

MLisML - Machine Learning is Matrix muLtiplication

Dario Salvati

Andrea Zuppolini

d.salvati2@studenti.unipi.it

a.zuppolini@studenti.unipi.it

Master Degree in Computer Science.

ML 654AA, Academic Year: 2020/21

Date: 04/01/2021

Type of project: **A**

Abstract

MLisML is a project developed for the Machine Learning course at the University of Pisa. The aim is to perform an experiment on a given dataset and to do so we've used a deep neural network made of two hidden layers of 50 and 30 units respectively. Since the given dataset for the experiment was quite small, we decided that the best validation technique was using 10-Folds cross validation for the model selection and Hold Out for the model assessment.

1 Introduction

This project has been realized to implement a simple but effective machine learning simulator which can be used to perform simple experiments, while using some of the most common machine learning techniques that exist nowadays. Even if the simulator we implemented is not as complex as the most famous ones like Tensorflow or Pytorch, we still managed to realize a set of practical tools that can be used on a small scale.

To prove that, we realized an experiment from a given dataset of unstructured data points, which can be used to train a model for a regression problem. After doing a grid search and evaluating different models, we decided that the best trade-off between generalization capabilities and the accuracy of predictions was given by a deep neural network made of two fully connected hidden layers of 50 and 30 hidden units respectively. More details about the model are provided later in the report. The chosen algorithm for model training is the Stochastic Gradient Descent (SGD from now on), enriched by the implementation of Tikhonov's regularization and Nesterov's momentum.

We don't have any additional information on the data points of the dataset, hence we didn't make any particular assumption, even if the input and output values seem bounded by a rule, which is unknown.

2 Method

2.1 The code

The language of choice for this project was `Python` and no external machine learning libraries were used. The only non built-in module that has been used is `Numpy`, which helped us speeding up the matrix calculation and provided us all the mathematical tools needed for the realization of the simulator. The code has been written following an hybrid approach using the imperative and the OOP paradigm, starting from the implementation of the `Layer` class, which composes the `NeuralNetwork` class. We also implemented the `Dataset` class, the parent class of the `MONKS` and `MLCup` classes, which are used to store and manage the data obtained from the various datasets. We also implemented some function modules, which are used to manage different activation, initialization and loss functions.

2.2 The architectures

For the experiment we tested different model architectures, using different topologies, activation functions and weights initialization functions.

- A single, 20 units hidden layer model, using the sigmoid activation function and the normal weights initialization function;
- A single, 50 units hidden layer model, using the sigmoid activation function and the normal weights initialization function;
- A single, 50 units hidden layer model, using the ReLU activation function and the uniform weight initialization function;
- A double hidden layer model, made of a 50 and 20 units layer respectively, using the sigmoid activation function and the He weights initialization function, which is the chosen model for the experiment.

We only used the SGD as learning algorithm, enriched by Nesterov's momentum and Tikhonov's regularization. We focused on the batch/mini-batch with large batch size approach, since it heavily speeds up the training time. This is guaranteed from the optimization on the matrix calculation given by `Numpy` which is able to make very fast computations on large matrices.

As the validation schema, we used a 10-Folds cross validation for the model selection and hold out for the model assessment. We partitioned the dataset into a **training set** composed of 75% of the data and an **internal test set** composed of the remaining 25%. More details are provided in section 3.

We didn't make any relevant assumption on the problem, hence we didn't feel confident enough to use an early stopping criteria.

3 Experiments

3.1 Monk Results

In this section are reported the results on the MONKS datasets.

Following the original paper, we encoded the patterns using the one-hot encoding technique on each component of the input pattern. As for the output, we used a single neuron, activated by the sigmoid function. If the output value is ≥ 0.5 , the pattern is classified as class 1.

We didn't make any validation for the used models since this experiment is only for checking the correctness of the implementation of the learning algorithm.

Task	#Units, epochs, eta, lambda	MSE (TR/TS)	Accuracy (TR/TS)(%) ⁱ
MONK1	10, 600, 0.065, 0	0.0014 / 0.0074	100 / 100
MONK2	10, 600, 0.05, 0	0.0016 / 0.0016	100 / 100
MONK3	10, 600, 0.075, 0	0.0019 / 0.049	98 / 94
MONK3+reg.	10, 600, 0.075, 0.0001	0.007 / 0.044	100 / 94

Table 1: Average prediction results obtained for the MONK's tasks.

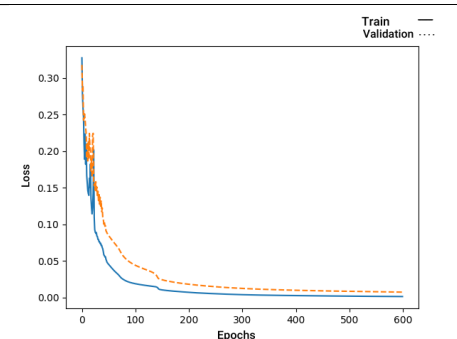
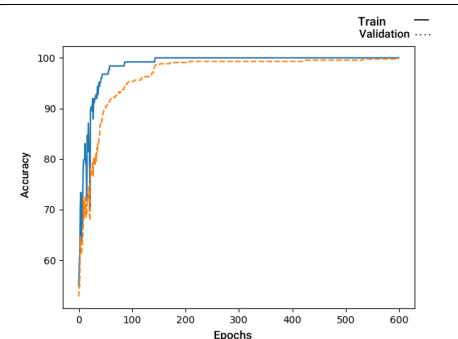
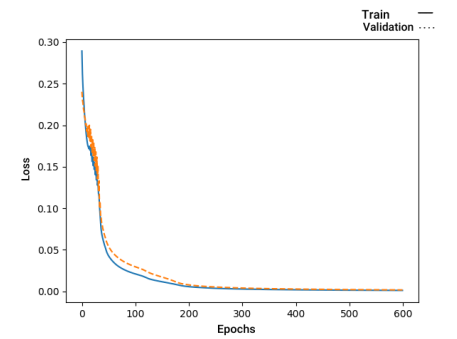
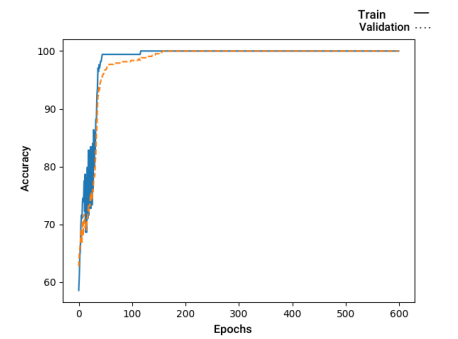
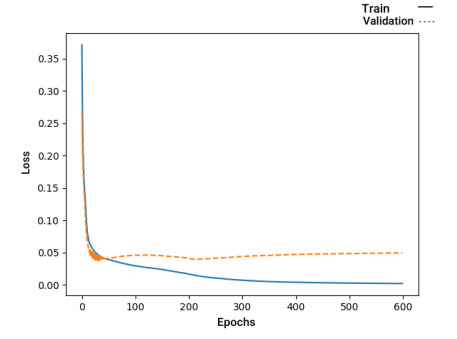
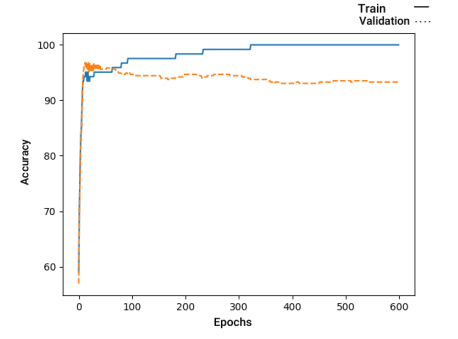
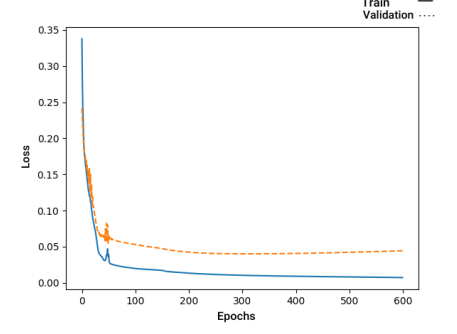
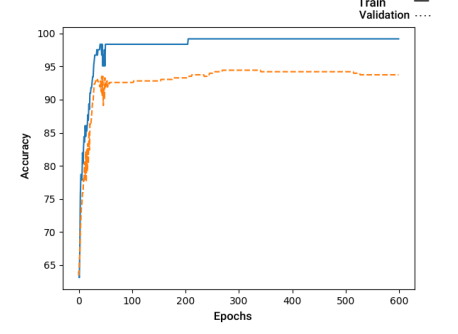
#	MSE	Accuracy %
MONK1		
MONK2		
MONK3 (no reg)		
MONK3 (reg)		

Table 2: Plot of the MSE and accuracy for the 3 MONK's benchmarks.

3.2 Cup Results

3.2.1 Validation Schema

As validation schema we applied a 10-Folds cross validation for the model selection, in conjunction with a hold out test for the model assessment. We partitioned the dataset as follows:

- training dataset, composed of 75% of the original dataset, used for the model selection phase and the retraining of the chosen model;
- internal test dataset, composed of the remain 25% of the original dataset, used for the internal test as the original dataset didn't contain any specific test set.

The number of folds we've chosen for the K-Folds cross validation schema is 10, because we've considered it the best possible trade-off between time complexity and the waste of data in training.

Using this parameters, each configuration was trained on 1029 patterns and validated on the remaining 114, using the Mean Euclidean Error. This error values has then been used to select the most performant model, which is going to be used to generate the results on the ML blind test.

3.2.2 Screening phase

Before proceeding with the grid search we made some empirical tests to find some ranges of hyperparameters which were effective for the particular model we were testing. We observed that some values made the loss explode, others made the decreasing of the loss value too slow and others were plausible and may be used for the final model.

3.2.3 Hyperparameters

In this subsection are specified all the values of various hyperparameters used in the grid search during the model selection phase. For obvious reasons we didn't use the same set of hyperparameters for the models that used the sigmoid and the ReLU activation functions, hence we present two different tables. **Please note that in the backpropagation algorithm the learning rate η is divided by the size of the batch.**

number of epochs	300	600							
λ	0.00008	0.00005	0.00003	0.00001					
α	0.8	0.5	0.3	0.1	0.08	0.05	0.03	0.01	
η	0.08	0.05	0.03	0.01	0.008	0.005	0.003	0.001	

Table 3: Grid used for sigmoid models

number of epochs	300	600							
λ	0.00008	0.00005	0.00003	0.00001					
α	0.008	0.005	0.003	0.001	0.0008	0.0005	0.0003	0.0001	
η	0.0008	0.0005	0.0003	0.0001	0.00008	0.00005	0.00003	0.00001	

Table 4: Grid used for ReLU models

3.2.4 Grid Search Results

In this section are reported the grid search results for each of the tested models, for a total of 2048 configurations. Below are reported the tables of the top 5 configurations for each topology.

#	Epochs	η	λ	α	Loss (MEE)	Validation (MEE)	Internal TS Accuracy
381	600	0.01	0.00001	0.05	2.7565	2.2632	2.2273
365	600	0.01	0.00005	0.05	2.7462	2.2695	2.3826
380	600	0.01	0.00001	0.08	5.6188	2.2750	2.3908
372	600	0.01	0.00003	0.08	2.3164	2.2833	2.4324
373	600	0.01	0.00003	0.05	2.1833	2.2845	2.4657

Table 5: Sigmoid model with a single hidden layer of 20 units

#	Epochs	η	λ	α	Loss (MEE)	Validation (MEE)	Internal TS Accuracy
347	600	0.03	0.00005	0.1	2.1171	2.0952	2.2324
339	600	0.03	0.00003	0.1	1.9868	2.0964	2.3469
340	600	0.03	0.00003	0.08	2.0365	2.0983	2.3511
349	600	0.03	0.00001	0.05	2.2110	2.1092	2.4369
331	600	0.03	0.00005	0.1	2.2823	2.1153	2.4629

Table 6: Sigmoid model with a single hidden layer of 50 units

#	Epochs	η	λ	α	Loss (MEE)	Validation (MEE)	Internal TS Accuracy
277	600	0.00008	0.00003	0.0005	5.2745	2.6176	3.8887
281	600	0.00008	0.00001	0.005	4.9291	2.7721	3.9025
286	600	0.00008	0.00001	0.0003	5.4407	2.7998	4.4397
287	600	0.00008	0.00001	0.0001	6.0228	2.8075	4.7169
269	600	0.00008	0.00005	0.0005	6.4091	2.8095	4.056

Table 7: ReLU model with a single hidden layer of 50 units

#	Epochs	η	λ	α	Loss (MEE)	Validation (MEE)	Internal TS Accuracy
337	600	0.03	0.00003	0.5	1.0074	1.9528	1.9860
306	600	0.05	0.00003	0.3	1.9188	1.9610	2.1120
322	600	0.03	0.00008	0.3	3.0225	1.9700	2.1080
298	600	0.05	0.00005	0.3	2.4877	1.9731	2.1200
314	600	0.05	0.00001	0.3	2.2373	1.9756	2.1619

Table 8: Sigmoid model with two hidden layer of 50 and 30 units

As we expected the model with two hidden layers performs the best from the pool of configurations we’ve observed, thanks to its increased complexity. We could have observed even more complex models, however we didn’t for two reasons: the first is that we didn’t want to obtain a model that overfits the data and loses generalization capabilities, which can happen with very complex models on easy problems. For this reason we also implemented Tikhonov’s regularization, which is useful at controlling the model complexity. The second reason is that the time complexity increases as the complexity of the model, hence the time required to perform a grid search was getting too large to manage.

The grid search results remark that similar configurations get similar performances, as it should be. This is also given from the fact that the 10-Folds Cross Validation averages the results obtained from 10 different initializations of the model, hence we get a result which is not so dependent on the weights initialization.

Note that we used the batch approach because it’s faster to compute respect to the online thanks to Numpy’s optimization in matrix operations.

Using this approach, we managed to compute the grid searches in an affordable time. Using a 3.6GHz processor, we managed to obtain the results in the following amount of time:

- model with a single hidden layer of 20 units using the sigmoid activation function: **1h20m**
- model with a single hidden layer of 50 units using the sigmoid activation function: **5h06m**
- model with a single hidden layer of 50 units using the ReLU activation function: **3h05m**
- model with two hidden layers of 50 and 30 units using the sigmoid activation function: **8h03m**

The algorithm can be heavily sped up by parallelizing the cross validation, since each iteration of the K-Folds is independent from the others.

3.2.5 Final Model

Given the grid search results showed above, we decided to select the configuration which minimizes the validation error, that represents the estimated risk. This is done to try to get the best possible generalization capability from the hypothesis space we observed, hence maximizing the accuracy on the internal test set.

In this case the configuration is:

#	Epochs	η	λ	α	Loss (MEE)	Validation (MEE)	Internal TS Accuracy
337	600	0.03	0.00003	0.5	1.0074	1.9528	1.9860

Table 9: Hyperparameters of the final model

which also obtains the smallest error on the train set and gets the highest accuracy on the internal test set.

Some observations on the final model we got.

Firstly, we noticed that for this problem, and the search space we explored, the sigmoid activation function performed better than the ReLU, which nowadays is typically preferred because of the vanishing gradient effect[1]. Obviously in our case we are working with very low complexity networks and for this reason we didn't face this problem.

After the retraining on the entire training set, the model has been used to generate the results for the ML Cup blind test.

In the following page are reported the graphs of the Loss (MSE) on the training set and the Error (MEE) on the validation and internal test set.

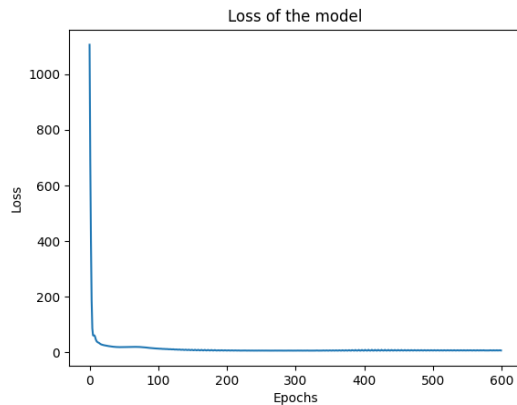


Figure 1: Loss of the final model

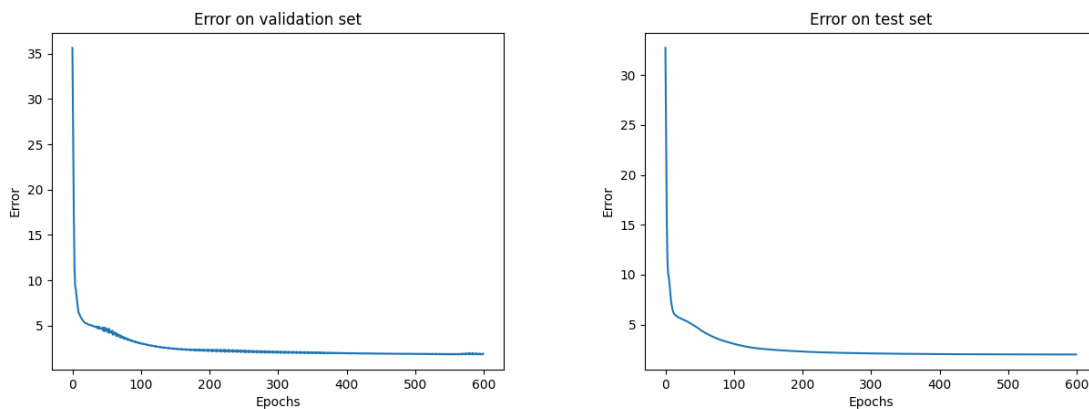


Figure 2: Error on validation and internal test set of the final model

Note that even if the loss of the model tends to be small, the good results obtained from the validation and internal test set suggest that the model has good generalization capabilities since it's approximating fairly well the unseen data points.

As we can see the curves are very smooth thanks to Tikhonov's regularization and the Nesterov's momentum helps to prevent "spikes" on the learning curve.

The learning curves without regularization or momentum can be found in the appendix.

4 Conclusion

This experience has been particularly formative and has given us the opportunity to apply plenty of concepts we've seen during the Machine Learning course. It allowed us to deepen our understanding of this field and Implementing the various fundamental algorithms made them crystal clear to us.

Name of the team: DeepMai

Name of the blind test results file: ML-CUP20-Results.csv

Acknowledgments

We agree to the disclosure and publication of my name, and of the results with preliminary and final ranking.

References

- [1] Sepp Hochreiter. The vanishing gradient problem during learning recurrent neural nets and problem solutions. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, 6(02):107–116, 1998.

Appendix A

In this plot is possible to see how the regularization and momentum coefficients are able to stabilize and smoothen the learning curve.

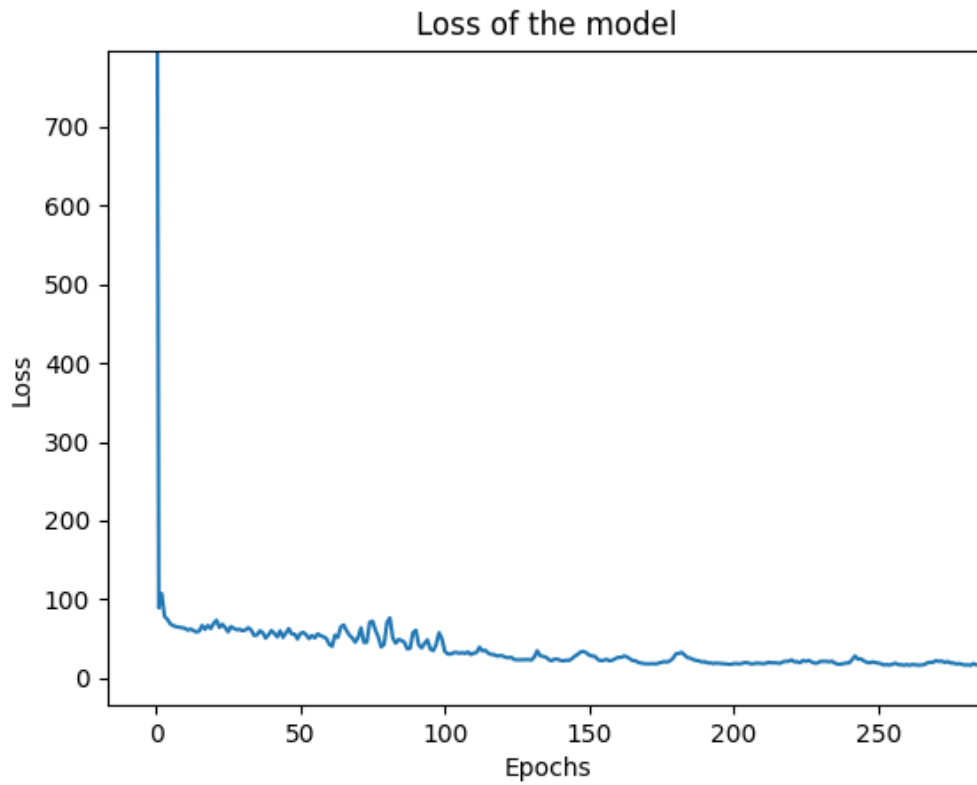


Figure 3: Learning curve of the final model without Tikhonov regularization and Nesterov's momentum