```
## Pytorch fundamentals
```

Why use Pytorch?
PyTorch helps take care of many things such as GPU acceleration (making your code run faster) behind the scenes.

```
print("I am excited to learn pytorch")
```

```
I am excited to learn pytorch
```

```
import torch
torch.__version__
```

Tensors are the fundamental building block of machine learning.

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import torch
torch.__version__
```

```
!nvidia-smi
```

```
Sun Feb 23 09:09:25 2025
+-----------------------------------------------------------------------------------------+
| NVIDIA-SMI 550.54.15              Driver Version: 550.54.15      CUDA Version: 12.4     |
|-----------------------------------------+------------------------+----------------------+
| 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   57C    P8             11W /   70W |       0MiB /  15360MiB |      0%      Default |
|                                         |                        |                  N/A |
+-----------------------------------------+------------------------+----------------------+

+-----------------------------------------------------------------------------------------+
| Processes:                                                                              |
|  GPU   GI   CI        PID   Type   Process name                              GPU Memory |
|        ID   ID                                                               Usage      |
|=========================================================================================|
|  No running processes found                                                             |
+-----------------------------------------------------------------------------------------+
```

```
## Introduction to tensors

## Creating tensors
```

```
# sclar
scalar=torch.tensor(7)
scalar
```

```
tensor(7)
```

```
scalar.ndim
```

```
0
```

```
# get the python number within a tensor(only works with one-element tensor)
scalar.item()
```

```
7
```

```
# Vector: a vector is a single dimension tensor but can contain many numbers
vector=torch.tensor([2,3,4,4])
vector
```

```
tensor([2, 3, 4, 4])
```

```
vector.ndim
```

```
1
```

```
vector.shape   # shape of the vector
```

```
torch.Size([4])
```

```
#matrix
MATRIX=torch.tensor([[2,3,4,5],
                    [3,5,6,7]])
MATRIX
```

```
tensor([[2, 3, 4, 5],
        [3, 5, 6, 7]])
```

```
MATRIX.ndim
```

```
2
```

```
MATRIX.shape
```

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

```
MATRIX[0]
```

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

```
MATRIX[1]
```

```
tensor([3, 5, 6, 7])
```

```
MATRIX[1][1]
```

```
tensor(5)
```

```
# tensor
TENSOR=torch.tensor([[[1,2,3],[5,6,7]],
                     [[3,4,5],[6,7,8]]])
TENSOR
```

```
tensor([[[1, 2, 3],
         [5, 6, 7]],

        [[3, 4, 5],
         [6, 7, 8]]])
```

```
TENSOR.shape
```

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

```
TENSOR.ndim
```

```
3
```

```
TENSOR[0]
```

```
tensor([[1, 2, 3],
        [5, 6, 7]])
```

```
TENSOR[1]
```

```
tensor([[3, 4, 5],
        [6, 7, 8]])
```

```
TENSOR[0][1]
```

```
tensor([5, 6, 7])
```

Random tensors

why random tensors?

Random tensors are important because the way many neural networks learn is that they start with tensors full of random numbers and then adjust those random numbers to better represent the data.

- Start with random numbers -> look at data -> update random numbers -> look at data -> update random numbers...
- As a data scientist, we can define how the machine learning model starts (initialization), looks at data (representation) and updates (optimization) its random numbers.

```
# Create a random tensors with pytorch
random_tensor=torch.rand(size=(3,5))
random_tensor
```

```
tensor([[0.4738, 0.8608, 0.6094, 0.6102, 0.2580],
        [0.5054, 0.8129, 0.0055, 0.7265, 0.3144],
        [0.3131, 0.1041, 0.5862, 0.8680, 0.4186]])
```

```
random_tensor.ndim
```

```
2
```

```
random_tensor.shape
```

```
torch.Size([3, 5])
```

```
random_tensor.dtype
```

```
torch.float32
```

```
random_tensor[0]
```

```
tensor([0.4738, 0.8608, 0.6094, 0.6102, 0.2580])
```

```
random_tensor[1]
```

```
tensor([0.5054, 0.8129, 0.0055, 0.7265, 0.3144])
```

```
random_tensor[1][2]
```

```
tensor(0.0055)
```

```
# Create a random tensor with similar shape to an image tensor
random_image_size_tensor=torch.rand(224,224,3)
random_image_size_tensor
```

```
tensor([[[0.0545, 0.5971, 0.0975],
         [0.0038, 0.2524, 0.7199],
         [0.2787, 0.9990, 0.1462],
         ...,
         [0.0024, 0.5211, 0.6993],
         [0.7047, 0.9425, 0.4253],
         [0.0513, 0.5616, 0.7264]],
        [[0.1143, 0.3742, 0.7347],
         [0.6207, 0.2966, 0.5405],
         [0.9714, 0.4159, 0.6701],
         ...,
         [0.2950, 0.4331, 0.5080],
         [0.7519, 0.2125, 0.5329],
         [0.4635, 0.7193, 0.8016]],

        [[0.4054, 0.0395, 0.3089],
         [0.7886, 0.9496, 0.1078],
         [0.5499, 0.0452, 0.6601],
         ...,
         [0.7272, 0.0779, 0.7841],
         [0.4435, 0.2011, 0.8243],
         [0.8585, 0.6389, 0.5917]],

        ...,

        [[0.7536, 0.5660, 0.7427],
         [0.1534, 0.4416, 0.3621],
         [0.1362, 0.0524, 0.6959],
         ...,
         [0.6748, 0.3933, 0.0173],
         [0.3941, 0.4317, 0.3309],
         [0.3877, 0.6680, 0.0893]],

        [[0.7304, 0.9822, 0.4546],
         [0.8345, 0.0485, 0.6226],
         [0.8370, 0.8803, 0.4537],
         ...,
         [0.3791, 0.1566, 0.3882],
         [0.4686, 0.6364, 0.5655],
         [0.4931, 0.3332, 0.1582]],

        [[0.1372, 0.3039, 0.7534],
         [0.3788, 0.4728, 0.8113],
         [0.6487, 0.1327, 0.6208],
         ...,
         [0.9226, 0.2908, 0.8571],
         [0.6030, 0.3112, 0.5003],
         [0.7293, 0.7231, 0.7127]]])
```

```
random_image_size_tensor.shape
```

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

```
random_image_size_tensor.ndim
```

```
3
```

```
# Creating tensors with ones and zeros
zeros=torch.zeros(3,4)
zeros
```

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

```
zeros.shape,zeros.ndim
```

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

```
ones=torch.ones(3,5)
ones
```

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

```
ones.dtype
```

```
torch.float32
```

```
ones.ndim
```

```
2
```

```
zeros.ndim
```

```
2
```

Creating a range and tensors like

```
# use torch.arange(), torch.range() is deprecated
zero_to_ten_deprecated = torch.range(0, 10) # Note: this may return an error in the future

zero_to_ten=torch.arange(0,10,1)
zero_to_ten
```

```
<ipython-input-42-b3db9713ecab>:2: UserWarning: torch.range is deprecated and will be removed in a future release because its b
  zero_to_ten_deprecated = torch.range(0, 10) # Note: this may return an error in the future
tensor([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

```
ten_zeros=torch.zeros_like(zero_to_ten)
ten_zeros
```

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

Tensor Datatypes

```
float_32_tensor=torch.tensor([3.0,6.0,9.0],
                             dtype=None,
                             device=None,
                             requires_grad=False)
float_32_tensor, float_32_tensor.shape,float_32_tensor.ndim
```

```
(tensor([3., 6., 9.]), torch.Size([3]), 1)
```

```
float_32_tensor.dtype,float_32_tensor.device
```

```
(torch.float32, device(type='cpu'))
```

PyTorch likes calculations between tensors to be on the same device

```
float_16_tensor = torch.tensor([3.0, 6.0, 9.0],
                               dtype=torch.float16) # torch.half would also work
float_16_tensor.dtype
```

```
torch.float16
```

```
tensor=torch.randn(size=(3,4))

print(tensor)
print(f"Shape of tensor :{tensor.shape}")
print(f"Datatype of a tensor: {tensor.dtype}")
print(f"Device tensor is stored on : {tensor.device}")
```

```
tensor([[-2.0590,  0.2000, -0.9437, -0.8007],
        [ 0.1902, -0.0035, -0.0603,  0.2187],
        [ 0.7267,  0.0230, -0.5599,  0.0843]])
Shape of tensor :torch.Size([3, 4])
Datatype of a tensor: torch.float32
Device tensor is stored on : cpu
```

Manipulating Tensor

```
tensor=torch.tensor([1,2,3])
tensor+10
```

```
tensor([11, 12, 13])
```

```
tensor*10
```

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

```
tensor
```

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

```
# Tensors don't change unless reassigned
tensor=tensor+1
tensor
```

```
tensor([2, 3, 4])
```

```
tensor-10
```

```
tensor([-8, -7, -6])
```

```
torch.multiply(tensor,10)
```

```
tensor([20, 30, 40])
```

```
tensor
```

```
tensor([2, 3, 4])
```

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

```
tensor([2, 3, 4]) * tensor([2, 3, 4])
Equals: tensor([ 4,  9, 16])
```

Matrix Multiplication

```
import torch
tensor=torch.tensor([1,2,3,4])
tensor.shape
```

```
torch.Size([4])
```

```
# elelment wise multiplication
tensor*tensor
```

```
tensor([ 1,  4,  9, 16])
```

```
# Matrix multilication
torch.matmul(tensor,tensor)
```

```
tensor(30)
```

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

```
tensor(30)
```

time comparison of matrix multiplication

```
%%time
# Matrix multiplication by hand
# (avoid doing operations with for loops at all cost, they are computationally expensive)
value = 0
for i in range(len(tensor)):
  value+=tensor[i]*tensor[i]
value
```

```
CPU times: user 157 µs, sys: 1.01 ms, total: 1.17 ms
Wall time: 1.13 ms
tensor(30)
```

```
%%time
torch.matmul(tensor,tensor)
```

```
CPU times: user 72 µs, sys: 6 µs, total: 78 µs
Wall time: 83.9 µs
tensor(30)
```

Matrix multplication most common error

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

tensor_B=torch.tensor([[1,2],
                       [3,4],
                       [5,6]],dtype=torch.float32)

torch.matmul(tensor_A,tensor_B)
```

```
---------------------------------------------------------------------------
RuntimeError                              Traceback (most recent call last)
<ipython-input-76-8ced62cd1868> in <cell line: 0>()
      7                               [5,6]],dtype=torch.float32)
      8 
----> 9 torch.matmul(tensor_A,tensor_B)

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

Next steps: Explain error

```
print(tensor_A)
print(tensor_B)
```

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

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

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

```
# 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([[ 5., 11., 17.],
        [11., 25., 39.],
        [17., 39., 61.]])

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

```
# We can also use torch.mm() instead of torch.matmult()
torch.mm(tensor_A,tensor_B.T)
```

```
tensor([[ 5., 11., 17.],
        [11., 25., 39.],
        [17., 39., 61.]])
```

```
# Since the linear layer starts with a random weights matrix, let's make it reproducible (more on this later)
torch.manual_seed(42)

linear=torch.nn.Linear(in_features=2,out_features=6)

x=tensor_A
output=linear(x)
print(x.shape)
print(output)
print(output.shape)
```

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

Finding the min,max,mean,sum

```
x=torch.arange(0,100,10)
x
```

```
tensor([ 0, 10, 20, 30, 40, 50, 60, 70, 80, 90])
```

```
print(f"Minimum : {x.min()}")
print(f"Maximum : {x.max()}")
print(f"Sum : {x.sum()}")
print(f"Mean :{x.type(torch.float).mean()}") # won't work without float datatype
```

```
Minimum : 0
Maximum : 90
Sum : 450
Mean :45.0
```

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

```
(tensor(90), tensor(0), tensor(45.), tensor(450))
```

Positional min/max

```
tensor=torch.arange(0,100,10)
print(f"Tensor{tensor}")

#return index of max and min values
```

```
print(f"Index where max value occurs:{tensor.argmax()}")
print(f"Index where min value occurs:{tensor.argmin()}")
```

```
Tensortensor([ 0, 10, 20, 30, 40, 50, 60, 70, 80, 90])
Index where max value occurs:9
Index where min value occurs:0
```

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

```
(tensor([10., 20., 30., 40., 50., 60., 70., 80., 90.]), torch.float32)
```

```
# Create a float16 tensor
tensor_float16=tensor.type(torch.float16)
tensor_float16,tensor_float16.dtype
```

```
(tensor([10., 20., 30., 40., 50., 60., 70., 80., 90.], dtype=torch.float16),
 torch.float16)
```

```
# create an int8 tensor
tensor_int8=tensor.type(torch.int8)
tensor_int8,tensor_int8.dtype
```

```
(tensor([10, 20, 30, 40, 50, 60, 70, 80, 90], dtype=torch.int8), torch.int8)
```

Reshaping, stacking , squeezing and unsequeezing

```
import torch
x=torch.arange(1.,9.)
x,x.shape
```

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

```
# add an extra dimension
x_reshaped=x.reshape(2,4)
x_reshaped,x_reshaped.shape
```

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

```
z=x.view(8,1) # we can also change the view by passing the dimesion in which we want to see
```

```
z
```

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

```
z=x.view(2,4)
z
```

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

```
z[0][0]=5.
z
```

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

```
x
```

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

```
z=x.view(1,8)
z
```

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

```
# stack
x_stacked=torch.stack([x,x,x,x,x],dim=0)
x_stacked
```

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

```
#stack
x_stacked=torch.stack([x,x,x,x,x,x],dim=1)
x_stacked
```

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

```
x_stacked=torch.stack([x,x,x,x,x,x])
x_stacked
```

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

```
x_reshaped=x_reshaped.reshape(1,8)
print(f"Previous tensor: {x_reshaped}")
print(f"Previous shape : {x_reshaped.shape}")

# Remove extra dimension form 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., 8.]])
Previous shape : torch.Size([1, 8])

New tensor: tensor([5., 2., 3., 4., 5., 6., 7., 8.])
New shape: torch.Size([8])
```

```
# unsqueeze

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., 8.])
Previous shape: torch.Size([8])

New tensor: tensor([[5., 2., 3., 4., 5., 6., 7., 8.]])
New shape: torch.Size([1, 8])
```

```
# unsqueeze

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=1)
print(f"\nNew tensor: {x_unsqueezed}")
print(f"New shape: {x_unsqueezed.shape}")
```

```
Previous tensor: tensor([5., 2., 3., 4., 5., 6., 7., 8.])
Previous shape: torch.Size([8])

New tensor: tensor([[5.],
        [2.],
        [3.],
        [4.],
        [5.],
        [6.],
        [7.],
        [8.]])
New shape: torch.Size([8, 1])
```

```
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=1)
print(f"\nNew tensor: {x_unsqueezed}")
print(f"New shape: {x_unsqueezed.shape}")
```

```
Previous tensor: tensor([5., 2., 3., 4., 5., 6., 7., 8.])
Previous shape: torch.Size([8])

New tensor: tensor([[5.],
        [2.],
        [3.],
        [4.],
        [5.],
        [6.],
        [7.],
        [8.]])
New shape: torch.Size([8, 1])
```

```
# Permute(permute)
x=torch.rand(size=(224,224,3))
x_permuted=x.permute(2,0,1)
print(f"Previous shaope: {x.shape}")
print(f"New shape: {x_permuted.shape}")
```

```
Previous shaope: torch.Size([224, 224, 3])
New shape: torch.Size([3, 224, 224])
```

Indexing(select data from tensors)

```
import torch
x=torch.arange(1,10)
x=x.reshape(1,3,3)
x,x.shape
```

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

```
# 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
```

```
# 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])
```

Pytorch tensors and NumPy

```
import torch
import numpy as np
array=np.arange(1.0,8.0)
tensor=torch.from_numpy(array)
array,tensor
```

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

Note: By default, NumPy arrays are created with the datatype float64 and if you convert it to a PyTorch tensor, it'll keep the same datatype (as above).

However, many PyTorch calculations default to using float32.

So if you want to convert your NumPy array (float64) -> PyTorch tensor (float64) -> PyTorch tensor (float32), you can use tensor = torch.from_numpy(array).type(torch.float32).

```
tensor=torch.from_numpy(array).type(torch.float32)
tensor,tensor.dtype
```

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

```
# tensor to numpy
tensor=torch.ones(10)
numpy_array=tensor.numpy()
tensor,numpy_array
```

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

Reproducibility

```
import torch

A=torch.rand(3,4)
B=torch.rand(3,4)

print(f"{A}")
print(f"{B}")
print(f"Does Tensor A equals to B? ")
A==B
```

```
tensor([[0.7588, 0.3036, 0.6450, 0.2378],
        [0.0098, 0.4922, 0.0686, 0.6345],
        [0.1099, 0.5221, 0.1246, 0.3182]])
tensor([[0.3700, 0.5144, 0.8704, 0.1053],
        [0.4608, 0.4464, 0.7738, 0.7564],
        [0.3194, 0.0896, 0.1095, 0.3024]])
Does Tensor A equals to B?
tensor([[False, False, False, False],
        [False, False, False, False],
        [False, False, False, False]])
```

```
import torch
import random

# Set the random seed
RANDOM_SEED=420 # 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([[0.8054, 0.1990, 0.9759, 0.1028],
        [0.3475, 0.1554, 0.8856, 0.6876],
        [0.2506, 0.1133, 0.2105, 0.4035]])

Tensor D:
tensor([[0.8054, 0.1990, 0.9759, 0.1028],
        [0.3475, 0.1554, 0.8856, 0.6876],
        [0.2506, 0.1133, 0.2105, 0.4035]])

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

```
!nvidia-smi
```

```
Sun Feb 23 10:31:46 2025
+-----------------------------------------------------------------------------------------+
| NVIDIA-SMI 550.54.15              Driver Version: 550.54.15      CUDA Version: 12.4     |
|-----------------------------------------+------------------------+----------------------+
| 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   41C    P8              9W /   70W |       0MiB /  15360MiB |      0%      Default |
|                                         |                        |                  N/A |
+-----------------------------------------+------------------------+----------------------+

+-----------------------------------------------------------------------------------------+
| Processes:                                                                              |
|  GPU   GI   CI        PID   Type   Process name                              GPU Memory |
|        ID   ID                                                               Usage      |
|=========================================================================================|
|  No running processes found                                                             |
+-----------------------------------------------------------------------------------------+
```

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

```
True
```

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

```
'cuda'
```

```
# count the number of devices
torch.cuda.device_count()
```

```
1
```

Start coding or generate with AI.