conv_2d(32, kernel_size = kernel_size, padding = "same") %>%
  layer_batch_normalization(momentum = 0.99) %>%
  layer_activation_elu() %>%
  layer_max_pooling_2d(c(2,2)) %>%
  layer_dropout(0.25) %>%
  layer_conv_2d(64, kernel_size = kernel_size,padding = "same") %>%
  layer_batch_normalization(momentum = 0.99) %>%
  layer_activation_elu() %>%
  layer_max_pooling_2d(c(2,2)) %>%
  layer_dropout(0.25)

## first residual
input_CNN_residual = input_CNN %>%
  layer_batch_normalization(momentum = 0.99) %>%
  layer_conv_2d(128, kernel_size = kernel_size,padding = "same") %>%
  layer_batch_normalization(momentum = 0.99) %>%
  layer_activation_elu() %>%
  layer_dropout(0.25) %>%
  layer_conv_2d(64, kernel_size = kernel_size,padding = "same") %>%
  layer_batch_normalization(momentum = 0.99) %>%
  layer_activation_elu()

input_CNN_residual = layer_add(list(input_CNN_residual,input_CNN))

# ## second residual
input_CNN_residual = input_CNN_residual %>%
  layer_batch_normalization(momentum = 0.99) %>%
  layer_conv_2d(128, kernel_size = kernel_size,padding = "same") %>%
  layer_batch_normalization(momentum = 0.99) %>%
  layer_activation_elu() %>%
  layer_dropout(0.25) %>%
  layer_conv_2d(64, kernel_size = kernel_size,padding = "same") %>%
  layer_batch_normalization(momentum = 0.99) %>%
  layer_activation_elu()

input_CNN_residual = layer_add(list(input_CNN_residual,input_CNN))

## final CNN
top_CNN = input_CNN_residual %>%
  layer_conv_2d(128, kernel_size = kernel_size,padding = "same") %>%
  layer_batch_normalization(momentum = 0.99) %>%
  layer_activation_elu() %>%
  layer_max_pooling_2d(c(2,2)) %>%
  layer_conv_2d(256, kernel_size = kernel_size,padding = "same") %>%
  layer_batch_normalization(momentum = 0.99) %>%
  layer_activation_elu() %>%
  layer_dropout(0.25) %>%
  layer_max_pooling_2d(c(2,2)) %>%
  layer_conv_2d(512, kernel_size = kernel_size,padding = "same") %>%
  layer_batch_normalization(momentum = 0.99) %>%
  layer_activation_elu() %>%
  layer_dropout(0.25) %>%
  layer_max_pooling_2d(c(2,2)) %>%
  layer_global_max_pooling_2d()

## Output layer
outputs = top_CNN %>%
  layer_dense(512,activation = NULL) %>%
  layer_batch_normalization(momentum = 0.99) %>%
  layer_activation_elu() %>%
  layer_dropout(0.5) %>%
  layer_dense(256,activation = NULL) %>%
  layer_batch_normalization(momentum = 0.99) %>%
  layer_activation_elu() %>%
  layer_dropout(0.5) %>%
  layer_dense(2,activation = "softmax") ## not sure using softmax is the right thing to
     do...

## Setting up model
model_resNN <- keras_model(inputs = list(input_img), outputs = list(outputs))

## Setting up functions for model evaluation and passes
model_resNN %>% compile(optimizer = optimizer_adam(lr = 0.001),
                  loss="binary_crossentropy",
                  metrics = c("accuracy"))

## Fitting the model
model_resNN %>% fit(train_iceberg, train_truth, epochs = 150)

## Trying on test data
predictions_resnet <-  mean(round(predict(model_resNN, test_iceberg))[,2] == test_truth
    [,2])
paste("Test Accuracy (ResNN):", predictions_resnet)
```

7.3 Neural ODE Information

7.3.1 Neural ODE Code

This code needs to be run using reticulate in a python code chunk (within markdown).

```
# Loading some packages
library(tidyverse)
library(reticulate)
use_virtualenv("r-reticulate")
py_available(TRUE)

# Here, first loaded are some dependencies
# These libraries range from the deep learning architectures
# required for the neural ODE to work (such as torch) and
# more essential libraries like math and numpy for
# ODE and array operations; the matplotlab library is for graphics
# purposes and the pandas library is for data frame manipulation
# Cude allows access to GPU use

##############
import math
import numpy as np
from IPython.display import clear_output
from tqdm import tqdm_notebook as tqdm

import matplotlib as mpl
import matplotlib.pyplot as plt
import seaborn as sns
```

```
sns.color_palette("bright")
import matplotlib as mpl
import matplotlib.cm as cm
import pandas as pd

import torch
from torch import Tensor
from torch import nn
from torch.nn  import functional as F
from torch.autograd import Variable

import torchvision

use_cuda = torch.cuda.is_available()
##############

# Next, here is the general ODE solve function we will use in the
# forward pass later on. Euler's method is used here because it is
# easy to implement - the step size is 0.05 (thus separating it
# from ResNets)

##############
def ode_solve(z0, t0, t1, f):
    """
    Simplest Euler ODE initial value solver
    """
    h_max = 0.05
    n_steps = math.ceil((abs(t1 - t0)/h_max).max().item())

    h = (t1 - t0)/n_steps
    t = t0
    z = z0

    for i_step in range(n_steps):
        z = z + h * f(z, t)
        t = t + h
    return z
##############

# This function computes the derivatives required in the
# forward pass and reduces the number of parameters with the
# flatten parameters function. Flattening lowers the "denseness"
# of your model layer to layer - more on this in the final report
class ODEF(nn.Module):
    def forward_with_grad(self, z, t, grad_outputs):
        """Compute f and a df/dz, a df/dp, a df/dt"""
        batch_size = z.shape[0]

        out = self.forward(z, t)

        a = grad_outputs
        adfdz, adfdt, *adfdp = torch.autograd.grad(
            (out,), (z, t) + tuple(self.parameters()), grad_outputs=(a),
            allow_unused=True, retain_graph=True
        )
        # grad method automatically sums gradients for batch items, we have to expand
            them back
        if adfdp is not None:
            adfdp = torch.cat([p_grad.flatten() for p_grad in adfdp]).unsqueeze(0)
            adfdp = adfdp.expand(batch_size, -1) / batch_size
        if adfdt is not None:
            adfdt = adfdt.expand(batch_size, 1) / batch_size
        return out, adfdz, adfdt, adfdp

    def flatten_parameters(self):
        p_shapes = []
        flat_parameters = []
        for p in self.parameters():
            p_shapes.append(p.size())
            flat_parameters.append(p.flatten())
        return torch.cat(flat_parameters)
##############

# Here, this is the adjoint call of the method. Remember, this is used
# in the backward pass and this is defined here as well along with the
# augmented dynamics. Moreover, the integrals in the backward trajectory
# of the backpropagation process are computed over here. The exact
# mathematical details of the "augmented" state, I am still trying to
# work out. I have more in the final report but for now, take this to be
# the funky source code that it is!

##############
class ODEAdjoint(torch.autograd.Function):
    @staticmethod
    def forward(ctx, z0, t, flat_parameters, func):
        assert isinstance(func, ODEF)
        bs, *z_shape = z0.size()
        time_len = t.size(0)

        with torch.no_grad():
            z = torch.zeros(time_len, bs, *z_shape).to(z0)
            z[0] = z0
            for i_t in range(time_len - 1):
                z0 = ode_solve(z0, t[i_t], t[i_t+1], func)
                z[i_t+1] = z0

        ctx.func = func
        ctx.save_for_backward(t, z.clone(), flat_parameters)
        return z

    @staticmethod
    def backward(ctx, dLdz):
        """
        dLdz shape: time_len, batch_size, *z_shape
        """
        func = ctx.func
        t, z, flat_parameters = ctx.saved_tensors
        time_len, bs, *z_shape = z.size()
        n_dim = np.prod(z_shape)
        n_params = flat_parameters.size(0)

        # Dynamics of augmented system to be calculated backwards in time
        def augmented_dynamics(aug_z_i, t_i):
            """
            tensors here are temporal slices
            t_i - is tensor with size: bs, 1
            aug_z_i - is tensor with size: bs, n_dim*2 + n_params + 1
            """
            z_i, a = aug_z_i[:, :n_dim], aug_z_i[:, n_dim:2*n_dim]  # ignore parameters
                and time

            # Unflatten z and a
            z_i = z_i.view(bs, *z_shape)
            a = a.view(bs, *z_shape)
            with torch.set_grad_enabled(True):
                t_i = t_i.detach().requires_grad_(True)
                z_i = z_i.detach().requires_grad_(True)
                func_eval, adfdz, adfdt, adfdp = func.forward_with_grad(z_i, t_i, grad_
                    outputs=a)  # bs, *z_shape
                adfdz = adfdz.to(z_i) if adfdz is not None else torch.zeros(bs, *z_shape)
                    .to(z_i)
                adfdp = adfdp.to(z_i) if adfdp is not None else torch.zeros(bs, n_params)
```

```
                    .to(z_i)
                adfdt = adfdt.to(z_i) if adfdt is not None else torch.zeros(bs, 1).to(z_i)

            # Flatten f and adfdz
            func_eval = func_eval.view(bs, n_dim)
            adfdz = adfdz.view(bs, n_dim)
            return torch.cat((func_eval, -adfdz, -adfdp, -adfdt), dim=1)

        dLdz = dLdz.view(time_len, bs, n_dim)  # flatten dLdz for convenience
        with torch.no_grad():
            ## Create placeholders for output gradients
            # Prev computed backwards adjoints to be adjusted by direct gradients
            adj_z = torch.zeros(bs, n_dim).to(dLdz)
            adj_p = torch.zeros(bs, n_params).to(dLdz)
            # In contrast to z and p we need to return gradients for all times
            adj_t = torch.zeros(time_len, bs, 1).to(dLdz)

            for i_t in range(time_len-1, 0, -1):
                z_i = z[i_t]
                t_i = t[i_t]
                f_i = func(z_i, t_i).view(bs, n_dim)

                # Compute direct gradients
                dLdz_i = dLdz[i_t]
                dLdt_i = torch.bmm(torch.transpose(dLdz_i.unsqueeze(-1), 1, 2), f_i.unsqueeze(-1))[:, 0]

                # Adjusting adjoints with direct gradients
                adj_z += dLdz_i
                adj_t[i_t] = adj_t[i_t] - dLdt_i

                # Pack augmented variable
                aug_z = torch.cat((z_i.view(bs, n_dim), adj_z, torch.zeros(bs, n_params).to(z), adj_t[i_t]), dim=-1)

                # Solve augmented system backwards
                aug_ans = ode_solve(aug_z, t_i, t[i_t-1], augmented_dynamics)

                # Unpack solved backwards augmented system
                adj_z[:] = aug_ans[:, n_dim:2*n_dim]
                adj_p[:] += aug_ans[:, 2*n_dim:2*n_dim + n_params]
                adj_t[i_t-1] = aug_ans[:, 2*n_dim + n_params:]

                del aug_z, aug_ans

            ## Adjust 0 time adjoint with direct gradients
            # Compute direct gradients
            dLdz_0 = dLdz[0]
            dLdt_0 = torch.bmm(torch.transpose(dLdz_0.unsqueeze(-1), 1, 2), f_i.unsqueeze(-1))[:, 0]

            # Adjust adjoints
            adj_z += dLdz_0
            adj_t[0] = adj_t[0] - dLdt_0
        return adj_z.view(bs, *z_shape), adj_t, adj_p, None
##############

# Next, the code is all bunched up nicely into a class NeuralODE
# This means that the previous classes all act as dependencies for
# this class. The previous classes will be called upon when this
# code is run. There is not much else to say here other than
# this is just a compacting of everything defined thus far

##############
class NeuralODE(nn.Module):
    def __init__(self, func):
        super(NeuralODE, self).__init__()
        assert isinstance(func, ODEF)
        self.func = func

    def forward(self, z0, t=Tensor([0., 1.]), return_whole_sequence=False):
        t = t.to(z0)
        z = ODEAdjoint.apply(z0, t, self.func.flatten_parameters(), self.func)
        if return_whole_sequence:
            return z
        else:
            return z[-1]
##############

# Here, we get batch normalization (defined in the final report)

##############
def norm(dim):
    return nn.BatchNorm2d(dim)
##############

# Next, we find a convolutional block. This is similar to the ResNet
# code. It's simply defining a convolutional Neural Net

##############
def conv3x3(in_feats, out_feats, stride=1):
    return nn.Conv2d(in_feats, out_feats, kernel_size=3, stride=stride, padding=1, bias=
        False)
##############

# Here, the code returns some relevant information about
# the process thus far. The first line ppulls out the
# dimensions of the tensor image and the cat function
# from torch simple puts together the results

##############
def add_time(in_tensor, t):
    bs, c, w, h = in_tensor.shape
    return torch.cat((in_tensor, t.expand(bs, 1, w, h)), dim=1)
##############

# These next two classes embed a neural ODE into a convolutional
# neural network. This is analgous to the Residual blocks being embedded
# in the convolutional neural network in the ResNet Secion III. The
# options for the convolutional blocks are similar to that of the
# R keras counterparts (number of neurons, kernel sizes, Relu activation, etc)
class ConvODEF(ODEF):
    def __init__(self, dim):
        super(ConvODEF, self).__init__()
        self.conv1 = conv3x3(dim + 1, dim)
        self.norm1 = norm(dim)
        self.conv2 = conv3x3(dim + 1, dim)
        self.norm2 = norm(dim)

    def forward(self, x, t):
        xt = add_time(x, t)
        h = self.norm1(torch.relu(self.conv1(xt)))
        ht = add_time(h, t)
        dxdt = self.norm2(torch.relu(self.conv2(ht)))
        return dxdt

class ContinuousNeuralMNISTClassifier(nn.Module):
    def __init__(self, ode):
        super(ContinuousNeuralMNISTClassifier, self).__init__()
        self.downsampling = nn.Sequential(
            nn.Conv2d(1, 64, 3, 1),
            norm(64),
            nn.ReLU(inplace=True),
            nn.Conv2d(64, 64, 4, 2, 1),
            norm(64),
            nn.ReLU(inplace=True),
            nn.Conv2d(64, 64, 4, 2, 1),
        )
        self.feature = ode
        self.norm = norm(64)
        self.avg_pool = nn.AdaptiveAvgPool2d((1, 1))
        self.fc = nn.Linear(64, 10)

    def forward(self, x):
        x = self.downsampling(x)
        x = self.feature(x)
        x = self.norm(x)
        x = self.avg_pool(x)
        shape = torch.prod(torch.tensor(x.shape[1:])).item()
        x = x.view(-1, shape)
        out = self.fc(x)
        return out
################

################
func = ConvODEF(64)
ode = NeuralODE(func)
model = ContinuousNeuralMNISTClassifier(ode)
if use_cuda:
    model = model.cuda()
################

# Here, the MNIST training data is loaded and normalized
# using the prespecified mean and standard deviation. This is
# a standard pre-processing in most neural net implementations
# as can be seen in my previous implementations

################  v
img_std = 0.3081
img_mean = 0.1307

batch_size = 32
train_loader = torch.utils.data.DataLoader(
    torchvision.datasets.MNIST("data/mnist", train=True, download=True,
                             transform=torchvision.transforms.Compose([
                                 torchvision.transforms.ToTensor(),
                                 torchvision.transforms.Normalize((img_mean,), (img_std,)
                                     )
                             ])
    ),
    batch_size=batch_size, shuffle=True
)

test_loader = torch.utils.data.DataLoader(
    torchvision.datasets.MNIST("data/mnist", train=False, download=True,
                             transform=torchvision.transforms.Compose([
                                 torchvision.transforms.ToTensor(),
                                 torchvision.transforms.Normalize((img_mean,), (img_std,)
                                     )
                             ])
    ),
    batch_size = 128, shuffle=True
)
################

# Here the optimizer is defined

```

```
################
optimizer = torch.optim.Adam(model.parameters())
################

# Now, this is where the training is done and the functions
# previously defined are called. The train and test functions
# are for the separate outputs. The loss function is used
# here as well with the "criterion" function. This is a call to
# cross-entropy function. The loss results are ultimately
# returned in the final outputs

################
def train(epoch):
    num_items = 0
    train_losses = []

    model.train()
    criterion = nn.CrossEntropyLoss()
    print(f"Training Epoch {epoch}...")
    for batch_idx, (data, target) in tqdm(enumerate(train_loader), total=len(train_loader
        )):
        if use_cuda:
            data = data.cuda()
            target = target.cuda()
        optimizer.zero_grad()
        output = model(data)
        loss = criterion(output, target)
        loss.backward()
        optimizer.step()

        train_losses += [loss.item()]
        num_items += data.shape[0]
    print('Train loss: {:.5f}'.format(np.mean(train_losses)))
    return train_losses


def test():
    accuracy = 0.0
    num_items = 0

    model.eval()
    criterion = nn.CrossEntropyLoss()
    print(f"Testing...")
    with torch.no_grad():
        for batch_idx, (data, target) in tqdm(enumerate(test_loader),  total=len(test_
            loader)):
            if use_cuda:
                data = data.cuda()
                target = target.cuda()
            output = model(data)
            accuracy += torch.sum(torch.argmax(output, dim=1) == target).item()
            num_items += data.shape[0]
    accuracy = accuracy * 100 / num_items
    print("Test Accuracy: {:.3f}%".format(accuracy))
################

# Next, here is some initialization and the number of epochs is defined

################
n_epochs = 1
test()
train_losses = []
################

# Finally, everything above is called and run

```

```
################
for epoch in range(1, n_epochs + 1):
    train_losses += train(epoch)
    test()
################

# The loss results are pulled out in the form of a CSV (using pandas)

################
loss_data = pd.DataFrame({"loss": train_losses})
loss_data["Trained_Images"] = loss_data.index * batch_size
loss_data["Halflife_Loss"] = loss_data.loss.ewm(halflife=10).mean()
loss_data.to_csv('neural_ode_loss.csv')
################

# Plotting
# Reading in loss results from python
neuralODELoss = read.csv("neural_ode_loss.csv", header = T)

# Plotting
neuralODELoss %>%
  ggplot(aes(x = Trained_Images, y = loss)) +
  geom_line(color = "red", size = 1.1) +
  theme_bw() +
  labs(x = "Number of Images Trained\nEpoch = Batch Size * Iteration", y = "Cross-Entropy
       Loss") +
  ggtitle("Neural ODE Training Loss") +
  theme(plot.title = element_text(hjust = 0.5))
```

7.4 Differential Equations Primer

Discussion in this section will be limited to first order ordinary differential equations. The purpose is to instill enough understanding so that their relevance in Section IV is apparent and clear.

7.4.1 General Methodology

Generally, a differential equation relates the values of some function to the values of its derivatives. A first order differential equation is limited to the relationship between a single derivative of a single variable. They are of the form:

$$\frac{dy}{dx} = f(x, y) \tag{22}$$

The function $f(x, y)$ is any of the set of functions which is defined for x (the independent variable) and y (the dependent variable). Accompanying the equation is usually an initial condition which defines the behaviour of the function at some point, x_0[24]. It is sometimes possible to find analytic solutions to differential equations provided they are of a particular form, for example:

$$g(y)\frac{dy}{dx} = f(x), \quad y(x_0) = y_0 \tag{23}$$

[24]The value here is sometimes apparent from the context; for example, consider half-life models in which you know the amount present at time, $t = 0$

But, in general, differential equations are solved numerically[25]. A method falling under the umbrella of numeric methods is presented in the next sub-section.

Let's consider the following ODE:

$$\frac{dy}{dx} + \frac{y}{2} = \frac{3}{2}, \quad y(0) = 2 \tag{24}$$

This differential equation can be solved analytically as follows:

$$\frac{dy}{3-y} = \frac{dx}{2} \tag{25}$$

$$\int_2^y \frac{dy}{3-y} = \frac{1}{2}\int_0^x dx \tag{26}$$

$$\ln(3-y) = -\frac{1}{2} \tag{27}$$

$$3 - y = \exp\{-\frac{x}{2}\} \tag{28}$$

$$y = 3 - \exp\{-\frac{x}{2}\} \tag{29}$$

The solution of the differential equation depends on the initial conditions provided however, the critical points of the function will be clear in any of them. In the above example, when $y = 3$, the derivative is 0 and hence we would expect a horizontal asymptote for any provided initial condition at this value. This behaviour is presented in Figure 11.

Another important visualization tool for differential equations is the phase portrait. The phase portrait allows us to discern important information about the original function, $f(x)$ without actually solving the differential equation. The phase portrait involves computing the roots of the function (the 0's of the derivative) and plotting the behaviour of the derivative for various values of the dependent variable, y. Consider the following differential equation [10]:

$$\frac{dy}{dt} = r(1 - \frac{y}{K})y \tag{30}$$

Where $K = \frac{r}{a}$ and r is known as an 'intrinsic growth rate'[26]. The first step in identifying the phase portrait is to find the 0's; in this case, if we let $y = f(x) = \{0 \cap K\}$, then the value of $\frac{dy}{dt}$ in (13) is 0 - these are known as the **equilibrium solutions**. This is when there is no change in the *variation* of y as t changes. From there, we

[25]The equation in (6) is known as a separable differential equation because you can split the $f(x, y)$ in (5) into two separate functions

[26]This equation is an extension on the exponential growth function and is commonly referred to as the Verhulst or logistic equation

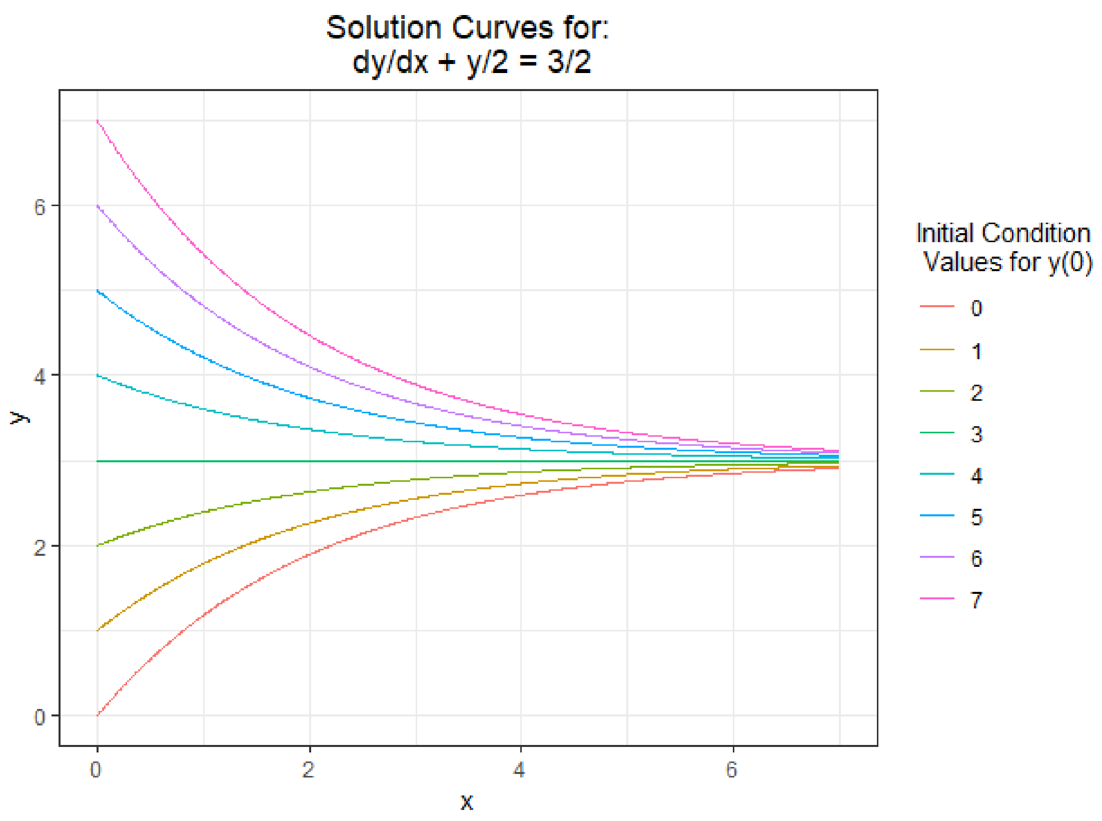

Figure 11: Solution curves for the ODE given in (7). The particular initial condition that solved for is given by the sea green curve

can observe the behaviour of the derivative around these equilibrium points.

In Figure 12, the arrows point to the right for positive derivative values and to the left otherwise. You can imagine these arrows as directions for convergence. The **stable equilibrium** is some value that the system being modelled tends to (also known as a saturation level). Intuitively, imagine you were modelling population growth; then, you would expect quicker growth the greater the population size however, at some point resources become limited. This results in a decrease (or stabilization) of the population at some level which, in Figure 12, is the second equilibrium point. The unstable equilibrium in this example, occurs when the population is 0 - we expect no growth if no one is around however, as soon as it is possible to move away from this position, we tend to do so.

7.4.2 A Numeric Approximation: Euler Method

In the previous section, I introduced differential equations, an example of an analytic solution, and visualization tool to glean information about the function $y(x)$ without actually solving the equation; here, I introduce a numerical approach to solving differential equations: Euler method[27]

[27]This approach is a specification of a more general approach known as Runge-Kutta

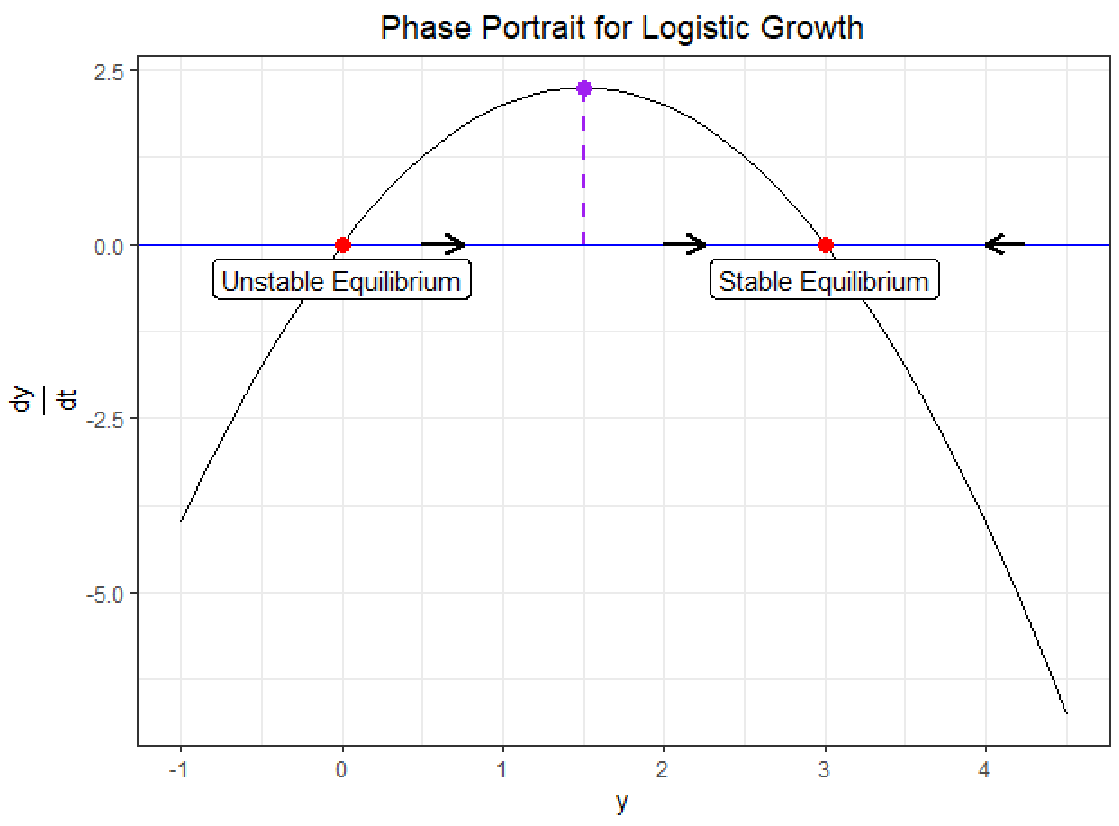

Figure 12: A phase portrait for the Verhulst Equation defined in (13)

Euler's method is an iterative approach that, for some change in x, provides an estimate of the function, $f(x)$ using the derivative in the interval, $\Delta(x)$. In order to make this more concrete, consider the differential equation [30]:

$$\frac{dy}{dx} = y, \qquad y(0) = 1 \tag{31}$$

The solution, $f(x)$ to this equation is $y = \exp\{x\}$. However, let's assume that we didn't have the means to find the analytic solution and instead, use Euler's method to approximate $f(x)$. Let $\Delta(x) = 1$ be the iterative interval and consider $x = [0, 4]$. At $x = 0$, we have $y = 1$. The derivative at this point is also 0. Then, using the iterative process: $y_{i+1} = y_i + \Delta(x)\dfrac{dy}{dx}$, we see that $y_1 = 1 + 1 \cdot 0$ and, redoing this process, we get the following:

Iteration	x	y	$\frac{dy}{dx}$
0	0	1	1
1	1	2	2
2	2	4	4
3	3	8	8
4	4	16	16

Essentially, we are figuring out the tangent lines for intervals and connecting them - this is our approximation! Note that, as $\Delta(x) \to 0$, our approximation approaches the exact solution. A visualization is provided in Figure 13.

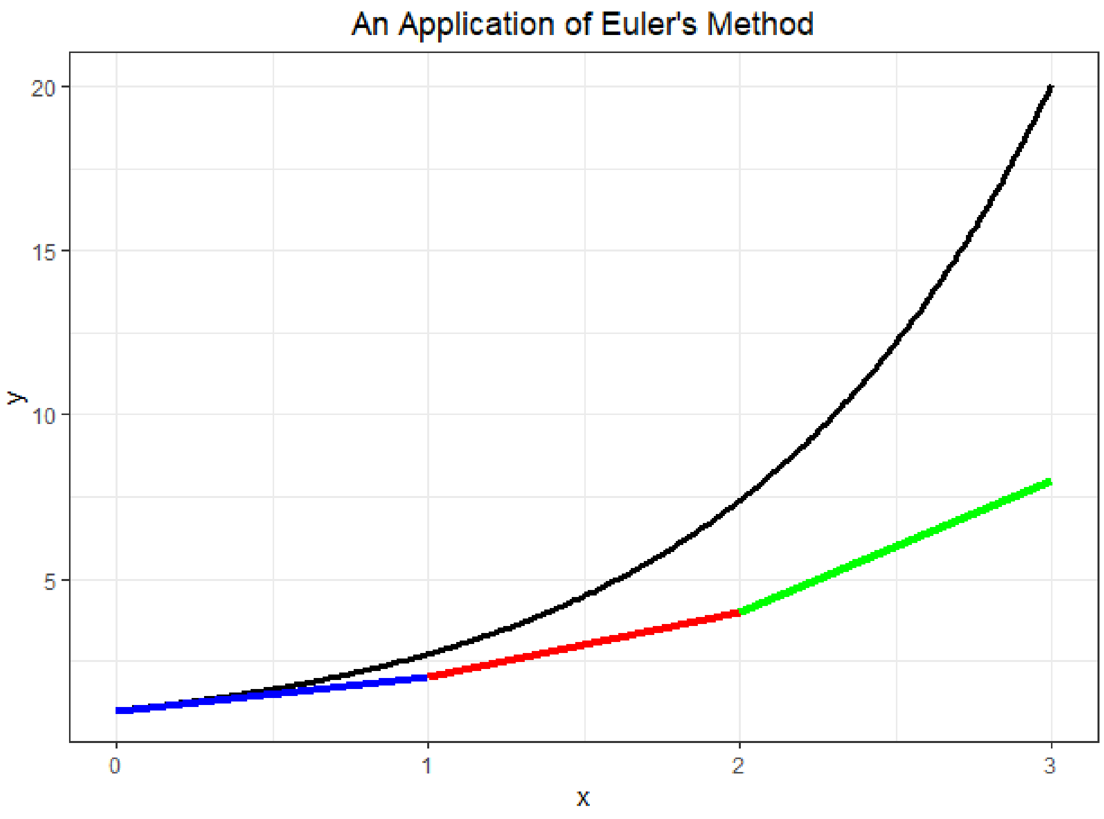

Figure 13: Euler method for the differential equation in (31). The colored segments represent the Euler approximation with a step size of 1. The black curve is the true function: $y = \exp\{x\}$

This brings an end to the primer. The approach in Euler's method is particularly important when comparing ResNets to NeuralODEs in the next section!

7.5 On the relationship between ResNets and Euler's Method

Consider again the general transformation performed in ResNets[28] [24]:

$$a_1 = f_0(x) + x \tag{32}$$
$$a_2 = f_1(a_1) + a_1 \tag{33}$$
$$a_3 = f_2(a_2) + a_2 \tag{34}$$

Rearranging these, we can observe that this, is almost exactly the form of Euler's method! Letting $a(t = 0) = x$,

[28]In the main section, $z^{(1)} = a_1$

$$a(1) - a(0) = f(a(0), t = 0) \tag{35}$$
$$a(2) - a(1) = f(a(1), t = 1) \tag{36}$$
$$... \tag{37}$$

Recall that Euler's method is a descritization dependent on the step size, Δh. In essence, the key insight of the ResNets approach is characterized by a rearranging of Euler's method. If that is the case, then we are essentially solving a differential equation. A Neural ODE takes another step backwards in the process by appealing to the fundamental equation underlying the neural net rather than looking at the intermediary step that ResNets do.