Lab Report on An Artificial Neural Network

Sumit Yadav (076BCT088)
Institute of Engineering, Pulchowk Campus

February 18, 2023

Abstract

Artificial Neural Networks (ANNs) are computational models inspired by the structure and function of the human brain, designed to solve various complex problems. The behavior of an ANN depends on both the weights and the input-output function specified for the units. There are three types of transfer functions: linear, threshold, and sigmoid. To make an ANN that performs a specific task, the units must be connected, and weights on the connections should be set appropriately. ANNs can be trained by presenting them with training examples and using the Delta Rule or Backpropagation algorithm to adjust the weights. The Adaline Learning algorithm uses the Delta Rule, while Backpropagation provides a computational efficient method for training multi-layer networks. These algorithms update the weights of the connections to produce better approximations of the desired output, ultimately making ANNs powerful tools for problem-solving in various fields.

1 Introduction

Artificial Neural Networks (ANN) are computational models that mimic the structure and function of the human brain. ANNs consist of interconnected processing units (neurons) that work together to learn and solve complex problems. The behavior of an ANN depends on the weights and the transfer function specified for the units. The transfer function can be linear, threshold, or sigmoid, and the weights determine the strength of the influence between units.

Two popular learning algorithms for training ANNs are Adaline learning and Backpropagation learning. Adaline learning uses the Delta Rule to adjust the weights and the bias to minimize the difference between the actual output and the desired output. Backpropagation learning is a computational efficient method for training multi-layer networks. It propagates the errors backward through the weights of the hidden units and adjusts the weights to reduce the error.

1.1 Adaline Learning:

Adaline learning, also known as the least mean squares (LMS) or Widrow-Hoff rule, uses the Delta Rule to adjust the weights and the bias. The Delta Rule updates the weights and the bias based on the difference between the actual output and the desired output. Adaline learning is a supervised learning algorithm that requires a training set of input-output pairs to learn from.

Steps involved in Adaline Learning:

1. Initialize the weights to small random values and select a learning rate, α
2. For each input vector $\mathbf{s}$, with target output $\mathbf{t}$, set the inputs to $\mathbf{s}$

3. Compute the neuron inputs $y_{in} = b + \sum_i x_i w_i$

4. Compute the error, $\delta = t - y_{in}$

5. Update the bias and weights using the delta rule:

$$b_{new} = b_{old} + \alpha\delta$$
$$w_{i,new} = w_{i,old} + \alpha\delta x_i$$

6. Stop if the largest weight change across all the training samples is less than a specified tolerance, otherwise cycle through the training set again

1.2 Backpropagation Learning:

Backpropagation learning is a popular algorithm for training multi-layer ANNs. It uses the chain rule of differentiation to propagate the errors backward through the weights of the hidden units and adjust the weights to reduce the error. Backpropagation learning requires a training set of input-output pairs and is a supervised learning algorithm.

Steps involved in Backpropagation:

1. Initialize the weights to small random values

2. Feed the training sample through the network and determine the final output

3. Compute the error for each output unit, for unit k it is:

$$\delta_k = (t_k - y_k) f'(y_{in,k})$$

4. Propagate the delta terms (errors) back through the weights of the hidden units where the delta input for the jth hidden unit is:

$$\delta_{in,j} = f'(z_{in,j}) \sum_k \delta_k w_{jk}$$

5. Calculate the weight correction term for each output unit, for unit k it is:

$$\Delta w_{jk} = \alpha\delta_k z_j$$

6. Calculate the weight correction term for the hidden units:

$$\Delta w_{ij} = \alpha\delta_{in,j} x_i$$

7. Update the weights:

$$w_{ij}(new) = w_{ij}(old) + \Delta w_{ij}$$
$$w_{jk}(new) = w_{jk}(old) + \Delta w_{jk}$$

8. Test for stopping (maximum epoch, small changes, etc.)

2 Experimental Setup

For this lab assignment, the experiment setup consisted of using a laptop with the following specifications:

1. Processor: Intel i5 8th generation
2. RAM: 8 GB
3. Operating System: Linux

The experiments were conducted on this laptop using Python.(Notebook)

3 Analysis

3.1 Question 1

Design a McCulloch-Pitts neural network which behaves as AND function using Adaline learning. Consider unipolar case. Perform analysis by varying parameters

3.2 Question 2

Similarly, develop a McCulloch-Pitts neural net for OR, NAND and NOR gate and draw neural nets.
Code

```
\begin{lstlisting}[language=Python]
\def adaline(X,y):
    w = np.zeros(2)
    b = 0.0
    alpha = 0.1
    iterations = 10
    deltas = []
    for iteration in range(iterations):
        for i in range(4):
            y_in = b + np.dot(X[i],w)
            y_out = np.where(y_in >=0,1,0)
            delta = y[i] - y_out
            w += alpha*deltaX[i]
            b += alpha*delta
            deltas.append(delta)
        print('Weight',w,'bias',b,'\n')
        for i in range(4):
            y_in = np.dot(X[i],w) +b
            y_out = np.where(y_in >= 0,1,0)
            print('input', X[i], 'output', y_out)
        print('\n')
    return deltas,w,b
```

AND gate

```
X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
y = np.array([0, 0, 0, 1])
```

```
delta,w,b = adaline(X,y)
```

OR gate

```
X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
y = np.array([0, 1, 1, 1])
delta,w,b = adaline(X,y)
```

NAND gate

```
X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
y = np.array([1, 1, 1, 0])
delta,w,b = adaline(X,y)
```

NOR gate

```
X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
y = np.array([1, 0, 0, 0])
delta,w,b = adaline(X,y)
```

3.3 Question 3

Perform test for bipolar model as well.
Code

```
\begin{lstlisting}[language=Python]
\def adaline(X,y):
    w = np.zeros(2)
    b = 0.0
    alpha = 0.1
    iterations = 10
    deltas = []
    for iteration in range(iterations):
        for i in range(4):
            y_in = b + np.dot(X[i],w)
            y_out = np.where(y_in >=0,1,-1)
            delta = y[i] - y_out
            w += alpha*deltaX[i]
            b += alpha*delta
            deltas.append(delta)
        print('Weight',w,'bias',b,'\n')
        for i in range(4):
            y_in = np.dot(X[i],w) +b
            y_out = np.where(y_in >= 0,1,0)
            print('input', X[i], 'output', y_out)
        print('\n')
    return deltas,w,b
```

AND gate

```
X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
y = np.array([0, 0, 0, 1])
delta,w,b = adaline(X,y)
```

OR gate

```
X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
y = np.array([0, 1, 1, 1])
delta,w,b = adaline(X,y)
```

NAND gate

```
X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
y = np.array([1, 1, 1, 0])
delta,w,b = adaline(X,y)
```

NOR gate

```
X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
y = np.array([1, 0, 0, 0])
delta,w,b = adaline(X,y)
```

3.4 Question 3

Implement McCulloch-Pitts neural network model for XOR and give all the formula you used in the implementation. Draw the MLPs used for the implementation of above functions.

The boolean representation of an XOR gate is;
x1x‘2 + x‘1x2
We first simplify the boolean expression
x‘1x2 + x1x‘2 + x‘1x1 + x‘2x2
x1(x‘1 + x‘2) + x2(x‘1 + x‘2)
(x1 + x2)(x‘1 + x‘2)
(x1 + x2)(x1x2)‘
From the simplified expression, we can say that the XOR gate consists of an OR gate $(x1 + x2)$, a NAND gate $(x1x2)$, and an AND gate of both.

CODE

```
X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
y = np.array([-1, 1, 1, -1])
y_nand = np.array([1,1,1,-1])
y_or = np.array([-1,1,1,1])

def bi_adaline(X,y):
    w = np.zeros(2)
    b = 0.0
    alpha = 0.1
    iterations = 10
    deltas = []
    y_output = []
    for iteration in range(iterations):
        for i in range(4):
            y_in = b + np.dot(X[i],w)
            y_out = np.where(y_in >=0,1,-1)
            delta = y[i] - y_out
            w += alphadeltaX[i]
```

```
            b += alpha*delta
            deltas.append(delta)
        print('Weight',w,'bias',b,'\n')
        for i in range(4):
            y_in = np.dot(X[i],w) +b
            y_out = np.where(y_in >= 0,1,-1)
            print('input', X[i], 'output', y_out)
        print('\n')
    for i in range(4):
        y_in = np.dot(X[i],w) +b
        y_out = np.where(y_in >= 0,1,-1)
        y_output.append(y_out)
    return deltas,w,b, y_output
```

Combining NAND AND OR

```
,,,nand_out = bi_adaline(X,y_nand)
,,,or_out = bi_adaline(X,y_or)
X_xor = np.vstack([nand_out, or_out]).T
deltas, w,b,out = bi_adaline(X_xor,y)
```

3.5 Question 5

Implement MLP model for XOR by using backpropagation algorithm CODE

```
X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
y = np.array([[0], [1], [1], [0]])
```

Activation Function

```
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

def sigmoid_derivative(x):
    return sigmoid(x) * (1 - sigmoid(x))
```

Back Propagation

```
def backprop(X, y, hidden_size, num_iterations, learning_rate):

    w1 = np.random.randn(2, hidden_size)
    b1 = np.zeros((1, hidden_size))
    w2 = np.random.randn(hidden_size, 1)
    b2 = np.zeros((1, 1))
    errors = []
    change_point = []

    for iteration in range(num_iterations):
        #Forward Pass
        z1 = np.dot(X, w1) + b1
        a1 = sigmoid(z1)
        z2 = np.dot(a1, w2) + b2
        y_hat = sigmoid(z2)
```

```
        #Error and derivative
        error = y - y_hat
        delta2 = error * sigmoid_derivative(z2)

        error1 = delta2.dot(w2.T)
        delta1 = error1 * sigmoid_derivative(z1)

        # Update the weights and biases
        w2 += learning_rate * a1.T.dot(delta2)
        b2 += learning_rate * np.sum(delta2, axis=0, keepdims=True)
        w1 += learning_rate * X.T.dot(delta1)
        b1 += learning_rate * np.sum(delta1, axis=0)
        errors.append(np.sum(error, axis = 0))
        print('error', np.sum(error, axis = 0))

        #Testing
        out1 = np.dot(X,w1) + b1
        out1_sig = sigmoid(out1)
        out2 = np.dot(out1_sig,w2) + b2
        final = sigmoid(out2)
        change_point.append(final)
        print('input','\n',X,'\n','output:-','\n',final)

    return w1, b1, w2, b2, errors,change_point
```

calling backprop

```
    w1, b1, w2, b2, error, changes = backprop(X,y,hidden_size = 2,
        num_iterations= 10000,learning_rate= 0.2)
```

4 Conclusion

In this series of assignments, we were tasked with designing and implementing neural networks for various logical functions using the McCulloch-Pitts and backpropagation algorithms. For the McCulloch-Pitts neural network, we designed networks for AND, OR, NAND, and NOR gates using Adaline learning in both unipolar and bipolar cases. We also implemented the network for the XOR gate and simplified the Boolean expression to understand its composition. We then used the output of the NAND and OR gate networks as input to a new McCulloch-Pitts network to implement the XOR function. Finally, we implemented the XOR function using the backpropagation algorithm in an MLP model.

Through the assignments, we gained a solid understanding of the workings of neural networks and their application to logical functions. We learned how the structure and parameters of the networks can affect their performance and how to optimize them for better accuracy. The assignments also allowed us to explore the differences between unipolar and bipolar cases and how they affect the implementation of the networks. Overall, these assignments provided a great foundation for understanding neural networks and their application in real-world scenarios.

References

1. McCulloch, W. S., Pitts, W. (1943). A logical calculus of the ideas immanent in nervous activity. The bulletin of mathematical biophysics, 5(4), 115-133.

2. Widrow, B., Hoff, M. E. (1960). Adaptive switching circuits. IRE WESCON Convention Record, 96-104.

3. Haykin, S. (1994). Neural networks: a comprehensive foundation (Vol. 2). Prentice Hall PTR.

4. Rumelhart, D. E., Hinton, G. E., Williams, R. J. (1986). Learning representations by back-propagating errors. nature, 323(6088), 533-536.

5. Hecht-Nielsen, R. (1992). Theory of the backpropagation neural network. Neural networks for perception, 1-42.

Results

Here is a graph showing the results of Adaline Learning:

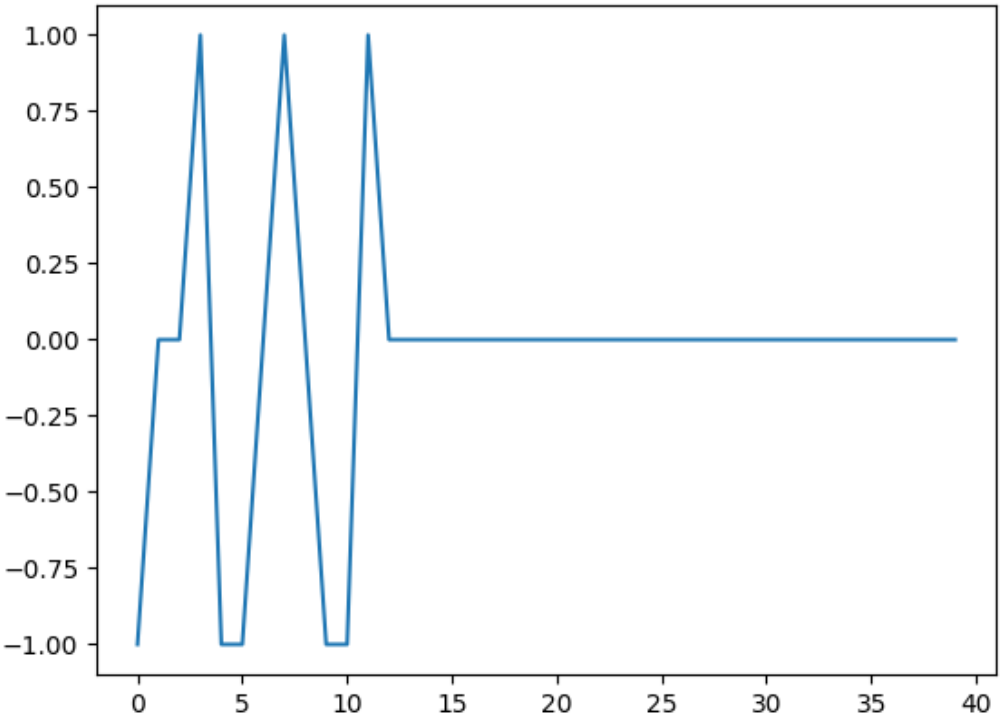

Figure 1: Plot of error converging in AND

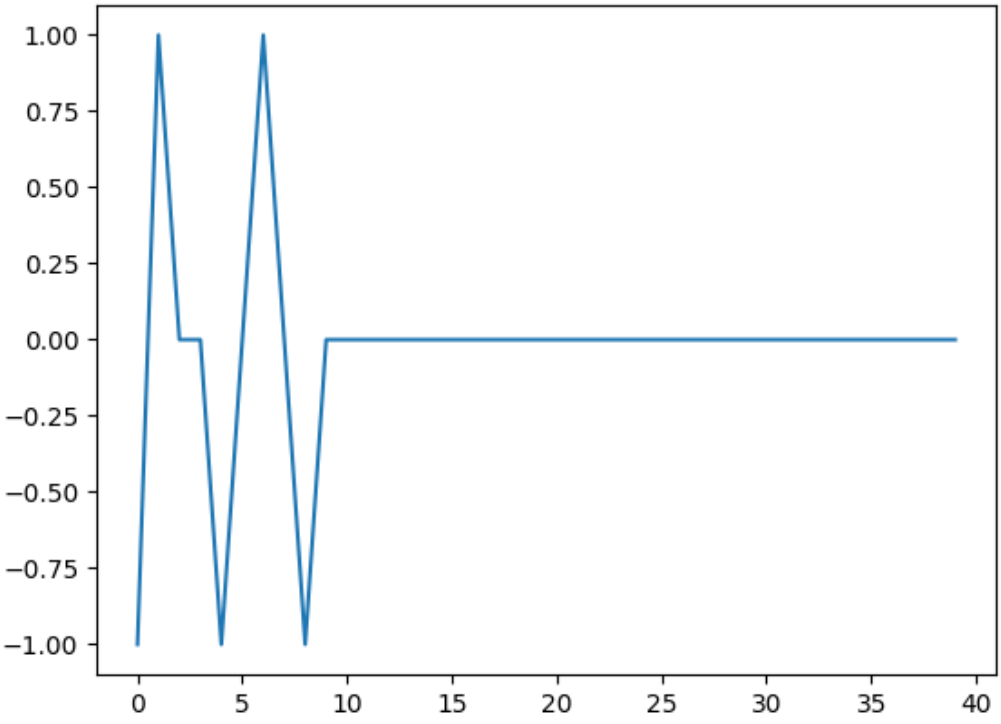

Figure 2: Plot of error converging in OR

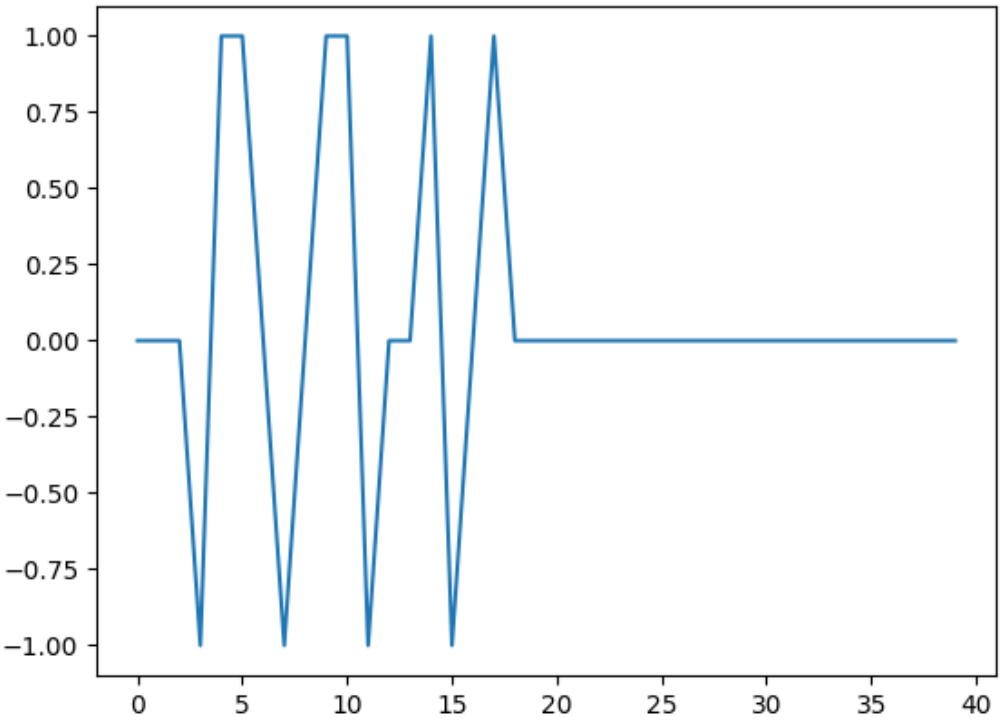

Figure 3: Plot of error converging in NAND

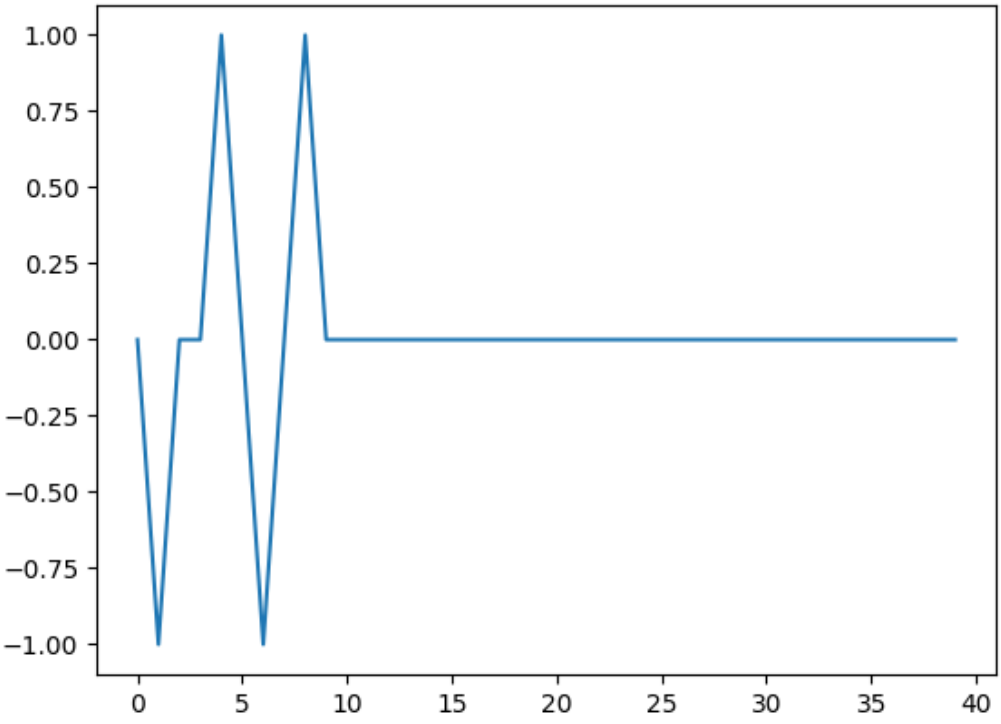

Figure 4: Plot of error converging in NOR

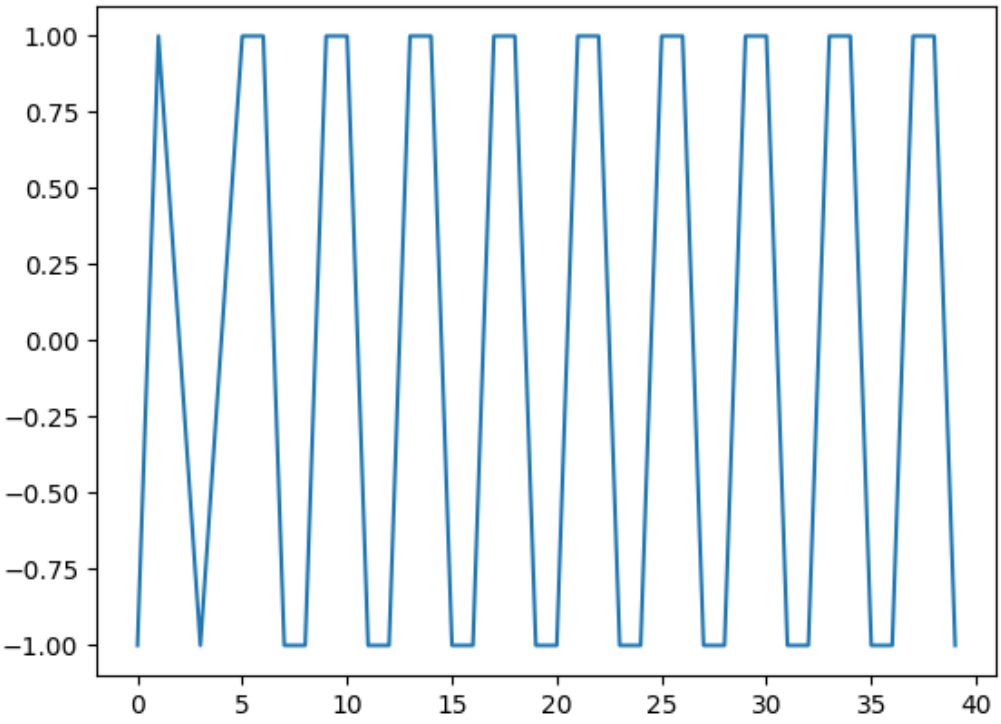

Figure 5: Plot of error not converging in XOR

Here is a graph showing the results of Back Propagation:

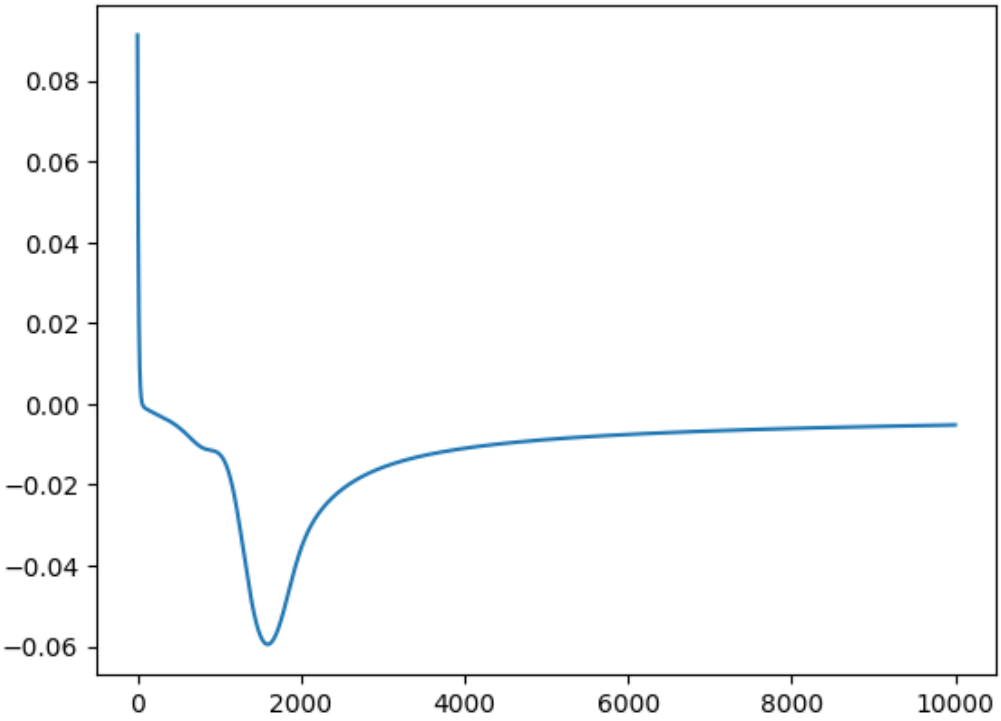

Figure 6: Plot of error vs iterations

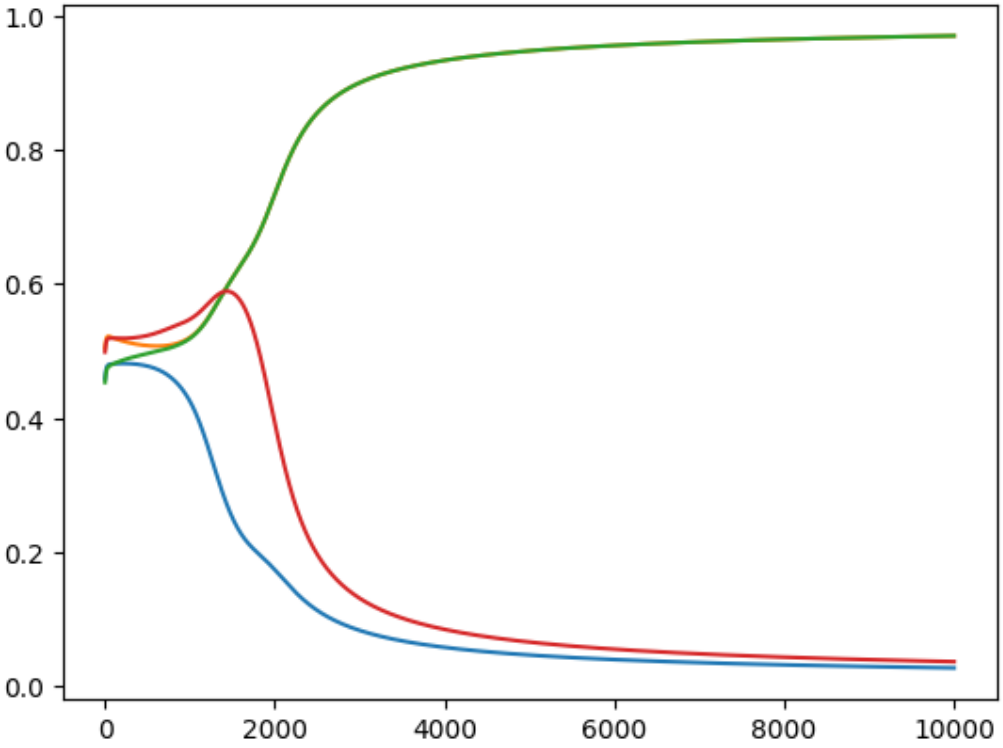

Figure 7: Conversion of x1,x2,x3,x4 value to 1 and 0