Pytorch Fundamental

In [2]:
```
import torch
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
```

Introduction to tensor

Creating Tensor

scaler

Scalars are single numbers and are an example of a 0th-order tensor.

In [3]:
```
scaler = torch.tensor(7)
scaler
```

Out[3]:
```
tensor(7)
```

In [4]:
```
scaler.ndim
```

Out[4]:
```
0
```

In [5]:
```
scaler.item()
```

Out[5]:
```
7
```

Vector

```
vectors are a fundamental concept in machine learning that allow us to
represent data in a structured and efficient manner. Vectors are used
in
various machine learning algorithms and operations such as regression,
classification, clustering, and dimensionality reduction
```

In [6]:
```
vector = torch.tensor([7,7])
vector
```

Out[6]:
```
tensor([7, 7])
```

In [7]:
```
vector.ndim
```

Out[7]:
```
1
```

In [8]:
```
vector.shape
```
Out[8]:
```
torch.Size([2])
```

###What are Matrices? Matrices are rectangular arrays consisting of numbers and can be seen as 2nd-order tensors. If m and n are positive integers, that is m, n ∈ ℕ then the m×n matrix contains m*n numbers of elements, with m number of rows and n number of columns.

In [9]:
```
matrix = torch.tensor([[3, 5],
                       [1, 3]])
matrix.ndim
```
Out[9]:
```
2
```

In [10]:
```
matrix[1]
```
Out[10]:
```
tensor([1, 3])
```

In [11]:
```
matrix[0]
```
Out[11]:
```
tensor([3, 5])
```

In [12]:
```
matrix.shape
```
Out[12]:
```
torch.Size([2, 2])
```

Tensor

```
The more general entity of a tensor encapsulates the scalar,
vector and the matrix. It is sometimes necessary—both in the physical
sciences and machine learning—to
make use of tensors with order that exceeds two.
```

In [13]:
```
TENSOR = torch.tensor([[[1,2,3],
                        [3,4,5],
                        [2,4,5]]])
```

In [14]:
```
TENSOR.ndim
```
Out[14]:
```
3
```

In [15]:
```
TENSOR.shape
```
Out[15]:
```
torch.Size([1, 3, 3])
```

In [16]:
```
TENSOR[0]
```
Out[16]:
```
tensor([[1, 2, 3],
        [3, 4, 5],
        [2, 4, 5]])
```

In [17]:
```
TENSOR = torch.tensor([
    [
    [[1,2,3],[3,4,5],[2,4,5]],
    [[41,42,34],[33,42,25],[12,24,45]],
    ],
    [
    [[11,21,3],[13,14,15],[21,41,15]],
    [[12,12,13],[31,41,51],[12,41,51]],
    ]
                  ])
```

In [18]:
```
TENSOR.shape
```

Out[18]:
```
torch.Size([2, 2, 3, 3])
```

In [19]:
```
TENSOR.ndim
```

Out[19]:
```
4
```

In [20]:
```
TENSOR
```

Out[20]:
```
tensor([[[[ 1,  2,  3],
          [ 3,  4,  5],
          [ 2,  4,  5]],

         [[41, 42, 34],
          [33, 42, 25],
          [12, 24, 45]]],


        [[[11, 21,  3],
          [13, 14, 15],
          [21, 41, 15]],

         [[12, 12, 13],
          [31, 41, 51],
          [12, 41, 51]]]])
```

Let's Access all the element

TENSOR tensor(

```
[[[[ 1,  2,  3],
   [ 3,  4,  5],
   [ 2,  4,  5]],

  [[41, 42, 34],
   [33, 42, 25],
```

In [21]:
```
TENSOR
```

Out[21]:
```
tensor([[[[ 1,  2,  3],
          [ 3,  4,  5],
          [ 2,  4,  5]],

         [[41, 42, 34],
          [33, 42, 25],
          [12, 24, 45]]],


        [[[11, 21,  3],
          [13, 14, 15],
          [21, 41, 15]],

         [[12, 12, 13],
          [31, 41, 51],
          [12, 41, 51]]]])
```

In [22]:
```
TENSOR[0]
```

Out[22]:
```
tensor([[[ 1,  2,  3],
         [ 3,  4,  5],
         [ 2,  4,  5]],

        [[41, 42, 34],
         [33, 42, 25],
         [12, 24, 45]]])
```

In [23]:
```
TENSOR[0][0]
```

Out[23]:
```
tensor([[1, 2, 3],
        [3, 4, 5],
        [2, 4, 5]])
```

In [24]:
```
TENSOR[0][1]
```

Out[24]:
```
tensor([[41, 42, 34],
        [33, 42, 25],
        [12, 24, 45]])
```

In [25]:
```
TENSOR[0][0][2]
```

Out[25]:
```
tensor([2, 4, 5])
```

In [26]:
```
TENSOR[0][1][2]
```

Out[26]:
```
tensor([12, 24, 45])
```

In [26]:
```
```

In [27]:
```
TENSOR = torch.tensor([
    [
    [
        [1,2,3],
        [3,4,5],
        [2,4,5]
        ],
    [
        [1,2,3],
        [3,4,5],
        [2,4,5]
        ],
    ],
    [
    [
        [1,2,3],
        [3,4,5],
        [2,4,5]
        ],
    [
        [1,2,3],
        [3,4,5],
        [2,4,5]
     ],
    ]
                          ])
```

In [28]:
```
TENSOR
```

Out[28]:
```
tensor([[[[1, 2, 3],
          [3, 4, 5],
          [2, 4, 5]],

         [[1, 2, 3],
          [3, 4, 5],
          [2, 4, 5]]],


        [[[1, 2, 3],
          [3, 4, 5],
          [2, 4, 5]],

         [[1, 2, 3],
          [3, 4, 5],
          [2, 4, 5]]]])
```

Random Tensors

```
provides a tensor object containing the random values between the spec
ified interval or, by default, between 0 to 1, [0,1] interval.
```

In [29]:
```
random_tensor = torch.rand(3,4) #3 x 4 , it means 3 Row and 4 Columns and this
random_tensor
```

Out[29]:
```
tensor([[0.1555, 0.6611, 0.8521, 0.6543],
        [0.8501, 0.7905, 0.6956, 0.6094],
        [0.7979, 0.7622, 0.3044, 0.6599]])
```

In [30]:
```
print(f"Number of Dimensions : {random_tensor.ndim}")
```

```
Number of Dimensions : 2
```

In [31]:
```
random_tensor = torch.rand(10,4)
random_tensor
```

Out[31]:
```
tensor([[0.7423, 0.9985, 0.9150, 0.7221],
        [0.4301, 0.0356, 0.3035, 0.7981],
        [0.3852, 0.6366, 0.3143, 0.1793],
        [0.5219, 0.4274, 0.7666, 0.3192],
        [0.5333, 0.8782, 0.6884, 0.6246],
        [0.4803, 0.4766, 0.5833, 0.0731],
        [0.4992, 0.6729, 0.7338, 0.3453],
        [0.8339, 0.0854, 0.5609, 0.7717],
        [0.1019, 0.3738, 0.4101, 0.0057],
        [0.6424, 0.2931, 0.9416, 0.9244]])
```

In [32]:
```
random_tensor = torch.rand(10,4,2) ## it will give  10 sets of 4x2 matrices
random_tensor
```

Out[32]:
```
tensor([[[0.6555, 0.8369],
         [0.3661, 0.4313],
         [0.7074, 0.2762],
         [0.6371, 0.4795]],

        [[0.2432, 0.9903],
         [0.6595, 0.2632],
         [0.8863, 0.5988],
         [0.9353, 0.5746]],

        [[0.1943, 0.6383],
         [0.6846, 0.1550],
         [0.4706, 0.2822],
         [0.4311, 0.8311]],

        [[0.4610, 0.4219],
         [0.5375, 0.4898],
         [0.4251, 0.4404],
         [0.2247, 0.3211]],

        [[0.7954, 0.4134],
         [0.0024, 0.1254],
         [0.1912, 0.2479],
         [0.8801, 0.7451]],

        [[0.8758, 0.6037],
         [0.7934, 0.7479],
         [0.7210, 0.0930],
         [0.1455, 0.4289]],

        [[0.2269, 0.6022],
         [0.8977, 0.7757],
         [0.6741, 0.3966],
         [0.2451, 0.2244]],

        [[0.5047, 0.8631],
         [0.2897, 0.2110],
         [0.4559, 0.6426],
         [0.1290, 0.0875]],

        [[0.9939, 0.0509],
         [0.1085, 0.7873],
         [0.2592, 0.4582],
         [0.4326, 0.2316]],

        [[0.4500, 0.0749],
         [0.3660, 0.4506],
         [0.3715, 0.3855],
         [0.6756, 0.7270]]])
```

In [33]:
```
print(f"Number of Dimension : {random_tensor.ndim}")
```

```
Number of Dimension : 3
```

In [34]:
```
sets ,row,col= random_tensor.shape
print(f"the shape of the matrix are :{sets} sets and {row}x{col} matrix")
```

```
the shape of the matrix are :10 sets and 4x2 matrix
```

Let's Create a random tensor with similar shape to an image tensor

In [35]:
```
random_image_size_tensor = torch.rand(size = (3,224,224))## color channel and h
random_image_size_tensor.shape
```

Out[35]:
```
torch.Size([3, 224, 224])
```

In [36]:
```
random_image_size_tensor.ndim
```

Out[36]:
```
3
```

In [37]:
```
random_image_size_tensor[0]
```

Out[37]:
```
tensor([[0.9558, 0.7138, 0.5210,  ..., 0.9366, 0.7285, 0.9719],
        [0.8308, 0.3797, 0.3006,  ..., 0.5096, 0.0447, 0.7332],
        [0.5302, 0.1362, 0.0955,  ..., 0.2957, 0.1420, 0.5451],
        ...,
        [0.2946, 0.7079, 0.6387,  ..., 0.7594, 0.7047, 0.9530],
        [0.2482, 0.6384, 0.1503,  ..., 0.4420, 0.2512, 0.6432],
        [0.9967, 0.7470, 0.6395,  ..., 0.6510, 0.9447, 0.2433]])
```

In [38]:
```
random_image_size_tensor[1]
```

Out[38]:
```
tensor([[0.4446, 0.9258, 0.5580,  ..., 0.3605, 0.7000, 0.5654],
        [0.2057, 0.3944, 0.7044,  ..., 0.5885, 0.9990, 0.4112],
        [0.2683, 0.9611, 0.0097,  ..., 0.6400, 0.0816, 0.5237],
        ...,
        [0.3357, 0.3686, 0.5761,  ..., 0.3601, 0.1931, 0.9288],
        [0.1208, 0.4742, 0.0981,  ..., 0.1951, 0.4322, 0.0799],
        [0.5676, 0.0457, 0.5716,  ..., 0.9094, 0.6390, 0.6246]])
```

In [39]:
```
random_image_size_tensor[2]
```

Out[39]:
```
tensor([[0.8492, 0.9278, 0.5319,  ..., 0.0578, 0.6819, 0.3238],
        [0.7797, 0.6014, 0.9521,  ..., 0.7569, 0.3700, 0.7112],
        [0.0429, 0.6293, 0.9097,  ..., 0.7003, 0.2087, 0.7482],
        ...,
        [0.8864, 0.6682, 0.7647,  ..., 0.1919, 0.2678, 0.8757],
        [0.4225, 0.4187, 0.0089,  ..., 0.1587, 0.7010, 0.3905],
        [0.3954, 0.8494, 0.0136,  ..., 0.5943, 0.4006, 0.2670]])
```

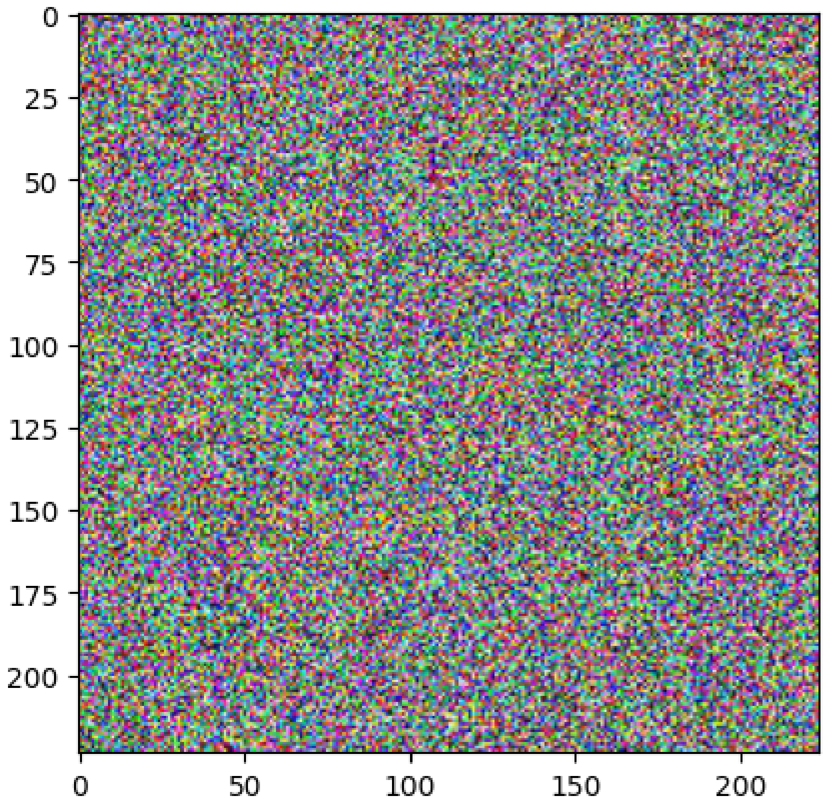
In [40]:
```
random_image_size_tensor
```

Out[40]:
```
tensor([[[0.9558, 0.7138, 0.5210,  ..., 0.9366, 0.7285, 0.9719],
         [0.8308, 0.3797, 0.3006,  ..., 0.5096, 0.0447, 0.7332],
         [0.5302, 0.1362, 0.0955,  ..., 0.2957, 0.1420, 0.5451],
         ...,
         [0.2946, 0.7079, 0.6387,  ..., 0.7594, 0.7047, 0.9530],
         [0.2482, 0.6384, 0.1503,  ..., 0.4420, 0.2512, 0.6432],
         [0.9967, 0.7470, 0.6395,  ..., 0.6510, 0.9447, 0.2433]],

        [[0.4446, 0.9258, 0.5580,  ..., 0.3605, 0.7000, 0.5654],
         [0.2057, 0.3944, 0.7044,  ..., 0.5885, 0.9990, 0.4112],
         [0.2683, 0.9611, 0.0097,  ..., 0.6400, 0.0816, 0.5237],
         ...,
         [0.3357, 0.3686, 0.5761,  ..., 0.3601, 0.1931, 0.9288],
         [0.1208, 0.4742, 0.0981,  ..., 0.1951, 0.4322, 0.0799],
         [0.5676, 0.0457, 0.5716,  ..., 0.9094, 0.6390, 0.6246]],

        [[0.8492, 0.9278, 0.5319,  ..., 0.0578, 0.6819, 0.3238],
         [0.7797, 0.6014, 0.9521,  ..., 0.7569, 0.3700, 0.7112],
         [0.0429, 0.6293, 0.9097,  ..., 0.7003, 0.2087, 0.7482],
         ...,
         [0.8864, 0.6682, 0.7647,  ..., 0.1919, 0.2678, 0.8757],
         [0.4225, 0.4187, 0.0089,  ..., 0.1587, 0.7010, 0.3905],
         [0.3954, 0.8494, 0.0136,  ..., 0.5943, 0.4006, 0.2670]]])
```

In [41]:
```
random_image_size_tensor.shape
```

Out[41]:
```
torch.Size([3, 224, 224])
```

In [42]:
```
random_image_size_tensor = torch.rand(size = (224,224,3))
imag = random_image_size_tensor
plt.imshow(imag)
plt.show()
```



In [43]:
```
#it's giving the error becuase i have declear 3 channel and this our 4 that's w
```

In [44]:
```
import matplotlib.pyplot as plt
import torch

# Generate a random tensor with image data
random_image_size_tensor = torch.randn(5, 3, 64, 64)  # Genrateing 5 images, 3

# Loop through the images in the tensor and visualize them
for i in range(random_image_size_tensor.shape[0]):
    img_data = random_image_size_tensor[i].permute(1, 2, 0)  # Adjust the permu
    plt.figure(figsize=(6, 4))
    plt.imshow(img_data)
    plt.title(f"Random Image {i + 1}")
    plt.axis('off')  # Turn off the axis
    plt.show()
```

```
WARNING:matplotlib.image:Clipping input data to the valid range for imshow
with RGB data ([0..1] for floats or [0..255] for integers).
```

Zeros and Ones

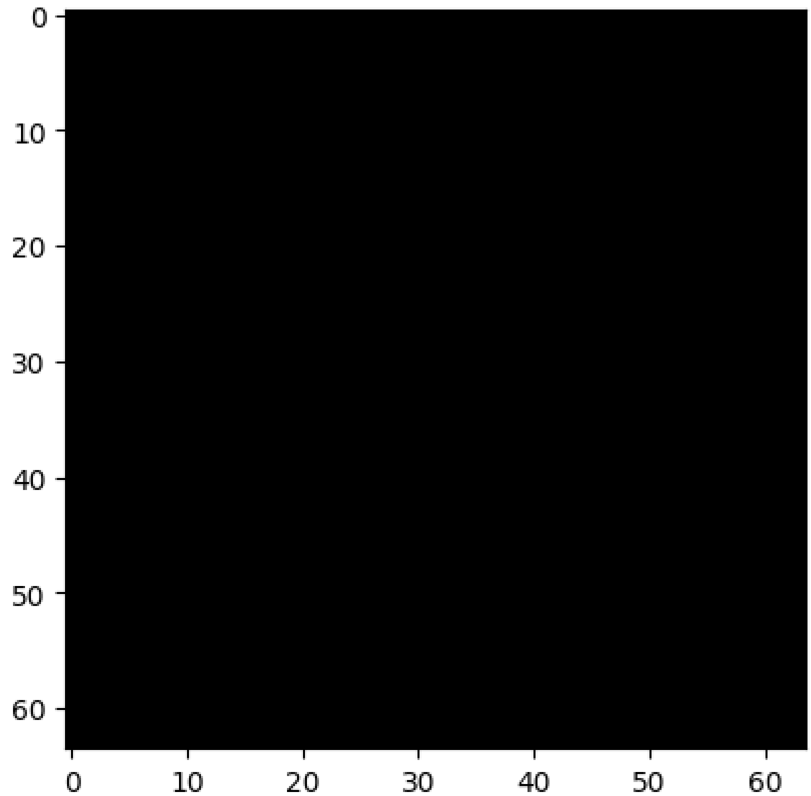
In [45]:
```
zeros = torch.zeros(size = (3,4))
zeros
```

Out[45]:
```
tensor([[0., 0., 0., 0.],
        [0., 0., 0., 0.],
        [0., 0., 0., 0.]])
```

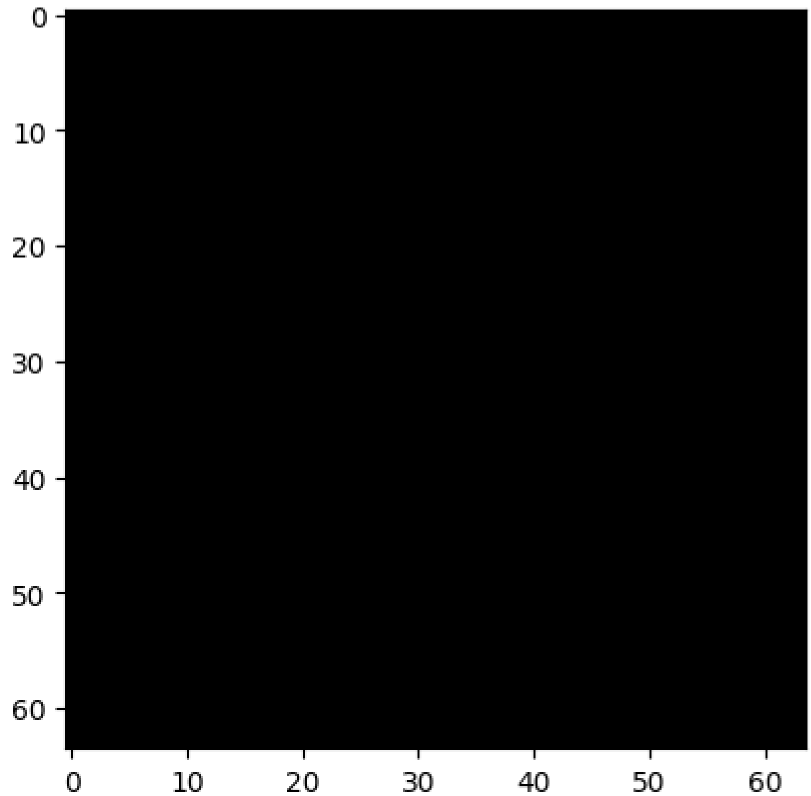
In [46]:
```
ones = torch.ones(size = (4,5,2)) ## here u can see 4 set and 5x2 matrix
ones
```

Out[46]:
```
tensor([[[1., 1.],
         [1., 1.],
         [1., 1.],
         [1., 1.],
         [1., 1.]],

        [[1., 1.],
         [1., 1.],
         [1., 1.],
         [1., 1.],
         [1., 1.]],

        [[1., 1.],
         [1., 1.],
         [1., 1.],
         [1., 1.],
         [1., 1.]],

        [[1., 1.],
         [1., 1.],
         [1., 1.],
         [1., 1.],
         [1., 1.]]])
```

In [47]:
```
### Ones matrix plot
### let's plot
img  = torch.ones(size = (64,64))
plt.imshow(img, cmap='gray')
plt.show()
```



In [48]:
```
### Zeros matrix plot
### let's plot
img  = torch.zeros(size = (64,64))
plt.imshow(img, cmap = 'gray')
plt.show()
```



Let's Play with range

In [49]:
```
torch.arange(0,10) ## here i am starting with 0 and end with 10
```

Out[49]:
```
tensor([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

In [50]:
```
torch.arange(start=0, end=10, step=2)
```

Out[50]:
```
tensor([0, 2, 4, 6, 8])
```

In [51]:
```
## Let's Create Tensor like
input = torch.empty(2,2, 3)
torch.zeros_like(input)
```

Out[51]:
```
tensor([[[0., 0., 0.],
         [0., 0., 0.]],

        [[0., 0., 0.],
         [0., 0., 0.]]])
```

In [52]:
```
input = torch.empty(2,2, 3)
torch.ones_like(input)
```

Out[52]:
```
tensor([[[1., 1., 1.],
         [1., 1., 1.]],

        [[1., 1., 1.],
         [1., 1., 1.]]])
```

Let's Play With Tensor Datatypes

In [53]:
```
float_32_tensor = torch.tensor([2,4,5.],
                      dtype = None)
float_32_tensor
```

Out[53]:
```
tensor([2., 4., 5.])
```

In [54]:
```
float_32_tensor.dtype ### i haven't decide any data type but it is by defult gi
```

Out[54]:
```
torch.float32
```

In [55]:
```
## here i am going to define the data types torch.float16
float_16_tensor = torch.tensor([2,4,5.],
                      dtype = torch.float16)
float_16_tensor
```

Out[55]:
```
tensor([2., 4., 5.], dtype=torch.float16)
```

In [56]:
```
float_16_tensor.dtype
```

Out[56]:
```
torch.float16
```

In [57]:
```
tensor = torch.tensor([2,4,5.],
                      dtype = None,## float16,float32
                      device=None, ## we can select the device as well
                      requires_grad=False)
tensor
```

Out[57]:
```
tensor([2., 4., 5.])
```

converting float32 into float16 tensor

In [58]:
```
float_16 = float_32_tensor.type(torch.float16)
float_16.dtype
```

Out[58]:
```
torch.float16
```

In [59]:
```
### Let's multiply float 16 and float32 tensor
float_16*float_32_tensor
```

Out[59]:
```
tensor([ 4., 16., 25.])
```

In [60]:
```
### let's make int 32 tensor
int_32_tensor = torch.tensor([1,2,4],
                             dtype = torch.int32)
int_32_tensor
```

Out[60]:
```
tensor([1, 2, 4], dtype=torch.int32)
```

In [61]:
```
## let's multiply int32 and float16 tensor
int_32_tensor*float_16
```

Out[61]:
```
tensor([ 2.,  8., 20.], dtype=torch.float16)
```

In [62]:
```
## let's change the shape of int and then multiply with float16
### let's make int 32 tensor
int_32_tensor = torch.tensor([1,2,4,32,54],
                             dtype = torch.int32)
int_32_tensor*float_16
```

```
---------------------------------------------------------------------------
RuntimeError                              Traceback (most recent call last)
<ipython-input-62-aa825c248fd8> in <cell line: 5>()
      3 int_32_tensor = torch.tensor([1,2,4,32,54],
      4                              dtype = torch.int32)
----> 5 int_32_tensor*float_16

RuntimeError: The size of tensor a (5) must match the size of tensor b (3) at
non-singleton dimension 0
```

In [63]:
```
## we are not able to multiply becuase the shape of the float and int shape is
```

Manipulating Tensor (Tensor operation)

- Addition
- Substraction
- Multiplication
- Division
- Matrix Multiplication

In [64]:
```
## frist let's create a tensor
tensor = torch.tensor([1, 2, 3])
tensor
```

Out[64]:
```
tensor([1, 2, 3])
```

Additon Operation

In [65]:
```
print(tensor)
print('Addition Opereation')
print(tensor+10)
print(tensor+23)
print(tensor+20)
print(tensor+100)
```

```
tensor([1, 2, 3])
Addition Opereation
tensor([11, 12, 13])
tensor([24, 25, 26])
tensor([21, 22, 23])
tensor([101, 102, 103])
```

In [66]:
```
print(tensor)
print('Substraction Opereation')
print(tensor-10)
print(tensor-23)
print(tensor-20)
print(tensor-100)
```

```
tensor([1, 2, 3])
Substraction Opereation
tensor([-9, -8, -7])
tensor([-22, -21, -20])
tensor([-19, -18, -17])
tensor([-99, -98, -97])
```

In [67]:
```
print(tensor)
print('Mupltiplication Opereation')
print(tensor*10)
print(tensor*23)
print(tensor*20)
print(tensor*100)
```

```
tensor([1, 2, 3])
Mupltiplication Opereation
tensor([10, 20, 30])
tensor([23, 46, 69])
tensor([20, 40, 60])
tensor([100, 200, 300])
```

In [68]:
```
## we have a in-built function as well
torch.mul(tensor,10)
```

Out[68]:
```
tensor([10, 20, 30])
```

In [69]:
```
torch.add(tensor,10)
```

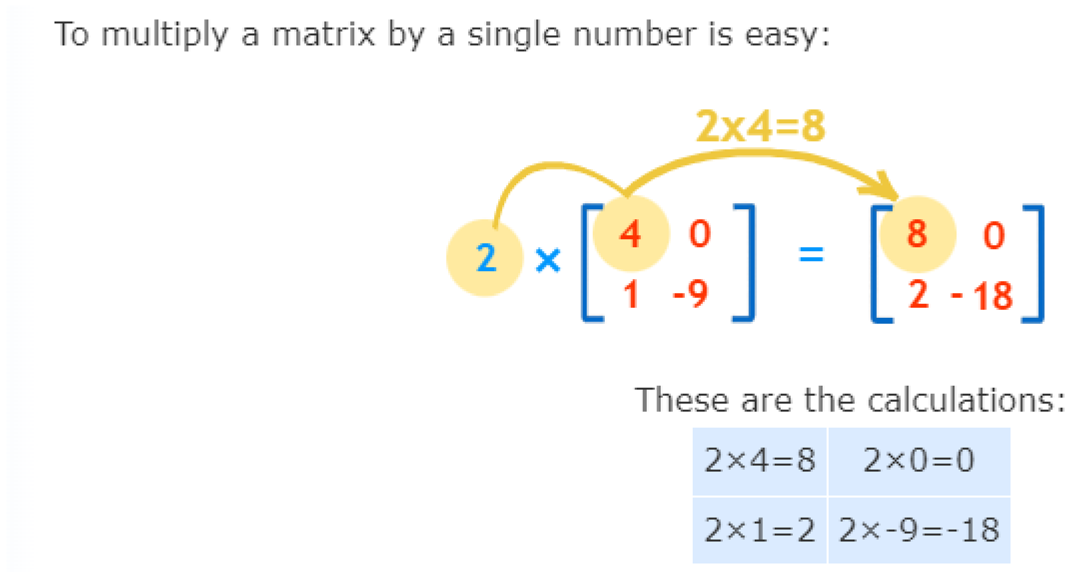
Out[69]:
```
tensor([11, 12, 13])
```

In [70]:
```
torch.sub(tensor,10)
```
Out[70]:
```
tensor([-9, -8, -7])
```

In [71]:
```
torch.div(tensor,10)
```
Out[71]:
```
tensor([0.1000, 0.2000, 0.3000])
```

Matrix Multiplication Two main way of performing multiplication in neural networks and deep learning:

1. Element-wise Multiplication



2. Matrix Multiplication(dot product)

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \times \begin{bmatrix} 7 & 8 \\ 9 & 10 \\ 11 & 12 \end{bmatrix} = \begin{bmatrix} 58 & 64 \end{bmatrix}$$

$$(1, 2, 3) \bullet (8, 10, 12) = 1\times8 + 2\times10 + 3\times12 = 64$$

We can do the same thing for the **2nd row** and **1st column**:

$$(4, 5, 6) \bullet (7, 9, 11) = 4\times7 + 5\times9 + 6\times11 = 139$$

And for the **2nd row** and **2nd column**:

$$(4, 5, 6) \bullet (8, 10, 12) = 4\times8 + 5\times10 + 6\times12 = 154$$

And we get:

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \times \begin{bmatrix} 7 & 8 \\ 9 & 10 \\ 11 & 12 \end{bmatrix} = \begin{bmatrix} 58 & 64 \\ 139 & 154 \end{bmatrix}$$

In [72]:
```
tensor
```

Out[72]:
```
tensor([1, 2, 3])
```

Element Wise MULTIPLICATION

In [73]:
```
print(tensor, '*', tensor)
print(f"Equal :{tensor * tensor}")
```

```
tensor([1, 2, 3]) * tensor([1, 2, 3])
Equal :tensor([1, 4, 9])
```

Matrix multiplication (dot product)

There are two main rules that performing matrix multiplication needs to satisfy: 1. the **inner dimensions** must match

```
* (3x4) @ (3x4) --->this will not work
* (3x4) @ (4X3) --->this will work
* (13x3) @ (3x40) --->this will work
* (3x14) @ (14x4) --->this will work
* (12x4) @ (12x4) --->this will not work
* (12x4) @ (4x12) --->this will work
```

In [74]:
```
torch.matmul(tensor,tensor)
```

Out[74]:
```
tensor(14)
```

In [75]:
```
matrix = torch.tensor([[3, 5],
                       [1, 3]])
torch.matmul(matrix,matrix)
```

Out[75]:
```
tensor([[14, 30],
        [ 6, 14]])
```

In [76]:
```
## this will not work
torch.matmul(torch.rand(3,4),torch.rand(3,4))
```

```
---------------------------------------------------------------------------
RuntimeError                              Traceback (most recent call last)
<ipython-input-76-9788f92496fc> in <cell line: 2>()
      1 ## this will not work
----> 2 torch.matmul(torch.rand(3,4),torch.rand(3,4))

RuntimeError: mat1 and mat2 shapes cannot be multiplied (3x4 and 3x4)
```

In [77]:
```
torch.matmul(torch.rand(13, 3), torch.rand(3, 10))
```

Out[77]:
```
tensor([[0.6275, 0.7144, 1.0201, 0.9469, 1.1527, 0.7308, 0.5921, 0.6045, 0.54
11,
         0.3177],
        [0.4205, 0.5360, 0.9325, 0.8748, 1.1833, 0.8371, 0.8474, 0.4270, 0.89
89,
         0.4196],
        [0.4533, 0.5359, 0.7764, 0.7203, 0.8903, 0.5692, 0.4871, 0.4519, 0.45
75,
         0.2612],
        [0.6219, 0.4699, 0.8939, 0.8603, 1.0972, 0.8318, 0.6104, 0.3596, 0.64
49,
         0.2373],
        [0.3549, 0.4244, 0.7785, 0.7344, 1.0029, 0.7280, 0.7287, 0.3314, 0.78
47,
         0.3494],
        [0.5653, 0.6821, 0.9738, 0.9016, 1.1106, 0.7015, 0.6036, 0.5776, 0.56
11,
         0.3292],
        [0.6791, 0.6375, 0.9990, 0.9418, 1.1523, 0.7874, 0.5750, 0.5240, 0.54
99,
         0.2670],
        [0.4671, 0.9805, 1.2744, 1.1474, 1.4766, 0.8389, 1.0047, 0.8538, 0.94
79,
         0.6334],
        [0.2607, 0.8078, 0.9336, 0.8203, 1.0442, 0.5097, 0.7296, 0.7234, 0.65
01,
         0.5187],
        [0.6779, 0.5628, 0.9447, 0.8996, 1.1067, 0.7916, 0.5475, 0.4518, 0.54
08,
         0.2286],
        [0.5454, 0.4974, 0.9346, 0.8916, 1.1755, 0.8749, 0.7516, 0.3836, 0.80
36,
         0.3285],
        [0.4618, 0.6180, 1.0407, 0.9725, 1.3090, 0.9097, 0.9322, 0.4981, 0.97
93,
         0.4722],
        [0.5648, 0.5723, 1.0234, 0.9696, 1.2778, 0.9265, 0.8293, 0.4504, 0.87
60,
         0.3822]])
```

Finding the min, max, mean, sum etc(Tensor Aggregation)

In [78]:
```
## Let's Create a tensor
x = torch.arange(0.,100,10)
x
```

Out[78]:
```
tensor([ 0., 10., 20., 30., 40., 50., 60., 70., 80., 90.])
```

In [79]:
```
### find the minmum
torch.min(x), x.min()
```
Out[79]:
```
(tensor(0.), tensor(0.))
```

In [80]:
```
### finding the max
torch.max(x), x.max()
```
Out[80]:
```
(tensor(90.), tensor(90.))
```

In [81]:
```
### find the mean
torch.mean(x)
```
Out[81]:
```
tensor(45.)
```

In [82]:
```
x.mean()
```
Out[82]:
```
tensor(45.)
```

In [83]:
```
### find the sum

torch.sum(x),x.sum()
```
Out[83]:
```
(tensor(450.), tensor(450.))
```

Finding the positional min and max

In [84]:
```
x = torch.arange(3,100,10)
x
```
Out[84]:
```
tensor([ 3, 13, 23, 33, 43, 53, 63, 73, 83, 93])
```

In [84]:
```
```

Argmin()

```
Find the position in tensor that has the minimum value with argmin() -
-> returns the index position of target tensor where  minimum value oc
cure
```

In [85]:
```
x.argmin()
## this will give the position of the minmum value
#tensor([ 3, 13, 23, 33, 43, 53, 63, 73, 83, 93])
# here we have 3 min value at index 0 then this will return the 0
```
Out[85]:
```
tensor(0)
```

In [86]:
```
## Let's access the item based of it's index value
x[0]
```

Out[86]:
```
tensor(3)
```

In [87]:
```
x.argmax()
## this will give the position of the minmum value
#tensor([ 3, 13, 23, 33, 43, 53, 63, 73, 83, 93])
# here we have 93 min value at index 9 then this will return the 9
```

Out[87]:
```
tensor(9)
```

In [88]:
```
## Let's access the item based of it's index value
x[9]
```

Out[88]:
```
tensor(93)
```

Reshaping , Stacking, Squeezing and Unsqueezing tensor

- Reshaping - reshapes an input tensor to defined shape
- View - Retrun a view of an input tensor of certain shape but keep the same memory as the original tensor
- Stacking - combine multiple tensor on top of each other (vstack) or side by side(hstack)
- Squeeze - Remove all '1' dimensions from a tensor
- Unsqueeze - add a '1' dimension to a target tensor
- Permute - Return a view of the input with dimensions permuted(swapped) in a certain way

In [108]:
```
### Frist Let's create a tensor
import torch
x = torch.arange(1.,13)
x,x.shape
```

Out[108]:
```
(tensor([ 1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9., 10., 11., 12.]),
 torch.Size([12]))
```

####Reshaping - reshapes an input tensor to defined shape

In [109]:
```
## Let's add a extra dimenstion
x_reshaped1 = x.reshape(1,12)
x_reshaped1,x_reshaped1.shape
```

Out[109]:
```
(tensor([[ 1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9., 10., 11., 12.]]),
 torch.Size([1, 12]))
```

In [110]:
```
## Let's add a extra dimenstion
x_reshaped = x.reshape(2,6)
x_reshaped,x_reshaped.shape
```

Out[110]:
```
(tensor([[ 1.,  2.,  3.,  4.,  5.,  6.],
         [ 7.,  8.,  9., 10., 11., 12.]]),
 torch.Size([2, 6]))
```

In [111]:
```
x_reshaped = x.reshape(4,3)
x_reshaped,x_reshaped.shape
```

Out[111]:
```
(tensor([[ 1.,  2.,  3.],
         [ 4.,  5.,  6.],
         [ 7.,  8.,  9.],
         [10., 11., 12.]]),
 torch.Size([4, 3]))
```

View -

```
Retrun a view of an input tensor of certain shape but keep the same me
mory as the original tensor
```

In [112]:
```
z = x.view(1,12)
z, z.shape
```

Out[112]:
```
(tensor([[ 1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9., 10., 11., 12.]]),
 torch.Size([1, 12]))
```

In [113]:
```
### Changing z changes x (because a view of a tensor share the same memory as t
```

In [114]:
```
z[:,0] = 5
z,x
```

Out[114]:
```
(tensor([[ 5.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9., 10., 11., 12.]]),
 tensor([ 5.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9., 10., 11., 12.]))
```

###Stacking - combine multiple tensor on top of each other (vstack) or side by side(hstack)

In [115]:
```
## Stack tensors on top of each other
### vertical stack
x_stacked = torch.stack([x,x,x,x],dim=0)
x_stacked,x_stacked.shape
```

Out[115]:
```
(tensor([[ 5.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9., 10., 11., 12.],
         [ 5.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9., 10., 11., 12.],
         [ 5.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9., 10., 11., 12.],
         [ 5.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9., 10., 11., 12.]]),
 torch.Size([4, 12]))
```

In [116]:
```
## stacking in tensor horizontally
x_stacked = torch.stack([x,x,x,x],dim=1)
x_stacked,x_stacked.shape
```

Out[116]:
```
(tensor([[ 5.,  5.,  5.,  5.],
         [ 2.,  2.,  2.,  2.],
         [ 3.,  3.,  3.,  3.],
         [ 4.,  4.,  4.,  4.],
         [ 5.,  5.,  5.,  5.],
         [ 6.,  6.,  6.,  6.],
         [ 7.,  7.,  7.,  7.],
         [ 8.,  8.,  8.,  8.],
         [ 9.,  9.,  9.,  9.],
         [10., 10., 10., 10.],
         [11., 11., 11., 11.],
         [12., 12., 12., 12.]]),
 torch.Size([12, 4]))
```

####Squeeze - Remove all '1' dimensions from a tensor

In [117]:
```
### touch.squeeze() - remove all single dimensions from target tensor
```

In [126]:
```
x_reshaped1,x_reshaped1.shape
```

Out[126]:
```
(tensor([[ 5.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9., 10., 11., 12.]]),
 torch.Size([1, 12]))
```

In [124]:
```
x_reshaped1.squeeze()
```

Out[124]:
```
tensor([ 5.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9., 10., 11., 12.])
```

In [125]:
```
x_reshaped1.squeeze().shape
```

Out[125]:
```
torch.Size([12])
```

In [130]:
```
### touch.squeeze() - remove all single dimensions from target tensor
print(f"Previous tensor : {x_reshaped1}")
print(f"Previous shape : {x_reshaped1.shape}")

### Remove extra dimensions from x_reshaped1

x_squeezed = x_reshaped1.squeeze()
print(f"\n New Tensor : {x_squeezed}")
print(f"New shape : {x_squeezed.shape}")
```

```
Previous tensor : tensor([[ 5.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9., 10.,
11., 12.]])
Previous shape : torch.Size([1, 12])

 New Tensor : tensor([ 5.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9., 10., 11.,
12.])
New shape : torch.Size([12])
```

####Unsqueeze - add a '1' dimension to a target tensor

In [137]:
```
print(f"Previous target : {x_squeezed}")
print(f"Previous target shape : {x_squeezed.shape}")

## Add extra dimension with unsqueeze
x_unsqueezed = x_squeezed.unsqueeze(dim=0) ## vertical Squeeze

print(f"New tensor : {x_unsqueezed}")
print(f"New tensor shape : {x_unsqueezed.shape}")
```

```
Previous target : tensor([ 5.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9., 10., 1
1., 12.])
Previous target shape : torch.Size([12])
New tensor : tensor([[ 5.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9., 10., 11.,
12.]])
New tensor shape : torch.Size([1, 12])
```

In [136]:
```
print(f"Previous target : {x_squeezed}")
print(f"Previous target shape : {x_squeezed.shape}")

## Add extra dimension with unsqueeze
x_unsqueezed = x_squeezed.unsqueeze(dim=1)## horizental squeeze

print(f"New tensor : {x_unsqueezed}")
print(f"New tensor shape : {x_unsqueezed.shape}")
```

```
Previous target : tensor([ 5.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9., 10., 1
1., 12.])
Previous target shape : torch.Size([12])
New tensor : tensor([[ 5.],
        [ 2.],
        [ 3.],
        [ 4.],
        [ 5.],
        [ 6.],
        [ 7.],
        [ 8.],
        [ 9.],
        [10.],
        [11.],
        [12.]])
New tensor shape : torch.Size([12, 1])
```

###Permute - Return a view of the input with dimensions permuted(swapped) in a certain way

In [138]:
```
x = torch.randn(2,3,5)
x
```

Out[138]:
```
tensor([[[ 0.4975, -1.7241,  0.4923, -0.9943,  0.8188],
         [-1.0584,  0.5601,  1.5020, -0.1064, -0.0160],
         [-2.3742,  0.6722, -1.5458,  0.1037,  1.2265]],

        [[ 0.7215,  0.2172, -0.8125,  0.2190, -1.5892],
         [-0.9954,  1.2192, -1.0764, -0.4911,  0.3347],
         [ 1.4075,  0.9269, -1.4671,  1.4958,  1.2592]]])
```

In [139]:
```
x.size() ## torch.Size([2, 3, 5]) this is representing 2 set and 3x5 matrix
```

Out[139]:
```
torch.Size([2, 3, 5])
```

```
this is nothing but [2, 3, 5]===>indx(0,1,2)
i will put (2,0,1)
this means

indx        items
2     --->   5
0     --->   2
1     --->   3
now it is looks like (2,0,1)==>[5, 2, 3] here 5 is set and 2x3 are mat
rix
```

In [141]:
```
### Let's do the permution
torch.permute(x, (2,0,1)) ##[5, 2, 3]
```

Out[141]:
```
tensor([[[ 0.4975, -1.0584, -2.3742],
         [ 0.7215, -0.9954,  1.4075]],

        [[-1.7241,  0.5601,  0.6722],
         [ 0.2172,  1.2192,  0.9269]],

        [[ 0.4923,  1.5020, -1.5458],
         [-0.8125, -1.0764, -1.4671]],

        [[-0.9943, -0.1064,  0.1037],
         [ 0.2190, -0.4911,  1.4958]],

        [[ 0.8188, -0.0160,  1.2265],
         [-1.5892,  0.3347,  1.2592]]])
```

In [142]:
```
torch.permute(x, (2,0,1)).size()
```

Out[142]:
```
torch.Size([5, 2, 3])
```

Permute basically use to manage the shape of image data

In [143]:
```
## torch.permute  - rearrange the dimensions of a target tensor in a specified
img = torch.rand(224, 224, 3) ##(height, width, channels)

img
```

Out[143]:
```
tensor([[[0.5976, 0.0082, 0.5484],
         [0.9645, 0.7396, 0.8253],
         [0.0444, 0.0719, 0.0780],
         ...,
         [0.4683, 0.0629, 0.6393],
         [0.7369, 0.8396, 0.6478],
         [0.2487, 0.0555, 0.2831]],

        [[0.6939, 0.3062, 0.6339],
         [0.2556, 0.1724, 0.8119],
         [0.2208, 0.1632, 0.3471],
         ...,
         [0.9312, 0.4045, 0.0445],
         [0.7536, 0.4527, 0.9397],
         [0.1720, 0.5982, 0.7933]],

        [[0.8273, 0.0746, 0.3493],
         [0.3916, 0.6470, 0.7225],
         [0.5232, 0.9968, 0.1912],
         ...,
         [0.2689, 0.8409, 0.5131],
         [0.1847, 0.5803, 0.6717],
         [0.6734, 0.2995, 0.5430]],

        ...,

        [[0.6579, 0.5637, 0.2499],
         [0.7173, 0.6947, 0.0411],
         [0.2575, 0.1324, 0.7151],
         ...,
         [0.8308, 0.0033, 0.7280],
         [0.2177, 0.0087, 0.7021],
         [0.4292, 0.9535, 0.2757]],

        [[0.9423, 0.1177, 0.6439],
         [0.8228, 0.5833, 0.7829],
         [0.9746, 0.0553, 0.4436],
         ...,
         [0.4733, 0.5872, 0.9277],
         [0.9015, 0.6323, 0.1087],
         [0.0364, 0.5501, 0.7538]],

        [[0.8598, 0.7872, 0.6591],
         [0.1540, 0.6229, 0.0061],
         [0.5590, 0.3046, 0.6638],
         ...,
         [0.2854, 0.0075, 0.0024],
         [0.5671, 0.2504, 0.7014],
         [0.5678, 0.6062, 0.7017]]])
```

In [144]:
```
### But in this out put it is giving me 224 sets and 224x3 matrix so for this i
### Permute the original tensor to rearrange the dim or axis order

img_permuted = img.permute(2,0,1)
img_permuted
```

Out[144]:
```
tensor([[[0.5976, 0.9645, 0.0444,  ..., 0.4683, 0.7369, 0.2487],
         [0.6939, 0.2556, 0.2208,  ..., 0.9312, 0.7536, 0.1720],
         [0.8273, 0.3916, 0.5232,  ..., 0.2689, 0.1847, 0.6734],
         ...,
         [0.6579, 0.7173, 0.2575,  ..., 0.8308, 0.2177, 0.4292],
         [0.9423, 0.8228, 0.9746,  ..., 0.4733, 0.9015, 0.0364],
         [0.8598, 0.1540, 0.5590,  ..., 0.2854, 0.5671, 0.5678]],

        [[0.0082, 0.7396, 0.0719,  ..., 0.0629, 0.8396, 0.0555],
         [0.3062, 0.1724, 0.1632,  ..., 0.4045, 0.4527, 0.5982],
         [0.0746, 0.6470, 0.9968,  ..., 0.8409, 0.5803, 0.2995],
         ...,
         [0.5637, 0.6947, 0.1324,  ..., 0.0033, 0.0087, 0.9535],
         [0.1177, 0.5833, 0.0553,  ..., 0.5872, 0.6323, 0.5501],
         [0.7872, 0.6229, 0.3046,  ..., 0.0075, 0.2504, 0.6062]],

        [[0.5484, 0.8253, 0.0780,  ..., 0.6393, 0.6478, 0.2831],
         [0.6339, 0.8119, 0.3471,  ..., 0.0445, 0.9397, 0.7933],
         [0.3493, 0.7225, 0.1912,  ..., 0.5131, 0.6717, 0.5430],
         ...,
         [0.2499, 0.0411, 0.7151,  ..., 0.7280, 0.7021, 0.2757],
         [0.6439, 0.7829, 0.4436,  ..., 0.9277, 0.1087, 0.7538],
         [0.6591, 0.0061, 0.6638,  ..., 0.0024, 0.7014, 0.7017]]])
```

In [145]:
```
print(f"previous shape of the image : {img.shape}")
print(f"permuted shape of the image : {img_permuted.shape}")
```

```
previous shape of the image : torch.Size([224, 224, 3])
permuted shape of the image : torch.Size([3, 224, 224])
```

#PyTorch Vs NumPy

PyTorch and NumPy are both powerful libraries for numerical and scientific computing in Python, but they have some key differences:

1. **Tensor Computation vs. Array Computation:**

   - **PyTorch**: PyTorch is primarily designed for deep learning and neural network computations. It provides a multi-dimensional array called a "tensor," which is similar to NumPy arrays but with additional features optimized for deep learning, such as GPU acceleration and automatic differentiation (autograd).
   - **NumPy**: NumPy is a fundamental library for numerical computing in Python. It provides multidimensional arrays (ndarrays) and a wide range of mathematical functions for array manipulation and mathematical operations.
2. **Automatic Differentiation:**

   - **PyTorch**: PyTorch includes a powerful automatic differentiation framework called autograd. It allows you to automatically compute gradients of tensors, which is

essential for training neural networks using techniques like backpropagation.
   - **NumPy**: NumPy doesn't have built-in automatic differentiation capabilities. To compute gradients in NumPy, you would need to implement them manually.
3. **GPU Support:**
   - **PyTorch**: PyTorch seamlessly supports GPU acceleration, making it the preferred choice for deep learning tasks that require significant computational power.
   - **NumPy**: NumPy can work with GPUs through libraries like CuPy, but it doesn't provide native GPU support.
4. **Deep Learning Ecosystem:**
   - **PyTorch**: PyTorch has gained popularity in the deep learning community and has a strong ecosystem for developing and training neural networks. It provides high-level neural network libraries like PyTorch Lightning and Transformers.
   - **NumPy**: NumPy is a more general-purpose library for scientific computing and lacks the specialized tools for deep learning tasks that PyTorch offers.
5. **Dynamic vs. Static Computation Graphs:**
   - **PyTorch**: PyTorch uses a dynamic computation graph, which means the graph is constructed on the fly as operations are performed. This flexibility is helpful for tasks that involve dynamic or varying graph structures.
   - **NumPy**: NumPy operates on static computation graphs. The graph is defined upfront, making it less flexible but potentially more efficient for certain numerical tasks.

In summary, while both PyTorch and NumPy are essential libraries for scientific computing in Python, PyTorch is particularly well-suited for deep learning tasks due to its GPU support, automatic differentiation, and dynamic computation graph. NumPy, on the other hand, is a versatile library for general numerical computing tasks.

Pytorch tensor and Numpy

Numpy is a popular scientific Python numerical computing library.

And because of this , Pytorch has functionality to interact with it.

- Data in Numpy, want on Pytorch tensor --> `torch.from_numpy(ndarray)`
- Pytorch tensor --> Numpy --> `torch.Tensor.numpy()`

In [146]:
```
## Numpy array to tensor
import torch
import numpy as np

array = np.arange(1.0,8)
tensor = torch.from_numpy(array)
```

In [147]:
```
print(f"numpy array data :{array}")
print(f"numpy array convert to tensor :{tensor}")
```

```
numpy array data :[1. 2. 3. 4. 5. 6. 7.]
numpy array convert to tensor :tensor([1., 2., 3., 4., 5., 6., 7.], dtype=tor
ch.float64)
```

In [148]:
```
array.dtype
```

Out[148]:
```
dtype('float64')
```

Reproducbility (Trying to take random out of random)

In short how a neural network learns:

```
 Start with random number -> tensor operations -> update random number to try
and make them better representations of the data -> again -> again ->
again...
```

To reduce the randomness in neural network for that Pytorch comes with the concept of **random seed**

In [150]:
```
import torch

## Let's create a two random tensors
random_tensor_A = torch.rand(3,4)
random_tensor_B = torch.rand(3,4)

print(random_tensor_A)
print(random_tensor_B)
print()
print(random_tensor_A == random_tensor_B)
```

```
tensor([[0.6609, 0.1646, 0.6901, 0.4621],
        [0.7351, 0.5098, 0.5930, 0.5717],
        [0.4954, 0.5085, 0.7580, 0.5162]])
tensor([[0.7766, 0.2561, 0.5633, 0.2653],
        [0.5972, 0.5313, 0.7024, 0.8182],
        [0.6165, 0.4132, 0.4401, 0.8239]])

tensor([[False, False, False, False],
        [False, False, False, False],
        [False, False, False, False]])
```

In [152]:
```
## Let's make some random but reproducible tensor

import torch
## let's set the random seed
random_seed = 10
torch.manual_seed(random_seed)
random_tensor_A = torch.rand(3,4)

torch.manual_seed(random_seed)
random_tensor_B = torch.rand(3,4)

print(random_tensor_A)
print(random_tensor_B)
print()
print(random_tensor_A == random_tensor_B)
```

```
tensor([[0.4581, 0.4829, 0.3125, 0.6150],
        [0.2139, 0.4118, 0.6938, 0.9693],
        [0.6178, 0.3304, 0.5479, 0.4440]])
tensor([[0.4581, 0.4829, 0.3125, 0.6150],
        [0.2139, 0.4118, 0.6938, 0.9693],
        [0.6178, 0.3304, 0.5479, 0.4440]])

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

In [ ]: