Experiment 3: Implementation of Neural Network Backpropagation for XOR Gate Classification

Tanmay Rathod (23MAI007)

February 1, 2024

1 Aim

The aim of this project is to implement a neural network back-propagation model to perform classification for the XOR gate problem.

2 Theory

The XOR gate is a binary operation that outputs true only when the number of true inputs is odd. This problem is not linearly separable, making it a suitable test case for neural networks. The sigmoid activation function and backpropagation algorithm are used in this implementation.

The sigmoid activation function is defined as:

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

The sigmoid derivative is given by:

$$\sigma'(x) = \sigma(x) \cdot (1 - \sigma(x))$$

Forward Pass:

$$\begin{aligned}
a_1 &= X \cdot w_1 \\
z_1 &= \text{sigmoid}(a_1) \\
\text{bias} &= \text{np.ones}((\text{len}(X), 1)) \\
z_1 &= \text{np.concatenate}((\text{bias}, z_1), \text{axis} = 1) \\
a_2 &= z_1 \cdot w_2 \\
z_2 &= \text{sigmoid}(a_2)
\end{aligned}$$

Backpropagation:

$$\delta^2 = z^2 - Y$$
$$\Delta^2 = z^{1T} \cdot \delta^2$$
$$\delta^1 = (\delta^2 \cdot w_2[1:,:].T) \cdot \text{sigmoid\_derivative}(a_1)$$
$$\Delta^1 = X^T \cdot \delta^1$$

Weight Updates:

$$w_1 = w_1 - \text{lr} \cdot (1/m) \cdot \Delta^1$$
$$w_2 = w_2 - \text{lr} \cdot (1/m) \cdot \Delta^2$$

Where $\text{sigmoid}(x) = \frac{1}{1+e^{-x}}$, and $\text{sigmoid\_derivative}(x) = \text{sigmoid}(x) \cdot (1 - \text{sigmoid}(x))$.

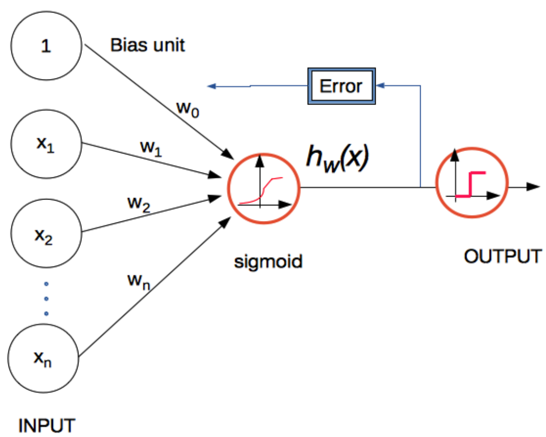

Figure 1: Back-Propagation Neural Network
Image source:
https://mashimo.wordpress.com/2015/09/13/back-propagation-for-neural-network/

The forward propagation algorithm calculates the output of the neural network. The backpropagation algorithm adjusts the weights of the network to minimize the error between the predicted and actual outputs.

3 Code

Below is the Python code implementing the neural network model.

```
import numpy as np
import matplotlib.pyplot as plt

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

def sigmoid_derivative(x):
    return sigmoid(x) * (1 - sigmoid(x))
def step_function(x) :
    if x  > 0.5 :
        return 1
    else :
        return 0
def forward(x, w1, w2, predict=False):
    a1 = np.dot(x, w1)
    z1 = sigmoid(a1)
    bias = np.ones((len(x), 1))
    z1 = np.concatenate((bias, z1), axis=1)
    a2 = np.dot(z1, w2)
    z2 = sigmoid(a2)
    if predict:
        return z2
    return a1, z1, a2, z2

def backprop(a2, X, z1, z2, y, w2, a1):
    delta2 = z2 - y
    Delta2 = np.dot(z1.T, delta2)
    delta1 = (delta2.dot(w2[1:, :].T)) * sigmoid_derivative(a1)
    Delta1 = np.dot(X.T, delta1)
    return delta2, delta1, Delta1, Delta2

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

Y = np.array([1, 1, 0, 0]).reshape(-1, 1)

w1 = np.random.randn(3, 5)
w2 = np.random.randn(6, 1)

lr = 0.1
```

```
costs = []
epochs = 2000
m = len(X)

for i in range(epochs):
    a1, z1, a2, z2 = forward(X, w1, w2)
    delta2, delta1, Delta1, Delta2 = backprop(a2, X, z1, z2, Y, w2, a1)
    w1 = w1 - lr * (1/m) * Delta1
    w2 = w2 - lr * (1/m) * Delta2
    c = np.mean(np.abs(delta2))
    costs.append(c)

predictions = forward(X, w1, w2, predict=True)
print("Predicted Output:")
for i in range(len(X)):
    print(f"Input: {X[i, 1:]}, Actual: {Y[i][0]}, Predicted Output: {predictions[i][0]} ~~ {
plt.plot(costs)
plt.title("Training Loss over Iterations")
plt.xlabel("Iterations")
plt.ylabel("Loss")
plt.show()

x_min, x_max = X[:, 1].min() - 0.1, X[:, 1].max() + 0.1
y_min, y_max = X[:, 2].min() - 0.1, X[:, 2].max() + 0.1
xx, yy = np.meshgrid(np.linspace(x_min, x_max, 100),
                     np.linspace(y_min, y_max, 100))

grid_data = np.c_[np.ones(xx.ravel().shape[0]), xx.ravel(), yy.ravel()]

Z = forward(grid_data, w1, w2, predict=True)
Z = Z.reshape(xx.shape)

plt.contourf(xx, yy, Z, cmap=plt.cm.RdBu, alpha=0.8)
plt.scatter(X[Y.ravel() == 0, 1], X[Y.ravel() == 0, 2], color='red', label='Class 0')
plt.scatter(X[Y.ravel() == 1, 1], X[Y.ravel() == 1, 2], color='green', label='Class 1')

plt.xlabel('Feature 1')
plt.ylabel('Feature 2')
plt.title('Classification with Neural Network (XOR Gate)')
plt.legend()
plt.show()
```

4 Results

The training loss over iterations is shown in Figure 3. The classification boundary of the neural network model is depicted in Figure **??**.

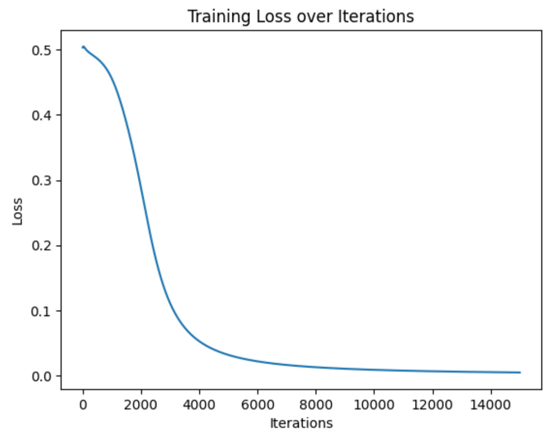

Figure 2: Training Loss over Iterations

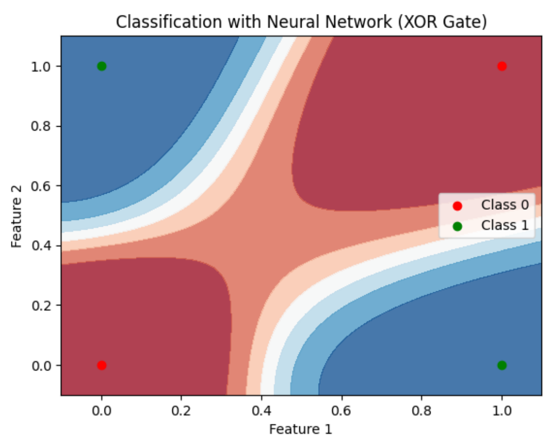

Figure 3: Classification with Sigmoid function

Final weights for $w1$:

$$\begin{bmatrix} -0.7719786 & -1.06872307 & 0.58092025 & -1.16993049 & -1.33351943 \\ 3.99620114 & -0.45337276 & -1.63226734 & -1.69568236 & -3.98596408 \\ 3.43489435 & 0.42070325 & 3.10199817 & 0.86284607 & 2.5147209 \end{bmatrix}$$

Final weights for $w2$:

$$\begin{bmatrix} -2.12135769 \\ 4.2332644 \\ -0.22921761 \\ -3.56937446 \\ 1.06821446 \\ 3.73097225 \end{bmatrix}$$

Delta values:

$$\text{Delta2: } \begin{bmatrix} -0.05306363 \\ -0.03766057 \\ 0.01583502 \\ 0.07078051 \end{bmatrix}$$

$$\text{Delta1: } \begin{bmatrix} -0.0069812 & 0.00242126 & 0.03424581 & -0.00314997 & -0.00028261 \\ -0.0076195 & 0.00248091 & 0.00044353 & -0.0177309 & -0.02437235 \\ 0.01669248 & -0.0008502 & -0.02079089 & 0.00595652 & 0.01373086 \\ 0.00012767 & -0.0042446 & -0.01921729 & 0.01261313 & 0.0140683 \end{bmatrix}$$

5 Conclusion

The neural network successfully learned to classify the XOR gate problem. Despite being non-linearly separable, the model was able to approximate the XOR function and achieve accurate classification.