Deep Learning Project - EE-559
Mini Deep-Learning Framework

Andrea Cassotti, Nathan Girard
Section of Life Science Engineering, Section of Financial Engineering, EPFL, Switzerland

***Abstract*—In this project we aim at designing a simple deep learning framework using the standard library *math* and nor basic *pytorch's tensor* operations. First of all, we needed to create the data set on which to train and assess the accuracy of the network we constructed. For this purpose, we implemented different modules and methods so as to build a network and optimize the latter.**

I. Introduction

Without the *nn* and *autograd* modules of pytorch, we built a deep learning framework such that the user can define models, train it and easily compute the accuracy of their model(s). The tools we designed allows more specifically for :

- build multi-layered netwroks with fully connected layers and the activations functions cited above,
- optimize the parameters using stochastic gradient descent (**SGD** hereafter) and the **Adam** optimizer,
- find the best combination of the hyperparameters values with cross-validation,
- and evaluate the accuracy of the model(s).

Also, we decided to predict the labels of the samples stochastically, hence the 2 output units instead of 1. In fact, the network predict the label with maximum probability. In the following sections, we will explain the methods implemented. We will then discuss our choices of the different modules with the results of different structures we implemented.

II. Data

The training and test sets consist of 1000 data points sampled from the uniform distribution in $[0, 1]^2$, labeled each with 1 if they are inside a disk of radius $1/\sqrt{2\pi}$ centered at (0.5, 0.5), and 0 otherwise.

III. Methods

In this section, we discuss the structure of the project. We wanted to design a framework which could be flexible. Each module implemented will use a forward pass and backward pass. The forward pass allows to compute output of the different modules given their input. Then we find the cumulative gradient with respect to the parameters (weights) across the layers with the backward pass, working in reverse order (from the last layer to the first one). Hence we divided the tasks into different classes inheriting from the same super class **Module**, which has the following form:

```
class Module(object):
    def __init__(self):
        pass

    def forward(self, *input):
        raise NotImplementedError

    def backward(self, *gradwrtoutput):
        raise NotImplementedError

    def param(self):
        return []

    def init_params(self, Xavier, Xavier_gain):
        pass

    def gradient_descent(self, learning_rate):
        pass

    def zero_grad(self):
        pass
```

A. Modules

1) Linear: We began to introduce a **Linear layer** class, a fully connected layer. The forward pass will calculaze the outputs through the layers and hence the final prediction on the samples:

$$z^l = w^l x^{l-1} + b^l$$

Then we build the **Sequential** class that allows the stacking of multiple layers. As the initialization of weights can be of great importance for the training model, and hence their prediction accuracy, we used the **Xavier initialization** method, although its implementation could be not significant for this project as the number of layers is small. We want, with this method, to prevent the potential explosion or vanishing of the outputs during the forward pass. As a consequence, the gradient should propagate more efficiently into the deep layers and thus the convergence of the network should be faster.

2) Activations: In between the linear layers, we need activation functions, which will determine if the output of the precedent layer should be taken into account.

$$a^l = g(z^l)$$

where g is the activation function.

We built the rectified linear unit (**ReLU** hereafter), hyperbolic tangent (**Tanh** hereafter), and sigmoid activations to test for the most appropriate for our prediction task.

Optimizer While training the model, we need to optimize the models(s) parameters with two different methods: **SGD** and **Adam**. We created the class **Optimizer** which train the model(s) with these methods in addition to calculate the prediction accuracy of the model(s).

- In SGD, parameters are updated according to the equation below:

$$w_t = w_{t-1} - lr * \nabla w_t$$

$$b_t = b_{t-1} - lr * \nabla b_t$$

where lr is the learning rate, w_t the weight, and b_t the bias.

- in Adam, the update formula is [1]:

$$w_t = w_{t-1} - lr * \frac{m_t}{\sqrt{v_t} - \epsilon}$$

where lr is the learning rate, m_t and v_t the estimates of the first and second moment, respectively. ϵ prevent from division by 0.

3) Cross-Validation: The class **Cross_Validation** allows to test several hyperparameters values. In this sense, we try to find the combination of parameters that yield the best performance (defined as the prediction accuracy), with the provided range of hyperparameters inputs. It is rather a brute-force search of the values, yet it is simple method that can be used, especially in our context (small networks are not really computationally expensive).

B. Weights and Biases updates

An evaluation of the error on new samples, different from the training set, enables the network to correct the weights and biases of the different layers. To do so, a measure of the error between the predicted and true label of the sample is used. We implemented the mean squared error (**MSE** hereafter) and mean absolute error (**MAE** hereafter) function, defined below:

$$Loss_{MSE} = \sum_{n=1}^{N} (label_{predicted} - label_{true})^n$$

where N is the number of samples, n the order (MAE if n = 1, MSE if n = 2).

In the idea of finding the best optimization method, we also build a module defining the cross entropy loss. The latter will influence the model to find the distribution (with the probability modeled according to the Bernoulli distribution) of the labels using the formula below :

$$H(P,T) = -\frac{1}{N} \sum_{n=1}^{N} (P(n) * log(T(n)))$$

where N is the number of samples, P and T are the predicted and true distribution of the labels, respectively.

The update of the parameters is proportional to the gradient of the error function (backward pass of the Loss class) with respect to the output of the parameters.

IV. Results

In this project, all our models use a batch size of 10 and 100 epochs for training. Although we already defined the modules, we have to perform test over different architectures, optimization and parameters initialization methods, and hyperparameters values. To do so, we first created three models using a different activation function each (**ReLU**, **Tanh**, **Sigmoid**). As for the optimization component, we tried both the famous SGD and Adam on all three models.

However, these optimization methods use hyperparameters, which need to be determined. Cross-validation was the simplest way to test for different combinations of these hyperparameters values. The values tested for each of the latter are shown in table I:

Hyperparameter	range	number of values
Learning rate	1e-4 - 1e-1	10
Epsilon	1e-8 - 1e-6	3
β_1	0.8 - 0.9	2
β_2	0.9 - 0.999	2

TABLE I

HYPERPARAMETERS VALUES TESTED WITH CROSS-VALIDATION. PARAMETERS EPSILON β_1 AND β_2 ARE USED IN ADAM.

Also, we tested for the influence of the Xavier weight initialization method on the three models, set with the best hyperparameters values. In this case, we set the gain of the standard deviation, used in the Xavier method, to **6.0**.

After we performed cross-validation on the three models, we found that the best values of the hyperparameters are the following :

Hyperparameter	Best value SGD	Best value Adam
Learning rate	1e-4	1e-4
Epsilon	1e-8	1e-8
β_1	0.9	0.8
β_2	0.999	0.899
accuracy	0.97	0.98

TABLE II

BEST VALUES OF THE HYPERPARAMETERS WITH SGD OR ADAM OPTIMIZATION.

In the continuity of our project, we determined the accuracy scores which have shown no significant differences (**0.97 - 0.98**), shown in figure 1, across the three models trained with the three activation functions: ReLU, Tanh, and Sigmoid. Also, the choice of the optimization method (SGD or Adam) as well as the choice of loss function are not crucial for the simple task we performed.

V. Discussion

We proved with this project that it is possible to achieve satisfying results with the framework we implemented. Although the prediction task is a simple problem, it still shows interesting aspects about the choice of activation function, optimization method, and loss function choices. From figure 1, we observed no advantages depending on the choice of activation function, as the accuracy scores are equal.

In the same way, the choice of optimization method has poor influence on the models scores, as seen in table II, since the variation of accuracy scores across the different models is small generally. The model using the cross entropy loss shows, as expected, more robust performance with the different optimization methods (**0.97** with SGD and **0.98** with Adam) than the other models. For practical reasons, optimization with SGD seems the best choice because this

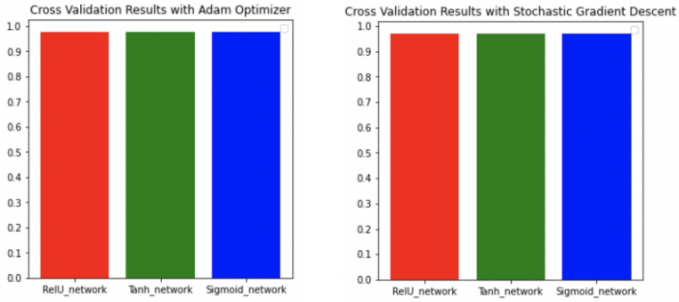

Fig. 1. Accuracy scores of the three different models (ReLU, Tanh, and Sigmoid networks) with best combination of parameters (see table II found with cross-validation. Left : Results with Adam optimization and cross entropy loss. Right : Results with SGD and cross entropy loss.

method requires less parameters that have to be tuned (only the learning rate instead of 4 parameters with Adam).

On the contrary, using cross entropy loss instead of standard MSE loss demonstrated an advantage in the accuracy scores of the different models. This has been seen through the stability while optimizing the thyperparameters values. In fact, the use of cross entropy loss did not show oscillation of accuracy scores during tuning of hyperparameters in comparison to large amplitude oscillation observed with MSE (range with MSE: 0.30 - 0.97). It is rather expected since the loss function measure probabilities of our binary classification task. Hence, the product of probabilities can yield scores too small to be analyzed correctly by computers.

The Xavier method proved to be not particularly relevant in our project as hypothesized earlier. We conclude then that small models like ours (3 hidden layers) do not suffer from the vanishing gradient issue.

Further research and improvement could be considered. We could have implemented a more user friendly code through systematic management of exceptions and errors (when choosing the values of the parameter). As the models and data used in this project are relatively small, the optimization could benefit from the use of penalization, so that the parameters values don't reach extreme values.

ACKNOWLEDGEMENTS

We would like to thank the professor Fleuret F. and the teaching assistants of the Deep Learning course for the quality of the course and the quality of the help proposed to the students (although not needed for the projects).

REFERENCES

[1] Adam: A Method for Stochastic Optimization *Diederik P. Kingma and Jimmy Ba (2017) arXiv:1412.6980*