Machine Learning Project Report

Salvatore Correnti, Computer Science (Artificial Intelligence)
Alberto L'Episcopo, Computer Science
Gaetano Nicassio, Computer Science (Artificial Intelligence)
s.correnti@studenti.unipi.it
a.lepiscopo1@studenti.unipi.it
g.nicassio1@studenti.unipi.it
ML course (654AA), Academic Year: 2022-2023
Date: January 2, 2023
Type of project: **A** *(Python)*

Abstract

The aim of this report is to illustrate the implementation and testing of a framework based on NumPy for creation and usage of basic Neural Networks, and tested through usage of the MONK and the ML-CUP22 datasets. Model selection is implemented by using 3-fold validation technique on several coarse-grained Grid Searches, followed by a finer one with 4-fold cross validation.

1 Introduction

The project consists in the implementation and testing from scratch of a framework for building Artificial Neural Networks with backpropagation algorithm for both classification and regression problems, providing the essential building blocks for defining the structure of a network, loss functions, optimizers, metrics to monitor performances and callbacks. Our objective was to find the best model for both MONK [2] and ML-CUP22 tasks by using 3-fold validation technique with several first coarse-grained Grid Searches to figure out good intervals for hyperparameters, followed by a finer-grained Grid Search on these intervals with 4-fold cross validation. The most performing model after this search is then used to predict outputs for the blind test set.
For installation and usage see the file `README.md` in the project package.

2 Method

2.1 Code

The framework has been implemented in Python using `numpy` (https://numpy.org/) for the computational part, some minor utilities from `sklearn` and `pandas` and the `pickle`

library for serialization; `tensorflow` is used in several tests for comparing our results with an industry-standard tool. Finally, the `typer` (https://typer.tiangolo.com/) library is used to build a command-line interface. For our implementation, we took inspiration from both `Tensorflow` and `PyTorch` frameworks, and in particular from `keras` API for its clarity, modularity, ease of use and for faster prototyping and comparisons (https://keras.io/api/).

Implementation is contained in the `core` package. Neural networks are implemented by the `Model` class, which contains a sequence of `Layer` objects and exposes the `compile()`, `train()` and `predict()` methods. The `Layer` class is specialized by `Dense`, `Linear` and activation layers. The `Dense` class implements a fully-connected layer, while `Linear` implements a fully-connected layer with identity as (implicit) activation. The optimizer provided is the standard `SGD` with momentum, while `L1`, `L2` and `L1L2` techniques are available for regularization. Provided losses are `MSE` and `Cross Entropy`. Training can be done either in batch, online or minibatch mode by using the `DataLoader` class. Weights initialization is available with Normal and Uniform distribution. Provided activation functions are `tanh`, `sigmoid`, `relu`, `softmax`.

The framework provides a system of metrics and callbacks for customizing behaviour: in particular, metrics shall be provided when calling `compile()`, while callbacks can implement a series of methods that are called before and after each training epoch/batch and each validation step. Examples of callbacks include `EarlyStopping` and loggers, while available metrics are `Mean Squared Error`, `Mean Absolute Error`, `Mean Euclidean Error`, `Root MSE`, `Binary` and `Categorical Accuracy`.

Finally, the base classes `BaseSearch` and `Validator` define the base structure for search strategies and validation techniques and are sub-classed by `GridSearch`, `Holdout` and `KFold`. Search can be done in parallel by using up to all the available cores.

2.2 Validation Schema

At first, we divided the whole CUP dataset into a **Development Set** (90%) and an **Internal Test Set** (10%) to be used for testing the final model. We conducted some preliminary trials with several architectures and hyperparameters for excluding irrelevant and under-performing ones.
After that, we performed several exhaustive **coarse-grained Grid Searches** with large steps and **3-fold cross validation** technique for finding good intervals for the hyperparameters through observation of the average performance of each one in this first Grid Search. In particular, we first ranked all the configurations according to their average validation MEE over all the folds and then we calculated a score for each hyperparameter as an heuristic for choosing which intervals to consider.

After that, we performed an exhaustive **finer-grained Grid Search** on the selected intervals or values with a **4-fold cross validation technique** and we ranked the configurations accordingly to their average Validation MEE over all the folds.
Finally, we selected the best model as our final model, trained it on the whole development set and used it for predicting outputs for the blind test set.

2.3 Preliminary Trials

Monks started working with 100% accuracy immediately after adopting one-hot encoding for representing the inputs, we had just to pay some attention to regularization for the third dataset.
For CUP dataset, instead, we needed to perform several preliminary trials to figure out the best intervals for the hyperparameters and for discarding under-performing values. We tested several orders of magnitude for initial learning rates, learning rate decay, weights initialization, (mini)batch size and maximum number of epochs. The results are later explained in the Experiments section.

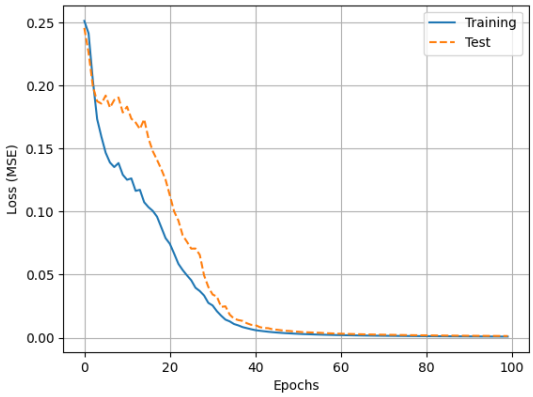
3 Experiments

3.1 Monk Results

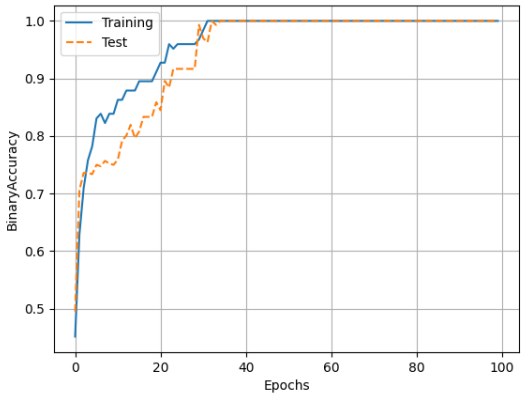
For all MONK problems, we adopted one-hot encoding for representing the inputs, so we used neural networks with 17 input units. We used 1 hidden layer with 4 units for the first two problems and 5 for the last one. For what concerns activation functions, we used `tanh` for the hidden layer, and `sigmoid` for the output layer to get outputs in the $[0, 1]$ range.
We chose `MSE` as the loss function over the "raw" output of the `sigmoid` in order to have differentiability, and instead for predictions every input is classified as 1 if the output of the `sigmoid` is ≥ 0.5 and 0 otherwise.
We used mini-batch gradient descent with `batch_size` $= 4$ for the first Monk problem and 2 for the others, and L2 regularization for Monk3.
Our choice of hyper-parameters is shown in table 1 below, along with the performance averaged over 5 different runs to avoid the bias due to weights initialization.
For MONK 3, we observed that without regularization there is overfitting in all the 5 runs, so we needed to introduce a regularization penalty (L2) of 10^{-4}.

3.2 Cup results

At the beginning, we split the CUP dataset in **Development Set** (90% of the dataset, 1342 records) and in **Internal Test Set** (10% of the dataset, 150 records). The aim of the first dataset was to be used for subsequent training-validation splits for model

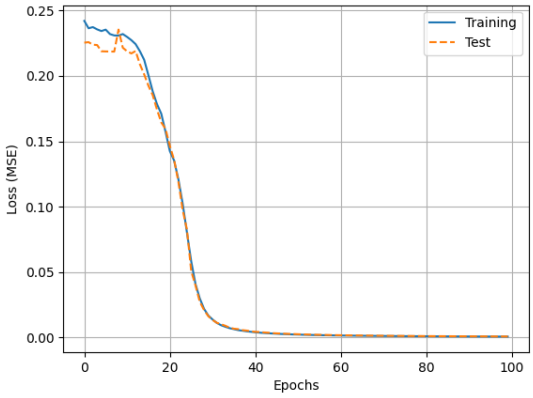


(a) Loss

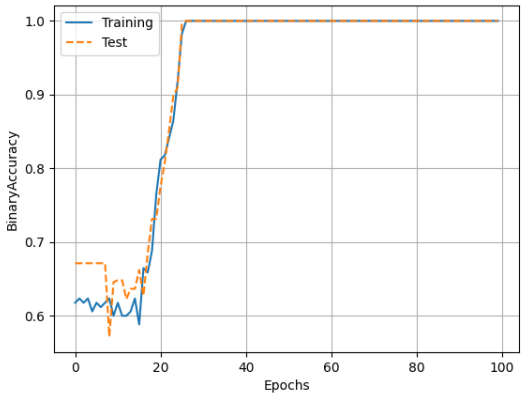


(b) Accuracy

Figure 1: Loss and accuracy of MONK 1

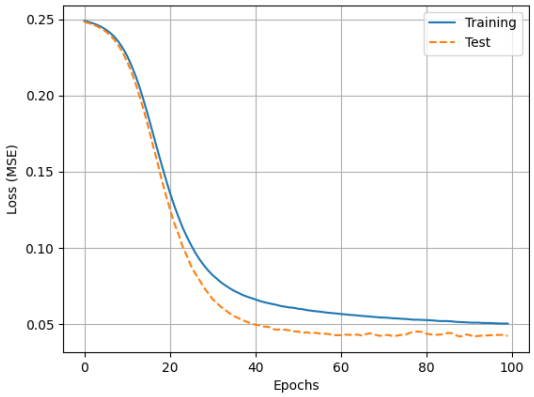


(a) Loss

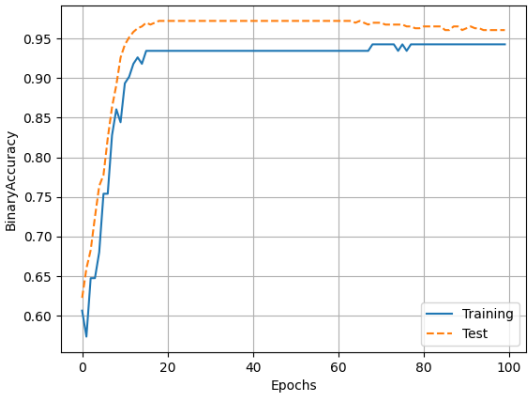


(b) Accuracy

Figure 2: Loss and accuracy of MONK 2

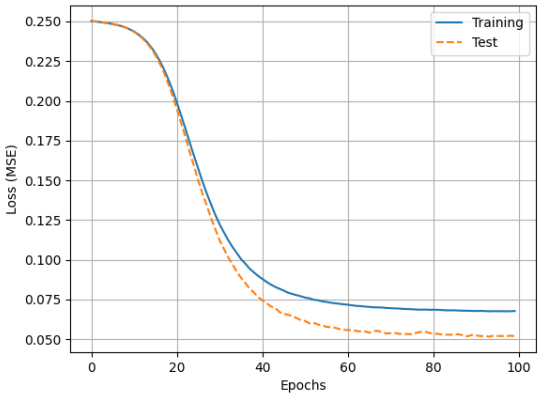


(a) Loss

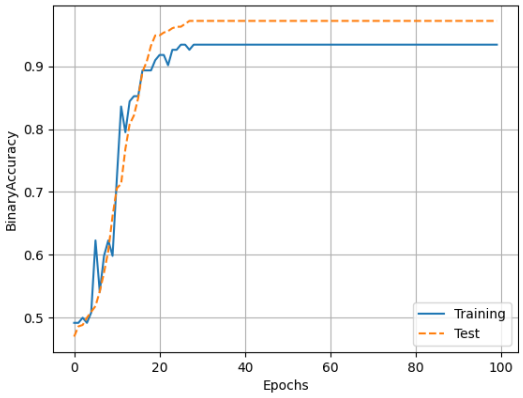


(b) Accuracy

Figure 3: Loss and accuracy of MONK 3 without regularization.



(a) Loss



(b) Accuracy

Figure 4: Loss and accuracy of MONK 3 with L2 regularization.

Task	η	λ	MSE (train/test)	Accuracy (train/test)
MONK 1	0.5	0	0.00084/0.00128	100%/100%
MONK 2	0.2	0	0.00062/0.00073	100%/100%
MONK 3 (no regularization)	0.02	0	0.05048/0.04236	94.26%/96.06%
MONK 3 (regularization)	0.02	10^{-4}	0.06781/0.05215	93.44%/97.22%

Table 1: Performance results on MONK datasets

selection and training, while the second one was used for testing. After that, we ran several coarse-grained Grid Searches using 3-fold cross validation on the Development Set and finally a fine-grained Grid Search over refined intervals for the hyperparameters using 4-fold cross validation on the Development Set.

3.2.1 Preliminary trials

We initially ran several experiments using `EarlyStopping` [1] for figuring out the order of magnitude of the maximum number of epochs for which there is still significant improvement; our `EarlyStopping` was configured such that it considered 10^{-4} or 10^{-5} to be minimum difference on Validation MEE to consider it an improvement. They revealed us that in many cases the improvement on training set was at most of order of $10^{-}4$ between epoch 500 and 1000, while the loss and MEE on the validation set either remained stable or started to grow. We then decided to put 500 as the maximum number of epochs, also to limit possible overfitting since we wanted to use k-fold cross-validation for our model selection. We similarly ran other test to discard some parameters and speed up the grid search.
After that, we discarded learning rates of order $\geq 10^{-}1$ or $\leq 10^{-}4$, since they showed worse results than $10^{-}2$ and $10^{-}3$, and we selected 256 aas minibatch size, since it appeared to us the best compromise between the capability of minibatch algorithm to reach better results than batch ones, the speed of training phases and the stability and smoothness of the learning curves. Similarly, for L2 regularization lambda we have discarded values of order $\geq 10^{-3}$ or $\leq 10^{-8}$.
We also tested different network architectures, and we noticed that the ones with two hidden layers performed better then the ones with a single hidden layer. Moreover, architectures with three hidden layers did not perform better than the former, so we decided to focus on architectures with 2 hidden layers.
Finally, we discarded small ranges for weights initialization (e.g. uniform distribution in $[-0.1, 0.1]$), since they performed worse than higher ones, and we fixed uniform distribution in the range $[-0.7, 0.7]$ for weights initialization.

3.2.2 Grid search

At first, we conducted a coarse-grained Grid Search over 1296 possible configurations with large steps for hyperparameters whose purpose was to figure out best intervals or values for the hyperparameters inside the possible values or orders of magnitude we selected after preliminary trials. This first Grid Search uses **3-fold cross validation** on the Development Set for evaluating configurations and was performed exhaustively among the following hyperparameters:

- **Architecture (hidden units)**: $32-16$, $16-16$, $16-8$;
- **Activation functions (for hidden layers)**: `tanh-tanh`, `tanh-sigmoid`, `sigmoid-tanh`, `sigmoid-sigmoid`;
- **(Initial) learning rate (η_0)**: 0.01, 0.005, 0.001;
- **Momentum**: 0, 0.4, 0.8;
- **L2 Regularization Lambda**: 10^{-4}, 10^{-5}, 10^{-6}, 10^{-7};
- **Learning Rate Decay**: no decay, linear decay up to $0.1 \cdot \eta_0$, linear decay up to $0.01 \cdot \eta_0$;

where with "linear decay up to η_1" we mean that if the number of epochs is N (in our case $N = 500$), at the epoch $0 \leq x \leq N$ (counting from 0) we had a learning rate of $\eta(x) := \left(1 - \frac{x}{N}\right)\eta_0 + \frac{x}{N}\eta_1$. We have used a "relative" value of $\frac{\eta_0}{\eta_1}$ rather than a fixed constant η_1 in order to be sure that it always holds $\eta_1 \leq \eta_0$ (otherwise we could have got $\eta_0 = \eta_1 = 10^{-3}$ for instance).
We decided to use k-fold cross validation right from the beginning since we wanted to get less biased results with respect to both weights initialization and training-validation splits.
After that first search, we ranked all the configurations according to their average validation MEE over all the folds and we assigned a score to each hyperparameter calculated as following: if the total number of configurations is N and a given value x for a given hyperparameter H appears in positions $i_1 \leq ... \leq i_k \leq N$, its score is: $\sum_{j=1}^{k}(N - i_j)$. We used this score as an indicative heuristics for choosing activation functions and for fine-tuning initial learning rate and momentum. For L2 lambda and learning rate decay, there was no clear "winner", so we ran another coarse-grained Grid Search with approximately the already selected values for the above-mentioned hyperparameters and we fine-tuned L2 lambda, while for decay having 10^{-1} or 10^{-2} seemed ininfluent, so we kept only the first one.
After that, we performed another exhaustive finer-grained Grid Search with **4-fold cross validation** over all the 2880 combinations of the following hyperparameters:

- **Architecture (hidden units)**: $\{32-16, 16-16, 16-8\}$;
- **Activation functions (for hidden layers)**: tanh-tanh, tanh-sigmoid;
- **(Initial) learning rate (η_0)**: $\{0.01, 0.009, 0.008, 0.007, 0.006, 0.005\}$;
- **Momentum**: $\{0.4, 0.5, 0.6, 0.7, 0.8\}$;
- **L2 Regularization Lambda**: $\{10^-5, 8 \cdot 10^-6, 6 \cdot 10^-6, 4 \cdot 10^-6, 2 \cdot 10^-6, 10^-6, 8 \cdot 10^-7, 6 \cdot 10^-7\}$;
- **Learning Rate Decay**: no decay, linear decay up to $0.1 \cdot \eta_0$.

As before, we ranked all the configurations according to their average Validation MEE over all the folds and we finally selected the best model as our final model, trained it on the whole development set and used it for predicting outputs for the blind test set.
Table 2 shows some of the heuristics values after the coarse-grained Grid Searches.

Hyperparameter	Values
Activation	`tanh-tanh` : 247924, `tanh-sigmoid` : 225057
Activation (2nd row)	`sigmoid-tanh` : 198399, `sigmoid-sigmoid` : 169076
Learning rate	0.01 : 379687, 0.005 : 324272, 0.001 : 136497
Momentum	0.8 : 373121, 0.4 : 261535, 0.0 : 205800
Lambda (1st search)	10^{-4} : 151572, 10^{-5} : 228108, 10^{-6} : 230144, 10^{-7} : 230632
Lambda (2nd search)	10^{-5} : 6101, $5 \cdot 10^{-6}$: 5812, 10^{-6} : 5486, $5 \cdot 10^{-7}$: 5985, 10^{-7} : 5536
Decay	no decay: 330805, linear 0.01: 244885, linear 0.1: 264766

Table 2: Heuristic values for some hyperparameters.

3.2.3 Grid Search Results

Table 3 reports the hyperparameters values and the average training, validation and test errors (MEE) over all the 4 folds of the 8 most performant configurations after the fine-grained Grid Search.

3.2.4 Computing time and hardware

The two coarse-grained searches cumulatively took 22 minutes to complete, while the second one took 57 minutes using an Intel Core i7-10870H @ 2.00 GHz (16 logical CPUs). We used the `joblib` library for parallelizing the search by using a pool of 16 worker processes. The average time requested for a training cycle of a model during the grid search is between 3 and 5 seconds for the three different architectures, which is slightly higher than the time needed for a full training of the final chosen model, which is about 3.2 seconds. For Monk problems, the time required is about 1.2 seconds for the first problem and 2.1 for the other two on the same hardware as above, with this difference substantially determined by the different minibatch sizes that we used.

Architecture	Activation	Learning Rate	Momentum	Lambda	Decay	Error TR	Error VL	Error TS
$32 - 16$	`tanh-tanh`	0.005	0.8	$6 \cdot 10^{-}6$	linear 0.1	1.2680	1.4246	1.3857
$32 - 16$	`tanh-tanh`	0.009	0.8	$6 \cdot 10^{-}7$	linear 0.1	1.2338	1.4268	1.4836
$16 - 16$	`tanh-tanh`	0.008	0.7	$2 \cdot 10^{-}6$	no decay	1.2490	1.4269	1.3950
$32 - 16$	`tanh-tanh`	0.006	0.8	$6 \cdot 10^{-}6$	linear 0.1	1.2289	1.4276	1.4281
$16 - 16$	`tanh-tanh`	0.008	0.8	10^{-5}	linear 0.1	1.2852	1.4301	1.3936
$16 - 16$	`tanh-tanh`	0.009	0.7	$4 \cdot 10^{-6}$	linear 0.1	1.2829	1.4304	1.4199
$32 - 16$	`tanh-tanh`	0.009	0.8	$6 \cdot 10^{-6}$	linear 0.1	1.2134	1.4305	1.4460
$32 - 16$	`tanh-tanh`	0.006	0.6	$8 \cdot 10^{-6}$	no decay	1.2400	1.4318	1.4704

Table 3: Hyperparameters and average training, validation and test error (MEE) of the 8 most performant configurations after last grid search

3.2.5 Chosen model and final discussion

We finally chose the model that performed best on the validation set, which has the following hyperparameters and training and test errors (MEE) after a full training on the whole development set, which took 3.2 seconds.

Architecture	Activation	Learning Rate	Momentum	Lambda	Decay	Error TR	Error TS
$32 - 16$	`tanh`	0.005	0.8	$6 \cdot 10^{-}6$	linear 0.1	1.2369	1.4401

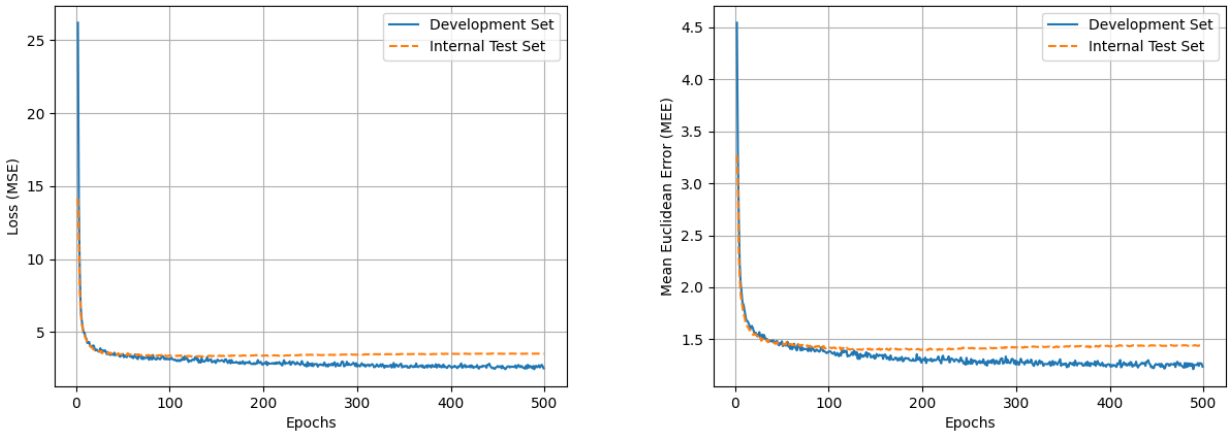
Table 4: Hyperparameters and training and test error (MEE) of the chosen model after a full training on the whole development set

We can see the learning curve for loss (MSE) and MEE in fig. 5 below.
We achieved a MEE of 1.4401 on our internal test set, and we estimate the performance on the blind test set as being near to that value. We have observed that for almost all the cases that we have analyzed both in Grid Searches and in preliminary experiments the validation error is in the range $[1.3, 1.6]$ for all the models with two hidden layers we have tried and for several ones with three hidden layers, regardless of the activation functions and the other hyperparameters. We obtained similar results also with equivalent `keras` models.

4 Conclusions

4.1 Final Remarks

This project gave us the opportunity to have an hands-on experience both in the implementation of a Neural Network and in experimenting model selection and training. We had the possibility to draw inspiration from industry-standard tools and to make design and implementation comparisons between them for our own implementation. We had the possibility to better understand the main theoretical contents of the course through implementation and debugging, for example of the backpropagation algorithm. Similarly,



(a) Loss

(b) MEE

Figure 5: Learning curves for CUP network

we learnt a lot on how to do model selection and validation from our first attempts to our final version, comparing different strategies and reflecting on how to use techniques like k-fold cross validation and searches results to identify the most promising intervals for hyperparameters. Overall it was a great experience, and we learnt a lot from it.

4.2 Blind Test Set Results

Blind test set results are contained in the file `TheBishops_ML-CUP22-TS.csv` in the `results` folder of the project package. Our team name is **The Bishops**.

4.3 Agreement

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

References

[1] Simon S. Haykin. *Neural networks and learning machines*. Pearson Education, Upper Saddle River, NJ, Third edition, 2009.

[2] Sebastian Thrun. The MONK's problems. A Performance Comparison of Different Learning Algorithms, CMU-CS-91-197, Sch. 1991.