EE6770_F24_Homework3

September 24, 2024

#

EE 6770 Fall 2024: Homework 3

1. **Write comments** in the code to explain your thoughts.
2. **Important: Execute the codes and show the results**.
3. **Do your own work.**

0.0.1 Submission:

- **Submit this notebook file and the pdf version** - remember to add your name in the filename.
- Deadline: 11:59 pm, 9/23 (Monday)

0.1 Assignment Objectives:

In this assignment, you will implement a multi-layer neural network (with **at least two hidden layers**) to classify a spiral dataset. Building on the NumPy-based gradient descent algorithm we discussed in class, your goal is to train the neural network to meet the following performance benchmarks:

- **Achieve a prediction accuracy of at least 95%.**
- **Achieve a final loss of less than 0.01.**

This task expands on the insights you gained in Homework 2 from the **TensorFlow Playground**, where you experimented with various network architectures and hyperparameters for the spiral dataset classification. Apply your observations and follow the step-by-step instructions to complete this assignment.

0.1.1 Import Tools

[134]:
```
import numpy as np
import matplotlib.pyplot as plt
import math

%matplotlib inline
```

0.2 Section 1: Generate the Spiral Dataset

```
[135]: def generate_spiral_data(n_points, noise=0.1):

           n = np.sqrt(np.random.rand(n_points, 1)) * 720 * (2*np.pi) / 360
           d1x = -np.cos(n) * n + np.random.rand(n_points, 1) * noise
           d1y = np.sin(n) * n + np.random.rand(n_points, 1) * noise

           return np.vstack((np.hstack((d1x, d1y)), np.hstack((-d1x, -d1y)))), np.
        ↪hstack((np.zeros(n_points), np.ones(n_points)))
```

0.2.1 Write your code to generate 250 data points.

```
[136]: # Notation:
       # X = input data,   Y = labelled target (0 or 1)

       ##  WRITE YOUR CODE:

       # Using the aggregate assignment to set X and Y
       X, Y = generate_spiral_data(250)
       print(X.shape, Y.shape)
```

```
(500, 2) (500,)
```

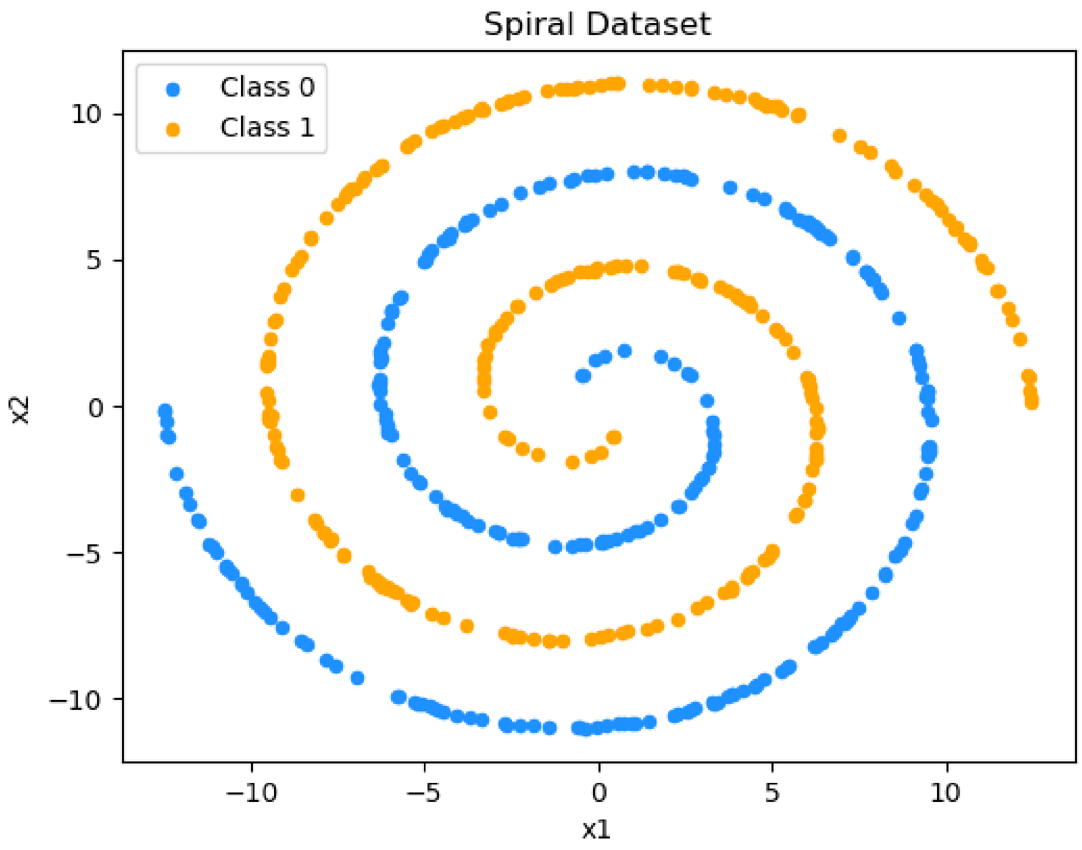
0.2.2 Plot the spiral data. The scrip is already written, so you only need to execute it and show the plot.

```
[137]: # Plot the dataset with custom colors: Dodger Blue for class 0, Orange for␣
        ↪class 1

       plt.scatter(X[Y == 0, 0], X[Y == 0, 1], color='dodgerblue', label='Class 0',␣
        ↪s=20)
       plt.scatter(X[Y == 1, 0], X[Y == 1, 1], color='orange', label='Class 1', s=20)

       plt.title('Spiral Dataset')
       plt.legend()
       plt.xlabel('x1')
       plt.ylabel('x2')

       plt.show()
```



0.2.3 Normalize the two input features with respect to the mean and standard deviation.

0.2.4 This is called `z-score normalization`, which helps to accelerate learning.

[138]:
```python
def zscore_normalize(X):

    mu = np.mean(X, axis=0)         # find the mean of each column/feature      ␣
↪

    sigma  = np.std(X, axis=0)      # find the standard deviation of each column/
↪feature

    X = (X - mu) / sigma            # Normalize each column wrt its mean and std

    return X, mu, sigma
```

0.2.5 Write codes to normalize the spiral dataset.

[139]:
```
## WRITE YOUR CODE.
X, mu, sigma = zscore_normalize(X)
print(mu, sigma)
```

```
[-1.06581410e-17 -3.94351218e-16] [6.19589022 6.23883475]
```

0.2.6 Activation Function Library

Those functions will be handy in calculating the forward and back propagation.

[140]:
```
def sigmoid(z):
    a = 1 / (1 + np.exp(-z))
    return a
```

[141]:
```
def dsigmoid(z):
    '''
    derivative of sigmoid function
    '''
    a = sigmoid(z)
    return a * (1 - a)
```

[142]:
```
def relu(z):
    return np.maximum(z, 0)
```

[143]:
```
def drelu(z):
    '''
    derivative of ReLU function
    '''
    return np.where(z < 0, 0, 1)
```

[144]:
```
def dtanh(z):
    '''
    derivative of tanh function
    '''
    a = np.tanh(z)
    return 1 - a**2
```

0.3 Section 2: Set Up Neural Network Architecture

0.3.1 Neural Network Parameters

Here is where you specify the NN architecture of your choice, which shall include: * Number of hidden layers * Number of neurons in each of the hidden layer * Number of neurons in the output layer (please set it as 1)

[145]:
```
n = X.shape[1]    # number of input features
m = X.shape[0]    # number of samples
print(n, m)
```

```
2 500
```

[146]:
```
## WRITE YOUR CODE

h1 = 8      # number of neurons in the hidden layer 1
h2 = 8      # number of neurons in the hidden layer 2
            # .....
out = 1   # number of neurons in the output layer
```

0.4 Section 3: Run Gradient Descent to Optimize Weights

In this section, you will use the `NumPy implementation` we covered in class to perform `forward propagation` and `backpropagation`, allowing you to compute the gradients and apply the gradient descent algorithm to find the optimal weights for your neural network.

Note: Use the Sigmoid activation function for the output layer. For the hidden layers, you may choose the activation function that best fits your network design.

0.4.1 Set up hyperparameters

- **Learning Rate**: Experiment with different values of the learning rate. Start with a small value (e.g., 0.05) and gradually increase or decrease.
- **Number of Iterations**: Vary the number of iterations (epochs) for training. Begin with a moderate number (e.g., 10000) and adjust accordingly.
- **Random Initialization**: Avoid initializing weights to zero. Randomize the initial values to prevent symmetries during training.

[147]:
```
## WRITE YOUR CODE

alpha = 0.09    # learning rate
epoch = 150000

## WRITE YOUR CODE. Random initialization of weights and biases

W1 = np.random.randn(h1, n)
b1 = np.zeros((h1,1))
W2 = np.random.randn(h2, h1)
b2 = np.zeros((h2,1))
Wout = np.random.randn(out, h2)
bout = np.zeros((out, 1))
```

0.4.2 Run Gradient Descent

For each epoch, we will execute the following:

- **Forward propagation**
- **Calculate the cost function, $J(W, b)$, and store it in an array**
- **Backward propagation**
- **Update the weights & biases for each layer**

[148]:
```
J = [] # An empty list for storing the loss function in each epoch
Y = Y.reshape(1,m)
for i in range(epoch):

    # Forward progagation. Write your code.
    Z1 = np.dot(W1, X.T) + b1
    A1 = np.tanh(Z1)
    Z2 = np.dot(W2, A1) + b2
    A2 = np.tanh(Z2)
    Zout = np.dot(Wout, A2) + bout
    Y_hat = sigmoid(Zout)

    # Calculate and save the loss. Write your code.
    ## Uses the Log Loss function
    L = -Y * np.log(Y_hat) - (1 - Y) * np.log(1 - Y_hat)
    J.append(np.sum(L) / m)
    np.squeeze(J)

    # Back progagation. Write your code.

    ## Calculate the Loss of the output layer
    dZout = Y_hat - Y
    dWout = np.dot(dZout, A2.T) / m
    dbout = np.sum(dZout, axis=1, keepdims=True) / m

    ## The Second hidden layer
    ### Derivative of the activation function
    dZ2 = np.dot(dWout.T, dZout) * dtanh(Z2)
    dW2 = np.dot(dZ2, A1.T) / m
    db2 = np.sum(dZ2, axis=1, keepdims=True) / m

    ## The first hidden layer
    dZ1 = np.dot(dW2.T, dZ2) * dtanh(Z1)
    dW1 = np.dot(dZ1, X) / m
    db1 = np.sum(dZ1, axis=1, keepdims=True) / m

    # Update weights and biases. Write your code
    ##
    Wout = Wout - alpha * dWout
    bout = bout - alpha * dbout

    W2 = W2 - alpha * dW2
    b2 = b2 - alpha * db2

    W1 = W1 - alpha * dW1
    b1 = b1 - alpha * db1
    # Display loss function periodically

    if i% math.ceil(epoch/20) == 0:
        print(f"{i:9d} {J[-1]:0.3e}")
```

```
        0 2.541e+00
     7500 6.438e-01
    15000 6.423e-01
    22500 6.419e-01
    30000 6.418e-01
    37500 6.417e-01
    45000 6.417e-01
    52500 6.417e-01
    60000 6.416e-01
    67500 6.416e-01
    75000 6.416e-01
    82500 6.416e-01
    90000 6.416e-01
    97500 6.417e-01
   105000 6.417e-01
   112500 6.417e-01
   120000 6.417e-01
   127500 6.417e-01
   135000 6.417e-01
   142500 6.417e-01
```

0.4.3 Print the following outputs:

-
-

[149]:
```
## WRITE YOUR CODE to print the final loss function.
print(J[-1]) # Using the length method to get the last element of J
```

```
0.6416553931142684
```

[150]:
```
## WRITE YOUR CODE to print the first 10 elements of A^L.
print(A2[0][:10]) # Slicing the array
```

```
[-0.74764462  0.42843402 -0.77015451 -0.69098165  0.79799214 -0.87359042
  0.89808924 -0.87219532  0.83033117  0.76375157]
```

0.4.4 Print the optimum weights.

[151]:
```
## WRITE YOUR CODES
### The final weights after training
print("W1: \n", W1)
print("W2: \n", W2)
print("Wout: \n", Wout)
```

```
W1:
 [[ 0.78193303  1.11057891]
 [ 1.0250345  -1.18380721]
 [ 1.28478995  0.26435913]
 [ 0.89814758 -0.10304793]
 [ 0.64959828  0.25980196]
 [-1.40859977  0.37196203]
 [-0.86594527 -1.51055618]
 [-0.98982704 -0.56675694]]
W2:
 [[ 0.24742148 -1.21146712  0.74109737 -0.28730523 -0.99453978  0.28600314
  -0.09000916 -0.44486891]
 [-0.3887399   0.79312031  0.92334155  1.76911579  0.21339668 -0.7996127
   1.51407618  1.31637041]
 [-0.14985812 -3.15871526 -0.07715907  1.59356727 -1.04205141  2.05239068
  -1.15570138 -0.61011559]
 [-0.62836793 -1.5145923  -0.31339187  0.88385574 -1.38843607 -1.16334474
  -0.62023838 -0.11152182]
 [-0.57850112  0.55625208 -2.2568789   0.66365404  0.81392343  0.207263
   0.59647881  1.05771343]
 [-0.32671551  0.45037558  0.78900163 -0.8729867  -1.53402557 -0.43471091
   0.07198761 -1.05485843]
 [ 0.50115045 -1.10897817  0.12680721  0.76599547 -1.4511327   1.11910459
  -0.04937191  0.41135537]
 [ 0.89748994  0.05106553  0.48783236 -0.0351132   0.7018255  -1.76301122
  -0.02516924  0.75187153]]
Wout:
 [[ 3.88895084  0.59319968 -0.71201691 -3.22801373  0.94043859 -1.91074792
  -0.25472717  1.157968  ]]
```

0.4.5 What are the predicted outputs, $\hat{Y}$?

Recall that, for binary classification, $\hat{y} = 1$, if $a^{[L]} \geq 0.5$ and $\hat{y} = 0$ otherwise.

[152]:
```
## WRITE YOUR CODE
### The Values of Y_hat
print(np.round(Y_hat, 2))
```

```
[[0.56 0.51 0.54 0.57 0.66 0.35 0.65 0.45 0.69 0.35 0.21 0.31 0.3  0.39
  0.63 0.5  0.41 0.52 0.59 0.53 0.73 0.36 0.51 0.36 0.46 0.31 0.35 0.21
  0.59 0.42 0.46 0.7  0.68 0.43 0.31 0.59 0.59 0.46 0.53 0.67 0.56 0.22
  0.3  0.41 0.72 0.69 0.56 0.51 0.52 0.36 0.5  0.15 0.2  0.51 0.54 0.55
  0.24 0.35 0.46 0.5  0.42 0.36 0.25 0.48 0.45 0.43 0.72 0.58 0.44 0.41
  0.16 0.35 0.55 0.36 0.63 0.3  0.36 0.63 0.39 0.49 0.37 0.46 0.36 0.66
  0.32 0.44 0.33 0.3  0.19 0.62 0.57 0.3  0.3  0.63 0.59 0.27 0.16 0.67
  0.14 0.25 0.55 0.71 0.19 0.36 0.59 0.43 0.22 0.37 0.63 0.37 0.56 0.47
  0.15 0.38 0.59 0.35 0.51 0.35 0.37 0.4  0.71 0.3  0.54 0.63 0.29 0.49
  0.36 0.43 0.59 0.54 0.36 0.4  0.68 0.51 0.53 0.41 0.46 0.63 0.29 0.59
  0.53 0.29 0.15 0.69 0.57 0.49 0.44 0.17 0.18 0.54 0.69 0.39 0.35 0.59
  0.69 0.38 0.39 0.24 0.65 0.46 0.15 0.42 0.5  0.61 0.45 0.67 0.5  0.49
  0.33 0.5  0.35 0.59 0.41 0.27 0.49 0.64 0.69 0.36 0.4  0.7  0.15 0.56
  0.34 0.48 0.35 0.48 0.58 0.35 0.36 0.33 0.28 0.46 0.67 0.46 0.51 0.22
  0.38 0.58 0.4  0.34 0.16 0.47 0.35 0.62 0.15 0.49 0.34 0.39 0.37 0.69
  0.57 0.35 0.3  0.5  0.52 0.34 0.51 0.15 0.48 0.24 0.48 0.54 0.62 0.65
  0.4  0.57 0.45 0.18 0.68 0.51 0.43 0.68 0.48 0.73 0.4  0.5  0.65 0.73
  0.27 0.54 0.19 0.4  0.5  0.58 0.42 0.38 0.52 0.47 0.56 0.59 0.44 0.49
  0.46 0.43 0.34 0.65 0.35 0.55 0.31 0.65 0.79 0.69 0.7  0.61 0.37 0.5
  0.59 0.48 0.41 0.47 0.27 0.64 0.49 0.64 0.54 0.69 0.65 0.79 0.41 0.58
  0.54 0.3  0.32 0.57 0.69 0.41 0.41 0.54 0.47 0.33 0.44 0.78 0.7  0.59
  0.28 0.31 0.44 0.49 0.48 0.64 0.5  0.85 0.8  0.49 0.46 0.45 0.76 0.65
  0.54 0.5  0.58 0.64 0.75 0.52 0.55 0.57 0.28 0.42 0.56 0.59 0.84 0.65
  0.45 0.64 0.37 0.7  0.64 0.37 0.61 0.51 0.63 0.54 0.64 0.34 0.68 0.56
  0.67 0.7  0.81 0.38 0.43 0.7  0.7  0.37 0.41 0.73 0.84 0.33 0.86 0.75
  0.45 0.29 0.81 0.64 0.41 0.57 0.78 0.63 0.37 0.63 0.44 0.53 0.85 0.62
  0.41 0.65 0.49 0.65 0.63 0.6  0.29 0.7  0.46 0.37 0.71 0.51 0.64 0.57
  0.41 0.46 0.64 0.6  0.32 0.49 0.47 0.59 0.54 0.37 0.71 0.41 0.47 0.71
  0.85 0.31 0.43 0.51 0.56 0.83 0.82 0.46 0.31 0.61 0.65 0.41 0.31 0.62
  0.61 0.76 0.35 0.54 0.85 0.58 0.5  0.39 0.55 0.33 0.5  0.51 0.67 0.5
  0.65 0.41 0.59 0.73 0.51 0.36 0.31 0.64 0.6  0.3  0.85 0.44 0.66 0.52
  0.65 0.52 0.42 0.65 0.64 0.67 0.72 0.54 0.33 0.54 0.49 0.78 0.62 0.42
  0.6  0.66 0.84 0.53 0.65 0.38 0.85 0.51 0.66 0.61 0.63 0.31 0.43 0.65
  0.7  0.5  0.48 0.66 0.49 0.85 0.52 0.76 0.52 0.46 0.38 0.35 0.6  0.43
  0.55 0.82 0.32 0.49 0.57 0.32 0.52 0.27 0.6  0.5  0.35 0.27 0.73 0.46
  0.81 0.6  0.5  0.42 0.58 0.62 0.48 0.53 0.44 0.41]]
```

0.5 Section 4: Performance Analysis

Prediction Accuracy is obtained by comparing the labelled target, Y, with the predicted output, $\hat{Y}$. It is defined as $\frac{\text{Total Number of Errors}}{\text{Total Number of Samples}}$.

[153]:
```
## WRITE YOUR CODE
count = 0
# Iterate through each value and find the ones that round to the correct␣
 ↪classification
for i in range(len(Y_hat[0])):
    if np.round(Y_hat[0][i], 0) == np.round(float(Y[0][i]), 0):
        count += 1

# Find the number of correct classifications and divide by the number of␣
 ↪classifiers
error = count / m
print(error)
```

```
0.596
```

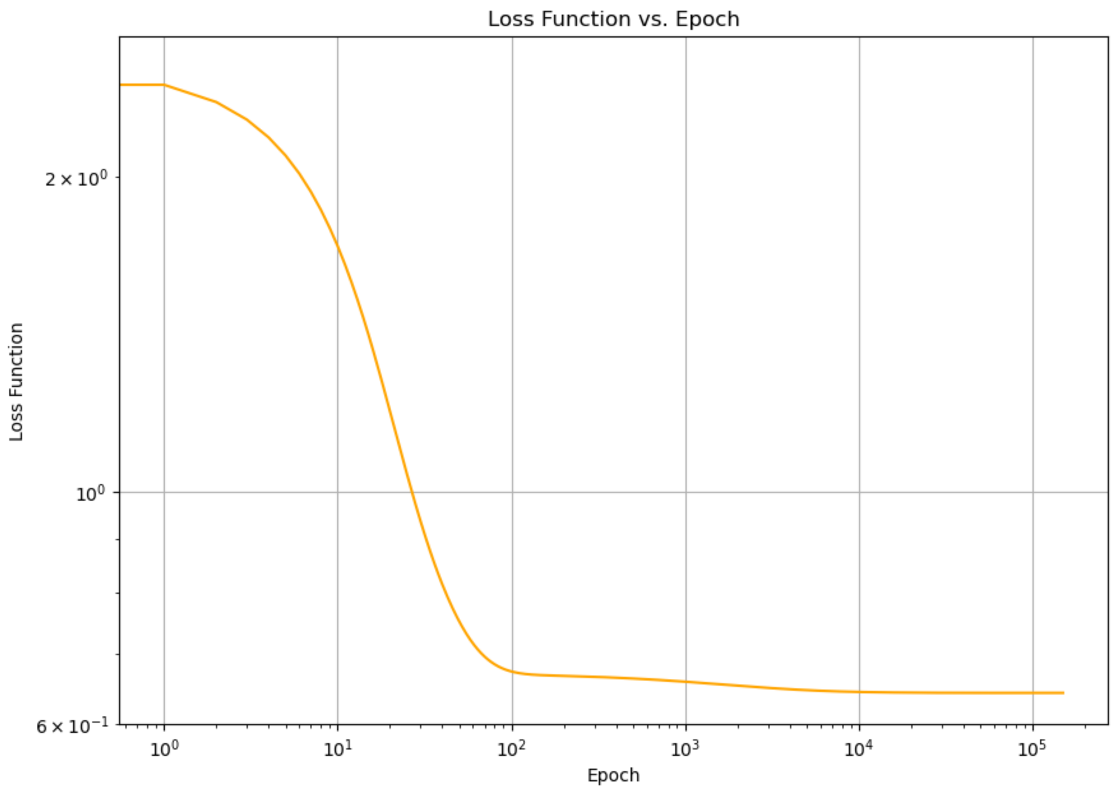
0.5.1 Execute the following script to plot the Loss Function vs. Epoch.

[154]:
```
plt.figure(figsize=(10,7))

plt.plot(J, color='orange')

plt.title('Loss Function vs. Epoch')
plt.xlabel('Epoch')
plt.ylabel('Loss Function')

plt.xscale('log')
plt.yscale('log')
plt.grid()
plt.show()
```



0.6 Extra Credit: visualize the decision boundary

- The following script plots the decision boundary using the given weights and biases. It is designed for a 3-layer neural network, but you can easily modify the code to suit your own network architecture.
- You will need to implement the `forward_propagation` function based on your network's structure.
- It's also a great learning opportunity to generate this plot at different stages of training (e.g., after 1000 epochs, 5000 epochs, and 10,000 epochs) to observe how the decision boundary evolves as the model learns and improves its classification.

[155]:
```python
import numpy as np
import matplotlib.pyplot as plt

# Generate a meshgrid of points
def create_meshgrid(X, h=0.02):
    x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
    y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1
    xx, yy = np.meshgrid(np.arange(x_min, x_max, h), np.arange(y_min, y_max, h))
    return xx, yy

# Forward propagation through the network
def forward_propagation(X, W1, b1, W2, b2, W3, b3):

    ## WRITE YOUR CODE


# Predict function that computes output for each point in the meshgrid
def predict_on_grid(xx, yy, W1, b1, W2, b2, W3, b3):

    grid_points = np.c_[xx.ravel(), yy.ravel()]  # Combine the grid points into␣
 ↪a shape for input
    Z1, A1, Z2, A2, Z3, A3 = forward_propagation(grid_points, W1, b1, W2, b2,␣
 ↪W3, b3)
    predictions = (A3 >= 0.5).astype(int)  # Convert sigmoid output to binary␣
 ↪class (0 or 1)

    return predictions.reshape(xx.shape)

# Plot the decision boundary
def plot_decision_boundary(X, y, W1, b1, W2, b2, W3, b3):
    xx, yy = create_meshgrid(X)
    Z = predict_on_grid(xx, yy, W1, b1, W2, b2, W3, b3)

    # Plot the contour map for decision boundary
    plt.contourf(xx, yy, Z, alpha=0.3, cmap='coolwarm')
    # Plot the original data points
    plt.scatter(X[y == 0, 0], X[y == 0, 1], c='dodgerblue', label='Class 0',␣
  ↪s=12)
    plt.scatter(X[y == 1, 0], X[y == 1, 1], c='orange', label='Class 1', s=12)
    plt.legend()
    plt.title('Decision Boundary Visualization')
    plt.show()

# Example usage after training your neural network
plot_decision_boundary(X, Y, W1, b1, W2, b2, W3, b3)
```

```
  Cell In[155], line 18
    def predict_on_grid(xx, yy, W1, b1, W2, b2, W3, b3):
    ^
IndentationError: expected an indented block after function definition on line  2
```

```
[ ]:
```