Optimal Selection of the hyper-parameters associated with the classification on MNIST

Choose an optimal set of hyper-parameters and design a neural network for the classification of MNIST dataset

```
import os

# load data
from torch.utils.data import DataLoader
from torchvision import datasets, transforms

# train
import torch
from torch import nn, optim
from torch.nn import functional as F
from torch.optim import lr_scheduler
import numpy as np

# visualization
import matplotlib.pyplot as plt
import pandas as pd
```

check device

```
device = torch.device('cuda:0' if torch.cuda.is_available() else 'cpu')
print('Device: {}'.format(device))
```

1. Data

- you can use any data normalisation method
- one example of the data normalisation is whitenning as given by:

```
transform_train = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize(mean=(0.5,), std=(0.5,))
])

transform_test = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize(mean=(0.5,), std=(0.5,))
])

```

- load the MNIST dataset
- use the original training dataset for testing your model
- use the original testing dataset for training your model

```
data_path = './MNIST'
```

```
data_test   = datasets.MNIST(root = data_path, train= True, download=True, transform= transform_test)
data_train  = datasets.MNIST(root = data_path, train= False, download=True, transform= transform_train)
```

- Note that the number of your training data must be 10,000
- Note that the number of your testing data must be 60,000

```
print("the number of your training data (must be 10,000) = ", data_train.__len__())
print("hte number of your testing data (must be 60,000) = ", data_test.__len__())
```

2. Model

- design a neural network architecture with three layers (input layer, one hidden layer and output layer)
- the input dimension of the input layer should be 784 (28 * 28)
- the output dimension of the output layer should be 10 (class of digits)
- all the layers should be fully connected layers
- use any type of activation functions

```
class classification(nn.Module):
    def __init__(self):
        super(classification, self).__init__()

        # construct layers for a neural network
        self.classifier1 = nn.Sequential(
            nn.Linear(in_features=28*28, out_features=512),
            nn.ReLU(inplace = True),
            nn.BatchNorm1d(512),
            nn.Dropout(0.3),
        )
        self.classifier2 = nn.Sequential(
            nn.Linear(in_features=512, out_features=512),
            nn.ReLU(inplace = True),
            nn.BatchNorm1d(512),
            nn.Dropout(0.3),
        )
        self.classifier3 = nn.Sequential(
            nn.Linear(in_features=512, out_features=10),
            nn.ReLU(inplace = True),
        )


    def forward(self, inputs):                 # [batchSize, 1, 28, 28]
        x = inputs.view(inputs.size(0), -1)    # [batchSize, 28*28]
        x = self.classifier1(x)                # [batchSize, 20*20]
        x = self.classifier2(x)                # [batchSize, 10*10]
        out = self.classifier3(x)              # [batchSize, 10]

        return out
```

3. Loss function

- use any type of loss function

- design the output of the output layer considering your loss function

```
criterion = nn.CrossEntropyLoss()
```

4. Optimization

- use any stochastic gradient descent algorithm for the optimization
- use any size of the mini-batch
- use any optimization algorithm (for example, Momentum, AdaGrad, RMSProp, Adam)
- use any regularization algorithm (for example, Dropout, Weight Decay)
- use any annealing scheme for the learning rate (for example, constant, decay, staircase)

```
BATCH_SIZE = 100
```

```
train_loader = torch.utils.data.DataLoader(data_train, batch_size=BATCH_SIZE, shuffle=True)
test_loader = torch.utils.data.DataLoader(data_test, batch_size=BATCH_SIZE, shuffle=False)
```

```
model = classification()
model.to(device)
```

```
epochs = 15
lr = 0.01
step_size = 5

optimizer = optim.Adam(model.parameters(), lr=lr)
exp_lr_scheduler = lr_scheduler.StepLR(optimizer, step_size=step_size, gamma=0.1)
```

5. Training

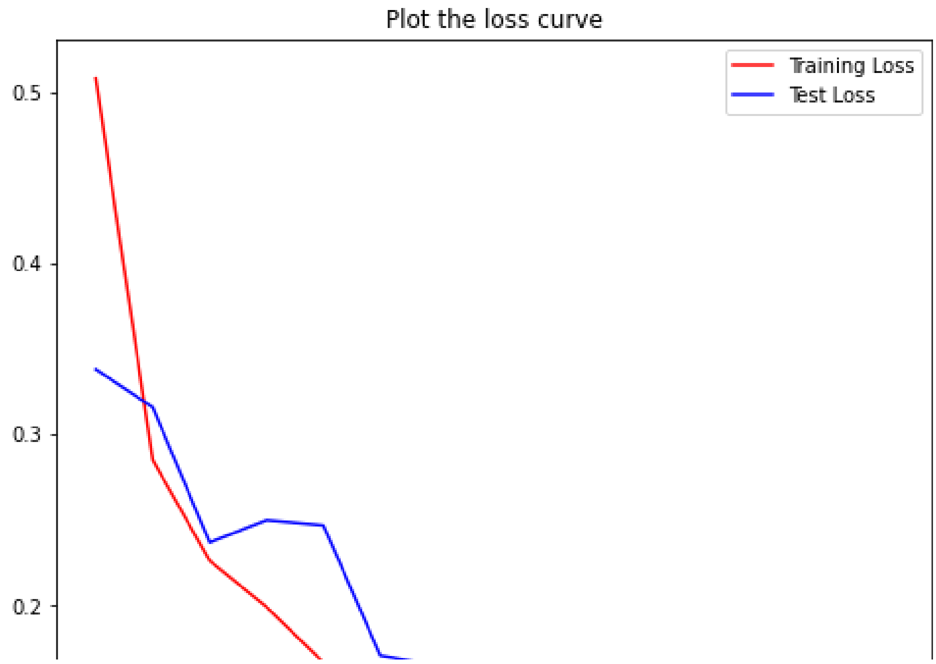
```
test_loss_min = np.Inf
train_losses = []
test_losses = []
history_accuracy = []
history_running_acc = []

for e in range(1, epochs+1):
    running_loss = 0
    running_acc = 0

    for images, labels in train_loader:
        model.train()
        images, labels = images.to(device), labels.to(device)

        optimizer.zero_grad()
        ps = model(images)
        _, top_class = ps.topk(1, dim=1)
        equals = top_class == labels.view(*top_class.shape)

        loss = criterion(ps, labels)
        running_acc += torch.mean(equals.type(torch.FloatTensor))
```

```
        loss.backward()
        optimizer.step()

        running_loss += loss.item()

    else:
        test_loss = 0
        accuracy = 0

        with torch.no_grad():
            model.eval()
            for images, labels in test_loader:
                images, labels = images.to(device), labels.to(device)

                ps = model(images)
                _, top_class = ps.topk(1, dim=1)
                equals = top_class == labels.view(*top_class.shape)

                test_loss += criterion(ps, labels).item()
                accuracy += torch.mean(equals.type(torch.FloatTensor))

        train_losses.append(running_loss/len(train_loader))
        test_losses.append(test_loss/len(test_loader))
        history_accuracy.append(accuracy/len(test_loader))
        history_running_acc.append(running_acc/len(train_loader))

        exp_lr_scheduler.step()


    print(f"Epoch: {e}/{epochs}.. ",
          f"Training Loss: {running_loss/len(train_loader):.3f}.. ",
          f"Testing Loss: {test_loss/len(test_loader):.3f}  / ",
          f"Train Accuracy: {running_acc/len(train_loader):.3f}  ",
          f"Test Accuracy: {accuracy/len(test_loader):.3f}")
```

6. Visualization

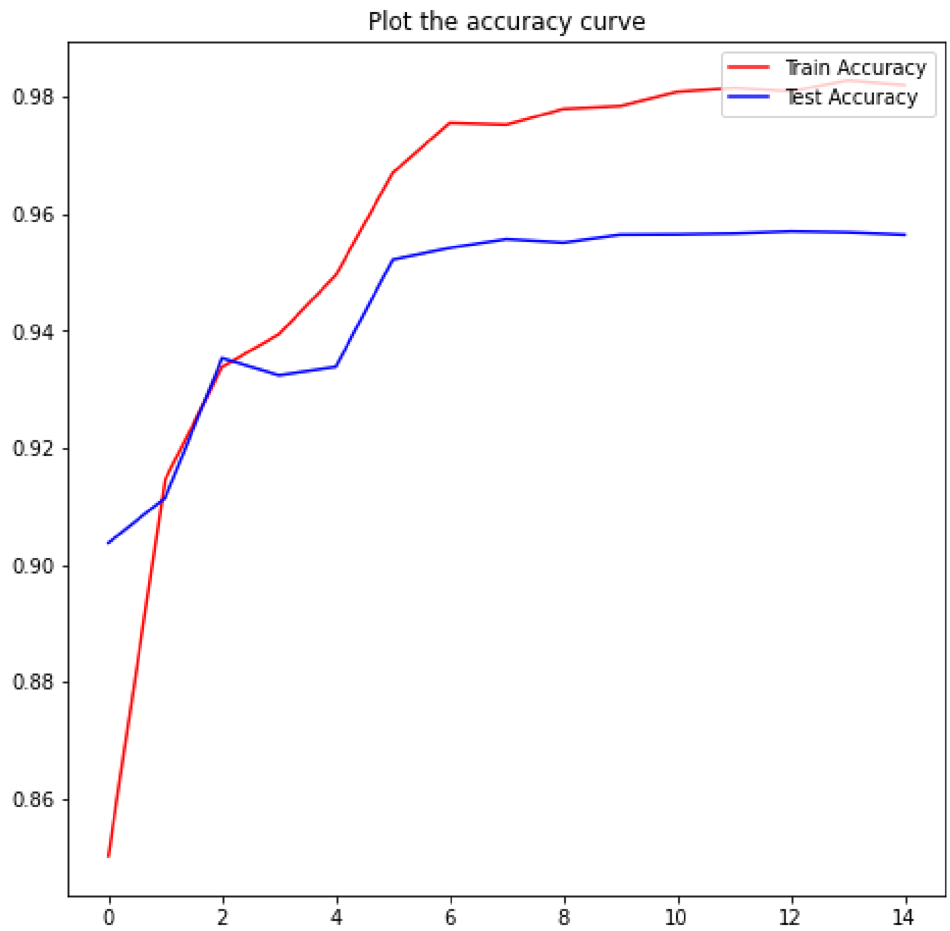
1. Plot the training and testing losses over epochs [2pt]

```
fig_1 = plt.figure(figsize=(8,8))
plt.plot(np.array(range(epochs)), train_losses, c = 'r', label = 'Training Loss')
plt.plot(np.array(range(epochs)), test_losses, c = 'b', label = 'Test Loss')
plt.legend(loc = 'upper right')
plt.title('Plot the loss curve')
plt.show()
fig_1.savefig('loss curve.png')
```



2. Plot the training and testing accuracies over epochs [2pt]


```
fig_2 = plt.figure(figsize=(8,8))
plt.plot(np.array(range(epochs)), history_running_acc, c = 'r', label = 'Train Accuracy')
plt.plot(np.array(range(epochs)), history_accuracy, c = 'b', label = 'Test Accuracy')
plt.legend(loc = 'upper right')
plt.title('Plot the accuracy curve')
plt.show()
fig_2.savefig('accuracy curve.png')
```



3. Print the final training and testing losses at convergence [2pt]

```
result_loss = pd.DataFrame({'loss':[train_losses[-1], test_losses[-1]]}, index = ['training loss','testing
result_loss
```

	loss
training loss	0.054692
testing loss	0.158594

4. Print the final training and testing accuracies at convergence [20pt]

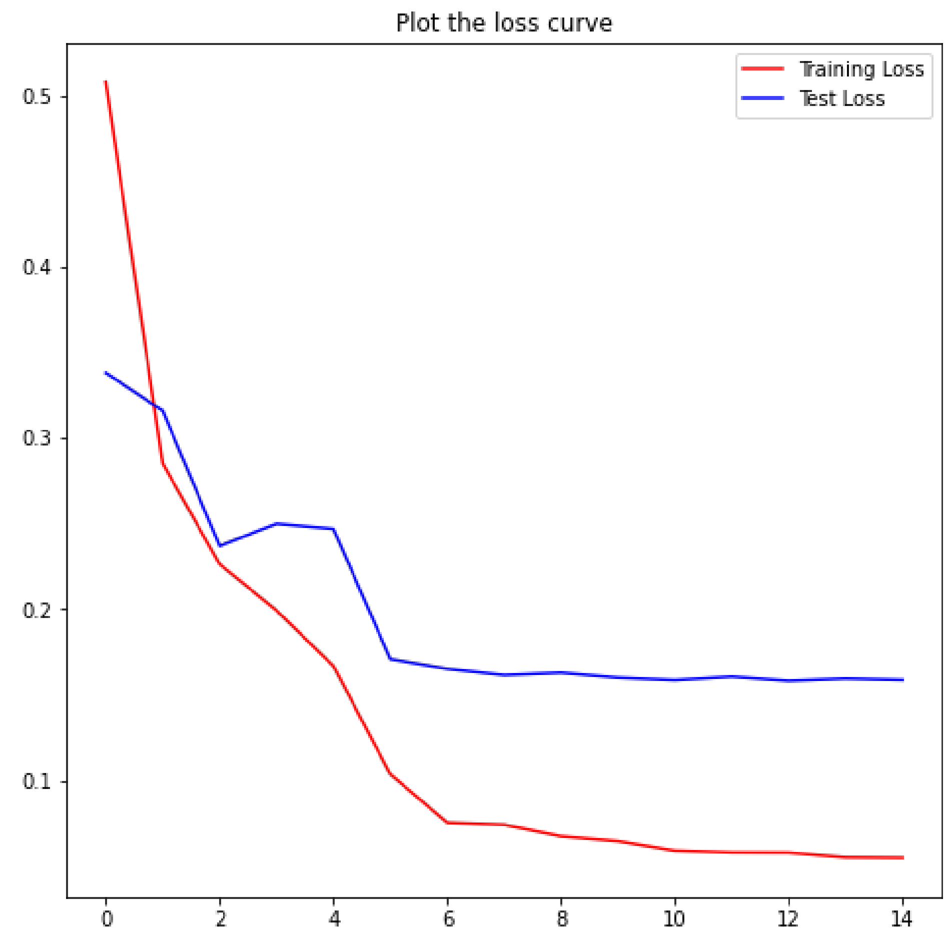
```
result_acc = pd.DataFrame({'accuracy':[history_running_acc[-1].item(), history_accuracy[-1].item()]}, index
result_acc
```

	accuracy
training accuracy	0.982000
testing accuracy	0.956401

Submission

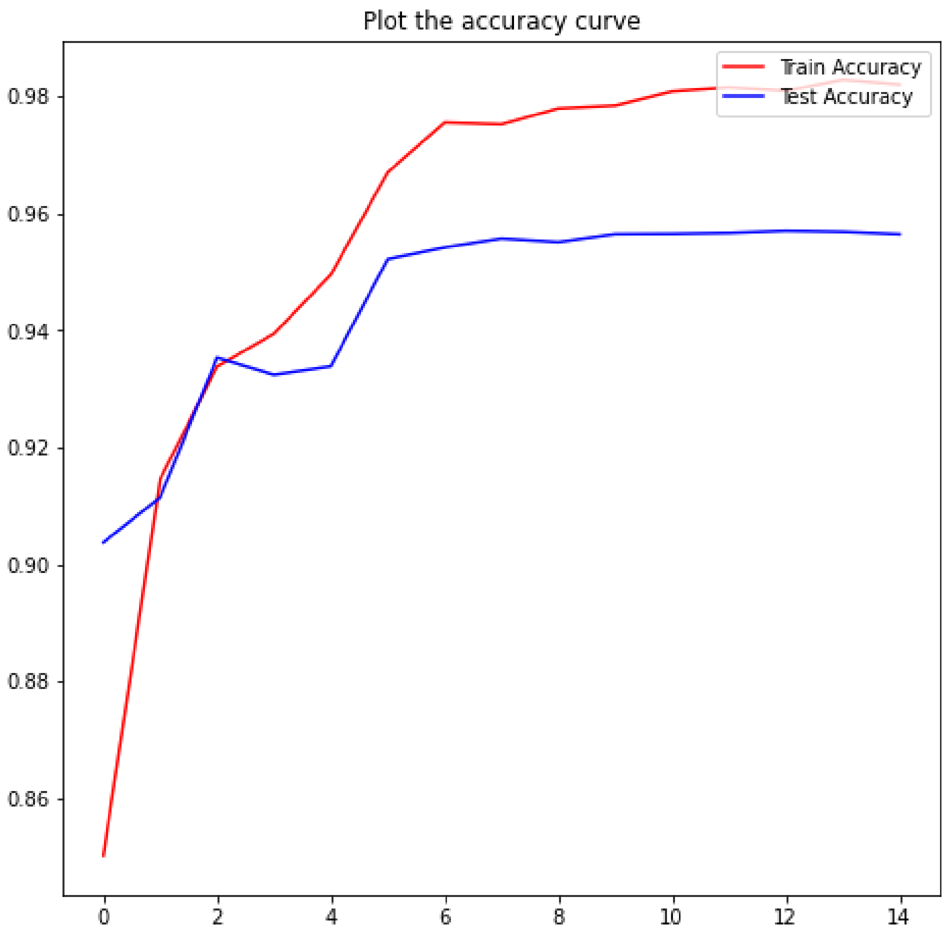
1. Plot the training and testing losses over epochs [2pt]

```
fig_1
```



2. Plot the training and testing accuracies over epochs [2pt]

```
1    fig_2
```



3. Print the final training and testing losses at convergence [2pt]

```
1    result_loss
```

	loss
training loss	0.054692
testing loss	0.158594

4. Print the final training and testing accuracies at convergence [20pt]

```
1    result_acc
```

	accuracy
training accuracy	0.982000
testing accuracy	0.956401