

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 [dk1]. They provide us with a way to compute for more complicated things.

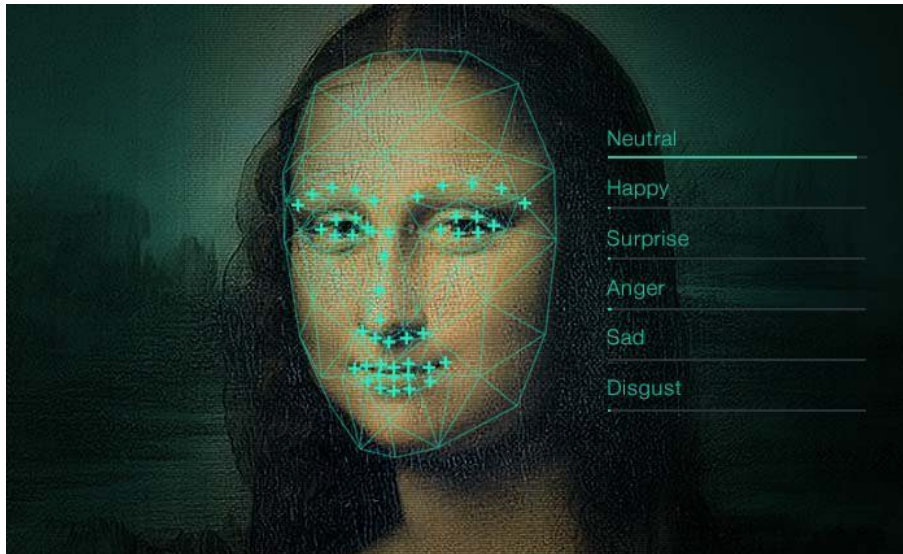


Figure 1: Face Recognition Example [targett'nunns'ball'2018]

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

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

2.2 Gradient Descent

Gradient descent is an algorithm that "tweaks it's parameters iteratively to minimize a given function to its local minimum [donges'2018]." 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.

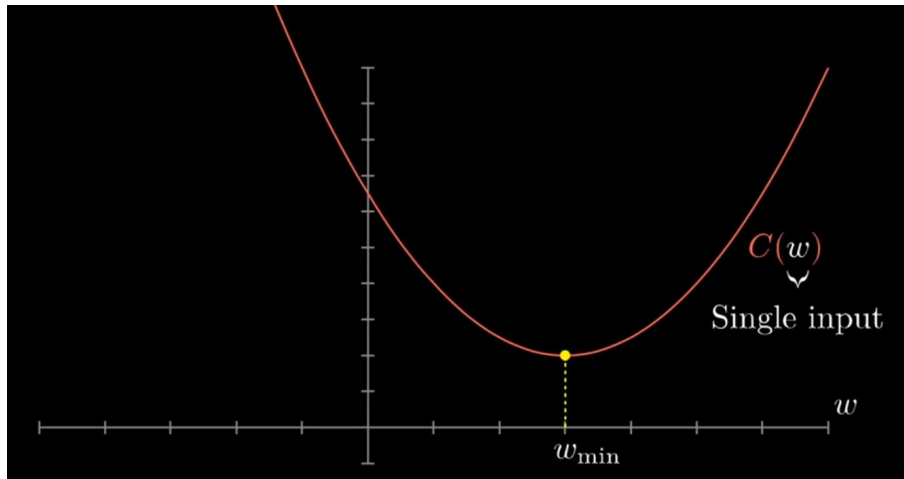


Figure 2: Point at a local minima[3blue1brown'2017'2]

This is useful because it can also tell us where the point should adjust 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.

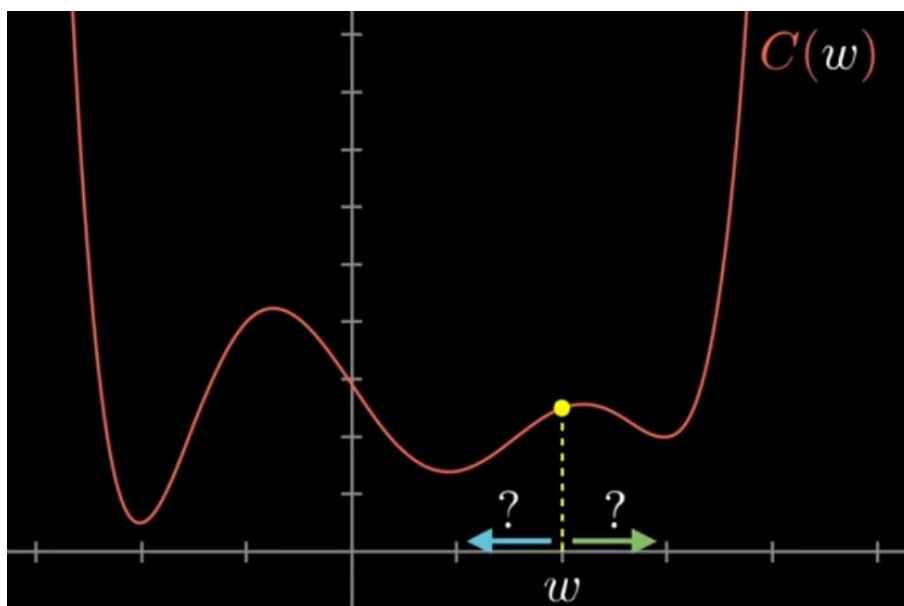


Figure 3: Deciding where to adjust given a point on a curve[3blue1brown'2017'2]

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.

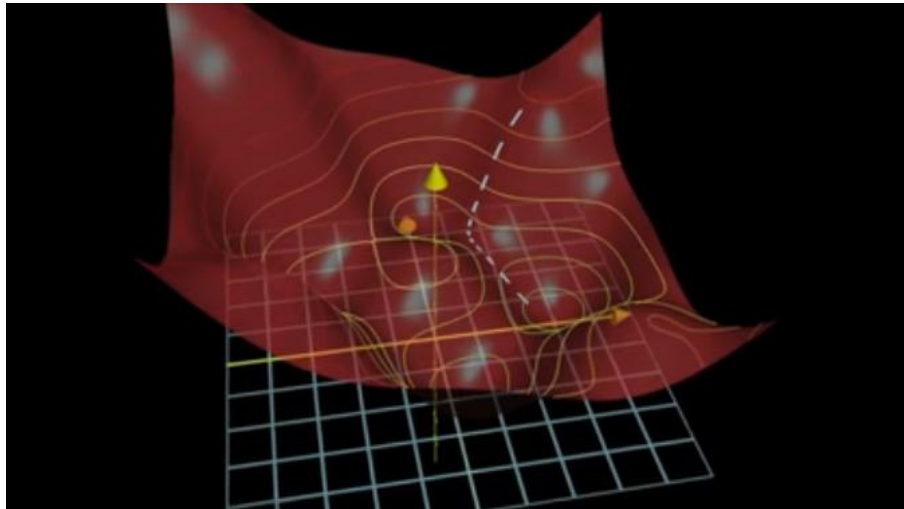


Figure 4: Example of a 3D curve with local minima[3blue1brown'2017'2]

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.

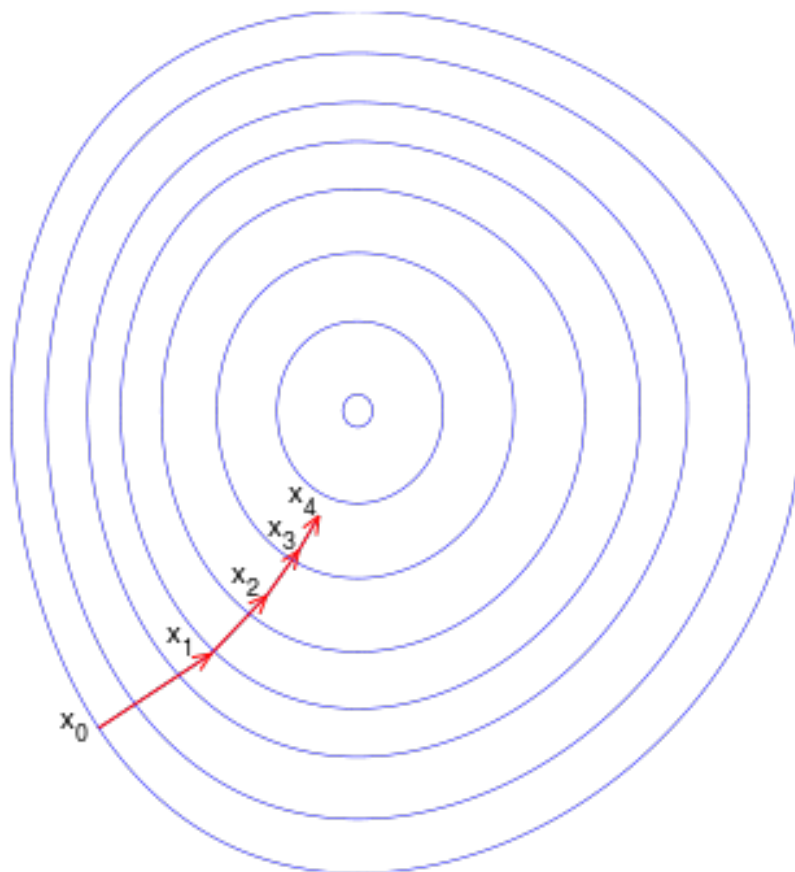


Figure 5: Iteratively approaching a minimum[3blue1brown'2017'2]

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.

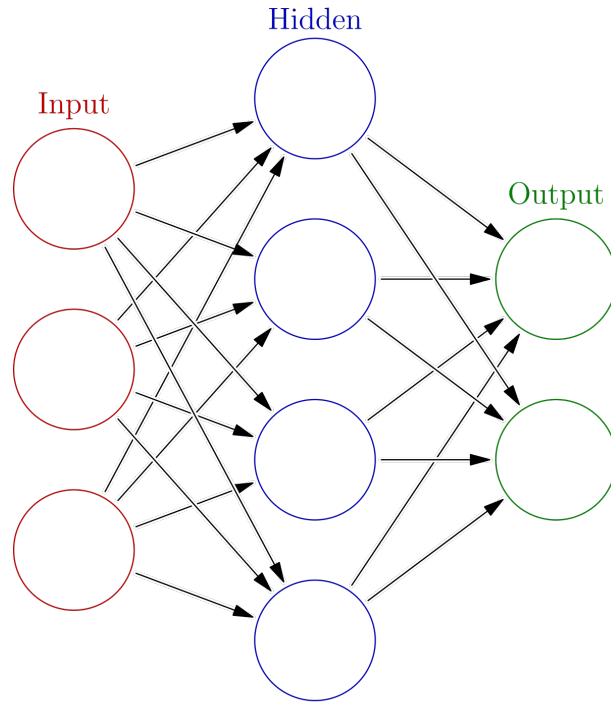


Figure 6: Neural network illustrated

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 i th neuron from the previous layer, a_i is the value of the i th neuron from the previous layer, and B is a "bias" value.

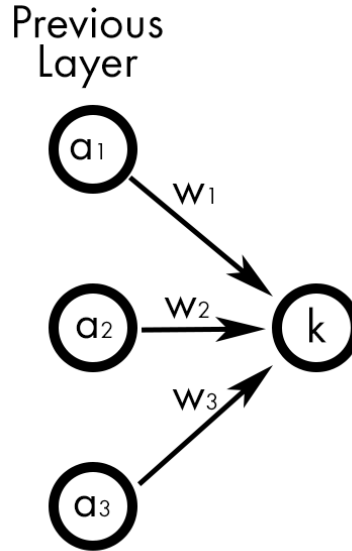


Figure 7: Illustration of the value of k

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

```

1 Input: error rates from output layer
2 Output: new set of weights
3 BackPropagation(error_rates):
4   # first step is to compute for weights between output layer and last hidden layer
5   # yi represents the ith value in the output layer
6   # f'(x) represents the derivative of the activation function for the current layer
7   derivatives = f'(y[i]) for i in each computed output value
8   gradients = error_rates[i] * derivatives[i] for i in each neuron #result is n x 1 matrix
9   delta_w = gradients * last_hidden_layer #last_hidden_layer transposed to become row matrix
10  new_weights_to_output = previous_weights_to_output - delta_w #previous weights between last
    hidden layer and output layer is replaced with new_weights
11
12  # succeeding steps for each hidden layer until the input
13  gradients = transpose(gradients)
14  current_layer = last_hidden_layer
15  while current_layer != input_layer:
16    derivatives = f'(current_layer[i]) for i in each computed value in current_layer
17    gradients = gradients * transpose(previous_weights)
18    gradients[i] = gradients[i] * derivatives[i]
19    delta_w = left_layer * gradients #left_layer is the layer to the left of current_layer;
    expressed as a column vector
20    new_weights_to_current = previous_weights_to_current - delta_w
21    current_layer = left_layer
22
23  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

3.6.1 Neuron Class

Listing 2: Neuron Class

```

1 class Neurons:
2     nodes = []
3     def __init__(self,n):
4         self.nodes = n
5     def getLength(self):
6         return len(self.nodes)
7     def get(i):
8         return self.nodes[i]
9     def toRow(self):
10        return matrix(self.nodes).transpose()
11    def toColumn(self):
12        return matrix(self.nodes)

```

3.6.2 Weight Class

Listing 3: Weight Class

```

1 class Weights:
2     w = []
3     def __init__(self, n):
4         self.w = n
5     def getRows(self):
6         return len(self.w)
7     def getColumns(self):
8         return len(self.w[0])
9     def toMatrix(self):
10        return matrix(self.w)

```

3.6.3 Important Math Functions

Listing 4: Important Math Functions

```

1 def sigmoid(x):
2     return 1/(1+e^(-x))
3 def relu(x):
4     return max(0,x)
5 def sigmoidPrime(x):
6     return x*(1-x)
7 def reluPrime(x):
8     if x > 0:
9         return 1
10    else:
11        return 0

```

3.6.4 Gradient Multiply

Listing 5: Gradient Multiply

```

1 def gradientMultiply(a, b):
2     a_rows = len(a)
3     a_cols = len(a[0])
4     b_rows = len(b)
5     b_cols = len(b[0])
6     if a_rows > 1 or b_rows > 1 or a_cols != b_cols:
7         return None
8     result = []
9     for i in xrange(a_cols):
10        result[0][i] = a[0][i]*b[0][i]
11    return result

```

3.6.5 Forward Phase

Listing 6: Forward Phase

```

1 def feedForward(inputI, output, topology, isSigmoid, w):
2     #input and output both have one row
3     error = [0 for x in xrange(len(output[0]))]
4     nodes.append(Neurons(inputI[0]))
5     hidden = []
6     if w is None: #random weights
7         w = Weights(len(topology)-1)
8         for i in xrange(len(w)):
9             weights = [[random() for j in xrange(len(w))] for k in xrange(len(w))]
10            w[i] = Weights(weights)
11    for i in xrange(len(w)):
12        weightList.append(w[i])
13        weights = w[i].toMatrix()
14        if i == 0:
15            hidden = matrix(inputI)*weights
16        else:
17            hidden = matrix(hidden)*weights

```

```
18         if isSigmoid[i]:
19             for j in xrange(len(hidden[0])):
20                 hidden[0,j] = sigmoid(hidden[0][j])
21         else:
22             for j in xrange(len(hidden[0])):
23                 hidden[0,j] = relu(hidden[0][j])
24         nodes.append(Neurons(hidden[0]))
25         # if the network is trained, then just return the last layer
26     for i in xrange(len(error)):
27         error[i] = (output[0][i]-hidden[0][i])**2
28     return error
```

3.6.6 Backward Phase

Listing 7: Backward Phase

```
1 def backProp(error, isSigmoidA):
2     newWeights = []
3     index = len(weightList)-1
4     last = nodes[-1]
5     nodes = nodes[:-1]
6     derivatives = []
7     if(isSigmoid(index)):
8         for i in xrange(len(derivatives[0])):
9             derivatives[0,i] = sigmoidPrime(last.get(i))
10    else:
11        for i in xrange(len(derivatives)):
12            derivatives[0,i] = reluPrime(last.get(i))
13    #print stuff
14    gradients = []
15    for i in xrange(len(error)):
16        gradients[i][0] = error[i]*derivatives[0][i]
17    #getting new weights between output and last hidden layer
18    nextLayer = nodes[-1]
19    gradients2 = matrix(gradients)
20    delta = gradients2*nextLayer.toRow()
21    prev = matrix(weightList[-1]) #not sure lol
22    weightList = weightList[:-1]
23    bago = prev - delta.transpose()
24    newWeights[index] = Weights(bago)
25    index = index - 1
26    #getting new weights between the other layers
27    gradients2 = gradients2.transpose()
28    while not (weightList == []):
29        last = nextLayer
30        nodes = nodes[-1]
31        derivatives = []
32        if isSigmoid[index]:
33            for i in xrange(len(derivatives[0])):
34                derivatives[0,i] = sigmoidPrime(last.get(i))
35        else:
36            for i in xrange(len(derivatives)):
37                derivatives[0,i] = reluPrime(last.get(i))
38        gradients2 = gradients2*(prev.transpose())
39        gradients2 = gradientMultiply(gradients2,derivatives)
40        nextLayer = nodes[-1]
41        z = nextLayer.toColumn()
42        prev = weightList[-1].toMatrix()
43        weightList = weightList[:-1]
44        delta = z*gradients2
45        bago = prev - delta
46        newWeights[index] = Weights(bago)
47    return newWeights
```

3.6.7 Full Code

Listing 8: Full Code

```
1 class Neurons:
2     nodes = []
3     def __init__(self,n):
```



```
4         self.nodes = n
5     def getLength(self):
6         return len(self.nodes)
7     def get(i):
8         return self.nodes[i]
9     def toRow(self):
10        return matrix(self.nodes).transpose()
11    def toColumn(self):
12        return matrix(self.nodes)
13
14    class Weights:
15        w = []
16        def __init__(self, n):
17            self.w = n
18        def getRows(self):
19            return len(self.w)
20        def getColumns(self):
21            return len(self.w[0])
22        def toMatrix(self):
23            return matrix(self.w)
24
25    def sigmoid(x):
26        return 1/(1+e^(-x))
27    def relu(x):
28        return max(0,x)
29    def sigmoidPrime(x):
30        return x*(1-x)
31    def reluPrime(x):
32        if x > 0:
33            return 1
34        else:
35            return 0
36    def gradientMultiply(a, b):
37        a_rows = len(a)
38        a_cols = len(a[0])
39        b_rows = len(b)
40        b_cols = len(b[0])
41        if a_rows > 1 or b_rows > 1 or a_cols != b_cols:
42            return None
43        result = []
44        for i in xrange(a_cols):
45            result[0][i] = a[0][i]*b[0][i]
46        return result
47
48    nodes = []
49    weightList = []
50
51    def clearWeights():
52        weightList = []
53
54    def getWeights():
55        w = [weightList[i] for i in xrange(len(weightList))]
56        return w
57
58    def setWeights(w):
59        clearWeights()
60        for i in xrange(len(w)):
61            weightList.append(w[i])
62
63    def feedForward(inputI, output, topology, isSigmoid, w):
64        #input and output both have one row
65        error = [0 for x in xrange(len(output[0]))]
66        nodes.append(Neurons(inputI[0]))
67        hidden = []
68        if w is None: #random weights
69            w = Weights(len(topology)-1)
70            for i in xrange(len(w)):
71                weights = [[random() for j in xrange(len(w))] for k in xrange(len(w))]
72                w[i] = Weights(weights)
73        for i in xrange(len(w)):
74            weightList.append(w[i])
75            weights = w[i].toMatrix()
76            if i == 0:
77                hidden = matrix(inputI)*weights
78            else:
79                hidden = matrix(hidden)*weights
```

```

80         if isSigmoid[i]:
81             for j in xrange(len(hidden[0])):
82                 hidden[0,j] = sigmoid(hidden[0][j])
83         else:
84             for j in xrange(len(hidden[0])):
85                 hidden[0,j] = relu(hidden[0][j])
86         nodes.append(Neurons(hidden[0]))
87         # if the network is trained, then just return the last layer
88     for i in xrange(len(error)):
89         error[i] = (output[0][i]-hidden[0][i])**2
90     return error
91
92 def backProp(error, isSigmoidA):
93     newWeights = []
94     index = len(weightList)-1
95     last = nodes[-1]
96     nodes = nodes[:-1]
97     derivatives = []
98     if(isSigmoid(index)):
99         for i in xrange(len(derivatives[0])):
100             derivatives[0,i] = sigmoidPrime(last.get(i))
101     else:
102         for i in xrange(len(derivatives)):
103             derivatives[0,i] = reluPrime(last.get(i))
104     #print stuff
105     gradients = []
106     for i in xrange(len(error)):
107         gradients[i][0] = error[i]*derivatives[0][i]
108     #getting new weights between output and last hidden layer
109     nextLayer = nodes[-1]
110     gradients2 = matrix(gradients)
111     delta = gradients2*nextLayer.toRow()
112     prev = matrix(weightList[-1]) #not sure lol
113     weightList = weightList[:-1]
114     bago = prev - delta.transpose()
115     newWeights[index] = Weights(bago)
116     index = index - 1
117     #getting new weights between the other layers
118     gradients2 = gradients2.transpose()
119     while not (weightList == []):
120         last = nextLayer
121         nodes = nodes[-1]
122         derivatives = []
123         if isSigmoid[index]:
124             for i in xrange(len(derivatives[0])):
125                 derivatives[0,i] = sigmoidPrime(last.get(i))
126         else:
127             for i in xrange(len(derivatives)):
128                 derivatives[0,i] = reluPrime(last.get(i))
129         gradients2 = gradients2*(prev.transpose())
130         gradients2 = gradientMultiply(gradients2, derivatives)
131         nextLayer = nodes[-1]
132         z = nextLayer.toColumn()
133         prev = weightList[-1].toMatrix()
134         weightList = weightList[:-1]
135         delta = z*gradients2
136         bago = prev - delta
137         newWeights[index] = Weights(bago)
138     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.


Listing 9: Sample Input

```

1 in1 = [[0.2,0.1,0.3,0.5]]
2 out1 = [[1,0,0]]
3 act = [True,True]
4 top = [4,2,3]
5 wait1 = [[0.15,0.14],[0.02,0.24],[0.62,0.2],[0.34,0.25]]
6 wait2 = [[0.22,0.07,0.58],[0.59,0.55,0.77]]

```

```
7 | bigat = [Weights(wait1), Weights(wait2)]  
8 | wth = feedForward(in1, out1, top, act, bigat)  
9 | 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.