1 Introduction

Neural networks have a lot of applications in today's world. They have applications in character recognition, image compression, stock market prediction, medicine, and much more [2]. They provide us with a way to compute for more complicated things.

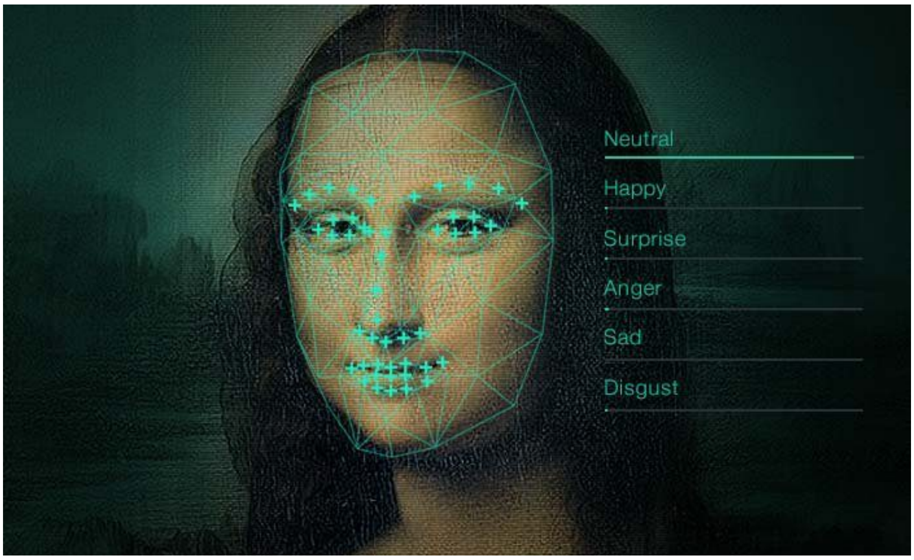

Figure 1: Face Recognition Example [4]

When neural networks are mentioned, people tend to panic as they think it is a very complicated topic. However, it is not incredibly complicated. For people with a background in linear algebra, neural networks become something more tangible, because it is just applying concepts in linear algebra.

For this topic, we will discuss a basic kind of neural network. It is not too complicated (like the modern versions of neural networks) and it is easy to follow.

2 Preliminaries

2.1 Activation Function

The activation function is simply a function that squeezes numbers to fit into a certain range. In this case, we try to fit numbers in the range $[0, 1]$. There are many kinds of activation functions, and the function we use for this topic is the sigmoid function. The sigmoid function is defined as

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

The derivative for this function is defined as

$$\sigma'(x) = \sigma(x)\left(1 - \sigma(x)\right)$$

2.2 Gradient Descent

Gradient descent is an algorithm that "tweaks it's parameters iteratively to minimize a given function to its local minimum [3]." In essence, it is a way to minimize some type of cost function.

Suppose we have a 2-dimensional curve on the Cartesian plane. If we consider any point on the plane, it is possible to determine if there is a downward slope either to the left or to the right of the point by getting the derivative of the function on that point.

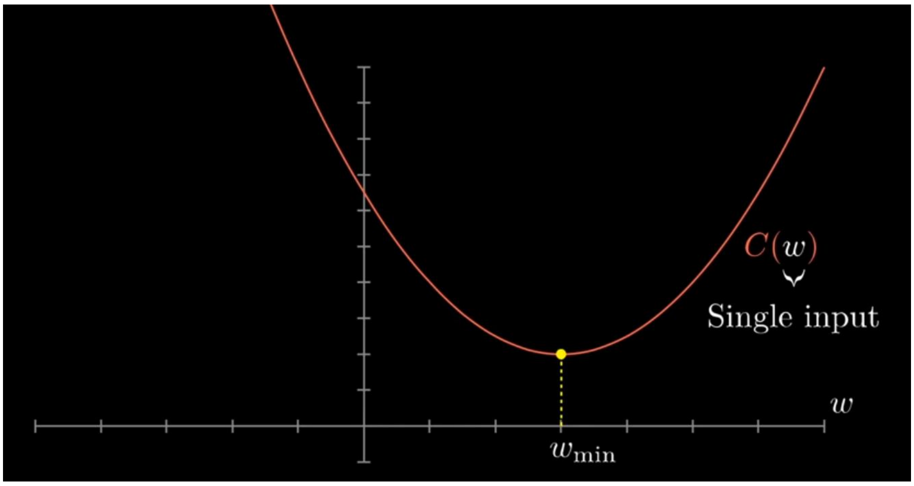

Figure 2: Point at a local minima[1]

This is useful because it can also tell us where the point should adjust to in order to get closer to the (or one of the) local minima of the function. Namely, make the point adjust to its left if the slope at that point is positive, and to its right if the slope is negative.

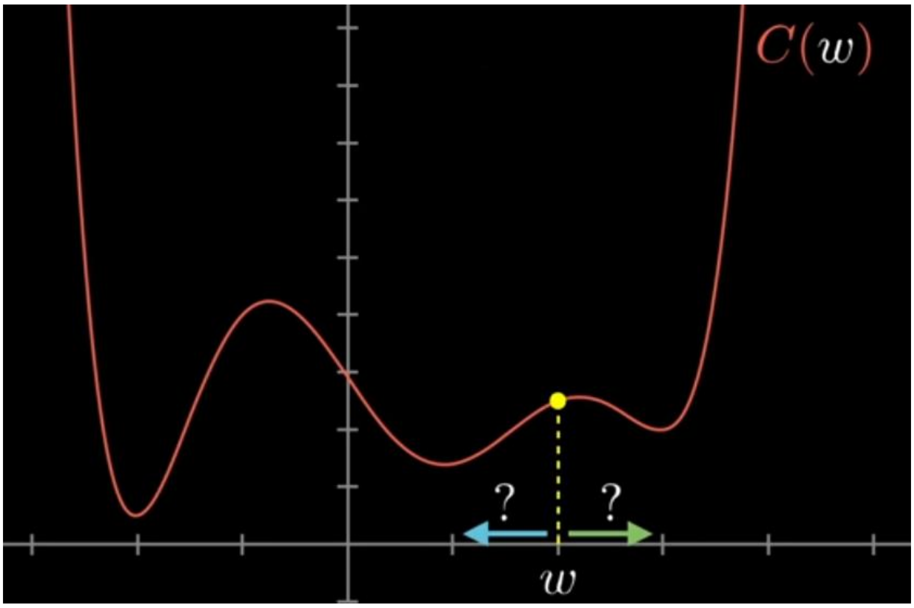

Figure 3: Deciding where to adjust given a point on a curve[1]

By extension, this can work for an n-dimensional figure. For example, this could work with a 3-dimensional curve because we could get the partial derivative of each point in the space.

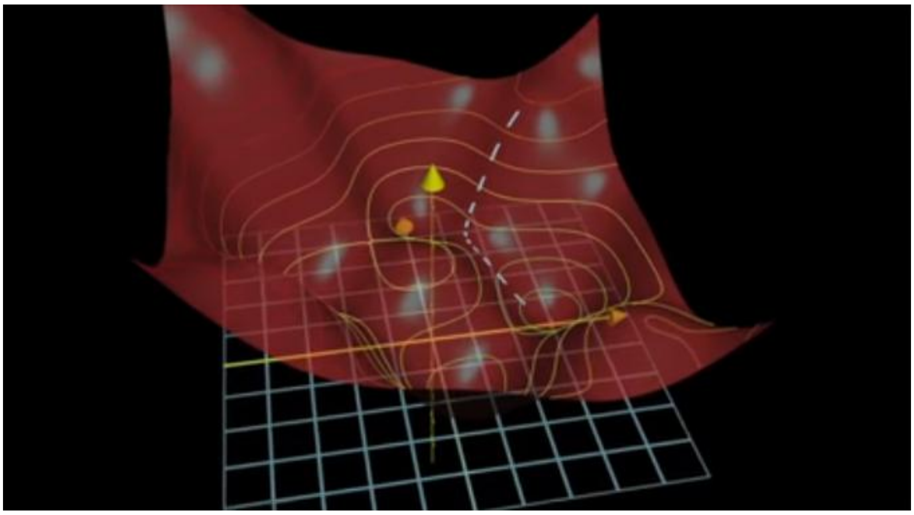

Figure 4: Example of a 3D curve with local minima[1]

Thus, for each point, we iteratively adjust it closer to a local minima. When this is applied repeatedly in units proportional to the slope of the curve at each point, you get what is called gradient. This gradient, when configured to move towards a local minima, is called a gradient descent.

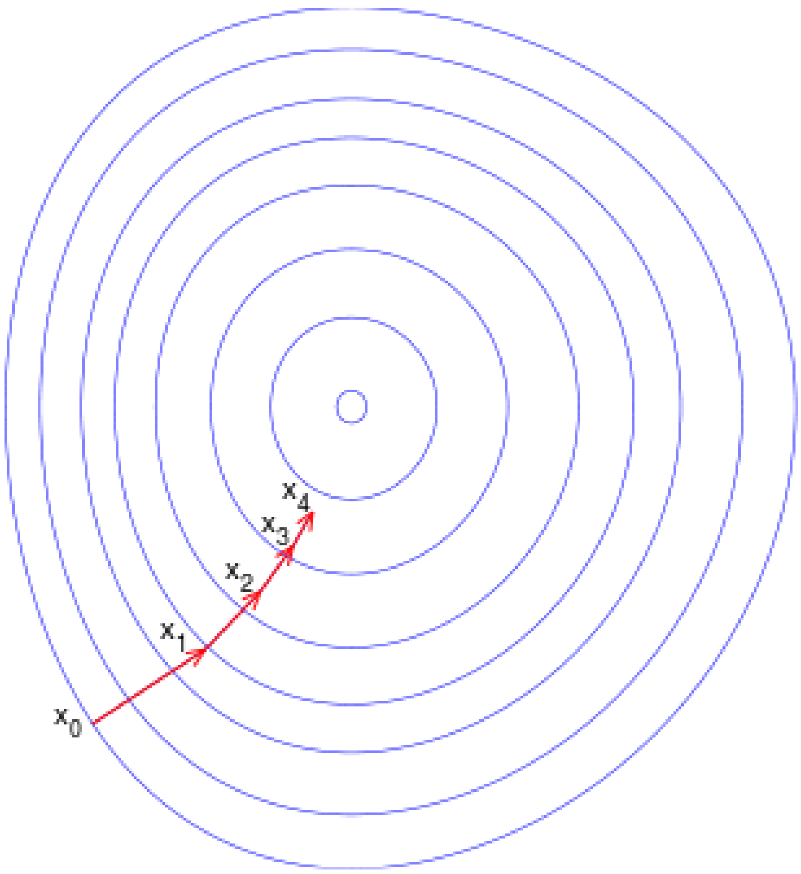

Figure 5: Iteratively approaching a minimum[1]

2.3 Definition of Terms

Before discussing how the neural network works, we must define the following terms:

Neurons
These are represented as the nodes in the neural network.

Layers
These are the vertical arrangements of neurons in the neural network.

Weights
These are represented as the edges connecting one neuron to the all neurons in the next layer.

3 Discussion

The neural network is composed of different phases, each having an effect to the actual network.

3.1 Forward Phase

A neural network consists of an input layer, an arbitrary amount of hidden layers, and an output layer. Each layer consists of an arbitrary amount of neurons depending on the purpose of the neural network.

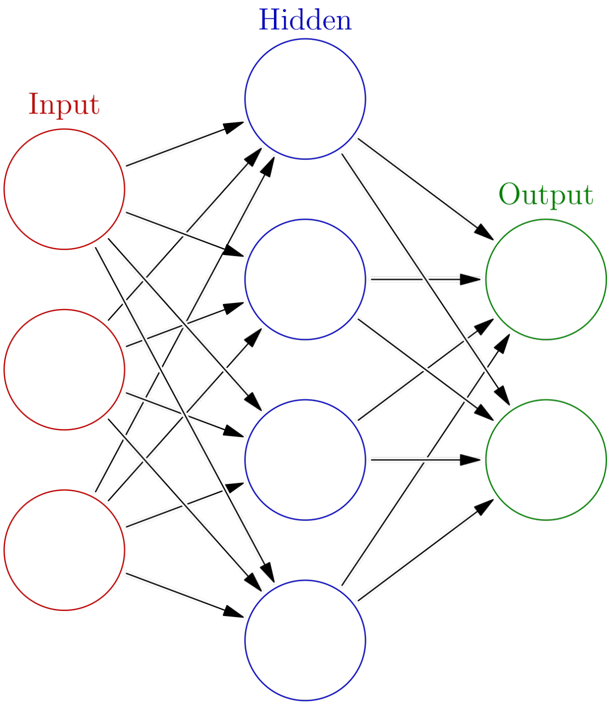

Figure 6: Neural network illustrated

The value of each neuron in the hidden and output layers is expressed by

$$k = \sigma\left(\sum_{i=1}^{n}(w_i a_i) + B\right)$$

where k is the value of a neuron in the current layer, w_i is the weight of the edge connecting the current neuron and the ith neuron from the previous layer, a_i is the value of the ith neuron from the previous layer, and B is a "bias" value.

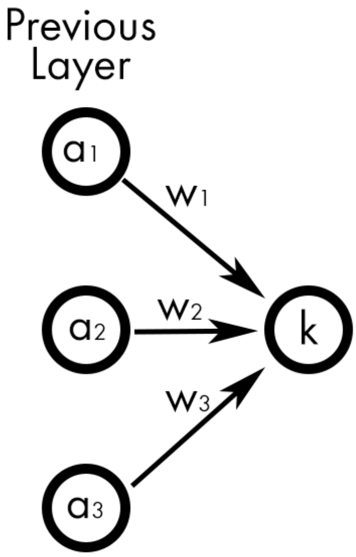

Figure 7: Illustration of the value of k

This phase can be done with linear algebra. If we look at the equation to find the value of a neuron in a layer, this is essentially multiplying a $m \times 1$ matrix to a 1×1 matrix. By extension, we can express the edges connecting one layer to another layer as a $m \times n$ matrix, with m as the number of neurons in the current layer, and n as the number of neurons in the previous layer. It will look something like so,

$$\begin{bmatrix} w_{0,0} & w_{0,1} & \cdots & w_{0,n} \\ w_{1,0} & w_{1,1} & \cdots & w_{1,n} \\ \vdots & \vdots & \ddots & \vdots \\ w_{m,0} & w_{m,1} & \cdots & w_{m,n} \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_n \end{bmatrix} = \begin{bmatrix} k_0 \\ k_1 \\ \vdots \\ k_m \end{bmatrix}.$$

Thus, the equation for the value of some neuron i is computed as follows:

$$k_i = \sigma\left(\sum_{j=1}^{n}(w_{i,j}a_j) + B_i\right).$$

3.2 Checking for Error

The values in the last layer would ideally contain the values on how the neural network classifies the input data. However, if the neural network is not that good, then it will wrongly classify the input data. Thus, it is important to check how much error the network made.

To do this, the input data must be accompanied with a correct answer so that the network would know how to respond when it sees the same thing again. There are multiple ways to check for the error of some output to some data, but for our implementation, we used the following equation:

$$\text{error} = (k_i - c_i)^2$$

where c_i is the ideal value of the neuron.

If we manipulate the neural network in such a way so as to minimize the “wrongness” of each output, we can force it to ”learn” from its mistakes, and this is where gradient descent is useful.

3.3 Backward Phase

Given that we know how much error the neural network produced with one forward phase, we have enough information to correct this mistake a bit. This will be done the same way how gradient descent is done.

In back-propagation, the main focus is to use the errors from the previous feed forward phase in order to correct the weights between each adjacent pair of layers. For this phase, it is needed to compute for the gradient values for each layer, and that information is needed to compute for the new set of weights that will be used in the succeeding iterations of the feed forward phase. The pseudocode for the back propagation step is shown below:

Listing 1: Back Propagation Pseudocode

```
Input: error rates from output layer
Output: new set of weights
BackPropagation(error_rates):
 # first step is to compute for weights between output layer and last hidden layer
 # yi represents the ith value in the output layer
 # f'(x) represents the derivative of the activation function for the current layer
 derivatives = f'(y[i]) for i in each computed output value
 gradients = error_rates[i] * derivatives[i] for i in each neuron #result is n x 1 matrix
 delta_w = gradients * last_hidden_layer #last_hidden_layer transposed to become row matrix
 new_weights_to_output = previous_weights_to_output - delta_w #previous weights between last
     hidden layer and output layer is replaced with new_weights

 # succeeding steps for each hidden layer until the input
 gradients = transpose(gradients)
 current_layer = last_hidden_layer
 while current_layer != input_layer:
  derivatives = f'(current_layer[i]) for i in each computed value in current_layer
  gradients = gradients * transpose(previous_weights)
  gradients[i] = gradients[i] * derivatives[i]
  delta_w = left_layer * gradients #left_layer is the layer to the left of current_layer;
      expressed as a column vector
  new_weights_to_current = previous_weights_to_current - delta_w
  current_layer = left_layer

 return set of new_weights
```

3.4 Training the Network

Now that we've learned about the forward and backward phase, we now know how a network learns. By feeding the neural network with input data with correct answers, we can "train" the network by letting it run the forward phase and it will be corrected by checking the error and it will attempt to fix the error with the backward phase.

With enough training data, the neural network should be modified enough to do what it has to do.

3.5 Getting an Answer

With a trained network, looking at the values in the output layer will give us an answer. Each neuron would contain a value which represents how "similar" the input data is to its classification. Getting the maximum among the other neurons of the output layer would be the decision of the neural network.

3.6 Sample Code in SageMath

The following are the code snippets we coded for this project.

3.6.1 Neuron Class

Listing 2: Neuron Class

```
class Neurons:
    nodes = []
    def __init__(self,n):
        self.nodes = n
    def getLength(self):
        return len(self.nodes)
    def get(i):
        return self.nodes[i]
```

```
    def toRow(self):
        return matrix(self.nodes).transpose()
    def toColumn(self):
        return matrix(self.nodes)
```

3.6.2 Weight Class

Listing 3: Weight Class

```
class Weights:
    w = []
    def __init__(self,n):
        self.w = n
    def getRows(self):
        return len(self.w)
    def getColumns(self):
        return len(self.w[0])
    def toMatrix(self):
        return matrix(self.w)
```

3.6.3 Important Math Functions

Listing 4: Important Math Functions

```
def sigmoid(x):
    return 1/(1+e^(-x))
def relu(x):
    return max(0,x)
def sigmoidPrime(x):
    return x*(1-x)
def reluPrime(x):
    if x > 0:
        return 1
    else:
        return 0
```

3.6.4 Gradient Multiply

Listing 5: Gradient Multiply

```
def gradientMultiply(a, b):
    a_rows = len(a)
    a_cols = len(a[0])
    b_rows = len(b)
    b_cols = len(b[0])
    if a_rows > 1 or b_rows > 1 or a_cols != b_cols:
        return None
    result = []
    for i in xrange(a_cols):
        result[0][i] = a[0][i]*b[0][i]
    return result
```

3.6.5 Forward Phase

Listing 6: Forward Phase

```
def feedForward(inputI, output, topology, isSigmoid, w):
    #input and output both have one row
    error = [0 for x in xrange(len(output[0]))]
    nodes.append(Neurons(inputI[0]))
    hidden = []
    if w is None: #random weights
        w = Weights(len(topology)-1)
        for i in xrange(len(w)):
            weights = [[random() for j in xrange(len(w))] for k in xrange(len(w))]
            w[i] = Weights(weights)
```

```
    for i in xrange(len(w)):
        weightList.append(w[i])
        weights = w[i].toMatrix()
        if i == 0:
            hidden = matrix(inputI)*weights
        else:
            hidden = matrix(hidden)*weights
        if isSigmoid[i]:
            for j in xrange(len(hidden[0])):
                hidden[0,j] = sigmoid(hidden[0][j])
        else:
            for j in xrange(len(hidden[0])):
                hidden[0,j] = relu(hidden[0][j])
        nodes.append(Neurons(hidden[0]))
        # if the network is trained, then just return the last layer
    for i in xrange(len(error)):
        error[i] = (output[0][i]-hidden[0][i])**2
    return error
```

3.6.6 Backward Phase

Listing 7: Backward Phase

```
def backProp(error, isSigmoidA):
    newWeights = []
    index = len(weightList)-1
    last = nodes[-1]
    nodes = nodes[:-1]
    derivatives = []
    if(isSigmoid(index)):
        for i in xrange(len(derivatives[0])):
            derivatives[0,i] = sigmoidPrime(last.get(i))
    else:
        for i in xrange(len(derivatives)):
            derivatives[0,i] = reluPrime(last.get(i))
    #print stuff
    gradients = []
    for i in xrange(len(error)):
        gradients[i][0] = error[i]*derivatives[0][i]
    #getting new weights between output and last hidden layer
    nextLayer = nodes[-1]
    gradients2 = matrix(gradients)
    delta = gradients2*nextLayer.toRow()
    prev = matrix(weightList[-1]) #not sure lol
    weightList = weightList[:-1]
    bago = prev - delta.transpose()
    newWeights[index] = Weights(bago)
    index = index - 1
    #getting new weights between the other layers
    gradients2 = gradients2.transpose()
    while not (weightList == []):
        last = nextLayer
        nodes = nodes[-1]
        derivatives = []
        if isSigmoid[index]:
            for i in xrange(len(derivatives[0])):
                derivatives[0,i] = sigmoidPrime(last.get(i))
        else:
            for i in xrange(len(derivatives)):
                derivatives[0,i] = reluPrime(last.get(i))
        gradients2 = gradients2*(prev.transpose())
        gradients2 = gradientMultiply(gradients2, derivatives)
        nextLayer = nodes[-1]
        z = nextLayer.toColumn()
        prev = weightList[-1].toMatrix()
        weightList = weightList[:-1]
        delta = z*gradients2
        bago = prev - delta
        newWeights[index] = Weights(bago)
    return newWeights
```

3.6.7 Full Code

Listing 8: Full Code

```
class Neurons:
    nodes = []
    def __init__(self,n):
        self.nodes = n
    def getLength(self):
        return len(self.nodes)
    def get(i):
        return self.nodes[i]
    def toRow(self):
        return matrix(self.nodes).transpose()
    def toColumn(self):
        return matrix(self.nodes)

class Weights:
    w = []
    def __init__(self,n):
        self.w = n
    def getRows(self):
        return len(self.w)
    def getColumns(self):
        return len(self.w[0])
    def toMatrix(self):
        return matrix(self.w)

def sigmoid(x):
    return 1/(1+e^(-x))
def relu(x):
    return max(0,x)
def sigmoidPrime(x):
    return x*(1-x)
def reluPrime(x):
    if x > 0:
        return 1
    else:
        return 0
def gradientMultiply(a, b):
    a_rows = len(a)
    a_cols = len(a[0])
    b_rows = len(b)
    b_cols = len(b[0])
    if a_rows > 1 or b_rows > 1 or a_cols != b_cols:
        return None
    result = []
    for i in xrange(a_cols):
        result[0][i] = a[0][i]*b[0][i]
    return result

nodes = []
weightList = []

def clearWeights():
    weightList = []

def getWeights():
    w = [weightList[i] for i in xrange(len(weightList))]
    return w

def setWeights(w):
    clearWeights()
    for i in xrange(len(w)):
        weightList.append(w[i])

def feedForward(inputI, output, topology, isSigmoid, w):
    #input and output both have one row
    error = [0 for x in xrange(len(output[0]))]
    nodes.append(Neurons(inputI[0]))
    hidden = []
    if w is None: #random weights
        w = Weights(len(topology)-1)
        for i in xrange(len(w)):
            weights = [[random() for j in xrange(len(w))] for k in xrange(len(w))]
            w[i] = Weights(weights)
    for i in xrange(len(w)):
        weightList.append(w[i])
        weights = w[i].toMatrix()
```

```
            if i == 0:
                hidden = matrix(inputI)*weights
            else:
                hidden = matrix(hidden)*weights
            if isSigmoid[i]:
                for j in xrange(len(hidden[0])):
                    hidden[0,j] = sigmoid(hidden[0][j])
            else:
                for j in xrange(len(hidden[0])):
                    hidden[0,j] = relu(hidden[0][j])
            nodes.append(Neurons(hidden[0]))
           # if the network is trained, then just return the last layer
        for i in xrange(len(error)):
            error[i] = (output[0][i]-hidden[0][i])**2
        return error

def backProp(error, isSigmoidA):
    newWeights = []
    index = len(weightList)-1
    last = nodes[-1]
    nodes = nodes[:-1]
    derivatives = []
    if(isSigmoid(index)):
        for i in xrange(len(derivatives[0])):
            derivatives[0,i] = sigmoidPrime(last.get(i))
    else:
        for i in xrange(len(derivatives)):
            derivatives[0,i] = reluPrime(last.get(i))
    #print stuff
    gradients = []
    for i in xrange(len(error)):
        gradients[i][0] = error[i]*derivatives[0][i]
    #getting new weights between output and last hidden layer
    nextLayer = nodes[-1]
    gradients2 = matrix(gradients)
    delta = gradients2*nextLayer.toRow()
    prev = matrix(weightList[-1]) #not sure lol
    weightList = weightList[:-1]
    bago = prev - delta.transpose()
    newWeights[index] = Weights(bago)
    index = index - 1
    #getting new weights between the other layers
    gradients2 = gradients2.transpose()
    while not (weightList == []):
        last = nextLayer
        nodes = nodes[-1]
        derivatives = []
        if isSigmoid[index]:
            for i in xrange(len(derivatives[0])):
                derivatives[0,i] = sigmoidPrime(last.get(i))
        else:
            for i in xrange(len(derivatives)):
                derivatives[0,i] = reluPrime(last.get(i))
        gradients2 = gradients2*(prev.transpose())
        gradients2 = gradientMultiply(gradients2, derivatives)
        nextLayer = nodes[-1]
        z = nextLayer.toColumn()
        prev = weightList[-1].toMatrix()
        weightList = weightList[:-1]
        delta = z*gradients2
        bago = prev - delta
        newWeights[index] = Weights(bago)
    return newWeights
```

4 Results

To test the code if it's actually working, the following snippet of code was used. This may also be used as a template to provide further inputs. The following test case demonstrates the forward phase and its resulting error values from the first run in the neural network.

Listing 9: Sample Input

```
in1 = [[0.2,0.1,0.3,0.5]]
```

```
out1 = [[1,0,0]]
act = [True,True]
top = [4,2,3]
wait1 = [[0.15,0.14],[0.02,0.24],[0.62,0.2],[0.34,0.25]]
wait2 = [[0.22,0.07,0.58],[0.59,0.55,0.77]]
bigat = [Weights(wait1), Weights(wait2)]
wth = feedForward(in1,out1,top,act,bigat)
print(wth)
```

```
[0.149598156939953, 0.343876073586092, 0.468967401097846]
```

Figure 8: Results of forward phase

5 Conclusion

Neural networks are prevalent in many technologies we have today and the theoretical concepts behind these rely heavily on concepts from linear algebra, statistics, and calculus. The study of neural networks is relatively recent and there are many ongoing studies about the different classifications of neural networks and how each of them can be used in different areas, and as well as possible improvements in terms of complexity and accuracy.

References

[1] 3Blue1Brown. *Gradient descent, how neural networks learn — Deep learning, chapter 2*. Oct. 2017. URL: https://www.youtube.com/watch?v=IHZwWFHWa-w.

[2] *Applications of neural networks*. URL: https://cs.stanford.edu/people/eroberts/courses/soco/projects/neural-networks/Applications/index.html.

[3] Niklas Donges. *Gradient Descent in a Nutshell – Towards Data Science*. Mar. 2018. URL: https://towardsdatascience.com/gradient-descent-in-a-nutshell-eaf8c18212f0.

[4] Ed Targett, James Nunns, and Tom Ball. *Home Office: We're Might Give Police Facial Recognition on their Phones*. July 2018. URL: https://www.cbronline.com/news/automatic-facial-recognition.