CS540 Spring 2025 Homework 3

Due: 13 February 2025, 11:59 PM

1 Assignment Goals

- Explore Principal Component Analysis (PCA) and the related Python packages (`numpy`, `scipy`, and `matplotlib`)
- Make pretty pictures :)

2 Summary

In this project, you'll be implementing a facial analysis program using PCA, using the skills from **the linear algebra + PCA lecture**. You'll also continue to build your Python skills. We'll walk you through the process step-by-step (at a high level).

3 Packages Needed for this Project

You are only allowed to use Python3 standard library as well as NumPy, SciPy, and matplotlib (installation instructions linked). You should use a SciPy version `>= 1.5.0` and the following import commands:

```
>>> from scipy.linalg import eigh
>>> import numpy as np
>>> import matplotlib.pyplot as plt
```

4 Dataset

You will be using part of the CelebFaces Attributes (CelebA) dataset [2]. The original full-resolution dataset contains $202,599$ sample images of $10,177$ celebrities, each of size 218×178. This dataset also contains 40 attribute annotations for computer vision tasks, which we will not use in this HW. To avoid extremely large computations on our CPU, we will use a small subset of CelebA and compress our images from 218×178 to 60×50 . We use n to refer to the number of images ($n = 100$) and d to refer to the number of features (in this case, pixels) for each sample image ($d = 60 \times 50 = 3000$). Note, we'll use x_i to refer to the ith sample image which is a d-dimensional feature vector.

Note: The setup here is different from the example in the PCA lecture. In the lecture, we have a *single* large image divided into multiple patches, treating each patch as a data point x_i. Here we have multiple ($n = 100$) small images, each of which is treated as a data point x_i.

The dataset is saved in the `celeba_60x50.npy` file, which you will find in the hw3.zip file on Canvas. We also provide a full-resolution version of this dataset in `celeba_218x78x3.npy`, which you will use for visualization. The `.npy` file format is used to store numpy arrays. We will test your code only using this provided dataset.

5 Program Specification

Implement these **seven** Python functions in `hw3.py` to perform PCA on the dataset:

1. `load_and_center_dataset(filename)`: load the dataset from the provided `.npy` file, center it around the origin, and **return** it as a `numpy` array of floats.
2. `get_covariance(dataset)`: calculate and **return** the covariance matrix of the dataset as a `numpy` matrix ($d \times d$ array).
3. `get_eig(S, k)`: perform eigendecomposition on the covariance matrix S and **return** a diagonal matrix (`numpy` array) with the largest k eigenvalues on the diagonal *in descending order*, and a matrix (`numpy` array) with the corresponding eigenvectors as columns.
4. `get_eig_prop(S, prop)`: similar to `get_eig`, but instead of returning the first k, **return** all eigenvalues and corresponding eigenvectors in a similar format that explain more than a `prop` proproportion of the variance (specifically, please make sure the eigenvalues are returned in descending order).
5. `project_and_reconstruct_image(image, U)`: project each $d \times 1$ image into your m-dimensional subspace (spanned by m vectors of size $d \times 1$) and **return** the new representation as a $d \times 1$ `numpy` array.
6. `display_image(im_orig_fullres, im_orig, im_reconstructed)`: use `matplotlib` to display a visual representation of the original (full-res) image, original (low-res + grayscale) image and the reconstructed (low-res + grayscale) image side-by-side.
7. `perturb_image(image, U, sigma)`: use Gaussian distribution with standard deviation σ to perturb the given image. **return** the reconstructed image as a $d \times 1$ `numpy` array.

5.1 Load and Center the Dataset ([20] points)

You'll want to use the the `numpy` function `load()` to load `npy` dataset files in Python.

```
>>> x = np.load(filename)
```

This should give you an $n \times d$ dataset, represented as an n-by-d matrix/array `x` (recall that $n = 100$ is the number of images in the dataset and $d = 3000$ is the number of dimensions of each image). In other words, each row of `x` represents an image feature vector $x_i^\top$ (as a row vector).

Your next step is to center this dataset around the origin. Recall the purpose of this step from lecture — it is a technical condition that makes it easier to perform PCA, but it does not lose any important information.

To center the dataset is simply to subtract the mean μ_x from each data point x_i (image, in our case), i.e., $x_i^{\text{cent}} = x_i - u_x$, where

$$\mu_x = \frac{1}{n} \sum_{i=1}^{n} x_i.$$

You can take advantage of the fact that `x` (as defined above) is a `numpy` array and, as such, has this convenient behavior:

```
>>> x = np.array([[1,2,5],[3,4,7]])
>>> np.mean(x, axis=0)
array([2., 3., 6.])
>>> x - np.mean(x, axis=0)
array([[-1., -1., -1.],
       [ 1.,  1.,  1.]])
```

Observe that the shape of the array after centering is the same as the original array. After you've implemented this function, it should work like this:

```
>>> x = load_and_center_dataset('celeba_60x50.npy')
>>> len(x)
100
>>> len(x[0])
3000
>>> np.average(x)
4.608106488982836e-16
```

(Its center isn't *exactly* zero, but taking into account precision errors over 100 arrays of 3000 floats, it's what we call "close enough.")

Hint: think about the difference between `np.mean()` and `np.average()`. What are each of these measuring in the example above? Try visualizing happens to two points in 2D space when you run these operations.

Note: From now on, we will use x_i to refer to x_i^{cent}.

5.2 Find the Covariance Matrix ([15] points)

Recall, from lecture, that one of the interpretations of PCA is that it is the eigendecomposition of the sample covariance matrix. We will rely on this interpretation in this assignment, with all of the information you need below.

The covariance matrix is defined as

$$S = \frac{1}{n-1}\sum_{i=1}^{n} x_i x_i^\top.$$

Note that x_i is one of the n images in the (centered) dataset and is considered to be a column vector of size $d \times 1$. Therefore, S is a d-by-d matrix ; in mathematical notation, we say $S \in \mathbb{R}^{d\times d}$.

To calculate S, you'll need a couple of tools from `numpy` again:

```
>>> x = np.array([[1,2,5],[3,4,7]])
>>> np.transpose(x)
array([[1, 3],
       [2, 4],
       [5, 7]])
>>> np.dot(x, np.transpose(x))
array([[30, 46],
       [46, 74]])
>>> np.dot(np.transpose(x), x)
array([[10, 14, 26],
       [14, 20, 38],
       [26, 38, 74]])
```

The result of this function for our sample dataset should be a $d \times d$ (that is, 3000×3000) matrix.

5.3 Get the m Largest Eigenvalues and their Eigenvectors ([17] points)

Again, recall from lecture that eigenvalues and eigenvectors are useful objects that characterize matrices. Better yet, PCA can be performed by doing an eigendecomposition and taking the eigenvectors corresponding to the largest eigenvalues.

You may find `scipy.linalg.eigh` from the `scipy` library very helpful when writing this function. The optional parameter called `subset_by_index` might be of particular use.

Return the *largest* m *eigenvalues* of S as a m-by-m diagonal matrix Λ, in descending order, and the corresponding normalized eigenvectors as columns in a d-by-m matrix U. That is,

$$\Lambda = \begin{bmatrix} \lambda_1 & & & \\ & \lambda_2 & & \\ & & \ddots & \\ & & & \lambda_m \end{bmatrix}$$

and the j-th column of U is u_j, where λ_j is the j-th largest eigenvalue of S and $u_j \in \mathbb{R}^{d\times 1}$ is the corresponding eigenvector. In other words, Λ has **orthonormal columns**, *i.e.* each pair of column vectors are both *orthogonal* ('perpendicular' to each other in the latent space, i.e. their dot product is zero) and are *normalized* tp Euclidean norm equal to 1. You can verify this in code with `np.linalg.norm()`.

To return more than one thing from a function in Python, you can do this:

```
def multi_return():
    return "a string", 5
my_string, my_int = multi_return()
```

Make sure to return the diagonal matrix of eigenvalues *first*, then the eigenvectors in corresponding columns. You may have to rearrange the output of `eigh()` to get the eigenvalues in decreasing order and *make sure to keep the eigenvectors in the corresponding columns* after that rearrangement.

```
>>> S = get_covariance(x)
>>> Lambda, U = get_eig(S, 3)
>>> print(Lambda)

[[4635329.67030613       0.               0.        ]
 [      0.         1536838.7681917        0.        ]
 [      0.               0.         1014416.34892849]]
```

5.4 Get all Eigenvalues/Eigenvectors that Explain More than a Certain proportion of the Variance ([8] points)

Instead of pre-determining m, the number of top eigenvalues/eigenvectors, one may want to choose m in a way that includes all the "important" eigenvectors. We do this as follows. Recall that λ_i is the ith eigenvalue of the covariance matrix S. Then the following quantity,

$$\frac{\lambda_i}{\sum_{j=1}^{d} \lambda_j},$$

represents the *proportion of variance* in the dataset explained by the ith eigenvector. The larger this quantity is, the more important the ith eigenvalue/eigenvector are in capturing the information in the original dataset.

For a given number $0 \leq p \leq 1$, we want to use *all* the eigenvalues/eigenvectors that explain more than a proportion p of the variance. Return the eigenvalues as a diagonal matrix, in descending order, and the corresponding eigenvectors as columns in a matrix. Hint: `subset_by_index` was useful for the previous function, so perhaps something similar could come in handy here. Hint: what is the trace of a matrix?

Again, make sure to return the diagonal matrix of eigenvalues *first*, then the eigenvectors in corresponding columns. You may have to rearrange the output of `eigh()` to get the eigenvalues in decreasing order and *make sure to keep the eigenvectors in the corresponding columns* after that rearrangement.

Below is an example with $p = 0.07$.

```
>>> Lambda, U = get_eig_prop(S, 0.07)
>>> print(Lambda)
[[4635329.67030613       0.               0.        ]
 [      0.         1536838.7681917        0.        ]
 [      0.               0.         1014416.34892849]]
>>> print(U)
[[-0.03341778 -0.01905075  0.00551891]
 [-0.03239126 -0.02007646  0.00331054]
 [-0.03307422 -0.0179472   0.0053395 ]
 ...
  [-0.00877291  0.03295192  0.00428658]
```

```
[-0.00931975  0.02893783  0.00485262]
 [-0.0114543   0.02474475  0.00317719]]
```

Interlude: Understanding PCA

mathematical exploration of PCA.

Before we describe the next task in your assignment, let's explain PCA a little more formally. Recall that we have n data points $x_1, x_2, ..., x_n$, where each x_i is a d-dimensional column vector (Notation: x_i is $d \times 1$ or $x_i \in \mathbb{R}^{d \times 1}$). In this section, we will differentiate

1. The PCA Projection of each data point, $\alpha_i \in \mathbb{R}^{m \times 1}$
2. The PCA Reconstruction of each data point, $x_i^{pca} \in \mathbb{R}^{d \times 1}$

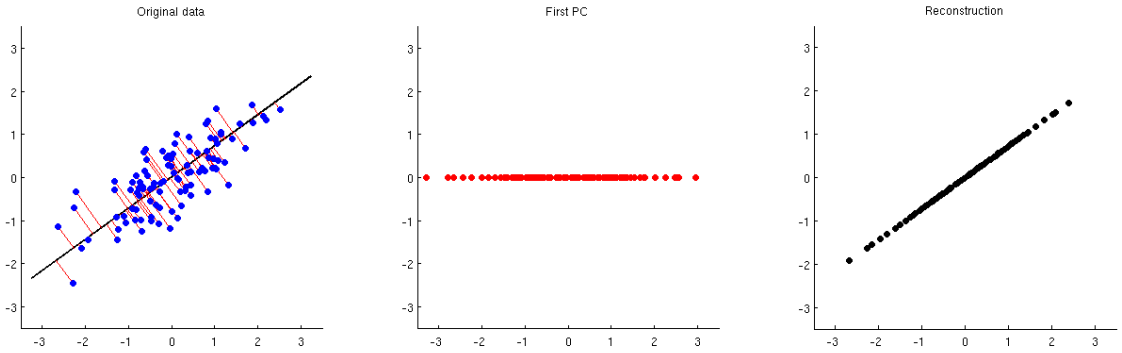

Figure 1: A visualization of PCA [1] with the blue dots representing some data points $x_1, \ldots, x_n \in \mathbb{R}^2$. The black line is the axis along the first principal component u_1, which minimizes the sum of squared projection errors. The second plot shows the value of the PCA projection as red dots, i.e., $\alpha_1, \ldots, \alpha_n \in \mathbb{R}^1$, where we have reduced the dimensionality of our data from 2 to 1. Finally the last plot shows the reconstructed data points $\boldsymbol{x_1^{pca}, \ldots, x_n^{pca}} \in \mathbb{R}^2$ in black dots, with visible reconstruction error when compared to the original data in the first plot.

5.4.1 PCA Projection

Recall that the covariance matrix of our data is

$$S = \frac{1}{n-1} \sum_{i=1}^{n} x_i x_i^\top \in \mathbb{R}^{d \times d}$$

Recall that our goal is to **minimize projection error** (sums of squared distances) while reducing the data dimensionality d. To do so, we project each data point $x_i \in \mathbb{R}^{d \times 1}$ on to the linear subspace spanned by the top-m eigenvectors of S. Explicitly, the PCA Projection (also called "score") of x_i is given by

$$\alpha_i = U^\top x_i,$$

where $U \in \mathbb{R}^{d \times m}$ is the matrix whose columns are the top eigenvectors $u_1, \ldots, u_m$, as defined earlier. As a sanity test, you can verify that α_i is a $m \times 1$ column vector, whose j-th element is $\alpha_{ij} = u_j^\top x_i$.

To summarize PCA Projection, the original data point $x_i \in \mathbb{R}^{d \times 1}$ has been projected to $\alpha_i \in \mathbb{R}^{m \times 1}$, and this projection is determined by the top-m eigenvectors of S contained in U.

5.4.2 PCA Reconstruction

How can we (approximately) reconstruct our data in our original d-dimensional feature space, i.e., compute $x_i^{pca} \in \mathbb{R}^{d \times 1}$ from our PCA projection α_i? We can use the eigenvectors U again and compute

$$x_i^{pca} = U\alpha_i = UU^\top x_i = \sum_{j=1}^{m} u_j u_j^\top x_i = \sum_{j=1}^{m} \alpha_{ij} u_j.$$

It is an exercise in linear algebra to prove that the three right-hand-side terms above are equal. (You do not need to prove this in this assignment, but feel free to use any of these equivalent expressions in your implementation.)

Figure 1 illustrates the PCA projection and reconstruction when we reduce the dimension from $d = 2$ to $m = 1$.

Hint: if we had not reduced the number of eigenvectors during PCA projection, i.e. maintained $m = d$, then from linear algebra we know that we would have $UU^\top = I$ (the identity matrix) and hence $x_i^{pca} = UU^\top x_i = Ix_i = x_i$, in which case we would perfectly reconstruct our original data. You may use this fact while debugging your code.

5.5 Project the Images ([15] points)

Given an image in the dataset and the eigenvectors from `get_eig()` or `get_eig_prop()`, compute the PCA representation of the image.

Recall that u_j is the jth column of U. Every u_j is an eigenvector of S with size $d \times 1$. If U has m eigenvectors, the image x_i is projected into an m dimensional subspace. The PCA projection represents images as a weighted sum of the eigenvectors. This projection only needs to store the weight for each eigenvector (m-dimensions) instead of the entire image (d-dimensions). The projection $\alpha_i \in \mathbb{R}^m$ is computed such that $\alpha_{ij} = u_j^T x_i$.

From each α_i we can compute the reconstruction $x_i^{\text{pca}} = \sum_{j=1}^{m} \alpha_{ij} u_j$. Notice that each eigenvector u_j is multiplied by its corresponding weight α_{ij}. The reconstructed image, x_i^{pca}, will not necessarily equal the original image because of the information lost projecting x_i to a smaller subspace. This information loss will increase as less eigenvectors are used. Implement `project_and_reconstruct_image()` to compute x_i^{pca} for an input image.

5.6 Visualize ([10] points)

Follow these steps to visualize your images using `matplotlib`. Pay careful attention to formatting and instructions, as this will matter for grading:

1. Reshape the images to be 60×50 (before this, they were being thought of as 3000 dimensional vectors in $\mathbb{R}^{3000}$).
2. Create a figure with one row of three subplots with `fig`, `ax1`, `ax2` and `ax3` objects. `ax1` should represent the original full resolution image, `ax2` should represent the original low-resolution image on which we run PCA, and `ax2` should represent the reconstructed image from PCA. Use `figsize=(9,3)` while initializing the figure (see starter code).
3. Title the first (leftmost) subplot as "Original High Res", the second (middle) subplot as "Original" and the last (rightmost) subplot as "Reconstructed".
4. Use `plt.imshow()` with the optional argument `aspect='equal'` to display the images on the correct axes. Pay careful attention to the format in which `imshow()` expects you to pass data.
5. Create a `colorbar` for `ax2` and `ax3` placed on the right (see sample plot in Figure 2 for a visualization. Your plot should match this).
6. Return the `fig`, `ax1`, `ax2` and `ax3` objects used in step 2 from `display_image()`.

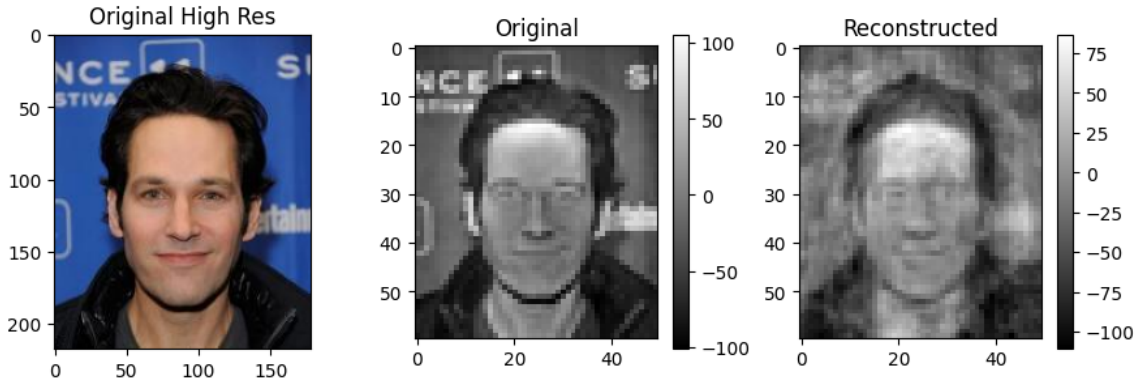

Figure 2: A visualization of the original high-resolution image, the compressed grayscale version of the original image on which we run PCA, and the reconstructed image after PCA projection with $m = 50$.

7. Testing: Render your plots. **Note that if the resulting figure needs rotation, you can transpose the matrix to render it properly.** DO NOT include this in your submission, this is only to test your implementation. We suggest calling `display_image()` from `main()`, and rendering images there, as shown in the example snippet below.

Below is a simple snippet of code for you to test your functions. Do **not** include it in your submission!

```
>>> X = load_and_center_dataset('celeba_60x50.npy')
>>> S = get_covariance(X)
>>> Lambda, U = get_eig(S, 50)
>>> celeb_idx = 34 # try out different indices to choose your favorite celebrity!
>>> x = X[celeb_idx]
>>> x_fullres = np.load('celeba_218x178x3.npy')[celeb_idx]
>>> reconstructed = project_and_reconstruct_image(x, U)
>>> fig, ax1, ax2 = display_image(x_fullres, x, reconstructed)
```

5.7 Reconstruction with Perturbed Weights ([15] points)

In this section, you will explore how altering the projection weights affects image reconstruction. Specifically, you will randomly perturb the projection weights and then reconstruct the images.

Modify the weights α obtained from the PCA projection of an image by adding small random perturbations. The perturbation should be drawn from a Gaussian distribution with mean 0 and a standard deviation σ, please note that the size of the Gaussian distribution should match the size of the weights. Use these perturbed weights to reconstruct the image using the eigenvectors from PCA. Compare the reconstructed image with the original projection-based reconstruction and the original image. Do **not** include these images in your submission.

Hint: You may find `np.random.normal()` useful to sample from a Gaussian distribution.

Hint: Perturbed weights are the addition of original weights and perturbation.

Below is a simple snippet of code for you to test your functions. Do **not** include this code in your submission.

```
>>> X = load_and_center_dataset('celeba_60x50.npy')
>>> S = get_covariance(X)
>>> Lambda, U = get_eig(S, 50)
>>> celeb_idx = 34 # try out different indices to choose your favorite celebrity!
>>> x = X[celeb_idx]
>>> x_fullres = np.load('celeba_218x178x3.npy')[celeb_idx]
```

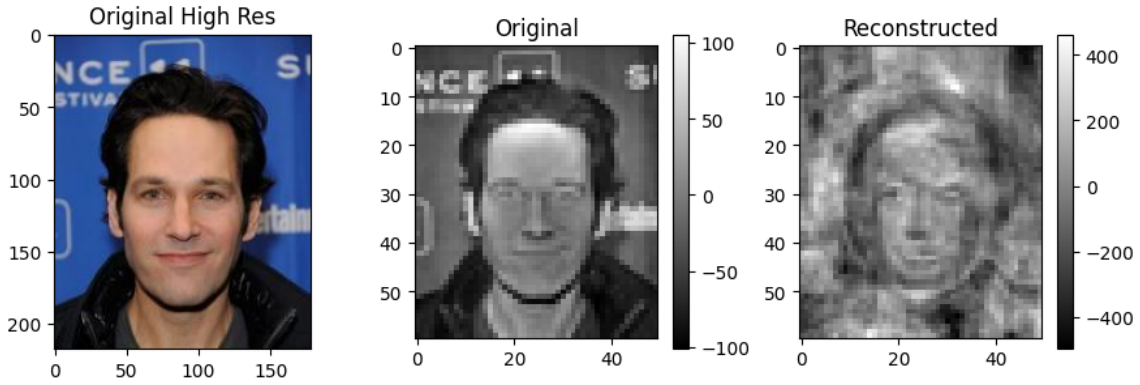

Figure 3: A visualization of the original high-resolution image, the compressed grayscale version of the original image on which we run PCA, and the reconstructed image after perturbing the PCA projection with Gaussian noise.

```
>>> x_perturbed = perturb_image(x, U, signma=1000)
>>> fig, ax1, ax2, ax3 = display_image(x_fullres, x, x_perturbed)
```

If you compare the original reconstruction in Figure 2 to the perturbed reconstruction in Figure 3, you can see that noise perturbation of the PCA projection has added a lot of noise to the reconstructed image as well!

6 Submission Notes

Please submit one file named `hw3.py` to Gradescope. Do *not* submit a Jupyter notebook `.ipynb` file.

- All of your functions should run silently (i.e. no output), except for the image rendering window in `display_image()`.
- No code should be put outside the function definitions (except for import statements; helper functions are allowed).

ALL THE BEST!

References

[1] amoeba. Making sense of principal component analysis, eigenvectors & eigenvalues. Cross Validated. url: https://stats.stackexchange.com/q/140579 (version: 2022-08-31).

[2] Ziwei Liu, Ping Luo, Xiaogang Wang, and Xiaoou Tang. Deep learning face attributes in the wild. In *Proceedings of International Conference on Computer Vision (ICCV)*, December 2015.