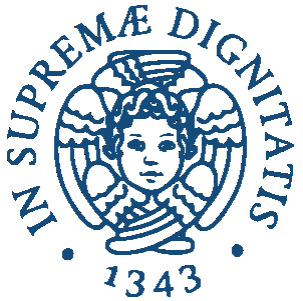

Master Degree in Computer Science
Curriculum Artificial Intelligence

Machine Learning course - 654AA
A.Y. 2022/2023

Machine Learning Project

Type of project: **A**

Bernardo D'Agostino - b.dagostino2@studenti.unipi.it
Andrea Roncoli - a.roncoli@studenti.unipi.it
Bianca Ziliotto - b.ziliotto37@studenti.unipi.it

Delivered on: 23/01/2023

Abstract

In this report we present a Python-based Neural Network for binary classification and regression. We evaluated the performance of the models using the MONK dataset for binary classification and the 2022-2023 Machine Learning Cup dataset for regression. We also conducted various experiments, including testing different combinations of hyperparameters and variants of the learning algorithm. We also implemented a Randomized NN version of our MLP class.

1 Introduction

In this document we will show a Python-based artificial neural network class that is designed to tackle both binary classification and regression challenges. The model was initially employed to address the MONK's problem, a binary classification task widely known and thoroughly explored in [5]. Next, it was tested on the Machine Learning Cup 2022-2023, a regression problem, with special attention paid to the validation process.

In the case of MONK's 1 and 2, there was little need for fine-tuning, as an accuracy of 100% could be achieved with a broad range of hyperparameters. Therefore, a straightforward set of hyperparameters without regularization was selected. For MONK's 3, where some noise is present in the samples distribution, both regularized and unregularized versions of the model were developed. Similarly to the previous cases, optimal hyperparameters combinations were identified with minimal tuning.

As for the Machine Learning Cup 2021-2022, which necessitated a more intricate validation process described in 3.4.1, an exhaustive grid search process was carried out to fine-tune the hyperparameters.

We'll also describe the experiments we performed with the models trying different optimization and backpropagation strategies, different values of its hyperparameters and also the implementation of a Randomized NN based on our MLP class.

2 Methodologies

The project was developed in Python entirely from scratch, with the exception of some mathematical functions from the `numpy` library and some minor utilities from `pandas` and `sklearn` libraries for dataset management and splitting. No other external libraries were used for the MLP implementation. `concurrent.futures` library was used for parallelizing the grid search process.

2.1 Implementation of the MLP model

We implemented a fully-connected feed-forward architecture that uses different variations of the Gradient Descent algorithm for learning.

2.1.1 Architecture and Basic Methods

The architecture is defined by the multi-layer perceptron `MLP` class, whose fundamental attributes are the layers of the network.

Class `Layer` has three subclasses, each one for a different type of layer:

- `FullyConnectedLayer`: a layer that outputs a linear combination of the outputs of all the units of the previous layer;
- `ActivationLayer`: a layer that outputs a non-linear function of its input, computed element-wise;
- `DenseLayer`: a layer obtained by the combination of the two previous classes.

Each type of layer has a `forwardprop` method to forward propagate the input and a `backprop` method to backward propagate the loss. These functions can be found in appendix **??**. The hidden layers of the MLP are all of class `DenseLayer`. The output layer is a `FullyConnectedLayer` for regression tasks, whilst it's a a `DenseLayer` with `tanh` activation function for classification tasks.

The method `fit` of the `MLP` class trains the model on the provided training set, given an error function and a specified combination of hyper-parameters.

The error function used for training is Mean Square Error (`MSE`) both in regression and classification. As evaluation functions for model selection and assessment, we used `Accuracy` for classification and Mean Euclidean Error (`MEE`) for regression. These statistics can be evaluated calling the `evaluate_model` method, and are saved in class attributes such as `learning_curve` and `validation_curve` for plotting and model evaluation.

2.1.2 Variations and Improvements

In building the model we explored different techniques, including:

- Regularization: L1, L2, ElasticNet
- Momentum: Standard, Nesterov
- Weights initialization methods: standard, Xavier, He
- Activation functions: ReLU, tanh, sigmoid
- Weight update policies: batch, mini-batch, online
- Early stopping
- Adaptive gradient optimization variation
- Resilient backpropagation variation

The AdaGrad (short for adaptive gradient) optimization algorithm adapts the learning rate for each weight individually, based on the historical gradient information [4], as shown in 1.

Learning step is adjusted, diminishing for weights that have been updated a lot (i.e. had high gradients in the past).

$$\theta_{t+1,i} = \theta_{t,i} - \frac{\eta}{\sqrt{G_{t,i} + \varepsilon}} \bullet g_{t,i}$$

Figure 1: Adagrad Weights Update (G keeps track of previous gradients)

We implemented resilient backpropagation (Rprop), which is a gradient descent algorithm that only uses the signs of gradients to compute updates. The idea behind Rprop is that comparing the gradient's sign of the current and previous iteration we can understand if we are approaching the minimum (gradients have same sign) or if we are jumping over it (gradients have opposite sign). The algorithm is shown in Figure 2.

$$\eta_i^{(t)} = \begin{cases} \min(\eta_i^{(t-1)} * \alpha, \eta_{\max}) & \text{if } \frac{\partial E^{(t)}}{\partial w_i^{(t)}} * \frac{\partial E^{(t-1)}}{\partial w_i^{(t-1)}} > 0 \\ \max(\eta_i^{(t-1)} * \beta, \eta_{\min}) & \text{if } \frac{\partial E^{(t)}}{\partial w_i^{(t)}} * \frac{\partial E^{(t-1)}}{\partial w_i^{(t-1)}} < 0 \\ \eta_i^{(t-1)} & \text{otherwise} \end{cases}$$

Figure 2: Rprop Step Update

We also implemented a `GridSearch` class which allows for a search over combinations of lists or ranges of values for multiple parameters (either testing all combinations or only a certain number of randomly extracted combinations), using k-fold cross-validation. `GridSearch` has an attribute `parallel` which may be set to `True` to perform the grid search with parallel processing, instead of simply computing an iteration over all combinations, greatly decreasing the time needed to train all models.

3 Experiments

3.1 MONK's Results

We transformed the original data using one-hot-encoding, which created a vector of 17 binary components. Binary encoding of classes was transformed from 0/1 to ± 1 as it was more suitable for `tanh` function, which we used as activation for all layers, including output. We initialized the weights using the Xavier method as suggested in [1]. During training, we used all the data observations in each batch as per the course instructions. The specific settings (hyper parameters) used for each task are listed in Table 1.

Each model was trained and tested 3 times, each time with a different random weights initialization. Table 2 shows the results obtained. The mean squared error and accuracy for training and test sets were computed in terms of average over the random runs. For each model, the learning curves of one of the runs are shown in Appendix A. These are smooth and converge

Task	Epochs	η	λ	α
M1	1500	0.15	0	0.3
M2	1500	0.15	0	0.3
M3	1500	0.15	0	0.3
M3 (L2 reg.)	1500	0.15	0.05	0.3

Table 1: Hyperparameters for MONK's problems

fast. Among the various hyperparameters combinations for which the model behave well, we chose the ones we preferred the most according to the training learning curve shape.

Task	MSE (TR/TS)	Accuracy (TR/TS)
M1	0.1e-2 / 0.2e-2	1.00 / 1.00
M2	0.09e-2 / 0.01e-2	1.00 / 1.00
M3	0.087/ 0.216	0.975 / 0.935
M3 (L2 reg.)	0.279 / 0.222	0.934 /0.972

Table 2: Results for MONK's problems

3.2 Hyper parameter testing

The aim of the following experiments is to isolate and observe the effect of some hyperparameters on the training process. We tested how changing one specific setting (hyperparameter) affected the performance of our MLP model on a regression task. In particular, we experimented with different values for step size, momentum and regularization and for Rprop and AdaGrad variants.

3.2.1 Learning step

Figure 3 shows different learning curves due to different values of the learning step in the same model (topology [40,20,2], activation function `sigm`), trained with momentum 0.5 and batch size 250. Using higher values for the learning step speeds up the learning process, as we can see from the training error falling abruptly. This holds until the value of the step becomes too high and the learning process is no more stable as the error diverges.

3.2.2 Momentum

In Figure 4 we plotted three different learning curves for different values of the momentum coefficient on the model (topology [60,20,2], activation function `ReLu`), trained using a learning step of 0.02 and full batch. The momentum increases the weights updates whenever the direction of the gradient remains stable, while it smooths the variations of the weights when the sign of the gradient oscillates at every step. Since there are no oscillations in the original learning curve, applying a higher momentum speeds up the learning process.

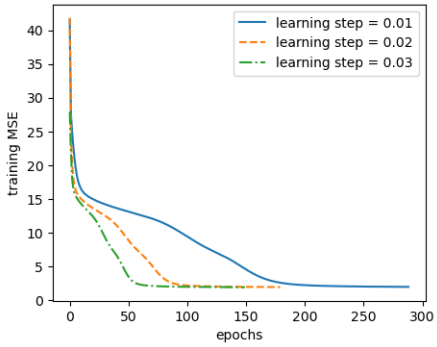

Figure 3: Learning curves with different learning steps

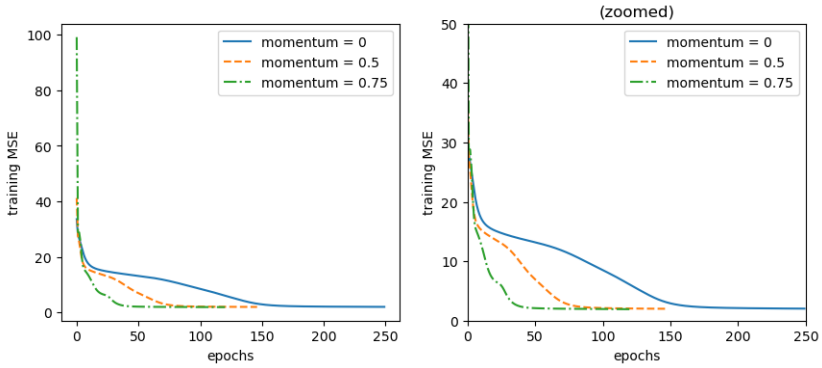

Figure 4: Learning curves with different momentum

3.2.3 Nesterov

In Figure 5 we can see three different learning curves: the first is obtained without applying momentum, the second one shows the effect of a standard momentum of value 0.8 and the last one results from applying Nesterov momentum with the same value 0.8. The model used for the experiment has topology [40,20,2] and activation function `ReLU`, and the learning step used is 0.05. The zoomed graph reveals that Nesterov can achieve slightly faster convergence.

3.2.4 Regularization

Figure 6 shows the effect of introducing a regularization term (L2 = 0.005) on the training and validation error of a complex MLP topology ([400,200,100,2]) with `relu` activation function, learning step = 0.03, momentum = 0.7 and batch size = 500. If we zoom the graph we can easily observe how weights decay reduces overfitting and stabilizes the learning curve.

3.2.5 Rprop

Figure 7 shows two different learning curves from the same model (topology [40,20,2], activation function `tanh`): the first one is obtained with learning step = 0.03, momentum = 0.7, elastic

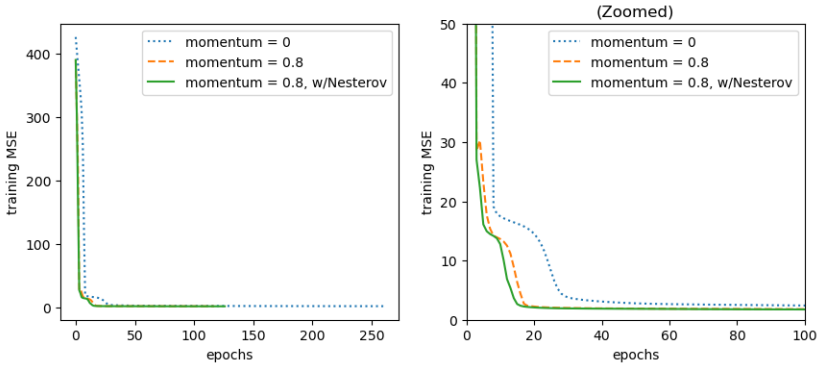

Figure 5: Learning curves with momentum and Nesterov momentum

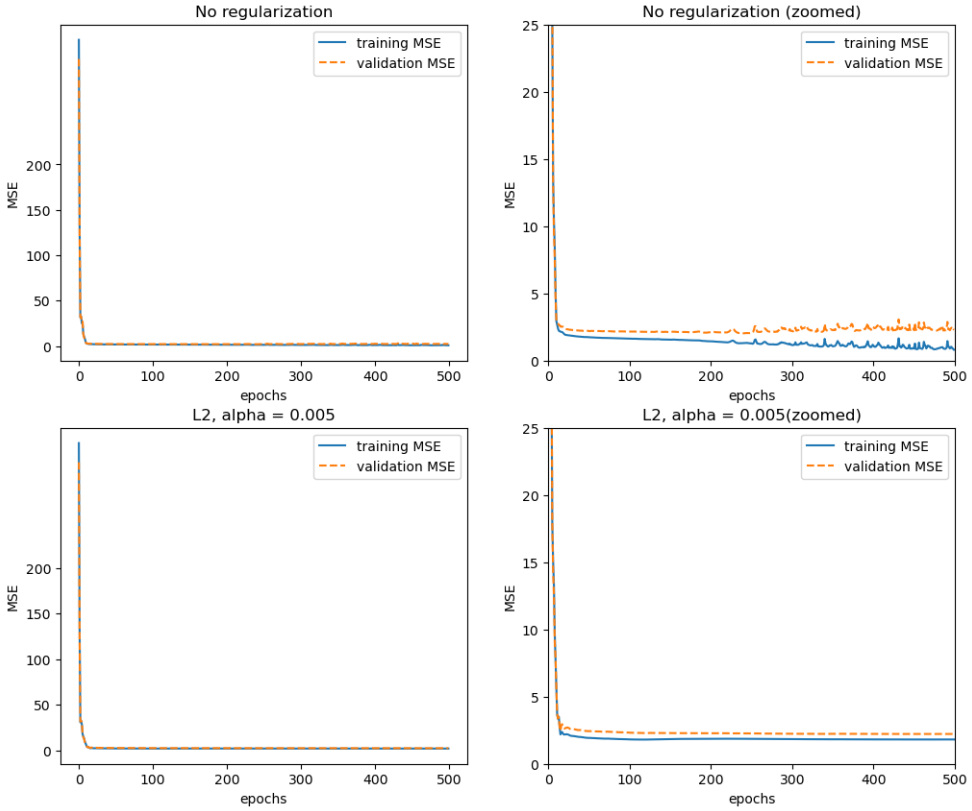

Figure 6: Training and validation errors with and without regularization

net (L1 = L2 = 0.0005), while the second one is obtained using rprop variant (with the same learning step and regularization, but no momentum). As we can observe, Rprop allows a faster convergence and a slight improvement of the performance.

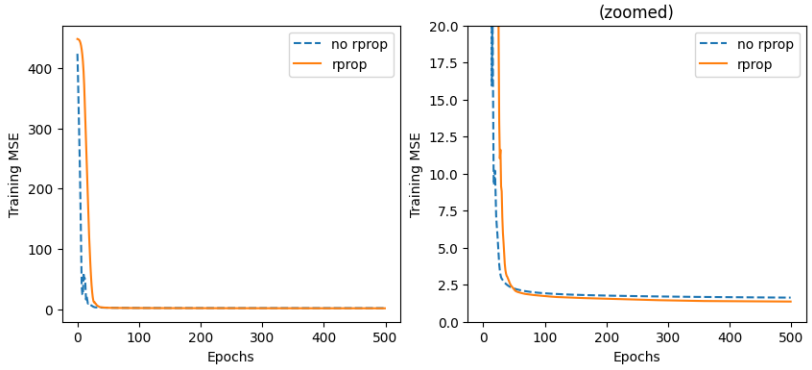

Figure 7: Learning curves with and without rprop

3.2.6 Adaptive gradient

Figure 8 shows the effect of using adaptive gradient on a learning curve of an MLP (topology [40,20,2] and activation function `sigm`) trained with learning step = 0.1, momentum = 0.5 and batch size = 250. The AdaGrad optimization method adjusts the learning rate of individual parameters in a neural network during training. By keeping track of the historical gradient information for each parameter, AdaGrad adjusts the learning rate so that parameters that are frequently updated have a lower learning rate and those that are infrequently updated have a higher learning rate, which possibly results in a smoother learning curve and an improved overall performance of the model, as can be seen from Figure 8.

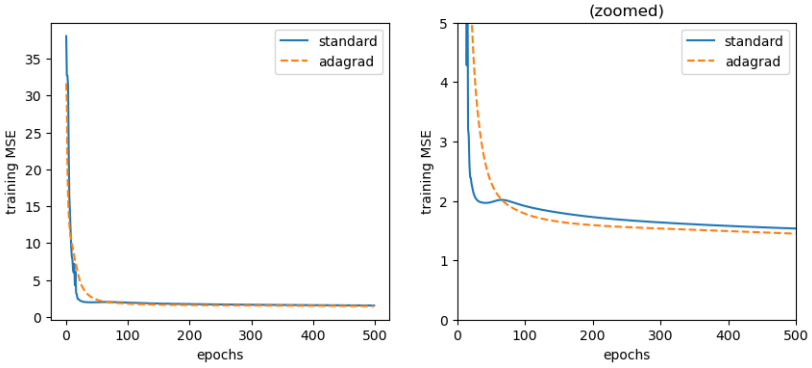

Figure 8: Learning curves with and without adaptive gradient

3.3 Randomized NN

We constructed and evaluated a Randomized Neural Network, which has the same structure as our Multi-Layer Perceptron model but only trains the final layer of the network.

We used the same validation method as the ML cup, which involved using 10% of the observations for testing and performing a 5-fold cross validation on the training set. Additionally, we

used 10% of the observations from the 4 training folds for early stopping.

We tested a neural network with two layers, each containing 100 units and found that the ReLu activation function performed better than tanh and sigmoid. We then used Grid-Search to determine the best combination of hyperparameters for this model.

Hyperparameters	[100, 100, 2]
Step	0.1, **0.3**
L1	0.001, 0.0005, 0.0001, **0**
L2	**0.001**, 0.0005, 0.0001, 0
Weight's mean	**0**, 0.05, 0.1
Weight's S.D.	0.2, **0.1**, 0.05
MEE on Train set	1.71
MEE on validation set	1.71
MEE on test set	1.73

Table 3: Grid search results for the Randomized MLP model, bold characters represent best model parameters.

As can be seen from table 3 the results of the best performing model are worse (but not too far behind) than those of the non-randomized MLP.

3.4 ML cup

3.4.1 Model Selection

For the ML Cup regression task we first split our data keeping 10% for testing and the remaining 90% for training and validation. We then did preliminary testing trying hyperparameters and network structures to identify a viable range of combinations to apply to the gird search. In this phase we focused on finding the best ranges of values for learning step, momentum and regularization on different activation functions and topologies.

We selected two topologies for further testing, [40, 20] and [60, 10], using both ReLu and tanh activation functions. We conducted a comprehensive grid search, using 5-fold cross validation, for the four models. In each iteration of the cross validation (CV) algorithm we used 4 folds (80% of the train validation set) to train the model and 1 fold (20% of the train validation set) to evaluate the model. We also selected 10% of the data from the 4 training folds as internal validation set for early stopping at each iteration of the CV algorithm.

We randomly initialized the weights of the networks using Xavier initialization for the tanh activation function, as suggested by [2], and used the He initialization method for the ReLu activation function as suggested by [3]. We set the max number of epochs to 2500 and we set patience to 20 and tolerance to 1e-5 for the early stopping. Each grid search used elastic net regularization trying combinations of L1 and L2 regularization, momentum and step parameters and whether to use Nesterov's momentum and Mini-Batch. The results for the Grid search are shown in Tables 4 and 5.

The grid search algorithm examined 768 sets of hyperparameters per model, resulting in a total of 3072 trained models per network structure. The search was parallelized and performed on a Mac Book Pro computer with 2,3 GHz 8-Core Intel Core i9. The grid search for each model took 1.5 hours, for a combined total of 6 hours.

The model with the best hyperparameters combination was then trained on the whole train-validation set, with 10% of that data used for early stopping, and tested on the test set.

	[40,20,2]	[60,10,2]
Step	0.008, 0.005, **0.003**, 0.001	0.008, **0.005**, 0.003, 0.001
Momentum	0.5, 0.7, **0.8**	.5, 0.7, **0.8**
L1	**0**, 0.0005, 0.001, 0.005	0, 0.0005, **0.001**, 0.005
L2	0, 0.0005, 0.001, **0.005**	0, **0.0005**, 0.001, 0.005
Nesterov	**True**, False	True, **False**
Mini Batch	**500**, None	**500**, None
Cross ValidatedMEE	1.5097	1.5082

Table 4: Results of the exhaustive Grid Search for the different network topologies with `relu` activation function (bold characters represent best model parameters)

	[40,20,2]	[60,10,2]
Step	0.0075, 0.015, **0.03**, 0.05	0.0075, **0.015**, 0.03, 0.05
Momentum	0.5, **0.7**, 0.8	.5, 0.7, **0.8**
L1	0, **0.0005**, 0.001, 0.005	0, 0.0005, **0.001**, 0.005
L2	0, **0.0005**, 0.001, 0.005	**0**, 0.0005, 0.001, 0.005
Nesterov	**True**, False	True, **False**
Mini Batch	500, **None**	**500**, None
MEE	1.4476	1.4492

Table 5: Results of the exaustive Grid Search for the different network topologies with `tanh` activation function (bold characters represent best model parameters)

3.4.2 Best Model Assessment

We choose as the best model the one with lowest mean MEE on the 5 validation folds. The best model was then retrained on the whole train-validation set and tested on the test set. The results are shown below in Figures 9 and 10.

- **topology**: [40,20,2];
- **Step**: 0.03;
- **Momentum**: 0.7;
- **L1**: 0.0005;
- **L2**: 0.0005.

  - **Nesterov**: True
  - **Mini Batch**: False
- **Training MEE**: 1.3511
- **Validation MEE**: 1.477
- **Test MEE**: 1.3440

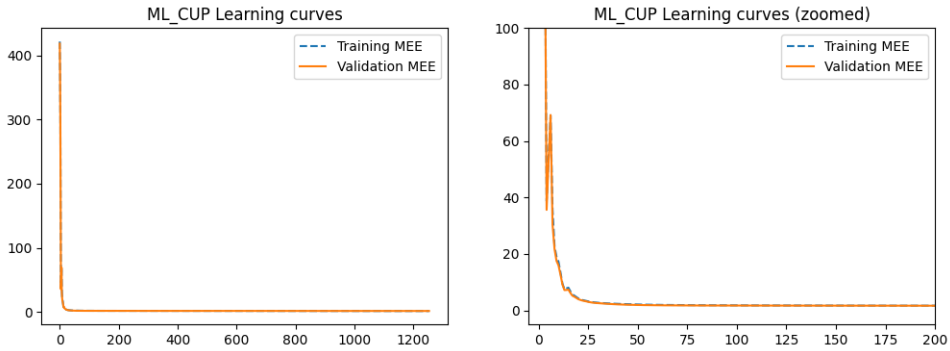

Figure 9: Train and validation curves for final training of best model

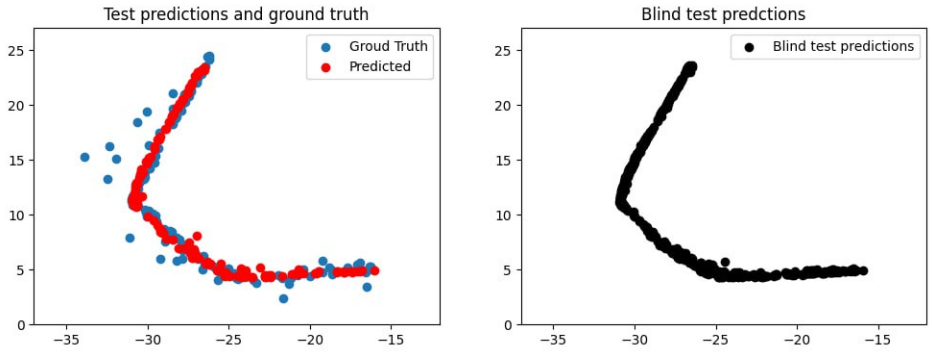

Figure 10: Train and blind test predictions

4 Conclusions

The retrained best model was used to compute the blind test predictions, which were then saved in `RGA_ML-CUP22-TS.csv`. We executed every phase of the project following the standard *modus operandi* proposed during the lectures, also trying to explore multiple variants in order to enrich our model selection process. We are confident that our results on the blind test will not diverge significantly from our assessments.

A Learning curves

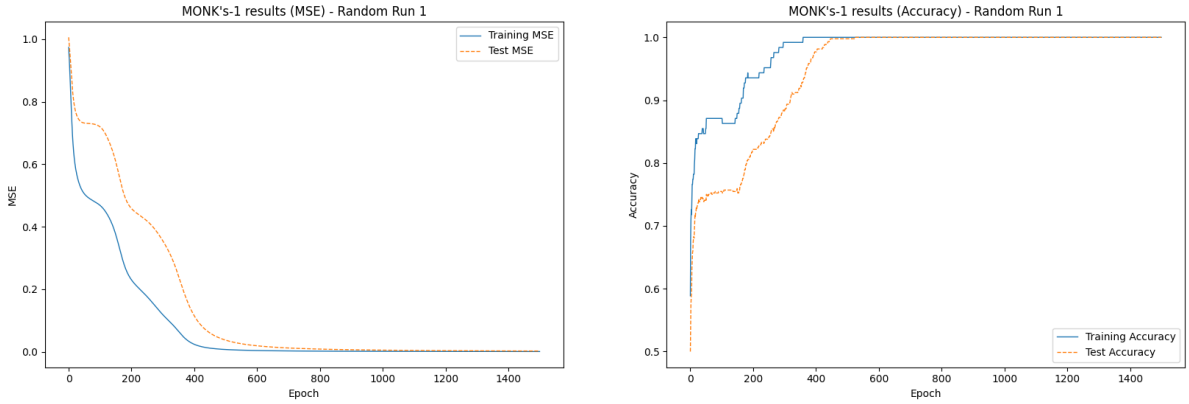

Figure 11: MONK's problem 1 - Learning curves of a random run

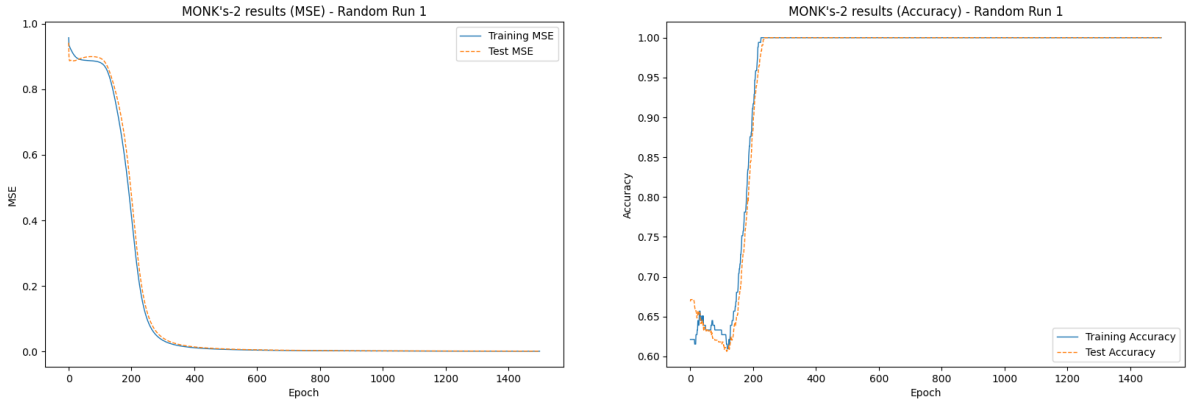

Figure 12: MONK's problem 2 - Learning curves of a random run

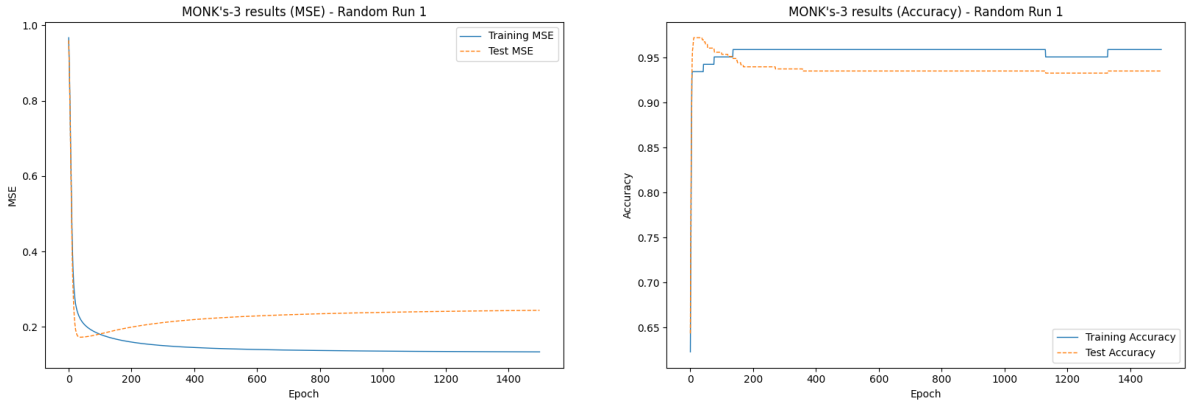

Figure 13: MONK's problem 3 - Learning curves of a random run (not regularized network)

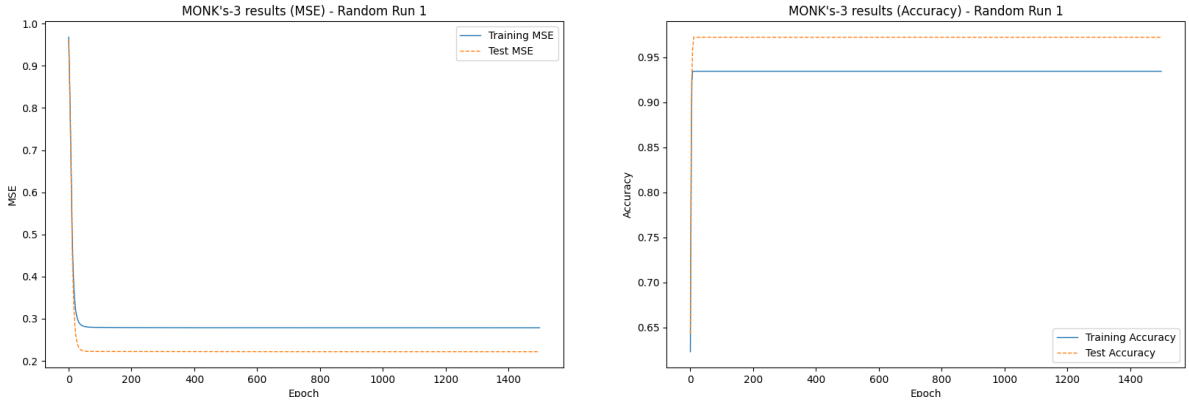

Figure 14: MONK's problem 3 - Learning curves of a random run (regularized network)