Machine Learning Project Report

Diletta Goglia - 530924 - `d.goglia@studenti.unipi.it` - M.Sc. in Artificial Intelligence
Paolo Murgia - 531108 - `p.murgia1@studenti.unipi.it` - M.Sc. in Artificial Intelligence

ML course (654AA)
Academic Year: 2020/2021
Date: 13/10/2021
Project Type **A**

Abstract

The project consists in the implementation of an Artificial Neural Network built from scratch using Python, without using pre-built libraries. The overall validation schema consists in a preliminary screening phase to reduce the hyperparameters search space, followed by a first coarse grid-search and a second but finer one. All the explored models are validated with a 5-fold cross validation. The resulting model is a 2 hidden layer network with 20 units each and ReLU activation for both layers.

1 Introduction

The project main aims are to develop a robust Artificial Neural Network capable to solve both classification and regression problems from scratch and define a rigorous method to select the best models and validate their results. We implemented a fully-connected feed-forward multi-layer perceptron network that learns through backpropagation and supports stochastic gradient descent (SGD) algorithm as optimizer, that can possibly be extended with the Nesterov accelerated gradient momentum.
In order to figure out the best configuration of values for hyperparameters, and so to find the best model in terms of complexity, flexibility and generalization capability, we adopted random search and grid-search techniques on the hyperparameters with 5-fold cross-validation. Our models have been tested on an internal validation set. The proper functioning and correctness of the implementation were firstly tested on the MONK dataset. Then, after this first benchmark, the models were used to predict the outputs of the CUP regression problem. We paid particular attention in performing a preliminary screening-phase for both tasks, and only after we switched to model selection phase.

2 Method

We started by defining the main framework of our architecture and then we gradually went deeper in refining it more and more in detail. We proceeded step-by-step in adding (and meanwhile testing) features and improvements to our structure: the result is an in-depth and modular software perfectly simulating an ANN.

2.1 Implementation

In this section we describe the structure and design of our code, listing all the classes built as well as the implementation choices we made.

- **Tools and libraries.** The software was developed using Python 3 and some ancillary libraries and modules, such as: `numpy, pandas, math, matplotlib` and some utilities of `sklearn`. We also exploited `joblib.Parallel` module for a more efficient grid search.

- **Code implementation.** For clarity, transparency and accessibility purposes, we decided to write our code following the "tacit and explicit conventions applied in Scikit-learn and its API", and so to follow the notation of the glossary[3], eg. using standard terms for methods, attributes, etc. This well-known "best practice" allowed us to write a good-quality code, well-commented and easy for reading, understanding and experiments reproducibility.
- **Preprocessing.** No pre-processing has been done on the ML-CUP20 datasets. Otherwise for all the MONK datasets we performed a One-Hot encoding in order to transform the input and create labels for classification task.
- **Additions, novelties and variations.** Since grid search is a fundamental and necessary step, but often high costly and time consuming, we opted for two smart approaches in order to optimize as much as possible this phase, without compromising the final result.
  - We aim at exaustive screening phases as a way of **preliminar pruning of the hyperparameter search space**. In this way we are able to know in advance those ranges that are unlikely to contain better hyperparameter configurations, and so not to consider them for grid search.
  - We propose **a filter to preliminary discard poor models from the hypotheses space** for accelerating the search of the best model, without compromising the final performance. More details about this strategy are described in subsection 2.4.
- **Overview of software structure.** The architecture of our network has been implemented with the aim of being as modular as possible, following the OOP paradigm. Specifically, the model we used for this project is a simple MLP: however, being the implementation modular, it is possible to add an arbitrary number of layers and units in order to build a more complex network. We list and briefly describe here all the resulting Python classes:
  - `Network` class, the main constructor that supports feed-forward computation through layers and backpropagation.
  - `Layer` class: a linear feed-forward layer that stores weights and biases and implements forward and backward pass. Every layer of the network has its own number of neurons and activation function.
  - `Training` class which perform the gradient descent algorithm with backpropagation needed for learning. We implemented the stochastic gradient descent (SGD) with momentum (with the possibility of using Nesterov), early stopping option (suggested in an article by Prechelt [2]) and a batch parameter for setting the minibatch size or if the method is online or batch.
  - `ActivationFunction` class, with methods: `identity`, `sigmoid`, `tanh`, `relu`, `leakyRelu`
  - `ErrorFunction` class that implements squared and euclidean error, and binary cross entropy.
  - `LearningRate` class, which decays linearly or exponentially eta parameter.
  - `Metric` class, which performs binary class accuracy and euclidean accuracy.
  - `Regularizations` class, for both lasso and ridge regression regularization techniques.
  - `WeightsInitialization` class that perform random or glorot uniform weight initialization.

  This implementation can solve both classification and regression problems simply by changing the activation functions and the number of units in the output layer (i.e. just modifying two parameter).

2.2 Hyperparameters Overview

- `input_dim` (int): dimension of input layer.
- `units_per_layer` (list of integers): number of units for each layer (input layer excluded).
- `act_functions` (list of str): name of the activation functions `ActivationFunction` class.
- `weights_init` (str): type of weight initialization ('random' or 'glorot').
- `bounds` (tuple): limits on the weights' values for random initialization.
- `error_func` (str): name of the error function used.
- `metr` (str): name of the metric used (see above `Metric` class) .
- `batch_size`: it can be "full" for full-batch or a number for mini-batch or online.
- `lr` (float): learning rate (eta parameter).
- `lr_decay` (str): type of decay for the learning rate. It can be `None` for constant learning rate otherwise 'linear' or 'exponential'.
- `limit_step` (int): number of steps of weights update to perform before stopping decaying the learning rate.
- `decay_rate` (float) the higher, the stronger the exponential decay.
- `decay_steps` (int): number of steps taken to decay the learning rate exponentially.
- `momentum` (float): momentum (alpha parameter).
- `lambda_` (float): regularization coefficient.
- `reg_type` (str): name of regularization technique used ('lasso' or 'ridge_regression').

2.3 Preliminary Trials Pursued

Starting from the *MONK's* problems, we obtained good result with the first models tried. As described in an article by Thrun et al.[4], it should be possible to perfectly fit with a model with less than 5 neurons for every test. We reduced the number of neurons as suggested and we succeeded in replicating the results with few trials, which are detailed in subsection 3.1.

2.3.1 Screening phase on MONK datasets

After having built our software architecture and before proceeding with the model selection phase we performed a long screening phase for both regression and classification tasks, trying different values an configuration of hyperparameters and plotting many learning curves in order to see the behavior of our raw network. It was a very long and detailed phase: here we just report some interesting results. Starting with a basic baseline model, we focused on parameters which have a high impact on learning, such as number of hidden units, learning rate, etc. A particular look has been given to the effects of the regularization. Since this was a preliminary stage, we performed hold-out validation with the following split: 70% training, 30% validation.

Exploring hyperparameters effects on model complexity. We noticed that with the MONK datasets it is very easy to overfit, since many learning curves have shown a quite low training error but a sizable validation error. The most straightforward measure we decided to take was reducing the number of hidden units, in order to decrease model complexity. Simultaneously, we increased the eta parameter value to induce an underfitting effect and reduce the gap between high variance and low bias. As a matter of fact, as shown in Figure 2 in Appendix, with smaller values of learning rate parameter we noticed not only an high validation error (overfitting), but also a pretty low value of accuracy, together with the training error that does not decrease anymore after a certain threshold: this is clearly the confirm that, if eta is too low, the network is unable to learn properly. We noticed that our network learns very well from the MONK datasets even with a very small number of nodes in the hidden layer: in general, just three or four units are enough.

Considerations on regularization and stability. As a remedy to overfitting behavior we also explored the effects of regularization (λ parameter), as well as the effects of momentum (α parameter) for what concerns learning curve stability. It is easy to see from Figure 3 in Appendix how a progressive increase of the momentum value is able to reduce the oscillations in the learning curve and to make it really smooth. At the same time, even a minimal increase in lambda has a huge impact in reducing the bias/variance gap, and so correcting overfitting and progressively bringing the learning to the best possible trade off.

One last consideration can be done on weight initialization parameter. Since Glorot initialization is typically used for shallow networks we tried it first, and we noticed an associated pretty good performance of our models.
Regarding random weight initialization, since our network is really small and shallow, we noticed that it is really sensitive to different initialization. This was confirmed by the fact that the resulting learning curve for different runs of the same model (i.e. different execution with the same parameter configuration) were quite different in performance results. For this reason we decided to use Glorot weight initialization.
Regarding the CUP dataset, we started with some preliminary tests to set up the first searches and we will explain all details in subsubsection 3.2.1

2.4 Validation Schema

The dataset for the CUP was split into development set (80% of ML-CUP20-TR) and internal test set (20%). The former was used to perform model selection, while the latter was used for the final evaluation of the chosen models. The validation for the selection is performed using a 5-fold cross validation on all model configurations generated by a grid-search, obtaining the mean and the standard deviation. In order to reduce the search space, we used two levels of grid search:

- **First coarse grid search** with different configurations inserted by hand to find good intervals. For this purpose the screening phase we did proved to be fundamental; this because it allowed us to do a strong filtering and selection on hyperparameters values, so that we were able to know in advance reasonable range to use in the grid search itself.

- **Finer grid search** over smaller interval on the three resulting best models (i.e. those who reported the best results in terms of average validation metric). This phase can be considered an equivalent of a **random search** since it is performed by perturbing each hyperparameter of these best models of a random value picked from a pre-determined range. More details are provided in subsubsection 3.2.2.

Filtering good performance for an efficient and optimized search.
In order to prevent instability and save time we defined that, in the coarse grid search, a model is discarded during the folds of the validation when it performs poorly at specific epoch. Specifically, we implemented

a stopping criteria in the `Training` class which suppresses the training if the validation error, after a fixed number of epochs, isn't below a certain threshold. It was possible to fix those values for threshold and epoch *a priori* on the basis of the results obtained during the screening phase. This strategy took the concrete form of a **hypotheses space pruning**, since it discards those models (hypotheses) for which we anticipate a poor performance. This reduction of the search space allowed us to save much computational effort and time. Combinations that led to overflow have been filtered and discarded too.

3 Experiments

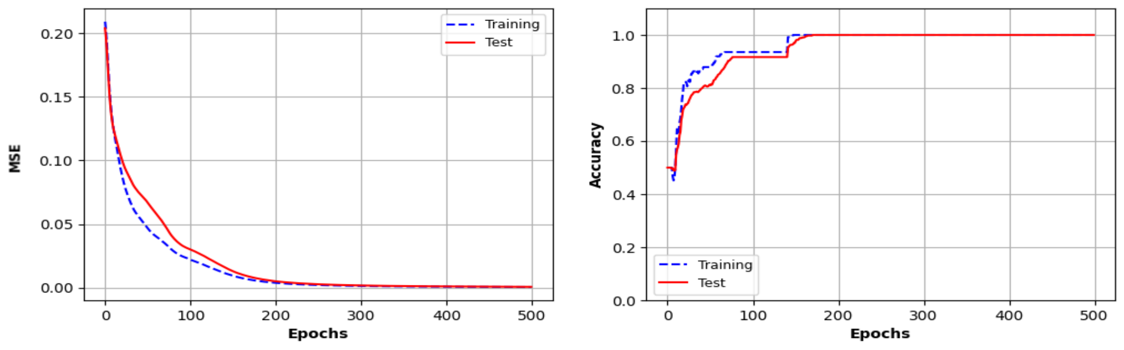
3.1 Monk Results

The inputs were transformed with a one-hot-encoding on all three MONK's datasets and we obtained 17 binary inputs per pattern. We used a single hidden layer neural network for all three MONK problems, the Nesterov momentum, the `relu` activation function for the hidden layer and the `sigmoid` for the output layer. The loss was computed using the Mean Square Error (MSE) on a full batch gradient descent over 500 epochs and the last epoch result is taken for the evaluation. The regularization applied in MONK 3 is L2 and no early stopping is used in this benchmark. In Table 1 are listed the number of hidden units and other hyperparameters used, as well as the scores obtained by averaging the results over 10 trials.
We noticed that MONK 1 and MONK 2 were highly sensitive on the initial weights (i.e. the models would yield good result most of the times but not always, dropping the mean accuracy below 100%). So, we decided to initialize the weights using the `glorot` initialization instead of the `random`, because it is suggested in a paper by Glorot et al. [1], especially for neural networks with a single hidden layer. Same initialization was used for MONK 3 that gave us good results and this permitted the model to consistently reach the desired accuracy for all the problems.

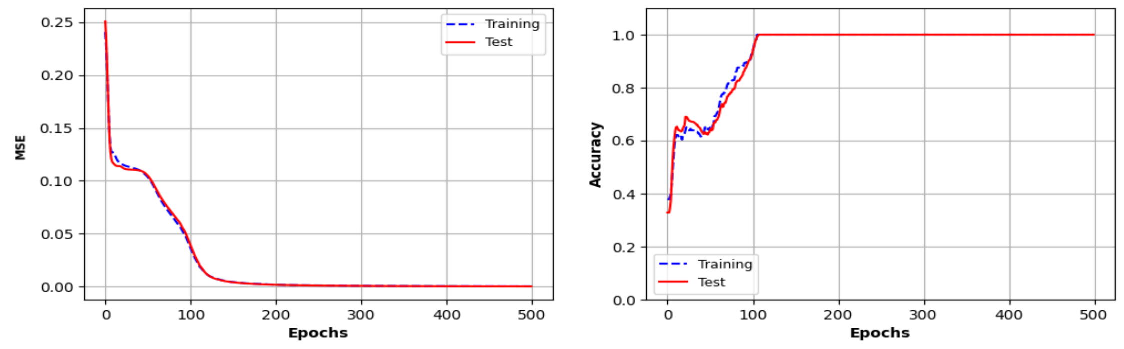
Task	**#Units**	**eta**	**alpha**	**lambda**	**MSE (TR/TS)**	**Accuracy(%) (TR/TS)**
MONK1	4	0.8	0.8	0	0.00572/0.00877	100%/100%
MONK2	4	0.8	0.8	0	0.00398/0.00743	100%/100%
MONK3	4	0.8	0.8	0	0.01411/0.02701	97.21%/93.63%
MONK3+reg.	4	0.76	0.8	0.005	0.01804/0.01782	95.90%/96.67%

Table 1: Average prediction results for 10 trials obtained for the MONK's tasks.

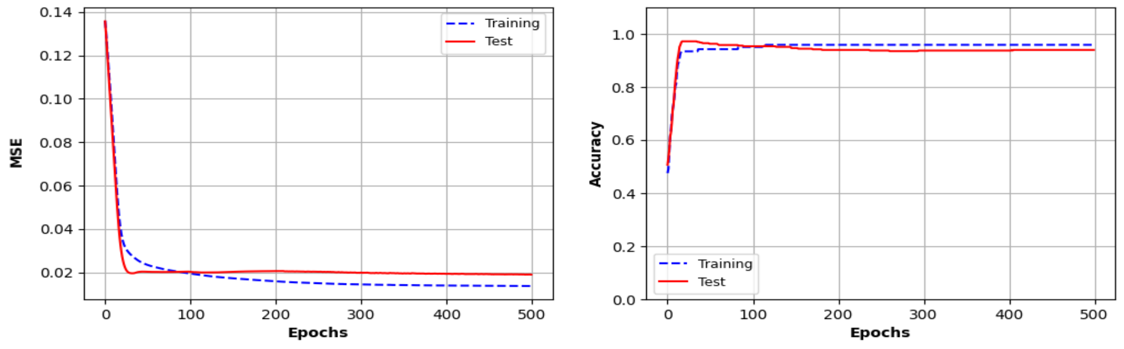
Figure 1 show the learning curves of only one trial for all three MONK's datasets, because the curves of different trials were very similar. We noticed that, on average, MONK 1 (Figure 1a) reaches convergence after 200 epochs and MONK 2 (Figure 1b) after 90 epochs. Instead, the MONK 3 task (Figure 1c) presents overfitting in case of no regularization due to the presence of noise in the dataset.
After applying L2 regularization, we improved the test accuracy but we doubled the average training computing time for one trial w.r.t. to non-regularization case ($\sim$ 35 seconds vs. $\sim$ 18 seconds). At the end, the training of the regularized model is very consistent with its mean results and we obtained a better test accuracy for MONK 3 task.



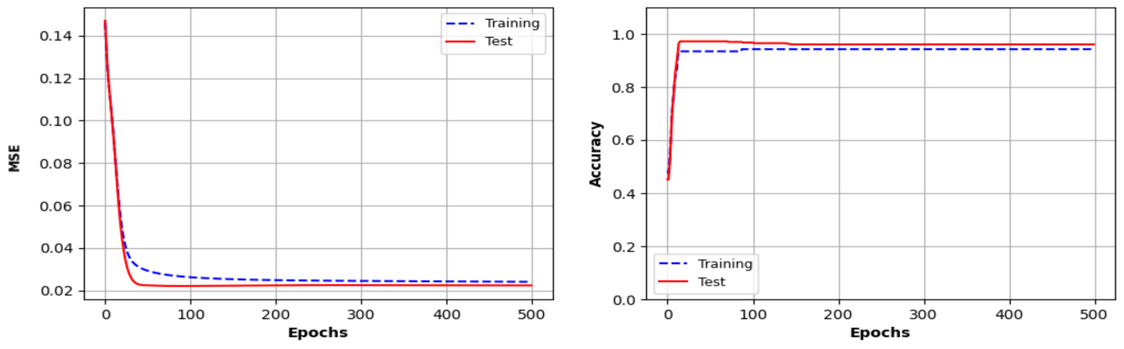
(a) MONK 1



(b) MONK 2



(c) MONK 3 (no regularization)



(d) MONK 3 (with regularization)

Figure 1: Learning curves for all MONK's problems

3.2 Cup Results

3.2.1 Screening Phase

First, we started with some manual random experiments to determine an initial parameter space to start with. We spent time in this phase to experiment different configurations and analyze different learning curves obtained. After some trials we determined that some parameters were not very good because could not reach good results (i.e. under 20 of Mean Squared Error), but also for bad learning curves:

- No big differences between `glorot` and `random` weight initialization, but the first seemed a little more stable.
- Momentum with too higher value (≥ 0.9) gave curves with overfitting. Moreover, Nesterov momentum helped us improving the rate of convergence.
- Batch size too low (≤ 64) caused instability, on the contrary a batch size too high (≥ 512) produced very smooth curves but the error decreased very slowly and, at the end, it wasn't so good (Figure 4).
- Low learning rate *eta* was preferred and it gave us different curves depending on the value:
  - eta > 0.1 generated overflow.
  - $0.05 \leq$ eta ≤ 0.1 produced unstable curves and the error didn't decrease and remained between 20 and 21 (Mean Squared Error).
  - eta $= 0.001$ made good curves as shown in Figure 5.
- `Lambda` value of 0.1 for regularization (both L1 and L2) produced underfitting. Furthermore, we have not been able to recognize a type of regularization much better than the other during this phase, so we kept both of them for the grid search.
- Models with few hidden units (i.e. under 10 units) performed poorly; moreover, more than two hidden layers didn't gave us better results compared to models with one or two hidden layers with a low regularization.
- Leaky ReLU and ReLU produced very similar learning curves, but the latter performed a little bit better. Moreover, they require an value for learning rate very small (≤ 0.001), otherwise they fell into overflow. In the end, however, with ReLU the error decreased slower but it was more stable compared to TanH.
- Both learning rate decay techniques didn't improve the learning, considering that we were already using quite small values for the learning rate. With *eta* values lower than 0.05, we obtained a "parabola error" that increased even after a few epochs (Figure 6). But with *linear decay* and higher values, it gave us interesting learning curves so we decided to keep it as learning rate schedule for the grid search.

At the end, we reduced drastically the hyperparameters search space to be used for the first coarse grid search and this phase was useful to us also because of the computing power at our disposal. The reader can consult the appendix to see some of the learning curves obtained during the screening phase.

3.2.2 Final results

Both the first coarse grid search and the second finer one have been performed as described in subsection 2.4, with a 80/20 split between development and internal test sets.
In total we tested around 1296 configurations for the coarse grid search, and 243 for the random one. They required approximately a total of 50 hours to be completed (32 for the coarse and 18 for the random one). The hardware we used consisted of a Dual-Core Intel Core i5 2,5 GHz and a 4-Core Intel i7 1,8 GHz. We exploited **parallel computing** by setting the maximum number of concurrently running jobs equal to the number of CPUs in our systems.

For the **coarse grid search**, it has been possible to start by inserting values at hand given the results and observations of the deep screening phase described above. Given the available computing power we had, we decided to discard those hyperparameters values that proved to be not so relevant during the preliminary trials, as well as to keep some other fixed.
Regarding discarded values, we decided not to consider exponential decay of eta parameter since during the preliminary trials it always led to poor results: this because we considered appropriate for the given dataset some learning rate values that were already very small.
Moreover, regarding fixed values, we decided *a priori* reasonable values for number of epochs and `limit_step` for learning rate decay. We also fixed `weight_init = 'glorot'`, since we noticed that performed well, so that to save us the search on `bounds` for random weight initialization. The loss for the training was computed using the Mean Square Error(MSE) and we used the Mean Euclidian Error(MEE) for the evaluation. For further motivations about the range of values chosen for each parameter see the observations listed in subsubsection 3.2.1. All values tried in the coarse grid search are shown in Table 2.

Coarse grid search configuration	
Hyperparameter	**Range of values**
`weights_init`	`'glorot'`
`momentum`	`(0.6, 0.7, 0.8)`
`units_per_layer`	`[10,2], [20, 2], [20, 20, 2]`
`act_functions`	`['relu', 'identity'], ['tanh', 'identity'], ['relu', 'relu', 'identity'], ['tanh', 'tanh', 'identity']`
`batch_size`	`(128, 256)`
`lr`	`(0.0001, 0.001, 0.01)`
`lr_decay`	`(None,'linear_decay')`
`limit_step`	`350`
`lambda`	`(0, 0.0001, 0.001)`
`reg_type`	`('lasso','ridge_regression')`
`epochs`	`350`

Table 2: Hyperparameters sets used in the first grid search. Further details on how they have been chosen and why some are fixed or removed can be found in subsubsection 3.2.1 and subsubsection 3.2.2.
For combinations to make sense, `units_per_layer` and `act_functions` are coupled only when their size match.

For each configuration we performed a **5-fold cross validation** on the development set, which returned the average over the 5 folds of Mean Euclidean Error and Mean Squared Error for both training and validation, as well as their respective standard deviations. Resulting **best models** were selected considering the **top 3 ones in terms of val MEE performance** (i.e. prioritizing those with minimum average validation metric achieved). At equal MEE, best models were selected by looking at smallest values of standard deviation.

Once finished the coarse grid search and selected the 3 best models, we proceeded with a **finer grid search** that basically consists on a **random search** because of the way in which we build it: we generated a number of random configurations by perturbating the values of four model's parameters. We chosen the most relevant concerning model complexity, stability and learning in general. Specifically, values for η, λ and α parameters were randomly generated from a normal distribution centered at their base value and with standard deviation of 0.001 for η and λ, and of 0.1 for α. Instead, the `batch_size` was generated by randomly varying its value within a range of 60.
We also inserted a filter that discards too small random generated perturbations that would be irrelevant. We decided to generate, for each hyperparameter, a total of 3 configurations (the base one and two randomly perturbed), so that we finally got a total of 3 different configurations of the 4 hyperparameters subject to variations, for each of the 3 best models.
These 3 fine searches (one per model) generated 81 trials each, for a total of 243 trials run as 8 parallel jobs, that required approximately 6 hours each.
Finally, we picked the best 5 models resulting after this fine grid search (Table 3), also presenting for each model all the values for hyparameters. We can notice that some considerations made during the screening phase has been confirmed such as that two hidden layer architecture performs better, as well as small values for eta parameter without decay. Something that we didn't expect and surprised us is the lambda best value for regularization, turned out to be always zero.

The **ultimate model** used on the blind test set was chosen among the candidates by selecting the most performant one in term of validation MEE. It exactly corresponds to the model ranked first in Table 3, in which its parameters are reported.
Since the average validation MEE alone is not a sufficient estimation of the proper flexibility and generalization capability of a model, we retrained the final model on the entire development set and finally estimated the generalization capabilities on previously unseen data by using the **internal test set**, which was rigorously kept separated from the development set during the whole model selection phase, and never exploited before for any kind of reason. Results are shown in Table 4.
Lastly, Figure 7 in Appendix shows the MSE and MEE curves for each of the five best models selected. The first thing that we noticed from plots is the similarity of performances for all the models, but the first one is a little more stable, confirming our good choice in selecting it as final model for blind test.

4 Conclusion

Model `fine_1` of Table 3 was considered the best performing and so we finalised our work by retraining it on the original training set (ML-CUP20-TR) in order for it to be used for predicting the outputs of the blind test set (ML-CUP20-TS). The network required 13 minutes to be trained on the entire training dataset for 350 epochs.
For **blind test results** we saved the predictions on a result file named `amaroluciano_ML-CUP20-TS.csv`, with the nickname `amaroluciano`.

Model (ranking)	Avg tr MEE	std on tr MEE	Avg val MEE	std on val MEE[b]	topology	act func	eta	alpha	lambda	batch size	eta decay
fine_1	3.0351	0.0525	**3.1901**	0.0389	[20, 20, 2]	['relu', 'relu', 'identity']	0.001	0.6	0	145	None
fine_2	3.0085	0.1077	3.2161	0.1347	[20, 20, 2]	['relu', 'relu', 'identity']	0.001	0.6	0	128	None
fine_3	3.0122	0.0301	3.2251	0.1566	[20, 20, 2]	['relu', 'relu', 'identity']	0.001	0.672663	0	150	None
fine_4	3.0543	0.03422	3.2326	0.2343	[20, 20, 2]	['relu', 'relu', 'identity']	0.001	0.6	0	150	None
fine_5	3.1310	0.0843	3.2744	0.1643	[20, 20, 2]	['relu', 'relu', 'identity']	0.001	0.43136	0	128	None

Table 3: Hyperparameters and performance of the five best models resulting after coarse and fine grid search, based on minimum val MEE. The final model used on the blind test set is the first in ranking, highlighted in green. N.B. we do not report here the regularization type obtained since it is `None` for all models.

Best Model Perfomance

Model	MEE on Development set	MEE on Internal test set
`fine_1`	3.0369	3.2071

Table 4: Average MEE on development set and internal test set for the best model chosen over 10 trials

Acknowledgments

The project development was a great learning opportunity, as it allowed us not only to confirm what we studied during the course, but also to fully understand the meaning and effects of each hyperparameter and how each of them influences the learning of the network. We have also learned how to recognize cases of overfitting, underfitting, instability, and to take actions in regard (e.g. properly balancing regularization, consider inserting early stopping strategies, ...) in order to obtain the best possible result in terms of bias / variance trade-off and generalization capabilities.
We agree to the disclosure and publication of my name, and of the results with preliminary and final ranking.

References

[1] X. Glorot and Y. Bengio. Understanding the difficulty of training deep feedforward neural networks. In Y. W. Teh and M. 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. PMLR.

[2] L. Prechelt. Early stopping - but when? pages 5–6, 03 2000.

[3] scikit learn. Glossary of common terms and api elements.

[4] S. Thrun, J. Bala, E. Bloedorn, I. Bratko, J. Cheng, K. De Jong, S. Džeroski, S. Fahlman, D. Fisher, R. Hamann, K. Kaufman, S. Keller, I. Kononenko, J. Kreuziger, T. Mitchell, P. Pachowicz, Y. Reich, H. Vafaie, and J. Wnek. The monk's problems a performance comparison of different learning algorithms. 01 1992.

Appendix

4.1 MONK screening phase plots

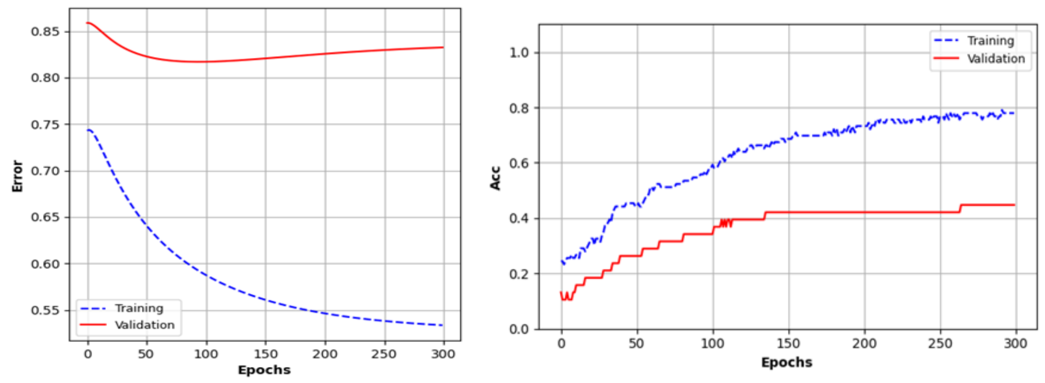
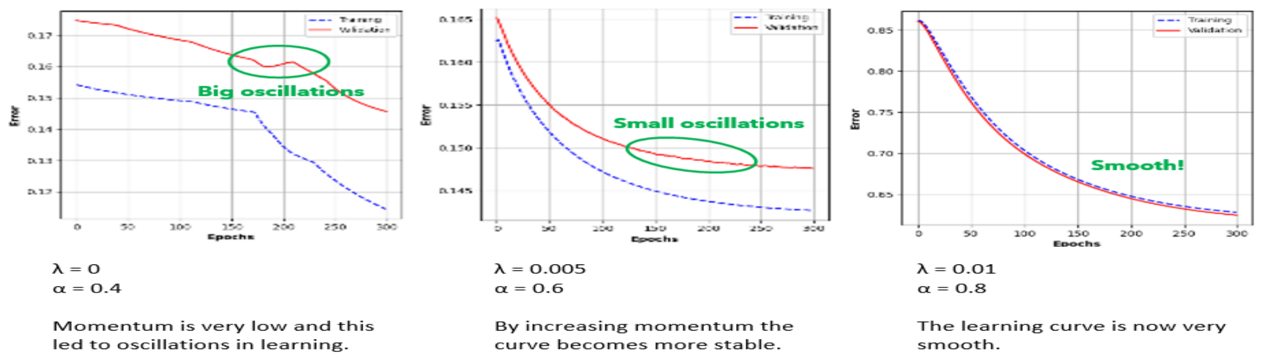


Figure 2: Overfitting performance of a very complex model (12 hidden neuron) with low learning rate value ($\eta = 0.1$) on MONK1 dataset.



λ = 0
α = 0.4

Momentum is very low and this led to oscillations in learning.

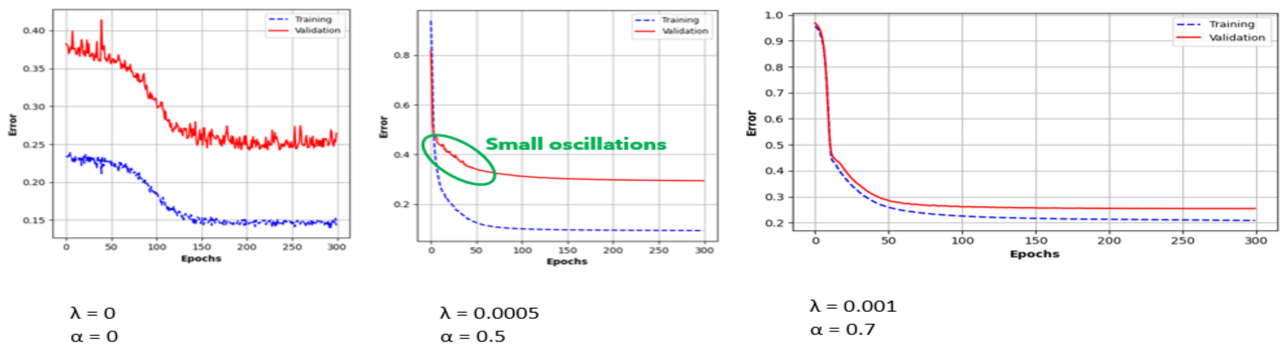
λ = 0.005
α = 0.6

By increasing momentum the curve becomes more stable.

λ = 0.01
α = 0.8

The learning curve is now very smooth.

(a) Monk3 dataset – simple model with 1 hidden layer and 2 neurons



λ = 0
α = 0

λ = 0.0005
α = 0.5

λ = 0.001
α = 0.7

(b) Monk3 dataset – more complex model with 1 hidden layer and 4 neurons

Figure 3: Exploring regularization effects on learning and stability.

4.2 CUP screening phase plots

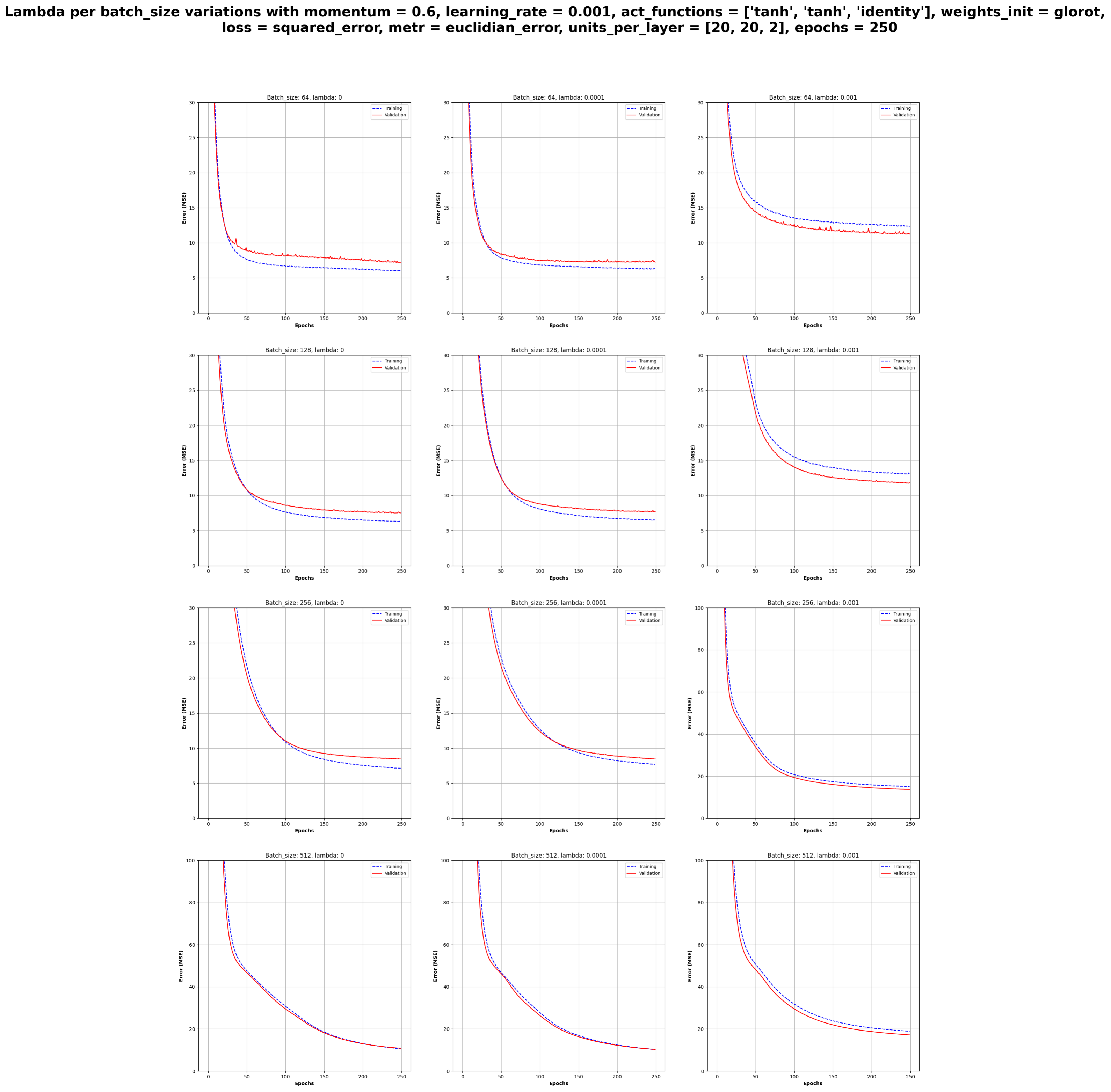


Figure 4: Comparison between batch size and lambda during the CUP screening phase

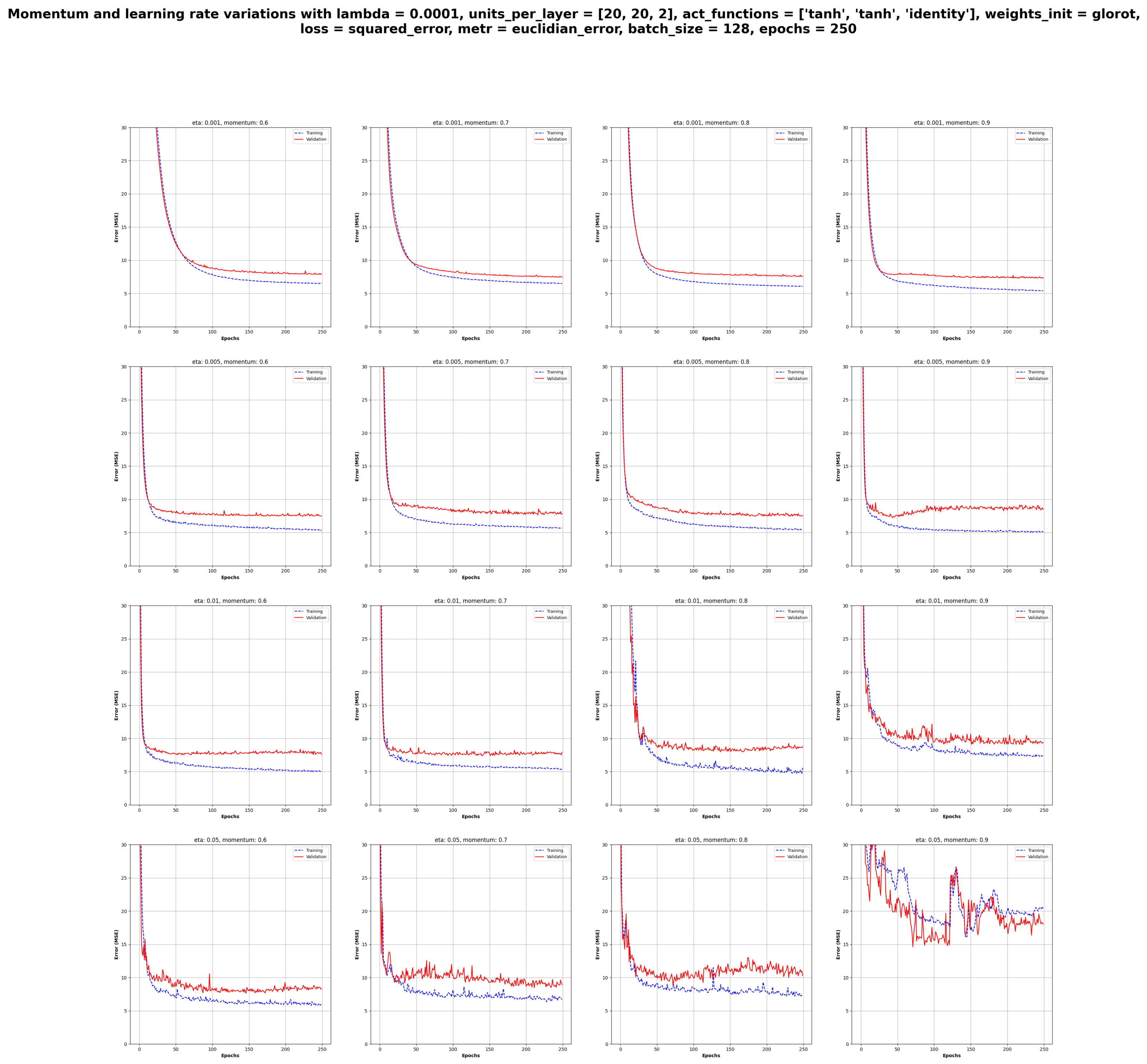

Momentum and learning rate variations with lambda = 0.0001, units_per_layer = [20, 20, 2], act_functions = ['tanh', 'tanh', 'identity'], weights_init = glorot, loss = squared_error, metr = euclidian_error, batch_size = 128, epochs = 250

Figure 5: Comparison between momentum and learning rate (eta) during the CUP screening phase

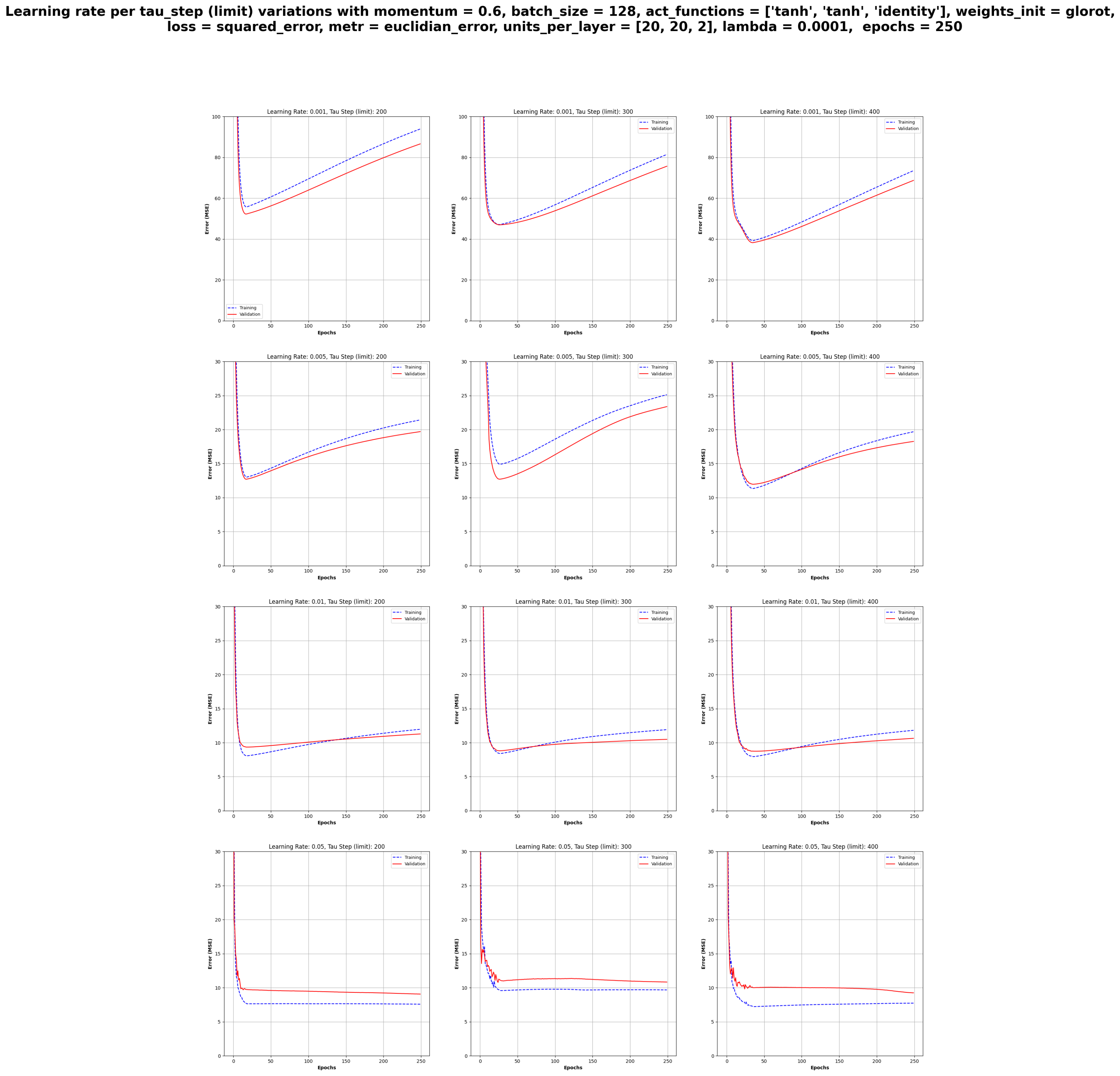
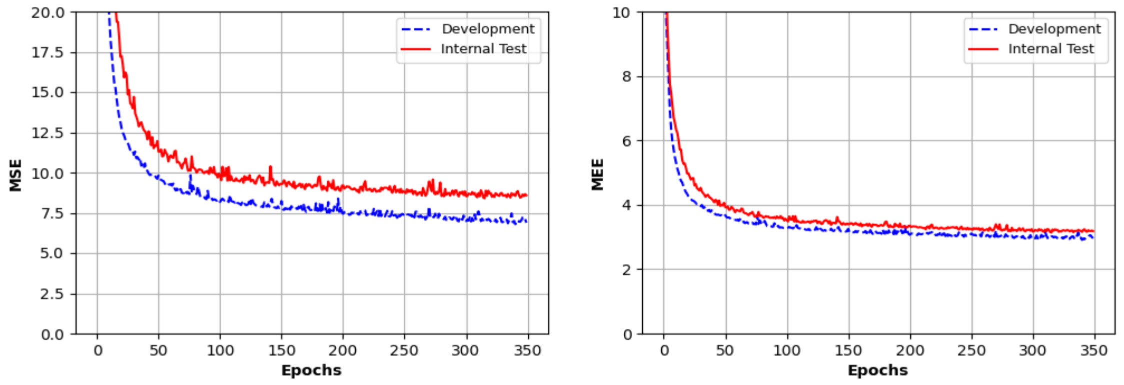
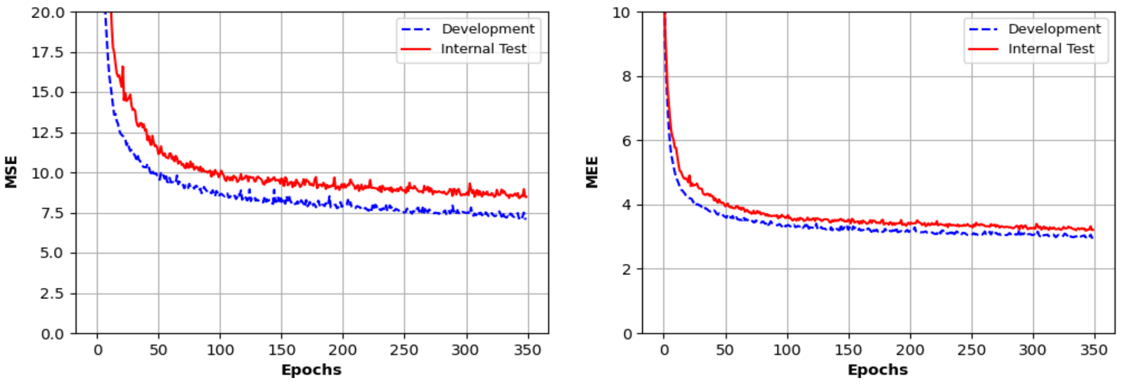

Learning rate per tau_step (limit) variations with momentum = 0.6, batch_size = 128, act_functions = ['tanh', 'tanh', 'identity'], weights_init = glorot, loss = squared_error, metr = euclidian_error, units_per_layer = [20, 20, 2], lambda = 0.0001, epochs = 250

Figure 6: Comparison between learning reta (eta) and limit step for linear decay during the CUP screening phase

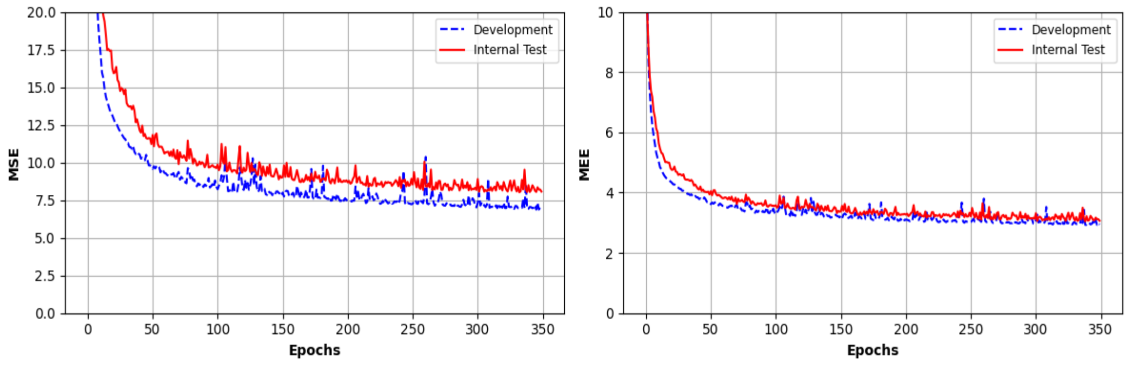
4.3 Best models performance plots for CUP dataset



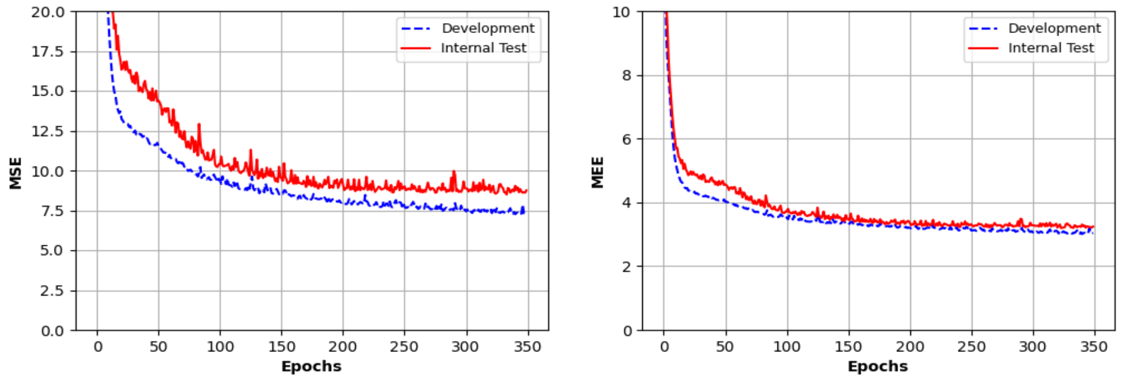
(a) fine_1



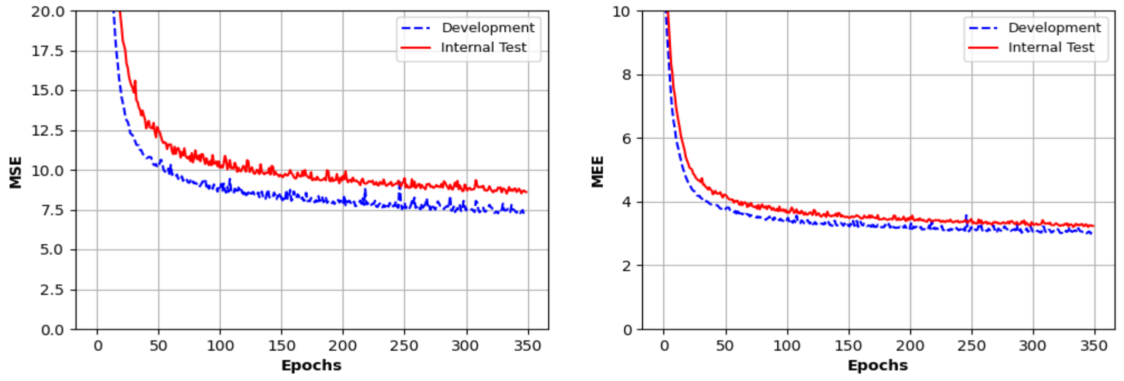
(b) fine_2



(c) fine_3



(d) fine_4



(e) fine_5

Figure 7: Loss (MSE) and metric (MEE) for the best five models