Deep Learning (EE553) - Project 2: Mini deep-learning framework

Hugo Grall Lucas, Théophile Schenker and Costa Georgantas
Deep Learning EE553, EPFL, Switzerland

***Abstract*— *PyTorch* is a widely used Python deep learning framework, including many essential features for handling Deep Neural Networks (DNN). In this project, an alternative custom framework is created from scratch. Linear layers and different activation functions are implemented, as well as necessary methods to create and train a network. The performance is compared to what is obtained with *PyTorch* functions on a simple problem. For the tested parameters, the results show a quicker convergence for the custom framework in terms of epochs (while *PyTorch* version runs much faster in time). An hypothesis is constructed to explain this behavior.**

I. INTRODUCTION

The aim of project 2 is to exercise several of the subjects introduced during the lectures of the *Deep Learning (EE553)* course in a practical way, by implementing a simple deep learning framework in Python with only *PyTorch* tensor operations and the basic math library as a starting point.
The work is split in two phases. The first one is the implementation of the framework, for which various necessary tools were coded: Activation functions, Fully connected layers, Standard forward and backward steps and training with a stochastic gradient descent (SGD), minimizing mean square error loss (MSELoss). To do so, characteristics of object oriented programming are of great help, enabling an abstraction level suitable for different models.
In the second phase, a simple toy data set is used to test and evaluate the overall performances of the custom framework against the *PyTorch* one.

II. MINI DEEP-LEARNING FRAMEWORK

A. Framework architecture

The different steps of a model are represented by modules which are stored in a list when initializing the whole model. Operations can be executed in the reverse or direct order of the list in order to be able to perform both forward and backward steps. Each module has $forward$ and $backward$ methods which are called during the forward and backward passes. The loss is also implemented as a class, with $forward$ and $backward$ methods. A Sequential class takes the list of modules and manages the relation between them.

B. Modules

TABLE I
IMPLEMENTED FUNCTIONALS.

Sigmoid	Standard sigmoid function
TanH	Hyperbolic tangent
ReLU	Rectified Linear Unit
Linear	Fully connected layer

Every one out of these modules has the following methods:

- $forward$ (see Section II-B.1)
- $backward$ (see Section II-B.2)
- $gradient\_step$ (see Section II-B.3)
- $zero\_grad$ (resets the derivatives to zero)
- $param$ (returns all the instance parameters)

Additionally, each functional has different attributes. Especially, the Linear class stores the derivative of the loss, that is updated each time the $backward$ method is called. It also stores the weights and bias, which are updated when the $gradient\_step$ method is called. Some functionals also need custom initialization, like the Linear class that randomly generates the weight matrix.

1) Forward: The $forward$ method takes a single input x and returns the value of the activation function or the result of the matrix multiplication for Linear layers.

2) Backward: The $backward$ method takes the result of the forward pass of the current module and the derivative computed during the backward step of the next module (with respect to the forward pass way). It returns the derivative of the function with respect to the loss. If the target module is a Linear layer, the backward method also updates the derivatives of the weights and bias. Additionally, to be able to perform the gradient step, these parameter derivatives are accumulated and stored.

3) Gradient step: The gradient step is only active in the case of Linear layers, for which it updates the weights according to a learning rate using the previously stored derivatives.

C. Sequential

The sequential class takes a list of modules as input. It then calls the different methods of each module to perform the different tasks. For instance, the $forward\_pass$ is performed by calling the $forward$ method for each module in the list in order, and returning all the outputs in a list.
To perform the $backward\_pass$, the forward methods of the modules are first called, before the result list is run through

in reverse to propagate the gradient of the loss using the backward method of each module. The loss is computed using a specific class.
The custom implementation does not take advantage of batch processing for improving the computation time either during the forward or backward pass. As a result, each sample pass is processed individually. However, the SGD is computed on each mini-batch of the training set.

III. RESULTS: COMPARISON WITH *PYTORCH* FRAMEWORK

As explained in the guidelines, both frameworks were tested with a simple model composed of four linear layers of size 25, followed by an output layer of size 2. Additionally, ReLU was chosen as the activation function between each layer. Table II provides the set of parameters used in both DNN.

TABLE II
SET OF PARAMETERS USED BY BOTH DNN

Epochs	η	**Mini batch size**	**Sample number**
20	0.1	100	1000

A. Dataset

A 2D toy data set with two classes is used. The samples are generated following a uniform random distribution in the domain $[-1, 1] \times [-1, 1]$. Then, each point is attributed to a class depending on its position inside - class 1 - or outside - class 0 - the circle centered in [0,0] with $\frac{2}{\sqrt{\pi}}$ as a radius. One-hot encoding is used. Furthermore, the two classes are well balanced in term of distribution thanks to the choice of the circle radius. The dataset is illustrated on Figure 1.

B. Results of framework comparison

The summary of the results during train and test phases for both models – i.e. the DNN using the custom framework and the one using *PyTorch* – are provided on Figure 2.

By looking at the plot for the loss with respect to the different epochs, three main points are visible. The first one is that *PyTorch* appears to perform the minimization of the loss slower than the proposed framework. An hypothesis may come from the difference during the MSELoss computation, which is further developed in Section IV. The second observation concerns the variance which is very small for the custom implementation and quite larger in the *PyTorch* framework. Finally, the *PyTorch* DNN reaches a plateau after only about 3 epochs. The model seems to be stuck in a local minimum. A possible improvement would be changing the optimizer parameters such as learning rate and mini-batch size or testing another optimizer.

On the error rates plots, the overall performance seems to be better for the custom DNN than the *PyTorch* one as well. However, at the very end of the train, a larger slope is observed for the *PyTorch* curves than for the custom implementation. Increasing the number of epochs would probably further improve the final *PyTorch* classifier results. The results for the test and the train are very similar. An explanation for this observation is that the toy dataset remains quite simple and well balanced.

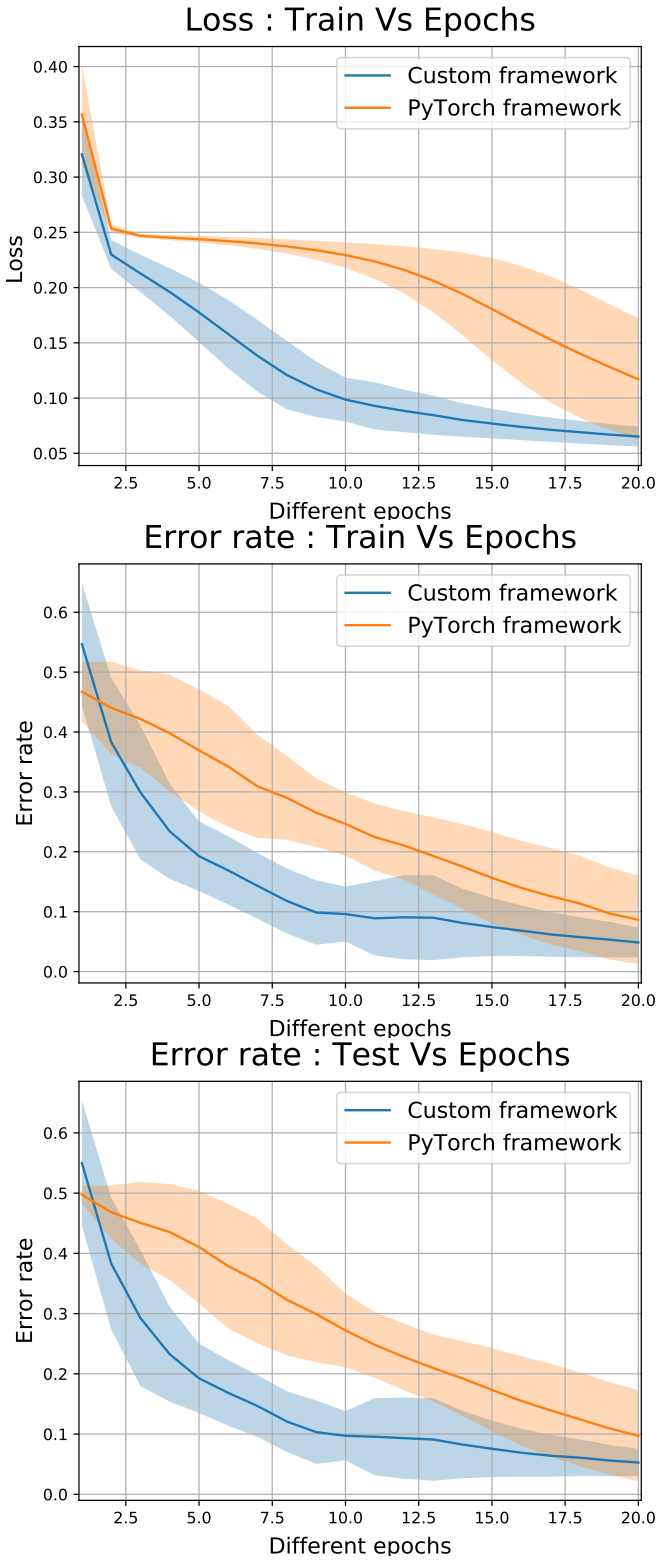

Fig. 2. Classification results for train and test phases. Each experiment is performed 10 times with different folds. The resulting mean and standard deviation are shown in the three plots. **[Top] :** Losses obtained during the train phase of both DNN. **[Middle] :** Error rate evolution during the train phase for both DNN. **[Bottom] :** Error rate evolution during the test phase for both DNN.

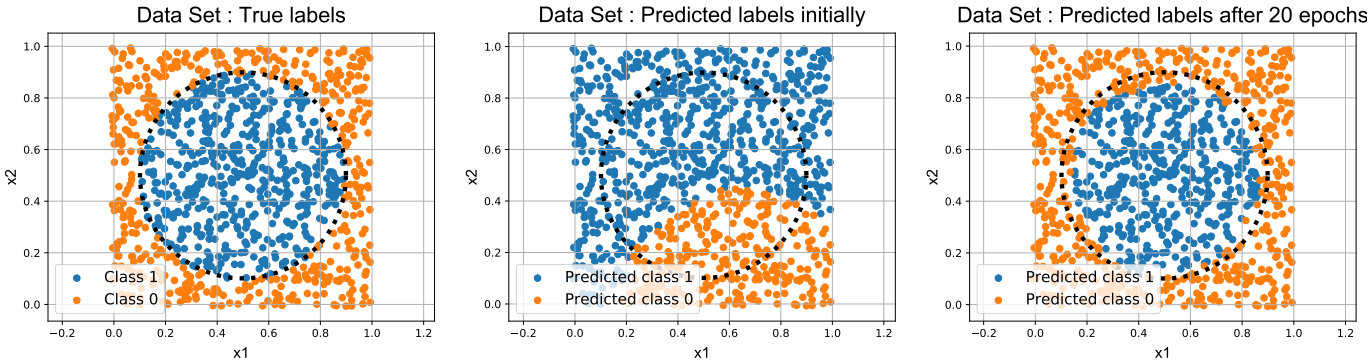

Fig. 1. Representation of the toy dataset, recentered in [0,1]x[0,1]. The black dashed line marks the border between both true classes.**[Left] :** Example of randomly generated data points used as test samples. **[Middle] :** Result of the test samples classification obtained with the untrained custom DNN (random weights). **[Right] :** Result of the classification after 20 epochs with the custom framework.

TABLE III

SUMMARY OF THE RESULTS.

Model	Error rate mean [%]	Error rate std [%]
Custom Framework, train	4.33	1.83
Custom Framework, test	4.88	2.23
PyTorch, train	10.17	8.88
PyTorch, test	16.59	14.13

C. Running the code

The code for this experiment is included in the *project2* folder. To run this code, simply type *python3 test.py* (provided python is installed) in the command window and the two frameworks will be trained with the already described setup. The *project2.py* and *project2_withAutoGrad.py* scripts can also be run independently.

IV. DISCUSSION

Despite using the same model and learning rate of the SGD, it is apparent that the output of both implementations differ importantly. This can be partly explained by the way the derivatives of the weights are updated in each implementation. In our framework, they are updated each time $backward\_pass$ is called (at each sample) whereas in *PyTorch*, the derivatives are updated once for every mini-batch. This change in the structure of the backward pass might explain why the error rates and loss vary between the two frameworks. As a consequence, the optimal learning rate for the two implementations also differs. Indeed a faster convergence was observed when using a larger learning rate for the *PyTorch* framework. It is also worth noting that *PyTorch* is a lot faster, as it relies less on loops and takes full advantage of the batch processing provided by the tensor operations.

Another factor that might have impacted the results is the way the weights were initialized. Simply using randomly generated weights usually results in an unsuccessful training. In the custom framework, Xavier initialization is used, meaning that the weights are randomly generated following a Gaussian distribution centered at zero and with a standard deviation of $\sigma_X = \sqrt{\frac{2}{l^{[n-1]}+l^{[n]}}}$, where $l^{[n]}$ denotes the size of the layer n [1]. *PyTorch* initialization uses a uniform distribution in the range $[-\sigma_u, \sigma_u]$, where $\sigma_u = \frac{1}{\sqrt{l^{[n-1]}}}$ [2]. Our initialization method guarantees that the activation value does not statistically vary between the layers, so that the gradient does not vanish. This should however not be a problem in this test as ReLU was chosen as the activation function [3].

V. CONCLUSION

The main goal of this work was to develop a custom deep learning framework without the use of any advanced library. To do so, a set of objectives were defined in the guidelines, such as:

1) Ability to build networks combining fully connected layers, TanH and ReLU
2) Computation of forward and backward passes
3) Optimization of the parameters using SGD with MSE loss

The proposed solution fulfills all the requirements listed above. Moreover, the discussed comparison between the custom DNN and the *PyTorch* one offers an understanding of some differences between both frameworks. Finally, this practical exercise was a precious learning experience because it pushed us to entirely understand the mechanism behind each main step of a simple deep learning framework, demystifying it.

REFERENCES

[1] Glorot, X., & Bengio, Y. (2018). Understanding the difficulty of training deep feedforward neural networks. 2010. Received February, 12.
[2] https://github.com/PyTorch/PyTorch/blob/master/torch/nn/modules/linear.py
[3] Glorot, X., Bordes, A., & Bengio, Y. (2011, June). Deep sparse rectifier neural networks. In Proceedings of the fourteenth international conference on artificial intelligence and statistics (pp. 315-323).