Submission Requirements and Description (Very Important !)

Format requirements

(i) Project materials including report template: `proj1.zip`

(ii) Please use the provided **proj1_report.pptx** template to write your report, and convert the slide deck into a PDF for your submission. The report should contain your name, student ID, and e-mail address;

(iii) You should choose python to write your code, and provide a README file to describe how to execute the code;

(iv) Pack your **proj1_report.pdf**, **proj1_code** and **README** into a zip file, named with your student ID, like MG1933001.zip. If you have an improved version, add an extra '_' with a number, like MG1933001_1.zip. We will take the final submitted version as your results.

(v) Do **not** install any additional packages inside the conda environment. The TAs will use the same environment as defined in the config files we provide you, so anything that's not in there by default will probably cause your code to break during grading. Do not use absolute paths in your code or your code will break. Use relative paths like the starter code already does. Failure to follow any of these instructions will lead to point deductions.

Submission Way

(i) Please submit your results to email nju.cvcourse@gmail.com , the email subject is "Assignment 1";

(ii) The deadline is 23:59 on April 2, 2022. No submission after this deadline is acceptable.

About Plagiarize

DO NOT PLAGIARIZE! We have no tolerance for plagiarizing and will penalize it with giving zero score. You may refer to some others' materials, please make citations such that one can tell which part is actually yours.

Evaluation Criterion

We mainly evaluate your submission according to your code and report. Efficient implementation, elegant code style, concise and logical report are all important factors towards a high score.

Overview

The goal of this assignment is to write an image filtering function and use it to create hybrid images using a simplified version of the SIGGRAPH 2006 paper by Oliva, Torralba, and Schyns. *Hybrid images* are static images that change in interpretation as a function of the viewing distance. The basic idea is that high frequency tends to dominate perception when it is available but, at a distance, only the low frequency (smooth) part of the signal can be seen. By blending the high frequency portion of one image with the low-frequency portion of another, you get a hybrid image that leads to different interpretations at different distances.

This project is intended to familiarize you with Python, PyTorch, and image filtering. Once you have created an image filtering function, it is relatively straightforward to construct hybrid images. If you don't already know Python, you may find this resource helpful. If you're unfamiliar with PyTorch, the tutorials from the official website are useful.

Setup

(i) Install Miniconda. It doesn't matter whether you use Python 2 or 3 because we will create our own environment that uses python3 anyways.

(ii) Download and extract the project starter code.

(iii) Create a conda environment using the appropriate command. On Windows, open the installed "Conda prompt" to run the command. On MacOS and Linux, you can just use a terminal window to run the command, Modify the command based on your OS (`linux, mac, or win`): `conda env create -f proj1_env_<OS>.yml`

(iv) This will create an environment named "cv_proj1". Activate it using the Windows command, `activate cv_proj1` or the MacOS / Linux command, `conda activate cv_proj1` or `source activate cv_proj1`

(v) Install the project package, by running `pip install -e .` inside the repo folder. This might be unnecessary for every project, but is good practice when setting up a new `conda` environment that may have `pip` requirements.

(vi) Run the notebook using `jupyter notebook ./proj1_code/proj1.ipynb`

(vii) After implementing all functions, ensure that all sanity checks are passing by running `pytest proj1_unit_tests` inside the repo folder.

(viii) Generate the zip folder for the code portion of your submission once you've finished the project.

1 Numpy(40 points)

1.1 Gaussian kernels

Gaussian filters are used for blurring images. You will first implement `create_Gaussian_kernel_1D()`, a function that creates a 1D Gaussian vector according to two parameters: the kernel size (length of the 1D vector) and σ, the standard deviation of the Gaussian. The vector should have values populated from evaluating the 1D Gaussian pdf at each coordinate. The 1D Gaussian is defined as:

$$p(x;\mu,\sigma^2) = \frac{1}{\sqrt{2\pi}\sigma}\exp\left(-\frac{1}{2\sigma^2}(x-\mu)^2\right) \tag{1.1}$$

Next, you will implement `create_Gaussian_kernel_2D()`, which creates a 2-dimensional Gaussian kernel according to a free parameter, cutoff frequency, which controls how much low frequency to leave in the image. Choosing an appropriate cutoff frequency value is an important step for later in the project when you create hybrid images. We recommend that you implement `create_Gaussian_kernel_2D()` by creating a 2D Gaussian kernel as the outer product of two 1D

Gaussians, which you have now already implemented in `create_Gaussian_kernel_1D()`. This is possible because the 2D Gaussian filter is separable (think about how $e^{(x+y)} = e^x \cdot e^y$). The multivariate Gaussian function is defined as:

$$p(x; \mu, \Sigma) = \frac{1}{(2\pi)^{n/2} \det(\Sigma)^{1/2}} \exp\left(-\frac{1}{2}(x-\mu)^\top \Sigma^{-1}(x-\mu)\right) \tag{1.2}$$

where n is equal to the dimension of x, μ is the mean coordinate (where the Gaussian has peak value), and Σ is the covariance matrix.

You will use the value of cutoff frequency to define the size, mean, and variance of the Gaussian kernel. Specifically, the kernel G should be size $(k; k)$ where $k = 4 \cdot$ cutoff frequency $+1$, have peak value at $\mu = \lfloor \frac{k}{2} \rfloor$, standard deviation $\sigma =$ cutoff frequency, and values that sum to 1, i.e., $\sum_{ij} G_{ij} = 1$. If your kernel doesn't sum to 1, you can normalize it as a postprocess by dividing each value by the sum of the kernel.

1.2 Image Filtering

Image filtering (or convolution) is a fundamental image processing tool. See chapter 3.2 of Szeliski and the lecture materials to learn about image filtering (specifically linear filtering). You will be writing your own function to implement image filtering from scratch. More specifically, you will implement `my_conv2d_numpy ()` which imitates the `filter2D()` function in the OpenCV library. As specified in `part1.py`, your filtering algorithm must: (1) support grayscale and color images, (2) support arbitrarily-shaped filters, as long as both dimensions are odd (e.g., 7 × 9 filters, but not 4 × 5 filters), (3) pad the input image with zeros, and (4) return a filtered image which is the same resolution as the input image. We have provided an iPython notebook, `proj1.ipynb` and some unit tests (which are called in the notebook) to help you debug your image filtering algorithm. Note that there is a time limit of 5 minutes for a single call to `my_conv2d_numpy()`, so try to optimize your implementation if it goes over.

1.3 Hybrid Images

A hybrid image is the sum of a low-pass filtered version of one image and a high-pass filtered version of another image. As mentioned above, *cutoff frequency* controls how much high frequency to leave in one image and how much low frequency to leave in the other image. In `cutoff_frequencies.txt`, we provide a default value of 7 for each pair of images (the value of line i corresponds to the cutoff frequency value for the i-th image pair). You should replace these values with the ones you find work best for each image pair. In the paper it is suggested to use two cutoff frequencies (one tuned for each image), and you are free to try that as well. In the starter code, the cutoff frequency is controlled by changing the standard deviation of the Gaussian filter used in constructing the hybrid images. You will first implement `create_hybrid_image()` according to the starter code in `part1.py`. Your function will call `my_conv2d_numpy()` using the kernel generated from `create_Gaussian_kernel()` to create low and high frequency images, and then combine them into a hybrid image.

2 PyTorch(30 points)

2.1 Dataloader

You will now implement creating hybrid images again but using PyTorch. The `HybridImageDataset` class in `part2_datasets.py` will create tuples using pairs of images with a corresponding cutoff frequency (which you should have found from experimenting in Part 1). The image paths will be loaded from `data/` using `make_dataset()` and the cutoff frequencies from `cutoff_frequencies.txt` using `get_cutoff_frequencies()`. Additionally, you will implement `__len__()`, which returns the number of image pairs, and `__getitem__()`, which returns the i-th tuple. Refer to this tutorial for additional information on data loading and processing.

2.2 Model

Next, you will implement the HybridImageModel class in `part2_models.py`. Instead of using your implementation of `my_conv2d_numpy()` to get the low and high frequencies from a pair of images, `low_pass()` should use the 2D convolution operator from `torch.nn.functional` to apply a low pass filter to a given image. You will have to implement `get_kernel()` which calls your `create_Gaussian_kernel()` function from `part1.py` for each pair of images using the cutoff frequencies as specified in `cutoff_frequencies.txt`, and reshape it to the appropriate dimensions for PyTorch. Then, similar to `create_hybrid_image()` from `part1.py`, `forward()` will call `get_kernel()` and `low_pass()` to create the low and high frequency images, and combine them into a hybrid image. Refer to this tutorial for additional information on defining neural networks using PyTorch.

You will compare the runtimes of your hybrid image implementations from Parts 1 and 2.

3 Understanding input/output shapes in PyTorch(30 points)

You will now implement `my_conv2d_pytorch()` in `part3.py` using the same 2D convolution operator from `torch.nn.functional` used in `low_pass()`.

Before we proceed, here are two quick definitions of terms we'll use often when describing convolution:

(i) **Stride**: When the stride is 1 then we move the filters one pixel at a time. When the stride is 2 (or uncommonly 3 or more, though this is rare in practice) then the filters jump 2 pixels at a time as we slide them around.

(ii) **Padding**: The amount of pixels added to an image when it is being convolved with a the kernel. Padding can help prevent an image from shrinking during the convolution operation.

Unlike `my_conv2d_numpy()` from `part1.py`, the shape of your output does not necessarily have to be the same as the input image. Instead, given an input image of shape $(1, d_1, h_1, w_1)$ and kernel of shape $(N, \frac{d_1}{g}, k, k)$, your output will be of shape $(1, d_2, h_2, w_2)$ where g is the number of groups, $d_2 = N$, $h_2 = \frac{h_1 - k + 2*p}{s} + 1$, and $w_2 = \frac{w_1 - k + 2*p}{s} + 1$ and p and s are padding and stride, respectively.

Think about why the equations for output width w_2 and output height h_2 are true - try sketching out a 5×5 grid, and seeing how many places you can place a 3×3 square within the grid with stride 1. What about with stride 2? Does your finding match what the equation states?

We demonstrate the effect of the value of the `groups` parameter on a simple example with an input image of shape (1,2, 3,3) and a kernel of shape (4,1,3,3):

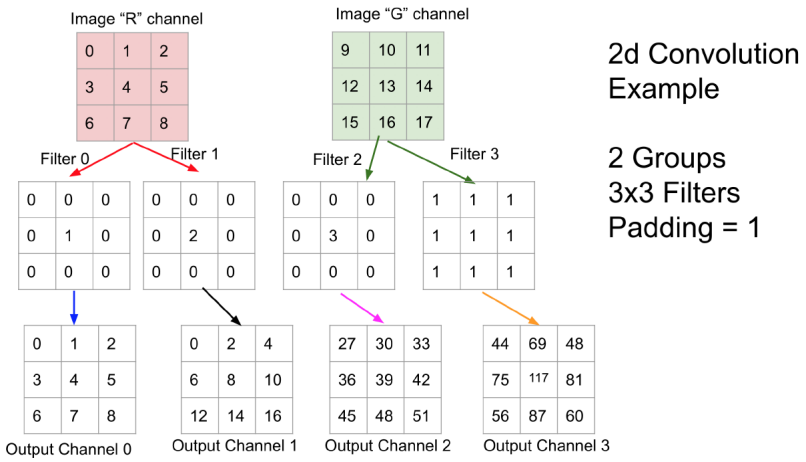

Figure 1: Visualization of a simple example using groups=2.

Figure 2: Look at the image from very close, then from far away.

Data

We provide you with 5 pairs of aligned images which can be merged reasonably well into hybrid images. The alignment is super important because it affects the perceptual grouping (read the paper for details). We encourage you to create additional examples (e.g., change of expression, morph between different objects, change over time, etc.).

For the example shown in Figure 2, the two original images look like this:

(a) Dog (b) Cat

Figure 3:

The low-pass (blurred) and high-pass version of these images look like this (Figure 4):

The high frequency image in Figure 4b is actually zero-mean with negative values, so it is visualized by adding 0.5. In the resulting visualization, bright values are positive and dark values are negative.

Adding the high and low frequencies together (Figures 4b and 4a, respectively) gives you the image in Figure 2. If you're having trouble seeing the multiple interpretations of the image, a useful way to visualize the effect is by progressively downsampling the hybrid image, as done in Figure 5. The starter code provides a function, `vis_image_scales_numpy()` in `utils.py`, which can be used to save and display such visualizations.

(a) Low frequencies of dog image.

(b) High frequencies of cat image.

Figure 4:

Figure 5:

Potentially useful NumPy (Python library) functions

`np.pad()`, which does many kinds of image padding for you, `np.clip()`, which "clips" out any values in an array outside of a specified range, `np.sum()` and `np.multiply()`, which makes it efficient to do the convolution (dot product) between the filter and windows of the image. Documentation for NumPy can be found here or by Googling the function in question.

Forbidden functions

(You can use these for testing, but not in your final code). Anything that takes care of the filter operation or creates a 2D Gaussian kernel directly for you is forbidden. If it feels like you're sidestepping the work, then it's probably not allowed.

Testing

We have provided a set of tests for you to evaluate your implementation. We have included tests inside `proj1.ipynb` so you can check your progress as you implement each section. When you're done with the entire project, you can call additional tests by running `pytest proj1_unit_tests` inside the root directory of the project, as well as checking against the tests on Gradescope.