COMP7409 Project

Shourya Mehra (UID 3035345411)

Group H

Table of Contents

Table of Figures

Table of Tables

1 Introduction

Financial forecasting is a popular area of research, and a particularly interesting one because of the number of variables that come into play to price any financial instrument at any time. These variables could be a combination of macroeconomic factors, investor sentiment, company news, political events, technical indicators, and fundamental analysis. The relations between these variables (if any) are so complex that it is virtually impossible to come up with an exact prediction of the price of any financial instrument, let alone doing it over and over.

Deep learning and machine learning have been extensively applied to this problem and have offered some promising results. Using neural networks capable of coming up with any kind of non-linear function by creating a complex network of neurons allows for the creation of models which can offer near accurate price predictions while abstracting out the model construction complexity. However, popular models such as LSTM that have been widely employed for this process suffer from 2 shortcomings- first, a long training time because of densely connected layers resulting in many parameters to train and second, the inability to accurately forecast prices of instruments with high volatility. This has sometimes been attributed to overfitting and to overcome these pitfalls other models have been proposed. One such model is the RNN family's Echo State Network (ESN) [1] based on the Reservoir Computing (RC) [2] architecture. The aim of this project is to compare the performance of RC based ESN to that of LSTM in financial forecasting. The 2 models will be trained and tested on currencies, equities, indices, and cryptocurrencies with varying levels of volatility and their performance will be compared based on MSE error and time taken for training.

The report has been divided as follows: section 2 will explain the architecture of the Echo State Network and Reservoir Computing, section 3 will explain the code, section 4 will discuss the experiments and results, and section 5 will offer a conclusion. The workings of LSTM and the code for the same has not been discussed in the report as it has already been extensively covered in class. In fact, the LSTM model used is the same as discussed in the lecture, except that we use a different period for the data.

In terms of my contributions to this group project, I, with the constant guidance and support of my groupmates, have worked on developing the code, training the models, tuning the hyperparameters for ESN, and generating the evaluations and visualizations of the results on both LSTM and ESN.

2 Reservoir Computing and Echo State Networks

2.1 ESN

ESN belongs to the RNN family and is characterized by its dynamic memory and feedback loops in its pathways. It makes use of a large RNN of 50 to 1000 neurons and only permits modification of neurons in the layer preceding the output layer. This eliminates any cyclic dependencies between trained readout connections, thereby making the training process an efficient linear regression problem [2].

The ESN construction begins by creating a random RNN with 50-1000 neurons inside it. This is referred to as the reservoir. The reservoir is connected to an output neuron with random connections feeding back into the reservoir.

A teacher sequence (train data) is fed into the output neuron. Through the output feedback connections, the internal neurons get excited and slowly start to exhibit properties of the data. This allows the internal neurons to work as echo functions for the data.

By allowing for a sparse connectivity of the internal neurons, the reservoir breaks up into several loosely coupled systems which allows it to capture properties of the time series it is being trained to predict.

After the reservoir has been created, the weights of the output neuron are trained by minimizing the Mean Square Error Loss between the model prediction and actual value (formula provided below). This is achieved by using Ridge Regression.

$$MSE_{train} = \left(\frac{1}{n}\right) * \left(d(n) - \sum w_i x_i(n)\right)^2$$

$$where\ d(n)\ is\ actual\ price$$

$$w_i's\ are\ the\ otuput\ layer\ weights$$

$$x_i(n)\ is\ the\ reservoir\ representation\ of\ nth\ training\ samples$$

$$n\ is\ the\ number\ of\ training\ samples$$

This combination of the Reservoir and the Linear Prediction model is referred to as the ESN.

The ESN model in [2] improved performance on its dataset by a factor of 2400 compared to previous techniques and analytically showed that the jump in modelling accuracy was because ESN's can capture a lot of short-term memory.

From the above discussion we can see that ESN offers 2 main advantages: first, the ability to capture an abundant short-term memory, allowing it to adapt to volatile market conditions, and second, a linear regression training task with minimal parameters, resulting in very fast training, thereby enabling testing of several models quickly.

2.2 Reservoir Computing

Theoretically, RNN's offer a promising solution to model dynamic and chaotic systems. However, the difficulty in training their weights to reach a global optimum makes it difficult to realize the theoretical capability in practice. RNN's suffer from the vanishing gradient problem while training and although the LSTM architecture can overcome this problem, it comes at a cost of higher computational complexity and training time.

Reservoir model spaces are an unsupervised approach to learning a fixed vector representation of a time series. They convert the training process from neural network backpropagation to a simple linear regression task, thus offering much faster training speeds. The work we use from [2] offers a better representation for the vector space and uses PCA for dimensionality reduction to improve computational performance.

The main idea of the RC framework is to generate the recurrent part of the RNN randomly and keep it fixed. This fixed representation carries all the necessary information to reconstruct the time-series from the vector representation. This leads to the idea that a system that can predict the next phase of the reservoir, i.e., the output layer, can provide an accurate time series forecasting model. Using a very large reservoir can lead the model to overfit and PCA is used to reduce the reservoir dimensions to tackle this problem.

The process begins by using a matrix representation of the time series (the closing price data for this case) as below:

$$X = [x(1), x(2) \ldots . x(T)]^T$$

An encoder-decoder sequence is used, where the encoder converts the input into a vector representation and the decoder functions as the prediction model. In this case, a ridge regression model will be applied for the prediction.

The encoder used to model the sequential time-series data takes inspiration from [3] and takes the following form:

$$\mathrm{h(t)} = \mathrm{f\,(x(t);\ h(t-1);\ \Theta enc)}$$

where h(t) is the state of the RNN at time t and is dependent on the RNN state at time t-1 and the current input x(t). $\Theta\mathrm{enc}$ are parameters that can be trained. The function f is a non-linear activation function such as tanh or sigmoid. The input and previous state vector representations are multiplied by matrices of weights W_{in} (input weights) and W_t (internal reservoir weights) which adjust the effect each of these have on the current state.

Using the RNN states at each time, $H = [h(1), h(2), \ldots \ldots h(t)]$, an input representation of the input X can be created such that r_x = r(H). Choosing r_x = h(t) is a good option because the information embedded in the last state will enable recreation of the input.

In the case of RNN's, the weights W_{in} and W_t can be trained. This is what makes the RNN training so complex and time-consuming. RC's take a different approach in that they randomly generate these weights and leave them untrained. The reservoir instead generates a pool of dynamic generalized features which can tackle many different problems. By using a large recurrent network, sparse connectivity, and a spectral radius, the chaotic reservoir model is brought to stability to adapt to the time series it is presently being employed on.

The following hyperparameters can be used to adapt the system:

i. Spectral radius: largest eigenvalue of the reservoir matrix of connection weights
ii. Connectivity: Percentage of nonzero connection weights
iii. Reservoir Units: The number of neurons in the Reservoir
iv. Input Scaling: Controls non-linearity in the system by scaling down the input weights
v. Gaussian Noise: Used for regularization purposes.

2.2.1 Reservoir Architecture and Formulation

Reservoir Computing creates several features that can be used to model the problem and forecast the next step of the time series. Initial versions pick only the useful features while discarding the others. This can create bias in the system because there could be cases where the discarded features could have been useful to predict the one step ahead system dynamics. Thus, it is better to not discard such features and instead employ a dimensionality reduction to simplify the system after it has been created.

The ridge regression model predicts the next step using $h(t+1) = U_h h(t) + u_h$.

The reservoir representation is implemented as $r_X = \Theta_h = [vec(U_h); u_h] \in \mathbb{R}^{R(R+1)}$

In [2], the authors also extend the formulation to a bidirectional reservoir which can learn about dependencies very far into the future. This is implemented in time series forecasting by providing future information during training. This has been proven to improve the prediction accuracy of the model [4]. Bidirectionality is achieved by feeding an input sequence in the forward and reverse order into the same reservoir as shown below:

$$h(t)_{forward} = f(W_{in}x(t) + W_r h(t-1)_{forward}$$

$$h(t)_{backward} = f(W_{in}x(T-t) + W_r h(t-1)_{backward}$$

The overall representation is a concatenation of the above 2 vectors. This kind of representation enables the model to encompass recent and past information, allowing for capture of long-term dependencies and better prediction accuracy.

The formulation of the linear model then changes to $[h(t+1)_{forward}, h(t+1)_{backward} = U_h[h(t)_{forward}, h(t)_{backward}] + u_h$. Now, $U_h \in \mathbb{R}^{2Rx2R}$ and $u_h \in \mathbb{R}^{2R}$. The linear model is now optimized for 2 separate problems: predicting the next time step or reproducing the previous one. This can result in better accuracy because the model is trained to model dependencies in both directions.

2.2.2 Dimensionality Reduction

The high dimensionality would make the number of parameters used for prediction too large, making the reservoir hard to deal with. This can result in overfitting and requirement of large computational resources for the ridge regression model. Applying PCA to reduce dimensions of the last reservoir state can improve performance [5]. Further, it allows the RC model to generalize and is efficient to compute as it is linearly formulated [6].

The method in [2] describes the dataset as a 3-mode tensor $H^{N x T x R}$ and transforms the dimension R to D (D<<R) while keeping the other 2 dimensions unaltered. This is equivalent to applying a two-dimensional PCA on A. The mode-3 matricization of A is computed to create a resulting matrix $A_{(3)} = A^{NT x R}$. Then, the rows of this matrix are projected on eigenvectors of the D largest eigenvalues of a covariance matrix $C \in \mathbb{R}^{R x R}$ as shown below:

$$C = \left(\frac{1}{NT-1}\right) * \sum_{i=1}^{i=NT} (h_i - h^-)(h_i - h^-)^T$$

In the above equation, h_i is the i-th row of $A_{(3)}$ and $h^- = \left(\frac{1}{N}\right) \sum_{i=1}^{i=NT} h_i$. Because the first dimensions in A have been combined, the covariance matrix C will evaluate variation in all samples and timesteps at the same time which can lead to loss of dataset structure and orderings. [2] suggests an alternate method to address this issue. They consider several individual sample prices of the

form $H_n \in \mathbb{R}^{TxR}$. This is accomplished by slicing A across its first dimension. The following covariance matrix is then employed for dimensionality reduction [2]:

$$S = \left(\frac{1}{N-1}\right) * \sum_{i=1}^{i=N} (H_n - H^-)^T (H_n - H^-)$$

Again, the D largest eigenvectors are selected. These are now combined into a matrix $E \in \mathbb{R}^{RxD}$ and the dimensionality reduction task is completed by computing $H^ = H \times_3 E$ ($\times_3$ represents 3-mode product). This method preserves the temporal orderings in the reservoir by treating all sequences of reservoir states as a single observation [2].

The model is now transformed to $h^(t+1) = U_h h^(t) + u_h$ where the $\hat{h}(.)$ values are the columns of a frontal slice of reduced matrix H^. Now, $U_h \in \mathbb{R}^{DxD}$ and $u_h \in \mathbb{R}^D$. And we also have $rx = \Theta_h = [vec(U_h); u_h] \in \mathbb{R}^{D(D+1)}$ [2].

Figure 1 below outlines the entire reservoir model space representation. For the time series input X[n], a sequence of states H[n] are generated. Using dimensionality reduction, the features in the reservoir are reduced from R to D. A separate model is applied to predict H^[n], the n^{th} slice of the dimensionally reduced H^. The parameters $\Theta_h[n]$ become the representation of the input X[n].

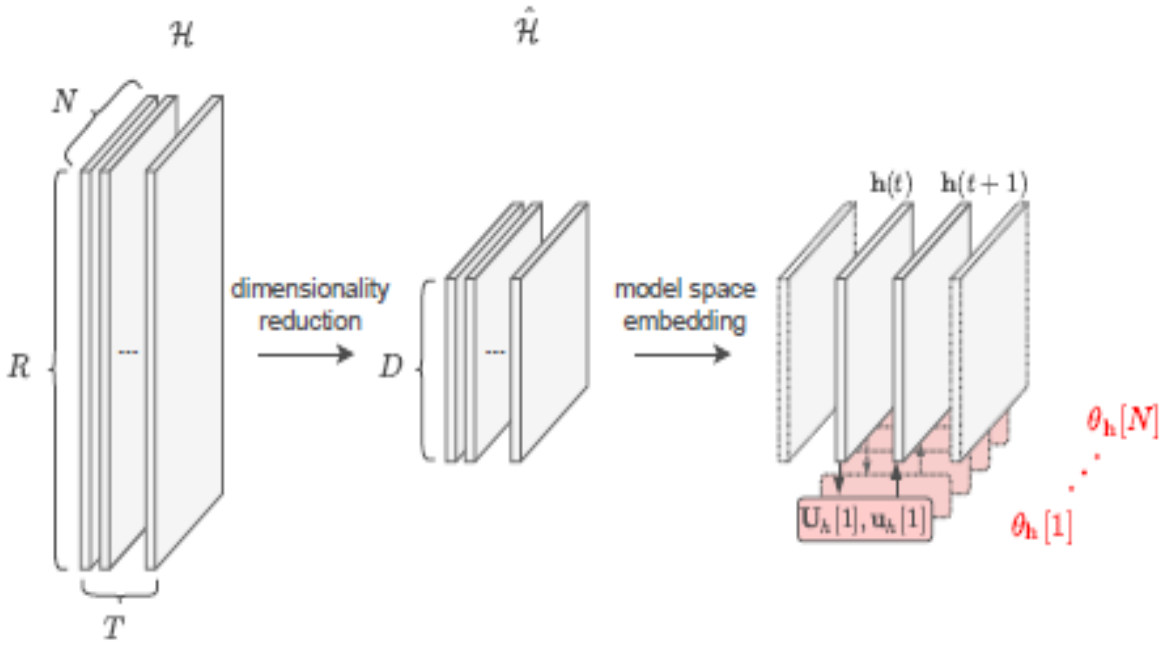

Figure 1: Generating the Reservoir Model Space Representation [2]

2.2.3 Summarizing the RC Framework

Figure 2 below outlines the entire Reservoir Computing Framework that has been explained above. The Encoder module generates a Reservoir representation of the input and employs dimensionality reduction to improve performance. The Decoder module can be used for classification or prediction tasks. For the case of financial forecasting, a ridge regression model is trained on the last state of the reservoir to forecast prices of the financial instruments.

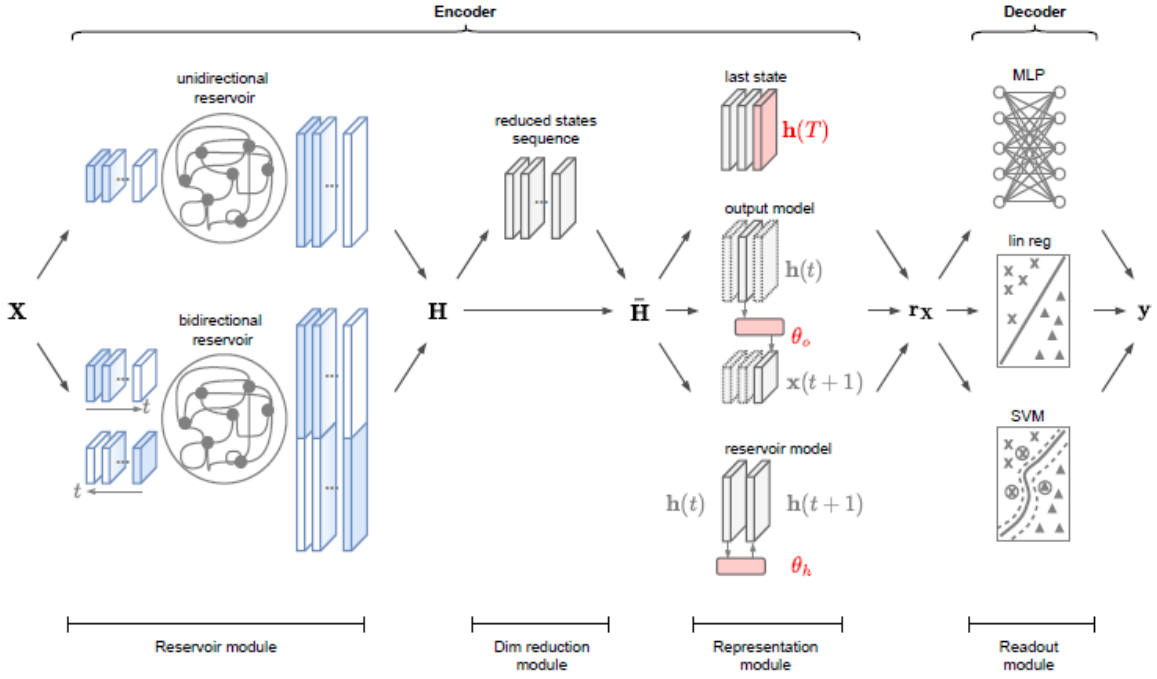

Figure 2: Overall RC Framework [2]

3 Code Explanation

This section will explain the code and is divided into 3 Parts: Data Preparation Pipeline, ESN-Reservoir Model Implementation, and Model Evaluation Framework.

3.1 Data Preparation Pipeline

3.1.1 Obtaining the Data

Data is obtained from yahoo finance by using the pandas datareader as shown in Figure 3. Data is obtained from 2nd January 2020 to 28th February 2022. and then the closing price data points are retained. Figure 4 shows the raw data obtained for BTC.

```
start = datetime.strptime('2020-01-02','%Y-%m-%d')
end = datetime.strptime('2022-02-28','%Y-%m-%d')

df = data.DataReader('BTC-USD',start=start, end=end, data_source='yahoo')
df = df['Close']
```

Figure 3: Obtaining Price Data

```
Date
2020-01-01     7200.174316
2020-01-02     6985.470215
2020-01-03     7344.884277
2020-01-04     7410.656738
2020-01-05     7411.317383
                   ...
2022-02-24    38332.609375
2022-02-25    39214.218750
2022-02-26    39105.148438
2022-02-27    37709.785156
2022-02-28    43193.234375
Name: Close, Length: 790, dtype: float64
```

Figure 4: Raw Data Obtained

3.1.2 Feature Scaling and Reshaping

The MinMaxScaler in sklearn is used to transform the price values to a range of 0 to 1. The following formula is used: (x-x_min)/(x_max-x_min). Next, the price data is reshaped by calling a numpy function. The reshaped array has 1 column (like before), but now each sample is in 1 row. This data structure is a requirement for inputting into the Neural Network model. The data will look like Figure 5 after the steps have been carried out:

```
array([[0.03561545],
       [0.03218546],
       [0.03792726],
       [0.038978  ],
       [0.03898856],
       [0.0447062 ],
       [0.05100809],
       [0.04966887],
       [0.04646114],
       [0.05105381]])
```

Figure 5: Data Shape After Feature Scaling

3.1.3 Train-Test Split

65% of data is used for training. Rest will be used for testing

3.1.4 Dividing the Data into X and y

X is the independent data and y is the dependent data. The model will use the X data to predict the y value. For this case, we will use 100 days data to predict the price on the 101st day. Data for 100 days is taken and input as an array into X. Data for the 101st day is taken and input as a scalar into y. The procedure is done for training and testing data.

```
def create_dataset(dataset, time_step=1):
    dataX, dataY = [], []
    for i in range(len(dataset)-time_step-1):
        a = dataset[i:(i+time_step), 0]
        dataX.append(a)
        dataY.append(dataset[i+time_step,0])
    return np.array(dataX), np.array(dataY)



time_step = 100
X_train, y_train = create_dataset(train_data, time_step)
X_test, y_test = create_dataset(test_data, time_step)
```

Figure 6: Creating the X and y Data

Figure 7 shows the structure of 1st two rows of X_train data. X_train is a 2D array with 100 values per row. A part of Y_train, as visible in Figure 8, shows that it is a 1D array with 1 value per row. The number of rows in X_train and y_train is the same.

```
array([[0.03561545, 0.03218546, 0.03792726, 0.038978  , 0.03898856,
        0.0447062 , 0.05100809, 0.04966887, 0.04646114, 0.05105381,
        0.04899271, 0.05146821, 0.0506966 , 0.06161694, 0.06128539,
        0.05995584, 0.06323483, 0.06345482, 0.05967561, 0.05889917,
        0.06030903, 0.05927033, 0.0548873 , 0.05550905, 0.05426956,
        0.05792766, 0.0629278 , 0.07009711, 0.06942678, 0.07249987,
        0.06996834, 0.07064483, 0.06986987, 0.06905762, 0.06725944,
        0.0741682 , 0.07602739, 0.07708403, 0.07818915, 0.08220785,
        0.07805323, 0.0836706 , 0.0855528 , 0.08376875, 0.08533013,
        0.07857744, 0.07929648, 0.07539382, 0.08261238, 0.07448712,
        0.07408915, 0.07533469, 0.07496311, 0.07913803, 0.07475532,
        0.06982737, 0.06150124, 0.06092568, 0.0591358 , 0.05797045,
        0.05737849, 0.0622864 , 0.06097827, 0.06045843, 0.06562675,
        0.06632621, 0.06292996, 0.05012023, 0.04717321, 0.04695092,
        0.04697809, 0.        , 0.00947215, 0.00366761, 0.00673408,
        0.000698  , 0.0040712 , 0.00427584, 0.01949652, 0.0196177 ,
        0.01939864, 0.01373037, 0.02309294, 0.02818095, 0.02732241,
        0.02788758, 0.02394737, 0.02031128, 0.01519673, 0.02330904,
        0.02344967, 0.02613565, 0.02912063, 0.02815832, 0.03030127,
        0.02908077, 0.0367594 , 0.03523588, 0.03775495, 0.03724359],
       [0.03218546, 0.03792726, 0.038978  , 0.03898856, 0.0447062 ,
        0.05100809, 0.04966887, 0.04646114, 0.05105381, 0.04899271,
        0.05146821, 0.0506966 , 0.06161694, 0.06128539, 0.05995584,
        0.06323483, 0.06345482, 0.05967561, 0.05889917, 0.06030903,
        0.05927033, 0.0548873 , 0.05550905, 0.05426956, 0.05792766,
...
        0.0040712 , 0.00427584, 0.01949652, 0.0196177 , 0.01939864,
        0.01373037, 0.02309294, 0.02818095, 0.02732241, 0.02788758,
        0.02394737, 0.02031128, 0.01519673, 0.02330904, 0.02344967,
        0.02613565, 0.02912063, 0.02815832, 0.03030127, 0.02908077,
        0.0367594 , 0.03523588, 0.03775495, 0.03724359, 0.03026877]])
```

Figure 7: X_train

```
array([0.03026877, 0.03016636, 0.03195575, 0.02994198, 0.02990029,
       0.02670012, 0.03428358, 0.03395417, 0.03653389, 0.03544372])
```

Figure 8: y_train

3.1.5 Final Reshaping

The X data will be reshaped once again such that each data sample is an array instead of a scalar. This is a Keras requirement for LSTM and for consistency, we use the same pipeline for ESN. The data, as shown in Figure 9, is a 3D array with a 1*1 inner most array, and a 100*1 middle array.

```
array([[[0.03561545],
        [0.03218546],
        [0.03792726],
        [0.038978  ],
        [0.03898856],
        [0.0447062 ],
        [0.05100809],
        [0.04966887],
        [0.04646114],
        [0.05105381],
        [0.04899271],
        [0.05146821],
        [0.0506966 ],
        [0.06161694],
        [0.06128539],
        [0.05995584],
        [0.06323483],
        [0.06345482],
        [0.05967561],
        [0.05889917],
        [0.06030903],
        [0.05927033],
        [0.0548873 ],
        [0.05550905],
        [0.05426956],
...
        [0.0367594 ],
        [0.03523588],
        [0.03775495],
        [0.03724359],
        [0.03026877]]])
```

Figure 9: Train Data After Preparation

3.2 ESN- Reservoir Model Implementation

The code to implement the reservoir architecture has been obtained from [7], where a simplified version of the overall architecture has been created by referencing the original implementation provided by the authors of the ESN paper. Using RC, an ESN is created by combining it with a ridge regression prediction. This sub-section will describe the procedure in detail.

3.2.1 Reservoir Class Basic Structure and Initialization

The reservoir architecture is implemented as a class which takes the following parameters as inputs:

a. n_internal_units = processing units in the reservoir
b. spectral_radius = largest eigenvalue of the reservoir matrix of connection weights
c. leak = amount of leakage in the reservoir state update (optional)
d. connectivity = percentage of nonzero connection weights (unused in circle reservoir)
e. input_scaling = scaling of the input connection weights
f. noise_level = deviation of the Gaussian noise injected in the state update
g. circle = generate deterministic reservoir with circle topology

When a class object is first created, the input weights are set to None. The internal reservoir weights are initialized randomly using the method described in 3.2.2. All the input parameters come with default values and will be set to those values if those parameters are not provided in the object initialization.

3.2.2 Internal Weight Initialization

The internal weight initialization of the RC depends on whether the circular topology is used. As mentioned previously, these weights will be fixed and are not available for training.

a. With Circular Topology

Figure 10 shows the code for internal weight initialization when circular topology is used. Start by creating a 2D matrix of size n_internal_units * n_internal_units. All values in this matrix are set to 0. Set the last value of the first row = spectral radius. Then, go through all the rows and set the i^{th} value in the $(i+1)^{th}$ row = spectral radius. The result will be an array with a sort of circular organization.

```
def _initialize_internal_weights_Circ(self, n_internal_units, spectral_radius):

    internal_weights = np.zeros((n_internal_units, n_internal_units))
    internal_weights[0,-1] = spectral_radius
    for i in range(n_internal_units-1):
        internal_weights[i+1,i] = spectral_radius

    return internal_weights
```

Figure 10: Weight Initialization with Circular Topology

An example with n_internal_units = 4 and spectral_radius = 0.1 is provided below.

$$\begin{matrix} 0 & 0 & 0 & 0.1 \\ 0.1 & 0 & 0 & 0 \\ 0 & 0.1 & 0 & 0 \\ 0 & 0 & 0.1 & 0 \end{matrix}$$

b. Without Circular Topology

Figure 11 shows the code for internal weight initialization when circular topology is not used. Start by creating a sparse matrix (a matrix where most values are 0) by using the sparse function in the scipy python library. The rand function is used to generate the samples from a random distribution, i.e. belonging to [0,1]. The sparseness will be determined by the connectivity parameter- a smaller connectivity will mean more zero values. This sparse matrix is then converted to the dense format to be used as the weight vector with the data. Then, 0.5 is subtracted from all the non-zero weights to make the weight value range [-0.5, 0.5]. Next, the eigen values and eigen vectors of the internal_weights vector are computed by using a numpy

function. The maximum absolute eigen value is found and each non-zero value in the internal_weights vector is divided by this maximum absolute eigen value and then multiplied by the spectral radius. This is done to adjust the weights according to the spectral radius provided.

```
def _initialize_internal_weights(self, n_internal_units,
                                 connectivity, spectral_radius):

    # Generate sparse, uniformly distributed weights.
    internal_weights = sparse.rand(n_internal_units,
                                   n_internal_units,
                                   density=connectivity).todense()

    # Ensure that the nonzero values are uniformly distributed in [-0.5, 0.5]
    internal_weights[np.where(internal_weights > 0)] -= 0.5

    # Adjust the spectral radius.
    E, _ = np.linalg.eig(internal_weights)
    e_max = np.max(np.abs(E))
    internal_weights /= np.abs(e_max)/spectral_radius

    return internal_weights
```

Figure 11: Weight Initialization without Circular Topology

3.2.3 Generate Reservoir Representation

3.2.3.1 Get Reservoir Embedding

The getReservoirEmbedding method is called and it takes the following parameters:

a. X: Data
b. Pca: Pca object
c. Ridge_embedding: Ridge Regression embedding required for building predictor
d. N_drop: number of units to be dropped out when computing next state in RNN. Helps in preventing overfitting.
e. Bidir: Boolean to specify whether a time reversed, or bidirectional reservoir is being employed (explained in 2.2.1) (True if bidirectional reservoir used, False otherwise)
f. Test: Boolean value to specify whether train or test data is provided (True for test, False otherwise)

Figure 12 shows the getReservoirEmbedding method. It first calls the getStates method (described below in 3.2.3.2) to obtain the reservoir state matrix. It then gets the number of samples in the data. Next, it reshapes the output of the reservoir state to a 2D matrix as described in section 2.2.2 to apply PCA to it. Depending on whether it is test data or not, the PCA's fit transform or transform method is applied to the data for dimensionality reduction. The reduced states data is then reshaped back to the original format of the reservoir representation, only that now it has much lesser dimensions. The ridge regression embedding of the model is then created and the concatenated combination of the slope and intercept is returned as the overall reservoir representation of the data.

```
def getReservoirEmbedding(self, X,pca, ridge_embedding,  n_drop=5, bidir=True, test = False):

    res_states = self.get_states(X, n_drop=5, bidir=True)


    N_samples = res_states.shape[0]
    res_states = res_states.reshape(-1, res_states.shape[2])
    # ..transform..
    if test:
        red_states = pca.transform(res_states)
    else:
        red_states = pca.fit_transform(res_states)
    # ..and put back in tensor form
    red_states = red_states.reshape(N_samples,-1,red_states.shape[1])

    coeff_tr = []
    biases_tr = []

    for i in range(X.shape[0]):
        ridge_embedding.fit(red_states[i, 0:-1, :], red_states[i, 1:, :])
        coeff_tr.append(ridge_embedding.coef_.ravel())
        biases_tr.append(ridge_embedding.intercept_.ravel())
    print(np.array(coeff_tr).shape,np.array(biases_tr).shape)
    input_repr = np.concatenate((np.vstack(coeff_tr), np.vstack(biases_tr)), axis=1)
    return input_repr
```

Figure 12: getReservoirEmbedding method

3.2.3.2 Get Reservoir States

The get_states method is shown in Figure 13. The shape of the 3D array is stored in variables N, T, V for later use. N is the number of samples, T is time series length of each sample, V is the innermost array and for our case has a default size of 1 as described in Figure 9. The input weights are then initialized by drawing samples from a Binomial Distribution with number of trials (n) = 1, probability of occurrence of a trial (p) = 0.5. The third parameter is the size of the array to be returned- we request a 2D array of size n_internal_units*1. The inputs are multiplied by 2 and then scaled down depending on the input scaling factor parameter provided. The smaller this value, the smaller the impact of the input weights on the overall model. The data with the dropout parameter is fed into the _compute_state_matrix method (described below in 3.2.3.3) which trains the reservoir state. If no bidirectional reservoir is used, this state is returned to the getReservoirEmbedding method for further processing into PCA and Ridge Regression as described above in 3.2.3.1. If a bidirectional reservoir is used, the data is reversed and again fed into the _compute_state_matrix method to create the reversed state. The state and reversed state matrices are concatenated and the combination is returned to getReservoirEmbedding for further processing.

```
def get_states(self, X, n_drop=0, bidir=True):
    N, T, V = X.shape
    if self._input_weights is None:
        self._input_weights = (2.0*np.random.binomial(1, 0.5 , [self._n_internal_units, V]) - 1.0)*self._input_scaling

    # compute sequence of reservoir states
    states = self._compute_state_matrix(X, n_drop)

    # reservoir states on time reversed input
    if bidir is True:
        X_r = X[:, ::-1, :]
        states_r = self._compute_state_matrix(X_r, n_drop)
        states = np.concatenate((states, states_r), axis=2)

    return states
```

Figure 13: getStates method

3.2.3.3 *Compute State Matrix*

This method handles the training of the reservoir state and is shown in Figure 14. The method begins by initializing some values. It stores the shape of the input data. N is the number of samples, T is time series length of each sample, and _ is the innermost array and for our case has a default size of 1 as described in Figure 9. The states are updated using equations in section 2.2. The current state matrix depends on the previous state and the current input sample. The previous states vector is initialized to an array of zeros with a size of N*(T-n_drops)*n_internal_units. We reduce the second dimension using the n_drop argument as a regularization technique to reduce overfitting and generalize the model. The state matrix is then initialized using an empty array of the same size as the previous state vector.

Then, we iterate through each step in the time series. The current input is the vector containing the value of the t^{th} time step in each time series. The current state is calculated by adding the dot product of internal weights and the previous state to the dot product of input weights and current input. Some random noise is added to this current state by sampling weights from a standard normal and multiplying them by a noise level defined in the class input parameters. Next, if no leakage is needed, the tanh function is applied to the current state and assigned as the previous state vector to prepare for the next iteration. Otherwise, the previous state is updated by multiplying it by 1-leakage. To this, the tanh function output of the current state is added. The previous state is then updated with this value. If more iterations have passed than the specified dropout value, the state matrix at position iteration - dropout is assigned the previous state value.

After the iteration is complete, the state matrix is returned to the get_state function for further processing as described in section 3.2.3.2 above.

```
def _compute_state_matrix(self, X, n_drop=0):
    N, T, _ = X.shape
    previous_state = np.zeros((N, self._n_internal_units), dtype=float)

    # Storage
    state_matrix = np.empty((N, T - n_drop, self._n_internal_units), dtype=float)
    for t in range(T):
        current_input = X[:, t, :]

        # Calculate state
        state_before_tanh = self._internal_weights.dot(previous_state.T) + self._input_weights.dot(current_input.T)

        # Add noise
        state_before_tanh += np.random.rand(self._n_internal_units, N)*self._noise_level

        # Apply nonlinearity and leakage (optional)
        if self._leak is None:
            previous_state = np.tanh(state_before_tanh).T
        else:
            previous_state = (1.0 - self._leak)*previous_state + np.tanh(state_before_tanh).T

        # Store everything after the dropout period
        if (t > n_drop - 1):
            state_matrix[:, t - n_drop, :] = previous_state

    return state_matrix
```

Figure 14: _compute_state_matrix method

3.2.4 ESN Implementation

Figure 15 outlines the implementation of ESN. A PCA object is initialized with the number of components specified. The number of components is how many features we desire after

dimensionality reduction. Next, a ridge regression estimator is initialized, and the alpha value is specified. Alpha balances amount of emphasis given to minimizing RSS vs minimizing sum of square of coefficients. Alpha can take any value between 0 and infinity. If alpha is 0, the coefficients will be same as linear regression. If alpha is infinity, the coefficients will be 0. Any other value between 0 and infinity will provide a value between 0 and 1 for linear regression. Next, a reservoir object is initialized with inputs as described in 3.2.1. Reservoir representations of the X_train and X_test data are created, and the fit method is called on the reservoir representation of the X_train data and the y_train data to create the prediction model.

```
pca = PCA(n_components=100)
ridge_embedding = Ridge(alpha=20, fit_intercept=True)
readout = Ridge(alpha=20)

res = Reservoir(n_internal_units=300, spectral_radius=0.2, leak=0.2,
                connectivity=0.85, input_scaling=0.1, noise_level=0.02, circle=False)

input_repr = res.getReservoirEmbedding(X_train,pca, ridge_embedding,  n_drop=20, bidir=True, test = False)
input_repr_te = res.getReservoirEmbedding(X_test,pca, ridge_embedding,  n_drop=20, bidir=True, test = True)

readout.fit(input_repr,y_train)
```

Figure 15: ESN Implementation

3.3 Model Evaluation Framework

This section outlines the framework created to compare ESN and LSTM models qualitatively and quantitatively. Section 4 will use this framework to describe the results of the experiments.

3.3.1 Visualization

The model is used to make predictions on the train and test data. The data is transformed back to its original values by applying the inverse_transform function. We had converted the data to a range of 0 to 1 using the min-max scaler. The inverse of the same scaler is applied to convert the price data to its original value. Line plots are created which show the actual price data and the prediction on the test set. This is done for both LSTM and ESN for a qualitative comparison of model performance.

3.3.2 Quantitative Evaluation

a. Mean Squared Error (MSE): The mean squared error is calculated for the prediction result and the actual y_test data for both LSTM and ESN. A lower mean squared error indicates a better performance because it means that the model's prediction results closely resemble the actual price data. To quantitatively evaluate the Mean Squared Error for different financial instruments with varying price levels and volatility, the mean squared error is calculated on the scaled data without applying the inverse_transform method.
b. Training Time: The training time taken for both LSTM and ESN is evaluated for comparison.

4 Experiments

LSTM and ESN models were created for different financial instruments with different levels of volatility. This section will describe the model features for each instrument and then qualitatively and quantitatively compare the performance of ESN and LSTM.

4.1 Financial Instruments

Experiments were conducted on forex, index funds, cryptocurrencies, and equities. The following instruments have been used for the experiments:

1. EUR/USD: Forex currency pair of Euro and US Dollar. Low volatility instrument
2. Hang Seng Index (HSI): Index of major corporations listed on HK Stock Exchange. Moderate volatility instrument.
3. Bitcoin (BTC-USD): Largest cryptocurrency by market cap. High level of volatility.
4. Ethereum (ETH-USD): Second largest cryptocurrency by market cap. High level of volatility.
5. GameStop (GME): Extremely volatile equity. Preferred by speculators as it has high levels of retail interest and is considered a meme stock.

The volatility values for the instruments are provided in Table 1. The annual volatility is calculated using the formulas below:

$$Daily\ Price\ Change = \log\left(\frac{Price[i]}{Price[i-1]}\right)$$

$$Annual\ Volatility = \sqrt{252} * StandardDeviation(Daily\ Price\ Change)$$

We multiply the standard deviation of Daily Price Change by the root of 252 to annualize the values because there are approximately 252 trading days in a year.

Instrument	EUR/USD	HSI	BTC-USD	ETH-USD	GME
Volatility	6.6%	21.9%	64.7%	85.2%	178.8%

Table 1: Annual Volatility for Financial Instruments

4.2 Experimental Setup

Table 2 below summarizes the ESN parameters selected for each instrument. The selection was done by simple trial and error by trying out several different combinations using an exhaustive brute force approach which tried various combinations and calculated the MSE. The combination with the least MSE was then selected.

It was observed that a small leak, a small spectral radius, and a small input scaling provided the best results. The noise level didn't impact the output too much indicating that the model was generalized enough. However, increasing the dropout from 5 to 10 to 20 improved performances. A high connectivity of 0.85 provided strong results maybe because of the correlations between the daily prices of an instrument. A higher number of components were needed for the more volatile crypto and equity instruments and hence the PCA value selected was higher for them.

Instrument	**PCA Components**	**Ridge Alpha**	**Internal units**	**Spectral radius**	**Leak**	**Connectivity**	**Input Scaling**	**Noise**	**Dropout**
EUR/USD	40	30	200	0.2	0.1	0.85	0.1	0.02	20
HSI	50	10	100	0.2	0.2	0.85	0.1	0.02	20
BTC-USD (Crypto)	100	20	300	0.2	0.2	0.85	0.1	0.02	20
ETH-USD	80	10	150	0.2	0.1	0.85	0.1	0.02	20

GME	100	40	500	0.2	0.1	0.85	0.1	0.02	20

Table 2: ESN Parameters

4.3 Quantitative Comparison of LSTM and ESN based on Mean Squared Error

This comparison explores how closely the model predictions fit the actual price data. As specified in 3.3.2, the MSE value is calculated on the scaled data to be able to compare the MSE for different instruments. A lower MSE value is better because it indicates that the prediction more closely fits the actual price data. Table 3 summarizes the results obtained. For low volatility instruments like EUR/USD and Hang Seng Index, LSTM significantly outperforms ESN. Both models offer similar results for Bitcoin, who's volatility is midway of the 5 instruments. But as the volatility is further increased to 85.2% for Ethereum, the ESN model outperforms LSTM. When the volatility is further increased to 178.8% for Gamestop, ESN significantly outperforms LSTM. Thus, we can conclude that for low volatility instruments LSTM is better, but ESN could be a suitable alternative choice to consider when predicting the price of highly volatile instruments.

Instrument	**LSTM**	**ESN**	**Volatility**
EUR/USD	**0.0054**	0.0372	6.6%
Hang Seng Index	**0.0059**	0.0078	21.9%
Bitcoin	0.0047	0.0050	64.7%
Ethereum	0.0173	**0.0124**	85.2%
Gamestop	0.0097	**0.0064**	178.8%

Table 3: Comparison on LSTM and ESN based on MSE

4.4 Quantitative Comparison of LSTM and ESN based on Training Time

This section explores the training time of the models. Being able to quickly try out different models and tune the hyperparameters might be beneficial to data scientists, traders, and quants. Further, if the model needs to be dynamically updated using current market data, training time might be of utmost importance, especially for high frequency algo trading applications. Table 4 summarizes the training time (in seconds) for LSTM and ESN models. ESN has a significantly shorter training time for all cases. This could be attributed to a few factors. First, ESN has lesser parameters to tune because the reservoir weights are fixed and cannot be trained. Second, the use of PCA aids in dimensionality reduction and further reduces training time for the predictor. Finally, a simple predictor (Ridge Regression) is used which in general has a short training time.

Instrument	**LSTM**	**ESN**
EUR/USD	59.3	**1.1**
Hang Seng Index	49.5	**4.8**
Bitcoin	85.5	**4.2**

Ethereum	82.4	**1.8**
Gamestop	50.8	**0.7**

Table 4: Comparison of LSTM and ESN based on Training Time

4.5 Qualitative Comparison of LSTM and ESN

This section qualitatively evaluates the models by looking at the time series plots generated on the test data. Depending on the application, the MSE might not be the most useful metric to evaluate how well a model performs. Sometimes, even though the prediction value might be way off, the model can predict the price trend well and this could prove useful in discovering local maxima and minima and deploying a buy low sell high trading strategy. Thus, a model that resembles the trend better could be more useful than one which has a lower MSE but with a delayed/ phase shifted prediction.

The results of all 5 cases will be evaluated below in increasing order of volatility. The blue line will be the actual data while the orange line will be the prediction result. LSTM will be on the left while ESN will be on the right.

4.5.1 EUR/USD (Volatility = 6.6%)

Figure 16 shows the charts generated for EUR/USD. The ESN model prediction is far from the actual price data, meaning it has a high MSE. Further, it does a poor job in capturing short-term price trends. LSTM's price prediction, on the other hand, is quite close to the actual price data and does a fairly good job in capturing short-term price trends, although with a small delay. Overall, the ESN model wouldn't be of much use here, but the LSTM model could provide some useful insights on price trend and value (with a margin for error specified).

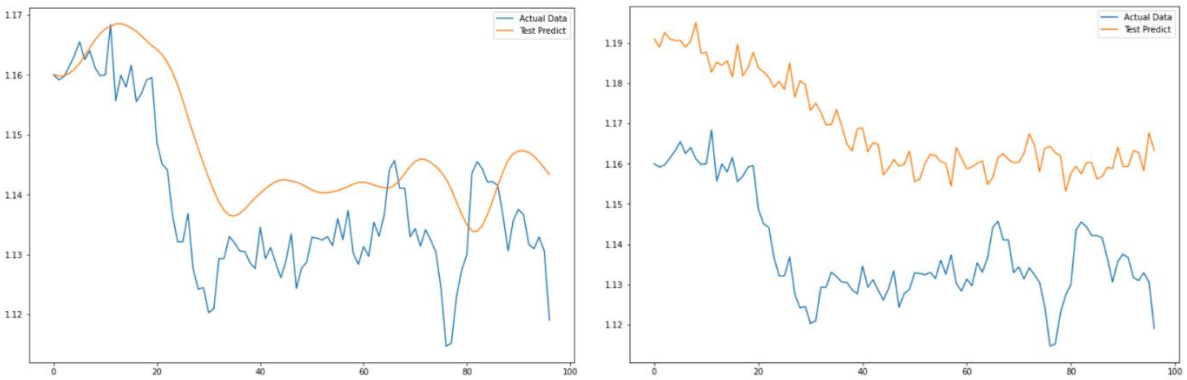

Figure 16: EUR/USD Plots (LSTM on Left ESN on Right)

4.5.2 Hang Seng Index (Volatility = 21.9%)

Figure 17 shows the charts generated for HSI. The ESN model prediction is a lot better than in the EUR/USD case. However, the prediction chart is a lot more volatile than the actual price. This could be attributed to the chaos introduced in the Reservoir of the ESN. In the second half of the chart, the prediction becomes even more chaotic. LSTM's price prediction on the other hand, again captures the price trend quite well. However, the delay is still there. Both models could be used to provide insights on the price value, with LSTM having a slight advantage due to lower MSE. The LSTM model would be more suitable to assess the price trend.

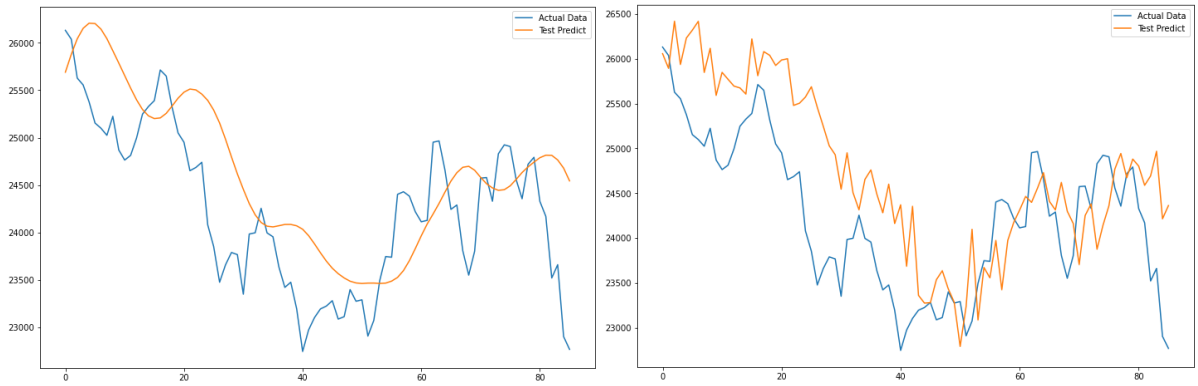

Figure 17: HSI Plots (LSTM on Left ESN on Right)

4.5.3 Bitcoin (Volatility = 64.7%)

Figure 18 shows the charts generated for BTC-USD. Both the models offer a comparable performance, in terms of price value and trend. This is also supported by the almost similar MSE score obtained in Table 3. It is useful to note that the ESN model in the beginning is far from the actual value, but the performance improves as time goes by. Either model could be employed to provide useful insights.

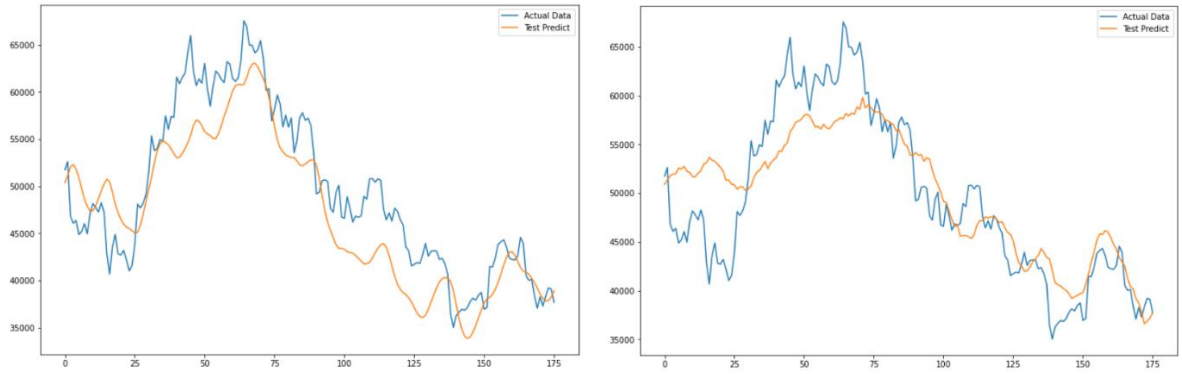

Figure 18: BTC-USD Plots (LSTM on Left ESN on Right)

4.5.4 Ethereum (Volatility = 85.2%)

Figure 19 shows the charts generated for ETH-USD. The LSTM model's predictions can be seen to oscillate a lot more than the actual price value. The LSTM model interprets a lot more price volatility than is present. However, it still manages to capture the price trend. The ESN model on the other hand, fits the actual price data a lot more closely, and does a good job in capturing the price trend. The ESN model in this case would be more useful than LSTM for both price prediction and trend interpretation.

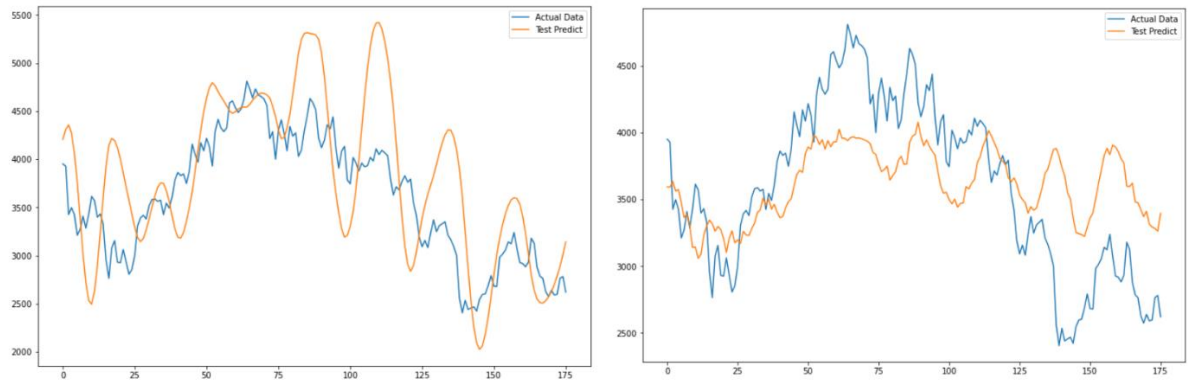

Figure 19: ETH-USD Plots (LSTM on Left ESN on Right)

4.5.5 GameStop (Volatility = 178.8%)

Figure 20 shows the plots generated for GME. The LSTM model is hardly able to capture the price trend for this highly volatile equity. Nor can it offer a close enough price prediction at any point except when the price reverts to its starting value. ESN on the other hand does a much better job of capturing the price trend, sometimes with a slight delay. The price prediction is also a lot closer than LSTM, although it isn't that close to the actual value. However, for such volatile price data which is attributed to special factors like Reddit investors trying to take down the institutional hedge funds through a short squeeze, an accurate price prediction is nearly impossible, and it is impressive to see ESN being able to follow the price trend to a considerable degree.

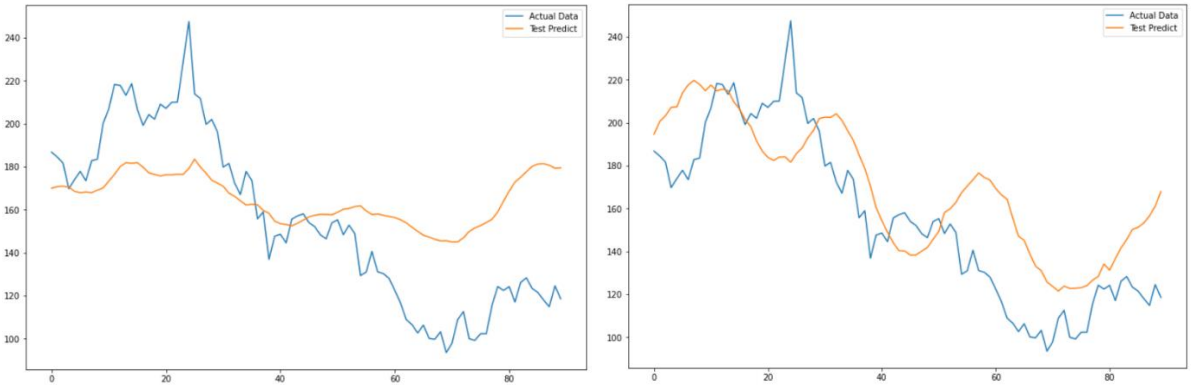

Figure 20: GME Plots (LSTM on Left ESN on Right)

4.6 Model Selected based on Quantitative and Qualitative Evaluation

Table 5 summarizes the suggested model for each case based on the evaluation conducted above. We can clearly see that for low volatility cases LSTM is preferable while for high volatility cases ESN becomes the preferable choice.

Instrument	Volatility	Quantitative	Qualitative	Overall
EUR/USD	6.6%	LSTM	LSTM	LSTM
HSI	21.9%	LSTM	LSTM	LSTM
BTC-USD	64.7%	LSTM/ ESN	LSTM/ ESN	LSTM/ ESN
ETH-USD	85.2%	ESN	ESN	ESN
GME	178.8%	ESN	ESN	ESN

Table 5: Experiment Summary

5 Conclusion

In this project we introduce Reservoir Computing based Echo State Networks as an RNN architecture for financial forecasting applications. After explaining the architecture and the code developed, we compare the results of financial forecasting by ESN and LSTM on various financial instruments such as EUR/USD, Hang Seng Index, Bitcoin, Ethereum, and Gamestop. By evaluating the qualitative and quantitative model performance on these instruments we conclude that LSTM is the preferable choice when forecasting low volatility instruments while ESN could be considered as an alternative option for high volatility cases. Further, ESN offers much faster training times because of the fixed reservoir weights, the use of PCA, and a simple ridge regression predictor. Being able to quickly try out different models and tune the hyperparameters might be beneficial to data scientists, traders, and quants. Further, if the model needs to be dynamically updated using current market data, training time might be of utmost importance, especially for high frequency algo trading applications. As a side note, due to time constraints, our models were created by using only 2 years of daily closing price data. Further research is recommended by testing the results with longer time series, more parameters (open price, volume, high, low), finer granularity, and a wider selection of financial instruments.

6 References:

[1] Filippo Maria Bianchi, Simone Scardapane, Sigurd Løkse, and Robert Jenssen: “Reservoir computing approaches for representation and classification of multivariate time series. 7th June 2020.

[2] Herbert Jaeger and Harald Haas: “Harnessing Nonlinearity: Predicting Chaotic Systems and Saving Energy in Wireless Communication. 2nd April 2004.

[3] F. M. Bianchi, E. Maiorino, M. C. Kampffmeyer, A. Rizzi, and R. Jenssen, Recurrent Neural Networks for Short-Term Load Forecasting: An Overview and Comparative Analysis. Springer, 2017

[4] A. Rodan, A. Sheta, and H. Faris, “Bidirectional reservoir networks trained using SVM+ privileged information for manufacturing process modeling,” Soft Computing, vol. 21, no. 22, pp. 6811–6824, 2017.

[5] F. M. Bianchi, S. Scardapane, S. Løkse, and R. Jenssen, “Bidirectional deep-readout echo state networks,” in European Symposium on Artificial Neural Networks, 2018.

[6] S. Løkse, F. M. Bianchi, and R. Jenssen, “Training echo state networks with regularization through dimensionality reduction,” Cognitive Computation, vol. 9, no. 3, pp. 364–378, Jun 2017.

[7] ciortanamadilina, “EchoStateNetwork”, https://github.com/ciortanmadalina/EchoStateNetwork, March 2019.