CS539 Assignment 04

Part 2

When the Network only has two fully connected network, that is **without CNN** layer, we got the results as follows:

```
[107]: device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
       model = Net().to(device)

       optimizer = optim.SGD(model.parameters(), lr=args['lr'], momentum=args['momentum'])

       for epoch in range(1, args['epochs'] + 1):
           train(epoch)
           test()

       Train Epoch: 1 [56640/60000 (94%)]      Loss: 0.285482
       Train Epoch: 1 [56960/60000 (95%)]      Loss: 0.054643
       Train Epoch: 1 [57280/60000 (95%)]      Loss: 0.015016
       Train Epoch: 1 [57600/60000 (96%)]      Loss: 0.023588
       Train Epoch: 1 [57920/60000 (97%)]      Loss: 0.188009
       Train Epoch: 1 [58240/60000 (97%)]      Loss: 0.290808
       Train Epoch: 1 [58560/60000 (98%)]      Loss: 0.135150
       Train Epoch: 1 [58880/60000 (98%)]      Loss: 0.015152
       Train Epoch: 1 [59200/60000 (99%)]      Loss: 0.104854
       Train Epoch: 1 [59520/60000 (99%)]      Loss: 0.047242
       Train Epoch: 1 [59840/60000 (100%)]     Loss: 0.048344
       /opt/anaconda3/lib/python3.12/site-packages/torch/nn/_reduction.py:42: UserWarning: size_average and reduce args will be deprecated, please
       se reduction='sum' instead.
         warnings.warn(warning.format(ret))

       Test set: Average loss: 0.0990, Accuracy: 9693/10000 (97%)
```

Average Loss: 0.0990

Accuracy: 9693/10000 (97%)

When the network has 1 CNN layer with pooling and two FC layers, that is **with CNN layer**, it resulted as:

```
[219]: device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
       model = NetWithCNN().to(device)

       optimizer = optim.SGD(model.parameters(), lr=args['lr'], momentum=args['momentum'])

       #optimizer = torch.optim.SGD(model.parameters(), lr=0.01, momentum=0.9)

       for epoch in range(1, args['epochs'] + 1):
           train(epoch)
           test()

       Train Epoch: 1 [53120/60000 (88%)]      Loss: 0.150327
       Train Epoch: 1 [53760/60000 (90%)]      Loss: 0.159112
       Train Epoch: 1 [54400/60000 (91%)]      Loss: 0.186541
       Train Epoch: 1 [55040/60000 (92%)]      Loss: 0.187486
       Train Epoch: 1 [55680/60000 (93%)]      Loss: 0.214006
       Train Epoch: 1 [56320/60000 (94%)]      Loss: 0.270832
       Train Epoch: 1 [56960/60000 (95%)]      Loss: 0.124888
       Train Epoch: 1 [57600/60000 (96%)]      Loss: 0.097894
       Train Epoch: 1 [58240/60000 (97%)]      Loss: 0.109332
       Train Epoch: 1 [58880/60000 (98%)]      Loss: 0.082911
       Train Epoch: 1 [59520/60000 (99%)]      Loss: 0.152225
       /opt/anaconda3/lib/python3.12/site-packages/torch/nn/_reduction.py:42: UserWarning: size_average and reduce args will be deprecated, please
       se reduction='sum' instead.
         warnings.warn(warning.format(ret))

       Test set: Average loss: 0.1585, Accuracy: 9554/10000 (96%)
```

Average Loss: 0.1585

Accuracy: 9554/10000 (96%)

Experimenting with 3 alternative network topologies and hyper-parameters as:

#1. 2 CNN layers + 2 FC layers

I used 2 CNN layers with pooling and 2 fully – connected (FC) layers.

```
[231]: device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
       model = NetExp1().to(device)

       optimizer = optim.SGD(model.parameters(), lr=args['lr'], momentum=args['momentum'])

       #optimizer = torch.optim.SGD(model.parameters(), lr=0.01, momentum=0.9)

       for epoch in range(1, args['epochs'] + 1):
           train(epoch)
           test()
```

```
Train Epoch: 1 [50560/60000 (84%)]      Loss: 0.060764
Train Epoch: 1 [51200/60000 (85%)]      Loss: 0.171581
Train Epoch: 1 [51840/60000 (86%)]      Loss: 0.300890
Train Epoch: 1 [52480/60000 (87%)]      Loss: 0.142780
Train Epoch: 1 [53120/60000 (88%)]      Loss: 0.265829
Train Epoch: 1 [53760/60000 (90%)]      Loss: 0.155733
Train Epoch: 1 [54400/60000 (91%)]      Loss: 0.223365
Train Epoch: 1 [55040/60000 (92%)]      Loss: 0.053974
Train Epoch: 1 [55680/60000 (93%)]      Loss: 0.130385
Train Epoch: 1 [56320/60000 (94%)]      Loss: 0.217469
Train Epoch: 1 [56960/60000 (95%)]      Loss: 0.138121
Train Epoch: 1 [57600/60000 (96%)]      Loss: 0.213913
Train Epoch: 1 [58240/60000 (97%)]      Loss: 0.137802
Train Epoch: 1 [58880/60000 (98%)]      Loss: 0.168160
Train Epoch: 1 [59520/60000 (99%)]      Loss: 0.169995

Test set: Average loss: 0.1690, Accuracy: 9494/10000 (95%)
```

Average Loss: 0.1690

Accuracy: 9494/10000 (95%)

#2. Learning Rate Tuning with 1 CNN layer model

I tested with 3 different learning rates with a simplified model (1 CNN layer):

Learning Rate	Test Loss (Avg Loss)	Test Accuracy (in %)	Notes
0.01	0.0585	97.90%	Best balance of speed/accuracy
0.001	0.1351	95.34%	Too slow convergence
0.5	2.3293	10.28%	Diverged

Best LR = 0.01: Achieved 97.9 % accuracy, the highest among the tested LRs.

#3. Batch Size Tuning

I tested with 3 different batch sizes with the model having 1 CNN layer with pooling and 2 FC layers with LR = 0.01 and momentum = 0.9

Batch Size	Test Loss (avg. loss)	Test Accuracy	Notes
16	0.0532	98.31%	Highest accuracy
32	0.0581	98.03%	Default; stable
64	0.0723	97.63%	Slightly higher loss

The best configuration was:

1 CNN + 2 FC with the Test accuracy as 98.31% having LR = 0.01 and batch size = 16

Through me experimenting, I learned that **hyperparameter tuning significantly impacts model performance**. First, the **learning rate** must be carefully chosen—too high (0.5) causes divergence, while too low (0.001) slows convergence, with 0.01 proving optimal for stable training. Second, **batch size** affects both accuracy and training dynamics: smaller batches (16) yielded the highest accuracy (98.31%) due to noisier but more exploratory updates, while larger batches (64) traded slight accuracy loss (97.63%) for faster training. Finally, **model complexity** isn't always better—a simpler 1-CNN-layer model outperformed a deeper 2-CNN-layer one (98.31% vs. 96.12%), suggesting that overly complex architectures can overfit even on simple datasets like MNIST. These results highlight the importance of **balanced hyperparameters**, **trade-offs between speed and precision**, and the **risk of over-engineering** models.