Spring 2022 Introduction to Deep Learning
Homework Assignment 3

Name: Lohitanvita Rompicharla **NetID: lr701**

Problem (Build a FC layer-based neural network to recognize hand-written digits).

In this problem, you are asked to train and test a neural network for entire MNIST hand-written digit dataset. Some information of the network is as follows:

- Its structure is 784-200-50-10. Here 784 means the input layer has 784 input neurons. This is because each image in MNIST dataset is 28x28 and you need to stretch them to a length-784 vector. 200 and 50 are the number of neurons in hidden layers. 10 is the number of neurons in output layer since there are 10 types of digits.
- The two hidden layers are followed by ReLU layers.
- The output layer is a softmax layer.

Performance Requirement and Submission:

- The test accuracy should achieve above 95%
- Submission should include your source codes and screen snapshot of your train and test accuracy, plus the training time

Suggestion for hyperparameter setting (not necessary to follow):

- Learning rate can be set as 0.01
- If you choose to use mini-batch SGD, the batch size can be set as 128
- Number of epochs can be set as 10.

(a) (Mandatory) Use deep learning framework to train and test this network. You are allowed to use the corresponding autograd or nn module to train the network.

Source Code:

```
import numpy as np
import torch
import matplotlib.pyplot as plt
from time import time
from torchvision import datasets, transforms
from torch import nn
from torch import optim


# Defining a transform to normalize the data
transform = transforms.Compose([transforms.ToTensor(),
                                transforms.Normalize((0.5,), (0.5,)),
                                ])

# Download and load the training data
training_data = datasets.MNIST(r'C:\Users\rlohi\Downloads', download=True,
```

```
train=True, transform=transform)
validation_data = datasets.MNIST(r'C:\Users\rlohi\Downloads', download=True,
train=False, transform=transform)
train_loader = torch.utils.data.DataLoader(training_data, batch_size=128,
shuffle=True)
val_loader = torch.utils.data.DataLoader(validation_data , batch_size=128,
shuffle=True)

data_iterator = iter(train_loader)
images, labels = data_iterator.next()
print(type(images))
print(images.shape)
print(labels.shape)

file = r'C:\Users\rlohi\Downloads'
figure = plt.figure()
num_of_images = 60
for index in range(1, num_of_images + 1):
    plt.subplot(6, 10, index)
    plt.axis('off')
    plt.imshow(images[index].numpy().squeeze(), cmap='gray_r')
plt.show()
plt.savefig(file+'\\fig2.png')
plt.close('all')

# Layer details for the neural network
input_size = 784
hidden_sizes = [200, 50]
output_size = 10

# Build a feed-forward network
NN_architecture = nn.Sequential(nn.Linear(input_size, hidden_sizes[0]),
                      nn.ReLU(),
                      nn.Linear(hidden_sizes[0], hidden_sizes[1]),
                      nn.ReLU(),
                      nn.Linear(hidden_sizes[1], output_size),
                      nn.Softmax(dim=1))
print(NN_architecture)

criterion = nn.NLLLoss()
images, labels = next(iter(train_loader))
images = images.view(images.shape[0], -1)

probability = NN_architecture(images)
loss = criterion(probability, labels)

print('Before backward pass: \n', NN_architecture[0].weight.grad)
loss.backward()
print('After backward pass: \n', NN_architecture[0].weight.grad)



# Optimizers require the parameters to optimize and a learning rate
optimizer = optim.SGD(NN_architecture.parameters(), lr=0.01, momentum=0.5)

print('Initial weights - ', NN_architecture[0].weight)
```

```
images, labels = next(iter(train_loader))
images.resize_(128, 784)

# Clear the gradients, do this because gradients are accumulated
optimizer.zero_grad()

# Forward pass, then backward pass, then update weights
output = NN_architecture(images)
loss = criterion(output, labels)
loss.backward()
print('Gradient -', NN_architecture[0].weight.grad)

# Take an update step and few the new weights
optimizer.step()
print('Updated weights - ', NN_architecture[0].weight)

#Core Training of NN
optimizer = optim.SGD(NN_architecture.parameters(), lr=0.003, momentum=0.9)
time0 = time()
epochs = 10
for e in range(epochs):
    running_loss = 0
    for images, labels in train_loader:
        # Flatten MNIST images into a 784 long vector
        images = images.view(images.shape[0], -1)

        # Training pass
        optimizer.zero_grad()

        output = NN_architecture(images)
        loss = criterion(output, labels)

        # This is where the NN_architecture learns by backpropagating
        loss.backward()

        # And optimizes its weights here
        optimizer.step()

        running_loss += loss.item()
    else:
        print("Epoch {} - Training loss: {}".format(e, running_loss /
len(train_loader)))
print("\nTraining Time (in minutes) =", (time() - time0) / 60)



def view_classify(self,img, ps, NN_architecture,val_loader):
    ''' Function for viewing an image and it's predicted classes.'''
    ps = ps.data.numpy().squeeze()

    fig, (ax1, ax2) = plt.subplots(figsize=(6,9), ncols=2)
    ax1.imshow(img.resize_(1, 28, 28).numpy().squeeze())
    ax1.axis('off')
    ax2.barh(np.arange(10), ps)
    ax2.set_aspect(0.1)
    ax2.set_yticks(np.arange(10))
    ax2.set_yticklabels(np.arange(10))
```

```
    ax2.set_title('Class Probability')
    ax2.set_xlim(0, 1.1)
    plt.tight_layout()

    images, labels = next(iter(val_loader))

    img = images[0].view(1, 784)
    # Turn off gradients to speed up this part
    with torch.no_grad():
        probability = NN_architecture(img)

    # Output of the network are log-probabilities, need to take exponential
for probabilities
    ps = torch.exp(probability)
    probab = list(ps.numpy()[0])
    print("Predicted Digit =", probab.index(max(probab)))
    self.view_classify(img.view(1, 28, 28), ps)


#NN_architecture Evaluation
correct_count, all_count = 0, 0
for images,labels in val_loader:
  for i in range(len(labels)):
    img = images[i].view(1, 784)
    # Turn off gradients to speed up this part
    with torch.no_grad():
        probability = NN_architecture(img)

    # Output of the network are log-probabilities, need to take exponential
for probabilities
    ps = torch.exp(probability)
    probab = list(ps.numpy()[0])
    pred_label = probab.index(max(probab))
    true_label = labels.numpy()[i]
    if(true_label == pred_label):
      correct_count += 1
    all_count += 1

print("Number Of Images Tested =", all_count)
print("\nModel Accuracy =", (correct_count/all_count))
```

// considering above given specification, I ran the code initially for 12-15 times to obtain the accuracy of 95%. My first training time gave an accuracy of 84%. I tried the code with output activation layeras both Softmax and LogSoftmax. Softmax took more training time compared to LogSoftmax to gain the required accuracy.

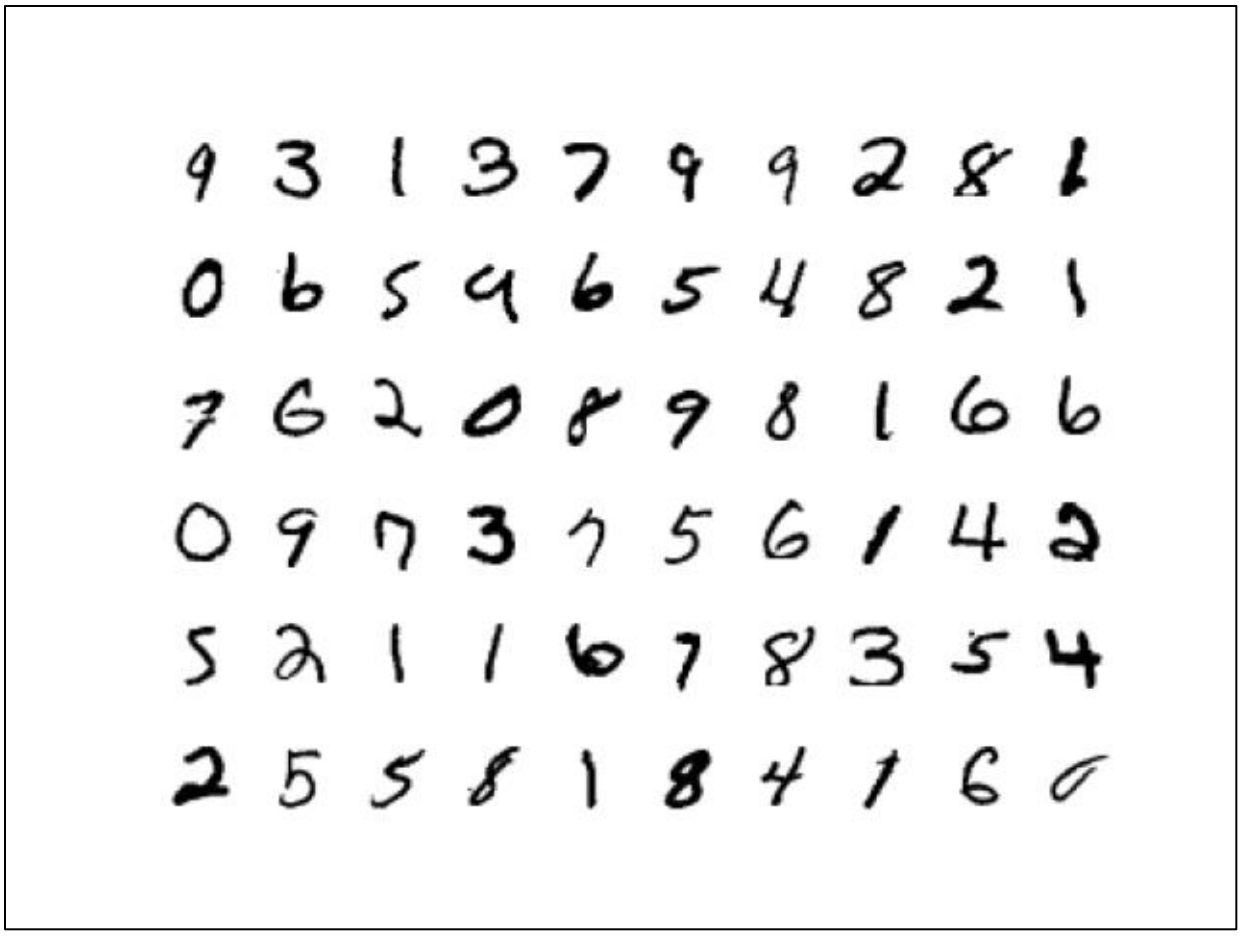
OUTPUT:

```
<class 'torch.Tensor'>
torch.Size([128, 1, 28, 28])

torch.Size([128])

Sequential(
  (0): Linear(in_features=784, out_features=200, bias=True)
  (1): ReLU()
  (2): Linear(in_features=200, out_features=50, bias=True)
  (3): ReLU()
  (4): Linear(in_features=50, out_features=10, bias=True)
  (5): LogSoftmax(dim=1)
)
Before backward pass:
 None
After backward pass:
 tensor([[-9.1588e-04, -9.1588e-04, -9.1588e-04,  ..., -9.1588e-04,
         -9.1588e-04, -9.1588e-04],
        [-4.7273e-05, -4.7273e-05, -4.7273e-05,  ..., -4.7273e-05,
         -4.7273e-05, -4.7273e-05],
        [ 3.7017e-04,  3.7017e-04,  3.7017e-04,  ...,  3.7017e-04,
          3.7017e-04,  3.7017e-04],
        ...,
        [-1.7279e-03, -1.7279e-03, -1.7279e-03,  ..., -1.7279e-03,
         -1.7279e-03, -1.7279e-03],
        [ 9.9673e-04,  9.9673e-04,  9.9673e-04,  ...,  9.9673e-04,
          9.9673e-04,  9.9673e-04],
        [ 1.6951e-03,  1.6951e-03,  1.6951e-03,  ...,  1.6951e-03,
          1.6951e-03,  1.6951e-03]])
Initial weights -  Parameter containing:
```

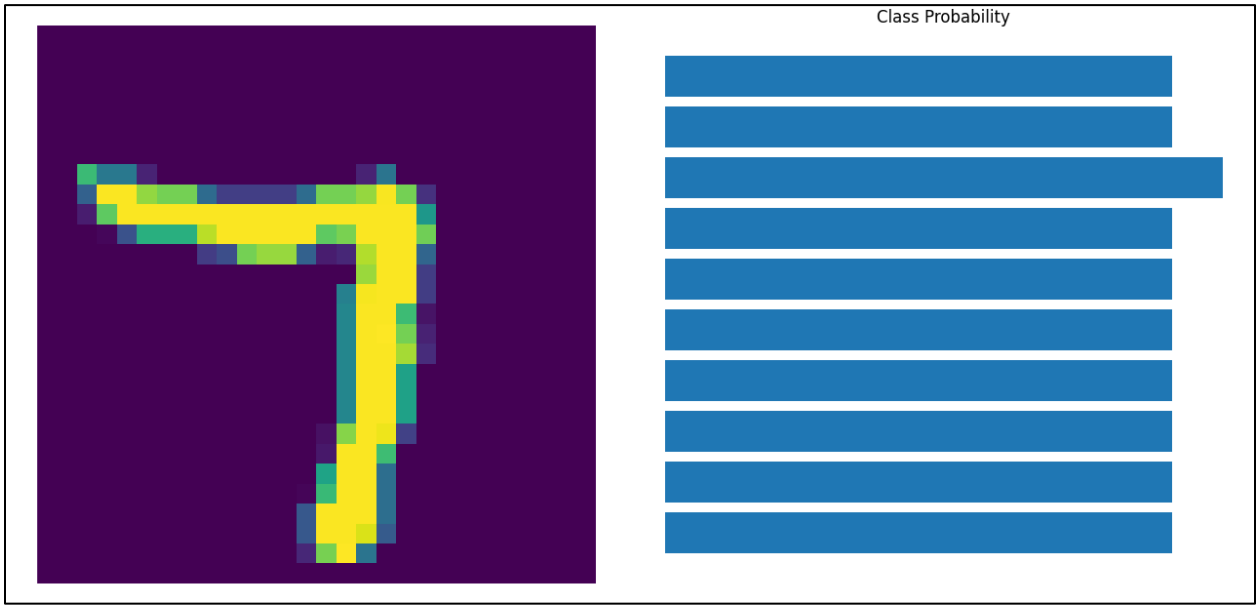
```
tensor([[ 0.0048,  0.0144, -0.0043,  ..., -0.0293,  0.0072, -0.0293],
        [ 0.0092, -0.0132, -0.0247,  ...,  0.0028,  0.0146, -0.0020],
        [-0.0156, -0.0094,  0.0008,  ..., -0.0175,  0.0170, -0.0243],
        ...,
        [-0.0343,  0.0353,  0.0152,  ...,  0.0182, -0.0198,  0.0134],
        [ 0.0066,  0.0031, -0.0254,  ..., -0.0215,  0.0192,  0.0210],
        [ 0.0032,  0.0288,  0.0308,  ...,  0.0063,  0.0088,  0.0051]],
       requires_grad=True)
Gradient - tensor([[ 2.7779e-05,  2.7779e-05,  2.7779e-05,  ...,  2.7779e-05,
          2.7779e-05,  2.7779e-05],
        [-1.7322e-03, -1.7322e-03, -1.7322e-03,  ..., -1.7322e-03,
         -1.7322e-03, -1.7322e-03],
        [-9.5710e-04, -9.5710e-04, -9.5710e-04,  ..., -9.5710e-04,
         -9.5710e-04, -9.5710e-04],
        ...,
        [-8.7384e-04, -8.7384e-04, -8.7384e-04,  ..., -8.7384e-04,
         -8.7384e-04, -8.7384e-04],
        [ 9.8977e-04,  9.8977e-04,  9.8977e-04,  ...,  9.8977e-04,
          9.8977e-04,  9.8977e-04],
        [ 1.0175e-03,  1.0175e-03,  1.0175e-03,  ...,  1.0175e-03,
          1.0175e-03,  1.0175e-03]])
Updated weights -  Parameter containing:
tensor([[ 0.0048,  0.0144, -0.0043,  ..., -0.0293,  0.0072, -0.0293],
        [ 0.0092, -0.0132, -0.0247,  ...,  0.0029,  0.0146, -0.0019],
        [-0.0156, -0.0094,  0.0008,  ..., -0.0175,  0.0170, -0.0243],
        ...,
        [-0.0343,  0.0353,  0.0152,  ...,  0.0182, -0.0198,  0.0134],
        [ 0.0065,  0.0031, -0.0254,  ..., -0.0215,  0.0192,  0.0209],
        [ 0.0032,  0.0288,  0.0308,  ...,  0.0063,  0.0088,  0.0051]],
```

```
    requires_grad=True)
Epoch 0 - Training loss: 0.9036219311294271
Epoch 1 - Training loss: 0.3439043142012696
Epoch 2 - Training loss: 0.2902518306682105
Epoch 3 - Training loss: 0.25655940484835393
Epoch 4 - Training loss: 0.22571180654423578
Epoch 5 - Training loss: 0.20029108553552932
Epoch 6 - Training loss: 0.1773502943771226
Epoch 7 - Training loss: 0.15937593997891014
Epoch 8 - Training loss: 0.14351457916597313
Epoch 9 - Training loss: 0.1317370911555758

Training Time (in minutes) = 1.2325106382369995
Predicted Digit = 7
Number Of Images Tested = 10000

Model Accuracy = 0.9587

Process finished with exit code 0
```

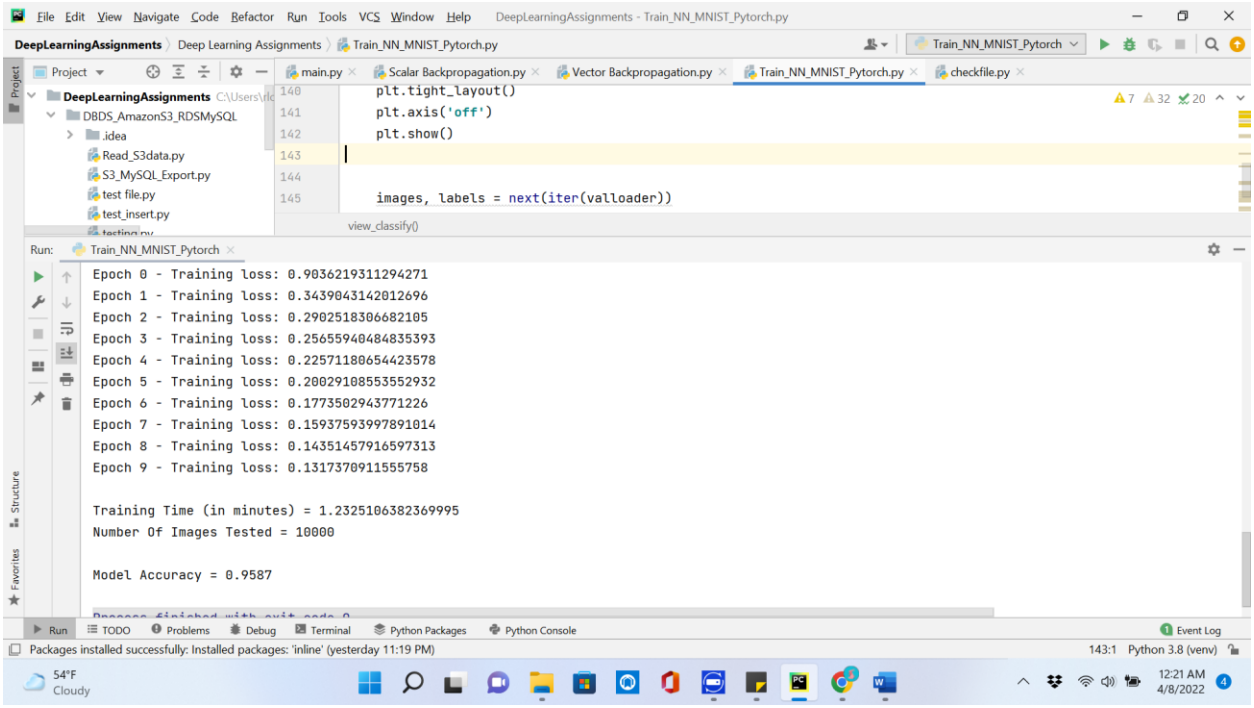
Part of Input Training Data



Predicted Data



Output Screen shot



(b) (Optional) Use only Numpy to train and test this network. You are NOT allowed to use deep earning framework (e.g. Pytorch, Tensorflow etc.) and the corresponding autograd or nn module to train the network.

SOURCE CODE:

```
import numpy as np
import matplotlib.pyplot as plt


def load_data(path):
    def one_hot(y):
        table = np.zeros((y.shape[0], 10))
        for i in range(y.shape[0]):
            table[i][int(y[i][0])] = 1
        return table

    def normalize(x):
        x = x / 255
        return x

    data = np.loadtxt('{}'.format(path), delimiter=',')
    return normalize(data[:, 1:]), one_hot(data[:, :1])


X_train, y_train = load_data('mnist_train.csv')
X_test, y_test = load_data('mnist_test.csv')
```

```
class NeuralNetwork:
    def __init__(self, X, y, batch=128, lr=0.01, epochs=10):
        self.input = X
        self.target = y
        self.batch = batch
        self.epochs = epochs
        self.lr = lr

        self.x = self.input[:self.batch]  # batch input
        self.y = self.target[:self.batch]  # batch target value
        self.loss = []
        self.acc = []

        self.init_weights()

    def init_weights(self):
        self.W1 = np.random.randn(self.input.shape[1], 200)
        self.W2 = np.random.randn(self.W1.shape[1], 50)
        self.W3 = np.random.randn(self.W2.shape[1], self.y.shape[1])

        self.b1 = np.random.randn(self.W1.shape[1], )
        self.b2 = np.random.randn(self.W2.shape[1], )
        self.b3 = np.random.randn(self.W3.shape[1], )

    def ReLU(self, x):
        return np.maximum(0, x)

    def dReLU(self, x):
        return 1 * (x > 0)

    def softmax(self, z):
        z = z - np.max(z, axis=1).reshape(z.shape[0], 1)
        return np.exp(z) / np.sum(np.exp(z), axis=1).reshape(z.shape[0], 1)

    def shuffle(self):
        idx = [i for i in range(self.input.shape[0])]
        np.random.shuffle(idx)
        self.input = self.input[idx]
        self.target = self.target[idx]

    def feedforward(self):
        assert self.x.shape[1] == self.W1.shape[0]
        self.z1 = self.x.dot(self.W1) + self.b1
        self.a1 = self.ReLU(self.z1)

        assert self.a1.shape[1] == self.W2.shape[0]
        self.z2 = self.a1.dot(self.W2) + self.b2
        self.a2 = self.ReLU(self.z2)

        assert self.a2.shape[1] == self.W3.shape[0]
        self.z3 = self.a2.dot(self.W3) + self.b3
        self.a3 = self.softmax(self.z3)
        self.error = self.a3 - self.y

    def backprop(self):
```

```
        dcost = (1 / self.batch) * self.error

        DW3 = np.dot(dcost.T, self.a2).T
        DW2 = np.dot((np.dot((dcost), self.W3.T) * self.dReLU(self.z2)).T,
self.a1).T
        DW1 = np.dot((np.dot(np.dot((dcost), self.W3.T) *
self.dReLU(self.z2), self.W2.T) * self.dReLU(self.z1)).T,
                     self.x).T

        db3 = np.sum(dcost, axis=0)
        db2 = np.sum(np.dot((dcost), self.W3.T) * self.dReLU(self.z2),
axis=0)
        db1 = np.sum((np.dot(np.dot((dcost), self.W3.T) *
self.dReLU(self.z2), self.W2.T) * self.dReLU(self.z1)),
                     axis=0)

        assert DW3.shape == self.W3.shape
        assert DW2.shape == self.W2.shape
        assert DW1.shape == self.W1.shape

        assert db3.shape == self.b3.shape
        assert db2.shape == self.b2.shape
        assert db1.shape == self.b1.shape

        self.W3 = self.W3 - self.lr * DW3
        self.W2 = self.W2 - self.lr * DW2
        self.W1 = self.W1 - self.lr * DW1

        self.b3 = self.b3 - self.lr * db3
        self.b2 = self.b2 - self.lr * db2
        self.b1 = self.b1 - self.lr * db1

    def train(self):
        for epoch in range(self.epochs):
            l = 0
            acc = 0
            self.shuffle()

            for batch in range(self.input.shape[0] // self.batch - 1):
                start = batch * self.batch
                end = (batch + 1) * self.batch
                self.x = self.input[start:end]
                self.y = self.target[start:end]
                self.feedforward()
                self.backprop()
                l += np.mean(self.error ** 2)
                acc += np.count_nonzero(np.argmax(self.a3, axis=1) ==
np.argmax(self.y, axis=1)) / self.batch

            self.loss.append(l / (self.input.shape[0] // self.batch))
            self.acc.append(acc * 100 / (self.input.shape[0] // self.batch))

    def plot(self):
        plt.figure(dpi=125)
        plt.plot(self.loss)
        plt.xlabel("Epochs")
        plt.ylabel("Loss")
```

```
    def acc_plot(self):
        plt.figure(dpi=125)
        plt.plot(self.acc)
        plt.xlabel("Epochs")
        plt.ylabel("Accuracy")

    def test(self, xtest, ytest):
        self.x = xtest
        self.y = ytest
        self.feedforward()
        acc = np.count_nonzero(np.argmax(self.a3, axis=1) ==
np.argmax(self.y, axis=1)) / self.x.shape[0]
        print("Accuracy:", 100 * acc, "%")


NN = NeuralNetwork(X_train, y_train)
NN.train()
NN.plot()
NN.test(X_test, y_test)
```

OUTPUT: