FeedInMngmt_LSTM_NN

December 16, 2020

1 Environment Set-Up

1.1 Load relevant Python Packages

[1]:
```
reset -fs
```

[2]:
```
# Importing the most important modules
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import warnings
import pickle
import time
from matplotlib import pyplot
import matplotlib.dates as mdates
from tqdm.notebook import tqdm

# Import plotly modules to view time series in a more interactive way
import plotly.graph_objects as go
import plotly.offline as pyo
from matplotlib.pyplot import cm
from IPython.display import Image

# Importing time series split for cross validation of time series models
from sklearn.model_selection import TimeSeriesSplit

# For Data Mining
import os, glob
from pandas import read_csv

# For Data Cleaning
from datetime import datetime
import missingno as msno

# Importing metrics to evaluate the implemented models
from sklearn.metrics import mean_squared_error, r2_score, mean_absolute_error
```

```python
# Imports for LSTM Neural Networks
from numpy import array
from numpy import hstack
from numpy import vstack

from keras.models import Sequential
from keras.layers import LSTM
from keras.layers import Dense
import tensorflow as tf
```

```
Using TensorFlow backend.
```

2 Data Import, Global Variables, Global Settings and Global Functions

2.1 Data Import

[3]:
```python
#data has been saved using a .pkl file.
path = './data/df_small.pkl'
df = pd.read_pickle(path)
df.head(2)
```

[3]:
```
                     power_available_mw_obsnorm  target_losses_norm  \
2018-01-01 06:00:00                    0.911849            0.425598
2018-01-01 06:10:00                    0.932739            0.404513

                     lagged_NetConsumption_MW  lagged_energyprice_euro_MWh  \
2018-01-01 06:00:00               3142.133333                   -71.616667
2018-01-01 06:10:00               3144.800000                   -72.540000

                     dswrf_sfc_wm2  gust_sfc_ms   hpbl_sfc_m     msl_ms_pa  \
2018-01-01 06:00:00            0.0    16.777032  1349.927656  99212.062500
2018-01-01 06:10:00            0.0    16.748651  1350.376965  99220.020833

                     r_pl925_perc  shtfl_sfc_wm2  …  month_transformed_x  \
2018-01-01 06:00:00     89.975000     -58.444885  …                  0.0
2018-01-01 06:10:00     89.854167     -58.558706  …                  0.0

                     month_transformed_y  weekday_transformed_x  \
2018-01-01 06:00:00                  1.0                    0.0
2018-01-01 06:10:00                  1.0                    0.0

                     weekday_transformed_y  ten_min_interval_transformed_x  \
2018-01-01 06:00:00                    1.0                        1.000000
2018-01-01 06:10:00                    1.0                        0.999048

                     ten_min_interval_transformed_y  \
```

```
2018-01-01 06:00:00                       6.123234e-17
2018-01-01 06:10:00                      -4.361939e-02

                     transformed_wdir_100m_dn_x  transformed_wdir_100m_dn_y  \
2018-01-01 06:00:00                    0.581339                    0.813661
2018-01-01 06:10:00                    0.562313                    0.826924

                     transformed_wdir_10m_dn_x  transformed_wdir_10m_dn_y
2018-01-01 06:00:00                    0.61653                   0.787331
2018-01-01 06:10:00                    0.59827                   0.801294

[2 rows x 26 columns]
```

[4]:
```
# Setting the random seed for reproducability and several plotting style␣
↪parameters
%matplotlib inline
plt.style.use('seaborn')
pyo.init_notebook_mode()
sns.set(rc={'figure.figsize':(14,8)})
warnings.filterwarnings('ignore')
pd.set_option('display.max_columns', None)
RSEED = 42
```

[5]:
```
model_name = '2layer_50neurons_30Dropout_Peephole'
```

2.2 Setting Up Training, Validation and Test Dataframes

The dataframe is split into a training set, a validation set (10 consecutive days of data) and a test set (10 consecutive days of data).

[6]:
```
#setting up the consecutive test days (last 10 days of dataframe)
test_timestamps = []
for i in range (10):
    test_timestamps.append(pd.to_datetime(df.index[-1]) - (i+1)*pd.
↪Timedelta(hours=24))
test_timestamps.sort()

#setting up the consecutive validation days (10 form March 17 2019 06:00)
val_timestamps = [pd.to_datetime("2019-03-17 06:00:00")]
for i in range (9):
    val_timestamps.append(pd.to_datetime(val_timestamps[0]) + (i+1)*pd.
↪Timedelta(hours=24))
val_timestamps.sort()
```

[7]:
```
#creating a variable for the lagged target variable in the dataframe
df["y_lag"] = df["target_losses_norm"].shift(1)
df.dropna(inplace = True)
```

```
#splitting dataframe in training, validation and test data
train_df = df[(df.index < val_timestamps[0])]
val_df = df[(df.index >= val_timestamps[0]) & (df.index < val_timestamps[0]+ pd.
↪Timedelta(hours=240))]
test_df = df[(df.index >= test_timestamps[0]) & (df.index < test_timestamps[0]+␣
↪pd.Timedelta(hours=240))]

#calculating means and standard deviations for scaling of features
train_mean = train_df.mean()
train_std = train_df.std()

#saving target variable and lagged target variable
train_y = train_df.target_losses_norm
val_y = val_df.target_losses_norm
test_y = test_df.target_losses_norm
train_lag = train_df.y_lag
val_lag = val_df.y_lag
test_lag = test_df.y_lag

#scaling the features with a fitted (on the training data) scaling method
train_df = (train_df - train_mean) / train_std
val_df = (val_df - train_mean) / train_std
test_df = (test_df - train_mean) / train_std

#replacing target variable and lagged target variable with the unscaled values
train_df["target_losses_norm"] = train_y
val_df["target_losses_norm"] = val_y
test_df["target_losses_norm"] = test_y
train_df["y_lag"] = train_lag
val_df["y_lag"] = val_lag
test_df["y_lag"] = test_lag
```

2.3 Error Metrics Function (RMSE, R2, MAE, MAPE)

[8]:
```
def error_metrics(y_pred, y_truth, model_name = "default"):
    """
    Calculate error metrics for a single comparison between predicted and␣
↪observed values
    """
    # calculating error metrics
    RMSE_return = np.sqrt(mean_squared_error(y_truth, y_pred))
    R2_return = r2_score(y_truth, y_pred)
    MAE_return = mean_absolute_error(y_truth, y_pred)
    MAPE_return = (np.mean(np.abs((y_truth - y_pred) / y_truth)) * 100)
```

```
    # saving error metrics in a dataframe and returning it
    name_error = ['RMSE', 'R2', 'MAE', 'MAPE']
    value_error = [RMSE_return, R2_return, MAE_return, MAPE_return/100]
    dict_error = dict()
    for i in range(len(name_error)):
        dict_error[name_error[i]] = [value_error[i]]
    errors = pd.DataFrame(dict_error).T
    errors.rename(columns={0 : model_name}, inplace = True)

    #path = './data/error_metrics_{}.pkl'.format(model_name)
    #errors.to_pickle(path)

    return(errors)
```

2.4 Compile and Fit

[14]:
```
def compile_and_fit(model, X_train, y_train, max_epochs=30, patience=2,␣
↪pred_step = 'X'):
    """
    Compiling and fitting a set up neural network with early stopping based on␣
↪the mean absoulte
    percentage error of the epochs
    """

    # TensorBoard Specification `log_dir` will later be the name used in the␣
↪TensorBoard
    log_dir = "logs/fit/" + datetime.now().strftime("%Y%m%d-%H%M") +'_'+␣
↪exp_name +'_s'+ str(pred_step)
    tensorboard_callback = tf.keras.callbacks.TensorBoard(log_dir=log_dir,␣
↪histogram_freq=1)

    # Early Stopping Specifications
    early_stopping = tf.keras.callbacks.EarlyStopping(monitor='loss',
                                                      patience=patience,
                                                      mode='min')

    model.compile(loss=tf.losses.MeanAbsolutePercentageError(),
                  optimizer=tf.optimizers.Adam(),
                  metrics=[tf.metrics.MeanAbsolutePercentageError()])
    # note: the tensorboard callback slows the model training down, if␣
↪TensorBoard is not used to analyse the performance of models, deletion of␣
↪tensorboard_callback is advised
    history = model.fit(X_train, y_train, epochs=max_epochs,
                        callbacks=[tensorboard_callback, early_stopping],
                        verbose = 0)
    return history
```

2.5 Split Sequence

[10]:
```
# split a univariate sequence into samples
def split_sequence(sequence, n_steps_in, n_steps_out, pred_step = 1):
    """
    Splitting a single input sequence based on the chosen number of input and␣
↪output steps and returning
    the chosen outputvalues in y
    - sequences: horizontally stacked sequences (last column must be target␣
↪column)
    - n_steps_in: size of the input window for predictions
    - n_steps_out: size of the possible output window (only one value will be␣
↪returned in y)
    - pred_step: chosen timestep to predict
    """
    X, y = list(), list()
    for i in range(len(sequence)):
        # find the end of this pattern
        end_ix = i + n_steps_in
        out_end_ix = end_ix + n_steps_out
        # check if we are beyond the sequence
        if out_end_ix > len(sequence):
            break
        # gather input and output parts of the pattern
        seq_x, seq_y = sequence[i:end_ix], sequence[end_ix:out_end_ix]
        X.append(seq_x)
        y.append(seq_y[pred_step-1])
    return array(X), array(np.vstack(el for el in y))
```

2.6 Split Sequences

[11]:
```
# split a multivariate sequence into samples
def split_sequences(sequences, n_steps_in, n_steps_out, pred_step = 1):
    """
    Splitting multiple input sequences based on the chosen number of input and␣
↪output steps and returning
    the chosen outputvalues in y
    - sequences: horizontally stacked sequences (last column must be target␣
↪column)
    - n_steps_in: size of the input window for predictions
    - n_steps_out: size of the possible output window (only one value will be␣
↪returned in y)
    - pred_step: chosen timestep to predict
    """
    X, y = list(), list()
    for i in range(len(sequences)):
        # find the end of this pattern
```

```
        end_ix = i + n_steps_in
        out_end_ix = end_ix + n_steps_out
        # check if beyond the dataset
        if end_ix > len(sequences)-n_steps_out:
            break
        # gather input and output parts of the pattern
        seq_x, seq_y = sequences[i:end_ix, :-1], sequences[end_ix:out_end_ix,␣
↪-1]
        X.append(seq_x)
        y.append(seq_y[pred_step-1])
    return array(X), array(np.vstack(el for el in y))
```

3 Modeling

3.0.1 Multivariate LSTM with Multi-Step Prediction

[12]:
```
prediction_steps = 18
```

In the following 18 different neural networks, will be set up, trained and tested; One for each of the 18 future timesteps to predict. The models will take the last 3 values as input and predict up to 18 steps into the future.

[ ]:
```
val_predictions = []
val_observed = []
test_predictions = []
test_observed = []

for j in tqdm(range(1,prediction_steps+1)):


    #### TRAINING


    # defining training input sequences
    train_df_X = train_df.copy(deep = True)
    train_df_X.drop(columns = ["target_losses_norm"], inplace = True)
    train_out_seq = train_df["target_losses_norm"]
    storage_list = list()
    for col in train_df_X.columns:
        storage_list.append(train_df_X[col].to_numpy())
    storage_list.append(train_out_seq.to_numpy())

    for i in range(len(storage_list)):
        storage_list[i] = storage_list[i].reshape((len(storage_list[i]), 1))

    # horizontally stack columns in training dataset
    train_dataset = hstack(tuple((seq for seq in storage_list)))
```

```
    # setting up the input timesteps, output timesteps and the timestep of the␣
↪outputs, which shall be predicted
    n_steps_in, n_steps_out, pred_step = 3, prediction_steps, j

    # convert sequences into input/output
    X_train, y_train = split_sequences(train_dataset, n_steps_in, n_steps_out,␣
↪pred_step)

    # the dataset knows the number of features, e.g. 2
    n_features = X_train.shape[2]

    # setting up the model architecture
    model = Sequential()
    model.add(tf.keras.layers.RNN(tf.keras.experimental.PeepholeLSTMCell(50,␣
↪activation = "relu"), return_sequences=True, input_shape=(n_steps_in,␣
↪n_features)))
    model.add(tf.keras.layers.RNN(tf.keras.experimental.PeepholeLSTMCell(50,␣
↪activation = "relu"), return_sequences=True,))
    model.add(Dense(1))

    # model without peephole
    #model = Sequential()
    #model.add(LSTM(50, activation='relu', return_sequences=True,␣
↪input_shape=(n_steps_in, n_features)))
    #model.add(LSTM(50, activation='relu'))
    #model.add(Dense(1))

    # compiling and fitting the model
    history = compile_and_fit(model, X_train, y_train, max_epochs = 30,␣
↪patience=2, pred_step = pred_step)

    #### VALIDATION


    # defining and shaping validation input sequences
    val_df_X = val_df.copy(deep = True)
    val_df_X.drop(columns = ["target_losses_norm"], inplace = True)
    val_out_seq = val_df["target_losses_norm"]
    storage_list = list()

    for col in val_df_X.columns:
        storage_list.append(val_df_X[col].to_numpy())
    storage_list.append(val_out_seq.to_numpy())

    for i in range(len(storage_list)):
```

```
        storage_list[i] = storage_list[i].reshape((len(storage_list[i]), 1))

    # horizontally stack columns
    val_dataset = hstack(tuple((seq for seq in storage_list)))

    # convert sequences into input/output
    X_val, y_val = split_sequences(val_dataset, n_steps_in, n_steps_out,␣
↪pred_step)

    # predicting the target variable for the validation set
    y_val_pred = (model.predict(X_val, verbose=0))

    val_predictions.append(y_val_pred)
    val_observed.append(y_val)


    #### TESTING


    # defining and shaping test input sequences
    test_df_X = test_df.copy(deep = True)
    test_df_X.drop(columns = ["target_losses_norm"], inplace = True)
    test_out_seq = test_df["target_losses_norm"]
    storage_list = list()

    for col in test_df_X.columns:
        storage_list.append(test_df_X[col].to_numpy())
    storage_list.append(test_out_seq.to_numpy())

    for i in range(len(storage_list)):
        storage_list[i] = storage_list[i].reshape((len(storage_list[i]), 1))

    # horizontally stack columns
    test_dataset = hstack(tuple((seq for seq in storage_list)))

    # convert sequences into input/output
    X_test, y_test = split_sequences(test_dataset, n_steps_in, n_steps_out,␣
↪pred_step)

    # predicting the target variable for the test set
    y_test_pred = (model.predict(X_test, verbose=0))

    test_predictions.append(y_test_pred)
    test_observed.append(y_test)


#### ERROR METRICS
```

```
val_pred_columnnames = list()
val_observed_columnnames = list()
test_pred_columnnames = list()
test_observed_columnnames = list()
val_errors_columnnames = list()
test_errors_columnnames = list()

for i in range(prediction_steps):
    val_pred_columnnames.append(f"y_val_pred Step {i+1}")
    val_observed_columnnames.append(f"y_val_observed Step {i+1}")
    test_pred_columnnames.append(f"y_test_pred Step {i+1}")
    test_observed_columnnames.append(f"y_test_observed Step {i+1}")
    val_errors_columnnames.append(f"Validation Errors Step {i+1}")
    test_errors_columnnames.append(f"Test Errors Step {i+1}")

y_val_pred = pd.DataFrame(columns = val_pred_columnnames)
y_val_observed = pd.DataFrame(columns = val_observed_columnnames)
y_test_pred = pd.DataFrame(columns = test_pred_columnnames)
y_test_observed = pd.DataFrame(columns = test_observed_columnnames)

val_errors = pd.DataFrame(columns = val_errors_columnnames)
test_errors = pd.DataFrame(columns = test_errors_columnnames)

for i in range(prediction_steps):
    y_val_pred[f"y_val_pred Step {i+1}"] = pd.Series(v[0] for v in␣
 ↪val_predictions[i])
    y_val_observed[f"y_val_observed Step {i+1}"] = pd.Series(v[0] for v in␣
 ↪val_observed[i])
    y_test_pred[f"y_test_pred Step {i+1}"] = pd.Series(v[0] for v in␣
 ↪test_predictions[i])
    y_test_observed[f"y_test_observed Step {i+1}"] = pd.Series(v[0] for v in␣
 ↪test_observed[i])
    val_errors[f"Validation Errors Step {i+1}"]  =␣
 ↪error_metrics(y_val_pred[f"y_val_pred Step␣
 ↪{i+1}"],y_val_observed[f"y_val_observed Step {i+1}"])["default"]
    test_errors[f"Test Errors Step {i+1}"]  =␣
 ↪error_metrics(y_test_pred[f"y_test_pred Step␣
 ↪{i+1}"],y_test_observed[f"y_test_observed Step {i+1}"])["default"]

val_errors_multi = val_errors.T.copy(deep = True)
test_errors_multi  = test_errors.T.copy(deep = True)
```

[ ]:
```
val_errors_multi
```

```
[ ]: test_errors_multi
```

3.0.2 Univariate LSTM with Multi-Step Prediction

```
[ ]: prediction_steps = 18
```

In the following 18 different neural networks, will be set up, trained and tested; One for each of the 18 future timesteps to predict. The models will take the last 3 values as input and predict up to 18 steps into the future.

```
[ ]: val_predictions = []
val_observed = []
test_predictions = []
test_observed = []

for j in tqdm(range(1,prediction_steps+1)):


    #### TRAINING


    # defining training input sequences
    train_out_seq = train_df["target_losses_norm"]

    # setting up the input timesteps, output timesteps and the timestep of the␣
↪outputs, which shall be predicted
    n_steps_in, n_steps_out, pred_step = 3, prediction_steps, j

    # convert sequences into input/output
    X_train, y_train = split_sequence(train_out_seq, n_steps_in, n_steps_out,␣
↪pred_step)

    # the dataset knows the number of features == 1
    n_features = 1
    X_train = X_train.reshape((X_train.shape[0], X_train.shape[1], n_features))

    # setting up the model architecture
    model = Sequential()
    model.add(tf.keras.layers.RNN(tf.keras.experimental.PeepholeLSTMCell(50,␣
↪activation = "relu"), return_sequences=True, input_shape=(n_steps_in,␣
↪n_features)))
    model.add(tf.keras.layers.RNN(tf.keras.experimental.PeepholeLSTMCell(50,␣
↪activation = "relu"), return_sequences=True,))
    model.add(Dense(1))

    # best performin model architecture without peephole
    #model = Sequential()
```

```
    #model.add(LSTM(50, activation='relu', return_sequences=True,␣
↪input_shape=(n_steps_in, n_features)))
    #model.add(LSTM(50, activation='relu'))
    #model.add(Dense(1))

    # compiling and fitting the model
    history = compile_and_fit(model, X_train, y_train, max_epochs = 30,␣
↪patience=2, pred_step = pred_step)


    #### VALIDATION


    # defining and shaping validation input sequences
    val_seq = val_df["target_losses_norm"]
    X_val, y_val = split_sequence(val_seq, n_steps_in, n_steps_out)

    # predicting target variable for validation set
    y_val_pred = list()
    for el in X_val:
        X_in = el.reshape((1, n_steps_in, n_features))
        y_val_pred.append(model.predict(X_in, verbose=0))

    val_predictions.append(y_val_pred)
    val_observed.append(y_val)


    #### TESTING


    # defining and shaping test input sequences
    test_seq = test_df["target_losses_norm"]
    X_test, y_test = split_sequence(test_seq, n_steps_in, n_steps_out)

    # predicting target variable for test set
    y_test_pred = list()
    for el in X_test:
        X_in = el.reshape((1, n_steps_in, n_features))
        y_test_pred.append(model.predict(X_in, verbose=0))

    test_predictions.append(y_test_pred)
    test_observed.append(y_test)


#### ERROR METRICS
```

```
val_pred_columnnames = list()
val_observed_columnnames = list()
test_pred_columnnames = list()
test_observed_columnnames = list()
val_errors_columnnames = list()
test_errors_columnnames = list()

for i in range(prediction_steps):
    val_pred_columnnames.append(f"y_val_pred Step {i+1}")
    val_observed_columnnames.append(f"y_val_observed Step {i+1}")
    test_pred_columnnames.append(f"y_test_pred Step {i+1}")
    test_observed_columnnames.append(f"y_test_observed Step {i+1}")
    val_errors_columnnames.append(f"Validation Errors Step {i+1}")
    test_errors_columnnames.append(f"Test Errors Step {i+1}")

y_val_pred = pd.DataFrame(columns = val_pred_columnnames)
y_val_observed = pd.DataFrame(columns = val_observed_columnnames)
y_test_pred = pd.DataFrame(columns = test_pred_columnnames)
y_test_observed = pd.DataFrame(columns = test_observed_columnnames)

val_errors = pd.DataFrame(columns = val_errors_columnnames)
test_errors = pd.DataFrame(columns = test_errors_columnnames)

for i in range(prediction_steps):
    y_val_pred[f"y_val_pred Step {i+1}"] = pd.Series(v[0][0] for v in␣
 ↪val_predictions[i])
    y_val_observed[f"y_val_observed Step {i+1}"] = pd.Series(v[0] for v in␣
 ↪val_observed[i])
    y_test_pred[f"y_test_pred Step {i+1}"] = pd.Series(v[0][0] for v in␣
 ↪test_predictions[i])
    y_test_observed[f"y_test_observed Step {i+1}"] = pd.Series(v[0] for v in␣
 ↪test_observed[i])
    val_errors[f"Validation Errors Step {i+1}"]  =␣
 ↪error_metrics(y_val_pred[f"y_val_pred Step␣
 ↪{i+1}"],y_val_observed[f"y_val_observed Step {i+1}"])["default"]
    test_errors[f"Test Errors Step {i+1}"]  =␣
 ↪error_metrics(y_test_pred[f"y_test_pred Step␣
 ↪{i+1}"],y_test_observed[f"y_test_observed Step {i+1}"])["default"]

val_errors_uni = val_errors.T
test_errors_uni  = test_errors.T
```

[ ]:
```
val_errors_uni
```

[ ]:
```
test_errors_uni
```

[ ]:
```
val_errors_multi.to_csv(f"./Validation Errors/
↪validation_errors_multivariate_lstmNN_{model_name}.csv", index_label =␣
↪"Step")
test_errors_multi.to_csv(f"./Test Errors/
↪test_errors_multivariate_lstmNN_{model_name}.csv", index_label = "Step")
val_errors_uni.to_csv(f"./Validation Errors/
↪validation_errors_univariate_lstmNN_{model_name}.csv", index_label = "Step")
test_errors_uni.to_csv(f"./Test Errors/
↪test_errors_univariate_lstmNN_{model_name}.csv", index_label = "Step")
```

3.1 Performance Analysis in TensorBoard

TensorBoard can be used to analyse the performance of different LSTMs over the traing epochs. The TensorBoard can be started via the following command

`!tensorboard --logdir logs/fit`

A Screenshot of the TensorBoard below. Within TensorBoard, the LSTMs are organized by the model name and the prediction step.

[15]:
```
#!tensorboard --logdir logs/fit
```

[16]:
```
print('This cell was last run on: ')
print(datetime.now())
```

```
This cell was last run on:
2020-11-26 12:37:56.565781
```