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?

A 3x3 matrix has shape (3, 3). A 3-element row vector has shape (1, 3). A 3-element column vector has shape (3, 1).

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

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

Shape: (10, 10, 3), dtype: float64, values: all 255.2

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

Shape: (10, 10, 3), dtype: uint8, values: all 255. The float 255.2 gets truncated to 255 when cast to uint8.

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

Shape: (10, 10, 3), dtype: uint8, values: all 0. Since uint8 range is 0-255, 256 overflows and wraps to 0.

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

Shape: (3,), dtype: uint8, values: [0, 0, 0]. Floats between 0 and 1 get truncated to 0 when cast to 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)`

(2, 100, 100, 3) - Creates a new dimension at axis 0, stacking the images.

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

(100, 2, 100, 3) - Creates a new dimension at axis 1, between height and width.

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

(200, 100, 3) - Joins along axis 0, doubling the height (stacking vertically).

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

(100, 200, 3) - Joins along axis 1, doubling the width (placing side by side).

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?

`arr[0]` returns a 1D array (the first row), while `arr[0:1]` returns a 2D array with shape (1, width) - preserves the row dimension.

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

`arr[0, :]  = 0` or `arr[0] = 0`

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

`arr[arr > 1] = 1.0` - Uses boolean indexing to select and modify elements.

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]])`?

`np.array([1,2,3])` has shape (3,) - 1D array. `np.array([[1],[2],[3]])` has shape (3,1) - 2D column vector.

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

Returns shape (5,3) - removes all singleton dimensions (size 1).

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

`np.expand_dims(arr, axis=0)` or `arr[np.newaxis, :, :]` - both create shape (1,4,4).

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

Shape is (1,1,3). `np.squeeze(arr)` returns shape (3,) - array([1,2,3]).

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

`np.concatenate([np.expand_dims(arr1, axis=1), np.expand_dims(arr2, axis=1)], axis=1)`
- First add singleton dimensions at axis 1, then concatenate along that axis.

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`?

(H, W, 3) - Height $\times$ Width $\times$ 3 channels (BGR)

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

`img[:, :, 2] = 0` - Red is channel 2 in BGR format

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

`img_rgb = np.flip(img, axis=2)` - Flips channels: BGR $\rightarrow$ RGB

Problem 2.4: How would I convert from channel-last to channel-first? (Hint: use `np.transpose` `img_chf = img.transpose(2, 0, 1)` - (H,W,C) $\rightarrow$ (C,H,W)

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

`img_float = img.astype(np.float32) / 255` - Converts [0,255] $\rightarrow$ [0,1]. It's important to cast to float32 *first* to avoid truncation! (Alternatively, you don't need to cast if dividing by a float (255.**0**)).

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

`batch = np.stack(img_list, axis=0)` - Creates (100, H, W, 3)

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

`minibatch = batch[-32:]` - Last 32 images from the batch

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

`minibatch_chf = minibatch.transpose(0, 3, 1, 2)` - (32,H,W,3) $\rightarrow$ (32,3,H,W)

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?

`motion_trail = np.mean(video, axis=0)` - Averages across time dimension

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`:

Shape: (3,3), Values: [[0,1,2], [1,2,3], [2,3,4]] - Each element of c added to each element of r.

(b) `X*c`:

Shape: (3,3), Values: [[0,0,0], [1,1,1], [2,2,2]] - Each row of X multiplied by corresponding element of c.

(c) `X*r`:

Shape: (3,3), Values: [[0,1,2], [0,1,2], [0,1,2]] - Each column of X multiplied by corresponding element of r.

(d) `X + r`:

Shape: (3,3), Values: [[1,2,3], [1,2,3], [1,2,3]] - r broadcasted across all rows of X (since X is all ones).

(e) `r + X`:

Shape: (3,3), Values: [[1,2,3], [1,2,3], [1,2,3]] - Same as X + r due to commutativity of addition.

*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,)`

Works. Shape: (2, 3) - The (3,) broadcasts to each row of the (2,3) array.

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

Error. Incompatible shapes - neither dimension matches and neither has size 1.

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

Error. Aligning from right: (2,3) $\rightarrow$ (1,2,3) vs (2,3,1). The middle dimensions (2 vs 3) are incompatible - neither is 1 and they don't match.

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

Works. Shape: (2, 2, 3) - The (2,3) becomes (1,2,3) and broadcasts with (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!)

`normalized = (minibatch_chf - mean[:, None, None]) / std[:, None, None]` - Broadcasting across spatial dims

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:

```
out = 0.299 * img[:, :, 0] + 0.587 * img[:, :, 1] + 0.114 * img[:, :, 2]
```

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:

```
out[1:-1, 1:-1] = (img[0:-2, 1:-1] + img[2:, 1:-1] + img[1:-1, 0:-2] + img[1:-1,
2:])  / 4.0
```

We approach this problem by "shifting" the array around, then averaging the shifted slices, which is equivalent to averaging the neighbors. The slicing extracts all 4 neighbors at once: `img[0:-2, 1:-1]` gets pixels above, `img[2:, 1:-1]` gets pixels below, `img[1:-1, 0:-2]` gets left neighbors, and `img[1:-1, 2:]` gets right neighbors.

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:

```
mse = np.mean((a - b) ** 2)
```

5 Bonus

5.1 Shape Shifting

Remember that NumPy arrays are by default *row-major*! This means that if you flatten an array into 1 axis, the index of the resulting array will change fastest in the rightmost index, and so on. Think of it like an n-dimension nested for loop, where the inner loop corresponds to the last axis.

Problem 5.1: What does `np.array([[1,2,3],[4,5,6]]).flatten()` print?

[1 2 3 4 5 6] - Flattens in row-major order (rightmost index changes fastest).

Problem 5.2: Given `arr = np.arange(6).reshape(2,3)`, what does `arr[:, 0]` return? `arr[0, :]`?

`arr[:,0]` returns [0 3] (first column). `arr[0,:]` returns [0 1 2] (first row).

Problem 5.3: `seq` has shape `(6,)` laid out like `[0,0,0,1,1,1]`. I'd like to convert it to a `(3,2)` array where the column vectors are labeled 0 or 1.

(i) Why won't naively reshaping work?

(ii) What's the correct way to convert it? (`transpose()` will be useful)

(i) Naive `seq.reshape(3,2)` gives [[0,0],[0,1],[1,1]] - each row mixes labels. We want columns to be pure 0s or 1s. **(ii)** `seq.reshape(2,3).transpose()` - First make (2,3) with each row being one label: [[0,0,0],[1,1,1]], then transpose to get columns as labels: [[0,1],[0,1],[0,1]].

Problem 5.4: `img` has shape `(6,6)` broken into four quadrants like `img[:3,:3]=0; img[:3,3:]=1; img[3:,:3]=2; img[3:,3:]=3`. Convert this into an array `tiles` of shape `(2,2,3,3)` where each of `tiles[i,j,...]` returns all of one number and `tiles[0,0]` returns all 0.

`img.reshape(2,3,2,3).transpose(0,2,1,3)` - First reshape to (2,3,2,3) which creates 2×2 blocks of 3×3 tiles, but in wrong order. Then transpose axes (0,2,1,3) to group the spatial dimensions correctly: tiles[0,0] gets top-left quadrant, tiles[0,1] gets top-right, etc.

5.2 `einsum`

`einsum` is a magical operation that folds large tensor operations into one specification, named for notation invented by Albert Einstein! It allows complicated reshape, dot-product, and reduction operations to be specified in a string of characters which are executed very efficiently. It takes as input N arrays and outputs one. The core principle is to separate operations into 1) which axes to *iterate* over, 2) which axes to *multiply* together in the output, and 3) which axes to *reduce* (sum) over.

This is captured by a string of the form `‘[arr1_indices],[arr2_indices]->[output_indices]`. Any indices missing in the output will be implicitly summed together, and any indices which *match* between `arr1` and `arr2` will have their values element-wise multiplied (their dimensions must match!)

For example, suppose we have a row vector r, column vector c, and square matrix X.

1. The outer product of r and c would be: `einsum(‘ij,ki -> jk’, r, c) or equivalently einsum(‘ij,jk->ik’, c, r)`
2. The matrix product $X \times X$ would be `einsum(‘ij,jk->ik’,X,X)`
3. Batched multiplication can be specified with `...` before or after the notation: for example `‘...ij,...jk->...ik’` means “I don’t care what the shapes are before the last 2 axes, just treat the last 2 axes as a matrix and multiply them”. In this case they must be broadcastable together!
4. You can use einsum to reduce a *single* axis of an array: `einsum(‘ij->i’,X)` is equivalent to `np.sum(X, axis=1)`

Problem B.0: `einsum` basics Suppose I had r and c from the example above but they were both shape `(3,)`.

(a): How could I compute the outer product without broadcasting?

`np.einsum(‘i,j->ij’, r, c)` - Creates a 2D outer product matrix where each element is r[i] * c[j].

(b): What about the dot product?

`np.einsum(‘i,i->’, r, c)` - Sums over matching indices to compute r[0]*c[0] + r[1]*c[1] + r[2]*c[2].

(c): What about the Hadamard (element-wise) product?

`np.einsum(‘i,i->i’, r, c)` - Element-wise multiplication without summing, returns array [r[0]*c[0], r[1]*c[1], r[2]*c[2]].

Problem B.1: Multi-headed attention

You may have heard of the transformer neural network architecture (the T in ChatGPT). A core operation inside this network is called multi-headed attention. One fundamental operation here is to multiply two big tensors `Q` and `K` together in a specific way. Initially, their shapes are `B, L, H, D`, and we would like to create an `LxL` matrix taking the outer-product of each element in Q to K along the L dimension. The resulting shape is `B, L, L, H`, where the D dimension disappears since it is involved in the dot product. Write this operation in one line with `einsum`.

`np.einsum('blhd,bkhd->blkh', Q, K)` - Performs dot product along D dimension between each position l in Q and position k in K, keeping batch B and head H dimensions intact.

Problem B.2: Vector-quantization

Suppose I have a set of vectors called `codes` of shape `(N, D)`, and a vector of `examples` of shape `(E,D)`. I would like to compute the nearest neighbor vector in `codes` for each vector in `examples`.

(a) First, find the all-pairs dot product similarities between `codes` and `examples`.

`similarities = np.einsum('nd,ed->ne', codes, examples)` - Computes dot product between each code vector (N) and each example vector (E), resulting in (N,E) similarity matrix.

(b) Next, use this to compute the nearest code ID for each example. (Hint: use `np.argmax` to find the most similar for each)

`nearest_codes = np.argmax(similarities, axis=0)` - For each example (column), finds the code (row) with highest similarity.

(c) What if I wanted to use L2 distance instead of dot product? (Hint: this is cumbersome to do with `einsum`, there's an easier way without it)

`distances = np.linalg.norm(codes[:, None, :]  - examples[None, :, :], axis=2)` then `np.argmin(distances, axis=0)` - Broadcasting creates (N,E,D), compute L2 norm along D, then find minimum distance for each example.

5.3 `einops`

A wonderful library for managing the shapes of arrays is `einops`, which provides the function `rearrange` which can be used to manipulate array shapes with strings! For example, shuffling HWC to CHW can be done with `rearrange(img, 'height width c -> c height width')`. This can make debugging code much easier, since it is essentially self-commenting. You can also reshape/-expand axes by grouping them with `(...)` for example to stack all video frames into one big batch dimension one could do `rearrange(video, 'B T H W C -> (B T) H W C')`. You can also go the other direction by specifying the values of these shapes as input to rearrange: `rearrange(batch, '(B T) H W C -> B T H W C', B=32, T=100)` (`einops` will throw an error if these shapes don't work out).

Problem B.3: Let's do Problem 5.4 again, but with `einops`! Convert an image `img` of shape `(B, C, H, W)` into tiles `(B, NH x NW, C, H', W')` where `H = NH x H'` and `W = NW x W'`.

`rearrange(img, 'B C (NH H_prime) (NW W_prime) -> B (NH NW) C H_prime W_prime', NH=NH, NW=NW)` - Decomposes spatial dimensions into tiles and flattens tile positions.

Problem B.4: Suppose I have an image pyramid of shape `(4,200,200,3)`. How can I rearrange this into a 400x400 image for visualization?

`rearrange(img, '(n m) H W C -> (n H) (m W) C', n=2, m=2)` - Arranges 4 images in a 2×2 grid, combining them into a single 400×400 visualization.

5.4 Vectorization challenges

Problem B.5: Image Downsampling

Downsampling reduces image resolution by averaging pixels. Convert the following unvectorized code that downsamples by averaging 2x2 blocks:

```python
# img is a numpy array with shape (h, w) where h and w are even
# out is a numpy array with shape (h//2, w//2)
h, w = img.shape
h_new, w_new = h // 2, w // 2
out = np.zeros((h_new, w_new))

for i in range(h_new):
    for j in range(w_new):
        # Average 2x2 blocks
        out[i, j] = (img[2*i, 2*j] + img[2*i, 2*j+1] +
                     img[2*i+1, 2*j] + img[2*i+1, 2*j+1]) / 4.0

print(out.shape)  # Should be (h//2, w//2)
```

Please rewrite without for loops using array slicing operations:

Hint: `arr[0::2, 0::2]` extracts every other element starting from (0,0).

```
out = (img[0::2, 0::2] + img[0::2, 1::2] + img[1::2, 0::2] + img[1::2, 1::2]) /
4.0
```

This extracts the four corners of each 2×2 block: top-left `[0::2, 0::2]`, top-right `[0::2, 1::2]`, bottom-left `[1::2, 0::2]`, and bottom-right `[1::2, 1::2]`, then averages them.