Deep Learning Framework Design based on Pytorch Linear Algebra Toolkit

Raphaël Reis Nunes, Sebastien Emery, Vincent Bernaert

Abstract— **The aim of this project is to design a deep learning framework from scratch based on the *Pytorch* linear algebra toolkit. To assess the efficiency of our framework we compare it with the same models from *Pytorch* modules. The results shows a good capability of our framework to learn a non-linear discrimination function with a limited amount of data turn over.**

I. INTRODUCTION

In the past decade, many libraries came out to satisfy the increasing demand for Deep Learning (DL) tools. These libraries simplified the DL approach for numerous academia and industry actors. To understand the principles making the libraries as *Pytorch*, we reverse engineered the implementation to built our own design.

In this project we solved a basic classification task. The model we built are flexible neural networks models with backpropagation feature. To implement the framework we based the linear algebra on the *torch.empty* tensor constructor and no other *torch* functions. We however used tensor methods.

II. METHOD

A. Classification task: dot in the quarter circle

The dataset we are aiming at classifying are uniformly distributed points in the 2D square $[0, 1]^2$. If the distance from the origin $(0, 0)$ to the point is lesser or equal to $\frac{1}{\sqrt{2\pi}}$, the point label is 1 and otherwise the label is 0. In other words, all the points in the quarter circle are labeled 1 and those outside are 0. The function that our model must approximate is a non-linear one. To solve such a task, deep networks (DN) are particularly suitable. Indeed, they are able to approximate any function[1]. Other classification algorithms would also work as Gaussian/RBF kernel support vector machine or k-nearest-neighbours.

B. Framework design

1) Modules: For the implementation we organised the object oriented design after the *Pytorch* model. Thus, every part of the framework except the utils functions inherit from a self-made *Module* parent class. The *Module* parent class has 3 methods: *forward*, *backward* and *param* where the first two are the functions for the backpropagation algorithm and the param function returns the parameters of the model.

2) Feedforward layer: Since this class inherit from the Module class, we have to implement the forward and the backward pass. The weights are initialized with *Kaming He* method. The forward pass consists in simple matrix multiplication between the input data and the weights. If bias is required, we add it to get the output of the layer. The backward pass is trickier to implement: it is a back propagation of the output loss. This class compute equations 4 and 3 and the left-hand side of equation 2 There is also a param method to send the values of the parameters (weights and bias) and of their gradient. The Feedforard class has also a zero_grad method whose role is to set every gradient value to 0.

3) Activation functions: This classes also implement *Module*. There are 3 of them: ReLU, Sigmoid, Tanh. Each one has two attributes; one for the function and one for its derivative. They also implement the forward and the backward pass. This class compute the right-had side of the equations 1 and 2.

$$\delta^L = \nabla_a L \sigma'(z^l) \tag{1}$$

$$\delta^l = ((w^{l+1})^T \delta^l) \odot \sigma'(z^l) \tag{2}$$

$$\frac{\partial L}{\partial b^l} = \delta^l \tag{3}$$

$$\frac{\partial L}{\partial w^{l+1}} = a^l \delta^{l+1} \tag{4}$$

4) optimizer: We implemented a stochastic gradient descent optimizer whose role is to apply a gradient descent optimization and update the weights after each sample. The reason for this is to decrease the loss value until reaching its minimum. In this project, we used mean squared error (MSE) as loss function. Since it is a convex function, if the algorithm converge to a minimal, we are assured to find the global minimum.

5) Dropout:

6) Sequential Module: The sequential module aggregate all the previously described modules with to its add() method. Its aim is to abstract the whole learning process of the deep network (DN). Since it's a module it also implement forward and backward passes. The forward pass call the forward pass method of all the layer and activation functions modules. Same for the backward pass. The sequential module is also able to set all the gradients to zero with the *zero_grad* function.

7) Side classes/functions: Beside the module sub-classes, we implemented several utils class and functions. The most important one is the MSE class . This class has 2 attributes: the MSE function (see 5) and the MSE derivative (see 6). We use the loss function to evaluate the model's learning capacities. We use the derivative in the backpropagation algorithm to propagate the error back to the network and calculate the parameters' gradient.

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(\hat{y} - y)^2 \tag{5}$$

$$\frac{\partial MSE}{\partial a} = 2\sum_{i=1}^{n}(\hat{y} - y) \tag{6}$$

8) Test framework: To assess the performances of our design, we recorded the accuracy and loss of the different models (Table II) throughout the training process. We validated it on the test set. We then compare the loss and accuracy curves from our framework with the equivalent model build with *Pytorch*. To initialize the weights of both models, we used a kaiming uniform method. We also built a unit test suit to validate the main parts of our framework, mainly: forward pass, backward pass, backpropagation and zero_grad.

TABLE I: Models Architecture

Model	Architecture	Activation	Last layer activation
model0	2 layers, 25 neurons	Tanh	Tanh
model1	3 layers, 25 neurons	Tanh	Tanh
model2	3 layers, 25 neurons	ReLU	Tanh

III. RESULTS

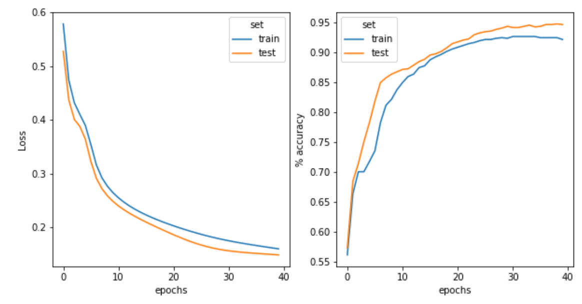
After 40 epochs, see the results on Table II. Model 1 is the most performent model and shows a train loss of 0.035, train accuracy of 0.986. The models performs slightly worse on the test set with an accurcy at 0.98. The same model in the *Pytorch* framework shows similar performance with a train loss at 0.021 and a train accurcy at 0.969 and for test loss is 0.017 and accuracy is 0.979.

TABLE II: Models performances

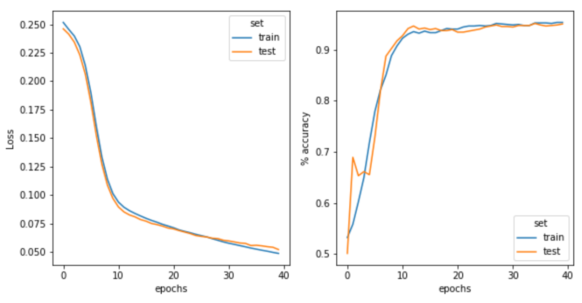
Framework	Self-made			
	Tr. Loss	Tr. acc.	Te. Loss	Te. acc
model0	0.19	0.912	0.17	0.953
model1	0.17	0.922	0.16	0.963
model2	0.39	0.734	0.42	0.708
	Pytorch			
	Tr. Loss	Tr. acc.	Te. Loss	Te. acc
model0	0.08	0.937	0.080	0.953
model1	0.07	0.995	0.071	0.952
model2	0.04	0.95	0.05	0.933

The learning curves show strong similarity of the apprentice capabilities between self-made and *Pytorch* model (Fig. 1,). Even if the pytorch loss is slightly lower that the self-made loss, we see they decrease at the same pace and are almost null after 40 epochs. For the accuracy, there is a rapid increase at the beginning and a similar shape for both. Note the common behaviour of the test learning curve. The reason that the learning curve of the test set is highly similar to the train set is probably due to the data building method. Indeed, we draw sample from the same uniform density function and since the neural network is learning the train distribution, it has no difficulty to generalize for the test distribution.

Figure 2 shows the evolution of the accuracy for the model architecture with 3 layers. We note that the *Pytorch* long term accuracy is better than ours. At the very beginning, our model learns faster than the *Pytorch* one. Overall, ReLU activation shows a faster learning rate and a better accuracy



(a) Self-made framework



(b) Pytorch

Fig. 1: Learning curves with model 1

than tanh. The ReLU activation we implemented is not efficient at learning the non-linear function. Indeed, it reach its plateau really fast and at low accuracy and does not increase afterwards.

IV. DISCUSSION

Overall, our DL framework implementation is working fine in terms of computation time, accuracy and loss performances. Of course, it is not as robust as *Pytorch* framework. Given a relevant parameter initialization, the loss behavior is coherent with the theory and proves that our parameter's update works as expected. This shows that our DN is able to learn form the data and generalize enough to give sound prediction.

The main issue with our implementation is the ReLU activation function learning process which is not as efficient it should be. Even when testing different initialization methods, a decent maximum is never reached. Even if ReLU is well known for its capacity to avoid gradient vanishing problem, it can be quite difficult to make an efficient learning with it. One potential solution would be to increase the size of the dataset or building a Adam optimizer which include momentum and decay in the learning process.

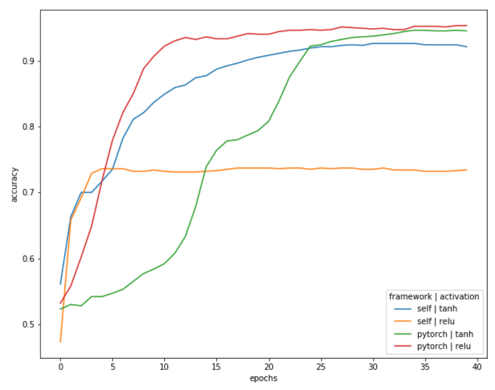


Fig. 2: Accuracy evolution base on activation

In further work, we would KFold the learning processes to build more robust learning curves and statistics. This indeed allows to compute the mean of the model prediction and loss and have standard deviation to handle uncertainties and noise.

V. CONCLUSIONS

In this work we built a framework whose design is inspired by the *Pytorch* python library. We implemented several modules to abstract the learning process of a DN. Overall, our design is working good. Of course, it shows some weaknesses but it is able to learn from a simple data and solve a non-linear classification problem as the theory predicts it.

REFERENCES

[1] Moshe Leshno et al. "Multilayer feedforward networks with a nonpolynomial activation function can approximate any function". en. In: *Neural Networks* 6.6 (Jan. 1993), pp. 861–867. ISSN: 08936080. DOI: 10.1016/S0893-6080(05)80131-5. URL: https://linkinghub.elsevier.com/retrieve/pii/S0893608005801315 (visited on 05/20/2020).