ASSIGNMENT NO. 03

TITLE: RECURRENT NEURAL NETWORK (RNN)

Problem Statement :

Use the Google stock prices dataset and design a time series analysis and prediction system using RNN..

Objective :

To understand and implement:

1. RNN
2. Analyze historical stock price data and predict future prices based on the learned patterns and trends in the data using Time series analysis and RNN based model.

Theory:

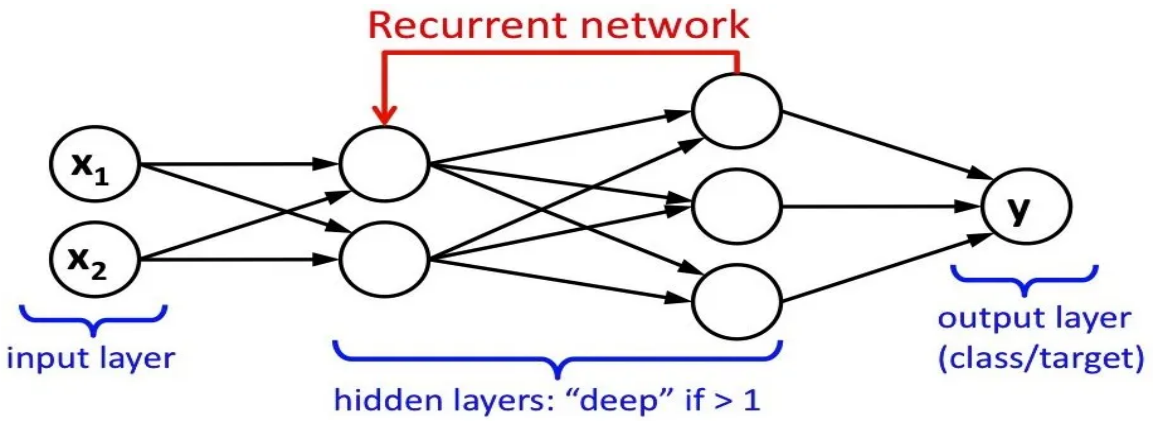
What is RNN?

RNN or Recurrent Neural Network is a type of neural network architecture that is specifically designed to process data that has sequential dependencies and temporal patterns., such as time series data or natural language data. Unlike traditional feedforward neural networks, the key component of an RNN is the recurrent connection, which allows information to flow from one time step to the next. This enables the network to capture long-term dependencies in the data. RNNs also have a hidden state or memory that can store information about the past inputs and use it to influence future predictions.

RNNs are widely used in various applications, including natural language processing (NLP), speech recognition, machine translation, and time series analysis. In the context of time series analysis, RNNs can be used to model and forecast future values based on historical data patterns.

Recurrent Neural Network Architecture :

The architecture of a Recurrent Neural Network (RNN) consists of three main components: the input layer, the recurrent layer, and the output layer



1. **Input Layer:**

   The input layer or the first layer receives sequential data as input. In the case of time series analysis, each time step's input could include features such as historical stock prices, trading volumes, or other relevant factors. The input layer passes the input data to the recurrent layer.

2. **Recurrent Layer:**

   The recurrent layer is the core component of the RNN architecture. It contains recurrent units, often referred to as memory cells or hidden units. Each recurrent unit processes a time step's input and its own hidden state from the previous time step. The hidden state serves as the memory of the network, capturing and storing information from previous time steps. The recurrent layer applies weights to the input and the previous hidden state, performs calculations, and generates an updated hidden state. The updated hidden state is passed to the next time step, allowing the network to retain information across multiple time steps.

   Formula for calculating the current state

$$h_t = f(h_{t-1}, x_t)$$

   where:

   h_t => current state

   h_{t-1} => previous state

   x_t => input state

   The recurrent layer can have multiple recurrent units, and the choice of the specific recurrent unit type, such as the vanilla RNN, LSTM (Long Short-Term Memory), or GRU (Gated Recurrent Unit), depends on the specific requirements of the task.

3. **Output Layer:**

   The output layer receives the final hidden state from the recurrent layer and generates the network's prediction or output. In time series analysis, the output layer might produce a single value prediction for the next time step or multiple predictions for future time steps.

   The output can be calculated as:

$$y_t = W_{hy} h_t$$

   where:

   y_t => output

   W_{hy} => weight at output layer

   The output layer can apply additional transformations, such as activation functions, to produce the desired output format. In the Keras Documentation, it is specified that the default activation function for LSTM based RNN is tanh and the default recurrent activation function is sigmoid. Therefore the formula for those default activation functions are:

   a. Tanh

$$f(x) = \frac{\left(e^{x} - e^{-x}\right)}{\left(e^{x} + e^{-x}\right)}$$

   b. Sigmoid

$$f(x) = \frac{1}{1 + e^{-x}}$$

Steps to create an RNN Model for our use case:

Step 1: Import the necessary libraries

We will be using TensorFlow Keras for building our RNN models. Apart from the model architecture, we will be needing numpy, pandas, sklearn for data preparation and matplotlib, seaborn for data visualization.

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import warnings
import datetime
import seaborn as sns
from sklearn.preprocessing import MinMaxScaler
from sklearn.decomposition import PCA
from sklearn.model_selection import train_test_split
from tensorflow.keras import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.layers import LSTM
from sklearn.metrics import r2_score
```

Step 2 : Loading the Dataset

Install the Google stock prices dataset from kaggle and load it using pandas

```
data=pd.read_csv("/content/Google_Stock_Price_Train.csv")
```

Step 3 : Data Preprocessing

In the dataset, the Volume column had thousand separated values, which is taken as string when usually loading to Pandas DataFrame. Convert such columns into numeric data

```
data['Volume']=data['Volume'].str.replace(',','').astype(float)
```

We don't encounter null values in this dataset and we do not need to handle outliers.

Step 4 : Feature Selection

We have to normalize or scale the features to ensure the data are within a similar range. This step is crucial for effective training of the RNN model. Drawing the Correlation Matrix and finding the Correlation Coefficient checking are a few mechanisms to check the relationship between the different features with the predicting attribute.

We can draw the correlation matrix using seaborn as follows:

```
plt.figure(figsize=(6,6))

sns.heatmap(data.corr())
```

Step 5 : Sequence Generation:

To develop an RNN model, we have to convert the dataset into sequences of inputs and corresponding output targets. To do that, first we have to find a pattern in the data available and define the number of timesteps according to the pattern. In our particular case, since we are dealing with time series data, we have to specify how many past data points we will be considering when generating the sequence. Create input sequences by sliding a window of the defined sequence length through the dataset. Associate each input sequence with its corresponding output target (e.g., futures's stock price)

Step 6 : Splitting the data and Creating the model

Split the dataset into training and testing sets. The training set will be used to train the RNN model, while the testing set will be used for evaluating its performance.

```
  X_train, X_test, y_train, y_test =
train_test_split(X,y,test_size=0.2,random_state=50)
```

We will be choosing RNN's variant, LSTM for our use case. The model's architecture will consist of layers including Tensorflow Keras LSTM and Dense layers with default activation function for LSTM being tanh and default recurrent activation function being sigmoid.

```
  model = Sequential()

  model.add(LSTM(612, input_shape=(n_steps,2)))

  model.add(Dense(50, activation='relu'))

  model.add(Dense(50, activation='relu'))

  model.add(Dense(30, activation='relu'))

  model.add(Dense(1))
```

After adding the layers to the model, we can compile it using the compile() method, which sets the optimizer, loss function, and evaluation metrics for the model. Finally, we can train the model on the training data using the fit() method and evaluate its performance on the test data using the evaluate() method.

Step 6 : Evaluating the model

We evaluate the trained model using the testing dataset. Here we have to calculate relevant metrics such as mean squared error (MSE), mean absolute error (MAE), to assess the model's predictive accuracy in predictions.

```
 mse, mae = model.evaluate(X_test, y_test, verbose=0)
```

Step 7 : Prediction

We utilize the trained RNN model to make predictions on unseen data or future time steps by providing the necessary input features for the model, such as previous stock prices and other relevant factors. We can evaluate the predictions and compare them with actual values to assess the model's performance in a real-world scenario using mode.predict().

Conclusion:

We were able to understand and implement RNN models for time series analysis and forecasting tasks. The developed model provided insights into the Google stock market and its potential future directions. It showcased the capability to leverage deep learning techniques, specifically RNNs, to make accurate predictions and informed decisions in the financial domain.