Manipulating Tensors

Curiculum

1. Tensors operations (basic)
2. Finding min, max, mean, sum etc (aggregation)
3. Positional min, max
4. Change tensor datatype
5. Tensor shape
6. Reshaping, stacking, squeezing and squeezing
7. Indexing
8. Rank, Axes, Shape
9. Reproducibility (make random out of random)

1. Tensors operations

1. Addition
2. Substraction
3. Multiplication
4. Division
5. Matrix multiplication

1.1 Addition, Subtraction, Multiplication

```
# Create a tensor of values and add a number to it
tensor = torch.tensor([1, 2, 3])
tensor + 10
>>>
tensor([11, 12, 13])
```

```
tensor * 10
>>>
tensor([10, 20, 30])
```

Notice how the tensor values above didn't end up being tensor([110, 120, 130]), this is because the values inside the tensor don't change unless they're reassigned. Let's subtract a number and this time we'll reassign the tensor variable

```
# Subtract and reassign
tensor = tensor - 10
tensor
>>>
tensor([-9, -8, -7])
```

```
# Add and reassign
tensor = tensor + 10
tensor
>>>
tensor([1, 2, 3])
```

1.1 Addition, Subtraction, Multiplication

PyTorch also has a bunch of built-in functions like torch.mul() (short for multiplcation) and torch.add() to perform basic operations.

```
# Can also use torch functions
torch.multiply(tensor, 10)
>>>
tensor([10, 20, 30])

# Original tensor is still unchanged
tensor
>>>
tensor([1, 2, 3])
```

However, it's more common to use the operator symbols like * instead of torch.mul()

```
# Element-wise multiplication (each element multiplies its equivalent, index 0->0, 1->1, 2->2)
print(tensor, "*", tensor)
print("Equals:", tensor * tensor)
```

1.2 Matrix multiplication

Pyorch implements matrix multiplication functionality in the torch.matmul() method.

The main two rules for matrix multiplication to remember are:

1. The **inner dimensions** must match:
   - `(3, 2) @ (3, 2)` won't work
   - `(2, 3) @ (3, 2)` will work
   - `(3, 2) @ (2, 3)` will work
2. The resulting matrix has the shape of the **outer dimensions**:
   - `(2, 3) @ (3, 2) -> (2, 2)`
   - `(3, 2) @ (2, 3) -> (3, 3)`

> **Note:** "`@`" in Python is the symbol for matrix multiplication.

all rule

all rules

```
>>> # vector x vector
>>> tensor1 = torch.randn(3)
>>> tensor2 = torch.randn(3)
>>> torch.matmul(tensor1, tensor2).size()
torch.Size([])
>>> # matrix x vector
>>> tensor1 = torch.randn(3, 4)
>>> tensor2 = torch.randn(4)
>>> torch.matmul(tensor1, tensor2).size()
torch.Size([3])
>>> # batched matrix x broadcasted vector
>>> tensor1 = torch.randn(10, 3, 4)
>>> tensor2 = torch.randn(4)
>>> torch.matmul(tensor1, tensor2).size()
torch.Size([10, 3])
>>> # batched matrix x batched matrix
>>> tensor1 = torch.randn(10, 3, 4)
>>> tensor2 = torch.randn(10, 4, 5)
>>> torch.matmul(tensor1, tensor2).size()
torch.Size([10, 3, 5])
>>> # batched matrix x broadcasted matrix
>>> tensor1 = torch.randn(10, 3, 4)
>>> tensor2 = torch.randn(4, 5)
>>> torch.matmul(tensor1, tensor2).size()
torch.Size([10, 3, 5])
```

1.2 Matrix multiplication

torch.matmul(*input*, *other*, *, *out=None*) → Tensor

Matrix product of two tensors.

The behavior depends on the dimensionality of the tensors as follows:

- If both tensors are 1-dimensional, the dot product (scalar) is returned.
- If both arguments are 2-dimensional, the matrix-matrix product is returned.
- If the first argument is 1-dimensional and the second argument is 2-dimensional, a 1 is prepended to its dimension for the purpose of the matrix multiply. After the matrix multiply, the prepended dimension is removed.
- If the first argument is 2-dimensional and the second argument is 1-dimensional, the matrix-vector product is returned.
- If both arguments are at least 1-dimensional and at least one argument is N-dimensional (where N > 2), then a batched matrix multiply is returned. If the first argument is 1-dimensional, a 1 is prepended to its dimension for the purpose of the batched matrix multiply and removed after. If the second argument is 1-dimensional, a 1 is appended to its dimension for the purpose of the batched matrix multiple and removed after. The non-matrix (i.e. batch) dimensions are broadcasted (and thus must be broadcastable). For example, if `input` is a $(j \times 1 \times n \times n)$ tensor and `other` is a $(k \times n \times n)$ tensor, `out` will be a $(j \times k \times n \times n)$ tensor.
  Note that the broadcasting logic only looks at the batch dimensions when determining if the inputs are broadcastable, and not the matrix dimensions. For example, if `input` is a $(j \times 1 \times n \times m)$ tensor and `other` is a $(k \times m \times p)$ tensor, these inputs are valid for broadcasting even though the final two dimensions (i.e. the matrix dimensions) are different. `out` will be a $(j \times k \times n \times p)$ tensor.

This operation has support for arguments with sparse layouts. In particular the matrix-matrix (both arguments 2-dimensional) supports sparse arguments with the same restrictions as `torch.mm()`

all rules

1.2 Matrix multiplication

```
import torch
tensor = torch.tensor([1, 2, 3])
tensor.shape
>>> torch.Size([3])
```

The difference between element-wise multiplication and matrix multiplication is the addition of values.
For our tensor variable with values [1, 2, 3]:

Operation	Calculation	Code
Element-wise multiplication	`[1*1, 2*2, 3*3] = [1, 4, 9]`	`tensor * tensor`
Matrix multiplication	`[1*1 + 2*2 + 3*3] = [14]`	`tensor.matmul(tensor)`

```
# Element-wise matrix mutlication
tensor * tensor
>>>
tensor([1, 4, 9])
```

torch.matmul(tensor, tensor)

```
``` python
# Can also use the "@" symbol for matrix multiplication, though not recommended
tensor @ tensor

>>>
tensor(14)
```
```

1.3 One of the most common errors in deep learning (shape errors)

Because much of deep learning is multiplying and performing operations on matrices and matrices have a strict rule about what shapes and sizes can be combined, one of the most common errors you'll run into in deep learning is shape mismatches.

```
In [37]: # Shapes need to be in the right way
         tensor_A = torch.tensor([[1, 2],
                                  [3, 4],
                                  [5, 6]], dtype=torch.float32)

         tensor_B = torch.tensor([[7, 10],
                                  [8, 11],
                                  [9, 12]], dtype=torch.float32)

         torch.matmul(tensor_A, tensor_B) # (this will error)
```

```
---------------------------------------------------------------------------
RuntimeError                              Traceback (most recent call last)
Input In [37], in <cell line: 10>()
      2 tensor_A = torch.tensor([[1, 2],
      3                          [3, 4],
      4                          [5, 6]], dtype=torch.float32)
      6 tensor_B = torch.tensor([[7, 10],
      7                          [8, 11],
      8                          [9, 12]], dtype=torch.float32)
---> 10 torch.matmul(tensor_A, tensor_B)

RuntimeError: mat1 and mat2 shapes cannot be multiplied (3x2 and 3x2)
```

We can make matrix multiplication work between tensor_A and tensor_B by making their inner dimensions match.
One of the ways to do this is with a transpose (switch the dimensions of a given tensor).

You can perform transposes in PyTorch using either:

- torch.transpose(input, dim0, dim1) - where input is the desired tensor to transpose and dim0 and dim1 are the dimensions to be swapped.
- tensor.T - where tensor is the desired tensor to transpose.

1.3 One of the most common errors in deep learning (shape errors)

```
# View tensor_A and tensor_B
print(tensor_A)
print(tensor_B)

>>>
tensor([[1., 2.],
        [3., 4.],
        [5., 6.]])
tensor([[ 7., 10.],
        [ 8., 11.],
        [ 9., 12.]])
```

```
# View tensor_A and tensor_B.T
print(tensor_A)
print(tensor_B.T)

>>>

tensor([[1., 2.],
        [3., 4.],
        [5., 6.]])
tensor([[ 7.,  8.,  9.],
        [10., 11., 12.]])
```

1.3 One of the most common errors in deep learning (shape errors)

```
# The operation works when tensor_B is transposed
print(f"Original shapes: tensor_A = {tensor_A.shape}, tensor_B = {tensor_B.shape}\n")
print(f"New shapes: tensor_A = {tensor_A.shape} (same as above), tensor_B.T = {tensor_B.T.shape}\n")
print(f"Multiplying: {tensor_A.shape} * {tensor_B.T.shape} <- inner dimensions match\n")
print("Output:\n")
output = torch.matmul(tensor_A, tensor_B.T)
print(output)
print(f"\nOutput shape: {output.shape}")

>>>

Original shapes: tensor_A = torch.Size([3, 2]), tensor_B = torch.Size([3, 2])

New shapes: tensor_A = torch.Size([3, 2]) (same as above), tensor_B.T = torch.Size([2, 3])

Multiplying: torch.Size([3, 2]) * torch.Size([2, 3]) <- inner dimensions match

Output:

tensor([[ 27.,  30.,  33.],
        [ 61.,  68.,  75.],
        [ 95., 106., 117.]])

Output shape: torch.Size([3, 3])
```

You can also use torch.mm() which is a short for torch.matmul().

```
# torch.mm is a shortcut for matmul
torch.mm(tensor_A, tensor_B.T)
>>>
tensor([[ 27.,  30.,  33.],
        [ 61.,  68.,  75.],
        [ 95., 106., 117.]])
```

1.3 One of the most common errors in deep learning (shape errors)

Without the transpose, the rules of matrix mulitplication aren't fulfilled and we get an error like above.
Neural networks are full of matrix multiplications and dot products.
The torch.nn.Linear() module (we'll see this in action later on), also known as a feed-forward layer or fully connected layer, implements a matrix multiplication between an input x and a weights matrix A.

$$y = x \cdot A^T + b$$

Where:

- `x` is the input to the layer (deep learning is a stack of layers like `torch.nn.Linear()` and others on top of each other).
- `A` is the weights matrix created by the layer, this starts out as random numbers that get adjusted as a neural network learns to better represent patterns in the data (notice the "`T`", that's because the weights matrix gets transposed).
  - **Note:** You might also often see `W` or another letter like `X` used to showcase the weights matrix.
- `b` is the bias term used to slightly offset the weights and inputs.
- `y` is the output (a manipulation of the input in the hopes to discover patterns in it).

This is a linear function (you may have seen something like $y = mx + b$ in high school or elsewhere), and can be used to draw a straight line!
Let's play around with a linear layer. Try changing the values of in_features and out_features below and see what happens.
Do you notice anything to do with the shapes?

1.3 One of the most common errors in deep learning (shape errors)

```
# Since the linear layer starts with a random weights matrix, let's make it reproducible (more on this later)
torch.manual_seed(42)
# This uses matrix mutliplcation
linear = torch.nn.Linear(in_features=2, # in_features = matches inner dimension of input
                         out_features=6) # out_features = describes outer value
x = tensor_A
output = linear(x)
print(f"Input shape: {x.shape}\n")
print(f"Output:\n{output}\n\nOutput shape: {output.shape}")

>>>
Input shape: torch.Size([3, 2])

Output:
tensor([[2.2368, 1.2292, 0.4714, 0.3864, 0.1309, 0.9838],
        [4.4919, 2.1970, 0.4469, 0.5285, 0.3401, 2.4777],
        [6.7469, 3.1648, 0.4224, 0.6705, 0.5493, 3.9716]],
       grad_fn=<AddmmBackward0>)

Output shape: torch.Size([3, 6])
```

if input_shape = (3,2) **you can interpret it as 3 batches where each sample has 2 features.**

If you've never done it before, matrix multiplication can be a confusing topic at first.

But after you've played around with it a few times and even cracked open a few neural networks, you'll notice it's everywhere.

Remember, matrix multiplication is all you need.

source

> **Question:** What happens if you change `in_features` from 2 to 3 above? Does it error? How could you change the shape of the input (`x`) to accomodate to the error? Hint: what did we have to do to `tensor_B` above?

check_this_source

```
y = x * A_t + b and the linear layer has in_f = 2, out_f = 6 => A_t = (2,6) (2 columns, 6 rows) for A matrix (the weight matrix).
y = (3,2) * (2,6) + b  = (3,6)
```

2. Finding min max, mean sum etc (aggregation)

Now we've seen a few ways to manipulate tensors, let's run through a few ways to aggregate them (go from more values to less values).
First we'll create a tensor and then find the max, min, mean and sum of it.

```
# Create a tensor
x = torch.arange(0, 100, 10)
x
>>>
tensor([ 0, 10, 20, 30, 40, 50, 60, 70, 80, 90])
```

Now let's perform some aggregation.

```
print(f"Minimum: {x.min()}")
print(f"Maximum: {x.max()}")
# print(f"Mean: {x.mean()}") # this will error
print(f"Mean: {x.type(torch.float32).mean()}") # won't work without float datatype
print(f"Sum: {x.sum()}")
>>>
Minimum: 0
Maximum: 90
Mean: 45.0
Sum: 450
```

Note: You may find some methods such as torch.mean() require tensors to be in torch.float32 (the most common) or another specific datatype, otherwise the operation will fail.
You can also do the same as above with torch methods.

```
torch.max(x), torch.min(x), torch.mean(x.type(torch.float32)), torch.sum(x)
```

3. Positional min/max

You can also find the index of a tensor where the max or minimum occurs with torch.argmax() and torch.argmin() respectively. This is helpful incase you just want the position where the highest (or lowest) value is and not the actual value itself (we'll see this in a later section when using the softmax activation function).

```
# Create a tensor
tensor = torch.arange(10, 100, 10)
print(f"Tensor: {tensor}")

# Returns index of max and min values
print(f"Index where max value occurs: {tensor.argmax()}")
print(f"Index where min value occurs: {tensor.argmin()}")
>>>

Tensor: tensor([10, 20, 30, 40, 50, 60, 70, 80, 90])
Index where max value occurs: 8
Index where min value occurs: 0
```

4. Change tensor datatype

As mentioned, a common issue with deep learning operations is having your tensors in different datatypes. If one tensor is in torch.float64 and another is in torch.float32, you might run into some errors. But there's a fix. You can change the datatypes of tensors using torch.Tensor.type(dtype=None) where the dtype parameter is the datatype you'd like to use.
First we'll create a tensor and check it's datatype (the default is torch.float32).

```
# Create a tensor and check its datatype
tensor = torch.arange(10., 100., 10.)
tensor.dtype
>>> torch.float32
```

Now we'll create another tensor the same as before but change its datatype to torch.float16.

```
# Create a float16 tensor
tensor_float16 = tensor.type(torch.float16)
tensor_float16
>>>
tensor([10., 20., 30., 40., 50., 60., 70., 80., 90.], dtype=torch.float16)
```

5. Tensor shape

Reshaping a tensor is a frequently used operation. We can change the shape of a tensor without the memory copying overhead. There are two methods for that: reshape and view.

The difference is the following:

- view tries to return the tensor, and it shares the same memory with the original tensor. In case, if it cannot reuse the same memory due to some reasons , it just fails.
- reshape always returns the tensor with the desired shape and tries to reuse the memory. If it cannot, it creates a copy.

5. Tensor shape

```
tensor = torch.rand(2, 3, 4)
print('Pointer to data: ', tensor.data_ptr())
print('Shape: ', tensor.shape)

reshaped = tensor.reshape(24)

view = tensor.view(3, 2, 4)
print('Reshaped tensor - pointer to data', reshaped.data_ptr())
print('Reshaped tensor shape ', reshaped.shape)

print('Viewed tensor - pointer to data', view.data_ptr())
print('Viewed tensor shape ', view.shape)

assert tensor.data_ptr() == view.data_ptr()

assert np.all(np.equal(tensor.numpy().flat, reshaped.numpy().flat))

print('Original stride: ', tensor.stride())
print('Reshaped stride: ', reshaped.stride())
print('Viewed stride: ', view.stride())


>>>
Pointer to data:  94161923319488
Shape:  torch.Size([2, 3, 4])
Reshaped tensor - pointer to data 94161923319488
Reshaped tensor shape  torch.Size([24])
Viewed tensor - pointer to data 94161923319488
Viewed tensor shape  torch.Size([3, 2, 4])
Original stride:  (12, 4, 1)
Reshaped stride:  (1,)
Viewed stride:  (8, 4, 1)
```

5. Tensor shape

The basic rule about reshaping the tensor is definitely that you cannot change the total number of elements in it, so the product of all tensor's dimensions should always be the same. It gives us the ability to avoid specifying one dimension when reshaping the tensor - Pytorch can calculate it for us:

```
print(tensor.reshape(3, 2, 4).shape)
print(tensor.reshape(3, 2, -1).shape)
print(tensor.reshape(3, -1, 4).shape)

>>
torch.Size([3, 2, 4])
torch.Size([3, 2, 4])
torch.Size([3, 2, 4])
```

Alternative ways to view tensors - `expand` or `expand_as` .

- `expand` - requires the desired shape as an input
- `expand_as` - uses the shape of another tensor.

These operaitions "repeat" tensor's values along the specified axes without actual copying the data.

As the documentation says, expand

> returns a new view of the self tensor with singleton dimensions expanded to a larger size. Tensor can be also expanded to a larger number of dimensions, and the new ones will be appended at the front. For the new dimensions, the size cannot be set to -1.

Use case:

- index multi-channel tensor with single-channel mask - imagine a color image with 3 channels (R, G and B) and binary mask for the area of interest on that image. We cannot index the image with this kind of mask directly since the dimensions are different, but we can use `expand_as` operation to create a view of the mask that has the same dimensions as the image we want to apply it to, but has not copied the data.

5. Tensor shape

```
%matplotlib inline
from matplotlib import pyplot as plt

# Create a black image
image = torch.zeros(size=(3, 256, 256), dtype=torch.int)

# Leave the borders and make the rest of the image Green
image[1, 18:256 - 18, 18:256 - 18] = 255

# Create a mask of the same size
mask = torch.zeros(size=(256, 256), dtype=torch.bool)

# Assuming the green region in the original image is the Region of interest, change the mask to white for that area
mask[18:256 - 18, 18:256 - 18] = 1

# Create a view of the mask with the same dimensions as the original image
mask_expanded = mask.expand_as(image)
print(mask_expanded.shape)

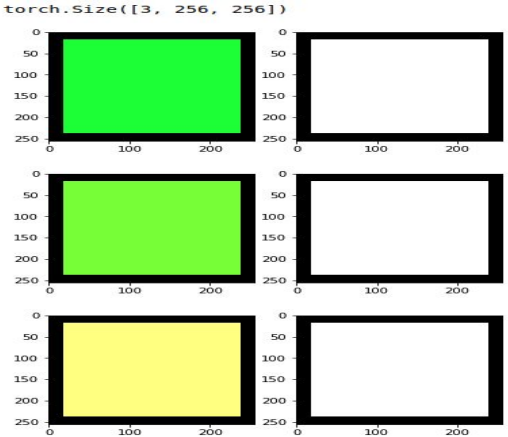
mask_np = mask_expanded.numpy().transpose(1, 2, 0) * 255
image_np = image.numpy().transpose(1, 2, 0)

fig, ax = plt.subplots(1, 2)
ax[0].imshow(image_np)
ax[1].imshow(mask_np)
plt.show()

image[0, mask] += 128
fig, ax = plt.subplots(1, 2)
ax[0].imshow(image_np)
ax[1].imshow(mask_np)
plt.show()

image[mask_expanded] += 128
image.clamp_(0, 255)
fig, ax = plt.subplots(1, 2)
ax[0].imshow(image_np)
ax[1].imshow(mask_np)
plt.show()
```

```
torch.Size([3, 256, 256])
```



In the example above, one can also find a couple of useful tricks:

- `clamp` method and function is a Pytorch's analogue of NumPy's `clip` function
- many operations on tensors have in-place form, that does not return modified data, but change values in the tensor. The in-place version of the operation has trailing underscore according to Pytorch's naming convention - in the example above it is `clamp_`
- tensors have the same indexing as Numpy's arrays - one can use `:`-separated range, negative indexes and so on.