CS-GY 6953 / ECE-GY 7123
Deep Learning

Homework #1

Artem Shlepchenko
as14836

Problem 1

a)
The loss function that measures the training error in terms of the l1-norm (also known as Lasso regression) is given by:

$$L(w) = ||Xw - y||_1 = \sum_{i=1}^{n} |Xw_i - y_i|$$

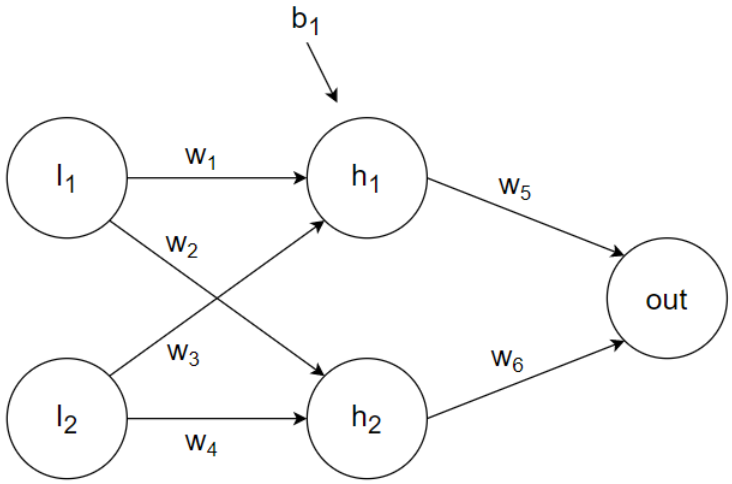
where w is the vector of weights (of size d *x 1*) for the linear model. X is the matrix of n training data points (of size n *x* d), and y is the vector of corresponding labels (of size n *x 1*).

b)
It is not possible to write down the optimal linear model for Lasso regression in the closed form because the objective function is not differentiable. However, it is possible to obtain closed form solutions for the special case of an orthonormal design matrix.
c)

- a representation
  - Linear regression.
- a measure of goodness
  - The measure of goodness is defined by the Loss function above.
- a method to optimize
  - There is no closed form.

Problem 2

a)
We can use two hidden neurons with step activations and an output neuron with no nonlinearity to realize a box function with a neural network.
The output neuron will simply sum up the inputs it receives.
We can choose the weights and biases of the network such that the two hidden neurons output h for inputs in the range $0 < x < \delta$, and 0 otherwise.
The output neuron will then receive h as input for $0 < x < \delta$, and 0 otherwise, producing the desired output $f(x) = h$ for $0 < x < \delta$, and 0 otherwise.
The diagram of this network is shown below.



Elements of the diagram:
Hidden layer (Layer 1):
$i_1 = Xw_1 + b_1$
$a_1 = z_1$

Here,

- z_1 is vectorized output of Layer 1.
- w_1 is the vectorized weight assigned to neurons of hidden layer (w_1, w_2, w_3, and w_4).
- X is the vectorized bias assigned to neurons in hidden layers (b_1 and b_2).
- $a1$ is the vectorized from of any linear function.

Output Layer (Layer 2):
$z_2 = w_2 a_1 + b_2$
$a_2 = z_2$

Problem 2 continued on next page. . .

Input for layers:

- layer 2 is output from layer 1.
- putting value of z_1 here
  - $z_2 = (w_2[w_1x + b_1] + b_2)$
  - $z_2 = [w_2 + w_1]x + [w_2b_1 + b_2]$

Now let,
$w_2w_1 = w$
$w_2b_1 + b_2 = b$
So Final Output:
$z_2 = wx + b$
Which is again a linear function.

b)
To approximate an arbitrary, smooth, bounded function defined over an interval [-B, B], we can use the same approach as in part (a), with a hidden layer of neurons.
By using enough hidden neurons, we can approximate the function to a desired level of accuracy.
This is because the functional form of a single neuron, $y = \sigma((w, x) + b, 0)$, is capable of approximating any continuous function to an arbitrary degree of accuracy, given enough weights and biases.

c)
This argument can be extended to the case of d-dimensional inputs, where x is a vector.
In this case, the weights of the hidden neurons would be d-dimensional vectors, and the input x would be transformed into a d-dimensional feature space by the hidden neurons.
The output neuron would then sum up the inputs it receives from the hidden layer to produce the final output of the network.
There are a few practical issues involved in defining such networks for d-dimensional inputs.
The first issue is the complexity of training the network, as the number of weights and biases increases with the dimensionality of the input.
The second issue is the risk of overfitting, which is when the network becomes too complex and fits the training data too closely, leading to poor generalization to unseen data.
To address these issues, techniques such as regularization and early stopping can be used to control the complexity of the network and prevent overfitting.

Problem 3

Given,

$$y = softmax(z) \tag{1}$$

Here, y and z are vectors. let us assume they are *'n'* dimensional, i.e.

$$\text{y} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix} \text{ and z} = \begin{bmatrix} z_1 \\ z_2 \\ \vdots \\ z_n \end{bmatrix}$$

From (1) we know that

$$y_i = \frac{e^{z_i}}{\sum\limits_{l=1}^{n} e^{z_l}}$$

i.e.

$$y_i = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix} \equiv \begin{bmatrix} \frac{e^{z_1}}{\sum\limits_{l=1}^{n} e^{z_l}} \\ \frac{e^{z_2}}{\sum\limits_{l=1}^{n} e^{z_l}} \\ \vdots \\ \frac{e^{z_n}}{\sum\limits_{l=1}^{n} e^{z_l}} \end{bmatrix}$$

We nee to find

$$J_{ij} = \frac{\partial y_i}{\partial z_j}$$

Note, that

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

$$\Rightarrow \frac{\partial y_i}{\partial z_j} = y_i \frac{\partial log y_i}{\partial z_j} \tag{2}$$

$$\log y_i = log\left(\frac{e^{z_i}}{\sum\limits_{l=1}^{n} e^{z_l}}\right)$$

$$= log e^{z_i} - log\left(\sum_{l=1}^{n} e^{z_l}\right) \tag{3}$$

Now,

$$\begin{aligned}
\frac{\partial log y_i}{\partial z_j} &= \frac{\partial}{\partial z_j}(log y_i) \\
&= \frac{\partial}{\partial z_j}\left(log e^{z_i} - log\left(\sum_{l=1}^{n} e^{z_l}\right)\right) \\
&= \frac{\partial log e^{z_i}}{\partial z_j} - \frac{\partial log\left(\sum_{l=1}^{n} e^{z_l}\right)}{\partial z_j} \\
&= \frac{1}{e^{z_i}}\frac{\partial e^{z_i}}{\partial z_j} - \frac{\partial}{\partial z_j} log\left(\sum_{l=1}^{n} e^{z_l}\right) \\
&= \frac{1}{e^{z_i}} e^{z_i}\frac{\partial z_i}{\partial z_j} - \frac{\partial}{\partial z_j} log\left(\sum_{l=1}^{n} e^{z_l}\right) \\
&= \frac{\partial z_i}{\partial z_j} - \frac{\partial}{\partial z_j} log\left(\sum_{l} e^{z_l}\right) \\
&= \delta_{ij} - \frac{1}{\left(\sum_{l} e^{z_l}\right)}\frac{\partial}{\partial z_j}\left(\sum_{l} e^{z_l}\right)
\end{aligned}$$

Note, that

$$\frac{\partial z_i}{\partial z_j} = 0 \text{ if } i \neq j, \text{ but, if } i = j \text{ then } \frac{\partial z_i}{\partial z_j} = \frac{\partial z_j}{\partial z_j} = 1$$

$$\text{Hence, } \frac{\partial z_i}{\partial z_j} = \delta_{ij} \leftarrow \text{dirac delta}$$

$$= \delta_{ij} - \frac{e^{z_i}}{\sum_{l} e^{z_l}}$$

where the last term is just y_i

$$= \delta_{ij} - y_i \tag{4}$$

From (2), we get

$$\begin{aligned}
\frac{\partial y_i}{\partial z_j} &= y_i \frac{\partial log y_i}{\partial z_j} \\
&= y_i(\delta_{ij} - y_j) \text{ from (4)}
\end{aligned}$$

Hence, proved.

Problem 4

Training Neural Network and Plotting Curves

```
## importing all necessary libraries
import torch
import torchvision
import torchvision.transforms as transforms
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from tqdm import tqdm

## Defining a transform function that converts our dataset into tensors
transform = transforms.Compose([transforms.ToTensor()])

## Importing our training set (we use the transform as an argument)
trainset = torchvision.datasets.FashionMNIST(root='./data', train=True, download=True,
            transform=transform)

## Creating a dataloader from training set
trainloader = torch.utils.data.DataLoader(trainset, batch_size=64, shuffle=True,
            num_workers=2)

## Importing our test set (we use the transform as an argument)
testset = torchvision.datasets.FashionMNIST(root='./data', train=False, download=True,
            transform=transform)

## Creating a dataloader from test set
testloader = torch.utils.data.DataLoader(testset, batch_size=64, shuffle=False,
            num_workers=2)

## Creating the neural network class (using nn.Module of pytorch)
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.fc1 = nn.Linear(784, 256)
        self.fc2 = nn.Linear(256, 128)
        self.fc3 = nn.Linear(128, 64)
        self.fc4 = nn.Linear(64, 10)

    def forward(self, x):
        x = x.view(-1, 784)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
```

Problem 4 continued on next page. . .

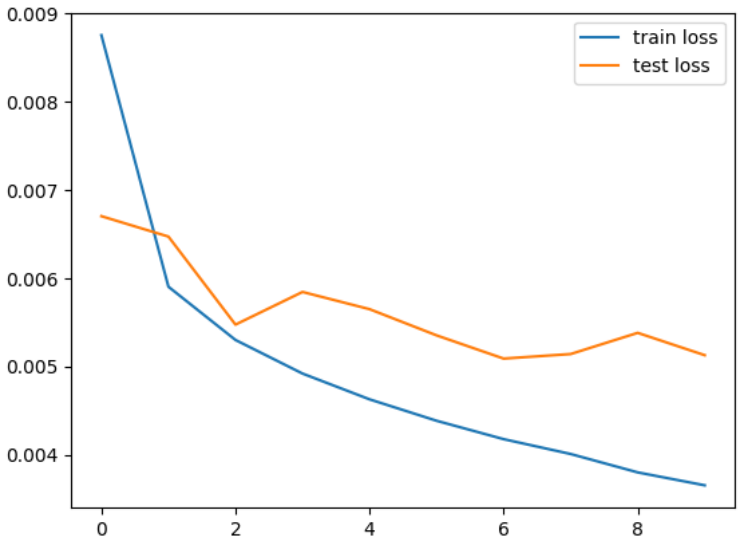
```
        x = F.relu(self.fc3(x))
        x = self.fc4(x)
        # print(x.shape)
        return x

## creating an instance of the model here.
net = Net()

## Define a loss function and optimizer
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(net.parameters(), lr = 0.001)

## creating three empty arrays to store training and test loss and test accuracy
train_loss = []
test_loss = []
test_acc = []

## number of epochs to loop over the dataset multiple times
epochs = 10

## The training loop
for epoch in range(epochs):
    running_train_loss = 0.0
    running_test_loss = 0.0
    total = 0
    ## Iterate through the training dataset
    for i, data in tqdm(enumerate(trainloader, 0), total=len(trainloader)):

        # get the inputs; data is a list of [inputs, labels]
        inputs, labels = data

        # zero the parameter gradients
        optimizer.zero_grad()

        # forward + backward + optimize
        outputs = net(inputs)
        # print(labels.shape, outputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        # store traing loss

        total += labels.size(0)
```

```
        running_train_loss += loss.item()

    train_loss.append(running_train_loss / total)

    ## Here we calculate all the test statistics (test loss, test accuracy)
    ## between training
    correct = 0
    total = 0
    with torch.no_grad():
        for data in testloader:
            images, labels = data
            outputs = net(images)

            ## loss
            test_loss_ = criterion(outputs, labels)
            running_test_loss += test_loss_.item()

            ## accuracy
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

    accuracy = (100 * correct / total)
    test_acc.append(accuracy)
    test_loss.append(running_test_loss / total)
    print("Train Loss:", running_train_loss, "Test Loss:", running_test_loss,
          "Test Acc:", accuracy)

print('Finished Training')

# plot train and test loss curves
import matplotlib.pyplot as plt

plt.plot(train_loss, label='train loss')
plt.plot(test_loss, label='test loss')
plt.legend()
plt.show()
```

```
100%|| 938/938 [00:05<00:00, 156.87it/s]
Train Loss: 525.3377365767956 Test Loss: 67.04034739732742 Test Acc: 84.75
100%|| 938/938 [00:06<00:00, 153.80it/s]
Train Loss: 354.25329430401325 Test Loss: 64.72430749237537 Test Acc: 84.31
100%|| 938/938 [00:06<00:00, 145.24it/s]
Train Loss: 318.0483229458332 Test Loss: 54.746697932481766 Test Acc: 87.29
```

Problem 4 continued on next page. . .

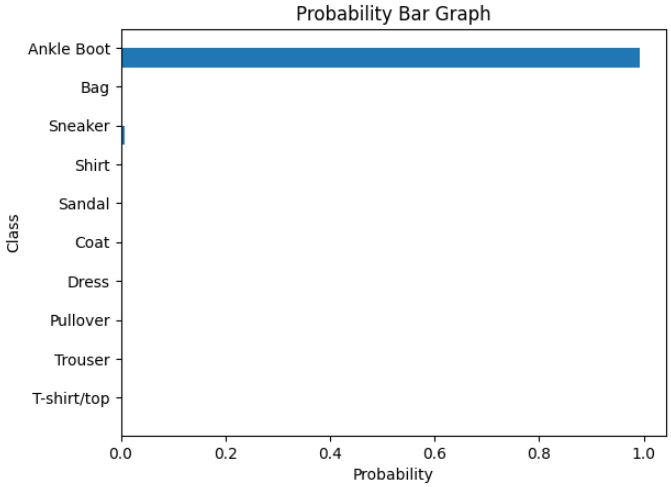
```
100%|| 938/938 [00:06<00:00, 150.85it/s]
Train Loss: 295.2006321325898 Test Loss: 58.45132350921631 Test Acc: 86.3
100%|| 938/938 [00:06<00:00, 144.39it/s]
Train Loss: 277.66891358047724 Test Loss: 56.49798436462879 Test Acc: 86.97
100%|| 938/938 [00:06<00:00, 148.55it/s]
Train Loss: 263.0947785079479 Test Loss: 53.528168603777885 Test Acc: 87.77
100%|| 938/938 [00:06<00:00, 143.27it/s]
Train Loss: 250.59266617149115 Test Loss: 50.89640510082245 Test Acc: 88.37
100%|| 938/938 [00:06<00:00, 136.44it/s]
Train Loss: 240.49185923859477 Test Loss: 51.412230141460896 Test Acc: 88.32
100%|| 938/938 [00:06<00:00, 143.38it/s]
Train Loss: 227.98804431408644 Test Loss: 53.81393413245678 Test Acc: 88.59
100%|| 938/938 [00:06<00:00, 134.79it/s]
Train Loss: 219.19707740843296 Test Loss: 51.28172568976879 Test Acc: 88.7
Finished Training
```

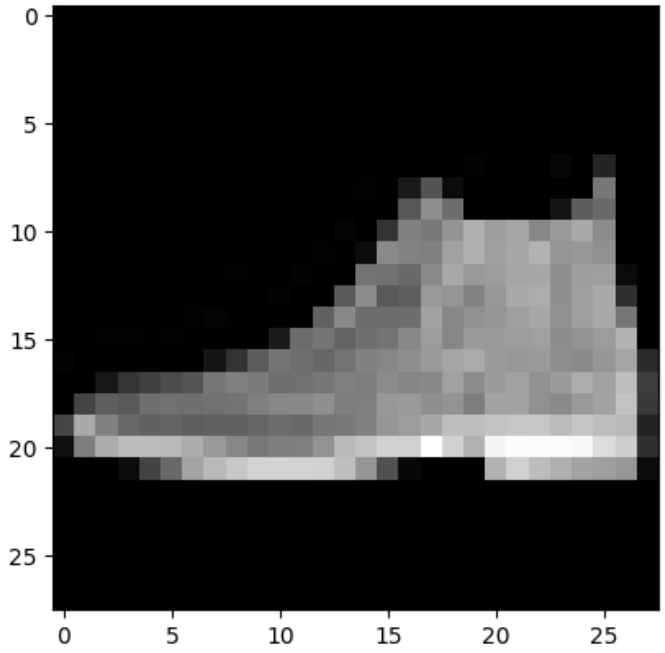


Plotting Probabilities of 3 samples from Test Set

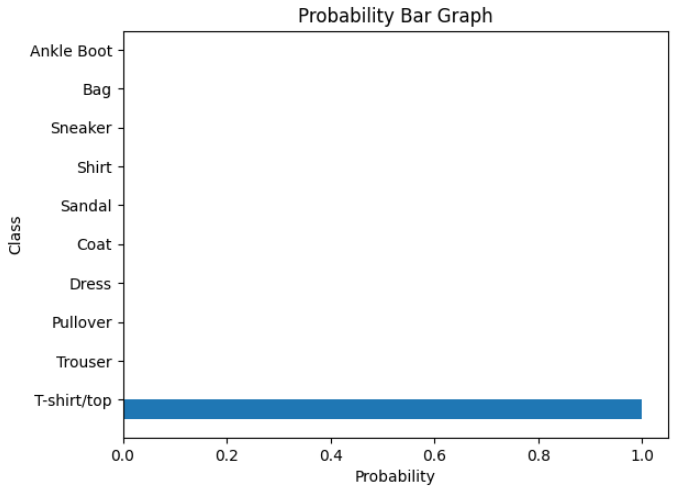
```
import numpy as np

def plot_probability(probabilities, image):
    print(probabilities.shape)
    classes = ['T-shirt/top', 'Trouser', 'Pullover', 'Dress', 'Coat', 'Sandal',
               'Shirt', 'Sneaker', 'Bag', 'Ankle Boot']
    num_classes = 10
    show = tuple(probabilities.detach().numpy())

    bar_width = 0.5
    figure, axis = plt.subplots()
    index = np.arange(num_classes)
    rects = axis.barh(index, show, bar_width, align="center", label='probability')

    axis.set_ylabel('Class')
    axis.set_xlabel('Probability')
    axis.set_yticks(index + bar_width / 2)
    axis.set_yticklabels(tuple(classes))
    axis.set_title('Probability Bar Graph')
    plt.show()

    image = image.view(28, 28)
    plt.imshow(image, cmap='gray')
    plt.show()
```

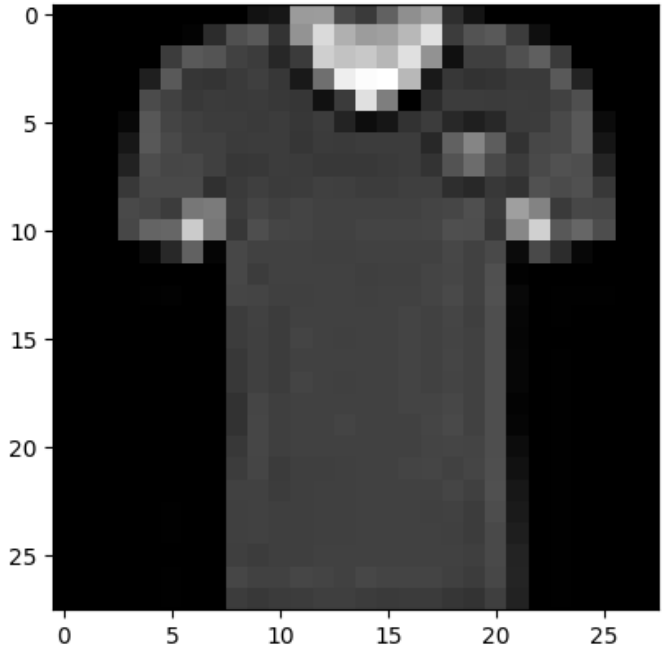
```
image, _ = next(iter(testloader))
out = net(image)
plot_probability(F.softmax(out[0], dim=0), image[0]);
```



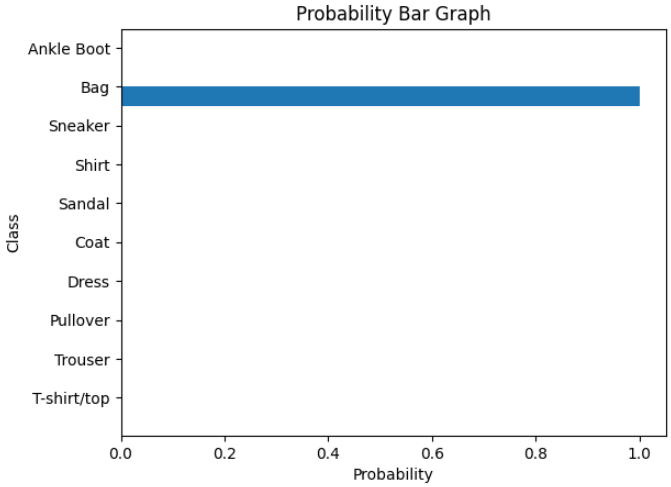


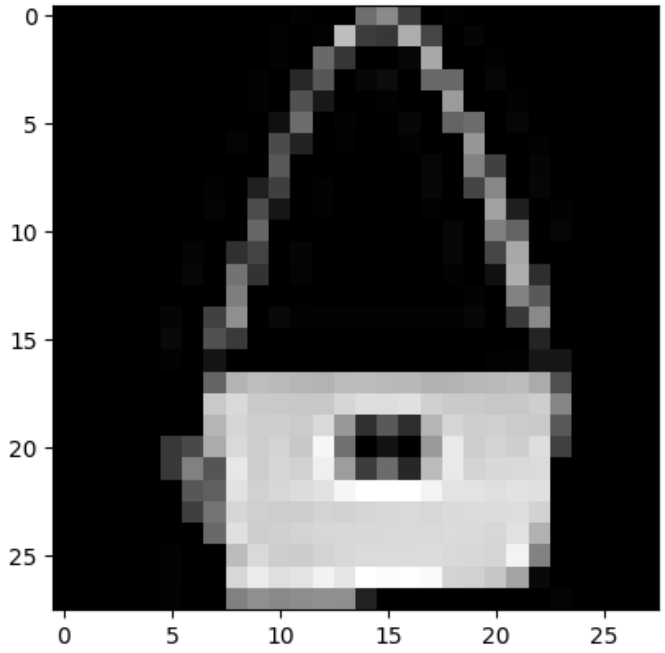
```
image, _ = next(iter(testloader))
out = net(image)
plot_probability(F.softmax(out[59], dim=0), image[59])
```





```
image, _ = next(iter(testloader))
out = net(image)
plot_probability(F.softmax(out[31], dim=0), image[31])
```



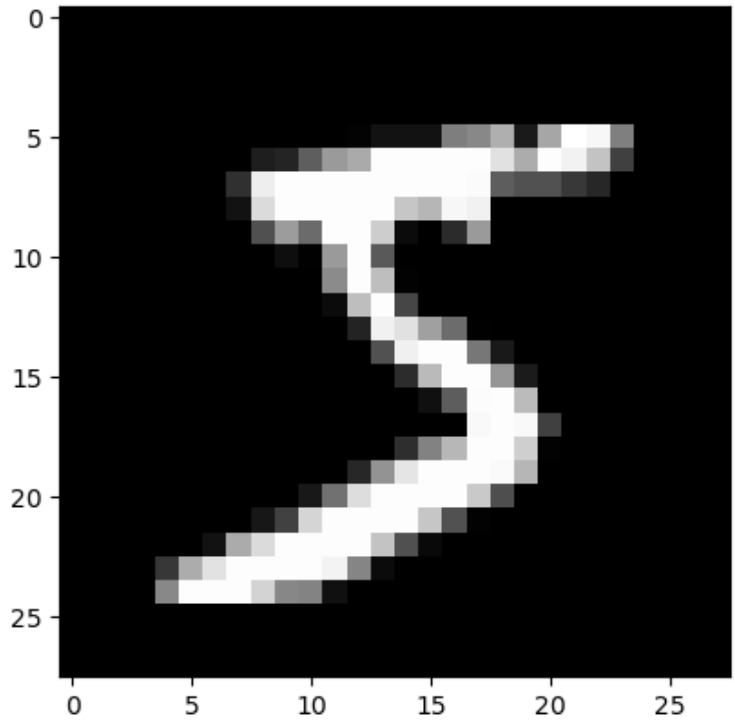


The probabilities of classes corresponding to the image is the highest - almost near 1. The probabilities of rest of the classes are negligible.

Problem 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.

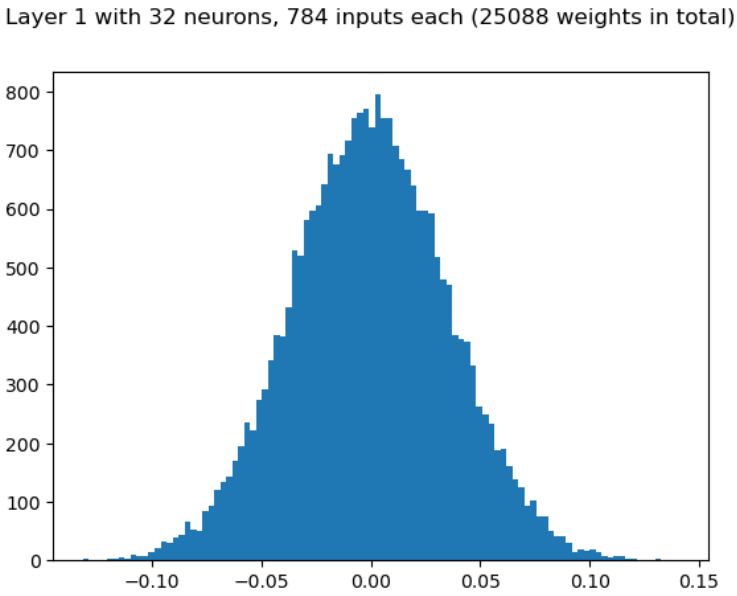
```
import numpy as np

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
```

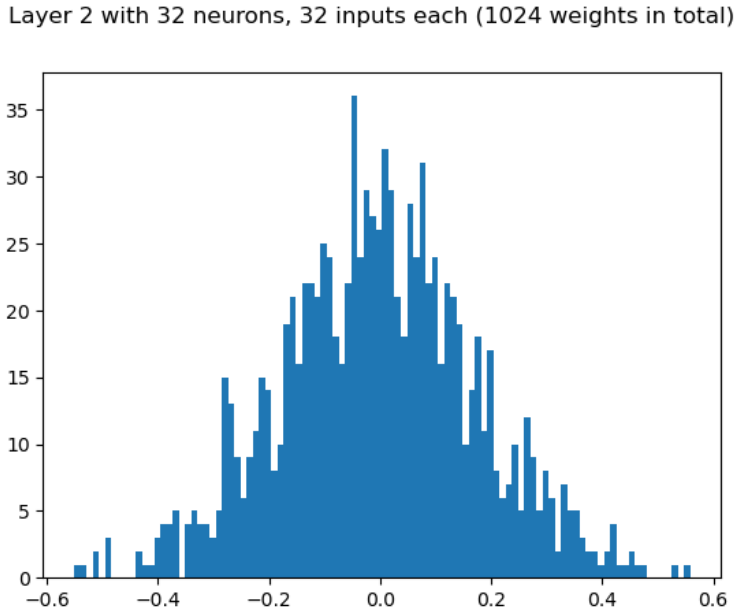
OK, we are now ready to build and train our model. The input is an image of size 28x28, and the output is one of 10 classes. So, first:
Q1. Initialize a 2-hidden layer neural network with 32 neurons in each hidden layer, i.e., your layer sizes should be:
$784 \rightarrow 32 \rightarrow 32 \rightarrow 10$

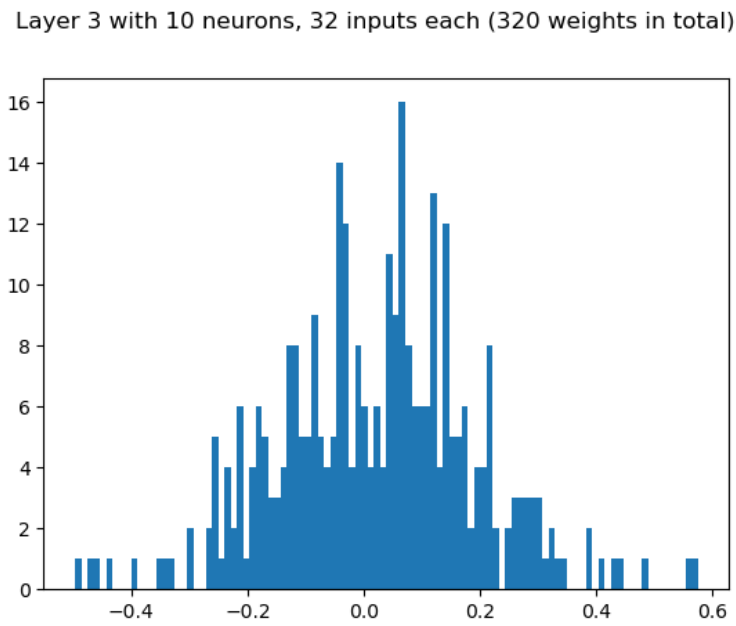
If the layer is $n_{in} \times n_{out}$ your layer weights should be initialized by sampling from a normal distribution with mean zero and variance $1/\max(n_{in}, n_{out})$.

```
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)

# Neural network layer sizes: 784 -> 32 -> 32 -> 10

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

weights = [
  rng.normal(0, 1/math.sqrt(784), (32, 784)),
  rng.normal(0, 1/math.sqrt(32), (32, 32)),
  rng.normal(0, 1/math.sqrt(32), (10, 32))
]

biases = [np.zeros(32), np.zeros(32), np.zeros(10)]

# Plot histogram of layer weights to check probability 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);
```

`Weight distribution per layer:`







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):
  """ Feeds a sample forward through the neural network.
    Parameters:
      sample: 1D numpy array. The input sample (an MNIST digit).
      label: An integer from 0 to 9.

    Returns: The cross entropy loss.
  """

  # Q2. Fill code here.
  # ...
  a = sample.flatten()

  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 encoded guess
  one_hot_guess = np.zeros_like(a)
  one_hot_guess[np.argmax(a)] = 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])

  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
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 = []

  # Forward pass
  for i, w in enumerate(weights):
    z = np.matmul(w, a) + biases[i]
    if (i < len(weights) - 1):
      a = sigmoid(z)
    else:
      a = softmax(z)
    activations.append(a)

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

  # Convert last layer's activations to one hot encoded guess
  one_hot_guess = np.zeros_like(a)
  one_hot_guess[np.argmax(a)] = 1

  # Check whether guess was correct
  correct_guess = (np.sum(one_hot_y * one_hot_guess) == 1)

  weight_gradients = [None] * len(weights)
  bias_gradients = [None] * len(weights)
  activation_gradients = [None] * (len(weights) - 1)
```

```
# 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
  second_to_last_layer = i == len(weights) - 2

  if last_layer:
    # Gather all needed variables, making vectors vertical
    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 all needed 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 second_to_last_layer:
      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 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: 0.49
Accuracy (# of correct guesses): 8561.0 / 10000 ( 85.61 %)

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
Accuracy (# of correct guesses): 8475.0 / 10000 ( 84.75 %)

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: 0.5
Accuracy (# of correct guesses): 8508.0 / 10000 ( 85.08 %)

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
Accuracy (# of correct guesses): 8511.0 / 10000 ( 85.11 %)
```

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