AI6121- Computer Vision Course Project

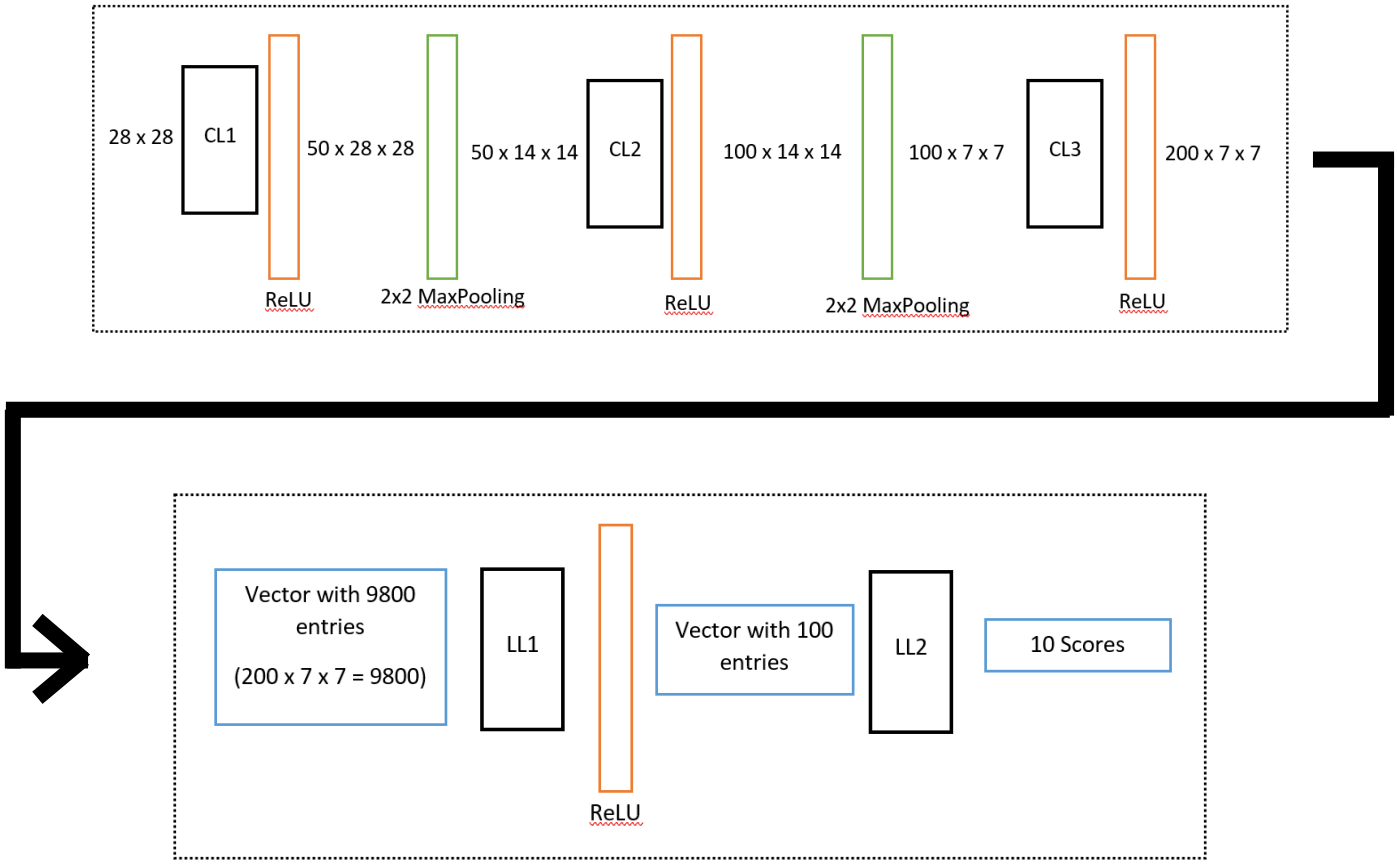
Brandon Chua Shaojie

G1903442H

1) Design and develop your handwritten digit recognition network. Train your network over the MNIST training set, evaluate over the test set, and report your evaluation results. Your project report should include detailed description and discussion of your network design as well as your evaluation results.

Network Design:

For the project, I will be using a convolution neural network. It will consist of 3 convolution layers, 2 max pooling layers and 2 fully connected layers.



Layer 1 (Convolution Layer 1):

Code:

```
self.conv1 = nn.Conv2d(1,   50,  kernel_size=3,  padding=1 )
```

Input Dimensions ----------> Output Dimensions

28 x 28 ----------> 50 x 28 x 28

This convolution layer takes in an MNIST image of size 28 x 28 pixels and outputs 50 activation maps of 28 x 28 pixels. It uses a convolution filter of 3x3, therefore a padding of 1 is applied so that the image size will remain the same for both input and output.

Layer 2 (Max Pooling Layer 1):

Code:

```
self.pool1  = nn.MaxPool2d(2,2)
```

Input Dimensions ----------> Output Dimensions

50 x 28 x 28 ----------> 50 x 14 x 14

This max pooling layer with a 2 x 2 window size takes in an input of 50 x 28 x 28 and outputs 50 x 14 x 14. This has two functionalities. The first function is to trade image resolution with features number. This means for a reduction in image resolution, which get to capture more data structures. The second function helps to control the computational complexity since the image size is reduced by 2.

Layer 3 (Convolution Layer 2):

Code:

```
self.conv2 = nn.Conv2d(50,  100,  kernel_size=3,  padding=1 )
```

Input Dimensions ----------> Output Dimensions

50 x 14 x 14 ----------> 100 x 14 x 14

This convolution layer takes in an input of size 50 x 14 x 14 and outputs 100 activation maps of 14 x 14 pixels. It uses a convolution filter of 3x3, therefore a padding of 1 is applied so that the image size will remain the same for both input and output.

Layer 4 (Max Pooling Layer 2):

Code:

```
self.pool2 = nn.MaxPool2d(2,2)
```

Input Dimensions ----------> Output Dimensions

100 x 14 x 14 ----------> 100 x 7 x 7

This max pooling layer with a 2 x 2 window size takes in an input of 100 x 14 x 14 and outputs 100 x 7 x 7.

Layer 5 (Convolution Layer 3):

```
Code:
self.conv3 = nn.Conv2d(100,  200,  kernel_size=3,  padding=1 )
```

Input Dimensions ----------> Output Dimensions
100 x 7 x 7----------> 200 x 7 x 7

This convolution layer takes in an input of size 100 x 7 x 7 and outputs 200 activation maps of 7 x 7 pixels. It uses a convolution filter of 3x3, therefore a padding of 1 is applied so that the image size will remain the same for both input and output.

Layer 6 (Linear Layer 1):

```
Code:
self.linear1 = nn.Linear(9800, 100)
```

Input Dimensions ----------> Output Dimensions
(200 x 7 x 7) 9800 ----------> 100

This fully connected layer takes in an input vector of 9800 entries and outputs a vector with 100 entries.

Layer 7 (Linear Layer 2):

```
Code:
self.linear2 = nn.Linear(100,10)
```

Input Dimensions ----------> Output Dimensions
100 ----------> 10

This fully connected layer takes in an input vector of 100 entries and outputs the scores for which digit the image belongs to. The reason we have 2 fully connected layers instead of using 1 to do the classification is because having a network with more depth is always better if does not result in the network being too computationally expensive.

Other network parameters:

Activation Function:

For my network, I chose to use ReLU as the activation function since sigmoid and tanh activation functions faces the vanishing gradient problem. Another advantage of ReLU is that it is computationally less expensive since the output will be 0 for negative values. For sigmoid and tanh activation functions, they will still output for negative values, which causes an issue since these activations does not contribute much and thus are often redundant.

Loss Function:

For the loss function, I have chosen to use cross entropy loss.

Optimizer Algorithm:

Stochastic gradient descent is chosen as the optimizer algorithm since it is better for generalization when compared to mini-batch gradient descent and gradient descent. Mini-batch gradient descent approximates the gradient by averaging over 200 data points. It is the fastest among these 3 but is weaker at generalization than stochastic gradient descent. Gradient descent averages over all data points, thus making it the slowest and weakest at generalization.

Learning rate:

Learning rate of 0.25 is used, and it is divided by 2 after every 5 epochs. This is to help the network converge since a larger learning rate causes the network to make larger “jumps” that misses the minimum. Furthermore, if the learning rate is too high, it will cause the gradient to explode, making the network unstable and unable to learn properly.

Evaluation on Test Results:

Epoch	Learning Rate	Total Time / min	Loss	Error %	Error on Test Set %
1	0.25	0.12347420454025268	0.2504749227045124	7.952425373134328	1.0383702531645569
4	0.25	0.46680911382039386	0.020042951521612624	0.623001066098081	0.8010284810126581
9	0.125	1.043944768110911	0.0030627430595114016	0.07162846481876334	0.6823575949367089
14	0.0625	1.6243728121121725	0.0010606160078967574	0.026652452025586353	0.6428006329113924
19	0.03125	2.214150297641754	0.0006743843859828103	0.009994669509594883	0.6922468354430379
24	0.015625	2.81508629322052	0.0005486787593827199	0.0049973347547974415	0.6625791139240506
29	0.0078125	3.4314669092496235	0.0005012606697550207	0.0049973347547974415	0.6625791139240506

Loss is the calculated using cross entropy loss.

Error is calculated as percentage of wrong predictions over all predictions.

Accuracy is calculated as percentage of right predictions over all predictions.

From the evaluation results we can see that by the end of 30 epochs, the error on the training set is at 0.005% which is very low. This is good. As for the error on the test set, it is 0.7%, meaning the network has an accuracy rate of 99.3%. This indicates that the network performs well on the MNIST data set.

From the table, it can be seen that as training goes on, the loss, error and error on test set decreases. This is good as it means our network is learning and improving in accuracy.

At around epoch 24, both error and error on test set have started to converge. This is evident since the error and error on test set is the same at epoch 24 and epoch 29.

As mentioned before, the learning rate is divided by 2 for every 5 epochs. Thus, as the training goes on, the learning rate decreases.

2) Investigate different hyper-parameters and design options such as learning rate, optimizers, loss functions, etc. to study how they affect the network performance.

For this part, I will be investigating the following hyper-parameters:

Epochs

Learning Rate

Activation Function

Optimizer

Batch Size

During investigation, I will be using the following hyper-parameters as the **benchmark**:

Epochs: 30

Learning Rate: 0.25 (Divided by 2 for every 5 epochs)

Loss Function: Cross Entropy Loss

Activation Function: ReLU

Optimizer: SGD

Batch Size: 128

The only hyper-parameter that will be changed will be the one that is currently being investigated.

Epochs

I will be investigating the effects of using different number of epochs of 10, 30 and 50.

Epochs	Total Time / min	Loss	Error %	Error on Test Set %
10	1.249999213218689	0.004203985542503756	0.09994669509594883	0.7318037974683544
BENCHMARK 30	4.2071090539296465	0.000583960006742287	0.0049973347547974415	0.6922468354430379
50	6.912435134251912	0.000544862783317124	0.0049973347547974415	0.6823575949367089

Findings	
Total Time / min	Time taken to train increases as epochs increases.
Loss	Loss decreases as epochs increases. Loss converges around epoch 30 as seen from the fact that loss barely decreases from epoch 30 to epoch 50.
Error %	Error decreases as epochs increases. Error converges around epoch 30 as seen from the fact that error is the same at epoch 30 and epoch 50.
Error on Test Set %	Error on test set decreases as epochs increases. Error on test set converges around epoch 30 since the error on test set barely decreases from epoch 30 to epoch 50.
Conclusion	If the number of epochs is too small, the network is not able to be fully trained and therefore converge, thus resulting in the network performing poorly. If the number of epochs is too large, it is a waste of time and computation since the network would have already converged. Thus, it is important to set the right number of epochs for the network to train efficiently and perform well.

Learning Rate (Epoch 30)

Learning Rate	Total Time / min	Loss	Error %	Error on Test Set %
0.01 (without dividing by 2 every 5 epochs)	3.9546990553538004	0.019687562305946895	0.5863539445628998	1.0284810126582278
0.25 (without dividing by 2 every 5 epochs)	4.068161328633626	2.7421896306213674e-05	0.0	0.5933544303797469
2 (without dividing by 2 every 5 epochs)	3.9039647976557412	NaN	90.12693230277186	90.23931962025317
0.01 (divide by 2 every 5 epochs)	4.157363220055898	0.046027906429665935	1.335954157782516	1.4438291139240507
BENCHMARK 0.25 (divide by 2 every 5 epochs)	4.2071090539296465	0.000583960006742287	0.0049973347547974415	0.6922468354430379
2 (divide by 2 every 5 epochs)	4.45417324701945	NaN	90.12915334467695	90.23931962025317

Findings	
Total Time / min	Time taken to train is not significantly affected by learning rate.
Performance based on loss and error	In both cases of starting with a learning rate of 2, there is the case of exploding gradient. This is apparent from how the network has a high error rate of 90% on both the training and test set. This means that learning rate cannot be too large. When a learning rate of 0.01 and 0.25 (divided by 2 every 5 epoch) is used, it converges slower than a learning rate of 0.01 and 0.25 (without dividing by 2 every 5 epoch). This shows that a smaller learning rate makes training slower. When a learning rate of 0.01 (without dividing by 2 every 5 epoch) is used, it converges slower than a learning rate of 0.25(without dividing by 2 every 5 epoch). This also shows that the learning rate should not be too small.
Conclusion	Learning rate should not be too large, or it will result in gradient explosion. It should not be too small, or the network will take too long to converge. This means that we must find a learning rate that is not too large or too small. From my findings, 0.01 is too small, 2 is too large, while a learning rate of 0.25 is good. Using an adaptive training approach can be useful. We can start with a large learning rate to increase training speed and then as training goes on, we can reduce the learning rate so that the network converges. However, from my findings, using adaptive training approach on the network performs worse than not using it. Thus, finding the right parameters for adaptive training is important.

Activation Function (Epoch 30)

Activation Function	Time / min	Loss	Error %	Error on Test Set %
BENCHMARK ReLU	4.2071090539296465	0.000583960006742287	0.0049973347547974415	0.6922468354430379
Tanh	3.6439873655637105	0.0036512535640787183	0.03664712153518124	0.7318037974683544
Sigmoid	3.5801227847735086	2.301120990883313	88.76432569296375	88.63726265822784

Findings	
Total Time / min	Time taken to train is not significantly affected by activation function.
Performance based on loss and error	Both Tanh and Sigmoid activation functions performed worse than ReLU. They both have higher error and loss. In the case of Sigmoid, it is observed that there was a gradient explosion.
Conclusion	ReLU performs better than Tanh and Sigmoid. The reason is that both Tanh and Sigmoid are more likely to output non-zero values when compared to ReLU. ReLU does not face this problem due to its characteristic of outputting zero when input is less than or equal to zero. This means ReLU has the property of sparsity and sparsity is more beneficial than dense representations. Tanh performs better than Sigmoid. The reason may due to the fact that Tanh ranges from -1 to 1 and Sigmoid ranges from 0 and 1. Since Tanh is centred at zero, it makes convergence for the network eaiser.

Optimizer (Epoch 30)

Optimizer	Total Time / min	Loss	Error %	Error on Test Set %
BENCHMARK SGD	4.2071090539296465	0.000583960006742287	0.0049973347547974415	0.6922468354430379
Adam	3.7233537276585897	2.301736035580828	88.83817519968761	88.63726265822784
ASGD	3.6042807896931968	0.0009363575652558188	0.009994669509594883	0.6922468354430379
Adagrad	3.6380571881930033	2.3013061150304797	88.76488095661725	88.63726265822784

Findings	
Total Time / min	Time taken to train is not significantly affected by optimizer.
Performance based on loss and error	Both Adam and Adagrad that uses adaptive learning rates perform badly. Both caused the network to fail to converge and train properly. SGD and ASGD are very close in performance, with SGD having better loss and error rate on training set.
Conclusion	For the using of CNN on MNIST dataset, optimizer with adaptive learning rates are not suitable. On the other hand, SGD and ASGD performs well.

Batch Size (Epoch 30)

Batch Size	Total Time / min	Loss	Error %	Error on Test Set %
1	130.413108984629	2.3030749865690865	89.18	89.68
BENCHMARK 128	4.2071090539296465	0.000583960006742287	0.0049973347547974415	0.6922468354430379
3000	2.7388245423634845	0.07416450381278991	2.181667685508728	2.175000309944153

Findings	
Total Time / min	Time taken to train increases as batch size decreases.
Performance based on loss and error	Performance is best when a batch size of 128 is used. Performance is worst when a batch size of 1 is used. A likely reason is that since it updates its weights for after every example, learning is very unstable and thus the network is unable to converge.
Conclusion	A batch size too small will lead to very slow training time. A big batch size allows a network to iterate faster but the network does not perform as well when compared to using a more suitable batch size. A big batch size also requires more GPU memory. Therefore, finding the right batch size that is not too small or big is best for training the network.

Overall Conclusion

Best Hyper-parameters for training my network

Epochs	30
Learning Rate	0.25 (Divided by 2 for every 5 epochs)
Loss Function	Cross Entropy Loss
Activation Function	ReLU
Optimizer	SGD /ASGD
Batch Size	128

From my findings, hyper-parameters that are too extreme (too small or too large) are bad choices, usually resulting in either slow training or poor performance. The best way is to choose hyper-parameters that balances both training time and performance of the network. However, there is no one size fits all. For different networks, different hyper-parameters might be needed to optimize the algorithm. Therefore, fine tuning is very crucial in neural networks.

References

Bresson, X. (2020, August 12). *AI6103_2020*. Retrieved from github: https://github.com/xbresson/AI6103_2020

Brownlee, J. (2019, August 14). *A Gentle Introduction to Exploding Gradients in Neural Networks*. Retrieved from machinelearningmastery: https://machinelearningmastery.com/exploding-gradients-in-neural-networks/

Chris. (2019, September 4). *ReLU, Sigmoid and Tanh: today's most used activation functions*. Retrieved from machinecurve: https://www.machinecurve.com/index.php/2019/09/04/relu-sigmoid-and-tanh-todays-most-used-activation-functions/#differences-between-tanh-and-sigmoid

Jha, P. (2019, January 6). *A Brief Overview of Loss Functions in Pytorch*. Retrieved from medium: https://medium.com/udacity-pytorch-challengers/a-brief-overview-of-loss-functions-in-pytorch-c0ddb78068f7

Radhakrishnan, P. (2017, August 10). *What are Hyperparameters ? and How to tune the Hyperparameters in a Deep Neural Network?* Retrieved from towardsdatascience: https://towardsdatascience.com/what-are-hyperparameters-and-how-to-tune-the-hyperparameters-in-a-deep-neural-network-d0604917584a

Ruder, S. (2016, January 19). *An overview of gradient descent optimization algorithms*. Retrieved from ruder: https://ruder.io/optimizing-gradient-descent/index.html#adagrad

Singh, A. V. (2020, June 6). *Vanishing and Exploding Gradients With Sigmoid Activation (function)*. Retrieved from levelup: https://levelup.gitconnected.com/vanishing-and-exploding-gradients-ae7fb88f3b66

V, A. S. (2017, March 31). *Understanding Activation Functions in Neural Networks*. Retrieved from medium: https://medium.com/the-theory-of-everything/understanding-activation-functions-in-neural-networks-9491262884e0

Appendix

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from random import randint
import time

#Use GPU
device= torch.device("cuda")
print(device)

#Read in files
trainData=torch.load('mnist/train_data.pt')
trainLabel=torch.load('mnist/train_label.pt')
testData=torch.load('mnist/test_data.pt')
testLabel=torch.load('mnist/test_label.pt')

print(trainData.size())
print(testData.size())

#mean and std to normalize
mean= trainData.mean()
std= trainData.std()

print(mean)
print(std)

#CNN fucntion
class CNN(nn.Module):

    def __init__(self):

        super(CNN, self).__init__()
```

```
        self.conv1 = nn.Conv2d(1,   50,  kernel_size=3,  padding=1 )
        self.pool1  = nn.MaxPool2d(2,2)

        self.conv2 = nn.Conv2d(50,  100,  kernel_size=3,  padding=1 )
        self.pool2 = nn.MaxPool2d(2,2)

        self.conv3 = nn.Conv2d(100,  200,  kernel_size=3,  padding=1 )

        self.linear1 = nn.Linear(9800, 100)

        self.linear2 = nn.Linear(100,10)


    def forward(self, x):

        x = self.conv1(x)
        x = F.relu(x)
        x = self.pool1(x)

        x = self.conv2(x)
        x = F.relu(x)
        x = self.pool2(x)

        x = self.conv3(x)
        x = F.relu(x)

        x = x.view(-1, 9800)
        x = self.linear1(x)
        x = F.relu(x)

        x = self.linear2(x)

        return x
```

```
net=CNN()

#Send to GPU
net = net.to(device)
mean=mean.to(device)
std=std.to(device)

#Hyperparameters

epochs = 30

lossFunction = nn.CrossEntropyLoss()

learningRate = 0.25

batchSize = 128

#Calculate Error
def calcError( scores , labels ):

    bs=scores.size(0)
    predictedLabels = scores.argmax(dim=1)
    indicator = (predictedLabels == labels)
    numMatches=indicator.sum()

  return 1-numMatches.float()/bs

  #Evaluation Function

  def evalTestData():

      runningError=0
      numBatches=0

      for i in range(0,10000,batchSize):
```

```
        miniBatchData =  testData[i:i+batchSize].unsqueeze(dim=1)
        miniBatchLabel= testLabel[i:i+batchSize]

        miniBatchData=miniBatchData.to(device)
        miniBatchLabel=miniBatchLabel.to(device)

        inputs = (miniBatchData - mean)/std

        scores=net( inputs )

        error = calcError( scores , miniBatchLabel)

        runningError += error.item()

        numBatches+=1


    totalError = runningError/numBatches
    print( 'error rate on test set =', totalError*100 ,'percent')

#Training Loop

start=time.time()

for epoch in range(1,epochs):

    if not epoch%5:
        learningRate = learningRate / 2

    #Optimizer Hyperparameter
    optimizer=torch.optim.SGD( net.parameters() , lr=learningRate )

    runningLoss=0
    runningError=0
```

```
numBatches=0

shuffledIndices=torch.randperm(60000)

for count in range(0,60000,batchSize):

    optimizer.zero_grad()

    indices=shuffledIndices[count:count+batchSize]
    miniBatchData =  trainData[indices].unsqueeze(dim=1)
    miniBatchLabel=  trainLabel[indices]

    miniBatchData=miniBatchData.to(device)
    miniBatchLabel=miniBatchLabel.to(device)

    inputs = (miniBatchData - mean)/std

    inputs.requires_grad_()

    scores=net( inputs )

    loss =  lossFunction( scores , miniBatchLabel)

    loss.backward()
    optimizer.step()


    # Calculate Error

    runningLoss += loss.detach().item()

    error = calcError( scores.detach() , miniBatchLabel)
    runningError += error.item()

    numBatches+=1
```

```
      # Display average error
      totalLoss = runningLoss/numBatches
      totalError = runningError/numBatches
      elapsed = (time.time()-start)/60

      print('epoch=',epoch, '\t time=', elapsed,'min', '\t lr=',
learningRate  ,'\t loss=', totalLoss , '\t error=',
totalError*100 ,'percent')
      evalTestData()
      print(' ')
```

In [ ]:
```
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from random import randint
import time
```

In [ ]:
```
#Use GPU
device= torch.device("cuda")
print(device)
```

In [ ]:
```
#Read in files
trainData=torch.load('mnist/train_data.pt')
trainLabel=torch.load('mnist/train_label.pt')
testData=torch.load('mnist/test_data.pt')
testLabel=torch.load('mnist/test_label.pt')

print(trainData.size())
print(testData.size())
```

In [ ]:
```
#mean and std to normalize
mean= trainData.mean()
std= trainData.std()

print(mean)
print(std)
```

In [ ]:

```python
#CNN fucntion
class CNN(nn.Module):

    def __init__(self):

        super(CNN, self).__init__()

        self.conv1 = nn.Conv2d(1,   50,  kernel_size=3,  padding=1 )
        self.pool1  = nn.MaxPool2d(2,2)

        self.conv2 = nn.Conv2d(50,  100,  kernel_size=3,  padding=1 )
        self.pool2 = nn.MaxPool2d(2,2)

        self.conv3 = nn.Conv2d(100,  200,  kernel_size=3,  padding=1 )

        self.linear1 = nn.Linear(9800, 100)

        self.linear2 = nn.Linear(100,10)


    def forward(self, x):

        x = self.conv1(x)
        x = F.relu(x)
        x = self.pool1(x)

        x = self.conv2(x)
        x = F.relu(x)
        x = self.pool2(x)

        x = self.conv3(x)
        x = F.relu(x)

        x = x.view(-1, 9800)
        x = self.linear1(x)
        x = F.relu(x)

        x = self.linear2(x)

        return x
```

In [ ]:

```python
net=CNN()

#Send to GPU
net = net.to(device)
mean=mean.to(device)
std=std.to(device)
```

In [ ]:

```python
#Hyperparameters

epochs = 30

lossFunction = nn.CrossEntropyLoss()

learningRate = 0.25

batchSize = 128
```

In [ ]:

```python
#Calculate Error
def calcError( scores , labels ):

    bs=scores.size(0)
    predictedLabels = scores.argmax(dim=1)
    indicator = (predictedLabels == labels)
    numMatches=indicator.sum()

    return 1-numMatches.float()/bs
```

In [ ]:

```python
#Evaluation Function

def evalTestData():

    runningError=0
    numBatches=0

    for i in range(0,10000,batchSize):

        miniBatchData =  testData[i:i+batchSize].unsqueeze(dim=1)
        miniBatchLabel= testLabel[i:i+batchSize]

        miniBatchData=miniBatchData.to(device)
        miniBatchLabel=miniBatchLabel.to(device)

        inputs = (miniBatchData - mean)/std

        scores=net( inputs )

        error = calcError( scores , miniBatchLabel)

        runningError += error.item()

        numBatches+=1


    totalError = runningError/numBatches
    print( 'error rate on test set =', totalError*100 ,'percent')
```

In [ ]:

```
#Training Loop

start=time.time()

for epoch in range(1,epochs):

    if not epoch%5:
        learningRate = learningRate / 2

    #Optimizer Hyperparameter
    optimizer=torch.optim.SGD( net.parameters() , lr=learningRate )

    runningLoss=0
    runningError=0
    numBatches=0

    shuffledIndices=torch.randperm(60000)

    for count in range(0,60000,batchSize):

        optimizer.zero_grad()

        indices=shuffledIndices[count:count+batchSize]
        miniBatchData =  trainData[indices].unsqueeze(dim=1)
        miniBatchLabel=  trainLabel[indices]

        miniBatchData=miniBatchData.to(device)
        miniBatchLabel=miniBatchLabel.to(device)

        inputs = (miniBatchData - mean)/std

        inputs.requires_grad_()

        scores=net( inputs )

        loss =  lossFunction( scores , miniBatchLabel)

        loss.backward()
        optimizer.step()


        # Calculate Error

        runningLoss += loss.detach().item()

        error = calcError( scores.detach() , miniBatchLabel)
        runningError += error.item()

        numBatches+=1


    # Display average error
    totalLoss = runningLoss/numBatches
    totalError = runningError/numBatches
    elapsed = (time.time()-start)/60

    print('epoch=',epoch, '\t time=', elapsed,'min', '\t lr=', learningRate  ,'\t loss=', totalLoss , '\t error=',
          totalError*100 ,'percent')
    evalTestData()
    print(' ')

```