EE-559: Mini-project II

Pegolotti Luca - Martin Matthieu [*]

May 18, 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 Numpy library, in particular without using autograd or the neural-network modules.

2 Code structure

The framework is composed by three modules: `modules`, `criterions` and `networks`, located in the framework/ folder in the root of the project. The classes implemented in each module interact as shown in Figure 1. A set of tests on these classes is implemented in the framework/testsuite.py script. The code has been tested with Python 3.6.5 and Pytorch version 0.4.0. We took inspiration from the Pytorch documentation[1] and from [1] for the implementation of the required functionalities.

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 $y_{ij} = \sum_k x_{ik} A_{jk} + b_j$, 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. In the backward pass, the module accepts as input the gradient of the loss function computed with respect to its output (this is computed by the nodes that immediately follows in the network) which we denote by $\partial\mathcal{L}/\partial y$. Then, the gradients of the loss with respect to the weights and bias are found as $\partial\mathcal{L}/\partial A = (\partial\mathcal{L}/\partial y)^T x$ and $\partial\mathcal{L}/\partial b = \sum_j (\partial\mathcal{L}/\partial y)_j$, where $\partial\mathcal{L}/\partial A \in \mathbb{R}^{n_\text{out} \times n_\text{in}}$, $\partial\mathcal{L}/\partial y \in \mathbb{R}^{n_\text{s} \times n_\text{out}}$, $\partial\mathcal{L}/\partial b \in \mathbb{R}^{n_\text{out}}$, and $(\partial\mathcal{L}/\partial y)_j$ denotes the j^th row of $\partial\mathcal{L}/\partial y$. The gradient of the loss with respect to the input of the module, which is to be passed to the previous node in the network, is found as $\partial\mathcal{L}/\partial x = \partial\mathcal{L}/\partial y A$.

- `ActivationFunction`: interface which provides a default implementation to the methods in `Module` that operate on or require internal weights or gradients. The backward pass of a generic activation function σ takes as argument the gradient of the loss function with respect to the output of the module, $\partial\mathcal{L}/\partial y$, and passes to the previous node the derivative of the loss function with respect to its input x, which is computed as $\partial\mathcal{L}/\partial x = \partial\mathcal{L}/\partial y \odot \sigma'(x)$. Classes which inherit from this template are:

[*] As agreed with Dr. F. Fleuret, L. Pegolotti has collaborated with group 78 for Project 1 and with group 79, with M. Martin, for Project 2. C. Bigoni and N. Ripamonti have worked together on both projects.

[1] `https://pytorch.org/docs/stable/index.html`

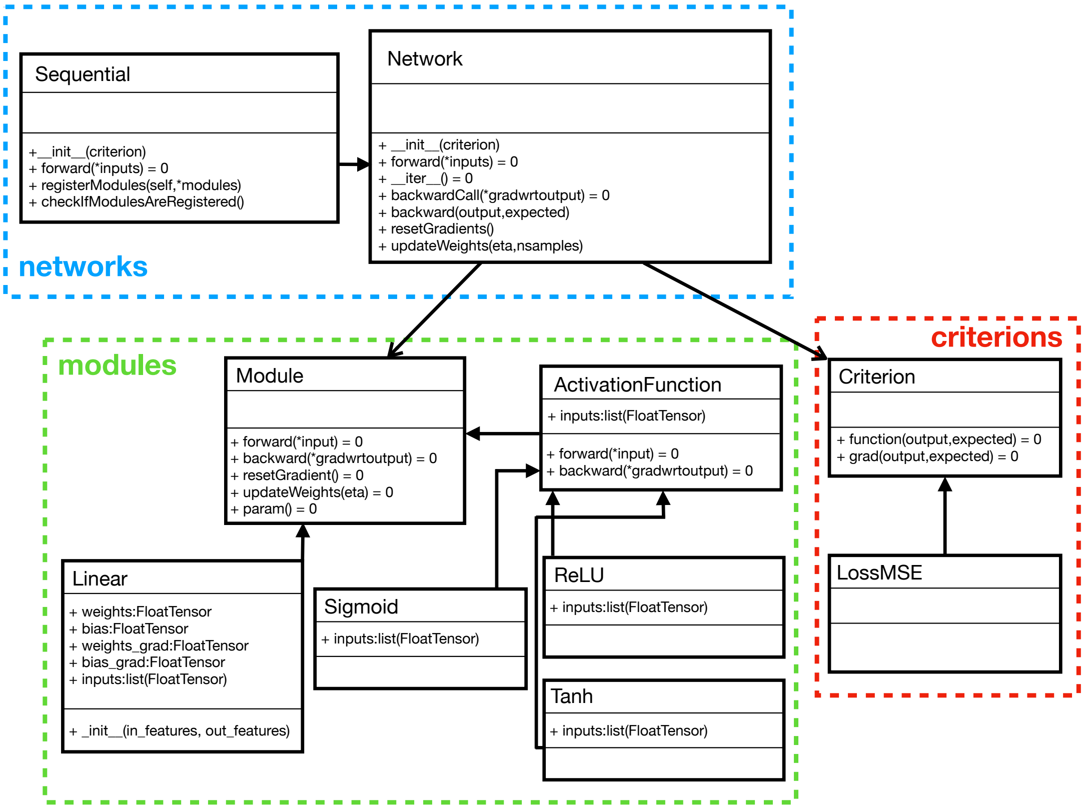
networks
Sequential
+__init__(criterion)
+ forward(*inputs) = 0
+ registerModules(self,*modules)
+ checkIfModulesAreRegistered()
Network
+ __init__(criterion)
+ forward(*inputs) = 0
+ __iter__() = 0
+ backwardCall(*gradwrtoutput) = 0
+ backward(output,expected)
+ resetGradients()
+ updateWeights(eta,nsamples)
modules
Module
+ forward(*input) = 0
+ backward(*gradwrtoutput) = 0
+ resetGradient() = 0
+ updateWeights(eta) = 0
+ param() = 0
ActivationFunction
+ inputs:list(FloatTensor)
+ forward(*input) = 0
+ backward(*gradwrtoutput) = 0
Linear
+ weights:FloatTensor
+ bias:FloatTensor
+ weights_grad:FloatTensor
+ bias_grad:FloatTensor
+ inputs:list(FloatTensor)
+ _init__(in_features, out_features)
Sigmoid
+ inputs:list(FloatTensor)
ReLU
+ inputs:list(FloatTensor)
Tanh
+ inputs:list(FloatTensor)
criterions
Criterion
+ function(output,expected) = 0
+ grad(output,expected) = 0
LossMSE

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.

◇ `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 $y_{ij} = \mathrm{ReLU}(x_{ij}) = \max\{0, x_{ij}\}$.

◇ `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 $y_{ij} = \tanh(x_{ij})$, where

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}.$$

◇ `Sigmoid`: this module applies the sigmoid function to the given input tensor. If $x \in \mathbb{R}^{n\times m}$ is the input tensor, then $y_{ij} = \mathrm{sigmoid}(x_{ij})$ and $y \in \mathbb{R}^{n\times m}$. The sigmoid function is defined as

$$\mathrm{sigmoid}(x) = \frac{e^x}{e^x + 1}.$$

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 target (`expected`), and the `grad` method computes the gradient of the loss corresponding to the same parameters.

The only available loss function is the following:

- `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

$$\mathrm{MSE}(x, y) = \frac{1}{N}\sum_{i,j}(x_{ij} - y_{ij})^2,$$

  where N is the total number of entries of the tensors x and y.

	Full gradient		SGD	
	Our framework	Pytorch	Our framework	Pytorch
min	0.7%	0.7%	0.6%	0.6%
median	1.65%	1.65%	0.95%	1.4%
mean	1.67%	1.63%	1.01%	1.32%
max	2.8%	3.1%	1.6%	2.1%

Table 1: Test error rates on 20 i.i.d. realizations of the generated points, both using the standard gradient descent algorithm and the SGD algorithms.

2.3 Networks

The `network` module provides a generic structure 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 only available network is the following:

- `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 (the gradient to be injected in the last node of the network is the gradient of the loss functions with respect to the output of the forward pass).

3 Test case

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

- Generate 1000 training sample, and 1000 testing points.
- Normalize them with to have a zero mean and unit standard deviation.
- Define a network `SimpleNet` (inheriting from `Sequential`) which is composed by two `Linear` modules followed by a `ReLU` and a `Tanh` modules respectively, and a final `Linear` module.
- Train the neural network using the training sample, for 1000 epochs and a constant learning rate $\eta = $ 1e-2.
- Plot the loss function and the train error during the stochastic gradient descent (SGD).
- Finally, print the train and test errors.

We compare the train and test error obtained with our framework and the ones obtained with the same neural network in Pytorch by using the same configuration, i.e. we consider the same epochs, learning rate and initialization weights (which are drawn from normal distributions with zero mean and unitary standard deviation). The results for the test error are presented in Table 1 when using gradient descent with no mini batches (i.e. mini batch size equals to the number of points in the train sample) and when using the SGD.

4 Conclusion

We have presented a simplified framework for neural networks that allows to easily concatenate building blocks (modules) sequentially. The simple numerical test we implemented shows that the stochastic gradient descent converges and that the errors we obtain on train and test data-sets are small and comparable to the ones found when solving the same problem with Pytorch.

With the goal of imitating the user-friendly interface of Pytorch, we came up with a solution to generate customized neural networks that require the definition of the modules inside the constructor of the `Network` subclass and the implementation of a `forward` method. We believe that the structure of the code is well-suited to be extended to other modules (e.g. convolutional layers, pooling . . .) or to account for more complicated networks. However, our solution consists in internally storing the modules into a list, and therefore does not account for situations in which some nodes require multiple inputs/outputs. The major obstacle to the generalization of the approach to more complicated networks lies in the auto-generation of the underlying graph. A possible way to overcome this issue, in our opinion, could consist in introducing a wrapper for the tensor passed along the graph (similarly to what is done in Pytorch, with the Variable class) in which also the information of the latest visited node is stored. This could allow to build the graph “on the fly” during the forward pass and to propagate the information backward in the back propagation algorithm.

References

[1] I. Goodfellow, Y. Bengio, A. Courville, and Y. Bengio. *Deep learning*, volume 1. MIT press Cambridge, 2016.