torch.Tensor:

generates a tensor with the given numbers

```
T = torch.Tensor([ [1, 2, 3], [4, 5, 6], [7, 8, 9] ])
```

Gives:

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

How torch.tensor works is that the function takes a python list or a numpy array and converts it to a torch.tensor datatype by copying the contents of the input (python list or numpy array). That is why we put brackets ("[]") on the outside of:

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

When writing

```
T = torch.Tensor([ [1, 2, 3], [4, 5, 6], [7, 8, 9] ])
```

Tensor.ndim:

returns the number of dimensions of the Tensor which is an attribute of the torch.tensor type objects (not a function).

If you think of the tensor X as an multidimensional array (like a 2D array in java) ndim is the number of dimensions of the multidimensional array X.

The dimension of an array can simply be defined as the number of subscripts or indices required to specify a particular element of the array.

X.shape:

Returns the shape of the tensor X. It is an attribute of the torch.tensor class.

X.size():

Returns the shape of the tensor X. It is a function of the torch.tensor class which returns the tensor's shape attribute.

torch.rand(size=(a, b, c)):

Returns a tensor with randomly generated values from [0, 1) with the given shape. Also you can give the size as a tuple as:

Torch.rand(size=(a, b, c))

You can also give only numbers representing the shape of the tensor as:

Torch.rand(a, b, c)

Which gives the same thing.

Ex:

x = torch.rand(size=(4,3)) gives you a 4x3 matrix of randomly generated numbers.

torch.matmul(A,B)

If the matrix A is of shape CxD and matrix B is of DxC shape, torch.matmul(A,B) returns the cross product of A and B as AxB matrix. It is important to remember that The behavior of the torch.matmul function depends on the dimensionality of the input tensors A and B.

torch.zeros(size=(a, b))

generates a matrix of zeros of the given size

You can also write

```
torch.zeros(a, b)
```

or

```
t = (a, b)
x = torch.zeros(a)
```

where t is a tuple

torch.ones(size=(a, b))

generates a matrix of ones of the given size

You can also write

```
torch.ones(a, b)
```

or

```
t = (a, b)
x = torch.ones(a)
```

where t is a tuple

torch.arange(start=a, end=b, step=c)

generates an array (1xN tensor) of numbers ranging from a to b with step size of c. a is included, b is not included.

```
x = torch.arange(start=3, end=33, step=5)
```

gives the output:

```
tensor([ 3, 8, 13, 18, 23, 28])
```

You can also use:

```
x = torch.arange(a, b, c)
```

which gives the same results.

If you don’t indicate step, it assumes step=1 as default:

```
x = torch.arange(3, 6)
```

gives:

```
tensor([3, 4, 5])
```

And if you don’t indicate start, it assumes start=0 as default:

```
x = torch.arange(6)
```

gives:

```
tensor([0, 1, 2, 3, 4, 5])
```

Note: torch.arange had the name torch.range in the previous versions of pytorch. They changed the syntax from torch.range to torch.arange. Only the syntax changed, it does the exact same thing.

torch.zeros_like(T):

generates a tensor of zeros in the shape of the given tensor T.

For example, if the tensor T is of size 3x7, torch. zeros_like(T) generates a matrix of zeros of the size 3x2.

```
T = torch.rand(3, 2)
xz = torch.zeros_like(T)
```

gives:

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

torch.ones_like(T):

generates a tensor of ones in the shape of the given tensor T.

For example, if the tensor T is of size 3x7, torch. ones_like(T) generates a matrix of ones of the size 3x2.

```
T = torch.rand(3, 2)
xz = torch.ones_like(T)
```

gives:

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

tensor.dtype:

gives the data type of the data of the tensor. dtype is a attribute of the tensor class.

It makes the dtype=torch.float32 as default if not specified. You can define the datatype as:

```
T = torch.rand(3, 2, dtype=torch.float16)
```

For example. And you can check the datatype by:

```
T.dtype
```

Which gives the ouput:

```
torch.float16
```

tensor.type()

Gives the given tensor but with the specified data type. For example:

```
MyTensor = torch.rand(3, 2, dtype=torch.float16)
MyTensor2 = MyTensor.type(torch.float64)
MyTensor2.dtype
```

Gives the output:

```
torch.float64
```

Using GPUs

tensor.device:

It is an attribute of the tensor class showing which device the tensor is on. This could be CPU, GPU, TPU etc.

```
MyTensor.device
```

torch.cuda.is_available():

```
torch.cuda.is_available()
```

gives if cuda gpu is available for you to use. If this gives true, you can use it. If this gives false, you cannot use cuda right now. To be able to use it, set GPU via Edit → Notebook Settings. Then run the imports and you should be able to use cuda right now.

You can also go Runtime→ Change runtime type→Hardware accelerator→ select gpu. This does the same thing as Edit → Notebook Settings

```
MyTensor = torch.rand(3, 2, device=torch.device("cuda"))
```

Gives:

```
tensor([[0.5452, 0.2887], [0.1943, 0.9144], [0.2411, 0.2569]],
device='cuda:0')
```

```
MyTensor.device
```

Gives the output:

```
device(type='cuda', index=0)
```

Since I defined the device of MyTensor as cuda.

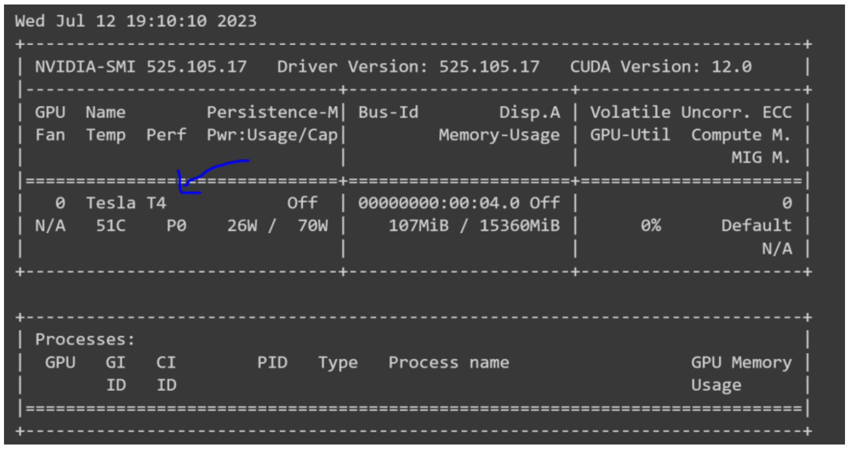
Note: Cuda is nvidia's gpu interface

!nvidia-smi

You can look up the exact gpu model you are using this command.

```
!nvidia-smi
```

Gives:


```
Wed Jul 12 19:10:10 2023
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 525.105.17   Driver Version: 525.105.17   CUDA Version: 12.0     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|                               |                      |               MIG M. |
|===============================+======================+======================|
|   0  Tesla T4            Off  | 00000000:00:04.0 Off |                    0 |
| N/A   51C    P0    26W /  70W |    107MiB / 15360MiB |      0%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                                  |
|  GPU   GI   CI        PID   Type   Process name                  GPU Memory |
|        ID   ID                                                   Usage      |
|=============================================================================|
+-----------------------------------------------------------------------------+
```

Here we can see that the gpu I am using is a Tesla T4.

Torch.mul(Tensor, a):

Just element wise multiplication. Returns the tensor: “ Tensor*a “. Exactly equal to the “ * ” sign.

```
MyTensor = torch.tensor([ [1, 2, 3], [4, 5, 6], [7, 8, 9]
])
MyTensor2 = torch.mul(MyTensor, 10)
MyTensor2
```

Gives:

```
tensor([[10, 20, 30],
        [40, 50, 60],
        [70, 80, 90]])
```

Also works with element-wise multiplication among tensors:

```
MyTensor = torch.tensor([ [1, 2, 3], [4, 5, 6] ])
MyTensor2 = torch.tensor([ [2, 2, 2], [4, 5, 6] ])
torch.mul(MyTensor, MyTensor2)
```

Gives:

```
tensor([[ 2, 4, 6],
        [16, 25, 36]])
```

torch.matmul(Tensor1, Tensor2)

Matrix product of two tensors. matmul stands for matrix multiplication.

The behavior depends on the dimensionality of the tensors as explained in the link below:

https://pytorch.org/docs/stable/generated/torch.matmul.html

```
MyTensor = torch.tensor([ [1, 2, 3], [4, 5, 6] ])
MyTensor2 = torch.tensor([ [2, 2], [3, 3], [4, 4] ])
torch.matmul(MyTensor, MyTensor2)
```

Gives:

```
tensor([[20, 20],
        [47, 47]])
```

Note: torch.mm(Tensor1, Tensor2) is the exact same thing as torch.matmul(Tensor1, Tensor2)

Tensor.T

Gives the transpose of the tensor. You write it with capital letter without a paranthesis.

```
a = torch.rand(2,3)
a
```

Gives:

```
tensor([[0.5075, 0.3229, 0.3319],
        [0.6085, 0.4522, 0.9616]])
```

```
b = a.T
b
```

Gives:

```
tensor([[0.5075, 0.6085],
        [0.3229, 0.4522],
        [0.3319, 0.9616]])
```

Torch.set_printoptions(precision=None, threshold=None, edgeitems=None, linewidth=None, profile=None, sci_mode=None)

It is used to set how you print sensors, the syntax. To explain things in this part, let MyTensor be defined as:

```
MyTensor = 100 * torch.rand(1, 7)
```

```
MyTensor
```
Which gives:

```
tensor([[80.7729, 97.2969, 86.2811, 83.5691, 58.8853, 6.3473, 58.4478]])
```

precision: how many digits will there in the decimal part of the representation of the number, how many digits will there be after the dot, “.”.

```
torch.set_printoptions(precision=2)
MyTensor
```
Gives:

```
tensor([[80.77, 97.30, 86.28, 83.57, 58.89, 6.35, 58.45]])
```

Note: **Summarization** in pytorch printing tensors is that only the first and last 3 elements(the number of element at the start of end can be changed with the edgeitems parameter described below) of the tensor is shown and other elements are represented by three dots, “…”.

but with summarization, it is represented as:

```
tensor([[80.7729, 97.2969, 86.2811, ..., 58.8853, 6.3473, 58.4478]])
```

Normally, the amount of elements, a tensor should have for pytorch to represent tensors with summarization is 1000 by Default but you can change the number with the threshold parameter given below.

Note: the number of elements that a tensor have should be more than twice the edgeitems and it should be more than the threshold for the tensor to be represented with summarization.

Edgeitems: The number of elements shown at the start and end of a tensor representation with summarization. It is set to 3 by default but you can change it as:

```
torch.set_printoptions(edgeitems=1)
```
Now, only 1 element at the start and end will be given with representation of a tensor with summarization.

```
tensor([[80.7729, ..., 58.4478]])
```

Note: the number of elements that a tensor have should be more than twice the edgeitems and it should be more than the threshold for the tensor to be represented with summarization.

Threshold: Sets the number of elements that a tensor should have for pytorch to print out tensors with summarization.

```
torch.set_printoptions(threshold=4)
MyTensor
```
Sets the threshold for printing tensors with threshold to 4. Now, the tensors which have more than 4 elements will be represented with summarization. The tensors which have less than or equal to 4 elements will be represented without summarization as:

```
MyTensor
```
Is represented with summarization as:

```
tensor([[80.7729, 97.2969, 86.2811, ..., 58.8853, 6.3473, 58.4478]])
```

while myTensor2 which has only 3 elements is represented normally, without summarization as:

```
myTensor2 = torch.rand(1,3)
myTensor2
```
Gives:

```
tensor([[0.5417, 0.2068, 0.3388]])
```

The same principles also applies for 2D tensors as:

```
myTensor3 = torch.rand(2, 2)
myTensor3
```
Gives:

```
tensor([[0.2924, 0.6989],
        [0.4625, 0.2115]])
```

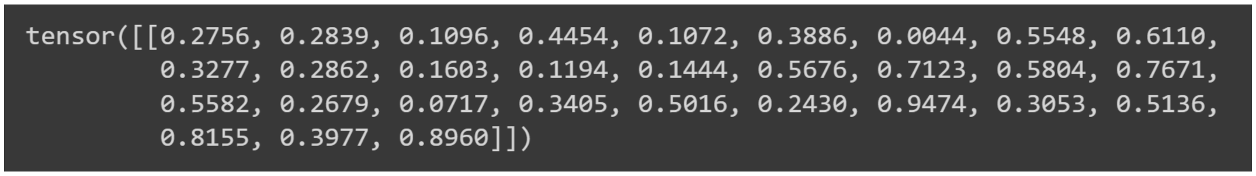
Or

```
myTensor3 = torch.rand(8, 8)
torch.set_printoptions(threshold=3)
myTensor3
```
Gives:
```
tensor([[0.4942, 0.0380, 0.2591, ..., 0.1231, 0.8324, 0.9810],
        [0.9214, 0.4088, 0.8844, ..., 0.1212, 0.6696, 0.8678],
        [0.2182, 0.5740, 0.6916, ..., 0.4163, 0.1514, 0.7151],
        ...,
        [0.2060, 0.1811, 0.9974, ..., 0.6618, 0.6597, 0.2418],
        [0.1322, 0.2776, 0.2266, ..., 0.4011, 0.5801, 0.7348],
        [0.8397, 0.5224, 0.1391, ..., 0.9419, 0.5239, 0.1312]])
```

Note: the number of elements that a tensor have should be more than twice the edgeitems and it should be more than the threshold for the tensor to be represented with summarization.

Linewidth: The number of characters per line for the purpose of inserting line breaks (default = 80). Thresholded (summarization) matrices will ignore this parameter and break the line before 80 characters.

```
myTensor3 = torch.rand(1, 30)
torch.set_printoptions(linewidth=80)

myTensor3
```
Gives:



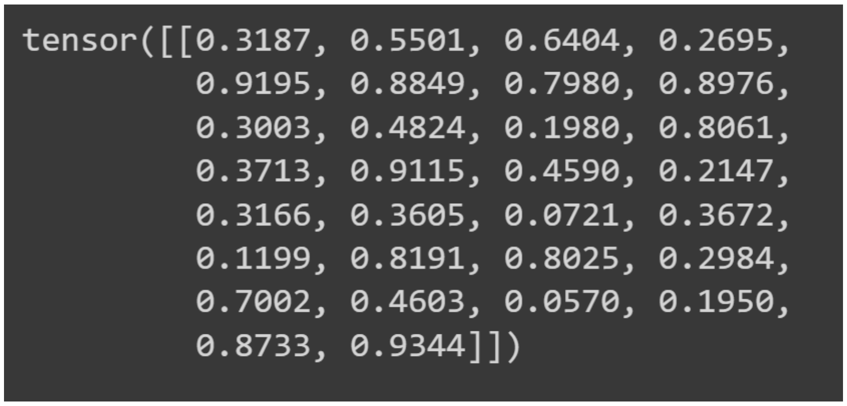
```
tensor([[0.2756, 0.2839, 0.1096, 0.4454, 0.1072, 0.3886, 0.0044, 0.5548, 0.6110,
         0.3277, 0.2862, 0.1603, 0.1194, 0.1444, 0.5676, 0.7123, 0.5804, 0.7671,
         0.5582, 0.2679, 0.0717, 0.3405, 0.5016, 0.2430, 0.9474, 0.3053, 0.5136,
         0.8155, 0.3977, 0.8960]])
```

Where it goes over to the next line after the 9th element in a row because the writing the 10th element in the same row would go over the linewidth of 80.

If we set the linewidth to 40 as:

```
myTensor3 = torch.rand(1, 30)
torch.set_printoptions(linewidth=40)
myTensor3
```

We get:

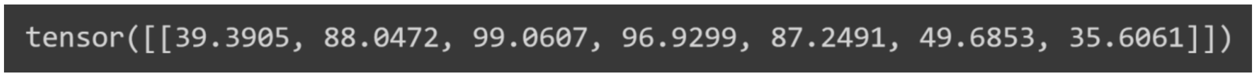


```
tensor([[0.3187, 0.5501, 0.6404, 0.2695,
         0.9195, 0.8849, 0.7980, 0.8976,
         0.3003, 0.4824, 0.1980, 0.8061,
         0.3713, 0.9115, 0.4590, 0.2147,
         0.3166, 0.3605, 0.0721, 0.3672,
         0.1199, 0.8191, 0.8025, 0.2984,
         0.7002, 0.4603, 0.0570, 0.1950,
         0.8733, 0.9344]])
```

Profile: Used for pretty printing. Can override with any of the above options. (any one of default, short, full)

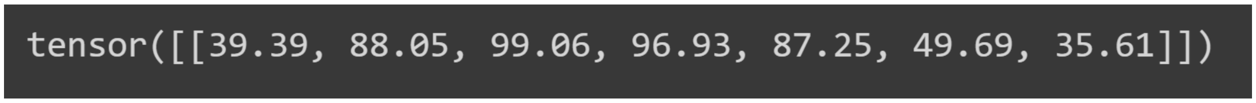
```
MyTensor4 = 100 * torch.rand(1, 7)
torch.set_printoptions(profile="full")
MyTensor4
```

Gives:



```
tensor([[39.3905, 88.0472, 99.0607, 96.9299, 87.2491, 49.6853, 35.6061]])
```

Note: I don't see much difference in writing style but in profile="short", it prints only 2 decimal points of the numbers in a tensor. I don't know why but I don't want to spend to much time n this so I am just leaving it as it is.



```
tensor([[39.39, 88.05, 99.06, 96.93, 87.25, 49.69, 35.61]])
```

sci_mode: Enable (sci_mode=True) or disable (sci_mode=False) scientific notation.

```
torch.set_printoptions(sci_mode=True)
```

```
MyTensor4
```
Which uses the MyTensor 4 from above Gives:

```
tensor([[3.9391e+01, 8.8047e+01, 9.9061e+01, 9.6930e+01, 8.7249e+01,
4.9685e+01, 3.5606e+01]])
```

Note: This is the command to get the default print options. You can use it if you want to set things to default:

```
torch.set_printoptions(precision=4, threshold=1000, edgeitems=3, linewidth=80, profile="default", sci_mode=False)
```

Torch.min()

Gives the minimum valued element in the given tensor.

Note: Torch.min(MyTensor) gives the exact same result as MyTensor.min(). They are identical.

```
MyTensor = 100 * torch.rand(3, 2)
MyTensor
```
Gives:

```
tensor([[82.8892, 85.6225],
        [59.2919, 44.9064],
        [79.0923, 25.2931]])
```

```
torch.min(MyTensor)
```
Gives:

```
tensor(25.2931)
```

Torch.max()

Gives the maximum valued element in the given tensor.

Note: Torch.max(MyTensor) gives the exact same result as MyTensor.max(). They are identical.

```
MyTensor = 100 * torch.rand(3, 2)
MyTensor
```
Gives:

```
tensor([[82.8892, 85.6225],
        [59.2919, 44.9064],
        [79.0923, 25.2931]])
```

```
torch.max(MyTensor)
```

Gives:

```
tensor(85.6225)
```

Torch.sum()

Gives the sum of all the elements in the given input tensor.

Note: torch.sum(MyTensor) gives the exact same result as MyTensor.sum(). They are the same things just with different syntax.

```
MyTensor = 100 * torch.rand(3, 2)
MyTensor
```

Gives:

```
tensor([[82.8892, 85.6225],
        [59.2919, 44.9064],
        [79.0923, 25.2931]])
```

```
MyTensor.sum()
```

Gives:

```
tensor(377.0953)
```

Torch.argmin()

Gives the position of the minimum valued element in the given input tensor.

Note: This position is the position when all the elements of the tensor are lined up in a single line, i.e. the position continues when we go from tensor to tensor in a multi-dimensional tensor.

Note: torch. argmin(MyTensor) gives the exact same result as MyTensor. argmin(). They are the same things just with different syntax.

```
MyTensor = 100 * torch.rand(3, 2)
MyTensor
```

Gives:

```
tensor([[82.8892, 85.6225],
        [59.2919, 44.9064],
        [79.0923, 25.2931]])
```

```
torch.argmin(MyTensor)
```

Gives:

```
tensor(5)
```

Torch.argmax()

Gives the position of the maximum valued element in the given input tensor.

Note: This position is the position when all the elements of the tensor are lined up in a single line, i.e. the position continues when we go from tensor to tensor in a multi-dimensional tensor.

Note: torch. argmax(MyTensor) gives the exact same result as MyTensor. argmax(). They are the same things just with different syntax.

```
MyTensor = 100 * torch.rand(3, 2)
MyTensor
```

Gives:

```
tensor([[82.8892, 85.6225],
        [59.2919, 44.9064],
        [79.0923, 25.2931]])
```

```
torch.argmax(MyTensor)
```

Gives:

```
tensor(1)
```

Deep Copy and Shallow Copy

A deep copy means that the copied object and the original object are 2 independent objects stored in different memory locations. If you change one of them, the other won't be effected whereas in the shallow copy, the 2 objects(original and the copy) are only 2 object referencing to the same memory location so if you change the data on of the objects the other will change as well. Or there is actually no other object since there is only 1 object with 2 references and if we change the object, since both references point to the same object, you get the same changed object when you reference the object with either of the referances.

Torch.reshape()

Returns a tensor with the <u>same data and same number of elements as input</u>, but with the specified shape.

Note: Torch.reshape() creates a deep copy instead of a shallow copy usually but in some circumstances, Torch.reshape() can give you a shallow copy as well so don't trust this to give you a deep copy.

Note: torch.reshape(MyTensor, (a, b)) is exactly the same thing as MyTensor.reshape(a, b). They are just different syntaxes that does the exact same thing.

```
MyTensor = 100 * torch.rand(3, 2)
MyTensor
```

Gives:

```
tensor([[13.5187, 16.3116],
        [42.8206, 32.0972],
        [14.2768, 58.4716]])
```

```
MyTensor2 = MyTensor.reshape(6, 1)
MyTensor2
```

Gives:

```
tensor([[13.5187],
        [16.3116],
        [42.8206],
        [32.0972],
        [14.2768],
        [58.4716]])
```

Torch.view()

Returns a tensor with the same data and same number of elements as input, but with the specified shape.

Note: Torch.view() creates a shallow copy instead of a deep copy. Torch.view() does not change tensor layout in memory. This means that let's say we have:

MyTensor2 = torch.view(MyTensor, (6, 1))

Now if we change an element in MyTensor2, MyTensor will also change.

Note: torch.view(MyTensor, (a, b)) is exactly the same thing as MyTensor.view(a, b). They are just different syntaxes that does the exact same thing.

```
MyTensor = 100 * torch.rand(3, 2)
MyTensor
```

Gives:

```
tensor([[42.3455, 89.6493],
        [45.3388, 97.7796],
        [27.8926, 32.4345]])
```

```
MyTensor2 = MyTensor.view(6, 1)
```

```
MyTensor2
```

Gives:

```
tensor([[42.3455],
        [89.6493],
        [45.3388],
        [97.7796],
        [27.8926],
        [32.4345]])
```

Note: Torch.view() can also be used to get the tensor with the same data but in another data type but it is a little bit complicated. For info about it: https://pytorch.org/docs/stable/generated/torch.Tensor.view.html

Torch.mean()

Gives the mean of all the elements in the given input tensor.

Note: torch.mean(MyTensor) gives the exact same result as MyTensor.mean(). They are the same things just with different syntax.

```
MyTensor = 100 * torch.rand(3, 2)
MyTensor
```

Gives:

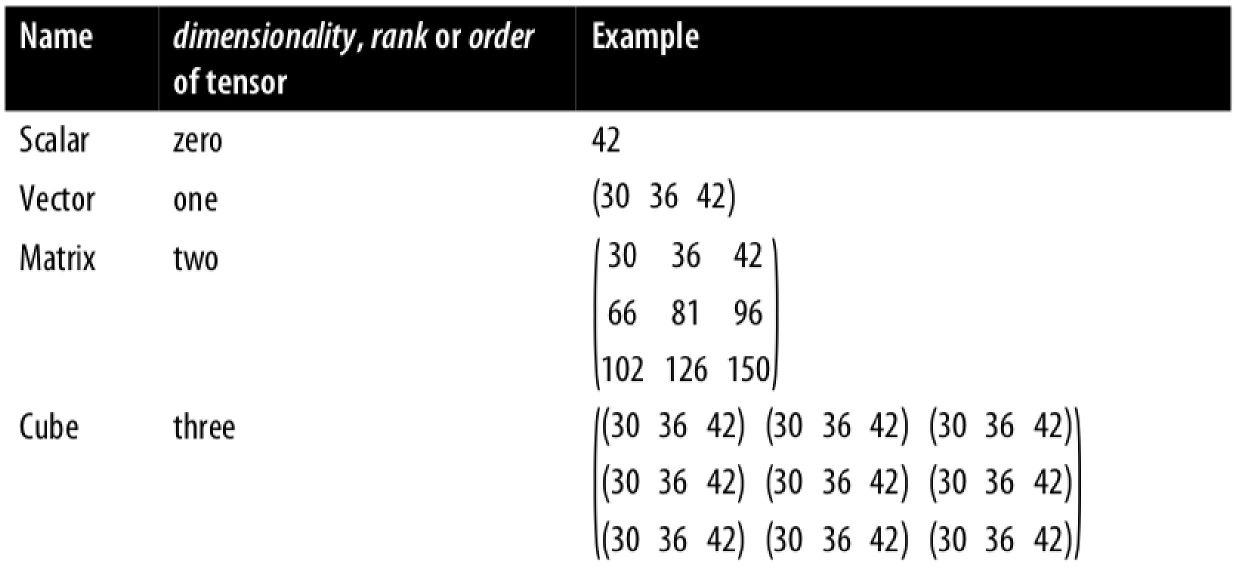
```
tensor([[82.8892, 85.6225],
        [59.2919, 44.9064],
        [79.0923, 25.2931]])
```

```
MyTensor.mean()
```

Gives:

```
tensor(62.8492)
```

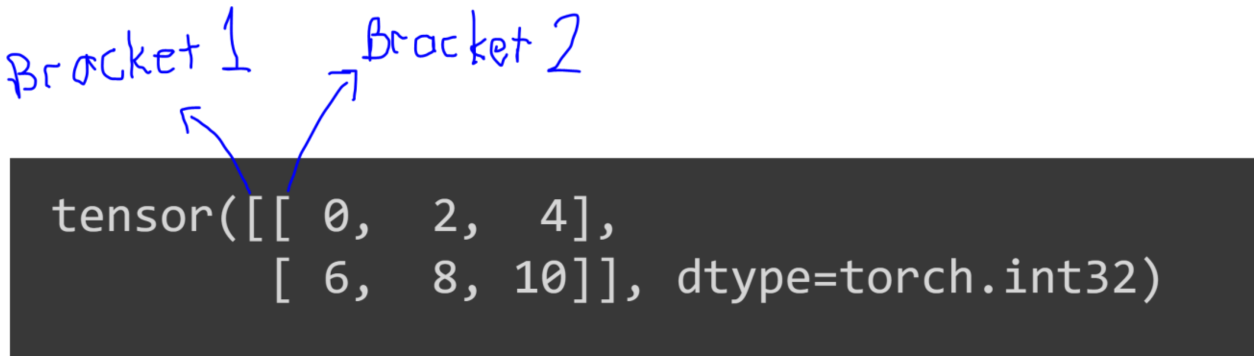
Information, Insight About Dimensions of a Tensor:



Name	*dimensionality, rank* or *order* of tensor	Example
Scalar	zero	42
Vector	one	$\begin{pmatrix} 30 & 36 & 42 \end{pmatrix}$
Matrix	two	$\begin{pmatrix} 30 & 36 & 42 \\ 66 & 81 & 96 \\ 102 & 126 & 150 \end{pmatrix}$
Cube	three	$\begin{pmatrix} (30\ 36\ 42) & (30\ 36\ 42) & (30\ 36\ 42) \\ (30\ 36\ 42) & (30\ 36\ 42) & (30\ 36\ 42) \\ (30\ 36\ 42) & (30\ 36\ 42) & (30\ 36\ 42) \end{pmatrix}$

Source: https://medium.com/codait/confused-about-tensors-dimensions-ranks-orders-matrices-and-vectors-e0e2ea480fcf

As I understand, this is the dimensions of a tensor. A vector of dimensionality 1 is a tensor with 1 bracket. The elements inside bracket 1 is the elements of dimension 1. Likewise, in a Matrix of 2 dimensions, the elements (element can be a scalar or a tensor) inside the first bracket is the elements of dimension 1. And the elements inside bracket 2 is the elements of dimension 2.



```
tensor([[ 0,  2,  4],
        [ 6,  8, 10]], dtype=torch.int32)
```

In this example, the elements of dimension 1 is the elements inside the bracket 1 which are:

[0, 2, 4] and [6, 8, 10]

The elements of dimension 2 are:

0, 2, 4, 6, 8, and 10

The elements of dimension 0 is the whole tensor which is only 1 element as:

[[0, 2, 4], [6, 8, 10]]

Torch.stack((T1, T2), dim=a)

Stacks, put together the input tensors T1 and T2 along the dimension a. The default dimension if you don't define is dim=0.

It puts the first element of T1 on dim=a, then puts the first element in T2 along dim=a. This becomes the first element of the stacked result tensor. Then, it does the same thing for the second elements of T1 and T2 along dim=a and goes like that through every element of T1 and T2 along dim=a.

To give an example, we use the tensors T1 and T2 defined as:

```
T1 = ( torch.arange(start=0, end=12, step=2) ).type(torch.int32)
T1 = T1.reshape(2, 3)
T2 = ( torch.arange(start=1, end=12, step=2) ).type(torch.int32)
T2 = T2.reshape(2, 3)
T1, T2
```

Which gives:

```
(tensor([[ 0,  2,  4],
         [ 6,  8, 10]], dtype=torch.int32),
 tensor([[ 1,  3,  5],
         [ 7,  9, 11]], dtype=torch.int32))
```

```
MyStackedTensor = torch.stack((T1, T2), dim=0)
MyStackedTensor
```

Gives:

```
tensor([[[ 0, 2, 4],
         [ 6, 8, 10]],
        [[ 1, 3, 5],
         [ 7, 9, 11]]], dtype=torch.int32)
```

```
MyStackedTensor = torch.stack((T1, T2), dim=1)
MyStackedTensor
```

Gives:

```
tensor([[[ 0, 2, 4],
         [ 1, 3, 5]],
        [[ 6, 8, 10],
         [ 7, 9, 11]]], dtype=torch.int32)
```

```
MyStackedTensor = torch.stack((T1, T2), dim=2)
MyStackedTensor
```

Gives:

```
tensor([[[ 0, 1],
         [ 2, 3],
         [ 4, 5]],
        [[ 6, 7],
         [ 8, 9],
         [10, 11]]], dtype=torch.int32)
```

Or equivalently:

```
tensor([[[ 0, 1], [ 2, 3], [ 4, 5]],
        [[ 6, 7], [ 8, 9], [10, 11]]], dtype=torch.int32)
```

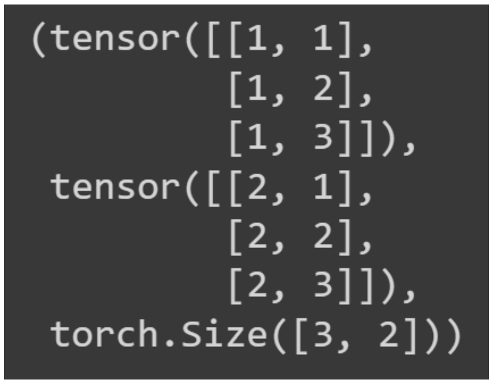
Note: Notice that MyStackedTensor is always 3 dimensional but the shape of the tensor changes depending on the dimension you add the tensors on.

Torch.vstack((T1, T2))

It concatenates T1 and T2 along the first dimension. It doesn't go through the first elements of T1 and T2 on dim=1, then second elements of T1 and T2 and so on. Instead, it puts every element of T1, then it puts (concatenates) every element of T2 and connect the elements of T1 and T2 in one element (one tensor) of the resulting tensor. The concatenation of the elements of T1 and T2 are along dim=1. It is similar to torch.stack at dim=0. But in torch.vstack, it concerts mxn tensors which is the generated by stacking the kth elements of T1 and T2 in torch.stack into one tensor of size mn.

```
T1 = torch.tensor([[1, 1], [1, 2], [1, 3]])
T2 = torch.tensor([[2, 1], [2, 2], [2, 3]])
T1, T2, T1.size()
```

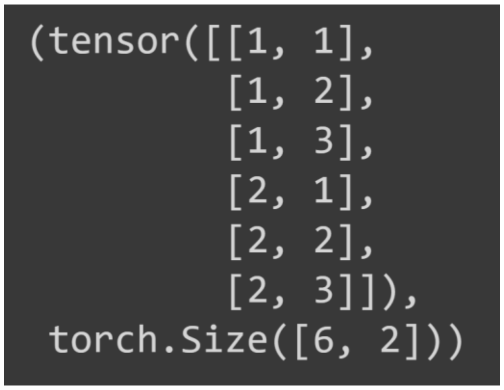
Gives:



```
(tensor([[1, 1],
         [1, 2],
         [1, 3]]),
 tensor([[2, 1],
         [2, 2],
         [2, 3]]),
 torch.Size([3, 2]))
```

```
MyStackedTensor = torch.vstack((T1, T2))
MyStackedTensor, MyStackedTensor.size()
```

Gives:

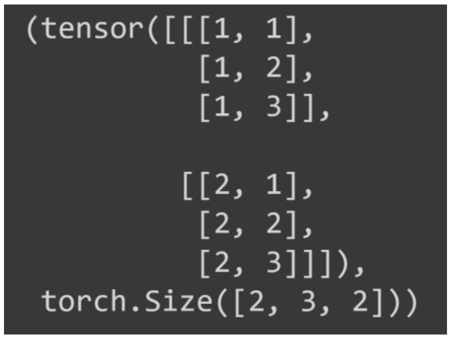


```
(tensor([[1, 1],
         [1, 2],
         [1, 3],
         [2, 1],
         [2, 2],
         [2, 3]]),
 torch.Size([6, 2]))
```

And if we use torch.stack at dim=0 as:

```
MyStackedTensor = torch.stack((T1, T2), dim=0)
MyStackedTensor, MyStackedTensor.size()
```

We get:



```
(tensor([[[1, 1],
          [1, 2],
          [1, 3]],

         [[2, 1],
          [2, 2],
          [2, 3]]]),
 torch.Size([2, 3, 2]))
```

As we can see from the above example, in the result of torch.vstack function, 3x2 tensors resulting from concatenating kth elements of T1 and T2 are represented as tensors of size 6 in the result of torch.vstack function.

Torch.hstack((T1, T2))

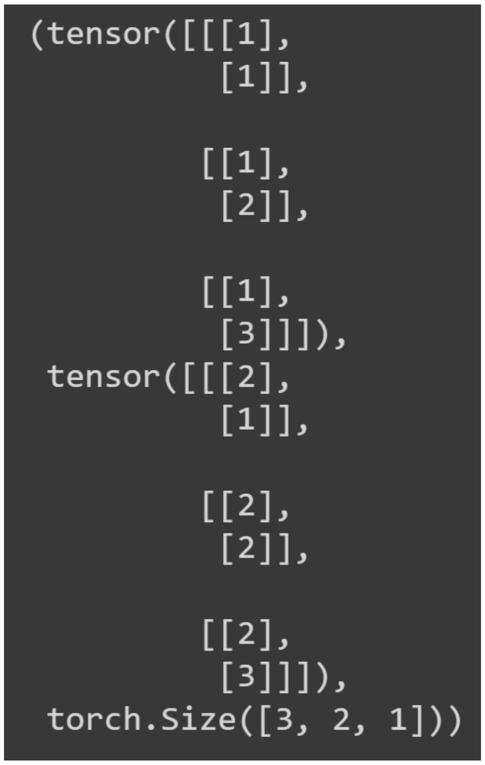
It concatenates T1 and T2 along the second dimension. It doesn’t go through the first elements of T1 and T2 then second elements of T1 and T2 and so on. Instead, it puts every element of T1 on dim=2, then it puts (concatenates) every element of T2 on dim=2 and connect the elements of T1 and T2 in one element (one tensor) of the resulting tensor. The concatenation of the elements of T1 and T2 are along dim=2.

It is similar to torch.stack at dim=1. But in torch.hstack, it concerts mxn tensors which is the generated by stacking the kth elements of T1 and T2 in torch.stack into one tensor of size mn.

```
T1 = torch.tensor([[[1], [1]], [[1], [2]], [[1], [3]]])
T2 = torch.tensor([[[2], [1]], [[2], [2]], [[2], [3]]])
T1, T2, T1.size()
```

Note: This is not the same T1 and T2 used in the torch.vstack explanation above.

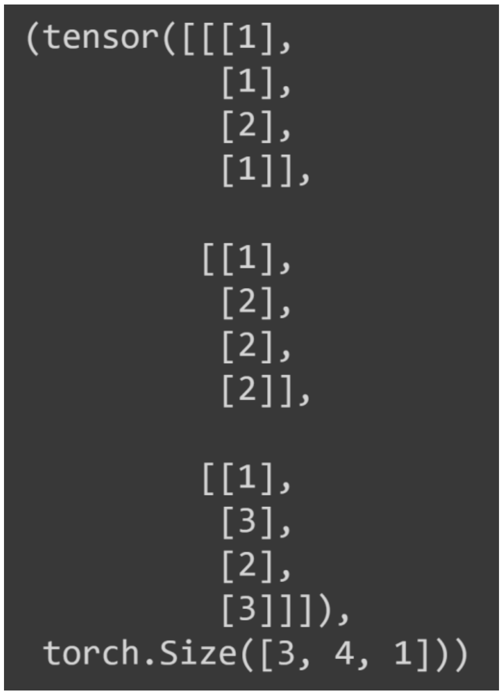
Gives:



```
(tensor([[[1],
          [1]],

         [[1],
          [2]],

         [[1],
          [3]]]),
 tensor([[[2],
          [1]],

         [[2],
          [2]],

         [[2],
          [3]]]),
 torch.Size([3, 2, 1]))
```

```
MyStackedTensor = torch.hstack((T1, T2))
MyStackedTensor, MyStackedTensor.size()
```

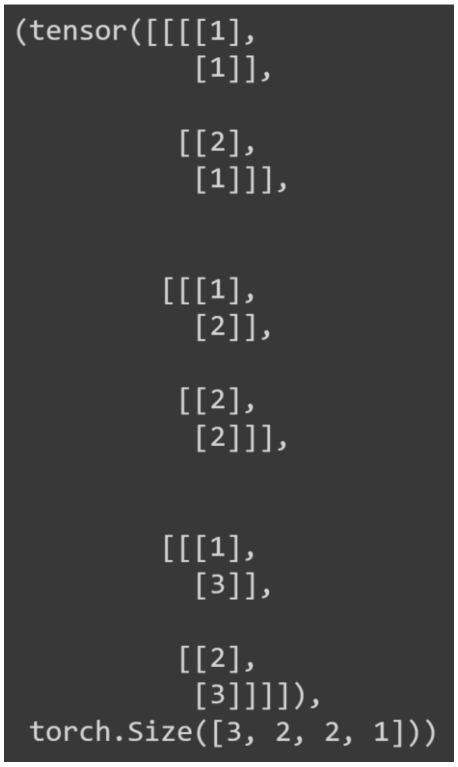
Gives:



```
(tensor([[[1],
          [1],
          [2],
          [1]],

         [[1],
          [2],
          [2],
          [2]],

         [[1],
          [3],
          [2],
          [3]]]),
 torch.Size([3, 4, 1]))
```

```
MyStackedTensor = torch.stack((T1, T2), dim=1)
MyStackedTensor, MyStackedTensor.size()
```

Gives:



```
(tensor([[[[1],
           [1]],

          [[2],
           [1]]],


         [[[1],
           [2]],

          [[2],
           [2]]],


         [[[1],
           [3]],

          [[2],
           [3]]]]),
 torch.Size([3, 2, 2, 1]))
```

As we can see from the above example, in the result of torch.hstack function, 2x2x1 tensors resulting from concatenating kth elements of T1 and T2 are represented as tensors of size 4x1 in the result of torch.hstack function.

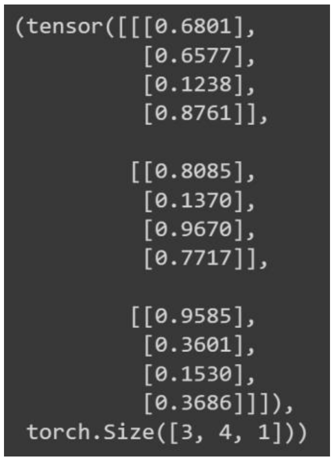
Torch.squeeze(myTensor, dim=a)

If the dimension(dim=) argument is not given, it reshapes the tensor to remove all the dimensions of size=1. For example, if myTensor is of size 3x1x5x8, the resulting tensor of torch.squeeze will be 3x5x8. Basically, it takes the dimensions that has only 1 element and stacks them together.

If the dimension argument is given as dim=a, it only looks at dimension a and if that dimension has size=1, it removes that dimension. If the size of dimension a is not 1, it doesn't remove the dimension and gives the same tensor without doing anything.

```
myTensor = torch.rand(3, 4, 1)
myTensor, myTensor.shape
```

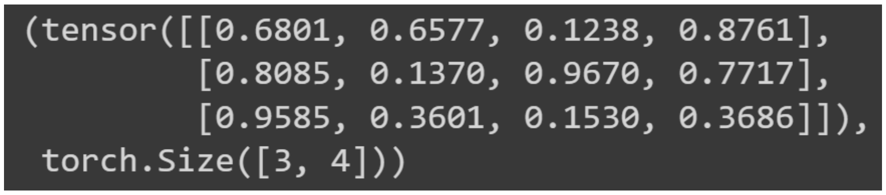
Gives:



```
(tensor([[[0.6801],
         [0.6577],
         [0.1238],
         [0.8761]],

        [[0.8085],
         [0.1370],
         [0.9670],
         [0.7717]],

        [[0.9585],
         [0.3601],
         [0.1530],
         [0.3686]]]),
 torch.Size([3, 4, 1]))
```

```
mySqueezedTensor = torch.squeeze(myTensor, dim=2)
mySqueezedTensor, mySqueezedTensor.shape
```

And this gives:

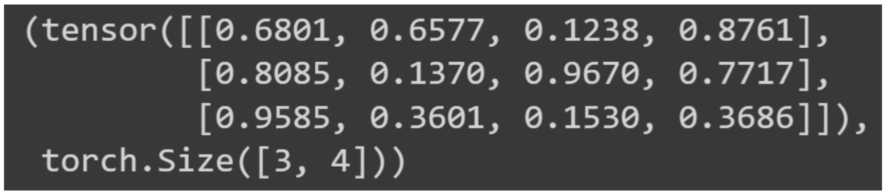


```
(tensor([[0.6801, 0.6577, 0.1238, 0.8761],
         [0.8085, 0.1370, 0.9670, 0.7717],
         [0.9585, 0.3601, 0.1530, 0.3686]]),
 torch.Size([3, 4]))
```

So it removes the 2nd dimension since there is only one element inside every element of the 2nd dimension. We can also say that the size of dim=2 is 1.

```
mySqueezedTensor = torch.squeeze(myTensor)
mySqueezedTensor, mySqueezedTensor.shape
```

Also gives the exact same output:



```
(tensor([[0.6801, 0.6577, 0.1238, 0.8761],
         [0.8085, 0.1370, 0.9670, 0.7717],
         [0.9585, 0.3601, 0.1530, 0.3686]]),
 torch.Size([3, 4]))
```

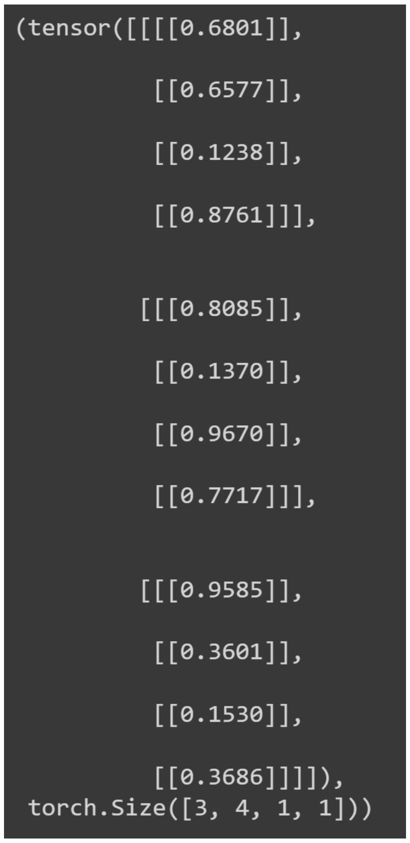
Torch.unsqueeze(myTensor, dim=a)

It returns a tensor with a dimension inserted at dimension a. In other words, it takes every element in dimension a and put each one of them in its own dimension (in its own bracket). So if myTensor is of size 3x4x1, torch.unsqueeze(myTensor, dim=1) returns a 3x1x4x1 matrix so it adds a dimension of size 1 outside dimension 1.

We use the myTensor from the torch.squeeze part above. myTensor is of size: torch.Size([3, 4, 1])

```
myUnsqueezedTensor = torch.unsqueeze(myTensor, 2)
myUnsqueezedTensor, myUnsqueezedTensor.shape
```

Gives:



```
(tensor([[[[0.6801]],

          [[0.6577]],

          [[0.1238]],

          [[0.8761]]],


         [[[0.8085]],

          [[0.1370]],

          [[0.9670]],

          [[0.7717]]],


         [[[0.9585]],

          [[0.3601]],

          [[0.1530]],

          [[0.3686]]]]),
 torch.Size([3, 4, 1, 1]))
```

Torch.permute(myTensor, (a, b, c))

It returns a matrix which has the same data as myTensor but its dimensions are switched. It is similar to torch.reshape() but not the same, they give different results.

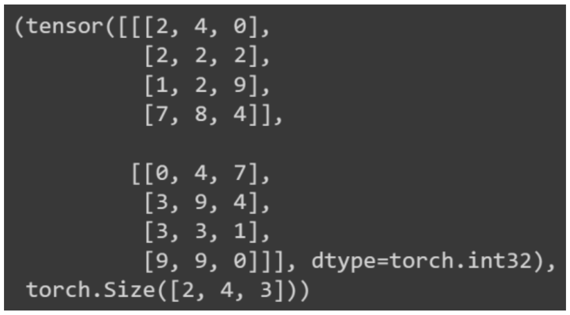
The tuple (a, b, c) represents the order of dimensions that you want to switch. a is the first dimension, b is second and c is the third in the resulting array. So if myTensor is a 2x3x4 tensor, Torch.permute(myTensor, (0, 1, 2)) returns a tensor of shape 3x4x2.

Keep in mind that this function gives a shallow copy of myTensor so if you change an element in myTensor, the same element in resulting matrix of Torch.permute(myTensor, (a, b, c)) also changes.

Couldn't really understand the algorithm behind it but when I think of how can I reshape, rearrange the elements to fit the specified dimensionality, I can guess the algorithm correctly.

```
myTensor = ( torch.rand(2, 4, 3)*10 ).type(torch.int32)
myTensor, myTensor.shape
```

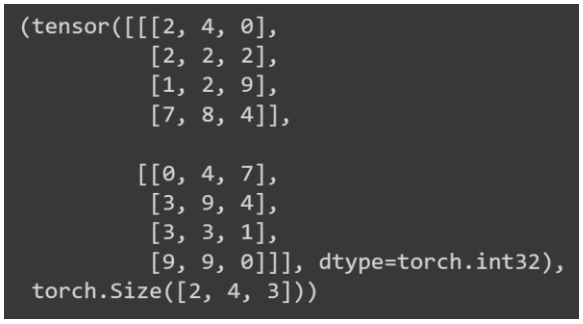
Gives:



```
(tensor([[[2, 4, 0],
          [2, 2, 2],
          [1, 2, 9],
          [7, 8, 4]],

         [[0, 4, 7],
          [3, 9, 4],
          [3, 3, 1],
          [9, 9, 0]]], dtype=torch.int32),
 torch.Size([2, 4, 3]))
```

```
myPermutedTensor = torch.permute(myTensor, (0, 1, 2))
myPermutedTensor,  myPermutedTensor.shape
```

Gives:



```
(tensor([[[2, 4, 0],
          [2, 2, 2],
          [1, 2, 9],
          [7, 8, 4]],

         [[0, 4, 7],
          [3, 9, 4],
          [3, 3, 1],
          [9, 9, 0]]], dtype=torch.int32),
 torch.Size([2, 4, 3]))
```

Torch. from_numpy(myNumpyArray)

Returns a numpy array with the same data. Basically a conversion from numpy array data type to torch.tensor data type.

This function generates a deep copy instead of a shallow copy so after you used this function, the torch.tensor output of this function is not effected by the changes of the numpy array that is used in the function.

Information might be lost since the default data type of torch.tensor is float32 and default datatype of numpy array is float64.

Let us use the following example to demonstrate this function:

```
myTensor = torch.arange(1.0, 8.0)
myTensor, myTensor.dtype
```

Gives:

```
(tensor([1., 2., 3., 4., 5., 6., 7.]), torch.float32)
```

And

```
myNumpyArray = np.arange(1, 8)
myNumpyArray, myNumpyArray.dtype
```

Gives:

```
(array([1, 2, 3, 4, 5, 6, 7]), dtype('int64'))
```

```
T1 = torch.from_numpy(myNumpyArray)
myNumpyArray = myNumpyArray + 4.4
T1, T1.dtype
```

This gives:

```
(tensor([1, 2, 3, 4, 5, 6, 7]), torch.int64)
```

And

```
myNumpyArray, myNumpyArray.dtype
```

Gives:

```
(array([ 5.4, 6.4, 7.4, 8.4, 9.4, 10.4, 11.4]), dtype('float64'))
```

Tensor.numpy(myTensor)

Returns a torch.tensor with the same data. Basically a conversion from torch.tensor data type to numpy array data type.

This function generates a deep copy instead of a shallow copy so after you used this function, the numpy array output of this function is not effected by the changes of the torch.tensor that is used in the function.

No information is lost since the default data type of numpy arrays are float64 and default datatype of torch.tensor is float32.

Let us use the following example to demonstrate this function:

```
A1 = myTensor.numpy()
myTensor = myTensor + 4.4
A1
```

Gives:

```
array([1., 2., 3., 4., 5., 6., 7.], dtype=float32)
```

Where;

```
myTensor, myTensor.dtype
```

Gives:

```
(tensor([ 5.4000, 6.4000, 7.4000, 8.4000, 9.4000, 10.4000, 11.4000]),
torch.float32)
```

RANDOM SEEDS:

Random seed is basically a key to a sequence of random numbers. For some key K, there is a sequence of numbers which are not random. We say that there are random but when you know the key, the numbers in the sequence can be calculated. But the numbers are generated in a random kind of manner in a sense that the numbers are statistically unlikely to repeat, they are statistically distributed uniformly etc. for any key/random seed/seed, you take it as it is a random number but remember that when someone knows the seed, s/he can produce the same numbers as you.

This can actually be useful in coding because it enables us to generate random numbers repeatably. We can use a key K to generate random numbers and if someone uses the same key as us, s/he can obtain the same results as us although the numbers are assumed to be random. This enables our code to be consistently repeatable and reproducible.

REPRODUCIBILITY:

Reproducibility is the measure of if a system, i.e. a code, can be implemented at different times, devices or environments and give the same results.

For example if you use myTensor = torch.rand(3, 3) in your code, your code will give different results each time you run it. We want the code to give the same results even if we run on another device 20 years later! There are some ways to do this or rather get good at this because this is not a thing you can learn in a day but it is something you get good at through experience. Remember, a good code is a code that is reproducible regardless of the time and environment/device it is run on.

Here is some documentation that might be helpful in making your pytorch code reproducible: https://pytorch.org/docs/stable/notes/randomness.html

Also check out the function torch.use_deterministic_algorithms() method which seem very helpful in making your code reproducible.

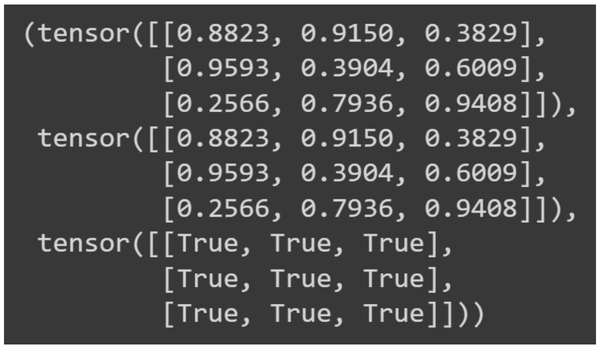
Note: We can say that determinism means reproducibility in most cases in programming

torch. manual_seed(mySeed)

It is used to set a random seed used for generating random numbers. Using this function, you can generate the same numbers repeatedly in different times and places. See RANDOM SEEDS section above for more info. Here is some example of the use of the function torch. manual_seed()

```
RANDOM_SEED = 42
torch.manual_seed(RANDOM_SEED)
T1 = torch.rand(3, 3)


torch.manual_seed(RANDOM_SEED)
T2 = torch.rand(3, 3)
T1, T2, T1==T2
```

Gives:

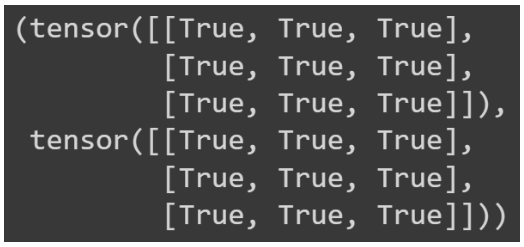


```
(tensor([[0.8823, 0.9150, 0.3829],
         [0.9593, 0.3904, 0.6009],
         [0.2566, 0.7936, 0.9408]]),
 tensor([[0.8823, 0.9150, 0.3829],
         [0.9593, 0.3904, 0.6009],
         [0.2566, 0.7936, 0.9408]]),
 tensor([[True, True, True],
         [True, True, True],
         [True, True, True]]))
```

Also,

```
RANDOM_SEED = 42
torch.manual_seed(RANDOM_SEED)
T1 = torch.rand(3, 3)
T2 = torch.rand(3, 3)
```

```
torch.manual_seed(RANDOM_SEED)
T3 = torch.rand(3, 3)
T4 = torch.rand(3, 3)

T1 == T3, T2==T4
```
Gives:



```
(tensor([[True, True, True],
         [True, True, True],
         [True, True, True]]),
 tensor([[True, True, True],
         [True, True, True],
         [True, True, True]]))
```

Note: Note that for one random seed, there isn’t just one random number but there is a sequence of numbers to be generated so if you want to generate the same number with torch.rand() back to back, you should re initialize the number sequence with that key again by setting the random seed again to the same value by:

```
torch.manual_seed(RANDOM_SEED)
```

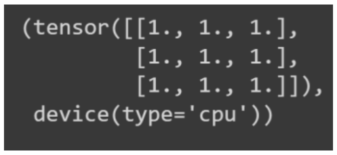
Writing Device Agnostic Code

If you define your device as:

```
myDevice = "cuda" if torch.cuda.is_available() else "cpu"
```
then you can use the myDevice as your device and you will use cuda if it is available and cpu if gpu is not available at the time you run the program.

For example:

```
myTensor = torch.rand(3, 2, device=myDevice)
myTensor.device
```
gives:



```
(tensor([[1., 1., 1.],
        [1., 1., 1.],
        [1., 1., 1.]]),
 device(type='cpu'))
```

Moving a Tensor from the Device to Device

Tensor.cpu()

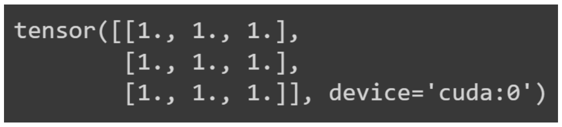
It is used to carry a tensor from gpu to cpu. Tensor.cpu() generates a deep copy of the tensor located in the cpu and returns that tensor

This can be used to use numpy functions on tensors on gpu by carrying them to cpu then use numpy.

Note: numpy only works with cpu including tensor.numpy() which converts a tensor with torch.tensor datatype to numpy datatype

```
myGpuTensor = torch.ones(3, 3, device="cuda")
myGpuTensor
```

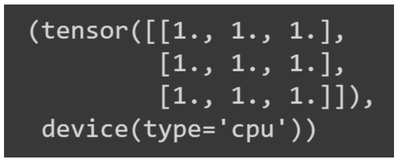
Gives



```
tensor([[1., 1., 1.],
        [1., 1., 1.],
        [1., 1., 1.]], device='cuda:0')
```

```
myTensor = myGpuTensor.cpu()
myTensor, myTensor.device
```

Gives:

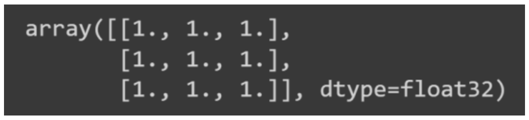


```
(tensor([[1., 1., 1.],
         [1., 1., 1.],
         [1., 1., 1.]]),
 device(type='cpu'))
```

And using tensor.cpu() we can use numpy functions as:

```
myNumpyTensor = myGpuTensor.cpu().numpy()
myNumpyTensor
```

gives:

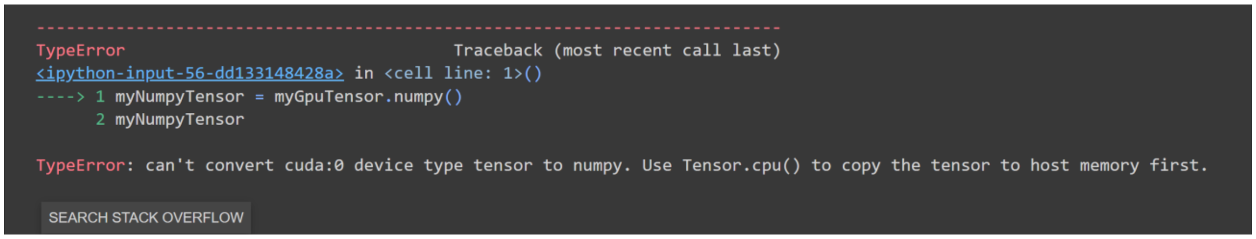


```
array([[1., 1., 1.],
       [1., 1., 1.],
       [1., 1., 1.]], dtype=float32)
```

Where if we didn't carry myGpuTensor from GPU to CPU using tensor.cpu(), we would get an error as:

```
myNumpyTensor = myGpuTensor.numpy()
myNumpyTensor
```

Gives:



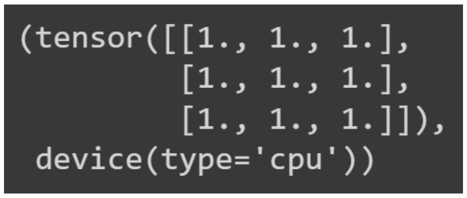
```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-56-dd133148428a> in <cell line: 1>()
----> 1 myNumpyTensor = myGpuTensor.numpy()
      2 myNumpyTensor

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

Note:

Instead of using

```
myTensor = myGpuTensor.cpu()
myTensor, myTensor.device
```

Which gives:

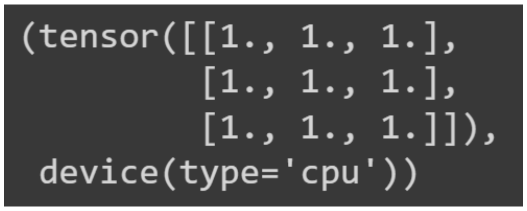


```
(tensor([[1., 1., 1.],
         [1., 1., 1.],
         [1., 1., 1.]]),
 device(type='cpu'))
```

We could use:

```
myTensor = myGpuTensor.to("cpu")
myTensor, myTensor.device
```

which gives the exact same result as:



```
(tensor([[1., 1., 1.],
         [1., 1., 1.],
         [1., 1., 1.]]),
 device(type='cpu'))
```

Tensor.to(device=myDevice, dtype=myDType)

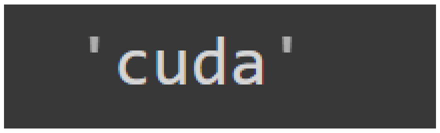
It is used to carry a tensor from one device to another. It returns a deep copy of the tensor but in the given device=myDevice.

It can also be used for changing a tensor’s datatype to a specified datatype.

We defined myDevice as:

```
myDevice = "cuda" if torch.cuda.is_available() else "cpu"
myDevice
```

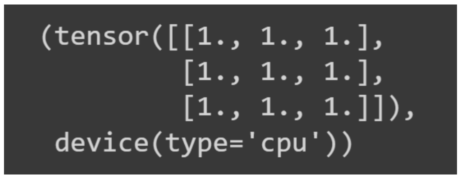
which gives



```
'cuda'
```

```
myCpuTensor, myCpuTensor.device
```

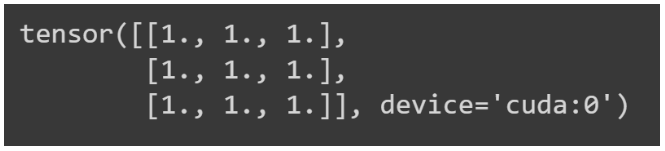
Gives:



```
(tensor([[1., 1., 1.],
         [1., 1., 1.],
         [1., 1., 1.]]),
 device(type='cpu'))
```

```
myNewtensor = myCpuTensor.to(myDevice)
myNewtensor
```

Gives:

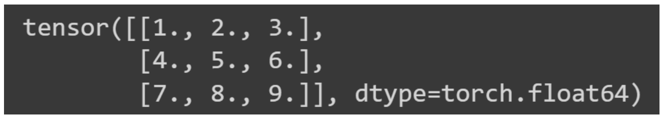


```
tensor([[1., 1., 1.],
        [1., 1., 1.],
        [1., 1., 1.]], device='cuda:0')
```

We use tensor.to() for changing a tensor's datatype to a specified datatype as:

```
myTensor = torch.tensor([[1, 2, 3], [4, 5, 6], [7, 8, 9]], dtype=torch.float64)

myTensor
```

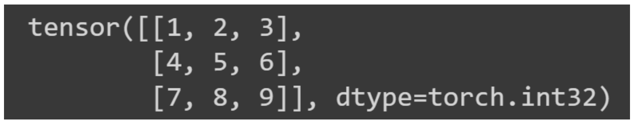
Gives:



```
tensor([[1., 2., 3.],
        [4., 5., 6.],
        [7., 8., 9.]], dtype=torch.float64)
```

```
myNewTensor = myTensor.to(dtype=torch.int32)
myNewTensor
```

Gives:



```
tensor([[1, 2, 3],
        [4, 5, 6],
        [7, 8, 9]], dtype=torch.int32)
```

Tensor.item()

Gives the number in a tensor when the tensor has only 1 element. It gives the element outside of the tensor no matter how many dimensions there are as long as there is only one element in the tensor

```
myTensor = torch.tensor([[[[34.58]]]])
myTensor.item()
```

Gives:

```
34.58000183105469
```

Which is the number we put inside the tensor. python is not accurate after some digits. If you cannot tolerate this, use a datatype that has more precision such as torch.float64:

```
myTensor = torch.tensor([[[[34.58]]]], dtype=torch.float64)
myTensor.item()
```

Gives:

```
(34.58, float)
```

Torch.clone(myTensor)

This function returns a deep copy of the tensor that is given as an argument (myTensor in our case) with the same data and datatype.

```
myTensor = torch.tensor([[1, 2, 3], [4, 5, 6], [7, 8, 9]], dtype=torch.float64)

myTensor
```

Gives:

```
tensor([[1., 2., 3.],
        [4., 5., 6.],
        [7., 8., 9.]], dtype=torch.float64)
```

```
myCopiedTensor = torch.clone(myTensor)
myTensor
```

Gives:

```
tensor([[1., 2., 3.],
        [4., 5., 6.],
        [7., 8., 9.]], dtype=torch.float64)
```