# 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" \t "_blank) 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" \t "_blank) 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" \t "_blank) (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" \t "_blank) 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])

# Matrix multiplication

The main two rules for matrix multiplication to remember are:

1. The **inner dimensions** must match:
   * (3, 2) @ (3, 2) won't work
   * (2, 3) @ (3, 2) will work
   * (3, 2) @ (2, 3) will work
2. The resulting matrix has the shape of the **outer dimensions**:
   * (2, 3) @ (3, 2) -> (2, 2)
   * (3, 2) @ (2, 3) -> (3, 3)
3. The difference between element-wise multiplication and matrix multiplication is the addition of values.
4. For our tensor variable with values [1, 2, 3]:

| **Operation** | **Calculation** | **Code** |
| --- | --- | --- |
| **Element-wise multiplication** | [1\*1, 2\*2, 3\*3] = [1, 4, 9] | tensor \* tensor |
| **Matrix multiplication** | [1\*1 + 2\*2 + 3\*3] = [14] | tensor.matmul(tensor) |

# Element-wise matrix multiplication

tensor \* tensor

torch.matmul()

# Matrix multiplication

torch.matmul(tensor, tensor)

%%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

%%time

torch.matmul(tensor, tensor)

# One of the most common errors in deep learning (shape errors)

# Shapes need to be in the right way

tensor\_A = torch.tensor([[1, 2],

                         [3, 4],

                         [5, 6]], dtype=torch.float32)

tensor\_B = torch.tensor([[7, 10],

                         [8, 11],

                         [9, 12]], dtype=torch.float32)

torch.matmul(tensor\_A, tensor\_B) # (this will error)

We can make matrix multiplication work between tensor\_A and tensor\_B by making their inner dimensions match.

One of the ways to do this is with a **transpose** (switch the dimensions of a given tensor).

You can perform transposes in PyTorch using either:

* torch.transpose(input, dim0, dim1) - where input is the desired tensor to transpose and dim0 and dim1 are the dimensions to be swapped.
* tensor.T - where tensor is the desired tensor to transpose.
* # View tensor\_A and tensor\_B
* print(tensor\_A)
* print(tensor\_B)
* # View tensor\_A and tensor\_B.T
* print(tensor\_A)
* print(tensor\_B.T)

# 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([[ 27., 30., 33.],

[ 61., 68., 75.],

[ 95., 106., 117.]])

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

You can also use [torch.mm()](https://colab.research.google.com/corgiredirector?site=https%3A%2F%2Fpytorch.org%2Fdocs%2Fstable%2Fgenerated%2Ftorch.mm.html&link_redirector=1) which is a short for torch.matmul().

# torch.mm is a shortcut for matmul

torch.mm(tensor\_A, tensor\_B.T)

# Tensor Aggregation(min , max , mean , sum)

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

print(f"Minimum: {x.min()}")

print(f"Maximum: {x.max()}")

# print(f"Mean: {x.mean()}") # this will error

print(f"Mean: {x.type(torch.float32).mean()}") # won't work without float datatype

print(f"Sum: {x.sum()}")

You can also do the same as above with torch methods.

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

## Positional min/max

You can also find the index of a tensor where the max or minimum occurs with [torch.argmax()](https://colab.research.google.com/corgiredirector?site=https%3A%2F%2Fpytorch.org%2Fdocs%2Fstable%2Fgenerated%2Ftorch.argmax.html&link_redirector=1" \t "_blank) and [torch.argmin()](https://colab.research.google.com/corgiredirector?site=https%3A%2F%2Fpytorch.org%2Fdocs%2Fstable%2Fgenerated%2Ftorch.argmin.html&link_redirector=1" \t "_blank) respectively.

# Create a tensor

tensor = torch.arange(10, 100, 10)

print(f"Tensor: {tensor}")

# Returns index of max and min values

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

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

## Change tensor datatype

You can change the datatypes of tensors using [torch.Tensor.type(dtype=None)](https://colab.research.google.com/corgiredirector?site=https%3A%2F%2Fpytorch.org%2Fdocs%2Fstable%2Fgenerated%2Ftorch.Tensor.type.html&link_redirector=1" \t "_blank) where the dtype parameter is the datatype you'd like to use.

# Create a tensor and check its datatype

tensor = torch.arange(10., 100., 10.)

tensor.dtype

# Create a float16 tensor

tensor\_float16 = tensor.type(torch.float16)

tensor\_float16

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

# Reshaping, stacking, squeezing and unsqueezing

Why do any of these?

Because deep learning models (neural networks) are all about manipulating tensors in some way. And because of the rules of matrix multiplication, if you've got shape mismatches, you'll run into errors. These methods help you make the right elements of your tensors are mixing with the right elements of other tensors.

Reshaping 🡪 change the shape of the tensor to a defined shape

View 🡪 return a view of an input tensor ( just like in DB in some way) **keeps the same memory as original tensor**

Stacking 🡪 combine multiple tensors ( vertical stacking on top of each other)

Squeeze 🡪 removed all `1`dimention from tensor

unSqueeze 🡪 add a `1`dimention from tensor

permute 🡪 returns view of the input with dimension permute ( changed , swapped) in a certain way

| **Method** | **One-line description** |
| --- | --- |
| [torch.reshape(input, shape)](https://colab.research.google.com/corgiredirector?site=https%3A%2F%2Fpytorch.org%2Fdocs%2Fstable%2Fgenerated%2Ftorch.reshape.html%23torch.reshape&link_redirector=1) | Reshapes input to shape (if compatible), can also use torch.Tensor.reshape(). |
| [torch.Tensor.view(shape)](https://colab.research.google.com/corgiredirector?site=https%3A%2F%2Fpytorch.org%2Fdocs%2Fstable%2Fgenerated%2Ftorch.Tensor.view.html&link_redirector=1) | Returns a view of the original tensor in a different shape but shares the same data as the original tensor. |
| [torch.stack(tensors, dim=0)](https://colab.research.google.com/corgiredirector?site=https%3A%2F%2Fpytorch.org%2Fdocs%2F1.9.1%2Fgenerated%2Ftorch.stack.html&link_redirector=1) | Concatenates a sequence of tensors along a new dimension (dim), all tensors must be same size. |
| [torch.squeeze(input)](https://colab.research.google.com/corgiredirector?site=https%3A%2F%2Fpytorch.org%2Fdocs%2Fstable%2Fgenerated%2Ftorch.squeeze.html&link_redirector=1) | Squeezes input to remove all the dimenions with value 1. |
| [torch.unsqueeze(input, dim)](https://colab.research.google.com/corgiredirector?site=https%3A%2F%2Fpytorch.org%2Fdocs%2F1.9.1%2Fgenerated%2Ftorch.unsqueeze.html&link_redirector=1) | Returns input with a dimension value of 1 added at dim. |
| [torch.permute(input, dims)](https://colab.research.google.com/corgiredirector?site=https%3A%2F%2Fpytorch.org%2Fdocs%2Fstable%2Fgenerated%2Ftorch.permute.html&link_redirector=1) | Returns a *view* of the original input with its dimensions permuted (rearranged) to dims. |

## Change the shape

# Create a tensor

import torch

x = torch.arange(1., 8.)

x, x.shape

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

Now let's add an extra dimension with torch.reshape().

# Add an extra dimension

x\_reshaped = x.reshape(1, 7)

x\_reshaped, x\_reshaped.shape

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

Will below work?

1. **x\_reshaped = x.reshape(2, 7)# increasing the dimension by one keeping the number of elements same**
2. **x\_reshaped = x.reshape(7,1)# increasing the dimension by 7 keeping the number of elements as one in each dimension**

## change the view

# Change view (keeps same data as original but changes view)

# See more: https://stackoverflow.com/a/54507446/7900723

z = x.view(1, 7)

z, z.shape

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

Remember though, changing the view of a tensor with torch.view() really only creates a new view of the same tensor.

So changing the view changes the original tensor too.

# Changing z changes x

z[:, 0] = 5

z, x

## Stack tensors

# Stack tensors on top of each other

x\_stacked = torch.stack([x, x, x, x], dim=0) # try changing **dim to dim=1 and see what happens**

x\_stacked

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

[5., 2., 3., 4., 5., 6., 7.],

[5., 2., 3., 4., 5., 6., 7.],

[5., 2., 3., 4., 5., 6., 7.]])

**Change dimension to 1,2 and try**

**Try vstack 🡪 usually dimension as 0**

**Try hstack 🡪 usually dimension as 1**

## Squeezing Tensors

Removes all single dimension from a tensor

print(f"Previous tensor: {x\_reshaped}")

print(f"Previous shape: {x\_reshaped.shape}")

# Remove extra dimension from 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.]])

Previous shape: torch.Size([1, 7])

New tensor: tensor([5., 2., 3., 4., 5., 6., 7.])

New shape: torch.Size([7])

Unsqueeze Tensors

To add a dimension to a tensor

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.])

Previous shape: torch.Size([7])

New tensor: tensor([[5., 2., 3., 4., 5., 6., 7.]])

New shape: torch.Size([1, 7])

Permute Tensors

Rearrange tensor dimension( only change the view)

# Create tensor with specific shape

x\_original = torch.rand(size=(224, 224, 3))# image tensor , height ,width and color

# Permute the original tensor to rearrange the axis order

x\_permuted = x\_original.permute(2, 0, 1) # shifts axis 0->1, 1->2, 2->0

print(f"Previous shape: {x\_original.shape}")

print(f"New shape: {x\_permuted.shape}")

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

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

**Change value in the view and see whether it impacts the original tensor**