
## What is PyTorch ?

**PyTorch is an open-source deep learning framework developed by Facebook's AI Research lab. It provides tools for building and training neural networks, offering flexibility and speed. PyTorch is popular for its dynamic computational graph, which allows for more intuitive model development and debugging. It is widely used in both research and production environments for tasks like image and speech recognition, natural language processing, and more.**

# Pytorch Fundamentals Part 1

https://www.learnpytorch.io/00_pytorch_fundamentals/

In [ ]:
```
import torch
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
print(torch.__version__)
```

```
2.3.1+cu121
```

# Introduction to Tensors

## Creating tensors

In [ ]:
```
# scaler
scaler = torch.tensor(8)
scaler
```

Out[ ]:
```
tensor(8)
```

In [ ]:
```
type(scaler)
```

Out[ ]:
```
torch.Tensor
```

In [ ]:
```
# dimension
scaler.ndim
```

Out[ ]:
```
0
```

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

Out[ ]:
```
8
```

# Vector

In [ ]:
```
# vector
vector = torch.tensor([1,2])
vector
```

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

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

Out[ ]:
```
1
```

In [ ]:
```
vector
```

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

In [ ]:
```
vector.shape
```

Out[ ]:
```
torch.Size([2])
```

In [ ]:
```
# MATRIX
MATRIX = torch.tensor([[7,8],
                       [9,10]])
MATRIX
```

Out[ ]:
```
tensor([[ 7,  8],
        [ 9, 10]])
```

In [ ]:
```
MATRIX.ndim
```

Out[ ]:
```
2
```

In [ ]:
```
MATRIX[1]
```

Out[ ]:
```
tensor([ 9, 10])
```

In [ ]:
```
MATRIX.shape
```

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

In [ ]:
```
# TENSOR
TENSOR =  torch.tensor([[[1,2,3],
                         [3,6,9],
                         [2,4,5]]])
TENSOR
```

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

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

Out[ ]:
```
3
```

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

Out[ ]:
```
torch.Size([1, 3, 3])
```

In [ ]:
```
TENSOR =  torch.tensor([[[1,2,3],
                         [3,6,9],
                         [2,4,5]],
                        [[1,2,3],
                         [3,6,9],
                         [2,4,5]]])
TENSOR
```

Out[ ]:
```
tensor([[[1, 2, 3],
         [3, 6, 9],
         [2, 4, 5]],

        [[1, 2, 3],
         [3, 6, 9],
         [2, 4, 5]]])
```

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

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

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

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

## Random tensors

Why random tensors?

Random tensors are important because they way many neural networks learn is that they start with tensors full of random numbers and then adjust those random numbers to better represent the data.

`Start with random numbers -> look at the data -> update random numbers -> look at data -> update random numbers`

Torch random tensors - https://pytorch.org/docs/stable/generated/torch.rand.html

In [ ]:
```
# create a random tensor of size(3,4)
random_tensor = torch.rand(3,4)
```

In [ ]:
```
random_tensor
```

Out[ ]:
```
tensor([[0.6071, 0.4498, 0.6507, 0.8511],
        [0.5302, 0.5842, 0.3872, 0.2635],
        [0.8236, 0.5723, 0.8080, 0.0020]])
```

In [ ]:
```
random_tensor.ndim
```

Out[ ]:
```
2
```

In [ ]:
```
random_tensor = torch.rand(2,3,4)
random_tensor
```

Out[ ]:
```
tensor([[[0.2516, 0.4484, 0.8563, 0.8429],
         [0.4279, 0.0298, 0.8348, 0.7139],
         [0.2731, 0.0311, 0.3201, 0.3922]],

        [[0.6499, 0.1625, 0.7921, 0.9410],
         [0.0483, 0.7576, 0.3115, 0.9539],
         [0.0554, 0.0861, 0.6052, 0.0196]]])
```

In [ ]:
```
random_tensor.ndim
```

Out[ ]:
```
3
```

In [ ]:
```
# create a random tensor with similar shape to an image tensor
random_image_size_tensor = torch.rand(size=(224,224,3)) # height, width, colour channels (R, G, B)
random_image_size_tensor.shape, random_image_size_tensor.ndim
```

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

## Zeros and Ones

In [ ]:
```
# create a tensor of all zeros
zeros = torch.zeros(size = (3,4))
zeros
```

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

In [ ]:
```
zeros * random_tensor
```

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

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

In [ ]:
```
# create a tensor of all ones
ones = torch.ones(size = (3,4))
ones
```

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

In [ ]:
```
ones.dtype
```

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

In [ ]:
```
random_tensor.dtype
```

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

## Creating a range of tensors and tensors-like

In [ ]:
```
# Use torch.range()
torch.range(0,10)
```

```
<ipython-input-90-a70e3231c961>:2: UserWarning: torch.range is deprecated and will be removed in a future relea
se because its behavior is inconsistent with Python's range builtin. Instead, use torch.arange, which produces
values in [start, end).
  torch.range(0,10)
```

Out[ ]:
```
tensor([ 0.,  1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9., 10.])
```

In [ ]:
```
torch.arange(0,10)
```

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

In [ ]:
```
one_to_ten = torch.arange(1,11)
one_to_ten
```

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

In [ ]:
```
torch.arange(start = 0, end = 1000, step =  77)
```

Out[ ]:
```
tensor([  0,  77, 154, 231, 308, 385, 462, 539, 616, 693, 770, 847, 924])
```

In [ ]:
```
# Create tensors like -> same shape like one_to_ten and value is 0
ten_zeros = torch.zeros_like(input = one_to_ten)
ten_zeros
```

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

## Tensor Datatype

**Note:** Tensor datatypes is one of the 3 big errors you'll run into with PyTorch & deep Learning:

1. Tensors not right datatype
2. Tensors not right shape

3. Tensors not on the right device

In [ ]:
```
# Float 32 tensor
float_32_tensor = torch.tensor([3.0, 6.0, 9.0],
                               dtype = None)
float_32_tensor
```

Out[ ]:
```
tensor([3., 6., 9.])
```

In [ ]:
```
float_32_tensor.dtype
```

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

In [ ]:
```
float_32_tensor = torch.tensor([3.0, 6.0, 9.0],
                               dtype = torch.float16)
float_32_tensor
```

Out[ ]:
```
tensor([3., 6., 9.], dtype=torch.float16)
```

In [ ]:
```
float_32_tensor.dtype
```

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

In [ ]:
```
float_32_tensor = torch.tensor([3.0, 6.0, 9.0],
                               dtype = None, # what datatype is the tensor (e.g. )
                               device = None, # using cpu "device = 'cpu", if using gpu "device = cuda"
                               requires_grad = False) # Whether or not to track gradients with this tensors ope
float_32_tensor
```

Out[ ]:
```
tensor([3., 6., 9.])
```

In [ ]:
```
float_16_tensor = float_32_tensor.type(torch.float16)
float_16_tensor
```

Out[ ]:
```
tensor([3., 6., 9.], dtype=torch.float16)
```

In [ ]:
```
float_16_tensor * float_32_tensor
```

Out[ ]:
```
tensor([ 9., 36., 81.])
```

In [ ]:
```
int_32_tensors = torch.tensor([3,6,9],dtype = torch.int32)
int_32_tensors
```

Out[ ]:
```
tensor([3, 6, 9], dtype=torch.int32)
```

In [ ]:
```
float_32_tensor * int_32_tensors
```

Out[ ]:
```
tensor([ 9., 36., 81.])
```

In [ ]:
```
int_long_tensors = torch.tensor([3,6,9],dtype = torch.long)
int_long_tensors
```

Out[ ]:
```
tensor([3, 6, 9])
```

In [ ]:
```
int_long_tensors * float_16_tensor
```

Out[ ]:
```
tensor([ 9., 36., 81.], dtype=torch.float16)
```

## Getting information from tensors (tensors attribute)

1. Tensors not right datatype - to do get datatype from a tensor, can use `tensor.dtype`.
2. Tensors not right shape - to get shape from a tensor, can use `tensor.shape`.
3. Tensors not on the right device - to get device from a tensor, can use `tensor.device`.

In [ ]:
```
# create a tensor
some_tensor = torch.rand(3,4)
some_tensor
```

Out[ ]:
```
tensor([[0.6757, 0.4375, 0.9422, 0.9297],
        [0.6587, 0.9022, 0.7940, 0.7310],
        [0.9959, 0.4484, 0.4022, 0.8468]])
```

In [ ]:
```
# find out details about some tensor
print(some_tensor)
print(f"Datatype of tensor : {some_tensor.dtype}")
print(f"Shape of tensor : {some_tensor.shape}")
print(f"Device of tensor : {some_tensor.device}")
```

```
tensor([[0.6757, 0.4375, 0.9422, 0.9297],
        [0.6587, 0.9022, 0.7940, 0.7310],
        [0.9959, 0.4484, 0.4022, 0.8468]])
Datatype of tensor : torch.float32
Shape of tensor : torch.Size([3, 4])
Device of tensor : cpu
```

## Manipulating Tensors (tensor operations)

Tensor operations incluse:

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

In [ ]:
```
# Create a tensor
tensor =  torch.tensor([1,2,3])
tensor + 10
```

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

In [ ]:
```
# Multiply tensor by 10
tensor * 10
```

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

In [ ]:
```
# subtract 10
tensor - 10
```

Out[ ]:
```
tensor([-9, -8, -7])
```

In [ ]:
```
# Try out PyTorch in-built functions
torch.mul(tensor,10)
```

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

## Matrix multiplication

http://matrixmultiplication.xyz/

Two main ways of performing multiplications in neural networks and deep learning:

1. Element-wise multiplication
2. Matrix multiplication(dot product)

There are two main rules that performing matrix multiplication needs to satisfy:

1. The **inner dimensuibs** must match:

- `(3, 2) @ (3, 2)` won't work
- `(2, 3) @ (3, 2)` will work
- `(3, 2) @ (2, 3)` will work

1. The resulting matrix has the shape of the **outer dimensions**:

- `(2, 3) @ (3, 2) -> (2, 2)`
- `(3, 2) @ (2, 3) -> (3, 3)`

In [ ]:
```
# Element Wise multiplication
print(tensor, "*", tensor)
print(f"Equals: {tensor * tensor}")
```

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

In [ ]:
```
# Matrix multiplication
torch.matmul(tensor , tensor)
```

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

In [ ]:
```
tensor @ tensor
```

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

In [ ]:
```
# matrix multiplication by hand
1*1 + 2*2 + 3*3
```

Out[ ]:
```
14
```

In [ ]:
```
%%time
value = 0
for i in range(len(tensor)):
  value += tensor[i] * tensor[i]
print(value)
```

```
tensor(14)
CPU times: user 1.47 ms, sys: 81 µs, total: 1.55 ms
Wall time: 1.42 ms
```

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

```
CPU times: user 609 µs, sys: 0 ns, total: 609 µs
Wall time: 573 µs
```

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

## One of the most common errors in deep learning: shape errors

In [ ]:
```
#  shapes for matrix multiplication
tensor_A = torch.tensor([[1, 2],
                         [3, 4],
                         [5, 6]])

tensor_B = torch.tensor([[7, 8],
                         [8, 11],
                         [9, 12]])

# torch.mm(tensor_A, tensor_B) # torch.mm is the same as torch.matmul (its an alias for writng less code)
torch.matmul(tensor_A, tensor_B)
```

```
---------------------------------------------------------------------------
RuntimeError                              Traceback (most recent call last)
<ipython-input-118-9cf80e10b890> in <cell line: 11>()
      9
     10 # torch.mm(tensor_A, tensor_B) # torch.mm is the same as torch.matmul (its an alias for writng less cod
e)
---> 11 torch.matmul(tensor_A, tensor_B)

RuntimeError: mat1 and mat2 shapes cannot be multiplied (3x2 and 3x2)
```

In [ ]:
```
tensor_A.shape, tensor_B.shape
```

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

To fix our tensor shape issues, we can manipulate the shape of one of our tensors using a **transpose**

A **transpose** switches the axes or dimensions of given tensor.

In [ ]:
```
tensor_B
```

Out[ ]:
```
tensor([[ 7,  8],
        [ 8, 11],
        [ 9, 12]])
```

In [ ]:
```
tensor_B.T
```

Out[ ]:
```
tensor([[ 7,  8,  9],
        [ 8, 11, 12]])
```

In [ ]:
```
# The matrix multiplication operation works when tensor_B is transposed

print(f"original shape: tensor_A = {tensor_A.shape}, tensor_B = {tensor_B.shape}")
print(f"New shapes: tensor_A {tensor_A.shape}, (same shape as above), tensor_B.T= {tensor_B.T.shape}")
print(f"Multiplying: {tensor_A.shape} @ {tensor_B.T.shape} <- inner dimensions must match")
print("Output:\n")
output = torch.matmul(tensor_A, tensor_B.T)
print(output)
print(f"\nOutput shape: {output.shape}")
```

```
original shape: tensor_A = torch.Size([3, 2]), tensor_B = torch.Size([3, 2])
New shapes: tensor_A torch.Size([3, 2]), (same shape as above), tensor_B.T= torch.Size([2, 3])
Multiplying: torch.Size([3, 2]) @ torch.Size([2, 3]) <- inner dimensions must match
Output:

tensor([[ 23,  30,  33],
        [ 53,  68,  75],
        [ 83, 106, 117]])

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

## Finding the min, max, mean, sum, etc (tensor aggregation)

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

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

In [ ]:
```
# Find the min
torch.min(x), x.min()
```

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

In [ ]:
```
# Find the max
torch.max(x), x.max()
```

Out[ ]:
```
(tensor(90), tensor(90))
```

In [ ]:
```
# Find the mean - note: the torch.mean() function requires a tensor of float32f datatype to work
torch.mean(x.type(torch.float32)), x.type(torch.float32)
```

Out[ ]:
```
(tensor(45.), tensor([ 0., 10., 20., 30., 40., 50., 60., 70., 80., 90.]))
```

In [ ]:
```
# find the sum
torch.sum(x), x.sum()
```

Out[ ]:
```
(tensor(450), tensor(450))
```

## Finding the positional min and max

In [ ]:
```
x
```

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

In [ ]:
```
''' find the positiioin in tensor that has the minimum value
with argmin() -> returns index position of target tensor where the minimum value occurs '''
x.argmin()
```

Out[ ]:
```
tensor(0)
```

In [ ]:
```
x[0]
```

Out[ ]:
```
tensor(0)
```

In [ ]:
```
# Find the position in tensor that has the maximum value with argmax()
x.argmax()
```

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

In [ ]:
```
x[9]
```

Out[ ]:
```
tensor(90)
```

## Reshaping , stacking, squeezing and unsqueezing tensors

- Reshaping - reshaping an input tensor to a defined shape
- View - Return a view of an input tensor of certain shape but keep the same memory as the original tensor
- Stacking - combine multiple tensors on top of eath other (vstack) or side by side (hstack)
    - Stack documentation - https://pytorch.org/docs/stable/generated/torch.stack.html
    - vstack documentation - https://pytorch.org/docs/stable/generated/torch.vstack.html
    - hstack documentation - https://pytorch.org/docs/stable/generated/torch.hstack.html
- Squeeze - remove all `1` dimensions from a tensor
    - https://pytorch.org/docs/stable/generated/torch.squeeze.html
- Unsqueeze - add a `1` dimension to a target tensor
    - https://pytorch.org/docs/stable/generated/torch.unsqueeze.html
- Permute - Return a view of the input with dimensions permuted (swapped) in a certain way
    - https://pytorch.org/docs/stable/generated/torch.permute.html

In [ ]:
```
# Let's create a tensor
import torch
x = torch.arange(1., 10.)
x, x.shape
```

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

In [ ]:
```
# add an extra dimension
x_reshaped = x.reshape(1,7) #1*7 =  7 but element is 10. So, not possible to reshape
x_reshaped, x_reshaped.shape
```

```
---------------------------------------------------------------------------
RuntimeError                              Traceback (most recent call last)
<ipython-input-134-8b576cbba5d6> in <cell line: 2>()
      1 # add an extra dimension
----> 2 x_reshaped = x.reshape(1,7) #1*7 =  7 but element is 10. So, not possible to reshape
      3 x_reshaped, x_reshaped.shape

RuntimeError: shape '[1, 7]' is invalid for input of size 9
```

In [ ]:
```
x_reshaped = x.reshape(9,1)
x_reshaped, x_reshaped.shape
```

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

In [ ]:
```
y = torch.arange(1., 15.)
y, y.shape
```

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

In [ ]:
```
y_reshaped = y.reshape(2,7) # 2*7 = 14 and elements also 14. So, which means this is possible to reshape
y_reshaped, y_reshaped.shape
```

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

In [ ]:
```
x_reshaped = x.reshape(1,9)
x_reshaped, x_reshaped.shape
```

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

In [ ]:
```
# change the view
z = x.view(1,9)
z, z.shape
```

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

In [ ]:
```
# Changing z changes x (because a view of a tensor shares the same memory as the original input)
z[:, 0] = 5
z, x
```

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

In [ ]:
```
# stack tensors on top
x_stacked = torch.stack([x,x,x,x], dim = 0)
x_stacked
```

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

In [ ]:
```
x_stacked = torch.stack([x,x,x,x], dim = 1)
x_stacked
```

Out[ ]:
```
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.]])
```

In [ ]:
```
# vstack
a = torch.tensor([1, 2, 3])
b = torch.tensor([4, 5, 6])
torch.vstack((a,b))
```

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

In [ ]:
```
a = torch.tensor([[1],[2],[3]])
b = torch.tensor([[4],[5],[6]])
```

```
torch.vstack((a,b))
```

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

In [ ]:
```
# hstack
a = torch.tensor([1, 2, 3])
b = torch.tensor([4, 5, 6])
torch.hstack((a,b))
```

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

In [ ]:
```
a = torch.tensor([[1],[2],[3]])
b = torch.tensor([[4],[5],[6]])
torch.hstack((a,b))
```

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

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

# Remove extra dimensions 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., 8., 9.]])
Previous shape:torch.Size([1, 9])

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

In [ ]:
```
# torch.unsqueeze() - add a single dimension to a target tensor at a specific dim (dimension)
print(f"Previous target:{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 target:tensor([5., 2., 3., 4., 5., 6., 7., 8., 9.])
Previous shape:torch.Size([9])

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

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

# Add an extra dimension with unsqueeze
x_unsqueezed = x_squeezed.unsqueeze(dim = 1)
print(f"\nNew tensor:{x_unsqueezed}")
print(f"New shape:{x_unsqueezed.shape}")
```

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

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

In [ ]:
```
# torch.permute() - rearrange the dimensions of a target tensor in a specified order
x = torch.randn(2, 3, 5)
x.size()
torch.permute(x, (2, 0, 1)).size()
```

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

In [ ]:
```
x_original = torch.rand(size = (224,224,3)) # [height, width, colour_channels]

# Permute the original tensor to rearrange the axis (or dim) 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])
```

In [ ]:
```
torch.rand(size = (3,3,5))
```

Out[ ]:
```
tensor([[[8.2115e-02, 4.2507e-01, 3.1158e-01, 5.9749e-02, 6.4100e-01],
         [3.9593e-01, 4.4083e-01, 9.8811e-01, 3.3514e-01, 7.2242e-02],
         [5.3166e-01, 7.2198e-01, 2.0165e-01, 6.7099e-01, 6.7512e-01]],

        [[6.5476e-01, 3.0361e-01, 8.6002e-01, 2.4069e-01, 7.0018e-01],
         [1.1613e-01, 5.1558e-05, 7.9979e-02, 6.8263e-01, 5.8552e-01],
         [2.4899e-02, 2.8236e-01, 3.6671e-01, 9.5290e-01, 9.8172e-01]],

        [[8.9017e-01, 4.7566e-02, 4.5251e-01, 9.7642e-01, 1.4787e-02],
         [4.7771e-01, 6.1474e-01, 4.6526e-01, 7.8078e-01, 7.5515e-01],
         [9.3192e-01, 5.5320e-01, 7.5630e-02, 7.8897e-01, 5.3769e-01]]])
```

## Indexing (selecting data from tensors)

Indexing with PyTorch is similar to indexing with NumPy.

In [ ]:
```
# Create a tensor
import torch
x = torch.arange(1,10).reshape(1,3,3)
x, x.shape
```

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

In [ ]:
```
# Let's index on our new tensor
x[0]
```

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

In [ ]:
```
# let's index on the middle bracket (dim = 1)
x[0][0]
```

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

In [ ]:
```
# Let's index on the most inner bracket (last dimension)
x[0][0][0]
```

Out[ ]:
```
tensor(1)
```

In [ ]:
```
x[0][2][2]
```

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

In [ ]:
```
# You can also use ":" to select "all" of a target dimension
x[:,0]
```

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

In [ ]:
```
# Get all values of 0th and 1st dimensions but only index 1 of 2nd dimension
x[:,:,1]
```

Out[ ]:
```
tensor([[2, 5, 8]])
```

In [ ]:
```
# Get all values of the 0th dimension but only the 1 index value of 1st and 2nd dimension
x[:,1,1]
```

Out[ ]:
```
tensor([5])
```

In [ ]:
```
# Get index 0 of 0th and 1st dimension and all values of 2nd dimension
x[0,0,:]
```

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

In [ ]:
```
# Index on x to return 9
print(x[0][2][2])

# Index on x to return 3, 6, 9
print(x[:,:,2])
```

```
tensor(9)
tensor([[3, 6, 9]])
```

## PyTorch tensors & NumPy

NumPy is a popular scietific Python numerical computing library.

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

- Data in NumPy, want in PyTorch tensor -> torch.form_numpy(ndarray)
- PyTorch tensor -> NumPy -> torch.Tensor.numpy()
  - https://pytorch.org/tutorials/beginner/examples_tensor/polynomial_numpy.html

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

array = np.arange(1.0, 8.0)
tensor = torch.from_numpy(array) # warning: when converting from numpy -> pytorch, pytorch reflects numpy's def
array, tensor
```

Out[ ]:
```
(array([1., 2., 3., 4., 5., 6., 7.]),
 tensor([1., 2., 3., 4., 5., 6., 7.], dtype=torch.float64))
```

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

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

In [ ]:
```
torch.arange(1.0, 8.0).dtype
```

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

In [ ]:
```
tensor = torch.from_numpy(array).type(torch.float32)
```

In [ ]:
```
tensor.dtype
```

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

In [ ]:
```
# change the value of array, what will this do to  `tensor`?

array = array + 1
array, tensor
```

Out[ ]:
```
(array([2., 3., 4., 5., 6., 7., 8.]), tensor([1., 2., 3., 4., 5., 6., 7.]))
```

In [ ]:
```
# Tensor to NumPy array
tensor =  torch.ones(7)
numpy_tensor = tensor.numpy()
tensor, numpy_tensor
```

Out[ ]:
```
(tensor([1., 1., 1., 1., 1., 1., 1.]),
 array([1., 1., 1., 1., 1., 1., 1.], dtype=float32))
```

In [ ]:
```
tensor.dtype
```

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

In [ ]:
```
# Change the tensor, what happens to `numpy_tensor`?
tensor = tensor + 1
tensor, numpy_tensor
```

Out[ ]:
```
(tensor([2., 2., 2., 2., 2., 2., 2.]),
 array([1., 1., 1., 1., 1., 1., 1.], dtype=float32))
```

## Reproducbility (trying to take random out of random)

In short how a neural network learns:

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

To reduce the randomness in neural networks and PyTorch comes the concept of a **random seed**>

Essentially what the random seed does is *flavour* the randomness.

- https://pytorch.org/docs/stable/notes/randomness.html

In [ ]:
```
import torch

# create 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(random_tensor_A == random_tensor_B)
```

```
tensor([[0.9728, 0.5582, 0.9514, 0.6633],
        [0.3924, 0.2452, 0.7548, 0.8859],
        [0.5792, 0.0677, 0.4573, 0.6661]])
tensor([[0.0984, 0.1097, 0.0100, 0.2526],
        [0.4272, 0.9519, 0.3572, 0.3070],
        [0.1312, 0.5938, 0.1416, 0.7310]])
tensor([[False, False, False, False],
        [False, False, False, False],
        [False, False, False, False]])
```

In [ ]:
```
# Let's make some random but reproducible tensors
import torch

# Set the random seed
RANDOM_SEED = 42
torch.manual_seed(RANDOM_SEED)
random_tensor_C = torch.rand(3, 4)

torch.manual_seed(RANDOM_SEED)
random_tensor_D = torch.rand(3, 4)

print(random_tensor_C)
print(random_tensor_D)
print(random_tensor_C == random_tensor_D)
```

```
tensor([[0.8823, 0.9150, 0.3829, 0.9593],
        [0.3904, 0.6009, 0.2566, 0.7936],
        [0.9408, 0.1332, 0.9346, 0.5936]])
tensor([[0.8823, 0.9150, 0.3829, 0.9593],
        [0.3904, 0.6009, 0.2566, 0.7936],
        [0.9408, 0.1332, 0.9346, 0.5936]])
tensor([[True, True, True, True],
        [True, True, True, True],
        [True, True, True, True]])
```