Project 2 - Minimalist PyTorch-like deep learning framework

Maxence Jouve, Paul Griesser, Valérian Rey
EPF Lausanne, Switzerland

***Abstract*—Deep learning methods are nowadays very popular and powerful when dealing with the task of learning from images, texts, audios and so on. Nevertheless the networks used in the context of deep learning are complex, so are the underlying mechanisms used to model and train them on a computer efficiently. Even if today's deep learning libraries implement these mechanisms efficiently for us, understanding how they work under the hood is essential for any engineer interested in this field. In this report we describe how we implemented our own minimalist PyTorch-like deep learning framework, to get more insight on these mechanisms. We also show its capabilities on a toy and on MNIST data set. We finally compare our framework with PyTorch on these two data sets, and discuss its shortcomings.**

I. Introduction

This project takes place in the context of the deep learning EE559 course from EPFL [1]. It aims to gain more insight in how deep learning libraries work under the hood, especially how neural networks are built, trained using forward and backward passes, and how their parameters are optimized. To this end, we built our own minimalist PyTorch-like deep learning framework. With this framework we are able to:

- build networks combining fully connected layers, Tanh, and ReLU,
- run the forward and backward passes,
- optimize parameters with SGD for MSE and cross entropy loss.

Below we describe how the core features were implemented and also show our reflections during the design of our framework. Eventually we make a demonstration of its effectiveness in section IV.

II. Basic Structure

As suggested in the project description, we defined a `Module` class which will be the superclass for all neural network submodules.

```
class Module(object):

    def forward(self, *input):
        raise NotImplementedError

    def backward(self, *gradwrtoutput):
        raise NotImplementedError

    def update_params(self, step_size):
        return

    def zero_grad(self):
        return

    def param(self):
        return []
```

[1]We advise the reader to be aware of the project description that can be found here: https://fleuret.org/ee559/materials/ee559-miniprojects.pdf

We added two methods to the recommended ones:

- `update_params` updates the parameters of the module by doing a step of size `step_size` in the opposite direction of the gradient. If the module is parameterless, just return.
- `zero_grad` sets the gradient of the parameters of the module to zero. If the module is parameterless, just return.

The concrete submodules all inherit from the `Module` class. Here is the list of the ones we implemented:

- Linear: represents a fully connected linear layer in a neural network.
- Losses:
  - LossMSE: represents the mean square error loss.
  - LossCrossEntropy: represents the cross entropy loss.
- Activations:
  - ReLU: represents the rectified linear unit activation function.
  - Tanh: represents the hyperbolic tangent activation function.
- Sequential: represents a neural network composed of several modules, in our case: fully connected linear layers and activations.

The important aspects of implementation and reflection are detailed below. However, most aspects are straightforward and will not be detailed (e.g. the activations and losses). Please refer to the code for more information.

A. Linear Module

This module's initializer takes as input the number of input features, the number of output features and whether or not to add a bias when forwarding. It then creates a weight, and bias tensor along with a similarly sized gradient tensor to accumulate the gradient during the backward pass. We also initialize the weight in the same manner as PyTorch i.e. $w \sim U[-\frac{1}{\sqrt{N}}, \frac{1}{\sqrt{N}}]$ independently for all w of the weight tensor where N is the number of input features to the layer. The rest of the implementation is straight forward and not detailed here.

B. *Sequential Module*

This module allows to combine several modules, such as linear layers and activations, in a sequential manner in order to represent a neural network. Its initializer takes several modules as input and store them in a list. Then each method consists of a loop over all element of this list and call the respective method on them.

III. Training

With the modules described above we are able to build a simple neural network composed of linear layers and activation functions between them. Now we also need a way of training the network given data i.e. optimizing the network's parameters for a given loss. For this, we implemented a method in charge of the training, here is its definition:

```
def train_model(model, criterion, train_input,
                train_target, nb_epoch,
                batch_size, step_size, logging=
    False):
```

Given a model, some data as well as a criterion i.e. a loss, the method will perform SGD to minimize the loss of the model. Once the training is finished, the method returns the final accuracy of the model on the training data.

IV. Demonstration

In order to asses the capabilities of our framework we tested it on a toy data set which consist of 1,000 points sampled uniformly in $[-1, 1]^2$, each with a label 0 if outside the disk of radius $1/\sqrt{2\pi}$ and 1 inside, see figure 1.

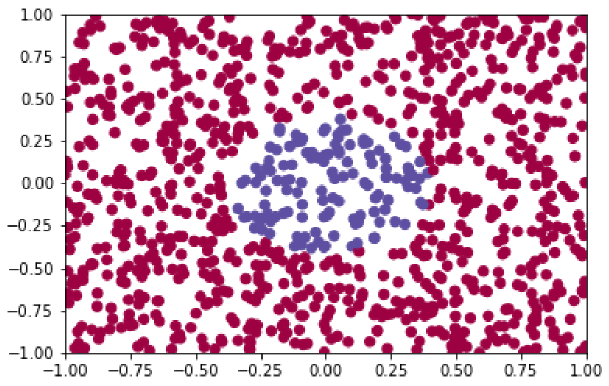

Figure 1. Data points

As asked, we built a network with two input units, two output units and three hidden layers of 25 units and ReLU as activations. We trained it with MSE and a step size of 0.03 over 2000 periods using mini batches of size 100 to speed up the training. Here is the classifier we got:

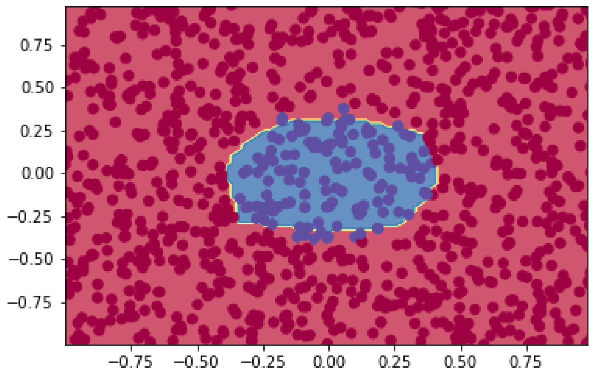

Figure 2. Classifier using MSE loss

We noticed that considering MSE as our loss function is not the best option as our task is a classification rather than a regression. We ran the same experiment with the same parameters but we considered the cross entropy loss, this is what we obtained:

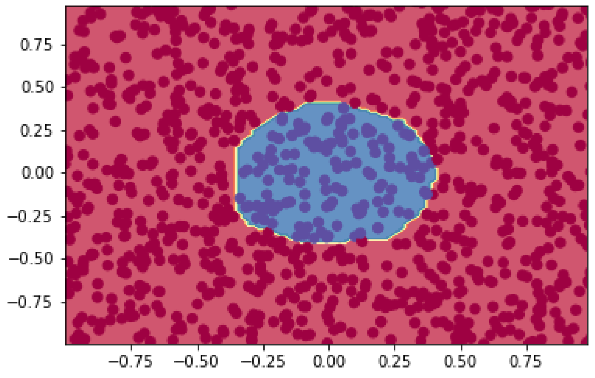

Figure 3. Classifier using cross entropy loss

The results are indeed better as the classifier's separation line looks more like a circle than before. To comfort us in our idea that using the cross entropy loss is more appropriate than the MSE loss, we averaged the accuracies and F1 score of both models on 10 runs. We also recorded the time to train the model and averaged it over the 10 runs. Note that in regards to the data, the F1 score is more suitable to assess the performance of the network, this is due to the imbalance between both classes. Results can be found in table I. It is clear that cross entropy loss leads us to better results but note that the training time is worse.

	MSE	CrossEntropy
Training time	14.00s	25.64s
Accuracy$_{train}$	99.69%	99.97%
F1$_{train}$	0.987	0.998
Accuracy$_{test}$	98.20%	98.47%
F1$_{test}$	0.938	0.948

Table I
RESULTS USING OUR FRAMEWORK

A. Comparison with PyTorch

A good way to asses the capabilities of our framework is simply to compare it to PyTorch. We repeated the same experiment described in the beginning of section IV but using PyTorch modules. We used the same structure for the neural networks, trained it on the same data, with the same hyperparameters. Again we averaged the results of 10 independents runs to have more accurate results. See table II.

	MSE	CrossEntropy
Training time	15.63s	15.17s
Accuracy$_{train}$	99.75%	100%
F1$_{train}$	0.990	1
Accuracy$_{test}$	98.19%	98.57%
F1$_{test}$	0.936	0.950

Table II
RESULTS USING PYTORCH'S FRAMEWORK

Our results are very close to the one given when using PyTorch's framework except for the training time when using our own cross entropy loss. This is very satisfying and shows that our implementation works fine.
Finally we used our framework to create and learn a model on the MNIST dataset. We again compared our framework to PyTorch. The structure of the neural network is composed of three hidden layers of size 512 and ReLU as activation, using 10,000 samples, a step size of 0.02 and training on 100 epochs, of course the cross entropy loss was used. The experiment was repeated 10 times to have more accurate results. These results can be found in table III.

	Our framework	PyTorch's framework
Training time	96.46s	112.4s
Accuracy$_{train}$	100%	100%
Accuracy$_{test}$	95.08%	95.23%

Table III
RESULTS ON MNIST

Again the accuracies are very similar. More surprisingly we outperformed PyTorch with our framework which is really satisfying.

V. DISCUSSION AND CONCLUSION

Regarding the results, we are pretty satisfied with our framework. Using it, we have the tools to build simple deep neural networks which are very powerful for many tasks such as classification or regression. Of course our framework lacks a lot of functionalities such as ways of regularizing, convolutions, batch normalization for deeper networks and so on. But the goal wasn't to re-code the entire PyTorch library but more to gain insight on some important functionalities such as the backward pass. Regarding the learning opportunity, we conclude that the project was a success.