Implementation of a Neural Network in pure Python

Nicola Dainese

I. INTRODUCTION

In this work I present my implementation of a Neural Network (NN) with two layers and a framework to search systematically for the best architecture and hyper-parameters (HPs) between some possible options defined with some a priori knowledge and heuristics. Finally I present the best model found and its performance on a simple dataset.

II. NEURAL NETWORK IMPLEMENTATION

The Neural Network is implemented in Python as a class. I expanded the attributes and the methods of the original class. The main differences are:

1. The __init__ method takes more parameters in input (discussed in the following subsections);
2. There is the possibility to save and load the weights of the network;
3. The training is now done with an internal method; this is more concise and it's useful because the network once initialized already has all the parameters that need to be used during the training,
4. There is the possibility to evaluate the performance of the model with the evaluate_mean_loss method and with a flag one can also save the predictions in a .txt file.

Here below is reported an overview of the various methods implemented and of the parameters that they accept as inputs.

```
class Network():

def __init__(self, N_neurons, act_func, \
         act_der_func, lr, n_epochs, \
         en_decay=True, lr_final=1e-4, \
         early_stopping = True, tol = 1e-3, \
         en_penalty=False, penalty=1e-4, \
         en_grad_clipping=True, grad_treshold=10)

def load_weights(self, WBh1, WBh2, WBo)

def save_weights(self)

def forward(self, x, additional_out=False)

def update(self, x, label)

def train(self, x_train, y_train, x_val, y_val,\
          train_log=False, verbose=False)

def evaluate_mean_loss(self, x_test, y_test, save=False)

def plot_weights(self)
```

A. Activation functions and gradient clipping

The first upgrade that I made to the network was to leave as a parameter the activation function that the two hidden layers use; outside the class in addition to the sigmoid I implemented the rectified linear unit (ReLU) and the leaky ReLU.

With those two activation functions I experienced the problem of the exploding gradient and solved it by applying the gradient clipping technique: if the norm of the whole gradient is greater than a certain threshold, I rescale all the components of the gradient dividing by its norm and multiplying by the threshold value, so that the direction of the gradient is maintained, but the module is equal to the threshold. This is equivalent of adapting the learning rate, so that we do not change the parameters with steps that are too large.

B. Regularization

The second improvement was to add the possibility of using a regularization term in the loss function; I implemented both L1 and L2 regularization.

In the case of the L1-penalty we get that the loss function becomes:

$$L_{L1} = \frac{1}{2}(Y - \hat{Y})^2 + \lambda \sum_{i,j,a} |W_{ij}^{(a)}| \tag{1}$$

where $a = h1, h2, o$ is the layer index (first and second hidden layers and output layer). Thus there is an additional term in the back-propagation:

$$\frac{\partial L_{L1}}{\partial W_{ij}^{(a)}} = (Y - \hat{Y}) + \lambda \cdot sgn(W_{ij}^{(a)}) \tag{2}$$

Instead for the L2-penalty the loss function is:

$$L = \frac{1}{2}(Y - \hat{Y})^2 + \lambda \sum_{i,j,a} (W_{ij}^{(a)})^2 \tag{3}$$

And the back-propagation with respect to a weight $W_{ij}^{(a)}$ becomes:

$$\frac{\partial L}{\partial W_{ij}^{(a)}} = (Y - \hat{Y}) + 2\lambda \cdot W_{ij}^{(a)} \tag{4}$$

C. Early stopping

Finally the last change made to the Neural Network was the introduction of the early stopping in the training. This was difficult to tune, because it should stop training when the validation (or training) error starts to increase due to over-fitting, but if there are oscillations during learning it is possible to stop too early the training. After some trials I decided to wait at least 100 epochs before enabling early stopping and it works as follows: if the minimum validation loss of the last 20 epochs is greater than the mean validation loss of the previous 20 epochs minus some specifiable tolerance, then either stop training if the learning rate is adaptive or adapt the learning rate (if it's constant) until it is smaller or equal than 10^{-4}, then stop. This specification is a bit conservative, but at least is robust. Moreover this can be tuned adjusting the tolerance: the greater the tolerance, the less the algorithm is conservative in stopping. In figure 1 is reported an example of the early stopping algorithm.

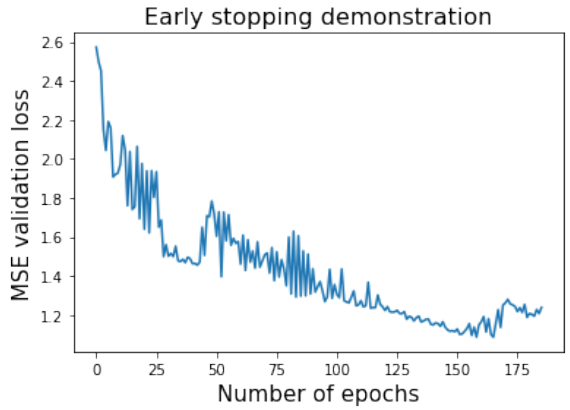

FIG. 1: Training stopped when the MSE on the validation set starts to increase. This method is quite robust to fluctuations in the descending part, yet it is able to understand when to stop properly.

III. MODEL SELECTION AND TRAINING

We have seen in the previous section that the model has a lot of hyper-parameters (HPs) to be specified before being trained, but we would like to know which combination of HPs is the best. Since the number of combinations grows exponentially in the number of parameters, we can not think of trying out all of the possibilities, but just the most promising ones (that could be hundreds of combinations anyway).

My choice was to explore manually some HPs, like the training rate, the number of epochs and the range of sensible penalties for the regularization and then use a search grid with 5-fold cross validation to extract the best model given the remaining combinations of parameters. It is inevitable that a lot of heuristics is involved in my choices, thus of course the final model won't be the best in absolute, because a lot of options were excluded given the computational cost of the grid search. The K-fold cross validation is done as follows:

1. Split training data in K subsets;
2. Use K-1 subsets to train and 1 to validate (like a test set);
3. Cycle over all K subset using each time a different validation set and store the validation losses.

In the grid search I evaluate each combination of HPs with the K-fold cross validation and finally rank the hyper-parameters using the mean and the standard deviation of their validation losses. In particular I choose to use as a final objective function to be minimized the mean plus two times the standard deviation, because I would prefer a slightly more imprecise model than an unstable one. One of the notable things of my implementation of the search grid is that works for a general model (and not only for our Network class) with some hyper-parameters, as long as that model has a "train" method with the following input parameters:

```
def train(self, x_train, y_train, x_val, y_val,\
          train_log=False, verbose=False)
```

Instead the greatest critical issues with respect to state-of-the-art grid search are the lack of parallelization, that could speed up significantly the search and the lack of a random search option.

After the grid search, the model that was ranked first is retrained using all the training set and evaluated with the test set (this final part is just a check and does not influence the model performance in any way). Then I save the parameters used and the weights of the NN so that they can be loaded by a new script used only to make predictions and to evaluate their accuracy.

IV. OBSERVATIONS AND HEURISTICS

1. I noticed that, all other things kept constant, changing the number of neurons of the two hidden layers did not have a clear effect on the performance of the NN, as reported in figure 3. Hence I could not restrict much the options a priori (but also I shouldn't have missed a super-performing architecture);
2. Given enough epochs (e.g. 2000), even small learning rates (between 10^{-3} and 10^{-4}) reach more or less the same accuracy of greater learning rates; conversely the greater ones can destabilize

more easily the training, particularly in the case of ReLU and Leaky ReLU activation functions. After some experiments I decided to use a starting learning rate of $5 \cdot 10^{-3}$ exponentially decaying to 10^{-4};

3. Again with some heuristics I choose the threshold at which the gradient gets clipped to be equal to 10.

4. Finally, one of the most influencing factors was the penalty term: as can be seen in figure 2, setting it to 10^{-2} was clearly too much, because even complex networks did not fitted properly the test set. Thus I leaved two possibilities, either 10^{-3} or $5 \cdot 10^{-4}$.

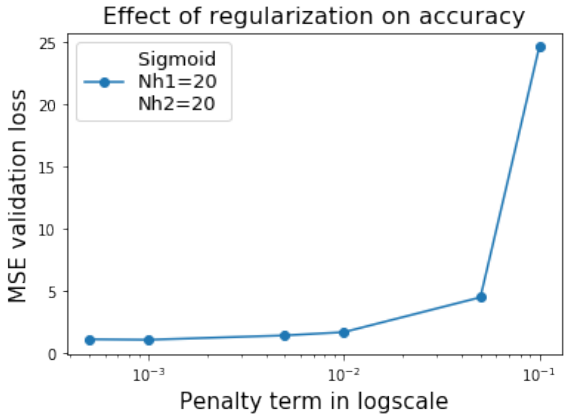

FIG. 2: The validation loss is a monotonically increasing function of the learning rate. For values equal or smaller than 10^{-3} it seems to stabilize a little and that is the range of values I used for the penalty.

V. RESULTS

The final model has 20 neurons in the first hidden layer and 10 in the second one, uses a ReLU activation function and has been trained for 599 epochs. That is the average number of epochs of effective training during the cross validation, where I used the early stopping option with a maximal of training epochs of 2000 and a tolerance of 10^{-3} in the stopping criteria. The model uses an L2 regularization with penalty 10^{-3} and an adaptive learning rate starting from $5 \cdot 10^{-3}$ and decaying to 10^{-4}. The final mean square error (MSE) achieved on the test set is of 0.8406. I reported all the hyper-parameters used on table I and a visualization of the learned function in figure 4.

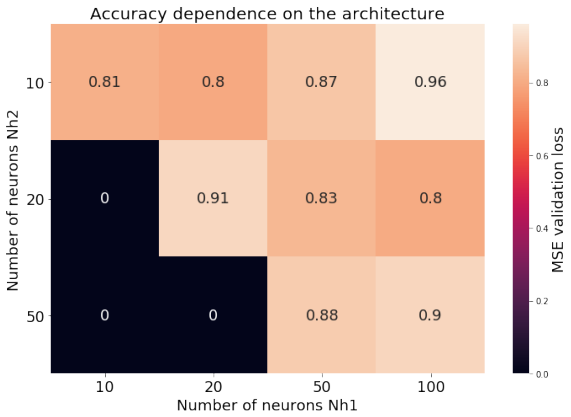

FIG. 3: The validation loss for different combinations of neurons of the two hidden layers. The cells with value zero are those that I do not consider in the training because $Nh2 > Nh1$. The key point here is that there is no clear trend in the loss with the number of neurons.

Hyper-parameters	
HP	Value
Number of neurons	[1, 20, 10, 1]
Activation function	ReLU
Learning rate	$5 \cdot 10^{-3}$
Adaptive learning	True
Final learning rate	10^{-4}
Number of epochs	599
Early stopping	False
Tolerance	10^{-3}
Type of penalty	L2
Penalty weight	10^{-3}
Clipping threshold	10

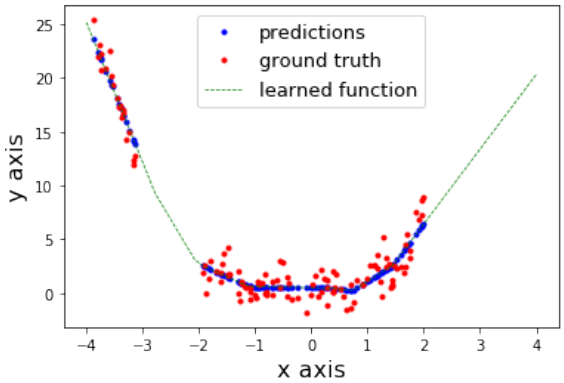

FIG. 4: Visualization of the training set and of the function learned by the neural network.