

# A comparison between three kinds of models

Authors (*Antonio Boffa, Davide Montagno B.*)

Email ([a.boffa@studenti.unipi.it](mailto:a.boffa@studenti.unipi.it) , [d.montagno@studenti.unipi.it](mailto:d.montagno@studenti.unipi.it) )

Exam: ML 654AA

Academic Year: 2019/2020

Date: 29/01/2020

Type of project: **B**

## ABSTRACT

After receiving the *task* to solve, our goal was to realize multiple solutions to solve it. We compare two different types of really important and famous frameworks for Neural Network: Keras[1] and Pytorch[2]. Then we compare the approach based on Neural Network against a completely different approach in Machine Learning, known as SVM. Lastly, we first validate, by cross validation, and then test all the models on two different machines to see the performances.

## 1. INTRODUCTION

Our first aim is to show our journey from having any previous knowledge of the frameworks to being able to write a two fully working Multi-Layer Perceptrons networks (MLP) with two different shapes. For each of these two frameworks, we use two different structures to better understand how they work and how to extend them. The development has been really tough because of the difference between the two frameworks.

We were curious about how to deal with Support Vectors Machine (SVM) and so, we choose it too; since we are dealing with a regression task we use a different kind of SVM, known as Support Vectors Regression (SVR) implemented by scikit-learn[3]. The choice of these three models has been done to test all parts of course and to solve in the best way possible the given task. Talking about the task, we have 20 features and 2 target values, each of these are real numbers (they come from sensors); no any other assumption of the data has been done. We have found interesting patterns in the target values; we will discussed more in detail later in this report. In order to test our models we used Hold-out technique initially, to split the data and then to validate the model; particularly, we used k-fold cross-validation and grid search.

The developed code has been done by Python 3.7. The code is completely automatized; from the main script the user can interact with the models by choosing either if he wants to cross-validate or test one particular instance of a single model, or all of those. All the code is available on Github [4].

## 2. METHOD

First of all, we chose some main functionality that we were sure to be common to all the model or at least common in the Neural Network models, they are: data import, data visualization, loss function (Mean Euclidian Distance), validation schema and test schema. Once we chose how to organize the shared parts, we start to implement them:

- For data import and other small tasks of numeric values manipulation, we used *numpy* library [5].
- For plotting learning curves and results of predictions in a 2D plot, we used *matplotlib* [6].
- Since the results on the blind set will be evaluated using the Mean Euclidian Error, we used it too as a *loss* in order to have more similar results. About the implementation of this function, for Keras we used the one provided by the framework; for Pytorch we implemented it by ourself using tensors. For SVM\*//\* too.\*/\*, we used just this function to evaluate the results using numpy lists instead of tensors. (Non la utilizziamo in tutte le fase di training, validazione e internal test?)
- For the Grid Search, we could use some automatic toll online to realize it, instead we decided to implement it on our own. We did it with just 3/4 nested *for* loops, with no parallelization. In the phase of grid search we also plotted every learning curve; Although we produced a huge amount of images; of course, in this report we will show you just the most relevant.
- As activation function for the MLP we used and tested both Sigmoid and Rectified Linear Unit (ReLU)
- We chose to use Stochastic Gradient Descend as learning algorithm because during the course, we understood very well how it works; moreover, it provides suitable smoothness properties for optimizing problems.
- About the preprocessing, we didn't do any kind of it in the beginning, not even plotting the targets in a 2D plot. We did it in the end; more detail will be discussed in Chapter 3.
- We chose Mini batch learning; the batch size instead, was chosen in the validation phase.
- As weights initialization technique we used the Glorot initiliazation (also known as Xavier). It initializes each weight with small Gaussian values with mean = 0.0 and variance based on the fan-in and fan-out of the weight.
- At the beginning of the project, we splitted the dataset into two parts, the first one consists in randomly chosen 1.595 samples (90.09% of the entire dataset); the second one of 170 examples (9.91% of the entire dataset). The latter, is used to test all the three models at the end of the validation process to choose the best one and give a true estimation of its generalization error.
- As validation schema, we used K-Fold cross-validation implemented by scikit-learn to not have any bias in the chose of the validation set. We always shuffled the sample before the division in folds. We use  $k = 10$  because for each fold we have 90% of the dataset for training and 10% for validation. For last score for a model we used the average over all the losses of the folds.

We started using Keras with TensorFlow backend [7] considering 3 hidden-layers perceptron and a quite small amount of units per layer. We also experimented a “pyramid” shape, where at each level there are fewer units than the previous one.

For the Pytorch model, we decided to implement a two hidden layers perceptron and a quite high amount of units per layer without any particular shape. In both Neural Network models all the connection between a level and its next one are dense.

As we said in the Introduction chapter, with the MLP we wanted to experiment a different approach to Machine Learning, and so we started to look for a good SVM framework to use. We found out that the best one is *libsvm* [8]. Instead of using it, we used the scikit-learn implementation based on *libsvm*. It provides a better integration in Python and the other utilities of scikit.

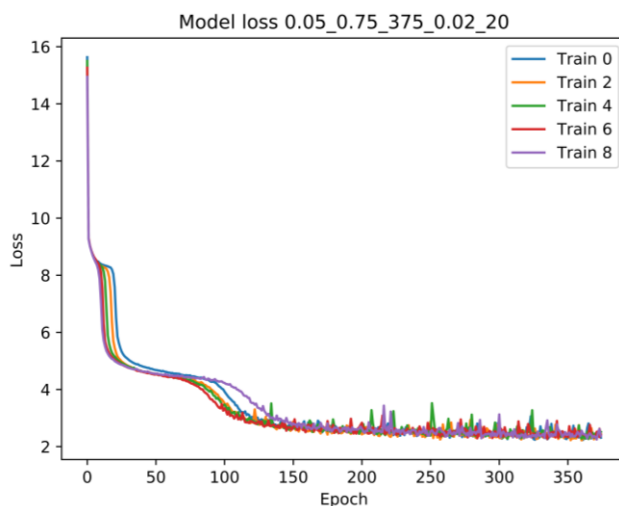
Since we were dealing with a regression task we used SVR, a special kind of SVM. They share the same principles, but it SVR differs on use an epsilon insensitive loss and two slack variables for each data point.

### 3. EXPERIMENTS

Once we chose the main structure of the models and the techniques to use, we started to develop the models and tune the hyperparameters.

Let’s talk shortly of how we plotted the learning curves of our Neural Network models. Every point that we plotted is defined in this way: (number of epoch, metrics at the end of that epoch), we used metric in sens of loss or accuracy. For “at the end of that epoch” we mean the value of metric after a number of updates = (number of samples / batch size). Since we ran a ten fold cross validation we should plot ten learning curves. In order to not have such confused plots, we just consider four or five random fold, and we plot them either on validation set and training set. In this type of plots it’s not important to understand which fold goes well or not, but just the variance in their behaviour on large scale. So, it’s ok to see them in black and white. In the Section 3.3 we talk about the SVM plot.

#### 3.1 KERAS Multi Layer Perceptrons



**Figure 1 Learning curve of an unstable model.**  
 $\eta = 0.05$ ,  $\alpha = 0.75$ , epochs = 375, lambda 0.02, batch size = 64

Implementing the basic structure in Keras was quite easy and quick; in our opinion, this framework works with high level of abstraction. It provides every aspect of a Neural Network; obviously, as a drawback it doesn’t let the programmer has the control of some aspect of what it is going on under the hood.

Thanks to the online documentation we started to develop our idea composed by 3 hidden-layer NN with this structure: the first one (the nearest to the input layer) consists of 26 unit, the second of 24 and the final hidden layer of 22 units. So following this rule, with

$c = 2$  we construct the Neural Network as:

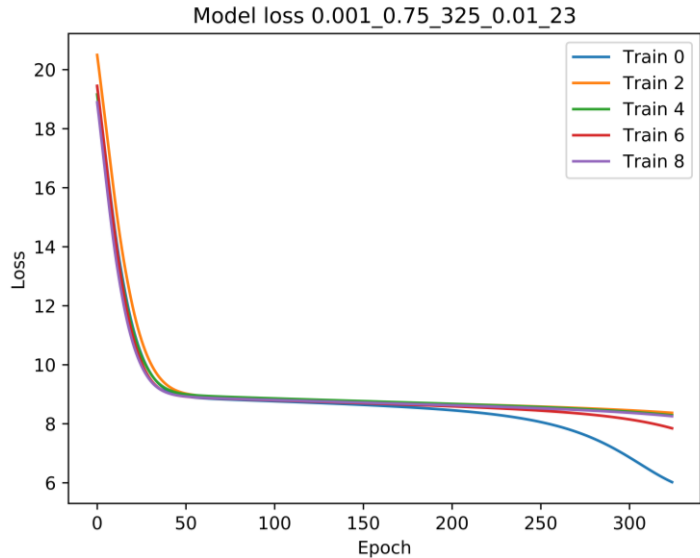
$$Unit\_at\_level_i = Unit\_at\_level_{i-1} - c$$

Then we started the grid search with a value of *eta* (learning rate) too high. For this reason our first models were really unstable as we can see in **Figure 1**.

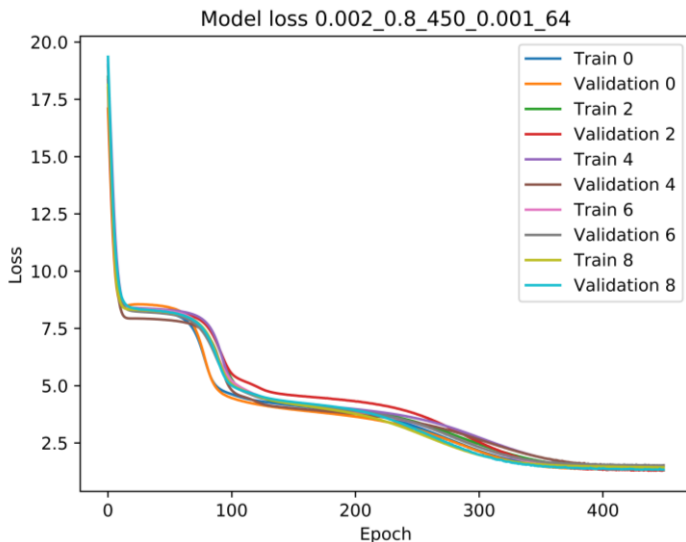
Then we restarted the grid search with a low value of *eta* around 0.001, in order to let the model go slow and stable to the solution. As *alpha* (momentum parameter) around 0.8 as *lambda* (penalization term for norm 2 regularization) 0.02 and batch size 64.

In the this initial phase of grid search there was some other problem, for example we selected a set of *lambda* too hig and the model was unable to learn, and so it went in underfitting. As we can see in **Figure 2**.

At this phase we chose the sigmoid activation function, and after some steps over the grid search, we found some good parameters in terms of smoothness of the learning curve and loss on the validation set (around 1.5). We found a weird behaviour, more or less, in all plots; there were a “stairs shape” with some plateaux and some part where loss function had steep descend. Moreover, the needed epochs were too much high (around 450). As we can see in **Figure 3**.

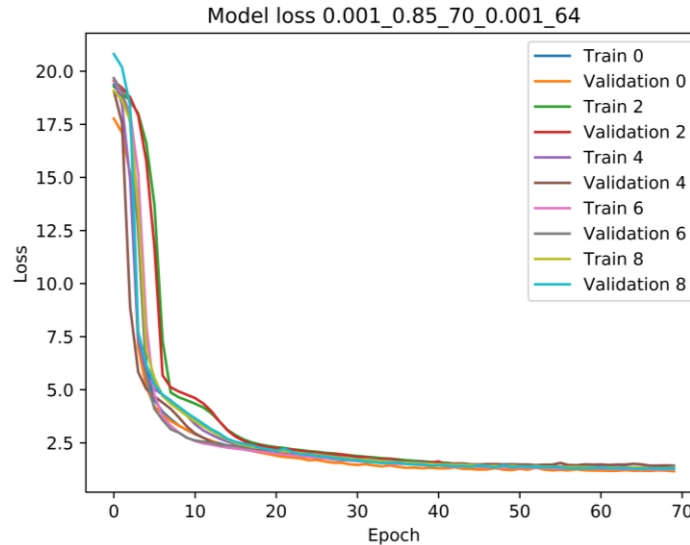


**Figure 2 Learning curve of unfitted model.**  
eta = 0.001, alpha = 0.75, epochs = 325,  
lambda 0.01, batch size = 23



**Figure 3 Learning curve of the first model with sigmoid activation function.**  
eta = 0.002, alpha = 0.8, epochs = 450,  
lambda 0.001, batch size = 64

After many other experiments we decided to use ReLU activation function, since we had 3 layers; in fact, it has been demonstrated that ReLU ensures better training of deeper networks [10]. This modification changed completely the learning curves. Indeed, with it we obtained the same results with fewer epochs (just 70!) than sigmoid, and there wasn't any more weird plateaux and “stairs”, as we can see in figure **Figure 4**.



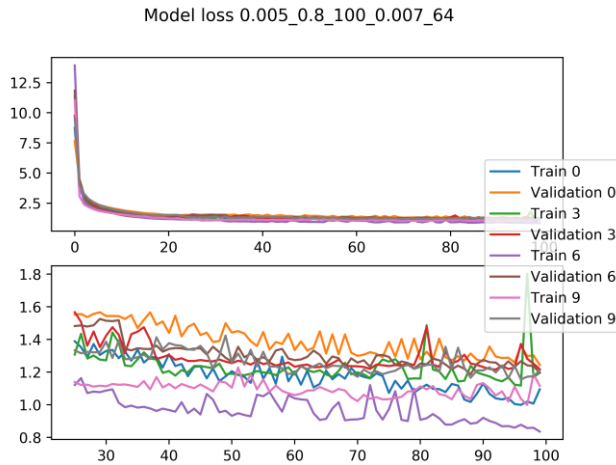
**Figure 4 Learning curve of the first model with ReLU activation function**

Then we also found out that the same results were reachable with fewer input, so we increased the  $c$  of the formula to 3, in order to have our levels with 25-22-19 units. Since we changed the activation function and a bit the number of units, we started again the grid search for the hyperparameters. We found what in our opinion were the best hyperparameters:  $\eta = 0.001$ ,  $\alpha = 0.84$ ,  $\lambda = 0.0006$ , batch size = 64, number of epochs = 170. The average result on the training set, validation set, and the result on our internal test set is in Section 3.5. We can see the final plot of the learning curve of this model in **APPENDIX A1**.

### 3.2 PYTORCH Multi Layer Perceptrons

The first thing to say about this framework is that it has a lower level of abstraction; for example, we need to implement the *forward* [10] function of the net and so how the net computes its output. Nevertheless, it provides high level functionality like *autograd* [10] for the execution of the backpropagation. In this framework we decided to build a different MLP, with one layer less than Keras and more units per layer; so, we had two hidden layer and 35-45 units per layer. The main issue compared to the previous framework was the implementation of the loss function. As we said in Pytorch, we implemented this Mean Euclidian Error on our own using tensors to work faster on GPU using *cuda semantic* [11], in order to make the tensors operation be done efficiently by our Nvidia© GPU and not simply in CPU. We will show you later, how our different Nvidia© GPUs works for the same task. Another issue found during the implementation of this model was the construction of mini-batch algorithm. In fact, we did it all on our own moving indexes according to the batch size. With the knowledge acquired in the previous model, the hyperparameters tuning was faster than before. For the same reason explained in the previous model, we always use ReLU activation function. In the first part, we thought that the learning curves were smooth. But, during the construction of the plots we found out another small problem, if we zoom in the epochs, the learning curves were a bit shaky as we can see in **Figure 5**. This discovery was

useful not only to understand the behaviour of the learning curves at the ending part of the training phase, but also to visualize better how the different curves of training and validation were distant from each other. So, we changed our plotting technique accordingly as we can see in **Figure 5**.

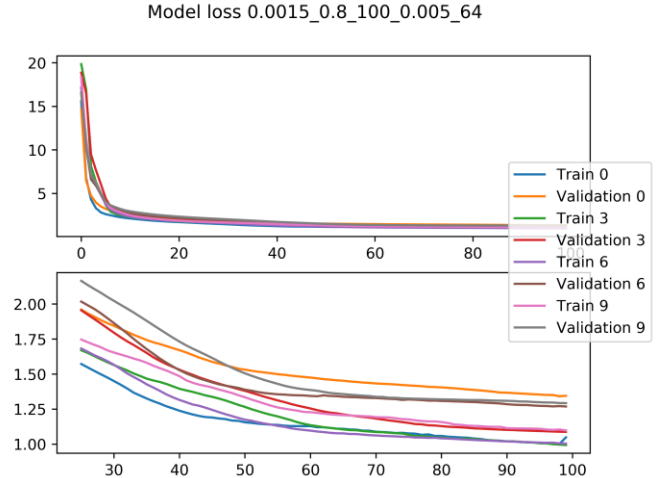


**Figure 5**

Top: Learning curve of epochs[0,100]

Bottom: Learning curve of epochs [25,100]

eta = 0.005, alpha = 0.8, epochs = 100, lambda 0.007,  
batch size = 64



**Figure 6**

Top: Learning curve of epochs[0,100]

Bottom: Learning curve of epochs [25,100]

eta = 0.0015, alpha = 0.8, epochs = 100, lambda 0.005,  
batch size = 64

Once found this issue, we restarted again a new phase of hyperparameters tuning, with less values than before. After that, the graphs were a lot smoother in the ending part of the learning phase. As we can see in **Figure 6**. Our best hyperparameters are: eta = 0.0015, alpha = 0.9, lambda = 0.002, batch size = 64, number of epochs = 135. The average result on the training set, validation set, and the result on our internal test set is in Section 3.5. We can see the final plot of the learning curve of this model in **APPENDIX A2**.

### 3.3 SCIKI-LEARN Support Vector Regresion

One of the biggest issue found in this model was the concept of learning in a totally different approach. As seen in the course, an important choice, to the construction of this model, regards the kernel to use. So, our starting point was that; the best one for theory part [citation] was Radial basis function (RBF), so we decided to use it. Our model needs three paramters, each of one controls the complexity of the system. The first is *gamma* that define how far the influence of a single training example reaches, with low values meaning “far” and high values mining “close”. The second parameter that we use is *C*, the third is *epsilon*. We started the grid search over the space to find the best value that minimizes the loss. As we said before, the task given is of regression, instead of use the simplest version of SVM, we used SVR. This model is different from the first one because of it use an epsilon (third parameter) to construct the epsilon-insensitive tube. Once we found what we wanted to tune, we found that if we use only the SVR, we cannot solve the task because of it is a multiple

regression task. Of course it is possible to solve the same task considering one regressor per target. At the beginning, we tried to do it by our own, but in order to be sure that all the execution runs efficiently, we decided to use a support library *MultiOutputRegressor*[12] provided by scikit-learn. Another import issue was the plotting of the learning curve; here the concept of learning is completely different because of we don't find epochs; instead, we work only with training sample. As we studied, the learning curve of SVR is composed by (number of example, loss). It implies that the system initially start to learn with a low error, and then start increasing with the number of example.

Since we were totally new with this type of model we started in a wide range of exponential, values = [0.01, 0.01, 0.1, 1, 10]. Because of the datatest is noise and we haven't enough

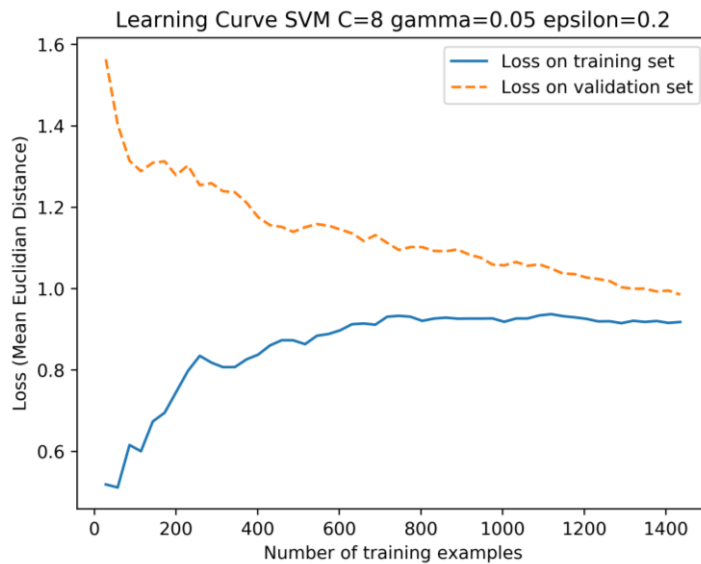


Figure 6 Learning curve of the final SVR model

confidence we didn't try values bigger than 10 since with high C less training error are allowed. As gamma we started with a small value in order to let the Gaussian function has a large variance. We also checked a range of exponential values for epsilon. In the APPENDIX A3, we can see some of the Learning Curves generate by the grid search.

The average result on the training set, validation set, and the result on our internal test set is in Section 3.5. We can see the plot of the learning curve of the best SVM model in Figure 6.

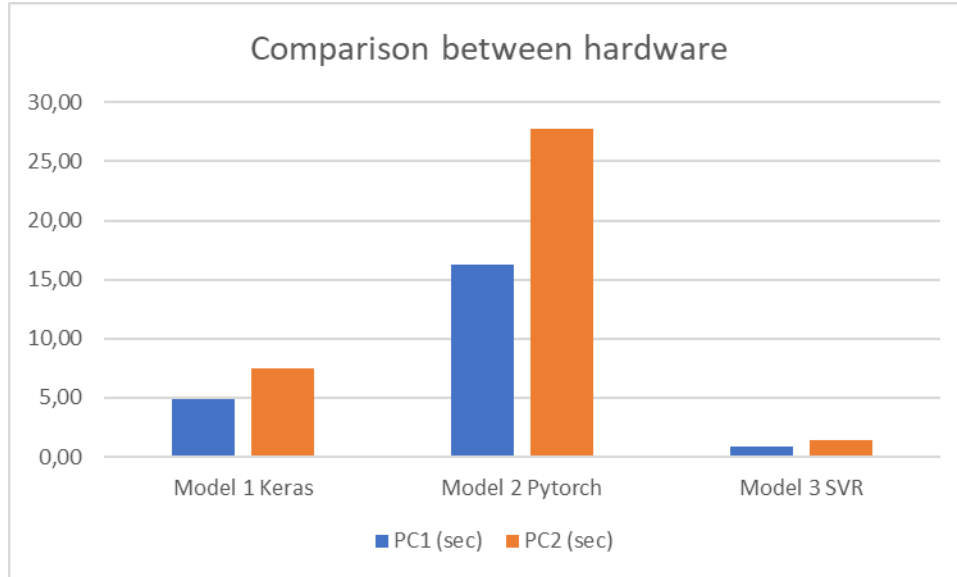
### 3.4 Comparison Nvidia© GTX 850M - Nvidia© GTX 1650M

Our laptops are:

- **PC1:** Asus N56J, Intel® Core™ i7 – 4710HQ, NVIDIA® GeForce® GTX 850M, 16 GB RAM, Windows 10.
- **PC2:** Dell Inspiron 7590, Intel® Core™ i7 – 9750H, NVIDIA® GeForce® GTX 1650M, 16 GB RAM, Windows 10.

In Figure 7 there is a simple bar chart that visualize the time in seconds needed by our laptops to execute all this steps: 1) train and evaluate the models in 10 folds 2) train again the models on the whole dataset 3) predict the result for the blind set. 4) plotting the results.





**Figure 7 Comparison of time needed by the models**

As we can see Pytorch got the worst performances than the other, also because our implementation has some problem in allocating tensors from *numpy* n-dimensional array.

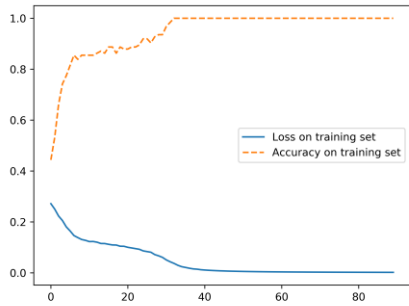
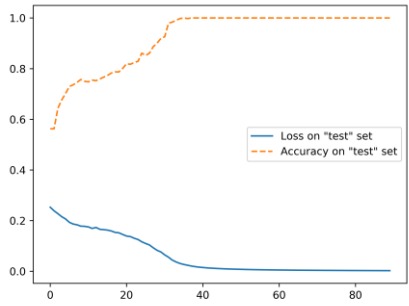
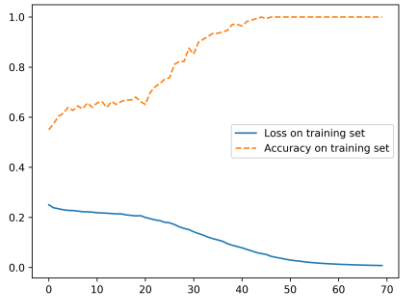
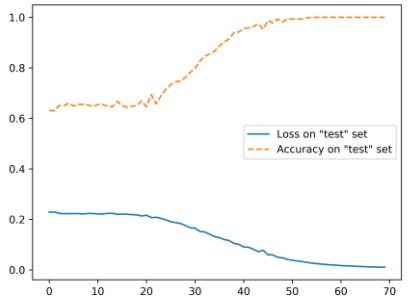
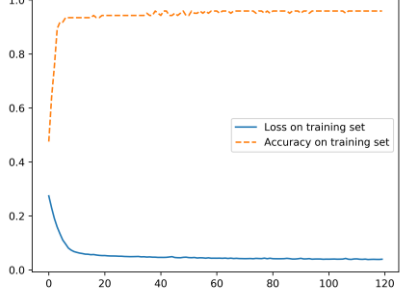
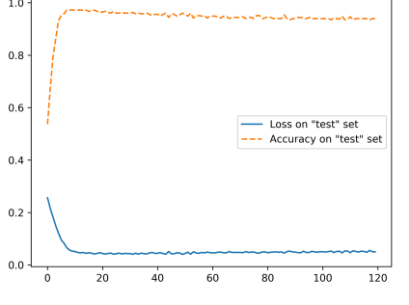
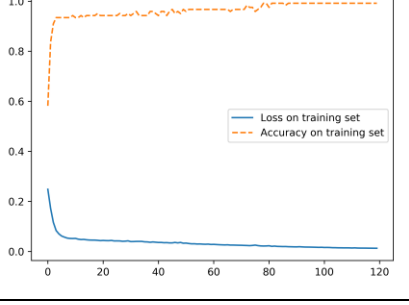
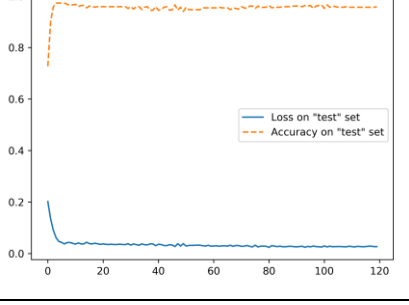
### 3.5 MONK'S RESULTS

To solve monk dataset we use a simple Keras MLP with one hidden layer of 4 units. All tests use batch size = 25.

| Task         | Eta  | Momentum | Lambda | MSE (TR/TS)   | Accuracy (TR/TS) (%) <sup>i</sup> |
|--------------|------|----------|--------|---------------|-----------------------------------|
| MONK 1       | 0.25 | 0.85     | 0      | 0.0016/0.0024 | 100%/100%                         |
| MONK 2       | 0.2  | 0.75     | 0      | 0.0007/0.0105 | 100%/100%                         |
| MONK3        | 0.2  | 0.75     | 0      | 0.0395/0.496  | 95%/93%                           |
| MONK3 (reg.) | 0.4  | 0.75     | 0.0001 | 0.0128/0.0270 | 99%/95%                           |

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



|              | Plot of loss and accuracy Training Set  | Plot of loss and accuracy Test Set   |
|--------------|---|--|
| MONK 1       |    |    |
| MONK 2       |    |    |
| MONK 3       |   |   |
| MONK 3 (reg) |  |  |

**Table 2.** Plots of the MSE and accuracy for the 3 MONK's benchmarks.

### 3.5 CUP RESULTS

In the end of model selection process, we wrote a script that tests all the final models on our internal test set. Results on training and validation are the average over the 10 folds. The plots and the results are in Table 2 and Table 3:

|                 | Training Set | Validation Test | Test Set |
|-----------------|--------------|-----------------|----------|
| Model 1 Keras   | 1.1552       | 1.2756          | 1.2353   |
| Model 2 Pytorch | 1.0385       | 1.1882          | 1.2081   |
| Model 3 SVR     | 0.9096       | 1.0457          | 1.0632   |

**Table 3 Final results**

Given this results, we choose Model 3 SVR.

### 4. CONCLUSIONS

In the APPENDIX B, we can see the plots in a cartesian plane of the results of our models just to “visual compare” the different shapes.

To summarize what we did and what we understood from this experience, we noticed that with an level of abstraction Keras permits to develop faster the code than Pytorch. Nevertheless, if you know what you are doing (in terms of theory background), Pytorch is more accurate than Keras because of the lower level of abstraction, since it permit a strongly control over the code. In the end we can say that SVM are very powerfull tools and in particular libsvm (and other libraries based on it, like scikit-learn) implements efficiently all the functionalities; also, it provides a quite easy programming interface to construct an high performant model.

**NICKNAME:** BOFMON

**BLIND TEST RESULTS:** BOFMON\_ML-CUP19-TS.csv

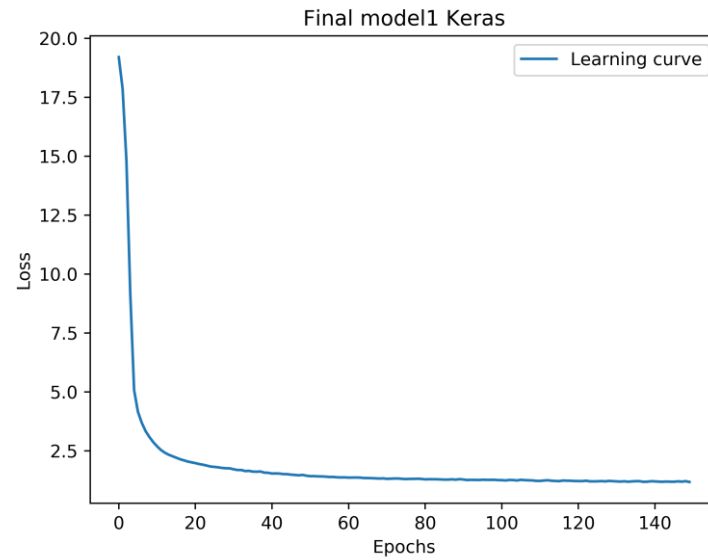
*We agree to the disclosure and publication of my name, and of the results with preliminary and final ranking.*

## REFERENCES

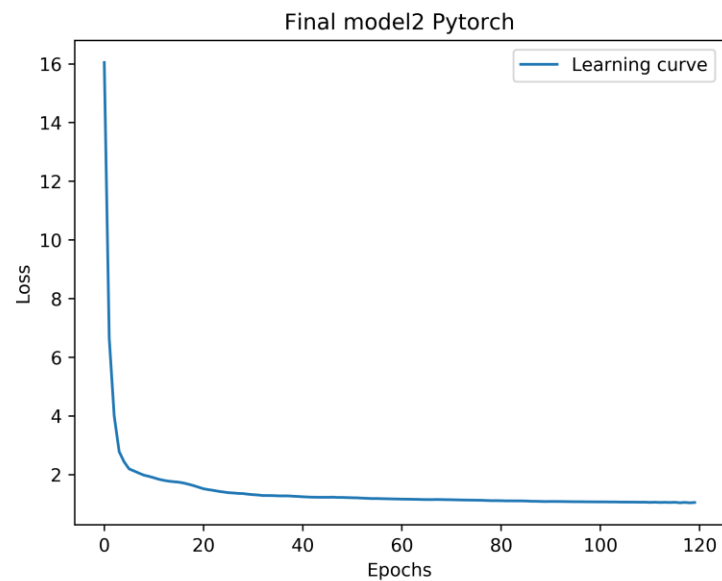
- [0] **BOOK PYTORCH ???**
  - [1] <https://keras.io/>
  - [2] <https://pytorch.org/>
  - [3] <https://scikit-learn.org/stable/>
  - [4] <https://github.com/aboffa/MachineLearningProject>
  - [5] <https://numpy.org/>
  - [6] <https://matplotlib.org/>
  - [7] [https://www.tensorflow.org/api\\_docs/python/tf/keras/backend](https://www.tensorflow.org/api_docs/python/tf/keras/backend)
  - [8] <https://www.csie.ntu.edu.tw/~cjlin/libsvm/>
  - [9] Xavier Glorot, Antoine Bordes and Yoshua Bengio (2011). Deep sparse rectifier neural networks. AISTATS. Rectifier and softplus activation functions. The second one is a smooth version of the first.
  - [10] <https://pytorch.org/docs/stable/index.html>
  - [11] <https://pytorch.org/docs/stable/notes/cuda.html>
  - [12] <https://scikit-learn.org/stable/modules/generated/sklearn.multioutput.MultiOutputRegressor.html>
-

---

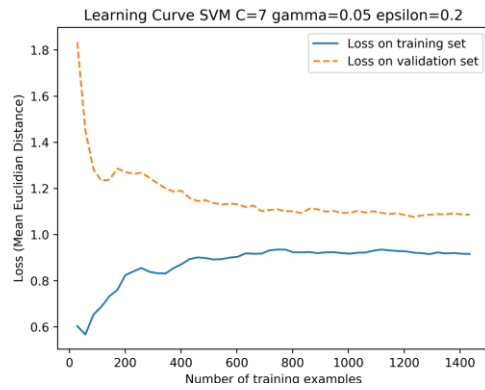
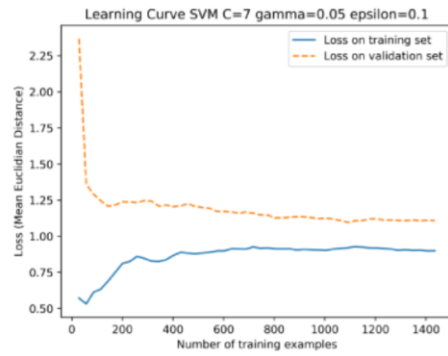
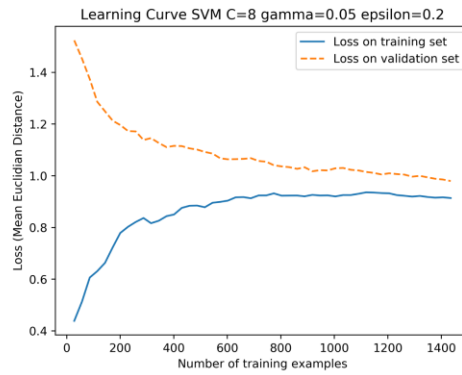
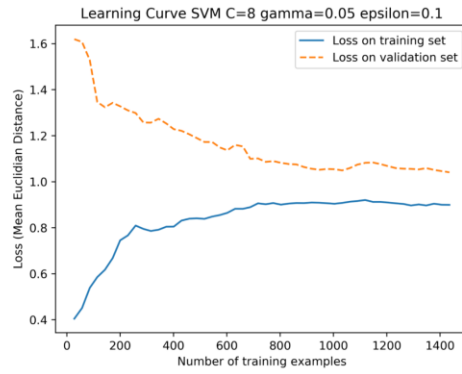
## APPENDIX A (LEARNING CURVES)



### A1. LEARNING CURVES OF THE FINAL MODEL 1



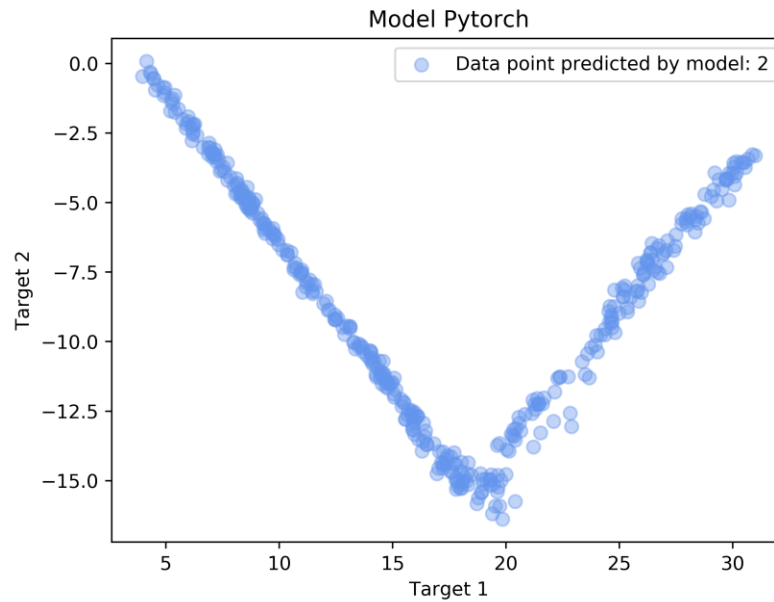
### A2. LEARNING CURVES OF THE FINAL MODEL 2



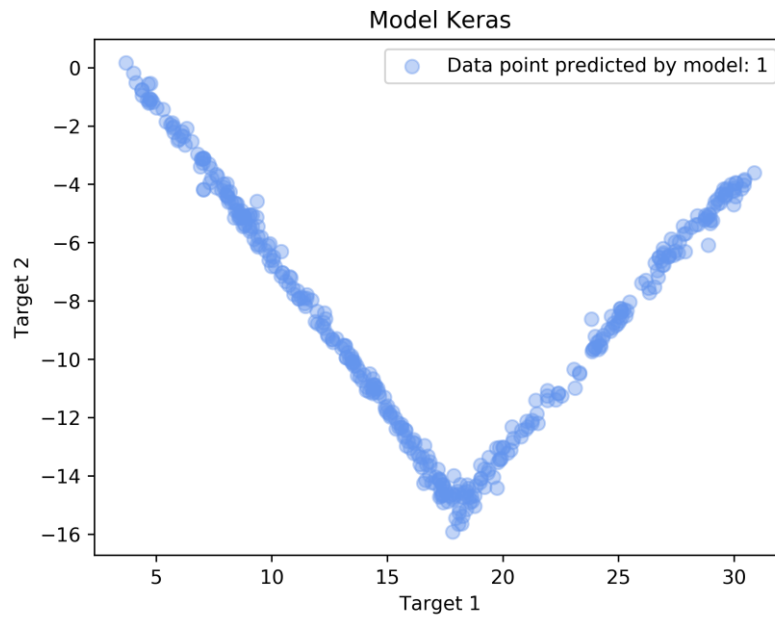
### A3. LEARNING CURVES OF THE MODEL 3

---

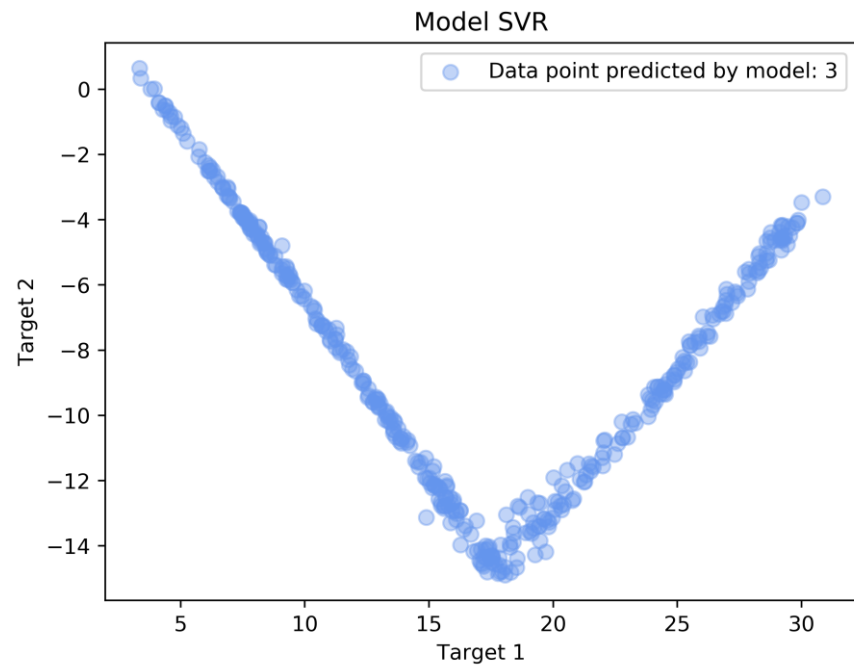
## APPENDIX B (DATA VISUALIZATION)



### B1. DATA VISUALIZATION OF THE FINAL MODEL 1



### B2. DATA VISUALIZATION OF THE FINAL MODEL 2



### **B3. DATA VISUALIZATION OF THE FINAL MODEL 3**