Machine Learning Project report

ML course a.a. 2019/2020 (654AA)

project type A

Lucio Messina
505134
lucio.messina@autistici.org

Paolo Labruna
532622
pielleunipi@gmail.com

June 15, 2020

Abstract

This short report describes the Python implementation of an artificial neural network. We performed the model selection through a grid search on various hyper-parameters which explored thousands of configurations by means of k-fold cross validation. The neural network was tested on the monk datasets and then applied to the cup dataset through an ensemble of 10 models.

Lecturer Feedback

The report is well written although a little bit too concise on the experiment part. The project is well conducted apart from these criticisms:

- We tested 5000 configurations of hyper-parameters, which is a lot. We could have tested some coarse grid and refined it according to the result of the first grid search.
- The learning curves are too unstable. We could have searched for more stable models.
- Our prediction on the blind test set results was way too much optimistic. We could have used more data for the internal test set or used a nested cross-fold validation.

1 Introduction

The project consists in developing from scratch an Artificial Neural Network simulator which is capable to solve both regression and classification tasks. Apart from implementing the model and testing its behavior under different conditions, the aim of the project was to develop a rigorous method to find the best configuration and to validate the proposed solution. To achieve this we implemented some utilities to perform a grid search on its hyper-parameters using a k-fold cross validation (described in section 2). The model was used to solve the well-known monk problem (see Thrun et al. (1991)) and to predict the output coordinates given the real data sensors of the ML-cup (see Micheli (2019)). The results of the model selection phase and the experiments are described in section 3.

2 Method

The software was developed using the `Python` programming language, the `numpy` library and some minor utilities from the library `sklearn`: no off-the-shelf machine learning library, tool or model were used for this project. Paragraph 2.1 describes the relevant implementation and design choices we took. We implemented a fully connected feed forward Neural Network that learns its weights through backpropagation using the Stochastic Gradient Descent technique (SGD) over multiple epochs. We also implemented some variations and improvements such as: regularization, momentum, different kind of weights initialization, different activation functions, different weight updating policies (batch, minibatch or online), different way to decide when to stop the training phase and different learning rate behavior (constant, adaptive or linear). Each one of the implemented features can be activated and/or fine tuned through a different hyper-parameter: paragraph 2.2 lists all the hyper-parameters along with their description. Each configuration of hyper-parameters is a different **model** with different performances on a specific task. Paragraph 2.3 describes how the model that performs better on the ML-CUP dataset was selected. No preprocessing was made on the ML-CUP dataset.

2.1 Implementation choices

We developed the class `BaseNeuralNetwork` that implements a Neural Network capable of optimizing any given loss through backpropagation. Its hidden layers number and sizes can be specified using the hyper-parameter `hidden_layer_sizes` although its topology is always fully connected. The activation functions of the hidden and output layers can also be specified using an hyper-parameter: the module `functions` implements all the activation and loss functions and their derivatives used by the neural network. The parameters of the constructor of `BaseNeuralNetwork` are the very same as the ones of the classes in the `sklearn.nerual_network` module. `BaseNeuralNetwork` also implements the functions `fit(X, y)`, `predict(X)` and others as the `sklearn` transformers.

The two derived classes `MLPRegressor` and `MLPClassifier` implement a Neural Network capable of solving regression and binary classification tasks respectively:

- `MLPRegressor` optimizes the *Mean Squared Loss* using a linear activation function for the output units.
- `MLPClassifier` optimizes the *Multiclass Logarithmic Loss* (crossentropy) using the normalized *tanh* activation function for its sole output unit. Its function `predict()` always outputs either 0 or 1 for any input pattern based on whether the output value is greater than 0.5 or not.

From a software engineering perspective our implementation of `MLPRegressor` and `MLPClassifier` are completely interchangeable with the ones of `sklearn.neural_network.MLPRegressor` and `sklearn.neural_network.MLPClassifier`. However, we did not implement some parameters that are out of scope for this project (such as Nesterovs Momentum and other solver techniques than Stochastic Gradient Descent) and we implemented some features not included in the `sklearn` models (such as different weights initialization functions and the *linear* learning rate behavior).

2.2 Hyper-parameters overview

- `hidden_layer_sizes`: number and size of the hidden layers.
- `activation`: name of the hidden layers activation function. Can be one of *relu*, *identity*, *threshold*, *logistic*, *tanh*, *zero_one_tanh*. The last one is the *tanh* function normalized such that its output range is $(0; 1)$.
- `lambda`: regularization coefficient.
- `batch_size`: number of patterns after which the weights are updated during the training phase.
- `max_iter`: number of epochs after which the training is aborted if the learning has not converged yet.
- `shuffle`: whether to shuffle the input samples before each epoch.
- `alpha`: momentum coefficient.
- `learning_rate`: learning rate behavior. Can be *constant*, *invscaling*, *linear* or *adaptive*. When `learning_rate` is *linear* or *invscaling* the learning rate gradually decreases over epochs linearly and according to the inverse function respectively. When `learning_rate` is *adaptive* the learning rate is halved each time the loss does not decrease for two consecutive training epochs.
- `eta`: initial value for the learning rate.
- `pow`: exponent for inverse scaling learning rate (*invscaling*). The learning rate at epoch t is $\frac{\text{eta}}{t^{\text{pow}}}$.
- `tol`: tolerance for the optimization. The training phase is stopped when the loss is not improving by at least `tol` for `n_iter_no_change` consecutive epochs.
- `n_iter_no_change`: number of epoch after to not met `tol` improvement in training loss.
- `early_stopping`: if `True` a part of the training data is set aside and not used for training. The learning phase is stopped when the loss on that data does not improve, rather than the loss on the whole training set.
- `validation_fraction`: the proportion of training data to set aside when `early_stopping` is `True`.
- `weights_init_fun`: distribution from which the weights of the neural network are initialized at the beginning of the training phase. Can be *random_normal* (gaussian distribution) or *random_uniform* (uniform distribution).
- `weights_init_value`: when `weights_init_fun` is *random_normal* this it is the mean of the random distribution from which the initial weights are drawn; otherwise (*random_uniform*), it provides the bounds of the symmetric distribution around zero.

2.3 Validation schema

We split the ML-CUP dataset in two parts: the **development set** (90%) and the **internal test set** (10%). The development set was used to find the best model while the internal test set was used to estimate its generalization performances on unseen data.

The module `utility` implements an utility to perform a **randomized grid search** on a set of hyper-parameter values (see Bergstra et al. (2012)): each randomly-selected configuration is evaluated using a **5-fold cross validation** on the development set. This procedure consists in splitting five times the development set into training set (80%) and validation set (20%), each time using a different part for the validation set, then training a model with the selected configuration of hyper-parameters on the training set and evaluating it on the validation set. For each configuration the average *Mean Euclidean Loss* on the training set and validation set is reported as well as their standard deviations. Configurations that lead to overflows and other numerical errors due to the instability of the hyper-parameters are discarded.

The 10 configurations that achieve the lowest *Mean Euclidean Loss* on the validation set are selected as best models; an ensemble of those 10 best models is selected as final model. The module `ensembler` implements the ensemble model which computes the output predictions by averaging the predictions of its constituent models: this is a simple but very effective technique as show by Galton (1907). The predictions for the internal test set were computed only once by the ensemble model trained on the whole development set: the loss on the internal test set should give a fair estimation of the error on unseen data without overfitting risk because the internal test set was never used during the model selection phase. The ensemble of the 10 best models trained on the whole ML-CUP dataset was used to predict the output for the blind test set.

3 Experiments

3.1 Monk

Applying the one-hot encoding technique led us to have 17 binary input values. We opted for a network with one hidden layer composed of 15 hidden units for all three monk tasks. We used the *relu* (Rectified Linear Unit) activation function for the hidden layer and the sigmoidal activation function for the output layer. We computed the loss through the *Logarithmic Loss* function by a full batch gradient descend. The monk tasks turned out to be particularly sensitive to weight initialization, we used values chosen from a random uniform distribution between -0.1 and 0.1 for monk 1 and monk 3 and between -0.25 and 0.25 for the monk 2.

Task	**eta**	**alpha**	**lambda**	**Loss(TR/VL)**	**Accuracy(TR/VL)**
Monk 1	0.8	0.8	0	0.0036/0.0069	100%/100%
Monk 2	0.8	0.8	0	0.00311/0.0041	100%/100%
Monk 3	0.8	0.8	0	0.0144/0.1479	100%/94.65%
Monk 3 (reg.)	0.8	0.8	0.005	0.1498/0.1378	94.75%/96.85%

Table 1: Average of the loss and accuracy obtained on ten trials. The model for Monk 3 (reg.) has a regularization coefficient lambda greater than zero.

Figure 1 shows the learning and accuracy curves for one trial on each one of the three monk datasets; figure 1d shows the same plots for the monk 3 dataset applying regularization: it is worth noting that higher accuracy can be achieved on the test set of the monk 3 task when the regularization is applied despite worse training performances. With our settings the learning converged after about 150 epochs (on average) on the monk 1 and 2 tasks. After the same number of epochs the validation accuracy reaches the trainig accuracy on the monk 3 dataset: training more than 150-200 epochs led to overfitting. We included only one learning curve plot for each dataset because the curves of different trials on the same dataset are very similar.

3.2 CUP

We used the **development set** (90% of the ML-CUP dataset) to perform a grid-search on the hyper-parameters values to select the best model. We tested 5000 different configurations of hyper-parameters: numerical hyper-parameters such as `eta` and `lambda` were drawn from a random uniform distribution which extremes are shown in the table 2 while categorical hyper-parameters such as `hidden_layer_sizes` and `early_stopping` were drawn with uniform probability from a list of fixed alternatives also shown in table 2. The values of a few hyper-parameters (namely `shuffle`, `max_iter`, `pow`, `tol`, `n_iter_no_change` and `validation_fraction`) were fixed either to reduce the space of the possible configurations or because after a few preliminary trials it turned out that their value was almost irrelevant for the results. For each configuration we performed a 5-fold cross validation on the development set reporting the average over the 5 folds of the *Mean Euclidean Error* on the training and validation sets along with their standard deviations. This model selection phase took around 17 hours using 4 cores of an amd Ryzen 3 @2.5 GHz processor. All the measured standard deviations are below 10% of the corresponding measure thus we did not include them in this report.

We selected the 10 best models in terms of *Mean Euclidean Error* on the validation set: their hyper-parameters and performances are reported in table 4. All the best performing networks are quite big (two hidden layers composed of 50 units each); also in all the models `early_stopping` is set to False: this means that better results can be achieved if all of the available data is used to train the network rather than saving 10% of the samples to implement early stopping, maybe the overfitting risk is kept under control just by the value of `lambda`. All the models have low batch size (1 to 5) thus the loss decrease over the epochs is a bit unstable. The *relu* activation function turned out to be less effective, in fact the 10 best models use either *tanh* or *logistic*. The values of the other numerical hyper-parameters (`alpha`, `lambda`, `eta`) are quite spread apart in their respective ranges, making the 10 models quite different despite the similar results: this is an important point that we considered when deciding how to chose our final model. The average number of epoch to reach convergence is also quite different between the models, ranging from 35 to 110.

Our final model is an ensemble of these 10 best models: since the validation *MEE* is not a good estimate of the real test error we retrained all the models on the whole development set and computed the predictions on the **internal test set** (the unused 10% of the ML-CUP dataset). The predictions of the ensemble model are just the average of the predictions of the constituent models.

To compare the performances of the 10 best models with the performances of the ensemble model we computed the *Mean Euclidean Error* on the internal test set on the predictions of both the constituent models and the ensemble model.

Numerical hyper-parameter	**range** $[min, max]$
`lambda` (regularization coefficient)	$[0, 0.05]$
`eta` (learning rate initial value)	$[0.001, 0.1]$
`alpha` (momentum coefficient)	$[0, 0.9]$
`weights_init_value`	$[0.2, 0.8]$
Categorical hyper-parameter	**possible options**
`hidden_layer_sizes`	$\{(10, 10)$, $(20,)$, $(50,)$,$(100,)$, $(50, 50)\}$
`weights_init_fun`	{"random_uniform", "random_normal"}
`learning_rate`	{'constant', 'adaptive', 'linear'}
`batch_size`	{1, 5, 10, 50, 100, 200, 500, 'batch'}
`early_stopping`	{`True`, `False`}
`activation`	{'relu', 'tanh', 'logistic'}
Fixed hyper-parameter	**used value**
`shuffle`	`True`
`max_iter`	500
`pow`	0.5
`tol`	0.0001
`n_iter_no_change`	10
`validation_fraction`	0.1

Table 2: Ranges, lists of options and fixed values of the used hyper-parameters. The role of each hyper-parameter is described in section 2.2. The alternatives for `hidden_layer_sizes` are tuples in which each value N refers to an hidden layer with N units. Networks with just one hidden layer are reported as $(N,)$: this is the standard representation of the singleton tuples in the Python programming language.

As shown by the table 3 the *Mean Euclidean Error* of the ensemble model (0.92241244) is significantly lower than the one of its constituents; this result is also theoretically grounded: for a detailed explanation see Maclin (1999). Figure 2 shows the learning curve for the models (*Mean Squared Error* value at each epoch) on both the development set and the internal test set. Because of the small batch size the learning curve of the ten models are a bit unstable: we noticed this behavior already on the model selection phase. Some of the 5000 tested configurations during the model selection phase had a significantly stabler learning curve but higher *Loss* and *Mean Euclidean Error*. We took in consideration the possibility of selecting a stabler model but our final decision was to try to get a stabler model using the ensemble technique. As shown by figure 2 we achieved our goal: the learning curve of the ensemble model is stabler than the constituent ones.

4 Conclusions

We exploited the ensemble technique to get a final model that achieves lower *Mean Euclidean Error* on the internal test set compared to the constituent ones; the learning curve of the ensemble model is also more stable. We used the ensemble model to compute the predictions on the blind test set; we saved the predictions on the file named `ehNonLoSo_ML-CUP19-TS.csv`. We are positive that the *Mean Euclidean Error* on the internal test set will be a good approximation of the real performances of our final model on unseen data since the internal test set was not used in the model selection phase. Thus we expect that the loss on the blind test set will be around 0.92241244.

model	**Development set *MEE***	**Internal test set *MEE***
model0	0.729490132	0.950690925
model1	0.888218691	0.988404175
model2	0.681912563	0.937223263
model3	0.664096355	0.998971616
model4	1.060627644	1.184769859
model5	0.833550662	0.966579113
model6	0.868996800	1.024628064
model7	0.871509238	0.987839927
model8	1.010520955	1.047056016
model9	0.978153313	1.098921759
ensemble	0.768043332	**0.92241244**

Table 3: *Mean Euclidean Error* on the development set and internal test set of the ten best models and their ensemble.

Acknowledgment

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

References

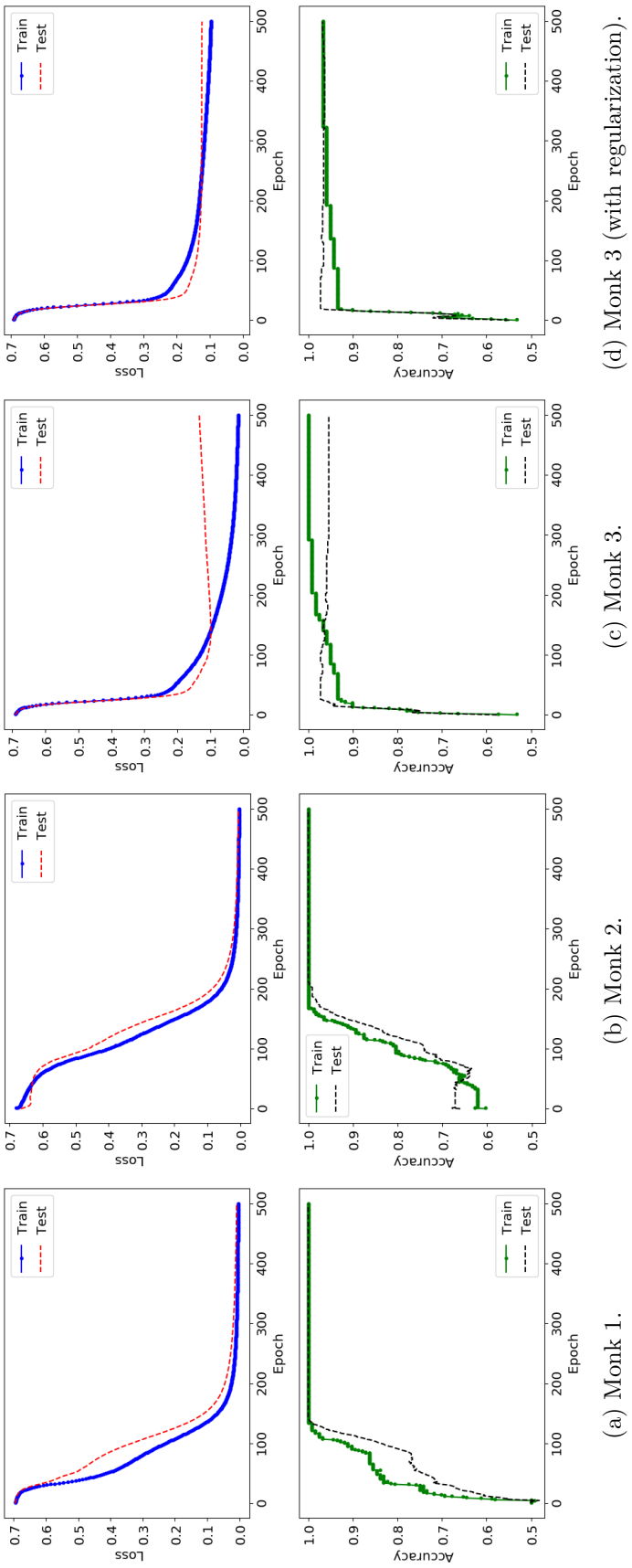
James Bergstra, Yoshua Bengio, and Leon Bottou. Random search for hyper-parameter optimization. *In: Journal of Machine Learning Research*, pages 281–305, 2012.

Francis Galton. Vox populi. *In: Nature*, pages 450–451, 1907. URL `https://doi.org/10.1038/075450a0`.

Richard Maclin. Popular ensemble methods: An empirical study. 11, 12 1999. doi: 10.1613/jair.614.

Alessio Micheli. Ml-cup dataset, 2019.

S. Thrun, J. Bala, E. Bloedorn, 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. Pachowicz, B. Roger, H. Vafaie, W. Van de Velde, W. Wenzel, J. Wnek, 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.

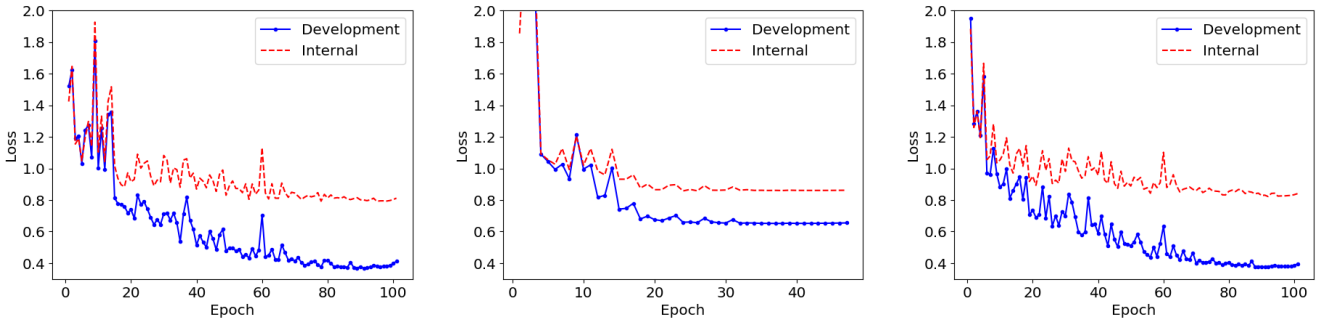


(a) Monk 1. (b) Monk 2. (c) Monk 3. (d) Monk 3 (with regularization).

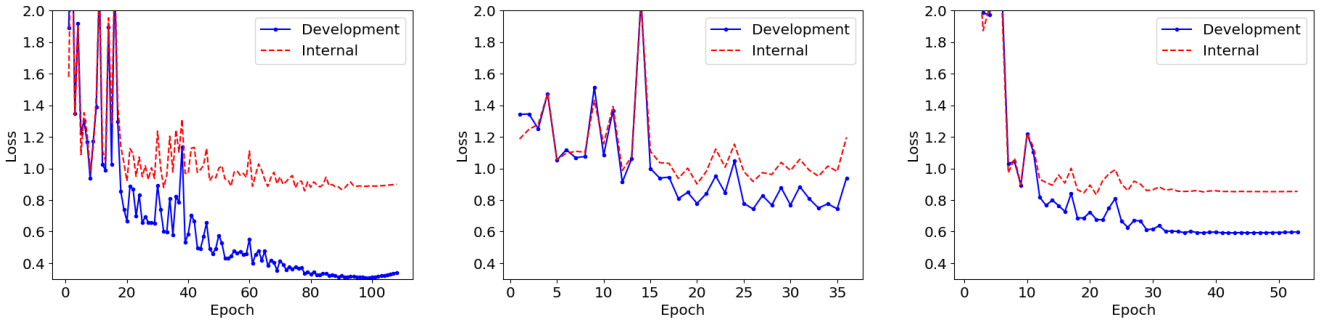
Figure 1: Learning curve (above) and accuracy over epochs (below) for the three monk tasks.

	avg train MEE	**avg validation MEE**	**activation**	**lambda**	**batch size**	**learning rate**	**eta**	**alpha**	**weights init fun**	**weights init value**
model0	0.67298315	0.99333171	tanh	0.00281810	1	linear	0.00588749	0.64150405	normal	0.39047741
model1	0.84554301	1.02229743	logistic	0.00059519	1	adaptive	0.05498002	0.80504194	normal	0.70000000
model2	0.63327172	1.03195820	tanh	0.00143471	5	linear	0.03018084	0.781902999	uniform	0.39013236
model3	0.62433689	1.03620072	tanh	0.00076123	5	linear	0.03817907	0.068887778	normal	0.33304170
model4	0.79642645	1.03657556	logistic	0.00021881	1	constant	0.01643717	0.483140617	normal	0.70000000
model5	0.85274914	1.04713386	logistic	0.00084636	1	adaptive	0.07968904	0.295413936	uniform	0.66822849
model6	0.88967873	1.04954635	logistic	0.00090442	1	adaptive	0.06205633	0.015103034	normal	0.70000000
model7	0.98377103	1.05369440	logistic	0.00171647	5	linear	0.05137566	0.55861828	normal	0.70000000
model8	0.88714749	1.05705671	logistic	0.00154276	5	constant	0.02023159	0.570495000	normal	0.70000000
model9	0.93709187	1.05901652	logistic	0.00167531	5	constant	0.06569622	0.554408733	normal	0.70000000

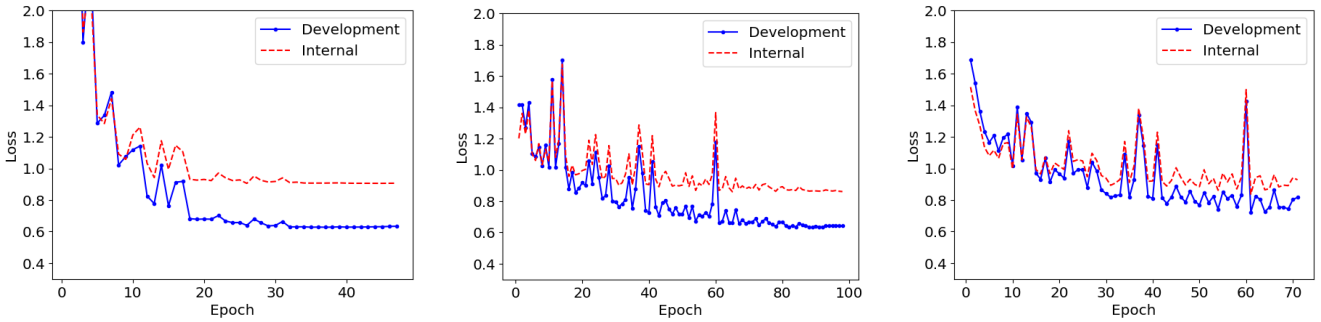
Table 4: Hyper-parameters and performances of the 10 best models in terms of validation MEE. All the models have two hidden layers of 50 units each and *early_stopping*=False



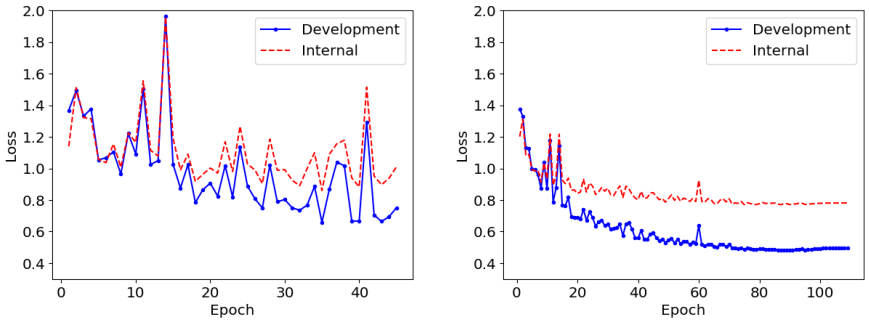
(a) model 0 (b) model 1 (c) model 2



(d) model 3 (e) model 4 (f) model 5



(g) model 6 (h) model 7 (i) model 8



(j) model 9 (k) **ensemble model**

Figure 2: Learning curve of the best ten models showing the loss (Mean Squared Error) at each epoch on the Development set and Internal Test set