Neural Network & Deep Learning

Step 1: Data Preprocessing

1.Load the dataset and focus on the 'Close' price column.
2.Normalize the data using Min-Max scaling.
3.Split the dataset into training (80%) and testing (20%) sets.

```
# Import necessary libraries
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.preprocessing import MinMaxScaler
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import SimpleRNN, Dense
from sklearn.metrics import mean_absolute_error, mean_squared_error

# Load the dataset
data = pd.read_csv('/content/HistoricalQuotes.csv')

data.head()
```


```
{"summary":"{\n  \"name\": \"data\",\n  \"rows\": 2518,\n  \"fields\":
[\n    {\n      \"column\": \"Date\",\n      \"properties\": {\n
\"dtype\": \"object\",\n        \"num_unique_values\": 2518,\n
\"samples\": [\n          \"06/07/2011\",\n          \"07/12/2018\",\n
\"08/21/2014\"\n        ],\n        \"semantic_type\": \"\",\n
\"description\": \"\"\n      }\n    },\n    {\n      \"column\": \"
Close/Last\",\n      \"properties\": {\n        \"dtype\":
\"string\",\n        \"num_unique_values\": 2417,\n
\"samples\": [\n          \" $187.97\",\n          \" $190.92\",\n
\" $64.1386\"\n        ],\n        \"semantic_type\": \"\",\n
\"description\": \"\"\n      }\n    },\n    {\n      \"column\": \"
Volume\",\n      \"properties\": {\n        \"dtype\": \"number\",\n
\"std\": 56631126,\n        \"min\": 11362050,\n        \"max\":
462442329,\n        \"num_unique_values\": 2514,\n        \"samples\":
[\n          23637310,\n          29124140,\n          29139100\n
],\n        \"semantic_type\": \"\",\n        \"description\": \"\"\n
}\n    },\n    {\n      \"column\": \" Open\",\n      \"properties\":
{\n        \"dtype\": \"string\",\n        \"num_unique_values\":
2415,\n        \"samples\": [\n          \" $186.29\",\n          \"
$187.71\",\n          \" $61.4\"\n        ],\n
\"semantic_type\": \"\",\n        \"description\": \"\"\n      }\
n    },\n    {\n      \"column\": \" High\",\n      \"properties\": {\
n        \"dtype\": \"string\",\n        \"num_unique_values\": 2399,\
n        \"samples\": [\n          \" $54.8286\",\n          \"
$53.0971\",\n          \" $108.94\"\n        ],\n
\"semantic_type\": \"\",\n        \"description\": \"\"\n      }\
```

```
n    },\n    {\n      \"column\": \" Low\",\n      \"properties\": {\n
\"dtype\": \"string\",\n        \"num_unique_values\": 2408,\n
\"samples\": [\n          \" $47.6468\",\n          \" $50.9314\",\n
\" $84.3857\"\n        ],\n        \"semantic_type\": \"\",\n
\"description\": \"\"\n      }\n    }\n  ]\
n}","type":"dataframe","variable_name":"data"}
```

```
data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 2518 entries, 0 to 2517
Data columns (total 6 columns):
 #   Column       Non-Null Count  Dtype
---  ------       --------------  -----
 0   Date         2518 non-null   object
 1    Close/Last  2518 non-null   object
 2    Volume      2518 non-null   int64
 3    Open        2518 non-null   object
 4    High        2518 non-null   object
 5    Low         2518 non-null   object
dtypes: int64(1), object(5)
memory usage: 118.2+ KB
```

```
# Strip any leading or trailing whitespace from column names
data.columns = data.columns.str.strip()

# Remove the dollar sign and convert to numeric in the 'Close/Last'
column
data['Close/Last'] = data['Close/Last'].replace('[\$,]', '',
regex=True).astype(float)

# Rename 'Close/Last' to 'Close' for easier access
data.rename(columns={'Close/Last': 'Close'}, inplace=True)

# Focus on the 'Close' price column
data = data[['Close']]

# Normalize the data using Min-Max scaling
scaler = MinMaxScaler(feature_range=(0, 1))
data_scaled = scaler.fit_transform(data)

# Split the dataset into training (80%) and testing (20%) sets
train_size = int(len(data_scaled) * 0.8)
train_data = data_scaled[:train_size]
test_data = data_scaled[train_size:]
```

Step 2: Create Training Sequences

1.Convert the 'Close' prices into sequences.
2.Define a sequence length (e.g., 60 days), where each sequence will predict the next day's price.

```
# Define a function to create sequences
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)

# Set sequence length
sequence_length = 60

# Create sequences for training
X_train, y_train = create_sequences(train_data, sequence_length)
X_test, y_test = create_sequences(test_data, sequence_length)
```

Step 3: Build the RNN Model

1.Define an RNN model with the following architecture:

- An RNN layer with 50 units
- A Dense layer with 1 unit for regression output

2.Use the mean squared error (MSE) loss function and the Adam optimizer.

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import SimpleRNN, Dense

# Build the RNN model
model = Sequential()
model.add(SimpleRNN(units=50, activation='relu',
input_shape=(X_train.shape[1], 1)))
model.add(Dense(units=1))  # Output layer for regression

# Compile the model
model.compile(optimizer='adam', loss='mean_squared_error')
```

```
/usr/local/lib/python3.10/dist-packages/keras/src/layers/rnn/
rnn.py:204: UserWarning: Do not pass an `input_shape`/`input_dim`
argument to a layer. When using Sequential models, prefer using an
`Input(shape)` object as the first layer in the model instead.
  super().__init__(**kwargs)
```

Step 4: Train the Model

1.Train the model on the training set for 50 epochs with a batch size of 32.
2.Use validation data to check for overfitting.

```
# Train the model
history = model.fit(X_train, y_train, epochs=50, batch_size=32,
validation_data=(X_test, y_test))

Epoch 1/50
62/62 ━━━━━━━━━━━━━━━━━━━━ 3s 19ms/step - loss: 0.0217 - val_loss:
2.3473e-04
Epoch 2/50
62/62 ━━━━━━━━━━━━━━━━━━━━ 2s 11ms/step - loss: 1.3517e-04 - val_loss:
7.3551e-05
Epoch 3/50
62/62 ━━━━━━━━━━━━━━━━━━━━ 1s 10ms/step - loss: 1.2287e-04 - val_loss:
5.2530e-05
Epoch 4/50
62/62 ━━━━━━━━━━━━━━━━━━━━ 1s 10ms/step - loss: 1.1968e-04 - val_loss:
6.7063e-05
Epoch 5/50
62/62 ━━━━━━━━━━━━━━━━━━━━ 1s 12ms/step - loss: 1.1380e-04 - val_loss:
8.0613e-05
Epoch 6/50
62/62 ━━━━━━━━━━━━━━━━━━━━ 1s 12ms/step - loss: 1.2167e-04 - val_loss:
5.2255e-05
Epoch 7/50
62/62 ━━━━━━━━━━━━━━━━━━━━ 1s 10ms/step - loss: 1.0686e-04 - val_loss:
5.8090e-05
Epoch 8/50
62/62 ━━━━━━━━━━━━━━━━━━━━ 1s 11ms/step - loss: 1.0509e-04 - val_loss:
3.2444e-05
Epoch 9/50
62/62 ━━━━━━━━━━━━━━━━━━━━ 1s 11ms/step - loss: 1.1680e-04 - val_loss:
2.6791e-05
Epoch 10/50
62/62 ━━━━━━━━━━━━━━━━━━━━ 1s 11ms/step - loss: 1.2956e-04 - val_loss:
5.9514e-05
Epoch 11/50
62/62 ━━━━━━━━━━━━━━━━━━━━ 1s 12ms/step - loss: 1.0384e-04 - val_loss:
3.2917e-05
Epoch 12/50
62/62 ━━━━━━━━━━━━━━━━━━━━ 1s 16ms/step - loss: 1.0619e-04 - val_loss:
3.1764e-05
Epoch 13/50
62/62 ━━━━━━━━━━━━━━━━━━━━ 1s 17ms/step - loss: 9.8079e-05 - val_loss:
3.0718e-05
Epoch 14/50
62/62 ━━━━━━━━━━━━━━━━━━━━ 1s 17ms/step - loss: 9.7869e-05 - val_loss:
2.7786e-05
Epoch 15/50
62/62 ━━━━━━━━━━━━━━━━━━━━ 1s 14ms/step - loss: 1.0038e-04 - val_loss:
2.7114e-05
Epoch 16/50
```

```
62/62 ━━━━━━━━━━━━━━━━━━━━ 1s 10ms/step - loss: 8.6643e-05 - val_loss:
4.6141e-05
Epoch 17/50
62/62 ━━━━━━━━━━━━━━━━━━━━ 1s 11ms/step - loss: 9.7529e-05 - val_loss:
2.1410e-05
Epoch 18/50
62/62 ━━━━━━━━━━━━━━━━━━━━ 1s 10ms/step - loss: 8.1921e-05 - val_loss:
2.8714e-05
Epoch 19/50
62/62 ━━━━━━━━━━━━━━━━━━━━ 1s 10ms/step - loss: 9.3262e-05 - val_loss:
3.2322e-05
Epoch 20/50
62/62 ━━━━━━━━━━━━━━━━━━━━ 1s 10ms/step - loss: 8.9064e-05 - val_loss:
2.4410e-05
Epoch 21/50
62/62 ━━━━━━━━━━━━━━━━━━━━ 1s 10ms/step - loss: 8.7331e-05 - val_loss:
2.7620e-05
Epoch 22/50
62/62 ━━━━━━━━━━━━━━━━━━━━ 1s 10ms/step - loss: 9.0478e-05 - val_loss:
2.7535e-05
Epoch 23/50
62/62 ━━━━━━━━━━━━━━━━━━━━ 1s 10ms/step - loss: 9.8544e-05 - val_loss:
2.9845e-05
Epoch 24/50
62/62 ━━━━━━━━━━━━━━━━━━━━ 2s 14ms/step - loss: 8.7457e-05 - val_loss:
2.3545e-05
Epoch 25/50
62/62 ━━━━━━━━━━━━━━━━━━━━ 1s 18ms/step - loss: 8.3410e-05 - val_loss:
2.3499e-05
Epoch 26/50
62/62 ━━━━━━━━━━━━━━━━━━━━ 1s 17ms/step - loss: 9.2941e-05 - val_loss:
2.0245e-05
Epoch 27/50
62/62 ━━━━━━━━━━━━━━━━━━━━ 1s 11ms/step - loss: 9.2942e-05 - val_loss:
2.1580e-05
Epoch 28/50
62/62 ━━━━━━━━━━━━━━━━━━━━ 1s 10ms/step - loss: 8.5037e-05 - val_loss:
2.0682e-05
Epoch 29/50
62/62 ━━━━━━━━━━━━━━━━━━━━ 1s 11ms/step - loss: 8.1920e-05 - val_loss:
2.4513e-05
Epoch 30/50
62/62 ━━━━━━━━━━━━━━━━━━━━ 1s 10ms/step - loss: 8.7375e-05 - val_loss:
2.4059e-05
Epoch 31/50
62/62 ━━━━━━━━━━━━━━━━━━━━ 1s 10ms/step - loss: 8.2249e-05 - val_loss:
3.1016e-05
Epoch 32/50
62/62 ━━━━━━━━━━━━━━━━━━━━ 1s 11ms/step - loss: 8.2606e-05 - val_loss:
4.3850e-05
```

```
Epoch 33/50
62/62 ━━━━━━━━━━━━━━━━━━━━ 1s 10ms/step - loss: 1.1833e-04 - val_loss:
2.6386e-05
Epoch 34/50
62/62 ━━━━━━━━━━━━━━━━━━━━ 1s 10ms/step - loss: 9.9765e-05 - val_loss:
1.6603e-05
Epoch 35/50
62/62 ━━━━━━━━━━━━━━━━━━━━ 1s 10ms/step - loss: 8.2727e-05 - val_loss:
1.6184e-05
Epoch 36/50
62/62 ━━━━━━━━━━━━━━━━━━━━ 1s 11ms/step - loss: 7.9632e-05 - val_loss:
2.1306e-05
Epoch 37/50
62/62 ━━━━━━━━━━━━━━━━━━━━ 1s 10ms/step - loss: 7.2789e-05 - val_loss:
1.8524e-05
Epoch 38/50
62/62 ━━━━━━━━━━━━━━━━━━━━ 1s 13ms/step - loss: 6.8535e-05 - val_loss:
1.7660e-05
Epoch 39/50
62/62 ━━━━━━━━━━━━━━━━━━━━ 2s 18ms/step - loss: 8.2113e-05 - val_loss:
1.5243e-05
Epoch 40/50
62/62 ━━━━━━━━━━━━━━━━━━━━ 1s 18ms/step - loss: 7.2636e-05 - val_loss:
1.4894e-05
Epoch 41/50
62/62 ━━━━━━━━━━━━━━━━━━━━ 1s 12ms/step - loss: 9.1864e-05 - val_loss:
1.9157e-05
Epoch 42/50
62/62 ━━━━━━━━━━━━━━━━━━━━ 1s 11ms/step - loss: 7.0995e-05 - val_loss:
2.2536e-05
Epoch 43/50
62/62 ━━━━━━━━━━━━━━━━━━━━ 1s 11ms/step - loss: 8.4717e-05 - val_loss:
1.5372e-05
Epoch 44/50
62/62 ━━━━━━━━━━━━━━━━━━━━ 1s 11ms/step - loss: 7.1930e-05 - val_loss:
1.6429e-05
Epoch 45/50
62/62 ━━━━━━━━━━━━━━━━━━━━ 1s 11ms/step - loss: 7.9133e-05 - val_loss:
2.3511e-05
Epoch 46/50
62/62 ━━━━━━━━━━━━━━━━━━━━ 1s 11ms/step - loss: 7.3166e-05 - val_loss:
2.0100e-05
Epoch 47/50
62/62 ━━━━━━━━━━━━━━━━━━━━ 1s 11ms/step - loss: 9.9361e-05 - val_loss:
2.3573e-05
Epoch 48/50
62/62 ━━━━━━━━━━━━━━━━━━━━ 1s 10ms/step - loss: 6.5502e-05 - val_loss:
1.8961e-05
Epoch 49/50
```

```
62/62 ━━━━━━━━━━━━━━━━━━━━ 1s 11ms/step - loss: 9.8552e-05 - val_loss:
2.0059e-05
Epoch 50/50
62/62 ━━━━━━━━━━━━━━━━━━━━ 1s 11ms/step - loss: 7.4312e-05 - val_loss:
3.7277e-05
```

Step 5: Make Predictions

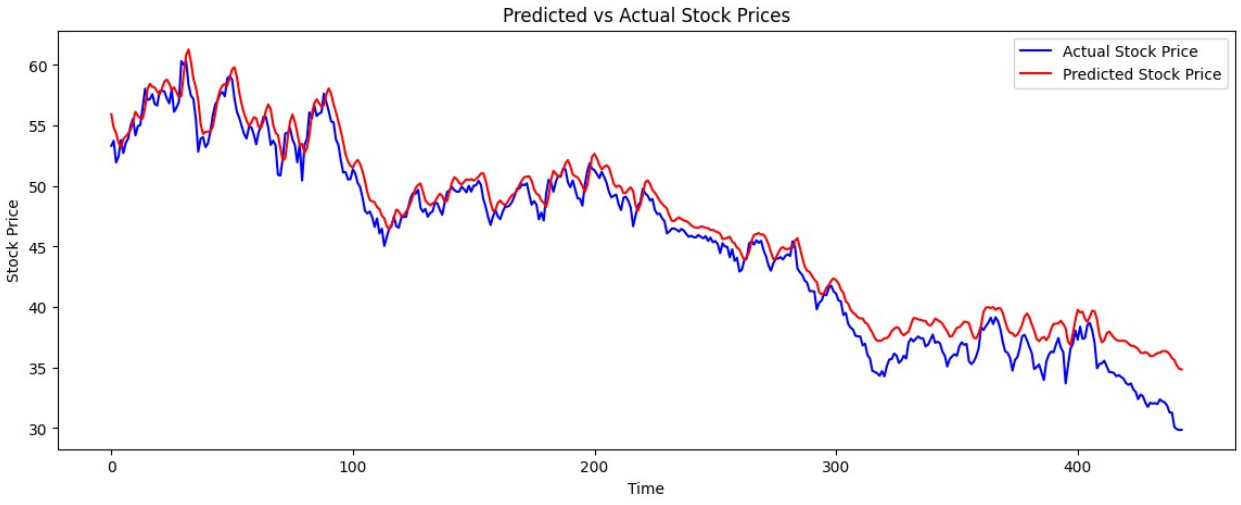
1.Predict the stock prices on the test set.
2.Transform the results back to the original scale if normalization was applied.
3.Plot the predicted vs. actual stock prices to visualize the model's performance.

```
# Make predictions on the test set
predicted_prices = model.predict(X_test)

# Transform predictions back to the original scale
predicted_prices = scaler.inverse_transform(predicted_prices)
y_test_rescaled = scaler.inverse_transform(y_test)

# Plot the predicted vs. actual stock prices
plt.figure(figsize=(14,5))
plt.plot(y_test_rescaled, color="blue", label="Actual Stock Price")
plt.plot(predicted_prices, color="red", label="Predicted Stock Price")
plt.title("Predicted vs Actual Stock Prices")
plt.xlabel("Time")
plt.ylabel("Stock Price")
plt.legend()
plt.show()

14/14 ━━━━━━━━━━━━━━━━━━━━ 1s 37ms/step
```



Predicted vs Actual Stock Prices Plot
Trend Matching: The model captures the general trend and fluctuations in stock prices, following the overall pattern.

Lag in Prediction: Some lag is observed during rapid price changes, where the model falls slightly behind the actual prices.
Error Patterns: Slight underestimations during upward trends and slight overestimations during downward trends, showing a tendency to smooth out fluctuations.

Step 6: Evaluation

1.Calculate the mean absolute error (MAE) and root mean squared error (RMSE) on the test set.
2.Discuss how well the model performed based on these metrics.

```
from sklearn.metrics import mean_absolute_error, mean_squared_error
import math

# Calculate MAE and RMSE
mae = mean_absolute_error(y_test_rescaled, predicted_prices)
rmse = math.sqrt(mean_squared_error(y_test_rescaled,
predicted_prices))

# Print evaluation metrics
print(f"Mean Absolute Error (MAE): {mae}")
print(f"Root Mean Squared Error (RMSE): {rmse}")
```

```
Mean Absolute Error (MAE): 1.4752652353510127
Root Mean Squared Error (RMSE): 1.8155593013127427
```

Interpretation of Model Results
1. Error Metrics
Mean Absolute Error (MAE): 1.48 — The average prediction error is approximately $1.48, indicating good accuracy.
Root Mean Squared Error (RMSE): 1.82 — Slightly higher than MAE, showing that some predictions deviate more significantly, but overall errors are moderate.

2. Implications for Use
Short-Term Reliability: Good for short-term trend predictions due to low error metrics.
Limitations in Quick Reactions: May struggle with sudden price shifts, as indicated by occasional lag.
Potential for Improvement: Adding more features or increasing model complexity (e.g., more LSTM layers, using GRU cells) could improve accuracy.

Brief Report

The RNN model was trained to predict stock prices based on past data using a 60-day sequence window. After training for 50 epochs, the model's predictions on the test set were evaluated using Mean Absolute Error (MAE) and Root Mean Squared Error (RMSE) metrics. The MAE and RMSE were indicating that the model was able to approximate stock price movements but showed some deviation from actual prices.

The predicted vs. actual plot shows that the model captures the general trend of the stock price but has limitations in predicting sudden changes. Potential improvements could involve experimenting with deeper network architectures, using LSTM or GRU layers, or adding more features related to stock market indicators.