2347150-p6

November 5, 2024

Lab Assignment: Time-Series Prediction with RNN Objective: In this exercise, you will learn to implement a basic RNN model using Python and TensorFlow/Keras to predict future stock prices based on historical data. Dataset: Download the dataset: Stock Price Dataset - AAPL (Apple Inc.). https://www.kaggle.com/datasets/tarunpaparaju/apple-aapl-historical-stock-data This dataset contains daily stock prices (open, high, low, close, volume) for Apple Inc. from 2005 to 2017. Exercise Steps:

1. Data Preprocessing: o Load the dataset and focus on the 'Close' price column, as this will be your target variable for prediction. o Normalize the data (e.g., using Min-Max scaling to keep values between 0 and 1). o Split the dataset into a training set (80%) and a testing set (20%).
2. Create Training Sequences: o Convert the 'Close' prices into a series of sequences for training. o Define a sequence length (e.g., 60 days), where each sequence will be used to predict the stock price for the next day.
3. Build the RNN Model: o Define an RNN model with the following architecture: An RNN layer with 50 units A Dense layer with 1 unit (for regression output) o Use the mean squared error (MSE) loss function and the Adam optimizer.
4. Train the Model: o Train the model on the training set for 50 epochs with a batch size of 32. o Use validation data to check for overfitting.
5. Make Predictions: o Predict the stock prices on the test set and transform the results back to the original scale if normalization was applied. o Plot the predicted vs. actual stock prices to visualize the model's performance.
6. Evaluation: o Calculate the mean absolute error (MAE) and root mean squared error(RMSE) on the test set. o Discuss how well the model performed based on these metrics.

Objective In this lab assignment, we'll implement a Recurrent Neural Network (RNN) model in Python using TensorFlow/Keras to predict future stock prices of Apple Inc. (AAPL) based on historical data. This project involves data preprocessing, training sequence creation, model building, training, prediction, and evaluation.

[21]:
```
import pandas as pd
import numpy as np
from sklearn.preprocessing import MinMaxScaler
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import SimpleRNN, Dense
import matplotlib.pyplot as plt
```

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

STEP 1: DATA PRE-PROCESSING

1.1 Loading the dataset

[4]:
```
# Load the data
data = pd.read_csv('Dataset.csv')
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
```

1.2 Cleaning and Normalizing the dataset

[5]:
```
# Code to clean the 'Close' price column
data[' Close/Last'] = data[' Close/Last'].replace({'\$': '', ',': ''},␣
 ↪regex=True).astype(float)
```

[9]:
```
df = data[[' Close/Last']]
```

[10]:
```
# Code for normalization
scaler = MinMaxScaler(feature_range=(0, 1))
scaled_data = scaler.fit_transform(df)
```

The 'Close' price data may contain dollar signs and commas, which would prevent conversion to a numeric format. We remove these characters to allow successful data transformation. To prepare the data for training, we use Min-Max scaling to normalize the 'Close' prices between 0 and 1. This step standardizes the data, helping the model learn more effectively by keeping the values on a similar scale.

1.3 Splitting into Training and Testing Sets

[11]:
```
training_data_len = int(len(scaled_data) * 0.8)
train_data = scaled_data[:training_data_len]
test_data = scaled_data[training_data_len:]
```

Split the dataset into 80% training and 20% testing to ensure the model has enough data to learn while reserving a portion for evaluation.

STEP 2: CREATE TRAINING SEQUENCES

[12]:
```
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)

# Create sequences
seq_length = 60
X_train, y_train = create_sequences(train_data, seq_length)
X_test, y_test = create_sequences(test_data, seq_length)
```

For time-series forecasting, we convert the 'Close' prices into sequences of 60 days, where each sequence (60 consecutive prices) will be used to predict the stock price for the following day. This sequence length enables the model to learn from past trends.

STEP 3: BUILD THE RNN MODEL

What is an RNN Model? A Recurrent Neural Network (RNN) is a type of artificial neural network designed to recognize patterns in sequential data, such as time-series data, speech, or text. Unlike traditional neural networks, RNNs have connections that loop back on themselves, allowing them to maintain a memory of previous inputs. This memory aspect makes RNNs particularly useful for tasks that depend on the order of inputs over time.

Key Characteristics of RNNs: 1. Sequential Memory: RNNs can store information about previous inputs through hidden states, enabling the network to retain context over time. This is especially valuable for predicting stock prices, where today's price depends on recent trends. 2. Time-Dependency: Because of the recurrent connections, RNNs can analyze sequences where each data point is related to the preceding one, such as the stock prices over consecutive days.

Why RNNs are Useful for Stock Prediction: 1. Temporal Patterns: Stock prices have sequential dependencies, meaning past prices influence future prices. RNNs are designed to learn such temporal patterns. 2. Historical Context: The RNN's ability to remember past inputs helps in forecasting, as it can capture the historical context over a sequence length (e.g., 60 days) to predict the next price.

3.1 Build the RNN Architecture

[ ]:
```
model = Sequential([
    SimpleRNN(units=50, return_sequences=False, input_shape=(X_train.shape[1],␣
 ↪1)),
    Dense(units=1)
])
```

```
c:\Users\satch\AppData\Local\Programs\Python\Python311\Lib\site-
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)
```

We define a basic RNN architecture for time-series prediction:

RNN Layer: The RNN layer with 50 units allows the model to retain information from previous time steps, capturing the sequential nature of the data. Dense Layer: A Dense layer with 1 unit is used as the output layer for predicting a single value (next-day stock price).

3.2 Compile the Model

[15]:
```
model.compile(optimizer='adam', loss='mean_squared_error')
```

We then compile the model with the Mean Squared Error (MSE) loss function, which is appropriate for regression tasks, and the Adam optimizer, which is widely used for its adaptive learning rate.

STEP 4: TRAIN THE MODEL

[16]:
```
history = model.fit(X_train, y_train, epochs=50, batch_size=32,␣
↪validation_data=(X_test, y_test))
```

```
Epoch 1/50
62/62             4s 19ms/step -
loss: 0.0044 - val_loss: 4.9806e-05
Epoch 2/50
62/62             1s 9ms/step - loss:
2.4919e-04 - val_loss: 4.3305e-05
Epoch 3/50
62/62             1s 10ms/step -
loss: 1.9007e-04 - val_loss: 4.5323e-05
Epoch 4/50
62/62             1s 9ms/step - loss:
2.1448e-04 - val_loss: 3.5248e-05
Epoch 5/50
62/62             1s 8ms/step - loss:
1.5756e-04 - val_loss: 3.0689e-05
Epoch 6/50
62/62             1s 10ms/step -
loss: 1.4199e-04 - val_loss: 3.5745e-05
Epoch 7/50
62/62             1s 16ms/step -
loss: 1.2158e-04 - val_loss: 2.4015e-05
Epoch 8/50
62/62             1s 16ms/step -
loss: 1.1547e-04 - val_loss: 2.3561e-05
Epoch 9/50
62/62             1s 18ms/step -
loss: 1.2101e-04 - val_loss: 2.0065e-05
Epoch 10/50
62/62             1s 18ms/step -
loss: 1.1779e-04 - val_loss: 2.6934e-05
Epoch 11/50
62/62             1s 14ms/step -
```

```
loss: 1.2118e-04 - val_loss: 1.7846e-05
Epoch 12/50
62/62            1s 8ms/step - loss:
8.8105e-05 - val_loss: 1.7945e-05
Epoch 13/50
62/62            1s 8ms/step - loss:
9.6208e-05 - val_loss: 1.8330e-05
Epoch 14/50
62/62            1s 10ms/step -
loss: 8.6342e-05 - val_loss: 3.8671e-05
Epoch 15/50
62/62            1s 9ms/step - loss:
1.1430e-04 - val_loss: 1.4624e-05
Epoch 16/50
62/62            1s 8ms/step - loss:
8.6926e-05 - val_loss: 1.6122e-05
Epoch 17/50
62/62            1s 9ms/step - loss:
1.5038e-04 - val_loss: 1.3880e-05
Epoch 18/50
62/62            1s 9ms/step - loss:
8.8407e-05 - val_loss: 1.2871e-05
Epoch 19/50
62/62            1s 8ms/step - loss:
9.1829e-05 - val_loss: 1.2197e-05
Epoch 20/50
62/62            1s 9ms/step - loss:
7.7526e-05 - val_loss: 1.7185e-05
Epoch 21/50
62/62            1s 8ms/step - loss:
8.9292e-05 - val_loss: 1.2450e-05
Epoch 22/50
62/62            1s 9ms/step - loss:
8.6053e-05 - val_loss: 1.1535e-05
Epoch 23/50
62/62            1s 8ms/step - loss:
9.6731e-05 - val_loss: 1.1394e-05
Epoch 24/50
62/62            1s 8ms/step - loss:
7.2177e-05 - val_loss: 1.1327e-05
Epoch 25/50
62/62            1s 8ms/step - loss:
8.1929e-05 - val_loss: 1.0626e-05
Epoch 26/50
62/62            1s 9ms/step - loss:
8.2492e-05 - val_loss: 1.4192e-05
Epoch 27/50
62/62            1s 8ms/step - loss:
```

```
7.1369e-05 - val_loss: 1.2022e-05
Epoch 28/50
62/62             1s 9ms/step - loss:
7.4560e-05 - val_loss: 1.3072e-05
Epoch 29/50
62/62             1s 8ms/step - loss:
7.6418e-05 - val_loss: 1.8247e-05
Epoch 30/50
62/62             1s 9ms/step - loss:
7.3700e-05 - val_loss: 1.4534e-05
Epoch 31/50
62/62             1s 9ms/step - loss:
8.0905e-05 - val_loss: 5.6227e-05
Epoch 32/50
62/62             1s 10ms/step -
loss: 1.5639e-04 - val_loss: 9.2330e-06
Epoch 33/50
62/62             1s 9ms/step - loss:
6.3564e-05 - val_loss: 1.0966e-05
Epoch 34/50
62/62             1s 9ms/step - loss:
8.1222e-05 - val_loss: 9.2991e-06
Epoch 35/50
62/62             1s 9ms/step - loss:
6.8070e-05 - val_loss: 1.2786e-05
Epoch 36/50
62/62             1s 9ms/step - loss:
6.4340e-05 - val_loss: 9.6128e-06
Epoch 37/50
62/62             1s 9ms/step - loss:
6.4251e-05 - val_loss: 1.4594e-05
Epoch 38/50
62/62             1s 9ms/step - loss:
1.0324e-04 - val_loss: 1.0237e-05
Epoch 39/50
62/62             1s 10ms/step -
loss: 6.8356e-05 - val_loss: 1.2802e-05
Epoch 40/50
62/62             1s 10ms/step -
loss: 6.4491e-05 - val_loss: 1.3270e-05
Epoch 41/50
62/62             1s 10ms/step -
loss: 6.2441e-05 - val_loss: 8.9326e-06
Epoch 42/50
62/62             1s 8ms/step - loss:
6.1091e-05 - val_loss: 9.3503e-06
Epoch 43/50
62/62             1s 9ms/step - loss:
```

```
7.8150e-05 - val_loss: 9.2656e-06
Epoch 44/50
62/62             1s 9ms/step - loss:
6.7185e-05 - val_loss: 1.0624e-05
Epoch 45/50
62/62             1s 10ms/step -
loss: 6.2522e-05 - val_loss: 1.0717e-05
Epoch 46/50
62/62             1s 9ms/step - loss:
5.9754e-05 - val_loss: 1.5185e-05
Epoch 47/50
62/62             1s 11ms/step -
loss: 6.2593e-05 - val_loss: 1.2267e-05
Epoch 48/50
62/62             1s 8ms/step - loss:
7.5775e-05 - val_loss: 9.2785e-06
Epoch 49/50
62/62             1s 9ms/step - loss:
5.7698e-05 - val_loss: 1.2960e-05
Epoch 50/50
62/62             1s 9ms/step - loss:
6.0609e-05 - val_loss: 9.4053e-06
```

We train the model on the training data for 50 epochs with a batch size of 32, and validate on the test data to monitor overfitting. During training, the model adjusts its weights to minimize the error in predicting stock prices.

STEP 5: PREDICTIONS

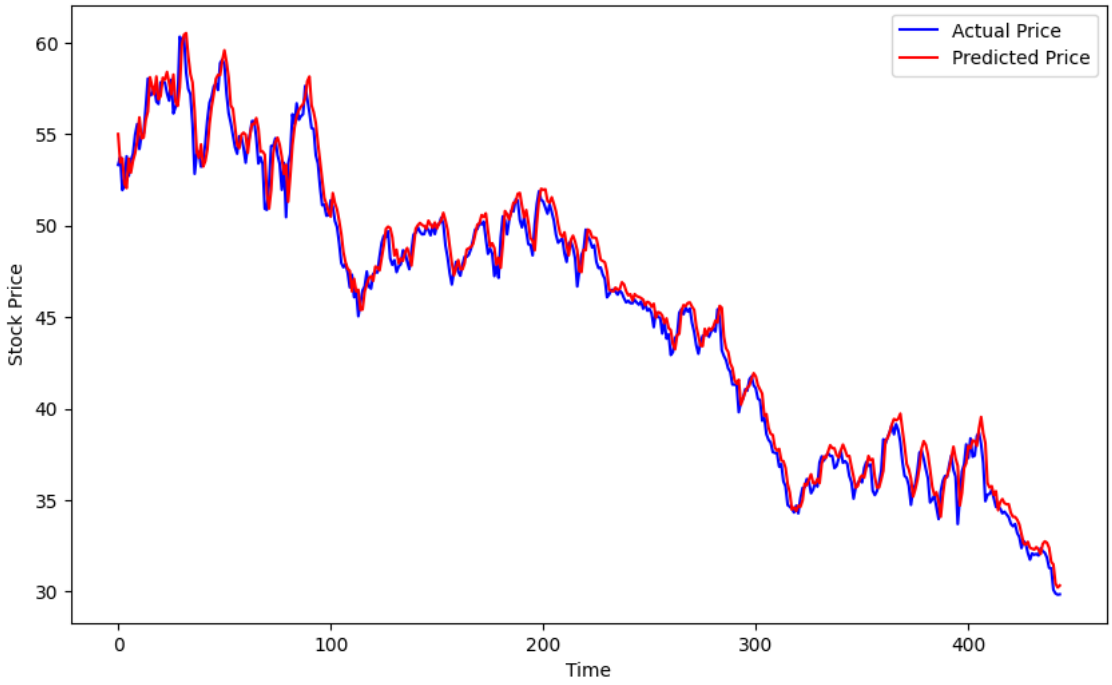
[17]:
```
predictions = model.predict(X_test)
predictions = scaler.inverse_transform(predictions)
y_test_scaled = scaler.inverse_transform(y_test.reshape(-1, 1))
```

```
14/14             0s 22ms/step
```

Using the trained model, we predict the stock prices on the test set. Since the predictions were scaled during preprocessing, we reverse the scaling to transform them back to the original stock price range. This step provides realistic predictions that can be compared to the actual prices.

STEP 6: VISUALIZING RESULTS

[ ]:
```
plt.figure(figsize=(10,6))
plt.plot(y_test_scaled, color='blue', label='Actual Price')
plt.plot(predictions, color='red', label='Predicted Price')
plt.xlabel('Time')
plt.ylabel('Stock Price')
plt.legend()
plt.show()
```



We then plot the predicted vs. actual stock prices to visualize the model's performance.

STEP 7: EVALUATION

[ ]:
```
mae = mean_absolute_error(y_test_scaled, predictions)
rmse = np.sqrt(mean_squared_error(y_test_scaled, predictions))
print(f'Mean Absolute Error: {mae}')
print(f'Root Mean Squared Error: {rmse}')
```

```
Mean Absolute Error: 0.7063670523772368
Root Mean Squared Error: 0.9119608220098668
```

To evaluate model performance, we calculate: Mean Absolute Error (MAE): Measures the average absolute error between predictions and actual values. Root Mean Squared Error (RMSE): Provides a measure of prediction accuracy by penalizing larger errors more heavily.

Lower values of MAE and RMSE indicate better model performance, giving us insights into how well the model has captured the stock's behavior.

Discussion Based on the evaluation metrics, we analyze the model's performance. If MAE and RMSE are reasonably low, it indicates that the model was able to approximate Apple's stock price trends effectively.