# Machine Learning Project report

Cignoni Giacomo (581112) - g.cignoni3@studenti.unipi.it
Marinelli Alberto Roberto (638283) - a.marinelli9@studenti.unipi.it
Melero Cavallo Martina (639305) - m.melerocavallo@studenti.unipi.it

Master Degree Computer Science, AI curriculum
ML course (654AA), Academic Year: 2021-2022
Date: 23/01/2022
Type of project: **A**

## 1 Introduction

Our goal was to create from scratch an Artificial Neural Network able to solve both classification and regression problems. Although it works on different datasets, in our case it was tested on the monk problem [6] and ML course's CUP. The objective of this project is to find the best configurations of our Neural Network for both of these problems. We implemented the building, training, K-fold cross validation and testing of a feed-forward Neural Network (with momentum and L2 regularization) using back-propagation. The algorithm is able to run on on-line, mini-batch and batch configurations and perform grid-search to find the best set of hyper-parameters, i.e. with a low and smooth loss curve. As extensions, we coded the linear learning rate decay, L1 regularization, the possibility to have an arbitrary number of hidden layers and also the option to pursue a randomized Neural Network and ensemble approach.

## 2 Method

### 2.1 Tools and libraries

The software was developed using the Python programming language. The following libraries were used:

- **numpy**, for managing arrays and matrices;
- **joblib**, for implementing multithreading to perform multiple cross validations in parallel;
- **pandas** to parse the `.csv` files of the datasets;
- **matplotlib.pyplot**, for creating the graphs for loss and accuracy;

- **sklearn.metrics**, for computing accuracy for classification tasks.

## 2.2 Software overview

The software has a **modular** approach, in which we kept independent the various parts of the model selection and testing pipeline, separating the grid search code from the cross validation code and the Neural Network object itself. This allowed a more agile development and progressive improvements to each component without the need of comprehensive revisions.

We opted for a partial representation as objects of the internal components of the Neural Network object itself. We chose to implement as internal object only the layers of the Neural Network and not the neurons themselves, a compromise between clarity and ease of use of the code and the efficiency of a Numpy ndarray-based approach.

We decided to keep independent the model selection phase with cross validation, the retrain and internal test phase with the best found hyperparameters, and the final phase in which the model can be used to predict on unseen data (blind test set for CUP). This approach is reached by saving the model weights and/or hyperparameters to be used in the successive phase.

## 2.3 Implementation choices

We developed the class `NeuralNetwork` with which it is possible to instantiate, by selecting different input parameters, various configurations of fully connected Neural Networks.

### 2.3.1 Hyper-parameters overview

- `layer_list`: A list that represents the **topology of the neural network**. The number of neurons is set for each layer, starting from the input, then specifying a variable size of hidden layers and finally the output
- `learning_rate`: **Learning rate** (*eta*)
- `activation_function_hidden`: Name of the **hidden layers activation function**. Can be one of *softmax*, *sigmoid*, *relu*, *leakyrelu*, *tanh* and *identity*
- `activation_function_output`: Name of the **output layer activation function**. The same activation functions of the hidden layers can be chosen
- `loss`: Name of the **Loss Function**. The implemented metrics are *MEE* (Mean Euclidean Error) and *MSE* (Mean Squared Error)

- `momentum`: **Momentum coefficient**
- `lambda_reg`: **Regularization coefficient**
- `reg_type`: Indicates the type of **regularization**, either *L1* (lasso) or *L2* (ridge)
- `rand`: Boolean parameter (by default false) that allows you to use the network as a **Randomized Neural Network**, meaning all weights of hidden layers are fixed and only the readout layer's weights are trained
- `lr_type`: Type of **Learning rate decaying**, it can be *fixed* (by default) or *linear*
- `tau`: Limit step for linear decay learning rate

This implementation can solve both classification and regression problems without specifying a flag, we simply need to choose the most appropriate activation functions.

The main methods of NeuralNetwork are **feedforward** (that given an input pattern, computes the output) and **backpropagation** (that propagates back the error to update each layer's gradient) with these there are related utility functions to ensure their use.

The **train** function allows to start a training by setting **stochastic**, **mini-batch** or **batch** gradient descent a maximum number of epochs. Finally, to use the trained neural network it is possible to execute the **predict** function to compute the outputs on a blind test.

In another class we also decided to implement a parallel version of the **Grid Search** to research the best models. Through this class it is possible to train models configured by lists of values for each input parameter of the class NeuralNetwork. Additional classes, such as **data visualization** and **K-fold cross validation**, are executed with this class to aid in model selection.
After choosing the model, it is possible to use the **test file** to retrain, start a test phase and in the end save the data relating to the tested model. The role of the **ensemble file** is to generate an ensemble model from already trained models and allows it to be used for predicting.
Finally, with the **main** file we can visualize for each dataset the best models' loss and accuracy curves for both Training and Test set.

## 2.4 Pre-processing

We separate input and target columns of the datasets both for Monk and CUP using the pandas library, after having imported the data from the `.csv` files, and save them in separate `.npy` files. For the MONK dataset the **one-hot-encoding** is also applied on both input and target for Training and Test set.

### 2.5 Validation schema

Initially the dataset is split into a Development set (**70**%) and internal Test set (**30**%) for the CUP dataset, whereas MONK was already divided. Afterwards, we performed a grid search to determine the best values for the network's hyper-parameters. For each combination of values a **K-fold cross validation**, with K=5, is performed. Taking advantage of the independence of trials, these cross validations are executed in parallel on different CPUs of our machines.
For each cross validation the loss curves and the mean loss over the validation folds are returned; although the system will recommend the set of hyper-parameters with minimum value of the latter, it's still possible to choose to visualize the results of all the different combinations and save them to a file for later use.

Although a maximum number of epochs can be set to stop the training of the Neural Network, we decided to implement different *stopping criteria*. Depending on the presence or absence of a Validation Set being passed to the `train` function of the Neural Network, either early stopping based on the Validation loss or a stopping criteria based on the absence of training loss shifts are chosen. Both criteria check if the conditions are met over a certain number of consecutive epochs and, in that case, they stop the training.
For **early stopping**, we checked every 5 epochs if the error on the Validation Set was increasing: if it wasn't we saved the current weights of the model; otherwise, if it was, a counter was increased until it reached a certain limit. If the limit is met, then the last weights saved before the validation error started growing are restored [5]. Instead, if no Validation Set is given, the training process stops when there are no more significant **loss changes**, i.e. the learning curve becomes "flat" or asymptotic.

## 3 Experiments

### 3.1 Monk Results

Firstly, **one-hot encoding** was applied to all the datasets, leading each one to have 17 binary input features. Thus, each Neural Network was comprised of 17 input units. For the hidden layers, we chose to have only one with a number of units going from a minimum of 3 to a maximum of 5, which feed directly the single output unit in the final layer. For all the layers the *sigmoid* activation function was used and for the output a threshold of 0.5 is applied (*e.g.* $x < 0.5$ then $x = 0$ otherwise 1). All weights were initialized randomly in a range [-0.7, 0.7].
The chosen loss function was *Mean Square Error* and a maximum of 500 epochs was set. For `MONK1`, we needed to shuffle the Development set to avoid overfitting during the K-fold cross validation. Additionally, regularization was applied only for the `MONK3`

data set, since without it we noticed overfitting, both L1 and L2 were implemented but **Tikhonov**'s (L2) ended up giving better results.
All data sets have been tested with *online*, *mini-batch* (with 4 mini-batches) and *batch* approaches, taking into account the different ranges of values for the hyper-parameters for their results to be comparable. `MONK1`, `MONK2` and `MONK3` without regularization performed better using **online** gradient descent, instead the results improved for `MONK3` with regularization using **batch** gradient descent.

| Task | #Units | eta | alpha | lambda | MSE(TR/TS) | Accuracy(TR/TS) |
|---|---|---|---|---|---|---|
| MONK1 | 4 | 0.6 | 0.7 | 0.0 | 3.58e-4/1.01e-3 | 100%/100% |
| MONK2 | 3 | 0.6 | 0.7 | 0.0 | 2.37e-4/2.77e-4 | 100%/100% |
| MONK3 | 4 | 0.4 | 0.5 | 0.0 | 8.97e-3/3.59e-2 | 99%/96% |
| MONK3+reg | 5 | 0.9 | 0.9 | 0.001 L2 | 2.79e-2/2.85e-2 | 97%/96% |

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

The learning and accuracy curves are shown respectively for `MONK1`, `MONK2` and `MONK3` in Figure 2, 3 and 4.

## 3.2 Cup Results

Initially, as already reported in the Validation Schema section 2.5 of this report, we used the 70% of the ML-CUP dataset as Development set to perform a grid-search on the hyper-parameters values to select the best model, through **5-fold cross validation**. The remaining 30% of the dataset was reserved for the Internal Test set.
All **weights** for the Neural Network were initialized randomly in a range [-0.7, 0.7].
Before starting a fine grid search on all possible hyperparameters, we have tested through **coarse grid** searches different topologies to discard cases where there was underfitting, usually models with too few units per layer, and without too many units, which were too heavy performance-wise without significant improvements. During these tests we also detected ranges of hyperparameters that avoided saturation of the network caused by overflows, and that we refined with a **fine grid search** successively.

We did not have doubts about the activation functions to use; to preserve the capability of the network to output negative continuous values we chose to use *leaky RELU* (for the hidden layers) and *identity* (for the output layer) because they do not limit the outputs to only positive or restricted ranges.

According to a scientific paper that analyzes **mini-batch** approach for Neural Networks, the number of patterns for each batch is advised to be in the range of 2 to 32 to provide improved generalization performance. After performing a coarse grid search with online and batch approaches, we confirmed that they didn't provide good results (e.g. best Validation loss of batch with a stable learning curve was 0.3213816997), so

we chose to explore mini-batch sizes mainly in this limited range and tune the other hyper-parameters accordingly[3].

We also chose to utilize the *early stopping* method based on the validation set loss for the cross validation phase, as we already have a validation fold at our disposal[5], and the stopping method based on asymptotic loss on the training set for the retrain. The upper bound for the number of epochs was set to *800*.

Additional expansions of the Neural Network tried during grid search were:

- **L1** regularization;
- **Randomized Neural Network**, meaning all weights of hidden layers are fixed and only the readout layer's weights are trained;
- **Variable Learning rate**, specifically with linear decay, since we used mini-batches;
- **Multiple hidden layers**;

Although the use of a Randomized Neural Network gave generally discrete results, they were not the best possible ones with respect to the ones without randomness, so we didn't insert it as a possibility in the fine grid search.

In Table 2 the hyper-parameters explored during the fine grid search are shown.

| **Numerical hyper-parameter** | **Range [min, max]** |
|---|---|
| learning_rate | [1e-3, 0.1] |
| momentum | [0.0 , 0.7] |
| lambda_reg | [0.0 , 0.01] |
| tau | [2000, 10000] |
| **Categorical hyper-parameter** | **Possible options** |
| layer_list | [10, 8, 5, 3, 2], [10, 10, 5, 2], [10, 50, 10, 2], [10, 20, 10, 2], [10, 30, 10, 2], [10, 15, 6, 2] |
| reg_type | 'l1', 'l2' |
| lr_type | 'fixed', 'linear_decay' |
| Batch size | 16, 32, 258 |
| **Fixed hyper-parameter** | **Used value** |
| activation_function_hidden: | 'leakyrelu' |
| activation_function_output: | 'identity' |
| loss | MEE |
| rand | False |
| Early stopping | True |
| Max epochs | [800] |
| Shuffle | False |

Table 2: Ranges, lists of options and fixed values of the tried hyper-parameters in grid search. The role of each hyper-parameter is described in section 2.3.1

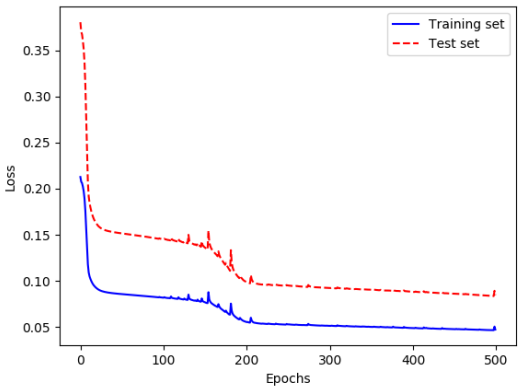
.

The six best models resulting from the grid search were chosen among those with the lowest value of Mean Euclidean Error and most stable learning curves on the validation phase in cross validation; these are reported in Table 3 and their retrain curves and hyper-parameters in Figures 1 and 5.

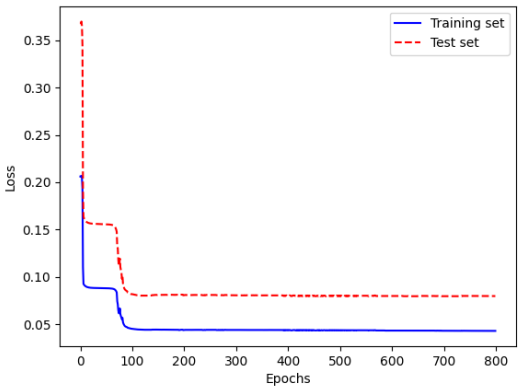
| **Model** | **Validation MEE** | **Development set MEE** | **Internal test set MEE** |
|---|---|---|---|
| Model 0 | 0.1616403085 | 0.0475007039 | 0.0848713805 |
| Model 1 | 0.2525338065 | 0.0412504950 | 0.0746889404 |
| Model 2 | 0.1489656149 | 0.0490893407 | 0.0873228723 |
| Model 3 | 0.1523888340 | 0.0429048350 | 0.0797160488 |
| Model 4 | 0.1147401246 | 0.0411089590 | 0.0726483563 |
| Model 5 | 0.1269484589 | 0.0552610428 | 0.0992050683 |
| Ensemble (models: 0, 3, 4, 5) | 0.1389294315 | 0.0466938852 | 0.0779390519 |

Table 3: Mean Euclidean Error of the Validation set (regarding cross validation), Development set (loss during final retrain) and Internal Test set of the 6 best models and the average for the ensemble only on Models 0, 3, 4 and 5.

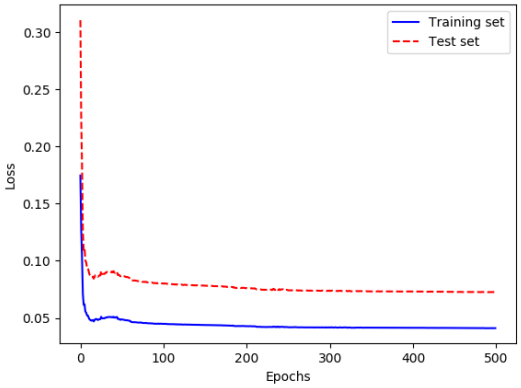
For the final model we opted for an **ensemble** approach [4]: we selected the best four models (0, 3, 4, 5 as shown in Fig.1 and Tab.3) among the previously best six, such that they had sufficiently distinct characteristics (layers, eta, alpha, and others). Afterwards, we **retrained** all the models on the whole Development set to make better use of the dataset at our disposal and computed the predictions on the Internal Test set. The average of the predictions of the models selected for the ensemble model made up the final predictions. The MEE values for Validation and Development shown in Table 3 for the ensemble model are computed as the average of the MEE on the respective sets of the constituent models (they are upper bounds of the true loss ensemble values). Instead for the Internal Test set we computed the MEE on the new averaged predicted outputs.



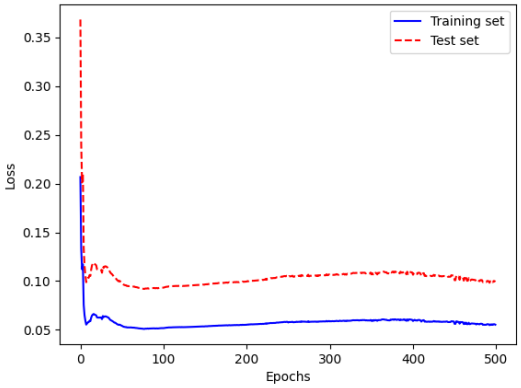

**Model 0** - layers: [10, 10, 5, 2], lr: 0.01, momentum: 0.7, lambda: 1e-05, reg_type: l2, lr_type: fixed, batch_size: 258




**Model 3** - layers: [10, 10, 5, 2], lr: 0.01, momentum: 0.7, lambda: 0.001, reg_type: l2, lr_type: fixed, batch_size: 32




**Model 4** - layers: [10, 15, 6, 2], lr: 0.001, momentum: 0.0, lambda: 0.001, reg_type: l2, lr_type: fixed, batch_size: 16




**Model 5** - layers: [10, 10, 5, 2], lr: 0.005, momentum: 0.0, lambda: 0.001, reg_type: l2, lr_type: linear_decay, tau: 5000, batch_size: 16

Figure 1: Loss learning curves for the Cup dataset, relative to final retrain and test for the four models used in the ensemble model. For each model are specified the significant hyper-parameters.

# 4 Conclusion

We used the ensemble model to compute the predictions on the blind test set, they are stored in file `.csv`. We are optimistic that the Mean Euclidean Error computed on the Internal Test set will be a good approximation of how it will behave on other unseen data, since the examples in the Internal Test set were not part of those used in the model selection phase. Thus we expect that the loss on the blind test set will be around 0.0779390519 (in terms of MEE).

# Acknowledgments

This experience was very useful because it let us deeply understand the way neural networks function. Implementing the theoretical aspects of a Machine Learning model made us understand the importance of its different components and how they interact. Building the whole system with our own hands was an all-around stimulating experience because it allowed us to approach problem solving with a creative attitude.

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

# References

[1] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. MIT Press, 2016.

[2] Simon Haykin. *The Neural Networks a comprehensive foundation*. Prentice-Hall, 3 edition, 2008.

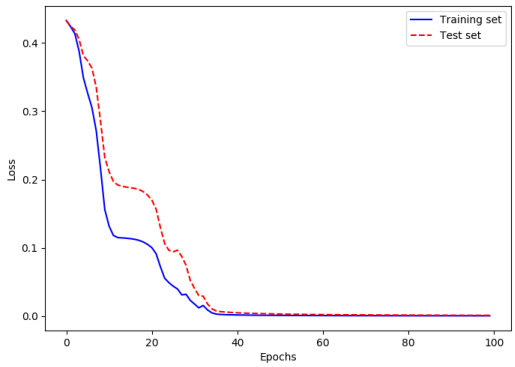
[3] Dominic Masters and Carlo Luschi. Revisiting small batch training for deep neural networks. *CoRR*, abs/1804.07612, 2018.

[4] David Opitz and Richard Maclin. Popular ensemble methods: An empirical study. 11, 12 1999.

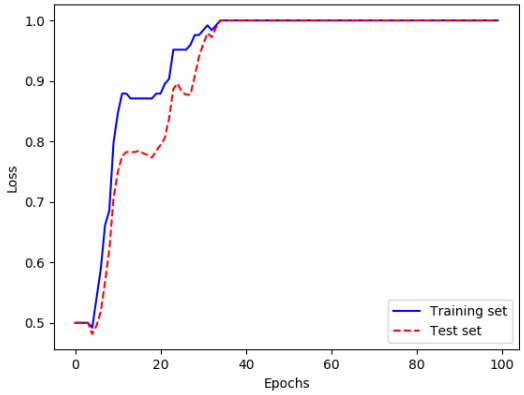
[5] Lutz Prechelt. Early stopping - but when? 03 2000.

[6] E. Bloedron I. Bratko B. Cestnik J. Cheng K. De Jong S. Dzeroski R. Hamann K. Kaufman S. Keller I. Kononenko J. Kreuziger R.S. Michalski T. Mitchell P. Paphowicz B. Roger H. Vafaie W. Van de Velde W. Wenzel J. Wnek S.Thurn, J. Bala and J. Zhang. The monk's problems: A performance comparison of different learning algorithms. *Technical Report CMU-CS-91-197, Carnegie Mellon University, Computer Science Department, Pittsburgh, PA*, 1991.

# Appendix

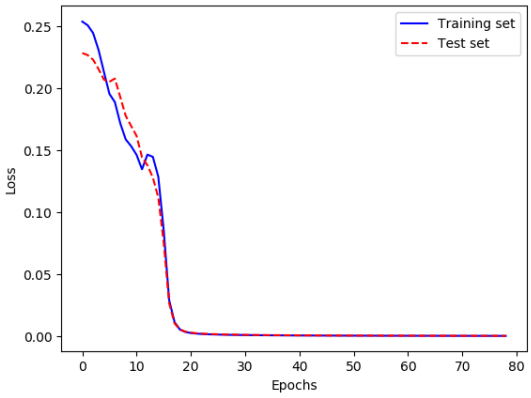



Monk1 MSE loss

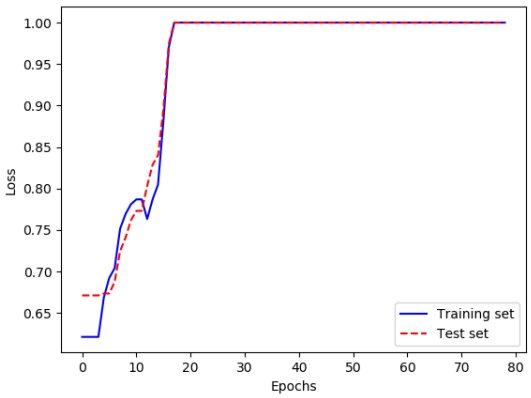



Monk1 accuracy

Figure 2: Loss and accuracy curves for Monk1 relative to final retrain and test.

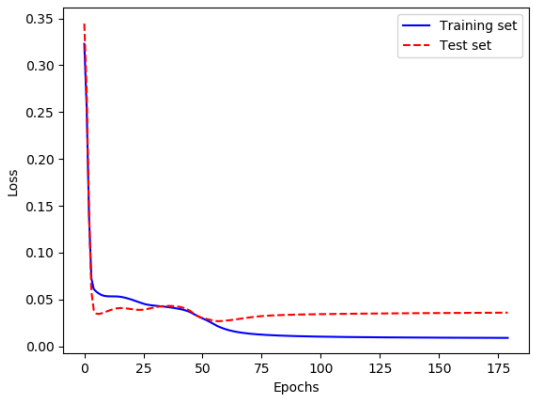



Monk2 MSE loss

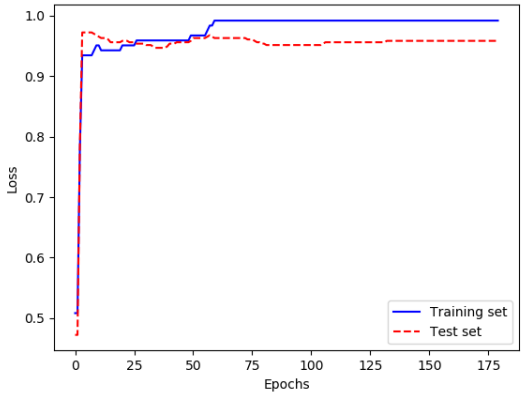



Monk2 accuracy

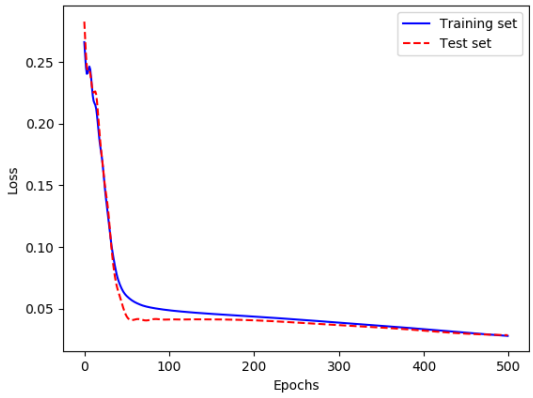
Figure 3: Loss and accuracy curves for Monk2, relative to final retrain and test.



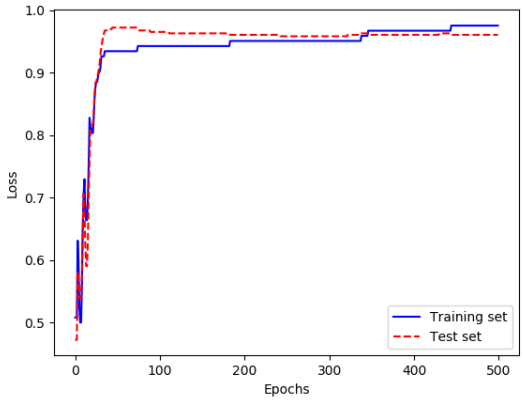

Monk3 MSE loss




Monk3 accuracy

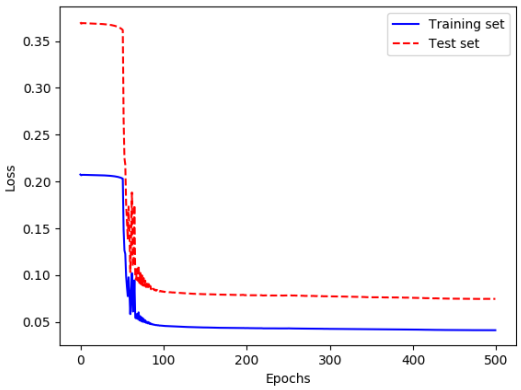



Monk3 + reg MSE loss

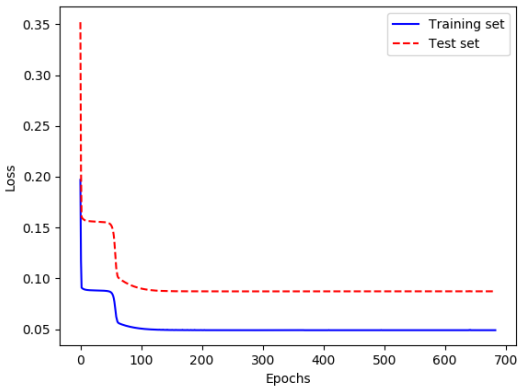



Monk3 + reg accuracy

Figure 4: Loss and accuracy curves for Monk3 and Monk3 + reg relative to final retrain and test.




**Model 1** - layers: [10, 10, 5, 2], lr: 0.01, momentum: 0.7, lambda: 1e-05, reg_type: l2, lr_type: fixed, batch_size: 258




**Model 2** - layers: [10, 10, 5, 2], lr: 0.01, momentum: 0.7, lambda: 0.001, reg_type: l2, lr_type: fixed, batch_size: 32

Figure 5: Loss learning curves for Cup relative to final retrain and test of Model 1 and Model 2. For each model are specified the significant hyperparameters.