Deep Learning framework implemented with Python

Chada Ben Nejma Marwen Mokni Léonard Pasi

EE-559: Deep Learning

***Abstract*—Deep learning implementations are gaining popularity in handling data and extracting useful features from the data. The frameworks are useful tools to have a proper implementation of Neural Nets insuring a very easy to use and clear structure code. For this purpose, we focus on implementing our own framework to test it on an actual dataset and compare it with the most used frameworks, like Pytorch.**

I. Introduction

In the past decade, deep learning has become a critical component of the AI industry. Naturally, deep learning frameworks that enable to build models more easily and quickly, without getting into the details of underlying algorithms, have gained in popularity. In this project we designed our own mini deep learning framework using only PyTorch's tensor operations and the standard math library. The report is organized as follows: first, we describe our framework following a top-down approach; and second, we demonstrate its performances by testing it on the dataset described in the specifications.

II. The framework

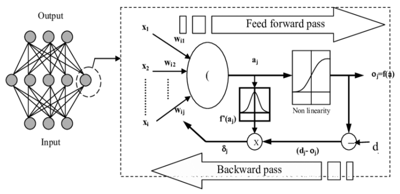

Fig. 1: Schematic representation of forward and backward mechanism

A multilayer perceptron (MLP) is usually defined as a sequence of at least three multidimensional perceptrons, each constituting a layer of the MLP. They are commonly used for classification. Training consists in optimizing the parameters of each layer through an optimization algorithm, such as gradient descent, to minimize a loss function, such as the mean squared error (MSE). It is therefore an iterative procedure: at each step, a forward pass is performed to compute both the output and all the internal variables of the MLP, and a backward pass is executed to compute the gradient of the loss with respect to all the parameters to be optimized. Figure. 1 shows a visual representation of these steps.

Each layer of the MLP is essentially an object, that has internal values and that needs to perform certain functions. Hence, the use of classes to represent elements of the MLP in our Python framework comes as a natural choice.
Our framework is organized in three modules: Model_NN.py and Activation_functions.py contain all the classes needed to construct an MLP, as well as an Optimizer and two Loss Criterion classes; The files functions.py and plots.py contain all the functions needed to train the network and display the results. We will now describe each of the tools provided by our framework.

A. *The classes*

The Sequential : The class allows to successively arrange the modules to be able to implement on them forward and backward propagation.

1) **Forward** : The modules are accessed successively in a ordered manner. They will be addressed by a loop to access each function of the forward process.
2) **Backward** : The modules are addressed in a backward manner. This will permit the computing of the loss gradient descent according to the architecture parameters.

The parameter re-initialization of all the linear layers is done through the Sequential class.

Fully connected layer: This class enables forward and backward passes:

1) **Forward** ($X \in R^{in}$): $S \in R^{out}$
$S = W^T X + b$
2) **Backward** ($\frac{\partial L}{\partial S} \in R^{out}$): $\in R^{in}$

$$bp(\frac{\partial L}{\partial S}) = (\frac{\partial L}{\partial X}) = (\frac{\partial L}{\partial S})W$$

$$\frac{\partial L}{\partial W} = \frac{\partial L}{\partial S}^T X \;\; ; \;\; \frac{\partial L}{\partial b} = \sum_{i=1}^{D} \frac{\partial L}{\partial s_i}$$

3) **Parameters**: In this class, we store the weight $W \in R^{out*in}$ and bias $b \in R^{out}$

The layer stores the input of the forward pass to use them in the backward pass and compute the gradient of the loss with respect to the weight matrix and the bias vector. The gradients are accumulated at every backward pass.

Initialization: The initialization of the weights and the bias of a Linear class are of great importance. Here we implemented the standard uniform PyTorch initialization (performed by default) and The Xavier initialization [1]. It is also known that the weights can be scaled to account for the activation function. As a Linear object is totally independent of an activation object, the **gain** to be applied to the weights is added externally, in the class Sequential.

Activation functions:

The framework implements four distinctive activation functions, as summarized in Table I. These classes store the input at every forward pass, as it is needed to compute the backward pass. We only apply the sigmoid function at the final output of our architecture, only suitable with one dimensional layers.

TABLE I: Forward and backward pass for the activation functions

Activation	Forward	Backward
Relu	$max(0, s)$	$\frac{\partial L}{\partial x} \odot \begin{cases} 0 & \text{if } s < 0 \\ 1 & \text{otherwise} \end{cases}$
Tanh	$tanh(s)$	$\frac{\partial L}{\partial x} \odot 1 - tanh(s)^2$
LeakyRelu	$\begin{cases} 0.01s & \text{if } s < 0 \\ s & \text{otherwise} \end{cases}$	$\frac{\partial L}{\partial x} \odot \begin{cases} 0.01 & \text{if } s < 0 \\ 1 & \text{otherwise} \end{cases}$
ELU	$\begin{cases} a(e^s - 1) & \text{if } s < 0 \\ s & \text{otherwise} \end{cases}$	$\frac{\partial L}{\partial x} \odot \begin{cases} ae^s & \text{if } s < 0 \\ 1 & \text{otherwise} \end{cases}$
Sigmoid	$\frac{1}{1+e^{-s}}$	$\frac{\partial L}{\partial x} \odot \sigma(s)(1 - \sigma(s))$

The Optimizer class: provides an easy-to-use tool to update the parameters of the whole network (i.e. the Sequential object). The user can choose between many algorithms by initializing the Optimizer accordingly. The stochastic gradient descent (SGD) is the most basic implementation using a corrective η value. A momentum can be applied to improve the algorithm. The Adaptive Moment Estimation (Adam) optimizer [2] and the Root Mean Squared (RMSprop) optimizer [3] are an extension of the SGD. As we will see in Section III, all the algorithms have been tested.

The Loss classes: We implemented two loss functions: the Mean Squared Error (MSE) and the Binary Cross Entropy (BCE). Both are represented by a class, LossMSE and BCELoss respectively. When the loss is computed, the inputs (i.e. the target vectors and the output of the network) are stored as a class attribute, such that the subsequent gradient operation is performed automatically.

TABLE II: Loss functions (per sample)

Loss	Forward	Backward
MSE	$\frac{1}{D}\sum_{i=1}^{D}(\hat{y_i} - y_i)^2$	$\frac{2}{D}\hat{Y} - Y$
BCE	$-\frac{1}{D}\sum_{i=1}^{D} y_i log(\hat{y_i}) + (1 - y_i) log(1 - \hat{y_i})$	$-\frac{Y}{\hat{Y}} + \frac{1-Y}{1-\hat{Y}}$

B. The functions

Our framework includes functions to train the model with the standard mini-batch procedure. In functions.py we use model_train to execute the prediction and assess it by returning test and train results. In the below section III , we test our framework on an architecture with a single output (a binary classification problem).

III. Results

A. Data set

As indicated in the specifications, we test our framework on a binary classification problem. The data points (x_1, x_2) are sampled from the uniform distribution inside $[0, 1]^2$. The data is not complex in order to evaluate efficiently our framework.

Data points are labelled $y = 1$ if they are situated within the circle centered at $[0.5, 0.5]$ with radius $1/\sqrt{2}$, and $y = 0$ otherwise. We use 1000 points for training and 1000 points for testing.
The fully connected architecture we use has two input nodes, one output node, and three hidden layers of 25 nodes each.

B. Results and Discussions

In this section, we test the performances of our framework according to the different implemented modules.
We decided to prove the well functioning of our framework by executing the training using the classical SGD, when choosing at each time a different activation function applied to the fully connected layers (Figure . 2). The given average error when using Leaky-Relu, Elu and

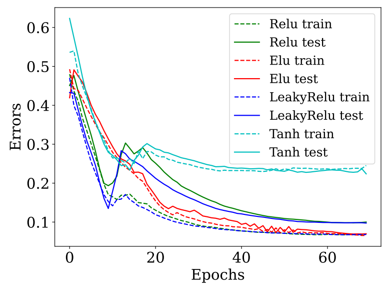

Fig. 2: 10 folds training with different activation functions

Relu activation functions is around 10%, we only notice that Tanh function yields the worst classification with an average error rate of 22%. It is interesting to notice that with Tanh functions we need to have Relu function at the output node of our architecture.

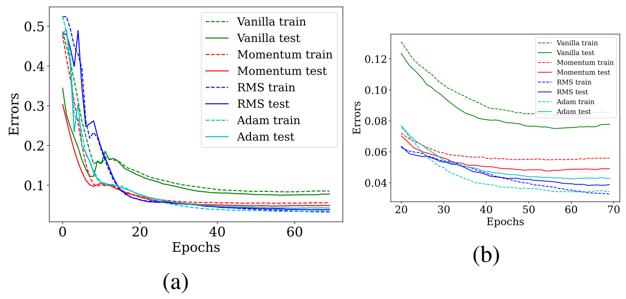

Fig. 3: Error rate over 10 folds training with different Optimizers (a) and zoomed on the last epochs (b)

One of the important features of our framework, is to give the user different choices of the optimizer. In Figure. 3 we verify the theoretical assumptions that both RMSprop and Adam optimizers are better techniques, due to the fact that RMSprop divides the learning rate by an exponentially decaying average of squared gradients, and Adam, additionally to that, computes the adaptive learning rates of each parameter.

To test the performance of the implemented framework better, we executed additional tests comparing the two available loss functions.

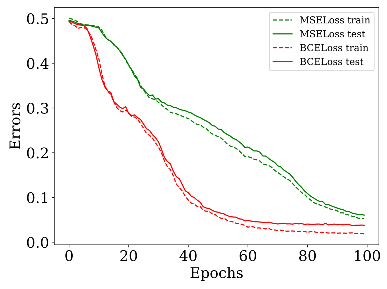

Fig. 4: Comparing the MSE and BCE loss functions

We can clearly notice that the BCE function is converging faster than the MSE function due to its higher adaptability to classification tasks. The Sigmoid function should be used at the entrance of the Binary Cross entropy loss criterion in order to ensure predicted values between 0 and 1.

Furthermore, we still need to know how our framework performs compared to a popular framework such as Pytorch.

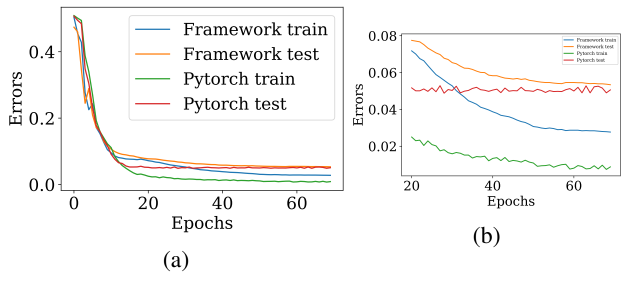

Fig. 5: Comparing the implemented framework with pytorch across 10 folds (a) and zoomed on the last epochs (b)

The Figure. 5 above, shows that both pytorch and the constructed framework are similar for the yielded performances. However, computationally, and if we study the train set prediction more precisely, we find that Pytorch is slightly better than our framework, as shown in Table III.

TABLE III: Framework performances

Framework	Train accuracy	Test accuracy	Time (s)
Constructed framework	0.972	0.947	2.012
Pytorch	0.991	0.949	1.614

IV. Conclusion

We conclude that our framework executes efficiently the classification task with several activation functions. It permits updating the parameters in an optimised manner using Adam and RMSprop techniques. We notice clearly that the overall performances are closely comparable to the ones of Pytorch. Finally, we can possibly improve our framework by trying to implement Convolution layers which will be more adequate to be used on image recognition tasks. It would also be interesting to implement batch normalization layers. The work executed here on implementing a framework from scratch, was very instructive and allowed us to really understand precisely the fundamentals of deep learning frameworks.

References

[1] X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks," in *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics* (Y. W.

Teh and M. Titterington, eds.), vol. 9 of *Proceedings of Machine Learning Research*, (Chia Laguna Resort, Sardinia, Italy), pp. 249–256, PMLR, 13–15 May 2010.

[2] D. P. Kingma and J. Ba, “Adam: A method for stochastic optimization,” 2017.

[3] A. Graves, “Generating sequences with recurrent neural networks,” *CoRR*, vol. abs/1308.0850, 2013.