NumPy and Image Processing Primer

Discussion #1

Written by:
Justin, Chung Min

Learning Objectives

Today, we will learn that images are just arrays – so it's time to learn how to use NumPy. You might be rusty with NumPy from that class you took years ago, or it's the first time you've really used it seriously. This time, ideally, using NumPy should start to feel like second nature.

Hopefully, by the end of the hour, you will feel comfortable with handling images in the form of arrays, and get to practice some practical coding knowledge for upcoming CS180/280A projects. If you feel comfortable with NumPy, feel free to look at the challenge questions uploaded online!

Logistics

1. Project 1 is out! due Fri. Sep. 12, 11:59PM.

1 Warmup!

1.1 `shape` and `dtype`

The two most important properties of arrays are their shape and dtype. A common terminology mixup is in referring to the mathematical concept of *dimension*, which refers to how big a given axis is, and the *shape* which refers to how many axes and how big each one is of an array.x2

Problem 1.1: What is the shape of a 3x3 matrix in NumPy? A 3-dimension row vector? Column vector?

What are the shape, dtype, values of the following one-liners:

(a) `np.full((10, 10, 3), 255.2)`

(b) `np.full((10, 10, 3), 255.2).astype(np.uint8)`

(c) `np.full((10, 10, 3), 256.0).astype(np.uint8)`

(d) `np.array([0.1, 0.2, 0.3]).astype(np.uint8)`

Note: casting from float to uint8 is the source of lots of hidden project bugs, be careful!

1.2 `stack` vs `concatenate`

Joining arrays together is usually done with these helper functions, let's remind ourselves how they work.

Problem 1.2: Suppose I have `img1` and `img2` which are both shape `(100,100,3)`. What shape do the following evaluate to?

(a) `np.stack([img1, img2], axis=0)`

(b) `np.stack([img1, img2], axis=1)`

(c) `np.concatenate([img1, img2], axis=0)`

(d) `np.concatenate([img1, img2], axis=1)`

1.3 Slicing

Slicing is how we obtain smaller chunks of arrays from a larger one, including for manipulating their values. Slicing can also be done with indices or mask arrays for logical operations!

Problem 1.3: What's the difference between `arr[0]` and `arr[0:1]` for a 2D array?

Problem 1.4: How would you set all elements in the first row of a 2D array to zero?

Problem 1.5: Given a floating point array `arr`, how do you set all values above 1 to exactly 1.0?

1.4 Singleton dimensions

One confusing aspect of arrays is that data can have dimensions of size `1` (for example depth images!), which are very important for downstream ops.

Problem 1.6: What's the shape of `np.array([1,2,3])`? `np.array([[1],[2],[3]])`?

Problem 1.7: What happens when you use `np.squeeze()` on an array of shape `(1,5,1,3)`?

Problem 1.8: How do you add a singleton dimension at `axis=0` to an array of shape `(4,4)`?

Problem 1.9: Given `arr = np.array([[[1,2,3]]])`, what's its shape? What is `np.squeeze(arr)`?

Problem 1.10: How could you replicate `np.stack([arr1, arr2], axis=1)` using only `np.expand_dims()` and `np.concatenate()`?

2 Pixels!

Typically, an image will be represented as a shape `(height, width, rgb)` array. These are typically abbreviated `(h, w, c)` (`c` for channel). There are two datatypes commonly used: `uint8` with values between 0 and 255, and `float` with values between 0.0 and 1.0. Some libraries like OpenCV default to BGR channel ordering, which will make your images look funny if you visualize with another library! Another convention is *channel-first* (CHW) vs *channel-last* (HWC).

2.1 Convention Confusion

Nobody likes these convention headaches, but they're important to be comfortable with to get to the fun part of CV! Suppose I start from a `uint8 img` with channel-last BGR conventions.

Problem 2.1: What is the shape of `img`?

Problem 2.2: How would I set all the *red* components of the pixels to 0?

Problem 2.3: How would I shift from a BGR to RGB format? (Hint: use `np.flip`)

Problem 2.4: How would I convert from channel-last to channel-first? (Hint: use `np.transpose`

Problem 2.5: How would I convert from uint8 format to float? (Be careful of the ranges!)

Note: it may seem pedantic, but what you just did is a common pattern for converting from OpenCV conventions to PyTorch conventions for machine learning applications.

2.2 Beautiful Batches

When working with multiple images, it's very common to use an initial dimension called a "batch" dimension. Let's suppose I have a list of 100 `float` images in HWC, RGB ordering called `img_list`.

Problem 2.6: Convert the list into a batch

Problem 2.7: Create a `minibatch` of the last 32 images.

Problem 2.8: Convert this minibatch to channel-first ordering (batch dim always stays first!)

2.3 Video Motion Trail

Videos are just batches of images! I have a video of my cat chasing its tail called `video` with shape `(60,1080,1920,3) (T, H, W, RGB)`.

Problem 2.9: I would like to create an output image which blends the frames to create a motion trail of this. What's a one-liner in NumPy to do this?

3 Will it broadcast?

When arrays don't have the same shape, NumPy will *implicitly* adapt the shapes of the inputs to allow arithmetic operations, this is called shape *broadcasting*. See https://numpy.org/doc/stable/user/basics.broadcasting.html for the official documentation around this behavior. The main thing to remember is that broadcasting goes from right to left along axes, and singleton axes will automatically be repeated to match the shape of the other operand at that axis.

For this problem, `c=np.arange(3).reshape((3,1))`, `r=np.arange(3).reshape((1,3))`, and `X=np.ones((3,3))`.

Problem 3.1: What are the shape and values of the following operations?

(a) `c + r`:

(b) `X*c`:

(c) `X*r`:

(d) `X + r`:

(e) `r + X`:

*Remember: the `*` operator is **not** a matrix multiply!*

Problem 3.2: Which of the following expressions throw an error for the indicated shape arrays? For the ones that work, what are their shapes?

(a) `(2, 3) + (3,)`

(b) `(2, 3) + (3, 2)`

(c) `(2, 3) + (2, 3, 1)`

(d) `(2, 3) + (2, 1, 1)`

Problem 3.3: Standardization. Use what you know about broadcasting to scale the RGB values of this batch by subtracting `mean = [0.485, 0.456, 0.406]` and dividing by `std = [0.229, 0.224, 0.225]`. (Remember the channel ordering is CHW!)

Note: This per-channel scaling is ImageNet standardization (maintaining variance of 1) applied by many neural networks today!

4 Vectorization Nation

The power of NumPy lies in its ability to perform operations on entire arrays at once, eliminating the need for explicit loops. This is called *vectorization*, and it's *much* faster than writing nested for loops. Let's practice converting loop-heavy code into vectorized operations.

Problem 4.1: Grayscale Conversion

```
# img is a numpy array with shape (h, w, 3) - RGB image
# out is a numpy array with shape (h, w) - grayscale output
h, w = img.shape[:2]
for i in range(h):
    for j in range(w):
        out[i, j] = 0.299 * img[i, j, 0] + 0.587 * img[i, j, 1] + 0.114 * img[i, j, 2]

print(out)
```

Please rewrite without for loops:

Fun fact: These weights (0.299, 0.587, 0.114) aren't arbitrary! They're based on human visual perception: our eyes are most sensitive to green, less to red, and least to blue. Simply averaging RGB channels equally would produce a grayscale image that looks "wrong" to us.

Problem 4.2: Pixel Neighbor Averaging

```
# img is a numpy array with shape (h, w)
# out is a numpy array with shape (h, w)
h, w = img.shape
for i in range(1, h-1): # Note the ranges
    for j in range(1, w-1): # Note the ranges
        out[i, j] = (img[i-1, j] + img[i+1, j] + img[i, j-1] + img[i, j+1]) / 4.0

print(out)
```

Please rewrite without for loops:

Problem 4.3: Mean-Squared Image Difference

```
# a is a numpy array with shape (h, w)
# b is a numpy array with shape (h, w)
# out is a numpy array with shape (h, w)
h, w = a.shape
mse = 0
for i in range(h):
    for j in range(w):
        mse += (a[i, j] - b[i, j]) ** 2
mse = mse/(h*w)

print(mse)
```

Please rewrite without for loops: