ISANET LIBRARY

Alessandro Cudazzo
alessandro@cudazzo.com

Giulia Volpi
giuliavolpi25.93@gmail.com

Department of Computer Science, University of Pisa, Pisa, Italy

ML - Academic Year: 2019/2020

January 29, 2020

Type of project: **A**

ABSTRACT

In this work, we developed a new Neural Network library in Python, and we called it **IsaNet Lib**. It provides high and low-level API with an entire module for model selection. First, it was tested on the MONK Datasets (classification tasks) and then on the CUP Dataset (regression task). All the analyses use well-known validation methods (Hold-out, K-fold Cross Validation) to provide a model that can generalize the data.

1. Introduction

The aims of project **A** of the Machine Learning (ML) course are to implement an ML model simulator (Neural Network, SVM, ...), understand the hyper-parameters effect on the model and solve a supervised regression learning task by using the CUP dataset provided in the course. For the first aim, we decided to implement a Multi-Layer Perceptron (MLP) model with *back-propagation* (gradient descent), *Classic Momentum* (also known as *Polyak's heavy ball method*[1]), *Nesterov* (introduced in [2]), the new *Accelerated Nesterov* (see [3]), *L2 regularization*, some *Early Stopping*, and other features. Since the project grows very fast, we decided to develop a neural network library that would allow us to focus on the other aims. We will explain all the major details of the library in the next session. In order to test and validate our library we used MONK datasets [4] and after that, we moved on to CUP dataset analysis. To determine the best hyper-parameters configuration for the CUP's task we performed a grid search with K-fold cross-validation and the final model was re-trained using hold out as validation method for the early stopping.

The remaining text is structured as follows. In Section 2, we have illustrated the methods used for all the analysis. The experiments are shown in section Section 3, Subsections 3.1 and 3.2 respectively contain the results of MONK and CUP, while Section 4 is devoted to conclusions.

2. Method

In this section, we first present the IsaNet Lib (Sec. 2.1). Then we describe the Early Stopping and Complexity Control (Sec. 2.2), the initialization methods (Sec. 2.3) used in the experimental phase, and finally, the preprocessing and validation schema (Sec. 2.4 and 2.5).

2.1 Design and Code Overview

The first step we made was to write a library that would provide us a flexible and modular neural network model. We called it **IsaNet**. We wrote the library entirely in Python using Numpy as a package for scientific computation. All the main operations of a neural network, as the feed-forward and the back-propagation algorithms, are performed by using matrices, and layers of nodes are stored in a list of matrices where columns are the weights of a single node (the bias is the first element). This implementation allowed us to speed up the computation compared to an object-oriented structure (layers, nodes views as object); this was possible thanks to Numpy that can efficiently perform matrix operation by parallelization under the hood. Numpy uses optimized math routines, written in C or Fortran, for linear algebra operation as *Blas*, *OpenBlas* or *Intel Math Kernel Library (MKL)* [5]. IsaNet is composed of low and high-level APIs divided into modules.

Modules - Low Level API:

- `model.py`: this module implement the main class, `Mlp`, which stands for Multi-Layer Perceptron. The class allows to add one or multi-layers and specify for each of them: a specific number of nodes, an activation function, the range used for the weights initialization, the value of L2 regularization term. Furthermore, it allows you to associate a specific optimizer used to train the model and provide to the user a fitting and predict methods to perform training and prediction operation. In the fitting method, we can specify the training set, the validation set, the maximum number of epochs, the early stopping technique, the number of elements for a minibatch (the library supports full batch, mini-batch and online) and the verbose mode.

- `optimizer.py`: this module contains the `SGD` class, which implements the Stochastic Gradient Descent with mean squared error as loss function and different types of momentum: Classic (CM) , Nesterov (NM) and the new Accelerated Nesterov (ANM) (see [3]). In the Appendix A, there are some comparisons between the various momentum. Hence, the available hyperparameters are η for the learning rate, α for the CM, True/False for the NM, and σ for the ANM. Besides, this module provides some Early Stopping to use in the fitting phase. In the `SGD` we can check if there are no improvements in optimizations (training error), and with the `EarlyStopping` class provided by the module, we can detect cases of overfitting. For both, we can specify a certain threshold and a maximum number of epochs for which the increase or decrease must occur. Further details on overfitting handling are left in Sec. 2.2.

- `activation.py`: this module implements Sigmoid, Identity, Tanh, and Relu activation functions, while `metrics.py` provide functions for MSE (Mean Square Error), MEE (Mean Euclidean Error), and Accuracy.

Modules - High Level API:

- `neural_network.py`: this module provides two main classes, `MLPRegressor` and `MLPClassifier`, respectively, a Multi-layer Perceptron Regressor and a Multi-layer Perceptron Classifier. Both classes simplify the creation of an MLP model by specifying the hyper-parameters, as constructor parameters of the class, besides the classic fitting and prediction operations. All hyper-parameters provided by low level APIs are present for these classes.

- `model_selection.py`: composed of two classes, `Kfold` and `GridSearchCV`. One provides the k-folds cross-validation method, and the other implements an exhaustive search on a subset of hyperparameters space called Grid Search.

Some examples, showing the use of APIs, have been added in Appendix B, and the library is distributed open on GitHub with the related documentation at https://github.com/alessandrocuda/IsaNetNN/.

2.2 Early Stopping and Complexity Control

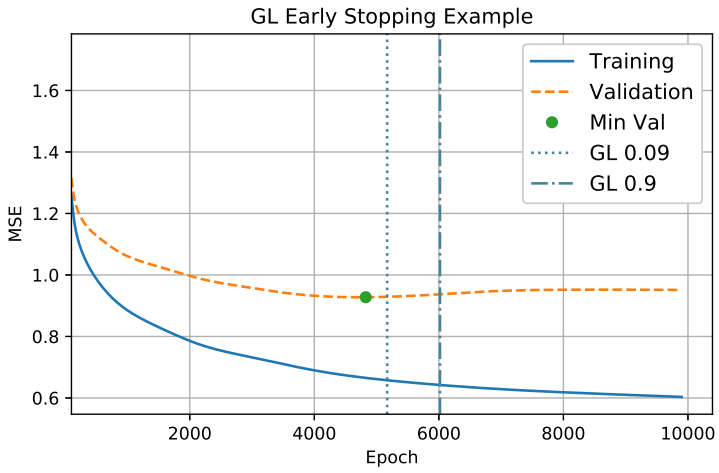


Figure 2.1: From the generalization error we can see that the model goes into overfitting but minimum of generalization ($MSE = 0.9275$) is reached at 4826 epochs. We can obtain different stop points changing the ε_{GL} value (suppose $s_{UP} = 200$ epochs): with $\varepsilon_{GL} = 0.09$ the stop point is at 5170 with Validation $MSE = 0.9283$ and with $\varepsilon_{GL} = 0.9$ the stop point is at 6018 with Validation $MSE = 0.9371$. With no early stopping, the fitting will end at the max number of epochs (10000) with 0.9514 as generalization error.

When training an MLP, we can use the validation error to detect overfitting and underfitting. We are generally interested in obtaining the minimum error of generalization and avoiding overfitting. In the ideal case, we expect that the validation curve will show a smooth increase when overfitting occurs, but in reality, this is not always true. In fact, a validation curve may have more local minima, this is well illustrated in [6], and many methods are proposed in order to handle this problem. Two of them have been implemented and used, as can be seen in Section 3, devoted to experiments. They are: GL (stops as soon as the generalization loss exceeds a certain threshold) and UP (stops when the generalization error increased in s successive epochs). From now on, we'll refer to ε_{GL} as the threshold used in the GL method and to s_{UP} as the max number of epochs in UP method. An example, applied on CUP dataset, is shown in Fig. 2.1.

Of course, in addition to the classic early stopping, in the experimental phase we used the regularization as a form of complexity control in order to manage the correct generalization of the model. Hereinafter, we will refer as λ as the hyperparameter of the weights regularization (see [7]).

2.3 Weights' Initialization

Weights' initialization is a very important phase and can lead to a better and faster convergence. Many deep learning applications make use of a random weights initialization where each weight is drawn from a zero-mean Gaussian distribution $N(0, v^2)$ with a fixed standard deviation v (e.g 0.01 in [8]) or from an uniform distribution $U(-a, a)$ in the interval $(-a, a)$. Another technique, proposed by Glorot and Bengio in [9], advice a properly scaled uniform distribution for initialization called **normalized initialization**:

$$W \sim U\left[-\frac{\sqrt{6}}{\sqrt{n+m}}, \frac{\sqrt{6}}{\sqrt{n+m}}\right]$$

where, n and m represent, respectively, the input and output size of a given layer. This initialization scheme is commonly referred to as the *Xavier initialization*. Some of these techniques have been used for our experiments and all the details are described in Sec. 3.

2.4 Preprocessing

During the analysis, we have considered some datasets: the MONK and the CUP. Since the features of the MONK datasets are categorical the *one-hot encoding* technique has been applied on them obtaining 17 binary features. On the other hand, the CUP's features have been reduced from 22 to 12 (10 features and 2 for the targets) by eliminating the redundant ones (more details can be found in Appendix D).

2.5 Validation Schema

Since the MONK datasets are only used to check the implementation of the library, they are already divided into training and test sets and there is no model assessment phase, we considered the test set as the validation set. Several tests have been done, with different hyperparameters, to find the best models for MONK tasks. Instead, the CUP dataset is composed of a training dataset of 1765 records and a blind test set (no target features) consisting of 411 records. We randomly divided the original training set into a *Design Set* (85%, 1500 records) and *Internal Test Set* (15%, 265 records). For the hyperparameters search, we performed two grid search using 4-folds as a cross-validation method. We chose to use 4 folds because we wanted to keep the trade-off between a good estimate of the generalization error and the computational time. We reported more details on model selection and model assessment about the CUP in Sec 3.2.

3. Experiments

Now, we can go further and see some experiments in detail. MONK tasks need very small weights to converge correctly, so we decided to use random weights extracted from an *uniform distribution* in the interval $(-0.003, +0.003)$, while for the CUP, we used the *normalized initialization* to compromise between the goal of initializing all layers to have the same activation variance and the same gradient variance [9]. For the experiments on both datasets, we used the *SGD full batch* (the 'batch' approach allows to obtain smoother learning curves respect to minibatch / online) with the *Nesterov momentum* and the *sigmoid* as activation function of the hidden layer. Since the MONK are *classification tasks*, we used the *sigmoid* for the output layer followed by a threshold function with 0.5 as threshold gate. For the CUP, we used a *linear* activation function for the output because we're dealing with a *regression task*. All our experiments were performed on a Cascade Lake Intel CPU with 8 cores at 3.4GHz and with Intel MKL as optimized math routine for Numpy.

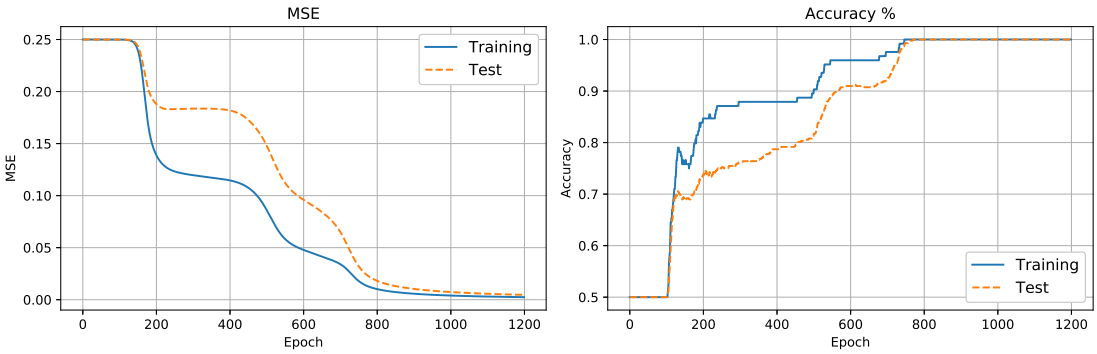
The MONK experiments are reported in Section 3.1 while, in Section 3.2, those on the CUP.

3.1 Monk's Results

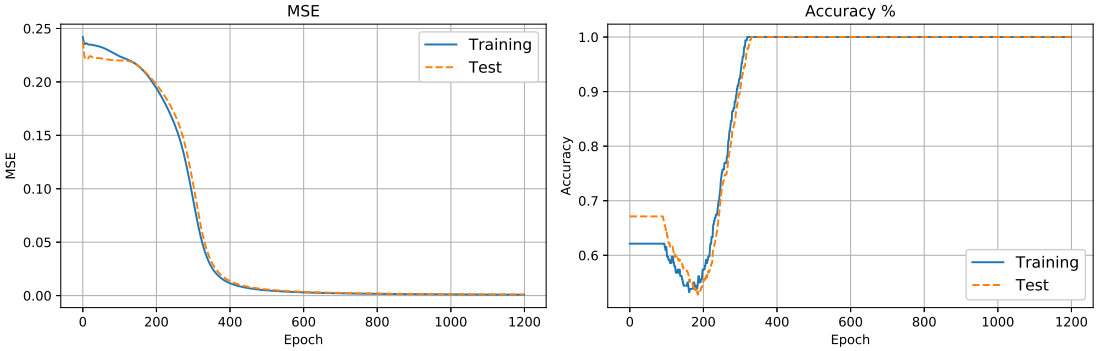
All the models in Tab. 3.1 have the same network topology: 17 (input units) - 4 (hidden units) - 1 (output unit). Fig. 3.1 shows the plots of the MSE and accuracy for the three MONK's benchmarks. The MONK 3 *overfits*, without regularization, after 1200 epochs and we can see how regularization can control the complexity of the model to avoid this problem. In Appendix C are shown more trials and the regularization behaviour on MONK 3.

Best models for the MONK tasks							
Task	η	α	λ	**MSE (TR)**	**MSE (TS)**	**Accuracy (TR)**	**Accuracy (TS)**
$Monk_1$	0.81	0.9	0	0.00949 ± 0.01978	0.0134 ± 0.0253	$99\% \pm 2\%$	$99\% \pm 3\%$
$Monk_2$	0.903	0.9	0	0.00067 ± 0.00007	0.0008 ± 0.0001	$100\% \pm 0\%$	$100\% \pm 0\%$
$Monk_3$	0.8	0.6	0	0.04497 ± 0.00001	0.0467 ± 0.0000	$95\% \pm 0\%$	$95\% \pm 0\%$
$Monk_3\,(r)$	0.8	0.6	0.001	0.05537 ± 0.0000	0.04605 ± 0.000	$94\% \pm 0\%$	$97\% \pm 0\%$

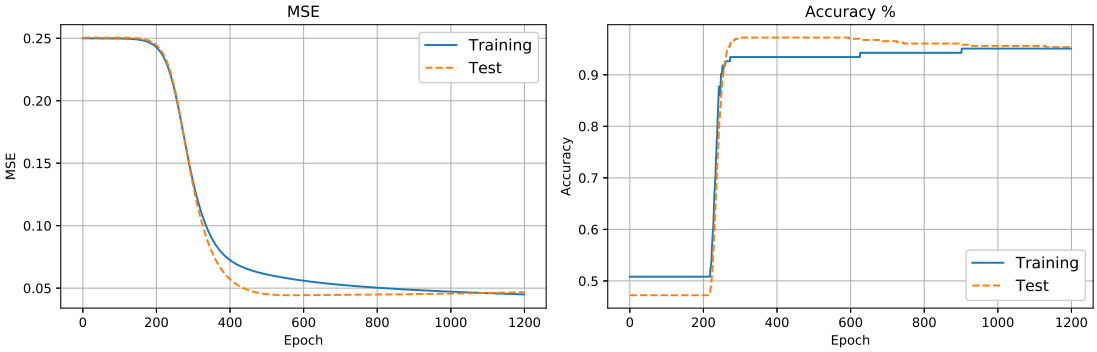
Table 3.1: MSE and Accuracy average prediction results obtained for the three MONK's tasks with 10 runs on training (TR) and test (TS) sets. Note $Monk_3\,(r)$ refers to the MONK 3 task with regularization.



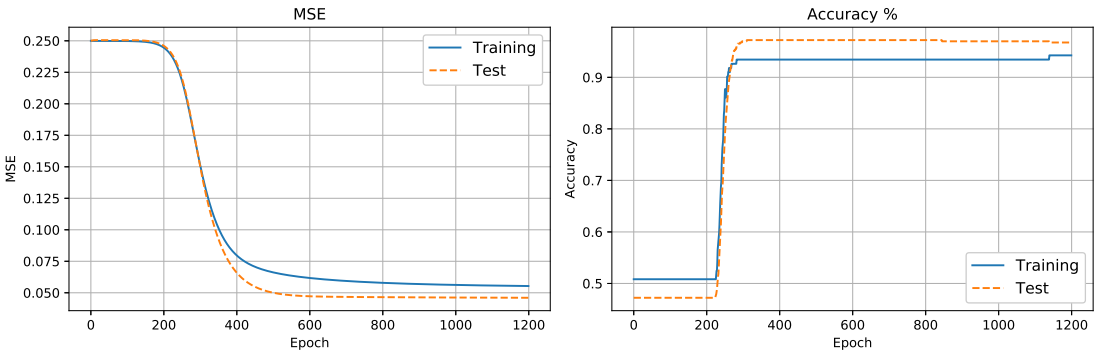
(a) MONK 1



(b) MONK 2



(c) MONK 3 without regularization



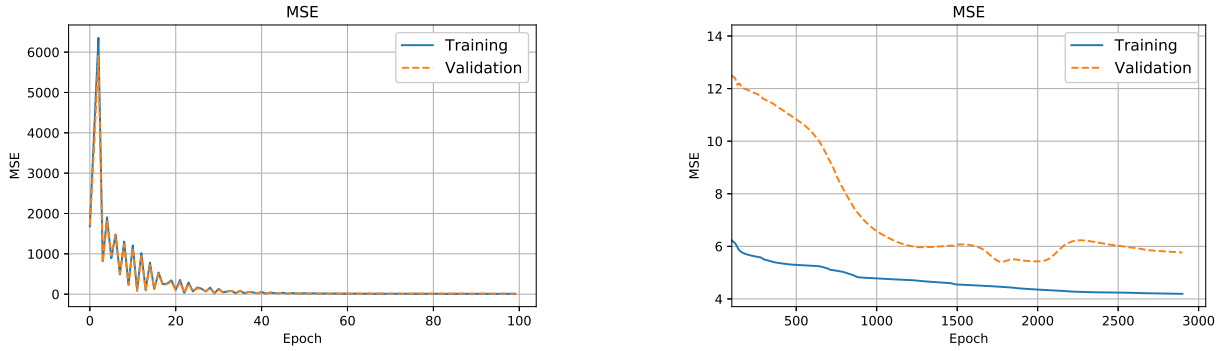
(d) MONK 3 with regularization

Figure 3.1: Plots of the MSE and Accuracy for the three MONK's benchmarks.

3.2 Cup Results

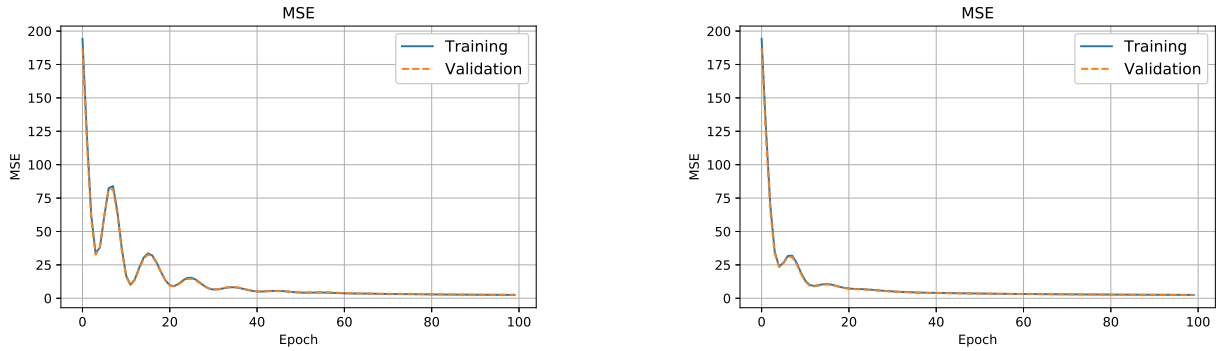
For the experiments on the CUP dataset a shallow architecture was used: 10 (input units) - $\#units$ (hidden units) - 2 (output units). Where $\#units$ is a hyperparameter. We used the design set for all the stages that led to the model selection. Then, the internal test set was used for the model assessment.

Screening Phase:
To identify a good hyperparameters range a *preliminary screening phase* has been performed by using the 15% of the design set as the *validation set*, and the remaining part as the *training set*. During these trials, we manually changed the model's hyperparameters and observed the resulting learning curves. We tried to achieve *overfitting*, without using regularization approaches, and we noticed that, for a wide range of a different number of units, it is perfectly observable in 25000 epochs and so we decided to set the maximum number of epochs at 30000. Then, we saw the effects of the regularization on these overfitting models: with too small λ values (e.g. 0.00001), the model still went into overfitting, while with too large λ values (e.g. 0.0015), it risked going underfitting. We also noticed that, with too high η values (e.g. 0.3), the models often overfit, and the curves have an initial instability (Fig. 3.2). Finally, as regards the Nesterov momentum we have seen that, with the right α values, it can stabilize the curve (Fig. 3.3) in the initial part (in the first 100 epochs) and accelerate convergence. On the other hand, with too large α and η values, it can make the model overfits and shows instability. Moreover, we noticed that when the model goes overfitting, the validation curve, in the 30000 epochs, goes up without ever going down. Thus, as early stopping, we chose a small $\varepsilon_{GL} = 0.009$ with $s_{UP} = 200$. Some plots of underfitting and overfitting are shown in Fig. 3.4.



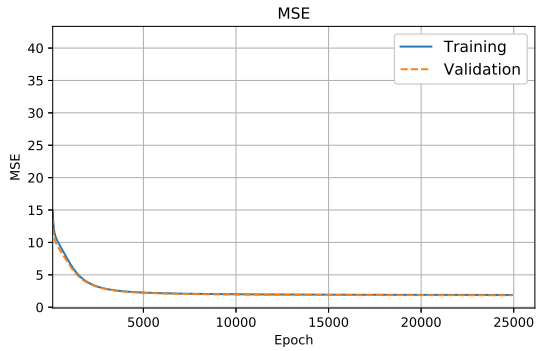
(a) From 0 to 100 epochs. **(b)** From 100 to the end of training phase.

Figure 3.2: Too high learning rate value can lead to instability in the training phase (e.g. 0.3 in this model).

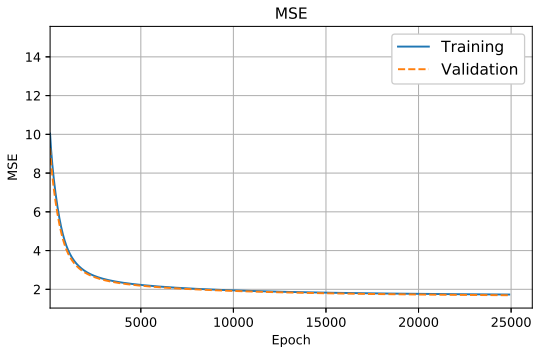


(a) $\eta = 0.01$ and No Momentum. **(b)** $\eta = 0.01$ and Nesterov Momentum with $\alpha = 0.9$.

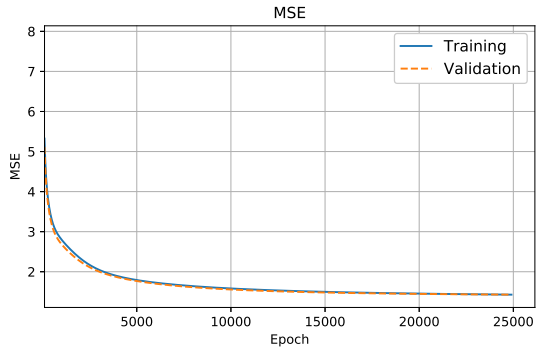
Figure 3.3: Nesterov momentum can make learning curves more stable. Model with #units=50 and $\lambda = 0$.



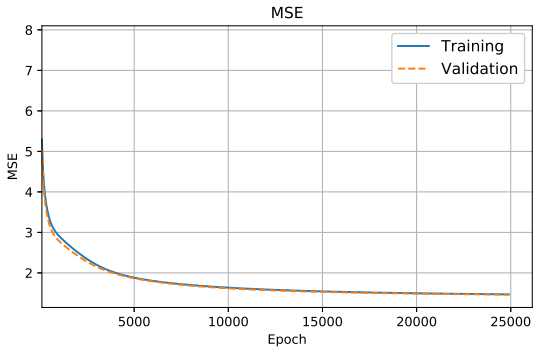
(a) $\#units = 10,\ \eta = 0.002,\ \alpha = 0.1,\ \lambda = 0$



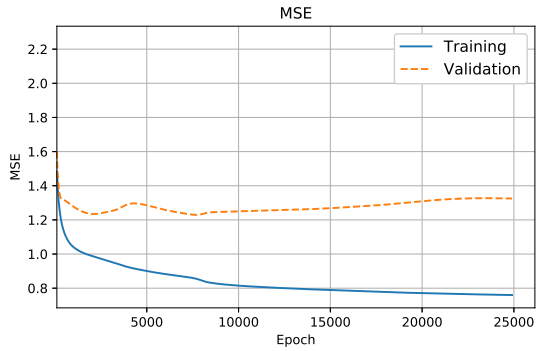
(b) $\#units = 20,\ \eta = 0.002,\ \alpha = 0.3,\ \lambda = 0$



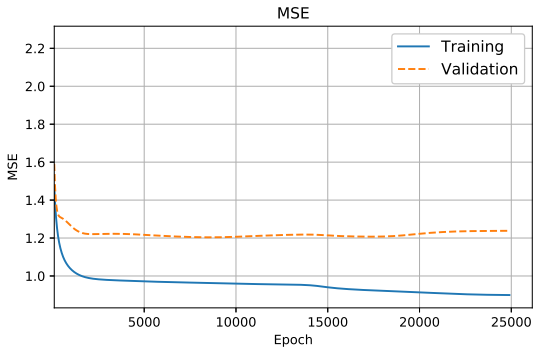
(c) $\#units = 50,\ \eta = 0.007,\ \alpha = 0.2,\ \lambda = 0$



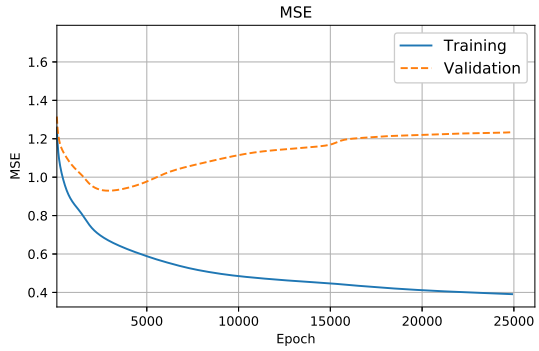
(d) $\#units = 80,\ \eta = 0.005,\ \alpha = 0.4,\ \lambda = 0$



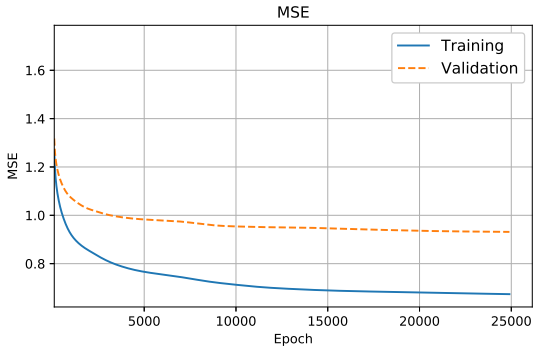
(e) $\#units = 100.\ n = 0.07.\ \alpha = 0.8.\ \lambda = 0$



(f) $\#units = 100.\ n = 0.07.\ \alpha = 0.8.\ \lambda = 0.0002$



(g) $\#units = 50,\ \eta = 0.06,\ \alpha = 0.9,\ \lambda = 0$



(h) $\#units = 50,\ \eta = 0.06,\ \alpha = 0.9,\ \lambda = 0.0007$

Figure 3.4: Examples of underfitting (3.4a, 3.4b, 3.4c, 3.4d), overfitting (3.4e, 3.4g) models and their regularized versions (3.4f, 3.4h). Plots have been generated from the 100th epochs to better visualize the behaviour of the learning curves.

Large Grid Search:
From the screening phase we were able to isolate some ranges and run a grid search with the hyperparameters reported in Tab. 3.2 using a 4-fold CV. Tab. 3.3 shows the six best hyperparameter combinations found using the estimate of the MEE on the validation set as a goodness criterion. We can see that the best performances were obtained with a number of units from 40 to 100, learning rate from 0.03 to 0.06, momentum from 0.6 to 0.9 and regularization from 0.0001 to 0.0005. Based on these results, we can shrink the hyperparameters ranges and perform a finer grid search with the same criteria and the ranges reported in Tab. 3.4.

Hyperparameters	**Values range**
# units	20, 40, 80, 100, 150
η	0.03, 0.06, 0.08, 0.098
α	0.2, 0.4, 0.6, 0.8, 0.9
λ	0.0001, 0.0005, 0.0009, 0.0013
Epochs	30000
Early Stop	(ε_{GL} = 0.009, s_{UP} = 200)

Table 3.2: Range of Hyper-parameters for the large grid search

Scoring Large Grid Search with 4 Folds								
	# units	η	α	λ	**MEE (TR)**	**MEE (VL)**	**MEE (Int. TS)**	**TR Time (s)**
cup L1	80	0.06	0.6	0.0001	0.878 ± 0.040	1.076 ± 0.035	1.031 ± 0.015	186.4 ± 65.7
cup L2	40	0.06	0.9	0.0005	0.879 ± 0.046	1.084 ± 0.045	1.016 ± 0.012	52.9 ± 30.1
cup L3	80	0.03	0.8	0.0001	0.903 ± 0.048	1.090 ± 0.023	1.041 ± 0.015	160.4 ± 71.2
cup L4	40	0.03	0.9	0.0001	0.887 ± 0.043	1.094 ± 0.027	1.050 ± 0.021	24.9 ± 8.1
cup L5	80	0.03	0.9	0.0001	0.870 ± 0.042	1.094 ± 0.032	1.042 ± 0.033	49.5 ± 14.7
cup L6	100	0.03	0.9	0.0001	0.859 ± 0.107	1.095 ± 0.016	1.061 ± 0.019	94.7 ± 77.2

Table 3.3: Best six hyperparameter combination obtained from the large grid search.

Finer Grid Search:
Tab. 3.5 shows the top three best hyperparameter combinations found in the finer grid search.

Hyperparameters	**Values range**
# units	40, 50, 60, 70, 80, 90, 100
η	0.03, 0.034, 0.038, 0.042, 0.046, 0.050, 0.054, 0.058, 0.06
α	0.6, 0.7, 0.8, 0.99
λ	0.0001, 0.0002, 0.0003, 0.0004, 0.0005
Epochs	30000
Early Stop	($\varepsilon_{GL} = 0.009$, $s_{UP} = 200$)

Table 3.4: Hyperparameters ranges for the finer grid search.

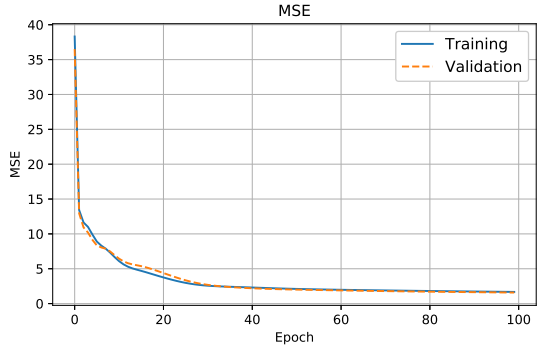
Scoring Finer Grid Search on 4 Folds								
	# units	η	α	λ	**MEE (TR)**	**MEE (VL)**	**MEE (Int. TS)**	**TR Time (s)**
cup F1	50	0.058	0.9	0.0005	0.865 ± 0.048	1.061 ± 0.015	1.013 ± 0.020	49.10 ± 21.07
cup F2	50	0.058	0.9	0.0004	0.867 ± 0.045	1.065 ± 0.028	1.028 ± 0.016	31.52 ± 13.76
cup F3	70	0.054	0.9	0.0004	0.905 ± 0.070	1.068 ± 0.042	1.018 ± 0.015	78.88 ± 91.18

Table 3.5: Best three hyperparameter combination obtained from the finer grid search.

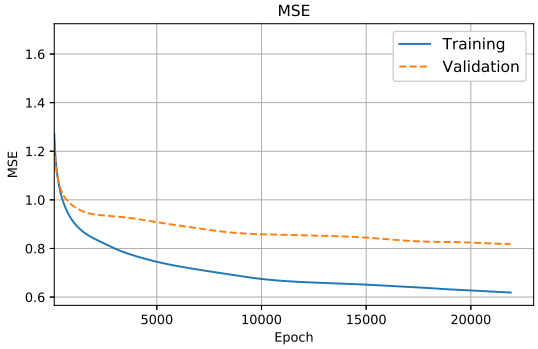
Final model selection:
We choose the hyperparameter combination of *cup F1* as hyperparameters of the final model because, in Tab. 3.5, it is the one with the lowest estimate of the MEE on the validation set. So, we re-trained *cup F1* on 90% of the design set, leaving the remaining 10% as a validation set for early stopping techniques. We repeated the re-training for ten trials to avoid bad weights initialization and we selected the model with the minimum MEE validation error. Tab. 3.6 shows the MEE values of the retraining (training and validation), the result of the *Model Assessment* performed on the *internal test set*, and the computing time during the final re-training. Also, the learning curves for training and validation have been included in Fig. 3.5, where we can observe smooth behaviour during training.

# units	η	α	λ	MEE (TR)	MEE (VL)	MEE (Internal TS)	TR Time (s)
50	0.058	0.9	0.0005	0.85648	0.960216	0.99397	54

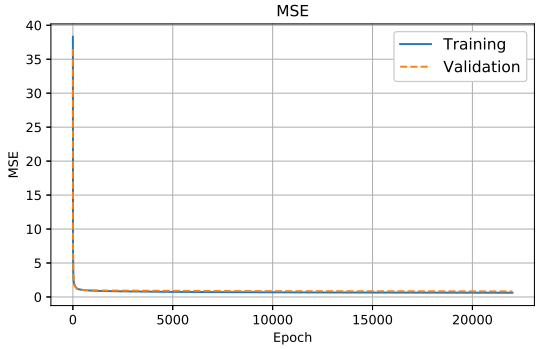
Table 3.6: Results of the final model retrained on the 90% design set and the Model Assessment on the internal test set.



(a) From 0 to 100 epochs.



(b) From 100 to 22005 epochs.



(c) Full plot with all the epochs.

Figure 3.5: Final model plots. To better observe the learning curves behaviors, in addition to the complete plot (3.5c), the plots from 0 to 100 epochs (3.5a) and 100 to 22005 epochs (3.5b) (when the training phase ends) have been added.

Finally, to generate the prediction on the "blind test set", we re-trained the model on the initial training dataset (design set plus internal test set), excluding 10% of the data as validation for the same reason as above. The results are in `isanet_ML-CUP19-TS.csv`.

4. Conclusions

During the development of the IsaNet, we have become increasingly familiar with the theory learned in class. In particular, we enjoyed the screening phase, because let us better understand the real influence that the hyperparameters have in the training phase. For example, we fully grasped the power of Tikhonov regularization after implementing and testing it in various cases. We mainly focused on writing a reliable library, so that we can use it in future works as well, and we preferred to work with quite standard neural networks for our first approach to Machine Learning. Despite this, we were pleased to insert something extra in the appendices.

As regards future work: it would be interesting to see how the result on the CUP varies using Super Accelerated Nesterov, or trying neural networks with more than one hidden layer. We would also have liked to try the Conjugate Gradient and the Quasi-Newton, but for lack of time, we had to give it up.

Acknowledgments

We agree to the disclosure and publication of my name, and of the results with preliminary and final ranking.

References

[1] Boris Polyak. Some methods of speeding up the convergence of iteration methods. *Ussr Computational Mathematics and Mathematical Physics*, 4:1–17, 12 1964.

[2] I. Sutskever, J. Martens, G. Dahl, and G. Hinton. On the importance of initialization and momentum in deep learning. *30th International Conference on Machine Learning, ICML 2013*, pages 1139–1147, 01 2013.

[3] Goran Nakerst, John Brennan, and Masudul Haque. Gradient descent with momentum — to accelerate or to super-accelerate?, 2020.

[4] Dheeru Dua and Casey Graff. UCI machine learning repository, 2017.

[5] *Intel Math Kernel Library. Reference Manual*. Intel Corporation, Santa Clara, USA, 2009. ISBN 630813-054US.

[6] Lutz Prechelt. *Early Stopping — But When?*, pages 53–67. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

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

[8] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems*, page 2012.

[9] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *In Proceedings of the International Conference on Artificial Intelligence and Statistics (AISTATS'10). Society for Artificial Intelligence and Statistics*, 2010.

Appendices

A. Comparisons between Momentum Methods

The delta rule, at iteration $i+1$, using the gradient descent algorithm with the Classic heavy-ball Momentum [1], Nesterov [2] and Accelerated Nesterov Momentum [3] can be summarized in this way:

$$\Delta^i = \alpha\Delta^{i-1} - \eta\nabla L(w^i + \sigma\Delta^{i-1})$$

$$w^{i+1} = w^i + \Delta^i$$

When $\sigma = 0$, the delta rule uses the heavy-ball method, and when $\sigma = \alpha$, the Nesterov momentum is used. In [3] is shown that can be advantageous to use values of σ larger than $\alpha \approx 1$. Instead of using the gradient at an estimated point one step ahead, the gradient is computed at a much further estimated point. This corresponds to an extension of Nesterov and that is why it is called *Accelerated Nesterov Momentum*. Figure A.1 shows the effect of momentum methods.

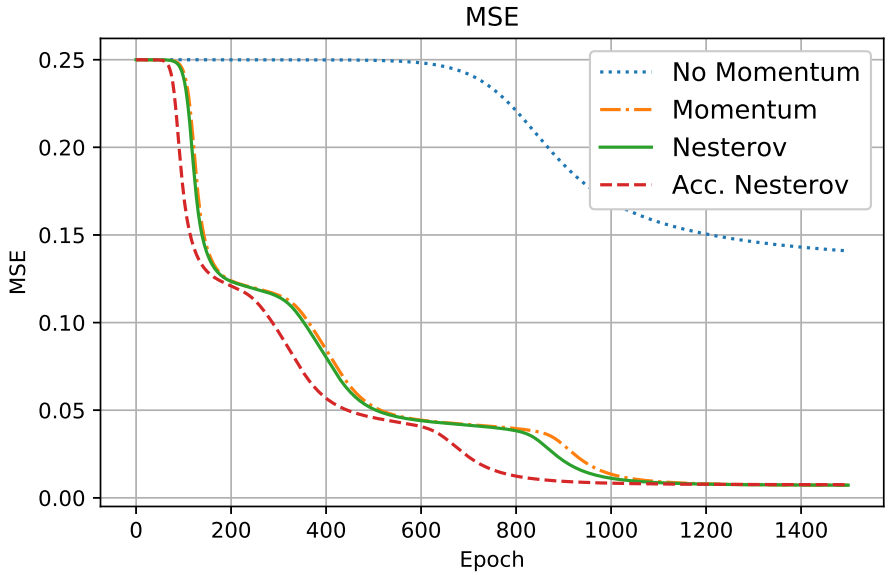


Figure A.1: Training Learning curves using Monk 1 as dataset with different momentum methods. To compare the learning curves and correctly observe the convergence speed, all the models have been trained starting from the same initial weights.

B. Code Examples

Listing 1: A low level API example on the Monk 1 dataset.

```
# ...
from isanet.model import Mlp
from isanet.optimizer import SGD, EarlyStopping
from isanet.datasets.monk import load_monk
import numpy as np

X_train, Y_train = load_monk("1", "train")
X_test, Y_test = load_monk("1", "test")

#create the model
model = Mlp()
# Specify the range for the weights and lambda for regularization
# Of course can be different for each layer
kernel_initializer = 0.003
kernel_regularizer = 0.001

# Add many layers with different number of units
model.add(4, input= 17, kernel_initializer, kernel_regularizer)
model.add(1, kernel_initializer, kernel_regularizer)

es = EarlyStopping(0.00009, 20) # eps_GL and s_UP

#fix which optimizer you want to use in the learning phase
model.setOptimizer(
    SGD(lr = 0.83,          # learning rate
        momentum = 0.9,     # alpha for the momentum
        nesterov = True,    # Specify if you want to use Nesterov
        sigma = None        # sigma for the Acc. Nesterov
    ))

#start the learning phase
model.fit(X_train,
          Y_train,
          epochs=600,
          #batch_size=31,
          validation_data = [X_test, Y_test],
          es = es,
          verbose=0)

# after trained the model the prediction operation can be
# perform with the predict method
outputNet = model.predict(X_test)
```

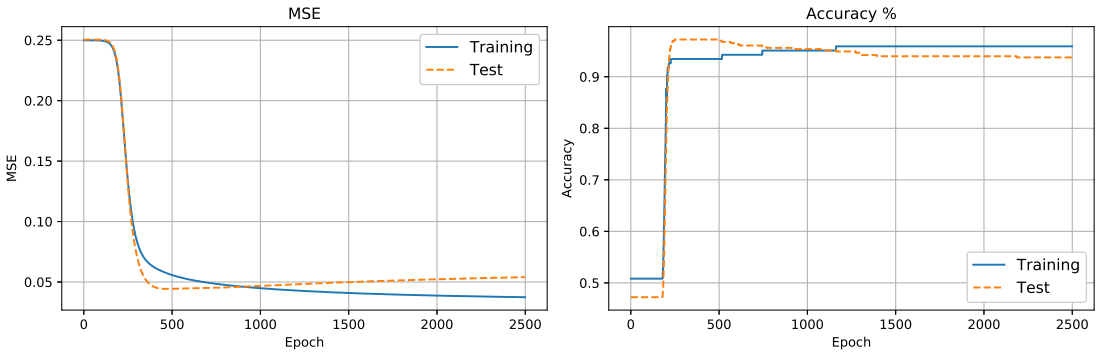
Listing 2: A high level API example on the Monk 1 dataset.

```
from isanet.neural_network import MLPClassifier
from isanet.datasets.monk import load_monk

X_train, Y_train = load_monk("1", "train")
X_test, Y_test = load_monk("1", "test")

mlp_c = MLPClassifier(X_train.shape[1],              # input dim
                      Y_train.shape[1],              # out dim
                      n_layer_units=[4],             # topology
                      activation="sigmoid",          # activation hidden layer
                      kernel_regularizer=0.001,      # l2 regularization term
                      max_epoch=600,                 # Max number of Epoch
                      learning_rate=0.83,            # learning rate
                      momentum=0.9,                  # momentum term
                      nesterov=True,                 # if Nesterov
                      sigma=None,                    # sigma Acc. Nesterov
                      early_stop=None,               # define the early stop
                      verbose=0)                     # verbosity
mlp_c.fit(X_train, Y_train, X_test, Y_test)
outputNet = mlp_r.predict(X_test)
```

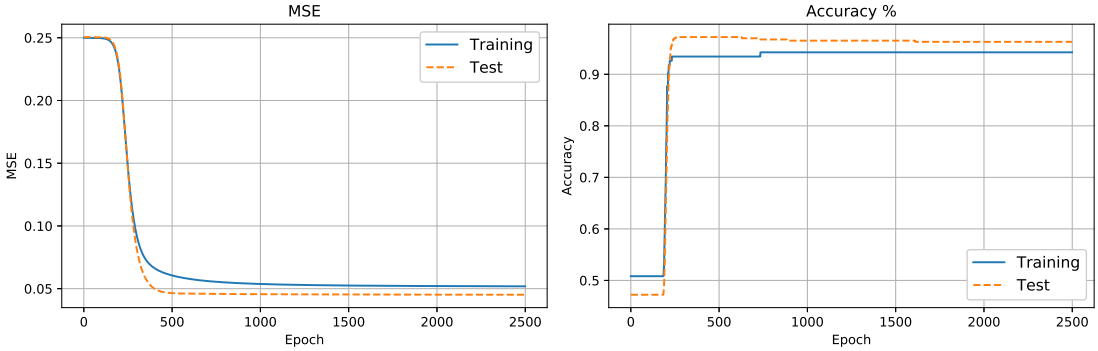
Listing 3: Model Selection API example on the CUP dataset.

```
from isanet.model_selection import Kfold, GridSearchCV
from isanet.neural_network import MLPRegressor

dataset = np.genfromtxt('CUP/ML-CUP19-TR_tr_vl.csv',delimiter=',')
X_train, Y_train = dataset[:,:-2], dataset[:,-2:]

grid = {"n_layer_units": [[38], [20, 32]], #[20, 32] means two hidden layer
         "learning_rate": [0.014, 0.017],
         "max_epoch": [13000, 1000],
         "momentum": [0.8, 0.6],
         "nesterov": [True, False],
         "sigma": [None, 0.8, 0.6, 2, 4]
         "kernel_regularizer": [0.0001],
         "activation": ["sigmoid"],
         "early_stop": [EarlyStopping(0.00009, 20), EarlyStopping(0.09, 200)]}

mlp_r = MLPRegressor(X_train.shape[1], Y_train.shape[1])
kf = Kfold(n_splits=5, shuffle=True, random_state=1)
gs = GridSearchCV(estimator=mlp_r, param_grid = grid, cv = kf, verbose=2)
result = gs.fit(X, Y) # dict with keys as column headers and values as columns
```

C. MONKS



(a) MONK 3 with no regularization



(b) MONK 3 with regularization

Figure C.1: Plot of the MSE and accuracy for the MONK 3. In Fig. C.1a, the model without regularization goes into *overfitting* just before 1000 epochs. Indeed the generalization error measured over the test examples increases, even as the error over the training examples continues to decrease. In Fig. C.1b is shown how the regularization prevents overfitting.

Task	Topology	Batch size	η	α	λ	MSE(TR/TS)	Accuracy(TR/TS)(%)
MONK 1	17→4→1	1	0.4	0.8	0	0.00016 / 0.00023	100% / 100%
MONK 1	17→10→1	1	0.4	0.7	0	0.00012 / 0.00017	100% / 100%
MONK 1	17→4→1	31	0.6	0.7	0	0.00115 / 0.00182	100% / 100%
MONK 2	17→3→1	1	0.1	0.4	0	0.00205 / 0.00235	100% / 100%
MONK 2	17→10→1	1	0.1	0.4	0	0.00197 / 0.00229	100% / 100%
MONK 2	17→3→1	68	0.1	0.4	0	0.00301 / 0.00349	100% / 100%
MONK 2	17→3→1	169	0.01	0.9	0	0.00970 / 0.01217	100% / 100%
MONK 3 (no reg.)	17→4→1	1	0.01	0.6	0	0.04519 / 0.04536	95% / 95%
MONK 3 (no reg.)	17→10→1	1	0.01	0.5	0	0.04828 / 0.04398	95% / 95%
MONK 3 (no reg.)	17→4→1	61	0.6	0.3	0	0.04904 / 0.04483	94% / 96%

Table C.1: Preliminary cases discovered during the screening phase of the MONKS dataset. Normal momentum was used for online and mini-batch, while Nesterov was used for full batch. Also we used the sigmoid as activation function.

D. Cup Prepocessing Phase

During this phase, we analyzed the CUP dataset composed of 20 inputs and 2 targets. From the distributions of the variables (Fig. D.1) and correlation matrix (Fig. D.2) it can be seen that many variables are strongly correlated to each other: A-I, B-U, C-T, D-P, E-L, F-M, G-S, H-R, N-O and Q-V. Therefore, by eliminating the redundant features (A, B, C, D, E, F, G, H, N, Q), those with correlation equal to 1, it is possible to reduce the dataset to 12 attributes (10 inputs and 2 targets).

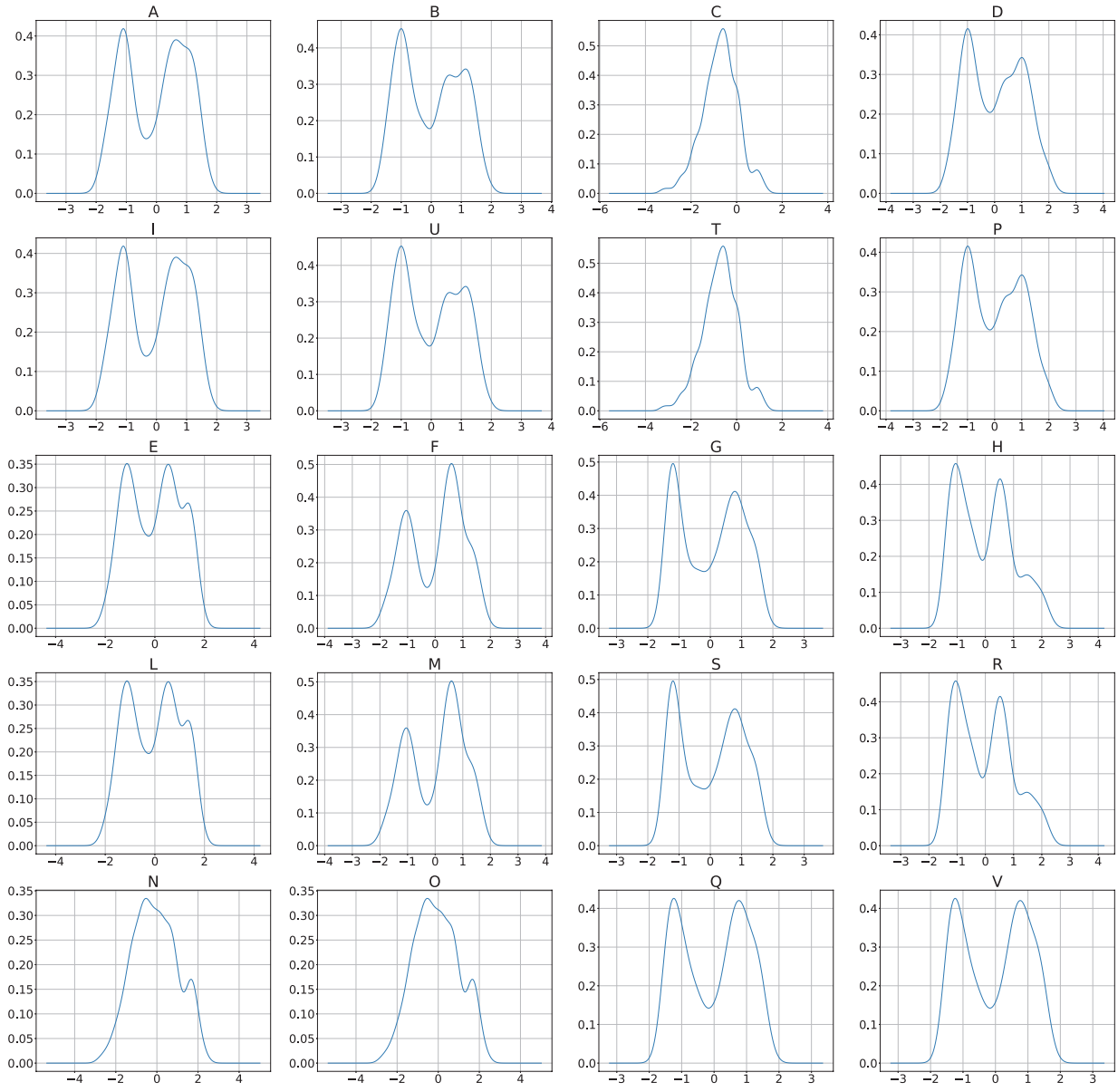


Figure D.1: Distributions of the features, excluded the one representing the id record and those relating to the target.

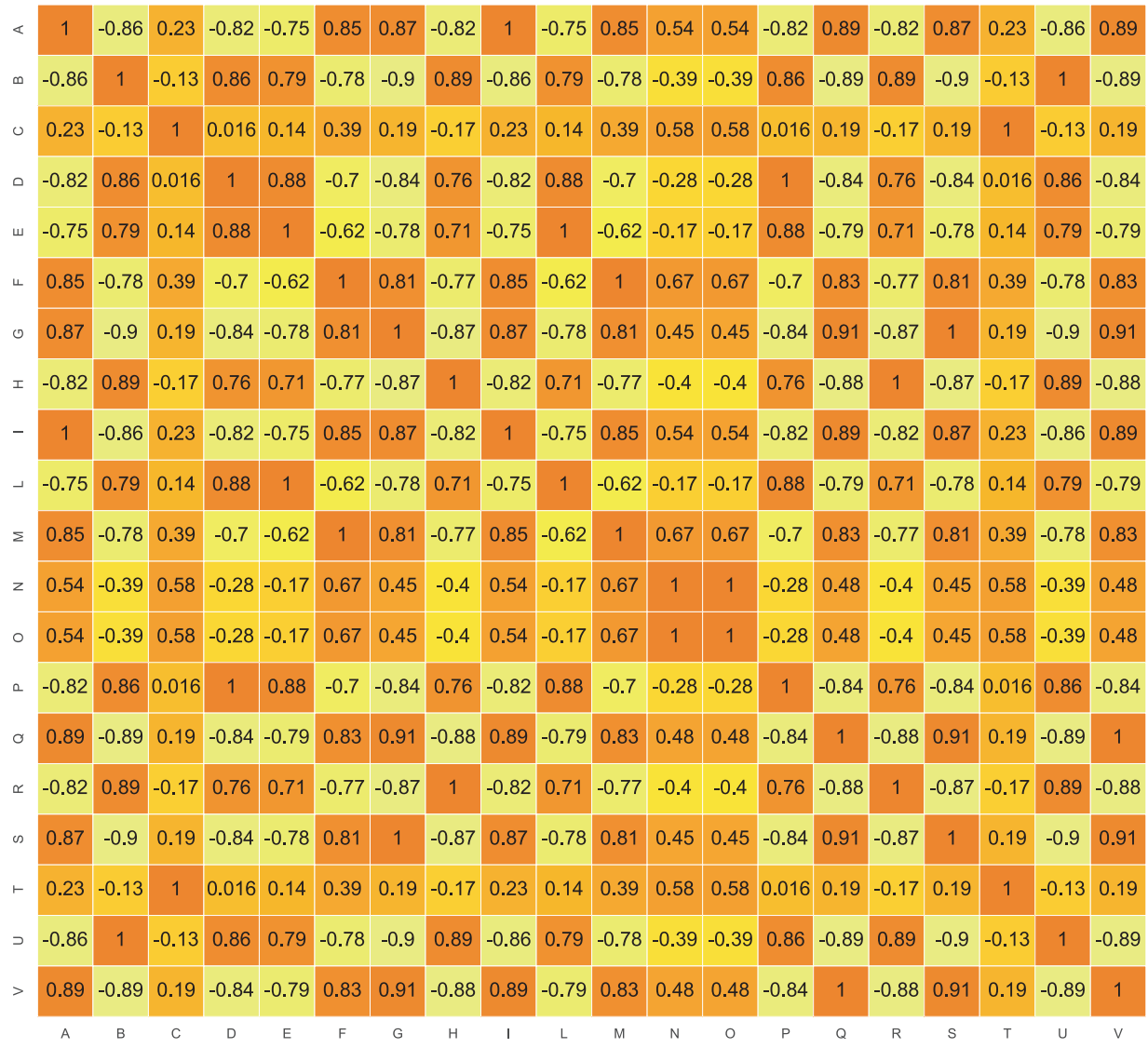


Figure D.2: Correlation matrix.