BLE WIRLESS OCCUPANCY REPORT

Urban Pistek 20802700

University of Waterloo, Waterloo, Canada N2L 3G1

1.0 BACKGROUND

1.1 Overview

In short Bluetooth low energy (BLE) wireless technology can be utilized detect human occupancy in closed indoor environments. The core principle is that human presence in a room can affect the RF signal parameters of a wireless devices. The main parameters of interest are received signal strength (RSS), angle of arrival (AoA) and time of flight (ToF). Through analysis of these parameters we can detect and predict human occupancy in a closed environment.

This first section **Background** aims to provide an overview into the problem, BLE technology and the reasoning and theory behind the method explored in solving this problem. **Implementation** will cover how things were build and utilized and **Analysis and Results** will walk through the results obtained. Finally **Discussion** will cover a summary of the results and some recommendations for further analysis.

1.2 Problem Definition and Scope

The main objective of this project was to develop some method of determining occupancy in a room and be able to detect the amount of people present using BLE modules.

1.3 BLE Occupancy Detection Theory

The received signal strength index (RSSI) is the measured strength of the broadcasted RF signal between a peripheral and central node. Since any radio frequency signal is prone to irregularity this value can be used to quantify and detect irregularities. RSSI is a relative index of the absolute dBm which corresponds to power in milliwatts of the received signal. It is important to note that RSSI scales are not standardized, however this does not affect the theory behind using BLE for occupancy detection; since we are more concerned with changes in RSSI values rather than absolute values.

Since this RSSI value can be polled at a regular interval, a plot of RSSI values over time can be produced for an individual BLE device. A peripheral and central BLE device sitting in a room will have some cyclic pattern to the RSSI values polled between the two nodes since there are only constant disturbances to the RSSI value. However, once a human passes through that area their physical presence alters the RSSI value between the peripheral and central node. Hence, it follows that this alteration in the RSSI plot can be used to predict the presence of a person.

As evident in Figure 1 below, the RSSI values of the empty room have a distinct pattern of two spikes followed by a constant value repeated over time. However, the signal with one person present in the room causes a much more chaotic and disturbed signal. This indicates just how much a person's presence in the room can affect the RSSI values over time.

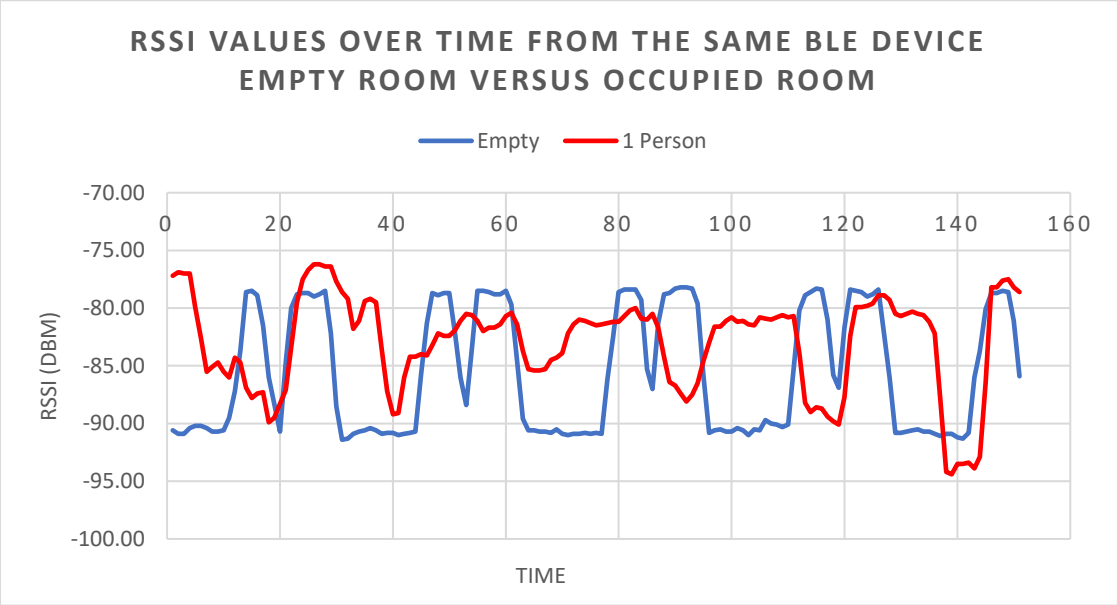

Figure 1 - RSSI Values empty room versus Occupied room over time from same BLE device

Other parameters such as the mentioned ToF and AoA can be utilized in this fashion for detecting human occupancy due the same reasoning; human presence creates a significant alteration in the signal that can be used to detect presence.

Implementation of algorithms and methods of signal processing to determine occupancy vary and can be application specific. For instance, one method utilized was creating a script that would detect any significant spikes in the RSSI when a person was present [1]. Alternatively, another method used a variation of a recurrent neural network with a combination of RSSI and ToF as parameters for the input vector [2]. The methodology explored for this problem will be explained in 1.7 Explored Methodology for Occupancy Detection.

1.4 Equipment and Measurement

The BLE devices used were the RuvviTag BLE modules; a small compact BLE sensor with a Nordic Semiconductor nrf52832 System-on-Chip. The Nordic semiconductor development board with the RuvviTag shield was used to flash the board with the appropriate firmware. For this application the nRF5 SDK was used for the firmware, in particular the *ble_peripheral* example *ble_app_beacon* for the *pca10056 , s140 ses* version was flashed to the RuvviTags via the development board. This firmware example had the RuvviTags function as a beacon that was constantly broadcasting the RSSI value to nearby monitoring devices. A smartphone with the nRF Connect app was used detect these broadcast signals and recorded 10-minute batches of RSSI values.

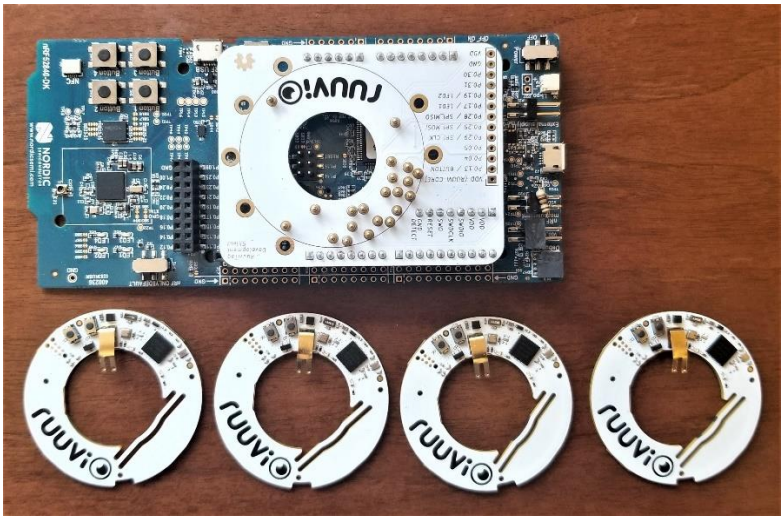

Figure 2 - RuvviTags and Development Board

To measure ToF or AoA parameters significant changes to the firmware would need to be made since these parameters are not native like the RSSI parameter. Further, the nRF Connnect application did not have the available functionality to measure other parameters – there where methods to broadcast other custom data, however the app did not have a feature to log this data as a timeseries built into it. Although much of the software is open source and a application could potentially be built, I choose not to invest my time into this portion of the project.

As a result, another solution for collecting custom data form multiple nodes was to configure a central node to which all other peripheral nodes would broadcast data. This central node could be connected to the development shield and configured to log all the incoming data to a laptop using UART – the SDK contained examples of utilizing UART for logging data.

However, this approach required a lot of overhead and development with the SDK to implement custom firmware for the devices. I was unable to get the UART configuration working with a laptop; using UART to log data from the central node. I managed to get a example working with the central node – peripheral node configuration however it was very limited and I never managed to implement a method of exchanging custom parameters. Therefore, after some investigation I decided to focus on using the nRF connect app for collecting RSSI values and only using RSSI values.

Additionally, the custom parameters (ToF and AoA) both had their own complications in regard to implementation. Implementation of ToF depends on whether the BLE devices have synchronized clocks or not; in the case of un-synchronized clocks a round trip time measurement (RTT) technique described in [2] would need to be implemented. This would involve the central node sending a message to the peripheral node while at the same time starting a timer. After receiving the message, the peripheral node sends a reply message after a fixed delay. Upon receiving the reply message, the central node samples its timer and calculates the propagation time of the signal.

Implementation of AoA requires significantly more overhead than ToF. Firstly, the receiving node requires an array antenna in order to implement a phase-based direction-finding system for AoA; in addition to RF switchs, multi-channel ADC and computing power to run the algorithm.

The RuvviTag modules only came standard with RSSI values built in, to implement ToF and AoA requires both hardware and software implements and for the scope of this project I decided not to pursue this implementation unless I first succeeded with utilizing RSSI values only.

1.5 Experimental Design [UP1]

In general, there were two states that data needed to be collected for; an empty space and a non-empty space. However, since part of the scope was to determine how many people are present, states with varying amounts of people was also required. To keep things simple the initial sets of data had 3 states that RSSI values where record for: Empty, One Person and Two People. For measurements with people present half the data recorded was the RSSI values as person(s) stood in various positions changing every minute. Whereas the other half involved recording RSSI values as the person(s) moved around in the space. This was to in order to cover the two states people could be in while in the space – moving or not moving [2].

This setup was used to collect 3 sets of data from 3 different rooms with the final dataset containing some alternations to try and capture more edge cases/realistic behaviours. Each iteration is explained below and links to a specific dataset. Data was collected with the person(s) moving and standing in positions to try an reflect different human behaviour; this data was later mixed together to have a one person dataset with a mix of movement and standing in various positions since the aim to predict human presence – not movement. This experimental approach was based on the approach outlined in [2].

1.5.1 Experiment 1 & Dataset 1

4 Ruvvi tags where placed around the edge of a relatively symmetrical room at various heights. These RuvviTags were flashed with the ble_app_beacon firmware to broadcast RSSI values. A phone was placed relatively central to all the modules. The layout of the room and the placement of the RuvviTags can be observed in Figure 3 (Where B1 = Beacon 1 etc.).

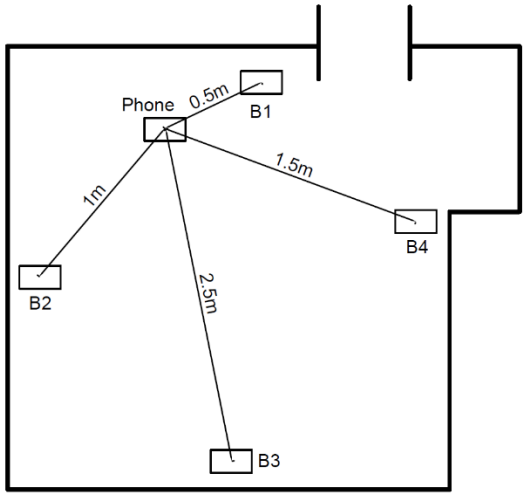

Figure 3 - Room 1 Layout of BLE Modules and Phone for Experiment and Dataset 1

The nRF Connect was used to log the RSSI values over time at a sample rate of 1s[UP2] for various scenarios and time intervals:

- Empty room for 10 minutes
- 1 person in the room standing in various positions – changing position every 1 minute for 5 minutes total
- 1 person in the room constantly moving around for 5 minutes
- Empty room for 10 minutes
- 2 people in the room standing in various positions – changing position every 1 minute for 5 minutes total
- 2 people constantly moving around for 5 minutes total

1.5.2 Experiment 2 & Dataset 2

This iteration follows similar guidelines as Experiment 1 & Dataset 1 with the key difference being the room, layout and amount of data collected. The layout of the room and the placement of the RuvviTags can be observed in Figure 4 (Where B1 = Beacon 1 etc.).

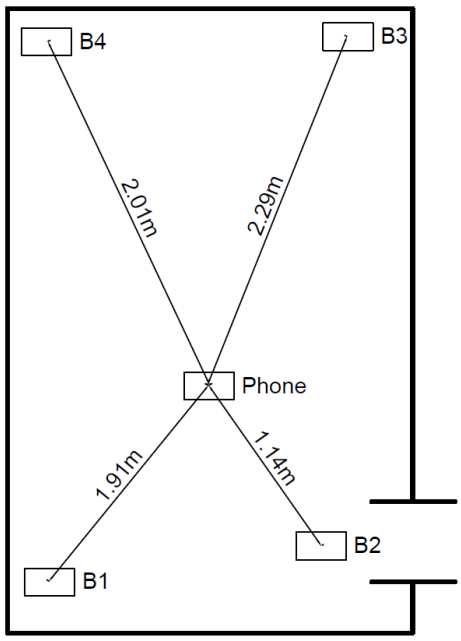

Figure 4 - Room 2 Layout of BLE Modules and Phone for Experiment and Dataset 2

The nRF Connect was used to log the RSSI values over time at a sample rate of 1s[UP3] for various scenarios and time intervals:

- Empty room for 30 minutes
- 1 person in the room standing in various positions – changing position every 1 minute for 15minutes
- 1 person in the room constantly moving around for 15 minutes
- 2 people in the room standing in various positions – changing position every 1 minute for 15 minutes
- 2 people constantly moving around for 15 minutes

1.5.3 Experiment 3 & Dataset 3 [UP4]

Again, this third iteration followed similar guidelines and featured a different room and layout as the previous 2. However, the key difference in this iteration is that the person(s) entered and left the room at random intervals for varying periods of time. The moment the person(s) entered/left the room was detected by an ultrasonic sensor at the doorway[UP5]. The layout of the room and the placement of the RuvviTags can be observed in Figure 5 (Where B1 = Beacon 1 etc.).

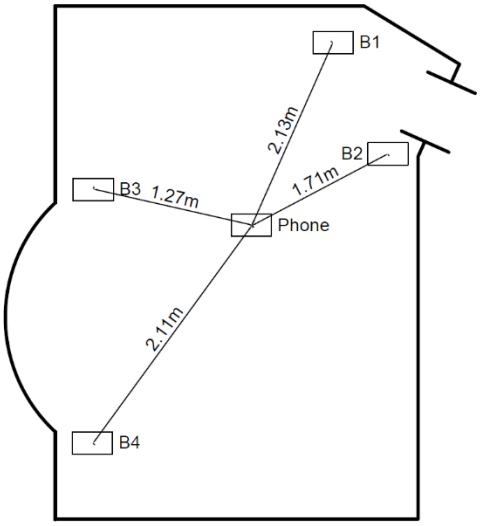

Figure 5 - Room 3 Layout of BLE Modules and Phone for Experiment and Dataset 3

The nRF Connect was used to log the RSSI values over time at a sample rate of 1s[UP6] for various scenarios and time intervals whereas the ultrasonic sensor logged when people entered/left the room:

- Empty room for 10 minutes
- 1 person entering and leaving the room at varying intervals and rates for 10 minutes
- 2 people entering and leaving the room at varying intervals and rates for 10 minutes

Please Note: Upon investigation of the data of Dataset 3 I was not confident in its quality and chose to not use it.

1.6 Data Collection

Data logged by the mobile phone was exported to a excel file for each round of data collection. Data was then initially compiled manually into a master CSV file and combined into three different datasets corresponding to the outlines mentioned above: dataset1, dataset2, dataset3. For each dataset there where 5 columns, one for each RuvviTag Beacon and the final column contained the labels: Empty Room, One Person and Two People. Each compiled dataset contained all the data collected for each state as outlined in Table 1.

Table 1 - Overview of dataset structure

Label	*B1*	*B2*	*B3*	*B4*
Empty	-90.6	-72.1	-79	-83.2
⋮	⋮	⋮	⋮	⋮
One Person	-90.9	-72.2	-79	-83
⋮	⋮	⋮	⋮	⋮

Two People	-84.7	-71.7	-74.8	-85.8

This data was processed further using python (a detailed explanation of the data processing will be covered in 2.2 Data Processing); each dataset can be found in the correspondingly named excel file. A summary of each dataset is as follows, including how any instances of Not a Number (NaN) which is a result of missing data was handled:

Table 2 - Overview of the datasets collected

Dataset	*Length (rows)*	*Size (4 x rows)*	*Description*	*How NaN was Handled*
dataset1	3666	14664	10 min of Empty, 1 Person, & 2 People	NaN was replaced by an interpolated value
dataset2	10201	40804	30 min of Empty, 1 Person, & 2 People	NaN was replaced by the mean (of the column)
dataset3	1199	4796	10 min of Empty, 1 Person, & 2 People leaving/entering	NaN was replaced by the mean (of the column)

1.7 Explored Methodology for Occupancy Detection

The nature of the data being received is multiple timeseries of some parameter over time for each BLE module. For a given BLE Beacon we can log either the RSSI, ToF or AoA over time for different states such as empty room, one person, two people etc. Then we can add multiple Beacons to capture multiple instances of each of these parameters for each time step. As mentioned previously, for the scope of this project ToF and AoA were not measured; if added all the same methodology would follow as it would just be another data point for each time step.

By comparing the differences in the waveform of the RSSI values between the room in the empty, one person and two people state we can distinguish how many and if people are present in a room since the RSSI waveforms changes significantly between each different state. As seen in Figure 6 below, there are significant differences between the RSSI values for each state for just this single BLE Beacon. The challenge becomes identifying the key features and patterns that distinguishes each state.

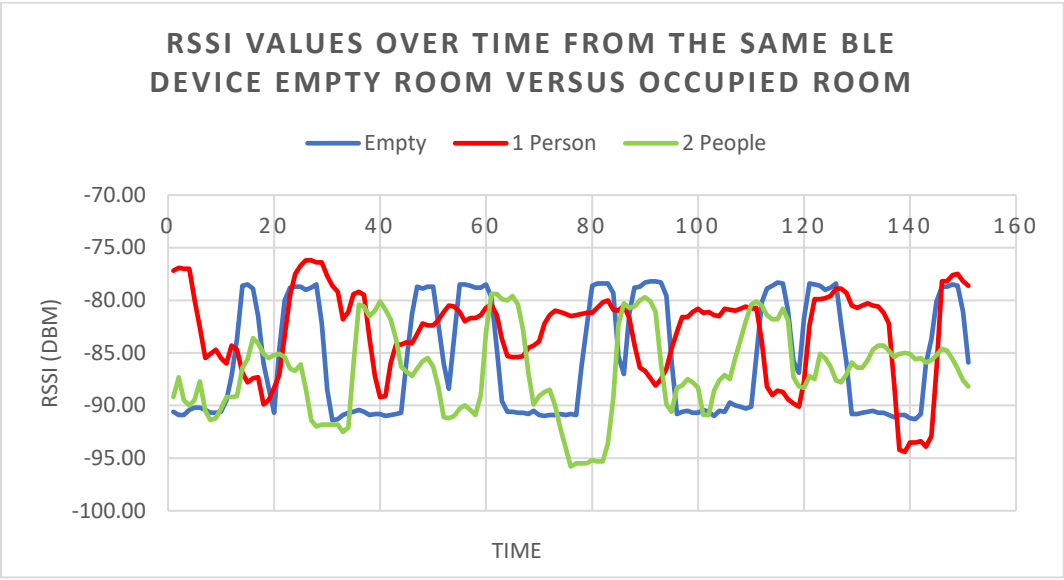

Figure 6 - Comparison of Empty, 1 Person and 2 People RSSI values over time

Various methodologies have been used to explore developing a system that predicts the occupancy of a room based on some combination of these input parameters[UP7]. Upon evaluating the various approaches, I decided to explore using deep learning and in particular, using neural nets as a prediction method [2] [3].

The primary reason was that neural networks are ideal for processing large amounts of data and are highly dimensional. Compiling data for this problem was relatively simple and could be automated heavily. As a result, large datasets can be produced relatively easily. This leads to the primary reason why neural networks are effective for this problem – they handle large numbers of dimensions extremely well. Further, in practice once configured in code, the dimensionality of the input data and its structure can be easily modified to experiment with different parameters and dataset sizes. Therefore, adding more parameters such as ToF or AoA to each beacon for the dataset does not require significant alterations to modify the neural net (the neural net will have to be re-trained and some parameters may need to change).

Take the simplest dataset for this problem; a 1D vector of RSSI values over time (where time is taken to be the index of the row and the RSSI value is the value at that instant in time) of size n x 1. Adding RSSI values over time from another BLE beacon transforms the vector into a matrix of size n x 2. Additional parameters can be added to make the data matrix any size of n x m. Figure 7 below outlines an example of how multiple parameters of time series data from different states can be processed by a neural net to output a prediction result.

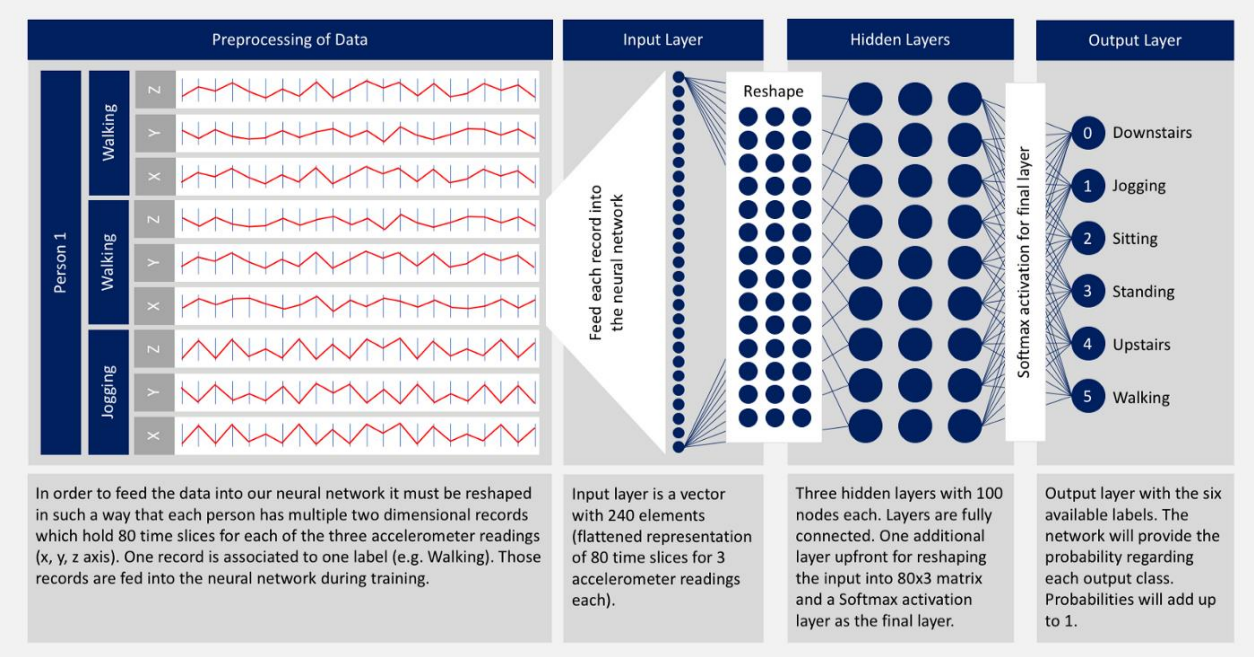

Figure 7 - "Deep Neural Network Example" by Nils Ackermann is licensed under Creative Commons CC BY-ND 4.0 ***Source:*** *https://towardsdatascience.com/human-activity-recognition-har-tutorial-with-keras-and-core-ml-part-1-8c05e365dfa0*

Further, neural networks are effective at modelling relationships between inputs and outputs that are non-linear and complex. They can generalize, reveal hidden relations and model highly volatile data [4]. These characteristics provide a strong case for why neural networks should be explored for this particular problem.

An issue persists; which type of neural network should be implemented for this problem. Upon exploration of different neural network architectures, I decided that a convolutional neural network would be best suited for this application of deep learning.

1.8 Convolutional Neural Networks for 1D Timeseries Data

Based on recent breakthroughs, 1D CNNs have been found to be more effective at 1D timeseries data than recurrent neural networks (RNN) (which were previously thought to be most effective for 1D timeseries data) [3] [5] [6]. Further, I was able to find multiple studies online using CNNs for timeseries data applications. As a result I decided to explore 1D CNNs for this application.

In principle convolutional neural networks use convolution (a mathematical operation between two functions that produces a third function describing how the shape of one is modified by the other) to extract features of a dataset. In the context of deep learning the convolution is a linear operation involving the multiplication of a set of weights with an input array of data. Therefore, you have the multiplication of two 2D arrays, one which is the input data and the other which is a 2D array of weights known as a kernel/filter. The kernel/filter is smaller than the input array and the dot product of these two 2D arrays (in essence matrices) results in a singular value. This operation is applied systematically across the entire input array with some overlap. This outputs a 2D array known as a feature map. What makes this operation powerful is that the

convolution is able to extract the **most critical features** of an input. Pooling is then applied to condense the input into smaller dimensionality while preserving the most important features in the data.

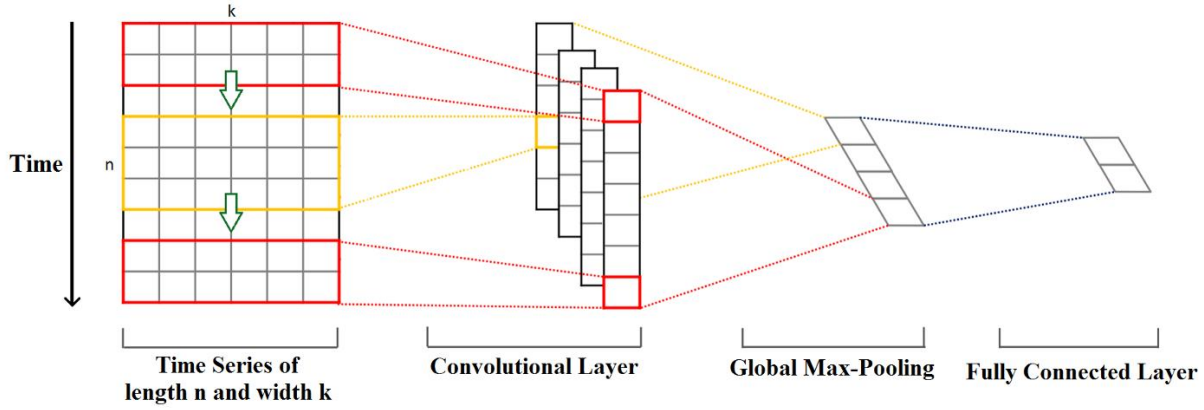

Figure 8 - 1D Convolution, ***Source****:https://towardsdatascience.com/how-to-use-convolutional-neural-networks-for-time-series-classification-56b1b0a07a57*

The convolution can then be stacked into multiple layers of the network. This results in the extraction of higher- and higher-level features as convolution layers are stacked. For example, for a 2D input such as an image (the most common application of CNNs), the first convolution layer will extract lines as features, followed by composite lines, followed by shapes and ultimately resulting in complex objects. It is the effectiveness of extracting features from data that makes CNNs effective for timeseries data [3] [5] [6].

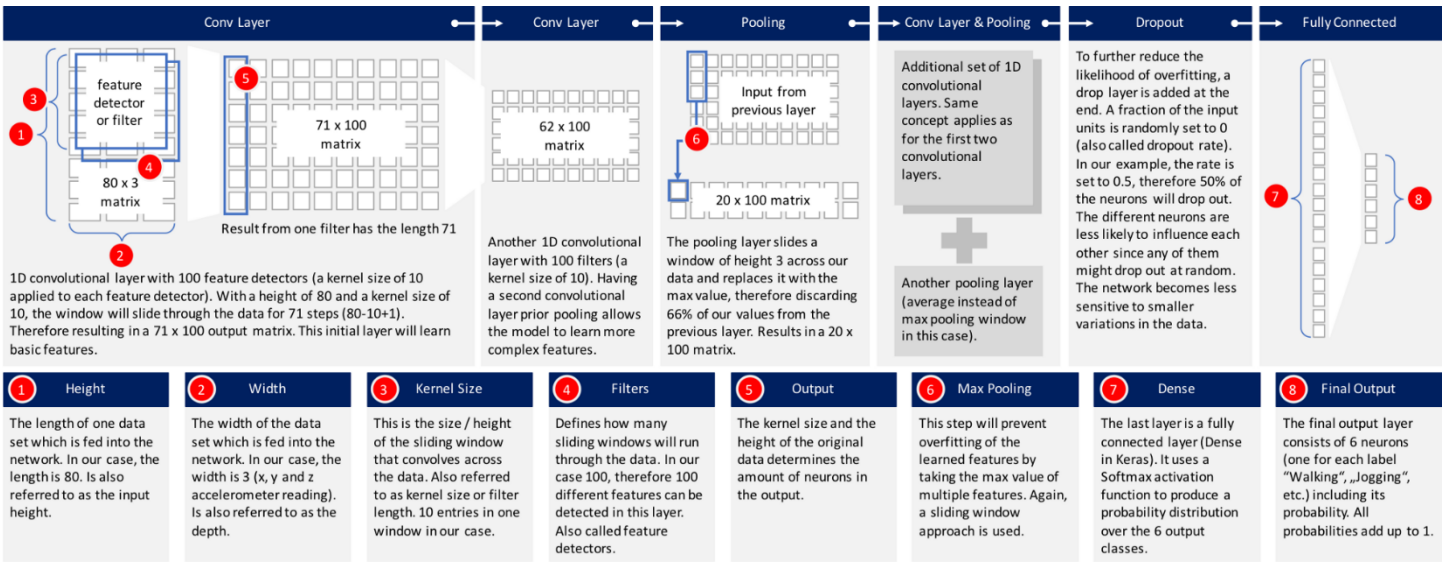

Figure 9 - "1D CNN Example" by Nils Ackermann is licensed under Creative Commons CC BY-ND 4.0 ***Source:*** *https://blog.goodaudience.com/introduction-to-1d-convolutional-neural-networks-in-keras-for-time-sequences-3a7ff801a2cf*

When comparing data such as RSSI values between the empty and non-empty state the difference in the shape of the graph or to put it differently; the fluctuations between the empty timeseries data which has a

constant an rhythmic pattern and the non-empty state data – which has disturbances and fluctuations can be effectively extracted using CNNs. Further, from Figure 6 it can be seen how these disturbances vary between the amount of people present in a room. Therefore, based on these reasons I decided to explore using CNNs for BLE occupancy detection.

In essence, a 1D convolution neural network functions similarly to a 2D CNN. The key difference is how the data is organized and inputted into the 1D CNN. A current application of 1D CNNs involves human activity recognition based off timeseries accelerometer data [6]. Here accelerometer data is collected over time in the x, y, z axis which is then transformed into a 3 x n matrix of data with each row corresponding to one axis of data. A convolution kernel/filter of size 3 x m where m < n is iterated through the dataset matrix performing the convolution operation.

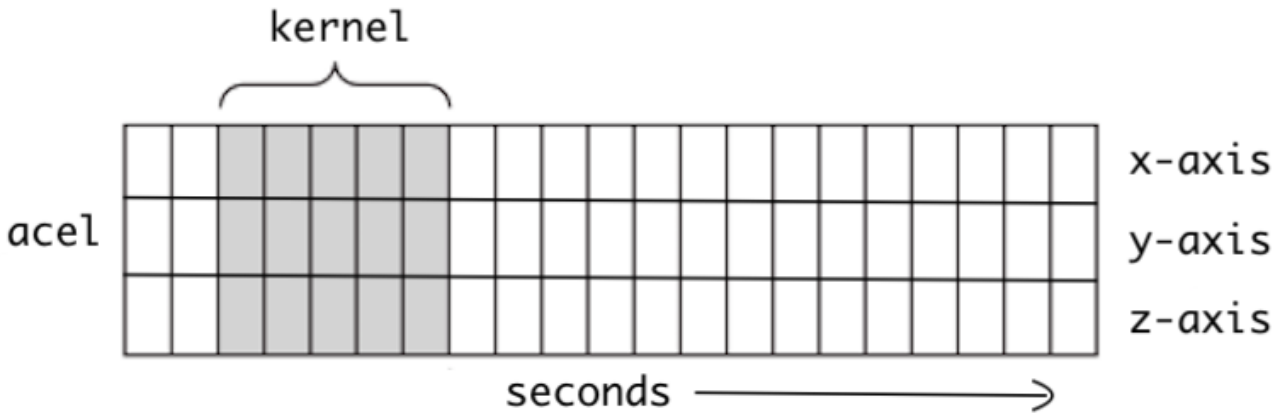

Figure 10 - 1D Convolution of x, y, z accelerometer data ***Source:*** *https://towardsdatascience.com/understanding-1d-and-3d-convolution-neural-network-keras-9d8f76e29610*

This implementation was mirrored for my application with the RSSI values over time from each of the 4 BLE beacons acting as the axis. Thus, the result was a 4 x n data matrix with n = TIME_STEPS (a global constant used in the python implantation), where TIME_STEPS is the size of the time interval or consecutive measurement per input matrix. Further, as mentioned previously, adding more data such as a 5th BLE beacon or adding ToF measurements for each BLE beacon simply corresponds to another row in the input data matrix.

2.0 IMPLEMENTATION

2.1 Developing the 1D CNN using Python [UP8]

All code has been provided with a zipped folder for reference, this section aims to give a quick overview of how the model was built in python.

There are a variety of deep learning libraries available, the most popular include Pytorch, TensorFlow and Keras. Pytorch is easy to implement and relatively user friendly whereas TensorFlow is more advanced and

also more powerful. Keras is also more user friendly and easier to pick up; Keras also uses TensorFlow as a backend.

Ultimately which library is used is mostly a matter of preference, I decided to build the CNN using Keras since it had a built-in 1D CNN object that could be used and I found a couple examples using Keras for 1D CNNs. In addition to the deep learning libraries I would recommend installing Anaconda which is a data science platform which includes many useful python packages such as pandas, matplotlib and numpy (the full list of libraries used can be found in the code provided). Additionally, with anaconda you can easily create development environments and switch between them. Finally, I used PyCharm as the IDE for development, I would highly recommend it for any python development.

2.2 Data Processing [UP9]

Before the neural network could be built in the python, all of the data needed to be processed further. This was largely done using the pandas module.

Data collected was manually compiled into a singular CSV file consisting of columns for each beacon and the state of the room (Empty Room, 1 Person and 2 Person) for the few missing values in each column the value was simply interpolated or replace by the mean as outlined in Table 2. As mentioned, the data collected for person(s) moving and standing in different positions was combined together. An equal amount of data points were allocated from each state to ensure a even split among the data. The format of the data is outlined in Table 1.

The formatted CSV file was then imported into python using the pandas module and converted into a dataframe object. The data was normalized for each iteration in each column with the minimum value in that column, thus producing data values that ranged between 0-1. For the state column, each state was modified to correspond to a numerical value as follows: "Empty": 0, "One person": 1, "Two people": 2.

```
38  def load_dataset():
39      data = pd.read_csv('dataset1.csv')
40      # Normalize the Data - Note Data is all negative so dividing by istelf twice will make it positive
41      data['B1'] = data['B1'] / data['B1'].min()
42      data['B2'] = data['B2'] / data['B2'].min()
43      data['B3'] = data['B3'] / data['B3'].min()
44      data['B4'] = data['B4'] / data['B4'].min()
45
46      data['Label'].replace({"Empty": 0, "One person": 1, "Two people": 2}, inplace=True) #Replace Strings with values
47
48      # Round numbers
49      data = data.round({'B1': 6, 'B2': 6, 'B3': 6, 'B4': 6})
50
51      return data
52
```

Figure 11 - Loading data into dataframe and normalizing

Subsequently the whole dataset needs to be converted into a tensor of tensors in order for it to be split into splices of length TIME_STEPS. Additionally, the states need to be converted into a state vector with one hot value. As result the data is configured as follows:

$$data = \left[\left(\begin{bmatrix} B_{1_1} & \cdots & B_{4_1} \\ \vdots & \ddots & \vdots \\ B_{1_n} & \cdots & B_{4_n} \end{bmatrix}, \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}\right), \left(\begin{bmatrix} B_{1_1} & \cdots & B_{4_1} \\ \vdots & \ddots & \vdots \\ B_{1_n} & \cdots & B_{4_n} \end{bmatrix}, \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}\right), \ldots \left(\begin{bmatrix} B_{1_1} & \cdots & B_{4_1} \\ \vdots & \ddots & \vdots \\ B_{1_n} & \cdots & B_{4_n} \end{bmatrix}, \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}\right)\right]$$

Where n = TIME_STEPS, B_{1_n}, B_{2_n}, B_{3_n}, B_{4_n} correspond to the beacon RSSI value at time step n and the vectors $\begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}$ = Empty, $\begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}$ = One person, $\begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$ = Two people.

When parsing the dataset there were two global constants that primarily affected how the data was parsed, TIME_STEPS and STEP_DISTANCE. TIME_STEPS refereed to how many iterations or rows were in a single tensor (each tensor is of size 4 columns x TIME_STEPS rows). Whereas, STEP_DISTANCE refers to how many data values from the starting value is the next starting value. Therefore, if TIME_STEPS == STEP_DISTANCE then there is no overlap in data. It follows that if TIME_STEPS = 50 and STEP_DISTANCE = 25 there is a 50% overlap in data between each tensor interation. Finally, the data is split into the input data (referred to as X) and result (referred to as y).

```
98  def make_training_data(df, time_steps, step):
99      segments = []
100     labels = []
101     for i in tqdm(range(0, len(df) - time_steps, step)):
102         B1 = df['B1'].values[i: i + time_steps]
103         B2 = df['B2'].values[i: i + time_steps]
104         B3 = df['B3'].values[i: i + time_steps]
105         B4 = df['B4'].values[i: i + time_steps]
106         # Retrieve the most often used label in this segment
107         label = stats.mode(df['Label'][i: i + time_steps])[0][0]
108         segments.append([B1, B2, B3, B4])
109         labels.append(label)
110
111     # Bring the segments into a better shape
112     reshaped_segments = np.asarray(segments, dtype=np.float64).reshape(-1, time_steps, N_FEATURES) #Modify
113     value = np.asarray(labels) #Modify
114
115     return reshaped_segments, value
```

Figure 12 - Creating tensor of tensor, splitting data into input(X) and output(y), defining the output one hot vector

The function was built to organize and parse the data passes TIME_STEPS and STEP_DISTANCE as arguments. This made it very simple to change the size of the 2D arrays in each tensor and the overlap between data; the neural network could be easily evaluated at different data configurations. Once arranged as a tensor of tensors the data is then randomly shuffled preserving each tensor – one-hot-vector pair since the input and result need to stay paired together.

```
157
158 def shuffle_data(X, y):
159     print("Object_Type_X: ", type(X))
160     print("Object_Type_y: ", type(y))
161     c = list(zip(X, y))
162     random.shuffle(c)
163     X, y = zip(*c)
164     X = np.asarray(X)
165     y = np.asarray(y)
166     print("After Shuffle: ")
167     print("Object_Type_X: ", type(X))
168     print("Object_Type_y: ", type(y))
169     return X, y
170
```

Figure 13 - Randomly shuffling the data keeping each X-y pair together

Once randomly shuffled the tensor list must be separated into a "input data list" and a "expected result list" known by convention as train_x and train_y. Once separated into two separate lists the data for each tensor in train_x needs to be must be flattened into a vector since the 1D convolutional neural net can only accept a vector as the input argument.

After all these steps have been complete the data is ready to be passed into the CNN.

2.3 The Neural Network

For any neural network development there are 3 key aspects: data processing, the neural net and training the neural net. Here we will cover the neural net used for this application.

Firstly, as part of the Keras library there is a Sequential() class that groups a linear stack of layers into a tensorflow.keras model object to be used. This standard 1D CNN model was utilized to simplify building the model. The most important consideration is that keras and Core ML are unable to process mutli-dimensional data input, in this case a size 4 x TIME_STEPS. Hence why the data was flattened into a input vector before passed into the neural net; once passed as a vector it is reshaped into its original size and dimensions. This input vector was always the length of 4 x TIME_STEPS.

Once data has been input the following components of the neural net are the "filter layers"; these filters aid in the convolution process by allowing the network to learn one feature for each layer and then pass that feature along, this is the process of convolution as explained previously. The base model I used had two filter layers both with 64 filters per layer and a kernel size of 3 (this is based on the model used in this [6]). Each layer runs a softmax activation function-in particular using a rectified linear unit instead of a sigmoid.

There are benefits and drawbacks to utilizing either but in general the rectified linear unit is the simplest non-linear activation function that can be used; research as shown that it is much faster for training large networks and many of the examples I found utilized this activation function [7]. To summarize, this activation function transforms the output of a layer into a probability distribution which is used when determining the loss

during each training cycle. Utilizing backpropagation this loss is used to tune the parameters as the model learns.

Further, the base model then employs a dropout layer which simply randomly assigns 0 weights to neurons in the network. The base rate chosen was 50% (which appeared to be standard). This operation is used to make the network less sensitive to react to small variations in the data and should increase accuracy. A max pooling layer is subsequently applied to reduce the complexity of the output and helps highlight the key features of the data.

This outputs a matrix which is then flattened and passed through a fully connected layer which will reduce the final output to the number of classes we want to predict; in this case being 3 prediction classes. A softmax activation is applied to this output to return a probability distribution of the classes.

The structure of the neural net can be summarized below, many of the parameters such as kernel size and drop fraction can be optimized to yield the best possible neural net, however these optimizations were out of the scope of this project.

Table 3 - CNN Overview

Layer (type)	*Output Shape*
reshape	(TIME_STEPS, 4)
filter	(64, kernel=3)
filter	(64, kernel=3)
dropout	(0.5)
max pooling	(2)
flatten1	()
dense 1	(100, 1)
dense 2	(3, 1)[UP10]

```
184
185     def create_model():
186         model_c = Sequential()
187         model_c.add(Reshape((TIME_STEPS, NUM_PARAMETERS), input_shape=(INPUT_SHAPE,)))
188         model_c.add(Conv1D(64, 3, activation='relu', input_shape=(TIME_STEPS, NUM_PARAMETERS)))
189         model_c.add(Conv1D(64, 3, activation='relu'))
190         model_c.add(Dropout(0.5))
191         model_c.add(MaxPooling1D(2))
192         model_c.add(Flatten())
193         model_c.add(Dense(100, activation='relu'))
194         model_c.add(Dense(NUM_RESULTS, activation='softmax'))
195         print(model_c.summary())
196
197         return model_c
```

Figure 14 - CNN Overview in python

2.4 Training the Neural Network

Overall, training the model primarily relies on a couple hyperparameters known as batch size and epochs along with a method of calculating loss, some optimizer and the validation split. A categorical crossentrophy loss function is utilized since this classification problem only has a single correct output. An optimizer is used to update the weight parameters to minimize the loss function (since the goal of most machine learning problems is to minimize the error between the predicted and actual output – also referred to as a loss/cost function). The validation split is used to divide the dataset into training and validation data (a validation split of 0.2-0.4 was used since the dataset being used was a small dataset in the context of deep learning; therefore 20-40% of the dataset was used to validate and the other 60%-80% was used for training the model).

In terms of the hyperparameters, a epoch is when the entire dataset is passed forward and backward through a neural network once. Whereas the batch size is the total number of training examples present in a single batch. These parameters are generally tweaked based on one's discretion. Total number of epochs a model is trained for however is generally based on the behavior of the training accuracy, loss and validation accuracy, loss over a number of epochs. In general, a model is trained until the loss and accuracy between the training and validation data begins to diverge. Training the model after this point will result in overfitting – when the model becomes proficient at predicting for this particular dataset but does not generalize well (when a model is overfitting it is essentially memorizing the data).

We can see this divergence occurring at about the 150th epoch in Figure 15 below.

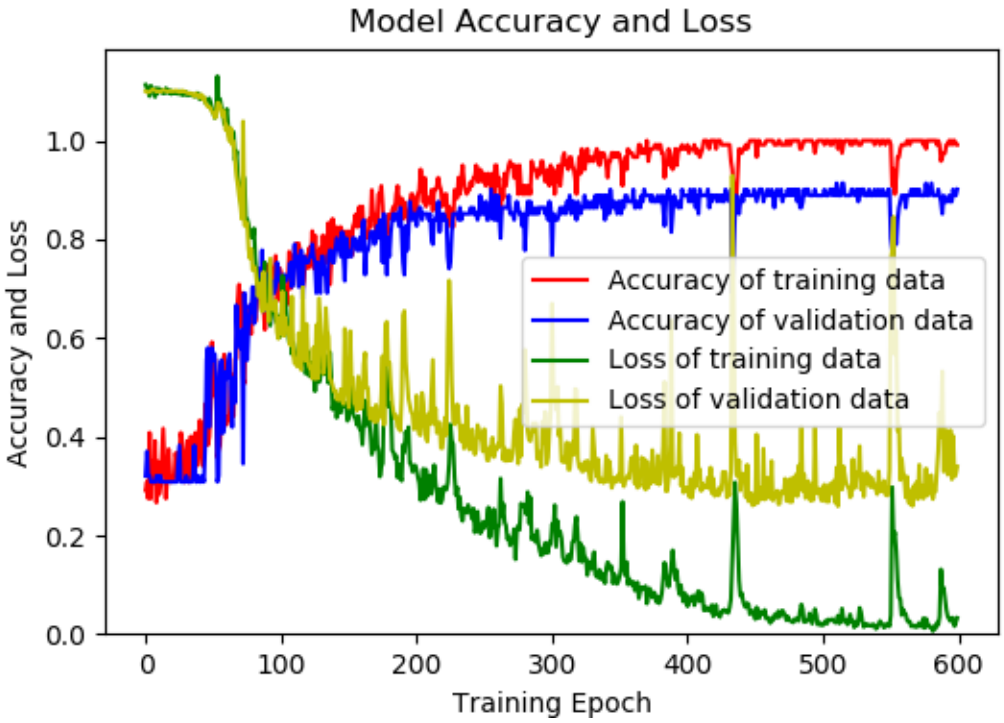

Figure 15 - Model Training, Validation loss and accuracy over epochs

As a result, when training the model to be well generalized this divergence must be carefully monitored to prevent overfitting. Further, various parameters described previously are tuned in order to yield a divergence with the highest training and validation accuracy.

The best parameters to tune and experiment with are as follows, the bolded parameters were experimented with [8]:

- **Learning rate**
- **Smaller batch sizes and larger epochs**
- The Optimizer
- **Using more data**

- **Network Topology**
- **Hyperparameters (batch size, epochs)**

Once such a model has been obtained that yields the desired accuracy along with any other characteristics it must be tested with a dataset it has never seen before to evaluate its true accuracy when generalized. The extent of training and results will be covered in the following section.

3.0 ANALYSIS AND RESULTS

Upon developing the base model, the next major step was training and tuning it. The goal is to train a model so that its training and validation accuracy would continue increasing at the same rate with the loss decreasing. In essence you did not want them to diverge until a desired accuracy (typically train until you reach a high accuracy). To achieve this, I ran multiple training sessions tuning the parameters, models and the data which I was using. I outlined the progression and results below.

3.1 Training and tuning the Model

3.1.1 Training using dataset1

Training first began using dataset1 for the training and validation data. One of the first key parameters experimented with was the learning rate, for each iteration the epochs were the same at 600 and the batch size was kept at around 25. The size of the input data was 4 x 300 and a step distance of about 25 was used – something I found early on was at data with a large overlap seemed to preform much better than data with very little overlap; I was unable to form a definitive explanation for this.

From Figure 16 below we see the data gradually improves until about 250 epochs were it begins to diverge at about a 75% accuracy. Therefore, if this particular model is to be used, we would only train it until 250 epochs and even then it only yields a 75% accuracy.

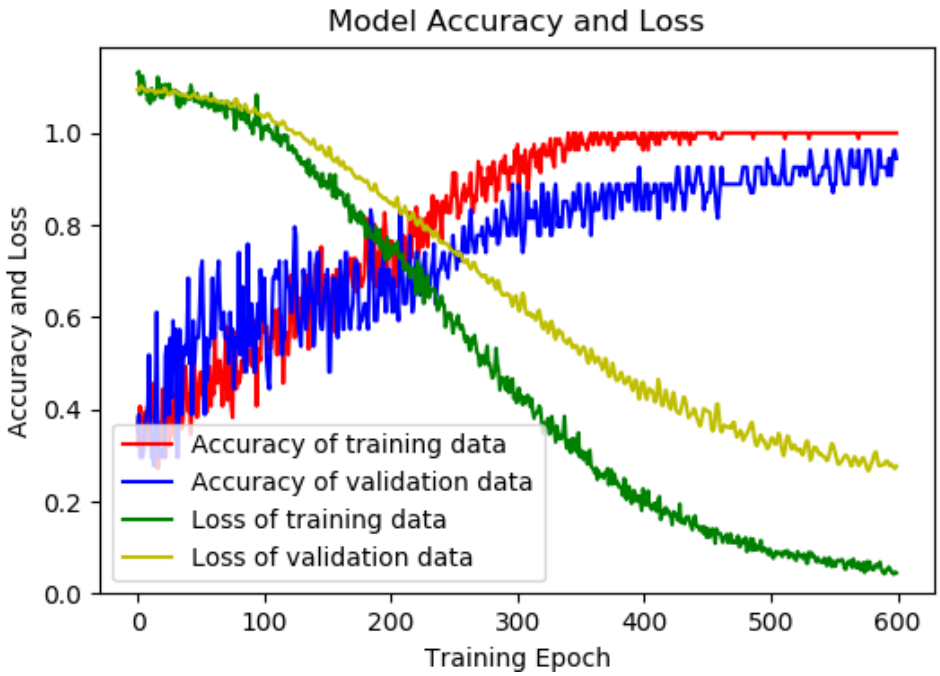

Figure 16 - Learning rate = 0.0001 , Batch Size = 25 , Timesteps = 300, Step Distance = 25 , dataset1

The learning rate was decreased to 0.00001 to see how this change would affect performance. From Figure 17 below we can see that performance drastically decreases where the data is barely improving and begins to diverge already at 200 epochs. This example illustrates how much altering some parameters can drastically change the outcome.

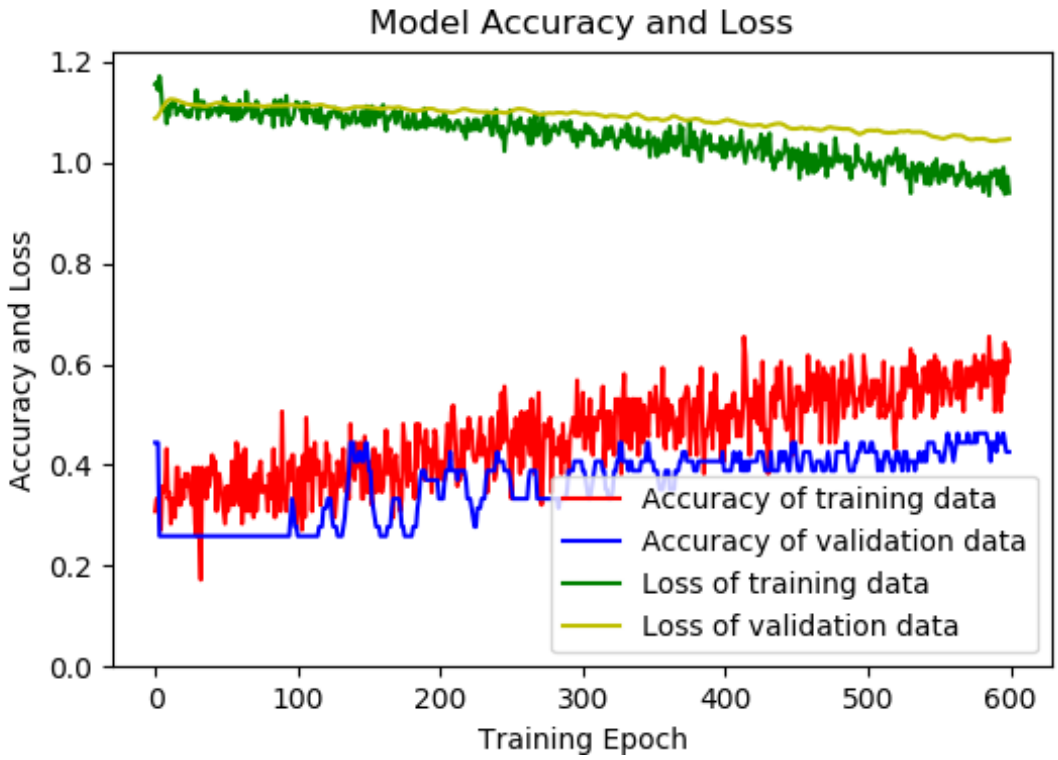

Figure 17 - Learning rate = 0.00001 , Batch Size = 25 , Timesteps = 300, Step Distance = 25 , dataset17

Rather than decreasing the learning rate it was increased to 0.001 with the same parameters and data. From Figure 18 below we can see that the models accuracy improved at a much faster rate before reaching a upper limit of around 97%. If this model where to be used the training would be stopped at around 150 epochs; however the data appears to diverge a significant amount from the beginning of training. Therefore, it may be overfitting even before 150 epochs.

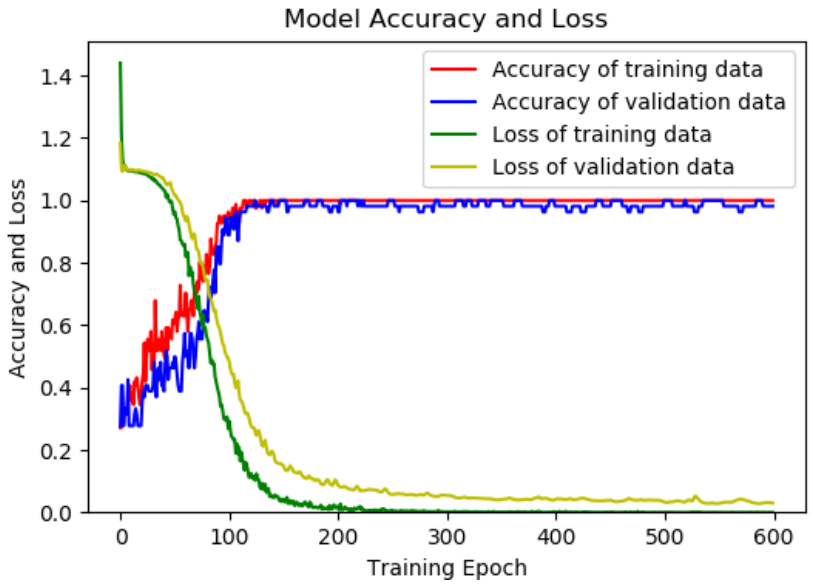

Figure 18 - Learning rate = 0.001 , Batch Size = 25 , Timesteps = 300, Step Distance = 25 , dataset1

From Figure 18 above we encounter another nuance to training a CNN or any neural net in general. From the graph it appears we reach a accuracy of about 97%. However, it must be noted that this is in-sample accuracy, it is hard to conclude from this how well the model generalized to other data that it was not train on. Therefore, any trained model must be tested on out of sample data to evaluate how well it has generalized.

3.1.2 Training using dataset2

For the remainder of training dataset2 was exclusively utilized; it boasted more data and a portion of it was separated to be used for out of sample testing (dataset2_test.csv) which contained 200 data points for each state from each beacon. However, from Figure 19 below we can see that the accuracy only improves until 80% at around 250 epochs. A few models were trained with different parameters however they did not yield better results than this. As a result a different model architecture was explored.

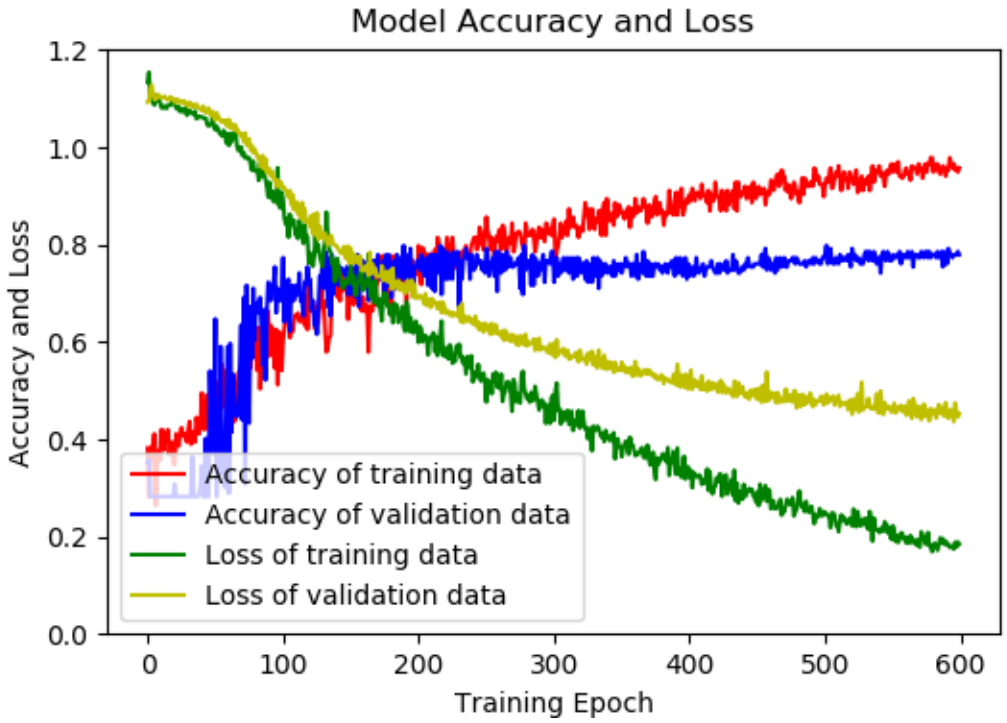

Figure 19 - Learning rate = 0.0001, Batch Size = 50, Timesteps = 300, Step Distance = 25, dataset2

The following table outlines a new model architecture experimented with:

Table 4 - Revised Model Architecture

Layer (type)	*Output Shape*
reshape	(TIME_STEPS, 4)
filter	(100, kernel=10)
filter	(100, kernel=10)
pooling	(3)
filter	(160, kernel=10)
filter	(160, kernel=10)
dropout	(0.5)
dense	(3)

```
def create_model():
    #modified from original
    model_c = Sequential()
    model_c.add(Reshape((TIME_STEPS, NUM_PARAMETERS), input_shape=(INPUT_SHAPE,)))
    model_c.add(Conv1D(100, 10, activation='relu', input_shape=(TIME_STEPS, NUM_PARAMETERS)))
    model_c.add(Conv1D(100, 10, activation='relu'))
    model_c.add(MaxPooling1D(3))
    model_c.add(Conv1D(160, 10, activation='relu'))
    model_c.add(Conv1D(160, 10, activation='relu'))
    model_c.add(GlobalAveragePooling1D())
    model_c.add(Dropout(0.5))
    #model_c.add(Flatten())
    #model_c.add(Dense(100, activation='relu'))
    model_c.add(Dense(NUM_RESULTS, activation='softmax'))
    print(model_c.summary())

    return model_c
```

Figure 20 - Revised CNN Model

After testing out some different parameters a model will a 90% accuracy was built, in Figure 21 below it was only trained up to 250 epochs since after it would begin to diverge and overfit. The parameters where as follows: Epochs: 250, Batch Size: 20, Timesteps: 100, Step Distance: 20, Learning Rate: 0.0001, Validation Split: 0.3.

The model was then saved as an h5 file and loaded into another python script where it was evaluated using the keras evaluate() function. It is important to note that the Timesteps value for the testing data needs to be same as the value used for testing (since the input data length needs to be the same). However, the Step

Distance parameter can vary since it only determines the overlap in data and does not affect the model itself. As a result, it was determined that the out of sample accuracy varies greatly based on which step distance value you use.

To explore this effect the model was evaluated with datasets with step distance values ranging from 1 – 100. The best accuracy measured was 72.7% and all the evaluated accuracies are plotted in Figure 22 below.

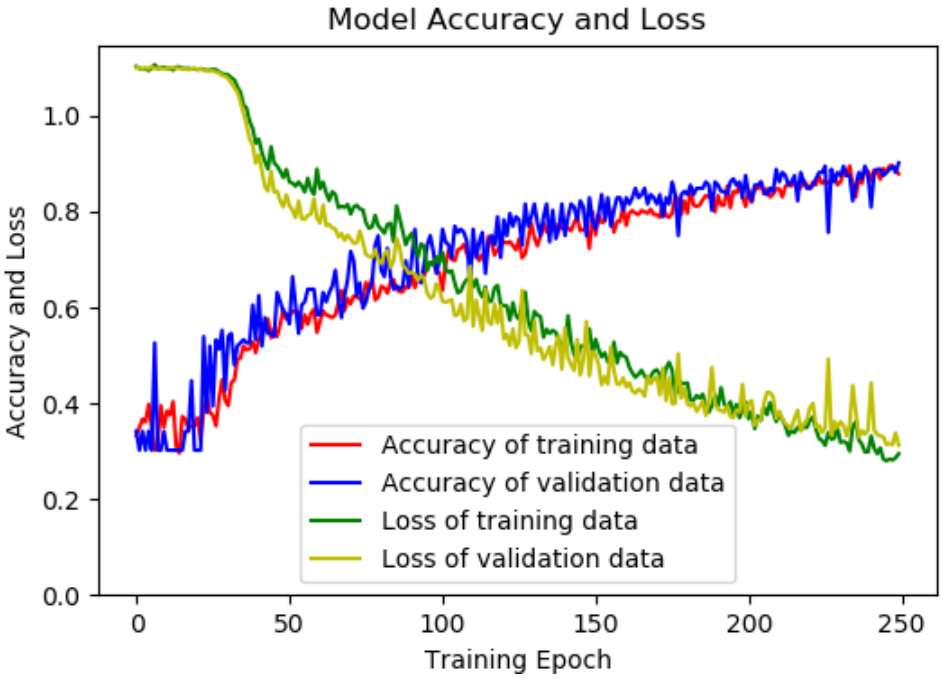

Figure 21- Epochs: 250, Batch Size: 20, Timesteps: 100, Step Distance: 20, Learning Rate: 0.0001, Validation Split: 0.3

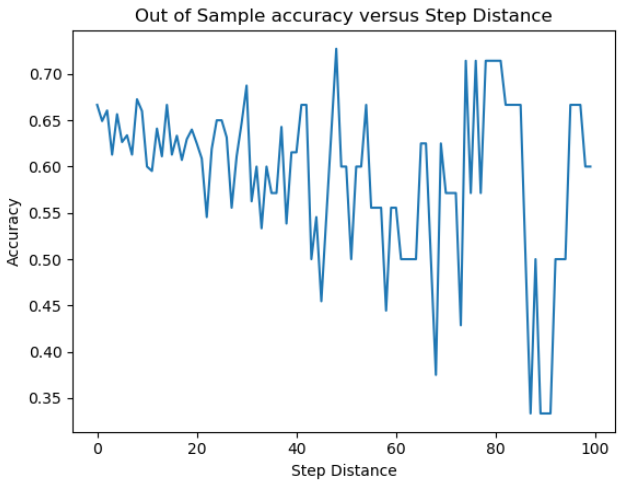

Figure 22 - Out of sample Accuracy versus Step Distance of testing data

As visible in Figure 22 the out of sample accuracy varies increasingly as the step distance increases. However, since this can be controlled for, you can decide to use the step distance that yields the best accuracy.

3.1.3 Improving Out of Sample Accuracy

As seen above, even models trained to accuracies of 90% and greater the out of sample accuracy was at best 73%, which is okay given the level of current optimization. However, I wanted to investigate whether I could increase it and generalize the model a bit more with some more optimization. The model was revised slightly so that the 2nd filter layer had a kernel size of 100. The parameters with the best results and the accuracy evaluation are below in Figure 23, Figure 24. This modification yielded a 97% in-sample accuracy and an 80% maximum out of sample accuracy.

Figure 23 - Epochs: 500, Batch Size: 20, Timesteps: 200, Step Distance: 25, Learning Rate: 0.0001, Validation Split: 0.4

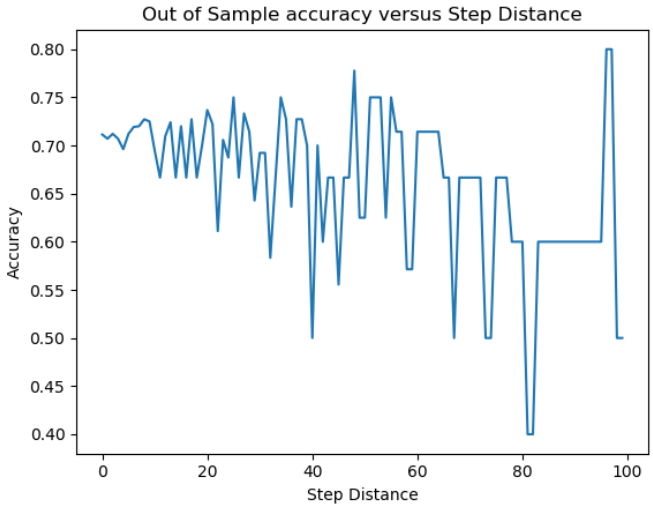

Figure 24 - Out of sample Accuracy versus Step Distance of testing data

4.0 DISCUSSION

4.1 Summary

Overall, from my investigation it appears that using a CNN for BLE occupancy detection has good potential. However, my investigation leads to more questions than answers, which will be discussed below. Additionally, there was so much more analysis that could be done, but for the scope of this project was left out or there was not enough time for it. For example, no analysis was done on the last dataset described – dataset3 which may yield interesting results; this was also coupled with my hesitation with the quality of data collected. Further, the tuning of the model to yield the best result was very rudimentary and the out of sample testing data was pretty small. However, despite this a working CNN model was developed which yielded a 97% in sample accuracy and a 80% out of sample accuracy that would distinguish between an empty room or one with one or two people. Although this project did not provide any conclusive results, it did provide a baseline of a potential solution and implementation, along with a range of possible improvements.

4.2 Improvements and Recommendations

Through testing and analysis there are a multitude of improvements, questions and recommendations that can be made, below is a list highlighting the major ones.

- Larger datasets collected, separate a larger out of sample dataset also to improve performance. With larger datasets the model could improve its accuracy but in particular its out of sample accuracy as well making it more generalized. A larger testing dataset could provide a better picture as to how well the model has generalized.
    - This could be simply collecting more data, sampling at a faster rate or measuring more paramters such as ToF, AoA.
- A more robust model optimization implementation. There are so many parameters that can be tuned to improve the model and method of model optimization that are left unexplored.
- There are a couple of implementation problems I foresee which I have not had the time to investigate:
    - Currently the model was being trained on data from a single room with the beacons in fixed positions. For practicality it would be interesting to investigate how it preforms and how well it can generalize to data from various rooms with the BLE beacons in varying positions.
    - Currently the best model inputs data with 100 timesteps which implies 100 measurements would need to be taken in order to make a prediction. Depending on how fast the data can be polled and what the lower bound is will affect how quickly the model can make a prediction.
        - Some models with a input data size of 25 timesteps were evaluated but they did not yield the best results.

REFERENCES

[1] J. W. G. S. (. Michael Meng (1), "BLE for Occupancy Detection".

[2] J. C-137, "Convolutional Neural Networks: Part 1," Medium, 26 September 2018. [Online]. Available: https://medium.com/@jon.froiland/convolutional-neural-networks-part-1-5eedc080e882.

[3] S. Verma, "Understanding 1D and 3D Convolution Neural Network | Keras," Medium, 20 September 2019. [Online]. Available: https://towardsdatascience.com/understanding-1d-and-3d-convolution-neural-network-keras-9d8f76e29610.

[4] M. Granat, "How to Use Convolutional Neural Networks for Time Series Classification," Medium, 4 October 2019. [Online]. Available: https://towardsdatascience.com/how-to-use-convolutional-neural-networks-for-time-series-classification-56b1b0a07a57.

[5] J. Brownlee, "How to Develop 1D Convolutional Neural Network Models for Human Activity Recognition," Machine Learning Mastery, 21 September 2018. [Online]. Available: https://machinelearningmastery.com/cnn-models-for-human-activity-recognition-time-series-classification/.

[6] N. Ackerman, "Human Activity Recognition (HAR) Tutorial with Keras and Core ML (Part 1)," Medium, August 2018. [Online]. Available: https://towardsdatascience.com/human-activity-recognition-har-tutorial-with-keras-and-core-ml-part-1-8c05e365dfa0.

[7] S. Sharma, "Epoch vs Batch Size vs Iterations," Medium, September 2017. [Online]. Available: https://towardsdatascience.com/epoch-vs-iterations-vs-batch-size-4dfb9c7ce9c9.

[8] P. Skalski, "Preventing Deep Neural Network from Overfitting," Medium, 7 September 2018. [Online]. Available: https://towardsdatascience.com/preventing-deep-neural-network-from-overfitting-953458db800a.

[9] Prabhu, "Understanding of Convolutional Neural Network (CNN) — Deep Learning," Medium, 4 March 2018. [Online]. Available: https://medium.com/@RaghavPrabhu/understanding-of-convolutional-neural-network-cnn-deep-learning-99760835f148.

[10] J. Yang, "ReLU and Softmax Activation Functions," Github, 11 Febuary 2017. [Online]. Available: https://github.com/Kulbear/deep-learning-nano-foundation/wiki/ReLU-and-Softmax-Activation-Functions.

[11] M. Stewart, "Simple Guide to Hyperparameter Tuning in Neural Networks," Medium, 8 July 2019. [Online]. Available: https://towardsdatascience.com/simple-guide-to-hyperparameter-tuning-in-neural-networks-3fe03dad8594.

[12] B. C. Md Fazlay Rabbi Masum Billah, "Unobtrusive Occupancy Detection with FastGRNN on," *Association for Computing Machinery*, p. 5, 2019.