Machine Learning Project

Authors
Piero Viscone, p.viscone@studenti.unipi.it
Adriano Del Vincio, a.delvincio@studenti.unipi.it
Edoardo Gabrielli, e.gabrielli2@studenti.unipi.it
Master Degree (Physics).
ML course (654AA), Academic Year: 2021/2022
Date: 04/01/2022
Type of project: **A**

Abstract

In this project we implemented a Neural Network (with tikhonov regularization, momentum and grid search) using python languages and few libraries like *numpy*, *numba*, *joblib* and *scipy*. We experiment our implementation on MONKS and CUP datasets. for the CUP dataset we select the model using the grid search algorithm and the k-fold cross validation. The best models obtained with the grid search are combined in an ensemble of neural network that average the results of each network. We also exploit the relations between outputs using a different approach based on regression algorithm (non-linear least square), defining analytically ourselves the target function between the two outputs of the ML-Cup.

1 Introduction

We want to experiment our implementation of the neural network and see how the different combinations of Hyperparameters and different type of regularization acts on the learning process of the network. In particular we tested the neural network in a supervised classification task (the three monks dataset) and then we analyzed the ML CUP dataset with a supervised regression task.
We implement the *multi-layer-perceptron* with backpropagation as learning algorithm (with gradient descend), classic momentum, Nesterov momentum, L2 regularization and early stopping. Then we searched the best hyperparameters with a grid search and used the best models to construct an ensemble and perform a model averaging evaluation.
We also exploit a regression technique predicting the first output as a function of the second one (that was predicted by a neural network), defining the fit function ourselves and averaging the result with the ensambling of neural networks.

2 Method

The features that are implemented in this projects are:

- different activation function for the units ("Linear", "Sigmoidal", "Tanh", "Relu")
- Tikhonov regularization (L2) to control model complexity
- *Classic* momentum and *Nesterov* momentum
- *minibatch* learning and *full-batch*
- K-fold cross validation

- early-stopping: we set a parameter *calm* to a given integer value given at the beginning of the training: *patience*. Then, during the training, we checked the Mean Euclidean Error on both training and validation dataset at each epoch i: MEE[i].
  Defining the total error difference at epoch j as $E \equiv$ MME[0] $-$ MEE[j] and the last error difference as $e \equiv$ MEE[$j-1$] $-$ MEE[j] we check the parameter:

  $$t = \frac{e \cdot j}{E}$$

  This parameter represent the relative error of the step j. If this parameter is lower than 0 (for training or validation dataset) the parameter *calm* is decreased by 1, otherwise $calm = min(calm+1, patience)$. If *calm* reach zero we stop the training process.

- Different weight initialization strategies.

  – For all layers, except the ones with the ReLU activation function, we used the **Xavier initialization** [1]

  $$W \sim U\left(-\sqrt{\frac{6}{N+M}}, \sqrt{\frac{6}{N+M}}\right)$$

  where N is the number of units in the layer and M is the number of units in the previous layer of the network

  – For the layers with the ReLU activation function we used the **He initialization** [2]

  $$W \sim \mathcal{N}\left(0, \sqrt{\frac{2}{N}}\right)$$

  where N is the number of the unit in the layer

- *MLP_W*: a class that initialize a number of n_candidates neural network with different starting weights. It train all the neural networks and return the one with the best validation error

- *RMSProp* optimization for adaptive learning rate. [3]

  Given $\boldsymbol{dW} = -\boldsymbol{\nabla_W} E$ and $\boldsymbol{db} = -\boldsymbol{\nabla_b} E$, where $\boldsymbol{W}$ are the weights, $\boldsymbol{b}$ are the biases, and E is the loss function. The weight update rule with RMSProp is:

  $$\boldsymbol{S_w} = \beta \boldsymbol{S_w} + (1-\beta)\boldsymbol{dW}^2$$
  $$\boldsymbol{S_b} = \beta \boldsymbol{S_b} + (1-\beta)\boldsymbol{db}^2$$
  $$\boldsymbol{W} = \boldsymbol{W} - \frac{\eta_0}{\sqrt{\boldsymbol{S_w}} + \delta}\boldsymbol{dW}$$
  $$\boldsymbol{b} = \boldsymbol{b} - \frac{\eta_0}{\sqrt{\boldsymbol{S_b}} + \delta}\boldsymbol{db}$$

  Where η_0 and β are hyperparameters and δ is a small values that we fixed to 10^{-8} that is needed to avoid divisions by 0.

  The idea behind the RMSProp is to normalize the gradient doing a moving average (elementwise) of squared gradients, decreasing the step for large gradients and increasing the step for small gradients.

- Saving the network : we implement a function to save the weights of the net in a JSON format. So the training session can be stopped and resumed every time it's needed.

- An ensambling class that perform Bootstrap resampling (bagging algorithm) and Model Averaging. Given a dataset $(x, d(x))$ and an ensemble of neural network E composed by N network n_i the ensemble prediction for a regression task is computed as:

  $$f_E(x) = \frac{1}{N}\sum_{i=1}^{N} f_i(x)$$

  Where $f_i(x)$ are the predictions of the network n_i.
  We can also obtain the correlation between to network of the ensemble as:

  $$C_{ij} = \sum_{k=1}^{m} E_i(x_k) \cdot E_j(x_k) \qquad\qquad (E_i(x) = f_i(x) - d(x))$$

  Since different networks could provide correlated outputs we also put a threshold on the correlation in order to prune unnecessary network from the ensemble. The network that respect the relation 1 can be pruned without sacrificing generalization capabilities[5].

  $$(2N-1)\sum_{i=1}^{N}\sum_{j=1}^{N} C_{ij} \leq 2N^2 \sum_{\substack{i=1\\ i\neq k}} C_{ik} + N^2 E_k \tag{1}$$

- A non-linear least square regressor based on Levenberg–Marquardt algorithm [4].
  This algorithm is an iterative procedure that, given a set of m points $[x_i, y_i]$ and a function model f dependent on a set of parameters β, minimize the sum of the squares of the deviations S with respect to β:

  $$S = argmin_\beta \sum_{i=1}^{m} \left[y_i - f(x_i, \beta)\right]^2$$

  The details of the implementation is in the file **regressor.py**.

2.1 Strategy of developement

We used *numpy* for linear algebra and *numba* and *joblib* for speed up the training process, and we used *scipy* for the "sigmoidal" activation function and numpy for the tanh. For the ReLU and the linear we choose to define them directly.
The network is implemented with two different classes, **Layer** and **MLP**. The MLP class contains a list of Layer representing the structure of the network, the first layer is initialized with the dataset (as a matrix where row represents the patterns and the columns are the features), the last layer return the predictions. The layer class contains a matrix where each element (i,j) represents the weights of the connection between the j-th unit of the previous layer and the i-th unit of the current layer. Besides, an array represents the bias the neurons in the layer. So the number of rows of the weights matrix is the number of the units of the layer, and the number of the columns is the number of units of the previous layer. The output of the layer is evaluated with matrix product between the output of the previous layer and the matrix of the weights, and then the activation function is applied to the result. The network repeats this process as far as the output layer is reached.
The learning algorithm that we used is the back-propagation in the matrix form.
In our strategy of implementation, since we evaluate dot product with different matrices, we update the weights for each batch at the same time. To summarize:

- **layer.py** contains the structure of the single layer, in form of weights matrix.
- **MLP.py** contains the structure of the entire neural network (as list of layers) and the main function to train it (like the learning step, the computation of the gradient, and besides *saving the net* function).

- **preprocessing.py** file where we implement the *K-fold* algorithm and a Function to normalize data
- **losses.py**, **grid_search.py**, **activations.py** files where we implement the different error functions, the activations function for the units of the net, and the *grid_search* algorithm.
- **regressor.py** file where we implemented the non linear least square regressor with Levenber-Marquardt algorithm.
- **ensemble.py** file where we implemented ensambling techniques like model averaging.

3 Experiments

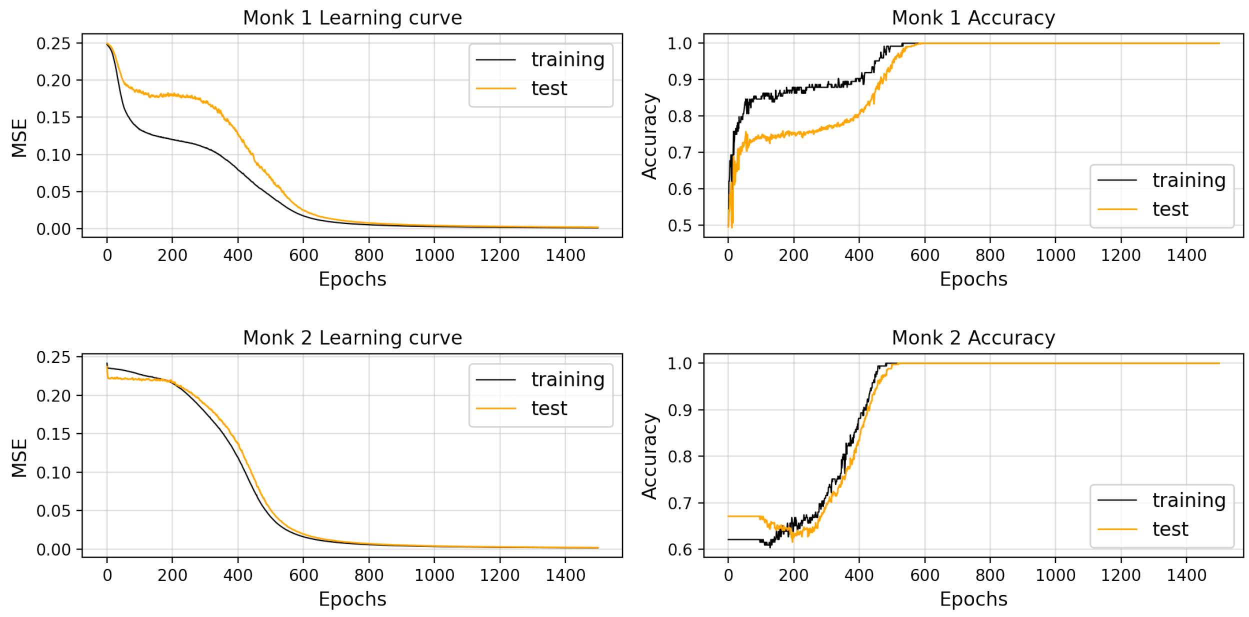
3.1 Monk Results

We decided to use a neural network with one hidden layer made of four units for all the MONK data-sets. This model, as shown in Table 1, is complex enough to solve each MONK's task.
The hyper-parameters of the networks trained are also reported in Table 1.

MONK	**N**	η	λ	α	**mb**		**MSE** $\cdot 10^2$	**Accuracy**
Monk1	4	0.95	0	0.9	50	**TrSet**	1 ± 2	0.99 ± 0.03
						TestSet	2 ± 3	0.98 ± 0.04
Monk2	4	0.7	0	0.6	50	**TrSet**	0.14 ± 0.02	1.0 ± 0.0
						TestSet	0.16 ± 0.03	1.0 ± 0.0
Monk3	4	0.9	0	0.8	Full batch	**TrSet**	4.09 ± 0.02	0.96 ± 0.00
						TestSet	4.80 ± 0.06	0.942 ± 0.002
Monk3+reg	4	0.9	0.001	0.8	Full batch	**TrSet**	6.40 ± 0.04	0.93 ± 0.00
						TestSet	5.14 ± 0.05	0.97 ± 0.00

Table 1: Average over 60 prediction results obtained for the MONK's tasks (N is the number of units, mb is the batch size used for the minibatch). The errors are obtained performing the mean and the standard deviation over the 60 prediction

In Figure 1 we show the trend of the MSE and the accuracy during the training.



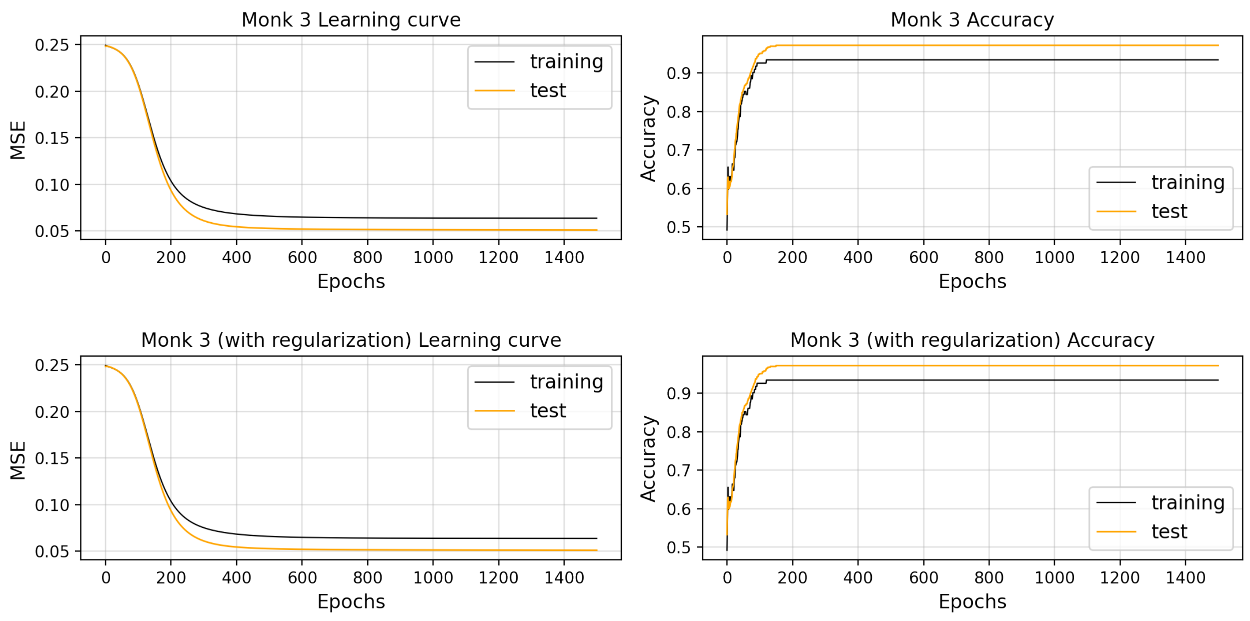


Figure 1: Plot of the MSE and accuracy for the MONK's benchmarks. We see that our neural network is able to reach 100% accuracy in the monk1 and monk2.

3.2 Cup Results

We started trying different models observing their learning curves and looking for a good range of hyper-parameters for the grid search.
We normalized all the data shifting each feature by his mean and dividing it by its standard deviation. For the labels, to preserve the scale, we decided to shift each label by its mean but divide both labels by the larger standard deviation of the two. For the prediction phase we normalize the input data with the mean and the standard deviation of the training dataset and we denormalize the predicted output with the mean and the standard deviation used to normalize the labels of the training set.

In all our trainings, we decided to exploit the *early-stopping*, setting the parameter error-threshold to 0 and patience to 300.

We spilt the data in : 85%, 15% (internal training set, test set).

3.2.1 Screening Phase

We divided the internal training set in training and validation set (75%,25%) to try different hyper-parameters and choose a specific set of values to put in the grid search, analyzing the learning curves of testing and validation error.

3.2.2 Grid Search

After this first screening phase, we performed a grid-search using k fold cross validation over the internal training set with $k = 4$. We report in Table 2 the parameters of the grid-search and in table 3 the list of the models.

batch-size	η learning-rate	α momentum	Nest.	RMS-prop β	λ Tikhonov reg.
32, full	$10^{-4}, 5 \cdot 10^{-4}$	$0.1, 0.2, 0.3$	✓	$0.75, 0.8, 0.85$	$10^{-7}, 10^{-6}, 10^{-5}$

Table 2: Grid search parameters explored.

Models	1	2	3	4	5	6	7	8
Layer1	20	5	10	10	10	15	10	15
Layer2	✗	5	10	5	20	15	10	15
Layer3	✗	✗	✗	✗	✗	✗	10	✗

Table 3: Structure of the hidden layers explored in grid search. The activation function of the output layer is the linear function. For each model,we fix tanh as the activation function of the first hidden layer, and then for the following hidden layers we tried both tanh and ReLU.

For each combination of hyperparameters in the grid search we used *MLP_W class* with n_candidates=3 i.e. for each model we have trained the same model 3 times with different starting weights and chosed the best one.
The training and the validation error is obtained performing the mean and the standard deviation over the 4 result of the k fold. We report the best models (in term of MEE error on validation data set) in Table 4.

model	η	hidden actv	α	β	λ	Full batch	MEE error	
5	$5 \cdot 10^{-4}$	tanh	0.2	0.75	$1 \cdot 10^{-5}$	✓	**TR**	1.03 ± 0.02
							VL	1.14 ± 0.02
3	$5 \cdot 10^{-4}$	tanh	0.2	0.8	$1 \cdot 10^{-7}$	✓	**TR**	1.02 ± 0.04
							VL	1.14 ± 0.04
5	$5 \cdot 10^{-4}$	tanh	0.1	0.8	$1 \cdot 10^{-6}$	✓	**TR**	1.04 ± 0.04
							VL	1.15 ± 0.03
7	$5 \cdot 10^{-4}$	tanh	0.2	0.75	$1 \cdot 10^{-6}$	✓	**TR**	1.05 ± 0.05
							VL	1.15 ± 0.03
4	$5 \cdot 10^{-4}$	tanh,ReLU	0.2	0.8	$1 \cdot 10^{-6}$	✓	**TR**	1.04 ± 0.03
							VL	1.17 ± 0.04

Table 4: Best models from the grid searches.

The best model took 7s to train all over the internal training set on a Intel i5 6600.
We report the learning curve of the best model in figure 2 with the residuals.

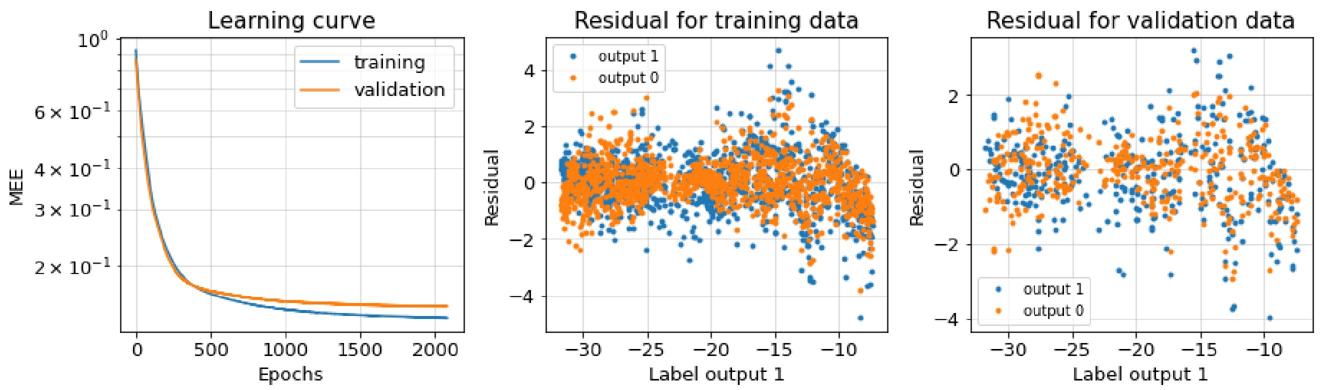


Figure 2: Learning curve and residuals of the best model. Note that the MEE in the learning curve is computed on the normalized data

3.2.3 Analysis of outputs relationship

In figure 3 we plotted the two predicted outputs.
We can see a non linear relationship between the two output in the middle of the output1 that the neural network is not capable to fit well.

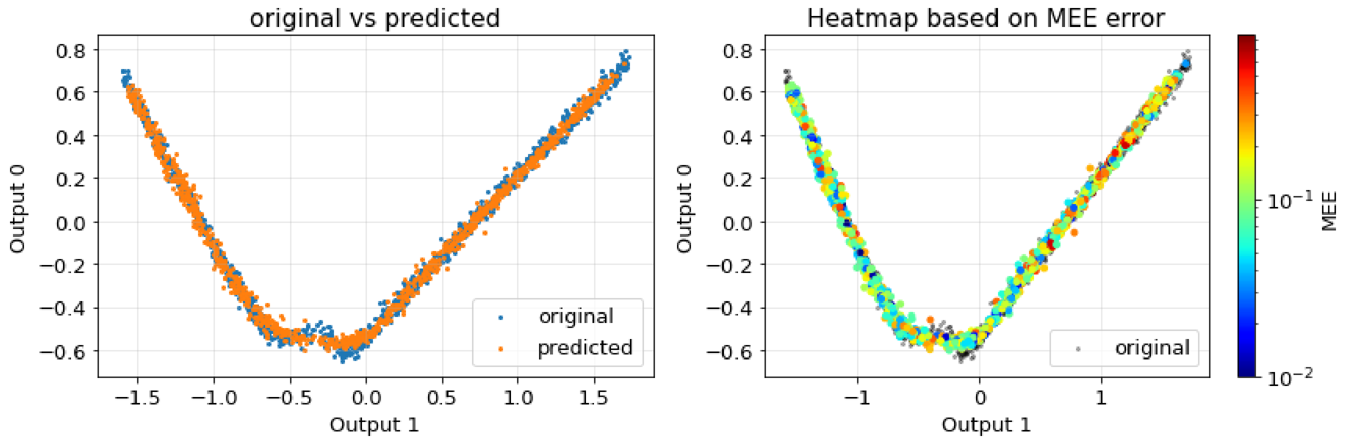


Figure 3: Scatter plot of predicted output0 in function of predicted output1. In the right figure there is a heatmap of the MEE for each single point

Therefore we treated an output as a function of the second one. We have choosen a function that fit the output using the regression algorithm introduced above. The model of function used for the regression is:

$$f(x) = \begin{cases} m_0 x + q_0 & \text{if } x < p_1 \\ (m_1 x + q_1)(\sin(\omega x + \phi) + \Phi) & \text{if } p_1 \leq x \leq p_2 \\ m_2 x + q_2 & \text{if } x > p_2 \end{cases}$$

The parameters q_0 and q_2 are fixed analytically so that the function is continuous, the rest are free parameters (also p_1 and p_2).
In figure 4 we plotted the result of the fit.

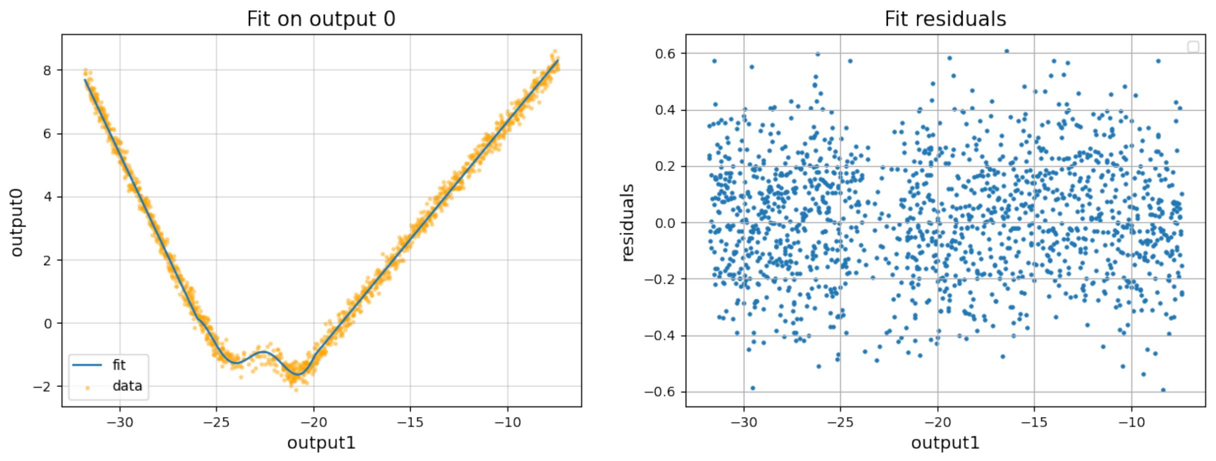


Figure 4: The best fit result on all the training data

We used the prediction of the output1 to estimate the output0 with the fit and then we took the average

between it and the prediction of the output0 obtained by the neural network.
In figure 5 it is possible to see how now also the non linear zone is well fitted.

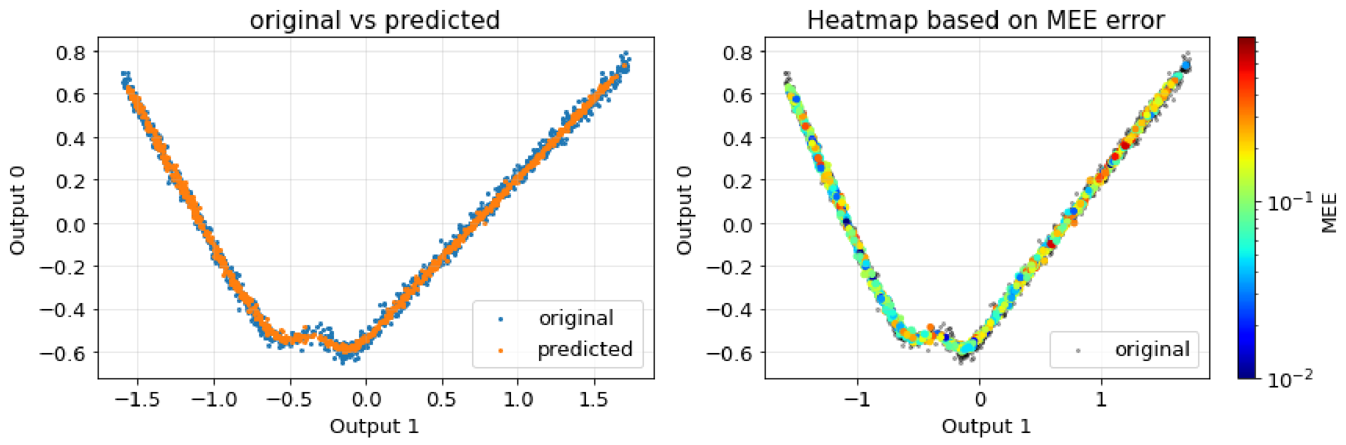


Figure 5: The output0 is obtained by the average between the prediction of the best neural network and the result of the regression on the function that we have defined

To evaluate this strategies we tested it splitting the internal training set in a proportion of (TR 75%, VL 25%). Then we retrained the best model and computed the MEE with and without the average of output0 over the fit.
The error with and without the average are reported in table 5

	with fit	without fit
TR MEE	1.006	1.016
VL MEE	1.095	1.105

Table 5: MEE on training and validation data with the average over the fit and without it. The fit strategies give us an improvement of $\sim 1\%$ over the MEE

This strategy seems to improve the results of the model so we decided to adopt it later also with the ensembling of neural networks.

3.2.4 Ensembling

After doing the grid-search, we decided to make an *ensemble* and perform a model averaging.
We choose the models with different architectures, in particular the model 4, model 5, model 3 and model 7 in table 3 and with hyperparameters in table 4.
For the model 5 we choosed only the hyper parameters with the best results.

We splitted the internal training set with the hold-out validation with the proportions (TR 75%, VL 25%).
For each model we created 5 different network and for each network we exploit MLP_w with n_candidates=3 (it choose the best model between 3 different networks with different starting weights). For each network we used, again, early stopping with error_threshold=0 and patience=300.
At the end of the training of the ensemble we runned the *pruning* method on the ensemble and it pruned just 1 regressor over 20. Then we averaged the prediction of output1 of the 19 models, while for the output0 we performed a further average using the prediction of the ensemble together with the fit function prediction.
The errors on training and validation data were in table 6

	MEE
TR	0.963
VL	1.070

Table 6: MEE on training and validation data of the average between the ensemble and the fit strategies. Also in this case using the fit strategies give us an improvement of the 1% over the MEE

To evaluate the performances of our model we computed the MEE over the test set that we set aside at the beginning. The **Test error** is 1.071.

3.2.5 Final model

In the end we retrained the model with all the data in TR.csv, splitting it in (TR 90%, VL 10%) using the validation data to exploit the early stopping and then we predicted the data in TS.csv

4 Conclusion

We implemented a model for neural network, using it to solve the task in the machine learning cup. We learned that using ensemble technique like models averaging and the fit strategies that we defined can lead to a better result than a single neural network.
Besides we explored different approach, joining the capability of a neural network model with a regression algorithm in order to take the best from every approaches.

The name of the result file: bestfitboy_ML-CUP21-TS.csv

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. In Yee Whye Teh and Mike Titterington, editors, *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, volume 9 of *Proceedings of Machine Learning Research*, pages 249–256, Chia Laguna Resort, Sardinia, Italy, 13–15 May 2010. PMLR.

[2] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *2015 IEEE International Conference on Computer Vision (ICCV)*, pages 1026–1034, 2015.

[3] Geoff Hinton. `http://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf`.

[4] Kenneth Levenberg. A method for the solution of certain non-linear problems in least squares. *The Quarterly of Applied Mathematics*, (2):164–168, 1944.

[5] Zhi-Hua Zhou, Jianxin Wu, and Wei Tang. Ensembling neural networks: Many could be better than all. *Artificial Intelligence*, 137(1):239–263, 2002.