25_Dim_ESN

January 2, 2020

1 Predicting 25 Dim System using Echo State Neural Network

1.0.1 Importing Required Libraries

[1]:
```
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
import ESN
import pandas as pd
```

1.0.2 Set seed for random weights generator

[2]:
```
def set_seed(seed=None):
    """Making the seed (for random values) variable if None"""

    # Set the seed
    if seed is None:
        import time
        seed = int((time.time()*10**6) % 4294967295)
    try:
        np.random.seed(seed)
    except Exception as e:
        print( "!!! WARNING !!!: Seed was not set correctly.")
        print( "!!! Seed that we tried to use: "+str(seed))
        print( "!!! Error message: "+str(e))
        seed = None
    print( "Seed used for random values:", seed)
    return seed
```

[3]:
```
## Set a particular seed for the random generator (for example seed = 42), or␣
 ↪use a "random" one (seed = None)
# NB: reservoir performances should be averaged accross at least 30 random␣
 ↪instances (with the same set of parameters)
seed = 42 #None #42
```

[4]:
```
set_seed(seed) #random.seed(seed)
```

```
Seed used for random values: 42
```

[4]: 42

[18]:
```
initLen = 1000
trainLen = initLen + 5000
testLen = 2900
```

[6]:
```
df =  pd.read_excel(r'C:\Users\INFO-DSK-02\Desktop\Lorentz Multi Dimension␣
↪Prediction-Phase-2\Final_Version\3D_ReservoirComputing\Input\25 Dim␣
↪Data\25_Dim.xlsx', index = False)
```

[17]:
```
df.shape
```

[17]: (8929, 25)

2 EDA

2.0.1 Split data for training and testing and creating teaches to train ESN on Input data

[10]:
```
df[:]
```

[10]:

	X1	X2	X3	X4	X5	X6	X7	X8	\
0	0.0633	0.0633	0.0632	0.0632	0.0632	0.0632	0.063175	0.063175	
1	0.0618	0.0634	0.0633	0.0637	0.0637	0.0637	0.063754	0.063754	
2	0.0619	0.0636	0.0635	0.0639	0.0639	0.0639	0.063878	0.063878	
3	0.0627	0.0639	0.0638	0.0641	0.0641	0.0641	0.064146	0.064146	
4	0.0634	0.0641	0.0640	0.0643	0.0643	0.0643	0.064357	0.064357	
…	…	…	…	…	…	…	…	…	
8924	0.0642	0.0649	0.0648	0.0666	0.0666	0.0666	0.069307	0.069307	
8925	0.0624	0.0659	0.0658	0.0674	0.0674	0.0674	0.070398	0.070398	
8926	0.0616	0.0664	0.0663	0.0670	0.0670	0.0670	0.071418	0.071418	
8927	0.0629	0.0660	0.0659	0.0660	0.0660	0.0660	0.071998	0.071998	
8928	0.0659	0.0648	0.0647	0.0653	0.0653	0.0653	0.071979	0.071979	

	X9	X10	…	X16	X17	X18	X19	X20	\
0	0.0631	0.0631	…	0.062359	0.062359	0.062126	0.061544	0.060380	
1	0.0636	0.0636	…	0.062924	0.062924	0.062693	0.062105	0.060928	
2	0.0638	0.0638	…	0.063048	0.063048	0.062812	0.062223	0.061049	
3	0.0640	0.0640	…	0.063314	0.063314	0.063079	0.062488	0.061307	
4	0.0643	0.0643	…	0.063524	0.063524	0.063294	0.062701	0.061516	
…	…	…	…	…	…	…	…	…	
8924	0.0697	0.0697	…	0.069487	0.069487	0.074169	0.073474	0.074096	
8925	0.0696	0.0696	…	0.070596	0.070596	0.074172	0.073477	0.074119	
8926	0.0698	0.0698	…	0.071639	0.071639	0.073812	0.073120	0.073238	
8927	0.0699	0.0699	…	0.072389	0.072389	0.073267	0.072580	0.072117	
8928	0.0698	0.0698	…	0.072698	0.072698	0.072600	0.071920	0.071385	

```
            X21       X22       X23       X24       X25
0      0.058283  0.055022  0.051412  0.047686  0.044891
1      0.058812  0.055522  0.051879  0.048118  0.045298
2      0.058929  0.055632  0.051982  0.048214  0.045388
3      0.059179  0.055868  0.052202  0.048418  0.045580
4      0.059380  0.056058  0.052380  0.048583  0.045736
…           …         …         …         …         …
8924   0.071524  0.067522  0.063092  0.058519  0.055089
8925   0.071545  0.067543  0.063111  0.058536  0.055105
8926   0.070695  0.066740  0.062361  0.057841  0.054450
8927   0.069614  0.065719  0.061407  0.056956  0.053617
8928   0.068907  0.065051  0.060783  0.056377  0.053073

[8929 rows x 25 columns]
```

[14]:
```
data_in = df[:]
```

[15]:
```
data_in = np.array(data_in)
```

[19]:
```
train_in = np.array(data_in[0:trainLen])
train_out = np.array(data_in[0+10:trainLen+10])
test_in = np.array(data_in[trainLen:trainLen+testLen])
test_out = np.array(data_in[trainLen+10:trainLen+testLen+10])
```

[20]:
```
len(test_in_t)
```

[20]:
```
800
```

2.0.2 Modify Parameters to tune ESN for better fit

[22]:
```
n_reservoir = 400 # number of recurrent units
leak_rate = 0.2 # leaking rate (=1/time_constant_of_neurons)
spectral_radius = 1.1 # Scaling of recurrent matrix
input_scaling = 1. # Scaling of input matrix
proba_non_zero_connec_W = 0.2 # Sparsity of recurrent matrix: Perceptage of␣
 ↪non-zero connections in W matrix
proba_non_zero_connec_Win = 1. # Sparsity of input matrix
proba_non_zero_connec_Wfb = 1. # Sparsity of feedback matrix
regularization_coef =   0.1 #None # regularization coefficient, if None,␣
 ↪pseudo-inverse is use instead of ridge regression
```

[24]:
```
n_inputs = 25
input_bias = True # add a constant input to 1
n_outputs = 25
```

[25]:
```
N = n_reservoir#100
dim_inp = n_inputs #26
```

2.0.3 Generating weights for input and hidden layers

[26]:
```
### Generating random weight matrices with custom method
W = np.random.rand(N,N) - 0.5
if input_bias:
    Win = np.random.rand(N,dim_inp+1) - 0.5
else:
    Win = np.random.rand(N,dim_inp) - 0.5
Wfb = np.random.rand(N,n_outputs) - 0.5
```

[27]:
```
## delete the fraction of connections given the sparsity (i.e. proba of␣
 ↪non-zero connections):
mask = np.random.rand(N,N) # create a mask Uniform[0;1]
W[mask > proba_non_zero_connec_W] = 0 # set to zero some connections given by␣
 ↪the mask
mask = np.random.rand(N,Win.shape[1])
Win[mask > proba_non_zero_connec_Win] = 0
# mask = np.random.rand(N,Wfb.shape[1])
# Wfb[mask > proba_non_zero_connec_Wfb] = 0
```

[28]:
```
## SCALING of matrices
# scaling of input matrix
Win = Win * input_scaling
# scaling of recurrent matrix
# compute the spectral radius of these weights:
print( 'Computing spectral radius...')
original_spectral_radius = np.max(np.abs(np.linalg.eigvals(W)))
#TODO: check if this operation is quicker: max(abs(linalg.eig(W)[0])) #from␣
 ↪scipy import linalg
print( "default spectral radius before scaling:", original_spectral_radius)
# rescale them to reach the requested spectral radius:
W = W * (spectral_radius / original_spectral_radius)
print( "spectral radius after scaling", np.max(np.abs(np.linalg.eigvals(W))))
```

```
Computing spectral radius…
default spectral radius before scaling: 2.6614164786222583
spectral radius after scaling 1.1000000000000014
```

2.1 Input data dimensions

[29]:
```
print('Dimensions of Training data: ', train_in.shape[1])
print('Dimensions of Testing data: ', test_in.shape[1])
```

```
Dimensions of Training data:  25
Dimensions of Testing data:  25
```

2.1.1 Pass Parameters to ESN

[30]:
```
reservoir = ESN.ESN(lr=leak_rate, W=W, Win=Win, input_bias=input_bias,␣
↪ridge=regularization_coef, Wfb=None, fbfunc=None)
```

2.2 Input data to reservoir model

[31]:
```
internal_trained = reservoir.train(inputs=[train_in], teachers=[train_out],␣
↪wash_nr_time_step=initLen, verbose=False)
output_pred, internal_pred = reservoir.run(inputs=[test_in,], reset_state=False)
errorLen = len(test_out[:]) #testLen #2000
```

2.3 Dimensions of the output data

[32]:
```
print('Shape of Output data Dimensions: ', output_pred[0].shape[1])
```

```
Shape of Output data Dimensions:  25
```

2.3.1 Create dataframe for predicted values and test values

[33]:
```
import pandas as pd
df_pred = pd.DataFrame(output_pred[0])
```

[34]:
```
df_pred
```

[34]:

	0	1	2	3	4	5	6	\
0	0.063513	0.064432	0.064336	0.064477	0.064477	0.064477	0.068031	
1	0.063574	0.064491	0.064394	0.064175	0.064175	0.064175	0.067069	
2	0.063635	0.064584	0.064488	0.064442	0.064442	0.064442	0.066378	
3	0.063936	0.064508	0.064411	0.065368	0.065368	0.065368	0.066209	
4	0.064340	0.064329	0.064233	0.066430	0.066430	0.066430	0.066684	
…	…	…	…	…	…	…	…	
2895	0.062788	0.065299	0.065200	0.066437	0.066437	0.066437	0.067021	
2896	0.062881	0.064992	0.064894	0.065976	0.065976	0.065976	0.066722	
2897	0.063024	0.064718	0.064621	0.065545	0.065545	0.065545	0.066816	
2898	0.063041	0.064661	0.064564	0.065368	0.065368	0.065368	0.067162	
2899	0.062981	0.064773	0.064676	0.065721	0.065721	0.065721	0.067581	

	7	8	9	…	15	16	17	\
0	0.068031	0.070078	0.070078	…	0.071188	0.071188	0.073350	
1	0.067069	0.070186	0.070186	…	0.071690	0.071690	0.073003	
2	0.066378	0.070209	0.070209	…	0.072061	0.072061	0.072640	
3	0.066209	0.070080	0.070080	…	0.072124	0.072124	0.072269	
4	0.066684	0.069771	0.069771	…	0.071900	0.071900	0.071884	

```
…         …         …         …  …         …         …         …
2895  0.067021  0.069243  0.069243  …  0.071244  0.071244  0.073286
2896  0.066722  0.069226  0.069226  …  0.071664  0.071664  0.073474
2897  0.066816  0.069131  0.069131  …  0.071912  0.071912  0.073416
2898  0.067162  0.069025  0.069025  …  0.071984  0.071984  0.073121
2899  0.067581  0.068949  0.068949  …  0.071981  0.071981  0.072636

             18        19        20        21        22        23        24
0     0.072662  0.070718  0.068263  0.064444  0.060215  0.055850  0.052577
1     0.072319  0.070960  0.068497  0.064664  0.060422  0.056042  0.052757
2     0.071959  0.071125  0.068656  0.064815  0.060562  0.056172  0.052880
3     0.071592  0.071231  0.068758  0.064912  0.060652  0.056256  0.052959
4     0.071210  0.071291  0.068816  0.064966  0.060704  0.056303  0.053003
…         …         …         …         …         …         …         …
2895  0.072599  0.068982  0.066587  0.062862  0.058737  0.054480  0.051286
2896  0.072785  0.069224  0.066821  0.063082  0.058943  0.054671  0.051466
2897  0.072728  0.069572  0.067157  0.063399  0.059240  0.054946  0.051725
2898  0.072436  0.069987  0.067557  0.063777  0.059593  0.055273  0.052033
2899  0.071955  0.070410  0.067965  0.064163  0.059953  0.055607  0.052348

[2900 rows x 25 columns]
```

[36]:
```
test_out = pd.DataFrame(test_out)
```

[37]:
```
test_out
```

[37]:
```
          0       1       2       3       4       5         6         7   \
0     0.0607  0.0641  0.0640  0.0646  0.0646  0.0646  0.067486  0.067486
1     0.0634  0.0608  0.0607  0.0646  0.0646  0.0646  0.066195  0.066195
2     0.0658  0.0590  0.0589  0.0648  0.0648  0.0648  0.065305  0.065305
3     0.0671  0.0596  0.0595  0.0657  0.0657  0.0657  0.065507  0.065507
4     0.0676  0.0619  0.0618  0.0672  0.0672  0.0672  0.066806  0.066806
…        …       …       …       …       …       …         …         …
2895  0.0614  0.0670  0.0669  0.0657  0.0657  0.0657  0.067453  0.067453
2896  0.0614  0.0686  0.0685  0.0655  0.0655  0.0655  0.066565  0.066565
2897  0.0622  0.0672  0.0671  0.0651  0.0651  0.0651  0.066005  0.066005
2898  0.0625  0.0637  0.0637  0.0647  0.0647  0.0647  0.065755  0.065755
2899  0.0620  0.0608  0.0607  0.0643  0.0643  0.0643  0.065768  0.065768

          8       9   …        15        16        17        18        19  \
0     0.0696  0.0696  …  0.072339  0.072339  0.074464  0.073766  0.071456
1     0.0692  0.0692  …  0.073372  0.073372  0.073457  0.072768  0.071317
2     0.0690  0.0690  …  0.073727  0.073727  0.072286  0.071609  0.071213
3     0.0693  0.0693  …  0.073571  0.073571  0.071569  0.070898  0.071781
4     0.0699  0.0699  …  0.072921  0.072921  0.071257  0.070589  0.072799
…        …       …  …         …         …         …         …         …
2895  0.0661  0.0661  …  0.072058  0.072058  0.074481  0.073783  0.067978
```

```
2896  0.0662  0.0662  …  0.073181  0.073181  0.074229  0.073533  0.067698
2897  0.0669  0.0669  …  0.073063  0.073063  0.073246  0.072559  0.067887
2898  0.0679  0.0679  …  0.071991  0.071991  0.071781  0.071108  0.068622
2899  0.0691  0.0691  …  0.070850  0.070850  0.070355  0.069695  0.069810

             20        21        22        23        24
0     0.068975  0.065116  0.060843  0.056433  0.053125
1     0.068841  0.064989  0.060725  0.056323  0.053022
2     0.068740  0.064894  0.060636  0.056241  0.052945
3     0.069289  0.065412  0.061121  0.056690  0.053367
4     0.070271  0.066340  0.061987  0.057494  0.054124
…          …         …         …         …         …
2895  0.065618  0.061946  0.057882  0.053686  0.050539
2896  0.065348  0.061692  0.057644  0.053466  0.050332
2897  0.065530  0.061863  0.057804  0.053614  0.050472
2898  0.066240  0.062534  0.058431  0.054195  0.051019
2899  0.067387  0.063616  0.059442  0.055134  0.051902

[2900 rows x 25 columns]
```

2.3.2 MSE for X1

[38]:
```
## printing errors made on test set
# mse = sum( np.square( test_out[:] - output_pred[0] ) ) / errorLen
# print( 'MSE = ' + str( mse ))
mse_x = np.mean((test_out[0][:] - df_pred[0])**2) # Mean Squared Error: see␣
 ↪https://en.wikipedia.org/wiki/Mean_squared_error
rmse_x = np.sqrt(mse_x) # Root Mean Squared Error: see https://en.wikipedia.org/
 ↪wiki/Root-mean-square_deviation for more info
nmrse_mean_x = abs(rmse_x / np.mean(test_out[0][:])) # Normalised RMSE (based␣
 ↪on mean)
nmrse_maxmin_x = rmse_x / abs(np.max(test_out[0][:]) - np.min(test_out[0][:]))␣
 ↪# Normalised RMSE (based on max - min)
```

[39]:
```
print("\n**********  MSE and RMSE for Predictions on X  **********")
print("Errors computed over %d time steps" % (errorLen))
print("\nMean Squared error (MSE) for x : \t\t%.4e " % (mse_x) )
print("Root Mean Squared error (RMSE) for x : \t\t%.4e\n " % rmse_x )
print("Normalized RMSE (based on mean) for x : \t%.4e " % (nmrse_mean_x) )
print("Normalized RMSE (based on max - min) for x : \t%.4e " % (nmrse_maxmin_x)␣
 ↪)
print("*************************************************************\n")
```

```
**********  MSE and RMSE for Predictions on X  **********
Errors computed over 2900 time steps
```

```
Mean Squared error (MSE) for x :                1.2286e-05
Root Mean Squared error (RMSE) for x :          3.5051e-03

Normalized RMSE (based on mean) for x :         5.4971e-02
Normalized RMSE (based on max - min) for x :    1.5240e-01
**************************************************
```

2.3.3 MSE for X2

[40]:
```
## printing errors made on test set
# mse = sum( np.square( test_out[:] - output_pred[0] ) ) / errorLen
# print( 'MSE = ' + str( mse ))
mse_y = np.mean((test_out[1][:] - df_pred[1])**2) # Mean Squared Error: see␣
 ↪https://en.wikipedia.org/wiki/Mean_squared_error
rmse_y = np.sqrt(mse_x) # Root Mean Squared Error: see https://en.wikipedia.org/
 ↪wiki/Root-mean-square_deviation for more info
nmrse_mean_y = abs(rmse_y / np.mean(test_out[1][:])) # Normalised RMSE (based␣
 ↪on mean)
nmrse_maxmin_y = rmse_y / abs(np.max(test_out[1][:]) - np.min(test_out[1][:]))␣
 ↪# Normalised RMSE (based on max - min)
```

[41]:
```
print("\n**********  MSE and RMSE for Predictions on Y  **********")
print("Errors computed over %d time steps" % (errorLen))
print("\nMean Squared error (MSE) for Y : \t\t%.4e " % (mse_y) )
print("Root Mean Squared error (RMSE) for Y : \t\t%.4e\n " % rmse_y )
print("Normalized RMSE (based on mean) for Y : \t%.4e " % (nmrse_mean_y) )
print("Normalized RMSE (based on max - min) for Y : \t%.4e " % (nmrse_maxmin_y)␣
 ↪)
print("**************************************************\n")
```

```
**********  MSE and RMSE for Predictions on Y  **********
Errors computed over 2900 time steps

Mean Squared error (MSE) for Y :                9.7241e-06
Root Mean Squared error (RMSE) for Y :          3.5051e-03

Normalized RMSE (based on mean) for Y :         5.4173e-02
Normalized RMSE (based on max - min) for Y :    1.6771e-01
**************************************************
```

2.3.4 MSE for X3

[42]:
```
## printing errors made on test set
# mse = sum( np.square( test_out[:] - output_pred[0] ) ) / errorLen
# print( 'MSE = ' + str( mse ))
mse_y = np.mean((test_out[2][:] - df_pred[1])**2) # Mean Squared Error: see␣
↪https://en.wikipedia.org/wiki/Mean_squared_error
rmse_y = np.sqrt(mse_x) # Root Mean Squared Error: see https://en.wikipedia.org/
↪wiki/Root-mean-square_deviation for more info
nmrse_mean_y = abs(rmse_y / np.mean(test_out[2][:])) # Normalised RMSE (based␣
↪on mean)
nmrse_maxmin_y = rmse_y / abs(np.max(test_out[2][:]) - np.min(test_out[2][:]))␣
↪# Normalised RMSE (based on max - min)
```

[43]:
```
print("\n**********  MSE and RMSE for Predictions on Z  **********")
print("Errors computed over %d time steps" % (errorLen))
print("\nMean Squared error (MSE) for Z : \t\t%.4e " % (mse_y) )
print("Root Mean Squared error (RMSE) for Z : \t\t%.4e\n " % rmse_y )
print("Normalized RMSE (based on mean) for Z : \t%.4e " % (nmrse_mean_y) )
print("Normalized RMSE (based on max - min) for Z : \t%.4e " % (nmrse_maxmin_y)␣
↪)
print("************************************************************\n")
```

```
**********  MSE and RMSE for Predictions on Z  **********
Errors computed over 2900 time steps

Mean Squared error (MSE) for Z :                9.6950e-06
Root Mean Squared error (RMSE) for Z :          3.5051e-03

Normalized RMSE (based on mean) for Z :         5.4254e-02
Normalized RMSE (based on max - min) for Z :    1.6771e-01
************************************************************
```

[48]:
```
df_local_error = pd.DataFrame()
```

[50]:
```
for i in range(0,25):
    df_local_error['X{}_Local_Error'.format(i+1)] = test_out[i] - df_pred[i]
```

[51]:
```
df_local_error.describe()
```

[51]:

	X1_Local_Error	X2_Local_Error	X3_Local_Error	X4_Local_Error	\
count	2900.000000	2900.000000	2900.000000	2900.000000	
mean	0.000063	0.000035	0.000034	0.000104	
std	0.003505	0.003119	0.003114	0.001428	
min	-0.009616	-0.008875	-0.008881	-0.004092	

```
25%         -0.002377       -0.002064       -0.002058       -0.000905
50%         -0.000027       -0.000154       -0.000155        0.000092
75%          0.002279        0.002056        0.002054        0.001058
max          0.011541        0.010139        0.010135        0.006249

       X5_Local_Error  X6_Local_Error  X7_Local_Error  X8_Local_Error  \
count     2900.000000     2900.000000     2900.000000     2900.000000
mean         0.000104        0.000104        0.000066        0.000066
std          0.001428        0.001428        0.001338        0.001338
min         -0.004092       -0.004092       -0.004083       -0.004083
25%         -0.000905       -0.000905       -0.000866       -0.000866
50%          0.000092        0.000092        0.000091        0.000091
75%          0.001058        0.001058        0.000963        0.000963
max          0.006249        0.006249        0.004487        0.004487

       X9_Local_Error  X10_Local_Error  …  X16_Local_Error  X17_Local_Error  \
count     2900.000000      2900.000000  …      2900.000000      2900.000000
mean         0.000097         0.000097  …         0.000051         0.000051
std          0.001838         0.001838  …         0.001170         0.001170
min         -0.005435        -0.005435  …        -0.003644        -0.003644
25%         -0.001138        -0.001138  …        -0.000750        -0.000750
50%          0.000055         0.000055  …         0.000066         0.000066
75%          0.001354         0.001354  …         0.000808         0.000808
max          0.006360         0.006360  …         0.004139         0.004139

       X18_Local_Error  X19_Local_Error  X20_Local_Error  X21_Local_Error  \
count      2900.000000      2900.000000      2900.000000      2900.000000
mean          0.000039         0.000039         0.000024         0.000023
std           0.001116         0.001106         0.001220         0.001178
min          -0.003312        -0.003281        -0.003969        -0.003831
25%          -0.000694        -0.000687        -0.000780        -0.000753
50%           0.000006         0.000006        -0.000006        -0.000006
75%           0.000731         0.000724         0.000862         0.000832
max           0.004244         0.004204         0.004140         0.003996

       X22_Local_Error  X23_Local_Error  X24_Local_Error  X25_Local_Error
count      2900.000000      2900.000000      2900.000000      2900.000000
mean          0.000022         0.000020         0.000019         0.000018
std           0.001112         0.001039         0.000963         0.000907
min          -0.003617        -0.003380        -0.003135        -0.002951
25%          -0.000711        -0.000664        -0.000616        -0.000580
50%          -0.000006        -0.000005        -0.000005        -0.000005
75%           0.000785         0.000734         0.000681         0.000641
max           0.003773         0.003525         0.003270         0.003078

[8 rows x 25 columns]
```

[53]:
```
df_pred.columns= ['X1_pred', 'X2_pred', 'X3_pred','X4_pred', 'X5_pred',␣
↪'X6_pred','X7_pred', 'X8_pred', 'X9_pred','X10_pred', 'X11_pred',␣
↪'X12_pred','X13_pred', 'X14_pred', 'X15_pred','X16_pred', 'X17_pred',␣
↪'X18_pred','X19_pred', 'X20_pred', 'X21_pred','X22_pred', 'X23_pred',␣
↪'X24_pred','X25_pred']
```

[54]:
```
df_pred.head()
```

[54]:

	X1_pred	X2_pred	X3_pred	X4_pred	X5_pred	X6_pred	X7_pred	\
0	0.063513	0.064432	0.064336	0.064477	0.064477	0.064477	0.068031	
1	0.063574	0.064491	0.064394	0.064175	0.064175	0.064175	0.067069	
2	0.063635	0.064584	0.064488	0.064442	0.064442	0.064442	0.066378	
3	0.063936	0.064508	0.064411	0.065368	0.065368	0.065368	0.066209	
4	0.064340	0.064329	0.064233	0.066430	0.066430	0.066430	0.066684	

	X8_pred	X9_pred	X10_pred	…	X16_pred	X17_pred	X18_pred	X19_pred	\
0	0.068031	0.070078	0.070078	…	0.071188	0.071188	0.073350	0.072662	
1	0.067069	0.070186	0.070186	…	0.071690	0.071690	0.073003	0.072319	
2	0.066378	0.070209	0.070209	…	0.072061	0.072061	0.072640	0.071959	
3	0.066209	0.070080	0.070080	…	0.072124	0.072124	0.072269	0.071592	
4	0.066684	0.069771	0.069771	…	0.071900	0.071900	0.071884	0.071210	

	X20_pred	X21_pred	X22_pred	X23_pred	X24_pred	X25_pred
0	0.070718	0.068263	0.064444	0.060215	0.055850	0.052577
1	0.070960	0.068497	0.064664	0.060422	0.056042	0.052757
2	0.071125	0.068656	0.064815	0.060562	0.056172	0.052880
3	0.071231	0.068758	0.064912	0.060652	0.056256	0.052959
4	0.071291	0.068816	0.064966	0.060704	0.056303	0.053003

[5 rows x 25 columns]

[55]:
```
test_out.columns = ['X1_test', 'X2_test', 'X3_test','X4_test', 'X5_test',␣
↪'X6_test','X7_test', 'X8_test', 'X9_test','X10_test', 'X11_test',␣
↪'X12_test','X13_test', 'X14_test', 'X15_test','X16_test', 'X17_test',␣
↪'X18_test','X19_test', 'X20_test', 'X21_test','X22_test', 'X23_test',␣
↪'X24_test','X25_test']
```

[56]:
```
df_out = pd.concat([df_pred, test_out], axis = 1)
```

[58]:
```
df_out.to_excel(r'C:\Users\INFO-DSK-02\Desktop\Lorentz Multi Dimension␣
↪Prediction-Phase-2\Final_Version\3D_ReservoirComputing\Output\25_Dim_Preds\Output_25_Dim.
↪xlsx', index= False)
```