Time Series Forecasting using Recurrent Neural Networks

JW du Toit
Computer Science Department
Stellenbosch University
Stellenbosch, South Africa
22808892@sun.ac.za
22808892

***Abstract*—Time series forecasting is an important task which has been applied to several different fields such as weather forecasting, stock price prediction and economic trend forecasting. Time series forecasting allows us to make future predictions based on historical time stamped data in order to inform strategic decision making. Recurrent Neural Networks (RNNs) are machine learning algorithms that are well-suited to the task of time series forecasting due to their ability to capture and use sequence information. This project performs a comparative study between three different types of simple RNNs, namely the Elman RNN (ERNN), Jordan RNN (JRNN) and multi-RNN (MRNN), for the task of time series forecasting. These algorithms are compared over five different datasets to determine which simple RNN performs best. We derive that the JRNN architecture performs the worst in general out of the three simple RNNs. Furthermore, we conclude that the MRNN algorithm performs the best in general on the datasets used in this project and it obtains competitive results on each of the datasets.**

I. Introduction

Time series forecasting has become an important task which is applied to several different fields such as stock price prediction and weather forecasting. Time series forecasting enables us to analyse historical time series data and make predictions to inform strategic decision making. Recurrent Neural Networks (RNN) are popular machine learning algorithms with the ability to capture sequence information and thus they are well-suited for the task of time series forecasting. This project performs a comparative study between three different types of simple RNNs, namely the Elman RNN (ERNN), Jordan RNN (JRNN) and multi-RNN (MRNN), for the task of time series forecasting.

The objective of this project is to determine which of the three simple RNN algorithms performs best with regard to time series forecasting. We compare their performance over five different time series datasets to establish which algorithm performs best. Furthermore, this project aims to determine the optimal configurations for each algorithm for each dataset.

The three algorithms are implemented, trained and their hyperparameters are tuned using a cross-validation strategy for each of the five datasets. The accuracies of these algorithms are then compared on each of the five datasets to determine which algorithm performs best with regard to the task of time series forecasting.

From the results we derive that the JRNN architecture performs the worst in general out of the three simple RNNs. Furthermore, we conclude that the MRNN algorithm performs the best in general on the datasets used in this project and it obtains competitive results on each of the datasets.

The rest of the report proceeds as follows: Section II provides a high-level discussion on time series forecasting, the artificial neuron and neural networks. Section III describes the three different simple RNN architectures and how they are evaluated. Section IV describes the five datasets and the process that was followed to obtain the empirical results. Section V presents and discusses the results obtained from the experiments and Section VI draws a conclusion for the report.

II. Background

This section provides a high-level discussion on time series forecasting, the artificial neuron and feedforward neural networks.

A. Time series forecasting

Time series data is a sequence of data points that are indexed by time and usually consist of successive measurements made from the same source over a certain time period. Some examples of time series data include stock prices, rainfall measurements and heartbeats per minute. Time series forecasting is the process of using time series data to make predictions to inform strategic decision making. Thus, time series forecasting attempts to build models through historical analysis to predict future outcomes. Examples of time series forecasting include forecasting of weather patterns, stock prices and economic trends.

B. The artificial neuron

The artificial neuron (AN) [2] is a machine learning method based on the working of neurons in the human brain. The artificial neuron implements a functional mapping from a number of input signals to an output signal. The AN receives a vector of I input signals,

$$\mathbf{z} = (z_1, z_2, \ldots, z_I) \tag{1}$$

where each input signal, z_i, is associated with a weight, v_i to determine the effect of that input signal. The net input signal

to the AN is most commonly calculated as a weighted sum of all input signals and is given by

$$net = \sum_{i=1}^{I} z_i v_i \tag{2}$$

For the remainder of this report we assume the use of this weighted sum for the calculation of the net input signal. The AN uses the net input signal along with a threshold value, θ, as input to an activation function, f_{AN}, to calculate the output signal, o. The input to the activation function is given by $net - \theta$. To simplify this we augment the input vector to include an additional input unit, z_{I+1}, which is referred to as the bias unit which always has a value of -1, and is associated with the weight v_{I+1} which is the value of the threshold. The net input signal to the AN is then calculated as

$$\begin{aligned} net &= \sum_{i=1}^{I} z_i v_i - \theta \\ &= \sum_{i=1}^{I} z_i v_i + z_{i+1} v_{i+1} \\ &= \sum_{i=1}^{I+1} z_{i+1} v_{i+1} \end{aligned} \tag{3}$$

where $\theta = z_{i+1} v_{i+1} = -v_{i+1}$. The output of the AN, o, is then given by

$$o = f_{AN}(net) \tag{4}$$

There are many different types of activation functions, f_{AN}, that can be used and which one to choose depends on the task the model is trying to solve.

The focus of this project is supervised-learning which is a machine learning method where the algorithm is provided with a dataset that consists of input vectors along with a target output, t, for each input vector. The aim of the algorithm is to learn the values of the weights v_i $i \in 1, 2, \ldots I$, and the threshold θ from the given data such that the difference between the target output, t, and the output of the AN, o, is minimised. There are many different optimisation techniques used to adjust the weights of the AN to minimise this difference. These optimisation techniques use an error function, $\mathcal{E}$, to measure the difference between the target and output values. These optimisation techniques aim to learn the weight values that minimise the error function $\mathcal{E}$.

C. Feedforward neural networks

Single ANs as described in Section II-B have limitations on the type of functional mapping they can learn and are only suitable for linearly separable functions. Thus, we use a layered network of ANs to learn more complex functions that are not linearly separable. These layered networks are referred to as neural networks (NN). There are many different multilayer NN architectures and we will describe the architecture for the standard feedforward neural network (FFNN) [4] which forms the basis for most NN architectures. Fig. 1 provides the standard architecture of a FFNN with a single hidden layer. However, FFNNs can have more than one hidden layer. The output of the FFNN for the p^{th} input instance, $\mathbf{z}_p$, is calculated with a forward pass through the network such that for each output unit, o_k, we have

$$\begin{aligned} o_{k,p} &= f_{o_k}(net_{o_{k,p}}) \\ &= f_{o_k}\left(\sum_{j=1}^{J+1} w_{kj} f_{y_j}(net_{y_{j,p}})\right) \\ &= f_{o_k}\left(\sum_{j=1}^{J+1} w_{kj} f_{y_j}(\sum_{i=1}^{I+1} v_{ji} z_{i,p})\right) \end{aligned} \tag{5}$$

where f_{o_k} and f_{y_j} are the activation functions for the output unit o_k and hidden unit y_j respectively, w_{kj} is the weight between output unit o_k and hidden unit y_j, v_{ji} is the weight between hidden unit y_j and input unit z_i, $z_{i,p}$ is the value of the input unit z_i of input instance $\mathbf{z}_p$, the input unit z_{I+1} and the hidden input y_{J+1} are the bias units representing the threshold values of the neurons in the next layer.

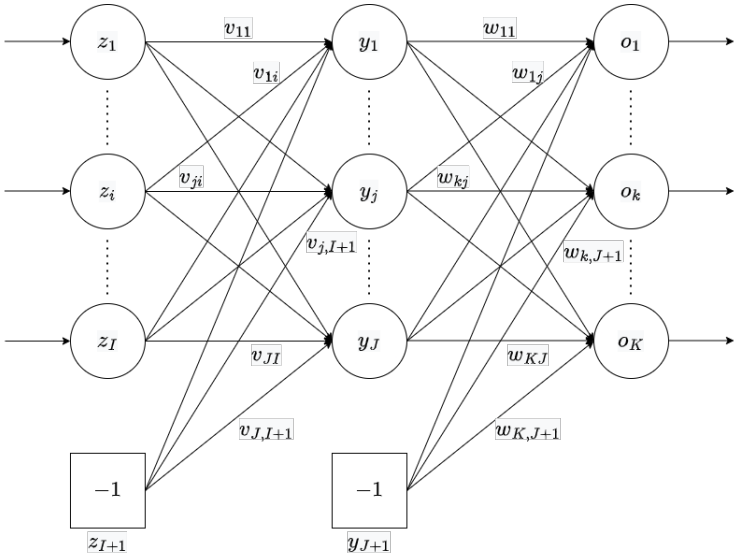

Fig. 1. Standard FFNN architecture

To determine how the weights are trained for a FFNN we describe one of the most popular learning algorithms called backpropagation [5] which uses a number of learning iterations known as $epochs$ to train the weights of a FFNN. The algorithm uses two phases in each $epoch$, namely the feedforward pass, which calculates the output values of the FFNN for each training instance as described above and a mechanism called backward propagation. The backward propagation phase uses the outputs of the feedforward phase to compute an error signal which is propagated to the preceding layer to adjust the associated weights. This process is repeated for each preceding layer until the input layer is reached. The pseudo code for the backpropagation algorithm is given by Algorithm 1 where $\mathcal{E}_p$ is the error between the output and

target values of an instance and the error signals, δ_{o_k} and δ_{y_j}, are given by

$$\begin{aligned}\delta_{o_k} &= \frac{\partial \mathcal{E}}{\partial net_{o_k}} \\ \delta_{y_j} &= \frac{\partial \mathcal{E}}{\partial net_{y_j}}\end{aligned} \tag{6}$$

We can then use different optimisation techniques to adjust the weights w_{kj} and v_{ji} using the associated error signals.

Algorithm 1 Backpropagation algorithm

```
Initialise weights, η, α and the number of epochs t = 0
while stopping condition(s) not true do
  Let E_T = 0
  for each training instance p do
    Do the feedforward phase to calculate y_{j,p}(∀j = 1, ..., J) and o_{k,p}(∀k = 1, ..., K)
    Compute the output error signals δ_{o_{k,p}} and the hidden layer error signals δ_{y_{j,p}}
    Adjust weights w_kj and v_ji through backpropagation of errors
    E_T += E_p
  end for
  t = t + 1
end while
```

III. Methodology

This section describes the three algorithms developed to perform the task of time series forecasting, namely ERNN, JRNN and MRNN. All three of these algorithms are extensions of the FFNN and use different forms of memory states and feedback connections which allows the learning of temporal characteristics of the dataset. Furthermore, we discuss the cross-validation procedure used to evaluate the three algorithms.

A. Elman recurrent neural networks

The ERNN algorithm [1] keeps a memory of previous hidden layer states which is referred to as the context layer. The context layer stores the previous state of the hidden layer, *i.e.* the activations of the hidden layer units at the previous instance presentation. The context layer serves as an extension to the input layer and feeds signals from the previous network states to the hidden layer. Thus, the input layer is given by

$$\mathbf{z} = (z_1, ..., z_{I+1}, z_{I+2}, \ldots, z_{I+1+J}) \tag{7}$$

where $z_1, ..., z_{I+1}$ and $z_{I+2}, \ldots, z_{I+1+J}$ are the input and context units respectively which are both fully connected to all hidden units. The output unit, o_k, for the p^{th} instance is calculated as

$$o_{k,p} = f_{o_k}\left(\sum_{j=1}^{J+1} w_{kj} f_{y_j}\Big(\sum_{i=1}^{I+1+J} v_{ji} z_{i,p}\Big)\right) \tag{8}$$

where $(z_{I+2,p}, ..., z_{I+1+J,p}) = (y_{1,p}(t-1), ..., y_{J,p}(t-1))$. This architecture can be extended through saving multiple previous hidden layer states to form the context layer. The algorithm will save the most recent hidden layer states and each of these saved hidden layer states will be fully connected to the hidden layer units.

B. Jordan recurrent neural networks

The JRNN algorithm [6] keeps a memory of previous output layer states which is referred to as the state layer.The state layer stores the previous state of the output layer, *i.e.* the activations of the output layer units at the previous instance presentation. The state layer serves as an extension to the input layer and feeds signals from the previous output layer to the current hidden layer. Thus, the input layer is given by

$$\mathbf{z} = (z_1, ..., z_{I+1}, z_{I+2}, \ldots, z_{I+1+K}) \tag{9}$$

where $z_1, ..., z_{I+1}$ and $z_{I+2}, \ldots, z_{I+1+K}$ are the input and state units respectively which are both fully connected to all hidden units. The output unit, o_k, for the p^{th} instance is calculated as

$$o_{k,p} = f_{o_k}\left(\sum_{j=1}^{J+1} w_{kj} f_{y_j}\Big(\sum_{i=1}^{I+1+K} v_{ji} z_{i,p}\Big)\right) \tag{10}$$

where $(z_{I+2,p}, ..., z_{I+1+K,p}) = (o_{1,p}(t-1), ..., o_{K,p}(t-1))$. This architecture can be extended through saving multiple previous output layer states to form the state layer. The algorithm will save the most recent output layer states and each of these saved output layer states will be fully connected to the hidden layer units.

C. Multi-recurrent neural networks

The MRNN algorithm is a combination of the architectures used in the ERNN and JRNN algorithms. MRNNs maintain a memory of the hidden and output layer states which is referred to as the context and state layers respectively. These layers serve as an extension to the input layer and feed signals from the previous hidden and output layers to the current hidden layer. Thus, the input layer is given by

$$\mathbf{z} = (z_1, ..., z_{I+1}, z_{I+2}, \ldots, z_{I+1+J}, z_{I+1+J+1}, ...z_{I+1+J+K}) \tag{11}$$

where $z_1, ..., z_{I+1}$, $z_{I+2}, \ldots, z_{I+1+J}$ and $z_{I+1+J+1}, ...z_{I+1+J+K}$ are the input, context and state units respectively which are all fully connected to all hidden units. The output unit, o_k, for the p^{th} instance is calculated as

$$o_{k,p} = f_{o_k}\left(\sum_{j=1}^{J+1} w_{kj} f_{y_j}\Big(\sum_{i=1}^{I+1+J+K} v_{ji} z_{i,p}\Big)\right) \tag{12}$$

where $(z_{I+2,p}, ..., z_{I+1+J,p}) = (y_{1,p}(t-1), ..., y_{J,p}(t-1))$ and $(z_{I+1+J+1,p}, ..., z_{I+1+J+K,p}) = (o_{1,p}(t-1), ..., y_{K,p}(t-1))$. This architecture can be extended through saving multiple previous hidden and output layer states to form the context and state layers. The algorithm will save the most recent hidden and output layer states and each of these saved states will have input connections to the current hidden layer units.

D. Evaluation

First, we split the given dataset into training and testing sets by taking the training set as the first n data points in chronological order and the testing set as the remaining data points. The training set will be used to train and tune the hyperparameters of the model to obtain the optimal configuration for that model while the testing set will be used to evaluate the generalisation performance of the model. It is important to tune the hyperparameters of the models to ensure that the models are not overfitting or underfitting. We perform a cross-validation process on the time series data to evaluate the performance of the three different algorithms and find the optimal configurations for these algorithms. The optimal configuration provides a good balance between overfitting and underfitting the model as it is tuned through the cross-validation process to determine the configuration with the best generalisation performance. Thus, we select a combination of hyperparameters that provides a good fit of the training set but is simple enough to not overfit the training data. We divide the training set into a number of sections referred to as folds. Since we are working with time series data is it imperative that these folds are ordered chronologically with regard to time. To evaluate the performance of the model we first train the algorithm on the first fold and test its performance on the second fold. Then, we concatenate the first two folds and train the model using this data and test its performance on the third fold. This process is repeated where at each iteration the previous testing fold is added to the training data and used to predict the following fold. For each of these iterations we obtain the error on the test data and take the average of these errors as the indication of the performance of the model.

IV. Empirical procedure

This section describes the five different datasets, the details of the model architectures used and procedure that was followed to obtain the empirical results for this project. These experiments were conducted to see how the three algorithms perform compared to each other as well as to find optimal configurations for each of them for the five different datasets.

A. Datasets

We used five different datasets to compare the performance of the three simple RNN architectures. These datasets were used to provide a general consensus as to which of these algorithms perform best, as some algorithms may be more suited to certain types of data than others. Each of the datasets contained a similar simple format with no need for processing as they contained no missing values or any data quality issues. The five datasets used:

1) New Delhi climate data. This dataset provides the mean temperature in the city of New Delhi for each day from 2013 to 2016. Fig. 2 provides the mean temperatures from this dataset. From Fig. 2 we observe that this dataset is approximately stationary.
2) IBM stock price data. This dataset provides the closing stock price of IBM for each day from 2006 to 2017. Fig. 3 provides the closing stock prices from this dataset. From Fig. 3 we observe that this dataset is non-stationary.
3) Google stock price data. This dataset provides the closing stock price of Google for each day from 2006 to 2017. Fig. 4 provides the closing stock prices from this dataset. From Fig. 4 we observe that this dataset is non-stationary.
4) Electricity consumption data. This dataset provides the electricity consumption of a factory for each month from 1985 to 2017. Fig. 5 provides the electricity consumption values from this dataset. From Fig. 5 we observe that this dataset is non-stationary.
5) Corn price data. This dataset provides the corn prices for each week from 2013 to 2017. Fig. 6 provides the corn prices from this dataset. From Fig. 6 we observe that this dataset is non-stationary.

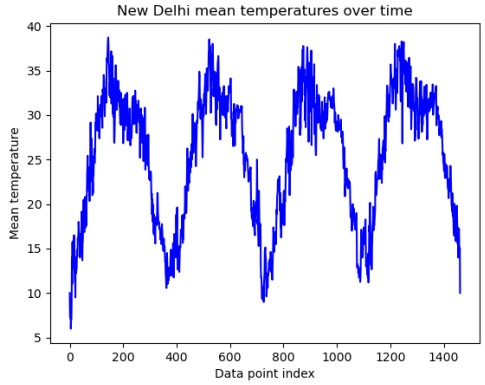

Fig. 2. New Delhi mean temperatures over time

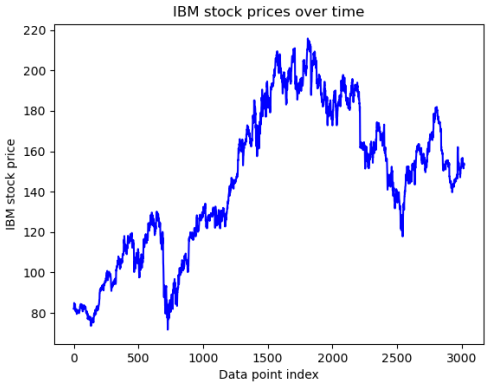

Fig. 3. IBM stock prices over time

Fig. 4. Google stock prices over time

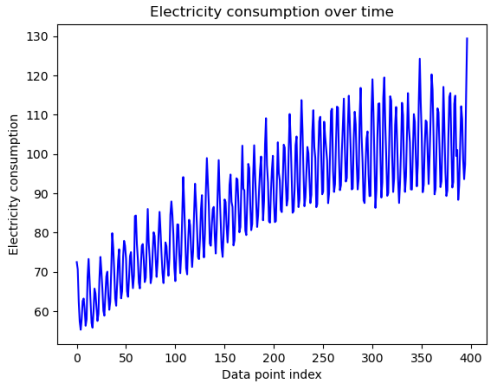

Fig. 5. Electricity consumption over time

Fig. 6. Corn prices over time

B. Model architectures

We used the sigmoid activation function for each unit in the hidden layer for each of the three algorithms. The sigmoid activation is given by

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{13}$$

where x is the net input signal for a hidden unit. Furthermore, we used the mean squared error (MSE) loss function for each of the three RNN algorithms as a measure of performance and for the training of the models. The MSE error for the p^{th} instance is given by

$$\mathcal{E}_p = \frac{1}{2}\left(\frac{\sum_{k=1}^{K}\left(t_{k,p} - o_{k,p}\right)^2}{K}\right) \tag{14}$$

where K is the number of output units, $t_{k,p}$ and $o_{k,p}$ are the target and output values of the k^{th} output unit for instance p. We used a stochastic gradient descent (SGD) [3] optimisation algorithm to learn the weights and biases of the three models during training. The SGD algorithm updates the weights w_{kj} and v_{ji} according to

$$\begin{aligned} w_{kj}(t) += \Delta w_{kj}(t) + \alpha \Delta w_{kj}(t-1) \\ v_{ji}(t) += \Delta v_{ji}(t) + \alpha \Delta v_{ji}(t-1) \end{aligned} \tag{15}$$

where α is a parameter known as the momentum. The Δw_{kj} and Δv_{ji} elements are calculated using the error signals δ_{o_k} and δ_{y_j} respectively and are given by

$$\begin{aligned} \Delta w_{kj} &= -\eta \delta_{o_k} y_j \\ \Delta v_{ji} &= -\eta \delta_{y_j} z_i \end{aligned} \tag{16}$$

where η is the learning rate.

C. Evaluation and hyperparameter tuning

For each of the five datasets the dataset was split up into a train and test set with 80% for training and 20% for testing.

For each dataset we performed the cross-validation procedure as described in Section III to obtain the optimal configurations for each of the algorithms. We split the training set into 10 folds to perform the cross-validation procedure to the different configurations of each algorithm. We experiment with different hyperparameters for each algorithm to obtain the optimal configuration for each algorithm on the associated dataset. Finally, we use the optimal configuration of hyperparameters to measure the performance of the algorithm on the testing set.

The type of hyperparameters for each of the three algorithms are the same and they include the number of hidden units, the number of *epochs*, the learning rate and the number of previous states in the feedback connection. The number of previous states in the feedback connection is the number of previous hidden layer states for the ERNN algorithm, the number of previous output layer states for the JRNN algorithm and the number of previous hidden layer and output layer states for the MRNN algorithm. To determine the optimal configurations for one of the algorithms on a specific dataset we test the following hyperparameter values:

- Hidden units: Increments of 10 from 10 to 100.
- *epochs*: Increments of 3 from 2 to 20.
- Learning rate: Increments of 0.002 from 0.001 to 0.013.
- Number of previous states: Increments of 3 from 2 to 20.

For each possible value of hidden units, we select each value of all of the other hyperparameters to form all of the

possible combinations of these four hyperparameters. Each of these combinations are then tested through the cross-validation mechanism and their performance is recorded. Furthermore, we use a momentum value, α, of 0.9 for each of the algorithms. The optimal combination is then chosen and used to determine the performance of the algorithm on the testing set of the dataset. We use 20 independent runs for each model and obtain the average and standard deviation of the MSE scores from the independent runs to determine the performance of the model on average. This procedure is followed for each algorithm on each of the five datasets.

V. Results

This section presents and discusses the results obtained from the experiments in Section IV for each of the five datasets. The results were obtained with the optimal configuration of each model for an associated dataset obtained through the cross-validation hyperparameter tuning process described in Section IV.

A. New Delhi climate results

Table I provides the mean and standard deviation of the MSE scores obtained from the 20 independent runs for the three models with optimal configurations on the New Delhi climate data. From Table I we observe that the MRNN outperformed the other two simple RNNs with a mean MSE of 3.05 over the 20 independent runs. Furthermore, we observe that the ERNN performed only slightly worse than the MRNN and that the JRNN achieved the worst performance. The performance of the MRNN can be attributed to the use of both the context and state layers such that it takes into account previous states from the hidden and output layers which provide more information in the hidden layer to assist with the prediction of future time series values. Finally, we observe that the MRNN algorithm also obtained the lowest standard deviation across the 20 independent runs which is an indication that it achieved more stable results on the given dataset.

TABLE I
New Delhi mean climate results

Algorithm	Mean MSE	Standard deviation
ERNN	3.25	0.46
JRNN	5.09	0.41
MRNN	3.05	0.38

B. IBM stock price results

Table II provides the mean and standard deviation of the MSE scores obtained from the 20 independent runs for the three models with optimal configurations on the IBM stock price data. From Table II we observe that the ERNN algorithm outperformed the other two simple RNNs with a mean MSE of 11.32 over the 20 independent runs. Furthermore, we observe that the JRNN algorithm performed much worse than the other two algorithms with a mean MSE of 229.97. These results can be attributed to the context layer of the ERNN algorithm providing helpful information to the simple RNN to predict the next time step value. The state layer used in the JRNN does not provide as useful information to be used in the hidden layer of the model for this dataset and thus the JRNN performed worse than the other two algorithms that incorporate memory from the previous hidden states. This can be attributed to the state layer of the JRNN only saving one unit for each saved output layer state where as the context layer in the ERNN saves all of the hidden unit states for each of hidden layer states and thus provides more information to the current hidden layer. Finally, we observe that the ERNN algorithm obtained the smallest standard deviation of 3.17 over the 20 independent runs and thus had the most stable execution on the IBM stock price dataset.

TABLE II
IBM stock price results

Algorithm	Mean MSE	Standard deviation
ERNN	11.32	3.17
JRNN	229.97	31.19
MRNN	45.26	15.41

C. Google stock price results

Table III provides the mean and standard deviation of the MSE scores obtained from the 20 independent runs for the three models with optimal configurations on the Google stock price data. From Table III we observe that the ERNN algorithm outperformed the other two simple RNNs with a mean MSE of 61.96 over the 20 independent runs. The JRNN and MRNN algorithms obtained higher mean MSE scores of 1436.65 and 128.32 respectively. These results can be attributed to the context layer of the ERNN algorithm providing helpful information to the hidden layer of the simple RNN to improve prediction accuracy. The state layer used in the JRNN and MRNN does not provide as useful information to be used in the hidden layer of the model for this dataset. Finally, we observe that the ERNN algorithm obtained the smallest standard deviation of 19.09 over the 20 independent runs and thus had the most stable execution over the Google stock price dataset.

TABLE III
Google stock price results

Algorithm	Mean MSE	Standard deviation
ERNN	61.96	19.09
JRNN	1436.65	61.32
MRNN	128.32	28.21

D. Electricity consumption results

Table IV provides the mean and standard deviation of the MSE scores obtained from the 20 independent runs for the three models with optimal configurations on the electricity consumption data. From Table IV we observe that the MRNN algorithm outperformed the other two simple RNNs with a

mean MSE of 393.03 over the 20 independent runs. The JRNN algorithm achieved the highest mean MSE of 417.21. These results can again be attributed to the context layer which maintains a memory of previous hidden layer unit activation states which provides better information to the hidden layer than the state layer used in the JRNN algorithm. Furthermore, we observe that the JRNN algorithm obtained the lowest standard deviation over the 20 independent runs and thus had the most stable execution on the electricity consumption dataset.

TABLE IV
ELECTRICITY CONSUMPTION RESULTS

Algorithm	Mean MSE	Standard deviation
ERNN	408.47	73.01
JRNN	417.21	54.32
MRNN	393.03	68.22

E. Corn price results

Table V provides the mean and standard deviation of the MSE scores obtained from the 20 independent runs for the three models with optimal configurations on the corn price data. From Table IV we observe that the MRNN algorithm outperformed the other two simple RNNs with a mean MSE of 1.28 over the 20 independent runs. The ERNN algorithm obtained the highest mean MSE of 1.85. The performance of the MRNN algorithm can be attributed to the use of both the context and state layers such that it takes into account previous states of the hidden and output layers to assist the simple RNN with the prediction of time series values. Furthermore, we observe that the MRNN algorithm also obtained the lowest standard deviation of 0.31 over the 20 independent runs and thus had the most stable execution on the corn price dataset.

TABLE V
CORN PRICE RESULTS

Algorithm	Mean MSE	Standard deviation
ERNN	1.85	0.36
JRNN	1.81	0.53
MRNN	1.28	0.31

VI. CONCLUSION

The objective of this project is to compare the performance of three simple RNN architectures, namely the ERNN, JRNN and MRNN, for the task of time series forecasting. We use five different datasets to perform the comparative analysis of the three algorithms.

For each dataset we train each of the three algorithms and tune their hyperparameters using a cross-validation method to find the optimal configuration of the algorithm for the associated dataset. The optimal configuration of each algorithm is then used to test the performance on the dataset so that we can compare the three algorithms.

From the results discussed in Section V we derive that the JRNN algorithm in general performs worse than the other two simple RNN architectures over the five datasets used in this project. We observe that the performance of the ERNN and MRNN architectures depend on the associated dataset, with the MRNN obtaining the best results on three of the datasets and ERRN performing the best on two of the datasets. However, the MRNN outperformed the JRNN on each of the datasets and achieved competitive results for each of the five datasets. Thus, we conclude that the MRNN algorithm is the best simple RNN architecture for time series forecasting. The performance of the MRNN algorithm can be attributed to the ability of keeping memory of the previous hidden and output layer states and using this additional information in the hidden layer to improve predictive performance.

This project can be extended by using more datasets to compare the three algorithms. Furthermore, we can implement other RNN architectures and compare their performance to the three simple RNNs used in this project.

REFERENCES

[1] J. Elman, "Distributed representations, simple recurrent networks, and grammatical structure", Machine Learning, vol. 7, no. 2-3, pp. 195-225, 1991.

[2] W. McCulloch and W. Pitts, "The Statistical Organization of Nervous Activity", Biometrics, vol. 4, no. 2, p. 91, 1948.

[3] H. Robbins and S. Monro, "A Stochastic Approximation Method", The Annals of Mathematical Statistics, vol. 22, no. 3, pp. 400-407, 1951.

[4] F. Rosenblatt, "The perceptron: A probabilistic model for information storage and organization in the brain", Psychological Review, vol. 65, no. 6, pp. 386-408, 1958.

[5] D. Rumelhart, G. Hinton and R. Williams, "Learning representations by back-propagating errors", Nature, vol. 323, no. 6088, pp. 533-536, 1986.

[6] L. Saul and M. Jordan, "Attractor Dynamics in Feedforward Neural Networks", Neural Computation, vol. 12, no. 6, pp. 1313-1335, 2000.