Project 2
Mini deep-learning framework

Marie Anselmet, Héloïse Monnet, Jasmine Nguyen Duc
Deep Learning EE-559, EPFL, Switzerland

***Abstract*—This paper explains the process that was followed to design a mini deep-learning framework from scratch, in particular using only PyTorch's tensor operations and the standard math library. The framework contains all the basic tools to build, train and evaluate a fully-connected neural network. This paper also assesses the performances of the different modules provided.**

I. Introduction

During the last years, deep learning has become a field of increasing interest and a large number of deep learning librairies and platforms have been released. These packages support a variety of deep learning architectures such as feed-forward networks, convolutional neural networks and recurrent neural networks.

The aim of this paper was to implement a simple deep-learning framework without using PyTorch's neural-network modules nor its automatic differentiation package. The framework designed contains all the modules needed to build networks combining fully connected layers and several popular activation functions (Sigmoid, Tanh, ReLU, LeakyReLU, ELU and SELU). It is able to run the forward and backward passes, and to optimize parameters for the MSE or the Cross Entropy loss criterion. The SGD and the ADADELTA optimizers are implemented. Two schedulers are also provided to optimize the learning rate during training: a step decay and a CLR scheduler.

II. Implementation

A. Module class

In order to implement the framework, it was decided to first design a parent class Module from which all the following modules would inherit and overwrite the *forward()*, *backward()* and *param()* methods. These methods are detailed for each child module below.

Notice that the framework is compatible with batch processing, which enables to considerably speed up learning.

B. Linear module

The linear module implements a simple fully connected layer. According to Xavier initialization [1], weights are initialized with a normal distribution centered in $\mu = 0$ and with variance $\sigma^2 = \frac{2}{n_{in}+n_{out}}$, where n_{in} and n_{out} are respectively the number of incoming and outgoing connections from the layer. This initialization strategy should help in avoiding vanishing or exploding gradients problems. Biases, meanwhile, are initialized with a normal distribution with mean $\mu = 0$ and variance $\sigma^2 = 1$.

The *forward()* method computes the forward pass of the backpropagation algorithm, that is:

$$y = Wx + b$$

with x and y respectively the input and output features, W the weights and b the biases of the layer.
backward() computes the backward pass as follows:

$$\frac{\partial \mathcal{L}}{\partial x} = \frac{\partial \mathcal{L}}{\partial y}\frac{\partial y}{\partial x} = \frac{\partial \mathcal{L}}{\partial y}W$$

$$\frac{\partial \mathcal{L}}{\partial W} = \frac{\partial \mathcal{L}}{\partial y}\frac{\partial y}{\partial W} = \frac{\partial \mathcal{L}}{\partial y}X^T \qquad \frac{\partial \mathcal{L}}{\partial b} = \sum_{i=1}^{N}\Big(\frac{\partial \mathcal{L}}{\partial y}\Big)_{:,i}$$

with $\mathcal{L}$ the loss and x, y, W as before. Thus this method computes the derivatives of the loss with respect to the weights and the biases that will later be used for parameters update in the optimizer.
param() stores the weights, biases and the gradients of the loss with respect to the weights and the biases.

C. Sequential module

The sequential module enables to combine several modules in a basic sequential structure, for a convenient building and training of the network. More precisely, *forward()* takes as input the list of individual modules and sequentially calls their own *forward()* method, whereas *backward()* executes the *backward()* method of each individual module but in the reverse order.

D. Activation functions modules

Six possibilities of activations are provided as modules: Sigmoid, Tanh, ReLU, LeakyReLU, ELU and SELU. Their *forward()* method returns as output the activation function applied to the input features. *backward()* on the other hand computes the gradient of the loss with respect to the input, since the gradient propagates also through activations between the successive linear layers as it flows back. Activation functions have no parameter to optimize over, so *param()* returns an empty list.

E. Loss criteria modules

Mean Squared Error (MSE) and Cross Entropy were implemented as loss criteria modules. Concerning MSE, its *forward()* and *backward()* methods compute respectively the loss and the gradient of the loss with respect to the output as follows:

$$\mathcal{L}(z,t) = \frac{1}{N}\sum_{i=1}^{N}(z_i - t_i)^2 \qquad \frac{\partial \mathcal{L}}{\partial z}(z,t) = \frac{2}{N}(z_i - t_i)$$

with z the output of the network, t the corresponding targets and N the number of train samples (or the batch size in case of batch processing).

Nonetheless, if MSE is a very popular criterion and has nice optimization properties like convexity, it is conceptually wrong for classification since it penalizes too much the responses that are classified "in the right side". On the contrary, the Cross Entropy loss does not penalize the outputs which are perfectly valid for prediction, and relies on probabilities to

compute the distance between the prediction and the target, so that the gradients are steeper than with the MSE. In order to get probabilities as output before applying the Cross Entropy criterion, a softmax activation was first applied in the *forward()* method of the Cross Entropy module. The *backward()* method, as before, computes the gradient of the loss with respect to the output of the network.

F. Optimizers modules

A class Optimizer was first defined as a parent class for the optimizers implemented later on. It contains two methods: *step()* to perform a gradient step on the model parameters and *zero_grad()* to zero the gradients of these parameters.

Two optimizers were then implemented: SGD (Stochastic Gradient Descent) and ADADELTA (standing for Adaptive Learning Rate Method).

SGD is a stochastic approximation of gradient descent optimization, since it replaces the gradient calculated from the entire dataset by an estimate calculated from a randomly selected subset of the data. More precisely, its *step()* method updates the model parameters as follows:

$$W = W - \lambda \frac{\partial \mathcal{L}}{\partial W} \qquad b = b - \lambda \frac{\partial \mathcal{L}}{\partial b}$$

with λ the learning rate, which should be carefully chosen. The SGD module can be used with or without momentum. Momentum consists in accumulating a velocity vector in directions of persistent reduction in the objective across iterations, in order to accelerate gradient descent and thus fast the convergence [2]. The implementation of momentum is defined as follows for the weights:

$$W_t = W_{t-1} - v_t \quad \text{with} \quad v_t = \mu v_{t-1} + \lambda \frac{\partial \mathcal{L}}{\partial W}$$

where μ is the momentum coefficient. The same principle is used for the biases.

The ADADELTA module was also implemented according to [3]. This optimization method dynamically adapts over time using only first order information, so that it does not require any manual tuning of the learning rate and appears robust to noisy gradient information. Moreover, it has minimal computational overhead beyond SGD.

G. Scheduler module

A scheduler module was finally implemented, providing two different methods to optimize the learning rate during training: *step_decay()* and *cyclical_lr()*.

The *step_decay()* method drops the learning rate by a factor every few epochs, such as:

$$\lambda = \lambda_{initial} \cdot \alpha^{\text{floor}(\frac{epoch}{\beta})}$$

By default, $\alpha = 0.5$ and $\beta = 10$, that is the learning rate is dropped by half every 10 epochs.

The *cyclical_lr()* method implements the Cyclical Learning Rate (CLR) algorithm described in [4]. It gives a way to schedule the learning rate in an efficient way during training, by varying it between an upper and a lower bound in a periodic triangular fashion.

III. Dataset

In order to test the performances of the different modules implemented, a quite simple data set was used. The train and the test sets consisted of points sampled uniformly in $[0, 1]^2$, each with a label equal to 1 if the sample point lays inside the disk with center $[0.5, 0.5]$ and radius $\frac{1}{\sqrt{2\pi}}$, or equal to 0 if outside the disk. The experiments were done on normalized data, randomly split into 1000 samples for training, 1000 samples for validation and 1000 samples for testing. The upper pannel on figure 2 shows an example of such a dataset before normalization.

IV. Performances

The network used for classification of the data set consisted of two input units, three hidden layers of 25 units and two output units.

All the models described below were trained during 400 epochs (unless specified). A batch size of 100 samples was used during the training process. Performance results (i.e mean and standard deviation of the classification error) provided below were estimated through 5 rounds for each architecture. Data was shuffled at the beginning of each round.

A. Loss criteria

The two loss criteria implemented were tested and compared in function of the evolution of the validation error with the epoch. The activation function used was Tanh with a learning rate of 1e-1. As it can be seen in figure 1, the error corresponding to Cross Entropy decreases in a stable manner at the beginning of the training, whereas the error fluctuates a lot with MSE.

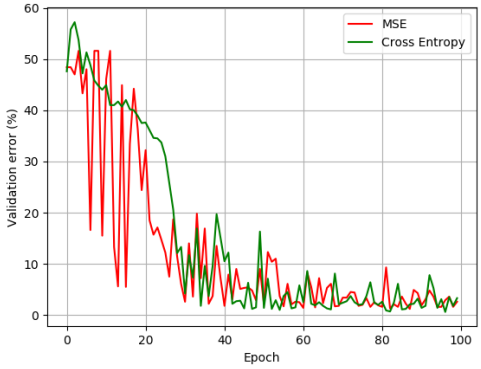

Fig. 1: Evolution of the validation error with the epoch in function of the loss criterion.

B. Activation functions

Each activation function was tested with Cross Entropy as loss criterion and SGD as optimizer (without moment). The learning rate used with each activation and the performances obtained are summarized in table I.

Notice that Sigmoid and ReLU used alone gave really variable performances, thus they were used in combination with Tanh.

TABLE I: Network performances with different activation functions.

	Learning rate	Error mean (%)	Error s.d. (%)
Sigmoid	1e-1	1.60	1.57
Tanh	1e-2	1.06	0.41
ReLU	1e-1	2.64	1.90
LeakyReLU(0.01)	1e-1	2.58	1.88
ELU	1e-2	1.80	0.53
SELU	1e-1	1.32	0.47

Figure 2 shows an example of dataset used for testing, and the resulting performance of the network (with Tanh as activation function). It can be observed on the lower panel that all the errors made by the network are at the border of the disk delimiting the two classes.

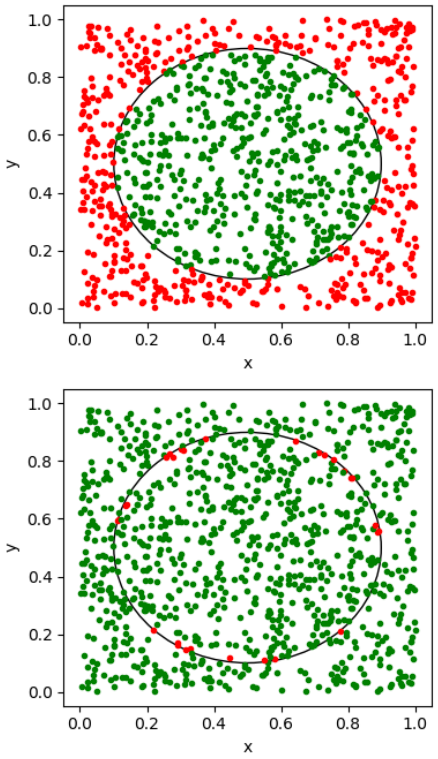

Fig. 2: Top: test set. Green points have label 1 and red points have label 0. Bottom: Green points are the ones correctly classified by the network and red point are the misclassified ones.

C. Optimizers

The SGD and ADADELTA modules were then tested, with MSE, Tanh and a learning rate of 1e-2. SGD was used with different values of momentum. Each model was trained during 100 epochs, in order to see a difference in the velocity of learning. Table II sums up the performances obtained.

TABLE II: Network performances with SGD and different momentums or with ADADELTA.

	Momentum	Error mean (%)	Error s.d. (%)
SGD	0	5.20	1.60
	0.5	5.00	1.82
	0.9	4.42	2.11
ADADELTA	X	4.96	2.54

D. Schedulers

The SGD modules was also tested with the two schedulers implemented: Step decay and CLR, which are illustrated in figure 3. The loss function used was Cross Entropy, the activation function Tanh and the learning rate 1e-2. The results are shown in table III.

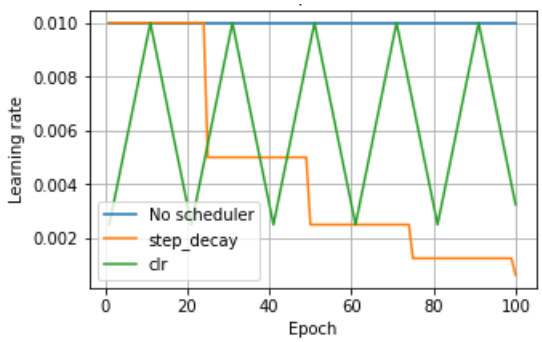

Fig. 3: Evolution of the learning rate with the epoch, in function of the scheduler used with SGD.

TABLE III: Network performances with SGD and different schedulers or with ADADELTA.

	Scheduler	Error mean (%)	Error s.d. (%)
SGD	X	1.66	0.81
	Step decay	1.58	0.71
	CLR	1.38	0.16
ADADELTA	X	1.86	0.93

V. Summary

To summarize, this paper presents the approach that was adopted to build a mini deep learning framework from scratch. To illustrate this, a simple fully connected neural network composed of 4 linear layers was implemented and tested on a quite simple classification dataset. Several activation functions such as Sigmoid, Tanh, ReLU, LeakyReLU, ELU and SELU were implemented. Optimizer modules such as SGD -with and without momentum- or ADADELTA were also built to enable more robustness in the optimization process. In addition, schedulers for the learning rate were also implemented. As loss criteria, MSE and Cross Entropy loss modules were both developed to enable more flexibility in the types of problems to optimize. If the proposed implementation seems to reach good performances on this simple classification task, the training process is yet longer and substantially less stable compared to popular frameworks. Finally, even if many improvements - like in loss criteria, optimizers or activation functions - may upgrade the performances, the diverse implementations explained in this paper have brought satisfying improvements in the obtained results and have enabled further understanding on their functioning.

References

[1] X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks," *Journal of Machine Learning Research - Proceedings Track*, vol. 9, pp. 249–256, 2010.

[2] I. Sutskever, J. Martens, G. Dahl, and G. Hinton, "On the importance of initialization and momentum in deep learning," in *Proceedings of the 30th International Conference on Machine Learning*, vol. 28, no. 3, 2013, pp. 1139–1147.

[3] M. D. Zeiler, "Adadelta: An adaptive learning rate method," 2012.

[4] L. N. Smith, "Cyclical learning rates for training neural networks," 2015.