Deep Learning Homework 1

Shubham Singh (sks9437)

February 2024

Question 1

1. **(4 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. (1pt) 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.)

   b. (2pts) 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.

   c. (1pt) 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.

Answer:

1a:

To design a neural network capable of representing the function $f(x) = h$ for $0 < x < \delta$ and 0 otherwise using a neural network.

Let's outline the structure of our neural network. We have a neural network with a scalar input x, two hidden layers, and a final output layer. The hidden layers utilize a unit step activation function given by:

$$y = \sigma(z, 0)$$

where $z =< w, x > +b$. Here, $\sigma(z, 0)$ represents the unit step function.

In the hidden layers, we obtain two intermediate values:

$$z_1 = x_1 \cdot w_1 + b$$

$$z_2 = x_2 \cdot w_2 + b$$

Then, the unit step function is applied to these values:

$$a_1 = \sigma(z_1)$$

$$a_2 = \sigma(z_2)$$

Finally, in the output layer, the network produces its final output:

$$y = a_1 + a_2$$

This structure allows the neural network to effectively represent the desired function within the specified range $0 < x < \delta$ if we set the values of w_1, w_2, and b correctly.

There are several ways to think and design this. One possible way involves the following approach:

$$\sigma(w_1 \cdot x + b) + \sigma(w_2 \cdot x + b) = h \text{ for } x < \delta$$

Here, h can be 2, and another weight can be used to scale it to h.

Also thinking about the design we know that at $x > \delta$, the value becomes 0 again. The factor containing weights will change with x, and the bias will stay constant. We can use this logic to initialize values of weights such that the overall sum $w \cdot x + b$ is positive between $0 < x < \delta$ and negative otherwise.

This gives us an intuition that weights w_1 and w_2 must be negative, and b must be greater than δ.

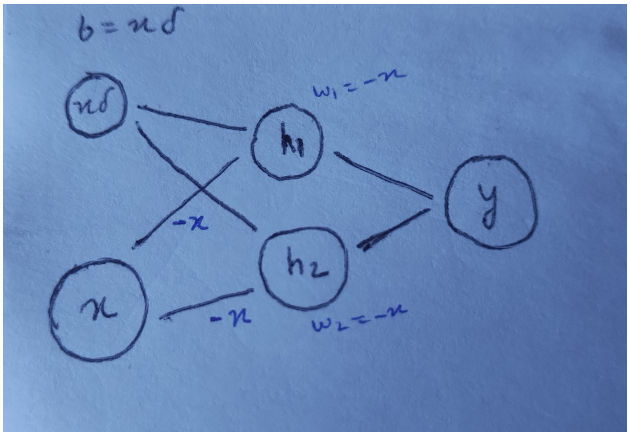
If we set w_1 and w_2 as -1, we have

$$\sigma(-x + \delta) + \sigma(-x + \delta) = 2$$

which stays true when $x < \delta$ and since this value is what we need, it is a good candidate for usage.

But we also need 0 when $x < 0$, but for values of $x < 0$, where $|x| < \delta$, this won't be true.

Then, we come up with an intuition of a better expression of the function, with $w_1 = -x$ and $w_2 = -x$, and bias $= x \cdot \delta$.



This neural network approximates our desired function well enough. It is still delta inclusive.

1b: Using the box function described in part (a), we can divide the input space into segments, enabling us to approximate the smooth function with greater precision by increasing the number of neurons in the hidden layer. Each pair of neurons forms a "box" that represents a small segment of the function. The height of these boxes is adjusted to align with the function's value within that segment. As more neurons are added, the network creates additional boxes, allowing for a finer approximation of the function. By summing the contributions of these boxes computed by subsequent layers, the network can effectively approximate the smooth function across the entire interval.

1c: Indeed the argument discussed in part (b) can be applied to scenarios involving d inputs well. These regions would cover the d input space enabling us to estimate the function within each region by a value, similar to what was explained in part (b).

However implementing this approach in dimensions brings about difficulties. One significant challenge is the increase in the number of neurons (and consequently parameters) needed as the input space dimensionality grows. This can result in inefficiency and memory limitations making it unfeasible to precisely define regions within dimensional spaces.

Moreover delineating regions in a dimensional space that correspond to specific function values becomes increasingly intricate as the dimensionality rises. In dimensions the myriad combinations of ranges along each dimension expand exponentially posing difficulties in depicting the function within each region.

Question 2:

2. **(3 points) Algebraic exercises with gradients**. Suppose that z is a vector with n elements. We would like to compute the gradient of $y = \text{softmax}(z)$. Show that the Jacobian of y with respect to z, J, is given by the expression:

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

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,* $\frac{\partial \log y_i}{\partial z_j}$.

Answer: We are given a vector z with n elements.

$$y = \text{softmax}(z)$$
$$y_i = \frac{e^{z_i}}{\sum_{k=1}^{n} e^{z_k}}$$

Taking logs on both sides:

$$\log y_i = \log \frac{e^{z_i}}{\sum_{k=1}^{n} e^{z_k}}$$

Taking the derivative:

$$\frac{d \log y_i}{dz_i} = \frac{d}{dz_i}\left(\log \frac{e^{z_i}}{\sum_{k=1}^{n} e^{z_k}}\right)$$
$$\frac{d \log y_i}{dz_i} = \frac{d}{dz_i}\left(\log e^{z_i} - \log \sum_{k=1}^{n} e^{z_k}\right)$$

We know, $\log(e^{z_i}) = z_i$. Using this,

$$\frac{d \log y_i}{dz_i} = \frac{d}{dz_i}\left(z_i - \log \sum_{k=1}^{n} e^{z_k}\right) \quad (i)$$

Given:

$$\delta_{ij} = \begin{cases} 1 & \text{if } i = j \\ 0 & \text{otherwise} \end{cases}$$

We need to show:

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

In equation (i), where $\frac{dz_i}{dz_j} = 1$ if $i = j$, it becomes:

$$\frac{d \log y_i}{dz_i} = 1 - \frac{d}{dz_i}\left(-\log \sum_{k=1}^{n} e^{z_k}\right)$$

Now, for $\frac{d}{dz_j} \log\left(\sum_{k=1}^{n} e^{z_k}\right)$, we have:

$$\frac{d}{dz_j} \log\left(\sum_{k=1}^{n} e^{z_k}\right) = \frac{1}{\sum_{k=1}^{n} e^{z_k}}\left(\frac{d}{dz_j}\left(\sum_{k=1}^{n} e^{z_k}\right)\right) = \frac{e^{z_j}}{\sum_{k=1}^{n} e^{z_k}} = y_j$$

Hence,

$$\frac{d \ln y_i}{dz_j} = \delta_{ij} - y_j = \frac{1}{y_i}\frac{dy_i}{dz_j}$$

Therefore,

$$\frac{dy_i}{dz_j} = y_i(\delta_{ij} - y_j)$$

Question 3

3. **(3 points)** *Improving the FashionMNIST classifier*. In the first demo, we trained a simple logistic regression model to classify MNIST digits. Repeat the same experiment, but now use a (dense) neural network with three (3) hidden layers with 256, 128, and 64 neurons respectively, all with ReLU activations. Display train- and test- loss curves, and report test accuracies of your final model. You may have to tweak the total number of training epochs to get reasonable accuracy. Finally, draw any 3 image samples from the test dataset, visualize the predicted class probabilities for each sample, and comment on what you can observe from these plots.

Answer:

```
# We start by importing the libraries we'll use today
import numpy as np
import torch
import torchvision
```

Training Dense models

We will train a dense model classifier with three (3) hidden layers with 256, 128, and 64 neurons respectively, all with ReLU activations on a popular image dataset called *Fashion-MNIST*. Torchvision also has several other image datasets which we can directly load as variables.

```
trainingdata = torchvision.datasets.FashionMNIST('./FashionMNIST/',train=True,download=True,transform=torch
testdata = torchvision.datasets.FashionMNIST('./FashionMNIST/',train=False,download=True,transform=torchvis
```

Let's check that everything has been downloaded.

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

```
60000
10000
```

Let's investigate to see what's inside the dataset.

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

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

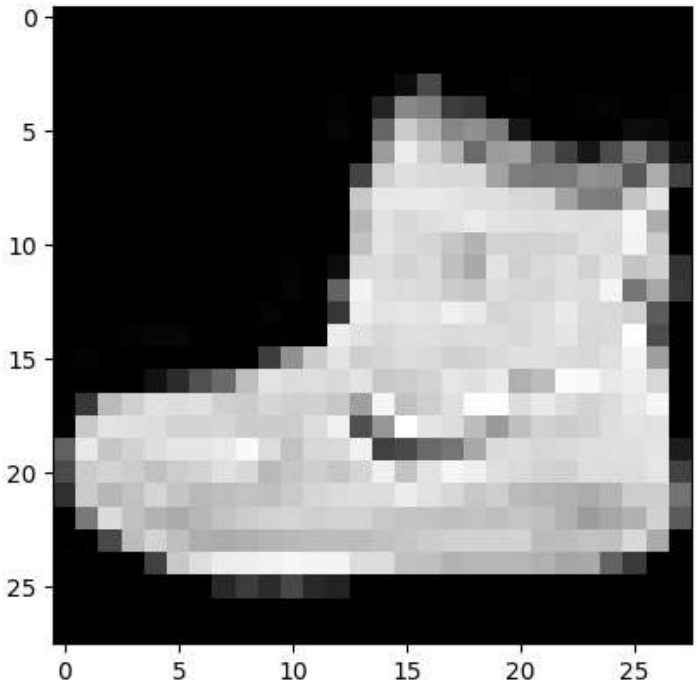
We cannot directly plot the image object given that its first dimension has a size of 1. So we will use the `squeeze` function to get rid of the first dimension.

```
print(image.squeeze().shape)
```

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

```
import matplotlib.pyplot as plt
%matplotlib inline
plt.imshow(image.squeeze(), cmap=plt.cm.gray)
```

```
<matplotlib.image.AxesImage at 0x7eb705d97460>
```



Looks like a shoe? Fashion-MNIST is a bunch of different black and white images of clothing with a corresponding label identifying the category the clothing belongs to. It looks like label 9 corresponds to shoes.

In order to nicely wrap the process of iterating through the dataset, we'll use a dataloader.

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

Let's also check the length of the train and test dataloader

```
print(len(trainDataLoader))
print(len(testDataLoader))
```

```
938
157
```

The length here depends upon the batch size defined above. Multiplying the length of our dataloader by the batch size should give us back the number of samples in each set.

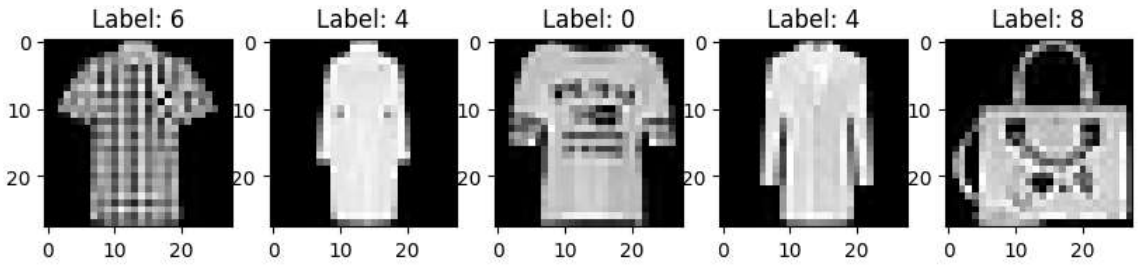
```
print(len(trainDataLoader) * 64) # batch_size from above
print(len(testDataLoader) * 64)
```

```
60032
10048
```

Now let's use it to look at a few images.

```

```

```
images, labels = next(iter(trainDataLoader))

plt.figure(figsize=(10,4))
for index in np.arange(0,5):
  plt.subplot(1,5,index+1)
  plt.title(f'Label: {labels[index].item()}')
  plt.imshow(images[index].squeeze(),cmap=plt.cm.gray)
```



Now let's set up our model.

```
from torch import nn
class DenseNetwork(nn.Module):
    def __init__(self):
        super().__init__()
        self.flatten = nn.Flatten()
        self.linear_relu_stack = nn.Sequential(
            nn.Linear(784, 784),
            nn.Linear(784, 256), # 256
            nn.ReLU(),
            nn.Linear(256, 128), # 128
            nn.ReLU(),
            nn.Linear(128, 64), # 64
            nn.ReLU(),
            nn.Linear(64, 10)
        )

    def forward(self, x):
      x = x.view(-1, 28*28) # change so 784 vector instead of 28x28 matrix
      return self.linear_relu_stack(x)

model = DenseNetwork().cuda() # Step 1: architecture
loss = torch.nn.CrossEntropyLoss() # Step 2: loss
optimizer = torch.optim.SGD(model.parameters(), lr=0.01) # Step 3: training method
```

```
'''#!pip install torchviz
from torchviz import make_dot
dot = make_dot(model(image.cuda()), params=dict(model.named_parameters()))
dot.render('model_architecture')'''
```

```
'#!pip install torchviz\nfrom torchviz import make_dot\ndot = make_dot(model(image.cud
a()), params=dict(model.named_parameters()))\ndot.render('model_architecture')'
```

Now let's train our model!

```
train_loss_history = []
```

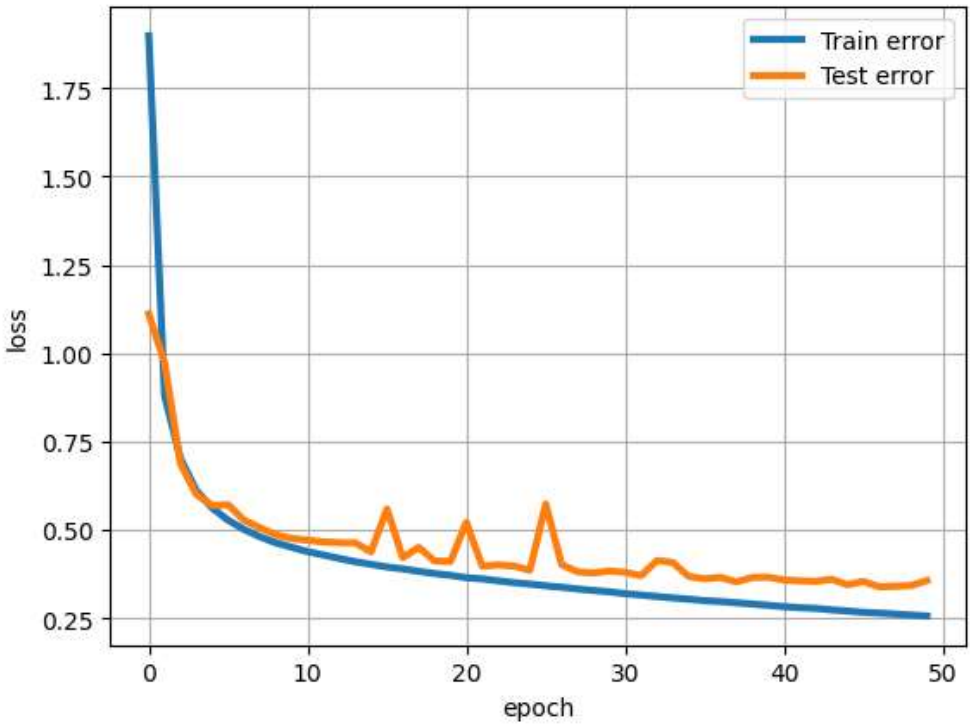
```
test_loss_history = []

for epoch in range(50):
  train_loss = 0.0
  test_loss = 0.0

  model.train()
  for i, data in enumerate(trainDataLoader):
    images, labels = data
    images = images.cuda()
    labels = labels.cuda()
    optimizer.zero_grad() # zero out any gradient values from the previous iteration
    predicted_output = model(images) # forward propagation
    fit = loss(predicted_output, labels)  # calculate our measure of goodness
    fit.backward() # backpropagation
    optimizer.step() # update the weights of our trainable parameters
    train_loss += fit.item()

  model.eval()
  for i, data in enumerate(testDataLoader):
    with torch.no_grad():
      images, labels = data
      images = images.cuda()
      labels = labels.cuda()
      predicted_output = model(images)
      fit = loss(predicted_output, labels)
      test_loss += fit.item()
  train_loss = train_loss / len(trainDataLoader)
  test_loss = test_loss / len(testDataLoader)
  train_loss_history += [train_loss]
  test_loss_history += [test_loss]
  print(f'Epoch {epoch}, Train loss {train_loss}, Test loss {test_loss}')
```

```
Epoch 0, Train loss 1.897869714033375, Test loss 1.111085258471738
Epoch 1, Train loss 0.8789164601866879, Test loss 0.9724211749757171
Epoch 2, Train loss 0.7014260781345083, Test loss 0.6870506762698957
Epoch 3, Train loss 0.6111731012302167, Test loss 0.6000811947379142
Epoch 4, Train loss 0.5618695622123381, Test loss 0.5691114159146692
Epoch 5, Train loss 0.5272792202196142, Test loss 0.5711028536034238
Epoch 6, Train loss 0.5011559316217263, Test loss 0.5272795275138442
Epoch 7, Train loss 0.4805594324938524, Test loss 0.5063282069127271
Epoch 8, Train loss 0.4633961077024942, Test loss 0.48604604450001077
Epoch 9, Train loss 0.4509246730759962, Test loss 0.47567720198707214
Epoch 10, Train loss 0.4383877603642976, Test loss 0.4707773722660769
Epoch 11, Train loss 0.4289978629649321, Test loss 0.46597911882552373
Epoch 12, Train loss 0.41905407999942046, Test loss 0.4638971975844377
Epoch 13, Train loss 0.41001356817257684, Test loss 0.4631393323088907
Epoch 14, Train loss 0.4024728394107524, Test loss 0.43800614878630184
Epoch 15, Train loss 0.39513283207027644, Test loss 0.5591531578142932
Epoch 16, Train loss 0.389657431065655, Test loss 0.42150106570523255
Epoch 17, Train loss 0.3827182360009344, Test loss 0.45077980532767664
Epoch 18, Train loss 0.3764442508535853, Test loss 0.41198014102543995
Epoch 19, Train loss 0.3715025987833548, Test loss 0.41046437231978034
Epoch 20, Train loss 0.364586381134448, Test loss 0.5212580487606632
Epoch 21, Train loss 0.3615021898166966, Test loss 0.397480691883974
Epoch 22, Train loss 0.3559939106549027, Test loss 0.4008031365028612
Epoch 23, Train loss 0.3506951570780928, Test loss 0.39750587247359526
Epoch 24, Train loss 0.3466787617296171, Test loss 0.3860179239967067
Epoch 25, Train loss 0.34171273596664226, Test loss 0.5732086609313443
Epoch 26, Train loss 0.33777543379744485, Test loss 0.40170566424442705
Epoch 27, Train loss 0.33282993744208866, Test loss 0.381438892738075
Epoch 28, Train loss 0.32854064603222966, Test loss 0.3780695710592209
```

```
Epoch 29, Train loss 0.3248245536343757, Test loss 0.3835146638807977
Epoch 30, Train loss 0.319600731182073, Test loss 0.38000406552651883
Epoch 31, Train loss 0.3156528295611522, Test loss 0.370384489275088
Epoch 32, Train loss 0.31142987696918595, Test loss 0.4131444319608105
Epoch 33, Train loss 0.3074787572375747, Test loss 0.4077653839330005
Epoch 34, Train loss 0.30414748461897184, Test loss 0.3681124541789863
Epoch 35, Train loss 0.2997405756908312, Test loss 0.36171506411710364
Epoch 36, Train loss 0.29732820098556434, Test loss 0.36544593836471534
Epoch 37, Train loss 0.2937592923371141, Test loss 0.352972103437041
Epoch 38, Train loss 0.29032680827544444, Test loss 0.36526796392574434
Epoch 39, Train loss 0.28660191268300705, Test loss 0.3661798995201755
Epoch 40, Train loss 0.28284773702369825, Test loss 0.3580830351562257
Epoch 41, Train loss 0.2798769452027293, Test loss 0.3559854039151198
Epoch 42, Train loss 0.2778182718386528, Test loss 0.35438834643288025
Epoch 43, Train loss 0.27368202514803486, Test loss 0.36033849161901294
Epoch 44, Train loss 0.27059085702877056, Test loss 0.34467601225634287
Epoch 45, Train loss 0.26675511782230343, Test loss 0.3541745537785208
Epoch 46, Train loss 0.2646856090343837, Test loss 0.3388934019644549
Epoch 47, Train loss 0.26181354392756784, Test loss 0.3405738333418111
Epoch 48, Train loss 0.25906748081575326, Test loss 0.34225938986441135
Epoch 49, Train loss 0.2564311968937103, Test loss 0.35657339035325747
```

Let's plot our loss by training epoch to see how we did.

```
plt.plot(range(50),train_loss_history,'-',linewidth=3,label='Train error')
plt.plot(range(50),test_loss_history,'-',linewidth=3,label='Test error')
plt.xlabel('epoch')
plt.ylabel('loss')
plt.grid(True)
plt.legend()
plt.show()
```



Why is test loss larger than training loss?

We definitely see some improvement. Let's look at the images, the predictions our model makes and the true label.

Now for the labels and predicted labels.

```
predicted_outputs = model(images)
predicted_classes = torch.max(predicted_outputs, 1)[1]
print('Predicted:', predicted_classes)
fit = loss(predicted_output, labels)
print('True labels:', labels)
print(fit.item())
```

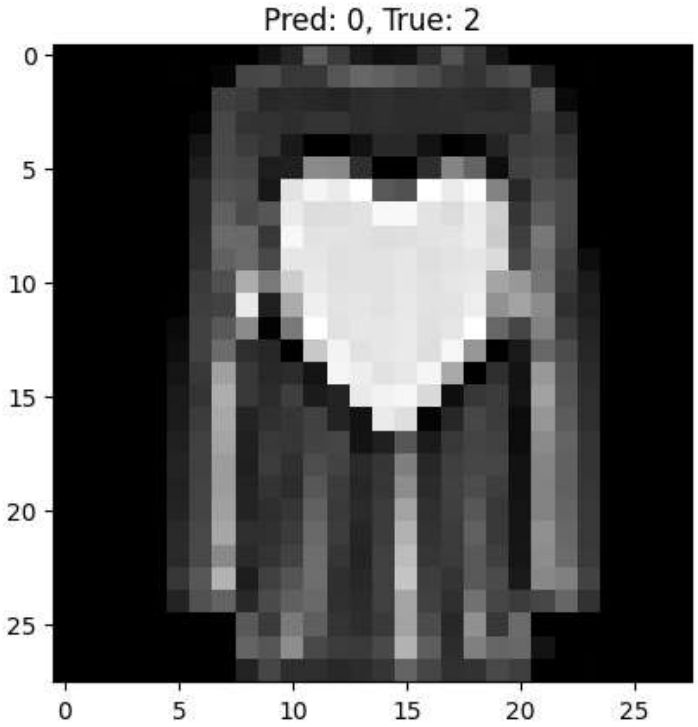
```
Predicted: tensor([3, 0, 7, 5, 8, 4, 5, 6, 8, 9, 1, 9, 1, 8, 1, 5], device='cuda:0')
True labels: tensor([3, 2, 7, 5, 8, 4, 5, 6, 8, 9, 1, 9, 1, 8, 1, 5], device='cuda:0')
0.12692052125930786
```

```
correct = 0
total = 0
with torch.no_grad():
    for data in testDataLoader:
        images, labels = data
        # Move data to the same device as the model
        images = images.cuda()
        labels = labels.cuda()
        # calculate outputs by running images through the network
        outputs = model(images)
        # the class with the highest energy is what we choose as prediction
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

print(f'Accuracy of the network on the test images: {100 * correct // total} %')
```

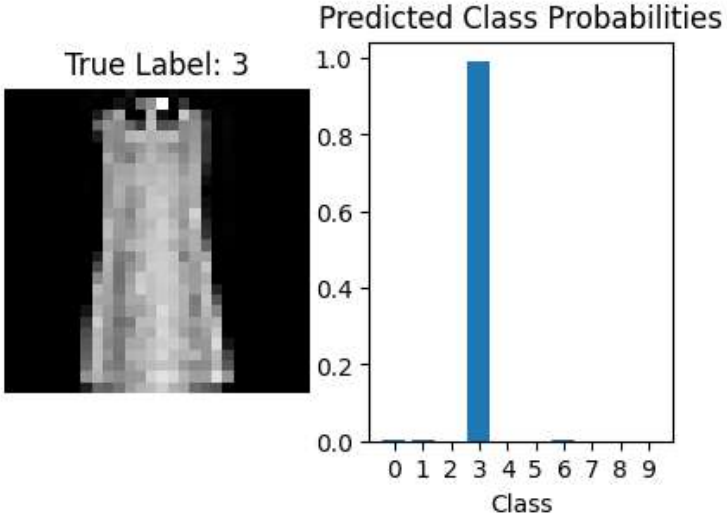
```
Accuracy of the network on the test images: 86 %
```

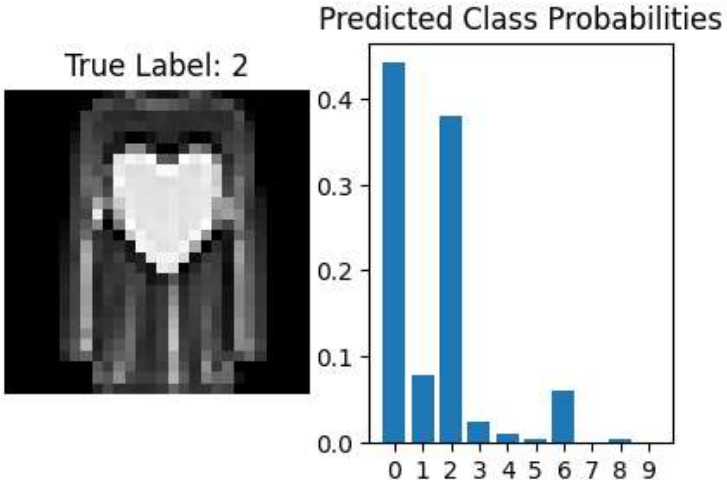
```
plt.imshow(images[1].squeeze().cpu(), cmap=plt.cm.gray)
plt.title(f'Pred: {predicted_classes[1].item()}, True: {labels[1].item()}')
```

```
Text(0.5, 1.0, 'Pred: 0, True: 2')
```




```
import matplotlib.pyplot as plt
import numpy as np

def plot_class_probabilities(image, predicted_probs, true_label):
    plt.figure(figsize=(5, 3))
    plt.subplot(1, 2, 1)
    plt.imshow(image.squeeze().cpu(), cmap=plt.cm.gray)
    plt.title(f'True Label: {true_label}')
    plt.axis('off')

    plt.subplot(1, 2, 2)
    plt.bar(np.arange(len(predicted_probs)), predicted_probs)
    plt.xlabel('Class')
    plt.title('Predicted Class Probabilities')
    plt.xticks(np.arange(len(predicted_probs)))
    plt.show()

# Visualize predictions for three samples
for i in range(3):
    plot_class_probabilities(images[i], torch.softmax(outputs[i], dim=0).cpu().numpy(), labels[i].item())
```


True Label: 3
Predicted Class Probabilities
1.0
0.8
0.6
0.4
0.2
0.0
0 1 2 3 4 5 6 7 8 9
Class


True Label: 2
Predicted Class Probabilities
0.4
0.3
0.2
0.1
0.0
0 1 2 3 4 5 6 7 8 9

Question 4

4. **(5 points)** *Implementing back-propagation in Python from scratch*. Open the (incomplete) Jupyter notebook provided as an attachment to this homework in Google Colab (or other Python IDE of your choice) and complete the missing items. 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.

Answer:

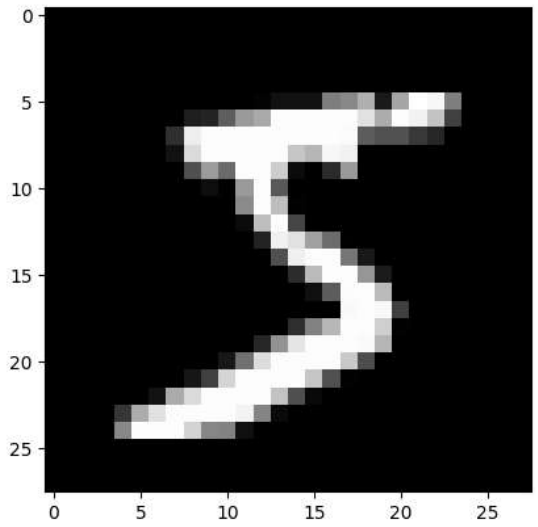
In this problem we will train a neural network from scratch using numpy. In practice, you will never need to do this (you'd just use TensorFlow or PyTorch). But hopefully this will give us a sense of what's happening under the hood.

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');
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/mnist.npz
11490434/11490434 [==============================] - 0s 0us/step
```



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 numpy as np

# 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.
# ...

input_size = 784  # Input layer size (28x28 images flattened)
hidden_size1 = 32  # First hidden layer size
hidden_size2 = 32  # Second hidden layer size
output_size = 10  # Output layer size (10 classes for digits 0-9)

# Function to initialize weights between two layers
def initialize_weights(n_in, n_out):
    # Initialize weights from a normal distribution with mean 0 and std 1/sqrt(max(n_in, n_out))
    stddev = 1 / np.sqrt(max(n_in, n_out))
    return rng.normal(loc=0, scale=stddev, size=(n_in, n_out))

# Initialize weights for each layer
weights = [initialize_weights(input_size, hidden_size1),initialize_weights(hidden_size1, hidden_size2),initialize_weights(hidden_size2, output_siz
biases = [np.zeros(hidden_size1),np.zeros(hidden_size2),np.zeros(output_size)]
```

```
print(len(weights[0]), len(weights[1]),len(weights[2]),len(weights[2]))
```

```
784 32 32 32
```

```
weights[2][1]
```

```
array([-0.03150972,  0.1428765 ,  0.14386447,  0.06791716, -0.46984052,
        0.08087086,  0.11044014,  0.11470577,  0.02369094, -0.03322626])
```

```
print(len(biases[0]), len(biases[1]),len(biases[2]))
```

```
     32 32 10
```

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
    """
    # Forward pass
    sample = sample.flatten()

    for i in range(len(weights)):
        #print(weights[i].shape, biases[i].shape)
        output = np.dot(sample, weights[i]) + biases[i]
        sample = sigmoid(output)

    yHat = softmax(sample)
    one_hot_y = integer_to_one_hot(y,10)
    one_hot_guess = integer_to_one_hot(np.argmax(yHat),10)
    loss = cross_entropy_loss(one_hot_y, yHat)
    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]):
        loss, one_hot_guess = feed_forward_sample(x[i], y[i])
        if np.isnan(loss):
          loss = 0
        losses[i] = loss
        one_hot_guesses[i] = one_hot_guess
        #print(loss)
  y_one_hot = np.zeros((y.size, 10))
  y_one_hot[np.arange(y.size), y] = 1
  #print(y_one_hot)
  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()
#feed_forward_sample(x_train[2], y_train[2])
```

```
Feeding forward all test data...

Average loss: 2.31
Accuracy (# of correct guesses): 880.0 / 10000 ( 8.80 %)
```

```
weights[2][1]
```

```
array([-0.03150972,  0.1428765 ,  0.14386447,  0.06791716, -0.46984052,
        0.08087086,  0.11044014,  0.11470577,  0.02369094, -0.03322626])
```

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 = []
    y_one_hot = integer_to_one_hot(y, 10)  # Assuming 10 classes
    # Forward pass

    # Q3. This should be the same as what you did in feed_forward_sample above.
    # ...

    a_n = [None]*3
    z_n = [None]*3
    for i in range(len(weights)):
      if i == 0:
```

```
      z = np.dot(a, weights[i]) + biases[i]
    else:
      z = np.dot(a_n[i-1], weights[i]) + biases[i]
    z_n[i] = z

    if i != len(weights)-1:
      a_n[i] = sigmoid(z)
    else:
      a_n[i] = softmax(z)

    activations.append(a_n[i])
  # Compute the loss (for monitoring/tracking purposes, not used in gradient computation here)
  loss = cross_entropy_loss(y_one_hot, a_n[2])


  # 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.
  # ...

  # Compute gradients for each layer starting from the last layer
  deltas = [None] * 3
  deltas[2] = a_n[2] - y_one_hot

  weight_gradients = [None] * 3
  bias_gradients = [None] * 3
  for i in reversed(range(len(weights)-1)):
    deltas[i] = np.dot(deltas[i+1], weights[i+1].T)*dsigmoid(z_n[i])
  # Compute weight and bias gradients

  for i in reversed(range(len(weights))):
    if i != 0:
      weight_gradients[i] = np.dot(activations[i-1].reshape(-1, 1), deltas[i].reshape(1, -1))
    else:
      weight_gradients[i] = np.dot(a.reshape(-1, 1), deltas[i].reshape(1, -1))
    bias_gradients[i] = deltas[i]

  # Update weights & biases based on your calculated gradient
  for i in range(len(weights)):
      weights[i] -= learning_rate * weight_gradients[i]
      biases[i] -= learning_rate * bias_gradients[i]
```

```
train_one_sample(x_train[0], y_train[0], learning_rate=0.1)
#train_one_sample(x_train[1], y_train[1], learning_rate=0.1)
```

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...")

    for sample, label in zip(x_train, y_train):
        train_one_sample(sample, label, 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...

 Average loss: 2.31
 Accuracy (# of correct guesses): 892.0 / 10000 ( 8.92 %)

 Training for one epoch over the training dataset...
 Finished training.

 Feeding forward all test data...

 Average loss: 1.97
 Accuracy (# of correct guesses): 6500.0 / 10000 ( 65.00 %)

 Training for one epoch over the training dataset...
 Finished training.

 Feeding forward all test data...

 Average loss: 1.97
 Accuracy (# of correct guesses): 6538.0 / 10000 ( 65.38 %)

 Training for one epoch over the training dataset...
```

```
Finished training.

Feeding forward all test data...

Average loss: 1.96
Accuracy (# of correct guesses): 7007.0 / 10000 ( 70.07 %)
```

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.