# Notes : how to make and train a network using PyTorch

# AUTOGRAD

It is a define-by-run framework, which means that your backprop (rétropropagation) is defined by how your code is run, and that every single iteration can be different.

## Tensor

**Central class of the PyTorch package : torch.Tensor .**

Attributes :

* *.requires\_grad* : boolean. If True, it tracks all operations on the tensor. To stop a tensor from tracking operations, one can call .detach().

requires\_grad is by default set to False. To modify the requires\_grad attribute of an already existing tensor, one can use .requires\_grad\_(desired value).

* *.grad\_fn* : reference to a Function. Stocks the operations used to build the tensor.

**Constructors** :

* *Torch.tensor( liste )* : Creates a tensor from a Python List or a np.array.
* *Torch.zeros([height, length], dtype = , device = )* : Creates a tensor with the required dimensions, of the required type, with coordinates 0 .
* *Torch.ones([height, length], dtype = , device = )* : Same as torch.zeros, but the coordinates are equal to 1.

**Methods** : we take x, y tensors of same dimensions.

* To access an element of a tensor : same as for matrixes : x[i][j].
* To access the value of a tensor of size 1x1 : x.item()
* x + int : adds int to each coordinate of x.
* z = x\*y : is a scalar product. z has the same dimensions as x and y. For all i j, z[i][j] = x[i][j] \* y[i][j].
* x.backward() : if x is a function of another tensor z, then it computes the gradient of x with regard to z. If x is a scalar (ie of size 1x1), there is no need for an argument for backward(). Z.grad will stock the value of the gradient calculated by backward() .
* <https://pytorch.org/docs/stable/tensors.html>

# NEURAL NETWORKS

A typical training procedure for a neural network is as follows:

1. Define the neural network that has some learnable parameters (or weights)
2. Iterate over a dataset of inputs
3. Process input through the network
4. Compute the loss (how far is the output from being correct)
5. Propagate gradients back into the network’s parameters
6. Update the weights of the network, typically using a simple update rule: weight = weight - learning\_rate \* gradient

## Define the nn and its learnable parameters

**Each class of nn one defines must herit from nn.Module.** In the class, *one must define*

* *A constructor (\_\_init\_\_(self, parameters),* in which appear the layers of the model and their learnables.
* *A method forward(self, input) that* pushes the input through the nn and returns an output. In our example, input = x, output = fapprox(x).

Attributes of nn.Modules :

* *.parameters()* : returns the learnables of the model (ie the parameters that will be modified during the training). In the case of our model (nn.Linear), It is a collection of tensors representing the matrixes A and b of the network.

Use command print( list(net.parameters()) ) to print the learnables. I verified that nn.Linear also initialized a bias.

## 4) Compute the loss

See lines 71 -> 76 of training.py

## 5) Backpropagade the loss

It is done by the command **loss.backward().**

For the command to be computed correctly, all learnables must have requires\_grad = True, otherwise their value won’t be updater.

## 6) Update the learnables of the network

The optimizer chosen will then correct the learnables of the network , taking the gradient into account. **In our project, we use the Adam optimizer.** Pseudo code for the Adam optimizer is shown below . Command : optimizer.step().

