Name 1: Md Mobashir Rahman
Student id 1: 7059086
Email 1: mdra00001@stud.uni-saarland.de

Name 2: Ratnadeep Chakraborty
Student id 2: 7022859
Email 2: rach00002@stud.uni-saarland.de

Exercise 5.4 - Feedforward Neural Network (3 points)

In the lecture, you were introduced to some problems that can be solved with a two-layer neural network but not with a single-layer one. In this practical exercise, let's try solving such a problem together and get familiar with building and training models using PyTorch.

Import libraries

In [36]:
```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.cm as cm

import torch
import torch.nn as nn
from torch.utils.data import Dataset, DataLoader
import torch.nn.functional as F
```

Generating Data

Here, we generate a dataset consisting of 3 classes in 2 dimensional space, where no two classes are linearly separable.

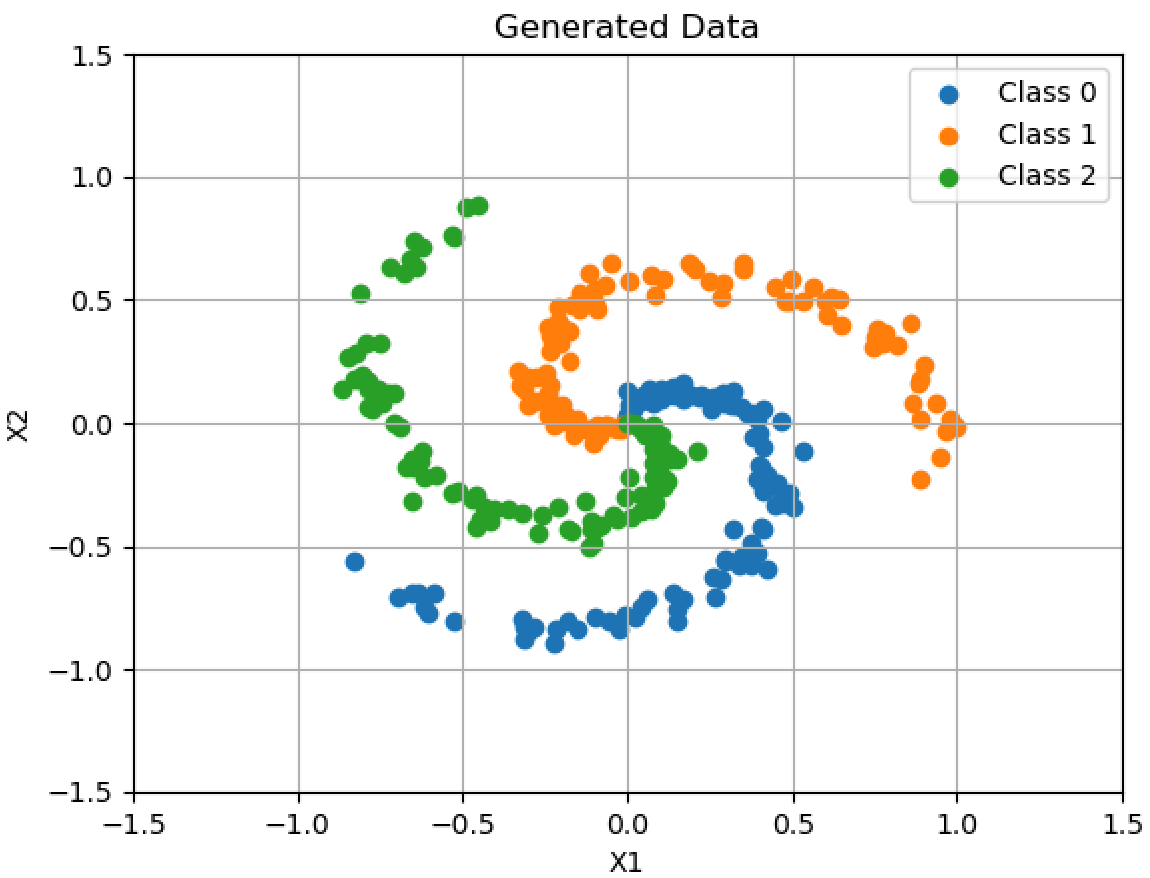
In [37]:
```
np.random.seed(352)
torch.manual_seed(0)

N = 100 # number of points per class
C = 3 # number of classes

def generate_data(N, num_class):
    X = np.zeros((2, N*num_class)) # data matrix
    y = np.zeros(N*num_class) # class abels

    for j in range(num_class):
        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)*0.2 # theta
        X[:,ix] = np.c_[r*np.sin(t), r*np.cos(t)].T
        y[ix] = j
```

```
    return X.T, y

# Generate data
data, labels = generate_data(N, C)

# # Uncomment to see the shape of data and labels
print("Data shape :", data.shape)
print("Labels shape :", labels.shape)
```

```
Data shape : (300, 2)
Labels shape : (300,)
```

At this point let's visualize the dataset we just generated.

In [38]:
```
for i in range(C):
  plt.scatter(data[i*N:(i+1)*N, 0], data[i*N:(i+1)*N, 1], label=('Class '

plt.xlim([-1.5, 1.5])
plt.ylim([-1.5, 1.5])
plt.title("Generated Data")
plt.xlabel("X1")
plt.ylabel("X2")
plt.legend()
plt.grid(True)
plt.show()
```

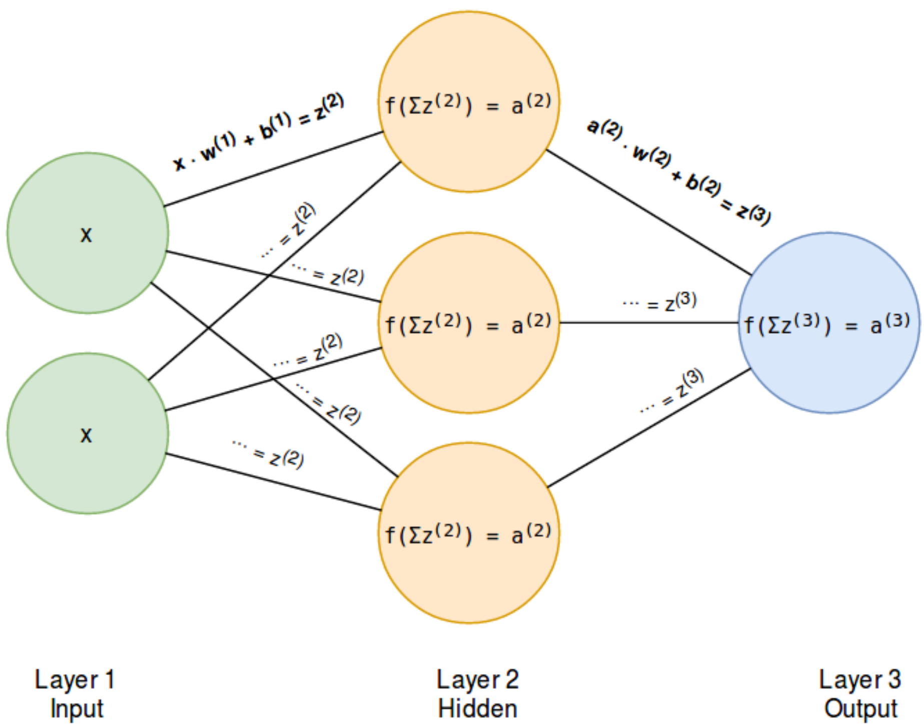


5.4.1 DataLoader (0.5 points)

Complete the PointDataset by replacing `NotImplemented` with your code

In [39]:
```
class PointsDataset(Dataset):
    def __init__(self, data, labels):
        self.data = torch.tensor(data, dtype=torch.float)
        self.labels = torch.tensor(labels, dtype=torch.long)

    def __getitem__(self, idx):
        return self.data[idx], self.labels[idx]

    def __len__(self):
        return len(self.data)

class PointsDataLoader(DataLoader):
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)

        self.transform = lambda x: x
    def __iter__(self):
        for batch in super().__iter__():
            yield self.transform(batch)
```

5.4.2 Building our 2-layer neural network (0.5 points)



In this exercise, you are required to complete a model using the layers provided by PyTorch. To complete this exercise, you can refer to the following: Building Models with PyTorch. Our model will be described as consisting of the following components:

Linear Layer:

$$\mathbf{z}^{(2)} = \mathbf{W}^{(1)T}\mathbf{x}$$

ReLU Activation:

$$\mathbf{A}^{(2)} = \max(\mathbf{Z}^{(2)}, \mathbf{0})$$

Linear Layer:

$$\mathbf{Z}^{(3)} = \mathbf{W}^{(2)T}\mathbf{A}^{(1)}$$

Softmax Activation:

$$\hat{\mathbf{Y}} = \mathbf{A}^{(3)} = \text{softmax}(\mathbf{Z}^{(3)})$$

In [40]:

```
class FFNN(nn.Module):
    def __init__(self, input_dim, hidden_dim, output_dim) -> None:
        super(FFNN, self).__init__()
        self.linear1 = nn.Linear(input_dim, hidden_dim)
        self.linear2 = nn.Linear(hidden_dim, output_dim)

    def forward(self, x):
        z2 = self.linear1(x)
        a2 = F.relu(z2)
        z3 = self.linear2(a2)
        y_hat = F.softmax(z3, dim=1)
        return y_hat
```

5.4.3 Train our model (1 point)

Here, we specify the important hyperparameters for training the model as well as provide a function to plot the decision boundary to help visualize the results after training.

In [41]:

```
config = {
    "input_dim": 2,
    "hidden_dim": 100,
    "output_dim": C,
    "device": torch.device("cuda") if torch.cuda.is_available() else torc
    "num_epochs": 1000,
    "batch_size": 300,
    "learning_rate": 1,
}
```

In [42]:

```
# Visualize function
def plot_decision_boundary(model, points, labels, title="Decision Boundar
    model.eval()
    model.to("cpu")
    plt.figure(figsize=(8, 6))

    # Plot the data points
    for label in np.unique(labels):
        label_points = points[labels.flatten() == label]
        plt.scatter(label_points[:, 0], label_points[:, 1], label=('Class

    # Create grid to evaluate the model
    x_min, x_max = points[:, 0].min() - 1, points[:, 0].max() + 1
```

```
    y_min, y_max = points[:, 1].min() - 1, points[:, 1].max() + 1
    xx, yy = np.meshgrid(np.linspace(x_min, x_max, 100), np.linspace(y_mi
    X0 = torch.tensor(np.c_[xx.ravel(), yy.ravel()], dtype=torch.float)
    Z = model(X0).argmax(dim=1)
    Z = Z.reshape(xx.shape)

    # Plot decision boundary
    plt.contourf(xx, yy, Z, alpha=0.2, cmap=plt.cm.coolwarm)
    plt.title(title)
    plt.xlabel("X1")
    plt.ylabel("X2")
    plt.legend()
    plt.grid(True)
    plt.show()
```

(1 point) Complete the train function below to train the model.

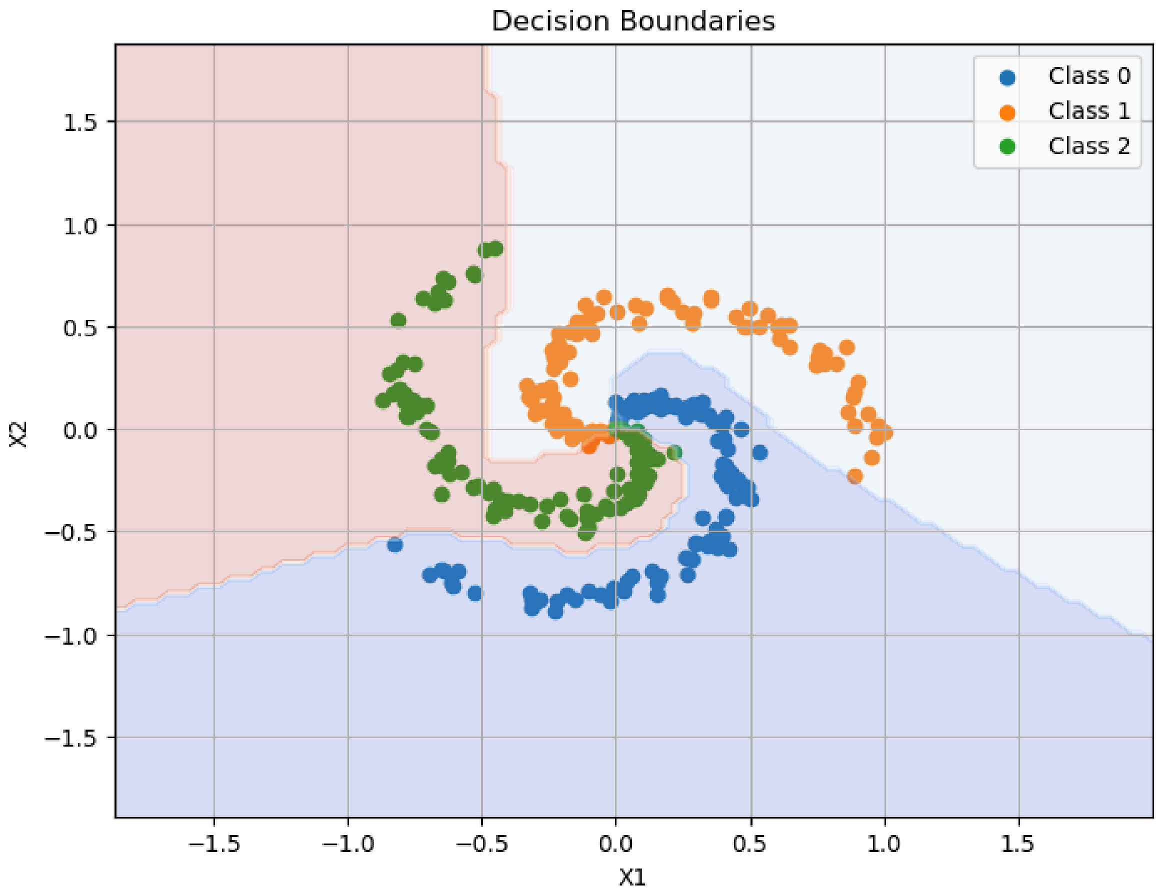
To complete this exercise, you can refer Training with PyTorch for a more detailed understanding.

In [43]:
```
def train(config, data, labels):
    # Create out data loader
    points_dataset = PointsDataset(data, labels)
    points_dataloader = PointsDataLoader(points_dataset, batch_size=confi

    # Init our model
    model = FFNN(config["input_dim"], config["hidden_dim"], config["outpu
    model.to(config["device"])

    # Loss function and optimizer
    criterion = nn.CrossEntropyLoss()
    optimizer = torch.optim.SGD(model.parameters(), lr=config["learning_r

    model.train()
    for epoch in range(config["num_epochs"]):
        running_loss = 0
        correct = 0

        for batch_point, batch_label in points_dataloader:
            batch_point = batch_point.to(config["device"])
            batch_label = batch_label.to(config["device"])

            # Compute the output of the model
            output = model(batch_point)

            # Compute the loss
            loss = criterion(output, batch_label)

            # Zero your gradients for every batch
            optimizer.zero_grad()

            # Compute the gradients of loss
            loss.backward()

            # Adjust learning weights
            optimizer.step()

            running_loss += loss.item()
            correct += (output.argmax(1) == batch_label).sum().item()
```

```
        if (epoch + 1) % 100 == 0:
            print(f'Epoch {epoch+1} | Running Loss: {running_loss/len(poi

    plot_decision_boundary(model, data, labels, title="Decision Boundarie
```

In [44]:
```
train(config, data, labels)
```

```
Epoch 100 | Running Loss: 0.8401603698730469 | Accuracy: 0.6633333333333333
3
Epoch 200 | Running Loss: 0.7595046758651733 | Accuracy: 0.7766666666666666
6
Epoch 300 | Running Loss: 0.7039985060691833 | Accuracy: 0.8833333333333333
3
Epoch 400 | Running Loss: 0.6738966107368469 | Accuracy: 0.9166666666666666
6
Epoch 500 | Running Loss: 0.6558569073677063 | Accuracy: 0.9333333333333333
3
Epoch 600 | Running Loss: 0.643133819103241 | Accuracy: 0.9366666666666666
Epoch 700 | Running Loss: 0.6333959698677063 | Accuracy: 0.95
Epoch 800 | Running Loss: 0.6254753470420837 | Accuracy: 0.95
Epoch 900 | Running Loss: 0.6189134120941162 | Accuracy: 0.9566666666666666
7
Epoch 1000 | Running Loss: 0.6132668852806091 | Accuracy: 0.963333333333333
34
```



5.4.4 Let's do some experiments. (1 point)

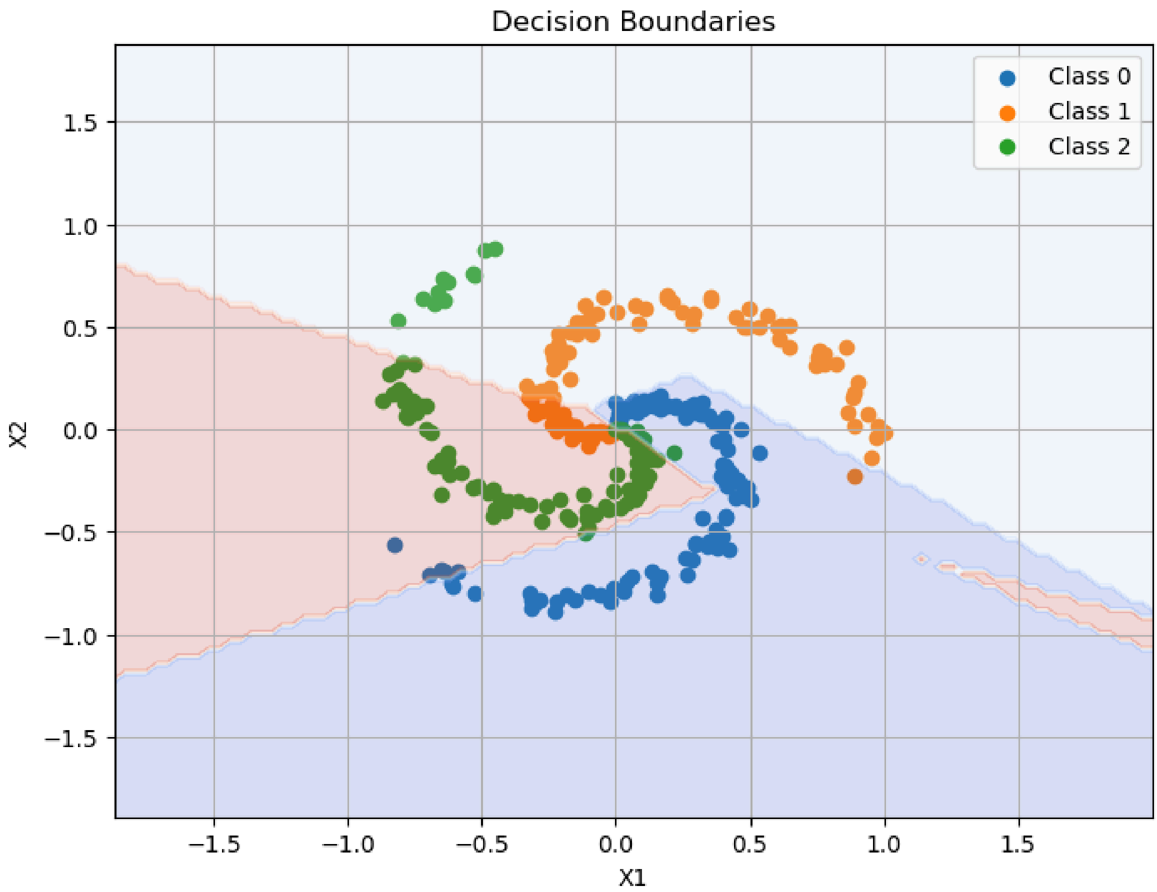
(0.5 points) Train the model again with `config_1` and explain what you observe. Provide an explanation for your observation (maximum 2 sentences).

With the hyperparameters provided in `config`, we changed `hidden_dim` from 100 to 5.

In [45]:
```
config_1 = {
    "input_dim": 2,
    "hidden_dim": 5,
    "output_dim": C,
    "device": torch.device("cuda") if torch.cuda.is_available() else torc
    "num_epochs": 1000,
    "batch_size": 300,
    "learning_rate": 1,
}
```

In [46]:
```
# Run this cell to train the model with config_1
train(config_1, data, labels)
```

```
Epoch 100 | Running Loss: 0.9916317462921143 | Accuracy: 0.533333333333333
3
Epoch 200 | Running Loss: 0.8871389627456665 | Accuracy: 0.646666666666666
6
Epoch 300 | Running Loss: 0.8503427505493164 | Accuracy: 0.736666666666666
7
Epoch 400 | Running Loss: 0.8221354484558105 | Accuracy: 0.756666666666666
7
Epoch 500 | Running Loss: 0.8050215840339661 | Accuracy: 0.77
Epoch 600 | Running Loss: 0.7939465045928955 | Accuracy: 0.776666666666666
6
Epoch 700 | Running Loss: 0.785828709602356 | Accuracy: 0.7933333333333333
Epoch 800 | Running Loss: 0.7794409990310669 | Accuracy: 0.8
Epoch 900 | Running Loss: 0.7741913199424744 | Accuracy: 0.81
Epoch 1000 | Running Loss: 0.7697529196739197 | Accuracy: 0.81666666666666
67
```



Observation

The model with a hidden layer size of **100** achieves significantly higher accuracy (up to ~96%) compared to the model with a hidden layer size of **5** (up to ~81%).

Explanation

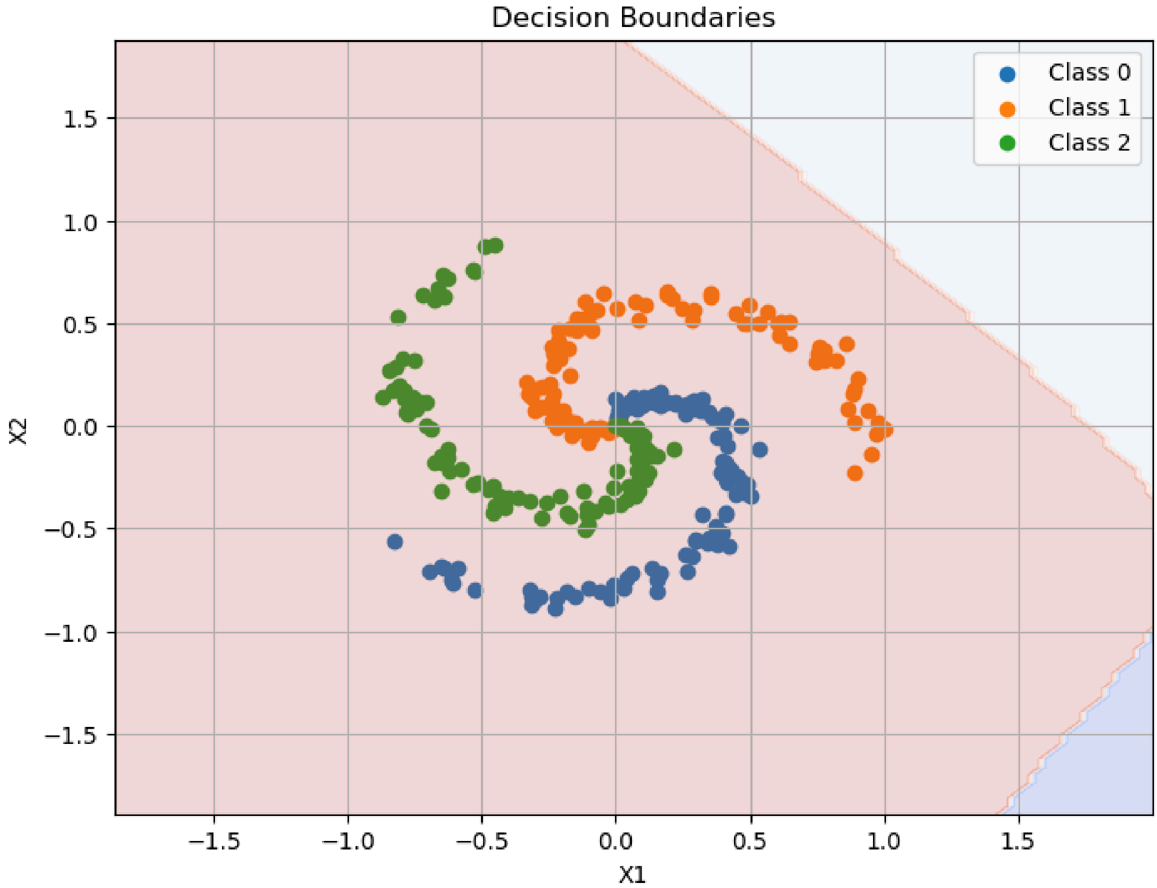
A larger hidden layer increases the model's capacity to learn complex patterns in the data, leading to better performance, whereas a smaller hidden layer limits this capacity, resulting in lower accuracy.

(0.5 points) Train the model again with `config_2` below and explain what you observe. Provide an explanation for your observation (maximum 2 sentences).

In [47]:
```
# In config_2, we keep the hyperparameters the same as in the config, exc
config_2 = {
    "input_dim": 2,
    "hidden_dim": 100,
    "output_dim": C,
    "device": torch.device("cuda") if torch.cuda.is_available() else torc
    "num_epochs": 1000,
    "batch_size": 300,
    "learning_rate": 100,
}
```

```python
# Train the model with config_2
train(config_2, data, labels)
```

```
Epoch 100 | Running Loss: 1.218111515045166 | Accuracy: 0.3333333333333333
Epoch 200 | Running Loss: 1.218111515045166 | Accuracy: 0.3333333333333333
Epoch 300 | Running Loss: 1.218111515045166 | Accuracy: 0.3333333333333333
Epoch 400 | Running Loss: 1.218111515045166 | Accuracy: 0.3333333333333333
Epoch 500 | Running Loss: 1.218111515045166 | Accuracy: 0.3333333333333333
Epoch 600 | Running Loss: 1.218111515045166 | Accuracy: 0.3333333333333333
Epoch 700 | Running Loss: 1.218111515045166 | Accuracy: 0.3333333333333333
Epoch 800 | Running Loss: 1.218111515045166 | Accuracy: 0.3333333333333333
Epoch 900 | Running Loss: 1.218111515045166 | Accuracy: 0.3333333333333333
Epoch 1000 | Running Loss: 1.218111515045166 | Accuracy: 0.333333333333333
3
```



Observation

When the learning rate is increased to **100**, the model's loss and accuracy remain constant over all epochs, with accuracy stuck at approximately **33%**, which is equivalent to random guessing among three classes.

Explanation

An excessively high learning rate causes the optimizer to take steps that are too large, which stops the model from learning, and the learning rate does not increase and the performance is stuck.