Assignment 1

Qiang Zhao (2814236)

November 2023

1 Answers

question 1
a) The local derivatives of the softmax activation function contain two situations.

The first condition is as follows:
When $i = j$:

$$\frac{\partial y_i}{\partial o_j} = \frac{\partial(exp^{o_i} / \sum_j exp^{o_j})}{\partial o_j}$$

$$= \frac{exp^{o_i} \sum_j exp^{o_j} - exp^{o_i} exp^{o_i}}{(\sum_j exp^{o_j})^2}$$

$$= \frac{exp^{o_i}}{\sum_j exp^{o_j}} \left(1 - \frac{exp^{o_i}}{\sum_j exp^{o_j}}\right)$$

$$= y_i(1 - y_i)$$

The second condition is as follows:
When $i \neq j$:

$$\frac{\partial y_i}{\partial o_j} = \frac{\partial(exp^{o_i} / \sum_j exp^{o_j})}{\partial o_j}$$

$$= -\frac{exp^{o_i} exp^{o_j}}{(\sum_j exp^{o_j})^2}$$

$$= -\frac{exp^{o_i}}{\sum_j exp^{o_j}} \frac{exp^{o_j}}{\sum_j exp^{o_j}}$$

$$= -y_i y_j$$

b) The local derivatives of the log loss function is as follows.
If c=i:

$$\frac{\partial l}{\partial y_i} = \frac{\partial(-\ln(y_c))}{\partial y_i}$$

$$= -\frac{1}{y_c}$$

Otherwise:

$$\frac{\partial l}{\partial y_i} = 0$$

question 2

If we assign the label as a_j, the local derivatives of the input of the softmax is as follows:

$$\frac{\partial l}{\partial o_i} = \frac{\partial(-\sum_j a_j \ln(y_i))}{\partial o_i}$$

$$= -\sum_j a_j \frac{1}{y_i}\frac{\partial y_i}{\partial o_i}$$

$$= -a_j \frac{1}{y_i} y_i(1-y_i) - \sum_{i \neq j} a_j \frac{1}{y_i}(-y_i y_j)$$

$$= a_j y_i - a_j + \sum_{i \neq j} a_j y_j$$

$$= \sum_j a_j y_j - a_j$$

$$= y_j \sum_j a_j - a_j$$

$$= y_j - a_j$$

We do not need the step because we can get the same result using **the chain rule** ($\frac{\partial l}{\partial o_i} = \frac{\partial l}{\partial y_i}\frac{\partial y_i}{\partial o_i}$).

question 3

The relevant code was showing as follows:

```
#Setting up some predefined functions
def layer(input,output,weight,bias):
    n_unit = len(output)
    n_feature = len(input)
    for i in range(n_unit):
        for j in range(n_feature):
            output[i] += weight[j][i] * input[j]
        output[i] += bias[i]
    return output

def sigmoid(size,layer_1):
    s = [float(0) for i in range(size)]
    for i in range(len(s)):
        s[i] = math.exp(layer_1[i]) / (1+(math.exp(layer_1[i])))
    return s

def softmax(size,layer_2):
    layer_2_soft = np.zeros(len(layer_2)).tolist()
    layer_2_soft_deno = sum([math.exp(i) for i in layer_2])
    for j in range(len(layer_2)):
        layer_2_soft[j] = math.exp(layer_2[j]) / layer_2_soft_deno
    return layer_2_soft
```

```
def log_loss(pred,label):
    return -((label[0] * math.log(pred[0])) + (label[1] * math.log(pred[1])))

def one_hot_scalar(_,size):
    a = np.asarray([1 if i == _ else 0 for i in range(size)]).tolist()
    return a

#Define the class of scalar Multi-layer Perceptron
class scalarNN:
    def __init__(self,feature_size,layer_1_size,layer_2_size,fix_weight = True):
        self.fix = fix_weight
        self.l1_size = layer_1_size
        self.l2_size = layer_2_size

        if self.fix: #Fixed weight only for question 3
            self.weight_1 = [[1, 1, 1], [-1, -1, -1]]
            self.weight_2 = [[1, 1], [-1, -1], [-1, -1]]
        else: #Setting up weights from the Standard Normal Distribution
            self.weight_1 = [np.random.rand(self.l1_size).tolist() for i in range(feature_size)]
            self.weight_2 = [np.random.rand(self.l2_size).tolist() for i in range(self.l1_size)]

        self.bias_1 = [0 for i in range(self.l1_size)]
        self.bias_2 = [0 for i in range(self.l2_size)]


    def forward(self,input):
        layer_1_init = [float(0) for i in range(self.l1_size)]
        self.layer_1 = layer(input,layer_1_init,self.weight_1,self.bias_1)
        self.sigmoid_out = sigmoid(self.l1_size,self.layer_1)

        layer_2_init = [float(0) for i in range(self.l2_size)]
        self.layer_2 = layer(self.sigmoid_out,layer_2_init,self.weight_2,self.bias_2)
        self.softmax_out = softmax(self.l2_size,self.layer_2)
        return self.softmax_out

    def backward(self,x,pred,label,lr):
        layer_2_ld = [float(0) for i in range(self.l2_size)] #zero gradients before start
        for i in range(self.l2_size):
            layer_2_ld[i] = pred[i] - label[i]

        layer_1_sig_ld = [float(0) for i in range(self.l1_size)]
        for i in range(self.l1_size):
            for j in range(self.l2_size):
                layer_1_sig_ld[i] += layer_2_ld[j] * self.weight_2[i][j]

        layer_2_v_deri = [np.zeros(self.l2_size).tolist() for i in range(self.l1_size)]
        layer_2_c_deri = np.zeros(self.l2_size).tolist()
        for i in range(self.l2_size):
            for j in range(self.l1_size):
                layer_2_v_deri[j][i] = layer_2_ld[i] * self.sigmoid_out[j]
                self.weight_2[j][i] -= lr * layer_2_v_deri[j][i] #update weight v
            layer_2_c_deri[i] = layer_2_ld[i]
            self.bias_2[i] -= lr * layer_2_c_deri[i] #update bias c
```

```
        layer_1_ld = np.zeros(self.l1_size).tolist()
        for i in range(self.l1_size):
            layer_1_ld[i] = layer_1_sig_ld[i] * self.sigmoid_out[i] * (1 - self.sigmoid_out[i])

        layer_1_w_deri = [np.zeros(self.l1_size).tolist() for i in range(self.l2_size)]
        layer_1_b_deri = np.zeros(self.l1_size).tolist()
        for i in range(self.l1_size):
            for j in range(self.l2_size):
                layer_1_w_deri[j][i] = layer_1_ld[i] * x[j]
                self.weight_1[j][i] -= lr * layer_1_w_deri[j][i] #update weight w
        layer_1_b_deri[i] = layer_1_ld[i]
        self.bias_1[i] -= lr * layer_1_b_deri[i] #update bias b

#Setting up the architecture of the network
feature_size = 2
layer_1_size = 3
layer_2_size = 2
lr = 0.1

#Network initialization
scalar_NN = scalarNN(feature_size,layer_1_size,layer_2_size,fix_weight=True)

#Prepare the input data
x = [1,-1]
label = [1,0]
pred = scalar_NN.forward(x)
loss = log_loss(pred, label)
scalar_NN.backward(x,pred,label,lr)
```

question 4

In question 4, we used the synthetic data from the following pre-defined function:
https://gist.github.com/pbloem/bd8348d58251872d9ca10de4816945e4

The training loop contained **100** epochs, each with **60,000** iterations; the parameters were updated for each instance. We calculated the average loss for each epoch. The relevant code was as follows:

```
#Load the synth data and normalizaiton
(xtrain, ytrain), (xval, yval), num_cls = load_synth()
data = xtrain
min_vals = np.min(data, axis=0)
max_vals = np.max(data, axis=0)
normal_data = (data - min_vals) / (max_vals - min_vals)

#Setting up the architecture of the network
feature_size = 2
layer_1_size = 3
layer_2_size = 2
lr = 0.1
iteration = normal_data.shape[0]
epoch = 100

#Network initialization
scalar_NN = scalarNN(layer_1_size,layer_2_size,fix_weight=False) #Close the fixed weight

#Feed the synthetic data to the network
```

```
y_loss = []
for k in range(epoch):
    total_loss = 0
    for i in range(1,iteration):
        x = list(i for i in list(normal_data[i-1]))
        label = one_hot_scalar(ytrain[i-1],2)
        print("epoch:",k+1,"round:",i,"/",normal_data.shape[0])
        pred = scalar_NN.forward(x)
        loss = log_loss(pred,label)
        total_loss += loss
        scalar_NN.backward(x,pred,label,lr)
    y_loss.append(total_loss / normal_data.shape[0])

#Plot the lossing curve (The plot_loss_4 function was provided in the appendix part)
plot_loss_4(y_loss,epoch,"Loss curve","Epoch","Average Loss over past one epoch")
```

From Figure **1**, we can see the loss drops as training progresses, and it reaches convergence after round the **20th** epoch.

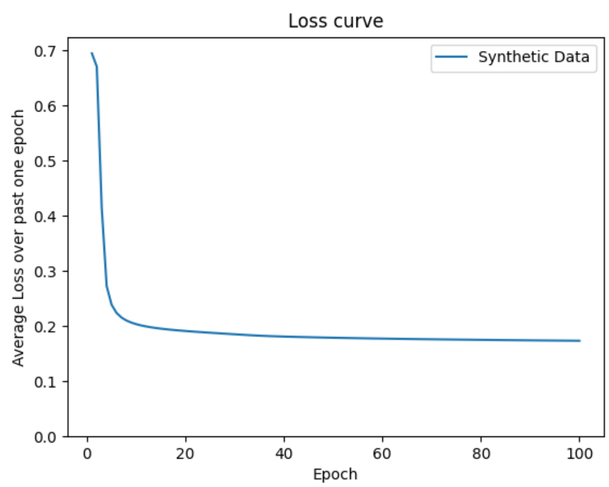

Figure 1: The loss curve of Synthetic Data

question 5 and 6

The code below demonstrates the process of implementing a **vectorized neural network**:

```
#Setting up some predefined functions
def log_loss_tensor(pred,label):
    pred = np.clip(pred, 1e-15, 1 - 1e-15)
    loss = - np.mean(np.log(pred)*label)
    return loss
def one_hot_vec(_,size):
    a = np.asarray([1 if i == _ else 0 for i in range(size) ]).reshape(size,1)
    return a

class tensorNN:
    def __init__(self,feature_size,layer_1_size,layer_2_size,batch_size):
        self.feature_s = feature_size #784
```

```python
        self.l1_s = layer_1_size #300
        self.l2_s = layer_2_size #10
        self.batch = batch_size

        self.w = np.expand_dims(np.random.rand(self.l1_s) * 0.01,axis=0)
        self.w = np.repeat(self.w, self.feature_s, axis=0).reshape(self.l1_s, self.feature_s)
        self.b = np.array([0 for i in range(self.l1_s)]).astype('float64')

        self.v = np.expand_dims(np.random.rand(self.l2_s) * 0.01,axis=0)
        self.v = np.repeat(self.v, self.l1_s, axis=0).reshape(self.l2_s, self.l1_s)
        self.c = np.array([0 for i in range(self.l2_s)]).astype('float64')

    def forward(self,x):
        x = x.reshape(self.batch,self.feature_s)
        self.k1 = np.matmul(x,self.w.T) + self.b
        self.sigmoid_output = np.exp(self.k1) / (1+np.exp(self.k1))
        self.k2 = np.matmul(self.sigmoid_output,self.v.T) + self.c
        self.softmax_output = np.exp(self.k2) / (np.exp(self.k2).sum(axis=-1, keepdims=True))
        return self.softmax_output

    def backward(self,x,label,lr):
        x = x.reshape(self.batch, self.feature_s)
        l2_ld = self.softmax_output - label
        l1_sig_ld = np.matmul(l2_ld, self.v)
        l2_v_deri =  np.matmul(l2_ld.T, self.sigmoid_output) / label.shape[0]
        l2_c_deri = np.sum(l2_ld, axis=0) / label.shape[0]

        s_pr = self.sigmoid_output * (1 - self.sigmoid_output)
        l1_ld = s_pr * l1_sig_ld
        l1_w_deri = np.matmul(l1_ld.T,x) / label.shape[0]
        l1_b_deri = np.sum(l1_ld, axis=0) / label.shape[0]

        self.w -= lr * l1_w_deri
        self.b -= lr * l1_b_deri
        self.v -= lr * l2_v_deri
        self.c -= lr * l2_c_deri

#Setting up the shape of vectorized neural network
feature_size = 784
layer_1_size = 300
layer_2_size = 10
batch_size = 32
lr = 0.03

#Load the MNIST data
(xtrain, ytrain), (xval, yval), num_cls = load_mnist(final=False, flatten=True)
#Initialization of the Network
tensor_NN = tensorNN(feature_size,layer_1_size,layer_2_size,batch_size)
xaxis,yaxis,accuracy = [],[],0
rounds = int(xtrain.shape[0]/batch_size)
for r in range(rounds):
        start = (r) * batch_size
        end = (r + 1) * batch_size
        x_flatten = xtrain[start:end].reshape(batch_size,feature_size).astype('float32')
```

```
        x_flatten /= 255 #Normalization of MNIST DATA
        label = list(ytrain[start:end])
        label_m = np.empty((batch_size, layer_2_size))
        for i in range(len(label)):
            label_inter = one_hot_vec(label[i], layer_2_size)
            label_m[i, :] = label_inter.squeeze()
        pred = tensor_NN.forward(x_flatten)
        loss = log_loss_tensor(pred,label_m)
        tensor_NN.backward(x_flatten, label_m, lr)
        pred_max = np.argmax(pred, axis=1)
        label_max = np.argmax(label_m, axis=1)
        accuracy += np.mean(pred_max == label_max) #record accuracy for each batch
        yaxis.append(loss)
        xaxis.append(end)
accuracy = round(accuracy/rounds,3) #calculate average accuracy
#Plot the lossing curve (The plot_loss_5 function was provided in the appendix part)
plot_loss_5("Loss curve","Number of training samples",
"Average Loss over past one batch of samples (n=32)",accuracy,"05-one_epoch.png")
```

Because questions 5 and 6 both require the vectorized network, we combine the questions together. When implementing the network for one sample, you only need to set the parameter batch size to **1**. The following analysis showed the result of batched gradient descent:

In tests on the **MNIST dataset**, we conducted the training process with a batch size of **32** and a learning rate of **0.03**. **Figure 2** shows the loss curve and average accuracy after just **one epoch**. **Figure 3** displays the loss curve and average accuracy after **100 epochs**. The above codes present the training process during one epoch. The code of 100 epochs was shown in the appendix part.

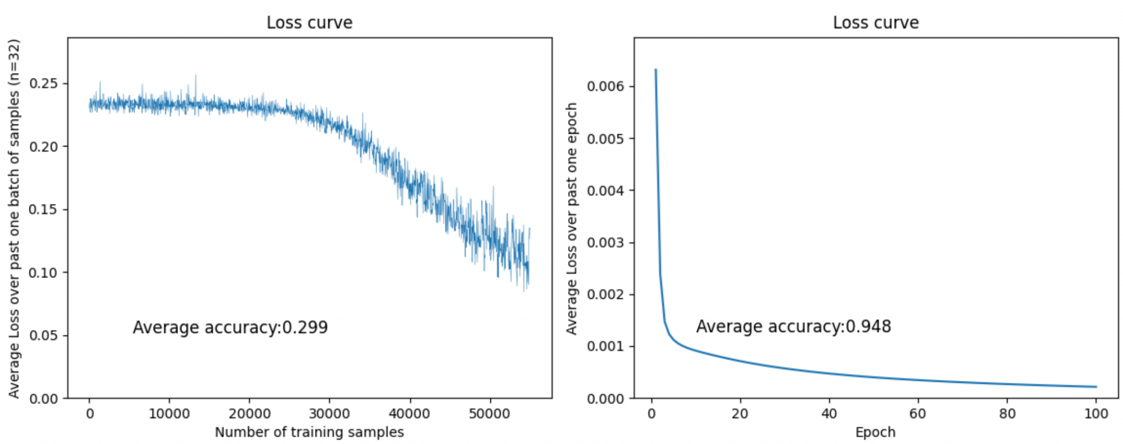

Figure 2: The loss curve of MNIST data after one epoch(left) and 100 epochs(right)

question 7

1) The parameter used in this question is as follows:
The batch size is **32**, and the learning rate is **0.03**, as before.
Figure 3 indicates the difference between the loss curves for the training and validation data, which lies in the fact that the training data starts with a lower loss and converges more quickly, achieving convergence after the fifth epoch. In contrast, the validation data begins with a higher initial loss and converges more slowly during the 5 epochs.

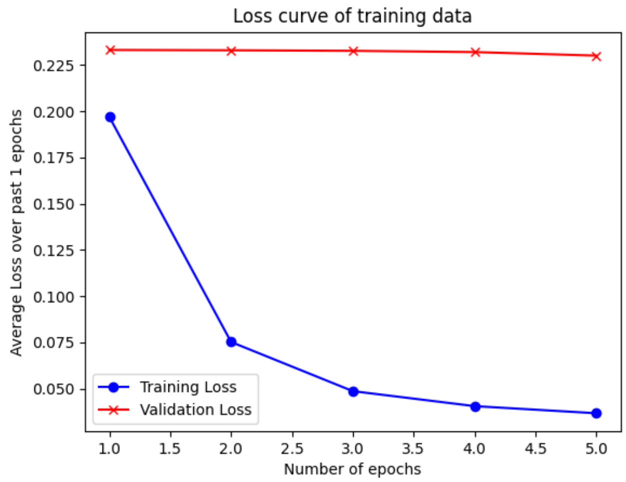

Figure 3: The loss curve of training data and validation data during 5 epochs

2)The parameter used in this question is as follows:

The batch size is **32**, and the learning rate is **0.03**, as before.

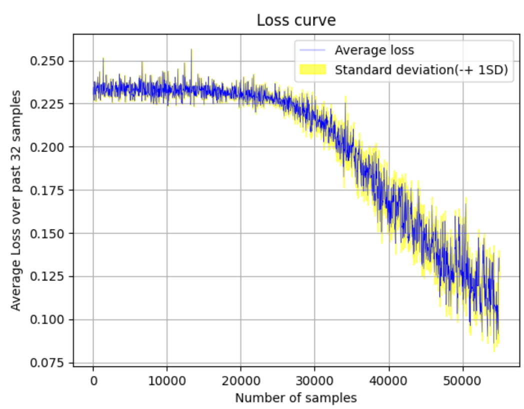

Figure 4: The loss curve of training data from 10 different initialzaitons

From **Figure 4**, we can see that under the random initialization parameter (Implemented **10 times** in this question), as indicated by the SD (Standard Deviation) results, the loss curve fluctuates less with a smaller number of training samples. As training progresses, the SD gradually increases, leading to more significant fluctuations in the loss.

3) We employed two types of SGD approaches for comparison: stochastic gradient descent (**SGD**; batch size = **1**) and mini-batch stochastic gradient descent(**mini-batch SGD**; batch size = **5,16,32**). The comparison of the four sets of data in **Figure 5** shows that as the learning rate increases, the model enters the convergence process more rapidly. Interestingly, a higher batch size does not lead to a faster convergence for the same learning rate. The gradient of the smaller batch size groups starts to drop earlier than that of the batch size 32 group.

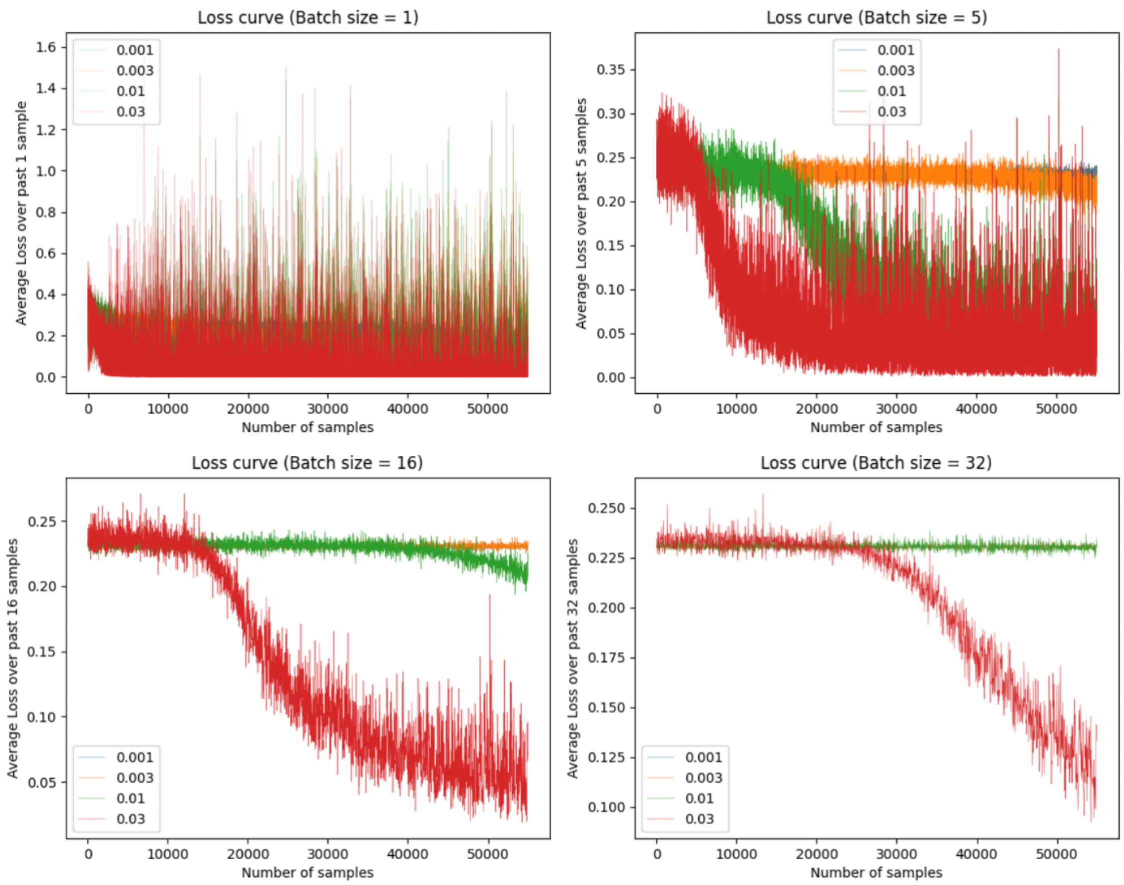

Figure 5: The loss curve training data with different learning rates and batch size

4) Based on the above analysis, we selected the following combination of hyperparameters:

a learning rate of **0.03** and a batch size of **16**.

We initially conducted five epochs on the complete MNIST training dataset. As shown in the left graph of **Figure 6**, the loss gradually decreased, and the accuracy steadily increased throughout training, reaching **0.916** in the fifth epoch. Subsequently, we performed a performance test for one epoch on the complete MNIST validation set. As seen in the right graph of **Figure 6**, the loss level remained stable, and the accuracy for this epoch reached **0.925**, which is higher than that of the training set. This indicates that our model has **good generalization ability** and did not exhibit overfitting.

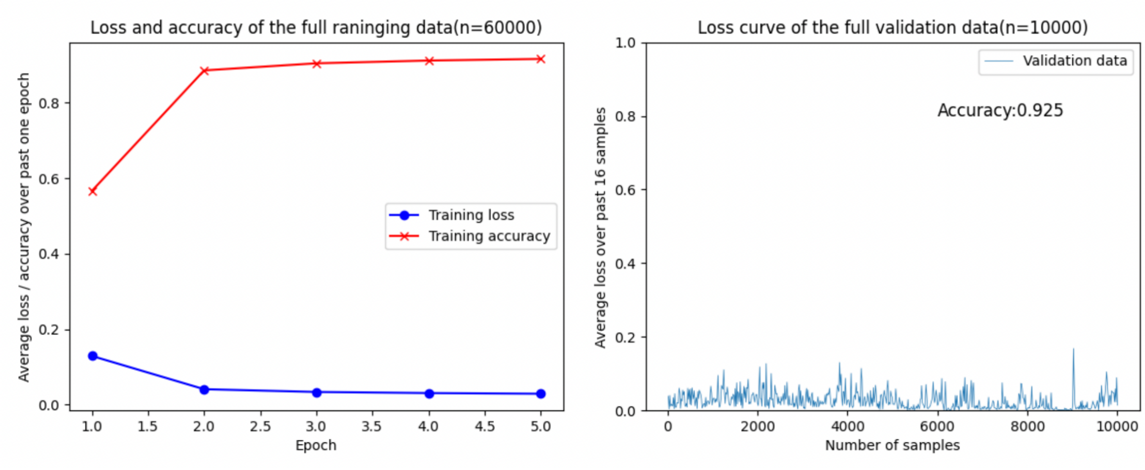

Figure 6: The performance of the final set of hyperparameters on the full training(left) and the full validation data(right)

A Appendix

Here are some relevant codes for different questions. Some of the code was pasted with line breaks done manually to account for page margins.

Question 4
The plot function for Figure 1

```
def plot_loss_4(loss_input,number_epoch,title,xtitle,ytitle):

    xaxis = list(range(1,number_epoch+1))
    yaxis = loss_input
    plt.plot(xaxis,yaxis,label='Synthetic Data')
    y_lim_max = max(yaxis) + 0.03
    y_lim_max = 1 if y_lim_max > 1 else y_lim_max

    plt.ylim(0, y_lim_max)
    plt.legend()
    plt.title(title)
    plt.xlabel(xtitle)
    plt.ylabel(ytitle)
    plt.savefig("04.png")
    plt.show()
```

Question 5 and 6
The training and plot code for 100 epochs of MNIST data

```
#Setting the shape of the network
feature_size = 784
layer_1_size = 300
layer_2_size = 10
batch_size = 32
lr = 0.03
epoch = 100
```

```
#Load the MNIST data
(xtrain, ytrain), (xval, yval), num_cls = load_mnist(final=False, flatten=True)

#Initialization
tensor_NN = tensorNN(feature_size,layer_1_size,layer_2_size,batch_size)

xaxis,yaxis = [],[]
rounds = int(xtrain.shape[0]/batch_size)
y_loss = []
accuracy = 0

#Start to training the model
for k in range(epoch):
    total_loss = 0
    for r in range(rounds):
            start = (r) * batch_size
            end = (r + 1) * batch_size
            x_flatten = xtrain[start:end].reshape(batch_size,feature_size).astype('float32')
            x_flatten /= 255 #Normalization of MNIST DATA

            label = list(ytrain[start:end])
            label_m = np.empty((batch_size, layer_2_size))
            for i in range(len(label)):
                label_inter = one_hot_vec(label[i], layer_2_size)
                label_m[i, :] = label_inter.squeeze()

            pred = tensor_NN.forward(x_flatten)
            loss = log_loss_tensor(pred,label_m)
            total_loss += loss

            pred_max = np.argmax(pred, axis=1)
            label_max = np.argmax(label_m, axis=1)

            accuracy += np.mean(pred_max == label_max) #record accuracy for each batch
            tensor_NN.backward(x_flatten,pred,label_m,lr)
            print("epoch:", k + 1, "round:", r, "/", xtrain.shape[0])
    y_loss.append(total_loss / xtrain.shape[0])

accuracy = round(accuracy/(rounds*epoch),3) #calculate average accuracy

#Define the plot function
def plot_loss_5_epoch(y_loss,number_epoch,title,xtitle,ytitle,accuracy,output):

    xaxis = list(range(1, number_epoch + 1))
    yaxis = y_loss
    xtitle = xtitle
    ytitle = ytitle
    title = title

    a = "Average accuracy:" + str(accuracy)
    plt.text(max(xaxis)*0.1, max(yaxis)*0.2, a, fontsize=12)

    plt.plot(xaxis,yaxis,label='MNIST Data')
    y_lim_max = max(yaxis)*1.1
```

```
    y_lim_max = 1 if y_lim_max > 1 else y_lim_max

    plt.ylim(0, y_lim_max)

    plt.title(title)
    plt.xlabel(xtitle)
    plt.ylabel(ytitle)
    plt.savefig(output)
    plt.legend()
    plt.show()

#Plot the loss curve of Figure 2(right)
plot_loss_5_epoch(y_loss,epoch,"Loss curve","Epoch","Average Loss over past one epoch",accuracy,
"05-100_epoch.png")
```

Question 7-1
The training and plot code for question 7-1:

```
#Setting the shape of the network
feature_size = 784
layer_1_size = 300
layer_2_size = 10
batch_size = 32
lr = 0.03

#Training_data
tensor_NN = tensorNN(feature_size,layer_1_size,layer_2_size,batch_size)
train = []
rounds = int(xtrain.shape[0]/batch_size)

for i in range(5):
    print("epoch:",i)
    train_epoch = 0
    for r in range(rounds):
            start = (r) * batch_size
            end = (r+1) * batch_size
            x_flatten = xtrain[start:end].reshape(batch_size,feature_size).astype('float32')
            x_flatten /= 255

            label = list(ytrain[start:end])
            label_m = np.empty((batch_size, layer_2_size))
            for i in range(len(label)):
                label_inter = one_hot_vec(label[i], layer_2_size)
                label_m[i, :] = label_inter.squeeze()

            pred = tensor_NN.forward(x_flatten)
            loss = log_loss_tensor(pred,label_m)
            tensor_NN.backward(x_flatten,label_m,lr)
            train_epoch += loss

    train.append(train_epoch/rounds)

#Validation data
```

```python
tensor_NN = tensorNN(feature_size,layer_1_size,layer_2_size,batch_size)
val = []
rounds = int(xval.shape[0]/batch_size)
for i in range(5):
    print("epoch:",i)
    test_epoch = 0
    for r in range(rounds):
            start = (r) * batch_size
            end = (r+1) * batch_size
            x_flatten = xval[start:end].reshape(batch_size,feature_size).astype('float32')
            x_flatten /= 255

            label = list(yval[start:end])
            label_m = np.empty((batch_size, layer_2_size))
            for i in range(len(label)):
                label_inter = one_hot_vec(label[i], layer_2_size)
                label_m[i, :] = label_inter.squeeze()

            pred = tensor_NN.forward(x_flatten)
            loss = log_loss_tensor(pred,label_m)
            tensor_NN.backward(x_flatten,label_m,lr)
            test_epoch += loss

    val.append(test_epoch/rounds)

#Define the plot function
def plot_epoch(train,val,title,xtitle,ytitle):
    epochs = [1, 2, 3, 4, 5]
    train_loss = train
    val_loss = val

    plt.plot(epochs, train_loss, color='blue', marker='o', label='Training Loss')
    plt.plot(epochs, val_loss, color='red', marker='x', label='Validation Loss')

    plt.title(title)
    plt.xlabel(xtitle)
    plt.ylabel(ytitle)
    plt.legend()
    plt.savefig('07-01.png')
    plt.show()

#Plot the loss curve of figure 3
plot_epoch(train,val,"Loss curve of training data", "Number of epochs", "Average Loss
over past 1 epochs")
```

Question 7-2

The training and plot code for question 7-2:

```python
#Setting the shape of the network
feature_size = 784
layer_1_size = 300
layer_2_size = 10
batch_size = 32
```

```
lr = 0.03

rounds = int(xtrain.shape[0]/batch_size)

total_loss_x = [[] for _ in range(10)]
total_loss_y = []

for k in range(10):
    print("epoch:",k)
    tensor_NN = tensorNN(feature_size,layer_1_size,layer_2_size,batch_size)
    for r in range(rounds):
            start = (r)*batch_size
            end = (r+1)*batch_size
            x_flatten = xtrain[start:end].reshape(batch_size,feature_size).astype('float32')
            x_flatten /= 255 #normalization

            label = list(ytrain[start:end])
            label_m = np.empty((batch_size, layer_2_size))
            for i in range(len(label)):
                label_inter = one_hot_vec(label[i], layer_2_size)
                label_m[i, :] = label_inter.squeeze()

            pred = tensor_NN.forward(x_flatten)
            loss = log_loss_tensor(pred,label_m)
            total_loss_x[k].append(loss)
            if  k == 0 : total_loss_y.append(end)
            tensor_NN.backward(x_flatten,label_m,lr)

def plot_mean_sd(x,y,title,xtitle,ytitle):
    y = y[0:len(x[0])]
    #for i in range(len(x)):
    #    plt.plot(y,x[i],color = "blue",linewidth=0.2)
    x = np.array(x)
    mean_x = np.mean(x, axis=0)
    sd_x = np.std(x, axis=0)

    plt.plot(y,mean_x, color ="blue",lw=0.3,label="Average loss")
    plt.fill_between(y, mean_x - sd_x, mean_x + sd_x,color='yellow', alpha=0.6,
    label = "Standard deviation(-+ 1SD)")
    plt.title(title)
    plt.xlabel(xtitle)
    plt.ylabel(ytitle)
    plt.legend()
    plt.grid()
    plt.savefig('07-02.png')
    plt.show()

#Plot the loss curve of figure 4
plot_mean_sd(total_loss_x,total_loss_y,"Loss curve","Number of samples","Average Loss
over past 32 samples")
```

Question 7-3
The training and plot code for question 7-3:

```
#Setting the shape of the network
feature_size = 784
layer_1_size = 300
layer_2_size = 10

batch_size = 1 # or 5,16,32 [Change the value manually everytime]
lr_list = [0.001, 0.003, 0.01, 0.03]

total_loss_x = [[],[],[],[]]
total_loss_y = []

rounds = int(xtrain.shape[0]/batch_size)

#Iterate all the learning rate
for k in range(len(lr_list)):
    lr = lr_list[k]
    print("epoch:",k)
    print("lr:",lr)
    tensor_NN = tensorNN(feature_size, layer_1_size, layer_2_size, batch_size)
    for r in range(rounds):
            start = (r)*batch_size
            end = (r+1)*batch_size
            x_flatten = xtrain[start:end].reshape(batch_size,feature_size).astype('float32')
            x_flatten /= 255
            label = list(ytrain[start:end])
            label_m = np.empty((batch_size, 10))
            for i in range(len(label)):
                label_inter = one_hot_vec(label[i], 10)
                label_m[i, :] = label_inter.squeeze()
            pred = tensor_NN.forward(x_flatten)
            loss = log_loss_tensor(pred,label_m)
            total_loss_x[k].append(loss)
            if k == 0:
                total_loss_y.append(end)
            tensor_NN.backward(x_flatten,label_m,lr)

def plot_lr(x,y,lr_list,title,xtitle,ytitle,output):
    x = x
    y = y
    for i in range(len(lr_list)):
        plt.plot(y, x[i], label=lr_list[i],lw=0.1)
    plt.title(title)
    plt.xlabel(xtitle)
    plt.ylabel(ytitle)
    plt.legend()
    plt.savefig(output)
    plt.show()

#Plot the loss curve of figure 4
#batchsize=1
plot_lr(total_loss_x,total_loss_y,lr_list,"Loss curve (Batch size = 1)",
"Number of samples","Average Loss over past 1 sample","07-03-batchsize=1.png")
```

```
#batchsize=5
plot_lr(total_loss_x,total_loss_y,lr_list,"Loss curve (Batch size = 5)",
"Number of samples","Average Loss over past 5 samples","07-03-batchsize=5.png")

#batchsize=16
plot_lr(total_loss_x,total_loss_y,lr_list,"Loss curve (Batch size = 16)",
"Number of samples","Average Loss over past 16 samples","07-03-batchsize=16.png")

#batchsize = 32
plot_lr(total_loss_x,total_loss_y,lr_list,"Loss curve (Batch size = 32)",
"Number of samples","Average Loss over past 32 samples","07-03-batchsize=32.png")
```

Question 7-4

The training and plot code for question 7-4:

```
#Load the final version of MNIST data
(xtrain_final, ytrain_final), (xval_final, yval_final), num_cls = load_mnist(final=True,
flatten=True)

#Set the final hyperparamters
feature_size = 784
layer_1_size = 300
layer_2_size = 10
batch_size = 16
lr = 0.03

#Start to training the full training_data with 5 epochs

tensor_NN = tensorNN(feature_size,layer_1_size,layer_2_size,batch_size)
train_x = []
train_y = []
accuracy_all_epoch = []
loss_all_epoch = []
rounds = int(xtrain_final.shape[0]/batch_size)
epoch = 5
for k in range(epoch):
    print("epoch:",i)
    accuracy = 0
    loss_epoch = 0
    for r in range(rounds):
            start = (r) * batch_size
            end = (r+1) * batch_size
            x_flatten = xtrain_final[start:end].reshape(batch_size,feature_size).astype('float32')
            x_flatten /= 255

            label = list(ytrain_final[start:end])
            label_m = np.empty((batch_size, layer_2_size))
            for i in range(len(label)):
                label_inter = one_hot_vec(label[i], 10)
                label_m[i, :] = label_inter.squeeze()

            pred = tensor_NN.forward(x_flatten)
            loss = log_loss_tensor(pred,label_m)
            tensor_NN.backward(x_flatten,label_m,lr)
```

```
            train_x.append(end)
            train_y.append(loss)
            pred_max = np.argmax(pred, axis=1)
            label_max = np.argmax(label_m, axis=1)
            accuracy += np.mean(pred_max == label_max)
            loss_epoch += loss
            print("epoch",k+1,"sampling",end,"/",xtrain_final.shape[0])
    loss_all_epoch.append(loss_epoch / rounds)
    accuracy_all_epoch.append(accuracy / rounds)

#Continue to use this model to test the full validation data in one epoch
val_x = []
val_y = []
accuracy = 0
rounds = int(xval_final.shape[0]/batch_size)
for i in range(1):
    print("epoch:", i)
    for r in range(rounds):
            start = (r)*batch_size
            end = (r+1)*batch_size
            x_flatten = xval_final[start:end].reshape(batch_size,feature_size).astype('float32')
            x_flatten /= 255 #normalization

            label = list(yval_final[start:end])
            label_m = np.empty((batch_size, layer_2_size))
            for i in range(len(label)):
                label_inter = one_hot_vec(label[i], 10)
                label_m[i, :] = label_inter.squeeze()

            pred = tensor_NN.forward(x_flatten)
            loss = log_loss_tensor(pred,label_m)
            tensor_NN.backward(x_flatten,label_m,lr)
            val_x.append(end)
            val_y.append(loss)
            pred_max = np.argmax(pred, axis=1)
            label_max = np.argmax(label_m, axis=1)
            accuracy += np.mean(pred_max == label_max)
    accuracy /= rounds
accuracy = round(accuracy,3)

#Define the plot function
def plot_training_epoch(loss_input,acc_input,title,xtitle,ytitle):
    epochs = [1, 2, 3, 4, 5]

    plt.plot(epochs, loss_input, color='blue', marker='o', label='Training loss')
    plt.plot(epochs, acc_input, color='red', marker='x', label='Training accuracy')

    plt.title(title)
    plt.xlabel(xtitle)
    plt.ylabel(ytitle)
    plt.legend()
    plt.savefig('07-04-training.png')
    plt.show()
```

```python
def plot_loss_final(x,y,title,xtitle,ytitle,acc,output):

    xaxis = x
    yaxis = y
    plt.plot(xaxis,yaxis,lw=0.5,label = "Validation data")
    y_lim_max = 1

    plt.ylim(0, y_lim_max)
    a = "Accuracy:" + str(acc)

    plt.text(6000, 0.8, a, fontsize=12)
    plt.title(title)
    plt.xlabel(xtitle)
    plt.ylabel(ytitle)
    plt.legend()
    output = output
    plt.savefig(output)
    plt.show()

#Plot the loss and accuracy curve of figure 6 (left side)
plot_training_epoch(loss_all_epoch,accuracy_all_epoch,
"Loss and accuracy of the full raninging data(n=60000)","Epoch",
"Average loss / accuracy over past one epoch")

#Plot the loss curve of figure 6 (right side)
plot_loss_final(val_x,val_y,"Loss curve of the full validation data(n=10000)",
"Number of samples","Average loss over past 16 samples",accuracy,"07-04.validation.png")
```