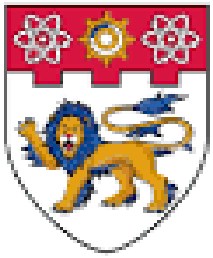


NANYANG TECHNOLOGICAL UNIVERSITY
SINGAPORE

Master in Science (Artificial Intelligence)
(MSAI)

AI-6121

Final project

Name of Student:
Teo Lim Fong

Matriculation No: G2101964G

Table of Content

Contents

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. For the source codes, you can either append them at the end of the project report, or submit them separately.

Question 1

The below code is reference from **[1].**

Step 1: The first step is to import all the necessary library.

```
import torch
import torchvision
import torchvision.datasets as datasets
import torchvision.transforms as transforms
import matplotlib.pyplot as plt
import torch.nn as nn
import torch.optim as optim
import torch.nn.functional as F
import numpy as np
```

Step 2: The main focus of transform_train and transform_test is to convert the incoming datasets to tensor form and apply some augmentation such as random cropping to prevent overfitting. The augmentation only applied to the training datasets. Next, we need to normalize the datasets to reduce the size of the datasets going into the network. Since the datasets is grayscale which contained only one channel, we only have one channel to normalize.
Pytorch has a function (torchvision.datasets…) that can download common datasets such as cifar10, mnist , etc. Therefore, I used the pytorch function to download both mnist training and testing datasets. To check if the number of datasets are correct, I print the value for training and testing datasets and the number are 60k and 10k respectively.
Next, we need to load the datasets into the loader. The loader will only use the amount of images in the batch size to compute the predicition. It is very expensive to compute all the datasets for each epoch in term of the mathematical formula, calculating the chain function in all layers and applying backproppagation in nerual network.

```
transform_train = transforms.Compose([
    transforms.RandomCrop(28, padding=2),
    transforms.ToTensor(),
    transforms.Normalize((0.1307,), (0.3081,))])

transform_test = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.1307,), (0.3081,))])


mnist_trainset = datasets.MNIST(root='C:/Users/LimFong/Desktop/MSAI/computer vision/', train=True, download=True, transform=trans
mnist_testset = datasets.MNIST(root='C:/Users/LimFong/Desktop/MSAI/computer vision/', train=False, download=True, transform=trans


trainloader = torch.utils.data.DataLoader(mnist_trainset, batch_size=128, shuffle=True, num_workers=0)
testloader = torch.utils.data.DataLoader(mnist_testset, batch_size=98, shuffle=False, num_workers=2)

print('Number of training datasets: ', len(mnist_trainset))
print('Number of testing datasets: ',len(mnist_testset))
```

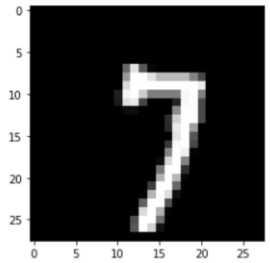
```
Number of training datasets:  60000
Number of testing datasets:  10000
```

Step 3: This step is to ensure that we load the correct datasets and the correct sizes. The correct sizes for mnist datasets is 28x28x1. In order to do this, we need to print the size of the images from the trainloader. The torch.sizes below shown 4 diminesions - (128,1,28,28) which also mean the number of batch x number of channels x width of the image x height of the image respectively. In Pytorch, the number of channels is more important than the width and height of the images. In nerual network, it is complusory to replace the channel by the number of fliter. However, for height and width, it is optional because if we do not apply max pooling, it will still be the same. Therefore, the position of the image size is swaped. Then, we use plt.imshow to show an image (7) randomly from the trainloader.

```
images, labels = next(iter(trainloader))
print(images.shape)
plt.imshow(images[0].reshape(28,28), cmap="gray")
```

```
torch.Size([128, 1, 28, 28])
```

```
<matplotlib.image.AxesImage at 0x1e293b6a460>
```



Step 4: This is a small neural network that only contained two convolutional layers, two max pooling layers and two fully connected layers.
I chose CNN instead of MLP because CNN extract feature maps from the image and reduce the parameters significantly.
The function of Conv2d is given:

nn.Conv2d (input channels, output channels, number of kernel, number of padding)

In the first convolutional layer, since it is grayscale, the input channel is 1, the output channel is given 25 and the kernel size is 3 with padding 1. Then, I applied max pooling of 2 to reduce the size of the images. The purpose of max pooling is to find the maximum value from the feature maps and eliminate the rest. There are also other pooling such as mean pooling. It is to find the mean in the window instead of finding the maximum value. In this case, we are using max pooling because mean pooling does not extract image structure such as edges.
In the second convolutional layer, I further increase number of output channel to 100 and repeat the same procedure by applying max pooling.
The next step is to connect convolutional layers into fully connected layers. The fully connected layers is also known as MLP which flatten the feature maps in the convolutional layers to one dimension vector. In the first fully connected layers, we reduce the 4900 x 1 vector to 100 x1 and since MNIST contains10 classes, it is further reduced it to 10 x1.

The summary dimension of the convolutional network:
First convolutional layers: 1 x 28 x 28 to **25 x 28 x 28**
First max pooling layers: 25 x 28 x 28 to **25 x 14 x 14**
Second convolutional layers: 25 x 14 x 14 to **100 x 14 x 14**
Second max pooling layers: 100 x 14 x 14 to **100 x 7 x 7**
First Fully Connected layers**: (100 * 7 * 7) x 1 to 100 x 1**
Second Fully Connected layers: 100 x 1 to **10 x 1**

```
class CNN(nn.Module):

    def __init__(self):

        super(CNN, self).__init__()


        self.conv_layer1 = nn.Conv2d(1,   25,  kernel_size=3,  padding=1 )
        self.pool_layer1  = nn.MaxPool2d(2,2)

        self.conv_layer2 = nn.Conv2d(25,  100,  kernel_size=3,  padding=1 )
        self.pool_layer2 = nn.MaxPool2d(2,2)

        self.linear_layer1 = nn.Linear(4900, 100)
        self.linear_layer2 = nn.Linear(100,10)


    def forward(self, x):


        x = self.conv_layer1(x)
        x = F.relu(x)
        x = self.pool_layer1(x)

        x = self.conv_layer2(x)
        x = F.relu(x)
        x = self.pool_layer2(x)


        x = x.view(-1, 4900)
        x = self.linear_layer1(x)
        x = F.relu(x)

        x = self.linear_layer2(x)

        return x
```

Step 5: Training function is defined in this step. First, initialize the training loss, correct predictions and the total number of labels to zero. Then, create a for loop to loop the training datasets for both images and labels into the network using GPU. The loss function, in general can be calculated by comparing the output and the ground truth. Then, we sum the training loss.
In theory, predicted output can be obtained from MLP or to be more precise the softmax layers. In the softmax, one output can have multiple predictions. The one with the highest probability is chosen. Next in order to find the accuracy, we sum up the correct prediction and divide it by the total number of labels. Lastly, I set 50 iterations for one epoch.

```
def train(epoch, net, criterion, trainloader, scheduler):

    print('\nEpoches: %d' % epoch)
    iteration = 50
    net.train()
    train_loss = 0
    correct = 0
    total = 0

    for batch_index, (inputs, target) in enumerate(trainloader):
        inputs, target = inputs.to('cuda'), target.to('cuda')
        outputs = net(inputs)
        optimizer.zero_grad()

        loss = criterion(outputs, target)
        loss.backward()
        optimizer.step()

        train_loss += loss.item()
        _, predicted = outputs.max(1)
        total += target.size(0)

        correct += (predicted == target).sum().item()
        loss_per_iteration = train_loss/ (batch_index+1)
        loss_per_iteration = np.round(loss_per_iteration, 3)

        accuracy = 100 *correct/total
        accuracy = np.round(accuracy,2)

        if (batch_index +1) % iteration == 0:
            print("Number of iteration : %3d, training loss : %0.4f, accuracy : %2.2f" % (batch_index+1, loss_per_iteration , acc

    scheduler.step()
    return loss_per_iteration, accuracy
```

Step 6: Testing function is defined in this step. It is similar to step 5. The only difference is that testing function does not include any optimizer.
The purpose of having testing datasets is to test our model with similar datasets that never seen before. Even though for some cases, the training loss and accuracy can be optimal however when the model is being evaluated by testing datasets, the accuracy drop dramatically. The reason to that is that the model is most likely overfitted. This mean that the model either train for too long that it memorized the training datasets or the training datasets is very different from the testing datasets.

```
def test(epoch, net, criterion, testloader):
    net.eval()

    test_loss = 0
    correct = 0
    total = 0
    with torch.no_grad():
        for batch_index, (inputs, target) in enumerate(testloader):
            inputs, target = inputs.to('cuda'), target.to('cuda')
            outputs = net(inputs)
            loss = criterion(outputs, target)

            test_loss += loss.item()
            _, predicted = outputs.max(1)
            total += target.size(0)

            correct += (predicted == target).sum().item()
            loss_per_iteration = test_loss/ (batch_index+1)
            accuracy = 100 *correct/total

    return loss_per_iteration, accuracy
```

Step 7: This step is to adjust the parameters such as the learning rate, number of epochs and weight decay and import the network into GPU. For default setting, I used cross entropy as the loss function and the optimizer as SGD (Stochastic Gradient Descent).

```
lr = 0.01
momentum =0.9
wd = 0
epochs = 200

net = CNN().to('cuda')
criterion = nn.CrossEntropyLoss().to('cuda')
optimizer = optim.SGD(net.parameters(), lr= lr,momentum= momentum, weight_decay=wd)
```

Step 8: Finally, we prepare to train a model by creating a for loop. To analysis the performance of the losses and accuracy better, I decided to plot a graph by creating four different lists - training loss, testing loss, training accuracy and testing accuracy and append them into the lists for every epoch.

```
train_loss_list = []
train_acc_list  = []
test_loss_list  = []
test_acc_list   = []

for epoch in range(0, epochs):
    training_loss, train_acc = train(epoch, net, criterion, trainloader, scheduler)
    testing_loss, test_acc = test(epoch, net, criterion, testloader)

    print(("Epoch : %3d, training loss : %0.4f, training accuracy : %2.2f, test loss " + \
      ": %0.4f, test accuracy : %2.2f") % (epoch, training_loss, train_acc, testing_loss, test_acc))

    train_loss_list.append(training_loss)
    test_loss_list.append(testing_loss)
    train_acc_list.append(train_acc)
    test_acc_list.append(test_acc)
```

2) Investigate different hyper-parameters and design options such as learning rate, optimizers, loss functions, etc. to study how they affect the network performance. The report should include detailed description and discussion of your study.

Question 2

Learning rate

Learning rate is one of the key parameters that can optimize the network. Our aim is to reach the lowest point of the curve. However, it is hard to reach the lowest point. If the learning rate is too fast, it will not learn anything and miss the lowest point which lead to underfitting. If the learning rate is too slow, it will take very long to converge. Therefore, I decided to test on three different learning rates – 0.1, 0.01, 0.001 with 100 epochs as default.
The below training parameters does not include any weight decay and learning rate scheduler.

	No. of epochs	Learning rate		
		0.001	0.01	0.1
Training loss	100	0.0168	0.0028	0.0508
Testing loss	100	0.0603	0.0245	0.0472
Training accuracy	100	99.59	99.90	99.42
Testing accuracy	100	98.95	99.37	99.00

Table 1: Comparing the losses and accuracy using different learning rate

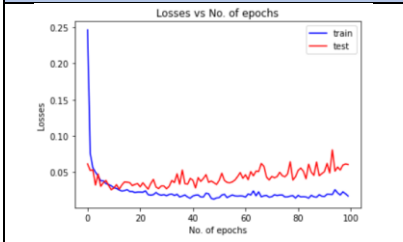
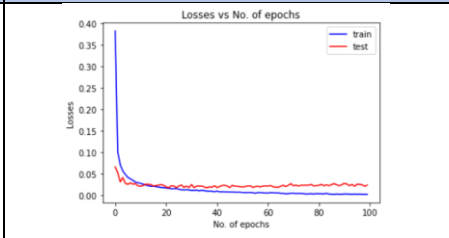
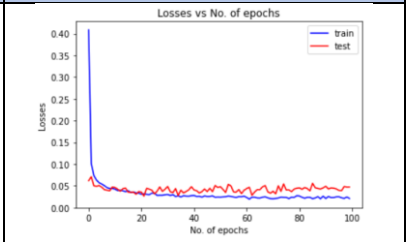
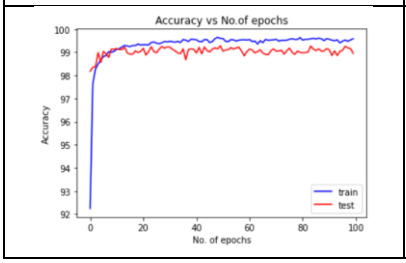
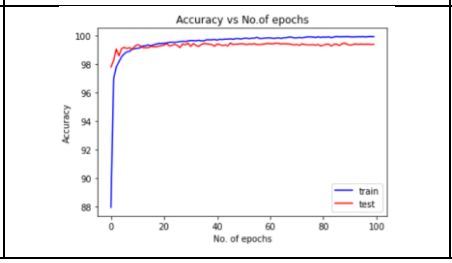
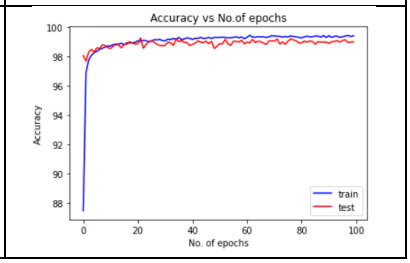
Learning rate		
0.001	0.01	0.1
		
		

Table 2: Diagram of the losses and accuracy for different learning rate at 100 epochs

From the table, we can see that the learning rate for 0.01 has the lowest training loss compared to the other two. For learning rate 0.001, the training loss is relatively low but there is overfitting in testing datasets. Therefore, I decided to choose learning rate 0.01.

No. of Epochs

We have discovered the optimal learning rate in the previous section. Now, we need to fine-tuned other parameters to optimize the network. Since we only trained the network with 100 epochs, is it possible for the network to be optimized if we train it longer.
I decided to experiment with 200 epochs with the three different learning rates.

	No. of epochs	Learning rate		
		0.001	0.01	0.1
Training loss	200	0.0084	0.0008	0.0519
Testing loss	200	0.0188	0.0305	0.0199
Training accuracy	200	99.75	99.95	99.08
Testing accuracy	200	99.27	99.43	98.58

Table 3: Comparing the losses and accuracy using different learning rate at 200 epochs

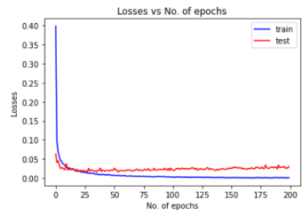
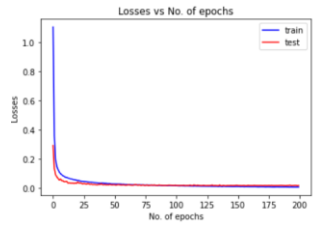
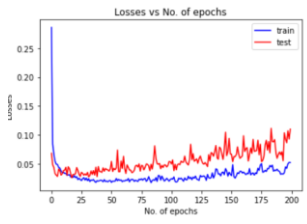
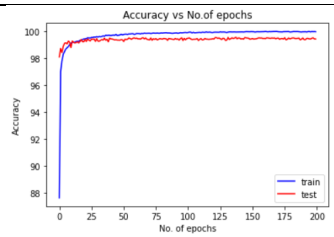
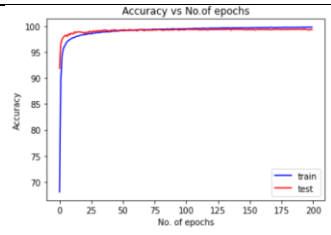
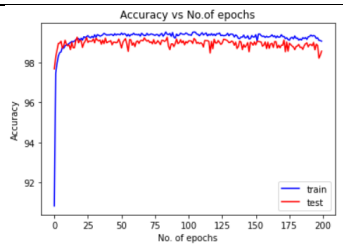
Learning rate @200 epochs		
0.001	**0.01**	**0.1**
		
		

Table 4: Diagram of the losses and accuracy for different learning rate at 200 epochs

The training loss reduced even further for learning rate at 0.01 from previous 0.0028 to current 0.0008. Learning rate 0.1 is a very good example of underfitting. The training loss does not coverage instead the losses increase. The reason is due to the large learning rate.

Different optimizers

The purpose of optimizers is to reduce the weight and the learning rate of the network during training so as to prevent skipping the optimal point. There are many optimizers available and the default optimizers I have been using is the SGD (Stochastic Gradient Descent). I will be testing other optimizers such as (Adaptive Moment Estimation) ADAM and ADAMW to compared the results.

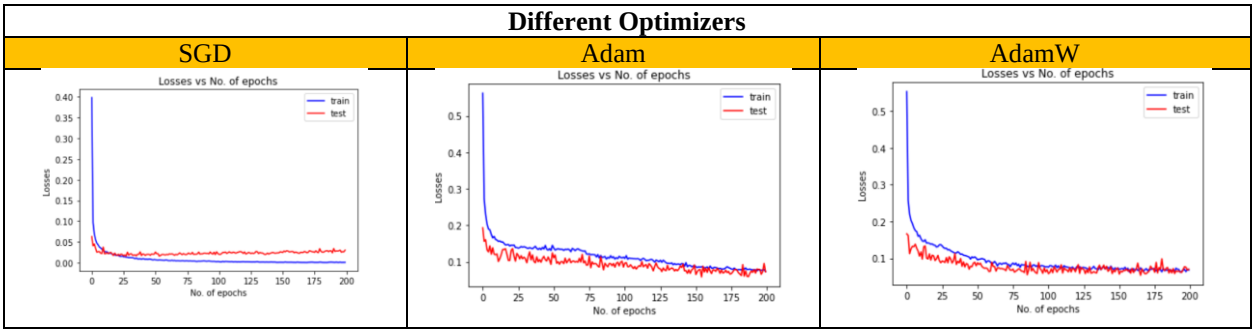
The other optimizers, ADAM **[2]** is the combination of two optimizers, momentum and root mean square propagation (RMSPro) **[3].** Momentum is an algorithm that accelerate rapidly toward the local minima. It also keeps updating the parameters to avoid being stuck at the local minimum and multiplied that with the previous updates. RMSPro is similar to SGD with momentum. It restricts the unstable oscillations toward the minima. Therefore, the learning rate can be large when converging. With the two combinations of optimizers, ADAM can provide an optimization that can solve issue such as sparse gradient on noisy problems.

ADAMW is the improved version of ADAM. The difference between ADAMW and ADAM is the different calculation for weight decay. For ADAM, the weight decay is added into the cost function to calculate the gradient which later found out that the regularization ends up in moving averages of the gradient. For ADAMW, weight decay is added after performing the learning rate or they called it parameter-wise step size. This improvement will prevent the regularization from ending up in the moving average.

However, the below experiments will different optimizers with not include any weight decay parameters.

			Different Optimizers		
	No. of epochs	Learning rate	SGD (default)	Adam	AdamW
Training loss	200	0.01	0.0008	0.0735	0.0680
Testing loss	200	0.01	0.0305	0.0727	0.0706
Training accuracy	200	0.01	99.95	97.84	98.05
Testing accuracy	200	0.01	99.43	97.81	98.02

Table 5: Comparing the losses and accuracy using different optimizers



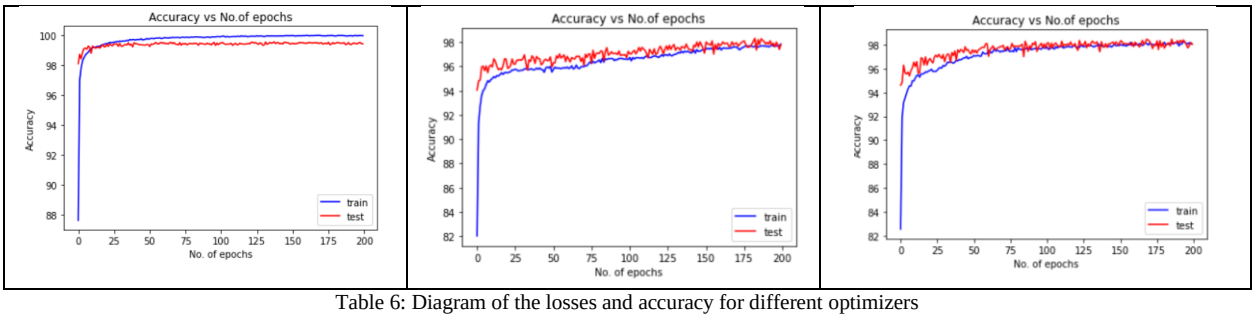
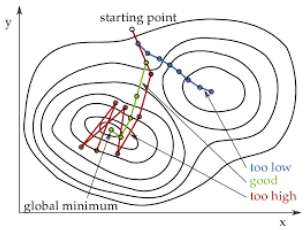


Table 6: Diagram of the losses and accuracy for different optimizers

Learning rate scheduler

Learning rate is an important factor to optimize the network. The below diagram shown an example of different learning rates going at the wrong direction instead of the minimum point.



A constant learning rate is more likely to miss the optimal point. However, is it possible if we adjust the learning rate such that it moves fast for the first quarter of the training and start to slow down when reaching the end of the training. This method is called the learning rate scheduler.

There are many learning rate schedulers such as cosine annealing, step learning rate and many more. In this section I will demonstrate these two learning rates to see if it improves the training losses.

The formula for Cosine Annealing **[4]** as shown below uses the concept of cosine curve to decrease the learning rate when approaching the minima. The cosine curve used in Cosine Annealing is shifted up by 1 to avoid any negative value. At the start of the training, when the epoch is 0, the learning rate for Cosine Annealing is the initial learning rate η_{max} set by the user and it will
decrease the learning rate accordingly for each epoch till the current epoch (T_{cut}) reach the end of the training (T_{max}).

$$\eta_t = \eta_{min} + \frac{1}{2}(\eta_{max} - \eta_{min})\left(1 + \cos\left(\frac{T_{cur}}{T_{max}}\pi\right)\right)$$

Next is the step learning rate scheduler. Step learning rate has two main parameters. The first parameter is the step size and the second parameter is the gamma value. Step size determine how much the learning rate to be decayed and the gamma value control the multiplicative factor of learning rate decay. For this experiment, I set step size to be 50 and the gamma value as default.

			Learning rate scheduler		
	No. of epochs	Learning rate	Constant	Cosine	Step LR
Training loss	200	0.01	0.0008	0.0002	0.00023
Testing loss	200	0.01	0.0305	0.0230	0.0280
Training accuracy	200	0.01	99.95	100.00	99.95
Testing accuracy	200	0.01	99.43	99.48	99.44

Table 7: Comparing the losses and accuracy using different learning rate scheduler

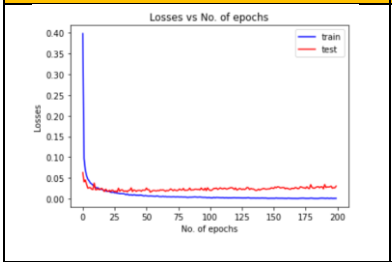
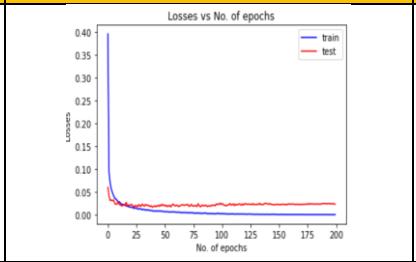
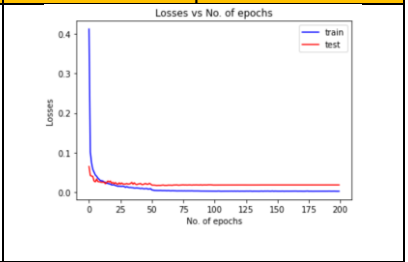
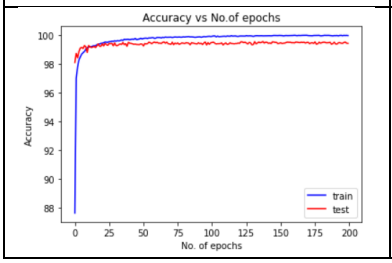
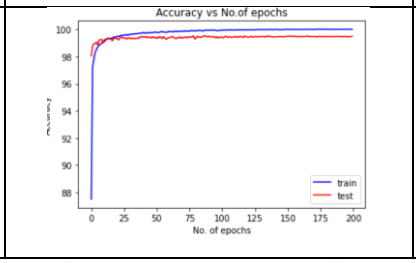
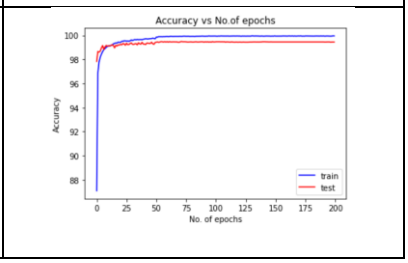
Different learning rate scheduler		
Constant	Cosine	Step LR
		
		

Table 8: Diagram of the losses and accuracy for different learning rate scheduler

From the table, we can see that by decaying the learning rate improve the optimization. The training loss reduce from 0.0008 to 0.0002 for cosine annealing and 0.00023 for step learning rate. The training accuracy improve for cosine annealing to 100%. Therefore, cosine annealing is the best learning rate scheduler among the other two.

However, we still can see there is still a bit of overfitting issue occurring at the testing datasets. In order to solve the overfitting issue, we can add regularization such as L2 or weight decay to give some bias to the training datasets so as to ‘fit’ the testing datasets.

3) Benchmark with the state-of-the-art, and discuss the constraint of your designed network and possible improvements. Implement and verify your ideas over the test set. This subtask is **optional**, and there will be bonus marks for nice addressing of this subtask.

Question 3

Benchmark with state-of-the-art

Below are a few of the benchmark for MNSIT.

Benchmark	Testing accuracy
Merging CNN + Homogeneous Vector [5]	99.87
EnsNet [6]	99.84
Efficient-CapsNet [7]	99.84
Current	99.48

Table 9: List of state-of-the-art benchmarks for MNIST

Convolutional Neural Network has a lot of parameters to fine-tuned. In the second section, we adjusted the learning rates, the number of epochs, different learning rate scheduler and different optimizers. All these adjustments will not prevent overfitting issue. And in some scenario, vanishing and exploding gradient also affect the training process. Therefore, in this section I will focus on a few methods that can prevent overfitting and solution to vanishing gradient.

Methods for Regularization

In neural network, there are two important factors that can improve the training and testing process. For training, it is called optimization and for testing it is called regularization. I will be using weight decay and data augmentation for

Weight Decay

Weight Decay is one of the regularization methods to prevent overfitting issues in neural network. The main reason why overfitting occurred is because the model is too dependent on the training datasets rather than learning the feature maps to identify similar datasets. We can see that the previous result, even though the training loss is 0, there is still overfitting. Therefore, I decided to add weight decay into SGD optimizers. I will be trying three different values, 0.1, 0.01 and 0.0005.

			Weight decay		
	No. of epochs	Learning rate	0.1	0.01	0.0005
Training loss	200	0.01	0.3245	0.0624	0.0055
Testing loss	200	0.01	0.2219	0.0406	0.0155
Training accuracy	200	0.01	92.65	98.41	99.89
Testing accuracy	200	0.01	95.13	98.90	99.49

Table10: Comparing the losses and accuracy using different weight decay

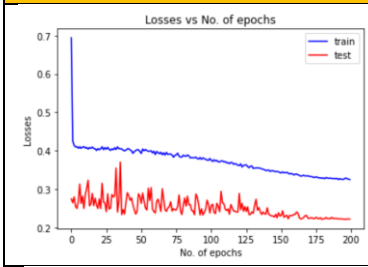
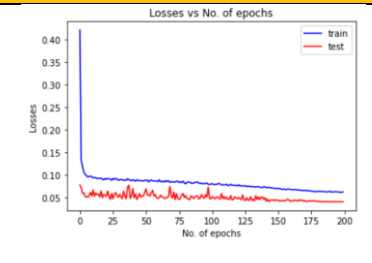
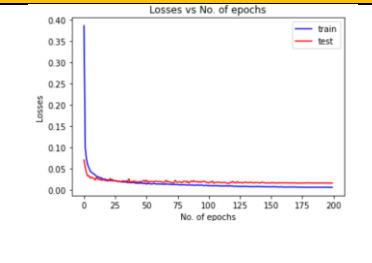
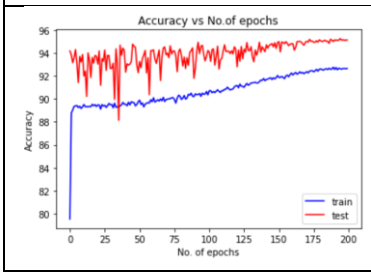
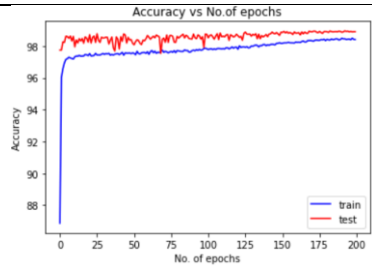
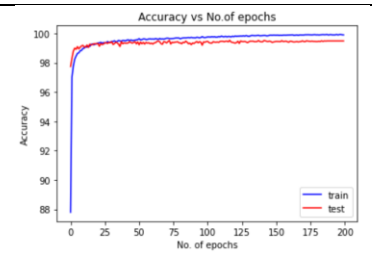
Weight Decay using SGD optimizers		
0.1	0.01	0.0005
		
		

Table11: Diagram of the losses and accuracy for different weight decay

We can see that when the weight decay is large, the model cannot train well. The bias added is too large in the training datasets. However, for weight decay 0.0005, the training and testing loss nearly converge to the optimal point. From my understanding, with additional of weight decay, the training loss will not coverage to 0 because of the biases. Therefore, weight decay 0.0005 is the best among the rest.

I decided to add weight decay on the other optimizers that I have experimented earlier and compared with SGD.

Different optimizers with weight decay

Previously, I experimented ADAM and ADAMW optimizers without weight decay. Since I have added weight decay to SGD, why not do the same for the other two optimizers.

			Weight decay = 0.0005		
	No. of epochs	Learning rate	SGD	Adam	AdamW
Training loss	200	0.01	0.0055	0.0291	0.0110
Testing loss	200	0.01	0.0155	0.0271	0.0279
Training accuracy	200	0.01	99.89	99.11	99.64
Testing accuracy	200	0.01	99.49	99.13	99.23

Table 12: Comparing the losses and accuracy using weight decay with different optimizers

Different optimizers with weight decay

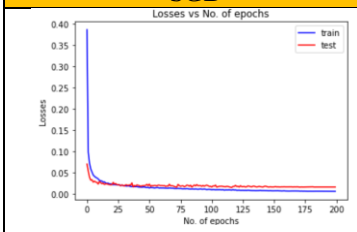
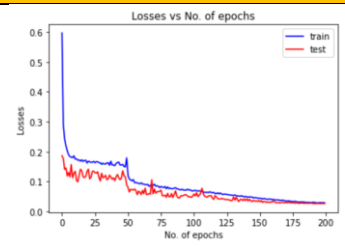
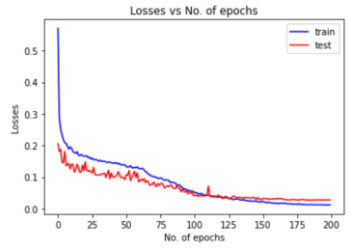
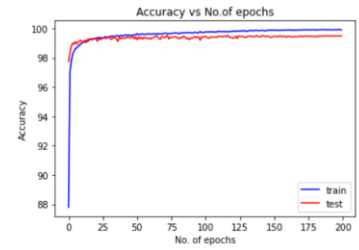
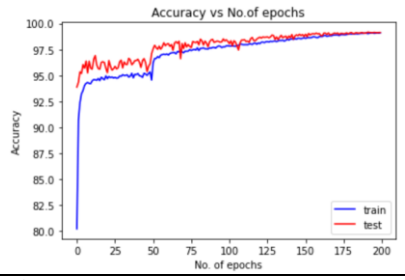
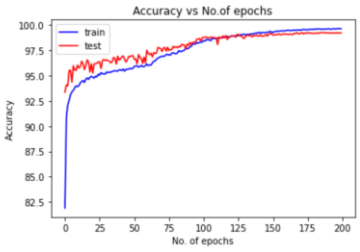
SGD	ADAM	ADAMW
		
		

Table 13: Diagram of the losses and accuracy using weight decay with different optimizers

From the table, we can see that by adding weight decay into the optimizers greatly enhance the overfitting issues. ADAM has an improved overall stat compared to the previous result (without weight decay). The overfitting issue has been solved for both ADAM and ADAMW. However, SGD is still the optimal optimizers for this case.

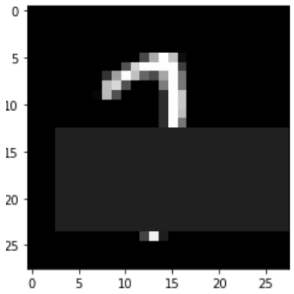
Augmentation

Augmentation is a very good technique that can increase the number of training datasets. It manipulates the datasets in different orientation to prevent the network from memorizing certain feature maps from the datasets. As such, the network will learn more features in the datasets. Thus, resulting a better model to identify similar datasets.

The augmentation that commonly used are vertical flipping, horizontal flipping, random cropping, rotation and many more. However, not all testing datasets will be visible. Some might have defect or have blurry resolution. Therefore, to make the model more robust, we need to let the model learn the individual features. In order to do that, we add random occlusion in the training datasets. The size of the occlusion can be adjusted. There is one function in pytorch that allowed such function which is called the random erasing. [**8**]

```
transform_train = transforms.Compose([
    transforms.RandomCrop(28, padding=2),
    transforms.ToTensor(),
    transforms.Normalize((0.1307,), (0.3081,)),
    transforms.RandomErasing(p=0.5, scale=(0.02, 0.4), ratio=(0.3, 3.3))
])
```

I just add one line of code under this section. Random Erase also have a few parameters to adjust such as the probability of the rectangle appearing, the size of the rectangle and also the colour of the rectangle. I will be using the default setting for the size and the colour of the rectangle. However, it will be testing different probability mainly 0.2, 0.5 and 0.9 in the training.



The figure above shown an example of datasets after applying random erase.

			Probability of occlusion that appear in datasets		
	No. of epochs	Learning rate	0.2	0.5	0.9
Training loss	200	0.01	0.0434	0.0851	0.164
Testing loss	200	0.01	0.0124	0.0126	0.0041
Training accuracy	200	0.01	98.52	97.09	94.08
Testing accuracy	200	0.01	99.51	99.54	99.42

Table 14: Comparing the losses and accuracy using different probability of occlusion that appear in datasets

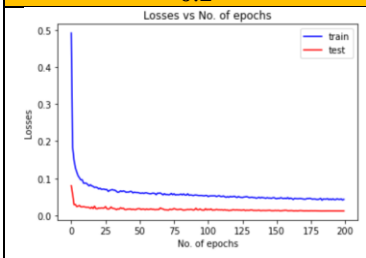
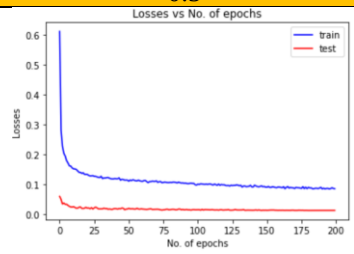
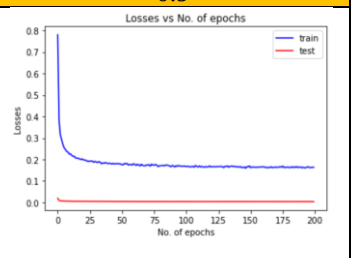
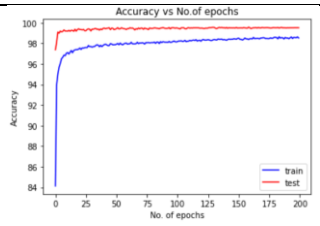
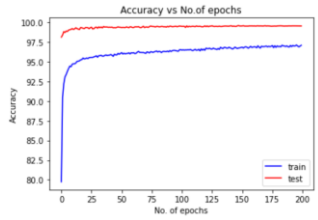
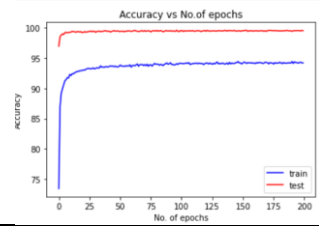
Probability of occlusion that appear in datasets		
0.2	0.5	0.9
		
		

Table 15: Diagram of the losses and accuracy for different probability of occlusion that appear in datasets

The number for the probability cannot be too high such as 0.9 or even 1.0 because the training datasets will be too different from the testing datasets. The article **[8]** proposed 0.5 probability as the ideal parameters in random erasing. Indeed, from the table, 0,5 is the best for testing accuracy. Therefore, 0.5 is the best.

Batch normalization

Batch normalization **[9]** can be considered as one of the regularization methods since it helped to solve the exploding gradient issue. Exploding gradient is opposite of vanishing gradient that contain large weight in the neutron and this will make the model become unstable and untrainable.

We normalized training datasets before training to ensure all datasets are in the same scale and also to reduce the parameters. However, this is not enough. We need to apply normalization for the activation function and hidden layers too. The first thing we do is to normalize the output from the activation function. After that, the normalized output will be multiple by some parameter **g** and add with another parameter **b**. These **g** and **b** parameters are trainable, thus will become optimized during training.

			Batch Normalization
	No. of epochs	Learning rate	
Training loss	200	0.01	0.1390
Testing loss	200	0.01	0.012
Training accuracy	200	0.01	95.02
Testing accuracy	200	0.01	99.56

Table 16: Diagram of the losses and accuracy for batch normalization

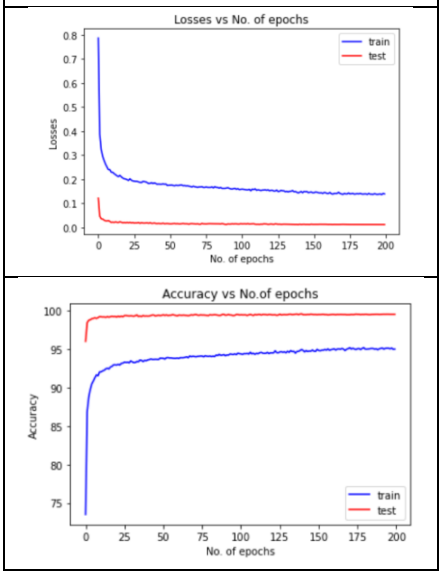
Batch Normalization


Table 17: Diagram of the losses and accuracy for batch normalization

Resnet

Resent was introduced to solve vanishing gradient by adding an identity connection between residual block which is also known as a skip connection in the neural network. Vanishing gradient occurred during the chain rule in backpropagation. When the weights are small and multiple with another smaller number, the output will be even smaller. If this continues, the weight will become very small and eventually vanish. This also mean that it does not learn anything. Resnet-18 contains five convolutional layers.

In the first layer, Resnet-18 will be using the kernel size of X and the number of channels will be replaced. Next, max pooling and stride will be used to down sample the image.

For the subsequent convolutional layers, Resnet-18 will be using kernel size of X and will double the number of filters in each layer. The skip connection is represented as a black arrow and dotted lines represents skip connection with unequal size of the image from the previous layer. In order to balance size of the image, the previous layers need to add a 1D (1x1) convolutional layer. Lastly, Resnet-18 applied an average pooling and flattened it into fully connected layers. [**10**]

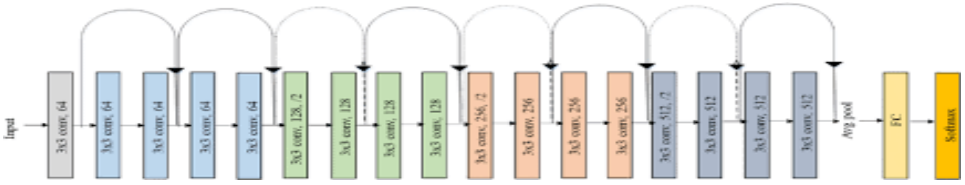


Figure ?: Example of Resnet-18 architecture

Resnet 18 and 34 is used to compare with my current network. Pytorch has a pre-defined library that stored the model and the network of Resnet. However, Resnet is created for Imagnet which contain 3 channels, RGB have output of 1000. With the current dataset, we need to modify the first layer and the last layer such that the input takes in one channel and the last layers contain 10 classes.

I create a class to download the pre-trained model and modify the first and last layer of the ResNet. Then continue from Step 5 to train the model using my existing training code.

```
import torchvision.models as models

class MnistNet(nn.Module):
    def __init__(self, in_channels=1):
        super(MnistNet, self).__init__()

        self.model = models.resnet18(pretrained=True)
        self.model.conv1 = nn.Conv2d(in_channels, 64, kernel_size=3, stride=1, padding=1)
        linear_last = self.model.fc.in_features
        self.model.fc = nn.Linear(linear_last, 10)

    def forward(self, x):
        return self.model(x)
```

```
my_resnet = MnistNet()
```

```
input = torch.randn((128,1,28,28))
output = my_resnet(input)
```

			Different Network		
	No. of epochs	Learning rate	Mine	Resnet 18	Resnet 34
Training loss	200	0.01	0.1390	0.0860	0.0770
Testing loss	200	0.01	0.012	0.00087	0.0098
Training accuracy	200	0.01	95.02	96.89	97.16
Testing accuracy	200	0.01	99.56	99.74	99.70

Table 18: Diagram of the losses and accuracy for different backbone

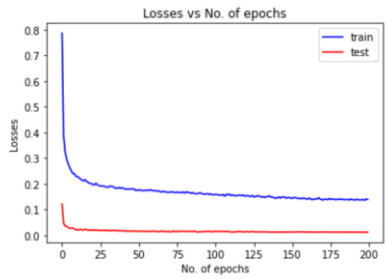
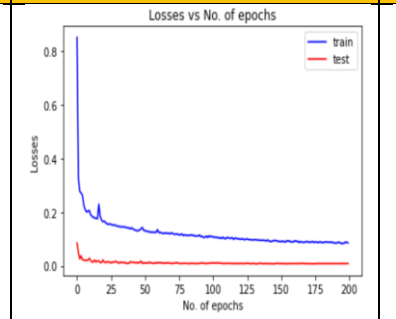
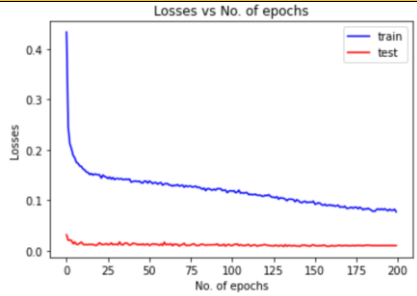
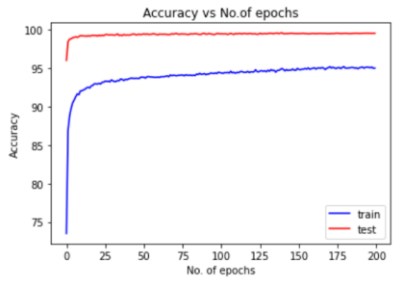
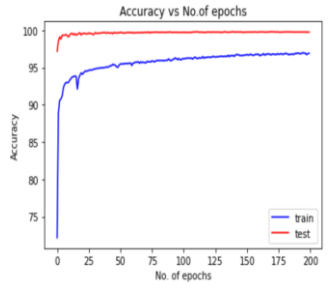
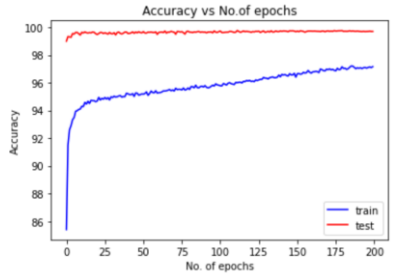
Different Network		
Mine	Resnet 18	Resnet 34
		
		

Table 19: Diagram of the losses and accuracy for different backbone

Resnet-18 perform better than my existing SGD. The reasons are probably because Renset-18 contains more layers and it skips layers to avoid any vanishing gradient.

Conclusion

In this assignment, we understand that it is time consuming to optimise the neural network. There are too many parameters that can affect the output of the neural network such as the learning rates, number of epochs and many others. The learning rate is one of the major parameters that optimise the network to reach the optimal point. However, we see that constant learning rate is not optimal to reach the lowest point. Therefore, we introduce learning rate scheduler to slower the learning rate when coming to the end of the training. Another issue that occurs very common in neural network is the overfitting issue. We use Regularization method to solve overfitting such as adding weight decay, augmentation with occlusion and batch normalization. Indeed, from the graph we can see that by adding the above factors improved the performance of the training and testing datasets. Lastly, we also tried ResNet just to compare the difference between mine. ResNet consists of 18 and 34 layers with skip connection included while ours is just 2 layers. Thus, my conclusion is that the more the number of layers the better the results. However, this is just to the extent of 34 layers not 152 layers because I understand that too many layers will also degrade the results.

Reference

[1] https://pytorch.org/tutorials/beginner/blitz/cifar10_tutorial.html

[2] Diederik P. Kingma, Jimmy Lei Ba. ADAM: A METHOD FOR STOCHASTIC OPTIMIZATION. 2015. ICLR

[3] Geoffrey Hinton, Ni@sh Srivastava, Kevin Swersky. Overview of mini-batch gradient descent.

[4] Ilya Loshchilov & Frank Hutter. SGDR: STOCHASTIC GRADIENT DESCENT WITH WARM RESTARTS. 2017. ICLR

[5] Adam Byerly, Tatiana Kalganova, Ian Dear. NO ROUTING NEEDED BETWEEN CAPSULES. 2021.

[6] Daiki Hirata, Norikazu Takahashi. ENSEMBLE LEARNING IN CNN AUGMENTED WITH FULLY CONNECTED SUBNETWORKS. 2020.

[7] Vittorio Mazzia , Francesco Salvetti , Marcello Chiaberge . EFFICIENT-CAPSNET: CAPSULE NETWORK WITH SELF-ATTENTION ROUTING. 2020

[8] Zhun Zhong, Liang Zheng, Guoliang Kang, Shaozi Li, Yi YangRandom. Erasing Data Augmentation. 2020.

[9] Sergey Ioffe, Christian Szegedy. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. 2015

[10] Kaiming He, Xiangyu Zhang, Shaoqing Ren, Jian Sun, Deep Residual Learning for Image Recognition. 2016