Implementing a Deep Learning framework from scratch

Alleon Antoine
Computational Science & Engineering
Email: antoine.alleon@epfl.ch

Dellafererra Giorgia
Physics
Email: giorgia.dellaferrera@epfl.ch

Zampieri Luca
Computational Science & Engineering
Email: luca.zampieri@epfl.ch

***Abstract*—The aim of this project is to design a "deep learning framework" without any pre-existing neural-network python toolbox. The network consists of two input units, three hidden layers of 25 units each and two output units, and it is trained to classify a set of 1000 data points sampled uniformly in $[0,1]^2$ into two classes, composed of the points lying respectively inside and outside a disk of radius $\frac{1}{\sqrt{2\pi}}$.**

I. Introduction

Facial recognition, speech processing, object detection and labeling and so much more... Deep Learning is revolutionizing many areas of science, aiming to improve our every day lives. Many high end libraries aim to give accessibility to these tools, hiding the technical details. But how to implement such a deep framework from scratch? This project answers this question implementing some key features for neural networks and then testing them on a generated toy dataset in order to easily understand the effect of each modification.

II. Dataset

A function has been implemented to generate a dataset as a set of 1000 uniformly distributed points between $[0,1]^2$ with the label 1 if inside the circle, and 0 if outside. Three datasets (the training set, the validation set and the testing set) have been generated using the implemented function: due to the randomness component of the function, the three data sets are characterized by the same features but are not identical. Figure 1 reports the dataset generated for the training.

Thanks to randomness, the dataset is close to be balanced. One could enforce it removing the points in excess or generating the needed points.

III. Implemented structure

A Sequential module has been implemented, in order to combine comfortably all modules and basic components of a neural network, such as linear layers, tanh & Relu activation functions and MSE loss.

The network structure consists of two input units, three hidden layers of sizes 25 units, and two output units. Each hidden layer combines one linear layer and one activation layer. Different alternatives for the activation functions have been implemented as explained in the section *Activation functions*.

Furthermore the Loss module has been implemented. The chosen loss function is the MSE loss, as described in the *Loss* section.

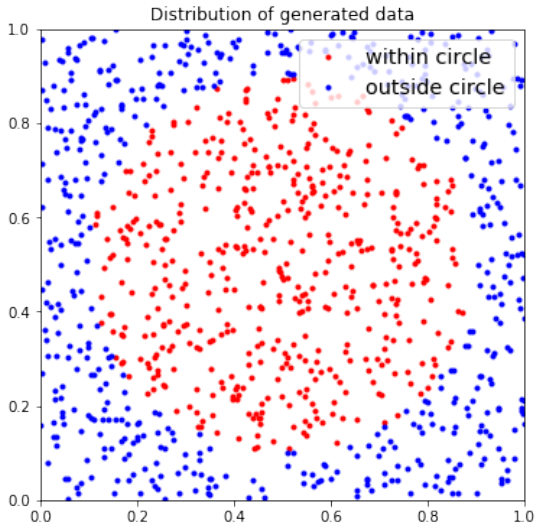

Fig. 1: Input training set

A. Setting a baseline

A baseline has been set through a function with a simple classifier. The size of the input has been increased in the following way. Being the distribution of the data circular, it is convenient to augment the input data by appending to the input data an array containing the sum of the squares of the coordinates. Indeed, this allows to have a better accuracy while training the model with a linear classifier.

B. Activation functions

Four choices of activations functions have been implemented and exploited in the Sequential module:

- Rectified hyperbolic tangent $x \to \frac{1}{2}(1+tanh)$
- Rectified Linear Unit (ReLU) $x \to max(0,x)$
- Leaky ReLU with a tuned to $a = 0.1$ $x \to max(ax,x)$
- Sigmoid $x \to \frac{1}{1+e^{-x}}$

C. Tuning of the basic structure

The activation functions presented above can be combined in different ways in the construction of the network. Among the various alternatives that have been investigated, three possibilities are, for instance:

- Leaky ReLU as first and second activation layers and the Sigmoid function as the third one
- ReLU as first and second activation layers and the rectified hyperbolic tangent function as the third one
- Rectified tanh for all the three layers

The two former alternatives lead to very similar results. The third option still gives satisfactory results but it takes slighlty more epochs to reach a comparable accuracy.
The results that are presented in the following sections have been obtained by employing the second structure.

D. Loss

The exploited loss function is the Mean-Squared Error loss, defined as

$$\mathcal{L} = \frac{1}{N}\sum_{n=1}^{N}(y_n - f(x_n; w))^2 \tag{1}$$

E. Optimizers

Four different optimizers have been implemented: "vanilla" Mini-batch Stochastic Gradient Descent (SGD), SGD with the addition of momentum, Adam optimizer, and BFGS algorithm.
Standard gradient descent algorithm updates the parameters only after the algorithm has been run through all the samples, following the update rule:

$$w_{t+1} = w_t - \eta \nabla \mathcal{L}(w_t) \tag{2}$$

This procedure does not allow to have the best improvement of the parameters at any instant. Therefore it is more convenient to visit the samples in "mini-batches", of size B of some tens of samples, and update the parameters each time:

$$w_{t+1} = w_t - \eta \sum_{b=1}^{B} \nabla \ell_{n(t,b)}(w_t) \tag{3}$$

This is the **"Vanilla" Mini-batch SGD**. In our implementation the samples in the mini-batches are shuffled at each epoch and are then visited in sequential order.
A fixed step size and the assumption on the isotropy of the local curvature of the function might result, for the SGD algorithm, in overshooting and divergence. Statistics over the past values of the gradient, considering higher order momenta, can be exploited to overcome these limitation: for example "momentum" is employed to add inertia in the choice of step direction, in the optimizer **SGD with the addition of momentum**.

$$w_{t+1} = w_t - u_t \tag{4}$$

where

$$u_t = \gamma t_{t-1} + \eta g_t \tag{5}$$

The third implemented optimizer is **Adam algorithm** which uses moving averages of each coordinate and its square to rescale each coordinate separately. The update rule, applied on each coordinate separately, is reported in the section A of the Appendix.
BFGS algorithm has been exploited as a fourth optimization technique. It is an iterative method to solve unconstrained non linear optimization problems; it belongs to the quasi-Newton method class, a class of high-hill optimization techniques seeking the stationary point of a function. It avoids the most computationally expensive step of the Newton step, that is the direct calculation of the inverse of the Hessian matrix, by approximating it using updates specified by gradient evaluations. The mathematical algorithm of BFGS method is reported in section B of the Appendix.
It should be pointed out that, while for the three previously described optimizers the learning rates and the standard deviation for the weight initialization are tuned to similar values, for BFGS η is set to be one to two orders of magnitude bigger and the standard deviation for weight initialization is increased to approximately $\sigma = 1$, as it will be described in the subsection *Weight initialization* of the Results section.

F. Weight initialization

The influence of different weight initializations on the outcomes of the algorithm, in particular concerning the variance of the curves, has been investigated for the SGD optimizer through a sensitivity analysis. The learning rate has been fixed at η = 0.005.
Three types of initializations have been implemented:

1) Type I : it is the initialization technique implemented in the linear module of Pytorch, with the difference that the standard deviation is divided by $\sqrt{3}$. Thus:

$$\mathbb{V}(w^{(l)}) = \frac{1}{\sqrt{3}\ N_{l-1}} \tag{6}$$

2) Type II : Xavier initialization. It allows for a compromise between the control of the variance of activations and the control of the variance of the gradient wrt activations.

$$\mathbb{V}(w^{(l)}) = \frac{2}{N_{l-1} + N_l} \tag{7}$$

3) Type III : Normal initialization. The weights are initialized by following a normal distribution of mean μ and standard deviation σ.

IV. Results

The model has been trained with MSE by investigating the results for the four implemented optimizers.

A. Optimizers with MSE loss

The model has been trained with MSE loss function for a number of epochs up to 1000, setting the mini-batch size to 50.
The four optimizers have been tested: SGD, SGD with momentum (SGDmom), Adam, BFGS. A grid search on the

learning rate has been carried out in order to find a reasonable learning rate for each optimizer. The obtained values are of the same order of magnitude for SGD, SGD with momentum (SGDmom) and Adam (order of magnitude $\eta \simeq 10^{-3}$), while for BFGS the respective values are one to two orders of magnitude greater.
The results obtained for the loss and the accuracy behavior during training are plotted respectively in Figures 2 and 3. The

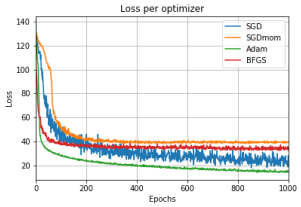

Fig. 2: Loss obtained with the four optimizers.

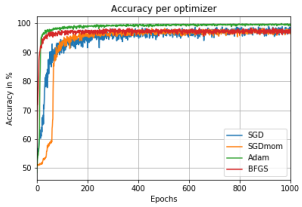

Fig. 3: Accuracy obtained with the four optimizers.

learning rates set for the simulation of the reported graphs are summarized for each optimizer in Table I. In the analysis of

TABLE I: Learning rates for the four optimizers used in the simulations of Figures 2 and 3

Optimizer	η
Vanilla SGD	0.005
SGD with momentum	0.005
Adam optimizer	0.005
BFGS algorithm	0.2

the graphs it is necessary to take into account that the reported curves have been obtained during one simulation only and that, especially for the BFGS, the variance characterizing the variations from one simulation to another is significant. For these reasons the inferences made from the graphs are not absolute and might slightly vary for different runs.
From the plots for the accuracy it can be seen that, on average, after approximately 500 epochs, vanilla SGD, SGD with momentum and BFGS tend to a similar value for the accuracy, with vanilla SGD fluctuating with a bigger variance. Adam optimizer leads to a slightly higher accuracy after a few tens of epochs. Furthermore, during the very first epochs, instead, the accuracy increases more steeply for BFGS. In particular, during the test for this run the obtained accuracies are summarized in Table II: It should be underlined that the

TABLE II: Accuracies obtained with the four optimizers

Optimizer	**Training accuracy**
Vanilla SGD	97.0%
SGD with momentum	96.5%
Adam optimizer	98.3%
BFGS algorithm	95.1%

graphs reported concern the training, while the accuracy results have been obtained during the testing. One could notice that in the graph for the testing Adam algorithm reaches an accuracy of 100%, while in the testing the accuracy is slightly smaller. This is due to the fact that Adam optimizer slightly overfits.

The analysis of the loss plot shows again that the minimal loss is achieved with Adam, and that only during the first few steps BFGS proved a better performance. As for the accuracy, the variance of the fluctuations is greater for vanilla SGD than for the other optimizers.
For the same values of the learning rate, the variance has been investigated over 10 runs. The graph in Figure 4 shows the variance obtained for the four optimizers. It should be emphasised that the reported plot has not been obtained with the optimal values of the learning rate: it is shown in order to represent the variance of the different optimizers.

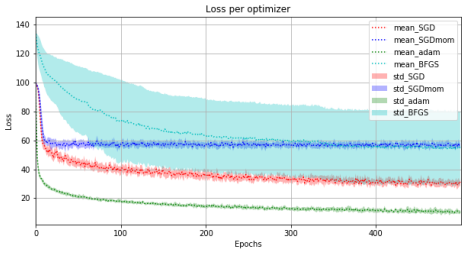

Fig. 4: Losses means and standard deviation over 10 runs on the same data for each optimizer.

their corresponding accuracies, which, after 500 epochs, are all between 90% and 100%, are reported in section D of the appendix. it should be underlined that BFGS is dependent on the initialization of the weights. This is probably due to the implementation of BFGS algorithm, in particular on the first step, which sets a value for the Hessian that is is then used for the rest of the optimization. The initial value of the weights is thus of central importance.
Adam confirms its supremacy also being the more stable optimizer w.r.t. the initialization of the weights.

1) Influence of mini-batch size: The outcomes reported above, as stated, have been obtained, for all the optimizers, with a fixed mini-batch size equal to 50. A further investigation for different sizes of the mini-batches has been carried out.
As expected, concerning the results obtained with vanilla SGD, as the mini-batch size decreases the performance improves; thus for mini-batch size equal to 1 the curves are smoother, their variance is smaller and the highest accuracy is reached after a smaller number of iterations. However the number of iterations per second decreases dramatically as the mini-batch size decreases. Table III reports the number of epochs per second for different choices of the mini-batch size. These values have been obtained on a HP elitebook 840 G4 on CPU (quad core).

B. Weight initialization

The loss curves obtained for the previously introduced weight initialization types with the SGD optimizer are reported in Figure 5. The learning rate has been set to $\eta = 0.005$. Concerning the Normal initialization, the graph reports the results obtained with two sets of parameters: the first one (in

TABLE III: Number of epochs per second for different mini-batch sizes

batch - size	# epochs/sec
1	7
50	225
100	330
500	700
1000	800

green) with $\mu_1 = 0$ and $\sigma_1 = 1$ and the second one (in light blue) with $\mu_2 = 0$ and $\sigma_2 = 0.01$.

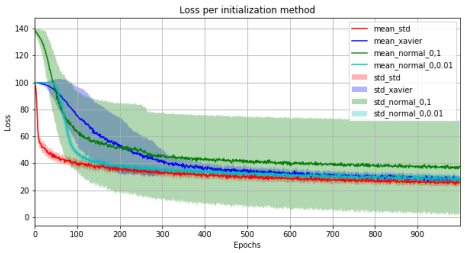

Fig. 5: Losses means and standard deviation over 20 runs on the same data different initialization. "std" stands as second argument for the PyTorch implementation with the standard deviation divided by sqrt(3)

As shown in the graph, all the initialization techniques lead, after approximately 1000 epochs, to the same value for the loss, except for the simulation run with normally initialized weights with $\sigma_1 = 1$ which converges to a higher value. From the Figure it emerges that the initialization of Type I reaches convergence faster.
The corresponding accuracies are reported in section D of the Appendix.

C. Comparison with results of PyTorch implementation

The performance of the network implemented from scratch has been compared with that of the Pytorch's implementation of a network exploiting the autograd and the nn.Modules.
The model employed in the former network has the ReLU activation function in the first and second layer, and Rectified tanh in the third; the chosen optimizer is Adam, with $\eta = 0.005$.
Figure 6 shows the accuracy curves for the Pytorch implementation and for the network implemented from scratch.

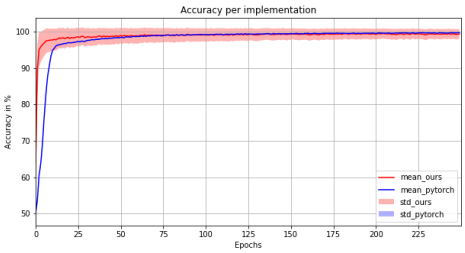

Fig. 6: averaged on 10 run over the same data

The plot shows that the two models reach the same mean for the accuracy, but the model implemented from scratch is 120% faster than the other, meaning that more iterations are computed per second. Furthermore convergence is reached in less epochs by our network with respect to the Pytorch one. However the variance for the former is greater than the one of the latter: this diversity could be explained as due to the different weight initialization employed by the algorithms.

V. Discussion

The results presented above allow to draw the following conclusions.
The optimizer giving the quickest attainment of convergence and the highest accuracy is Adam algorithm.
Concerning the SGD optimizer, instead, the best performance is achieved with the type I of weight initialization.
From the comparison with Pytorch implementation results, both in terms of speed to reach convergence and of accuracy attained, the implementation of the model from scratch proves to give comparable results. It could be interesting to see whether outcomes are as satisfactory if the same model is used for classification with another data set, for instance MNIST, and to compare whether also in this case the implementation from scratch is faster and allows more iterations per second than the Pytorch implementation.

VI. Conclusion

A neural network consisting of two input units, three hidden layers of 25 units, and two output units, has been implemented from scratch. The framework has given results comparable with those obtained with a Pytorch implementation, for four different optimizers when applied to a simple data set of points. A recommendation for a follow-up research would be to test the accuracy and the speed of the model on more complex datasets.
Further improvements could be obtained by extending the structure with more layers and by adding other modules. Additional techniques that could be investigated to enhance the performance are, for instance, weight decay and drop out. It could be interesting to implement automatic differentiation as well.
Finally, instead of the MSE loss, an alternative loss function could be employed: the Binary Cross Entropy. Indeed, while MSE loss could incorrectly penalize outputs valid for prediction, BCE loss provides for classification an alternative criterion not affected by this limitation.

VII. Appendix

A. Adam algorithm

Update rule:

$$w_{t+1} = w_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon}\hat{m}_t \tag{8}$$

where

$$\begin{aligned} m_t &= \beta_1 m_{t-1} + (1-\beta_1) g_t \\ \hat{m}_t &= \frac{m_t}{1-\beta_1^t} \\ v_t &= \beta_2 v_{t-1} + (1-\beta_2) g_t^2 \\ \hat{v}_t &= \frac{v_t}{1-\beta_2^t} \end{aligned}$$

B. BFGS algorithm

The step size is assumed to be constant, equal to the learning rate.
Update rule: [1]

$$w_{t+1} = w_t - \eta \mathbf{H}_t^{-1} \nabla \mathcal{L}(w_t) \tag{9}$$

with

$$\begin{aligned} \mathbf{H}_{t+1}^{-1} &= (I - (y_n^\top s_n)^{-1} y_n s_n^\top)\mathbf{H}_{t+1}^{-1}(I - (y_t^\top s_t)^{-1} s_n y_n^\top) + (y_t^\top s_t)^{-1} s_n s_n^\top \\ s_n &= w_t - w_{t-1} \\ y_n &= \nabla \mathcal{L}(w_t) - \nabla \mathcal{L}(w_{t-1}) \end{aligned}$$

This update ensure that $\mathbf{H}^{-1}$ is symmetric and that $\mathbf{H}^{-1} y_t = s_n$ (i.e. the secant condition)

C. Accuracies corresponding to the losses with variance

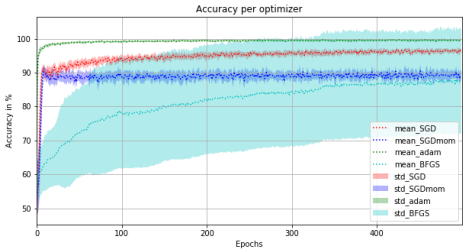

Fig. 7: Accuracies means and standard deviation over 10 runs on the same data for each optimizer.

D. Accuracies for the plot on initialization

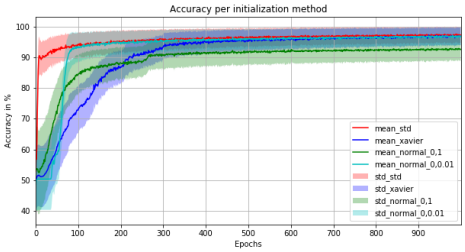

Fig. 8: Accuracies means and standard deviation over 20 runs on the same data different initialization. "std" stands as second argument for the PyTorch implementation with the standard deviation divided by sqrt(3)

E. Loss for PyTorch implementation

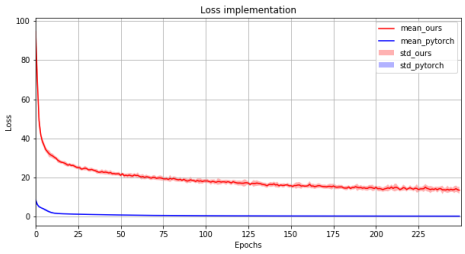

Fig. 9: averaged on 10 run over the same data

References

[1] *Numerical optimization: understanding L-BFGS*
Available at: http://aria42.com/blog/2014/12/understanding-lbfgs