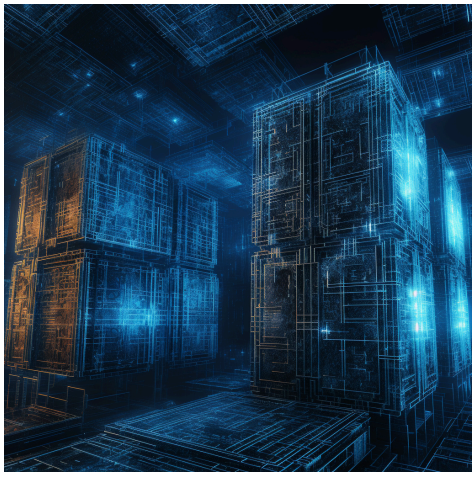
[0.0] - Prerequisites

Colab: exercises | solutions

ARENA 3.0 Streamlit page

Please send any problems / bugs on the `#errata` channel in the Slack group, and ask any questions on the dedicated channels for this chapter of material.



This notebook contains important prerequisites for the rest of the material in this chapter.

The first section 1️⃣ **Core Concepts / Knowledge** is a long list of important concepts and libraries that you should be familiar with before starting the rest of the material.

The second section 2️⃣ **Einops, Einsum & Tensor Manipulation** provides some coding exercises to help you get familiar with the `einops` and `einsum` libraries, as well as some exercises on indexing and other aspects of tensor manipulation. This section contains a large number of exercises. You should generally feel free to skim through them (skipping some or most) if you feel comfortable with the basic ideas.

Setup (don't read, just run!)

[ ] ↳ *2 cells hidden*

1️⃣ Core Concepts / Knowledge

↳ *1 cell hidden*

2️⃣ Einops, Einsum & Tensor Manipulation

> ### Learning objectives
>
> - Understand the basics of Einstein summation convention
> - Learn how to use `einops` to perform basic tensor rearrangement, and `einsum` to to perform standard linear algebra operations on tensors

Note - this section contains a large number of exercises. You should generally feel free to skim through them (skipping some or most) if you feel comfortable with the basic ideas.

Reading

- Read about the benefits of the `einops` library here.
- If you haven't already, then review the Einops basics tutorial (up to the "fancy examples" section).
- Read einsum is all you need) for a brief overview of the `einsum` function and how it works. (You don't need to read past section 2.10.)

Einops

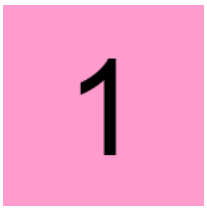
```
arr = np.load(section_dir / "numbers.npy")
```

`arr` is a 4D numpy array. The first axes corresponds to the number, and the next three axes are channels (i.e. RGB), height and width respectively. You have the function `utils.display_array_as_img` which takes in a numpy array and displays it as an image. There are two possible ways this function can be run:

- If the input is three-dimensional, the dimensions are interpreted as `(channel, height, width)` - in other words, as an RGB image.
- If the input is two-dimensional, the dimensions are interpreted as `(height, width)` - i.e. a monochrome image.

For example:

```
display_array_as_img(arr[1])
```



A series of images follow below, which have been created using `einops` functions performed on `arr`. You should work through these and try to produce each of the images yourself. This page also includes solutions, but you should only look at them after you've tried for at least five minutes.

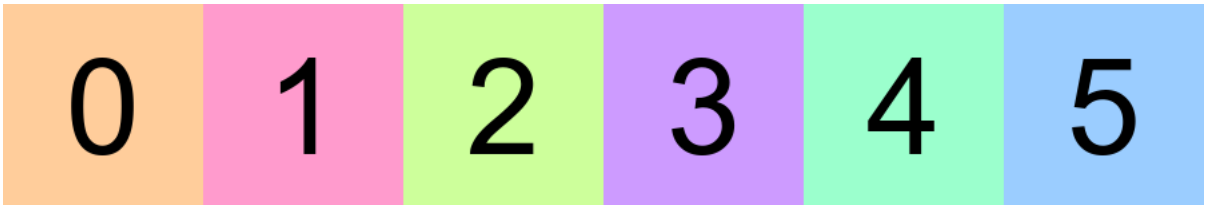
> *Note - if you find you're comfortable with the first ~half of these, you can skip to later sections if you'd prefer, since these aren't particularly conceptually important.*

Einops exercises - images

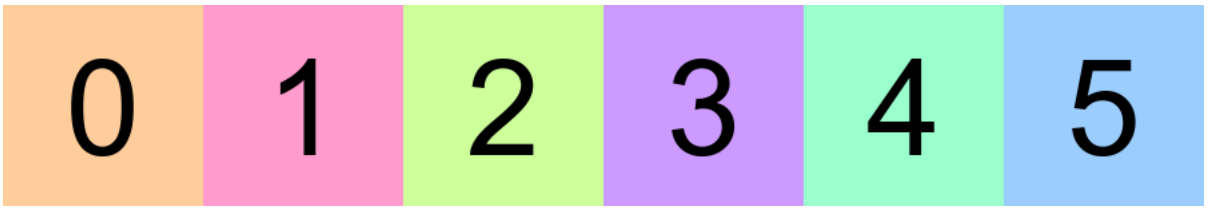
```
Difficulty: 🔴🔴⚪⚪⚪
Importance: 🔵🔵🔵⚪⚪

You should spend up to ~45 minutes on these exercises collectively.

If you think you get the general idea, then you can skip to the next section.
```

Exercise 1

```
display_soln_array_as_img(1)
```




```
# Your code here - define arr1
arr1 = einops.rearrange(arr, "b c h w -> c h (b w)")
display_array_as_img(arr1)
```



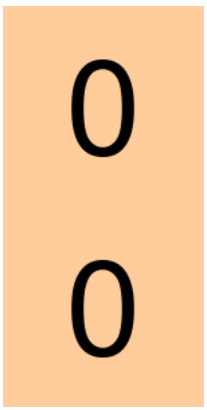
▶ Details

Solution

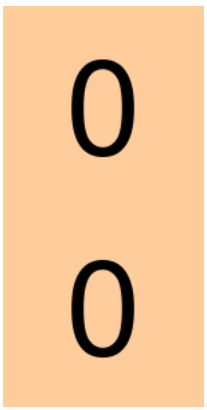
```
arr1 = einops.rearrange(arr, "b c h w -> c h (b w)")
```

Exercise 2

```
display_soln_array_as_img(2)
```



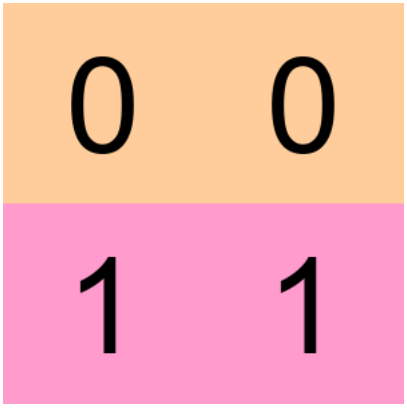

```
# Your code here - define arr2
arr2 = einops.repeat(arr[0], 'c h w -> c (2 h) w')
display_array_as_img(arr2)
```



▶ Solution

Exercise 3

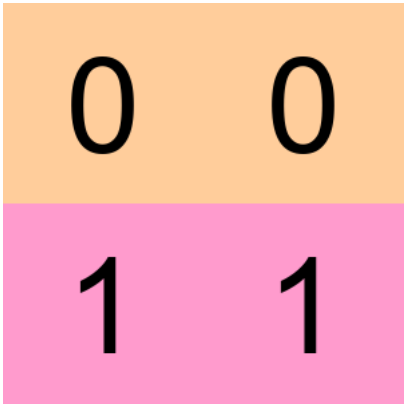
```
display_soln_array_as_img(3)
```




```
arr[:2].shape
```

```
(2, 3, 150, 150)
```

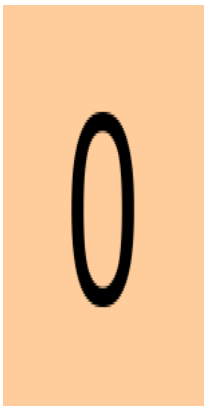
```
# Your code here - define arr3
arr3 = einops.repeat(arr[:2], 'b c h w -> c (b h) (2 w)')
display_array_as_img(arr3)
```



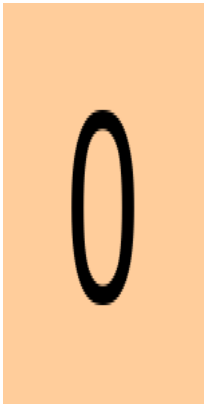
▶ Solution

Exercise 4

```
display_soln_array_as_img(4)
```




```
# Your code here - define arr4
arr4 = einops.repeat(arr[0], 'c h w -> c (h 2) w')
display_array_as_img(arr4)
```



▶ Solution

Exercise 5

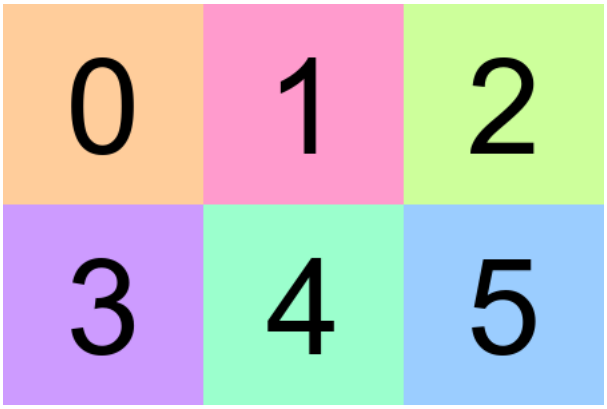
```
display_soln_array_as_img(5)
```


```
# Your code here - define arr5
arr5 = einops.rearrange(arr[0], "c h w -> h (c w)")
display_array_as_img(arr5)
```

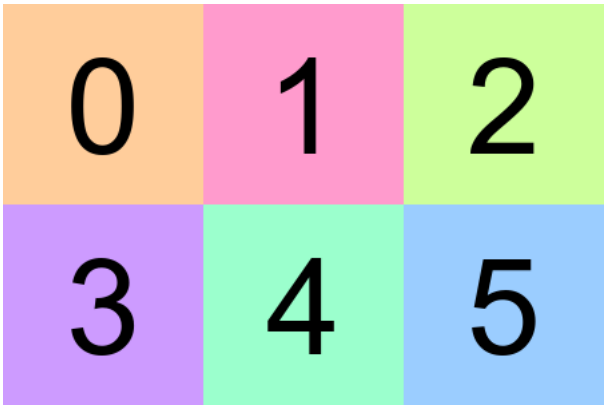
▶ Solution

Exercise 6

```
display_soln_array_as_img(6)
```




```
# Your code here - define arr6
arr6 = einops.repeat(arr, "(b1 b2) c h w -> c (b1 h) (b2 w)", b1=2)
display_array_as_img(arr6)
```



▶ Solution

Exercise 7

```
display_soln_array_as_img(7)
```


```
# Your code here - define arr7
arr7 = einops.reduce(arr, 'b c h w -> h (b w)', 'max')
display_array_as_img(arr7)
```

▶ Solution

Exercise 8

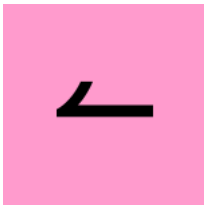
```
display_soln_array_as_img(8)
```

```
# Your code here - define arr8
arr8 = einops.reduce(arr, 'b c h w -> h w', 'min')
display_array_as_img(arr8)
```

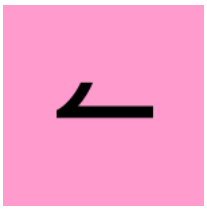
▶ Solution

Exercise 9

```
display_soln_array_as_img(9)
```



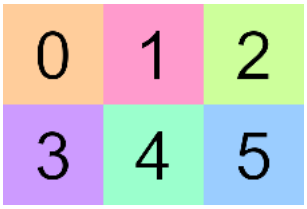
```
# Your code here - define arr9
arr9 = einops.rearrange(arr[1], 'c h w -> c w h')
display_array_as_img(arr9)
```



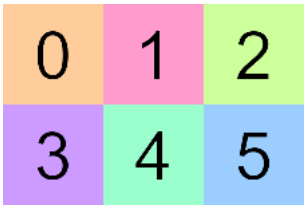
▶ Solution

Exercise 10

```
display_soln_array_as_img(10)
```




```
# Your code here - define arr10
arr10 = einops.reduce(arr, '(b1 b2) c (h h2) (w w2) -> c (b1 h) (b2 w)', 'max', b1=2, h2=2, w2=2)
display_array_as_img(arr10)
```



▶ Solution

Einops exercises - operations

Next, we have a series of functions which you should implement using `einops`. In the dropdown below these exercises, you can find solutions to all of them.

First, let's define some functions to help us test our solutions:

```
import torch as t


def assert_all_equal(actual: t.Tensor, expected: t.Tensor) -> None:
    assert actual.shape == expected.shape, f"Shape mismatch, got: {actual.shape}"
    assert (actual == expected).all(), f"Value mismatch, got: {actual}"
    print("Passed!")

def assert_all_close(actual: t.Tensor, expected: t.Tensor, rtol=1e-05, atol=0.0001) -> None:
    assert actual.shape == expected.shape, f"Shape mismatch, got: {actual.shape}"
    assert t.allclose(actual, expected, rtol=rtol, atol=atol)
    print("Passed!")
```

Exercise A.1 - rearrange (1)

```
def rearrange_1() -> t.Tensor:
    '''Return the following tensor using only torch.arange and einops.rearrange:

    [[3, 4],
     [5, 6],
     [7, 8]]
    '''
    return einops.rearrange(t.arange(3, 9), '(b1 b2) -> b1 b2', b2=2)

expected = t.tensor([[3, 4], [5, 6], [7, 8]])
assert_all_equal(rearrange_1(), expected)
```

```
Passed!
```

▶ Solution

Exercise A.2 - rearrange (2)

```
def rearrange_2() -> t.Tensor:
    '''Return the following tensor using only torch.arange and einops.rearrange:

    [[1, 2, 3],
```

```
     [4, 5, 6]]
    '''
    return einops.rearrange(t.arange(1, 7), '(b1 b2) -> b1 b2', b1=2)


assert_all_equal(rearrange_2(), t.tensor([[1, 2, 3], [4, 5, 6]]))
```

Passed!

▶ Solution

Exercise A.3 - rearrange (3)

```
def rearrange_3() -> t.Tensor:
    '''Return the following tensor using only torch.arange and einops.rearrange:

    [[[1], [2], [3], [4], [5], [6]]]
    '''
    return einops.rearrange(t.arange(1, 7), '(b1 b2) -> 1 b2 b1', b2=6)


assert_all_equal(rearrange_3(), t.tensor([[[1], [2], [3], [4], [5], [6]]]))
```

Passed!

▶ Solution

Exercise B.1 - temperature average

```
def temperatures_average(temps: t.Tensor) -> t.Tensor:
    '''Return the average temperature for each week.

    temps: a 1D temperature containing temperatures for each day.
    Length will be a multiple of 7 and the first 7 days are for the first week, second 7 days for the second week, etc.

    You can do this with a single call to reduce.
    '''
    assert len(temps) % 7 == 0
    return einops.reduce(temps, '(b1 b2) -> b1', 'mean', b2=7)


temps = t.Tensor([71, 72, 70, 75, 71, 72, 70, 68, 65, 60, 68, 60, 55, 59, 75, 80, 85, 80, 78, 72, 83])
expected = t.tensor([71.5714, 62.1429, 79.0])
assert_all_close(temperatures_average(temps), expected)
```

Passed!

▶ Solution

Exercise B.2 - temperature difference

```
def temperatures_differences(temps: t.Tensor) -> t.Tensor:
    '''For each day, subtract the average for the week the day belongs to.

    temps: as above
    '''
    assert len(temps) % 7 == 0
    return temps - einops.repeat(temperatures_average(temps), "w -> (w 7)")


expected = t.tensor(
    [
        -0.5714,
        0.4286,
        -1.5714,
        3.4286,
        -0.5714,
        0.4286,
        -1.5714,
```

```
        5.8571,
        2.8571,
        -2.1429,
        5.8571,
        -2.1429,
        -7.1429,
        -3.1429,
        -4.0,
        1.0,
        6.0,
        1.0,
        -1.0,
        -7.0,
        4.0,
    ]
)
actual = temperatures_differences(temps)
assert_all_close(actual, expected)
```

```
Passed!
```

▶ Solution

Exercise B.3 - temperature normalized

```
def temperatures_normalized(temps: t.Tensor) -> t.Tensor:
    '''For each day, subtract the weekly average and divide by the weekly standard deviation.

    temps: as above

    Pass torch.std to reduce.
    '''
    return temperatures_differences(temps) / einops.repeat(einops.reduce(temps, "(w 7) -> w", t.std), 'w -> (w 7)')


expected = t.tensor(
    [
        -0.3326,
        0.2494,
        -0.9146,
        1.9954,
        -0.3326,
        0.2494,
        -0.9146,
        1.1839,
        0.5775,
        -0.4331,
        1.1839,
        -0.4331,
        -1.4438,
        -0.6353,
        -0.8944,
        0.2236,
        1.3416,
        0.2236,
        -0.2236,
        -1.5652,
        0.8944,
    ]
)
actual = temperatures_normalized(temps)
assert_all_close(actual, expected)
```

```
Passed!
```

▶ Solution

Exercise C - identity matrix

```
def identity_matrix(n: int) -> t.Tensor:
    '''Return the identity matrix of size nxn.
```

```
    Don't use torch.eye or similar.

    Hint: you can do it with arange, rearrange, and ==.
    Bonus: find a different way to do it.
    '''
    assert n >= 0
    return t.arange(0, n) == einops.rearrange(t.arange(0, n), "(n 1) -> n 1")


assert_all_equal(identity_matrix(3), t.Tensor([[1, 0, 0], [0, 1, 0], [0, 0, 1]]))
assert_all_equal(identity_matrix(0), t.zeros((0, 0)))
```

```
Passed!
Passed!
```

▶ Solution

Exercise D - sample distribution

```
def sample_distribution(probs: t.Tensor, n: int) -> t.Tensor:
    '''Return n random samples from probs, where probs is a normalized probability distribution.

    probs: shape (k,) where probs[i] is the probability of event i occurring.
    n: number of random samples

    Return: shape (n,) where out[i] is an integer indicating which event was sampled.

    Use torch.rand and torch.cumsum to do this without any explicit loops.

    Note: if you think your solution is correct but the test is failing, try increasing the value of n.
    '''
    assert abs(probs.sum() - 1.0) < 0.001
    assert (probs >= 0).all()
    return (t.rand(n, 1) > t.cumsum(probs, dim=0)).sum(-1)
    # Intuition: For how many bins is the random sample larger than the cumsum == Event


n = 10000000
probs = t.tensor([0.05, 0.1, 0.1, 0.2, 0.15, 0.4])
freqs = t.bincount(sample_distribution(probs, n)) / n
assert_all_close(freqs, probs, rtol=0.001, atol=0.001)
```

```
Passed!
```

▶ Solution

Exercise E - classifier accuracy

```
def classifier_accuracy(scores: t.Tensor, true_classes: t.Tensor) -> t.Tensor:
    '''Return the fraction of inputs for which the maximum score corresponds to the true class for that input.

    scores: shape (batch, n_classes). A higher score[b, i] means that the classifier thinks class i is more likely.
    true_classes: shape (batch, ). true_classes[b] is an integer from [0...n_classes).

    Use torch.argmax.
    '''
    assert true_classes.max() < scores.shape[1]
    return (t.argmax(scores, dim=1) == true_classes).float().mean()


scores = t.tensor([[0.75, 0.5, 0.25], [0.1, 0.5, 0.4], [0.1, 0.7, 0.2]])
true_classes = t.tensor([0, 1, 0])
expected = 2.0 / 3.0
assert classifier_accuracy(scores, true_classes) == expected
```

▶ Solution

Exercise F.1 - total price indexing

```
def total_price_indexing(prices: t.Tensor, items: t.Tensor) -> float:
    '''Given prices for each kind of item and a tensor of items purchased, return the total price.

    prices: shape (k, ). prices[i] is the price of the ith item.
    items: shape (n, ). A 1D tensor where each value is an item index from [0..k).

    Use integer array indexing. The below document describes this for NumPy but it's the same in PyTorch:

    https://numpy.org/doc/stable/user/basics.indexing.html#integer-array-indexing
    '''
    assert items.max() < prices.shape[0]
    return prices[items].sum()


prices = t.tensor([0.5, 1, 1.5, 2, 2.5])
items = t.tensor([0, 0, 1, 1, 4, 3, 2])
assert total_price_indexing(prices, items) == 9.0
```

▶ Solution

Exercise F.2 - gather 2D

```
def gather_2d(matrix: t.Tensor, indexes: t.Tensor) -> t.Tensor:
    '''Perform a gather operation along the second dimension.

    matrix: shape (m, n)
    indexes: shape (m, k)

    Return: shape (m, k). out[i][j] = matrix[i][indexes[i][j]]

    For this problem, the test already passes and it's your job to write at least three asserts relating the arguments and the o

    See: https://pytorch.org/docs/stable/generated/torch.gather.html?highlight=gather#torch.gather
    '''
    out = matrix.gather(1, indexes)
    return out


matrix = t.arange(15).view(3, 5)
indexes = t.tensor([[4], [3], [2]])
expected = t.tensor([[4], [8], [12]])
assert_all_equal(gather_2d(matrix, indexes), expected)
indexes2 = t.tensor([[2, 4], [1, 3], [0, 2]])
expected2 = t.tensor([[2, 4], [6, 8], [10, 12]])
assert_all_equal(gather_2d(matrix, indexes2), expected2)
```

```
Passed!
Passed!
```

▶ Solution

Exercise F.3 - total price gather

```
def total_price_gather(prices: t.Tensor, items: t.Tensor) -> float:
    '''Compute the same as total_price_indexing, but use torch.gather.'''
    assert items.max() < prices.shape[0]
    return prices.gather(0, items).sum()


prices = t.tensor([0.5, 1, 1.5, 2, 2.5])
items = t.tensor([0, 0, 1, 1, 4, 3, 2])
assert total_price_gather(prices, items) == 9.0
```

▶ Solution

Exercise G - indexing

```
def integer_array_indexing(matrix: t.Tensor, coords: t.Tensor) -> t.Tensor:
    '''Return the values at each coordinate using integer array indexing.

    For details on integer array indexing, see:
    https://numpy.org/doc/stable/user/basics.indexing.html#integer-array-indexing

    matrix: shape (d_0, d_1, ..., d_n)
    coords: shape (batch, n)

    Return: (batch, )
    '''
    return matrix[tuple(coords.T)]


mat_2d = t.arange(15).view(3, 5)
coords_2d = t.tensor([[0, 1], [0, 4], [1, 4]])
actual = integer_array_indexing(mat_2d, coords_2d)
assert_all_equal(actual, t.tensor([1, 4, 9]))
mat_3d = t.arange(2 * 3 * 4).view((2, 3, 4))
coords_3d = t.tensor([[0, 0, 0], [0, 1, 1], [0, 2, 2], [1, 0, 3], [1, 2, 0]])
actual = integer_array_indexing(mat_3d, coords_3d)
assert_all_equal(actual, t.tensor([0, 5, 10, 15, 20]))
```

```
Passed!
Passed!
```

▶ Solution

Exercise H.1 - batched logsumexp

```
import math


def batched_logsumexp(matrix: t.Tensor) -> t.Tensor:
    '''For each row of the matrix, compute log(sum(exp(row))) in a numerically stable way.

    matrix: shape (batch, n)

    Return: (batch, ). For each i, out[i] = log(sum(exp(matrix[i]))).

    Do this without using PyTorch's logsumexp function.

    A couple useful blogs about this function:
    - https://leimao.github.io/blog/LogSumExp/
    - https://gregorygundersen.com/blog/2020/02/09/log-sum-exp/
    '''
    a = matrix.max(-1).values
    return a + t.log(t.sum(t.exp(matrix - einops.rearrange(a, "n -> n 1")), dim=-1))



matrix = t.tensor([[-1000, -1000, -1000, -1000], [1000, 1000, 1000, 1000]])
expected = t.tensor([-1000 + math.log(4), 1000 + math.log(4)])
actual = batched_logsumexp(matrix)
assert_all_close(actual, expected)
matrix2 = t.randn((10, 20))
expected2 = t.logsumexp(matrix2, dim=-1)
actual2 = batched_logsumexp(matrix2)
assert_all_close(actual2, expected2)
```

```
Passed!
Passed!
```

▶ Solution

Exercise H.2 - batched softmax

```
def batched_softmax(matrix: t.Tensor) -> t.Tensor:
    '''For each row of the matrix, compute softmax(row).

    Do this without using PyTorch's softmax function.
    Instead, use the definition of softmax: https://en.wikipedia.org/wiki/Softmax_function
```

```
    matrix: shape (batch, n)

    Return: (batch, n). For each i, out[i] should sum to 1.
    '''
    exp = matrix.exp()
    return exp / exp.sum(-1, keepdim=True)


matrix = t.arange(1, 6).view((1, 5)).float().log()
expected = t.arange(1, 6).view((1, 5)) / 15.0
actual = batched_softmax(matrix)
assert_all_close(actual, expected)
for i in [0.12, 3.4, -5, 6.7]:
    assert_all_close(actual, batched_softmax(matrix + i))
matrix2 = t.rand((10, 20))
actual2 = batched_softmax(matrix2)
assert actual2.min() >= 0.0
assert actual2.max() <= 1.0
assert_all_equal(actual2.argsort(), matrix2.argsort())
assert_all_close(actual2.sum(dim=-1), t.ones(matrix2.shape[:-1]))
```

```
Passed!
Passed!
Passed!
Passed!
Passed!
Passed!
Passed!
```

▶ Solution

Exercise H.3 - batched logsoftmax

```
def batched_logsoftmax(matrix: t.Tensor) -> t.Tensor:
    '''Compute log(softmax(row)) for each row of the matrix.

    matrix: shape (batch, n)

    Return: (batch, n). For each i, out[i] should sum to 1.

    Do this without using PyTorch's logsoftmax function.
    For each row, subtract the maximum first to avoid overflow if the row contains large values.
    '''
    max = matrix.max(-1, keepdim=True).values
    return matrix - max - t.log(t.exp(matrix - max).sum(-1, keepdim=True))


matrix = t.arange(1, 6).view((1, 5)).float()
start = 1000
matrix2 = t.arange(start + 1, start + 6).view((1, 5)).float()
actual = batched_logsoftmax(matrix2)
print(actual)
expected = t.tensor([[-4.4519, -3.4519, -2.4519, -1.4519, -0.4519]])
assert_all_close(actual, expected)
```

```
tensor([[-4.4519, -3.4519, -2.4519, -1.4519, -0.4519]])
Passed!
```

▶ Solution

Exercise H.4 - batched cross entropy loss

```
def batched_cross_entropy_loss(logits: t.Tensor, true_labels: t.Tensor) -> t.Tensor:
    '''Compute the cross entropy loss for each example in the batch.

    logits: shape (batch, classes). logits[i][j] is the unnormalized prediction for example i and class j.
    true_labels: shape (batch, ). true_labels[i] is an integer index representing the true class for example i.

    Return: shape (batch, ). out[i] is the loss for example i.

    Hint: convert the logits to log-probabilities using your batched_logsoftmax from above.
```

```
    Then the loss for an example is just the negative of the log-probability that the model assigned to the true class. Use torc
    '''
    return - einops.rearrange(
        batched_logsoftmax(logits).gather(
            1,
            einops.rearrange(
                true_labels,
                "n -> n 1"
            )
        ),
        "n 1 -> n"
    )


logits = t.tensor([[float("-inf"), float("-inf"), 0], [1 / 3, 1 / 3, 1 / 3], [float("-inf"), 0, 0]])
true_labels = t.tensor([2, 0, 0])
expected = t.tensor([0.0, math.log(3), float("inf")])
actual = batched_cross_entropy_loss(logits, true_labels)
assert_all_close(actual, expected)
```

Passed!

▶ Solution

Exercise I.1 - collect rows

```
def collect_rows(matrix: t.Tensor, row_indexes: t.Tensor) -> t.Tensor:
    '''Return a 2D matrix whose rows are taken from the input matrix in order according to row_indexes.

    matrix: shape (m, n)
    row_indexes: shape (k,). Each value is an integer in [0..m).

    Return: shape (k, n). out[i] is matrix[row_indexes[i]].
    '''
    assert row_indexes.max() < matrix.shape[0]
    return matrix[row_indexes,:]


matrix = t.arange(15).view((5, 3))
row_indexes = t.tensor([0, 2, 1, 0])
actual = collect_rows(matrix, row_indexes)
expected = t.tensor([[0, 1, 2], [6, 7, 8], [3, 4, 5], [0, 1, 2]])
assert_all_equal(actual, expected)
```

Passed!

▶ Solution

Exercise I.2 - collect columns

```
def collect_columns(matrix: t.Tensor, column_indexes: t.Tensor) -> t.Tensor:
    '''Return a 2D matrix whose columns are taken from the input matrix in order according to column_indexes.

    matrix: shape (m, n)
    column_indexes: shape (k,). Each value is an integer in [0..n).

    Return: shape (m, k). out[:, i] is matrix[:, column_indexes[i]].
    '''
    assert column_indexes.max() < matrix.shape[1]
    return matrix[:, column_indexes]


matrix = t.arange(15).view((5, 3))
column_indexes = t.tensor([0, 2, 1, 0])
actual = collect_columns(matrix, column_indexes)
expected = t.tensor([[0, 2, 1, 0], [3, 5, 4, 3], [6, 8, 7, 6], [9, 11, 10, 9], [12, 14, 13, 12]])
assert_all_equal(actual, expected)
```

Passed!

▶ Solution

Einsum

Einsum is a very useful function for performing linear operations, which you'll probably be using a lot during this programme.

> Note - we'll be using the `einops.einsum` version of the function, which works differently to the more conventional `torch.einsum`:
>
> - `einops.einsum` has the arrays as the first arguments, and uses spaces to separate dimensions in the string.
> - `torch.einsum` has the string as its first argument, and doesn't use spaces to separate dimensions (each dim is represented by a single character).
>
> For instance, `torch.einsum("ij,i->j", A, b)` is equivalent to `einops.einsum(A, b, "i j, i -> j")`. (Note, einops doesn't care whether there are spaces either side of `,` and `->`, so you don't need to match this syntax exactly.)

Although there are many different kinds of operations you can perform, they are all derived from three key rules:

1. Repeating letters in different inputs means those values will be multiplied, and those products will be in the output.
    - For example, `M = einops.einsum(A, B, "i j, i j -> i j")` just corresponds to the elementwise product `M = A * B` (because $M_{ij}=A_{ij}B_{ij}$).
2. Omitting a letter means that the axis will be summed over.
    - For example, if `x` is a 2D array with shape `(I, J)`, then `einops.einsum(x, "i j -> i")` will be a 1D array of length `I` containing the row sums of `x` (we're summing along the `j`-index, i.e. across rows).
3. We can return the unsummed axes in any order.
    - For example, `einops.einsum(x, "i j k -> k j i")` does the same thing as `einops.rearrange(x, "i j k -> k j i")`.

Note - the einops creators supposedly have plans to support shape rearrangement, e.g. with operations like `einops.einsum(x, y, "i j, j k l -> i (k l)")` (i.e. combining the features of rearrange and einsum), so we can all look forward to that day!

Einsum exercises

```
Difficulty: 🔴🔴⚪⚪⚪
Importance: 🔵🔵🔵🔵⚪

You should spend up to 15-20 minutes on these exercises collectively.

If you think you get the general idea, then you can skip to the next section.
```

In the following exercises, you'll write simple functions using `einsum` which replicate the functionality of standard NumPy functions: trace, matrix multiplication, inner and outer products. We've also included some test functions which you should run.

Note - this version of einsum will require that you include `->`, even if you're summing to a scalar (i.e. the right hand side of your string expression is empty).

```
def einsum_trace(mat: np.ndarray):
    '''
    Returns the same as `np.trace`.
    '''
    assert mat.shape[0] == mat.shape[1]
    return einops.einsum(
        mat,
        np.identity(mat.shape[0]),
        "m m, m m -> "
    )


def einsum_mv(mat: np.ndarray, vec: np.ndarray):
    '''
    Returns the same as `np.matmul`, when `mat` is a 2D array and `vec` is 1D.
    '''
    assert mat.shape[1] == vec.shape[0], f"Shapes are not the same {mat.shape[1]} != {vec.shape[0]}"
    return einops.einsum(
        mat,
        vec,
        "m n, n -> m"
```

```
        )

    def einsum_mm(mat1: np.ndarray, mat2: np.ndarray):
        '''
        Returns the same as `np.matmul`, when `mat1` and `mat2` are both 2D arrays.
        '''
        assert mat1.shape[1] == mat2.shape[0], f"Shapes are not the same {mat1.shape[1]} != {mat2.shape[0]}"
        return einops.einsum(
            mat1,
            mat2,
            "m n, n k -> m k"
        )

    def einsum_inner(vec1: np.ndarray, vec2: np.ndarray):
        '''
        Returns the same as `np.inner`.
        '''
        assert vec1.shape[-1] == vec2.shape[-1], f"Shapes are not the same {vec1.shape[-1]} != {vec2.shape[-1]}"
        return einops.einsum(
            vec1,
            vec2,
            "... m, ... m -> ..."
        )

    def einsum_outer(vec1: np.ndarray, vec2: np.ndarray):
        '''
        Returns the same as `np.outer`.
        '''
        return einops.einsum(
            vec1,
            vec2,
            "i, j -> i j"
        )

    tests.test_einsum_trace(einsum_trace)
    tests.test_einsum_mv(einsum_mv)
    tests.test_einsum_mm(einsum_mm)
    tests.test_einsum_inner(einsum_inner)
    tests.test_einsum_outer(einsum_outer)
```

```
All tests in `test_einsum_trace` passed!
All tests in `test_einsum_mv` passed!
All tests in `test_einsum_mm` passed!
All tests in `test_einsum_inner` passed!
All tests in `test_einsum_outer` passed!
```