PyTorch

Introduction to tensors

Tensors are the fundamental building block of machine learning. Their job is to represent data in a numerical way.

For example, you could represent an image as a tensor with shape `[3, 224, 224]` which would mean `[colour_channels, height, width]`, as in the image has `3` colour channels (red, green, blue), a height of `224` pixels and a width of `224` pixels.

Creating tensors

A `torch.Tensor` is a multi-dimensional matrix containing elements of a single data type.

Scalar

A scalar is a single number and in tensor-speak it's a zero dimension tensor.

1. *# Scalar*
2. *scalar = torch.tensor(7)*
3. *scalar*

We can check the dimensions of a tensor using the `ndim` attribute.

1. *scalar.ndim*

What if we wanted to retrieve the number from the tensor?

1. *# Get the Python number within a tensor (only works with one-element tensors)*
2. *scalar.item()*

Vector

A vector is a single dimension tensor but can contain many numbers.

1. *# Vector*
2. *vector = torch.tensor([7, 7])*
3. *Vector*
4. *vector.ndim()*
    a. *1*
5. *# Check shape of vector*
6. *Vector.shape*

a. *torch.Size([2])*

Matrix

1. *# Matrix*
2. *MATRIX = torch.tensor([[7, 8],*

   [9, 10]])

3. *MATRIX*

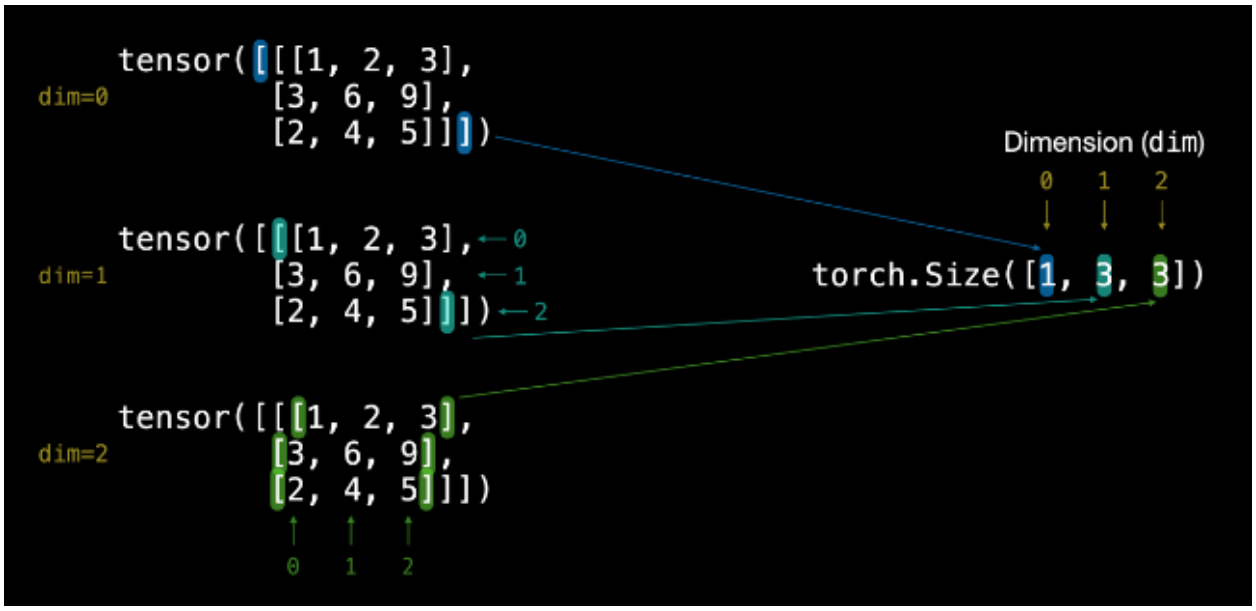
MATRIX has two dimensions

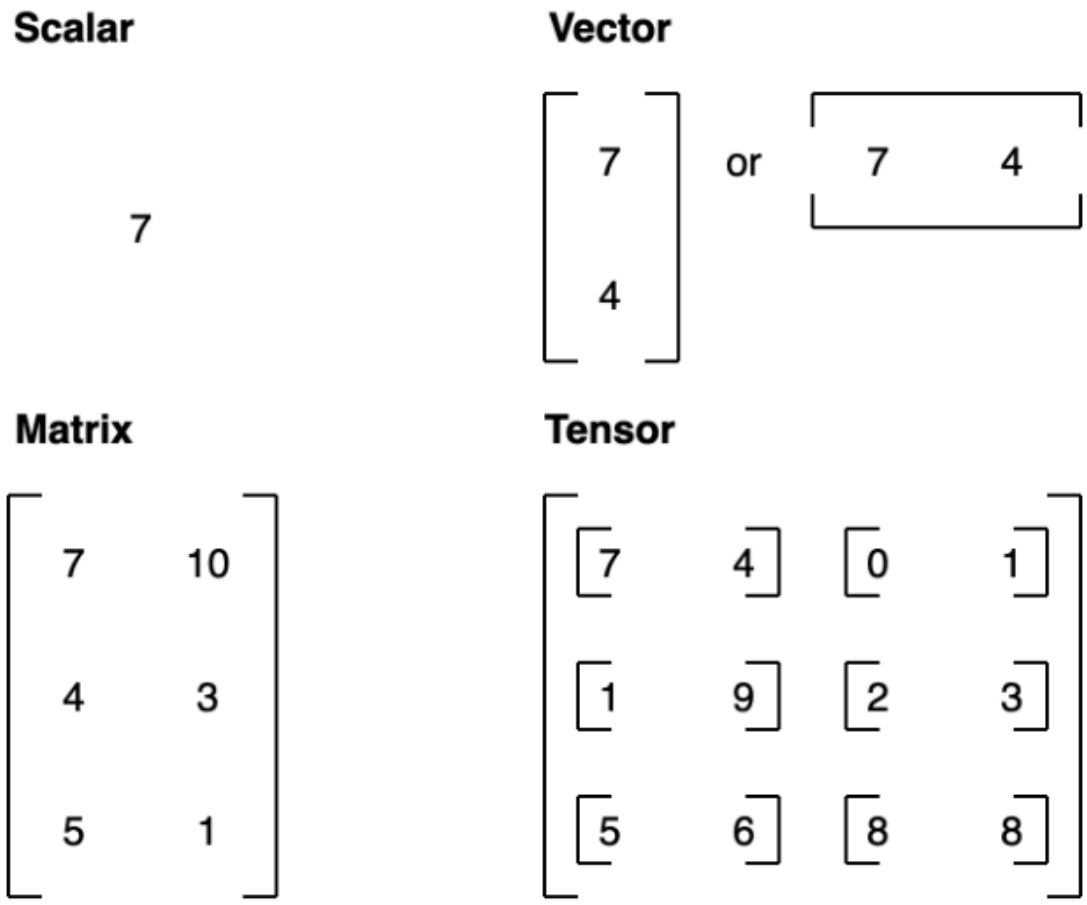
a. *MATRIX.shape*

torch.Size([2, 2])

Tensor

An n-dimensional array of numbers.	Can be any number, a 0-dimension tensor is a scalar, a 1-dimension tensor is a vector.





Random Tensor

Instead, a machine learning model often starts out with large random tensors of numbers and adjusts these random numbers as it works through data to better represent it.

In essence:

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

1. *# Create a random tensor of size (3, 4)*
2. random_tensor **=** torch**.**rand(size**=**(3, 4))
3. random_tensor, random_tensor**.**dtype

(tensor([[0.6541, 0.4807, 0.2162, 0.6168],
[0.4428, 0.6608, 0.6194, 0.8620],

```
        [0.2795, 0.6055, 0.4958, 0.5483]]),
torch.float32)
```

The flexibility of `torch.rand()` is that we can adjust the `size` to be whatever we want.

For example, say you wanted a random tensor in the common image shape of `[224, 224, 3]` (`[height, width, color_channels]`).

```
1. # Create a random tensor of size (224, 224, 3)
2. random_image_size_tensor = torch.rand(size=(224, 224, 3))
3. random_image_size_tensor.shape, random_image_size_tensor.ndim
```

```
(torch.Size([224, 224, 3]), 3)
```

Zeros and ones

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

Creating a range and tensors like

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

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

Tensor datatypes

There are many different tensor datatypes available in PyTorch.

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

Getting to know which is which can take some time.

Generally if you see `torch.cuda` anywhere, the 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`).

1. *# Default datatype for tensors is float32*
2. *float_32_tensor = torch.tensor([3.0, 6.0, 9.0],*
   dtype=None, # defaults to None, which is torch.float32 or whatever datatype is passed
   device=None, # defaults to None, which uses the default tensor type
   requires_grad=False) # if True, operations performed on the tensor are recorded

3. *float_32_tensor.shape, float_32_tensor.dtype, float_32_tensor.device*

   (torch.Size([3]), torch.float32, device(type='cpu'))

Manipulating tensors (tensor operations)

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

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

Matrix multiplication (is all you need)

`torch.matmul()`

The **inner dimensions** must match.

The resulting matrix has the shape of the **outer dimensions.**

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)`

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

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.

The `torch.nn.Linear()` module, also known as a feed-forward layer or fully connected layer, implements a matrix multiplication between an input `x` and a weights matrix `A`.

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

1. *print(f"Minimum: {x.min()}")*
2. *print(f"Maximum: {x.max()}")*
3. *# print(f"Mean: {x.mean()}") # this will error*
4. *print(f"Mean: {x.type(torch.float32).mean()}") # won't work without float datatype*
5. *print(f"Sum: {x.sum()}")*

Minimum: 0

Maximum: 90

Mean: 45.0

Sum: 450

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.

```
# 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
```

Change tensor datatype

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.

Reshaping, stacking, squeezing and unsqueezing

Method	One-line description
`torch.reshape(input, shape)`	Reshapes `input` to `shape` (if compatible), can also use `torch.Tensor.reshape()`.
`Tensor.view(shape)`	Returns a view of the original tensor in a different `shape` but shares the same data as the original tensor.
`torch.stack(tensors, dim=0)`	Concatenates a sequence of `tensors` along a new dimension (`dim`), all `tensors` must be same size.
`torch.squeeze(input)`	Squeezes `input` to remove all the dimenions with value `1`.
`torch.unsqueeze(input, dim)`	Returns `input` with a dimension value of `1` added at `dim`.
`torch.permute(input, dims)`	Returns a *view* of the original `input` with its dimensions permuted (rearranged) to `dims`.

Indexing (selecting data from tensors)

1. *# Let's index bracket by bracket*
2. print(f"First square bracket:\n{x[0]}")
3. print(f"Second square bracket: {x[0][0]}")
4. print(f"Third square bracket: {x[0][0][0]}")

```
First square bracket:
tensor([[1, 2, 3],
        [4, 5, 6],
        [7, 8, 9]])
Second square bracket: tensor([1, 2, 3])
Third square bracket: 1
```

1. *# Get all values of 0th dimension and the 0 index of 1st dimension*
2. x[:, 0]

3. *# Get all values of 0th & 1st dimensions but only index 1 of 2nd dimension*

4. x[:, :, 1]

5. *# Get all values of the 0 dimension but only the 1 index value of the 1st and 2nd dimension*
6. x[:, 1, 1]

PyTorch tensors & NumPy

Since NumPy is a popular Python numerical computing library, PyTorch has functionality to interact with it nicely.

The two main methods you'll want to use for NumPy to PyTorch (and back again) are:

- `torch.from_numpy(ndarray)` - NumPy array -> PyTorch tensor.
- `torch.Tensor.numpy()` - PyTorch tensor -> NumPy array.

Reproducibility (trying to take the random out of random)

```
start with random numbers -> tensor operations -> try to make better
(again and again and again)
```

Just as you might've expected, the tensors come out with different values.

But what if you wanted to created two random tensors with the *same* values.

As in, the tensors would still contain random values but they would be of the same flavour.

That's where `torch.manual_seed(seed)` comes in, where `seed` is an integer (like `42` but it could be anything) that flavours the randomness.