01 Basics!

Random tensors

We've established tensors represent some form of data.

And machine learning models such as neural networks manipulate and seek patterns within tensors.

But when building machine learning models with PyTorch, it's rare you'll create tensors by hand

In essence:

`Start with random numbers -> look at data -> update random numbers -> look at data -> update random numbers...`

Zeros and ones

Sometimes you'll just want to fill tensors with zeros or ones.

This happens a lot with masking (like masking some of the values in one tensor with zeros to let a model know not to learn them).

```
# Create a tensor of all zeros
zeros = torch.zeros(size=(3, 4))
zeros, zeros.dtype

# Create a tensor of all ones
ones = torch.ones(size=(3, 4))
ones, ones.dtype
```

Creating a range and tensors like

Sometimes you might want a range of numbers, such as 1 to 10 or 0 to 100.

You can use `torch.arange(start, end, step)` to do so.

Where:

- `start` = start of range (e.g. 0)
- `end` = end of range (e.g. 10)
- `step` = how many steps in between each value (e.g. 1)

```
# Use torch.arange(), torch.range() is deprecated
zero_to_ten_deprecated = torch.range(0, 10) # Note: this may ret

# Create a range of values 0 to 10
zero_to_ten = torch.arange(start=0, end=10, step=1)
zero_to_ten

# Can also create a tensor of zeros similar to another tensor
ten_zeros = torch.zeros_like(input=zero_to_ten) # will have same
ten_zeros
```

Tensor datatypes

There are many different tensor data types

Some are specific for CPU and some are better for GPU.

Generally `torch.cuda` , this tensor is being used for GPU (since Nvidia GPUs use a computing toolkit called CUDA).

The most common type (and generally the default) is `torch.float32` or `torch.float`.

This is referred to as "32-bit floating point".

But there's also 16-bit floating point (`torch.float16` or `torch.half`) and 64-bit floating point (`torch.float64` or `torch.double`).

Getting information from tensors

Once you've created tensors (or someone else or a PyTorch module has created them for you), you might want to get some information from them.

We've seen these before but three of the most common attributes you'll want to find out about tensors are:

- `shape` - what shape is the tensor? (some operations require specific shape rules)
- `dtype` - what datatype are the elements within the tensor stored in?
- `device` - what device is the tensor stored on? (usually GPU or CPU)

```
# Create a tensor
some_tensor = torch.rand(3, 4)

# Find out details about it
print(some_tensor)
print(f"Shape of tensor: {some_tensor.shape}")
print(f"Datatype of tensor: {some_tensor.dtype}")
print(f"Device tensor is stored on: {some_tensor.device}") # wi

output :
tensor([[0.4688, 0.0055, 0.8551, 0.0646],
        [0.6538, 0.5157, 0.4071, 0.2109],
        [0.9960, 0.3061, 0.9369, 0.7008]])
Shape of tensor: torch.Size([3, 4])
Datatype of tensor: torch.float32
Device tensor is stored on: cpu
```

Manipulating tensors (tensor operations)

These operations are often between:

- Addition

- Substraction
- Multiplication (element-wise)
- Division
- Matrix multiplication

Basic operations

Let's start with a few of the fundamental operations, addition (`+`), subtraction (`-`), mutliplication (`*`).

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

# Multiply it by 10
tensor * 10

# Tensors don't change unless reassigned
tensor

# Subtract and reassign
tensor = tensor - 10
tensor
```

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

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

Matrix multiplication

PyTorch 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

1. 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.

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)`

a strict rule about what shapes and sizes can be combined, one of the most common errors in deep learning is shape mismatches.

One of the way to fix this is with a **transpose**

You can perform transposes in PyTorch using

- `tensor.T` - where `tensor` is the desired tensor to transpose.

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

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

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

Finding the min, max, mean, sum, etc (aggregation)

create a tensor and then find the max, min, mean and sum of it.

```
# Create a tensor
x = torch.arange(0, 100, 10)
x

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 with
print(f"Sum: {x.sum()}")

output:
Minimum: 0
Maximum: 90
```

```
Mean: 45.0
Sum: 450
```

`torch.mean()` require tensors to be in `torch.float32`

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.

Change tensor datatype

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.