Report of homework 1: Feed forward network with numpy

Clara Eminente

January 19, 2021

Abstract

In this report some aspects of the implementation and the training procedure of a feed forward neural network are described. After a brief introduction on the data and the network, in the first section we describe the introduction of cross validation in order to make the basic training procedure more robust. In order to avoid overfitting early stopping was also implemented: its functioning is described in the second section. In the third section some additional features are described, such as the implementation of different activation functions and regularizers. In the last section all the previous features are put together in order to random search the best set of parameters for the model.

1. Introduction

The aim of this assignment is to expand the code made available during the second laboratory session in order to successfully train a 2 hidden layer feed forward neural network over a dataset of points, as shown in igure 1.

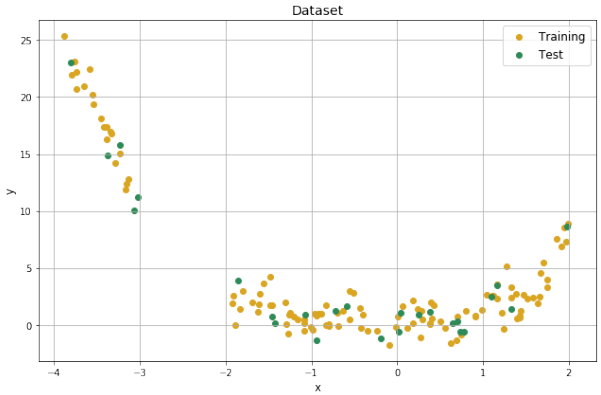

Figure 1: *Dataset*

The available code mainly consisted in a python class, *Network*, which provided an initialization function, a function to evaluate the output of the network given the weights and an input (*forward*) and a function to update the network weights through gradient descent (*update*).

All the functions (and the modified class *Network* itself) are available in a custom module, **NN_H1.py**. Further analysis and the pictures in this report were made using a Jupyter Notebook.

1.1. Initial adjustments: batches

The first modification made on the class *Network* was to introduce the possibility to **train the network using batches of data**. In order to do so the input to the network must have dimensions [batch_dim, (input_dim)] (e.g. for batches of 10 points, so that the feature and the label have both dimension 1 as in our case, the input shape would be (10,1)).
To implement training with batches it was considered more convenient to transpose the weight matrices so that they have dimension (Ni,No).

2. Cross-Validation and first training

The first task of the homework was to successfully train the network on the dataset. In order to make this procedure and the followings more stable, cross-validation was introduced in the code. Since some parts of the training are stochastic (starting with the initialization of the weights) the score obtained training a network just once can be unstable. Cross-validation consists in splitting the available dataset in k folds: $k-1$ folds are used as training set and the last one is used as validation set, a sort of test set (it is not fed to network as the training set) that keeps track of the improvements made during the training. Supposing we split the dataset in k folds it will be possible to train the network using the same set of hyperparameters k times, reinitializing the network each time, and evaluate the final score of the set of hyperparameters as the mean over the k validation losses.

Cross-validating a model makes the results of the training more robust, which is fundamental in order to compare different sets of hyperparameters, as we will see in the last section of this report. Moreover, having a validation set gives the possibility to implement *early stopping*, as we will see in the following

section.
Cross-validation was implemented introducing a function, $kfold$, that takes in input a dataset, k (the number of folds) and a boolean variable, $shuffle$. The output of the function is a list of k lists, each list containing the indeces of a validation set. If $shuffle = True$ the indeces are shuffled before being split.

Figure 2 shows the result of the training of the network using the parameters in Table 2. Table 1 shows the results at the end of the training.

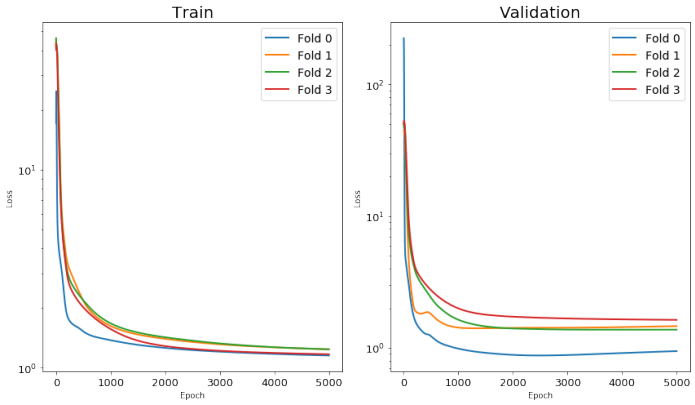

Figure 2: *Losses of the first training using cross-validation*

Fold	Final train loss	Final validation loss
0	1.151	0.946
1	1.237	1.463
2	1.238	1.376
3	1.168	1.633
mean	1.199	1.355

Table 1: *Final losses after first training with cross-validation*

The results obtained after this first phase of training are more then satisfactory (considering that the parameters where not tuned in any way). It is however possible to notice that the validation loss of fold 0, after an initial decrease, starts slowly increasing, resulting in the overfitting of the model. This phenomenon can be avoided introducing an *early stopping* procedure.

Ni	1	n_epochs	5000
Nh1	50	lr	0.01
Nh2	50	decay	True
No	1	lr_final	0.001
function	"sig"	n_folds(k)	4

Table 2: *Initial training and network parameters*

3. Early stopping

As mentioned in the previous section, a way to avoid overtraining a model is stopping the training when the validation error starts increasing. This procedure is called *early stopping*. The code below was added to the training in order to keep track of the validation error and stop the training while keeping a record of the best model (weights).

```
if (val_loss < best_val_loss):
    #keeping track of best parameters
    best_val_loss = val_loss
    best_train_loss = train_loss
    best_model = model
    count = 0 #count is reset everytime i
    get a better performance
elif val_loss >= best_val_loss:
    count += 1
if (count >= patience):
    break
```

Listing 1: *Early Stopping*

patience is a parameter introduced to decide how many epochs without improvement to wait before stopping the training.
Figure 3 and Table 3 shows the difference in score with and without *early stopping*.

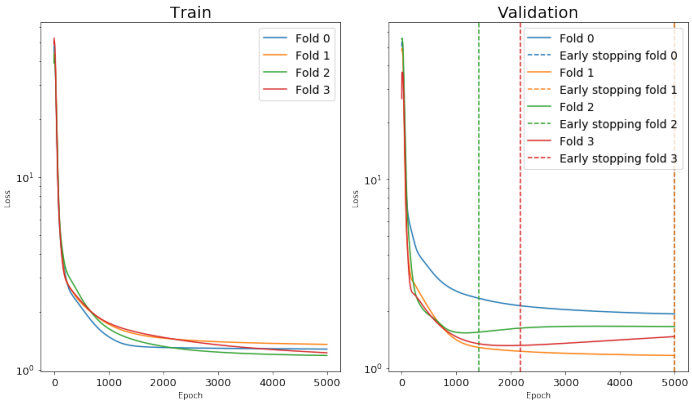

Figure 3: *Losses with early stopping. Note that the dashed line represents the epoch at which the training was stopped so the best model (with best validation loss) was obtained "patience" epochs before*

fold	early stopped?	Final val_loss	Val_loss at early stopping
0	No	1.939	1.939
1	No	1.169	1.169
2	Yes, at epoch 1414	1.662	1.534
3	Yes, at epoch 2179	1.472	1.318
mean	-	1.561	1.490

Table 3: *Scores with early stopping*

4. More features and final training function

At this point there are two things left: implementing some optional features (such as more activation functions and regularizers) and implementing a search over the parameters of the model in order to evaluate the best set.
In order to do so (and slim the code) the training procedure was included in a function, *train_model*, included in the module *NN_H1*.

```
def train_model(model, train_set, val_set,
    batchsize, n_epochs, lr, decay=True,
    lr_final=0.0001, reg_type=None,
    alpha_reg=0.0001, patience = 300,
    return_log=False)
```

Listing 2: *Train Model function*

As mentioned, some optional features were also implemented (and are included in the same module):

- More activation functions: ReLU and leaky ReLU (Sigmoid was the default one)
- Regulizers: L1, L2 and None

5. Random Search

At this point all the functions necessary for the search of the best set of hyperparameters are available.
There are basically two main sets of hyperparameters to choose: the hyperparameters of the network and the training parameters. In particular:

- Network_pars:
  - hidden neurons (Nh1, Nh2)
  - activation function in ["ReLU", "leaky_ReLU", "sig"]
- Training_pars:
  - number of folds (k)
  - batchsize
  - n_epochs
  - learning rate(lr)
  - learning rate decay(decay)
  - final learning rate(lr_final)
  - regularizer(reg_type) in ["L1", "L2", "None"]
  - regularizer parameter(alpha_reg)
  - patience

Given the huge amount of parameters it was considered more convenient to perform a random search over combinations. Moreover, some parameters were kept fixed, using some observations made while implementing the training procedure. The fixed parameters are shown in Table 4. Each set of hyperparameters (a set is made of both Network and Training parameters) was trained over the k folds and the validation loss of the set was computed as average over all k validation losses. Each set was saved along with its validation loss in a .csv file. After all the sets have been trained (and their validation losses computed) the one with smaller validation loss is picked and trained again in order to obtain the final trained network. Its performance is then checked over the test set. In the following figures and tables is possible to see the final model performances and parameters.

Ni	1	n_epochs	5000
No	1	batchsize	1
patience	400	decay	True
n_folds(k)	4	lr_final	0.0005

Table 4: *Fixed parameters during random search*

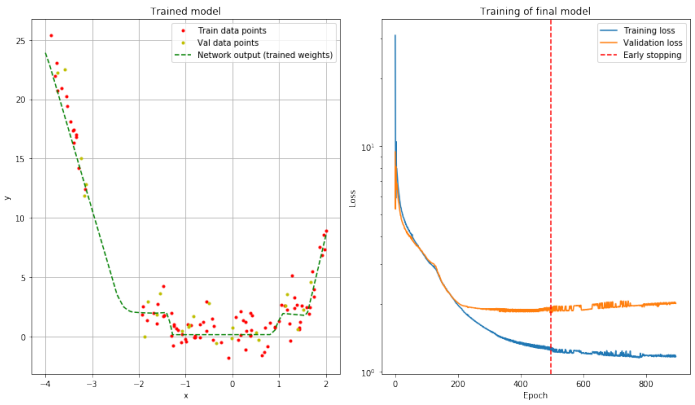

Figure 4: *Results of the final model and the final training*

Nh1=Nh2	83
act_f	ReLU
lr	0.0046
reg_type	L1
alpha_reg	0.005

Table 5: *Final parameters of the model*

train_loss	val_loss	test_loss
1.227	1.826	1.329

Table 6: *Final losses: train and validation loss were computed splitting the training set in 80-20 and training the model once*

Some notes on the final model. The 10 best models have some features in common: none of them has a sigmoidal activation function and the initial learning rate lays in the range [0.002-0.008] with most of the values being around 0.004. On the other hand all the types of regularizers appear and different values of the regularizer parameter *alpha_reg* are present.