Deep Learning Course - LAB 1

Intro to PyTorch

PyTorch (PT) is a Python (and C++ and Java) library for Machine Learning (ML) particularly suited for Neural Networks and their applications.

Its great selection of built-in modules, models, functions, CUDA capability, tensor arithmetic support and automatic differentiation functionality make it one of the most used scientific libraries for Deep Learning.

Note: for this series of labs, we advise to install Python >= 3.8

Installing PyTorch

We advise to install PyTorch following the directions given in its home page. Just typing `pip install torch` may not be the correct action as you have to take into account the compatibility with `cuda`. If you have `cuda` installed, you can find your version by typing `nvcc --version` in a terminal (Linux/iOS). We suggest installing PyTorch inside a Conda environment as for the link indicated before.

If you're using Windows, we first suggest first to install Anaconda and then install PyTorch from the `anaconda prompt` software via `conda` (preferably) or `pip`.

If you're using Google Colab, all the libraries needed to follow this lecture should be pre-installed there.

We see now how to operate on Colab.

For Colab users

Google Colab is a handy tool that we suggest you use for this course---especially if your laptop does not support CUDA or has limited hardware capabilities. Anyway, note that **we'll try to avoid GPU code as much as possible**.

Essentially, Colab renders available to you a virtual machine with a limited hardware capability and disk where you can execute your code inside a given time window. You can even ask for a GPU (if you use it too much you'll need to start waiting a lot before it's available though).

Your (maybe) first Colab commands

Colab Jupyter-style notebook interface with a few tweaks.

For instance, you may run (some) bash command from here prepending `!` to your code.

In [1]:
```
!ls
```

```
01-assign.ipynb       03-sgd-training.ipynb scripts
01-intro-to-pt.ipynb  data
02-autograd.ipynb     imgs
```

In [2]:
```
!pwd
```

```
/Users/valeria/Desktop/Corsi/DL/DSSC_DL_2022/labs
```

In [3]:
```
!git clone https://github.com/mnzluca/IntroToAI
```

```
Cloning into 'IntroToAI'...
remote: Enumerating objects: 105, done.
remote: Counting objects: 100% (39/39), done.
remote: Compressing objects: 100% (32/32), done.ts:  25% (8/32)
^Cceiving objects:  24% (26/105), 25.72 MiB | 5.00 MiB/s
```

This makes it very easy to operate your virtual machine without the need for a terminal.

File transfer on Colab

One of the most intricate action in Colab is file transfer. Since your files reside on the virtual machine, there're two main ways to operate file transfer on Colab.

- `files.download()` / `.upload()`

In [ ]:
```
from google.colab import files
files.upload()
```

```
---------------------------------------------------------------------------
ModuleNotFoundError                       Traceback (most recent call last)
/var/folders/yc/j6_6k9gx7997h34vyd30xf_m0000gn/T/ipykernel_8562/1613494533.py in <module>
----> 1 from google.colab import files
      2 files.upload()

ModuleNotFoundError: No module named 'google'
```

In [ ]:
```
files.download("sample_data/README.md")
```

Although it may be much more handy to connect your Google Drive to Colab. Here is a snippet that lets you do this.

In [ ]:
```
from google.colab import drive

folder_mount = '/content/drive' # Your Drive will be mounted on top of this path

drive.mount(folder_mount)
```

For VsCode users

Visual Studio Code works perfectly if you're using conda environments.

Just create your own conda environment, then, on VsCode, hit `Ctrl + Shift + p` and type `Python: select interpreter`. The dropdown menu should list the Python binaries within the envs you have created on your machine.

Moreover, is you're running a Jupyter Notebook, you may select an interpreter the first time you run a cell during the current session.

Dive into PyTorch - connections with NumPy

Numpy is a Python library with support for numerical features and tensor calculus: for e.g.can support calculus of arrays of arrays (tensor), can do single value decomposition of a matrix (linear algebra).

Tensor is an extension of the concept of vector/matrix. A vector is a 1 dim tensor, a matrix is a 2 dim Tensor, etc... so vectors of matrices.

Like NumPy, PyTorch provides its own multidimensional array class, called `Tensor`. `Tensor`s are essentially the equivalent of NumPy `ndarray`s. If we wish to operate a very superficial comparison between `Tensor` and `ndarray`, we can say that:

- `Tensor` draws a lot of methods from NumPy, although it's missing some. The gap though is getting smaller and smaller each new release of PyTorch
- `Tensor` is more object oriented than `ndarray` and solves some inconsistencies within NumPy
- `Tensor` has CUDA support

In [4]:
```
import torch
import numpy as np

# create custom Tensor and ndarray
x = torch.Tensor([[1,5,4],[3,2,1]]) #initialize them with floating point numbers: float32!
# ndarray from list of lists with size 6, shape 2x3, 2 dims.
y = np.array([[1,5,4],[3,2,1]])

def pretty_print(obj, title=None):
    if title is not None:
        print(title)
    print(obj)
    print("\n")

pretty_print(x, "x")
pretty_print(y, "y")
```

```
x
tensor([[1., 5., 4.],
        [3., 2., 1.]])


y
[[1 5 4]
 [3 2 1]]
```

What are the types of these objs?

In [5]:
```
x.dtype, y.dtype
```

Out[5]:
```
(torch.float32, dtype('int64'))
```

You can change the type, halving the precision or changing into int

In [6]:
```
x.half() #float16
```

Out[6]:
```
tensor([[1., 5., 4.],
        [3., 2., 1.]], dtype=torch.float16)
```

In [7]:
```
x.int()
```

Out[7]:
```
tensor([[1, 5, 4],
        [3, 2, 1]], dtype=torch.int32)
```

`torch` already thinks with Machine Learning in mind as the `Tensor` is implicitly converted to `dtype float32`, while NumPy makes no such assumption.

For more info on `Tensor` data types, please check the beginning of this page. We can use `Tensor` methods such as `.half()` or `.bool()` to convert the tensor from `float32` to `float16` or `bool` respectively.

As in NumPy, we can call the `.shape` attribute to get the shape of the structures. Moreover, `Tensor`s have also the `.size()` method which is analogous to `.shape`.

In [8]:
```
x.shape, y.shape, x.size()
# In Numpy shape is a tuple, while in Tensor is an object! You must convert it to a tuple in case
```

Out[8]:
```
(torch.Size([2, 3]), (2, 3), torch.Size([2, 3]))
```

Notice how a `Tensor` shape is **not** a tuple.

We can also create a random `Tensor` analogously to NumPy.

A `2 × 3 × 3` `Tensor` is the same as saying "2 3 × 3 matrices", or a "cubic matrix"

In [9]:
```
x = torch.rand([2, 3, 3]) #pass the shape of the tensor via list/tuple
x # it initialized with random numbers
```

Out[9]:
```
tensor([[[0.2356, 0.0150, 0.7885],
         [0.3446, 0.8987, 0.8362],
         [0.6984, 0.1716, 0.0535]],

        [[0.8088, 0.4677, 0.1808],
         [0.8010, 0.1625, 0.5048],
         [0.0949, 0.3538, 0.9389]]])
```

In [10]:
```
y = np.random.rand(2, 3, 3) #no list/tuple
y
```

Out[10]:
```
array([[[0.7579999 , 0.80733101, 0.15583248],
        [0.6663677 , 0.35246106, 0.1205396 ],
        [0.51757254, 0.08632965, 0.67967427]],

       [[0.32294577, 0.88689132, 0.82016614],
        [0.95889093, 0.05294121, 0.66106224],
        [0.08388725, 0.03165091, 0.79532465]]])
```

We can get the total number of elements in a `Tensor` via the `numel()` method

In [11]:
```
x.numel()
```

Out[11]:
```
18
```

We can get the memory occupied by each element of a `Tensor` via `element_size()`

In [12]:
```
x.element_size() #Bytes occupacy for a singe element
```

Out[12]:
```
4
```

Hence, we can quickly calculate the size of the `Tensor` within the RAM

In [13]:
```
x.numel() * x.element_size()
```

Out[13]:
```
72
```

Slicing a `Tensor`

You can slice a `Tensor` (*i.e.*, extract a substructure of a `Tensor`) as in NumPy using the square brackets:

In [14]:
```
# extract first element (i.e., matrix) of first dimension
pretty_print(x[0], "Slice first element (x[0])") #access the first of the 2 matrcies

# extract a specific element: second matrix, third row, first col
pretty_print(x[1,2,0], "Slice element at (1, 2, 0) (x[1, 2, 0])")

# extract first element of second dimension (":" means all the elements of the given dim)
# le prime righe di tutte le dimensioni
pretty_print(x[:, 0], "Slice first element of second dim (x[:, 0])")

# note that it is equivalent to
pretty_print(x[:, 0, :], "As above (x[:, 0] equivalent to x[:, 0, :])")

# extract range of dimensions (first and second element of third dim)
pretty_print(x[:, :, 0:2], "Slice first and second el of third dim (x[:, :, 0:2])")
# 0:2 means 0,1 (no 2!) so it's like [0,2)

# note that it is equivalent to (i.e., you can also pass list for slicing, as opposed to Py vanilla lists/tuples)
pretty_print(x[:,
```

```
:, (0, 1)], "As above (x[:, :, 0:2] equivalent to x[:, :, (0, 2)])")
# tuple or list for specifying e.g (1,2) : this doesn't work with regular Python, but with Numpy and Pytorch yes!
```

```
Slice first element (x[0])
tensor([[0.2356, 0.0150, 0.7885],
        [0.3446, 0.8987, 0.8362],
        [0.6984, 0.1716, 0.0535]])


Slice element at (1, 2, 0) (x[1, 2, 0])
tensor(0.0949)


Slice first element of second dim (x[:, 0])
tensor([[0.2356, 0.0150, 0.7885],
        [0.8088, 0.4677, 0.1808]])


As above (x[:, 0] equivalent to x[:, 0, :])
tensor([[0.2356, 0.0150, 0.7885],
        [0.8088, 0.4677, 0.1808]])


Slice first and second el of third dim (x[:, :, 0:2])
tensor([[[0.2356, 0.0150],
         [0.3446, 0.8987],
         [0.6984, 0.1716]],

        [[0.8088, 0.4677],
         [0.8010, 0.1625],
         [0.0949, 0.3538]]])


As above (x[:, :, 0:2] equivalent to x[:, :, (0, 2)])
tensor([[[0.2356, 0.0150],
         [0.3446, 0.8987],
         [0.6984, 0.1716]],

        [[0.8088, 0.4677],
         [0.8010, 0.1625],
         [0.0949, 0.3538]]])
```

In Py, you can also slice any list by interval via the "double colon" notation `::` (`from` : `(to - 1)` : `step`). Note that `::3` means "take all elements of the object by step of 3 starting from 0 until the list ends".

In [15]:
```
torch.arange(0, 10)[::] # it works with arrays and list also!
```

Out[15]:
```
tensor([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

In [16]:
```
torch.arange(0, 10)[0:7:3]
```

Out[16]:
```
tensor([0, 3, 6])
```

In [17]:
```
torch.arange(0, 10)[0:7]
```

Out[17]:
```
tensor([0, 1, 2, 3, 4, 5, 6])
```

`Tensor` supports linear algebra

In [18]:
```
z1 = torch.rand([4, 5])
print("z1")
print("shape", z1.shape)
print(z1)

# transposition .T: works only with matrices, vectors
z2 = z1.T
# or z2 = torch.transpose(z1) -> enables you to specify which way to transpose

print("\nz2")
print("shape", z2.shape)
print(z2)
```

```
z1
shape torch.Size([4, 5])
tensor([[0.3409, 0.2072, 0.2189, 0.5536, 0.2270],
        [0.4117, 0.2328, 0.5285, 0.7068, 0.1158],
        [0.7592, 0.3955, 0.4703, 0.2904, 0.9452],
        [0.7758, 0.5915, 0.4794, 0.8648, 0.3969]])

z2
shape torch.Size([5, 4])
tensor([[0.3409, 0.4117, 0.7592, 0.7758],
        [0.2072, 0.2328, 0.3955, 0.5915],
        [0.2189, 0.5285, 0.4703, 0.4794],
        [0.5536, 0.7068, 0.2904, 0.8648],
        [0.2270, 0.1158, 0.9452, 0.3969]])
```

In [19]:
```
# matrix multiplication @ same as Numpy
```

```
pretty_print(z1 @ z2, "Matrix multiplication: with '@'")

# equivalent to
pretty_print(torch.matmul(z1, z2), "Matrix multiplication: with torch.matmul")

# and also
pretty_print(z1.matmul(z2), "Matrix multiplication: with Tensor.matmul")
```

```
Matrix multiplication: with '@'
tensor([[0.5651, 0.7218, 0.8190, 1.0608],
        [0.7218, 1.0159, 0.9679, 1.3676],
        [0.8190, 0.9679, 1.9317, 1.6747],
        [1.0608, 1.3676, 1.6747, 2.0870]])


Matrix multiplication: with torch.matmul
tensor([[0.5651, 0.7218, 0.8190, 1.0608],
        [0.7218, 1.0159, 0.9679, 1.3676],
        [0.8190, 0.9679, 1.9317, 1.6747],
        [1.0608, 1.3676, 1.6747, 2.0870]])


Matrix multiplication: with Tensor.matmul
tensor([[0.5651, 0.7218, 0.8190, 1.0608],
        [0.7218, 1.0159, 0.9679, 1.3676],
        [0.8190, 0.9679, 1.9317, 1.6747],
        [1.0608, 1.3676, 1.6747, 2.0870]])
```

Note that `@` identifies the matrix product.

Don't mistake `@` and `*` as the latter is the Hadamard (element-by-element) product!

Question: is it possible to do scalar product between vectors?

Note: the scalar product is also called inner or dot or projection product.

Like we have seen with matrices, we can proceed as follows for vectors:

In [20]:
```
(5) # this is a scalar for PY
```

Out[20]:
```
5
```

In [21]:
```
[5] # this is a single tuple/list in Py
```

Out[21]:
```
[5]
```

In [22]:
```
# So to make a singleton list, one must:
(5,)
```

Out[22]:
```
(5,)
```

In [23]:
```
vec1 = torch.rand((5,)) #vec1 = torch.rand([5])
vec2 = torch.rand((5,))

pretty_print(vec1, "vec1")
pretty_print(vec2, "vec2")

pretty_print(vec1 @ vec2, "I can use '@' even if the two vectors aren't conformable for matrix moltiplication (i.e. b
# Pytorch doesn't care if they are row vector or column vector: they are all tensor of dim 1!

# Here I get a row vector x a column vector
pretty_print(vec1.unsqueeze(0) @ vec2.unsqueeze(-1), "This is the 'classical' inner product between row and column ve

pretty_print(torch.matmul(vec1, vec2), "Notice that the same result can be achieved via the `matmul` method of torch

pretty_print(torch.dot(vec1, vec2), "...and also via torch.dot") # torch.dot != dot in Numpy (CHIEDI)
```

```
vec1
tensor([0.9408, 0.3934, 0.1220, 0.5033, 0.9368])


vec2
tensor([0.7281, 0.6375, 0.8509, 0.5179, 0.8573])


I can use '@' even if the two vectors aren't conformable for matrix moltiplication (i.e. both colum or row vectors
tensor(2.1033)


This is the 'classical' inner product between row and column vector -- Notice the dim of the output
tensor([[2.1033]])


Notice that the same result can be achieved via the `matmul` method of torch -- as we have seen, it's the inner produ
ct
tensor(2.1033)


...and also via torch.dot
tensor(2.1033)
```

In [24]:
```
print(vec1.shape)
vec1
```

Out[24]:
```
torch.Size([5])
tensor([0.9408, 0.3934, 0.1220, 0.5033, 0.9368])
```

In [25]:
```
# unsqeeze add dimensions to a vector
# with 0 it means that I am adding a dimension to the tensor at the first position!
print(vec1.unsqueeze(0).shape)
vec1.unsqueeze(0)
```

Out[25]:
```
torch.Size([1, 5])
tensor([[0.9408, 0.3934, 0.1220, 0.5033, 0.9368]])
```

In [26]:
```
# unsqeeze -1
# Transform vec1 from a row vector to a column vector: I have adde 1 dim as previous too.
print(vec1.unsqueeze(-1).shape)
vec1.unsqueeze(-1)
```

Out[26]:
```
torch.Size([5, 1])
tensor([[0.9408],
        [0.3934],
        [0.1220],
        [0.5033],
        [0.9368]])
```

In [27]:
```
# dot works only with vectors!
# torch.dot(torch.rand((3,4)), torch.rand(4,3))

# do this instead!
torch.matmul(torch.rand((3,4)), torch.rand(4,3))
```

Out[27]:
```
tensor([[1.0440, 0.6820, 0.9166],
        [1.9238, 1.0181, 1.6744],
        [0.9212, 0.2860, 0.7804]])
```

On to the tutorial, let us see more examples of operations on tensors and conformability between tensors....

In [28]:
```
z1 * z2 # this gives an Exception: * is the element-by-element moltiplication!
```

```
---------------------------------------------------------------------------
RuntimeError                              Traceback (most recent call last)
/Users/valeria/Desktop/Corsi/DL/DSSC_DL_2022/labs/01-intro-to-pt.ipynb Cell 53' in <cell line: 1>()
----> <a href='vscode-notebook-cell:/Users/valeria/Desktop/Corsi/DL/DSSC_DL_2022/labs/01-intro-to-pt.ipynb#ch0000052?
line=0'>1</a> z1 * z2

RuntimeError: The size of tensor a (5) must match the size of tensor b (4) at non-singleton dimension 1
```

In [29]:
```
z1
```

Out[29]:
```
tensor([[0.3409, 0.2072, 0.2189, 0.5536, 0.2270],
        [0.4117, 0.2328, 0.5285, 0.7068, 0.1158],
        [0.7592, 0.3955, 0.4703, 0.2904, 0.9452],
        [0.7758, 0.5915, 0.4794, 0.8648, 0.3969]])
```

In [30]:
```
z1 * z1 # I get the square elements
```

Out[30]:
```
tensor([[0.1162, 0.0429, 0.0479, 0.3065, 0.0515],
        [0.1695, 0.0542, 0.2793, 0.4995, 0.0134],
        [0.5764, 0.1564, 0.2212, 0.0843, 0.8933],
        [0.6019, 0.3499, 0.2298, 0.7479, 0.1575]])
```

Generally, the "regular" arithmetic operators for Python act as element-wise operators in `Tensor`s (as in `ndarrays`)

In [31]:
```
z1 ** 2 # Equivalent to above: *** is raising to the power in Py
```

Out[31]:
```
tensor([[0.1162, 0.0429, 0.0479, 0.3065, 0.0515],
        [0.1695, 0.0542, 0.2793, 0.4995, 0.0134],
        [0.5764, 0.1564, 0.2212, 0.0843, 0.8933],
        [0.6019, 0.3499, 0.2298, 0.7479, 0.1575]])
```

In [32]:
```
z3 = torch.Tensor([[1,2,3,4,7],[0.2,2,4,5,3],[-1,3,-4,2,2],[1,1,1,1,2]])
pretty_print(z1 % z3, "z1 % z3 (remainder of integer division)") # module op
pretty_print(z3 // z1, "z3 // z1 (integer division)") # integer division
z3 /= z1
pretty_print(z3, "in-place tensor division (z3 /= z1)")
```

```
z1 % z3 (remainder of integer division)
tensor([[ 0.3409,  0.2072,  0.2189,  0.5536,  0.2270],
        [ 0.0117,  0.2328,  0.5285,  0.7068,  0.1158],
        [-0.2408,  0.3955, -3.5297,  0.2904,  0.9452],
        [ 0.7758,  0.5915,  0.4794,  0.8648,  0.3969]])


z3 // z1 (integer division)
tensor([[ 2.,  9., 13.,  7., 30.],
        [ 0.,  8.,  7.,  7., 25.],
        [-1.,  7., -8.,  6.,  2.],
        [ 1.,  1.,  2.,  1.,  5.]])


in-place tensor division (z3 /= z1)
tensor([[ 2.9335,  9.6529, 13.7038,  7.2248, 30.8435],
        [ 0.4858,  8.5912,  7.5682,  7.0745, 25.9143],
        [-1.3172,  7.5854, -8.5044,  6.8874,  2.1160],
        [ 1.2890,  1.6906,  2.0861,  1.1563,  5.0388]])
```

```
/var/folders/yc/j6_6k9gx7997h34vyd30xf_m0000gn/T/ipykernel_22170/3080503218.py:3: UserWarning: __floordiv__ is deprec
ated, and its behavior will change in a future version of pytorch. It currently rounds toward 0 (like the 'trunc' fun
ction NOT 'floor'). This results in incorrect rounding for negative values. To keep the current behavior, use torch.d
iv(a, b, rounding_mode='trunc'), or for actual floor division, use torch.div(a, b, rounding_mode='floor').
  pretty_print(z3 // z1, "z3 // z1 (integer division)") # integer division
```

In some PyTorch-based libraries, you will also see things like

In [33]:
```
#z3 * z1
z3.mul(z1) # PyTorch way: quicker usually!
#this is equivalent to z3*z1
```

Out[33]:
```
tensor([[ 1.0000,  2.0000,  3.0000,  4.0000,  7.0000],
        [ 0.2000,  2.0000,  4.0000,  5.0000,  3.0000],
        [-1.0000,  3.0000, -4.0000,  2.0000,  2.0000],
        [ 1.0000,  1.0000,  1.0000,  1.0000,  2.0000]])
```

or, for in-place operations, you will also see

In [34]:
```
z3.mul_(z1)
```

Out[34]:
```
tensor([[ 1.0000,  2.0000,  3.0000,  4.0000,  7.0000],
        [ 0.2000,  2.0000,  4.0000,  5.0000,  3.0000],
        [-1.0000,  3.0000, -4.0000,  2.0000,  2.0000],
        [ 1.0000,  1.0000,  1.0000,  1.0000,  2.0000]])
```

this also applies to other operations (not necessarily arithmetical, also boolean):

In [35]:
```
z3 = torch.Tensor([[1,2,3],[1,1,1]])
```

In [36]:
```
# If I confront to tensors a, b like this:
# a == b
# it compares element-wise b and a
```

In [37]:
```
pretty_print(z3 == z3 ** 2, "In the Pythonic way (z3 == z3 ** 2)...")

pretty_print(z3.eq(z3.pow(2)), "...and in the 'PyTorch' way (z3.eq(z3.pow(2))")
```

```
In the Pythonic way (z3 == z3 ** 2)...
tensor([[ True, False, False],
        [ True,  True,  True]])


...and in the 'PyTorch' way (z3.eq(z3.pow(2))
tensor([[ True, False, False],
        [ True,  True,  True]])
```

PyTorch abides to one of Python unwritten rules: method names ending with a single "_" and not starting with one are meant to indicate in-place operations, which mutate the object they're called from.

As for `ndarrays`, `Tensor`s arithmetic operations support **broadcasting**. Roughly speaking, when two `Tensor`s have different shapes and a binary operator is applied to them, PT will try to find a way to make these objects "compatible" for the operation.

Of course, broadcasting is not always possible, but as a rule of thumb, if some dimensions of a `Tensor` are one and the other dimensions are the same, broadcasting works.

In [38]:
```
small_vector_5 = torch.Tensor([1,2,3,5,2]) # this is treated as a row vector (1 x 5 matrix)
print("small_vector_5:", small_vector_5, "; Shape:", small_vector_5.shape, "\n")

pretty_print(z1 / small_vector_5, "Broadcasting: dividing matrix by row vector")

small_vector_4 = torch.Tensor([4,2,3,1])
small_vector_4 = small_vector_4.unsqueeze(-1) # this operation "transposes" the vector into a column vector (4 x 1 ma
print("small_vector_4:\n", small_vector_4, "\nShape:", small_vector_4.shape, "\n")

pretty_print(z1 / small_vector_4, "Broadcasting: dividing matrix by column vector")
```

```
small_vector_5: tensor([1., 2., 3., 5., 2.]) ; Shape: torch.Size([5])

Broadcasting: dividing matrix by row vector
tensor([[0.3409, 0.1036, 0.0730, 0.1107, 0.1135],
        [0.4117, 0.1164, 0.1762, 0.1414, 0.0579],
        [0.7592, 0.1977, 0.1568, 0.0581, 0.4726],
        [0.7758, 0.2957, 0.1598, 0.1730, 0.1985]])


small_vector_4:
 tensor([[4.],
        [2.],
        [3.],
        [1.]])
Shape: torch.Size([4, 1])

Broadcasting: dividing matrix by column vector
tensor([[0.0852, 0.0518, 0.0547, 0.1384, 0.0567],
        [0.2058, 0.1164, 0.2643, 0.3534, 0.0579],
        [0.2531, 0.1318, 0.1568, 0.0968, 0.3151],
        [0.7758, 0.5915, 0.4794, 0.8648, 0.3969]])
```

In [39]:
```
torch.Tensor([1,2,3]) == torch.Tensor([[1,2,3]]) # single-dim Tensors are also row vectors
```

Out[39]:
```
tensor([[True, True, True]])
```

Reshaping and permuting

Sometimes it may be necessary to reshape the tensors to apply some specific operators: reshaping the way this are stored in memory.

Take the example of RGB images: they can be seen as `3 x h x w` tensors, where `h` is the height and `w` the width.

Sometimes, it may be necessary to "flatten" the three matrices into vectors, thus obtaining a `3 x hw` tensor.

In [40]:
```
image = torch.load("data/img.pt")
image.shape
```

Out[40]:
```
torch.Size([3, 243, 880])
```

This flattening may be achieved via the `reshape` method.

In [41]:
```
image_reshaped = image.reshape(3, 243*880)
pretty_print(image_reshaped.shape, "shape of image_reshaped")
```

```
shape of image_reshaped
torch.Size([3, 213840])
```

We can alternatively use the `view` method...

In [42]:
```
image_view = image.view(3, 243*880)
pretty_print(image_view.shape, "shape of image_view")
```

```
shape of image_view
torch.Size([3, 213840])
```

Q: what is the difference between `reshape` and `view`? USE RESHAPE FOR CHANGING DIM (safier)

- `view` does not copy the object. It's just a *view* of the original tensor.
- `reshape` sometimes copy the object, sometimes it does not and resorts to `view`

The cases when the two methods behave differently are marginal and we don't have time to discuss that here (you can see for instance this stackoverflow).

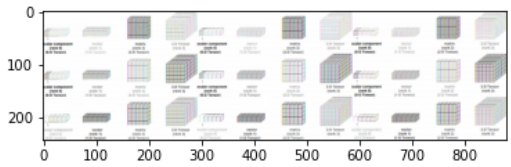
As a general rule, use `reshape` instead of `view` because the latter may throw unexpected errors, while with the former you're always sure your code will run seamlessly.

USE permute WHEN SWAPPING DIM (NOT CHANGING THEM)

Some libraries encode images as `h x w x 3` tensors instead of `3 x h x w`.

To convert between these two format, one may be tempted to `reshape` or `view` the tensor: in the end, they share the number of elements.

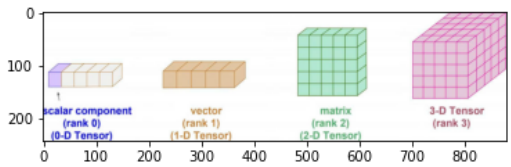
In [45]:
```
from matplotlib import pyplot as plt

image2 = image.reshape(243, 880, 3)

plt.imshow(image2)
plt.show()
```



That does not seem to work though: reshape does not change the order of the elements within the memory.

In order to do so, we need to use `permute`, which changes shape **and** the order of the elements. We need to pass the new order of the dimensions to it.

In [46]:
```
image3 = image.permute(1,2,0) # 1,2,0 --> old dim1 goes first, old dim2 goes second, dim0 goes last
# can also do image.permute(-2,-1,0) -- works with negative indices as well
plt.imshow(image3)
plt.show()
```



More linear algebra

In [47]:
```
z3_norm = z3.norm()
pretty_print(z3_norm, "Tensor norm")
pretty_print(np.linalg.norm(z3), "ndarray norm") # notice how torch is more OO
```

```
Tensor norm
tensor(4.1231)


ndarray norm
4.1231055
```

Notice how methods reducing `Tensor`s to scalars still return singleton `Tensor`s. (be wary of this feature when scripting something in PT)

To "disentangle" the scalar from a `Tensor` use the `.item()` method.

In [48]:
```
z3_norm.item()
```

Out[48]:
```
4.123105525970459
```

Note that, as for NumPy, PT supports `Tensor`s operator on a subset of its dimensions.

For example, given a `3x4x4 Tensor`, we might want to calculate the norm of each of the three `4x4` matrices composing it. We must hence specify to the `norm` method the dimensions on which we want it to operate the reduction:

In [49]:
```
z4 = torch.rand((3,4,5))
pretty_print(z4, "z4")
pretty_print(z4.norm(dim=(1,2)), "Norm of the three matrices composing z4 -- z4.norm(dim=(1,2))")
```

```
z4
tensor([[[0.9923, 0.0283, 0.0972, 0.6168, 0.8089],
         [0.2335, 0.3752, 0.1232, 0.7296, 0.0687],
         [0.2046, 0.6010, 0.6961, 0.4202, 0.8865],
         [0.5561, 0.7843, 0.9025, 0.3368, 0.6892]],

        [[0.9241, 0.0075, 0.6674, 0.0060, 0.3532],
         [0.3986, 0.5949, 0.5254, 0.3468, 0.9539],
         [0.3176, 0.5720, 0.9201, 0.2268, 0.0018],
         [0.7807, 0.1921, 0.6763, 0.2127, 0.0033]],

        [[0.7369, 0.2852, 0.4102, 0.6050, 0.0690],
         [0.3253, 0.8272, 0.0187, 0.7882, 0.5513],
         [0.7850, 0.0994, 0.7476, 0.6355, 0.9103],
         [0.4709, 0.6102, 0.9653, 0.0083, 0.4451]]])


Norm of the three matrices composing z4 -- z4.norm(dim=(1,2))
tensor([2.6372, 2.3913, 2.6497])
```

As expected, the result is a `1x3 Tensor`, showing the norm of each of the matrices.

We can notice this behaviour in other `Tensor` operator applying a reduction, for example `.sum()` and `.prod()` (sum/product of the elements within the tensor).

By specifying `z4.prod(dim=1)`, we **fix** the second dimension and loop through the other dimensions, applying the product for all of the resulting tensor slices. Let's see a code for clarity.

In [50]:
```
print(torch.empty([1,2,3])) # is the FASTES
print(torch.ones([1,2,3]))
torch.zeros([1,2,3])
```

Out[50]:
```
tensor([[[9.8091e-45, 0.0000e+00, 0.0000e+00],
         [0.0000e+00, 0.0000e+00, 0.0000e+00]]])
tensor([[[1., 1., 1.],
         [1., 1., 1.]]])
tensor([[[0., 0., 0.],
         [0., 0., 0.]]])
```

In [51]:
```
# 1. let us create an empty vector with the shape of the dimension we did NOT specify in `prod` above:
product_fix_dim = torch.empty([z4.shape[0], z4.shape[2]])

# 2. we loop through these dimensions to get the result of `z4.prod(dim=1)`
for i in range(product_fix_dim.shape[0]):
    for j in range(product_fix_dim.shape[1]):
        # the result for the (i,j) position is the product of the corresponding sub-tensor z4[i, :, j]
        # (we consider the whole of the 2nd dim, loop through the 1st and 3rd)
        product_fix_dim[i, j] = z4[i, :, j].prod()

pretty_print(product_fix_dim, "The result of our fancy loop ...")
pretty_print(z4.prod(dim=1), "... is the same as z4.prod(dim=1)")
```

```
The result of our fancy loop ...
tensor([[2.6366e-02, 5.0140e-03, 7.5216e-03, 6.3682e-02, 3.3938e-02],
        [9.1318e-02, 4.8829e-04, 2.1818e-01, 1.0109e-04, 1.9681e-06],
        [8.8635e-02, 1.4308e-02, 5.5350e-03, 2.5121e-03, 1.5415e-02]])


... is the same as z4.prod(dim=1)
tensor([[2.6366e-02, 5.0140e-03, 7.5216e-03, 6.3682e-02, 3.3938e-02],
        [9.1318e-02, 4.8829e-04, 2.1818e-01, 1.0109e-04, 1.9681e-06],
        [8.8635e-02, 1.4308e-02, 5.5350e-03, 2.5121e-03, 1.5415e-02]])
```

Analogously, we may apply the same reasoning to the `sum()` method:

`z4.sum(dim=0)`:

- we fix the first dim
- and loop through the other dims

In [52]:
```
pretty_print(z4.sum(dim=0), "z4.sum(dim=0) -- Element-wise sum of the three matrices composing the tensor -- is a 4x5
```

```
z4.sum(dim=0) -- Element-wise sum of the three matrices composing the tensor -- is a 4x5 matrix
tensor([[2.6534, 0.3210, 1.1748, 1.2279, 1.2311],
        [0.9574, 1.7973, 0.6673, 1.8646, 1.5739],
        [1.3073, 1.2724, 2.3638, 1.2825, 1.7986],
        [1.8077, 1.5866, 2.5441, 0.5578, 1.1376]])
```

this is analogous to

In [53]:
```
z4[0] + z4[1] + z4[2]
```

Out[53]:
```
tensor([[2.6534, 0.3210, 1.1748, 1.2279, 1.2311],
        [0.9574, 1.7973, 0.6673, 1.8646, 1.5739],
        [1.3073, 1.2724, 2.3638, 1.2825, 1.7986],
        [1.8077, 1.5866, 2.5441, 0.5578, 1.1376]])
```

Seamless conversion from NumPy to PT

In [55]:
```
y_numpy = np.random.rand(3,5)
y_torch = torch.from_numpy(y_numpy)
pretty_print(y_torch, "y converted to torch.Tensor")

x = torch.rand(6,4)
x_numpy = x.numpy()
pretty_print(x_numpy, "x converted to numpy.ndarray")

# Note that NumPy implicitly converts Tensor to ndarray whenever it can; the same doesn't happen for PT
pretty_print(np.linalg.norm(x), "Example of implicit conversion Tensor → ndarray (np.linalg.norm(x) where x is torch.

torch.norm(x_numpy) # this does not work
```

```
y converted to torch.Tensor
tensor([[0.1448, 0.6102, 0.3166, 0.7379, 0.5893],
        [0.2867, 0.3577, 0.7697, 0.2877, 0.7250],
        [0.4293, 0.3501, 0.9752, 0.8396, 0.6851]], dtype=torch.float64)


x converted to numpy.ndarray
[[0.45187956 0.65065175 0.47273642 0.18536246]
 [0.77414393 0.9459216  0.8039478  0.34784895]
 [0.76584774 0.48643023 0.30375892 0.73474973]
 [0.28091663 0.64067054 0.0202527  0.71824545]
 [0.0262472  0.62616765 0.9512985  0.9723446 ]
 [0.4328648  0.35083276 0.11401761 0.36862534]]


Example of implicit conversion Tensor → ndarray (np.linalg.norm(x) where x is torch.tensor)
2.8827488
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
/Users/valeria/Desktop/Corsi/DL/DSSC_DL_2022/labs/01-intro-to-pt.ipynb Cell 96' in <cell line: 12>()
      <a href='vscode-notebook-cell:/Users/valeria/Desktop/Corsi/DL/DSSC_DL_2022/labs/01-intro-to-pt.ipynb#ch0000095?
line=8'>9</a> # Note that NumPy implicitly converts Tensor to ndarray whenever it can; the same doesn't happen for PT
     <a href='vscode-notebook-cell:/Users/valeria/Desktop/Corsi/DL/DSSC_DL_2022/labs/01-intro-to-pt.ipynb#ch0000095?l
ine=9'>10</a> pretty_print(np.linalg.norm(x), "Example of implicit conversion Tensor → ndarray (np.linalg.norm(x) whe
re x is torch.tensor)")
---> <a href='vscode-notebook-cell:/Users/valeria/Desktop/Corsi/DL/DSSC_DL_2022/labs/01-intro-to-pt.ipynb#ch0000095?l
ine=11'>12</a> torch.norm(x_numpy)

File ~/opt/anaconda3/envs/DL_Lab/lib/python3.9/site-packages/torch/functional.py:1537, in norm(input, p, dim, keepdi
m, out, dtype)
   <a href='file:///Users/valeria/opt/anaconda3/envs/DL_Lab/lib/python3.9/site-packages/torch/functional.py?line=153
2'>1533</a> if has_torch_function_unary(input):
   <a href='file:///Users/valeria/opt/anaconda3/envs/DL_Lab/lib/python3.9/site-packages/torch/functional.py?line=153
3'>1534</a>     return handle_torch_function(
   <a href='file:///Users/valeria/opt/anaconda3/envs/DL_Lab/lib/python3.9/site-packages/torch/functional.py?line=153
4'>1535</a>         norm, (input,), input, p=p, dim=dim, keepdim=keepdim, out=out, dtype=dtype)
-> <a href='file:///Users/valeria/opt/anaconda3/envs/DL_Lab/lib/python3.9/site-packages/torch/functional.py?line=153
6'>1537</a> ndim = input.dim()
   <a href='file:///Users/valeria/opt/anaconda3/envs/DL_Lab/lib/python3.9/site-packages/torch/functional.py?line=153
8'>1539</a> # catch default case
   <a href='file:///Users/valeria/opt/anaconda3/envs/DL_Lab/lib/python3.9/site-packages/torch/functional.py?line=153
9'>1540</a> if dim is None and out is None and dtype is None and p is not None:

AttributeError: 'numpy.ndarray' object has no attribute 'dim'
```

Stochastic functionalities

We can render the (pseudo)random number generator deterministic by calling `torch.manual_seed(integer)`.

This works for both CPU and CUDA RNG calls.

In [56]:
```
torch.manual_seed(123456)
print("...from now on our random tensor should be the same...")
```

```
...from now on our random tensor should be the same...
```

In [57]:
```
pretty_print(torch.randperm(10), "(randperm) Random permutation of 0:10")

pretty_print(torch.rand_like(z1), "(rand_like) Create random vector with the same shape of z_1, but random elem betwe

pretty_print(torch.randint(10, (3, 3)), "(randint) Like rand, but with integers up to 10")

pretty_print(torch.normal(0, 1, (3, 3)), "(normal) Sampling a 3x3 iid scalars from N(0,1)")

pretty_print(torch.normal(torch.Tensor([[1,2,3],[4,5,6],[0,0,0]]), torch.Tensor([[1,0.5,0.9],[0.5,1,0.1],[3,4,1]])),
```

```
(randperm) Random permutation of 0:10
tensor([5, 2, 4, 9, 1, 7, 3, 6, 0, 8])


(rand_like) Create random vector with the same shape of z_1, but random elem between 0,1
tensor([[0.4954, 0.0728, 0.9644, 0.5524, 0.0060],
        [0.1053, 0.2431, 0.5141, 0.2926, 0.0147],
        [0.3049, 0.4911, 0.6739, 0.6872, 0.9038],
        [0.1368, 0.2698, 0.1289, 0.3020, 0.0194]])


(randint) Like rand, but with integers up to 10
tensor([[7, 6, 2],
        [9, 0, 3],
        [0, 4, 4]])


(normal) Sampling a 3x3 iid scalars from N(0,1)
tensor([[-0.4107, -1.5861, -0.4778],
        [ 1.0268,  1.2428,  0.4147],
        [-0.6491,  0.0784, -0.7173]])


Sampling from 9 normals with different means and std into a (3x3) Tensor
tensor([[ 1.1070,  1.9439,  2.3757],
        [ 3.6595,  5.8618,  5.9344],
        [ 5.0647, -1.0821, -0.0713]])
```

Using GPUs

All `Torch.Tensor` methods support GPU computation via built-in CUDA wrappers.

Just transfer the involved `Tensor`s to CUDA and let the magic happen :)

In [58]:
```
# check if cuda is available on this machine
torch.cuda.is_available()

has_cuda_gpu = torch.cuda.is_available()
```

In [59]:
```
print(has_cuda_gpu)
```

```
False
```

In [60]:
```
if has_cuda_gpu:
    dim = 10000
    large_cpu_matrix = torch.rand((dim, dim))
    large_gpu_matrix = large_cpu_matrix.cuda() # Can also specify "cuda:gpu_id" if multiple GPUs
    # alternatively, you may also call large_cpu_matrix.cuda() or large_cpu_matrix.cuda(0)
else:
    print("Sorry, this part of the notebook is inaccessible since it seems you don't have a CUDA-capable GPU on your
```

```
Sorry, this part of the notebook is inaccessible since it seems you don't have a CUDA-capable GPU on your device :/
```

In [61]:
```
pretty_print(large_cpu_matrix.device, "Device of large_cpu_matrix")
pretty_print(large_gpu_matrix.device, "Device of large_gpu_matrix")
pretty_print(large_gpu_matrix, "If a tensor is not on CPU, the device will also be printed if you print the tensor it
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
/Users/valeria/Desktop/Corsi/DL/DSSC_DL_2022/labs/01-intro-to-pt.ipynb Cell 105' in <cell line: 1>()
----> <a href='vscode-notebook-cell:/Users/valeria/Desktop/Corsi/DL/DSSC_DL_2022/labs/01-intro-to-pt.ipynb#ch0000104?
line=0'>1</a> pretty_print(large_cpu_matrix.device, "Device of large_cpu_matrix")
      <a href='vscode-notebook-cell:/Users/valeria/Desktop/Corsi/DL/DSSC_DL_2022/labs/01-intro-to-pt.ipynb#ch0000104?
line=1'>2</a> pretty_print(large_gpu_matrix.device, "Device of large_gpu_matrix")
      <a href='vscode-notebook-cell:/Users/valeria/Desktop/Corsi/DL/DSSC_DL_2022/labs/01-intro-to-pt.ipynb#ch0000104?
line=2'>3</a> pretty_print(large_gpu_matrix, "If a tensor is not on CPU, the device will also be printed if you print
the tensor itself")

NameError: name 'large_cpu_matrix' is not defined
```

In [62]:
```
if has_cuda_gpu:
    import timeit

    # NOTE: please fix this number w.r.t. your GPU and CPU
    repetitions = 100

    print("Norm of large cpu matrix. Time:", timeit.timeit("large_cpu_matrix.norm()", number=repetitions, globals=loc
    print("Norm of large gpu matrix. Time:", timeit.timeit("large_gpu_matrix.norm()", number=repetitions, globals=loc
else:
    print("Sorry, this part of the notebook is inaccessible since it seems you don't have a CUDA-capable GPU on your
```

```
Sorry, this part of the notebook is inaccessible since it seems you don't have a CUDA-capable GPU on your device :/
```

In [63]:
```
large_cpu_matrix2 = torch.rand((dim, dim))


print("Norm of large cpu matrix. Time:", timeit.timeit("large_cpu_matrix.norm()", number=100, globals=locals()))
print("Norm of large gpu matrix. Time:", timeit.timeit("large_cpu_matrix.cuda().norm()", number=100, globals=locals()
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
/Users/valeria/Desktop/Corsi/DL/DSSC_DL_2022/labs/01-intro-to-pt.ipynb Cell 107' in <cell line: 1>()
----> <a href='vscode-notebook-cell:/Users/valeria/Desktop/Corsi/DL/DSSC_DL_2022/labs/01-intro-to-pt.ipynb#ch0000106?line=0'>1</a> large_cpu_matrix2 = torch.rand((dim, dim))
      <a href='vscode-notebook-cell:/Users/valeria/Desktop/Corsi/DL/DSSC_DL_2022/labs/01-intro-to-pt.ipynb#ch0000106?line=3'>4</a> print("Norm of large cpu matrix. Time:", timeit.timeit("large_cpu_matrix.norm()", number=100, globals=locals()))
      <a href='vscode-notebook-cell:/Users/valeria/Desktop/Corsi/DL/DSSC_DL_2022/labs/01-intro-to-pt.ipynb#ch0000106?line=4'>5</a> print("Norm of large gpu matrix. Time:", timeit.timeit("large_cpu_matrix.cuda().norm()", number=100, globals=locals()))

NameError: name 'dim' is not defined
```

Q: Why that difference between the first and the second cells?

Captain obvious: Use `tensor.cpu()` or `tensor.to("cpu")` to move a tensor to your CPU

Building easy ML models

By using all the pieces we've seen till now, we can build our first ML model using PyTorch: a linear regressor, whose model is

$$y = XW + b$$

which can also be simplified as

$$y = XW$$

if we incorporate the bias b inside W and add to the X a column of ones to the right.

We'll first create our data. The X 's are the 0:9 range plus some iid white noise, while the y is just the 0:9 range

In [64]:
```
x1 = torch.arange(0, 10).float().unsqueeze(-1)#from row to column vector
x2 = torch.arange(0, 10).float().unsqueeze(-1)
x3 = torch.arange(0, 10).float().unsqueeze(-1)
x0 = torch.ones([10]).float().unsqueeze(-1)
# first, concatenate by column (dim=1 not 0) x1, x2, and x3 to form a single matrix X
X = torch.cat((x1, x2, x3), dim=1)
# add small noise to the Xs, so that we don't have a trivial "interpolation" problem
eps = torch.normal(0, .3, (10, 3)) #creates a matrix of same shape with values from a gaussian with mean=0 and std.de
X += eps
# concatenate also x0 so we can express the model as y=XW
X = torch.cat((X, x0), dim=1)

pretty_print(X, "X (covariates)")
```

```
X (covariates)
tensor([[ 0.0698, -0.0294, -0.0875,  1.0000],
        [ 1.1936,  0.9067,  1.4257,  1.0000],
        [ 2.0887,  2.0520,  2.0334,  1.0000],
        [ 3.0333,  2.7806,  2.8838,  1.0000],
        [ 4.2332,  3.8806,  3.9750,  1.0000],
        [ 4.9830,  4.9553,  5.3448,  1.0000],
        [ 5.6296,  6.3334,  6.0269,  1.0000],
        [ 7.1786,  6.7816,  7.2288,  1.0000],
        [ 7.7853,  7.9722,  7.8963,  1.0000],
        [ 8.5020,  9.2573,  8.9184,  1.0000]])
```

Q: how can we obtain the same `X` but first concatenating, then summing the white noise?

In [65]:
```
x1 = torch.arange(0, 10).float().unsqueeze(-1)
x2 = torch.arange(0, 10).float().unsqueeze(-1)
x3 = torch.arange(0, 10).float().unsqueeze(-1)
x0 = torch.ones([10]).float().unsqueeze(-1)

X = torch.cat((x1, x2, x3, x0), dim=1)
X[:, (0,1,2)] += (eps)
X
```

Out[65]:
```
tensor([[ 0.0698, -0.0294, -0.0875,  1.0000],
        [ 1.1936,  0.9067,  1.4257,  1.0000],
        [ 2.0887,  2.0520,  2.0334,  1.0000],
        [ 3.0333,  2.7806,  2.8838,  1.0000],
        [ 4.2332,  3.8806,  3.9750,  1.0000],
        [ 4.9830,  4.9553,  5.3448,  1.0000],
        [ 5.6296,  6.3334,  6.0269,  1.0000],
        [ 7.1786,  6.7816,  7.2288,  1.0000],
        [ 7.7853,  7.9722,  7.8963,  1.0000],
        [ 8.5020,  9.2573,  8.9184,  1.0000]])
```

In [66]:
```
y = torch.arange(0, 10).float().unsqueeze(-1)

pretty_print(y, "y (response)")
```

```
y (response)
tensor([[0.],
        [1.],
        [2.],
        [3.],
        [4.],
        [5.],
        [6.],
        [7.],
        [8.],
        [9.]])
```

$$y = XW$$

In [67]:
```
print(y, X)
```

```
tensor([[0.],
        [1.],
        [2.],
        [3.],
        [4.],
        [5.],
        [6.],
        [7.],
        [8.],
        [9.]]) tensor([[ 0.0698, -0.0294, -0.0875,  1.0000],
        [ 1.1936,  0.9067,  1.4257,  1.0000],
        [ 2.0887,  2.0520,  2.0334,  1.0000],
        [ 3.0333,  2.7806,  2.8838,  1.0000],
        [ 4.2332,  3.8806,  3.9750,  1.0000],
        [ 4.9830,  4.9553,  5.3448,  1.0000],
        [ 5.6296,  6.3334,  6.0269,  1.0000],
        [ 7.1786,  6.7816,  7.2288,  1.0000],
        [ 7.7853,  7.9722,  7.8963,  1.0000],
        [ 8.5020,  9.2573,  8.9184,  1.0000]])
```

For the case of linear regression, we usually wish to obtain a set of weights minimizing the so called mean square error/loss (MSE), which is the squared difference between the ground truth and the model prediction, summed for each data instance.

We know that the OLS/Max Likelihood esitmator is the one yielding the optimal set of weights in that regard.

$$\hat{W} = (X^\top X)^{-1} X^\top y$$

In [68]:
```
W_hat = ((X.T @ X).inverse()) @ X.T @ y # OLS estimator

pretty_print(W_hat, "W (optimal weights/coefficients [first 3] and bias/intercept [last one])")
```

```
W (optimal weights/coefficients [first 3] and bias/intercept [last one])
tensor([[ 0.5034],
        [ 0.5443],
        [-0.0322],
        [-0.0463]])
```

In Linear Regression we don't need to train the model since the optimal choice for the coeff can be easly compute via maximum likelihood estimation.

This is not the case for Neural Networks most of the times, but we must to train it with stochastic gradiant descendent.

We can evaluate our model on the mean square loss

Q: what is its formula?

$$L(y, \hat{y}) = \sum_{i=1}^{n} (y_i - \hat{y}_i)^2 / n$$

In [69]:
```
def mean_square_loss(y, y_hat):
    return ((y - y_hat)**2).sum() / y.size(0) # or y.shape[0] -> first elem of the shape
```

In [70]:
```
# GPUs can perform basic operation in parallel (not the power): are more stupid than CPUs
tensor1 = torch.rand((3,4))
tensor2 = torch.rand_like(tensor1).cuda()
tensor1 + tensor2.cpu()
# without .cpu() on tensor2, I get an error: ALL TENSORS MUST BE ON SAME DEVICE (either cpu o gpu)
# or I could : tensor1.cuda() + tensor2
```

```
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
/Users/valeria/Desktop/Corsi/DL/DSSC_DL_2022/labs/01-intro-to-pt.ipynb Cell 122' in <cell line: 3>()
      <a href='vscode-notebook-cell:/Users/valeria/Desktop/Corsi/DL/DSSC_DL_2022/labs/01-intro-to-pt.ipynb#ch0000121?line=0'>1</a> # GPUs can perform basic operation in parallel (not the power): are more stupid than CPUs
      <a href='vscode-notebook-cell:/Users/valeria/Desktop/Corsi/DL/DSSC_DL_2022/labs/01-intro-to-pt.ipynb#ch0000121?line=1'>2</a> tensor1 = torch.rand((3,4))
----> <a href='vscode-notebook-cell:/Users/valeria/Desktop/Corsi/DL/DSSC_DL_2022/labs/01-intro-to-pt.ipynb#ch0000121?line=2'>3</a> tensor2 = torch.rand_like(tensor1).cuda()
      <a href='vscode-notebook-cell:/Users/valeria/Desktop/Corsi/DL/DSSC_DL_2022/labs/01-intro-to-pt.ipynb#ch0000121?line=3'>4</a> tensor1 + tensor2.cpu()

File ~/opt/anaconda3/envs/DL_Lab/lib/python3.9/site-packages/torch/cuda/__init__.py:210, in _lazy_init()
    <a href='file:///Users/valeria/opt/anaconda3/envs/DL_Lab/lib/python3.9/site-packages/torch/cuda/__init__.py?line=205'>206</a>     raise RuntimeError(
    <a href='file:///Users/valeria/opt/anaconda3/envs/DL_Lab/lib/python3.9/site-packages/torch/cuda/__init__.py?line=206'>207</a>         "Cannot re-initialize CUDA in forked subprocess. To use CUDA with "
    <a href='file:///Users/valeria/opt/anaconda3/envs/DL_Lab/lib/python3.9/site-packages/torch/cuda/__init__.py?line=207'>208</a>         "multiprocessing, you must use the 'spawn' start method")
    <a href='file:///Users/valeria/opt/anaconda3/envs/DL_Lab/lib/python3.9/site-packages/torch/cuda/__init__.py?line=208'>209</a> if not hasattr(torch._C, '_cuda_getDeviceCount'):
--> <a href='file:///Users/valeria/opt/anaconda3/envs/DL_Lab/lib/python3.9/site-packages/torch/cuda/__init__.py?line=209'>210</a>     raise AssertionError("Torch not compiled with CUDA enabled")
    <a href='file:///Users/valeria/opt/anaconda3/envs/DL_Lab/lib/python3.9/site-packages/torch/cuda/__init__.py?line=210'>211</a> if _cudart is None:
    <a href='file:///Users/valeria/opt/anaconda3/envs/DL_Lab/lib/python3.9/site-packages/torch/cuda/__init__.py?line=211'>212</a>     raise AssertionError(
    <a href='file:///Users/valeria/opt/anaconda3/envs/DL_Lab/lib/python3.9/site-packages/torch/cuda/__init__.py?line=212'>213</a>         "libcudart functions unavailable. It looks like you have a broken build?")

AssertionError: Torch not compiled with CUDA enabled
```

Let's apply it to our data.

First we need to obtain the predictions, then we can evaluate the MSE.

In [71]:
```
y_hat = X @ W_hat
pretty_print(y_hat, "Predictions (y_hat)")

pretty_print(mean_square_loss(y, y_hat), "Loss (MSE)")
```

```
Predictions (y_hat)
tensor([[-0.0243],
        [ 1.0021],
        [ 2.0566],
        [ 2.9013],
        [ 4.0689],
        [ 4.9872],
        [ 6.0409],
        [ 7.0258],
        [ 7.9579],
        [ 8.9852]])


Loss (MSE)
tensor(0.0023)
```

In [72]:
```
# We arrived here
```

Using PT built-ins

We will now be exploring the second chunk of PT functionalities, namely the built-in structures and routines supporting the creation of ML models.

We can create the same model we have seen before using PT built-in structures, so we start to see them right away.

Usually, a PT model is a `class` inheriting from `torch.nn.Module`. Inside this class, we'll define two methods:

- the constructor (`__init__`) in which we define the building blocks of our model as class variables (aka W the set of parameters) (later during our lectures we'll see more "elegant" methods to build models architectures)
- the `forward` method, which specifies how the data fed into the model needs to be processed in order to produce the output

Note for those who already know something about NNs: we don't need to define `backward` methods (used to evaluate the gradient) since we're constructing our model with built-in PT building blocks. PT automatically creates a `backward` routine based upon the `forward` method.

Our model only has one building block (layer doing the matrix moltiplication between the input and the params) which is a `Linear` layer. We need to specify the size of the input (i.e. the coefficients `W` of our linear regressor) and the size of the output (i.e. how many scalars it produces) of the layer. We additionaly request our layer to have a bias term `b` (which acts as the intercept of the hyperplane we saw before).

The `Linear` layer processes its input as `XW + b`, which is exactly the (first) equation of the linear regressor we saw before.

In [73]:
```
class LinearRegressor(torch.nn.Module): #inherites
```

```
    def __init__(self): #constructor: overrides the father unless you specify super
        super().__init__() # always when inheriting: supers refers to the father's constructor
        # We create a Linear layer which does X*W+b with X as input (W  weights, b bias)
        # We encorporate X1 X2 X3 (input_features=3) + dummy X0 for the bias
        self.regressor = torch.nn.Linear(in_features=3, out_features=1, bias=True) # 1 output, bias=True by default
    #OSS: wieghts are initialized randomly depending on the type you use: usually for Pytorch there are 2 types
    # Usually it'a gaussian initialization
    # Anyways weights should enhence information flow

    # How data flows through our model to create an output
    def forward(self, X): #forward: it's the propagation of information from the input layer to the output
        return self.regressor(X) # we return the output of the regressor
    # Although it seems a kind of over head, in complex neural networks it is not
```

We can create an instance of our model and inspect the current parameters by using the `state_dict` method, which prints the building blocks of our model and their current parameters. Note that `state_dict` is essentially a dictonary indexed by the names of the building blocks which we defined inside the constructor (plus some additional identifiers if a layer has more than one set of parameters).

In [74]:
```
lin_reg = LinearRegressor()

for param_name, param in lin_reg.state_dict().items():#returns a dictonary of parameters of my lin_reg
    #.items() applied to a dictionary: returns tuples wich are the keys (param_name) amnd the values (param)
    # you can iterate dictonaries by keys, values, values and keys
    print(param_name, param) # random initialization
```

```
regressor.weight tensor([[0.5098, 0.0164, 0.4876]])
regressor.bias tensor([-0.4917])
```

We can update the parameters via `state_dict` and re-using the same OLS estimates we obtained before.

Note that PT is thought of for Deep Learning: it does not have (I think) the routines to solve different ML problems.

Next time, we'll see how we can unleash PT's gradient-based iterative training routines and compare the results w.r.t. the OLS estimators.

In [75]:
```
state_dict = lin_reg.state_dict()
# now state_dict is a copy of the param of the model: if we change stat.dict
# we will not update the model lin_reg, so let's first update the regressor_weight
# regressor_weight it has 3 param
state_dict["regressor.weight"] = W_hat[:3].T # we transpose: remember the output of W_hat was a column vector!
state_dict["regressor.bias"] = W_hat[3]
lin_reg.load_state_dict(state_dict) # this for override
```

Out[75]:
```
<All keys matched successfully>
```

In [76]:
```
# Check if it worked
for param_name, param in lin_reg.state_dict().items():
    print(param_name, param)
```

```
regressor.weight tensor([[ 0.5034,  0.5443, -0.0322]])
regressor.bias tensor([-0.0463])
```

The `forward` method gets implicitly called by passing the data to our model's instance `lin_reg`:

In [77]:
```
X_lin_reg = X[:,:3] #X has 10 rows and 4 columns : the 4th column is not needed (dummy variable)
predictions_lin_reg = lin_reg(X_lin_reg)# forward method is implicity called: how is it possible?
# cause directly I call the instance!
# Thanks to self.regressor (?) ASK
# or I can write: lin_reg.forward(X_lin_reg) ->github.Pythorc (?)
pretty_print(predictions_lin_reg, "Predictions of torch class")
# now we have grad_fn=<AddmmBackward> from the output tensor
# We didn't train anything It states what we will call as a backward method:
# the last building block we used to get the output is a matrix or an add multiply Addmm
```

```
Predictions of torch class
tensor([[-0.0243],
        [ 1.0021],
        [ 2.0566],
        [ 2.9013],
        [ 4.0689],
        [ 4.9872],
        [ 6.0409],
        [ 7.0258],
        [ 7.9579],
        [ 8.9852]], grad_fn=<AddmmBackward0>)
```

The predictions are the same as before

In [78]:
```
pretty_print(y_hat, "Predictions of linear model")
```

```
Predictions of linear model
tensor([[-0.0243],
        [ 1.0021],
        [ 2.0566],
        [ 2.9013],
        [ 4.0689],
        [ 4.9872],
        [ 6.0409],
        [ 7.0258],
        [ 7.9579],
        [ 8.9852]])
```

Adding non-linearity

One of the staples of DL is that the relationship between the `X` s and the predictions is **non-linear**.

The non-linearity is obtain by applying a non-linear function (called *activation function*) after each linear layer.

We can complicate just a little bit our linear model to create a **logistic regressor**:

$y = \text{logistic}(XW + b),$

where $\text{logistic}(z) = \exp(z)/(1 + \exp(z))$

The logistic function has different names:

- in statistics, it's usually called *inverse logit* as well
- in DL and mathematics, it's called *sigmoid function* due to its "S" shape and is often denoted with the symbol σ

Hystorically, the sigmoid was between the first activation functions used in NNs.

Logistic regression is usually used as a **binary classification model** instead of a regression model. In this setting, we suppose we have two destination classes to which we assign values 0 and 1: `y ∈ {0, 1}`. Since the codomain of the sigmoid is `[0,1]`, we can interpret its output `ŷ` as a probability value, and assign each data to the class 0 if `ŷ <= 0.5`, to the class 1 otherwise.

In [79]:
```
y = torch.Tensor([0,1,0,0,1,1,1,0,1,1])# vector of floating point numbers
pretty_print(y, "y for our classification problem")
```

```
y for our classification problem
tensor([0., 1., 0., 0., 1., 1., 1., 0., 1., 1.])
```

Note that we may also want our y to be a vector of `int` s. We can convert the `Tensor` type to `int` by calling the method `.long()` or `.int()` of `Tensor`. Usually, PT uses the long datatype for labels.

As in NumPy, the type of the `Tensor` is found within the `dtype` variable of the given `Tensor`.

In [80]:
```
y = y.long()
pretty_print(y, "y converted to int")
pretty_print(y.dtype, "Data type of y")
```

```
y converted to int
tensor([0, 1, 0, 0, 1, 1, 1, 0, 1, 1])


Data type of y
torch.int64
```

Let us now build our logistic regressor in PT.

We only need one single addition wrt the linear regressor: in the `forward` method, we'll add the sigmoidal non-linearity by calling the `sigmoid` function within the `torch.nn.functional` library.

Note that there also exist some "mirror" alias of these functionals within `torch.nn` (e.g. `torch.nn.Sigmoid`): we'll learn in the following lecture why these aliases are there and how to use them.

In [81]:
```
class LogisticRegressor(torch.nn.Module):
    # how can I rewrite it by using inheritance?
    # Logistic regressor could inherit from Linear regressor in the forward part
    # So any change in Linear is propagated to the other
    def __init__(self):
        super().__init__()
        # no difference wrt linear regressor
        self.regressor = torch.nn.Linear(in_features=3, out_features=1)# bias is by def True

    def forward(self, X):
        # if LogisticRegressor inherites from Linear I could write:
        # super.forward()
        out = self.regressor(X) #intermidate var
```

```
        # here we apply the sigmoid fct to the output of regressor
        out = torch.sigmoid(out)
        return out
```

We can instantiate our logistic regressor and use it to calculate our predictions on the same X as before.

Note that **we're using the initial (random) weights which PT has assigned to the model parameters**. For our linear regressor, we were able to analytically obtain the OLS value of the parameters. In the case of logistic regression, there's no MaxLikelihood estimator obtainable in close form and we need to resort to numerical methods to obtain them. Since the part concerning numerical optimization will be discussed during the next Lab, we will not be training our model (hence results will obviously be sub-par).

In [82]:
```
log_reg = LogisticRegressor()
y_hat = log_reg(X_lin_reg)
pretty_print(y_hat, "logistic regressor predictions")
```

```
logistic regressor predictions
tensor([[4.5397e-01],
        [2.0006e-01],
        [8.4300e-02],
        [3.8079e-02],
        [1.2066e-02],
        [3.7630e-03],
        [1.2369e-03],
        [5.0629e-04],
        [1.8706e-04],
        [5.7492e-05]], grad_fn=<SigmoidBackward0>)
```

There exist many ways to evaluate the performance of the logistic regressor: one of them is **accuracy** (correctly identified units / total number of units). We can define a function to evaluate accuracy and calculate it on our model and data

Other measures of performance in binary classification are FPR, FNR or you can change the threshold of the sigmoid function (from 0.5 to 0.2 for example), and evaluate the ROC curve of your classifier. Accuracy in fact is not good when you have imbalanced data!

In [83]:
```
def accuracy(y, y_hat):
    # Assign each y_hat to its predicted class
    pred_classes = torch.where(y_hat < .5, 0, 1).squeeze().long() # where y_hat is < .5 puts a 0 otherwise a 1
    # .squeeze() eliminate all the singleton dimensions! So it becomes a vector of size 4
    # Recall that the predicted values are not the predicted classes
    correct = (pred_classes == y).sum()
    # .sum() we are summing booleans so what does sum do?
    # it does an implicit conversion and then produces a sum
    return (correct / y.shape[0]).item()
```

In [84]:
```
accuracy(y, y_hat)
```

Out[84]:
```
0.4000000059604645
```

OSS: If accuracy is 0.10 is very good at guessing wrongly so just swap it!

Visualizing linear and logistic regression as a computational graph

We now need to convert the equation of the linear and the logistic regression:

- `y = σ(WX + b)`

where `σ` is a generic `ℝ → ℝ` function: sigmoid for logistic regression, identity for linear regression.

We organize the input in *nodes* (on the left part) s.t. each node represents one dimension/covariate. For each data instance, we substitute to each node the corresponding numeric value. The nodes undergo one or more operations, namely, from left to right:

1. Each node is multiplied by its corresponding weight (a value placed on the edge indicates that the node is multiplied by said value)
2. All the corresponding outputs are summed together

These two operations identify the dot product between vectors `X` and `W`

1. The bias term `b` is added
2. The non-linear function `σ` is applied to the result of this sum
3. Finally, we assign that value to the variable `ŷ`, which is also indicated as a node

Q: what is the relationship between b and X_4?

Our first MultiLayer Perceptron (MLP)

The MLP is a family of Artificial NNs in which the input is a vector of size `ℝ^d` and the output is again a vector of size `ℝ^p`, where `p` is determined upon the nature of the problem we wish to solve. Additionally, a MLP is characterized by multiple stages (*layers*) of sequential vector-matrix multiplication and non-linearity (steps 1., 2., 3. above) in which each output of the layer `l-1` acts as input to the layer `l`.

Taking inspiration to the graph of the logistic regression, we can translate all into an image to give sense to these words:

In NNs, each of the nodes within the graph is called a **neuron**

Neurons are organized in **layers**

In computational graphs, layers are shown from left to right (or bottom to top sometimes), which is the direction of the flow of information inside the NN.

The first layer is called **input layer** and represents the dimensions of our data.

The last layer is called **output layer** and represent the output of our NN.

All the intermediate layers are called **hidden layers**. To be defined MLP, there must be at least one hidden layer inside our model.

If the NN is an MLP, each neuron in a given layer (except for the input) receives information from every neuron of the previous; moreover, each neuron in any layer (except for the output) sends information to every neuron of the next layer. There's no communication between neurons of the same layer (if it happens, we have a **Recursive Neural Network**).

For the sake of brevity, usually in NN computational graphs we drop the blocks `+` and `σ`, the values of weights, and the reference to the bias terms, remaining with a scheme conveying info about

- the number of neurons per layer
- the connectivity of neurons

The graph above becomes:

We can then start programming our simple MLP in PT.

We will suppose that our MLP is for **binary classification**, hence the activation function `τ` is the sigmoid.

In [85]:
```
class MLP(torch.nn.Module):
    def __init__(self):
        super().__init__()
        # 3 input variables, 2 neurons as output in the hidden layer
        self.layer1 = torch.nn.Linear(in_features=3, out_features=2)
        # 2 input neurons, 1 single output
        self.layer2 = torch.nn.Linear(in_features=2, out_features=1)

    def forward(self, X):
        #activation functions:
        out = self.layer1(X)
        out = torch.nn.functional.relu(out)
        # relu function is the most famous for hidden layers:
        # it is a non-linear function mathmatically described as:
        # f(x)= max(0,x)
        # so if x>0 return x,
        # otherwise return 0.
        out = self.layer2(out)
        out = torch.nn.functional.sigmoid(out)
        return out
```

For the great majority of MLP, it's very hard to get analytical solutions to our sets of weights and biases. We then resort to numerical methods for optimization.

In DL, we normally used gradient-based methods like Stochastic Gradient Descent with *backpropagation* to find approximate solutions.

We'll cover these topics in future lectures. For now, the focus is to build a MLP in PT and perform the *forward pass* (=evaluate the model on a set of data).

We can analyse the structure of our MLP by just printing the model

In [86]:
```
model = MLP()
model # print does the summay of the layers
```

Out[86]:
```
MLP(
  (layer1): Linear(in_features=3, out_features=2, bias=True)
  (layer2): Linear(in_features=2, out_features=1, bias=True)
)
```

although we might wanna have additional informations.

There's an additional package, called `torchinfo` which helps us producing more informative and exhaustive model summaries.

We can install it from the terminal:

- activate the conda env
- `conda install -c conda-forge torchinfo`

On Colab, remember that you can execute bash commands prepending a `!` to the command itself.

In [89]:
```
from torchinfo import summary
summary(model)
```

Out[89]:
```
==========================================================================
Layer (type:depth-idx)                   Param #
==========================================================================
MLP                                      --
├─Linear: 1-1                            8
├─Linear: 1-2                            3
==========================================================================
Total params: 11
Trainable params: 11
Non-trainable params: 0
==========================================================================
```

Usage of `Sequential`

If we have a neural network whose forward pass just processes the layers in sequence in a given order, we can also wrap the layers inside a `torch.nn.Sequential` structure.

Note that:

1. In this case, all the activation functions must be passed as their `nn.Module` counterpart, not as `torch.nn.functional` functions
2. The data flows in the layers according to the order dictated in the constructor

Q: How can I build the `forward` method now?

In [90]:
```
class MLP(torch.nn.Module):
    def __init__(self):
        super().__init__()
        self.layers = torch.nn.Sequential(
            torch.nn.Linear(3, 2),
            torch.nn.ReLU(),
            torch.nn.Linear(2, 1),
            torch.nn.Sigmoid()
        )

    def forward(self, X):
        # how can I complete it?
        return self.layers(X)
        # return self.layers(X, torch.nn.functional.relu(), ?, torch.nn.functional.sigmoid())
```

Let us suppose we wish to build a larger model from the graph below.

We suppose that

1. The layers have no bias units
2. The activation function for hidden layers is `ReLU`

$$\mathrm{ReLU}(x) = \max(0, x)$$

Moreover, we suppose that this is a classification problem.

As you might recall, when the number of classes is > 2, we encode the problem in such a way that the output layer has a no. of neurons corresponding to the no. of classes. Doing so, we establish a correspondence between output units and classes. The value of the j-th neuron represents the **confidence** of the network in assigning a given data instance to the j-th class.

Classically, when the network is encoded in such way, the activation function for the final layer is the **softmax** function. If C is the total number of classes,

$$softmax(z_j) = \frac{\exp(z_j)}{\sum_{k=1}^{C} \exp(z_k)}$$

where $j \in \{1, \cdots, C\}$ is one of the classes.

If we repeat this calculation for all j s, we end up with C normalized values (i.e., between 0 and 1) which can be interpreted as a confidence that the network has in assigning the instance to the corresponding class.

Homework

1. build the MLP in the image above using PT built-ins
2. Provide calculation for the exact number of parameters of the MLP
   - Do it first supposing that the layers don't have a bias term, then supposing that the bias is present wherever it's possible
3. Calculate the L_1 (vectorial) norm and the Frobenius norm for the params of each layer
4. Given 10 random datapoints, feed them into the network. This operation must be done all in one single command and must **not** make use of loops.
   - Given the output of the network, using PyTorch code, find the class of assignment of each datapoint. This also must be done in a single PyTorch command without using loops.
   - Drafting a vector of ground truths (whichever labels you like), provide code for the calculation of the accuracy
     - Tip: first get the number of correct assignments, then...