Writing a mini deep learning module from scratch using PyTorch

Project for the course EE-559, EPFL, Spring semester 2019

William Cappelletti, MA, Charles Dufour, MA, Fanny Huguelet, MA

1 Introduction

In this project, we implement a mini deep learning module, without using the autograd machinery. We try to make our module close in term of use and performance to `torch.nn`, the framework implemented in PyTorch [1]. We then proceed to test our implementation on a dummy dataset and compare the results with those from `torch.nn`.The module can be found at [3].

2 Structure of the module

The structure of the module is represented in Figure 1. It is made to resemble PyTorch as close as possible without using autograd. More details can be found at [3].

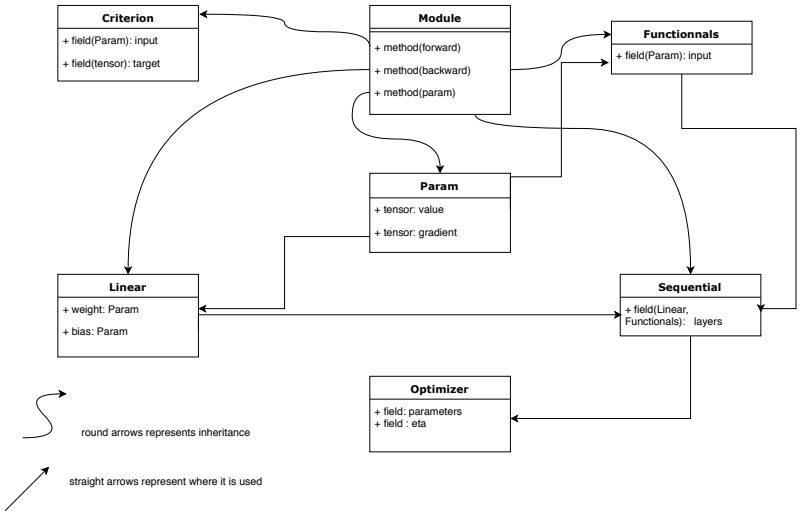

Figure 1: Representation of the different classes and their relation

The Linear layer are created with Xavier initialization for the weights and the bias are set to 0 at the creation of the layer.

The forward and backward passes for the linear layers and the activation functions (`Functionnals`) are implemented following the lecture notes [2].

The `Optimizer` class contains the vanilla stochastic gradient descent and the one with momentum. Those are implemented following the Pytorch formula, with the learning rate multiplying the

updated gradient by the momentum. In the criterion we implement both `MSE` and `CrossEntropy`.

3 Training on dummy dataset and comparison with `torch.nn`

We train on the following data, with 1000 for both training and test: *sampled uniformly in* $[0,1]^2$, *each with a label* 0 *if outside the disk of radius* $1/\sqrt{2\pi}$ *and* 1 *if inside.*

The structure of the model is imposed by the guidelines: *two input units, two output units, three hidden layers of 25 units.* We then decide to use the different losses (`MSE` or `CrossEntropy`) and optimizer, and compare it to the PyTorch framework.

For each model and loss, we train on 200 epochs 20 times, which give us the estimations in Tables 1,2. The parameters are $lr = 0.1$ and $\mu = 0.2$ (parameter for the momentum). If nothing is indicated, the training was done using vanilla stochastic gradient descent.

Module used	loss train (std)	loss test (std)	accuracy train (std)	accuracy test (std)
ours	0.3578 (0.11)	0.06505 (0.05)	98.35 (1.24)	97.61 (1.37)
torch.nn	0.3717 (0.09)	0.05693 (0.03)	98.47 (1.2)	97.65 (1.28)
ours (mom)	0.3221 (0.10)	0.05463 (0.02)	98.34 (0.99)	97.66 (0.94)
torch.nn (mom)	0.359 (0.14)	0.0537 (0.023)	98.46 (0.86)	97.74 (0.88)

Table 1: Comparison using the CrossEntropy Loss

Module used	loss train (std)	loss test (std)	accuracy train (std)	accuracy test (std)
ours	0.2308 (0.08)	0.02694 (0.01)	98.6 (0.51)	97.63 (0.77)
torch.nn	0.2098 (0.07)	0.02619 (0.01)	98.41 (0.72)	97.66 (1.24)
ours (mom)	0.1987 (0.09)	0.02319 (0.01)	98.71 (0.59)	97.95 (0.74)
torch.nn (mom)	0.1631 (0.05)	0.02085 (0.01)	98.25 (0.80)	97.46 (0.95)

Table 2: Comparison using the MSE Loss

So we see no difference in the performance of our module compared to `torch.nn`, even if it is worth noting that our module takes more time up to a multiplicative factor of 2 to do the training of those models.

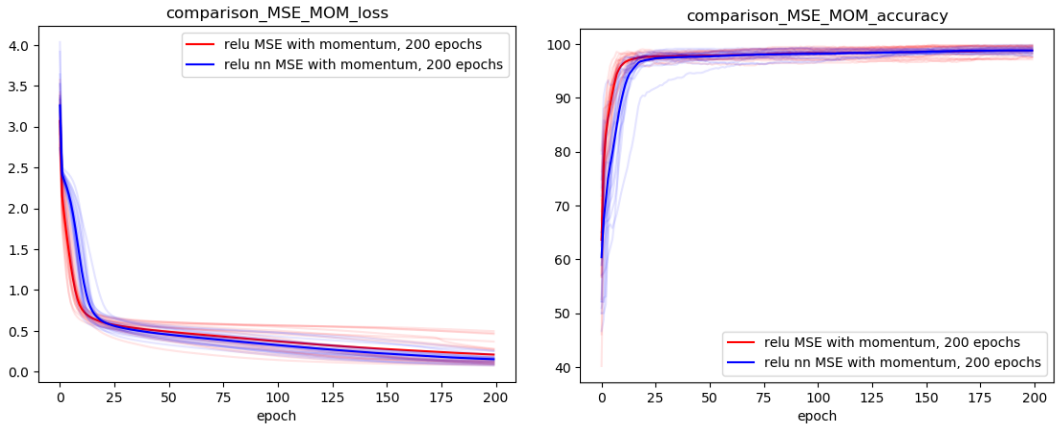

Figure 2: Comparison during the training for our module (red) and `torch.nn` (blue). Bold lines represent the average curves over the 20 experiments, done using MSE and train with momentum.

Appendix

For more plots and visualization of the training, and the comparison between the learning curve of our module compared to `torch.nn`, please see the github [3].

References

[1] Adam Paszke, Sam Gross,. . . ,Adam Lerer *Automatic differentiation in PyTorch*, 2017

[2] François Fleuret, 2019, Deep Learning *Lecture notes*.

[3] Cappelletti, W. and Dufour, C. and Huguelet, F. *Deep-learning-mini-projects*, (2019), GitHub repository, `https://github.com/dufourc1/Deep-Learning-mini-projects/tree/master/Proj2`