STOCK PRICE PREDICTION -Phase 4

Madras Institute of Technology

2021506045-Mahamudha Begam A

2021506037-Kavisri S

2021506025-Gowri R

2021506061-Pavithra S

INTRODUCTION

In this phase of our stock price prediction project, we advance by focusing on feature engineering, model training, and evaluation. Feature engineering involves crafting and selecting meaningful attributes from our data to enhance predictive accuracy. Subsequently, we'll train our model, employing suitable machine learning algorithms and parameter optimization. Evaluation then allows us to gauge our model's performance, refining it for improved stock price forecasts. This progression is essential to empower us with informed investment decisions.

INITIALIZING AN RNN MODEL

In this code snippet, we are initializing a Recurrent Neural Network (RNN) model using TensorFlow. An RNN is a type of neural network commonly used for sequence data, making it particularly useful for tasks like time series prediction, natural language processing, and more. We create an RNN model using the `tf.keras.models.Sequential()` function, which allows us to define a sequence of neural network layers. In the subsequent layers, we can add various RNN cells, such as LSTM (Long Short-Term Memory) or GRU (Gated Recurrent Unit), to capture temporal dependencies in the data. This is the foundation for building powerful predictive models for time series and sequential data.

```
[ ]  #define an object (initializing RNN)
     import tensorflow as tf
     model = tf.keras.models.Sequential()
```

CONFIGURING THE RNN ARCHITECTURE

In this code segment, we're configuring the architecture of our Recurrent Neural Network (RNN) model. Each model.add() statement represents a layer that we are adding to the RNN sequentially. Here's a breakdown of the architecture:

- We start with an LSTM (Long Short-Term Memory) layer with 60 units, using the ReLU activation function. This layer is designed to return sequences, as indicated by return_sequences=True. The input_shape parameter specifies that we expect input sequences of length 60 with one feature.
- After the first LSTM layer, we add a dropout layer to mitigate overfitting by randomly setting a fraction (20%) of the input units to zero.
- We continue by adding two more LSTM layers with 60 and 80 units, respectively, both using the ReLU activation function and returning sequences. Each is followed by a dropout layer.
- The final LSTM layer consists of 120 units and uses the ReLU activation function. We do not return sequences here, as this is the last LSTM layer in the sequence. Again, we add a dropout layer to prevent overfitting.
- Finally, we add an output layer with a single unit using a dense layer, which will provide our model's predictions.

```
model.add(tf.keras.layers.LSTM(units=60, activation='relu', return_sequences=True, input_shape=(60, 1)))
# dropout layer
model.add(tf.keras.layers.Dropout(0.2))

model.add(tf.keras.layers.LSTM(units=60, activation='relu', return_sequences=True))
# dropout layer
model.add(tf.keras.layers.Dropout(0.2))

model.add(tf.keras.layers.LSTM(units=80, activation='relu', return_sequences=True))
# dropout layer
model.add(tf.keras.layers.Dropout(0.2))

model.add(tf.keras.layers.LSTM(units=120, activation='relu'))
# dropout layer
model.add(tf.keras.layers.Dropout(0.2))
#output layer
model.add(tf.keras.layers.Dense(units=1))
```

MODEL SUMMARY

```
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 lstm (LSTM)                 (None, 60, 60)            14880

 dropout (Dropout)           (None, 60, 60)            0

 lstm_1 (LSTM)               (None, 60, 60)            29040

 dropout_1 (Dropout)         (None, 60, 60)            0

 lstm_2 (LSTM)               (None, 60, 80)            45120

 dropout_2 (Dropout)         (None, 60, 80)            0

 lstm_3 (LSTM)               (None, 120)               96480

 dropout_3 (Dropout)         (None, 120)               0

 dense (Dense)               (None, 1)                 121

=================================================================
Total params: 185641 (725.16 KB)
Trainable params: 185641 (725.16 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

COMPILING THE MODEL

In this code snippet, we compile the neural network model using Keras. Compiling the model is a crucial step before training it. The model.compile() function configures several essential settings for the training process. Here's a brief explanation:

- optimizer='adam': We specify the optimizer to be used during training. In this case, we use the popular 'adam' optimizer, which is known for its efficient gradient descent optimization.
- loss='mean_squared_error': The 'loss' parameter defines the loss function that the model will minimize during training. For stock price prediction, 'mean_squared_error' is often suitable, as it measures the mean squared difference between predicted and actual prices.
- metrics=['mean_absolute_error']: We specify a list of metrics that the model should track during training. In this case, we're monitoring the 'mean_absolute_error,' which provides a measure of the average absolute difference between predicted and actual values.

```
#compile the model
model.compile(optimizer='adam', loss='mean_squared_error',metrics=['mean_absolute_error'])
```

TRAINING THE MODEL

In this code snippet, we initiate the training process for our neural network model using the model.fit() function. Here's a breakdown of what each parameter does:

- x_train and y_train: These are the training data, where x_train represents the input features, and y_train represents the corresponding target values (in this case, stock prices). The model will learn to make predictions based on these data.
- batch_size=32: This parameter determines the number of data points used in each iteration of training. A batch size of 32 means that the model will update its weights after processing 32 data points at a time. Batch training can improve efficiency and convergence.
- epochs=50: An epoch is one complete pass through the entire training dataset. By setting **epochs** to 50, we specify that the model will undergo 50 complete iterations over the training data to learn and improve its predictive capabilities.
- The model.fit() function will execute the training process, during which the model's weights will be adjusted to minimize the defined loss function (mean squared error, in this case) over the specified number of epochs. As a result, the model should become better at making predictions based on the training data.

```
model.fit(x_train,y_train, batch_size=32, epochs=50)
```

```
Epoch 1/50
212/212 [==============================] - 44s 209ms/step - loss: 2.7223e-04 - mean_absolute_error: 0.0114
Epoch 2/50
212/212 [==============================] - 41s 191ms/step - loss: 2.4819e-04 - mean_absolute_error: 0.0110
Epoch 3/50
212/212 [==============================] - 41s 191ms/step - loss: 2.2007e-04 - mean_absolute_error: 0.0106
Epoch 4/50
212/212 [==============================] - 41s 193ms/step - loss: 2.1465e-04 - mean_absolute_error: 0.0106
Epoch 5/50
212/212 [==============================] - 41s 195ms/step - loss: 1.8652e-04 - mean_absolute_error: 0.0097
Epoch 6/50
212/212 [==============================] - 40s 190ms/step - loss: 1.6690e-04 - mean_absolute_error: 0.0092
Epoch 7/50
212/212 [==============================] - 40s 189ms/step - loss: 1.6119e-04 - mean_absolute_error: 0.0092
Epoch 8/50
212/212 [==============================] - 40s 189ms/step - loss: 1.5942e-04 - mean_absolute_error: 0.0093
Epoch 9/50
212/212 [==============================] - 40s 189ms/step - loss: 1.4819e-04 - mean_absolute_error: 0.0088
Epoch 10/50
212/212 [==============================] - 40s 188ms/step - loss: 1.3498e-04 - mean_absolute_error: 0.0085
Epoch 11/50
212/212 [==============================] - 40s 190ms/step - loss: 1.4643e-04 - mean_absolute_error: 0.0089
Epoch 12/50
212/212 [==============================] - 41s 191ms/step - loss: 1.5221e-04 - mean_absolute_error: 0.0090
Epoch 13/50
212/212 [==============================] - 41s 192ms/step - loss: 1.3320e-04 - mean_absolute_error: 0.0086
Epoch 14/50
212/212 [==============================] - 42s 195ms/step - loss: 1.2507e-04 - mean_absolute_error: 0.0083
Epoch 15/50
212/212 [==============================] - 41s 192ms/step - loss: 1.1923e-04 - mean_absolute_error: 0.0081
```

```
Epoch 16/50
212/212 [==============================] - 40s 190ms/step - loss: 1.4352e-04 - mean_absolute_error: 0.0088
Epoch 17/50
212/212 [==============================] - 41s 193ms/step - loss: 1.1807e-04 - mean_absolute_error: 0.0081
Epoch 18/50
212/212 [==============================] - 40s 190ms/step - loss: 1.1111e-04 - mean_absolute_error: 0.0079
Epoch 19/50
212/212 [==============================] - 40s 188ms/step - loss: 1.2149e-04 - mean_absolute_error: 0.0082
Epoch 20/50
212/212 [==============================] - 40s 189ms/step - loss: 1.1051e-04 - mean_absolute_error: 0.0078
Epoch 21/50
212/212 [==============================] - 40s 191ms/step - loss: 1.1132e-04 - mean_absolute_error: 0.0079
Epoch 22/50
212/212 [==============================] - 41s 193ms/step - loss: 1.0930e-04 - mean_absolute_error: 0.0078
Epoch 23/50
212/212 [==============================] - 41s 192ms/step - loss: 1.1039e-04 - mean_absolute_error: 0.0079
Epoch 24/50
212/212 [==============================] - 41s 193ms/step - loss: 1.0834e-04 - mean_absolute_error: 0.0078
Epoch 25/50
212/212 [==============================] - 40s 190ms/step - loss: 1.1007e-04 - mean_absolute_error: 0.0078
Epoch 26/50
212/212 [==============================] - 40s 190ms/step - loss: 1.0655e-04 - mean_absolute_error: 0.0077
Epoch 27/50
212/212 [==============================] - 40s 188ms/step - loss: 1.1022e-04 - mean_absolute_error: 0.0079
Epoch 28/50
212/212 [==============================] - 39s 186ms/step - loss: 1.0995e-04 - mean_absolute_error: 0.0078
Epoch 29/50
212/212 [==============================] - 39s 184ms/step - loss: 1.0020e-04 - mean_absolute_error: 0.0076
Epoch 30/50
212/212 [==============================] - 39s 183ms/step - loss: 9.8919e-05 - mean_absolute_error: 0.0074
Epoch 31/50
212/212 [==============================] - 39s 182ms/step - loss: 1.0475e-04 - mean_absolute_error: 0.0076
Epoch 32/50
```

```
Epoch 33/50
212/212 [==============================] - 39s 184ms/step - loss: 1.0392e-04 - mean_absolute_error: 0.0077
Epoch 34/50
212/212 [==============================] - 39s 183ms/step - loss: 9.8563e-05 - mean_absolute_error: 0.0074
Epoch 35/50
212/212 [==============================] - 39s 183ms/step - loss: 9.6425e-05 - mean_absolute_error: 0.0073
Epoch 36/50
212/212 [==============================] - 40s 188ms/step - loss: 9.7854e-05 - mean_absolute_error: 0.0075
Epoch 37/50
212/212 [==============================] - 39s 183ms/step - loss: 9.3506e-05 - mean_absolute_error: 0.0073
Epoch 38/50
212/212 [==============================] - 39s 183ms/step - loss: 9.7332e-05 - mean_absolute_error: 0.0074
Epoch 39/50
212/212 [==============================] - 39s 184ms/step - loss: 9.9501e-05 - mean_absolute_error: 0.0074
Epoch 40/50
212/212 [==============================] - 39s 185ms/step - loss: 1.0370e-04 - mean_absolute_error: 0.0076
```

```
Epoch 41/50
212/212 [==============================] - 39s 184ms/step - loss: 9.2521e-05 - mean_absolute_error: 0.0072
Epoch 42/50
212/212 [==============================] - 39s 183ms/step - loss: 9.2871e-05 - mean_absolute_error: 0.0072
Epoch 43/50
212/212 [==============================] - 39s 185ms/step - loss: 9.2017e-05 - mean_absolute_error: 0.0071
Epoch 44/50
212/212 [==============================] - 39s 183ms/step - loss: 9.1057e-05 - mean_absolute_error: 0.0072
Epoch 45/50
212/212 [==============================] - 39s 183ms/step - loss: 9.1488e-05 - mean_absolute_error: 0.0072
Epoch 46/50
212/212 [==============================] - 39s 182ms/step - loss: 9.3139e-05 - mean_absolute_error: 0.0072
Epoch 47/50
212/212 [==============================] - 39s 186ms/step - loss: 8.9622e-05 - mean_absolute_error: 0.0071
Epoch 48/50
212/212 [==============================] - 39s 183ms/step - loss: 0.0106 - mean_absolute_error: 0.0128
Epoch 49/50
212/212 [==============================] - 39s 183ms/step - loss: 1.3550e-04 - mean_absolute_error: 0.0088
Epoch 50/50
212/212 [==============================] - 39s 183ms/step - loss: 1.2311e-04 - mean_absolute_error: 0.0082
<keras.src.callbacks.History at 0x7953cc63c5b0>
```

MAKING PREDICTIONS AND INVERSE TRANSFORMATIONS

In this code snippet, we are making predictions on the test data using our trained neural network model, and then we are performing an inverse transformation to convert the scaled predictions back to their original, actual values. Here's a brief explanation:

- `predictions = model.predict(x_test)`: This line uses the trained model to make predictions on the test data (`x_test`). The model has learned from the training data and now applies its learned patterns to generate predictions for the test set.
- `predictions = scaler.inverse_transform(predictions)`: After obtaining predictions, we apply an inverse transformation to convert the scaled predictions back to their original scale. It's common to scale data before feeding it into a neural network to improve training efficiency and convergence. The `scaler` object (not shown in the code) is used to reverse this scaling, ensuring that the predictions are in the same units as the original stock prices.

```
# Make predictions on the test data
predictions = model.predict(x_test)

# Inverse transform the scaled predictions to get actual values
predictions = scaler.inverse_transform(predictions)
```

```
212/212 [==============================] - 12s 52ms/step
```

CONCLUSION

In conclusion, we've successfully constructed and trained a recurrent neural network (RNN) model for stock price prediction. Through careful feature engineering and model architecture design, we've harnessed the power of sequential data to make meaningful predictions. By compiling and training the model, we've equipped it with the ability to learn from historical data and forecast future stock prices. The inverse transformation of predictions to their original scale allows for direct comparison with actual stock prices, facilitating performance evaluation. The model's efficacy ultimately depends on dataset quality, feature selection, and further fine-tuning, with the potential to be a valuable tool for informed investment decisions in the stock market.