Università di Pisa

320AA

Machine Learning: Fundamentals

AA1 Project

Author:
Andrea Madotto

Student Number:
540794

December 27, 2016

Contents

1 Abstract

In this project, we implemented a learning simulator system, i.e. a Neural Network. Furthermore, we implemented MLP (MultiLayer Perceptron) with Back Propagation algorithm. To improve the learning algorithm we include Momentum and a L_2 regularization term. The code has been realized using Python (2.7) with the support of a math library called Theano [9]. Finally, we employed our implementation to generate the regression prediction on the AA1 cup data.

2 Introduction

The aim of this NN implementation is to create a suitable model to predicted the proposed AA1 cup data. The data provided are two files, training set and a blind test set (without target values). The training set is made of 1016 training examples, where each of that has 12 variables (columns) with continuous values. The first 10 variables are our features to learn and the last 2 are the values to predict (target). The blind set, instead, has got 315 patterns but in this case with the last two columns (prediction) empty. This Cup requests a regression task on the latter 2 target variables.

The model used to make the predictions is a Multilayer Perceptron (MLP), that is a network of interconnected units. A typical architecture is the following: an input layer, with as many unit as the input feature, one or more Hidden Layer, with a parametric number of unit, where each of this has got an activation function (preferably differentiable), and an output layer, with as many unit as the output required (2 in the case of AA1 Cup).Each unit of a layer L_i, a part of the output ones, is connected through a weighed edge with the all the unit of the layer L_j (with $i < j$). The last thing to notice is that each unit sum up all the inbound edge and apply its unit function. Figure 2 show an graphical representation of a MLP.

This latter graphical representation can also be seen as a multidimensional function. For example if we have 3 layers i, k, j respectively, and we start from the input layer i to the output layer k, feedforward steps, and we apply summation and all the activation function we obtain a function in $x =< x_1, \ldots, x_n >$ as follow:

$$h(x) = f_k(\sum_j w_{kj} f_j(\sum_i w_{ji} x_i))$$

where f_k is the activation function of the output layer, w_{kj} and w_{ji} are the weight between the layer k j and j i respectively, and f_j is the activation function of the hidden units. The w_{kj} and w_{ji} are the free parameters (the one that we must learn) of the model and f_j are non liner function. A well known results[2, 4] tell us that a single hidden-layer network (with logistic activation functions) can approximate every continuous function (provided enough units in the hidden layer). A less obvious result, from few years ago, was the algorithm to infer the weights of the network. Indeed just in the late 90' the back-propagation algorithm was discovered [8].

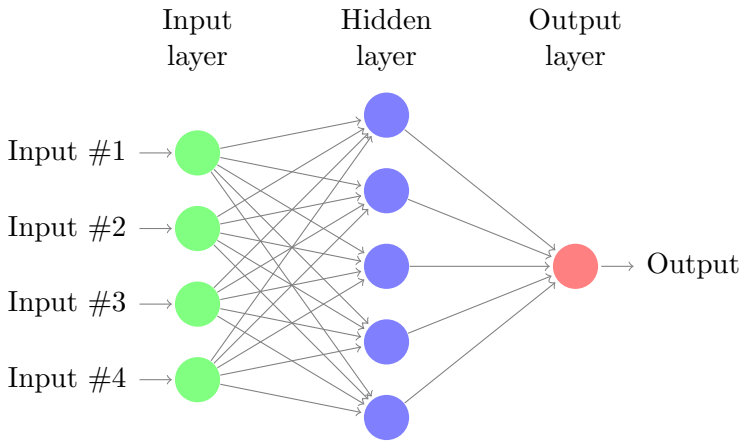


Figure 1: Typical structure of a feed-forward neural network. In this example we have 4 input, 5 hidden units, and 1 output

Therefore, we implemented a Neural Network model with the back-propagation algorithm. To speed up the learning process, we used a well known technique called Momentum. Instead, to control the model complexity (VC-dimensions) and then to avoid model over-fitting, we include an L_2 regularization[6] term in the Loss function.

3 Method

In this section we explain our design choices. First of all, it's worthwhile to spend some words on Theano, the math library used in this project. Theano is a Python library that allows to define, optimize, and evaluate mathematical expressions involving multi-dimensional arrays. The great advantage of this library is the possibility to declare symbolic function (in the straight mathematical sense). Therefore, Theano offers an efficient symbolic differentiation.

3.1 Design

First of all, we divided the symbolic part of the code to keep a clearer code (mostly for didactic purpose). The implementation of the neural networks is divided in 3 classes. Two of them (Layer and MLP) purely symbolic, and the remaining ones which instantiates and inits the model. Let's see more details of the class structure:

Layer It keeps the information relative to the layer. Therefore, it keeps a matrix for the weight (W), a bias vector (b), an activation function, and an array called params, which keeps track of the variable (W,b) that we will use to compute the gradient. This class provides a method called output, which receives an input vector x and returns the output of the layer.

MLP It is a container of layers. It takes the initial values as input for all the layer,

including the activation function. The class also provided two methods: recursive function output that computes the values in the output from the network, and mean_squared_error that returns a Theano function that computes the Mean Square Error.

NN This class initializes all the weights matrix [1] and instantiate the MLP. This class also creates a train function, which is a Theano function that takes the feature matrix (X), the expected output (y), and a cost function (defined in the MLP class (MSE)) as inputs.The Theano method creates a function allowing to update the parameter of the model according to the a specific function. Obviously, the function is going to be the gradient descent that will update all the weight in layers. This is possible thanks to the function grad of Theano which allows differentiation. It also must be noticed that Theano builds computational graph which allows to compute and properly propagate the gradient.

Model This class is a model container, and it's what the user interfaces with. In the case of the neural networks model, we will have an interface which takes all the model parameters and will init the model with such information. This class will also host the method train, which takes an integer number n (number of epochs) as input and applys the function train, previously defined, for n times.

3.2 Preprocessing

Since we tested our neural network in two datasets, monks (1,2,3) and AA1 cup data, we applied two different pre-processing steps. For Monks datasets we simply binary encode the input, as also suggested in [10]. The result is an encoding of 6 numerical feature into 17 binary inputs. Instead, for the AA1 cup data we do not apply any preprocessing step since we have noticed that the mean of each feature is closed to 0 (0.005 average among features) and the standard deviation to 1 (0.99089 average among features). Figure A in the Appendix shows the distribution of the feature.

3.3 Validation schema

Monks datasets have been used just for benchmark of our implementation, therefore we did not apply any validation scheme. Instead, for AA1 cup data we firstly take off 30% of the training set[2] to create our test set, and we keep the rest, from now on we call developing set, to train our model. So, we use 5 fold cross-validation in our developing test for model selection, and once we found our best model, we estimate its risk with the test set.

[1] The weight matrix has been initialized using scipy randn that is given a matrix of random floats sampled from a univariate "normal" (Gaussian) distribution of mean 0 and variance 1. Further we divided the latter term by $\frac{2}{\sqrt{fun_i n}}$, "This ensures that all neurons in the network initially have approximately the same output distribution and empirically improve the rate of convergence."[3, 5]

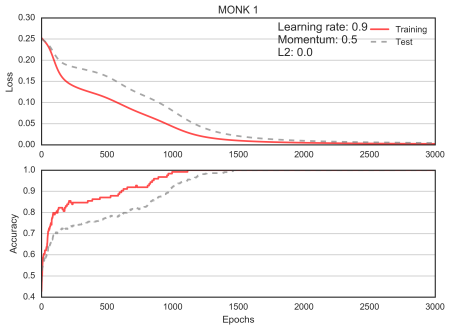
[2] Notice that the dataset has been randomly shuffle.

4 Experiental results

4.1 Benchmark

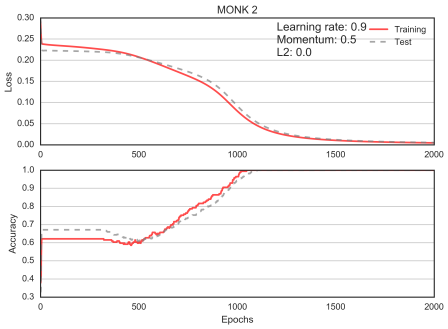
Using the Monk 1/2/3 datasets we benchmark our implementation. To be sure our model is working properly we used the hyper-parameters suggested in the literature [10]. In this experiment we have used the Mean Square Error (MSE hereafter) as Loss function. We can notice that even if we use a quite high learning rate we have a pretty slow convergence. However, the accuracy value, obtained in all the dataset, is the same as the one of the paper[10]. We summarized our result in Table 4.1, and we show the Loss and accuracy plot in Figure 2, with the hyper-parameters.

Monk	Hidden unit	λ	Training Loss	Test Loss	Accuracy test
1	3	0.00	0.003	0.004	100%
2	2	0.00	0.002	0.003	100%
3	4	0.00	0.031	0.054	93.1%
3	4	0.01	0,240	0.238	97.2%

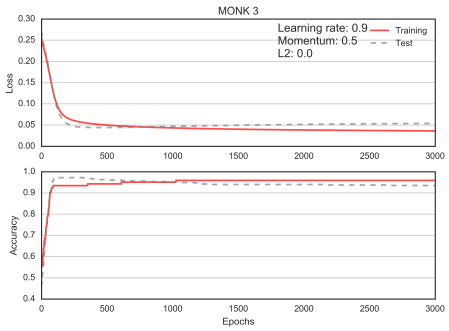


(a) MONK 1

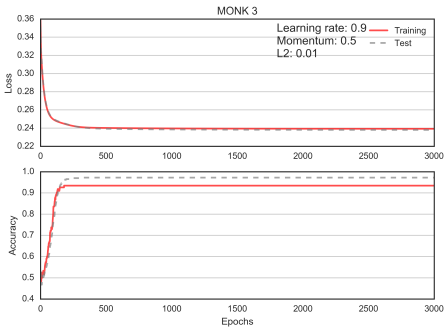
SU



(b) MONK 2



(c) MONK 3



(d) MONK 3 $L2$ regularization ($\lambda = 0.01$)

Figure 2: **Loss and accuracy plot**. In the four subplots we show how the loss decrease among the epoch and the accuracy increase in the 3 monk datasets.

4.2 AA1 cup data

In this section we are going to show how did we train our model. As explained in the previous section we take off, after a shuffling, 30% of the training set to obtain our test set. The remaining data, also called developing test, will be used to make a model selection. To do so we employed a 5 fold cross-validation technique.

In each fold we have tested the the following hyperparameters: hidden units number, learning rate η, momentum α, and regularization parameter λ. First of all, we have run some experiment to find a subsets of value in which we have a smooth loss function. The main source of oscillation results to be the combination between η, α, and Hidden unit number. As we can see from Figure 4.2 and Figure 4.2, with 100 hidden units almost all the combination of value are unstable. On the other hand, with 25 hidden units almost all the loss function are quite stable. Other combinations of hyper-parameter doesn't has much oscillation, in particular using the parameter λ, which doesn't affect the smoothness of the curve. More plots are of the latter observation are available in the Appendix.

Therefore, thanks to the previous steps we select the subset of candidate hyper-parameters, the one that gives us a smooth loss plot. Hence, we can run the cross validation in our developing test using the select subset parameters. In Table 4.2 we summarize the hyper-parameters values used in the model selection. So as you can notice we limited the number of hyper-parameters for two main reasons: stability, as explained before, and bound the possible number of combination. The latter is due to the fact that if we increase the parameter, the possible combination increase exponentially. So we manually select the possible values for each parameter in order to cover a reasonable amount of range. For example the selected hidden unit number are [10,25,50,100]. Obviously, we could have added a finer-grain, but it wouldn't change a lot the final result. Last but not least, a parameter that we didn't have is the number of epoch, which is set to 5000. This number is quite high, but is guided by the fact that also in the monk dataset we had quite slow convergence.

Hidden Unit	η	α	λ
100	0.01	[0.1,0.3,0.6,0.8]	[0.01,0.001,0.0001]
50	0.01	[0.1,0.3,0.6,0.8]	[0.01,0.001,0.0001]
	0.03	[0.1,0.3]	[0.01,0.001,0.0001]
25	0.01	[0.1,0.3,0.6,0.8]	[0.01,0.001,0.0001]
	0.03	[0.1,0.3,0.6,0.8]	[0.01,0.001,0.0001]
	0.05	[0.1,0.3,0.6]	[0.01,0.001,0.0001]
10	0.01	[0.1,0.3,0.6,0.8]	[0.01,0.001,0.0001]
	0.03	[0.1,0.3,0.6,0.8]	[0.01,0.001,0.0001]
	0.05	[0.1,0.3,0.6,0.8]	[0.01,0.001,0.0001]

Table 2: **Hyperparameters**. The table shows the values tried for the model selection. The last column represent all the value that we have tried in combination with the previous.

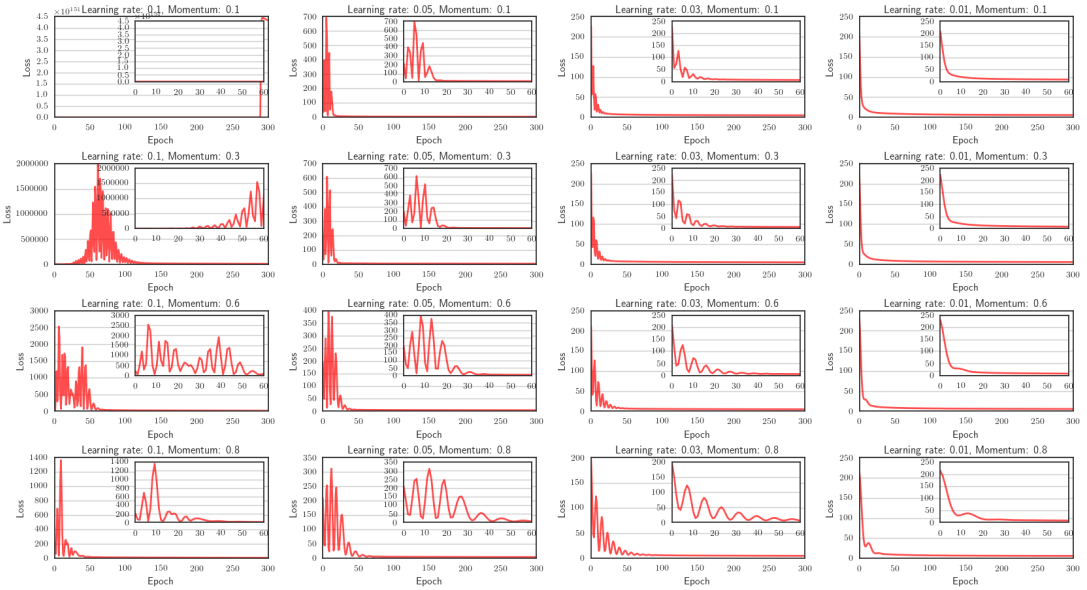


Figure 3: **Learning rate η vs Momentum α**. Each subplot represent the MSE (e.g., loss function) using different parameters of η and α. In this plot we fixed the hidden unit number to 100 and the λ to 0.01.

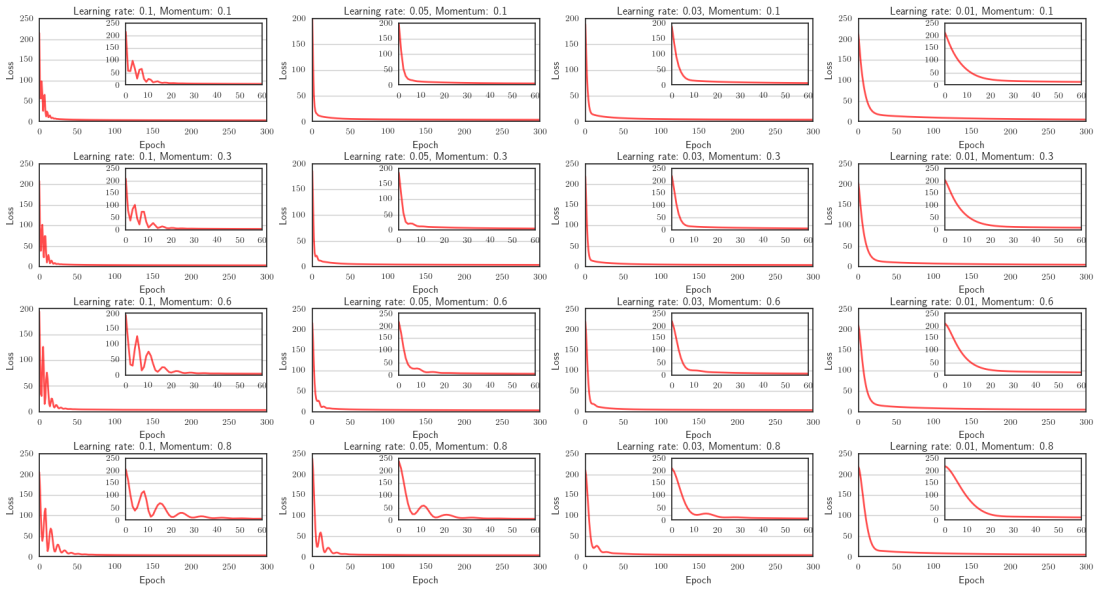


Figure 4: **Learning rate η vs Momentum α**. Each subplot represent the MSE (e.g., loss function) using different parameters of η and α. In this plot we fixed the hidden unit number to 25 and the λ to 0.01.

	H.U.	η	α	λ	Train MEE	Train MSE	Val MEE	Val MSE
Fold 1	25	0.03	0.8	0.0001	1.086434	1.011060	1.292398	1.565190
	10	0.08	0.6	0.001	1.066568	1.236531	1.292884	1.789303
Fold 2	10	0.08	0.1	0.001	1.074933	1.271130	1.207243	1.563573
	10	0.05	0.8	0.0001	1.072685	1.006237	1.216842	1.297183
Fold 3	25	0.05	0.6	0.0001	1.012810	0.930972	1.186042	1.214386
	25	0.03	0.3	0.0001	1.109116	1.092933	1.211743	1.221347
Fold 4	10	0.08	0.6	0.001	1.094210	1.338774	1.136196	1.311816
	25	0.05	0.3	0.001	1.100677	1.342305	1.136536	1.278232
Fold 5	25	0.05	0.6	0.0001	1.052122	0.998776	1.101299	1.189474
	25	0.05	0.1	0.0001	1.064009	1.013232	1.110535	1.207075
Average					1.071903	1.136765	1.177702	1.341376
Standard Deviation					0.028927	0.160730	0.061913	0.202629

Table 3: **Best model in each Fold**. In the table we show the best (based on validation MEE) two models for each fold.

In Table 4.2 we show the best two models obtained in each fold of the cross validation. The first thing that we can notice, is that the standard deviation among the validation MSE (Mean Squared Error) of the best solutions is high (around 0.2). This can be due to the cardinality of each fold, which is quite small. Indeed, we have around 700 samples in the developing test, so each fold has got 560 samples for training and 140 for validation. However, the MEE in the validation set is quite stable: 1.17 ± 0.06

So now comes the difficult parts, the choice of final hyper-parameters. Let's go through each of them:

- **Hidden unit**. The choice is between 10 and 25. We choose 25 because it appears the major number of times in the best two folds, and because the MEE in the validation is lower.
- **Learning rate** η. Since we have chosen 25 hidden units the only possible choice is between 0.05 and 0.03. We choose 0.05 since, as in the previous case, it appears more times and has got lower MEE in validation.
- **Momentum** α. This choice is guided by the previous points, indeed we choose 0.06 because this value in combination with all the other hyper-parameters gives the smallest MEE values.
- **Regularization** λ. The only possible value compatible with the other is 0.0001. It could be quite low, and one can think to increase it to avoid overfitting. However, we don't want to have bias in the analysis, so we keep the one selected by the model selection process.

So our final model is the following: $< \text{HU} = 25, \eta = 0.05, \alpha = 0.6, \lambda = 0.0001 >$. Now that we choose our model, it is time to estimate the error in the test set. Hence we are going to train our model with the selected parameters, using all the developing set, and use the test set to check our prediction power.

The final results, averaged among 10 experiments, are shown in Table 4.2. So we obtain an MEE equal to 1.2436 ± 0.012 using 10000 epochs.

Train MEE	**Train MSE**	**Test MEE**	**Test MSE**
1.030 ± 0.0122	0.972 ± 0.0204	1.2264 ± 0.0151	1.402 ± 0.0255

Table 4: **Final model results**

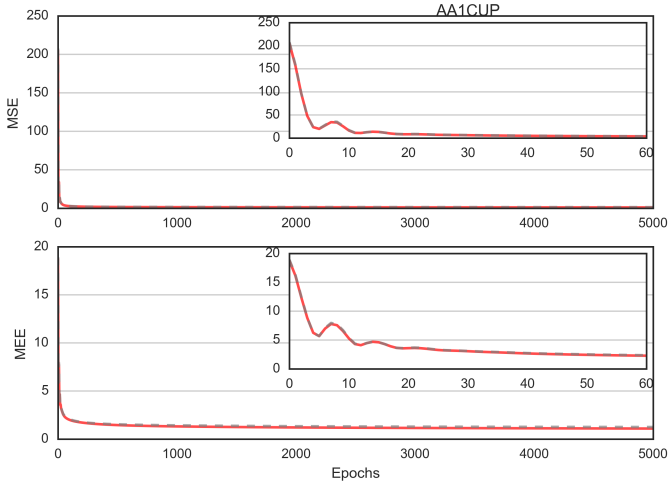


Figure 5: **Plot of the MSE and MEE using the selected model**. In this plot we show the plot of the MSE and MEE, in the train and in the test set using the selected model. We can notice a little oscillation in the first 10 epochs, but overall we have a quite smooth curve.

Since the obtained MEE by itself is not very meaningful, we employed a Linear Regression Model to compare our result. The library scikit-learn [7] offers us a simple LinearRegression model. Thus, we train the linear model using the developing set, and we use the test set con calculate our MEE. As result we obtained an MEE equal to 2.19. So our result using the NN has got 78.57% lower than MEE compare to the linear model.

Furthermore, we implement the Neural Network using Keras[1], an high level library to build neural networks. In the Appendix we reported the results of the 5 cross fold validation using the same hyper-parameters subset used before (obviously we should have choose another subset, but we want to keep it the same to have a measure of comparison). The

results are quite similar to the one obtained using our implementation based on Theano library. So we also choose the same model ($< \text{HU} = 25, \eta = 0.05, \alpha = 0.6, \lambda = 0.0001 >$) as the one that is chosen before (e.g. choosing the parameter that gives lowest MEE in the validation set). So we also tested this model with our training set and we obtained the following result: Test MEE equal to 1.2272 ± 0.0110 Test MSE equal to 1.3600 ± 0.0202. So we have obtained a quite similar results to our model.

5 Conclusions

In this project we implemented a MLP using python with the support of Theano library. Furthermore, we have benchmarked our model with monks datasets. Once we ultimate our model we employed a regression task on the AA1CUP data. In doing so, we have been going through a model selection phase, which includes a grid search with a 5 fold cross validation, and a model assessment phase to estimate the prediction power of our model.

The MEE obtained in our test set, using the best model selected by the grid search in all the fold, is 1.2264 ± 0.0151, which is quite closed to the value obtained in all the fold of the cross validation. Furthermore, we compared our solution with a Linear regression model, and with another Neural Network model using Keras. Our result greatly improves the linear model, and we obtains a comparable result with the other NN model.

The overall approach in this project is to follow a straight procedure in all the process, starting from the division of the dataset, to a clear separation between model selection and model assessment. However, we made also an heuristic choice such as the subset selection of the hyper-parameters for the model selection, which was half guided by the smoothness of the curve.

The name of the result file is samurai_LOC-OSM2-TS.csv and it's in attachment with the email. The loss obtained in the training set of the model used to make the prediction is 0.916, with an MEE equal to 1.016. The nickname that I would like to use for the competition is 'samurai'. I also would like to explicitly allow the publication of my result in the leader board of the AA1CUP.

References

[1] François Chollet. Keras. https://github.com/fchollet/keras, 2015.

[2] G Gybenko. Approximation by superposition of sigmoidal functions. *Mathematics of Control, Signals and Systems*, 2(4):303–314, 1989.

[3] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 1026–1034, 2015.

[4] Kurt Hornik. Approximation capabilities of multilayer feedforward networks. *Neural networks*, 4(2):251–257, 1991.

[5] Andrej Karpathy, 2016.

[6] Andrew Y Ng. Feature selection, l 1 vs. l 2 regularization, and rotational invariance. In *Proceedings of the twenty-first international conference on Machine learning*, page 78. ACM, 2004.

[7] 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.

[8] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *Cognitive modeling*, 5(3):1, 1988.

[9] Theano Development Team. Theano: A Python framework for fast computation of mathematical expressions. *arXiv e-prints*, abs/1605.02688, May 2016.

[10] S. B. Thrun, J. Bala, E. Bloedorn, I. Bratko, B. Cestnik, J. Cheng, K. De Jong, S. Dzeroski, S. E. Fahlman, D. Fisher, R. Hamann, K. Kaufman, S. Keller, I. Kononenko, J. Kreuziger, R.S. Michalski, T. Mitchell, P. Pachowicz, Y. Reich, H. Vafaie, W. Van De Welde, W. Wenzel, J. Wnek, and J. Zhang. The monk's problems a performance comparison of different learning algorithms. Technical report, 1991.

A Appendix

Table 5: **5 fold cross validation using Keras model**

	H.U.	η	α	λ	**Val MSE**	**Val MEE**
Fold 1	25	0.03	0.8	0.001	1.525609	1.272510
	10	0.08	0.3	0.0001	1.519781	1.277532
Fold 2	10	0.05	0.6	0.001	1.224283	1.198710
	10	0.08	0.3	0.001	1.233863	1.198718
Fold 3	25	0.05	0.6	0.0001	1.179464	1.200955
	25	0.03	0.8	0.0001	1.227767	1.202943
Fold 4	10	0.03	0.6	0.001	0.992829	1.131059
	10	0.08	0.3	0.001	0.993882	1.133195
Fold 5	25	0.05	0.6	0.0001	1.176011	1.106533
	25	0.03	0.8	0.0001	1.189026	1.111337
Average					1.19299	1.17344
Standard Deviation					0.17963	0.06175

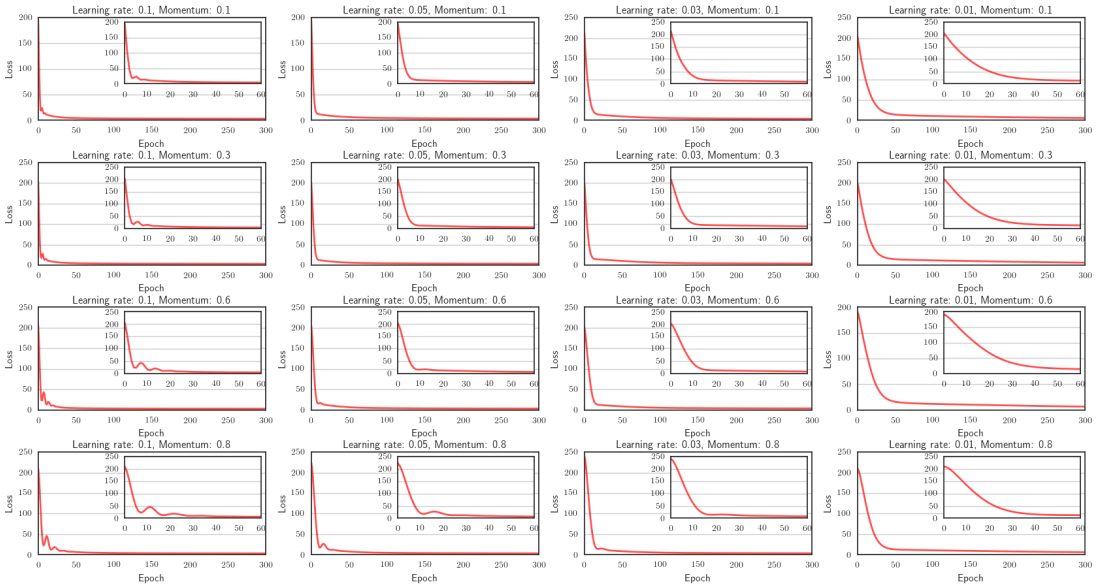


Figure 6: **Learning rate η vs Momentum α**. Each subplot represent the MSE (e.g., loss function) using different parameters of η and α. In this plot we fixed the hidden unit number to 10 and the λ to 0.01.

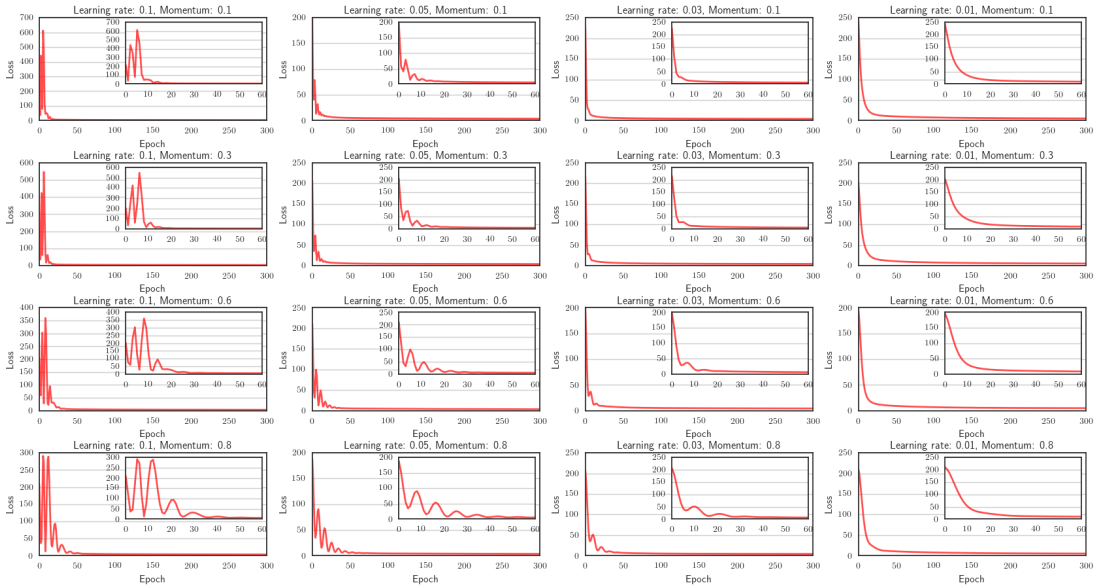


Figure 7: **Learning rate η vs Momentum α**. Each subplot represent the MSE (e.g., loss function) using different parameters of η and α. In this plot we fixed the hidden unit number to 50 and the λ to 0.01.

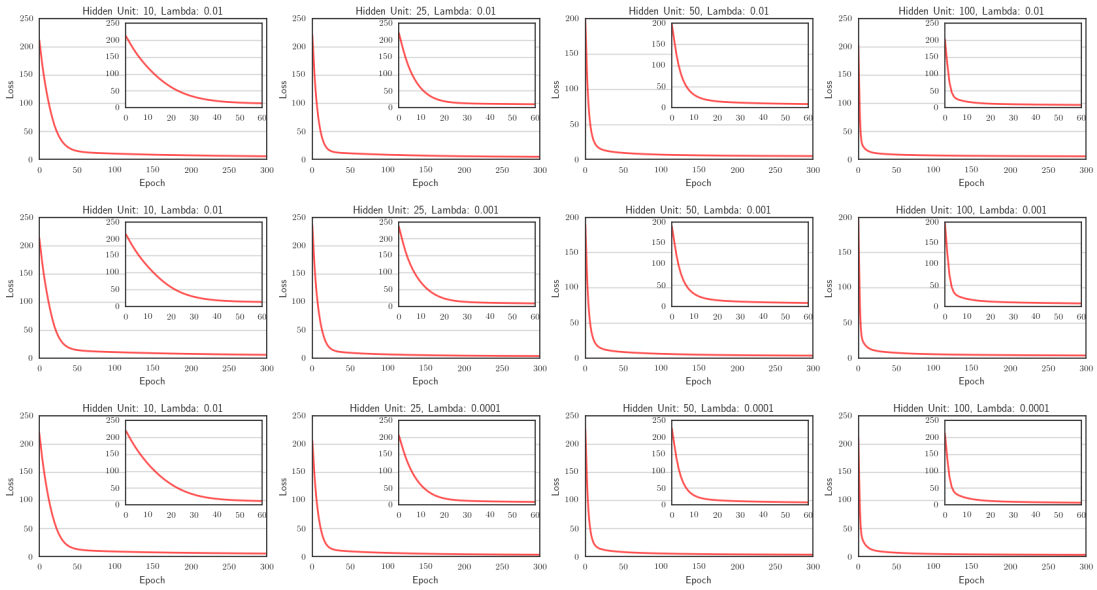


Figure 8: **Hidden Unit vs λ**. Each subplot represent the MSE (e.g., loss function) using different parameters of Hidden Unit and λ. In this plot we fixed the α to 0.1 and the η to 0.01.

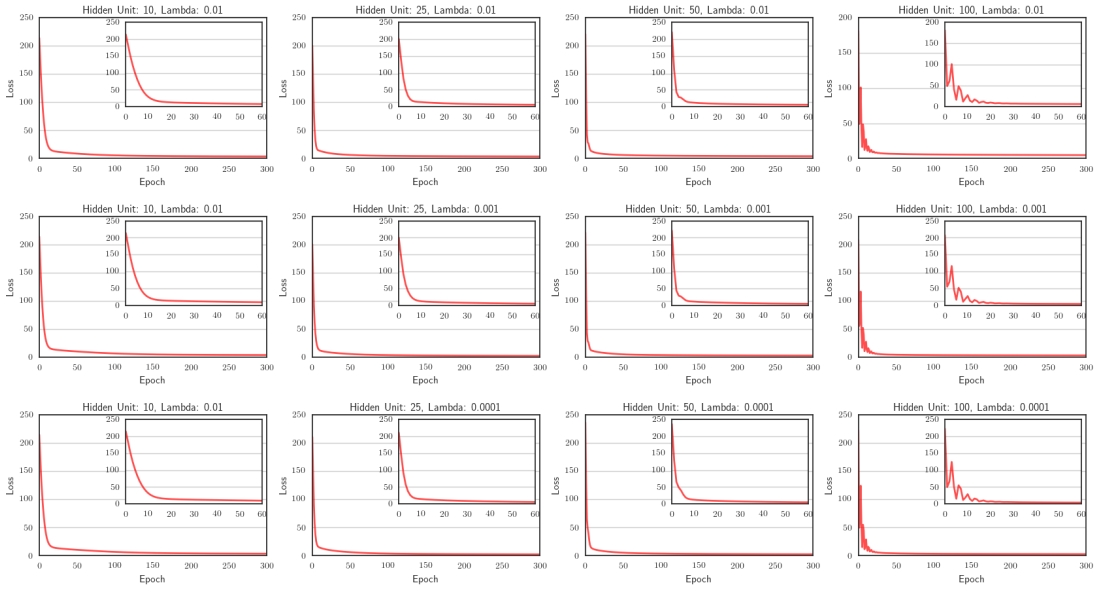


Figure 9: **Hidden Unit vs** λ. Each subplot represent the MSE (e.g., loss function) using different parameters of Hidden Unit and λ. In this plot we fixed the α to 0.1 and the η to 0.03.

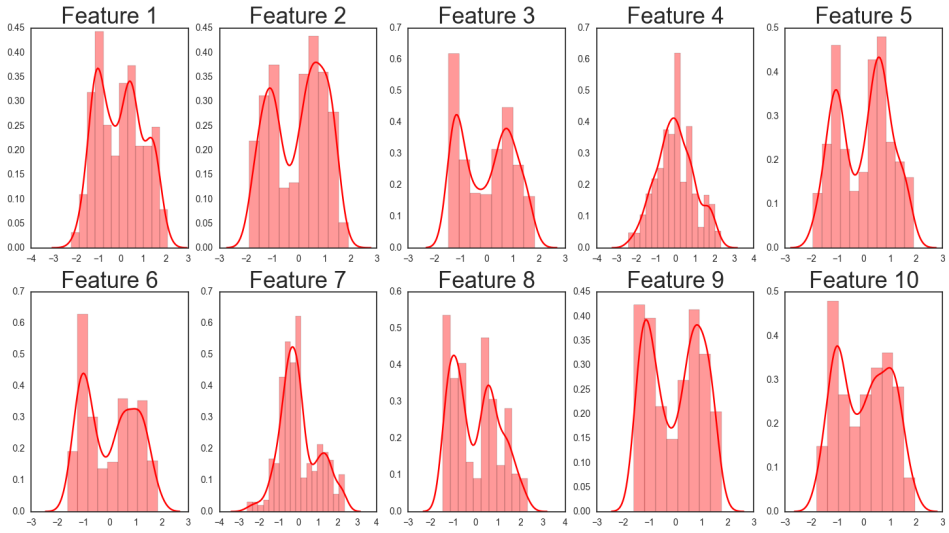


Figure 10: **Feature distribution**.