Machine Learning Project Report

Gennaro Daniele Acciaro, Annachiara Aiello, Alessandro Bucci
g.acciaro@studenti.unipi.it, a.aiello16@studenti.unipi.it, a.bucci12@unipi.it
Master Degree (Artificial Intelligence curriculum)
ML course 654AA, Academic Year: 2021/2022
Date: 04/01/2022
Type of project: **A**

Abstract

This report describes the Python implementation of an Artificial Neural Network, capable to deal with both classification and regression tasks, and shows its results on Monk and ML-CUP21 problems. The final model for the CUP problem was selected through a grid search over hundreds of hyper-parametric configurations, using the K-Fold cross-validation strategy, and then tested on an internal test set with Hold Out.

1 Introduction

The aim of our ML project was to develop an Artificial Neural Network in order to solve the supervised regression problem on the ML-CUP21 dataset. In particular, we implemented a feed-forward fully connected neural network (Multi- Layer Perceptron), with Back-propagation training algorithm [2], using Python programming language. We first tested our network on the Monk classification problem [4], achieving good results in terms of error and accuracy on the external Test sets. Then we adapted our model to the CUP regression learning task, now estimating the error on an Internal Test Set. In both cases, we performed model selection with K-Fold cross-validation and model assessment with Hold Out Test. In particular, for CUP problem we used a distributed computation across several computers, using a centralized database, in order to reduce K-Fold computational time.
In this project we had the opportunity to implement the best known convergence improving and regularization techniques of ML theory, and to observe the differences of their effects on the learning curves. All the details about implementation choices and results are shown in the sections below.

2 Method

This section describes which are the core implementation of the neural network, the functions and tools, the pre-processing, the initialization methods, the regularization and convergence speed improvements and finally the validation schema we used.

2.1 Core implementation

One of the most interesting programming challenge of *Type A project* was implementing a neural network in an efficient way: to achieve this, we made extensive use of matrices. See for

example the simple neural network in Figure 1, with three nodes in the input layer L_0, two in the hidden layer L_1 and one in the output layer L_2.

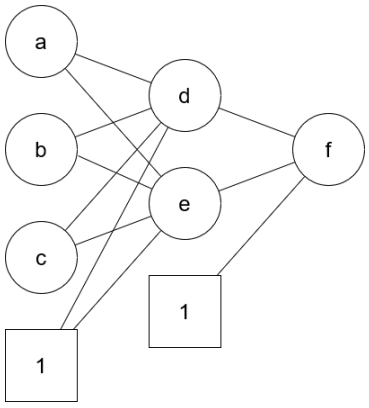

Figure 1: Example of a simple neural network.

The matrices representing the weights between the layers have dimension $n \cdot m$, where n is the number of nodes of the next layer and m the number of nodes of the current layer, following the same notation we saw during lectures while referring to the weights. Matrices for the NN in Figure 1 are represented in Tables 1 and 2.

Weights between L_0 and L_1				
	a	b	c	bias
d	w_{da}	w_{db}	w_{dc}	w_{d1}
e	w_{ea}	w_{eb}	w_{ec}	w_{e1}

Table 1: Weights matrix between layer 0 and layer 1.

Weights between L_1 and L_2			
	d	e	bias
f	w_{fd}	w_{fe}	w_{f1}

Table 2: Weights matrix between layer 1 and layer 2.

2.2 Functions

We implemented some *Activation* functions such as *Identity*, *Sigmoid*, *ReLU*, *LeakyReLU* and *Tanh*. In order to achieve better maintainability of the code, we implemented all these functions using a factory design pattern. The same idea was applied to the *Loss* functions and the *Learning Rate Decay* functions. This easily allowed us to implement *Back-propagation algorithm* so that using other Loss functions instead of standard *Mean Square Error* is possible.
It's eventually worth noting that we dealt the problem of non-derivability in zero of *ReLU* by considering $f'(0) = 0$.

2.3 Tools

The project was developed using **Python** language (**v3.9**) and a few libraries such as **Numpy** (used for scientific computing), **Sklearn** (used for 1-Hot Encoding) , **Pandas** (used for read

datasets) and **Math** (used for randomness).

2.4 Pre-pocessing

We didn't execute any pre-processing process on the Cup dataset, while we applied the 1-Hot Encoding on the Monk datasets, since its features are categorical, and then we obtained 17 binary features.

2.5 Initialization methods

For initialization methods, at the beginning we implemented two functions in the class `Utility`: `UniformRandomInitialization` and `GlorotInitialization`. As the name suggests, the first one initializes the NN weights according to a *random uniform distribution* with values chosen between -0.7 and 0.7. This could work well for very simple NNs (with a single hidden layer), but for deeper ones we need a more sophisticated approach. In Glorot Initialization [1], every weight is set with a small Gaussian value with mean equals to 0 and variance based on the fan-in and fan-out of the weight. Remember that if we have a weight connecting a node u in the layer L_u to a node t in the layer L_t, the fan-in is the number of units in L_u, analogously the fan-out is the number of units in L_t. We decided to use the second method also for Monk task, to avoid the network to be too much sensible to weights initialization.

2.6 Regularization and Convergence speed improvements

As we know from *Statistical Learning Theory* [6], finding the best trade-off between underfitting and overfitting is the main issue of Machine Learning. For this reason, we implemented **L2 Regularization** to control the complexity of the model, using λ as hyper-parameter. We also decided to exploit some **Early Stopping criteria**: we'll stop the training if the error on the Validation Set doesn't significantly decrease anymore or conversely if it begins to increase (details are referred to subsection 3.2). As for convergence speed improvement techniques, we implemented both the **Heavy Ball Momentum** and **Nesterov's Accelerated Gradient** [3], even if we didn't achieve better results on CUP task with the second one. Moreover we implemented **Variable Learning Rate**, that guarantees the gradient decrease to zero close to a minimum in the mini-batch approach; following this logic, we used linear and exponential decays of the learning rate.

2.7 Validation schema

For both classification and regression tasks, we performed a K-Fold cross-validation for model selection and an Hold Out Test for model assessment.
The choice of the K-Fold CV was mainly due to the insufficient availability of data in Monk datasets to make three significant size splits (TR/VL/TS), but also to avoid to be sensible to the specific partitioning of examples. For the same reason, for Monk problem we took $K = 6$ in order to have larger training subsets, while for the CUP we could take $K = 4$.
In any case we used the following hyper-parameters for the grid searches:

- `lr` (η): initial learning rate of the *Update rule*;

- `alpha_momentum` (α): momentum parameter;
- `L2` (λ): regularization parameter of Ridge regression (L2);
- `batch_size`: size of minibatches (if equals to 1 we have online version, if `"full"` we have batch version);
- `lr_decay_type`: type of decay of the learning rate; we implemented both linear and exponential decay. In the first method, after 250 epochs the learning rate is fixed at 1% of the initial value. In any case we decided to update the learning rate every epoch.
- `topologies`: we proceeded in different ways for the two tasks. For Monk problem, we only had few possible network topologies to choose from (details in section 3.1). For CUP problem, we used two parameters: the number of hidden layers (`hidden_layers`) and the number of units per layer (`hidden_units`). For computational cost reasons we assumed every layer to have the same number of nodes. This choice allowed us to generate much more combinations in order to create more complex NNs.

It goes without saying that for the CUP problem we have much more options, in particular for the architecture parameters, since a more complex model is required. More details will be shown in section 3.2.

3 Experiments

In this section we report and discuss in details our numerical and graphical results for Monk and CUP tasks.

3.1 Monk Results

Due to the simplicity of the task, we assumed a single hidden layer for all four Monk problems, with *Sigmoid* activation function for both the hidden and output layer. For each Monk dataset we transformed inputs with *1-Hot encoding*: in this way we have 17 binary input patterns, hence each topology starts with an input layer of 17 units.
The loss in the *Backpropagation algorithm* was computed using the *Mean Squared Error*; at this point the same metric was also used for computing the error on TR and TS sets.
Furthermore we now disabled Early Stopping criteria, in order to obtain meaningful plots.
Table 3 presents the explored values for the Monk grid searches, while Table 4 shows our results for each Monk dataset.

Hyper-parameter	**values**
`lr`	0.9, 0.5, 0.1, 0.01, 0.001
`alpha_momentum`	0.5, 0.75, 0.9
`L2`	0
`batch_size`	1, 62, "full"
`lr_decay_type`	"linear", "exponential"
`topologies`	[17, 4, 1], [17, 8, 1], [17, 12, 1]

Table 3: Range of explored hyper-parameters for the Monk grid searches.

Task	**Topology**	η	α	λ	**Batch size**	**MSE (TR/TS)**	**Accuracy (TR/TS)(%)**
Monk1.	[17,8,1]	0.5	0.9	0	1	1.43901e-05/1.15772e-4	100% / 100%
Monk2	[17,4,1]	0.9	0.8	0	1	2.6.02761e-06/9.27074e-06	100% / 100%
Monk3	[17,8,1]	0.1	0.5	0	1	3.73931e-02/4.85213e-02	96.721%/93.981%
Monk3(r)	[17,8,1]	0.1	0.5	1-e05	1	4.35073e-02/4.61554e-02	95.082%/95.139%

Table 4: Average prediction results obtained for the Monk tasks.

Figure 2 and Figure 3 show the accuracies and the learning curves for Monk1 and Monk2, here we can clearly see how the accuracy jumps right to 100% almost instantly, the curves are nice and smooth and are plotted in logarithmic scale to better see how the loss still decreases even after many epochs. Monk3 indicates us a case of overfitting, due to the noise within the dataset. An evidence of this problem is shown in Figure 4. In order to fix this problem, we used the *Tikhonov Regularization* (L2), with $\lambda = 0.00001$. We can see in Figure 5 that, even if we still have overfitting, since we have temporarily disabled the Early Stopping criteria, it begins after much more epochs. Indeed, using the Tikhonov Regularization the test accuracy improved slightly at the expense of accuracy on the training set.

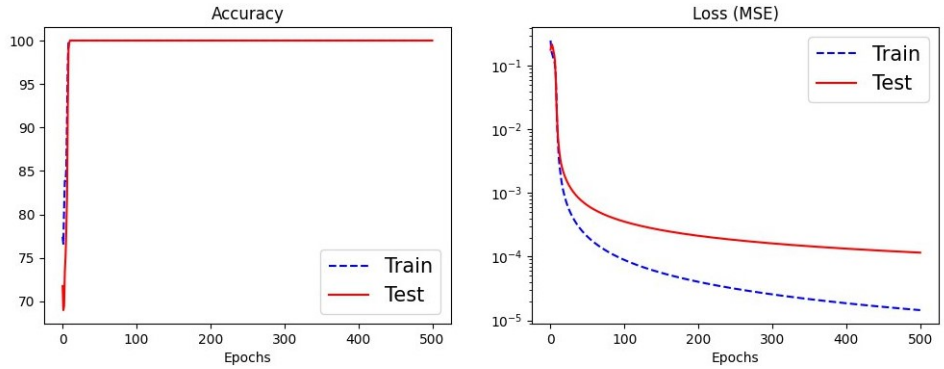

Figure 2: Monk 1 results

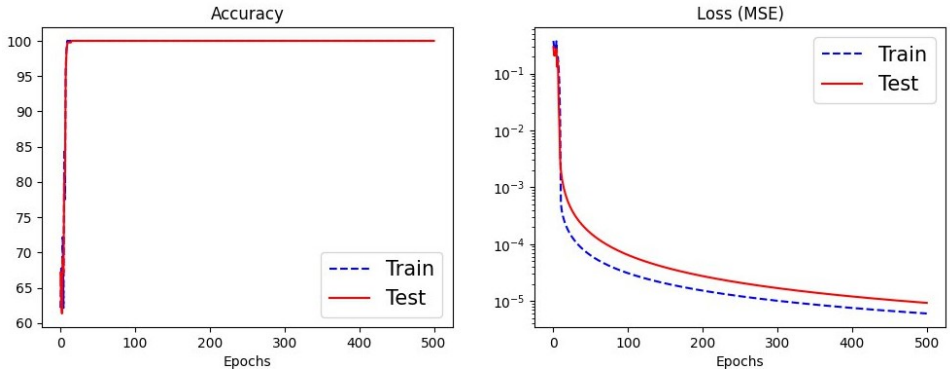

Figure 3: Monk 2 results

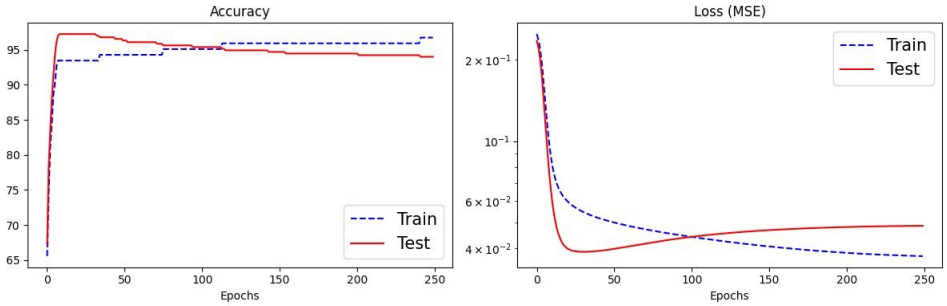

Figure 4: Monk 3 results

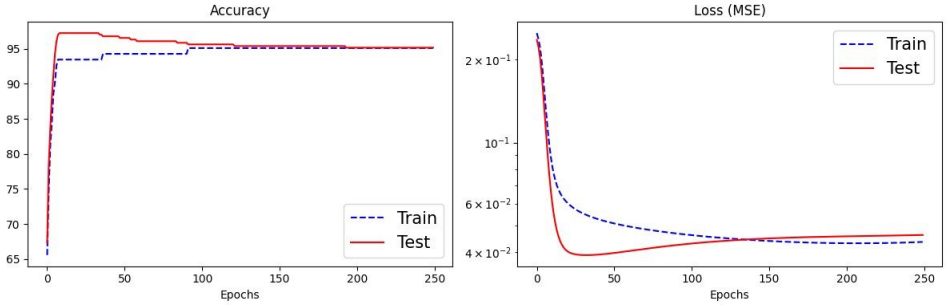

Figure 5: Monk 3 (with L2 reg) results

3.2 CUP Results

First of all, we splitted ML-CUP21-TR dataset into two subsets: 70% of data were used for the **Development/Design Set** and the remaining 30% for the **Internal Test Set**.
So we proceeded as follows: we implemented a 4-Fold cross-validation on Development Set for model selection, as already mentioned in section 2.7. At this phase, *Mean Euclidean Error* was used to calculate the mean error over the **Validation** sets obtained from the all folds (each Validation Set is a quarter of the Design Set) and to rank the models. We performed every fold in parallel, in order to exploit all the resources of the computer running them.
As for the activation functions, for the hidden layers we used the $ReLU$ that allows us to avoid the problem of vanishing gradient and for the output layer we chose the *Identity* function, because the CUP is a regression task.
Some constants manage the Early Stopping criteria:

- `max_non_decrescent_epochs` is the maximum number of epochs that can elapse after the model have encountered a minimum on the VL set while not encountering a smaller one; the default value is 15.
- `threshold_variance` is the lower bound for the variance of the last 20 epochs' error (it's set at 10^{-6});
- `max_epochs` is the upper bound for the number of epochs (it's set at 350 in the process of training and at 250 in the process of re-training of the final model).

For model assessment we performed an Hold Out Test, using *MEE* again, now to evaluate the generalization error on the Internal Test Set. Here the list of hyper-parameters for the preliminary coarse grid search and their values in the race (Table 5).

Hyper-parameter	**values**
`lr`	0.01, 0.001, 0.0001, 0.00001
`alpha_momentum`	0.5, 0.75, 0.9
`L2`	0.01, 0.001, 0.0001, 0.00001
`batch_size`	1, 32, 64, "full"
`lr_decay_type`	"linear"
`hidden_layers`	1, 2, 3
`hidden_units`	10, 15, 20, 25

Table 5: Range of explored hyper-parameters for the CUP coarse grid search.

In order to try such a huge number of possible combinations, we performed the grid searches in parallel on several computers, after having developed a very simple distributed calculus exploiting a common database.
Table 6 shows the results of the three best performing cases of this preliminary search.

Model	**Topology**	η	α	λ	**Batch size**	**MEE(TR)**	**MEE(VL)**	**MEE(TS)**
Model 1	$[10, 25, 25, 25, 2]$	0.01	0.5	0.00001	32	0.9242027374	1.175097728	1.20031036
Model 2	$[10, 10, 10, 2]$	0.01	0.9	0.00001	64	1.024509813	1.180088274	1.191176871
Model 3	$[10, 25, 25, 2]$	0.01	0.75	0.00001	64	1.038551823	1.184066887	1.198293098

Table 6: The three best performing models obtained from the CUP coarse grid search. Note that with TS we mean the Internal Test Set.

We then ran a finer grid search with values in a small ($\pm 10\%$) and discrete range of the winner ones, for some hyper-parameters such as, η, α and λ, while for `batch_size` we tried adding ± 10 to its previous value of the coarse.
New hyper-parametric values are shown in Table 7, while Table 8 shows the best three models again.

Hyper-parameter	**values**
`lr`	0.009, 0.01, 0.011
`alpha_momentum`	0.45, 0.5, 0.55
`L2`	0.000009, 0.00001, 0.000011
`batch_size`	22, 32, 42
`lr_decay_type`	"linear"
`Topology`	[10, 25, 25, 25, 2]

Table 7: Range of explored hyper-parameters for the CUP fine grid search, *Model 1*. Note that now the topology is fixed.

Model	**Topology**	η	α	λ	**Batch size**	**MEE(TR)**	**MEE(VL)**	**MEE(TS)**
Model 1F	$[10, 25, 25, 25, 2]$	0.009	0.55	0.000011	42	0.9373514104	1.138469941	1.17199421
Model 2F	$[10, 25, 25, 25, 2]$	0.009	0.55	0.000011	32	0.9693845614	1.148155783	1.185481874
Model 3F	$[10, 25, 25, 25, 2]$	0.009	0.55	0.000011	32	0.9665867645	1.148856579	1.18803047

Table 8: The three best performing models obtained from the CUP fine grid search.

Once selected the best model (*Model 1F*), we performed a *Random Grid Search* to have better weights initialization. So we re-trained *Model 1F* **ten times** on the 85% of the Design Set, looking for the seed that minimized the *MEE* on the remaining 15% of Development Set, used for Validation. Table 9 shows its hyper-parametric values and its results.

Topology	η	α	λ	**Batch size**	**MEE(TR)**	**MEE(VL)**	**MEE(TS)**
$[10, 25, 25, 25, 2]$	0.009	0.55	0.000011	42	1.006540340	1.14152178	1.152053826

Table 9: Final Model results on CUP task.

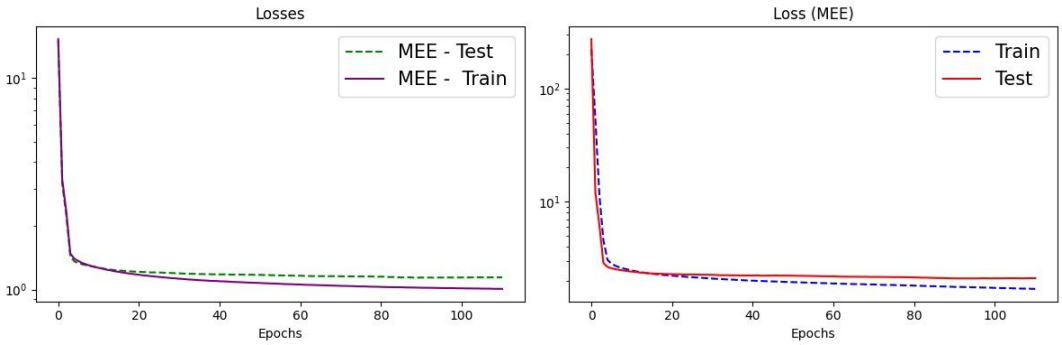

Figure 6: Final Model results on CUP task.

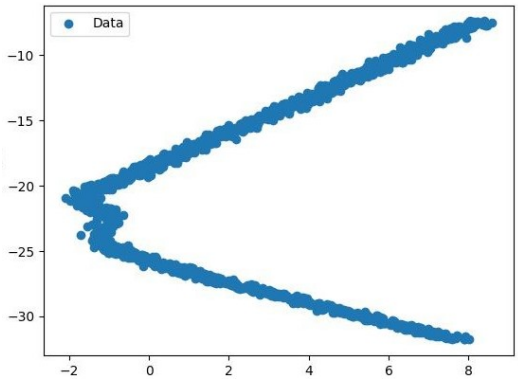

Figure 7: Plot of ML-CUP21-TR targets.

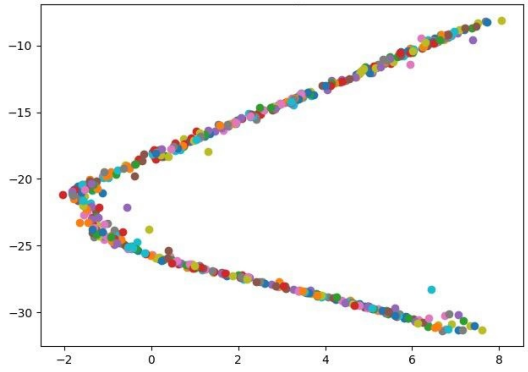

Figure 8: Plot of Final Model outputs on the Blind Test Set.

Figure 7 shows the plot of ML-CUP21-TR targets, while in Figure 8 we can see our Final Model outputs plot on the Blind Test Set. We can easily observe how similarly the points are distributed, although they are calculated on different input datasets (known data vs unknown data). This is a foreseeable result, due to the *Empirical Risk Minimization Inductive Principle* [5]. Note that in Figure 8 some outliers come up: we think it's a good sign, since a perfect overlapping would represent an overfitting evidence.

4 Conclusion

In this project we explored how to implement a Neural Network from scratch. Our results on Blind Test Set were stored into the **AIAIAI_ML-CUP21-TS.csv** file within the project; our nickname is AIAIAI. This work made us strongly aware of the importance of the choices of the model hyper-parameters, regardless of the correctness of the NN structure implementation. Furthermore, we had the opportunity to put into practice a complex and fascinating mathematical theory, which has so many applications.

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 feedforward neural networks. *DIRO, Université de Montréal, Montréal, Québec, Canada.*

[2] Alessio Micheli. Derivation of the back-propagation based learning algorithm. 2021.

[3] Tom M. Mitchell. *Machine Learning.* McGraw-Hill, New York, 1997.

[4] S. Thrun, Jerzy Bala, Eric Bloedorn, Ivan Bratko, J. Cheng, Kenneth De Jong, Sašo Džeroski, Scott Fahlman, Doug Fisher, R. Hamann, Kenneth Kaufman, S. Keller, I. Kononenko, J. Kreuziger, T. Mitchell, P. Pachowicz, Yoram Reich, Haleh Vafaie, and J. Wnek. The monk's problems a performance comparison of different learning algorithms. 01 1992.

[5] Vladimir Vapnik. Principles of risk minimization for learning theory. In *Advances in neural information processing systems*, pages 831–838, 1992.

[6] Vladimir Vapnik. *The nature of statistical learning theory.* Springer science & business media, 1999.