Part II: Machine Learning

## Google Colab Setup

Since training the models is quite expensive and time consuming on regular CPUs, we moved the training process to Google Colab using GPUs.

from google.colab import drive  
drive.mount('/content/drive/')

Mounted at /content/drive/

First, we installed all requirements of the project without nilmtk and nilm-metadata.

!pip install -r ./drive/MyDrive/Energy/req\_all\_but\_nilmtk.txt

Next, we cloned the repos of nilmtk and nilm\_metadata from GitHub to install the packages from the folder as editables. The reason behind this is that the time consuming installation of nilmtk wastes “computing units” without doing any meaningful computations.

!pip install -e ./drive/MyDrive/Energy/packages/nilm\_metadata

For nilmtk, we removed the outdated pins on numpy, pandas, matplotlib and networkx within setup.py to fasten the installation process from 60 minutes down to a few minutes, which saves “computing units” on Colab. Most of the time was used to build a wheel of the outdated pandas=0.25.3.

# nilmtk > setup.py  
setup(  
 # ...  
 install\_requires=[  
 "pandas", #"pandas==0.25.3",  
 "numpy", # "numpy >= 1.13.3, < 1.20.0",  
 "networkx", #"networkx==2.1",  
 "scipy",  
 "tables",  
 "scikit-learn>=0.21.2",  
 "hmmlearn>=0.2.1",  
 "pyyaml",  
 "matplotlib", #"matplotlib==3.1.3",  
 "jupyterlab"  
 ],  
 # ...  
)

# Run the line below twice to get nilmtk installed  
!pip install -e ./drive/MyDrive/Energy/packages/nilmtk  
!pip install -e ./drive/MyDrive/Energy/packages/nilmtk

After the installation, *restart the Colab Runtime*.

Moreover, we have a Python version mismatch on Colab. nilmtk wants us to have python=3.8, but Colab uses python=3.9+. The networkx package might lead to problems, when dag.py is called, since gcd moved:

# networkx > dag.py  
# ...  
from fractions import gcd # for python 3.8  
from math import gcd # for python 3.9+  
# ...

An update to dag.py is necessary, if the code below shows an error. *Restart the Colab Runtime afterwards.*

# shouldn't error if the steps above are followed  
from nilmtk import DataSet

## NILMTK API

It’s possible to use custom hand-crafted deep learning models and training procedures. However, nilmtk comes pre-packaged with a quite useful model training API, which we are going to use.

# Check if Google Drive is used  
from pathlib import Path  
gdrive = Path("./drive/MyDrive/Energy/data").exists()

# Load Data  
from nilmtk import DataSet  
  
data\_path = "./drive/MyDrive/Energy/data" if gdrive else "./data"  
file\_path = f"{data\_path}/REFIT.h5" # google drive  
refit = DataSet(file\_path)

# Helper function  
def ndir(x):  
 """ Show properties and methods with no magic methods """  
 return [x for x in dir(x) if not x.\_\_contains\_\_("\_\_")]

# Load API and joblib (more efficient pickle replacement)  
from nilmtk.api import API  
import joblib  
import matplotlib.pyplot as plt  
  
import warnings  
warnings.filterwarnings("ignore")

### Models

Unfortunately, we have failed to install nilmtk-contrib, most likely due to a versioning problem of nilmtk. As a work-around, we have copied the three models seq2point, seq2seq and BERT verbatim from the GitHub repository. In the BERT code, we made slight adjustments to the import statements of keras, to reflect our newer version of tensorflow.

Also, we added a learning rate parameter for the Adam optimizer, to see if variations of the learning rate led to better results.

Looking at the code of the models, we see that the models take care of any pre-processing themselves.

# bert.py  
"""  
This code is copied verbatim from the nitlmk-contrib repo  
https://github.com/nilmtk/nilmtk-contrib/blob/master/nilmtk\_contrib/disaggregate/bert.py  
  
LICENCE: Apache License 2.0  
  
-- Changes made --  
  
\* Due to the error:  
 AttributeError: module 'tensorflow.compat.v2.\_\_internal\_\_' has no attribute 'dispatch'  
 => We switched the imports `from keras` to `from tensorflow.keras`  
  
\* Changed the variable file\_path for the weights to reflect appliance name  
  
\* Added learning rate parameter, updated optimizer in model.compile()  
"""  
  
from \_\_future\_\_ import print\_function, division  
from warnings import warn  
  
from nilmtk.disaggregate import Disaggregator  
from tensorflow.keras.layers import Conv1D, Dense, Dropout, Reshape  
from tensorflow.keras.layers import Flatten,Input,GlobalAveragePooling1D, AveragePooling1D  
import os  
import pandas as pd  
import numpy as np  
import pickle  
from collections import OrderedDict  
  
from tensorflow.keras.optimizers import SGD  
from tensorflow.keras.models import Sequential, load\_model  
from tensorflow.keras.layers import Layer,MultiHeadAttention,LayerNormalization,Embedding  
import matplotlib.pyplot as plt  
from sklearn.model\_selection import train\_test\_split  
from tensorflow.keras.callbacks import ModelCheckpoint  
import tensorflow.keras.backend as K  
import random  
random.seed(10)  
np.random.seed(10)  
import tensorflow as tf  
gpus=tf.config.experimental.list\_physical\_devices("GPU")  
for gpu in gpus:  
 tf.config.experimental.set\_memory\_growth(gpu,True)  
  
class SequenceLengthError(Exception):  
 pass  
  
class ApplianceNotFoundError(Exception):  
 pass  
  
#This code is inspired from :  
# https://github.com/keras-team/keras-io/blob/master/examples/nlp/text\_classification\_with\_transformer.py  
  
class TransformerBlock(Layer):  
 def \_\_init\_\_(self, embed\_dim, num\_heads, ff\_dim, rate=0.1):  
 super(TransformerBlock, self).\_\_init\_\_()  
 self.att = MultiHeadAttention(num\_heads=num\_heads, key\_dim=embed\_dim)  
 self.ffn = Sequential(  
 [Dense(ff\_dim, activation="relu"), Dense(embed\_dim),]  
 )  
 self.layernorm1 = LayerNormalization(epsilon=1e-6)  
 self.layernorm2 = LayerNormalization(epsilon=1e-6)  
 self.dropout1 = Dropout(rate)  
 self.dropout2 = Dropout(rate)  
  
 def call(self, inputs, training):  
 attn\_output,att\_weights = self.att(inputs, inputs,return\_attention\_scores=True)  
 attn\_output = self.dropout1(attn\_output, training=training)  
 out1 = self.layernorm1(inputs + attn\_output)  
 ffn\_output = self.ffn(out1)  
 ffn\_output = self.dropout2(ffn\_output, training=training)  
 return self.layernorm2(out1 + ffn\_output)  
  
 def get\_config(self):  
 config = super().get\_config().copy()  
 config.update({  
 'att' : self.att,  
 'ffn' : self.ffn,  
 'layernorm1': self.layernorm1,  
 'layernorm2': self.layernorm2,  
 'dropout1': self.dropout1,  
 'dropout2': self.dropout2,  
 })  
 return config  
  
class TokenAndPositionEmbedding(Layer):  
 def \_\_init\_\_(self, maxlen, vocab\_size, embed\_dim):  
 super(TokenAndPositionEmbedding, self).\_\_init\_\_()  
 self.token\_emb = Embedding(input\_dim=vocab\_size, output\_dim=embed\_dim)  
 self.pos\_emb = Embedding(input\_dim=maxlen, output\_dim=embed\_dim)  
  
 def call(self, x):  
 maxlen = tf.shape(x)[-1]  
 positions = tf.range(start=0, limit=maxlen, delta=1)  
 positions = self.pos\_emb(positions)  
 x = self.token\_emb(x)  
 return x + positions  
  
 def get\_config(self):  
 config = super().get\_config().copy()  
 config.update({  
 'token\_emb' : self.token\_emb,  
 'pos\_emb' : self.pos\_emb,  
 })  
 return config   
  
  
class LPpool(Layer):  
 def \_\_init\_\_(self, pool\_size, strides=None, padding='same'):  
 super(LPpool,self).\_\_init\_\_()  
 self.avgpool=tf.keras.layers.AveragePooling1D(pool\_size,strides,padding)  
  
 def call(self, x):  
 x = tf.math.pow(tf.math.abs(x), 2)  
 x = self.avgpool(x)  
 x = tf.math.pow(x, 1.0 / 2)  
 return x  
  
 def get\_config(self):  
 config = super().get\_config().copy()  
 config.update({  
 'avgpool' : self.avgpool,  
 })  
 return config  
  
class BERT(Disaggregator):  
  
 def \_\_init\_\_(self, params):  
  
 self.MODEL\_NAME = "BERT"  
 self.chunk\_wise\_training = params.get('chunk\_wise\_training',False)  
 self.sequence\_length = params.get('sequence\_length',99)  
 self.n\_epochs = params.get('n\_epochs', 10)  
 self.models = OrderedDict()  
 self.mains\_mean = 1800  
 self.mains\_std = 600  
 self.batch\_size = params.get('batch\_size',512)  
 self.appliance\_params = params.get('appliance\_params',{})  
 if self.sequence\_length%2==0:  
 print ("Sequence length should be odd!")  
 raise (SequenceLengthError)  
 self.learning\_rate = 0.001 # added  
  
 def partial\_fit(self,train\_main,train\_appliances,do\_preprocessing=True,\*\*load\_kwargs):  
  
 print("...............BERT partial\_fit running...............")  
 if len(self.appliance\_params) == 0:  
 self.set\_appliance\_params(train\_appliances)  
  
 if do\_preprocessing:  
 train\_main, train\_appliances = self.call\_preprocessing(  
 train\_main, train\_appliances, 'train')  
 train\_main = pd.concat(train\_main,axis=0)  
 train\_main = train\_main.values.reshape((-1,self.sequence\_length,1))  
   
 new\_train\_appliances = []  
 for app\_name, app\_dfs in train\_appliances:  
 app\_df = pd.concat(app\_dfs,axis=0)  
 app\_df\_values = app\_df.values.reshape((-1,self.sequence\_length))  
 new\_train\_appliances.append((app\_name, app\_df\_values))  
 train\_appliances = new\_train\_appliances  
  
 for appliance\_name, power in train\_appliances:  
 if appliance\_name not in self.models:  
 print("First model training for ", appliance\_name)  
 self.models[appliance\_name] = self.return\_network()  
 else:  
 print("Started Retraining model for ", appliance\_name)  
  
 model = self.models[appliance\_name]  
 if train\_main.size > 0:  
 # Sometimes chunks can be empty after dropping NANS  
 if len(train\_main) > 10:  
 # Do validation when you have sufficient samples  
 filepath = 'BERT-'+"\_".join(appliance\_name.split())+'.h5' # change  
 checkpoint = ModelCheckpoint(filepath,monitor='val\_loss',verbose=1,save\_best\_only=True,mode='min')  
 train\_x, v\_x, train\_y, v\_y = train\_test\_split(train\_main, power, test\_size=.15,random\_state=10)  
 model.fit(train\_x,train\_y,validation\_data=(v\_x,v\_y),epochs=self.n\_epochs,callbacks=[checkpoint],batch\_size=self.batch\_size)  
 model.load\_weights(filepath)  
  
 def disaggregate\_chunk(self,test\_main\_list,model=None,do\_preprocessing=True):  
  
 if model is not None:  
 self.models = model  
  
 if do\_preprocessing:  
 test\_main\_list = self.call\_preprocessing(  
 test\_main\_list, submeters\_lst=None, method='test')  
  
 test\_predictions = []  
 for test\_mains\_df in test\_main\_list:  
  
 disggregation\_dict = {}  
 test\_main\_array = test\_mains\_df.values.reshape((-1, self.sequence\_length, 1))  
  
 for appliance in self.models:  
  
 prediction = []  
 model = self.models[appliance]  
 prediction = model.predict(test\_main\_array ,batch\_size=self.batch\_size)  
  
 #####################  
 # This block is for creating the average of predictions over the different sequences  
 # the counts\_arr keeps the number of times a particular timestamp has occured  
 # the sum\_arr keeps the number of times a particular timestamp has occured  
 # the predictions are summed for agiven time, and is divided by the number of times it has occured  
   
 l = self.sequence\_length  
 n = len(prediction) + l - 1  
 sum\_arr = np.zeros((n))  
 counts\_arr = np.zeros((n))  
 o = len(sum\_arr)  
 for i in range(len(prediction)):  
 sum\_arr[i:i + l] += prediction[i].flatten()  
 counts\_arr[i:i + l] += 1  
 for i in range(len(sum\_arr)):  
 sum\_arr[i] = sum\_arr[i] / counts\_arr[i]  
  
 #################  
 prediction = self.appliance\_params[appliance]['mean'] + (sum\_arr \* self.appliance\_params[appliance]['std'])  
 valid\_predictions = prediction.flatten()  
 valid\_predictions = np.where(valid\_predictions > 0, valid\_predictions, 0)  
 df = pd.Series(valid\_predictions)  
 disggregation\_dict[appliance] = df  
 results = pd.DataFrame(disggregation\_dict, dtype='float32')  
 test\_predictions.append(results)  
 return test\_predictions  
  
 def return\_network(self):  
 '''Creates the BERT module  
 '''  
 embed\_dim = 32 # Embedding size for each token  
 num\_heads = 2 # Number of attention heads  
 ff\_dim = 32 # Hidden layer size in feed forward network inside transformer  
 vocab\_size = 20000 #vocab for different patterns in reading  
 maxlen = self.sequence\_length #maxlength for attention  
  
 model = Sequential()  
 model.add(Conv1D(16,4,activation="linear",input\_shape=(self.sequence\_length,1),padding="same",strides=1))  
 model.add(LPpool(pool\_size=2))  
  
 #Token and Positional embedding and Encoder part of the transformer  
 model.add(TokenAndPositionEmbedding(maxlen, vocab\_size, embed\_dim))  
 model.add(TransformerBlock(embed\_dim, num\_heads, ff\_dim))  
  
 #Fully connected layer  
 model.add(Flatten())  
 model.add(Dropout(0.1))  
 model.add(Dense(self.sequence\_length))  
 model.add(Dropout(0.1))  
 model.summary()  
 model.compile(loss='mse',   
 optimizer=tf.keras.optimizers.Adam(self.learning\_rate), # changed  
 metrics=['mse'])  
 return model  
  
 def call\_preprocessing(self, mains\_lst, submeters\_lst, method):  
  
 if method == 'train':   
 processed\_mains\_lst = []  
 for mains in mains\_lst:  
 new\_mains = mains.values.flatten()  
 n = self.sequence\_length  
 units\_to\_pad = n // 2  
 new\_mains = np.pad(new\_mains, (units\_to\_pad,units\_to\_pad),'constant',constant\_values = (0,0))  
 new\_mains = np.array([new\_mains[i:i + n] for i in range(len(new\_mains) - n + 1)])  
 new\_mains = (new\_mains - self.mains\_mean) / self.mains\_std  
 processed\_mains\_lst.append(pd.DataFrame(new\_mains))  
 appliance\_list = []  
 for app\_index, (app\_name, app\_df\_lst) in enumerate(submeters\_lst):  
  
 if app\_name in self.appliance\_params:  
 app\_mean = self.appliance\_params[app\_name]['mean']  
 app\_std = self.appliance\_params[app\_name]['std']  
 else:  
 print ("Parameters for ", app\_name ," were not found!")  
 raise ApplianceNotFoundError()  
  
  
 processed\_app\_dfs = []  
 for app\_df in app\_df\_lst:   
 new\_app\_readings = app\_df.values.flatten()  
 new\_app\_readings = np.pad(new\_app\_readings, (units\_to\_pad,units\_to\_pad),'constant',constant\_values = (0,0))  
 new\_app\_readings = np.array([new\_app\_readings[i:i + n] for i in range(len(new\_app\_readings) - n + 1)])   
 new\_app\_readings = (new\_app\_readings - app\_mean) / app\_std # /self.max\_val  
 processed\_app\_dfs.append(pd.DataFrame(new\_app\_readings))  
   
   
 appliance\_list.append((app\_name, processed\_app\_dfs))  
  
  
 return processed\_mains\_lst, appliance\_list  
  
 else:  
 processed\_mains\_lst = []  
 for mains in mains\_lst:  
 new\_mains = mains.values.flatten()  
 n = self.sequence\_length  
 units\_to\_pad = n // 2  
 #new\_mains = np.pad(new\_mains, (units\_to\_pad,units\_to\_pad),'constant',constant\_values = (0,0))  
 new\_mains = np.array([new\_mains[i:i + n] for i in range(len(new\_mains) - n + 1)])  
 new\_mains = (new\_mains - self.mains\_mean) / self.mains\_std  
 new\_mains = new\_mains.reshape((-1, self.sequence\_length))  
 processed\_mains\_lst.append(pd.DataFrame(new\_mains))  
 return processed\_mains\_lst  
  
 def set\_appliance\_params(self,train\_appliances):  
  
 for (app\_name,df\_list) in train\_appliances:  
 l = np.array(pd.concat(df\_list,axis=0))  
 app\_mean = np.mean(l)  
 app\_std = np.std(l)  
 if app\_std<1:  
 app\_std = 100  
 self.appliance\_params.update({app\_name:{'mean':app\_mean,'std':app\_std}})

# seq2seq.py  
"""  
This code is copied verbatim from the nitlmk-contrib repo  
https://github.com/nilmtk/nilmtk-contrib/blob/master/nilmtk\_contrib/disaggregate/seq2seq.py  
  
LICENCE: Apache License 2.0  
  
-- Changes made: --  
\* Added learning rate parameter, updated optimizer in model.compile()  
  
"""  
  
from collections import OrderedDict  
import numpy as np  
import pandas as pd  
from nilmtk.disaggregate import Disaggregator  
from tensorflow.keras.callbacks import ModelCheckpoint  
from tensorflow.keras.layers import Conv1D, Dense, Dropout, Flatten  
from tensorflow.keras.models import Sequential  
  
  
class SequenceLengthError(Exception):  
 pass  
  
class ApplianceNotFoundError(Exception):  
 pass  
  
  
  
class Seq2Seq(Disaggregator):  
  
 def \_\_init\_\_(self, params):  
  
 self.MODEL\_NAME = "Seq2Seq"  
 self.file\_prefix = "{}-temp-weights".format(self.MODEL\_NAME.lower())  
 self.chunk\_wise\_training = params.get('chunk\_wise\_training',False)  
 self.sequence\_length = params.get('sequence\_length',99)  
 self.n\_epochs = params.get('n\_epochs', 10)  
 self.models = OrderedDict()  
 self.mains\_mean = 1800  
 self.mains\_std = 600  
 self.batch\_size = params.get('batch\_size',512)  
 self.appliance\_params = params.get('appliance\_params',{})  
 if self.sequence\_length%2==0:  
 print ("Sequence length should be odd!")  
 raise (SequenceLengthError)  
 self.learning\_rate = 0.001 # added   
  
 def partial\_fit(self, train\_main, train\_appliances, do\_preprocessing=True, current\_epoch=0, \*\*load\_kwargs):  
 print("...............Seq2Seq partial\_fit running...............")  
 if len(self.appliance\_params) == 0:  
 self.set\_appliance\_params(train\_appliances)  
  
 if do\_preprocessing:  
 train\_main, train\_appliances = self.call\_preprocessing(  
 train\_main, train\_appliances, 'train')  
  
 train\_main = pd.concat(train\_main, axis=0)  
 train\_main = train\_main.values.reshape((-1, self.sequence\_length, 1))  
 new\_train\_appliances = []  
 for app\_name, app\_dfs in train\_appliances:  
 app\_df = pd.concat(app\_dfs, axis=0)  
 app\_df\_values = app\_df.values.reshape((-1, self.sequence\_length))  
 new\_train\_appliances.append((app\_name, app\_df\_values))  
  
 train\_appliances = new\_train\_appliances  
 for appliance\_name, power in train\_appliances:  
 if appliance\_name not in self.models:  
 print("First model training for ", appliance\_name)  
 self.models[appliance\_name] = self.return\_network()  
 else:  
 print("Started Retraining model for ", appliance\_name)  
  
 model = self.models[appliance\_name]  
 if train\_main.size > 0:  
 # Sometimes chunks can be empty after dropping NANS  
 if len(train\_main) > 10:  
 # Do validation when you have sufficient samples  
 filepath = self.file\_prefix + "-{}-epoch{}.h5".format(  
 "\_".join(appliance\_name.split()),  
 current\_epoch,  
 )  
 checkpoint = ModelCheckpoint(filepath,monitor='val\_loss',verbose=1,save\_best\_only=True,mode='min')  
 model.fit(  
 train\_main, power,  
 validation\_split=.15,  
 epochs=self.n\_epochs,  
 batch\_size=self.batch\_size,  
 callbacks=[ checkpoint ],  
 )  
 model.load\_weights(filepath)  
  
   
 def disaggregate\_chunk(self,test\_main\_list,model=None,do\_preprocessing=True):  
 if model is not None:  
 self.models = model  
  
 if do\_preprocessing:  
 test\_main\_list = self.call\_preprocessing(  
 test\_main\_list, submeters\_lst=None, method='test')  
  
 test\_predictions = []  
 for test\_mains\_df in test\_main\_list:  
  
 disggregation\_dict = {}  
 test\_main\_array = test\_mains\_df.values.reshape((-1, self.sequence\_length, 1))  
  
 for appliance in self.models:  
  
 prediction = []  
 model = self.models[appliance]  
 prediction = model.predict(test\_main\_array ,batch\_size=self.batch\_size)  
  
 #####################  
 # This block is for creating the average of predictions over the different sequences  
 # the counts\_arr keeps the number of times a particular timestamp has occured  
 # the sum\_arr keeps the number of times a particular timestamp has occured  
 # the predictions are summed for agiven time, and is divided by the number of times it has occured  
   
 l = self.sequence\_length  
 n = len(prediction) + l - 1  
 sum\_arr = np.zeros((n))  
 counts\_arr = np.zeros((n))  
 o = len(sum\_arr)  
 for i in range(len(prediction)):  
 sum\_arr[i:i + l] += prediction[i].flatten()  
 counts\_arr[i:i + l] += 1  
 for i in range(len(sum\_arr)):  
 sum\_arr[i] = sum\_arr[i] / counts\_arr[i]  
  
 #################  
 prediction = self.appliance\_params[appliance]['mean'] + (sum\_arr \* self.appliance\_params[appliance]['std'])  
 valid\_predictions = prediction.flatten()  
 valid\_predictions = np.where(valid\_predictions > 0, valid\_predictions, 0)  
 df = pd.Series(valid\_predictions)  
 disggregation\_dict[appliance] = df  
 results = pd.DataFrame(disggregation\_dict, dtype='float32')  
 test\_predictions.append(results)  
  
 return test\_predictions  
  
 def return\_network(self):  
  
 model = Sequential()  
 # 1D Conv  
 model.add(Conv1D(30,10,activation="relu",input\_shape=(self.sequence\_length,1),strides=2))  
 model.add(Conv1D(30, 8, activation='relu', strides=2))  
 model.add(Conv1D(40, 6, activation='relu', strides=1))  
 model.add(Conv1D(50, 5, activation='relu', strides=1))  
 model.add(Dropout(.2))  
 model.add(Conv1D(50, 5, activation='relu', strides=1))  
 model.add(Dropout(.2))  
 model.add(Flatten())  
 model.add(Dense(1024, activation='relu'))  
 model.add(Dropout(.2))  
 model.add(Dense(self.sequence\_length))  
 model.compile(loss='mse', optimizer=tf.keras.optimizers.Adam(self.learning\_rate)) # changed  
  
 return model  
  
 def call\_preprocessing(self, mains\_lst, submeters\_lst, method):  
  
 if method == 'train':   
 processed\_mains\_lst = []  
 for mains in mains\_lst:  
 new\_mains = mains.values.flatten()  
 n = self.sequence\_length  
 units\_to\_pad = n // 2  
 new\_mains = np.pad(new\_mains, (units\_to\_pad,units\_to\_pad),'constant',constant\_values = (0,0))  
 new\_mains = np.array([new\_mains[i:i + n] for i in range(len(new\_mains) - n + 1)])  
 new\_mains = (new\_mains - self.mains\_mean) / self.mains\_std  
 processed\_mains\_lst.append(pd.DataFrame(new\_mains))  
 #new\_mains = pd.DataFrame(new\_mains)  
 appliance\_list = []  
 for app\_index, (app\_name, app\_df\_lst) in enumerate(submeters\_lst):  
  
 if app\_name in self.appliance\_params:  
 app\_mean = self.appliance\_params[app\_name]['mean']  
 app\_std = self.appliance\_params[app\_name]['std']  
 else:  
 print ("Parameters for ", app\_name ," were not found!")  
 raise ApplianceNotFoundError()  
  
  
 processed\_app\_dfs = []  
 for app\_df in app\_df\_lst:   
 new\_app\_readings = app\_df.values.flatten()  
 new\_app\_readings = np.pad(new\_app\_readings, (units\_to\_pad,units\_to\_pad),'constant',constant\_values = (0,0))  
 new\_app\_readings = np.array([new\_app\_readings[i:i + n] for i in range(len(new\_app\_readings) - n + 1)])   
 new\_app\_readings = (new\_app\_readings - app\_mean) / app\_std # /self.max\_val  
 processed\_app\_dfs.append(pd.DataFrame(new\_app\_readings))  
   
   
 appliance\_list.append((app\_name, processed\_app\_dfs))  
 #new\_app\_readings = np.array([ new\_app\_readings[i:i+n] for i in range(len(new\_app\_readings)-n+1) ])  
 #print (new\_mains.shape, new\_app\_readings.shape, app\_name)  
  
 return processed\_mains\_lst, appliance\_list  
  
 else:  
 processed\_mains\_lst = []  
 for mains in mains\_lst:  
 new\_mains = mains.values.flatten()  
 n = self.sequence\_length  
 units\_to\_pad = n // 2  
 #new\_mains = np.pad(new\_mains, (units\_to\_pad,units\_to\_pad),'constant',constant\_values = (0,0))  
 new\_mains = np.array([new\_mains[i:i + n] for i in range(len(new\_mains) - n + 1)])  
 new\_mains = (new\_mains - self.mains\_mean) / self.mains\_std  
 new\_mains = new\_mains.reshape((-1, self.sequence\_length))  
 processed\_mains\_lst.append(pd.DataFrame(new\_mains))  
 return processed\_mains\_lst  
  
 def set\_appliance\_params(self,train\_appliances):  
  
 for (app\_name,df\_list) in train\_appliances:  
 l = np.array(pd.concat(df\_list,axis=0))  
 app\_mean = np.mean(l)  
 app\_std = np.std(l)  
 if app\_std<1:  
 app\_std = 100  
 self.appliance\_params.update({app\_name:{'mean':app\_mean,'std':app\_std}})

# seq2point.py  
"""  
This code is copied verbatim from the nitlmk-contrib repo  
https://github.com/nilmtk/nilmtk-contrib/blob/master/nilmtk\_contrib/disaggregate/seq2point.py  
  
LICENCE: Apache License 2.0  
  
-- Changes made: --  
\* Added learning rate parameter, updated optimizer in model.compile()  
  
"""  
  
from collections import OrderedDict  
import numpy as np  
import pandas as pd  
from nilmtk.disaggregate import Disaggregator  
from tensorflow.keras.callbacks import ModelCheckpoint  
from tensorflow.keras.layers import Conv1D, Dense, Dropout, Reshape, Flatten  
from tensorflow.keras.models import Sequential  
  
  
class SequenceLengthError(Exception):  
 pass  
  
class ApplianceNotFoundError(Exception):  
 pass  
  
class Seq2Point(Disaggregator):  
  
 def \_\_init\_\_(self, params):  
 """  
 Parameters to be specified for the model  
 """  
  
 self.MODEL\_NAME = "Seq2Point"  
 self.models = OrderedDict()  
 self.file\_prefix = "{}-temp-weights".format(self.MODEL\_NAME.lower())  
 self.chunk\_wise\_training = params.get('chunk\_wise\_training',False)  
 self.sequence\_length = params.get('sequence\_length',99)  
 self.n\_epochs = params.get('n\_epochs', 10 )  
 self.batch\_size = params.get('batch\_size',512)  
 self.appliance\_params = params.get('appliance\_params',{})  
 self.mains\_mean = params.get('mains\_mean',1800)  
 self.mains\_std = params.get('mains\_std',600)  
 if self.sequence\_length%2==0:  
 print ("Sequence length should be odd!")  
 raise (SequenceLengthError)  
 self.learning\_rate = 0.001 # added  
  
 def partial\_fit(self, train\_main, train\_appliances, do\_preprocessing=True, current\_epoch=0, \*\*load\_kwargs):  
 # If no appliance wise parameters are provided, then copmute them using the first chunk  
 if len(self.appliance\_params) == 0:  
 self.set\_appliance\_params(train\_appliances)  
  
 print("...............Seq2Point partial\_fit running...............")  
 # Do the pre-processing, such as windowing and normalizing  
 if do\_preprocessing:  
 train\_main, train\_appliances = self.call\_preprocessing(  
 train\_main, train\_appliances, 'train')  
  
 train\_main = pd.concat(train\_main, axis=0)  
 train\_main = train\_main.values.reshape((-1, self.sequence\_length, 1))  
 new\_train\_appliances = []  
 for app\_name, app\_df in train\_appliances:  
 app\_df = pd.concat(app\_df, axis=0)  
 app\_df\_values = app\_df.values.reshape((-1, 1))  
 new\_train\_appliances.append((app\_name, app\_df\_values))  
 train\_appliances = new\_train\_appliances  
  
 for appliance\_name, power in train\_appliances:  
 # Check if the appliance was already trained. If not then create a new model for it  
 if appliance\_name not in self.models:  
 print("First model training for", appliance\_name)  
 self.models[appliance\_name] = self.return\_network()  
 # Retrain the particular appliance  
 else:  
 print("Started Retraining model for", appliance\_name)  
  
 model = self.models[appliance\_name]  
 if train\_main.size > 0:  
 # Sometimes chunks can be empty after dropping NANS  
 if len(train\_main) > 10:  
 # Do validation when you have sufficient samples  
 filepath = self.file\_prefix + "-{}-epoch{}.h5".format(  
 "\_".join(appliance\_name.split()),  
 current\_epoch,  
 )  
 checkpoint = ModelCheckpoint(filepath,monitor='val\_loss',verbose=1,save\_best\_only=True,mode='min')  
 model.fit(  
 train\_main, power,  
 validation\_split=0.15,  
 epochs=self.n\_epochs,  
 batch\_size=self.batch\_size,  
 callbacks=[checkpoint],  
 )  
 model.load\_weights(filepath)  
  
   
 def disaggregate\_chunk(self,test\_main\_list,model=None,do\_preprocessing=True):  
 if model is not None:  
 self.models = model  
  
 # Preprocess the test mains such as windowing and normalizing  
  
 if do\_preprocessing:  
 test\_main\_list = self.call\_preprocessing(test\_main\_list, submeters\_lst=None, method='test')  
  
 test\_predictions = []  
 for test\_main in test\_main\_list:  
 test\_main = test\_main.values  
 test\_main = test\_main.reshape((-1, self.sequence\_length, 1))  
 disggregation\_dict = {}  
 for appliance in self.models:  
 prediction = self.models[appliance].predict(test\_main,batch\_size=self.batch\_size)  
 prediction = self.appliance\_params[appliance]['mean'] + prediction \* self.appliance\_params[appliance]['std']  
 valid\_predictions = prediction.flatten()  
 valid\_predictions = np.where(valid\_predictions > 0, valid\_predictions, 0)  
 df = pd.Series(valid\_predictions)  
 disggregation\_dict[appliance] = df  
 results = pd.DataFrame(disggregation\_dict, dtype='float32')  
 test\_predictions.append(results)  
 return test\_predictions  
  
 def return\_network(self):  
 # Model architecture  
 model = Sequential()  
 model.add(Conv1D(30,10,activation="relu",input\_shape=(self.sequence\_length,1),strides=1))  
 model.add(Conv1D(30, 8, activation='relu', strides=1))  
 model.add(Conv1D(40, 6, activation='relu', strides=1))  
 model.add(Conv1D(50, 5, activation='relu', strides=1))  
 model.add(Dropout(.2))  
 model.add(Conv1D(50, 5, activation='relu', strides=1))  
 model.add(Dropout(.2))  
 model.add(Flatten())  
 model.add(Dense(1024, activation='relu'))  
 model.add(Dropout(.2))  
 model.add(Dense(1))  
 model.compile(loss='mse', optimizer=tf.keras.optimizers.Adam(self.learning\_rate)) # ,metrics=[self.mse])  
 return model  
  
 def call\_preprocessing(self, mains\_lst, submeters\_lst, method):  
  
 if method == 'train':  
 # Preprocessing for the train data  
 mains\_df\_list = []  
 for mains in mains\_lst:  
 new\_mains = mains.values.flatten()  
 n = self.sequence\_length  
 units\_to\_pad = n // 2  
 new\_mains = np.pad(new\_mains,(units\_to\_pad,units\_to\_pad),'constant',constant\_values=(0,0))  
 new\_mains = np.array([new\_mains[i:i + n] for i in range(len(new\_mains) - n + 1)])  
 new\_mains = (new\_mains - self.mains\_mean) / self.mains\_std  
 mains\_df\_list.append(pd.DataFrame(new\_mains))  
  
 appliance\_list = []  
 for app\_index, (app\_name, app\_df\_list) in enumerate(submeters\_lst):  
 if app\_name in self.appliance\_params:  
 app\_mean = self.appliance\_params[app\_name]['mean']  
 app\_std = self.appliance\_params[app\_name]['std']  
 else:  
 print ("Parameters for ", app\_name ," were not found!")  
 raise ApplianceNotFoundError()  
  
 processed\_appliance\_dfs = []  
  
 for app\_df in app\_df\_list:  
 new\_app\_readings = app\_df.values.reshape((-1, 1))  
 # This is for choosing windows  
 new\_app\_readings = (new\_app\_readings - app\_mean) / app\_std   
 # Return as a list of dataframe  
 processed\_appliance\_dfs.append(pd.DataFrame(new\_app\_readings))  
 appliance\_list.append((app\_name, processed\_appliance\_dfs))  
 return mains\_df\_list, appliance\_list  
  
 else:  
 # Preprocessing for the test data  
 mains\_df\_list = []  
  
 for mains in mains\_lst:  
 new\_mains = mains.values.flatten()  
 n = self.sequence\_length  
 units\_to\_pad = n // 2  
 new\_mains = np.pad(new\_mains,(units\_to\_pad,units\_to\_pad),'constant',constant\_values=(0,0))  
 new\_mains = np.array([new\_mains[i:i + n] for i in range(len(new\_mains) - n + 1)])  
 new\_mains = (new\_mains - self.mains\_mean) / self.mains\_std  
 mains\_df\_list.append(pd.DataFrame(new\_mains))  
 return mains\_df\_list  
  
 def set\_appliance\_params(self,train\_appliances):  
 # Find the parameters using the first  
 for (app\_name,df\_list) in train\_appliances:  
 l = np.array(pd.concat(df\_list,axis=0))  
 app\_mean = np.mean(l)  
 app\_std = np.std(l)  
 if app\_std<1:  
 app\_std = 100  
 self.appliance\_params.update({app\_name:{'mean':app\_mean,'std':app\_std}})  
 print (self.appliance\_params)

Structure of the models:

* Seq2Seq is the smallest of the three models with about 447k parameters.
* According to the summary BERT is the second largest model with 3.1M parameters. Though we are not sure whether keras is calculating the number of parameters correctly as it has some custom layers.
* The largest model seems to be the Seq2Point model with 3.6M parameters
* All models make heavy use of convolutional layers which work very well for computer vision tasks.
* The BERT model uses a transformer architecture with the so-called “attention” mechanism

Seq2Point({"n\_epochs": 5, "learning\_rate": 0.001}).return\_network().summary()

Model: "sequential\_1"  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
 Layer (type) Output Shape Param #   
=================================================================  
 conv1d\_5 (Conv1D) (None, 90, 30) 330   
   
 conv1d\_6 (Conv1D) (None, 83, 30) 7230   
   
 conv1d\_7 (Conv1D) (None, 78, 40) 7240   
   
 conv1d\_8 (Conv1D) (None, 74, 50) 10050   
   
 dropout\_3 (Dropout) (None, 74, 50) 0   
   
 conv1d\_9 (Conv1D) (None, 70, 50) 12550   
   
 dropout\_4 (Dropout) (None, 70, 50) 0   
   
 flatten\_1 (Flatten) (None, 3500) 0   
   
 dense\_2 (Dense) (None, 1024) 3585024   
   
 dropout\_5 (Dropout) (None, 1024) 0   
   
 dense\_3 (Dense) (None, 1) 1025   
   
=================================================================  
Total params: 3,623,449  
Trainable params: 3,623,449  
Non-trainable params: 0  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Seq2Seq({"n\_epochs": 5, "learning\_rate": 0.001}).return\_network().summary()

Model: "sequential\_2"  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
 Layer (type) Output Shape Param #   
=================================================================  
 conv1d\_10 (Conv1D) (None, 45, 30) 330   
   
 conv1d\_11 (Conv1D) (None, 19, 30) 7230   
   
 conv1d\_12 (Conv1D) (None, 14, 40) 7240   
   
 conv1d\_13 (Conv1D) (None, 10, 50) 10050   
   
 dropout\_6 (Dropout) (None, 10, 50) 0   
   
 conv1d\_14 (Conv1D) (None, 6, 50) 12550   
   
 dropout\_7 (Dropout) (None, 6, 50) 0   
   
 flatten\_2 (Flatten) (None, 300) 0   
   
 dense\_4 (Dense) (None, 1024) 308224   
   
 dropout\_8 (Dropout) (None, 1024) 0   
   
 dense\_5 (Dense) (None, 99) 101475   
   
=================================================================  
Total params: 447,099  
Trainable params: 447,099  
Non-trainable params: 0  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

BERT({"n\_epochs": 5, "learning\_rate": 0.001}).return\_network().summary()

Model: "sequential\_3"  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
 Layer (type) Output Shape Param #   
=================================================================  
 conv1d\_15 (Conv1D) (None, 99, 16) 80   
   
 l\_ppool (LPpool) (None, 50, 16) 0   
   
 token\_and\_position\_embeddin (None, 50, 16, 32) 643168   
 g (TokenAndPositionEmbeddin   
 g)   
   
 transformer\_block (Transfor (None, 50, 16, 32) 10656   
 merBlock)   
   
 flatten\_3 (Flatten) (None, 25600) 0   
   
 dropout\_11 (Dropout) (None, 25600) 0   
   
 dense\_8 (Dense) (None, 99) 2534499   
   
 dropout\_12 (Dropout) (None, 99) 0   
   
=================================================================  
Total params: 3,188,403  
Trainable params: 3,188,403  
Non-trainable params: 0  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
Model: "sequential\_3"  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
 Layer (type) Output Shape Param #   
=================================================================  
 conv1d\_15 (Conv1D) (None, 99, 16) 80   
   
 l\_ppool (LPpool) (None, 50, 16) 0   
   
 token\_and\_position\_embeddin (None, 50, 16, 32) 643168   
 g (TokenAndPositionEmbeddin   
 g)   
   
 transformer\_block (Transfor (None, 50, 16, 32) 10656   
 merBlock)   
   
 flatten\_3 (Flatten) (None, 25600) 0   
   
 dropout\_11 (Dropout) (None, 25600) 0   
   
 dense\_8 (Dense) (None, 99) 2534499   
   
 dropout\_12 (Dropout) (None, 99) 0   
   
=================================================================  
Total params: 3,188,403  
Trainable params: 3,188,403  
Non-trainable params: 0  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

### Building 5

The first builiding we tried to analyse was building 5, as it is one of the more interesting ones according to our data analysis. On building 5 we modeled the tumble dryer and the computer.

refit.buildings[5].elec

MeterGroup(meters=  
 ElecMeter(instance=1, building=5, dataset='REFIT', site\_meter, appliances=[])  
 ElecMeter(instance=2, building=5, dataset='REFIT', appliances=[Appliance(type='fridge freezer', instance=1)])  
 ElecMeter(instance=3, building=5, dataset='REFIT', appliances=[Appliance(type='tumble dryer', instance=1)])  
 ElecMeter(instance=4, building=5, dataset='REFIT', appliances=[Appliance(type='washing machine', instance=1)])  
 ElecMeter(instance=5, building=5, dataset='REFIT', appliances=[Appliance(type='dish washer', instance=1)])  
 ElecMeter(instance=6, building=5, dataset='REFIT', appliances=[Appliance(type='computer', instance=1)])  
 ElecMeter(instance=7, building=5, dataset='REFIT', appliances=[Appliance(type='television', instance=1)])  
 ElecMeter(instance=8, building=5, dataset='REFIT', appliances=[Appliance(type='microwave', instance=1)])  
 ElecMeter(instance=9, building=5, dataset='REFIT', appliances=[Appliance(type='kettle', instance=1)])  
 ElecMeter(instance=10, building=5, dataset='REFIT', appliances=[Appliance(type='toaster', instance=1)])  
)

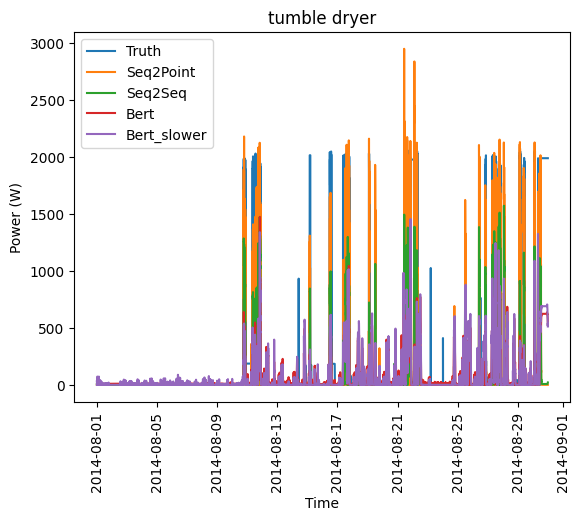
Here, we specified the parameters needed for the API. For the Seq2Seq and Seq2Point we used 20 epochs and for the computationally intensive BERT models we used only 10 epochs each. More epochs could lead to a better model performance, but we were constrained by Colab “computing units” and tried to be economical. The second BERT model also has a slower learning rate (reduced by 50%). We wanted to see if it performed better than with the default learning rate.

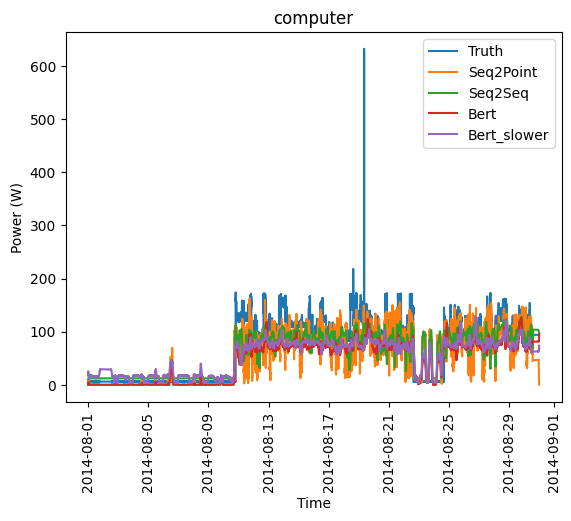
For the training process we used a sample rate of 60 (which means one data point every minute), a training period of four months (April - July 2014), and a testing period of one month (August 2014).

building5\_param = {  
 "power": {"mains": ["apparent","active"],"appliance": ["apparent","active"]},  
 "sample\_rate": 60,  
 "appliances": [ "tumble dryer", "computer" ],  
 "methods": {"Seq2Point": Seq2Point({"n\_epochs": 20}),   
 "Seq2Seq": Seq2Seq({"n\_epochs": 20}),   
 "Bert": BERT({"n\_epochs": 10}),  
 "Bert\_slower": BERT({"n\_epochs": 10, "learning\_rate": 0.0005})  
 },  
 "display\_predictions": True,  
 "train": {   
 "datasets": {  
 "Dataport": {  
 "path": file\_path,  
 "buildings": {  
 5: {  
 "start\_time": "2014-04-01",  
 "end\_time": "2014-07-31"  
 }  
 }   
 }  
 }  
 },  
 "test": {  
 "datasets": {  
 "Dataport": {  
 "path": file\_path,  
 "buildings": {  
 5: {  
 "start\_time": "2014-08-01",  
 "end\_time": "2014-08-31"  
 }  
 }  
 }  
 },  
 "metrics":["rmse"]  
 }  
 }

# Model Training. Saving results to a file  
if Path(f"{data\_path}/building5.joblib").exists() == False:  
 building5\_mod = API(building5\_param)  
 results = {   
 "pred\_overall": building5\_mod.pred\_overall,   
 "errors": building5\_mod.errors,  
 "test\_mains": building5\_mod.test\_mains,  
 "test\_submeters": building5\_mod.test\_submeters  
 }  
 with open(f"{data\_path}/building5.joblib", "wb") as f:  
 joblib.dump(results, f)

Joint Testing for all algorithms  
Loading data for Dataport dataset  
Dropping missing values  
Generating predictions for : Seq2Point  
85/85 [==============================] - 0s 2ms/step  
85/85 [==============================] - 0s 2ms/step  
Generating predictions for : Seq2Seq  
84/84 [==============================] - 0s 2ms/step  
84/84 [==============================] - 0s 2ms/step  
Generating predictions for : BERT  
84/84 [==============================] - 3s 36ms/step  
84/84 [==============================] - 3s 36ms/step  
Generating predictions for : BERT  
84/84 [==============================] - 3s 36ms/step  
84/84 [==============================] - 3s 36ms/step  
............ rmse ..............  
 Seq2Point Seq2Seq Bert Bert\_slower  
tumble dryer 405.726374 400.311207 386.986838 387.437535  
computer 31.641851 28.285084 30.947019 30.834153





with open(f"{data\_path}/building5.joblib", "rb") as f:  
 results = joblib.load(f)  
print(results["errors"])

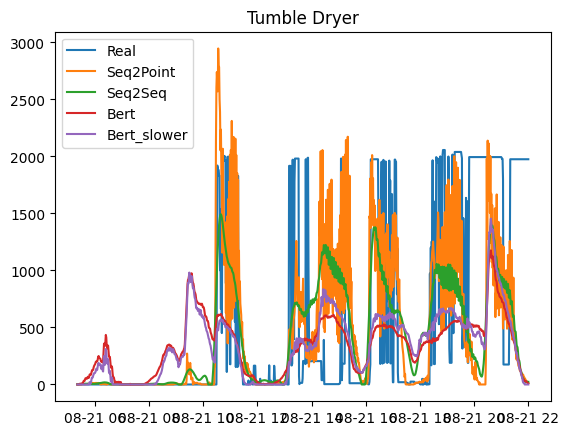
[ Seq2Point Seq2Seq Bert Bert\_slower  
tumble dryer 405.726374 400.311207 386.986838 387.437535  
computer 31.641851 28.285084 30.947019 30.834153]

Above we reported the rmse errors. The regular BERT performed best for the tumble dryer, while the Seq2Seq model was best for the computer. Interestingly, the slower learning rate for the BERT didn´t really materialize into a significantly better prediction performance.

Looking at the plots the sequence models Seq2Point and Seq2Seq seemed to better capture the spikes (variance), while the BERT predictions had less variance.

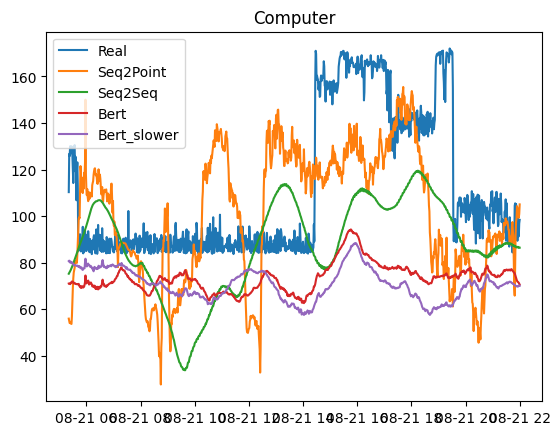
Zooming into an arbitrary window leads to the following plots:

a = 29000  
b = 30000  
col = 0 # Tumble Dryer  
  
plt.plot(results["test\_submeters"][col][1][0][a:b], label = "Real")  
plt.plot(results["pred\_overall"]["Seq2Point"].iloc[a:b,col], label = "Seq2Point")  
plt.plot(results["pred\_overall"]["Seq2Seq"].iloc[a:b,col], label = "Seq2Seq")  
plt.plot(results["pred\_overall"]["Bert"].iloc[a:b,col], label = "Bert")  
plt.plot(results["pred\_overall"]["Bert\_slower"].iloc[a:b,col], label = "Bert\_slower")  
plt.title("Tumble Dryer")  
plt.legend()  
plt.plot()



Again, we see the same pattern: BERT models capture the location of the spikes but not the complete magnitude. Moreover, it seems that the BERT models sometimes captured random noise.

a = 29000  
b = 30000  
col = 1 # Computer  
  
plt.plot(results["test\_submeters"][col][1][0][a:b], label = "Real")  
plt.plot(results["pred\_overall"]["Seq2Point"].iloc[a:b,col], label = "Seq2Point")  
plt.plot(results["pred\_overall"]["Seq2Seq"].iloc[a:b,col], label = "Seq2Seq")  
plt.plot(results["pred\_overall"]["Bert"].iloc[a:b,col], label = "Bert")  
plt.plot(results["pred\_overall"]["Bert\_slower"].iloc[a:b,col], label = "Bert\_slower")  
plt.title("Computer")  
plt.legend()  
plt.plot()



For the computer the BERT models were much more consistent than the sequence models. The Seq2Seq and Seq2Point seemed to struggle with noise in the time series leading to random spikes, making them rather impractical for the computer.

### Building 7

refit.buildings[7].elec

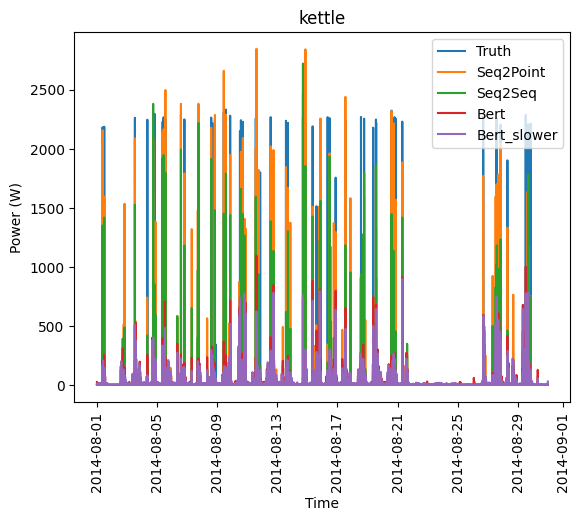
MeterGroup(meters=  
 ElecMeter(instance=1, building=7, dataset='REFIT', site\_meter, appliances=[])  
 ElecMeter(instance=2, building=7, dataset='REFIT', appliances=[Appliance(type='fridge', instance=1)])  
 ElecMeter(instance=3, building=7, dataset='REFIT', appliances=[Appliance(type='freezer', instance=1)])  
 ElecMeter(instance=4, building=7, dataset='REFIT', appliances=[Appliance(type='freezer', instance=2)])  
 ElecMeter(instance=5, building=7, dataset='REFIT', appliances=[Appliance(type='tumble dryer', instance=1)])  
 ElecMeter(instance=6, building=7, dataset='REFIT', appliances=[Appliance(type='washing machine', instance=1)])  
 ElecMeter(instance=7, building=7, dataset='REFIT', appliances=[Appliance(type='dish washer', instance=1)])  
 ElecMeter(instance=8, building=7, dataset='REFIT', appliances=[Appliance(type='television', instance=1)])  
 ElecMeter(instance=9, building=7, dataset='REFIT', appliances=[Appliance(type='toaster', instance=1)])  
 ElecMeter(instance=10, building=7, dataset='REFIT', appliances=[Appliance(type='kettle', instance=1)])  
)

For building 7 we tried to predict the kettle and the dish washer. Both appliances are used for a very short period of time only.

building7\_param = {  
 "power": {"mains": ["apparent","active"],"appliance": ["apparent","active"]},  
 "sample\_rate": 60,  
 "appliances": ["kettle", "dish washer"],  
 "methods": {"Seq2Point": Seq2Point({"n\_epochs": 20}),   
 "Seq2Seq": Seq2Seq({"n\_epochs": 20}),   
 "Bert": BERT({"n\_epochs": 10}),  
 "Bert\_slower": BERT({"n\_epochs": 10, "learning\_rate": 0.0005})  
 },  
 "display\_predictions": True,  
 "train": {   
 "datasets": {  
 "Dataport": {  
 "path": file\_path,  
 "buildings": {  
 7: {  
 "start\_time": "2014-04-01",  
 "end\_time": "2014-07-31"  
 }  
 }   
 }  
 }  
 },  
 "test": {  
 "datasets": {  
 "Dataport": {  
 "path": file\_path,  
 "buildings": {  
 7: {  
 "start\_time": "2014-08-01",  
 "end\_time": "2014-08-31"  
 }  
 }  
 }  
 },  
 "metrics":["rmse"]  
 }  
 }

if Path(f"{data\_path}/building7.joblib").exists() == False:  
 building7\_mod = API(building7\_param)  
 results = {   
 "pred\_overall": building7\_mod.pred\_overall,   
 "errors": building7\_mod.errors,  
 "test\_mains": building7\_mod.test\_mains,  
 "test\_submeters": building7\_mod.test\_submeters   
 }  
 with open(f"{data\_path}/building7.joblib", "wb") as f:  
 joblib.dump(results, f)

Joint Testing for all algorithms  
Loading data for Dataport dataset  
Dropping missing values  
Generating predictions for : Seq2Point  
84/84 [==============================] - 0s 3ms/step  
84/84 [==============================] - 0s 2ms/step  
Generating predictions for : Seq2Seq  
84/84 [==============================] - 0s 2ms/step  
84/84 [==============================] - 0s 2ms/step  
Generating predictions for : BERT  
84/84 [==============================] - 3s 36ms/step  
84/84 [==============================] - 3s 36ms/step  
Generating predictions for : BERT  
84/84 [==============================] - 3s 36ms/step  
84/84 [==============================] - 3s 36ms/step  
............ rmse ..............  
 Seq2Point Seq2Seq Bert Bert\_slower  
kettle 87.887507 86.512243 110.119227 112.316187  
dish washer 133.175607 125.308619 259.270627 264.857701



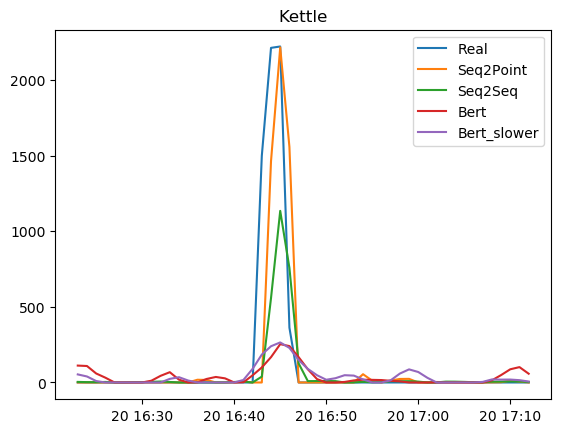


with open(f"{data\_path}/building7.joblib", "rb") as f:  
 results = joblib.load(f)  
print(results["errors"])

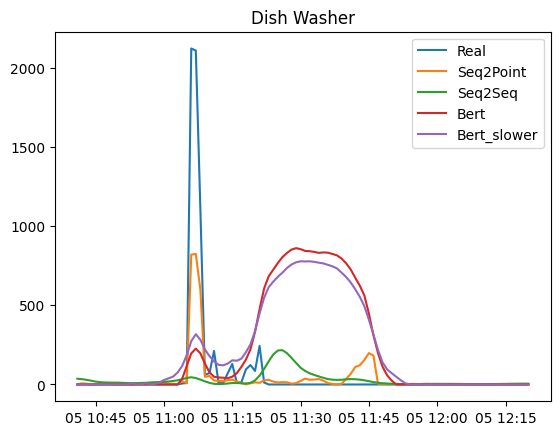
[ Seq2Point Seq2Seq Bert Bert\_slower  
kettle 87.887507 86.512243 110.119227 112.316187  
dish washer 133.175607 125.308619 259.270627 264.857701]

Looking at the errors, the Seq2Seq and Seq2Point clearly outperformed the BERT transformer models. It’s likely that the outperformance is due to the architecture of the models.

a = 28250  
b = 28300  
col = 0 # Kettle  
  
plt.plot(results["test\_submeters"][col][1][0][a:b], label = "Real")  
plt.plot(results["pred\_overall"]["Seq2Point"].iloc[a:b,col], label = "Seq2Point")  
plt.plot(results["pred\_overall"]["Seq2Seq"].iloc[a:b,col], label = "Seq2Seq")  
plt.plot(results["pred\_overall"]["Bert"].iloc[a:b,col], label = "Bert")  
plt.plot(results["pred\_overall"]["Bert\_slower"].iloc[a:b,col], label = "Bert\_slower")  
plt.title("Kettle")  
plt.legend()  
plt.plot()



a = 6400  
b = 6500  
col = 1 # Dish Washer  
  
plt.plot(results["test\_submeters"][col][1][0][a:b], label = "Real")  
plt.plot(results["pred\_overall"]["Seq2Point"].iloc[a:b,col], label = "Seq2Point")  
plt.plot(results["pred\_overall"]["Seq2Seq"].iloc[a:b,col], label = "Seq2Seq")  
plt.plot(results["pred\_overall"]["Bert"].iloc[a:b,col], label = "Bert")  
plt.plot(results["pred\_overall"]["Bert\_slower"].iloc[a:b,col], label = "Bert\_slower")  
plt.title("Dish Washer")  
plt.legend()  
plt.plot()



The transformer models predicted a second “bump” for the dish washer probably due to some random noise.

### Building 14

refit.buildings[14].elec

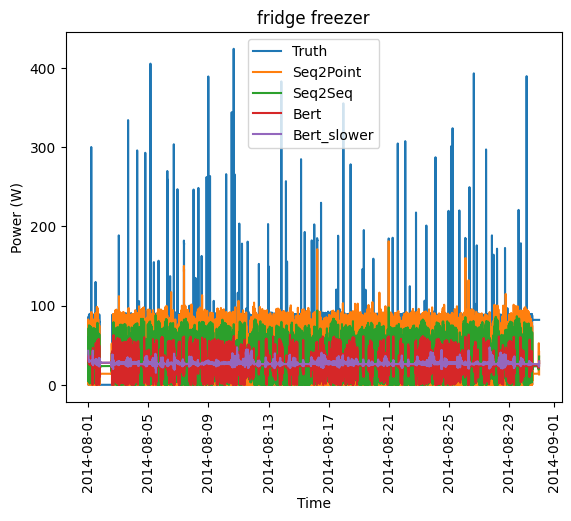
MeterGroup(meters=  
 ElecMeter(instance=1, building=14, dataset='REFIT', site\_meter, appliances=[])  
 ElecMeter(instance=2, building=14, dataset='REFIT', appliances=[Appliance(type='fridge freezer', instance=1)])  
 ElecMeter(instance=3, building=14, dataset='REFIT', appliances=[Appliance(type='tumble dryer', instance=1)])  
 ElecMeter(instance=4, building=14, dataset='REFIT', appliances=[Appliance(type='washing machine', instance=1)])  
 ElecMeter(instance=5, building=14, dataset='REFIT', appliances=[Appliance(type='dish washer', instance=1)])  
 ElecMeter(instance=6, building=14, dataset='REFIT', appliances=[Appliance(type='computer', instance=1)])  
 ElecMeter(instance=7, building=14, dataset='REFIT', appliances=[Appliance(type='television', instance=1)])  
 ElecMeter(instance=8, building=14, dataset='REFIT', appliances=[Appliance(type='microwave', instance=1)])  
 ElecMeter(instance=9, building=14, dataset='REFIT', appliances=[Appliance(type='audio system', instance=1)])  
 ElecMeter(instance=10, building=14, dataset='REFIT', appliances=[Appliance(type='toaster', instance=1)])  
)

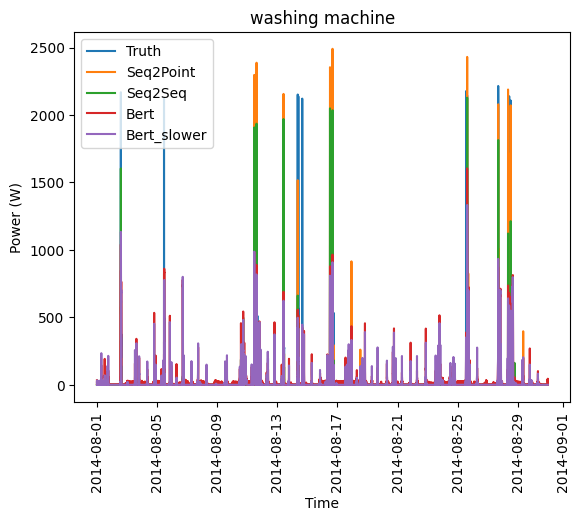
For building 14 we tried the models on a varity of appliance patterns.

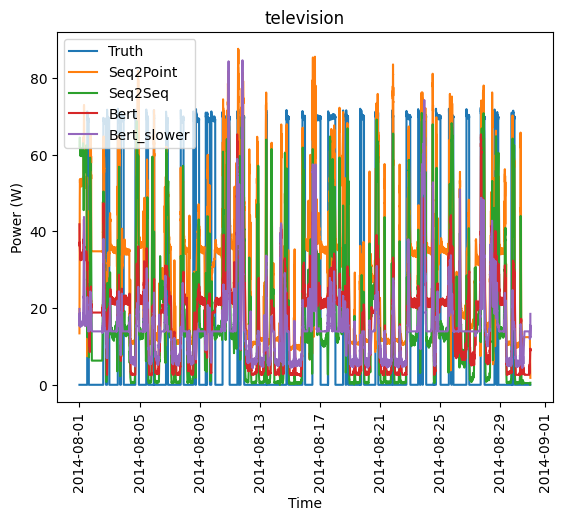
building14\_param = {  
 "power": {"mains": ["apparent","active"],"appliance": ["apparent","active"]},  
 "sample\_rate": 60,  
 "appliances": [ "fridge freezer", "washing machine", "television", "audio system" ],  
 "methods": {"Seq2Point": Seq2Point({"n\_epochs": 20}),   
 "Seq2Seq": Seq2Seq({"n\_epochs": 20}),   
 "Bert": BERT({"n\_epochs": 10}),  
 "Bert\_slower": BERT({"n\_epochs": 10, "learning\_rate": 0.0005})  
 },  
 "display\_predictions": True,  
 "train": {   
 "datasets": {  
 "Dataport": {  
 "path": file\_path,  
 "buildings": {  
 14: {  
 "start\_time": "2014-04-01",  
 "end\_time": "2014-07-31"  
 }  
 }   
 }  
 }  
 },  
 "test": {  
 "datasets": {  
 "Dataport": {  
 "path": file\_path,  
 "buildings": {  
 14: {  
 "start\_time": "2014-08-01",  
 "end\_time": "2014-08-31"  
 }  
 }  
 }  
 },  
 "metrics":["rmse"]  
 }  
 }

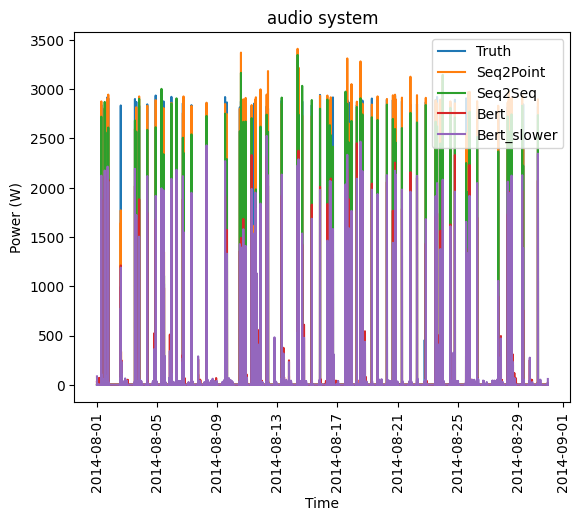
if Path(f"{data\_path}/building14.joblib").exists() == False:  
 building14\_mod = API(building14\_param)  
 results = {   
 "pred\_overall": building14\_mod.pred\_overall,   
 "errors": building14\_mod.errors,  
 "test\_mains": building14\_mod.test\_mains,  
 "test\_submeters": building14\_mod.test\_submeters   
 }  
 with open(f"{data\_path}/building14.joblib", "wb") as f:  
 joblib.dump(results, f)

Joint Testing for all algorithms  
Loading data for Dataport dataset  
Dropping missing values  
Generating predictions for : Seq2Point  
85/85 [==============================] - 1s 6ms/step  
85/85 [==============================] - 1s 5ms/step  
85/85 [==============================] - 1s 4ms/step  
85/85 [==============================] - 1s 4ms/step  
Generating predictions for : Seq2Seq  
85/85 [==============================] - 0s 4ms/step  
85/85 [==============================] - 0s 3ms/step  
85/85 [==============================] - 0s 2ms/step  
85/85 [==============================] - 0s 2ms/step  
Generating predictions for : BERT  
85/85 [==============================] - 18s 215ms/step  
85/85 [==============================] - 19s 218ms/step  
85/85 [==============================] - 19s 215ms/step  
85/85 [==============================] - 18s 214ms/step  
Generating predictions for : BERT  
85/85 [==============================] - 18s 214ms/step  
85/85 [==============================] - 19s 215ms/step  
85/85 [==============================] - 18s 215ms/step  
85/85 [==============================] - 18s 214ms/step  
............ rmse ..............  
 Seq2Point Seq2Seq Bert Bert\_slower  
fridge freezer 21.736150 25.764762 34.535043 38.285467  
washing machine 52.042083 61.049155 104.372881 104.982779  
television 29.410966 24.490124 26.568183 27.223311  
audio system 47.394926 50.714894 94.528559 90.211560







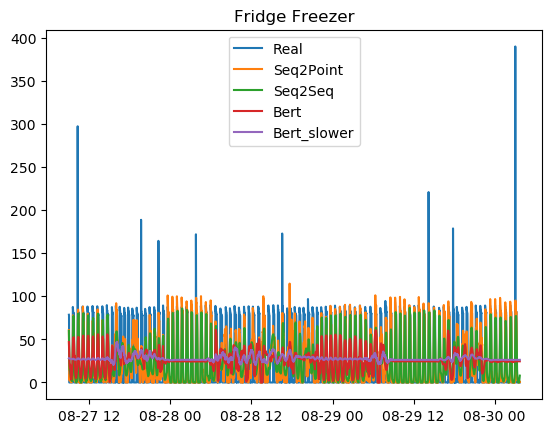


with open(f"{data\_path}/building14.joblib", "rb") as f:  
 results = joblib.load(f)  
print(results["errors"])

[ Seq2Point Seq2Seq Bert Bert\_slower  
fridge freezer 21.736150 25.764762 34.535043 38.285467  
washing machine 52.042083 61.049155 104.372881 104.982779  
television 29.410966 24.490124 26.568183 27.223311  
audio system 47.394926 50.714894 94.528559 90.211560]

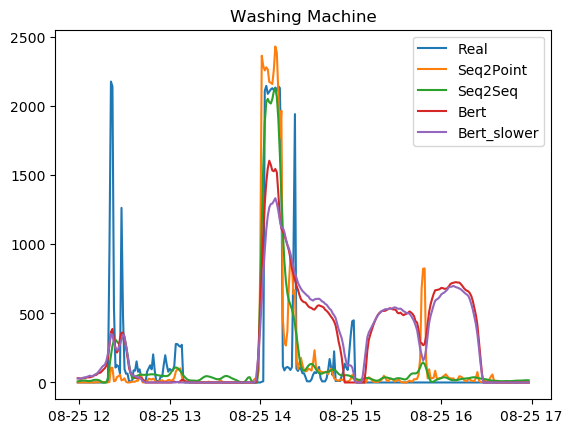
The sequence models again outperformed the transformer models on the RMSE metric. The transformer models seemed to be able to predict the time of the spikes but not their entire magnitude.

a = 38000  
b = 42000  
col = 0 # Fridge Freezer  
  
plt.plot(results["test\_submeters"][col][1][0][a:b], label = "Real")  
plt.plot(results["pred\_overall"]["Seq2Point"].iloc[a:b,col], label = "Seq2Point")  
plt.plot(results["pred\_overall"]["Seq2Seq"].iloc[a:b,col], label = "Seq2Seq")  
plt.plot(results["pred\_overall"]["Bert"].iloc[a:b,col], label = "Bert")  
plt.plot(results["pred\_overall"]["Bert\_slower"].iloc[a:b,col], label = "Bert\_slower")  
plt.title("Fridge Freezer")  
plt.legend()  
plt.plot()



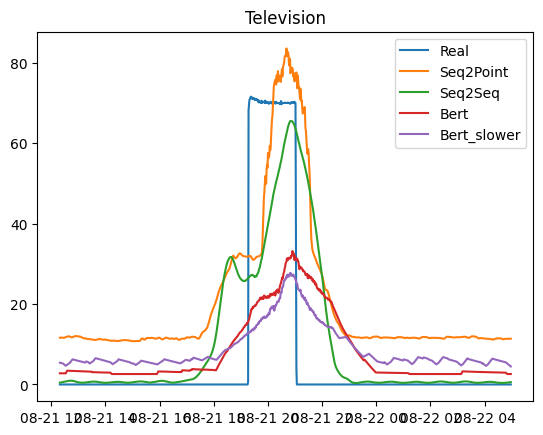
The fridge has an oscillating pattern, which works well for the sequence models but not for the transformer models. Sometimes the BERT predictions were oscillating and sometimes the prediction was only a flat line. Neither architecture recognized the high magnitude spikes. A slower learning rate worsened the performance.

a = 35300  
b = 35600  
col = 1 # Washing Machine  
  
plt.plot(results["test\_submeters"][col][1][0][a:b], label = "Real")  
plt.plot(results["pred\_overall"]["Seq2Point"].iloc[a:b,col], label = "Seq2Point")  
plt.plot(results["pred\_overall"]["Seq2Seq"].iloc[a:b,col], label = "Seq2Seq")  
plt.plot(results["pred\_overall"]["Bert"].iloc[a:b,col], label = "Bert")  
plt.plot(results["pred\_overall"]["Bert\_slower"].iloc[a:b,col], label = "Bert\_slower")  
plt.title("Washing Machine")  
plt.legend()  
plt.plot()



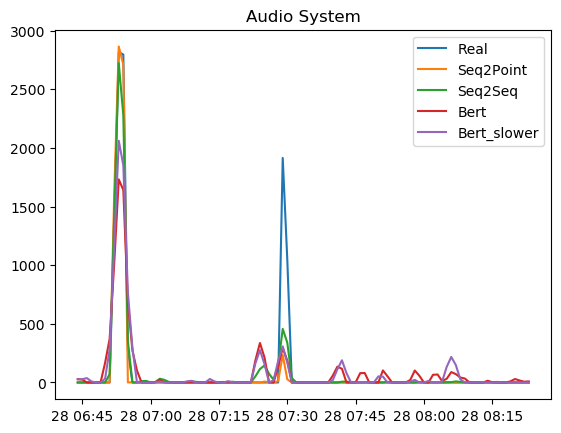
The washing mashine pattern is a typical example for the transformer models predicting a second “bump” after a real high. Maybe due to the self-attention mechanism?

a = 29500  
b = 30500  
col = 2 # Television  
  
plt.plot(results["test\_submeters"][col][1][0][a:b], label = "Real")  
plt.plot(results["pred\_overall"]["Seq2Point"].iloc[a:b,col], label = "Seq2Point")  
plt.plot(results["pred\_overall"]["Seq2Seq"].iloc[a:b,col], label = "Seq2Seq")  
plt.plot(results["pred\_overall"]["Bert"].iloc[a:b,col], label = "Bert")  
plt.plot(results["pred\_overall"]["Bert\_slower"].iloc[a:b,col], label = "Bert\_slower")  
plt.title("Television")  
plt.legend()  
plt.plot()



The sequence models responded faster to a high, while transformers were slower and responded with a lower magnitude.

a = 39300  
b = 39400  
col = 3 # Audio System  
  
plt.plot(results["test\_submeters"][col][1][0][a:b], label = "Real")  
plt.plot(results["pred\_overall"]["Seq2Point"].iloc[a:b,col], label = "Seq2Point")  
plt.plot(results["pred\_overall"]["Seq2Seq"].iloc[a:b,col], label = "Seq2Seq")  
plt.plot(results["pred\_overall"]["Bert"].iloc[a:b,col], label = "Bert")  
plt.plot(results["pred\_overall"]["Bert\_slower"].iloc[a:b,col], label = "Bert\_slower")  
plt.title("Audio System")  
plt.legend()  
plt.plot()



The audio system had a very short usage-time, and all models recognized the spikes. The BERT models fit some random noise and as a result, predicted phantom spikes.

## Conclusion

* It’s not clear whether the transformer architecture is superior to the sequence models. Quite often the transformers struggle with oscillating patterns or very high magnitudes. Also, their training is computationally much more expensive than sequence models.
* More epochs for the BERT models would have been better, but we were constrained by Google Colab computing units.
* There is room for a lot more experiments, for example by changing learning rates or the pre-processing functions.