Lab 2 - Function Estimation

Vidaich Francesco

1 Introduction

The goal of this assignment is to build a neural network to solve a regression problem. In particular, It has to estimate the function generating the Train and Test datasets shown in Fig 1. These datasets are relatively small, containing respectively 120 and 25 samples.

For learning purposes, in this exercise it was forbidden the use of pre-built frameworks for developing and training the model. Nevertheless, it was provided a script called *Lab_02.py* containing a simple implementation of a neural network with two hidden layers and its training on a set of points generated with some random noise. This script (and in particular the `Network` class) was intended to be used as a starting point for this project.

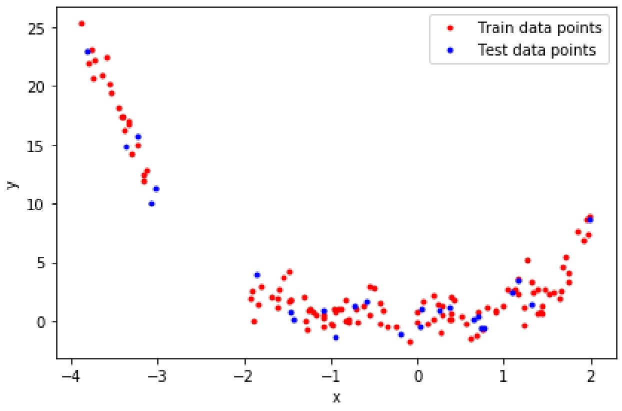

Figure 1: *Train and Test datasets*

2 `Network` class improvements

2.1 `Train` method

For simplicity purposes, the training procedure has been added to the class as a method and it can receive multiple optional parameters as arguments. The method is defined as follows:

```
def train(self, x_train, y_train, x_test, y_test, num_epochs,
          lr, en_decay=False, lr_final=0, return_log=False, out_log=False):
```

Let's talk about the different parameters starting from `x_train`, `y_train`, `x_test` and `y_train`, which are the standard arrays containing the splitted datasets. Then, `num_epochs` indicates the number of forward and backward passes of all the training examples. The `lr`, `en_decay` and `lr_final` variables dictate the learning rate evolution. To use a constant learning rate, the `en_decay` must be `False` and the `lr_final` can be omitted or set to any value (it will not be used). If `en_decay` is `True`, the learning rate will start at `lr` and will decrease linearly at each epoch, finishing at `lr_final` during the last one. The last two arguments are just two flags that regulate the output of the training. If `return_log = True` , the function will not return the average train and test loss values for the trained network, but it will return their evolution epoch after epoch. If `out_log = True` , the average train and test loss values are printed for each epoch.

2.2 Custom Activation Functions

In the original script, the only activation function that was used was the sigmoid, imported in the code as the `expit` function of the `scipy` library. For this exercise, the sigmoid was compared to the `ReLU` and `Leaky_ReLU` activation functions to check if the model could obtain different better results with them. To do so, these functions (and their derivatives) were built from scratch to work with both single values and numpy arrays.

The choice of the activation function was made by adding a correspondent argument to the `__init__` method of the `Network` class. This was wrote as follows:

```
def __init__(self, Ni, Nh1, Nh2, No, act, L1_lambda):

    ### ACTIVATION FUNCTION (given as argument)
    self.act = eval(act)
    self.act_der = eval(act + '_der')

    # ... weight initialization ...
```

The `act` variable passed to the method is a string containing the name of the wanted activation function. Using the `eval()` function it is possible to retrieve the activation function with the same name of the argument and save it to a class member (the same applies to its derivative, if it is called in the correct way).

A problem encountered with these new activation functions was the gradient explosion, caused by increasingly large weight updates. This was solved using a Gradient Clipping trick. This means that if the norm of the gradient was greater than a specific value (called `clip_norm` in the exercise), then it would be hard-clipped to have that value and each derivative that composed the gradient had to be rescaled accordingly like shown in the code below.

```
grad = np.concatenate([dWBo.flatten(),dWBh2.flatten(),dWBh1.flatten()])
grad_norm = np.linalg.norm(grad)
if grad_norm >= clip_norm:
    dWBo  = dWBo/grad_norm * clip_norm
    dWBh2 = dWBh2/grad_norm * clip_norm
    dWBh1 = dWBh1/grad_norm * clip_norm
```

2.3 L1 Regularization

In the last subsection it was possible to see that the `__init__` method takes also a `L1_lambda` argument. This is because the last improvement to the network class was the implementation of a L1 Regularization to avoid overfitting. To do this, it was necessary to add to the loss function a regularization term during training (when testing the model, it is omitted). In the case of L1, this new term is $\lambda \sum_i^{N_{tot}} |w_i|$, where N_{tot} is the total number of weights in the network. So the loss values during training were computed in this way:

```
loss  =  (Y - label)**2/2 + self.L1_lambda*( np.abs(self.WBh1).sum() +
                                             np.abs(self.WBh2).sum() +
                                             np.abs(self.WBo).sum() )
```

To implement the regularization correctly, and since a new term was added to the loss function, it is required to add also a correspondent term computing each component of the gradient during back-propagation. It is fairly easy to check that the components of the new gradient are made by the old ones (represented by `dWB` in the code) plus $\lambda \text{sgn}(w_i)$.

3 Parameters search and Training

3.1 Cross Validation

After checking that the new class was working properly, the Cross Validation procedure was implemented within the `Kfold_Cross_Validation` function.

```
def Kfold_Cross_Validation (x_train, y_train, Ni, Nh1, Nh2, No, act_func, L1_lambda,
                            num_epochs, lr, en_decay, lr_final, log=False, K=4):
```

The function takes as arguments all the parameters required to initialize and train a NN model; then, after completing the procedure, returns the average validation error. By default the training dataset is splitted into `K=4` subset and, for each iteration of the procedure, each one of them is used as a validation set for the network trained with the other ones. If `log = True`, the function will print the final train and test loss values for each validation set.

3.2 Grid Search

The main goal of the Grid Search method is to find the best set of parameters of the network and its training schedule. To do so, it was first necessary to build dictionary with five entries, each one containing an array of the values of the interested parameters that we want to test. Using the `itertools.product` function it is possible to build an array containing all the possible combinations of these values, represented with tuples that have this form:

```
(Ni, Nh1, Nh2, No, act_func, L1_lambda, num_epochs, lr, en_decay, lr_final)
```

Then this tuples are passed to the `Kfold_Cross_Validation` function to be evaluated; more in specific, the function associates to each combination of parameters the average validation error. After this procedure, all the scores and their corresponding sets of parameters are sorted to produce a ranking of all network configurations.

Since there are many parameters that are shuffled during the Grid Search, by just adding some extra values to be tested the number of possible combinations increases exponentially. Training in a standard way all the combinations defined in the exercise would have taken around a day, for this reason it was necessary to parallelize the code by splitting the work between all cores available in the machine (four in this case). This job was done by the `multiprocessing` library: its `Pool` function creates an object representing the four threads whose `starmap` function allows to execute for each tuple of parameters a wrapper of the `Kfold_Cross_Validation` function (simplified version below).

```
def Grid_Search_single_iteration(params, x_train, y_train):
    score = Kfold_Cross_Validation(x_train, y_train, *params)
    return score
threads_pool = Pool(4)
scores = threads_pool.starmap(Grid_Search_single_iteration,
                              zip(params_combinations),
                              it.repeat(x_train), it.repeat(y_train)))
```

In this way the Grid Search process took just over 10 hours, which is still a lot of time (but it is understandable given the size of the problem). After completing the process, the program stores in two different files the sorted scores and the indexes of the correspondent combinations of parameters, so this is not required to run again if the kernel is interrupted or the machine is restarted.

After completing the Grid Search, the programs takes the 20 best combinations and re-train the NN using the whole training dataset (without splitting it for cross validation) and test it using the test dataset. The network that obtains the lowest values is then considered the best one.

4 Results

In the beginning the values inserted into the parameters dictionary were not chosen with a systematic method, so the best configurations had almost the same structure. After tuning the possible values for `lr_policy` and `L1_lambda`, the Grid Search process was repeated and the best model was characterized by the following parameters:

```
(1, 75, 50, 1, 'leaky_ReLU', 0.001, 1000, 0.075, True, 0.001)
```

This winning network achieved a final test loss value of 0.5678, that corresponds to a MSE of 1.1356. The loss evolution of the network and the final approximation of the function are shown in the next plots.

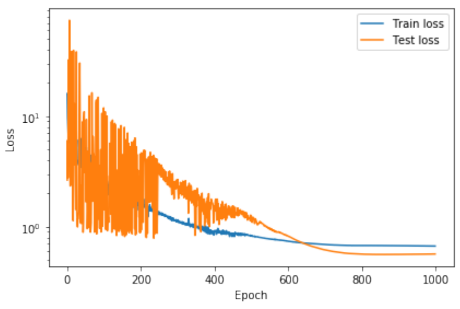

Figure 2: *Final train and test loss*

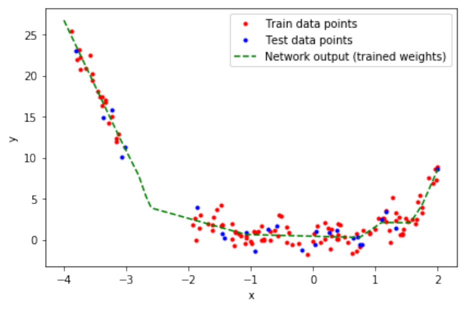

Figure 3: *Final estimated function*