Machine Learning Project Report

Authors (*Raffaele Villani, Luca de Martino*)
Master Degree Artificial Intelligence.
Emails(r.villani4@studenti.unipi.it, l.demartino1@studenti.unipi.it)
ML course (654AA), Academic Year: 2020/2021
Date: 24/01/2021
Type of project: **B**

Abstract

The aim of this report is to compare multiple ML frameworks to solve the classification task on the MONK Dataset and the regression task on the CUP Dataset provided during the course. For the first aim, we compare different Neural Network models using Keras [2] and Pytorch [8], while for the second one we decided to compare also SVM model using scikit-learn [9].

1 Introduction

Our goal was to explore different types of frameworks and models, to identify their strengths and weaknesses. In order to do that we developed different Multi Layer Perceptron networks (MLP) and a Support Vector machine to achieve the state of the art results on the MONK's task. We passed after the first approach with this frameworks to the CUP's task, which was formed by 10 features and 2 labels where each feature and label is a real number, therefore instead of having a classification problem like in the MONK here we had to deal with a regression problem.
In order to test and validate the frameworks we used the MONK and we tuned the hyperparameters by ourselves taking a look to the plot and changing them accordingly. For the CUP's task, we applied the Hold-out and we used even K-fold Cross-Validation methods to validate the models. We will explain the relevant details in the following sections.

2 Method

To support the usage of Keras, Pytorch, and scikit-learn, we used:

- *Numpy* [7] for the numerical matrices operations
- *Pandas* [6] to handle the various datasets and to store the information obtained from the grid searches
- *MatPlotLib* [5] for the experimental and final plots
- *Skorch* [10] to allow Pytorch and scikit-learn to work together

We tried different kind of architectures, we started with a simple architecture composed of one hidden layer with a low number of neurons, then we added neurons and layers. As activation functions, we tried the `sigmoid`, the `tanh`, and the `reLu` to decide which one works better for our purpose. We chose to use only the Gradient Descent Algorithm as Optimizer with different batches, indeed we experimented it with the on-line, full batch, and mini-batch version. In order to implement the Mean Euclidean Distance function, we used the *custom_function* of Keras and Pytorch, for the 2 Network models, and for the validation schema, we used the *make_scorer* function of scikit-learn to use the MEE as score instead of the default ones.

2.1 Preprocessing

For the MONK's dataset, we applied the `One-hot encoding` technique, obtaining 17 binary features. In the CUP's dataset we tried to study the distribution of the features and the correlation between them (for more detail see Appendix B), but we decided to not apply any kind of preprocessing.

2.2 Weight Initialization

The Weight Initialization phase can lead to a better and faster convergence. For our purpose we have used the Xavier Initialization [3] (also called Glorot) where the Biases are initialized to 0 and the weights as:

$$W \sim U\left[-\frac{\sqrt{1}}{\sqrt{n}}, \frac{\sqrt{1}}{\sqrt{n}}\right]$$

Where U is a Uniform distribution and n is the size of the previous layer.

2.3 Validation Schema

For the MONK's task, we decided to not divide the dataset into training, validation and test, instead, we used the test set as the validation set. We did several tests, with different hyperparameters to find the best model.
The CUP's dataset is composed by a training set of 1524 records each one with 10 features and 2 labels. The test set is composed by 472 with no target prediction (*Blind Test Set*). For this type of task, we decided to split the original training dataset in 80% for training purposes and 20% as the internal test set.
To search the best hyperparameters, we used the cross-validation technique with 3-fold as a satisfying trade-off between the computation time and the estimation of the error.

3 Experiments

3.1 Monk Results

The Network Topology is composed of 17 units as the input layer, 5 units as the hidden layer and 1 unit as the output layer. We used `tanh` as activation the function of the hidden layer and the `sigmoid` as the activation function of the output layer, therefore it classifies as 1 if the output is greater than 0.5 and 0 otherwise, and using as loss function the `Mean Squared Error`. We tried stochastic, mini-batch and gradient descent. We also used regularization for the MONK 3 to control the complexity of the model and avoid to overfit as in Figure 4.

Best models for the MONK tasks								
Task	η	α	λ	**Batch**	**MSE (TR)**	**MSE (TS)**	**Accuracy (TR)**	**Accuracy (TS)**
$Monk_1$	0.1	0.8	0	32	0.0001233	0.0003103	100%	100%
$Monk_2$	0.1	0.9	0	32	0.0000272	0.0002712	100%	100%
$Monk_3$	0.1	0.9	0	'batch'	0.0262829	0.0427807	99 %	95%
$Monk_3\,(r)$	0.1	0.	0	'batch'	0.0653470	0.0666204	95%	98%

Table 1: MSE and Accuracy obtained for the three MONK's tasks on Keras. Note $Monk_3\,(r)$ refers to the MONK 3 task with regularization.

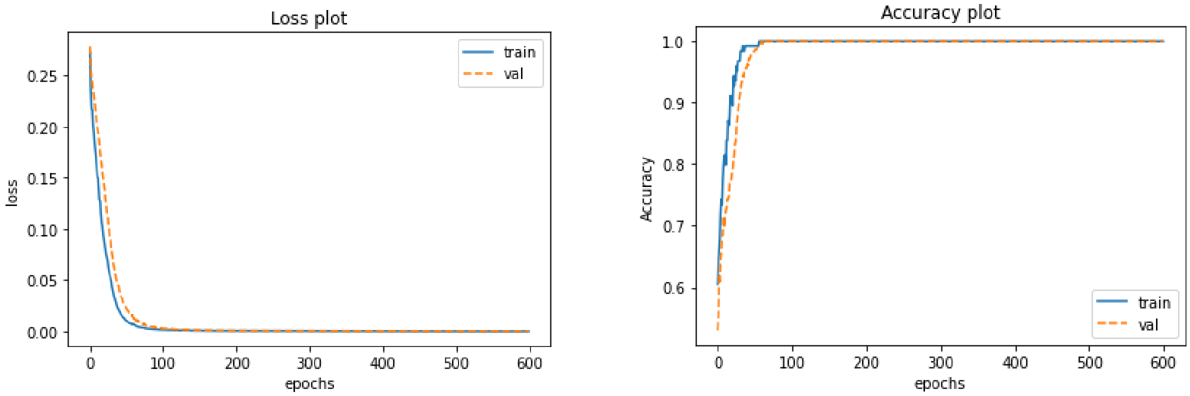

Figure 1: MSE and Accuracy for MONK's 1

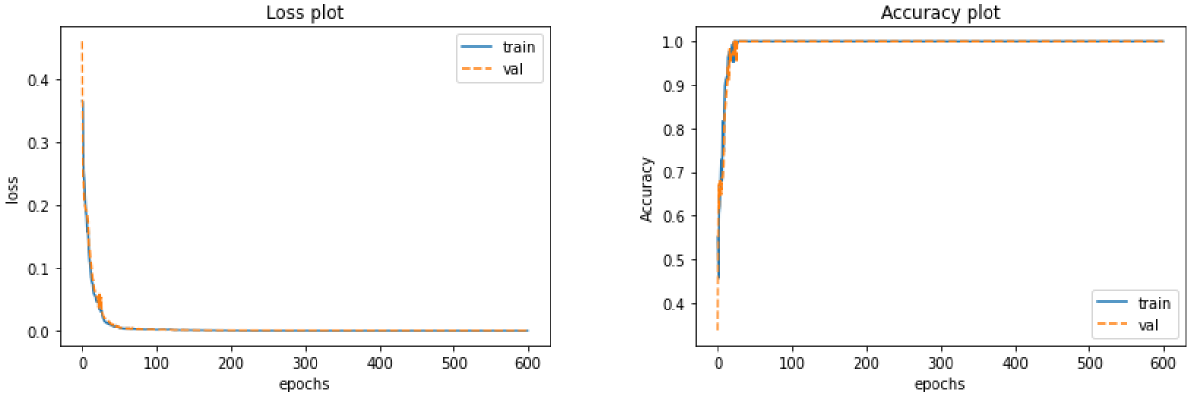

Figure 2: MSE and Accuracy for MONK's 2

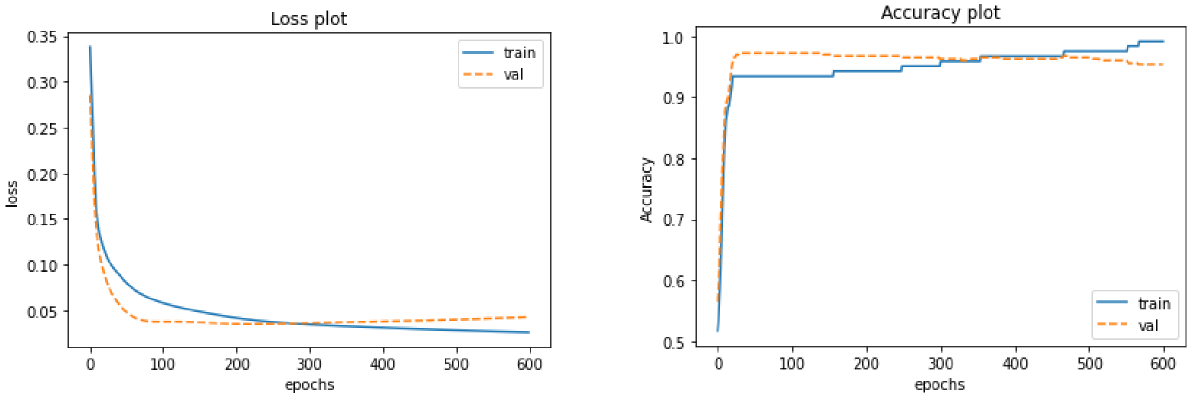

Figure 3: MSE and Accuracy for MONK's 3

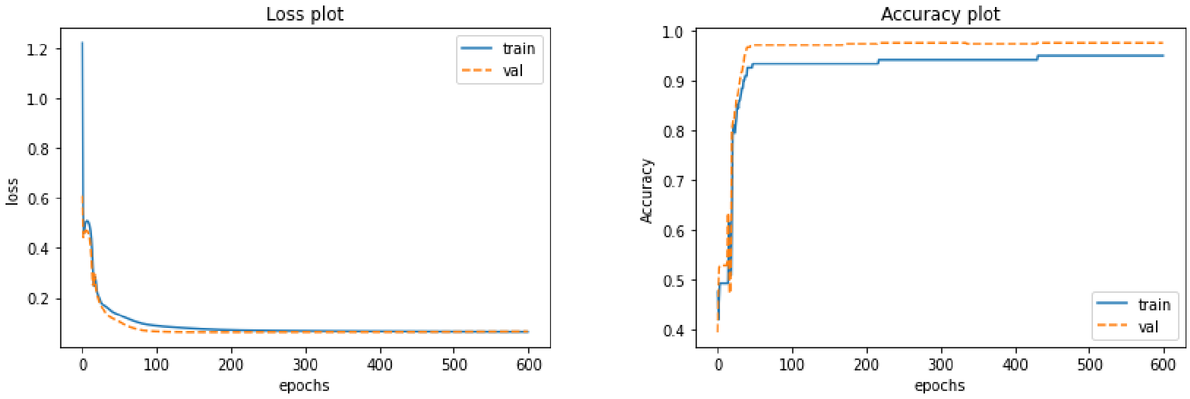

Figure 4: MSE and Accuracy for MONK's 3 regularized

3.2 Cup Results

3.2.1 Screening Phase

Firstly we tried one layer neural networks and different set of values for hyperparameters, to analyze the effects of them and discard them from the grid search, if the results weren't enough good. We tried to train the network with the `MSE` loss function and validate with the `MEE` loss function, but finally we have decided to use the `MEE` function for both train and validation steps. Then to tune the hyperparameters we started to perform first a large grid search and then a more granular one based on the previous results. Finally, as the output layer, we have used the `Linear` activation function, unlike the MONK's task, because it is a regression task.

3.2.2 Keras

Keras is a deep learning API written in Python, running on top of the machine learning platform TensorFlow [1]. It was developed with a focus on enabling fast experimentation.*Being* *able to go from idea to result as fast as possible is key to doing good research* [2]. As you can see from their definition it was designed directly for rapid experimentation, you can specify to create a sequential model and add how many layers you want, the activation function, the momentum, the regularization and so on without any particular effort.

3.2.3 Grid Search

Initially, as mentioned in the Screening Phase section, we devoted ourselves to a phase of hyperparameters tuning to try to define a first architecture of the model and a range for the most relevant hyperparameters. After this, we decide to execute different grid searches trying with different architectures and hyperparams obtained from the previous analysis. Finally, we get satisfying results using the Grid Search presented in the Table 2 which gave us the set of models presented in the Table 3.

Hyperparameters	**Values range**
# layer 1 units	130, 70
# layer 2 units	30
lr	0.0001, 0.001, 0.01
Momentum	0.4, 0.8, 0.9
λ	0.001, 0.0001, 0.00001
Epochs	1000
Batch-size	64, 128, 'Batch'
Nesterov	[True, False]
Activation	`reLu`, `tanh`, `sigmoid`

Table 2: Range of Hyperparameters for the Large Grid Search on Keras

Keras - Large Grid Search Scores								
# Units	**Batch**	**lr**	**Momentum**	$\boldsymbol{\lambda}$	**MEE (TR)**	**MEE (VL)**	**MEE (TS)**	**TR time**
[130 30]	128	0.01	0.9	0.0001	2.342	2.826	2.962	10.95
[130 30]	64	0.01	0.8	0.0001	2.370	2.846	2.977	20.07
[70 30]	128	0.01	0.8	0.0001	2.547	2.872	3.045	10.97
[130 30]	128	0.01	0.8	0.0001	2.557	2.873	3.010	10.89
[70 30]	128	0.001	0.9	0.0001	2.262	2.879	3.001	11.19

Table 3: Best five network configurations on the Large Grid Search with MEE

The first four models use the `sigmoid` function as activation and only the last one uses the `tanh`. All the five models use Nesterov momentum with a high value. Consequently, we decided to perform a more granular grid, according to the results obtained, presented in the Table 4.

Finally, we reached to improve the results as we can see in Table 5, all the networks use `sigmoid` activation function.

Hyperparameters	Values range
# layer 1 units	130, 70
# layer 2 units	30
lr	0.0001, 0.001, 0.006, 0.009, 0.01, 0.03
Momentum	0.7, 0.75 0.8, 0.85, 0.90, 0.95
λ	0.00001, 0.00009, 0.0001, 0.0003, 0.001
Epochs	1000
Batch-size	32, 64, 128
Nesterov	[True]
Activation	`tanh`, `sigmoid`

Table 4: Range of Hyperparameters for the Granular Grid Search on Keras

Keras - Granular Grid Search Scores								
# Units	**Batch**	**lr**	**Momentum**	$\boldsymbol{\lambda}$	**MEE (TR)**	**MEE (VL)**	**MEE (TS)**	**TR time**
[130 30]	128	0.01	0.9	0.00001	2.222	2.752	2.968	14.00
[130 30]	64	0.01	0.8	0.00001	2.255	2.756	2.978	21.57
[130 30]	64	0.01	0.7	0.00001	2.384	2.769	3.042	20.82
[130 30]	128	0.01	0.8	0.00001	2.458	2.781	2.927	14.14
[70 30]	128	0.01	0.7	0.00001	2.548	2.790	2.935	12.34

Table 5: Best five network configurations on the Granular Grid Search with MEE

3.2.4 Pytorch

Pytorch is an open-source machine learning framework that accelerates the path from research prototyping to production deployment [8]. It is a low-level framework which provides an interface to build custom networks, where the workflow can be summarized in: specifying the forward pass, the weights initialization, run the forward and backward pass, compute and update the weights resulting. It works in a more complex way with respect to Keras where you can use directly the `fit()` function after the definition of your model. Another issue is that Pytorch doesn't have any method to perform a grid search to tune the hyperparamenters, indeed it is necessary to use a third-part library called skorch [10] that provides a wrapper around Pytorch that has a scikit-learn interface.

3.2.5 Grid Search

Thanks to the Screening Phase we had acquired some intuition to choose the values of the hyperparameters for the grid search, therefore according to them, we had chosen to not include too much high learning rate or too much small batch size (Figure A7, Figure A8). In the Table 6 are reported the hyperparamenters and the values used, instead in the Table 7 are reported the best four models. For each model, the result batch size is 64 and all the models use Nesterov momentum.

We decided to perform a new grid search, based on the previous results, using the new hyperparameters range presented in the Table 8. The results are presented in the Table 9, the batch size of the models is 64 and in this case, every model uses the `sigmoid` as the activation function and Nesterov momentum.

Hyperparameters	**Values range**
# units	50, 75, 100, 150
lr	0.01, 0.001, 0.0001
Momentum	0.2, 0.4, 0.6, 0.8, 0.9
λ	0.001, 0.0001, 0.00001
Epochs	1000
Batch-size	64, 128, 'batch'
Activation	`reLu`, `sigmoid`, `tanh`

Table 6: Range of Hyperparameters for the Large Grid Search on Pytorch

Pytorch - Large Grid Search Scores								
Layer	**Units**	**lr**	**Momentum**	$\boldsymbol{\lambda}$	**MEE(TR)**	**MEE(VL)**	**MEE(TS)**	**TR time**
1	100	0.01	0.8	0.0001	2.432	2.831	3.102	81
1	50	0.01	0.8	0.0001	2.399	2.851	3.092	81
1	50	0.01	0.9	0.00001	2.405	2.851	3.159	81
1	100	0.01	0.9	0.0001	2.374	2.855	3.173	81

Table 7: Best four network configurations on the Large Grid Search with MEE

Hyperparameters	**Values range**
# units	50, 100
lr	0.01, 0.03, 0.06, 0.006, 0.007, 0.008, 0.009
Momentum	0.7, 0.8, 0.85, 0.9, 0.95
λ	0.0001, 0.0003, 0.0006, 0.00001, 0.00003, 0.00006, 0.00009
Epochs	1000
Batch-size	64, 128
Activation	`reLu`, `tanh`, `sigmoid`

Table 8: Range of Hyperparameters for the Granular Grid Search on Pytorch

Pytorch - Granular Grid Search Scores								
Layer	**Units**	**lr**	**Momentum**	$\boldsymbol{\lambda}$	**MEE(TR)**	**MEE(VL)**	**MEE(TS)**	**TR time**
1	100	0.006	0.9	0.0001	2.365	2.819	3.121	51
1	100	0.01	0.8	0.00001	2.418	2.829	3.089	51
1	100	0.06	0.9	0.0003	2.395	2.831	3.107	50
1	50	0.001	0.9	0.0003	2.361	2.831	3.343	51
2	[100, 50]	0.006	0.8	0.00001	2.439	2.782	3.041	99
2	[100, 50]	0.006	0.8	0.0001	2.434	2.788	3.045	90

Table 9: Best six network configurations on the Granular Grid Search with MEE

3.2.6 SVR

The last model tested is a particular model of Support Vector Machine, the Support Vector Regressor, capable to deal with regression tasks. Since we had 2 output labels we decided to use the MultiOutputRegressor [4] function of scikit-learn, because the SVR provides just 1 output as default. The SVR uses three fundamental parameters to control the complexity:

- ***epsilon***: which gives the radius of the epsilon-intensity tube
- ***gamma***: which defines how far the influence of a single training example reaches, with low values meaning 'far' and high values meaning 'close'
- ***C***: as regularization parameter, which for larger values will use smaller margin and for lower values it will encourage larger margin

As in the previous framework, we decided to perform a grid search with the parameters presented in the Table 10 which obtains the following results in the Table 11.

Hyperparameters	**Values range**
C	0.1, 1, 10, 100, 1000
Gamma	0.0001, 0.001, 0.005, 0.01, 0.1, 1, 3, 5
Epsilon	0.0001, 0.0005, 0.001, 0.005, 0.01, 0.05, 0.1, 0.5, 1, 5, 10
Kernel	`rbf`, `linear`, `sigmoid`

Table 10: Range of Hyperparameters for the Large Grid Search for the SVR

SVR - Large Grid Search Score						
Epsilon	**C**	**Gamma**	**MEE (TR)**	**MEE (VL)**	**MEE (TS)**	**TR time**
0.5	10	0.1	2.481	3.055	3.055	0.20
1	10	0.1	2.574	3.0631	3.066	0.16
0.1	10	0.1	2.426	3.069	3.068	0.22
0.05	10	0.1	2.421	3.073	3.075	0.23
0.01	10	0.1	2.419	3.076	3.080	0.23

Table 11: Best five SVR configurations on the Large Grid Search with MEE

Given the results of the Table 11, we assumed to use Radial basis function as kernel and we executed a more granular grid on:

Hyperparameters	**Values range**
C	10, 11, 12, 13, 14, 15, 16, 18, 19, 20, 25, 30
Gamma	0.01, 0.03, 0.05, 0.07, 0.08, 0.09, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1, 2, 3, 5
Epsilon	0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1

Table 12: Range of Hyperparameters for the Large Granular Search for the SVR

SVR - Granular Grid Search Score						
Epsilon	**C**	**Gamma**	**MEE (TR)**	**MEE (VL)**	**MEE (TS)**	**TR time**
0.9	14	0.08	2.5549	3.0039	3.098	0.17
0.9	13	0.09	2.5190	3.0040	3.087	0.17
0.9	13	0.08	2.5700	3.0041	3.102	0.17
0.9	15	0.08	2.5409	3.0043	3.092	0.18
0.9	12	0.09	2.5360	3.0043	3.095	0.17

Table 13: Best five SVR configurations on the Granular Grid Search with MEE

3.3 Computing Time Comparison

All the computations were executed in our machines, which have this specifics:

- Intel(R) Core(TM) i5-4460 CPU @ 3.20GHz, AMD Radeon R7 370 and 8GB of RAM
- Intel(R) Core(TM) i7-1065G7 CPU @ 1.30GHz, NVIDIA GeForce MX250 and 16GB of RAM

We trained and evaluated the model 10 times to compare the different models and frameworks. Given the results in Figure 5 we can see how Pytorch is much slower than Keras and how, in terms of training time, the best one is the SVR.

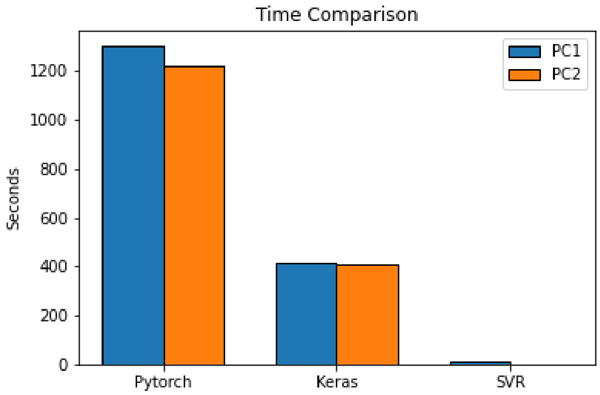

Figure 5: Comparison of frameworks on our PC's

3.4 Final Model

The best models were chosen from the results of the Pytorch in the Table 9, Keras in the Table 5 and SVR in the Table 13. We retrained the models on 90% of the dataset and validate on the 10%, we did it 10 times for each model to avoid bad weights initialization and to get the average error. In the Table 14 we present the final model which is a Neural Network obtained with the Keras framework, it uses a batch-size of 128 with Nesterov momentum.

Type	Layer	Units	lr	Momentum	λ	MEE(TR)	MEE(VL)	MEE(TS)
Keras	2	[130, 30]	0.01	0.8	0.00001	2.560	2.658	2.914

Table 14: Best models

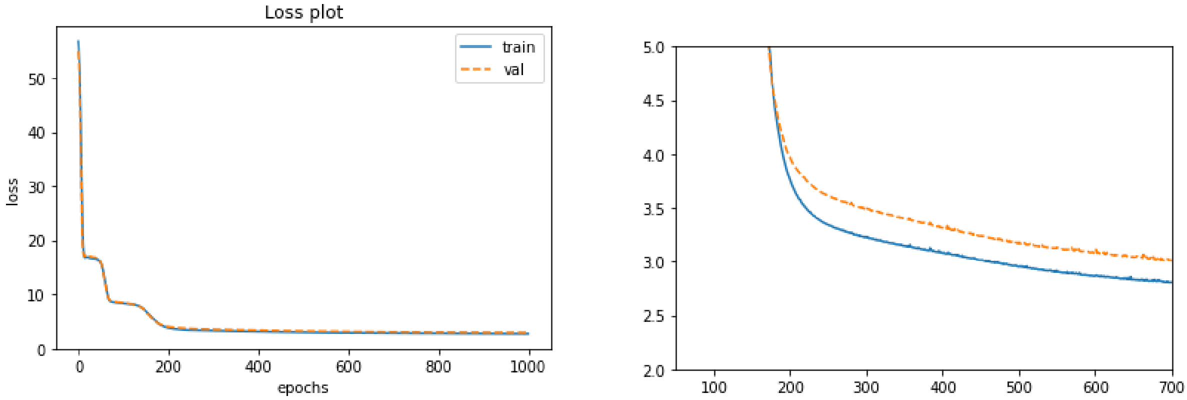

Figure 6: MEE loss and zoomed MEE loss of the final model.

4 Conclusions

The project gave us the possibility to try different frameworks and models to better understand the theory we studied during the course. After this, we can assert that Keras allows developing faster than Pytorch because of the higher level of abstraction paid with a worse flexibility. Moreover, the scikit-learn library allowed us to test a different model, the SVR, of which initially we were a little more skeptical, given the fame the Neural Networks carry with them. We had to reconsider the SVR because not only the performances are similar but also allow a faster training phase. Besides, thanks to MONK we were able to test how regularization is particularly important for the realization of a good model. This first phase of hyperparameters tuning was crucial for us to gain a better understanding of the theory, which in turn was necessary for approaching the CUP. Finally, all this has taught us the relevance of a well defined grid search and how much is important the use of validation techniques to produce a model how much more generalizable possible.

NICK: Marvil
BLIND TEST RESULTS: Marvil_ML-CUP20-TS.csv

Acknowledgments

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

References

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[2] François Chollet et al. Keras. https://keras.io, 2015.

[3] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In Yee Whye Teh and Mike Titterington, editors, *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, volume 9 of *Proceedings of Machine Learning Research*, pages 249–256, Chia Laguna Resort, Sardinia, Italy, 13–15 May 2010. JMLR Workshop and Conference Proceedings.

[4] https://scikit learn.org/stable/modules/generated/sklearn.multioutput.MultiOutputRegressor.html.

[5] J. D. Hunter. Matplotlib: A 2d graphics environment. *Computing in Science & Engineering*, 9(3):90–95, 2007.

[6] Wes McKinney et al. Data structures for statistical computing in python. In *Proceedings of the 9th Python in Science Conference*, volume 445, pages 51–56. Austin, TX, 2010.

[7] Travis Oliphant. NumPy: A guide to NumPy. USA: Trelgol Publishing, 2006–. [Online; accessed ¡today¿].

[8] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. 2017.

[9] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[10] Marian Tietz, Thomas J. Fan, Daniel Nouri, Benjamin Bossan, and skorch Developers. *skorch: A scikit-learn compatible neural network library that wraps PyTorch*, July 2017.

Appendix A

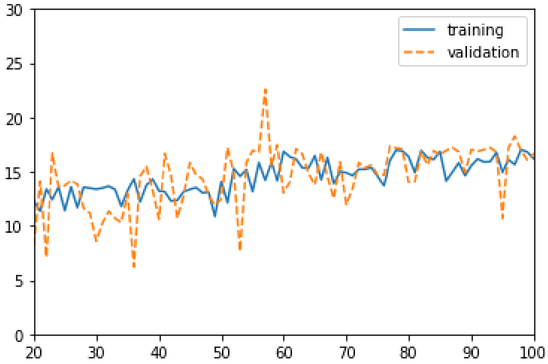

Figure 7: High learning rate

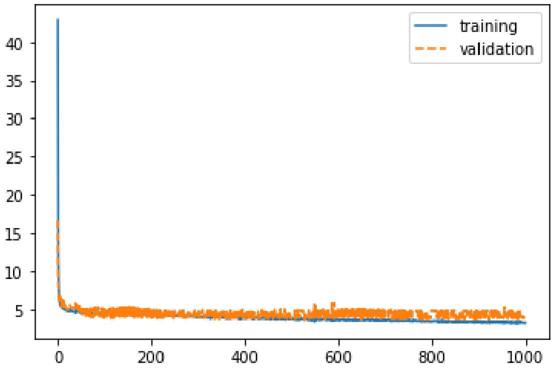

Figure 8: Low batch-size

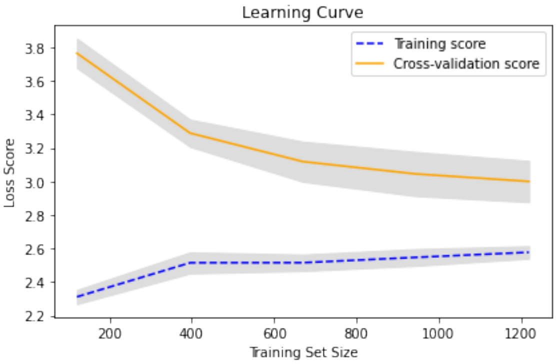

Figure 9: SVR - Learning Curve

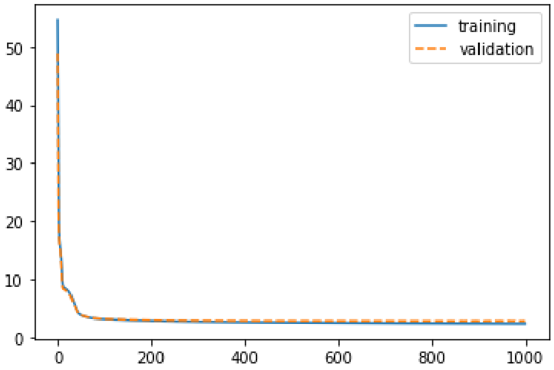

Figure 10: Pytorch - Final Learning Curve

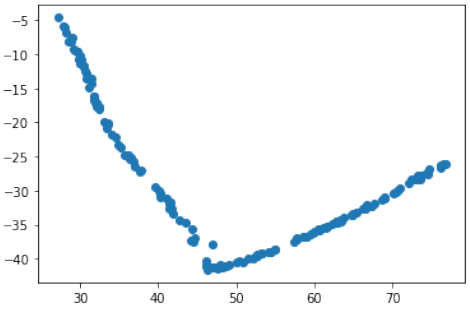

Figure 11: Keras model prediction

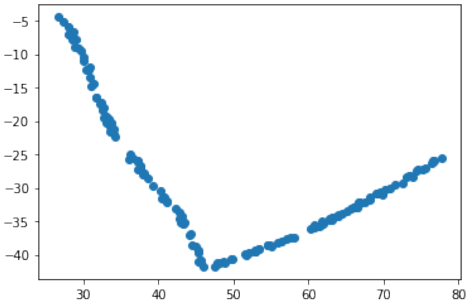

Figure 12: Real internal test set

Appendix B

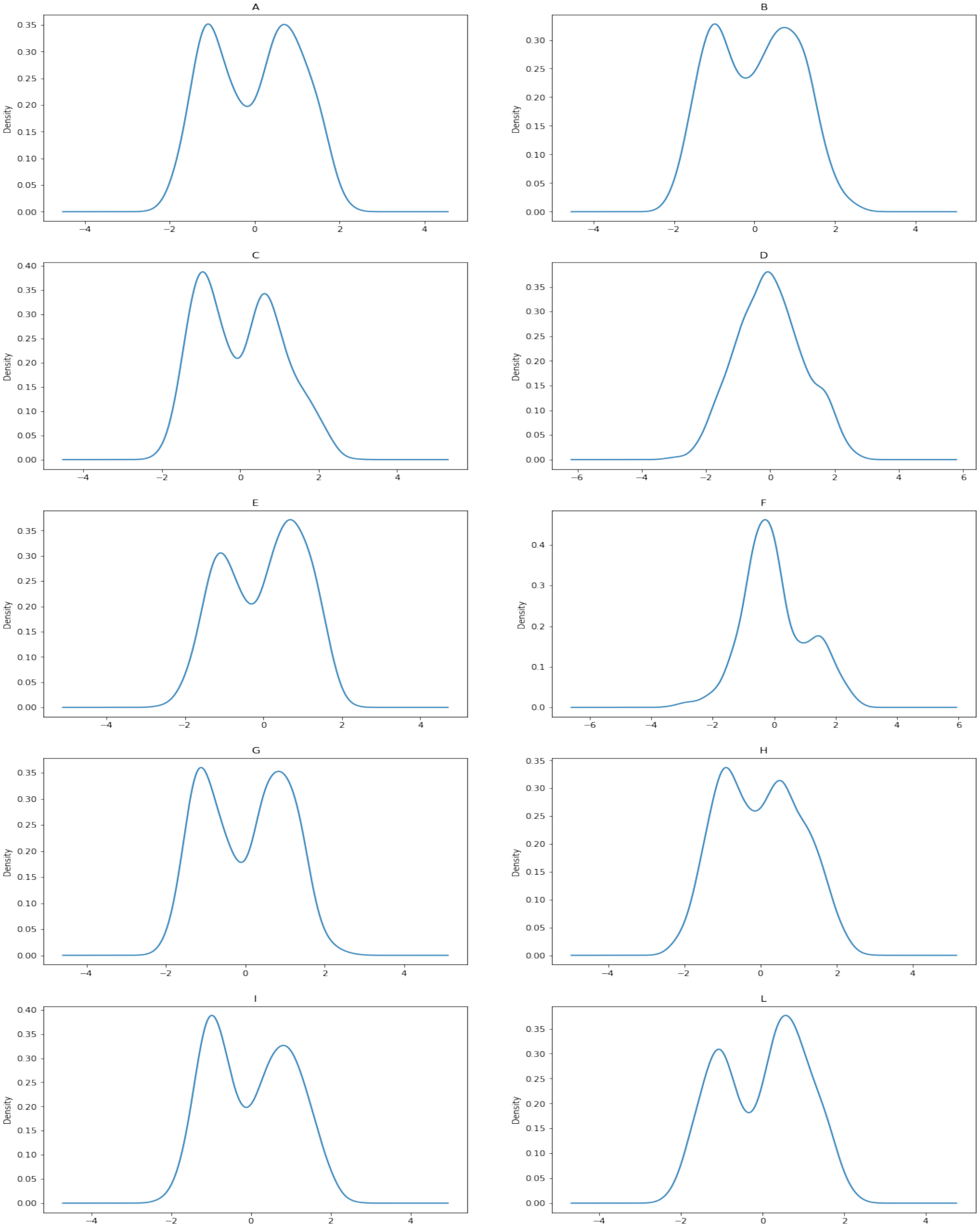

Figure 13: Distribution of the features

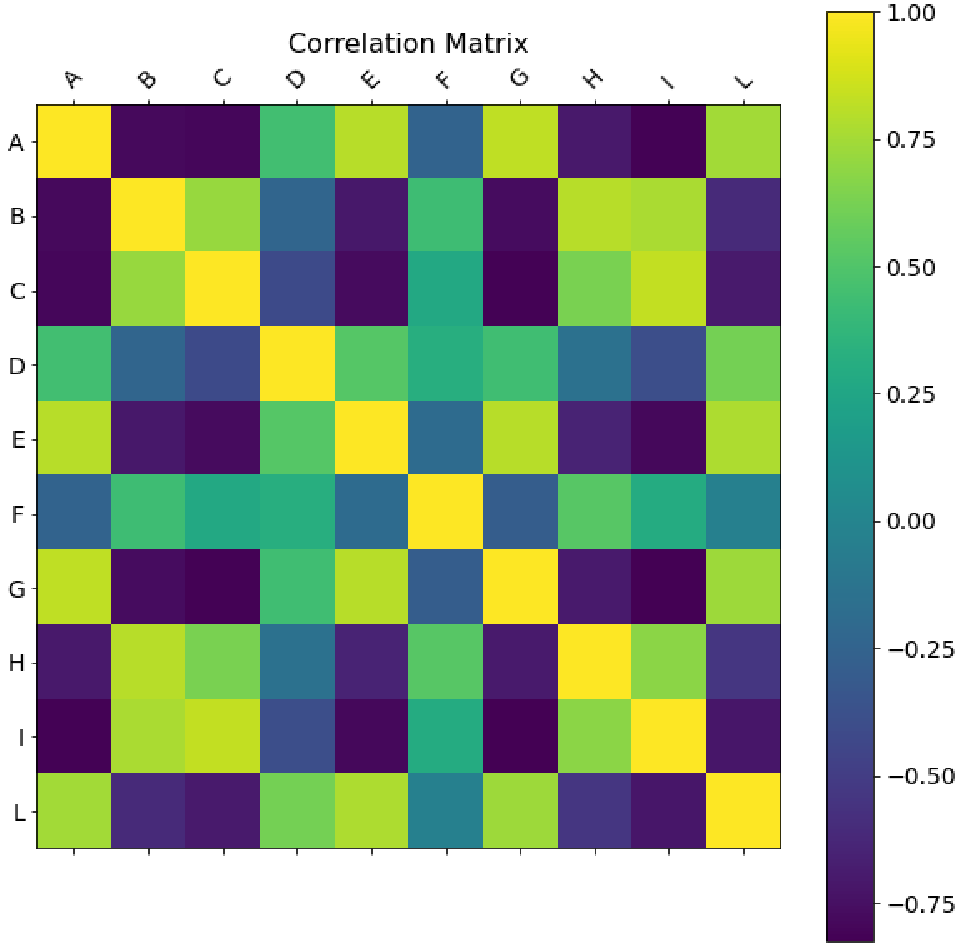

Figure 14: Correlation Matrix