Deep Learning Mini-Project 2 Report
Implementing from Scratch a Mini Deep-Learning Framework.

Group 78*
Caterina Bigoni and Nicolò Ripamonti

18 May 2018

1 Introduction

The goal of this project consists in designing a mini deep learning network frameworks "by scratch". We have therefore built a network that combines fully connected layers, Tanh, the logistic sigmoid and ReLu. Moreover, we have implemented a forward and backward pass as well as the update of parameters using the stochastic gradient descent (SGD). For this part we followed the mathematical framework described in the lecture notes.

This report is organised as follows: in Section 2 we describe the generalities of the implementation, explaining in particular the concept of dictionaries used to define the operators and their connectivity map. Then, in Section 3, we test the constructed network on a two-class classification problem both for a linearly separable dataset and a non-linearly separable one. Satisfactory results, i.e. from 0.5 to 2% error, are achieved in both cases. Conclusions and further remarks are presented in Section 4.

2 General Network

The library handles the network structure by means of dictionaries, the ideal type to represent graphs in Python language . The generic dictionary considered for this project can be written as

$$\begin{aligned}\texttt{Dictionary} = \{\ &\text{key}_1\ : [\ \text{value}_{1,1},\ \text{value}_{1,2}, \dots],\\ &\text{key}_2\ : [\ \text{value}_{2,1},\ \text{value}_{2,2}, \dots],\\ &\dots\\ &\text{key}_N\ : [\ \text{value}_{N,1},\ \text{value}_{N,2}, \dots]\},\end{aligned}$$

where for each key we may have an arbitrary (and possibly different) number of values. We use the list constructor $[\dots]$ to gather all the values related to a certain key because lists are containers easily iterable and we need this feature as explained later on.
The first dictionary that the user has to define is called `operators` in `test.py` and it records the modules used in the network. For example, let us consider a simple network, as the one represented in Figure 1. Its related dictionary is

$$\begin{aligned}\texttt{operators} = \{&1\ :\ \text{Linear1}\ ,\ 2\ :\ \text{Nonlinear1}\ ,\\ &3\ :\ \text{Linear2}\ ,\ 4\ :\ \text{Nonlinear2}\ ,\\ &5\ :\ \text{Sum}\ ,\ 6\ :\ \text{Linear3}\ \}.\end{aligned}$$

The next step is the definition of a `connectivity` dictionary: the rationale behind this dictionary is to describe the relations between each module and its followings in the natural ordering of the network

*As agreed with Dr. F. Fleuret, L. Pegolotti has collaborated with group 78 for Project 1 and with group 79, with M. Martin, for Project 2. C. Bigoni and N. Ripamonti have worked together on both projects.

from the inputs to the output. In this interpretation of Python dictionary, a key represents the module we are considering in a certain moment and the related values are other modules waiting for its output. Then `operators` dictionary is used to translate numerical values to proper modules.The `connectivity` matrix of the considered example would then be

$$\begin{aligned}\texttt{connectivity} = \{&1 : [\,2\,],\ 2 : [\,5\,],\\ &3 : [\,4\,],\ 4 : [\,5\,],\\ &5 : [\,6\,]\}.\end{aligned}$$

During the initialization of `Network` object, the `connectivity` matrix is translated into two additional dictionaries that are more useful for the forward and backward passes. Consider the forward pass first: the idea is that, given a certain module, we have to know the outputs of the previous modules in the graph, the exact opposite of what is offered by `connectivity`. Hence, after storing the necessary data, the constructor of `Network` generates the dictionary `connectivity_forward`. In the test case we have

$$\begin{aligned}\texttt{connectivity\_forward} = \{&2 : [\,1\,],\ 4 : [\,3\,],\\ &5 : [\,2,4\,],\ 5 : [\,6]\}.\end{aligned}$$

After `Python 3.60`, looping over a dictionary is done in the order in which the dictionary is defined but, to avoid unexpected behaviour with previous releases, we force this property by using the `OrderedDict` container provided by the `Python` module `collections`. While the structure of `connectivity` might work for the backpropagation of the gradients, we need to reverse the order and obtain

$$\begin{aligned}\texttt{connectivity\_backward} = \{&5 : [\,6\,], 4 : [\,5\,],\\ &2 : [\,5\,], 1 : [\,2\,],\\ &3 : [\,4\,]\}.\end{aligned}$$

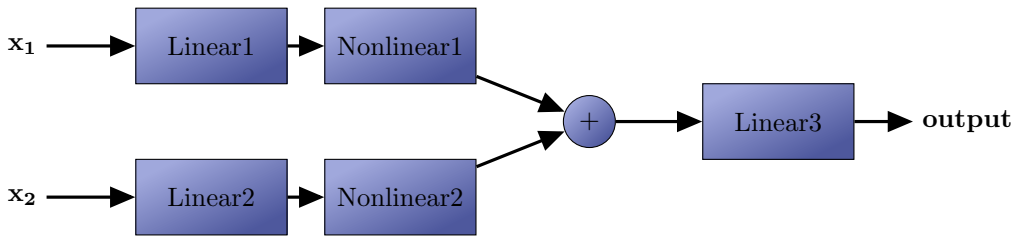

Figure 1: Example of network that can be represented using our library. In the graph, x_1 and x_2 represent possible inputs.

2.1 Modules

With the term *module* we mean each possible element of the network from linear or nonlinear layers to the loss function. Moreover, other operators such as the sum of two different inputs are treated as modules, and hence we obtain an simpler representation for more complex networks. The general structure of a module object is given in `ModuleBase.py` and in each derived object we have to define the following methods:

- `forward`: Implement the forward pass given the input of the module and store both the input and the output as attributes of the class.
- `backward`: Compute the gradient of the loss with respect to the input given the gradient of the loss with respect to the output. If the module contains parameters, it also computes the gradient of the loss with respect to them.

- `update_param`: Given a learning rate as input, updates the parameters of the module using a gradient descent based approach.

The reason behind this structure is that the for the optimization phase we are using a gradient descent based method. Given a functional $\mathcal{L} : \mathbb{R}^D \mapsto \mathbb{R}$, and a starting point $\omega_0 \in \mathbb{R}^D$, a local approximation of $\mathcal{L}$ allows to find a point

$$\omega = \omega_0 - \eta \nabla \mathcal{L}(\omega_0)$$

such that, if η is small enough, $\mathcal{L}(\omega) < \mathcal{L}(\omega_0)$.
In our case ω represents the set of parameters related to each module and $\mathcal{L}$ a functional measuring the prediction error of the method. To use this in a neural network code we need to understand how the gradient of the loss $\mathcal{L}$ with respect to the parameters is computed. The action of a generic σ-parametrized module on an input signal x^{l-1} to produce the output x^l can be written as

$$x^l = f(x^{l-1}; \sigma).$$

Thanks to the chain-rule of differentiation, we have

$$\nabla_\sigma \mathcal{L} = \nabla_{x^l} \mathcal{L} \left(\nabla_\sigma x^l \right)^T, \tag{1}$$

where $\nabla_\sigma x^l$ depends on the specific function. Hence, $\nabla_\sigma \mathcal{L}$ can be computed if the module receives as input $\nabla_{x^l} \mathcal{L}$.
The next step consists in the backpropagation of the information regarding the derivative of the loss function with respect to the input, so that previous models can compute $\nabla_\sigma \mathcal{L}$. Using again the chain-rule for differentiation we get

$$\nabla_{x^{l-1}} \mathcal{L} = \left(\nabla_{x^{l-1}} x^l \right)^T \nabla_{x^l} \mathcal{L}. \tag{2}$$

If f is a nonlinearity such that x_i^l depends just on x_i^{l-1}, then $\nabla_{x^{l-1}} x^l$ is a diagonal matrix and (2) is reduced to a pointwise multiplication and we exploit this in the code.

2.1.1 Linear Layer

In our code the linear layer, generically represented by the operation

$$x^l = Ax^{l-1} + b,$$

with $x^{l-1} \in \mathbb{R}^{n_{inp}}, x^l \in \mathbb{R}^{n_{out}}, b \in \mathbb{R}^{n_{out}}, A \in \mathbb{R}^{n_{inp} \times n_{out}}$, is the only one containing parameters. Since gradient descent is implemented using a stochastic mini-batch approach, we adapted (2) and (1) to have a vectorized code. A common choice is to have the number of samples as first dimension of the input, hence each contribution of relations (2) and (1) has to be transpose. Particular attention has been paid to the update of $\nabla_\sigma \mathcal{L}$: suppose that we are considering batches of size n_b, (1) is given by

$$\nabla_A \mathcal{L} = \left(\nabla_{x^l} \mathcal{L} \right)^T X^{l-1},$$

with $X^{l-1} \in \mathbb{R}^{n_p \times n_{inp}}$. Since in this way in $\nabla_A \mathcal{L}$ we compute the sum of the gradients, we rescale by n_p to have the average.

2.1.2 Nonlinear Activation functions

As previously mentioned, for nonlinear operators, equation (2) is simplified to

$$\nabla_{x^{l-1}} \mathcal{L} = \operatorname{diag}\left(\nabla_{x^{l-1}} x^l \right) \circ \nabla_{x^l} \mathcal{L}.$$

We implemented 3 different types of nonlinear activation.

- Rectified Linear Unit:

$$x^l = f(x^{l-1}) = \begin{cases} x^{l-1}, & \text{if } x^{l-1} > 0, \\ 0, & \text{otherwise.} \end{cases} \qquad \operatorname{diag}\left(\nabla_{x^{l-1}} x^l \right) = \begin{cases} 1, & \text{if } x^{l-1} > 0, \\ 0, & \text{otherwise,} \end{cases}$$

- Logistic Sigmoid:

$$x^l = f(x^{l-1}) = \frac{1}{1-e^{-x^{l-1}}}, \qquad \text{diag}\left(\nabla_{x^{l-1}} x^l\right) = f(x^{l-1})(1-f(x^{l-1})),$$

- Hyperbolic Tangent:

$$x^l = f(x^{l-1}) = \tanh(x^{l-1}), \qquad \text{diag}\left(\nabla_{x^{l-1}} x^l\right) = 1-(\tanh(x^{l-1}))^2,$$

2.1.3 Sum operator

The sum operator does not have any parameter and is simply described by

$$x^l = x_1^{l-1} + x_2^{l-1} + \cdots + x_N^{l-1}$$

Moreover,

$$\text{diag}\left(\nabla_{x_i^{l-1}} x^l\right) = 1, \qquad i = 1, \ldots, N.$$

3 Numerical Experiments

We run a simple "sanity check" test to verify if our setup behaves correctly. We first consider a simpler classification problem: a two-class problem where the discriminant line that separates points classified with label 0 or 1 is linear. In particular, we sample points from a standard Normal distribution and assign them a soft label 0.8 or 0.2 if they lie on the left or on the right side of a line, respectively. The line is fixed with intercept in the origin and slope equal 4. To train this problem, we use a very simple network where the operators and connectivity map are chosen as follows:

$$\text{operators} = \{1 \ : \ \text{Linear } \}, \quad \text{connectivity} = \{1 \ : [\ ]\}$$

As shown in Figure 2, even a very simple network of this kind can provide satisfactory results on unseen data: 0.3% error on the training dataset and 0.5% error on the test set.

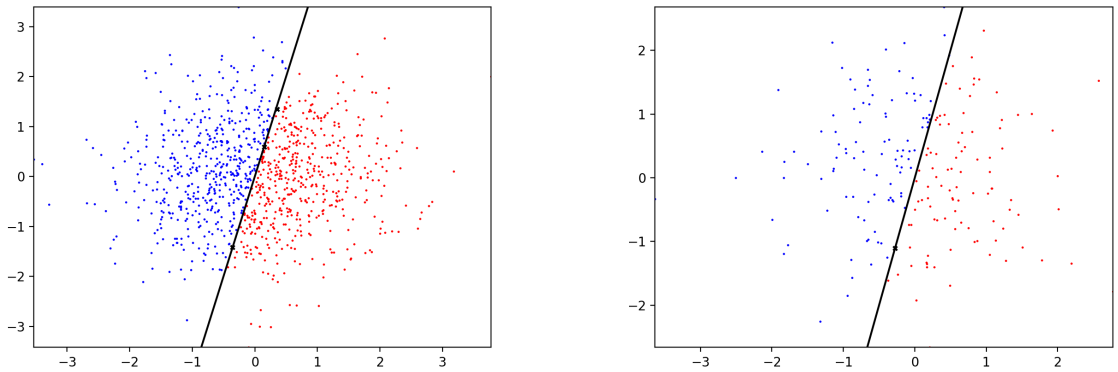

Figure 2: Training (left) and test (right) sets for the first simple model. The discriminant between the two classes is a line, points belonging to class 0 and 1 are marked in blue and red, respectively. Points misclassified are marked with a black x symbol.

For this problem and all the following once we sampled 1000 points in the train set and 200 points in test set. The learning rate is set equal to 0.05 for all problems and we run it for 1000 epochs.

As one could have expected, this very simple linear model does not generalise to more complex classification problems, where there discriminant plane is non linear, as for example the dataset required

for this project. After randomly generating points in $[0,1]^2$ (sampled from a uniform distribution) with label 0 if they lie inside the circle of radius $\frac{1}{\sqrt{2\pi}}$ or 1 otherwise, we test the classifier using the simple linear model presented above. Figure 3 shows the results on an unseen test set. All the points close to circle, the discriminant border, are misclassified (black cross) leading to an error of 10.2% in the training set and 11% in the test set.

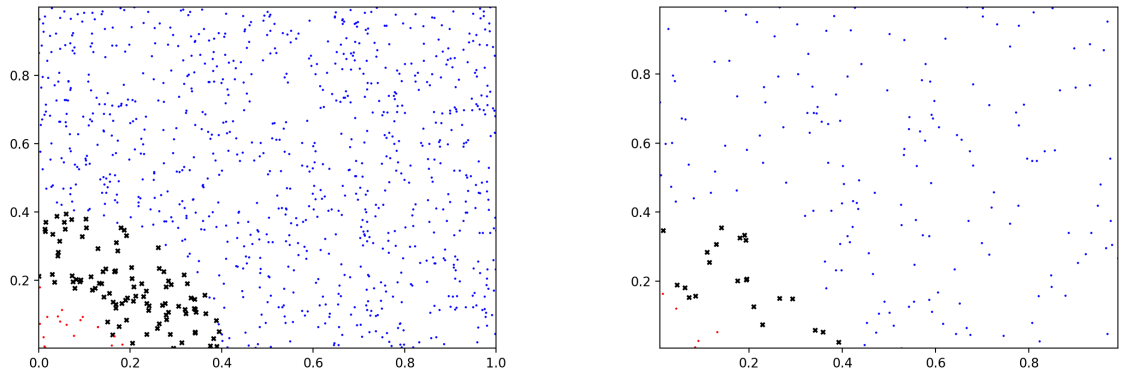

Figure 3: Training (left) and test (right) sets for the first simple model on the requested data. The discriminant between the two classes is a circle, points belonging to class 0 and 1 are marked in blue and red, respectively. Points misclassified are marked with a black x symbol.

It seems evident that for this setup a network with more capacity and nonlinear activation functions is needed to increase the accuracy. After this preliminary set, we are ready to test the performance of the model described in the assignement, characterized by 3 hidden layers of dimension 25. See Figure 4 for qualitatively results on training and test sets. Here we obtain 0.5% error on the training set and 0% error on the test set. These results are very satisfying and confirm the well behaviour of the model we implemented.

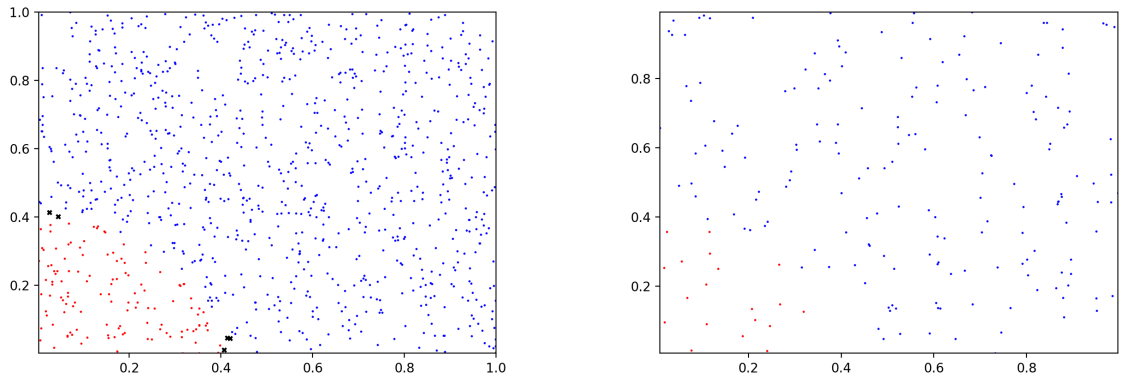

Figure 4: Training (left) and test (right) sets for the non-linear discriminant problem using the model described in Section 2. Points belonging to class 0 and 1 are marked in blue and red, respectively and points which are misclassified are marked with a black x symbol.

4 Conclusion

We developed a framework for Neural Network and tested it on different toy problems. The interface of the code, in particular the design of the network, is easy to understand from the user point of view and permits several different configurations with minor changes with respect to the test case. Even though the structure of the code allows only "converging" graph, and not general acyclic structure, the possibilities offered by our approach go beyond simple sequential networks. Standard nonlinear activation functions have been tested, showing results that are in accordance with the expected ones. Simple networks seems to give an accurate solution for trivial classification problems while more articulate and long structures seem necessary as the complexity grows.