Spring 3-3-2019

Ordinary Differential Equation NeuralNetworks: Mathematics and Application using Diffeqflux.jl

Muhammad Moiz Saeed
msaeed@arcadia.edu

Ordinary Differential Equation Neural Networks: Mathematics and Application using Diffeqflux.jl

Muhammad Moiz Saeed
Arcadia University
Glenside, Pennsylvania 19095 USA

August 8, 2019

Abstract

This paper has two objectives.

1. It simplifies the Mathematics behind a simple Neural Network. Furthermore it explores how Neural Networks can be modeled using Ordinary Differential Equations(ODE).
2. It implements a simple example of an ODE Neural network using diffeqflux.jl library.

My paper is based on the paper "Neural Ordinary Differential equations"[1] paper and contains multiple extracts from this paper and hence the work in chapter 4 should not be considered original work as it aims to explain the mathematics in the original paper and all credit is due to the authors of the paper [1]. This paper[1] was among on the 5 papers to be recognized at the 2018 annual conference NeurIPS(Neural Information Processing Systems).

Contents

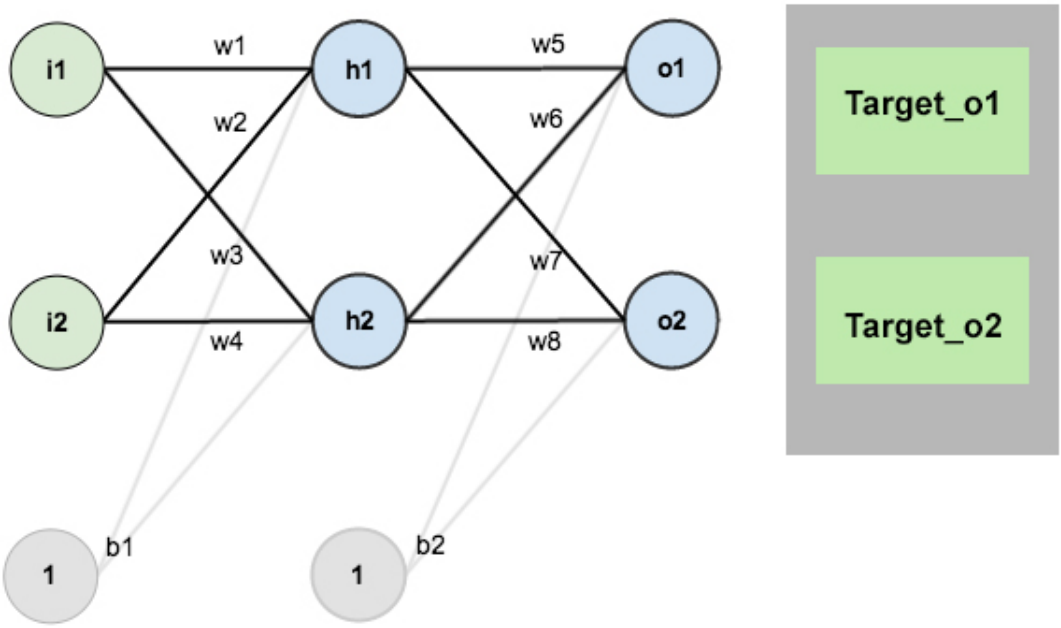
1 Introduction to Deep Learning Neural Networks.

Deep Learning is a branch of Machine Learning that aims to mimic the human brain. That is to do functions by learning/training repeatedly until they're able to do a certain task with a high probability. Some practical examples of deep learning are classifying images, self driving cars, prediction of stock prices and analyzing data to predict arrhythmia etc. So all in all, Deep Learning is a way for computers to do tasks that traditional programming hasn't been able to do. It's a way for us to automate so many jobs that currently require humans and that in turn saves us time to focus on so many other tasks.

An over-simplified way of defining traditional programming would be to define a function that maps an input to a desired output. The function is then tweaked by the programmer to produce a desired result. Supervised Deep Learning on the other hand defines the function with known inputs and outputs. The function is then optimized using gradient descent to produce a function that is probabilistic-ally accurate. Neural Networks traditionally have required a stochastic Propagation method, however we'll be diving deep into how we can model Back-propagation into a continuous ordinary differential equation which will help us model different problems with more accuracy.
Neural networks are made up of nodes and layers connected with functions. An input is passed through these functions which yields an output and then the values of the functions are adjusted through a method called back-propagation so all these functions produce a desired result. The best way to understand forward and back propagation is to work through an example and we'll work through that in the following section.

2 Neural Network Setup(Multi-layer Percepteron)

This is a basic example which consists of 3 layers with two nodes in each layer. Layer I, which can also be denoted as an Input layer. Layer H, which we can denote as the hidden layer and Layer O, which can be denoted as an Output Layer. Nodes B_1 and B_2 are biases for the layer H and O respectively. The layers included in the following diagram will be referenced throughout this paper.



2.1 Layer I (Input Layer)

Layer I has two nodes labelled i1 and i2

$$I = \begin{bmatrix} i_1 \\ i_2 \end{bmatrix} \tag{1}$$

2.2 Layer H (Hidden Layer)

Layer h has two nodes labelled h1 and h2.

$$H = \begin{bmatrix} h_1 \\ h_2 \end{bmatrix} \tag{2}$$

Then the matrix of weights is the following with w_1, w_2, w_3, w_4

$$W_{[1]} = \begin{bmatrix} w_1 & w_2 \\ w_3 & w_4 \end{bmatrix} \tag{3}$$

and

$$B_{[1]} = \begin{bmatrix} b_1 \\ b_1 \end{bmatrix} \tag{4}$$

2.3 Definitions

1. **Hadamard Product**
   The Hadamard Product or the Schur product is an element-wise multiplication of two vectors.Suppose **S** and **T** are two vectors of the same dimension. Then we use S $\odot$ T to denote the element-wise product of the two vectors. As an example,

$$S \odot T = \begin{bmatrix} s_1 \\ s_2 \end{bmatrix} \odot \begin{bmatrix} t_1 \\ t_2 \end{bmatrix} = \begin{bmatrix} s_1 * t_1 \\ s_2 * t_2 \end{bmatrix} \tag{5}$$

2. **Sigmoid Function** The sigmoid functions purpose is to compress the value of its parameter to a number between 0 and 1 where $\sigma(x) \in \mathbb{R}$. If we denote by σ as the sigmoid function, it can be denoted as the following:

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{6}$$

3. **Function Forward Propagation/Activation Function**
   The following function Z is a function that takes in the weight matrix, the input matrix and the bias matrix and produces an output that is used in the sigmoid function to produce the value for the following layer. This works out perfectly as there weight matrix row size matches the input layers vector size. It is calculated as:

$$Z(W^{[n]}, B^{[n]}, I) = W^{[n]}I + b^{[n]} = \begin{bmatrix} w_{n_1} & w_{n_2} \\ w_{n_3} & w_{n_4} \end{bmatrix} \begin{bmatrix} i_{n_1} \\ i_{n_2} \end{bmatrix} + \begin{bmatrix} b_{n_1} \\ b_{n_1} \end{bmatrix} = \begin{bmatrix} w_{n_1} i_{n_1} + w_{n_2} i_{n_2} + b_{n_1} \\ w_{n_3} i_{n_1} + w_{n_4} i_{n_2} + b_{n_1} \end{bmatrix} \tag{7}$$

then the output for this layer, denoted by H is:

$$Z^1 = Z(W^{[1]}, B^{[1]}, I) = W^{[1]}I + b^{[1]} = \begin{bmatrix} w_1 & w_2 \\ w_3 & w_4 \end{bmatrix} \begin{bmatrix} i_1 \\ i_2 \end{bmatrix} + \begin{bmatrix} b_1 \\ b_1 \end{bmatrix} = \begin{bmatrix} w_1 i_1 + w_2 i_2 + b_1 \\ w_3 i_1 + w_4 i_2 + b_1 \end{bmatrix} \tag{8}$$

$$H = \sigma(Z^1)$$

$$H = \begin{bmatrix} h_1 \\ h_2 \end{bmatrix} = \begin{bmatrix} \sigma(w_1 i_1 + w_2 i_2 + b_1) \\ \sigma(w_3 i_1 + w_4 i_2 + b_1) \end{bmatrix} \tag{9}$$

2.4 Layer O (Output Layer)

Layer O has two nodes labelled o1 and o2.

$$O = \begin{bmatrix} o_1 \\ o_2 \end{bmatrix} \tag{10}$$

$$W^{[2]} = \begin{bmatrix} w_5 & w_6 \\ w_7 & w_8 \end{bmatrix} \tag{11}$$

and the matrix of the bias is

$$B^{[2]} = \begin{bmatrix} b_2 \\ b_2 \end{bmatrix} \tag{12}$$

We have that the pre-output for the nodes in this layer can be calculated as: $O = \sigma(Z^2)$

$$Z^2 = Z(W^{[2]}, B^{[2]}, H) = W^{[2]}I + b^{[2]}) = \begin{bmatrix} w_5 & w_6 \\ w_7 & w_8 \end{bmatrix} \begin{bmatrix} h_1 \\ h_2 \end{bmatrix} + \begin{bmatrix} b_2 \\ b_2 \end{bmatrix} = \begin{bmatrix} w_5 h_1 + w_6 h_2 + b_2 \\ w_7 h_1 + w_8 h_2 + b_2 \end{bmatrix} \tag{13}$$

then the output for this layer, denoted by O is:

$$O = \sigma(Z^2) = \begin{bmatrix} o_1 \\ o_2 \end{bmatrix} = \sigma(\begin{bmatrix} w_5 h_1 + w_6 h_2 + b_2 \\ w_7 h_1 + w_8 h_2 + b_2 \end{bmatrix}) = \begin{bmatrix} \sigma(w_5 h_1 + w_6 h_2 + b_2) \\ \sigma(w_7 h_1 + w_8 h_2 + b_2) \end{bmatrix} \tag{14}$$

2.5 Layer Y(Target Layer)

This layer will be used in the following sub-section. This layer contains the desired values that we want our Neural Network to produce.
Layer Y has two nodes labelled "$target_{o1}$" and "$target_{o2}$". The amount of nodes in the **Target layer** have to equal the number of nodes in the output layer O so that the cost function can work.

$$Y = \begin{bmatrix} target_{o1} \\ target_{o2} \end{bmatrix} \tag{15}$$

2.6 Cost Function

The Cost function in machine learning is a function that measures the difference between the hypothesis and the real values. The hypothesis being our output and real values being are desired output. using this information we're able to calculate the error value in our output and then we adjust our NN parameters accordingly. In the following section we will show how weights are updated in our example of the NN.

The Cost function is denoted by C_{total}

$$C_{total} = \sum \frac{1}{2}(Target - Output)^2 \tag{16}$$

$$C_{total} = \sum \frac{1}{2}(Y - O)^2 \tag{17}$$

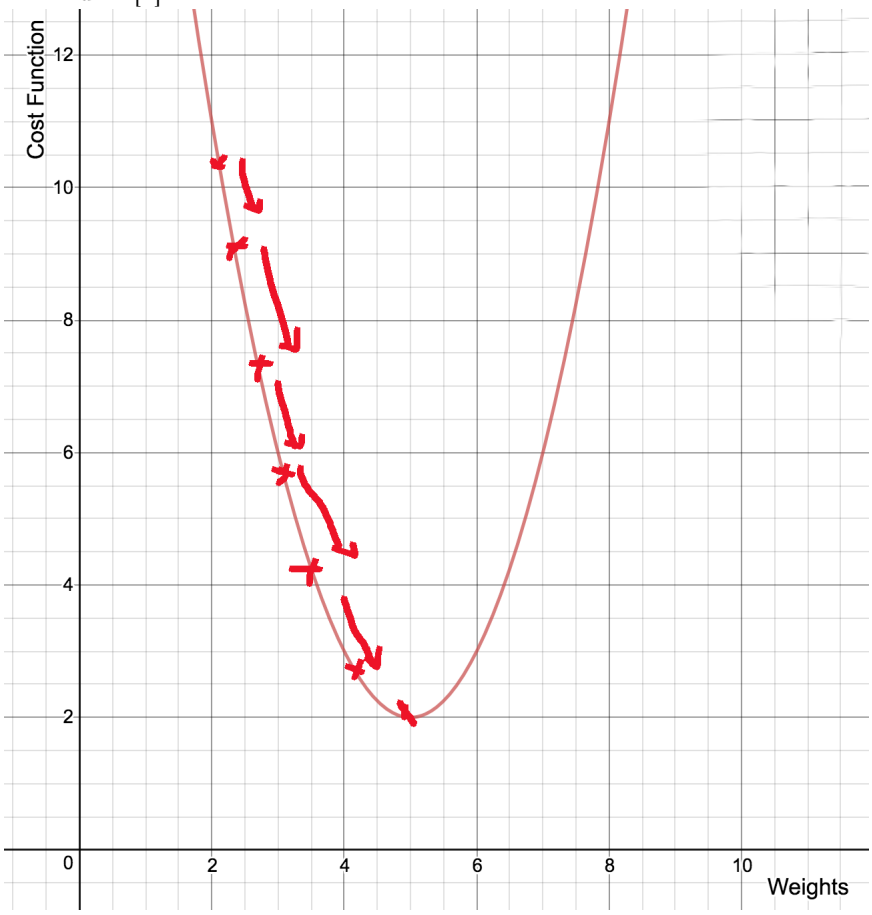
$$C_{o1} = \frac{1}{2}(target_{o1} - o_1)^2 \tag{18}$$

$$C_{o2} = \frac{1}{2}(target_{o2} - o_2)^2 \tag{19}$$

$$C_{total} = C_{o1} + C_{o2} \tag{20}$$

2.7 Gradient Descent

Gradient Descent is used while training a machine learning model. It is an optimization algorithm, based on a convex function, that tweaks the parameters through several iterations to minimize a given function to its local minimum. We will use the following function to minimize our cost function to a local minimum. [4]



The above image is a hypothetical example in simple terms. The cost-function is derived with respect to weight at a random point on the curve. if the gradient at that point is zero we're done. we move in one direction using a step size that we represent using η. If the gradient at that point is positive, we head in the other direction. if its negative, we keep taking steps in that direction. if the gradient is zero, we're done. There are limitations as this method finds a local minimum and not the minimum point of the entire function. There is also the possibility of starting at a local maximum instead of local minimum which would skew our results at times.

2.8 Backward Propagation

The Backward Propagation is probably one of the most difficult concepts to grasp in a Neural Network. We update the weights to match the cost function's error so that the next time we run a forward propagation, the neural network outputs a value closer to our desired target value.

Updating weights using the cost function. Since neural networks can be arranged in the form of matrices and vectors, all of this can be done by using functions on matrices to have all calculations asynchronously. For simplicity of understanding we will take a weight in between layer H and layer O. We will calculate how to update w_5 .

The following equation shows us how the derivative of the cost function with respect to the weight matrix. All the weights are updated simultaneously in between two layers which comes from the concept of "Neurons that wire together, fire together".

$$\frac{\partial C_{total}}{\partial W^{[2]}} = \frac{\partial C_{total}}{\partial o_1} \odot \frac{\partial o_1}{\partial Z^2} \odot \frac{\partial Z^2}{\partial W^{[2]}} \tag{21}$$

However, for simplicity we'll continue to do so for just one weight, w_5. To calculate w_5 we need to take the $\frac{\partial C_{total}}{\partial w_5}$. Using Chain Rule, we can write the expression as the Following.

$$\frac{\partial C_{total}}{\partial w_5} = \frac{\partial C_{total}}{\partial o_1} * \frac{\partial o_1}{\partial Z^2} * \frac{\partial Z^2}{\partial w_5} \tag{22}$$

$C_{total} = \frac{1}{2}(target_{o1} - o_1)^2 + \frac{1}{2}(target_{o2} - o_2)^2$

$$\frac{\partial C_{total}}{\partial o_1} = 2 * \frac{1}{2}(target_{o1} - o_1)^{2-1} * -1 + 0 = -target_{o1} + o_1 \tag{23}$$

$o_1 = \frac{1}{1+e^{-Z^2}}$

$$\frac{\partial o_1}{\partial Z^2} = o_1(1 - o_1) \tag{24}$$

$Z^2 = w_5 h_1 + w_6 h_2 + b_2$

$$\frac{\partial Z^2}{\partial w_5} = 1 * h_1 * w_5^{(1-1)} + 0 + 0 = h_1 \tag{25}$$

Using Equation

$$\frac{\partial C_{total}}{\partial w_5} = (-target_{o1} + o_1) * o_1(1 - o_1) * h_1 \tag{26}$$

To decrease the error, we then subtract this value from the current weight (optionally multiplied by some learning rate, eta, which weâĂŹll set to η): Updated weight $w_5 => w_5^+$

$$w_5^+ = w_5 - \eta * \frac{\partial C_{total}}{\partial w_5} \tag{27}$$

Using the same process we'll update all the other weights in this layer which will translate to w_6^+, w_7^+, w_8^+ . So the updated matrix using the the Hadamard Product(equation 5) would translate into the following.

$$W^{[2]+} = \begin{bmatrix} w_5^+ & w_6^+ \\ w_7^+ & w_8^+ \end{bmatrix} \tag{28}$$

2.9 Backward Propagation II (Layer I and Layer H)

Now we'll be updating the weights in-between Layer I and Layer H. This is significant because as we add more layers, we will be following the same process to update the weights in each preceding layer from the output layer to the input layer.

$$\frac{\partial C_{total}}{\partial w_1} = \frac{\partial C_{total}}{\partial h_1} * \frac{\partial h_1}{\partial Z^1} * \frac{\partial Z^1}{\partial w_1} \tag{29}$$

We know that h_1 affects both o_1 and o_2 therefore the $\frac{\partial C_{total}}{\partial h_1}$ needs to take into consideration its effect on the both output neurons:

$$\frac{\partial C_{total}}{\partial h_1} = \frac{\partial C_{o1}}{\partial h_1} + \frac{\partial C_{o2}}{\partial h_1} \tag{30}$$

$$\frac{\partial C_{o1}}{\partial h_1} = \frac{\partial C_{o1}}{\partial Z^2} * \frac{\partial Z^2}{\partial h_1} \tag{31}$$

$$\frac{\partial C_{o1}}{\partial Z^2} = \frac{\partial C_{o1}}{\partial o_1} * \frac{\partial o_1}{\partial Z^2} \tag{32}$$

$$\frac{\partial C_{o1}}{\partial o_1} = 2 * \frac{1}{2}(target_{o1} - o_1)^{2-1} * -1 = -target_{o1} + o_1$$

$$\frac{\partial o_1}{\partial Z^2} = o_1(1 - o_1)$$

$$Z^2 = w_5 * h_1 + w_6 * h_2 + b_2$$

$$\frac{\partial Z^2}{\partial h_1} = w_5$$

$$\frac{\partial C_{o1}}{\partial h_1} = \frac{\partial C_{o1}}{\partial Z^2} * \frac{\partial Z^2}{\partial h_1} = (o_1(1 - o_1) * (-target_{o1} + o_1)) * w_5$$

Using the same process We calculate $\frac{\partial C_{o2}}{\partial h_1}$

$$\frac{\partial C_{o2}}{\partial h_1} = \frac{\partial C_{o1}}{\partial Z^2} * \frac{\partial Z^2}{\partial h_1} = (o_2(1 - o_2) * (-target_{o2} + o_2)) * w_5 \tag{33}$$

$$\frac{\partial C_{total}}{\partial h_1} = \frac{\partial C_{o1}}{\partial h_1} + \frac{\partial C_{o2}}{\partial h_1} = [(o_2(1-o_2)*(-target_{o2}+o_2))*w_5] + [(o_1(1-o_1)*(-target_{o1}+o_1))*w_5] \tag{34}$$

Now lets find $\frac{\partial h_1}{\partial Z^1} and \frac{\partial Z^1}{\partial w_1}$ to Complete Equation (29)

$h_1 = \frac{1}{1+e^{-Z_1}}$

$$\frac{\partial h_1}{\partial Z^1} = h_1(1 - h_1) \tag{35}$$

$Z_1 = w_1 * i_1 + w_3 * i_2 + b_1$

$$\frac{\partial Z_1}{\partial w_1} = i_1 \tag{36}$$

For simplicity and having to deal with less variables we'll assume

$$K = \frac{\partial C_{total}}{\partial h_1} \tag{37}$$

$$\frac{\partial C_{total}}{\partial w_1} = \frac{\partial C_{total}}{\partial h_1} * \frac{\partial h_1}{\partial Z^1} * \frac{\partial Z^1}{\partial w_1} = K * i_1 * h_1(1 - h_1) \tag{38}$$

Updating the weight as we did before. Updated weight $w_1 => w_1^+$

$$w_1^+ = w_1 - \eta * \frac{\partial C_{total}}{\partial w_1}$$

Using the same process we'll update all the other weights in this layer which will translate to w_2^+, w_3^+, w_4^+ . So the updated weight matrix will be the following.

$$W^{[1]+} = \begin{bmatrix} w_1^+ & w_2^+ \\ w_3^+ & w_4^+ \end{bmatrix}$$

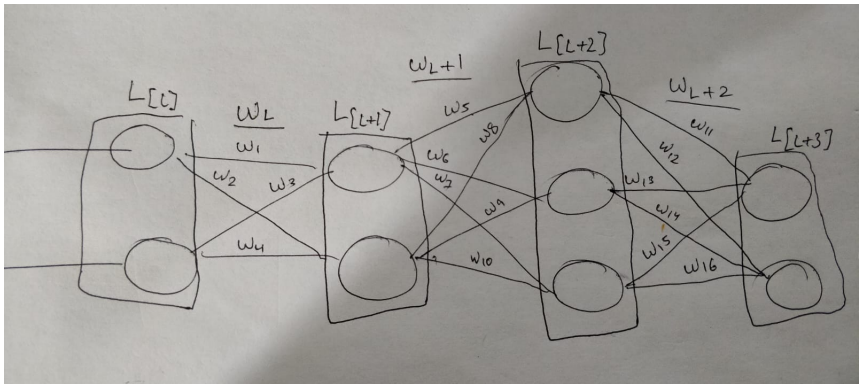
Finally, we've updated all our weights. We run the Neural Network once again to get another solution and we'll continue to do so recursively until our cost function error decreases with each iteration. The + subscript indicates an update in the value of the variable.

$$O^+ = \sigma(Z^2(W^{[2]+}, B^{[2]}, H^+)) = \sigma(W^{[2]}H^+ + B^{[2]}) = \sigma(\begin{bmatrix} w_5^+ & w_6^+ \\ w_7^+ & w_8^+ \end{bmatrix} \begin{bmatrix} h_1^+ \\ h_2^+ \end{bmatrix} + \begin{bmatrix} b_2 \\ b_2 \end{bmatrix})$$

$$O^+ = \begin{bmatrix} \sigma(w_5^+ h_1^+ + w_6^+ h_2^+ + b_2) \\ \sigma(w_7^+ h_1^+ + w_8^+ h_2^+ + b_2) \end{bmatrix} = \begin{bmatrix} o_1^+ \\ o_2^+ \end{bmatrix}$$

We updated the weight parameters in our example but a similar process can be repeated to update the bias.

2.10 Back Propagation Generalized Equations

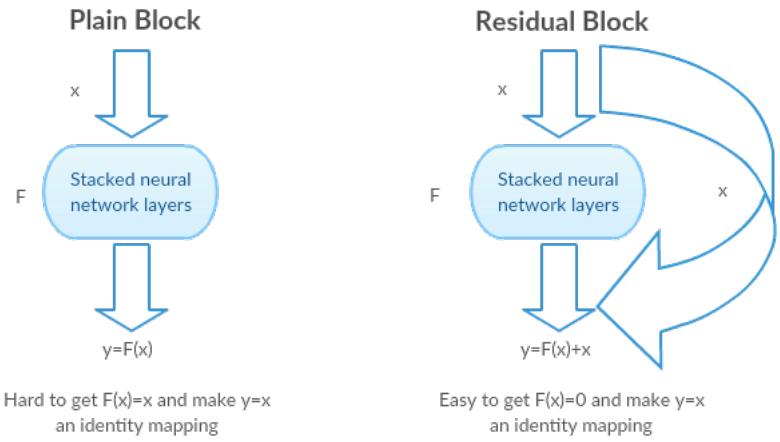


$$\frac{\partial C_{total}}{\partial W_l} = \frac{\partial C_{total}}{\partial L_{l+1}} \odot \frac{\partial L_{l+1}}{\partial Z_{l+1}} \odot \frac{\partial Z_{l+1}}{\partial W_l} \tag{39}$$

The above equation represents a way to use this equation for any multi-layer perceptron (MLP). The parameters will have to be calculated the way we had done above but for each layer using the chain rule to calculate the following. In equation 39 , We generalize the chain rule to work as the following. The weight matrix W_l connects layer L_l and L_{l+1} and Z_{l+1} is the activation function layer L_{l+1} before it has the σ function has been used on it. This equation can therefore be used to update any weight matrix in any MLP.

3 Residual Neural Network(RNN) Model

Residual Neural Networks were introduced earlier in this decade and showed greater optimization speed than many other neural networks. They were specially efficient for image recognition. I will explain how they work using the following image.



We generalize the multi-layer percepteron as the following function:

$$h_{[t+1]} = \sigma\left(W_{[t+1]}h_{[t]} + b_{[t+1]}\right). \tag{40}$$

Notice that there is an impossibility to transform equation 40 into a differential equation.However if we use Residual Networks, we can transform our equation to generate an equation of the form:

$$h_{[t+1]} = h_{[t]} + \sigma\left(W_{[t+1]}h_{[t]} + b_{[t+1]}\right). \tag{41}$$

4 Ordinary Differential Equation(ODE) Neural Network

In the Equation Above (Equation 41), $h_{[t+1]}$ can be translated as the next layer. So if we consider I to be $h_{[t]}$ then H will be $h_{[t+1]}$ and O will be $h_{[t+2]}$. In that manner we can notice the pattern in the subscripts. In the same manner $W^{[t+1]}$ and $b^{[t+1]}$ are the respected weight matrix and the Bias matrix that correspond to $h^{[t+1]}$. A residual network can be seen as **Euler discretization** of a continuous equation because:

$$h_{[t+1]} = h_{[t]} + \sigma\left(W^{[t+1]}h_{[t]} + b^{[t+1]}\right) = h_{[t]} + (t+1-t)\sigma\left(W^{[t+1]}h_{[t]} + b^{[t+1]}\right).$$

$$h_{[t+1]} - h_{[t]} = \Delta t\sigma\left(W^{[t+1]}h_{[t]} + b^{[t+1]}\right)$$

thus we can generalize this to the following equation.

$$\frac{\Delta h_{[t]}}{\Delta t} = \sigma\left(W^{[t+1]}h_{[t]} + b^{[t+1]}\right),$$

so, we can conclude that we have the equation:

$$\frac{dh_{[t]}}{dt} = \sigma\left(W^{[t]}h_{[t]} + b^{[t]}\right). \tag{42}$$

Following the above steps we are able to conclude that the setup of neural network can be seen as a differential equation.

4.1 Setup of ODE Neural Net

Lets take the above equation and substitute it with an equal function. $f\left(W^{[t]}, h_{[t]}, b^{[t]}\right) = \sigma\left(W^{[t]} h_{[t]} + b^{[t]}\right)$

To simplify our future calculation we will remove the bias $b^{[t]}$ from the function.

Note: RNN's have discrete solutions in comparison to ODE neural networks which provide a continious solution.

$$\frac{dh_{[t]}}{dt} = f\left(W^{[t]}, h_{[t]}\right) \tag{43}$$

The authors of the Neural ordinary differential equations paper [1] present an alternative approach to calculating the gradients of the ODE by using the adjoint sensitivity method by Pontryagin. This method works by solving a second, augmented ODE backwards in time, which can be used with all ODE's integrator and has a low memory footprint.

Lets unpack the paragraph above. If you want to find the output at hidden node $h_{[t_1]}$ you would have to solve the following function for times between t_1 and t_0 and that can be seen below. The $ODESolve$ below is an a way to script the differential equation into a function showing the input variables required for this function to work. The following is the equation for forward propagation of an ODE neural network:

$$h_{[t_1]} = h_{[t_0]} + \int_{t_0}^{t_1} f\left(W^{[t]}, h_{[t]}\right) dt = ODESolve(h_{[t_0]}, t_1, t_0, f, W^{[t]}) \tag{44}$$

The Loss function is defined as an arbitrary function taking in our hidden layer output at time t_1 to minimize the error e.g gradient descent. It is defined as the following:

$$L\left(h_{[t_1]}\right) = L\left(h_{[t_0]} + \int_{t_0}^{t_1} f\left(W^{[t]}, h_{[t]}\right) dt\right) = L(ODESolve(h_{[t_0]}, t_1, t_0, f, W^{[t]})) \tag{45}$$

The command $ODESolve(h_{[t_0]}, t_1, t_0, f, W^{[t]})$ solves the differential equation. As we previously calculated the partial derivative of the cost function with respect to the the parameters of the function, we'll calculate the partial derivative of the loss function with each parameter using the Adjoint method.

4.2 The Adjoint Method

The **Adjoint sensitivity method** now determines the gradient of the loss function with respect to the hidden state. The **Adjoint state** is the gradient

with respect to the particular state at a specified time t. In standard neural networks, the gradient of the layer h_t depends on the gradient from the next layer h_{t+1} by chain rule

$$A = \frac{dL}{dh_t} = \frac{dL}{dh_{t+1}} \frac{dh_{t+1}}{dh_t}. \tag{46}$$

To calculate this Adjoint A for the ODE neural network,we need to derive this equation with respect to time which will give us a Chain rule as follows:

$$\frac{dA(t)}{dt} = -\mathbf{A}^\top \frac{\partial f\left(W^{[t]}, h^{[t]}\right)}{\partial h} \tag{47}$$

Equation 47 has a transpose within it's equation to accommodate Vector/Matrix Multiplication.

With h continuous hidden state, we can write the transformation after an ϵ change in time as

$$h(t+\epsilon) = \int_t^{t+\epsilon} f(h(t), t, W)dt + h(t) = T_\epsilon(h(t), t) \tag{48}$$

where and chain rule can also be applied

$$\frac{dL}{\partial h(t)} = \frac{dL}{dh(t+\epsilon)} \frac{dh(t+\epsilon)}{dh(t)} \qquad \text{or} \qquad A = A(t+\epsilon)\frac{\partial T_\epsilon(h(t), t)}{\partial h(t)} \tag{49}$$

The following is the proof of equation 47

Proof.

$$\begin{aligned}
\frac{dA}{dt} &= \lim_{\epsilon \to 0^+} \frac{A(t+\epsilon) - A}{\epsilon} && (50)\\
&= \lim_{\epsilon \to 0^+} \frac{A(t+\epsilon) - A(t+\epsilon)\frac{\partial}{\partial h(t)} T_\epsilon(h(t))}{\epsilon} && \text{(by Eq 49)} \quad (51)\\
&= \lim_{\epsilon \to 0^+} \frac{A(t+\epsilon) - A(t+\epsilon)\frac{\partial}{\partial h(t)} \left(h(t) + \epsilon f(h(t), t, W_{[t]}) + \mathcal{O}(\epsilon^2)\right)}{\epsilon} && \text{(Taylor series around } h(t)) \quad (52)\\
&= \lim_{\epsilon \to 0^+} \frac{A(t+\epsilon) - A(t+\epsilon)\left(I + \epsilon \frac{\partial f(h(t), t, W_{[t]})}{\partial h(t)} + \mathcal{O}(\epsilon^2)\right)}{\epsilon} && (53)\\
&= \lim_{\epsilon \to 0^+} \frac{-\epsilon A(t+\epsilon)\frac{\partial f(h(t), t, W_{[t]})}{\partial h(t)} + \mathcal{O}(\epsilon^2)}{\epsilon} && (54)\\
&= \lim_{\epsilon \to 0^+} -A(t+\epsilon)\frac{\partial f(h(t), t, W_{[t]})}{\partial h(t)} + \mathcal{O}(\epsilon) && (55)\\
&= -A\frac{\partial f(h(t), t, W_{[t]})}{\partial h(t)} && (56)
\end{aligned}$$

■

We pointed out the similarity between adjoint method and backpropagation (eq. 49). Similarly to backpropagation, ODE for the adjoint state needs to be solved *backwards* in time. We specify the constraint on the last time point, which is simply the gradient of the loss wrt the last time point, and can obtain the gradients with respect to the hidden state at any time, including the initial value.

$$\underbrace{A(t_N) = \frac{dL}{dh(t_N)}}_{\text{initial condition of adjoint diffeq.}}$$

$$\underbrace{A(t_0) = A(t_N) + \int_{t_N}^{t_0} \frac{dA}{dt}\, dt = A(t_N) - \int_{t_N}^{t_0} A^T \frac{\partial f(h(t), t, W_{[t]})}{\partial h(t)}}_{\text{gradient wrt. initial value}}$$

Here we assumed that loss function L depends only on the last time point t_N. If function L depends also on intermediate time points $t_1, t_2, \ldots, t_{N-1}$, etc., we can repeat the ad joint step for each of the intervals $[t_{N-1}, t_N]$, $[t_{N-2}, t_{N-1}]$ in the backward order and sum up the obtained gradients.

We can generalize equation(47) to obtain gradients with respect to $W_{[t]}$ and $h_{[t]}$ constants with respect to t and and the initial and end times, t_0 and t_N. We view $W_{[t]}$ and t as states with constant differential equations and write

$$\frac{\partial W_{[t]}(t)}{\partial t} = \mathbf{0} \qquad \frac{dt(t)}{dt} = 1 \tag{57}$$

We can then combine these with z to form an augmented stateNote that we've overloaded t to be both h part of the state and the (dummy) independent variable. The distinction is clear given context, so we keep t as the independent variable for consistency with the rest of the text. with corresponding differential equation and ad joint state,

$$f_{aug}([h_{[t]}, W_{[t]}, t]) = \frac{d}{dt}\begin{bmatrix} h_{[t]} \\ W_{[t]} \\ t \end{bmatrix}(t) := \begin{bmatrix} f([h_{[t]}, W_{[t]}, t]) \\ \mathbf{0} \\ 1 \end{bmatrix},$$

$$A_{aug} := \begin{bmatrix} A \\ A_{W[t]} \\ A_t \end{bmatrix},\ A_{W[t]}(t) := \frac{dL}{dW_{[t]}(t)},\ A_t(t) := \frac{dL}{dt(t)}$$

JACOBIAN TRANSFORMATION

Definition: The **Jacobian** of the function u_1, u_2 and u_3 with respect to x_1, x_2, x_3 is:

$$\frac{\partial(u_1, u_2, u_3)}{\partial(x_1, x_2, x_3)} = \begin{bmatrix} \frac{\partial u_1}{\partial x_1} & \frac{\partial u_1}{\partial x_2} & \frac{\partial u_1}{\partial x_3} \\ \frac{\partial u_2}{\partial x_1} & \frac{\partial u_2}{\partial x_2} & \frac{\partial u_2}{\partial x_3} \\ \frac{\partial u_3}{\partial x_1} & \frac{\partial u_3}{\partial x_2} & \frac{\partial u_3}{\partial x_3} \end{bmatrix}$$

In a similar manner we're going to transform our augmented function to produce a vector to so we can get the partial gradient of the loss function with respect to the Weights so We can use that to Update the Weights as a continuous function. By doing this we will iterate the Differential equation until the Loss function in minimized.

Note this formulates the augmented ODE as an autonomous (time-invariant) ODE, but the derivations in the previous section still hold as this is h special case of h time-variant ODE. The Jacobian of f_{aug} has the form

$$\frac{\partial f_{aug}}{\partial [h_{[t]}, W_{[t]}, t]} = \begin{bmatrix} \frac{\partial f}{\partial h_{[t]}} & \frac{\partial f}{\partial W_{[t]}} & \frac{\partial f}{\partial t} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} \end{bmatrix} (t) \tag{58}$$

$$\frac{dA_{aug}(t)}{dt} = -\begin{bmatrix} A(t) & A_{W[t]}(t) & A_t(t) \end{bmatrix} \frac{\partial f_{aug}}{\partial [h_{[t]}, W_{[t]}, t]}(t) = -\begin{bmatrix} A\frac{\partial f}{\partial h_{[t]}} & A\frac{\partial f}{\partial W_{[t]}} & A\frac{\partial f}{\partial t} \end{bmatrix} (t) \tag{59}$$

The first element is the adjoint differential equation (47), as expected. The second element can be used to obtain the total gradient with respect to the parameters, by integrating over the full interval and setting
$A_{W[t]}(t_N) = \mathbf{0}$.

$$\frac{dL}{dW_{[t]}} = A_{W[t]}(t_0) = -\int_{t_N}^{t_0} A(t) \frac{\partial f(h_{[t]}(t), t, W_{[t]})}{\partial W_{[t]}} \, dt \tag{60}$$

Finally, we also get gradients with respect to t_0 and t_N, the start and end of the integration interval.

$$\frac{dL}{dt_N} = A(t_N) f(h_{[t]}(t_N), t_N, W_{[t]}) \qquad \frac{dL}{dt_0} = A_t(t_0) = A_t(t_N) - \int_{t_N}^{t_0} A(t) \frac{\partial f(h_{[t]}(t), t, W_{[t]})}{\partial t} \, dt \tag{61}$$

The Adjoint method is for all the parameters is done using the following command that we mentioned above: $ODESolve(h_{[t_0]}, t_1, t_0, f, W^{[t]})$

The Complete Algorithim was summerised in the original paper as the following [1]

Algorithm 1 (h) *Reverse-mode derivative of an ODE initial value problem*

Input*: dynamics parameters* W*, start time* t_0*, stop time* t_1*, final state* $h(t_1)$*, loss gradient* $\frac{\partial L}{\partial h(t_1)}$ $s_0 = [h(t_1), \frac{\partial L}{\partial h(t_1)}, 0_{|W|}]$ *Define initial augmented state* $\text{aug\_dynamics}[h(t), A(t), \cdot], t, W$*: Define dynamics on augmented state* ***return*** $[f(h(t), t, W), -A(t)^\top \frac{\partial f}{\partial h}, -A(t)^\top \frac{\partial f}{\partial W}]$ *Compute vector-Jacobian products* $[h(t_0), \frac{\partial L}{\partial h(t_0)}, \frac{\partial L}{\partial W}] = ODESolve(s_0, \text{aug\_dynamics}, t_1, t_0, W)$ *Solve reverse-time ODE* ***return*** $\frac{\partial L}{\partial h(t_0)}, \frac{\partial L}{\partial W}$ *Return gradients*

5 Diffeqflux.jl Implementation

DiffEqFlux.jl fuses the world of differential equations with machine learning by helping users put differential equation solvers into neural networks. This package utilizes DifferentialEquations.jl and Flux.jl as its building blocks. We used this library to create a ODE neural network. Mapping the first function of the lotka volterra to the second function of lotka volterra. Here the second equation is used as our training data.

The Following Neural Network Code uses the lotka volterra. The lotka volterra is a system of differential equations that measures the population of a species with respect to predators,deaths and births of a specie. We use this function inside a differential equation with three layers, with 2 and 3 nodes in the respective layers. The activation function used for each layer is the sigmoid. The output is of the NN is used inside the cost function, defined within the code. After that $Flux.train!(loss4, [p], data1, opt, cb = cb)$ is used to run the NN with *Descent* as the optimizer. The Network is run 10 times and the error goes down from approximately from 22 percent to 0.06 percent error. The code for Julia is given below. It can be run in any Julia notebook.

```
# THIS MODEL IS FOR A NEURAL NETWORK WITH A ODE LAYER AND
INPUT TO THE LAYER BEING THE PARAMETERS OF IT
#THIS RUNS CORRECTLY!
using Flux, DiffEqFlux, DifferentialEquations, Plots
############################################################
## Setup ODE to optimize
function lotka_volterra(du,u,p,t)
  x, y = u
  Îś, Îš, Ît', Îş = p
  du[1] = dx = Îś*x - Îš*x*y
  du[2] = dy = -Ît'*y + Îş*x*y
end
u0 =Float32[1.0,1.0]
tspan = (0.0,1.0)
p = [1.5,1.0,3.0,1.0]
prob = ODEProblem(lotka_volterra,u0,tspan,p)
```

```
#######################################################
#First we create a solution of the Diff Eq that accepst parameters
#using the forward solution method diff_rd
p = Flux.param([1.5,1.0,3.0,1.0])#We set the parameters to track
function predict_rd2() #THis call the differential equation solver
  diffeq_rd(p,prob,Tsit5(),saveat=0.1)
end
println("we print the values of predict_rd2()=")
println(predict_rd2())#We check the format of the solution
#####################################################################
mymodel4 = Chain(
#we create the perceptron with the ODE layer based on parameters
  Dense(2,3,ÏČ),
   p->predict_rd2()
)
#####################################################################
println("We test the run of the perceptron, mymodel4([0.5,0.5])")
println(mymodel4([0.5,0.5]))#We test that the perceptron is well defined
#The perceptron inputs and array of two values and outputs the
entire solution of the
#differential equation for the generated parameters of the system.
#The goal here is to optimize the parameters of the solution
#of the ODE so that the two solutions
#will converge to the second function.
#############################################################
#We now calculate the error between the values generated
#by the perceptron and the constant functions 1.
#The loss function must take 2 parameters
# so we make our function depend on the second parameter
#although there is no use for the second one
#since we want the solutions of the ODE to converge to function 2.
#The loss function will calculate the error between the functions
#and each one of the
#two solutions of the ODE at the input values of 0.0, 0.1, 0.2,...1.0.
function loss4(x,y)
    T=0;
    for i in 1:11
        T=T+(mymodel4(x)[i][1]-mymodel4(x)[i][2])^2;
    end
    return T
end
println("Example value of loss function.
#Value of loss4([0.5,0.5],[1.0,1.0])=",loss4([0.5,0.5],[1.0,1.0]))
# We ilustrate the run of the loss4 function
#############################################################
```

```
#We proceed to the training of the peceptron and plotting of
#solutions
# We begin by creating the training data.
#The format is weird but this is what worked
newx=[[0.1,0.1],[0.2,0.2],[0.3,0.3],[0.4,0.4],[0.5,0.5],
[0.6,0.6],[0.7,0.7],[0.8,0.8],[0.9,0.9],[1.0,1.0]]
newy=[[1,1],[1,1],[1,1],[1,1],[1,1],
[1,1],[1,1],[1,1],[1,1],[1,1]]
#This part of the data is never used
data1 =[(newx[1], newy[1]),(newx[2],newy[2])
,(newx[3],newy[3]),(newx[4],newy[4]),
(newx[5],newy[5]), (newx[6],newy[6]),
(newx[7],newy[7]),(newx[8],newy[8]),
(newx[9],newy[9]),(newx[10],newy[10])]
println()
function totalloss4()#This is the total error function.
#It is not used in the training, but just to calculate the total error
    T=0;
    for i in 0:40
        T=T+loss4([i*0.1,i*0.1],[1.0,1.0]);
    end
    return T/40
end
opt = ADAM(0.1)#This is the optimization parameter
cb = function () #callback function to observe training
    println("Value of totalloss4() in this iteration=")
    display(totalloss4())
  # using ‘remake‘ to re-create our ‘prob‘ with current parameters ‘p‘
   display(scatter(
   solve(remake(prob,p=Flux.data(p)),Tsit5(),saveat=0.1),ylim=(0,6))
   )
end
# Display the ODE with the initial parameter values.
println("Initial plot of solutions and total error n\n\n\n")
cb()
#Display values of parameters before and after training
println("Value of parameter p=", p)
println("starting training......\n\n\n\n\n\n\n\n\n")
println()
Flux.train!(loss4, [p],data1 , opt, cb=cb)
println()
println("New Value of parameter p=", p)
println("New value of totalloss4()=",totalloss4())
println()
println("Plot of solutions with final parameter\n")
display(
```

```
scatter(solve(remake(prob,p=Flux.data(p)),Tsit5(),saveat=0.1),ylim=(0,6))
)
println("END")
```

References

[1] 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.

[2] Weinan E. "A Proposal on Machine Learning via Dynamical Systems". In: *Communications in Mathematics and Statistics* 5.1 (Mar. 2017), pp. 1–11. ISSN: 2194-671X. DOI: 10.1007/s40304-017-0103-z. URL: https://doi.org/10.1007/s40304-017-0103-z.

[3] Catherine F. Higham and Desmond J. Higham. *Deep Learning: An Introduction for Applied Mathematicians*. 2018. eprint: arXiv:1801.05894.

[4] Matt Mazur. *A Step by Step Backpropagation Example*. URL: https://mattmazur.com/2015/03/17/a-step-by-step-backpropagation-example/. (accessed: 02.02.2019).

[5] Michael Nielsen. *Neural Networks and DeepLearning*. ebook, 2018. URL: http://neuralnetworksanddeeplearning.com/.

[6] Christopher Rackauckas et al. "DiffEqFlux.jl - A Julia Library for Neural Differential Equations". In: *CoRR* abs/1902.02376 (2019).

[7] Lars Ruthotto and Eldad Haber. "Deep Neural Networks motivated by Partial Differential Equations". In: *CoRR* abs/1804.04272 (2018).