Carnegie Mellon University
16-385: Computer Vision

Homework 0: Introduction

- **Grading.** This assignment will not be graded, but all students that successfully complete the submission process outlined below will receive 1 % extra-credit on your final course grade.
- **Due date.** Please refer to course schedule for the due date for HW0.
- **Gradescope submission.** You will need to submit both (1) a YourAndrewName.pdf and (2) a YourAndrewName.zip file containing your code, either as standalone python (.py or colab notebooks (.ipynb). We suggest you create a PDF by printing your colab notebook from your browser. However, this can improperly cutoff images that straddle multiple pages. In this case, we suggest you download such images separately and explicitly append them to your PDF using online tools such as https://combinepdf.com/. You may also find it useful to look at this post: https://askubuntu.com/questions/2799/how-to-merge-several-pdf-files. Remember to use Gradescope's functionality to mark pages that provide answers for specific questions, such as code or visualizations.
- **Acknowledgements.** This homework was created by David Fouhey.

1 Overview

In this assignment, you'll work through three tasks that help set you up for success in the class as well as a short assignment involving playing with color. The document seems big, but most of it is information, advice, and hand-holding. Indeed, one section was a bit more open-ended and terse in the past, and has been expanded to four pages that walk you through visualizing images. The assignment has two goals.

First, the assignment will show you bugs in a low-stakes setting. You'll encounter a lot of programming mistakes in the course. If you have buggy code, it could be that the concept is incorrect (or incorrectly understood) or it could be that the implementation in code is incorrect. You'll have to sort out the difference. It's a lot easier if you've seen the bugs in a controlled environment where you know what the answer is. Here, the programming problems are deliberately easy, and we even provide the solution for one!

Second, the assignment incentivizes you to learn how to write reasonably good python and numpy code. You should learn to do this anyway, so this gives you credit for doing it and incentivizes you to learn things in advance.

Collaboration Policy. For this assignment, you are encouraged to collaborate in any way, shape or form. Note that future assignments will require you to perform your own work.

The actual assignment. The assignment has two parts and corresponding folders in the starter code:

1. Numpy (Section 2 - folder numpy/)
2. Data visualization (Section 3 folder visualize/)

Here's our recommendation for approaching this homework:

1. If you have not had any experience with Numpy, read this tutorial. Numpy is like a lot of other high-level numerical programming languages. Once you get the hang of it, it makes a lot of things easy. However, you need to get the hang of it and it won't happen overnight!
2. You should then do Section 2.
3. You should then read our description about images in Section 3. Some will make sense; some may not. That's OK! This is a bit like learning to ride a bike, swim, cook a new recipe, or play a new game by being told by someone. A little teaching in advance helps, but actually doing it yourself is crucial. Then, once you've tried yourself, you can revisit the instructions (which might make more sense). *If you haven't recently thought much about the difference between an integer and a floating point number, or thought about multidimensional arrays, it is worth brushing up on both.*
4. You should then do Section 3, which is designed to produce common bugs, issues, and challenges that you will likely run into during the course.

1.1 Assignment Setup:

This assignment is provided in a Colab notebook for convenience.

Here is a link to the Google Colab Notebook. Save a copy of this notebook in your Drive by clicking on `File -> Save a copy to Drive`. Alternatively, you can also upload the `HW0.ipynb` file in the `hw0` folder to a new Colab notebook.

Start by running all cells in Section 1 to download the necessary data, create the required folders, and set up the environment. The data will be saved to the Google Drive account linked to the Colab session.

2 Numpy Intro (40 points)

Fill in the code stubs in Sections 2.1: Test functions and 2.2: Warmup functions.

2.1 Test and Warmup functions

For each function ti or wi, you will see starter code that looks like this:

```python
def sample1(xs):
 """
 Inputs:
 - xs: A list of values

 Returns:
 The first entry of the list
 """
 return None
```

You should fill in an implementation of the function like this:

```python
def sample1(xs):
 """
 Inputs:
 - xs: A list of values

 Returns:
 The first entry of the list
 """
 return xs[0]
```

You can test your implementation by using the code in Section 2.3: Checking your implementation. Use these function parameters to test selective functions:

```python
# Run specific problems
test_list=["t1", "t2","w1","w2"]
run_tests(get_test_list(specific_tests=test_list), store=True)
```

Use these parameters to check all tests and warmup functions. These results should be visible in your final report.

```python
# Run all tests
run_tests(get_test_list(all_tests=True), store=True)

# Run all warmups
run_tests(get_test_list(all_warmups=True), store=True)
```

Do I have to get every question right? We give partial credit: each warmup exercise is worth 2% of the total grade for this question and each test is worth 3% of the total grade for this question.

2.2 Test functions

You need to solve all 20 problems in tests.py. Many are not solvable in one line. You may not use a loop to solve any of the problems, although you may want to first figure out a slow for-loop solution to make sure you know what the right computation is, before changing the for-loop solution to a non for-loop solution. The one exception to the no-loop rule is t10 (although this can also be solved without loops). Here is one example:

```python
def t4(R, X):
    """
    Inputs:
    - R: A numpy array of shape (3, 3) giving a rotation matrix
    - X: A numpy array of shape (N, 3) giving a set of 3-dimensional vectors

    Returns:
    A numpy array Y of shape (N, 3) where Y[i] is X[i] rotated by R

    Par: 3 lines
    Instructor: 1 line

    Hint:
    1) If v is a vector, then the matrix-vector product Rv rotates the vector
    by the matrix R.
    2) .T gives the transpose of a matrix
    """
    return None
```

2.3 Warmup functions

You need to solve all 20 of the warmup problems in warmups.py. They are all solvable with one line of code.

2.4 What We Provide

For each problem, we provide:

Inputs: The arguments that are provided to the function

Returns: What you are supposed to return from the function

Par: How many lines of code it should take. We don't grade on this, but if it takes more lines than this, there is probably a better way to solve it. Except for t10, you should not use any explicit loops.

Instructor: How many lines our solution takes. Can you do better?

Hints: Functions and other tips you might find useful for this problem.

2.5 Walkthroughs and Hints

Test 8: If you get the axes wrong, numpy will do its best to make sure that the computations go through but the answer will be wrong. If your mean variable is 1×1, you may find yourself with a matrix where the full matrix has mean zero. If your standard deviation variable is 1×M, you may find each column has standard deviation one.

Test 9: This sort of functional form appears throughout data-processing. This is primarily an exercise in writing out code with multiple nested levels of calculations. Write each part of the expression one line at a time, checking the sizes at each step.

Test 10: This is an exercise in handling weird data formats. You may want to do this with for loops first.

1. First, make a loop that calculates the vector `C[i]` that is the centroid of the data in `Xs[i]`. To figureout the meaning of things, there is no shame in trying operations and seeing which produce the right shape. Here we have to specify that the centroid is M-dimensions to point out how we want `Xs[i]` interpreted. The centroid (or average of the vectors) has to be calculated with the average going down the columns (i.e., rows are individual vectors and columns are dimensions).
2. Allocate a matrix that can store all the pairwise distances. Then, double for loop over the centroids i and j and produce the distances.

Test 11: You are given a set of vectors $x_1, \ldots, x_N$, where each vector $x_i \in \mathbb{R}^M$. These vectors are stacked together to create a matrix $X \in \mathbb{R}^{N \times M}$. Your goal is to create a matrix $D \in \mathbb{R}^{N \times N}$ such that $D_{i,j} = \|x_i - x_j\|$. Note that $\|x_i - x_j\|$ is the L2-norm or Euclidean length of the vector. The useful identity you are given is $\|x - y\|^2 = \|x\|^2 + \|y\|^2 - 2x^\top y$, which holds for any pair of vectors x and y and can be used to calculate distances efficiently.

At each step, your goal is to replace slow but correct code with fast but correct code. If the code breaks at a particular step, you know where the bug is.

1. First, write a correct but slow solution using two for-loops. In the inner loop, plug in the identity to compute $D_{i,j} = \|x_i\|^2 + \|x_j\|^2 - 2x_i^\top x_j$. DO this in three separate terms.

2. Next, write a version that first computes a matrix that contains all the dot products, or $P \in \mathbb{R}^{N \times N}$ such that $P_{i,j} = x_i^\top x_j$. This can be done in a single matrix-matrix operation. You can then calculate the distance by doing $D_{i,j} = \|x_i\|^2 + \|x_j\|^2 - 2P_{i,j}$.

3. Finally, calculate a matrix containing the norms, $N \in \mathbb{R}^{N \times N}$ such that $N_{i,j} = \|x_i\|^2 + \|x_j\|^2$. You can do this in two stages: first, calculate the squared norm of each row of X by summing the squared values across the row. Suppose S is this array, (i.e. $S_i = \|x_i\|^2$ and $S \in \mathbb{R}^N$) but be careful that you look at the shape that you get as output. If you compute $S + S^\top$ you should get N. Now you can calculate the distance inside the double for loop as $D_{i,j} = N_{i,j} - 2P_{i,j}$.

4. The double for-loop is now not very useful. You can just add/scale the two arrays together element wise.

Test 18: Here you draw a circle by finding all entries in a matrix that are within r cells of a row y and column x. This is a not particularly intellectually stimulating exercise, but it is practice in writing (and debugging) code that reasons about rows and columns.

1. First, write a correct but slow solution that uses two for loops. In the inner body, plug in the correct test for the given i, j. Make sure the test passes; be careful about rows and columns.
2. Look at the documentation for `np.meshgrid` briefly. Then call it with `np.arange(3)` and `np.arange(5)` as arguments. See if you can create two arrays such that `IndexI[i,j] = i` and `IndexJ[i,j] = j`.
3. Replace your test that uses two for loops with something that just uses `IndexI and IndexJ`.

3 Data Interpretation and Making Your Own Visualization (10 points)

Throughout the course, a lot of the data you have access to will be in the form of an image. These won't be stored and saved in the same format that you're used to when interacting with ordinary images, such as off your cell phone: sometimes they'll have negative values, really really big values, or invalid values. If you can look at images quickly, then you'll find bugs quicker. If you only print debug, you'll have a bad time. To teach you about interpreting things, I've got a bunch of mystery data1 that we'll analyze together. You'll write a brief version of the important imsave function for visualizing.

Let's load some of this mysterious data.

```
>> data = np.load("mysterydata.npy")
>> data
  [[[0 0 0 ... 0 0 1]
  [0 0 0 ... 0 0 0]
  [0 0 0 ... 0 0 0]
  ...
  [0 0 0 ... 0 0 0]
  [0 0 0 ... 0 0 0]
  [0 0 0 ... 0 0 0]]

 some more zeros

 [[0 0 0 ... 0 0 0]
  [0 0 0 ... 0 0 0]
  [0 0 0 ... 0 0 0]
  ...
  [0 0 0 ... 0 0 0]
  [0 0 0 ... 0 0 0]
  [0 0 0 ... 0 0 0]]]
```

Looks like it's a bunch of zeros. Nothing to see here folks! For better or worse: Python only shows the sides of an array when printing it and the sides of images that have gone through some processing tend to not be representative. After you print it out, you should always look at the shape of the array and the data type.

```
>>> data.shape
 (512, 512, 9)
 >>> data.dtype
 dtype('uint32')
```

The shape and datatype are really important. If something is an unexpected shape, that's really bad. This is similar to doing a physics problem where you're trying to measure the speed of light (which should be in m/s), and getting a result that is in gauss. The computer may happily chug along and produce a number, but if the units or shape are wrong, the answer is almost certainly wrong. The datatype is also really important, because data type conversion can be lossy: if you have values between 0 and 1, and you convert to an integer format, you'll end up with 0s and 1s.

In this particular case, the data has height 512 (the first dimension), width 512 (the second), and 9 channels (the last). These order of dimensions here is a convention, meaning that there are multiple equivalent options. For instance, in other conventions, the channel is the first dimension. Generally, you'll be given data that is HxWxC and we'll try to be clear about how data is stored. If later on, you're given some other piece of data, figuring out the convention is a bit like rotating a picture until it looks right: figure out what you expect, and flip things until it looks like you'd expect.

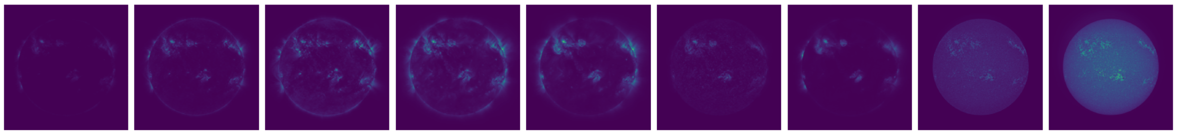
vis_0.png vis_1.png vis_2.png vis_3.png vis_4.png vis_5.png vis_6.png vis_7.png vis_8.png

Figure 1: The Mystery Data

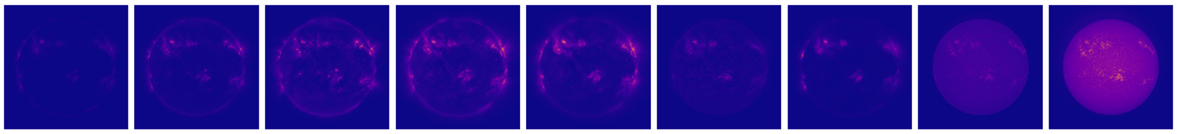
vis_0.png vis_1.png vis_2.png vis_3.png vis_4.png vis_5.png vis_6.png vis_7.png vis_8.png

Figure 2: The Mystery Data, Visualized with the Plasma Colormap

If you've got an image, after you print it, you probably want to visualize the output. We don't see in 9 color channels, and so you can't look at them all at once. If you've got a bug, one of the most important things to do is to look at the image. You can look at either the first channel or all of the channels as individual images:

```
    plt.imsave("vis.png",data[:,:,0])
```

You can see what the output looks like in Figure 1. This is a false color image. You can read about these in Section A.3, but the short version is that the given image, you find the minimum value (vmin) and the maximum value (vmax) and assign colors based on where each pixel's value falls between those. These colors look like: Low ▨ High. plt.imsave finds vmin and vmax for you on its own by computing the minimum and maximum of the array. If you'd like to look at all 9 channels, save 9 images:

```
for i in range(9):
    plt.imsave("vis_%d.png" % i,data[:,:,i])
```

If you'd like to change the colormap, you can specify this with cmap. For instance

```
for i in range(9):
    plt.imsave("vis_%d.png" % i,data[:,:,i])
```

produces the outputs in Figure 2. These use the plasma colormap.

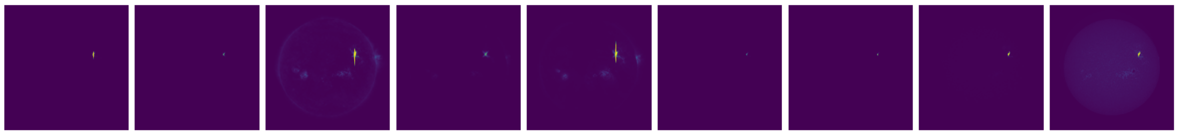
vis2_0.png vis2_1.png vis2_2.png vis2_3.png vis2_4.png vis2_5.png vis2_6.png vis2_7.png vis2_8.png

Figure 3: The Mystery Data #2.

3.1 Images with wide ranges of values (2 points)

Try loading mysterydata2.npy and visualizing it. You should get something like Figure 3. It's hard to see stuff because one spot is really bright. In this case, it's because there's a solar flare that's producing immense amounts of light. A common trick for making things easier to see is to apply a nonlinear correction. Here are a few options:

$$p^\gamma \text{ with } \gamma \in [0,1], \quad \log(p), \quad \log(1+p)$$

where the last one can be done with `np.log1p`. Apply a nonlinear correction to the second mystery data and visualize it. Put two of the channels (i.e. data[:,:,i] as images into the report)

3.2 Images that screw up and knowing your limits (2 points)

vis3_0.png vis3_1.png vis3_2.png vis3_3.png vis3_4.png vis3_5.png vis3_6.png vis3_7.png vis3_8.png

Figure 4: The Mystery Data # 3, Visualized!

Let's try this again, using one of the other data.

```python
data = np.load("mysterydata/mysterydata3.npy")
for i in range(9):
    plt.imsave("vis3_%d.png" % i,data[:,:,i])
```

The results are shown in Figure 4. They're all white. What's going on? If you've got an uncooperative piece of data that won't visualize or produces buggy results, it's worth checking to see if all the values are reasonable. One option that'll cover all your cases is `np.isfinite`, which is False for values that are NaN (not a number) or $\pm\infty$ and True otherwise. If you then take the mean over the array, you get the fraction of entries that are normalish. If the mean value is *anything* other than 1, you may be in trouble. Here:

```python
np.mean(np.isfinite(X))
0.6824828253851997
```

Alternatively, this also works:

```python
np.sum(~np.isfinite(X))
749117
```

Other things that check things are np.isnan (which returns True for NaNs) and np.isinf (which returns True for infinite values). Even a single NaN in your data is a problem: any number that touches another NaN turns into a NaN. The totally white values happen because plt.imsave tries to find vmin/vmax, and the minimum of a value and a NaN is a NaN. The resulting color is a NaN as well. If you've got NaNs in your data, many functions you may want to use (e.g., mean, sum, min, max, median, quantile) have a nan version.

Section 3.2: Fix the images by determining the right range (use np.nanmin, np.nanmax) and then pass arguments into plt.imsave to set the range for the visualization. To figure out what arguments to set, look at the documentation for plt.imsave. Put two images from mystery data 3 in the report.

3.3 Rolling your own plt.imsave

You'll make your own plt.imsave by filling in colormapArray. Here's how the false color image works: You're given a H $\times$ W matrix X and a colormap matrix C that is N $\times$ 3 where each row is a color red/green/blue. You produce a H $\times$ W $\times$ 3 matrix O. Each scalar-valued pixel X[i, j] gets converted into red/green/blue values for O[i, j, :] following this rule: if the pixel has value v the corresponding output color comes from a row determined by (approximately)

$$(N-1)\frac{(v-v_{\text{min}})}{(v_{\text{max}}-v_{\text{min}})}$$

However, you'll have to be very careful – this precise equation won't always work. As an exercise – can you spot something that might cause the expression to be a NaN?

Coding 3.3 (3 points): Fill in colormapArray. You can use either plt.imsave or cv2.imwrite to save the image to a file.

Report 3.4 (3 points): Visualize mysterydata4.npy using your system without it crashing and put all nine images into your report. You may have to make a design decision about what to do about results that are undefined. If the results are undefined, then any option that seems reasonable is fine. Your colormap should look similar to Figure 2. If the colors look inverted, see Beware 3!

Beware 1! There are a bunch of edge cases in the equation for the color: it won't always return an integer between 0 and N - 1. It will also definitely blow up under certain input conditions (also, watch the type).

Beware 2! You're asked by the code to return a H$\times$ W uint8 image. There are a lot of shortcuts/implied sizes and shapes in computer vision notation – since this is a HxW color image, it should have 3 channels (i.e.,be H$\times$ W$\times$ 3). Since it's uint8, you should make the image go from 0 to 255 before returning it (otherwise everything gets clipped to 0 and 1, which correspond to the two lowest brightness settings). Like all other jargon, this is annoying until it is learned; after it is learned, it is then useful.

Beware 3! If you choose to save the results using opencv, you may have blue and red flipped – opencv assumes blue is first and the rest of the world assumes red is first. You can identify this by the fact that the columns of the are defined as Red/Green/Blue and there is a lot of blue and not much red in the lowest entry.