Deep Learning Predictive Modeling in Finance | RNNs

aiquants Research | © Dayton Nyamai

Application of RNNs in Market Direction Prediction

Summary

This project focuses on the application of Recurrent Neural Networks (RNNs) in the market. It involves the implementation of a Python code that utilizes historical data to predict the direction of price movement for a financial instrument. The code encompasses various stages such as data preprocessing, feature engineering, and visualization.

[41]:
```
# Import the necessary libraries
import pandas as pd
import numpy as np
```

[42]:
```
from pylab import mpl, plt
plt.style.use('seaborn-v0_8')
mpl.rcParams['font.family'] = 'serif'
%matplotlib inline
```

The Data

[43]:
```
# Load the historical data and drop any row with missing values
url = 'https://raw.githubusercontent.com/dayton-nyamai/MarketDLModels/main/data/
 ↪historical_data.csv'
raw = pd.read_csv(url, index_col=0, parse_dates=True).dropna()
raw.info() #the raw data meta information
```

```
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 3535 entries, 2010-01-01 to 2023-07-28
Data columns (total 6 columns):
 #   Column    Non-Null Count  Dtype
---  ------    --------------  -----
 0   EURUSD=X  3535 non-null   float64
 1   GBPUSD=X  3535 non-null   float64
 2   AUDUSD=X  3535 non-null   float64
 3   NZDUSD=X  3535 non-null   float64
 4   JPY=X     3535 non-null   float64
 5   EURJPY=X  3535 non-null   float64
```

```
dtypes: float64(6)
memory usage: 193.3 KB
```

Select the symbol and create a DataFrame

[44]:
```python
symbol = ['EURUSD=X']
data = pd.DataFrame(raw[symbol])
```

Align dates and rename the column containing the price data to 'price'.

[45]:
```python
start_date = data.index.min()
end_date = data.index.max()
data = data.loc[start_date:end_date]
data.rename(columns={'EURUSD=X': 'price'}, inplace=True)
data.round(4).head()
```

[45]:

	price
2010-01-01	1.4390
2010-01-04	1.4424
2010-01-05	1.4366
2010-01-06	1.4404
2010-01-07	1.4318

Calculate log returns and create direction column

[46]:
```python
data['returns'] = np.log(data['price'] / data['price'].shift(1))
data.dropna(inplace=True)
data['direction'] = np.where(data['returns'] > 0, 1, 0)
data.round(4).head()
```

[46]:

	price	returns	direction
2010-01-04	1.4424	0.0024	1
2010-01-05	1.4366	-0.0040	0
2010-01-06	1.4404	0.0026	1
2010-01-07	1.4318	-0.0060	0
2010-01-08	1.4411	0.0065	1

A histogram providing visual representation of the EUR log returns distribution

[47]:
```python
data['returns'].hist(bins=35, figsize=(10, 6));
# Add figure caption
plt.figtext(0.5, -0.01, 'Fig. 1.1 A histogram  showing the distribution of EUR␣
↪log returns ', style='italic',ha='center')

# Show the plot
plt.show()
```

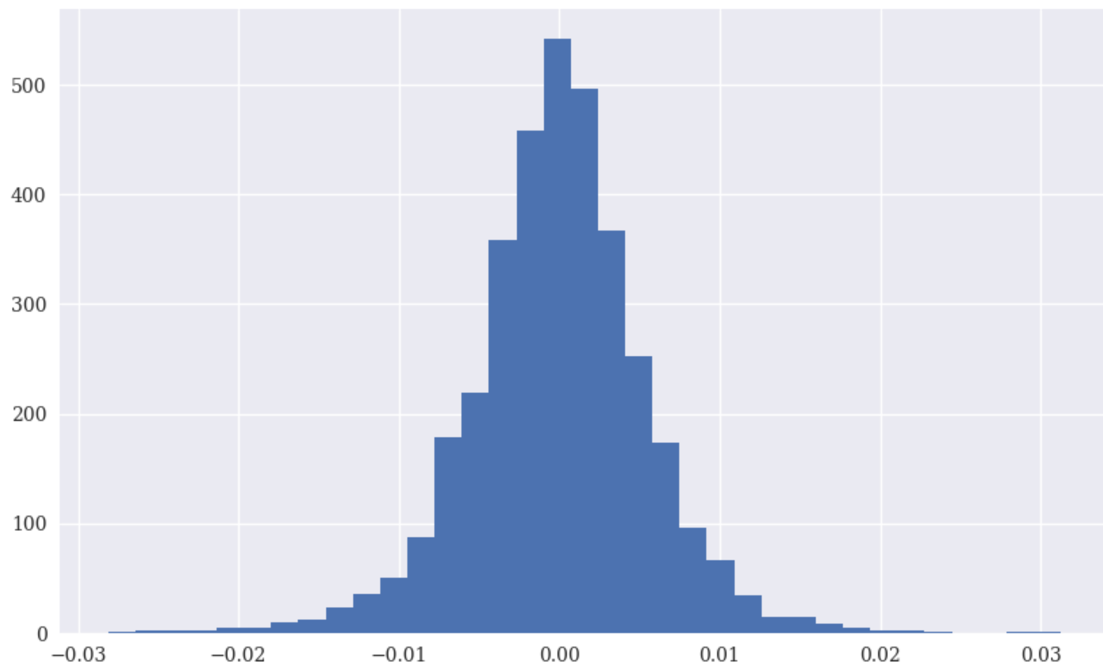

Fig. 1.1 A histogram showing the distribution of EUR log returns

Second, create the features data by lagging the log returns and visualize it in combination with the returns data. We can use various visualization techniques such as scatter plots or line plots to compare the lagged log returns with the returns data.

Create lagged columns

[48]:
```
lags = 5

cols =[ ]
for lag in range(1, lags+1):
    col =  f'lag_{lag}'
    data[col] = data['returns'].shift(lag)
    cols.append(col)
data.dropna(inplace=True)

data.round(4).tail()
```

[48]:

	price	returns	direction	lag_1	lag_2	lag_3	lag_4	lag_5
2023-07-24	1.1125	-0.0011	0	-0.0061	-0.0021	-0.0008	0.0009	0.0004
2023-07-25	1.1063	-0.0056	0	-0.0011	-0.0061	-0.0021	-0.0008	0.0009
2023-07-26	1.1050	-0.0011	0	-0.0056	-0.0011	-0.0061	-0.0021	-0.0008
2023-07-27	1.1078	0.0025	1	-0.0011	-0.0056	-0.0011	-0.0061	-0.0021
2023-07-28	1.0979	-0.0090	0	0.0025	-0.0011	-0.0056	-0.0011	-0.0061

Scatter plot based on features and labels data

[49]:
```
data.plot.scatter(x='lag_1', y='lag_2', c='returns',
                  cmap='coolwarm', figsize=(10, 6), colorbar=True)

# Adding vertical and horizontal lines at 0
plt.axvline(0, c='r', ls='--')
plt.axhline(0, c='r', ls='--')

# Add figure caption
plt.figtext(0.4, -0.03,
            'Fig. 1.2 A scatter plot based on features and labels data',
            style='italic',ha='center')

# Show the plot
plt.show()
```

Fig. 1.2 A scatter plot based on features and labels data

With the dataset fully prepared, various deep learning techniques can be employed to forecast market movements based on the provided features. Additionally, these predictions can be utilized to rigorously backtest a trading strategy.

Deep Learning Models: Recurrent Neural Networks

Recurrent Neural Networks (RNNs) are a type of neural network architecture that is specifically designed to handle sequential data. They are widely used in various applications, including natural language processing, speech recognition, and time series analysis. RNNs are particularly effective for sequence prediction tasks because they can capture the temporal dependencies in the data. There are several types of RNNs that are well-suited for sequence prediction tasks. In our case, we will explore the **Long Short-Term Memory (LSTM)**.

Summary

In this task, we will create an LSTM (Long Short-Term Memory) model for predicting future market movements. We will also utilize the TPU (Tensor Processing Unit) VM cloud infrastructure from Google, for efficient training and inference.

Import the necessary libraries, tensorFlow and its submodules

[50]:
```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, LSTM
from sklearn.preprocessing import StandardScaler
#from sklearn.model_selection import train_test_split
from tensorflow.keras.optimizers import Adam
```

Split the data into training and test sets

[51]:
```
cutoff = '2018-12-31'
training_data = data[data.index < cutoff].copy()
test_data = data[data.index >= cutoff].copy()
```

Standardize the training and test data.

[52]:
```
mu, std = training_data.mean(), training_data.std()
training_data_ = (training_data - mu) / std
test_data_ = (test_data - mu) / std
```

Reshape the training and test data for LSTM input

[53]:
```
X_train = np.array(training_data_[cols])
X_train = np.reshape(X_train, (X_train.shape[0], X_train.shape[1], 1))
y_train = np.array(training_data['direction'])

X_test = np.array(test_data_[cols])
X_test = np.reshape(X_test, (X_test.shape[0], X_test.shape[1], 1))
y_test = np.array(test_data['direction'])
```

Build the LSTM model

[54]:
```
model = Sequential()
model.add(LSTM(128, activation='relu', input_shape=(lags, 1)))
model.add(Dense(1, activation='sigmoid'))
```

Compile the model

[55]:
```
optimizer = Adam(learning_rate=0.0001)
model.compile(optimizer=optimizer, loss='binary_crossentropy',␣
 ↪metrics=['accuracy'])
```

Train the model

[56]:
```
model.fit(X_train, y_train, epochs=100, verbose=False, validation_split=0.2,␣
 ↪shuffle=False)

res = pd.DataFrame(model.history.history)
res[['accuracy', 'val_accuracy']].plot(figsize=(10, 6), style='--');
plt.figtext(0.5, 0.02, 'Fig. 1.3 Accuracy of the LSTM model on training and␣
 ↪validation data per training step', style='italic',ha='center')
plt.show()
```

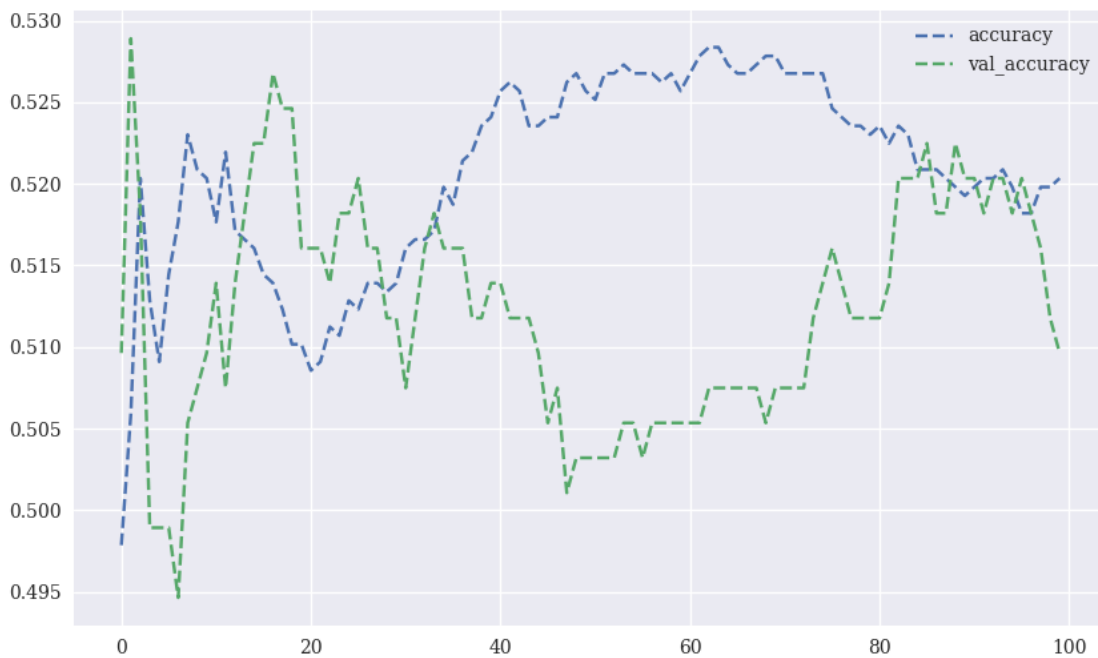

Fig. 1.3 Accuracy of the LSTM model on training and validation data per training step

Evaluate the performance of the model on training data

[57]:
```
train_loss, train_accuracy = model.evaluate(X_train, y_train)
```

```
73/73 [==============================] - 0s 2ms/step - loss: 0.6893 - accuracy:
0.5225
```

Analyzing Accuracy in a Machine Learning Model: The accuracy metrics are essential in evaluating the performance of a machine learning model. Let's break down the metrics and understand their functionalities.

1. **Accuracy / train_accuracy**: The accuracy metric measures the proportion of correctly predicted instances out of the total number of instances. It provides an overall assessment of the model's performance. In our case, the accuracy value is 0.5225, indicating that the model correctly predicted 52.25% of the instances.

2. **Val_Accuracy**: The val_accuracy metric, also known as validation accuracy, measures the accuracy of the model on a separate validation dataset. It helps assess the model's generalization capability. In our case, the val_accuracy value is not explicitly mentioned in the output. However, it is expected to be plotted alongside the accuracy metric in the generated plot.

3. **train_loss**: The loss value calculated during the evaluation of the model on the training dataset. Loss is a measure of how well the model is performing in terms of the discrepancy between the predicted and actual values. In this case, the value of train_loss is 0.6893, suggesting that the model's predictions have a relatively high discrepancy from the actual values in the training dataset. A lower loss value indicates better model performance, so this relatively high loss value implies that the model may not be performing optimally on the training data.

Ideally, we want both accuracy and val_accuracy to be high. A high accuracy indicates that the model has learned the patterns in the training data well, while a high val_accuracy suggests that the model can generalize well to new data.

Insights on Accuracy Metrics

Let's discuss the insights we can gain from analyzing these metrics.

- **Trend**: The plot shows the trend of accuracy and val_accuracy over the epochs. We can analyze whether these metrics are improving, plateauing, or deteriorating over time. A consistent increase in both metrics indicates that the model is learning and improving its performance.

- **Overfitting**: If the accuracy metric keeps improving while the val_accuracy metric starts to plateau or decrease, it suggests that the model is overfitting. Overfitting occurs when the model becomes too specialized in the training data and fails to generalize well on unseen data.

- **Underfitting**: On the other hand, if both accuracy and val_accuracy remain low or do not show significant improvement, it indicates underfitting. Underfitting occurs when the model fails to capture the underlying patterns in the data and performs poorly on both training and validation datasets.

- **Convergence:** If both accuracy and val_accuracy reach a stable value and remain constant over the epochs, it suggests that the model has converged. Convergence indicates that the model has learned the patterns in the data and is not likely to improve further.

Analyzing the accuracy metrics help us understand the performance and behavior of a machine learning model. It allows us to make informed decisions regarding model optimization, such as adjusting hyperparameters, increasing training data, or implementing regularization techniques.

Training Data

Make predictions on the training data

[58]:
```
train_predictions = np.where(model.predict(X_train) > 0.5, 1, 0)
```

```
73/73 [==============================] - 0s 2ms/step
```

Transforms the predictions into long-short positions, +1 and -1

[59]:
```
training_data['prediction'] = np.where(train_predictions > 0, 1, -1)
```

The number of the resulting short and long positions, respectively.

[60]:
```
training_data['prediction'].value_counts()
```

[60]:
```
-1    1498
 1     837
Name: prediction, dtype: int64
```

Trading Rules

In the benchmark case, i.e training_data['returns'], we adopt a long position on the asset throughout the entire period. This means that we hold one unit of the asset for the entire duration. On the other hand, in the case of the DNNs strategy, i.e training_data['strategy'], we take either a long or short position on the asset, i.e one unit of the asset.

Calculates the strategy returns given the positions

[61]:
```
training_data['strategy'] = training_data['prediction'] *␣
  ↪training_data['returns']
training_data[['returns', 'strategy']].sum().apply(np.exp)
```

[61]:
```
returns     0.793215
strategy    1.665913
dtype: float64
```

Plots and compares the strategy performance to the benchmark performance (in-sample)

[62]:
```
training_data[['returns', 'strategy']].cumsum().apply(np.exp).plot(figsize=(10,␣
  ↪6));
plt.figtext(0.5, 0.05, 'Fig. 1.4 Gross performance of EUR/USD compared to the␣
  ↪DNNs-based strategy', style='italic',ha='center')
plt.figtext(0.5, -0.01, '(in-sample, no transaction costs)',␣
  ↪style='italic',ha='center')

plt.show()
```

Fig. 1.4 Gross performance of EUR/USD compared to the DNNs-based strategy

(in-sample, no transaction costs)

The strategy exhibits a modestly superior performance compared to the passive benchmark case on the training data set (in-sample, excluding transaction costs). However, the true measure of its effectiveness lies in its out-of-sample performance on the test data set, which is of utmost interest.

Testing Data

Evaluate the performance of the model on testing data

[63]:
```
model.evaluate(X_test, y_test)
```

```
38/38 [==============================] - 0s 2ms/step - loss: 0.6943 - accuracy:
0.5067
```

[63]:
```
[0.6942607164382935, 0.5067001581192017]
```

Make predictions on the test data

[64]:
```
test_predictions = np.where(model.predict(X_test) > 0.5, 1, 0)

# Transforms the predictions into long-short positions, +1 and -1
test_data['prediction'] = np.where(test_predictions > 0, 1, -1)
```

```
38/38 [==============================] - 0s 2ms/step
```

The number of the resulting short and long positions, respectively.

[65]:
```
test_data['prediction'].value_counts()
```

[65]:
```
-1    858
 1    336
Name: prediction, dtype: int64
```

Calculate the strategy returns given the positions

[66]:
```
test_data['strategy'] = test_data['prediction'] * test_data['returns']
test_data[['returns', 'strategy']].sum().apply(np.exp)
```

[66]:
```
returns     0.960433
strategy    1.046551
dtype: float64
```

Plots and compares the strategy performance to the benchmark performance (out-of-sample)

[67]:
```
test_data[['returns', 'strategy']].cumsum().apply(np.exp).plot(figsize=(10, 6));
plt.figtext(0.5, 0.05, 'Fig. 1.5 Gross performance of EUR/USD compared to the␣
 ↪DNNs-based strategy', style='italic',ha='center')
plt.figtext(0.5, -0.01, '(out-of-sample, no transaction costs)',␣
 ↪style='italic',ha='center')

plt.show()
```

Fig. 1.5 Gross performance of EUR/USD compared to the DNNs-based strategy

(out-of-sample, no transaction costs)

Adding Different Types of Features

To this end, the analysis has primarily focused on the use of log returns. It is, of course, possible to not only incorporate more classes or categories, but also add other types of features to the mix, such as volatility, momentum, distance measures, etc – the sky is the limit. In our next notebook we will graft all these features into the data.