# Niloo-Assignment1_proj

February 3, 2024

## 1 Deep Learning Course

### 1.1 Assignment 1

**Assignment Goals:**

- Start with PyTorch.
- Implement and apply logistic regression and multi-layer feed-forward neural network classifiers.
- Understand the differences and trade-offs between linear regression, logistic regression, and multi-layer feed-forward neural network.

In this assignment, you will be asked to install PyTorch and Jupyter Notebook. (TA's environment to run your code is Python 3.11 + Torch 2.1.2). In addition, you are required to design several models to classify a Toy Dataset (Figure 1).

**Dataset:** We provide a toy dataset, which has 200 instances and 2 features. See below "Toy Data and Helper Functions" section for toy data generation code.

You do not need to generate separated training dataset and test dataset for this assignment. Both training and prediction will both be on one dataset. Directly use the "sample, target" variables we provide as the dataset for your assignment.

In the following accuracy is defined as the empirical accuracy on the training set, that is, accuracy = {number of correctly predicted instances}/{number of all instances in dataset}).

**Requirements**

1. Install Pytorch() and Jupyter Notebook. (10 points)
2. Implement a logistic regression to classify the Toy Dataset. (20 points) We have provided a very simple linear regression example, please refer to the example and implement your logistic regression model.
    - You should determine: what loss function and what optimization algorithm do you plan to use? (4 points)
    - Try to reach $> 72\%$ accuracy. (4 points)
    - We have provided a visualize() helper function, you can visualize the model's decision boundary using that function. What's more, you are asked to compute and visualize **the equation of the decision boundary** of your trained **logistic regression**. Fill in the 'equation of decision boundary' column in the following table. Then you can modify the visualize() function or implement a new visualization function to draw the linear decision

boundary (Hint: should be a straight line aligned with the decision boundary plotted in visualize()). (5 points)

3. Implement a multi-layer linear neural network (>= 2 hidden layers) to classify the Toy Dataset. (20 points) A deep linear neural network is a deep feed-forward neural network without activation functions (See here, page 11-13 for detail introduction of linear neural networks).
   - You should determine: what loss function and what optimization algorithm do you plan to use, what is your network structure? (4 points)
   - Try to reach > 72% accuracy. (4 points)
   - Compute and visualize **the equation of the decision boundary** of your trained **linear neural network**. Fill in the 'equation of decision boundary' column in the following table. Then you can modify the visualize() function or implement a new visualization function to draw the linear decision boundary. (5 points)
4. Implement a multi-layer feed-forward neural network (>= 2 hidden layers). (20 points)
   - You should determine: what loss function and what optimization algorithm do you plan to use? what is your network structure? what activation function do you use? (5 points)
   - Try to reach 100% accuracy. (5 points)
5. Add L2-regularization to your implemented nonlinear neural network in (4.). Set the coefficient of L2-regularization to be 0.01, 2, 100, respectively. How do different values of coefficient of L2-regularization affect the model (i.e., model parameters, loss value, accuracy, decision boundary)? You can use a table to compare models trained without regularization, with different coefficients of regularization. (20 points)
   - Please draw your table and analysis in the '**Answers and Analysis**' section.

You should:

- Train each of your models to its best accuracy. Then fill in the following table in the '**Answers and Analysis**' section.
- Complete the '**Answers and Analysis**' section.

**Answers and Analysis**

- First, fill in the following table. The '-' indicates a cell that does not need to be filled in.

| Model | Loss | Accuracy | Equation of Decision Boundary | NN Structure | Activation Function | Optimization Algorithm | Loss Function |
|---|---|---|---|---|---|---|---|
| Linear Regression | 0.15 | 74% | $0.1817x_1 + 0.5237x_2 + 0.4758 = 0$ | - | - | SGD | Mean Square Error |
| Logistic Regression | 0.5939655303955078 | 77% | $x_2 = -(0.26430854201316833 * x_1 + tensor([-0.1892]))/0.8816277384757996$ | - | - | Adam | Binary Cross Entropy |

| Model | Loss | Accuracy | Equation of Decision Boundary | NN Structure | Activation Function | Optimization Algorithm | Loss Function |
|---|---|---|---|---|---|---|---|
| Linear Neural Network | 0.0076 | 74.8% | $x_2 = -(3.105049133300781 * x_1 + tensor([1.1808, 3.2083, 9.8797]))/9.809708595275879$ | layer input sizes: 2, | Sigmoid(only the last layer) | SGD | Binary Cross Entropy |
| Feedforward Neural Network | 0.0036 | 99.5% | - | layer input sizes:2, 8, 23, 30, 62, 16, 8, 4 | Relu for all the hidden layers, sigmoid for the last layer | SGD | Binary Cross Entropy |

- Then, compare and analyze the classification results of your models. In particular, are there any differences between the performance (i.e., accuracy, loss value) of linear regression, logistic regression, linear neural network and deep nonlinear neural network? What do you think is the reason for the difference? (10 points)
- Your table and analysis of (5. Add L2-regularization) here.

| L2 coefficient | Loss | Accuracy | Decision Boundary |
|---|---|---|---|
| None | 0.0036 | 99.5% | Ok |
| 0.01 | 0.027 | 99.1% | Ok |
| 2 | 0.693 | 50% | seems random |
| 100 | 0.693 | 50% | seems random |

#### 1.1.1 All the models start at the exact same place. Seems like lambda = 0.01 does a fantastic job! It goes down but comes back up to 91 percent. Others fall down and get stuck at the 50% accuracy.

**Submission Notes:** Please use Jupyter Notebook. The notebook should include the final code, results and your answers. You should submit your Notebook in both .pdf and .ipynb format.

**Instructions:** The university policy on academic dishonesty and plagiarism (cheating) will be taken very seriously in this course. Everything submitted should be your own writing or coding. You must not let other students copy your work. Spelling and grammar count.

Your assignments will be marked based on correctness, originality (the implementations and ideas are from yourself), clarity and performance. Clarity means whether the logic of your code is easy to follow. This includes 1) comments to explain the logic of your code 2) meaningful variable names. Performance includes loss value and accuracy after training.

## 1.2 Your Implementation

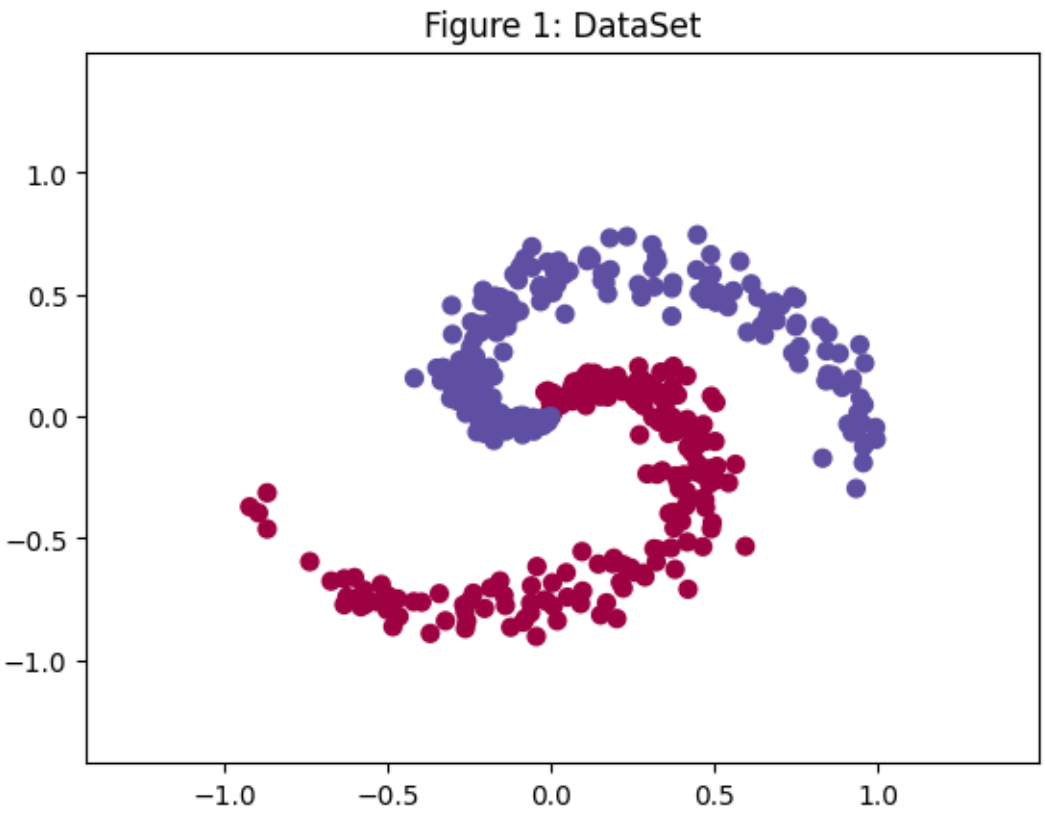
### 1.2.1 Toy Data and Helper Functions

[117]:
```
import numpy as np
import torch
import torch.nn as nn
import torch.optim as optim
import matplotlib.pyplot as plt
from sklearn.metrics import accuracy_score
```

[118]:
```
# helper functions

# helper function for geterating the data
def data_generator(N = 200,D = 2,K = 2):
    """
    N: number of points per class;
    D: dimensionality;
    K: number of classes
    """

    np.random.seed(10)
    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) # radius
      t = np.linspace(j*4,(j+1)*4,N) + np.random.randn(N)*0.3 # theta
      X[ix] = np.c_[r*np.sin(t), r*np.cos(t)]
      y[ix] = j

    fig = plt.figure()
    plt.title('Figure 1: DataSet')
    plt.scatter(X[:, 0], X[:, 1], c=y, s=40, cmap=plt.cm.Spectral)

    plt.xlim(X.min()-.5, X.max()+.5)
    plt.ylim(X.min()-.5, X.max()+.5)

    return X,y


# helper function for visualizing the decision boundaries
def visualize(sample, target, model):
    """
    Function for visualizing the classifier boundaries on the TOY dataset.

    sample: Training data features (PyTorch tensor)
```

```
    target: Target (PyTorch tensor)
    model: The PyTorch model
    """
    h = 0.02  # Step size in the meshgrid
    x_min, x_max = sample[:, 0].min() - 1, sample[:, 0].max() + 1
    y_min, y_max = sample[:, 1].min() - 1, sample[:, 1].max() + 1

    # Create a meshgrid for visualization
    xx, yy = torch.meshgrid(torch.arange(x_min, x_max, h), torch.arange(y_min,␣
↪y_max, h))

    # Flatten and concatenate the meshgrid for prediction
    grid_tensor = torch.cat((xx.reshape(-1, 1), yy.reshape(-1, 1)), dim=1)

    # Predict the class labels for each point in the meshgrid
    with torch.no_grad():
        model.eval()  # Set the model to evaluation mode
        predictions = model(grid_tensor)

    #Binary Classification
    Z = torch.where(predictions>0.5,1.0,0.0)
    Z = Z.reshape(xx.shape)

    # Create a contour plot to visualize the decision boundaries
    fig = plt.figure()
    plt.contourf(xx, yy, Z, cmap=plt.cm.Spectral, alpha=0.8)

    # Scatter plot the training data points
    plt.scatter(sample[:, 0], sample[:, 1], c=target, s=40, cmap=plt.cm.Spectral)

    # Set plot limits
    plt.xlim(xx.min(), xx.max())
    plt.ylim(yy.min(), yy.max())

    plt.show()
```

[119]:
```
#  TOY DataSet
sample, target = data_generator(N = 200)
# print(target.shape)
```




### 1.2.2 Given Example: Linear Regression

Note that linear regression is usually used for regression tasks, not classification tasks. However, it can be used for binary classification problems (be labeled 0, 1) with a threshold classifier. That is, when linear regression outputs $> 0.5$, the prediction is 1; otherwise, the prediction is 0.

[29]:
```
# Convert data to PyTorch tensors
X_tensor = torch.from_numpy(sample).float()
y_tensor = torch.from_numpy(target).float()

# Define the linear regression model
class LinearRegressionModel(nn.Module):
    def __init__(self, input_size):
        super(LinearRegressionModel, self).__init__()
        self.linear = nn.Linear(input_size,1)

    def forward(self, x):
        return self.linear(x)

# Instantiate the model, loss function, and optimizer
# input_size = 1  # Number of features in the input data
```

```
model = LinearRegressionModel(X_tensor.shape[1])
criterion = nn.MSELoss()
optimizer = optim.SGD(model.parameters(), lr=0.01)

# Training loop
num_epochs = 500
for epoch in range(num_epochs):
    # Forward pass
    y_pred = model(X_tensor)
    y_pred = y_pred.reshape(y_tensor.shape)

    # Compute the loss
    loss = criterion(y_pred, y_tensor)

    #Calculate Accuracy

    output = torch.where(y_pred>0.5, 1.0,0.0)
    acc = accuracy_score(y_tensor, output)

    # Backward pass and optimization
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

    print(f'Epoch [{epoch+1}/{num_epochs}], Loss: {loss.item():.4f}, Accuracy:␣
 ↪{acc}')
```

```
Epoch [1/500], Loss: 0.4022, Accuracy: 0.5
Epoch [2/500], Loss: 0.3936, Accuracy: 0.5
Epoch [3/500], Loss: 0.3854, Accuracy: 0.5
Epoch [4/500], Loss: 0.3775, Accuracy: 0.5
Epoch [5/500], Loss: 0.3698, Accuracy: 0.5
Epoch [6/500], Loss: 0.3625, Accuracy: 0.5
Epoch [7/500], Loss: 0.3555, Accuracy: 0.5
Epoch [8/500], Loss: 0.3487, Accuracy: 0.5
Epoch [9/500], Loss: 0.3422, Accuracy: 0.5
Epoch [10/500], Loss: 0.3360, Accuracy: 0.5
Epoch [11/500], Loss: 0.3300, Accuracy: 0.5
Epoch [12/500], Loss: 0.3242, Accuracy: 0.5
Epoch [13/500], Loss: 0.3187, Accuracy: 0.5
Epoch [14/500], Loss: 0.3133, Accuracy: 0.5
Epoch [15/500], Loss: 0.3082, Accuracy: 0.5
Epoch [16/500], Loss: 0.3033, Accuracy: 0.5
Epoch [17/500], Loss: 0.2986, Accuracy: 0.5025
Epoch [18/500], Loss: 0.2940, Accuracy: 0.505
Epoch [19/500], Loss: 0.2896, Accuracy: 0.51
Epoch [20/500], Loss: 0.2854, Accuracy: 0.5125
Epoch [21/500], Loss: 0.2814, Accuracy: 0.52
```

```
Epoch [22/500], Loss: 0.2775, Accuracy: 0.5275
Epoch [23/500], Loss: 0.2737, Accuracy: 0.535
Epoch [24/500], Loss: 0.2701, Accuracy: 0.55
Epoch [25/500], Loss: 0.2666, Accuracy: 0.56
Epoch [26/500], Loss: 0.2633, Accuracy: 0.5725
Epoch [27/500], Loss: 0.2601, Accuracy: 0.58
Epoch [28/500], Loss: 0.2570, Accuracy: 0.5825
Epoch [29/500], Loss: 0.2540, Accuracy: 0.5975
Epoch [30/500], Loss: 0.2512, Accuracy: 0.605
Epoch [31/500], Loss: 0.2484, Accuracy: 0.6125
Epoch [32/500], Loss: 0.2458, Accuracy: 0.6225
Epoch [33/500], Loss: 0.2432, Accuracy: 0.6275
Epoch [34/500], Loss: 0.2407, Accuracy: 0.6375
Epoch [35/500], Loss: 0.2384, Accuracy: 0.645
Epoch [36/500], Loss: 0.2361, Accuracy: 0.65
Epoch [37/500], Loss: 0.2339, Accuracy: 0.6525
Epoch [38/500], Loss: 0.2318, Accuracy: 0.655
Epoch [39/500], Loss: 0.2297, Accuracy: 0.66
Epoch [40/500], Loss: 0.2278, Accuracy: 0.6675
Epoch [41/500], Loss: 0.2259, Accuracy: 0.675
Epoch [42/500], Loss: 0.2241, Accuracy: 0.68
Epoch [43/500], Loss: 0.2223, Accuracy: 0.6875
Epoch [44/500], Loss: 0.2206, Accuracy: 0.69
Epoch [45/500], Loss: 0.2190, Accuracy: 0.6925
Epoch [46/500], Loss: 0.2174, Accuracy: 0.7
Epoch [47/500], Loss: 0.2159, Accuracy: 0.7025
Epoch [48/500], Loss: 0.2144, Accuracy: 0.705
Epoch [49/500], Loss: 0.2130, Accuracy: 0.71
Epoch [50/500], Loss: 0.2116, Accuracy: 0.7125
Epoch [51/500], Loss: 0.2103, Accuracy: 0.7125
Epoch [52/500], Loss: 0.2090, Accuracy: 0.7175
Epoch [53/500], Loss: 0.2078, Accuracy: 0.7225
Epoch [54/500], Loss: 0.2066, Accuracy: 0.725
Epoch [55/500], Loss: 0.2055, Accuracy: 0.73
Epoch [56/500], Loss: 0.2043, Accuracy: 0.7325
Epoch [57/500], Loss: 0.2033, Accuracy: 0.7325
Epoch [58/500], Loss: 0.2022, Accuracy: 0.7325
Epoch [59/500], Loss: 0.2012, Accuracy: 0.735
Epoch [60/500], Loss: 0.2003, Accuracy: 0.735
Epoch [61/500], Loss: 0.1993, Accuracy: 0.7325
Epoch [62/500], Loss: 0.1984, Accuracy: 0.73
Epoch [63/500], Loss: 0.1976, Accuracy: 0.7325
Epoch [64/500], Loss: 0.1967, Accuracy: 0.7325
Epoch [65/500], Loss: 0.1959, Accuracy: 0.7325
Epoch [66/500], Loss: 0.1951, Accuracy: 0.7325
Epoch [67/500], Loss: 0.1943, Accuracy: 0.74
Epoch [68/500], Loss: 0.1936, Accuracy: 0.7425
Epoch [69/500], Loss: 0.1929, Accuracy: 0.74
```

```
Epoch [70/500], Loss: 0.1922, Accuracy: 0.7425
Epoch [71/500], Loss: 0.1915, Accuracy: 0.7425
Epoch [72/500], Loss: 0.1908, Accuracy: 0.74
Epoch [73/500], Loss: 0.1902, Accuracy: 0.745
Epoch [74/500], Loss: 0.1896, Accuracy: 0.75
Epoch [75/500], Loss: 0.1890, Accuracy: 0.75
Epoch [76/500], Loss: 0.1884, Accuracy: 0.7525
Epoch [77/500], Loss: 0.1878, Accuracy: 0.7475
Epoch [78/500], Loss: 0.1873, Accuracy: 0.7475
Epoch [79/500], Loss: 0.1867, Accuracy: 0.745
Epoch [80/500], Loss: 0.1862, Accuracy: 0.7425
Epoch [81/500], Loss: 0.1857, Accuracy: 0.7425
Epoch [82/500], Loss: 0.1852, Accuracy: 0.7425
Epoch [83/500], Loss: 0.1847, Accuracy: 0.74
Epoch [84/500], Loss: 0.1843, Accuracy: 0.74
Epoch [85/500], Loss: 0.1838, Accuracy: 0.7375
Epoch [86/500], Loss: 0.1834, Accuracy: 0.73
Epoch [87/500], Loss: 0.1829, Accuracy: 0.7325
Epoch [88/500], Loss: 0.1825, Accuracy: 0.7325
Epoch [89/500], Loss: 0.1821, Accuracy: 0.7275
Epoch [90/500], Loss: 0.1817, Accuracy: 0.7275
Epoch [91/500], Loss: 0.1813, Accuracy: 0.7275
Epoch [92/500], Loss: 0.1809, Accuracy: 0.7225
Epoch [93/500], Loss: 0.1806, Accuracy: 0.72
Epoch [94/500], Loss: 0.1802, Accuracy: 0.72
Epoch [95/500], Loss: 0.1799, Accuracy: 0.7175
Epoch [96/500], Loss: 0.1795, Accuracy: 0.7175
Epoch [97/500], Loss: 0.1792, Accuracy: 0.715
Epoch [98/500], Loss: 0.1789, Accuracy: 0.7125
Epoch [99/500], Loss: 0.1785, Accuracy: 0.71
Epoch [100/500], Loss: 0.1782, Accuracy: 0.705
Epoch [101/500], Loss: 0.1779, Accuracy: 0.7
Epoch [102/500], Loss: 0.1776, Accuracy: 0.7
Epoch [103/500], Loss: 0.1773, Accuracy: 0.6975
Epoch [104/500], Loss: 0.1770, Accuracy: 0.6975
Epoch [105/500], Loss: 0.1767, Accuracy: 0.6975
Epoch [106/500], Loss: 0.1765, Accuracy: 0.6975
Epoch [107/500], Loss: 0.1762, Accuracy: 0.6925
Epoch [108/500], Loss: 0.1759, Accuracy: 0.6875
Epoch [109/500], Loss: 0.1757, Accuracy: 0.6875
Epoch [110/500], Loss: 0.1754, Accuracy: 0.6875
Epoch [111/500], Loss: 0.1752, Accuracy: 0.685
Epoch [112/500], Loss: 0.1749, Accuracy: 0.6825
Epoch [113/500], Loss: 0.1747, Accuracy: 0.6825
Epoch [114/500], Loss: 0.1744, Accuracy: 0.68
Epoch [115/500], Loss: 0.1742, Accuracy: 0.675
Epoch [116/500], Loss: 0.1740, Accuracy: 0.68
Epoch [117/500], Loss: 0.1737, Accuracy: 0.68
```

```
Epoch [118/500], Loss: 0.1735, Accuracy: 0.68
Epoch [119/500], Loss: 0.1733, Accuracy: 0.6775
Epoch [120/500], Loss: 0.1731, Accuracy: 0.6725
Epoch [121/500], Loss: 0.1729, Accuracy: 0.6725
Epoch [122/500], Loss: 0.1727, Accuracy: 0.6725
Epoch [123/500], Loss: 0.1725, Accuracy: 0.6725
Epoch [124/500], Loss: 0.1723, Accuracy: 0.67
Epoch [125/500], Loss: 0.1721, Accuracy: 0.6675
Epoch [126/500], Loss: 0.1719, Accuracy: 0.6675
Epoch [127/500], Loss: 0.1717, Accuracy: 0.67
Epoch [128/500], Loss: 0.1715, Accuracy: 0.6675
Epoch [129/500], Loss: 0.1713, Accuracy: 0.6675
Epoch [130/500], Loss: 0.1711, Accuracy: 0.6675
Epoch [131/500], Loss: 0.1709, Accuracy: 0.6675
Epoch [132/500], Loss: 0.1707, Accuracy: 0.665
Epoch [133/500], Loss: 0.1706, Accuracy: 0.6625
Epoch [134/500], Loss: 0.1704, Accuracy: 0.665
Epoch [135/500], Loss: 0.1702, Accuracy: 0.665
Epoch [136/500], Loss: 0.1700, Accuracy: 0.6625
Epoch [137/500], Loss: 0.1699, Accuracy: 0.66
Epoch [138/500], Loss: 0.1697, Accuracy: 0.66
Epoch [139/500], Loss: 0.1695, Accuracy: 0.6625
Epoch [140/500], Loss: 0.1694, Accuracy: 0.6625
Epoch [141/500], Loss: 0.1692, Accuracy: 0.66
Epoch [142/500], Loss: 0.1690, Accuracy: 0.66
Epoch [143/500], Loss: 0.1689, Accuracy: 0.66
Epoch [144/500], Loss: 0.1687, Accuracy: 0.6625
Epoch [145/500], Loss: 0.1686, Accuracy: 0.6625
Epoch [146/500], Loss: 0.1684, Accuracy: 0.665
Epoch [147/500], Loss: 0.1683, Accuracy: 0.665
Epoch [148/500], Loss: 0.1681, Accuracy: 0.665
Epoch [149/500], Loss: 0.1680, Accuracy: 0.665
Epoch [150/500], Loss: 0.1678, Accuracy: 0.6625
Epoch [151/500], Loss: 0.1677, Accuracy: 0.6625
Epoch [152/500], Loss: 0.1675, Accuracy: 0.66
Epoch [153/500], Loss: 0.1674, Accuracy: 0.66
Epoch [154/500], Loss: 0.1672, Accuracy: 0.66
Epoch [155/500], Loss: 0.1671, Accuracy: 0.6625
Epoch [156/500], Loss: 0.1669, Accuracy: 0.6625
Epoch [157/500], Loss: 0.1668, Accuracy: 0.66
Epoch [158/500], Loss: 0.1667, Accuracy: 0.6575
Epoch [159/500], Loss: 0.1665, Accuracy: 0.6575
Epoch [160/500], Loss: 0.1664, Accuracy: 0.655
Epoch [161/500], Loss: 0.1663, Accuracy: 0.655
Epoch [162/500], Loss: 0.1661, Accuracy: 0.6525
Epoch [163/500], Loss: 0.1660, Accuracy: 0.6525
Epoch [164/500], Loss: 0.1659, Accuracy: 0.6525
Epoch [165/500], Loss: 0.1657, Accuracy: 0.6525
```

```
Epoch [166/500], Loss: 0.1656, Accuracy: 0.6525
Epoch [167/500], Loss: 0.1655, Accuracy: 0.6525
Epoch [168/500], Loss: 0.1653, Accuracy: 0.65
Epoch [169/500], Loss: 0.1652, Accuracy: 0.6475
Epoch [170/500], Loss: 0.1651, Accuracy: 0.6475
Epoch [171/500], Loss: 0.1650, Accuracy: 0.65
Epoch [172/500], Loss: 0.1648, Accuracy: 0.6525
Epoch [173/500], Loss: 0.1647, Accuracy: 0.6525
Epoch [174/500], Loss: 0.1646, Accuracy: 0.65
Epoch [175/500], Loss: 0.1645, Accuracy: 0.6525
Epoch [176/500], Loss: 0.1644, Accuracy: 0.6525
Epoch [177/500], Loss: 0.1642, Accuracy: 0.6525
Epoch [178/500], Loss: 0.1641, Accuracy: 0.655
Epoch [179/500], Loss: 0.1640, Accuracy: 0.66
Epoch [180/500], Loss: 0.1639, Accuracy: 0.66
Epoch [181/500], Loss: 0.1638, Accuracy: 0.66
Epoch [182/500], Loss: 0.1636, Accuracy: 0.66
Epoch [183/500], Loss: 0.1635, Accuracy: 0.66
Epoch [184/500], Loss: 0.1634, Accuracy: 0.6575
Epoch [185/500], Loss: 0.1633, Accuracy: 0.6575
Epoch [186/500], Loss: 0.1632, Accuracy: 0.6575
Epoch [187/500], Loss: 0.1631, Accuracy: 0.6575
Epoch [188/500], Loss: 0.1630, Accuracy: 0.6575
Epoch [189/500], Loss: 0.1629, Accuracy: 0.6575
Epoch [190/500], Loss: 0.1627, Accuracy: 0.6575
Epoch [191/500], Loss: 0.1626, Accuracy: 0.6575
Epoch [192/500], Loss: 0.1625, Accuracy: 0.6575
Epoch [193/500], Loss: 0.1624, Accuracy: 0.6575
Epoch [194/500], Loss: 0.1623, Accuracy: 0.6575
Epoch [195/500], Loss: 0.1622, Accuracy: 0.6575
Epoch [196/500], Loss: 0.1621, Accuracy: 0.655
Epoch [197/500], Loss: 0.1620, Accuracy: 0.6575
Epoch [198/500], Loss: 0.1619, Accuracy: 0.6575
Epoch [199/500], Loss: 0.1618, Accuracy: 0.6575
Epoch [200/500], Loss: 0.1617, Accuracy: 0.6575
Epoch [201/500], Loss: 0.1616, Accuracy: 0.6575
Epoch [202/500], Loss: 0.1615, Accuracy: 0.655
Epoch [203/500], Loss: 0.1614, Accuracy: 0.655
Epoch [204/500], Loss: 0.1613, Accuracy: 0.655
Epoch [205/500], Loss: 0.1612, Accuracy: 0.655
Epoch [206/500], Loss: 0.1611, Accuracy: 0.6575
Epoch [207/500], Loss: 0.1610, Accuracy: 0.6575
Epoch [208/500], Loss: 0.1609, Accuracy: 0.6575
Epoch [209/500], Loss: 0.1608, Accuracy: 0.655
Epoch [210/500], Loss: 0.1607, Accuracy: 0.655
Epoch [211/500], Loss: 0.1606, Accuracy: 0.6575
Epoch [212/500], Loss: 0.1605, Accuracy: 0.6575
Epoch [213/500], Loss: 0.1604, Accuracy: 0.66
```

```
Epoch [214/500], Loss: 0.1603, Accuracy: 0.66
Epoch [215/500], Loss: 0.1602, Accuracy: 0.66
Epoch [216/500], Loss: 0.1601, Accuracy: 0.66
Epoch [217/500], Loss: 0.1600, Accuracy: 0.66
Epoch [218/500], Loss: 0.1599, Accuracy: 0.66
Epoch [219/500], Loss: 0.1598, Accuracy: 0.665
Epoch [220/500], Loss: 0.1597, Accuracy: 0.665
Epoch [221/500], Loss: 0.1596, Accuracy: 0.665
Epoch [222/500], Loss: 0.1595, Accuracy: 0.6625
Epoch [223/500], Loss: 0.1594, Accuracy: 0.6625
Epoch [224/500], Loss: 0.1594, Accuracy: 0.6625
Epoch [225/500], Loss: 0.1593, Accuracy: 0.6625
Epoch [226/500], Loss: 0.1592, Accuracy: 0.6625
Epoch [227/500], Loss: 0.1591, Accuracy: 0.665
Epoch [228/500], Loss: 0.1590, Accuracy: 0.665
Epoch [229/500], Loss: 0.1589, Accuracy: 0.665
Epoch [230/500], Loss: 0.1588, Accuracy: 0.665
Epoch [231/500], Loss: 0.1587, Accuracy: 0.665
Epoch [232/500], Loss: 0.1586, Accuracy: 0.665
Epoch [233/500], Loss: 0.1586, Accuracy: 0.6675
Epoch [234/500], Loss: 0.1585, Accuracy: 0.6675
Epoch [235/500], Loss: 0.1584, Accuracy: 0.665
Epoch [236/500], Loss: 0.1583, Accuracy: 0.665
Epoch [237/500], Loss: 0.1582, Accuracy: 0.6675
Epoch [238/500], Loss: 0.1581, Accuracy: 0.6675
Epoch [239/500], Loss: 0.1580, Accuracy: 0.6675
Epoch [240/500], Loss: 0.1580, Accuracy: 0.6675
Epoch [241/500], Loss: 0.1579, Accuracy: 0.6675
Epoch [242/500], Loss: 0.1578, Accuracy: 0.67
Epoch [243/500], Loss: 0.1577, Accuracy: 0.67
Epoch [244/500], Loss: 0.1576, Accuracy: 0.67
Epoch [245/500], Loss: 0.1576, Accuracy: 0.67
Epoch [246/500], Loss: 0.1575, Accuracy: 0.67
Epoch [247/500], Loss: 0.1574, Accuracy: 0.67
Epoch [248/500], Loss: 0.1573, Accuracy: 0.67
Epoch [249/500], Loss: 0.1572, Accuracy: 0.67
Epoch [250/500], Loss: 0.1572, Accuracy: 0.67
Epoch [251/500], Loss: 0.1571, Accuracy: 0.67
Epoch [252/500], Loss: 0.1570, Accuracy: 0.6675
Epoch [253/500], Loss: 0.1569, Accuracy: 0.6675
Epoch [254/500], Loss: 0.1568, Accuracy: 0.6675
Epoch [255/500], Loss: 0.1568, Accuracy: 0.6675
Epoch [256/500], Loss: 0.1567, Accuracy: 0.6675
Epoch [257/500], Loss: 0.1566, Accuracy: 0.6675
Epoch [258/500], Loss: 0.1565, Accuracy: 0.6675
Epoch [259/500], Loss: 0.1565, Accuracy: 0.67
Epoch [260/500], Loss: 0.1564, Accuracy: 0.67
Epoch [261/500], Loss: 0.1563, Accuracy: 0.67
```

```
Epoch [262/500], Loss: 0.1562, Accuracy: 0.6675
Epoch [263/500], Loss: 0.1562, Accuracy: 0.6675
Epoch [264/500], Loss: 0.1561, Accuracy: 0.6675
Epoch [265/500], Loss: 0.1560, Accuracy: 0.6675
Epoch [266/500], Loss: 0.1559, Accuracy: 0.6675
Epoch [267/500], Loss: 0.1559, Accuracy: 0.6675
Epoch [268/500], Loss: 0.1558, Accuracy: 0.6675
Epoch [269/500], Loss: 0.1557, Accuracy: 0.6675
Epoch [270/500], Loss: 0.1556, Accuracy: 0.6675
Epoch [271/500], Loss: 0.1556, Accuracy: 0.6675
Epoch [272/500], Loss: 0.1555, Accuracy: 0.665
Epoch [273/500], Loss: 0.1554, Accuracy: 0.665
Epoch [274/500], Loss: 0.1554, Accuracy: 0.665
Epoch [275/500], Loss: 0.1553, Accuracy: 0.665
Epoch [276/500], Loss: 0.1552, Accuracy: 0.665
Epoch [277/500], Loss: 0.1551, Accuracy: 0.665
Epoch [278/500], Loss: 0.1551, Accuracy: 0.665
Epoch [279/500], Loss: 0.1550, Accuracy: 0.665
Epoch [280/500], Loss: 0.1549, Accuracy: 0.665
Epoch [281/500], Loss: 0.1549, Accuracy: 0.6675
Epoch [282/500], Loss: 0.1548, Accuracy: 0.6675
Epoch [283/500], Loss: 0.1547, Accuracy: 0.6675
Epoch [284/500], Loss: 0.1547, Accuracy: 0.6675
Epoch [285/500], Loss: 0.1546, Accuracy: 0.6675
Epoch [286/500], Loss: 0.1545, Accuracy: 0.6675
Epoch [287/500], Loss: 0.1545, Accuracy: 0.6675
Epoch [288/500], Loss: 0.1544, Accuracy: 0.6675
Epoch [289/500], Loss: 0.1543, Accuracy: 0.6675
Epoch [290/500], Loss: 0.1543, Accuracy: 0.6675
Epoch [291/500], Loss: 0.1542, Accuracy: 0.6675
Epoch [292/500], Loss: 0.1541, Accuracy: 0.6675
Epoch [293/500], Loss: 0.1541, Accuracy: 0.6675
Epoch [294/500], Loss: 0.1540, Accuracy: 0.6675
Epoch [295/500], Loss: 0.1540, Accuracy: 0.6675
Epoch [296/500], Loss: 0.1539, Accuracy: 0.6675
Epoch [297/500], Loss: 0.1538, Accuracy: 0.6675
Epoch [298/500], Loss: 0.1538, Accuracy: 0.6675
Epoch [299/500], Loss: 0.1537, Accuracy: 0.6675
Epoch [300/500], Loss: 0.1536, Accuracy: 0.6675
Epoch [301/500], Loss: 0.1536, Accuracy: 0.6675
Epoch [302/500], Loss: 0.1535, Accuracy: 0.6675
Epoch [303/500], Loss: 0.1535, Accuracy: 0.665
Epoch [304/500], Loss: 0.1534, Accuracy: 0.665
Epoch [305/500], Loss: 0.1533, Accuracy: 0.665
Epoch [306/500], Loss: 0.1533, Accuracy: 0.665
Epoch [307/500], Loss: 0.1532, Accuracy: 0.665
Epoch [308/500], Loss: 0.1532, Accuracy: 0.665
Epoch [309/500], Loss: 0.1531, Accuracy: 0.665
```

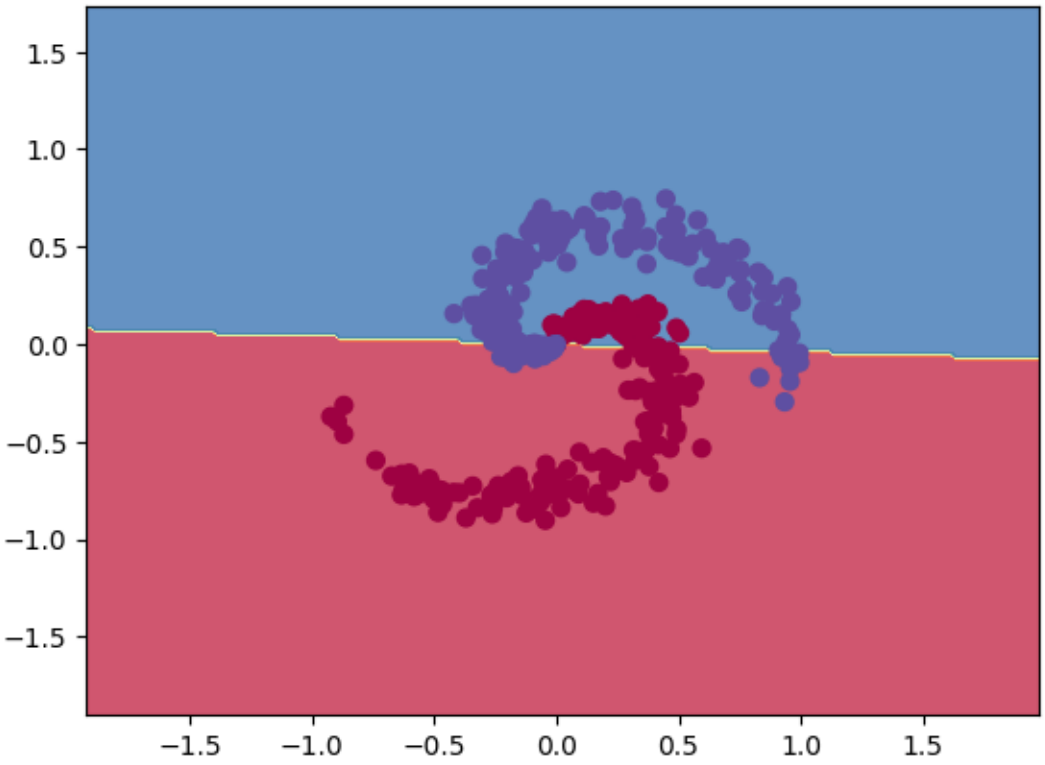
```
Epoch [310/500], Loss: 0.1530, Accuracy: 0.665
Epoch [311/500], Loss: 0.1530, Accuracy: 0.665
Epoch [312/500], Loss: 0.1529, Accuracy: 0.665
Epoch [313/500], Loss: 0.1529, Accuracy: 0.665
Epoch [314/500], Loss: 0.1528, Accuracy: 0.665
Epoch [315/500], Loss: 0.1527, Accuracy: 0.665
Epoch [316/500], Loss: 0.1527, Accuracy: 0.665
Epoch [317/500], Loss: 0.1526, Accuracy: 0.665
Epoch [318/500], Loss: 0.1526, Accuracy: 0.6675
Epoch [319/500], Loss: 0.1525, Accuracy: 0.6675
Epoch [320/500], Loss: 0.1525, Accuracy: 0.67
Epoch [321/500], Loss: 0.1524, Accuracy: 0.6725
Epoch [322/500], Loss: 0.1523, Accuracy: 0.6725
Epoch [323/500], Loss: 0.1523, Accuracy: 0.6725
Epoch [324/500], Loss: 0.1522, Accuracy: 0.6725
Epoch [325/500], Loss: 0.1522, Accuracy: 0.6725
Epoch [326/500], Loss: 0.1521, Accuracy: 0.6725
Epoch [327/500], Loss: 0.1521, Accuracy: 0.67
Epoch [328/500], Loss: 0.1520, Accuracy: 0.67
Epoch [329/500], Loss: 0.1520, Accuracy: 0.67
Epoch [330/500], Loss: 0.1519, Accuracy: 0.67
Epoch [331/500], Loss: 0.1519, Accuracy: 0.67
Epoch [332/500], Loss: 0.1518, Accuracy: 0.67
Epoch [333/500], Loss: 0.1518, Accuracy: 0.67
Epoch [334/500], Loss: 0.1517, Accuracy: 0.6725
Epoch [335/500], Loss: 0.1516, Accuracy: 0.6725
Epoch [336/500], Loss: 0.1516, Accuracy: 0.675
Epoch [337/500], Loss: 0.1515, Accuracy: 0.675
Epoch [338/500], Loss: 0.1515, Accuracy: 0.675
Epoch [339/500], Loss: 0.1514, Accuracy: 0.675
Epoch [340/500], Loss: 0.1514, Accuracy: 0.675
Epoch [341/500], Loss: 0.1513, Accuracy: 0.675
Epoch [342/500], Loss: 0.1513, Accuracy: 0.675
Epoch [343/500], Loss: 0.1512, Accuracy: 0.675
Epoch [344/500], Loss: 0.1512, Accuracy: 0.6725
Epoch [345/500], Loss: 0.1511, Accuracy: 0.675
Epoch [346/500], Loss: 0.1511, Accuracy: 0.675
Epoch [347/500], Loss: 0.1510, Accuracy: 0.675
Epoch [348/500], Loss: 0.1510, Accuracy: 0.675
Epoch [349/500], Loss: 0.1509, Accuracy: 0.675
Epoch [350/500], Loss: 0.1509, Accuracy: 0.675
Epoch [351/500], Loss: 0.1508, Accuracy: 0.675
Epoch [352/500], Loss: 0.1508, Accuracy: 0.675
Epoch [353/500], Loss: 0.1507, Accuracy: 0.675
Epoch [354/500], Loss: 0.1507, Accuracy: 0.675
Epoch [355/500], Loss: 0.1507, Accuracy: 0.675
Epoch [356/500], Loss: 0.1506, Accuracy: 0.675
Epoch [357/500], Loss: 0.1506, Accuracy: 0.675
```

```
Epoch [358/500], Loss: 0.1505, Accuracy: 0.6775
Epoch [359/500], Loss: 0.1505, Accuracy: 0.6775
Epoch [360/500], Loss: 0.1504, Accuracy: 0.6775
Epoch [361/500], Loss: 0.1504, Accuracy: 0.6775
Epoch [362/500], Loss: 0.1503, Accuracy: 0.675
Epoch [363/500], Loss: 0.1503, Accuracy: 0.675
Epoch [364/500], Loss: 0.1502, Accuracy: 0.675
Epoch [365/500], Loss: 0.1502, Accuracy: 0.675
Epoch [366/500], Loss: 0.1501, Accuracy: 0.675
Epoch [367/500], Loss: 0.1501, Accuracy: 0.675
Epoch [368/500], Loss: 0.1501, Accuracy: 0.6725
Epoch [369/500], Loss: 0.1500, Accuracy: 0.6725
Epoch [370/500], Loss: 0.1500, Accuracy: 0.6725
Epoch [371/500], Loss: 0.1499, Accuracy: 0.6725
Epoch [372/500], Loss: 0.1499, Accuracy: 0.6725
Epoch [373/500], Loss: 0.1498, Accuracy: 0.6725
Epoch [374/500], Loss: 0.1498, Accuracy: 0.6725
Epoch [375/500], Loss: 0.1497, Accuracy: 0.6725
Epoch [376/500], Loss: 0.1497, Accuracy: 0.6725
Epoch [377/500], Loss: 0.1497, Accuracy: 0.6725
Epoch [378/500], Loss: 0.1496, Accuracy: 0.6725
Epoch [379/500], Loss: 0.1496, Accuracy: 0.6725
Epoch [380/500], Loss: 0.1495, Accuracy: 0.6725
Epoch [381/500], Loss: 0.1495, Accuracy: 0.6725
Epoch [382/500], Loss: 0.1495, Accuracy: 0.6725
Epoch [383/500], Loss: 0.1494, Accuracy: 0.6725
Epoch [384/500], Loss: 0.1494, Accuracy: 0.6725
Epoch [385/500], Loss: 0.1493, Accuracy: 0.6725
Epoch [386/500], Loss: 0.1493, Accuracy: 0.6725
Epoch [387/500], Loss: 0.1492, Accuracy: 0.6725
Epoch [388/500], Loss: 0.1492, Accuracy: 0.6725
Epoch [389/500], Loss: 0.1492, Accuracy: 0.6725
Epoch [390/500], Loss: 0.1491, Accuracy: 0.6725
Epoch [391/500], Loss: 0.1491, Accuracy: 0.6725
Epoch [392/500], Loss: 0.1490, Accuracy: 0.6725
Epoch [393/500], Loss: 0.1490, Accuracy: 0.6725
Epoch [394/500], Loss: 0.1490, Accuracy: 0.6725
Epoch [395/500], Loss: 0.1489, Accuracy: 0.67
Epoch [396/500], Loss: 0.1489, Accuracy: 0.67
Epoch [397/500], Loss: 0.1489, Accuracy: 0.67
Epoch [398/500], Loss: 0.1488, Accuracy: 0.67
Epoch [399/500], Loss: 0.1488, Accuracy: 0.67
Epoch [400/500], Loss: 0.1487, Accuracy: 0.67
Epoch [401/500], Loss: 0.1487, Accuracy: 0.67
Epoch [402/500], Loss: 0.1487, Accuracy: 0.6725
Epoch [403/500], Loss: 0.1486, Accuracy: 0.6725
Epoch [404/500], Loss: 0.1486, Accuracy: 0.6725
Epoch [405/500], Loss: 0.1485, Accuracy: 0.6725
```

```
Epoch [406/500], Loss: 0.1485, Accuracy: 0.6725
Epoch [407/500], Loss: 0.1485, Accuracy: 0.6725
Epoch [408/500], Loss: 0.1484, Accuracy: 0.6725
Epoch [409/500], Loss: 0.1484, Accuracy: 0.6725
Epoch [410/500], Loss: 0.1484, Accuracy: 0.6725
Epoch [411/500], Loss: 0.1483, Accuracy: 0.6725
Epoch [412/500], Loss: 0.1483, Accuracy: 0.6725
Epoch [413/500], Loss: 0.1483, Accuracy: 0.6725
Epoch [414/500], Loss: 0.1482, Accuracy: 0.6725
Epoch [415/500], Loss: 0.1482, Accuracy: 0.675
Epoch [416/500], Loss: 0.1481, Accuracy: 0.675
Epoch [417/500], Loss: 0.1481, Accuracy: 0.6725
Epoch [418/500], Loss: 0.1481, Accuracy: 0.6725
Epoch [419/500], Loss: 0.1480, Accuracy: 0.6725
Epoch [420/500], Loss: 0.1480, Accuracy: 0.6725
Epoch [421/500], Loss: 0.1480, Accuracy: 0.6725
Epoch [422/500], Loss: 0.1479, Accuracy: 0.6725
Epoch [423/500], Loss: 0.1479, Accuracy: 0.6725
Epoch [424/500], Loss: 0.1479, Accuracy: 0.6725
Epoch [425/500], Loss: 0.1478, Accuracy: 0.6725
Epoch [426/500], Loss: 0.1478, Accuracy: 0.6725
Epoch [427/500], Loss: 0.1478, Accuracy: 0.6725
Epoch [428/500], Loss: 0.1477, Accuracy: 0.675
Epoch [429/500], Loss: 0.1477, Accuracy: 0.675
Epoch [430/500], Loss: 0.1477, Accuracy: 0.6775
Epoch [431/500], Loss: 0.1476, Accuracy: 0.6775
Epoch [432/500], Loss: 0.1476, Accuracy: 0.675
Epoch [433/500], Loss: 0.1476, Accuracy: 0.675
Epoch [434/500], Loss: 0.1475, Accuracy: 0.675
Epoch [435/500], Loss: 0.1475, Accuracy: 0.675
Epoch [436/500], Loss: 0.1475, Accuracy: 0.675
Epoch [437/500], Loss: 0.1474, Accuracy: 0.675
Epoch [438/500], Loss: 0.1474, Accuracy: 0.675
Epoch [439/500], Loss: 0.1474, Accuracy: 0.675
Epoch [440/500], Loss: 0.1473, Accuracy: 0.675
Epoch [441/500], Loss: 0.1473, Accuracy: 0.675
Epoch [442/500], Loss: 0.1473, Accuracy: 0.675
Epoch [443/500], Loss: 0.1473, Accuracy: 0.675
Epoch [444/500], Loss: 0.1472, Accuracy: 0.675
Epoch [445/500], Loss: 0.1472, Accuracy: 0.675
Epoch [446/500], Loss: 0.1472, Accuracy: 0.675
Epoch [447/500], Loss: 0.1471, Accuracy: 0.675
Epoch [448/500], Loss: 0.1471, Accuracy: 0.675
Epoch [449/500], Loss: 0.1471, Accuracy: 0.675
Epoch [450/500], Loss: 0.1470, Accuracy: 0.675
Epoch [451/500], Loss: 0.1470, Accuracy: 0.675
Epoch [452/500], Loss: 0.1470, Accuracy: 0.675
Epoch [453/500], Loss: 0.1470, Accuracy: 0.675
```

```
Epoch [454/500], Loss: 0.1469, Accuracy: 0.675
Epoch [455/500], Loss: 0.1469, Accuracy: 0.675
Epoch [456/500], Loss: 0.1469, Accuracy: 0.6725
Epoch [457/500], Loss: 0.1468, Accuracy: 0.6725
Epoch [458/500], Loss: 0.1468, Accuracy: 0.6725
Epoch [459/500], Loss: 0.1468, Accuracy: 0.6725
Epoch [460/500], Loss: 0.1467, Accuracy: 0.6725
Epoch [461/500], Loss: 0.1467, Accuracy: 0.6725
Epoch [462/500], Loss: 0.1467, Accuracy: 0.6725
Epoch [463/500], Loss: 0.1467, Accuracy: 0.6725
Epoch [464/500], Loss: 0.1466, Accuracy: 0.675
Epoch [465/500], Loss: 0.1466, Accuracy: 0.675
Epoch [466/500], Loss: 0.1466, Accuracy: 0.675
Epoch [467/500], Loss: 0.1466, Accuracy: 0.675
Epoch [468/500], Loss: 0.1465, Accuracy: 0.675
Epoch [469/500], Loss: 0.1465, Accuracy: 0.675
Epoch [470/500], Loss: 0.1465, Accuracy: 0.675
Epoch [471/500], Loss: 0.1464, Accuracy: 0.675
Epoch [472/500], Loss: 0.1464, Accuracy: 0.675
Epoch [473/500], Loss: 0.1464, Accuracy: 0.675
Epoch [474/500], Loss: 0.1464, Accuracy: 0.675
Epoch [475/500], Loss: 0.1463, Accuracy: 0.675
Epoch [476/500], Loss: 0.1463, Accuracy: 0.675
Epoch [477/500], Loss: 0.1463, Accuracy: 0.675
Epoch [478/500], Loss: 0.1463, Accuracy: 0.675
Epoch [479/500], Loss: 0.1462, Accuracy: 0.675
Epoch [480/500], Loss: 0.1462, Accuracy: 0.675
Epoch [481/500], Loss: 0.1462, Accuracy: 0.675
Epoch [482/500], Loss: 0.1462, Accuracy: 0.675
Epoch [483/500], Loss: 0.1461, Accuracy: 0.675
Epoch [484/500], Loss: 0.1461, Accuracy: 0.675
Epoch [485/500], Loss: 0.1461, Accuracy: 0.6725
Epoch [486/500], Loss: 0.1460, Accuracy: 0.6725
Epoch [487/500], Loss: 0.1460, Accuracy: 0.6725
Epoch [488/500], Loss: 0.1460, Accuracy: 0.6725
Epoch [489/500], Loss: 0.1460, Accuracy: 0.6725
Epoch [490/500], Loss: 0.1459, Accuracy: 0.6725
Epoch [491/500], Loss: 0.1459, Accuracy: 0.6725
Epoch [492/500], Loss: 0.1459, Accuracy: 0.6725
Epoch [493/500], Loss: 0.1459, Accuracy: 0.6725
Epoch [494/500], Loss: 0.1458, Accuracy: 0.6725
Epoch [495/500], Loss: 0.1458, Accuracy: 0.6725
Epoch [496/500], Loss: 0.1458, Accuracy: 0.6725
Epoch [497/500], Loss: 0.1458, Accuracy: 0.6725
Epoch [498/500], Loss: 0.1458, Accuracy: 0.6725
Epoch [499/500], Loss: 0.1457, Accuracy: 0.675
Epoch [500/500], Loss: 0.1457, Accuracy: 0.675
```

[30]:
```
visualize(sample,target, model)
```




**Here is an example: the green line is the line of the decision boundary. You should draw the linear decision boundary like this.**

**Given Example: Weights and Bias** You can use weight and bias attributes of your model to find the equation of the decision boundary.

[31]:
```
print("Weights: \n",model.linear.weight)
print("Bias: \n",model.linear.bias)
```

```
Weights:
 Parameter containing:
tensor([[0.0271, 0.6808]], requires_grad=True)
Bias:
 Parameter containing:
tensor([0.4992], requires_grad=True)
```

### 1.2.3 Logistic Regression

[41]:
```
import seaborn as sns
import numpy as np
import matplotlib
from tqdm import tqdm
import torch
from sklearn.datasets import make_classification
import matplotlib.pyplot as plt
X_tensor = torch.from_numpy(sample).float()
y_tensor = torch.from_numpy(target).float()

class LogisticRegression_class(nn.Module):
    def __init__(self, input_size):
        super(LogisticRegression_class, self).__init__()
        self.linear = nn.Linear(input_size, 1)

    def forward(self, x):
        outputs = torch.sigmoid(self.linear(x))
        return outputs

epochs = 45
# input_dim = 2 # Two inputs x1 and x2
# output_dim = 1 # Two possible outputs
# learning_rate = 0.001

LogisticRegression_model = LogisticRegression_class(X_tensor.shape[1])
criterion = nn.BCELoss()

losses = []
losses_test = []
Iterations = []
iter = 0
learning_rate = 0.01
for epoch in range(epochs):
    y_pred = LogisticRegression_model(X_tensor)
    y_pred = y_pred.reshape(y_tensor.shape)


    loss = criterion(y_pred, y_tensor)


    output = torch.where(y_pred>0.5, 1.0,0.0)

    acc = accuracy_score(y_tensor, output)
    optimizer = optim.Adam(LogisticRegression_model.parameters(),␣
 ↪lr=learning_rate)
```

```
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

    print(f'Epoch [{epoch+1}/{num_epochs}], Loss: {loss.item():.4f}, Accuracy:␣
↪{acc}')
```

```
Epoch [1/500], Loss: 0.6946, Accuracy: 0.5
Epoch [2/500], Loss: 0.6914, Accuracy: 0.5
Epoch [3/500], Loss: 0.6883, Accuracy: 0.5
Epoch [4/500], Loss: 0.6853, Accuracy: 0.5
Epoch [5/500], Loss: 0.6823, Accuracy: 0.5
Epoch [6/500], Loss: 0.6793, Accuracy: 0.5
Epoch [7/500], Loss: 0.6764, Accuracy: 0.5
Epoch [8/500], Loss: 0.6734, Accuracy: 0.5
Epoch [9/500], Loss: 0.6706, Accuracy: 0.5
Epoch [10/500], Loss: 0.6678, Accuracy: 0.5
Epoch [11/500], Loss: 0.6650, Accuracy: 0.5
Epoch [12/500], Loss: 0.6622, Accuracy: 0.5
Epoch [13/500], Loss: 0.6595, Accuracy: 0.5
Epoch [14/500], Loss: 0.6568, Accuracy: 0.5
Epoch [15/500], Loss: 0.6542, Accuracy: 0.5
Epoch [16/500], Loss: 0.6516, Accuracy: 0.5
Epoch [17/500], Loss: 0.6491, Accuracy: 0.5
Epoch [18/500], Loss: 0.6466, Accuracy: 0.5
Epoch [19/500], Loss: 0.6441, Accuracy: 0.5
Epoch [20/500], Loss: 0.6416, Accuracy: 0.5075
Epoch [21/500], Loss: 0.6393, Accuracy: 0.51
Epoch [22/500], Loss: 0.6369, Accuracy: 0.515
Epoch [23/500], Loss: 0.6346, Accuracy: 0.5325
Epoch [24/500], Loss: 0.6323, Accuracy: 0.5425
Epoch [25/500], Loss: 0.6301, Accuracy: 0.56
Epoch [26/500], Loss: 0.6279, Accuracy: 0.5775
Epoch [27/500], Loss: 0.6257, Accuracy: 0.605
Epoch [28/500], Loss: 0.6236, Accuracy: 0.625
Epoch [29/500], Loss: 0.6216, Accuracy: 0.64
Epoch [30/500], Loss: 0.6195, Accuracy: 0.6575
Epoch [31/500], Loss: 0.6175, Accuracy: 0.675
Epoch [32/500], Loss: 0.6156, Accuracy: 0.6825
Epoch [33/500], Loss: 0.6137, Accuracy: 0.695
Epoch [34/500], Loss: 0.6118, Accuracy: 0.7125
Epoch [35/500], Loss: 0.6100, Accuracy: 0.7225
Epoch [36/500], Loss: 0.6082, Accuracy: 0.725
Epoch [37/500], Loss: 0.6065, Accuracy: 0.7325
Epoch [38/500], Loss: 0.6048, Accuracy: 0.745
Epoch [39/500], Loss: 0.6031, Accuracy: 0.755
Epoch [40/500], Loss: 0.6015, Accuracy: 0.76
```

```
Epoch [41/500], Loss: 0.5999, Accuracy: 0.765
Epoch [42/500], Loss: 0.5983, Accuracy: 0.7575
Epoch [43/500], Loss: 0.5968, Accuracy: 0.7575
Epoch [44/500], Loss: 0.5954, Accuracy: 0.7625
Epoch [45/500], Loss: 0.5940, Accuracy: 0.7675
```

#### 1.2.4 The accuracy of the model changes every time I run it because of the random weight initialization. To confirm that I have reached the accuracy, I have saved my models and included their files in my submission. I load them from here and get their accuracy on the data once again

[42]:
```
#torch.save(LogisticRegression_model, 'logistic_regression75.pth')
```

[46]:
```
LogisticRegression_loaded = torch.load('logistic_regression75.pth')
```

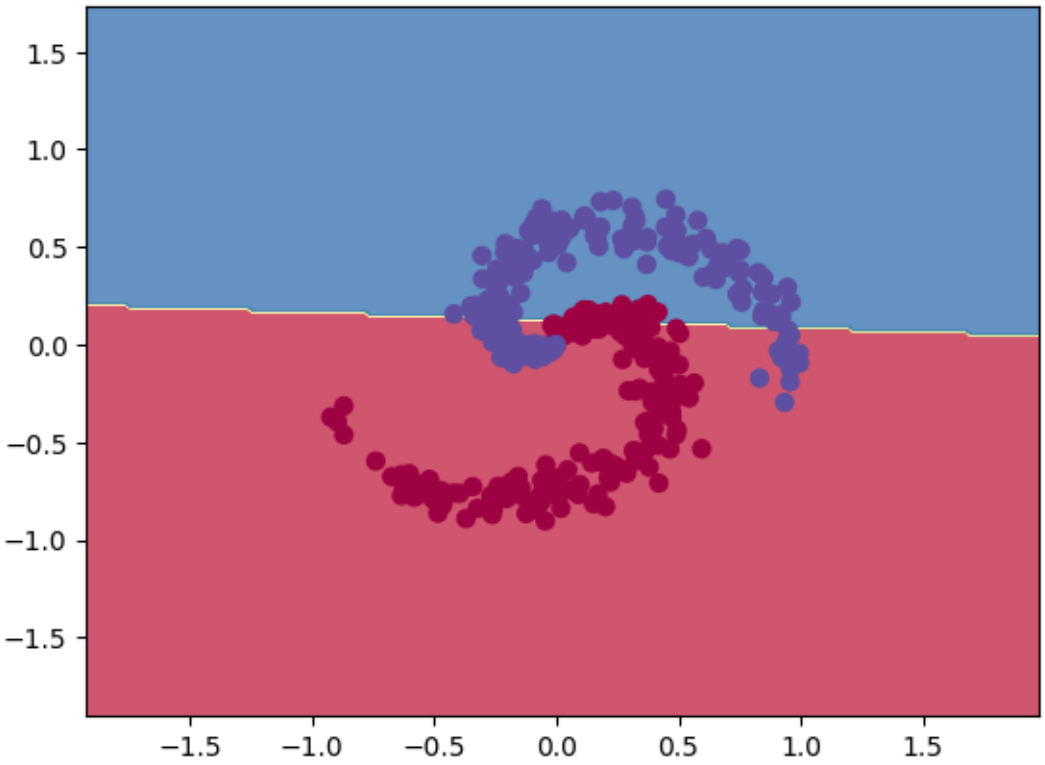
[47]:
```
LogisticRegression_loaded.eval()
```

[47]:
```
LogisticRegression_class(
  (linear): Linear(in_features=2, out_features=1, bias=True)
)
```

[52]:
```
y_pred = LogisticRegression_loaded(X_tensor)
output = torch.where(y_pred>0.5, 1.0,0.0)
acc = accuracy_score(y_tensor, output)
final_loss = loss.item()
print("accuracy: "+str(acc)+"  loss: "+str(final_loss))
```

```
accuracy: 0.755  loss: 0.5939655303955078
```

[53]:
```
visualize(sample, target, LogisticRegression)
```

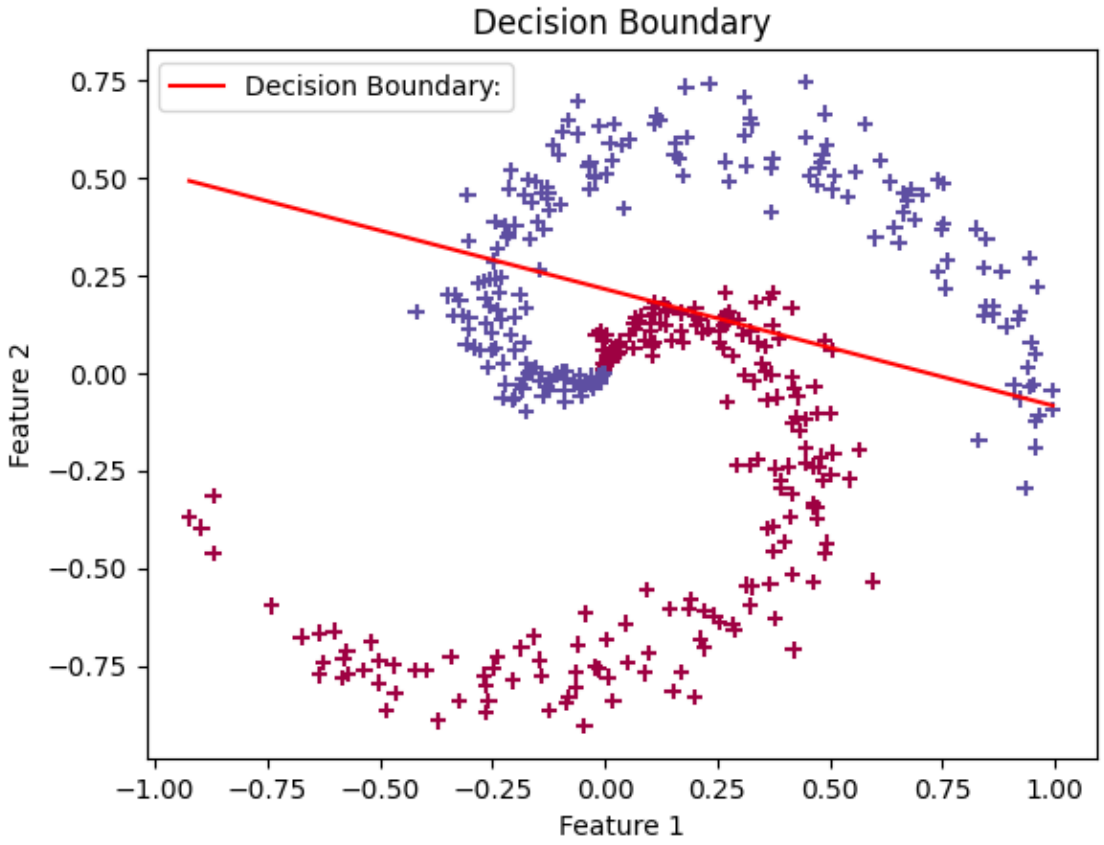



### 1.3 visualization:

[55]:
```
coefficients = LogisticRegression_loaded.linear.weight.data
intercept = LogisticRegression_loaded.linear.bias.data
import matplotlib.pyplot as plt
X = sample
y = target
y_pred = output
# plt.scatter(X[:, 0], X[:, 1], c=y_pred, cmap='viridis')
plt.scatter(X[:, 0], X[:, 1], c=y, cmap=plt.cm.Spectral,  marker='+') #coolwarm
# Plot the decision boundary
x1 = np.linspace(np.min(X[:, 0]), np.max(X[:, 0]), 100)
x2 = -(coefficients[0][0]*x1 + intercept) / coefficients[0][1]
equation = f"x2 = -({coefficients[0][0]}*x1 + {intercept}) /␣
 ↪{coefficients[0][1]}"

plt.plot(x1, x2, 'r-', label='Decision Boundary:')

plt.legend()
plt.xlabel('Feature 1')
plt.ylabel('Feature 2')
plt.title('Decision Boundary')
```

```
print(equation)
plt.show()
```

```
x2 = -(0.26430854201316833*x1 + tensor([-0.1892])) / 0.8816277384757996
```




[ ]:

[ ]:

### 1.3.1 Deep Linear Neural Network

[59]:
```
import numpy as np
import torch
import torch.nn as nn
import torch.optim as optim
import torch.nn.functional as F


X = sample
y = target
```

```
X = torch.tensor(X, dtype=torch.float32)
y = torch.tensor(y, dtype=torch.float32).reshape(-1, 1)


class LinearNN(nn.Module):
    def __init__(self, input_size, hidden_sizes, num_classes):
        super(LinearNN, self).__init__()
        self.layers = nn.ModuleList()
        self.layers.append(nn.Linear(input_size, hidden_sizes[0], bias = True))

        # Add hidden layers
        for i in range(1, len(hidden_sizes)):
            self.layers.append(nn.Linear(hidden_sizes[i-1], hidden_sizes[i],␣
 ↪bias = True))

        self.layers.append(nn.Linear(hidden_sizes[-1], num_classes, bias = True))
        self.sigmoid = nn.Sigmoid()


    def forward(self, x):
        for layer in self.layers:
            x = layer(x)
            x = self.sigmoid(x)
        return x


input_size = 2
hidden_sizes = [ 32,8, 4]
num_classes = 2
LinearNN_model = LinearNN(input_size, hidden_sizes, num_classes)
print(LinearNN_model)

loss_fn   = nn.BCELoss()  # binary cross entropy
optimizer = optim.SGD(LinearNN_model.parameters(), lr=0.01)

n_epochs = 500000


for epoch in range(n_epochs):
    y_pred = LinearNN_model(X)
    acc = (y_pred.round() == y).float().mean()
    loss = loss_fn(y_pred[:, 0].unsqueeze(1) , y)
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
    if (epoch%10000==0):
```

```
        print(f'Epoch [{epoch+1}/{n_epochs}], Loss: {loss.item():.4f}, Accuracy:␣
↪{acc}')


y_pred = LinearNN_model(X)
accuracy = (y_pred.round() == y).float().mean()
print(f"Accuracy {accuracy}")


predictions = (LinearNN_model(X) > 0.5).int()
```

```
LinearNN(
  (layers): ModuleList(
    (0): Linear(in_features=2, out_features=32, bias=True)
    (1): Linear(in_features=32, out_features=8, bias=True)
    (2): Linear(in_features=8, out_features=4, bias=True)
    (3): Linear(in_features=4, out_features=2, bias=True)
  )
  (sigmoid): Sigmoid()
)
Epoch [1/500000], Loss: 0.6932, Accuracy: 0.5
Epoch [10001/500000], Loss: 0.6927, Accuracy: 0.6462500095367432
Epoch [20001/500000], Loss: 0.6918, Accuracy: 0.6349999904632568
Epoch [30001/500000], Loss: 0.6887, Accuracy: 0.6112499833106995
Epoch [40001/500000], Loss: 0.6624, Accuracy: 0.5987499952316284
Epoch [50001/500000], Loss: 0.4349, Accuracy: 0.59375
Epoch [60001/500000], Loss: 0.4174, Accuracy: 0.5962499976158142
Epoch [70001/500000], Loss: 0.4114, Accuracy: 0.6000000238418579
Epoch [80001/500000], Loss: 0.4049, Accuracy: 0.6012499928474426
Epoch [90001/500000], Loss: 0.3925, Accuracy: 0.6012499928474426
Epoch [100001/500000], Loss: 0.3618, Accuracy: 0.6287500262260437
Epoch [110001/500000], Loss: 0.2470, Accuracy: 0.699999988079071
Epoch [120001/500000], Loss: 0.1312, Accuracy: 0.7275000214576721
Epoch [130001/500000], Loss: 0.0863, Accuracy: 0.7337499856948853
Epoch [140001/500000], Loss: 0.0662, Accuracy: 0.737500011920929
Epoch [150001/500000], Loss: 0.0546, Accuracy: 0.7400000095367432
Epoch [160001/500000], Loss: 0.0474, Accuracy: 0.7400000095367432
Epoch [170001/500000], Loss: 0.0421, Accuracy: 0.7425000071525574
Epoch [180001/500000], Loss: 0.0377, Accuracy: 0.7450000047683716
Epoch [190001/500000], Loss: 0.0336, Accuracy: 0.7450000047683716
Epoch [200001/500000], Loss: 0.0298, Accuracy: 0.7450000047683716
Epoch [210001/500000], Loss: 0.0264, Accuracy: 0.7450000047683716
Epoch [220001/500000], Loss: 0.0234, Accuracy: 0.7475000023841858
Epoch [230001/500000], Loss: 0.0209, Accuracy: 0.7475000023841858
Epoch [240001/500000], Loss: 0.0189, Accuracy: 0.7487499713897705
Epoch [250001/500000], Loss: 0.0172, Accuracy: 0.7487499713897705
Epoch [260001/500000], Loss: 0.0158, Accuracy: 0.7487499713897705
Epoch [270001/500000], Loss: 0.0147, Accuracy: 0.7487499713897705
```

```
Epoch [280001/500000], Loss: 0.0137, Accuracy: 0.7487499713897705
Epoch [290001/500000], Loss: 0.0129, Accuracy: 0.7487499713897705
Epoch [300001/500000], Loss: 0.0122, Accuracy: 0.7487499713897705
Epoch [310001/500000], Loss: 0.0117, Accuracy: 0.7487499713897705
Epoch [320001/500000], Loss: 0.0112, Accuracy: 0.7487499713897705
Epoch [330001/500000], Loss: 0.0107, Accuracy: 0.7487499713897705
Epoch [340001/500000], Loss: 0.0103, Accuracy: 0.7487499713897705
Epoch [350001/500000], Loss: 0.0100, Accuracy: 0.7487499713897705
Epoch [360001/500000], Loss: 0.0097, Accuracy: 0.7487499713897705
Epoch [370001/500000], Loss: 0.0094, Accuracy: 0.7487499713897705
Epoch [380001/500000], Loss: 0.0092, Accuracy: 0.7487499713897705
Epoch [390001/500000], Loss: 0.0090, Accuracy: 0.7487499713897705
Epoch [400001/500000], Loss: 0.0088, Accuracy: 0.7487499713897705
Epoch [410001/500000], Loss: 0.0086, Accuracy: 0.7487499713897705
Epoch [420001/500000], Loss: 0.0085, Accuracy: 0.7487499713897705
Epoch [430001/500000], Loss: 0.0083, Accuracy: 0.7487499713897705
Epoch [440001/500000], Loss: 0.0082, Accuracy: 0.7487499713897705
Epoch [450001/500000], Loss: 0.0080, Accuracy: 0.7487499713897705
Epoch [460001/500000], Loss: 0.0079, Accuracy: 0.7487499713897705
Epoch [470001/500000], Loss: 0.0078, Accuracy: 0.7487499713897705
Epoch [480001/500000], Loss: 0.0077, Accuracy: 0.7487499713897705
Epoch [490001/500000], Loss: 0.0076, Accuracy: 0.7487499713897705
Accuracy 0.7487499713897705
```

[60]:
```python
print(loss.item())
```

```
0.007498025428503752
```

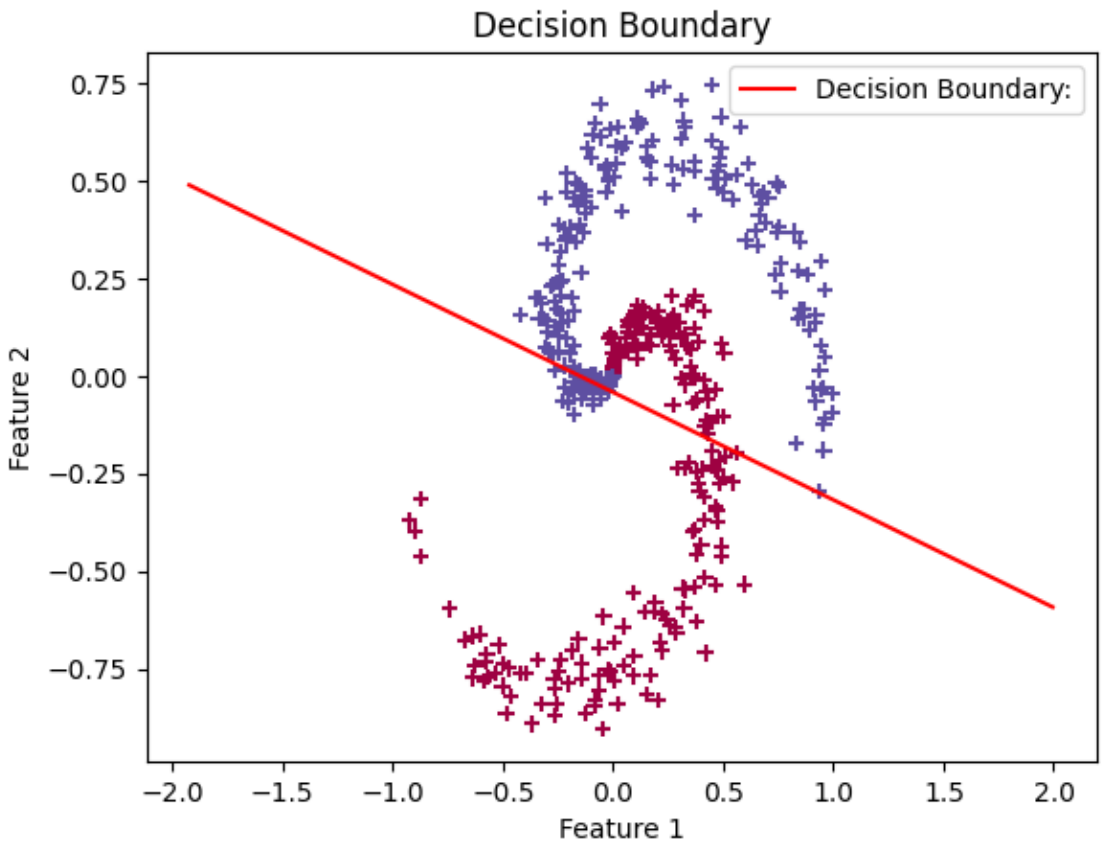
[63]:
```python
#torch.save(LinearNN_model, 'linearNN74.pth')
LinearNN_loaded = torch.load('linearNN74.pth')
LinearNN_loaded.eval()
y_pred = LinearNN_loaded(X)
accuracy = (y_pred.round() == y).float().mean()
print(f"Accuracy {accuracy}")
print(f"loss {loss.item()}")
```

```
Accuracy 0.7487499713897705
loss 0.007498025428503752
```

[104]:
```python
# for i in range(4):
#     layer = LinearNN_loaded.layers[i]
#     if hasattr(layer, 'weight'):
#         print(f"self.layers[{i}].weight.data = {layer.weight.data}")
```

### 1.4 Visualization:

[67]:
```
X = sample
y = target

# plt.scatter(X[:, 0], X[:, 1], c=y_pred, cmap='viridis')
x1_min, x1_max = sample[:, 0].min() - 1, sample[:, 0].max() + 1
x2_min, x2_max = sample[:, 1].min() - 1, sample[:, 1].max() + 1
plt.scatter(X[:, 0], X[:, 1], c=y, cmap=plt.cm.Spectral,  marker='+') #coolwarm

w = LinearNN_loaded.layers[-1].weight.data
b = LinearNN_loaded.layers[-1].bias.data
x_plot = np.array([x1_min, x1_max])
#x_plot = np.linspace(np.min(X[:, 0]), np.max(X[:, 0]), 100)
y_plot = -(w[0,0] * x_plot + b) / w[0,1]
#plt.plot(x_plot, y_plot, 'k-')
plt.plot(x_plot, y_plot, 'r-', label='Decision Boundary:')

plt.legend()
plt.xlabel('Feature 1')
plt.ylabel('Feature 2')
plt.title('Decision Boundary')
# print(equation)
plt.show()
print(f"y = -({w[0, 0]} * x + {b}) / {w[0, 1]}")
print(f"y = {y_plot}")
```




```
y = -(3.1050491333007812 * x + tensor([ 1.1808, -0.3879])) / 9.809708595275879
y = tensor([ 0.4884, -0.5921], dtype=torch.float64)
```

[ ]:

[ ]:

[ ]:

[ ]:

[ ]:

### 1.4.1 Deep Neural Network

[110]:
```
import numpy as np
import torch
import torch.nn as nn
import torch.optim as optim
import torch.nn.functional as F
```

```
X = sample
y = target

X = torch.tensor(X, dtype=torch.float32)
y = torch.tensor(y, dtype=torch.float32).reshape(-1, 1)

class NonlinearNN(nn.Module):
    def __init__(self, input_size, hidden_sizes, num_classes):
        super(NonlinearNN, self).__init__()
        self.layers = nn.ModuleList()
        self.layers.append(nn.Linear(input_size, hidden_sizes[0], bias=True))


        for i in range(1, len(hidden_sizes)):
            self.layers.append(nn.Linear(hidden_sizes[i-1], hidden_sizes[i],␣
 ↪bias=True))
            self.layers.append(nn.ReLU())

        self.layers.append(nn.Linear(hidden_sizes[-1], num_classes, bias=True))
        self.sigmoid = nn.Sigmoid()

    def forward(self, x):
        for layer in self.layers:
            x = layer(x)
        x = self.sigmoid(x)
        return x

input_size = 2
hidden_sizes = [8,23,30,62, 16,8, 4]
num_classes = 2
NonlinearNN_model = NonlinearNN(input_size, hidden_sizes, num_classes)
print(NonlinearNN_model)


loss_fn = nn.BCELoss() # binary cross-entropy loss_fn = nn.BCELoss() nn.MSELoss()
optimizer = optim.SGD(NonlinearNN_model.parameters(), lr=0.01)

#n_epochs = 1000000
total_samples = y.size(0)
n_misclassifications = 0
acc =0
epoch=0
while(acc<0.999):
#for epoch in range(n_epochs):
    y_pred = NonlinearNN_model(X)
    acc = (y_pred.round() == y).float().mean()
```

```
    loss = loss_fn(y_pred[:, 0].unsqueeze(1), y)
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
    if (epoch % 10000 == 0):
        print(f'Epoch [{epoch+1}/{n_epochs}], Loss: {loss.item():.4f}, Accuracy:␣
↪{acc}')
    epoch = epoch +1

y_pred = NonlinearNN_model(X)
accuracy = (y_pred.round() == y).float().mean()
print(f"Accuracy: {accuracy}")


predictions = (NonlinearNN_model(X) > 0.5).int()
```

```
NonlinearNN(
  (layers): ModuleList(
    (0): Linear(in_features=2, out_features=8, bias=True)
    (1): Linear(in_features=8, out_features=23, bias=True)
    (2): ReLU()
    (3): Linear(in_features=23, out_features=30, bias=True)
    (4): ReLU()
    (5): Linear(in_features=30, out_features=62, bias=True)
    (6): ReLU()
    (7): Linear(in_features=62, out_features=16, bias=True)
    (8): ReLU()
    (9): Linear(in_features=16, out_features=8, bias=True)
    (10): ReLU()
    (11): Linear(in_features=8, out_features=4, bias=True)
    (12): ReLU()
    (13): Linear(in_features=4, out_features=2, bias=True)
  )
  (sigmoid): Sigmoid()
)
Epoch [1/500000], Loss: 0.7191, Accuracy: 0.5
Epoch [10001/500000], Loss: 0.6479, Accuracy: 0.5950000286102295
Epoch [20001/500000], Loss: 0.0090, Accuracy: 0.9900000095367432
Epoch [30001/500000], Loss: 0.0071, Accuracy: 0.9950000047683716
Epoch [40001/500000], Loss: 0.0062, Accuracy: 0.9950000047683716
Epoch [50001/500000], Loss: 0.0057, Accuracy: 0.9950000047683716
Epoch [60001/500000], Loss: 0.0054, Accuracy: 0.9950000047683716
Epoch [70001/500000], Loss: 0.0051, Accuracy: 0.9950000047683716
Epoch [80001/500000], Loss: 0.0049, Accuracy: 0.9950000047683716
Epoch [90001/500000], Loss: 0.0047, Accuracy: 0.9950000047683716
Epoch [100001/500000], Loss: 0.0046, Accuracy: 0.9950000047683716
Epoch [110001/500000], Loss: 0.0045, Accuracy: 0.9950000047683716
Epoch [120001/500000], Loss: 0.0044, Accuracy: 0.9950000047683716
```

```
Epoch [130001/500000], Loss: 0.0043, Accuracy: 0.9950000047683716
Epoch [140001/500000], Loss: 0.0043, Accuracy: 0.9950000047683716
Epoch [150001/500000], Loss: 0.0042, Accuracy: 0.9950000047683716
Epoch [160001/500000], Loss: 0.0042, Accuracy: 0.9950000047683716
Epoch [170001/500000], Loss: 0.0042, Accuracy: 0.9950000047683716
Epoch [180001/500000], Loss: 0.0041, Accuracy: 0.9950000047683716
Epoch [190001/500000], Loss: 0.0041, Accuracy: 0.9950000047683716
Epoch [200001/500000], Loss: 0.0040, Accuracy: 0.9950000047683716
Epoch [210001/500000], Loss: 0.0040, Accuracy: 0.9950000047683716
Epoch [220001/500000], Loss: 0.0040, Accuracy: 0.9950000047683716
Epoch [230001/500000], Loss: 0.0040, Accuracy: 0.9950000047683716
Epoch [240001/500000], Loss: 0.0040, Accuracy: 0.9950000047683716
Epoch [250001/500000], Loss: 0.0039, Accuracy: 0.9950000047683716
Epoch [260001/500000], Loss: 0.0039, Accuracy: 0.9950000047683716
Epoch [270001/500000], Loss: 0.0039, Accuracy: 0.9950000047683716
Epoch [280001/500000], Loss: 0.0039, Accuracy: 0.9950000047683716
Epoch [290001/500000], Loss: 0.0039, Accuracy: 0.9950000047683716
Epoch [300001/500000], Loss: 0.0039, Accuracy: 0.9950000047683716
Epoch [310001/500000], Loss: 0.0039, Accuracy: 0.9950000047683716
Epoch [320001/500000], Loss: 0.0039, Accuracy: 0.9950000047683716
Epoch [330001/500000], Loss: 0.0038, Accuracy: 0.9950000047683716
Epoch [340001/500000], Loss: 0.0038, Accuracy: 0.9950000047683716
Epoch [350001/500000], Loss: 0.0038, Accuracy: 0.9950000047683716
Epoch [360001/500000], Loss: 0.0038, Accuracy: 0.9950000047683716
Epoch [370001/500000], Loss: 0.0038, Accuracy: 0.9950000047683716
Epoch [380001/500000], Loss: 0.0038, Accuracy: 0.9950000047683716
Epoch [390001/500000], Loss: 0.0038, Accuracy: 0.9950000047683716
Epoch [400001/500000], Loss: 0.0038, Accuracy: 0.9950000047683716
Epoch [410001/500000], Loss: 0.0038, Accuracy: 0.9950000047683716
Epoch [420001/500000], Loss: 0.0038, Accuracy: 0.9950000047683716
Epoch [430001/500000], Loss: 0.0038, Accuracy: 0.9950000047683716
Epoch [440001/500000], Loss: 0.0038, Accuracy: 0.9950000047683716
Epoch [450001/500000], Loss: 0.0037, Accuracy: 0.9950000047683716
Epoch [460001/500000], Loss: 0.0037, Accuracy: 0.9950000047683716
Epoch [470001/500000], Loss: 0.0037, Accuracy: 0.9950000047683716
Epoch [480001/500000], Loss: 0.0037, Accuracy: 0.9950000047683716
Epoch [490001/500000], Loss: 0.0037, Accuracy: 0.9950000047683716
Epoch [500001/500000], Loss: 0.0037, Accuracy: 0.9950000047683716
Epoch [510001/500000], Loss: 0.0037, Accuracy: 0.9950000047683716
Epoch [520001/500000], Loss: 0.0037, Accuracy: 0.9950000047683716
Epoch [530001/500000], Loss: 0.0037, Accuracy: 0.9950000047683716
Epoch [540001/500000], Loss: 0.0037, Accuracy: 0.9950000047683716
Epoch [550001/500000], Loss: 0.0037, Accuracy: 0.9950000047683716
Epoch [560001/500000], Loss: 0.0037, Accuracy: 0.9950000047683716
Epoch [570001/500000], Loss: 0.0037, Accuracy: 0.9950000047683716
Epoch [580001/500000], Loss: 0.0037, Accuracy: 0.9950000047683716
Epoch [590001/500000], Loss: 0.0037, Accuracy: 0.9950000047683716
Epoch [600001/500000], Loss: 0.0037, Accuracy: 0.9950000047683716
```

```
Epoch [610001/500000], Loss: 0.0037, Accuracy: 0.9950000047683716
Epoch [620001/500000], Loss: 0.0037, Accuracy: 0.9950000047683716
Epoch [630001/500000], Loss: 0.0037, Accuracy: 0.9950000047683716
Epoch [640001/500000], Loss: 0.0037, Accuracy: 0.9950000047683716
Epoch [650001/500000], Loss: 0.0037, Accuracy: 0.9950000047683716
Epoch [660001/500000], Loss: 0.0037, Accuracy: 0.9950000047683716
Epoch [670001/500000], Loss: 0.0037, Accuracy: 0.9950000047683716
Epoch [680001/500000], Loss: 0.0037, Accuracy: 0.9950000047683716
Epoch [690001/500000], Loss: 0.0037, Accuracy: 0.9950000047683716
Epoch [700001/500000], Loss: 0.0037, Accuracy: 0.9950000047683716
Epoch [710001/500000], Loss: 0.0037, Accuracy: 0.9950000047683716
```

```
---------------------------------------------------------------------------
KeyboardInterrupt                         Traceback (most recent call last)
Cell In[110], line 50
     47 epoch=0
     48 while(acc<0.999):
     49 #for epoch in range(n_epochs):
---> 50     y_pred = NonlinearNN_model(X)
     51     acc = (y_pred.round() == y).float().mean()
     52     loss = loss_fn(y_pred[:, 0].unsqueeze(1), y)

File ~/miniconda3/lib/python3.11/site-packages/torch/nn/modules/module.py:1518, in␣
 ↪Module._wrapped_call_impl(self, *args, **kwargs)
   1516     return self._compiled_call_impl(*args, **kwargs)  # type: ignore[misc
   1517 else:
-> 1518     return self._call_impl(*args, **kwargs)

File ~/miniconda3/lib/python3.11/site-packages/torch/nn/modules/module.py:1527, in␣
 ↪Module._call_impl(self, *args, **kwargs)
   1522 # If we don't have any hooks, we want to skip the rest of the logic in
   1523 # this function, and just call forward.
   1524 if not (self._backward_hooks or self._backward_pre_hooks or self.
 ↪_forward_hooks or self._forward_pre_hooks
   1525         or _global_backward_pre_hooks or _global_backward_hooks
   1526         or _global_forward_hooks or _global_forward_pre_hooks):
-> 1527     return forward_call(*args, **kwargs)
   1529 try:
   1530     result = None

Cell In[110], line 29, in NonlinearNN.forward(self, x)
     27 def forward(self, x):
     28     for layer in self.layers:
---> 29         x = layer(x)
     30     x = self.sigmoid(x)
     31     return x
```

```
File ~/miniconda3/lib/python3.11/site-packages/torch/nn/modules/module.py:1518, i
↪Module._wrapped_call_impl(self, *args, **kwargs)
   1516     return self._compiled_call_impl(*args, **kwargs)  # type: ignore[misc
   1517 else:
-> 1518     return self._call_impl(*args, **kwargs)

File ~/miniconda3/lib/python3.11/site-packages/torch/nn/modules/module.py:1527, i
↪Module._call_impl(self, *args, **kwargs)
   1522 # If we don't have any hooks, we want to skip the rest of the logic in
   1523 # this function, and just call forward.
   1524 if not (self._backward_hooks or self._backward_pre_hooks or self.
↪_forward_hooks or self._forward_pre_hooks
   1525         or _global_backward_pre_hooks or _global_backward_hooks
   1526         or _global_forward_hooks or _global_forward_pre_hooks):
-> 1527     return forward_call(*args, **kwargs)
   1529 try:
   1530     result = None

File ~/miniconda3/lib/python3.11/site-packages/torch/nn/modules/linear.py:114, in
↪Linear.forward(self, input)
    113 def forward(self, input: Tensor) -> Tensor:
--> 114     return F.linear(input, self.weight, self.bias)

KeyboardInterrupt:
```

## 2 I reached 99.5 percent accuracy!

[115]:
```
#torch.save(NonlinearNN_model, 'NonlinearNN_model99.5.pth')
NonLinearNN_loaded = torch.load('NonlinearNN_model99.5.pth')
NonLinearNN_loaded.eval()
y_pred = NonLinearNN_loaded(X)
accuracy = (y_pred.round() == y).float().mean()
print(f"Accuracy {accuracy}")
print(f"loss {loss.item()}")
```

```
Accuracy 0.9950000047683716
loss 0.003655920037999749
```

[ ]:
```

```

[116]:
```
for i in range(len(hidden_sizes)+7):
    layer = NonLinearNN_loaded.layers[i]
    if hasattr(layer, 'weight'):
        print(f"self.layers[{i}].weight.data = {layer.weight.data}")
```

```
self.layers[0].weight.data = tensor([[ 0.5564,  1.0073],
```

```
        [ 1.4956,  1.8406],
        [-0.7261, -0.5969],
        [ 0.6665,  1.6432],
        [-0.9924,  0.4938],
        [ 0.2678, -0.0455],
        [ 1.3064,  0.7571],
        [ 0.8253,  0.5619]])
self.layers[1].weight.data = tensor([[ 2.2025e-01, -1.4237e-01, -1.3649e-01,
-8.6062e-02,  3.7924e-01,
         -2.3687e-01,  6.2361e-02, -2.3242e-01],
        [-2.5173e-02,  1.5443e-01,  2.1476e-01,  8.3543e-03, -1.0531e-01,
         -2.1696e-02,  2.2592e-02,  1.2369e-01],
        [ 2.0749e-01,  3.9302e-01,  4.6487e-03, -2.7780e-01, -3.3725e-01,
          1.9725e-01,  6.0690e-01,  4.4710e-01],
        [-1.9766e-01, -2.3589e-01, -1.4664e-01,  2.9553e-01, -3.4445e-01,
          2.8917e-01, -2.0350e-01, -2.5782e-01],
        [-3.9425e-01, -3.4172e-01,  3.8191e-01, -1.6667e-01, -4.9383e-02,
          1.2919e-01, -2.9402e-01, -4.4066e-01],
        [-3.3132e-01,  5.3761e-02, -3.3386e-01, -1.3102e-02,  5.3558e-02,
         -3.0067e-01, -3.0755e-01,  1.8714e-01],
        [-1.6927e-01,  4.8944e-01, -2.0682e-01,  4.3496e-01,  2.6433e-01,
         -2.7192e-01, -7.3337e-02,  3.6232e-02],
        [-3.4590e-01,  2.2894e-01,  7.9954e-02, -1.4792e-01,  1.8293e-01,
          1.3037e-02, -1.4984e-01,  1.4403e-01],
        [ 4.4088e-01, -8.9411e-02,  3.7693e-01, -1.4733e-03,  5.7640e-01,
          1.2495e-01, -5.2787e-01, -4.6162e-01],
        [-3.1357e-01, -8.2788e-02, -9.1722e-02, -1.4567e-01,  1.4964e-01,
          2.1057e-01,  9.3278e-02,  1.6961e-01],
        [-7.7233e-03, -1.4357e-02, -2.4869e-03,  1.2128e-01, -3.0484e-01,
          3.3464e-01, -2.6354e-01,  9.2608e-02],
        [-1.8323e-02,  3.9403e-01, -4.4302e-01,  1.9053e-01, -1.9441e-01,
          3.1056e-01, -1.5586e-01, -3.4953e-02],
        [-1.2342e-01, -2.8595e-01,  3.4046e-01, -1.0705e-01, -3.4983e-01,
          2.9882e-01,  3.0297e-01, -1.0606e-02],
        [ 2.3133e-01, -2.2908e-01, -2.5389e-01, -3.3089e-01, -3.4454e-01,
         -2.8153e-01, -2.8346e-01,  1.7235e-01],
        [ 1.0501e-01, -1.5478e-01,  7.8802e-03,  7.6198e-02, -2.6840e-01,
         -1.1940e-01,  3.2675e-02,  8.4541e-02],
        [ 2.7610e-01, -5.1470e-03,  1.1831e-01,  7.7947e-02,  3.0857e-02,
          2.9186e-01,  2.7647e-01,  3.3684e-01],
        [-3.2635e-01,  2.4210e-01, -8.3329e-02,  1.2565e-03, -2.4601e-01,
          3.7144e-01,  3.0078e-01, -1.6684e-01],
        [-3.4745e-02, -2.0092e-01, -1.9177e-01,  5.0690e-02,  1.6270e-01,
          1.5878e-01, -2.2440e-01,  3.2884e-01],
        [-2.7883e-02,  1.8947e-01, -7.6653e-02,  2.6972e-01, -3.0975e-01,
         -2.2961e-01,  1.1431e-02,  1.5699e-01],
        [-8.9004e-01, -1.4955e+00,  4.2320e-01, -1.0973e+00,  4.0314e-02,
         -1.6251e-01, -9.4786e-01, -5.6366e-01],
```

```
        [ 3.1960e-01,  4.3764e-01, -1.0699e-01,  3.5796e-01,  9.6511e-02,
         -3.4842e-02, -2.3808e-01, -2.9468e-01],
        [-1.5051e-01, -4.1880e-01, -1.1179e-01, -2.3509e-01, -3.4034e-01,
         -4.1714e-02, -9.6730e-02, -3.0545e-01],
        [ 7.3567e-02, -1.0527e-01,  1.5395e-01, -2.6221e-01, -1.8253e-02,
         -3.2172e-02, -1.7958e-01, -3.8857e-01]])
self.layers[3].weight.data = tensor([[ 2.4343e-02,  1.9653e-01,  3.3475e-01,
1.1648e-01, -8.4838e-03,
          5.4228e-02,  3.4910e-01,  5.5950e-03,  5.8601e-02,  1.0937e-03,
          1.4304e-02, -1.3993e-01,  8.3931e-02, -1.2379e-01, -1.3910e-01,
          2.1182e-01,  2.3238e-01, -2.9415e-02,  8.9859e-02,  4.1079e-01,
          2.2570e-01,  1.1362e-01,  1.5304e-01],
        [ 1.5066e-02, -1.5580e-01, -1.8884e-02, -1.0370e-01,  4.9621e-02,
         -8.1436e-02, -1.6920e-01,  9.9933e-02,  1.3396e-02, -2.0055e-01,
          1.9983e-01, -6.8605e-02, -1.2913e-01, -1.4053e-01,  5.8005e-02,
         -6.6468e-02,  1.9736e-01, -1.3492e-01, -1.7201e-01,  5.4553e-02,
         -1.6399e-01, -1.5581e-03,  1.7708e-01],
        [-1.1049e-01,  6.3897e-02,  9.3500e-02,  1.7494e-01, -1.2294e-01,
         -7.2096e-03,  1.6194e-01,  1.5716e-01,  8.9002e-02, -8.7662e-02,
         -1.1203e-01, -2.4530e-02,  1.2811e-01, -9.8660e-02,  1.0180e-01,
         -1.1821e-01, -1.1501e-01,  8.3769e-02, -1.2500e-01,  1.9487e-01,
          1.1450e-01,  2.0031e-01, -8.5387e-02],
        [ 2.9775e-03, -7.2419e-02,  1.8376e-01,  6.4141e-02,  8.2788e-02,
         -1.1413e-01,  1.0083e-01, -2.4664e-02,  4.7858e-02,  1.1269e-01,
         -2.0410e-01, -6.6184e-02, -2.1675e-03, -1.5704e-01,  1.1514e-01,
          1.3867e-02, -1.0493e-01, -1.0599e-01, -1.2323e-01, -2.1091e-01,
          3.6953e-02,  1.1743e-01,  1.1869e-01],
        [ 8.5764e-02,  1.9937e-01,  2.9121e-01, -1.3763e-01, -4.9794e-02,
          1.7612e-01,  2.8023e-01,  1.6398e-01, -5.0712e-02, -1.9597e-01,
          1.4540e-01, -1.6164e-01,  8.7679e-02,  1.7538e-01, -1.4812e-01,
         -1.2047e-01,  2.1872e-02,  1.4800e-01,  1.3704e-01,  7.1940e-03,
          5.6136e-02, -9.4005e-02,  8.1916e-02],
        [-1.8980e-02, -1.6942e-01, -5.9670e-01,  2.9927e-01,  5.3739e-01,
          6.4079e-02, -2.8977e-01, -1.2762e-01, -3.5827e-01,  1.2594e-01,
          1.4792e-01,  4.2976e-03, -9.4211e-02,  2.8348e-01, -1.9468e-03,
          2.0282e-01,  1.9051e-02, -2.4594e-02,  6.7005e-02, -5.2080e-01,
         -7.1477e-02, -1.2629e-01,  3.1362e-01],
        [-5.8028e-02,  9.9610e-02,  3.0251e-02,  3.3498e-02, -1.6058e-01,
         -1.8198e-01,  1.9956e-01,  8.5344e-02, -1.3685e-01,  2.0257e-01,
          5.2080e-02, -5.1101e-02, -1.0716e-01, -1.5501e-01,  1.5734e-01,
          9.3045e-02,  5.4303e-02, -2.6404e-02, -4.8731e-02, -1.5900e-01,
         -2.0218e-01, -2.0575e-01,  5.0743e-03],
        [-7.4483e-02, -3.8151e-03, -3.2881e-01,  1.9232e-01,  2.0996e-01,
         -6.2123e-02, -7.6089e-02, -1.4240e-01, -1.2961e-01,  9.4350e-02,
          1.2853e-01,  1.9109e-01,  1.1406e-02,  5.3473e-02,  1.8102e-01,
         -3.4088e-02, -1.2508e-01,  4.6602e-02, -8.3709e-02, -6.6095e-01,
         -1.5333e-01, -7.0075e-02,  6.3685e-02],
        [-6.2538e-03,  1.9180e-01, -6.3641e-03, -5.4449e-02,  3.8632e-01,
```

```
 -1.2161e-01,  1.5872e-01, -3.3973e-03,  2.5722e-01,  1.3925e-01,
  1.2206e-01, -1.9385e-01, -9.0096e-03,  2.2424e-01,  2.1986e-01,
 -1.2816e-01, -8.8499e-02,  2.3981e-02,  1.8219e-01,  5.5510e-01,
 -3.3323e-01,  4.3010e-01,  1.8588e-01],
[-6.0840e-02, -9.2148e-02,  6.4323e-02,  3.1886e-02, -1.9487e-01,
 -2.7368e-02, -1.0601e-01, -4.4433e-02, -2.0028e-01,  6.5495e-02,
  7.6835e-02, -4.7173e-02, -9.2459e-02,  7.2688e-02,  6.7672e-02,
  6.4842e-02,  7.2986e-02, -6.4686e-02, -2.0770e-01,  1.1243e-01,
 -1.8447e-01, -1.1924e-01,  1.0873e-01],
[ 2.8580e-02, -1.9254e-01, -1.2277e-01,  2.9283e-02, -1.6646e-01,
 -1.7678e-01,  2.3375e-01,  1.2726e-01, -7.5427e-02, -1.3028e-01,
 -1.7543e-01,  9.1689e-02, -1.5880e-01, -9.4252e-02,  9.5960e-02,
 -6.6581e-02, -1.1379e-01,  1.7997e-01, -6.1582e-02,  8.7304e-02,
  1.6754e-01, -2.0532e-01, -6.2046e-02],
[ 8.5378e-03, -1.5239e-01, -2.1645e-02, -1.2543e-01,  1.1353e-01,
  2.0570e-01, -2.2784e-03, -4.4213e-02,  1.7856e-01,  4.8287e-02,
  1.3736e-01, -1.9578e-02, -1.7188e-01,  1.1508e-01, -4.2793e-02,
  1.2371e-01,  1.5706e-01,  2.0036e-01, -1.4797e-01,  1.5122e-01,
  1.0677e-01, -7.9302e-02, -1.7607e-01],
[-9.3273e-02,  8.6993e-02, -1.9701e-01, -1.1747e-02, -1.8351e-01,
 -8.6053e-02, -3.8055e-02, -1.1333e-01, -4.6254e-02,  1.1647e-01,
  1.0948e-01,  1.3123e-01, -6.3439e-03,  1.8936e-01, -4.5022e-02,
 -1.3461e-01, -1.7526e-01,  1.1214e-01,  9.3246e-04,  5.1292e-02,
 -2.5428e-03,  1.0912e-01, -1.9040e-01],
[ 9.3535e-02,  2.4981e-02,  8.8043e-02, -5.2609e-02, -1.9105e-01,
 -1.8070e-01, -7.6215e-02, -1.2599e-01,  2.2940e-01,  8.5970e-02,
 -1.2047e-01, -2.1339e-01, -1.7499e-01, -9.1867e-03, -1.8335e-01,
 -1.4702e-01,  1.8162e-01, -7.2280e-02, -1.9509e-01,  9.1263e-02,
  1.1242e-01,  8.2211e-03, -6.7560e-02],
[-1.9419e-01,  1.3775e-02, -1.2890e-01, -1.8671e-01,  1.7808e-01,
  1.4814e-01,  2.8728e-02,  3.6204e-02, -1.0123e-01,  1.4046e-01,
  3.0918e-02, -7.4488e-02,  1.5963e-01,  1.7845e-01,  1.1005e-01,
 -1.1871e-01,  9.7446e-02, -9.8922e-02,  4.0403e-02, -1.1487e-01,
 -7.2096e-02,  9.9885e-02,  1.3282e-01],
[ 2.0911e-01,  6.9366e-02,  1.9698e-01, -8.5748e-02,  7.6386e-02,
 -1.2623e-01,  8.9226e-02,  6.8844e-02,  3.8071e-02,  1.1130e-01,
 -7.6604e-02, -9.9873e-02,  1.8202e-01, -2.0580e-01,  9.8749e-02,
 -1.4889e-01, -5.0488e-02,  6.4992e-02, -1.9762e-01,  6.0297e-02,
 -8.5445e-02, -1.9692e-01, -1.0650e-01],
[-1.0552e-01,  1.5052e-01,  4.2321e-02, -1.2150e-01,  1.2824e-02,
 -9.2868e-02,  1.9144e-01, -1.1866e-01,  1.0498e-01, -1.1533e-01,
  1.0777e-01, -1.7264e-01,  2.6062e-02, -5.7462e-02,  3.2518e-02,
 -1.1735e-01, -1.9305e-02,  1.7496e-01, -3.4868e-02,  1.2255e-01,
 -1.3618e-01,  1.4049e-01, -2.2501e-01],
[-2.0235e-01, -1.1840e-01, -8.8678e-02, -1.2082e-01,  5.6941e-02,
  1.2703e-01, -1.7520e-01,  1.0374e-02, -1.9913e-01,  1.9034e-01,
 -1.9309e-01,  1.7135e-01,  4.8396e-02, -8.4286e-02,  1.0897e-01,
 -1.1157e-01, -2.1900e-01, -1.7847e-01, -7.7904e-02, -3.0233e-01,
```

```
  1.5150e-01, -2.0306e-02,  1.6957e-01],
[ 1.5224e-01,  1.4911e-01, -1.2800e-01, -1.5214e-01, -6.8815e-02,
  1.7389e-01,  1.7027e-01,  1.7759e-02, -1.1118e-01,  8.3359e-02,
 -1.4692e-01,  1.4768e-01,  1.7385e-01,  1.0795e-01,  8.5142e-02,
 -6.7649e-03,  9.4129e-02, -1.2500e-01, -1.1808e-01,  1.7385e-01,
 -9.0960e-02,  1.2068e-01,  4.6029e-03],
[ 8.8022e-02, -4.2999e-02, -1.1760e-01, -1.1876e-01, -8.7470e-03,
 -1.4745e-01, -1.5338e-02,  1.7596e-01,  2.0071e-01,  3.9752e-02,
 -1.9810e-01, -1.1912e-01,  8.9776e-02, -1.6687e-01, -1.4811e-01,
 -1.4806e-01,  2.0885e-02,  8.8766e-02, -1.0564e-02,  1.8276e-01,
 -1.3350e-01,  1.0991e-01,  1.4833e-01],
[-1.6234e-01, -8.2089e-02, -1.8447e-01,  8.4326e-02,  6.0829e-02,
  1.1106e-01,  1.0305e-01, -1.6003e-01, -5.9223e-02,  1.8792e-01,
  2.0302e-01,  2.0074e-01,  5.8856e-02,  1.9713e-01, -1.4005e-01,
 -9.7728e-02, -1.4457e-01,  5.6746e-02,  4.1502e-02,  6.4425e-02,
 -7.3519e-02,  6.0120e-02,  1.3997e-01],
[-1.3551e-01, -1.0269e-01,  2.0426e-01,  2.9530e-01,  2.9734e-02,
 -1.2729e-01, -6.3073e-02,  1.7065e-02, -6.8374e-02,  2.0708e-01,
 -6.5090e-02,  3.7117e-02, -1.9420e-01, -6.4328e-02, -6.0634e-02,
  2.4509e-02, -1.9566e-01,  9.8407e-02,  2.1062e-01,  4.1282e-02,
  9.0563e-02,  5.1739e-02, -1.0086e-01],
[ 6.3266e-02, -1.4218e-02,  1.0210e-01,  9.6189e-04, -1.3361e-01,
 -7.2210e-02,  2.3414e-01, -1.3154e-01,  2.9485e-01,  8.2534e-02,
 -1.4470e-01, -1.7046e-01,  1.7893e-01, -1.5724e-01,  1.4176e-01,
 -4.1103e-02,  3.8164e-02,  7.1404e-02,  1.1080e-01,  5.6909e-01,
  9.6248e-02, -2.8004e-02, -1.0868e-01],
[-9.0672e-03,  7.6116e-02,  1.0306e-01,  1.4490e-01, -6.6319e-02,
  1.1194e-02, -1.4872e-02,  1.8050e-01, -1.5887e-02, -1.0437e-01,
  1.8706e-01,  1.4499e-01, -8.8756e-02, -6.9040e-02,  1.8624e-01,
 -4.7511e-02,  3.4624e-02, -1.7685e-01, -2.1018e-02,  7.9935e-02,
  2.3447e-01,  4.8869e-02, -2.0332e-01],
[ 9.2374e-03,  1.7359e-01,  2.1459e-01,  1.3070e-01, -2.9110e-01,
  6.1756e-02,  1.6207e-01,  9.9766e-02,  2.5426e-01,  1.6631e-01,
 -1.5068e-01, -2.3603e-01,  2.7023e-02, -1.2390e-01,  4.9998e-02,
  5.3357e-03, -8.8040e-02, -6.1636e-02, -4.7193e-02,  3.0646e-01,
  2.4710e-01, -1.0686e-01,  1.0248e-01],
[ 2.9595e-02,  9.6064e-02,  1.0920e-01,  1.5550e-01, -3.2983e-02,
  1.5681e-01,  3.2797e-02,  6.7613e-02, -1.2600e-01, -4.4269e-02,
  9.7356e-02,  7.2985e-02,  1.0457e-01, -7.5450e-02, -7.4941e-02,
 -1.2867e-01, -1.6995e-01, -3.3762e-02, -1.5047e-01,  1.6997e-02,
  8.0769e-02,  6.2479e-02, -5.4476e-02],
[-9.2745e-02, -5.4829e-02,  1.1211e-01,  1.2526e-01, -2.4394e-02,
 -9.9748e-02, -5.6558e-02, -7.9440e-02, -4.5870e-02,  9.8856e-02,
  2.1503e-02,  6.5915e-02,  6.0077e-02, -5.6998e-02,  1.0179e-01,
  1.5704e-01, -1.5250e-01,  1.4385e-02,  6.2930e-02, -3.6071e-01,
 -2.3667e-02, -1.5171e-01, -7.0185e-04],
[-5.2090e-02,  7.3195e-02,  1.4197e-01,  4.4852e-04,  2.0899e-01,
  1.1475e-01, -7.5711e-02,  2.0574e-01,  1.5688e-01, -1.5451e-01,
```

```
         -1.8443e-02,  6.0726e-02,  1.4035e-01,  2.0204e-02,  7.0889e-02,
          1.3616e-01,  8.1167e-02,  1.6939e-01,  4.7589e-03,  3.7221e-03,
         -2.2432e-01, -1.7286e-01,  2.1572e-02],
        [-9.5203e-02,  1.9861e-01, -1.1436e-01, -1.8429e-01,  1.3053e-01,
         -4.0851e-03, -1.2188e-01,  1.4573e-01,  2.4689e-01, -9.6027e-02,
          1.5699e-01,  1.6629e-01, -1.5715e-01, -7.9350e-02, -3.4655e-02,
         -1.5794e-01, -1.9996e-01,  4.7925e-02, -1.3083e-01,  3.4741e-02,
          6.5902e-02,  1.0214e-01, -1.1595e-01],
        [-1.0327e-01, -9.9766e-02, -3.5711e-01, -1.2387e-01,  1.1905e-01,
         -1.5655e-01, -2.8582e-01, -1.6389e-01, -3.9476e-01,  2.6198e-02,
         -6.2406e-02,  6.6853e-02,  1.4420e-01, -1.5929e-01,  1.0124e-01,
          1.1855e-01, -2.6329e-01,  4.7002e-02, -8.8417e-02, -1.1099e+00,
          3.7296e-01, -1.5559e-01, -4.5152e-03]])
self.layers[5].weight.data = tensor([[-0.0095, -0.1596,  0.0284,  ...,  0.0555,
0.1374, -0.0369],
        [-0.2111,  0.0853, -0.1740,  ...,  0.1218,  0.0667,  0.1444],
        [-0.0404, -0.0700, -0.1532,  ..., -0.0706, -0.1752,  0.1224],
        ...,
        [-0.0933, -0.0397, -0.0330,  ...,  0.0103,  0.0281, -0.0598],
        [ 0.1242,  0.0676, -0.1173,  ..., -0.1428, -0.0809,  0.0810],
        [ 0.0730, -0.0723,  0.1466,  ...,  0.1046, -0.0352, -0.1754]])
self.layers[7].weight.data = tensor([[-9.5057e-02,  2.7418e-01, -4.6729e-02,
1.1491e-01,  4.7841e-02,
          1.0273e-01, -5.2567e-02,  2.7252e-01,  3.6382e-02,  3.8144e-02,
          1.9801e-01, -1.7921e-01,  9.9850e-02,  4.1974e-03, -8.0498e-02,
          2.3876e-02,  4.5879e-02, -4.9064e-02,  4.5817e-02, -8.7131e-02,
         -7.1628e-02,  1.3153e-01, -8.5427e-02,  5.5798e-02, -1.5156e-01,
          1.5301e-01, -9.6899e-02,  9.5813e-02, -8.2031e-04,  2.2482e-01,
          2.9542e-02,  3.8397e-01,  5.0937e-02,  1.3276e-01, -9.7410e-02,
         -2.8088e-02,  3.0720e-01,  1.0350e-01, -5.6062e-03,  5.7522e-02,
         -8.9220e-02, -2.4714e-02,  8.0210e-02,  1.0026e-01,  9.1156e-02,
          7.6622e-02,  1.5476e-01,  1.8230e-02, -4.3298e-02,  3.7774e-01,
         -2.6337e-01, -4.8468e-02,  1.5802e-01,  1.7859e-01,  1.9892e-01,
          9.1020e-02,  4.8803e-01, -7.2916e-02,  3.8367e-02, -4.8667e-02,
          1.1889e-01,  7.6533e-02],
        [-9.6202e-02, -4.4332e-02,  7.2733e-02,  5.3400e-02,  1.1016e-01,
         -5.1411e-02,  1.1957e-01, -1.0433e-02,  5.7092e-03, -4.7696e-02,
          2.3861e-02, -1.0911e-01, -4.8122e-02,  9.7255e-02, -7.2582e-02,
          9.1576e-02,  4.2301e-02, -5.0502e-02, -6.6665e-02, -7.5068e-02,
          1.1739e-01, -1.0082e-01,  6.8279e-02,  3.6108e-02, -1.0621e-01,
          3.8977e-02, -1.8429e-02, -1.0586e-01, -6.4611e-02,  3.6259e-02,
         -1.0332e-01, -1.5117e-02, -7.8225e-02,  6.5009e-02, -8.0006e-02,
         -3.6393e-02,  1.1487e-01, -9.0067e-02,  5.0751e-02, -3.6587e-02,
         -9.9342e-02,  1.1223e-01,  3.9065e-02, -6.1213e-02, -2.7375e-02,
          2.8814e-02,  6.1631e-05,  5.1222e-02,  5.7068e-02,  1.5470e-02,
          6.2688e-02,  8.6755e-03, -2.3053e-02, -1.0365e-01, -5.1680e-02,
         -7.7834e-02,  9.8861e-02,  4.2250e-02,  1.0618e-01, -4.4858e-02,
         -4.4157e-02, -5.7849e-02],
```

```
[-7.0666e-02,  1.5856e-02, -9.2984e-02,  4.2669e-03, -7.3044e-02,
 -6.1085e-02, -8.8796e-04,  1.7686e-01,  3.4549e-02, -1.1899e-01,
  2.1097e-02,  2.7227e-02,  6.1979e-02, -1.1391e-01,  1.2415e-01,
 -5.5778e-02, -1.1494e-01,  1.5001e-02,  2.8100e-02,  5.9741e-02,
 -3.6631e-03,  5.8689e-02, -1.8902e-02,  8.6822e-02, -8.8262e-02,
  2.9367e-02, -9.5373e-02,  3.3836e-03, -8.7259e-02,  8.8342e-02,
 -4.0946e-02,  1.5496e-01,  3.2197e-02,  4.4135e-02, -3.0091e-02,
  3.9359e-02,  1.7179e-01, -4.5162e-02, -6.9658e-02,  3.6586e-02,
  3.5044e-02,  2.5900e-02,  6.2682e-02,  3.7373e-02,  1.1183e-01,
  1.1194e-01, -5.9007e-02,  8.2187e-02,  8.2948e-02,  1.0441e-01,
 -2.6883e-02,  4.8978e-02,  8.7095e-02,  6.2433e-02, -3.8336e-02,
  1.2222e-01,  2.0461e-01,  1.1198e-01,  3.6694e-02,  1.3299e-01,
 -1.2033e-01, -5.0285e-02],
[ 6.5919e-02,  2.1430e-01, -8.3904e-02, -3.7901e-02, -1.8877e-01,
  2.7682e-02, -3.7573e-02,  1.9079e-01,  3.9889e-02,  8.1268e-02,
  1.3926e-01, -2.4317e-01,  1.3067e-01,  4.6223e-02,  5.5465e-02,
  5.3754e-02, -1.0663e-01, -1.0179e-01, -1.2438e-01,  5.6001e-02,
  7.1708e-02,  1.7372e-01, -1.3104e-02,  1.0872e-02, -7.8522e-02,
 -1.4716e-02, -9.3741e-03, -4.0266e-02, -8.6159e-02,  2.3720e-01,
 -1.9852e-01,  3.9649e-01,  5.0475e-02, -9.9409e-02, -1.5039e-01,
  2.7566e-02,  3.2267e-01,  6.0437e-03, -2.1435e-01,  1.0420e-01,
 -5.9816e-02,  8.1211e-02,  1.2326e-01, -3.0291e-02,  3.4330e-02,
  1.5625e-02,  3.5826e-03, -1.0977e-02, -8.9166e-02,  3.0889e-01,
 -1.9004e-01, -5.6744e-02, -8.5155e-02,  1.3674e-01,  2.3881e-01,
  3.1318e-02,  2.5687e-01,  2.9037e-02,  7.4843e-02,  1.1373e-01,
 -7.2684e-02, -1.0830e-01],
[ 1.8728e-02, -7.7862e-02, -1.0957e-02,  3.9879e-02, -2.7693e-02,
 -1.0363e-01, -1.1519e-01,  3.8843e-02, -1.1306e-01, -1.1691e-01,
  1.6854e-02,  4.1190e-02,  1.6957e-03, -1.3028e-01, -1.0568e-01,
 -1.1189e-01, -2.8164e-02,  1.1819e-01,  1.0783e-01, -5.0698e-02,
  7.8447e-02, -5.8323e-02, -2.1547e-02,  1.2410e-01,  1.0800e-01,
  6.2643e-02, -5.0374e-02, -9.7538e-02,  1.1423e-01,  3.8672e-02,
 -2.4464e-02, -5.5046e-02, -1.1146e-01, -7.5621e-02, -1.0217e-01,
  9.1446e-03, -6.9754e-02,  1.0492e-01,  3.5740e-02,  1.0883e-01,
  4.1773e-02, -7.5369e-02, -9.8068e-02, -8.2504e-02, -2.1230e-02,
 -6.3786e-03, -2.1760e-02,  1.0485e-01, -6.7619e-02,  1.1116e-01,
 -1.5676e-02,  6.0888e-02,  4.7665e-02, -3.1327e-02, -5.5870e-02,
  4.5020e-02, -6.0187e-02, -1.2540e-02,  8.1254e-02, -3.6717e-02,
  5.9977e-02, -8.7067e-02],
[-6.4660e-02,  6.1494e-02,  1.0700e-01,  6.5553e-02,  1.0760e-01,
  9.2845e-02, -6.8849e-02,  2.2700e-01,  1.1248e-03,  8.9617e-02,
  1.3048e-01, -2.1890e-02, -1.0657e-01,  1.4619e-01,  5.5173e-02,
  3.7429e-02,  5.6374e-02,  6.3339e-02,  1.5618e-01,  3.1369e-02,
  7.4793e-02,  6.9802e-02, -7.5148e-02,  8.0534e-02,  6.5810e-02,
 -6.2239e-02, -1.8770e-02,  3.7327e-02,  8.8399e-02,  1.4262e-01,
  1.3426e-01,  6.7728e-02,  4.7162e-02,  1.5480e-01, -9.7648e-02,
  1.2324e-01, -6.2274e-02, -1.9190e-02,  3.3269e-03,  1.1540e-02,
  5.0358e-02, -5.7417e-02, -9.2738e-02, -5.8940e-02, -1.0401e-01,
```

```
 -4.2055e-02,  8.9134e-02,  1.2353e-01, -1.4025e-02,  9.4367e-02,
  3.9524e-02,  8.2095e-02,  1.3355e-01,  1.5650e-01,  7.5229e-02,
 -9.3485e-02,  1.5074e-01,  6.1418e-02,  3.7761e-03,  1.4173e-02,
  5.8636e-02,  1.0815e-01],
[ 6.7992e-02, -5.0099e-02, -3.3496e-02,  1.2558e-01, -8.2720e-02,
  9.4658e-02, -1.8793e-02, -7.9712e-03,  3.7819e-02, -4.4715e-03,
 -1.0518e-01,  2.8351e-02,  9.6885e-02,  9.6321e-02, -9.1101e-02,
  3.5268e-02,  9.0854e-02, -1.1456e-01,  3.0326e-02, -2.1514e-02,
  2.3122e-02,  8.7157e-02, -6.0712e-02,  5.0896e-02, -2.7031e-02,
 -8.7965e-02,  1.1860e-01,  2.2119e-03, -5.9501e-02,  1.2020e-02,
  7.6540e-02, -4.0709e-02,  6.9397e-02, -3.0613e-02, -1.7515e-02,
 -9.7703e-02,  9.9533e-03,  1.0498e-02, -7.5445e-02,  1.2258e-01,
 -4.1491e-02, -7.1377e-02, -9.3577e-02,  2.3168e-03,  1.0240e-01,
  2.9728e-02, -1.1434e-01,  1.1149e-01,  9.7017e-02, -7.9314e-02,
  1.6349e-02, -8.8376e-03, -6.1563e-02,  1.1628e-01,  6.0056e-02,
 -4.3167e-02, -1.1613e-02,  1.1761e-01, -2.9168e-02,  4.5311e-02,
  7.7718e-02, -8.5150e-02],
[-1.1496e-01,  2.5937e-01, -2.0390e-02, -3.9489e-03, -4.4041e-02,
 -7.1106e-02,  1.0144e-01,  7.8839e-02, -4.6249e-02,  6.0731e-03,
  4.7171e-02, -1.2278e-01,  8.0148e-02, -1.2435e-01, -5.0943e-02,
 -7.6361e-02,  1.0394e-02,  6.9274e-02,  4.3259e-02, -4.2331e-02,
 -7.5600e-02,  1.7336e-01,  1.3111e-02, -8.8780e-02, -1.2830e-01,
  1.5129e-01, -1.2026e-01, -1.1640e-01,  7.9996e-02,  1.2600e-01,
 -9.7111e-02,  3.6405e-01,  1.6189e-01, -8.6679e-02, -2.3820e-01,
  9.6673e-02,  1.9735e-01, -7.9821e-02, -9.6079e-02,  8.8151e-02,
  1.7310e-02,  2.9455e-02,  1.7061e-01, -5.5200e-02,  1.3782e-01,
 -5.7863e-02,  1.6897e-01, -1.0304e-01,  8.1626e-02,  2.5244e-01,
 -3.1149e-01,  7.9945e-02, -4.6623e-02,  3.7573e-02,  2.6038e-02,
  3.4975e-02,  4.0172e-01,  4.6656e-02,  8.8866e-02, -3.1671e-02,
 -5.2963e-02, -3.0603e-03],
[ 1.5017e-02,  9.8927e-02,  3.2580e-02,  3.2069e-02,  1.2339e-01,
 -7.9087e-02,  5.2328e-02,  6.2804e-02,  5.1411e-02,  1.1809e-01,
 -6.4805e-02, -1.0856e-01,  3.6936e-02,  3.5526e-02,  1.2139e-01,
  7.4850e-02,  9.7581e-02, -1.5553e-02,  1.4349e-02, -5.9876e-02,
 -6.1798e-02, -4.3026e-02,  5.2348e-02,  1.7192e-02,  1.2377e-01,
 -1.0805e-01, -7.1671e-02, -1.1257e-01,  1.1196e-01, -3.9291e-02,
 -7.4444e-02,  3.4817e-02,  7.8568e-04, -1.1162e-01, -1.2262e-03,
 -4.9472e-02, -7.0061e-02,  7.5842e-02, -9.7907e-02, -6.1858e-02,
 -1.2132e-02,  7.7258e-02,  1.1609e-02,  9.7573e-02,  8.4100e-02,
 -2.3973e-02, -5.9106e-02,  4.2150e-03,  1.1752e-01,  8.6860e-02,
  7.3106e-04,  1.1460e-03, -1.3378e-02, -7.1006e-02,  7.8716e-02,
 -1.1037e-01, -8.6705e-02, -9.3783e-02,  8.1537e-02,  8.6091e-02,
 -7.8178e-02,  3.3090e-02],
[ 1.1748e-01, -3.1508e-02,  2.0560e-03,  1.9667e-02,  2.3653e-02,
 -1.1403e-01,  1.0803e-01, -3.0835e-02,  1.8864e-02, -2.7819e-02,
  2.8743e-02,  4.0968e-02, -3.8358e-02, -8.5189e-02, -1.1085e-01,
 -6.9827e-02,  1.0320e-01, -1.0580e-02, -1.0145e-01,  1.2092e-01,
  2.7102e-02, -1.0110e-01,  1.1624e-01, -4.8501e-02, -5.8942e-03,
```

```
  6.0438e-02,  1.6385e-02,  1.0979e-01,  4.1401e-02,  5.0793e-02,
  9.2804e-02, -3.8023e-02,  4.4309e-03, -1.0003e-01,  4.5529e-02,
  4.7185e-02, -8.8711e-02, -7.5937e-02,  5.4553e-03,  7.7376e-02,
  1.1828e-01, -1.0094e-01, -2.7835e-02,  2.6105e-02,  1.2775e-03,
  4.9908e-02,  1.1488e-01, -6.5701e-02,  9.1553e-02,  3.2878e-02,
  2.8704e-02, -2.0725e-03,  5.0946e-02, -5.6019e-02,  4.0141e-03,
 -4.5749e-02, -1.0455e-01,  5.5510e-02, -3.2456e-02, -9.6239e-02,
  8.1051e-02,  7.8669e-02],
[ 1.1325e-01, -2.3765e-01, -9.8263e-02,  8.4119e-02,  2.2731e-01,
 -7.3940e-02, -8.2398e-02,  3.9960e-02,  2.5783e-02,  9.3052e-02,
 -1.6024e-01,  5.1951e-01, -1.0642e-01,  2.0567e-01, -1.3884e-02,
 -2.2470e-02,  1.7148e-03, -1.4666e-02,  3.4221e-01,  8.3988e-02,
  1.3202e-01,  1.1012e-02,  7.1584e-02, -2.8851e-02,  3.4336e-01,
 -1.2569e-01,  2.8721e-02,  3.5115e-01,  9.9011e-02, -1.1934e-01,
  2.3622e-01, -2.8659e-01, -1.2438e-01, -8.2859e-02,  4.3569e-01,
 -8.2305e-02, -2.6079e-01, -1.0814e-01,  3.8412e-01, -5.5570e-03,
 -1.8803e-02, -1.1394e-01, -7.3024e-02, -8.8377e-02,  3.1881e-02,
 -1.1820e-02, -1.1358e-01,  6.1620e-03,  3.9980e-02, -2.1379e-01,
  3.0052e-01, -1.1280e-01,  1.2441e-01, -2.2796e-01, -7.3783e-02,
  9.3219e-02, -5.5974e-01,  5.3739e-02, -8.1396e-02, -4.0800e-03,
  7.0810e-02,  3.2608e-02],
[ 7.1379e-02, -8.4935e-02, -3.5575e-02,  1.0209e-01, -5.8002e-02,
  1.0332e-02, -9.9042e-02, -8.6784e-02,  7.3279e-02,  2.4427e-02,
  6.6485e-02, -8.4932e-02,  8.7545e-02, -1.9775e-02,  9.3406e-02,
  4.5611e-02,  1.2234e-02,  1.0448e-01,  1.1177e-01, -2.0165e-02,
 -1.8669e-02,  1.2126e-01,  9.6517e-02, -3.8759e-02,  3.4522e-02,
  1.0996e-01, -3.6635e-02, -1.1667e-01, -5.6355e-02, -1.2110e-01,
 -7.9034e-02, -7.3476e-02, -9.0423e-02, -7.6262e-04,  3.1730e-02,
  2.6538e-02, -9.1735e-02,  2.3544e-02, -8.3145e-02, -4.7614e-02,
 -1.0214e-01, -1.2211e-02, -3.2884e-02,  5.2594e-02, -8.4137e-02,
  2.8162e-02,  1.1758e-01,  4.7529e-02, -5.2742e-02, -1.1197e-01,
  8.0529e-02,  7.2039e-02,  5.2712e-02, -3.7272e-02,  4.9709e-02,
  1.2261e-01,  1.0799e-01, -9.7642e-02,  1.1873e-01,  4.3543e-02,
  2.6595e-02, -6.9426e-02],
[-4.8000e-02,  9.5898e-02, -1.6216e-02, -7.6105e-02,  2.4230e-03,
 -1.7766e-02,  9.6005e-03,  2.5953e-01, -6.4593e-03,  1.1092e-01,
  1.3960e-01, -1.7021e-01, -2.3599e-02, -1.5544e-01, -6.4391e-02,
  8.1059e-03,  1.0108e-01, -3.2332e-02, -1.6972e-01, -1.4680e-01,
 -1.2674e-01,  1.6923e-02,  3.3346e-02, -1.6344e-02, -2.7687e-02,
 -1.2962e-02, -5.6497e-02,  9.2495e-03, -6.7886e-02,  7.1723e-02,
 -1.1354e-01,  2.3280e-01,  2.2791e-01, -1.7001e-02, -3.0644e-01,
 -1.1613e-01,  1.2788e-01,  2.6843e-02, -1.2468e-01, -1.2430e-03,
  8.1239e-02,  9.8171e-02, -1.0111e-02, -1.6604e-02,  5.2963e-02,
  8.8858e-02,  1.0431e-01, -1.2424e-01,  8.9532e-02,  3.3146e-01,
 -2.8473e-01, -3.3803e-02, -2.3674e-02,  2.4444e-01,  2.0230e-01,
 -7.1602e-02,  4.5273e-01, -9.2867e-02, -4.9689e-02,  1.7849e-01,
  9.1628e-02, -2.6091e-02],
[ 3.4982e-02,  1.0238e-01,  3.6595e-02,  1.1679e-01,  3.7859e-02,
```

```
          6.3304e-02, -1.2131e-01, -2.1377e-02, -1.1387e-01, -1.1735e-01,
          1.0493e-01, -6.4614e-02,  7.2564e-02,  1.0228e-01, -3.9199e-02,
         -6.1526e-02, -9.1869e-02, -4.7026e-02,  8.3083e-02, -3.9730e-02,
          2.8063e-02,  1.0233e-01,  1.0765e-01, -1.0563e-01, -6.8351e-02,
          5.5443e-02, -2.8749e-02,  2.7561e-02,  9.2107e-02, -3.6220e-03,
          5.7585e-02,  6.7393e-02,  7.4567e-02,  3.3463e-02, -4.1963e-02,
         -3.8298e-02, -5.8151e-02, -5.3994e-03, -5.4642e-02,  5.5051e-02,
          2.1872e-02, -4.5549e-02, -9.0519e-02,  7.3884e-03, -4.4260e-02,
          1.1310e-01,  4.4590e-02,  3.4173e-02,  1.2210e-01,  2.1263e-02,
          3.1001e-02, -3.5005e-02,  1.7505e-02, -5.8063e-02,  7.7988e-02,
         -1.1164e-01, -1.1480e-01,  1.0718e-02,  5.0747e-03,  9.6857e-03,
          7.2045e-02, -3.4138e-02],
        [ 8.3923e-02, -1.0447e-01, -1.9271e-02, -4.2311e-03,  2.3809e-02,
          7.8110e-03, -4.9189e-02, -1.5366e-02,  1.1901e-01,  2.4161e-02,
          6.9078e-02, -7.3133e-03, -1.0411e-01, -3.8701e-02,  7.6623e-02,
         -3.4561e-02,  3.6525e-02,  5.6171e-02,  1.0097e-01, -8.1881e-02,
         -2.5876e-03, -5.1705e-02,  6.9176e-03, -2.7668e-02,  9.3069e-02,
         -6.0863e-02,  1.2269e-01,  7.6752e-02,  1.3682e-04,  1.3720e-01,
         -7.7224e-02,  1.9211e-02, -9.4508e-02,  9.1765e-02, -7.3359e-02,
         -1.3738e-02,  3.0250e-02,  1.2277e-01,  9.6756e-02, -7.4472e-02,
          1.1676e-03,  3.6431e-02, -1.2157e-01,  2.7431e-02, -7.1608e-02,
          1.1643e-01,  1.4985e-02,  2.3346e-02,  2.0856e-02,  9.9416e-03,
          7.4320e-04,  7.4648e-02,  1.0631e-01, -9.4242e-03,  5.4254e-02,
          1.1459e-01,  1.0883e-01, -9.0091e-02,  6.9171e-02,  4.0187e-02,
          9.6680e-02, -2.2657e-02],
        [-1.1995e-02, -3.6069e-02, -1.2508e-01, -1.1132e-01,  1.1044e-02,
         -8.6024e-02, -1.1962e-01, -3.7599e-02, -1.0195e-01,  4.2990e-02,
         -1.1877e-01, -1.0058e-01,  7.3929e-02, -9.4647e-03,  6.8002e-02,
          9.0595e-02,  6.1538e-02,  3.0693e-02,  1.5920e-02,  6.2161e-02,
         -8.8084e-02, -8.9872e-03,  6.5346e-02,  6.9147e-02,  1.7853e-02,
         -8.8340e-02,  8.0231e-02, -6.8428e-02,  4.2410e-02, -5.1435e-02,
          1.1237e-01, -8.8560e-02,  8.0794e-02, -1.3799e-02,  1.0311e-01,
          4.8702e-02, -2.5889e-02, -1.8838e-02, -6.3398e-02,  1.1599e-01,
          5.4854e-03,  3.3652e-02,  5.5822e-03, -1.5001e-02, -9.7702e-02,
         -1.1315e-01,  6.4790e-02, -1.4515e-02,  1.1912e-01, -5.4900e-02,
          6.3467e-03,  1.0354e-01, -4.5246e-03, -5.4512e-02, -1.0094e-01,
         -7.5870e-02, -9.3372e-02,  7.8392e-02, -2.5749e-02, -1.9589e-02,
         -2.7813e-02, -1.7624e-03]])
self.layers[9].weight.data = tensor([[-0.0539,  0.1359, -0.2045, -0.0563,
-0.2327, -0.0073, -0.0537,  0.0154,
          0.0193,  0.2333,  0.1298,  0.2090, -0.0087, -0.2149,  0.1485,
-0.1177],
        [-0.0898,  0.0694,  0.2365,  0.1432, -0.0378, -0.2003, -0.0409, -0.2199,
         -0.1708, -0.1167,  0.0138, -0.2118, -0.1600, -0.1683, -0.2293,
0.1358],
        [-0.0467,  0.1471, -0.1467, -0.2229, -0.0109, -0.1055,  0.0663, -0.1448,
          0.0568,  0.0342,  0.1373,  0.0688, -0.2099,  0.0115, -0.1964,
-0.1932],
```

```
        [ 0.8997, -0.2304,  0.3127,  0.8544,  0.1229,  0.3695, -0.1366,  0.8022,
          0.0035,  0.1410, -0.9441,  0.0275,  0.7806, -0.1190,  0.2289,
0.0041],
        [-0.0411,  0.1775, -0.0247,  0.2376, -0.1637,  0.1407, -0.1511, -0.2091,
         -0.0566,  0.2401, -0.1491, -0.0278, -0.1331, -0.1208,  0.0398,
0.1313],
        [ 0.1888,  0.1579,  0.0425,  0.0660, -0.1863,  0.0160,  0.1872,  0.2139,
          0.0455,  0.0339, -0.0386,  0.2011, -0.2329, -0.2383, -0.0214,
0.1086],
        [ 0.5282,  0.0669,  0.2108,  0.1814,  0.2400,  0.0435,  0.1228,  0.3175,
         -0.0155, -0.1515, -0.5589, -0.2131,  0.3673, -0.0798, -0.0523,
-0.1532],
        [ 0.0782, -0.1159,  0.0254, -0.4100, -0.1157,  0.3891, -0.2373, -0.2022,
          0.2178,  0.1349,  0.9287, -0.2195, -0.3960, -0.0874,  0.2110,
-0.1512]])
self.layers[11].weight.data = tensor([[-0.1617, -0.0756,  0.2116, -0.8824,
0.1317, -0.2014, -0.4151,  1.2450],
        [-0.1985, -0.3134, -0.2223, -0.3556,  0.0625, -0.3493, -0.2428,
0.6675],
        [-0.1757, -0.3352, -0.2556, -0.1718, -0.3485,  0.1244, -0.1352,
-0.1628],
        [-0.0163, -0.3279, -0.0075,  1.7810, -0.0696,  0.0341,  0.8887,
-0.2697]])
self.layers[13].weight.data = tensor([[ 2.0269,  0.8390,  0.1320, -2.0248],
        [ 0.0845,  0.1797, -0.1090, -0.1259]])
```

[ ]:

[ ]:

[ ]:

[ ]:

[ ]:

[ ]:

[ ]:

[ ]:

[ ]:

### 2.0.1 Deep Neural Network with L2-regularization

$\lambda = 0.01, 2, 100$

[134]:
```
#   Coefficient of L2-regularization = 0.01:


NonLinearNN_loaded = torch.load('NonlinearNN_model99.5.pth')
optimizer = optim.SGD(NonLinearNN_loaded.parameters(), lr=0.01, weight_decay=0.
 ↪01)
n_epochs = 150000
#while(acc<0.999):
for epoch in range(n_epochs):
    y_pred = NonLinearNN_loaded(X)
    acc = (y_pred.round() == y).float().mean()
    loss = loss_fn(y_pred[:, 0].unsqueeze(1), y)
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
    if (epoch % 10000 == 0):
        print(f'Epoch [{epoch+1}/{n_epochs}], Loss: {loss.item():.4f}, Accuracy:␣
 ↪{acc}')
    epoch = epoch +1

y_pred = NonLinearNN_loaded(X)
accuracy = (y_pred.round() == y).float().mean()
print(f"Accuracy {accuracy}")
print(f"loss {loss.item()}")
```

```
Epoch [1/150000], Loss: 0.0037, Accuracy: 0.9950000047683716
Epoch [10001/150000], Loss: 0.0277, Accuracy: 0.918749988079071
Epoch [20001/150000], Loss: 0.0278, Accuracy: 0.8949999809265137
Epoch [30001/150000], Loss: 0.0277, Accuracy: 0.8887500166893005
Epoch [40001/150000], Loss: 0.0276, Accuracy: 0.8837500214576721
Epoch [50001/150000], Loss: 0.0275, Accuracy: 0.8774999976158142
Epoch [60001/150000], Loss: 0.0275, Accuracy: 0.875
Epoch [70001/150000], Loss: 0.0275, Accuracy: 0.875
Epoch [80001/150000], Loss: 0.0275, Accuracy: 0.8737499713897705
Epoch [90001/150000], Loss: 0.0275, Accuracy: 0.8712499737739563
Epoch [100001/150000], Loss: 0.0274, Accuracy: 0.8725000023841858
Epoch [110001/150000], Loss: 0.0274, Accuracy: 0.8762500286102295
Epoch [120001/150000], Loss: 0.0274, Accuracy: 0.8887500166893005
Epoch [130001/150000], Loss: 0.0274, Accuracy: 0.9087499976158142
Epoch [140001/150000], Loss: 0.0274, Accuracy: 0.9649999737739563
Accuracy 0.9912499785423279
loss 0.027385184541344643
```

[135]:
```
#   Coefficient of L2-regularization = 2:


NonLinearNN_loaded = torch.load('NonlinearNN_model99.5.pth')
```

```
optimizer = optim.SGD(NonLinearNN_loaded.parameters(), lr=0.01, weight_decay=2)
n_epochs = 150000
#while(acc<0.999):
for epoch in range(n_epochs):
    y_pred = NonLinearNN_loaded(X)
    acc = (y_pred.round() == y).float().mean()
    loss = loss_fn(y_pred[:, 0].unsqueeze(1), y)
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
    if (epoch % 10000 == 0):
        print(f'Epoch [{epoch+1}/{n_epochs}], Loss: {loss.item():.4f}, Accuracy:␣
 ↪{acc}')
    epoch = epoch +1

y_pred = NonLinearNN_loaded(X)
accuracy = (y_pred.round() == y).float().mean()
print(f"Accuracy {accuracy}")
print(f"loss {loss.item()}")
```

```
Epoch [1/150000], Loss: 0.0037, Accuracy: 0.9950000047683716
Epoch [10001/150000], Loss: 0.6931, Accuracy: 0.5
Epoch [20001/150000], Loss: 0.6931, Accuracy: 0.5
Epoch [30001/150000], Loss: 0.6931, Accuracy: 0.5
Epoch [40001/150000], Loss: 0.6931, Accuracy: 0.5
Epoch [50001/150000], Loss: 0.6931, Accuracy: 0.5
Epoch [60001/150000], Loss: 0.6931, Accuracy: 0.5
Epoch [70001/150000], Loss: 0.6931, Accuracy: 0.5
Epoch [80001/150000], Loss: 0.6931, Accuracy: 0.5
Epoch [90001/150000], Loss: 0.6931, Accuracy: 0.5
Epoch [100001/150000], Loss: 0.6931, Accuracy: 0.5
Epoch [110001/150000], Loss: 0.6931, Accuracy: 0.5
Epoch [120001/150000], Loss: 0.6931, Accuracy: 0.5
Epoch [130001/150000], Loss: 0.6931, Accuracy: 0.5
Epoch [140001/150000], Loss: 0.6931, Accuracy: 0.5
Accuracy 0.5
loss 0.6931471824645996
```

[136]:
```
#   Coefficient of L2-regularization = 100:


NonLinearNN_loaded = torch.load('NonlinearNN_model99.5.pth')
optimizer = optim.SGD(NonLinearNN_loaded.parameters(), lr=0.01, weight_decay=100)
n_epochs = 150000
#while(acc<0.999):
for epoch in range(n_epochs):
    y_pred = NonLinearNN_loaded(X)
```

```
    acc = (y_pred.round() == y).float().mean()
    loss = loss_fn(y_pred[:, 0].unsqueeze(1), y)
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
    if (epoch % 10000 == 0):
        print(f'Epoch [{epoch+1}/{n_epochs}], Loss: {loss.item():.4f}, Accuracy:␣
↪{acc}')
    epoch = epoch +1

y_pred = NonLinearNN_loaded(X)
accuracy = (y_pred.round() == y).float().mean()
print(f"Accuracy {accuracy}")
print(f"loss {loss.item()}")
```

```
Epoch [1/150000], Loss: 0.0037, Accuracy: 0.9950000047683716
Epoch [10001/150000], Loss: 0.6931, Accuracy: 0.5
Epoch [20001/150000], Loss: 0.6931, Accuracy: 0.5
Epoch [30001/150000], Loss: 0.6931, Accuracy: 0.5
Epoch [40001/150000], Loss: 0.6931, Accuracy: 0.5
Epoch [50001/150000], Loss: 0.6931, Accuracy: 0.5
Epoch [60001/150000], Loss: 0.6931, Accuracy: 0.5
Epoch [70001/150000], Loss: 0.6931, Accuracy: 0.5
Epoch [80001/150000], Loss: 0.6931, Accuracy: 0.5
Epoch [90001/150000], Loss: 0.6931, Accuracy: 0.5
Epoch [100001/150000], Loss: 0.6931, Accuracy: 0.5
Epoch [110001/150000], Loss: 0.6931, Accuracy: 0.5
Epoch [120001/150000], Loss: 0.6931, Accuracy: 0.5
Epoch [130001/150000], Loss: 0.6931, Accuracy: 0.5
Epoch [140001/150000], Loss: 0.6931, Accuracy: 0.5
Accuracy 0.5
loss 0.6931471824645996
```

#### 2.0.2 All the models start at the exact same place. Seems like lambda = 0.01 does a fantastic job! Others fall down and get stuck at the 50% accuracy.

[ ]: