Introduction to Neural Networks Assignment 02
Backpropagation Algorithm

Mangesh Sakordekar

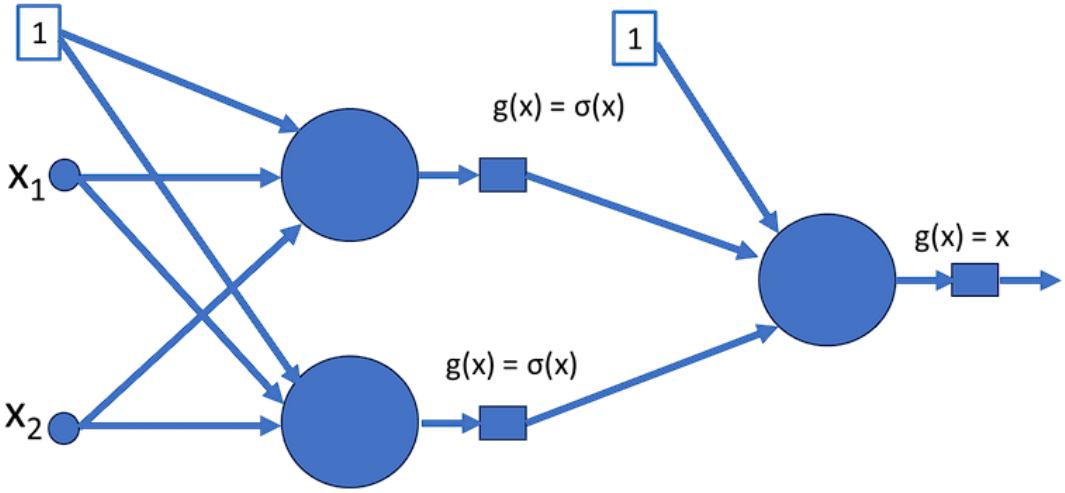
Overview

The aim of this project was to use the 3-layer network, with 1 input layer, 1 hidden layer, and 1 output layer, to implement the NEXOR function.

- Specify the minimum training input data, X, necessary to complete NEXOR.
- Calculate, manually, the results of the forward pass response to the input x = [0.25,0.75].
- Calculate the backward pass to find the corresponding weight adjustments, using learning rate=0.1.
- Write code to implement back-propagation for this kind of network
- Verify that the code produces the same results as manual calculations.
- Use the code to fully train the network
- Plot resulting output values in the range [0,1]
- Use this plot to validate code results.

The Model

Throughout this assignment, the following model will be used.



We assume that the initial weights of this model are as follows

$$\Theta^1 = \begin{bmatrix} 1 & -1 \\ 1 & -1 \\ 1 & -1 \end{bmatrix} \qquad \Theta^2 = \begin{bmatrix} 1 \\ -1 \\ 1 \end{bmatrix}$$

Minimum Training Input Data

To train this model, we need to have some known training data and labels so that we can successfully adjust the weights. In this case, we use the outputs of a NEXOR gate.

$$X = \begin{bmatrix} 1 & 1 \\ 1 & 0 \\ 0 & 1 \\ 0 & 0 \end{bmatrix} \qquad y = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 1 \end{bmatrix}$$

Manual Calculations for x = [0.25, 0.75]

Forward Pass

In the forward pass we need to calculate the values of the outputs of each layer. Apart from the input layer, all the other layers will have outputs before (a^k) and after (o^k) the activation function. In the input layer we just have the inputs and the bias as o^0. In the below notation, terms before the vertical line (|) are the bias terms.

$$o^0 = \begin{bmatrix} 1 & | & 0.25 & 0.75 \end{bmatrix}$$

To calculate the output of the neuron, we take the dot product of the values coming in from the previous layer and the weights of the current neuron. For the first hidden layer, a sigmoid activation function is used and we get the following outputs

$$a^1 = \begin{bmatrix} 1 & | & 2 & -2 \end{bmatrix} \qquad g(x) = \sigma(x) \qquad o^1 = \begin{bmatrix} 1 & | & 0.8808 & 0.1192 \end{bmatrix}$$

For the output layer, there is only one term.

$$a^2 = \begin{bmatrix} 0.2384 \end{bmatrix} \qquad g(x) = x \qquad o^2 = \begin{bmatrix} 0.2384 \end{bmatrix}$$

Backward Pass

Finding Error Terms

For this task, it is assumed that the correct output of [0.25, 0,75] is 0. First we calculate the error terms of each neuron, starting with the one in the output layer. We know, for the output layer, we can use the following formula.

$$\delta_1^m = \hat{y}_d - y_d \tag{1}$$

We know that $\hat{y}_1^2 = o^2$ and as we stated before, $y_d = 0$. Plugging this in equation 1

$$\delta_1^2 = 0.2384 - 0 = 0.2384$$

For the rest of the error terms we use the following formula

$$\delta_j^k = o_j^k \cdot (1 - o_j^k) \cdot \sum_{l=1}^{r^{k+1}} w_{jl}^{k+1} \cdot \delta_{jl}^{k+1}$$

Since in this case we only have one term in the output layer, we can simplify the formula to

$$\delta_j^1 = o_j^1 \cdot (1 - o_j^1) \cdot w_1^2 \cdot \delta_1^2 \tag{2}$$

We can use equation 2 to find the error terms of nodes in the hidden layer.

$$\delta_2^1 = o_2^1 \cdot (1 - o_2^1) \cdot w_1^2 \cdot \delta_1^2 = 0.8088 \cdot (1 - 0.8088) \cdot (-1) \cdot 0.2384 = -0.02503$$

$$\delta_1^1 = o_1^1 \cdot (1 - o_1^1) \cdot w_1^2 \cdot \delta_1^2 = 0.1192 \cdot (1 - 0.1192) \cdot 1 \cdot 0.2384 = 0.02503$$

Calculating Weight Adjustments
To calculate the weight adjustments (Δw_{ij}^k), we calculate the partial derivative of the error with respect to each weight and multiply it by the negative of the learning rate (α).

$$\Delta w_{ij}^k = -\alpha \cdot \frac{\partial E_d}{\partial w_{ij}^k} \tag{3}$$

We are using a learning rate of 0.1 in this model. To calculate the partial, we can use the following formula

$$\frac{\partial E_d}{\partial w_{ij}^k} = \delta_j^k \cdot o_i^{k-1} \tag{4}$$

Using equation 3 and equation 4, we can calculate the deltas of each weight.

Output Layer Deltas

$\frac{\partial E_d}{\partial w_{01}^2} = \delta_1^2 \cdot o_0^1 = 0.2384 \times 1 = 0.2384$ $\quad \Delta w_{01}^2 = -0.1 \times 0.2384 = -0.02384$

$\frac{\partial E_d}{\partial w_{11}^2} = \delta_1^2 \cdot o_1^1 = 0.2384 \times 0.8808 = 0.20998$ $\quad \Delta w_{11}^2 = -0.1 \times 0.20998 = -0.020998$

$\frac{\partial E_d}{\partial w_{21}^2} = \delta_1^2 \cdot o_2^1 = 0.2384 \times 0.1192 = 0.02842$ $\quad \Delta w_{21}^2 = -0.1 \times 0.02842 = -0.002842$

Hidden Layer Neuron 1 Deltas

$\frac{\partial E_d}{\partial w_{01}^1} = \delta_1^1 \cdot o_0^0 = -0.02503 \times 1 = -0.02503$ $\quad \Delta w_{01}^1 = -0.1 \times -0.02503 = 0.002503$

$\frac{\partial E_d}{\partial w_{11}^1} = \delta_1^1 \cdot o_1^0 = -0.02503 \times 0.25 = -0.00626$ $\quad \Delta w_{11}^1 = -0.1 \times -0.00626 = 0.000626$

$\frac{\partial E_d}{\partial w_{21}^1} = \delta_1^1 \cdot o_2^1 = -0.02503 \times 0.75 = -0.01877$ $\quad \Delta w_{21}^1 = -0.1 \times -0.01877 = 0.001877$

Hidden Layer Neuron 2 Deltas

$$\frac{\partial E_d}{\partial w_{02}^1} = \delta_2^1 \cdot o_0^0 = 0.02503 \times 1 = 0.02503 \qquad \Delta w_{02}^1 = -0.1 \times 0.2503 = -0.002503$$

$$\frac{\partial E_d}{\partial w_{12}^1} = \delta_2^1 \cdot o_1^0 = 0.02503 \times 0.25 = 0.00626 \qquad \Delta w_{12}^1 = -0.1 \times 0.00626 = -0.000626$$

$$\frac{\partial E_d}{\partial w_{22}^1} = \delta_2^1 \cdot o_2^1 = 0.02503 \times 0.75 = 0.01877 \qquad \Delta w_{22}^1 = -0.1 \times 0.01877 = -0.001877$$

Weight Adjustment Matrices

Thus, from the above calculations we get the following matrices.

$$\Delta\Theta^1 = \begin{bmatrix} 0.002503 & -0.002503 \\ 0.000626 & -0.000626 \\ 0.001877 & -0.001877 \end{bmatrix} \qquad \Delta\Theta^2 = \begin{bmatrix} -0.02384 \\ -0.020998 \\ -0.002842 \end{bmatrix} \tag{5}$$

Python Implementation

To implement the model in code, I created a python class called Model and set it's attributes to match the model provided in the diagram.

```
class Model:

    ######################### Variables ###################################
    weights = [ np.array([[1, 1, 1], [-1, -1, -1]]).T, np.array([1, -1, 1]).T]
    bias = [1,1]
    functs = [sigmoid, doNothing]
    learning_rate = 0.1

    epochs = 200000
    train_data = [np.array([0, 0]).T, np.array([1, 0]).T, np.array([0, 1]).T, np.array([1, 1]).T]
    train_labels = [1, 0, 0, 1]
```

Figure 1: Model declaration

Forward Pass

To emulate the forward pass, I put the weights matrix of each layer into a list and traversed the list from 0 to n. While doing so, I took the dot product of the weights matrix with the outputs of the previous layer and generated the outputs of the current layer. Then I added the bias to the output vector if necessary and move to the next iteration. The implementation can be seen in the screenshot below.

```
############################ Forward Pass ################################
def forwardPass(self, inputs):
    outputs = []

    inp = np.insert(inputs, 0, self.bias[0])

    outputs.append(inp)

    for i in range(0, len(self.weights)):
        a = np.dot(outputs[i], self.weights[i])
        o = self.functs[i](a)

        if i+1 < len(self.bias):
            o = np.insert(o, 0, self.bias[i+1])

        outputs.append(o)

    return outputs


def predict(self, inputs):
    outputs = self.forwardPass(inputs)
    return outputs[2]
```

Figure 2: Forward pass implementation

To assist this function, I coded implementations of the activation functions separately.

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

def doNothing(x):
    return x
```

Figure 3: Activation functions

Backward Pass

To implement the backward pass, I spit the tasks into two parts - calculating errors and calculating weight adjustments. The code implementation can be seen below.

```
######################### Backwards Pass ###################################
def backwardPass(self, outputs, exp_out):
    errors = self.calcErrors(outputs, exp_out)

    deltas = self.calcDeltas(outputs, errors)
    return deltas
```

Figure 4: Backward pass implementation

The error of the output layer, is just the difference between the predicted output and expected output. Once that was calculated, I traversed the outputs list created in the forward pass in reverse order and implemented equation 2 to calculate errors for each neuron. The error terms for each layer were stores in a list and returned. These terms will be used in calculating weight adjustments.

```
def calcErrors(self, outs, exp_out):
    errors = []

    eo = [outs[-1] - exp_out]
    errors.append(eo)

    i = len(outs) - 2

    while i > 0:

        el = []
        j = 1

        while j < len(outs[i]):
            er = outs[i][j] * (1-outs[i][j]) * self.weights[i][j] * errors[0][0]
            el.append(er)
            j+=1

        errors.insert(0, el)
        i-=1

    return errors
```

Figure 5: Calculating error terms implementation

Since we have all the required terms, calculating the weight adjustments is just implementing the equation 3 and equation 4. To achieve this, I traversed over each of the weights and calculated its adjustment and stored it in an array of the same shape as the weights.

```
def calcDeltas(self, outs, errors):
    deltas = []
    for i in range(0, len(self.weights)):
        dlts = np.zeros(self.weights[i].shape)
        for j in range(0, dlts.shape[0]):
            if self.weights[i].ndim > 1:
                for k in range(0, dlts.shape[1]):
                    dlts[j,k] = -1 * self.learning_rate * errors[i][k] * outs[i][j]
            else:
                dlts[j] = -1 * self.learning_rate * errors[i][0] * outs[i][j]

        deltas.append(dlts)

    return deltas
```

Figure 6: Calculating weight adjustments implementation

Verifying Outputs

Upon running the forward and backward pass on x = [0.25, 0.75], we got the following results.

```
model.reset()
forwardpass_check = model.forwardPass(np.array([0.25, 0.75]).T)
forwardpass_check
```

```
[array([1.  , 0.25, 0.75]),
 array([1.        , 0.88079708, 0.11920292]),
 0.23840584404423526]
```

```
backwardpass_check = model.backwardPass(forwardpass_check, 0)
backwardpass_check
```

```
[array([[ 0.00250311, -0.00250311],
        [ 0.00062578, -0.00062578],
        [ 0.00187733, -0.00187733]]),
 array([-0.02384058, -0.02099872, -0.00284187])]
```

Figure 7: Outputs of the implementation

Comparing these outputs to the results of our calculations, we can see that, even though the code outputs have a higher accuracy, these values are approximately the same . Thus we confirm that our implantation of forward and backward pass is working as desired.

Training The Model

To train the model, the minimum training dataset mentioned earlier in this document was used. The model was trained for 200000 epochs. The number of epochs was determined through trial and error. To implement training I ran the forward pass, followed by the backward pass, then I updated the weights and repeated the process. This functions returns the final state of the weights.

I also coded up an option to train the model for a specific [x1, x2] values. I tried to use this method to reduce the number of epochs by trying to introduce biased data. Though this attempt wasn't successful, I decided to keep this function.

```
######################### Training Options ####################################
def trainManual(self, inputs, ans):
    prev = 100
    preds = []
    while True:
        outs = self.forwardPass(inputs)
        preds.append(outs[2])
        deltas = self.backwardPass(outs, ans)

        for j in range(0, len(self.weights)):
            self.weights[j] = self.weights[j] + deltas[j]

        if abs(prev - outs[2]) == 0:
            break
        prev = outs[2]
    return self.weights, preds

def train(self):
    for i in range(0, self.epochs):
        for ind in range(0, len(self.train_data)):
            outs = self.forwardPass(self.train_data[ind])
            deltas = self.backwardPass(outs, self.train_labels[ind])

            for j in range(0, len(self.weights)):
                self.weights[j] = self.weights[j] + deltas[j]

    return self.weights
```

Figure 8: Training methods

The model was trained using the train function and the final weights of the model can be seen in the screen shot below.

```
model.reset()
model.train()
```

```
[array([[-1.85185461,  0.19857669],
        [ 1.51386759, -5.75476816],
        [ 1.51002081, -5.56127349]]),
 array([-1.22711622,  2.91687027,  3.33300892])]
```

Figure 9: Final weights

Plots

To plot the model surface, a meshgrid of the range [0, 1] was created. Predictions for each point of the grid were obtained using the predict method of the model.

```
xx, yy = np.meshgrid(np.linspace(0, 1, 50), np.linspace(0, 1, 50), indexing="xy")
xy = np.dstack((xx, yy))
```

```
zz = np.ndarray((50 , 50))
for i in range(0, 50):
    for j in range(0, 50):
        zz[i][j] = model.predict(xy[i][j])
```

Figure 10: Generating the surface

Using these values the surface was plotted in a 2d as well as a 3d plot.

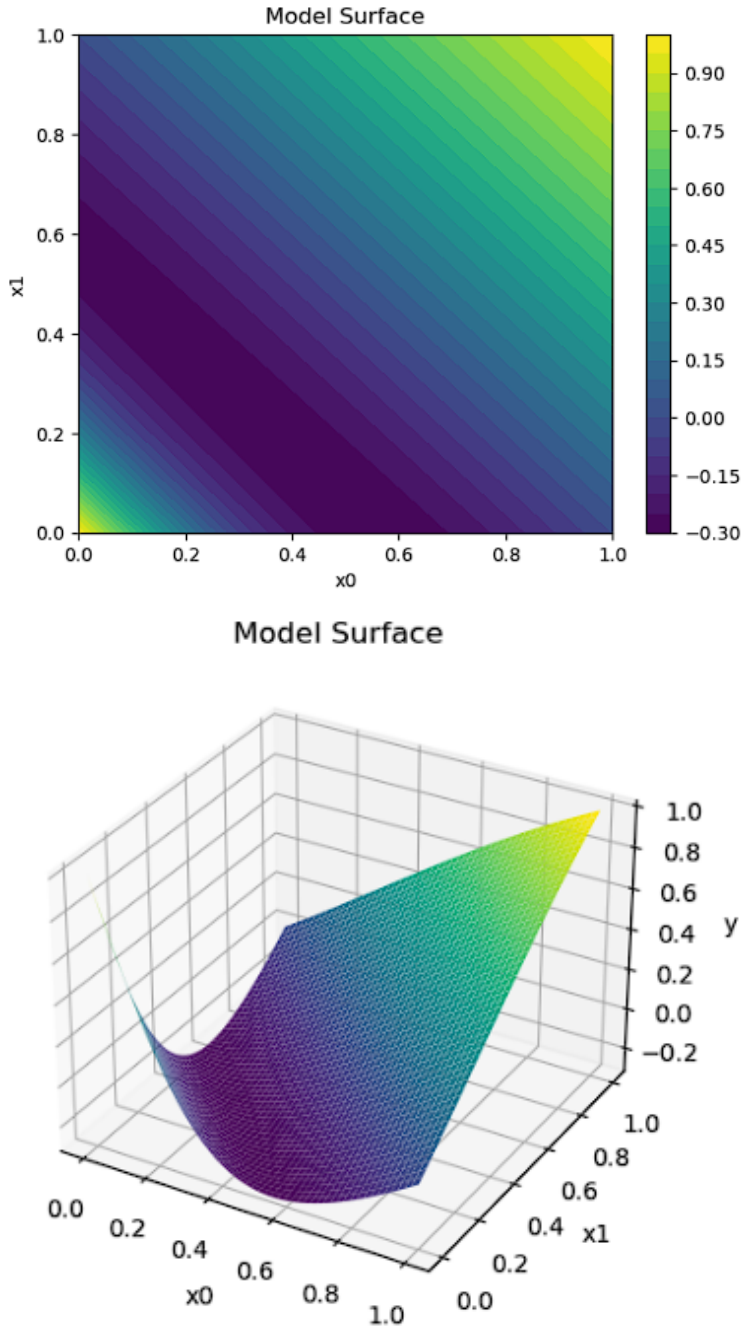


Figure 11: Model Surface

Validating the Model

As we can see the model surface does tend to 1 when the x = [0, 0] or x = [1, 1] and goes to 0 when x = [0, 1] or x = [1, 0]. I was expecting a symmetric surface which folds along the [0,1] - [1,0] diagonal, but the surface is curved steeply near the [0,0] point. It also predicts negative values in some regions. But, over the training data, the model predicts quite accurately. This can be confirmed by the test shown in the screen shot below.

```
tests = [np.array([1, 1]).T, np.array([1, 0]).T, np.array([0, 1]).T, np.array([0, 0]).T]
ans = [1, 0, 0, 1]
```

```
for inp in tests:
    print(model.predict(inp))
```

```
0.9999999999999984
1.1102230246251565e-15
1.5543122344752192e-15
1.0
```

Figure 12: Predictions over the training data set