COMP3314 Assignment 2 - Report

G13 - Lohia Suyash (3035550406) & Gupta Aditya (3035662297) 20/12/21

Note: Late submission was approved

In this assignment we are applying convolutional neural network (CNN) methods on a digit recognition classification based problems to obtain a model with optimum accuracy and lowered error. The CNN is implemented using the PyTorch framework, and we have referred to the template provided by the course instructors.

This is a 10-category classification program in which the samples need to be classified into 0-9 based on the image pixelation. The neural net architecture so implemented contains CNN layers with Maxpool layers functions and ReLU activation functions complementing it. There are multiple parameters which could be tuned to obtain models with different characteristics including optimizer, data augmentation method, network architecture, initial learning rate, number of epochs, learning rate etc.

For this assignment we shall be keeping the data augmentation and network architecture as given and shall be experimenting with the optimizer setting, learning rate, learning rate decay and the number of epochs. Moreover, to make our analysis easier we shall be assuming that all these parameters are independent such that the number of experiments to be conducted remains reasonable.

System Configuration for the model development:

Computer Model: MacBook Pro (13-inch, 2019) CPU: 1.4 GHz 4-Core Intel Core i5

RAM: 8 GB 2133 MHz LPDDR3

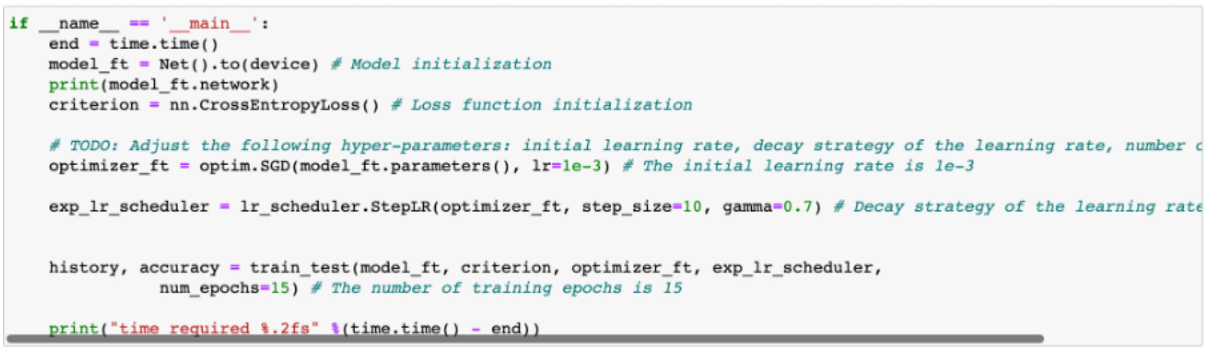
GPU: Intel Iris Plus Graphics 645 1536 MB

Optimizer Setting:

The two most commonly used optimizers are ADAM and SGD. We decided to experiment with the base settings as provided in the template and determine which optimizer better suits this classification task.

With ADAM: Testing accuracy = 88% (As also mentioned in the problem set)

With SGD: Testing accuracy = 9%



```python
if __name__ == '__main__':
    end = time.time()
    model_ft = Net().to(device) # Model initialization
    print(model_ft.network)
    criterion = nn.CrossEntropyLoss() # Loss function initialization

    # TODO: Adjust the following hyper-parameters: initial learning rate, decay strategy of the learning rate, number c
    optimizer_ft = optim.SGD(model_ft.parameters(), lr=1e-3) # The initial learning rate is 1e-3

    exp_lr_scheduler = lr_scheduler.StepLR(optimizer_ft, step_size=10, gamma=0.7) # Decay strategy of the learning rate


    history, accuracy = train_test(model_ft, criterion, optimizer_ft, exp_lr_scheduler,
                num_epochs=15) # The number of training epochs is 15

    print("time required %.2fs" %(time.time() - end))
```

```
Epoch:[10], training accuracy: 10.1, training loss: 2.303
Epoch:[11]-Iteration:[100], training loss: 2.303
Epoch:[11]-Iteration:[200], training loss: 2.303
Time cost of one epoch: [5569]s
Epoch:[11], training accuracy: 10.1, training loss: 2.303
Epoch:[12]-Iteration:[100], training loss: 2.303
Epoch:[12]-Iteration:[200], training loss: 2.303
Time cost of one epoch: [6264]s
Epoch:[12], training accuracy: 10.2, training loss: 2.303
Epoch:[13]-Iteration:[100], training loss: 2.303
Epoch:[13]-Iteration:[200], training loss: 2.303
Time cost of one epoch: [173]s
Epoch:[13], training accuracy: 10.1, training loss: 2.303
Epoch:[14]-Iteration:[100], training loss: 2.303
Epoch:[14]-Iteration:[200], training loss: 2.303
Time cost of one epoch: [172]s
Epoch:[14], training accuracy: 10.1, training loss: 2.303
Epoch:[15]-Iteration:[100], training loss: 2.303
Epoch:[15]-Iteration:[200], training loss: 2.303
Time cost of one epoch: [174]s
Epoch:[15], training accuracy: 10.2, training loss: 2.303
Finished Training
Accuracy of the network on test images: 9 %
time required 16952.52s
```

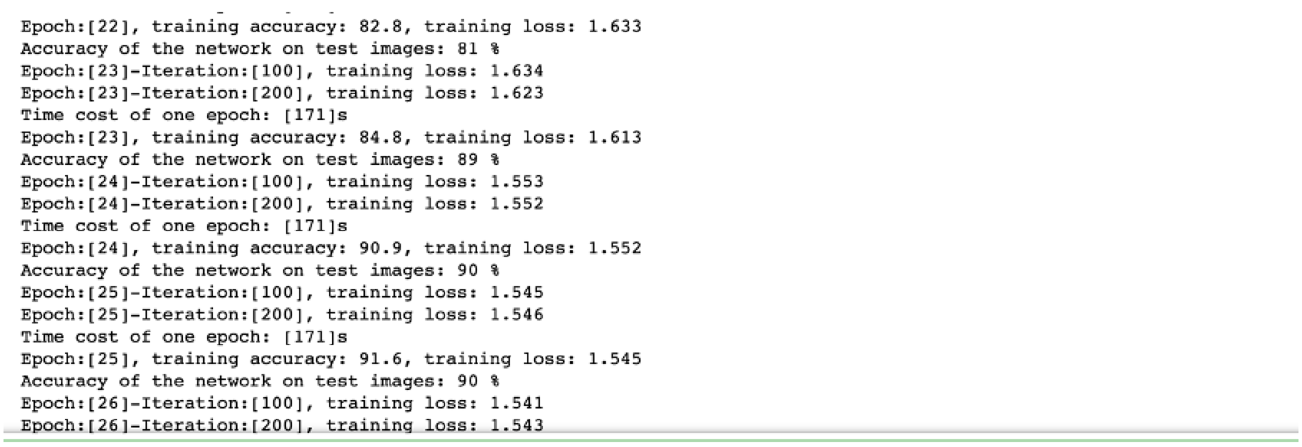
Hence based on the above analysis, we initially decided to use the ADAM optimizer for the model. After attempting many iterations with ADAM, we realised that while the convergence is good, it is not quick enough to finish within 45 minutes. Hence, we researched the ADAbelief optimizer, which is a newly formed optimization method and has higher average performance accuracy than other traditional methods. On comparing AdaBelief optimiser and ADAM optimiser, we saw a significant improvement and hence ended up using ADAbelief.

Number of Epochs:

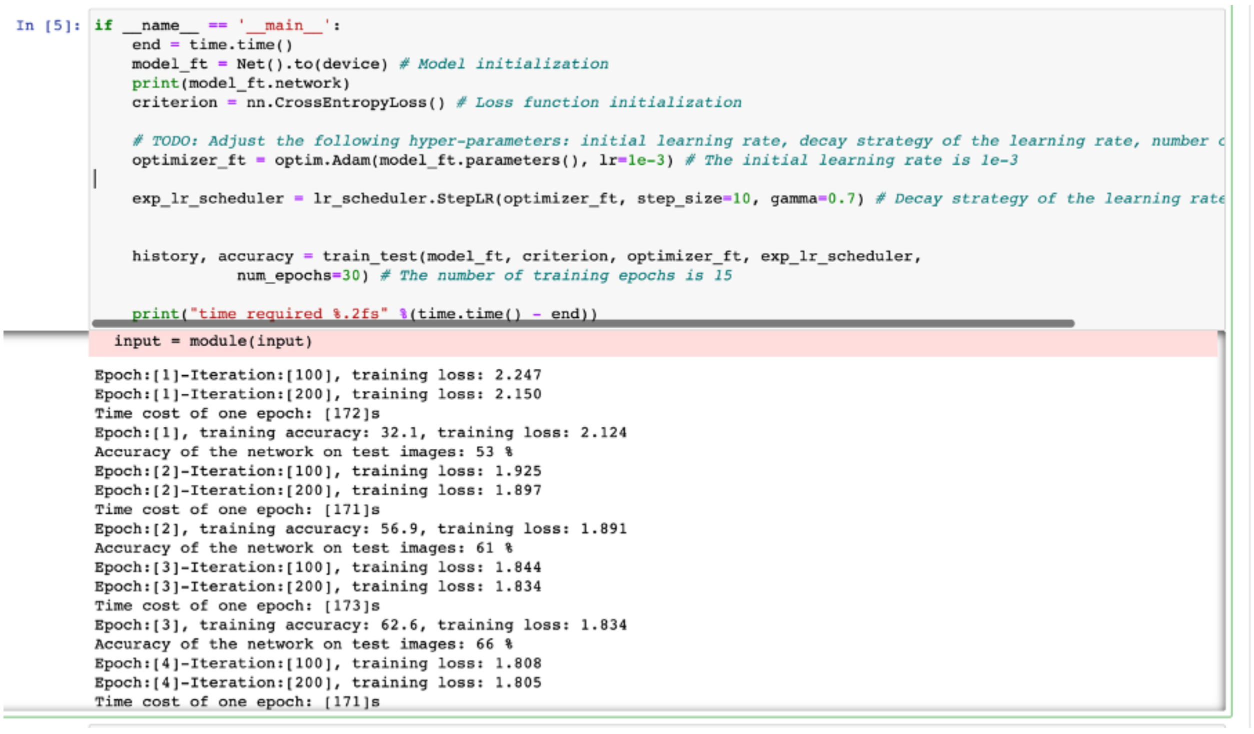
Number of epochs represents the number of iterations which the model undergoes in order to find the global minima of the cost function. The more the number of iterations, the higher the accuracy with the condition that the learning rate is decreased gradually.

On running the template file, we deduced that each epoch takes approximately 90 seconds. As one of the constraints of the assignment is a total training and testing time of 45 mins. We limited our experiments to a maximum epoch of 30.

To further deduce the ideal number of epochs, we introduced the same testing set for each iteration of the model development and set the number of epochs to 30.



```
Epoch:[22], training accuracy: 82.8, training loss: 1.633
Accuracy of the network on test images: 81 %
Epoch:[23]-Iteration:[100], training loss: 1.634
Epoch:[23]-Iteration:[200], training loss: 1.623
Time cost of one epoch: [171]s
Epoch:[23], training accuracy: 84.8, training loss: 1.613
Accuracy of the network on test images: 89 %
Epoch:[24]-Iteration:[100], training loss: 1.553
Epoch:[24]-Iteration:[200], training loss: 1.552
Time cost of one epoch: [171]s
Epoch:[24], training accuracy: 90.9, training loss: 1.552
Accuracy of the network on test images: 90 %
Epoch:[25]-Iteration:[100], training loss: 1.545
Epoch:[25]-Iteration:[200], training loss: 1.546
Time cost of one epoch: [171]s
Epoch:[25], training accuracy: 91.6, training loss: 1.545
Accuracy of the network on test images: 90 %
Epoch:[26]-Iteration:[100], training loss: 1.541
Epoch:[26]-Iteration:[200], training loss: 1.543
```



```
In [5]: if __name__ == '__main__':
            end = time.time()
            model_ft = Net().to(device) # Model initialization
            print(model_ft.network)
            criterion = nn.CrossEntropyLoss() # Loss function initialization

            # TODO: Adjust the following hyper-parameters: initial learning rate, decay strategy of the learning rate, number c
            optimizer_ft = optim.Adam(model_ft.parameters(), lr=1e-3) # The initial learning rate is 1e-3

            exp_lr_scheduler = lr_scheduler.StepLR(optimizer_ft, step_size=10, gamma=0.7) # Decay strategy of the learning rate


            history, accuracy = train_test(model_ft, criterion, optimizer_ft, exp_lr_scheduler,
                                           num_epochs=30) # The number of training epochs is 15

            print("time required %.2fs" %(time.time() - end))
```

```
  input = module(input)

Epoch:[1]-Iteration:[100], training loss: 2.247
Epoch:[1]-Iteration:[200], training loss: 2.150
Time cost of one epoch: [172]s
Epoch:[1], training accuracy: 32.1, training loss: 2.124
Accuracy of the network on test images: 53 %
Epoch:[2]-Iteration:[100], training loss: 1.925
Epoch:[2]-Iteration:[200], training loss: 1.897
Time cost of one epoch: [171]s
Epoch:[2], training accuracy: 56.9, training loss: 1.891
Accuracy of the network on test images: 61 %
Epoch:[3]-Iteration:[100], training loss: 1.844
Epoch:[3]-Iteration:[200], training loss: 1.834
Time cost of one epoch: [173]s
Epoch:[3], training accuracy: 62.6, training loss: 1.834
Accuracy of the network on test images: 66 %
Epoch:[4]-Iteration:[100], training loss: 1.808
Epoch:[4]-Iteration:[200], training loss: 1.805
Time cost of one epoch: [171]s
```

On conducting the experiment we observed that the initial accuracy of the model is very low and it makes big jumps in order to move towards the direction of the global minima. Around the 24th iteration, the testing accuracy crossed 90%, and post that the increase in the testing accuracy per iteration was getting less significant.

Hence based on the above analysis, we decided to use the number of epochs = 23

Initial Learning Rate:

Initial learning rate is the multiplication factor by which the weights are updated for each iteration. It directly affects the magnitude of the bounce for the gradient descent cost minimization approach. An ideal learning rate is one which isn't too high nor too low.

In order to determine the ideal learning rate we iterated the model over multiple values of the learning rate, and then found the one with the highest increase in accuracy over a single epoch.

3]:

```
learning_rate = [1e-4,2.5e-4,5e-4,7.5e-4,1e-3,2.5e-3,5e-3,7.5e-3,1e-2,2.5e-2,5e-2,
                 7.5e-2,1e-1,2.5e-1,5e-1,1,5,10,50,100]
```

1]:

```
def iterate_lr(learning_rate):
    end = time.time()
    model_ft = Net().to(device) # Model initialization
    print(model_ft.network)
    criterion = nn.CrossEntropyLoss() # Loss function initialization

    # TODO: Adjust the following hyper-parameters: initial learning rate, decay strategy of the learning rate, number of training epochs
    optimizer_ft = optim.Adam(model_ft.parameters(), lr=learning_rate) # The initial learning rate is 1e-3

    #    exp_lr_scheduler = lr_scheduler.StepLR(optimizer_ft, step_size=10, gamma=0.7) # Decay strategy of the learning rate


    history, accuracy = train_test(model_ft, criterion, optimizer_ft,# exp_lr_scheduler,ignored for this ipynb
                num_epochs=1) # The number of training epochs is 15

    print("time required %.2fs" %(time.time() - end))

    return history, accuracy
```

2]:

```
%%notify

for x in learning_rate:
    temp1,temp2 = iterate_lr(x)
    train_loss_plot.append(temp1)
    test_loss_plot.append(temp2)
```

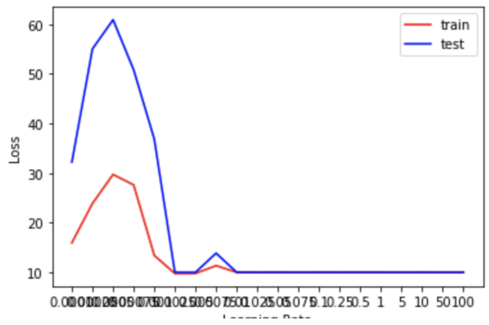
```
print(max(train_loss_accuracy))
print(max(test_loss_plot))

print(learning_rate[2])

print(train_loss_accuracy[2] == max(train_loss_accuracy))
print(test_loss_plot[2] == max(test_loss_plot))
```

```
29.720000000000002
60.92
0.0005
True
True
```

```
print(learning_rate)
```

```
[0.0001, 0.00025, 0.0005, 0.00075, 0.001, 0.0025, 0.005, 0.0075, 0.01, 0.025, 0.05, 0.075, 0.1, 0.25, 0.5, 1, 5, 10, 50, 100]
```

```
plt.plot(x_axis,train_loss_accuracy, 'r',label='train')
plt.plot(x_axis,test_loss_plot,'b',label='test')
plt.xlabel("Learning Rate")
plt.ylabel("Loss")
plt.legend()
plt.show()
```



Ideally we would run each learning rate value for every epoch, but an initial optimal value is sufficient after which the decay strategy can be optimised accordingly. As seen, the best learning rate value with no decay strategy seems to be = 0.0005 with a training accuracy of 30% and a testing accuracy of 61% after just one epoch.

Learning Rate Decay:

Learning rate decay is the value by which the learning rate is multiplied recursively in order to decrease it. Hence it is less than 1. This is significant because if the learning rate isn't decreased then the model will keep on bouncing around the global minima and not actually fixate on it.

There are multiple methods to implement learning rate decay. The ones experimented by us include StepLR, Multi-StepLR, Exponential LR.

StepLR is when we choose a number of epochs after which our given learning rate is multiplied by gamma. Multi-Step LR is slightly more powerful in that it allows us to specify the exact epochs where we want the learning rate to decrease. Exponential LR multiplies the learning rate by gamma at every epoch.

We performed multiple experiments in order to get a model with high accuracy. A brief overview of which is below.

Step LR: Step Size ranging from 8-12 and gamma ranging from .6 to .9.
MultiStepLR: Different no. of milestones with different epoch numbers and gamma ranging from .6 to .9.
ExponentialLR: Different gamma rates ranging from .8 to .99.

We used all three approaches, with a combination of different initial learning rates and gamma values in order to increase the accuracy of our model within 23 epochs.

Final Model:

In the final model we used the following hyperparameters:
Number of epochs = 23
Optimised = AdaBelief
Learning rate = 0.0007
Learning rate decay strategy = MultiStepLR with milestones = [15,20]
Learning rate decay gamma = 0.6

As seen below, the model finished training and testing with 45 minutes and we received a final training accuracy of **91.2 %**.

In [5]:
```python
from adabelief_pytorch import AdaBelief

if __name__ == '__main__':
    end = time.time()
    model_ft = Net().to(device) # Model initialization
    print(model_ft.network)
    criterion = nn.CrossEntropyLoss() # Loss function initialization

    # TODO: Adjust the following hyper-parameters: initial learning rate, decay strategy of the learning rate, numl
    optimizer_ft = AdaBelief(model_ft.parameters(), lr=0.0007) # The initial learning rate is 1e-3

    exp_lr_scheduler = lr_scheduler.MultiStepLR(optimizer_ft, milestones=[15,20],gamma=0.6) # Decay strategy of the



    history, accuracy = train_test(model_ft, criterion, optimizer_ft, exp_lr_scheduler,
                num_epochs=23) # The number of training epochs is 15

    print("time required %.2fs" %(time.time() - end))
```

```
Epoch:[20]-Iteration:[100], training loss: 1.566
Epoch:[20]-Iteration:[200], training loss: 1.567
Time cost of one epoch: [133]s
Epoch:[20], training accuracy: 89.6, training loss: 1.566
Epoch:[21]-Iteration:[100], training loss: 1.555
Epoch:[21]-Iteration:[200], training loss: 1.558
Time cost of one epoch: [139]s
Epoch:[21], training accuracy: 90.4, training loss: 1.557
Epoch:[22]-Iteration:[100], training loss: 1.553
Epoch:[22]-Iteration:[200], training loss: 1.553
Time cost of one epoch: [141]s
Epoch:[22], training accuracy: 90.8, training loss: 1.553
Epoch:[23]-Iteration:[100], training loss: 1.551
Epoch:[23]-Iteration:[200], training loss: 1.550
Time cost of one epoch: [135]s
Epoch:[23], training accuracy: 91.1, training loss: 1.551
Finished Training
Accuracy of the network on test images: 91.200 %
time required 2722.86s
```

In [7]:
```python
print(accuracy)
```

```
91.2
```

In [6]:
```python
history
```

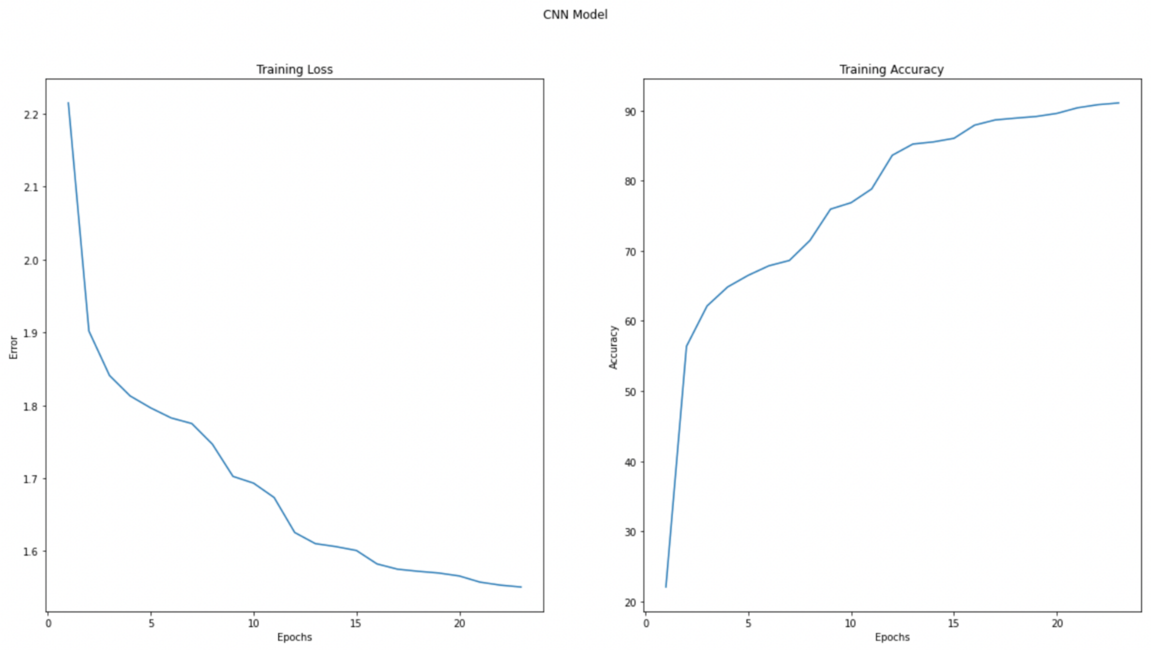
Accuracy of each class:

```python
import pandas as pd

class_accuracy = dict({})
class_total = [0 for i in range(10)]
class_correct = [0 for i in range(10)]
correct = 0
total = 0
model_ft.eval()
with torch.no_grad():
        for data in test_dataloader:
            images, labels = data[0].to(device), data[1].to(device)
            # calculate outputs by running images through the network
            outputs = model_ft(images)
            # the class with the highest energy is what we choose as prediction
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()
            for c in range(10): #all classes
                class_total[c] += (labels==c).sum().item()
                class_correct[c] += ((predicted==labels)*(labels==c)).sum().item()
                if(class_total[c] != 0):
                    class_accuracy[c] = class_correct[c]/class_total[c] *100

accuracy = 100 * correct / total
print('Accuracy Statistics:\n')
print(f'Overall Accuracy: {accuracy}%')
df = pd.DataFrame(list(class_accuracy.items()), columns=['Class','Accuracy %'])
print(df.to_string(index=False))
```

```
Accuracy Statistics:

Overall Accuracy: 91.2%
 Class  Accuracy %
     0        92.6
     1        92.6
     2        92.8
     3        86.6
     4        95.6
     5        93.0
     6        89.0
     7        92.6
     8        87.4
     9        89.8
```

Summary of Hyperparameters & Training Loss/Accuracy Plots:

```
CNN Model Hyperparameters:
Learning Rate: 0.0007
Decay Strategy: Milestones - Counter({15: 1, 20: 1}), Gamma - 0.6
Epochs: 23
Text(0, 0.5, 'Accuracy')
```


CNN Model
Training Loss
Error
Epochs
Training Accuracy
Accuracy
Epochs