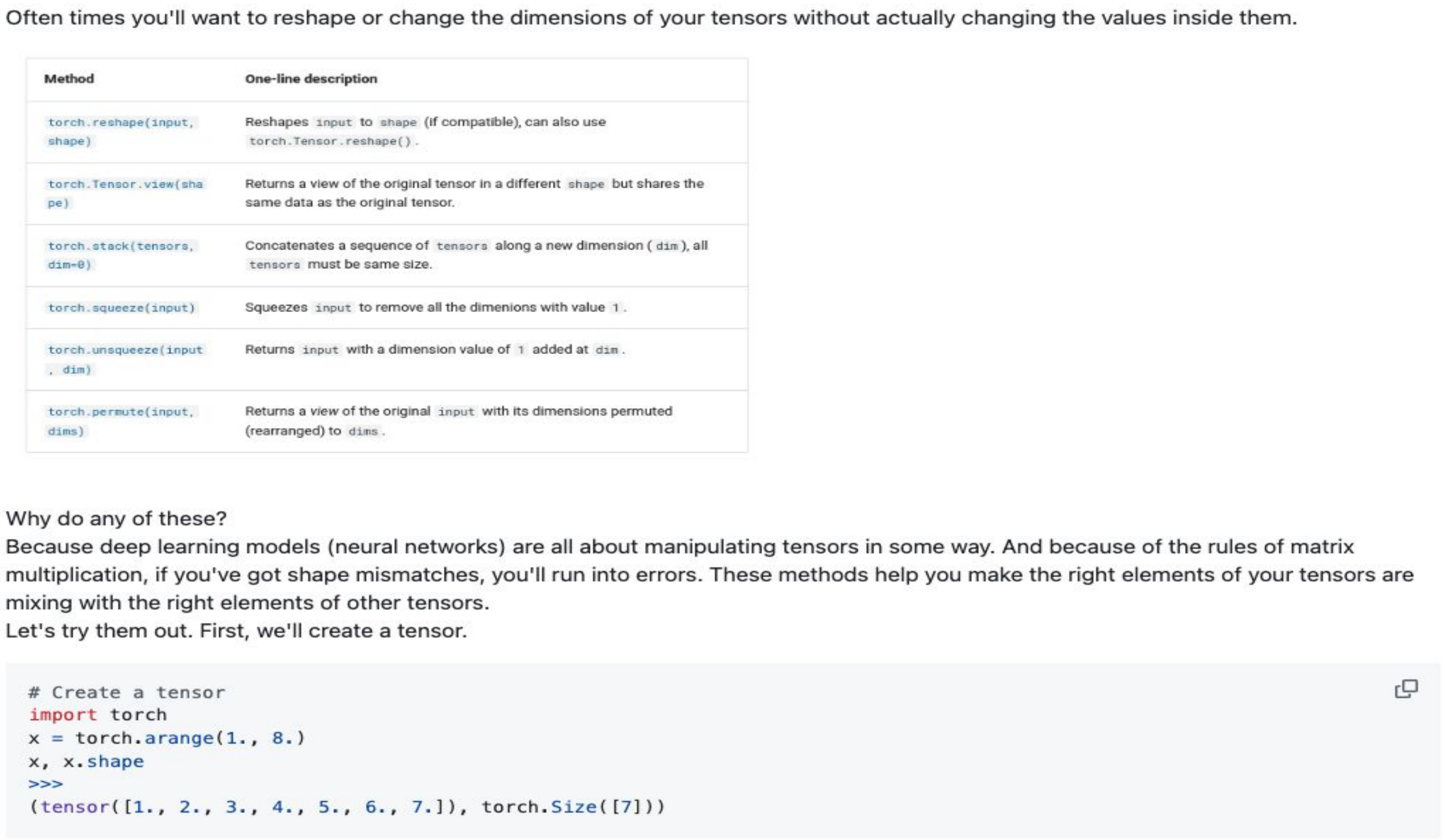
Manipulating Tensors

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)

6. Reshaping, stacking, squeezing and unsqueezing



Often times you'll want to reshape or change the dimensions of your tensors without actually changing the values inside them.

Method	One-line description
`torch.reshape(input, shape)`	Reshapes `input` to `shape` (if compatible), can also use `torch.Tensor.reshape()`.
`torch.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`.

Why do any of these?
Because deep learning models (neural networks) are all about manipulating tensors in some way. And because of the rules of matrix multiplication, if you've got shape mismatches, you'll run into errors. These methods help you make the right elements of your tensors are mixing with the right elements of other tensors.
Let's try them out. First, we'll create a tensor.

```
# Create a tensor
import torch
x = torch.arange(1., 8.)
x, x.shape
>>>
(tensor([1., 2., 3., 4., 5., 6., 7.]), torch.Size([7]))
```

6. Reshaping, stacking, squeezing and unsqueezing

Now let's add an extra dimension with torch.reshape().

```
# Add an extra dimension
x_reshaped = x.reshape(1, 7)
x_reshaped, x_reshaped.shape
>>>
(tensor([[1., 2., 3., 4., 5., 6., 7.]]), torch.Size([1, 7]))
```

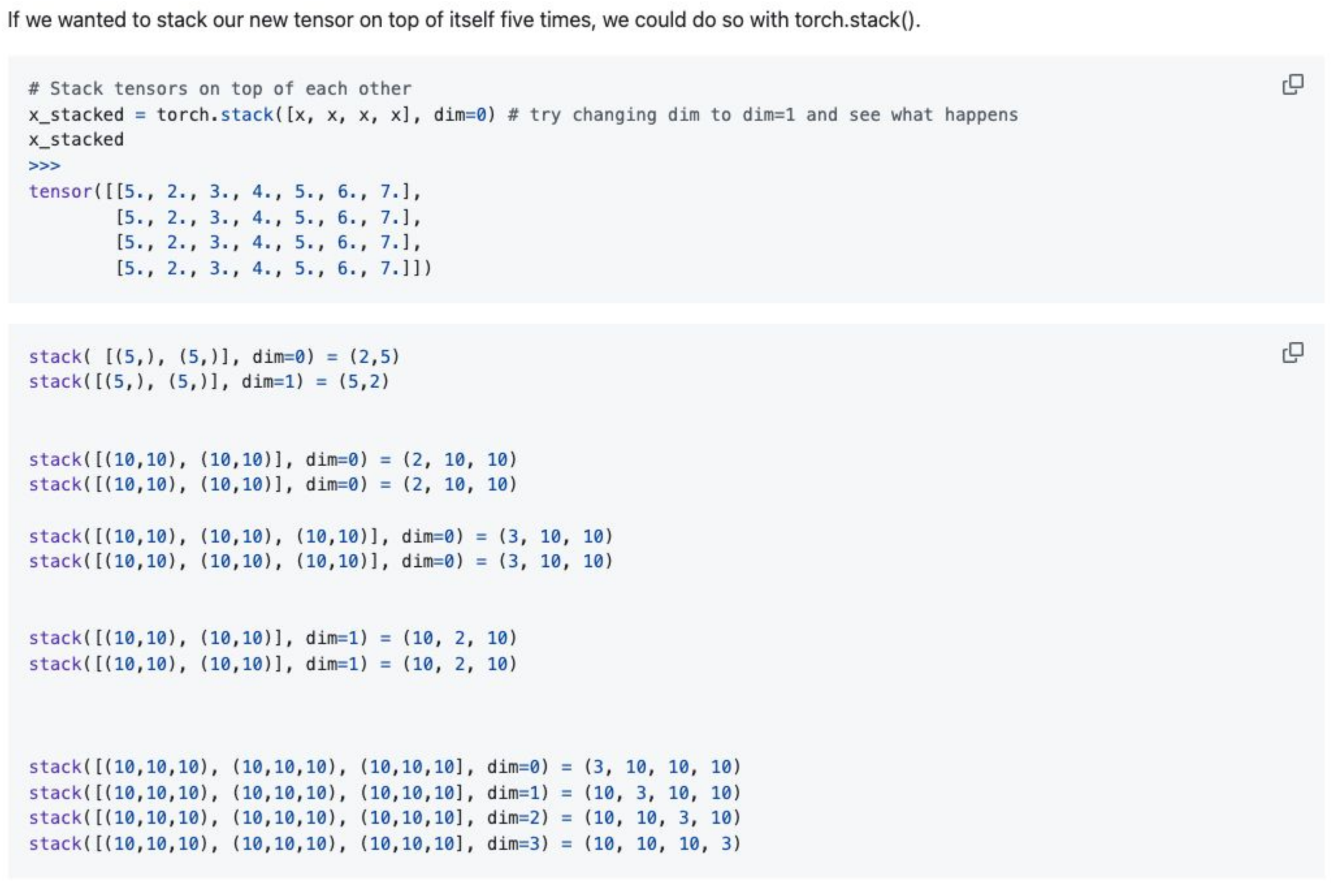
We can also change the view with torch.view()

```
# Change view (keeps same data as original but changes view)
# See more: https://stackoverflow.com/a/54507446/7900723
z = x.view(1, 7)
z, z.shape
>>>
(tensor([[1., 2., 3., 4., 5., 6., 7.]]), torch.Size([1, 7]))
```

Remember though, changing the view of a tensor with torch.view() really only creates a new view of the same tensor.
So changing the view changes the original tensor too.

```
# Changing z changes x
z[:, 0] = 5
z, x
>>>
(tensor([[5., 2., 3., 4., 5., 6., 7.]]), tensor([5., 2., 3., 4., 5., 6., 7.]))
```

6. Reshaping, stacking, squeezing and unsqueezing



If we wanted to stack our new tensor on top of itself five times, we could do so with torch.stack().

```
# Stack tensors on top of each other
x_stacked = torch.stack([x, x, x, x], dim=0) # try changing dim to dim=1 and see what happens
x_stacked
>>>
tensor([[5., 2., 3., 4., 5., 6., 7.],
        [5., 2., 3., 4., 5., 6., 7.],
        [5., 2., 3., 4., 5., 6., 7.],
        [5., 2., 3., 4., 5., 6., 7.]])
```

```
stack( [(5,), (5,)], dim=0) = (2,5)
stack([(5,), (5,)], dim=1) = (5,2)


stack([(10,10), (10,10)], dim=0) = (2, 10, 10)
stack([(10,10), (10,10)], dim=0) = (2, 10, 10)

stack([(10,10), (10,10), (10,10)], dim=0) = (3, 10, 10)
stack([(10,10), (10,10), (10,10)], dim=0) = (3, 10, 10)


stack([(10,10), (10,10)], dim=1) = (10, 2, 10)
stack([(10,10), (10,10)], dim=1) = (10, 2, 10)



stack([(10,10,10), (10,10,10), (10,10,10], dim=0) = (3, 10, 10, 10)
stack([(10,10,10), (10,10,10), (10,10,10], dim=1) = (10, 3, 10, 10)
stack([(10,10,10), (10,10,10), (10,10,10], dim=2) = (10, 10, 3, 10)
stack([(10,10,10), (10,10,10), (10,10,10], dim=3) = (10, 10, 10, 3)
```

6. Reshaping, stacking, squeezing and unsqueezing

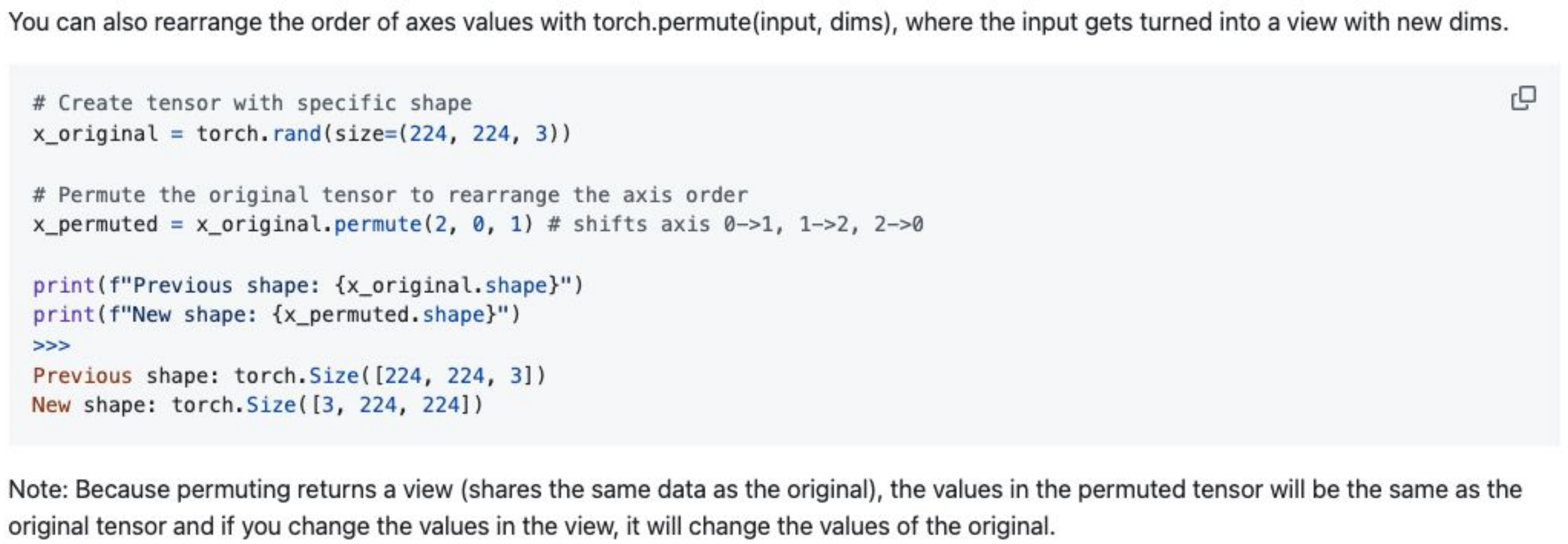
How about removing all single dimensions from a tensor?
To do so you can use torch.squeeze() (I remember this as squeezing the tensor to only have dimensions over 1).

```
print(f"Previous tensor: {x_reshaped}")
print(f"Previous shape: {x_reshaped.shape}")

# Remove extra dimension from x_reshaped
x_squeezed = x_reshaped.squeeze()
print(f"\nNew tensor: {x_squeezed}")
print(f"New shape: {x_squeezed.shape}")

>>>
Previous tensor: tensor([[5., 2., 3., 4., 5., 6., 7.]])
Previous shape: torch.Size([1, 7])

New tensor: tensor([5., 2., 3., 4., 5., 6., 7.])
New shape: torch.Size([7])
```

And to do the reverse of torch.squeeze() you can use torch.unsqueeze() to add a dimension value of 1 at a specific index.

```
print(f"Previous tensor: {x_squeezed}")
print(f"Previous shape: {x_squeezed.shape}")

## Add an extra dimension with unsqueeze
x_unsqueezed = x_squeezed.unsqueeze(dim=0)
print(f"\nNew tensor: {x_unsqueezed}")
print(f"New shape: {x_unsqueezed.shape}")

>>>
Previous tensor: tensor([5., 2., 3., 4., 5., 6., 7.])
Previous shape: torch.Size([7])

New tensor: tensor([[5., 2., 3., 4., 5., 6., 7.]])
New shape: torch.Size([1, 7])
```

6. Reshaping, stacking, squeezing and unsqueezing



You can also rearrange the order of axes values with torch.permute(input, dims), where the input gets turned into a view with new dims.

```
# Create tensor with specific shape
x_original = torch.rand(size=(224, 224, 3))

# Permute the original tensor to rearrange the axis order
x_permuted = x_original.permute(2, 0, 1) # shifts axis 0->1, 1->2, 2->0

print(f"Previous shape: {x_original.shape}")
print(f"New shape: {x_permuted.shape}")
>>>
Previous shape: torch.Size([224, 224, 3])
New shape: torch.Size([3, 224, 224])
```

Note: Because permuting returns a view (shares the same data as the original), the values in the permuted tensor will be the same as the original tensor and if you change the values in the view, it will change the values of the original.

7. Indexing(selecting data from tensors)

Tensor provides access to its elements via the same [] operation as a regular python list or NumPy array. However, as you may recall from NumPy usage, the full power of math libraries is accessible only via vectorized operations, i.e. operations without explicit looping over all vector elements in python and using implicit optimized loops in C/C++/CUDA/Fortran/etc. available via special functions calls. Pytorch employs the same paradigm and provides a wide range of vectorized operations. Let's take a look at some examples.

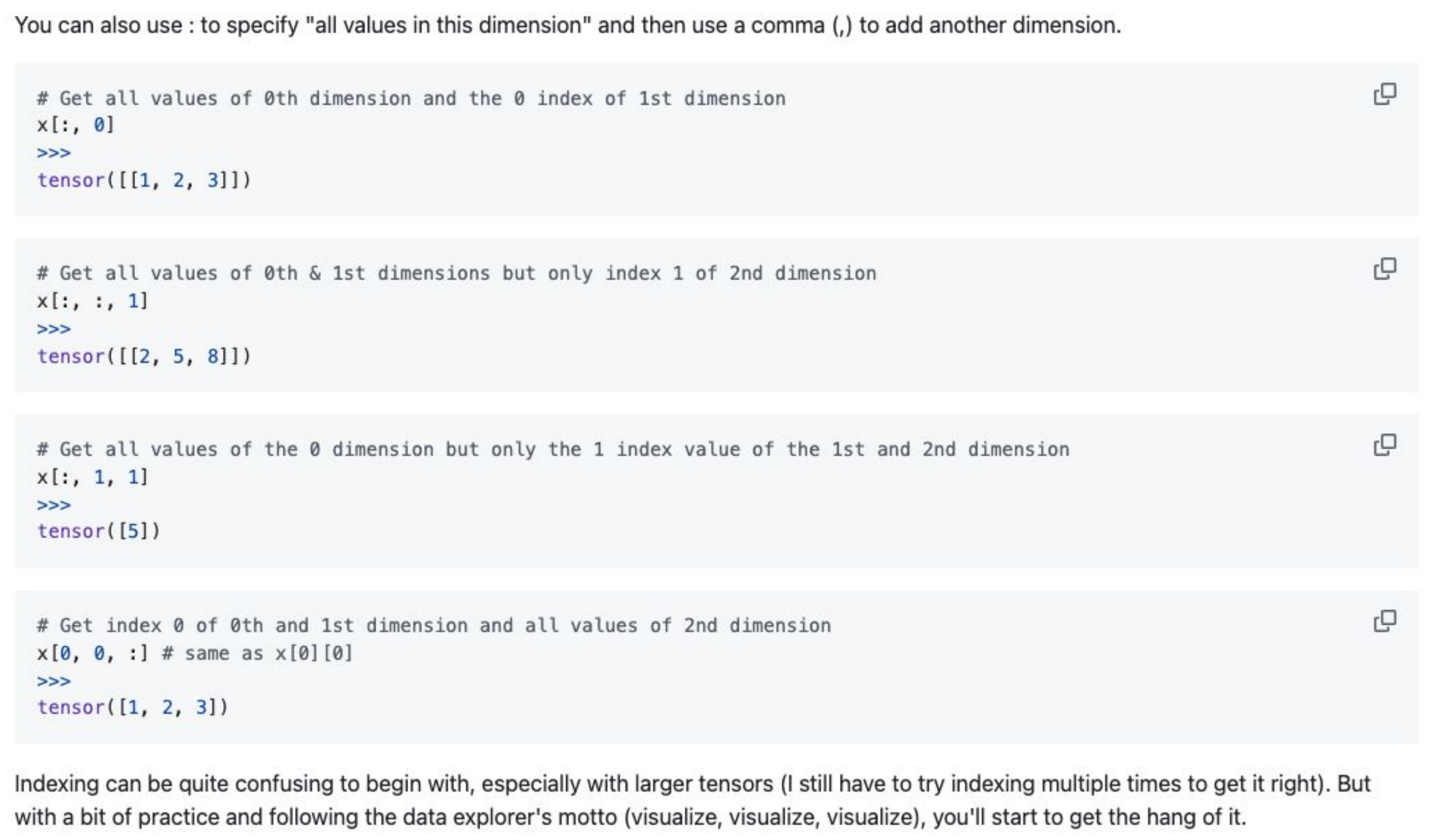
Sometimes you'll want to select specific data from tensors (for example, only the first column or second row).
To do so, you can use indexing.
If you've ever done indexing on Python lists or NumPy arrays, indexing in PyTorch with tensors is very similar.

```
# Create a tensor
import torch
x = torch.arange(1, 10).reshape(1, 3, 3)
x, x.shape
>>>
(tensor([[[1, 2, 3],
          [4, 5, 6],
          [7, 8, 9]]]),
 torch.Size([1, 3, 3]))
```

Indexing values goes outer dimension -> inner dimension (check out the square brackets).

```
# Let's index bracket by bracket
print(f"First square bracket:\n{x[0]}")
print(f"Second square bracket: {x[0][0]}")
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
```

7. Indexing(selecting data from tensors)



You can also use : to specify "all values in this dimension" and then use a comma (,) to add another dimension.

```
# Get all values of 0th dimension and the 0 index of 1st dimension
x[:, 0]
>>>
tensor([[1, 2, 3]])
```

```
# Get all values of 0th & 1st dimensions but only index 1 of 2nd dimension
x[:, :, 1]
>>>
tensor([[2, 5, 8]])
```

```
# Get all values of the 0 dimension but only the 1 index value of the 1st and 2nd dimension
x[:, 1, 1]
>>>
tensor([5])
```

```
# Get index 0 of 0th and 1st dimension and all values of 2nd dimension
x[0, 0, :] # same as x[0][0]
>>>
tensor([1, 2, 3])
```

Indexing can be quite confusing to begin with, especially with larger tensors (I still have to try indexing multiple times to get it right). But with a bit of practice and following the data explorer's motto (visualize, visualize, visualize), you'll start to get the hang of it.

7. Indexing(selecting data from tensors)

Joining a list of tensors together with torch.cat

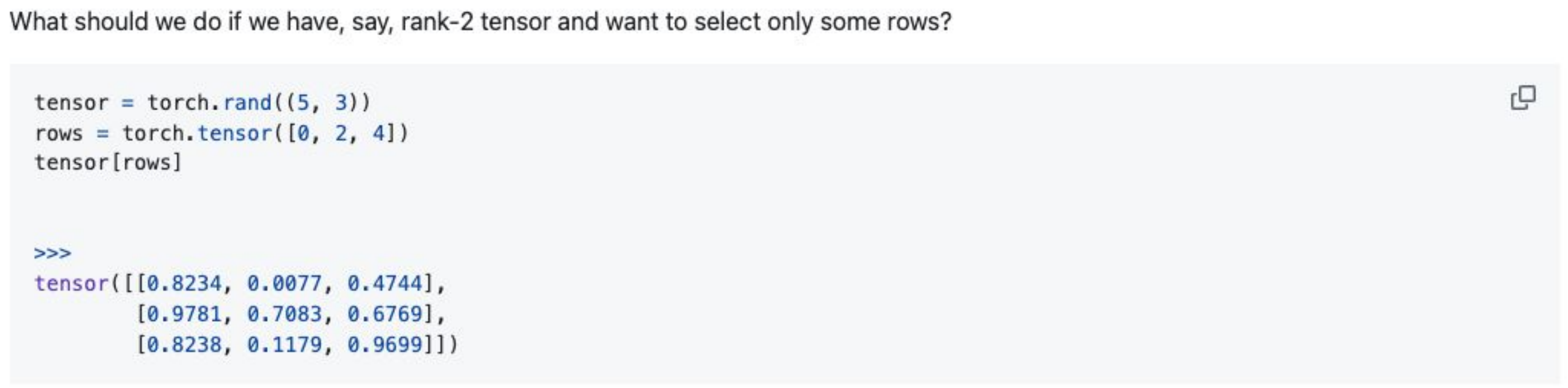
```
a = torch.zeros(3, 2)
b = torch.ones(3, 2)
print(torch.cat((a, b), dim=0))
print(torch.cat((a, b), dim=1))


tensor([[0., 0.],
        [0., 0.],
        [0., 0.],
        [1., 1.],
        [1., 1.],
        [1., 1.]])
tensor([[0., 0., 1., 1.],
        [0., 0., 1., 1.],
        [0., 0., 1., 1.]])
```

Indexing with another tensor/array:

```
a = torch.arange(start=0, end=10)
indices = np.arange(0, 10) > 5
print(a)
print(indices)
print(a[indices])

indices = torch.arange(start=0, end=10) % 5
print(indices)
print(a[indices])


tensor([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
[False False False False False False  True  True  True  True]
tensor([6, 7, 8, 9])
tensor([0, 1, 2, 3, 4, 0, 1, 2, 3, 4])
tensor([0, 1, 2, 3, 4, 0, 1, 2, 3, 4])
```

What should we do if we have, say, rank-2 tensor and want to select only some rows?

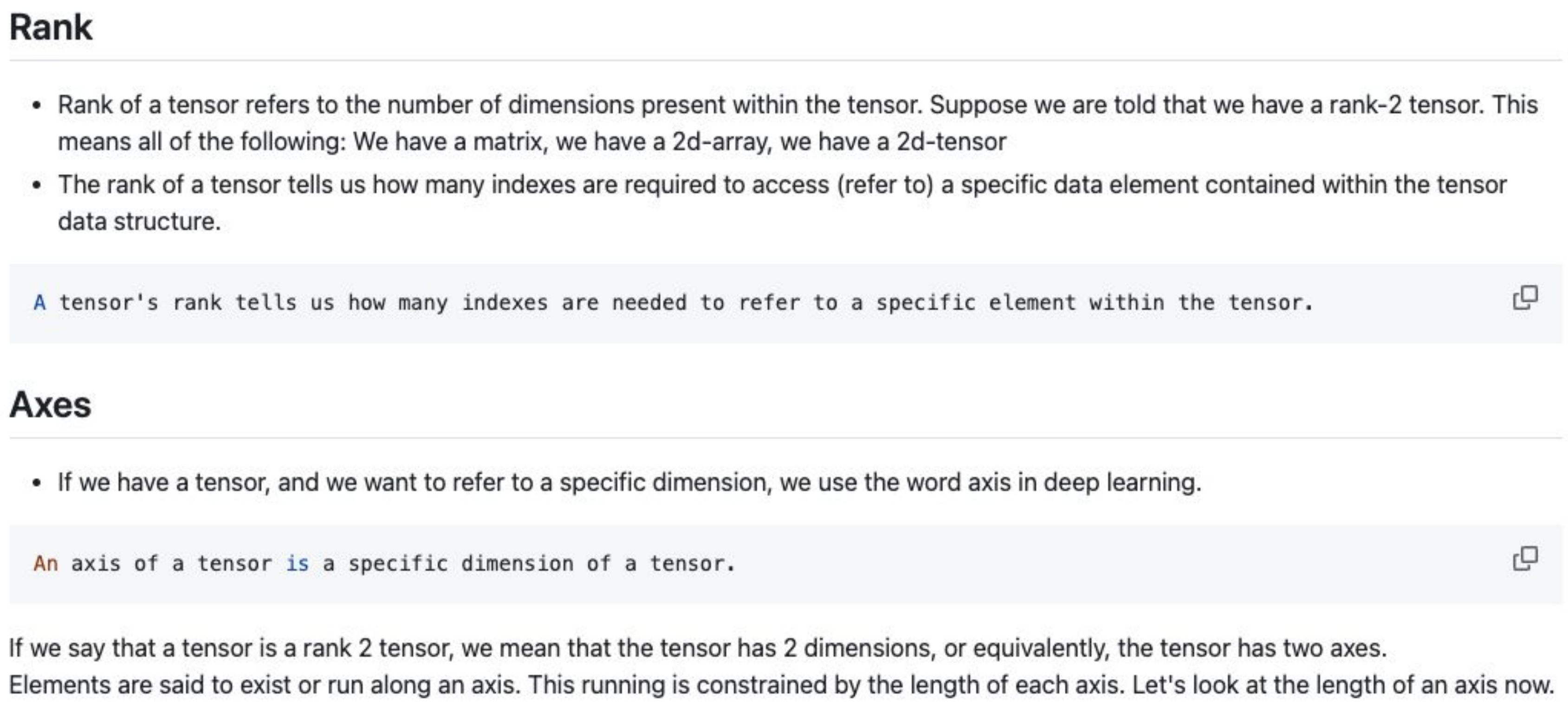
7. Indexing(selecting data from tensors)


What should we do if we have, say, rank-2 tensor and want to select only some rows?

```
tensor = torch.rand((5, 3))
rows = torch.tensor([0, 2, 4])
tensor[rows]
```

```
>>>
tensor([[0.8234, 0.0077, 0.4744],
        [0.9781, 0.7083, 0.6769],
        [0.8238, 0.1179, 0.9699]])
```

8. Rank, Axes, Shape



Rank

- Rank of a tensor refers to the number of dimensions present within the tensor. Suppose we are told that we have a rank-2 tensor. This means all of the following: We have a matrix, we have a 2d-array, we have a 2d-tensor
- The rank of a tensor tells us how many indexes are required to access (refer to) a specific data element contained within the tensor data structure.

```
A tensor's rank tells us how many indexes are needed to refer to a specific element within the tensor.
```

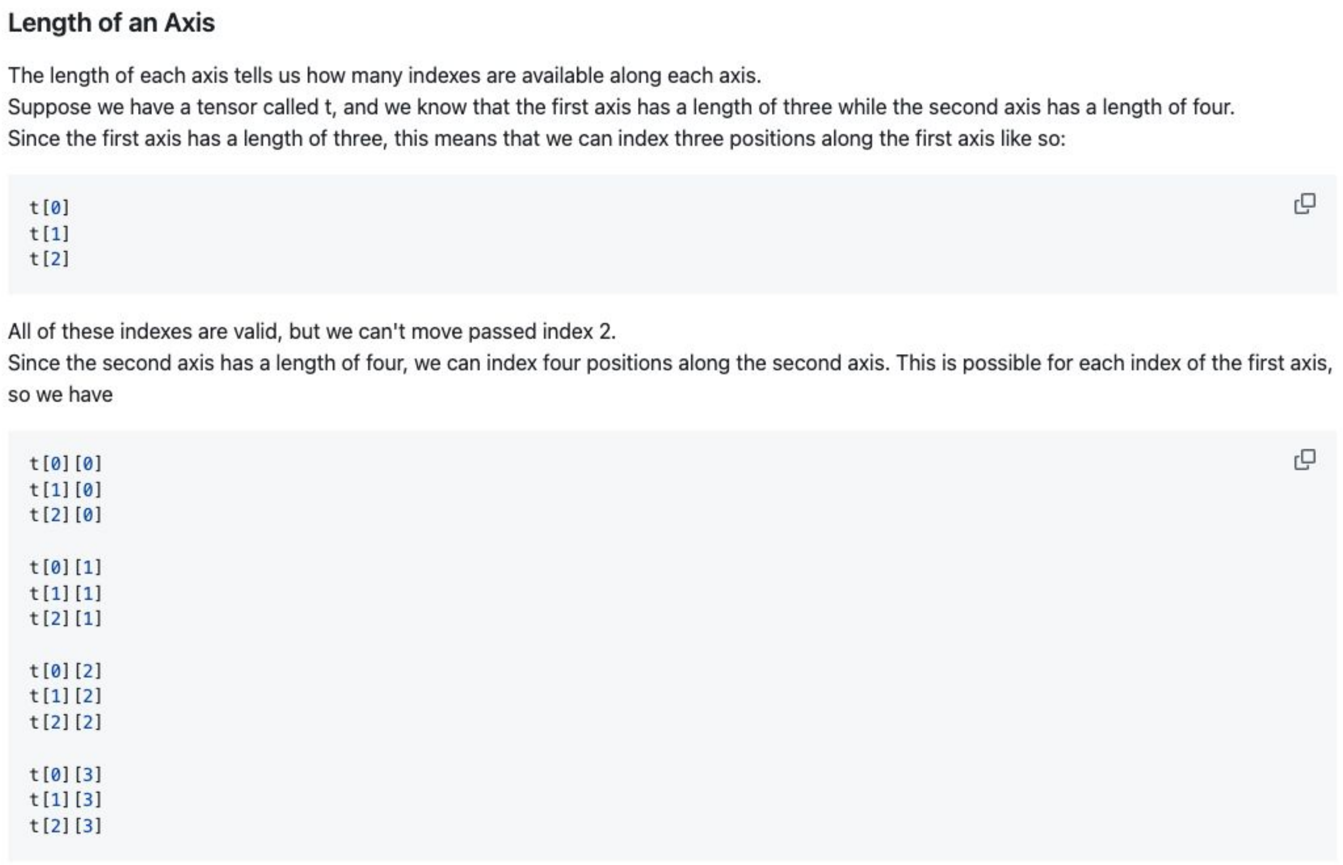
Axes

- If we have a tensor, and we want to refer to a specific dimension, we use the word axis in deep learning.

```
An axis of a tensor is a specific dimension of a tensor.
```

If we say that a tensor is a rank 2 tensor, we mean that the tensor has 2 dimensions, or equivalently, the tensor has two axes.
Elements are said to exist or run along an axis. This running is constrained by the length of each axis. Let's look at the length of an axis now.

8. Rank, Axes, Shape



Length of an Axis

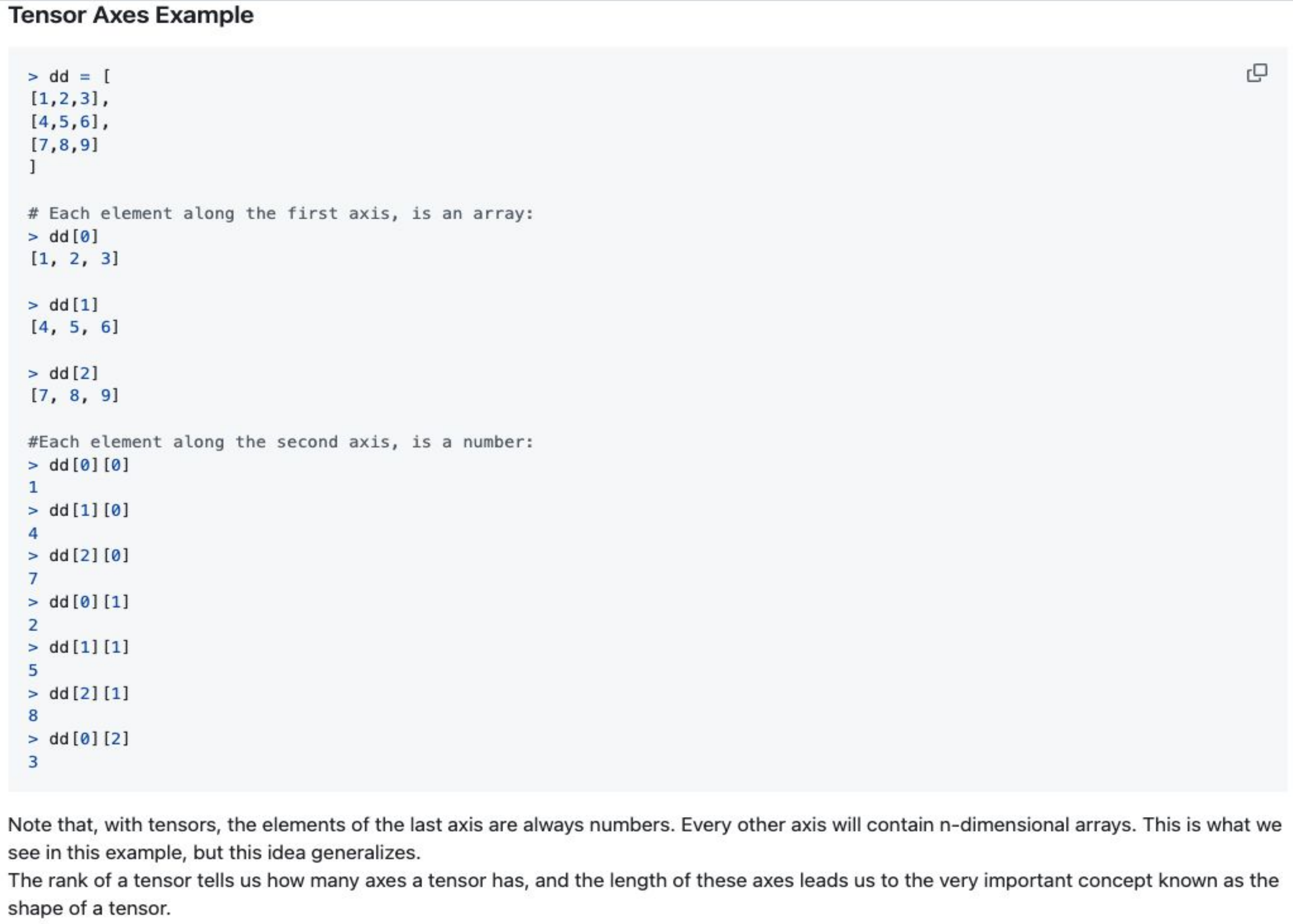
The length of each axis tells us how many indexes are available along each axis.
Suppose we have a tensor called t, and we know that the first axis has a length of three while the second axis has a length of four.
Since the first axis has a length of three, this means that we can index three positions along the first axis like so:

```
t[0]
t[1]
t[2]
```

All of these indexes are valid, but we can't move passed index 2.
Since the second axis has a length of four, we can index four positions along the second axis. This is possible for each index of the first axis, so we have

```
t[0][0]
t[1][0]
t[2][0]

t[0][1]
t[1][1]
t[2][1]

t[0][2]
t[1][2]
t[2][2]

t[0][3]
t[1][3]
t[2][3]
```

8. Rank, Axes, Shape

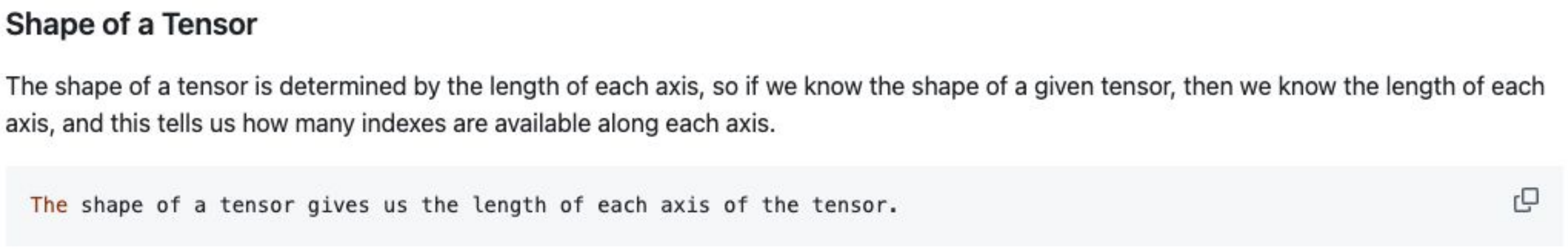


Tensor Axes Example

```
> dd = [
[1,2,3],
[4,5,6],
[7,8,9]
]

# Each element along the first axis, is an array:
> dd[0]
[1, 2, 3]

> dd[1]
[4, 5, 6]

> dd[2]
[7, 8, 9]

#Each element along the second axis, is a number:
> dd[0][0]
1
> dd[1][0]
4
> dd[2][0]
7
> dd[0][1]
2
> dd[1][1]
5
> dd[2][1]
8
> dd[0][2]
3
```

Note that, with tensors, the elements of the last axis are always numbers. Every other axis will contain n-dimensional arrays. This is what we see in this example, but this idea generalizes.
The rank of a tensor tells us how many axes a tensor has, and the length of these axes leads us to the very important concept known as the shape of a tensor.

8. Rank, Axes, Shape



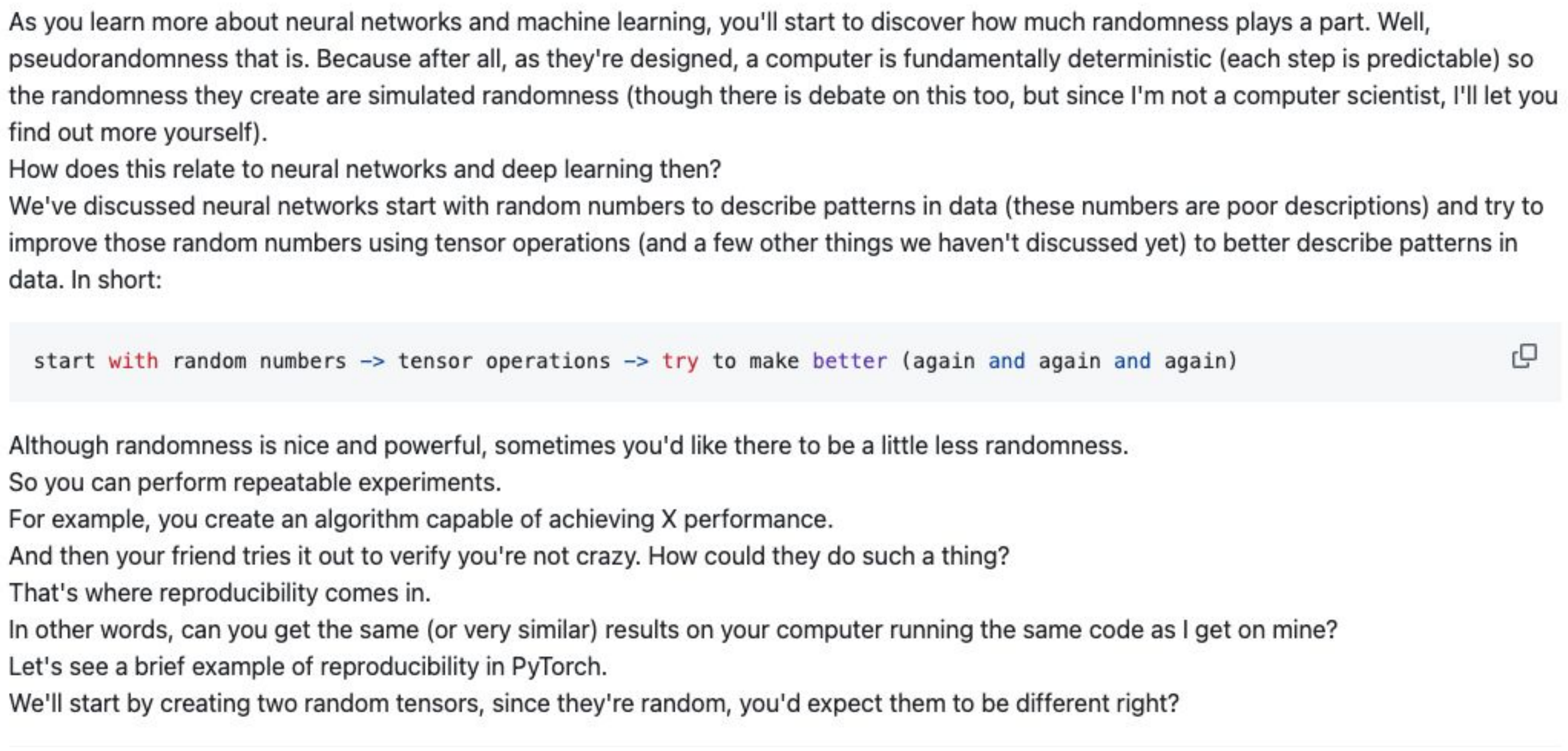
Shape of a Tensor

The shape of a tensor is determined by the length of each axis, so if we know the shape of a given tensor, then we know the length of each axis, and this tells us how many indexes are available along each axis.

```
The shape of a tensor gives us the length of each axis of the tensor.
```

More here: https://deeplizard.com/learn/video/AiyK0idr4uM

9. Reproducibility (trying to take the random out of random



As you learn more about neural networks and machine learning, you'll start to discover how much randomness plays a part. Well, pseudorandomness that is. Because after all, as they're designed, a computer is fundamentally deterministic (each step is predictable) so the randomness they create are simulated randomness (though there is debate on this too, but since I'm not a computer scientist, I'll let you find out more yourself).

How does this relate to neural networks and deep learning then?

We've discussed neural networks start with random numbers to describe patterns in data (these numbers are poor descriptions) and try to improve those random numbers using tensor operations (and a few other things we haven't discussed yet) to better describe patterns in data. In short:

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

Although randomness is nice and powerful, sometimes you'd like there to be a little less randomness.

So you can perform repeatable experiments.

For example, you create an algorithm capable of achieving X performance.

And then your friend tries it out to verify you're not crazy. How could they do such a thing?

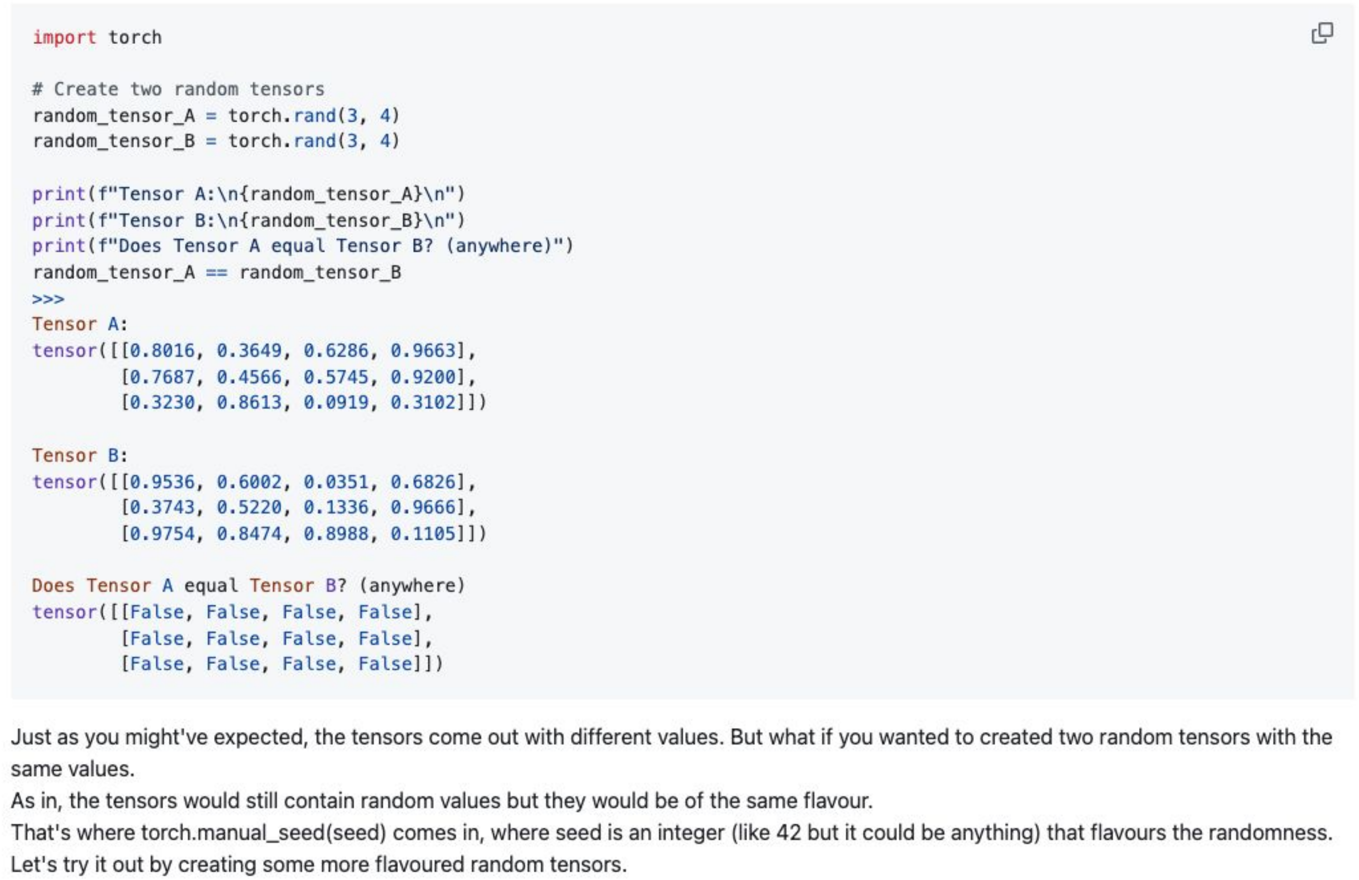
That's where reproducibility comes in.

In other words, can you get the same (or very similar) results on your computer running the same code as I get on mine?

Let's see a brief example of reproducibility in PyTorch.

We'll start by creating two random tensors, since they're random, you'd expect them to be different right?

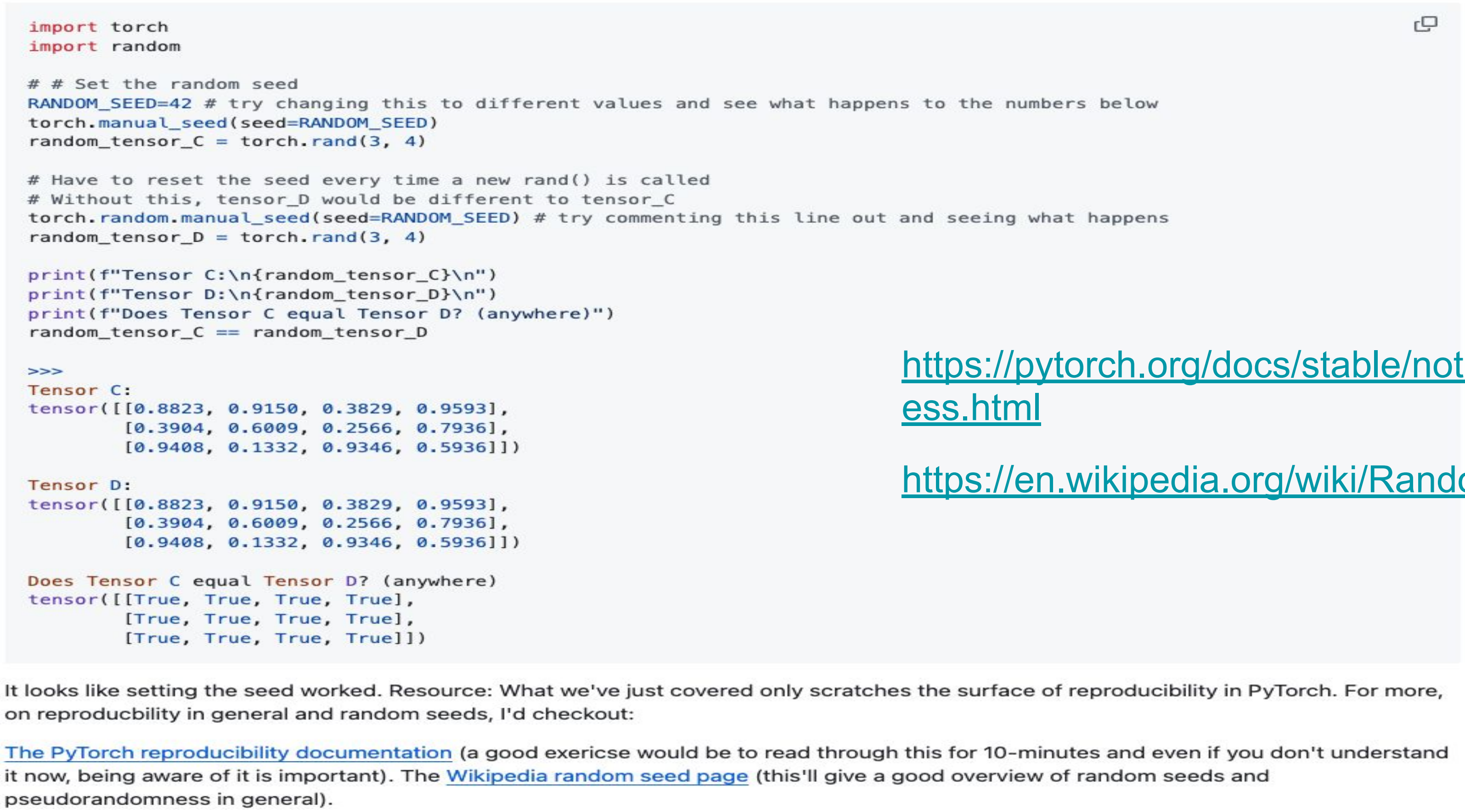
9. Reproducibility (trying to take the random out of random



```python
import torch

# Create two random tensors
random_tensor_A = torch.rand(3, 4)
random_tensor_B = torch.rand(3, 4)

print(f"Tensor A:\n{random_tensor_A}\n")
print(f"Tensor B:\n{random_tensor_B}\n")
print(f"Does Tensor A equal Tensor B? (anywhere)")
random_tensor_A == random_tensor_B
>>>
Tensor A:
tensor([[0.8016, 0.3649, 0.6286, 0.9663],
        [0.7687, 0.4566, 0.5745, 0.9200],
        [0.3230, 0.8613, 0.0919, 0.3102]])

Tensor B:
tensor([[0.9536, 0.6002, 0.0351, 0.6826],
        [0.3743, 0.5220, 0.1336, 0.9666],
        [0.9754, 0.8474, 0.8988, 0.1105]])

Does Tensor A equal Tensor B? (anywhere)
tensor([[False, False, False, False],
        [False, False, False, False],
        [False, False, False, False]])
```

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.
Let's try it out by creating some more flavoured random tensors.

9. Reproducibility (trying to take the random out of random



```python
import torch
import random

# # Set the random seed
RANDOM_SEED=42 # try changing this to different values and see what happens to the numbers below
torch.manual_seed(seed=RANDOM_SEED)
random_tensor_C = torch.rand(3, 4)

# Have to reset the seed every time a new rand() is called
# Without this, tensor_D would be different to tensor_C
torch.random.manual_seed(seed=RANDOM_SEED) # try commenting this line out and seeing what happens
random_tensor_D = torch.rand(3, 4)

print(f"Tensor C:\n{random_tensor_C}\n")
print(f"Tensor D:\n{random_tensor_D}\n")
print(f"Does Tensor C equal Tensor D? (anywhere)")
random_tensor_C == random_tensor_D

>>>
Tensor C:
tensor([[0.8823, 0.9150, 0.3829, 0.9593],
        [0.3904, 0.6009, 0.2566, 0.7936],
        [0.9408, 0.1332, 0.9346, 0.5936]])

Tensor D:
tensor([[0.8823, 0.9150, 0.3829, 0.9593],
        [0.3904, 0.6009, 0.2566, 0.7936],
        [0.9408, 0.1332, 0.9346, 0.5936]])

Does Tensor C equal Tensor D? (anywhere)
tensor([[True, True, True, True],
        [True, True, True, True],
        [True, True, True, True]])
```

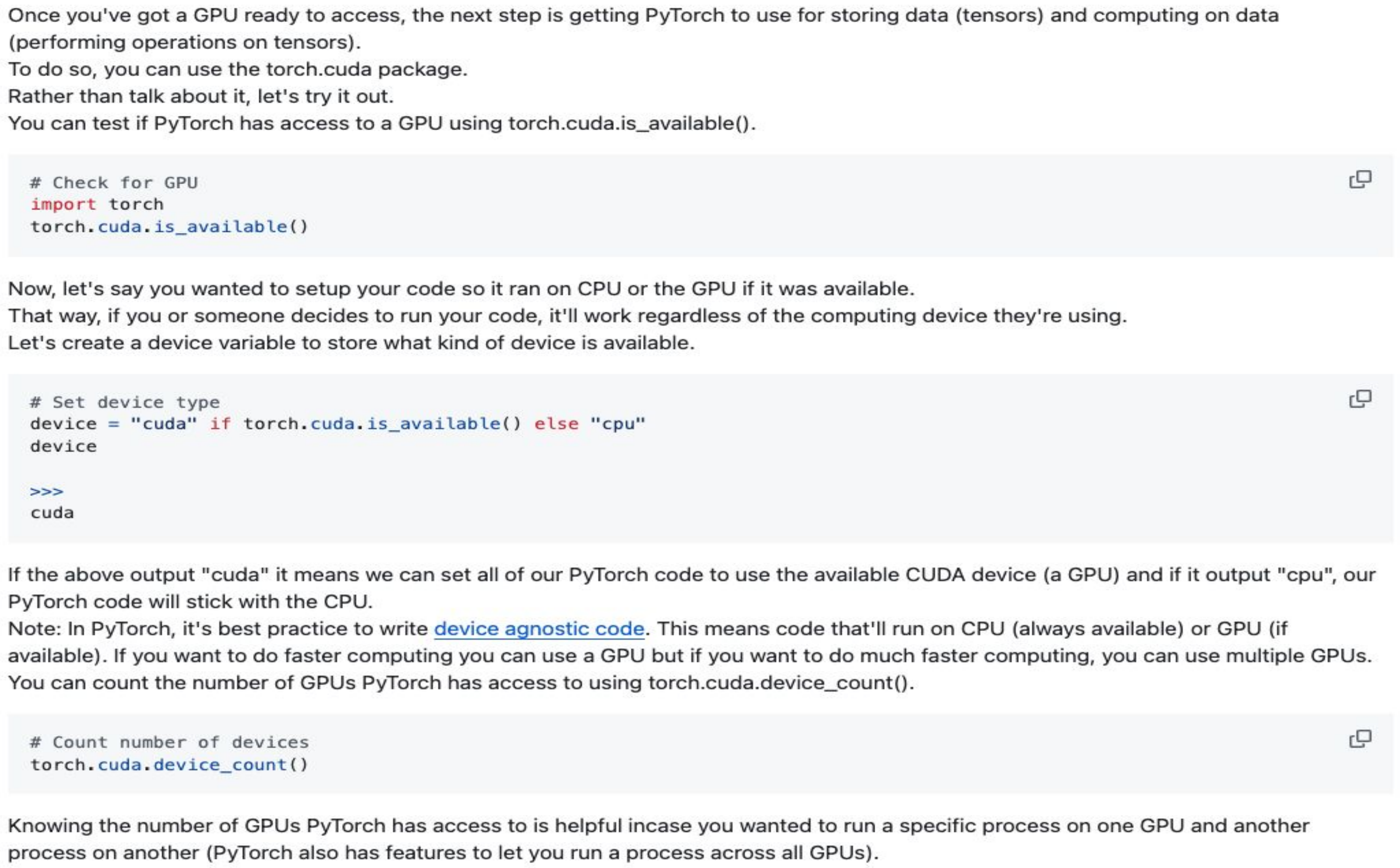
It looks like setting the seed worked. Resource: What we've just covered only scratches the surface of reproducibility in PyTorch. For more, on reproducbility in general and random seeds, I'd checkout:

The PyTorch reproducibility documentation (a good exericse would be to read through this for 10-minutes and even if you don't understand it now, being aware of it is important). The Wikipedia random seed page (this'll give a good overview of random seeds and pseudorandomness in general).

https://pytorch.org/docs/stable/notes/randomness.html

https://en.wikipedia.org/wiki/Random_seed

10. Getting Pytorch to run on the GPU



Once you've got a GPU ready to access, the next step is getting PyTorch to use for storing data (tensors) and computing on data (performing operations on tensors).
To do so, you can use the torch.cuda package.
Rather than talk about it, let's try it out.
You can test if PyTorch has access to a GPU using torch.cuda.is_available().

```
# Check for GPU
import torch
torch.cuda.is_available()
```

Now, let's say you wanted to setup your code so it ran on CPU or the GPU if it was available.
That way, if you or someone decides to run your code, it'll work regardless of the computing device they're using.
Let's create a device variable to store what kind of device is available.

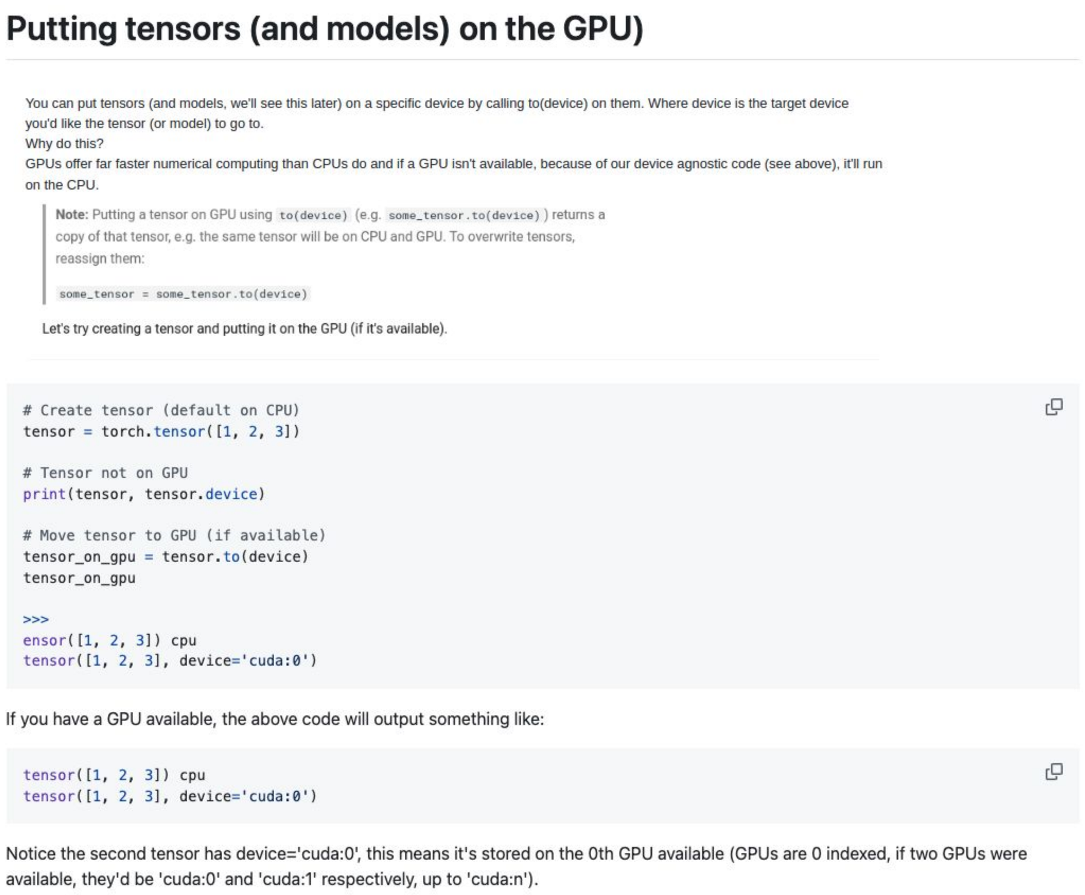
```
# Set device type
device = "cuda" if torch.cuda.is_available() else "cpu"
device

>>>
cuda
```

If the above output "cuda" it means we can set all of our PyTorch code to use the available CUDA device (a GPU) and if it output "cpu", our PyTorch code will stick with the CPU.
Note: In PyTorch, it's best practice to write device agnostic code. This means code that'll run on CPU (always available) or GPU (if available). If you want to do faster computing you can use a GPU but if you want to do much faster computing, you can use multiple GPUs.
You can count the number of GPUs PyTorch has access to using torch.cuda.device_count().

```
# Count number of devices
torch.cuda.device_count()
```

Knowing the number of GPUs PyTorch has access to is helpful incase you wanted to run a specific process on one GPU and another process on another (PyTorch also has features to let you run a process across all GPUs).

10. Getting Pytorch to run on the GPU



Putting tensors (and models) on the GPU)

You can put tensors (and models, we'll see this later) on a specific device by calling to(device) on them. Where device is the target device you'd like the tensor (or model) to go to.
Why do this?
GPUs offer far faster numerical computing than CPUs do and if a GPU isn't available, because of our device agnostic code (see above), it'll run on the CPU.

> **Note:** Putting a tensor on GPU using `to(device)` (e.g. `some_tensor.to(device)`) returns a copy of that tensor, e.g. the same tensor will be on CPU and GPU. To overwrite tensors, reassign them:
>
> `some_tensor = some_tensor.to(device)`

Let's try creating a tensor and putting it on the GPU (if it's available).

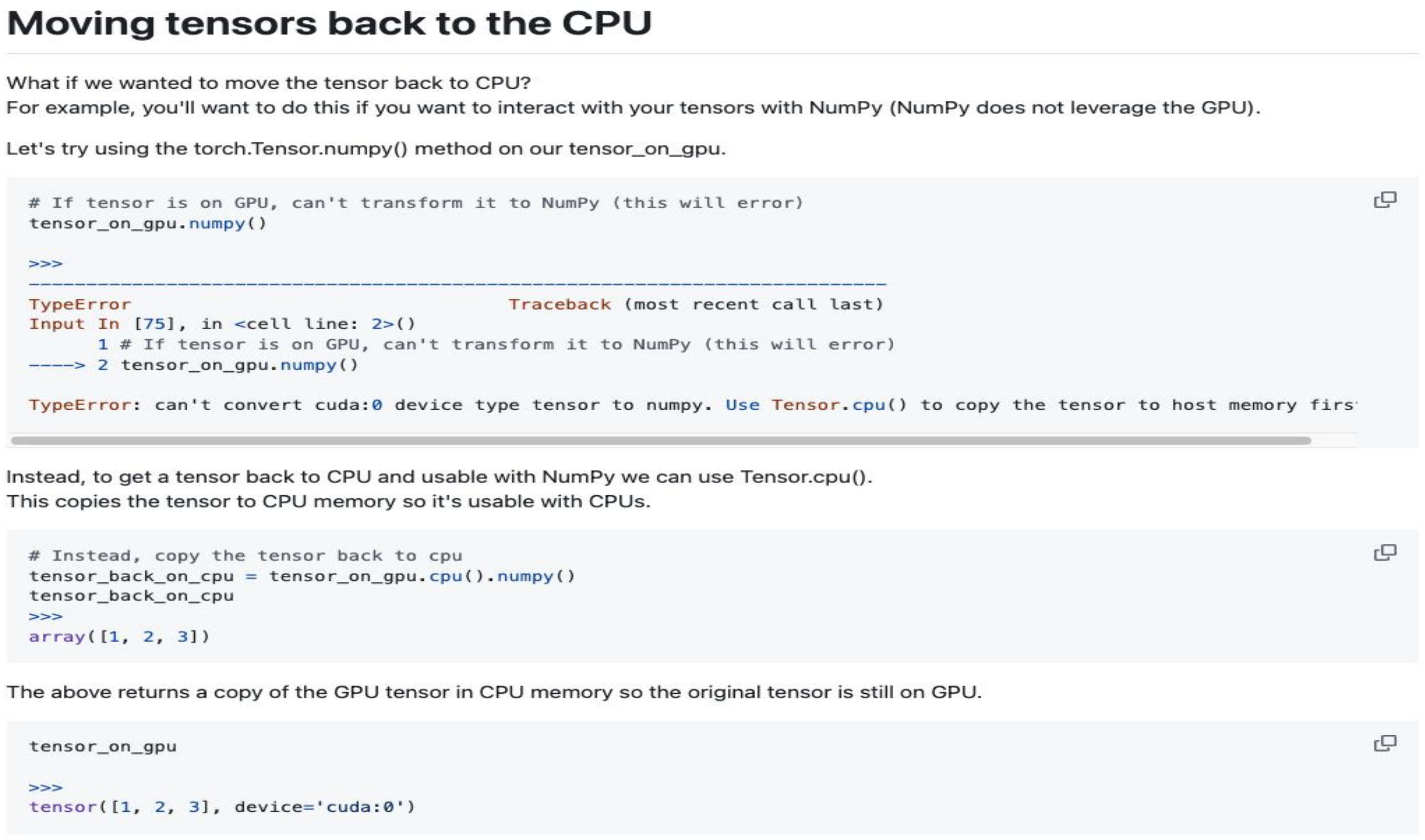
```
# Create tensor (default on CPU)
tensor = torch.tensor([1, 2, 3])

# Tensor not on GPU
print(tensor, tensor.device)

# Move tensor to GPU (if available)
tensor_on_gpu = tensor.to(device)
tensor_on_gpu

>>>
ensor([1, 2, 3]) cpu
tensor([1, 2, 3], device='cuda:0')
```

If you have a GPU available, the above code will output something like:

```
tensor([1, 2, 3]) cpu
tensor([1, 2, 3], device='cuda:0')
```

Notice the second tensor has device='cuda:0', this means it's stored on the 0th GPU available (GPUs are 0 indexed, if two GPUs were available, they'd be 'cuda:0' and 'cuda:1' respectively, up to 'cuda:n').

10. Getting Pytorch to run on the GPU



Moving tensors back to the CPU

What if we wanted to move the tensor back to CPU?
For example, you'll want to do this if you want to interact with your tensors with NumPy (NumPy does not leverage the GPU).

Let's try using the torch.Tensor.numpy() method on our tensor_on_gpu.

```
# If tensor is on GPU, can't transform it to NumPy (this will error)
tensor_on_gpu.numpy()

>>>
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Input In [75], in <cell line: 2>()
      1 # If tensor is on GPU, can't transform it to NumPy (this will error)
----> 2 tensor_on_gpu.numpy()

TypeError: can't convert cuda:0 device type tensor to numpy. Use Tensor.cpu() to copy the tensor to host memory firs
```

Instead, to get a tensor back to CPU and usable with NumPy we can use Tensor.cpu().
This copies the tensor to CPU memory so it's usable with CPUs.

```
# Instead, copy the tensor back to cpu
tensor_back_on_cpu = tensor_on_gpu.cpu().numpy()
tensor_back_on_cpu
>>>
array([1, 2, 3])
```

The above returns a copy of the GPU tensor in CPU memory so the original tensor is still on GPU.

```
tensor_on_gpu

>>>
tensor([1, 2, 3], device='cuda:0')
```