

# Machine Learning Project Report

Authors ( *Giulio Purgatorio, Jacopo Massa* ).

ICT Solutions Architect

Email( [g.purgatorio@studenti.unipi.it](mailto:g.purgatorio@studenti.unipi.it), [j.massa@studenti.unipi.it](mailto:j.massa@studenti.unipi.it) )

Exam: ML 654AA

Academic Year: 2020/21

Date: 25/01/2021

Type of project: **B**

## Abstract

We've compared two neural networks with a Support Vector Machine to see differences both on models and tools (on the same model). Finally, we made model selection and assessment by applying grid search and Cross-Validation before testing all models on an internal test set. The selected model for the CUP is made on **PyTorch** because it retrieved the best results.

## 1 Introduction

For neural networks, we used **Keras** [3] and **PyTorch** [6] implementing the same *Multi-Layer Perceptron* network and exploiting the different levels of abstraction provided by these two frameworks. The first one in fact had higher-level functionalities, while the second one allowed us to go more in-depth on some aspects.

To have a comparison with a different model we dealt with *Support Vector Machines*, specifically the one implemented in **scikit-learn** [8]. The task to solve was still the same, so a regression task with **10** input features and **2** output values. To tune hyperparameters, we've firstly done model selection with Grid Search and K-Fold Cross-Validation. For testing, we've exploited the Hold-out method, splitting the data to train and validate models.

We also built a small neural network to perform the classification tasks on the MONK's problem.

## 2 Method

All the developed code was written in `Python 3.7`, so we've used some of its common libraries:

- For plotting, we've used `matplotlib` [4].
- For data manipulation, we've used `numpy` [5].
- For 2 of the 3 models, we exploited the *GridSearchCV* provided by `scikit-learn`, also for computational efficiency.
- We used the *Mean Euclidean Error* (MEE) as **loss function** during the training and test phase, as requested.

As previously described in Section 1, we've made 3 models and we'll list our most important choices (consider that the two neural networks share the same structure and characteristics):

- Following the *Hold-out* method, we've split the dataset into two parts: some data for training the model and an *internal test set* used to choose the best model, after the validation phase. The former set has been further re-split to obtain *training* and *validation* sets. This will be further discussed in Section 3.
- As validation schema, we used *grid search* with *K-Fold Cross-Validation*, while for scoring we used the average of losses on all folds. Specifically, for `PyTorch` we've implemented it on our own, exploiting also the `multiprocessing` library, while for `Keras` and *SVM* we used the `scikit-learn` implementation.
- For the *Multi-Layer Perceptron*, we considered a **3** hidden-layers network, with **30** units per layer. All internal layers are *dense*, so each one of them is fully connected to the next one.
- As activation functions, we've used `tanh` for the regression task and `sigmoid` for the classification one (MONK).
- We've used *Stochastic Gradient Descent* as **learning algorithm** to include a bit of randomness and because it converges faster than batch training [2].
- So, we opted for *mini-batch learning*, searching for the `batch_size` during the model selection phase.
- We initialized weights using the *Glorot initialization* [1], which follows a Normal distribution centered in 0.
- Since we're dealing with a regression task, we've opted for SVR (Support Vector Regressor) which is a particular SVM with two slack variables for each data and an epsilon-insensitive loss.

### 3 Experiments

Before all else, we split the provided dataset into two subsets. The first (**1295** elements - 85%) has been used as development set during the model selection phase, while the latter (**229** elements - 15%) was the *internal test set*, used to evaluate the refitted models.

The next step has been the model selection, performed by using Grid Search and K-Fold Cross-Validation. The value for  $k$  has been fixed to **10**, which has been found through experimentation, but it's also a very common choice [7].

For each model, we'll show in Table 1 the range of explored hyperparameters during the exhaustive grid search performed to find the order of magnitudes for all the parameters. In the following sections, the top 5 combinations of parameters are shown: the highlighted green row (when present) has been used to perform a finer search to obtain hyperparameters' values used to refit and test each final model.

In fact, the last step was to split the development set into *training* and *validation* sets, use them to refit each model with the best hyperparameters, and test them using our internal test set.

Model	Parameter	Range
<b>Keras / PyTorch</b>	eta	[0.005, 0.05, 0.5]
	alpha	[0.2, 0.4, 0.6, 0.8]
	lmb	[0.0001, 0.001, 0.01]
	batch_size	[16, 32, 64]
<b>SVM</b>	kernel	[linear, poly, sigmoid, rbf]
	gamma	[0.0001, 0.001, 0.01, scale]
	C	[10, 100, 1000]
	epsilon	[0.1, 0.2 ... 0.9]
	degree	[2, 3, 4, 5]

**Table 1:** Hyperparameters' ranges for model selection.

- #Configuration for Keras/PyTorch model: **108**, so **1080** fits.
- #Configuration for SVM model: **432**, so **4320** fits (degree parameter used only for *polynomial* kernel function).
- **N.B.** *scale* value for **gamma** in SVM is one of the default values provided by scikit-learn, and is equal to:

$$\frac{1}{n\_features * variance(data)}$$

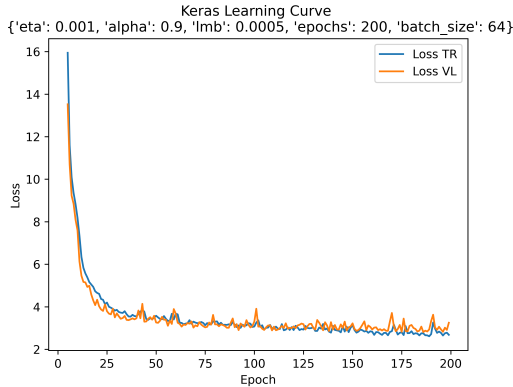
### 3.1 Keras

We quickly developed our MLP by using the high-level framework provided by Keras. Our first grid search stabilizes the values for **eta** and **lambda** around **0.005**, but, after some manual tuning, we've noticed that we've obtained better results by decreasing the **lambda**'s order of magnitude like in the last row of Table 2. Despite this, the internal test score for each refitted model was always around  $3.1 \sim 3.2$ .

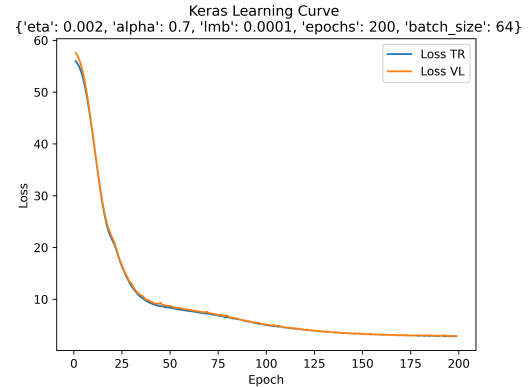
Eta	Alpha	Lambda	Batch size	Loss (TR)	Loss (VL)	Training Time
0.005	0.4	0.005	16	2.4072	2.7794	16
0.005	0.6	0.005	16	2.3593	2.7847	15
0.005	0.8	0.005	32	2.4013	2.7870	8.5
0.005	0.8	0.005	16	2.3571	2.8003	16.3
0.005	0.6	0.0005	64	2.4355	2.8131	5

**Table 2:** First model selection results for Keras (ordered by Loss on VL).

Since our Neural Network has a good number of layers and the **ReLU** function is well-known to work with deeper networks, we tried it but results were worse than the ones retrieved with **tanh**, especially on the training set. For example, we couldn't ever reach scores on TR, VL (and subsequently also on (I)TS) less than 3. Furthermore, the learning curve (Figure 1) isn't very smooth: in fact, it shows many spikes that are absent with the other function (Figure 2).



**Figure 1:** Learning curve with **ReLU** as the activation function.

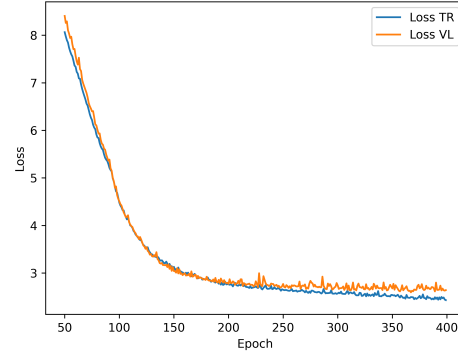
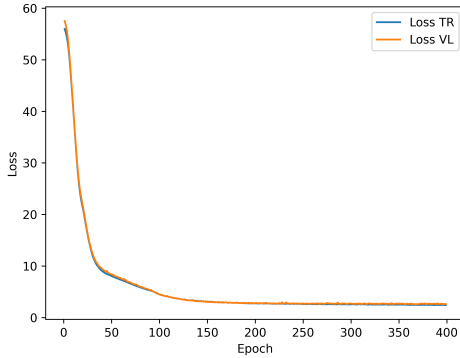


**Figure 2:** Learning curve with **tanh** as the activation function.

Following the suggested range on hyperparameters shown by the green row of Table 2, we performed a finer model selection and final values for the Keras model (whose learning curve is shown in Figure 2) are:

**eta** = 0.002, **alpha** = 0.7, **lambda** = 0.0001, **batch\_size** = 64.

We didn't run more than **200** epochs, because this led to overtraining and it didn't retrieve better loss scores anyway.



**Figure 3:** Plateau after the 200<sup>th</sup> epoch, **Figure 4:** Overtrained model (zoomed learning curve, starting from 50<sup>th</sup> epoch).

## 3.2 PyTorch

PyTorch is a lower-level framework when compared to the previous one. It allows getting more in-depth with some aspects of the architecture and network's behavior. We had to implement:

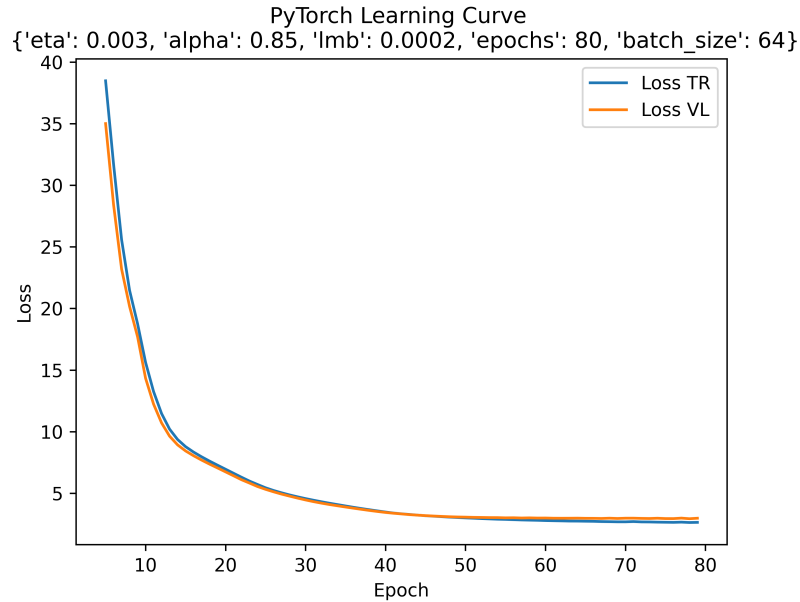
- the *loss function*, because PyTorch works with tensors and not with Python primitive types.
- the *forward function*, which is the one used to compute each layer's output.
- the *training step* following the mini-batch algorithm, so splitting the data according to the `batch_size` parameter and running as many steps as the defined number of epochs.
- the *k-fold cross-validation schema*, splitting data in folds, collecting and averaging respective scores to be coherent with the other validation schemas.

As previously stated, PyTorch works with tensors and it provides optimizations to work with them. Specifically, these optimizations are grouped in the *CUDA* semantics, which computes tensors' operations through the GPU rather than the CPU: we couldn't exploit this advantage, because our laptops hadn't a compatible GPU. As a confirmation of this, time performances for PyTorch's grid search and refitting were worse than the Keras' ones, as shown in Section 3.5.

Eta	Alpha	Lambda	Batch size	Loss (TR)	Loss (VL)	Training Time
0.005	0.8	0.005	64	2.3148	2.6415	14.86
0.005	0.4	0.005	64	2.5875	2.6915	14.82
0.005	0.6	0.0005	32	2.1869	2.7110	23.15
0.005	0.6	0.0005	64	2.3774	2.7380	14.8
0.005	0.4	0.0005	32	2.3337	2.7387	23.07

**Table 3:** First model selection results for PyTorch.

We performed model selection and, as expected by the fact that we had the same neural network, we’ve obtained similar results to the ones from **Keras**’ model selection. We’ve directly used the **tanh** activation function for the same reason explained above (Section 3.1). During the hyperparameter tuning phase, we discovered some relevant differences from the previous model: learning curves were smoother and we’ve obtained the same (or better) results with a fewer number of epochs (only **80!**), as shown in Figure 5.



**Figure 5:** Learning curve (PyTorch).

Final values for the PyTorch model are:  
**eta** = 0.003, **alpha** = 0.85, **lambda** = 0.0002, **batch\_size** = 64.

### 3.3 scikit-learn SVM

The Support Vector Machine has a different approach than neural networks, in fact we had to deal with kernels and margin manipulation. For our purpose (a multi-output regression task) we used a *Support Vector Regressor* (SVR) in combination with a *Multi-Output Regressor* (MOR). The former has the peculiarity of constructing an epsilon-tube within which no penalty is associated in the training loss function with points predicted within a distance epsilon from the actual value. Since SVR doesn't support multi-target regression, we used MOR which fits one regressor per target (so, in our case, **2** regressors). Since we didn't have epochs in SVM, we built learning curves by plotting train (and validation/test) scores for different training set sizes, using again as a metric the MEE.

After having chosen the appropriate tools, we moved towards the kernel (and its parameters) choice and after some experiments, confirmed also by our model selection results in Table 4, we found that the best solution was the **rbf** kernel. For the sake of completeness, outcomes for the other types of kernels can be seen in Table 5.

Kernel	C	Gamma	Epsilon	Loss (TR)	Loss (VL)	Training Time
<b>rbf</b>	5	scale	0.7	2.6929	2.9926	0.1684
rbf	5	scale	0.5	2.6741	2.9931	0.1770
rbf	5	scale	0.3	2.6559	3.0016	0.1889
rbf	5	scale	0.1	2.6435	3.0099	0.2
rbf	50	scale	0.7	2.2359	3.0148	0.3958

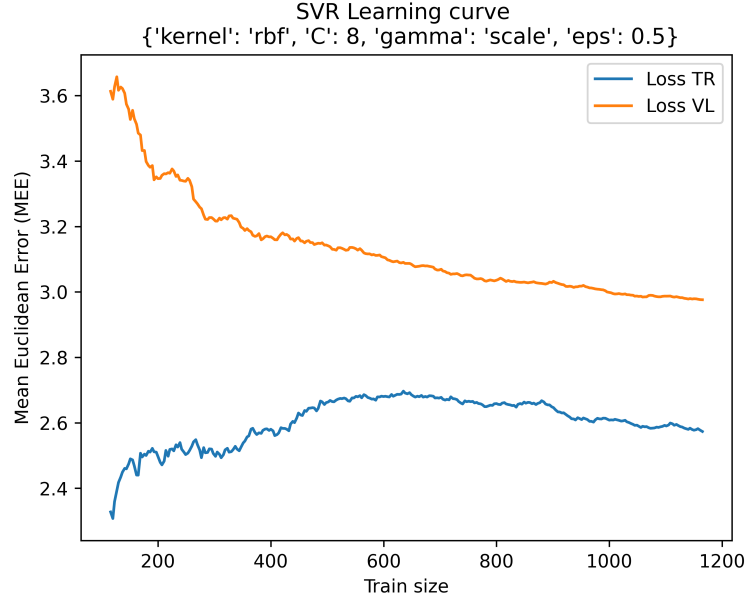
**Table 4:** First model selection results for SVR.

We didn't perform a second model selection because more or less all parameters were already defined by the first run, and so we manually tuned **epsilon** and **C** parameters defining the final values for the SVM model as:

**kernel** = rbf, **C** = 8, **epsilon** = 0.6, **gamma** = scale. These values led to the learning curve shown in Figure 6.

Kernel	Parameters	Loss (TR)	Loss (VL)
linear	C = 10, epsilon = 0.2, gamma = scale	5.14	5.31
poly	C = 10, epsilon = 0.1, gamma = scale, degree = 3	5.71	6.33
sigmoid	C = 8, epsilon = 0.4, gamma = scale	64.65	63.82

**Table 5:** Results for other kernel functions.



**Figure 6:** Learning curve (SVM).

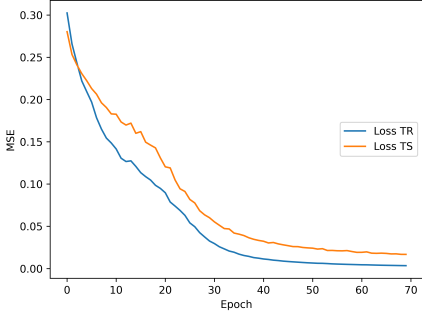
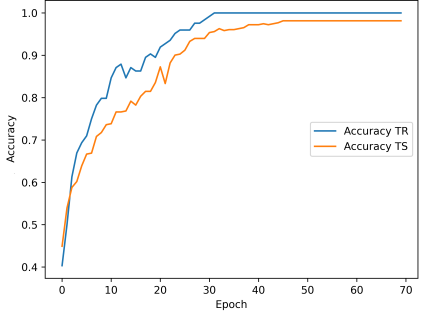
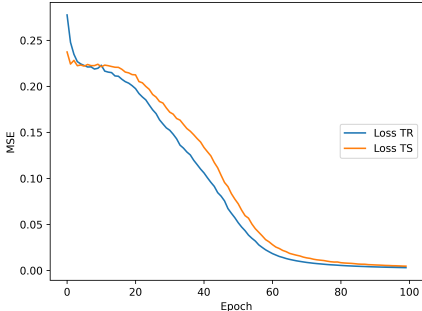
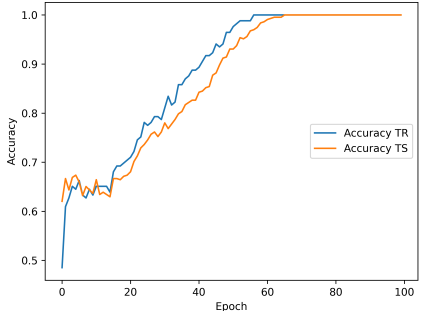
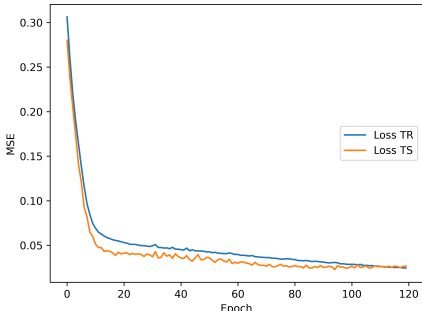
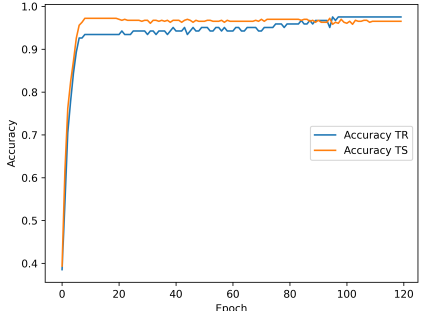
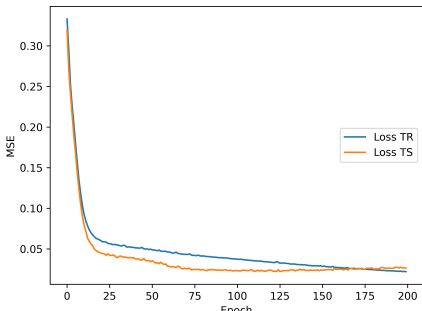
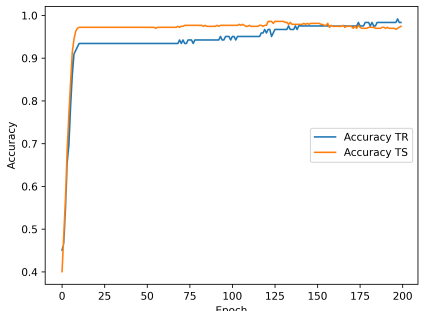
### 3.4 Monk Results

We implemented a `Keras` MLP with one hidden-layer of 4 units. The batch size for each MONK task was **25**. For each problem, we used different parameters (visible in Table 6), which led to results shown in Table 7.

Task	Eta	Alpha	Lambda	Epochs	MSE (TR/TS)	Accuracy (TR/TS)
MONK1	0.22	0.85	-	70	0.0034/0.0058	100%/100%
MONK2	0.21	0.77	-	100	0.0024/0.0030	100%/100%
MONK3	0.2	0.76	-	120	0.0247/0.0269	98%/97%
MONK3+reg.	0.3	0.5	0.0001	200	0.02/0.0262	99%/97%

**Table 6:** Average prediction results obtained for the MONK’s tasks.



	MSE	Accuracy
MONK 1	 <p>MSE plot for MONK 1. The x-axis is 'Epoch' (0 to 70) and the y-axis is 'MSE' (0.00 to 0.30). Two lines are shown: 'Loss TR' (blue) and 'Loss TS' (orange). Both lines decrease over time, with 'Loss TR' reaching a lower final value than 'Loss TS'.</p>	 <p>Accuracy plot for MONK 1. The x-axis is 'Epoch' (0 to 70) and the y-axis is 'Accuracy' (0.4 to 1.0). Two lines are shown: 'Accuracy TR' (blue) and 'Accuracy TS' (orange). Both lines increase over time, with 'Accuracy TR' reaching a higher final value than 'Accuracy TS'.</p>
MONK 2	 <p>MSE plot for MONK 2. The x-axis is 'Epoch' (0 to 100) and the y-axis is 'MSE' (0.00 to 0.25). Two lines are shown: 'Loss TR' (blue) and 'Loss TS' (orange). Both lines decrease over time, with 'Loss TR' reaching a lower final value than 'Loss TS'.</p>	 <p>Accuracy plot for MONK 2. The x-axis is 'Epoch' (0 to 100) and the y-axis is 'Accuracy' (0.5 to 1.0). Two lines are shown: 'Accuracy TR' (blue) and 'Accuracy TS' (orange). Both lines increase over time, with 'Accuracy TR' reaching a higher final value than 'Accuracy TS'.</p>
MONK 3	 <p>MSE plot for MONK 3. The x-axis is 'Epoch' (0 to 120) and the y-axis is 'MSE' (0.00 to 0.30). Two lines are shown: 'Loss TR' (blue) and 'Loss TS' (orange). Both lines decrease over time, with 'Loss TR' reaching a lower final value than 'Loss TS'.</p>	 <p>Accuracy plot for MONK 3. The x-axis is 'Epoch' (0 to 120) and the y-axis is 'Accuracy' (0.4 to 1.0). Two lines are shown: 'Accuracy TR' (blue) and 'Accuracy TS' (orange). Both lines increase over time, with 'Accuracy TR' reaching a higher final value than 'Accuracy TS'.</p>
MONK3+reg.	 <p>MSE plot for MONK3+reg. The x-axis is 'Epoch' (0 to 200) and the y-axis is 'MSE' (0.00 to 0.30). Two lines are shown: 'Loss TR' (blue) and 'Loss TS' (orange). Both lines decrease over time, with 'Loss TR' reaching a lower final value than 'Loss TS'.</p>	 <p>Accuracy plot for MONK3+reg. The x-axis is 'Epoch' (0 to 200) and the y-axis is 'Accuracy' (0.4 to 1.0). Two lines are shown: 'Accuracy TR' (blue) and 'Accuracy TS' (orange). Both lines increase over time, with 'Accuracy TR' reaching a higher final value than 'Accuracy TS'.</p>

**Table 7:** Plots of the MSE and accuracy for the 3 MONK's benchmarks.

### 3.5 Cup Results

At the end of the model selection phase, we compared the results from our three models that are summarized in Table 8. We'd like to recall that all scores were obtained after refitting the model on the entire training set.

	Training	Validation	Test	Grid Search Time
Keras	2.7636	2.8463	3.2206	16 mins
PyTorch	2.6297	2.6873	3.0165	1 hour
SVM	2.5869	2.9610	3.2329	0.5 mins

**Table 8:** Final results comparison.

Given these results, we've chosen the PyTorch model to perform prediction on the provided test set. Learning curves for each model were already shown in respective Sections.

## 4 Conclusion

**Nickname:** MARIO

**Blind test:** MARIO\_ML-CUP20-TS.csv

As we could expect, choosing a lower-level framework implies the usual trade-off: more accurate models at the cost of slower development. Of course, being able to inspect deeper into what happens helps both at debugging and understanding, due to the low abstraction. In any case, results are similar because the Keras framework version we have used in this project is built on PyTorch itself, making it easier by abstracting it: that's its strength and also why it's a very common framework nowadays.

Finally, about the SVM, we've shown that it's a very good model and its efficiency is further proved by the time required for its training and fitting phase: the fastest out of all of them, while still being very close to other scores.

## Acknowledgments

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

## References

- [1] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feed-forward neural networks. In Yee Whye Teh and Mike Titterton, 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.
- [2] Priya Goyal, Piotr Dollár, Ross B. Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. Accurate, large minibatch SGD: training imagenet in 1 hour. *CoRR*, abs/1706.02677, 2017.
- [3] Keras Site. <https://keras.io/>.
- [4] Matplotlib Site. <https://matplotlib.org/>.
- [5] NumPy Site. <https://numpy.org/>.
- [6] PyTorch Site. <https://pytorch.org/>.
- [7] Payam Refaeilzadeh, Lei Tang, and Huan Liu. *Cross-Validation*, pages 532–538. Springer US, Boston, MA, 2009.
- [8] scikit-learn Site. <https://scikit-learn.org/>.