03_multivariate_timeseries

September 29, 2021

1 Multivariate Time Series Regression

So far, we have limited our modeling efforts to single time series. RNNs are naturally well suited to multivariate time series and represent a non-linear alternative to the Vector Autoregressive (VAR) models we covered in Chapter 8, Time Series Models.

1.1 Run inside docker container for GPU acceleration

See tensorflow guide and more detailed instructions

```
docker run -it -p 8889:8888 -v /path/to/machine-learning-for-trading/18_recurrent_neural_nets:/
--name tensorflow tensorflow/tensorflow:latest-gpu-py3 bash
```

Inside docker container: `jupyter notebook --ip 0.0.0.0 --no-browser --allow-root`

1.2 Imports & Settings

[59]:
```
%matplotlib inline
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import pandas_datareader.data as web
from datetime import datetime, date
from sklearn.metrics import mean_squared_error, mean_absolute_error
from sklearn.preprocessing import minmax_scale
from keras.callbacks import ModelCheckpoint, EarlyStopping
from keras.models import Sequential, Model
from keras.layers import Dense, LSTM, Input, concatenate, Embedding, Reshape
import keras
import keras.backend as K
import tensorflow as tf
```

[2]:
```
sns.set_style('whitegrid')
np.random.seed(42)
K.clear_session()
```

1.3 Load Data

For comparison, we illustrate the application of RNNs to modeling and forecasting several time series using the same dataset we used for the VAR example, monthly data on consumer sentiment, and industrial production from the Federal Reserve's FRED service in Chapter 8, Time Series Models:

[3]:
```
df = web.DataReader(['UMCSENT', 'IPGMFN'], 'fred', '1980', '2017-12').dropna()
df.columns = ['sentiment', 'ip']
df.info()
```

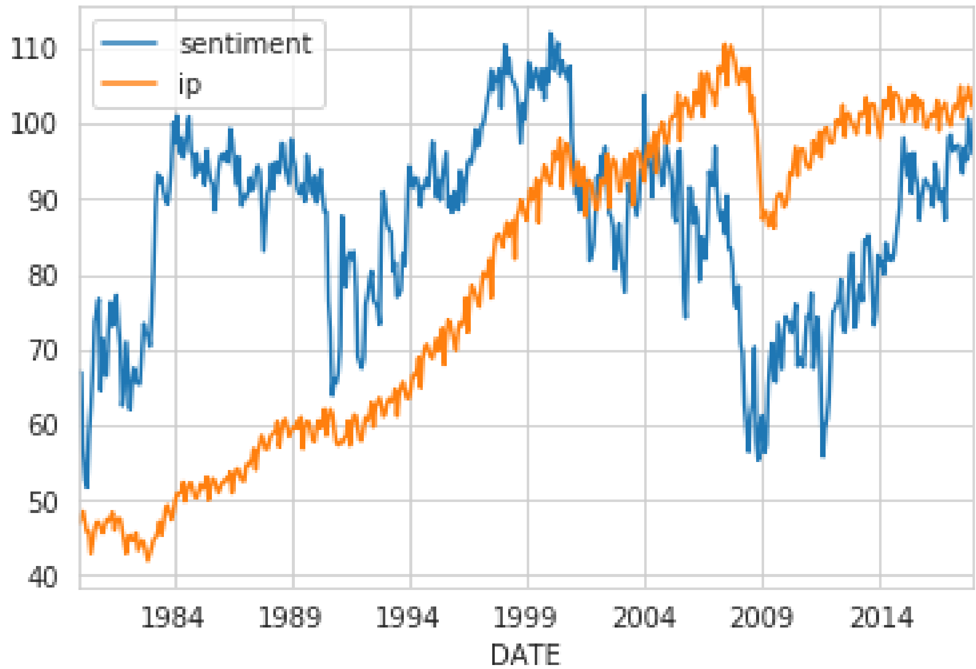
```
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 456 entries, 1980-01-01 to 2017-12-01
Data columns (total 2 columns):
sentiment    456 non-null float64
ip           456 non-null float64
dtypes: float64(2)
memory usage: 10.7 KB
```

[4]:
```
df['1980':].head()
```

[4]:

	sentiment	ip
DATE		
1980-01-01	67.0	46.8853
1980-02-01	66.9	47.9806
1980-03-01	56.5	48.4758
1980-04-01	52.7	47.0631
1980-05-01	51.7	45.6939

[5]:
```
df.plot();
```



1.4 Prepare Data

1.4.1 Scaling

First we scale the data to the [0,1] interval:

[6]:
```
df_scaled = df.apply(minmax_scale)
```

1.4.2 Stationarity

We apply the same transformation—annual difference for both series, prior log-transform for industrial production—to achieve stationarity that we used in Chapter 8 on Time Series Models:

[35]:
```
df_transformed = pd.DataFrame({'ip': np.log(df.ip).diff(12),
                               'sentiment': df.sentiment.diff(12)}).dropna()
```

We can reshape directly to get non-overlapping series, i.e., one sample for each year (works only if the number of samples is divisible by window size):

[7]:
```
df_scaled.values.reshape(-1, 12, 2).shape
```

[7]:
```
(38, 12, 2)
```

1.4.3 Reshape data into RNN format

The create_multivariate_rnn_data function transforms a dataset of several time series into the shape required by the Keras RNN layers, namely `n_samples` x `window_size` x `n_series`, as follows:

[9]:
```
def create_multivariate_rnn_data(data, window_size):
    y = data[window_size:]
    n = data.shape[0]
    X = np.stack([data[i: j] for i, j in enumerate(range(window_size, n))],␣
 ↪axis=0)
    return X, y
```

We will use window_size of 24 months and obtain the desired inputs for our RNN model, as follows:

[8]:
```
window_size = 24
```

[36]:
```
X, y = create_multivariate_rnn_data(df_transformed, window_size=window_size)
```

[37]:
```
X.shape, y.shape
```

[37]:
```
((420, 24, 2), (420, 2))
```

Finally, we split our data into a train and a test set, using the last 24 months to test the out-of-sample performance, as shown here:

[38]:
```
test_size =24
train_size = X.shape[0]-test_size
```

[39]:
```
X_train, y_train = X[:train_size], y[:train_size]
X_test, y_test = X[train_size:], y[train_size:]
```

[40]:
```
X_train.shape, X_test.shape
```

[40]:
```
((396, 24, 2), (24, 24, 2))
```

1.5 Define Model Architecture

We use a similar architecture with two stacked LSTM layers with 12 and 6 units, respectively, followed by a fully-connected layer with 10 units. The output layer has two units, one for each time series. We compile them using mean absolute loss and the recommended RMSProp optimizer, as follows:

[70]:
```
n_features = output_size = 2
```

[71]:
```
lstm1_units = 12
lstm2_units = 6
```

[72]:
```
rnn = Sequential([
    LSTM(units=lstm1_units,
```

```
            dropout=.2,
            recurrent_dropout=.2,
            input_shape=(window_size, n_features), name='LSTM1',
            return_sequences=True),
    LSTM(units=lstm2_units,
            dropout=.2,
            recurrent_dropout=.2,
            name='LSTM2'),
    Dense(10, name='FC1'),
    Dense(output_size, name='Output')
])
```

The model has 1,268 parameters, as shown here:

[73]:
```
rnn.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
LSTM1 (LSTM)                 (None, 24, 12)            720
_________________________________________________________________
LSTM2 (LSTM)                 (None, 6)                 456
_________________________________________________________________
FC1 (Dense)                  (None, 10)                70
_________________________________________________________________
Output (Dense)               (None, 2)                 22
=================================================================
Total params: 1,268
Trainable params: 1,268
Non-trainable params: 0
_________________________________________________________________
```

[45]:
```
rnn.compile(loss='mae', optimizer='RMSProp')
```

1.6 Train the Model

We train for 50 epochs with a batch_size value of 20 using early stopping:

[46]:
```
early_stopping = EarlyStopping(monitor='val_loss',
                              patience=5,
                              restore_best_weights=True)
```

[47]:
```
rnn_path = 'models/fred.lstm_{}_{}.weights.best.hdf5'.format(lstm1_units,␣
 ↪lstm2_units)
checkpointer = ModelCheckpoint(filepath=rnn_path,
                              monitor='val_loss',
                              save_best_only=True,
                              save_weights_only=True,
```

```
                                   period=5)
```

[50]:

```
result = rnn.fit(X_train,
                 y_train,
                 epochs=50,
                 batch_size=20,
                 validation_data=(X_test, y_test),
                 callbacks=[checkpointer, early_stopping],
                 verbose=1)
```

```
Train on 396 samples, validate on 24 samples
Epoch 1/50
396/396 [==============================] - 2s 5ms/step - loss: 3.8474 -
val_loss: 2.2458
Epoch 2/50
396/396 [==============================] - 1s 2ms/step - loss: 3.8011 -
val_loss: 2.2199
Epoch 3/50
396/396 [==============================] - 1s 2ms/step - loss: 3.7493 -
val_loss: 2.1867
Epoch 4/50
396/396 [==============================] - 1s 2ms/step - loss: 3.7158 -
val_loss: 2.1477
Epoch 5/50
396/396 [==============================] - 1s 2ms/step - loss: 3.6758 -
val_loss: 2.0966
Epoch 6/50
396/396 [==============================] - 1s 2ms/step - loss: 3.6266 -
val_loss: 2.0551
Epoch 7/50
396/396 [==============================] - 1s 2ms/step - loss: 3.5552 -
val_loss: 2.0113
Epoch 8/50
396/396 [==============================] - 1s 2ms/step - loss: 3.4837 -
val_loss: 1.9549
Epoch 9/50
396/396 [==============================] - 1s 2ms/step - loss: 3.4383 -
val_loss: 1.9126
Epoch 10/50
396/396 [==============================] - 1s 2ms/step - loss: 3.4017 -
val_loss: 1.8605
Epoch 11/50
396/396 [==============================] - 1s 2ms/step - loss: 3.3652 -
val_loss: 1.8217
Epoch 12/50
396/396 [==============================] - 1s 2ms/step - loss: 3.2948 -
val_loss: 1.7720
```

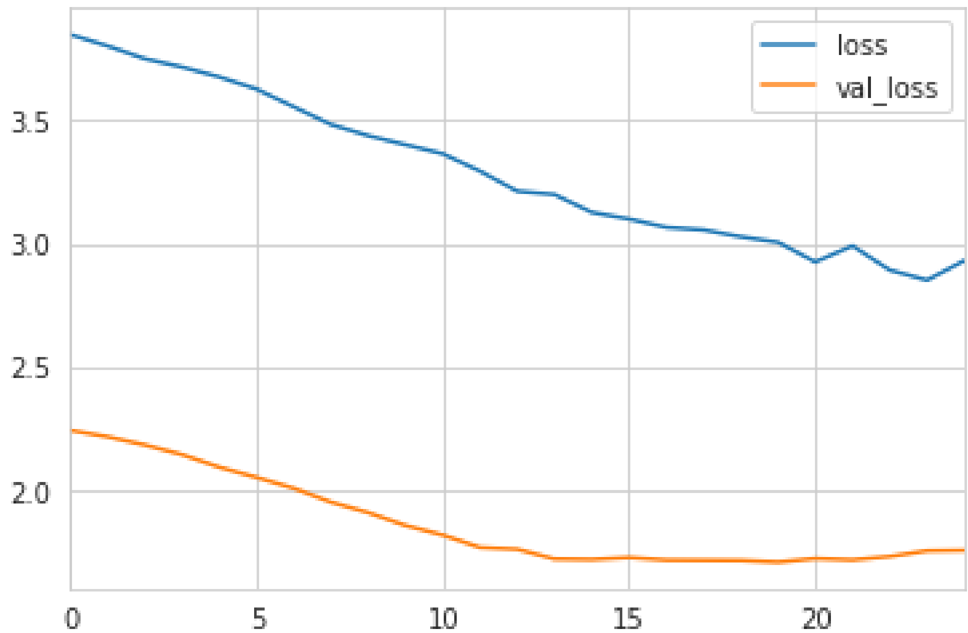
```
Epoch 13/50
396/396 [==============================] - 1s 2ms/step - loss: 3.2134 -
val_loss: 1.7650
Epoch 14/50
396/396 [==============================] - 1s 2ms/step - loss: 3.2003 -
val_loss: 1.7255
Epoch 15/50
396/396 [==============================] - 1s 2ms/step - loss: 3.1277 -
val_loss: 1.7235
Epoch 16/50
396/396 [==============================] - 1s 2ms/step - loss: 3.1009 -
val_loss: 1.7312
Epoch 17/50
396/396 [==============================] - 1s 2ms/step - loss: 3.0671 -
val_loss: 1.7212
Epoch 18/50
396/396 [==============================] - 1s 2ms/step - loss: 3.0568 -
val_loss: 1.7204
Epoch 19/50
396/396 [==============================] - 1s 2ms/step - loss: 3.0291 -
val_loss: 1.7195
Epoch 20/50
396/396 [==============================] - 1s 2ms/step - loss: 3.0069 -
val_loss: 1.7127
Epoch 21/50
396/396 [==============================] - 1s 2ms/step - loss: 2.9270 -
val_loss: 1.7272
Epoch 22/50
396/396 [==============================] - 1s 2ms/step - loss: 2.9920 -
val_loss: 1.7219
Epoch 23/50
396/396 [==============================] - 1s 2ms/step - loss: 2.8941 -
val_loss: 1.7354
Epoch 24/50
396/396 [==============================] - 1s 2ms/step - loss: 2.8532 -
val_loss: 1.7585
Epoch 25/50
396/396 [==============================] - 1s 2ms/step - loss: 2.9323 -
val_loss: 1.7604
```

1.7 Evaluate the Results

Training stops early after 25 epochs, yielding a test MAE of 1.71, which compares favorably to the test MAE for the VAR model of 1.91.

However, the two results are not fully comparable because the RNN model produces 24 one-step-ahead forecasts, whereas the VAR model uses its own predictions as input for its out-of-sample forecast. You may want to tweak the VAR setup to obtain comparable forecasts and compare their performance:

[51]:
```
pd.DataFrame(result.history).plot();
```



[52]:
```
y_pred = pd.DataFrame(rnn.predict(X_test), columns=y_test.columns, index=y_test.
 ↪index)
y_pred.info()
```

```
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 24 entries, 2016-01-01 to 2017-12-01
Data columns (total 2 columns):
ip           24 non-null float32
sentiment    24 non-null float32
dtypes: float32(2)
memory usage: 384.0 bytes
```

[65]:
```
test_mae = mean_absolute_error(y_pred, y_test)
```

[69]:
```
fig, axes = plt.subplots(ncols=3, figsize=(16, 5))
pd.DataFrame(result.history).plot(ax=axes[0], title='Train & Validiation Error')
axes[0].set_xlabel('Epoch')
axes[0].set_ylabel('MAE')
for i, col in enumerate(y_test.columns, 1):
    y_train.loc['2010':, col].plot(ax=axes[i], label='training', title=col)
    y_test[col].plot(ax=axes[i], label='out-of-sample')
    y_pred[col].plot(ax=axes[i], label='prediction')
```

```
    axes[i].set_xlabel('')
plt.legend()
fig.suptitle('Multivariate RNN - Results | Test MAE = {:.2f}'.format(test_mae),␣
 ↪fontsize=16)
fig.tight_layout()
fig.subplots_adjust(top=.85)
fig.savefig('figures/multivariate_results', dpi=300);
```