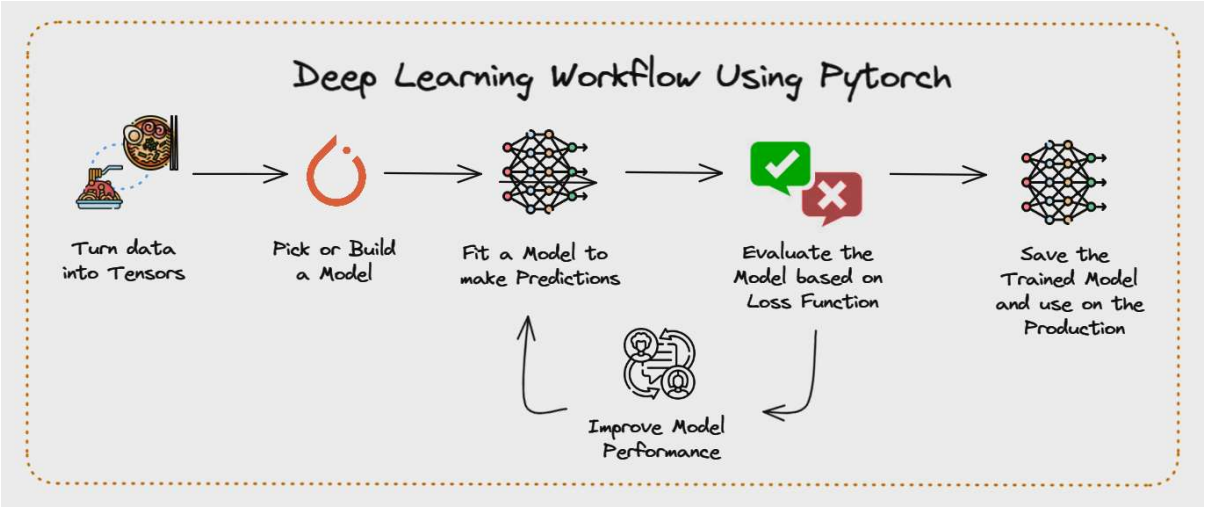
DL-Cheat-Codes (/github/nikitaprasad21/DL-Cheat-Codes/tree/main)
/ Py-Torch (/github/nikitaprasad21/DL-Cheat-Codes/tree/main/Py-Torch)

Creating and Manipulating Tensors using PyTorch

The game of Machine Learning has essentially two parts:

1. Turn your data (table, texts, images, videos, audios, whatever it is or combination of all), into a **numbers representation**, usually called **Tensors**.
2. Pick (Pretained Model for Transfer Learning) or build a model to learn the representation to make **predictions, classify, cluster, or gain insights from your data**.



But what if you don't have data?

Well, that's where we're at now.

No data.

But we can create some.

Creating Synthetic Data with PyTorch

PyTorch, a powerful and flexible deep learning framework, makes it straightforward to handle tensors, build and train models, and generate data.

Let's dive in!

What are Tensors?

Tensors are multi-dimensional arrays used in the field of machine learning and scientific computing, a fundamental data structure.

How to Create Tensors in Python?

First, we'll import 'torch'.

In [23]:
```
import torch
```

Let's see a few basic tensor manipulations.

First, just a few of the ways to create tensors:

1. Using a List

In [ ]:
```
# Creating a 2D tensor (matrix)

torch_list = list(range(5))
print(f"List : {torch_list}")

tensor_1d = torch.tensor(torch_list)
print(f"1D Tensor (Vector): {tensor_1d}")
```

```
List : [0, 1, 2, 3, 4]
1D Tensor (Vector): tensor([0, 1, 2, 3, 4])
```

In [ ]:
```
# Creating a 2D tensor (matrix)

tensor_2d = torch.tensor([[1, 2, 3],
                          [4, 5, 6]])

# Print the tensors
print("2D Tensor (Matrix):")
print(tensor_2d)
print("Shape:", tensor_2d.shape)
```

```
2D Tensor (Matrix):
tensor([[1, 2, 3],
        [4, 5, 6]])
Shape: torch.Size([2, 3])
```

In [ ]:
```
# Creating a 3D tensor

tensor_3d = torch.tensor([[[1, 2, 3],
                           [4, 5, 6]],
                          [[7, 8, 9],
                           [10, 11, 12]]])

print("3D Tensor:")
print(tensor_3d)
print("Shape:", tensor_3d.shape)
```

```
3D Tensor:
tensor([[[ 1,  2,  3],
         [ 4,  5,  6]],

        [[ 7,  8,  9],
         [10, 11, 12]]])
Shape: torch.Size([2, 2, 3])
```

In [ ]:
```
# Creating a 4D

tensor_4d = torch.tensor([[[[1, 2],
                            [3, 4]],
                           [[5, 6],
                            [7, 8]]],
                          [[[9, 10],
                            [11, 12]],
                           [[13, 14],
                            [15, 16]]]])

# Print the tensors
print("4D Tensor:")
print(tensor_4d)
print("Shape:", tensor_4d.shape)
```

```
4D Tensor:
tensor([[[[ 1,  2],
          [ 3,  4]],

         [[ 5,  6],
          [ 7,  8]]],


        [[[ 9, 10],
          [11, 12]],

         [[13, 14],
          [15, 16]]]])
Shape: torch.Size([2, 2, 2, 2])
```

Datatype Using torch

In [ ]:
```
print("Datatype tensor_1d:", tensor_1d.dtype)
print("Datatype tensor_2d:", tensor_2d.dtype)
print("Datatype tensor_3d:", tensor_3d.dtype)
print("Datatype tensor_4d:", tensor_4d.dtype)
```

```
Datatype: torch.int64
Datatype: torch.int64
Datatype: torch.int64
Datatype: torch.int64
```

2. Using a Numpy Array

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

In [ ]:
```
np_array = np.array(list(range(5)))
print(f"Numpy array: {np_array}  || Datatype: {type(np_array)}")

# Convert NumPy Array to PyTorch Tensor

torch_tensor = torch.tensor(np_array)
print(f"Torch Tensor: {torch_tensor} || datatype : {type(torch_tensor)}")
```

```
Numpy array: [0 1 2 3 4]  || Datatype: <class 'numpy.ndarray'>
Torch Tensor: tensor([0, 1, 2, 3, 4]) || datatype : <class 'torch.Tensor'>
```

Or

In [ ]:
```
torch_tensor1 = torch.from_numpy(np_array)
print(f"Torch Tensor: {torch_tensor1} || datatype : {type(torch_tensor)}")
```

```
Torch Tensor: tensor([0, 1, 2, 3, 4]) || datatype : <class 'torch.Tensor'>
```

In [ ]:
```
# Convert tensor to numpy array

numpy_array = torch_tensor.numpy()
print(f"Numpy array: {numpy_array} || datatype : {type(numpy_array)}")
```

```
Numpy array: [0 1 2 3 4] || datatype : <class 'numpy.ndarray'>
```

3. **Using a Tuple**

In [ ]:
```
# Homogenous Datatype

tup = tuple([(1,2),(2,4),(3,6)])
print("Datatype:", type(tup))
print(torch.tensor(tup))
```

```
Datatype: <class 'tuple'>
tensor([[1, 2],
        [2, 4],
        [3, 6]])
```

In [ ]:
```
# Heterogenous Datatype

tup = tuple([("a",2),("b",4),("c",6)])
print(type(tup))
print(torch.tensor(tup))
```

```
<class 'tuple'>
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-19-cf0c32bb24d5> in <cell line: 5>()
      3 tup = tuple([("a",2),("b",4),("c",6)])
      4 print(type(tup))
----> 5 print(torch.tensor(tup))

ValueError: too many dimensions 'str'
```

4. **Using a HashMap**

In [ ]:
```
hashmap = {1:1,2:2,3:3}
print("Datatype",type(hashmap))
print(torch.tensor(hashmap))
```

```
Datatype <class 'dict'>
---------------------------------------------------------------------------
RuntimeError                              Traceback (most recent call last)
<ipython-input-17-727ec46efb90> in <cell line: 3>()
      1 hashmap = {1:1,2:2,3:3}
      2 print("Datatype",type(hashmap))
----> 3 print(torch.tensor(hashmap))

RuntimeError: Could not infer dtype of dict
```

5. **Using a HashSet**

In [ ]:
```
hashset = set([1,22,3,5,6,4,8])
print(type(hashset))
print(torch.tensor(hashset))
```

```
<class 'set'>
---------------------------------------------------------------------------
RuntimeError                              Traceback (most recent call last)
<ipython-input-18-7ccd3f5e91ce> in <cell line: 3>()
      1 hashset = set([1,22,3,5,6,4,8])
      2 print(type(hashset))
----> 3 print(torch.tensor(hashset))

RuntimeError: Could not infer dtype of set
```

zeros() method

This method returns a tensor where all elements are zeros, of specified size (shape).

The size can be given as a tuple or a list or neither.

In [ ]:
```
zero_tensor1 = torch.zeros(3,6)
print(zero_tensor1)
print(zero_tensor1.dtype)
```

```
tensor([[0., 0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0., 0.]])
torch.float32
```

Wondering

What if you pass an empty tuple or an empty list?

Let's code and check.

In [24]:
```
zero_tensor2 = torch.zeros([])
print(zero_tensor2)
print(zero_tensor2.dtype)
```

```
tensor(0.)
torch.float32
```

It will give a tensor of size (dimension) 0, having 0 as its only element, whose data type is float.

ones() method

Similar to zeros(), ones() returns a tensor where all elements are 1, of specified size (shape).

The size can be given as a tuple or a list or neither.

In [25]:
```
ones_tensor1 = torch.ones((5, 3), dtype=torch.int16)
print(ones_tensor1)
print(ones_tensor1.dtype)
```

```
tensor([[1, 1, 1],
        [1, 1, 1],
        [1, 1, 1],
        [1, 1, 1],
        [1, 1, 1]], dtype=torch.int16)
torch.int16
```

Like zeroa(), passing an empty tuple or list gives a tensor of 0 dimension, having 1 as the sole element, whose data type is float.

In [27]:
```
ones_tensor2 = torch.ones([])
print(ones_tensor2)
print(ones_tensor2.dtype)
```

```
tensor(1.)
torch.float32
```

What if you want all the elements of a tensor to be equal to some value but not only 0 and 1? Maybe 0.9?

full() method

It will return a tensor of a shape given by the size argument, with all its elements equal to the fill_value placeholder.

In [29]:
```
full_tensor = torch.full([4,7], fill_value = 2)
print(full_tensor)
print(full_tensor.dtype)
```

```
tensor([[2, 2, 2, 2, 2, 2, 2],
        [2, 2, 2, 2, 2, 2, 2],
        [2, 2, 2, 2, 2, 2, 2],
        [2, 2, 2, 2, 2, 2, 2]])
torch.int64
```

Here, we have created a tensor of shape 4, 7 with the fill_value as 2.

Again, we can pass an empty tuple or list to create a scalar tensor of zero dimension.

In [31]:
```
full_tensor1 = torch.full([], fill_value = 29)
print(full_tensor1)
print(full_tensor1.dtype)
```

```
tensor(29)
torch.int64
```

arange() method

It returns 1-D tensor, with elements from start (inclusive) to end (exclusive) with a common difference step, also called **Arithmetic Progression**.

Note: The default value for start is 0 while that for step is 1. While choosing start, end, and step, we need to ensure that start and end are consistent with the step sign.

In [35]:
```
arrange_tensor = torch.arange(2, 30, 2)
print(arrange_tensor)
print(arrange_tensor.dtype)
```

```
tensor([ 2,  4,  6,  8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28])
torch.int64
```

Here, we created a tensor which starts from 2 and goes until 30 with a step (common difference) of 2.

linspace() method

This method also returns a 1-D dimensional tensor, with elements from start (inclusive) to end (inclusive).

But, unlike arange() method, here, the steps aren't the common difference. Instead, they represent the number of elements to be in the tensor.

In [38]:
```
linspace_tensor = torch.linspace(2, 30, 6)
print(linspace_tensor)
print(linspace_tensor.dtype)
```

```
tensor([ 2.0000,  7.6000, 13.2000, 18.8000, 24.4000, 30.0000])
torch.float32
```

PyTorch automatically decides the common difference based on the steps given.

rand() method

This method returns a tensor filled with random numbers from a uniform distribution on the interval 0 (inclusive) to 1 (exclusive).

The shape is given by the size argument. The size argument can be given as a tuple or list or neither.

Also it's common to initialize learning weights randomly, often with a specific seed for reproducibility of results:

In [39]:
```
torch.manual_seed(29)
r1 = torch.rand(2, 2)
print('A random tensor:')
print(r1)

torch.manual_seed(42)
r2 = torch.rand(2, 2)
print('\nA different random tensor:')
print(r2) # new values

torch.manual_seed(29)
r3 = torch.rand(2, 2)
print('\nShould match r1:')
print(r3) # repeats values of r1 because of re-seed
```

```
A random tensor:
tensor([[0.1226, 0.9355],
        [0.9359, 0.7038]])

A different random tensor:
tensor([[0.8823, 0.9150],
        [0.3829, 0.9593]])

Should match r1:
tensor([[0.1226, 0.9355],
        [0.9359, 0.7038]])
```

randint() method

It will return a tensor filled with random integers generated uniformly between low (inclusive) and high (exclusive).

The shape is given by the size argument. The default value for low is 0.

In [40]:
```
r_int = torch.randint(2, 29, (9,7))
print('A random tensor:')
print(r_int)
```

```
A random tensor:
tensor([[10, 28, 16, 26, 23,  9, 19],
        [ 6,  8,  4, 25, 26, 16,  6],
        [27, 26, 27, 18, 15, 20, 26],
        [14, 25,  7, 27,  7, 20, 27],
        [ 9,  7, 27, 13, 14, 10, 22],
        [26, 20, 21, 24, 19,  9, 21],
        [ 7, 11, 12,  5,  5, 19, 27],
        [16, 12, 15, 24,  2,  4, 18],
        [15, 21, 16, 22, 22,  3,  5]])
```

When only one int argument is passed, low gets the value 0, by default, and high gets the passed value.

eye() method

This method returns a 2-D tensor with ones on the diagonal and zeros elsewhere. The number of rows is given by n and columns is given by m.

In [46]:
```
eye_tensor1 = torch.eye(9,7)
print('Tensor:')
print(eye_tensor1)
```

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

The default value for m is the value of n.

When only n is passed, it creates a tensor in the form of an identity matrix.

In [47]:
```
eye_tensor2 = torch.eye(7)
print('Tensor:')
print(eye_tensor2)
```

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

An identity matrix has its diagonal elements as 1 and all others as 0.

complex() method

This method returns a complex tensor with its real part equal to real and its imaginary part equal to imag. Both real and imag are tensors.

In [48]:
```
real1 = torch.rand(2,2)
print(real1)

imag1 = torch.rand(2,2)
print(imag1)

complex_tensor1 = torch.complex(real1, imag1)
complex_tensor1
```

```
tensor([[0.0465, 0.3384],
        [0.2243, 0.9131]])
tensor([[0.6176, 0.6807],
        [0.5859, 0.3605]])
```

Out[48]:
```
tensor([[0.0465+0.6176j, 0.3384+0.6807j],
        [0.2243+0.5859j, 0.9131+0.3605j]])
```

Note: The data type of both the real and imag tensors should be either float or double.

Also, the size of both tensors, real and imag, should be the same, since the corresponding elements of the two matrices form a complex number.

Accessing Tensors elements

In [49]:
```
# Create a tensor

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

In [50]:
```
# Basic indexing: Access a single element
print("Basic indexing - Single element:", tensor[1, 2])
```

```
Basic indexing - Single element: tensor(6)
```

In [51]:
```
# Slicing: Access a sub-tensor

print("Slicing - Sub-tensor:")
print(tensor[1:, :])
```

```
Slicing - Sub-tensor:
tensor([[4, 5, 6],
        [7, 8, 9]])
```

In [52]:
```
# Fancy indexing: Access specific elements using a list of indices

indices = torch.tensor([0, 2])
print("Fancy indexing - Specific elements:")
print(tensor[:, indices])
```

```
Fancy indexing - Specific elements:
tensor([[1, 3],
        [4, 6],
        [7, 9]])
```

In [53]:
```
# Masked indexing: Access elements based on a boolean mask

mask = tensor > 5
print("Masked indexing - Elements satisfying condition:")
print(tensor[mask])
```

```
Masked indexing - Elements satisfying condition:
tensor([6, 7, 8, 9])
```

In [58]:
```
# reshape() method

tensor_1 = torch.tensor(list(range(27)))

tensor_1.reshape(-1,3)
```

Out[58]:
```
tensor([[ 0,  1,  2],
        [ 3,  4,  5],
        [ 6,  7,  8],
        [ 9, 10, 11],
        [12, 13, 14],
        [15, 16, 17],
        [18, 19, 20],
        [21, 22, 23],
        [24, 25, 26]])
```

Mathematical operations:

PyTorch tensors perform arithmetic operations intuitively. Tensors of similar shapes may be added, multiplied, etc.

In [59]:
```
ones = torch.ones(2, 3)
print(ones)

twos = torch.ones(2, 3) * 2 # every element is multiplied by 2
print(twos)

threes = ones + twos       # additon allowed because shapes are similar
print(threes)              # tensors are added element-wise
print(threes.shape)        # output has the same dimensions as input tensors
```

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

It also support Broadcasting similar to numpy arrays.

In [60]:
```
# Supports Broadcasting

a = torch.tensor([[1, 2, 3],
                  [4, 5, 6]])
print(f"a1 shape :{a.shape}")

b = torch.tensor([[10],
                  [20]])
print(f"b1 shape :{b.shape}")

# Perform element-wise addition (broadcasting)
c = a + b

print(c)
```

```
a1 shape :torch.Size([2, 3])
b1 shape :torch.Size([2, 1])
tensor([[11, 12, 13],
        [24, 25, 26]])
```

In [61]:
```
# Set the random seed for reproducibility
torch.manual_seed(2)

# Generate a random matrix 'r' with values between -1 and 1
r = torch.rand(2, 2) - 0.5 * 2
print('A random matrix, r:')
print(r)
```

```
A random matrix, r:
tensor([[-0.3853, -0.6190],
        [-0.3629, -0.5255]])
```

In [62]:
```
# Common mathematical operations are supported:

print('Absolute value of r:')
print(torch.abs(r))
```

```
Absolute value of r:
tensor([[0.3853, 0.6190],
        [0.3629, 0.5255]])
```

In [64]:
```
# Trigonometric functions:
print('Inverse sine of r:')
print(torch.asin(r))
```

```
Inverse sine of r:
tensor([[-0.3955, -0.6675],
        [-0.3714, -0.5533]])
```

In [65]:
```
# Statistical and aggregate operations:
print('Average and standard deviation of r:')
print(torch.std_mean(r))
print('\nMaximum value of r:')
print(torch.max(r))
```

```
Average and standard deviation of r:
(tensor(0.1210), tensor(-0.4732))

Maximum value of r:
tensor(-0.3629)
```

In [66]:
```
# Linear algebra operations like determinant and singular value decomposition
print('Determinant of r:')
print(torch.det(r))
print('\nSingular value decomposition of r:')
print(torch.svd(r))
```

```
Determinant of r:
tensor(-0.0221)

Singular value decomposition of r:
torch.return_types.svd(
U=tensor([[-0.7523, -0.6588],
        [-0.6588,  0.7523]]),
S=tensor([0.9690, 0.0228]),
V=tensor([[ 0.5459, -0.8379],
        [ 0.8379,  0.5459]]))
```

PyTorch offers a robust framework for managing tensors.

By mastering these tensor operations and manipulations, you can effectively leverage PyTorch to create synthetic data, build and train models, and perform various machine learning tasks, even when starting with no initial data.

But definitely **Data** is truly necessary for Machine Learning Models as per your Problem Statement.

In [ ]: