Fully Connected Feed-Forward Neural Network From Scratch in Python

Matteo Guida[†]

[†] *Physics of Data M.D. - Dipartimento di Fisica e Astronomia G. Galilei, Università degli Studi di Padova, Via Marzolo 8, I-35131 Padova, Italy.*

January 10, 2021

1 Introduction

In the present work a fully connected neural network and the search for the best model class among the ones defined by different configurations of the selected hyperparameters is implemented from scratch in "pure" Python. The model is trained in order to tackle a regression problem, i.e. the approximation of an unknown function, in the analyzed case $f(x) : \mathbb{R} \longrightarrow \mathbb{R}$. In the first part an overview of the implementation of the program and choices made is given. In the second part, some results related to the best found hypothesis are presented.

2 Program description

The assignment is solved through one python script FFNN.py and a jupyter notebook EX1_Matteo_Guida.ipynb. Moreover in the folder trained_model the script trained_model.py can be found which write the requested results on the mse.txt file. In the script FFNN.py the class Network is defined with the initialization and other 16 methods.

```
class Network():
def __init__(self, params_dict):
def set_network(self):
def forward(self, x, additional_out=False):
def update(self, x, label):
def relu(self,x):
def grad_relu(self,x):
def elu(self,x,alpha=0.9):
def grad_elu(self,x,alpha=0.9):
def leaky_relu(self,x, alpha=0.01):
def grad_leaky_relu(self,x, alpha=0.01):
def swish(self,x):
def grad_swish(self,x):
def save_model(self):
def load_model(self, WBh1, WBh2, WBo):
def train(self, x_train, y_train, x_val,
y_val,params_dict, early_stopping=True,
return_log=False, out_log=False,save=False):
def Kfold_cross_val(self, x_data, label,
    params_dict,k_fold=6):
def grid_search(self, params_dict,x_data,
    label):
```

Following an up-down approach to describe them, firstly, in the jupyter EX1_MatteoGuida.ipynb. an object of the class Network is instantiated passing a dictionary, where each element is a list with all possible values that the hyperparameters can assume during the **grid search**. At first, the object is instantiated assigning to the hyperparameters the first value of each list related to them. Than the grid_search method is called and it creates all the possible combinations achievable from the lists present in the dictionary and it starts considering each of them in a loop. After that, for each configuration a standard **k-fold cross validation** is implemented through the function Kfold_cross_val. The number of non-overlapping subsets k is fixed to 6, it is decided by any method except making some trial and balancing execution time with a reasonable division between the k-1 folds on which the training is performed and the remaining one on which the validation error is computed. For each iteration of the procedure the train method is called. The choices made in the Lab_02.py script are preserved so a decay of the learning rate is present and **mean square error** (MSE) $\text{MSE} = \frac{1}{2n}\sum_{i=1}^{n} ||y_i - \hat{y}_i||^2$ is used as loss function.
Moreover an **early stopping** is added with the following condition: the training cannot stop before the first 200 epochs. After that, if in the last 200 epochs an improvement of the validation loss of 5% does not occur w.r.t. the previous 200 epochs the training is stopped.
An important aspect in the different model classes is the **activation function**, which determines the output of each neuron given an input. A wide variety of functions can be used for neural networks and in the present work just a subset among the ones known for good performance are evaluated: ReLU, Leaky ReLU, ELU, Swish. An exhaustive analysis of the difference among them is beyond the scope of this report, but we can underline that all this function for an input $x > 0$ have an output which grows increasing x, while for $x < 0$ ReLU returns 0, instead all the other a negative output.
In addition, in the function update are included **gradient clipping** and $\mathbf{L^1}$ - $\mathbf{L^2}$ regularization, a.k.a. respectively Lasso and Tikhonov regularization. The first technique is introduced because the exploding gradients problem is found and involves the computation of the norm of the gradient w.r.t. the vector of the weights of each layer. If the result exceed a fixed *threshold* the gradient is rescaled so that its value does not fall out-

side the range [-clip_norm, clip_norm]. In this way the parameter update has the same direction of the true gradient but it avoids performing a detrimental step when the gradient explodes.
So if $|g(\boldsymbol{\theta})| > \text{clip\_norm}$:

$$g(\boldsymbol{\theta}) = \frac{1}{n}\sum_{i=1} \nabla_{\boldsymbol{\theta}} \ell_{\boldsymbol{\theta}}(\mathbf{x_n}, y_n), \quad g(\boldsymbol{\theta}) \longleftarrow \frac{g(\boldsymbol{\theta})\text{clip\_norm}}{|g(\boldsymbol{\theta})|}.$$

The second and third techniques involve the add of a regularizer $\Omega(\boldsymbol{\theta})$ and a new hyperparameter α which weights the relative contribution of the new term to the loss function:

$$L(\boldsymbol{\theta}, \mathbf{X}, \mathbf{y}) \longleftarrow L(\boldsymbol{\theta}, \mathbf{X}, \mathbf{y}) + \alpha\Omega(\boldsymbol{\theta}).$$

In the case of L^1 and L^2 we have respectively $\Omega(\boldsymbol{\theta}) = ||\boldsymbol{\theta}||$ and $\Omega(\boldsymbol{\theta}) = 1/2||\boldsymbol{\theta}||^2$. Consistently with this change we need also to modify the update rule for the gradient:

$$\nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta}, \mathbf{X}, \mathbf{y}) \longleftarrow \nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta}, \mathbf{X}, \mathbf{y}) + \alpha\nabla_{\boldsymbol{\theta}}\Omega(\boldsymbol{\theta}).$$

Those procedure are implemented mainly to prevent overfitting, penalizing the more complex models and in a certain sense applying the principle of Ockham's razor.

The **number of epochs**, i.e. the number of times the forward-backward and update procedure is repeated, is fixed to $2\,10^3$. This is often an exaggerated value for the considered learning rates, but we always apply early stopping procedure, so that the most of the times the training stops before.

The function *train* can return the train/validation loss for each epoch or just their last values. After finishing the procedure related to the k-fold, we have k values of the results of the loss function, evaluated for each of the k iteration on a different set. Averaging them we have a good estimate of the validation error and following the empirical risk minimization principle at the end we select the hypothesis class, in other words the configuration of the hyperparameters, for which the lowest validation error is obtained.

Once the best hypothesis class is chosen, the model is retrained using the entire training set, no more further divided in training set and validation set, and at the end the true error is estimated using the test set, never involved in the choice of the hypothesis.

3 Results and Comments

In table 1 the hyperparameters used for the grid search are reported. The choices are made after having performed some trails in order to select reasonable possibilities, in fact we want to limit them because the number of combinations grows drastically with the possible values that each hyperparameter can assume. So the final choices are limited in order to complete the model selection in a feasible time (of the order of three hours) which allows to select a class of hypothesis among 320 combinations. The architecture with two hidden layers proposed in Lab_02.py is preserved.

# Neurons	[[1,10,10,1],[1,15,10,1], [1,10,15,1], [1,20,10,1], [1,10,20,1],[1,20,20,1], [1,50,20,1],[1,20,50,1],[1,50,50,1]]
Activation	[ReLU, Leaky ReLU, ELU, Swish]
# Epochs	2000
Decay Learning rate	True
Start Learning rate	$[10^{-2}, 5\,10^{-3}]$
Final Learning rate	$[10^{-3}, 10^{-4}]$
Regularization	[L1,L2]
Penalty Parameter	$[10^{-3}, 5\,10^{-4}]$
Gradient Clipping	True
Norm Clip	20

Table 1: Possible values of the hyperparameters.

As mentioned before, the number of epochs are virtually always excessive and the training stops with the early stop condition, however a conservative choice is made with small learning rate in order to guarantee an accurate learning. Thanks to exponential decay is even more true passing the epochs. The number of neurons in the hidden layers does not seem to impact significantly on the final performance, so the complexity of the simpler models is sufficient to tackle the problem. After some trials the norm clip is set to a value higher than the mean of the gradients, but not too much in order to reduce the step size in the gradient descent. The penalty parameter in the regularization is fixed to quite small values because otherwise underfit problems are encountered.

In table 2 the best model among the possible combinations set by table 1 is shown.

# Neurons	[1,10,20,1]
Activation	LeakyReLU
# Epochs	2000
Decay Learning rate	True
Start Learning rate	10^{-2}
Final Learning rate	10^{-3}
Regularization	L2
Penalty Parameter	10^{-3}
Gradient Clipping	True
Norm Clip	20

Table 2: Hyperparameters of the final model.

In figure 1 the learning curve for the selected hypothesis class is presented. We retrain it over the entire training set no longer with the validation set splitting and we use the test set to estimate the generalization error. The large test loss oscillations present in the first epochs is maybe due to the fact that the model has not yet learnt sufficient information to make predictions.

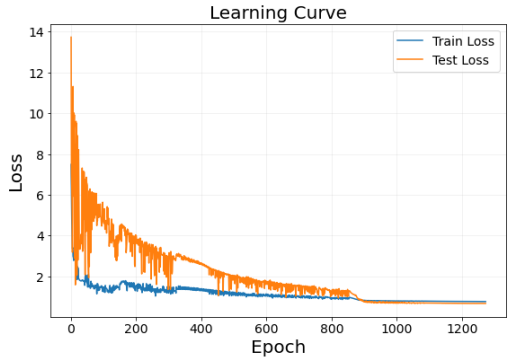

Figure 1: Learning curve for the selected hypothesis.

The distribution of the weights for each layer are also reported in figure 2.

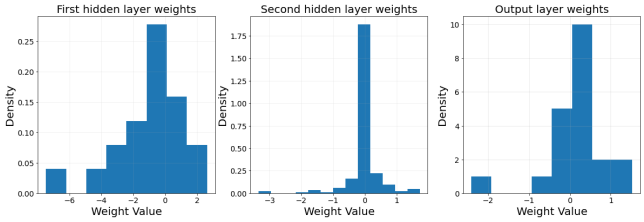

Figure 2: Weights distributions for each layer of the network.

Honestly, I have not an enlightening interpretation of those distributions and not even an expected behaviour in mind. However, according to my current knowledge, there is not some expected values for the weights because they would be related to the interpretation of the model, which is currently an active field of research in ML.

The final mean square error obtained on the test set is:

$$\boxed{\text{MSE} = 0.68 \pm 0.74,}$$

where the added error is the standard deviation. The regression curve obtained with the final model in the range of the inputs in the training and test set is shown in figure 3, to the eye satisfactory agreement can be observed.

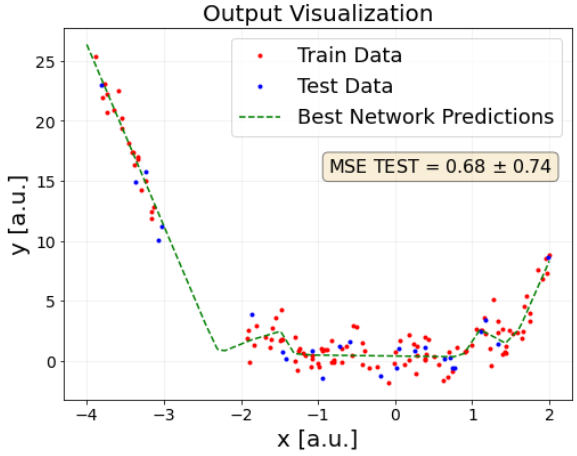

Figure 3: Dataset visualization and network output.