Energy Disaggregation using Non-Intrusive Load Monitoring

Karen Yu, Nick Vasios and Thibaut Perol
Final project for AM207

1 Abstract

2 Introduction

Energy disaggregation is the procedure that infers the energy consumption of appliances in a household given the total energy consumption from a single meter of that household. In recent years, this field has become increasingly popular as smart meters have begun to deploy and are installed in many households across the world, providing energy consumption data at high temporal resolution. This high resolution data enables the use of computer algorithms to estimate the energy consumption of each appliance in the household without having to install meters on individual appliances (hence, non-intrusive load monitoring). Appliance-specific energy consupmtion information can be provided to homeowners to encourage adoption of energy efficiency habits and identify the appliances that would result in the most cost savings if replaced with more energy efficient ones. From the electric utility's perspective, the appliance specific information can be used for demand-response programs and also provide information about how their customers are using electricity within their homes.

Research on NILM began in the 1980s with Hart, 1985. In these earlier methods, the patterns in electricity consumption of different appliances were identified by humans and these hand-designed feature extractors were then applied to the aggregate signals. Later works (**?**) developed methods to automatically identify appliances and perform disaggregation. The technology has also been monetized by several startups (eg. Bidgeley, PlotWatt).

Several data sets have been released for the purpose of comparing disaggregation methods. The Reference Energy Disaggregation Dataset (REDD) (**?**) was released in 2011 from MIT. It includes data from 6 houses spaning a period of 3-19 days for each house and contains sub-metered (appliance-level) data for each household. The Smart* dataset was released in 2012, with sub-metered data for one household and aggregate data for 3 households over a period of 3 months. Most recently, the UK-DALE datset was released in 2014, with 3-17 months of data for 4 households with appliance-level submeters. Due to the cost and intrusiveness of installing appliance-level meters, these datasets tend to have only a few households or span a short period of time. Additionally, the datasets are differently formatted depending on the research group that collected the data, making comparisons among different datasets difficult.

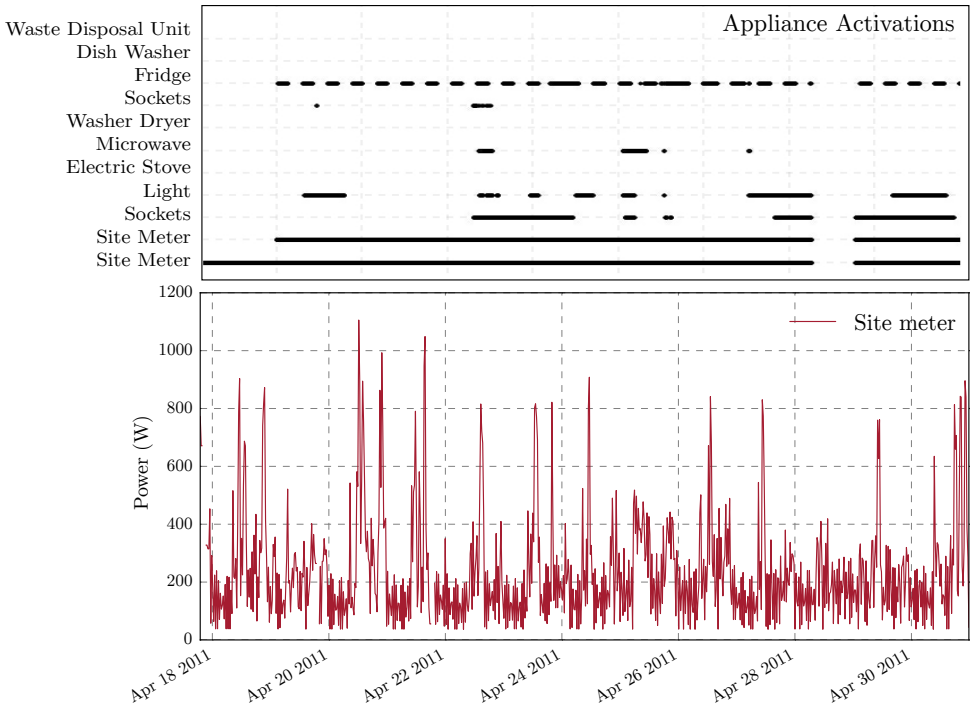
In an attempt to standardize comparison of different disaggregation methods, **?** released the Non-intrusive Load Monitoring Toolkit (NILMTK). NILMTK is an open-source toolkit that provides tools for data processing and evaluation metrics. NILMTK also provides two benchmark disaggregation algorithms, combinatorial optimization (CO) and factorial hidden markov model (FHMM).

In this project, we compare a new approach using convolutional neural networks against the standard CO and FHMM implementations in NILMTK.

3 Methods

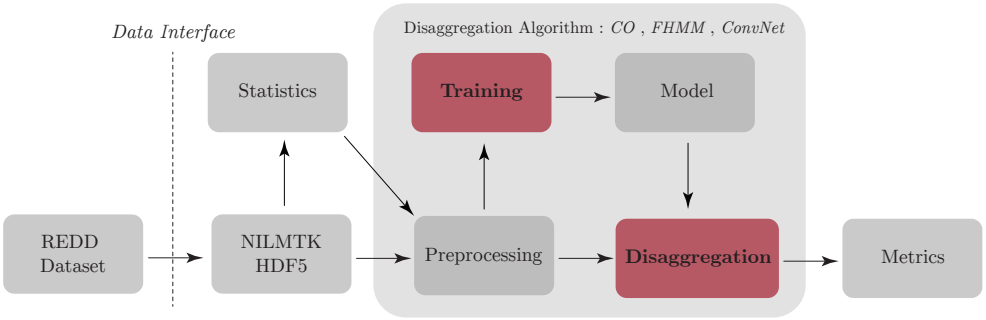
3.1 Data processing

We use the REDD dataset, which contains data aggregated and sub-metered data from 6 houses. We use the functions provided in `NILMTK` to read in the data. In the figure below, we show the aggregated energy consumption of a household (building 2) over a 13 day period as well as the times each individual appliance was on during a period of 1 day.



The data is organized by buildings. Within each building, we can get the electricity meter readings using the `.elec` attribute. The `.mains` sub-meters refer to the aggregated readings. We can refer to each appliance by giving the appliance as a keyword to `data.buildings[building_num].elec[appliance]`. Some typical appliances in these households include refrigerators, washter dryers, lights, dish washers, microwaves, and sockets. We decided to use fridge and microwave as the two appliances for which we determine the power consumption because they are used in most households (buildings 1, 2, 3, 5).

The data pipeline is shown in the following diagram



The REDD dataset if first converted to HDF5 format. `NILMTK` provides functions to compute statistcs on the dataset as well as preprocessing algorithms. The data from individual meters is then fed into the model for training. After training, the aggregated data can then be fed into the model for disaggregation. `NILMTK` also provides functions for computing metrics of how well each disaggregation algorithm performs.

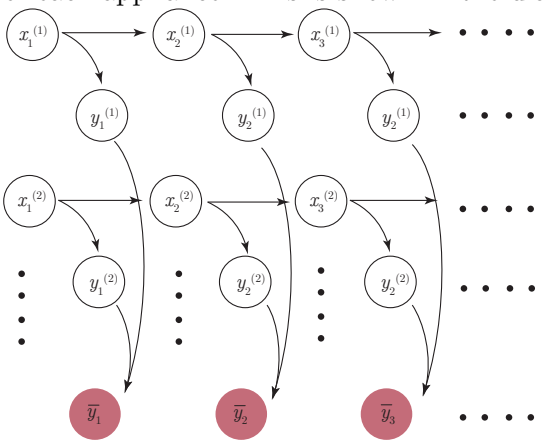
The metrics that we use for comparison of methods are recall, precision, F1, and accuracy, which are created from combinations of the metrics true positives (TP), false positives (FP). false negatives (FN), false negatives (FN), true negatives (TN), number of positives in ground truth (P), and number of negatives in ground truth (N).

$$\text{recall} = \frac{TP}{FP+FN}, \quad \text{precision} = \frac{TP}{TP+FP}, \quad F1 = 2 \times \frac{\text{precision} \times \text{recall}}{\text{precision} + \text{recall}} \quad \text{accuracy} = \frac{TP+TN}{P+N}$$

3.2 Combinatorial Optimization

3.3 Factorial Hidden Markov Model

Factorial hidden markov models (FHMM) are an extension to hidden markov models (HMM) where the hidden state is factored into multiple state variables (**?**). In a hidden markov model, the observed states of the system are the aggregated power consumption of the household. The hidden states are the states of each appliance. This is shown in the diagram below.



In the above diagram, y_t is the observed state (aggregate power consumption) at time t, $y_t^{(i)}$ is the power consumption of appliance i at time t (used only in training), and $x_t^{(i)}$ is the hidden state of appliance

i at time t. Since the observed state is continuous, the emission probabilities are modeled with a Gaussian distribution:

$$P(y_t|x_t^{(1:N)}) = \mathcal{N}\left(\sum_{i=1}^{N} \mu^{(i)}, \Sigma\right)$$

where the mean is the sum of the means of the individual appliances. The transition probabilities can be factored as

$$P(x_t^{1:N)}|x_{t-1}^{(1:N)}) = \prod_{i=1}^{N} P(x_t^{(i)}|x_{t-1}^{(i)})$$

and the transition probabilities for each individual appliance follow a multinomial distribution. The model also needs an initial probability distribution, π_0 which also follows a multinomial probability distribution.

In the NILMTK implementation, the `train_across_buildings` function takes as input the sub-metered data for each house, a list of the appliances we want to train for, and a list of the houses we want to include in the training set. The sub-metered data for each appliance is then combined into an array that includes all training houses. An HMM with Gaussian emissions is built for each appliance using the `GaussianHMM` function in the `hmmlearn` package. The parameters (transition and emission probabilities) for this HMM is then trained using the `fit` function from `hmmlearn`, which uses expectation-maximization (EM) to find the optimal parameters. In the original NILMTK implementation, there are two hidden states for each appliance to represent on and off states. We modify the function to allow for a user-specified number of hidden states.

Once these parameters are fit for all appliances, they are combined to form the FHMM. The FHMM has a hidden state for every possible combination of states. To create the combined emission matrix, the mean power consumption for each appliance is summed (if the fridge comsumes 0.5W while off and 150W while on, and a microwave consumes 3 W while off and 350 W while on, then the states would be 3.5 W for both devices off, 153 W for fridge on, microwave off, 350.5W for fridge off, microwave on, and 500 W for both devices on). To create the combined transition matrix, we take the product of the transition probabilities of individual appliances. A new `GaussianHMM` model is created using these combined parameters.

To disaggregate (infer the hidden states), the `predict` function of `hmmlearn` is used, which has the option of using the Viterbi algorithm or maximum a posteriori (MAP) to decode the most likely state sequence.

4 Convolutional Neural Network (ConvNet)

The implementation of the method presented in this section can be found in the notebook `https://github.com/tperol/am207-NILM-project/blob/master/Report_convnet.ipynb`. However the main codes are available in a separate repository (`https://github.com/tperol/neuralnilm`) to keep this final repository clean. Most of the codes for preprocessing of the data are borrowed from Jack Kelly repository that was forked (`https://github.com/JackKelly/neuralnilm`). However the implementation of the python generator for the data augmentation on CPU, the ConvNet implementation (trained on GPU) and post processing for the metrics are our own implementation.

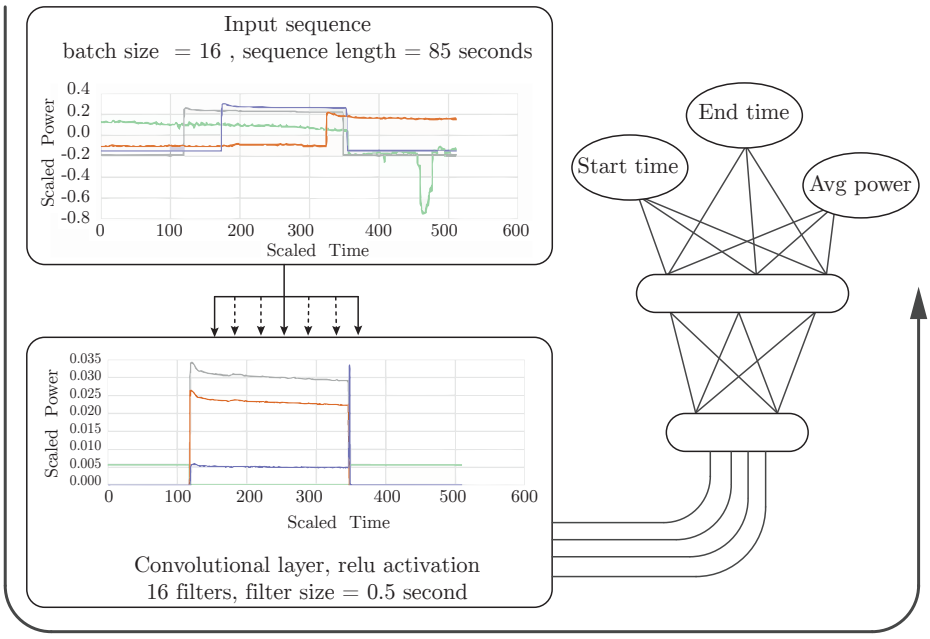


Figure 1: A schematic representation of the architecture of the convolutional neural network

4.1 ConvNet introduction

Convolutional Neural Networks are similar to ordinary Neural Networks (multi-layer perceptrons). Each neuron receive an input, perform a dot product with its weights and follow this with a non-linearity (here we only use ReLu activation functions). The whole network has a loss function that is here the Root Mean Square (RMS) error (details later). The network implements the 'rectangle method'. From the input sequence we invert for the start time, the end time and the average power of only one appliance (see Figure 1).

Convolutional neural networks have revolutionized computer vision. From an image the convolutional layer learns through its weights low level features. In the case of an image the features detectors (filters) would be: horizontal lines, blobs etc. These filters are built using a small receptive field and share weights across the entire input, which makes them translation invariant. Similarly, in the case of time series, the filters extract low level feature in the time series. By experimenting we found that only using 16 of these filters gives a good predictive power to the ConvNet. This convolutional layer is then flatten and we use 2 hidden layers of 1024 and 512 neurons with ReLu activation function before the output layer of 3 neurons (start time, end time and average power).

4.2 Data pipeline

4.2.1 Selecting appliances

We train each neural network per appliance. This is different from the CO and FHMM methods. For this report we only try to invert for the activation of the fridge and the microwave in the aggregated data. This two appliances have very different activation signatures (see Figure !!!).

4.2.2 Selecting time sequences

We downsampled the main meters and the submeters to 6 samples per seconds in order to have the aggregated and the submeter sequences properly aligned. We throw away any activation shorter than some threshold duration to avoid spurious spikes. For each sequence we use 512 samples (about 85 seconds of recording).

4.2.3 Selecting houses

We choose to train the algorithm on house 1,2,3 and 6 and test the data on house 5.

4.2.4 Dealing with unbalanced data: selecting aggregated data windows

We first extract using NILMTK libraries (`http://nilmtk.github.io`) the target appliance (fridge or microwave) activations in the time series. We concatenate the times series from house 1,2,3, and 6 for the training set and will test on house 5. We feed to our neural network algorithm (detailed later) balanced mini-batches of data sequences of aggregated data in which the fridge is activated and sequences in which it is not activated. This is a way to deal with unbalanced data – there are more sequences where the fridge is not activated than sequences with the fridge activated. Most of the data pipeline used is borrowed from `https://github.com/JackKelly/neuralnilm`.

4.2.5 Synthetic aggregated data

We use the method from Jack Kelly to create synthetic data. Two vectors of the size of the window fed to the network are initialized: the input and the target. The input is create with the combination of activations of the five most active appliances in each house. There is a 50 % chance that the target appliance will appear in the sequence and a 25 % chance for each other 'distractor' appliance. We ran neural networks with and without synthetic aggregated data. We found that synthetic data acts as a regularizer, it improves the scores on unseen house.

4.2.6 Standardization of the input data (aggregated data)

A typical step in the data pipeline of neural network is the standardization of data. For each sequences of 512 samples (= 85 seconds) we subtract the mean to center the sequence. Furthermore every input sequence is divided by the standard deviation of a random sample in the training set. In this case we

cannot divide each sequence by its own standard deviation because it would delete information about the scale of the signal. An example of 4 input sequences is shown in Figure 1.

4.2.7 Output data (start time, end time and average power)

The output of the neural network is 3 neurons: start time, end time and average power. We rescale the time to the interval [0,1]. Therefore if the fridge starts in the middle of the input sequences the output of the first neuron is 0.5. If its stops after the end of the input window the output of the second neuron is set to 1. The third neuron is the average power during the activation period. Of course this is set to 0 when it is not activated during the input sequence. We also post process the data by setting any start time lower than 0 to 0 and end time higher than 1 to 1. We create a average power threshold set to 0.1 that indicates if the appliance was active or not (under the threshold the appliance is considered off, above it is considered on).

Here we show as an example the input data and the output calculated by a trained network. We compare this with the real appliance activation.

4.3 Scores

Because of the dimension of the ouput we choose classification score metrics. When the starting time and the ending time are both 0 we call this a negative. We also call negative if the power average is lower than threshold. Otherwise this is positive (the appliance is activated). We call TP true positive, TN true negative, FP false positive and FN false negative. The various metrics/scores used in this study are

$$recall = \frac{TP}{TP + FN} \tag{1}$$

$$precision = \frac{TP}{TP + FP} \tag{2}$$

$$F1 = 2 * \frac{precision * recall}{precision + recall} \tag{3}$$

$$accuracy = \frac{TP + TN}{P + N} \tag{4}$$

4.4 Implementation strategy for real time data augmentation

While the neural network runs an NVIDIA GeForce GT 750M (GPU) we maintain the CPU busy doing the data augmentation in real time (load aggregated data, create the synthetic data, preprocess the mini-batch to be fed to the neural network). For this we create a python generator that creates a queue of 50 mini-batch and feed them successively to the GPU for training. The pipeline class can be found in neuralnilm.data.datapiline at `https://github.com/tperol/neuralnilm` and is partially reproduced here. We do the same to generate the validation and test set.

4.5 ConvNet architecture

We use a convolutional neural network (ConvNet) to take advantage of the translation invariance. We want the ConvNet to recognize target appliance activation anywhere in the sequence. For this project we have tried multiple architecture that are reported later on. These architecture all have a first convolutional layer of filter size 3 and stride 1. We have played with both the filter size and the number of output filters on the first layer. We have found that 16 filters is a reasonable number – increasing the number of filters in the first layer did not improve significantly the scores. The best neural network we found consist of

1. Input layer: one channel and lenght of 512 samples
2. 1D convolutional layer (filter size = 3, stride = 1 , number of filters = 16, activation function = relu, border mode = valid, weight initialization = normal distribution)
3. Fully connected layer (N = 1024, activation function = relu, weight initialization = normal distribution)
4. Fully connected layer (N = 512, activation function = relu, weight initialization = normal distribution)
5. Fully connected layer (N= 3, activation function = relu)

The ouput has 3 neurons activated by a relu activation function since the output cannot be negative. We have tried other networks that are reported later in this notebook. However this is the layout of the best one we found.

4.6 Loss function and optimizer

4.6.1 Loss function

Since the output neurons spans the real axis there is no other choice than using a L_2 norm for the loss function (Root Mean Square error). This is (predicted start time - true start time)^2 + (predicted end time - true end time)^2 + (predicted average power - true average power)^2. The total loss function is the sum of the loss function for all the sample in a mini-batch.

4.7 Optimizer

We have tried various optimizer to find the best one. We used a classical Stochastic Gradient Descent to update the weights where we feed one mini-batch chosen randomly to the neural network and then update each weight

$$w_j = w_j - \eta \frac{\partial L}{\partial w_j} \tag{5}$$

where L is the loss function evaluated for the given mini-batch. The gradient of the loss function is calculated using the backpropagation algorithm (not detailed here for simplicity). At each epoch we decrease the learning rate η to allow the algorithm to converge towards a local minimum.

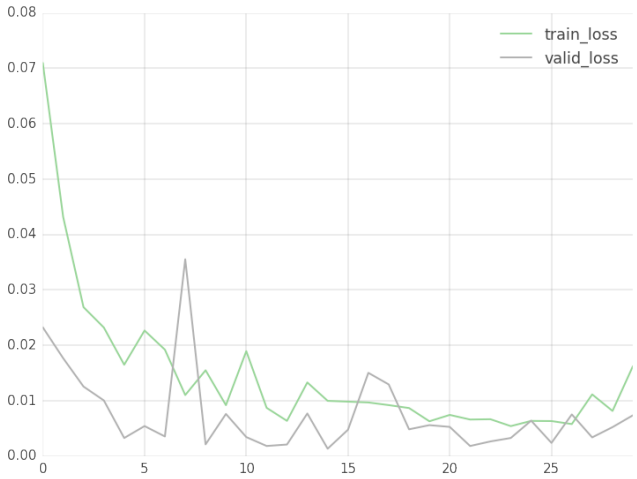


Figure 2: Training and validation loss during training of the ConvNet

We tried a variation of SGD by using the momentum method. This method has some physical interpretation where μ is the friction coefficient. In this case the weights are update using

$$w_j = w_j + \mu v - \eta \frac{\partial L}{\partial w_j} \tag{6}$$

where v is the velocity. An other tested implementation is the Nesterov momentum in which, at a given position in the landscape of weight we look one step ahead with the momentum and then evaluate the gradient there to calculate the new value of the weight. A pseudo code for this method is provided in the notebook (`https://github.com/tperol/am207-NILM-project/blob/master/Report_convnet.ipynb`). We found by experimenting that the best optimizer is Adam (http://arxiv.org/pdf/1412.6980v8.pdf). A pseudo code for this optimizer is provided in the notebook.

4.8 Training and Validation losses

We trained on GPU the network detailed earlier. We also experimented a network with two convolutional layers (see notebook). However it did not improve significantly the results. The training and validation losses are shown in Figure 3. We stop the network after 20 epochs but the network was still learning ! The training is approximately 1 hours long on a GPU.

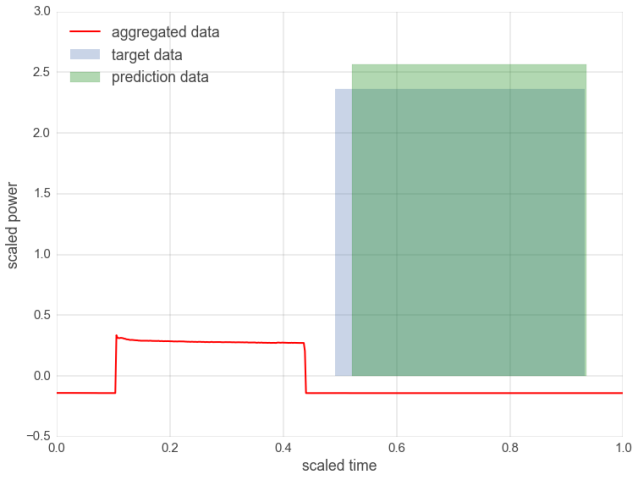
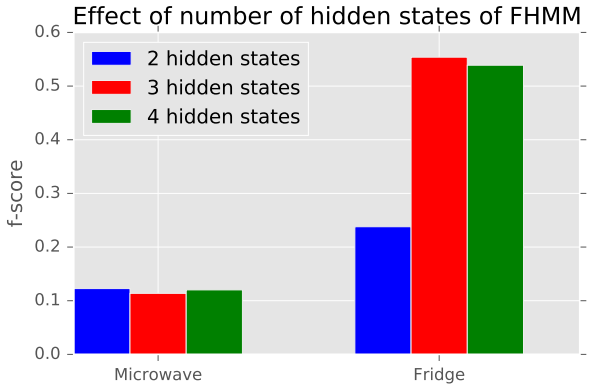


Figure 3: Example of input and output of the ConvNet algorithm.

5 Results and Discussion



We find that for the FHMM, increasing the number of hidden states from 2 to 3 improved the f1-score for fridge, but had little effect on the microwave. Further increasing the number of hidden states to 4 had only a small effect for both fridge and microwave. An FHMM with 2 hidden states will have one state representing the appliance being off and one state representing the appliance being one. The third state can represent the appliance operating at a lower power level, such as spin and fill cycles of a washing machine. In this case of the microwave, the appliance likely operates only at one power setting and increasing the number of hidden states does not improve the model.

6 Conclusion

References