Karan Vora (kv2154)
ECE-GY 7143 Introduction to Deep Learning, Assignment 1

Problem 1):

Solution a):
The vector expression for l_1 – norm is

$$L(w)=\|Xw-Y\|_1$$

where, X is a $n\times d$ matrix, w is $1\times d$ vector and Y is a $n\times 1$ vector.

Solution b):
Unlike, l_2 – norm, There's no closed form expression for l_1 – norm as it is non differentiable. We cannot solve for the gradient being zero.

Solution c):
Referred the provided notes for lecture 1, topic Warmup: Linear models. In three step recipe, in the 2nd step where we measure the quality of model, losses are an important metric. The lower the loss, the lower the overall penalty for an incorrect prediction.

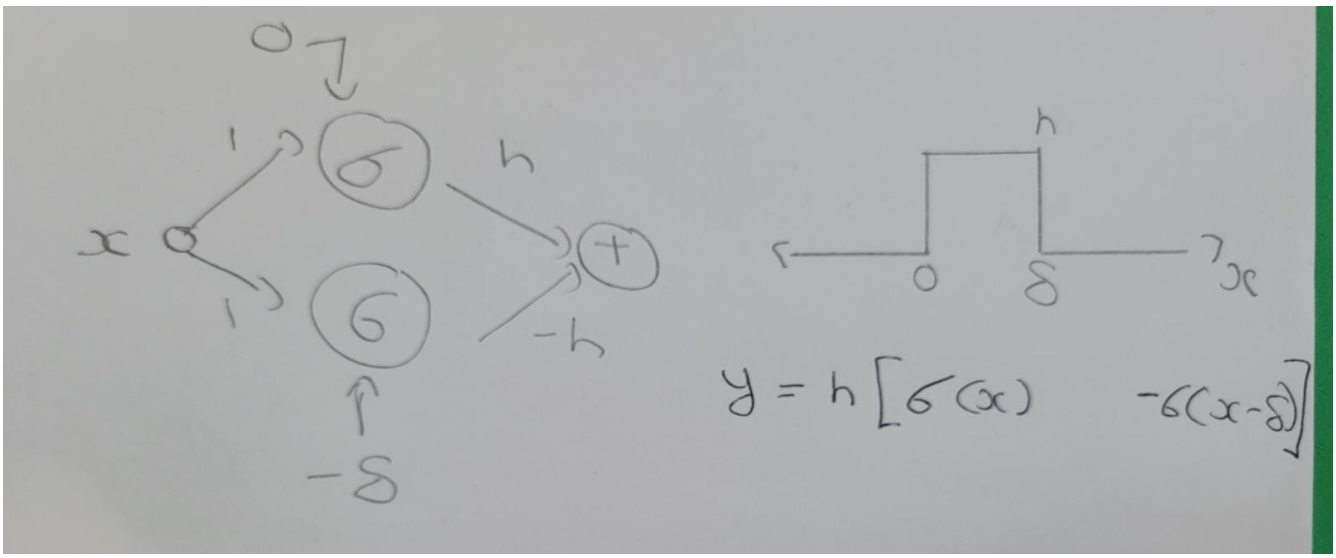
==========

Problem 2):

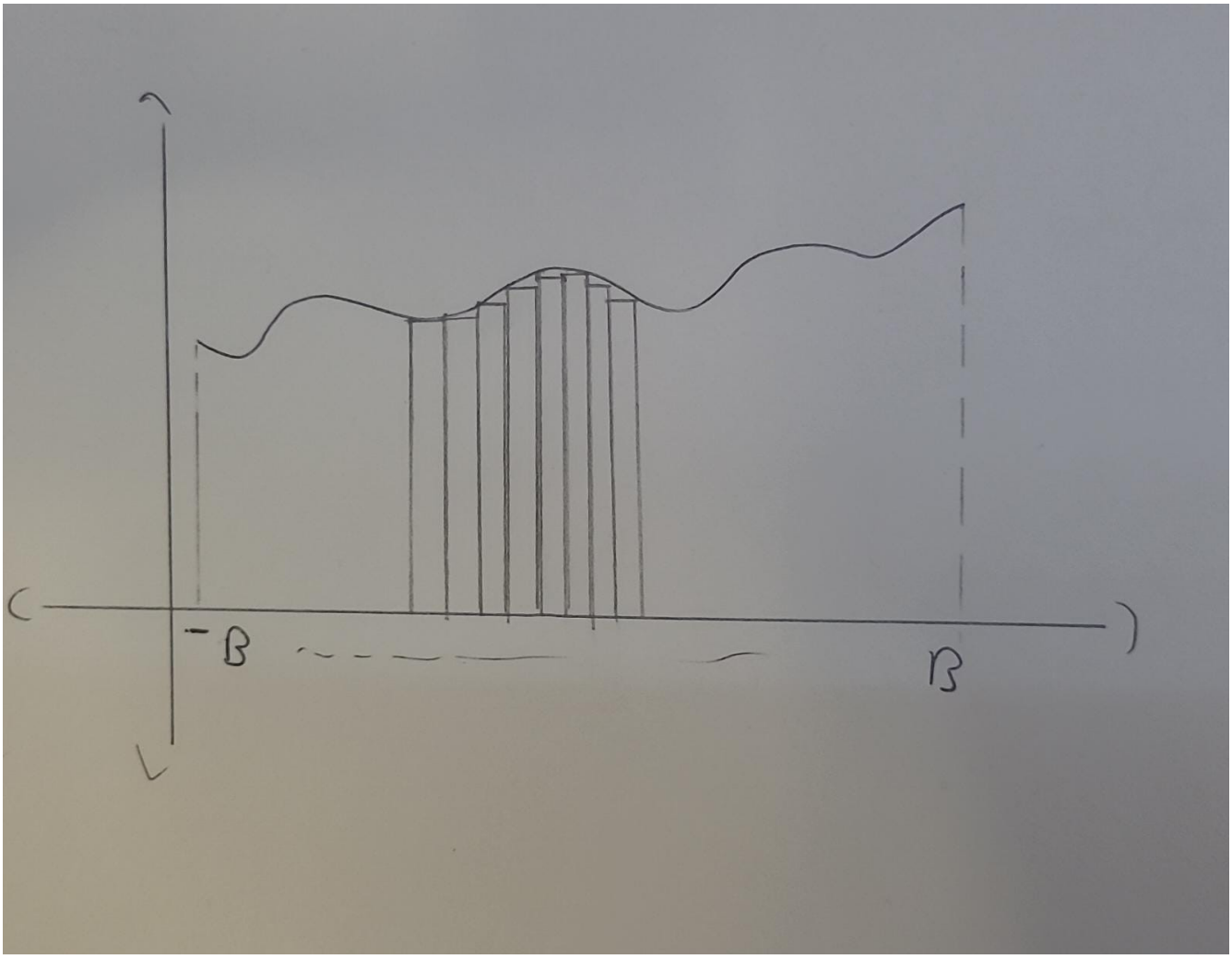
Solution a):
A box function can be realized by using the difference between two step function:

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

This produces the following graph.



Solution b):
The approximate plot of smoothening a curve is given below. We can separate any arbitrary function distribution into an approximate superposition of box function. Each chunk can be plotted using 2 hidden neurons. To get the plot, we change the bias of first layer and value of second layer.



Solution c):
We can apply the same principle to a different dataset types aswell. As the dimensionality of the data increases so is the dimensions of the plotted boxes, thus we require more boxes to plot. Consider a domain,

$D=[-B,B]$

the approximate relation between dimensionality *d* and number of boxes required is given by

$N=(\frac{B}{\delta})^{d}$

With increase in *d* the number of trainable parameters also explodes requiring a more complex network. The network architecture will always be a combination of depth and width to sufficiently

approximate the said function.

Solution d):
From the above mentioned discussion, we can conclude that any sufficiently wide and deep neural network can approximate any arbitrary function with a great degree of accuracy considering the data provided as input is sufficiently good enough in terms of quality and the other parameters or an NN architecture are also optimal.

==========

Problem 3):

The softmax function is given by,

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

Taking log,

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

Now taking partial derivative,

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

When j = i,

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

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

$$= 1 - y_i$$

and when j ≠ i,

$$\frac{\partial y_i}{\partial z_j} = -\frac{e^{z_j}}{\sum_{k=1}^{n} e^{z_k}} = -y_j$$

Therefore, putting it together using Dirac Delta as indicator

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

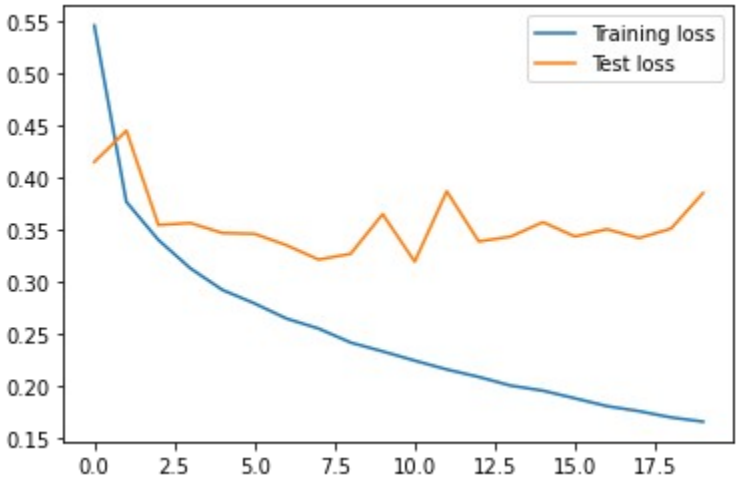
==========

Problem 4):

Disclosure:
I've discussed this particular problem with Rithviik Srinivasan (rs8385) and Charmee Mehta (cm6389). I've also used online resources like StackOverflow, GitHub repositories, Kaggle, ChatGPT and official documentations on pytorch, numpy, matplotlib and pandas to code, debug and optimise.

```
#Import all required libraries
import torch
import torchvision
import torch.nn as nn
import torch.optim as optim
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd

# Set device
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

# Load FashionMNIST dataset
TrainDataset = torchvision.datasets.FashionMNIST(root='./FashionMNIST/', train=True,
transform=torchvision.transforms.ToTensor(), download=True)
TestDataset = torchvision.datasets.FashionMNIST(root='./FashionMNIST/', train=False,
transform=torchvision.transforms.ToTensor(), download=True)

# Define dataloaders
TrainDataLoader = torch.utils.data.DataLoader(TrainDataset, batch_size=64, shuffle=True)
TestDataLoader = torch.utils.data.DataLoader(TestDataset, batch_size=64, shuffle=False)

# Define model
class NeuralNetwork(nn.Module):
def __init__(self):
super(NeuralNetwork, 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)
self.relu = nn.ReLU()

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

```
x = self.relu(self.fc2(x))
x = self.relu(self.fc3(x))
x = self.fc4(x)
return x

model = NeuralNetwork().to(device)

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

# Train model
train_losses = []
test_losses = []
num_epochs = 20

for epoch in range(num_epochs):
train_loss = 0.0
test_loss = 0.0
for images, labels in TrainDataLoader:
images, labels = images.to(device), labels.to(device)
optimizer.zero_grad()
output = model(images)
loss = criterion(output, labels)
loss.backward()
optimizer.step()
train_loss += loss.item() * images.size(0)
train_loss = train_loss / len(TrainDataLoader.dataset)
train_losses.append(train_loss)

with torch.no_grad():
for images, labels in TestDataLoader:
images, labels = images.to(device), labels.to(device)
output = model(images)
loss = criterion(output, labels)
test_loss += loss.item() * images.size(0)
test_loss = test_loss / len(TestDataLoader.dataset)
test_losses.append(test_loss)
print('Epoch: {} \tTraining Loss: {:.6f} \tTest Loss: {:.6f}'.format(epoch+1, train_loss, test_loss))

# Plot loss curves
plt.plot(train_losses, label='Training loss')
plt.plot(test_losses, label='Test loss')
plt.legend()
plt.show()
```

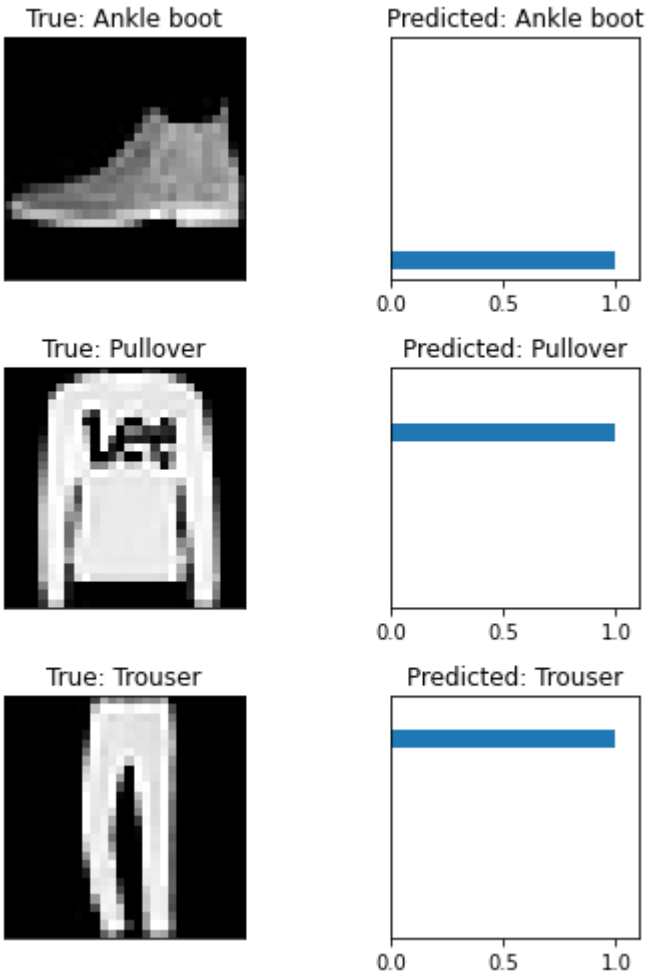



```
# Evaluate model
correct = 0
total = 0
with torch.no_grad():
model.eval()
for images, labels in TestDataLoader:
images, labels = images.to(device), labels.to(device)
output = model(images)
_, predicted = torch.max(output.data, 1)
total += labels.size(0)
correct += (predicted == labels).sum().item()
print('Test Accuracy: {:.2f}%'.format(correct / total * 100))
model.train()
```

Accuracy is 89.28%

```
# Define class names
class_names = ['T-shirt/top', 'Trouser', 'Pullover', 'Dress', 'Coat', 'Sandal', 'Shirt', 'Sneaker', 'Bag', 'Ankle boot']

# Choose 3 random images from the test dataset
num_images = 3
images, labels = next(iter(TestDataLoader))
images, labels = images[:num_images], labels[:num_images]
images, labels = images.to(device), labels.to(device)

# Make predictions and get predicted class probabilities
model.eval()
```

```
outputs = model(images)
probs = torch.softmax(outputs, dim=1)
predicted_classes = torch.argmax(probs, dim=1)

# Visualize images and predicted class probabilities
with torch.no_grad():
fig, axs = plt.subplots(nrows=num_images, ncols=2, figsize=(6, 7))
for i in range(num_images):
img = np.transpose(images[i].cpu().numpy(), (1, 2, 0)).squeeze()
axs[i, 0].imshow(img, cmap='gray')
axs[i, 0].set_xticks([])
axs[i, 0].set_yticks([])
axs[i, 0].set_title('True: {}'.format(class_names[labels[i]]))
axs[i, 1].barh(np.arange(len(class_names)), probs[i].cpu().numpy())
axs[i, 1].set_aspect(0.1)
axs[i, 1].set_yticks([])
axs[i, 1].set_xlim([0, 1.1])
axs[i, 1].invert_yaxis()
axs[i, 1].set_title('Predicted: {}'.format(class_names[predicted_classes[i]]))
plt.tight_layout()
plt.show()
```



==========

Problem 5):

I've discussed this particular problem with Ratik Vig (rv2292) and Charmee Mehta (cm6389).
I've also used online resources like StackOverflow, GitHub repositories, Kaggle, ChatGPT, Youtube lecture on backpropagation by Prof. Chinmay Hegde and official documentations on numpy, matplotlib and pandas to code, debug and optimise.