Training Feedforward Neural Networks

Research Question: Can a Perceptron Make a Prediction to Produce the Target Output When Provided a Set of Example Data?

Fatima Siddiqui

Abstract

The human brain stands to be an archetype for world's leading design in artificial intelligence. It is a multifaceted processor that is capable of handling simultaneous loads of information within a split second, due to 100 billion neurons interconnected throughout the brain.

Artificial Neural Networks (ANN) are designed to replicate the brain's neural networks to solve problems requiring intelligence such as image or speech recognition. This paper introduces two types of neural networks: the perceptron and multi-layer perceptron (MLP). We examine their network architectures and provide a mathematical framework used by the network to solve problems. For a network to solve a problem, it must learn it by making predictions about the given data. To make the correct prediction, it must minimize the error: the difference between the target and predicted value. We formulated the research question: "can a perceptron make a prediction to produce the target output when provided a set of example data?" We investigate this question through training the network with three problems: 'AND', 'OR', and 'XOR'. We found the perceptron learned the 'AND' and 'OR' problem but failed to learn the 'XOR' problem. We concluded the investigation by determining that the perceptron failed to learn the 'XOR' problem because its architecture is modeled off a linear function; whereas the solution to the 'XOR' problem is based off a nonlinear model. To solve the 'XOR' problem, we used the second neural network: MLP that is effective in finding a solution to a nonlinear problem. Finally, we found through experimentation that the MLP was successful in learning the 'XOR' problem.

Word Count: 261

Table of Contents

1. Introduction

Artificial Neural networks (ANN) are computer algorithms designed to solve problems through a process of repetitive learning. ANN's were invented in 1980's and they have recently had considerable breakthroughs in solving problems such as speech recognition, computer vision, big data analytics, bioinformatics, and natural language processing (Artificial Neural Networks Technology, n.d.). The advantages of an ANN are its flexibility to learn, generalize, and adapt to situations based on its surroundings; making it applicable in all areas of research.

Two classes of ANNs we will be discussing in this paper is the "perceptron" and the "multilayer perceptron" (MLP). In Section 2, we introduce the architecture of the perceptron. We show how the network makes a prediction utilizing an 'activation' function that consists of a mapping between the inputs and the network's prediction. Section 3 discusses the mathematical foundation behind the network's process of learning. When a network makes a prediction, we compute an associated 'error' which is the difference between the prediction and the desired output. For the network to learn the problem, the error from each prediction must be minimized. This is done by utilizing one learning algorithm called "backpropagation". We use backpropagation in Section 4 to train the perceptron to learn three problems: 'AND', 'OR', and 'XOR'. We can then state our research question as: "can a perceptron adjust its prediction to produce the desired output for the three problems: 'AND', 'OR', and 'XOR'?" In Section 5, we breakdown the architecture of a second neural network: the multi-layer perceptron (MLP). We then train the MLP in Section 5.2 to learn the XOR problem.

2. Perceptron Architecture

One type of neural network is called the perceptron. The network contains a set of input connections, i_t, connected to a single node in a hidden layer, s. Each connection between the input and the hidden layer is referred to as a synapse and contains a corresponding weight value W_i. The receiving node (neuron) can process a signal to the output through an activation function σ (Cilimkovic, 2010). A prediction, $\hat{y}$, is then computed using the output of the activation function (see Fig. 1).

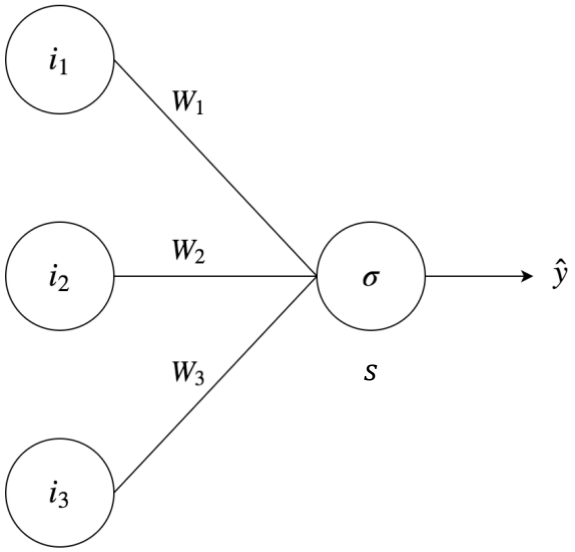

Figure 1: The perceptron architecture with $n = 3$ input connections. i_n is the input and W_n is the weight value for a corresponding input connection n, σ is the activation function, and $\hat{y}$ is the prediction.

There are a variety of activation functions to choose from that can affect the network's prediction and performance. Two activation functions this paper will focus on will be the linear and sigmoid activation function, seen by equations (2.1) and (2.2) respectively[1] (Sharma, 2016),

$$\sigma(i, W) = i \bullet W \tag{2.1}$$

$$\sigma(i, W) = \frac{1}{1 + e^{-i \bullet W}} \tag{2.2}$$

where $\bullet$ represents the vector dot product. Figure 2 plots the activation function from equation (2.2).

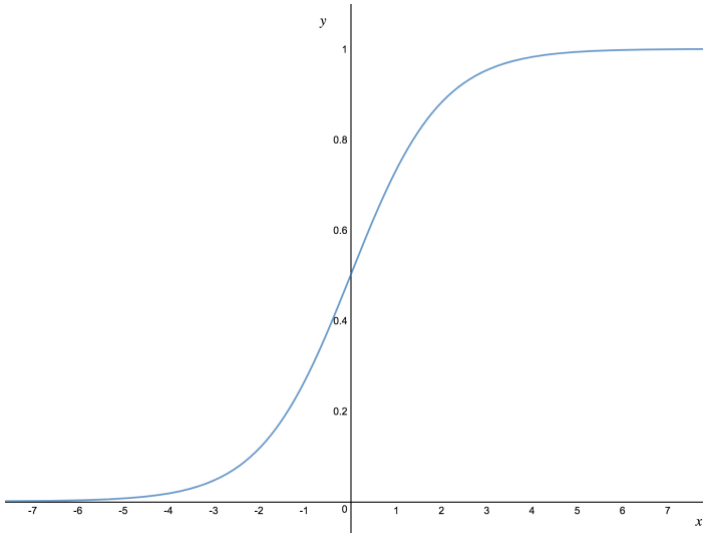

Figure 2: Plot of $y = \frac{1}{1+e^{-x}}$ over $x \in [-8,8]$.

The range of the sigmoid activation function from Figure 2 is between 0 and 1. When the result of the vector dot product of our input values and weights is a large number, such as 5, the activation function will send a corresponding signal of 1 to the output. When the vector dot product of our input values and weights is a very small number, such as -5, the synapse is considered to have not 'fired' and the signal will send forward a value of 0. The sigmoid activation function is often used for problems involving a binary classifier of two categories (1 as 'Yes', 0 as 'No') (Sharma, 2016).

3. Backpropagation and Gradient Descent

When the network makes a prediction, we can compute a corresponding error that measures the difference between our network's prediction, $\hat{y}$, and our target output, y, given by the following cost function (Gibiansky, 2012):

$$E(y, \hat{y}) = \left(\frac{1}{2}\right)(y - \hat{y})^2 \tag{3.1}$$

[1] $i \bullet W$ can be further simplified to $i \bullet W = \sum_n i_n * W_n$ for all input connections n.

For the network to 'learn' the problem, the cost function, $E(y, \hat{y})$, from equation (3.1) must be locally minimized for each set of inputs in our training data. This is done by finding an appropriate set of weights on our input neurons, W, that successfully predicts our target output, y. The learning algorithm to minimize the cost, $E(y, \hat{y})$, can be divided into two parts: error propagation (backpropagation), and weight updates (gradient descent) (Roja, 1996).

3.1. Backpropagation

Error backpropagation is a method to calculate the error contributions of each input neuron after a prediction, $\hat{y}$, is made over one sample of test data. When an input is presented to our network, it is propagated forward through a single hidden layer to reach an output prediction, $\hat{y}$. The prediction is then compared with our desired output to calculate an error, $E(y, \hat{y})$, given by the cost function in equation (3.1). The error value is then backwards propagated through the network (from the output back to the input) until each input neuron has a corresponding error value that reflects its contributions to the original output.

Backpropagation is done by taking the derivative of our cost function, $E(y, \hat{y})$, in equation (3.1) with respect to each input weight, W_i, given by (Nielsen, 2017):

$$\frac{\partial E(y, \hat{y})}{\partial W} = -(y - \hat{y}) \frac{\partial \hat{y}}{\partial W} \tag{3.2}$$

The factor of $(\frac{1}{2})$ in equation (3.1) conveniently cancels the exponent once the error function is differentiated in equation (3.2). Since our prediction, $\hat{y}$, is computed using the hidden layer, s, we can apply the chain rule of calculus to simplify $\frac{\partial \hat{y}}{\partial W}$ in the right side of equation (3.2) to[2]:

$$\frac{\partial E(y, \hat{y})}{\partial W} = -(y - \hat{y}) \frac{\partial \hat{y}}{\partial s} \frac{\partial s}{\partial W} \tag{3.3}$$

Note that the output from the hidden layer, s, is just the activation function σ applied to the neuron's input, i. $\frac{\partial \hat{y}}{\partial s}$ can be then represented as the first derivative of our activation function, σ, at the input value i, given by the derivations in equation (3.4) to (3.6).

$$\hat{y} = s(i) \tag{3.4}$$

$$\frac{\partial \hat{y}}{\partial s} = s'(i) \tag{3.5}$$

$$\frac{\partial \hat{y}}{\partial s} = \sigma'(i) \tag{3.6}$$

[2] Since $\hat{y} = s(i, W)$ applying the chain rule to $\frac{\partial \hat{y}}{\partial W}$ we get $\frac{\partial \hat{y}}{\partial W} = \frac{\partial \hat{y}}{\partial s} \frac{\partial s}{\partial W}$.

When considering the j^{th} input and weight (i_j, W_j) in our connection to the hidden layer, s, only one term in $\frac{\partial s}{\partial W}$ is relevant. $\frac{\partial s}{\partial W}$ can be then simplified to:

$$\frac{\partial s}{\partial W_j} = \frac{\partial}{\partial W}\left(\sum_{k=1}^{n} i_k * W_k\right) = \frac{\partial}{\partial W_j}\left(i_j * W_j\right) = i_j \tag{3.7}$$

Finally, putting equations (3.3), (3.6), and (3.7) together, we get the contribution of error with respect to the j^{th} input connection:

$$\frac{\partial E(y, \hat{y})}{\partial W_j} = -(y - \hat{y})\sigma'(i) * i_j \tag{3.8}$$

Having an equation to represent the error contributions from each input connection (and its respective weight), we can update each of our weight values W_j to minimize the error from equation (2.3) using the gradient descent algorithm.

3.2. Gradient Descent

To minimize our cost function, $E(y, \hat{y})$, we update each weight value by subtracting a percentage of the weight's gradient from equation (3.8). This can be seen by:

$$W_j^{new} = W_j^{old} - \eta * \frac{\partial E(y, \hat{y})}{\partial W_j} \tag{3.9}$$

The proportion of the weight's gradient we subtract, η, is defined as the 'learning rate'. Having a larger learning rate makes the neural network learn more efficiently, but can also miss the local minimum and place us at a higher cost in our cost function (failing to learn). Having a slower learning rate makes the neural network train slower and can also trap us in an undesirable local minimum that fails to learn the problem. The appropriate learning rate to set for our network is relative and problem-specific, depending entirely on the curvature of our cost function (Nielsen, 2017).

Figure 3 demonstrates this by plotting the cost function for a perceptron with a single input connection and hidden layer. Since our neural network is initialized with a random set of weights, our initial cost can appear in any point on this function (one example is shown by the red point). Using the gradient descent algorithm, we can iterate over training data and descend towards the global minimum indicated by the red arrow. Figure 4 demonstrates the same problem using a perceptron with two input connections (and two weights). Having n inputs makes our cost function that we are looking to minimize exist in $\mathbb{R}^n$. We notice that having different initialized weight values will send us in different directions when applying the gradient descent algorithm. Gradient descent with backpropagation does not guarantee finding a global minimum in our cost function, and achieving this target is heavily dependent on the weight values initialized and the learning rate set.

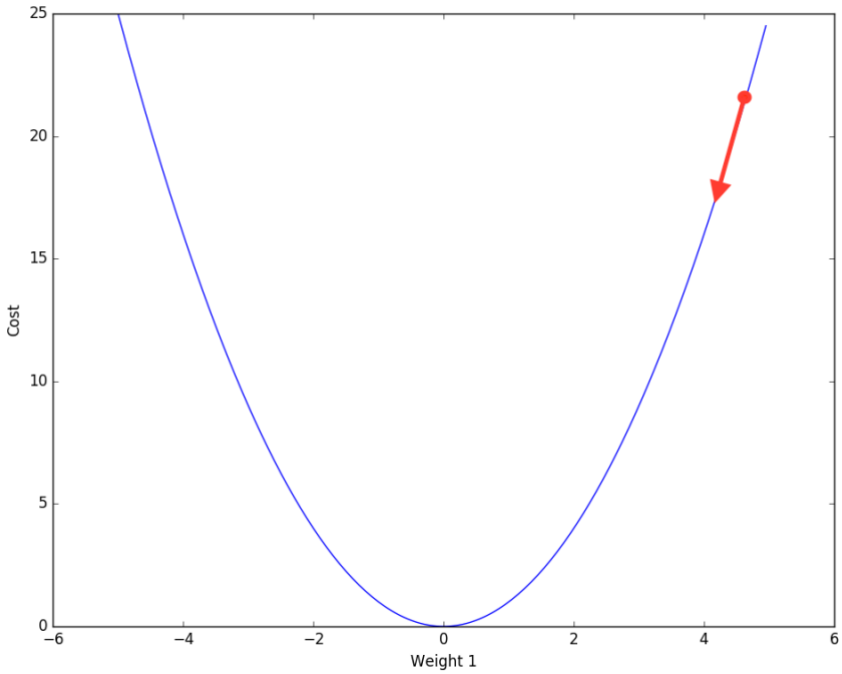

Figure 3: The cost function for a neural network containing a single input connection and hidden layer. The red point represents the error for an initialed weight value, and red arrow represents the direction to descend that will minimize the cost.

The following pseudocode demonstrates the gradient descent algorithm for training a perceptron with three input connections and one hidden layer:

1. Initialize network weights as random values and set the learning rate: (W, η)
2. For each training example in the training data:
    a. Compute a prediction, $\hat{y}$ (forward propagation)
    b. Compute an error, $E(y, \hat{y})$
    c. Compute the gradient using the error from our prediction, $\frac{\partial E(y,\hat{y})}{\partial W}$
    d. Update the network's weight values, W^{nnew}

Applying this algorithm over the training data will adjust our weight values and locally minimize our cost function.

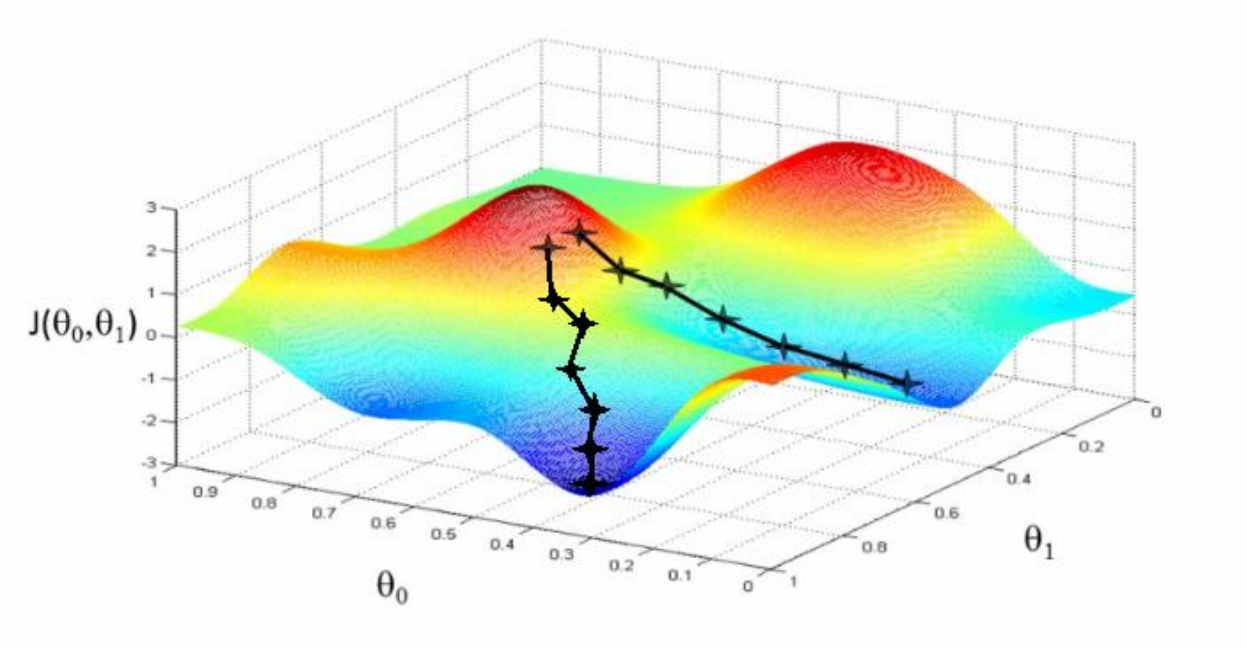

Figure 4: The cost function for a neural network containing two input connections and one hidden layer. The initial weight and learning rate influence which local minimum we may end up in. $J(\theta_1, \theta_2)$ is our cost and θ_1, θ_2 are our two weights in the network (https://i.stack.imgur.com/6j2gg.png, n.d.).

4. AND, OR, and XOR Problem

To investigate the research question, we train a perceptron to learn the 'AND' problem. The input and target output of the 'AND' problem can be seen by the following table:

Inputs	Target Output
(1, 1) (True and True)	1 (True)
(1, 0) (True and False)	0 (False)
(0, 1) (False and True)	0 (False)
(0, 0) (False and False)	0 (False)

Table 1: Inputs and Output of the 'AND' problem

where input/output values of '1' represents 'True', and '0' represents 'False'. When both of the inputs are true, the target output is true. If either of the inputs are 0 then the target output is false. This is representative of the 'AND' function, where both inputs must be true (true *and* true) for the output to be true.

We are looking to train a perceptron to learn the 'AND' function specified by the inputs and outputs from Table 1. Figure 4 illustrates the perceptron with two inputs and one output.

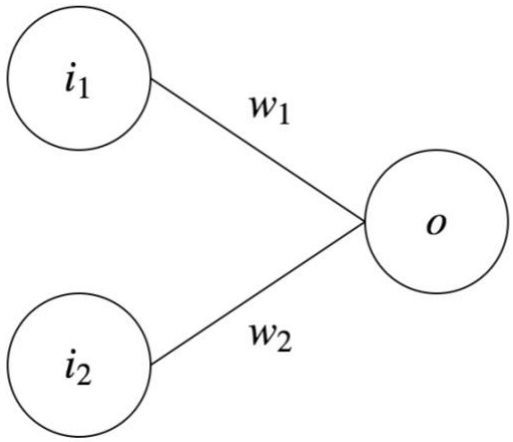

Figure 4: The relationship of the weights to the inputs and the output values

We begin by initializing two random weights of our network, where $(w_1, w_2) = (1.3, 0.1)$. Using a linear activation function from Equation 2.1, the network prediction when processing an input of $(i_1, i_2) = (1,0)$ is then:

$$o = i_1 * w_1 + i_2 * w_2$$

$$o = 1 * 1.3 + 0 * 0.1 = 1.3$$

which can also be seen by the following figure:

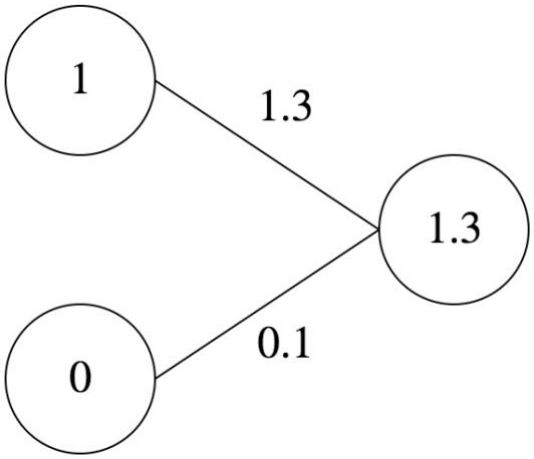

Figure 5: The relationship of the weights to the inputs and the output values where numerical values for each variable are given

The network made an incorrect prediction given the initialized set of random weights. The error of our prediction was -1.3 ($target - prediction = 0 - 1.3$). To adjust this, we must apply the gradient descent algorithm described in Section 3.2 to find a set of weights that can learn the 'AND' problem for each data in our input set. We apply the gradient descent algorithm over 600 iterations on the input data. The following graph summarizes the results of our training session:

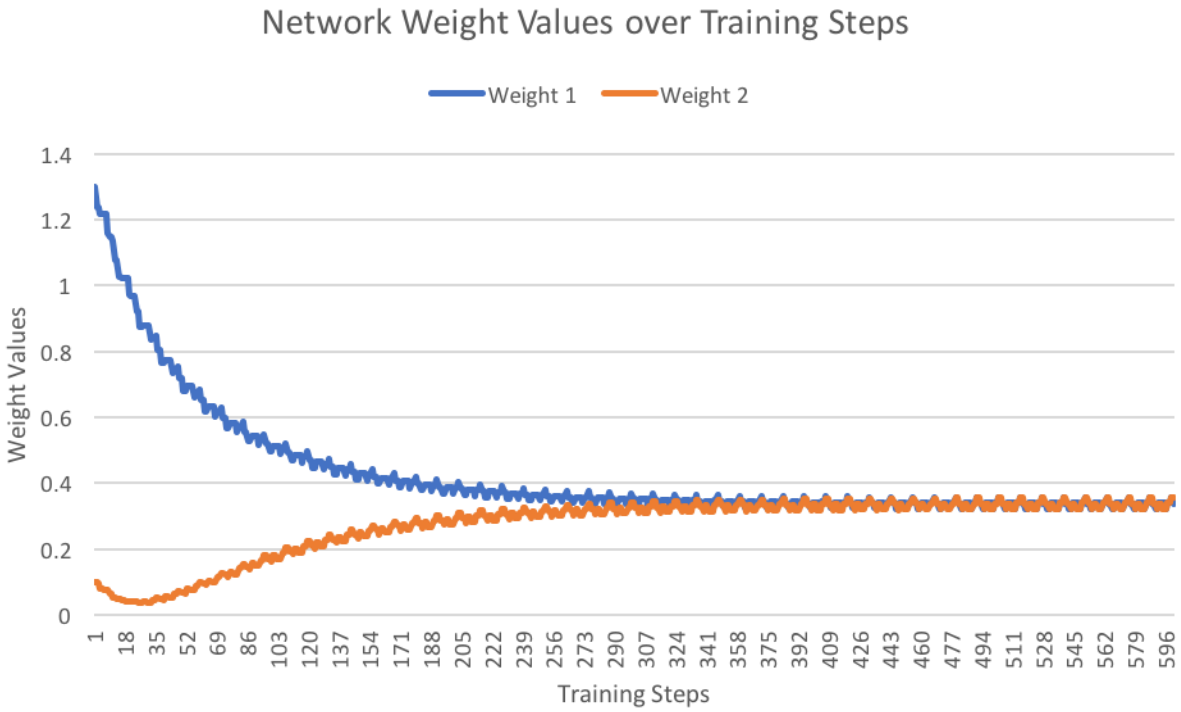

Figure 5: Both the weight values for the respective inputs are graphed for every iteration in the 'AND' problem

Having an initialized weight values of $(w_1, w_2) = (1.3, 0.1)$, both weights converged to the values of $(w_1, w_2) = (0.33, 0.33)$. With these converged values, the network now makes a correct prediction over each input set and successfully learns the training data. The following table demonstrates the results of each prediction using the new set of weights:

Input 1	**Input 2**	**Target Output**	**Network Prediction** $o = round(0.33 * i_1 + 0.33 * i_2)$
1	1	1	1
1	0	0	0
0	1	0	0
0	0	0	0

Table 3: Network's prediction using the converged weight values. The network successfully predicted the target output data for the 'AND' problem.

We conduct a similar experiment and train the network to learn the 'OR' problem. The input and target output of the 'OR' problem can be seen by the following table:

Inputs	Target Output
(1, 1) (True and True)	1 (True)
(1, 0) (True and False)	1 (True)
(0, 1) (False and True)	1 (True)
(0, 0) (False and False)	0 (False)

Table 2: Inputs and Output of the 'OR' problem

where input/output values of '1' represents 'True', and '0' represents 'False'. When both of the inputs are true, the target output is true. If either of the inputs are true then the target output is also true. This is representative of the 'OR' function, where at least one of the inputs must be true for the output to be true.

We apply the gradient descent algorithm over 600 iterations on the input data. The following graph summarizes the results of our training session:

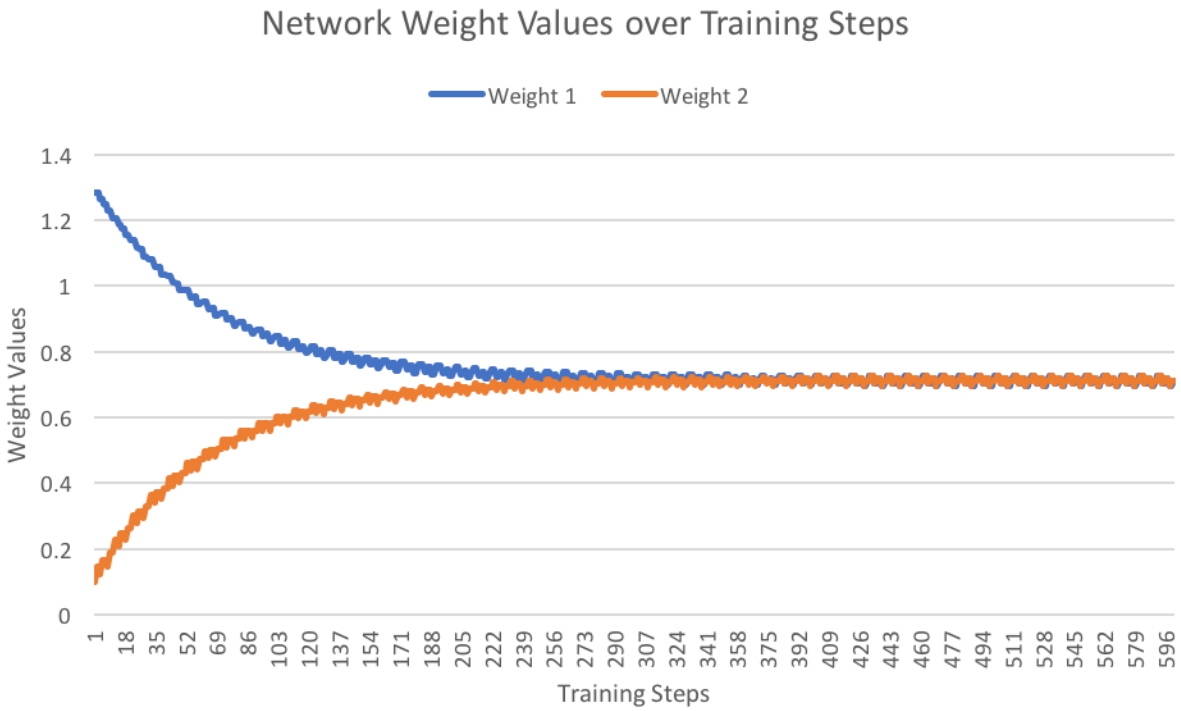

Figure 5: Both the weight values for the respective inputs are graphed for every iteration in the 'OR' problem

Both weights converged to a value of 0.66. With these converged values, the network now makes a correct prediction over each input set and successfully learns the training data. The following table demonstrates the results of each prediction using the new set of weights:

Input 1	Input 2	Target Output	Network Prediction $o = round(0.66 * i_1 + 0.66 * i_2)$
1	1	1	1
1	0	1	1
0	1	1	1
0	0	0	0

Table 4: Network's prediction using the converged weight values. The network successfully predicted the target output data for the 'OR' problem.

The following graph illustrates the cost function for the training session of the 'OR' problem:

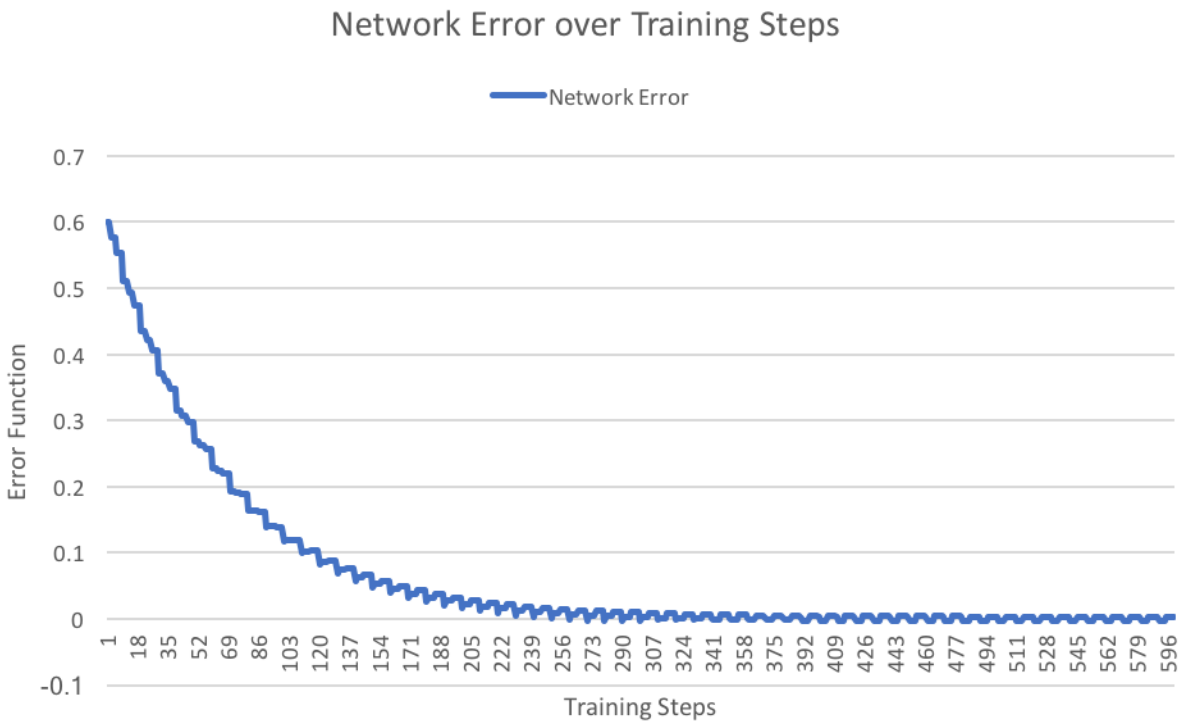

Figure 6: Network's error over 600 iterations on input data.

Our network globally minimizes the error after 400 iterations on the input data. This is also reflected on Figure 5 where the weights converged after 400 iterations.

We conduct a similar experiment and train the network to learn the 'XOR' problem. The input and target output of the 'XOR' problem can be seen by the table on the following page.

Inputs	Target Output
(1, 1) (True and True)	0 (True)
(1, 0) (True and False)	1 (True)
(0, 1) (False and True)	1 (True)
(0, 0) (False and False)	0 (False)

Table 3: Inputs and Output of the 'XOR' problem

where input/output values of '1' represents 'True', and '0' represents 'False'. When both of the inputs are true, the target output is false. If only one of the inputs are true then the target output is true. This is representative of the exclusive OR (XOR) function, where exactly one of the inputs must be true for the output to be true.

We apply the gradient descent algorithm over 600 iterations on the input data. The following graph summarizes the results of our training session:

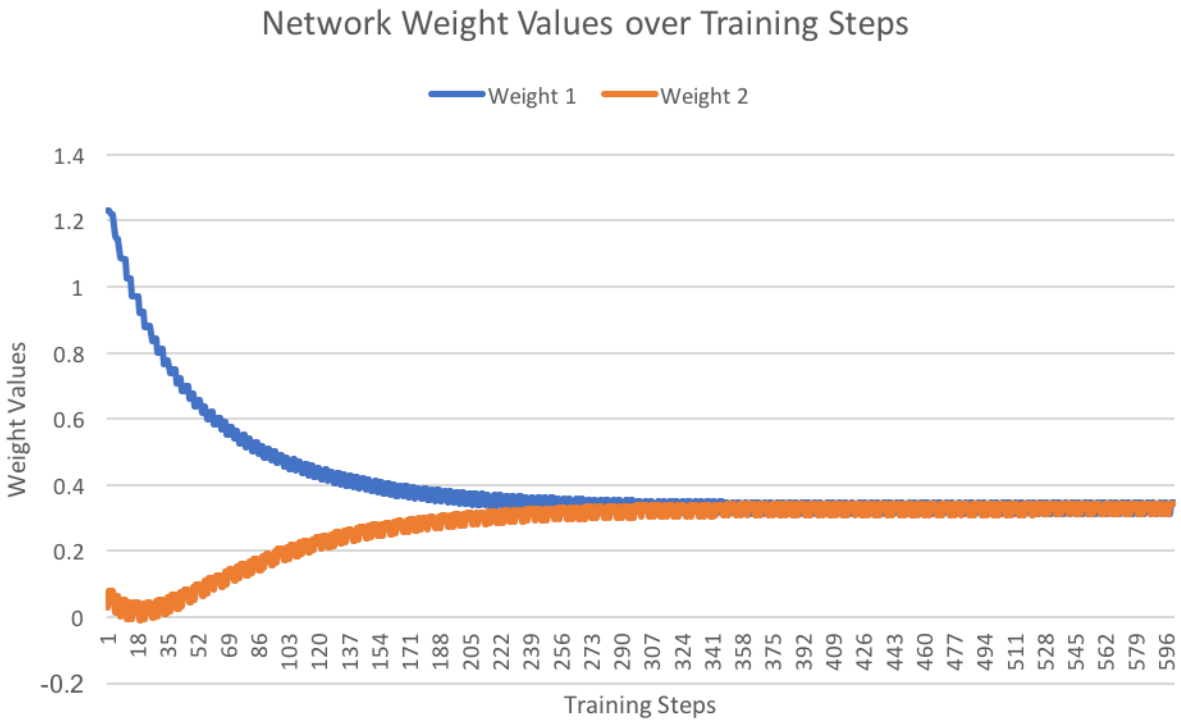

Figure 6: Both the weight values for the respective inputs are graphed for every iteration in the 'XOR' problem

Both weights converged to a value of 0.33. Although the network weight values converged, the network failed to make correct predictions over the input data. The table on the following page demonstrates the results of each prediction using the new set of weights:

Input 1	Input 2	Target Output	Network Prediction $o = round(0.66 * i_1 + 0.66 * i_2)$
1	1	0	1
1	0	1	1
0	1	1	1
0	0	0	0

Table 4: Network's prediction using the converged weight values. The network failed to predict the target output data for the 'XOR' problem.

This can also be seen by plotting the cost function over each iteration:

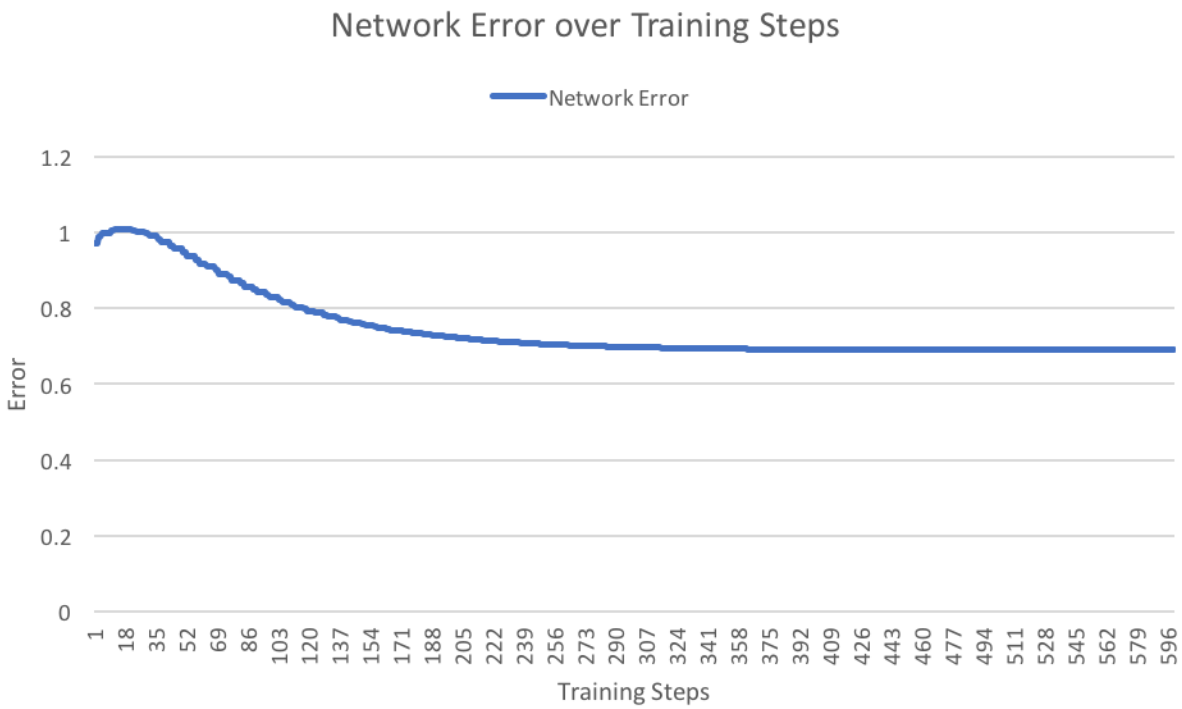

Figure 7: Network's error over 600 iterations on input data.

From figure 7, we can see the cost function was not minimized over 600 iterations on our input data.

We say the perceptron successfully learned the 'AND' and 'OR' problem but had failed to learn the 'XOR' problem. Going back to our research question, we have found that the perceptron is able to automatically adjust its predictions to output the target results for the AND' and 'OR' problem but not the 'XOR' problem. We are interested to understand why the network succeeded in learning the 'AND' and 'OR' problem but not the exclusive OR (XOR) problem.

The domain of our input space (i_1, i_2) can be represented by the following points on the plot:

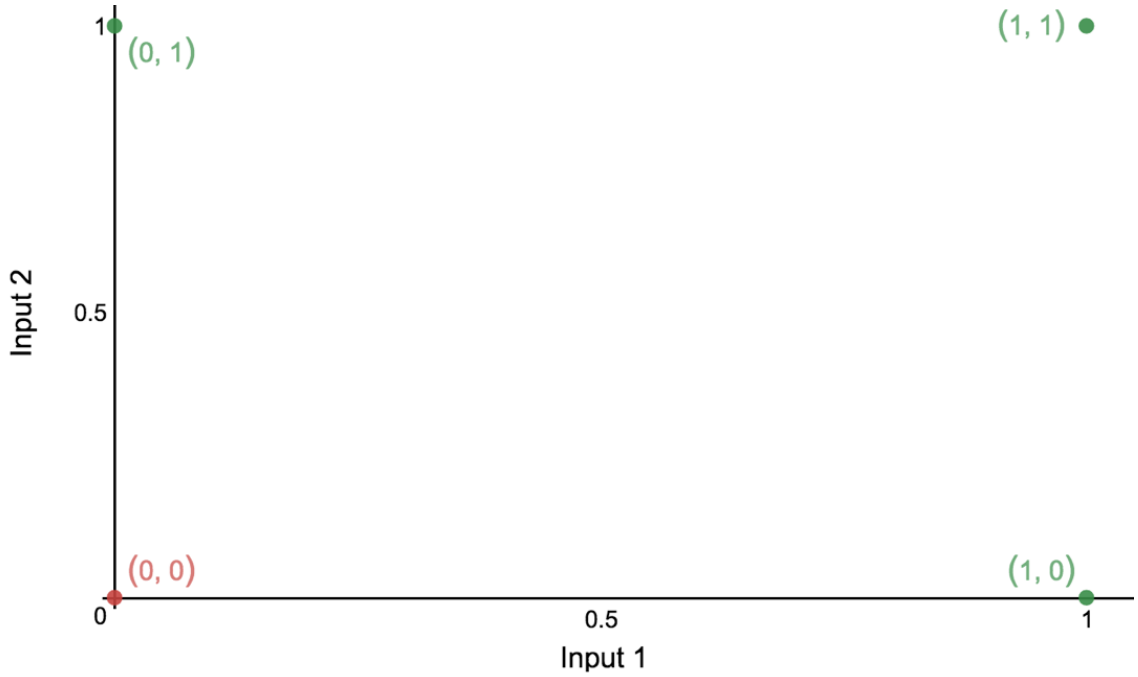

Figure 8: Graphical representation of the 'OR' input space. This plot represents the 'OR' problem where green is True {(1,1),(1,0),(0,1)}, and red is False {(0,0)}

The network's prediction is given by the following equation:

$$o = w_1 * i_1 + w_2 * i_2 \tag{4.1}$$

Equation 4.1 can be seen as a hyperplane in $\mathbb{R}^3$ where (w_1, w_2) are the slopes of the hyperplane. For the network to successfully learn the 'AND' or 'OR' problem, we are looking for an appropriate set of weights/slope values that creates a function that can linearly separate the true and false values.

Figure 9 shows an example of this where the function $y = \frac{1}{2} - x$ linearly separates the true values from the false values (green vs red) for the 'OR' problem. All the points above y are the true values and anything below y are the false values. Figure 10 shows another example where the function $y = \frac{4}{3} - x$ linearly separates the true and false values (green vs red) for the 'AND' problem.

The input space for the XOR problem is provided in Figure 11. The problem can then be restated as: 'does there exist a linear function that can separate the true values from the false values in the input space?'

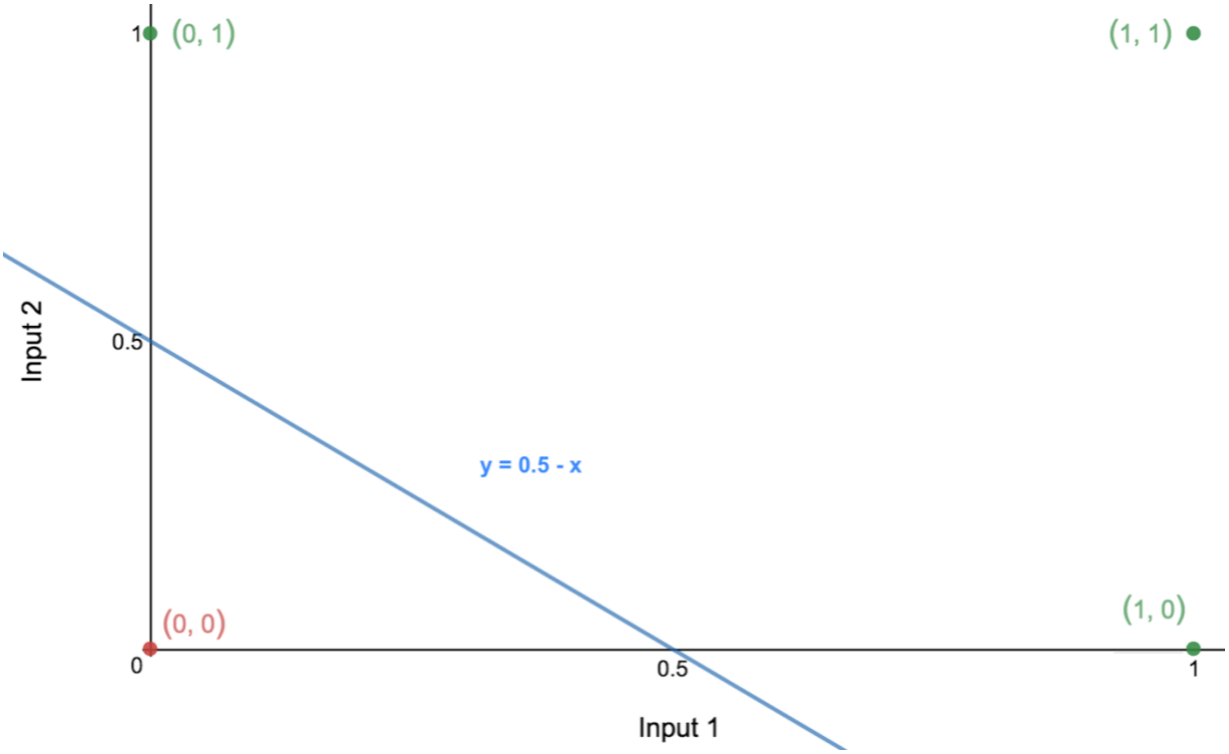

Figure 9: Example of a perceptron's trained prediction function that linearly separates the true and false values for the 'OR' problem.

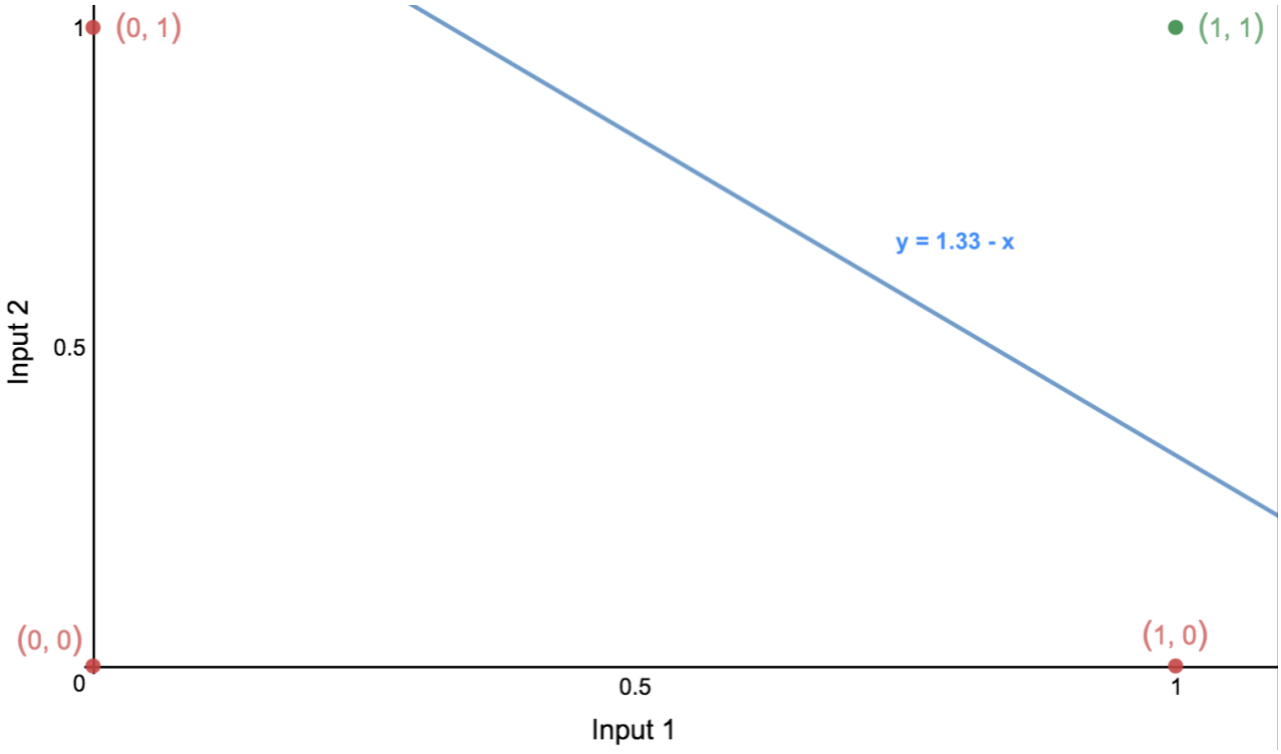

Figure 10: Example of a perceptron's trained prediction function that linearly separates the true and false values for the 'AND' problem.

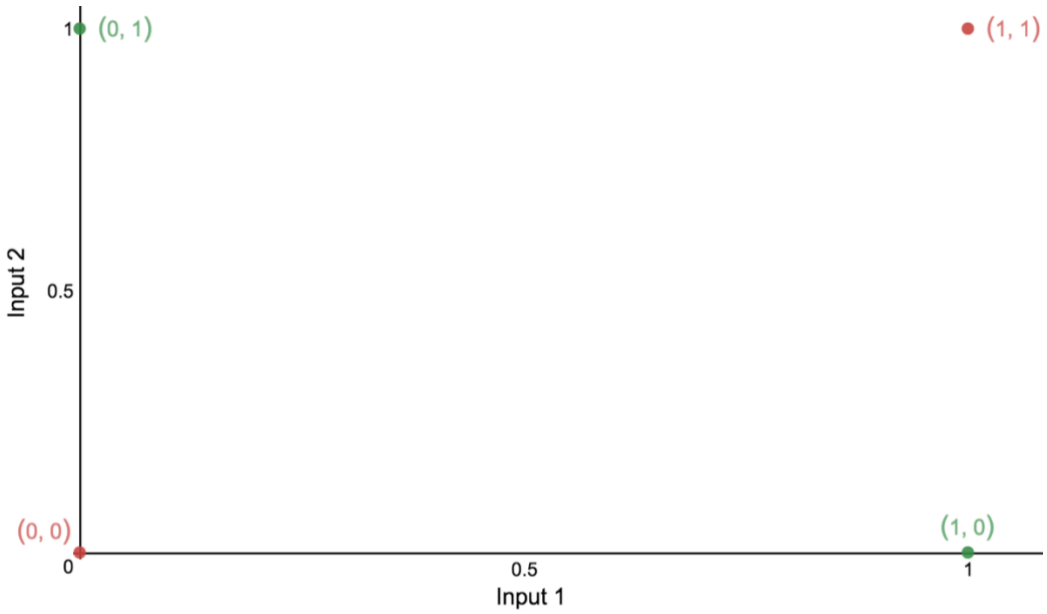

Figure 11: Graphical representation of the 'exclusive OR' input space. This plot represents the 'XOR' problem where green is True $\{(1,0),(0,1)\}$, and red is False $\{(0,0),(1,1)\}$.

Based on inspection of Figure 11, there exists no linear function that can separate the true and false values for the 'XOR' problem. The only way to separate the points is through a non-linear function such as an ellipse. Figure 12 provides an example of a solution to the 'Exclusive OR' function where an ellipse is used to successfully separate the true and false points.

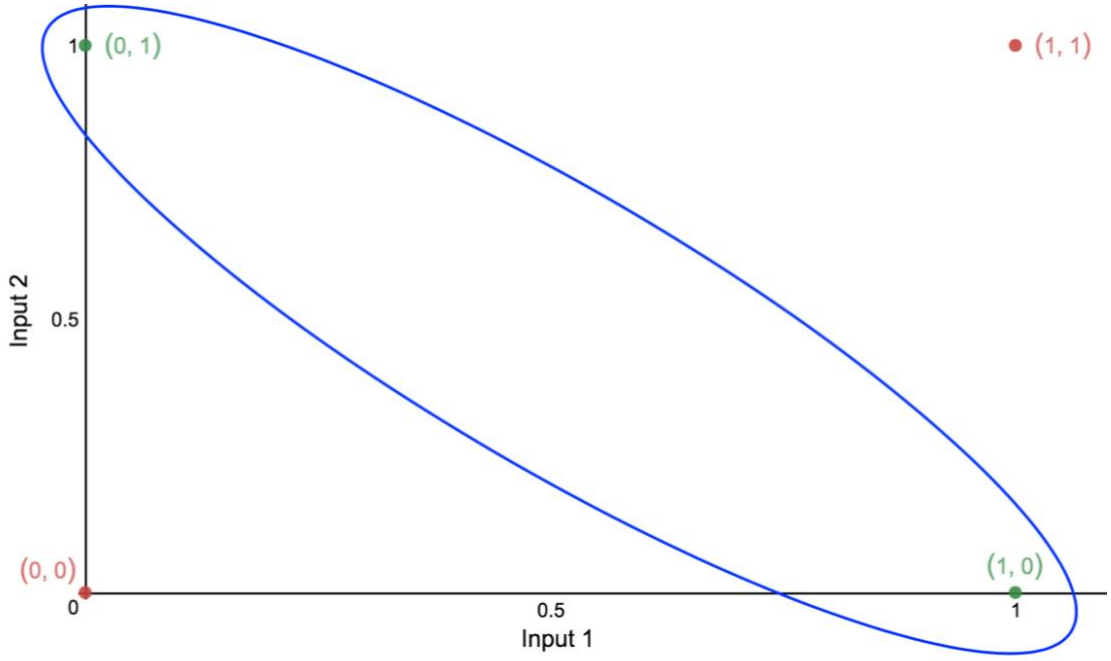

Figure 12: A non-linear solution to the 'Exclusive OR' problem. An ellipse is used to separate the true $\{(1,0),(0,1)\}$ and false $\{(0,0),(1,1)\}$ values.

As a result, the perceptron's prediction function (see Equation 4.1) cannot linearly separate the true/false points of the 'XOR' function because the solution is non-linear. Graphically speaking, this is not possible because you cannot draw a straight line to separate the points $(0,0), (1,1)$ from the points $(0,1), (1,0)$.

Coming back to the research question, we can see that the perceptron fails to automatically predict the target output for the 'XOR' problem because the neural network is made of 1 layer so it only succeeds with linearly separable data (Neural Learning). One approach to constructing a non-linear function using a neural network is to introduce additional nodes into the hidden layer of our perceptron (Burkill, 2016).

5. Multi-Layer Perceptron (MLP)

5.1. MLP Architecture

A multi-layer perceptron contains a set of input connections, i_t, connected to a set of nodes in a hidden layer s (See Fig. 13). Note that an MLP can consist of more than one hidden layer. For the purpose of this paper, we limit the MLP architecture to a single hidden layer containing 3 nodes.

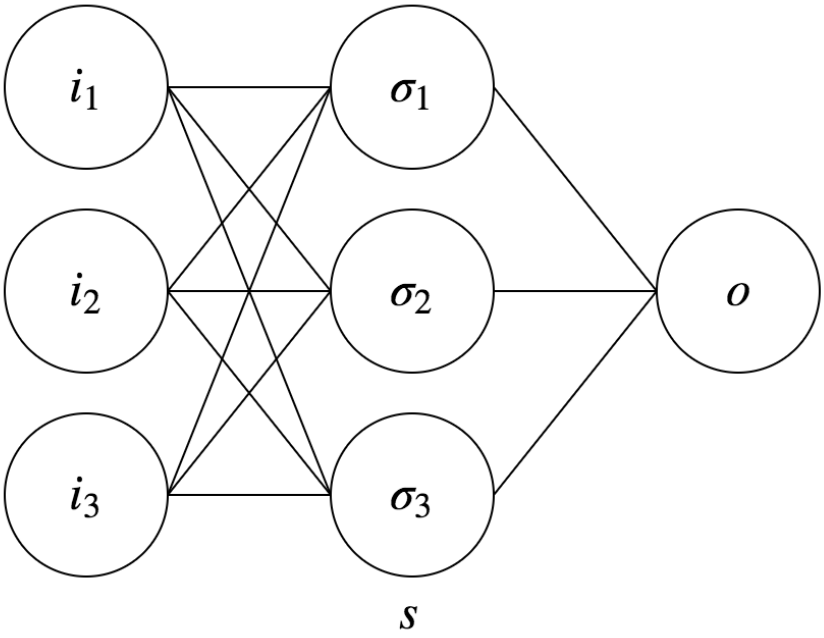

Figure 13: The multi-layer perceptron architecture with $n = 3$ input connections and $n = 3$ nodes in the hidden layer.

Each connection between an input and a node in the hidden layer contains a corresponding weight value, w. An MLP with $n = 3$ nodes in a hidden layer corresponds to a total of 9 connection weights. The formula for the three receiving neurons (nodes) in the hidden layer can be expressed by the following equations (C, 2014):

$$s_1 = i_1 w_{11} + i_2 w_{21} + i_3 w_{31} \quad (5.1)$$
$$s_2 = i_1 w_{21} + i_2 w_{22} + i_3 w_{32} \quad (5.2)$$
$$s_3 = i_1 w_{31} + i_2 w_{32} + i_3 w_{33} \quad (5.3)$$

where w_{ij} corresponds to the i_{th} input connection connecting to the j_{th} node in the hidden layer. Using matrix algebra, the input connections can be seen as the vector:

$$I = \begin{bmatrix} i_1 \\ i_2 \\ i_3 \end{bmatrix} \quad (5.4)$$

and the weights connected to the hidden layer can be seen by the matrix:

$$W = \begin{bmatrix} w_{11} & w_{21} & w_{31} \\ w_{21} & w_{22} & w_{32} \\ w_{31} & w_{32} & w_{33} \end{bmatrix} \quad (5.5)$$

The values in the hidden layer, prior to the activation function, becomes the matrix multiplication of our input vector, I, with the associated weight matrix, W:

$$S = I \odot W \quad (5.6)$$

where $\odot$ refers to matrix multiplication. Equation 5.6 is the matrix representation of equations 5.1-5.3. The neurons then go through an activation function where the values for the nodes in the hidden layer are:

$$\sigma_1 = \sigma(i_1 w_{11} + i_2 w_{21} + i_3 w_{31}) \quad (5.7)$$
$$\sigma_2 = \sigma(i_1 w_{21} + i_2 w_{22} + i_3 w_{32}) \quad (5.8)$$
$$\sigma_3 = \sigma(i_1 w_{31} + i_2 w_{32} + i_3 w_{33}) \quad (5.9)$$

where $\sigma(x)$ is the linear or sigmoid activation function discussed in Section 2. The matrix representation of equations 5.7-5.9 can be seen by the following equation:

$$\sigma = \sigma(S) \quad (5.10)$$

where the activation function is applied to each element in the matrix of S.

The output is then computed by another set of weights connecting our hidden layer, S, to the output layer, o:

$$o = \sigma_1 w_{11}^2 + \sigma_2 w_{21}^2 + \sigma_3 w_{31}^2 \quad (5.11)$$

where w_{ij}^k corresponds to the weight value of the k_{th} layer connecting the i_{th} input connection to the j_{th} receiving node. In this case $j = 1$ as there is only one receiving output node. The matrix representation of our output can be seen by:

$$o = \sigma \odot W^2 \quad (5.12)$$

where $\odot$ refers to matrix multiplication. It is interesting to see the dimension of the vector σ is $[1 \times 3]$ and the dimension of the weight vector W^2 is $[3 \times 1]$. Multiplying these two vectors results in a vector with dimensions $[1 \times 1]$ which is a scalar value.

The final prediction, $\hat{y}$, uses an activation function on the output node:

$$\hat{y} = \sigma(o) \tag{5.13}$$

To make a successful prediction, $\hat{y}$, to learn the problem, we are looking for a set of weight values, $\{W, W^2\}$, that minimize the cost function discussed in Section 3.

5.2. Solving the XOR Problem

We construct an MLP with $n = 3$ nodes in one hidden layer to solve the 'exclusive OR' problem (XOR) from Section 4. Our architecture is similar to Figure 13; however, we have $n = 2$ input connections as opposed to 3 (values of 1 or 0).

The following table demonstrates the results of each prediction using the multi-layer perceptron:

Input 1	**Input 2**	**Target Output**	**Network Prediction**
1	1	0	0
1	0	1	1
0	1	1	1
0	0	0	0

Table 5: Network's prediction using the multi-layer perceptron. The network succeeded to predict the target output data for the 'XOR' problem.

The following graph illustrates the cost function for the training session of the 'XOR' problem using the MLP:

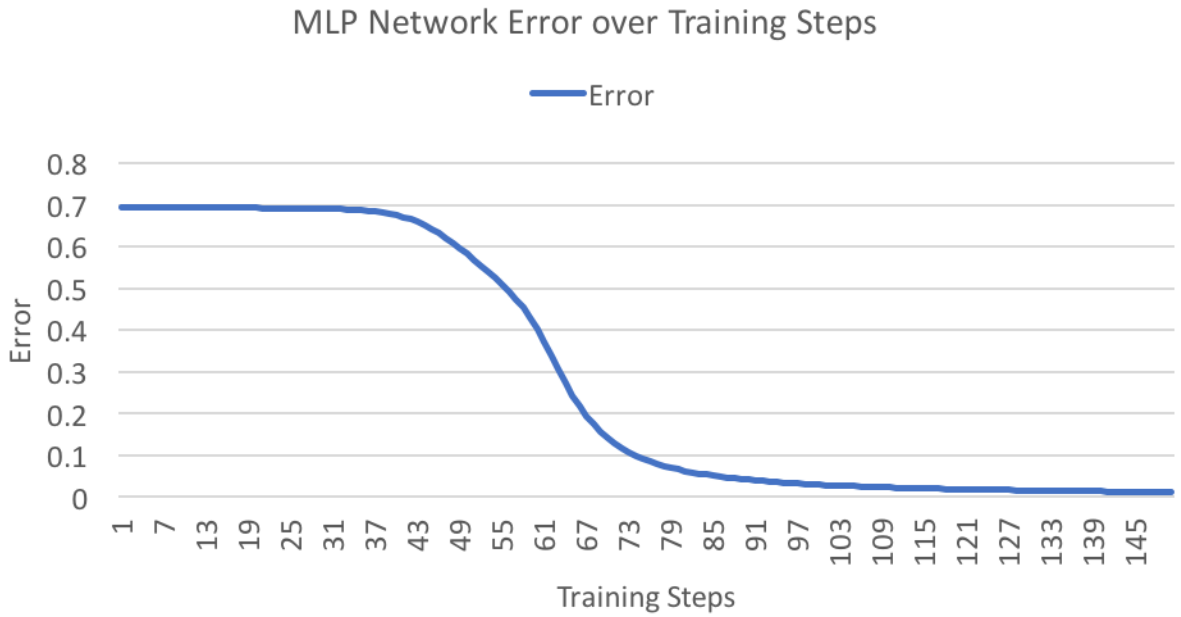

Figure 14: Network's error over 150 iterations on input data.

From Figure 14, we see the network minimized the error after 150 iterations. Unlike the perceptron, the MLP was able to successfully learn the 'XOR' problem. This is primarily due to the fact that the MLP is a non-linear classifier (Burkill, 2016).

6. Conclusion

From the investigation, we were able to answer our research question: can a perceptron make a prediction to produce the target output when provided a set of example data? We concluded that the perceptron is able to predict the desired output when it is trained with a given set of inputs for the 'AND' and 'OR' problems. However, the perceptron failed to answer the 'XOR' problem. This lead us to determine that the perceptron can only produce the correct output for data that is linearly separable. This is seen by Figures 9 and 10 where the target outputs for the 'AND' and 'OR' problems can be graphically represented by a line. Whilst in Figure 11, we see that the target prediction for the 'XOR' problem is graphically showcased as a non-linear function modeled close to an ellipse. To solve the 'XOR' problem, we used a MLP as it is able to find the answer for data that is nonlinearly separable. We were able to successfully solve the 'XOR' problem and determine the target outputs.

We can conclude that neural networks should be trained according to their number of layers. Since a perceptron consists of one layer, it could be trained with data that is linearly separable such as the 'AND' and 'OR' problems investigated in this paper. Similarly, an Multilayer Perceptron (MLP) can be trained to learn data that is non-linearly separable. This could encompass a wide variety of problems such as the 'XOR' problem discussed in this paper or even more predictive data such as determining temperatures with empirical weather data. In fact, multilayered perceptron is used in different areas within science. For example, the network was used in brain cancer research to classify tumour array expression data (Fisher, 2007). I

would like to continue learning about the various application of MLPs in sciences and other subjects, to learn the diversity in the methods which people used to classify and model data using MLP structures.

As our ability to expand the number of layers within a neural network increases, we are able to train the network to learn more complex sets of data. This includes facial, language and voice recognition, or even learning a video game. Soon, the network can evolve to perform tasks as efficiently and eventually, better than the human brain. If so, the Artificial Intelligence can surpass humanity in general intelligence becoming the world's most intelligent being. However, their sudden increase in intelligence will be impossible for humans to match making it difficult for us to control their actions and thoughts in any coherent manner. Many influential figures such as Alan Turing, Stephen Hawking, Elon Musk and Bill Gates have expressed concerns about the risks of growing intelligence of artificial neural networks (ANN) (Edaicicco, 2015). The magazine *Nature* citied: "Machines and robots that outperform humans across the board could self-improve beyond our control — and their interests might not align with ours." (Nature, 2016) As we move forward to grow the capabilities of ANNs, we must keep in mind the change such innovation can bring.

7. References

Artificial Neural Networks Technology. (n.d.). Retrieved December 29, 2017, from http://www.psych.utoronto.ca/users/reingold/courses/ai/cache/neural2.html

Burkill, J. (2016, July 9). *Artificial Neural Networks – Part 1: The XOr Problem* . Retrieved December 29, 2017, from Machine Learning and Optimisation: http://www.mlopt.com/?p=160

C, S. (2014, Novermber 7). Neural Networks Demystified [Part 2: Forward Propagation]. Welch .

Cilimkovic, M. (2010). *Neural Networks and Back Propagation Algorithm.* Institute of Technology Blanchardstown, Dublin.

Edaicicco, L. (2015, January 28). *Bill Gates: Elon Musk Is Right, We Should All Be Scared Of Artificial Intelligence Wiping Out Humanity*. Retrieved December 31, 2017, from Business Insider: http://www.businessinsider.com/bill-gates-artificial-intelligence-2015-1

Fisher, A. F. (2007). *Outcome Prediction in Cancer.* Elsevier Science Limited.

Gibiansky, A. (2012, June 3). *Machine Learning: the Basics*. Retrieved December 29, 2017, from Andrew Gibiansky - Math Code: http://andrew.gibiansky.com/blog/machine-learning/machine-learning-the-basics/

https://i.stack.imgur.com/6j2gg.png. (n.d.).

Nature. (2016, 28 April). Anticipating Artificial Intelligence. *Nature, 532*(7600), 1.

(n.d.). *Neural Learning.*

Nielsen, M. (2017, December 2). *Using neural nets to recognize handwritten digits*. Retrieved December 29, 2017, from Neural Networks and Deep Learning: http://neuralnetworksanddeeplearning.com/chap1.html

Roja, R. (1996). *Weighted Networks – The Perceptron.* Freie Universität Berlin. Berlin: Freie Universität Berlin.

Sharma, S. (2016, September 6). *Activation Functions: Neural Networks*. Retrieved December 29, 2017, from Towards Data Science : https://towardsdatascience.com/activation-functions-neural-networks-1cbd9f8d91d6

8. Appendix

8.1. Perceptron Programming

Code written on Visual Basic to build a Perceptron that trains to learn the 'AND', 'OR' and 'XOR' problem:

```
Sub NeuralNetAND()

' Neural Network for the AND Problem

' Actual Data:

' Input 1, Input 2, Output
' 1, 1 -> 1
' 1, 0 -> 1
' 0, 1 -> 1
' 0, 0 -> 0

Dim InputValues(1 To 2) As Integer
Dim Weights(1 To 2) As Double
Dim Prediction As Double
Dim Error As Double

learningRate = 0.05

Dim WeightValues1(1 To 1000) As Double
Dim WeightValues2(1 To 1000) As Double

' Initialize Weights as Random Numbers between 0 and 1

For i = 1 To 2
   Weights(i) = Rnd()
Next

Range("D2:D62").ClearContents

' Backpropagation of the Weights

For j = 1 To 10

For i = 2 To 61

   InputValues(1) = Cells(i, 1)
   InputValues(2) = Cells(i, 2)
   Prediction = InputValues(1) * Weights(1) + InputValues(2) * Weights(2)
```

```
    Actual = Cells(i, 3)
    Error = Actual - Prediction

    ' Backpropagation: Delta Rule

    Weights(1) = Weights(1) + InputValues(1) * Error * learningRate
    Weights(2) = Weights(2) + InputValues(2) * Error * learningRate

    Z = Z + 1
    WeightValues1(Z) = Weights(1)
    WeightValues2(Z) = Weights(2)

Next
Next

' Do a final prediction on the Training Data
For i = 2 To 61
    Cells(i, 4) = Round(Cells(i, 1) * Weights(1) + Cells(i, 2) * Weights(2), 0)
Next

Cells(4, 8) = Weights(1)
Cells(4, 9) = Weights(2)

For i = 1 To Z
    Cells(i + 1, 6) = WeightValues1(i)
    Cells(i + 1, 7) = WeightValues2(i)
Next

End Sub
```

8.2. MLP Programming

Programming written on python 2 to construct an MLP that learns the 'XOR' problem:

```
import tensorflow as tf
import numpy as np

x_ = tf.placeholder(tf.float32, shape=[4,2], name="x-input")
y_ = tf.placeholder(tf.float32, shape=[4,1], name="y-input")

Theta1 = tf.Variable(tf.random_uniform([2,2], -1, 1), name="Theta1")
Theta2 = tf.Variable(tf.random_uniform([2,1], -1, 1), name="Theta2")

Bias1 = tf.Variable(tf.zeros([2]), name="Bias1")
Bias2 = tf.Variable(tf.zeros([1]), name="Bias2")
```

```
A2 = tf.sigmoid(tf.matmul(x_, Theta1) + Bias1)
Hypothesis = tf.sigmoid(tf.matmul(A2, Theta2) + Bias2)

cost = tf.reduce_mean(( (y_ * tf.log(Hypothesis)) +
       ((1 - y_) * tf.log(1.0 - Hypothesis)) ) * -1)

train_step = tf.train.GradientDescentOptimizer(0.5).minimize(cost)

XOR_X = [[0,0],[0,1],[1,0],[1,1]]
XOR_Y = [[0],[1],[1],[0]]


with tf.Session() as sess:
   sess.run(tf.global_variables_initializer())

   print("Input Data: (0,0), (0, 1), (1, 0), (1,1)")
   print("Expected Output Data: (0, 1, 1, 0)")

   print("Beginning Hypothesis for XOR Data: ")
   print('Hypothesis ', sess.run(Hypothesis, feed_dict={x_: XOR_X, y_: XOR_Y}))

   for i in range(2000):
         sess.run(train_step, feed_dict={x_: XOR_X, y_: XOR_Y})
         print('cost ', sess.run(cost, feed_dict={x_: XOR_X, y_: XOR_Y}))

   print("Resulting Hypothesis for XOR Data: ")
   print('Hypothesis ', sess.run(Hypothesis, feed_dict={x_: XOR_X, y_: XOR_Y}))
```