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?

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

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)

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.

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?

(b): What about the dot product?

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

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`.

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`.

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

(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)

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'`.

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?

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:

```
# 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).