ECE 566

Project 1 Report

By: Michael Schmidt

Due: 11/2/2025

Contents

Objective:

The goal of this project is to build and understand how machine learning models work from “scratch”. This implies that built-in functions that do back-propagation, training, classifications, etc, are not allowed. Everything must be coded by hand to understand the functionality of the models.

This project will review two different types of models to classify handwritten numbers, specifically handwritten 0’s from 1’s and handwritten 5’s from 6’s. The two models to study are one that uses the Fisher Discriminant, and another that is a single layer neural network.

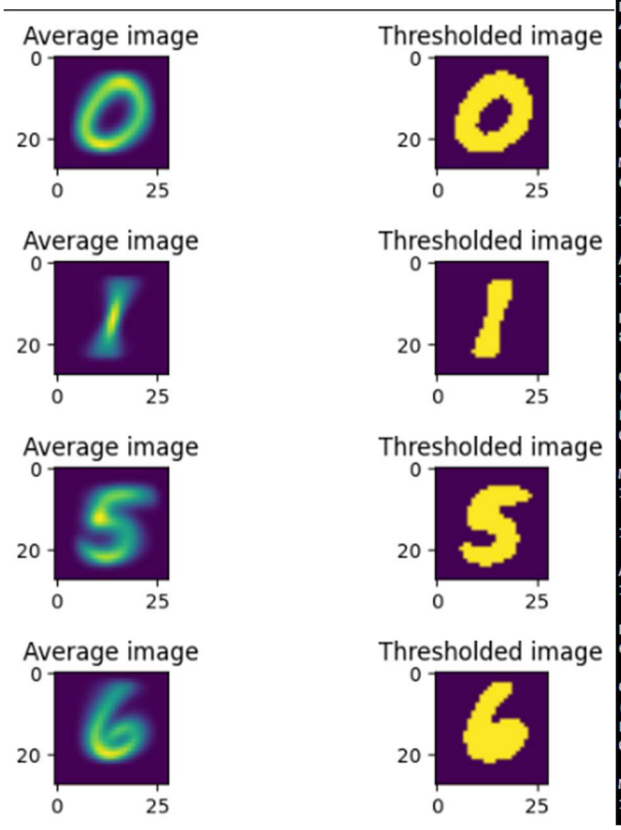
Dataset

For this project, handwritten numbers from 0 to 9 will be used, collected from the keras “mnist” dataset. The dataset contains 70,000 total samples of handwritten numbers, where 60,000 is used for training, and 10,000 is used for testing. The number “0” has 5923 samples, “1” has 6742 samples, “5” has 5421 samples, and “6” has 5918 samples. Each image is 28x28 pixel size, and each pixel value is a grayscale between 0 and 255. For testing, the class 0 has 980 samples, class 1 has 1135 samples, class 5 has 892 samples, and class 6 has 958 samples.

Part A

Feature extraction:

For building the model using the Fisher Discriminant, features need to be extracted from the images to process their data. It is possible to extract only pixel by pixel data, however using features extracted from skimage “measure” library can give more information.




```
1 region/s were found for number 0

Area (in pixels):
237.0

Perimeter (in pixels):
93.74011537017762

Centroid (pixel coordinates):
(np.float64(13.978902953586498), np.float64(14.080168776371307))
Eccentricity:
0.6181813464414365

Minor axis length:
18.53176476195907

1 region/s were found for number 1

Area (in pixels):
100.0

Perimeter (in pixels):
48.14213562373095

Centroid (pixel coordinates):
(np.float64(14.08), np.float64(13.95))
Eccentricity:
0.965964935004834

Minor axis length:
6.075181761985079

1 region/s were found for number 5

Area (in pixels):
196.0

Perimeter (in pixels):
82.18376618407356

Centroid (pixel coordinates):
(np.float64(13.994897959183673), np.float64(14.066326530612244))
Eccentricity:
0.7913404594738829

Minor axis length:
14.30801632317288

1 region/s were found for number 6

Area (in pixels):
192.0

Perimeter (in pixels):
65.94112549695427

Centroid (pixel coordinates):
(np.float64(13.833333333333334), np.float64(14.020833333333334))
Eccentricity:
0.6822322048449042

Minor axis length:
14.818627058217432
```

[Figures 1 & 2: figure 1 on the left is a comparison of the average pixel density for the numbers 0, 1, 5, 6. The yellow is when they are thresholded to have a density of over 60. The image on the right are the features extracted from the average pixel density.]

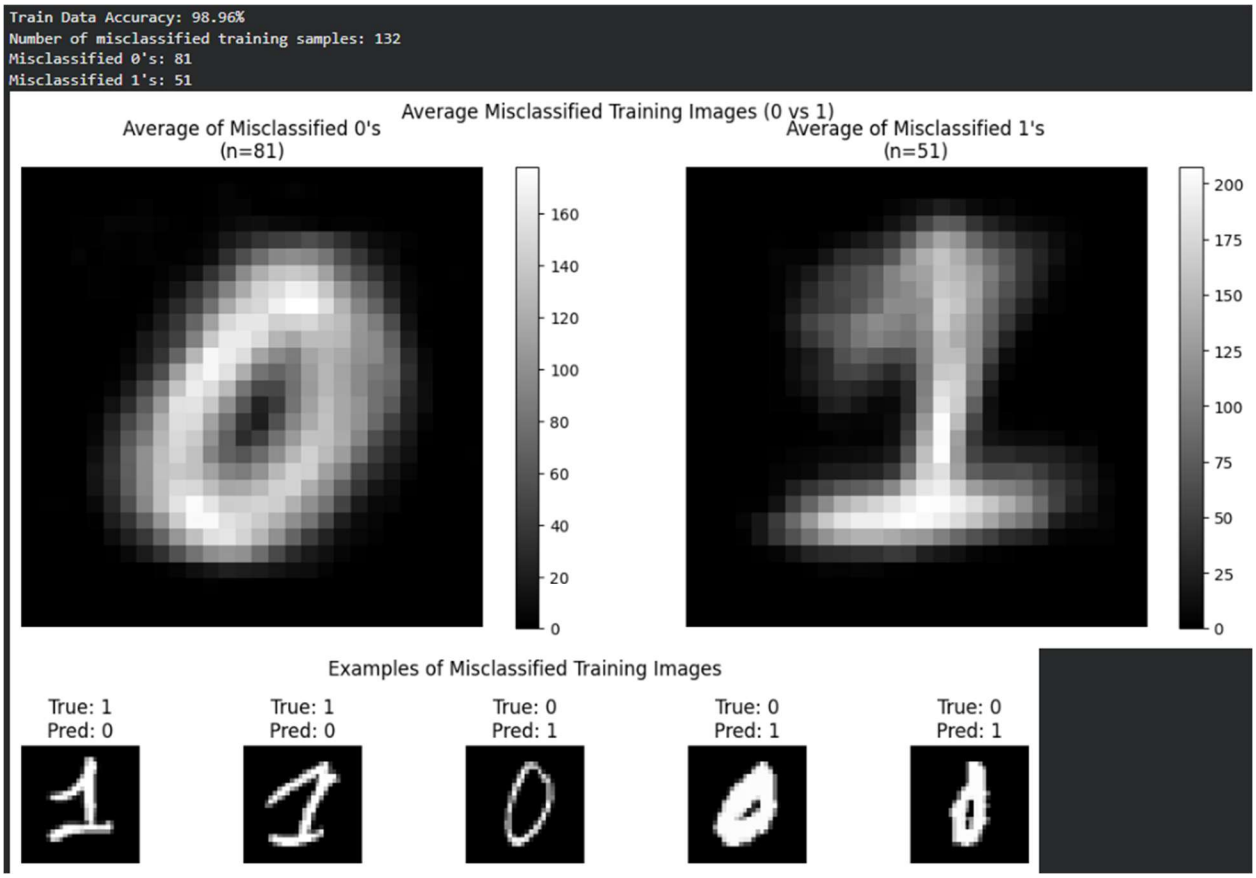
The two images above are examples of the average pixel density of the four numbers being used, and the extracted features. Comparing the extracted features numbers, it would appear the clearest features to use for the discriminant is area and perimeter. Looking at the values for the features of 5 & 6, centroid, eccentricity, and minor axis length seem to have values close to each other when comparing the two numbers, which indicates that the discriminant may have trouble finding the best threshold value to separate the two classes with low variance of each other.

Model performance on training data:

Building the fisher discriminant model works as follows:

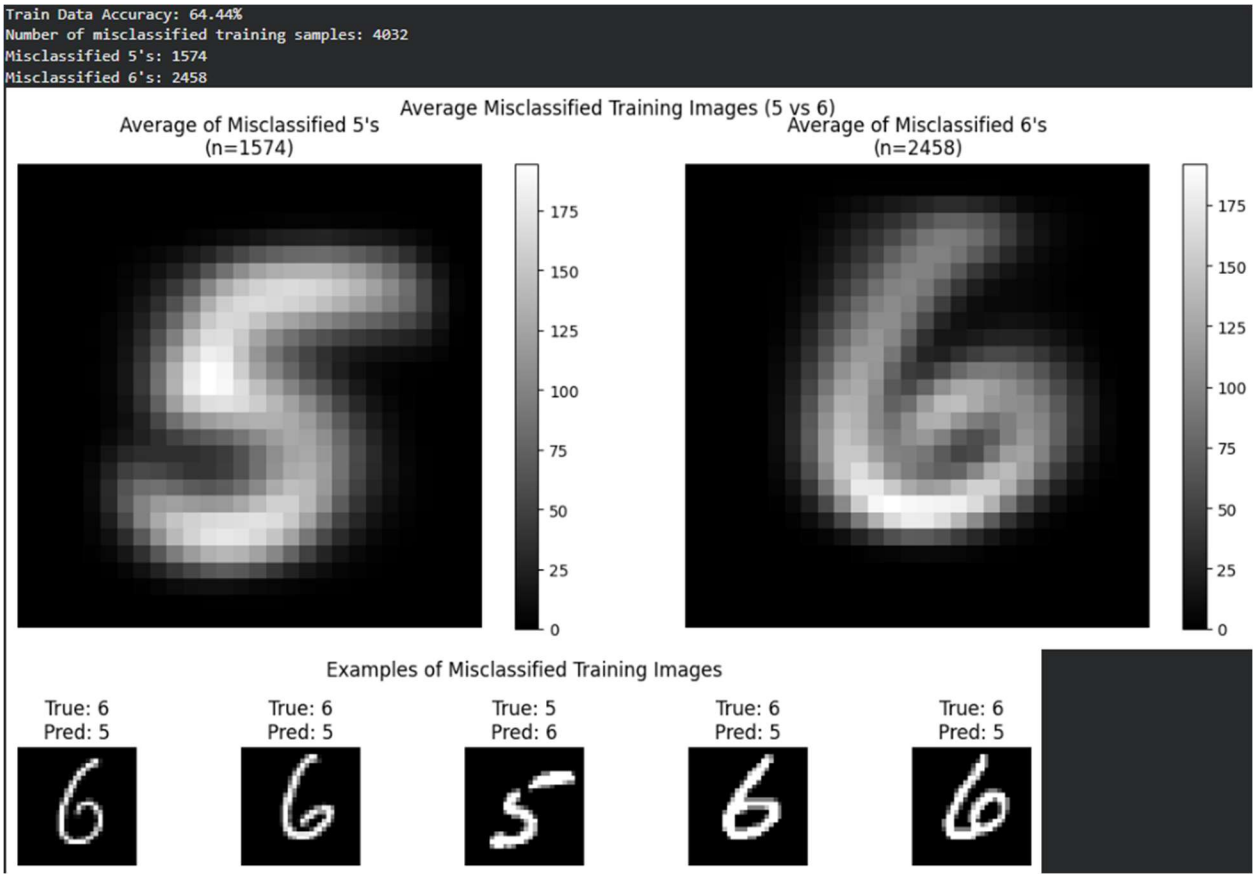
1. Collect only the two chosen numbers to train on from the dataset
2. Extract the area and perimeter features of each sample from both classes
3. Generate the mean of the two sample classes
4. Calculate the within-class scatter S_W
5. Calculate the discriminant vector W
6. Project the vector into the training & testing data.
7. Find the threshold value by averaging the mean of both projected data of the two classes.
8. Predict the two classes by seeing if the data is higher or lower than the threshold

Following the build of the model above, the final accuracy of the training data from comparing the numbers "0" and "1" was 98.96%. This can be considered a good score, as it means that the model is able to predict the training data well. It did misclassify some of the classes, hence why it was not 100%. This could be due to some of the variance of the two classes for area or perimeter was overlapping each other beyond the calculated threshold value.



[Figure 3: Examining misclassifications of 0's and 1's on training data]

Looking at figure 3, which is an average of the misclassifications of the two classes, the model struggles to classify them as they are slightly deformed. This is noticeable when looking at the examples of misclassifications. The number 0 typically has a higher area and perimeter than number 1, however with deformed samples, this confuses the model and predicts them as the other.

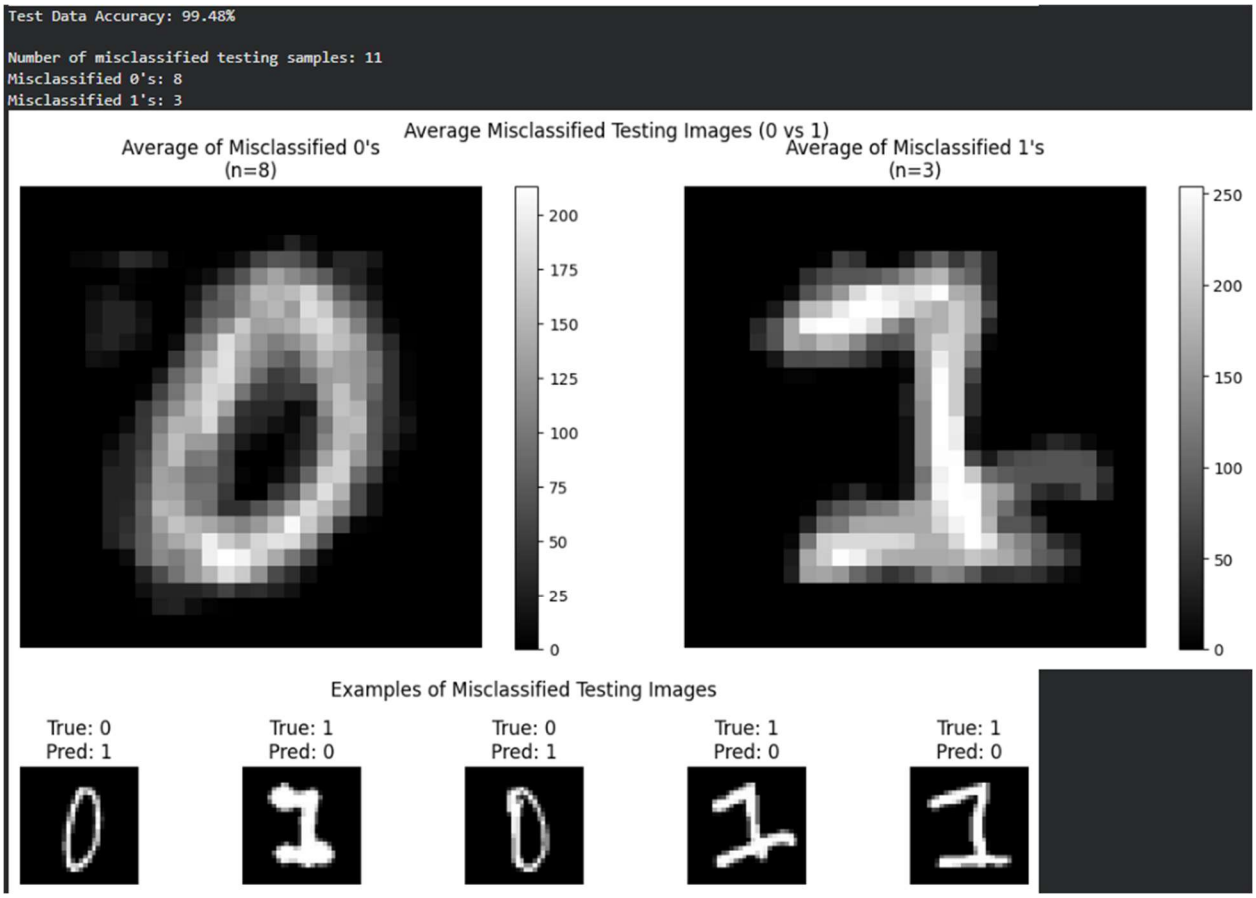


[Figure 4: Examining misclassifications of 5's and 6's on training data]

When using the same model to predict the numbers 5 and 6, it has achieved an accuracy of 64.44% on the training data. This is significantly lower than with the numbers 0 and 1, which is expected as the features of area and perimeter are much closer. The feature of area and perimeter has a much higher variance to each other than the numbers 0 and 1, which causes the threshold to be unclear on how to best separate the two classes.

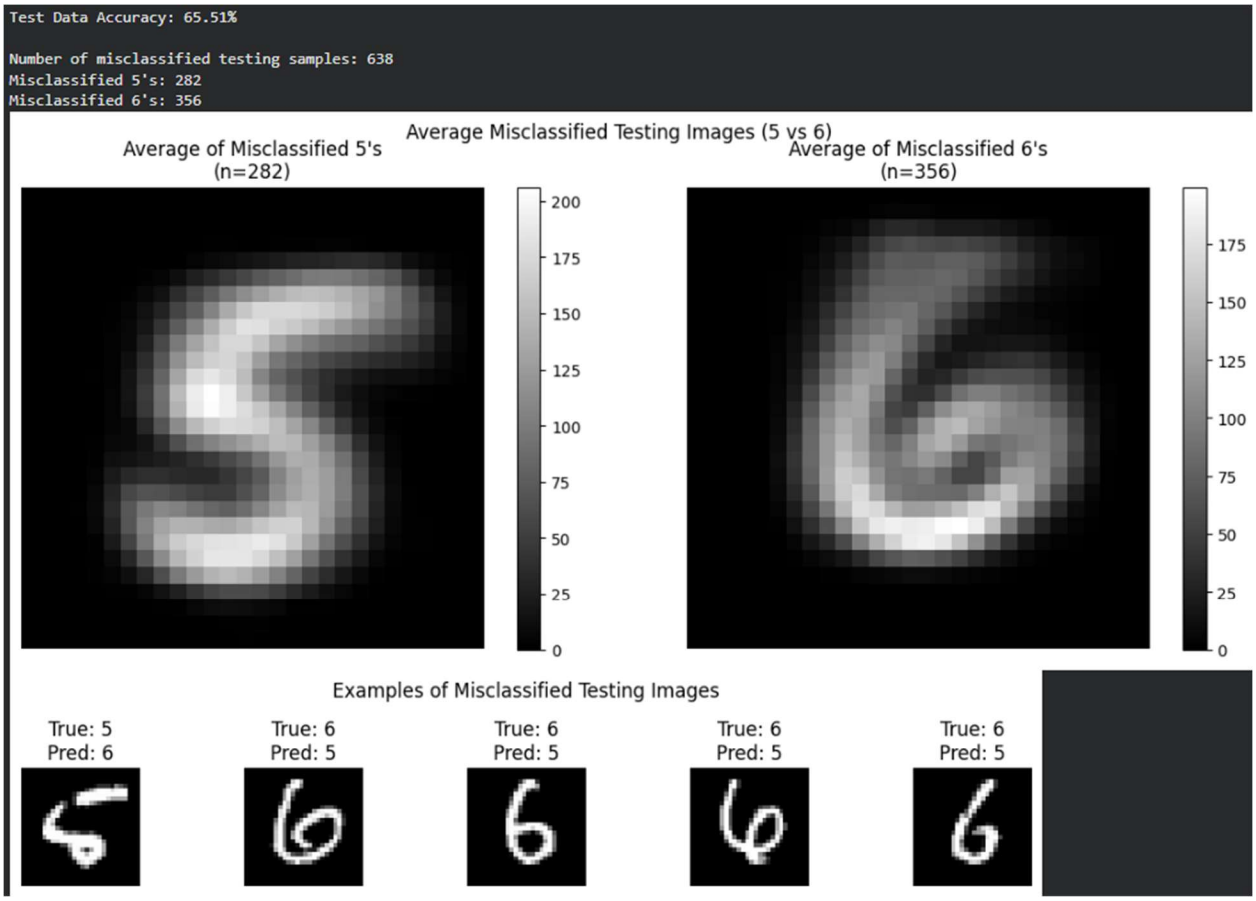
Model performance on testing data:

When testing the model on the number 0's and 1's testing dataset, it achieved an accuracy of 99.48%. The accuracy is higher; however, the testing dataset samples is much lower. The model still misclassifies some samples, and looking at figure 5 below, the samples are deformed, which causes the model to fail on them.



[Figure 5: Examination of misclassifications on 0's and 1's on testing data]

Testing the model on the testing data of numbers 5 and 6 gave an accuracy of 65.51%. This is slightly higher than what was on the training data, however the number of samples is much lower. As with the training data, the model fails to find a clean threshold that clearly separates the area and perimeter features of the two numbers, due to them having a high variance that overlaps each other.



[Figure 6: Examination of misclassifications on 5's and 6's on testing data]

Comments:

Overall, the fisher discriminant is a good model to get started with how basic machine learning works. Using the features of area and perimeter works well for classifying handwritten numbers 0 and 1 but struggles to classify numbers 5 and 6. This is mostly due to the features of 5 and 6 having a high variance that overlaps each other and causes the model to misclassify them. Using other features such as eccentricity or centroid could help the model perform better.

Part B:

Model Architecture:

The model to study is a single layer logistic regression neural network that will be used to classify handwritten 0's from 1's, and handwritten 5's from 6's. It works by having each pixel be sent to a neuron with a single weight and bias. It uses the sigmoid activation function and uses the binary cross-entropy loss.

$$z^{(i)} = w^T x^{(i)} + b$$

$$\hat{y}^{(i)} = sigmoid(z^{(i)})$$

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

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

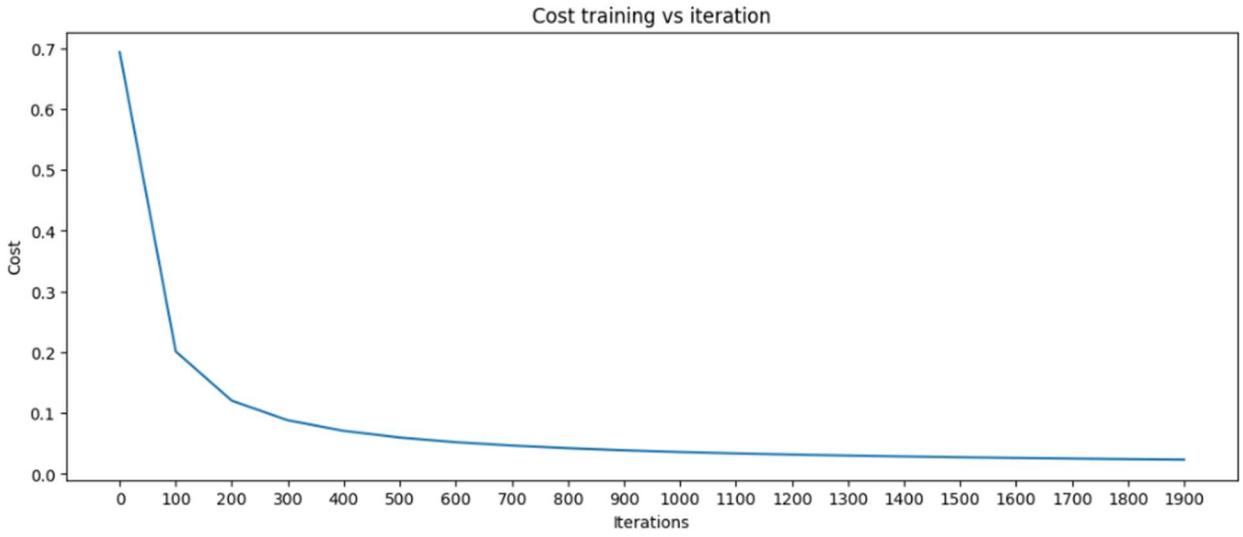
[Figure 7 & 8: Formulas for the neural network]

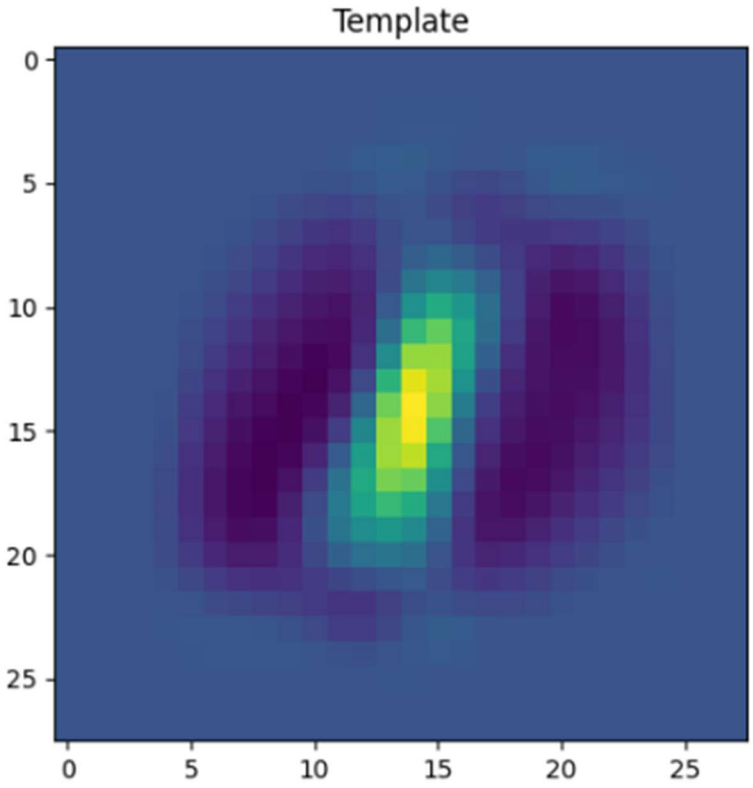
The network will work as follows:

1. Load the two classes to compare
2. Standardize the images by dividing by 255.
3. Initialize the weights by setting them all to 0 and the bias to 0
4. Start a gradient descent function that calculates the forward and back propagation and updates the weights for each iteration and learning rate.
5. Predict the model on training and testing data

Model performance on training data:

After setting up the model for training using classes 0 and 1, the number of iterations is set to 2000 with a learning rate of 0.005. The final accuracy the model achieved on the training dataset is 50.21%.





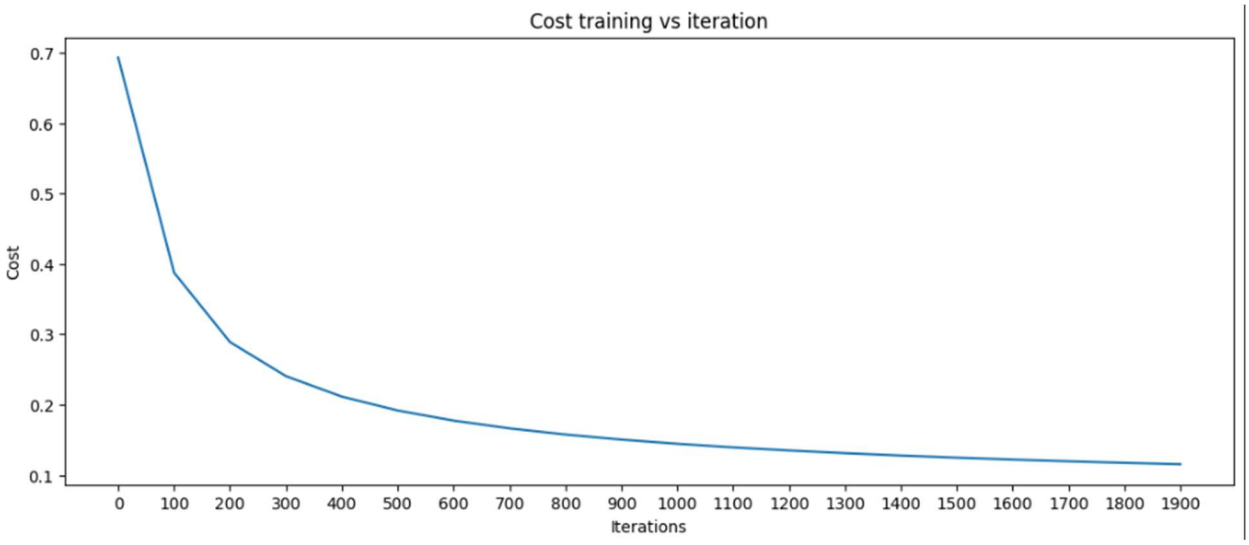
[Figure 9 & 10: Examining results of model weights for class 0 and 1 on training data]

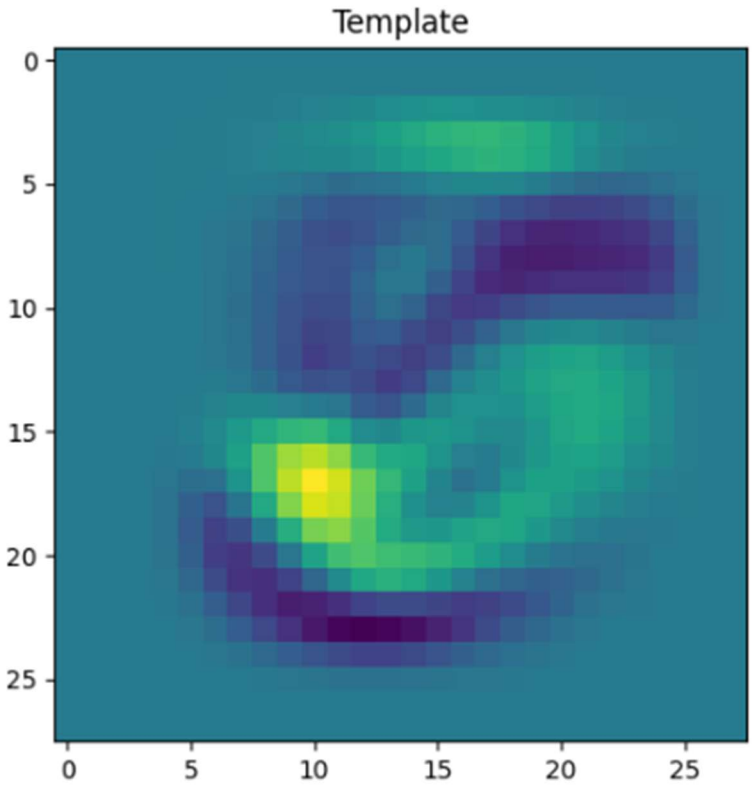
Looking at figure 9 and 10 above, the model seems to have learnt the basic patterns of class 0 and 1 after around 900 iterations. All the iterations after that seem to give a slower improvement as the model is finding it more difficult to classify the two classes. Image 10 above helps give a representation of what the model has learnt. All the bright greens and yellows represent how the model classifies class 1, and the purple represents how the model classifies class 0. The dark blue means the pixels don't contribute much to the classifications. It appears

that the model understands that number 1 generally has vertical pixels in the middle, and the number 0 has pixels around the center. It still has issues accurately classifying the two images, which could indicate that there are a lot of pixels that share similar values for 0 and 1 what makes the model have trouble learning. Using other features such as area or perimeter as used in the Fisher discriminant model might help improve the classification.

Using the same model training on the training data for handwritten 5 and 6 gave a final accuracy of 50.11% on the training data. Looking at figure 11 & 12 below, unlike with the handwritten 0's and 1's, the model training seems to stagnate around 1400 iterations. Any more iterations after that do not make many improvements. This indicates that the model has much more trouble trying to correctly classify the difference between numbers 5 and 6, and that there are many pixels that share the same values for the two numbers.

Looking at the weights image below, the model seems to have found a general pattern for the number 6 in green, and number 5 in purple. At the top left, you can see that the weights are a light mixture of green and purple, and there are other faint spots as well. This indicates that there are multiple pixels that share similar values in those areas, causing the model to have issues trying to correctly identify them.





[Figure 11 & 12: Examining results of model weights for class 5 and 6 on training data

Model performance on testing data:

After training the model in classes 0 and 1, the final accuracy it achieved on the testing data is 50.27%. This is slightly higher than what it achieved on the training data, but the improvement is negligible as it means that there are 1 or 2 samples that are a better fit for the model than what was in the training data.

After training the model in classes 5 and 6, the final accuracy it achieved on the testing data is 50.07%. This is slightly lower than what it achieved with the training data, and the same reasons why it failed there are valid for this case.

Comments:

Overall, the model achieved an accuracy of 50.27% on the testing data for classes 0 and 1 and achieved an accuracy of 50.07% on the testing data for the classes 5 and 6. Compared to using the Fisher discriminant, this model performed much lower for both datasets, however there are some benefits.

For the fisher model, the model was trained on extracted features calculated from each individual sample, which intuitively works better for the class of 0s and 1s as the area and perimeter are vastly different, whereas the neural network was trained on the values of each individual pixel of the samples. This makes it much harder to learn to classify the images as there are multiple pixels that share similar values for both classes. However, the benefit of training the neural network comes from the fact of learning by iterations. The fisher discriminate value is hardcoded so that the model must decide if the projected data is above or below the threshold. This makes it harder to make slight adjustments to the model as it is constricted by how it was coded. The neural network however is trained by iterative learning, meaning that the model is not hardcoded to decide what weights and bias works best for each pixel, but keeps working towards the optimum solution. It is possible to improve the score more by adding more iterations, or adjusting the learning rate, however this could make the model overfit the data, where it learns the best weight for only the training data, but does not generalize well. In addition, the neural network does not have to be constricted to learning only the pixel values of the images, it can also learn from the same feature extractions as well. By adding another neuron just for each of the features extracted from the image, the classifier can learn to better decide what the class is.

Conclusion:

Two models have been made to classify handwritten 0's from 1's and handwritten 5's from 6's. They were built from scratch and examined to see their performance. The Fisher discriminant model appears to perform better than the neural network, however this is an unfair comparison as their "learning technique" are different. The Fisher model is created to classify the images based on their extracted features, whereas the neural network is trained to learn the weights and bias of each individual pixel. The neural network fails to compete with the fisher model as there are multiple pixels that share the same values within the two classes it has been trained on, whereas the fisher model excels as the average features extracted from the two classes are different enough to make a good comparison. By using the same features in the fisher model within the neural network by adding another neuron specifically for each feature, the neural network could challenge or exceed the performance of the fisher model.

Appendix A: Fisher Discriminant Code

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

import warnings
warnings.filterwarnings("ignore") # Added this at the end to show a clean output
with no warnings but not necessary

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

# Function to extract features (area and perimeter)
def extract_features(images):
    features = []
    for img in images:
        # threshold the image
        thres_img = 1*(img > 0)

        # Get region properties
        props = measure.regionprops(measure.label(thres_img))

        if len(props) > 0:
            # Use the first (largest) region
            area = props[0].area
            perimeter = props[0].perimeter
            features.append([area, perimeter])
        else:
            # If no region found, use zeros
            features.append([0, 0])

    return np.array(features)

def plot_misclassified(x_train_new, y_train_new, y_pred_train, first_num,
sec_num, test=False):
    """
    Plot average misclassified images by class and show individual examples.

    Arguments:
    x_train_new -- training images (original 28x28 images)
    y_train_new -- true labels
    y_pred_train -- predicted labels
    first_num -- first class number
    sec_num -- second class number
```

```
    """
    # Find misclassified samples
    misclassified_mask = (y_pred_train != y_train_new)
    misclassified_images = x_train_new[misclassified_mask]
    misclassified_true_labels = y_train_new[misclassified_mask]
    misclassified_pred_labels = y_pred_train[misclassified_mask]

    if test:
        print(f"Number of misclassified testing samples:
{np.sum(misclassified_mask)}")
    else:
      print(f"Number of misclassified training samples:
{np.sum(misclassified_mask)}")

    # Separate misclassified samples by true class
    if np.sum(misclassified_mask) > 0:
        misclassified_class0 = misclassified_images[misclassified_true_labels ==
first_num]
        misclassified_class1 = misclassified_images[misclassified_true_labels ==
sec_num]

        print(f"Misclassified {first_num}'s: {len(misclassified_class0)}")
        print(f"Misclassified {sec_num}'s: {len(misclassified_class1)}")

        # Plot average of misclassified images for each class
        fig, axes = plt.subplots(1, 2, figsize=(12, 5))

        # Average of misclassified class 0
        if len(misclassified_class0) > 0:
            avg_misclassified_class0 = np.mean(misclassified_class0, axis=0)
            im0 = axes[0].imshow(avg_misclassified_class0, cmap='gray')
            axes[0].set_title(f'Average of Misclassified
{first_num}\'s\n(n={len(misclassified_class0)})')
            axes[0].axis('off')
            plt.colorbar(im0, ax=axes[0])
        else:
            axes[0].text(0.5, 0.5, f'No misclassified {first_num}\'s',
                         ha='center', va='center', transform=axes[0].transAxes)
            axes[0].axis('off')

        # Average of misclassified class 1
        if len(misclassified_class1) > 0:
            avg_misclassified_class1 = np.mean(misclassified_class1, axis=0)
            im1 = axes[1].imshow(avg_misclassified_class1, cmap='gray')
```

```
            axes[1].set_title(f'Average of Misclassified
{sec_num}\'s\n(n={len(misclassified_class1)})')
            axes[1].axis('off')
            plt.colorbar(im1, ax=axes[1])
        else:
            axes[1].text(0.5, 0.5, f'No misclassified {sec_num}\'s',
                         ha='center', va='center', transform=axes[1].transAxes)
            axes[1].axis('off')

        if test:
            plt.suptitle(f'Average Misclassified Testing Images ({first_num} vs
{sec_num})')
        else:
            plt.suptitle(f'Average Misclassified Training Images ({first_num} vs
{sec_num})')

        plt.tight_layout()
        plt.show()

        # Show individual examples of misclassified images
        num_examples = min(5, np.sum(misclassified_mask))
        if num_examples > 0:
            fig, axes = plt.subplots(1, num_examples, figsize=(num_examples*2,
2))
            if num_examples == 1:
                axes = [axes]

            for i in range(num_examples):
                axes[i].imshow(misclassified_images[i], cmap='gray')
                axes[i].set_title(f'True: {misclassified_true_labels[i]}\nPred:
{misclassified_pred_labels[i]}')
                axes[i].axis('off')

            if test:
                plt.suptitle('Examples of Misclassified Testing Images')
            else:
                plt.suptitle('Examples of Misclassified Training Images')

            plt.tight_layout()
            plt.show()
    else:
        print("No misclassified samples to plot!")

    return
```

```
def FisherDis(first_num, sec_num):
    #Data
    data1 = x_train[y_train==first_num]
    data2 = x_train[y_train==sec_num]

    numberSamplesTest1 = x_test[y_test==first_num]
    numberSamplesTest2 = x_test[y_test==sec_num]

    print(f"Number of test images for class {first_num} =
{numberSamplesTest1.shape}")
    print(f"Number of test images for class {sec_num} =
{numberSamplesTest2.shape}")

    #Feature extreaction: ONLY CONTAINS (Area, Perimeter)
    features1 = extract_features(data1)
    features2 = extract_features(data2)

    print(f"Number of {first_num}'s: {features1.shape[0]}")
    print(f"Number of {sec_num}'s: {features2.shape[0]}")
    print(f"Feature shape for {first_num}: {features1.shape}")
    print(f"Feature shape for {sec_num}: {features2.shape}")

    #Mean
    m1 = np.mean(features1, axis=0)
    m2 = np.mean(features2, axis=0)

    print(f"Mean features for {first_num}: Area={m1[0]:.2f},
Perimeter={m1[1]:.2f}")
    print(f"Mean features for {sec_num}: Area={m2[0]:.2f},
Perimeter={m2[1]:.2f}")

    #Scatter
    scatter1 = np.cov(features1, rowvar=False)
    scatter2 = np.cov(features2, rowvar=False)

    #Within-class scatter
    sw = scatter1 + scatter2

    # Discriminant vector
    w = np.linalg.pinv(sw) @ (m2 - m1)
    print(f"Discriminant vector w: {w}")

    #Project w onto training data
    x_train_new = x_train[(y_train==first_num) | (y_train==sec_num)]
    x_train_new_features = extract_features(x_train_new)
```

```
    x_train_new_projections = x_train_new_features @ w

    # Project w onto test data
    x_test_new = x_test[(y_test==first_num) | (y_test==sec_num)]
    x_test_new_features = extract_features(x_test_new)
    x_test_new_projections = x_test_new_features @ w

    # Threshold
    y_train_new = y_train[(y_train == first_num) | (y_train == sec_num)]
    new_m1 = np.mean(x_train_new_projections[y_train_new == first_num])
    new_m2 = np.mean(x_train_new_projections[y_train_new == sec_num])
    thres = (new_m1 + new_m2) / 2
    print(f"Threshold: {thres:.4f}")

    # Predict Training
    y_pred_train = (x_train_new_projections > thres).astype(int)
    # Map predictions: 0 -> first_num, 1 -> sec_num
    y_pred_train = np.where(y_pred_train == 0, first_num, sec_num)
    accuracy_train = np.mean(y_pred_train == y_train_new)
    print(f"Train Data Accuracy: {accuracy_train*100:.2f}%")

    #-------------------------------------------------------------
    #Plotting misclassifications
    plot_misclassified(x_train_new, y_train_new, y_pred_train, first_num,
sec_num, False)

    # Predict Test
    y_test_new = y_test[(y_test == first_num) | (y_test == sec_num)]
    y_pred_test = (x_test_new_projections > thres).astype(int)
    # Map predictions: 0 -> first_num, 1 -> sec_num
    y_pred_test = np.where(y_pred_test == 0, first_num, sec_num)
    accuracy_test = np.mean(y_pred_test == y_test_new)
    print(f"Test Data Accuracy: {accuracy_test*100:.2f}%")
    print()

    #-------------------------------------------------------------
    #Plotting misclassifications
    plot_misclassified(x_test_new, y_test_new, y_pred_test, first_num, sec_num,
True)

    return



#-------------------------------------------------------------
```

```
#for numbers 0 & 1
FisherDis(0,1)
#-------------------------------------------------------------------
#for numbers 5 & 6
FisherDis(5,6)
```

Appendix B: Logistic Regression Neural Network Code

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

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

# Test the sigmoid function
print("Testing sigmoid function:")
print(f"sigmoid(0) = {sigmoid(0)}")  # Should be 0.5
print(f"sigmoid(2) = {sigmoid(2)}")  # Should be ~0.88
print(f"sigmoid(-2) = {sigmoid(-2)}")  # Should be ~0.12
print(f"sigmoid(array) = {sigmoid(np.array([1, 2, 3]))}")  # Should work with
arrays
print()

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((dim, 1))
    b = 0.0
```

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

    return w, b

# Test the initialize_weights function
print("Testing initialize_weights function:")
w, b = initialize_weights(784)
print(f"w shape: {w.shape}")  # Should be (784, 1)
print(f"b value: {b}")  # Should be 0 or 0.0
print(f"First 5 weights: {w[:5].flatten()}")  # Should be all zeros
print()

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)-----------------------------------

    # Compute activation: sigmoid(w^T * X + b)
    Z = X @ w + b  #Note: X is (m, 784), w is (784, 1), so we compute X @ w which
gives (m, 1)
    Yhat = sigmoid(Z)  # (m, 1) - predictions

    # Compute cost
    # J = 1/m * sum(-Y * log(Yhat) - (1-Y) * log(1-Yhat))
    cost = 1/m * np.sum(-Y * np.log(Yhat) - (1 - Y) * np.log(1 - Yhat))
```

```
    # BACKWARD PROPAGATION (TO FIND GRAD)--------------------------------

    # dw = 1/m * X^T * (A - Y)
    # db = 1/m * sum(A - Y)
    dw = 1/m * (X.T @ (Yhat - Y))  # (784, 1)
    db = 1/m * np.sum(Yhat - Y)  # scalar

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

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

    return grads, cost

# Test the propagate function
print("Testing propagate function:")
# Create small test data
X_test = np.random.randn(5, 784)  # 5 examples, 784 features
Y_test = np.array([[0, 1, 0, 1, 1]]).T  # 5 labels as column vector
w_test, b_test = initialize_weights(784)

grads, cost = propagate(w_test, b_test, X_test, Y_test)
print(f"Cost: {cost}")
print(f"dw shape: {grads['dw'].shape}")
print(f"db value: {grads['db']}")
print()

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 := w - learning_rate * dw
        # b := b - learning_rate * db
        w = w - learning_rate * dw
        b = 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
```

```
# Test the gradient_descent function
print("Testing gradient_descent function:")
X_test = np.random.randn(10, 784)  # 10 examples
Y_test = np.random.randint(0, 2, (10, 1))  # Random labels
w_test, b_test = initialize_weights(784)

params, grads, costs = gradient_descent(w_test, b_test, X_test, Y_test,
num_iterations=500, learning_rate=0.01)
print(f"Final cost: {costs[-1]}")
print(f"Number of costs recorded: {len(costs)}")
print()

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)


    Y_prediction = (A > 0.5).astype(int)


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

```
    return Y_prediction

# Test the predict function
print("Testing predict function:")
X_test = np.random.randn(5, 784)
w_test = np.random.randn(784, 1) * 0.01
b_test = 0.0

predictions = predict(w_test, b_test, X_test)
print(f"Predictions shape: {predictions.shape}")
print(f"Predictions: {predictions.flatten()}")
print(f"All predictions are 0 or 1: {np.all((predictions == 0) | (predictions ==
1))}")
print()
print("="*70)
print("All helper functions tested successfully!")
print("="*70)
print()

# 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)
y_train = y_train.reshape(-1, 1)  # Make it (n, 1) instead of (n,)
x_test = x_test[np.isin(y_test,[class0,class1]),:,:]
y_test = 1*(y_test[np.isin(y_test,[class0,class1])]>class0)
y_test = y_test.reshape(-1, 1) # Make it (n, 1) instead of (n,)

# 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')
```