Neuradillo

A neural network simulator

Giovanni Sorice
Francesco Corti

f.corti3@studenti.unipi.it
g.sorice@studenti.unipi.it

Machine Learning (654AA), A.Y. 2019-2020

Date: *28/01/2020*

Type of project: **A**

Abstract

The aim of the following document is to illustrate the work done for the Machine Learning course project. We developed a library to build neural networks in C++ exploiting Armadillo numerical library. We built models with one or more hidden layer, we implemented cross-validation using the k-fold technique and grid search for hyperparameter tuning. The implementation was designed to be modular and extensible.

1 Introduction

Our goal was to create a Neural Network model simulator and apply it to MONK's classification problems and the ML CUP regression problem. We developed a library with the aim of building, training and testing feedforward neural networks using back-propagation and multiple versions of gradient descent. Namely stochastic, mini-batch and batch with Nesterov Momentum and weight decay as regularization and using different activation functions and loss functions. Our assumption was to find the best model for each problem. This was reached by looking for a smooth and compact loss function. We achieved this by using grid search technique with k-fold cross validation. We paid attention not to commit overfitting using regularization technique such as weight decay.

2 Method

2.1 Code

The Neural Network simulator was implemented in C++17 using Armadillo library for linear algebra operations. Moreover, we used Conan package manager, for automating the installation and distribution of the library. We focused on modularity and extensibility. For this reason, we put some effort into made parametric class.
The library is structured as follows: the class `Network` has a list of layers, added by the user and a loss function. It exposes all the methods for training and testing the network. Every `Layer` object, that is composed of an armadillo matrix for the weights and a bias array, has a specific activation function. We can build multilayer feedforward neural networks with a different activation function. The training was developed following the back-propagation algorithm with a forward and back operation using gradient descent for weights adjustment. The user can either specify the Nesterov momentum and the weight decay parameters or leave the default behaviour as it is. Also the stop condition, the difference between the current validation loss and the previous validation loss, can be passed to the method train. The training was implemented for support stochastic, mini-batch and batch gradient descent.
We defined a `Cross-validation` class that performs k-fold cross validation and computes the average error on validation. We used this class to choose the best model in the ML CUP.
In addition, we developed a `Grid Search` class for which the user can specify minimum, maximum and step values for all parameters. The parameters are the following: epoch, number of units, learning rate, weight decay and Nesterov momentum coefficient. We also decided to implement a parallel version of the `Grid Search` to speed up the computation.
The class `Activation Function` had been designed to abstract all possible activation functions. The available functions are `sigmoid`, `tanh`, `reLU` and `linear`. With the aim of extensibility the `Cost Function` class had been designed to abstract all the cost functions. The available functions are `mean squared error`, `mean euclidean error` and `binary cross entropy`.
To reduce the cost of moving matrices we exploited the *move* operator available in C++17.

2.1.1 Preprocessing

Preprocessing is divided between two classes, `Preprocessing` and `LoadDataset`. The former reads, shuffles and splits the dataset whereas the latter performs the one-to-k decoding. All the operations done in the classes manage the memory usage. Particular attention was given to minimize data copying. The MONK dataset was decoded using the `LoadDataset` class.

2.1.2 Validation

For MONK's problems, the training dataset was split into a new 80% training set and a 20% validation set. The error on the validation set was used as a proxy for the generalization error to determine when overfitting had begun. The test set was given to us already separated from the training set, we used it to evaluate the prediction error on new unseen data.
For the ML CUP problems the dataset was split into a new 60% training set, 20% validation set and a 20% test set. We decided to use k-fold cross validation, with k=3, to find the best model among the grid search results. We shuffled the dataset before the splitting step and the training set before each epoch of the training phase. The latter had been done to avoid the stopover at a local minimum.

2.1.3 Preliminary trials

For MONK's problems, we initially trained the network without the one-hot-encoding but we didn't reach a good accuracy. After the one-hot-encoding was applied, we reached a high accuracy in all the three problems. For the third set, we had to tune the regularization parameters to reach a good result.
For ML CUP the entire learning process was harder. Firstly we conducted some experiments to figure out the size of the grid search. Next, we studied the results and we started a smaller range grid search on the previous best results. Once we obtained the best models we retrained them on the training set. We checked that the output predictions about the test set were as expected. Initially, the model selection and assessment had been done on models with one hidden layer. Next, we decided to try with different models having three or four hidden layers. The results will be shown later.

3 Experiments

3.1 MONK's results

The neural networks used for the problems have 17 input units, the number of hidden units are written in table 1 and the output layer has always one unit. We used `tanh` as a hidden layer activation function and `sigmoid` as an output layer activation function. We chose `Mean squared error` as the loss function. An input is classified as 1 if the output of the neural network is greater than 0.5 or 0 otherwise.
We used stochastic, mini-batch and batch gradient descent but in the end we decided to use the batch gradient descent because loss function plots were smoother than the others. We also used the weight decay regularization as the lambda parameter and Nesterov momentum as the momentum parameter. Table 1 reports the average of the values found for TR and TS after five different trainings of the networks. For the training phase we initialized the weight with a uniform distribution in the interval [-1e-3, 1e-3].

Task	**#Units**	**eta**	**lambda**	**momentum**	**MSE(TR/TS)**	**Accuracy(TR/TS)**
MONK 1	3	0.9	0	0.7	6.3e-4/1e-3	100%/100%
MONK 2	4	0.8	0	0.7	1e-3/1.3e-3	100%/100%
MONK 3	5	0.4	5e-3	0.2	7.8e-2/5.5e-2	93.44%/97.22%
MONK 3 (no reg)	5	0.6	0	0.7	1.7e-2/2.7e-2	95.90%/93.51%

Table 1: MONK's problems parameter and results.

3.1.1 MONK 1

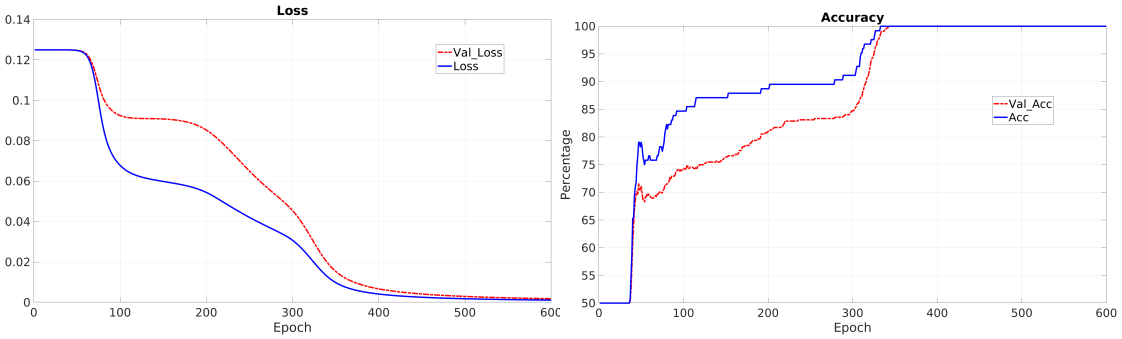

Figure 1: MSE and accuracy for MONK's 1.

3.1.2 MONK 2

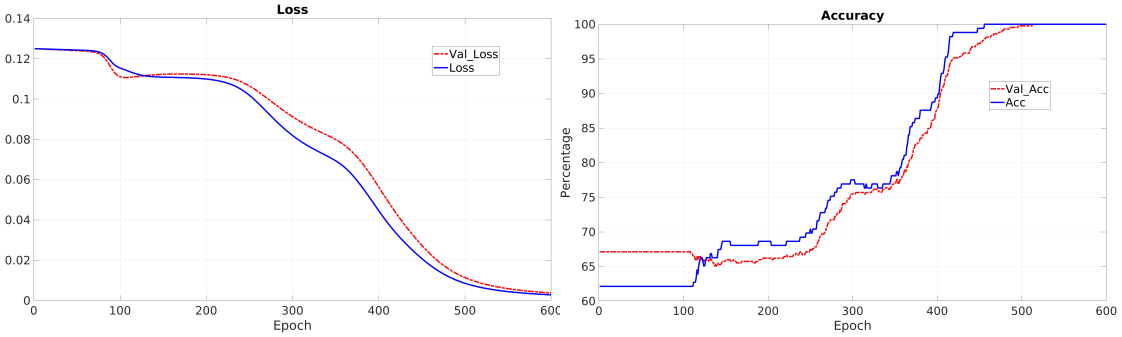

Figure 2: MSE and accuracy for MONK's 2.

3.1.3 MONK 3

We figured out during the training phase that the learning curves of MONK's 3 without regularization showed overfitting in the output (see fig. 3). In response to that, we fixed it by adding regularization. We obtained better results since the training didn't show overfitting in the validation error (see fig. 4).

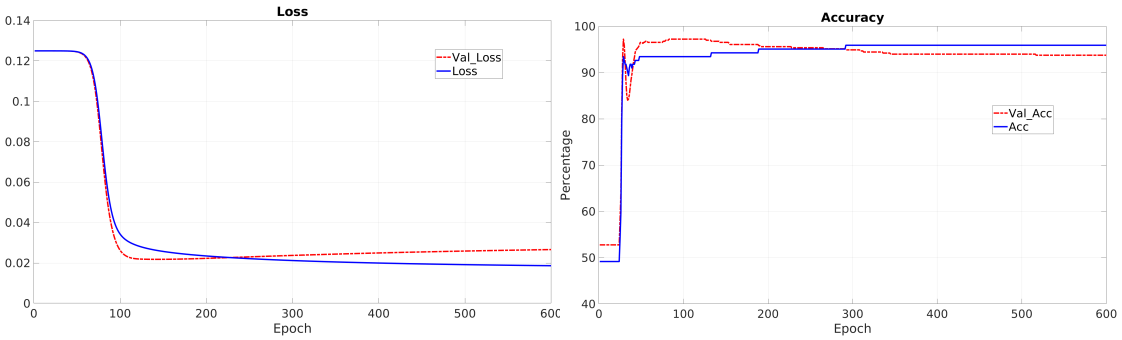

Figure 3: MSE and accuracy for MONK's 3 not regularized.

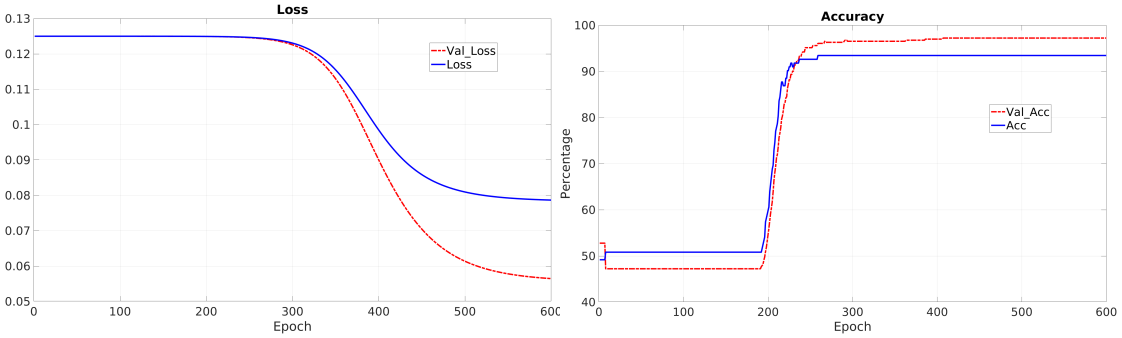

Figure 4: MSE and accuracy for MONK's 3 regularized.

3.2 Cup results

3.2.1 Validation schema

Initially, we split the dataset in training set and test set with 80% (1412 records) for training and 20% (353 records) for the test. For the selection of the best model, we decided to use k-fold cross validation with k=3. In the end, we chose the best models among all the models trained during the grid search step. We used stochastic, mini-batch and batch gradient descent and we observed that the batch type had smoother learning curves.

3.2.2 Screening phase

Firstly we tried with single hidden layer networks and we could already observe good results. For this reason, we tuned the hyper-parameters for this type of network through grid searches. Then we chose MEE as the loss function for validation and test set. We trained the network with MSE but we plotted the results with MEE to make them comparable with the learning curves of test and validation set. Later, we focused on finding some variants of the network with more hidden layers. We tuned the hyper-parameters through grid search. The screening phase turned out to be fundamental since choosing the right search setting of the hyper-parameters saved us a lot of time. We understood which hyper-parameters needed to be discarded a priori (e.g. high learning rate fig. 5 or small mini-batch fig. 6).

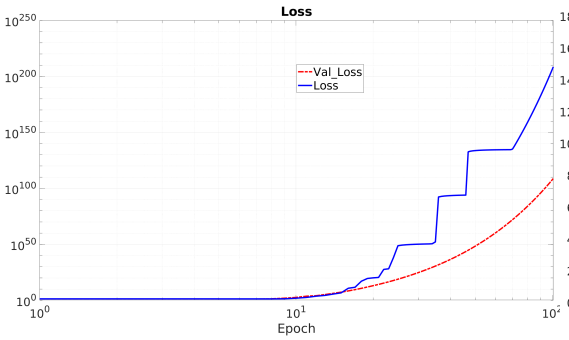

Figure 5: MEE neural network divergent.

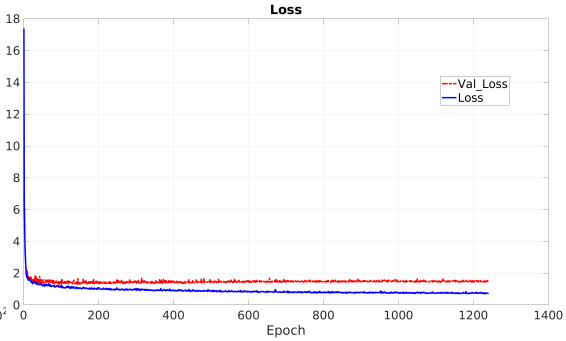

Figure 6: MEE small mini-batch (32).

3.2.3 Explored hyper-parameters

At the beginning, we ran grid search with large intervals and high steps. When we found the right interval we ran a grid search for each combination of the values:

- Unit $\in$ {25, 200} with step $\in$ {25, 50};
- Learning rate $\in$ {0.0001, 0.01} with step $\in$ {0.0001, 0.001};
- Lambda $\in$ {0, 0.004} with step $\in$ {0.0001, 0.0005};
- Momentum $\in$ {0, 0.8} with step $\in$ {0.05, 0.1}.

We used the `tanh` activation function for all hidden layers and `linear` activation function in the output layer. We picked `tanh` because it gave the best result within all the other grid searches. `Linear` was chosen because it is often used as an output layer activation function for the regression problem. For the training phase, we initialized the weight with a uniform distribution in the [-0.7, 0.7] interval.

3.2.4 Grid search result

Table 2 reports the average of the values found for TR, VS and TS during the k-fold cross validation with k=3.

Layer	Units	Learning rate	Lambda	Momentum	Error TR	Error VS	Error TS
1	75	0.00450	0.00001	0.6	0.9676	1.1202	1.2151
1	100	0.00087	0.0000	0.8	0.9832	1.2750	1.2662
1	100	0.00097	0.0000	0.8	0.9851	1.2759	1.2786
1	100	0.00500	0.0000	0.5	0.9859	1.2845	1.2784
1	100	0.00500	0.0000	0.7	0.9752	1.2654	1.2569
1	100	0.00500	0.00001	0.6	0.8731	1.2412	1.2375
2	150	0.00875	0.0002	0.8	0.8467	1.1332	1.1372
5	375	0.00450	0.0001	0.7	0.6323	1.1341	1.1341

Table 2: Best network configurations with MEE.

The two models with more than one hidden layer have the following structure:

- Two hidden layers: 100-50-2 (see fig. 7);
- Five hidden layers: 100-100-75-50-50-2 (see fig. 8).

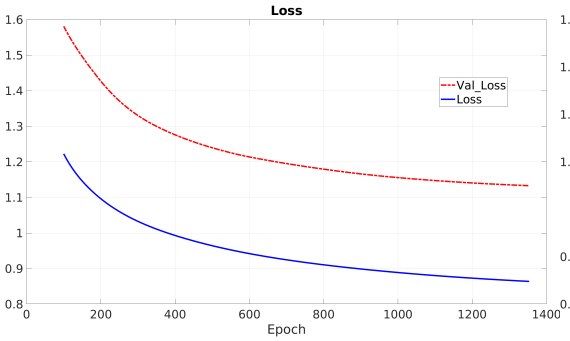

Figure 7: MEE two hidden layer.

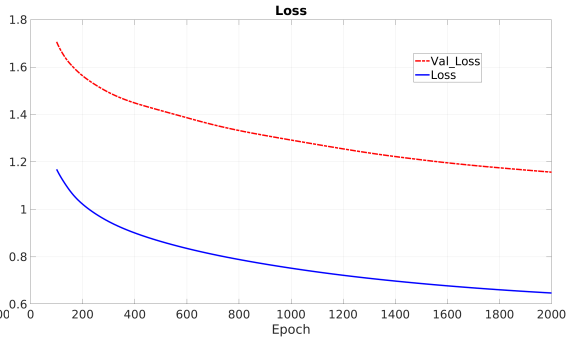

Figure 8: MEE five hidden layer.

3.2.5 Computing time

We have developed a parallel grid search that allows us to take advantage of all our CPU cores automatically. All the computations were launched on the following machines:

- Intel i7-8500Y, 1.5GHz;
- Intel i7-4720HQ, 2.6GHz.

The project was developed in C++ and the computational time and memory had to be strictly managed. The time it takes to train on the MONK dataset with 800 epochs and 5 units is 1.5 seconds, for the ML CUP dataset with 8000 epochs and 75 units is 1.10 minutes.

3.2.6 Comparisons

We tried multiple neural networks with different number of layers and distinct types of gradient descent (stochastic, mini-batch and batch). We came to the conclusion that the number of layers does not significantly affect network generalization performance. Moreover with batch gradient descent we obtained the best learning curve.

3.2.7 Chosen model

The final model was chosen from the best models found after the grid search (table 2). It has the following hyper-parameters (table 3) and MEE error for the training set, validation set and test set. We achieved an average MEE of 1.2093 on our test set, taken as an average of seven different trainings of the network (to avoid the bias due to the random weight initialization).

Layer	Units	Learning rate	Lambda	Momentum	Error TR	Error VS	Error TS
1	75	0.00450	0.00001	0.6	0.9676	1.1202	1.2093

Table 3: Best network configuration with MEE.

We chose it because it was the model that performed better in the validation set. Also, its learning curve was smooth and stable. The learning curve, also with an enlargement, is shown in figure 9.

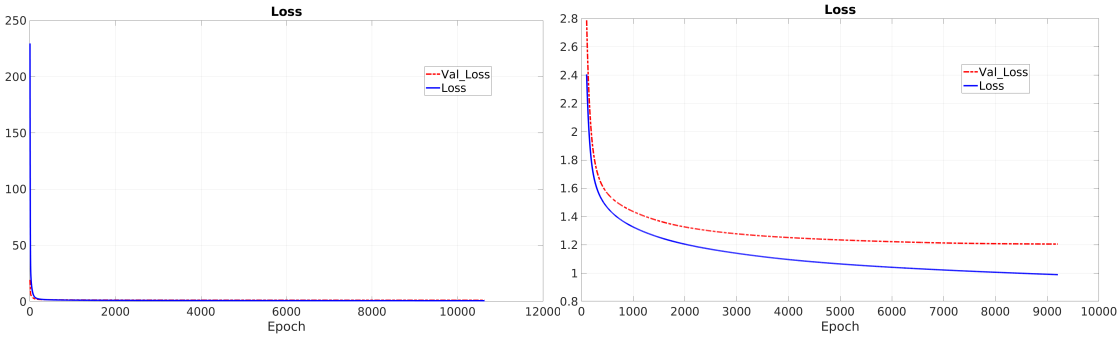

Figure 9: MEE and zoomed MEE for ML cup regularized.

4 Conclusions

The project gave us the understanding of how the theory seen during the course is strictly correlated to the implementation of a real neural network library. Thanks to MONK's dataset we discovered how really the regularization techniques prevent overfitting, controlling model complexity. We understood the importance of choosing a good model and how to do it using grid search and k-cross validation. The hyper-parameter tuning was also the most difficult phase with the debugging and testing of the backpropagation algorithm. This experience thought us the foundations of learning algorithms on neural networks on which many machine learning frameworks are currently based. Overall this was a great experience where us, as a team, helped each other when the struggles came in.

4.1 Agreement

We agree to the disclosure and publication of our names, and of the results with the preliminary and final ranking.

References

[1] S.B. Thrun, J. Bala, E. Boloederon and I. Bratko. *The MONK's Problems.* [*A performance comparison of different learning algoritms*]. Chapter 9. Carnegie Mellon University CMU-CS-91-197, 1991.
`https://pdfs.semanticscholar.org/94c0/418c4bd9d719e1203acfd42741ebbd343073.pdf`

[2] Conrad Sanderson and Ryan Curtin. *Armadillo: a template-based C++ library for linear algebra.*
`http://arma.sourceforge.net/armadillo_joss_2016.pdf`

[3] Simon Haykin. *Neural Networks and Learning Machines.* Prentice-Hall, (3rd Edition, 2008).

[4] T. M. Mitchell. *Machine learning.* McGraw-Hill, 1997.

[5] I. Goodfellow, Y. Bengio, A. Courville. *Deep Learning*. MIT Press, 2016.

[6] Michael A. Nielsen. *Neural Networks and Deep Learning*. Determination Press, 2015. `http://neuralnetworksanddeeplearning.com/chap2.html`