Module 2: Logistic Regression and Neural Network

To write legible answers you will need to be familiar with both Markdown and Latex

Before you turn this problem in, make sure everything runs as expected. First, restart the kernel (in the menubar, select Kernel→→Restart) and then run all cells (in the menubar, select Cell→→Run All).

Make sure you fill in any place that says "YOUR CODE HERE" or "YOUR ANSWER HERE", as well as your name below:

In [ ]:
```
NAME = ""
STUDENT_ID = ""
```

In [ ]:
```
# TensorFlow and tf.keras
import tensorflow as tf
from tensorflow import keras

# Helper libraries
import numpy as np
import matplotlib.pyplot as plt

import numpy as np
import seaborn as sns
import pandas as pd
```

Data Exploration and Preprocessing

Load the Fashion-MNIST dataset

Keras has lots of datsets that you can just load right into python numpy arrays, see: https://keras.io/datasets/

We will be using the Fashion-MNIST dataset, which is a cool little dataset with gray scale 28×28 images of articles of clothing. Keep in mind that they will be downloaded from the internet, so it may take a while.

In [ ]:
```
fashion_mnist = keras.datasets.fashion_mnist
# splitting training and test data and corresponding labels
(train_images, train_labels), (test_images, test_labels) = fashion_mnist.load_data(
```

In [ ]:
```
class_names = ['T-shirt/top', 'Trouser', 'Pullover', 'Dress', 'Coat',
               'Sandal', 'Shirt', 'Sneaker', 'Bag', 'Ankle boot']
class_dict = {i:class_name for i,class_name in enumerate(class_names)}
```

In [ ]:
```
def show_image(index):
    plt.figure()
    # cmap=plt.cm.binary allows us to show the picture in grayscale
    plt.imshow(train_images[index], cmap=plt.cm.binary)
    plt.title(class_names[train_labels[index]])
    plt.colorbar() # adds a bar to the side with values
    plt.show()
```

In [ ]:
```
show_image(0)
```

Question 1: Data Preprocessing

As you can see above, the images are valued from $[0, 255]$. This is the normal range for images and excercises that we need to normalize our dataset.

In order to normalize our data to $[0, 1]$ we use the equation:

$$x_{norm} = \frac{x - x_{min}}{x_{max} - x_{min}}$$

In our case we can assume that $x_{min} = 0$ and $x_{max} = 255$, this is a safe assumption since we are working with image data.

This means that for image data, if we want to normalize to $[0, 1]$ the equation simplifies to:

$$img_{norm} = \frac{img}{255}$$

Anytime you work with image data in any kind of model you will be normalizing with this equation. Unless the range you want to normalize is different. Sometimes you want to normalize between $[-1, 1]$, for that you would use a slightly different equation.

Question 1.1) Normalizing the data

Normalize BOTH the training and testing images using the above equation.

In [ ]:
```
train_images = # YOUR CODE HERE
test_images =  # YOUR CODE HERE
```

If we show the image again, you will see the values are all scaled correctly.

In [ ]:
```
show_image(0)
```

In [ ]:
```
# Lets sample our data to see what kind of images are stored.
# see documentation for subplot here:
# https://matplotlib.org/3.2.1/api/_as_gen/matplotlib.pyplot.subplot.html
plt.figure(figsize=(10,10))
for i in range(25):
    plt.subplot(5,5,i+1)
    plt.xticks([])
    plt.yticks([])
    plt.grid(False)
    plt.imshow(train_images[i], cmap=plt.cm.binary)
    plt.xlabel(class_names[train_labels[i]])
plt.show()
```

Question 1.2) Data visualization

Since our data is composed of grayscale images (one channel) with a resolution of 28×28, we can think of this as the images existing in a $28 \times 28 = 784$ dimensional space. This means that every single image in our dataset can be represented by a vector of length 784.

Please reshape BOTH the training and testing images to be $784D$.

Hint: look into numpy.reshape().

In [ ]:
```
print(f'Before reshape, train_images shape: {train_images.shape} test_images shape:
train_images = # YOUR CODE HERE
test_images = # YOUR CODE HERE
print(f'Before reshape, train_images shape: {train_images.shape} test_images shape:
```

We create a dataframe using our training and testing data to keep everything tidy.

In [ ]:
```
# Add training data into a dataframe
img_data = {f"z{i}":train_images[:,i] for i in range(784)}
img_data["label"] = train_labels
df_img_train = pd.DataFrame(img_data)
df_img_train["class"] = df_img_train["label"].map(class_dict)
df_img_train.head()
```

In [ ]:
```
# Add test data into a dataframe
img_data = {f"z{i}":test_images[:,i] for i in range(784)}
img_data["label"] = test_labels
df_img_test = pd.DataFrame(img_data)
df_img_test["class"] = df_img_test["label"].map(class_dict)
df_img_test.head()
```

Now we have our data reshaped into the $784D$ vectors, using these we can try and visualize the space they live in. However, accurately visualizing high dimensional spaces like these would be practically impossible. We use tools like principle component analysis (PCA), and tSNE to create a projection of this $784D$ space into a more digestible $2D$ or even $3D$.

Don't worry about how they work just yet, we are just using them to visualize our data. Just run the cells to see some cool visuals.

You can direcly use the built in PCA and TSNE models from sklearn, you import them like so:

In [ ]:
```
from sklearn.manifold import TSNE
from sklearn.decomposition import PCA
```

These tools require our data to be scaled correctly.

In [ ]:
```
from sklearn.preprocessing import StandardScaler
```

In [ ]:
```
standardized_data = StandardScaler().fit_transform(train_images)
```

In [ ]:
```
n_comps = 50
pca = PCA(n_components=n_comps)
pca_features = pca.fit_transform(standardized_data)
```

In [ ]:
```
# Add data into a dataframe
pca_data = {f"z{i}":pca_features[:,i] for i in range(n_comps)}
pca_data["label"] = train_labels
df_pca = pd.DataFrame(pca_data)
df_pca["class"] = df_pca["label"].map(class_dict)
df_pca.head()
```

In [ ]:
```
print(f'Compressed dimension of {train_images.shape[1]} to {n_comps} maintaining {p
```

In [ ]:
```
model = TSNE(n_components=2, random_state=0, perplexity=30, learning_rate=200, n_it
```

In [ ]:
```
print('This may take a few minutes...')
# We are only using the first 10000 data points, this is sufficient for this applic
visualization_data = model.fit_transform(pca_features[:10000])
print('Done.')
```

In [ ]:
```
data_to_visualize = {"z1":visualization_data[:,0], "z2":visualization_data[:,1], "l
df_visualize = pd.DataFrame(data_to_visualize)
df_visualize["class"] = df_visualize["label"].map(class_dict)
df_visualize.head()
```

Projecting the classes in 2D for visualization

These images, which started out as 784 dimensional vectors, are now being projected into a $2D$ space. Don't worry about how tSNE works, it essentially tries to estimate and project distances in a high dimensional space to $2D$. Below we can see a pretty good representation of the space where our data lives.

In [ ]:
```
sns.lmplot(x='z1',
           y='z2',
           data=df_visualize,
           fit_reg=False,
           hue='class',
           height=9,
           scatter_kws={"s":50,"alpha":0.5})
```

Question 2: Linearly Separable

Some of the data is easily to separate with a line, this concept is called linear separability. Below we plot only the Ankle Boot and Trouser class. See? It's easy to draw a line between them. This makes sense because it's easy to distinguish between a shoe and pants.

In [ ]:
```
sns.lmplot(x='z1',
           y='z2',
           data=df_visualize[(df_visualize["class"] == "Ankle boot") | (df_visualiz
           fit_reg=False,
           hue='class',
           height=9,
           scatter_kws={"s":50,"alpha":0.5})
```

How do you think our logistic regression model will fare at distinguishing Ankle boots from Trousers? Please explain.

YOUR ANSWER HERE

Question 3: Not Linearly Separable

Now lets plot something harder to classify, the Pullover and Coat classes which are basically mixed together. You can't easily draw a line between them. This makes sense since a coat and a pullover are pretty similar.

In [ ]:
```
sns.lmplot(x='z1',
           y='z2',
           data=df_visualize[(df_visualize["class"] == "Pullover") | (df_visualize[
           fit_reg=False,
           hue='class',
           height=9,
           scatter_kws={"s":50,"alpha":0.5})
```

How do you think our logistic regression model will fare at distinguishing Pullovers from Coats? Please explain.

YOUR ANSWER HERE

Question 4: Logistic Regression

Recall that each image is $28 \times 28 \times 1$ matrix which we flatten to a 784-dimensional row vector. Image i looks like:

$$\mathbf{x}_i = \begin{bmatrix} x_1 & x_2 & \cdots & x_{784} \end{bmatrix} \tag{1}$$

Our dataset $\mathbf{X}$ is then the collection of all these n images.

$$\mathbf{X} = \begin{bmatrix} x_{1,1} & x_{1,2} & \cdots & x_{1,784} \\ x_{2,1} & x_{2,2} & \cdots & x_{2,784} \\ \vdots & \vdots & \ddots & \vdots \\ x_{n,1} & x_{n,2} & \cdots & x_{n,784} \end{bmatrix} \tag{2}$$

Logistic regression uses a bias term, we can easily incorporate this by adding a column of ones at the beginning.

$$\mathbf{X} = \begin{bmatrix} 1 & x_{1,1} & x_{1,2} & \cdots & x_{1,784} \\ 1 & x_{2,1} & x_{2,2} & \cdots & x_{2,784} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_{n,1} & x_{n,2} & \cdots & x_{n,784} \end{bmatrix} \tag{3}$$

For each image we have a corresponding label y_i. This is the class "Coat", "Ankle boot", "Pullover", etc. Each of which is mapped to a unique number.

$$\mathbf{Y} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix} \tag{4}$$

We will try to find the optimal parameter values $\theta = [\theta_0, \theta_1, \cdots, \theta_{784}]^T$ of our logistic regression model, where θ_0 is the bias weight. To simplify our notation, let

$$Z = X\theta = \begin{bmatrix} 1 & x_{1,1} & x_{1,2} & \cdots & x_{1,784} \\ 1 & x_{2,1} & x_{2,2} & \cdots & x_{2,784} \\ \vdots & \vdots & \ddots & \vdots & \\ 1 & x_{n,1} & x_{n,2} & \cdots & x_{n,784} \end{bmatrix} \begin{bmatrix} \theta_0 \\ \theta_1 \\ \vdots \\ \theta_{784} \end{bmatrix} = \begin{bmatrix} \theta_0 + x_{1,1}\theta_1 + \cdots + x_{1,784}\theta_{748} \\ \theta_0 + x_{2,1}\theta_1 + \cdots + x_{2,784}\theta_{784} \\ \vdots \\ \theta_0 + X_{n,1}\theta_1 + \cdots + x_{n,784}\theta_{784} \end{bmatrix}$$

Since each z_i is in the range $(-\infty, \infty)$ and our labels are $[0, 1]$ we pass z_i into the sigmoid function.

$$\sigma(z_i) = \frac{1}{1 + e^{-z_i}} = h_i \tag{6}$$

Now we can make predictions using h_i by simply rounding it to 0 or 1.

$$pred_i = round(h_i) \tag{7}$$

In order to train our logistic regression model we need to find the parameter vector θ that

In order to train our logistic regression model we need to find the parameter vector θ that minimizes our cost function J, we will be Binary Cross Entropy (BCE).

$$\mathrm{J} = -\frac{1}{n}\sum_{i=1}^{n}\left[y_i \log(h_i) + (1-y_i)\log(1-h_i)\right] \tag{8}$$

Where y_i is your true label and $h_i = \sigma(z_i)$.

In order to know whether we want to increment or decrement our weights to minimize the loss, we calculate its partial derivative with respect to weights, we call this the gradient, the matrix form of the gradient is:

$$\nabla \mathrm{J} = \frac{1}{n} X^T (H - Y) \tag{9}$$

with $X = \begin{bmatrix} 1 & x_{1,1} & x_{1,2} & \cdots & x_{1,784} \\ 1 & x_{2,1} & x_{2,2} & \cdots & x_{2,784} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_{n,1} & x_{n,2} & \cdots & x_{n,784} \end{bmatrix}$, $\mathbf{Y} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix}$, and $\mathbf{H} = \begin{bmatrix} h_1 \\ h_2 \\ \vdots \\ h_n \end{bmatrix}$

We then use this gradient in an iterative algorithm called gradient descent where every iteration we change the weights like so:

$$\theta^{(t+1)} = \theta^{(t)} - \alpha \nabla \mathrm{J} \tag{10}$$

Where t is the iteration number, α is your learning rate, $\theta^{(t)}$ is your current weight column vector, and $\theta^{(t+1)}$ is your updated weight column vector.

For this question you will implement some of the necessary functions for logistic regression:

- The BCE loss function
- The gradient/derivative of the BCE loss function
- The sigmoid function
- The predict function
- The accuracy function

You will also complete the weight update within the gradient descent algorithm.

In [ ]:
```
def get_data_subset(df, classes=[], shuffle=True, shuffle_seed=42):
  if classes == []:
    return None
  else:
    df_filtered = df[(df["class"] == classes[0]) | (df["class"] == classes[1])].cop
    df_filtered["binary_label"] = 0
    df_filtered.loc[df["class"] == classes[1], "binary_label"] = 1
    data = df_filtered.filter(regex=("z[0-9]+")).values
    labels = df_filtered["binary_label"].values
    if shuffle:
      np.random.seed(shuffle_seed)
      np.random.shuffle(data)
      np.random.seed(shuffle_seed)
      np.random.shuffle(labels)

  return data, labels.reshape(-1,1)
```

In [ ]:
```
def train_test_validation_split(X, y, test_size=.20, validation_size=.20):
    trainIdx = int((1. - test_size - validation_size)*X.shape[0])
    testIdx = int((1. - test_size)*X.shape[0])
    validationIdx = int(1.0 * X.shape[0])
    X_train, y_train = X[:trainIdx], y[:trainIdx]
    X_test, y_test = X[trainIdx:testIdx], y[trainIdx:testIdx]
    X_val, y_val = X[testIdx:validationIdx], y[testIdx:validationIdx]
    return X_train, y_train, X_test, y_test, X_val, y_val

def bias_trick(X):
    return np.hstack((np.ones((X.shape[0],1)), X))
```

In [ ]:
```
# applying sigmoid function (equation 6) to the vector input of z_i's
def sigmoid(z):
    sig_z = # YOUR CODE HERE
    return sig_z

# implementing equation 8
def binary_cross_entropy(y, h):
  n = len(y)
  return # YOUR CODE HERE

# returns the derivation of the BCE function based on equation 9
def binary_cross_entropy_derivative(X, y, theta):
    z = # YOUR CODE HERE
    h = # YOUR CODE HERE
    n = len(y)
    return # YOUR CODE HERE

# returns percentage of correct predictions
def accuracy(y, h):
    return # YOUR CODE HERE

# returns predictions for all inputs. Given input is h_i based on equations 7.
def predict(h):
    return # YOUR CODE HERE


def logistic_regression(X, y, learning_rate, num_steps):

    # split your data into train and test subsets. Don't forget to apply the bias t
    X = bias_trick(X)
    X_train, y_train, X_test, y_test, X_val, y_val = train_test_validation_split(X,

    # start with intial parameters theta_i = 0, how many parameters/weights do we n
    theta = # YOUR CODE HERE


    z = np.dot(X_train,theta)
    h = sigmoid(z)
    print(f'Initial Accuracy:{accuracy(y_train, predict(h)):.4f}')
    val_losses = []
    train_losses = []
    val_accuracies = []
    train_accuracies = []

    for step in range(num_steps):
        # Calculate the current output of your logistic network (h_train and h_val)
        z_train = # YOUR CODE HERE
        h_train = # YOUR CODE HERE

        z_val = # YOUR CODE HERE
        h_val = # YOUR CODE HERE

        # Calculate your current training/validation accuracy and BCE loss
        training_accuracy = # YOUR CODE HERE
        training_loss = # YOUR CODE HERE
        validation accuracy = # YOUR CODE HERE
```

```
        validation_loss = # YOUR CODE HERE

        val_losses.append(validation_loss)
        val_accuracies.append(validation_accuracy)
        train_losses.append(training_loss)
        train_accuracies.append(training_accuracy)

        # Calculate the gradient using the derivative of your loss function.
        gradient = # YOUR CODE HERE

        # Adjust your weights
        theta = # YOUR CODE HERE

        print(f'Epoch [{step+1}/{num_steps}] '.ljust(20) + f'loss: {training_loss:.
        print(f"val_loss: {validation_loss:.4f} - val_accuracy: {validation_accurac
    return theta, train_losses, train_accuracies, val_losses, val_accuracies
```

Now that we have our logistic implementation complete, let's try and classify Ankle boots against Trousers.

We get the subset of the data containing these classes using the function defined above.

Don't change the learning rate or number of steps.

In [ ]:
```
X, y = get_data_subset(df_img_train, classes=["Trouser", "Ankle boot"])
```

In [ ]:
```
# train the model until it converges, use the plotted losses below to verify
learning_rate = 0.05
num_steps = 100
theta, train_losses, train_accuracies, val_losses, val_accuracies = logistic_regres
```

In [ ]:
```
import matplotlib.pyplot as plt
# plot your training accuracy and validation accuracy curves together
plt.plot(train_accuracies)
plt.plot(val_accuracies)
plt.show()

# plot your training losses and validation losses curves together
plt.plot(train_losses)
plt.plot(val_losses)
plt.show()
```

Question 4.1) Observation

What accuracy did it reach? Why do you think it reached this accuracy? How long did it take?

YOUR ANSWER HERE

Now train it to classify Pullover vs. Coat.

In [ ]:
```
X, y = get_data_subset(df_img_train, classes=["Pullover", "Coat"])
```

In [ ]:
```
# train the model until it converges, use the plotted losses below to verify
# modify learning_rate and num_steps to accomplish this
learning_rate = 0.01
num_steps = 100
theta, train_losses, train_accuracies, val_losses, val_accuracies = logistic_regres
```

In [ ]:
```
import matplotlib.pyplot as plt
# plot your training accuracy and validation accuracy curves together
plt.plot(train_accuracies)
plt.plot(val_accuracies)
plt.show()

# plot your training losses and validation losses curves together
plt.plot(train_losses)
plt.plot(val_losses)
plt.show()
```

Question 4.2) Observation

What accuracy did it reach? Why do you think it reached this accuracy? How long did it take?

YOUR ANSWER HERE

Question 4.3) Comparison

Compare and contrast both runs above, are they the same? Different? Then explain why this is the case.

YOUR ANSWER HERE

Question 5: Neural Networks

In this question we will build a neural network with Keras that can beat our logistic regression model at classifying Pullovers vs. Coats.

In [ ]:
```
from tensorflow.keras.layers import Input, Dense # only use these layers
from tensorflow.keras.models import Model
from tensorflow.keras.optimizers import * # you can use any optimizer

# Define an input layer with the correct shape for your data
input_layer = # YOUR CODE HERE


# Pass the input layer's output to a dense of size 100, choose whatever activation
x = # YOUR CODE HERE

# Pass the previouses hidden layer's output to a dense of size 1 for classification
output = # YOUR CODE HERE

# Define a model with it's input as your input layer and output as your output laye
model = # YOUR CODE HERE
```

In [ ]:
```
# Show a summary of your model
model.summary()
```

In [ ]:
```
# Compile your model with your chosen optimizer, binary cross entropy for the loss,

# YOUR CODE HERE
```

In [ ]:
```
X, y = get_data_subset(df_img_train, classes=["Pullover", "Coat"])

# Call fit on your model passing in the X, y data above with validation split of 0.
hist = # YOUR CODE HERE
```

In [ ]:
```
def plot_losses(hist):
    plt.plot(hist.history['loss'])
    plt.plot(hist.history['val_loss'])
    plt.title('Model Loss')
    plt.ylabel('Loss')
    plt.xlabel('Epoch')
    plt.legend(['Train', 'Val'])
    plt.show()
def plot_accuracies(hist):
    plt.plot(hist.history['accuracy'])
    plt.plot(hist.history['val_accuracy'])
    plt.title('Model Accuracy')
    plt.ylabel('Accuracy')
    plt.xlabel('Epoch')
    plt.legend(['Train', 'Val'])
    plt.show()
```

In [ ]:
```
# plot your losses and accuracies
plot_losses(hist)
plot_accuracies(hist)
```

Question 5.1) Observation

How did your neural network perform? What hyperparameters and optimizer did you choose to beat logistic regression? Why do you think your neural network beat your logistic regression model?

YOUR ANSWER HERE