Deep Learning: Mini Project 2
deep learning framework

Pratibha Bhatt, Quentin Desaulty, Axel Manguy

May 17, 2018

1 Introduction

In terms of understanding of neural networks, the basic use of pytorch modules is more close to a black box model than a theoretical implementation. In order to better understand the underlying concepts of deep learning architectures, building our own framework is expected to be fruitful. Especially concerning the connections and interactions between modules as well as the backpropagation mechanisms, the difficulties inherent to the implementation of these concepts is expected to produce a deeper understanding of neural network mechanisms.

2 Architecture description

Before starting any implementation, we need to have a general idea of the framework structure we're going to implement. Since all different parts of the framework will have to interact together on the same basis, the class Module intends to offer a flexible abstract class for all other modules. Since all layers or sequences of layers will inherit from that class, they share a list of parameter and a dictionary of modules. All descendants will have to implement the two methods forward and backward. Using different implementation, this structure allows propagating the gradient computation in a straightforward and flexible way. The activation function is considered as a layer just as the standard linear layer but with a different implementation of the forward and backward method. Figure 1 shows the UML diagram of our architecture.

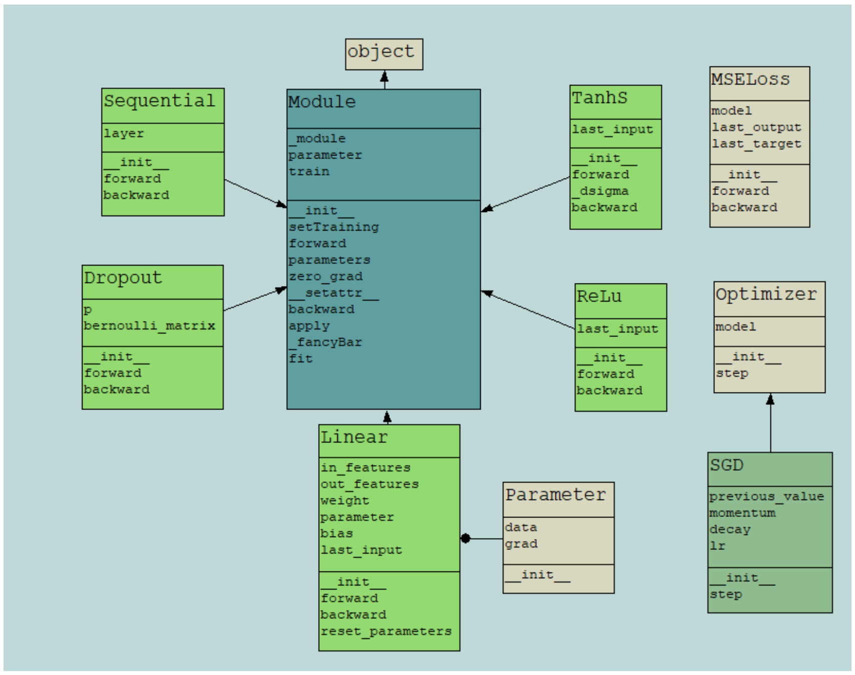

Figure 1: UML framework diagram

3 Neural Networks modules and Layers

The user is free to make the sequence of module he wants. The *backward* and *forward* methods of each module were made so, they are compatible with whatever module is put after. Note that, in case of a custom module, the user has to reimplement these methods for his module, as our achitecture does not have autograd like Pytorch. However an easy way to build a sequence of layer is to use *Sequential* that is describe below. In this case, the user doesn't have to reimplement neither *backward* nor *forward* method.

When making a custom model, modules that are used in the model are automatically recognized and added to the list of inner-modules. Thus, when the method *parameters* is called, the model return the parameters of all modules it is composed of. An example of a custom model can be seen in the test file.

Note that, to easily train his model, an user can call the *fit* method of its model instance. This idea came from the Keras deep-learning library. It appears to us as an useful method, as it is user-friendly and thus we decide to reproduce it in our architecture.

3.1 Linear Layer

The linear layer class implements one of the most simple types of neural networks layer. the output of that layer is defined by $output = weight.input + bias$ The bias here is optional as the final user may not want to use them. In that case, the output is simply $weight.input$. the parameters the model will modify during training are the weights themselves, so why they are stored in the parameter instance variable. The backward pass is implemented in a way that supports batch.

3.2 Sequential

In order to get the possibility to aggregate several layers together, the class Sequential is constructed as a sequence of submodules. Using the legacy from the Module class, Sequential adds every layer in the order passed in the parameters of the method. A list as instance variable stores the order of submodules, this way modules are directly linked in the right order. The forward and backward is then nothing more than a call of the forward and backward methods for each submodules (using the list previously instantiated)

3.3 Dropout

As an optional module, not requested in the subject, the dropout layer has been implemented for test purpose and to improve the overall performance of custom models. In terms of implementation, the dropout follows a Bernoulli law with a probability p. the user can define p if the by default 0.5 probability needs to be modified.

4 Activation layers

4.1 TanH

The TanH layer implements the hyperbolic tangent activation function as $Tanh(x) = \tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$. The backward passes implement the derivative of tanh : $tanh'(x) = \frac{4}{(e^{-x} + e^x)^2}$

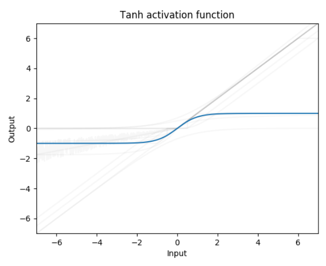

Figure 2: Tanh Activation Function

4.2 ReLu

We used ReLu as activation function in the implementation. The ReLu function is define as follow :

$$ReLu(x) = \begin{cases} x & \text{if } x \in [0, +\infty[\\ 0 & \text{else.} \end{cases}$$

and thus

$$ReLu'(x) = \begin{cases} 1 & \text{if } x \in]0, +\infty[\\ 0 & \text{else.} \end{cases}$$

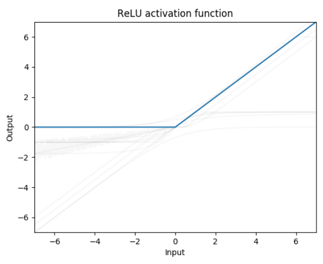

Figure 3: ReLU Activation Function

5 Loss and Gradients

Unlike Pytorch, in our architecture, the Loss function (implemented in our code as *MSELoss* class) is directly linked to a model. When the user want to proceed to the *backward* of a model, he have to call the *backward* method of an *MSELoss* instance. When a user want to compute the loss of an input, according to a target, he has to call te *forward* method of an *MSELoss* instance (that is linked to his model). These differences are mainly due to the fact that Pytorch uses autograd, and thus directly apply the *backward* function to an output.

The way of using *SGD* is pretty much the same than in Pytorch. The user has to specify his model, and the **lr** (learning rate), **decay** and **momentum** parameters he wants to train his model with. To apply one step of optimizer, the user just has to call the method *step* of *SGD*.

5.1 Mean Square Error

Mean Square Error was used as the loss function. The expression of mean square error is

$$l(x, y) = \frac{1}{n} \sum_{k=1}^{n} (x_k - y_k)^2$$

and thus

$$\frac{\partial l}{\partial x_k} = \frac{2}{n}(x_k - y_k)$$

where x is the output and y the target.
For performance purpose the implementation is based on the pytorch tensorial operations.

5.2 Stochastic Gradient Descent

Considering we want to minimize the error, the stochastic gradient descent moves in the decreasing gradient direction, following the iteration scheme :

$$w_{i+1} = w_i - \eta \frac{\partial l}{\partial w_i}$$

where η defines the learning rate, and l the loss function.

6 Pytorch Comparison

A good implementation should not be only user-friendly, it should have reasonable performance too. Let us see the performance of our architecture.

6.1 Time Performance

The time computation for each test epoch is plotted in Figure 4. On average our custom model is better than pytorch by 11 ms in terms of computation time.

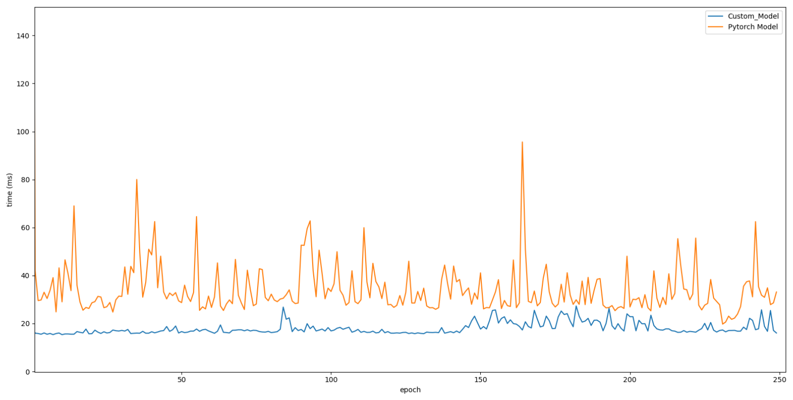

Figure 4: Time per epoch

To explain these differences one must keep in mind that Pytorch has a much more complex architecture that implements numerous other modules it has to be compatible with. This search of compatibility imposes to Pytorch the need to check conditions and interact with other modules and libraries. All these operations and complexity takes time and explains in some way why our implementation performs better.

If we were searching for an even better time performance, a common practice is to import and use libraries built in C. This way the time-consuming task will be performed in a closer to machine language and thus improve optimization and performance. However, since the main objective is to build the entire mini-framework in python, the better we can do is to use tensorial operations as much as possible. Especially in the forward and backward modules functions, the for loops have been avoided whenever it was possible. The use of yield function in parameter allows saving computation time since it will only build iterator when asked. All these small improvements help to reduce the computation time and improve the overall performance of the custom model.

6.2 Test on toy dataset

To be able to test and compare our mini framework with a Pytorch implementation, we built a test procedure based on the recommendation of the subjects and taking our implementation into account. The results of that test procedure are provided in Figure 5, 6 and 7. Concerning the test dataset, it is

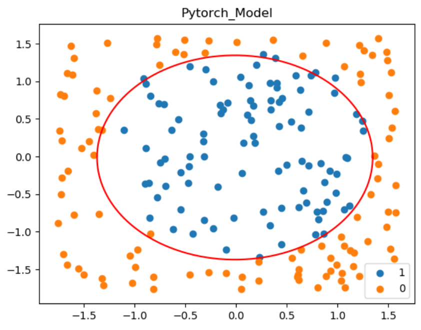

Figure 5: Pytorch model classification

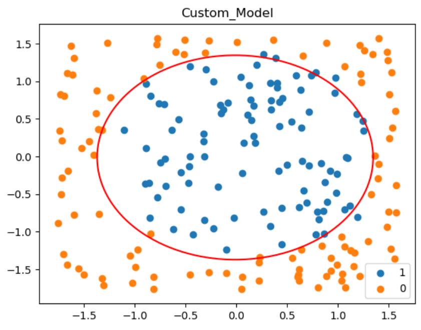

Figure 6: Custom model classification

built so it follows a random uniform distribution between 0 and 1. The ones labels are then located inside a disk centered on (0.5,0.5) with a radius of $\frac{1}{\sqrt{2*\pi}}$. This disk is plotted in red in Figure 5 and 6. While there are still some mis-classifications on the borders of the disk, the overall performance is quite good especially when a dropout layer is implemented. Finally, custom and Pytorch model behave the same way and both perform well with less than 0.5% of errors.

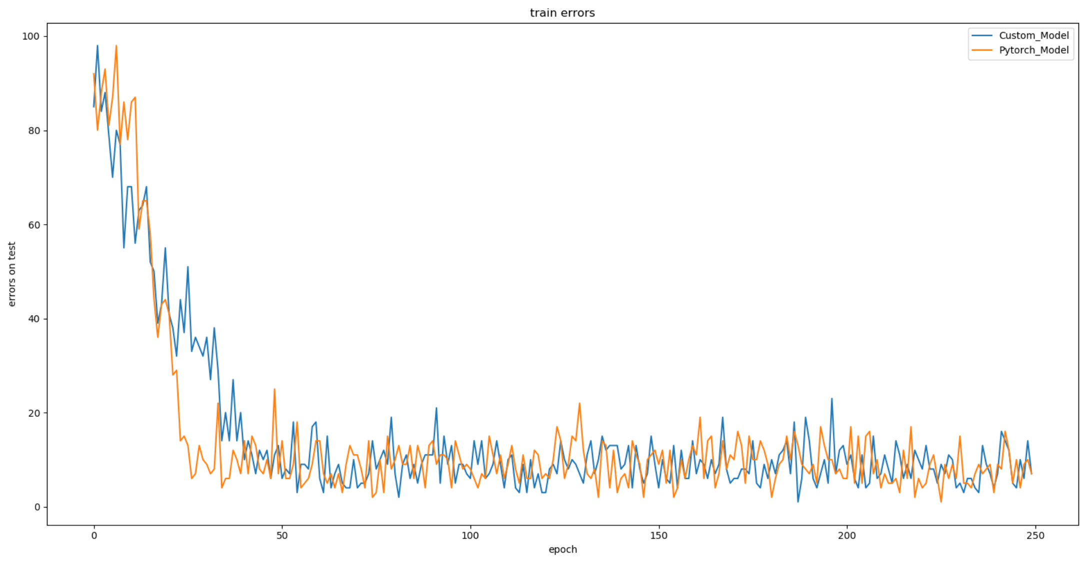

Figure 7: Errors on test

7 Conclusion

All along the building process of that mini framework, we learned how to implement and understand the most basic key concepts of a deep learning architecture. The forward and backward implementation, as well as the gradient back-propagation mechanism, led us to reshape our way of thinking tensorial operations. The search for time performance and the comparison with Pytorch model has been a fruitful process in terms of deep learning understanding. Finally and thanks to the better understanding of numerous key concepts we have been able to design our own implementation of less basic modules such as the dropout layer.