Demand Forecasting Project

1. Download electricityLoadData.csv from the shared box and upload to your own google drive
2. Instructions for coding (What you need to code):
    - Load data and preprocess data using the predefined functions.
    - Go to `build_model` section and define your own network.
    - Choose your parameters to set up the training routine.
    - Plot your result by using the `Analyze Result` section.
    - Print your notebook by clicking File > Save > PDF to upload your work.

Import necessary libraries

```
import numpy as np
import pandas as pd
import plotly.graph_objs as go
from sklearn import preprocessing
from sklearn.metrics import mean_absolute_error
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, LSTM
```

Set up Plotly credentials

```
import plotly.io as pio
pio.renderers.default = "notebook_connected"
```

Define constants

```
FILE_PATH = "/content/electricityLoadData - Q3.csv"
WINDOW = 48
```

Load and preprocess data

```
def load_data(file_path):
    df = pd.read_csv(file_path, header=1, error_bad_lines=False)
    df.drop(df.columns[[2]], axis=1, inplace=True)
    return df


def normalize_data(dataset):
    values = dataset.values
    minima_demand = np.amin(values[:, -1])
    maxima_demand = np.amax(values[:, -1])
    scaling_parameter_demand = maxima_demand - minima_demand
    for i in range(values.shape[1]):
        values[:, i] = (values[:, i]-np.amin(values[:, i]))/(np.amax(values[:, i])-np.amin(values[:, i]))
    return minima_demand, maxima_demand, scaling_parameter_demand, pd.DataFrame(values)


def prepare_data(dataset, window_size):
    amount_of_features = len(dataset.columns)
    data = dataset.values
    sequence_length = window_size + 1
    result = []
    for index in range(len(data) - sequence_length):
        result.append(data[index: index + sequence_length])
    windowed_mat = np.array(result)

    train_split = int(round(0.8 * windowed_mat.shape[0]))
    x_train = windowed_mat[:train_split, :-1]
```

```
    y_train = windowed_mat[:train_split, -1][:,-1]
    x_test = windowed_mat[train_split:, :-1]
    y_test = windowed_mat[train_split:, -1][:,-1]
    x_train = np.reshape(x_train, (x_train.shape[0], x_train.shape[1], amount_of_features))
    x_test = np.reshape(x_test, (x_test.shape[0], x_test.shape[1], amount_of_features))
    return x_train, y_train, x_test, y_test


dataset = load_data(FILE_PATH)
min_demand, max_demand, demand_scaling_param, dataset = normalize_data(dataset)
x_train, y_train, x_test, y_test = prepare_data(dataset[::-1], WINDOW)
```

```
<ipython-input-5-568ddd437e24>:2: FutureWarning:

The error_bad_lines argument has been deprecated and will be removed in a future version. Use on_bad_lines in the future.
```

Define and train model

```
model = Sequential()
model.add(LSTM(256, activation='relu', input_shape=(48,5)))
model.add(Dense(128))
model.add(Dense(64))
model.add(Dense(32))
model.add(Dense(1))
```

```
WARNING:tensorflow:Layer lstm_1 will not use cuDNN kernels since it doesn't meet the criteria. It will use a generic GPU kernel as fall
```

```
model.compile(optimizer='adam', loss='mse')
```

```
model.summary()
```

```
Model: "sequential_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 lstm_1 (LSTM)               (None, 256)               268288

 dense_4 (Dense)             (None, 128)               32896

 dense_5 (Dense)             (None, 64)                8256

 dense_6 (Dense)             (None, 32)                2080

 dense_7 (Dense)             (None, 1)                 33

=================================================================
Total params: 311,553
Trainable params: 311,553
Non-trainable params: 0
_________________________________________________________________
```

```
model.fit(
    x_train,
    y_train,
    batch_size=128,
    epochs=15,
    validation_split=0.2,
    verbose=2)
```

```
Epoch 1/15
351/351 - 31s - loss: 0.0027 - val_loss: 0.0011 - 31s/epoch - 89ms/step
Epoch 2/15
351/351 - 28s - loss: 3.5377e-04 - val_loss: 2.8489e-04 - 28s/epoch - 79ms/step
Epoch 3/15
351/351 - 26s - loss: 1.9747e-04 - val_loss: 2.1317e-04 - 26s/epoch - 75ms/step
Epoch 4/15
351/351 - 26s - loss: 1.6798e-04 - val_loss: 1.7832e-04 - 26s/epoch - 75ms/step
Epoch 5/15
351/351 - 27s - loss: 1.5642e-04 - val_loss: 1.4995e-04 - 27s/epoch - 78ms/step
Epoch 6/15
351/351 - 28s - loss: 1.3955e-04 - val_loss: 3.9076e-04 - 28s/epoch - 79ms/step
Epoch 7/15
```

```
351/351 - 26s - loss: 1.2729e-04 - val_loss: 2.3933e-04 - 26s/epoch - 74ms/step
Epoch 8/15
351/351 - 28s - loss: 1.2534e-04 - val_loss: 1.0341e-04 - 28s/epoch - 79ms/step
Epoch 9/15
351/351 - 26s - loss: 1.2767e-04 - val_loss: 1.2603e-04 - 26s/epoch - 75ms/step
Epoch 10/15
351/351 - 26s - loss: 9.9502e-05 - val_loss: 1.6090e-04 - 26s/epoch - 75ms/step
Epoch 11/15
351/351 - 28s - loss: 1.0736e-04 - val_loss: 1.0997e-04 - 28s/epoch - 79ms/step
Epoch 12/15
351/351 - 26s - loss: 1.0032e-04 - val_loss: 1.1996e-04 - 26s/epoch - 74ms/step
Epoch 13/15
351/351 - 28s - loss: 9.3174e-05 - val_loss: 8.8033e-05 - 28s/epoch - 81ms/step
Epoch 14/15
351/351 - 26s - loss: 8.5377e-05 - val_loss: 1.0445e-04 - 26s/epoch - 75ms/step
Epoch 15/15
351/351 - 26s - loss: 8.7348e-05 - val_loss: 1.1130e-04 - 26s/epoch - 74ms/step
<keras.callbacks.History at 0x7f8168335340>
```

Get the predicted and actual data

```
def denormalize_data(data, scaling_parameter, minima):
    return (data * scaling_parameter) + minima
```

```
predicted_data = denormalize_data(model.predict(x_test), demand_scaling_param, min_demand)
actual_data = denormalize_data(y_test, demand_scaling_param, min_demand)
```

```
438/438 [==============================] - 3s 7ms/step
```

Calculate and print the mean absolute percentage error and mean absolute error

```
def calculate_mape(y_true, y_pred):
    return np.mean(np.abs((y_true - y_pred) / y_true)) * 100
```

```
predicted_data.shape
```

```
(14016, 1)
```

```
predicted_data = predicted_data.reshape((14016,))
mape = calculate_mape(actual_data, predicted_data)
mae = mean_absolute_error(actual_data, predicted_data)
```

```
print(f'Test MAPE: {mape:.3f}')
print(f'Test MAE: {mae:.3f}')
```

```
Test MAPE: 0.877
Test MAE: 483.143
```

Plot the forecast using Plotly

```
def plot_forecast(actual, predicted, time_window=24):
    import plotly.graph_objs as go

    actual_trace = go.Scatter(x=list(range(time_window)),
                              y=actual[:time_window],
                              mode='lines',
                              name='Actual',
                              line=dict(color='blue'))

    predicted_trace = go.Scatter(x=list(range(time_window)),
                                 y=predicted[:time_window],
                                 mode='lines',
                                 name='Predicted',
                                 line=dict(color='red'))

    layout = go.Layout(title=f'Actual vs Predicted Results in the Next {time_window * 1800} Seconds',
                       xaxis=dict(title='Time Sequence'),
                       yaxis=dict(title='Load (MW)'))
```

```
                    yaxis=dict(title='Load (MW)'),
                    legend=dict(x=0, y=1))

    fig = go.Figure(data=[actual_trace, predicted_trace], layout=layout)
    fig.show()


plot_forecast(actual_data, predicted_data)
```