Deep Learning Mini Project 2

Authors: Gaia Carparelli, Leonardo Pollina, Hugues Vinzant

May 14, 2019

1 Introduction and Dataset

This second Mini Project aims at designing a "Deep Learning framework" using only `Pytorch` tensor operations and standard math library, therefore without using autograd or other neural network modules already implemented. This framework was assessed on a training and test set of 1000 points sampled uniformly in $[0,1]^2$ having label of value 0 if outside a disk of radius $\frac{1}{\sqrt{2\pi}}$ centered at [0,0] and of value 1 if inside the disk. Notice that, because of the particular values chosen for the generation of our dataset, there was no need to normalize the data, since every point lied in the same range, being [0,1].
In particular, the goal was to implement a neural network consisting of an input of dimension 2 (corresponding to the two coordinates of the points in the 2D space), three hidden layers of 25 units each and an output layer of 2 units (corresponding to the probability of belonging to the two classes). The general structure of such network is shown in Figure 1.

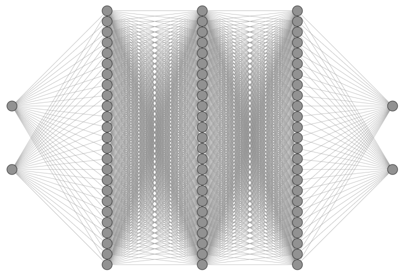

Figure 1: General architecture of the network to implement: input layers (containing 2 units), three hidden layers (containing 25 units each), output layer (containing 2 units).

2 Models and Methods

2.1 Linear (Fully-connected) layers

In the frame of this Mini-Project, only fully-connected linear layers were implemented. A fully-connected layer is a hidden layer in which all units (neurons) receive an input from every unit of the previous layer and send an output to every unit of the next one. It is called 'linear', since the forward pass computed in this layer consists simply in the weighted sum of the input received plus the bias. In addition, weights and the biases are updated in the backward pass of this layer. The optimization of such parameters was implemented through Stochastic Gradient Descent (SGD). The forward pass, the backward pass and the update of the weights are shown mathematically in the following formulas (where x is the input received by a linear layer):

Forward Pass:

$$s^{(l)} = w^{(l)} x^{(l-1)} + b^{(l)} \tag{1}$$

Backward Pass:

$$\left[\frac{\partial l}{\partial x^{(l))}}\right] = (w^{(l+1)})^T \left[\frac{\partial l}{\partial s^{(l+1))}}\right] \tag{2}$$

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

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

Gradient (update) step:

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

Therefore, the necessary parameters to define a linear layer are the weights (w), being stored in a matrix having a shape of $\#units_{output} \times \#units_{input}$ and the biases (b), being a vector of dimension $1 \times \#units_{output}$. Notice that, in order to use a mini-batch optimization, samples were treated in group, meaning that the layer was fed with a tensor of dimension $mini\_batch\_size \times \#units_{output}$. Thus, the implemented module **Linear** contains the functions *forward_pass()*, *backward_pass()*, *update()*, *param()* and *zerograd()*. *param()* returns the parameters of the layer and their gradients, while *zerograd()* sets the gradients of the parameters to zero (this was applied after each mini-batch backward pass was performed).

2.2 Activation functions

In Deep Learning, an activation function is a function applying the non-linearity to the output of the linear layer (in our specific case). Three different activation functions were implemented in our case: ReLU, Tanh and Sigmoid. The respective forward pass and the backward pass are shown in the equations below, where σ represent the specific activation function:

Forward Pass:

$$x^{(l)} = \sigma(s^{(l)}) \tag{6}$$

Backward Pass:

$$\left[\frac{\partial l}{\partial s^{(l))}}\right] = \left[\frac{\partial l}{\partial x^{(l))}}\right] \sigma' \odot (s^{(l)}) \tag{7}$$

2.2.1 ReLU (Rectified Linear Unit)

ReLU represents probably the most commonly used activation function nowadays [1]. The implemented module **ReLU** contains the functions *forward_pass()*, *backward_pass()*, *ReLU_fun()* and *d_ReLU()*. The forward pass just applies the ReLU function to the input received, while the backward pass applies the derivative of ReLU to the gradient received as input (during backpropagation). The ReLU activation function expression, as well as its derivative, is given in the following equations:

$$\sigma_{ReLU}(x) = max(0, x) \tag{8}$$

$$\sigma'_{ReLU}(x) = \begin{cases} 1 & \text{if x > 0} \\ 0 & \text{otherwise} \end{cases} \tag{9}$$

2.2.2 Tanh

In order to try another activation function, the module **Tanh** was also implemented. The use and the general principle of this module are exactly the same as the ones of the **ReLU** module. The only difference is that, instead of the rectifier function, the hyperbolic tangent function is employed as σ function as follows:

$$\sigma_{Tanh}(x) = \frac{2}{1 + e^{-2x}} - 1 \tag{10}$$

$$\sigma'_{Tanh}(x) = 1 - \sigma_{Tanh}(x)^2 \tag{11}$$

2.2.3 Sigmoid

Finally, a last activation function was implemented. Sigmoid was chosen as the activation function applied to our output layer. Indeed, this function maps all values in the space [0,1], hence giving the possibility to interpret the result as the probability to belong to a given class. The implementation of the module **Sigmoid** was again similar to the one of **ReLU** and **Tanh**. Only the σ function changed accordingly:

$$\sigma_{Sigmoid}(x) = \frac{1}{1 + e^{-x}} \tag{12}$$

$$\sigma'_{Sigmoid}(x) = \sigma_{Sigmoid}(x)(1 - \sigma_{Sigmoid}(x)) \tag{13}$$

2.3 Loss function

Even if the final goal of this Mini-Project was to perform a classification (whether the point is inside or not the disk with the given radius), the mean-squared loss (MSE), a classical loss function used in the case of regression task, was implemented and used. It is important to notice that, in order to do so, the target values were expressed in the one-hot encoding vector form. This means that if the ground truth class was the class 1, the target would be [0,1], while if the the ground truth class was the class 0, the target would be [1,0].
It was necessary to implement both the *LossMSE()* function, (computing the loss between the estimated labels and the ground truth ones) and its derivative *d_LossMSE()* (computing the gradient with respect to the output). These two functions are presented below (where x represent the output and t is the target):

$$MSE = \frac{1}{N}\sum_{i=1}^{N}(x_i - t_i)^2 \tag{14}$$

$$\frac{\partial MSE}{\partial x} = 2(\mathbf{x} - \mathbf{t}) \tag{15}$$

2.4 Sequential

Finally, a **Sequential** module was implemented in order to allow the creation of an arbitrary network by combining the previously implemented modules **Linear**, **ReLU**, **Tanh** and **Sigmoid**. This module contained the functions *forward_pass()*, *backward_pass()*, *update()* and *zerograd()*. All these functions simply call the respective functions of the specific modules by iterating through them all in a subsequent fashion (naturally the iteration is performed in a reversed order in the case of the backward pass). Notice that the derivative of the loss is computed inside the backward pass of the **Sequential**, as it is the first input of the backpropagation.

2.5 Training

In order to allow the optimization of our neural network, the implementation of a *train()* function was necessary. This function takes as inputs the model (previously built with **Sequential**), the train samples, their ground truth labels, the value of the learning rate, the size of the mini batch and the number of epochs. The training samples are divided in mini-batches according to the batch size given as input and a forward and backward pass are performed for each mini-batch. This means that the network parameters are updated after each mini-batch is passed forward and backward. At the end of each forward pass the loss is computed and the losses over all mini-batches are summed to keep trace of the loss behavior over the epochs (an epoch is performed when the whole dataset is gone through the neural net). After the backward pass has been called for a mini-batch, the weights are updated and the gradients are set to zero again (to not accumulate the gradients over the batches).

2.6 Validation of results

In order to be able to assess the performance of our network, a simple function *error()* was implemented. This function just computes the number of miss-classified samples.

2.7 Additional implementations

2.7.1 Xavier initialization

The weights of the linear layers could be randomly initialized according to a normal distribution having mean equal to zero and a very low standard deviation (in the order of 10^{-6} for example). However, it has been shown [2] that a more accurate weights initialization could be achieved by using the so-called Xavier initialization. This allows to train deeper networks avoiding the problem of the vanishing gradient. Therefore, the weights were initialized with a normal distribution with zero mean and standard deviation expressed in the following equation:

$$std = \sqrt{\frac{2}{(out_{features} + in_{features})}} \tag{16}$$

2.7.2 Dropout

Finally, also a **Dropout** module was implemented. The dropout is another common strategy used to allow the training of deeper network by decreasing the risk of overfitting. This is done by setting some input units to zero accordingly to a Bernoulli sampling. This means that a unit will be dropped with probability p and will be kept with probability *1-p*. In order to have the same range of input values in expectancy in the training phase and the test phase (in the test phase there is no dropout), the units which are kept are scaled with a factor $\frac{1}{1-p}$. Thus, the module **Dropout** simply contains the functions *forward_pass* and *backward_pass*, which set to zero some of the input units in the forward pass and keep trace of such sampling in the backward pass, respectively.

3 Results

In Figure 2 the evolution of the training loss over the increasing number of epochs is shown. The training and test percentages of errors obtained with the neural network with the architecture described in Table 1 were respectively of 2.5% and 1.9%. Finally, the classification results on the test dataset are displayed in Figure 3.

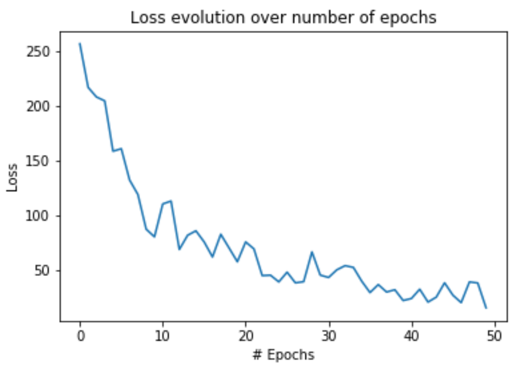

Figure 2: Evolution of the training loss over the increasing number of epochs (from 0 to 50).

4 Discussion

In order to check the correct functioning of the whole framework, the behaviour of the training loss over epochs was analyzed and it is shown in Figure 2. The trend was as expected, in fact the loss diminished until reaching a stagnating point. The training and test errors were very low, which was expected since the given dataset had the two classes well separated thanks to the clearly defined geometric arrangement.

	Type of layer
1	Linear(2,25)
2	ReLU()
3	Linear(25,25)
4	Tanh()
5	Dropout(p = 0.5)
6	Linear(25,25)
7	ReLU()
8	Linear(25,2)
9	Sigmoid()

Table 1: Structure of the Neural Network architecture used for training.

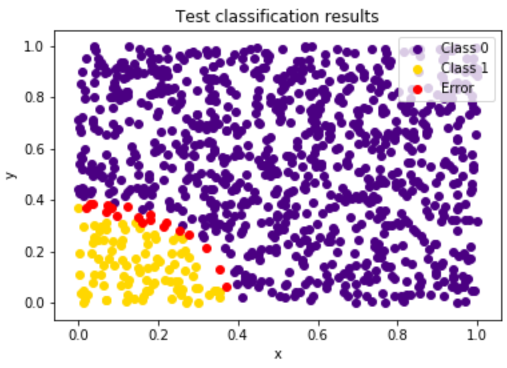

Figure 3: Test dataset distribution after the classification. In purple and yellow correctly classified points belonging to class 0 and 1 respectively. In red miss-classified points.

In addition, the miss-classified samples were identified and plotted with the correctly classified samples in order to highlight the most difficult region to classify. As expected, it turned out that these points lied on the border of the disk, which is the most critical region. Even if not shown in this report, the found results were validated by comparing them to the `Pytorch` Neural Network frame by using both the SGD and the Adam optimizer and the results were quite comparable.

Of course, our Deep Learning framework could be further developed by adding new modules and by improving the already existing ones. One idea could be to implement mini-batch normalization to avoid problems when training very deep neural networks. Moreover, a major improvement would be to implement *Convolutional layers* modules in order to allow even basic images classification tasks.

References

[1] LeCun, Y; Bengio, Y; Hinton, G (2015). "Deep learning". *Nature*, 521 (7553): 436–444.

[2] X. Glorot and Y. Bengio. Understanding the difficulty of training deep feedforward neural networks. In International Conference on Artificial Intelligence and Statistics (AISTATS), 2010.