This Notebook is a sort of tutorial for the beginners in Deep Learning and time-series data analysis.

* The aim is just to show how to build the simplest Long Short-Term Memory (LSTM) recurrent neural network for the data.

The description of data can be found here:
http://archive.ics.uci.edu/ml/datasets/Individual+household+electric+power+consumption

Attribute Information: 1.date: Date in format dd/mm/yyyy

2.time: time in format hh:mm:ss

3.global_active_power: household global minute-averaged active power (in kilowatt)

4.global_reactive_power: household global minute-averaged reactive power (in kilowatt)

5.voltage: minute-averaged voltage (in volt)

6.global_intensity: household global minute-averaged current intensity (in ampere)

7.sub_metering_1: energy sub-metering No. 1 (in watt-hour of active energy). It corresponds to the kitchen, containing mainly a dishwasher, an oven and a microwave (hot plates are not electric but gas powered).

8.sub_metering_2: energy sub-metering No. 2 (in watt-hour of active energy). It corresponds to the laundry room, containing a washing-machine, a tumble-drier, a refrigerator and a light.

9.sub_metering_3: energy sub-metering No. 3 (in watt-hour of active energy). It corresponds to an electric water-heater and an air-conditioner.

```
# Let`s import all packages that we may need:

import sys
import numpy as np # linear algebra
from scipy.stats import randint
import pandas as pd # data processing, CSV file I/O (e.g. pd.read_csv), data manipulation as
import matplotlib.pyplot as plt # this is used for the plot the graph
import seaborn as sns # used for plot interactive graph.
from sklearn.model_selection import train_test_split # to split the data into two parts

from sklearn.preprocessing import StandardScaler # for normalization
from sklearn.preprocessing import MinMaxScaler
from sklearn.pipeline import Pipeline # pipeline making
from sklearn.model_selection import cross_val_score
```

```
from sklearn.feature_selection import SelectFromModel
from sklearn import metrics # for the check the error and accuracy of the model
from sklearn.metrics import mean_squared_error,r2_score

## for Deep-learing:
import keras
from keras.layers import Dense
from keras.models import Sequential
from tensorflow.keras.utils import to_categorical
from tensorflow.keras.optimizers import SGD
from keras.callbacks import EarlyStopping
from keras.utils import np_utils
import itertools
from keras.layers import LSTM
from keras.layers.convolutional import Conv1D
from keras.layers.convolutional import MaxPooling1D
from keras.layers import Dropout
```

```
from google.colab import drive

drive.mount('/content/drive/')
```

```
     Mounted at /content/drive/
```

```
## Just open the zip file and grab the file 'household_power_consumption.txt' put it in the d
## that you would like to run the code.
```

```
df = pd.read_csv('/content/drive/MyDrive/DTS/household_power_consumption.txt', sep=';',
                 parse_dates={'dt' : ['Date', 'Time']}, infer_datetime_format=True,
                 low_memory=False, na_values=['nan','?'], index_col='dt')
```

Exploratory Data

```
df.head()
```

```
df.info()
```

```
df.dtypes
```

```
df.shape
```

```
df.describe()
```

```
df.columns
```

```
Index(['Global_active_power', 'Global_reactive_power', 'Voltage',
       'Global_intensity', 'Sub_metering_1', 'Sub_metering_2',
       'Sub_metering_3'],
      dtype='object')
```

Dealing with missing values 'nan' with a test statistic

```
## finding all columns that have nan:

droping_list_all=[]
for j in range(0,7):
    if not df.iloc[:, j].notnull().all():
        droping_list_all.append(j)
        #print(df.iloc[:,j].unique())
droping_list_all
```

```
[0, 1, 2, 3, 4, 5, 6]
```

```
# filling nan with mean in any columns

for j in range(0,7):
        df.iloc[:,j]=df.iloc[:,j].fillna(df.iloc[:,j].mean())
```

```
# another sanity check to make sure that there are not more any nan
df.isnull().sum()
```

```
Global_active_power      0
Global_reactive_power    0
Voltage                  0
Global_intensity         0
Sub_metering_1           0
Sub_metering_2           0
Sub_metering_3           0
dtype: int64
```

Data visualization

* Below I resample over day, and show the sum and mean of Global_active_power. It is seen that mean and sum of resampled data set, have similar structure.

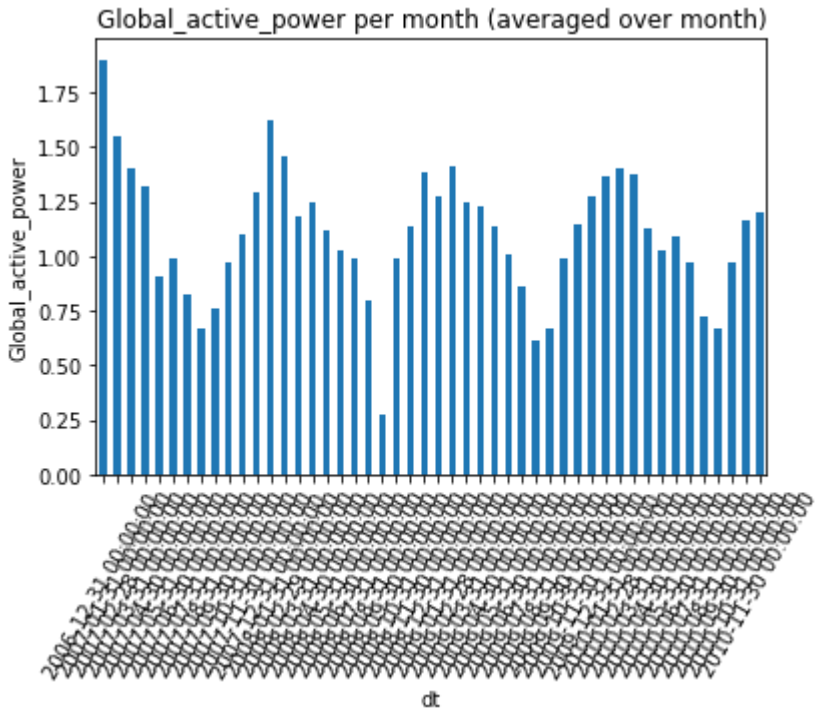
```
df.Global_active_power.resample('D').sum().plot(title='Global_active_power resampled over day
#df.Global_active_power.resample('D').mean().plot(title='Global_active_power resampled over d
plt.tight_layout()
plt.show()

df.Global_active_power.resample('D').mean().plot(title='Global_active_power resampled over da
plt.tight_layout()
plt.show()


### Below I show mean and std of 'Global_intensity' resampled over day
r = df.Global_intensity.resample('D').agg(['mean', 'std'])
r.plot(subplots = True, title='Global_intensity resampled over day')
plt.show()


### Below I show mean and std of 'Global_reactive_power' resampled over day
r2 = df.Global_reactive_power.resample('D').agg(['mean', 'std'])
r2.plot(subplots = True, title='Global_reactive_power resampled over day', color='red')
plt.show()


### Sum of 'Global_active_power' resampled over month
# Sum of 'Global_active_power' resampled over month
df['Global_active_power'].resample('M').mean().plot(kind='bar')
plt.xticks(rotation=60)
plt.ylabel('Global_active_power')
plt.title('Global_active_power per month (averaged over month)')
plt.show()
```

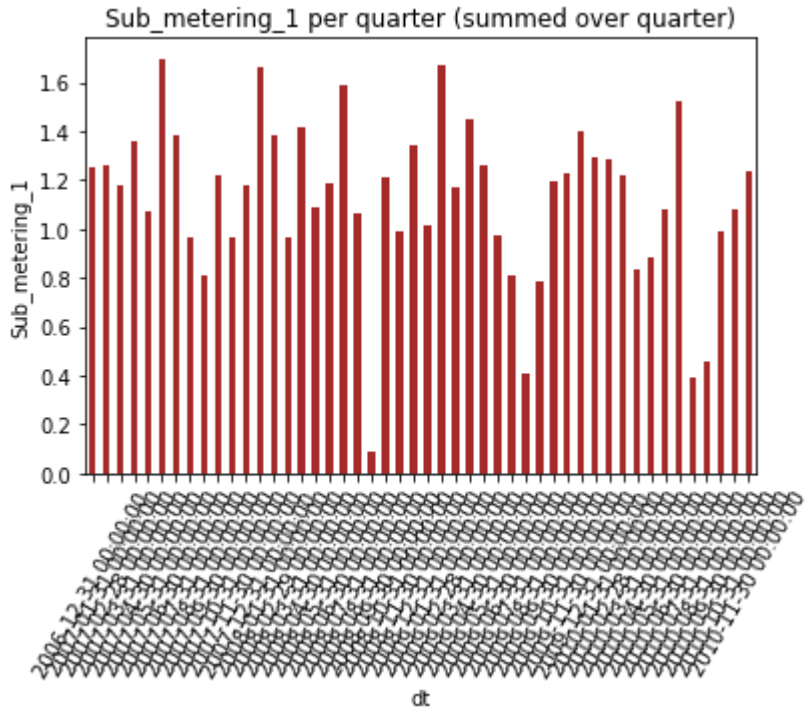


```
## Mean of 'Global_active_power' resampled over quarter
df['Global_active_power'].resample('Q').mean().plot(kind='bar')
plt.xticks(rotation=60)
plt.ylabel('Global_active_power')
plt.title('Global_active_power per quarter (averaged over quarter)')
plt.show()
```

It is very important to note from above two plots that resampling over larger time inteval, will diminish the periodicity of system as we expect. This is important for machine learning feature engineering.

```
## mean of 'Voltage' resampled over month
df['Voltage'].resample('M').mean().plot(kind='bar', color='red')
plt.xticks(rotation=60)
plt.ylabel('Voltage')
plt.title('Voltage per quarter (summed over quarter)')
plt.show()
```

```
df['Sub_metering_1'].resample('M').mean().plot(kind='bar', color='brown')
plt.xticks(rotation=60)
plt.ylabel('Sub_metering_1')
plt.title('Sub_metering_1 per quarter (summed over quarter)')
plt.show()
```



* It is seen from the above plots that the mean of 'Volage' over month is pretty much constant compared to other features. This is important again in feature selection.

```
# Below I compare the mean of different featuresresampled over day.
# specify columns to plot
cols = [0, 1, 2, 3, 5, 6]
i = 1
groups=cols
values = df.resample('D').mean().values
# plot each column
plt.figure(figsize=(15, 10))
for group in groups:
    plt.subplot(len(cols), 1, i)
    plt.plot(values[:, group])
    plt.title(df.columns[group], y=0.75, loc='right')
    i += 1
plt.show()
```

```
## resampling over week and computing mean
df.Global_reactive_power.resample('W').mean().plot(color='y', legend=True)
df.Global_active_power.resample('W').mean().plot(color='r', legend=True)
df.Sub_metering_1.resample('W').mean().plot(color='b', legend=True)
df.Global_intensity.resample('W').mean().plot(color='g', legend=True)
plt.show()
```

```
# Below I show hist plot of the mean of different feature resampled over month
df.Global_active_power.resample('M').mean().plot(kind='hist', color='r', legend=True )
df.Global_reactive_power.resample('M').mean().plot(kind='hist',color='b', legend=True)
#df.Voltage.resample('M').sum().plot(kind='hist',color='g', legend=True)
df.Global_intensity.resample('M').mean().plot(kind='hist', color='g', legend=True)
df.Sub_metering_1.resample('M').mean().plot(kind='hist', color='y', legend=True)
plt.show()
```


```
## The correlations between 'Global_intensity', 'Global_active_power'
data_returns = df.pct_change()
sns.jointplot(x='Global_intensity', y='Global_active_power', data=data_returns)

plt.show()
```

* From above two plots it is seen that 'Global_intensity' and 'Global_active_power' correlated. But 'Voltage', 'Global_active_power' are less correlated. This is important observation for machine learning purpose.

```
## The correlations between 'Voltage' and  'Global_active_power'
sns.jointplot(x='Voltage', y='Global_active_power', data=data_returns)
plt.show()
```

Correlations among features

```
# Correlations among columns
plt.matshow(df.corr(method='spearman'),vmax=1,vmin=-1,cmap='PRGn')
plt.title('without resampling', size=15)
plt.colorbar()
plt.show()
```

```
# Correlations of mean of features resampled over months
plt.matshow(df.resample('M').mean().corr(method='spearman'),vmax=1,vmin=-1,cmap='PRGn')
plt.title('resampled over month', size=15)
plt.colorbar()
plt.margins(0.02)
plt.matshow(df.resample('A').mean().corr(method='spearman'),vmax=1,vmin=-1,cmap='PRGn')
plt.title('resampled over year', size=15)
plt.colorbar()
plt.show()
```

* It is seen from above that with resampling techniques one can change the correlations among features. This is important for feature engineering.

Machine-Leaning: LSTM Data Preparation and feature engineering

* I will apply recurrent nueral network (LSTM) which is best suited for time-seriers and sequential problem. This approach is the best if we have large data.

* I will frame the supervised learning problem as predicting the Global_active_power at the current time (t) given the Global_active_power measurement and other features at the prior time step.

```
def series_to_supervised(data, n_in=1, n_out=1, dropnan=True):
    n_vars = 1 if type(data) is list else data.shape[1]
    dff = pd.DataFrame(data)
    cols, names = list(), list()
    # input sequence (t-n, ... t-1)
    for i in range(n_in, 0, -1):
        cols.append(dff.shift(i))
        names += [('var%d(t-%d)' % (j+1, i)) for j in range(n_vars)]
    # forecast sequence (t, t+1, ... t+n)
    for i in range(0, n_out):
        cols.append(dff.shift(-i))
        if i == 0:
            names += [('var%d(t)' % (j+1)) for j in range(n_vars)]
        else:
            names += [('var%d(t+%d)' % (j+1, i)) for j in range(n_vars)]
    # put it all together
    agg = pd.concat(cols, axis=1)
    agg.columns = names
    # drop rows with NaN values
    if dropnan:
        agg.dropna(inplace=True)
    return agg
```

* In order to reduce the computation time, and also get a quick result to test the model. One can resmaple the data over hour (the original data are given in minutes). This will reduce the size of data from 2075259 to 34589 but keep the overall strucure of data as shown in the above.

```
## resampling of data over hour
df_resample = df.resample('h').mean()
df_resample.shape
```

```
(34589, 7)
```

```
## * Note: I scale all features in range of [0,1].

## If you would like to train based on the resampled data (over hour), then used below
values = df_resample.values


## full data without resampling
#values = df.values

# integer encode direction
# ensure all data is float
#values = values.astype('float32')
# normalize features
scaler = MinMaxScaler(feature_range=(0, 1))
scaled = scaler.fit_transform(values)
# frame as supervised learning
reframed = series_to_supervised(scaled, 1, 1)

# drop columns we don't want to predict
reframed.drop(reframed.columns[[8,9,10,11,12,13]], axis=1, inplace=True)
print(reframed.head())
```

* Above I showed 7 input variables (input series) and the 1 output variable for 'Global_active_power' at the current time in hour (depending on resampling).

Splitting the rest of data to train and validation sets

* First, I split the prepared dataset into train and test sets. To speed up the training of the model (for the sake of the demonstration), we will only train the model on the first year of data, then evaluate it on the next 3 years of data.

```
# split into train and test sets
values = reframed.values

n_train_time = 365*24
train = values[:n_train_time, :]
```

```
test = values[n_train_time:, :]
##test = values[n_train_time:n_test_time, :]
# split into input and outputs
train_X, train_y = train[:, :-1], train[:, -1]
test_X, test_y = test[:, :-1], test[:, -1]
# reshape input to be 3D [samples, timesteps, features]
train_X = train_X.reshape((train_X.shape[0], 1, train_X.shape[1]))
test_X = test_X.reshape((test_X.shape[0], 1, test_X.shape[1]))
print(train_X.shape, train_y.shape, test_X.shape, test_y.shape)
# We reshaped the input into the 3D format as expected by LSTMs, namely [samples, timesteps,
```

```
(8760, 1, 7) (8760,) (25828, 1, 7) (25828,)
```

Model architecture

1) LSTM with 100 neurons in the first visible layer

3) dropout 20%

4) 1 neuron in the output layer for predicting Global_active_power.

5) The input shape will be 1 time step with 7 features.

6) I use the Mean Absolute Error (MAE) loss function and the efficient Adam version of stochastic gradient descent.

7) The model will be fit for 20 training epochs with a batch size of 70.

```
model = Sequential()
model.add(LSTM(100, input_shape=(train_X.shape[1], train_X.shape[2])))
model.add(Dropout(0.2))
model.add(Dense(1))
model.compile(loss='mean_squared_error', optimizer='adam')

# fit network
history = model.fit(train_X, train_y, epochs=20, batch_size=70, validation_data=(test_X, test_

# summarize history for loss
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('model loss')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='upper right')
plt.show()
```

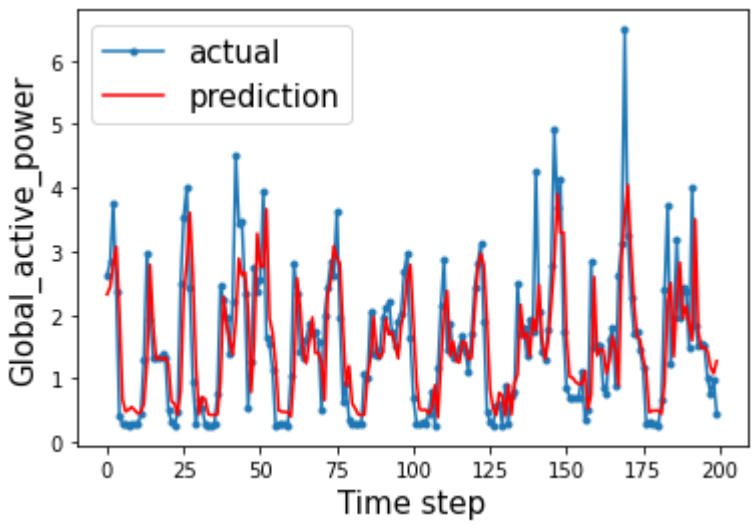
```
#·make·a·prediction
yhat·=·model.predict(test_X)
test_X·=·test_X.reshape((test_X.shape[0],·7))
#·invert·scaling·for·forecast
inv_yhat·=·np.concatenate((yhat,·test_X[:,·-6:]),·axis=1)
inv_yhat·=·scaler.inverse_transform(inv_yhat)
inv_yhat·=·inv_yhat[:,0]
#·invert·scaling·for·actual
test_y·=·test_y.reshape((len(test_y),·1))
inv_y·=·np.concatenate((test_y,·test_X[:,·-6:]),·axis=1)
inv_y·=·scaler.inverse_transform(inv_y)
inv_y·=·inv_y[:,0]
#·calculate·RMSE
rmse·=·np.sqrt(mean_squared_error(inv_y,·inv_yhat))
print('Test·RMSE:·%.3f'·%·rmse)
```

```
Epoch 1/20
126/126 - 4s - loss: 0.0201 - val_loss: 0.0118 - 4s/epoch - 29ms/step
Epoch 2/20
126/126 - 1s - loss: 0.0126 - val_loss: 0.0106 - 901ms/epoch - 7ms/step
Epoch 3/20
126/126 - 1s - loss: 0.0114 - val_loss: 0.0097 - 943ms/epoch - 7ms/step
Epoch 4/20
126/126 - 1s - loss: 0.0109 - val_loss: 0.0094 - 860ms/epoch - 7ms/step
Epoch 5/20
126/126 - 1s - loss: 0.0106 - val_loss: 0.0093 - 1s/epoch - 8ms/step
Epoch 6/20
126/126 - 1s - loss: 0.0106 - val_loss: 0.0093 - 934ms/epoch - 7ms/step
Epoch 7/20
126/126 - 1s - loss: 0.0105 - val_loss: 0.0093 - 1s/epoch - 8ms/step
Epoch 8/20
126/126 - 1s - loss: 0.0105 - val_loss: 0.0092 - 993ms/epoch - 8ms/step
Epoch 9/20
126/126 - 1s - loss: 0.0105 - val_loss: 0.0093 - 898ms/epoch - 7ms/step
Epoch 10/20
```

Note that in order to improve the model, one has to adjust epochs and batch_size.

```
##·time·steps,·every·step·is·one·hour·(you·can·easily·convert·the·time·step·to·the·actual·tim
##·for·a·demonstration·purpose,·I·only·compare·the·predictions·in·200·hours.·

aa=[x·for·x·in·range(200)]
plt.plot(aa,·inv_y[:200],·marker='.',·label="actual")
plt.plot(aa,·inv_yhat[:200],·'r',·label="prediction")
plt.ylabel('Global_active_power',·size=15)
plt.xlabel('Time·step',·size=15)
plt.legend(fontsize=15)
plt.show()
```



Final remarks

* Here I have used the LSTM neural network which is now the state-of-the-art for sequencial problems.

* In order to reduce the computation time, and get some results quickly, I took the first year of data (resampled over hour) to train the model and the rest of data to test the model.

* I put together a very simple LSTM neural-network to show that one can obtain reasonable predictions. However numbers of rows is too high and as a result the computation is very time-consuming (even for the simple model in the above it took few mins to be run on 2.8 GHz Intel Core i7). The Best is to write the last part of code using Spark (MLlib) running on GPU.

* Moreover, the neural-network architecture that I have designed is a toy model. It can be easily improved by adding CNN and dropout layers. The CNN is useful here since there are correlations in data (CNN layer is a good way to probe the local structure of data).

✓ 0s completed at 6:54 PM