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

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. 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.

TODO: Mathematically, the hidden state is formulated as

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'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.

TODO:The mathematical formulation for the hidden state is

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.

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 multi-RNNs, it handles both types of feedback simultaneously to support more robust temporal modeling in time-series applications.

III. Methodology

A. Datasets description

TODO: ensure Data Preprocessing justifications

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, volatility shifts, and complex non-linear dependencies, which provide a robust benchmark for comparing the ability of different models to learn temporal patterns 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 multicollinearity among the highly correlated OHLC features while retaining discriminatory power, the Open, High, and Low variables were dropped and replaced with two engineered features: the high-low range and log returns. 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. Feature scaling 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 sharp fluctuations, sudden spikes, and shifting sequential dependencies that create challenging non-stationary signals. 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 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 provides a distinct 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 10,000 sequential observations, indexed from time step 0 to 9,999, 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, requiring no differencing or detrending.

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

For each dataset the models were trained using the Adam optimization algorithm, a method well-suited for training shallow/simple neural networks. The key hyperparameters for the optimizer, namely the learning rate and weight decay determined through the hyperparameter tuning process.

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 [2]. Standard k-fold cross-validation is inappropriate for time series data because it shuffles observations, thereby destroying their temporal order [2].

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

V. Results

VI. Conclusion

[3] [4] [5]

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

[3] Y. Finance, "S&p 500 (^ gspc)," https://finance.yahoo.com/quote/%5EGSPC/, 2025, [Accessed 22-09-2025].

[4] ——, "Chicago board options exchange volatility index (^ vix)," https://finance.yahoo.com/quote/%5EVIX/, 2025, [Accessed 22-09-2025].

[5] W. Xiong, "Electricityloaddiagrams20112014," 2024. [Online]. Available: https://dx.doi.org/10.21227/e40r-rf81

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