Miniproject 2 Report

Olivier Cloux[1]

[1]Department of Computer Science, EPFL

***Abstract*—content**

I. Introduction

The emergence of Deep Learning frameworks such as PyTorch [1] or TensorFlow [2] has allowed for easy developments of ever increasing complexity. But constantly using these hides the true mechanisms behind deep learning. This is why in this report we will try and re-implement a basic deep learning framework for ourselves, to delve into how such a framework works.

II. Task

There are two tasks here. The first one, "front one" is to build a model to classify binary points, according to their position in the plane. The second, "actual one" is to do the other task without using the niceties of PyTorch or other deep learning frameworks, in particular without the `torch.nn` module.

A. Classification

Let's talk briefly about the underlying task. It's quite simple: we generate N random points in the $[0,1]^2$ square, and then assign a label of 1 to all points inside the $1/\sqrt{2\pi}$ circle, and 0 outside. Figure 1 shows an instance of such dataset.

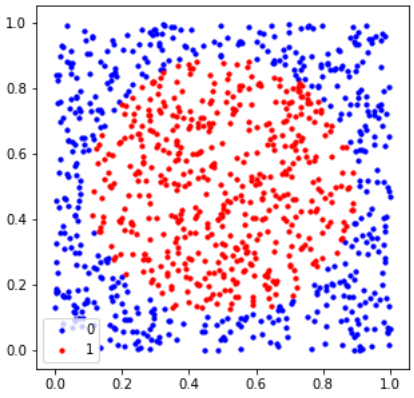

Fig. 1. A 1000-points dataset

III. Framework

The framework I created follows the general idea of PyTorch but allows for more flexibility. It is split in 3 files:

A. modules.py

This file contains a meta class `Module`, and all other classes inherit this one. A "module" is an element of the network: a layer type, an activation function,... The file contains as well the 'Sequential' module, that models a simple MLP. All modules can go in the Sequential module, to be called sequentially. Then, all operations (such as forward, backward, param,...) are called on the MLP, which in turn will propagate through all of its components.

B. criterion.py

Pretty self-explanatory, this file contains the classes that allow for loss computing, such as MSE or CrossEntropy. Just like `modules.py`, there is a meta-class of which all optimizers must inherit. Currently, only MSE and CrossEntropy are coded, but we could easily extend with others, such as MAE, Hinge, Multi-Class Cross-Entropy,...

C. optim.py

The shortest of files. Here we put optimizers. For scope reasons, only SGD (and a quite simple version) was included. But we could extend it with others [3], such as the Adam Optimizer, AdaGrad, AdaDelta, or RMSProp or by adding momentum.

D. Other files

The directory also contains other files, such as `models.py`, `util.py`, and `test.py`. They are here to help showcase the results, and should not be counted as part of the framework. For example, `models.py` contains a list of interesting models for testing purpose, and `util.py` contains useful functions to create datasets, plot them, compute the error,... They are scenario-dependent, contrary to the other files.

IV. Results

V. Summary

References

[1] "PyTorch," library Catalog: pytorch.org. [Online]. Available: https://www.pytorch.org

[2] "TensorFlow," library Catalog: www.tensorflow.org. [Online]. Available: https://www.tensorflow.org/

[3] S. Postalcıoğlu, "Performance Analysis of Different Optimizers for Deep Learning-Based Image Recognition," *International Journal of Pattern Recognition and Artificial Intelligence*, vol. 34, no. 02, p. 2051003, Apr. 2019, publisher: World Scientific Publishing Co. [Online]. Available: https://www.worldscientific.com/doi/abs/10.1142/S0218001420510039

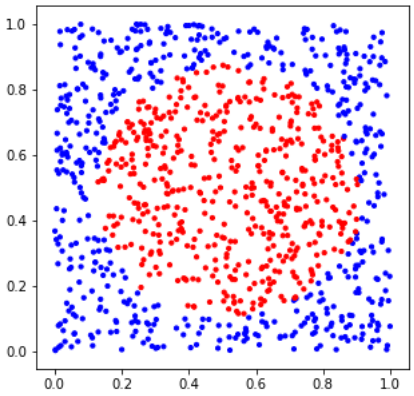

Fig. 2. Predictions of the best model