ECE4179 Assignment 2

Zhiyue Li

28280016

Zlii0010@student.monash.edu

Q 1. 1

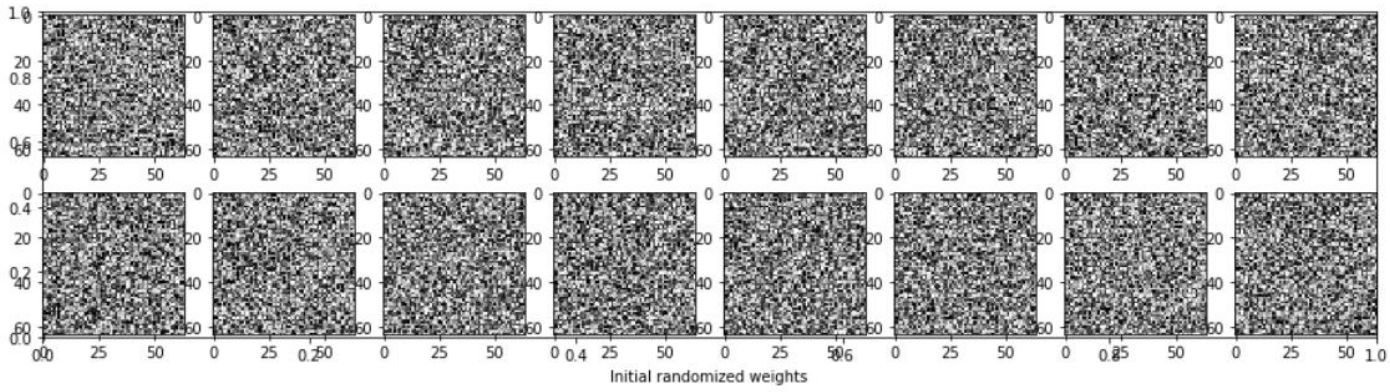
The learning rate is set as 0.01. Optimizer algorithm is SGD. The loss function of the network is the difference between x and predicted x, which gives better results. The next 3 figures are screenshot for the code. Then the next 6 figures are results when n is 16.

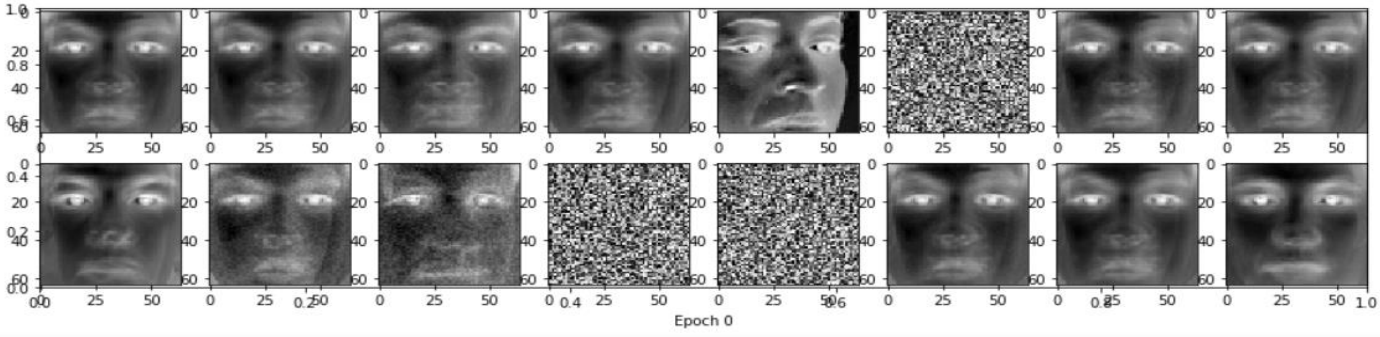
```
n= 16

class MLP(nn.Module):

    def __init__(self):
        super(MLP, self).__init__()

        inputSize= 4096
        hiddenSize = n
        outputSize = 4096
        # mapped to an n dimensional vector
        self.linear1=nn.Linear(inputSize,hiddenSize)
        self.linear2=nn.Linear(hiddenSize,outputSize)

    def forward(self, x):
        # linear layer fc1
        x=self.linear1(x)
        # ReLU nonlinearity
        x=torch.relu(x)
        # Linear layer fc2
        x=self.linear2(x)

        return x
```

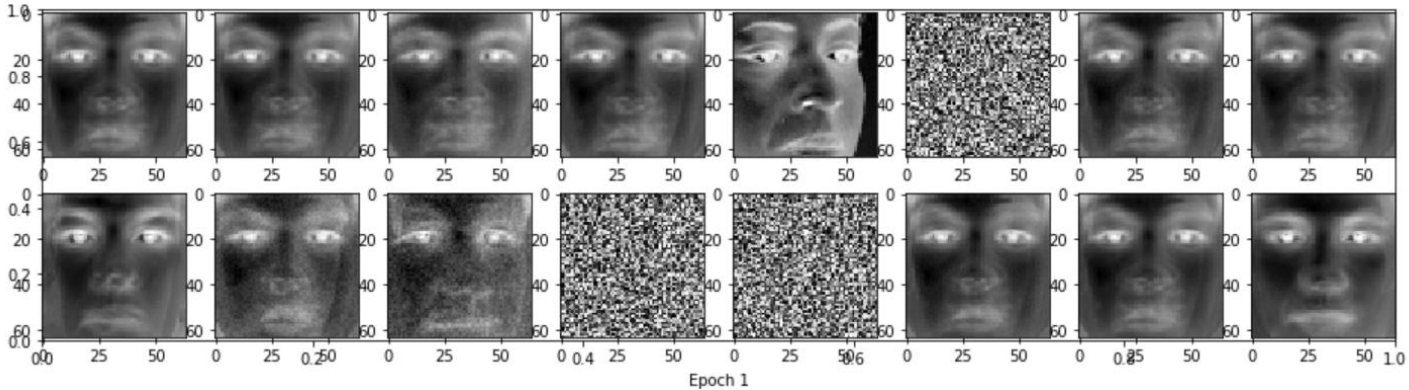
```
def train_model(model, optimizer, dataloader, loss_fn, loss_logger,epochs):

    for iteration in range(epochs):
        for x in dataloader:
            y_predict = model(x)
            loss = loss_fn(y_predict,x)
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()
            loss_logger.append(loss.data.item())

        img_array=list(model.parameters())[0].data.numpy()
        img_array = img_array.reshape([-1,64,64])
        fig = plt.figure(figsize=(16,4))
        plt.xlabel("Epoch"+' ' +str(iteration))

        for img in range(16):
            sub = fig.add_subplot(2,8,img+1)
            sub.imshow(img_array[img],cmap = 'gray')


    return loss_logger
```

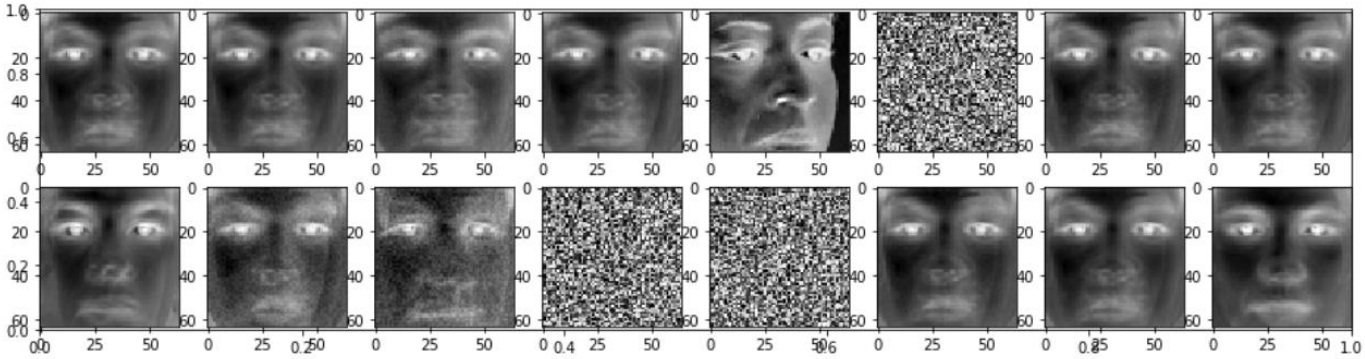
```python
def model(dataset_train):
    learning_rate = 0.01
    model = MLP()
    loss_fn = nn.MSELoss()
    img_array=list(model.parameters())[0].data.numpy()
    img_array = img_array.reshape([-1,64,64])
    fig = plt.figure(figsize=(16,4))
    plt.xlabel("Initial randomized weights")
    for i in range(16):
        sub = fig.add_subplot(2,8,i+1)
        sub.imshow(img_array[i], cmap = 'gray')

    optimizer = optim.SGD(model.parameters(), lr=learning_rate)
    loss_fn = nn.MSELoss(reduction='sum')
    loss = train_model(model=model, optimizer= optimizer, dataloader=trainloader, loss_fn= loss_fn, loss_logger = loss_logger,epo

    return loss

loss_logger = []   # Keep track of the loss values
acc = [] # Keep track of accuracy values
ImageData = FaceData()



losses = model(ImageData)
#Lets visualise an entire batch of images!
fig=plt.figure()
plt.title("n = "+str(n))
plt.plot(range(len(losses)),losses)
plt.xlabel('Batch number')
plt.ylabel('Loss')
```

N = 16

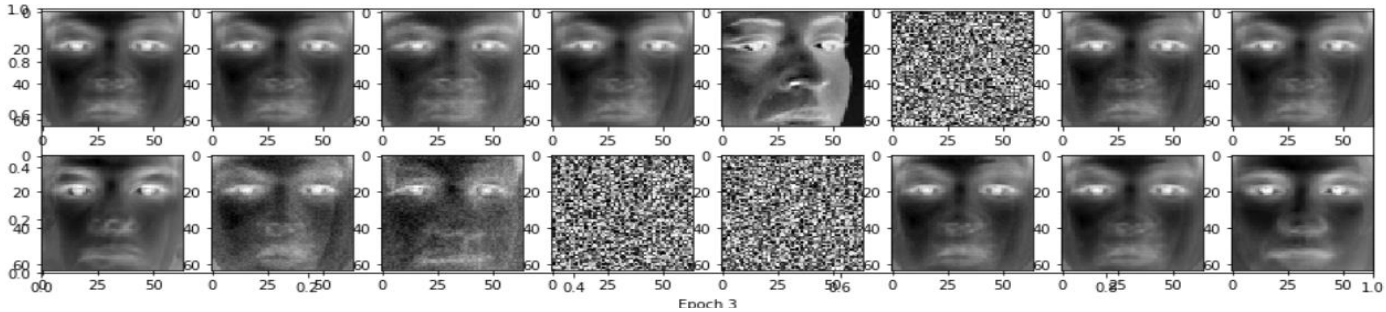




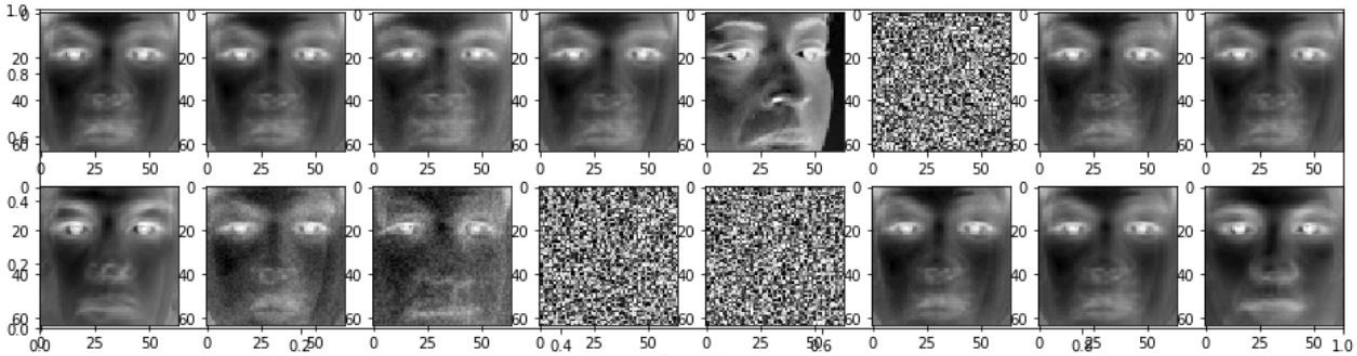




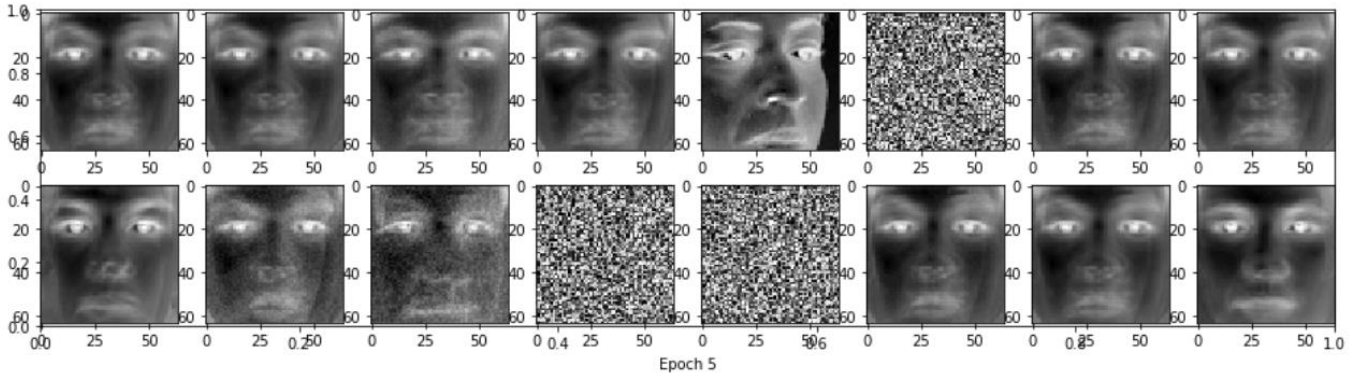
Epoch 2



Epoch 3

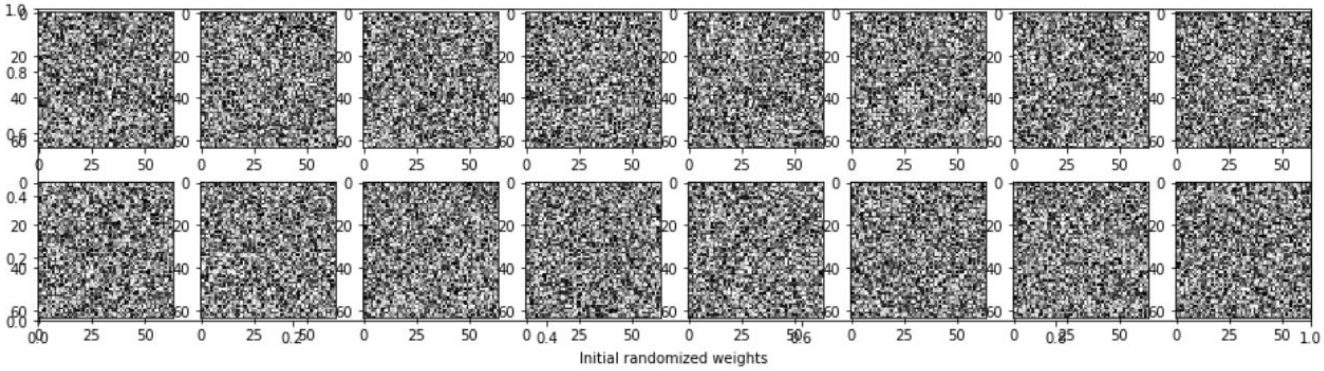


Epoch 4

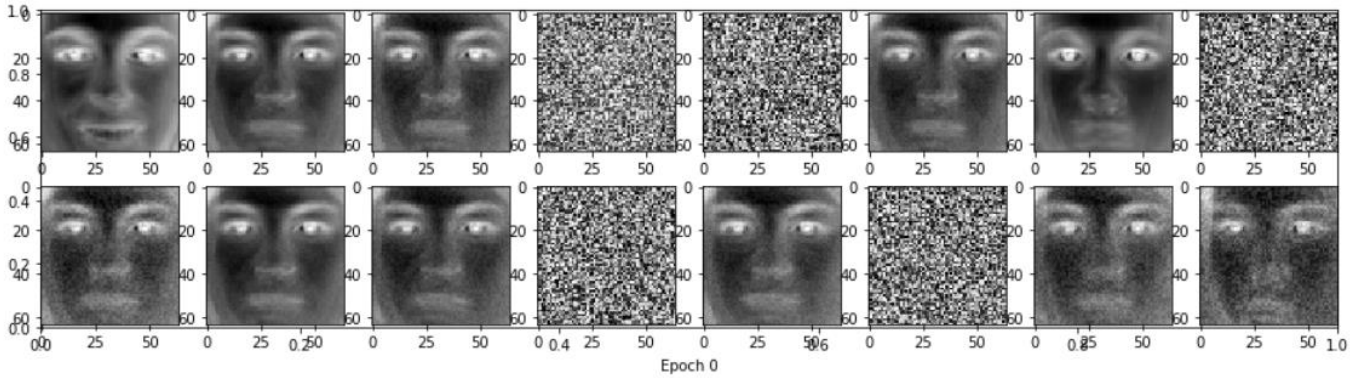


Epoch 5

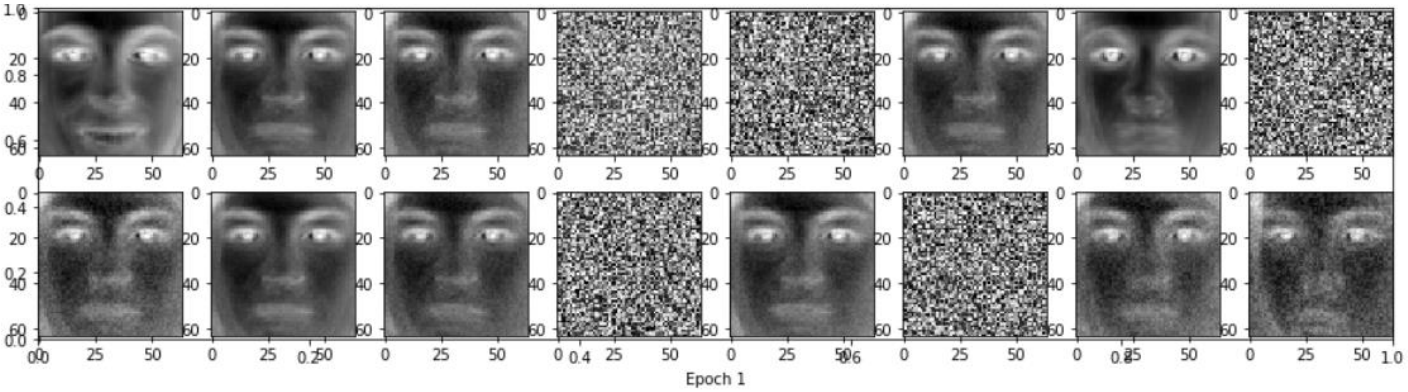
The following results are obtained when N = 64.



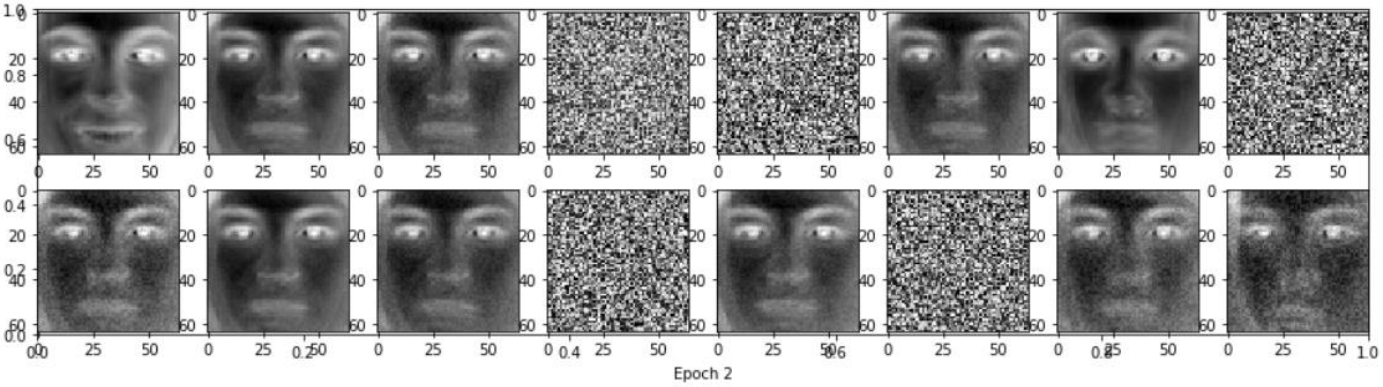
Initial randomized weights



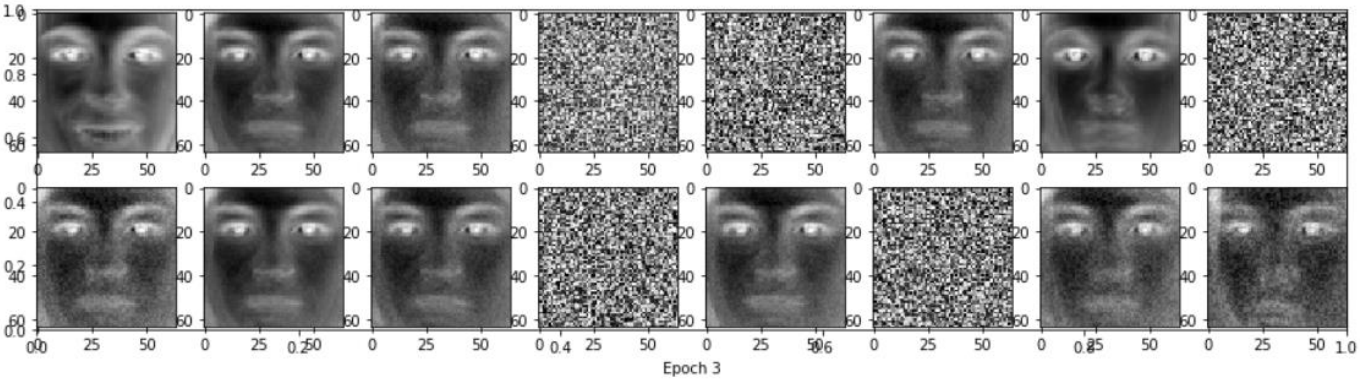
Epoch 0



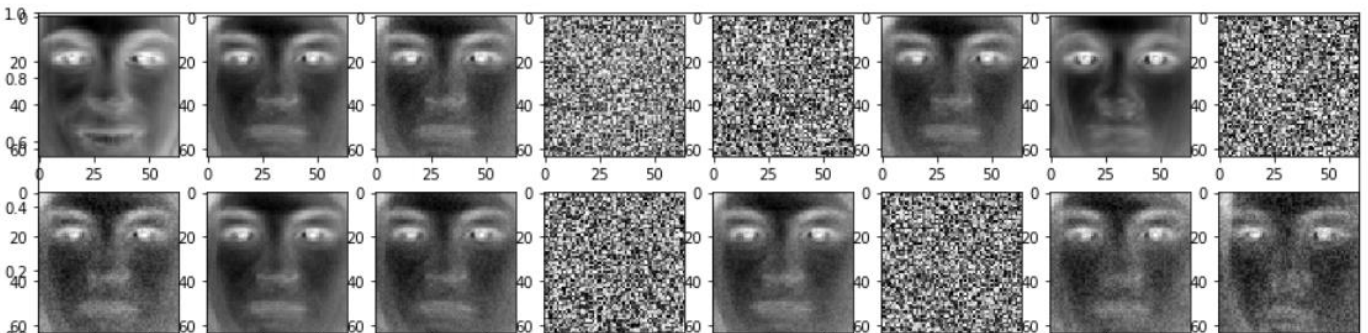
Epoch 1

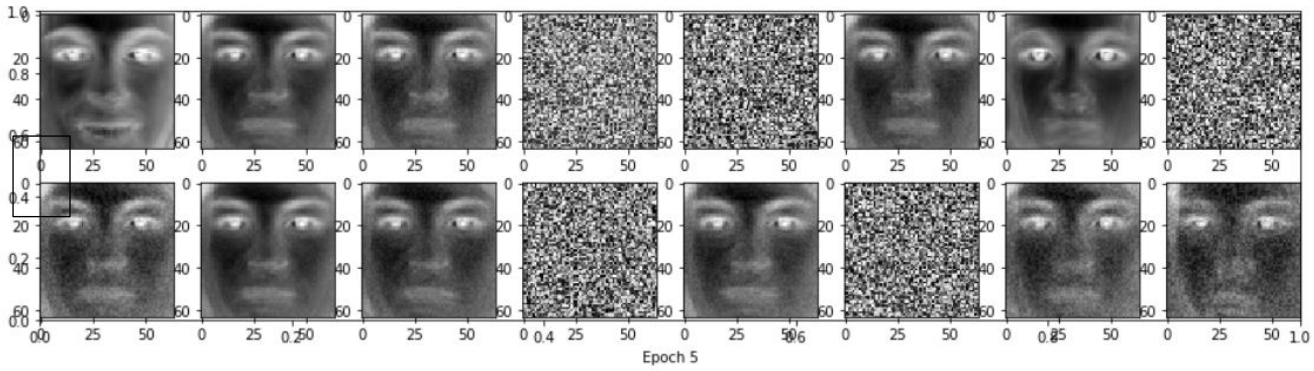


Epoch 2



Epoch 3





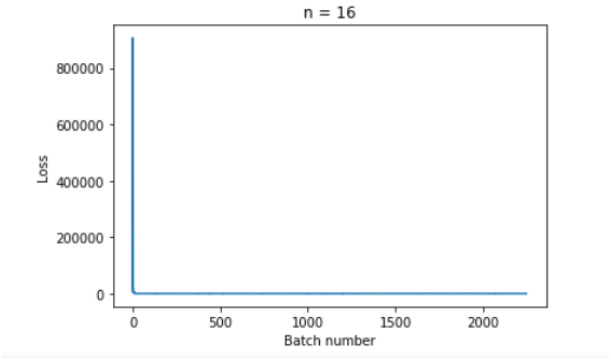


Figure: Losses vs Batch Number when N = 16

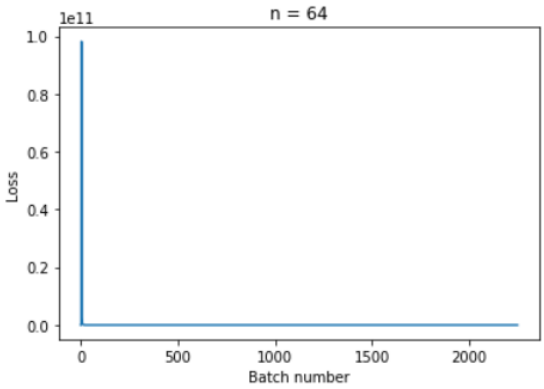
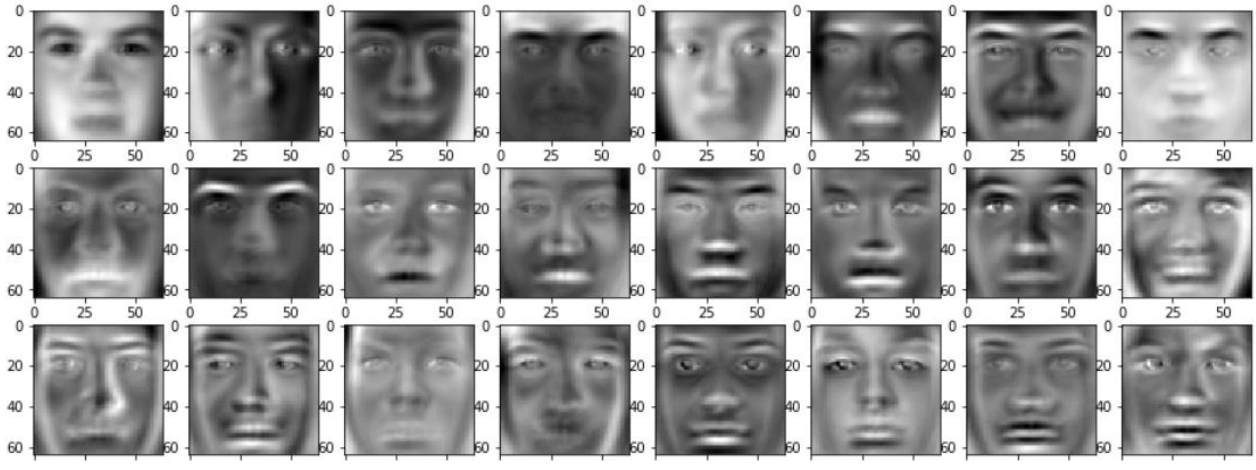


Figure: Losses vs Batch Number when N = 64

Conclusion: According to previous results plot, the model seems to get converged in the first epoch. Then there is no obvious change after the first epoch. The losses also dramatically decrease when batch number is increasing little from 0 to around 50. To see more changes, maybe we can decrease the learning rate and decrease the Batch Number to see more details on changes on loss and accuracies. See how the model get converged.

Q1.2



The figure above obtained are the first 24 eigenfaces.

An eigenface basically a set of eigenvectors which form a basis set of all images used to construct the covariance matrix. Eigenface is a dimension reduction method, thus the system can represent multiple subjects with a small set of data.

Eigenfaces are weighted to create faces, the previous section (the network) does a similar weighting of inputs but it was based on per-pixel. More similarities can be noticed between how eigenfaces and faces created in the visualized parameters.

Q2

Shallow MLP

a) In part a, learning rate is set as 0.1 and epochs is 500. The following plots were obtained for each value of n = 32, 128 and 512. Moreover, some screenshots for the code are also attached below.

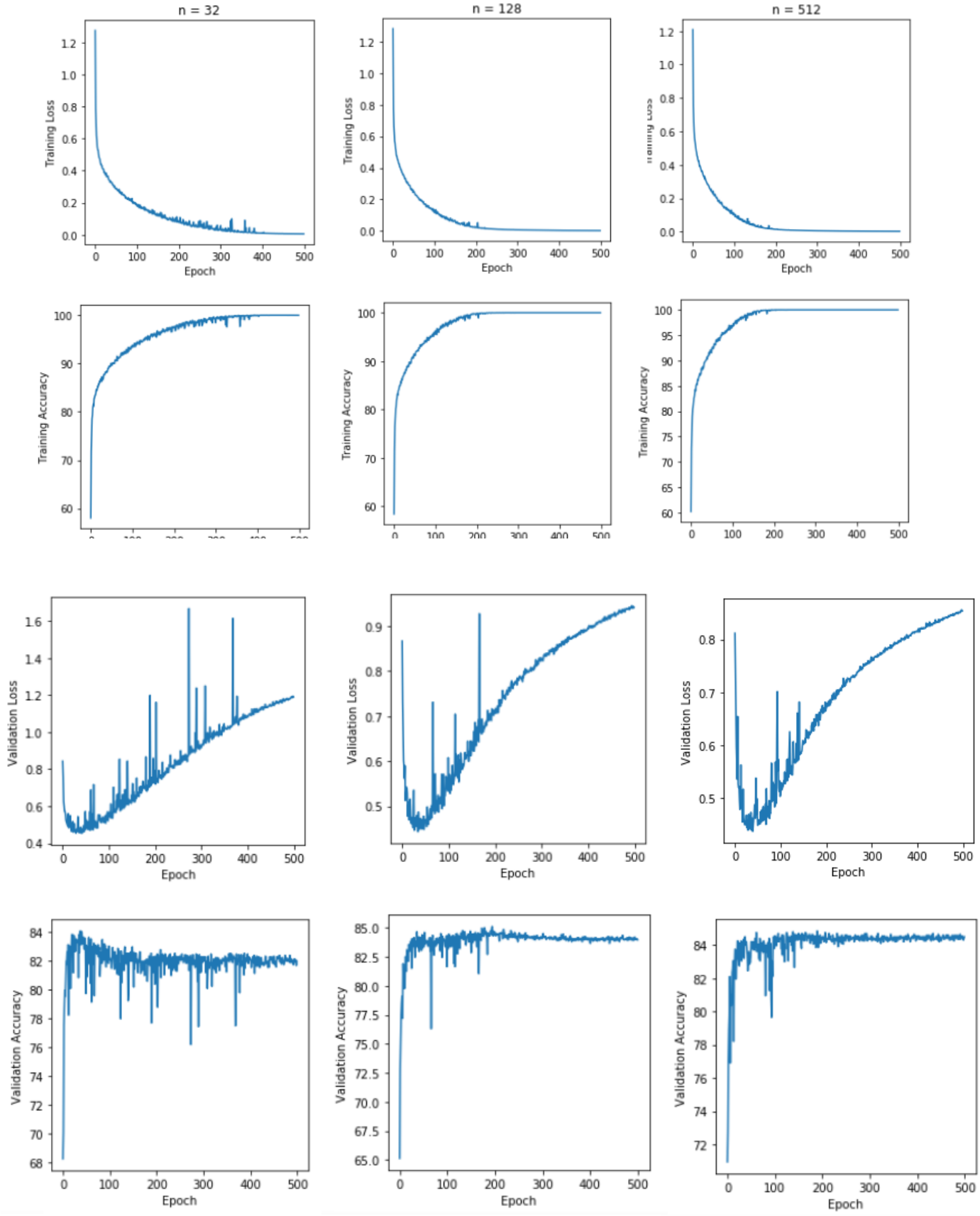
```
# Set device to GPU_index if GPU is available
GPU_index = 0
device = torch.device(GPU_index if torch.cuda.is_available() else 'cpu')

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

        input_size = 784
        h1 = 32
        output_size = 10
        self.linear1=nn.Linear(input_size,h1)
        self.linear2=nn.Linear(h1,output_size)

    def forward(self, x):

        x=self.linear1(x)
        x=torch.relu(x)
        x=self.linear2(x)

        return x
```

```
# The train_epoch is quite similar from Lab 5 Practice
def train_epoch(model, train_loader, criterion, optimizer):
    model.train()
    model.to(device)

    total_loss = 0
    correct_predictions = 0
    total_predictions = 0

    for batch_idx, (data, target) in enumerate(train_loader):
        optimizer.zero_grad()
        outputs = model(data.to(device))
        loss = criterion(outputs,target.to(device))

        _,predicted = torch.max(outputs.data,1)
        correct_predictions += (predicted == target.to(device)).sum().item()
        total_predictions +=target.shape[0]
        total_loss = total_loss + loss.item()
        #loss_progress = loss_progress.append(loss.item())
        loss.backward()
        optimizer.step()

    acc = (correct_predictions/total_predictions)*100
    #loss_avg = sum(loss_progress)/length(loss_progress)
    loss_avg = total_loss / len(train_loader)
    return loss_avg,acc
```

```python
# The train_epoch is quite similar from Lab 5 Practice
def test_model(model,test_loader,criterion,optimizer):
    loss_avg = 0
    total_loss = 0
    model.to(device)
    total_predictions = 0
    correct_predictions = 0
    with torch.no_grad():
        model.eval()


        for batch_idx,(data,target) in enumerate(test_loader):
            outputs = model(data.to(device))
            target = target.to(device)
            _,predicted = torch.max(outputs.data,1)
            correct_predictions+=(predicted == target).sum().item()
            total_predictions += target.shape[0]

            # Calcualte teh loss
            loss = criterion(outputs,target.to(device))
            total_loss = total_loss + loss.item()

    loss_avg = total_loss/len(test_loader)
    acc = (correct_predictions / total_predictions)*100
    return loss_avg,acc
```

```python
    return loss_avg,acc


train_loss_set = list()
test_loss_set = list()
test_acc_set = list()
train_acc_set = list()

# Define the hyperparameters
lr = 1e-3
n_epoches = 1000
n = 32
# Create Model
model = MLP()


# Create the Loss function
criterion = nn.CrossEntropyLoss()
# Initialize the optimizer with the above pareameters
optimizer = optim.SGD(model.parameters(), lr)
#optimizer = optim.Adam(model.parameters(),lr = lr)


for i in range(n_epoches):
    print('Epoch', i+1)
    train_loss,train_acc = train_epoch(model, trainloader, criterion, optimizer)
    test_loss,test_acc = test_model(model,testloader,criterion,optimizer)
    train_loss_set.append(train_loss)
    test_loss_set.append(test_loss)
    test_acc_set.append(test_acc)
    train_acc_set.append(train_acc)
    print("--"*30)
```

```python
# Visualize the Training Data
figure = plt.figure(figsize = (4,4))
plt.title('n = '+str(n))
plt.ylabel('Training Loss')
plt.xlabel('Epoch')
plt.plot(range(len(train_loss_set)),train_loss_set)

figure = plt.figure(figsize = (4,4))
plt.title('n = '+str(n))
plt.ylabel('Training Accuracy')
plt.xlabel('Epoch')
plt.plot(range(len(train_acc_set)),train_acc_set)

figure = plt.figure(figsize = (4,4))
plt.title('n = '+str(n))
plt.ylabel('Validation Loss')
plt.xlabel('Epoch')
plt.plot(range(len(test_loss_set)),test_loss_set)

figure = plt.figure(figsize = (4,4))
plt.title('n = '+str(n))
plt.ylabel('Validation Loss')
plt.xlabel('Epoch')
plt.plot(range(len(test_acc_set)),test_acc_set)
```



Accuracy: 82%

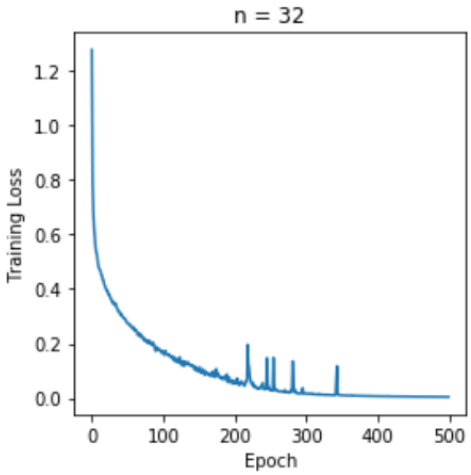
Accuracy: 84%

Accuracy: 84.5%

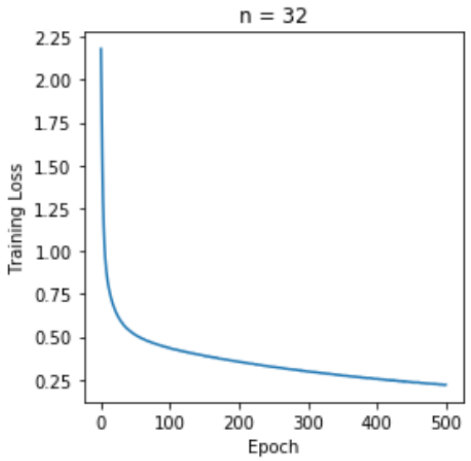
Part a conclusion: The higher the value of n is, higher accuracy it had against the validation set. And it takes less epochs to get converged. However, as the n is becoming larger, it will take more time to complete. Thus, it needs practice to find out the optimal n value to fit accuracy and timing issue.

b)

Initially, learning rate was set as 0.1, then it has dropped to 0.01. Both epochs are set as 500. When looking at the left diagram with higher learning rate, we can tell when the epoch reaches 200, the slope has decreased.



Learning rate = 0.1

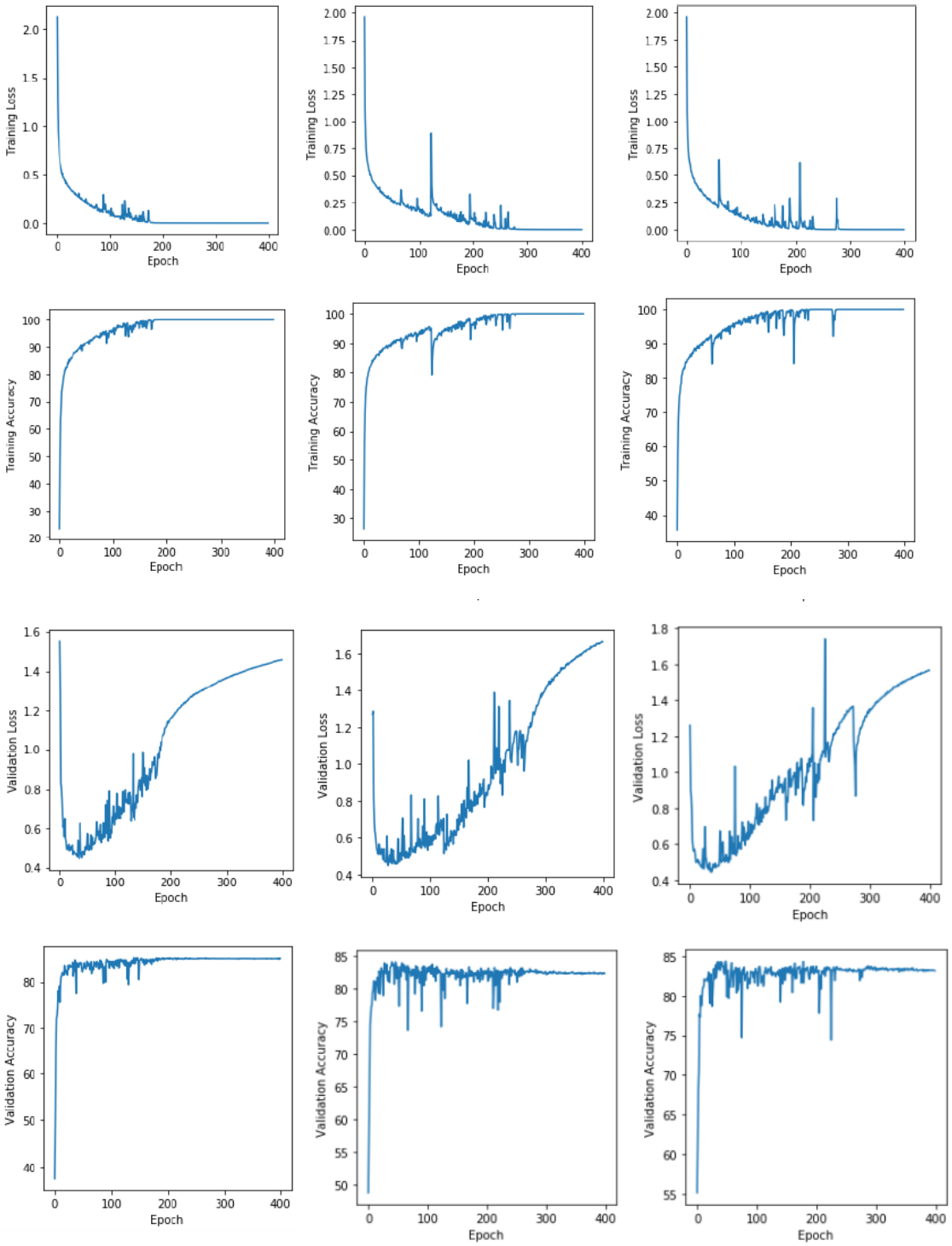


Learning rate = 0.01

Conclusion: Based on the two plots above, decreasing the learning rate will not speed up the convergence because the gradient descent will be relatively slow. However, if the learning rate was set too high, then it is not possible to reach the convergence point as high learning rate might cause some overshoot (jump too much). Decreasing learning rate will decrease the step size to reach global minimum point quicker and sometimes it will cause faster convergence. While the lower learning rate might take more time to get converged and get the plot regarding training loss against epoch smoother.

Deep MLP

The following results are obtained by setting the learning rate as 0.1 and epochs are 400.



Model A (Part A)	Model B (Part B)	Model C (Part C)
Accuracy: 85%	Accuracy: 82.3%	Accuracy: 83.2%

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

        input_size = 784
        h1 = 128
        h2 = 64
        h3 = 32
        output_size = 10
        self.linear1=nn.Linear(input_size,h1)
        self.linear2=nn.Linear(h1,h2)
        self.linear3=nn.Linear(h2,h3)
        self.linear4=nn.Linear(h3,output_size)

    def forward(self, x):

        x = self.linear1(x)
        x = torch.relu(x)
        x = self.linear2(x)
        x = torch.relu(x)
        x = self.linear3(x)
        x = torch.relu(x)
        x=self.linear4(x)

        return x
```

Figure: code for Deep MLP

Conclusion:

From the simulation result, the Model A has the highest accuracy among three models, however, the difference is not a lot compared with model C and model B. In addition, the model C only use half number of parameters. Model A is the quickest model to get converged in terms of epochs. However, due to high parameters, each epoch takes more time to complete. Hence, it might need little more effort to test which model is the quickest to get converged in terms of time.

Compared with previous section (shallow networks), model A is still the highest model. However, there is still not a lot of difference, the difference between model A and the lowest model is just 3%. The shallow network with 128 hidden units performed little better than model B and C from deep network in terms of accuracy.

Sigmoid

Model A was chosen due to highest accuracy from previous practice. In this case, the ReLU was replaced by sigmoid function.

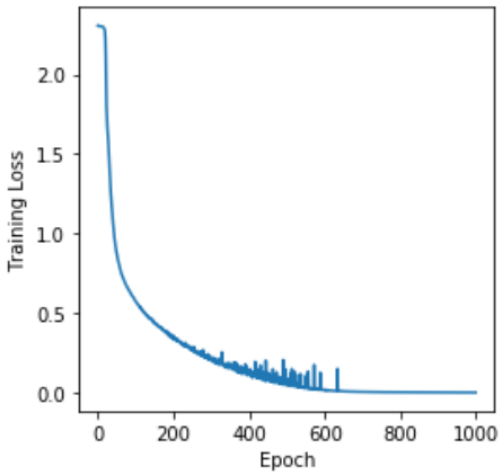


Figure for Training Loss (Sigmoid)

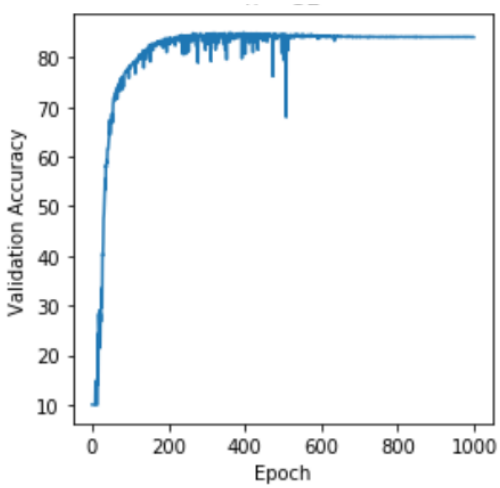


Figure for Validation Accuracy (Sigmoid)

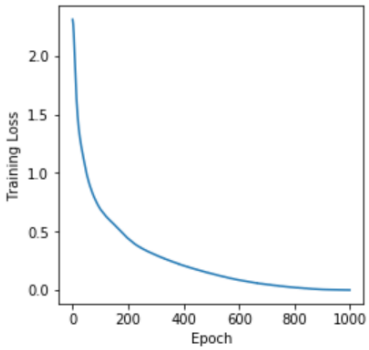
Conclusion: When staring at the following figure, it is clear to tell that sigmoid function give lower accuracy and longer convergence time. Accuracy is 83.3 when using Sigmoid, which is lower than ReLU (85%). In terms of number of epoch, it takes about 600 for validation accuracy with sigmoid while ReLU needs 200 epochs(seen from previous plot).

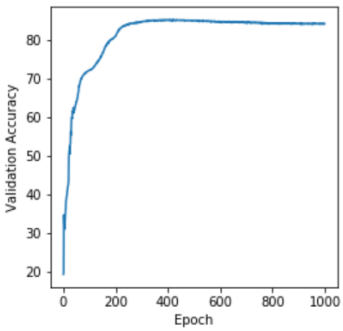
Optimizer

The learning rate for SGD is 0.1 and 0.0001 for ADAM. In addition to see more changes, epochs are set higher like 1000. The next four plots are shallow and deep model with ADAM optimizer.

Deep – ADAM

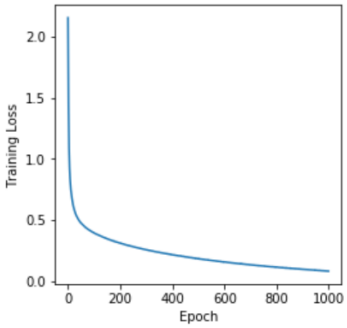
Accuracy: 82.25%

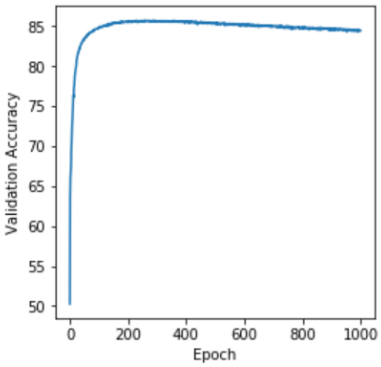




Shallow – ADAM

Accuracy: 81.65%



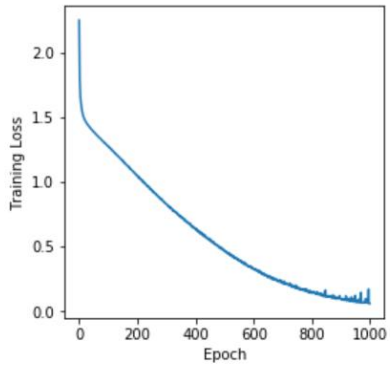


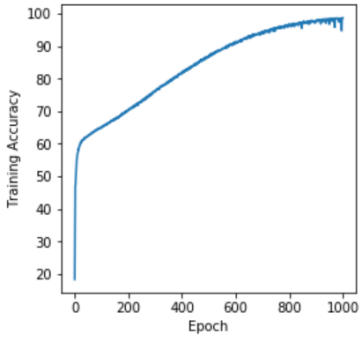
Conclusion: It seems that the ADAM optimizer takes much longer time to get converged at around 800 epochs for the deep MLP compared with SGD optimizer method. However, the results plots are quite smooth.

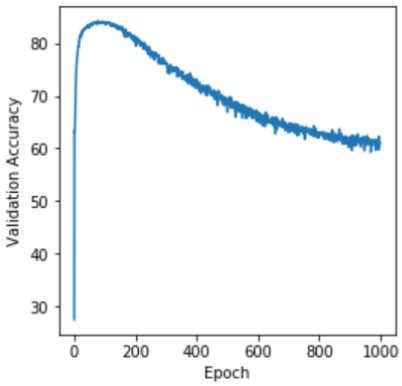
In addition, for the shallow ADAM, it seems it does not get converged at 1000. Because even if the training loss is decreasing all the time while the validation accuracy reaches on global maximum point, then it begins to have a decrease thread.

For all the ADAM optimizer plot, testing loss and validation accuracy curves are smoother than SGD optimizer plots. Moreover, ADAM seems to provide better accuracies, but it took more epochs (tradeoff between time in terms of epochs and accuracy) compared with SGD. And ADAM seems to be more complex than SGD.

Q3.1







The Learning rate is set as 0.0001, and epochs are 1000. It was not possible to train the model to get as the trend is clear where it reaches one global point then it starts to decrease in term of validation accuracies. Even if the training accuracy keeps increasing, while the test accuracy is decreasing after reaching the peak. Unlike previous practice, the validation accuracy has somehow negative correlation with training accuracy after 200 epochs. Moreover, there seems some noisy data on validation accuracy when the epoch is higher than 300.

When the epoch reaches 1000, the training loss is quite small, which indicates that the model fits the data. But while when it comes to validation accuracy, it has turned out another situation. It might indicate the overfitting phenomenon. Overfitting to the training data will allow the test data not feel comfortable.