# Deep Learning Course

## 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 a separate 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) I plan to use BCELoss and SGD optimization algorithm.
    - 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) I plan to use MSELoss and SGD optimization algorithm. My net work structure is (64,128,1)
    - 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) I plan to use BCELoss and Adam optimization algorithm, my network structure is (128,64,64,1),my activation function is ReLu.
    - 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. Better yet, use a tool for keeping track of machine learning experiments, such as Wandb or TensorBoard. These tools are more powerful than what you need in the assignment, but if you plan on doing more complex experiments in future applications, you will find them very useful. (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. We provide example values for linear regression learned with gradient descent from a random starting point.

| Model | Loss | Accuracy | Equation of Decision Boundary | NN Structure | Activation Function | Loss Function |
|---|---|---|---|---|---|---|
| Linear Regression | 0.14 | 72.96% | $0.1817x_1 + 0.5237x_2 + 0.4758 = 0$ | - | - | Mean Square Error |
| Logistic Regression | 0.6489 | 72.25% | $-0.1548x_1 + 0.3680x_2 + 0.1138 = 0$ | - | - | Binary Cross Entropy |
| Linear Neural Network | 0.1407 | 75% | $-1.298x_1 + 0.8133x_2 + 0.5228 = 0$ | (64,128,1 ) | None | Mean Square Error |
| Feedforward Neural Network | 0.0035 | 99.75% | - | (128,64,64,1) | Relu | 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)

Logistic regression has the worst performance, its loss ofen got higher than 0.6 than others and the accuracy is around 68% to 76%, linear regression is a little bit better with loss around 0.14 and accuracy from 63% to 74%,linear neural network is more stable, with Loss=0.1407 and its accuracy stay on 75%. Deep nonlinear neural network is the best one with loss less than 0.0035 and accuracy around 99.75. The performance differences between linear regression, logistic regression, linear neural networks, and deep nonlinear neural networks are primarily due to their respective capacities to model complex relationships within the data. Linear regression is typically unsuitable for classification due to its focus on continuous outcomes. Logistic regression can model binary outcomes and provides probabilities for classification, making it more appropriate than linear regression for such tasks. Linear neural networks are limited to linear decision boundaries, which constrains their effectiveness to linearly separable data. Deep nonlinear neural networks, with their multiple layers and non-linear activation functions, can capture complex, non-linear relationships and interactions between features, providing the highest flexibility and potential accuracy among the models discussed, especially for complex datasets that are not linearly separable.

- Your table and analysis of (5. Add L2-regularization) here.

| value of λ | Loss | Accuracy |
|---|---|---|
| None | 0.0035 | 99.75% |
| λ=0.01 | 0.0662 | 99.25% |
| λ=2 | 0.6931 | 50% |
| λ=100 | 0.6931 | 50% |

### 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.

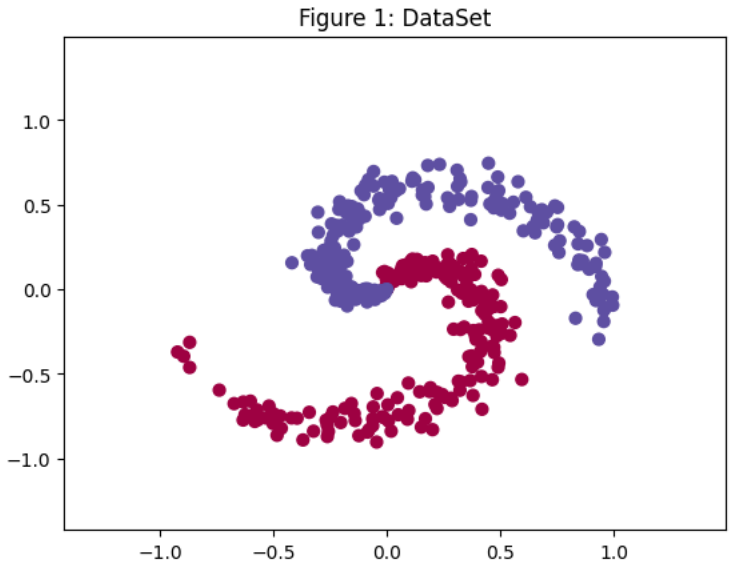
Your assignments will be marked based on correctness, originality (the implementations and ideas are from yourself), clarity and performance. Clarity means whether the logic of your code is easy to follow. This includes 1) comments to explain the logic of your code 2) meaningful variable names. Performance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

## Your Implementation

### Toy Data and Helper Functions

```
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
```

```
# 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()



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



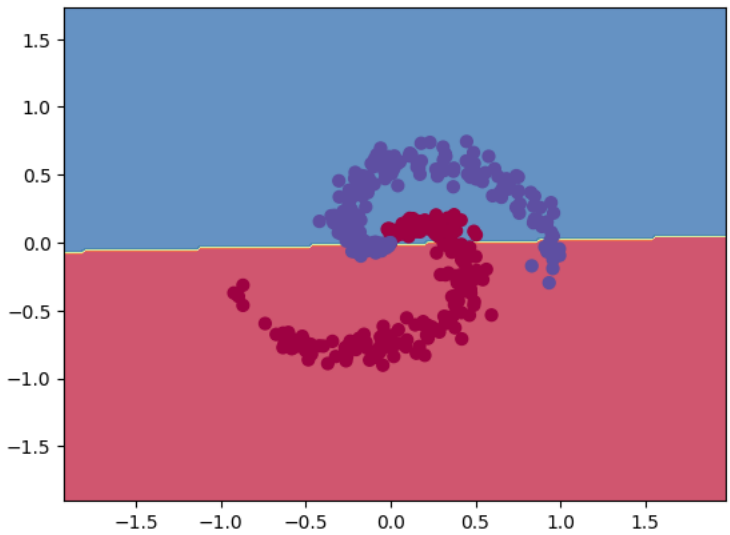

## Given Example: Linear Regression

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.

```
# 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 [450/500], Loss: 0.1516, Accuracy: 0.7
Epoch [451/500], Loss: 0.1516, Accuracy: 0.7
Epoch [452/500], Loss: 0.1515, Accuracy: 0.7
Epoch [453/500], Loss: 0.1514, Accuracy: 0.7
Epoch [454/500], Loss: 0.1514, Accuracy: 0.7
Epoch [455/500], Loss: 0.1513, Accuracy: 0.7
Epoch [456/500], Loss: 0.1512, Accuracy: 0.7
Epoch [457/500], Loss: 0.1512, Accuracy: 0.7
Epoch [458/500], Loss: 0.1511, Accuracy: 0.7
Epoch [459/500], Loss: 0.1511, Accuracy: 0.7
Epoch [460/500], Loss: 0.1510, Accuracy: 0.7
Epoch [461/500], Loss: 0.1509, Accuracy: 0.7
Epoch [462/500], Loss: 0.1509, Accuracy: 0.7
Epoch [463/500], Loss: 0.1508, Accuracy: 0.7025
Epoch [464/500], Loss: 0.1508, Accuracy: 0.7025
Epoch [465/500], Loss: 0.1507, Accuracy: 0.705
Epoch [466/500], Loss: 0.1506, Accuracy: 0.705
Epoch [467/500], Loss: 0.1506, Accuracy: 0.7075
Epoch [468/500], Loss: 0.1505, Accuracy: 0.7075
Epoch [469/500], Loss: 0.1505, Accuracy: 0.7075
Epoch [470/500], Loss: 0.1504, Accuracy: 0.7075
Epoch [471/500], Loss: 0.1503, Accuracy: 0.7075
Epoch [472/500], Loss: 0.1503, Accuracy: 0.705
Epoch [473/500], Loss: 0.1502, Accuracy: 0.705
Epoch [474/500], Loss: 0.1502, Accuracy: 0.705
Epoch [475/500], Loss: 0.1501, Accuracy: 0.705
Epoch [476/500], Loss: 0.1501, Accuracy: 0.705
Epoch [477/500], Loss: 0.1500, Accuracy: 0.705
Epoch [478/500], Loss: 0.1500, Accuracy: 0.705
Epoch [479/500], Loss: 0.1499, Accuracy: 0.705
Epoch [480/500], Loss: 0.1499, Accuracy: 0.705
Epoch [481/500], Loss: 0.1498, Accuracy: 0.705
Epoch [482/500], Loss: 0.1497, Accuracy: 0.705
Epoch [483/500], Loss: 0.1497, Accuracy: 0.7075
Epoch [484/500], Loss: 0.1496, Accuracy: 0.7075
Epoch [485/500], Loss: 0.1496, Accuracy: 0.7075
Epoch [486/500], Loss: 0.1495, Accuracy: 0.7075
Epoch [487/500], Loss: 0.1495, Accuracy: 0.7075
Epoch [488/500], Loss: 0.1494, Accuracy: 0.7075
Epoch [489/500], Loss: 0.1494, Accuracy: 0.7075
Epoch [490/500], Loss: 0.1493, Accuracy: 0.7075
Epoch [491/500], Loss: 0.1493, Accuracy: 0.7075
Epoch [492/500], Loss: 0.1492, Accuracy: 0.7075
Epoch [493/500], Loss: 0.1492, Accuracy: 0.7075
Epoch [494/500], Loss: 0.1491, Accuracy: 0.7075
Epoch [495/500], Loss: 0.1491, Accuracy: 0.7075
Epoch [496/500], Loss: 0.1490, Accuracy: 0.7075
Epoch [497/500], Loss: 0.1490, Accuracy: 0.7075
Epoch [498/500], Loss: 0.1489, Accuracy: 0.7075
Epoch [499/500], Loss: 0.1489, Accuracy: 0.7075
Epoch [500/500], Loss: 0.1488, Accuracy: 0.7075
```

```
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.

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

```
Weights:
 Parameter containing:
tensor([[-0.0178,  0.5965]], requires_grad=True)
Bias:
 Parameter containing:
tensor([0.5056], requires_grad=True)
```

## Logistic Regression

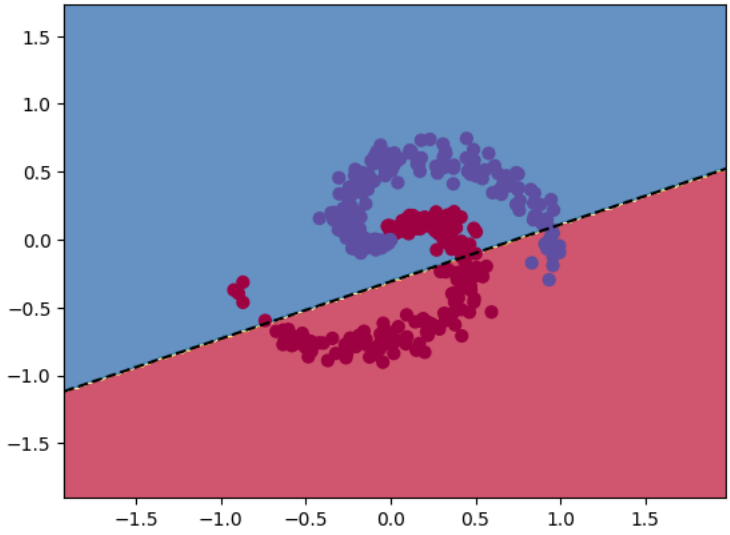
```
# implement your logistic regression here
import torch
import torch.nn as nn
import torch.optim as optim
from sklearn.metrics import accuracy_score

X_tensor = torch.from_numpy(sample).float()
y_tensor = torch.from_numpy(target).float()

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

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


model = LogisticRegressionModel(X_tensor.shape[1])
criterion = nn.BCELoss()
optimizer = optim.SGD(model.parameters(), lr=0.01)


num_epochs = 500
for epoch in range(num_epochs):

        y_pred = 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.zero_grad()
        loss.backward()
        optimizer.step()

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

```
Epoch [468/500], Loss: 0.6540, Accuracy: 0.71
Epoch [469/500], Loss: 0.6538, Accuracy: 0.7125
Epoch [470/500], Loss: 0.6537, Accuracy: 0.7125
Epoch [471/500], Loss: 0.6535, Accuracy: 0.7125
Epoch [472/500], Loss: 0.6534, Accuracy: 0.715
Epoch [473/500], Loss: 0.6532, Accuracy: 0.715
Epoch [474/500], Loss: 0.6530, Accuracy: 0.715
Epoch [475/500], Loss: 0.6529, Accuracy: 0.7175
Epoch [476/500], Loss: 0.6527, Accuracy: 0.7175
Epoch [477/500], Loss: 0.6526, Accuracy: 0.7175
Epoch [478/500], Loss: 0.6524, Accuracy: 0.7175
Epoch [479/500], Loss: 0.6522, Accuracy: 0.7175
Epoch [480/500], Loss: 0.6521, Accuracy: 0.72
Epoch [481/500], Loss: 0.6519, Accuracy: 0.72
Epoch [482/500], Loss: 0.6518, Accuracy: 0.72
Epoch [483/500], Loss: 0.6516, Accuracy: 0.72
Epoch [484/500], Loss: 0.6514, Accuracy: 0.72
Epoch [485/500], Loss: 0.6513, Accuracy: 0.72
Epoch [486/500], Loss: 0.6511, Accuracy: 0.72
Epoch [487/500], Loss: 0.6510, Accuracy: 0.72
Epoch [488/500], Loss: 0.6508, Accuracy: 0.72
Epoch [489/500], Loss: 0.6506, Accuracy: 0.72
Epoch [490/500], Loss: 0.6505, Accuracy: 0.72
Epoch [491/500], Loss: 0.6503, Accuracy: 0.72
Epoch [492/500], Loss: 0.6502, Accuracy: 0.72
Epoch [493/500], Loss: 0.6500, Accuracy: 0.72
Epoch [494/500], Loss: 0.6499, Accuracy: 0.72
Epoch [495/500], Loss: 0.6497, Accuracy: 0.7225
Epoch [496/500], Loss: 0.6495, Accuracy: 0.7225
Epoch [497/500], Loss: 0.6494, Accuracy: 0.7225
Epoch [498/500], Loss: 0.6492, Accuracy: 0.7225
Epoch [499/500], Loss: 0.6491, Accuracy: 0.7225
Epoch [500/500], Loss: 0.6489, Accuracy: 0.7225
```

```
def visualize_logestic(sample, target, model,weight1, weight2,bias):
    """
    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)

    x1=xx[:,0]
    x2= (-bias-(x1*weight1))/weight2
    # Create a contour plot to visualize the decision boundaries
    fig = plt.figure()
    plt.plot(x1,x2,'k--')
    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()
```

```
print("Weights: \n",model.linear.weight)
print("Bias: \n",model.linear.bias)
weight1 = model.linear.weight.data[0][0]
weight2 = model.linear.weight.data[0][1]
bias=model.linear.bias.data[0]
visualize_logestic(sample, target, model,weight1, weight2,bias)
```

```
Weights:
 Parameter containing:
tensor([[-0.1548,  0.3680]], requires_grad=True)
Bias:
 Parameter containing:
tensor([0.1138], requires_grad=True)
```



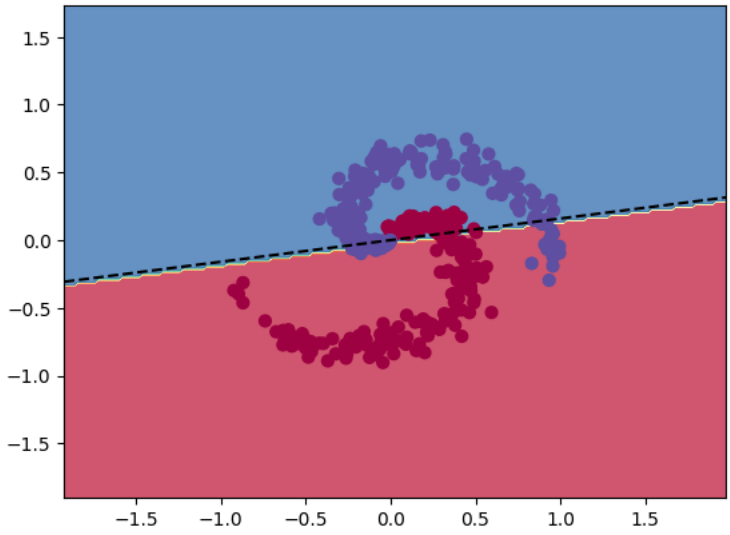

## Deep Linear Neural Network

```
# # implement your nonlinear feed forward neural network here
import torch
import torch.nn as nn
import torch.optim as optim

X_tensor = torch.from_numpy(sample).float()
y_tensor = torch.from_numpy(target).float()


class LinearNNModel(nn.Module):
        def __init__(self, input_size):
                super(LinearNNModel, self).__init__()
                self.linear1= nn.Linear(input_size, 64)
                self.linear2=nn.Linear(64,128)
                self.linear3=nn.Linear(128,1)

        def forward(self, x):
                x= self.linear1(x)
                x= self.linear2(x)
                x= self.linear3(x)
                return x


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

num_epochs = 500
for epoch in range(num_epochs):

        y_pred = 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.zero_grad()
        loss.backward()
        optimizer.step()

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

```
Epoch [443/500], Loss: 0.1407, Accuracy: 0.75
Epoch [444/500], Loss: 0.1407, Accuracy: 0.75
Epoch [445/500], Loss: 0.1407, Accuracy: 0.75
Epoch [446/500], Loss: 0.1407, Accuracy: 0.75
Epoch [447/500], Loss: 0.1407, Accuracy: 0.75
Epoch [448/500], Loss: 0.1407, Accuracy: 0.75
Epoch [449/500], Loss: 0.1407, Accuracy: 0.75
Epoch [450/500], Loss: 0.1407, Accuracy: 0.75
Epoch [451/500], Loss: 0.1407, Accuracy: 0.75
Epoch [452/500], Loss: 0.1407, Accuracy: 0.75
Epoch [453/500], Loss: 0.1407, Accuracy: 0.75
Epoch [454/500], Loss: 0.1407, Accuracy: 0.75
Epoch [455/500], Loss: 0.1407, Accuracy: 0.75
Epoch [456/500], Loss: 0.1407, Accuracy: 0.75
Epoch [457/500], Loss: 0.1407, Accuracy: 0.75
Epoch [458/500], Loss: 0.1407, Accuracy: 0.75
Epoch [459/500], Loss: 0.1407, Accuracy: 0.75
Epoch [460/500], Loss: 0.1407, Accuracy: 0.75
Epoch [461/500], Loss: 0.1407, Accuracy: 0.75
Epoch [462/500], Loss: 0.1407, Accuracy: 0.75
Epoch [463/500], Loss: 0.1407, Accuracy: 0.75
Epoch [464/500], Loss: 0.1407, Accuracy: 0.75
Epoch [465/500], Loss: 0.1407, Accuracy: 0.75
Epoch [466/500], Loss: 0.1407, Accuracy: 0.75
Epoch [467/500], Loss: 0.1407, Accuracy: 0.75
Epoch [468/500], Loss: 0.1407, Accuracy: 0.75
Epoch [469/500], Loss: 0.1407, Accuracy: 0.75
Epoch [470/500], Loss: 0.1407, Accuracy: 0.75
Epoch [471/500], Loss: 0.1407, Accuracy: 0.75
Epoch [472/500], Loss: 0.1407, Accuracy: 0.75
Epoch [473/500], Loss: 0.1407, Accuracy: 0.75
Epoch [474/500], Loss: 0.1407, Accuracy: 0.75
Epoch [475/500], Loss: 0.1407, Accuracy: 0.75
Epoch [476/500], Loss: 0.1407, Accuracy: 0.75
Epoch [477/500], Loss: 0.1407, Accuracy: 0.75
Epoch [478/500], Loss: 0.1407, Accuracy: 0.75
Epoch [479/500], Loss: 0.1407, Accuracy: 0.75
Epoch [480/500], Loss: 0.1407, Accuracy: 0.75
Epoch [481/500], Loss: 0.1407, Accuracy: 0.75
Epoch [482/500], Loss: 0.1407, Accuracy: 0.75
Epoch [483/500], Loss: 0.1407, Accuracy: 0.75
Epoch [484/500], Loss: 0.1407, Accuracy: 0.75
Epoch [485/500], Loss: 0.1407, Accuracy: 0.75
Epoch [486/500], Loss: 0.1407, Accuracy: 0.75
Epoch [487/500], Loss: 0.1407, Accuracy: 0.75
Epoch [488/500], Loss: 0.1407, Accuracy: 0.75
Epoch [489/500], Loss: 0.1407, Accuracy: 0.75
Epoch [490/500], Loss: 0.1407, Accuracy: 0.75
Epoch [491/500], Loss: 0.1407, Accuracy: 0.75
Epoch [492/500], Loss: 0.1407, Accuracy: 0.75
Epoch [493/500], Loss: 0.1407, Accuracy: 0.75
Epoch [494/500], Loss: 0.1407, Accuracy: 0.75
Epoch [495/500], Loss: 0.1407, Accuracy: 0.75
Epoch [496/500], Loss: 0.1407, Accuracy: 0.75
Epoch [497/500], Loss: 0.1407, Accuracy: 0.75
Epoch [498/500], Loss: 0.1407, Accuracy: 0.75
Epoch [499/500], Loss: 0.1407, Accuracy: 0.75
Epoch [500/500], Loss: 0.1407, Accuracy: 0.75
```

```
def visualize_linearnn(sample, target, model,weight1, weight2,bias):
        """
        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)

        x1=xx[:,0]
        x2= (-(x1*weight1))/weight2
        # Create a contour plot to visualize the decision boundaries
        fig = plt.figure()
        plt.plot(x1,x2,'k--')
        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()



W1 = model.linear1.weight
b1 = model.linear1.bias
W2 = model.linear2.weight
b2 = model.linear2.bias
W3 = model.linear3.weight
b3 = model.linear3.bias

b1 = b1.view(-1,1)
b2 = b2.view(-1,1)
b3 = b3.view(-1,1)
temp1 = torch.cat((W1,b1),dim=1)
temp1 = torch.matmul(W2,temp1)
print(temp1.shape)
temp2 = torch.cat((temp1,b2),dim=1)
print(temp2.shape)
temp2 = torch.matmul(W3,temp2)
print(temp2.shape)
temp3 = torch.cat((temp2,b3),dim=1)

weight1=temp3.data[0][0]
weight2=temp3.data[0][1]
bias=temp3.data[0][2]+temp3.data[0][3]+temp3.data[0][4]


print(weight1)
print(weight2)
print(bias)
visualize_linearnn(sample, target, model,weight1, weight2,bias)
```

```
torch.Size([128, 3])
torch.Size([128, 4])
torch.Size([1, 4])
tensor(-0.1298)
tensor(0.8133)
tensor(0.5228)
```

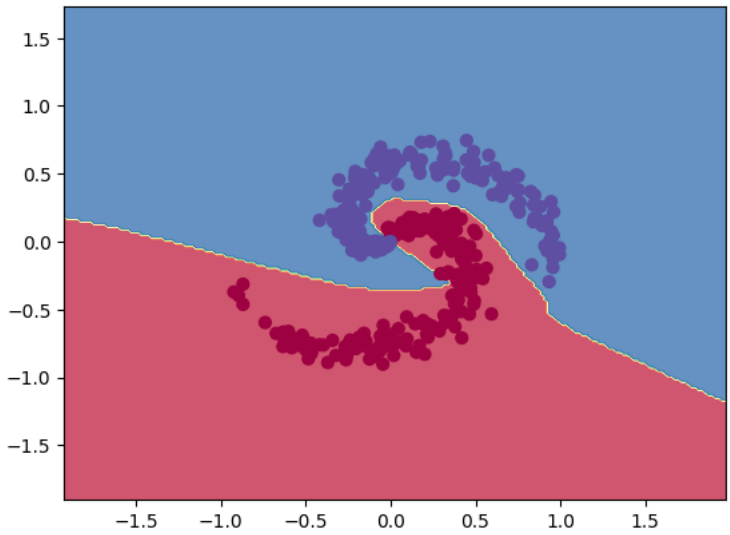



## Deep Neural Network

```
# implement your nonlinear feed forward neural network here
import torch
import torch.nn as nn
import torch.optim as optim

X_tensor = torch.from_numpy(sample).float()
y_tensor = torch.from_numpy(target).float()

class DeepNNModel(nn.Module):
        def __init__(self, input_size):
                super(DeepNNModel, self).__init__()
                self.linear1= nn.Linear(input_size, 128)
                self.linear2=nn.Linear(128,64)
                self.linear3=nn.Linear(64,64)
                self.linear4=nn.Linear(64,1)
                self.output_activation = nn.Sigmoid()

                self.activation = nn.ReLU()


        def forward(self, x):
                x=self.activation(self.linear1(x))
                x=self.activation(self.linear2(x))
                x=self.activation(self.linear3(x))
                x=self.output_activation(self.linear4(x))
                return x

model = DeepNNModel(X_tensor.shape[1])
criterion = nn.BCELoss()
optimizer = optim.Adam(model.parameters(), lr=0.01)

num_epochs = 10000
for epoch in range(num_epochs):

        y_pred = 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.zero_grad()
        loss.backward()
        optimizer.step()

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

```
Epoch [9976/10000], Loss: 0.0035, Accuracy: 0.9975
```

```
visualize(sample, target, model)
```




## Deep Neural Network with L2-regularization

$\lambda = 0.01, 2, 100$

```
# implement your nonlinear feed forward neural network here
import torch
import torch.nn as nn
import torch.optim as optim

X_tensor = torch.from_numpy(sample).float()
y_tensor = torch.from_numpy(target).float()

class DeepNNModel(nn.Module):
        def __init__(self, input_size):
                super(DeepNNModel, self).__init__()
                self.linear1= nn.Linear(input_size, 128)
                self.linear2=nn.Linear(128,64)
                self.linear3=nn.Linear(64,64)
                self.linear4=nn.Linear(64,1)
                self.output_activation = nn.Sigmoid()

                self.activation = nn.ReLU()


        def forward(self, x):
                x=self.activation(self.linear1(x))
                x=self.activation(self.linear2(x))
                x=self.activation(self.linear3(x))
                x=self.output_activation(self.linear4(x))
                return x

model = DeepNNModel(X_tensor.shape[1])
criterion = nn.BCELoss()
optimizer = optim.Adam(model.parameters(), lr=0.01,weight_decay=0.01)

num_epochs = 10000
for epoch in range(num_epochs):

        y_pred = 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.zero_grad()
        loss.backward()
        optimizer.step()

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