ECE566 | Project 1

Due date: 11/11/2022

Anas Puthawala - A20416308
Professor Jovan Brankov

Introduction & Background

The goal of this report is to go over a medium - high level overview of the two main components discussed in this project:

1. Part A which consists of applying the Fisher Discriminant to classify between two distinct classes.
2. Part B consists of a neural network from scratch including coding up a loss function, implementing gradient descent, and training the network to classify between two distinct classes.

The goal is to implement these two algorithms by 'scratch' to get a deeper understanding of how they work and compare and contrast the difference in terms of performance between the two.

Background | Data Description

The data that we're working with is the mnist dataset. The mnist dataset consists of handwritten digits from 0-9. It can be used as a multi-class classification problem or binary classification if we take two specific classes only. The model from *Part B* was developed to be used for binary classification where we are separating between two digits (handwritten 0s and 1s and 5s and 6s). In addition, when utilizing the data, there are steps performed such as standardizing the data by dividing the pixel values by 255. and reshaping the data by flattening it and collapsing the two dimensions of 28x28 into a single input vector consisting of 784 units.

Background | Fisher Discriminant Task

The Fisher discriminant has a few moving parts to it. The overall goal is if you have features from a dataset and you want to classify wether something is 0 or 1 you can use the features and project the data down to a lower dimension and have a discriminant that classifies on the projected data to classify wether something belongs to class 0 or 1. This is similar to probabilistic or a gaussian approach. I unfortunately wasn't able to complete this portion due to some inconsistencies in my code but there are several approaches I tried. The figure describes the general outcome of applying Fisher Discriminant to decrease the dimensionality and improve separation:

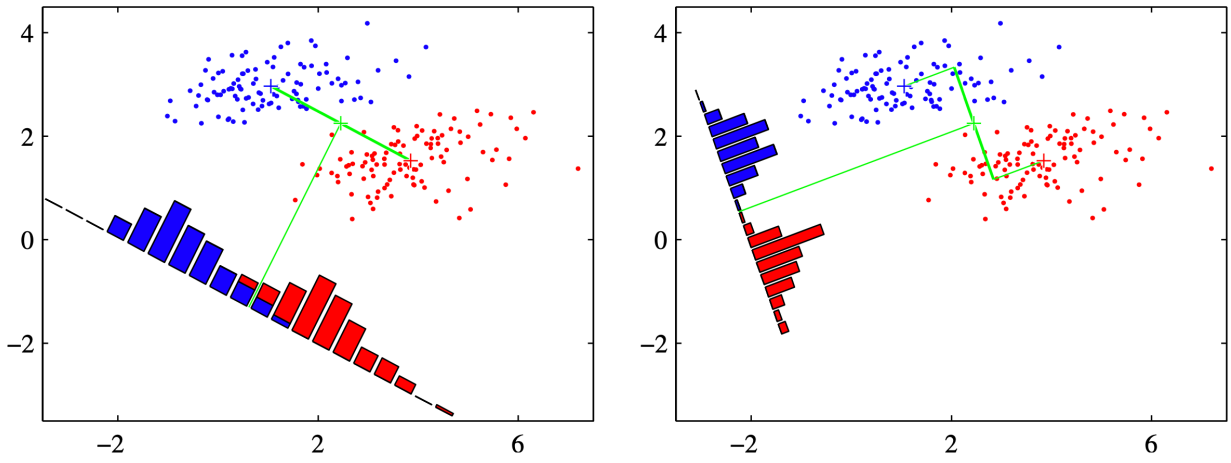


Fig. 1 - Improving separation via Fisher Discriminant [1]

The figure on the left shows the before, whereas the right shows the after. Now we decide on some threshold y_o to separate the classes and we can classify them based on the following rule:

decide C_1 if $y(x) \geq y_o$ else decide C_2 — Equation. (1)

Additionally, the Fisher criterion is defined to be the ratio of the between-class variance to the within-class variance and is given by:

$$J(w) = \frac{(m_2 - m_1)^2}{s_1^2 + s_2^2} \text{ — (2)}$$

Where S_i is the scatter for the respective classes and m is the means. The goal is ultimately to project the data to another axis and we can do that by multiply the feature vector x_n by some matrix w^T to get y_n. The matrix w^T can be approximated via the following:

$$w \simeq S_w^{-1}(m_2 - m_1) \text{ — (3)}$$

$$S_w = \sum_{n \varepsilon C_1} (x_n - m_1)(x_n - m_1)^T + \sum_{n \varepsilon C_2} (x_n - m_2)(x_n - m_2)^T \text{ — (4)}$$

and in this case $w \simeq \underset{w}{\operatorname{argmin}}(J(w))$

Background | Neural Network Task

The goal here is to build out a logistic regression model in the framework of a neural network to separate two classes of data. The input of the network is 784 inputs (a 28 x 28 image), there are no neurons, no hidden layers, just a single output which takes a weighted sum of the input and applies a sigmoid activation to it for binary classification:

$$\hat{y} = sigmoid(w^T x^i + b) \text{ — (5)}$$

The loss function we use to train the network is binary crossentropy:

$$L(\hat{y}^{(i)}, y^{(i)}) = -y^{(i)} log(\hat{y}^{(i)}) - (1 - y^{(i)}) log(1 - \hat{y}^{(i)}) \text{ — (6)}$$

and the cost function is simply the average of the binary cross entropy for all *m* images in the training set.

$$J = \frac{1}{m} \sum_{i=1}^{m} L(\hat{y}, y) \text{ — (7)}$$

The functions that were implemented were the sigmoid function, the propagate function (pushes the data through the network and gets the cost and dw and db), gradient descent which optimizes the W and B by running the gradient descent algorithm which updates the parameters W and B based on the following update rules:

$$w = w - \eta(\frac{dL}{dW}) \text{ — (8)}$$

$$b = b - \eta(\frac{dL}{db}) \text{ — (9)}$$

where w is the weights, b is the bias, η is the learning rate and dw, db are the gradients propagated back and adjusted based on the loss. the gradients dw, db can be calculated as:

$$\frac{dL}{dW} = \frac{dL}{dy} * \frac{dy}{dz} * \frac{dz}{dW} \text{ — (10)}$$

$$\frac{dL}{db} = \frac{dL}{dy} * \frac{dy}{dz} * \frac{dz}{db} \text{ — (11)}$$

Where z is simply the weighted sum:

$$z = w^T x^i + b \text{ — (12)}$$

With these equations we can implement back-prop and update the parameters based on the gradients to come up with the optimal parameters for the network. The weights and bias get initialized to 0 and the model begins training with the training data from the classes given. Once the model is trained, we evaluate the performance of the model by utilizing the test set which consists of unseen data from the same classes. We can compute the accuracy after we get test set predictions. Another step is to re-train the model this time with classes separating handwritten 5's from 6's and computing the balanced accuracy which is the following:

$$\frac{1}{2}\left(\frac{TP}{TP + FN} + \frac{TN}{TN + FP}\right) \text{ — (13)}$$

Where TP is the true positive between the predicted labels and the test labels (i.e. every positive class that was predicted was correct between the predicted label), TN is the true negative between the predicted labels and the test labels (i.e. how often the model correctly predicts all from the negative class), FN is the false negative between the predicted labels and the test labels (i.e. how often does the model incorrectly predict a positive when in reality it should've been a negative) and FP is the false positive between the predicted labels and the test labels (how often the model incorrectly predicted that a positive when in reality it should've been a negative). The balanced accuracy is useful for imbalanced datasets and holistically is just a better approach to gauge model performance since the accuracy can be misleading as a metric to gauge model performance with.

Results & Discussion

Results & Discussion | Part A: Fisher Discriminant

For this part, the goal was to apply fisher discriminant to separate between handwritten 0s and 1s and then use the same discriminant function to evaluate the performance on the test set. Afterwards, using the same discriminant function, evaluate the performance from separating handwritten 5s from 6s.

Choosing features

The features I chose for applying the fisher discriminant are area and perimeter. I think these are excellent features for having a fisher discriminant learn to separate between the classes simply because I was able to extract these features for the 0s and 1s and visualize them via a plot and be able to tell that they are indeed valid features for this specific task:

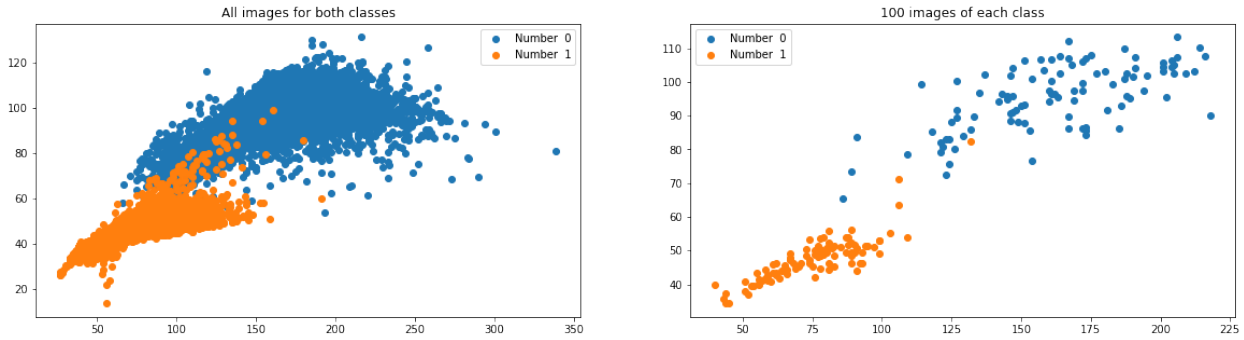


Fig. 2 - Scatter plot of features | Area & Perimeter for classes handwritten 0s and 1s

The x-axis represents the perimeter and the x-axis represents the area. It's evident that these are good features to use, however they don't need to be constrained to this solely. The features were extracted using the skimage library more specifically the 'reigonprops' module of that library. The total sample size initially was 12665, with 5923 samples belonging to class label '0' and 6752 samples belonging to class label '1'.
Unfortunately, I wasn't able to continue with this section and complete it since I was encountering some errors with my fisher discriminant portion. I have tried the following things:

1. I didn't use any features to start with, just simply the entire dataset with the different labels and tried to calculate the W and scatter matrices Sw.
2. I used the features for area and perimeter and concatenated them to build a feature vector and tried to perform the fisher discriminant analysis on that. That didn't work because when calculating Sw I was receiving matrices for the summation terms and they were of uneven size. To counter this, I shaved some samples off from class label '1' to make it the same number of samples as class label '0' to actually perform the matrix addition. However i got stuck on the last step, where I'd multiply Sw^{-1} by $(m_2 - m_1)$ since $(m_2 - m_1)$ was a matrix of size (2,) and Sw^{-1} was of size (5923, 5923).

The next steps would've been to simply project the initial feature vector using the new matrix W that I would've calculated from the equation up above and define a threshold from which you can perform classification based on that. You can then use the testing data to apply the discriminant function and get actual y_predictions and compare to the y_test and get the accuracy and the balanced accuracy.
One last step in Part A consisted of analyzing handwritten 5's from 6's and comparing the analysis (balanced accuracy, etc.) of this the previous results. Since I didn't get a previous result I can attempt to make an analysis by relying on the scatter plot of area & perimeter for the handwritten 5's and 6's.

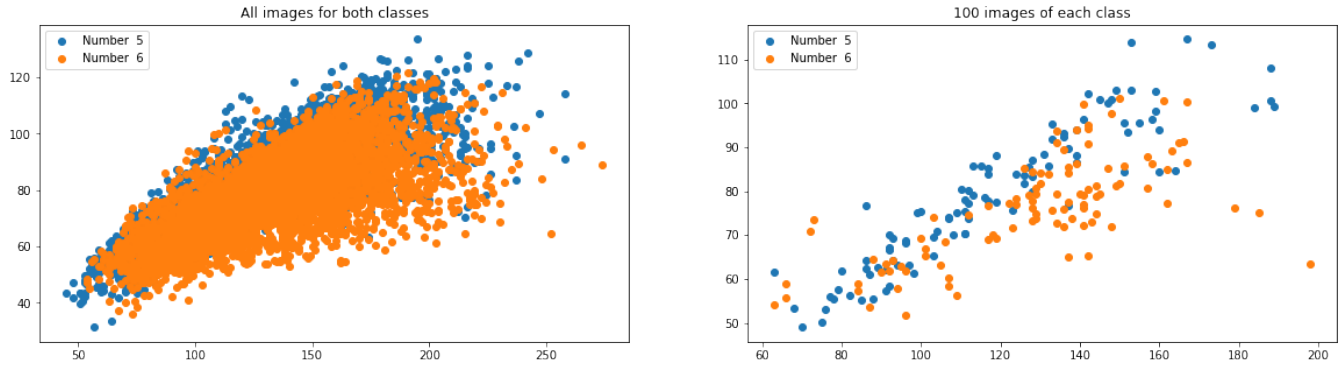


Fig. 3 - Scatter plot of features | Area & Perimeter for classes handwritten 5s and 6s

Just visually interpreting the scatter plot, I can conclude that these won't be good features to use for fisher discriminant function and analysis. There's too much overlap and the function won't be able to accurately separate the two as seen from Fig. 1 and 2.

Results & Discussion | Part B: Neural Network

Initial pre-processing for this section consisted of loading in the image data from mnist, standardizing by dividing image vectors by 255 and reshaping the images to pass in as 784 units instead of a 28 x 28 image. For initiating gradient descent, the weights and bias were set to 0, the learning rate was held at 0.005, and the model was trained for 2000 iterations.

Training a model to separate handwritten 0s from 1s - Results

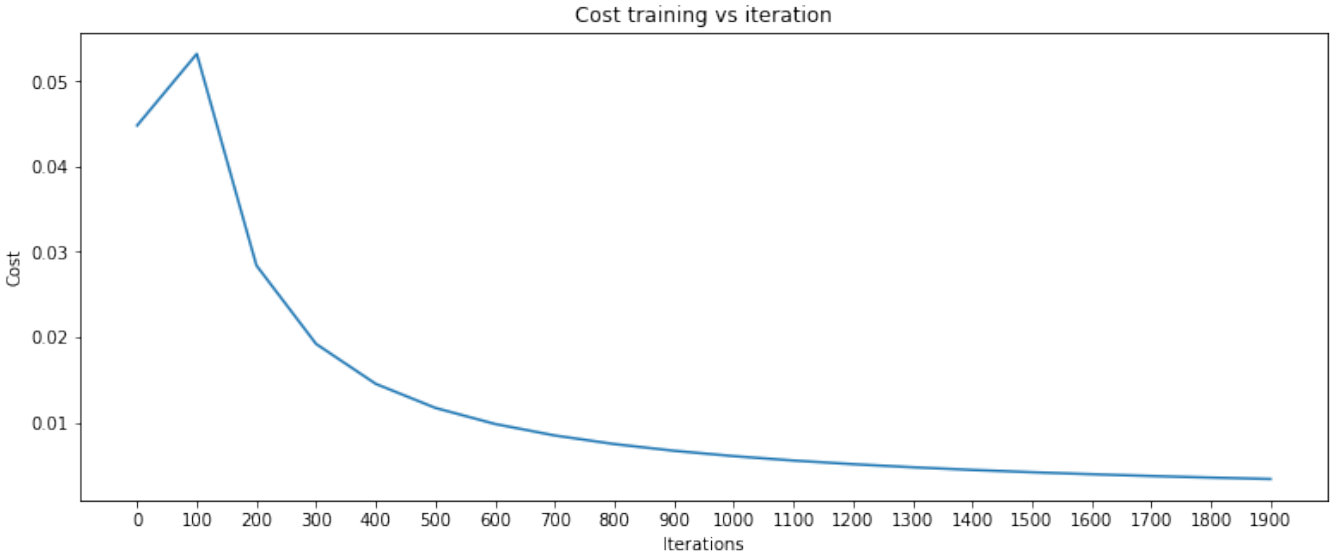


Fig. 4 - Training curve for model | Separating 0's and 1's

The cost decrease became rather constant after around 1900 iterations, so 2000 iterations training is a good point. The lowest cost was .003389. The model was then used to make predictions on the test set and once we had the y_predictions we can evaluate the testing accuracy. The training accuracy was ~99.74% and the testing accuracy was ~99.91%. The balanced accuracy was ~99.9%

Training a model to separate handwritten 5s from 6s - Results

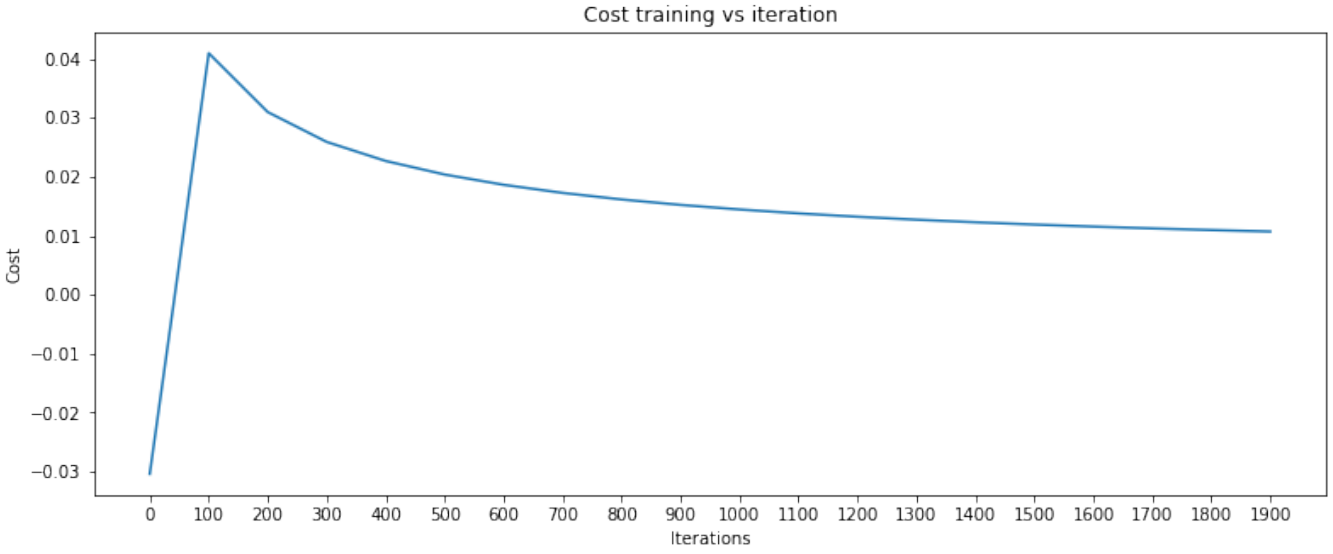


Fig. 5 - Training curve for model | Separating 5's and 6's

The model's cost for some reason started very low and then shot up before slowly decreasing. It seems that after around ~1400 iterations the decrease was minimal and we can consider that instead of the 2000 iterations a good stopping point. The testing accuracy was ~97.24%, the training accuracy was 96.77%, the balanced accuracy was 97.2%. The following table briefly summarizes results from this part of the project:

	Seperating 0's and 1's	**Seperating 5's and 6's**
Training Accuracy	99.74%	96.77%
Testing Accuracy	99.91%	97.24%
Balanced Accuracy	99.9%	97.2%
Lowest Cost	0.003389	0.010716

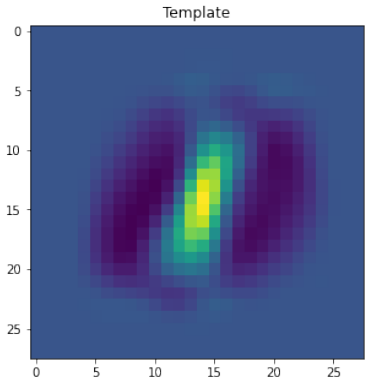
Conclusion

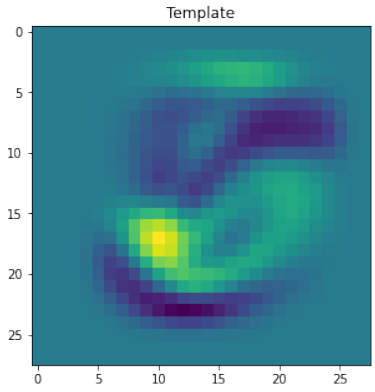
Conclusion | Part A: Fisher Discriminant

There's a clearly defined process for applying the fisher discriminant. The benefits are that it reduces dimensionality by projecting the feature set to a lower dimension and therefore if we choose the correct features we should be able to make it easier for the linear discriminant function to separate the two classes. Unfortunately, I was unable to get my code working for the fisher discriminant, but I laid out the required steps and procedure in the *Introduction* and some in the *Results & Discussion* section of this report. The features we used were area and perimeter and by visualizing the scatter plots of the area and perimeter for class 0 and class 1 they were seemingly excellent choices – for class 5 and 6 though, not really a good choice because of the sheer amount of overlap between the features and the two classes.

Conclusion | Part B: Neural Network

The model had a much better time separating 0s from 1s, as expected. A 0 and a 1 is much more distinct than a 5 and a 6 which could easily be confused for one another. The following images are the templates for 0s and 1s (left) and for 5s and 6s (right)





We can see how the template for 5s and 6s looks like a mess and can begin to think about how the model may have a harder time separating the two. Despite that, it still performed sufficiently well in both cases, with the case in separating handwritten 0's and 1's outperforming the case when we try and separate 5's and 6's.

Some advantages of a neural network over fisher discriminant method may be that in a neural network approach, the features are learned automatically and are embedded in the weights which may improve performance and save time. What we effectively do with the fisher discriminant is that we manually are trying to pick the best features and then applying the analysis. This can have several disadvantages because our features that we pick are only as good as the engineer that decides to pick those features, and that it can be time consuming to pick the right feature.

Some disadvantages of using a neural network over the fisher discriminant may be that it can begin to require a lot more computing power to train a model. In our case it wasn't noticeable, but if we begin to scale-up then computing power and cost associated with that can be problematic. In addition, if the engineer is good and he/she has valuable industry experience then they may be able to hand-craft features which can separate two classes much better than what a network may learn. And that's because of the fact that they're able to generate the new features via feature engineering as opposed to relying on the given features and just throwing the data at a neural network.

References

[1] MLA. Bishop, Christopher M. Pattern Recognition and Machine Learning. New York :Springer, 2006.

Appendix

```
# -*- coding: utf-8 -*-
"""NN_FromScratch_&LDA.ipynb

Automatically generated by Colaboratory.

Original file is located at
    https://colab.research.google.com/drive/
1RPPJh_8VIkrZoTByPGXp1CCPD05jb2wE

## Load packages
"""

import numpy as np
import matplotlib.pyplot as plt
from keras.datasets import mnist

from skimage import measure

from IPython import display #note this is only to display images (my
screenshots)

"""## Define functions

### Sigmoid
"""

def sigmoid(z):
    """
    Compute the sigmoid of z

    Arguments:
    x -- A scalar or numpy array of any size.

    Return:
    s -- sigmoid(z)
    """

    s = (1) / (1+np.exp(-z))

    return s

"""### Initialize weights"""
```

```
def initialize_weights(dim):
    """
    This function creates a vector of zeros of shape (dim, 1) for w
and initializes b to 0.

    Argument:
    dim -- size of the w vector we want (or number of parameters in
this case)

    Returns:
    w -- initialized vector of shape (dim, 1)
    b -- initialized scalar (corresponds to the bias)
    """
    w = np.zeros(shape=(dim, 1))
    b = 0

    assert(w.shape == (dim, 1))
    assert(isinstance(b, float) or isinstance(b, int))

    return w, b

"""### Forward and backward propagation"""

def propagate(w, b, X, Y):
    """
    Implement the cost function and its gradient for the propagation
explained in the assignment

    Arguments:
    w -- weights, a numpy array of size (num_px * num_px, 1)
    b -- bias, a scalar
    X -- data of size (number of examples, num_px * num_px)
    Y -- true "label" vector of size (1, number of examples)

    Return:
    cost -- negative log-likelihood cost for logistic regression
    dw -- gradient of the loss with respect to w, thus same shape as
w
    db -- gradient of the loss with respect to b, thus same shape as
b

    """

    m = X.shape[0]

    # FORWARD PROPAGATION (FROM X TO COST)
```

```
    y_hat = sigmoid(np.dot(w.T, X.T) + b)

    #-y(") log/y"(")0 - /1 - y(")0 log/1 - y"(")0
    term1 = np.dot(-Y, (np.log(y_hat)).T)
    term2 = np.dot(1-Y, (np.log(1-y_hat)).T)
    cost = (-1/m) * np.sum(term1 + term2)

    # BACKWARD PROPAGATION (TO FIND GRAD)
    # calcualte dw and db
    # dw = (dj/dy_hat) * (dy_hat/dz) * (dz/dw) or simply dj/dw
    # db = (dj/dy_hat) * (dy_hat/dz) * (dz/db) or simply dj/dz where
j = loss

    dw = (1/m) * np.dot(X.T,(y_hat-Y).T)
    db = (1/m) * np.sum((y_hat-Y))

    assert(dw.shape == w.shape)
    assert(db.dtype == float)
    cost = np.squeeze(cost)
    assert(cost.shape == ())

    grads = {"dw": dw,
             "db": db}

    return grads, cost

"""### Gradient descent"""

def gradient_descent(w, b, X, Y, num_iterations, learning_rate):
    """
    This function optimizes w and b by running a gradient descent
algorithm

    Arguments:
    w -- weights, a numpy array of size (num_px * num_px, 1)
    b -- bias, a scalar
    X -- data of shape (num_px * num_px, number of examples)
    Y -- true "label" vector of shape (1, number of examples)
    num_iterations -- number of iterations of the optimization loop
    learning_rate -- learning rate of the gradient descent update
rule

    Returns:
    params -- dictionary containing the weights w and bias b
```

```
    grads -- dictionary containing the gradients of the weights and
bias with respect to the cost function
    costs -- list of all the costs computed during the optimization,
this will be used to plot the learning curve.

    Tips:
    You basically need to write down two steps and iterate through
them:
        1) Calculate the cost and the gradient for the current
parameters. Use propagate().
        2) Update the parameters using gradient descent rule for w
and b.
    """

    costs = []

    for i in range(num_iterations):


        # Cost and gradient calculation

        grads, cost = propagate(w, b, X, Y)

        # Retrieve derivatives from grads
        dw = grads["dw"]
        db = grads["db"]

        # update rule

        w -= learning_rate * dw
        b -= learning_rate * db

        # Record the costs
        if i % 100 == 0:
            costs.append(cost)
            # Print the cost every 100 training examples
            print ("Cost after iteration %i: %f" % (i, cost))

    params = {"w": w,
              "b": b}

    grads = {"dw": dw,
             "db": db}

    return params, grads, costs
```

```
"""### Make predictions"""

def predict(w, b, X):
    '''
    Predict whether the label is 0 or 1 using learned logistic
regression parameters (w, b)

    Arguments:
    w -- weights, a numpy array of size (num_px * num_px, 1)
    b -- bias, a scalar
    X -- data of size (num_px * num_px, number of examples)

    Returns:
    Y_prediction -- a numpy array (vector) containing all predictions
(0/1) for the examples in X
    '''

    m = X.shape[0]
    Y_prediction = np.zeros((1, m))
    w = w.reshape(X.shape[1], 1)

    # Compute vector "A" predicting the probabilities of the picture
containing a 1

    A = sigmoid(np.dot(w.T, X.T) + b)


    for i in range(A.shape[1]):
        # Convert probabilities A[0,i] to actual predictions p[0,i]

        # if it's greater than or equal to 0.5, we are classifying as
1, else 0.
        if A[0][i] >= 0.5:
            Y_prediction[0][i] = 1
        else:
            Y_prediction[0][i] = 0

    assert(Y_prediction.shape == (1, m))

    return Y_prediction

"""## Merge functions and run your model"""

# LOAD DATA
class0 = 0
class1 = 1
```

```
(x_train, y_train), (x_test, y_test) = mnist.load_data()
x_train = x_train[np.isin(y_train,[class0,class1]),:,:]
y_train = 1*(y_train[np.isin(y_train,[class0,class1])]>class0)
x_test = x_test[np.isin(y_test,[class0,class1]),:,:]
y_test = 1*(y_test[np.isin(y_test,[class0,class1])]>class0)

# RESHAPE

x_train_flat = x_train.reshape(x_train.shape[0],-1)
print(x_train_flat.shape)
print('Train: '+str(x_train_flat.shape[0])+' images and
'+str(x_train_flat.shape[1])+' neurons \n')

x_test_flat = x_test.reshape(x_test.shape[0],-1)
print(x_test_flat.shape)
print('Test: '+str(x_test_flat.shape[0])+' images and
'+str(x_test_flat.shape[1])+' neurons \n')

# STRANDARIZE
x_train_flat = x_train_flat / 255
x_test_flat = x_test_flat / 255

"""### Train the model (in training set)"""

# Initialize parameters with zeros (≈ 1 line of code)
w, b = initialize_weights(x_train_flat.shape[1])

# Gradient descent (≈ 1 line of code)
learning_rate = 0.005
num_iterations = 2000
parameters, grads, costs = gradient_descent(w, b, x_train_flat,
y_train, 2000, 0.005)

"""### Test the model (in testing set)"""

# Retrieve parameters w and b from dictionary "parameters"
w = parameters["w"]
b = parameters["b"]

# Predict test/train set examples (≈ 2 lines of code)
y_prediction_test = predict(w, b, x_test_flat)
y_prediction_train = predict(w, b, x_train_flat)

# Print train/test Errors
print('')
```

```
print("train accuracy: {} %".format(100 -
np.mean(np.abs(y_prediction_train - y_train)) * 100))
print("test accuracy: {} %".format(100 -
np.mean(np.abs(y_prediction_test - y_test)) * 100))
print('')

plt.figure(figsize=(13,5))
plt.plot(range(0,2000,100),costs)
plt.title('Cost training vs iteration')
plt.xlabel('Iterations')
plt.ylabel('Cost')
plt.xticks(range(0,2000,100))


plt.figure(figsize=(13,5))
plt.imshow(w.reshape(28,28))
plt.title('Template')

def balanced_accuracy(y_true, y_pred):
    y_true = y_true.squeeze()
    y_pred = y_pred.squeeze()
    def tp(y_true, y_pred):
        '''calculates true positive'''
        ct = 0
        for idx in range(len(y_pred)):
            if y_pred[idx]==1 and y_true[idx]==1:
                ct+=1
        return ct

    def tn(y_true, y_pred):
        '''calculate true negative'''
        ct = 0
        for idx in range(len(y_pred)):
            if y_pred[idx]==0 and y_true[idx]==0:
                ct+=1
        return ct

    def fp(y_true, y_pred):
        '''calculates false positives (predicting 1 when its actually
0)'''
        ct = 0
        for idx in range(len(y_pred)):
            if y_pred[idx]==1 and y_true[idx]==0: #predicted positive
when in reality it was negative
                ct+=1
        return ct
```

```
    def fn(y_true, y_pred):
        '''calculates false negatives (predicting 0 when it's
actually 1)'''
        ct = 0
        for idx in range(len(y_pred)):
            if y_pred[idx]==0 and y_true[idx]==1:
                ct+=1
        return ct

    # Now we can return the balanced classification accuracy
    sensitivity = tp(y_true, y_pred) / (tp(y_true, y_pred) +
fn(y_true, y_pred))
    specificity = tn(y_true, y_pred) / (tn(y_true, y_pred) +
fp(y_true, y_pred))

    return np.round(0.5 * (sensitivity + specificity), 3) * 100

balanced_accuracy(y_true=y_test, y_pred=y_prediction_test)

"""Pretty good balanced accuracy for seperating 0s and 1s

### Re-training network and evaluating for seperating `5`s and `6`s
"""

# LOAD DATA
class0 = 5
class1 = 6

(x_train, y_train), (x_test, y_test) = mnist.load_data()
x_train = x_train[np.isin(y_train,[class0,class1]),:,:]
y_train = 1*(y_train[np.isin(y_train,[class0,class1])]>class0)
x_test = x_test[np.isin(y_test,[class0,class1]),:,:]
y_test = 1*(y_test[np.isin(y_test,[class0,class1])]>class0)

# RESHAPE

x_train_flat = x_train.reshape(x_train.shape[0],-1)
print(x_train_flat.shape)
print('Train: '+str(x_train_flat.shape[0])+' images and
'+str(x_train_flat.shape[1])+' neurons \n')

x_test_flat = x_test.reshape(x_test.shape[0],-1)
print(x_test_flat.shape)
print('Test: '+str(x_test_flat.shape[0])+' images and
'+str(x_test_flat.shape[1])+' neurons \n')
```

```
# STRANDARIZE
x_train_flat = x_train_flat / 255
x_test_flat = x_test_flat / 255

# Initialize parameters with zeros (≈ 1 line of code)
w, b = initialize_weights(x_train_flat.shape[1])

# Gradient descent (≈ 1 line of code)
learning_rate = 0.005
num_iterations = 2000
parameters, grads, costs = gradient_descent(w, b, x_train_flat,
y_train, num_iterations, learning_rate)

# Retrieve parameters w and b from dictionary "parameters"
w = parameters["w"]
b = parameters["b"]

# Predict test/train set examples (≈ 2 lines of code)
y_prediction_test = predict(w, b, x_test_flat)
y_prediction_train = predict(w, b, x_train_flat)

# Print train/test Errors
print('')
print("train accuracy: {} %".format(100 -
np.mean(np.abs(y_prediction_train - y_train)) * 100))
print("test accuracy: {} %".format(100 -
np.mean(np.abs(y_prediction_test - y_test)) * 100))
print('')

plt.figure(figsize=(13,5))
plt.plot(range(0,2000,100),costs)
plt.title('Cost training vs iteration')
plt.xlabel('Iterations')
plt.ylabel('Cost')
plt.xticks(range(0,2000,100))


plt.figure(figsize=(13,5))
plt.imshow(w.reshape(28,28))
plt.title('Template')

y_preds = predict(w, b, x_test_flat)
```

```
"""Comparing the accuracy for the test set we can see that there is a
clear decrease in performance when comparing it to the classifier for
0s and 1s.
We had almost 99.5% accuracy across the board for 0s and 1s
"""



balanced_accuracy(y_true=y_test, y_pred=y_preds)

display.Image('/content/bal_acc.png')

"""97.7% is the balanced accuracy in this case. Not bad, but
definitely less than 99.9 percent that we had with seperating 0s and
1s

## Fisher Discriminant
"""

# Collect features from 1's and 0's -- training data only
# LOAD DATA
class0 = 0
class1 = 1

(x_train, y_train), (x_test, y_test) = mnist.load_data()
x_train = x_train[np.isin(y_train,[class0,class1]),:,:]
y_train = 1*(y_train[np.isin(y_train,[class0,class1])]>class0)
x_test = x_test[np.isin(y_test,[class0,class1]),:,:]
y_test = 1*(y_test[np.isin(y_test,[class0,class1])]>class0)

numzeros = len(x_train[y_train==0])
numones = len(x_train[y_train==1])
print(f'Number of 0s in x trian: {numzeros}')
print(f'Number of 1s in x trian: {numones}')
print(f'Percentage of 0s in x train: {np.round(numzeros /
len(x_train), 2) * 100}%')
print(f'Percentage of 1s in x train: {np.round(numones /
len(x_train), 2) * 100}%')

"""We need a few things to be able to calculate the objective
function for good seperation
1. Between-Class-Scatter --> Distance between means (so we need the
means of the clusters)
2. Within-Class-Scatter
"""
```

```
# First let's calculate between class scatter (distance between
means)

# m1 = class 0
m1 = np.mean(x_train[y_train==0], axis=0)
plt.imshow(m1) # visualize average image for mean of class 0

#m2 = class 1
m2 = np.mean(x_train[y_train==1], axis=0)
plt.imshow(m2) # visualize average image for mean of class 1

"""w ∝ S_W^-1 ∗ (m2 − m1)"""

term_2 = m2-m1 # this is the term 2.
plt.imshow(m2 - m1)

"""Need to calculate S_w and takes it inverse and multiply it by
term_2 (distance between means) to get our W."""

display.Image('/content/Sw.png')

"""Need to calculate Sw.

Left term is for C1, right term is for C2
C1 is Class 0 for us (all the 0s) and C2 is class 1 for us (all the
1s)
"""

#instantiate np arrays of zeros
term1_arr = term2_arr = np.zeros(shape=(28, 28))

for i in range(numzeros):
    term1_arr += np.multiply((x_train[y_train==0][i] - m1),
(x_train[y_train==0][i] - m1).T)

for i in range(numones):
    term2_arr += np.multiply((x_train[y_train==1][i] - m2),
(x_train[y_train==1][i] - m2).T)
# np.multiply((x_train[y_train==0] - m1), (x_train[y_train==0] -
m1).T), axis=1)

Sw = term1_arr + term2_arr
print(f'Shape of Sw: {Sw.shape}')
plt.imshow(Sw);

display.Image('/content/sw2.png')
```

```
W = np.linalg.pinv(Sw) * (m2 - m1) #note: I had to use np.linalg.pinv
instead of np.linalg.inv because for some reason Sw was a singualar
matrix for me

np.shape(W)

"""Now that we have calculated our W, we can compress the
dimensionality down as such:

"""

display.Image('/content/Screenshot 2022-11-09 at 9.34.54 PM.png')

W.T @ x_train[y_train==0][0]

"""### This approach doesnt seem correct. I didn't consider features
from the samples. Let's try it again but consider features:"""

number = 0

x5 = x_train[y_train==number,:,:]
x6 = x_train[y_train==number+1,:,:]
buf5 = "Number  %d" % number
buf6 = "Number  %d" % (number+1)
# Threshold images
t5 = 1*(x5 > 60)
t6 = 1*(x6 > 60)

# Region properties
area5 = np.zeros(t5.shape[0])
perimeter5 = np.zeros(t5.shape[0])
for i in range(0,t5.shape[0]):
  props = measure.regionprops(t5[i,:,:])
  area5[i] = props[0].area
  perimeter5[i] = props[0].perimeter

area6 = np.zeros(t6.shape[0])
perimeter6 = np.zeros(t6.shape[0])
for i in range(0,t6.shape[0]):
  props = measure.regionprops(t6[i,:,:])
  area6[i] = props[0].area
  perimeter6[i] = props[0].perimeter

plt.figure(figsize=(20,5))
plt.subplot(1,2,1)
```

```
plt.scatter(area5,perimeter5, label=buf5)
plt.scatter(area6,perimeter6, label=buf6)
plt.title('All images for both classes')
plt.legend()

plt.subplot(1,2,2)
plt.scatter(area5[0:100],perimeter5[0:100], label=buf5)
plt.scatter(area6[0:100],perimeter6[0:100], label=buf6)
plt.title('100 images of each class')
plt.legend()

area0 = area5
area1 = area6
perim0 = perimeter5
perim1 = perimeter6

areas = np.concatenate((area0, area1))
perims = np.concatenate((perim0, perim1))

# we also need a way to track what the label for the feature is (i.e.
what class does it belong to)
labels = np.zeros(perims.shape)
#so we're dealing with 0 and 1 as the label
labels[0:len(area0)] = 0
labels[len(area0):] = 1

# So each point will have an area and a perimeter, we can make the
area the first column and the perimeter the second column
# We also will need labels for these points
arr = np.stack((areas, perims), axis=1)
arr

print(f'labels shape: {labels.shape}')
print(f'arr shape: {arr.shape}')

"""Now we can continue to calculate the fisher discriminate

#### Class means
"""

# first lets get m1, which will consist of the average of the mean
and perimeter of all elements in class 1
m1 = np.mean(arr[labels==0], axis=0)
print(f'Mean area for class label 0: {m1[0]}\nMean perimeter for
class label 0: {m1[1]}')
```

```
#let's get m2 the same way
m2 = np.mean(arr[labels==1], axis=0)
print(f'\nMean area for class label 1: {m2[0]}\nMean perimeter for
class label 1: {m2[1]}')

"""#### Scatter"""

display.Image('/content/Sw.png')

# Let's calculate first term 1, nEC1
term1 = np.dot((arr[labels==0] - m1), (arr[labels==0]-m1).T)

#Calc the second term
term2 = np.dot((arr[labels==1]-m2), (arr[labels==1]-m2).T)

term1 + term2

"""Let's drop some data from the other class and try this again
because then the data sets will be of equal sample size."""

sum(labels==1) - sum(labels==0)

"""We need to get rid of 819 points from the end of the array because
that is how many extra data points there are for class with label =
1.

So just take the entire np array from 0:len(arr)-819
"""

arr = arr[0:len(arr)-819]
print(f'Arr new shape: {arr.shape}')
labels = labels[0:len(labels)-819]
print(f'Labels new shape: {labels.shape}')

# Let's confirm that the number of classes are equal now
assert np.sum(labels==0) == np.sum(labels==1), 'Labels are not equal'

"""Good. Now we can continue with the process"""

# first lets get m1, which will consist of the average of the mean
and perimeter of all elements in class 1
m1 = np.mean(arr[labels==0], axis=0)
print(f'Mean area for class label 0: {m1[0]}\nMean perimeter for
class label 0: {m1[1]}')

#let's get m2 the same way
```

```
m2 = np.mean(arr[labels==1], axis=0)
print(f'\nMean area for class label 1: {m2[0]}\nMean perimeter for
class label 1: {m2[1]}')

# Let's calculate first term 1, nEC1
term1 = np.dot((arr[labels==0] - m1), (arr[labels==0]-m1).T)

#Calc the second term
term2 = np.dot((arr[labels==1]-m2), (arr[labels==1]-m2).T)

Sw = term1+ term2
print(f'Sw shape: {Sw.shape}')

display.Image('/content/Screenshot 2022-11-09 at 9.28.49 PM.png')

# Lastly, calculate the W
W = np.linalg.inv(Sw) * (m2 - m1)

"""Well unfortunately I ran into another error. Next steps would've
just been to project the dimension down and then come up with a
threshold and classify. Lastly, run it on the test set and get
y_preds and calculate the balanced classification accuracy from
before"""

display.Image('/content/Screenshot 2022-11-10 at 3.34.34 PM.png')

"""Let's visualize the features area and perimeter for 5s and 6s and
see if they'd be a good feature to separate 5s and 6s"""

(x_train, y_train), (x_test, y_test) = mnist.load_data()

number = 5

x5 = x_train[y_train==number,:,:]
x6 = x_train[y_train==number+1,:,:]
buf5 = "Number  %d" % number
buf6 = "Number  %d" % (number+1)
# Threshold images
t5 = 1*(x5 > 60)
t6 = 1*(x6 > 60)

# Region properties
area5 = np.zeros(t5.shape[0])
perimeter5 = np.zeros(t5.shape[0])
for i in range(0,t5.shape[0]):
  props = measure.regionprops(t5[i,:,:])
```

```
  area5[i] = props[0].area
  perimeter5[i] = props[0].perimeter

area6 = np.zeros(t6.shape[0])
perimeter6 = np.zeros(t6.shape[0])
for i in range(0,t6.shape[0]):
  props = measure.regionprops(t6[i,:,:])
  area6[i] = props[0].area
  perimeter6[i] = props[0].perimeter

plt.figure(figsize=(20,5))
plt.subplot(1,2,1)
plt.scatter(area5,perimeter5, label=buf5)
plt.scatter(area6,perimeter6, label=buf6)
plt.title('All images for both classes')
plt.legend()

plt.subplot(1,2,2)
plt.scatter(area5[0:100],perimeter5[0:100], label=buf5)
plt.scatter(area6[0:100],perimeter6[0:100], label=buf6)
plt.title('100 images of each class')
plt.legend()

"""There's a lot of over-lap so these features, area and perimeter
are not good to try and classify between the classes 5 and 6."""
```