**Assignment No: - 4**

**Time Series Prediction Using Recurrent Neural Networks (RNNs)**

**Problem Statement:**

Implementing time series prediction using Recurrent Neural Networks (RNNs) for stock market analysis or weather forecasting.

**Objective:**

* To understand the architecture and functioning of Recurrent Neural Networks.
* To learn how to preprocess time series data for RNN training.
* To implement an RNN model using Keras and TensorFlow for time series prediction.
* To evaluate model performance using test data.
* To visualize predictions and compare them with actual values.

**S/W Packages and H/W apparatus used:**

* **Operating System:** Windows/Linux/MacOS
* **Kernel:** Python 3.x
* **Tools:** Jupyter Notebook, Anaconda, or Google Colab
* **Hardware:** CPU with minimum 4GB RAM; optional GPU for faster processing

**Libraries and packages used:**

* **TensorFlow**
* **Keras**
* **NumPy**
* **Pandas**
* **Matplotlib**

**Theory:**

**Definition:**

Recurrent Neural Networks (RNNs) are a class of artificial neural networks designed for processing sequential data. They have the capability to maintain information about previous inputs in their internal memory, making them particularly suitable for time series prediction tasks.

**Structure:**

Input Layer: Accepts sequences of data points (e.g., stock prices or weather measurements).

Recurrent Layers: Consist of RNN cells (e.g., LSTM or GRU) that process sequences, maintaining a hidden state to capture temporal dependencies.

Fully Connected Layer: Connects the output from the recurrent layers to the final prediction output.

Output Layer: Produces the predicted values for the next time step in the sequence.

**Activation Functions:**

Common activation functions used in RNNs include Tanh and Sigmoid, which help regulate the values flowing through the network and maintain stability during training.

**Memory Cells:**

RNNs utilize memory cells, such as Long Short-Term Memory (LSTM) or Gated Recurrent Unit (GRU) cells, to address issues like vanishing gradients and to retain long-term dependencies in sequential data.

**Methodology:**

1. **Data Acquisition:**

* Load historical stock market data (e.g., stock prices) or weather data (e.g., temperature, humidity) from reliable sources like Yahoo Finance or weather APIs.

1. **Data Preparation:**

* Process the dataset by selecting relevant features and normalizing values to a range between 0 and 1.

1. **Sequence Creation:**

* Create sequences from the time series data to prepare input and output pairs. For example, use the past 60 time steps to predict the next value.

1. **Model Architecture:**

* Create a sequential model using Keras.
* Add one or more recurrent layers (e.g., LSTM or GRU) with a specified number of units.
* Add a fully connected layer and an output layer for regression tasks.

1. **Model Compilation:**

* Compile the model using the Adam optimizer and Mean Squared Error (MSE) as the loss function.

1. **Model Training:**

* Fit the model on the training data while validating on a separate validation set. Track loss and performance metrics.

1. **Model Evaluation:**

* Evaluate the model on the test dataset to measure prediction accuracy.

1. **Prediction Visualization:**

* Plot predicted values against actual values over time to visually assess model performance.

**Advantages:**

* **Sequential Data Handling:** RNNs are specifically designed for sequential data, allowing them to capture temporal dependencies effectively.
* **Long-Term Memory:** With architectures like LSTM and GRU, RNNs can remember information over long sequences, making them ideal for time series tasks where past values influence future outcomes.
* **Flexibility:** RNNs can handle varying input lengths, making them versatile for different types of time series data.
* **Dynamic Computation:** They process sequences of varying lengths without the need for fixed-size input, adapting to the data's natural structure.

**Limitations:**

* **Computational Complexity:** Training RNNs can be computationally intensive, especially with long sequences and large datasets.
* **Vanishing Gradient Problem:** Traditional RNNs can struggle with long-term dependencies due to vanishing gradients, though this is mitigated with LSTM and GRU architectures.
* **Overfitting Risk:** RNNs are prone to overfitting, particularly with small datasets, necessitating regularization techniques.
* **Data Requirements:** Effective training requires a significant amount of historical data to capture the underlying patterns accurately.

**Applications:**

* **Stock Market Analysis: RNNs are commonly used for predicting stock prices and trends based on historical price data.**
* **Weather Forecasting: They are employed to forecast weather conditions by analysing time series data from previous weather patterns.**
* **Natural Language Processing: RNNs are utilized in applications such as language modelling and text generation, where sequential data plays a crucial role.**

Here is an updated theory explanation of the working algorithm based exactly on the provided assignment code for the SimpleRNN stock price prediction model:

**Working / Algorithm:**

**Step 1: Import Necessary Libraries**  
Import libraries needed for numerical operations, data handling, plotting, stock data retrieval, and machine learning with RNN. These include NumPy, Pandas, Matplotlib, yfinance, scikit-learn metrics and preprocessing, and TensorFlow Keras modules.

**Step 2: Download Stock Data**  
Download historical stock data (closing prices) for the specified ticker (e.g., AAPL) using yfinance from the start date "2020-01-01" to end date "2025-01-01".

**Step 3: Extract and Reshape Closing Prices**  
Extract the 'Close' price column from the downloaded data and reshape it into a 2D NumPy array suitable for scaling.

**Step 4: Scale Data using MinMaxScaler**  
Normalize the closing prices to a range between 0 and 1 using MinMaxScaler. This scaling helps improve training stability and convergence of the RNN.

**Step 5: Create Dataset Sequences**  
Define a function create\_dataset to transform the scaled data into sequences of a fixed time step length (60). For each sequence of 60 time steps, the label is the next time step's price. Return arrays X (input sequences) and y (corresponding labels).

**Step 6: Reshape Input for RNN**  
Reshape the sequence data X to a 3D array of shape [samples, time\_steps, features] where features=1 (univariate time series).

**Step 7: Split into Training and Testing Sets**  
Split the data into training (80%) and testing (20%) sets based on the number of samples for both input sequences and labels.

**Step 8: Build the RNN Model**  
Create a Keras sequential model consisting of two stacked SimpleRNN layers with 50 units each. The first layer returns sequences for the second RNN layer. Followed by a Dense layer with a single unit to predict the stock price.

**Step 9: Compile the Model**  
Compile the model using the Adam optimizer and mean squared error (MSE) as the loss function.

**Step 10: Train the Model**  
Train (fit) the model on the training data for 20 epochs with a batch size of 64.

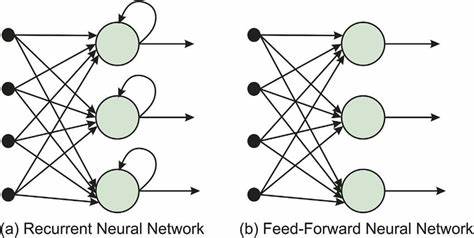
**Step 11: Predict Stock Prices**  
Use the trained model to predict stock prices on the test set sequences. Inverse transform the scaled predictions back to the original price range.

**Step 12: Evaluate Model Performance**  
Inverse transform the test labels to original scale and calculate error metrics: Mean Squared Error (MSE), Root Mean Squared Error (RMSE), and Mean Absolute Error (MAE) to quantify model accuracy on unscaled data.

**Step 13: Visualize Results**  
Plot the original (real) vs predicted stock prices on a time series graph for visual comparison.

**Step 14: Print Data Shape and Metrics**  
Print the shape of the training data and the calculated error metrics.

**Diagram:**

**Conclusion**

Recurrent Neural Networks (RNNs), specifically the SimpleRNN architecture applied here, offer a robust framework for time series forecasting tasks such as stock price prediction. By effectively capturing temporal dependencies in sequential data, these models can leverage historical stock prices to generate future price estimates. The assignment demonstrated that through proper data preprocessing (scaling and sequence generation), careful model design with stacked SimpleRNN layers, and appropriate training strategies, the model can produce meaningful predictions that approximate real stock price movements.

While RNNs are powerful, the model’s accuracy depends heavily on factors including data quality, sequence length, network architecture, and hyperparameter tuning. Challenges such as overfitting or computational resource demands are present but can be managed with suitable techniques. The model evaluation using unscaled error metrics (MSE, RMSE, MAE) and visual comparison confirmed the model's ability to learn underlying stock price patterns reasonably well.

Overall, this approach highlights the practical applicability of SimpleRNNs in financial forecasting, providing a foundation that can be further enhanced with more advanced recurrent models (like LSTM or GRU) or additional market features for improved predictive performance.