(COM4523-B) ARTIFICIAL NEURAL NETWORKS

CODE EXPLANATIONS

Prepared By

Selim Koç

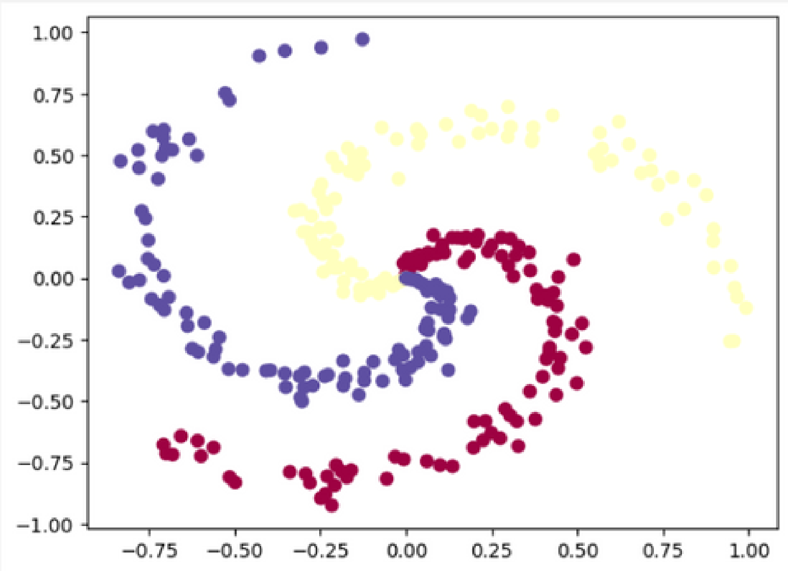
19290317

Softmax Linear Classifier

Firstly we create our dataset for the model

```
N=100
D=2
K=3
```
[339] 0.0s

```
X = np.zeros((N * K, D))
y = np.zeros((N * K), dtype="uint8")

for j in range(K):
    ix = range(N * j, N * (j + 1))
    r = np.linspace(0.0, 1, N)
    t = np.linspace(j * 4, (j + 1) * 4, N) + np.random.randn(N) * 0.2
    X[ix] = np.c_[r * np.sin(t), r * np.cos(t)]
    y[ix] = j

plt.scatter(X[:, 0], X[:, 1], c=y, s=40, cmap=plt.cm.Spectral)
plt.show()
```
[340] 0.0s



Let's examine the content and shape of our data.

```
X[:10],X.shape
```
[385] 0.0s

```
(array([[-0.        ,  0.        ],
        [-0.00260007,  0.00976064],
        [ 0.00660869,  0.01909049],
        [ 0.00616457,  0.02966937],
        [ 0.01154256,  0.03872022],
        [ 0.00394193,  0.05035098],
        [-0.00785092,  0.0600954 ],
        [ 0.04417696,  0.05520766],
        [ 0.00429644,  0.08069378],
        [ 0.02709865,  0.0867763 ]]),
 (300, 2))
```

```
y,y.shape
```

[386] ✓ 0.0s

```
(array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
        1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
        1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
        1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
        1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
        1, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
        2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
        2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
        2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
        2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2], dtype=uint8),
 (300,))
```

Define a function to separate the data into train ,validate and test sets

```
def random_split_data(X, y, train_percentage=0.7, val_percentage=0.15, test_percentage=0.15):
    """
    Randomly splits the data into training, validation, and test sets.

    Parameters:
    - X: Feature matrix
    - y: Labels
    - train_percentage: Percentage of data for training (default is 0.7)
    - val_percentage: Percentage of data for validation (default is 0.15)
    - test_percentage: Percentage of data for testing (default is 0.15)

    Returns:
    - X_train, y_train: Training set
    - X_val, y_val: Validation set
    - X_test, y_test: Test set
    """

    # Check if percentages sum up to 1
    total_percentage = train_percentage + val_percentage + test_percentage
    if total_percentage != 1.0:
        raise ValueError("Percentages should sum up to 1.0")

    # Get the total number of examples
    total_examples = len(X)

    # Shuffle the data
    indices = np.arange(total_examples)
    np.random.shuffle(indices)

    # Split data into training, validation, and test sets
    train_size = int(train_percentage * total_examples)
    val_size = int(val_percentage * total_examples)

    train_indices = indices[:train_size]
    remaining_indices = indices[train_size:]

    val_indices = remaining_indices[:val_size]
    test_indices = remaining_indices[val_size:]

    # Create sets
    X_train, y_train = X[train_indices], y[train_indices]
    X_val, y_val = X[val_indices], y[val_indices]
    X_test, y_test = X[test_indices], y[test_indices]

    return X_train, y_train, X_val, y_val, X_test, y_test
```

[338] ✓ 0.0s

Seperate the data using “random_split_data” function

```
X_train, y_train, X_val, y_val, X_test, y_test = random_split_data(X, y)
```

[342] ✓ 0.0s

Initialize the weight matrix W_exp with random values from a standard normal distribution

```
W_exp=np.random.randn(D, K)
W_exp   #W_exp matrix has dimensions D (number of features) by K (number of classes)
```

[343] ✓ 0.0s

```
array([[ 0.19054612, -0.65011432, -0.12252903],
       [ 0.66658672,  0.53478674, -0.42044854]])
```

We use a small scaling factor (0.01) to initialize the weights (W) in order to prevent issues like the gradients becoming too high during training

```
W = 0.01 * W_exp
W
```

[344] ✓ 0.0s

```
array([[ 0.00190546, -0.00650114, -0.00122529],
       [ 0.00666587,  0.00534787, -0.00420449]])
```

Initialize the bias vector b with zeros.

```
b = np.zeros((1, K))
b   # The vector has dimensions 1 by K, where K is the number of classes.
```

[345] ✓ 0.0s

```
array([[0., 0., 0.]])
```

Calculate the scores for each class by performing a dot product between the input features (X) and the weight matrix (W),and then adding the bias vector (b).

```
scores = np.dot(X_train, W) + b
scores[:10],scores.shape
```

[346] ✓ 0.0s

```
(array([[-4.80355658e-06, -7.04177790e-04,  9.86941533e-07],
        [ 1.86070408e-03,  3.53705407e-03, -1.16767298e-03],
        [-6.10878849e-04, -3.97662089e-03,  3.75140443e-04],
        [-8.33088978e-04, -1.65696371e-03,  5.22585779e-04],
        [ 1.86949002e-04,  2.10739073e-03, -1.12207903e-04],
        [-2.39780453e-03, -2.02744055e-03,  1.51210888e-03],
        [ 3.04645074e-03, -3.66833765e-03, -1.93937448e-03],
        [-2.83402249e-04,  3.08273823e-04,  1.80318032e-04],
        [ 1.35025703e-03,  4.58403652e-06, -8.54819164e-04],
        [-1.40804762e-03,  5.59862021e-03,  9.07750672e-04]]),
 (210, 3))
```

We create a variable to use in loops later in the code which contains the number of rows of our input data

```
num_examples = X_train.shape[0]   # This variable is created to be used in loops later in the code.
num_examples
[347]  ✓ 0.0s
...  210
```

Calculate the scores corresponding to the correct classes for each example.

```
correct_class_scores = scores[range(num_examples), y_train]
correct_class_scores[:10],correct_class_scores[-10:]
[348]  ✓ 0.0s
...  (array([ 9.86941533e-07,  3.53705407e-03, -6.10878849e-04,  5.22585779e-04,
             2.10739073e-03,  1.51210888e-03, -3.66833765e-03,  3.08273823e-04,
             1.35025703e-03,  9.07750672e-04]),
      array([ 0.00102005, -0.00131674, -0.00710276,  0.00319932,  0.00197148,
              0.00325313, -0.00559717, -0.00669607,  0.00217245,  0.00258924]))
```

Calculate the exponential of the scores to obtain unnormalized probabilities.

```
exp_scores = np.exp(scores)
exp_scores[:10],exp_scores[-10:]
[349]  ✓ 0.0s
...  (array([[0.9999952 , 0.99929607, 1.00000099],
             [1.00186244, 1.00354332, 0.99883301],
             [0.99938931, 0.99603128, 1.00037521],
             [0.99916726, 0.99834441, 1.00052272],
             [1.00018697, 1.00210961, 0.9998878 ],
             [0.99760507, 0.99797461, 1.00151325],
             [1.0030511 , 0.99633838, 0.9980625 ],
             [0.99971664, 1.00030832, 1.00018033],
             [1.00135117, 1.00000458, 0.99914555],
             [0.99859294, 1.00561432, 1.00090816]]),
      array([[1.00102057, 0.99997735, 0.99935436],
             [1.00488224, 0.99868413, 0.99691755],
             [1.00108201, 0.99292241, 0.99929489],
             [1.00372097, 1.00320444, 0.99766074],
             [0.99690236, 1.00252489, 1.00197343],
             [1.00282522, 1.00325842, 0.99822498],
             [0.99441846, 0.99702854, 1.00354108],
             [1.00132942, 0.99332629, 0.99913976],
             [0.99657176, 0.99943059, 1.00217481],
             [1.00047676, 1.0025926 , 0.99970584]]))
```

Calculate the normalized probabilities by dividing the exponential scores by the sum of exponential scores for each example.

```
probs = exp_scores / np.sum(exp_scores, axis=1, keepdims=True)
probs[:10]
[350]  ✓ 0.0s
...  array([[0.33341039, 0.33317729, 0.33341232],
            [0.33348296, 0.33404246, 0.33247458],
            [0.33359727, 0.33247636, 0.33392637],
            [0.33327412, 0.33299965, 0.33372623],
            [0.33315308, 0.33379349, 0.33305343],
            [0.33285757, 0.33298087, 0.33416156],
            [0.33463458, 0.33239511, 0.33297031],
            [0.33321608, 0.33341329, 0.33337063],
            [0.33372796, 0.33327917, 0.33299287],
            [0.3322977 , 0.33463417, 0.33306813]])
```

Calculate the sum of probabilities for each example, ensuring that the total probability for each example sums to 1

```
probs_val = np.sum(probs,axis=1,keepdims=True)
probs_val[:10]
```

[351] ✓ 0.0s

```
array([[1.],
       [1.],
       [1.],
       [1.],
       [1.],
       [1.],
       [1.],
       [1.],
       [1.],
       [1.]])
```

Calculate the cross-entropy loss by summing the negative logarithm of the predicted probabilities for the correct classes.

```
loss = -np.sum(np.log(probs[range(num_examples), y_train])) / num_examples
loss
```

[352] ✓ 0.0s

```
1.098683179274858
```

This demonstrates that even if all values are randomly assigned, the worst-case loss can be determined by log2 of the number of classes.

```
import math
result = math.log2(3)
result
```

[353] ✓ 0.0s

```
1.584962500721156
```

Create a copy of the probabilities array to represent the gradient of the scores with respect to the loss.

```
dscores = probs.copy()
dscores[range(num_examples), y_train] -= 1    #Subtracting 1 from the probabilities of correct classes inc
dscores
```

[354] ✓ 0.0s

```
array([[ 0.33341039,  0.33317729, -0.66658768],
       [ 0.33348296, -0.66595754,  0.33247458],
       [-0.66640273,  0.33247636,  0.33392637],
       [ 0.33327412,  0.33299965, -0.66627377],
       [ 0.33315308, -0.66620651,  0.33305343],
       [ 0.33285757,  0.33298087, -0.66583844],
       [ 0.33463458, -0.66760489,  0.33297031],
       [ 0.33321608, -0.66658671,  0.33337063],
       [-0.66627204,  0.33327917,  0.33299287],
       [ 0.3322977 ,  0.33463417, -0.66693187],
       [ 0.33417994, -0.66619912,  0.33201917],
       [ 0.33421139, -0.66835249,  0.3341411 ],
       [ 0.33465896, -0.66648381,  0.33182485],
       [-0.66609129,  0.33266172,  0.33342957],
       [ 0.3329997 ,  0.33293455, -0.66593424],
       [-0.66646164,  0.33331936,  0.33314227],
       [-0.66618765,  0.33299132,  0.33319633],
```

Calculate the gradient of the weights dW :
Multiply these probabilities (dscores) with the transpose of our input data x, we would essentially be applying a weighted sum to each row of x based on the calculated probabilities for each class.
Each row corresponds to a class, and the columns represent the weighted sum of the input features based on the calculated probabilities for that class.

Calculate the gradient of the bias terms db:
Summing the probabilities (dscores) along the rows (axis=0) provides the contribution of each class to the gradient of the bias terms. Each element in the resulting array corresponds to the sum of probabilities for the respective class across all input samples.

```
dW = np.dot(X_train.T, dscores)
db = np.sum(dscores, axis=0, keepdims=True)
```
[355] ✓ 0.0s

```
dW
```
[356] ✓ 0.0s

```
array([[ -9.09618303, -10.36651998,  19.46270301],
       [ 16.18079783, -19.53639229,   3.35559446]])
```

```
db
```
[357] ✓ 0.0s

```
array([[ 1.98814908,  1.0052981 , -2.99344718]])
```

Update the weights (W) and biases (b) using the learning rate (alpha)

```
alpha = 0.01
W -= alpha * dW
b -= alpha * db
```
[358] ✓ 0.0s

+ Code

New weights and biases

```
W,b
```
[359] ✓ 0.0s

```
(array([[ 0.09286729,  0.09716406, -0.19585232],
        [-0.15514211,  0.20071179, -0.03776043]]),
 array([[-0.01988149, -0.01005298,  0.02993447]]))
```

We create a function for traning our model

```python
def trainModel(X_train, y_train, X_val, y_val, W, b, epochs=400, alpha=0.01):
    """
    Train a model using gradient descent.

    Parameters:
    - X_train: Training feature matrix
    - y_train: Training true labels
    - X_val: Validation feature matrix
    - y_val: Validation true labels
    - W: Model weights
    - b: Model bias
    - epochs: Number of training epochs (default is 1000)
    - alpha: Learning rate (default is 0.01)

    Returns:
    - W: Updated weights after training
    - b: Updated bias after training
    - train_losses: List of training losses for each epoch
    - val_losses: List of validation losses for each epoch
    """
    # Lists to store losses for plotting.We can use these to make sure we don't overfit the training data.
    train_losses = []
    val_losses = []

    # Training loop
    for epoch in range(epochs):
        # Forward pass on the training set
        scores_train = np.dot(X_train, W) + b
        exp_scores_train = np.exp(scores_train)
        probs_train = exp_scores_train / np.sum(exp_scores_train, axis=1, keepdims=True)
        loss_train = -np.sum(np.log(probs_train[range(len(y_train)), y_train])) / len(y_train)

        # Backward pass on the training set
        dscores_train = probs_train.copy()
        dscores_train[range(len(y_train)), y_train] -= 1
        dW_train = np.dot(X_train.T, dscores_train)
        db_train = np.sum(dscores_train, axis=0, keepdims=True)

        # Update parameters for the training set
        W -= alpha * dW_train
        b -= alpha * db_train

        # Store the training loss for plotting
        train_losses.append(loss_train)

        # Forward pass on the validation set
        scores_val = np.dot(X_val, W) + b
        exp_scores_val = np.exp(scores_val)
        probs_val = exp_scores_val / np.sum(exp_scores_val, axis=1, keepdims=True)
        loss_val = -np.sum(np.log(probs_val[range(len(y_val)), y_val])) / len(y_val)

        # Store the validation loss for plotting
        val_losses.append(loss_val)

    return W, b, train_losses, val_losses
```

[360] ✓ 0.0s

Update the weights (W) and biases (b) using the function .We use both traind and validation data because we try to see our model gonna be overfit or underfit .We change te epoch value 1000 to 400 beacuse we easily see there is no change on the line on loss function and also our data is so small for 1000.

```python
WTrained, bTrained, train_losses, val_losses = trainModel(X_train, y_train, X_val, y_val, W, b, epochs=1000, alpha=0.01)
train_losses[:10], val_losses[:10]
```

[372] ✓ 0.1s

```
([0.7758220992972699,
  0.7758220992972699,
  0.7758220992972699,
  0.7758220992972699,
  0.7758220992972699,
  0.7758220992972699,
  0.7758220992972699,
  0.7758220992972699,
  0.7758220992972699,
  0.7758220992972699],
 [0.7081532989720446,
  0.7081532989720446,
  0.7081532989720446,
  0.7081532989720446,
  0.7081532989720446,
  0.7081532989720446,
  0.7081532989720446,
  0.7081532989720446,
  0.7081532989720446,
  0.7081532989720446])
```

Checking accuracy using the model with the test set and updated weights and biases.

```
def AccuracyCheck(X, y, W, b):
    """
    Evaluate the accuracy of the model on a given dataset.

    Parameters:
    - X: Feature matrix
    - y: True labels
    - W: Weights
    - b: Bias

    Returns:
    - accuracy: Accuracy of the model on the given dataset
    """
    scores = np.dot(X, W) + b
    # Determine predicted classes (class with the highest probability)
    predicted_classes = np.argmax(scores, axis=1)
    # Count the number of correct predictions
    correct_prediction_count = np.sum(predicted_classes == y)
    # Calculate the overall accuracy rate
    accuracy = correct_prediction_count / X.shape[0]
    return accuracy
```

[362] ✓ 0.0s

```
accuaryOnTest = AccuracyCheck(X_test, y_test, WTrained,bTrained)
accuaryOnTest
```

[370] ✓ 0.0s

```
0.6
```

We also made a functions for showing decision boundaries and our datas

```
def plot_decision_boundary(X, y, W, b):
    """
    Plot the decision boundary of a trained model along with the data points.

    Parameters:
    - X: Feature matrix
    - y: True labels
    - W: Model weights
    - b: Model bias
    """
    # Create a meshgrid of points
    h = 0.02  # step size in the mesh
    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))

    # Predict the class labels for each point in the meshgrid
    Z = np.dot(np.c_[xx.ravel(), yy.ravel()], W) + b
    Z = np.argmax(Z, axis=1)
    Z = Z.reshape(xx.shape)

    # Plot the decision boundary
    plt.contourf(xx, yy, Z, cmap=plt.cm.Spectral, alpha=0.8)
    plt.scatter(X[:, 0], X[:, 1], c=y, s=40, cmap=plt.cm.Spectral, edgecolors='k')
    plt.xlabel('Feature 1')
    plt.ylabel('Feature 2')
    plt.title('Decision Boundary')
```

[364] ✓ 0.0s

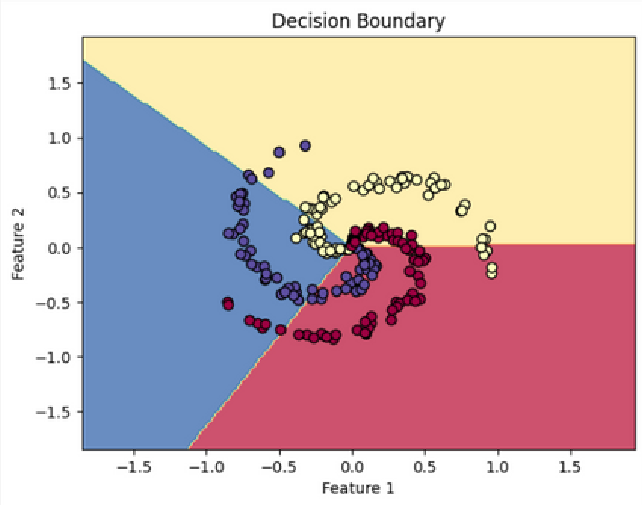
```
def plot_data(features,output,type):
    plt.scatter(features[:, 0], features[:, 1], c=output, s=40, cmap=plt.cm.Spectral,edgecolors='k')
    plt.title(f'{type} Data')
    plt.show()
```

[380] ✓ 0.0s

Then we plot the decision boundary,and our data sets

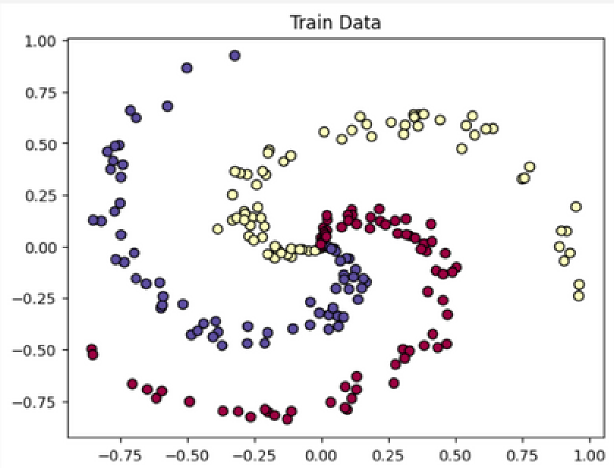
```python
# Plot decision boundary on training set
plot_decision_boundary(X_train, y_train, WTrained, bTrained)
plt.show()
```

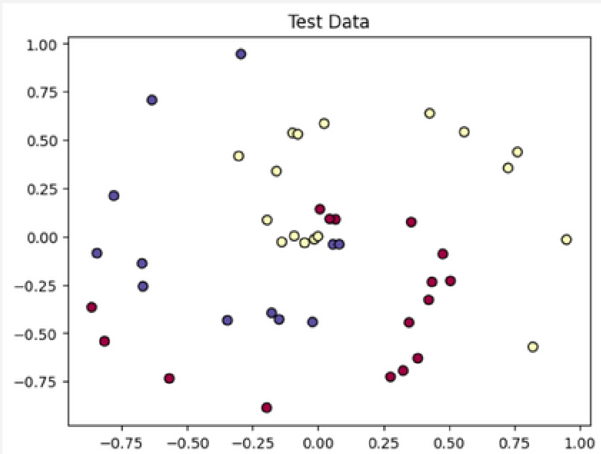
[418] ✓ 0.1s

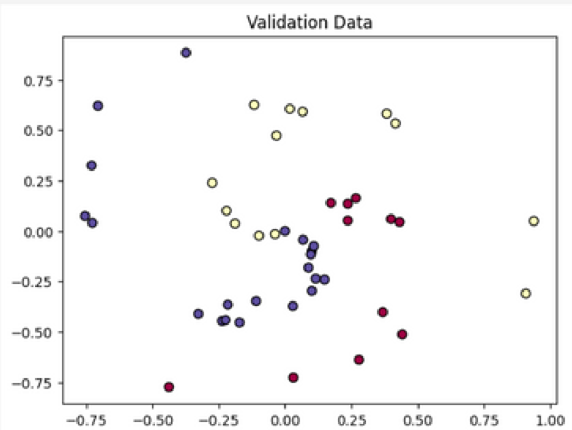



```python
plot_data(X_train, y_train,"Train")
plot_data(X_val, y_val,"Validation")
plot_data(X_test, y_test,"Test")
```

[419] ✓ 0.2s



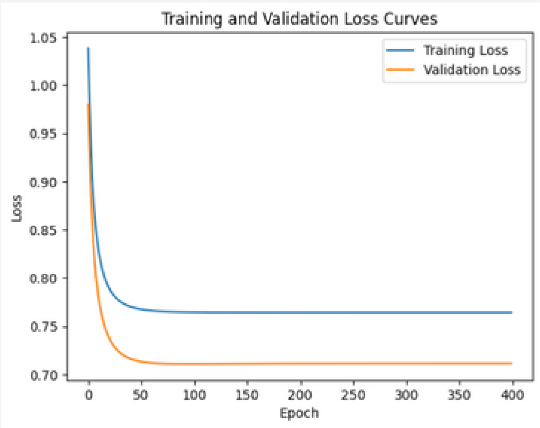




Finally we can see our model's loss

```
plt.plot(train_losses, label='Training Loss')
plt.plot(val_losses, label='Validation Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.title('Training and Validation Loss Curves')
plt.legend()
plt.show()
```

[481] ✓ 0.1s



Neural Network

Firstly we create our data and divide into train test and validation set using random_split_data funciton which we mentioned before

```python

# Generating synthetic data
N = 100
D = 2
K = 3
X = np.zeros((N * K, D))
y = np.zeros((N * K), dtype="uint8")

for j in range(K):
    ix = range(N * j, N * (j + 1))
    r = np.linspace(0.0, 1, N)
    t = np.linspace(j * 4, (j + 1) * 4, N) + np.random.randn(N) * 0.2
    X[ix] = np.c_[r * np.sin(t), r * np.cos(t)]
    y[ix] = j

plt.scatter(X[:, 0], X[:, 1], c=y, s=40, cmap=plt.cm.Spectral)
plt.show()

# Splitting the data into training, validation, and test sets
X_train, y_train, X_val, y_val, X_test, y_test = random_split_data(X, y)
```

Since we will create a 2-layer neurol network we create 2 weight and bias

```python
# Function to initialize neural network parameters
def initialize_parameters(D, neuron_count, K):
    # Initialize weights W1 with small random values.
    W1 = 0.01 * np.random.randn(D, neuron_count)

    # Initialize biases b1 with zeros.
    b1 = np.zeros((1, neuron_count))

    # Initialize weights W2 with small random values.
    W2 = 0.01 * np.random.randn(neuron_count, K)

    # Initialize biases b2 with zeros.
    b2 = np.zeros((1, K))

    return W1, b1, W2, b2
```

```python

# Initializing parameters for the neural network
W1, b1, W2, b2 = initialize_parameters(D, 100, K)

```

In forward_pass firstly,we calculate the dot product of input and weight matrix then add the bias to find output then we use the relu activicion function for feature transform and we get the hidden layer value .Then we apply same calculation on hidden layer using W2 and b2 we get a final output.Lastly we apply Softmax to get probabilities

```python
# Forward Pass Function
def forward_pass(X, W1, b1, W2, b2):
    # Calculate the first layer output z1 and apply ReLU activation function.
    output1 = np.dot(X, W1) + b1
    hidden = np.maximum(0, output1)  # ReLU activation

    # Calculate the second layer output z2.
    finalOutput = np.dot(hidden, W2) + b2

    # Apply the softmax function to get probabilities.
    exp_scores = np.exp(finalOutput)
    probs = exp_scores / np.sum(exp_scores, axis=1, keepdims=True)

    return hidden, probs
```

We call the loss function with the output of forward pass which is probabilities and calculate the cross-entorpy loss in the calculate_loss function

```
# Loss Calculation Function
def calculate_loss(probs, y):
    # Calculate cross-entropy loss for each class and sum them for the total loss.
    num_examples = len(y)
    corect_logprobs = -np.log(probs[range(num_examples), y])
    loss = np.sum(corect_logprobs) / num_examples

    return loss
```

```
# Forward pass to calculate loss before training
hidden, probs = forward_pass(X_train, W1, b1, W2, b2)
loss = calculate_loss(probs, y_train)
print("Loss on the first iteration: ", loss)
```

In the backward pass function, dscores is initially created as a copy of the probabilities (probs) to ensure that the probabilities remain unchanged throughout the subsequent computations. To expedite learning, the loss is increased by directly modifying dscores for the correct class, achieved by subtracting 1 from the probability associated with the correct class for each example. This manipulation effectively boosts the gradient, potentially accelerating the learning process.

The gradients for the second layer parameters, dW2 and db2, are then computed by utilizing the transposed hidden layer (hidden.T) and the modified dscores. Subsequently, the gradient of the loss with respect to the output of the first layer, da1, is calculated. This involves multiplying dscores by the transposed weights of the second layer (W2.T), with an additional application of the ReLU activation function gradient by setting values to zero where the corresponding values in the hidden layer are less than or equal to 0.

The gradients for the first layer parameters, dW1 and db1, are determined using the transposed input data (X.T) and the previously calculated da1. Following this, both layers' parameters are updated using gradient descent: the learning rate is multiplied by the corresponding gradients, and the results are subtracted from the current parameter values.

```
# Backward Pass and Parameter Update Function
def backward_pass(X, y, hidden, probs, W1, W2, b1, b2, learning_rate):
    # Compute gradients.
    num_examples = len(y)
    dscores = probs.copy()
    dscores[range(num_examples), y] -= 1

    # Compute gradients for the second layer parameters.
    dW2 = np.dot(hidden.T, dscores)
    db2 = np.sum(dscores, axis=0, keepdims=True)

    # Compute gradients for the first layer output.
    da1 = np.dot(dscores, W2.T)
    da1[hidden <= 0] = 0  # ReLU activation gradient

    # Compute gradients for the first layer parameters.
    dW1 = np.dot(X.T, da1)
    db1 = np.sum(da1, axis=0, keepdims=True)

    # Update parameters.
    W1 -= learning_rate * dW1
    b1 -= learning_rate * db1
    W2 -= learning_rate * dW2
    b2 -= learning_rate * db2

    return W1, b1, W2, b2
```

After updating our values, we find our new loss by repeating the previous process and we see that we have a lower loss.

```
# Backward Pass and Parameter Update
W1, b1, W2, b2 = backward_pass(X_train, y_train, hidden, probs, W1, W2, b1, b2, learning_rate=0.01)

# After the first iteration, calculate loss again to see if it decreases
hidden, probs = forward_pass(X_train, W1, b1, W2, b2)
loss = calculate_loss(probs, y_train)
print("Loss after one backpropagation: ", loss)
```

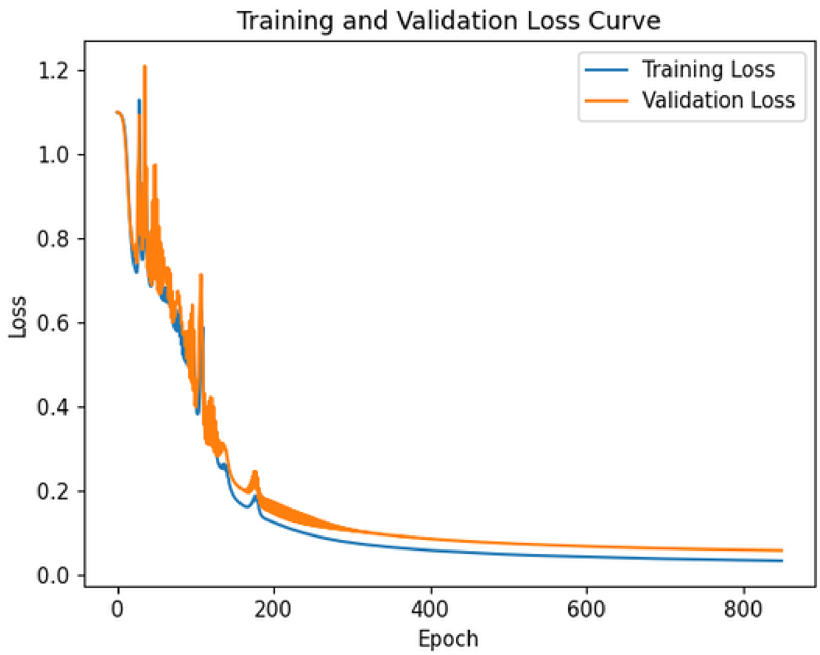
Now we come to the most important part we define a function which istrain_neural_network.
This function takes in both a training set (X_train, y_train) and a validation set (X_val, y_val), along with the initial parameters of the neural network (W1, b1, W2, b2), the learning rate (learning_rate), and the number of training epochs (epochs).Within the function, there's a loop that iterates over the specified number of epochs. During each epoch:
Forward Pass: The function computes the forward pass on the training and validation set, generating the hidden layer activations (hidden) and output probabilities (probs).
Loss Calculation: It calculates the cross-entropy loss for the training and validation set using the calculate_loss function.
Backward Pass and Parameter Update: The backward pass is performed to compute gradients, and the neural network parameters (W1, b1, W2, b2) are updated using the backward_pass function.
At the end of each epoch, the training and validation losses are stored in separate lists (train_losses and val_losses), allowing for later analysis, such as plotting learning curves.
The function eventually returns the final parameters of the neural network (W1, b1, W2, b2) after the training process, along with the lists of training and validation losses. This systematic approach enables monitoring the model's performance on both the training and validation sets throughout the training process.

```
# Training Neural Network Function
def train_neural_network(X_train, y_train, X_val, y_val, W1, b1, W2, b2, learning_rate, epochs):
    # Store losses for plotting.
    train_losses = []
    val_losses = []

    for epoch in range(epochs):
        # Forward pass for training set.
        hidden, probs = forward_pass(X_train, W1, b1, W2, b2)

        # Loss calculation for training set.
        train_loss = calculate_loss(probs, y_train)
        train_losses.append(train_loss)

        # Backward pass and parameter update for training set.
        W1, b1, W2, b2 = backward_pass(X_train, y_train, hidden, probs, W1, W2, b1, b2, learning_rate)

        # Forward pass for validation set.
        val_hidden, val_probs = forward_pass(X_val, W1, b1, W2, b2)

        # Loss calculation for validation set.
        val_loss = calculate_loss(val_probs, y_val)
        val_losses.append(val_loss)

    return W1, b1, W2, b2, train_losses, val_losses
```

```
# Training the neural network
W1, b1, W2, b2, train_losses, val_losses = train_neural_network(X_train, y_train, X_val, y_val, W1, b1, W2, b2, learning_rate=0.01, epochs=850)
```

Finally we plot all the necessary things

```python

# Plot the training loss curve
plt.plot(train_losses, label='Training Loss')
plt.plot(val_losses, label='Validation Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.title('Training and Validation Loss Curve')
plt.legend()
plt.show()



# Calculate accuracy on the test set
testAccuracy = calculate_accuracy(X_test, y_test, W1, b1, W2, b2)
print("Accuracy on the test set: {:.4%}".format(testAccuracy))


# Plot the learned decision boundaries on the training data
plot_decision_boundaries(X_train, y_train, W1, b1, W2, b2)

```




```
Loss on the first iteration:  1.0985255770082003
Loss after one backpropagation:  1.097886958650164
Accuracy on the test set: 97.7778%
```