In [288]:
```
import torch as tr
import numpy as np
```

Tensors

In the context of PyTorch and other deep learning frameworks, tensors are multi-dimensional arrays that represent data. They are fundamental building blocks used for storing and manipulating data in neural networks. Tensors can have different dimensions, such as 0-dimensional (scalar), 1-dimensional (vector), 2-dimensional (matrix), or higher-dimensional arrays.

Here are some key characteristics of tensors:

Data Representation: Tensors can represent various types of data, including numerical values (e.g., integers or floating-point numbers), boolean values (True or False), and even categorical data. For example, a grayscale image can be represented as a 2-dimensional tensor, where each element represents the pixel intensity.

Shape:

The shape of a tensor refers to its dimensions or size along each axis. For example, a 2-dimensional tensor representing a grayscale image with dimensions 28x28 would have a shape of (28, 28). Tensors in PyTorch can have arbitrary shapes, allowing for flexibility in representing complex data structures.

Data Types:

Tensors can have different data types, such as integers (int), floating-point numbers (float), or boolean values (bool). PyTorch supports various data types, including 32-bit and 64-bit floating-point numbers (float32 and float64), as well as 8-bit and 16-bit integers (int8 and int16).

Operations:

Tensors support a wide range of mathematical operations, such as addition, subtraction, multiplication, division, matrix multiplication, and more. These operations can be performed element-wise or across specific dimensions of the tensor.

GPU Acceleration:

Tensors can be moved and manipulated on GPU devices to leverage the computational power of GPUs for faster processing. This is particularly useful for training deep neural networks on large datasets, as it allows for parallel computation across multiple GPU cores.

In [289]:
```
# Floating-Number
t1 = tr.tensor(4.)
# 4. = 4.0
# if wrote just 4 it will be Integer number but in Deep Learning we do
# we get float number that's why I wrote 4. instead of 4.0
# By default it gives 4 digits after decimal point " . "
```

In [290]:
```
t1
```

Out[290]:
```
tensor(4.)
```

In [291]:
```
print("{:.6f}".format(t1)) # just printing 6 digits after floating poin
```

```
4.000000
```

In [292]:
```
t1.dtype
```

Out[292]:
```
torch.float32
```

In [293]:
```
t1.shape # empty list mean just one number
```

Out[293]:
```
torch.Size([])
```

In [294]:
```
t1 = tr.tensor([1])
```

In [295]:
```
t1.shape # only one number in list
```

Out[295]:
```
torch.Size([1])
```

In [296]:
```
# Integer number
t2 = tr.tensor(3)
```

In [297]:
```
t2
```

Out[297]:
```
tensor(3)
```

In [298]:
```
print("{:.0f}".format(t2))
```

```
3
```

In [299]:
```
t2.dtype
```

Out[299]:
```
torch.int64
```

In [300]:
```
# Let's Try to create more complex Tensors
```

In [301]:
```
# Vector of Integer
t3 = tr.tensor([1,2,3,4,5])
```

In [302]:
```
t3
```

Out[302]:
```
tensor([1, 2, 3, 4, 5])
```

In [303]:
```
t3.dtype
```

Out[303]:
```
torch.int64
```

In [304]:
```
t3.shape
```

Out[304]:
```
torch.Size([5])
```

In [305]:
```
# Vector of float
t4 = tr.tensor([1,2.1,3,4,5])
# If we add one float in integers all will be floats by default
```

In [306]:
```
t4
```

Out[306]:
```
tensor([1.0000, 2.1000, 3.0000, 4.0000, 5.0000])
```

In [307]:
```
t4.shape
```

Out[307]:
```
torch.Size([5])
```

In [308]:
```
# Matrix of Integer
t5 = tr.tensor([[1, 2, 3, 4, 5],[6, 7, 8, 9, 10],[11, 12, 13, 14, 15]])
```

In [309]:
```
t5
```

Out[309]:
```
tensor([[ 1,  2,  3,  4,  5],
        [ 6,  7,  8,  9, 10],
        [11, 12, 13, 14, 15]])
```

In [310]:
```
t5.dtype
```

Out[310]:
```
torch.int64
```

In [311]:
```
row , column = t5.shape
```

In [312]:
```
row
```

Out[312]:
```
3
```

In [313]:
```
column
```

Out[313]:
```
5
```

In [314]:
```
# Matrix of Floats
t6 = tr.tensor([[1.1, 2, 3, 4, 5],[6, 7, 8, 9, 10],[11, 12, 13, 14, 15]
```

In [315]:
```
t6
```

Out[315]:
```
tensor([[ 1.1000,  2.0000,  3.0000,  4.0000,  5.0000],
        [ 6.0000,  7.0000,  8.0000,  9.0000, 10.0000],
        [11.0000, 12.0000, 13.0000, 14.0000, 15.0000]])
```

In [347]:
```
# 3D-Array of Integer
t5 = tr.tensor([
    [[1, 2, 3, 4, 5],[6, 7, 8, 9, 10]],
    [[11, 12, 13, 14, 15],[16, 17, 18, 19, 20]]
])
```

In [348]:
```
t5
```

Out[348]:
```
tensor([[[ 1,  2,  3,  4,  5],
         [ 6,  7,  8,  9, 10]],

        [[11, 12, 13, 14, 15],
         [16, 17, 18, 19, 20]]])
```

In [349]:
```
# Tensors can have any number of dimensions and different lengths along
```

In [350]:
```
t5.shape
```

Out[350]:
```
torch.Size([2, 2, 5])
```

In [351]:
```
row, back, column = t5.shape
```

In [352]:
```
row
```

Out[352]:
```
2
```

In [353]:
```
back
```

Out[353]:
```
2
```

In [354]:
```
column
```

Out[354]:
```
5
```

In [355]:
```
# 3D-Array of Integer
t6 = tr.tensor([
    [[1, 2, 3, 4],[ 7, 8, 9, 10, 11]],
    [[11, 12, 14, 15],[17, 18, 19, 20]]
])

# Error because of different lengths
```

```
---------------------------------------------------------------------------
-----
ValueError                                Traceback (most recent call
last)
Cell In[355], line 2
      1 # 3D-Array of Integer
----> 2 t6 = tr.tensor([
      3     [[1, 2, 3, 4],[ 7, 8, 9, 10, 11]],
      4     [[11, 12, 14, 15],[17, 18, 19, 20]]
      5 ])

ValueError: expected sequence of length 4 at dim 2 (got 5)
```

Tensor operations and gradients

I can combine tensors with the usual arithmetic operations. Let's look an example and see whats happening

I've created 3 tensors `x`, `w` and `b`, all numbers. `w` and `b` have an additional parameter `requires_grad` set to `True`

In [356]:
```
# Create tensors. for y = wx + b
x = torch.tensor(5., requires_grad=False)# False mean we are NOT going
w = torch.tensor(6., requires_grad=True) # True mean we are going to us
b = torch.tensor(7., requires_grad=True) # True mean we are going to us
x, w, b
```

Out[356]:
```
(tensor(5.), tensor(6., requires_grad=True), tensor(7., requires_grad=
True))
```

Let's create a new tensor `y` by combining these tensors:

In [357]:
```
# Arithmatic operation
y = w*x+b
```

In [358]:
```
y
```

Out[358]:
```
tensor(37., grad_fn=<AddBackward0>)
```

As expected, `y` is a tensor with the value `5 * 6 + 7 = 37`. What makes PyTorch special is that we can automatically compute the derivative of `y` w.r.t. the tensors that have `requires_grad` set to `True` i.e. w and b. To compute the derivatives, we can call the `.backward` method on our result `y`.

In [359]:
```
# Compute Derivatives
y.backward()
```

The derivates of `y` w.r.t the input tensors are stored in the `.grad` property of the respective tensors.

In [360]:
```
# Display gradients
print('dy/dx:', x.grad) # because we didn't save as know we make it Fal
print('dy/dw:', w.grad)
print('dy/db:', b.grad)
```

```
dy/dx: None
dy/dw: tensor(5.)
dy/db: tensor(1.)
```

As expected, `dy/dw` has the same value as `x` i.e. `5.`, and `dy/db` has the value `1.`. Note that `x.grad` is `None`, because `x` doesn't have `requires_grad` set to `True`.

The "grad" in `w.grad` stands for gradient, which is another term for derivative, used mainly when dealing with matrices.

Interoperability with Numpy

Numpy (http://www.numpy.org/) is a popular open source library used for mathematical and scientific computing in Python. It enables efficient operations on large multi-dimensional arrays, and has a large ecosystem of supporting libraries:

- Matplotlib (https://matplotlib.org/) for plotting and visualization
- OpenCV (https://opencv.org/) for image and video processing
- Pandas (https://pandas.pydata.org/) for file I/O and data analysis

Instead of reinventing the wheel, PyTorch interoperates really well with Numpy to leverage its

Here's how we create an array in Numpy:

In [361]:
```
import numpy as np
```

In [369]:
```
x = np.array([[1,2,3],[4,5,6]])
```

In [370]:
```
x
```

Out[370]:
```
array([[1, 2, 3],
       [4, 5, 6]])
```

In [372]:
```
x.dtype
```

Out[372]:
```
dtype('int64')
```

We can convert a Numpy array to s PyTorch using `torch.from_numpy`

In [374]:
```
# Convert the numpy array to a tensor.
y = tr.from_numpy(x)
```

In [375]:
```
y
```

Out[375]:
```
tensor([[1, 2, 3],
        [4, 5, 6]])
```

In [376]:
```
y.dtype
```

Out[376]:
```
torch.int64
```

In [377]:
```
y.shape
```

Out[377]:
```
torch.Size([2, 3])
```

Now we verify that Numpy Array and Torch Tensor have same Data Types Numpy and PyTorch have one->to->one data type corespondance

In [382]:
```
array_data_type, tensor_data_type = x.dtype, y.dtype
```

In [383]:
```
array_data_type
```

Out[383]:
```
dtype('int64')
```

In [384]:
```
tensor_data_type
```

Out[384]:
```
torch.int64
```

In [386]:
```
# From Tensor to Numpy Array
z = y.numpy()
```

In [387]:
```
z
```

Out[387]:
```
array([[1, 2, 3],
       [4, 5, 6]])
```

In [388]:
```
z.dtype
```

Out[388]:
```
dtype('int64')
```

Interoperability between PyTorch and Numpy is essential because most datasets we'll work with will likely be red and preprocessed as Numpy Arrays. Why we need a Library like PyTorch at all since Numpy already provides data structures and utilities for working with multi-dimensional numeric data. There are two main reasons:

Automatic Differentiation:
One of the key features of PyTorch is its automatic differentiation capability through the autograd module. This feature allows PyTorch to automatically compute gradients of the loss function with respect to model parameters, facilitating gradient-based optimization methods like gradient descent. While NumPy provides array operations, it doesn't have built-in support for automatic differentiation, which is essential for training neural networks efficiently.

GPU Acceleration:
PyTorch seamlessly integrates with CUDA, NVIDIA's parallel computing platform, enabling GPU acceleration for numerical computations. This allows PyTorch to leverage the computational power of GPUs for training deep neural networks, resulting in significant speedups compared to CPU-only implementations. While NumPy can be used with GPU-accelerated libraries like CuPy, PyTorch provides a more streamlined approach to GPU programming, especially for deep learning tasks.

In [ ]:
```

```