Project Based Learning in Data Science

With Applications of Neural Networks

Mid-Term Report

**Group Members**

**2/c Cody Bellamy \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

[Cody.R.Bellamy@uscga.edu](mailto:Cody.R.Bellamy@uscga.edu)

**2/c Justin Steiner \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

[Justin.C.Steiner@uscga.edu](mailto:Justin.C.Steiner@uscga.edu)

Table of Contents

[1 Introduction to Neural Networks 3](#_Toc33980284)

[1.1 Inputs, Outputs, and Neurons 3](#_Toc33980285)

[1.2 Activation Functions 4](#_Toc33980286)

[1.2.1 Linear 4](#_Toc33980287)

[1.2.2 Sigmoid 5](#_Toc33980288)

[1.2.3 Rectified Linear Unit (ReLU) 6](#_Toc33980289)

[1.3 Feed Forward 7](#_Toc33980290)

[1.4 Back Propagation 8](#_Toc33980291)

[1.4.1 Calculating Overall Error 8](#_Toc33980292)

[1.4.2 Steepest Ascent/Gradient Descent 9](#_Toc33980293)

[1.4.3 Backpropagation Algorithm 9](#_Toc33980294)

[1.5 Summary of Introduction 10](#_Toc33980295)

[2 Types of Neural Networks Studied 11](#_Toc33980296)

[2.1 Deep Learning 11](#_Toc33980297)

[2.1.1 Fully Connected Neural Network 11](#_Toc33980298)

[2.1.2 Convolutional Neural Network 12](#_Toc33980299)

[2.1.3 Recurrent Neural Network 12](#_Toc33980300)

[2.2 Genetic Programming 12](#_Toc33980301)

[2.3 Reinforcement Learning 12](#_Toc33980302)

[3 Summary and Lessons Learned 13](#_Toc33980303)

[4 References 14](#_Toc33980304)

# Introduction to Neural Networks

When considering a neural network for applications in data science, it is important to understand the basic structure, process, and common algorithms that are used in application. There are two primary branches of study revolving around neural networks. The first branch is artificial intelligence. Artificial intelligence is the study of decision making. The goal of artificial intelligence is to design a neural network that considers many factors and simulates natural intelligence to solve a problem. The second branch of study is machine learning. Machine learning is the process of analyzing data and making decisions based on previous data. The goal of machine learning is to reach an optimal solution without regard for the process and/or emulating intelligence. A machine learning algorithm only cares for a solution that is optimal and will “learn” to disregard any non-optimal solutions. In this directed study, we have placed an emphasis on machine learning and related sub-branches such as deep learning. This introduction will provide a basis into the basic structure and functionality of a deep-learning network.

## Inputs, Outputs, and Neurons

As the name implies, a neural network is designed with the human brain as inspiration. Each node in the network is called a neuron. Each neuron in the network contains a series of inputs and outputs like the synapsis in a brain.

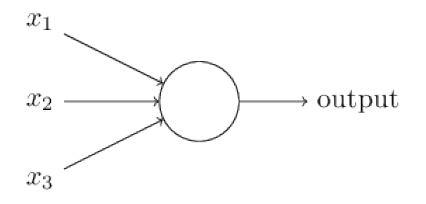


Fig. 1.1 – A neuron with 3 inputs and 1 output

In figure 1.1, we can see a neuron which as 3 inputs and one output. We can represent the inputs and outputs as the vectors and y respectively. It is important to note that each neuron contains a variable number of inputs and only one output. Neurons contain activation function that determines its output based on its inputs. Each neuron in the network calculates its output based on a predetermined function called an “activation function.” Using this basic structure, we can create a neural network with multiple neurons as shown in figure 1.2.

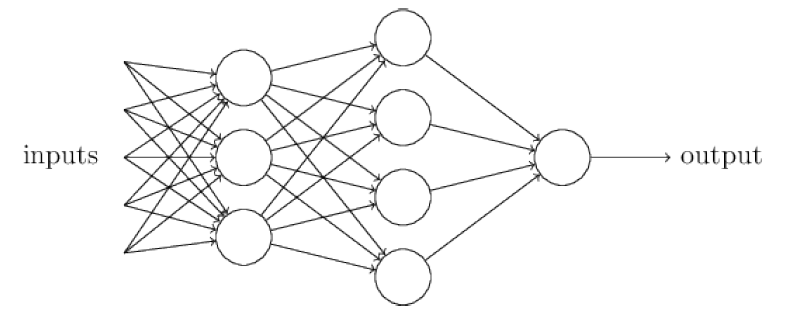


Fig. 1.2 – A deep learning neural network with 5 inputs, 2 hidden layers[[1]](#footnote-1), and 1 output

## Activation Functions

The learning behavior of a neural network is dependent on many factors. But, one of the largest factors in the ability of a neural network to learn is the activation function. Depending on what results are desired, a different activation function will be chosen. Three activation functions that we will be covering are Linear, Sigmoid, and Rectified Linear Unit (ReLU).

### Linear

The linear function is a simple representation of a linear function. Neurons with the linear activation function are given the name, “perceptron.” To understand this function, we will add some features to our understanding of a neural network. First, each input will be given a weight, w, that is specific to only that input. Second, each neuron will contain an overall bias, b. This activation function is powerful in small networks. For example, the weights and biases could be hard-coded in such a way to create logic gates.

Function 1.1 – Linear activation function

Although linear models would appear to be the obvious choice when attempting to model machine learning, this is not always the case. As we can see by function 1.1, our output is a binary response as a function of the inputs. Put in simpler terms, a linear activation function would allow inputs of any value but could only output a 1 or 0 as shown in figure 1.3. This useful in applications where binary decisions are required. But, this model comes with its flaws. Since the response is binary, y could be equal to .000001. Which, to a human would be almost blatantly obvious that the value should be 0. In a linear model, this value is changed to 1. That is a lot of information that is lost in the network that could be passed on to the next neuron as input. Another issue with linear models is the fact that a small change to the weights and biases could result in a drastic change within the network. Take the last example for instance. If the bias is adjusted such that Δb= -0.000001, the neuron would now flip its output completely changing the behavior of the network.

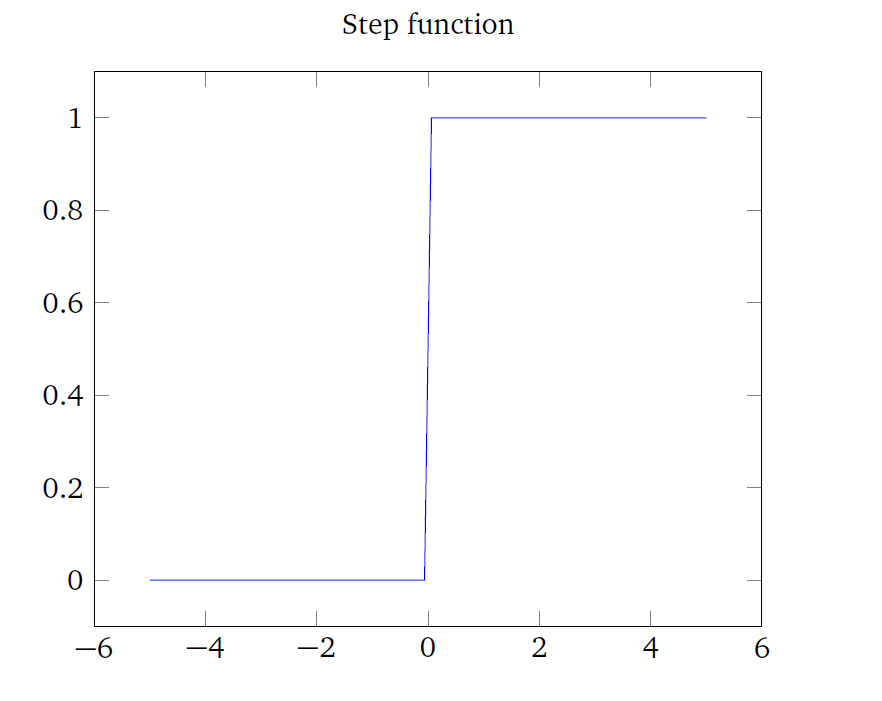


Fig 1.3 – The response of a linear activation function

### Sigmoid

The sigmoid function is a solution to the issues presented at the end of the previous section, 1.2.1. We would like to keep many of the same traits to allow for the capability of training our network. For example. Since the response of a linear network is a step function, we would like positive values of y to result in values closer to 1 and negative values of y to result in values closer to 0.

Function 1.2 – Sigmoid activation function

This equation is structured such that as z >> 0, y≈1 and as z << 0, y≈0. This behavior mimics the behavior of a step function with the added benefit of allowing a smoother transition between values. At values where z≈0, the value of y≈0.5 and the rate of change of y is greatest as shown in figure 1.4. This helps to discourage the values of y to be close to 0.5 and to have a higher probability of being closer to 0 or 1. Another important note is that the output is exactly the output of the sigmoid function. This behavior of neurons allows for more dynamic learning behavior.

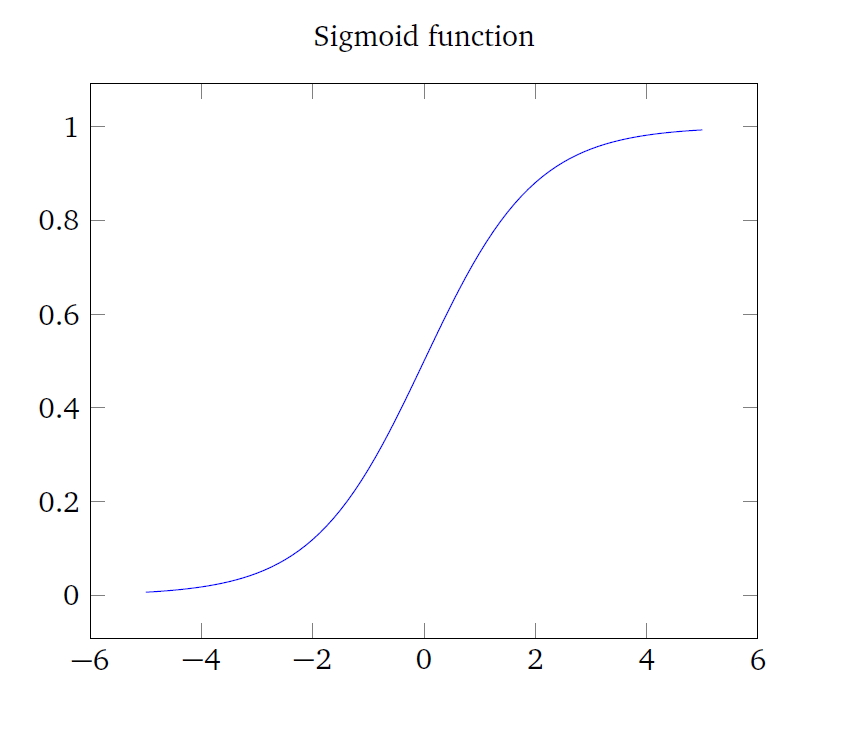


Fig 1.4 – The response of a sigmoid neuron

Like perceptron neurons, there are downsides to the sigmoid. Most notably, is the mathematical efficiency with respect to time. In the feed forward[[2]](#footnote-2) process, the outputs are calculated for each neuron independently. Calculating exponentials becomes a very expensive process for large networks. For smaller networks, this is not a big deal. But, the sigmoid function has an exponential in big-O.

### Rectified Linear Unit (ReLU)

The final activation function that we will explore will be the ReLU function. Although there are several derivations and deviations of the ReLU function, we will only explore one for the sake of simplicity.

Function 1.3 – ReLU activation function

As shown in function 1.3, the ReLU function almost brings us full circle back to the linear activation function. The major difference is that the output of the ReLU function has the range [0,y). The values are no longer 0, 1, or a range between the two. We now have the capability of reaching values of y that approach infinity. Although this might seem to break our concept of a neural network that we have been building, this has little to no negative repercussions. Recall that the values for our inputs can be any value and are simply a function of our weights and biases. At first, our neural network will start with random weights and biases and over time, these will adjust to our inputs to achieve desired results. With this in mind, we quickly realize that the outputs do not matter if they are inputs to another layer.

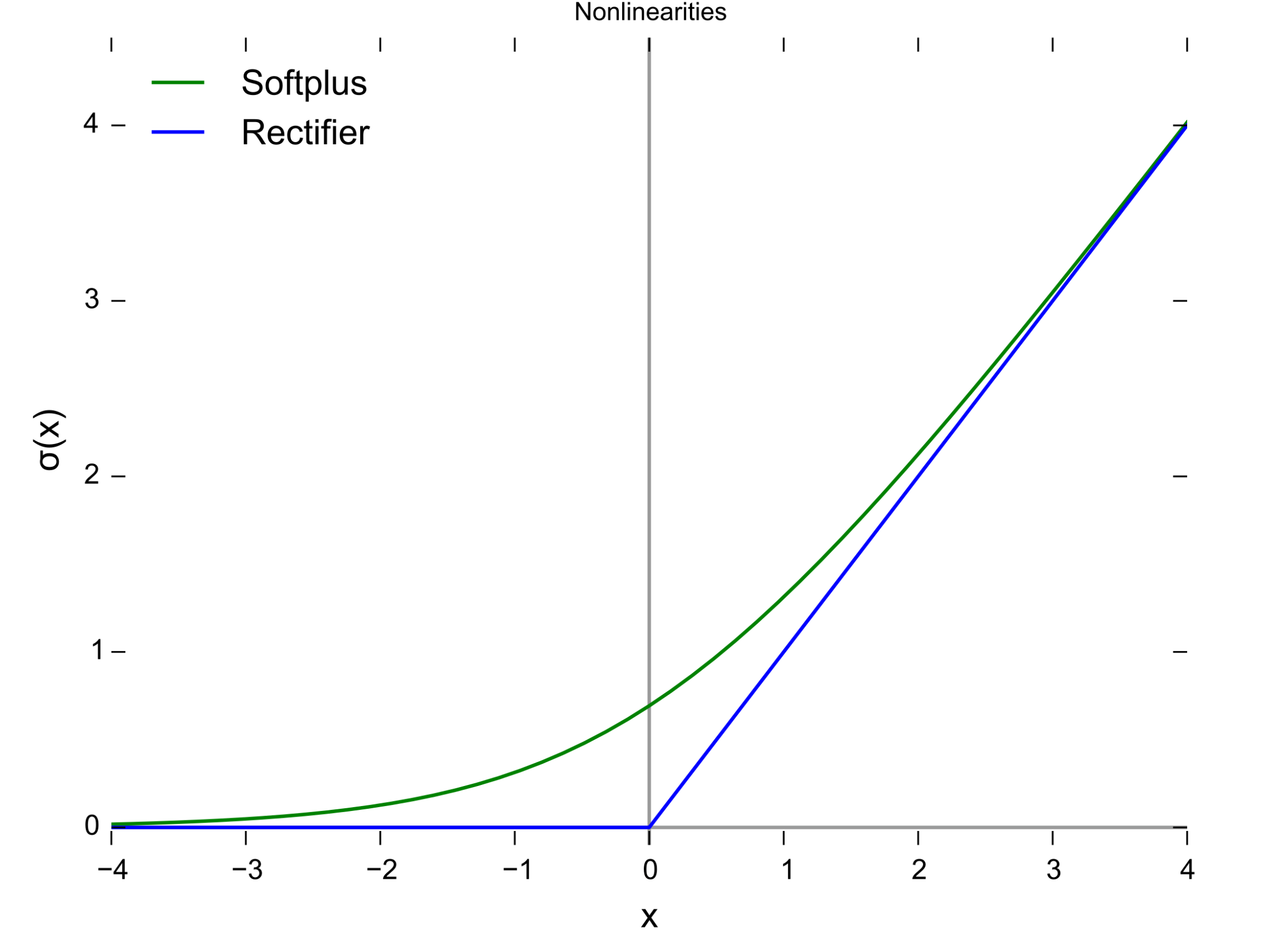


Fig 1.5 – ReLU activation function (blue) and SmoothReLU/Softplus activation function (green)

## Feed Forward

Almost all neural networks follow this model. Feed forward describes the process of accepting inputs, performing calculations to determine output, and feeding the output “forward” to the next layer. In a feed forward model, this allows us the freedom to design our neural network with any structure as we please so long as the overall network’s inputs and outputs satisfy our needs. Additionally, using a feed forward model makes the math quite a bit simpler. When designing our neural network, we need a means of quantifying the performance and a method that allows us to adjust the network towards our goal. This process becomes very complicated when a neuron is capable of feeding outputs to itself or even previous layers in the network. As you will see in the next section about back propagation, our calculations are dependent on the feed forward model due to the assumption that the error in our network started from the input and propagated throughout the network.

## Back Propagation

At this point, we’ve discussed the setup and layout of a simple neural network; but, what of the learning process? To achieve learning from a neural network, we need a way to adjust our weights and biases in a way that achieves beneficial results. Assuming that our network follows a feed-forward model, we can assume that the error that we have calculated had propagated throughout our network, beginning at the inputs and compounding throughout the network at each layer. We will correct the error by starting at the end of the network with a known error and move backwards, one layer at a time, recalculating the error at that layer and making appropriate adjustments. This process is called back propagation.

### Calculating Overall Error

The first step in back propagation is to determine what our error is. We will consider our final output, desired output, all weights, and all biases in our network. To consider all of these and quantify the results of our network, we need to define a cost function. For simplicity, we will discuss the mean squared error (MSE) function as our cost function. Since MSE is a strictly positive exponential function, we can quantify how well our network is doing by minimizing the result of MSE. Since our goal is to adjust our weights and biases, our MSE will be a function of weights and biases.

Function 1.4 – Mean squared error cost function

As shown in function 1.4, n is the number of training inputs (commonly referred to the batch size), y is the vector of outputs from our neural network, x is a vector of inputs to our neural network, a is the actual result from our training data, w is our weights, and b is our biases. Using this function, we are challenged with minimizing C(w,b). Our cost function can be visualized in figure 1.6.

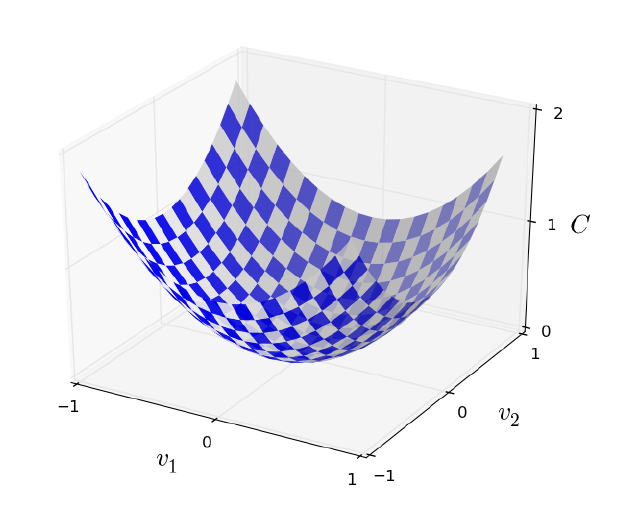


Fig 1.6 – An oversimplified visualization of our cost function plotted in 3 dimensions

### Steepest Ascent/Gradient Descent

From figure 1.6, our goal becomes a matter of minimizing our cost. This can be accomplished with some calculus. The solution of the steepest ascent problem becomes a solution to our neural network. Since our cost function is a function of a series of activation functions in series, we can apply some calculus at each layer to determine what our error is and where the error propagated from. Thinking of our cost function as a function of weights and biases, we can imagine that finding the gradient of steepest descent as taking the partial derivatives of our weights and biases. As intuitive as this sounds, since we have a series of activation functions occurring at multiple layers, taking the partial derivative of each activation function would not give us the solutions we desire. Instead, we will use a backpropagation algorithm for a fully connected network in the next section.

### Backpropagation Algorithm

Since we cannot solve for the partial derivatives directly, we will follow the algorithm in functions 1.5-8 below. Using 4 equations, we can determine the error at each layer and the gradient of our cost function in terms of the weights and biases for each layer. For the proof on why these functions give us our gradient, see reference 1 (Chapter 2, section 2.5).

Functions 1.5-8 – The backpropagation functions

First, we will define some terms that are listed above to help make some sense of the formulas provided. It is an important assumption that these formulas are to be used on the vectors of inputs, weights, biases, and outputs rather than on individual inputs, outputs, weights, and biases. is the error at the last layer and is the first step to determining how to adjust our weights and biases. Since we do not know where the error originated from, we assume that the error has propagated from the beginning to the end. Solving for the error in the last layer is the first step in backpropagation. is the vector of our outputs’ gradients from the cost function as a function of our inputs, a. This comes from the outputs of one batch. Assuming we have a batch of inputs that we’ve fed forward through our neural network, we will run the results through the derivative of our cost function to determine the gradient at the output. We then perform element-wise multiplication of these results with . Where is the output of our activation function at the last layer, L. Therefore, we take the derivative of our activation function and perform element-wise multiplication.

After calculating the error at the last layer, , we can now calculate the error at the previous layer, . In equation 1.6, we can see that we take the weights of the next layer, , and transpose them. We then multiply this vector by the error calculated at the previous step, . Finally, this is element-wise multiplied by the derivative of the activation function of the current layer as a function of its input vector, .

Function 1.7 is telling us that we have solved for the error at the lth layer, and the jth element is the error at the bias, b. Since we have the gradient, we can now adjust our biases appropriately. To solve for the weights, we simply perform some multiplication to find the gradient at where j is the weight in our vector of weights for the lth layer and the kth neuron. Finding this gradient is simply taking the activation of the previous layer where it connects to neuron k, , and we multiply this by the error at our current layer at element j, .

As mentioned before, the choice of activation function plays a huge role in the speed of our neural network. Calculating the derivatives repeatedly of our activation function can be very costly when dealing with exponentials. ReLU has been shown to be more efficient at classification problems than linear or sigmoid activation functions. Ultimately, the choice of activation function boils down the what questions are a part of the problem statement and what type of answers we expect from our neural network. See chapter 3 – “Lessons Learned” for more details on activation function choices.

## Summary of Introduction

At this point, we have a basic grasp of what it takes to build a neural network. When learning about neural networks, the concept of building a neural network appears simple on the surface; but, the layers of complexity complicate the process. Although complex, the benefit of neural networks is the ability to solve complex problems. With this basic understanding of the structure and the layout of a neural network, we can create simple neural networks that are capable of solving classification problems, and forecasting data. After this introduction, we were able to build two neural networks that did just that. Our first neural network could recognize handwritten digits between 0 and 9, inclusive. After training, it could recognize handwritten digits it had never seen before with 95% accuracy. The second neural network accepted 3 columns of data as input and attempted to forecast a binary response variable as output. After training, it could successfully forecast with an 87% accuracy with new rows of data it had never seen before[[3]](#footnote-3).

# Types of Neural Networks Studied

With our knowledge of neural networks to this point, we can now begin to explore neural networks and their capabilities in further detail. In this chapter, we will explore the different types of algorithms and iterations of neural networks. We will discuss the pros and cons of each and which applications are best suited for the type of algorithm in question.

## Deep Learning

Deep learning is more of an umbrella term used for a network that contains multiple hidden layers. Deep learning assumes that the network has the ability to make more complicated decisions with the added complexity of multiple neurons. The theory behind a deep learning network is that the first layer has direct access to the inputs. These neurons make decisions on their outputs based on the raw input data. The next layer can see all the outputs of the initial inputs. The outputs of the first layer directly influence the second layer. The second layer now can see what the first layer deems as important. Depending on what the first layer weights heavily, the second layer can make even deeper decisions than what the first layer can do. After passing through multiple layers, the end result will be an output that has had many layers of decisions to determine the output. Typically, the more inputs there are, more intermediate layers are required. If the inputs have a linear relationship, a deep learning network might not be necessary.

### Fully Connected Neural Network

A fully connected (FC) neural network is a network where all outputs are connected to another input in the next layer. The only exception to this layout would be the output from the network. For example, The neural network shown in figure 1.2 is an FC. FCs are useful when the relationship among the inputs is unknown. Additionally, an FC is useful in general applications, but hard to specialize. For example, the forecasting problem introduced in 1.5, Summary of Introduction, was a fully connected network. Perhaps these results could be improved by utilizing a different type of network. We had no information about correlation of inputs and outputs. We thought, “what could an FC do if we chucked inputs at it?” For a general purpose network, an FC is very versatile and useful when creating a baseline for your results. The downside to an FC is that it is hard to know which nodes and layers are unnecessary and redundant. For example, if you have an extra layer that is redundant to the results, it will only take longer to train the network to achieve the same results. It would be hard to tell if the extra layer is necessary unless by trail and error. Additionally, the same could be said about the individual neurons within the network.

### Convolutional Neural Network

Convolutional neural networks (CNN) are a specialty type of network that will typically feed output directly into an FC. CNNs specialize in classification problems typically in the realm of images. Images are a particularly difficult issue to tackle when it comes to applying neural networks. This is because images vary in so many ways that it makes it hard to control input to a neural network. For example, a 100 x 100 black and white image contains 10,000 pixels. Each pixel will contain typically a value from 0 to 255. This is fairly simple input to deal with in a neural net. The counter example is the following: a 4k with full color. The resolution is typically 4096 x 2160 that contains 8,847,360 pixels. Each pixel has 3 or 4 separate values to distinguish color. The values are red, blue, green, and sometimes an alpha value to distinguish transparency. Each of those values range from 0 to 255. If we were going to attempt to assign this to a neural network, we would need at most, 4 x 8,847,360 = 35,389,440 input neurons. That is a massive network!

Where CNNs come into play is with the process of convolutions. Instead of focusing and weighing our inputs equally at first, this could become cumbersome. Instead, we take portions of the input, say a 20 x 20 set of pixels. Then, we apply a series of filters to this section. Sometimes, we are interested in the max of this region. Other times, we might be interested in the average values. The idea of taking the max of a region is called pooling. We implement pooling layers that take a section of the image and subdivide this region into smaller regions. We then pool these regions together by taking the max (or average, or minimum) value of these. Sometimes we are only interested in the general area where a max is and not necessarily the exact pixel of its location. We can combine these into a series of feature maps. We then subsample our feature maps. We can repeat this process (convolution and subsampling) a few times until the output is small enough for our FC ReLU neural network to process this data.

Another important key about looking at convolutions is the concept of spatial arrangement. We are looking at features and relationships of the pixels and performing convolutions to obtain a more valuable input to our network. Essentially, this process is “cleaning” our input and providing our network with more useful information as opposed to the entire image. If we had only looked at the raw data of the image at first, our network would first have to learn what is important and what information to ignore. This could take a network consisting of billions of neurons and millions of epochs to achieve. Even if a network that large was capable of learning what to look at, it would then have to learn how to classify the image. RNNs help to speed this process up.

### Recurrent Neural Network

Sometimes, it is important to factor in time. Perhaps a decision made during this time interval should be impacted by a decision made 20 time intervals ago.

## Genetic Algorithms/Programming

One issue with a FC neural network arises when considering the cost function associated with optimal results. When constructing a neural network, training the network is the fundamental key to good results. Another approach to training a neural network is to use genetic programming to generate a “species” that optimally solves a problem. A good application for genetic programming is a neural network that must make real-time decisions based on live input. It becomes difficult to write a cost function for a neural network that has consistently changing goals and a dynamic set of input data.

For example, if we wanted to create a network that could solve the first level of Mario, how do we incentivize the network to improve each frame the game is rendering to the screen? It is difficult to determine what the current goal is. We can, however identify some objectives that we want to accomplish. We can write a function based on distance traveled, coins collected, and the time it takes to accomplish these goals. The issue with this is that we cannot evaluate its performance during a level and backpropagate very easily. Alternatively, we can wait until the player dies or runs out of time. Then, we evaluate this behavior against our function. We can create new iterations of our littler plumber’s neural network with some “mutations” to the weights and biases (along with some other mutations to the network). Then run it again to see if this mutated version performs better or worse.

This is how genetic programming works. We can modify some of these rules to create more specific and specialized versions of our neural network that might work better in different situations. In this section we will explore 3 different genetic programming algorithms that we have learned about this semester.

### Genetic Learning

Genetic Learning is a learning algorithm modeled after Darwinian Evolution. A rough outline of these algorithms consist of a random population of Neural Networks being created, those networks being allowed to make decisions until an end state is reached. When the networks reach the end state, their performance is evaluated. The network with the highest performance, referred to as a fitness score, is chosen to start the next generation of networks. This next generation consists of clones of the previous best network with slight modifications referred to as mutations. Then the process is repeated until either a network is created that is incapable of reaching the end state or the user desires to stop. We implemented two algorithms from scratch that have this structure and implemented a third using a library.

The first algorithm was created with the intention of a green box, controlled by a neural network, towards a red box. To visualize our results, we used a library called pygame for drawing shapes and images to a window. Our end state for this problem was considered either a green box leaving the bounds of the window, a green box colliding with the red box or the generation timing out (4 seconds passes). We considered our fitness score to be . This incentivizes the green boxes to get as close to the goal as they can as quickly as possible. To initialize a population of 50 random networks we chose a fixed network structure of 100 hidden neurons. The weights for the initial networks are sampled from a normal distribution with a mean of 0 and a standard deviation of 10. This further allows for complex movements. We have a single parent for each generation after the first generation and mutate the weights by adding each weight by a sample from a normal distribution with a mean of 0 and a standard deviation of ½. This has the effect of many small changes in weights compounding to large changes in behavior that potentially negatively or positively influence the network’s fitness score. This crude first attempt at a genetic algorithm still allows for the networks to converge to a solution after a few generations.

### NEAT

The second genetic algorithm implemented is a stepping stone to the NEAT algorithm. NEAT stands for Neuro-evolution of Augmenting Topologies. Instead of how we initialized our networks in our first genetic algorithm, we instead initialize our networks with a single hidden node and our mutations are: add\_node (adding a hidden node), add\_connection (adds a connection between two nodes that didn’t previously exist), and change\_weight (changes the weight by adding a number on the interval [-2, 2].) We still only had a single parent in this implementation. To test this next algorithm, we created a recreation of several year old phone game called “Flappy Bird”. The game has a bird jump through small gaps between pipes. In this first version, the bird and pipes are just represented with a blue and green box(es) respectfully.  The goal was to have the network play the game perfectly. However, given this limited implementation we were only able to get a maximum score of 5.

Our final implementation was of the full NEAT algorithm. The addition here is our new populations are created from multiple parents. This small change and most likely a more elegant implementation of the algorithm since we used a library (neat-python) allowed us to create a perfect network for playing Flappy Bird.

## Reinforcement Learning

We applied a technique referred to as Q-Learning (short for Quality Learning). The basic principle of the Q-Learning algorithm is to separate the problem space into discrete states. Each of these states is assigned a “reward” (negative rewards are referred to as “punishments”) then the goal of the program is to maximize the reward it receives. To practice implementing this algorithm, we set out to create a Neural Network that could learn to solve mazes.

The maze was created as a 10x10 tile map with tiles 64x64 pixels in size. In reference to the maze, we refer to x as the column number of the maze and y as the row number of the maze. Our reward function was declared as the distance formula (xFinish-xPlayer)2-(yFinish-yPlayer)2  with the walls of the maze being assigned a value of -210. We created a recurrent neural network to approximate this function, following the path of increasing rewards. The inputs for the network are the current x, current y, and previous direction moved (direction is encoded as 0 = up, 1 = down, 2 = left, 3 = right). A problem we discovered was that the neural network was not appearing to learn from its mistakes on the maze. What we chose to do to fix this was to train it for 100 iterations of back propagation at each tile. This artificially increases our input size and allows for the neural network to learn.

The speed of learning the path to the finish is outweighed (at least in this case) by the potential negative of running the backpropagation algorithm 99 additional times. Another problem we discovered was that our maze had an edge case. It is possible to have two divergent paths, one of which leads to a dead end, the other leads to the finish and both of these paths are the same distance away from the finish. We chose to solve this by adding a small amount of bias to the reward function, we subtracted the reward value of the tile leading to the dead end by 1 so it was not seen as an identical option to the correct path.

A more elegant way to implement this which would avoid adding bias would be to remove the reward as a function of distance. Instead, we would not assign rewards at all. We instead would have the network continue moving until it started to run into walls, the more walls the network ran into, the more negative of a reward it would associate with the tile it occupied. In the case of finding a dead end, the network would see itself cornered by walls and a highly negative reward value would be associated with that tile. It would then begin to backtrack out of the path, backpropagating that negative association along the way.

# Summary and Lessons Learned

# References

1. Although figure 2.2 has multiple connections for outputs from many of the neurons in the hidden layers, the value that is output from the neuron is the same across all outputs for that neuron. [↑](#footnote-ref-1)
2. Process of accepting inputs, calculating the output, and feeding the result to the next layer. [↑](#footnote-ref-2)
3. We recognize that 87% is not all that great compared to mathematical models; but, this is without cleaning our inputs, performing PCA, or adjusting our data inputs in any way. For a neural network to create correlation among the data as input to the output with no context was impressive to us. We have plans to attempt to improve this network and the data to increase our success rate in the future. [↑](#footnote-ref-3)