Homework 5 **PSY4219/6219**
Due October 12 in class **Fall 2022**
18 points

Your code should use good Python style, avoid hard-coding, use vectorized numpy operations unless directed otherwise, and define Python functions where directed.

Q1. Explore matrix multiplication.

(a) (3 points) Write a Python function that performs matrix multiplication of numpy arrays **A** and **B** using `for` loops. Your function should use a conditional `if-then-else` or an `assert` (your choice) to make sure the dimensions of the numpy arrays support matrix multiplication. Do not use built-in operators for performing matrix multiplication or for calculating a dot product. Show that your code works by comparing the value of the matrix multiplication of **A** and **B** calculated using your function with the value calculated using vectorized mathematical operations (the `@` operator or `np.matmul()`).

(b) (1 point) Create randomized numpy arrays **A** and **B** using `np.random.rand()` with dimensions suitable to allow matrix multiplication; the dimensions (rows and columns) should be specified by variables that can be set (not hardcoded into the `rand()` function). Explore the timing of matrix multiplication using your function (defined in terms of `for` loops) and the vectorized mathematical operations using the cell magic `%%timeit`. Explore using relatively small matrices and relatively large matrices (in other words, have one cell with small arrays and another cell with large arrays).

Q2. Given two-dimensional numpy arrays **A**, **B**, and **C**, I want you to explore the commutativity and associativity of elementwise multiplication and matrix multiplication in Python code (of course a demonstration is no substitute for a formal proof). These should each reside in their own Jupyter notebook cell.

(a) (1 point) In a Jupyter cell, using vectorized numpy array operations, test whether **A** x **B** = **B** x **A** (commutativity) for elementwise multiplication. Use a conditional (`if-then-else`) to test whether the dimensions of the two arrays allow elementwise multiplication. Your code should work for any two numpy arrays (**A** and **B**).

(b) (1 point) In a Jupyter cell, using vectorized numpy array operations, test whether **A** x (**B** x **C**) = (**A** x **B**) x **C** (associativity) for elementwise multiplication. Use a conditional (`if-then-else`) to test whether the dimensions of the three arrays allow elementwise multiplication. Your code should work for any three numpy arrays (**A**, **B**, and **C**).

(c) (1 point) In a Jupyter cell, using vectorized numpy array operations, test whether **A** x **B** = **B** x **A** (commutativity) for matrix multiplication. Use a conditional (`if-then-else`) to test whether the dimensions of the two arrays allow matrix multiplication. Your code should work for any two numpy arrays (**A** and **B**).

(d) (1 point) In a Jupyter cell, using vectorized numpy array operations, test whether **A** x (**B** x **C**) = (**A** x **B**) x **C** (associativity) for matrix multiplication. Use a condition (`if-then-else`) to test whether the dimensions of the three arrays allow matrix multiplication. Your code should work for any three numpy arrays (**A**, **B**, and **C**).

At the start of each cell, include a set of **A**, **B**, and **C** arrays that have the proper dimensions to allow that form of multiplication (that pass the conditional test) and include a set (commented out) of **A**, **B**, and **C** arrays that do not have the proper dimensions (that fail the conditional test). The **A**, **B**, and **C** arrays do not need to be the same across the four cells.
Q3. Explore matrix inversion.

(a) (2 points) In Python code, convince yourself (and me) that the inverse of a matrix (using `np.linalg.inv()`) is not the same as the matrix of the inverses ($1/x$) of each of its elements. First, I want you to show me that they are not equal to one another. Second, I also want you to show me that the former (the inverse of a matrix) and not the latter (the matrix of the inverse of its elements) satisfies the key property for what it means to be an inverse of a matrix – that the $\mathbf{A}^{-1}$ x **A** = **I** (where **I** is the identity matrix) – though note that depending on the values of the elements of your matrix, you might only see an approximation to the identity matrix. Use vectorized operations to calculate the inverse matrix and to calculate the inverse ($1/x$) of the elements of the matrix.

Note that for some matrices it is impossible to calculate an inverse. Python will often signal this by saying that the matrix is "singular" (sometimes called "degenerate"). If you find that your matrix is singular (noninvertible) try another matrix until you find one that can be inverted (I showed some examples in class where non-invertible singular matrices might be found). I am not asking you to test that a matrix is singular (there are ways of testing for that, for example by calculating whether the so-called "determinant of the matrix" equals, or is numerically close to, zero).

(b) (1 point) Create a singular matrix. Show that its inverse is undefined using both the matrix inversion function `np.linalg.inv()` (it either throws an error that the matrix is singular, or results in a matrix with huge values) and by showing that the determinant of the matrix (using `np.linalg.det()`) equals zero (or numerically close to zero). Recall that singular matrices include ones where a row (or column) is a linear combination of another row (or column). Make clear in your code how you generated the singular matrix.

(c) (1 point) Convince yourself (and me) that for the inverse of a diagonal matrix (with non-zero diagonal elements) is equal to the matrix of the inverses ($1/x$) of its non-zero elements. Recall from class that you can create a diagonal matrix using `np.diag()`.

Q4. Explore Singular Value Decomposition (SVD).

I want you to read in the image file `BoysBW.jpg` using the example code provided in class (in `SVD.ipynb`). Treat this image as a matrix (its elements are the pixel intensity values of a black-and-white image, but it can still just be treated as a matrix).

(a) (1 point) Following the examples from class (on smaller matrices) perform an SVD on this matrix; note that since this matrix is large, it will take a few seconds for the SVD computation to be performed. Show that multiplying the components together recreates a reconstructed version of the original image by both displaying the reconstruction (using `plt.imshow()`) and by comparing numerically the reconstruction with the original matrix (e.g., showing that the sum of the values in the difference matrix is numerically close to zero).

(b) (3 points) Next, I want you to explore what happens when you reduce the dimensionality of the component **U**, **Σ**, and $\mathbf{V}^T$ matrices and then try to reconstruct the original image. I explain what this means below (and I discussed it a bit in class).

Recall that given an n x m matrix **X**, the SVD components have dimensions as follows:

U	n x n
Σ	n x m
$\mathbf{V}^T$	m x m

The image is landscape (not portrait), with n < m, so the n x m **Σ** matrix has extra columns with 0s.

Recall that with n < m, the SVD algorithm

```
(U, S, Vt) = np.linalg.svd(X, full_matrices=False)
S = np.diag(S)
```

will return numpy arrays with dimensions as follows:

`U`	n x n
`S`	n x n
`Vt`	n x m

A reduced SVD is one that uses only the first r rows or columns as follows (the r added to the variable names denote that these are reduced versions):

`Ur`	n x r
`Sr`	r x r
`Vtr`	r x m

Obviously, r <= n < m.

It should be obvious that multiplying `Ur`, `Sr`, and `Vtr` results in a new n x m matrix.

Recall from class that the entries in `S` are ordered from largest to smallest along the diagonal. Commensurate with the ordered values along the diagonals, the columns and

rows of `U` and `Vt` are ordered in terms of their "importance" as well. These values relate to the principal components in the data **X**; indeed, SVD is one method for computing PCA, for those who have heard of PCA before. Using the first r columns of `U`, the first r diagonals of `S`, and the first r rows of `Vt` means using the r most important pieces of information in the matrix decomposition.

Create code for producing a reduced SVD based on a value stored in variable `r`.

Explore how the reconstruction of the original image is affected by the value of r. For this assignment, that means showing the resulting matrix using `imshow()`. r = 1 represents the minimal amount of information used to reconstruct the image (indeed, the reconstruction should look nothing like the original image). r = n represents the maximal amount of information (the reconstruction will be identical to the original image). Explore systematically what happens as you vary the value of r. What you should observe is that you do not need r to be very large to produce a reasonable reconstruction of the original image.

(c) (2 points) This approach to dimensionality reduction can be used as a method for image compression (there are many algorithms that do this, SVD is just one). Write code that computes the percent reduction in size for the reduced SVD representation (assuming a particular value of r and the resulting `Ur`, `Sr`, and `Vtr` matrices) compared to the original matrix `X`. To compute this, you will need to compute the sizes (number of elements) of all of the relevant matrices and use these values to appropriately compute the percent reduction in size. (So if the reduced SVD representation across `Ur`, `Sr`, and `Vtr` required 10% of the number of elements in these matrices as the original matrix `X` read from the file, this would represent a 90% decrease).

Unexcused late assignments will be penalized 10% for every 24 hours late, starting from the time class ends, for a maximum of two days, after which they will earn a 0.