Time Series Forecasting using Recurrent Neural Networks

Schalk Visagie
Computer Science Department
Stellenbosch University
Stellenbosch, South Africa
25349589@sun.ac.za

***Abstract*—**
***Index Terms*—**

I. Introduction

Time series forecasting is a critical task in many domains, involving the prediction of future values based on historical data. Recurrent Neural Networks (RNNs) are a class of neural networks specifically designed for sequential data, making them highly suitable for such forecasting tasks. The defining feature of RNNs is the presence of feedback loops, which allow the networks to maintain an internal state or memory of past information to influence future predictions.

Among the foundational RNN architectures are the Elman Recurrent Neural Network (Elman-RNN) and the Jordan Recurrent Neural Network (Jordan-RNN). The Elman-RNN captures temporal dynamics by feeding the previous hidden state back into the current hidden layer computation. In contrast, the Jordan-RNN utilizes feedback from the network's previous output to inform the current hidden state. The Multi-Recurrent Neural Network (MRNN) is a hybrid architecture that integrates both of these feedback mechanisms, theoretically enabling the capture of more complex temporal dependencies.

While these three architectures are well-established, a direct and rigorous comparison of their performance across diverse time series problems is necessary to guide model selection. This paper aims to fill that gap by evaluating the predictive capabilities of the Elman-RNN, Jordan-RNN, and MRNN. The primary objective is to determine which architecture provides the most accurate forecasts across datasets with varying characteristics, such as stationarity, seasonality, and multivariate dependencies.

The comparative analysis is conducted using five distinct datasets: a synthetic autoregressive series, the S&P 500 and VIX financial indices, an electricity load profile, and a multivariate weather dataset. To ensure a fair and robust evaluation, a comprehensive hyperparameter tuning process is implemented for each model on each dataset using a growing-window cross-validation strategy, which respects the temporal order of the data. The final performance of the optimized models is then assessed on an unseen holdout test set, using Root Mean Squared Error (RMSE) and Mean Absolute Error (MAE) as the primary evaluation metrics.

The remainder of this paper details this investigation. The background section provides a technical overview of the three RNN architectures and their training process. The methodology section describes the datasets, the cross-validation procedure, and the measures taken to prevent overfitting. Following this, the empirical procedure outlines the experimental setup and hyperparameter tuning framework. The results section presents the performance of each model on the holdout test sets, leading to a final conclusion on the relative strengths of the Elman-RNN, Jordan-RNN, and MRNN for time series forecasting.

II. Background

Elman Recurrent Neural Network

The Elman recurrent neural network (Elman-RNN) is a simple RNN architecture consisting of an input layer, a hidden layer, a context layer, and an output layer. The context layer, with the same number of neurons as the hidden layer, stores a copy of the previous hidden state and feeds it back to the hidden layer, enabling the network to maintain short-term memory of past inputs. The Elman-RNN hidden state is computed as follow:

$$h_t = \tanh(W_{xh}X_t + W_{hh}h_{t-1} + b_h)$$

TODO: add explaination of the notation

This feedback loop introduces temporal dynamics, distinguishing it from feedforward networks.

Elman-RNN is suitable for time series prediction because its recurrent structure captures sequential dependencies, allowing it to model patterns in data where current values depend on historical context, such as a simple sequence of XOR operations, stock prices or weather.

Jordan Recurrent Neural Network

The Jordan recurrent neural network (Jordan-RNN) features an input layer, hidden layer, context layer, and output layer. Unlike Elman-RNN, the context layer stores previous output values and feeds them back to the hidden layer, creating a feedback loop from outputs to influence future computations. The Jordan-RNN hidden states are computed as:

$$h_t = \tanh(W_{xh}X_t + W_{yh}y_{t-1} + b_h)$$

The Jordan-RNN's context layer has the same amount of neurons as its output layer in contrast to the Elman-RNN having a context layer with the same size as its hidden layer.

Jordan-RNN applies to time series prediction by leveraging output feedback to handle temporal correlations, making it effective for forecasting tasks like price trends where past predictions inform future ones.

Multi-Recurrent Neural Network

The multi-recurrent neural network (MRNN) integrates features of both Elman-RNN and Jordan-RNN. It includes input, hidden, context, and output layers. The context layer receives feedback from both previous hidden states and outputs and presented to the hidden layer at each time step. This hybrid architecture enhances memory by combining multiple recurrent paths. The MRNN hidden states are computed by the following equation:

$$h_t = \tanh(W_{xh}X_t + W_{hh}h_{t-1} + W_{yh}y_{t-1} + b_h)$$

MRNN is apt for time series prediction as its dual feedback mechanisms provide an alternative way to capture complex temporal dependencies in sequential data compared to the previous two single-loop variants.

While specific equations vary, MRNN extends Elman-RNN and Jordan-RNN formulations by incorporating both hidden and output feedbacks in the hidden state computation, such as combining terms from $h(t-1)$ and $y(t-1)$ in the activation.

Backpropagation Training Process

The training process relies on backpropagation through time (BPTT) to effectively manage the sequential nature of the data. This method works by unfolding the recurrent network across multiple time steps, essentially converting it into a layered feedforward network where the weights are shared among the time-unfolded layers, allowing the use of standard backpropagation techniques to calculate gradients [1].

The process starts with a forward pass through the network. For each time step t in the sequence, the hidden state is calculated as $h_t = \tanh(W_{xh} \cdot X_t + W_{hh} \cdot h_{t-1} + b_h)$, where X_t represents the input at that step, W_{xh} is the weight matrix from input to hidden, W_{hh} is the recurrent weight matrix from previous hidden to current hidden, and b_h is the hidden bias. Following this, the output at each time step is computed as $y_t = \phi(W_{oh} \cdot h_t + b_o)$, with W_{oh} as the weight matrix from hidden to output, b_o as the output bias, tanh as the activation for the hidden layer, and ϕ (such as sigmoid or softmax) for the output layer [1].

Next, the loss is evaluated, often using mean squared error for regression tasks like time series prediction: $L = \frac{1}{N}\sum_{t=1}^{N}(y - y_t)^2$, where y is the target value and N is the number of time steps. The derivative of this loss with respect to the predicted output is $\frac{\partial L}{\partial y_t} = \frac{2}{n}\sum_{t=t}^{n}(y - y_t)$ providing the starting point for error propagation.

The backward pass then proceeds by applying the chain rule to propagate errors back through the unfolded network, accumulating gradients over all time steps. For the output weights, the gradient is $\frac{\partial L}{\partial W_{oh}} = \sum_{i=0}^{t} \frac{\partial L}{\partial y_{t-i}} \cdot \frac{\partial y_{t-i}}{\partial W_{oh}}$, which sums contributions from each relevant time step. For the input-to-hidden weights, it is $\frac{\partial L}{\partial W_{xh}} = \sum_{i=0}^{t}\left[\left(\frac{\partial L}{\partial y_{t-i}} \cdot \frac{\partial y_{t-i}}{\partial h_{t-i}}\right) \cdot \left(\prod_{j=(t-i+1)}^{t} \frac{\partial h_j}{\partial h_{j-1}}\right) \cdot \frac{\partial h_{t-i}}{\partial W_{xh}}\right]$, accounting for how errors flow through the recurrent connections. Similarly, for the hidden-to-hidden weights: $\frac{\partial L}{\partial W_{hh}} = \sum_{i=0}^{t}\left[\left(\frac{\partial L}{\partial y_{t-i}} \cdot \frac{\partial y_{t-i}}{\partial h_{t-i}}\right) \cdot \left(\prod_{j=(t-i+1)}^{t} \frac{\partial h_j}{\partial h_{j-1}}\right) \cdot \frac{\partial h_{t-i}}{\partial W_{hh}}\right]$.

Once these gradients are computed, the weights are updated using gradient descent with a learning rate α. Care must be taken with long sequences to mitigate issues like vanishing or exploding gradients that can arise from repeated multiplications in the product terms [1].

In the context of specific architectures, BPTT in Elman-RNNs emphasizes updating the hidden-to-context feedback to capture internal dynamics; in Jordan-RNNs, it prioritizes the output-to-context loop for incorporating prior predictions; and in MRNN, it handles both types of feedback simultaneously.

Adam Optimization Algorithm The Adam (Adaptive Moment Estimation) optimizer is a widely used algorithm for stochastic gradient-based optimization in machine learning, particularly effective for training simple RNNs on time series data. It adapts the learning rate for each parameter by computing exponentially decaying averages of past gradients and past squared gradients, combining ideas from Root Mean Square Propagation and momentum methods. This adaptive approach helps in handling sparse gradients and non-stationary objectives, leading to faster convergence and better performance in noisy environments [2] [3].

Adam integrates with backpropagation by using the gradients computed through backpropagation to update the model's parameters in an adaptive manner during RNN training. Backpropagation calculates the error gradients by propagating the loss backward through the unrolled RNN over time steps, addressing issues like vanishing gradients in recurrent structures. Adam then applies these gradients to its moment estimates, adjusting learning rates per parameter to optimize weight updates, which enhances convergence speed and stability for time series prediction tasks in shallow RNNs [3].

III. Methodology

A. Dataset descriptions

S&P 500 ETF Daily OHLCV dataset was obtained from Yahoo Finance. This dataset comprises 5,031 trading days of Open, High, Low, Close, and Volume data from 19 September 2005 to 19 September 2025. This dataset is particularly relevant for evaluating RNN architectures due to its inherent non-stationarity, inconsistent variance, and complex non-linear dependencies, which provide a robust benchmark for comparing the ability of different models to learn patterns in time series data and predict next-step log returns. Preprocessing was necessary to ensure data quality and model stability. The 85 instances identified as outliers ($\sigma > 3$) were clamp-transformed to 3 standard deviations from the mean. To address highly correlated features while retaining discriminatory power, the Open, High, and Low features were dropped and replaced with an engineered feature namely the high-low range. The closing price, confirmed as non-stationary by Augmented Dickey-Fuller (ADF) and Kwiatkowski-Phillips-Schmidt-Shin (KPSS) tests, was stabilized by conversion to log returns [4].

$$log\_return_t = log(close_t / close_{t-1})$$

Using log returns is a common way to make financial time series data stationary. Feature scaling for all datasets will be discussed in subsection C along with the time series cross validation.

VIX Daily OHLC dataset, sourced from Yahoo Finance, provides 5,031 observations of daily Open, High, Low, and Close data for the period spanning 19 September 2005 to 19 September 2025. This time series is well-suited for testing RNNs because its structural properties. These structural properties include sudden spikes, and shifting sequential dependencies that create challenging non-stationary patterns. These characteristics differ significantly from the other datasets which make it an ideal candidate for diversifying training data. Data preparation involved several steps. First, 73 outliers exceeding 3 standard deviations were clamped. The non-stationarity of the closing price, verified with ADF and KPSS tests, was resolved by transforming the series' Close feature into log returns. Similar to the S&P 500 data. The redundant and highly correlated Open, High, and Low features were removed and replaced with the engineered high-low range and log return features.

The **ElectricityLoadDiagrams20112014** dataset contains high-frequency electricity consumption readings in kilowatts for 370 clients, recorded every 15 minutes from January 2011 to the end of 2014. For this assignment, a single client's consumption profile, consisting of 140,256 measurements, was arbitrarily selected to create a univariate time series forecasting scenario. This dataset was reduced to a size of 2500 measurements by only using the first few observations. This was done to reduce model training and hyperparameter tuning time to be reasonable. This provides a high-granularity, cyclical test case for evaluating the predictive performance of the RNN models. The data required minimal cleaning as it contained no missing values. However, power consumption values exceeding 3 standard deviations from the mean were clamp-transformed to handle outliers. Both ADF and KPSS tests confirmed that the series was non-stationary. To address this, a 24-hour seasonal differencing was applied, accounting for the inherent daily cyclicality of energy usage.

The **Synthetic Autoregressive Stationary (AR(1))** dataset was generated using Python code with the `NumPy` library, simulating a univariate time series from an AR(1) process defined by the equation $x_t = 0.5x_{t-1} + \epsilon_t$, where ϵ_t is white noise drawn from a normal distribution with mean 0 and standard deviation 1. This dataset consists of 500 sequential observations, following a 500-point burn-in period to ensure the process reaches stationarity. It is particularly suitable for benchmarking RNN architectures in this project due to its inherent stationarity, linear autoregressive dependencies, and absence of trends or seasonality, offering a controlled environment to evaluate the models' ability to capture simple recurrent patterns without the confounding factors present in real-world data, thereby serving as a baseline for comparison with non-stationary datasets. The synthetic nature ensured no missing values or structural anomalies. Stationary was confirmed both by design (autoregressive coefficient $|\phi| = 0.5 < 1$) and through ADF and KPSS tests.

The **TimeSeries Weather Dataset**, sourced from Kaggle, contains hourly historical weather data for two locations. The location having more records (389,496 observations spanning January 1, 1980, to June 6, 2024) was selected. This dataset is a multivariate time series with 17 continuous features, including temperature, humidity, dew point, precipitation, pressure and cloud cover to name a few. This dataset was chosen for its high temporal granularity, pronounced daily and seasonal cycles as well as multivariate interactions. Its inclusion enhances dataset diversity by introducing time series data with multiple cyclical tendencies and non-stationarity, contrasting the previously mentioned datasets. The data exhibited no missing values or major irregularities. Outliers exceeding 3 standard deviations were clamp-transformed. Highly correlated features with absolute correlation greater than 0.75 were dropped to reduce multicollinearity while preserving predictive power. The target feature is defined as the next hour's differenced temperature. Non-stationarity, confirmed by ADF and KPSS tests, was addressed through 24-hour seasonal differencing to account for daily periodicity.

B. Recurrent Neural Networks Implemented

Elman, Jordan and MRNNs were implemented exactly as outlined in the background section. For each dataset the models were trained using the Adam optimization algorithm that uses gradients computed via BPTT. The key hyperparameters for the Adam optimizer, namely the learning rate and weight decay were determined through hyperparameter tuning.

The Mean Squared Error (MSE) was employed as the loss function to guide the training process. MSE is a standard choice for regression tasks, as it measures the average squared difference between the predicted values ($\hat{y}$) and the actual values (y). It is defined by the formula:

$$L_{MSE} = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2$$

This function is differentiable and penalizes larger prediction errors more heavily, making it effective for model optimization.

C. Cross Validation Implementation

To tune hyperparameters and assess model performance, a growing-window cross-validation procedure was implemented as described in [5]. Standard k-fold cross-validation is inappropriate for time series data because it shuffles observations, thereby destroying their temporal order [5].

The growing-window approach respects this temporal structure. The process begins by training the model on an initial, chronological segment of the data and validating it on the immediately following block of data. In each subsequent fold, the training set is expanded to include the data from the previous validation block, while the next sequential block is used for validation. This method simulates a realistic forecasting scenario where a model is periodically retrained as new data becomes available, ensuring that the model is always validated on "future" data relative to its training set. For each fold, the training data was standardized using scikit-learn's `StandardScaler()`, which removes the mean and scales features to unit variance. The corresponding validation fold was transformed using the scaler fitted on its respective training data. This procedure ensures that the scaling parameters are derived exclusively from the training set, thereby preventing any information leakage from the validation set. As a result, the validation data is transformed using the statistics of the training data, maintaining the integrity of the evaluation and reflecting a realistic forecasting scenario.

D. *Overfitting/Underfitting Prevention*

A multi-faceted strategy was implemented to ensure the models achieved a balance between underfitting and overfitting.

To prevent underfitting, a **grid search** was conducted over a range of hyperparameters, including the number of hidden units, sequence length, weight decay, early stopping and learning rate. This systematic exploration ensures that the final model has sufficient complexity and capacity to capture the underlying patterns present in the data.

To prevent overfitting, two distinct regularization techniques were applied during training:

Weight Decay (L2 Regularization): This was incorporated directly into the Adam optimizer. By adding a penalty term to the loss function proportional to the squared magnitude of the model weights, this technique discourages the learning of overly complex models that might fit the noise in the training data.

Early Stopping: The model's performance on the validation set of each fold was measured at the end of every epoch. If the validation loss did not show improvement for a predefined number of epochs (the patience parameter), the training process was halted. This prevents the model from continuing to train once starts to overfit the training data.

IV. Empirical Procedure

This section details the systematic process used to train, tune, and evaluate the Elman-RNN, Jordan-RNN, and MRNNs across all five datasets. The procedure is designed to ensure reproducibility and a fair comparison between the models.

A. *Experimental setup*

The experiments were conducted using the `PyTorch` library for neural network implementation, supplemented by `scikit-learn` for data scaling and `pandas` for data management. To ensure reproducibility, the random seeds for both `NumPy` and `PyTorch` were fixed to a constant value of 42.

For each dataset, a chronological data partitioning strategy was employed. The data was first split into a 70% training/validation set and a 30% holdout test set. The holdout set was strictly used only for a single final evaluation of the optimized models to provide an unbiased assessment of generalization performance.

Model performance was evaluated using Root Mean Squared Error (RMSE) and Mean Absolute Error (MAE). During the hyperparameter tuning phase, the mean RMSE across cross-validation folds served as the primary metric for hyperparameter selection. Final performance on the holdout set was reported in terms of RMSE on both the scaled data and the inverse-transformed, original scale of the data for better interpretability.

B. *Hyperparameter Tuning*

A comprehensive grid search coupled with a 5-fold growing-window cross-validation was performed to identify the optimal hyperparameters for each of the three RNN architectures on each dataset.

The hyperparameter search space was defined as follows:

- **Hidden Layer Size:** [150, 160, 170, 180, 190]
- **Learning Rate:** [0.001, 0.005, 0.01, 0.015, 0.02]
- **Sequence Length:** [10, 20, 30]
- **Weight Decay:** [1e-5, 1e-4, 1e-3]
- **Early Stopping Patience:** [10, 15, 20] epochs

Throughout all experiments, several architectural and training parameters were held constant to ensure a controlled comparison. These included the Adam optimizer, Mean Squared Error (MSE) loss function, a single recurrent layer, a batch size of 32, and a `tanh` hidden layer activation function. Training was conducted for a maximum of 500 epochs, with early stopping implemented to prevent overfitting.

The described grid search's results are included in the appendix Table VI

C. *Final Training and Evaluation*

The final evaluation followed three steps. First, the hyperparameter combination that achieved the lowest average RMSE during the growing-window cross-validation was identified for each RNN model. Second, a new model was made with these optimal hyperparameters and retrained from scratch on the entire 70% training dataset. Finally, this fully trained model was evaluated one time on the unseen 30% holdout test set to measure its definitive performance.

D. *Model Comparison Framework*

To determine the best-performing RNN architecture for each dataset, the final holdout set RMSE and MAE scores from the three optimized models (Elman, Jordan, and MRNN) were directly compared. The architecture yielding the lowest error on the holdout data was concluded to be the most effective for that particular time series. This complete empirical procedure was consistently repeated for all five datasets, providing a standardized basis for the final results and conclusions.

V. Results

Results This section presents the empirical results obtained from evaluating the Elman, Jordan, and MRNN on the five selected time series datasets. The results are derived from the empirical procedure outlined in the previous section, utilizing the optimal hyperparameters identified via grid search and growing-window cross-validation. Performance is reported in terms of Root Mean Squared Error (RMSE) and Mean Absolute Error (MAE) on the 30% holdout test set, computed both on the scaled data (for model-internal consistency) and the inverse-transformed original scale (for practical interpretability).

For each dataset, the optimal hyperparameters are summarized, followed by a comparison of the three models. The best-performing model per dataset is highlighted based on the lowest holdout RMSE.

	Scaled		**Original**	
Model	**RMSE**	**MAE**	**RMSE**	**MAE**
Elman-RNN	1.0773	0.8584	1.1638	0.9273
Jordan-RNN	1.0825	0.8755	1.1694	0.9459
MRNN	1.0418	0.8479	1.1255	0.9160
Best Model	1.0418	8479	MRNN	

TABLE I
PERFORMANCE METRICS FOR SYNTHETIC AUTOREGRESSIVE STATIONARY DATASET

	Scaled		**Original**	
Model	**RMSE**	**MAE**	**RMSE**	**MAE**
Elman-RNN	69	69	69	69
Jordan-RNN	69	69	69	69
MRNN	69	69	69	69
Best Model	test	test	**[Name]**	**[Name]**

TABLE II
PERFORMANCE METRICS FOR S&P500

	Scaled		**Original**	
Model	**RMSE**	**MAE**	**RMSE**	**MAE**
Elman-RNN	1.0187	0.7427	0.0719	0.0525
Jordan-RNN	1.0167	0.7423	0.0718	0.0524
MRNN	1.0204	0.7437	0.0721	0.0525
Best Model	1.0167	0.7423	Jordan-RNN	

TABLE III
PERFORMANCE METRICS FOR CBOE VIX

	Scaled		**Original**	
Model	**RMSE**	**MAE**	**RMSE**	**MAE**
Elman-RNN	0.8267	0.5947	62.7868	45.1673
Jordan-RNN	0.8307	0.5991	63.0906	45.5011
MRNN	0.8396	0.6020	63.7686	45.7201
Best Model	0.8267	0.5947	Elman-RNN	

TABLE IV
PERFORMANCE METRICS FOR ELECTRICITYLOADDIAGRAMS20112014

VI. Conclusion

	Scaled		**Original**	
Model	**RMSE**	**MAE**	**RMSE**	**MAE**
Elman-RNN	1.0773	0.8584	1.163819	0.927380
Jordan-RNN	69	69	69	69
MRNN	69	69	69	69
Best Model	test	test	**[Name]**	**[Name]**

TABLE V
PERFORMANCE METRICS FOR WEATHER DATASET

References

[1] S. GN, "Backpropagation Through Time (BPTT): Explained With Derivations," https://www.quarkml.com/2023/08/backpropagation-through-time-explained-with-derivations.html, 2025, [Accessed 24-09-2025].

[2] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," 2017. [Online]. Available: https://arxiv.org/abs/1412.6980

[3] J. Brownlee, "Gentle Introduction to the Adam Optimization Algorithm for Deep Learning - MachineLearningMastery.com," https://machinelearningmastery.com/adam-optimization-algorithm-for-deep-learning/, 2023, [Accessed 01-10-2025].

[4] T. Sharma, "COMPARISION STUDY OF ADF vs KPSS TEST," https://medium.com/@tannyasharma21/comparision-study-of-adf-vs-kpss-test-c9d8dec4f62a, 2023, [Accessed 01-10-2025].

[5] S. Shrivastava, "Cross Validation in Time Series," https://medium.com/@soumyachess1496/cross-validation-in-time-series-566ae4981ce4, 2020, [Accessed 30-09-2025].

[6] S. Visagie, "Assignment 3 ml," https://github.com/ThreadAgain/ml-assignment-3, 2025.

Appendix

Dataset 1			
Hyperparameter	**Elman**	**Jordan**	**Multi**
Hidden Size	170	170	170
Learning Rate	0.001	0.001	0.001
Sequence Length	10	10	10
Weight Decay	0.0001	0.0001	0.0001
Patience	15	15	15
Dataset 2			
Hyperparameter	**Elman**	**Jordan**	**Multi**
Hidden Size	150	160	155
Learning Rate	0.002	0.001	0.0015
Sequence Length	15	12	14
Weight Decay	0.0002	0.0001	0.00015
Patience	10	12	11
Dataset 3			
Hyperparameter	**Elman**	**Jordan**	**Multi**
Hidden Size	...	...	...
Learning Rate	...	...	...
Sequence Length	...	...	...
Weight Decay	...	...	...
Patience	...	...	...
Dataset 4			
Hyperparameter	**Elman**	**Jordan**	**Multi**
Hidden Size	...	...	...
Learning Rate	...	...	...
Sequence Length	...	...	...
Weight Decay	...	...	...
Patience	...	...	...
Dataset 5			
Hyperparameter	**Elman**	**Jordan**	**Multi**
Hidden Size	...	...	...
Learning Rate	...	...	...
Sequence Length	...	...	...
Weight Decay	...	...	...
Patience	...	...	...

TABLE VI
OPTIMAL HYPERPARAMETER VALUES FOR ELMAN, JORDAN, AND MULTI RNN MODELS ACROSS FIVE DATASETS.