Creating an AI model with Technical Indicator Features to predict future prices (Python Tutorial)

B/O TRADING BLOG
SEP 21, 2022 · PAID

9 2 Share

Photo by [DeepMind on Unsplash.com]

This is an advanced Python tutorial in which we will attempt the 'holy grail' of timeseries forecasting by trying to predict the future price of an asset using the [Tensorflow]framework.

When I first started trading, I tried to use TensorFlow to [predict the stock price by using just the closing price in this post]. The results where a disappointing with a 50/50% success rate that was not better than flipping a coin.

However, as we programmers say 'garbage in, garbage out' meaning what you feed into the model will determine the quality of the model output and the predictive accuracy. So in this post I wanted to use several popular technical indicators such as EMAs, Bollinger Bands and RSI as additional features that are fed into the model as input to see if the forecasting capabilities of the model improved.

If you are curious about the results, this post is for you!

> *This story is solely for general information purposes, and should not be relied upon for trading recommendations or financial advice. Source code and information is provided for educational purposes only, and should not be relied upon to make an investment decision. Please review my full cautionary guidance before continuing.*

What is TensorFlow?

TensorFlow is a one of them most free, open-source frameworks for Machine Learning, Vision-, Audio, and Natural Language Processing (NLP). Next to PyTorch, Tensorflow is one of the most popular AI frameworks for Python.

TensorFlow was developed by the Google Brain Team for internal Google use but later released to the public. It is used by a large variety of companies like Google, airbnb, Coca Cola, Intel for a variety of AI applications.

Here the link to the TensorFlow developer docs.

What are Long-Short Term Memory (LSTM) Nodes?

LSTM stands for 'Long-Short Term Memory' and is a type of node that can be used as part of a TensorFlow Recurring Neural Network (RNN) model. They are specialized nodes used for time series forecasting and NLP and different from conventional node types because they have the ability to 'remember' longer sets of previous data.

To understand the LSTM architecture in more detail, check out this post by javatpoint.com.

In this post we will use LSTM nodes to build our model.

I realize there is a universe of stuff to learn about AI and I can't cover it all in this post but here is a good introductory article about Tensorflow time series prediction. In this post would like to focus on our trading prediction model implementation.

Implementation

You can download the complete Jupyter Notebook from my GDrive repo here.

First you need to install Conda for your platform from here. Below the setup instructions for a setup with a Conda virtual environment using Python 3.10.

Then create a file called 'requirements.txt' and paste the lines below.

```
jupyterlab
numpy
pandas
pandas-ta
ta
matplotlib
tensorflow==2.9.2
keras==2.9.0
sklearn
```

Here the steps to create a Conda environment, install the requirements and start the Jupyter lab.

```
conda create -n tf_forecaster1 python=3.10
conda activate tf_forecaster1
pip install -r requirements.txt
ipython kernel install --user --name=tf_forecaster1
jupyter notebook
```

Next, open the 'tf_forecaster_v2.ipynb' from my repo or create another file called 'tf_forecaster.py' and add the sections below.

Here are the prerequisite Python imports:

```
import os
import locale
locale.setlocale(locale.LC_ALL, 'en_US')
import numpy as np
import pandas as pd
import pandas_ta as pta
from ta.volatility import BollingerBands
import matplotlib.pyplot as plt
from sklearn.preprocessing import MinMaxScaler
import tensorflow as tf
from keras.models import Sequential
from keras.layers import Dense, Dropout, LSTM, Bidirectional
from keras.callbacks import EarlyStopping
```

This function calculates the technical indicators EMA, Bollinger Bands and RSI using the Python packages pandas-ta and ta.

```
def calculate_tis(df):

    # EMAs
    df['ema200'] = pta.ema(close=df['close'], length=200)
    df['ema50'] = pta.ema(close=df['close'], length=50)

    #  RSI
    df['rsi'] = pta.rsi(close=df['close'], window=14)

    #  Bollinger Bands
    indicator_bb = BollingerBands(close=df['close'], window=20,
window_dev=2)
    df['bb_low'] = indicator_bb.bollinger_lband()
    df['bb_high'] = indicator_bb.bollinger_hband()

    #  Reorder columns
    df = df[['open', 'high', 'low', 'close', 'volume', 'ema200',
'ema50', 'rsi', 'bb_low', 'bb_high']]

    #  Drop the first 200 periods which we needed for EMA 200
calculation
    df = df.tail(len(df) - 200)
    df.dropna(inplace=True)

    return df
```

This function splits our data set into training data used to train the model and test data, used to verify the accuracy of the model.

```
def train_test_split(data_array, rows, train_percent):

    display_len_train_records = int(rows * train_percent)
    train_data = data_array[0: display_len_train_records]
    test_data = data_array[display_len_train_records:]

    return train_data, test_data
```

In this function we are scaling our price- and feature data into values from zero to one, which is ensures better results.

For this we are using MinMaxScaler from sklearn. As you can see, I am scaling different sets of the data like train and test data independently to ensure no cross-contamination of the data.

```
def scale_data(train_data, test_data):

    full_scaler = MinMaxScaler(feature_range=(0, 1))
    full_df = np.concatenate((train_data, test_data), axis=0)
    full_scaler.fit(full_df)
    scaled_full_data = full_scaler.transform(full_df)
    final_scaler = MinMaxScaler(feature_range=(0, 1))
    final_scaler.fit(full_df[:,3].reshape(-1, 1))

    train_scaler = MinMaxScaler(feature_range=(0, 1))
    train_scaler.fit(train_data)
    scaled_train_data = train_scaler.transform(train_data)

    test_scaler = MinMaxScaler(feature_range=(0, 1))
    test_scaler.fit(train_data)
    scaled_test_data = test_scaler.transform(test_data)

    return scaled_full_data, scaled_train_data, scaled_test_data,
final_scaler, train_scaler, test_scaler
```

This function builds the TensorFlow model. Here I used 4 layers with a number of nodes that decrease by 50% at each layer until the last layer that outputs the number of forecast steps.

Dropout layers randomly set input units to 0 with a frequency of rate at each step during training time, which helps prevent overfitting.

The model architecture is somewhat subjective and something that needs to be optimized. For this task you may want to experiment with Keras Tuner, which is a utility that automatically varies the hyperparameters of the model.

```
def build_model(time_steps, forecast_steps, n_features):

    input_shape = (time_steps, n_features)

    model = Sequential()
    model.add(LSTM(units=800, return_sequences=True,
input_shape=input_shape))
    model.add(Dropout(0.5))
    model.add(LSTM(units=400, return_sequences=True))
    model.add(Dropout(0.5))
    model.add(LSTM(units=100))
    model.add(Dropout(0.5))
    model.add(Dense(forecast_steps))

    return model
```

In this next block we are training the model. As you can see I'm using EarlyStopping, which halts the training if the mean squared error does not improve after iterations specified by the patience value.

```
def train_model(model, X_train, y_train, X_test, y_test, epochs,
batch_size, patience):

    early_stop = EarlyStopping(monitor='val_loss', patience=patience,
mode='auto')
    model.compile(optimizer='adam', loss='mean_squared_error',
metrics=[tf.keras.metrics.MeanSquaredError(name='MSE')])
    history = model.fit(X_train, y_train, epochs=epochs,
batch_size=batch_size, validation_data=(X_test, y_test), callbacks=
[early_stop])

    return history
```

This next function separates our dependent and independent variables so the features (open/low/volume/technical indicators) from the close price to be predicted.

Here we are also creating batches based on time steps (the steps used for the forecast) and forecast steps (the number steps we want to forecast).

```
def create_X_ys(X_scaled, time_steps, forecast_steps):

    X = []
    y = []
    for i in range(time_steps, X_scaled.shape[0] - forecast_steps +
1):
        X.append(X_scaled[i - time_steps:i])
        y.append(X_scaled[i:i + forecast_steps,3])

    X = np.array(X)
    y = np.array(y)

    return X, y

def create_X(X_scaled, time_steps):

    X = []
    for i in range(time_steps, X_scaled.shape[0] + 1):
        X.append(X_scaled[i - time_steps:i])

    X = np.array(X)

    return X
```

This function creates model predictions using the predict() function of the model.

```
def create_predictions(model, X_full, batch_size):
    full_pred_y = model.predict(X_full, batch_size=batch_size)

    return full_pred_y
```

Here we are configuring the model.

- train_percent is the percentage used to split the data into training and test data.
- epochs are the number of iterations used in training
- batch_size is the size of the batches of time step data used for training
- Time steps are the number of time intervals of data used for prediction
- Forecast steps are the number of time intervals we want to predict into the future.
- Patience is the number of training intervals that is used for early stopping if the value to be optimized does not improve.

```
#  Configuration
file_name = 'ETH-USD_1m_ohlc.csv'
interval = '1min'
train_percent = 0.8
epochs = 300
batch_size = 128
time_steps = 20
forecast_steps = 4
patience = 10
```

In the next block we actually perform the tasks of:

- Loading the input data
- Calculating the technical indicators
- Splitting the data into train and test data
- Scaling the data
- Separating the features from the price data
- Building, training and saving the model.

You may want to put the price data in the same column order that I used: ,date,low,high,open,close,volume.

```
#  Load data file
if os.path.exists(file_name):
    df = pd.read_csv(file_name)
    print("Data File Loaded...")
else:
    print("Load Data File Failed...")

#  Calculate technical indicators
df = calculate_tis(df)

#  Get dimensions
rows = df.shape[0]
columns = df.shape[1]

#  Convert df to ndarray
data_array = df.values

#  Split train & test data
train_data, test_data = train_test_split(data_array, rows,
train_percent)

#  Scale data
scaled_full_data, scaled_train_data, scaled_test_data, full_scaler,
train_scaler, test_scaler = scale_data(train_data, test_data)
n_features = columns

#  Create X_train, y_train, X_test, y_test
X_train, y_train = create_X_ys(scaled_train_data, time_steps,
forecast_steps)
X_test, y_test = create_X_ys(scaled_test_data, time_steps,
forecast_steps)
X_full = create_X(scaled_full_data, time_steps)

#  Build model
model = build_model(time_steps, forecast_steps, n_features)
print(model.summary())

#  Train model
print("Training Model...")
train_model(model, X_train, y_train, X_test, y_test, epochs,
batch_size, patience)
```

```
#  Save Model
model.save("tf_forecaster.h5")
```

In the next section we are creating predictions on the full set of data, reverse scaling the price data and plotting it using the matplotlib package.

```
#  Create prediction
print("Evaluating Model...")
full_pred_y = create_predictions(model, X_full, batch_size)

full_real_y_norm = np.concatenate((train_data, test_data), axis=0)
full_real_y_norm = full_real_y_norm[:,3]

train_len = len(train_data)
test_len = len(test_data)
real_len = len(full_real_y_norm)
pred_len = len(full_pred_y)

#  Plot train/test data and prediction result
plt.figure(figsize=(20, 10))
plt.axvline(x=int(real_len*train_percent), color='b', linewidth=1)
plt.plot(full_real_y_norm, color='blue', label='Actual', linewidth=1)

full_pred_y_norm =
full_scaler.inverse_transform(full_pred_y[:,0].reshape(-1, 1))
full_pred_y_norm = pd.DataFrame(full_pred_y_norm)
full_pred_y_norm.index += time_steps - 1

plt.plot(range(time_steps, train_len+1), full_pred_y_norm[:train_len-
time_steps+1], color='green', label='Train', linewidth=1)
plt.plot(range(train_len,len(full_real_y_norm)),
full_pred_y_norm[train_len-time_steps:-1], color='red',
label='Valid', linewidth=1)
plt.xlabel('Time')
plt.ylabel('Price')
plt.legend()
plt.title('Prediction')

plt.show()
```

The last part is creating the forecast into the future. It is using the last values of the full_pred_y which we already created in the last section. Those values are also reverse scaled for plotting.

Then we are plotting the last 400 time intervals of the training data and the future predictions.

```
#  Create forecast
display_len = 400
x_range = range(train_len-display_len, train_len)
x_pred_range = range(train_len-display_len-time_steps, train_len-
time_steps)

future_df = []
for i in range(forecast_steps):
    full_pred_y_norm =
full_scaler.inverse_transform(full_pred_y[:,i].reshape(-1, 1))
    future_df.append(full_pred_y_norm[train_len-time_steps-1])

#  Plot train/test data and prediction result
plt.figure(figsize=(20, 10))
plt.plot(x_range, full_real_y_norm[x_range], color='blue',
label='Actual', linewidth=1)
full_pred_y_norm =
full_scaler.inverse_transform(full_pred_y[:,0].reshape(-1, 1))
plt.plot(x_range, full_pred_y_norm[x_pred_range], color='green',
label='Train', linewidth=1)
plt.plot(range(len(train_data)-1,len(train_data)+3), future_df,
color='red', label='Prediction', linewidth=1)
plt.xlabel('Time')
plt.ylabel('Price')
plt.legend()
plt.title('Prediction')
plt.show()
```

Results

As input for this model I used 1 year of 1-minute data of Ethereum. However, you can use any instrument data you like.

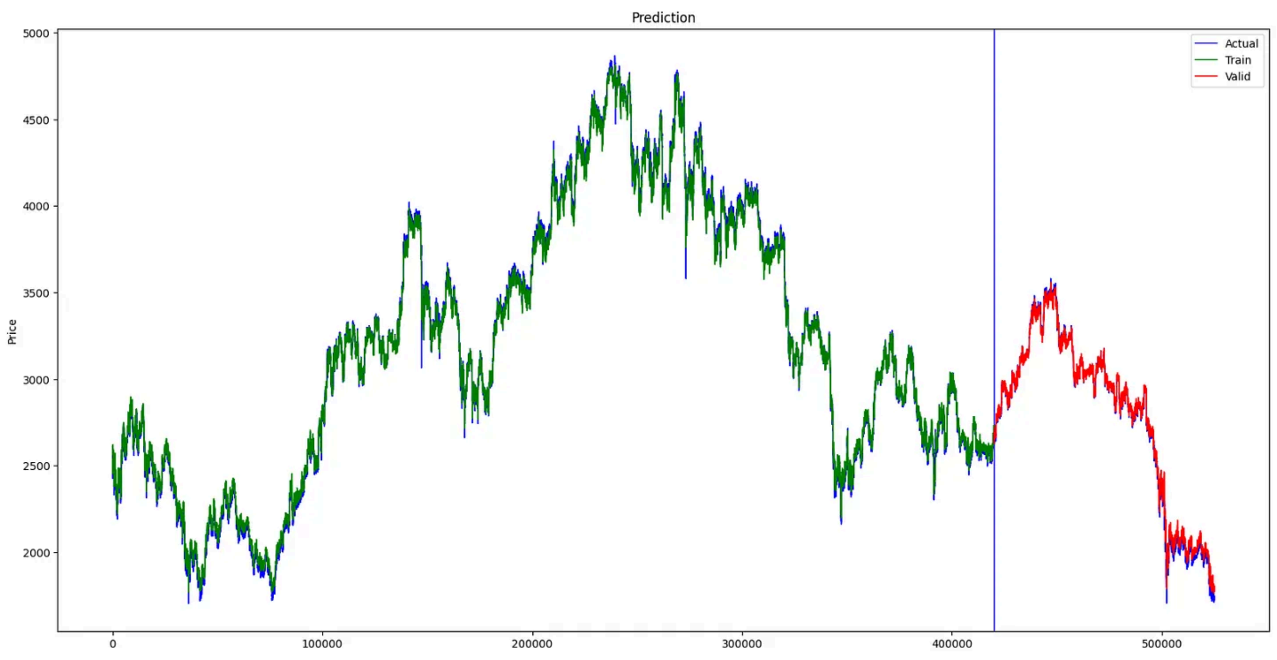
Here the training stats:

- loss: 6.4343e-04
- MSE: 6.4343e-04

- val_loss: 3.3061e-05
- val_MSE: 3.3061e-05

The MSE means the squared error between the training data and the predictions was 0.00064343 which is a pretty good result. The validation MSE is 0.000033.

Below a chart of the training results. The blue line is the actual close price. The green line is the price the model predicted during training. The blue vertical line is the time horizon between training and test data. The red line are the price values the model predicted on the test data. Not too bad.

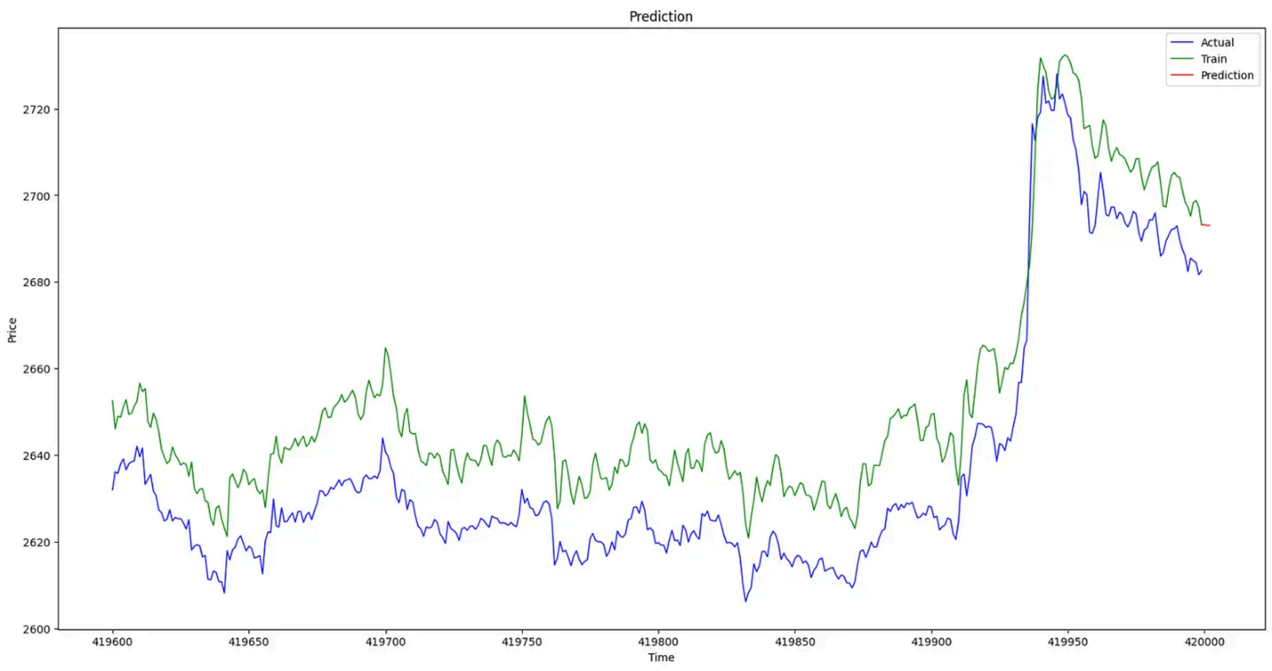


Here is where things get interesting: The chart shows the last 400 minutes of test data. The blue line are the actual price values, the green line are values of the test data the model predicted.

As you can see there are differences so there is some room for optimizing the model, which I haven’t spent much time on.

The little red flag at the end of the chart is the result of all our efforts: the 4 minutes of price data we predicted into the future. That information could eventually be used as a trading signal.

So in this case the model predicted a slight downward trend. Let's compare this against the actual price data to see if the model was accurate.

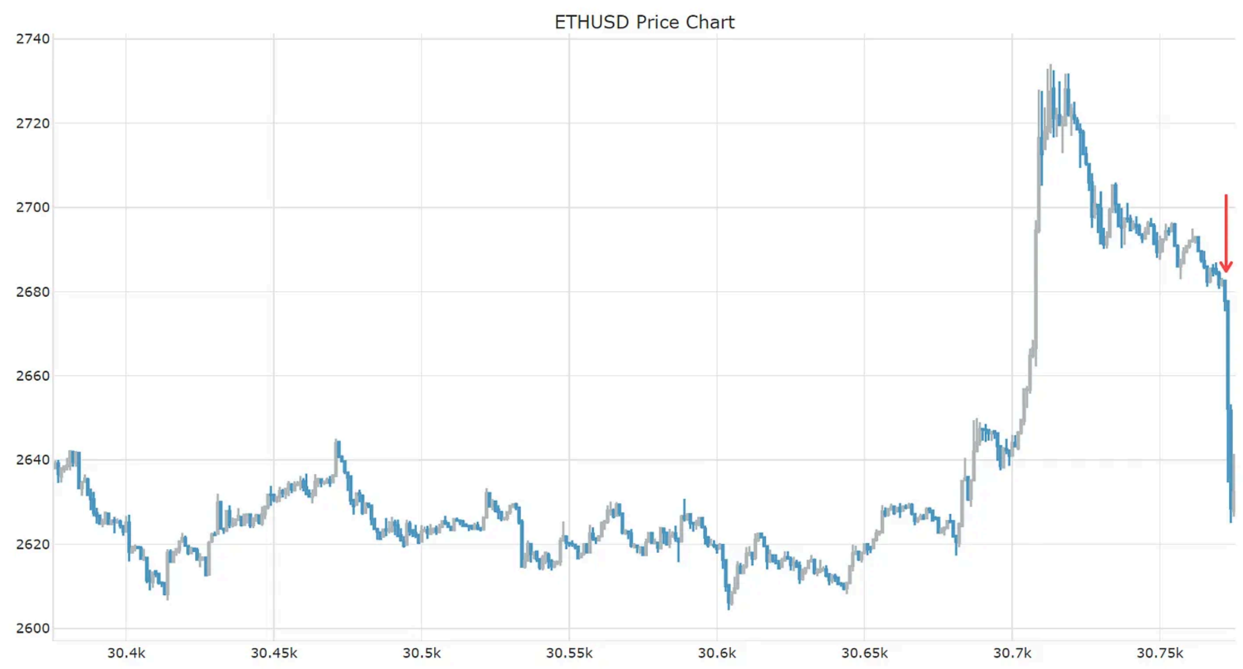


Below the chart of the price data with a small time period after the test data and the forecast occurred.

The model actually predicted the right direction - so great. As you can see, next there was a huge price drop in the minutes that followed. Wow - what the…?

Would the AI have predicted the huge drop? We don't know because we only looked 4 minutes into the future. To find out, we would have to either extend the forecast period or keep running more series of predictions into the future.

But that's a subject for another post. :)



Wrapping Up

In this post we went over the steps of how to use a set of price data and technical indicators as input for a TensorFlow model to try to predict the future price of an asset.

I hope this post was worth your time. Please 'like' it.

Have a great day!

9 Likes

2 Comments

Write a comment...

B/O Trading Blog Nov 5, 2022 Author

Hello Julian,

Thanks for your comment.

I'm also new to AI time series prediction so it's possible there is an issue. My understanding is that we start off with a time range of OHLC data for which the technical indicators are calculated. For training the model, we split the data into training and test data. Then we use the training data to train and we can use the test data set to validate. The way the model is designed is that we use 20 time periods (minutes) of features (price + tech indicators) to predict 4 minutes of future close price changes. When it comes to future predictions, we feed training + test data into the model to generate predictions and get the next 4 minutes of close price predictions. Hopefully this makes sense.

LIKE REPLY SHARE

Julian Nov 5, 2022

I am not sure I understand how this model can be put into practice.

The technical indicators are defined and constructed from the closing price and used in X_test to predict the closing price.

But in practice we cannot know the values of these indicators without the closing price that we are trying to predict.

Could you please provide some clarification?

Thank you very much.

LIKE REPLY SHARE

© 2024 Chris Shield · Privacy · Terms · Collection notice
Substack is the home for great culture