Assignment 1

Part 1

a: In your opinion, what were the most important turning points in the history of deep learning?

There have been many important turning points in the history of deep learning. From the development of the first artificial neuron in the 1940s to today, a lot has happened. To me personally, the most important turning points have been the advancements in visual recognition. This is a key moment where deep learning began to show a clear advantage over traditional machine learning.

b: Explain the ADAM optimizer.

The typical optimizer uses some form of gradient descent to find a minimum loss. Problems occur when there are local minima along this path. The algorithm might get stuck and never optimize for minimal loss. Momentum is a way to reduce this problem. The momentum algorithm remembers the direction of previous iterations and pushes in that direction. Another technique that accelerates convergence is by adjusting the learning rate. This is known as the adaptive gradient algorithm, where the learning rate becomes smaller over time as it's divided by the sum of all previous gradients.

The ADAM optimizer combines both of these features, making it one of the fastest optimizers to converge and, therefore, quite convenient. However, studies have shown that standard SGD with momentum can perform better overall.

c: Assume data input is a single 30x40 pixel image. First layer is a convolutional layer with 5 filters, with kernel size 3x2, step size (1,1) and padding='valid'. What are the output dimensions?

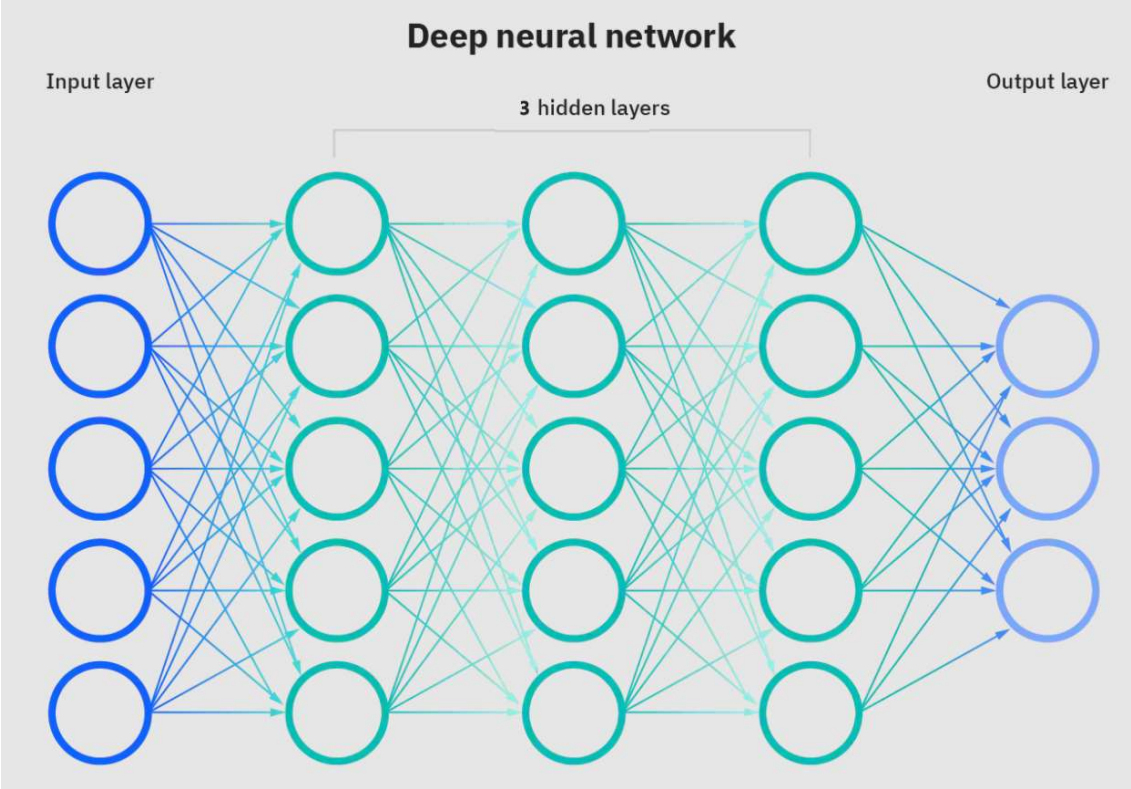
Since the padding is valid, it means that we wont add a padding around the image. The step size is (1,1), so the kernel will only move one step at the time. We have 5 filters, whih means that there will be 5 dimensions.

$$\text{Output Height} = \frac{\text{Input Height} - \text{Kernel height}}{\text{Stide Height}} + 1$$

$$\text{Output Width} = \frac{\text{Input Width} - \text{Kernel Width}}{\text{Stide Width}} + 1$$

Following these equations, the output dimensions will be 28x39x5, since we have no padding.

d: Assuming ReLU activations and offsets, and that the last layer is softmax, how many parameters does this network have:



The inputer layer is 5 and we have 3 hidden layers of size 5 and an output layer of 3. The number of parameters will be:

$$\text{Parameters} = (5 \cdot 5 + 5) + (5 \cdot 5 + 5) + (5 \cdot 5 + 5) + (5 \cdot 3 + 3) = 3 \cdot 30 + 18 = 108$$

e: For a given minibatch, the targets are [1,4, 5, 8] and the network output is [0.1,4.4,0.2,10]. If the loss function is "torch.nn.HuberLoss(reduction='mean', delta=1.0)", what is the loss for this minibatch?

For a batch of size N, the unreduced loss can be described as:

$$L = \{l_1, \dots, l_N\}^T$$

with

$$l_n = \begin{cases} \frac{1}{2}(x_n - y_n)^2, & \text{if } |x_n - y_n| < \delta \\ \delta\left(|x_n - y_n| - \frac{1}{2}\delta\right), & \text{otherwise} \end{cases}$$

If reduction is not none, then:

$$\ell(x, y) = \begin{cases} \text{mean}(L), & \text{if reduction} = \text{'mean'} \\ \text{sum}(L), & \text{if reduction} = \text{'sum'} \end{cases}$$

This will give the result of:

$$L = \left\{\frac{1}{2}(1 - 0.1)^2, \frac{1}{2}(4 - 4.4)^2, \delta\left(|5 - 0.2| - \frac{1}{2}\delta\right), \delta\left(|8 - 10| - \frac{1}{2}\delta\right)\right\}^T$$

$$L = \{0.405, 0.08, 4.3, 1.5\}^T$$

$$\ell(x, y) = \frac{0.405 + 0.08 + 4.3 + 1.5}{4} = \frac{6.285}{4} = 1.57125$$

The loss of the minibatch will be 1.57125

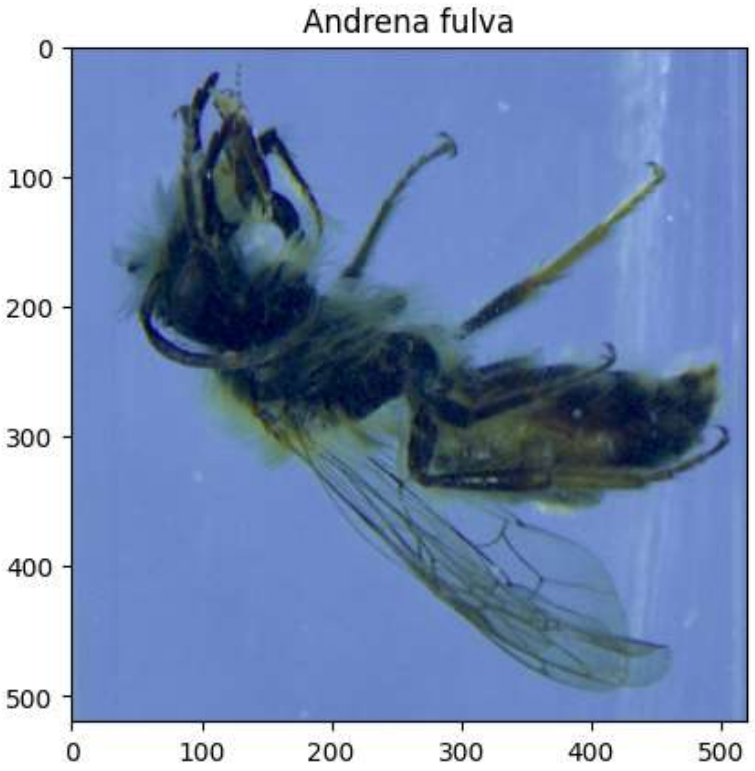
Part 2: Writing a PyTorch dataset

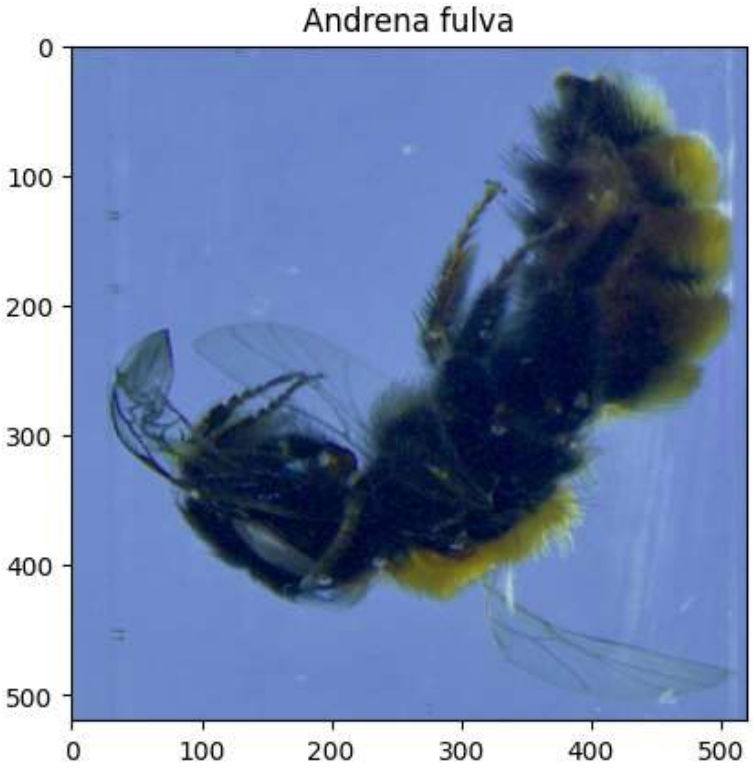
In [1]:
```
import os
import pandas as pd
from PIL import Image
from torch.utils.data import Dataset
from torchvision import transforms
import torch
import matplotlib.pyplot as plt
import numpy as np
from torch import nn
```

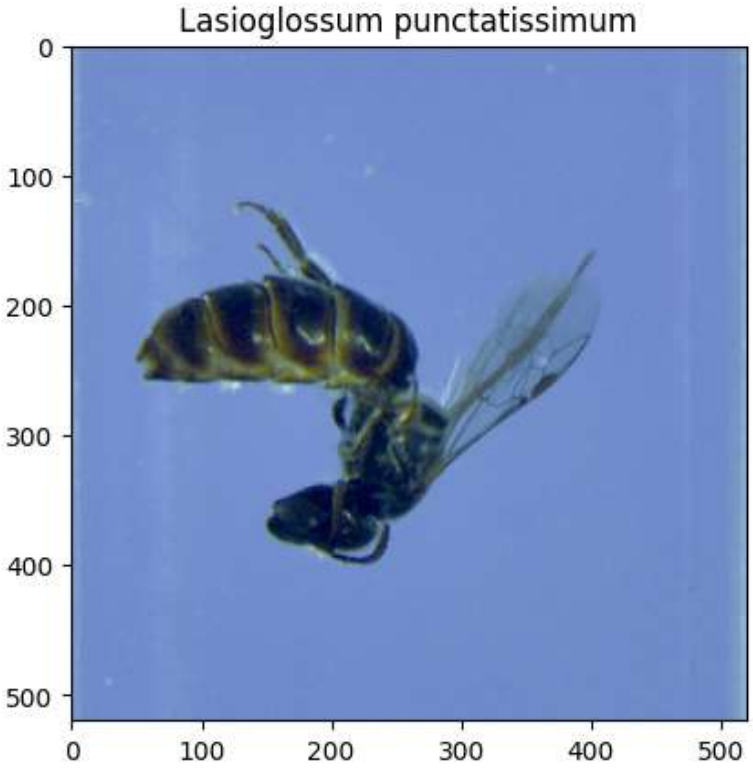
In [2]:
```
class InsectsDataset(Dataset):
    def __init__(self, csv_file, image_folder, root_dir, transform=None):
        """
        Args:
            root_dir (string): Directory with csv and image folder.
            csv_file (string): Path to the csv file with filenames and species.
            image_folder (string): Directory with images.
            transform (callable, optional): Optional transform to be applied on a sample.
        """
        self.root_dir = root_dir
        self.image_folder = image_folder
        csv_path =  os.path.join(self.root_dir, csv_file)
        self.insects_df = pd.read_csv(csv_path)
        self.transform = transform

        # Create a dictionary with the species as keys and a unique index as values
        self.species_to_label = {species: idx for idx, species in enumerate(self.insects_d
        self.label_to_species = {idx: species for species, idx in self.species_to_label.it

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

    def __getitem__(self, idx):
        # Get the image file path
        img_name = os.path.join(self.root_dir, self.image_folder, self.insects_df.iloc[idx
        image = Image.open(img_name)

        # Get the species label
        species = self.insects_df.iloc[idx, 1]  # assuming 'species' is the second column
        label = self.species_to_label[species]

        # Apply transformations if needed
        if self.transform:
            image = self.transform(image)

        return image, label

transform = transforms.Compose([
    transforms.Resize((520, 520)), # Resizing the image to 520x520
    transforms.ToTensor()
])

dataset = InsectsDataset(csv_file='insects.csv',image_folder = "Insects", root_dir='data/

print(dataset.species_to_label)
print(dataset.label_to_species)
```

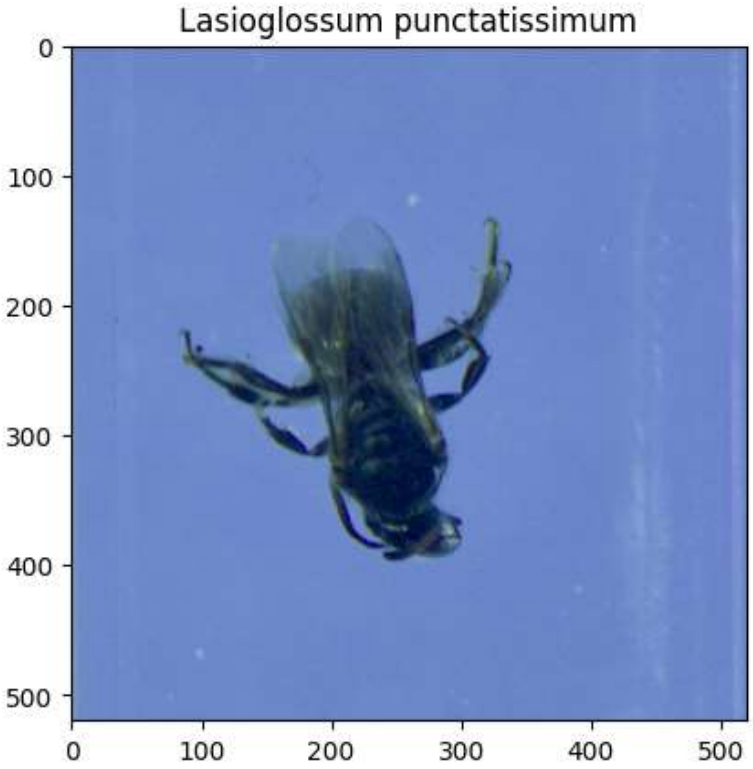
```
{'Andrena fulva': 0, 'Panurgus banksianus': 1, 'Lasioglossum punctatissimum': 2}
{0: 'Andrena fulva', 1: 'Panurgus banksianus', 2: 'Lasioglossum punctatissimum'}
```

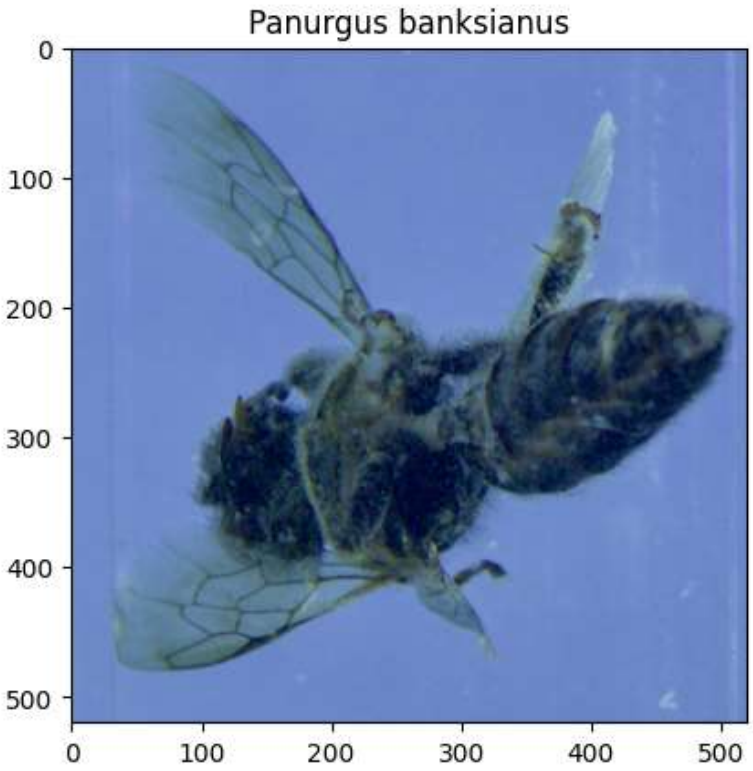
In [3]:
```
batch_size = 4

# Set up the dataset.
dataset = dataset

# Set up the dataset.
trainloader = torch.utils.data.DataLoader(dataset,
                                          batch_size=batch_size,
                                          shuffle=True,
                                          num_workers=0)

# get some images
dataiter = iter(trainloader)
images, labels = next(dataiter)


for i in range(2): #Run through 5 batches
    images, labels = next(dataiter)
    for image, label in zip(images,labels): # Run through all samples in a batch
        plt.figure()
        plt.imshow(np.transpose(image.numpy(), (1, 2, 0)))
        plt.title(dataset.label_to_species[label.item()])
```

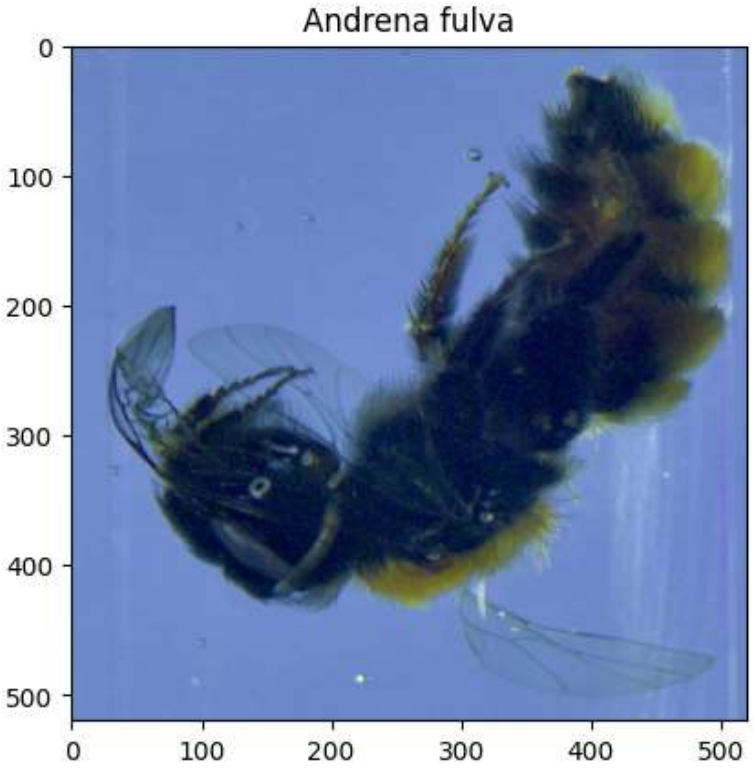


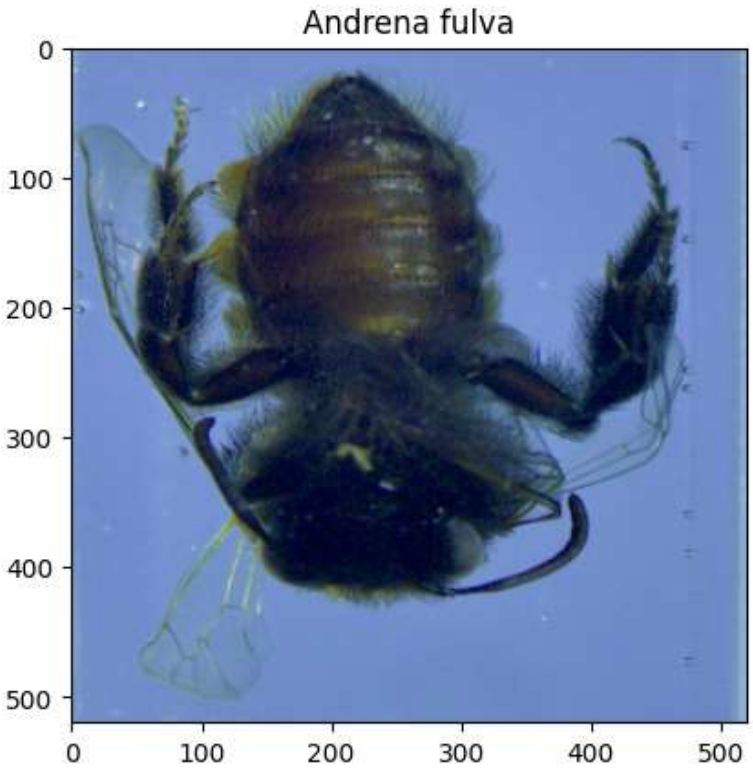

Andrena fulva
0
100
200
300
400
500
0
100
200
300
400
500

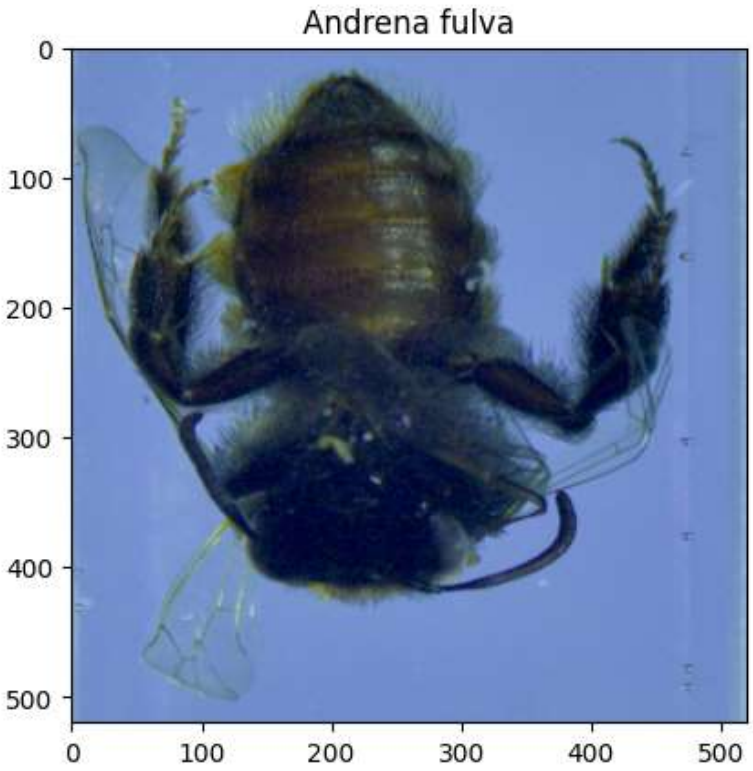

Lasioglossum punctatissimum
0
100
200
300
400
500
0
100
200
300
400
500


Lasioglossum punctatissimum
0
100
200
300
400
500
0
100
200
300
400
500


Panurgus banksianus
0
100
200
300
400
500
0
100
200
300
400
500


Andrena fulva
0
100
200
300
400
500
0
100
200
300
400
500


Andrena fulva
0
100
200
300
400
500
0
100
200
300
400
500



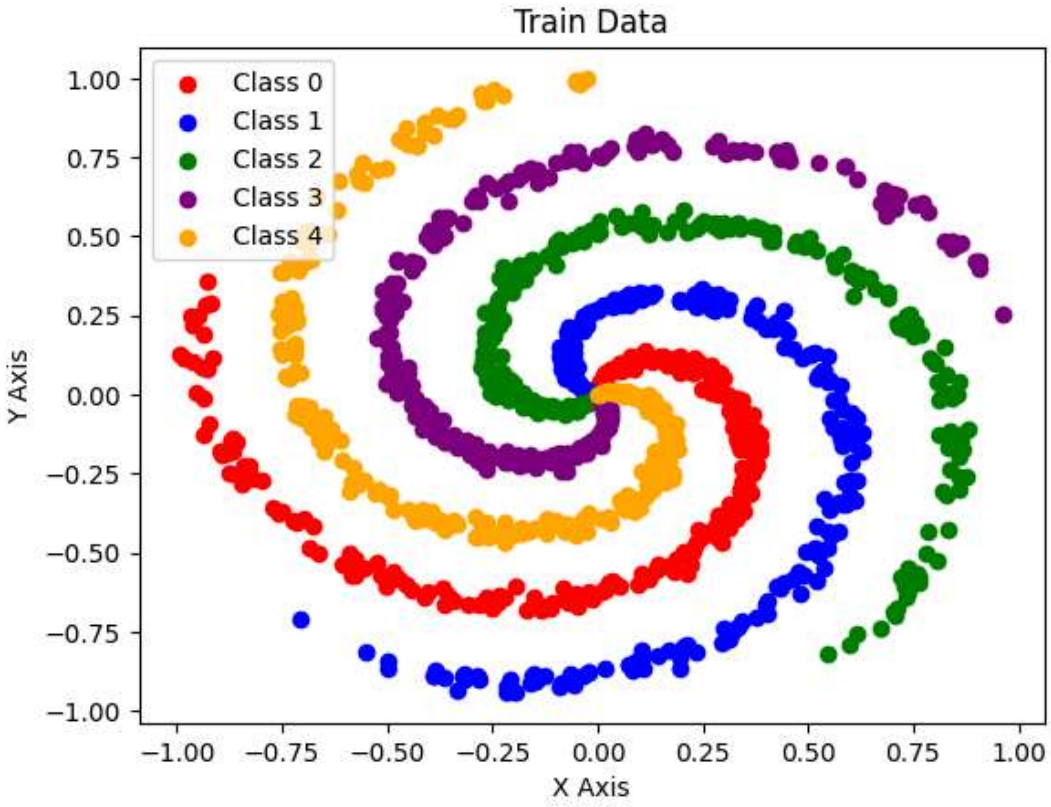
Part 3:

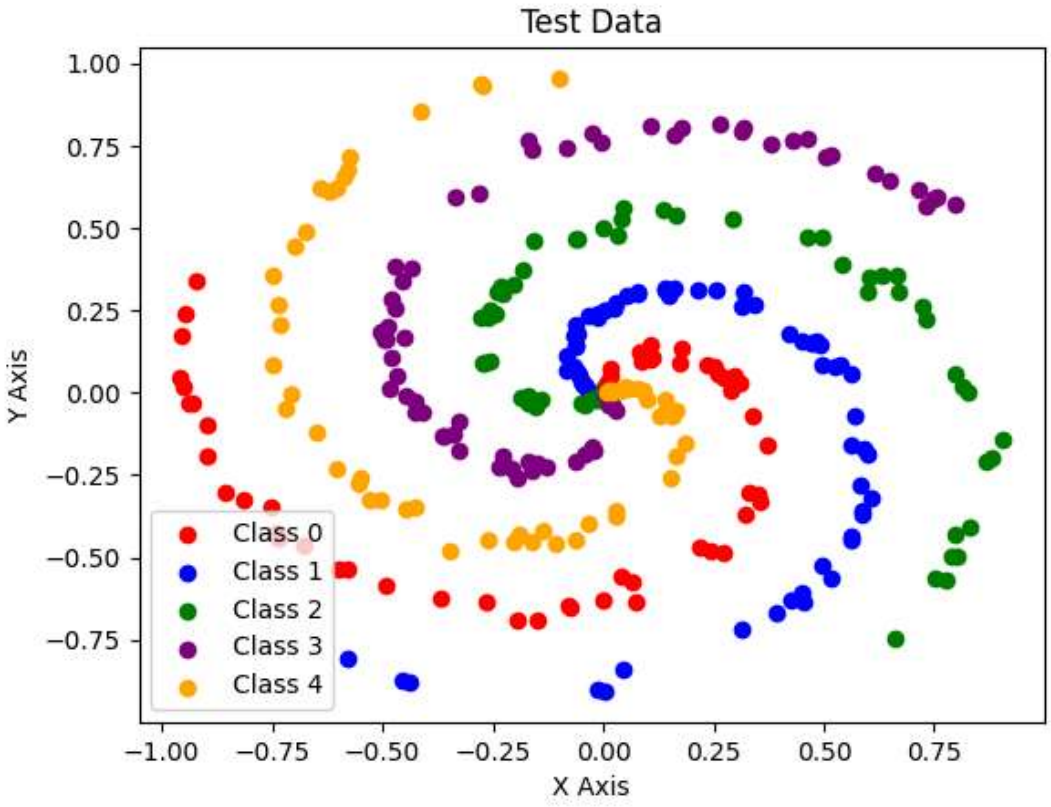
In [4]:
```python
class TwoDimensionalData:
    def __init__(self, train_csv, test_csv, root_dir):
        """
        Args:
            root_dir (string): Directory with csv
            csv_file (string): Path to the csv file with filenames and species.
        """
        self.root_dir = root_dir
        train_path =  os.path.join(self.root_dir, train_csv)
        test_path =  os.path.join(self.root_dir, test_csv)
        self.train_df = pd.read_csv(train_path, sep=";")
        self.test_df = pd.read_csv(test_path, sep=";")

    def plot_data(self,df, title="Data Plot"):
        # Plot x,y with with color based on the Class with matplotlib
        # Use the color_map to assign colors to the classes
        color_map = {0: 'red', 1: 'blue', 2: 'green', 3: 'purple', 4: 'orange'}
        plt.figure()
        for class_value, color in color_map.items():
            plt.scatter(df[df['Class'] == class_value]['X'], df[df['Class'] == class_value
        plt.title(title)
        plt.xlabel('X Axis')
        plt.ylabel('Y Axis')
        plt.legend()
        plt.show()

    def get_data_to_tensor(self, df):
        # Convert the data to a tensor
        data = torch.tensor(df[['X', 'Y']].values, dtype=torch.float32)
        labels = torch.tensor(df['Class'].values, dtype=torch.long)
        return data, labels
twoD = TwoDimensionalData('train_data.csv', 'test_data.csv', 'data/')
```

a: describe & visualize the data

The data looks like a Spiral galaxy, with five arms. Each of the arms have a different class. The data is plotted below.

In [5]:
```
twoD.plot_data(twoD.train_df, "Train Data")
twoD.plot_data(twoD.test_df, title="Test Data")
```





b: design a neural network using pytorch to correctly assign labels

In [6]:
```
class NNclassifier(nn.Module):
    def __init__(self, input_size, num_classes):
```

```
        super(NNclassifier, self).__init__()
        # Linear layers
        self.fc1 = nn.Linear(input_size, 256)
        self.fc2 = nn.Linear(256, 64)
        self.fc3 = nn.Linear(64, num_classes)

        # Activation function
        self.relu = nn.ReLU()

    def forward(self, x):
        out = self.fc1(x)
        out = self.relu(out)
        out = self.fc2(out)
        out = self.relu(out)
        out = self.fc3(out)
        return out

# Hyperparameters
input_size = 2
num_classes = 5
learning_rate = 0.001

# Device configuration
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

# Load the data
dataset = TwoDimensionalData('train_data.csv', 'test_data.csv', 'data/')
train_data, train_labels = dataset.get_data_to_tensor(dataset.train_df)
test_data, test_labels = dataset.get_data_to_tensor(dataset.test_df)

train_dataset = torch.utils.data.TensorDataset(train_data, train_labels)
test_dataset = torch.utils.data.TensorDataset(test_data, test_labels)

train_dataloader = torch.utils.data.DataLoader(train_dataset, batch_size=batch_size, shuf
test_dataloader = torch.utils.data.DataLoader(test_dataset, batch_size=batch_size, shuffle

# Initialize the neural network
model = NNclassifier(input_size, num_classes).to(device)

# Loss and optimizer
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)

class trainNN:
    train_losses = []
    test_losses = []
    train_accuracies = []
    test_accuracies = []
    accuracy = []
    convergence = False

    def __init__(self, model, train_dataloader, test_dataloader, criterion, optimizer, nu
        self.model = model
        self.train_dataloader = train_dataloader
        self.test_dataloader = test_dataloader
        self.criterion = criterion
        self.optimizer = optimizer
        self.num_epochs = num_epochs

    def _train(self, dataloader, model, loss_fn, optimizer):
        size = len(dataloader.dataset)
        model.train()
        total_loss, correct = 0, 0  # Track the total loss for the epoch

        for _, (X, y) in enumerate(dataloader):
            X, y = X.to(device), y.to(device)
```

```
            # Compute prediction error
            pred = model(X)
            loss = loss_fn(pred, y)
            total_loss += loss.item()  # Accumulate loss

            # Backpropagation
            loss.backward()
            optimizer.step()
            optimizer.zero_grad()

            # Calculate accuracy
            correct += (pred.argmax(1) == y).type(torch.float).sum().item()

        avg_loss = total_loss / len(dataloader)  # Calculate average loss for the epoch
        avg_accuracy = correct / size

        self.train_losses.append(avg_loss)  # Store the average loss for the epoch
        self.train_accuracies.append(avg_accuracy) # Store the average accuracy for the ep
        print(f"Train loss: {avg_loss:>7f}, Accuracy: {(100*avg_accuracy):>0.1f}%")

    def _test(self, dataloader, model, loss_fn):
        size = len(dataloader.dataset)
        num_batches = len(dataloader)
        model.eval()
        test_loss, correct = 0, 0
        with torch.no_grad():
            for X, y in dataloader:
                X, y = X.to(device), y.to(device)
                pred = model(X)
                test_loss += loss_fn(pred, y).item()
                correct += (pred.argmax(1) == y).type(torch.float).sum().item()
        test_loss /= num_batches
        correct /= size
        self.test_losses.append(test_loss)  # Store the test loss for the epoch
        self.test_accuracies.append(correct)

        print(f"Test Error: \nAvg loss: {test_loss:>8f}, Accuracy: {(100*correct):>0.1f}%
        return float(correct*100)

    def _plot_training_result(self):
        plt.subplot(2,1,1)
        plt.plot(range(1, self.num_epochs+1), self.train_losses, label="Train Loss")
        plt.plot(range(1, self.num_epochs+1), self.test_losses, label="Test Loss")
        plt.xlabel("Epochs")
        plt.ylabel("Loss")
        plt.title("Training and Testing Loss over Epochs")
        plt.grid()
        plt.legend()
        plt.show()

        plt.subplot(2,1,2)
        plt.plot(range(1, self.num_epochs+1), self.train_accuracies, label="Train Accuracy
        plt.plot(range(1, self.num_epochs+1), self.test_accuracies, label="Test Accuracy")
        plt.xlabel("Epochs")
        plt.ylabel("Accuracy")
        plt.title("Training and Testing Accuracy over Epochs")
        plt.grid()
        plt.legend()
        plt.show()

    def train_model(self):
        accuracy = []
        convergence = False
        for t in range(self.num_epochs):
            print(f"Epoch {t+1}\n-------------------------------")
```

```
            self._train(self.train_dataloader, self.model, self.criterion, self.optimizer)
            test_acc = self._test(self.test_dataloader, self.model, self.criterion)
            accuracy.append(test_acc)
            # Check for convergence, meaning the test loss has not decreased for 30 epochs
            if t > 31 and self.test_losses[-1] >= np.mean(self.test_losses[-31:-1]) and no
                convergence = True
            if convergence and accuracy[-1] >= np.max(accuracy):
                self.num_epochs = t + 1
                break
        print("Done!")
        self._plot_training_result()
```

In [7]:
```
test = trainNN(model, train_dataloader, test_dataloader, criterion, optimizer, 1000)
test.train_model()
```

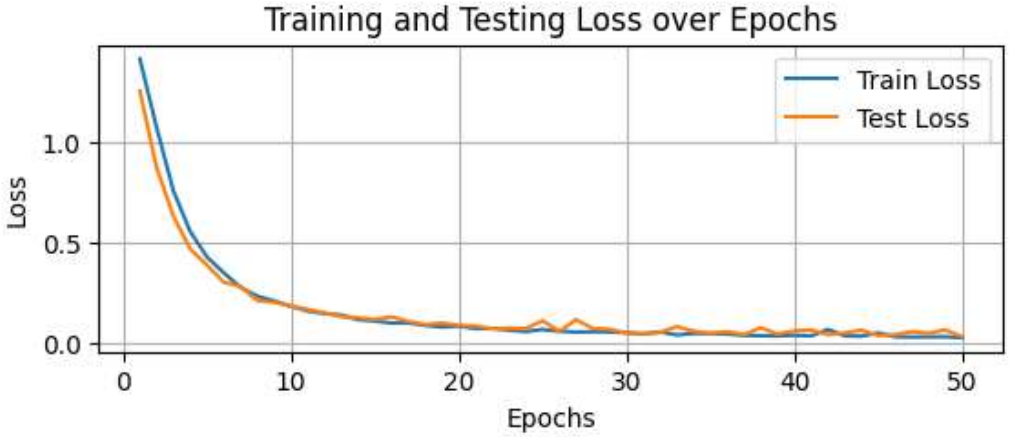
```
Epoch 1
-------------------------------
Train loss: 1.412095, Accuracy: 30.2%
Test Error:
Avg loss: 1.253044, Accuracy: 42.1%

Epoch 2
-------------------------------
Train loss: 1.070897, Accuracy: 55.3%
Test Error:
Avg loss: 0.870811, Accuracy: 65.6%

Epoch 3
-------------------------------
Train loss: 0.756207, Accuracy: 72.8%
Test Error:
Avg loss: 0.630442, Accuracy: 76.9%

Epoch 4
-------------------------------
Train loss: 0.555226, Accuracy: 79.5%
Test Error:
Avg loss: 0.467917, Accuracy: 85.3%

Epoch 5
-------------------------------
Train loss: 0.428471, Accuracy: 85.4%
Test Error:
Avg loss: 0.388163, Accuracy: 87.6%

Epoch 6
-------------------------------
Train loss: 0.349995, Accuracy: 86.8%
Test Error:
Avg loss: 0.302842, Accuracy: 89.0%

Epoch 7
-------------------------------
Train loss: 0.277726, Accuracy: 90.2%
Test Error:
Avg loss: 0.281831, Accuracy: 91.3%

Epoch 8
-------------------------------
Train loss: 0.233534, Accuracy: 92.6%
Test Error:
Avg loss: 0.211579, Accuracy: 91.3%

Epoch 9
-------------------------------
Train loss: 0.210135, Accuracy: 92.1%
Test Error:
Avg loss: 0.202949, Accuracy: 91.6%

Epoch 10
-------------------------------
Train loss: 0.183296, Accuracy: 93.7%
Test Error:
Avg loss: 0.184218, Accuracy: 91.3%

Epoch 11
-------------------------------
Train loss: 0.160243, Accuracy: 94.5%
Test Error:
Avg loss: 0.167812, Accuracy: 94.3%
```

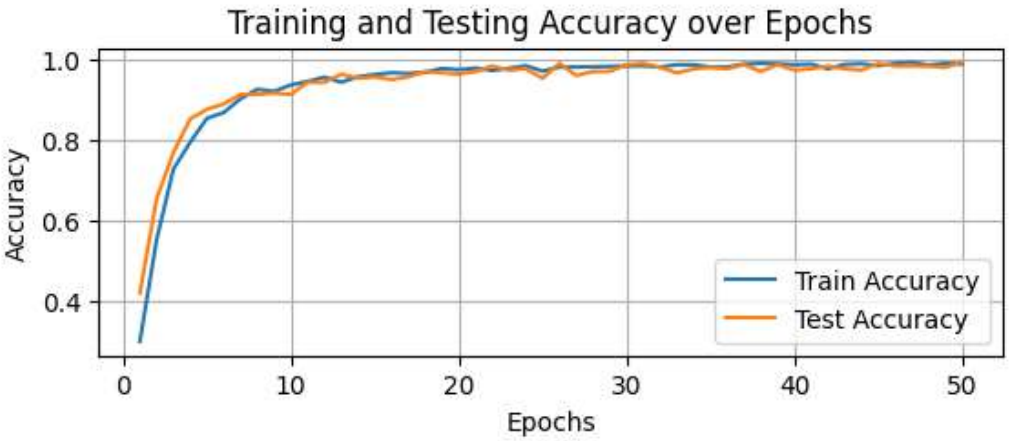
```
Epoch 12
-------------------------------
Train loss: 0.147170, Accuracy: 95.6%
Test Error:
Avg loss: 0.150958, Accuracy: 94.3%

Epoch 13
-------------------------------
Train loss: 0.139857, Accuracy: 94.3%
Test Error:
Avg loss: 0.132336, Accuracy: 96.3%

Epoch 14
-------------------------------
Train loss: 0.116900, Accuracy: 95.7%
Test Error:
Avg loss: 0.126748, Accuracy: 95.3%

Epoch 15
-------------------------------
Train loss: 0.110410, Accuracy: 96.2%
Test Error:
Avg loss: 0.118313, Accuracy: 95.7%

Epoch 16
-------------------------------
Train loss: 0.100038, Accuracy: 96.7%
Test Error:
Avg loss: 0.131014, Accuracy: 95.0%

Epoch 17
-------------------------------
Train loss: 0.100529, Accuracy: 96.5%
Test Error:
Avg loss: 0.108257, Accuracy: 95.7%

Epoch 18
-------------------------------
Train loss: 0.088449, Accuracy: 96.8%
Test Error:
Avg loss: 0.093085, Accuracy: 97.0%

Epoch 19
-------------------------------
Train loss: 0.080294, Accuracy: 97.8%
Test Error:
Avg loss: 0.100391, Accuracy: 96.7%

Epoch 20
-------------------------------
Train loss: 0.084843, Accuracy: 97.4%
Test Error:
Avg loss: 0.088513, Accuracy: 96.3%

Epoch 21
-------------------------------
Train loss: 0.072644, Accuracy: 97.8%
Test Error:
Avg loss: 0.085064, Accuracy: 97.0%

Epoch 22
-------------------------------
Train loss: 0.073089, Accuracy: 97.2%
Test Error:
Avg loss: 0.071873, Accuracy: 98.3%
```

```
Epoch 23
-------------------------------
Train loss: 0.065152, Accuracy: 97.8%
Test Error:
Avg loss: 0.075005, Accuracy: 97.3%

Epoch 24
-------------------------------
Train loss: 0.058062, Accuracy: 98.4%
Test Error:
Avg loss: 0.072698, Accuracy: 97.7%

Epoch 25
-------------------------------
Train loss: 0.068026, Accuracy: 97.1%
Test Error:
Avg loss: 0.111721, Accuracy: 95.3%

Epoch 26
-------------------------------
Train loss: 0.059478, Accuracy: 98.1%
Test Error:
Avg loss: 0.059249, Accuracy: 99.0%

Epoch 27
-------------------------------
Train loss: 0.055335, Accuracy: 98.1%
Test Error:
Avg loss: 0.116745, Accuracy: 96.0%

Epoch 28
-------------------------------
Train loss: 0.056902, Accuracy: 98.2%
Test Error:
Avg loss: 0.072702, Accuracy: 97.0%

Epoch 29
-------------------------------
Train loss: 0.055891, Accuracy: 98.2%
Test Error:
Avg loss: 0.069560, Accuracy: 97.0%

Epoch 30
-------------------------------
Train loss: 0.052457, Accuracy: 98.3%
Test Error:
Avg loss: 0.050184, Accuracy: 98.7%

Epoch 31
-------------------------------
Train loss: 0.046009, Accuracy: 98.4%
Test Error:
Avg loss: 0.048548, Accuracy: 99.0%

Epoch 32
-------------------------------
Train loss: 0.056049, Accuracy: 98.1%
Test Error:
Avg loss: 0.053807, Accuracy: 98.0%

Epoch 33
-------------------------------
Train loss: 0.041201, Accuracy: 98.7%
Test Error:
Avg loss: 0.082201, Accuracy: 96.7%
```

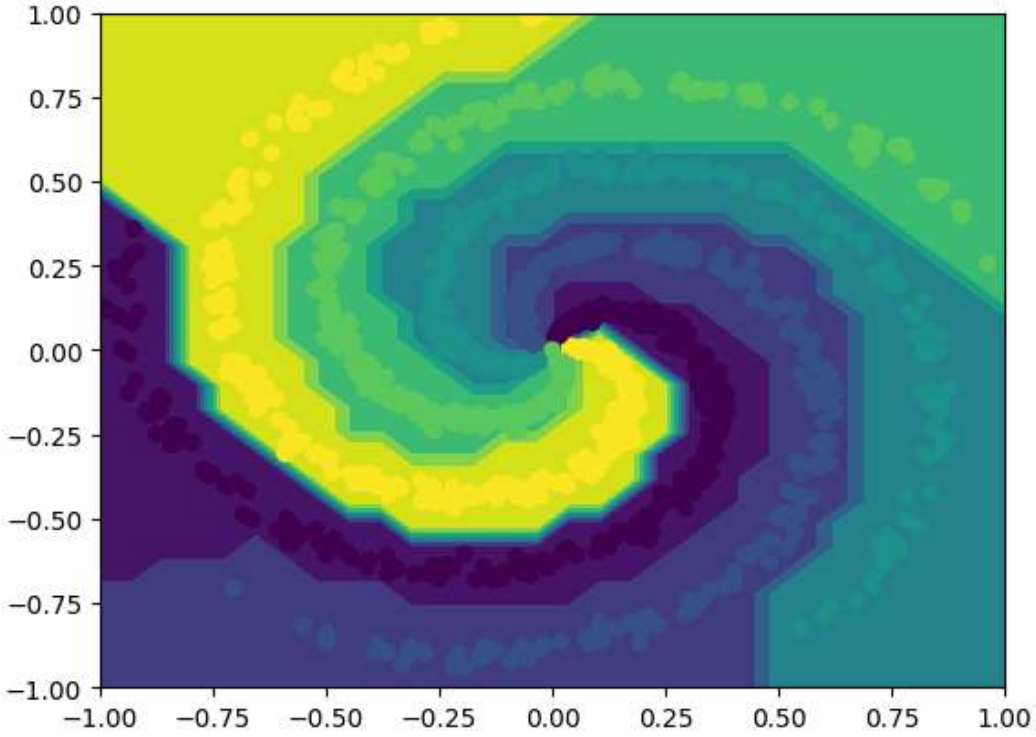
```
Epoch 34
-------------------------------
Train loss: 0.048475, Accuracy: 98.7%
Test Error:
Avg loss: 0.059776, Accuracy: 97.7%

Epoch 35
-------------------------------
Train loss: 0.047802, Accuracy: 98.1%
Test Error:
Avg loss: 0.049374, Accuracy: 98.0%

Epoch 36
-------------------------------
Train loss: 0.045964, Accuracy: 98.1%
Test Error:
Avg loss: 0.057413, Accuracy: 97.7%

Epoch 37
-------------------------------
Train loss: 0.038544, Accuracy: 98.8%
Test Error:
Avg loss: 0.042291, Accuracy: 98.7%

Epoch 38
-------------------------------
Train loss: 0.037254, Accuracy: 99.0%
Test Error:
Avg loss: 0.077234, Accuracy: 97.0%

Epoch 39
-------------------------------
Train loss: 0.036103, Accuracy: 98.9%
Test Error:
Avg loss: 0.046188, Accuracy: 98.7%

Epoch 40
-------------------------------
Train loss: 0.039386, Accuracy: 98.7%
Test Error:
Avg loss: 0.061046, Accuracy: 97.3%

Epoch 41
-------------------------------
Train loss: 0.036142, Accuracy: 98.8%
Test Error:
Avg loss: 0.065662, Accuracy: 97.7%

Epoch 42
-------------------------------
Train loss: 0.068177, Accuracy: 97.7%
Test Error:
Avg loss: 0.042533, Accuracy: 98.3%

Epoch 43
-------------------------------
Train loss: 0.036268, Accuracy: 98.8%
Test Error:
Avg loss: 0.050800, Accuracy: 97.7%

Epoch 44
-------------------------------
Train loss: 0.034831, Accuracy: 98.9%
Test Error:
Avg loss: 0.064737, Accuracy: 97.3%
```

```
Epoch 45
-------------------------------
Train loss: 0.051191, Accuracy: 98.4%
Test Error:
Avg loss: 0.038016, Accuracy: 99.0%

Epoch 46
-------------------------------
Train loss: 0.031838, Accuracy: 99.0%
Test Error:
Avg loss: 0.042148, Accuracy: 98.3%

Epoch 47
-------------------------------
Train loss: 0.030869, Accuracy: 99.2%
Test Error:
Avg loss: 0.057429, Accuracy: 98.3%

Epoch 48
-------------------------------
Train loss: 0.031805, Accuracy: 98.6%
Test Error:
Avg loss: 0.049654, Accuracy: 98.3%

Epoch 49
-------------------------------
Train loss: 0.032123, Accuracy: 99.0%
Test Error:
Avg loss: 0.066229, Accuracy: 98.0%

Epoch 50
-------------------------------
Train loss: 0.027077, Accuracy: 98.8%
Test Error:
Avg loss: 0.031959, Accuracy: 99.3%

Done!
```





In [8]:
```
x,y = np.meshgrid(np.linspace(-1,1,30),np.linspace(-1,1,30))
xy = np.concatenate((x.reshape(-1,1),y.reshape(-1,1)),axis=1)
z = model(torch.tensor(xy).float()).detach().numpy()
z = np.argmax(z,1).reshape(30,30)
plt.contourf(x,y,z)
plt.scatter(train_data[:,0],train_data[:,1],c=train_labels)
plt.show()
```



2D feature space network

The network I have designed consits of two layers. Where the 2-dimension input goes to 256 neurons to 64 to the final 5 neurons, that is the output layer. I have drawn a lot of inspiration from exercises from week 3, which have provided a very stable network.

I have trained my network untill it have converged. I have defined this as the current loss is higher than or equal to the average of the last 30 epochs. When the network have converged, I wait until the network gets the best performance on the training data.

The method I have chosen work really well for this scenario, but might not work optimal with different data. If there is a "big" local minima, the algorithm will think it have converged, and will

break when the accuracy will peak. This method works best with the ADAM optimizer, that uses momentum to push forward.

The network reaches uptill 99.7% accuracy on the test data, which is quite good. The meshgrid plot also shows that the different classes gets labled correctly.