# Implementation of XOR Using Python

CSE 401: Artificial Intelligence

Utkarsh Gupta
A2305217557
7CSE 8Y

October 07th, 2020

## 1 Implementation of XOR using Python

Implementing logic gates using neural networks help understand the mathematical computation by which a neural network processes its inputs to arrive at a certain output. This neural network will deal with the XOR logic problem. An XOR (exclusive OR gate) is a digital logic gate that gives a true output only when both its inputs differ from each other. The truth table for an XOR gate is shown below:

The goal of the neural network is to classify the input patterns according to the above truth table. If the input patterns are plotted according to their outputs, it is seen that these points are not linearly separable. Hence the neural network has to be modeled to separate these input patterns using decision planes.

### 1.1 THE NEURAL NETWORK MODEL

As mentioned before, the neural network needs to produce two different decision planes to linearly separate the input data based on the output patterns. This is achieved by using the concept of hidden layers. The neural network will consist of one input layer with two nodes (X1,X2); one hidden layer with two nodes (since two decision planes are needed); and one output layer with one node (Y). Hence, the neural network looks like this:

### 1.2 THE SIGMOID NEURON

To implement an XOR gate, I will be using a Sigmoid Neuron as nodes in the neural network. The characteristics of a Sigmoid Neuron are: 1. Can accept real values as input. 2. The value of the activation is equal to the weighted sum of its inputs i.e. wi xi 3. The output of the sigmoid neuron is a function of the sigmoid function, which is also known as a logistic regression function. The sigmoid function is a continuous function which outputs values between 0 and 1:

### 1.3 THE LEARNING ALGORITHM

The information of a neural network is stored in the interconnections between the neurons i.e. the weights. A neural network learns by updating its weights according to a learning algorithm that helps it converge to the expected output. The learning algorithm is a principled way of changing the weights and biases based on the loss function. 1. Initialize the weights and biases randomly. 2. Iterate over the data i. Compute the predicted output using the sigmoid function ii. Compute the loss using the square error loss function iii. W(new) = W(old) — W iv. B(new) = B(old) — B 3. Repeat until the error is minimal

This is a fairly simple learning algorithm consisting of only arithmetic operations to update the weights and biases. The algorithm can be divided into two parts: the forward pass and the backward pass also known as "backpropagation."

## 1.4 GRADIENT DESCENT

The loss function of the sigmoid neuron is the squared error loss. If we plot the loss/error against the weights we get something like this:

Our goal is to find the weight vector corresponding to the point where the error is minimum i.e. the minima of the error gradient. And here is where calculus comes into play.

## 1.5 THE MATH BEHIND GRADIENT DESCENT

Error can be simply written as the difference between the predicted outcome and the actual outcome. Mathematically:

where t is the targeted/expected output & y is the predicted output.

However, is it fair to assign different error values for the same amount of error? For example, the absolute difference between -1 and 0 & 1 and 0 is the same, however the above formula would sway things negatively for the outcome that predicted -1. To solve this problem, we use square error loss.(Note modulus is not used, as it makes it harder to differentiate). Further, this error is divided by 2, to make it easier to differentiate, as we'll see in the following steps.

Since, there may be many weights contributing to this error, we take the partial derivative, to find the minimum error, with respect to each weight at a time. The change in weights are different for the output layer weights ($W_{31}$ & $W_{32}$) and different for the hidden layer weights ($W_{11}$, $W_{12}$, $W_{21}$, $W_{22}$). Let the outer layer weights be $w_o$ while the hidden layer weights be $w_h$.

We'll first find W for the outer layer weights. Since the outcome is a function of activation and further activation is a function of weights, by chain rule:

On solving,

Note that for $X_o$ is nothing but the output from the hidden layer nodes. This output from the hidden layer node is again a function of the activation and correspondingly a function of weights. Hence, the chain rule expands for the hidden layer weights:

Which comes to,

NOTE: $X_o$ can also be considered to be $Y_h$ i.e. the output from the hidden layer is the input to the output layer. $X_h$ is the input to the hidden layer, which are the actual input patterns from the truth table.

[1]:
```
# Import the necessary library
import numpy as np
# np.random.seed(0)
```

[2]:
```python
def sigmoid (x):
    return 1/(1 + np.exp(-x))
```

[3]:
```python
def sigmoid_derivative(x):
    return x * (1 - x)
```

[4]:
```python
# Set the Input datasets
inputs = np.array([[0,0],[0,1],[1,0],[1,1]])
# Set the expected output
expected_output = np.array([[0],[1],[1],[0]])
```

[5]:
```python
epochs = 10000
lr = 0.3

inputLayerNeurons = 2
hiddenLayerNeurons = 2
outputLayerNeurons = 1
```

[6]:
```python
# Random weights and bias initialization
hidden_weights = np.random.uniform(size=(inputLayerNeurons,hiddenLayerNeurons))
hidden_bias = np.random.uniform(size=(1,hiddenLayerNeurons))
output_weights = np.random.uniform(size=(hiddenLayerNeurons,outputLayerNeurons))
output_bias = np.random.uniform(size=(1,outputLayerNeurons))
```

[7]:
```python
# Display the hidden_weights, hidden_bias, output_weights, and output_bias
print("Initial hidden weights: ",end='')
print(*hidden_weights)
print("Initial hidden biases: ",end='')
print(*hidden_bias)
print("Initial output weights: ",end='')
print(*output_weights)
print("Initial output biases: ",end='')
print(*output_bias)
```

```
Initial hidden weights: [0.34082471 0.58838842] [0.03983071 0.61347076]
Initial hidden biases: [0.70793896 0.1218982 ]
Initial output weights: [0.45504822] [0.80003615]
Initial output biases: [0.23861191]
```

[8]:
```python
# Training algorithm

for _ in range(epochs):
    # Forward Propagation
    hidden_layer_activation = np.dot(inputs,hidden_weights)
    hidden_layer_activation += hidden_bias
    hidden_layer_output = sigmoid(hidden_layer_activation)
    output_layer_activation = np.dot(hidden_layer_output,output_weights)
    output_layer_activation += output_bias
    predicted_output = sigmoid(output_layer_activation)

    # Backpropagation
    error = expected_output - predicted_output
    d_predicted_output = error * sigmoid_derivative(predicted_output)

    error_hidden_layer = d_predicted_output.dot(output_weights.T)

    d_hidden_layer = error_hidden_layer * sigmoid_derivative(hidden_layer_output)

    # Updating Weights and Biases
    output_weights += hidden_layer_output.T.dot(d_predicted_output) * lr
    output_bias += np.sum(d_predicted_output,axis=0,keepdims=True) * lr
    hidden_weights += inputs.T.dot(d_hidden_layer) * lr
    hidden_bias += np.sum(d_hidden_layer,axis=0,keepdims=True) * lr
```

[9]:
```
# Display the hidden_weights, hidden_bias, output_weights and output_bias after␣
 ↪training
print("Final hidden weights: ",end='')
print(*hidden_weights)
print("Final hidden bias: ",end='')
print(*hidden_bias)
print("Final output weights: ",end='')
print(*output_weights)
print("Final output bias: ",end='')
print(*output_bias)
```

```
Final hidden weights: [4.34277639 6.39037088] [4.33371443 6.34946103]
Final hidden bias: [-6.65875322 -2.80326596]
Final output weights: [-9.76838325] [9.03058284]
Final output bias: [-4.13619758]
```

[10]:
```
# Finally, display the predicted output
print("\nOutput from neural network after 10,000 epochs: ",end='')
print(*predicted_output)
```

```

Output from neural network after 10,000 epochs: [0.02576666] [0.97749278]
[0.97755129] [0.02347608]
```

## 2 BONUS QUESTION:

Implement any other logic gate using the above steps

### 2.1 AND Gate

[11]:
```
# Set the Input datasets
inputs = np.array([[0,0],[0,1],[1,0],[1,1]])
# Set the expected output
expected_output_AND = np.array([[0],[0],[0],[1]])
```

[12]:
```
# Training algorithm
for _ in range(epochs):
    # Forward Propagation
    hidden_layer_activation = np.dot(inputs,hidden_weights)
    hidden_layer_activation += hidden_bias
    hidden_layer_output = sigmoid(hidden_layer_activation)

    output_layer_activation = np.dot(hidden_layer_output,output_weights)
    output_layer_activation += output_bias
    predicted_output = sigmoid(output_layer_activation)

    # Backpropagation
    error = expected_output_AND - predicted_output
    d_predicted_output = error * sigmoid_derivative(predicted_output)

    error_hidden_layer = d_predicted_output.dot(output_weights.T)

    d_hidden_layer = error_hidden_layer * sigmoid_derivative(hidden_layer_output)

    # Updating Weights and Biases
    output_weights = output_weights + hidden_layer_output.T.
 ↪dot(d_predicted_output) * lr
    output_bias += np.sum(d_predicted_output,axis=0,keepdims=True) * lr
    hidden_weights = hidden_weights + inputs.T.dot(d_hidden_layer) * lr
    hidden_bias += np.sum(d_hidden_layer,axis=0,keepdims=True) * lr
```

[13]:
```
#Finally, display the predicted output for Bonus part (AND gate)
print("\nOutput from neural network after 10,000 epochs for AND gate: ",end='')
print(*predicted_output)
```

```
Output from neural network after 10,000 epochs for AND gate: [0.01980805]
[0.00534951] [0.00536385] [0.98345295]
```

### 2.2 NAND Gate

[14]:
```
# Set the Input datasets
inputs = np.array([[0,0],[0,1],[1,0],[1,1]])
# Set the expected output
expected_output = np.array([[1],[1],[1],[0]])
```

[15]:
```
for _ in range(epochs):
    # Forward Propagation
    hidden_layer_activation = np.dot(inputs,hidden_weights)
    hidden_layer_activation += hidden_bias
    hidden_layer_output = sigmoid(hidden_layer_activation)

    output_layer_activation = np.dot(hidden_layer_output,output_weights)
    output_layer_activation += output_bias
    predicted_output = sigmoid(output_layer_activation)

    # Backpropagation
    error = expected_output - predicted_output
    d_predicted_output =  error * sigmoid_derivative(predicted_output)

    error_hidden_layer = d_predicted_output.dot(output_weights.T)

    d_hidden_layer = error_hidden_layer * sigmoid_derivative(hidden_layer_output)

    # Updating Weights and Biases
    output_weights = output_weights + hidden_layer_output.T.
 ↪dot(d_predicted_output) * lr
    output_bias += np.sum(d_predicted_output,axis=0,keepdims=True) * lr
    hidden_weights = hidden_weights + inputs.T.dot(d_hidden_layer) * lr
    hidden_bias += np.sum(d_hidden_layer,axis=0,keepdims=True) * lr
```

[16]:
```
# Finally, display the predicted output
print("\nOutput from neural network after 10,000 epochs: ",end='')
print(*predicted_output)
```

```
Output from neural network after 10,000 epochs: [0.99022717] [0.99999448]
[0.99999443] [0.99999443]
```