Math6373 Final

Application of MLP to predict next day stock price of company based on moving averages and past prices

Submitted by

Kishore Tumarada

1 Prediction Task :

Select one major stock on the US stock market. On day "t", let S(t) be the price of this stock at closing time. On each day "t", we want *to predict* the future stock price S(t+1) given the last 20 observed stock prices S(t) , S(t-1), S(t-2), ..., S(t-19).

Data Set : **Let t=1, 2, ... N be the days on which the US stock exchange was open during the time period 2014-2015-2016-2017 . Download the time series S(t) for t= 1,2, ..., N .**

Answer:

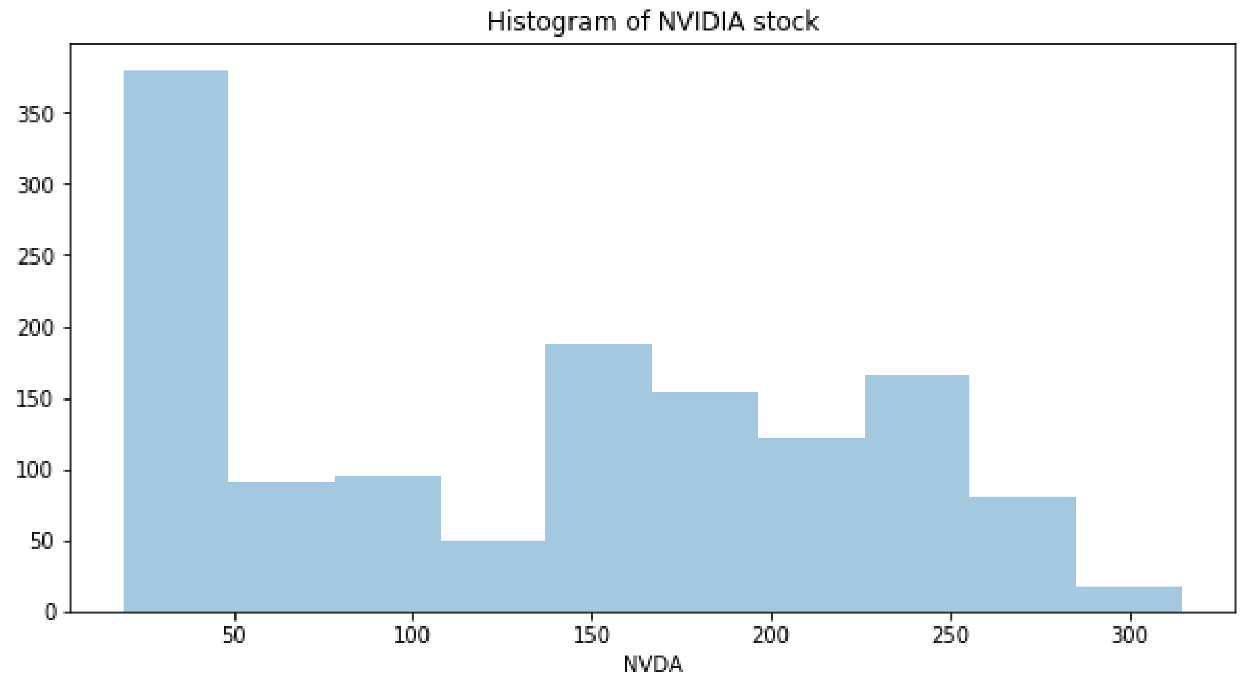
I have chosen NVIDIA Corporation (NVDA) stock for this prediction problem. The selected time range was from January 2nd , 2015 to April 30th , 2020. The stock market is open for 1341 days during this period. So total number of cases are N = 1341. Following table shows the descriptive statistics of the stock:

index	NVDA
count	1341
mean	134.906
std	85.978
min	19.14
25%	36.45
50%	148.9
75%	209.16
max	314.7

Table 1: Descriptive statistics of input

As per above table, the price range of stock is quite high from $ 19.14 to $ 314.70.

The following figure shoes the histogram of closing stock price of Nvidia:



The histogram indicates that highest number of stock prices are below $50.

2 Pre-Processing:

Replace isolated missing values S(t) by the mean of two actual values closest to time t. If there are too many missing values, download another stock . For 20 ≤t≤N-1 , compute the following three moving averages of the time series S :

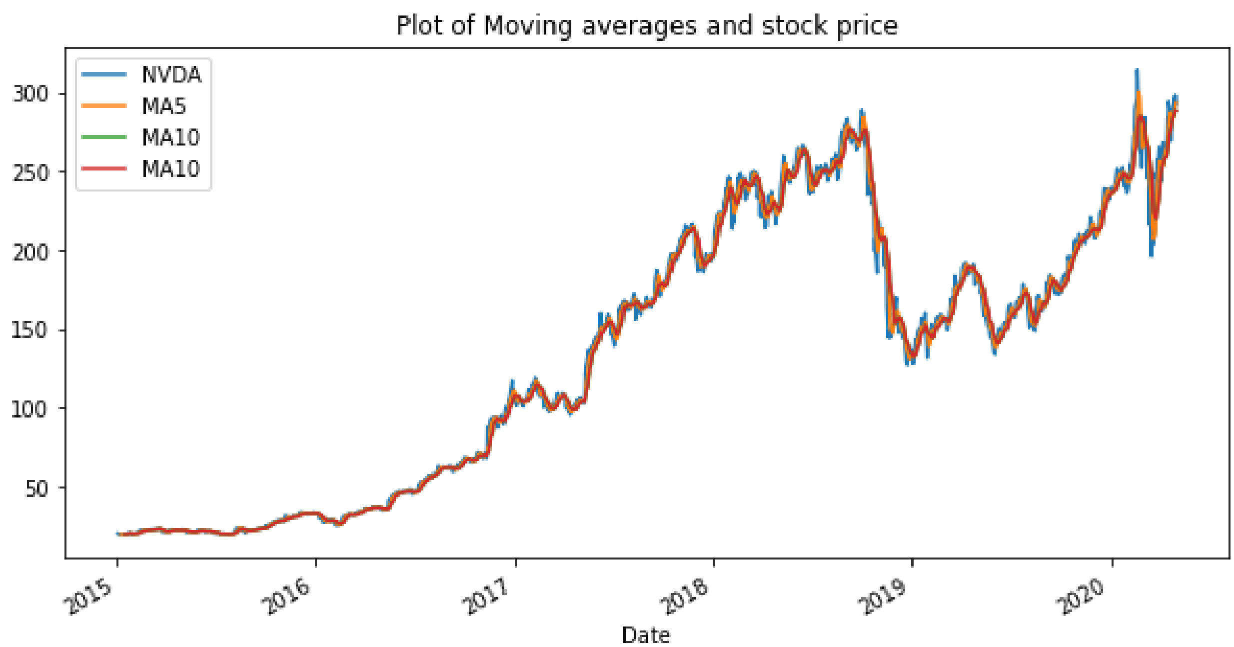
MA5(t) = [S(t-4) + S(t-3) + S(t-2) +S(t-1) + S(t)] /5
MA10(t) = [S(t-9) + S(t-8) + ... + S(t)] /10
MA20(t) = [S(t-19) + S(t-18) + ... + S(t)] /20
Plot the 4 curves S(t) , MA5(t), MA10(t), MA20(t), on the same graph

Answer:

As the stock does not have any missing values, I did not do any imputation of the data.

Moving averages smoothen the price trend by filtering noise from random short-term price fluctuations. It would also highlight longer term trends or cycles. It is often used as metric for technical analysis of financial data. Here I have calculated Moving average for 5 days, 10 days and 20 days.

Following figure shows all 4 curves – S(t) (NVDA) , MA5, MA10, MA20.



3. Training and Test sets for an MLP predictor:

3.1. On each day t ≥ 20, the *recent past* of the series S will be defined as the 1x18 line vector

Vt = [MA5(t), MA10(t), MA20(t), S(t), S(t-1), S(t-2) ..., S(t-13), S(t-14)]

For 20 ≤ t ≤ N-1 the input vector Vt will be the input of our MLP predictor, which will have a *single* output neuron with state Zt. This output Zt will be the MLP prediction computed on day t for the *target* TARGt = S(t+1), which is not known at time t.

For this prediction task, we have a data set of (N-20) "cases" Case20 Case21 Case22 ... CaseN-1 , indexed by t= 20, 21, ... N-1. Each Caset is described by 18 features = 18 coordinates of vector Vt. The TRUE output to be predicted at time t is the yet unknown TARGt = S(t+1).

The data set of (N-20) cases for MLP prediction learning is denoted *PredCases* = {all pairs (Vt, TARGt) with t= 20, 21, ... N-1}

Answer:

Original dataset has 1341 cases of closing stock prices. After calculating moving averages , stock prices have been transformed into the format - [MA5(t), MA10(t), MA20(t), S(t), S(t-1), S(t-2) ..., S(t-13), S(t-14)] . As MA20 cannot be calculated for first 19 cases, this metric is null for these rows, hence they have been dropped. Similarly last case does not have Target price(S(t+1), hence it has also been dropped. As a result, there will be 1321 cases(1341 – 20) in total.

After data transformation as described above, Predcases dataset has been constructed. Following table shows the descriptive statistics of first 5 columns of PredCases dataset :

index	MA5	MA10	MA20	NVDA(t)	NVDA(t-1)
count	1321	1321	1321	1321	1321
mean	136.02	135.52	134.54	136.441	136.23
std	85.189	84.973	84.604	85.397	85.34
min	19.504	19.695	19.775	19.2	19.2
25%	43.584	40.868	38.306	44.4	44.33
50%	150.64	150.01	149.79	149.97	149.95
75%	209.09	208.79	207.43	209.61	209.61
max	300.77	289.02	276.23	314.7	314.7

Table 2: Descriptive statistics of Predcases -1

Following table shows descriptive statistics of last 5 columns :

index	NVDA(t-11)	NVDA(t-12)	NVDA(t-13)	NVDA(t-14)	TARGt
count	1321	1321	1321	1321	1321
mean	134.208	134.007	133.818	133.634	136.648
std	84.899	84.858	84.833	84.817	85.443
min	19.2	19.2	19.2	19.2	19.31
25%	36.74	36.48	36.45	36.45	45.17
50%	148.84	148.83	148.77	148.77	150.07
75%	207.84	207.78	207.66	207.63	209.63
max	314.7	314.7	314.7	314.7	314.7

Table 3: Descriptive statistics of Predcases -2

3.2. Randomly Split the set *PredCases,* with 90% cases in the training set *PredTRAIN*, and 10 % cases in the test set *PredTEST*

Answer:

The PredCases dataset has been split to train and test data in the ratio of 9:1 using scikit learn library's test_train_split function.

After splitting PredCases dataset of size (1321,18), I have obtained:

1. Size of Train data as (1188, 18) and
2. Size of Test data as (133,18) .

4 MLP predictor:

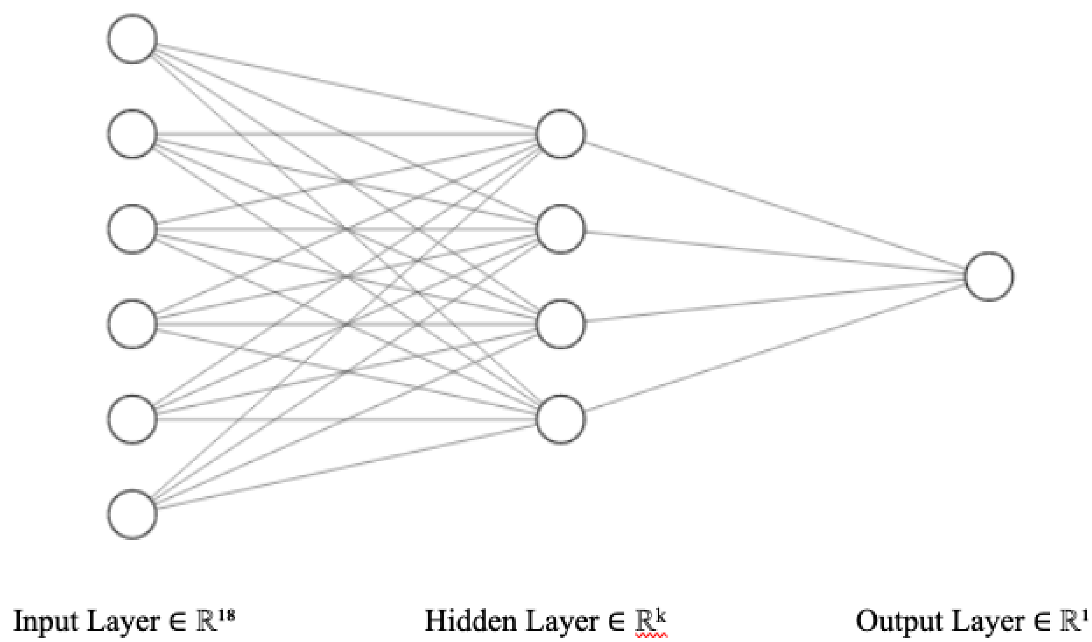
4.1. Our MLP predictor *(MLPpred)* will have the simple 3 layers architecture

INPUT ==> HiddenLayer K ==> OUTPUT with dim(INPUT) = 18 , dim(OUTPUT) = 1

dim(K) = k *to be selected below*. For each training input Vt we want the MLP output Zt to be close to TARGt= S(t+1).

Answer:

Following figure shows the schematic diagram of MLP predictor with 3 layers.

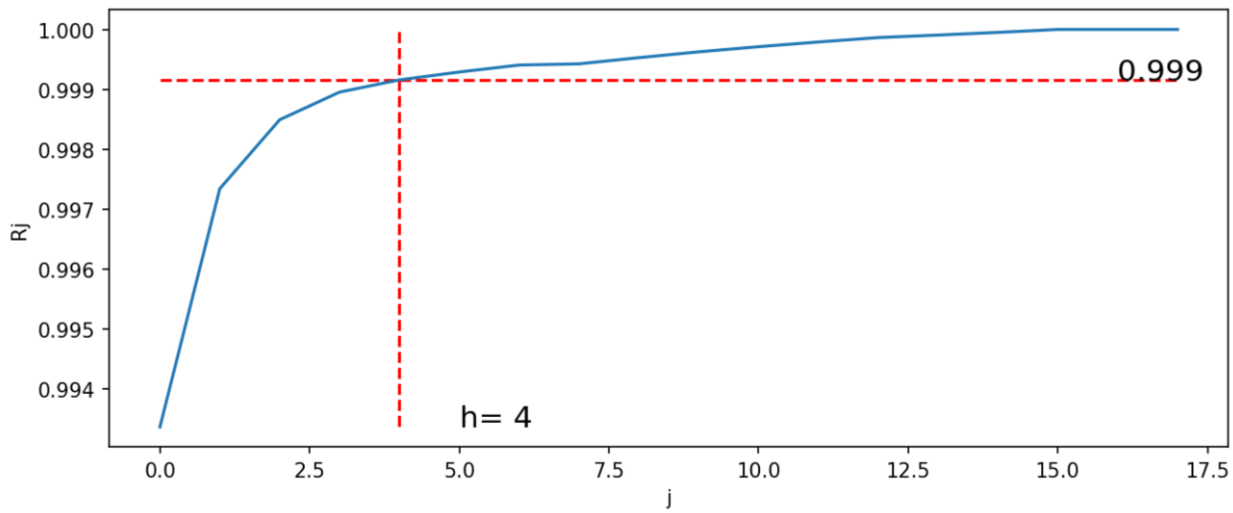


4.2. Implement PCA on the set of all input vectors Vt , with t= 20 ,21, ..., N . Determine the number k of principal components which preserves *95%* of the variance (see HW3) and fix *dim(K)= k*.

Answer:

After implementing PCA and calculating Proportion of Variance explained, PVE, for 18 eigen values, I found that just the first Principal component can explain about 99.35% variance in the data. This might be because as all the 18 features belong to only one company, they are highly correlated. But building a hidden layer with one dimension may not be robust network.

Hence a higher variance proportion of 99.9% is considered and following scree plot shows eigen value count, j Vs Rj(PVE ratio).



From above plot, we can see that 4 eigen values are required to explain a variance of 99.91% in the data. Hence hidden layer dimension k = 4 has been chosen for the neural network.

4.3. Compute the number w of weights and thresholds in this MLP and compare w to the number of informations provided by the training set.

Answer:

Robustness ratio is a useful metric to know, as it shows us the proportions of known to unknown, or constraints to unknown parameters.

$$Robustness = \frac{constraints}{parameters}$$

As training data has 1188 cases and output has 1 dimension(since Regression problem), number of constraints = 1188 * 1 = 1188

In the MLP, dimensions of all three layers are :

1. dim (INPUT) = 18 ,
2. dim (HIDDEN LAYER) k = 4,
3. dim (OUTPUT) = 1

So total parameters in 3 layered MLP = weights [18 * 4 + 4 * 1] + biases [4 + 1] = 81

Therefore, robustness ratio = 1188/81 = 14.67. It is a reasonable ratio to build a neural network.

5. Training of the MLP predictor:

5.1. Implement an automatic training on the training set PredTRAIN, with the options :

RELU response, Loss = "MSE", Stochastic Gradient Descent *or* ADAM, Batch Learning, Early Stopping.

5.2. Let RMSE be the root mean squared error $\sqrt{MSE}$. Plot the evolution of RMSE versus the number of batches (one curve for the training set and one for the test set) . Compare these two curves.

Answer:

Base model with arbitrary parameters and performance:

Keras model has following structure with 81 parameters:

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense (Dense)                (None, 4)                 76
_________________________________________________________________
dense_1 (Dense)              (None, 1)                 5
=================================================================
Total params: 81
Trainable params: 81
Non-trainable params: 0
_________________________________________________________________
```

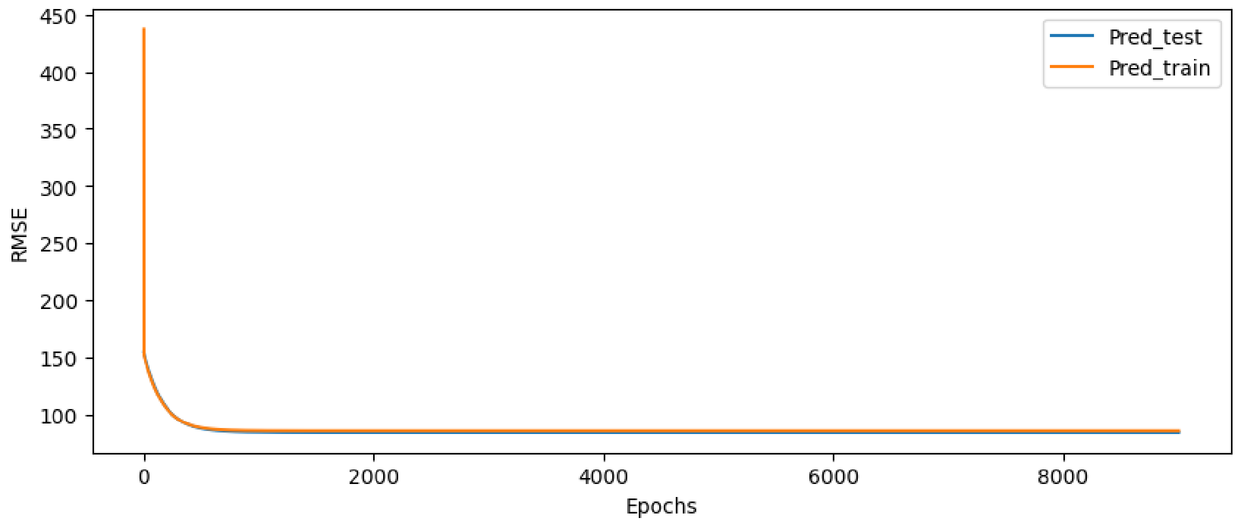
For base level model , following hyper parameters have been selected :

```
MLP_HIDDEN_LAYER_SIZE = 4
#model parameters
BIA_INI_H = 10
BIA_INI_O = 10
KERNEL_INI = 'glorot_uniform'
#fitting parameters
MLP_LEARNING_RATE = 5e-5
MLP_DECAY_RATE = 1e-5
MLP_EPOCH_SIZE = 40000
PATIENCE = 2000
MLP_BATCH_SIZE = 32
```

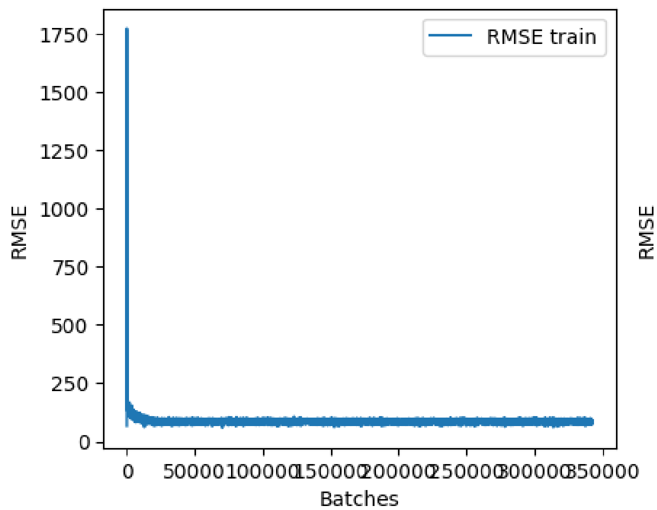
```
selected_optimizer = keras.optimizers.SGD(learning_rate =
MLP_LEARNING_RATE, decay= MLP_DECAY_RATE)
```

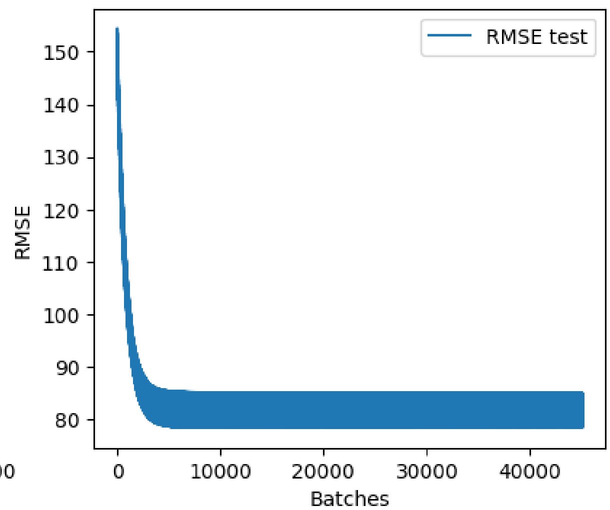
Loss of the above model on test data is MSE(test) = 7065.44 and RMSE(test) = 84.05.

Following plot shows the rmse loss for this model. It shows that RMSE has fallen down to approx 80-90 by around 800-900 epochs and later flattened. Both Train and test have same error range.

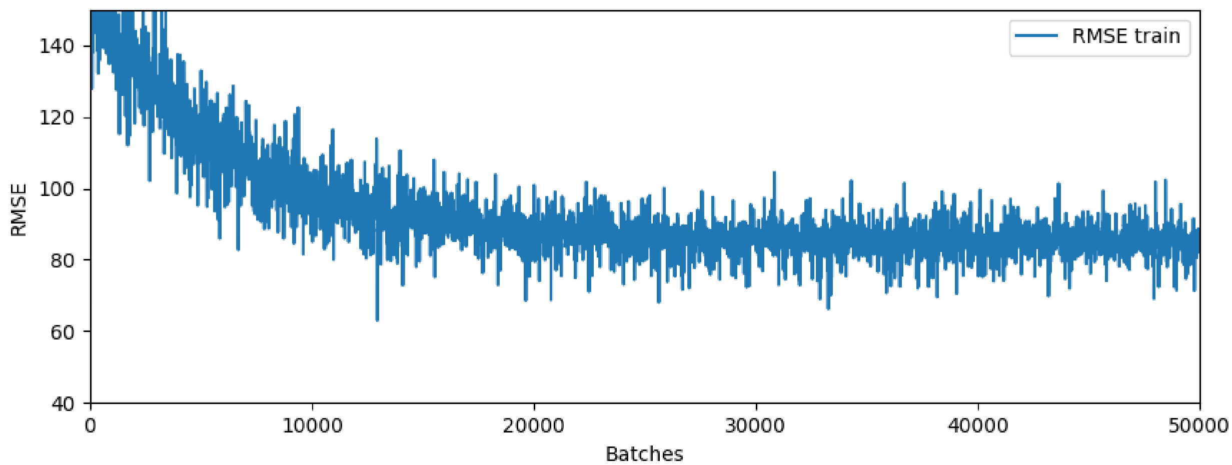


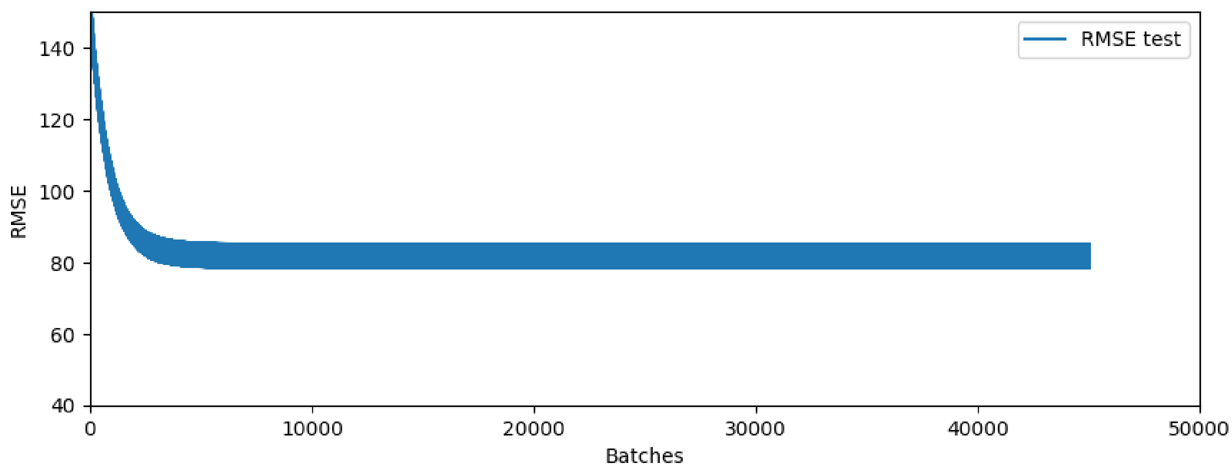
The below plot shows the RMSE per batch. As they are on different scale, comparison is difficult. Hence I have plotted same scale plot again.





Following plot shows RMSE plot of test and train on same scale. It shows that RMSE has stabilized in the range of 80 to 100 after 10000 batches for train and 80 to 90 by 5000 batches.

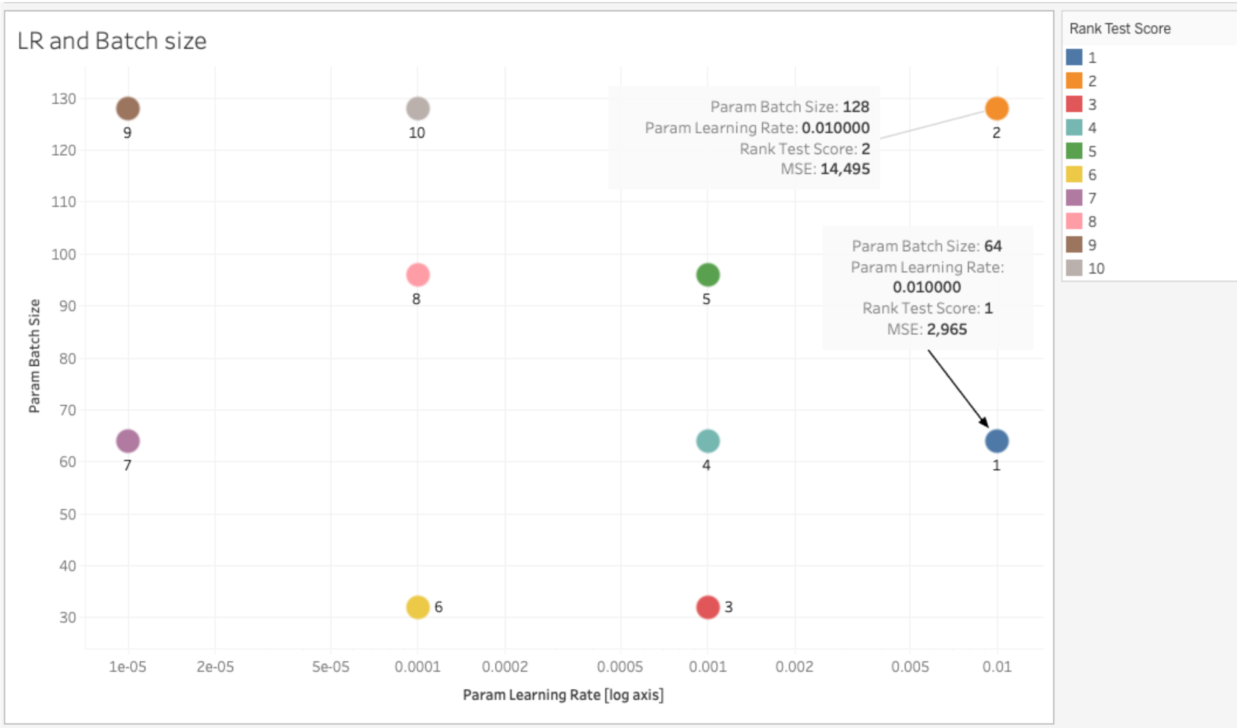


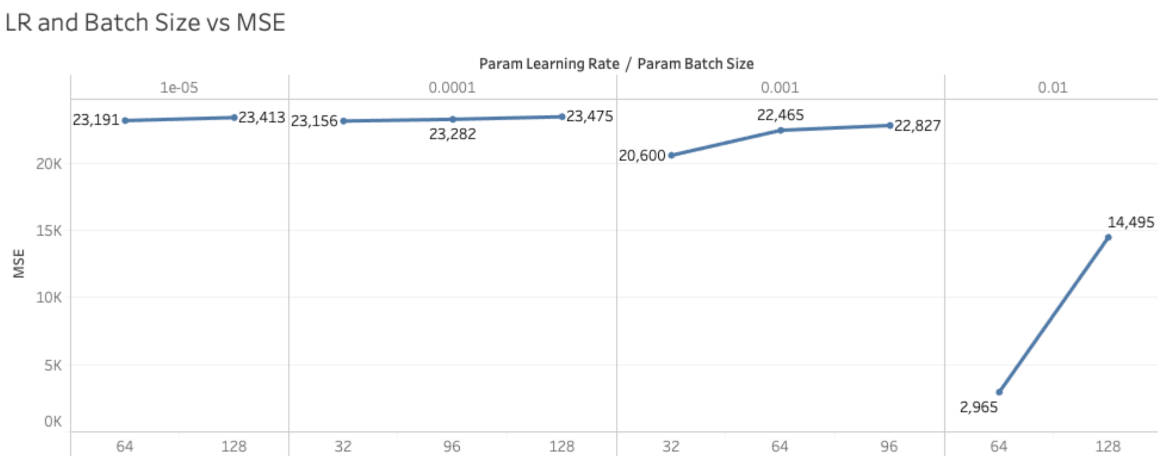


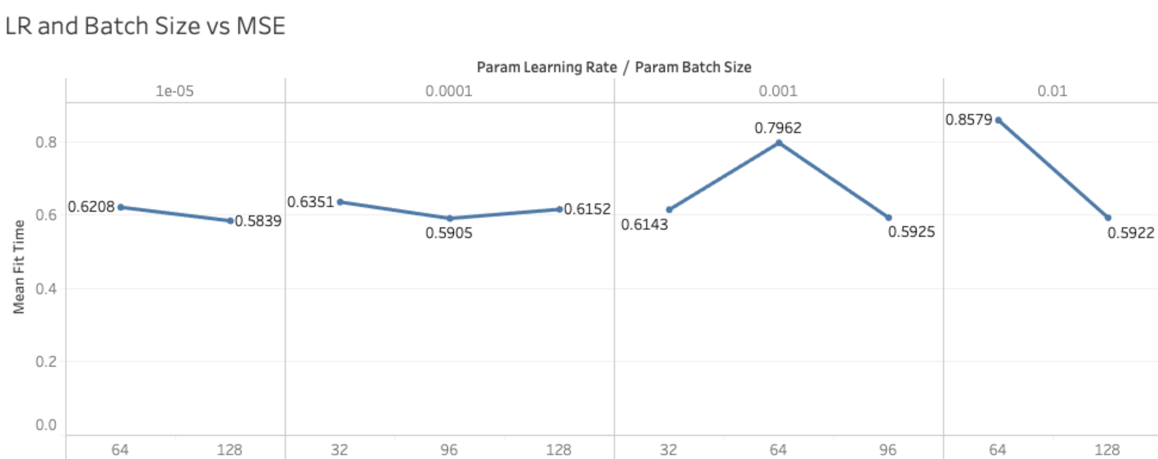
Hyper parameter tuning :

I have used Adam optimizer in place of Stochastic gradient descent, as the former is optimizing the model at faster rate.
For tuning, I have chosen Randomized search with cross validation over Grid search Cross validation as later is exhaustive and less efficient in arriving at best parameters. For quick tuning, number of epochs are chosen as 3000 with early stopping of 200.

1st iteration of tuning results on the hyper parameter space shown in below scatterplot.

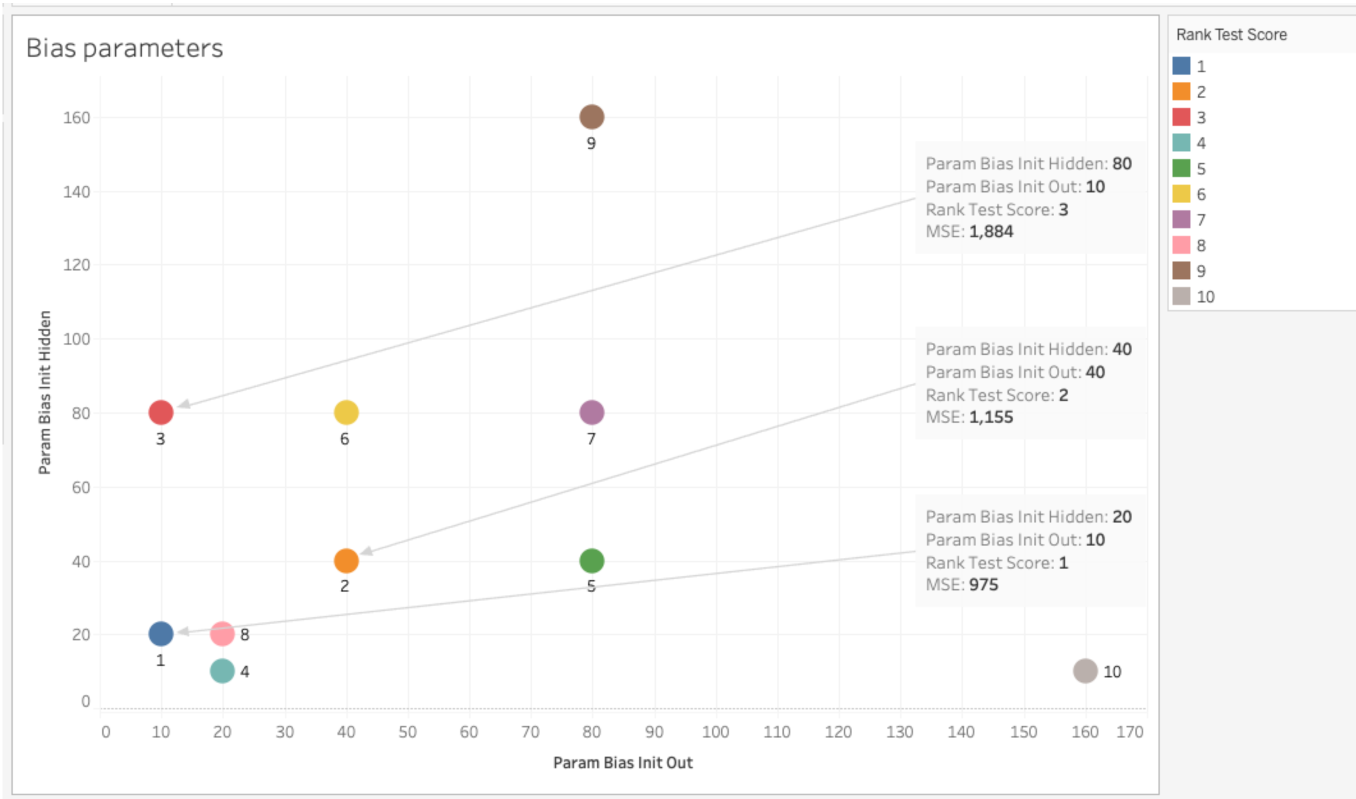

LR and Batch size
Rank Test Score
Param Batch Size: 128
Param Learning Rate: 0.010000
Rank Test Score: 2
MSE: 14,495
Param Batch Size: 64
Param Learning Rate:
0.010000
Rank Test Score: 1
MSE: 2,965
Param Batch Size
Param Learning Rate [log axis]


LR and Batch Size vs MSE
Param Learning Rate / Param Batch Size
1e-05
0.0001
0.001
0.01
23,191
23,413
23,156
23,282
23,475
20,600
22,465
22,827
14,495
2,965
MSE

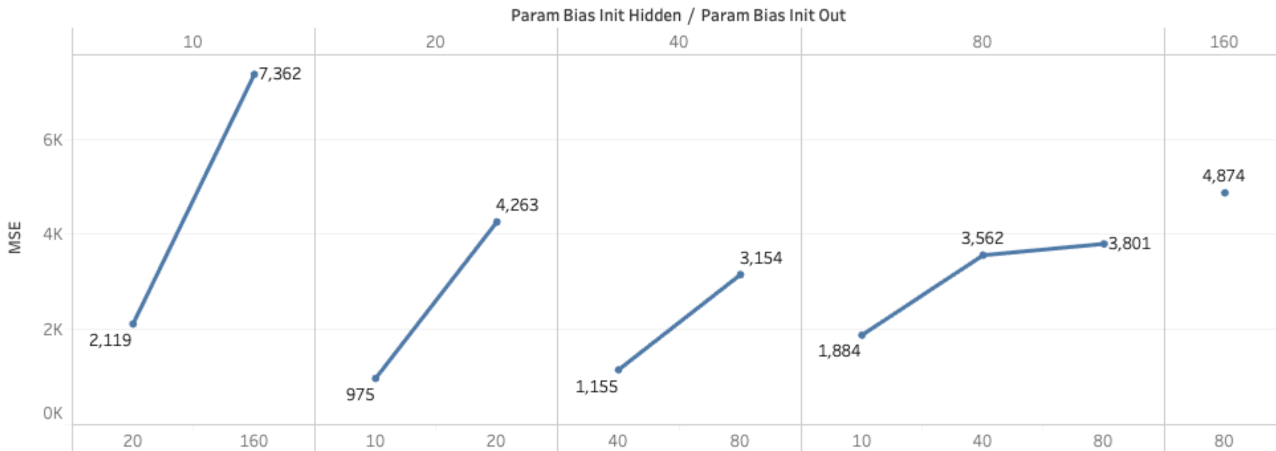

LR and Batch Size vs MSE
Param Learning Rate / Param Batch Size
1e-05
0.0001
0.001
0.01
0.6208
0.5839
0.6351
0.5905
0.6152
0.6143
0.7962
0.5925
0.8579
0.5922
Mean Fit Time

Both of the above plots show that for Learning rate = 0.01, MSE is quite less and decreases further for batch size of 64, though with a slight increase in fitting time.
Best parameters chosen are Learning rate = 0.01 and Batch size = 64.

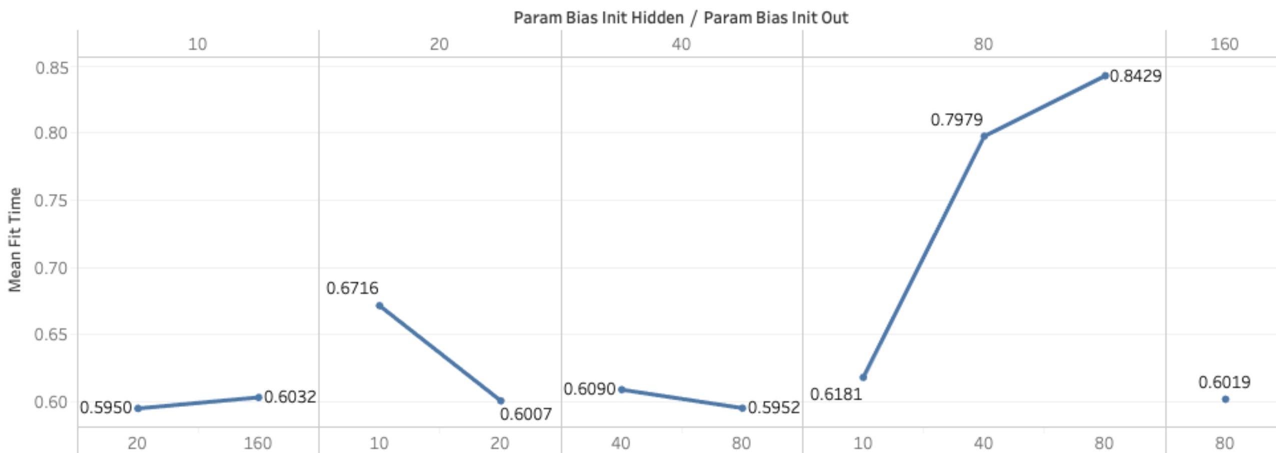
2nd iteration of tuning for Bias initializer for hidden layer and output layer is shown in below figure:



Bias initialization vs MSE

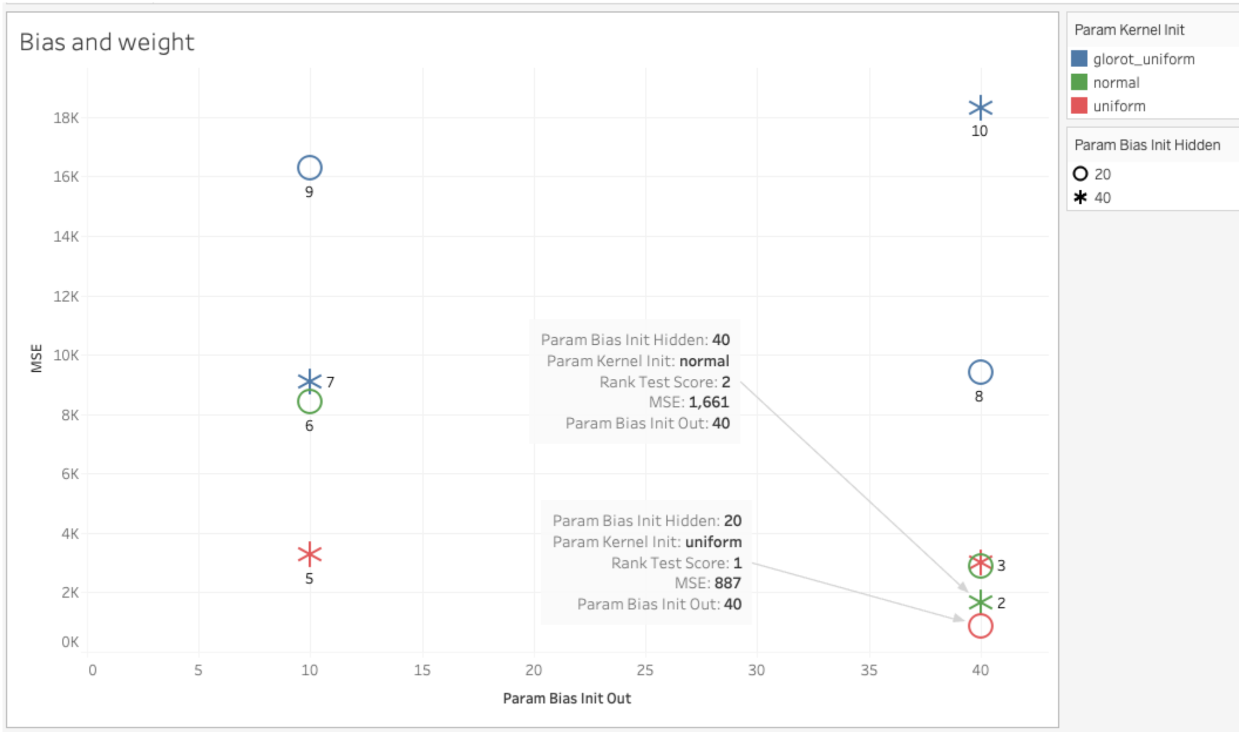


Bias parameters and time



Above plots show that bias parameters – (hidden, outer) – (20,10) and (40,40) have low MSE with marginal difference in fitting time.
So both sets of parameters are used for fine tuning along with weight initializer.

Following scatter plot shows tuning hyper parameter space for kernel initializer (weights) along with bias initializers.



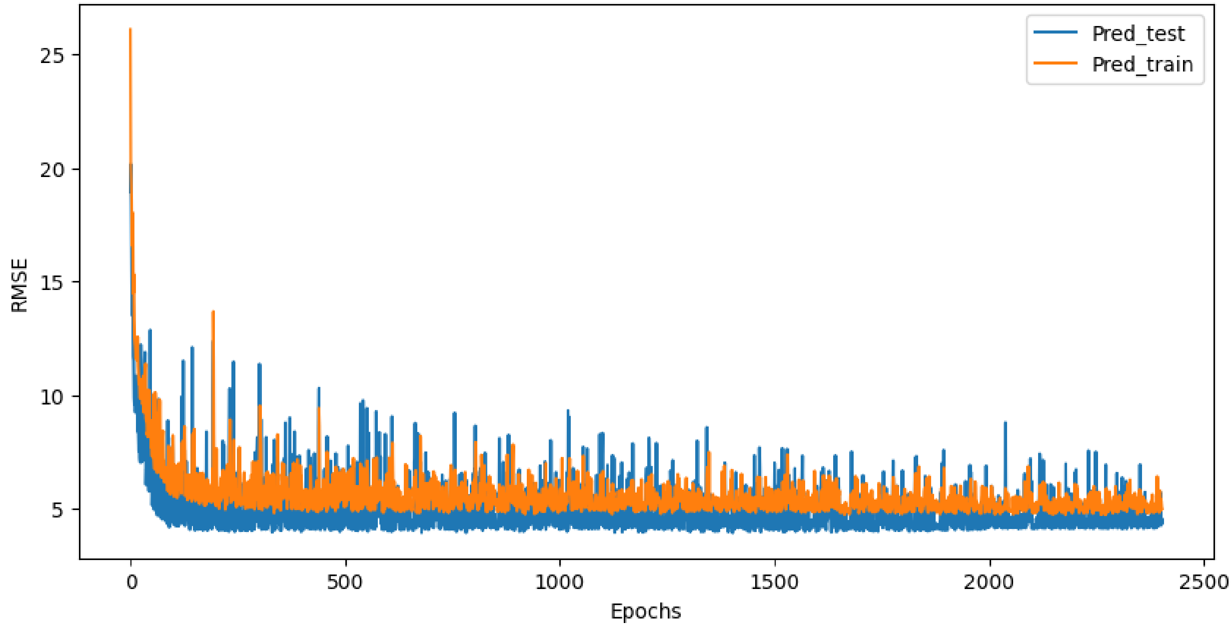
From the above plot, we can see that best parameters with low MSE are weight Kernel initializer with 'uniform' distribution, bias initializers are 20 for Hidden layer and 40 for outer layer.

Total set of best parameters are **{'learning_rate': 0.01, 'kernel_init': 'uniform', 'bias_init_hidden': 20, 'bias_init_out': 40, 'batch_size': 64}.**

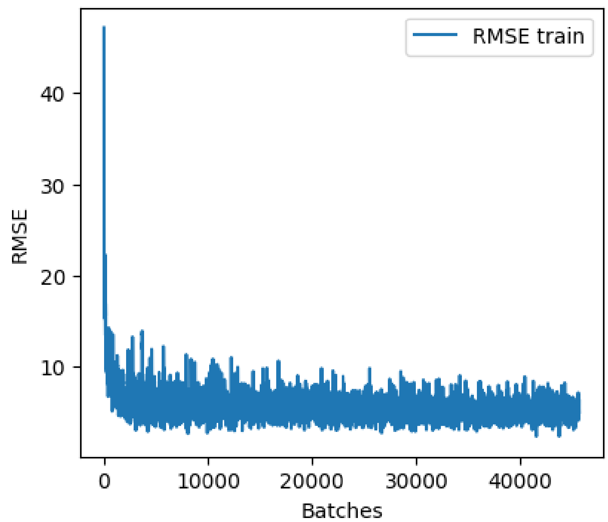
Now full training on data with above parameters along with epochs = 40,000 and early stopping patience level = 1000 gave an MSE loss on test set of loss: 15.56 and RMSE = 3.95.This is much lower than the base level MSE(test) = 7065.44 and RMSE(test) = 84.05.

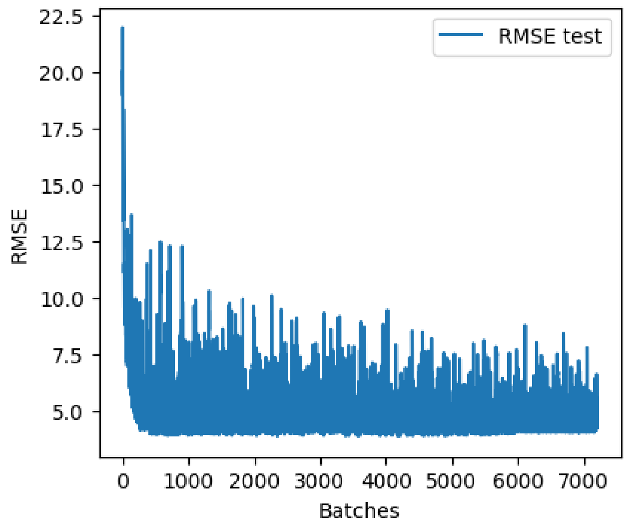
Following are the resultant RMSE plots for this model:

1. Below RMSE vs Epochs plot shows that the model's RMSE is settling down to optimum level after around 300 epochs. Test RMSE is oscillating more than Train RMSE.

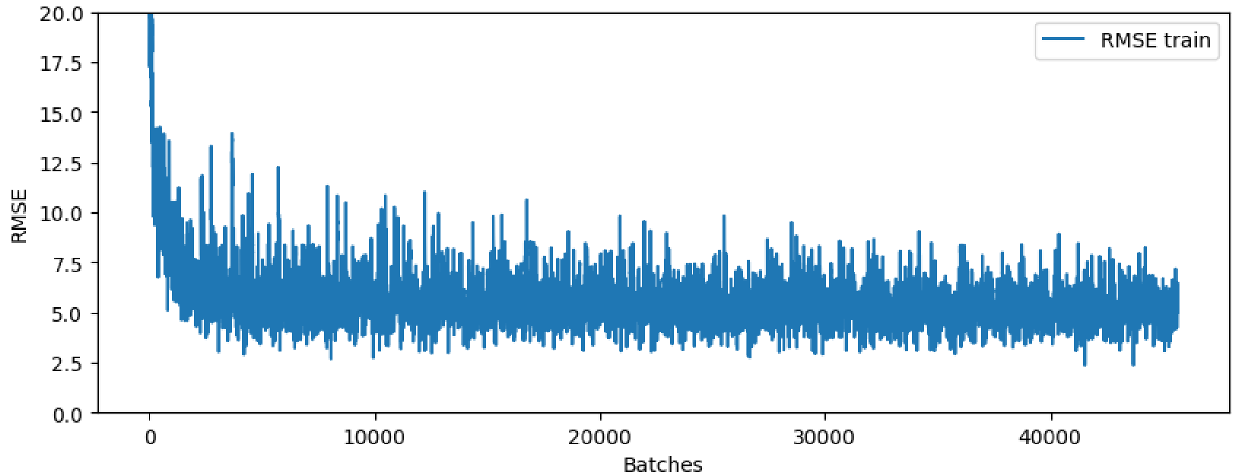


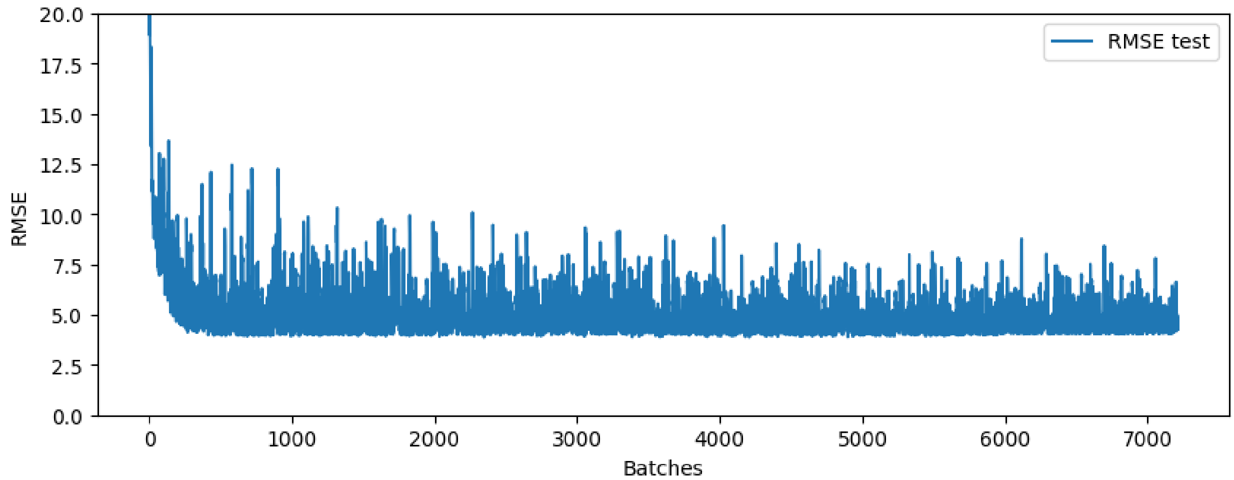
2. Following RMSE vs Batches plots, similar to our arbitrary base model, cannot be compared at different scales.





3. Following is the scaled plot of test and train RMSE vs Batches. It shows that RMSE is settling down to optimum level in first few batches and after that oscillating around that level.

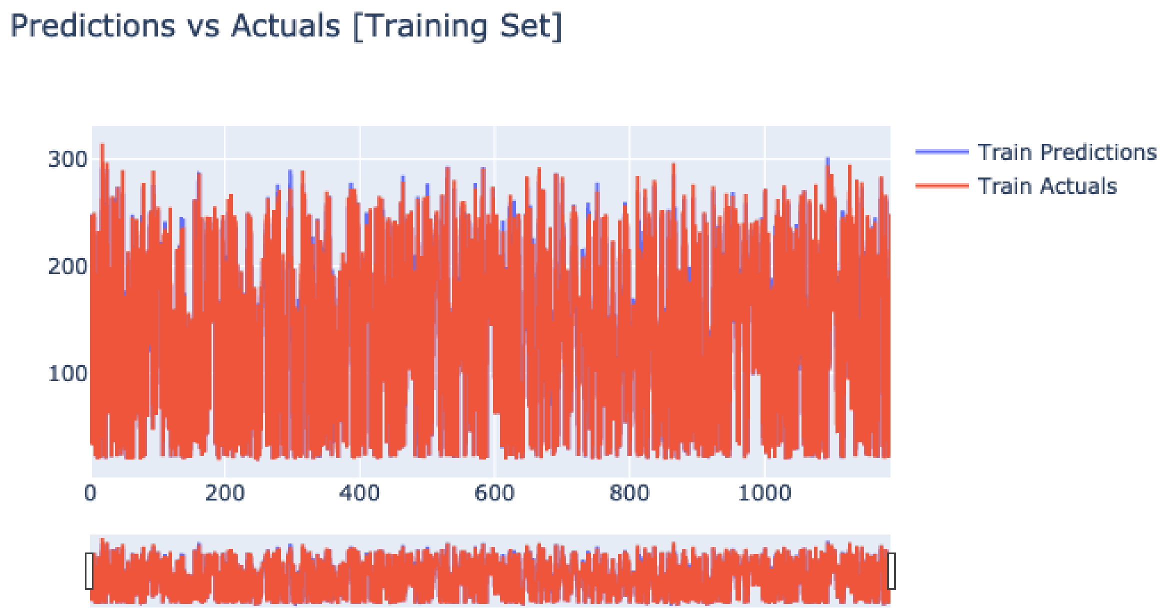




5.3. Plot on the same graph the true values TARGt = S(t+1) and the predicted values Zt . Comments.

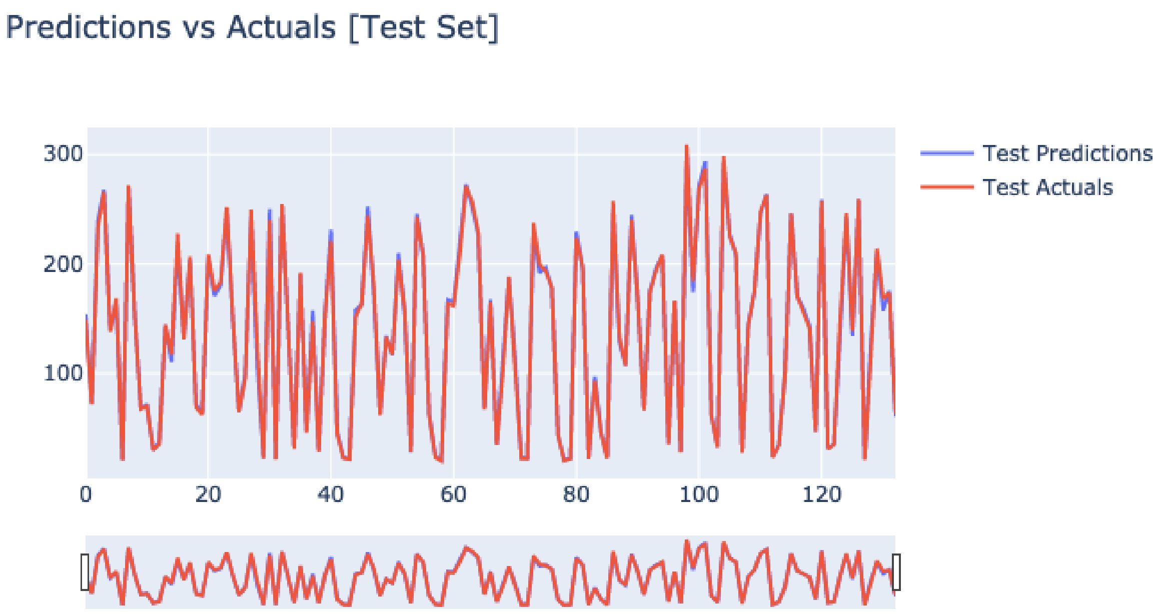
Answer:

Following plot shows True Targt vs predicted Zt for train data. As the number of observations are high, we cannot see the two lines distinctly.



For better view, I have changed the scale of plot to see last 100 observations out of 1188 cases. This plot shows that predicted train is very close to true values, except at extreme values.

Below plot shows that test data predicted and actual values. Similar to previous plot, predicted value is closely following true values. But it is not accurate at detecting sudden rise or fall in the stock values.



5.4. Compute the Mean Relative Errors of Prediction MREP on the training set :

MREP= average (| Zt - TARG t | / TARGt) over all cases in the Training set

Compute similarly MREP on the test set. Comments .

Answer:

MREP for train data is 2.25%. It implies that the model is able to predict stock price with a deviation of about 2.25% on an average, which is a very low bias.

The MREP_Train in % is 2.25 with a 95% confidence interval of [1.4, 3.09]
The MREP_Test in % is 2.11 with a 95% confidence interval of [0, 4.56]

As the confidence intervals of train and test MREPs are overlapping, there is no statistically significant difference between train and test performance. But wide range in confidence interval for test compared to train indicates that there is slightly higher uncertainty in prediction, which is expected as the model has not used test data during training stage.

From the above results, we can say that the model has been able to reach a scenario of low bias and low variance for the given dataset.

Practical impact:

From the Test actual vs predicted figure and MREP % of 2.11%, we can say that the model over-estimates the peaks and crests in the stock price more often.

6. Denote NOD1 NOD2 ... NODk the hidden neurons . For j= 1...k, compute and display the mean activity Yj of NODj over all cases in the Training set. Display all the weights W1 ... Wk linking the neurons NOD1 ... NODk to the output node.

For each hidden NODj compute IMPj = Wj Yj = *average impact* of NODj on the prediction Zt. Display these k impacts and comment. Identify the hidden neuron NOD* with maximal impact on Zt.

<u>Answer:</u>

Following table shows Mean activity, Weights from hidden layer to output node and impact of each node :

neuron	activity Yj	weights Wj	Impact IMPj
1	518.413	0.259	134.17
2	35.615	-0.298	-10.631
3	9.545	-0.139	-1.324
4	43.855	-0.199	-8.736

Table 4: Mean activity, weights and impact on output node

From above table, we can see that neuron 1(NOD 1) has highest activity level and also maximal impact on output Zt. We can also infer that 1st neuron gives the rough estimate of output and remaining three neurons finetune the output to get closer to true value. Moreover 3rd neuron has comparatively lowest impact. If we have to decrease the size of hidden layer, then we can take 3 neuron size hidden layer to achieve almost similar accuracy of 4 layered neuron.

By seeing the above impact table along with the PCA plot in Q 4.2, we can even use a model with 1 neuron, as neuron 1 can show maximum variance in the data.

7. Denote INP1 INP2 ... INP18 the 18 input neurons. Compute and display the mean activities X1 ... X18 of the 18 input neurons. Display all the weights U1 ... U18 linking the input nodes INP1 ... INP18 to the neuron NOD*. For each input neuron INPs compute Fs= Us Xs which is the *average impact* of input feature "s" on the key hidden neuron NOD*.

Identify the 5 input features with the largest impact on NOD*. Comments.

<u>Answer:</u>

Mean activities, Xj are :

index	Xj_mean_activity
MA5	135.7
MA10	135.227
MA20	134.27
NVDA(t)	136.136
NVDA(t-1)	135.904
NVDA(t-2)	135.702
NVDA(t-3)	135.473
NVDA(t-4)	135.285
NVDA(t-5)	135.141
NVDA(t-6)	134.932
NVDA(t-7)	134.777
NVDA(t-8)	134.545
NVDA(t-9)	134.376
NVDA(t-10)	134.178
NVDA(t-11)	134.009
NVDA(t-12)	133.787
NVDA(t-13)	133.527
NVDA(t-14)	133.387

Table 5: Mean activity of input node

Weights matrix linking Input to hidden Node, Uij with right most column showing the name of features :

	HIDDEN LAYER NODES			
index	**1**	**2**	**3**	**4**

0	0.73	0.245	-0.539	-0.179	**MA5**
1	0.386	-0.295	0.048	0.023	**MA10**
2	0.294	-0.197	-0.047	0.556	**MA20**
3	1.276	-1.155	-0.455	-0.447	**NVDA(t)**
4	0.341	-0.018	0.524	-0.544	**NVDA(t-1)**
5	-0.412	0.148	-0.03	-0.271	**NVDA(t-2)**
6	-0.111	0.15	0.345	-0.091	**NVDA(t-3)**
7	0.006	0.501	0.276	-0.273	**NVDA(t-4)**
8	-0.228	0.104	-0.371	0.132	**NVDA(t-5)**
9	0.621	-0.182	-0.083	0.246	**NVDA(t-6)**
10	-0.106	0.557	0.467	0.154	**NVDA(t-7)**
11	0.251	-0.301	-0.263	0.087	**NVDA(t-8)**
12	0.231	0.005	0.095	0.217	**NVDA(t-9)**
13	0.072	0.43	-0.219	0.044	**NVDA(t-10)**
14	0.324	-0.002	-0.182	0.064	**NVDA(t-11)**
15	-0.37	-0.167	0.06	0.071	**NVDA(t-12)**
16	0.603	0.172	0.003	0.13	**NVDA(t-13)**
17	-0.099	-0.001	0.048	0.154	**NVDA(t-14)**

Table 6: Weights matrix of input to hidden layer.

Feature impacts matrix, Fs = Us * Xs is shown in below table :

index	NOD1	NOD2	NOD3	NOD4	
0	99.08	33.28	-73.129	-24.35	**MA5**
1	52.17	-39.831	6.546	3.135	**MA10**
2	39.507	-26.465	-6.283	74.701	**MA20**
3	173.7	-157.26	-61.943	-60.85	**NVDA(t)**
4	46.354	-2.431	71.22	-73.91	**NVDA(t-1)**
5	-55.915	20.089	-4.103	-36.78	**NVDA(t-2)**
6	-14.99	20.256	46.694	-12.35	**NVDA(t-3)**
7	0.754	67.789	37.341	-36.92	**NVDA(t-4)**
8	-30.775	14.077	-50.076	17.807	**NVDA(t-5)**
9	83.762	-24.542	-11.201	33.154	**NVDA(t-6)**
10	-14.264	75.07	62.966	20.74	**NVDA(t-7)**
11	33.783	-40.553	-35.375	11.741	**NVDA(t-8)**
12	31.102	0.648	12.829	29.111	**NVDA(t-9)**
13	9.635	57.636	-29.358	5.927	**NVDA(t-10)**
14	43.408	-0.285	-24.45	8.606	**NVDA(t-11)**
15	-49.452	-22.327	7.967	9.529	**NVDA(t-12)**

16	80.527	22.992	0.338	17.366	**NVDA(t-13)**
17	-13.226	-0.091	6.413	20.489	**NVDA(t-14)**

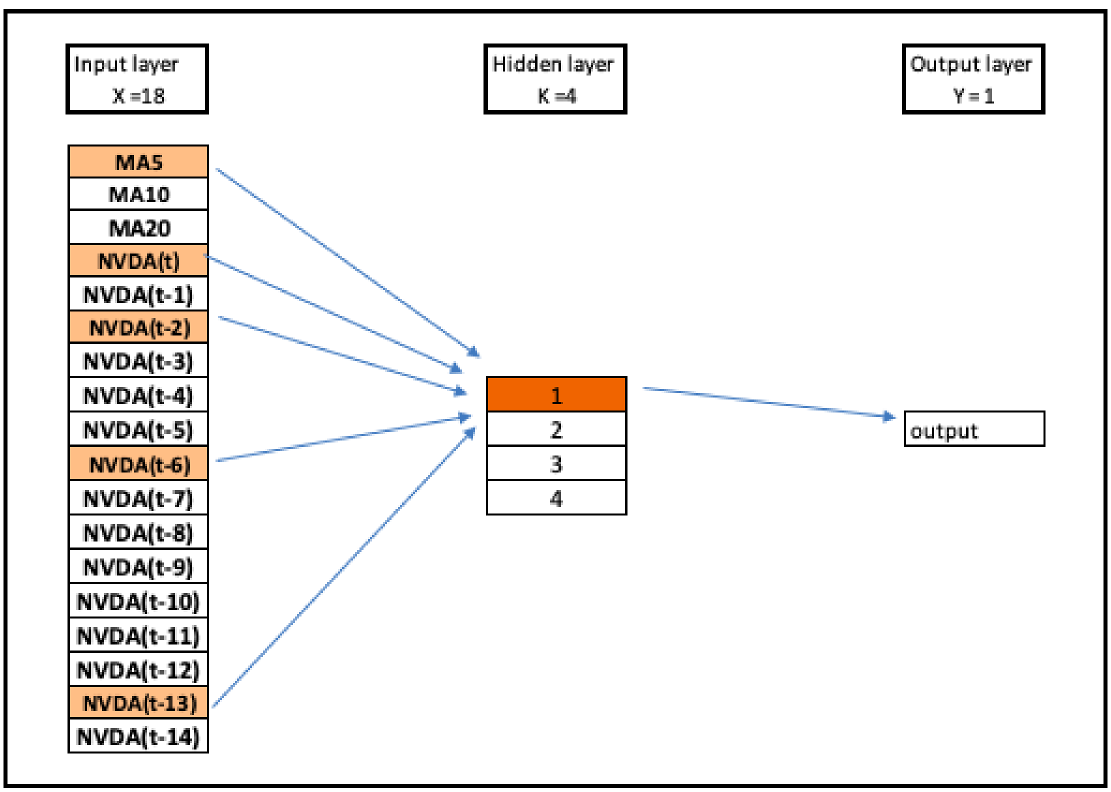
Table 7: Feature impacts matrix of input on hidden layer.

After taking absolute values of all elements of Fs matrix, following table shows top 5 features creating high impact on NOD*(NOD1) node of hidden layer are :

index	NOD1	
3	173.703	**NVDA(t)**
0	99.08	**MA5**
9	83.762	**NVDA(t-6)**
16	80.527	**NVDA(t-13)**
5	55.915	**NVDA(t-2)**

Table 8:Top 5 Features with highest impact of input on hidden layer.

From Question 6, NOD*, node with highest impact, is NOD1. From above table for NOD1, we can say that immediate day before target day has highest impact. By looking at 5 features, we can say that stock price for previous day,2 days before, 7 days before and 14 days before prices are important features along with 5 day moving averages. The following schematic figure shows the neurons and features with highest impact on output.



Schematic figure showing the features and nodes with maximal impact on output

Further analysis:

Maximum possible value of hidden layer size, k can be determined as follows :

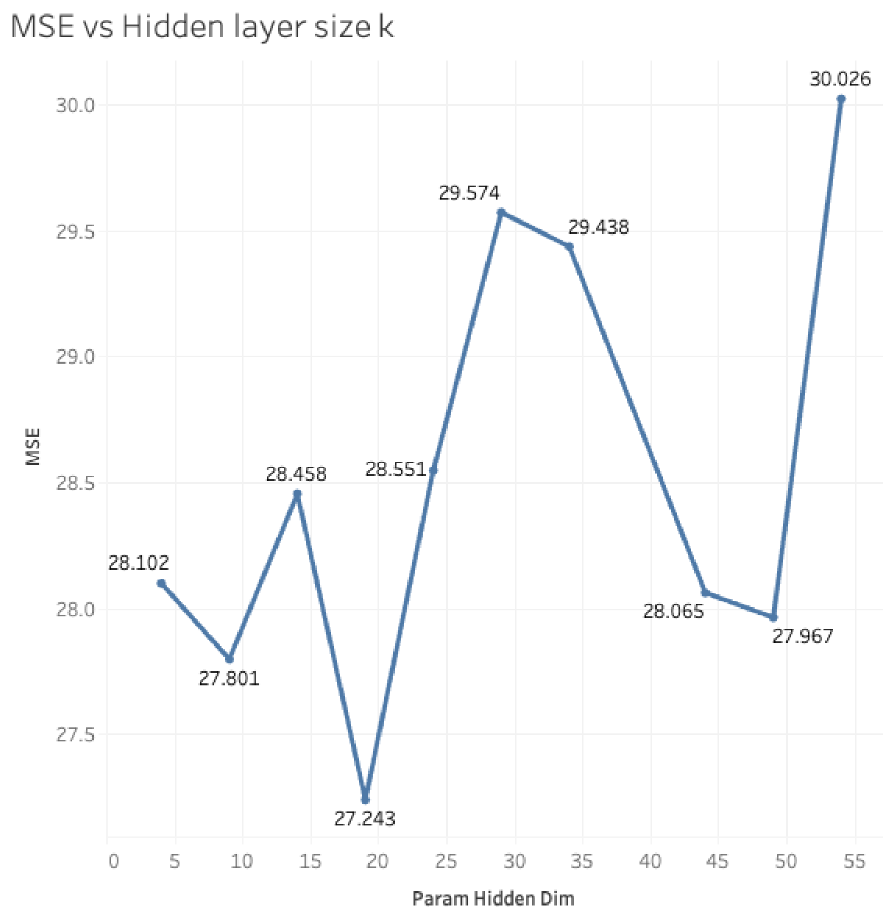
- Total number of unknown parameters in the 3-layer model
  - = weights [18 * k + k * 1] + biases [k + 1]
- Maximum number of constraints = number of cases in train data = 1188

For a robustness ratio of 1, number of unknowns < = number of constraints

By solving above equation, 20k + 1 < = 1188, we can get k < = 59.35

So we can create a hidden layer of size 59 and still maintain a robustness ratio greater than 1 .
So I have done a random search with an early stopping patience of 1000 for a range of k values [1, 4, 9, 14, 19, 24, 29, 34, 39, 44, 49, 54].

Following is the plot showing MSE vs k value. This indicates that MSE is lowest of this model at k = 19. Generally, MSE decreases with increasing k value. But here it is curved pattern. This might be because of short training with low patience and epoch size.

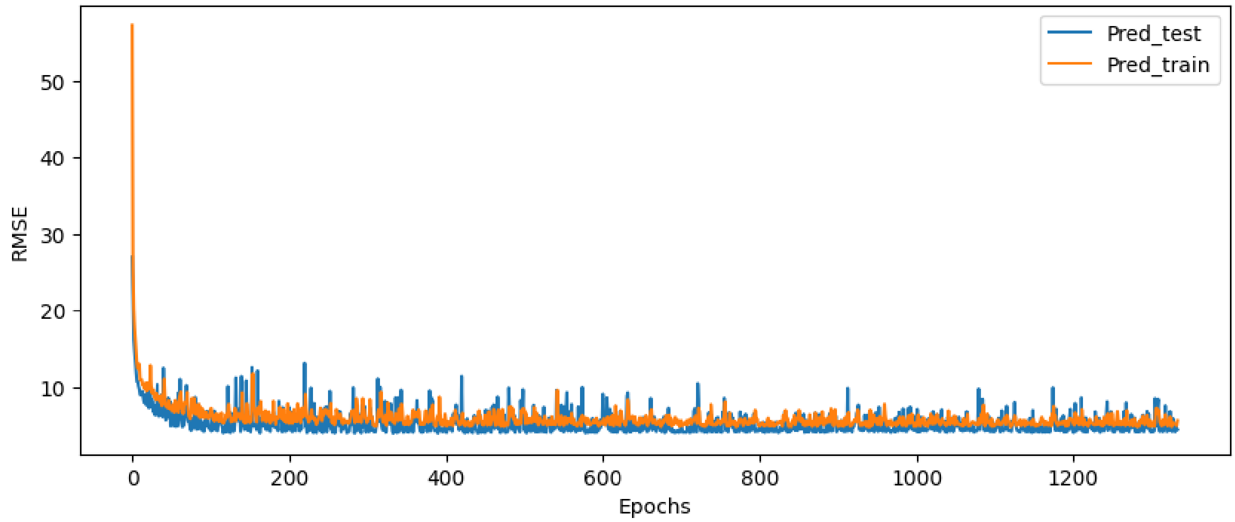


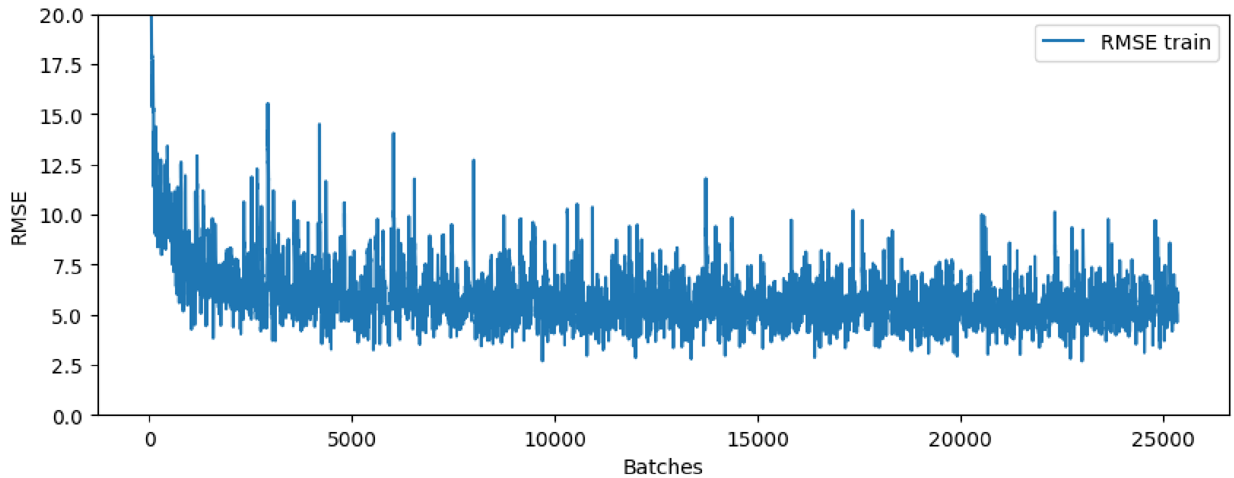
So I have created a new model with k = 19 neurons and trained the model. Following table shows metrics of this model compared to previous models :

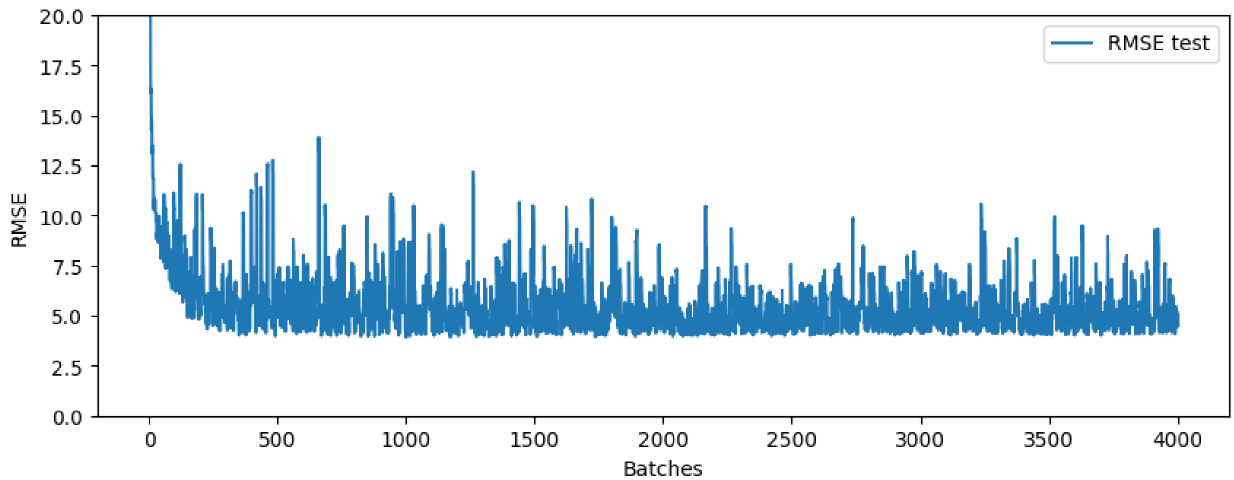
	Base model	**k = 4 model**	**k = 19 model**
Robustness ratio	14.67	14.67	3.12
MSE	7065	15.56	15.79
RMSE	84	3.95	3.97

Table 9: Metrics comparison

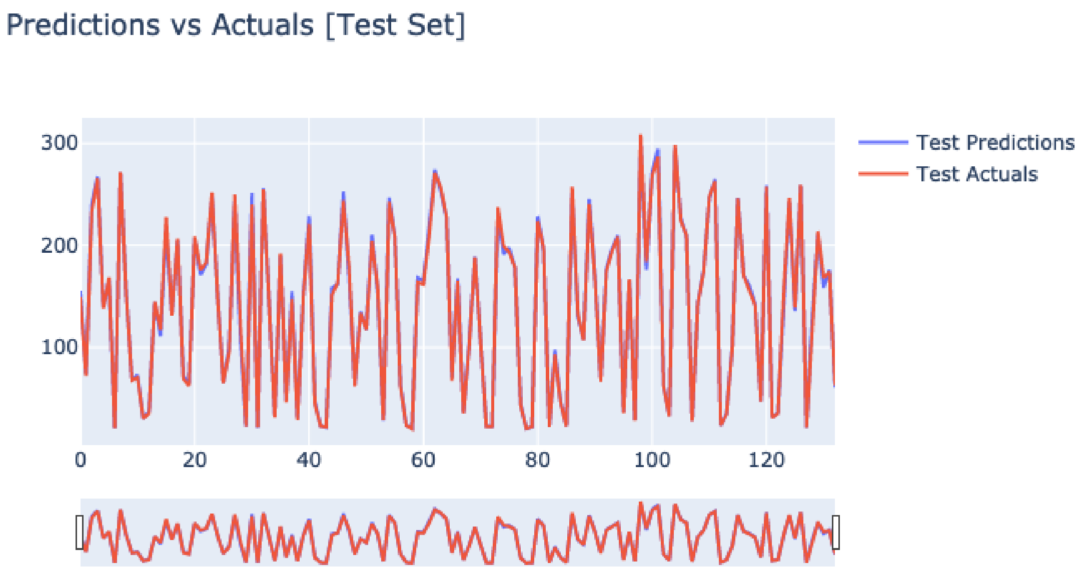
RMSE indicates that 4-neurons model and new model have very close values and former performs slightly better. Following plots show RMSE for new model :



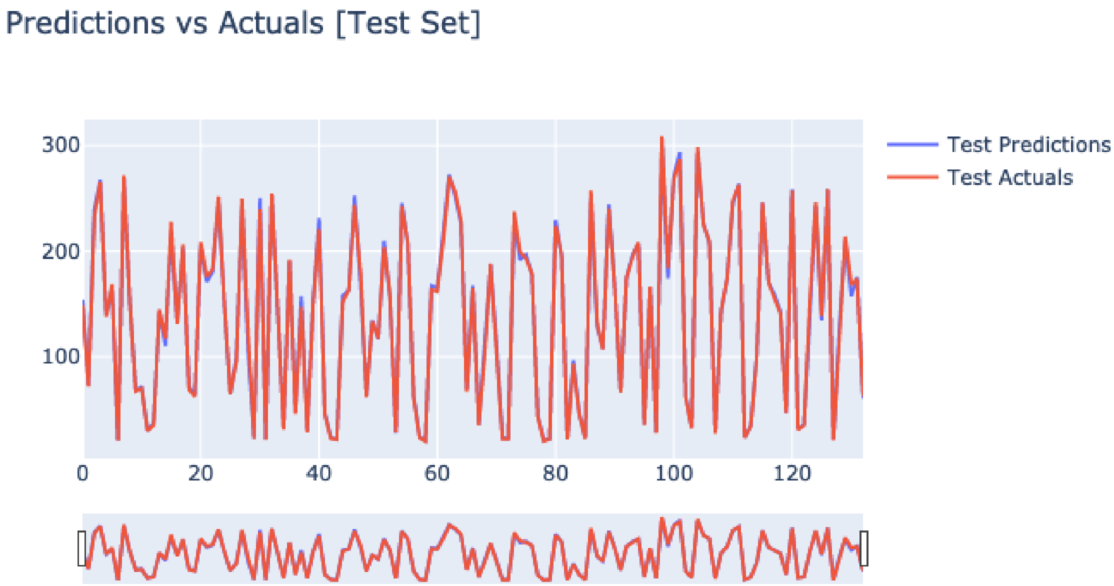




These plots are quite similar to k= 4 model.



Above plot is for k = 19 model and below plot is for k = 4(drawn here for quick comparision). Both plots show that, the model is able to predict non-extreme values accurately but unable to closely follow the extreme values.



Identify the hidden neuron NOD* with maximal impact on Zt ?

Below weights matrix shows that some neurons have very low weightage, for example index 2 neuron. But Mean activity shows that almost 9 out of 19 neurons has 0 mean activity, hence they are redundant.

Weights h to out Wj	
index	**0**
0	-0.45
1	-0.111
2	-0.008
3	-0.091
4	-0.306
5	-0.246
6	-0.528
7	-0.484
8	-0.48
9	0.243
10	-0.316
11	0.082
12	0.419
13	-0.202
14	0.53899997
15	-0.372
16	0.447
17	-0.082
18	-0.149

Mean activity Yj	
index	**0**
0	0
1	21.7
2	7.191
3	1.23
4	0
5	32.425
6	0
7	29.996
8	16.986
9	490.53
10	0
11	206.676
12	0
13	7.703
14	0
15	0
16	0
17	4.542
18	0

Table 10: Weights and mean activity from hidden to output layer

Below table shows the impact of each neuron on output sorted in descending order. It shows that neuron 9 has highest absolute impact followed by 11 and 7 neurons. So similar to 4-neuron model, one node is predicting the broad range and remaining 9 neurons fine tune it.

Impj IMPj		
index	IMPj	ABS(IMPj)
9	119.42	119.42
11	16.998	16.998
7	-14.52	14.52
8	-8.159	8.159
5	-7.989	7.989
1	-2.419	2.419
13	-1.558	1.558
17	-0.371	0.371
3	-0.111	0.111
2	-0.056	0.056
0	0	0
4	0	0
6	0	0
10	0	0
12	0	0
14	0	0
15	0	0
16	0	0
18	0	0

Table 11: Impact of nodes from hidden to output layer

Identify the 5 input features with the largest impact on NOD*. Comments.

Following table shows mean activity of input nodes.

Mean activity Xj	
index	**0**
0	135.7
1	135.227
2	134.27
3	136.136
4	135.904
5	135.702
6	135.473
7	135.285
8	135.141
9	134.932

10	134.777
11	134.545
12	134.376
13	134.178
14	134.009
15	133.787
16	133.527
17	133.387

Table 12: Mean activity of nodes from input to hidden layer

Following table shows the impact after multiplying weights of input to hidden layer for 9th neuron (NOD*) and Mean activities. After sorting the impacts in descending order, we can see that top 5 features are 5-day and 20-day MAs and t-4, t-12, t-8 features have higher impact on the focus node NOD*

Fs feature impact on NOD*			
index	**impact**	**absolute**	
2	69.276	69.276	**MA20**
0	57.792	57.792	**MA5**
7	57.637	57.637	**NVDA(t-4)**
15	47.331	47.331	**NVDA(t-12)**
11	43.737	43.737	**NVDA(t-8)**
3	38.976	38.976	**NVDA(t)**
1	31.994	31.994	**MA10**
16	31.197	31.197	**NVDA(t-13)**
6	30.914	30.914	**NVDA(t-3)**
8	29.443	29.443	**NVDA(t-5)**
13	28.538	28.538	**NVDA(t-10)**
9	-19.202	19.202	**NVDA(t-6)**
5	16.785	16.785	**NVDA(t-2)**
14	-15.817	15.817	**NVDA(t-11)**
17	15.483	15.483	**NVDA(t-14)**
10	5.393	5.393	**NVDA(t-7)**
4	2.919	2.919	**NVDA(t-1)**
12	2.076	2.076	**NVDA(t-9)**

Table 13: Feature Impact on NOD*

Finally, we can say that, k=4 model is an efficient model, compared to k= 19, with a smaller number of parameters and consequently high robustness ratio.

APPENDIX

The code is organized into two files :

A.1 Main code – colab link

https://colab.research.google.com/drive/1wRh-glzDALKrJGneI-_goAYmn2HiQ8mS?usp=sharing

A.2 Tuning module – colab link

https://colab.research.google.com/drive/18D2WtmWDYtEu_0E7YH16usBkbOMsXDQT?usp=sharing

A.1. Main code

```
# -*- coding: utf-8 -*-
"""MATH6373_Final.ipynb

Automatically generated by Colaboratory.

Original file is located at
    https://colab.research.google.com/drive/1wRh-glzDALKrJGneI-_goAYmn2HiQ8mS

# Import libraries
"""

# Commented out IPython magic to ensure Python compatibility.
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import plotly.graph_objects as go
from plotly.offline import plot
import plotly.express as px
import seaborn as sns
# %matplotlib inline

import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.initializers import Constant, RandomNormal, RandomUniform
from tensorflow import keras
```

```
# show nice table in colab
# %load_ext google.colab.data_table
from google.colab import data_table

"""# 1. Prediction Task :
Select one major stock on the US stockmarket. On day "t", let S(t) be the price of this stock at closing time. On each day "t" , we want to predict the future stock price S(t+1) given the last 20 oberved stock prices S(t) , S(t-1), S(t-2), ..., S(t-19)

# Data Set :
 Let t=1, 2, ... N be the days on which the US stock exchange was open during the time period 2014-2015-2016-2017 . Download the time series S(t) for t= 1,2, ..., N
"""

#import data
input_filename =
'https://raw.githubusercontent.com/kishoret04/Deeplearning_6373/master/Final/stocks_final6373_dataset.csv'
df_stocks = pd.read_csv(input_filename)
df_stocks.head()

data_table.DataTable(df_stocks, include_index=False, num_rows_per_page=5)

"""# 2. PreProcessing:
Replace isolated missing values S(t) by the mean of two actual values closest to time t. If there are too many missing values, download another stock .

For 20 ≤t≤N-1 , compute the following three moving averages of the time series S :

MA5(t) = [S(t-4) + S(t-3) + S(t-2) +S(t-1) + S(t) ] /5

MA10(t) = [S(t-9) + S(t-8) + ... + S(t) ] /10

MA20(t) = [S(t-19) + S(t-18) + ... + S(t) ] /20

Plot the 4 curves S(t) , MA5(t), MA10(t), MA20(t), on the same graph
"""
```

```
#missing values
df_stocks.describe().round(3)

count_missing_values = np.sum(df_stocks['NVDA'].isnull())
print('There are {} missing values in Nvidia stocks dataset'.format(count_missing_values))

#creating rolling means with window size 5
MA5_WINDOW = 5
df_stocks['MA5'] =  df_stocks.rolling(window = MA5_WINDOW).mean()
df_stocks

#creating rolling means with window size 10
MA10_WINDOW = 10
df_stocks['MA10'] =  df_stocks['NVDA'].rolling(window = MA10_WINDOW ).mean()
df_stocks

#creating rolling means with window size 20
MA20_WINDOW = 20
df_stocks['MA20'] =  df_stocks['NVDA'].rolling(window = MA20_WINDOW).mean()
df_stocks

fig = px.line(df_stocks, x = 'Date', y='NVDA', title='Nvidia stock values with moving averages')

fig.add_trace(go.Scatter(x= df_stocks['Date'], y= df_stocks['MA5'],
            mode='lines',
            name='MA5'))



fig.add_trace(go.Scatter(x= df_stocks['Date'], y= df_stocks['MA10'],
            mode='lines',
            name='MA10'))



fig.add_trace(go.Scatter(x= df_stocks['Date'], y= df_stocks['MA20'],
            mode='lines',
            name='MA20'))
```

```
fig.update_xaxes(
    rangeslider_visible=True,
    rangeselector=dict(
        buttons=list([
            dict(count=1, label="1m", step="month", stepmode="backward"),
            dict(count=6, label="6m", step="month", stepmode="backward"),
            dict(count=1, label="YTD", step="year", stepmode="todate"),
            dict(count=1, label="1y", step="year", stepmode="backward"),
            dict(step="all")
        ])
    )
)



fig.show()


df_stocks['Date'] = pd.to_datetime(df_stocks['Date'])
df_stocks['Date'].dtypes


#what are the dtypes of the columns
df_stocks.dtypes.value_counts()


print('The shape is {}'.format(df_stocks.shape))


FIG_SIZE = (10,5)
df_stocks.plot(x = 'Date', y = ['NVDA', 'MA5', 'MA10', 'MA10'] ,
               figsize= FIG_SIZE,title = 'Plot of Moving averages and stock price')


"""Statistical description"""


# Histogram


plt.figure(figsize =FIG_SIZE)
sns.distplot(df_stocks['NVDA'], kde= False)
plt.title('Histogram of NVIDIA stock')


"""# 3.1.  Training and Test sets for an MLP predictor :
```

```
On each day t ≥ 20 , the recent past of the series S will be defined as the 1x18 line vector

Vt = [ MA5(t), MA10(t), MA20(t), S(t), S(t-1), S(t-2)..., S(t-13), S(t-14)]

For 20 ≤ t ≤ N-1 the input vector Vt will be the input of our MLP predictor , which will have a single output neuron with
state Zt. This output Zt will be the MLP prediction computed on day t for the target TARGt = S(t+1), which is not
known at time t.

For this prediction task , we have a data set of (N-20) "cases" Case20 Case21 Case22 ... CaseN-1 , indexed by t=
20, 21, .... N-1. Each Caset is described by 18 features = 18 coordinates of vector Vt. The TRUE output to be
predicted at time t is the yet unknown TARGt = S(t+1).
"""

def series_to_supervised(data, n_in=1, n_out=1, dropnan=True):
 """
  Frame a time series as a supervised learning dataset.
  Arguments:
    data: Sequence of observations as a list or NumPy array.
    n_in: Number of lag observations as input (X).
    n_out: Number of observations as output (y).
    dropnan: Boolean whether or not to drop rows with NaN values.
  Returns:
    Pandas DataFrame of series framed for supervised learning.
  """#n_vars = 1 if type(data) is list else data.shape[1]

 variables = list(data.columns)
 df = data.copy(deep = True)
 cols, names = list(), list()
 # input sequence (t-n, ... t-1)
 for i in range(n_in, 0, -1):
  cols.append(df.shift(i))
  names += ['{}(t-{})'.format(j, i) for j in variables]
 # forecast sequence (t, t+1, ... t+n)
 for i in range(0, n_out):
  cols.append(df.shift(-i))
  if i == 0:
```

```
      names += ['{}(t)'.format(j) for j in variables]
    else:
      names += ['{}(t+{})'.format(j, i) for j in variables]
  # put it all together
  agg = pd.concat(cols, axis=1)
  agg.columns = names
  # drop rows with NaN values
  if dropnan:
    agg.dropna(inplace=True)
  return agg

df_modified = series_to_supervised(pd.DataFrame( df_stocks.loc[:,'NVDA']),
                                   n_in=14, n_out=2, dropnan=False)
df_modified.head()

#dropping first 19 rows with NaN values, as they have NaN values in any of the columns
#merging moving average data with stock prices beginning from day 20( index = 19)
df_predcases = pd.merge(df_modified.loc[19:], df_stocks.loc[19:,'MA5':],
                        left_index= True, right_index= True )

df_predcases.head()

#Reordering columns
columns = df_predcases.columns.tolist()
columns = columns[::-1]
columns_MA = columns[0:3]

columns = columns_MA[::-1] + columns[3:]
df_predcases = df_predcases[columns]
df_predcases.head()

#reframing predcases by renaming columns and dropping last row

df_predcases['TARGt'] = df_predcases['NVDA(t+1)']
df_predcases.drop(columns='NVDA(t+1)', inplace= True)

#dropping N-1 row
```

```
df_predcases.drop(axis = 0, labels = 1340, inplace= True)
df_predcases.reset_index( drop = True, inplace= True)
df_predcases.head()

df_predcases.describe().round(3)

"""3.2. The data set of (N-20) cases for MLP prediction learning is denoted PredCases = { all pairs (Vt, TARGt) with t=
20, 21, .... N-1 }

Randomly Split the set PredCases, with 90% cases in the training set PredTRAIN, and 10 % cases in the test set
PredTEST
"""

from sklearn.model_selection import train_test_split

# fix random seed for reproducibility
SEED = 2020

X_train, X_test, y_train, y_test = train_test_split( df_predcases.iloc[:,:-1] , df_predcases.iloc[:,-1],
                                                    test_size=0.1, random_state=SEED)

print('Train X: {}\tTrain y = {}'.format(X_train.shape, y_train.shape))
print('Test X: {}\tTest y = {}'.format(X_test.shape, y_test.shape))

df_pred_train = pd.merge(X_train,y_train, left_index=True,right_index=True).reset_index(drop=True)
df_pred_test = pd.merge(X_test,y_test, left_index=True,right_index=True).reset_index(drop = True)

#downloading file
from pandas import ExcelWriter

#write to excel dataset
filepath = 'processed_data.xlsx'

with ExcelWriter(filepath) as writer:
    df_stocks.to_excel(writer,sheet_name = 'total_data' )
    df_pred_train.to_excel(writer,sheet_name = 'df_pred_train' )
    df_pred_test.to_excel(writer,sheet_name = 'df_pred_test' )
```

```
    writer.save()

df_pred_train.head()

df_pred_test.head()

"""# 4. MLP predictor :
Our MLP predictor (MLPpred) will have the simple 3 layers architecture INPUT ==> HiddenLayer K ==> OUTPUT
with dim(INPUT) = 18 , dim(OUTPUT) = 1
dim(K) = k to be selected below.

For each training input Vt we want the MLP output Zt to be close to TARGt= S(t+1).

Implement PCA on the set of all input vectors Vt , with t= 20 ,21, ..., N . Determine the number k of principal
components which preserves 95% of the variance (see HW3) and fix dim(K)= k.
"""

df_predcases.iloc[:,:-1]

Vt = df_predcases.iloc[:,:-1]
corr = Vt.corr()
corr.round(3)

eigs, eig_vectors = np.linalg.eig(corr)

ratio = np.cumsum(np.real(eigs))/np.sum(np.real(eigs))
print('min: {}\t max: {}'.format(min(ratio),max(ratio)))

# k value
threshold=0.95
h = np.min(np.nonzero(ratio>threshold))
print('h = ',h,': ',ratio[h])

# k value
threshold=0.999
h = np.min(np.nonzero(ratio>threshold))
print('h = ',h,': ',ratio[h])
```

```
plt.figure(figsize=(10, 4), dpi=150)
plt.plot(ratio)
plt.xlabel('j')
plt.ylabel('Rj')
plt.vlines(x=4,ymin= min(ratio), ymax= max(ratio),linestyles='--', color = 'r')
plt.hlines(y=ratio[h], xmin=0,xmax=17,linestyles='--', color = 'r')

plt.text(x = h+1, y = min(ratio), s = 'h= '+ str(h), fontsize=15 )
plt.text(x = 16, y = ratio[h], s = round(ratio[h],3), fontsize=15 )
plt.savefig('pca.png',dpi=200)
plt.show()

"""Compute the number w of weights and thresholds in this MLP, and compare w to the number of informations
provided by the training set.

dim(INPUT) = 18 , k = 4, dim(OUTPUT) = 1

total weights in 3 layered MLP = weights [ 18 * 4 + 4 * 1 ] + biases [ 4 + 1 ]

total parameters = 81

total informations = 1188 * 1

robustness ratio =  1188/81 = 14.66

# 5. Training of the MLP predictor:

5.1. Implement an automatic training on the training set PredTRAIN, with the options :

RELU response,
Loss = "MSE",
Stochastic Gradient Descent or ADAM,
Batch Learning,
Early Stopping
"""
```

```
df_pred_train.iloc[:,:-1]

#data transformation to numpy arrays
pred_train_X = df_pred_train.iloc[:,:-1].values
pred_train_y = df_pred_train.iloc[:,-1].values
pred_test_X = df_pred_test.iloc[:, :-1].values
pred_test_y = df_pred_test.iloc[:,-1].values

print('shapes of train data: X: ', pred_train_X.shape, '\t y: ',pred_train_y.shape)
print('shapes of test data: X: ', pred_test_X.shape, '\t y: ',pred_test_y.shape)

"""# Base level model"""

#Dimensions for the MLP

MLP_INPUT_DIM = df_pred_train.shape[1] - 1
MLP_OUTPUT_DIM = 1
MLP_HIDDEN_LAYER_SIZE = 4

#default parameters
#model parameters
BIA_INI_H = 10
BIA_INI_O = 10

#fitting parameters
MLP_LEARNING_RATE = 5e-5
MLP_DECAY_RATE = 1e-5
MLP_EPOCH_SIZE = 40000
PATIENCE = 1000
MLP_BATCH_SIZE = 32

selected_optimizer = keras.optimizers.SGD(learning_rate = MLP_LEARNING_RATE, decay= MLP_DECAY_RATE)

#Adam optimizer
#new hyperparameter β, simply called the momentum, which must be set between 0 (high friction) and 1 (no friction).
#A typical momentum value is 0.9.
# selected_optimizer = keras.optimizers.Adam(learning_rate = MLP_LEARNING_RATE )
```

```python
"""**For stochastic gradient descent**

learning_rate = initial_lrate * (1 / (1 + decay_rate * epoch))
"""

MLPpred_model = keras.Sequential()
MLPpred_model.add(keras.layers.Dense(units = MLP_HIDDEN_LAYER_SIZE,
                             activation='relu',
                             input_dim = MLP_INPUT_DIM,
                             bias_initializer = keras.initializers.Constant(
                                value = BIA_INI_H) ))

MLPpred_model.add(keras.layers.Dense(units = MLP_OUTPUT_DIM,
                             activation='relu',
                             bias_initializer= keras.initializers.Constant(
                                value = BIA_INI_O)))

MLPpred_model.summary()

#creation of root directory for tensor board
import os
root_logdir = os.path.join(os.curdir, "my_logs")

def get_run_logdir():
  import time
  run_id = time.strftime("run_%Y_%m_%d-%H_%M_%S")
  return os.path.join(root_logdir, run_id)

run_logdir = get_run_logdir()# e.g., './my_logs/run_2019_06_07-15_15_22'
run_logdir

# the customized callback to record losses after each batch

class mlpMyHistory(keras.callbacks.Callback):
  def on_train_begin(self, logs={}):
    self.MSEtrain = []
```

```
        self.MSEtest = []

    def on_train_batch_end(self, batch, logs={}):
        self.MSEtrain.append(logs['loss'])

    def on_test_batch_end(self, batch, logs={}):
        self.MSEtest.append(logs['loss'])


#object created for custom class
batch_monitor_cb = mlpMyHistory()

#callbacks
earlystopping_cb = keras.callbacks.EarlyStopping(monitor='val_loss',
                                                 mode='min', verbose=1,
                                                 patience= PATIENCE,
                                                 restore_best_weights=True)

tensorboard_cb = tf.keras.callbacks.TensorBoard(run_logdir)

callbacks_list = [batch_monitor_cb, earlystopping_cb, tensorboard_cb ]

MLPpred_model.compile(optimizer= selected_optimizer,
                      loss='mean_squared_error')

Monitor2 = MLPpred_model.fit( x = pred_train_X,
                        y = pred_train_y,
                        batch_size = MLP_BATCH_SIZE,
                        epochs = MLP_EPOCH_SIZE,
                        callbacks = callbacks_list,
                        validation_data = (pred_test_X,
                                           pred_test_y),
                        verbose = 1)

# After training, access MSE(AutoTrain) and MSE(AutoTest) through MyMonitor.MSEtrain and MyMonitor.MSEtest.

# Commented out IPython magic to ensure Python compatibility.
```

```
# %load_ext tensorboard
# %tensorboard --logdir=./my_logs --port=6006

#Plotting RMSE per epoch
plt.figure(figsize=(10, 4), dpi=100)
plt.plot( np.sqrt( Monitor2.history['val_loss']), label='Pred_test')
plt.plot( np.sqrt( Monitor2.history['loss']), label='Pred_train')
plt.xlabel('Epochs')
plt.ylabel('RMSE')
#plt.ylim([0,0.1])
plt.legend()
# plt.savefig('MSE vs epoch.png')

plt.show()

plt.figure(figsize=(10, 4), dpi=100)
plt.subplot(1,2,1)
plt.plot(np.sqrt(batch_monitor_cb.MSEtrain), label='RMSE train')
plt.xlabel('Batches')
plt.ylabel('RMSE')
#plt.ylim([0,0.1])
plt.legend()
plt.subplot(1,2,2)
plt.plot( np.sqrt(batch_monitor_cb.MSEtest), label='RMSE test')
plt.xlabel('Batches')
plt.ylabel('RMSE')
#plt.ylim([0,0.1])
plt.legend()
# plt.savefig('RMSE vs batches.png')

plt.show()

common_x_range = [ -20,50000]
common_y_range = [ 40,150]



plt.figure(figsize=(10, 8), dpi=100)
```

```
plt.subplot(2,1,1)
plt.plot(np.sqrt(batch_monitor_cb.MSEtrain), label='RMSE train')
plt.xlabel('Batches')
plt.ylabel('RMSE')
plt.xlim(common_x_range)
plt.ylim( common_y_range)
plt.legend()
plt.subplot(2,1,2)
plt.plot( np.sqrt(batch_monitor_cb.MSEtest), label='RMSE test')
plt.xlabel('Batches')
plt.ylabel('RMSE')
plt.xlim(common_x_range)
plt.ylim(common_y_range)
plt.legend()
# plt.savefig('RMSE vs batches.png')

plt.show()

#def for plotting
# def plotly_rmse(data_plot, title = '', update_y = []):
# Initialize figure
fig = go.Figure()
# Add Traces
for i in data_plot.keys():
  fig.add_trace(go.Scatter(x=list(range(0, len(data_plot[i]))),
                           y= np.sqrt(batch_monitor_cb.MSEtrain),
                           name=i,
                           marker = dict(size = 10)))
plt.plot(np.sqrt(batch_monitor_cb.MSEtrain), label='RMSE train')
plt.xlabel('Batches')
plt.ylabel('RMSE')
#plt.ylim([0,0.1])
plt.legend()

#set range
fig.update_xaxes(rangeslider_visible=True)
```

```
# Set title
fig.update_layout(title_text=title)

fig.show()
plot(fig, auto_open = True)

"""# *Tuning parameters*"""

#Dimensions for the MLP

MLP_INPUT_DIM = df_pred_train.shape[1] - 1
MLP_OUTPUT_DIM = 1
MLP_HIDDEN_LAYER_SIZE = 4#h

#Optimized Parameters found in other program

#tuned parameters
BIA_INI_H = 20
BIA_INI_O = 40
MLP_LEARNING_RATE = 1e-2
# MLP_DECAY_RATE = 1e-5
MLP_EPOCH_SIZE = 40000
PATIENCE = 1000
MLP_BATCH_SIZE = 64#32

#Dimensions were stated before the optimization

# selected_optim = keras.optimizers.SGD(learning_rate = MLP_LEARNING_RATE, decay= MLP_DECAY_RATE)

#Adam optimizer
#new hyperparameter β, simply called the momentum, which must be set between 0 (high friction) and 1 (no friction).
#A typical momentum value is 0.9.
selected_optimizer = keras.optimizers.Adam(learning_rate= MLP_LEARNING_RATE)

MLPpred_model = keras.Sequential()
MLPpred_model.add(keras.layers.Dense(units = MLP_HIDDEN_LAYER_SIZE,
                          activation='relu',
```

```
                                input_dim = MLP_INPUT_DIM,
                                bias_initializer = keras.initializers.Constant(
                                  value = BIA_INI_H) , ))

MLPpred_model.add(keras.layers.Dense(units = MLP_OUTPUT_DIM,
                                activation='relu',
                                bias_initializer= keras.initializers.Constant(
                                  value = BIA_INI_O)))

MLPpred_model.summary()

#creation of root directory for tensor board
import os
root_logdir = os.path.join(os.curdir, "my_logs")

def get_run_logdir():
  import time
  run_id = time.strftime("run_%Y_%m_%d-%H_%M_%S")
  return os.path.join(root_logdir, run_id)

run_logdir = get_run_logdir()# e.g., './my_logs/run_2019_06_07-15_15_22'
run_logdir

# the customized callback to record losses after each batch

class mlpMyHistory(keras.callbacks.Callback):
  def on_train_begin(self, logs={}):
    self.MSEtrain = []
    self.MSEtest = []

  def on_train_batch_end(self, batch, logs={}):
    self.MSEtrain.append(logs['loss'])

  def on_test_batch_end(self, batch, logs={}):
    self.MSEtest.append(logs['loss'])
```

```
#object created for custom class
batch_monitor_cb = mlpMyHistory()

#callbacks
earlystopping_cb = keras.callbacks.EarlyStopping(monitor='val_loss',
                                                 mode='min', verbose=1,
                                                 patience= PATIENCE,
                                                 restore_best_weights=True)

tensorboard_cb = tf.keras.callbacks.TensorBoard(run_logdir)

callbacks_list = [batch_monitor_cb, earlystopping_cb, tensorboard_cb ]

MLPpred_model.compile(optimizer= selected_optimizer,
                      loss='mean_squared_error')

Monitor2 = MLPpred_model.fit( x = pred_train_X,
                      y = pred_train_y,
                      batch_size = MLP_BATCH_SIZE,
                      epochs = MLP_EPOCH_SIZE,
                      callbacks = callbacks_list,
                      validation_data = (pred_test_X,
                                         pred_test_y),
                      verbose = 1)

# After training, access MSE(AutoTrain) and MSE(AutoTest) through MyMonitor.MSEtrain and MyMonitor.MSEtest.

# Commented out IPython magic to ensure Python compatibility.
# %reload_ext tensorboard
# %tensorboard --logdir=./my_logs --port=6006

MLPpred_model.evaluate(pred_test_X, pred_test_y)

"""# 5.2. Let RMSE be the root mean squared error √MSE.

Plot the evolution of RMSE versus the number of
batches (one curve for the training set and one for the test set) .
```

Compare these two curves.
"""

```
#Plotting RMSE per epoch
plt.figure(figsize= FIG_SIZE, dpi=100)
plt.plot( np.sqrt( Monitor2.history['val_loss']), label='Pred_test')
plt.plot( np.sqrt( Monitor2.history['loss']), label='Pred_train')
plt.xlabel('Epochs')
plt.ylabel('RMSE')
#plt.ylim([0,0.1])
plt.legend()
# plt.savefig('MSE vs epoch.png')

plt.show()

plt.figure(figsize=(10, 4), dpi=100)
plt.subplot(1,2,1)
plt.plot(np.sqrt(batch_monitor_cb.MSEtrain), label='RMSE train')
plt.xlabel('Batches')
plt.ylabel('RMSE')
#plt.ylim([0,0.1])
plt.legend()
plt.subplot(1,2,2)
plt.plot( np.sqrt(batch_monitor_cb.MSEtest), label='RMSE test')
plt.xlabel('Batches')
plt.ylabel('RMSE')
#plt.ylim([0,0.1])
plt.legend()
# plt.savefig('RMSE vs batches.png')

plt.show()

# common_x_range = [ -1000,50000]
common_y_range = [ 0,20]
```

```
plt.figure(figsize=(10, 8), dpi=100)
plt.subplot(2,1,1)
plt.plot(np.sqrt(batch_monitor_cb.MSEtrain), label='RMSE train')
plt.xlabel('Batches')
plt.ylabel('RMSE')
# plt.xlim(common_x_range)
plt.ylim( common_y_range)
plt.legend()
plt.subplot(2,1,2)
plt.plot( np.sqrt(batch_monitor_cb.MSEtest), label='RMSE test')
plt.xlabel('Batches')
plt.ylabel('RMSE')
# plt.xlim(common_x_range)
plt.ylim(common_y_range)
plt.legend()
# plt.savefig('RMSE vs batches.png')

plt.show()

#save the model
# Save the entire model as a SavedModel.
!mkdir -p saved_model
MLPpred_model.save('saved_model/MLPpred_model_best')

"""5.3. Plot on the same graph the true values TARGt = S(t+1) and the predicted values Zt . Comments."""

#def for plotting
def plot_mse(data_plot, title = '', update_y = []):
 # Initialize figure
 fig = go.Figure()
 # Add Traces
 for i in data_plot.keys():
  fig.add_trace(go.Scatter(x=list(range(0, len(data_plot[i]))),
                           y=data_plot[i],
                           name=i,
                           marker = dict(size = 10)))
 if len(update_y) > 0:
```

```
    fig.update_yaxes(range=update_y)

  #set range
  fig.update_xaxes(rangeslider_visible=True)

  # Set title
  fig.update_layout(title_text=title)

  fig.show()
  plot(fig, auto_open = True)

pred_train_X.shape

#new_train_X.shape
Zt_train  = MLPpred_model.predict(pred_train_X)
Zt_train.shape

#Plot the training set comparisons
data_plot = {'Train Predictions' : Zt_train.reshape(Zt_train.shape[0]),
           'Train Actuals' : pred_train_y.reshape(pred_train_y.shape[0])}
plot_mse(data_plot, title = 'Predictions vs Actuals [Training Set]')

Zt_test  = MLPpred_model.predict(pred_test_X)
Zt_test.shape

#Plot the training set comparisons
data_plot = {'Test Predictions' : Zt_test.reshape(Zt_test.shape[0]),
           'Test Actuals' : pred_test_y.reshape(pred_test_y.shape[0])}
plot_mse(data_plot, title = 'Predictions vs Actuals [Test Set]')

"""5.4. Compute the Mean Relative Errors of Prediction MREP on the training set :

MREP= average ( | Zt - TARG t | / TARGt) over all cases in the Training set

Compute similarly MREP on the test set. Comments .
"""
```

```python
(Zt_train.flatten() - pred_train_y)/pred_train_y

#MEAN RELATIVE ERROR FOR TRAIN
MREP_Train = np.mean(np.abs(Zt_train.flatten() - pred_train_y)/pred_train_y)
MREP_Train

#MEAN RELATIVE ERROR FOR TEST
MREP_Test = np.mean(np.abs(Zt_test.flatten() - pred_test_y)/pred_test_y)
MREP_Test

df_pred_train.shape[0]

import math
def err_est_element(term,size,return_sigma = False):
  sigma = np.around(math.sqrt(term*(100-term)/size),2)

  #for 95% confidence level
  Z_VAL = 1.96
  limit_lower = np.around((term - Z_VAL*sigma),2)
  if limit_lower < 0:
    #print('before if:',limit_lower)
    limit_lower = 0
    #print('after if:',limit_lower)

  limit_upper = np.around((term + Z_VAL*sigma),2)
  if limit_upper > 100:
    #print('before if:',limit_upper)
    limit_upper = 100
    #print('after if:',limit_upper)

  conf_int = [limit_lower,limit_upper]

  if return_sigma:
    return sigma
  else:
    return str(conf_int)
```

```
#comparision
def compare_MREP(MREP_Train, MREP_Test):

  train_size = df_pred_train.shape[0]
  test_size = df_pred_test.shape[0]

  CI_MREP_Train = err_est_element(MREP_Train*100 , train_size)
  print('The {0} is {1} with a 95% confidence interval of {2}'.
        format('MREP_Train in %', round(MREP_Train*100,2) , CI_MREP_Train))

  CI_MREP_Test = err_est_element(MREP_Test*100 , test_size)
  print('The {0} is {1} with a 95% confidence interval of {2}'.
        format('MREP_Test in %', round(MREP_Test*100,2) , CI_MREP_Test))

compare_MREP(MREP_Train,MREP_Test)

"""6. Denote NOD1 NOD2 ... NODk the hidden neurons . For j= 1...k, compute and display the mean activity Yj of NODj over all cases in the Training set. Display all the weights W1 ... Wk linking the neurons NOD1 ... NODk to the output node.
For each hidden NODj compute IMPj = Wj Yj = average impact of NODj on the prediction Zt. Display these k impacts and comment. Identify the hidden neuron NOD* with maximal impact on Zt
"""

W_weights_h_to_out = MLPpred_model.layers[1].get_weights()
W_weights_h_to_out

pd.DataFrame(np.around(W_weights_h_to_out[0],3))

#get output from a layer
from keras import backend as K

# with a Sequential model
get_hidden_layer_output = K.function([MLPpred_model.layers[0].input],
                                  [MLPpred_model.layers[0].output])
layer_output = get_hidden_layer_output(pred_train_X)[0]
np.shape(layer_output)
```

```
Yj_mean_activity = np.mean(layer_output,axis = 0)
pd.DataFrame( (np.around(Yj_mean_activity , 3)) )

IMPj_impact = W_weights_h_to_out[0].flatten()*Yj_mean_activity
pd.DataFrame( np.around(IMPj_impact, 3) )

"""7. Denote INP1 INP2 ... INP18 the 18 input neurons. Compute and display the mean activities X1 ... X18 of the 18 input neurons. Display all the weights U1 ... U18 linking the input nodes INP1 ... INP18 to the neuron NOD*. For each input neuron INPs compute Fs= Us Xs which is the average impact of input feature "s" on the key hidden neuron NOD*. Identify the 5 input features with the largest impact on NOD*. Comments."""

# Mean activities of input layer
Xj_mean_activity = np.mean(pred_train_X, axis = 0)
pd.DataFrame(Xj_mean_activity.round(3))

Uj_weights_in_to_hidden = MLPpred_model.layers[0].get_weights()
Uj_weights_in_to_hidden

pd.DataFrame( np.around(Uj_weights_in_to_hidden[0], 3 ) )

Uj_weights_in_to_hidden[0][:,2] *Xj_mean_activity

Uj_weights_in_to_hidden[0].shape

x = pd.DataFrame()

def cal_mean_featureimpact(Us, Xs):
  Fs = pd.DataFrame(np.zeros_like(Us))
  for i in range(Us.shape[1]):
    Fs.loc[:,i] = Us[:,i] * Xs

  return Fs



Fs_feature_impact = cal_mean_featureimpact(Uj_weights_in_to_hidden[0], Xj_mean_activity)

np.around(Fs_feature_impact,3)
```

```
"""## K= 19 Model"""



#Dimensions for the MLP

MLP_INPUT_DIM = df_pred_train.shape[1] - 1
MLP_OUTPUT_DIM = 1
MLP_HIDDEN_LAYER_SIZE = hidden_dim_list[0]

#default parameters
#model parameters
BIA_INI_H = bias_init_hidden[0]
BIA_INI_O = bias_init_out[0]
KERNEL_INIT = weight_init_mode[0]

#fitting parameters
MLP_LEARNING_RATE = lr_list[0]#5e-5
MLP_EPOCH_SIZE = epochs[0]
PATIENCE = 1000
MLP_BATCH_SIZE = batches[0]



selected_optimizer = keras.optimizers.Adam(learning_rate = MLP_LEARNING_RATE )

"""**For stochastic gradient descent**

learning_rate = initial_lrate * (1 / (1 + decay_rate * epoch))
"""


MLPpred_model = keras.Sequential()
MLPpred_model.add(keras.layers.Dense(units = MLP_HIDDEN_LAYER_SIZE,
                    activation='relu',
                    input_dim = MLP_INPUT_DIM,
                    bias_initializer = keras.initializers.Constant(
                      value = BIA_INI_H) ))
```

```
MLPpred_model.add(keras.layers.Dense(units = MLP_OUTPUT_DIM,
                                     activation='relu',
                                     bias_initializer= keras.initializers.Constant(
                                         value = BIA_INI_O)))

MLPpred_model.summary()



MLPpred_model.compile(optimizer= selected_optimizer,
                      loss='mean_squared_error')

Monitor2 = MLPpred_model.fit( x = pred_train_X,
                        y = pred_train_y,
                        batch_size = MLP_BATCH_SIZE,
                        epochs = MLP_EPOCH_SIZE,
                        callbacks = callbacks_list,
                        validation_data = (pred_test_X,
                                           pred_test_y),
                        verbose = 1)

# After training, access MSE(AutoTrain) and MSE(AutoTest) through MyMonitor.MSEtrain and MyMonitor.MSEtest.



# Commented out IPython magic to ensure Python compatibility.
# %load_ext tensorboard
# %tensorboard --logdir=./my_logs --port=6006



plt.figure(figsize=(10, 4), dpi=100)
plt.subplot(1,2,1)
plt.plot(np.sqrt(batch_monitor_cb.MSEtrain), label='RMSE train')
plt.xlabel('Batches')
plt.ylabel('RMSE')
```

```python
#plt.ylim([0,0.1])
plt.legend()
plt.subplot(1,2,2)
plt.plot( np.sqrt(batch_monitor_cb.MSEtest), label='RMSE test')
plt.xlabel('Batches')
plt.ylabel('RMSE')
#plt.ylim([0,0.1])
plt.legend()
# plt.savefig('RMSE vs batches.png')

plt.show()

# common_x_range = [ -20,50000]
common_y_range = [ 0,20]



plt.figure(figsize=(10, 8), dpi=100)
plt.subplot(2,1,1)
plt.plot(np.sqrt(batch_monitor_cb.MSEtrain), label='RMSE train')
plt.xlabel('Batches')
plt.ylabel('RMSE')
# plt.xlim(common_x_range)
plt.ylim( common_y_range)
plt.legend()
plt.subplot(2,1,2)
plt.plot( np.sqrt(batch_monitor_cb.MSEtest), label='RMSE test')
plt.xlabel('Batches')
plt.ylabel('RMSE')
# plt.xlim(common_x_range)
plt.ylim(common_y_range)
plt.legend()
# plt.savefig('RMSE vs batches.png')

plt.show()
```

```
#Plotting RMSE per epoch
plt.figure(figsize= FIG_SIZE, dpi=100)
plt.plot( np.sqrt( Monitor2.history['val_loss']), label='Pred_test')
plt.plot( np.sqrt( Monitor2.history['loss']), label='Pred_train')
plt.xlabel('Epochs')
plt.ylabel('RMSE')
#plt.ylim([0,0.1])
plt.legend()
# plt.savefig('MSE vs epoch.png')

plt.show()

plt.figure(figsize=(10, 4), dpi=100)
plt.subplot(1,2,1)
plt.plot(np.sqrt(batch_monitor_cb.MSEtrain), label='RMSE train')
plt.xlabel('Batches')
plt.ylabel('RMSE')
#plt.ylim([0,0.1])
plt.legend()
plt.subplot(1,2,2)
plt.plot( np.sqrt(batch_monitor_cb.MSEtest), label='RMSE test')
plt.xlabel('Batches')
plt.ylabel('RMSE')
#plt.ylim([0,0.1])
plt.legend()
# plt.savefig('RMSE vs batches.png')

plt.show()

# common_x_range = [ -1000,50000]
common_y_range = [ 0,20]


plt.figure(figsize=(10, 8), dpi=100)
plt.subplot(2,1,1)
plt.plot(np.sqrt(batch_monitor_cb.MSEtrain), label='RMSE train')
plt.xlabel('Batches')
```

```
plt.ylabel('RMSE')
# plt.xlim(common_x_range)
plt.ylim( common_y_range)
plt.legend()
plt.subplot(2,1,2)
plt.plot( np.sqrt(batch_monitor_cb.MSEtest), label='RMSE test')
plt.xlabel('Batches')
plt.ylabel('RMSE')
# plt.xlim(common_x_range)
plt.ylim(common_y_range)
plt.legend()
# plt.savefig('RMSE vs batches.png')

plt.show()



"""5.3. Plot on the same graph the true values TARGt = S(t+1) and the predicted values Zt . Comments."""

#def for plotting
def plot_mse(data_plot, title = '', update_y = []):
 # Initialize figure
 fig = go.Figure()
 # Add Traces
 for i in data_plot.keys():
  fig.add_trace(go.Scatter(x=list(range(0, len(data_plot[i]))),
                        y=data_plot[i],
                        name=i,
                        marker = dict(size = 10)))
 if len(update_y) > 0:
  fig.update_yaxes(range=update_y)

 #set range
 fig.update_xaxes(rangeslider_visible=True)

 # Set title
 fig.update_layout(title_text=title)
```

```
  fig.show()
  plot(fig, auto_open = True)

pred_train_X.shape

#new_train_X.shape
Zt_train  = MLPpred_model.predict(pred_train_X)
Zt_train.shape

#Plot the training set comparisons
data_plot = {'Train Predictions' : Zt_train.reshape(Zt_train.shape[0]),
            'Train Actuals' : pred_train_y.reshape(pred_train_y.shape[0])}
plot_mse(data_plot, title = 'Predictions vs Actuals [Training Set]')

Zt_test  = MLPpred_model.predict(pred_test_X)
Zt_test.shape

#Plot the training set comparisons
data_plot = {'Test Predictions' : Zt_test.reshape(Zt_test.shape[0]),
            'Test Actuals' : pred_test_y.reshape(pred_test_y.shape[0])}
plot_mse(data_plot, title = 'Predictions vs Actuals [Test Set]')

"""5.4. Compute the Mean Relative Errors of Prediction MREP on the training set :

MREP= average ( | Zt - TARG t | / TARGt) over all cases in the Training set

Compute similarly MREP on the test set. Comments .
"""

(Zt_train.flatten() - pred_train_y)/pred_train_y

#MEAN RELATIVE ERROR FOR TRAIN
MREP_Train = np.mean(np.abs(Zt_train.flatten() - pred_train_y)/pred_train_y)
MREP_Train

#MEAN RELATIVE ERROR FOR TEST
```

```
MREP_Test = np.mean(np.abs(Zt_test.flatten() - pred_test_y)/pred_test_y)
MREP_Test

df_pred_train.shape[0]

import math
def err_est_element(term,size,return_sigma = False):
    sigma = np.around(math.sqrt(term*(100-term)/size),2)

    #for 95% confidence level
    Z_VAL = 1.96
    limit_lower = np.around((term - Z_VAL*sigma),2)
    if limit_lower < 0:
      #print('before if:',limit_lower)
      limit_lower = 0
      #print('after if:',limit_lower)

    limit_upper = np.around((term + Z_VAL*sigma),2)
    if limit_upper > 100:
      #print('before if:',limit_upper)
      limit_upper = 100
      #print('after if:',limit_upper)

    conf_int = [limit_lower,limit_upper]

    if return_sigma:
      return sigma
    else:
      return str(conf_int)

#comparision
def compare_MREP(MREP_Train, MREP_Test):

  train_size = df_pred_train.shape[0]
  test_size = df_pred_test.shape[0]

  CI_MREP_Train = err_est_element(MREP_Train*100 , train_size)
```

```
  print('The {0} is {1} with a 95% confidence interval of {2}'.
      format('MREP_Train in %', round(MREP_Train*100,2) , CI_MREP_Train))

  CI_MREP_Test = err_est_element(MREP_Test*100 , test_size)
  print('The {0} is {1} with a 95% confidence interval of {2}'.
      format('MREP_Test in %', round(MREP_Test*100,2) , CI_MREP_Test))

compare_MREP(MREP_Train,MREP_Test)

"""6. Denote NOD1 NOD2 ... NODk the hidden neurons . For j= 1...k, compute and display the mean activity Yj of
NODj over all cases in the Training set. Display all the weights W1 ... Wk linking the neurons NOD1 ... NODk to the
output node.
For each hidden NODj compute IMPj = Wj Yj = average impact of NODj on the prediction Zt. Display these k impacts
and comment. Identify the hidden neuron NOD* with maximal impact on Zt
"""

W_weights_h_to_out = MLPpred_model.layers[1].get_weights()
W_weights_h_to_out

pd.DataFrame(np.around(W_weights_h_to_out[0],3))

#get output from a layer
from keras import backend as K

# with a Sequential model
get_hidden_layer_output = K.function([MLPpred_model.layers[0].input],
                                  [MLPpred_model.layers[0].output])
layer_output = get_hidden_layer_output(pred_train_X)[0]
np.shape(layer_output)

Yj_mean_activity = np.mean(layer_output,axis = 0)
pd.DataFrame( (np.around(Yj_mean_activity , 3)) )

IMPj_impact = W_weights_h_to_out[0].flatten()*Yj_mean_activity
pd.DataFrame( np.around(IMPj_impact, 3) )

"""7. Denote INP1 INP2 ... INP18 the 18 input neurons. Compute and display the mean activities X1 ... X18 of the 18
```

```
input neurons. Display all the weights U1 ... U18 linking the input nodes INP1 ... INP18 to the neuron NOD*. For each
input neuron INPs compute Fs= Us Xs which is the average impact of input feature "s" on the key hidden neuron
NOD*. Identify the 5 input features with the largest impact on NOD*. Comments."""

# Mean activities of input layer
Xj_mean_activity = np.mean(pred_train_X, axis = 0)
pd.DataFrame(Xj_mean_activity.round(3))

Uj_weights_in_to_hidden = MLPpred_model.layers[0].get_weights()
Uj_weights_in_to_hidden

pd.DataFrame( np.around(Uj_weights_in_to_hidden[0], 3 ) )

Uj_weights_in_to_hidden[0][:,2] *Xj_mean_activity

Uj_weights_in_to_hidden[0].shape

x = pd.DataFrame()

def cal_mean_featureimpact(Us, Xs):
  Fs = pd.DataFrame(np.zeros_like(Us))
  for i in range(Us.shape[1]):
    Fs.loc[:,i] = Us[:,i] * Xs

  return Fs



Fs_feature_impact = cal_mean_featureimpact(Uj_weights_in_to_hidden[0], Xj_mean_activity)

np.around(Fs_feature_impact,3)

Xj_mean_activity.shape

Fs_feature_impact = np.matmul( Xj_mean_activity, Uj_weights_in_to_hidden[0])
pd.DataFrame(Fs_feature_impact.round(3))
```

Appendix A.2

```
# -*- coding: utf-8 -*-
"""MATH6373_Final_tuning_V1.ipynb

Automatically generated by Colaboratory.

Original file is located at
    https://colab.research.google.com/drive/18D2WtmWDYtEu_0E7YH16usBkbOMsXDQT
"""

# Commented out IPython magic to ensure Python compatibility.
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import plotly.graph_objects as go
from plotly.offline import plot
import plotly.express as px
import seaborn as sns
# %matplotlib inline



import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.initializers import Constant, RandomNormal, RandomUniform
from tensorflow import keras

# show nice table in colab
# %load_ext google.colab.data_table
from google.colab import data_table

file_name = 'processed_data.xlsx'
train_sheet_name = 'df_pred_train'
test_sheet_name = 'df_pred_test'
df_pred_train = pd.read_excel(file_name , sheet_name= train_sheet_name,index_col=0)
```

```
df_pred_test = pd.read_excel(file_name , sheet_name= test_sheet_name, index_col=0)

df_pred_train.head()

df_pred_test.head()

#data transformation to numpy arrays
pred_train_X = df_pred_train.iloc[:,:-1].values
pred_train_y = df_pred_train.iloc[:,-1].values
pred_test_X = df_pred_test.iloc[:, :-1].values
pred_test_y = df_pred_test.iloc[:,-1].values

print('shapes of train data: X: ', pred_train_X.shape, '\t y: ',pred_train_y.shape)
print('shapes of test data: X: ', pred_test_X.shape, '\t y: ',pred_test_y.shape)

"""# EXHAUSTIVE RUN OF SELECTED MODEL"""

# tuned parameters
epochs = [5000]
batches = [64]#[32,64,96,128]

bias_init_hidden = [20] #[ 10, 20,40, 80,160]
bias_init_out = [40] #[ 10, 20,40, 80, 160]
weight_init_mode = ['uniform'] #['uniform', 'glorot_uniform','normal']

# current tuning
# Untuned parameters
lr_list =[1e-2]#,1e-3 ]# [1e-2, 1e-3, 1e-4,1e-5]
hidden_dim_list = [19]#[4,9,14,19,24,29,34,39,44,49,54] #[10,20,30,40,50,55]

#Dimensions for the MLP

MLP_INPUT_DIM = df_pred_train.shape[1] - 1
MLP_OUTPUT_DIM = 1
MLP_HIDDEN_LAYER_SIZE = hidden_dim_list[0]

#default parameters
```

```
#model parameters
BIA_INI_H = bias_init_hidden[0]
BIA_INI_O = bias_init_out[0]
KERNEL_INIT = weight_init_mode[0]

#fitting parameters
MLP_LEARNING_RATE = lr_list[0]#5e-5
MLP_DECAY_RATE = 1e-5
MLP_EPOCH_SIZE = 40000
PATIENCE = 1000
MLP_BATCH_SIZE = batches[0]



selected_optimizer = keras.optimizers.Adam(learning_rate = MLP_LEARNING_RATE )

MLPpred_model = keras.Sequential()
MLPpred_model.add(keras.layers.Dense(units = MLP_HIDDEN_LAYER_SIZE,
                                activation='relu',
                                input_dim = MLP_INPUT_DIM,
                                bias_initializer = keras.initializers.Constant(
                                   value = BIA_INI_H) ))

MLPpred_model.add(keras.layers.Dense(units = MLP_OUTPUT_DIM,
                                activation='relu',
                                bias_initializer= keras.initializers.Constant(
                                   value = BIA_INI_O)))

MLPpred_model.summary()

# the customized callback to record losses after each batch

class mlpMyHistory(keras.callbacks.Callback):
   def on_train_begin(self, logs={}):
     self.MSEtrain = []
     self.MSEtest = []

   def on_train_batch_end(self, batch, logs={}):
```

```
      self.MSEtrain.append(logs['loss'])

    def on_test_batch_end(self, batch, logs={}):
      self.MSEtest.append(logs['loss'])


#object created for custom class
batch_monitor_cb = mlpMyHistory()

#callbacks
earlystopping_cb = keras.callbacks.EarlyStopping(monitor='val_loss',
                                                 mode='min', verbose=1,
                                                 patience= PATIENCE,
                                                 restore_best_weights=True)

tensorboard_cb = tf.keras.callbacks.TensorBoard(run_logdir)

callbacks_list = [batch_monitor_cb, earlystopping_cb, tensorboard_cb ]

MLPpred_model.compile(optimizer= selected_optimizer,
                      loss='mean_squared_error')
Monitor = MLPpred_model.fit( x = pred_train_X,
                       y = pred_train_y,
                       batch_size = MLP_BATCH_SIZE,
                       epochs = MLP_EPOCH_SIZE,
                       callbacks = callbacks_list,
                       validation_data = (pred_test_X,
                                          pred_test_y),
                       verbose = 1)

MLPpred_model.evaluate(pred_test_X, pred_test_y)



"""# Randomsearch CV for hyper parameter tuning

**TUNING OF MODEL**
```

```
"""

#Dimensions for the MLP
MLP_INPUT_DIM = df_pred_train.shape[1] - 1
MLP_OUTPUT_DIM = 1
MLP_HIDDEN_LAYER_SIZE = 4

#default parameters
#model parameters
BIA_INI_H = 10
BIA_INI_O = 10

#fitting parameters
MLP_LEARNING_RATE = 1e-3
MLP_EPOCH_SIZE = 4000
PATIENCE = 1000#500
MLP_BATCH_SIZE = 32

#Adam optimizer
#new hyperparameter β, simply called the momentum, which must be set between 0 (high friction) and 1 (no friction).
#A typical momentum value is 0.9.
selected_optimizer = keras.optimizers.Adam(learning_rate = MLP_LEARNING_RATE )

from sklearn.model_selection import GridSearchCV
from sklearn.model_selection import RandomizedSearchCV
from keras.wrappers.scikit_learn import KerasRegressor

# let's create a function that creates the model (required for KerasRegressor)
# while accepting the hyperparameters we want to tune
# we also pass some default values such as optimizer='Adam'
# repeat some of the initial values here so we make sure they were not changed

def create_model(learning_rate = 1e-3,
        kernel_init='uniform',
        bias_init_hidden = BIA_INI_H, bias_init_out = BIA_INI_O,
        hidden_dim = MLP_HIDDEN_LAYER_SIZE ):
```

```
  MLPpred_model = keras.Sequential()
  MLPpred_model.add(keras.layers.Dense(units = hidden_dim,
                                       activation='relu',
                                       input_dim = MLP_INPUT_DIM,
                                       kernel_initializer = kernel_init,
                                       bias_initializer = keras.initializers.Constant(
                                          value = bias_init_hidden) ))

  MLPpred_model.add(keras.layers.Dense(units = MLP_OUTPUT_DIM,
                                       activation='relu',
                                       kernel_initializer = kernel_init,
                                       bias_initializer= keras.initializers.Constant(
                                          value = bias_init_out)))

  #compile model
  selected_optimizer = keras.optimizers.Adam(learning_rate = learning_rate )
  MLPpred_model.compile(optimizer= selected_optimizer, loss='mean_squared_error')

  return MLPpred_model

# STEp BY STEP tuning
# fix random seed for reproducibility (this might work or might not work
# depending on each library's implenentation)
SEED = 2020
np.random.seed(SEED)

# create the sklearn model for the network
# model_init_batch_epoch_CV = KerasClassifier(build_fn=create_model_2, verbose=1)
modeltuning_CV = KerasRegressor(build_fn= create_model, verbose =1)

# tuned parameters
epochs = [40000]#[5000]
batches = [64]#[32,64,96,128]

bias_init_hidden = [20] #[ 10, 20,40, 80,160]
bias_init_out = [40] #[ 10, 20,40, 80, 160]
weight_init_mode = ['uniform'] #['uniform', 'glorot_uniform','normal']
```

```
# current tuning
# Untuned parameters
lr_list =[1e-2]# [1e-2, 1e-3, 1e-4,1e-5]
hidden_dim_list = [1,4,9,14,19,24,29,34,39,44,49,54] #[10,20,30,40,50,55]


es = keras.callbacks.EarlyStopping(monitor='val_loss', patience= 100,restore_best_weights=True)
# grid search for initializer, batch size and number of epochs
param_grid = dict(batch_size=batches,
           learning_rate = lr_list,
           bias_init_hidden = bias_init_hidden,
           bias_init_out = bias_init_out,
           kernel_init = weight_init_mode,
           hidden_dim = hidden_dim_list)

grid = RandomizedSearchCV(estimator= modeltuning_CV,
              param_distributions= param_grid,
              cv = 10, random_state = SEED,
              n_jobs = -1  )



grid_result = grid.fit(pred_train_X, pred_train_y, callbacks=[es], validation_split = 0.1,
             verbose = 1,epochs = epochs[0])



# grid_search = GridSearchCV(estimator= modeltuning_CV,
#                param_grid = param_grid,
#                cv = 10, n_jobs = -1)

# grid_result = grid.fit(pred_train_X, pred_train_y, callbacks=[es],
#              validation_data = (pred_test_X, pred_test_y),
#              verbose = 1, epochs = epochs[0])

# print results - 7 with  run - for hidden layer size - limited training 1e-2
print(f'Best Accuracy for {grid_result.best_score_:.4} using {grid_result.best_params_}')
means = grid_result.cv_results_['mean_test_score']
```

```
stds = grid_result.cv_results_['std_test_score']
params = grid_result.cv_results_['params']
for mean, stdev, param in zip(means, stds, params):
  print(f'mean={mean:.4}, std={stdev:.4} using {param}')

pd.DataFrame.from_dict(grid_result.cv_results_).to_excel('gridsearch_predictor_0606_4.xlsx')

# print results - 6 with  run - for hidden layer size - limited training 1e-3
print(f'Best Accuracy for {grid_result.best_score_:.4} using {grid_result.best_params_}')
means = grid_result.cv_results_['mean_test_score']
stds = grid_result.cv_results_['std_test_score']
params = grid_result.cv_results_['params']
for mean, stdev, param in zip(means, stds, params):
  print(f'mean={mean:.4}, std={stdev:.4} using {param}')

pd.DataFrame.from_dict(grid_result.cv_results_).to_excel('gridsearch_predictor_0606_4.xlsx')

# print results - 5 run - for hidden layer size - limited training
print(f'Best Accuracy for {grid_result.best_score_:.4} using {grid_result.best_params_}')
means = grid_result.cv_results_['mean_test_score']
stds = grid_result.cv_results_['std_test_score']
params = grid_result.cv_results_['params']
for mean, stdev, param in zip(means, stds, params):
  print(f'mean={mean:.4}, std={stdev:.4} using {param}')

pd.DataFrame.from_dict(grid_result.cv_results_).to_excel('gridsearch_predictor_0606_4.xlsx')

# print results - 4 run - for hidden layer size - full training
print(f'Best Accuracy for {grid_result.best_score_:.4} using {grid_result.best_params_}')
means = grid_result.cv_results_['mean_test_score']
stds = grid_result.cv_results_['std_test_score']
params = grid_result.cv_results_['params']
for mean, stdev, param in zip(means, stds, params):
  print(f'mean={mean:.4}, std={stdev:.4} using {param}')

pd.DataFrame.from_dict(grid_result.cv_results_).to_excel('gridsearch_predictor_0606_4.xlsx')
```

```
# print results - 3 run - for bias hidden, bias out, weights
print(f'Best Accuracy for {grid_result.best_score_:.4} using {grid_result.best_params_}')
means = grid_result.cv_results_['mean_test_score']
stds = grid_result.cv_results_['std_test_score']
params = grid_result.cv_results_['params']
for mean, stdev, param in zip(means, stds, params):
    print(f'mean={mean:.4}, std={stdev:.4} using {param}')

pd.DataFrame.from_dict(grid_result.cv_results_).to_excel('gridsearch_predictor_0606_3.xlsx')

# print results - 2 run - for bias hidden and bias out
print(f'Best Accuracy for {grid_result.best_score_:.4} using {grid_result.best_params_}')
means = grid_result.cv_results_['mean_test_score']
stds = grid_result.cv_results_['std_test_score']
params = grid_result.cv_results_['params']
for mean, stdev, param in zip(means, stds, params):
    print(f'mean={mean:.4}, std={stdev:.4} using {param}')

pd.DataFrame.from_dict(grid_result.cv_results_).to_excel('gridsearch_predictor_0606_2.xlsx')

# print results - 1 run - for Batch size and learning rates
print(f'Best Accuracy for {grid_result.best_score_:.4} using {grid_result.best_params_}')
means = grid_result.cv_results_['mean_test_score']
stds = grid_result.cv_results_['std_test_score']
params = grid_result.cv_results_['params']
for mean, stdev, param in zip(means, stds, params):
    print(f'mean={mean:.4}, std={stdev:.4} using {param}')

pd.DataFrame.from_dict(grid_result.cv_results_).to_excel('gridsearch_predictor_0506.xlsx')

# print results - ninth run - for hidden layer size
print(f'Best Accuracy for {grid_result.best_score_:.4} using {grid_result.best_params_}')
means = grid_result.cv_results_['mean_test_score']
stds = grid_result.cv_results_['std_test_score']
params = grid_result.cv_results_['params']
for mean, stdev, param in zip(means, stds, params):
    print(f'mean={mean:.4}, std={stdev:.4} using {param}')
```

```
pd.DataFrame.from_dict(grid_result.cv_results_).to_excel('gridsearch_predictor_1000e.xlsx')

# print results - eighth run - for hidden layer size
print(f'Best Accuracy for {grid_result.best_score_:.4} using {grid_result.best_params_}')
means = grid_result.cv_results_['mean_test_score']
stds = grid_result.cv_results_['std_test_score']
params = grid_result.cv_results_['params']
for mean, stdev, param in zip(means, stds, params):
    print(f'mean={mean:.4}, std={stdev:.4} using {param}')

# print results - seventh run - for hidden layer size
print(f'Best Accuracy for {grid_result.best_score_:.4} using {grid_result.best_params_}')
means = grid_result.cv_results_['mean_test_score']
stds = grid_result.cv_results_['std_test_score']
params = grid_result.cv_results_['params']
for mean, stdev, param in zip(means, stds, params):
    print(f'mean={mean:.4}, std={stdev:.4} using {param}')

# print results - sixth run - with Adam
print(f'Best Accuracy for {grid_result.best_score_:.4} using {grid_result.best_params_}')
means = grid_result.cv_results_['mean_test_score']
stds = grid_result.cv_results_['std_test_score']
params = grid_result.cv_results_['params']
for mean, stdev, param in zip(means, stds, params):
    print(f'mean={mean:.4}, std={stdev:.4} using {param}')

# print results - fifth run - with Adam
print(f'Best Accuracy for {grid_result.best_score_:.4} using {grid_result.best_params_}')
means = grid_result.cv_results_['mean_test_score']
stds = grid_result.cv_results_['std_test_score']
params = grid_result.cv_results_['params']
for mean, stdev, param in zip(means, stds, params):
    print(f'mean={mean:.4}, std={stdev:.4} using {param}')

"""Fifth run shows that :
1. higher hidden layer size did not increase accuracy
```

```
2. optimum epochs is 4000
3. batch size 16 is optimum
4.
"""

# print results - fourth run
print(f'Best Accuracy for {grid_result.best_score_:.4} using {grid_result.best_params_}')
means = grid_result.cv_results_['mean_test_score']
stds = grid_result.cv_results_['std_test_score']
params = grid_result.cv_results_['params']
for mean, stdev, param in zip(means, stds, params):
    print(f'mean={mean:.4}, std={stdev:.4} using {param}')

# print results - third run
print(f'Best Accuracy for {grid_result.best_score_:.4} using {grid_result.best_params_}')
means = grid_result.cv_results_['mean_test_score']
stds = grid_result.cv_results_['std_test_score']
params = grid_result.cv_results_['params']
for mean, stdev, param in zip(means, stds, params):
    print(f'mean={mean:.4}, std={stdev:.4} using {param}')

"""Third run shows that :
All parameters have close scores, hence run again with higher patience.
"""

# print results - second run
print(f'Best Accuracy for {grid_result.best_score_:.4} using {grid_result.best_params_}')
means = grid_result.cv_results_['mean_test_score']
stds = grid_result.cv_results_['std_test_score']
params = grid_result.cv_results_['params']
for mean, stdev, param in zip(means, stds, params):
    print(f'mean={mean:.4}, std={stdev:.4} using {param}')

"""Second run shows that follwoing are best candidates :
1. kernel_init = 'uniform'
2. epochs = 3000
3. bias_init = [5,10,20]
```

```
4. batch_size = [16,32,64]
"""

# print results - first run
print(f'Best Accuracy for {grid_result.best_score_:.4} using {grid_result.best_params_}')
means = grid_result.cv_results_['mean_test_score']
stds = grid_result.cv_results_['std_test_score']
params = grid_result.cv_results_['params']
for mean, stdev, param in zip(means, stds, params):
    print(f'mean={mean:.4}, std={stdev:.4} using {param}')

"""First run shows that following values are best candidates
1. kernel_init = 'uniform'
"""
```