EE-559: Mini-project II

Pegolotti Luca - Martin Matthieu

May 15, 2018

1 Objective

The objective of this project is to design a mini "deep learning framework" using only tensor operations in pytorch and the standard math library, in particular without using autograd or the neural-network modules.

2 Code structure

The framework is composed by two modules: `modules`, `criterions` and `networks`. The classes implemented in each module interact as shown in Figure 1.

2.1 Modules

Modules implement some of the typical building blocks of a neural network. Each of these building blocks derives from the class `Module` (see Figure 1), the basic structure of which (except for the methods `resetGradient` and `updateParameters`) was suggested in the description of the project. The methods `forward` and `backward` implement a forward and backward pass respectively. The method `resetGradient` resets all the gradient tensors (if any) to zero, whereas the method `updateWeights` updates the weights (if any) of the module according to the learning rate `eta`.

All the modules deriving from `Module` posses a variable `input` taking the value of the input which was provided during the latest call to the `forward` method: this is necessary to compute the `backward` call. The classes which derive from `Module` are:

- `Linear`: given an input tensor $x \in \mathbb{R}^{n_{\text{s}} \times n_{\text{in}}}$, the output $y \in \mathbb{R}^{n_{\text{s}} \times n_{\text{out}}}$ of this module is given by $y_{ij} = \sum_k x_{ik} A_{kj} + b_j$, where $A \in \mathbb{R}^{n_{\text{out}} \times n_{\text{in}}}$ is the weight matrix, $b \in \mathbb{R}^{n_{\text{out}}}$ is the bias, and n_{in}, n_{out} and n_{s} are the number of input/output features and samples respectively. This module has the attributes `weight` and `bias`, which store the values of A and b respectively, and `weight_grad` and `bias_grad` which store the respective gradients.

- `ReLU`: this module applies the ReLU function to any given input tensor, namely, given $x \in \mathbb{R}^{n \times m}$, the output tensor $y \in \mathbb{R}^{n \times m}$ is defined by $y_{ij} = \text{ReLU}(x_{ij}) = \max\{0, x_{ij}\}$.

- `Tanh`: this module applies the tanh function to each component of the input tensor $x \in \mathbb{R}^{n \times m}$, i.e. the output tensor $y \in \mathbb{R}^{n \times m}$ is defined by $y_{ij} = \tanh(x_{ij})$.

2.2 Criterions

The `criterions` module contains the implementation of the loss functions. These share the interface given by the following base class `Loss` (see Figure 1). The method `function` computes the loss function of the `output` of a forward pass of a network with respect to the `expected`, and the `grad` method computes the gradient of the loss corresponding to the same parameters.

The available criterions are:

- `LossMSE`: Mean Square Error loss function. If $x \in \mathbb{R}^{N \times m}$ is the output of the forward pass and $y \in \mathbb{R}^{N \times m}$ is the expected output, the loss function is computed as $\text{MSE}(x, y) = \sum_{i,j} (x_{ij} - y_{ij})^2$ **we need to check this, in the solution of ex 3 they don't do the mean**.

- `LossCrossEntropy`: to be implemented?

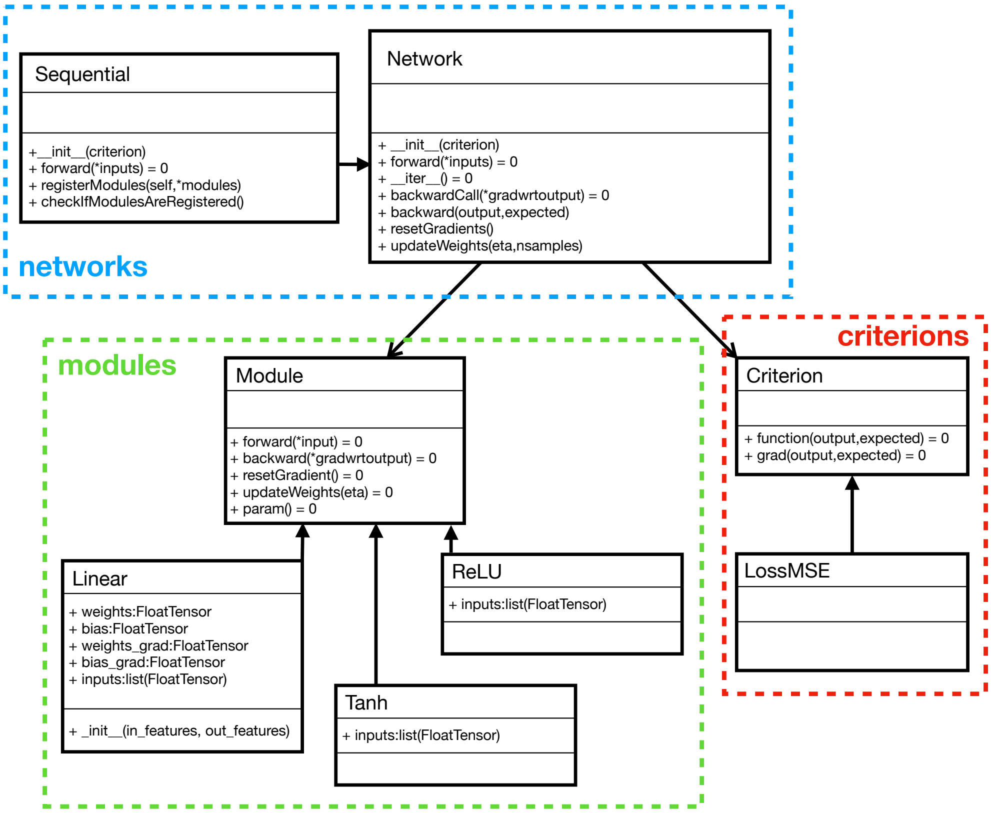

Figure 1: Class diagram of the framework. We indicate with the notation "= 0" the methods that need to be implemented by the derived classes, even though such methods are not actually abstract in a strict sense.

2.3 Networks

The `criterions` module provides generic structures for the implementation of neural networks. The base class is the `Network` class (see Figure 1). Each derived class must be iterable (namely, it must provide the abstract method `__iter__`) and must implement the `forward` and the `backward` methods, which depend on the topology of the underlying graph.

The available networks are:

- `Sequential`: template for neural networks: it provides a blueprint for derived classes implementing fully connected neural networks. Each graph must include a unique "source" (a node with only one input), a unique "sink" (a node with only one input) and nodes with exactly one input and one output. Each node is in fact an object derived from the `Module` class and must be defined as an attribute of a derived class in the constructor. Importantly, at the end of the constructor these objects must be registered via the method `registerModules(self,*modules)`, where the `*modules` must contain the modules in the order they appear in the network itself. Internally, the `registerModules` method builds a list of the modules, which provides a way to iterate over the elements composing the `Sequential` network. The forward and backward passes are executed in a straightforward manner by calling the `forward` and `backward` method of all the modules of the graph sequentially and propagating the result to the next or the previous node.

3 Test case

The structure of the test code, implementing a network with two input units, two output units, three hidden layers of 25 units is:

- Generate 1000 training sample, and 1000 testing points

- Normalize them with a zero mean and unit std

- Built a three hidden layer with linear neural network, and ReLU after each linear module "SimpleNet"

- Train the neural network using the 1000 training sample, for 1000 epochs and a constant learning rate $= 1e - 2$.

- Plot the training error and the testing error while training the network, and verify these results with the framework PyTorch.

The parameters of the sample length is hidden in the *mean* function, and we had to modify the eta in the final code: $eta = eta/nsample$.

```
class LossMSE(object):
    def function(self,output,expected):
        return torch.mean(torch.pow(expected - output,2))

    def grad(self,output,expected):
        return -2 * (expected - output)
```

The sequential class work as follow:

```
class Sequential(Module):
    def __init__(self,criterion):


    def registerModules(self,*modules):


    def checkIfModulesAreRegistered(self):


    def resetGradient(self):


    def updateParameters(self,eta,nsamples):


    def backward(self,*gradwrtoutput):


    def backwardPass(self, output, expected):
```

where the $registerModules$ needs to be called when we define a new network, in order to store the modules in a list. We will use the list ordered when we will call the methods $forward$, $backward$, and $update_Parameters$

References