UNIVERSITY OF CALIFORNIA

SANTA CRUZ

A STUDY OF ENERGY DISAGGREGATION USING DEEP LEARNING

A thesis submitted in partial satisfaction of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER ENGINEERING

by

Abhiram Madenur Venkatesha

December 2018

Table of Contents

List of Figures

List of Tables

Abstract

A Study of Energy Disaggregation Using Deep Learning

by

Abhiram Madenur Venkatesha

Energy disaggregation estimates appliance-by-appliance electricity consumption from a single meter that measures the total energy consumed in a house. Recently, deep neural networks have driven remarkable improvements in classification performance in neighbouring fields such as image classification and automatic speech recognition. In this work, I make use of recurrent neural networks with three datasets (two of which are publicly available) in order to study the energy disaggregation ability of recurrent neural networks.

Chapter 1

Non-intrusive Load Monitoring (NILM)

Non-Intrusive Load Monitoring (NILM) methodologies are used to extract and infer the amount of power consumed by an individual appliance using the aggregated time-series power data. NILM helps provide important feedback to both the user and the electric utility which can help toward energy saving behaviour.

The initial studies on NILM were presented by Hart in the 90's. Hart's [18] work concentrated on residential appliances that were modeled as finite state machines (FSM) with two states (On/Off).

1.1 Appliance Types

Appliances differ in the number of operational states and their power consumption behaviour. Three of the commonly used abstract models namely on/off, multi-state and infinite state appliances are presented below.

1.1.1 On/Off

On/Off appliances include common household appliances such as a toaster or a light bulb.

On/Off appliances draw a constant amount of power during their steady-state operation. Most on/off appliances are purely resistive.

1.1.2 Multi-State

Multi-state appliances have more than one state in which they are actively operating. Each of these appliance's states draws a specific amount of power. A common way to represent multi-state appliances is a finite state machine (FSM) representation.

1.1.3 Infinite State Appliances

Infinite state appliances are the ones in which the set of active states is not observable. As an example, the power consumption of light-dimmers changes continuously with no stepwise change. In comparison to on/off appliances and multi-state appliances that change their state in detectable steps, infinite state appliances show a behavior of drawing a continuously varying amount of power.

An example chart below from Non-Intrusive Load Monitoring: A Review and Outlook by Christoph Klemenjak1 and Peter Goldsborough shows a representation to summarize the power requirements of different appliance types. [24]

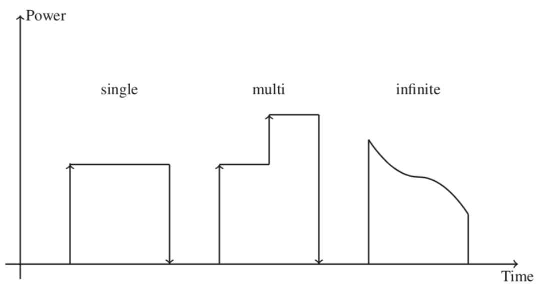


Figure 1.1: A representation to summarize the power consumption of different appliance types

1.1.4 Learning Approaches

Learning approaches for NILM can be divided into supervised and unsupervised techniques. The fundamental difference between an unsupervised and supervised approach is the availability of ground truth data for individual appliances for training the algorithm.

Supervised Learning is the approach in which the algorithm is trained using both the aggregate data and the individual appliance data. Labeled data for training purposes is collected using an intrusive approach by measuring appliance level data.

Unsupervised Learning is the approach in which the algorithm is trained using only aggregate power data. Ground truth appliance-level data is not made use of in order to train the algorithm. In comparison to a supervised approach an unsupervised approach does not help in obtaining better results.

Chapter 2

Deep Learning

Deep Learning is a subset of machine learning and has been the trending research for many tasks such as object recognition, object detection, image classification, speech translation, question-answering systems and speech recognition.

In classic machine learning manual feature engineering is necessary in order to improve the performance of the machine learning algorithm. In the case of deep learning, the algorithm is built from composing multiple stages of non-linear feature maps, thereby reducing the need for feature engineering as the composed feature maps learn the features necessary during training of the algorithm.

Deep learning approaches are useful in NILM because of their feature-learning ability. Although deep learning techniques do not require feature engineering, they do require training using a supervised approach in order to improve performance during inference time. Hence labeled data is important to train a deep learning algorithm in order to achieve high performance during real world testing and application.

2.1 Neural Networks

A neural network is a hierarchical set of weighted summations followed by a non-linearity applied to the output of the weighted sum. A Neural Network (NN) can be represented as directed acyclic graph (DAG) where the nodes or "neurons" are the components that calculate weighted sum of its inputs, adds a bias and then "activates" itself based on a non linearity function. A neural network can be divided into a set of layers namely: input layer, output layer and hidden layers. Input layer is the layer of neurons that accept the input data and the output layer is the final layer at which the output of the NN is obtained. Every layer between input and output layers is termed to be a hidden layer. A hidden layer is a set of neurons in which the activations are calculated to produce hidden feature maps. The activations of any of the layer l is sent to the next layer $l + 1$ through edges. Typically in a simple Neural Network, neurons of every layer l are connected to neurons of layers $l - 1$ and neurons of layer $l + 1$.

The "learning" part of a NN takes place in two steps: the forward pass and the backward pass. **Forward pass** is the step in which the information from the input layer is passed through the hidden layers, and the calculated activations are utilized to produce an output that can be obtained in the output layer. **Backpropogation** is a common term used to define backward pass of a NN. During backpropogation, gradient descent is used to update the weights of the hidden layer depending upon the error calculated between the actual output and the predicted value.

The following sections present different types of NNs specially built to solve specific application domains.

2.2 Recurrent Neural Network

Recurrent Neural Network (RNN) is a type of artificial neural network whose output at any time step depends on the current input and the network's previous hidden state.[13] [20] RNNs have hidden states that store previous information, thereby acting as a memory element. Hence, RNNs perform well when the inputs are sequences such as words in a sentence or time series data.

The training of RNNs happens in a similar manner to backpropogation in an artificial neural network. Backpropogation through time (BPTT) is a technique in which weights are updated by considering the states of all the previous timesteps.

2.2.1 "Vanilla" RNN

A "vanilla" or the simple RNN is the most basic RNN. The most common vanilla RNN used is the Elman Network (shown below): [13]

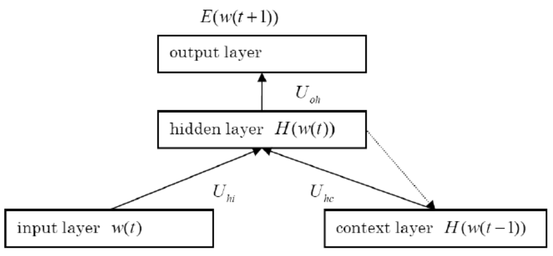


Figure 2.1: Elman Network

The update rule of the parameters is defined in the equation below:

$$h_t = \sigma(x_t W_x + h_{t-1} W_h + b)$$

$$o_t = \sigma(h_t W_h o + b_o)$$

where h_t is the hidden state of the RNN at timestep t, h_{t-1} is the hidden state of the RNN at timestep $t-1$, o_t is the output state of the RNN at timestep t, w_x is the input weight matrix, W_h is the hidden to hidden weight matrix, $W_h o$ is the output to hidden weight matrix.

Vanilla RNNs perform well on short sequences. However as sequences get longer, the issue of vanishing/exploding gradients starts to occur. Vanishing/exploding gradients is the problem in which during backpropogation the gradient values decrease (vanish) or increase (explode) significantly thereby causing the neural network to saturate and rendering it ineffective. A more complex type of RNN known as the Long-Short Term Memory (LSTM) network aims to solve this problem with the help of additional activations and weights, composed as gating mechanisms.

2.2.2 Long-Short Term Memory Networks

Long short-term memory (LSTM) is a type of recurrent neural network and was introduced in 1997 by Hochreiter et. al. [20] However, in the LSTM that Hochreiter et. al formulated, a phenomenon in which the error becomes trapped in memory known as 'error carousel' occurs. Gers et al. [14] introduced peephole connections that reduced the error carousel problem, Gers et al. in his formulation of the LSTM network also introduced the ”forget gates” which led to substantial improvement in the performance of LSTM enabling it to partially or completely reset the hidden state that it deems to be of no value at any time step.

A diagram of a LSTM is shown below [35]

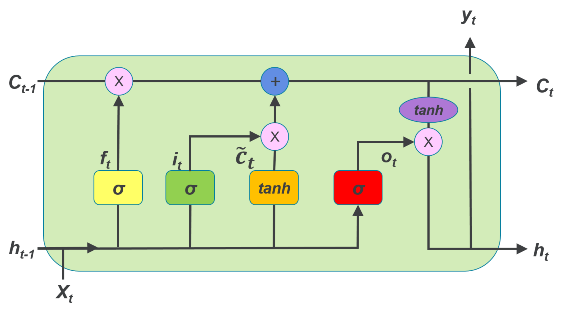


Figure 2.2: Representation of a LSTM Cell

The expressions below represent the update rule for the LSTM network:

$$i_t = \sigma_i(x_t W_{xi} + h_{t-1} W_{hi} + w_c i \odot c_{t-1} + b_i)$$

$$f_t = \sigma_f(x_t W_{xf} + h_{t-1} W_{hf} + w_c f \odot c_{t-1} + b_f)$$

$$c_t = f_t \odot c_{t-1} + i_t \sigma_c(x_t W_{xc} + h_{t-1} W_{hc} + b_c)$$

$$o_t = \sigma_o(x_t W_{xo} + h_{t-1} W_{ho} + w_{co} \odot c_t + b_o)$$

$$h_t = o_t \odot \sigma_h(c_t)$$

2.3 Convolutional Neural Network

Convolutional Neural Network (CNN) is a type of Neural Network that is mainly used in the field of image recognition [29] due to its ability to recognize complex patterns in images while delivering high performance. Convolutional Neural Networks are composed of multiple layers that are made up of any of the layer types namely: convolution layer, pooling layer, the activation layer and the fully connected layer.

- Convolution layer: This layer is made up of one or more kernels (filters). Each kernel extracts features from the image. Multiple kernels ensure that more complex features are recognized in a single layer. The kernels act in a similar way as that of hand coded kernels (edge detectors etc.) but are learnt during the backpropogation stage of the training.

- Pooling layer: The output of the convolutional layer is generally a high dimensional feature map. In order to reduce the amount of data to be processed in further layers, the feature maps are downsampled. This in turn helps the network be translationally and rotationally invariant thereby increasing performance. Max-pooling and average pooling are the most common types of pooling.

- Activation Layer is the layer at which an elementwise non-linear activation is applied to the feature maps in order to reduce the vanishing gradient problem. Non linear activations also help improve computational speed and induce sparsity in the feature maps. Rectified Linear Unit (ReLU) is the most common type of activation function used and is defined as $f(x) = max(0, x)$ (fundamentally a thresholding function)[31]. A variant of ReLU, called the Leaky ReLU may also be used. The only difference is the lower threshold may not be 0 in Leaky ReLU.

- Fully Connected Layer is generally the final layer and is used to make predictions using the weighted sum of the feature maps in the previous layer. A fully connected layer, also known as the dense layer is made up of an artificial neural network. The output of the fully connected layer is generally the prediction of the Convolutional Neural Network.

A representation of a Convolutional Neural Network is as shown below: [11]

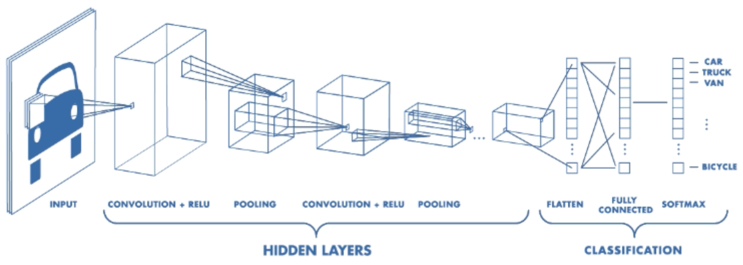


Figure 2.3: Representation of a Convolutional Neural Network

2.4 Optimization Methods

Optimization in the context of machine learning and deep learning is the minimization of an objective function known as the loss function in order to improve performance of the algorithm. Optimization in a neural network is achieved by updating the hyper-parameters of the network in proportion to the error.

2.4.1 Stochastic Gradient Descent

Stochastic Gradient Descent (SGD) is one of the most used backpropogation approaches to train a neural network. The hyperparameters of the neural network must be updated in proportion to the error generated by making a prediction on each training example. While training larger and more complex neural networks. The update equation of SGD is:

Algorithm 1: Stochastic Gradient Descent (SGD)

```
Input: Training data $S$, regularization parameters $\lambda$, learning rate $\eta$, initialization $\sigma$
Output: Model parameters $\Theta = (w_0, \mathbf{w}, \mathbf{V})$
$w_0 \leftarrow 0$; $\mathbf{w} \leftarrow (0, \dots, 0)$; $\mathbf{V} \sim \mathcal{N}(0, \sigma)$;
repeat
    for $(x, y) \in S$ do
        $w_0 \leftarrow w_0 - \eta(\frac{\partial}{\partial w_0} l(y(\mathbf{x} \mid \Theta), y) + 2\lambda^0 w_0)$;
        for $i \in \{1, \dots, p\} \wedge x_i \neq 0$ do
            $w_i \leftarrow w_i - \eta(\frac{\partial}{\partial w_i} l(y(x \mid \Theta), y) + 2\lambda^w_\pi w_i)$;
            for $f \in \{1, \dots, k\}$ do
                $v_{i,f} \leftarrow v_{i,f} - \eta(\frac{\partial}{\partial v_{i,f}} l(y(x \mid \Theta), y) + 2\lambda^v_{f,\pi(i)} v_{i,f})$;
            end
        end
    end
until stopping criterion is not met;
```

SGD is not seen to perform well because of less stable convergence and the computational time it takes to achieve convergence close to the global minimum without getting stuck at local minima.

2.4.2 Batch Gradient Descent

Batch Gradient Descent or commonly known as ”mini-batch” gradient descent is a variant of SGD in which the updating of hyperparameters of the neural network takes place in proportion to the average of the errors caused by a specific number of training examples. The number of training examples after which the backpropogation takes place is called the batch size and it is a hyperparameter that is normally decided by the user. A higher batch size results in increased memory consumption of the processing component.

2.4.3 Momentum

SGD has trouble with local optima and oscillates across the ravines (areas where the surface curves steeply in one dimension compared to others) and makes minimal progress toward the optimum.

The below image shows the behaviour of SGD without momentum: [9]

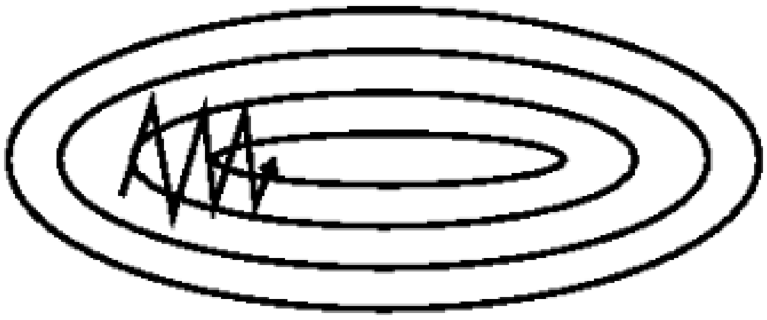


Figure 2.4: SGD without momentum

Momentum is a method that helps the SGD algorithm by accelerating SGD in the relevant direction and reduces the oscillations. Momentum achieves this by adding to the current update a fraction γ of the previous update value. [9]

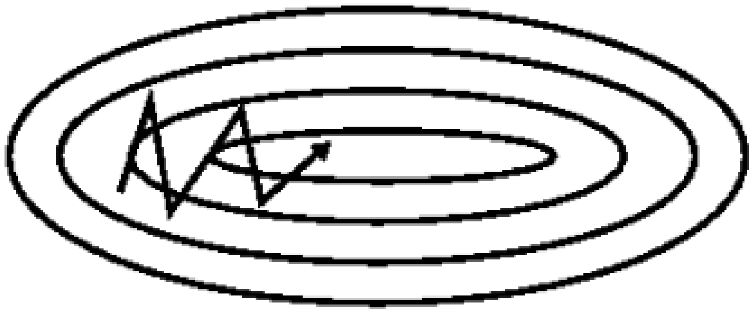


Figure 2.5: SGD with momentum

$$v_t = \gamma v_{t-1} + \eta \nabla_\theta J(\theta)$$

$$\theta = \theta - v_t$$

The momentum term γ is normally set to a value close to 0.9.

2.4.4 ADAM optimization

Adaptive Moment Estimation (ADAM) [12] [17] is a method that computes the adaptive multipliers to control the update of hyperparameters. It stores exponentially decaying average of past squared gradients v_t and exponentially decaying average of past gradients m_t. The decaying averages are computed as follows:

$$\begin{aligned} m_t &= \beta_1 m_{t-1} + (1-\beta_1) g_t \\ v_t &= \beta_2 v_{t-1} + (1-\beta_2) g_t^2 \end{aligned} \tag{2.1}$$

m_t is the estimate of the first moment (mean) of the gradient and v_t is the estimate of the second moment (uncentered variance) of the gradient. Since m_t and v_t are initialized as zero vectors, during the inital steps of training they remain biased toward zero. This is compensated by using the bias corrected m_t and v_t expressions:

$$\begin{aligned} \hat{m}_t &= \frac{m_t}{1-\beta_1^t} \\ \hat{v}_t &= \frac{v_t}{1-\beta_2^t} \end{aligned} \tag{2.2}$$

The Adam update rule is thus defined as:

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \hat{m}_t$$

The authors propose default values of β_1 to be 0.9 and β_2 to be 0.999 and ϵ to be 10^{-8}

2.5 Initialization Methods

Initialization of hyperparameters, mainly weights of a neural network help improve performance and achieve faster convergence thereby reducing computational costs. Poor initialization may lead the neural network to get caught in local minima and increase convergence times. Initializing weights with zeros causes an effect of symmetry in a neural network, during which the output of every neuron in every layer is zero, thereby rendering the network ineffective. Some of the better initialization approaches is presented in this section.

2.5.1 Random Initialization

In order to break the symmetry that is caused by zero initialization of weights, weights can be randomly initialized using small random numbers not equal to zero. Example: $W \sim \mathcal{N}(0, \sigma)$ Using this expression, every neurons weight vector is initialized as a random vector sampled from a multi-dimensional gaussian.

2.5.2 Xavier Initialization

Glorot and Bengio (2009) [15] suggested to initialize the weights from a distribution with zero mean and variance:

$$Var(W) = \frac{2}{n_{in} + n_{out}}$$

where n_{in} and n_{out} are respectively the number of inputs and outputs of a layer. Xavier initialization is technique that tries to make the variance of the outputs of a layer to be equal to the variance of its inputs. In the paper Glorot and Bengio considered logistic sigmoid activation function, which was the default choice at that moment. Later, ReLU started to become the more common choice for activation function in comparision to the the sigmoid activation since it allowed to solve vanishing/exploding gradients problem.

Consequently, a new initialization technique that applied the same idea (balancing of the variance of the activation) to this new activation function was proposed by Kaiming He at al. [19] It is now referred to as He initialization.

Chapter 3

Related Work

Academic work on energy disaggregation began with the work of Hart et. al during 1980s. Preliminary approaches were mostly event detection based approaches using real and reactive power.

Kolter et. al [26] made use of Factorial Hidden Markov Models (FHMMs) to approach the energy disaggregation problem. The inference was difficult and the algorithm tends to easily get stuck in local minimas. They constrained the model at inference by making the algorithm consider only one appliance state change at a time. In order to avoid local minimas, they developed an effective inference step and achieved good results.

Parson et al. [32] also utilized HMMs for energy disaggregation and developed an algorithm without using sub-metered data from appliances. They made use of prior models of appliance types that were tuned using the aggregate signal and achieved results that were comparable with systems that used sub-metered data.

Kelly and Knottenbelt [23] used deep learning techniques and achieved very good results. They made use of appliance signatures in order to extract activations from the aggregate data and train the algorithm. They make use of 3 different methods and compare them. Recurrent Neural Networks with an LSTM variant seem to not perform well in their work.

In the current work, the training methodologies are adopted from the work Kelly et. al in order to prepare the dataset to train the deep learning algorithm. The algorithm (recurrent neural network architecture) used is in this work is different from the work of Kelly et. al.

3.1 Datasets

3.1.1 REDD

Reference Energy Disaggregation Data Set (REDD) [25] is a publicly available dataset that contains detailed power usage information from several homes. The data is specifically geared towards the task of energy disaggregation.

REDD consists of both whole-home and device level specific electricity consumption for a number houses over a duration of several months.

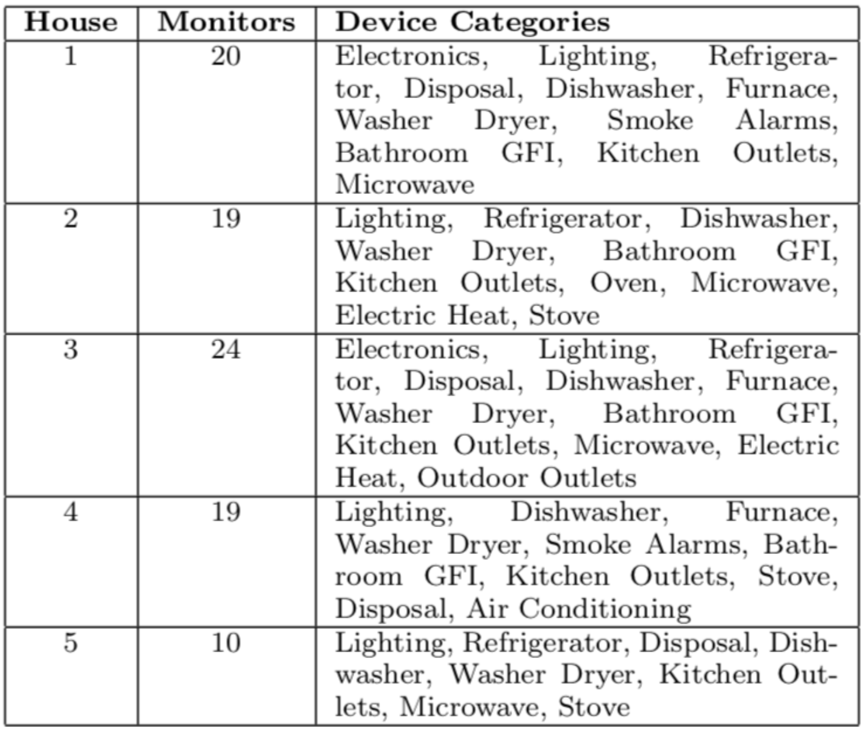
For each of the houses REDD contains information about:

- The whole home electricity signal (current monitors on both phases of power and a voltage monitor on one phase) recorded at a high frequency (15kHz)
- Up to 24 individual circuits in the home, each labeled with its category of appliance

or appliances, recorded at 0.5 Hz

- Up to 20 plug-level monitors in the home, recorded at 1 Hz, with a focus on logging electronics devices where multiple devices are grouped to a single circuit.

The image below shows a table containing information regarding the monitors and device categories at each home. (Table 1 in original paper)



House	Monitors	Device Categories
1	20	Electronics, Lighting, Refrigerator, Disposal, Dishwasher, Furnace, Washer Dryer, Smoke Alarms, Bathroom GFI, Kitchen Outlets, Microwave
2	19	Lighting, Refrigerator, Dishwasher, Washer Dryer, Bathroom GFI, Kitchen Outlets, Oven, Microwave, Electric Heat, Stove
3	24	Electronics, Lighting, Refrigerator, Disposal, Dishwasher, Furnace, Washer Dryer, Bathroom GFI, Kitchen Outlets, Microwave, Electric Heat, Outdoor Outlets
4	19	Lighting, Dishwasher, Furnace, Washer Dryer, Smoke Alarms, Bathroom GFI, Kitchen Outlets, Stove, Disposal, Air Conditioning
5	10	Lighting, Refrigerator, Disposal, Dishwasher, Washer Dryer, Kitchen Outlets, Microwave, Stove

Figure 3.1: REDD information

Attached below are images of 2 plots of house_1 data that I created as a part of CMPS 263 coursework at UC Santa Cruz under the guidance of Prof. Suresh Lodha. The web-link [1] has 2 more plots to get an overview of the day-to-day power consumption:

[1] `https://abhirammv.github.io/d3PowerData/`

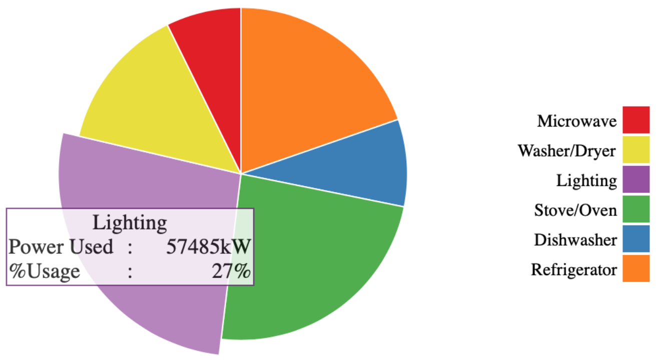


Figure 3.2: Pie Chart depicting the total power consumption of house_1

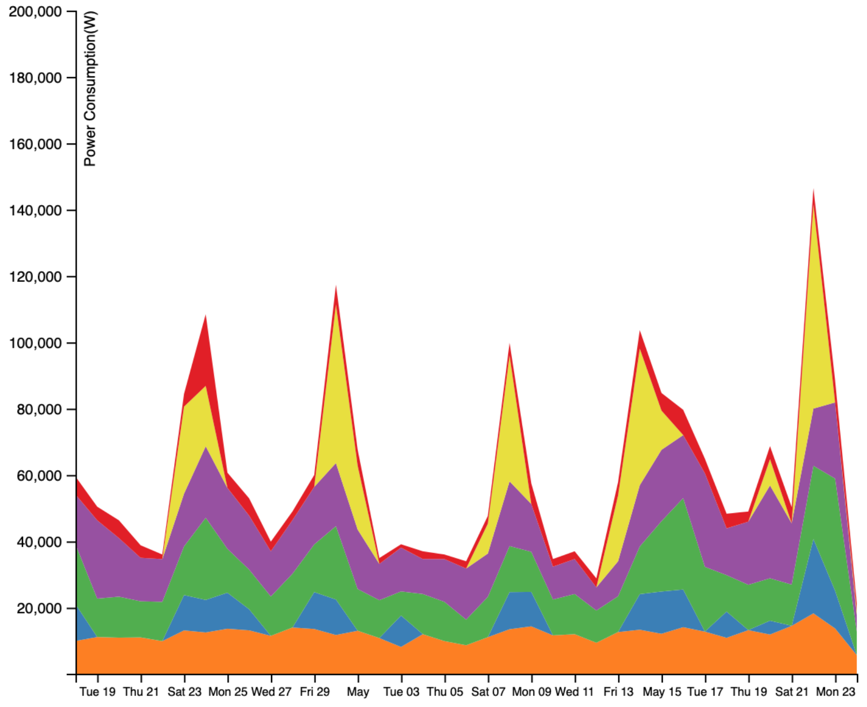


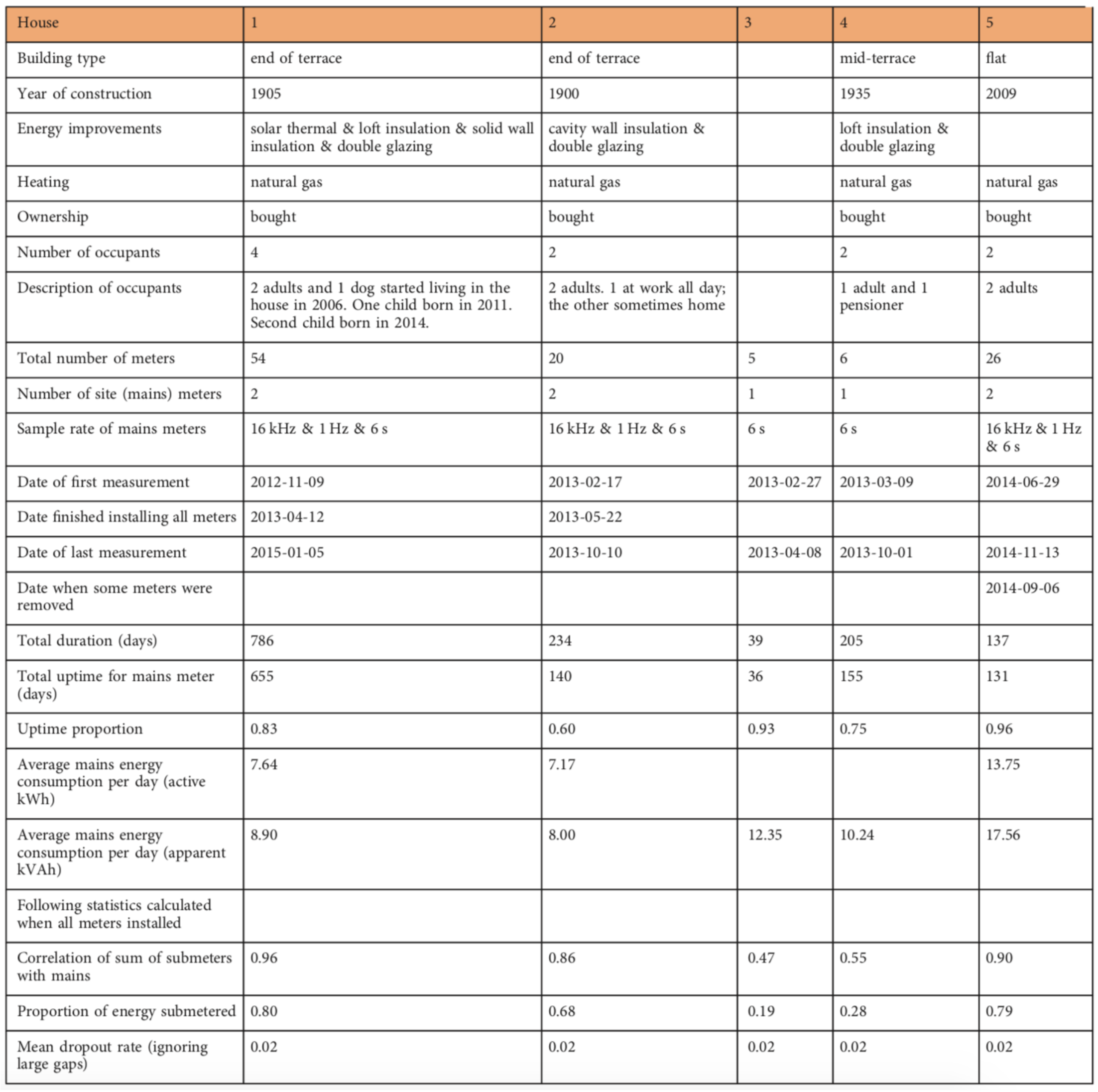
Figure 3.3: Stacked Area Chart depicting the overview of the power consumption of house_1 for the duration of the data

3.1.2 UK-DALE Dataset

UK-DALE Dataset [22] is a publicly available dataset that uses 16 KHz of sampling rate for whole-house and 1/6 Hz sampling for individual appliances. It is the first publicly available UK data with the aforementioned temporal resolution. The dataset consists of measurements from 5 houses, out of which one of them was for a duration of 655 days.

Every six seconds the active power drawn by individual appliances and the whole-house apparent power demand was recorded. Additionally, in three houses, whole-house voltage and current at were sampled at 44.1 kHz and was later down-sampled to 16 kHz for storage and also calculated the active power, apparent power and RMS voltage at 1 Hz.

The image below shows a table containing details regarding the residence and the measurements. (Table 1 in original paper)



House	1	2	3	4	5
Building type	end of terrace	end of terrace		mid-terrace	flat
Year of construction	1905	1900		1935	2009
Energy improvements	solar thermal & loft insulation & solid wall insulation & double glazing	cavity wall insulation & double glazing		loft insulation & double glazing	
Heating	natural gas	natural gas		natural gas	natural gas
Ownership	bought	bought		bought	bought
Number of occupants	4	2		2	2
Description of occupants	2 adults and 1 dog started living in the house in 2006. One child born in 2011. Second child born in 2014.	2 adults. 1 at work all day; the other sometimes home		1 adult and 1 pensioner	2 adults
Total number of meters	54	20	5	6	26
Number of site (mains) meters	2	2	1	1	2
Sample rate of mains meters	16 kHz & 1 Hz & 6 s	16 kHz & 1 Hz & 6 s	6 s	6 s	16 kHz & 1 Hz & 6 s
Date of first measurement	2012-11-09	2013-02-17	2013-02-27	2013-03-09	2014-06-29
Date finished installing all meters	2013-04-12	2013-05-22			
Date of last measurement	2015-01-05	2013-10-10	2013-04-08	2013-10-01	2014-11-13
Date when some meters were removed					2014-09-06
Total duration (days)	786	234	39	205	137
Total uptime for mains meter (days)	655	140	36	155	131
Uptime proportion	0.83	0.60	0.93	0.75	0.96
Average mains energy consumption per day (active kWh)	7.64	7.17			13.75
Average mains energy consumption per day (apparent kVAh)	8.90	8.00	12.35	10.24	17.56
Following statistics calculated when all meters installed					
Correlation of sum of submeters with mains	0.96	0.86	0.47	0.55	0.90
Proportion of energy submetered	0.80	0.68	0.19	0.28	0.79
Mean dropout rate (ignoring large gaps)	0.02	0.02	0.02	0.02	0.02

Figure 3.4: UK-DALE dataset details

3.1.3 SEADS Dataset

Smart Energy Analytic Disaggregation System (SEADS) [7] dataset is a private dataset provided by Dr. Patrick Mantey and Dr. Ali Adabi and is stored on a local server in the CITRIS graduate Lab at UC Santa Cruz.

SEADS provides capabilities not found in other energy monitoring products. The goal of the SEADS is to design and build smart sensors powered by intelligent software capable of collecting and disaggregating information regarding energy usage from individual appliances in real time. SEADS advanced algorithm optimizes the cost of managing microgrids. SEADS aims at providing an efficient control platform for microgrids based on information available through various sensors.

One of the core parts of Dr. Adabi's doctorate thesis [7] ("Economical Real-time Energy Management For Microgrids via NILM and With User Decision Support") concentrates on the SEADS system's hardware hardware and software designs and implementations.

The SEADS hardware unit is capable of sampling up to 65kHz and 24 bits of data. The optimal sampling rate range is found to be 4kHz to 8kHz according to Dr. Adabi's work.

The data [6, 7] is available from the following panels:

- Power G - Generator
- Power S - Solar
- Panel 1 - HVAC, heat pumps and some lights & outlets
- Panel 2 - Water Heater, lights & outlets in bathrooms and bedrooms
- Panel 3 - Kitchen, laundry, garage and some outlets & lights in adjacent rooms

There are 7 channels of data for panel 3. For this work I have used the harmonic data of channels 3, 4, 5 and 6 from Panel 3 (Kitchen Panel). The data is obtained by using an API endpoint served to access the data froma NoSQL (pseudo-JSON response type) database on the local server.

The image below shows the radial heat map representation of Panel 3 energy consumption data as a part of the work done by Sharad et. al. in collaboration with me, Dr. Adabi and the SEADS laboratory. The work of Sharad et. al. includes visualization of the rest of the panels as well and can be found at the web-link [2]:

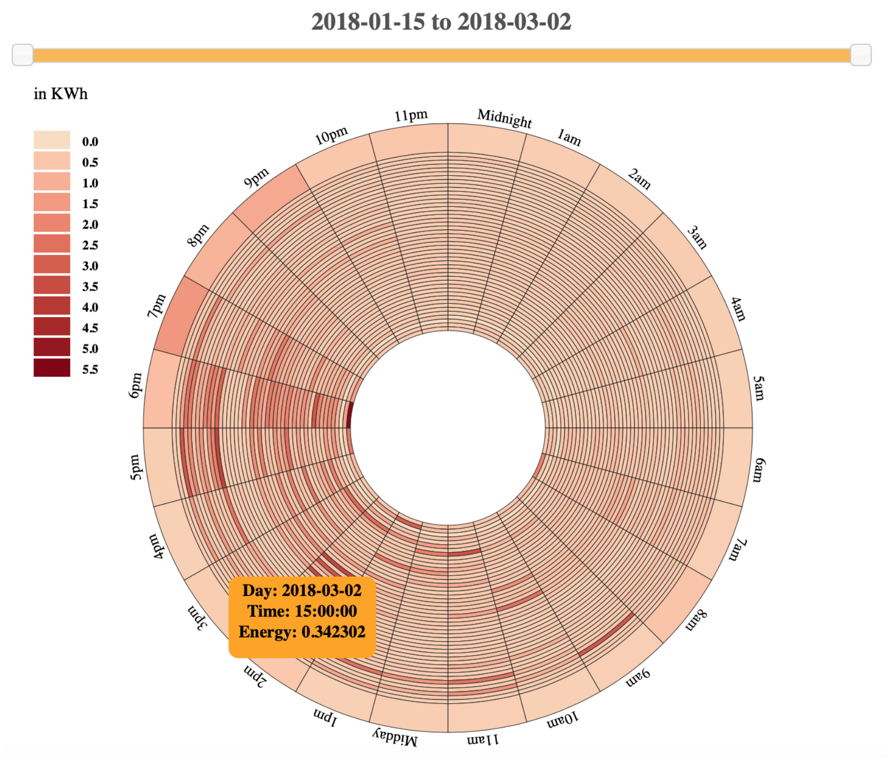


Figure 3.5: Panel 3 of SEADS data visualized over the depicted period

[2]https://sharad97.github.io/SEADS_Visualization/FInal/index.html

Chapter 4

Implementations

The present work focuses on energy disaggregation with a deep learning approach. I made use of a LSTM neural network architecture to better encode the sequential data in order to make predictions. In the following sections I compare and present the results that were obtained for the three datasets namely: REDD, UK-DALE dataset and the SEADS dataset.

A Long Short Term Memory Recurrent Neural Network architecture is used with the preprocessed datasets (REDD, UK-DALE, SEADS). In the following section, the approaches taken to preprocess datasets and the architectures of the neural network are presented.

The task of disaggregation using neural networks involves two key steps namely: training and testing. Section 4.1 describes the preprocessing of the data that has to be done in order to efficiently train and test the neural network architectures. Section 4.2 describes the architecture of the LSTM network used along with the initializers and the regularization techniques used. Section 4.3 briefly describes the initialization of hyperparameters to improve the speed of convergence of the neural network to a global minima of the loss function. Section 4.4 describes the training and testing metrics used.

All the code is written in the Python v3.4 [4] programming language. Pandas [30] and Numpy [21] are the packages that are used to prepare the training data along with the NILMTK [3] package. The neural network algorithm is implemented with the Keras [2] package with TensorFlow [5] backend and the data is efficiently fed into the network using the TensorFlow's new data 'tf.data()' module. Compute instances of Amazon Web Sevices [1] (a cloud platform) was utilized through a virtual machine compute instance running the Ubuntu Operating System 14.04 with 4 NVIDIA K80 GPUs that help in parallelizing the training of the neural network.

4.1 Data Preprocessing

The input to the neural network is a window (time varying samples) of the aggregate power demand given that the window has a 50% probability that it is from a time at which the target appliance is either active or inactive. The 50% probability ensures a balanced dataset which is a requirement to improve the efficacy of the neural network algorithm.

The target label or the ground truth label is a vector of the presence or absence of activation (device or channel); if present, the label vector of the activation (label vector of the channel or label vector of the device), the start and end times of the activation.

The ground truth label as a vector was inspired by the bounding box prediction made by the You Only Look Once (YOLO) [33] algorithm used to detect the presence or absence of an object in an image along with the location of the object defined by rectangular co-ordinates, if the object is detected and the present flag is set.

4.1.1 REDD & UK-DALE

REDD dataset and UK-DALE dataset were preprocessed using a process similar to that of Kelly et. al. The first step was to extract activations of the appliances. Activations here refers to power consumed by the appliance during one cycle of that appliance being active. A variant of NILMTK's 'get_activations()' function was implemented. The variation made sure that all the information necessary to prepare the target vectors was extracted from the dataset.

In order to prepare the training data as examples of $(input, label)$ windows of aggregate data were obtained. In a similar manner as implemented in Kelly et. al. all the activations of the target appliance are located in the home's submeter data for each target appliance. For each training example, the algorithm makes a decision whether to include the details of the target appliance or not with a probability of 50%. If the code decides to include the target details, a random window of activation of the device is selected and the details to construct the target vector are obtained. The corresponding window of aggregate data is selected as the input. If the code decides to not include the target appliance, a random window of aggregate data in which there are no activations of the appliance is selected.

80% of the total data was used as training data and 20% is used as testing data.

Houses 1 and 2 were used from both REDD and UK-DALE datasets because of similar appliances among the datasets in order to ensure valid comparision of results. The appliances used in REDD Dataset are:

- Refrigerator
- Dishwasher
- Washer Dryer
- Microwave

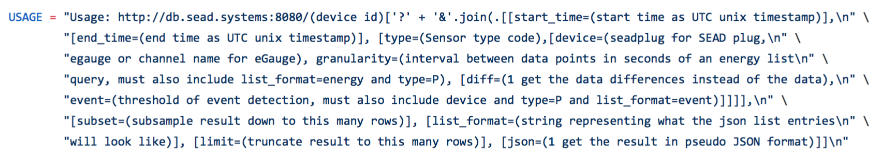
The appliances used in UK-DALE Dataset are:

- Fridge
- Dish washer
- Washing machine
- Microwave

4.1.2 SEADS

SEADS dataset was available on the local server in the lab and was accessed through the API endpoint provided.

The image below presents the structure of the API



```
USAGE = "Usage: http://db.sead.systems:8080/(device id)['?' + '&'.join(.[[start_time=(start time as UTC unix timestamp)],\n" \
        "[end_time=(end time as UTC unix timestamp)], [type=(Sensor type code),[device=(seadplug for SEAD plug,\n" \
        "egauge or channel name for eGauge), granularity=(interval between data points in seconds of an energy list\n" \
        "query, must also include list_format=energy and type=P), [diff=(1 get the data differences instead of the data),\n" \
        "event=(threshold of event detection, must also include device and type=P and list_format=event)]]]],\n" \
        "[subset=(subsample result down to this many rows)], [list_format=(string representing what the json list entries\n" \
        "will look like)], [limit=(truncate result to this many rows)], [json=(1 get the result in pseudo JSON format)]]\n"
```

Figure 4.1: The structure of the API used to access SEADS data

It was a bit tricky to efficiently query and structure it to be able to input to the neural network algorithm. The first 50 harmonics were used to make predictions. Data from channels 3, 4, 5 and 6 queried over a period of 2 months day by day and were stored in a different comma-seperated values (csv) file format files. The aggregate values were computed for the same time duration for channels 3, 4, 5 and 6 and were stored in a separate comma-seperated values (csv) file.

Once the csv file for aggregate and submetered channel data were ready, the variant of the 'get_activations()' function was implemented in a similar manner to that of REDD and UK-DALE datasets. 100% of the training data was used for training the neural network algorithms.

Testing of the algorithm for SEADS dataset was done for 15 days worth of data. In order to simulate the real-time scenario of testing with data coming in streams, each day's data was fed day-by-day through a function. The function was implemented to query data of channels 3, 4, 5 and 6, parse and aggregate the data. The aggregated dataset for each day was used to test the neural networks. After all the predictions were made the performances of the neural networks was averaged out over 15 days.

This method of testing by utilizing streams of data per day provides a sense of the inference time that the algorithm would take in order to make the predictions. This method also simulates the amount of data available to make predictions at the end of the day and the computing required to parse the available data.

4.2 Neural Network Architectures

Two different versions of neural networks were constructed. The first version or 'LSTM v1', was a relatively simple network in comparison to the second LSTM version or 'LSTM v2'.

LSTM v1 consists of:

1. Input (of length MAX_WINDOW_SIZE)
2. LSTM layer with 64 hidden units
3. Fully connected layer with input size 64 units and 32 output units
4. Fully connected with 32 input units and 4 output units

LSTM v2 consists of:

1. Input (of length MAX_WINDOW_SIZE)
2. LSTM layer with 128 hidden units with recurrent dropout
3. LSTM layer with 64 hidden units with recurrent dropout
4. Fully connected layer with input size 64 units and 32 output units with dropout

5. Fully connected with 32 input units and 4 output units

LSTMv2 took more longer time to train in comparision to LSTMv1 due to its increased complexity.

For each dataset, and for each architecture one network per target (appliance or channel) was performed in a manner similar to that of Kelly et. al. Training a different neural network per target appliance ensures that when aggregate data is fed in during testing, only one of the neural networks corresponding to the target appliance that is active at the time would be producing a high output probability. This can help in detecting multiple active appliances at the same time.

4.3 Hyperparameters

The initial settings of the hyperparameters prior to training are briefly described in this section. Initial values of hyperparameters play a significant role in determining the rate of convergence and the effectiveness of the neural network. The following bullet points describe the initial values of *number_of_epochs*, *weights*, *biases* and the *learning_rate*

- The number of times the algorithm must process the entire training dataset or *number_of_epochs* of training was not set to a value initially. I trained it in increments of 500 steps each and upon measuring the test accuracies of each 500-step increment, I later decided on the model trained until 5000 epochs. This is further explained in the Results and Conclusion chapter of this work.

- He initialization is used for initializing the *weights* of the neural network. *biases* of the neural network are initialized with a random normal distribution with a mean of 0.0001 and standard deviation 0.003.

- The value that controls the update of the parameters of the neural network with respect to the gradient or *learning_rate* is initialized at 0.45 and is set to decay for every 100 steps of the 5000 steps of training. $tf.train.exponential\_decay()$ is a built-in tensorflow function that helps to achieve the decay.

4.4 Metrics

4.4.1 Loss Functions

A combination of two different types of loss functions are used to train the algorithm using the batch gradient descent method. The cross entropy loss function is used to train the algorithm to predict the presence or absence of an appliance or a channel in the data and if the object is present, to predict the label of the object. A square loss is used to train the algorithm to predict the start and end time bounds.

4.4.2 Test Metrics

A number of test metrics exist in order to measure the effectiveness of the neural network during testing such as accuracy, area under the ROC curve, F1 score etc. [1].

[1] `https://machinelearningmastery.com/metrics-evaluate-machine-learning-algorithms-python/`

In the present work accuracy is the metric used to determine how well the algorithm performs during validation and testing. Accuracy is the percentage of number of correct predictions vs. total number of predictions. Accuracy is the most commonly used metric for classification problem. Since the implementation makes sure that the dataset is balanced, accuracy is a simple metric that can be used. The reasoning behind the usage of the accuracy metric is the interpretation of the value directly.

Chapter 5

Results and Conclusion

The results of performances of the LSTM architectures LSTMv1 and LSTMv2 on REDD, UK-DALE and SEADS datasets are presented here. LSTMv2 shows an improvement in performance on the SEADS dataset and has the highest test accuracy of 92.3%.

Training of both the neural networks were performed using the loss function described in the previous section. Adam optimization technique was used to update the hyperparameters in order to achieve faster convergence to the global minima of the loss function in comparison to stochastic gradient descent algorithm.

The training was conducted in 500 step increments and the intermediate models were saved in order to test the model's accuracy. The models which were trained for 5000 epochs were later used since the models trained for more than 5000 epochs showed signs of over-fitting to the training data.

Testing of the neural networks was conducted after each 500 steps of training.

Tensorflow's feature of checkpointing and saving a model helped achieve the intermediate testing step with ease. The results of both intermediate and final testing are presented in this chapter.

5.1 Training

Graphs of the loss functions vs. number of epochs are presented below in figures 5.1 and 5.2.

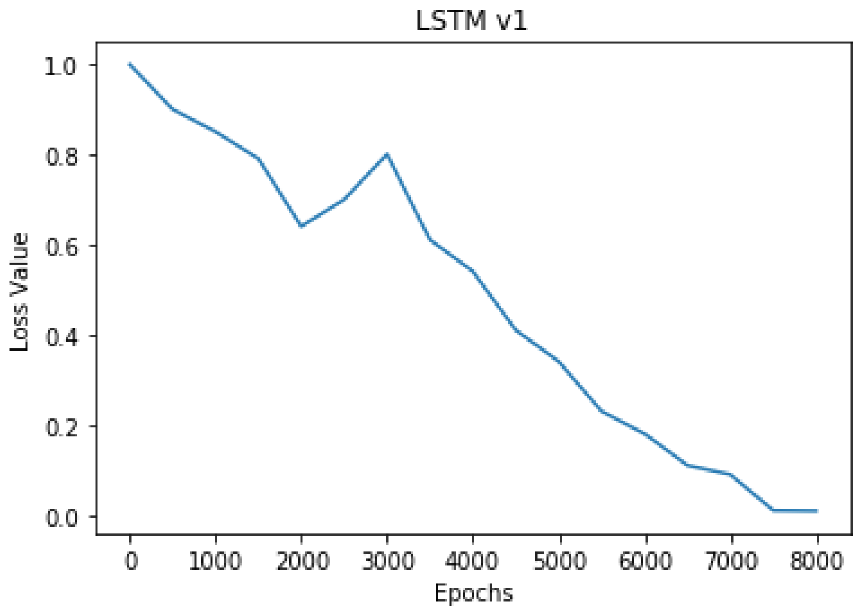


Figure 5.1: LSTMv1 loss value vs. number of epochs

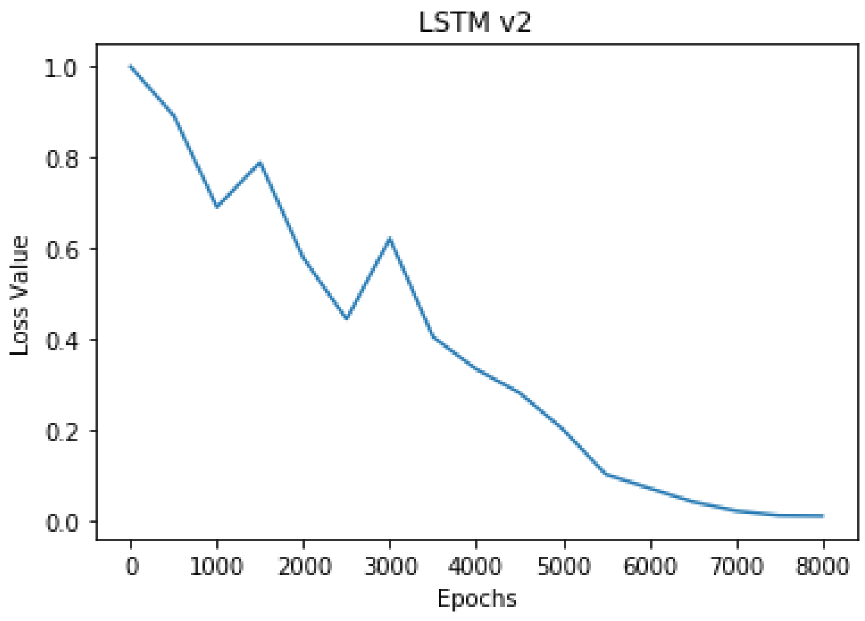


Figure 5.2: LSTMv2 loss value vs. number of epochs

It can be seen that in LSTMv1 the loss at the end is higher in comparison to loss of LSTMv2 at the end of the training phase.

In my hypothesis this disparity can be attributed to the relatively low complexity of LSTMv1 and it's lower ability to extract and store feature representations of a higher complexity in relation to LSTMv2.

In both LSTMv1 and LSTMv2 the losses seem to be trending toward a linear trend with the loss values starting at a higher value during the start of the training and a relatively lower value at the end of the training phase.

Testing accuracies were measured during training by checkpointing the models. This was done since I had not decided on the number of epochs of training initially. It can clearly be seen that the accuracy peaks and then starts to fall when the network starts to overfit. The graphs of the intermediate accuracies of LSTMv1 and LSTMv2 are presented in figures 5.3 and 5.4.

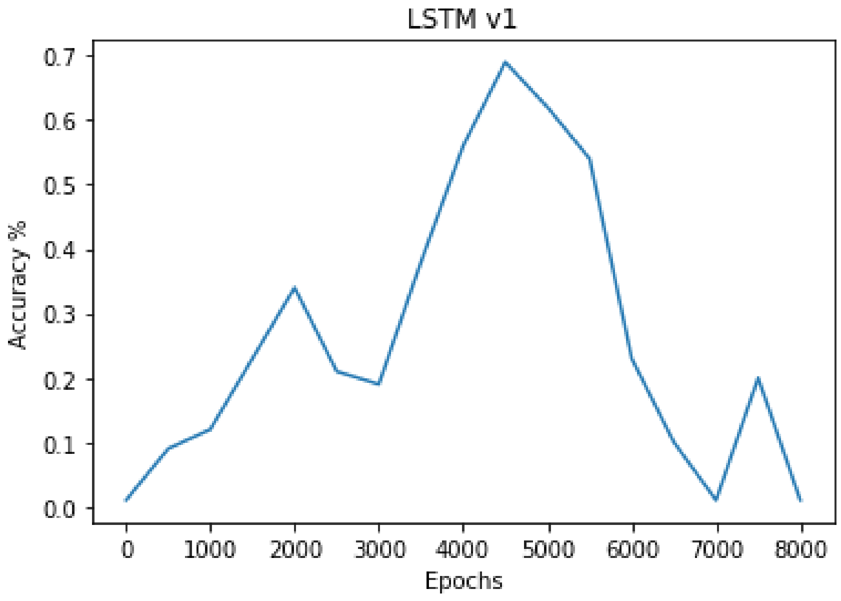


Figure 5.3: Intermediate LSTMv1 accuracy vs. number of epochs

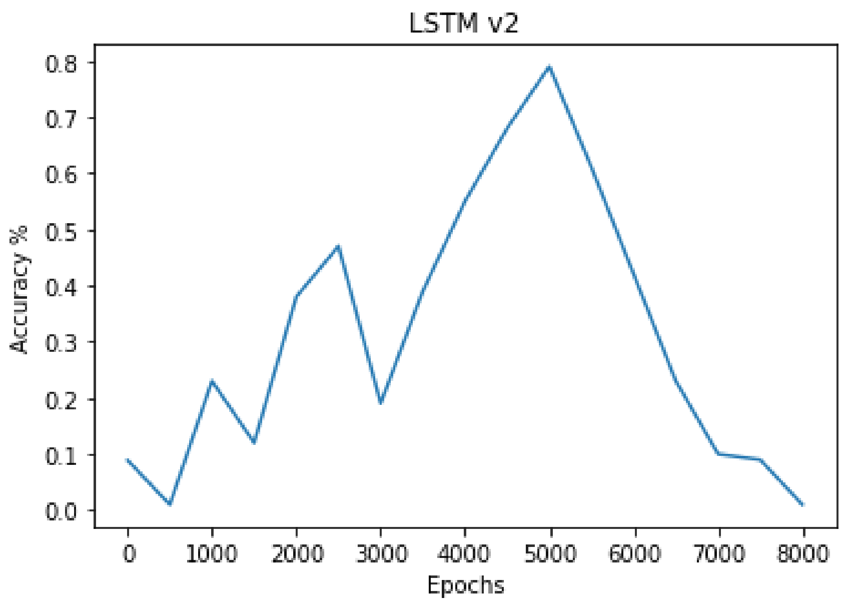


Figure 5.4: Intermediate LSTMv2 accuracy vs. number of epochs

It can be seen that after 5000 epochs of training, the decrease in losses (as shown above in figures 5.1 and 5.2) correspond to decrease in accuracy. Thus it can be seen that the network starts to overfit to the training data and would later not be viable for testing due to it's decreased accuracy. Hence the trained model at 5000 steps is considered the "best" model and is used to present the results in the Results section of this chapter.

5.2 Results

Table 5.1 presents validation accuracies obtained by LSTMv1 and LSTMv2 on REDD, UK-DALE and SEADS validations datasets.

Table 5.1: Validation Accuracies of the LSTM architectures on the datasets

	LSTMv1	LSTMv2
REDD	81.2	88.1
UK-DALE	68.4	72.2
SEADS	82.4	94.2

Table 5.2 presents the accuracies obtained by LSTMv1 and LSTMv2 on the test datasets. It can be seen from the table that LSTMv2 which is relatively complex in comparison to LSTMv1 performs better on all the datasets.

Table 5.2: Test Accuracies of the LSTM architectures on the datasets

	LSTMv1	LSTMv2
REDD	80.2	89.1
UK-DALE	65.4	70.2
SEADS	78.4	92.3

Table 5.3 presents the average inference time (in seconds) of LSTMv1 and LSTMv2 on the test datasets

Table 5.3: Average inference time (in seconds) of the LSTM architectures on the test datasets

	LSTMv1	LSTMv2
REDD	0.1	0.6
UK-DALE	0.1	0.6
SEADS	0.2	0.8

5.3 Conclusion

On basic comparison of the validation and the test dataset accuracies, an improved performance on the SEADS dataset can be noticed. SEADS dataset being a high frequency dataset seems to be very helpful in identifying detailed channel (or appliance) signatures even when multiple channels (or appliances) are active. The present work also is in line with Dr. Adabi's [7] thesis in that the dataset was used as a complementary tool along with a variation of the NIMLTK package in order to streamline the training process.

The test accuracy of 92.3% was obtained when the first 50 harmonics of SEADS data was used to make predictions. 92.3% accuracy conveys that approximately 92 out of a 100 samples are predicted correctly. Another interpretation that can be drawn from the value is that, for the current version of the neural network 92.3% is the highest accuracy it can achieve without overfitting the training data.

Although the inference time of LSTMv2 on SEADS dataset is on the higher side compared to other inference times, it can be noted that the higher inference time is influenced by both the dimensionality (50 dimensions) of the input and also the complexity of the LSTMv2 network. Advancements in the field have developed neural network compression techniques such that the algorithms can efficiently work on edge devices (mobile devices or low-energy computing devices).

I would like to conclude that the availability of real-time high frequency data such as SEADS and the presence of a sufficiently complex and well trained neural network algorithm with a low memory footprint, can bring us closer to disaggregating power at low latencies without the need for measurement of power at every device.

5.4 Future Work

Future work can concentrate on improving the effectiveness of the neural network by improving it's complexity and at the same time making the best model work on a real-time inference cycle. Techniques such as batch normalization can be used in conjunction with advanced training techniques in order to achieve an optimal model. Deploying the model on the existing SEADS hardware can be a challenge due to storage and memory constraints. Recent trends in model compression techniques can shed light on effective compression of the model thereby making it "light-weight" to deploy it on edge devices. Once deployed, it can be later considered an IoT device which can help in consolidation of data from a myriad of sensors. Utilizing the same data, tasks such as anomaly detection and usage trends of each appliance can be analyzed.

Bibliography

[1] Amazon web services.

[2] Keras.

[3] Nilmtk.

[4] Python.

[5] Tensorflow.

[6] A. Adabi, P. Manovi, and P. Mantey. Seads: A modifiable platform for real time monitoring of residential appliance energy consumption. In *2015 Sixth International Green and Sustainable Computing Conference (IGSC)*, pages 1–4, Dec 2015.

[7] Ali Adabi. Economical real-time energy management for microgrids via nilm and with user decision support. 2016.

[8] Yoshua Bengio and Yann Lecun. *Scaling learning algorithms towards AI.* MIT Press, 2007.

[9] Vitaly Bushaev. Stochastic gradient descent with momentum.

[10] Kyunghyun Cho, Bart van Merrienboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation, 2014.

[11] Daphne Cornelisse. An intuitive guide to convolutional neural networks.

[12] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *J. Mach. Learn. Res.*, 12:2121–2159, July 2011.

[13] Jeffrey L. Elman. Finding structure in time. *Cognitive Science*, 14(2):179 – 211, 1990.

[14] Felix A. Gers, Jürgen Schmidhuber, and Fred A. Cummins. Learning to forget: Continual prediction with lstm. *Neural Computation*, 12:2451–2471, 2000.

[15] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In Yee Whye Teh and Mike Titterington, editors, *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, volume 9 of *Proceedings of Machine Learning Research*, pages 249–256, Chia Laguna Resort, Sardinia, Italy, 13–15 May 2010. PMLR.

[16] Alex Graves, Abdel-rahman Mohamed, and Geoffrey Hinton. Speech recognition with deep recurrent neural networks. *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*, May 2013.

[17] Alex Graves and Jrgen Schmidhuber. Framewise phoneme classification with bidirectional lstm and other neural network architectures. *Neural Networks*, 18(5):602 – 610, 2005. IJCNN 2005.

[18] G. W. Hart. Nonintrusive appliance load monitoring. *Proceedings of the IEEE*, 80(12):1870–1891, Dec 1992.

[19] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. *2015 IEEE International Conference on Computer Vision (ICCV)*, Dec 2015.

[20] Sepp Hochreiter and Jrgen Schmidhuber. Long short-term memory. *Neural computation*, 9:1735–80, 12 1997.

[21] Eric Jones, Travis Oliphant, Pearu Peterson, et al. SciPy: Open source scientific tools for Python, 2001–. [Online; accessed ¡today¿].

[22] Jack Kelly and William Knottenbelt. The uk-dale dataset, domestic appliance-level electricity demand and whole-house demand from five uk homes. *Scientific Data*, 2:150007, Mar 2015.

[23] Jack Kelly and William J. Knottenbelt. Neural nilm: Deep neural networks applied to energy disaggregation. In *BuildSys@SenSys*, 2015.

[24] Christoph Klemenjak and Peter Goldsborough. Non-intrusive load monitoring: A review and outlook, 2016.

[25] J. Zico Kolter. Redd : A public data set for energy disaggregation research. 2011.

[26] J. Zico Kolter and Tommi Jaakkola. Approximate inference in additive factorial hmms with application to energy disaggregation. In Neil D. Lawrence and Mark Girolami, editors, *Proceedings of the Fifteenth International Conference on Artificial Intelligence and Statistics*, volume 22 of *Proceedings of Machine Learning Research*, pages 1472–1482, La Palma, Canary Islands, 21–23 Apr 2012. PMLR.

[27] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. *Commun. ACM*, 60(6):84–90, May 2017.

[28] Quoc V. Le, Navdeep Jaitly, and Geoffrey E. Hinton. A simple way to initialize recurrent networks of rectified linear units, 2015.

[29] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, Nov 1998.

[30] Wes McKinney. Data structures for statistical computing in python. In Stéfan van der Walt and Jarrod Millman, editors, *Proceedings of the 9th Python in Science Conference*, pages 51 – 56, 2010.

[31] Vinod Nair and Geoffrey E. Hinton. Rectified linear units improve restricted boltzmann machines. In *ICML*, 2010.

[32] Oliver Parson, Siddhartha Ghosh, Mark Weal, and Alex Rogers. Non-intrusive load monitoring using prior models of general appliance types. In *Proceedings of theTwenty-Sixth Conference on Artificial Intelligence (AAAI-12)*, pages 356–362, July 2012.

[33] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. You only look once: Unified, real-time object detection. *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, Jun 2016.

[34] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15:1929–1958, 2014.

[35] Huiting Zheng, Jiabin Yuan, and Long Chen. Short-term load forecasting using emd-lstm neural networks with a xgboost algorithm for feature importance evaluation. 2017.