# Computer Vision 2023 Assignment 3: Deep Learning for Perception Tasks

This assignment contains 2 questions. The first question probes understanding of deep learning for classification. The second question is a more challenging classification experiment on a larger dataset. Answer the questions in separate Python notebooks.

## Question 1: A simple classifier, 20 marks

For this exercise, we provide demo code showing how to train a network on a small dataset called Fashion-MNIST. Please run through the code "tutorial-style" to get a sense of what it is doing. Then use the code alongside lecture notes and other resources to understand how to use pytorch libraries to implement, train and use a neural network.

For the Fashion-MNIST dataset the lables from 0-9 correspond to various clothing classes so you might find it convenient to create a python list as follows:

class_names = ['T-shirt/top', 'Trouser', 'Pullover', 'Dress', 'Coat', 'Sandal', 'Shirt', 'Sneaker', 'Bag', 'Ankle boot']

You will need to answer various questions about the system, how it operates, the results of experiments with it and make modifications to it yourself. You can change the training scheme and the network structure.

Organize your own text and code cell to show the answer of each questions.

Detailed requirements:

**Q1.1 (1 point)**

Extract 3 images of different types of clothing from the training dataset, print out the size/shape of the training images, and display the three with their corresponding labels.

**Q1.2 (2 points)**

Run the training code for 10 epochs, for different values of the learning rate. Fill in the table below and plot the loss curves for each experiment:

| Lr | Accuracy |
|---|---|
| 1 | 10% |
| 0.1 | 87.3 |
| 0.01 | 83.9% |
| 0.001 | 70.5% |

**Q1.3 (3 points)**

Report the number of epochs when the accuracy reaches 85%. Fill in the table below and plot the loass curve for each experiment:

| Lr | Accuracy | Epoch |
|---|---|---|
| 1 | nan | nan |
| 0.1 | 85.4% | 4 |
| 0.01 | 85.2% | 16 |
| 0.001 | 82% | 41 |

**Q1.4 (2 points)**

Based on the provided notebook content, I couldn't locate "table 1" or "table 2" explicitly. The understanding of learning rate typically comes from observing how the model's performance varies with different learning rates. A high learning rate might make the model converge faster but can overshoot the optimal solution, leading to oscillation or divergence. A low learning rate might converge to a better solution but can take longer and might get stuck in local minima. Without the tables, it's challenging to make a definitive observation.

We don't have the explicit tables to make direct observations. However, from the general knowledge of learning rates: the right learning rate is crucial for training a model. If it's too high, the model might not converge, and if it's too low, the model might take too long to converge or get stuck in local optima.

**Q1.5 (5 points)**

The base model structure you provided consists of two hidden layers, each with 512 perceptrons. For the wider network, you could double the perceptrons in each hidden layer (e.g., 1024). For the deeper network, you could add more hidden layers (e.g., two additional layers with 512 perceptrons each). The effect of making a network wider or deeper can vary. A wider network can capture more features, but it also has more parameters, which can lead to overfitting. A deeper network can capture more complex representations but might also be harder to train due to vanishing or exploding gradients.

| Structures | Accuracy |
|---|---|
| Base | 86.1% |
| Deeper | 84.1% |
| Wider | 84.6% |

**Q1.6 (2 points)**

the magnitude of gradients to decrease as training progresses, especially when using techniques like gradient clipping or normalization. This is because as the model approaches a minimum in the loss surface, the gradient (or slope) tends to decrease.

it's typical for the mean of gradients to decrease as training progresses, especially if the model is converging. This phenomenon can be attributed to the model getting closer to a local minimum where the slope of the loss curve is smaller.

**Q1.7 (5 points)**

By transitioning to a CNN for image data, we would expect:

Convergence: Faster convergence compared to the MLP, especially if the architecture is designed optimally for the given dataset. Accuracy: Potentially higher accuracy. The MLP in the notebook achieved around 85.2% accuracy after 17 epochs. A well-designed CNN might surpass this accuracy in fewer epochs because it can exploit the spatial structure of images. Number of Parameters: CNNs can achieve good performance with fewer parameters compared to MLPs for image data. Convolutional layers share weights across spatial locations, which reduces the number of parameters. This, combined with pooling layers, allows CNNs to be more parameter-efficient than MLPs for image tasks. In conclusion, while MLPs are versatile and can work for a range of tasks, CNNs are specialized for image data and typically outperform MLPs in terms of convergence speed, accuracy, and parameter efficiency for image classification tasks.

```
import numpy as np # This is for mathematical operations

# this is used in plotting
import matplotlib.pyplot as plt
import time
import pylab as pl
from IPython import display

%matplotlib inline

%load_ext autoreload
%autoreload 2
%reload_ext autoreload
```

```
%reload_ext autoreload


#### Tutorial Code
####PyTorch has two primitives to work with data: torch.utils.data.DataLoader and torch.utils.data.Dataset.
#####Dataset stores samples and their corresponding labels, and DataLoader wraps an iterable around the Dataset.
import torch
from torch import nn
from torch.utils.data import DataLoader
from torchvision import datasets
from torchvision.transforms import ToTensor, Lambda, Compose
import matplotlib.pyplot as plt

# Download training data from open datasets.
##Every TorchVision Dataset includes two arguments:
##transform and target_transform to modify the samples and labels respectively.

training_data = datasets.FashionMNIST(
    root="data",
    train=True,
    download=True,
    transform=ToTensor(),
)

# Download test data from open datasets.
test_data = datasets.FashionMNIST(
    root="data",
    train=False,
    download=True,
    transform=ToTensor(),
)
```

```
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/train-images-idx3-ubyte.gz
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/train-images-idx3-ubyte.gz to data/FashionMNIST/raw/train-images-idx3-ubyte.gz
100%|██████████| 26421880/26421880 [00:01<00:00, 18118936.84it/s]
Extracting data/FashionMNIST/raw/train-images-idx3-ubyte.gz to data/FashionMNIST/raw

Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/train-labels-idx1-ubyte.gz
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/train-labels-idx1-ubyte.gz to data/FashionMNIST/raw/train-labels-idx1-ubyte.gz
100%|██████████| 29515/29515 [00:00<00:00, 313474.60it/s]
Extracting data/FashionMNIST/raw/train-labels-idx1-ubyte.gz to data/FashionMNIST/raw

Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/t10k-images-idx3-ubyte.gz
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/t10k-images-idx3-ubyte.gz to data/FashionMNIST/raw/t10k-images-idx3-ubyte.gz
100%|██████████| 4422102/4422102 [00:00<00:00, 6106198.86it/s]
Extracting data/FashionMNIST/raw/t10k-images-idx3-ubyte.gz to data/FashionMNIST/raw

Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/t10k-labels-idx1-ubyte.gz
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/t10k-labels-idx1-ubyte.gz to data/FashionMNIST/raw/t10k-labels-idx1-ubyte.gz
100%|██████████| 5148/5148 [00:00<00:00, 15067883.46it/s]
Extracting data/FashionMNIST/raw/t10k-labels-idx1-ubyte.gz to data/FashionMNIST/raw
```

**NOTE**: For consistency with the original data set, we call our validation data "test_data". It is important to keep in mind though that we are using the data for model validation and not for testing the final, trained model (which requires data not used when training the model parameters).

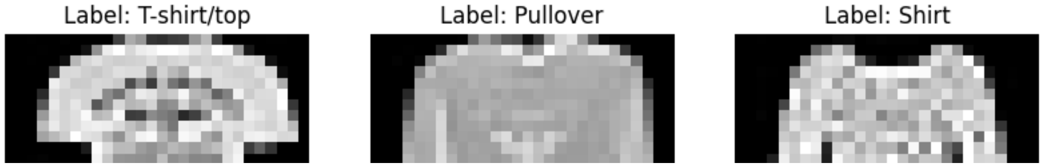
We pass the Dataset as an argument to DataLoader. This wraps an iterable over our dataset and supports automatic batching, sampling, shuffling, and multiprocess data loading. Here we define a batch size of 64, i.e. each element in the dataloader iterable will return a batch of 64 features and labels.

```
batch_size = 64

# Create data loaders.
train_dataloader = DataLoader(training_data, batch_size=batch_size)
test_dataloader = DataLoader(test_data, batch_size=batch_size)

for X, y in test_dataloader:
    print("Shape of X [N, C, H, W]: ", X.shape)
    print("Shape of y: ", y.shape, y.dtype)
    break
```

```
Shape of X [N, C, H, W]:  torch.Size([64, 1, 28, 28])
Shape of y:  torch.Size([64]) torch.int64
```

Add in a code cell to inspect the training data, as per Q1.1. Each element of the training_data structure has a greyscale image (which you can use plt.imshow(img[0,:,:]) to display, just like you did in previous assignments.

```
# Code cell for training image display
# For reproducibility, we set a seed
np.random.seed(0)

# Define class names
class_names = ['T-shirt/top', 'Trouser', 'Pullover', 'Dress', 'Coat',
               'Sandal', 'Shirt', 'Sneaker', 'Bag', 'Ankle boot']

# Select three random indices from the training set
indices = np.random.choice(np.arange(len(training_data)), size=3, replace=False)

# Extract the images and labels at these indices
images, labels = zip(*[(training_data[i][0], training_data[i][1]) for i in indices])

# Display the images along with their labels
plt.figure(figsize=(10, 10))
for i, (img, label) in enumerate(zip(images, labels)):
    plt.subplot(1, 3, i+1)
    plt.imshow(img.squeeze(), cmap='gray')
    plt.title(f'Label: {class_names[label]}')
    plt.axis('off')
plt.show()

# Print out the size/shape of the training images
print(f'Size/shape of the training images: {images[0].shape, images[1].shape, images[2].shape}')
```




To define a neural network in PyTorch, we create a class that inherits from nn.Module. We define the layers of the network in the init function and specify how data will pass through the network in the forward function. To accelerate operations in the neural network, we move it to the GPU if available.

```
# Get cpu or gpu device for training.
device = "cuda" if torch.cuda.is_available() else "cpu"
print("Using {} device".format(device))

import torch.nn.functional as F

# Define model
class NeuralNetwork(nn.Module):
    def __init__(self):
        super(NeuralNetwork, self).__init__()
        self.flatten = nn.Flatten()
        self.linear_relu_stack = nn.Sequential(
            nn.Linear(28*28, 512),
            nn.ReLU(),
            nn.Linear(512, 512),
            nn.ReLU(),
            nn.Linear(512, 10)
        )

    def forward(self, x):
        x = self.flatten(x)
        logits = self.linear_relu_stack(x)

        return logits

model = NeuralNetwork().to(device)
print(model)
###Define the loss function and the optimizer
loss_fn = nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(), lr=1e-3)
```

```
Using cpu device
NeuralNetwork(
  (flatten): Flatten(start_dim=1, end_dim=-1)
  (linear_relu_stack): Sequential(
    (0): Linear(in_features=784, out_features=512, bias=True)
    (1): ReLU()
    (2): Linear(in_features=512, out_features=512, bias=True)
    (3): ReLU()
    (4): Linear(in_features=512, out_features=10, bias=True)
  )
)
```

In a single training loop, the model makes predictions on the training dataset (fed to it in batches), and backpropagates the prediction error to adjust the model's parameters.

```
def train(dataloader, model, loss_fn, optimizer):
    size = len(dataloader.dataset)
    model.train()
    for batch, (X, y) in enumerate(dataloader):
        X, y = X.to(device), y.to(device)

        # Compute prediction error
        pred = model(X)
        loss = loss_fn(pred, y)

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

        if batch % 100 == 0:
            loss, current = loss.item(), batch * len(X)
            print(f"loss: {loss:>7f}  [{current:>5d}/{size:>5d}]")



##Define a test function
def test(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
    print(f"Test Error: \n Accuracy: {(100*correct):>0.1f}%, Avg loss: {test_loss:>8f} \n")
    accuracy = 100 * correct
    return accuracy


#Train and test the model
epochs = 10
for t in range(epochs):
    print(f"Epoch {t+1}\n-------------------------------")
    train(train_dataloader, model, loss_fn, optimizer)
    test(test_dataloader, model, loss_fn)
print("Done!")
```

```
loss: 0.927950  [32000/60000]
loss: 0.983535  [38400/60000]
loss: 0.921739  [44800/60000]
loss: 0.950817  [51200/60000]
loss: 0.892684  [57600/60000]
Test Error: 
 Accuracy: 66.6%, Avg loss: 0.905666 

Epoch 8
-------------------------------
loss: 0.949293  [    0/60000]
loss: 0.984045  [ 6400/60000]
loss: 0.783582  [12800/60000]
loss: 0.945712  [19200/60000]
loss: 0.835991  [25600/60000]
loss: 0.859373  [32000/60000]
loss: 0.930438  [38400/60000]
loss: 0.872411  [44800/60000]
loss: 0.894415  [51200/60000]
loss: 0.843675  [57600/60000]
Test Error: 
 Accuracy: 67.7%, Avg loss: 0.853958 

Epoch 9
-------------------------------
loss: 0.882755  [    0/60000]
loss: 0.933588  [ 6400/60000]
loss: 0.723064  [12800/60000]
loss: 0.897423  [19200/60000]
loss: 0.795548  [25600/60000]
loss: 0.808405  [32000/60000]
loss: 0.890632  [38400/60000]
loss: 0.838168  [44800/60000]
loss: 0.852271  [51200/60000]
loss: 0.806557  [57600/60000]
Test Error: 
 Accuracy: 68.8%, Avg loss: 0.815127 

Epoch 10
-------------------------------
loss: 0.829898  [    0/60000]
loss: 0.893820  [ 6400/60000]
loss: 0.676290  [12800/60000]
loss: 0.860487  [19200/60000]
loss: 0.765712  [25600/60000]
loss: 0.769456  [32000/60000]
loss: 0.858830  [38400/60000]
loss: 0.813053  [44800/60000]
loss: 0.819268  [51200/60000]
loss: 0.777071  [57600/60000]
Test Error: 
 Accuracy: 70.5%, Avg loss: 0.784408 

Done!
```

```
###Define the loss function and the optimizer
loss_fn = nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(), lr=1e-2)
#Train and test the model
epochs = 10
```

```
for t in range(epochs):
    print(f"Epoch {t+1}\n-------------------------------")
    train(train_dataloader, model, loss_fn, optimizer)
    test(test_dataloader, model, loss_fn)
print("Done!")
```

```
 Accuracy: 83.9%, Avg loss: 0.452731

Done!
```

```
###Define the loss function and the optimizer
loss_fn = nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(), lr=1e-1)
#Train and test the model
epochs = 10
for t in range(epochs):
   print(f"Epoch {t+1}\n-------------------------------")
   train(train_dataloader, model, loss_fn, optimizer)
   test(test_dataloader, model, loss_fn)
print("Done!")
```

```
loss: 0.209742  [ 6400/60000]
loss: 0.149156  [12800/60000]
loss: 0.228591  [19200/60000]
loss: 0.280156  [25600/60000]
loss: 0.307512  [32000/60000]
loss: 0.212144  [38400/60000]
loss: 0.267253  [44800/60000]
loss: 0.285549  [51200/60000]
loss: 0.319775  [57600/60000]
Test Error:
 Accuracy: 87.3%, Avg loss: 0.347752

Done!
```

```
# Continuous training (with learning rate of 1e-2 and reset epochs) until 85% accuracy is reached. Uncomment with caution.

###Define the loss function and the optimizer
loss_fn = nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(), lr=1)
#Train and test the model
epochs = 10
for t in range(epochs):
   print(f"Epoch {t+1}\n-------------------------------")
   train(train_dataloader, model, loss_fn, optimizer)
   test(test_dataloader, model, loss_fn)
print("Done!")
```

## Not error! stopped due to long run

```
model = NeuralNetwork().to(device)
accuracy = 0
epochs = 0
loss_fn = nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(), lr=1e-3)
print(model)
while accuracy < 85:
    epochs += 1
    print(f"Epoch {epochs}\n-------------------------------")
    train(train_dataloader, model, loss_fn, optimizer)
    accuracy = test(test_dataloader, model, loss_fn)
print("Reached 85%!")
```

```
Epoch 39
-------------------------------
loss: 0.405476  [    0/60000]
loss: 0.540403  [ 6400/60000]
loss: 0.342822  [12800/60000]
loss: 0.565030  [19200/60000]
loss: 0.522677  [25600/60000]
loss: 0.501867  [32000/60000]
loss: 0.512337  [38400/60000]
loss: 0.664316  [44800/60000]
loss: 0.620161  [51200/60000]
loss: 0.474522  [57600/60000]
Test Error:
 Accuracy: 82.0%, Avg loss: 0.514369

Epoch 40
-------------------------------
loss: 0.400469  [    0/60000]
loss: 0.536707  [ 6400/60000]
loss: 0.339347  [12800/60000]
loss: 0.560589  [19200/60000]
loss: 0.517832  [25600/60000]
loss: 0.498239  [32000/60000]
loss: 0.508635  [38400/60000]
loss: 0.664397  [44800/60000]
loss: 0.618256  [51200/60000]
loss: 0.469450  [57600/60000]
Test Error:
 Accuracy: 82.1%, Avg loss: 0.511503

Epoch 41
-------------------------------
loss: 0.395651  [    0/60000]
loss: 0.533257  [ 6400/60000]
loss: 0.336032  [12800/60000]
loss: 0.556328  [19200/60000]
loss: 0.513070  [25600/60000]
loss: 0.494690  [32000/60000]
loss: 0.505145  [38400/60000]
loss: 0.664337  [44800/60000]
loss: 0.616382  [51200/60000]
loss: 0.464638  [57600/60000]
Test Error:
 Accuracy: 82.2%, Avg loss: 0.508782

Epoch 42
-------------------------------
loss: 0.390928  [    0/60000]
loss: 0.529994  [ 6400/60000]
loss: 0.332862  [12800/60000]
---------------------------------------------------------------------------
KeyboardInterrupt                         Traceback (most recent call last)
<ipython-input-33-3ef4d1ea8fbc> in <cell line: 7>()
      8     epochs += 1
      9     print(f"Epoch {epochs}\n-------------------------------")
---> 10     train(train_dataloader, model, loss_fn, optimizer)
     11     accuracy = test(test_dataloader, model, loss_fn)
```

```
# Base Neural Network model definition and training/testing.
model = NeuralNetwork().to(device)
print(model)
# Another variant of continuous training until 85% accuracy is reached with a learning rate of 1. Uncomment with caution.

###Define the loss function and the optimizer
loss_fn = nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(), lr=1e-2)
#Train and test the model
accuracy = 0
epochs = 0
while accuracy < 85:
   print(f"Epoch {epochs+1}\n-------------------------------")
   train(train_dataloader, model, loss_fn, optimizer)
   accuracy = test(test_dataloader, model, loss_fn)
   epochs+=1
print("Reached 85%!")
```

```
Test Error:
 Accuracy: 84.9%, Avg loss: 0.423722

Epoch 16
-------------------------------
loss: 0.240957  [    0/60000]
loss: 0.379679  [ 6400/60000]
loss: 0.253850  [12800/60000]
loss: 0.425076  [19200/60000]
loss: 0.321392  [25600/60000]
loss: 0.389235  [32000/60000]
loss: 0.361777  [38400/60000]
loss: 0.502944  [44800/60000]
loss: 0.488382  [51200/60000]
loss: 0.385316  [57600/60000]
Test Error:
 Accuracy: 84.9%, Avg loss: 0.417985

Epoch 17
-------------------------------
loss: 0.235521  [    0/60000]
loss: 0.372527  [ 6400/60000]
loss: 0.251393  [12800/60000]
loss: 0.417645  [19200/60000]
loss: 0.315015  [25600/60000]
loss: 0.382263  [32000/60000]
loss: 0.355632  [38400/60000]
loss: 0.495066  [44800/60000]
loss: 0.480331  [51200/60000]
loss: 0.383706  [57600/60000]
Test Error:
 Accuracy: 85.2%, Avg loss: 0.411988

Reached 85%!
```

```
# Base Neural Network model definition and training/testing.
model = NeuralNetwork().to(device)
print(model)
###Define the loss function and the optimizer
loss_fn = nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(), lr=1e-1)
#Train and test the model
epochs = 0
accuracy=0
while accuracy < 85:
    print(f"Epoch {epochs+1}\n-------------------------------")
    train(train_dataloader, model, loss_fn, optimizer)
    accuracy = test(test_dataloader, model, loss_fn)
    epochs+=1
print("Reached 85%!")
```

```
    (3): ReLU()
    (4): Linear(in_features=512, out_features=10, bias=True)
```

```
loss: 0.693625  [19200/60000]
loss: 0.598871  [25600/60000]
loss: 0.499419  [32000/60000]
loss: 0.544398  [38400/60000]
loss: 0.601305  [44800/60000]
loss: 0.609945  [51200/60000]
loss: 0.467853  [57600/60000]
Test Error: 
 Accuracy: 79.2%, Avg loss: 0.549600 

Epoch 2
-------------------------------
loss: 0.434148  [    0/60000]
loss: 0.438480  [ 6400/60000]
loss: 0.376246  [12800/60000]
loss: 0.437265  [19200/60000]
loss: 0.411576  [25600/60000]
loss: 0.443233  [32000/60000]
loss: 0.414029  [38400/60000]
loss: 0.508594  [44800/60000]
loss: 0.497658  [51200/60000]
loss: 0.434544  [57600/60000]
Test Error: 
 Accuracy: 83.2%, Avg loss: 0.453243 

Epoch 3
-------------------------------
loss: 0.302706  [    0/60000]
loss: 0.358633  [ 6400/60000]
loss: 0.326532  [12800/60000]
loss: 0.370630  [19200/60000]
loss: 0.360748  [25600/60000]
loss: 0.418142  [32000/60000]
loss: 0.347265  [38400/60000]
loss: 0.453526  [44800/60000]
loss: 0.452856  [51200/60000]
loss: 0.411705  [57600/60000]
Test Error: 
 Accuracy: 84.4%, Avg loss: 0.421282 

Epoch 4
-------------------------------
loss: 0.253715  [    0/60000]
loss: 0.327413  [ 6400/60000]
loss: 0.282704  [12800/60000]
loss: 0.321405  [19200/60000]
loss: 0.336284  [25600/60000]
loss: 0.388981  [32000/60000]
loss: 0.309775  [38400/60000]
```

```
# Wider Neural Network model definition and training/testing.
model = NeuralNetwork().to(device)
print(model)
###Define the loss function and the optimizer
loss_fn = nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(), lr=1)
#Train and test the model
epochs = 0
accuracy=0
while accuracy < 85:
```

```
while accuracy < 85:
    print(f"Epoch {epochs+1}\n-------------------------------")
    train(train_dataloader, model, loss_fn, optimizer)
    accuracy = test(test_dataloader, model, loss_fn)
    epochs+=1
print("Reached 85%!")
```

Double-click (or enter) to edit

```
# Get cpu or gpu device for training.
device = "cuda" if torch.cuda.is_available() else "cpu"
print("Using {} device".format(device))

import torch.nn.functional as F

# Define model
class NeuralNetwork(nn.Module):
    def __init__(self):
        super(NeuralNetwork, self).__init__()
        self.flatten = nn.Flatten()
        self.linear_relu_stack = nn.Sequential(
            nn.Linear(28*28, 512),
            nn.ReLU(),
            nn.Linear(512, 512),
            nn.ReLU(),
            nn.Linear(512, 10)
        )

    def forward(self, x):
        x = self.flatten(x)
        logits = self.linear_relu_stack(x)

        return logits

model = NeuralNetwork().to(device)
print(model)

###Define the loss function and the optimizer
loss_fn = nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(), lr=1e-1)
#Train and test the model
epochs = 5
for t in range(epochs):
    print(f"Epoch {t+1}\n-------------------------------")
    train(train_dataloader, model, loss_fn, optimizer)
    test(test_dataloader, model, loss_fn)
print("Done!")
```

```
Using cpu device
NeuralNetwork(
  (flatten): Flatten(start_dim=1, end_dim=-1)
  (linear_relu_stack): Sequential(
    (0): Linear(in_features=784, out_features=512, bias=True)
    (1): ReLU()
    (2): Linear(in_features=512, out_features=512, bias=True)
    (3): ReLU()
```

```
        (4): Linear(in_features=512, out_features=10, bias=True)
      )
    )
    Epoch 1
    -------------------------------
    loss: 2.309982  [    0/60000]
    loss: 0.902847  [ 6400/60000]
    loss: 0.593169  [12800/60000]
    loss: 0.700042  [19200/60000]
    loss: 0.604879  [25600/60000]
    loss: 0.508695  [32000/60000]
    loss: 0.535796  [38400/60000]
    loss: 0.597670  [44800/60000]
    loss: 0.583273  [51200/60000]
    loss: 0.449860  [57600/60000]
    Test Error:
     Accuracy: 79.1%, Avg loss: 0.554113

    Epoch 2
    -------------------------------
    loss: 0.435844  [    0/60000]
    loss: 0.451145  [ 6400/60000]
    loss: 0.382021  [12800/60000]
    loss: 0.433599  [19200/60000]
    loss: 0.417255  [25600/60000]
    loss: 0.457310  [32000/60000]
    loss: 0.410194  [38400/60000]
    loss: 0.503276  [44800/60000]
    loss: 0.506389  [51200/60000]
    loss: 0.429007  [57600/60000]
    Test Error:
     Accuracy: 82.1%, Avg loss: 0.472644

    Epoch 3
    -------------------------------
    loss: 0.335525  [    0/60000]
    loss: 0.366189  [ 6400/60000]
    loss: 0.319186  [12800/60000]
    loss: 0.358969  [19200/60000]
    loss: 0.357107  [25600/60000]
    loss: 0.426372  [32000/60000]
    loss: 0.352470  [38400/60000]
    loss: 0.450525  [44800/60000]
    loss: 0.453973  [51200/60000]
    loss: 0.406734  [57600/60000]
    Test Error:
     Accuracy: 84.1%, Avg loss: 0.426516

    Epoch 4
```

```
# Define model
class WiderNetwork(nn.Module):
    def __init__(self):
        super(WiderNetwork, self).__init__()
        self.flatten = nn.Flatten()
        self.linear_relu_stack = nn.Sequential(
            nn.Linear(28*28, 1024),
            nn.ReLU(),
            nn.Linear(1024, 1024),
            nn.ReLU(),
```