Computational Mathematics

PROJECT #3 WITH MACHINE LEARNING
2017/2018

Ahmad Alleboudy | Lanpei Li
[ahmad.alleboudy@outlook.com] | [lilanpei@gmail.com]

Contents

1. Optimization Problems Solved

The main problem approached is the UNIPI AA1 CUP challenge, a regression problem utilizing data collected from noisy sensors.
However, during the building process of our lightweight library, we extensively utilized; the Monk's problem datasets, used for verifying the correctness of a machine learning algorithm.
In the following sections, we provide details on the development journey of our hopefully, MEGA project, for 12 Credits of the computational mathematics course, please enjoy ☺

1) MONK'S PROBLEMS DATASETS

The MONK's problem were introduced as artificial problems to verify the correctness of a machine learning algorithm and to compare different algorithms.

Which makes them a very great means for building one's own machine learning algorithms.

They are three classification problems, sharing the same artificial domain of a robot described by six attributes

1- x1: head shape:
   round, square, octagon
2- x2: body shape:
   round, square, octagon
3- x3: is smiling?
   yes, no
4- x4: holding an object of the following:
   sword, balloon, flag
5- x5: jacket color:
   red, yellow, green, blue
6- x6: has tie?
   yes, no

Each problem defines a logical expression that should be evaluated;

Problem M1:

(Head shape = body shape) or (jacket color = red)

From 432 possible examples, 124 were randomly selected for the training set. There were no misclassications.

Problem M2:

Exactly two of the six attributes have their first value.

(E.g.: body shape = head shape = round implies that robot is not smiling, holding no sword, jacket color is not red and has no tie, since then exactly two (body shape and head shape) attributes have their first value) From 432 possible examples, 169 were randomly selected. Again, there was no noise.

Problem M3:

(Jacket color is green and holding a sword) or (jacket color is not blue and body shape is not octagon)

From 432 examples, 122 were selected randomly, and among them there were 5% misclassications, i.e. noise in the training set.

2) AA1 CUP CHALLENGE

The provided data are of the format:

Training set:
id,input1,input2,input3,input4,input5,input6,input7,input8,input9,input10,target_x,target_y

And it has 1016 records, for building and selecting the model

Blind Test set:
id,input1,input2,input3,input4,input5,input6,input7,input8,input9,input10

And it has 315 records, for the final challenge

2. The implemented solution methods

1) MLP

The main algorithm is a multilayer perceptron topology with a Tanh activation along with its trainer algorithms as required.

All of the experiments reported here are utilizing the Grot weight initialization using the same seed to generate the random weights

a. Adam Optimizer

We also provided our Adam trainer implementation as the algorithm of our choice from the class of accelerated gradient methods

For the Adam optimizer, we introduced the following hyperparameters;

Learning Rates, here it represents the initial learning rate for adam

Regularization Rates

A grid search is utilized to decide the final values of the hyperparameters:

Initial Learnig Rate	0.001,0.005,0.01
Regularization Rate	0,0.001,0.01
Number Of hidden units	From 10 to 100 with a step of 10

Table 1 – **Adam hyperparameters**

The line search used:
Is a simple fixed step in the case of SGD+Momentum with the possibility to increase or decrease it utilizing the resilient setting

And is of course updated in the case of the Adam.

Stopping criteria:
For each setting, we trained with initially 1000 epochs, then examined all of the learning curves, to decide on a proper stopping criterion

we found that the runs are converging at very different number of epochs, some are actually flickering, some diverging and some still require more training epochs, which made it difficult to set a unified criterion for early stopping or decide the last number of epochs for the training on the whole dataset we decided to utilize regularization + a standard stopping criterion[2] i.e. no improvement in the loss anymore after a quite excessive number of epochs which could be examined easily from the learning curve of the runs, to decide the final number of epochs to be 5000 epochs for Adam's experiments

Here we report the top 20 smallest average MSE experiments run with Adam,

We used k-folds cross validation with k=5.
The following table reports for the top 10 experiments, the average MSE of the 5-folds for each experiment hyperparameters setting ordered by the average MSE ascending.

How to read the title?
hdn{x}=means number of hidden units in the hidden layer is x
k{5}= 5-fold experiment
lr{x} = learning rate is x
reg{x} = regularization is x

mo{x} = momentum value for SGD+Momentum

	name	avgMSE
73	hdn20_k5_lr0.001_reg0.001	2.869099
64	hdn30_k5_lr0.001_reg0.001	2.941200
37	hdn60_k5_lr0.001_reg0.001	2.945531
22	hdn80_k5_lr0.005_reg0.001	2.974550
55	hdn40_k5_lr0.001_reg0.001	2.980975
46	hdn50_k5_lr0.001_reg0.001	2.991854
19	hdn80_k5_lr0.001_reg0.001	3.024321
10	hdn90_k5_lr0.001_reg0.001	3.041776
1	hdn100_k5_lr0.001_reg0.001	3.045349
68	hdn30_k5_lr0.005_reg0.01	3.050320
31	hdn70_k5_lr0.005_reg0.001	3.064476
76	hdn20_k5_lr0.005_reg0.001	3.068522
67	hdn30_k5_lr0.005_reg0.001	3.103125
59	hdn40_k5_lr0.005_reg0.01	3.106687
13	hdn90_k5_lr0.005_reg0.001	3.119385
58	hdn40_k5_lr0.005_reg0.001	3.139477
50	hdn50_k5_lr0.005_reg0.01	3.139826
77	hdn20_k5_lr0.005_reg0.01	3.146856
82	hdn10_k5_lr0.001_reg0.001	3.156930
40	hdn60_k5_lr0.005_reg0.001	3.164501

Table 2 - Top 10 experiments

Here we report the top 20 experiments plots, for a comprehensive list of experiments plots and results, please find the folder adamexperimentsfinalPLOTS

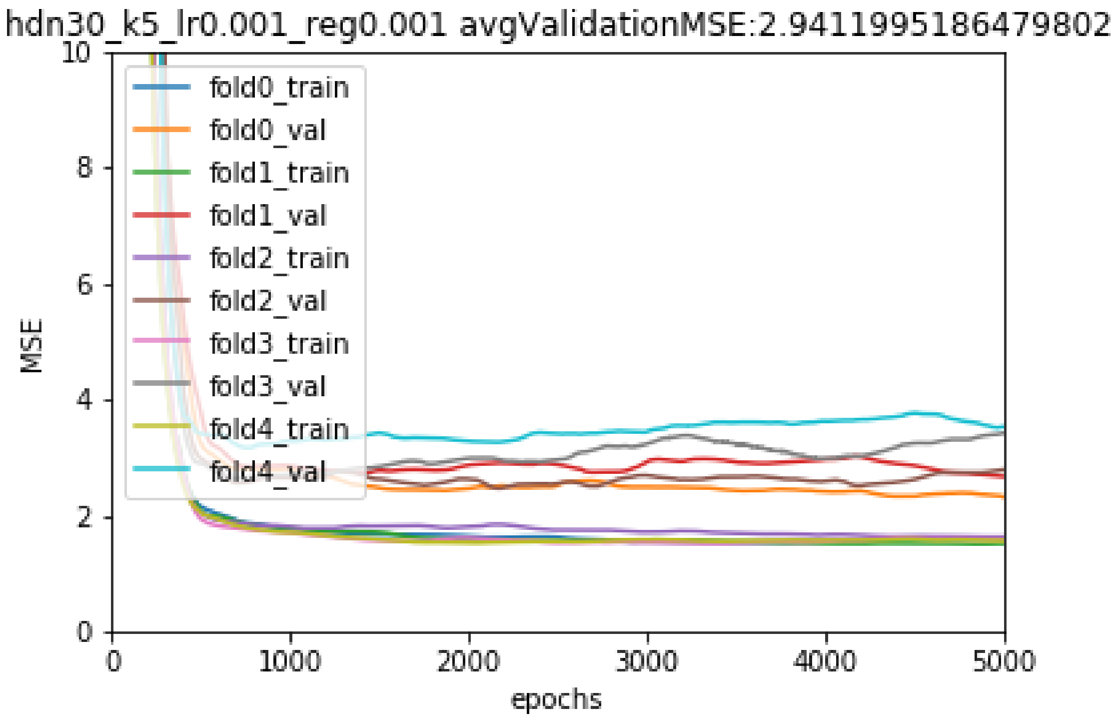


Figure 1: Learning Curve of hdn30_k5_lr0.001_reg0.001

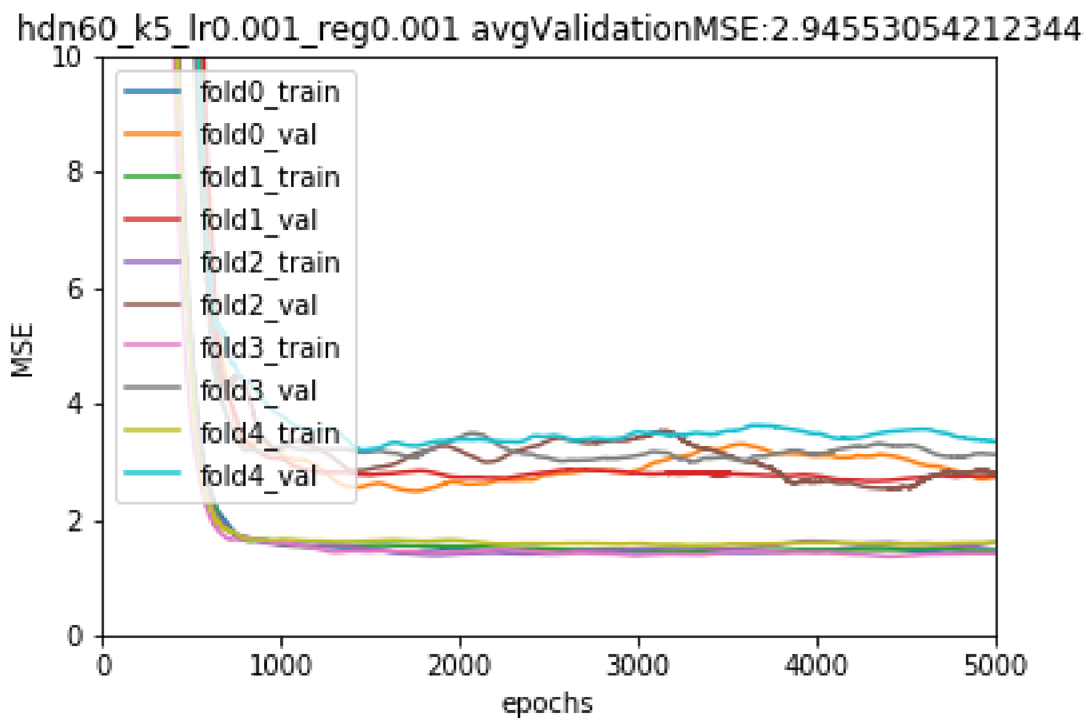


Figure 2: Learning Curve of hdn60_k5_lr0.001_reg0.001

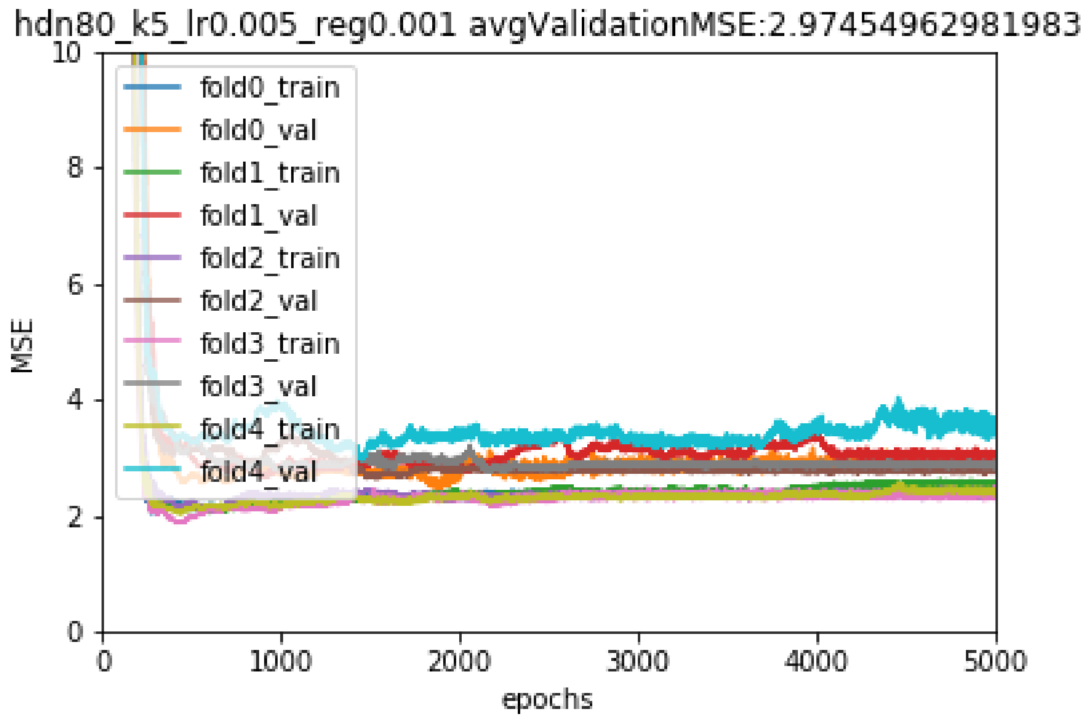


Figure 3: Learning Curve of hdn80_k5_lr0.005_reg0.001

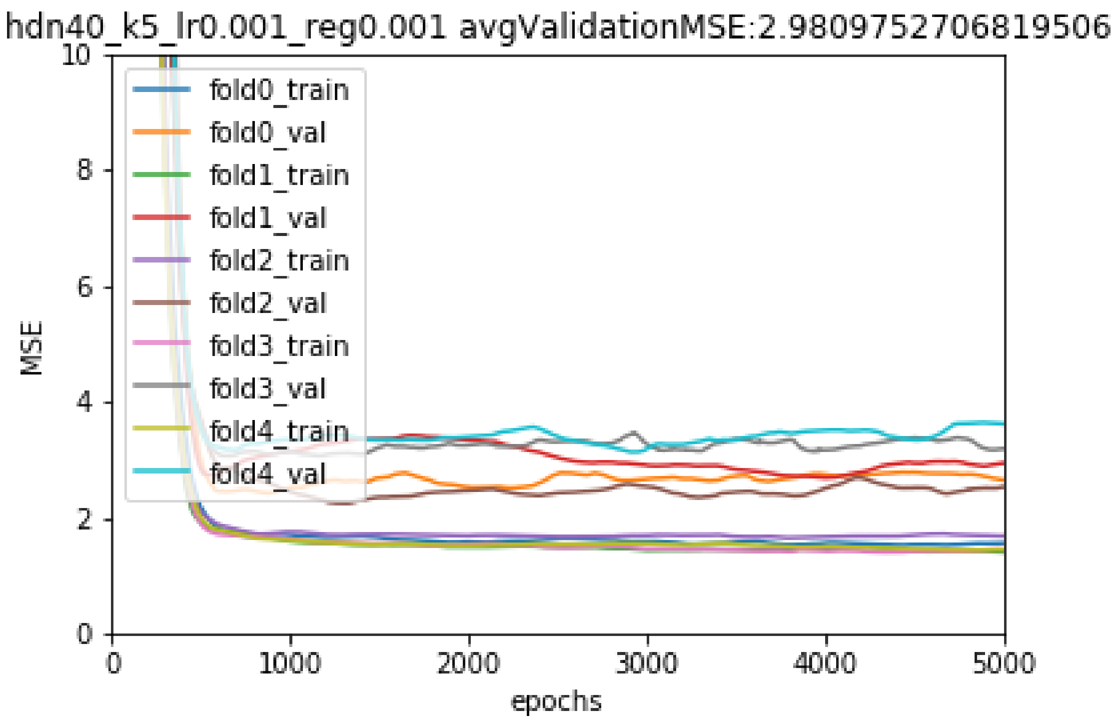


Figure 4: Learning Curve of hdn40_k5_lr0.001_reg0.001

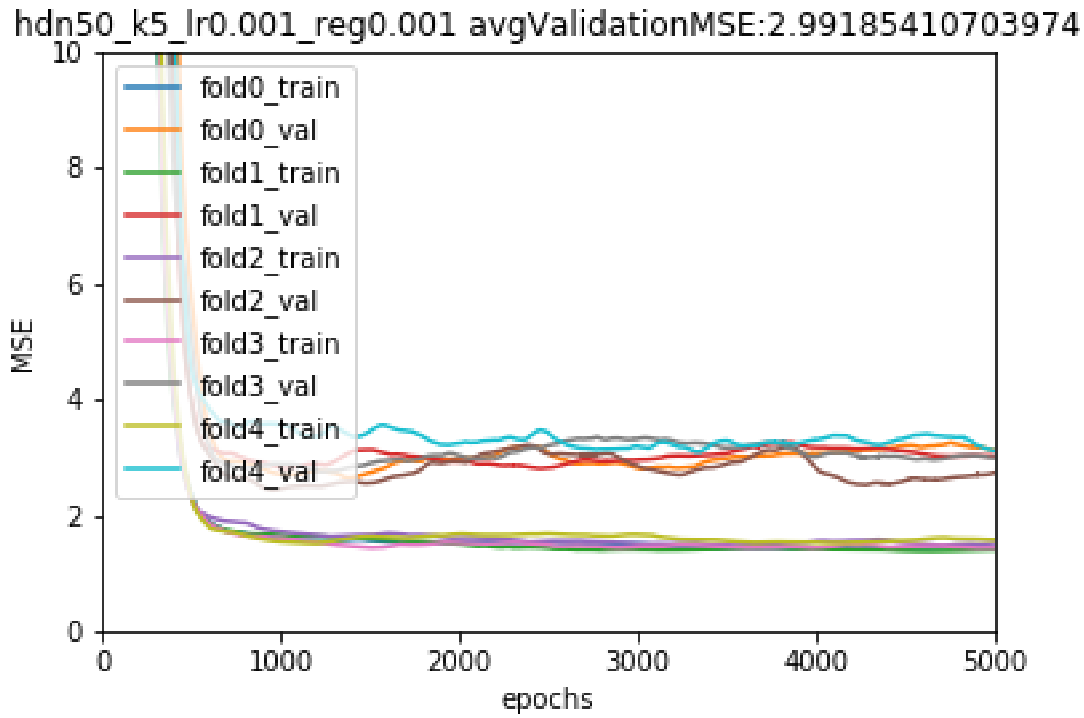


Figure 5: Learning Curve of hdn50_k5_lr0.001_reg0.001

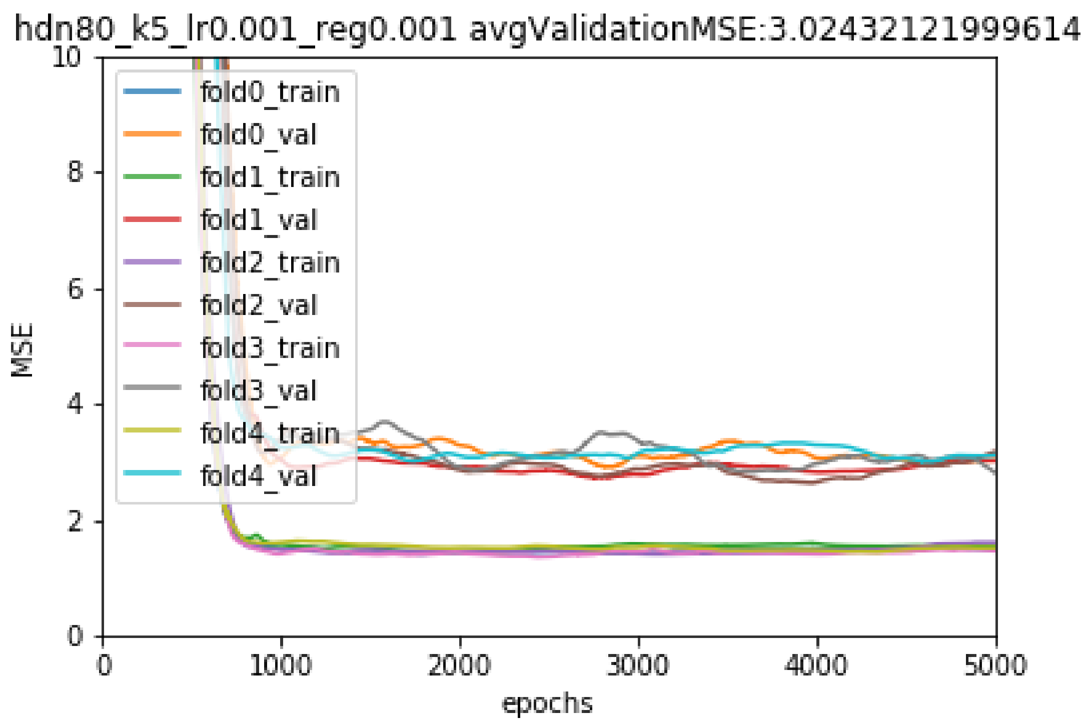


Figure 6: Learning Curve of hdn80_k5_lr0.001_reg0.001

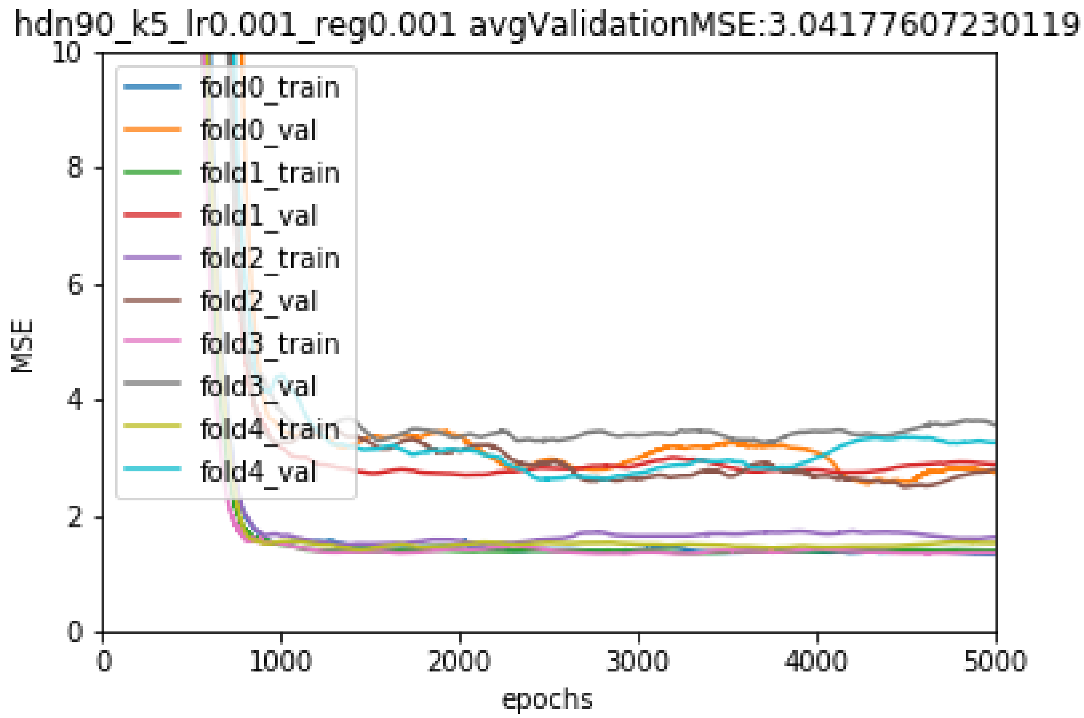


Figure 7: Learning Curve of hdn90_k5_lr0.001_reg0.001

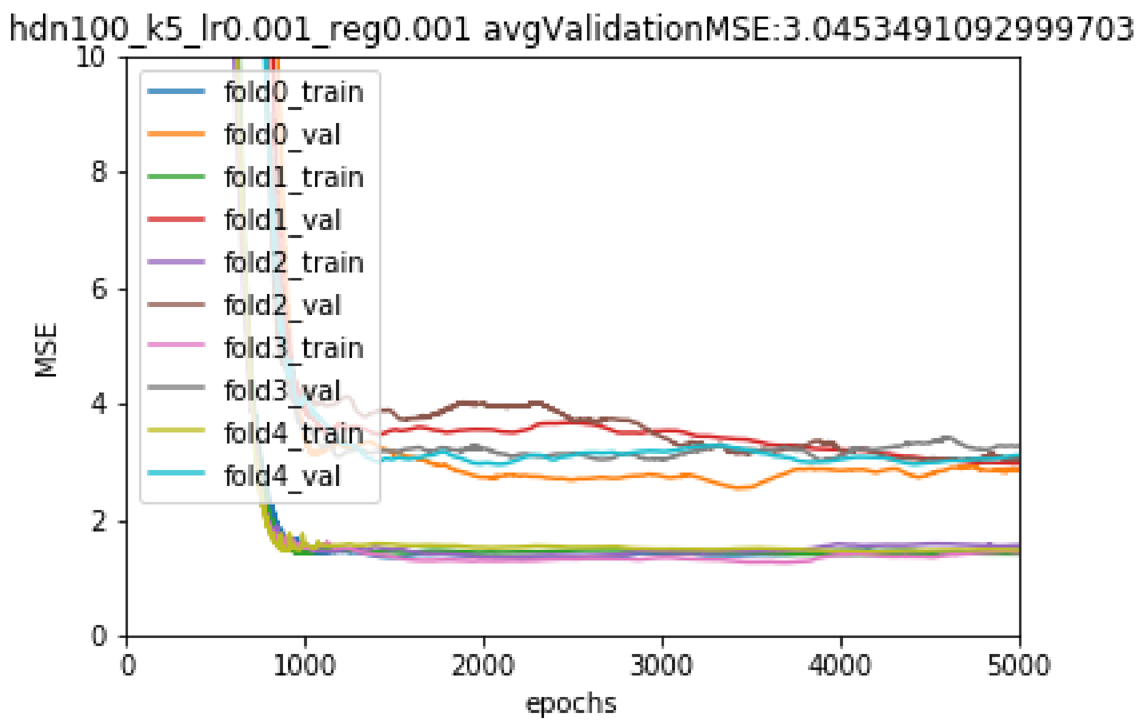


Figure 8: Learning Curve of hdn100_k5_lr0.001_reg0.001

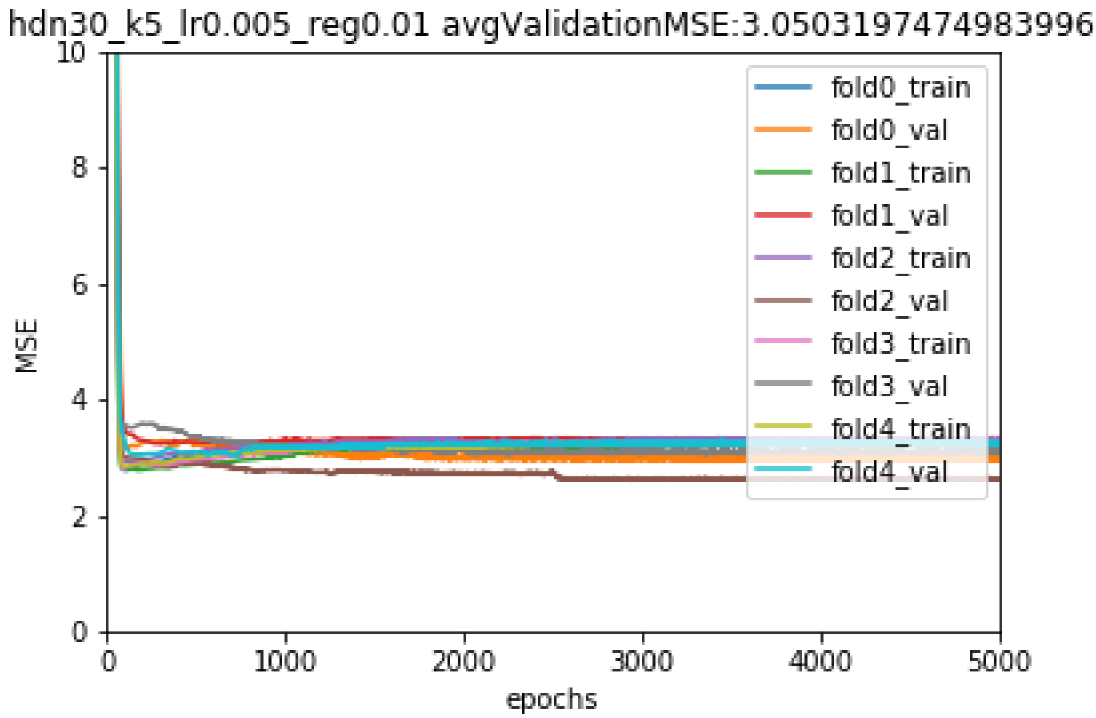


Figure 9: Learning Curve of hdn30_k5_lr0.005_reg0.01

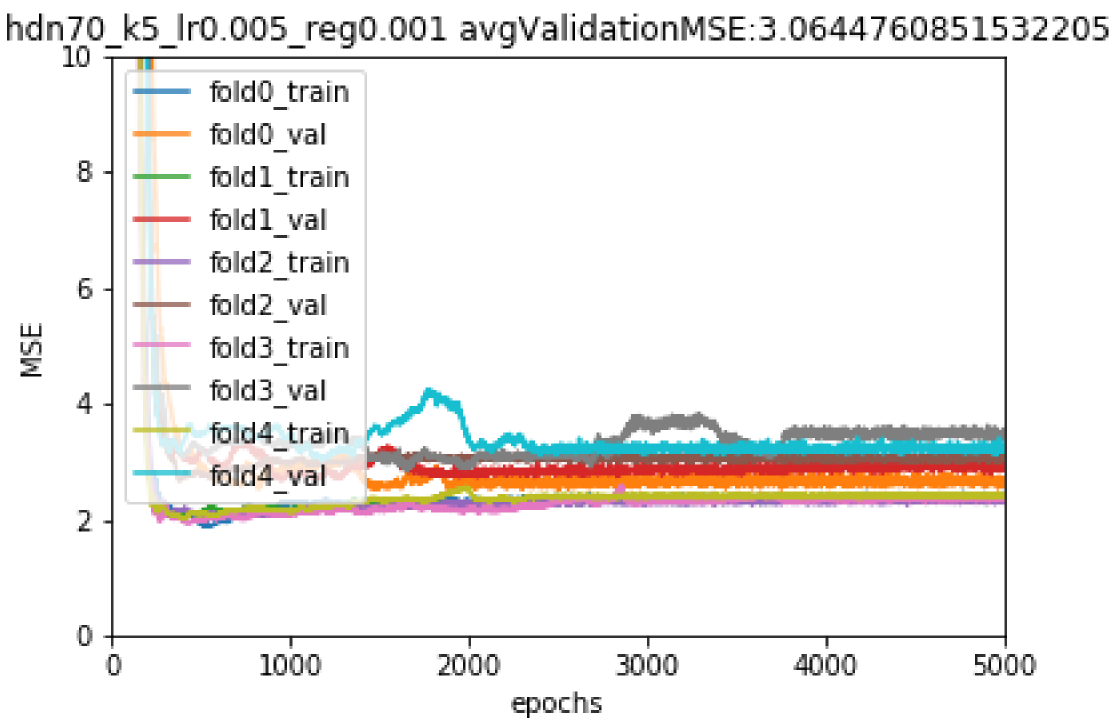


Figure 10: Learning Curve of hdn70_k5_lr0.005_reg0.001

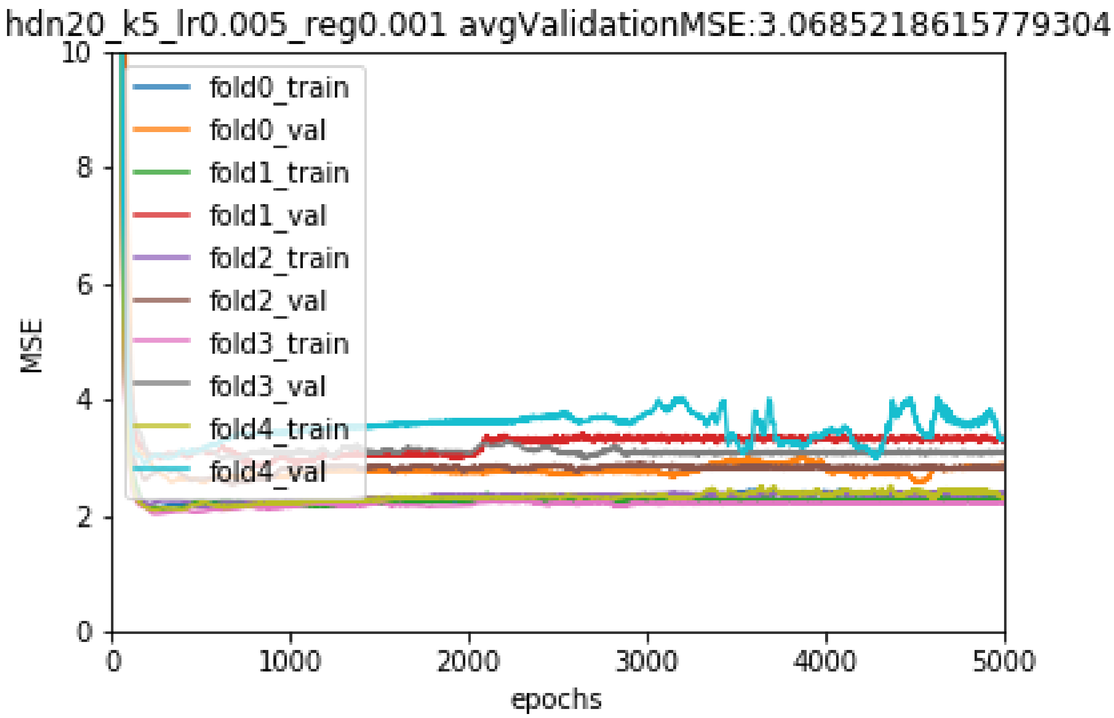


Figure 11: Learning Curve of hdn20_k5_lr0.005_reg0.001

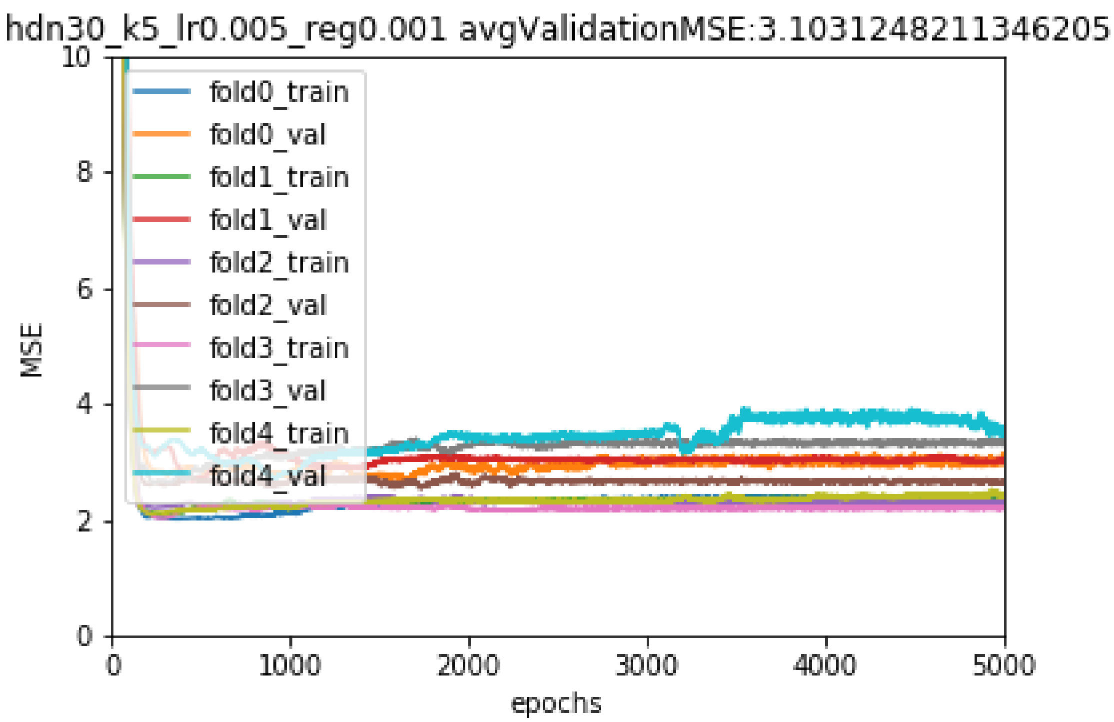


Figure 12: Learning Curve of hdn30_k5_lr0.005_reg0.001

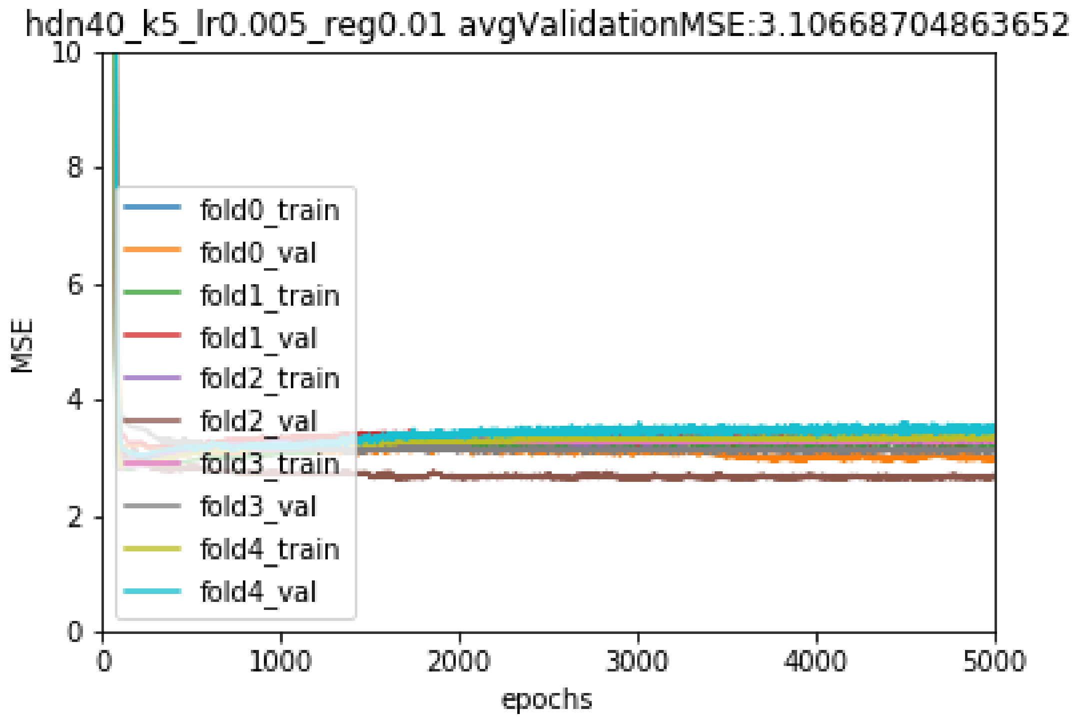


Figure 13: Learning Curve of hdn40_k5_lr0.005_reg0.01

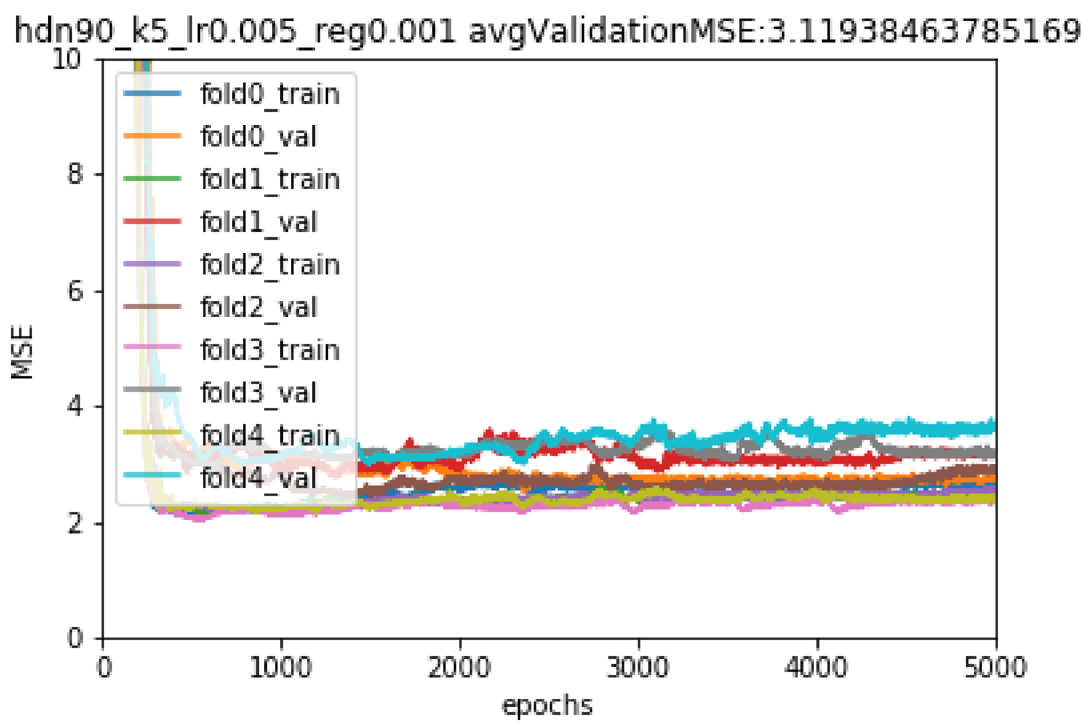


Figure 14: Learning Curve of hdn90_k5_lr0.005_reg0.001

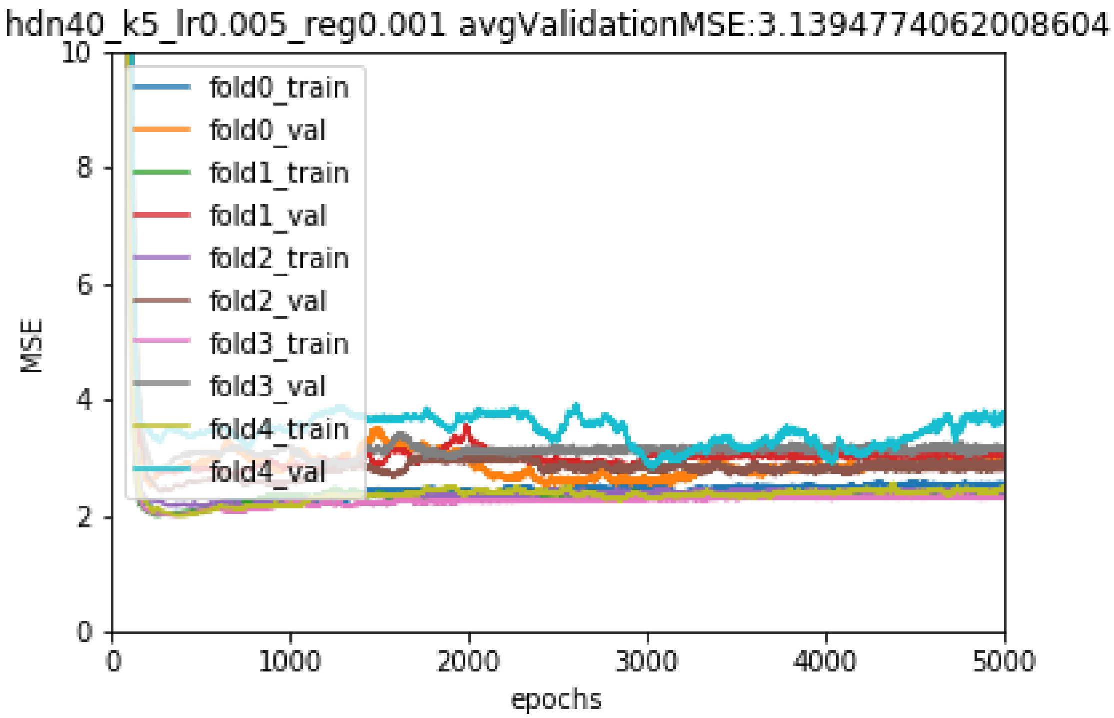


Figure 15: Learning Curve of hdn40_k5_lr0.005_reg0.001

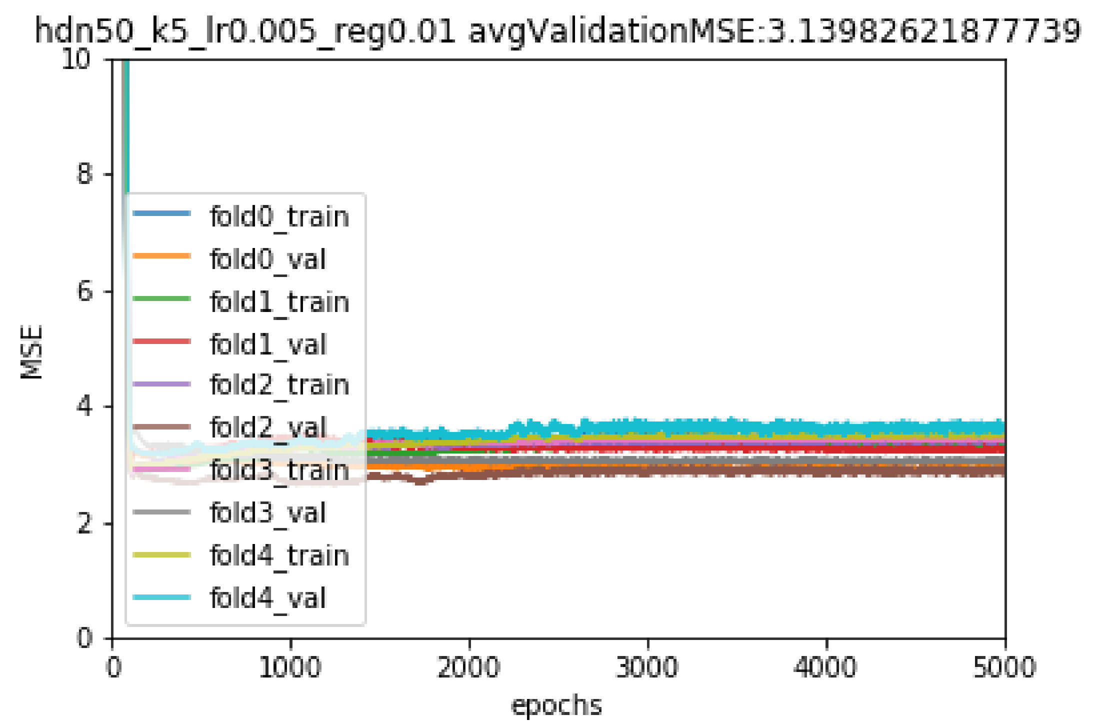


Figure 16: Learning Curve of hdn50_k5_lr0.005_reg0.01

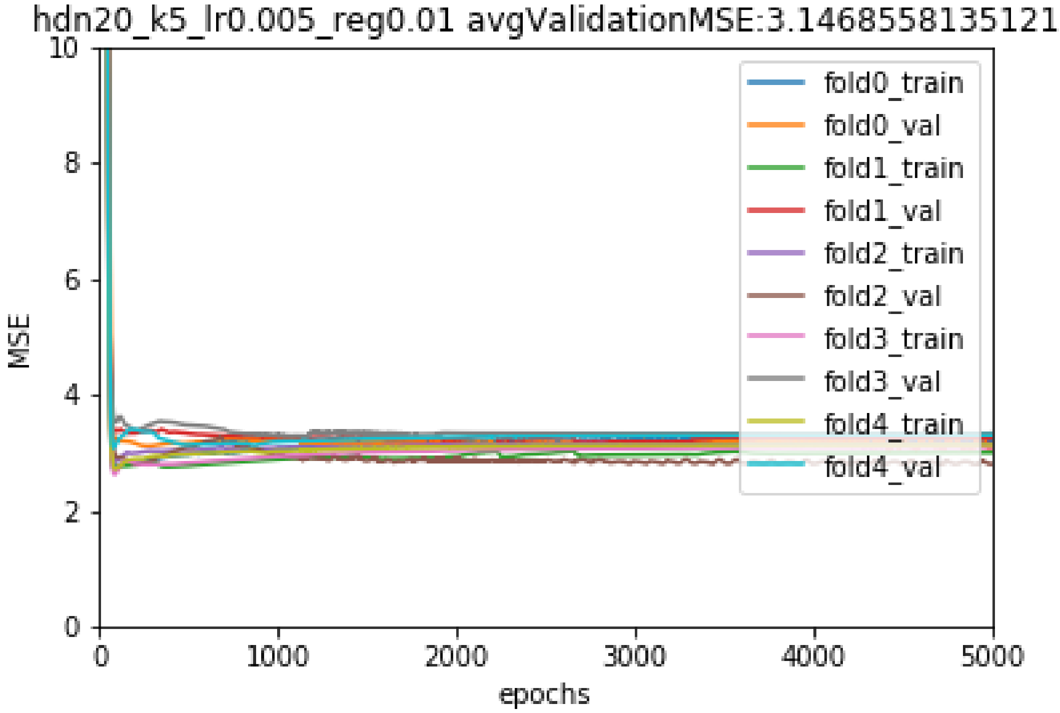


Figure 17: Learning Curve of hdn20_k5_lr0.005_reg0.01

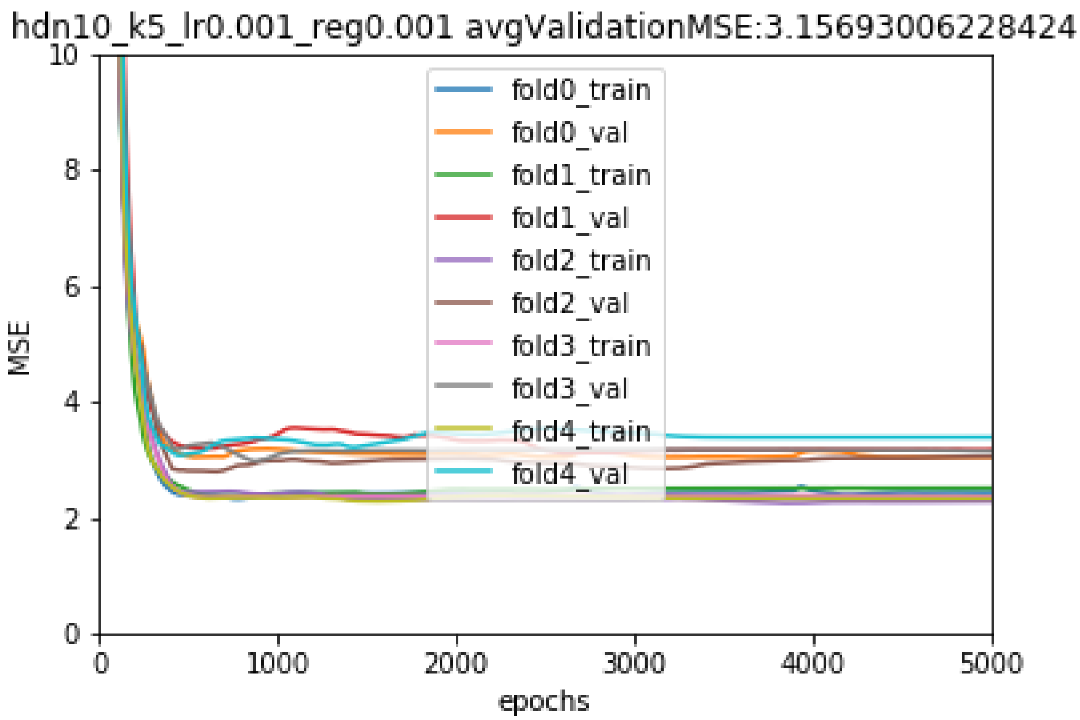


Figure 18: Learning Curve of hdn10_k5_lr0.001_reg0.001

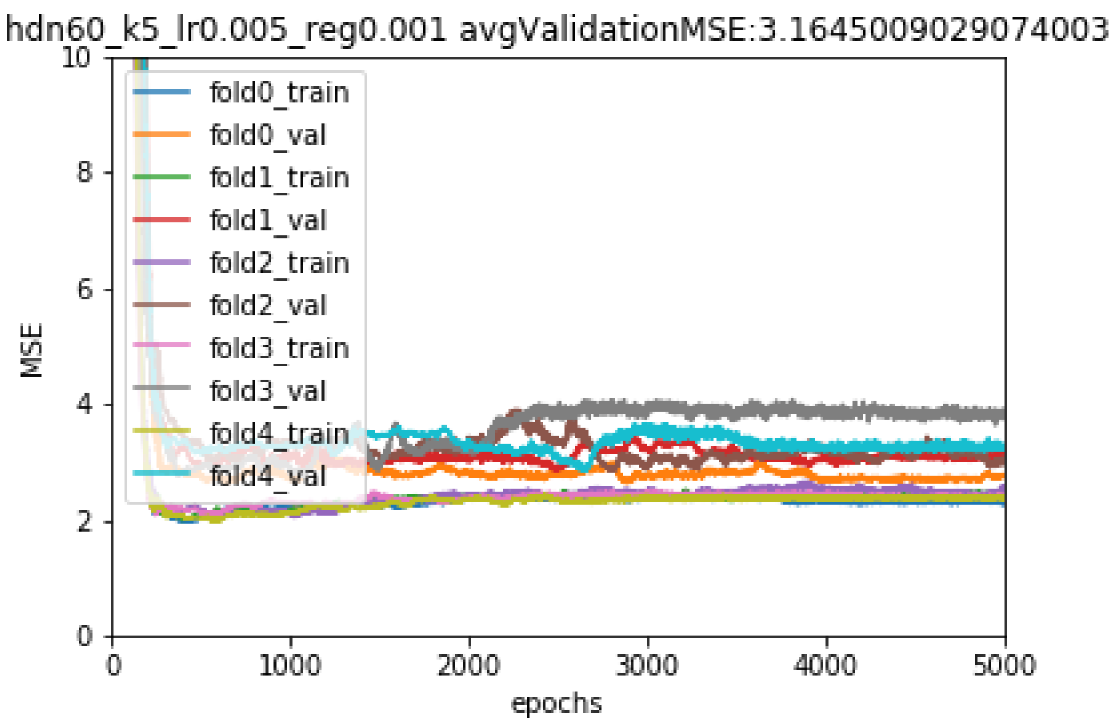


Figure 19: Learning Curve of hdn60_k5_lr0.005_reg0.001

The final choice is based on the following properties we defined:

1- it belongs to the top 20 least average MSEs
2- its curve is quite stable, [not so much flickering]
3- the validation curves are not performing radically different
4- the plot shows potential to either improve for more than 5000 epochs or at least already converging, in the former case, we will re-run the winner experiment to an extended number of epochs
5- vague plots where validation MSE is starting lower than training MSE are not considered

Perhaps the winner could be hdn100_lr0.001_reg0.001 as it belongs to the top 20, the curves are stable, not performing radically different and actually showing a potential to run for more than 5000 epochs and still improve, alternatively hdn80_lr0.001_reg0.001 seems also as a good choice, but without the same potential to improve after 5k epochs.

We decided a final experiment using the traditional Train/validation/test cross validation strategy with a 60% training, 20% validation and 20% testing reporting validation MSE and MEE for testing and validation splits, choosing the best of both based on the validation MSE score

Reproducing the same experiments results:
Please run the project AA1_CUP Program.cs for SGD+momentum and Adam screening

Simple usage example:
SGD+Momentum[Nestrov]

```
AA1_MLP.DataManagers.CupDataManager dm = new
AA1_MLP.DataManagers.CupDataManager();
DataSet trainDS = dm.LoadData("D:\\dropbox\\Dropbox\\Master Course\\SEM-
3\\ML\\CM_CUP_Datasets\\60percenttrain.txt", 10, 2, standardize: true);
DataSet testDS = dm.LoadData("D:\\dropbox\\Dropbox\\Master Course\\SEM-
3\\ML\\CM_CUP_Datasets\\60percenttest.txt", 10, 2, standardize: true);
GradientDescentParams passedParams = new GradientDescentParams();
Gradientdescent trainer = new Gradientdescent();
passedParams.numberOfEpochs = 5000;
passedParams.batchSize = 10;
passedParams.trainingSet = trainDS;
passedParams.validationSet = testDS;
passedParams.learningRate = 0.001;
passedParams.regularization = Regularizations.L2;
passedParams.regularizationRate = 0.001;
passedParams.nestrov = true;
passedParams.resilient = false;
passedParams.resilientUpdateAccelerationRate = 2;
passedParams.resilientUpdateSlowDownRate = 0.5;

passedParams.momentum = 0.5;
passedParams.NumberOfHiddenUnits = 100;

LastTrain(testDS, passedParams, trainer, "5kitr_mo0.5_100_final_sgdnestrov_hdn");
```

And Adam:

```
AA1_MLP.DataManagers.CupDataManager dm = new
AA1_MLP.DataManagers.CupDataManager();
DataSet trainDS = dm.LoadData("D:\\dropbox\\Dropbox\\Master Course\\SEM-
3\\ML\\CM_CUP_Datasets\\60percenttrain.txt", 10, 2, standardize: true);
DataSet testDS = dm.LoadData("D:\\dropbox\\Dropbox\\Master Course\\SEM-
3\\ML\\CM_CUP_Datasets\\60percenttest.txt", 10, 2, standardize: true);
AdamParams passedParams = new AdamParams();
IOptimizer trainer = new Adam();
passedParams.numberOfEpochs = 5000;
passedParams.batchSize = 10;
passedParams.trainingSet = trainDS;
passedParams.validationSet = testDS;
passedParams.learningRate = 0.001;
passedParams.regularization = Regularizations.L2;
passedParams.regularizationRate = 0.001;
passedParams.NumberOfHiddenUnits = 100;
```

```
LastTrain(testDS, passedParams, trainer, "5kitr_mo0.5_100_final_sgdnestrov_hdn");
```

The function `LastTrain` takes a training setting and dumps the log history of the learning curve train and validation MSEs along with the validation MEE in a text file and also the trained model as well.

b. SGD+Momentum

We implemented a trainer utilizing the stochastic gradient descent algorithm with the following hyper parameters

SGD+Momentum hyperparameters:

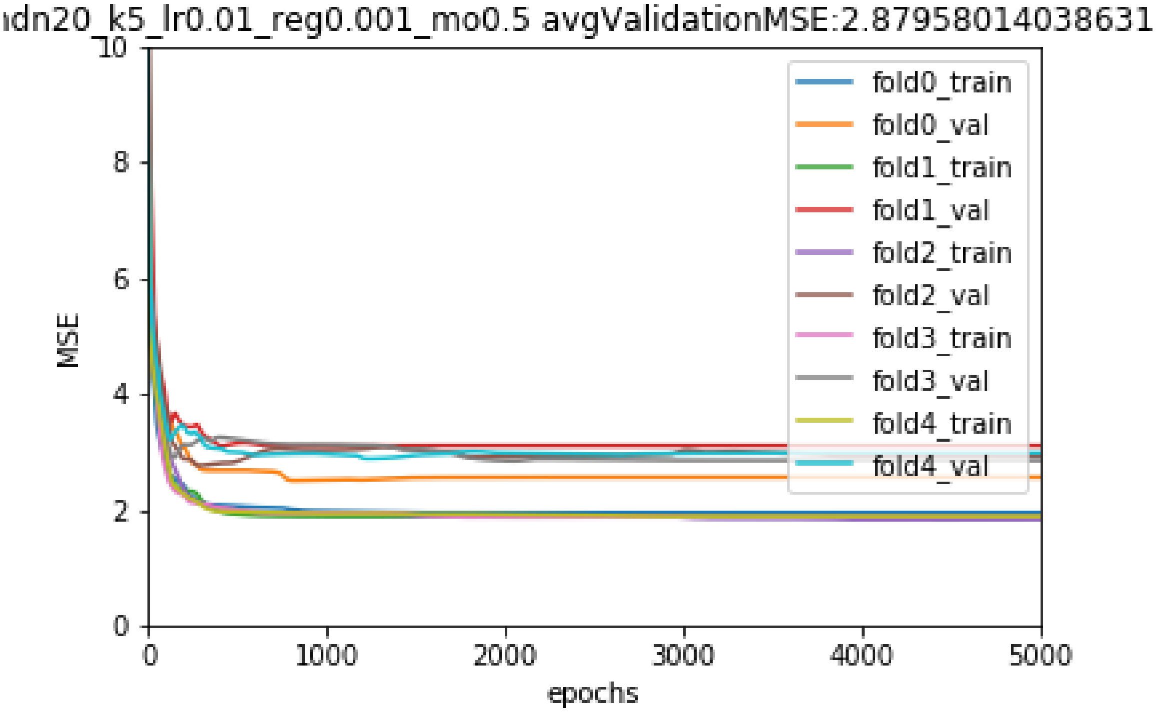
Momentum [both classic and nesterov implementations]
For the SGD+Momentum optimizer, we introduced the following hyperparameters;
Momentum rates [0, 0.5]
Learning Rates [0.005, 0.01]
Regularization Rates [0, 0.001]
Nesterov: [on or off]

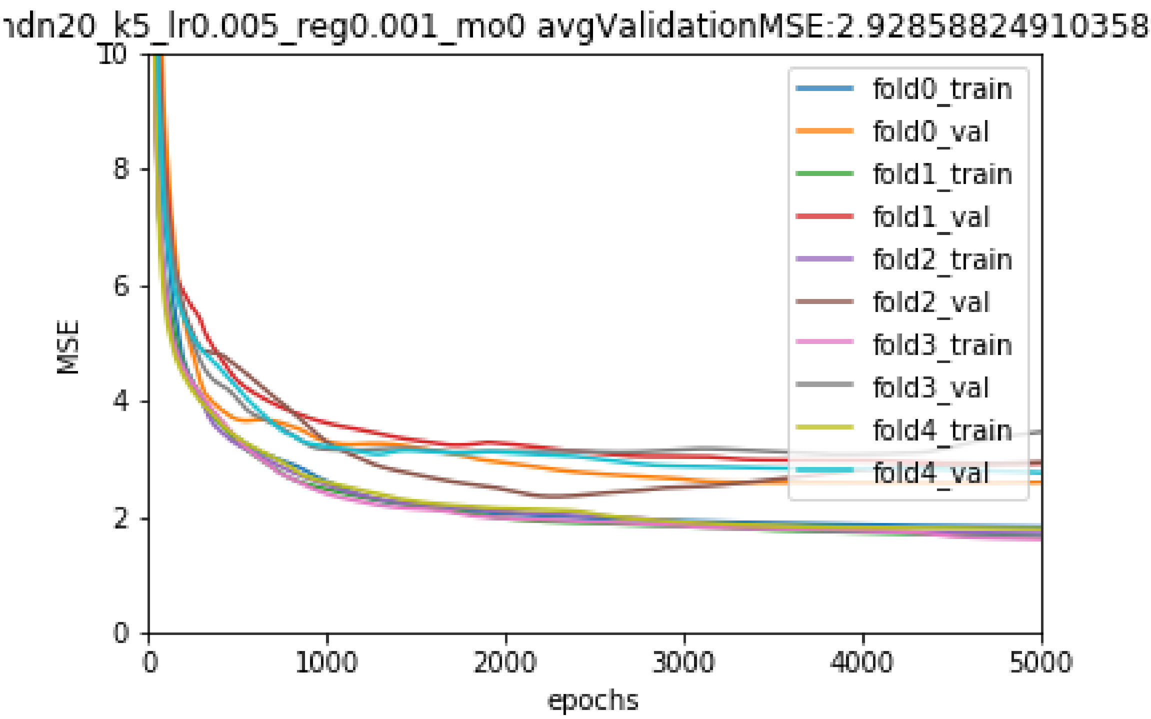
SGD+Momentum [Nesterov] top 20 experiments report:

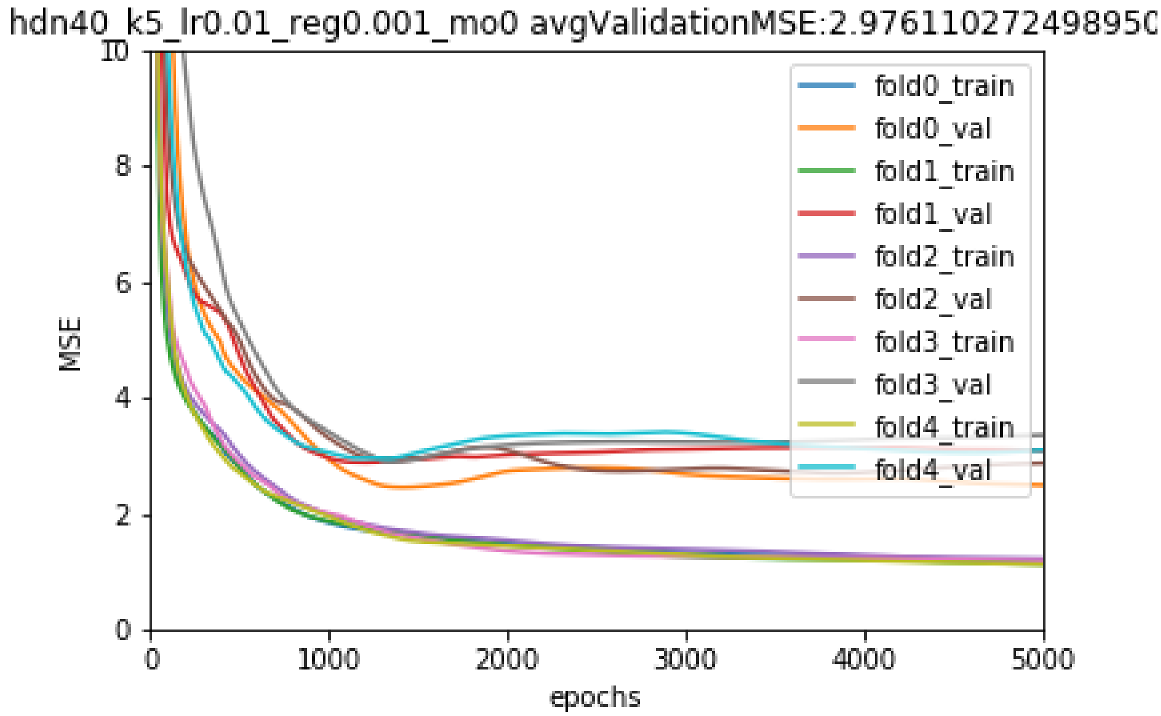
39	hdn20_k5_lr0.01_reg0.001_mo0.5	2.879580
33	hdn20_k5_lr0.005_reg0.001_mo0	2.928588
27	hdn40_k5_lr0.01_reg0.001_mo0	2.976110
25	hdn40_k5_lr0.005_reg0.001_mo0	3.011973
35	hdn20_k5_lr0.01_reg0.001_mo0	3.017527
11	hdn80_k5_lr0.01_reg0.001_mo0	3.040099
1	hdn100_k5_lr0.005_reg0.001_mo0	3.049197
19	hdn60_k5_lr0.01_reg0.001_mo0	3.069511
37	hdn20_k5_lr0.005_reg0.001_mo0.5	3.077974
17	hdn60_k5_lr0.005_reg0.001_mo0	3.088708
9	hdn80_k5_lr0.005_reg0.001_mo0	3.155544
29	hdn40_k5_lr0.005_reg0.001_mo0.5	3.251418
31	hdn40_k5_lr0.01_reg0.001_mo0.5	3.267038
5	hdn100_k5_lr0.005_reg0.001_mo0.5	3.289246
3	hdn100_k5_lr0.01_reg0.001_mo0	3.292397
21	hdn60_k5_lr0.005_reg0.001_mo0.5	3.292790
13	hdn80_k5_lr0.005_reg0.001_mo0.5	3.380744
7	hdn100_k5_lr0.01_reg0.001_mo0.5	3.397745
15	hdn80_k5_lr0.01_reg0.001_mo0.5	3.500005
23	hdn60_k5_lr0.01_reg0.001_mo0.5	3.589723

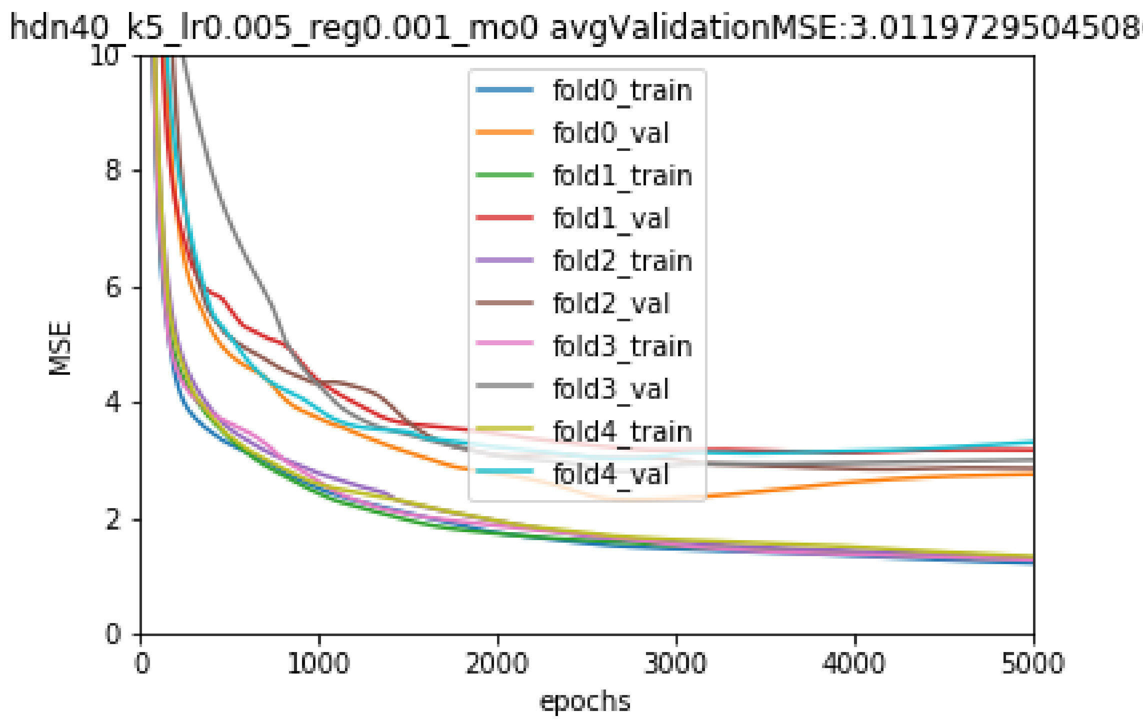
Table 6 - Top 20 experiments

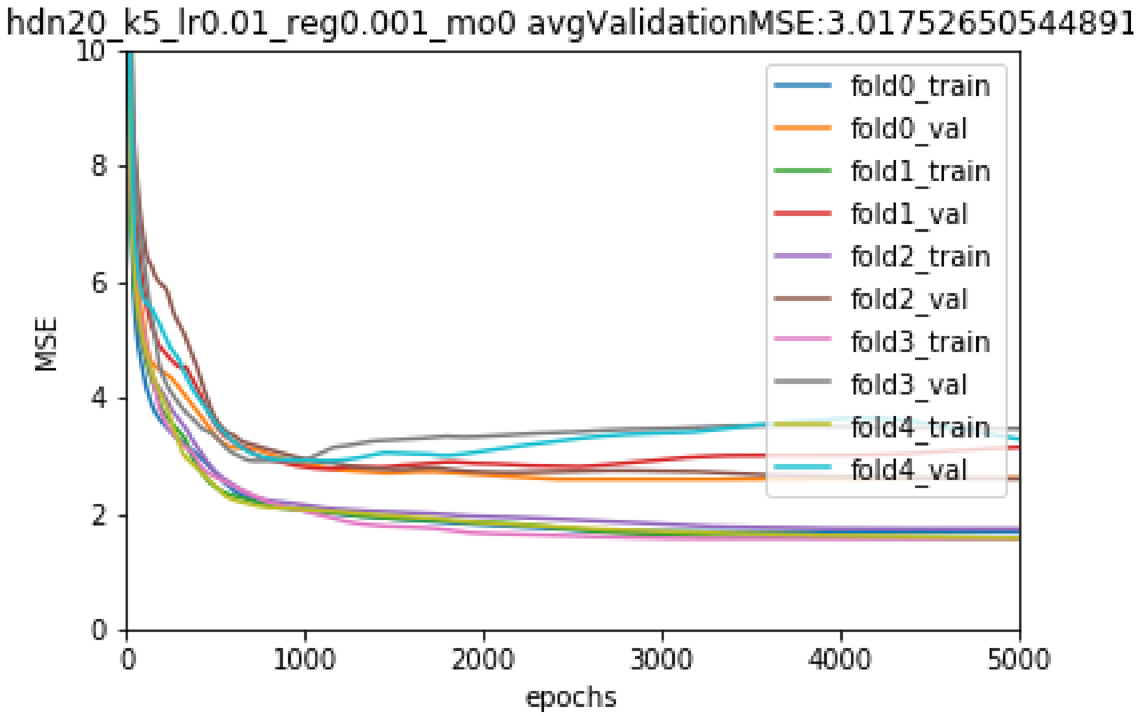
Plots:

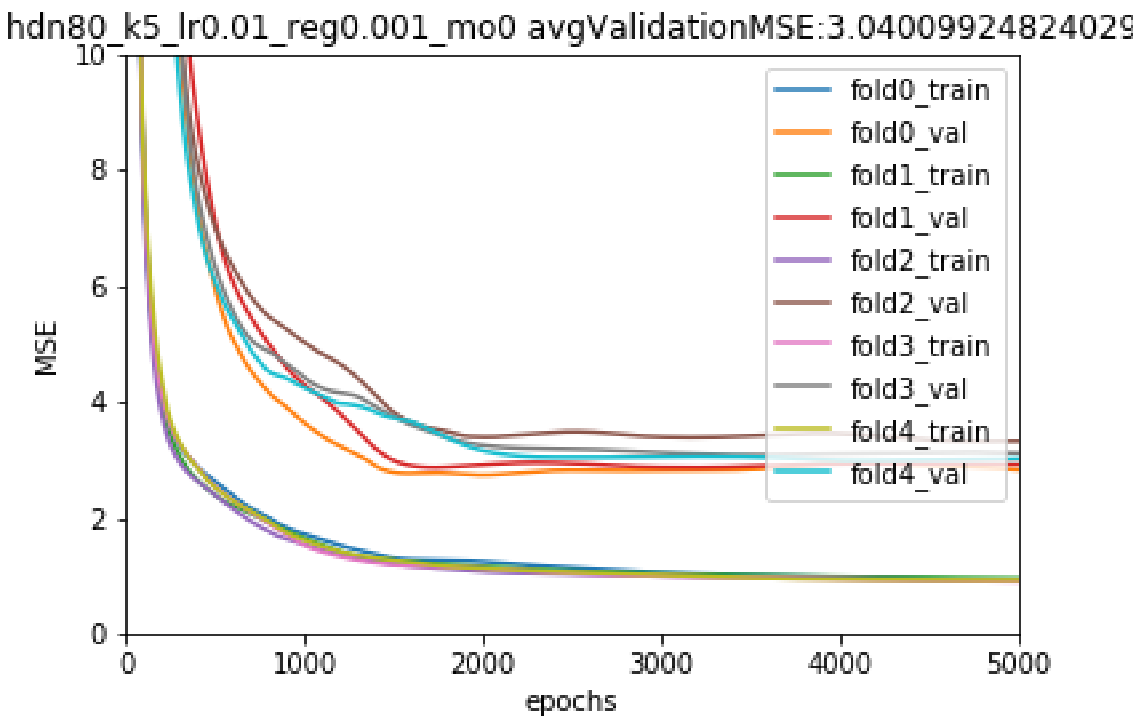

ıdn20_k5_lr0.01_reg0.001_mo0.5 avgValidationMSE:2.87958014038631
MSE
epochs
0
2
4
6
8
10
0
1000
2000
3000
4000
5000
fold0_train
fold0_val
fold1_train
fold1_val
fold2_train
fold2_val
fold3_train
fold3_val
fold4_train
fold4_val

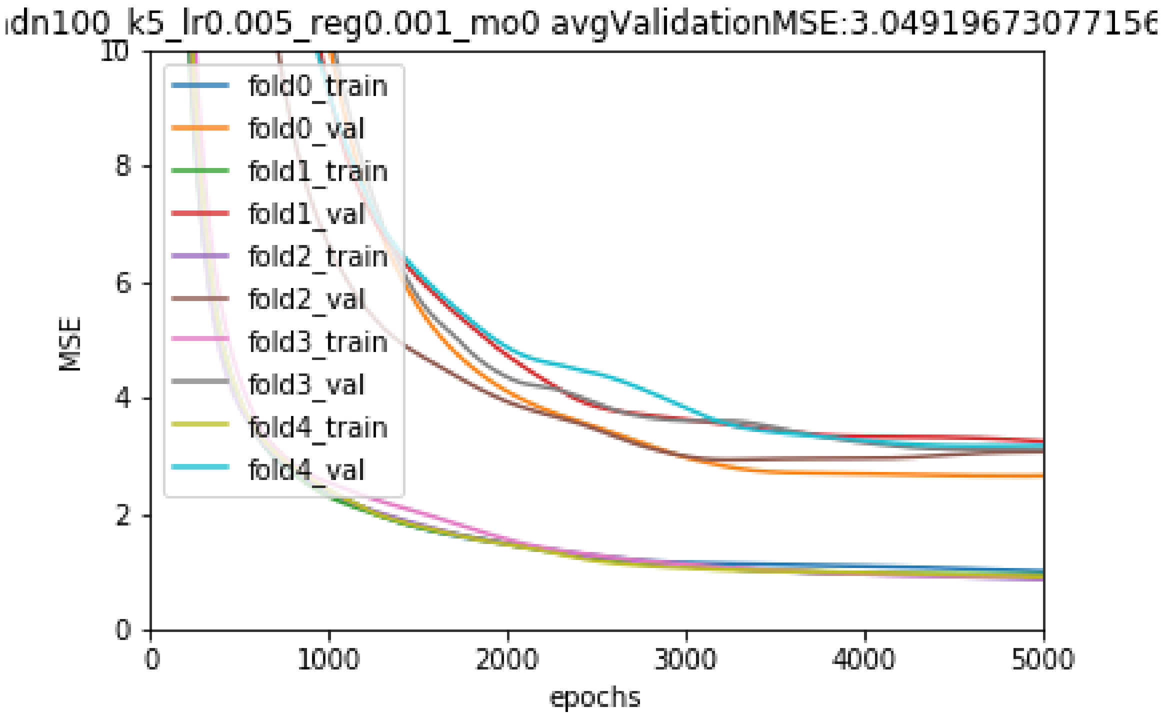

ıdn20_k5_lr0.005_reg0.001_mo0 avgValidationMSE:2.92858824910358
MSE
epochs
0
2
4
6
8
10
0
1000
2000
3000
4000
5000
fold0_train
fold0_val
fold1_train
fold1_val
fold2_train
fold2_val
fold3_train
fold3_val
fold4_train
fold4_val

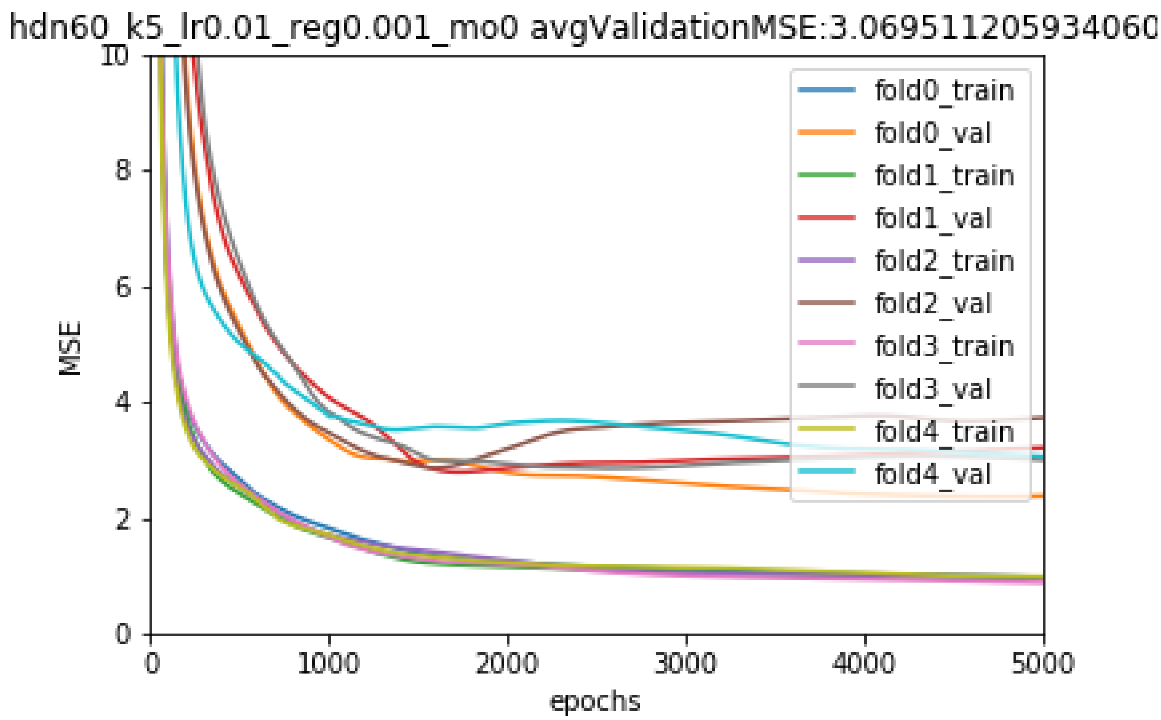

hdn40_k5_lr0.01_reg0.001_mo0 avgValidationMSE:2.97611027249895C
MSE
epochs
fold0_train
fold0_val
fold1_train
fold1_val
fold2_train
fold2_val
fold3_train
fold3_val
fold4_train
fold4_val
0
2
4
6
8
10
0
1000
2000
3000
4000
5000

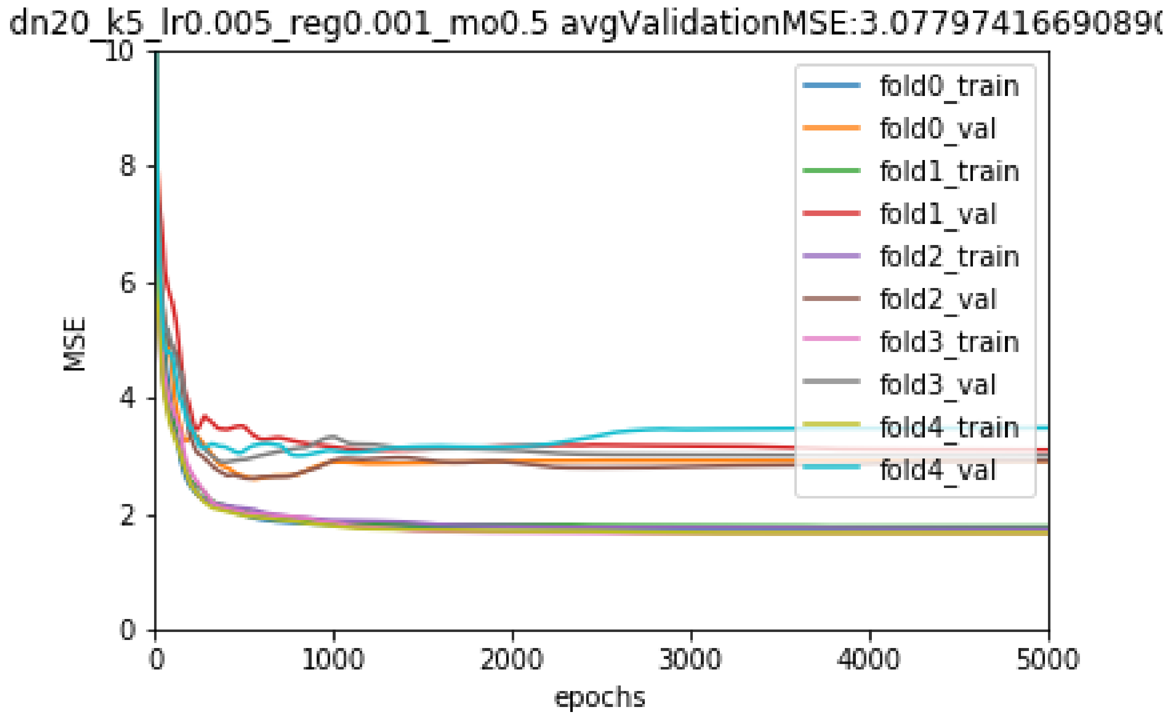

hdn40_k5_lr0.005_reg0.001_mo0 avgValidationMSE:3.0119729504508
MSE
epochs
fold0_train
fold0_val
fold1_train
fold1_val
fold2_train
fold2_val
fold3_train
fold3_val
fold4_train
fold4_val
0
2
4
6
8
10
0
1000
2000
3000
4000
5000

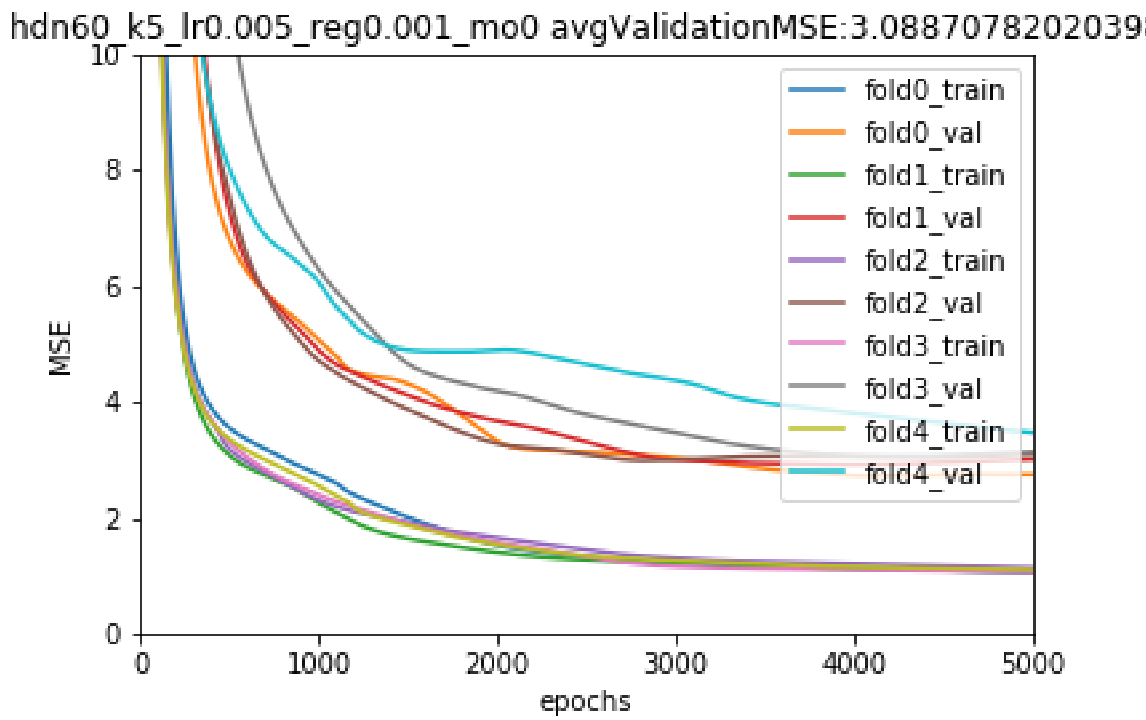

hdn20_k5_lr0.01_reg0.001_mo0 avgValidationMSE:3.01752650544891
MSE
epochs
fold0_train
fold0_val
fold1_train
fold1_val
fold2_train
fold2_val
fold3_train
fold3_val
fold4_train
fold4_val
0
2
4
6
8
10
0
1000
2000
3000
4000
5000

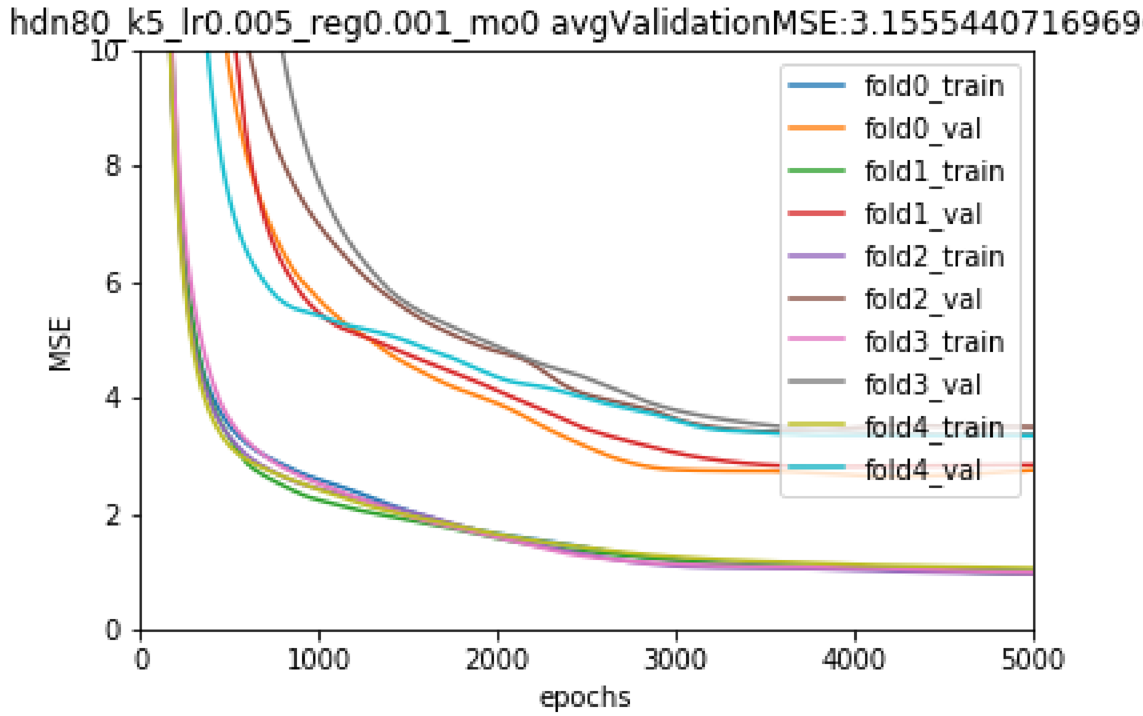

hdn80_k5_lr0.01_reg0.001_mo0 avgValidationMSE:3.04009924824029
MSE
epochs
fold0_train
fold0_val
fold1_train
fold1_val
fold2_train
fold2_val
fold3_train
fold3_val
fold4_train
fold4_val
0
2
4
6
8
10
0
1000
2000
3000
4000
5000

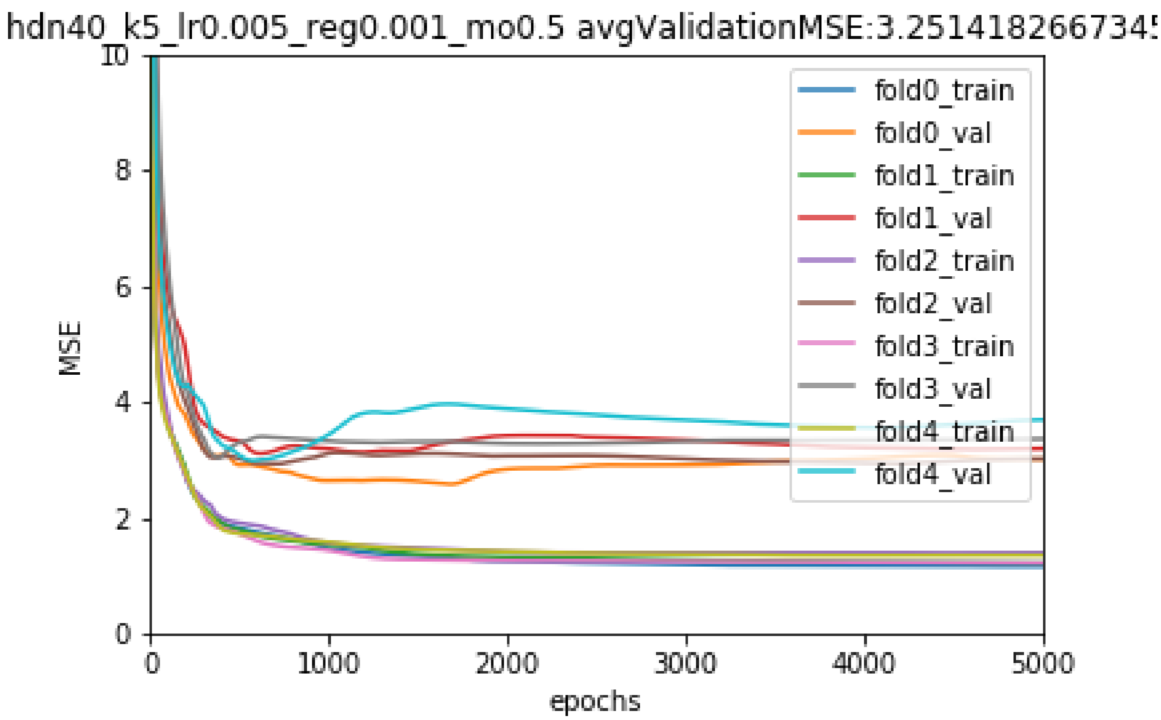

ıdn100_k5_lr0.005_reg0.001_mo0 avgValidationMSE:3.0491967307715
MSE
epochs
0
2
4
6
8
10
0
1000
2000
3000
4000
5000
fold0_train
fold0_val
fold1_train
fold1_val
fold2_train
fold2_val
fold3_train
fold3_val
fold4_train
fold4_val

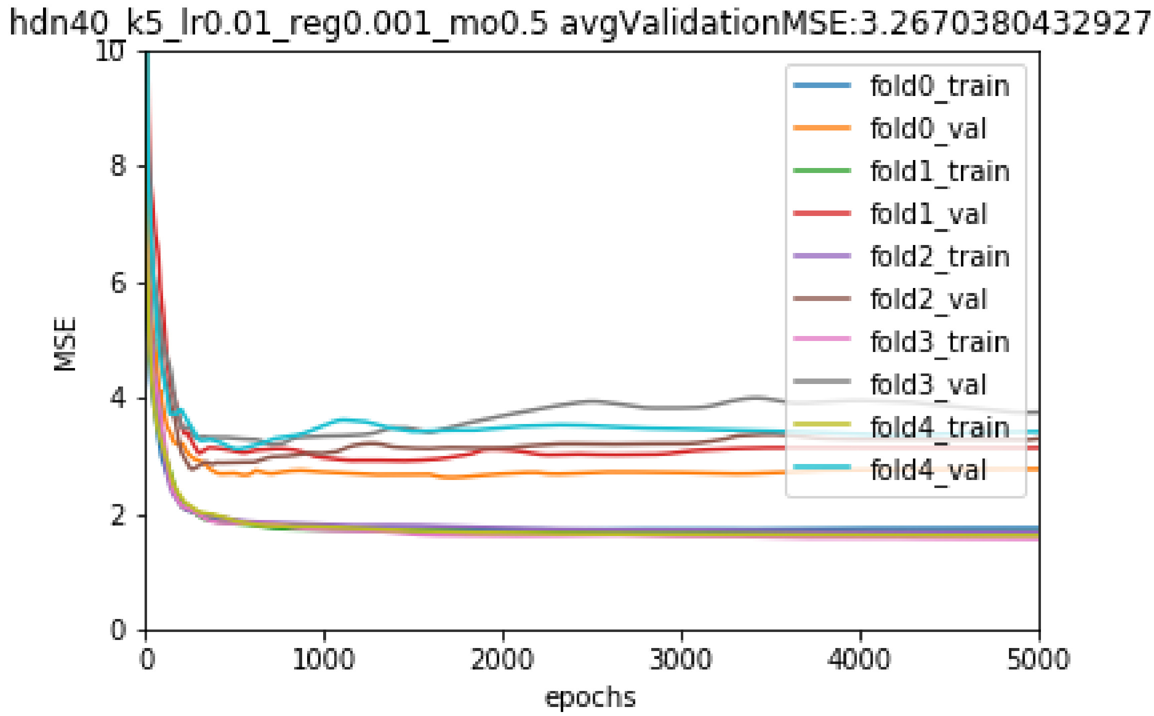

hdn60_k5_lr0.01_reg0.001_mo0 avgValidationMSE:3.069511205934060
MSE
epochs
0
2
4
6
8
10
0
1000
2000
3000
4000
5000
fold0_train
fold0_val
fold1_train
fold1_val
fold2_train
fold2_val
fold3_train
fold3_val
fold4_train
fold4_val

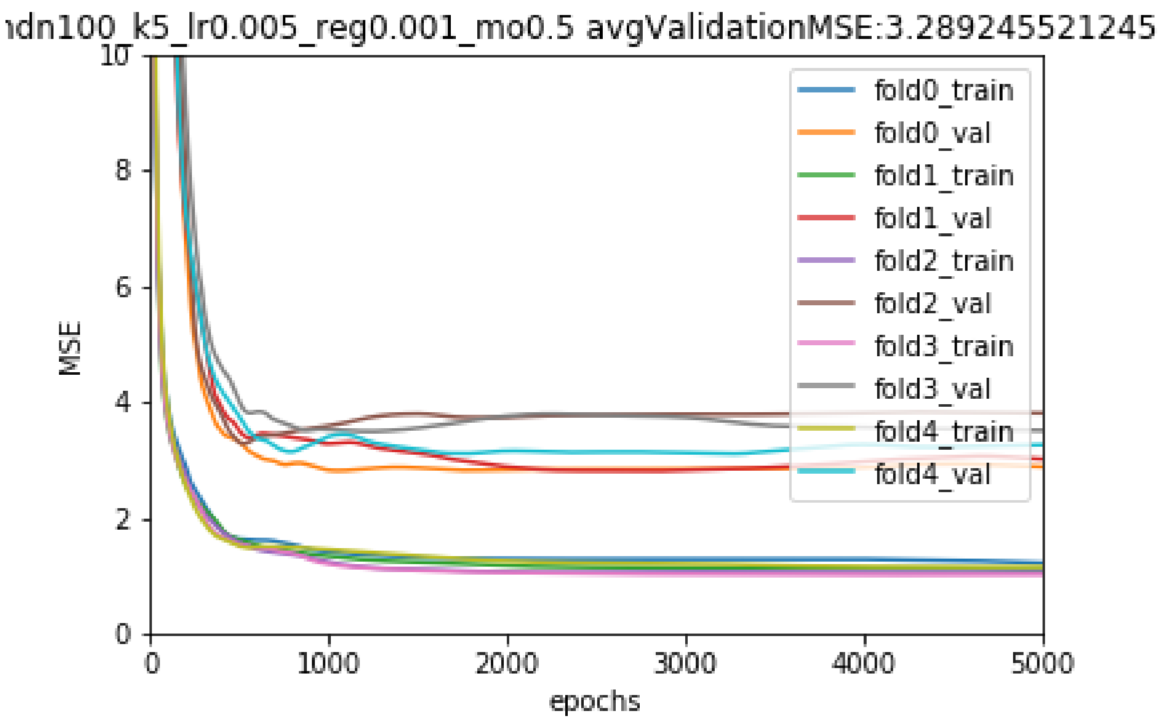

dn20_k5_lr0.005_reg0.001_mo0.5 avgValidationMSE:3.0779741669089
MSE
epochs
fold0_train
fold0_val
fold1_train
fold1_val
fold2_train
fold2_val
fold3_train
fold3_val
fold4_train
fold4_val

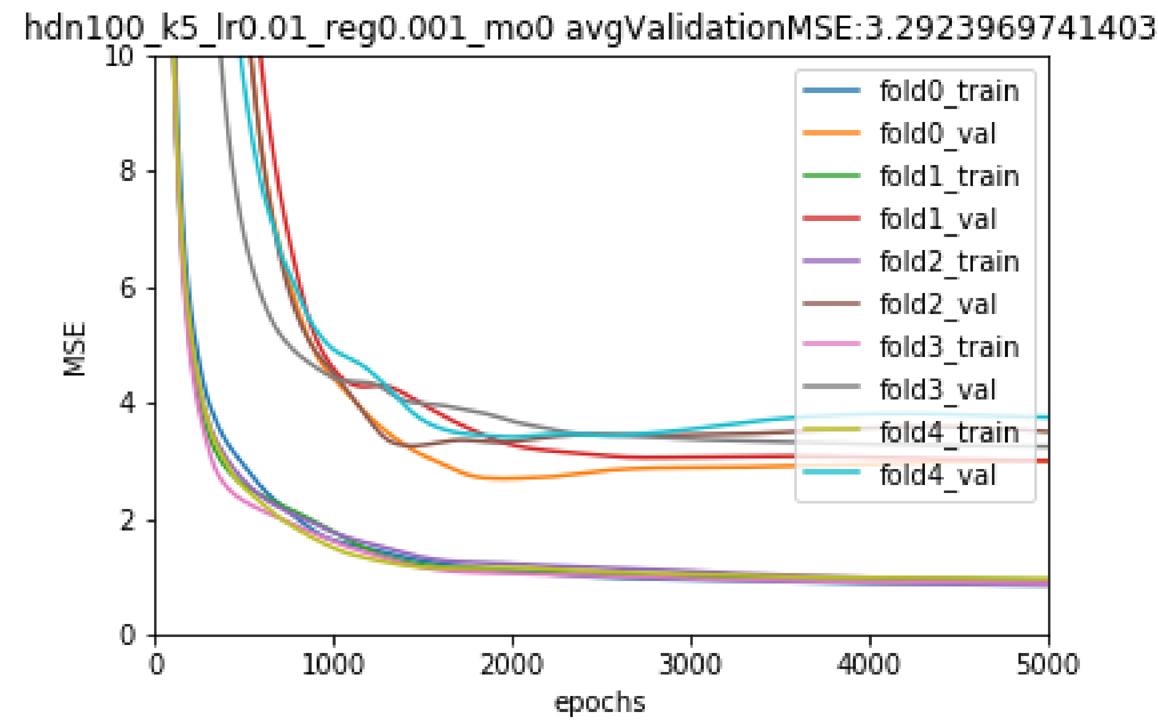

hdn60_k5_lr0.005_reg0.001_mo0 avgValidationMSE:3.088707820203
MSE
epochs
fold0_train
fold0_val
fold1_train
fold1_val
fold2_train
fold2_val
fold3_train
fold3_val
fold4_train
fold4_val

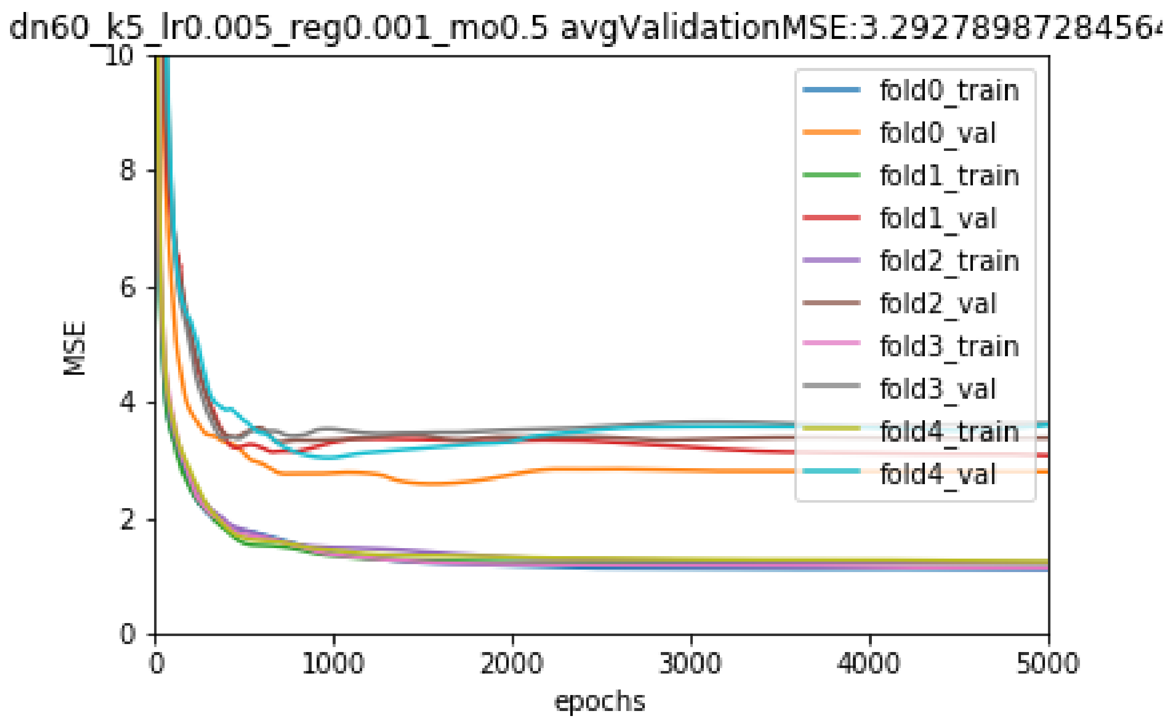

hdn80_k5_lr0.005_reg0.001_mo0 avgValidationMSE:3.1555440716969
MSE
epochs
fold0_train
fold0_val
fold1_train
fold1_val
fold2_train
fold2_val
fold3_train
fold3_val
fold4_train
fold4_val

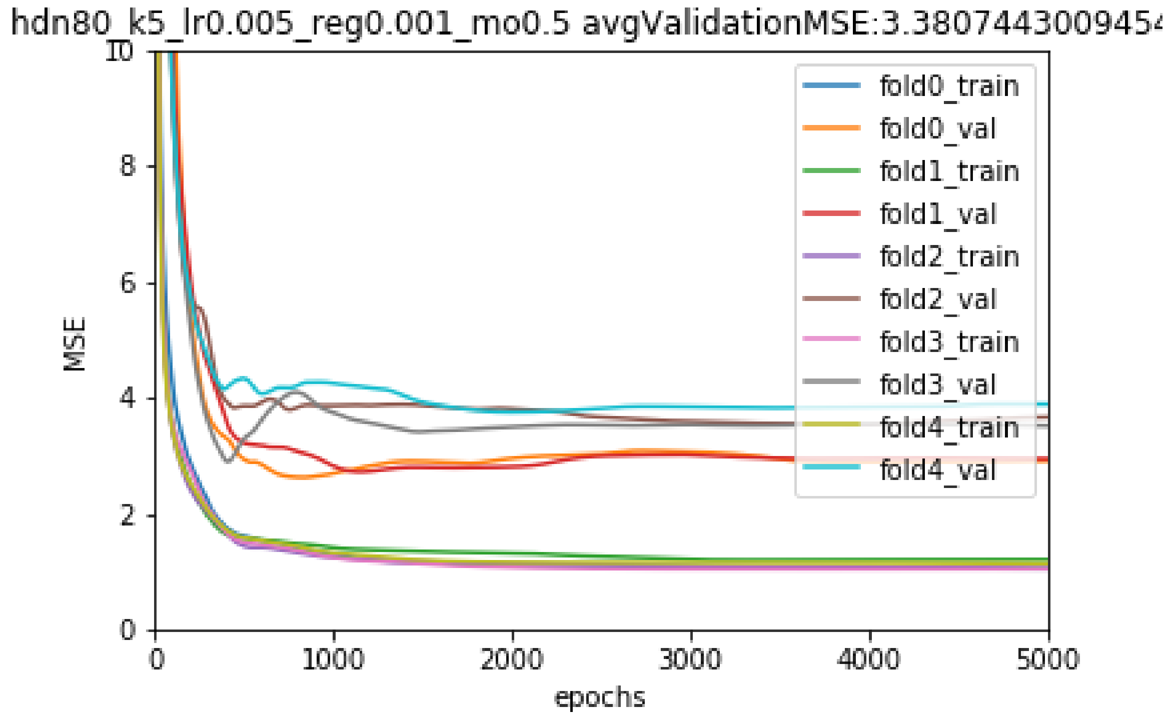

hdn40_k5_lr0.005_reg0.001_mo0.5 avgValidationMSE:3.251418267345
MSE
epochs
fold0_train
fold0_val
fold1_train
fold1_val
fold2_train
fold2_val
fold3_train
fold3_val
fold4_train
fold4_val

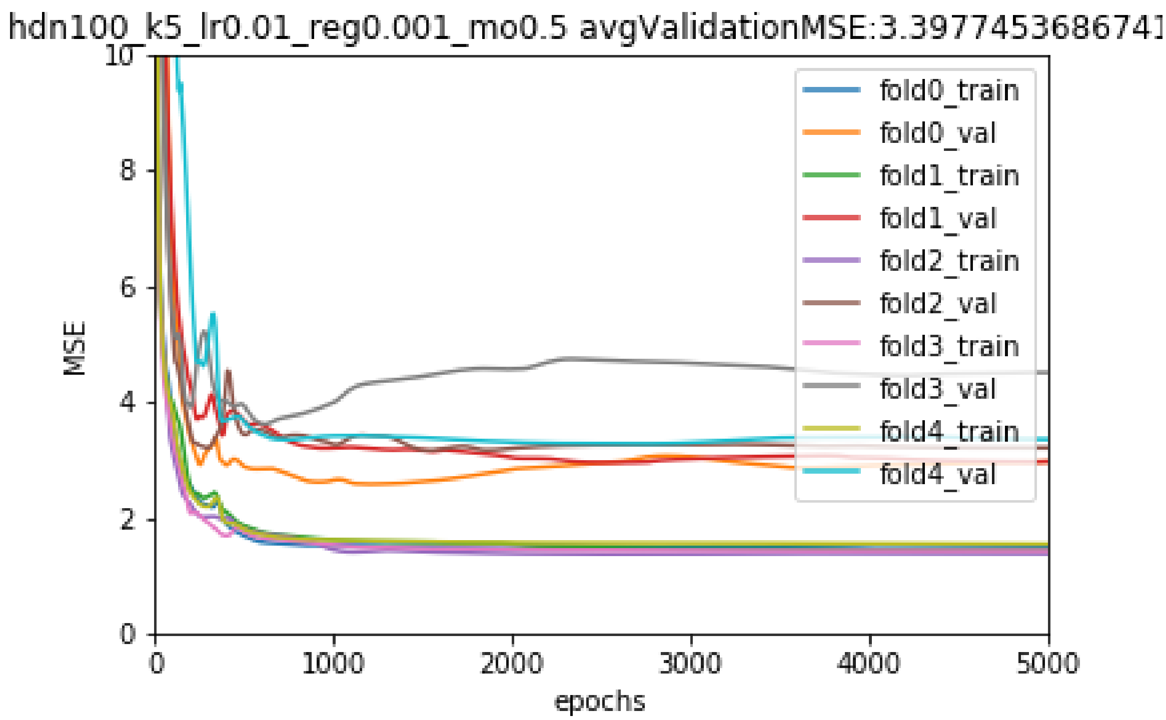

hdn40_k5_lr0.01_reg0.001_mo0.5 avgValidationMSE:3.2670380432927
MSE
epochs
0
2
4
6
8
10
0
1000
2000
3000
4000
5000
fold0_train
fold0_val
fold1_train
fold1_val
fold2_train
fold2_val
fold3_train
fold3_val
fold4_train
fold4_val

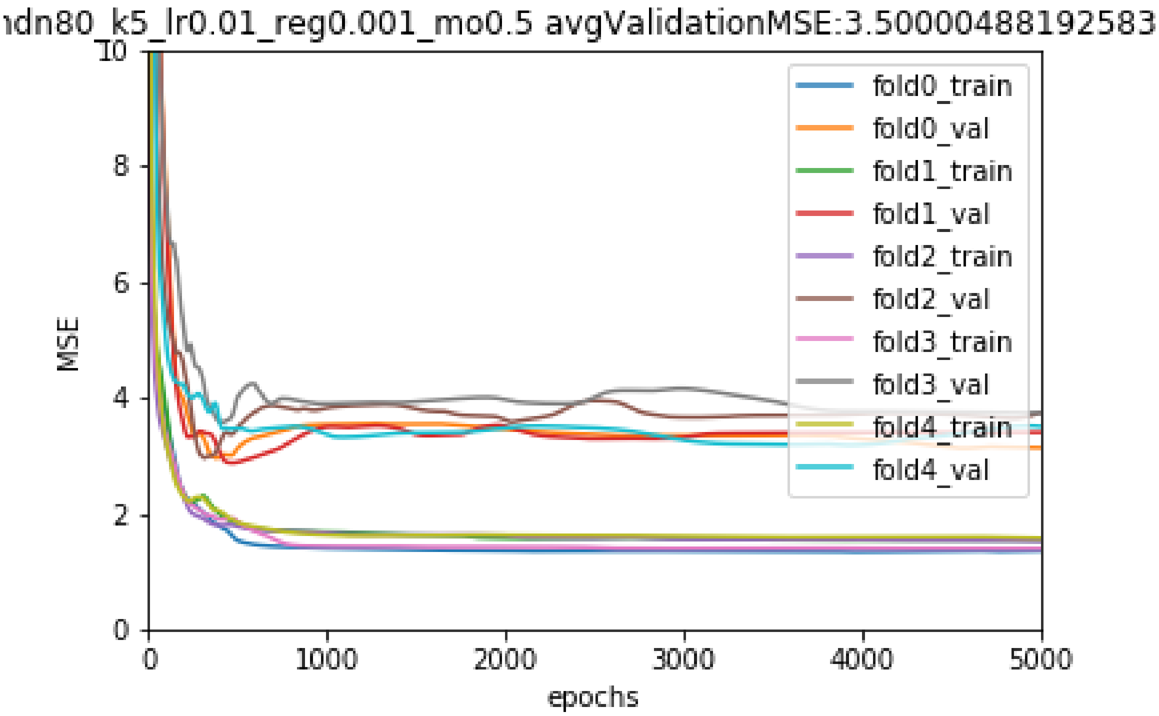

dn100_k5_lr0.005_reg0.001_mo0.5 avgValidationMSE:3.289245521245
MSE
epochs
0
2
4
6
8
10
0
1000
2000
3000
4000
5000
fold0_train
fold0_val
fold1_train
fold1_val
fold2_train
fold2_val
fold3_train
fold3_val
fold4_train
fold4_val

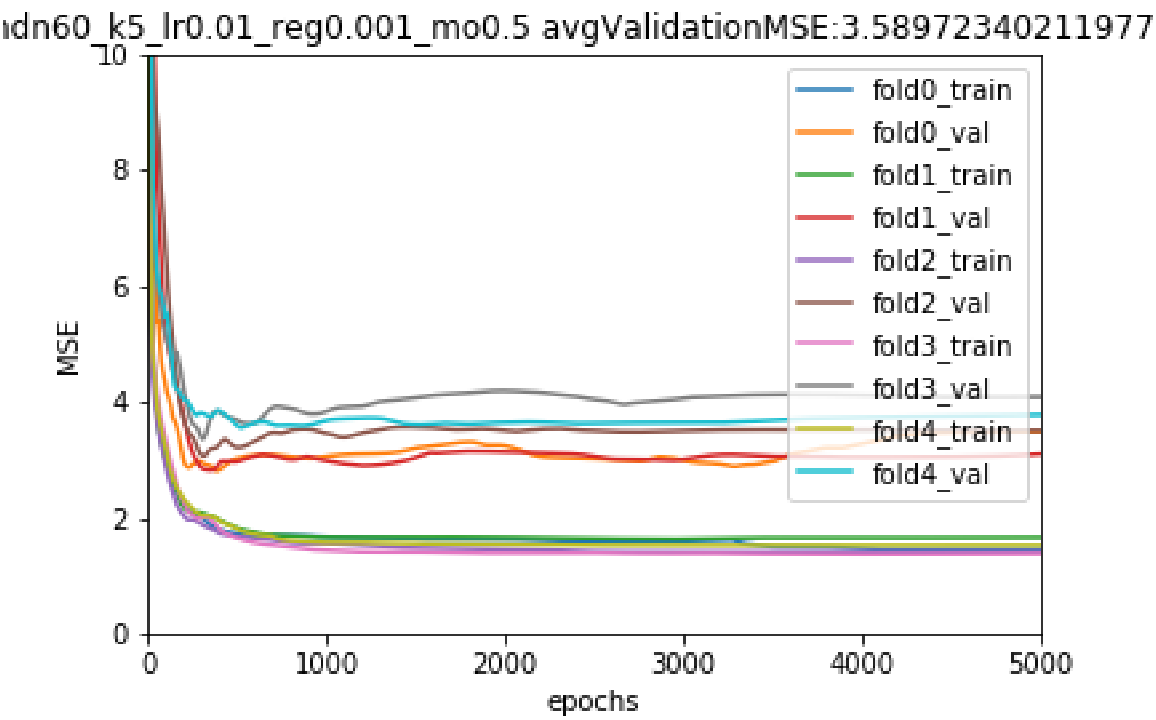

hdn100_k5_lr0.01_reg0.001_mo0 avgValidationMSE:3.29239697414038
MSE
epochs
0
2
4
6
8
10
0
1000
2000
3000
4000
5000
fold0_train
fold0_val
fold1_train
fold1_val
fold2_train
fold2_val
fold3_train
fold3_val
fold4_train
fold4_val


dn60_k5_lr0.005_reg0.001_mo0.5 avgValidationMSE:3.2927898728456
MSE
epochs
0
2
4
6
8
10
0
1000
2000
3000
4000
5000
fold0_train
fold0_val
fold1_train
fold1_val
fold2_train
fold2_val
fold3_train
fold3_val
fold4_train
fold4_val


hdn80_k5_lr0.005_reg0.001_mo0.5 avgValidationMSE:3.380744300945
MSE
epochs
fold0_train
fold0_val
fold1_train
fold1_val
fold2_train
fold2_val
fold3_train
fold3_val
fold4_train
fold4_val


hdn100_k5_lr0.01_reg0.001_mo0.5 avgValidationMSE:3.397745368674
MSE
epochs
fold0_train
fold0_val
fold1_train
fold1_val
fold2_train
fold2_val
fold3_train
fold3_val
fold4_train
fold4_val





For a complete list of experiments results and learning curves plot, please visit the folder sgdnestrovfinal

Again, with the same criteria we defined for picking the model earlier in Adam's report,

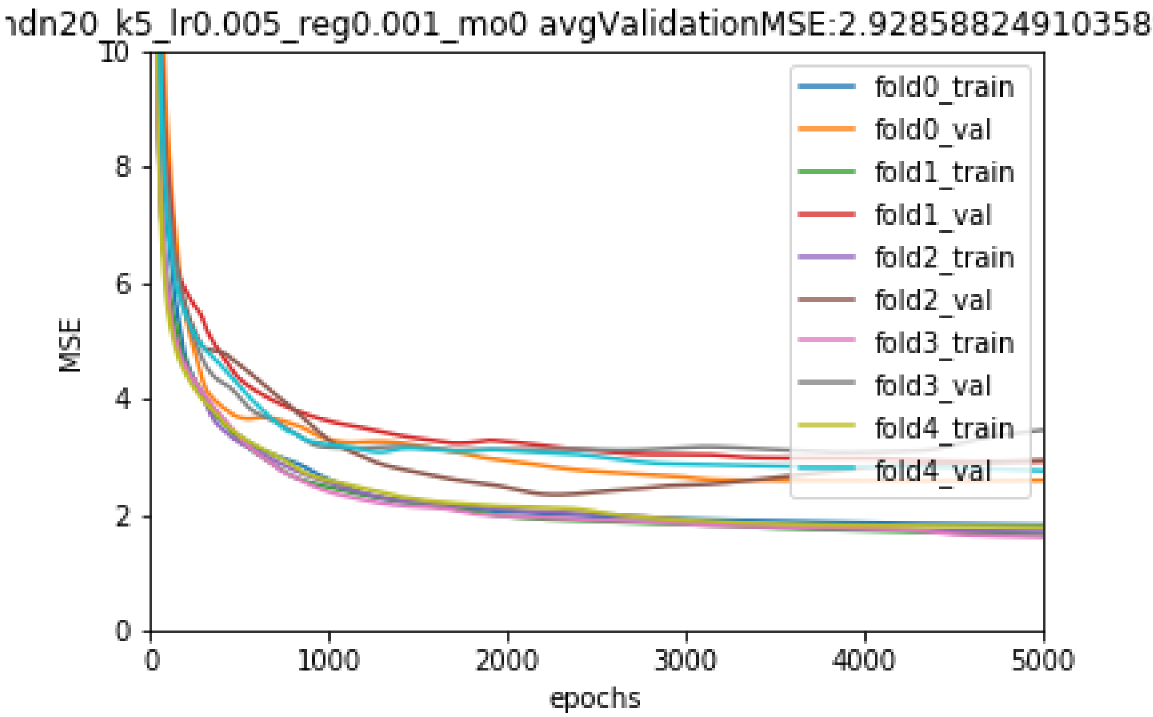
sgd+ nestrov

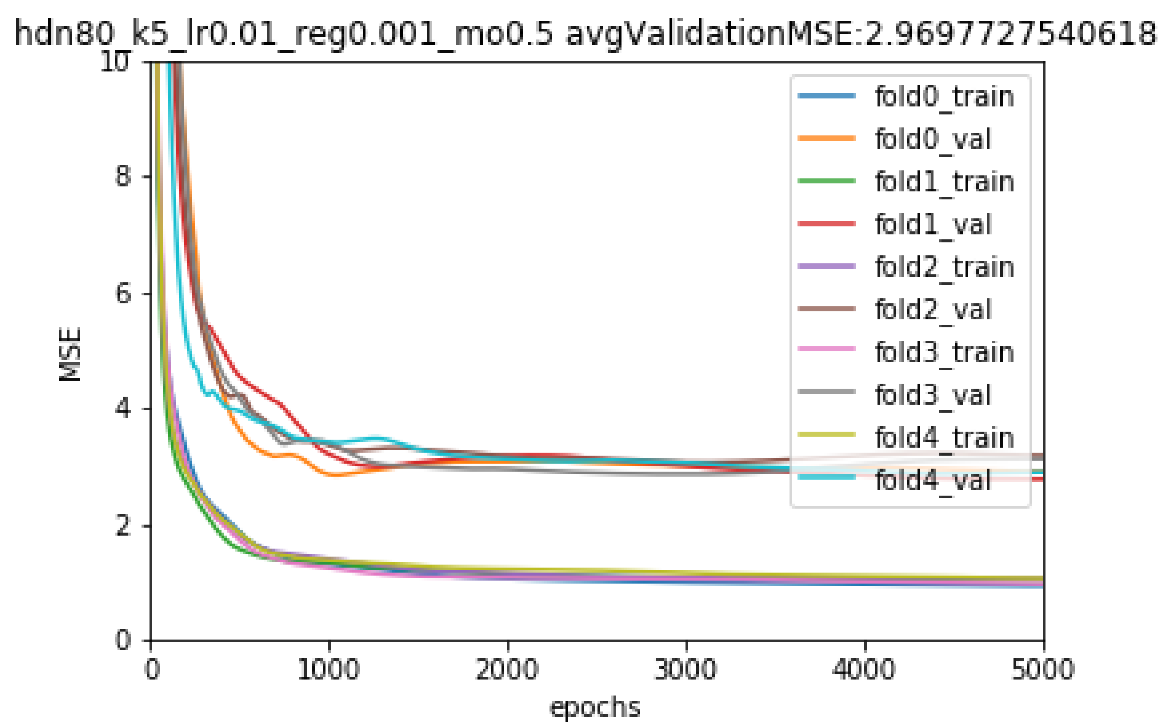
hdn100_lr0.005_reg0.001_mo0.5

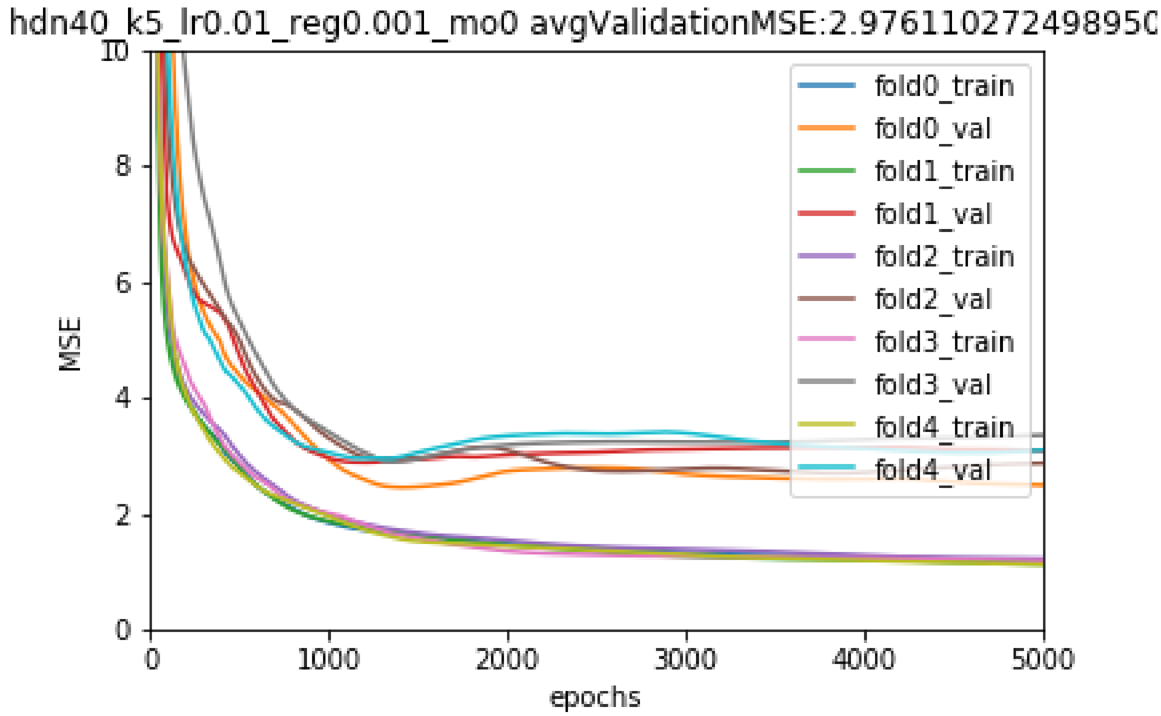
hidden 100
LearningRate 0.005
Regularization 0.001
Momentum 0.5

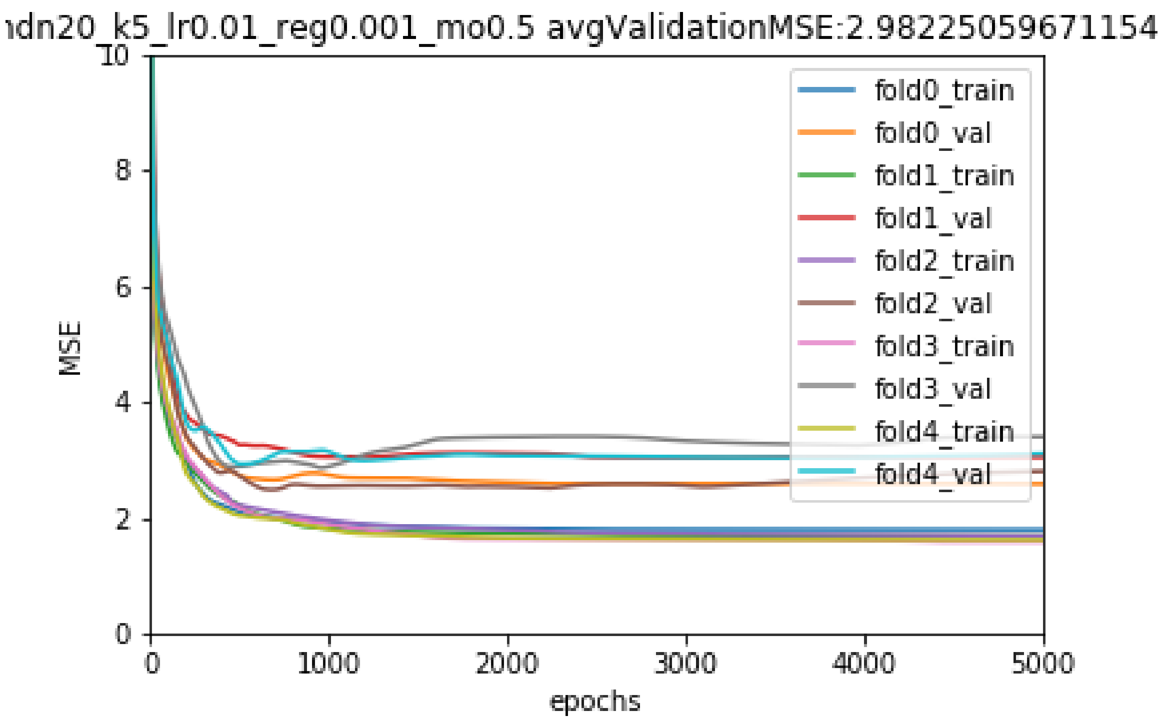
SGD+Momentum [Classic Momentum] [No Nesterov] top 20 experiments report:

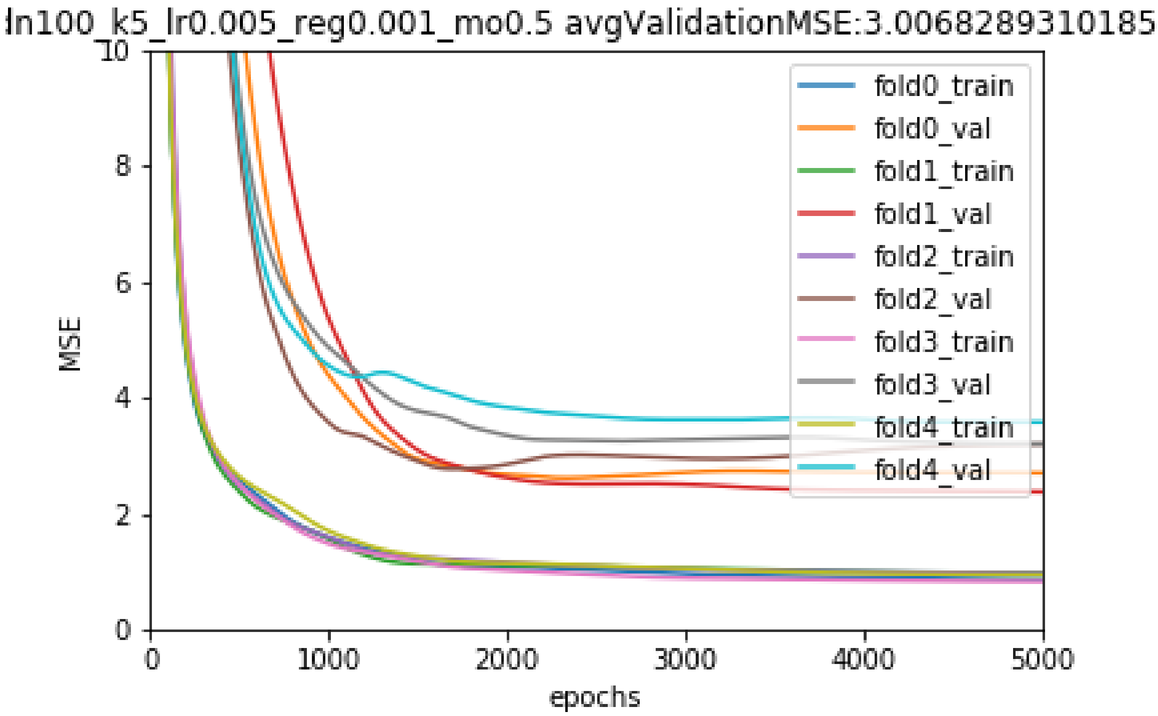
33	hdn20_k5_lr0.005_reg0.001_mo0	2.928588
15	hdn80_k5_lr0.01_reg0.001_mo0.5	2.969773
27	hdn40_k5_lr0.01_reg0.001_mo0	2.976110
39	hdn20_k5_lr0.01_reg0.001_mo0.5	2.982251
5	hdn100_k5_lr0.005_reg0.001_mo0.5	3.006829
23	hdn60_k5_lr0.01_reg0.001_mo0.5	3.007029
29	hdn40_k5_lr0.005_reg0.001_mo0.5	3.011795
25	hdn40_k5_lr0.005_reg0.001_mo0	3.011973
35	hdn20_k5_lr0.01_reg0.001_mo0	3.017527
21	hdn60_k5_lr0.005_reg0.001_mo0.5	3.032281
37	hdn20_k5_lr0.005_reg0.001_mo0.5	3.033151
7	hdn100_k5_lr0.01_reg0.001_mo0.5	3.039885
11	hdn80_k5_lr0.01_reg0.001_mo0	3.040099
13	hdn80_k5_lr0.005_reg0.001_mo0.5	3.040472
1	hdn100_k5_lr0.005_reg0.001_mo0	3.049197
19	hdn60_k5_lr0.01_reg0.001_mo0	3.069511
17	hdn60_k5_lr0.005_reg0.001_mo0	3.088708
9	hdn80_k5_lr0.005_reg0.001_mo0	3.155544
31	hdn40_k5_lr0.01_reg0.001_mo0.5	3.280586
3	hdn100_k5_lr0.01_reg0.001_mo0	3.292397


ıdn20_k5_lr0.005_reg0.001_mo0 avgValidationMSE:2.92858824910358!
MSE
epochs
fold0_train
fold0_val
fold1_train
fold1_val
fold2_train
fold2_val
fold3_train
fold3_val
fold4_train
fold4_val

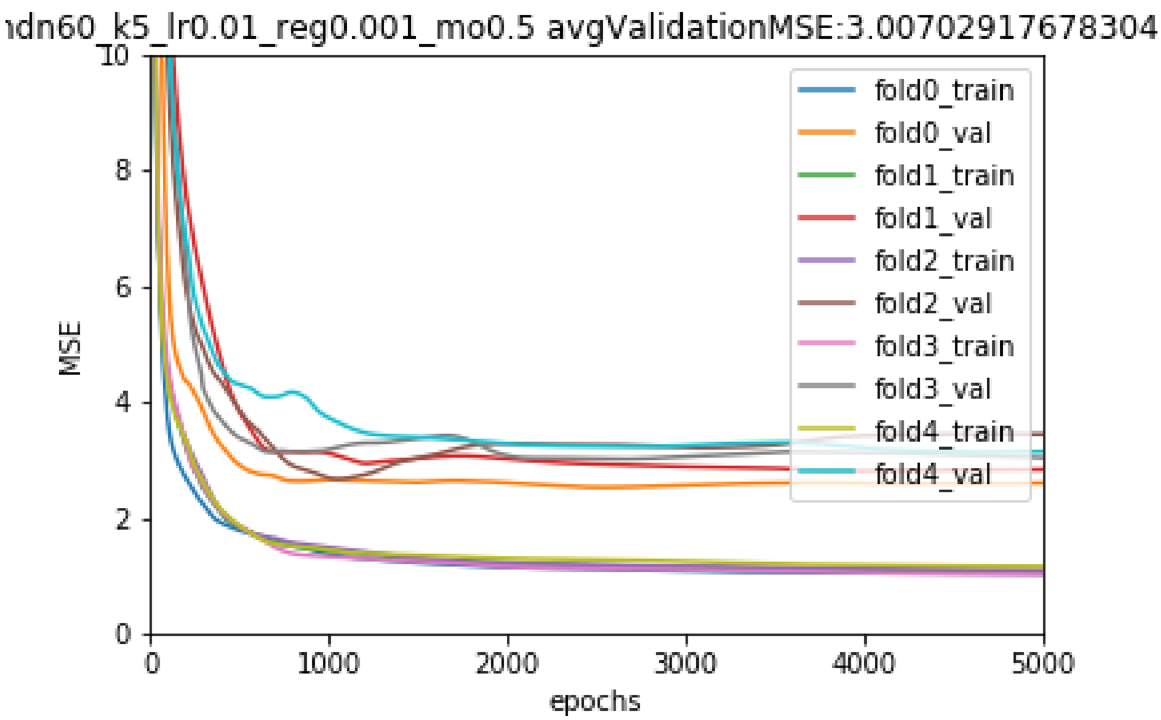

hdn80_k5_lr0.01_reg0.001_mo0.5 avgValidationMSE:2.9697727540618
MSE
epochs
fold0_train
fold0_val
fold1_train
fold1_val
fold2_train
fold2_val
fold3_train
fold3_val
fold4_train
fold4_val


hdn40_k5_lr0.01_reg0.001_mo0 avgValidationMSE:2.97611027249895C
MSE
epochs
fold0_train
fold0_val
fold1_train
fold1_val
fold2_train
fold2_val
fold3_train
fold3_val
fold4_train
fold4_val

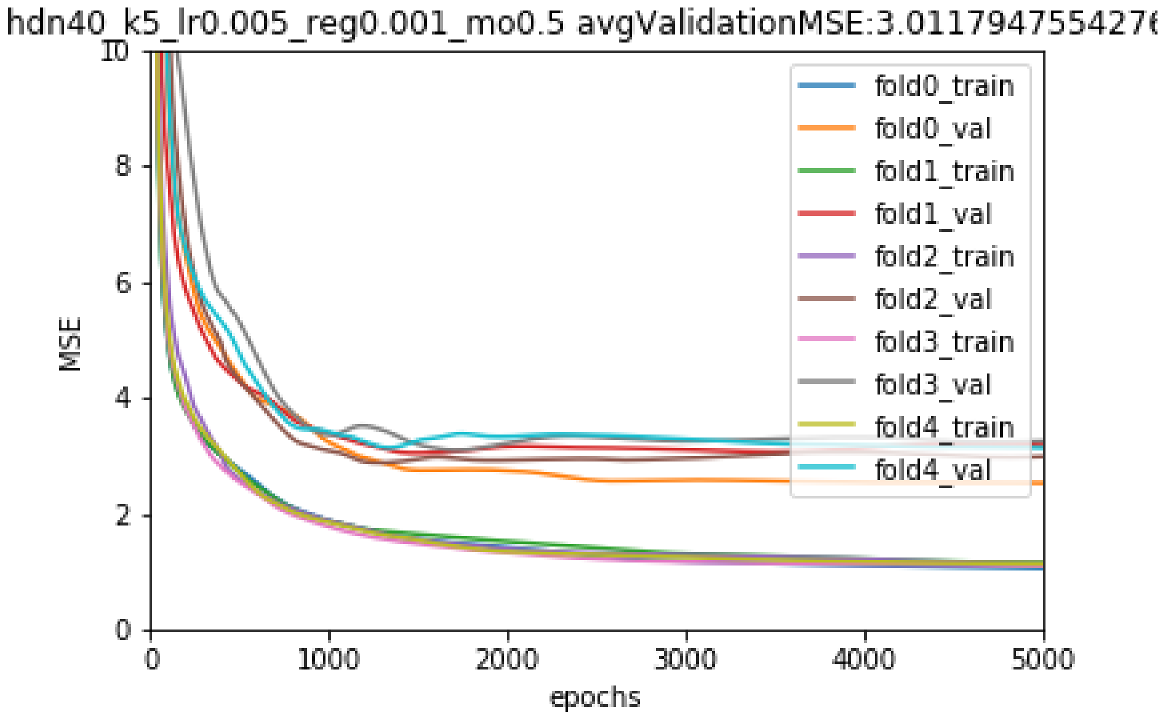

ıdn20_k5_lr0.01_reg0.001_mo0.5 avgValidationMSE:2.98225059671154
MSE
epochs
fold0_train
fold0_val
fold1_train
fold1_val
fold2_train
fold2_val
fold3_train
fold3_val
fold4_train
fold4_val

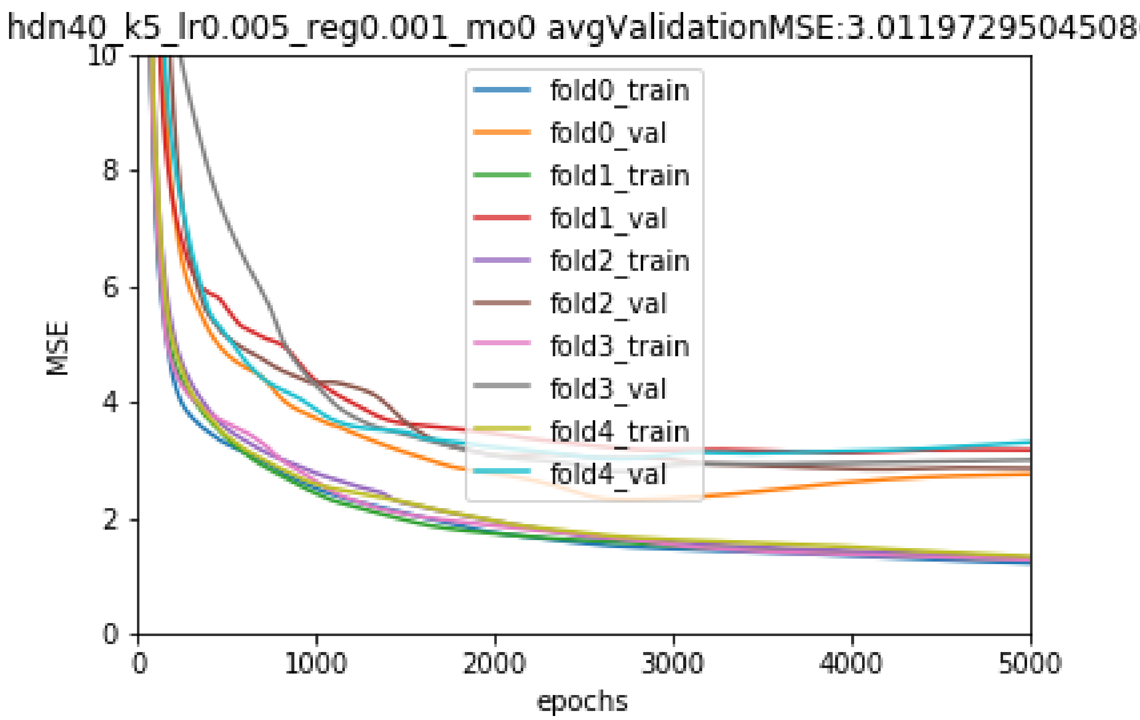


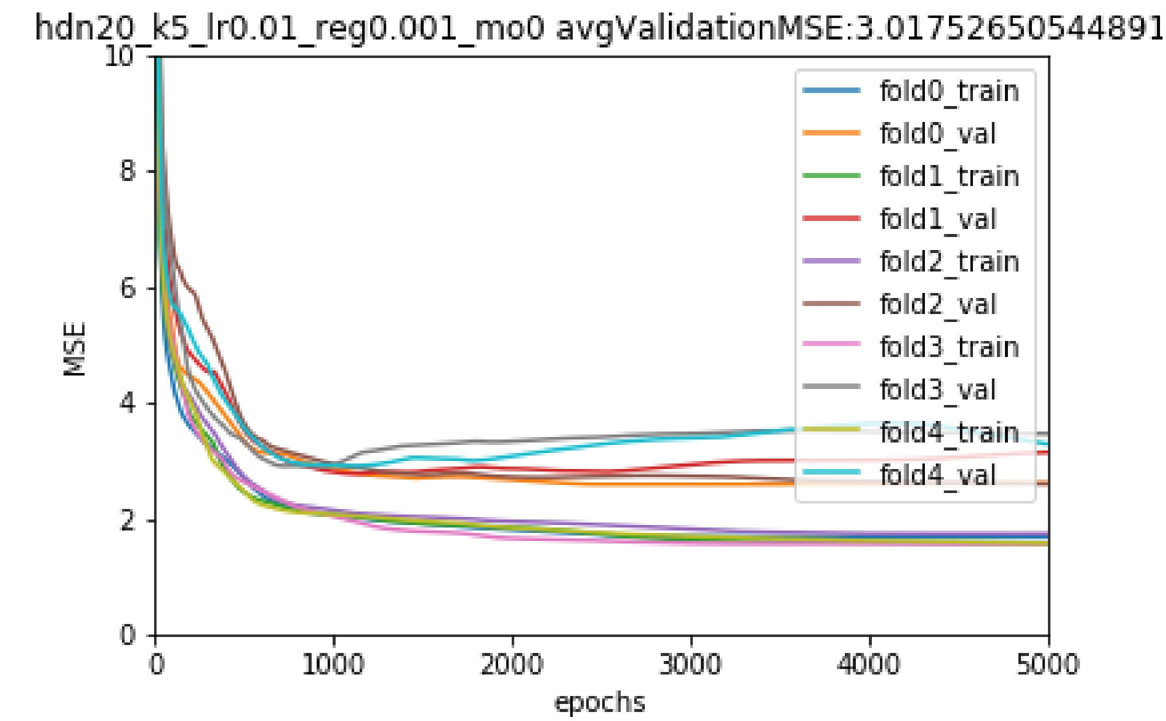
dn100_k5_lr0.005_reg0.001_mo0.5 avgValidationMSE:3.0068289310185
MSE
epochs
fold0_train
fold0_val
fold1_train
fold1_val
fold2_train
fold2_val
fold3_train
fold3_val
fold4_train
fold4_val

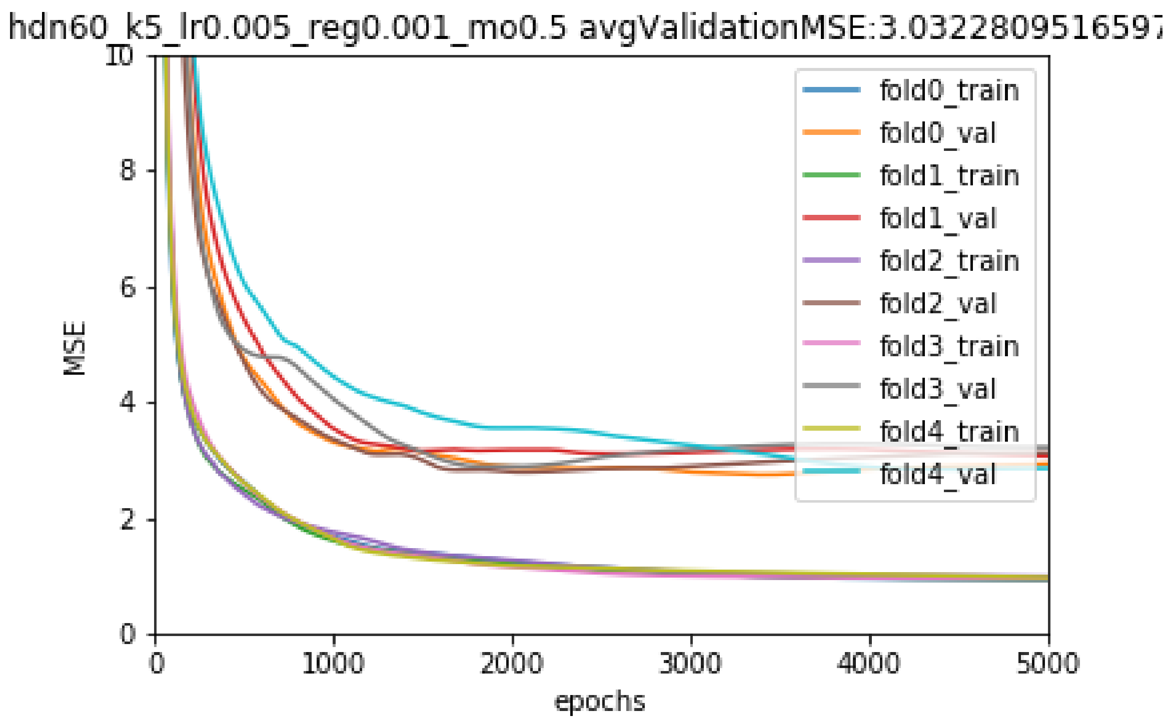


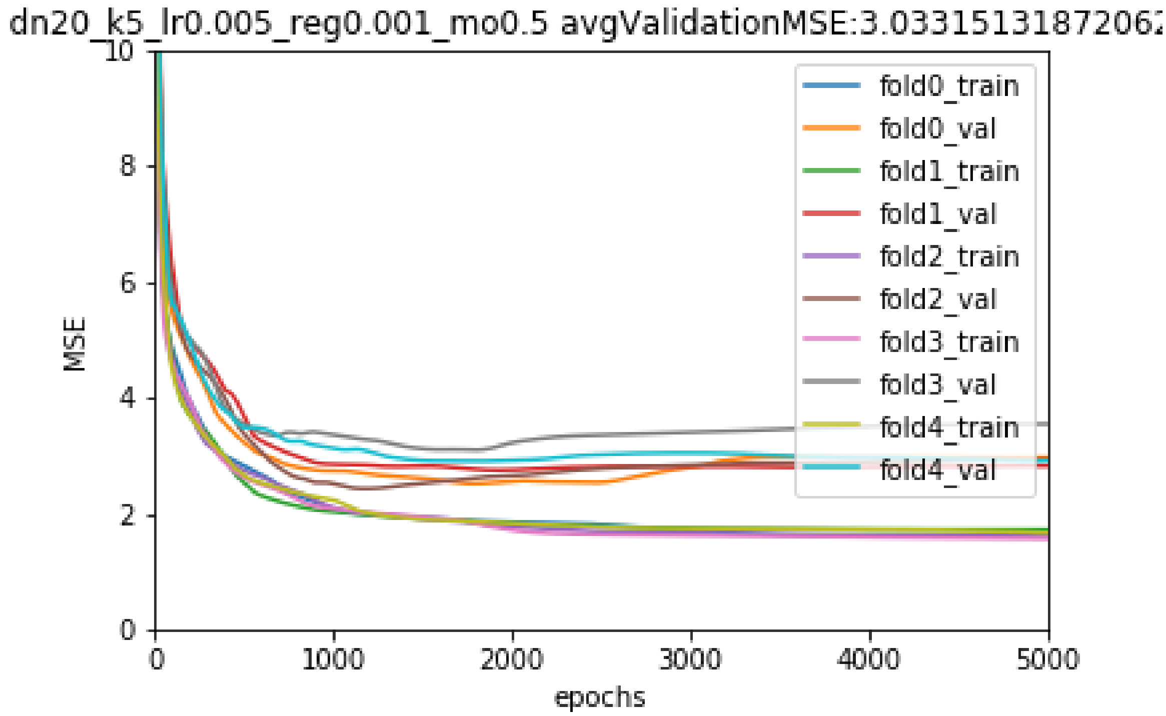
ndn60_k5_lr0.01_reg0.001_mo0.5 avgValidationMSE:3.00702917678304
MSE
epochs
fold0_train
fold0_val
fold1_train
fold1_val
fold2_train
fold2_val
fold3_train
fold3_val
fold4_train
fold4_val

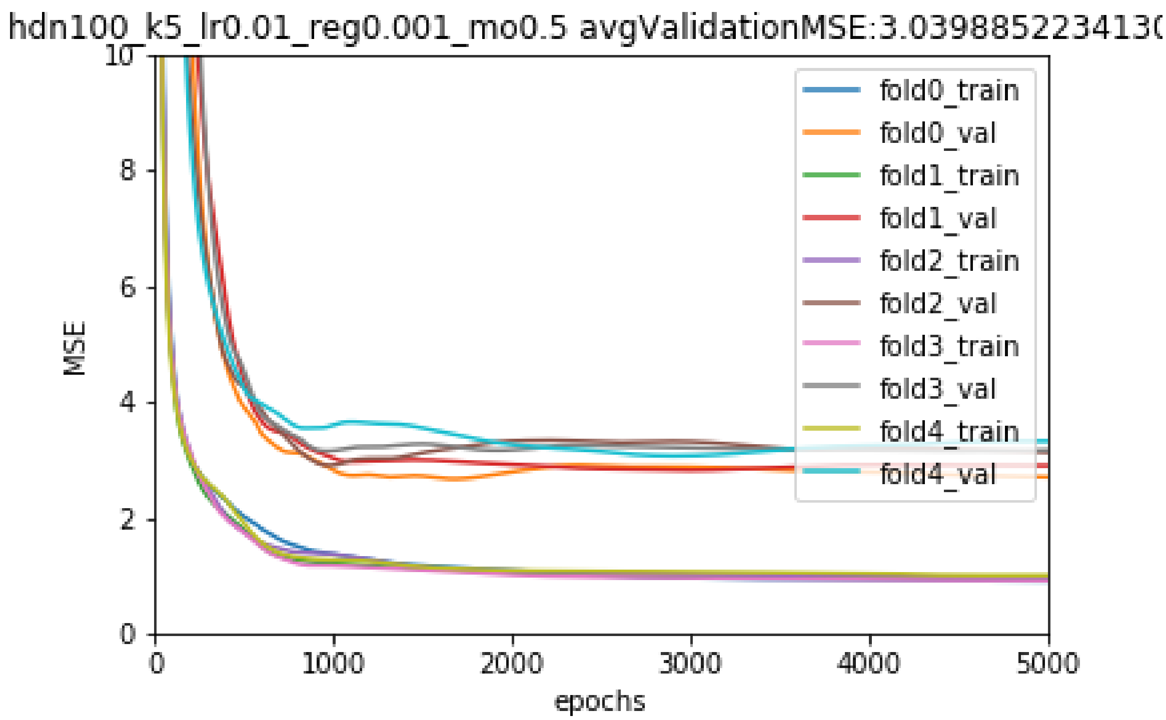

hdn40_k5_lr0.005_reg0.001_mo0.5 avgValidationMSE:3.0117947554276
MSE
epochs
fold0_train
fold0_val
fold1_train
fold1_val
fold2_train
fold2_val
fold3_train
fold3_val
fold4_train
fold4_val

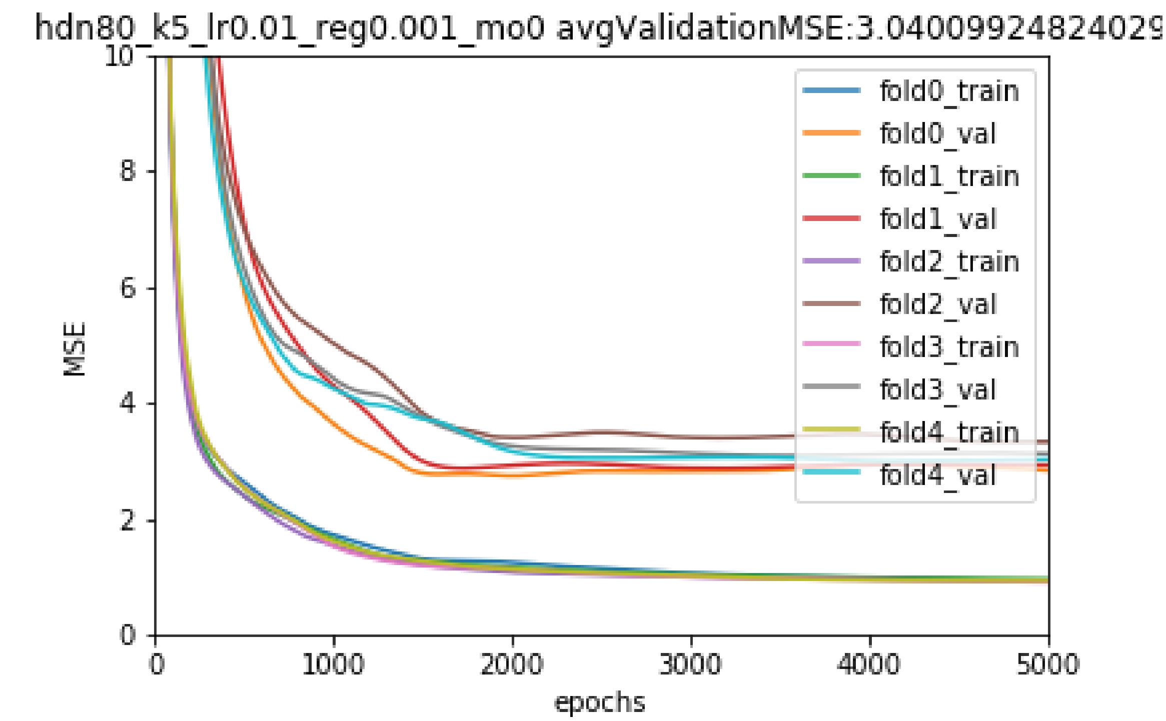

hdn40_k5_lr0.005_reg0.001_mo0 avgValidationMSE:3.0119729504508
MSE
epochs
fold0_train
fold0_val
fold1_train
fold1_val
fold2_train
fold2_val
fold3_train
fold3_val
fold4_train
fold4_val

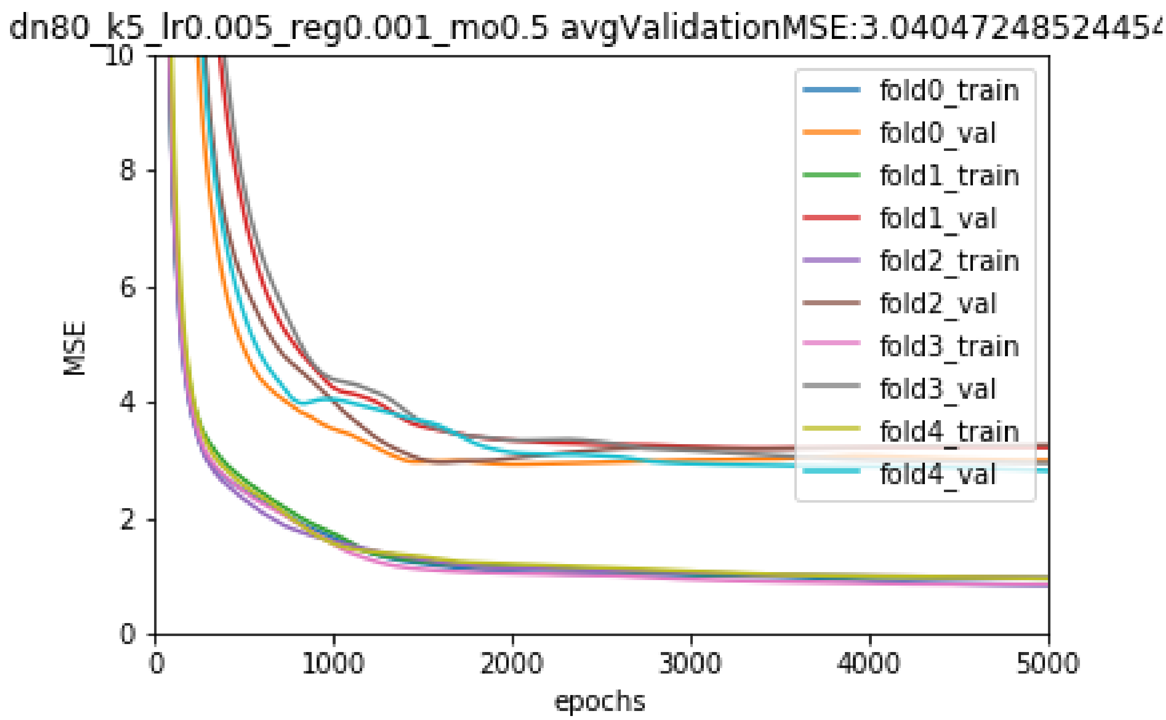

hdn20_k5_lr0.01_reg0.001_mo0 avgValidationMSE:3.01752650544891
MSE
epochs
0
2
4
6
8
10
0
1000
2000
3000
4000
5000
fold0_train
fold0_val
fold1_train
fold1_val
fold2_train
fold2_val
fold3_train
fold3_val
fold4_train
fold4_val

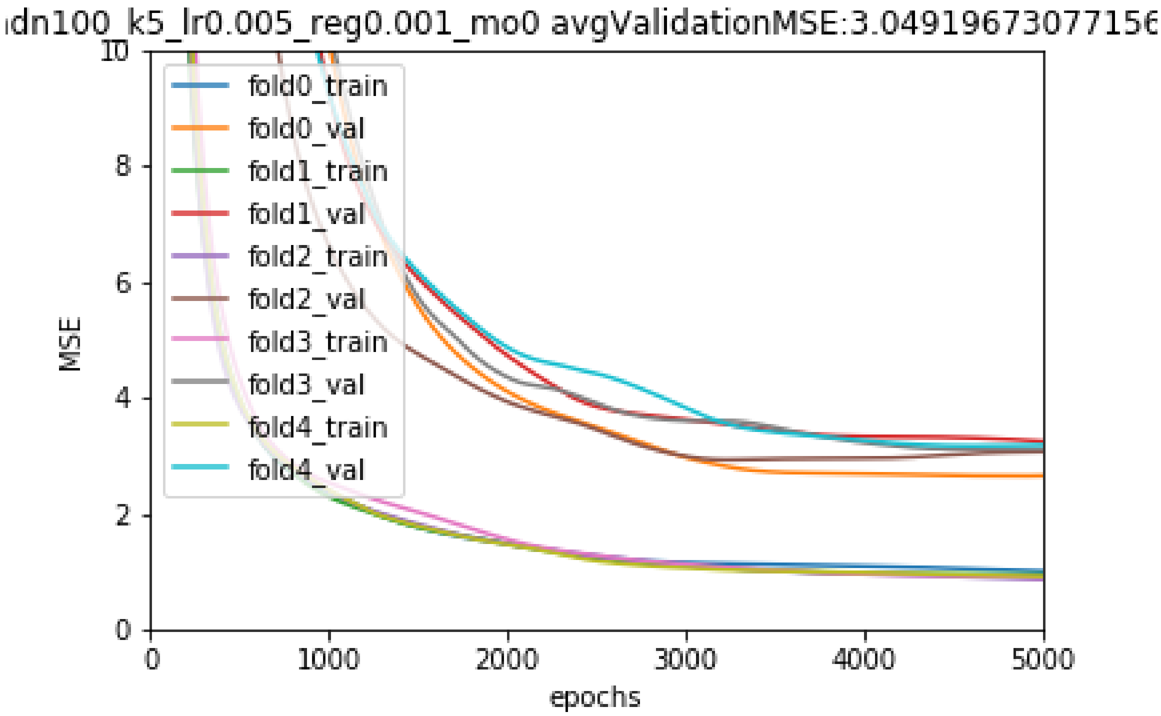

hdn60_k5_lr0.005_reg0.001_mo0.5 avgValidationMSE:3.0322809516597
MSE
epochs
0
2
4
6
8
10
0
1000
2000
3000
4000
5000
fold0_train
fold0_val
fold1_train
fold1_val
fold2_train
fold2_val
fold3_train
fold3_val
fold4_train
fold4_val

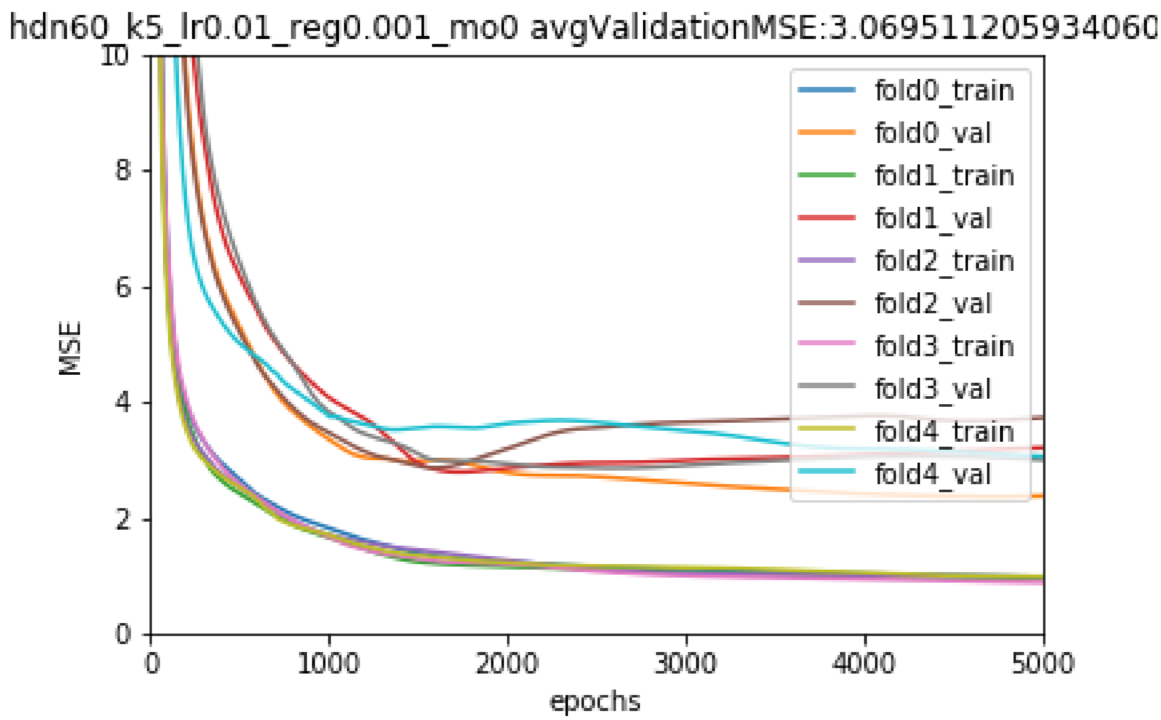

dn20_k5_lr0.005_reg0.001_mo0.5 avgValidationMSE:3.0331513187206
MSE
epochs
fold0_train
fold0_val
fold1_train
fold1_val
fold2_train
fold2_val
fold3_train
fold3_val
fold4_train
fold4_val

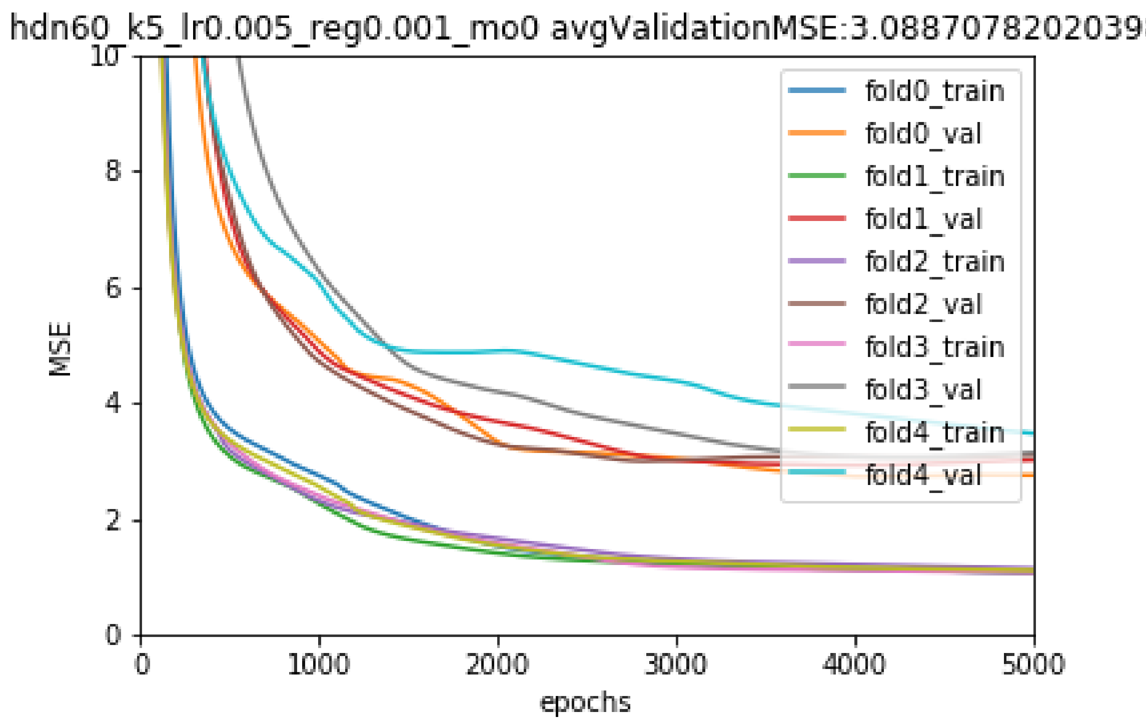

hdn100_k5_lr0.01_reg0.001_mo0.5 avgValidationMSE:3.039885223413
MSE
epochs
fold0_train
fold0_val
fold1_train
fold1_val
fold2_train
fold2_val
fold3_train
fold3_val
fold4_train
fold4_val

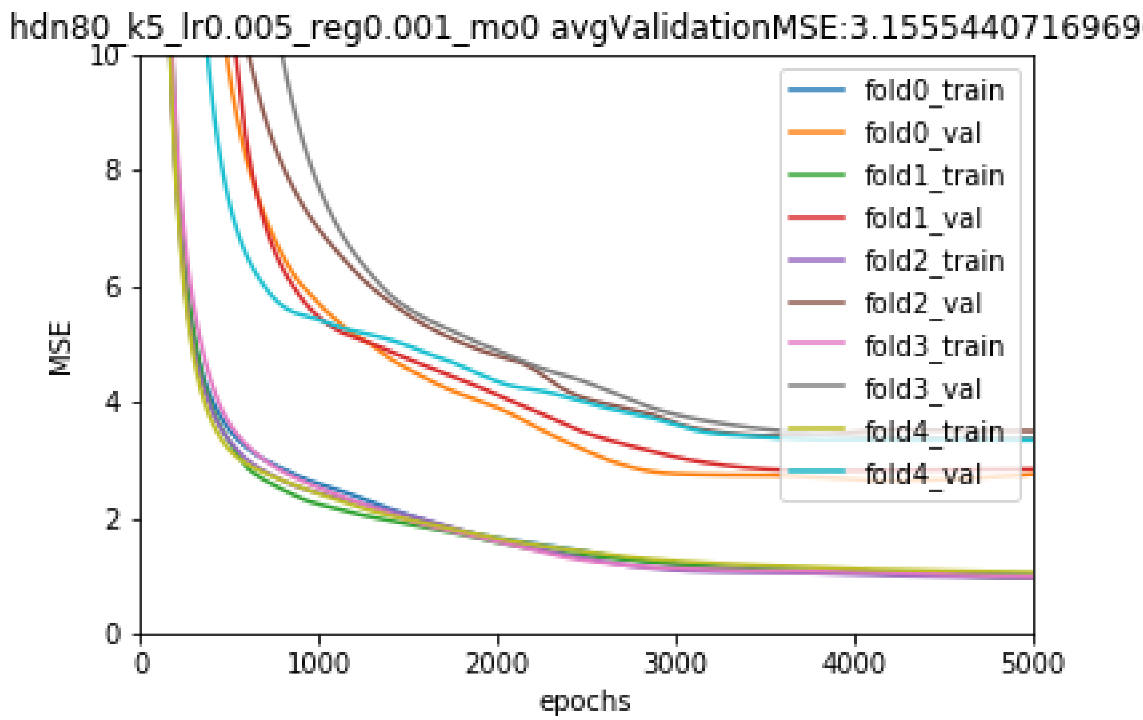

hdn80_k5_lr0.01_reg0.001_mo0 avgValidationMSE:3.04009924824029
MSE
epochs
0
2
4
6
8
10
0
1000
2000
3000
4000
5000
fold0_train
fold0_val
fold1_train
fold1_val
fold2_train
fold2_val
fold3_train
fold3_val
fold4_train
fold4_val

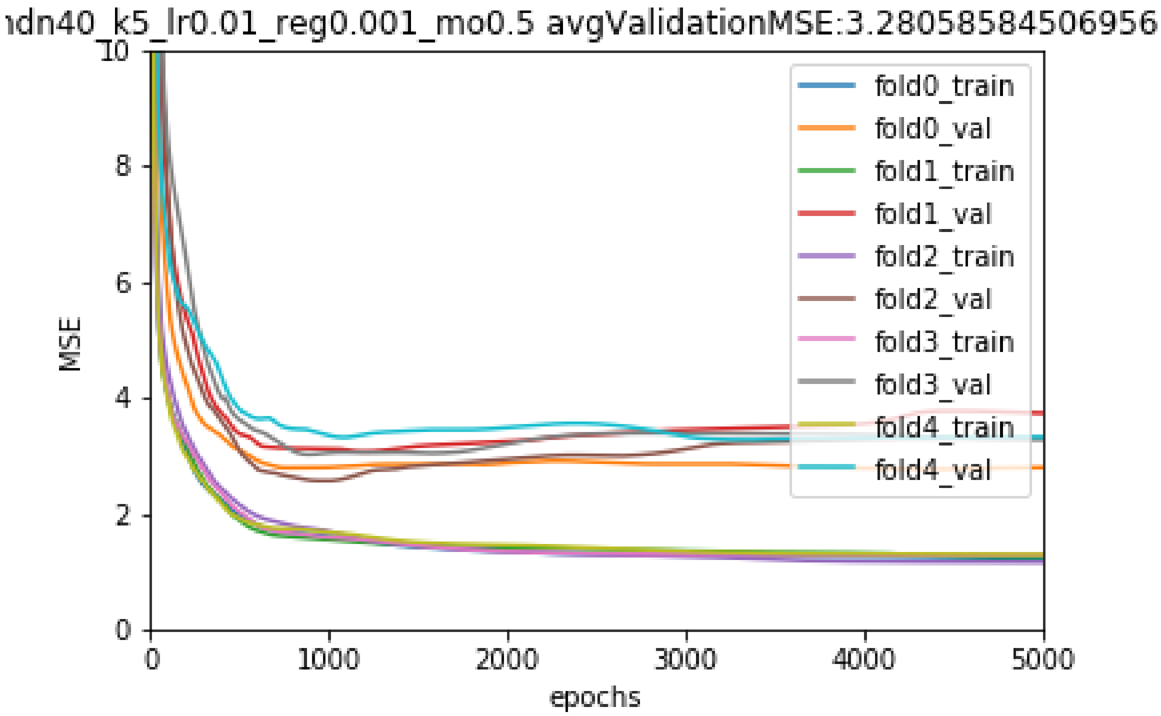

dn80_k5_lr0.005_reg0.001_mo0.5 avgValidationMSE:3.04047248524454
MSE
epochs
0
2
4
6
8
10
0
1000
2000
3000
4000
5000
fold0_train
fold0_val
fold1_train
fold1_val
fold2_train
fold2_val
fold3_train
fold3_val
fold4_train
fold4_val

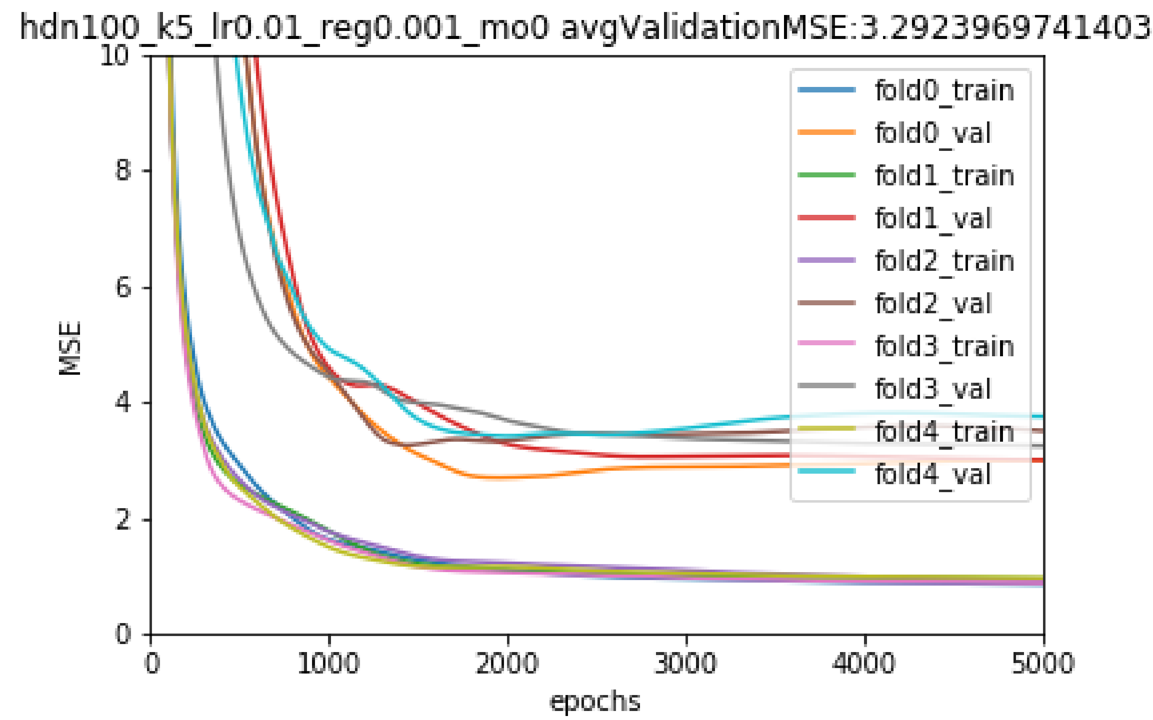

hdn100_k5_lr0.005_reg0.001_mo0 avgValidationMSE:3.0491967307715
MSE
epochs
fold0_train
fold0_val
fold1_train
fold1_val
fold2_train
fold2_val
fold3_train
fold3_val
fold4_train
fold4_val


hdn60_k5_lr0.01_reg0.001_mo0 avgValidationMSE:3.069511205934060
MSE
epochs
fold0_train
fold0_val
fold1_train
fold1_val
fold2_train
fold2_val
fold3_train
fold3_val
fold4_train
fold4_val


hdn60_k5_lr0.005_reg0.001_mo0 avgValidationMSE:3.088707820203
MSE
epochs
0
2
4
6
8
10
0
1000
2000
3000
4000
5000
fold0_train
fold0_val
fold1_train
fold1_val
fold2_train
fold2_val
fold3_train
fold3_val
fold4_train
fold4_val


hdn80_k5_lr0.005_reg0.001_mo0 avgValidationMSE:3.15554407169969
MSE
epochs
0
2
4
6
8
10
0
1000
2000
3000
4000
5000
fold0_train
fold0_val
fold1_train
fold1_val
fold2_train
fold2_val
fold3_train
fold3_val
fold4_train
fold4_val





We pick hdn80_k5_lr0.01_reg0.001_mo0.5 based on the same previous criteria to train on the train 60% cross validation 40% splits

c. Final Models

Here we report the Final Models selected from the previous k-fold validation strategy to be trained on the 60% training 40% validation splits for a final cross validation to choose the CUP challenge model

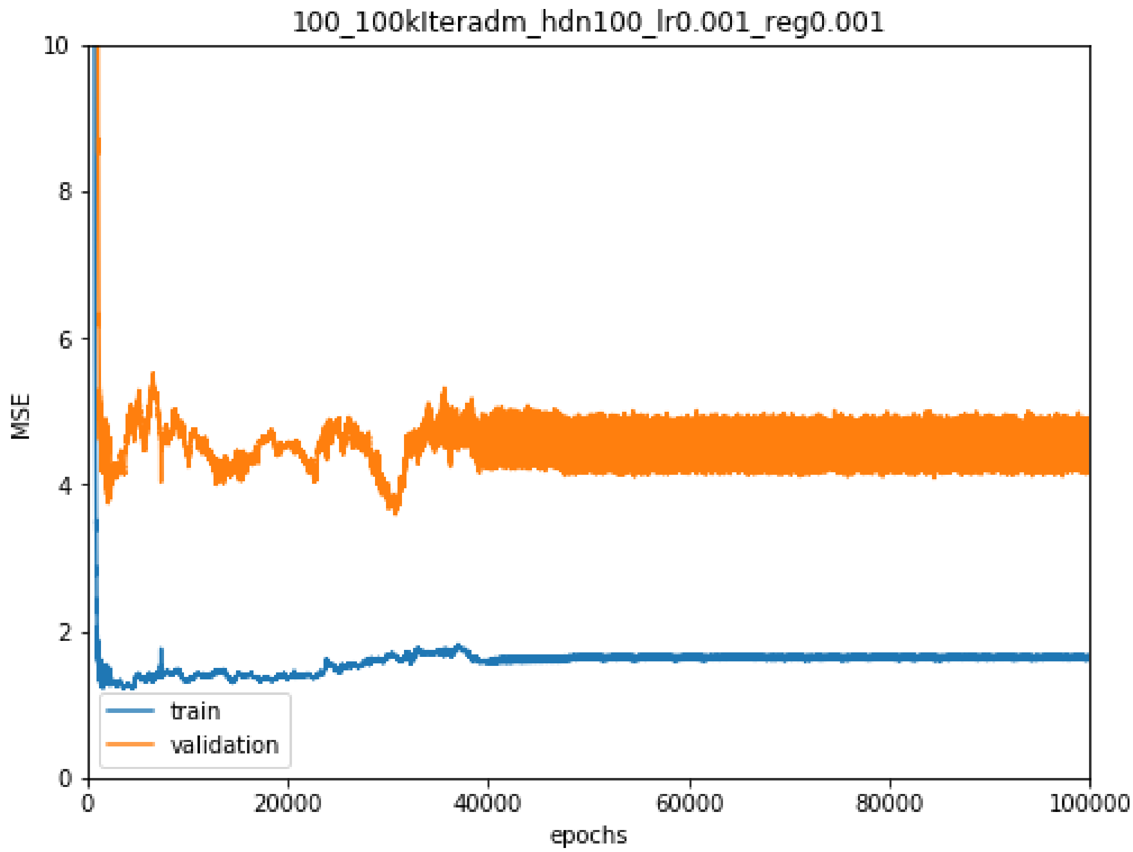


Figure showing the learning curve of the model trained with adam
hdn100_lr0.001_reg0.001
Validation MEE:1.49854175400599 MSE:3.89719414024494

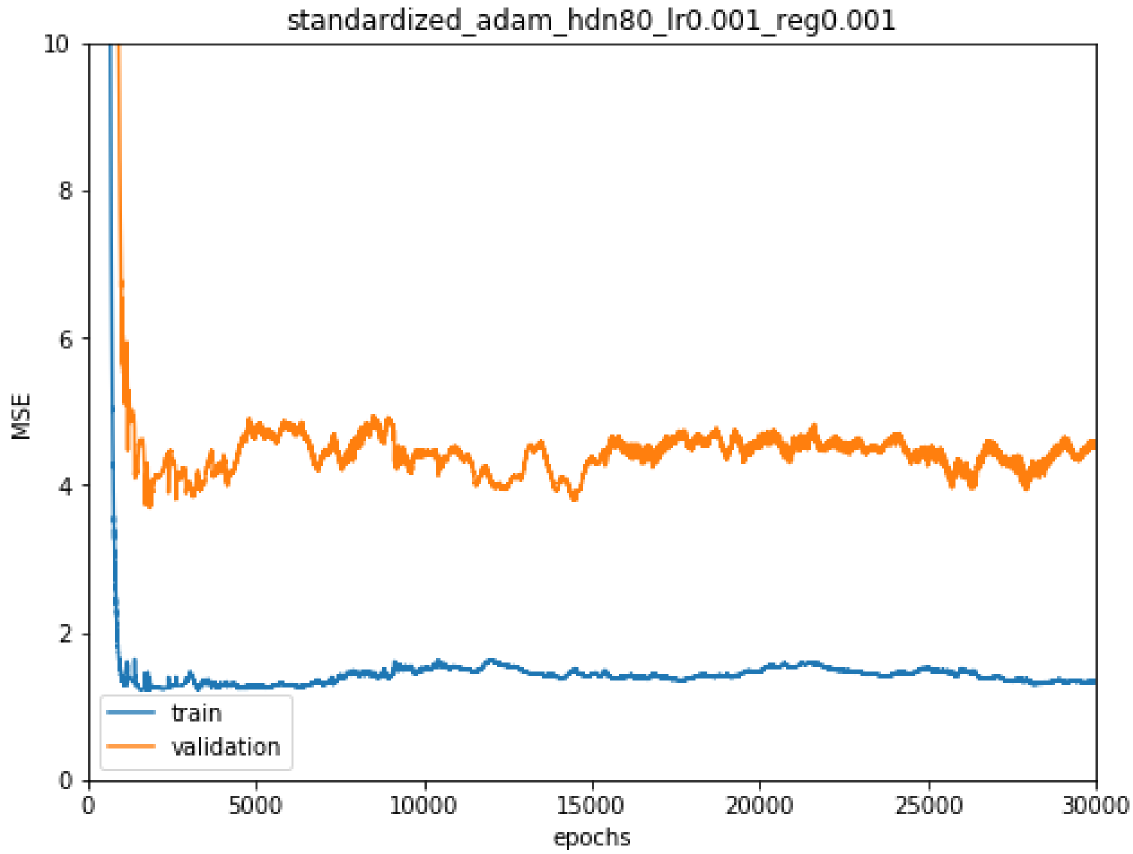


Figure showing the learning curve of the model trained with adam hdn80_lr0.001_reg0.001
Validation MEE:1.669036223757774 MSE:4.5103993425969

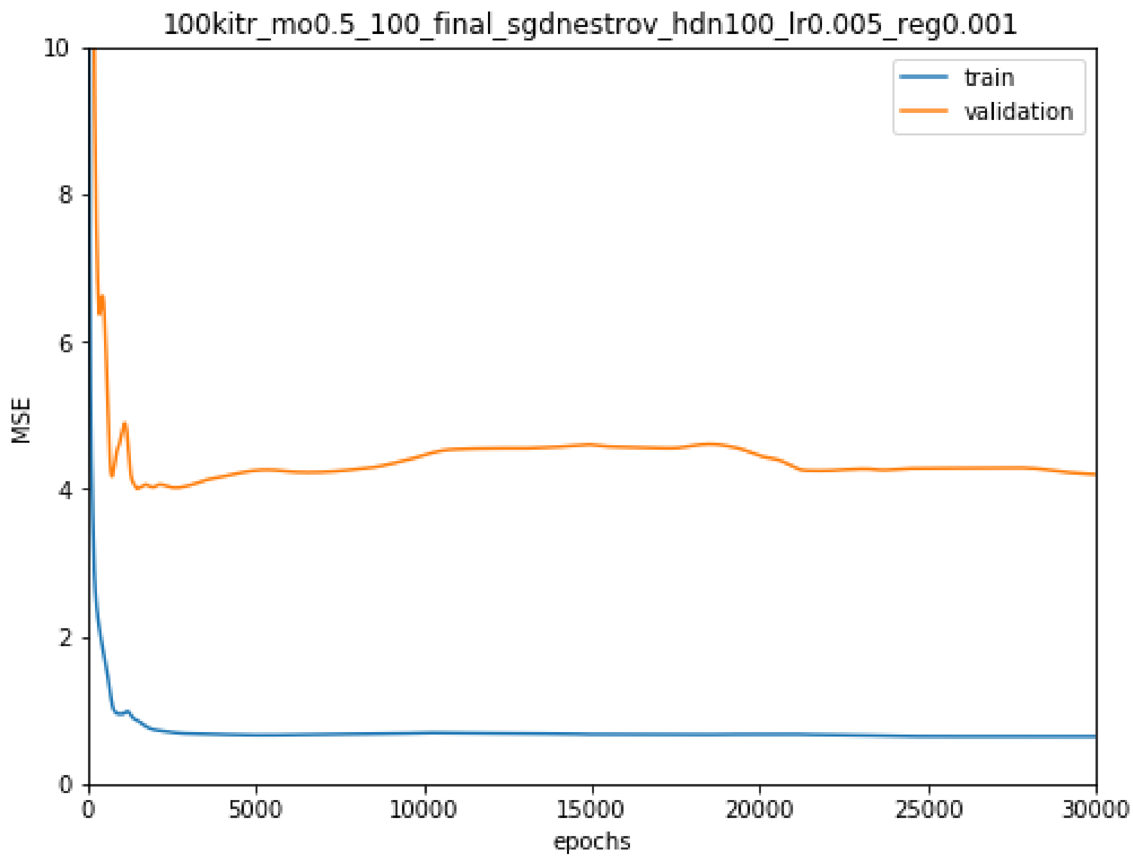


Figure showing the learning curve of the model trained with SGD+Nestrov
hdn100_lr0.005_reg0.001_mo0.5
Validation MEE:1.6140627868185 MSE:4.3567326789068

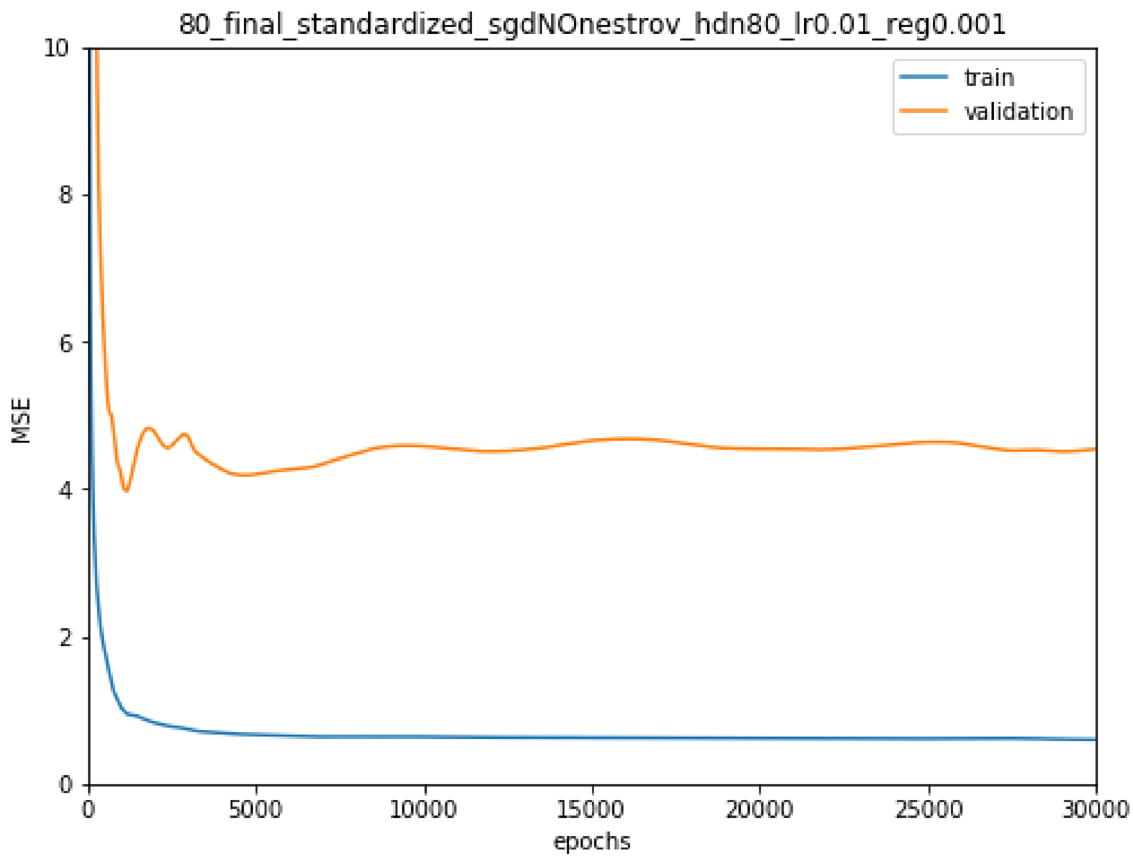
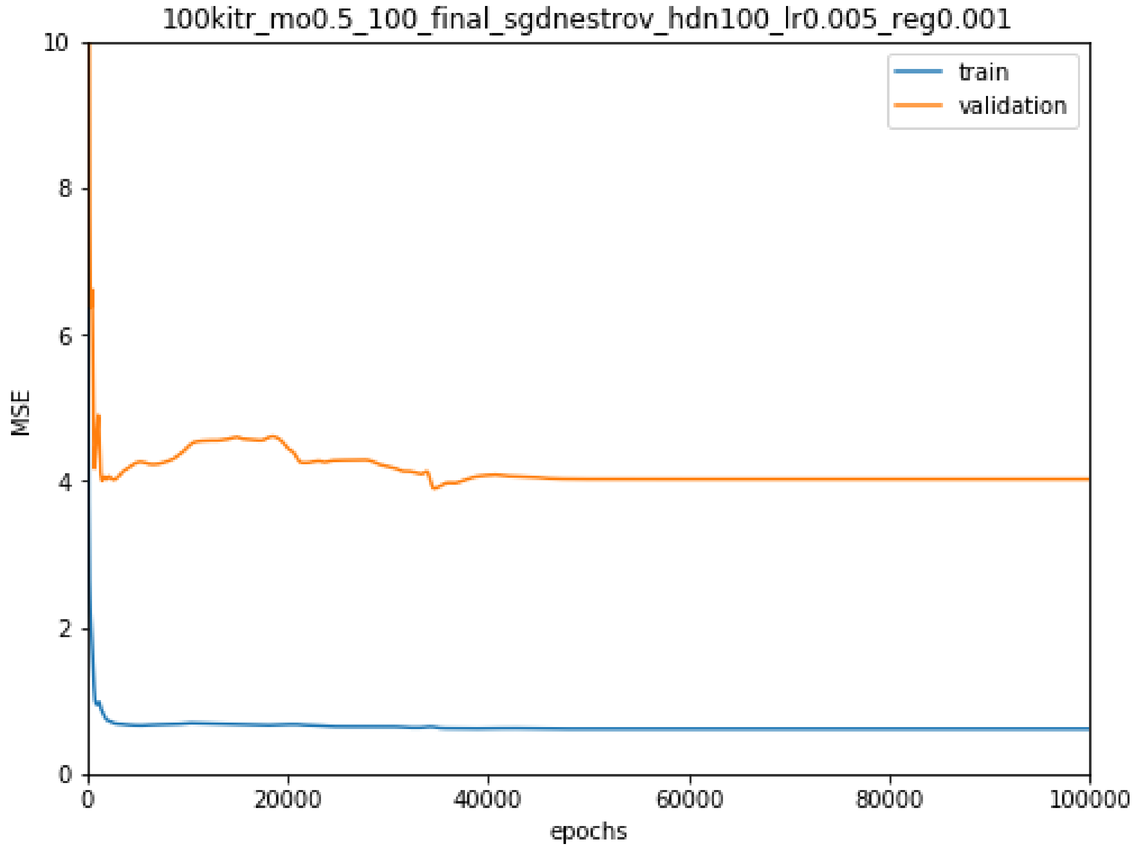


Figure showing the learning curve of the model trained with sgd+Classic
hdn80_lr0.01_reg0.001_mo0.5
Validation MEE:1.63588659234221 MSE:4.53702431588675

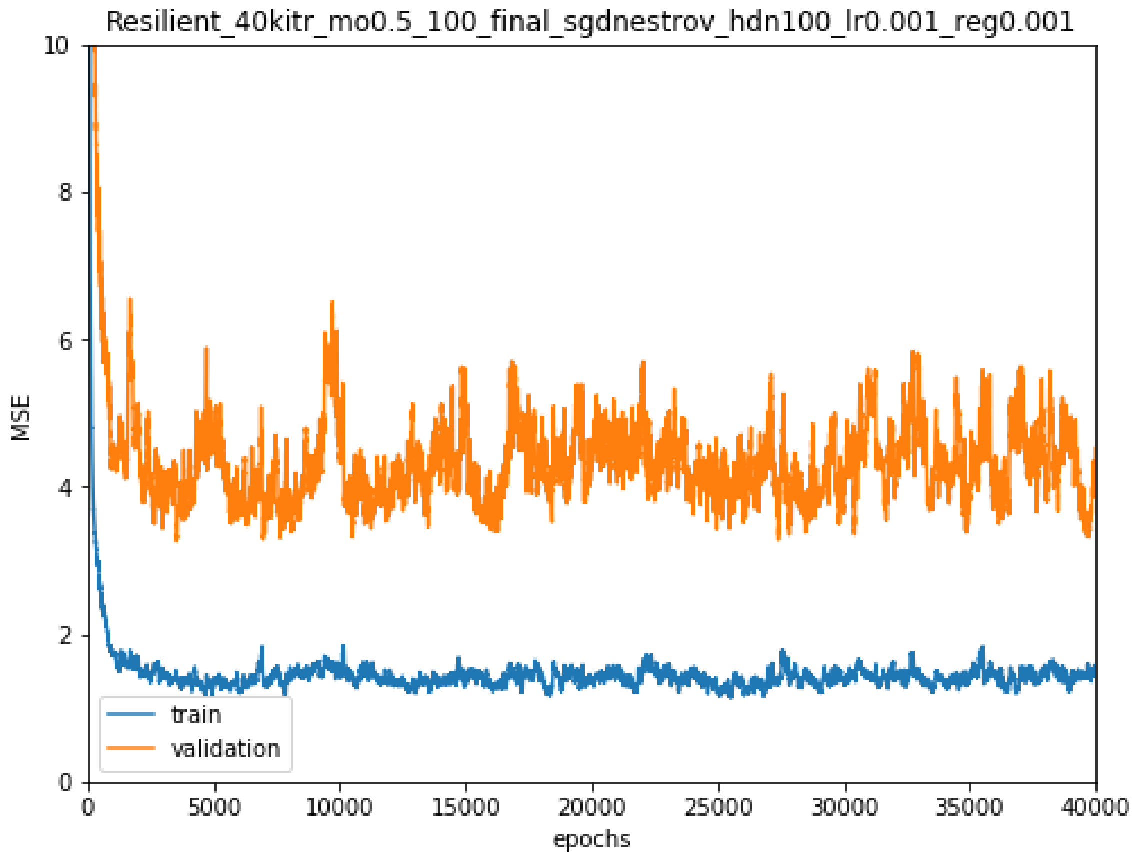
So, we think the model trained with SGD+Nestrov hdn100_lr0.005_reg0.001_mo0.5 has the highest potential, we retrained it for 100k epochs on the same dataset



And it seems to have reached a stable result with validation
Validation MEE:1.547052945611185   MSE:4.023131973336354

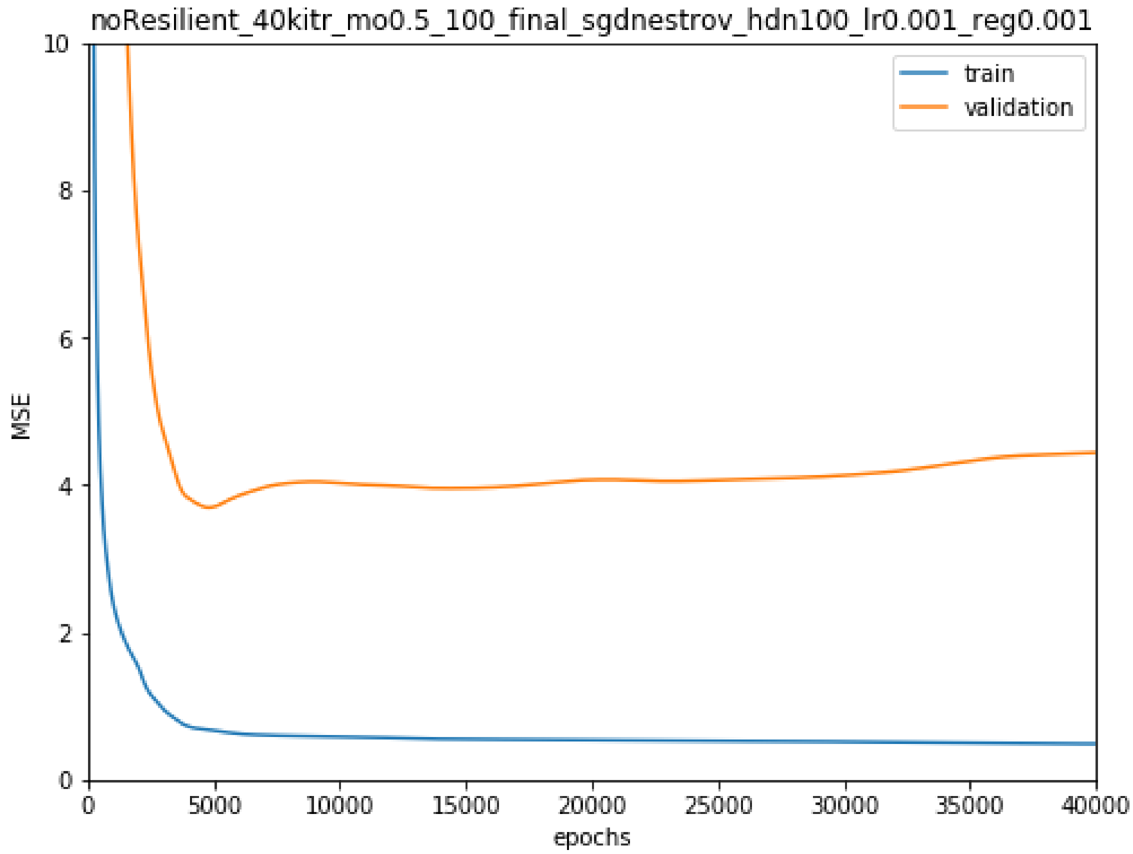
It gets stable after ~35k epochs maintaining the lowest validation MEE and MSE loss reported so far, so, we decided to use it for the final challenge.

A feature we tried to introduce to speedup the learning process, resilient weights updates, simply use a learning rate of learningRate*2 if the sign of the previous gradient is the same as the current one and learningRate/2 if the sign is different, mimicking the idea of speeding up when we get a similar direction as previous and slowing down if the the direction changes, however, it wasn't as successful as advertised, unfortunately, thus, not used for the final training.



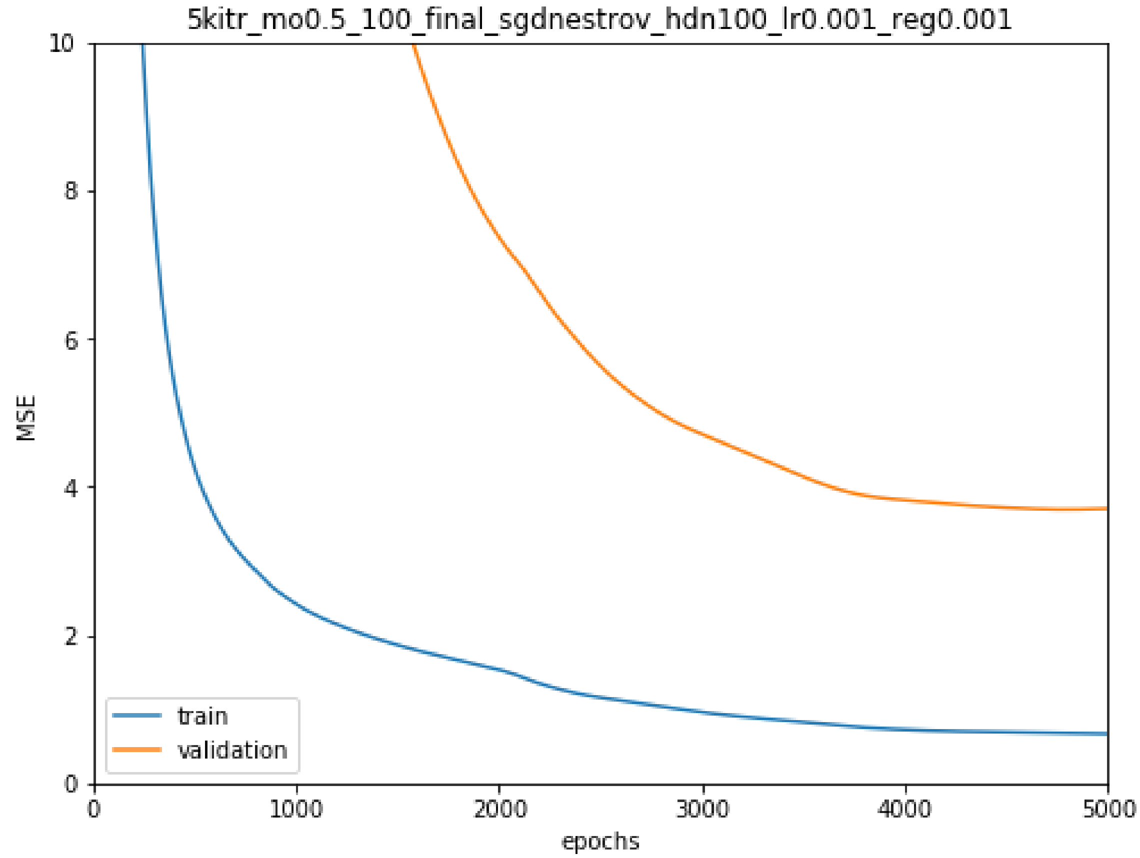
Same model with resilient, unfortunately, it is unstable
MEE:1.63194315584053MSE:4.51586183031232

We also experimented a little with the data preprocessing, the following is the learning curve with data standardization instead of normalization:



MEE:1.56983790054965 MSE:4.435713232622243

The curve is even smoother with the best MSE around 5k epochs, so, for the final challenge we are only training for 5k epochs



Quite nice and smooth, also fast!
MEE:1.4626515238698 MSE:3.70174966319864

d. Comparison Against an off-the-shelf Library

One last comparison is introduced with off-the-shelf solutions using the same 60% 40% train/validation splits:

For the same model with the same stochastic gradient descent trainer

Learning rate = 0.001
Momentum[Nestrov] rate =0.5
L2 Regularization rate=0.001
number of hidden neurons = 100
Number of Epochs = 5000
Batch size =10

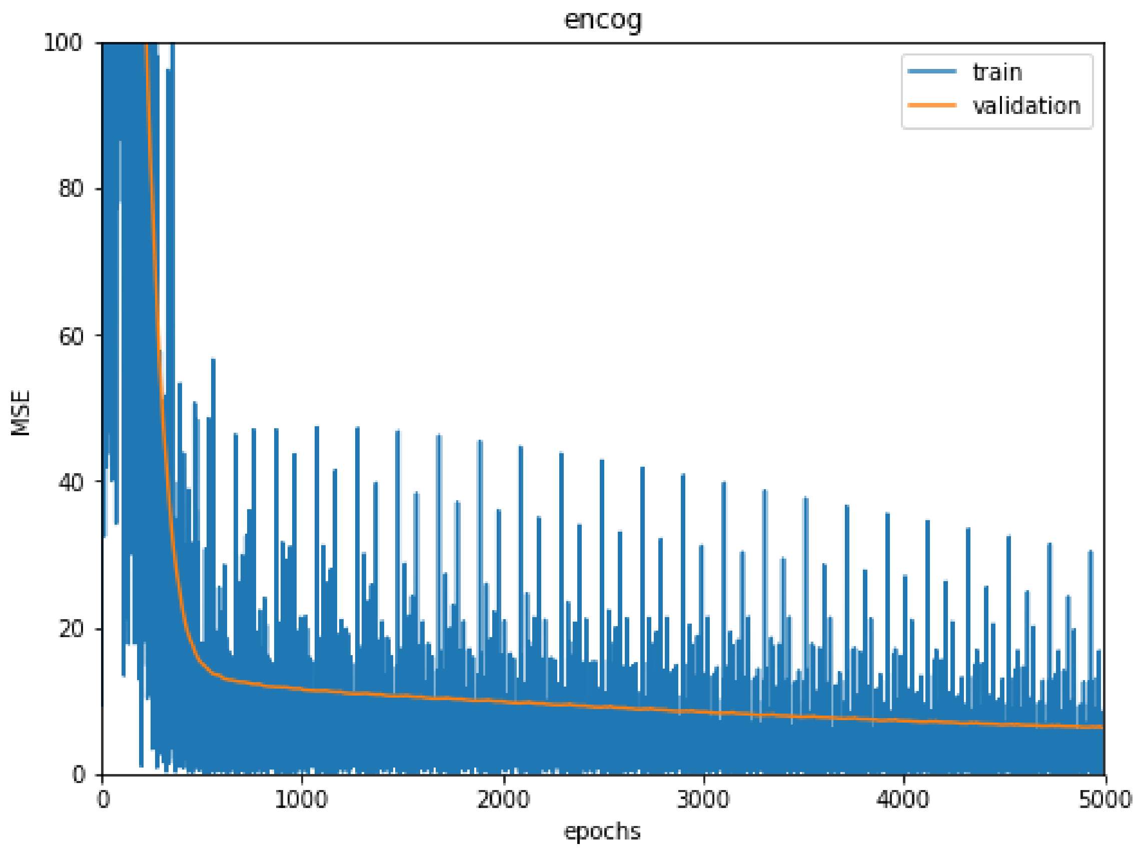


Figure Showing Encog learning curve on the same dataset, unfortunately, it is very unstable with the result Validation MEE2.15300193591599, MSE6.32110423932233

	Time	Validation MEE
Encog's	27386 ms	2.15300193591599
Ours	1000192 ms	1.4626515238698

Possible main reason for this difference in the MEE results aside from the implementation is the weights initialization and the random seed used.

In terms of speed, definitely their code is better optimized, in this project we didn't focus on code speed.

To reproduce similar results, please run AA1_Encog.program.cs with the correct paths to the standardized 60% training and 40% test datasets

2) LINEAR LEAST SQUARES

The second task was to provide a basic version of the linear least squares solver of our choice.

We provide the following three solvers:

a. Normal equations

As seen in the lecture, the simplest solver with the cheapest cost despite its instability, we introduced a simple implementation for a normal equations solver to our problems

b. SVD

A more robust, but also costly solver

c. Plain Gradient Descent

The plain gradient descent algorithm just updating the weights of the linear model with a fixed step as a learning rate against the gradient of the loss of the model without a line search.

Following we provide a comparison of the results acquired by the three solvers on both of the introduced problems

The Following is a performance comparison between the three solutions for the CUP problem:

	Time	Validation MEE
SVD solution	72 ms	18.2217739575164
Normal Equations Solution	30 ms	18.2217739575164
Gradient Descent Solution	862 ms for 1k iterations with a degree of 1 and learning rate of 0.1	2.21447507466273

Table 7 – The performance comparison between the three solutions

To reproduce the results, please run `CupTestingLLS` in MLPTestDemo.Program.cs with the correct path to the train and test splits.

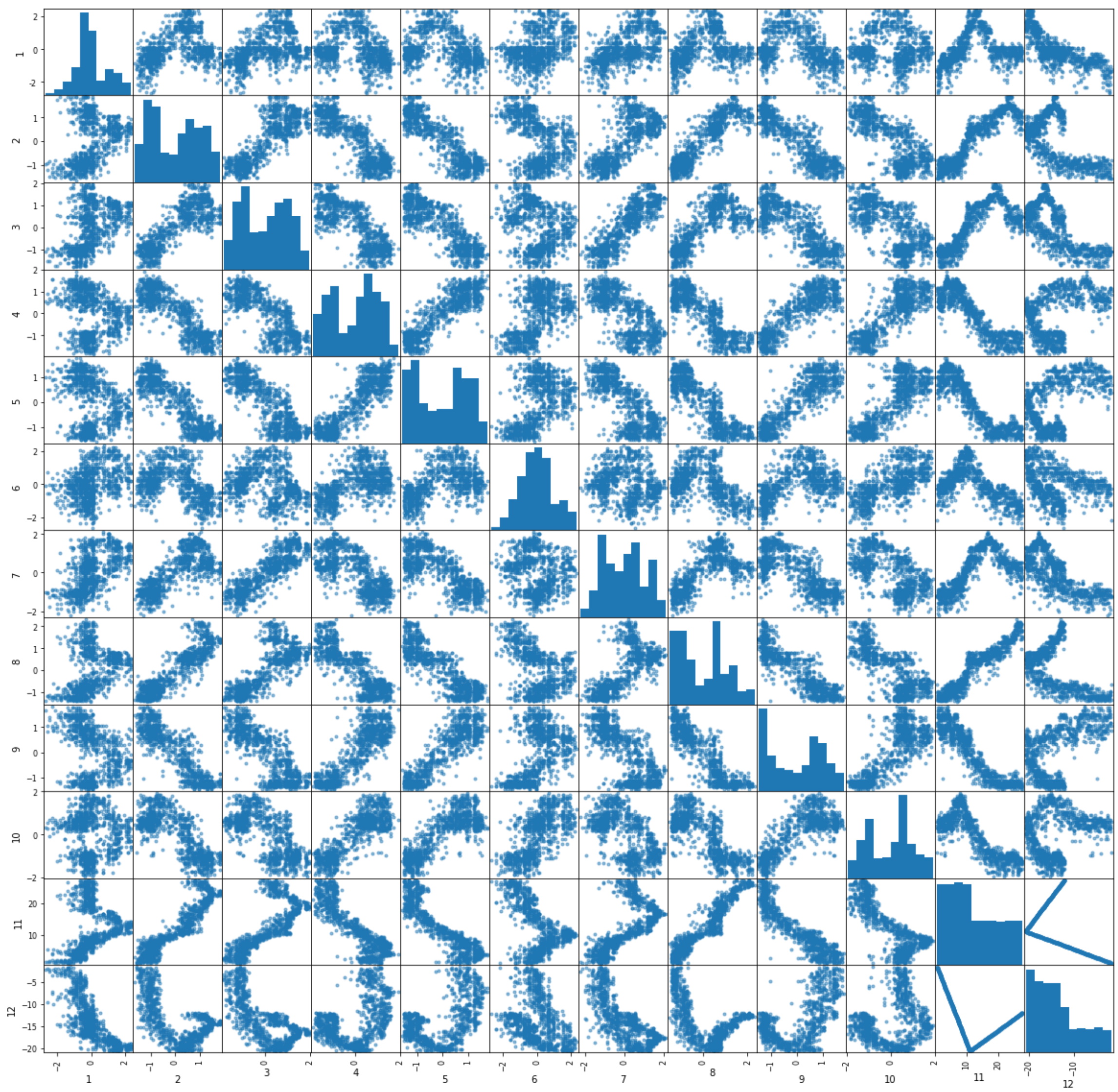
To obtain the data splits, please run the python notebook "SikitLearnLLS.ipynb" up to cell 7 inclusive.

From these experiments provided, we believe the iterative solution seen in the Optimization lessons is a clear winner, it is able to find a much better solution than all of the others, despite its drawback of the computational cost

3. On the capacity& efficiency comparison of the solutions

To answer these points, we first conducted a few steps to help understand the nature of the provided data in the notebook DataUnderstanding.ipynb

Perhaps the most important step is the following figure:



Showing the correlations between the 10 inputs among themselves and also the two outputs.
Any model we use shall capture the correlation between the inputs and the outputs which

is mostly non-linear, making any non-linear model in a complete disadvantage!

Since, unlike the linear model, MLPs can capture such relations, perhaps it is already a clear winner and we have already seen that in the report results in the previous sections of the report.

	Time	Validation MEE
LLS: SVD solution	72 ms	18.2217739575164
LLS: Normal Equations Solution	30 ms	18.2217739575164
LLS: Gradient Descent Solution	4469 ms for 5k iterations with a degree of 1 and learning rate of 0.1	2.21447507466273
MLP: SGD+Nestrov Momentum	1000192 ms 100 hidden neurons, .001 learning rate and regularization rate, momentum 0.5 5k iterations	1.4626515238698

Table showing The performance comparison between the LLS and MLP solutions

From this we conclude that in terms of the model capacity, ability to find a good solution and generalization, MLPs are a clear winner, but in terms of computational cost, LLS will be favored

4. LLS implemented vs Off-the-shelf

Unfortunately, LLS is not available in C# off-the-shelf, however, we provide here a comparison between our implementation and that of scikit-learn.org

	Time	**Validation MEE**
Our LLS+SVD	73 ms	18.2217739575164
Scikit-learn LLS+SVD	231 ms	18.221791608547466
Our LLS+GradientDescnet	862 ms for 1k iterations with a degree of 1 and learning rate of 0.1	2.21447507466273
Scikit-learn LLS+GradientDescnet [solver='sag']	19 ms for 1k iterations with a degree of 1 and learning rate of 0.1	18.221775282262154

Table 8 – The performance comparison between ours and that of scikit-learn.org

For reproducing the results:

1- Run the python notebook "SikitLearnLLS_MEE_SavingTrainDataStandardized.ipynb"
We split the training set of the CUP challenge into two subsets, 60% training and 40% testing, save the data splits to be used by C# as well
And the book forms the problem and solves it with SVD and Gradient Descnet
2- Call the function `CupTestingLLS` in MLPTestDemo.Program.cs with the correct path to the train and test splits.

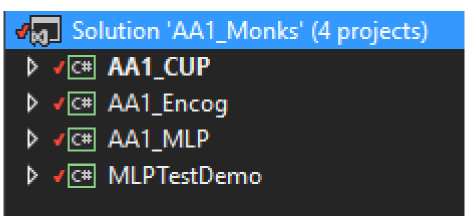
5. Implementation

In this section, we are glad to introduce our implementation of the work done in the form of a lightweight library with several features and possibility to extend.
The whole work is available on the following Git repository:
https://github.com/lilanpei/Monks

1) FEATURES

1. Possibility to build, train and test Multilayer perceptrons with different settings;
2. Number of layers, number of neurons per layer
3. Different activation functions: Sigmoid, Tanh, Relu
4. Different weight initialization mechanisms: Uniform, Xavier, He
5. MLP Trainers: Stochastic Gradient Descent + Momentum [classic and nesterov],Adam Optimizer, L2 regularization
6. Possibility to build and train a linear regression model
7. Data managers and interfaces to help handle regression and classification datasets
8. A solution to help screening the experiments and decide the final hyper parameters to choose the best model for the problem being solved
9. And most importantly, the ability to extend and add to the library more solvers, activations, or even different machine learning algorithms later by implementing our interfaces

2) SOLUTION PROJECTS



The projects implemented in the solution

We have mainly four project as follows:

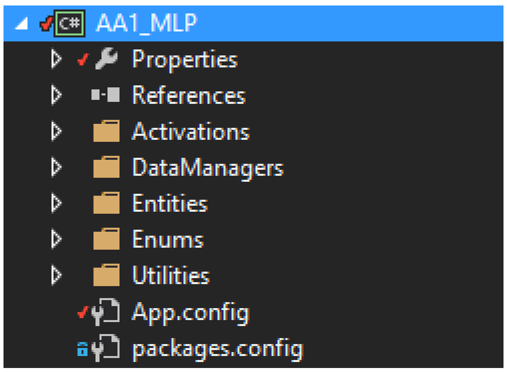
1. AA1_CUP is a reusable project for screening a model
2. AA1_Encog is a project used for comparing our selected model against an off-the-shelf library, namely; Encog by heatonresearch.com
3. AA1_MLP is a project for building, training and testing MLPs and Linear regression models

4. MLPTestDemo is a project for testing the features of the models, trainers and the introduced settings in AA1_MLP, we used extensively this project with the Monk's problem datasets to build the features of AA1_MLP

Our sincere apologies for the bad namings [developers issues ☺]

The following are more details about each project and its usage:

AA1_MLP

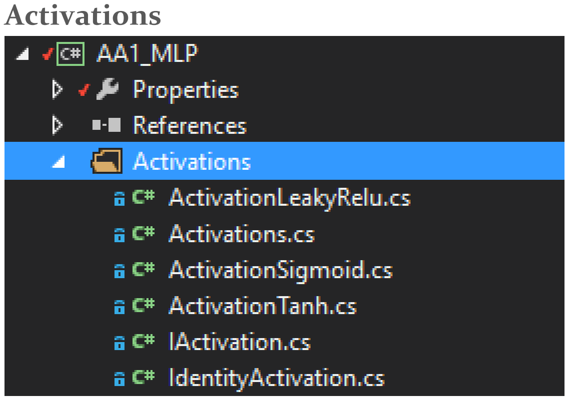


The main dish on our menu here☺

Activations is a folder providing activation function of MLPs
DataMangers is for providing managers to load training and testing datasets
Entities provides the models implementations along with their solvers/optimizers
Enums aggregates our used enums
Utilities holds our own extension methods for C# along with utilities and tools to facilitate the development process and the usage of the code

The following are more details on each folder of the AA1_MLP project

Activations



IActivation.cs provides and interface that must be implemented by any activation function desired with two functions:
CalculateActivation:

```
        Vector<double> CalculateActivation(Vector<double> x);
```

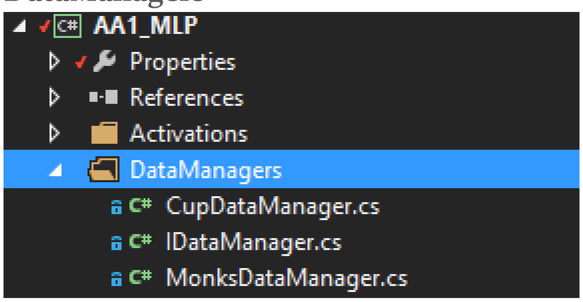
Given a vector of doubles x, applies the activation in a pointwise fashion and returns the result
And Calculate Derivative

```
        Vector<double>  CalculateDerivative(Vector<double> x);
```

Applies the derivative of the activation in a pointwise fashion on the input vector

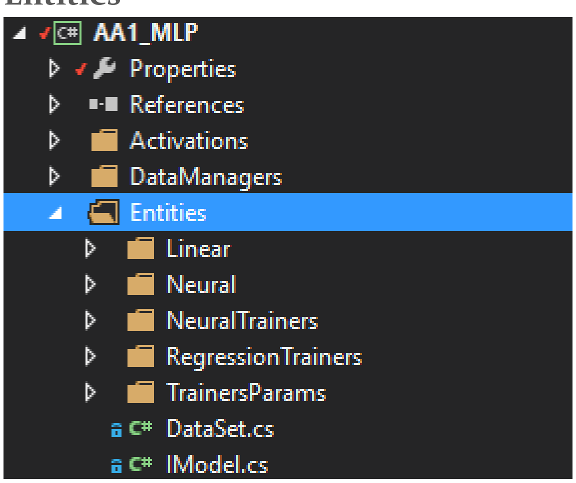
ActivationLeakyRelu.cs is a simple leaky relu activation function [x > 0 ? x : 0.01 * x]
Activations.cs is an enum listing all of the activations [currently unused, but for future refactoring we intend to utilize it]
ActivationSigmoid.cs is the sigmoid activation function
ActivationTanh.cs is the tanh activation function
IdentityActivation.cs is an identity activation function given x, return x, this is useful for layers without activation like the input layer in our case

DataManagers



A data manager is a class handling the loading and parsing of dataset files, it must provide an implementation for the IDataManger.cs interface function LoadData.
In the DataManagers folder, we provide our data managers for cup and monk's datasets, handling the parsing from the provided csv files and encoding the monks datasets into categorical features

Entities

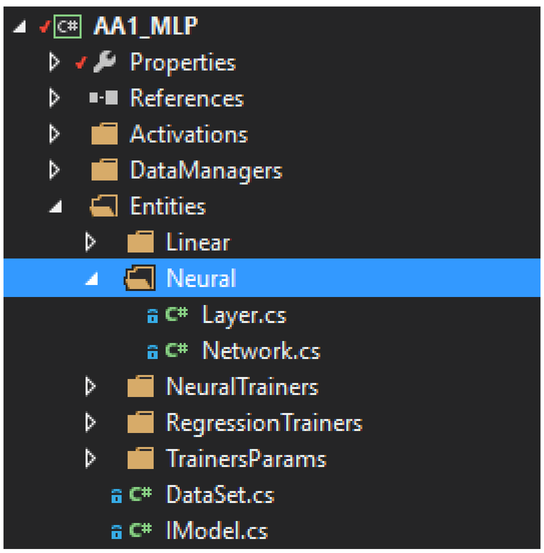


Contains our implementations for Linear and Neural models along with their trainers and

the parameters required per trainer.
DataSet.cs is a class representing a dataset with two properties; 1-Inputs, a matrix of doubles, each row in it represents one example in the dataset provided, 2- Outputs, also a matrix of doubles representing the corresponding targets per example
IModel.cs is a serializable abstract class providing a single function “Predict”, taking a vector of inputs [a row in the dataset for an example] and returns a vector of doubles representing the corresponding predicted output,
Any Model provided must implement the IModel.cs abstract class.

Following are more details on the folders contained in Entities

Entities.Neural



The subfolder “Neural” contains our implementations for Layer.cs a MLP layer and Network.cs a MLP.
Layer.cs contains the following properties:

```
int NumberOfNeurons { get; set; }
IActivation Activation { get; set; }
bool Bias { get; set; }
Vector<double> LayerActivationsSumInputs { get; set; }//The inputs to the layer’s neurons
Vector<double> LayerActivations { get; set; }//the output from the layer’s neurons after applying the activation on LayerActivationsSumInputs
Vector<double> Delta { get; set; }//layer local error
```

And it also provides the implementation of a forward propagation function that takes

```
public Vector<double> ForwardPropagation(Vector<double> inputOfPrevLayer, Matrix<double> weights, bool debug = false)
```

```
"inputOfPrevLayer" the output of the previous layer, before multiplying the weights between the two layers with it
"weights"the weights between the current layer and the previous one
```

The function multiplies the inputOfPrevLayer column vector by the weights matrix and sets the result to LayerActivationsSumInputs
Then applies the activation function on it and set the result in LayerActivations which is returned by the function eventually.

Networks.cs has two properties:
A list of layers that will hold the layers of the network and a list of weight matrices that will hold the weights matrices between the layers of the network

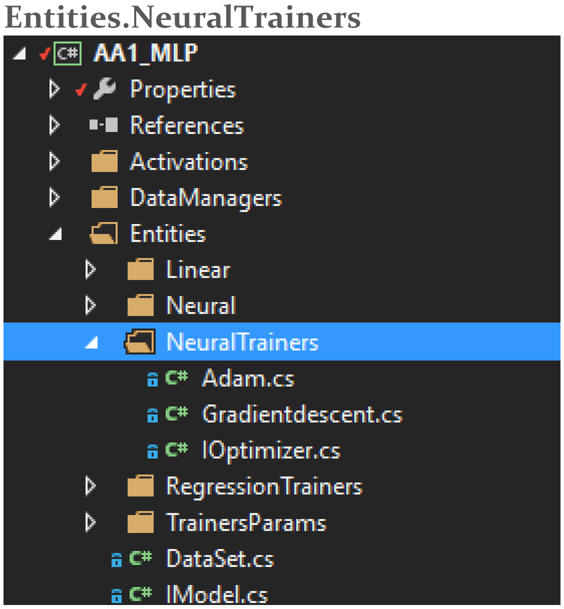
```
List<Layer> Layers { get; set; }
List<Matrix<double>> Weights { get; set; }
```

In the constructor of the Network, we provide three weight initialization mechanisms 1- Uniform, initializes the weights by values drawn from a uniform distribution between -0.7 to 0.7
2- Xavier, initializes the weights with values drawn from normal distributions with 0 mean and a variance of (2/fanIn+fanOut) where FanIn and FanOut are the number of incoming connections to the layer's neurons and the number of leaving connections from each neuron
3- He, similar to Xavier but with only fainIn considered.

In both of the given problems a simple uniform initialization should be sufficient since the architecture we provided for the network is definitely too shallow.

As the Network represents a model, it inherits the IModel abstract class and provides an implementation for its "predict function" which simply passes the signal of the input through the network and provides the network output.

Entities.NeuralTrainers



Holds the implementations for our optimizers,
An Optimizer algorithm must inherit the IOptimizer.cs abstract class and provide the implementation to its function Train

```
public abstract List<double[]> Train(TrainerParams trainParams);
```

Also, it needs to provide a TrainerParams class representing the required parameters for the algorithm [more in the TrainerParams is introduced shortly in the TrainersParams folder]

The Train function returns a list of arrays of doubles that will hold the learning curve of the training process

Adam.cs

Provides an implementation of the Adam optimizer, our chosen algorithm of the class of accelerated gradient methods

Quoting from [3]

Algorithm 1: *Adam*, our proposed algorithm for stochastic optimization. See section 2 for details, and for a slightly more efficient (but less clear) order of computation. g_t^2 indicates the elementwise square $g_t \odot g_t$. Good default settings for the tested machine learning problems are $\alpha = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\epsilon = 10^{-8}$. All operations on vectors are element-wise. With β_1^t and β_2^t we denote β_1 and β_2 to the power t.

Require: α: Stepsize
Require: $\beta_1, \beta_2 \in [0, 1)$: Exponential decay rates for the moment estimates
Require: $f(\theta)$: Stochastic objective function with parameters θ
Require: θ_0: Initial parameter vector
  $m_0 \leftarrow 0$ (Initialize 1st moment vector)
  $v_0 \leftarrow 0$ (Initialize 2nd moment vector)
  $t \leftarrow 0$ (Initialize timestep)
  while θ_t not converged **do**
    $t \leftarrow t + 1$
    $g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$ (Get gradients w.r.t. stochastic objective at timestep t)
    $m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$ (Update biased first moment estimate)
    $v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$ (Update biased second raw moment estimate)
    $\widehat{m}_t \leftarrow m_t / (1 - \beta_1^t)$ (Compute bias-corrected first moment estimate)
    $\widehat{v}_t \leftarrow v_t / (1 - \beta_2^t)$ (Compute bias-corrected second raw moment estimate)
    $\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \widehat{m}_t / (\sqrt{\widehat{v}_t} + \epsilon)$ (Update parameters)
  end while
  return θ_t (Resulting parameters)

```
//The simple implementation of adam at the beginning of the professor's paper
weightsUpdates[y] = weightsUpdates[y] / numberOfBatchExamples;
firstMoment[y] = passedParams.beta1 * prevFirstMoment[y] + (1 -
passedParams.beta1) * (-1 * weightsUpdates[y]);
secondMoment[y] = passedParams.beta2 * prevSecondMoment[y] + (1 -
passedParams.beta2) * weightsUpdates[y].PointwisePower(2);
mhat[y] = firstMoment[y] / (1 - Math.Pow(passedParams.beta1, adamUpdateStep));
vhat[y] = secondMoment[y] / (1 - Math.Pow(passedParams.beta2, adamUpdateStep));
var finalUpdates = (passedParams.learningRate *
mhat[y]).PointwiseDivide((vhat[y].PointwiseSqrt() + passedParams.epsilon));
passedParams.network.Weights[y] -= finalUpdates;
```

Gradientdescent.cs

Provides our implementation of the stochastic gradient descent + Momentum both the classic [new weights Updates = momentum rate* old weights updates +learning rate the weights gradient] [4]

$$
\begin{aligned}
v_{t+1} &= \mu v_t - \varepsilon \nabla f(\theta_t) && (1)\\
\theta_{t+1} &= \theta_t + v_{t+1} && (2)
\end{aligned}
$$

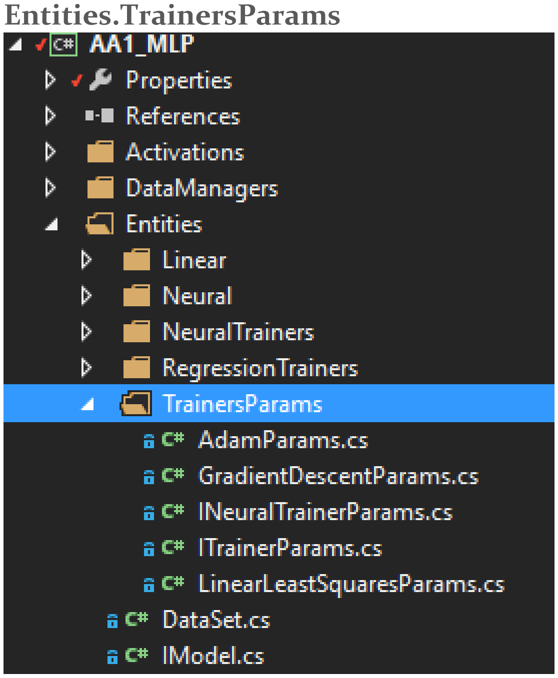
and Nestrov [new weights Updates = momentum rate* old weights updates + learning rate*the weights gradient at the updated weights][4]

$$
\begin{aligned}
v_{t+1} &= \mu v_t - \varepsilon \nabla f(\theta_t + \mu v_t) && (3)\\
\theta_{t+1} &= \theta_t + v_{t+1} && (4)
\end{aligned}
$$

Where v_t is the weights updates at step t and Theta_t are the weights at step t
mu is the momentum weight and epsilon is the learning rate.

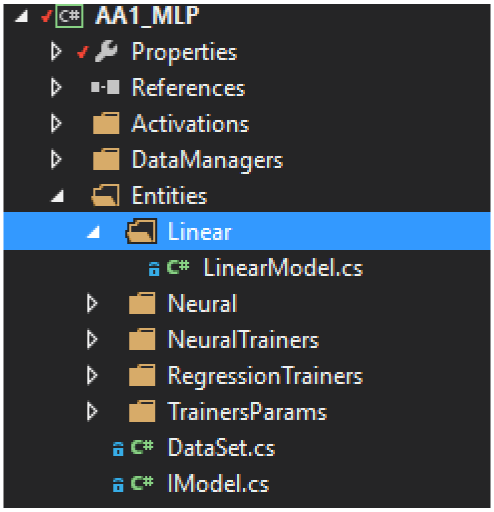
With the possibility of using an L2 regularization by penalizing the weights with 2*regularization rate * current weights

Entities.TrainersParams



Holds the trainers parameters needed to pass around a trainer's hyperparameters, each provided trainer needs to have a class for its parameters inheriting the TrainerParams class and adding any additional parameters required by the optimizer
We see also a Params.cs for each solver or trainer we have where ITrainerParams is inherited by the LinearLeastSquaresParams.cs for the linear models params and INeuralTrainerParams.cs is inherited by the MLP optimizers params[AdamParams.cs and gradientDescnetParams.cs]

Entities.Linear



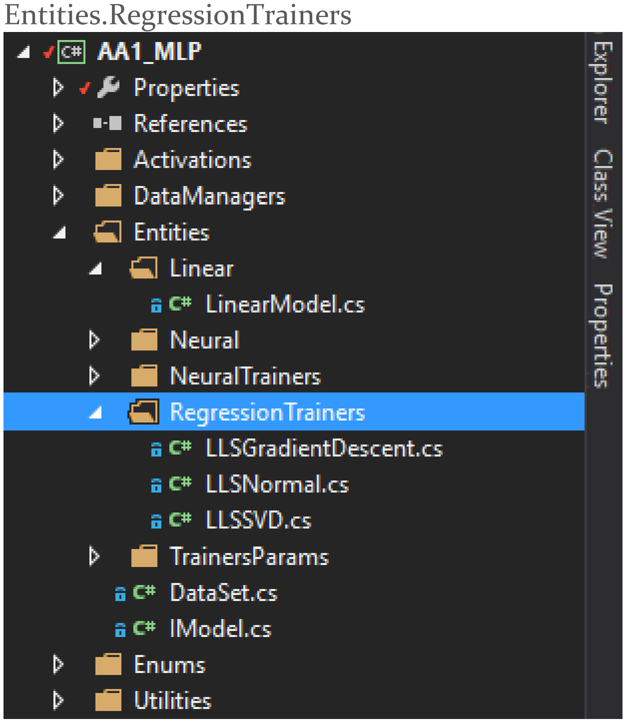
Contains the LinearModel.cs a simple linear model inheriting IModel and implementing its function predict, also providing properties required by a LinearModel

Weights is the weights [single row]matrix of the model

Degree, a property we introduced to provide a Polynomial regression model,

Set to higher than one, while using an LLSGradientDescnt solver, will utilize higher order features in the model, for example set to two the features and each feature squared will be considered, set to three, the features, the squared features and the cubed features will be considered

Entities.RegressionTrainers



Holds our implementation for the linear regression solver of our choice, we provide three solvers

LLSGradientDescent.cs is the plain vanilla gradient descent, with a learning rate parameters as a step without line search, or any other hyperparameters
LLSNormal.cs a simple normal equations solver

```
//Assuming the data matrix A has linearly independent rows or columns to properly use the pseudo inverse otherwise we might have a problem!
//Ax=b  ->   A'Ax=A'b -> x = (inv(A'A))A'b ->pinv(A)b  where A is our data, b is our vector of targets
```

LLSSVD.cs a simple SVD solver;
Quoting from the CM lecture Linear Algebra part [1]

Least squares with the SVD

One can solve least-squares problem also with (thin) $A = USV^T$.
Same derivation as with QR:

$$\|Ax - b\| = \|USV^Tx - b\| = \|S\underbrace{V^Tx}_{=y} - U^Tb\|$$

$$= \left\| \begin{bmatrix} \sigma_1 y_1 \\ \sigma_2 y_2 \\ \vdots \\ \sigma_n y_n \\ 0 \\ \vdots \\ 0 \end{bmatrix} - \begin{bmatrix} u_1^T b \\ u_2^T b \\ \vdots \\ u_n^T b \\ u_{n+1}^T b \\ \vdots \\ u_m^T b \end{bmatrix} \right\|$$

If all the σ_n are different from 0, the minimum is when $y_i = \frac{u_i^T b}{\sigma_i}$ (and then $x = Vy$).

Putting everything together, one gets

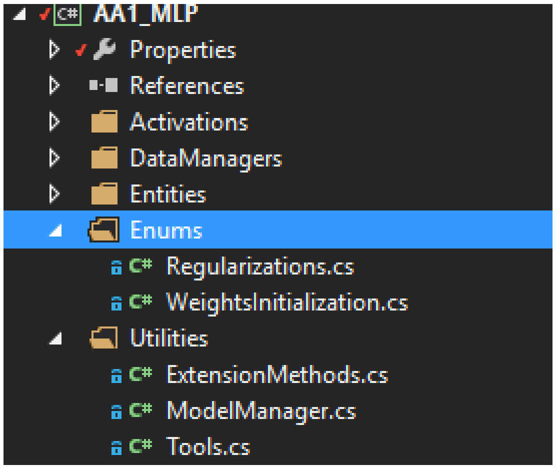
$$x = \sum_{i=1}^{n} v_i \frac{u_i^T b}{\sigma_i}.$$

Again, we need only the thin SVD to compute it.

Note that the small σ_i's contribute more to the solution (unless also $u_i^T b \approx 0$).

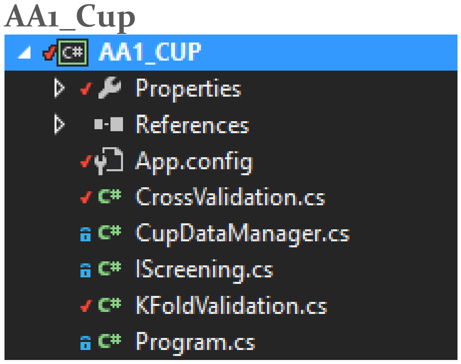
```
var svd = trainParams.trainingSet.Inputs.Svd();
int r = trainParams.trainingSet.Inputs.Rank();
var d = svd.U.Transpose().Multiply(trainParams.trainingSet.Labels);
var temp = CreateMatrix.Dense<double>(svd.VT.ColumnCount, 1);
temp.SetSubMatrix(0, 0, d.SubMatrix(0, r, 0,
1).Column(0).PointwiseDivide(svd.S.SubVector(0, r)).ToRowMatrix().Transpose());
passedParams.model.Weights = svd.VT.Transpose().Multiply(temp);
```

Enums and Utilities



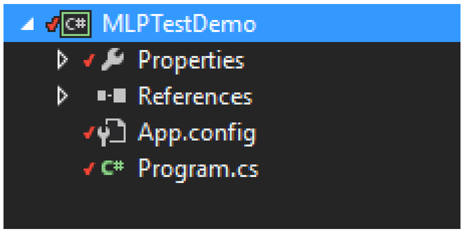
Enums holds holds our enums for Regularizations.cs [either L2 or None for now]
WeightsInitialization.cs [He, Uniform and Xavier]
Utilities holds our extension methods to C#, we found we needed a shuffle function for a C# List<> and a matrix to vector multiplication and a vector to matrix multiplication functions
Both returning a single row matrix as the result
Tools.cs provides a compute accuracy function for a model
ModelManager.cs to save and load a model

AA1_Cup



The AA1_CUP project is provided mainly to screen a model by running experiments using train/validation/test utilizing CrossValidation.cs or k-fold cross validation utilizing KFoldValidation.cs
Any validation strategy needs to implement the IScreening.cs interface and provide an implementation to the "Screen" function.
CupDataManager.cs is a Data Manger implementing the IDataManager.cs abstract class in AA1_MLP providing the body for the LoadData function
Program.cs is the tradition C# class with a main function to run code in a console app utilizing the screening mechanisms implemented

MLPTestDemo



Program.cs is providing the main function doing several tests using the different models we had on the monk's datasets, we utilized this project mainly for building our library.

3) FUTURE IMPROVEMENTS

Moving the parameters from the train function into their respective trainers and keeping only the common parameters

Moving the enums into one folder

Adding SVM

Adding more advanced artificial neural network topologies, namely Convolutional Neural Networks and Recurrent Neural Networks

Parallel computation for better performance

Better names!!!

Adding a complete documentation to the code provided and generating the standard docs

Refactoring the screening project to allow for different models and algorithms without a custom function per each, this will be possible after moving the trainers parameters inside the trainers

6. Provided Files and Folders

The main project is under the Folder Monks,
please open the solution from \Monks\AA1_Monks\ AA1_Monks.sln

Under Monks\UsedFiles exists the following:

Folder/File	Description
KFoldValidationPlots	the plots of the K-Fold validations reported in section 2>1 MLP
LastModelsPlots	the plots of the last models reported in 2>1>Final Models
ML-17-PRJ lecture package-20171225	the folder provided by prof. Miceli for the CUP [2]
Notebooks	the notebooks used in this project
TrainValSplits	the 60%40% train-validation splits both in regular form and standardized
OMG_ML-CUP17TS.csv	our CUP submission
libs	Where encog and math.NET .dll files live, please reference these files in the projects provided before running them

7. References

[1] Antonio Frangioni, Federico Poloni, UNIPI Computational Mathematics for Learning and Data Analysis course

[2] Alessio Micheli, UNIPI AA1 Machine learning foundations course

[3] Diederik P. Kingma, Jimmy Lei Ba: ADAM: A METHOD FOR STOCHASTIC OPTIMIZATION
https://arxiv.org/pdf/1412.6980.pdf

[4] Ilya Sutskever,James Martens, George Dahl, Geoffrey Hinton, On the importance of initialization and momentum in deep learning
http://www.cs.toronto.edu/~fritz/absps/momentum.pdf

[5] EncogML the C# library we compared our MLP results against
http://www.heatonresearch.com/encog/

[6] scikit learn the python library we compared our linear models against
http://scikit-learn.org/stable

[7] Math.Net our library of choice for providing linear algebra to C#
https://www.mathdotnet.com/

[8] Adam Tutorial
https://machinelearningmastery.com/adam-optimization-algorithm-for-deep-learning/

[9] CS231n Convolutional Neural Networks for Visual Recognition
http://cs231n.github.io/neural-networks-3/

[10] Xavier weight initialization
http://proceedings.mlr.press/v9/glorot10a/glorot10a.pdf

[11] He Weight initialization

https://www.cv-foundation.org/openaccess/content_iccv_2015/papers/He_Delving_Deep_into_ICCV_2015_paper.pdf

[8] The MONK's Problems
A Performance Comparison of Dierent Learning Algorithms
S.B. Thrun, J. Bala, E. Bloedorn, I. Bratko, B. Cestnik, J. Cheng, K. De Jong, S. D zeroski,

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