Assignment:4

AIM: Recurrent neural network (RNN): Use the Google stock prices dataset and design a time series analysis and prediction system using RNN.

OBJECTIVES: We should be able to design stock price prediction system by using Recurrent Neural Network using Google stock prices dataset.

PREREQUISITE:
1. Basic of programming language
2. Concept of RNN

THEORY: A Recurrent neural network (RNN) is a deep learning model that is trained to process and convert a sequential data input into a specific sequential data output. Sequential data is data—such as words, sentences, or time-series data—where sequential components interrelate based on complex semantics and syntax rules. An RNN is a software system that consists of many interconnected components mimicking how humans perform sequential data conversions, such as translating text from one language to another. RNNs are largely being replaced by transformer-based artificial intelligence (AI) and large language models (LLM), which are much more efficient in sequential data processing.

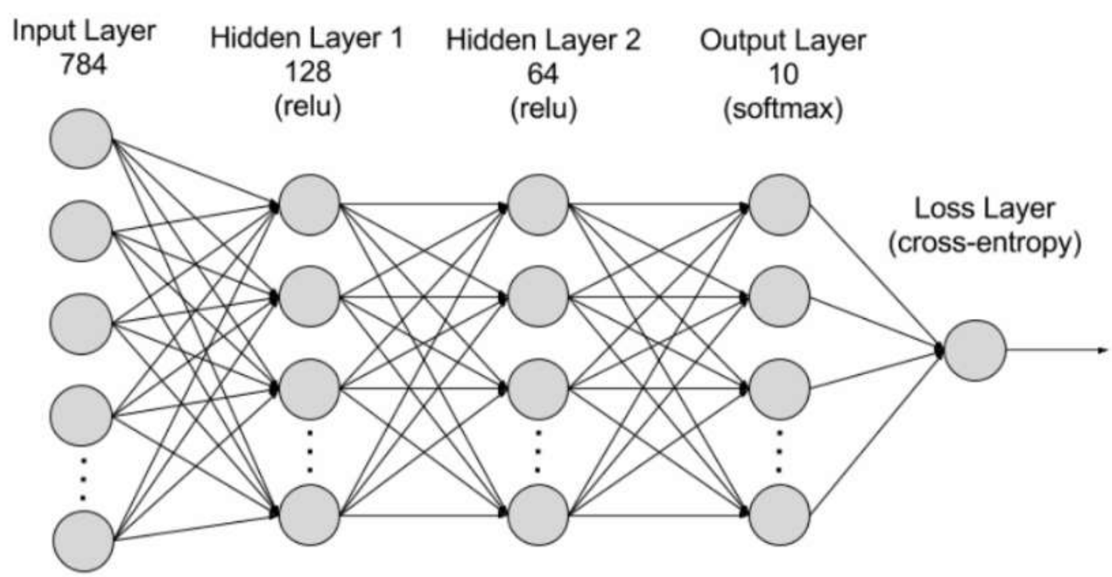


Figure: Working of Recurrent Neural Network

RNNs are made of neurons: data-processing nodes that work together to perform complex tasks. The neurons are organized as input, output, and hidden layers. The input layer receives the information to process, and the output layer provides the result. Data processing, analysis, and prediction take place in the hidden layer. Hidden layer RNNs work by passing the sequential data that they receive to the hidden layers one-step at a time. However, they also have a self-looping or recurrent workflow: the hidden layer can remember and use previous inputs for future predictions in a short-term memory component. It uses the current input and the stored memory to predict the next sequence. This makes RNNs useful in speech recognition, machine translation, and other language modelling tasks.

Machine learning (ML) engineers train deep neural networks like RNNs by feeding the model with training data and refining its performance. In ML, the neuron's weights are signals to determine how influential the information learned during training is when predicting the output. Each layer in an RNN shares the same weight. There is need to adjust weights to improve prediction accuracy. It uses a technique called backpropagation through time (BPTT) to calculate model error and adjust its weight accordingly. BPTT rolls back the output to the previous time step and recalculates the error rate. This way, it can identify which hidden state in the sequence is causing a significant error and readjust the weight to reduce the error margin.

TYPES OF RECURRENT NEURAL NETWORKS:

RNNs are often characterized by one-to-one architecture: one input sequence is associated with one output. However, one can flexibly adjust them into various configurations for specific purposes. The following are several common RNN types.

1. One-to-many: This RNN type channels one input to several outputs. It enables linguistic Applications like image captioning by generating a sentence from a single keyword.

2. Many-to-many: The model uses multiple inputs to predict multiple outputs. For example, You can create a language translator with an RNN, which analyses a sentence and correctly Structures the words in a different language.

3. Many-to-one: Several inputs are mapped to an output. This is helpful in applications like sentiment analysis, where the model predicts customers' sentiments like positive, negative, and neutral from input testimonials.

RNNs are one of several different neural network architectures:

Recurrent neural network vs. feed-forward neural network

Like RNNs, feed-forward neural networks are artificial neural networks that pass information from one end to the other end of the architecture. A feed-forward neural network can perform simple classification, regression, or recognition tasks, but it cannot remember the previous input that it has processed. The RNN overcomes this memory limitation by including a hidden memory state in the neuron.

Recurrent neural network vs. convolutional neural networks

Convolutional neural networks are artificial neural networks that are designed to process spatial data. One can use convolutional neural networks to extract spatial information from videos and images by passing them through a series of convolutional and pooling layers in the neural network. RNNs are designed to capture long-term dependencies in sequential data

CONCLUSION: We have successfully developed a time series analysis and prediction system using RNN on Google stock prices dataset.

Source Code:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.preprocessing import MinMaxScaler
import yfinance as yf

# RNN Implementation from scratch
class SimpleRNN:
    def __init__(self, input_size, hidden_size, output_size):
        # Initialize weights and biases
        self.hidden_size = hidden_size

        # Weight matrices
        self.Wxh = np.random.randn(hidden_size, input_size) * 0.01  # Input to hidden
        self.Whh = np.random.randn(hidden_size, hidden_size) * 0.01  # Hidden to hidden
        self.Why = np.random.randn(output_size, hidden_size) * 0.01  # Hidden to output

        # Biases
        self.bh = np.zeros((hidden_size, 1))  # Hidden bias
        self.by = np.zeros((output_size, 1))  # Output bias

    def forward(self, inputs):
        # Forward pass through RNN
        h = np.zeros((self.hidden_size, 1))
        self.hs = [h]  # Store hidden states
```

```python
        for x in inputs:
            h = np.tanh(np.dot(self.Wxh, x) + np.dot(self.Whh, h) + self.bh)
            self.hs.append(h)

        y = np.dot(self.Why, h) + self.by
        return y, h

    def train(self, inputs, targets, learning_rate=0.01):
        # Simple backpropagation through time (BPTT)
        h = np.zeros((self.hidden_size, 1))
        self.hs = [h]

        # Forward pass
        for x in inputs:
            h = np.tanh(np.dot(self.Wxh, x) + np.dot(self.Whh, h) + self.bh)
            self.hs.append(h)

        y = np.dot(self.Why, h) + self.by

        # Backward pass
        dy = y - targets  # Shape: (1, 1)
        dWhy = np.dot(dy, h.T)  # Shape: (1, hidden_size)
        dby = dy  # Shape: (1, 1)

        dh = np.dot(self.Why.T, dy)  # Shape: (hidden_size, 1)
        dWxh, dWhh, dbh = 0, 0, 0

        for t in range(len(inputs)-1, -1, -1):
            dtanh = (1 - self.hs[t+1]**2) * dh  # Shape: (hidden_size, 1)
            dbh += dtanh  # Shape: (hidden_size, 1)
```

```
            dWxh += np.dot(dtanh, inputs[t].T)  # Shape: (hidden_size, 1)
            if t > 0:
                dWhh += np.dot(dtanh, self.hs[t].T)  # Shape: (hidden_size, hidden_size)
            dh = np.dot(self.Whh.T, dtanh)  # Shape: (hidden_size, 1)

        # Update weights
        self.Wxh -= learning_rate * dWxh
        self.Whh -= learning_rate * dWhh
        self.Why -= learning_rate * dWhy
        self.bh -= learning_rate * dbh
        self.by -= learning_rate * dby

        return y

# Data handling functions
def get_stock_data():
    stock_symbol = 'GOOGL'
    start_date = '2015-01-01'
    end_date = '2023-12-31'
    df = yf.download(stock_symbol, start=start_date, end=end_date)
    return df

def preprocess_data(df):
    data = df['Close'].values.reshape(-1, 1)
    scaler = MinMaxScaler(feature_range=(0, 1))
    scaled_data = scaler.fit_transform(data)
    return scaled_data, scaler

def create_sequences(data, seq_length):
    X, y = [], []
```

```
    for i in range(len(data) - seq_length):
        X.append(data[i:(i + seq_length)])
        y.append(data[i + seq_length])
    return np.array(X), np.array(y)

# Main execution
def main():
    # Parameters
    SEQ_LENGTH = 20
    HIDDEN_SIZE = 10
    TRAIN_SIZE = 0.8
    EPOCHS = 50

    # Get and prepare data
    df = get_stock_data()
    scaled_data, scaler = preprocess_data(df)

    # Split data
    train_size = int(len(scaled_data) * TRAIN_SIZE)
    train_data = scaled_data[:train_size]
    test_data = scaled_data[train_size:]

    # Create sequences
    X_train, y_train = create_sequences(train_data, SEQ_LENGTH)
    X_test, y_test = create_sequences(test_data, SEQ_LENGTH)

    # Initialize RNN
    rnn = SimpleRNN(input_size=1, hidden_size=HIDDEN_SIZE, output_size=1)

    # Training
```

```
train_losses = []
for epoch in range(EPOCHS):
    total_loss = 0
    for i in range(len(X_train)):
        inputs = X_train[i].reshape(SEQ_LENGTH, 1, 1)  # Shape: (SEQ_LENGTH, 1, 1)
        target = y_train[i].reshape(1, 1)  # Shape: (1, 1)
        pred = rnn.train(inputs, target)
        total_loss += np.mean((pred - target)**2)

    avg_loss = total_loss / len(X_train)
    train_losses.append(avg_loss)
    if epoch % 10 == 0:
        print(f'Epoch {epoch}, Loss: {avg_loss:.6f}')

# Make predictions
train_predict = []
for i in range(len(X_train)):
    pred, _ = rnn.forward(X_train[i].reshape(SEQ_LENGTH, 1, 1))
    train_predict.append(pred[0,0])

test_predict = []
for i in range(len(X_test)):
    pred, _ = rnn.forward(X_test[i].reshape(SEQ_LENGTH, 1, 1))
    test_predict.append(pred[0,0])

# Inverse transform
train_predict = scaler.inverse_transform(np.array(train_predict).reshape(-1, 1))
y_train_inv = scaler.inverse_transform(y_train)
test_predict = scaler.inverse_transform(np.array(test_predict).reshape(-1, 1))
y_test_inv = scaler.inverse_transform(y_test)
```

```
# Calculate RMSE
train_rmse = np.sqrt(np.mean((train_predict - y_train_inv)**2))
test_rmse = np.sqrt(np.mean((test_predict - y_test_inv)**2))
print(f'Train RMSE: {train_rmse:.2f}')
print(f'Test RMSE: {test_rmse:.2f}')

# Plotting
plt.figure(figsize=(15, 6))
plt.plot(df.index[SEQ_LENGTH:train_size], y_train_inv, label='Actual Train')
plt.plot(df.index[SEQ_LENGTH:train_size], train_predict, label='Predicted Train')
test_start_idx = train_size + SEQ_LENGTH
plt.plot(df.index[test_start_idx:], y_test_inv, label='Actual Test')
plt.plot(df.index[test_start_idx:], test_predict, label='Predicted Test')
plt.title('Google Stock Price Prediction using Custom RNN')
plt.xlabel('Date')
plt.ylabel('Price')
plt.legend()
plt.xticks(rotation=45)
plt.grid(True)
plt.show()

plt.figure(figsize=(10, 5))
plt.plot(train_losses, label='Training Loss')
plt.title('Training Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()
plt.grid(True)
plt.show()
```

```
if __name__ == "__main__":
    main()
```

Output:

[********************100%***********************] 1 of 1 completed

Epoch 0, Loss: 0.005685

Epoch 10, Loss: 0.003304

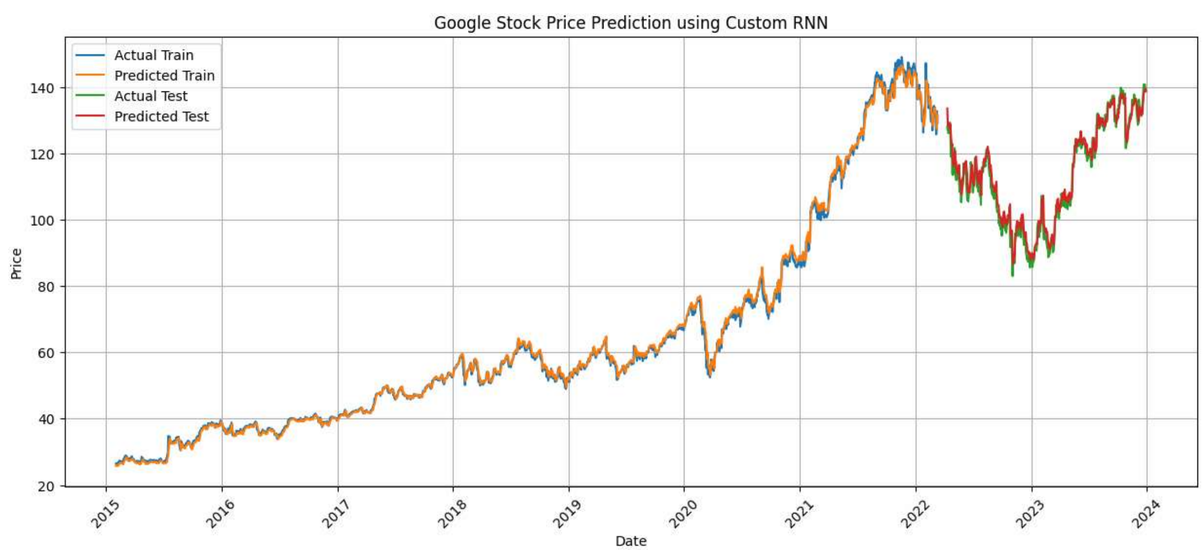
Epoch 20, Loss: 0.000104

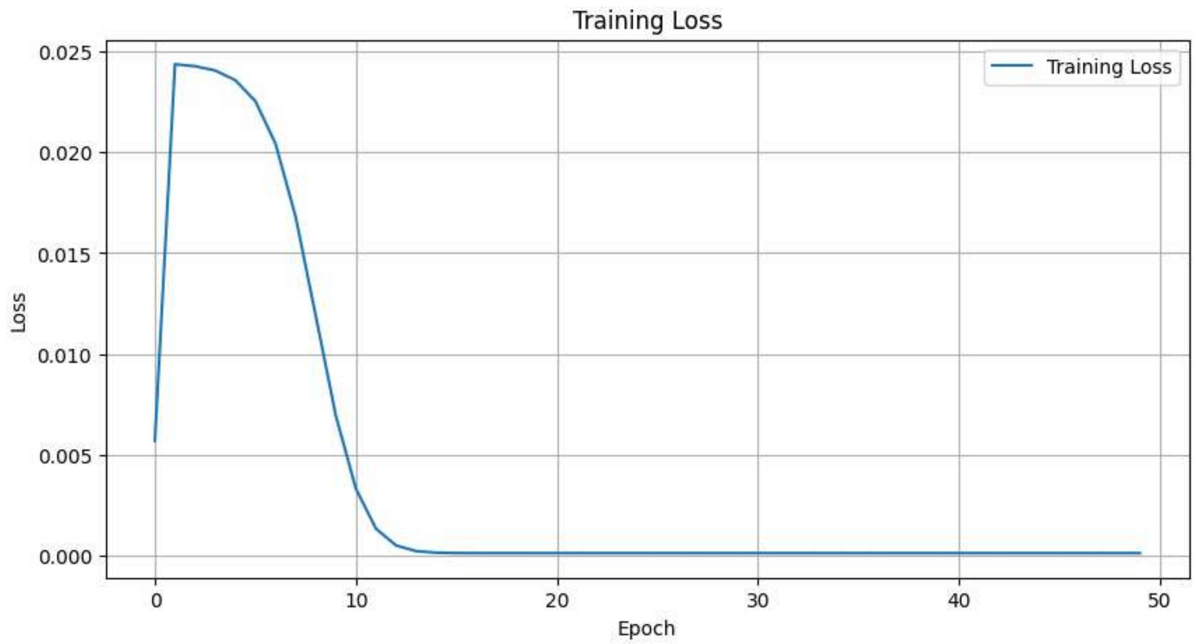
Epoch 30, Loss: 0.000104

Epoch 40, Loss: 0.000104

Train RMSE: 1.41

Test RMSE: 2.86




Training Loss
Training Loss
Loss
Epoch
0.025
0.020
0.015
0.010
0.005
0.000
0
10
20
30
40
50