Anomaly Detection using Autoencoder

We are going perform unsupervised anomaly detection on the voltage Time series. We are going to train our model in our normal dataset and then we will test it on the given abnormal dataset to test its ability to spot anomalies. The autoencoder will try to reproduce the given input, but since it is trained on the normal dataset, the reconstruction error of the abnormal samples shouldbe larger that the error on the normal dataset.So by setting a threshold on this reconstruction error we can detect outliers on anomalies.

In [1]:

```
import pandas as pd
from pathlib import Path
import numpy as np
import tensorflow as tf
from tensorflow.keras.layers import *
from tensorflow.keras.models import Model
import seaborn as sns
from pylab import rcParams
from sklearn.utils import shuffle
from pdb import set_trace
import matplotlib.pylab as plt
import sys
from sklearn.preprocessing import MinMaxScaler
from sklearn.model_selection import KFold, StratifiedKFold
from sklearn.model_selection import cross_val_score
from tensorflow.keras.wrappers.scikit_learn import KerasClassifier
from sklearn.model_selection import cross_val_predict
from sklearn.metrics import accuracy_score, confusion_matrix, recall_score, precision_score, roc
_auc_score, roc_curve
from sklearn.utils.multiclass import unique_labels
```

In [2]:

```
sys.path.append('/home/aggelos-i3/ForecastingLib/')
from tsutils import SequenceSpliter
from utils import utilities
```

In [3]:

```
rcParams['figure.figsize'] = 18, 10
sns.set()
```

Preprocessing

In this step we chose the features our model is going to use. To reduce noise we use a moving average of the ten most recent samples, smoothing our sequences making it easier for our models to learn the underlying features.

In [4]:

```
LOOKBACK = 100
LOOK_AHEAD = 1
ROLL_WINDOW = 10
features=['voltage [V]',
          'acceleration (actual) [m/(s*s)]',
          'tractive effort (actual) [kN]',
           'track-earth voltage [V]',
          'speed (actual) [km/h]',
          'current [A]',
          'energy balance [kWh]',
          'way (actual) [km]',
          'line and running resistance [kN]',
          'train configuration [1]',
          'energy input [kWh]',
          'train configuration [1]',
          'usable braking energy [kWh]',
          'used braking energy [kWh]'
         ]
nb_features = len(features)
```

In [5]:

```
df_new = pd.DataFrame()
s = 0.
pathlist = Path("/home/aggelos-i3/Downloads/simu Elbas/7h33NO").glob(
    '**/*.xls')
for path in pathlist:
    path_in_str = str(path)
    df = pd.read_csv(path_in_str, delimiter='\t', usecols=features)
    if df_new.empty:
        df_new = df[:1000]
    else:
        df_new += df[:1000]
    s += 1.

df_new /= s
df_new = df_new.rolling(window=ROLL_WINDOW).mean().dropna()
```

Therefore we have a new smoothed dataset of the features we chose. We standarize our dataset featurewise, so that every column has a mean of 1 and a variation of 0.

In [6]:

```
from sklearn.preprocessing import MinMaxScaler, StandardScaler
scaler = StandardScaler()
y_idx = df_new.columns.get_loc('voltage [V]') #which series we try to replicate
scaler = scaler.fit(df_new)
df_scaled = scaler.transform(df_new)
spliter = SequenceSpliter(lookback=LOOKBACK, look_ahead=LOOK_AHEAD)
```

We split our timeseries into sequences of 100 samples with one step difference. And we select the feature we want our autoencoder to reproduce, in our case the **Voltage**.

In [7]:

```
X, y = spliter.fit_transform(df_scaled)
y = y[:, :, y_idx]
```

The choice of the preprocessing hyperparameters, like the window for the smoothing and the splitting window, is empirical and based on trial and error.

We are going to implement and test 2 neural network autoencoders. One using recurrent cells (GRUs or LSTMs) and the second one is a convolutional autoencoder.

Reccurent Autoencoder

Here we define and train our reccurent autoencoder.

In [8]:

```
inputs = Input(shape=X.shape[1:])
encoder = GRU(LOOKBACK, return_sequences=True)(inputs)
encoder = Dropout(0.7)(encoder)
encoder = GRU(32, return_sequences=True)(encoder)
encoder = Dropout(0.7)(encoder)
encoder = GRU(4, return_sequences=True)(encoder)
decoder = GRU(32, return_sequences=True)(encoder)
decoder = Dropout(0.7)(decoder)
out = GRU(LOOKBACK, return_sequences=False)(decoder)
LSTM_AE = Model(inputs, out)
LSTM_AE.compile(loss='mse', optimizer='adam')
```

In [9]:

```
LSTM_AE.summary()
```

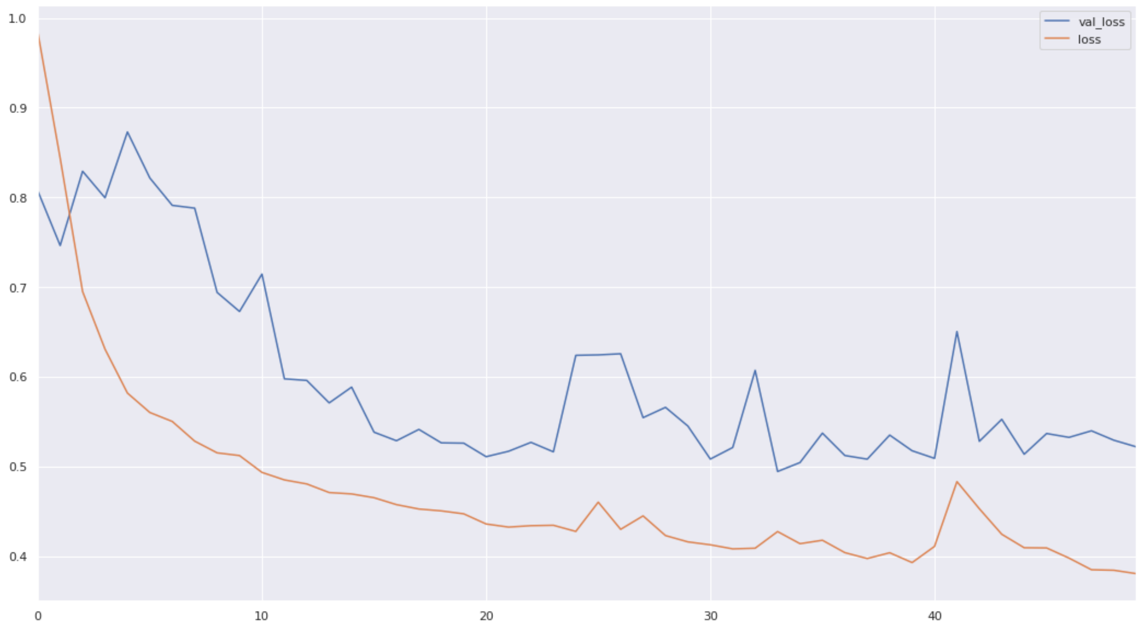
```
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
input_1 (InputLayer)         (None, 100, 13)           0         
_________________________________________________________________
gru (GRU)                    (None, 100, 100)          34200     
_________________________________________________________________
dropout (Dropout)            (None, 100, 100)          0         
_________________________________________________________________
gru_1 (GRU)                  (None, 100, 32)           12768     
_________________________________________________________________
dropout_1 (Dropout)          (None, 100, 32)           0         
_________________________________________________________________
gru_2 (GRU)                  (None, 100, 4)            444       
_________________________________________________________________
gru_3 (GRU)                  (None, 100, 32)           3552      
_________________________________________________________________
dropout_2 (Dropout)          (None, 100, 32)           0         
_________________________________________________________________
gru_4 (GRU)                  (None, 100)               39900     
=================================================================
Total params: 90,864
Trainable params: 90,864
Non-trainable params: 0
_________________________________________________________________
```

In [10]:

```
history = LSTM_AE.fit(X, X[:,:,y_idx], epochs=50, validation_split=0.1, verbose=0)
history = pd.DataFrame(history.history)
history.plot()
```

Out[10]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x7fa820590898>
```



In [11]:

```
ERROR_TYPE = [1, 2, 3] #we can hav different labels for each error or the same label?
max_range = 800
min_range = 0
DATASET_IDX = 0

def model_test(model, min_range, dataset_idx):
    i=1
    for error_type in ERROR_TYPE:
        df_faulty = pd.DataFrame()
        s = 0.
        pathlist = Path(f"/home/aggelos-i3/Downloads/simu Elbas/7h33D{error_type}").glob(
            '**/*.xls')

        for path in pathlist:
            path_in_str = str(path)
            df = pd.read_csv(path_in_str, delimiter='\t', usecols=features)
            if df_faulty.empty:
                df_faulty = df[:1000]
            else:
                df_faulty += df[:1000]
            s += 1.

        df_faulty /= s
        df_faulty = df_faulty.rolling(window=ROLL_WINDOW).mean().dropna()
        y_idx = df_faulty.columns.get_loc('voltage [V]')
        df_scaled_faulty = scaler.fit_transform(df_faulty)
        X_test, y_test = spliter.fit_transform(df_scaled_faulty)
        #X_test = np.delete(X_test, y_idx, 2)
        y_test = y_test[:, :, y_idx]

        yhat = model.predict(X_test)
        #yhat = yhat.reshape(X_test.shape)
        mse = np.mean(np.power(yhat-X_test[:,:,y_idx],2), axis=1)
        df_error = pd.DataFrame({'reconstruction_error': mse,
                                 'Label': y_test[:,0]})
        df_error = df_error[:max_range]
        #df_error['reconstruction_error'].plot()

        threshold = mse.mean() + mse.std()
        anomaly = mse[min_range:max_range] > threshold
        plt.subplot(3,2,i)
        plt.title(f"Error Type {error_type}")
        plt.plot(df_error['reconstruction_error'][min_range:], label='Reconstruction Error')

        plt.subplot(3,2,i+1)
        plt.scatter(range(min_range,max_range), np.where(anomaly, y_test[min_range:max_range, 0
], None), c='r', label='Outliers')
        plt.plot(range(min_range,max_range), y_test[min_range:max_range,0], label='Abnormal Volt
age')
        plt.plot(range(min_range,max_range), y[min_range:max_range,0], label='Normal Voltage')
        #plt.plot(range(max_range), yhat[:max_range,0], label='Predicted Voltage')

        plt.legend(loc='best')
        i+=2
```

Convolutional Autoencoder

In [12]:

```
inputs = Input(shape=X.shape[1:])
encoder = Conv1D(100, 50, padding='same')(inputs)
encoder = BatchNormalization()(encoder)
encoder = Activation('relu')(encoder)
encoder = Conv1D(16, 25, padding='same')(encoder)
encoder = BatchNormalization()(encoder)
encoder = Activation('relu')(encoder)
encoder = Conv1D(8, 15, padding='same')(encoder)
encoder = BatchNormalization()(encoder)
encoder = Activation('relu')(encoder)
decoder = Conv1D(16, 25, padding='same')(encoder)
decoder = BatchNormalization()(decoder)
decoder = Activation('relu')(decoder)
decoder = Conv1D(100, 50, padding='same')(decoder)
out = GlobalAveragePooling1D()(decoder)
#out = Dense(LOOKBACK)(decoder)
FCN_AE = Model(inputs, out)
FCN_AE.compile(loss='mse', optimizer='adam')
FCN_AE.summary()
```

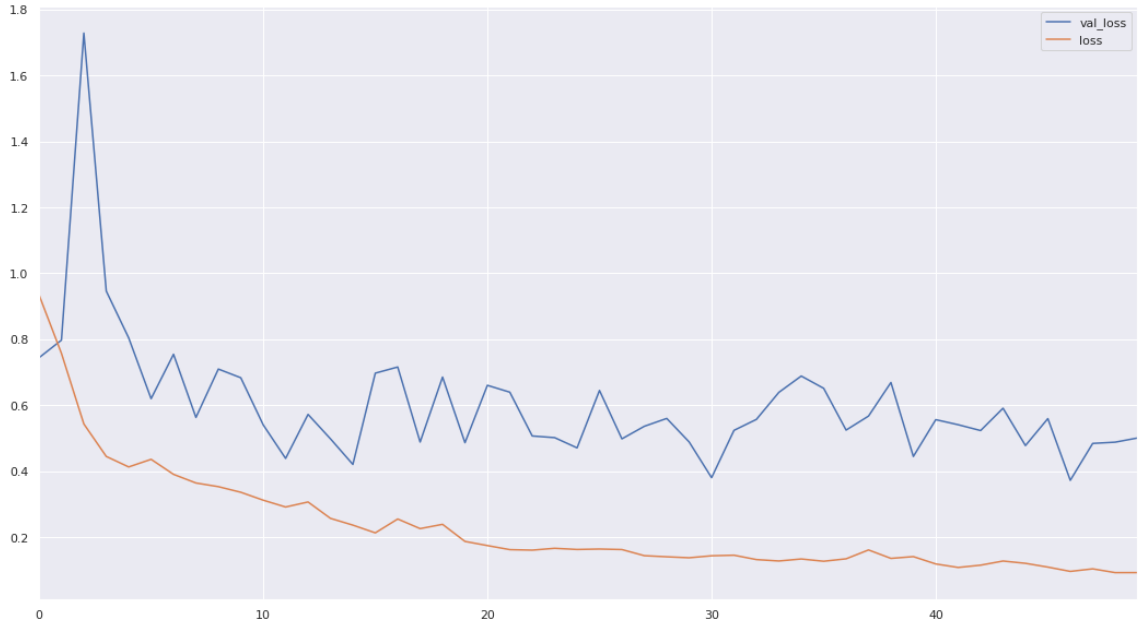
```
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
input_2 (InputLayer)         (None, 100, 13)           0         
_________________________________________________________________
conv1d (Conv1D)              (None, 100, 100)          65100     
_________________________________________________________________
batch_normalization (BatchNo (None, 100, 100)          400       
_________________________________________________________________
activation (Activation)      (None, 100, 100)          0         
_________________________________________________________________
conv1d_1 (Conv1D)            (None, 100, 16)           40016     
_________________________________________________________________
batch_normalization_1 (Batch (None, 100, 16)           64        
_________________________________________________________________
activation_1 (Activation)    (None, 100, 16)           0         
_________________________________________________________________
conv1d_2 (Conv1D)            (None, 100, 8)            1928      
_________________________________________________________________
batch_normalization_2 (Batch (None, 100, 8)            32        
_________________________________________________________________
activation_2 (Activation)    (None, 100, 8)            0         
_________________________________________________________________
conv1d_3 (Conv1D)            (None, 100, 16)           3216      
_________________________________________________________________
batch_normalization_3 (Batch (None, 100, 16)           64        
_________________________________________________________________
activation_3 (Activation)    (None, 100, 16)           0         
_________________________________________________________________
conv1d_4 (Conv1D)            (None, 100, 100)          80100     
_________________________________________________________________
global_average_pooling1d (Gl (None, 100)               0         
=================================================================
Total params: 190,920
Trainable params: 190,640
Non-trainable params: 280
_________________________________________________________________
```

In [13]:

```
history = FCN_AE.fit(X, X[:,:,y_idx], epochs=50, validation_split=0.1, verbose=0)
history = pd.DataFrame(history.history)
history.plot()
```

Out[13]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x7fa7ecf9a400>
```



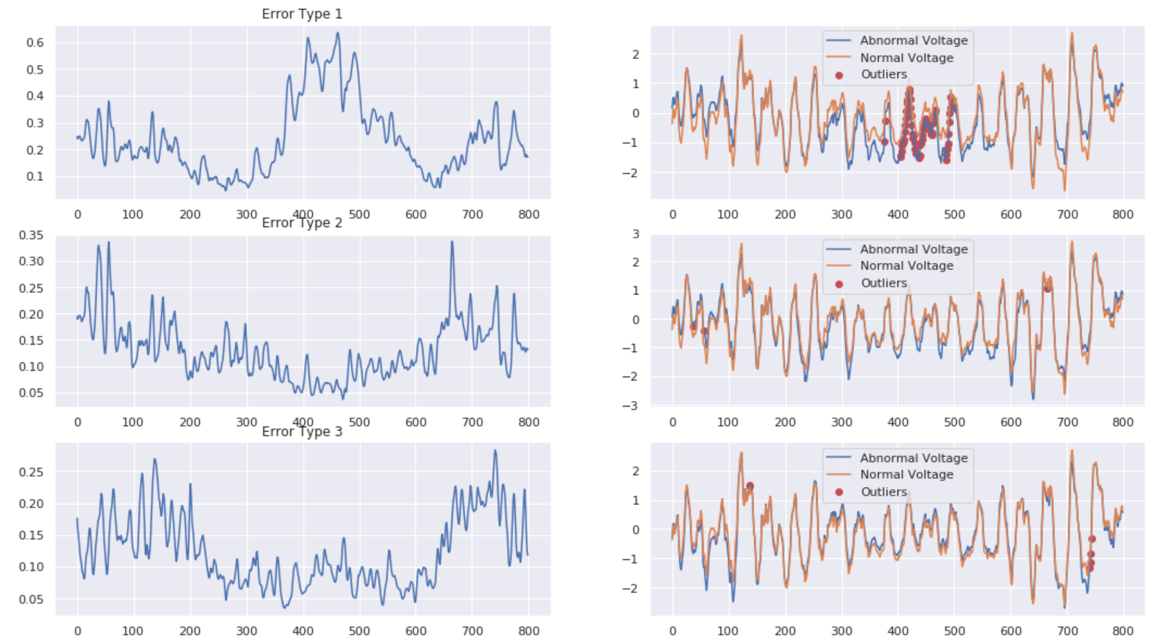
Experiments

As a reconstruction error we use the mean squaered error of our predictions. Below we plot the reconstruction error for each type of simulated error and the detected anomalies.

In the example tests the convolutional autoencoder showed superior training time.

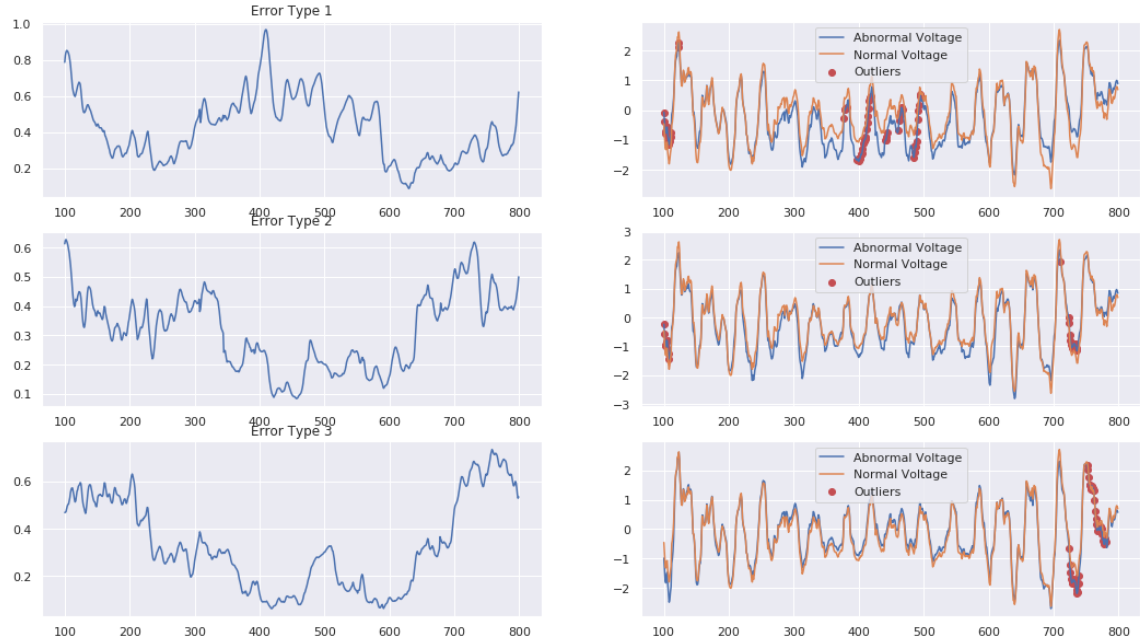
In [16]:

```
model_test(FCN_AE, min_range=0, dataset_idx=1)
```



In [17]:

```
model_test(LSTM_AE, min_range=100, dataset_idx=1)
```



In [ ]:

```
```