CS 559 Homework 1

Stephen Szemis

October 1, 2020

I pledge my honor that I have abided by the Stevens honor system.

Problem 1: Perceptron

Part 1: Yes. The NAND function can be linearly separable.

Part 2: Let's run perception. We have to account for bias (not mentioned in slides) in order to actually converge here. All Points now will have an extra dimension appended, this is just a 1 to help clean up the dot product. Example: $P_1 = (0, 1, 1)$ and $P_1 = (1, 1, 1)$. Our initial weights are then $w = (1, 1, -\frac{1}{2})$ to account for our initial boundary.

1. $w_1^T = (1, 1, -\frac{1}{2})\, P_1 = (0, 1, 1)$ since $(w_1 * P_1) > 0$ that means P_1 is classified as positive, which is wrong. So update $w_2 = w_1 - P_1 = (1, 0, -\frac{3}{2})$
2. $w_2^T = (1, 0, -\frac{3}{2})\, P_2 = (1, 1, 1)$ since $(w_2 * P_2) < 0$ that means P_2 is classified as negative, which is wrong. So update $w_3 = w_2 + P_2 = (2, 1, -\frac{1}{2})$
3. $w_3^T = (2, 1, -\frac{1}{2})\, P_3 = (1, 0, 1)$ since $(w_3 * P_3) > 0$ that means P_3 is classified as positive, which is wrong. So update $w_4 = w_3 - P_3 = (1, 1, -\frac{3}{2})$
4. $w_4^T = (1, 1, -\frac{3}{2})\, P_4 = (0, 0, 1)$ since $(w_4 * P_4) < 0$ that means P_4 is classified as negative, which is correct. No update.
5. $w_5^T = (1, 1, -\frac{3}{2})\, P_1 = (0, 1, 1)$ since $(w_5 * P_1) < 0$ that means P_1 is classified as negative, which is correct. No update.

6. $w_6^T = (1, 1, -\frac{3}{2})\, P_2 = (1, 1, 1)$ since $(w_6 * P_2) > 0$ that means P_2 is classified as positive, which is correct. No update.

7. $w_7^T = (1, 1, -\frac{3}{2})\, P_3 = (1, 0, 1)$ since $(w_7 * P_3) < 0$ that means P_3 is classified as negative, which is correct. No update.

We've successfully classified all four points, so our weights and bias is correct. Our final decision boundary is $x_1 + x_2 - \frac{3}{2} = 0$.

Problem 2: PCA and Eigenfaces

Part 1: The code and eigenfaces

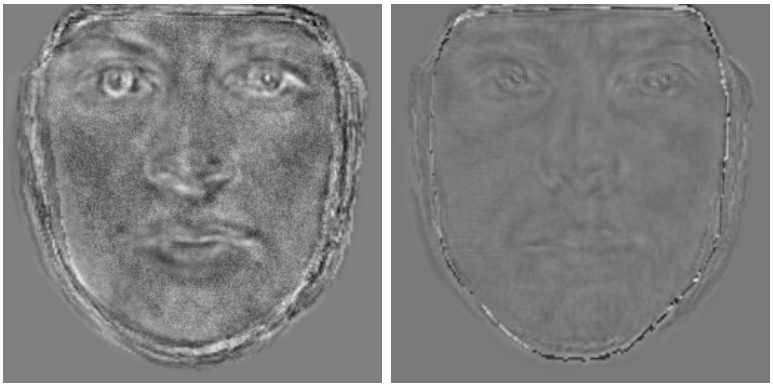
```
# Author: Stephen Szemis
# Date: 11/30/2020

#Imports and includes
import glob
import numpy as np
import matplotlib.pyplot as plt
from numpy import linalg as LA
from sklearn.model_selection import train_test_split
from PIL import Image

N = 177
D = 256

test_size = 20
train_size = N - test_size

# Helper for showing face in a window
def show_face(face):
    temp_face = face.reshape((256, 256))
    imgplot = plt.imshow(temp_face, cmap='gray')
    plt.show()

# Helper for saving save to a path
def save_face(face, path):
```

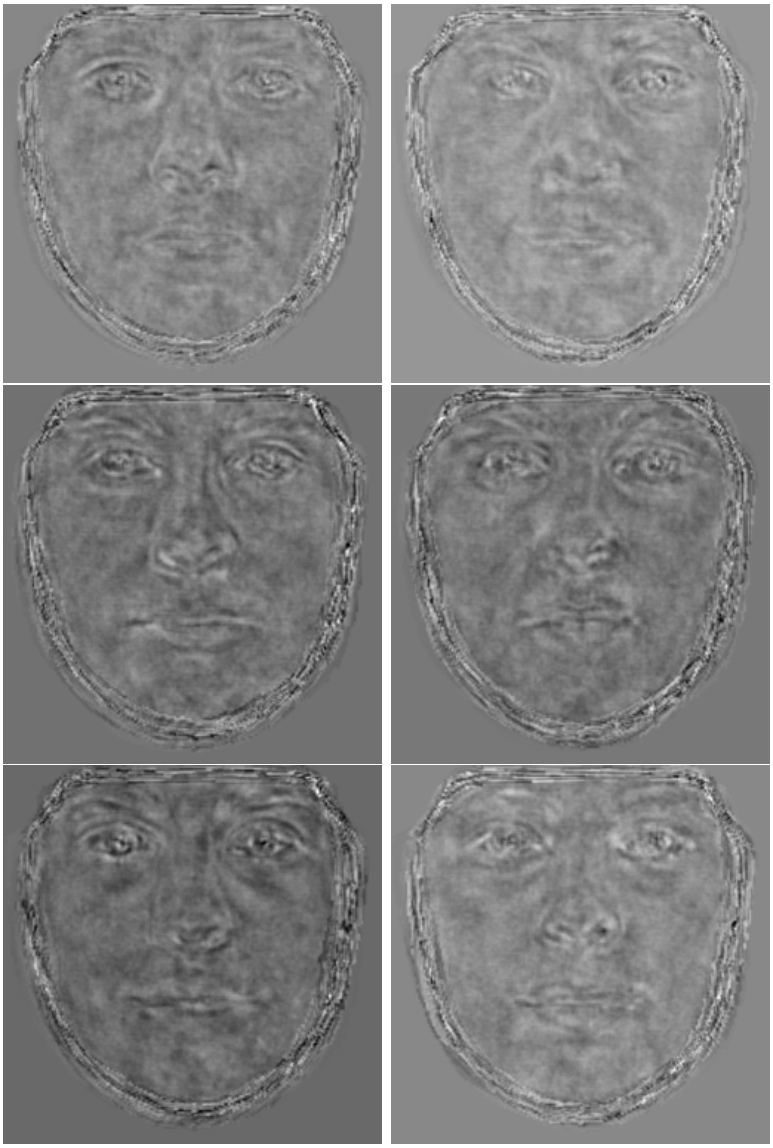
```
    result = face.reshape((256, 256))
    # result = Image.fromarray((temp_face * 255).astype(np.uint8))
    plt.imsave(path, result, cmap='gray')

# A simple helper for sorting our eignvectors before returning
def get_eigen(S):
    eigenValues, eigenVectors = LA.eig(S)
    idx = np.argsort(eigenValues)#[::-1]
    eigenValues = eigenValues[idx]
    eigenVectors = eigenVectors[:,idx]
    return (eigenValues, eigenVectors)

# Normalize vectors inside a matrix
def normalize(M):
    for index, m in enumerate(M):
        M[index] = m / LA.norm(m)
    return M

# Array size is hard coded for number of image files, not great
# practice, but probably good enough for this homework
faces = np.zeros((N, D * D))

# Read in faces. Note that we flatten images into 1-D arrays
for index, filepath in enumerate(glob.iglob('./face_data/*.bmp')):
    faces[index] = np.ravel(np.array(Image.open(filepath), dtype='float'))

# Create test / train sets (set random state for reproducable results)
train, test = train_test_split(faces, test_size=test_size, random_state=1)

# Center images
mean_face = np.sum(train, axis=0) / train_size
centered = train - mean_face
Xt = np.transpose(centered)

# Create S
S = (np.dot(centered, Xt) / train_size)

# Calculate eigenvalues and eigenvectors
```

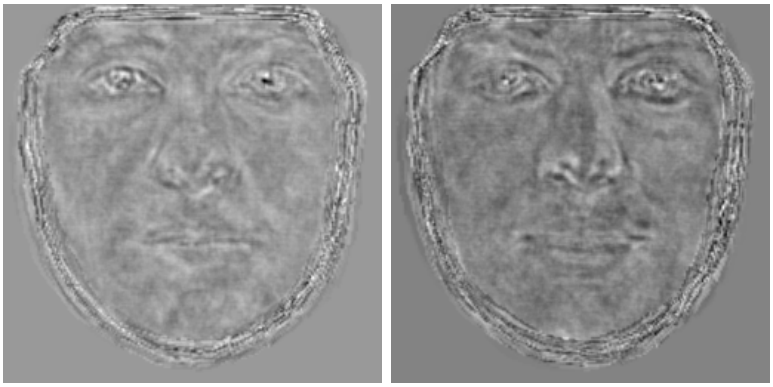
```
eigVals, eigVecs = get_eigen(S)

# Create and normalize eigenfaces
eigenfaces = np.zeros((N, D * D))
eigenfaces = np.transpose(np.dot(Xt, eigVecs[:,:N]))
eigenfaces = normalize(eigenfaces)

# Create our principal components set
K = 30
Wt = eigenfaces[:K]
W = np.transpose(Wt)

# Save images for part 1
for index in range(K):
    save_face(eigenfaces[index], "output_part1/eigenface_" + str(index) + ".png")

# Reconstruct images for part 2
def reconstruct(face, W, Wt):
    x = mean_face + np.dot(np.dot((face - mean_face), W), Wt)
    return x

refaced = np.zeros((test_size, D * D))
for index, face in enumerate(test):
    refaced[index] = reconstruct(face, W, Wt)

for index in range(5):
    save_face(refaced[index], "output_part2/reconstruct_" + str(index) + ".png")
    save_face(test[index], "output_part2/original_" + str(index) + ".png")

error_calc = lambda x: np.sum(np.abs((x - test))) / (test_size * D * D)

# Calculate error for part 2
part2_error = error_calc(refaced)

print("Error for part 2 is " + str(part2_error))

# Create loop for part 3
step = 10
```

```
error = np.zeros((train_size // step))
k_values = range(0, train_size - step, step)
for index, k in enumerate(k_values):
    wt = eigenfaces[:k]
    w = np.transpose(wt)
    const = np.zeros((test_size, D * D))
    for i, face in enumerate(test):
        const[i] = reconstruct(face, w, wt)
    error[index] = error_calc(const)

# Graph data for part 3
fig = plt.figure()
ax = fig.add_subplot(111)
ax.scatter([i for i in k_values], [j for j in error], s=25, c='b', marker='s')

# Produce Graph
ax.set_title("Plot hw2")
ax.set_xlabel('K Values')
ax.set_ylabel('Error Rate')
plt.show()
```

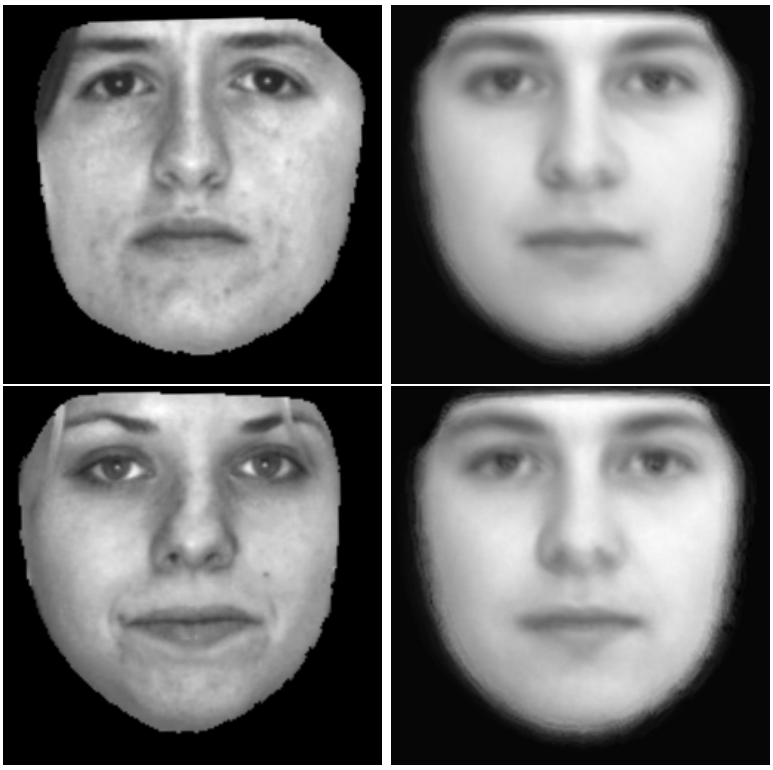
And here are the top ten eigenfaces.





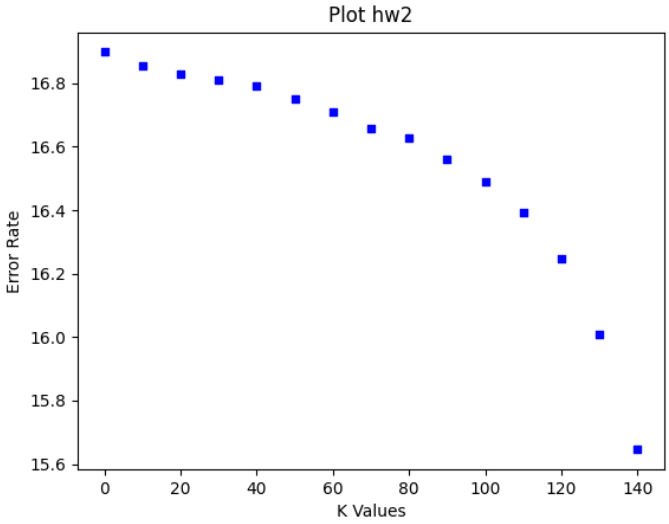


Part 2: Reconstruction The necessary code is commented above. Here are the reconstructed images side by side with the originals.



The reconstruction error was 16.81. I should note that we calculate the error per pixel, not just per image. So we end up dividing the normal error calculation by 256 * 256.

Part 3: Plotting K As per usual, the comments will tell you which piece of code does what. Here is our graph of k increasing by 10, up to our max k value of 140.



It's clear to see that we get an exponential drop off in error as we increase k. This makes sense since as k approaches our original dimension D, we end up simply containing all the information in our principal components. Of course, even at a large k, we still have a massive decrease in dimensionality. And since our error is so low, it's easy to see that PCA and eigenfaces are quiet effective at describing the necessary parts of the images.