Comparison among different machine learning models

University of Pisa

Pasquale Esposito 649153, Master programme in Computer Science,
curriculum Big Data Technologies,
p.esposito8@studenti.unipi.it

Innocenzo Fulginiti 594051, Master programme in Computer Science,
curriculum Software: Programming, Principles, and Technologies
i.fulginiti@studenti.unipi.it

Carlo Tosoni 644824, Master programme in Computer Science,
curriculum Big Data Technologies,
c.tosoni@studenti.unipi.it

654AA Machine Learning, AY 2022/2023

20 January 2023

Project Type: B

Abstract

In this project we evaluated the performance of various models, in particular neural networks, SVM, KRR, KNN, decision trees and random forests, in order to choose the most performing model. The validation technique chosen is the K-Fold Cross Validation and we mainly used a Grid Search approach. After having validated all the models, we decided to use an ensemble of the two NNs implemented, the KRR, the SVM and the KNR for the ML cup.

1 Introduction

The main purpose of our project was to train different models, using different libraries to show and compare the final results of each model. All the models were trained for both the three monk datasets and the ML cup 2022. To solve the classification and regression tasks we trained the following models: neural networks, support vector machines, decision trees and k-nearest neighbors; for each of them, we tried either several variants of the model, different libraries or various strategies for the hyperparameter tuning phase. The performance of the models was compared using the mean Euclidean error on the validation set as metric, and we chose the final model for the ML 2022 competition based on this value and other aspects. For the model selection, we always performed cross-validation, while the model assessment phase was performed through a hold-out, using a test set composed of some patterns exclusively selected for this aim.

2 Methods

The first thing we did was to divide the ML cup set into a design set and a test set. We decided to use 80% of the patterns (1194 patterns) for the model selection and the remaining 20% for the internal model assessment (298 patterns).

Neural Networks

To train the neural networks we used two different libraries, which are PyTorch [1] and Keras [2]; with PyTorch, the hyperparameter tuning phase was performed using grid searches, while with Keras it was performed using random searches. We also tried different regularization techniques,

implementing L2 regularization with PyTorch and L1 regularization with Keras. The hyperparameters that we tuned in the model selection process are the learning rate, the momentum coefficient, the L2 regularization coefficient (only with PyTorch), the L1 regularization coefficient (only with Keras), the number of units of the hidden layers and the number of hidden layers. We always used ReLU as the activation function of the internal units. With PyTorch, the grid search algorithm was directly implemented by us from scratch, without using further libraries, and we parallelized the algorithm by implementing a multiprocessing version of it. For each combination of hyperparameters, we performed cross-validation with 4 folds and we created 3 different models for each fold. At the end of the grid search, we returned the best combination based on the averaged mean Euclidean error on the validation set computed on each model created. On the other hand, the random search algorithm was implemented using the methods provided by the Keras framework, albeit we implemented on our own the cross-validation algorithm extending one of the framework's classes, since it was not supported by the framework itself. Finally, the best combination of hyperparameters was returned similarly to PyTorch; for each random combination of hyperparameters, we executed cross-validation over 3 folds, computing 2 different models for each fold, and we chose the best combination based on the averaged mean Euclidean error computed on the validation set of each model created. For both libraries, we used early stopping as stop condition, setting a patience equal to 10; with PyTorch we had again to implement it by ourselves, while with Keras we used it exploiting a functionality of the framework. With both algorithms, we used the mean squared error as loss to train the neural networks and the mean Euclidean error as metric. With both libraries we had to perform a min-max normalization on both inputs and targets, in particular Keras had problems in computing the gradient of not normalized values, therefore we had at the beginning to normalize both inputs and targets, and then denormalize the predicted values of the final model of each library. Concerning the weight initialization, we decided to use the default policies of the two libraries; PyTorch initializes the weight using a uniform random variable which ranges, for each layer, from $-\sqrt{number\_of\_weights}$ to $+\sqrt{number\_of\_weights}$, also Keras initializes the weights of its neural networks according to a uniform random variable, but it ranges, for each layer, from $-\sqrt{6/(fan\_in + fan\_out)}$ to $+\sqrt{6/(fan\_in + fan\_out)}$. Finally, we used the best combination of hyperparameters found either by a grid search or by a random search to train 10 different models, choosing as final model the median one in terms of mean Euclidean error on the validation set to avoid overfitting; also to train the final model we used early stopping.

Support Vector Machines

To train the support vector machines we used the scikit-learn [3] library and the hyperparameter tuning phase was performed using grid searches. The hyperparameters that we tuned in the model selection process are the kernel, the L2 regularization coefficient, the degree for the polynomial kernel, the kernel coefficient (only for the rbf and polynomial kernels), the epsilon-tube (only for regression). We used the `GridSearchCV` class to perform the grid search and cross-validation and in the end we got the best estimator given to us as result. With the algorithm, we used the mean Euclidean error as metric. Also for SVMs we used the min-max normalization.

K-Nearest Neighbors

To train the k-nearest neighbors for classification (KNN) and regression (KNR) estimators we used the scikit-learn library and the hyperparameter tuning phase was performed using grid searches. The hyperparameters that we tuned in the model selection process are the number of neighbors to use and the algorithm used to compute the nearest neighbors. We used the `GridSearchCV` class to perform the grid search and cross-validation and in the end we got the best estimator given to us as result. With the algorithm, we used the mean Euclidean error as metric. Also in this case we used the min-max normalization.

Kernel Ridge Regression

The Kernel Ridge Regression (KRR) is a model similar to the Support Vector Regression (SVR) but the difference is in the loss functions: in fact, KRR uses squared error loss combined with L2 regularization. To train the Kernel Ridge Regression estimators we used the scikit-learn library and

the hyperparameter tuning phase was performed using grid searches. The hyperparameters that we tuned in the model selection process are the kernel, the regularization coefficient, the degree for the polynomial kernel and the kernel coefficient (only for the rbf and polynomial kernels). We used the `GridSearchCV` class to perform the grid search and cross-validation to get the best estimator given to us as result. With this model, we used the mean Euclidean error as metric. Also for the KRRs, we used the min-max normalization as preprocessing step.

Decision trees and Random forests

We also decided to use Decision Trees (DT) and Random Forests (RF) which are models not presented during the class. Both these models can be used for classification and regression. In this section, we are dealing with regression tasks.
DTs predict the value of a target variable using inferred decision rules from the data features. DTs for classification tasks are trees where leaves represent the classes and branches represent the decision rules that allow us to identify the class. DTs for regression are trees where the target variables take real number values. For DTs, we used the scikit-learn [3] library. We performed grid searches to tune the following hyperparameters: the maximum depth of the tree, the minimum number of samples required to split an internal node, the minimum number of samples required to be at a leaf node, the minimum weighted fraction of the sum total of weights required to be at a leaf node and a complexity parameter used for Minimal Cost-Complexity Pruning. The metric we employed to choose the best hyperparameters is the mean Euclidean distance and we used the `GridSearchCV` class to perform grid search and cross-validation.
RF is an ensemble method that works by putting together more decision trees at training time. If we have a classification task, the output class is the one selected by most trees. For regression tasks, the output is the mean prediction of the individual trees. Also for RFs, we used the scikit-learn library and we tuned the hyperparameters performing grid searches in a similar way as we did for DTs. Compared to DTs, with RFs we have one more hyperparameter, which is the number of trees in the forest. Also in this case we used the mean Euclidean distance as a metric and we performed the `GridSearchCV` class to perform grid search and cross-validation.

3 Experiments

3.1 Monk Results

We have classified the patterns of the three monks dataset with every model presented in this report. Despite that, plots and tables present in this part of the relation refer only to the neural network trained using PyTorch. The hyperparameters were found performing a grid search; in Monk1 and Monk2 the hyperparameters taken into account were: the number of units of the hidden layer, the momentum coefficient and the learning rate, while in Monk3 we added also L2 regularization to the loss, therefore we used the grid search also to search the best coefficient for L2 regularization. To train the models we used again the mean squared error as loss. Table 1 reports relevant features regarding the final model built for each monk dataset using PyTorch, figure 1 shows the learning curves for each final model created for the monk datasets using PyTorch.

dataset	**LR**	**Alpha**	**Units**	**Lambda**	**Loss TR**	**Loss VL**	**Acc TR**	**Acc VL**	**Acc TS**
monk1	0.3	0.9	8	-	$4.07e^{-5}$	0.0002	1.0	1.0	1.0
monk2	0.3	0.8	8	-	0.0001	0.0006	1.0	1.0	1.0
monk3	0.5	0.6	8	0.01	0.0663	0.2708	0.928	0.958	0.972

Table 1: Final set of hyperparameters found with PyTorch performing a grid search (LR = learning rate, Alpha = momentum coefficient, Units = units for each hidden layer, Lambda = coefficient for L2 regularization)

3.2 Cup Results

Neural Networks

With PyTorch, as mentioned in section 2, we implemented a grid search algorithm to find the best combination of hyperparameters and in particular, we executed a double nested grid search. For the first grid search, we considered the following values for the learning rate: 0.02, 0.1, 0.25 and 0.4; these values were chosen because we observed that learning rates larger than 0.4 were likely to lead to unstable learning curves and with a learning rate too small it was unable to generalize properly the function. Regarding the momentum, we tried the following values: 0.0 (i.e no momentum), 0.4 and 0.8, while for the L2 regularization we put on the test only very small coefficients, i.e. 0.0, 0.0001 and 0.01, because we noticed that even apparently small values for the Ridge coefficient could cause underfitting on the trained models. We put in the grid search also the number of nodes of the hidden layers, setting two possible values: 20 units and 40 units; last but not least, we trained models with either one hidden layer or two. In total, we tried 144 different combinations for the first grid search. In the second nested gird search, we explored more deeply the surrounding values of the best combination of hyperparameters returned by the first grid search; for the number of units and hidden layers, we simply decided to use the best value returned by the first grid search as the only possible value for the second grid search. On the other hand, for the learning rate, the momentum coefficient, and the L2 regularization coefficient, we searched over 5 different values for each of these three hyperparameters, therefore the second nested grid search searched over 125 different possible combinations. The two hyperparameters that affected the most the learning curves and the final model were the learning rate and the L2 regularization coefficient; the former was revealed to be essential to have a smooth and adequate learning curve, while the latter tended to shrink down too much the weights of the neural network if not chosen properly. On a laptop having a processor Intel Core i7 8565u processor 1.8 GHz, the execution of the two nested grid searches took about 2 hours, this time was partially shrunk down thanks to the fact that the algorithm was executed involving 4 different processes in parallel. After this second grid search, we trained 10 different models using the best combination of hyperparameters returned by the second nested grid search, which was: nodes for each hidden layer 40, number of hidden layers 2, learning rate 0.15, momentum coefficient 0.8 and L2 regularization coefficient 0.0 (the grid searches preferred to not use L2 regularization to train the models). To avoid overfitting, we returned as final model the one having the median mean Euclidean error on the validation set. Finally, we computed the final mean Euclidean error on both training set and validation denormalizing the predicted targets calculated by the final model; in fact, since PyTorch revealed to struggle in computing the gradient of not normalized values, we had to normalize them according to a min-max normalization schema as a preprocessing step. To create reproducible results, we set up the following seeds `torch.manual\_seed(2)`, `random.seed(2)` and `np.random.seed(2)`. On fig 2 is reported the learning curve of the final model, on fig 3 is reported the normalized mean Euclidean error during the learning process, on table 3 is reported the final mean Euclidean distance on both training set and validation set in the original scale, on table 2 is reported the final set of hyperparameters returned by PyTorch and Keras

The second library to build neural networks which we used is Keras. With Keras, we used a different regularization technique, i.e. L1 regularization, and we searched the best combination of hyperparameters through random searches. For each hyperparameter, we set a range from which to extract random values and a random variable, either uniform or logarithmic, to assign to each value in the range a probability to be chosen. Moreover, we implemented a double nested random search, where the second random search searches the best hyperparameters in the surroundings of the hyperparameters returned by the first random search. For the first random search, we searched the learning rate between 0.05 and 0.15, because we noticed that the learning curves were particularly unstable with a learning rate larger than 0.15, and the mean Euclidean error on the validation set remained too large with a learning rate smaller than 0.05. We searched the momentum coefficient between 0.0 and 0.8 and in general, the choice of the momentum coefficient was revealed to be less important than the learning rate for the smoothness of the learning curve.

Another hyperparameter that we used is the number of nodes for each hidden layer, in the first random search we set up the range of this hyperparameter between 30 and 50 units; we set up this range because with a number of units smaller than 30 the mean Euclidean error on the validation set remained pretty high, while with a number of units larger than 50 the performance of the final model did not change very much with respect to a model with 50 units for each hidden layer. The choice of the L1 regularization coefficient was revealed to be very important too, indeed a too large value for this hyperparameter compromises the generalization capabilities of the final model and at the end, we decided to search this hyperparameter between 0.00001 and 0.01 with a logarithmic distribution law. For the second nested random search, we simply explored more deeply the search space of the best combination of hyperparameters returned by the first random search; for each hyperparameter, except the number of nodes, we set up a new range from which to extract new values in this way; the lower extreme of the interval is the value of the hyperparameter returned by the first random search multiplied by 0.8, while the upper extreme of the interval is the value of the hyperparameter returned by the first random search multiplied by 1.2. For the second nested random search, we simply used the same number of nodes of the hidden layers returned by the first random search. Finally, we repeated from scratch this operation to build models with either one or two hidden layers; the latter models were revealed to be slightly more accurate than the former ones, therefore we decided to use two different hidden layers for the final model. Similarly to PyTorch, for each combination of hyperparameters randomly extracted we executed a cross-validation over 3 folds, training 2 neural networks for each fold and returning the averaged mean Euclidean error on the validation set and each random search searched over 50 different random extracted combinations of hyperparameters. We also used early stopping, setting a patience equal to 10 and we fixed 1 as seed for `np.random.seed()` and `tf.random.set\_seed()` for reproducibility. On a laptop having a processor Intel Core i7 8565u processor 1.8 GHz, the execution of the two nested random searches took about 6 hours, but this time the code was executed sequentially. We trained 10 different models with the best combination of hyperparameters returned by the double nested random search and, to avoid overfitting, we returned as final model the one having the median mean Euclidean error on the validation set. Since also Keras was not able to compute the gradient of not normalized values, we had first to normalize both inputs and targets, and then to denormalize the final labels predicted by the model to compute the mean Euclidean error in the original scale. The final set of hyperparameters returned by the double nested random search is: learning rate 0.128, momentum coefficient 0.73, number of units for each hidden layer 42, L1 regularization coefficient 0.0015, number of hidden layer 2; please note that these hyperparameters are very similar to the ones returned by the double nested grid search implemented using PyTorch. On fig 4 is reported the learning curve of the final model, on fig 5 is reported the normalized mean Euclidean error during the learning process, on table 3 is reported the final mean Euclidean distance on both training set and validation set in the original scale, on table 2 is reported the final set of hyperparameters returned by PyTorch and Keras.

Support Vector Machine and Kernel Ridge Regression

Since the ML Cup contains two targets, we needed to get two estimators for the SVM, one for each target. To do this, we split the search into two parts: during the first part, we focused on the first target, so we executed three grid searches, one for each SVM kernel. For the regularization parameter, the epsilon tube and kernel coefficient we used the logspace function so that we had numbers spaced evenly on a log scale and in case we were not satisfied with the MEE we found out we could tune the search zooming in the range we got from the first grid search, while about the polynomial degree, we used 2, 3 and 4 as parameters; finally, the kernels we decided to consider were the rbf, linear and polynomial. After the computation of all the three grid searches, we first analyzed the MEE given by the three estimators and then we selected the best one, according to the best combination of parameters and the MEE. We did the exact same work also using the second target and in the end, we acquired the two best estimators. On table 3 is reported the final mean Euclidean error on both training and validation set in the original scale, on table 4 is reported the final set of hyperparameters for both SVM models returned by `GridSearchCV`.
A similar approach was used for the KRR models, so we performed the search in two parts: we

first searched for the best estimator for the first target and then we did the same for the second target. The parameters we decided to tune with the grid search were the kernel, the regularization parameter, the kernel coefficient and the polynomial degree. Again, we used the logspace function to set the values of the regularization parameter and the kernel coefficient. About the kernel, once again we tuned the rbf, linear and polynomial ones; finally, the degree parameters were 2, 3 and 4. After the computation, we analyzed the MEE given by the three estimators and then we selected the best one, according to the best combination of parameters and the MEE. We did the exact same work also using the second target and in the end, we acquired the two best estimators. On table 3 is reported the final mean Euclidean error on both training and validation set in the original scale, on table 4 is reported the final set of hyperparameters for both KRR models returned by `GridSearchCV`.

K-nearest neighbors

With the K-Nearest Regression (KNR) model we performed a grid search using the following parameters: as number of neighbors all the integers between 1 and 49 (included), as algorithm we tuned on *auto*, *ball_tree*, *kd_tree* and *brute*. The result of the grid search gave us 19 as number of neighbors and *auto* as algorithm. On table 3 is reported the final mean Euclidean error on both training and validation set in the original scale.

Decision trees and Random forests

For both DT and RF there was no need to normalize data and also we did not split the target into two parts, since these models are able to predict two-dimensional targets. We trained the two models by performing two grid searches using similar values. On table 3 is reported the final mean Euclidean error on both training and validation sets. We can see how the RF performs better than the DT, but both models, with respect to the others, overfitted a bit, since they have a low score on TR and a high score on VL.

Choice of the final model and model assessment

For the final model for the ML cup 2022, we decided to implement an ensemble using the best models trained during the model selection phase. We discarded the decision tree and the random tree models, the first one because it had a too high error on the validation set, and the second one because it had a too low error on the training set, implying it has probably overfitted the problem. Therefore for the ensemble, we used the two neural networks, a KRR model (only for the first target) and a SVM model (only for the second target) and the KNR model, so in total we used four different models, considering the KRR and the SVR as only one. We used the test set to assess our final model, and we registered a mean Euclidean error approximately equal to 1.3536 as the final test error. After that, we used the final model to predict the targets for the blind set.

Library	**Strategy**	**LR**	**Alpha**	**Units**	**L1 lambda**	**L2 lambda**	**layers**
PyTorch	Grid search	0.15	0.8	40	-	0.0	2
Keras	Random search	0.128	0.73	42	0.0015	-	2

Table 2: Final combination of hyperparameters for the neural networks for each library (LR = learning rate, Alpha = momentum coefficient, Units = units for each hidden layer, L1 Lambda = coefficient for L1 regularization, L2 Lambda = coefficient for L2 regularization, layers = number of hidden layers)

MEE	PyTorch	Keras	SVM	KRR	KNR	Decision Trees	Random Forests
TR	1.4594	1.5731	1.2742	1.3637	1.3685	1.26278	0.93271
VL	1.5842	1.6498	0.807/1.029	0.806/1.099	1.452	1.85260	1.52407

Table 3: Denormalized mean Euclidean error of the final models implemented for the ML Cup (TR = training set, VL = validation set)

Target	Model	Strategy	Kernel	Reg. par.	Gamma	Epsilon
1	SVR	Grid search	rbf	16.387	1.2599	0.1862
2	SVR	Grid search	rbf	1.8616	4.5077	0.0009
1	KRR	Grid search	rbf	0.0396	1.2599	-
2	KRR	Grid search	rbf	0.0046	0.5	-

Table 4: Final combination of hyperparameters for SVM and KRR models (Reg. par. = regularization strength, Gamma = kernel coefficient, Epsilon = epsilon tube, Degree = polynomial kernel degree)

4 Conclusion

There were two kinds of tasks, the first one of classification of the 3 monk datasets, and the second one of regression using the ML Cup dataset. To solve both tasks we trained two neural networks implemented with PyTorch and Keras, and SVM, KRR, KNN, KNR, Decision Tree and Random Forest models implemented with scikit-learn library. In particular, we applied a grid search on all the models, except for the neural network implemented with Keras, which was trained using random searches. We found out that some models, such as SVMs and KNRs, performed slightly better with respect to others like the decision trees, since their validation error revealed to be lower. In the end, we decided to implement an ensemble of NNs, KNR, KRR for the first target and SVR for the second target, using 4 models in total, to predict the targets of the ML Cup's blind set.

Acknowledgments

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

References

[1] Francois and others Chollet. Keras, 2015. URL: https://github.com/fchollet/keras.

[2] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019. URL: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

[3] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

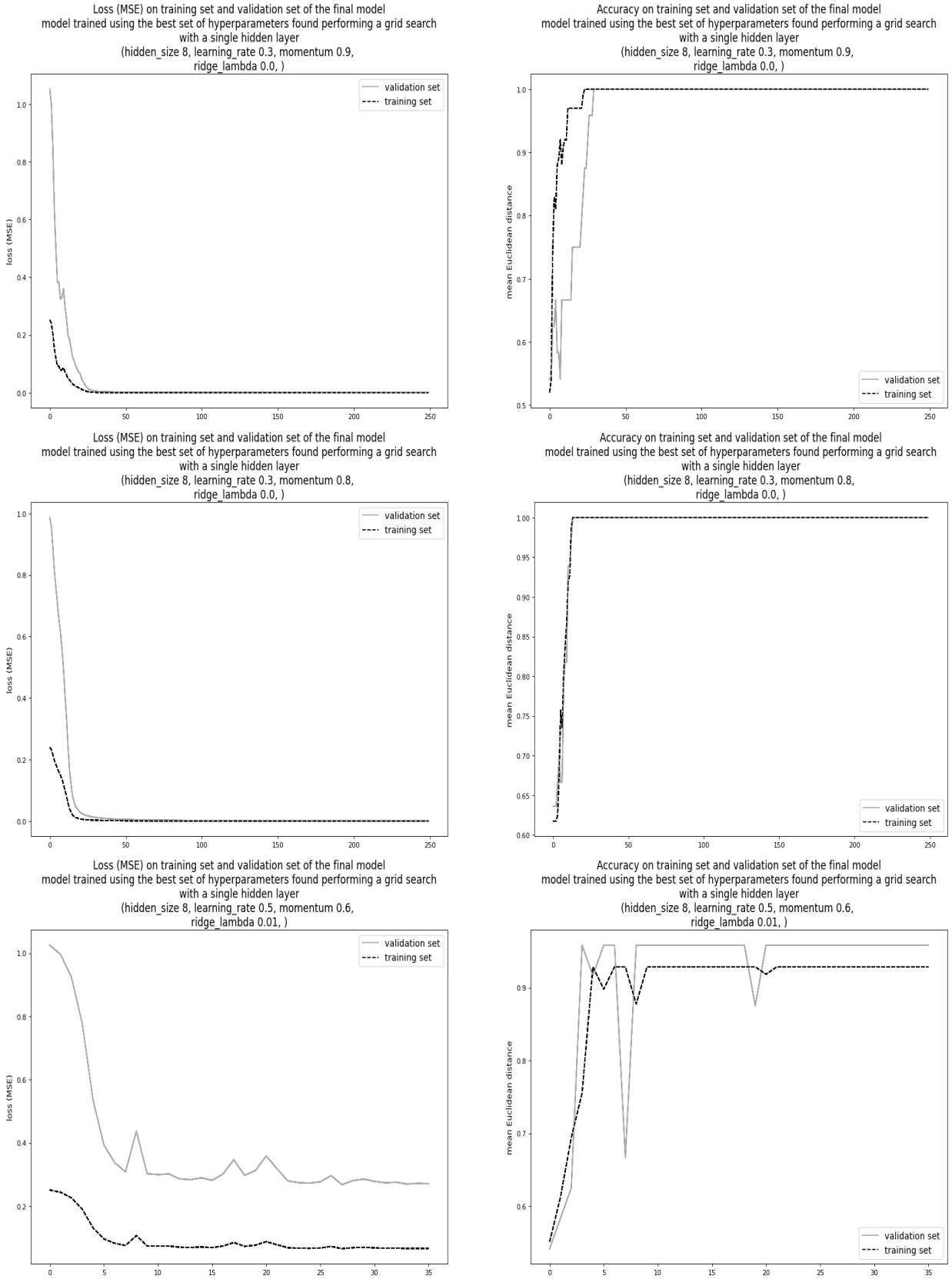

Figure 1: In the first row there are the learning curve and the accuracy of the final model on Monk1, while in the second row the plots refer to dataset Monk2 and in the third on dataset Monk3. The plots' titles also indicate the combination of hyperparameters used to train the final model (hidden_size refers to the number of units used for the hidden layer)

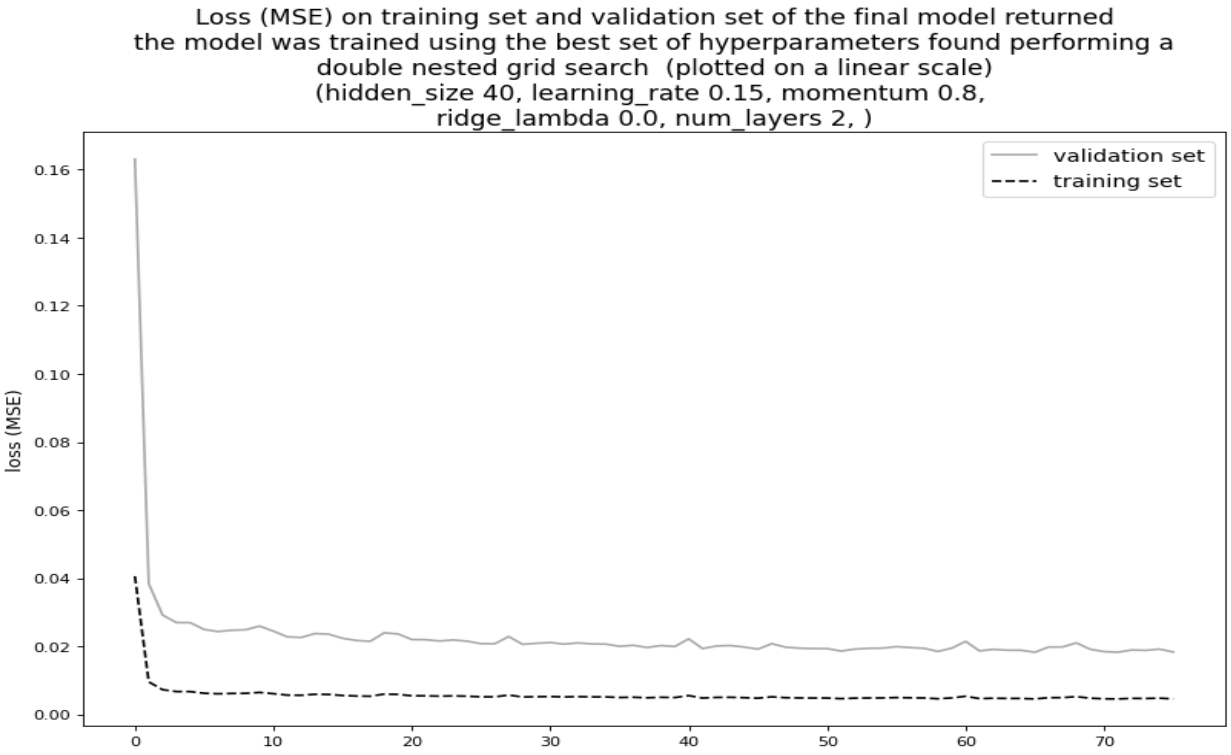

Figure 2: Learning curve of the final model trained with PyTorch for the ML cup 2022. The loss used was the mean squared error (MSE)

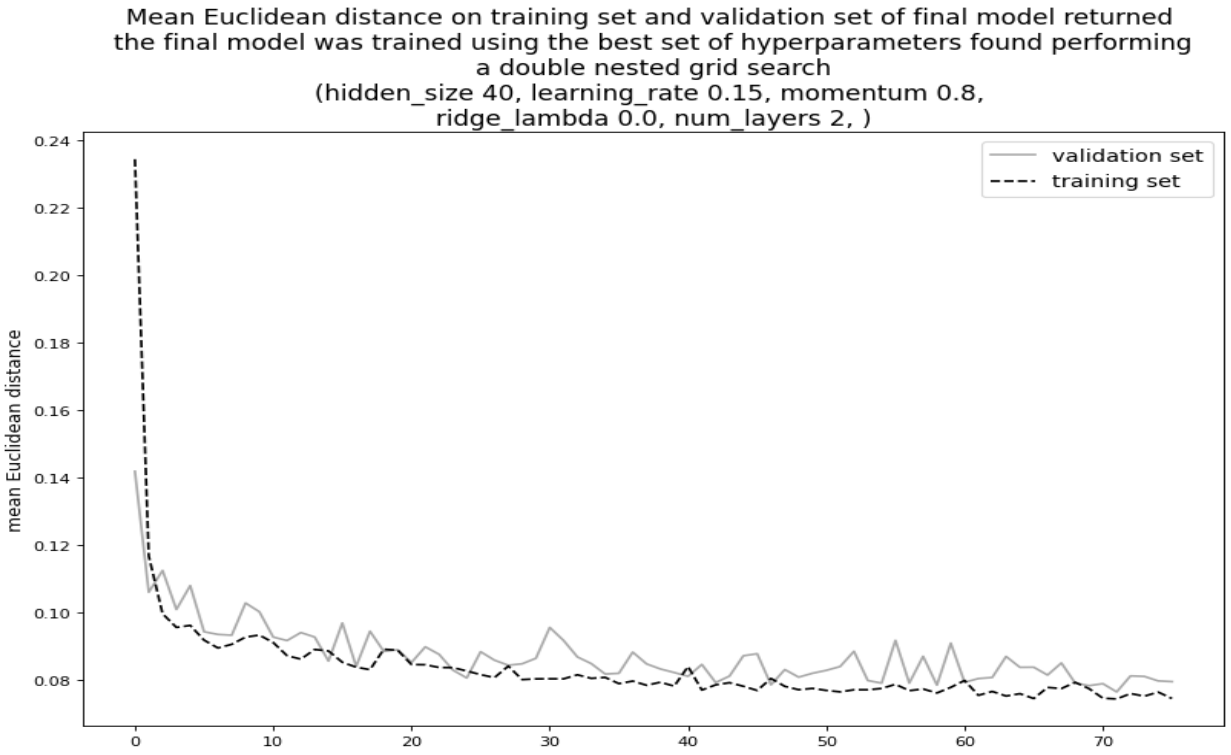

Figure 3: History of the (normalized) mean Euclidean error during the training of the final model trained with PyTorch

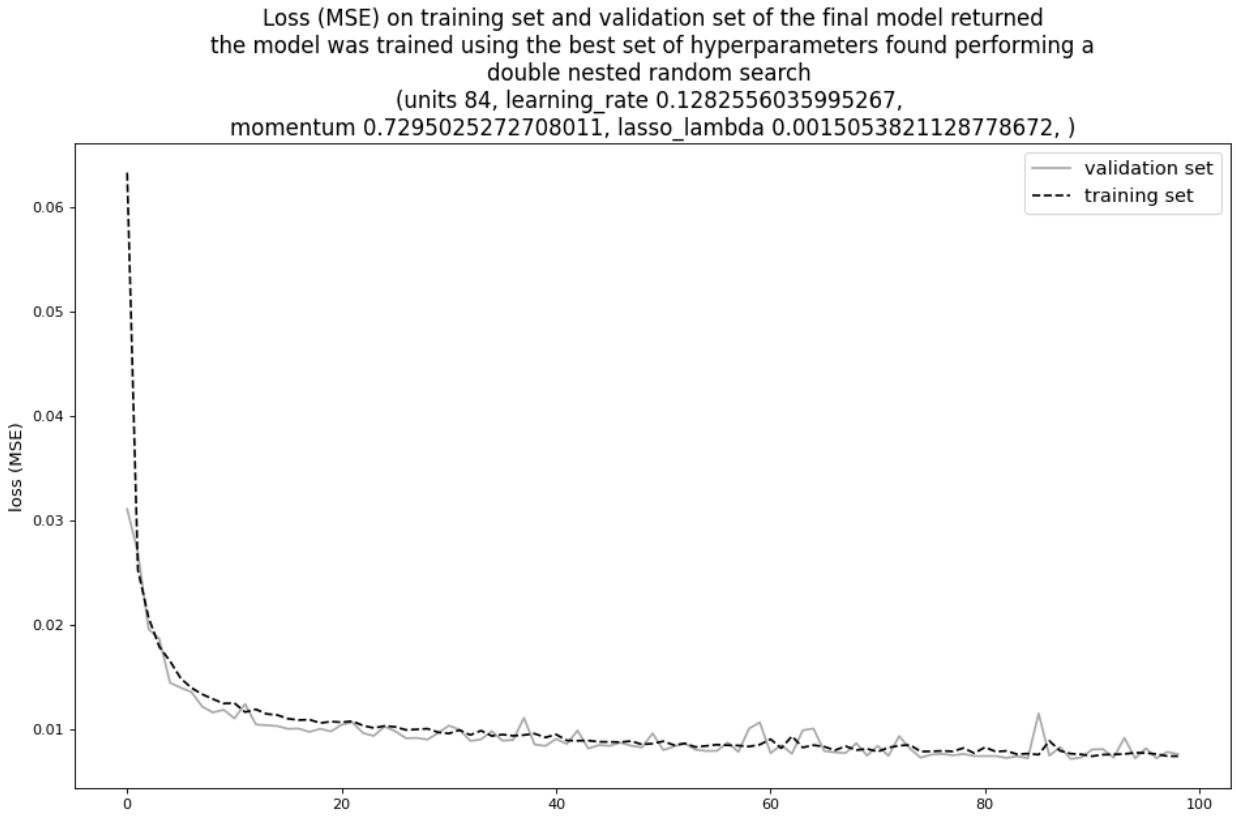

Figure 4: Learning curve of the final model trained with Keras for the ML cup 2022. The loss used was the mean squared error (MSE). The model has two distinct hidden layers each one with 42 units.

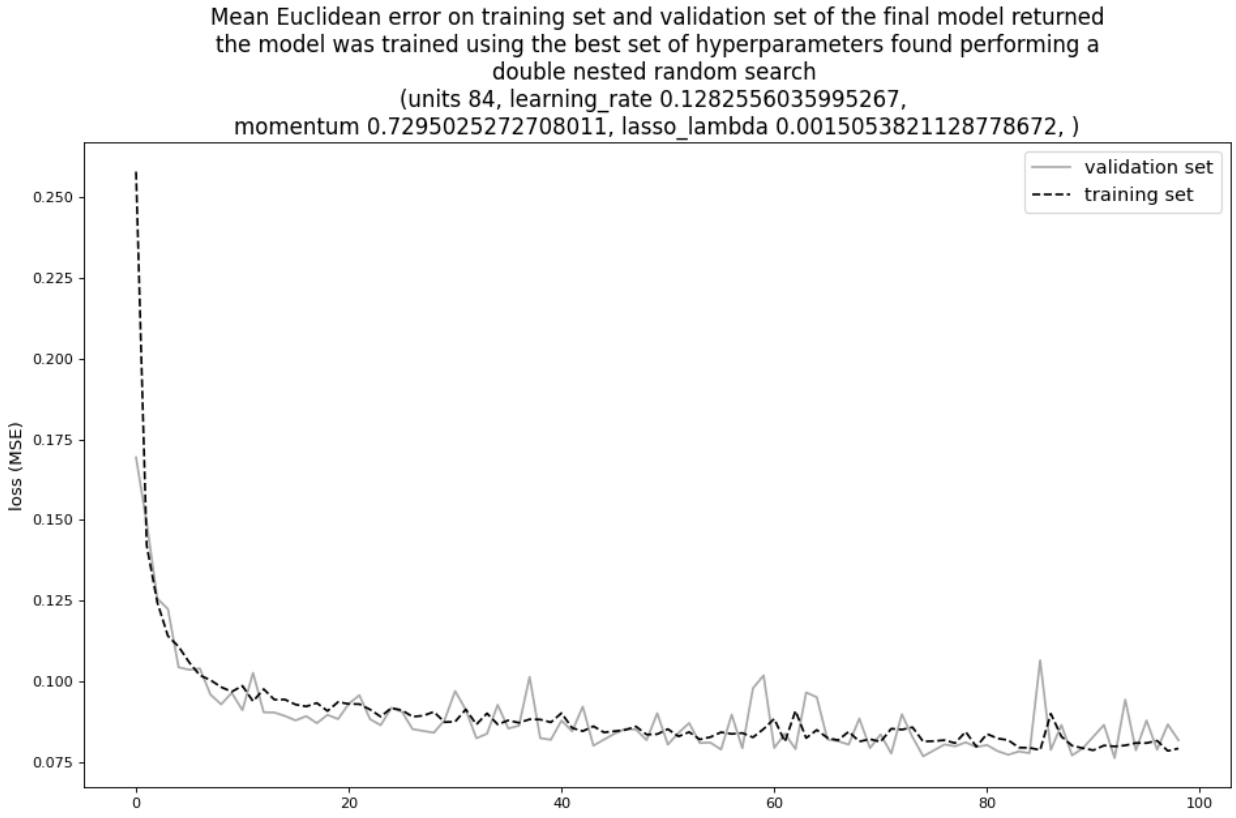

Figure 5: History of the (normalized) mean Euclidean error during the training of the final model trained with Keras. The model has two distinct hidden layers each one with 42 units.

A Appendix

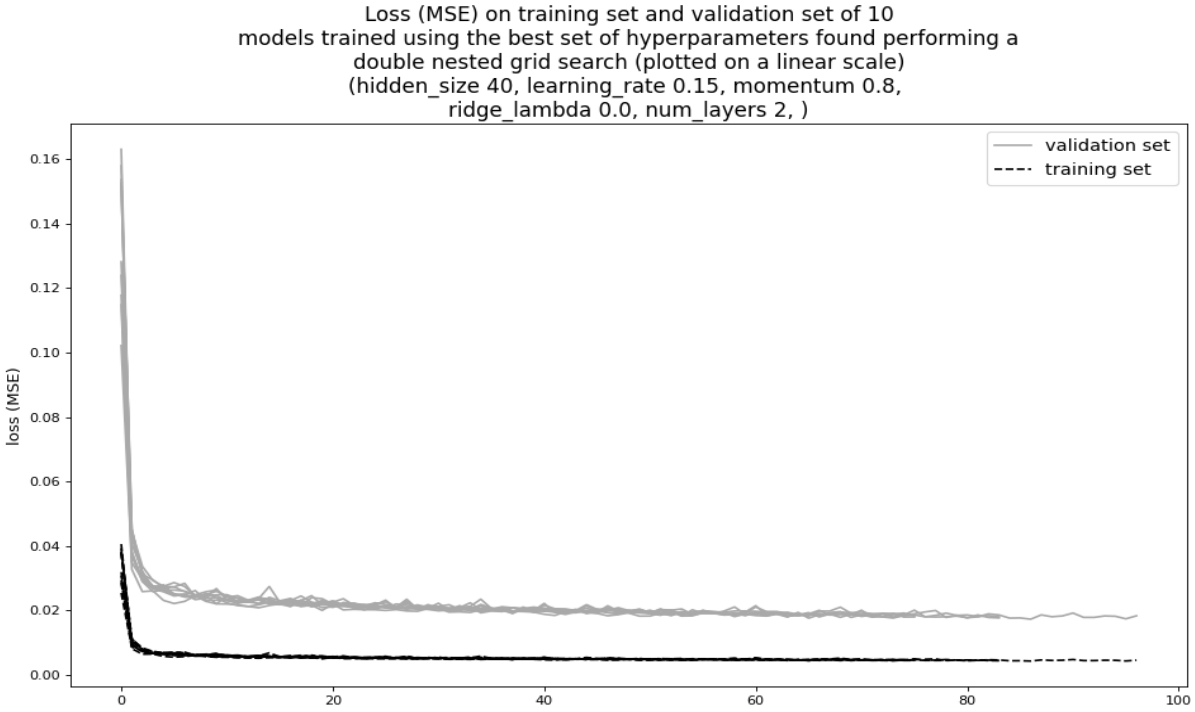

Figure 6: ML Cup 2022, Learning curves of different models created using different weights initialization, with the combination of hyperparameters returned by the double nested grid search. Models created using PyTorch

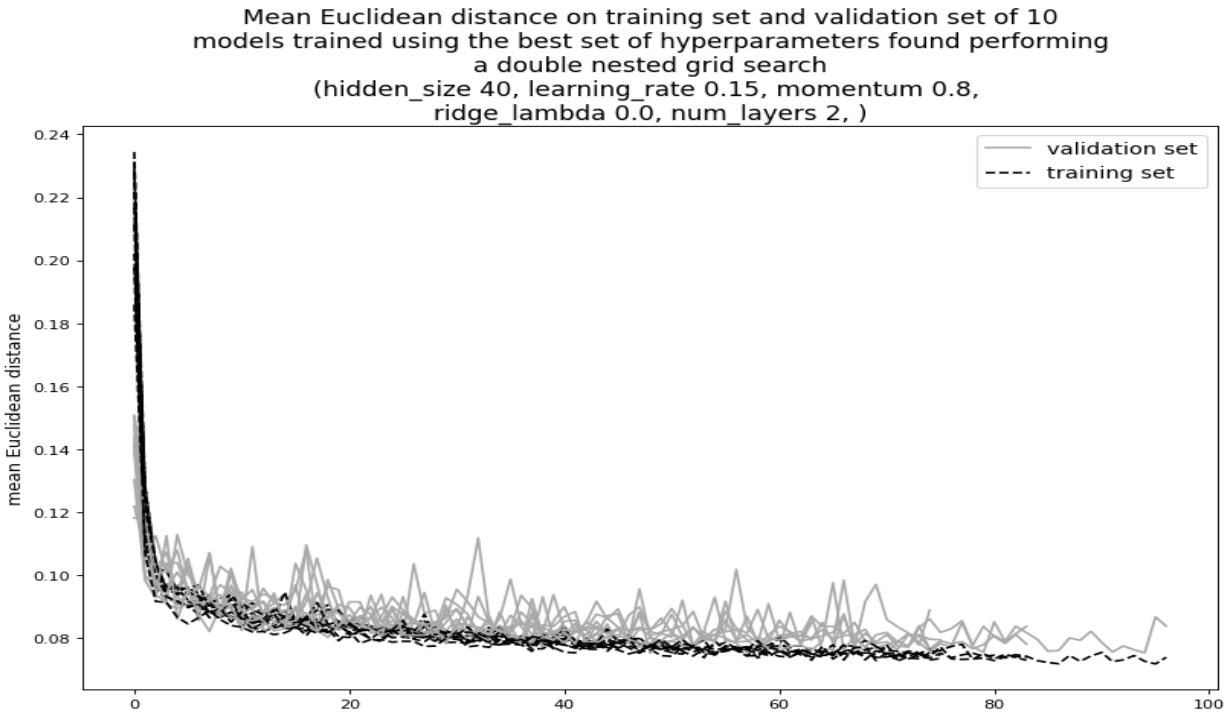

Figure 7: ML Cup 2022, mean Euclidean error of different models created using different weights initialization, with the combination of hyperparameters returned by the double nested grid search. Models created using PyTorch)

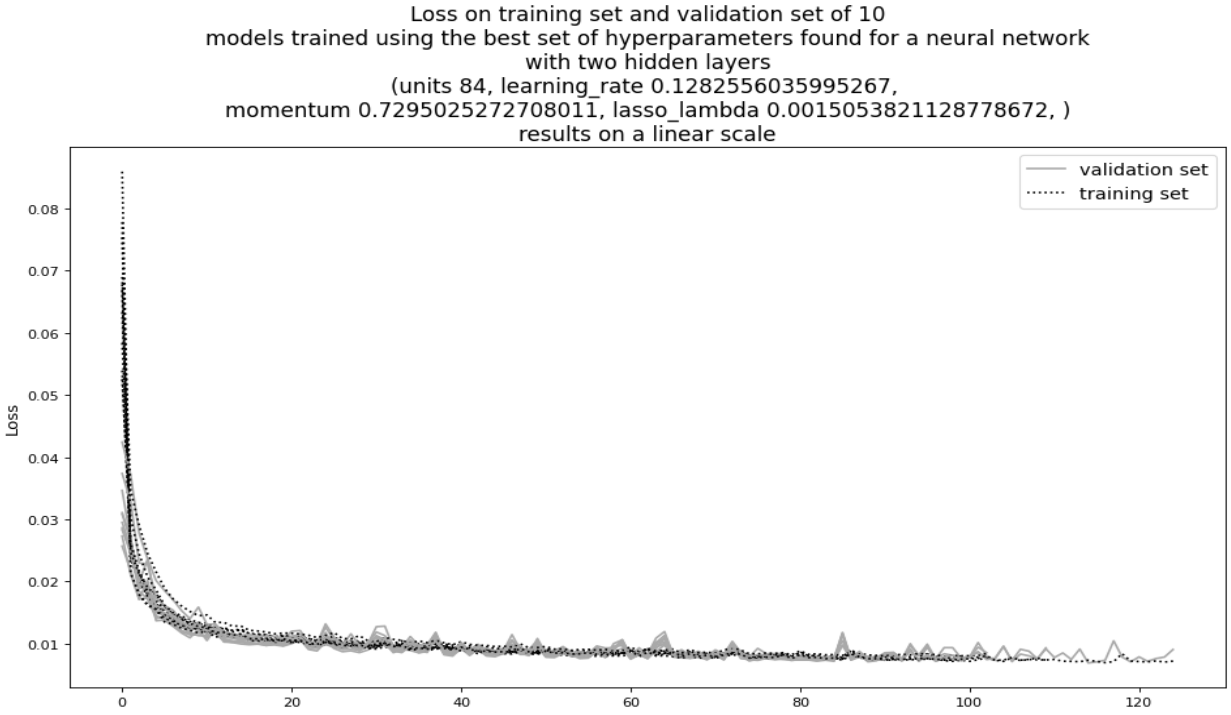

Figure 8: ML Cup 2022, Learning curves of different models created using different weights initialization, with the combination of hyperparameters returned by the double nested random search. Models created using Keras (these models were created with two hidden layers each one having 42 units)

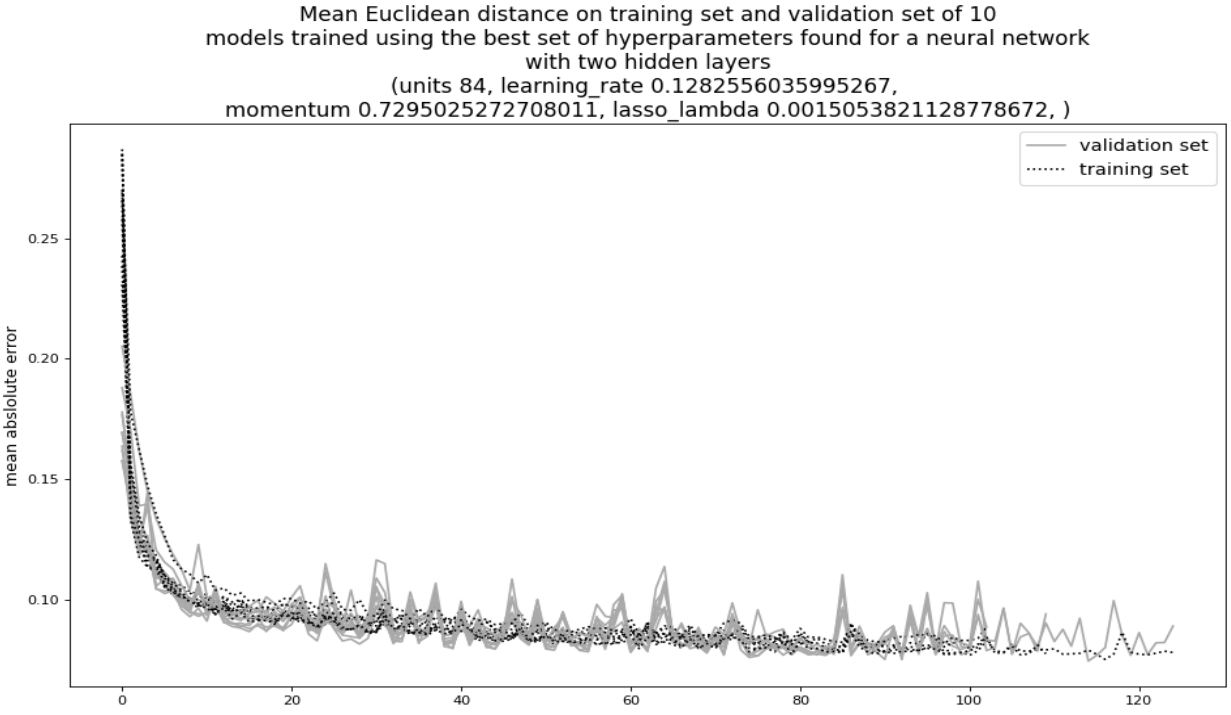

Figure 9: ML Cup 2022, mean Euclidean error of different models created using different weights initialization, with the combination of hyperparameters returned by the double nested random search. Models created using Keras (these models were created with two hidden layers each one having 42 units)