Coding a neural network for XOR logic from scratch

In this repository, I implemented a proof of concept of all my theoretical knowledge of neural network to code a simple neural network from scratch in Python without using any machine learning library.

Introduction:

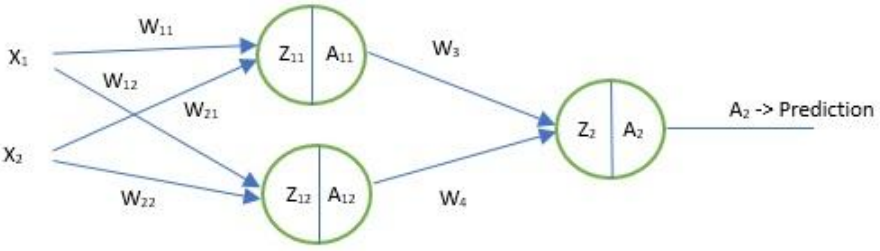
Some machine learning algorithms like neural networks are already a black box, we enter input in them and expect magic to happen. Still, it is important to understand what is happening behind the scenes in a neural network. We can use libraries such as Keras to make our life easy but if we dont understand what happens inside a neural network, then we can easily get stuck in an infinite loop without understanding what is going wrong with our neural network. Coding a simple neural network from scratch acts as a Proof of Concept in this regard and further strengthens our understanding of neural networks.

In this project, a single hidden layer neural network is used, with sigmoid activation function in hidden layer units and sigmoid activation function for output layer too, since the output of XOR logic is binary i.e. 0 or 1 only one neuron is in the output layer. The maths behind neural network is explained below:

Explanation of Maths behind Neural Network:

Following work shows the maths behind a single hidden layer neural network:

X_1	X_2	$Y = X_1$ XOR X_2
0	0	0
0	1	1
1	0	1
1	1	0



For simplicity, we consider weights 'W_{11}','W_{12}','W_{21}' and 'W_{22}' as weight vector 'W_1'. Weights 'W_3' and 'W_4' as weight vector 'W_2'.

Here 'Z' is the dot product of weight vector 'W' and input vector 'X' added with bias. Again, we vectorize 'Z_{11}' and 'Z_{12}' as 'Z_1' and 'Z_2' remains the same.

Lastly, 'A' is the activation function. We used sigmoid activation functions in our network. Vectorizing 'A_{11}' and 'A_{12}' gives us 'A_1' and 'A_2' is kept the same.

We consider Z For simplicity, we didn't include bias in the network. A model can work without bias unit but it cannot work without weights, as then machine will not learn anything if weights don't change.

After all calculations, 'Y' is our network prediction.

Lets see what happens inside a neural network training process. It has 4 parts:

a) Forward Propagation:

In forward propagation, all 'Z' and 'A' will be calculated until we get to the end of our network giving us our prediction 'Y'. This process is quite straight-forward.

Mathematically Z and A are given as,

$$Z_1 = W_1.X+b_1 \quad \text{---------- (1)}$$

$$A_1 = \frac{1}{1+e^{-Z_1}} \quad \text{---------- (2)}$$

$$Z_2 = W_2.A_1+b_2 \quad \text{---------- (3)}$$

$$Y = A_2 = \frac{1}{1+e^{-Z_2}} \quad \text{---------- (4)}$$

Where 'Z_1' is dot product of weight and input vector (we didn't include bias b_1 in our network), 'A_1' is the mathematical formula of sigmoid function, 'Z_2' is the dot product of hidden layer output 'A_1' and 'W_2' (ignoring bias b_2) and 'Y' is the output prediction given by sigmoid activation in output neuron.

b) Calculating Loss function value:

Loss function value is an estimate of how our network is performing. If this value is high and keeps increasing, that means our network is not performing well. If this value is decreasing and at the end of iterations, reaches a low value, then we can say that our network is doing well, our model is learning

because loss functions is converging. For binary outputs, we use formula of binary cross-entropy for loss function value in Equation (5).

$$L = -\left(\frac{1}{m}\right) * \sum(Y * log(\mathrm{A2}) + (1 - \mathrm{Y}) * \log(1 - \mathrm{A2})) \quad \text{---------- (5)}$$

Where 'L' is the loss value,
'Y' is the output that we already know, since this is supervised learning, we will train our model with known outputs contained in 'Y',
'A_2' is output of our network after each forward propagation.
Using this formula, we plot the loss function value to see if our network loss is decreasing or not after iterations.

c) Backward Propagation:

In backward propagation, we back propagate the loss that we encountered in our model to tweak the weights and biases such that it decreases the loss function value. Considering our network, we start backpropagation with A2 and we try to find the rate of change of loss with respect to weight vector W_2 using chain rule of derivative and then use gradient descent's formula to tweak the weights. Equation 6 shows that process:

$$\frac{\partial L}{\partial W2} = \frac{\partial L}{\partial A2} * \frac{\partial A2}{\partial Z2} * \frac{\partial Z2}{\partial W2} \quad \text{---------- (6)}$$

We use the above equation to find rate of change of loss with respect to weight vector 'W_2' where $\frac{\partial L}{\partial A2}$ is found by taking derivative of loss function 'L' with respect to activation function 'A_2'. After derivative, we get output shown in Equation 7,

$$\frac{\partial L}{\partial A2} = (-\frac{1}{m})(\frac{Y-A2}{A2(1-A2)}) \quad \text{---------- (7)}$$

Then we find $\frac{\partial A2}{\partial Z2}$ by taking derivative of 'A_2' with respect to 'Z_2'. Taking derivative of sigmoid function in equation 4 gives us the following output:

$$\frac{\partial A2}{\partial Z2} = \mathrm{A2}*(1\text{-}\mathrm{A2}) \quad \text{---------- (8)}$$

Lastly, we take derivative of 'Z_2' with respect to 'W_2' in equation 3 as,

$$\frac{\partial Z2}{\partial W2} = A1 \quad \text{---------- (9)}$$

Now we try to find the rate of change of Loss with respect to Weight vector 'W1'. First, we find $\frac{\partial A1}{\partial Z1}$

$$\frac{\partial A1}{\partial Z1} = \mathrm{A_1} * (1\text{-}\mathrm{A_1}) \quad \text{---------- (10)}$$

Then we find $\frac{\partial Z1}{\partial W1}$ by taking derivative of equation 1,

$$\frac{\partial Z1}{\partial W1} = X \quad \text{---------- (11)}$$

Lastly, we multiply $\frac{\partial Z1}{\partial W1}$ and $\frac{\partial A1}{\partial Z1}$ with the input derivative coming from output layer i.e. W_2 and $\frac{\partial L}{\partial Z2}$, we get

$$\frac{\partial L}{\partial W1} = X * A1 * (1 - A1) * W2 * (A2 - Y) \quad \text{---------- (12)}$$

d) Weight update via Gradient Descent

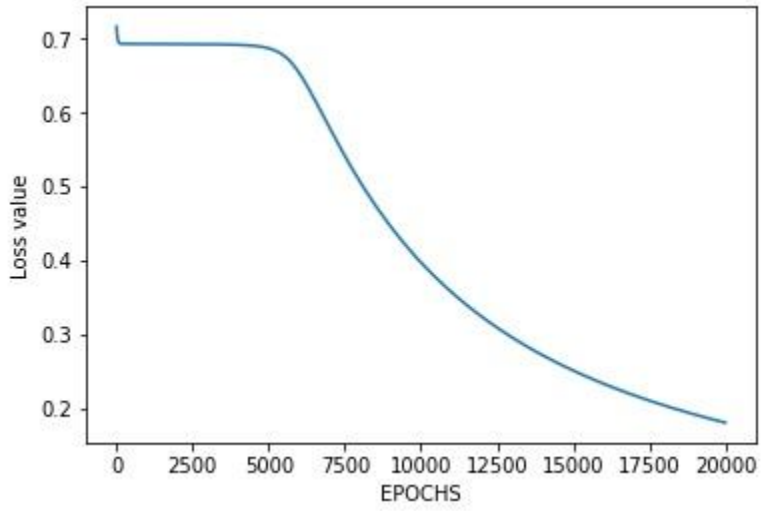
Finally, we update the weights using gradient descent formula as,

$$W = W - lr * \frac{\partial L}{\partial W} \quad \text{---------- (13)}$$

Where 'W' is weight vector, 'lr' is learning rate and $\frac{\partial L}{\partial W}$ is the rate of change of Loss w.r.t. weight. Thus, weights are updated per iteration also called EPOCH which is made up of one forward and one backward propagation, and we keep doing these EPOCHS and see if loss is decreasing until we settle at a point where loss is minimum and our classes are predicted perfectly.

Results

Following image shows the loss function for our network, it can be seen that it is decreasing.



Following are the predictions of neural network on test inputs:

```
In [19]: test = np.array([[1],[0]])
         predict(test)
         test = np.array([[0],[0]])
         predict(test)
         test = np.array([[0],[1]])
         predict(test)
         test = np.array([[1],[1]])
         predict(test)

         For input [1, 0] output is 1
         For input [0, 0] output is 0
         For input [0, 1] output is 1
         For input [1, 1] output is 0
```

Please view the jupyter notebook file attached, it has the code with comments to make it easy to understand for the readers.

Conclusion:

Coding a neural network from scratch strengthened my understanding of what goes on behind the scenes in a neural network. I hope this project helps other readers understand the working of a neural network.