# Deep Learning basics

<add more details>

# PyTorch Fundamentals

## **What is PyTorch?**

[PyTorch](https://colab.research.google.com/corgiredirector?site=https%3A%2F%2Fpytorch.org%2F&link_redirector=1) is an open source machine learning and deep learning framework.

## **What can PyTorch be used for?**

PyTorch allows you to manipulate and process data and write machine learning algorithms using Python code.

## **Who uses PyTorch?**

Many of the worlds largest technology companies such as [Meta (Facebook)](https://colab.research.google.com/corgiredirector?site=https%3A%2F%2Fai.facebook.com%2Fblog%2Fpytorch-builds-the-future-of-ai-and-machine-learning-at-facebook%2F&link_redirector=1), Tesla and Microsoft as well as artificial intelligence research companies such as [OpenAI use PyTorch](https://colab.research.google.com/corgiredirector?site=https%3A%2F%2Fopenai.com%2Fblog%2Fopenai-pytorch%2F&link_redirector=1) to power research and bring machine learning to their products.

PyTorch is also used in other industries such as agriculture to [power computer vision on tractors](https://colab.research.google.com/corgiredirector?site=https%3A%2F%2Fmedium.com%2Fpytorch%2Fai-for-ag-production-machine-learning-for-agriculture-e8cfdb9849a1&link_redirector=1).

### **Importing PyTorch**

import torch

torch.\_\_version\_\_

# Tensor

<describe more about vectors as well before going for tensors>

Tensors are the fundamental building block of machine learning.

Their job is to represent data in a numerical way.

For example, you could represent an image as a tensor with shape [3, 224, 224] which would mean [colour\_channels, height, width], as in the image has 3 colour channels (red, green, blue), a height of 224 pixels and a width of 224 pixels.

**Link -->** <https://pytorch.org/docs/stable/torch.html>

1. **Scaler ( only magitude)**
   1. Ex: --> Scaler = torch.tensor(7)
   2. Scaler.ndim ( 0 dimension)
   3. Scaler.item() ( 7)
2. **Vector ( magnitude and direction)**
   1. Vector = torch.Tensor([7,7])
   2. Vector.ndim ( 1 dimension) ( no. Of pair Square brackets)
   3. Vector.shape --> torch.Size([2])
3. **Matrix**
   1. Matrix = torch.tensor ([[7,8],  
       [1,1]])
   2. MATRIX.ndim -> 2
   3. Fetching elements in matrix --> MATRIX[1] --> [7,8]
   4. Shape of the matrix -> torch.Size([2,2])
4. **Tensor**
   1. TENSOR = torch.tensor([[  
      [1,2,3],  
      [4,5,6],  
      [7,8,9]  
       ]  
         
       ])
   2. 3 dimension
   3. Shape --> 1,3,3

scalar 
tensor 
What is it? 
a single number 
a number with direction 
(e.g. wind speed with 
direction) but can also 
have many other numbers 
a 2-dimensional array of 
numbers 
an n-dimensional array of 
numbers 
Number of dimensions 
can be any number, a O- 
dimension tensor is a 
scalar, a I-dimension 
tensor is a vector 
Lower or upper 
(usually/example) 
Lower ( a ) 
Lower ( y ) 
Upper ( Q ) 
Upper ( x) 

# Random Tensors

**Link -->** <https://pytorch.org/docs/stable/generated/torch.rand.html>

1. Why random tensors?
   1. Important cos many neural networks learn is that they start with tensor full of random numbers and then adjust those random numbers to better represent the data.
   2. ' start with random numbers --> look at data --> update random numbers --> look at data' --> update random numbers
2. Create random tensor of size/shape (3,4)
   1. Random\_tensor = Torch.rand(3,4)

(tensor([[0.6541, 0.4807, 0.2162, 0.6168], [0.4428, 0.6608, 0.6194, 0.8620], [0.2795, 0.6055, 0.4958, 0.5483]]), torch.float32)

The flexibility of torch.rand() is that we can adjust the size to be whatever we want.

For example, say you wanted a random tensor in the common image shape of [224, 224, 3] ([height, width, color\_channels]).

# Zeros and ones

Sometimes you'll just want to fill tensors with zeros or ones.

This happens a lot with masking (like masking some of the values in one tensor with zeros to let a model know not to learn them).

Again, the `size` parameter comes into play.

# Create a tensor of all zeros

zeros = torch.zeros(size=(3, 4))

zeros, zeros.dtype

We can do the same to create a tensor of all ones except using [`torch.ones()`]

# Create a tensor of all zeros

zeros = torch.zeros(size=(3, 4))

zeros, zeros.dtype

(tensor([[0., 0., 0., 0.],

[0., 0., 0., 0.],

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

torch.float32)

# Create a tensor of all ones

ones = torch.ones(size=(3, 4))

ones, ones.dtype

(tensor([[1., 1., 1., 1.],

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

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

torch.float32)

# Creating a range and tensors like

Sometimes you might want a range of numbers, such as 1 to 10 or 0 to 100.

You can use torch.arange(start, end, step) to do so.

Where:

* start = start of range (e.g. 0)
* end = end of range (e.g. 10)
* step = how many steps in between each value (e.g. 1)

ex:

# Create a range of values 0 to 10

zero\_to\_ten = torch.arange(start=0, end=10, step=1)

Sometimes you might want one tensor of a certain type with the same shape as another tensor.

For example, a tensor of all zeros with the same shape as a previous tensor.

To do so you can use [torch.zeros\_like(input)](https://colab.research.google.com/corgiredirector?site=https%3A%2F%2Fpytorch.org%2Fdocs%2Fstable%2Fgenerated%2Ftorch.zeros_like.html&link_redirector=1) or [torch.ones\_like(input)](https://colab.research.google.com/corgiredirector?site=https%3A%2F%2Fpytorch.org%2Fdocs%2F1.9.1%2Fgenerated%2Ftorch.ones_like.html&link_redirector=1) which return a tensor filled with zeros or ones in the same shape as the input respectively.

# Can also create a tensor of zeros similar to another tensor

ten\_zeros = torch.zeros\_like(input=zero\_to\_ten) # will have same shape

ten\_zeros

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

# Tensor datatypes

Link 🡪 <https://pytorch.org/docs/stable/tensors.html#data-types>

The most common type (and generally the default) is torch.float32 or torch.float.

This is referred to as "32-bit floating point". ( 4 byte)

**Note:** An integer is a flat round number like 7 whereas a float has a decimal 7.0.

The reason for all of these is to do with **precision in computing**.

Precision is the amount of detail used to describe a number.

The higher the precision value (8, 16, 32), the more detail and hence data used to express a number.

# Default datatype for tensors is float32

float\_32\_tensor = torch.tensor([3.0, 6.0, 9.0],

                               dtype=None, # defaults to None, which is torch.float32 or whatever datatype is passed

                               device=None, # defaults to None, which uses the default tensor type

                               requires\_grad=False) # if True, operations performed on the tensor are recorded

float\_32\_tensor.shape, float\_32\_tensor.dtype, float\_32\_tensor.device

(torch.Size([3]), torch.float32, device(type='cpu'))

Aside from shape issues (tensor shapes don't match up), two of the other most common issues you'll come across in PyTorch are datatype and device issues.

For example, one of tensors is torch.float32 and the other is torch.float16 (PyTorch often likes tensors to be the same format).

Or one of your tensors is on the CPU and the other is on the GPU (PyTorch likes calculations between tensors to be on the same device).

# Getting information from tensors

three of the most common attributes

* shape - what shape is the tensor? (some operations require specific shape rules)
* dtype - what datatype are the elements within the tensor stored in?
* device - what device is the tensor stored on? (usually GPU or CPU)

code

# Create a tensor

some\_tensor = torch.rand(3, 4)

# Find out details about it

print(some\_tensor)

print(f"Shape of tensor: {some\_tensor.shape}")

print(f"Datatype of tensor: {some\_tensor.dtype}")

print(f"Device tensor is stored on: {some\_tensor.device}") # will default to CPU

# Manipulating tensors (tensor operations)

In deep learning, data (images, text, video, audio, protein structures, etc) gets represented as tensors.

A model learns by investigating those tensors and performing a series of operations (could be 1,000,000s+) on tensors to create a representation of the patterns in the input data.

Below operations are the building blocks of neural networks

* Addition
* Substraction
* Multiplication (element-wise)
* Division
* Matrix multiplication

# Basic operations

## Addition , subtraction and multiplication

# Create a tensor of values and add a number to it

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

tensor + 10

tensor([11, 12, 13])

# Multiply it by 10

tensor \* 10

tensor([10, 20, 30])

PyTorch also has a bunch of built-in functions like [torch.mul()](https://colab.research.google.com/corgiredirector?site=https%3A%2F%2Fpytorch.org%2Fdocs%2Fstable%2Fgenerated%2Ftorch.mul.html%23torch.mul&link_redirector=1) (short for multiplication) and [torch.add()](https://colab.research.google.com/corgiredirector?site=https%3A%2F%2Fpytorch.org%2Fdocs%2Fstable%2Fgenerated%2Ftorch.add.html&link_redirector=1) to perform basic operations.

# Element-wise multiplication (each element multiplies its equivalent, index 0->0, 1->1, 2->2)

print(tensor, "\*", tensor)

print("Equals:", tensor \* tensor)

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

Equals: tensor([1, 4, 9])