Mini Deep-learning framework

Mauro Leidi, *Sciper:284713,* Denis Kravtsov, *Sciper:282379,* Sophie Du Couédic, *Sciper:260007,*

I. Introduction

The objective of this project is to design a mini "deep learning framework" using only PyTorch's tensor operations and the standard math library, hence in particular without using autograd or the neural-network modules. The designed framework should be able to:

1) Build networks combining fully connected layers, Tanh, and ReLU.
2) Run the forward and backward passes.
3) Optimize parameters with SGD for MSE.

In particular our deep learning framework had to be tested with the following conditions:

1) A training and a test set of 1,000 points sampled uniformly in $[0,1]^2$, each with a label 0 if outside the disk centered at (0.5,0.5) of radius $1/\sqrt{(2\pi)}$, and 1 inside
2) A network with two input units, one output unit, three hidden layers of 25 units
3) This network has to be trained with MSE, logging the loss
4) Final train and the test errors have to be logged and printed

II. Training algorithm

Our framework uses backpropagation algorithm to compute gradients for SGD. First, let's take a look at the implementation of SGD, we were following this algorithm:

Algorithm 1 SGD(ω_0, η)

```
1: for all epochs do
2:     Shuffle batches
3:     for all batches do
4:         for all samples in batch do
5:             w ← w - $\eta\nabla$w
6:             b ← b - $\eta\nabla$b
7:         end for
8:     end for
9: end for
```

Where η is the learning rate of the Neural Network, and ∇w and ∇b are the loss gradients with respect to weights and biases respectively. In order to calculate the gradients, we execute the following procedure:

1) Forward pass

Firstly, the forward pass is executed in order to compute the output values of the perceptron. It is done as following:

Algorithm 2 Forward pass

```
for each layer l in [1,L] do
    $s^{(l)} = w^{(l)}x^{(l-1)}+b^{(l)}$
    $x^{(l)} = \sigma(\mathrm{s}^{(l)})$
end for
```

Where σ is the activation function of choice (ReLU, Tanh, Sigmoid).

2) Backward pass

After having calculated the output of the perceptron, we calculate the loss, via the loss function of the choice (MSE, MAE, MBE) and its derivative with respect to the output $\left[\frac{\partial l}{\partial x^{(L)}}\right]$. Next we use the loss to find the gradients with respect to all parameters:

$$\left[\frac{\partial l}{\partial x^{(l)}}\right] = (w^{(l+1)})^T \left[\frac{\partial l}{\partial s^{(l+1)}}\right] \tag{1}$$

$$\left[\frac{\partial l}{\partial s^{(l)}}\right] = \left[\frac{\partial l}{\partial x^{(l)}}\right] \circ \sigma'(s^{(l)}) \tag{2}$$

$$\left[\!\left[\frac{\partial l}{\partial w^{(l)}}\right]\!\right] = \left[\frac{\partial l}{\partial s^{(l)}}\right] (x^{(l-1)})^T \tag{3}$$

$$\left[\frac{\partial l}{\partial b^{(l)}}\right] = \left[\frac{\partial l}{\partial s^{(l)}}\right] \tag{4}$$

3) Gradient step

After having calculated all the gradients we use them in the gradient update state.

$$w^{(l)} \leftarrow w^{(l)} - \eta \left[\!\left[\frac{\partial l}{\partial w^{(l)}}\right]\!\right] \tag{5}$$

$$b^{(l)} \leftarrow b^{(l)} - \eta \left[\frac{\partial l}{\partial b^{(l)}}\right] \tag{6}$$

III. Modules implementation

For the development of our framework, we followed the recommended structure. This structure consists of modules containing three methods: forward, backward and param. We created four different types of modules: activation functions, loss functions, linear model and sequential module. The role of all the different modules is explained below following a bottom-up approach, starting from the simpler blocks, and then going up in complexity. We will not mention anymore the param method which, for all the methods, only returns the list of parameters.

1) Activation functions

The activation functions modules are composed exclusively of **forward** and **backward** methods which are used to make the backward and the forward pass. These modules store input and output in memory. The idea is that the input will be the output of a linear model $Wx+b$ and the output would therefore be $\sigma(Wx+b)$. The forward pass is simply an application of the activation function. The backward pass applies the derivative of the activation function, which as seen in equation 2, is a necessary operation in the back-propagation algorithm. The following activation functions were implemented: ReLU and Tanh.

2) Loss functions

A second type of modules are the loss functions. These modules do not have the forward and backward methods, instead, they have the **loss_value** and **loss_grad** methods. The first one is used to calculate a quality metric of the predictions, while the second method is used to calculate the gradient of the loss with respect to activations. The following loss functions were implemented: MSE, MAE and MBE.

3) Linear layer

A third type of module is the linear layer. This layer is used to create a fully connected linear layer which, if put in series with an activation function, creates a complete layer of our neural network. This module store the weights, the biases, their gradients, the input and the output. The **forward** method contains the linear term: $y = Wx + b$, which if put in series with an activation function becomes $y_2 = \sigma(y) = \sigma(Wx + b)$. The **backward** method is used to compute the backward pass by appling three of the first four equations presented in order 1,4,3. The last method of the linear model is *gradient_update*. This method updates the weights and biases by performing one step of the stochastic gradient descent: w $\leftarrow$ w - $\eta\nabla$w and b $\leftarrow$ b - $\eta\nabla$b .

4) Sequential

The sequential module is used to compose the structure of a neural network composed of modules described previously. This module stores a list containing the layers that compose it and the loss. This module contains multiple methods.The method **add_layer** is used to add a module at the end of the current model. Modules are added sequentially, alternating linear layers with activation functions, hence the name Sequential.The forward method is used to make a forward pass for all layers in a sequential manner. The input of the next level is the output of the one that precedes it. This way, from the starting input, we get after the final layer, to the predictions. The backward method is used to call the backward passes of the individual modules in a sequential but reversed manner.This is necessary to the back propagation algorithm. The **gradient_update** method calls the gradient update methods of all linear layers so that one step of the stochastic gradient descent can be performed, by updating the weights and biases in the direction of the gradient of the loss with respect to the batch (and, if activated also with the momentum influence). The **predict** method is used to make a prediction on a dataset and compare it with the real outputs. This way we can calculate the accuracy that our model has achieved and the current loss value. The last method, and most important of all, is named **fit**. This method allows to perform the stochastic gradient descent and optimize the weights. This method can calculate the SGD as presented in algorithm 1: we first call the **sequential.forward** method, then compute the **loss_gradient** with the **loss.loss_grad** method to be able to call the sequential.backward method and finally we can do the **sequential.grad_update**.

IV. Additions to the network

A. Batch processing

In order to increase the training speed of our network, we have implemented the batch processing for our gradient descent. In order to use it, one has to specify the **minibatch_size**. The network, therefore, computes the gradient of the loss in batches instead of computing the complete gradient.

B. Batches shuffling

A randomized shuffling of batches is carried out between one epoch and the next. In this way the gradient directions of the batches are not identical for each epoch.

C. Momentum

Another addition to the gradient update step is the momentum, that adds inertia in the choice of step direction. It was implemeted as following:

$$u_t = \gamma u_{t-1} + \eta g_t \tag{7}$$

$$w_t = w_t - u_t \tag{8}$$

Where g_t is the weights gradient at moment t, η is the learning rate, γ is the momentum and w_t are the weights at time t. In order to activate it, the user has to specify **momentum** parameter value

D. Adaptive Learning Rate

Finally, we decided to implement the adaptive learning rate, as it can lead to improved accuracy, as we can avoid bouncing around the minimum for too large learning rate values. We are using the step decay of the learning rate. In order to use it, one has to specify **schedule** value.

V. Performance analysis on a simple test

For a fixed number of fifty epochs, we have optimized the learning rate and batch size in order to obtain maximum test accuracy. The values obtained are summarized in table I.

	Batch size	η	Scheduling	Inertia
Shuffling	5	0.001	0.9	0.5
No shuffling	5	0.001	0.9	0.6

TABLE I: optimal parameters (Shuffling)

The maximum accuracy on the test set obtained is 0.98.We plotted the correctly predicted values in green and the errors in red, as well as the circle of $\rho = 0.5$ in black.

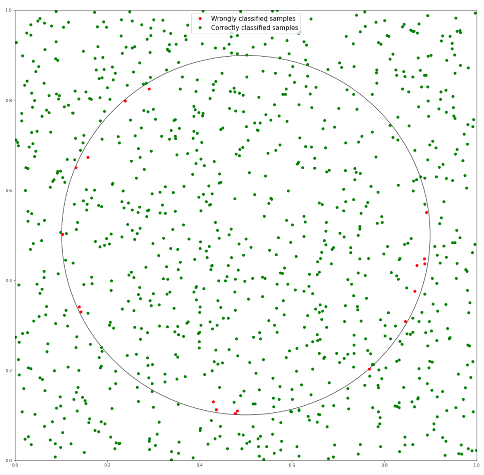

Fig. 1: Results of the prediction on test set

As we can see, the errors are found in the proximity of the circle and are mainly due to the fact that although the points follow a uniform law on [-1,1], the points distribution has higher concentrations in some areas and less dense areas in others. This is because our data-set has only 1000 points. Secondly we plotted the convergence of the loss values and of the accuracies.

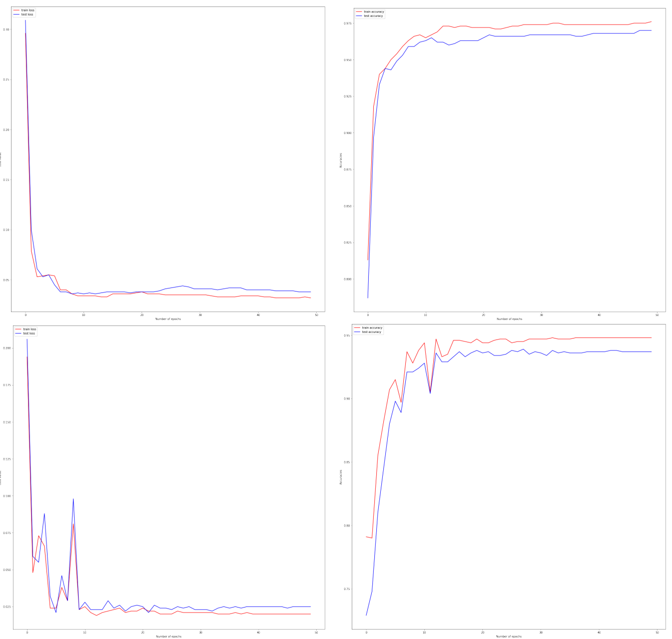

Fig. 2: Loss and accuracy plots for learning with and without shuffling(50 epochs)

We can see that the decrease in the loss is showing that twenty epochs are sufficient, because the loss reaches a plateau with and without shuffling. It can also be noted that the accuracies achieved are rather high and also converge to a certain value after around 20 epochs. The results show that our algorithm achieves the desired result, learning with high accuracy to understand if the points are inside the circle.

VI. Conclusion

We are satisfied with the result obtained in this project. We managed to develop a small framework capable of training neural networks with SGD with or without momentum, of using different loss and activation functions, and we have shown that it can achieve good performance on a simple test. This project allowed us to implement from scratch the SGD algorithm, already studied several times before, giving us a more concrete idea of how it works.