Homework1

Name: Youming Zhang

NetID: yz7399

Question 1

(2 points) *Linear regression with non-standard losses*. In class we derived an analytical expression for the optimal linear regression model using the least squares loss. **If *X*** is the matrix of *n* training data points (stacked row-wise) and y is the vector of their corresponding labels, then:

a. Using matrix/vector notation, write down a loss function that measures the training error in terms of the l_1-norm. Write down the sizes of all matrices/vectors.

Solution:

We can use Mean Absolute Error(MAE) to represent loss function using l_1-*norm*

given

$y = (y_1, \cdots, y_n)^T$ is an $n \times 1$ vector

$X = (x_1^T, \cdots, x_n^T)$ is an $n \times d$ matrix

$w = (w_1, \cdots, w_d)$ is an $d \times 1$ vector

$L(w) = |y - Xw|$

b. Can you simply write down the optimal linear model in closed form, as we did for standard linear regression? If not, why not?

solution:

No, we may not get the optimal linear model in closed form,

because $\frac{d(L(w))}{dw} = -X^T * sign(y - Xw)$, and we cannot do linear optimization based on it.

Question2

(3 points) *Expressivity of neural networks*. Recall that the functional form for a single neuron is given by $y = \sigma(\langle w,x \rangle + b, 0)$, where *x* is the input and *y* is the output. In this exercise, assume that *x* and *y* are 1-dimensional (i.e., they are both just real-valued scalars) and σ is the unit step activation. We will use multiple layers of such neurons to approximate pretty much any function *f*. There is no learning/training required for this problem; you should be able to guess/derive the weights and biases of the networks by hand.

a.A *box* function with height *h* and width δ is the function $f(x) = h$ for $0 < x < \delta$ and 0 otherwise. Show that a simple neural network with 2 hidden neurons with step activations can realize this function. Draw this network and identify all the weights and biases. (Assume that the output neuron only sums up inputs and does not have a nonlinearity.)

Solution:

We can use difference between two step functions to realize the box function:

$f(x) = h\sigma(x) - h\sigma(x - \delta).$

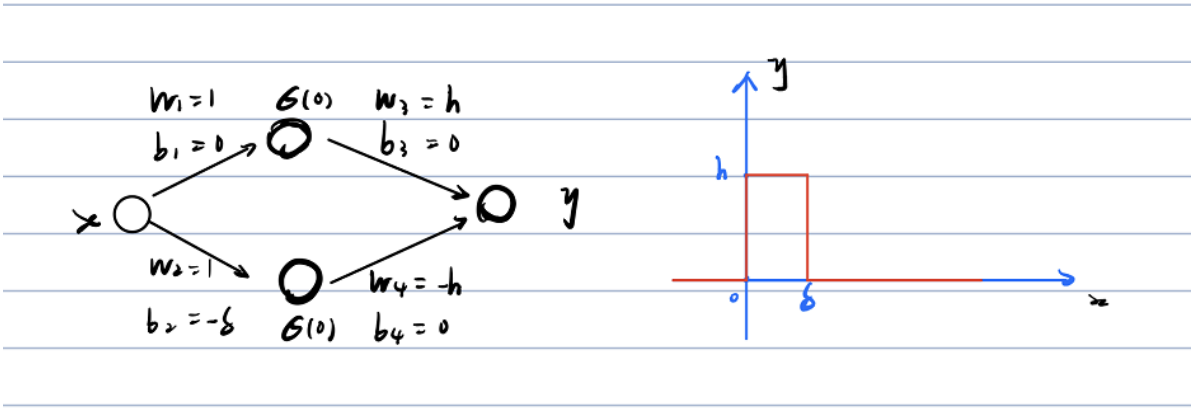


Figure1: Boxfunction network

b.Now suppose that *f* is *any arbitrary, smooth, bounded* function defined over an interval [− *B,B*]. (You can ignore what happens to the function outside this interval, or just assume it is zero). Use part a to show that this function can be closely approximated by a neural network with a hidden layer of neurons. You don't need a rigorous mathematical proof here; a handwavy argument or even a figure is okay here, as long as you convey the right intuition.

Solution:

As you can see in the pic below. each box function can be described used 2 neurons which is exactly the same as we illustrate in part a. We can adjust the width, height of the box by changing bias of the hidden layer and the weight value of the output layer respectively. Last we can add those functions by adding putting each group of two neurons to the hidden layer. Therefore, the function can be expressed by a single hidden layer neural network with multiple neurons.

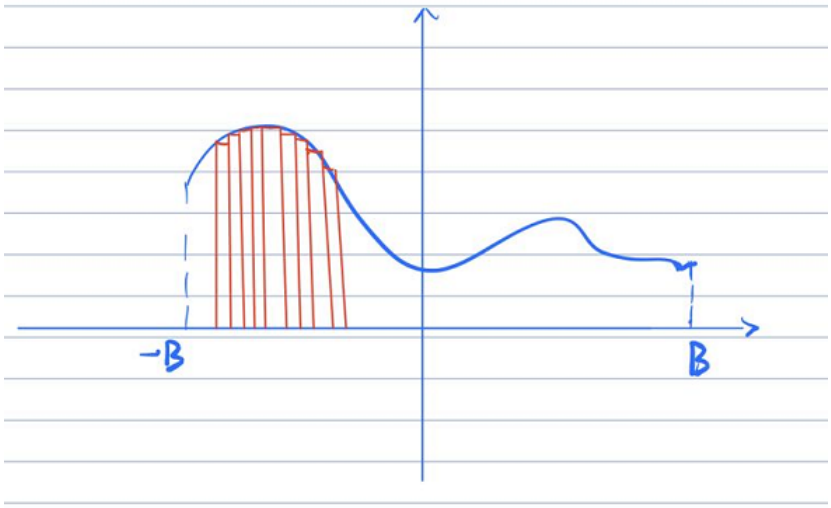


Figure2: sum of boxfunction to produce any function in 2d

c. Do you think the argument in part b can be extended to the case of *d* -dimensional inputs? (i.e., where the input *x* is a vector – think of it as an image, or text query, etc). If yes, comment on potential practical issues involved in defining such networks. If not, explain why not.

Solution:

Yes, the difference is the input changes from 1 dimensional to d dimensional, so the output is also changing to d dimensional. So we can conclude that given enough neurons we can represent any function using one single hidden layer, but the layer can be extremely giant so it may fail to learn and generalize correctly.

Question3

(3 points) Calculating gradients. Suppose that *z* is a vector with *n* elements. We would like to compute the gradient of *y* =softmax(*z*). Show that the Jacobian of *y* with respect to *z* , *J* , is given by the $J_{ij} = \frac{\partial y_i}{\partial z_j} = y_i(\delta_{ij} - y_j)$. where δ_{ij} is the Dirac delta, i.e.., 1 if $i = j$ and 0 else. *Hint: Your algebra could be simplified if you try computing the log derivative,* $\frac{\partial \log y_i}{\partial z_j}$.

Solution:

$y_i = \frac{e^{z_i}}{\sum_{j=1}^{n} e^{z_j}}$

$Loss = -\sum_{i=1}^{n} t_i \log y_i$ here, t_i is ground truth value, y_i is value producted by softmax.

$\log y_i = \log e^{z_i} - \log \sum_{j=1}^{n} e^{z_j} = z_i - \log \sum_{j=1}^{n} e^{z_i}$

$\frac{\partial \log y_i}{\partial z_j} = \frac{\partial \log y_i}{\partial y_i} \frac{\partial y_i}{\partial z_j} = \frac{1}{y_i} \frac{\partial y_i}{\partial z_j}$

$loss_i = -\log y_i = \log \sum_{j=1}^{n} e^{z_j} - z_i$

we want to minimize $loss_i$, then we need to calculate the gradient with respect to z_i

$\frac{\partial loss_i}{\partial z_i} = \frac{e^{z_i}}{\sum_{j=1}^{n} e^{z_j}} - 1 = y_i - 1$

And to further calculate Jacobian of y with respect to z, we need to consider the gradient with respect to z_j where $i \neq j$

$\frac{\partial loss_i}{\partial z_j}$ = $\frac{e^{z_i}}{\sum_{j=1}^{n} e^{z_j}}$ = y_i

Therefore, putting both cases using the Dirac delta and consider $\frac{\partial y_i}{\partial z_j} = y_i \frac{\partial \log y_i}{\partial z_j}$ we can get:

$J_{ij} = y_i(\delta_{ij} - y_j)$

Coding Problems:

Question4

```
import numpy as np
import torch
import torchvision
```

Download the data

We will download the data of MNIST and transfer it into tensor format

```
trainingdata =
torchvision.datasets.FashionMNIST('./FashionMNIST/',train=True,download=True,tra
nsform=torchvision.transforms.ToTensor())
testdata =
torchvision.datasets.FashionMNIST('./FashionMNIST/',train=False,download=True,tr
ansform=torchvision.transforms.ToTensor())
```

```
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/train-
images-idx3-ubyte.gz
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/train-
images-idx3-ubyte.gz to ./FashionMNIST/FashionMNIST/raw/train-images-idx3-
ubyte.gz
```

```
  0%|          | 0/26421880 [00:00<?, ?it/s]
```

```
Extracting ./FashionMNIST/FashionMNIST/raw/train-images-idx3-ubyte.gz to
./FashionMNIST/FashionMNIST/raw

Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/train-
labels-idx1-ubyte.gz
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/train-
labels-idx1-ubyte.gz to ./FashionMNIST/FashionMNIST/raw/train-labels-idx1-
ubyte.gz
```

```
  0%|          | 0/29515 [00:00<?, ?it/s]
```

```
Extracting ./FashionMNIST/FashionMNIST/raw/train-labels-idx1-ubyte.gz to
./FashionMNIST/FashionMNIST/raw

Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/t10k-
images-idx3-ubyte.gz
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/t10k-
images-idx3-ubyte.gz to ./FashionMNIST/FashionMNIST/raw/t10k-images-idx3-
ubyte.gz
```

```
  0%|          | 0/4422102 [00:00<?, ?it/s]
```

```
Extracting ./FashionMNIST/FashionMNIST/raw/t10k-images-idx3-ubyte.gz to
./FashionMNIST/FashionMNIST/raw

Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/t10k-
labels-idx1-ubyte.gz
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/t10k-
labels-idx1-ubyte.gz to ./FashionMNIST/FashionMNIST/raw/t10k-labels-idx1-
ubyte.gz
```

```
  0%|          | 0/5148 [00:00<?, ?it/s]
```

```
Extracting ./FashionMNIST/FashionMNIST/raw/t10k-labels-idx1-ubyte.gz to
./FashionMNIST/FashionMNIST/raw
```

Let's quickly check that everything has been downloaded.

```
print(len(trainingdata))
print(len(testdata))
```

```
60000
10000
```

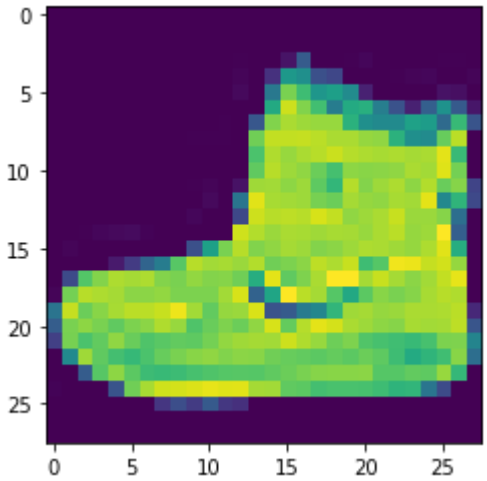
Some of you might know that this is the same size as the (very commonly used) MNIST dataset. Let's plot some images.

```
image, label = trainingdata[0]
print(image.shape, label)
```

```
torch.Size([1, 28, 28]) 9
```

We cannot directly plot the 'image' object since it is a Torch tensor, so let's convert it back into a numpy array before displaying it. We will use matplotlib to show images.

```
import matplotlib.pyplot as plt
%matplotlib inline
plt.imshow(image.squeeze().numpy())
plt.show()
```



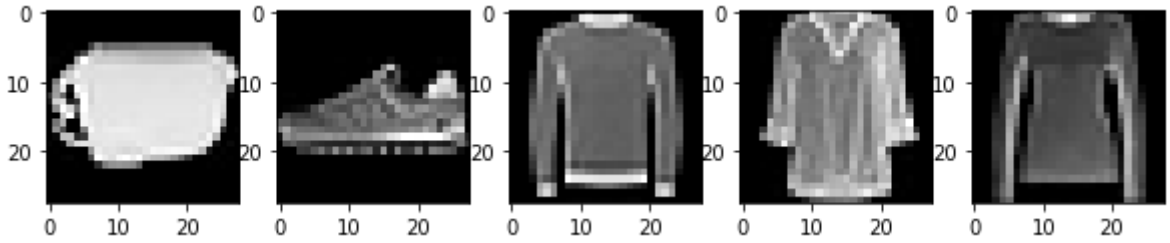
OK, that kinda sorta looks like a shoe? FashionMNIST is basically a bunch of black-white thumbnail images of different pieces of clothing. Let's try and plot a few of them. To step through the dataset, it will be helpful to construct a *data loader* object; we will use this while training our model below as well.

```
trainDataLoader =
torch.utils.data.DataLoader(trainingdata,batch_size=64,shuffle=True)
testDataLoader =
torch.utils.data.DataLoader(testdata,batch_size=64,shuffle=False)
```

Let's now use the DataLoader object to spit out a few images from the dataset.

```
images, labels = iter(trainDataLoader).next()
print(labels.shape[0])
plt.figure(figsize=(10,4))
for index in np.arange(0,5):
  plt.subplot(1,5,index+1)
  plt.imshow(images[index].squeeze().numpy(),cmap=plt.cm.gray)
```

```
64
```



OK! Time to set up our model.

In the initialization function, we build up 3 hidden layers with 256, 128 ,64 neurons respectively and 1 output layer with input size 64, output size 10.

```
class LinearReg(torch.nn.Module):
```

```
  def __init__(self):
    super(LinearReg, self).__init__()
    self.hidden_layer1 = torch.nn.Linear(28*28,256)
    self.hidden_layer2 = torch.nn.Linear(256,128)
    self.hidden_layer3 = torch.nn.Linear(128,64)
    self.output_layer = torch.nn.Linear(64,10)

  def forward(self, x):
    x = x.view(-1,28*28)
    transformed_x = torch.nn.functional.relu(self.hidden_layer1(x))
    transformed_x = torch.nn.functional.relu(self.hidden_layer2(transformed_x))
    transformed_x = torch.nn.functional.relu(self.hidden_layer3(transformed_x))
    transformed_x = self.output_layer(transformed_x)
    return transformed_x

net = LinearReg().cuda()
Loss = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(net.parameters(), lr=0.01)
```

Cool! Everything is set up. Let's now train our network.

```
train_loss_history = []
test_loss_history = []
matchSample = 0
total = 0

for epoch in range(60):
  train_loss = 0.0
  test_loss = 0.0
  for i, data in enumerate(trainDataLoader): # every time we use
trainDataLoader, it will read 64 sample from 60000 samples
    images, labels = data
    images = images.cuda()
    labels = labels.cuda()
    optimizer.zero_grad()
    predicted_output = net(images)
    fit = Loss(predicted_output,labels)
    fit.backward()
    optimizer.step()
    train_loss += fit.item()
  for i, data in enumerate(testDataLoader):
    with torch.no_grad():
      images, labels = data
      images = images.cuda()
      labels = labels.cuda()
      predicted_output = net(images)
      fit = Loss(predicted_output,labels)
      test_loss += fit.item()
      predicted = torch.max(predicted_output.data,1)[1]
      matchSample += (predicted == labels).sum().item()
      total += labels.shape[0]
  train_loss = train_loss/len(trainDataLoader)
  test_loss = test_loss/len(testDataLoader)
  train_loss_history.append(train_loss)
  test_loss_history.append(test_loss)
```

```
    print('Epoch %s, Train loss %s, Test loss %s'%(epoch, train_loss, test_loss))
```

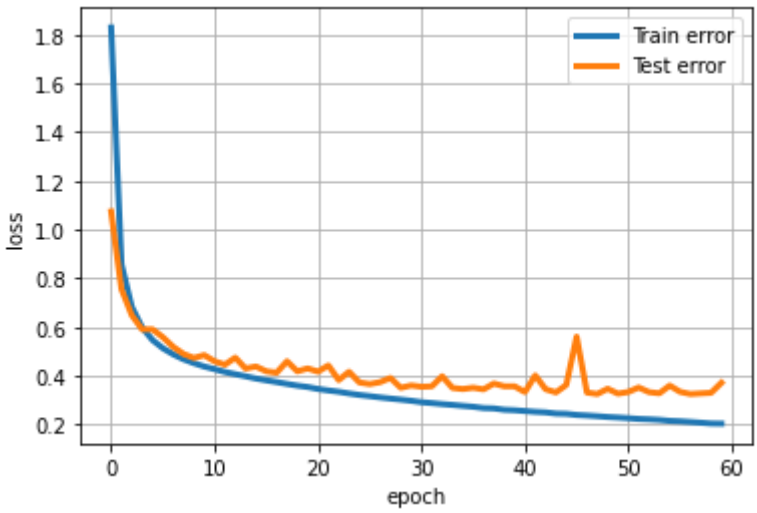
```
Epoch 0, Train loss 1.8291424095376467, Test loss 1.0742837338690545
Epoch 1, Train loss 0.8578743217215101, Test loss 0.7557273949407468
Epoch 2, Train loss 0.6829571581598538, Test loss 0.6487537109927767
Epoch 3, Train loss 0.5985138725433776, Test loss 0.5907559206910954
Epoch 4, Train loss 0.5469908123490399, Test loss 0.5909633285300747
Epoch 5, Train loss 0.513526411993163, Test loss 0.5588198158012074
Epoch 6, Train loss 0.4884722268562327, Test loss 0.5192150794396735
Epoch 7, Train loss 0.46828202877852965, Test loss 0.48984338827193924
Epoch 8, Train loss 0.45231327258828863, Test loss 0.4736497064304959
Epoch 9, Train loss 0.4384900077955047, Test loss 0.4851857104878517
Epoch 10, Train loss 0.4276218363629984, Test loss 0.45991426981558464
Epoch 11, Train loss 0.4160708460980641, Test loss 0.44607398008844656
Epoch 12, Train loss 0.4071153639666816, Test loss 0.47497020206254
Epoch 13, Train loss 0.39842337836970143, Test loss 0.43033252333759503
Epoch 14, Train loss 0.3891984690735335, Test loss 0.4386184955858121
Epoch 15, Train loss 0.381807627104747, Test loss 0.4189257849553588
Epoch 16, Train loss 0.37468297134584455, Test loss 0.4131521818933973
Epoch 17, Train loss 0.3671426185126752, Test loss 0.4599229693412781
Epoch 18, Train loss 0.36003396587013436, Test loss 0.4188010385082026
Epoch 19, Train loss 0.35443577052814873, Test loss 0.4307574634529223
Epoch 20, Train loss 0.34639814008337094, Test loss 0.41743936821533617
Epoch 21, Train loss 0.3403589428583188, Test loss 0.44306587214302867
Epoch 22, Train loss 0.3349380204751929, Test loss 0.3830624422070327
Epoch 23, Train loss 0.3279073566738476, Test loss 0.4168383112758588
Epoch 24, Train loss 0.32181296455485225, Test loss 0.37305442599733923
Epoch 25, Train loss 0.31698396500906967, Test loss 0.36649203860456014
Epoch 26, Train loss 0.3117383590210348, Test loss 0.37430047542805883
Epoch 27, Train loss 0.30711960910892944, Test loss 0.3917199937970775
Epoch 28, Train loss 0.3027531934826613, Test loss 0.3512899433351626
Epoch 29, Train loss 0.2980483406102225, Test loss 0.360885212091124
Epoch 30, Train loss 0.2920724952986627, Test loss 0.35553761139796797
Epoch 31, Train loss 0.28851301385077843, Test loss 0.3571920289070743
Epoch 32, Train loss 0.28442543407461285, Test loss 0.39847164292624043
Epoch 33, Train loss 0.28104058213071276, Test loss 0.35065230252636465
Epoch 34, Train loss 0.277069719861756, Test loss 0.34614884331347834
Epoch 35, Train loss 0.27352399722154713, Test loss 0.3515710236540266
Epoch 36, Train loss 0.2680646366894499, Test loss 0.34485116468113697
Epoch 37, Train loss 0.2667891388731216, Test loss 0.36878075910981295
Epoch 38, Train loss 0.26107427353925033, Test loss 0.35737416537324335
Epoch 39, Train loss 0.25928475169230625, Test loss 0.3573936345945498
Epoch 40, Train loss 0.25589264803794404, Test loss 0.3338349426438095
Epoch 41, Train loss 0.2526311063007124, Test loss 0.401784341426412
Epoch 42, Train loss 0.25049315674155, Test loss 0.34552347081102386
Epoch 43, Train loss 0.2455826408541533, Test loss 0.3323235987288177
Epoch 44, Train loss 0.24447010598504848, Test loss 0.36282950929205887
Epoch 45, Train loss 0.23978330083350255, Test loss 0.560956140612341
Epoch 46, Train loss 0.23753064219504277, Test loss 0.33179455136607405
Epoch 47, Train loss 0.23472793238249415, Test loss 0.32585127419157395
Epoch 48, Train loss 0.23149643138622933, Test loss 0.34719340664565945
Epoch 49, Train loss 0.22926783668937714, Test loss 0.32723322354114737
Epoch 50, Train loss 0.22661949018997424, Test loss 0.334168549413514
Epoch 51, Train loss 0.22354833158586962, Test loss 0.3525407247862239
Epoch 52, Train loss 0.2219080931421663, Test loss 0.33346878931780527
```

```
Epoch 53, Train loss 0.219591640766019, Test loss 0.3286011695482169
Epoch 54, Train loss 0.21559945587267373, Test loss 0.3604054083679892
Epoch 55, Train loss 0.21383147772504832, Test loss 0.33471677229282965
Epoch 56, Train loss 0.21101167082770675, Test loss 0.3256832432879764
Epoch 57, Train loss 0.2080324299490528, Test loss 0.3282862573767164
Epoch 58, Train loss 0.20483699428247237, Test loss 0.33060618238464284
Epoch 59, Train loss 0.2042275173927167, Test loss 0.37198529401972036
```

Let's see how we did! We have tracked the losses so let's plot it.

```
plt.plot(range(60),train_loss_history,'-',linewidth=3,label='Train error')
plt.plot(range(60),test_loss_history,'-',linewidth=3,label='Test error')
plt.xlabel('epoch')
plt.ylabel('loss')
plt.grid(True)
plt.legend()
acc = 100.0 * matchSample / total
print("accuracy:",acc)
```

```
accuracy: 85.00866666666667
```

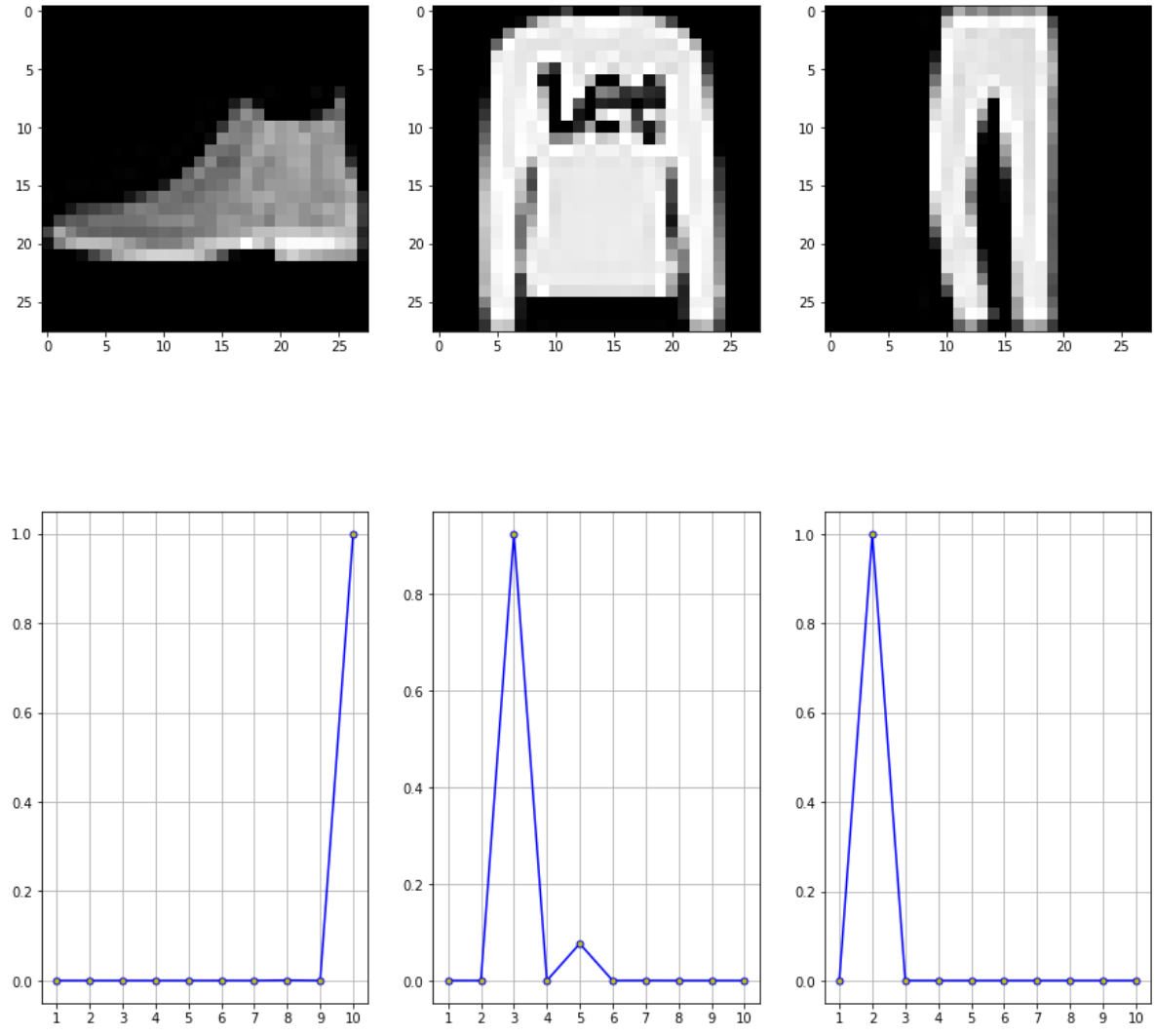


We can see that first plot is a shoe, the second is a cloth, the last one is a trousers from the first row.

As we see in the second row. The first graph is most likely belongs to label 10, the second graph is most likely belongs to label 3, the last graph is most likely belongs to label 2

```
from pylab import xticks,yticks
plt.figure(figsize=(15,15))
for index in np.arange(0,3):
  image = testdata[index][0]
  image = image.cuda()
  plt.subplot(2,3,index+1)
  plt.imshow(image.cpu().squeeze().numpy(),cmap=plt.cm.gray)
  predicted_output = net(image)
  softmax_output = torch.nn.functional.softmax(predicted_output,1)
```

```
  # print(softmax_output)
  # print(predicted_output)
  plt.subplot(2,3,index+4)
  xticks(np.linspace(1,10,10,endpoint=True))
  plt.ylim = [0,1]
  plt.xlabel = 'class'
  plt.ylabel = 'prob'
  x = np.linspace(1,10,10)
  y = softmax_output.cpu().squeeze().detach().numpy()
  # print(softmax_output.cpu().squeeze().detach().numpy())
  plt.plot(x,y,label = 'prob distribution',color =
'b',marker='o',markerfacecolor='y',markersize=5)
  # for i,j in zip(x,y):
  #   if j == 0:continue
  #   plt.text(i,j,j,ha='center',va='bottom',fontsize=15)
  plt.grid()
```



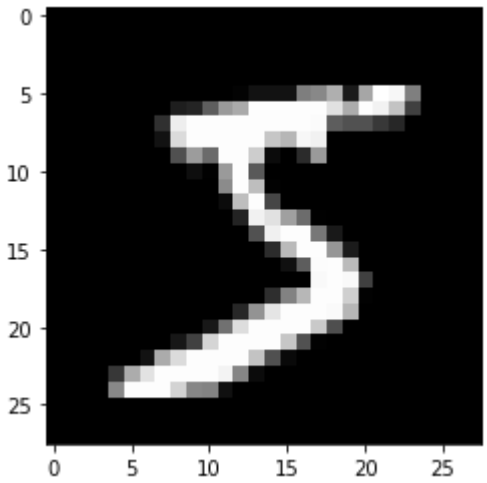
Question 5

In this problem we will train a neural network from scratch using numpy. In practice, you will never need to do this (you'd just use TensorFlow or PyTorch). But hopefully this will give us a sense of what's happening under the hood.

For training/testing, we will use the standard MNIST benchmark consisting of images of handwritten images.

In the second demo, we worked with autodiff. Autodiff enables us to implicitly store how to calculate the gradient when we call backward. We implemented some basic operations (addition, multiplication, power, and ReLU). In this homework problem, you will implement backprop for more complicated operations directly. Instead of using autodiff, you will manually compute the gradient of the loss function for each parameter.

```
import tensorflow as tf
import matplotlib.pyplot as plt

(x_train, y_train), (x_test, y_test) =
tf.keras.datasets.mnist.load_data(path="mnist.npz")

plt.imshow(x_train[0],cmap='gray');
```



Loading MNIST is the only place where we will use TensorFlow; the rest of the code will be pure numpy.

Let us now set up a few helper functions. We will use sigmoid activations for neurons, the softmax activation for the last layer, and the cross entropy loss.

```
import numpy as np

def sigmoid(x):
  # Numerically stable sigmoid function based on
  # http://timvieira.github.io/blog/post/2014/02/11/exp-normalize-trick/

  x = np.clip(x, -500, 500) # We get an overflow warning without this

  return np.where(
    x >= 0,
    1 / (1 + np.exp(-x)),
    np.exp(x) / (1 + np.exp(x))
  )
```

```
def dsigmoid(x): # Derivative of sigmoid
  return sigmoid(x) * (1 - sigmoid(x))

def softmax(x):
  # Numerically stable softmax based on (same source as sigmoid)
  # http://timvieira.github.io/blog/post/2014/02/11/exp-normalize-trick/
  b = x.max()
  y = np.exp(x - b)
  return y / y.sum()

def cross_entropy_loss(y, yHat):
  return -np.sum(y * np.log(yHat))

def integer_to_one_hot(x, max):
  # x: integer to convert to one hot encoding
  # max: the size of the one hot encoded array
  result = np.zeros(10)
  result[x] = 1
  return result
```

OK, we are now ready to build and train our model. The input is an image of size 28x28, and the output is one of 10 classes. So, first:

Q1. Initialize a 2-hidden layer neural network with 32 neurons in each hidden layer, i.e., your layer sizes should be:

784 -> 32 -> 32 -> 10

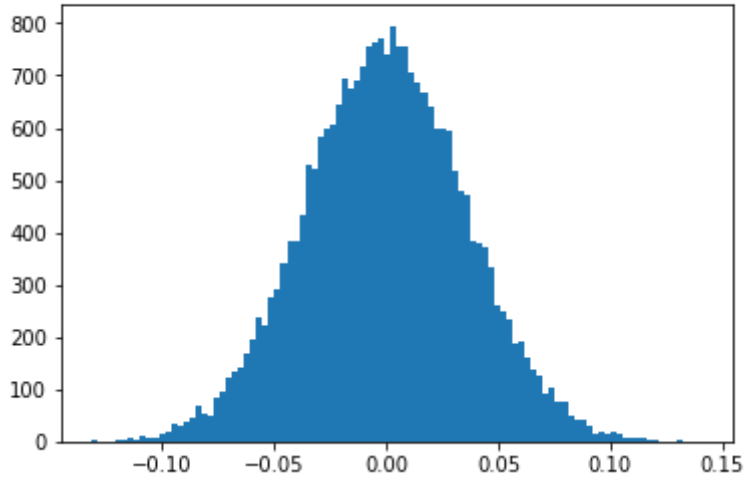
If the layer is $n_{in} \times n_{out}$ your layer weights should be initialized by sampling from a normal distribution with mean zero and variance $1/\max(n_{in}, n_{out})$.

```
import math

# Initialize weights of each layer with a normal distribution of mean 0 and
# standard deviation 1/sqrt(n), where n is the number of inputs.
# This means the weighted input will be a random variable itself with mean
# 0 and standard deviation close to 1 (if biases are initialized as 0, standard
# deviation will be exactly 1)

from numpy.random import default_rng

rng = default_rng(80085)

# Q1. Fill initialization code here.
# ...

weights = [
    rng.normal(0,1/math.sqrt(784),(32,784)),#生成32行784列数据
    rng.normal(0,1/math.sqrt(32),(32,32)),
    rng.normal(0,1/math.sqrt(32),(10,32))
]
print(len(weights))
biases = [np.zeros(32),np.zeros(32),np.zeros(10)]

#Plot the weights to check out the distribution
print("Weight distribution per layer:")
```

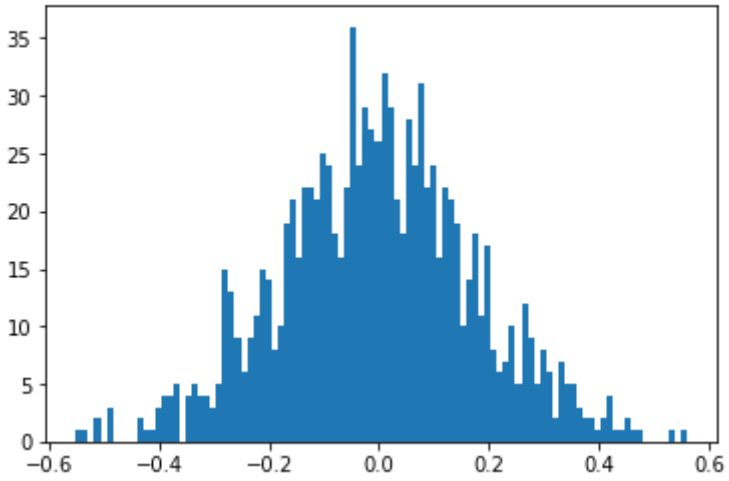
```
for index, layer in enumerate(weights):
  plt.figure()
  plt.suptitle(
    "Layer " + str(index + 1) + " with " + str(layer.shape[0]) +
    " neurons, " + str(layer.shape[1]) + " inputs each (" + str(layer.size) +
    " weights in total)"
  )
  plt.hist(layer.flatten(), bins=100);
```

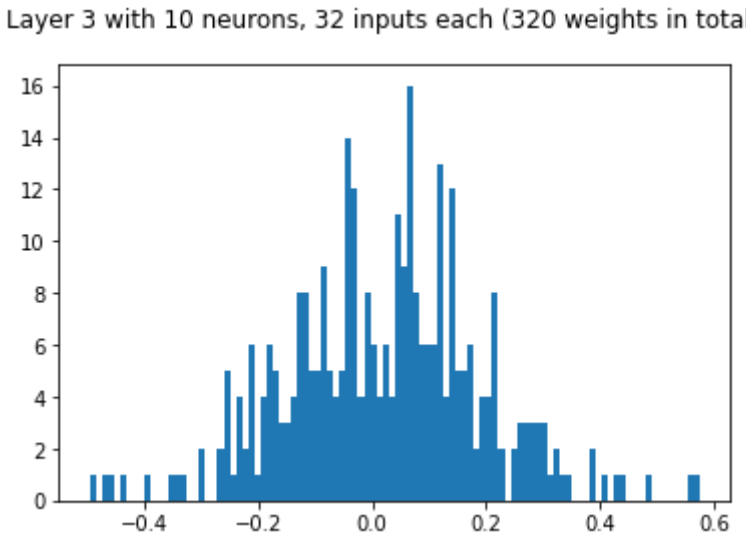
```
3
Weight distribution per layer:
```

Layer 1 with 32 neurons, 784 inputs each (25088 weights in total)



Layer 2 with 32 neurons, 32 inputs each (1024 weights in total)





Next, we will set up the forward pass. We will implement this by looping over the layers and successively computing the activations of each layer.

Q2. Implement the forward pass for a single sample, and for the entire dataset.

Right now, your network weights should be random, so doing a forward pass with the data should not give you any meaningful information. Therefore, in the last line, when you calculate test accuracy, it should be somewhere around 1/10 (i.e., a random guess).

```
def feed_forward_sample(sample, y):
  """ Forward pass through the neural network.
    Inputs:
      sample: 1D numpy array. The input sample (an MNIST digit).
      label: An integer from 0 to 9.

    Returns: the cross entropy loss, most likely class
  """
  # Q2. Fill code here.
  # ...
  a = sample.flatten() #perform dimensionality reduction
  for index, w in enumerate(weights):
    z = np.matmul(w,a) + biases[index]
    if index < len(weights) - 1:
      a = sigmoid(z)
    else:
      a = softmax(z)

  #calculate loss
  one_hot_y = integer_to_one_hot(y,10)
  loss = cross_entropy_loss(one_hot_y, a)

  #convert activations to one hot encodes guess
  # one_hot_list = np.zeros_like(a)
  # one_hot_list[np.argmax(a)] = 1
  # one_hot_guess = one_hot_list.index(max(one_hot_list))
  one_hot_guess = np.zeros_like(a)
  index = np.argmax(a)
  one_hot_guess[index] = 1
  return loss, one_hot_guess
```

```
def feed_forward_dataset(x, y):
  losses = np.empty(x.shape[0])
  one_hot_guesses = np.empty((x.shape[0], 10))


  # ...
  # Q2. Fill code here to calculate losses, one_hot_guesses
  # ...
  for i in range(x.shape[0]):
    if i == 0 or ((i+1) % 10000 == 0):
      print(i + 1, "/", x.shape[0], "(", format(((i + 1) / x.shape[0]) * 100,
".2f"), "%)")
    losses[i], one_hot_guesses[i] = feed_forward_sample(x[i], y[i])
  print("\nAverage loss:",np.round(np.average(losses),decimals = 2))

  y_one_hot = np.zeros((y.size, 10))
  y_one_hot[np.arange(y.size), y] = 1

  correct_guesses = np.sum(y_one_hot * one_hot_guesses)
  correct_guess_percent = format((correct_guesses / y.shape[0]) * 100, ".2f")

  print("\nAverage loss:", np.round(np.average(losses), decimals=2))
  print("Accuracy (# of correct guesses):", correct_guesses, "/", y.shape[0], "
(", correct_guess_percent, "%)")

def feed_forward_training_data():
  print("Feeding forward all training data...")
  feed_forward_dataset(x_train, y_train)
  print("")

def feed_forward_test_data():
  print("Feeding forward all test data...")
  feed_forward_dataset(x_test, y_test)
  print("")

feed_forward_test_data()
```

```
Feeding forward all test data...
1 / 10000 ( 0.01 %)
10000 / 10000 ( 100.00 %)

Average loss: 2.36

Average loss: 2.36
Accuracy (# of correct guesses): 994.0 / 10000 ( 9.94 %)
```

OK, now we will implement the backward pass using backpropagation. We will keep it simple and just do training sample-by-sample (no minibatching, no randomness).

Q3: Compute the gradient of all the weights and biases by backpropagating derivatives all the way from the output to the first layer.

```
def train_one_sample(sample, y, learning_rate=0.003):
  a = sample.flatten()

  # We will store each layer's activations to calculate gradient
  activations = []
  # Q3. This should be the same as what you did in feed_forward_sample above.
  # ...
  # Forward pass
  for i, w in enumerate(weights):# Each w is a 2D weight matrix
    z = np.matmul(w,a) + biases[i]
    if( i < len(weights) - 1):#activation is sigmoid, the last layer we use
softmax to do classification
      a = sigmoid(z)
    else:
      a = softmax(z)
    activations.append(a)
  # Calculate the loss
  one_hot_y = integer_to_one_hot(y,10)
  loss = cross_entropy_loss(one_hot_y,a)
  # Calculate one hot encoded guess
  one_hot_guess = np.zeros_like(a)
  one_hot_guess[np.argmax(a)] = 1
  correct_guess = (np.sum(one_hot_y * one_hot_guess) == 1)
  # correct_guess = (np.sum(np.matmul(one_hot_guess,one_hot_y) == 1))
  weight_gradients = [None] * len(weights)
  bias_gradients = [None] * len(weights)
  activation_gradients = [None] * (len(weights) - 1)#the last layer is for
classification
  # Backward pass
  # Q3. Implement backpropagation by backward-stepping gradients through each
layer.
  # You may need to be careful to make sure your Jacobian matrices are the right
shape.
  # At the end, you should get two vectors: weight_gradients and bias_gradients.
  # ...
  for i in range(len(weights)-1,-1,-1):#Traverse layers in reverse
    last_layer = i == len(weights) - 1
    last_activation = i == len(weights) - 2

    #
    if last_layer:
      y = one_hot_y[:, np.newaxis]
      a = activations[i][:, np.newaxis]
      a_prev = activations[i-1][:, np.newaxis]

      weight_gradients[i] = np.matmul((a-y),a_prev.T)
      bias_gradients[i] = a - y
    else:
      # gather variables, making vectors vertical
      w_next = weights[i+1]
      a_next = activations[i + 1][:, np.newaxis]
```

```
      y = one_hot_y[:, np.newaxis]
      a = activations[i][:, np.newaxis]
      if i > 0:
        a_prev = activations[i-1][:, np.newaxis]
      else:
        # Previous activation is the sample itself
        a_prev = sample.flatten()[:, np.newaxis]

      # Activation gradient
      if last_activation:
        dCda = np.matmul(w_next.T, (a_next - y))
        activation_gradients[i] = dCda
      else:
        dCda_next = activation_gradients[i+1]
        dCda = np.matmul(w_next.T, (dsigmoid(a_next) * dCda_next))
        activation_gradients[i] = dCda

      # Weights & biases gradients
      x = dsigmoid(a) * dCda
      weight_gradients[i] = np.matmul(x, a_prev.T)
      bias_gradients[i] = x
    # Update weights & biases based on your calculated gradient
    weights[i] -= weight_gradients[i] * learning_rate
    biases[i] -= bias_gradients[i].flatten() * learning_rate
```

Finally, train for 3 epochs by looping over the entire training dataset 3 times.

Q4. Train your model for 3 epochs.

```
def train_one_epoch(learning_rate=0.003):
  print("Training for one epoch over the training dataset...")

  # Q4. Write the training loop over the epoch here.
  # ...
  for i in range(x_train.shape[0]):
    if i == 0 or ((i + 1) % 10000 == 0):
      completion_percent = format(((i + 1) / x_train.shape[0]) * 100, ".2f")
      print(i + 1, "/", x_train.shape[0], "(", completion_percent, "%)")
    train_one_sample(x_train[i], y_train[i], learning_rate)
  print("Finished training.\n")


feed_forward_test_data()

def test_and_train():
  train_one_epoch()
  feed_forward_test_data()

for i in range(3):
  test_and_train()
```

```
Feeding forward all test data...
1 / 10000 ( 0.01 %)
```

```
10000 / 10000 ( 100.00 %)

Average loss: 2.36

Average loss: 2.36
Accuracy (# of correct guesses): 994.0 / 10000 ( 9.94 %)

Training for one epoch over the training dataset...
1 / 60000 ( 0.00 %)
10000 / 60000 ( 16.67 %)
20000 / 60000 ( 33.33 %)
30000 / 60000 ( 50.00 %)
40000 / 60000 ( 66.67 %)
50000 / 60000 ( 83.33 %)
60000 / 60000 ( 100.00 %)
Finished training.

Feeding forward all test data...
1 / 10000 ( 0.01 %)
10000 / 10000 ( 100.00 %)

Average loss: 0.63

Average loss: 0.63
Accuracy (# of correct guesses): 8023.0 / 10000 ( 80.23 %)

Training for one epoch over the training dataset...
1 / 60000 ( 0.00 %)
10000 / 60000 ( 16.67 %)
20000 / 60000 ( 33.33 %)
30000 / 60000 ( 50.00 %)
40000 / 60000 ( 66.67 %)
50000 / 60000 ( 83.33 %)
60000 / 60000 ( 100.00 %)
Finished training.

Feeding forward all test data...
1 / 10000 ( 0.01 %)
10000 / 10000 ( 100.00 %)

Average loss: 0.51

Average loss: 0.51
Accuracy (# of correct guesses): 8481.0 / 10000 ( 84.81 %)

Training for one epoch over the training dataset...
1 / 60000 ( 0.00 %)
10000 / 60000 ( 16.67 %)
20000 / 60000 ( 33.33 %)
30000 / 60000 ( 50.00 %)
40000 / 60000 ( 66.67 %)
50000 / 60000 ( 83.33 %)
60000 / 60000 ( 100.00 %)
Finished training.
```

```
Feeding forward all test data...
1 / 10000 ( 0.01 %)
10000 / 10000 ( 100.00 %)

Average loss: 0.49

Average loss: 0.49
Accuracy (# of correct guesses): 8563.0 / 10000 ( 85.63 %)
```

That's it!

Your code is probably very time- and memory-inefficient; that's ok. There is a ton of optimization under the hood in professional deep learning frameworks which we won't get into.

If everything is working well, you should be able to raise the accuracy from ~10% to ~70% accuracy after 3 epochs.