POLITECNICO DI TORINO

Master degree course in Ingegneria Informatica
(Computer Engineering)

Master Degree Thesis

Camera-less context detection in mobile platforms

Supervisors
prof. Massimo Poncino
doc. Daniele Jahier Pagliari

Candidate
Matteo Ansaldi
matricola: 243847

Academic Year 2018-2019

Contents

Abstract

Ever since their introduction, smartphones have become more and more powerful, causing their battery life to become much shorter. However, now they offer an ever increasing array of sensors that are useful for multiple applications, including context detection. Most applications of context detection have been for activity recognition, and very few have been used for power management. The goal of this study is to evaluate the feasibility of implementing a deep learning classifier on an Android device to save power when the device changes from an “in use” state to a “not in use state”.

To this end, this work explains how data was collected to train a Convolutional Neural Network for the task, how the neural network was trained and how parameters were selected, and an analysis on which sensors were most important for the classification, and which could be removed in a later work.

Results show that a Convolutional Neural Network works very well for this case, and proves that with just 2 seconds of data collected from few low-powered sensors, in 96% of the cases, the classifier will recognise that the device is not in use. This allows for an energy savings of 59% during a transition from "in use" to "not in use" compared to a camera-based solution, and up to 91.1% compared to a standard timeout-based solution.

Chapter 1

Introduction

Ever since smartphones have been first introduced to the public, they have become more and more powerful, offering an ever increasing array of sensors and becoming a part of everyday life. The ever increasing speed of these devices opens up the possibility of using deep learning algorithms on them, in particular in the field of context detection. By using the sensors available on these devices, information about what the user is doing can be inferred, for example by looking at the movement of the device or the position in which the device is being held. Examples of context detection include fitness tracking (frequently used in devices such as Fitbit or MiBand), location-based context detection (for example used by Google assistant to infer what a user is doing), or activity detection (used by Google's Activity Recognition API). However, this large increase performance also comes at the cost of battery life.

The general goal of this work is to evaluate the feasibility of using context detection to understand if a device is in use or not, opening up the possibility of using this context detection to rapidly switch the devices between different states, saving battery every time the device passes from "in use" to "not in use".

Although datasets for smartphone context detection are widely available online, none of them cover the classes we are interested in ("device in use" and "device not in use"). Because of this, the first necessary step was to create a dataset containing the necessary sensor data and classes. Due to its wide availability and ease of access to the available sensors, Android was chosen as the operating system on which to collect data. An android app was created for the purpose, and collected data from multiple sensors, including accelerometer, light sensor, proximity sensor, touch events, gyroscope and

microphone.

To easily classify the data, the front-facing camera would take a photo every 10 seconds, and the app would check to see if a face was visible. If it was, the device was considered "in use" otherwise it would be considered "not in use". While this classification was not perfect, the people collecting data were informed about the limitations of the app, and avoided collecting data in conditions where the classification would not work (such as in a dark environment).

Different options were explored for the classifier. Initially a Fully Connected Neural Network was used, but after providing a very poor performance it was discarded. A Convolutional Neural Network was then considered for the task, given that previous experiments suggest that they perform very well with context detection on smartphones. PyTorch was chosen as the framework to develop and build the network, due to its ease of use and its deep integration with Python. It provides good speed for training and executing, along with a large amount of flexibility for modifying the network.

Having collected all the data necessary and chosen the classifier, another step was to clean and format the data in such a way that the classifier could be easily trained and tested on the collected dataset. A 2 second window was selected for classification, so all the data that was outside that window was removed, and all data inside the window was standardized such that all sensors gave readings at the same frequency. The sensor sampling frequency was chosen to be 10Hz, allowing the sensors to give useful data for context detection, but not so high that battery life would be significantly impacted.

After further experimentation, the microphone, gyroscope and linear accelerometer were discarded from the dataset. Previous experiments with context detection had found that the microphone data didn't provide any useful patterns, and made the prediction worse. Moreover, this input would have generated a large amount of data which would have increased the Convolutional Neural Network complexity. The gyroscope and linear accelerometer were also discarded because most lower-end Android devices don't have those sensors. Uncalibrated accelerometer, light sensor, proximity sensor and touches were kept, due to their wide availability and low power consumption. All of those sensors are running while the device is in use, so using them wouldn't increase power consumption. The data was transformed into a 6x20 "image", where each row represented a sensor (light sensor, proximity sensor, touches on a screen, x, y, z values of the accelerometer). Each column represented a fraction of a second: we had 20 columns due to the sampling frequency (10Hz) and the window width (2 seconds). The camera input was

used as the class label: 0 for "not in use", 1 for "in use".

This data was randomly split into 10 subsets, and the network parameters were evaluated using a 10-fold cross validation approach to avoid outliers. Initial network parameters were chosen starting from previous context detection experiments, and were modified to improve performance through trial-and-error. Parameters that have been explored included the shape of the final fully connected neural network, the number of training epochs, the format of input data, the learning rate, batch size, activation function, decaying learning rate, data normalization, and removing sensors.

The last test in particular showed that the performance of the network would slightly degrade by removing sensors, underlining the fact that all sensors were actually useful for the classification and weren't adding noise to the input data. The only truly vital sensor was the accelerometer, which would cause a huge degradation of performance when removed.

Given that the input data had largely unbalanced classes (all devices were more often not in use than in use), multiple formulae were used to evaluate the network performance, such as precision, recall and f-score.

Results were overall very positive: the best case scenario gave an average f-score of 0.785, with a standard deviation of just 0.03 over all 10 folds. The average accuracy for "not in use" was 0.959, meaning that in 95.9% of the cases after 2 seconds of data collection, the algorithm would be able to recognise that the device wasn't being used. Accuracy for "in use" was lower, at 0.782, but this is mitigated by the possibility of overriding the algorithm's result depending on other information that is available to the device, such as knowledge that a game is being played or that a video is being reproduced. Moreover, automatically detecting that the device is in use is evidently less important than detecting that is isn't in use for energy management purposes.

Knowing these results, comparisons can be drawn between using this algorithm, a simple timeout, and using the front-facing camera to recognise when the device isn't in use and to switch off unnecessary functions. A simple timeout uses no extra power to be run, but is limited to the set time for timeout. If given a timeout of 15 seconds, the device will always take exactly 15 seconds before transitioning to a standby state, wasting battery power. Using the front-facing camera would allow the device to recognise that it isn't being used in less than 15 second, but would be limited by the outside environment (it wouldn't work in a dark room), and would waste large amounts of power by taking and analysing photos periodically while the device is in use. Finally, the proposed neural network based solution wouldn't use much more power than the regular timeout solution, given that

all the used sensors are already active while the device is in use, and would use a minimal amount of extra power for the classification. According to our results, the network has an overall accuracy of 92.98%, and a specificity of 95.66% after collecting 2 seconds of data. This means that the neural network based approach consumes 59% less power on a transition from "in use" to "not in use" compared to the camera-based solution and 91.1% less than the timeout-based solution, assuming a timeout of 10s for the display, and 30s for the transition into idle state.

Implementing this solution on a smartphone is part of envisioned future work, allowing us to properly measure and compare the energy savings between current solutions and the proposed one.

Chapter 2

Background and Related Work

2.1 Activity and Context Recognition

Ever since the development of smartphones, there have been multiple attempts to use the sensors available on these devices for context detection. The idea of context detection is to use the data available from the sensors of a smartphone (such as phone radio, accelerometer, gyroscope, light sensor, proximity sensor, microphone, etc.) to infer contextual information about the user.

This contextual information could be used to predict what actions the user is doing [1][5][6], or it could be used to predict how the user is going to interact with their device in the near future [21]. Some studies on context detection that are related to this thesis are summarized below.

Wu et al have analysed using sensor fusion for context understanding in Human-Computer Interaction [3]. They initially analysed what common contextual information can be useful for Human-Computer Interaction, making it easy to decide which information should be included in the system design depending on the effectiveness versus the cost. They proposed 6 different categories of useful information for context detection:

- *Location*: City, altitude, weather, location, orientation, travelling, speed, heading
- *Proximity*: Closeness to a building, room, car, change of proximity
- *Time*: Day, date, season, schedule

- *People*: Presence of other people, social relationships
- *Audio-visual*: Human talking, surrounding scenery, noise level
- *Computing & Connectivity*: Network Connectivity, computing environment

F. Attal et al have used wearable sensors to recognise human activities using multiple different classification techniques [5]. Data was collected using 3 inertial units placed on the chest, right thigh and left ankle of the participants. The sampling frequency was set to 25Hz, which was considered to be sufficient to measure daily human activities. 12 different categories were defined, including standing, sitting, stair descent, lying down, walking, etc.

Two different techniques (supervised and unsupervised learning) were used to recognise the activities. For supervised techniques, k-Nearest Neighbour (kNN), Support-Vector Machine (SVM), Gaussian Mixture Models (SLGMM), Random Forest (RF) were selected. Instead for unsupervised techniques, k-Means, Gaussian Mixture Models (GMM) and Hidden Markov Model (HMM) were used. Models were trained both with raw data and manually extracted features. 10-fold cross validation was used to create the training set and test set.

Evaluation of the different algorithms was done using Accuracy, Precision, Recall, F-measure, and Specificity

In the case of raw data, supervised algorithms outperformed unsupervised algorithms in all cases. The k-NN algorithm gave the best results overall (Accuracy: 96.53 ± 0.20, F-Measure: 94.60, Recall: 94.57, Precision: 94.62, Specificity: 99.67), followed by RF, SVM, SLGMM [5].

For extracted features, 11 time-domain features and 6 frequency-domain features were extracted. With this data, both supervised and unsupervised techniques improved their performance, but supervised techniques still gave the best results overall. k-NN still had the best overall performance (Accuracy: 99.25 ± 0.17, F-Measure: 98.85, Recall: 98.85, Precision: 98.85, Specificity: 99.96).

This study proved that it is very possible to accurately infer what the user is doing using only accelerometer data, even when there are a large amount of different possible classes.

U. Christoph et al created a context-detection prototype to detect if the user is at home, at university, cycling, on a bus, on a train, or walking on an Android G1 smartphone [6]. As inputs, they used radio signals (GPS, UMTS, WLAN), the camera for light detection, microphone, accelerometer, compass.

They used a neural network recognise a set of predefined patterns to infer the user's context. With all sensors, the detection rate was 80%. Once the neural network was implemented on the device, they tried removing one sensor at a time from the input, to see if the result improved. As shown in Figure

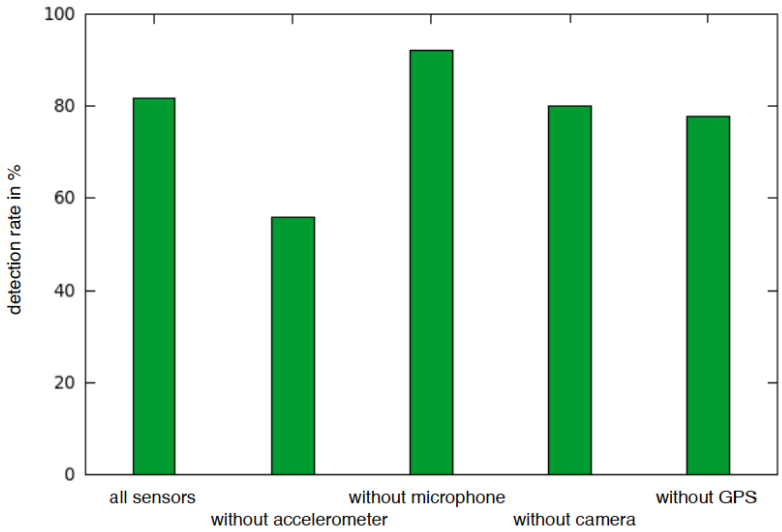

Figure 2.1. Performance of the classifier, removing sensors [6]

2.1, the performance of the classifier improved when the microphone was removed. The microphone wouldn't provide any easily recognisable patterns due to the noise and the distortion effects.

Narseo Vallina-Rodriguez and Jon Crowcrof studied how contextual information could be used to extend the battery life of a smartphone without compromising the user's experience [4]. Using data collected off 18 Android users during 2 weeks [21], they proved that spatial context affects how much the battery of the phone is used, and depends a lot on the person using the device as shown in Figure 2.2. They claim that the two often used resources (GPS and Cellular Interfaces) aren't always available, depending on the location of the phone. Because of this, by knowing the context of the phone by using other sensors the way these resources are used could be modified to save battery.

The energy consumption of the *cellular interface* depends on the quality of the signal the device is receiving. The worse the signal is, the more often data must be retransmitted, consuming more energy. Given that the signal strength is linked to the location of the device, knowing the device's location can give us information that can be used to reduce the amount of energy

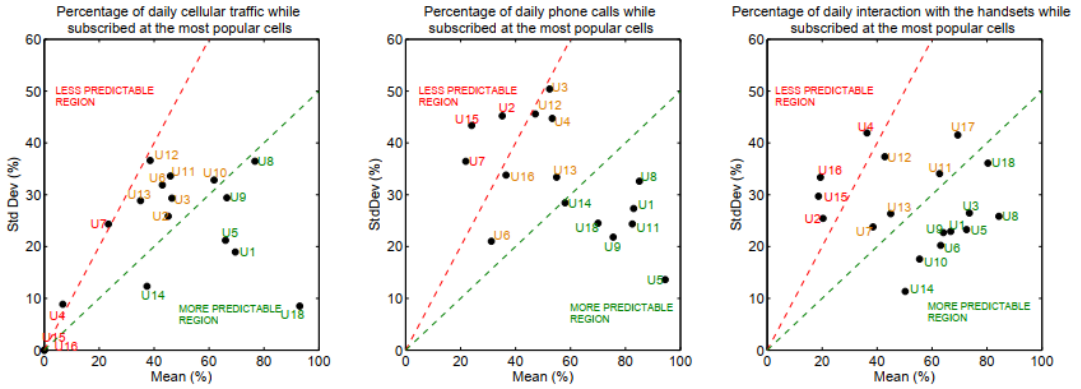

Figure 2.2. User classification by their percentage of the usage/interaction with the 3G interface, telephony service and screen while subscribed at the most common cells (likely to be users' workplace and home). The x axis represents the daily average usage and the y axis the standard deviation. This information can be used to identify the places where the energy consumption will be higher and also to infer the predictability of the user interaction and the state of a resource. [4]

used [4].

Concerning *location sensors*, the authors have shown that continuous location sensing can be very battery intensive, Vallina-Rodriguez et al analysed multiple different solutions to reduce the energy consumption of those sensors. In particular, [22] offered 4 different methods to reduce the battery usage: *Substitution and Suppression*, *Piggybacking*, *Adaption*, and *Probabilistic Models*.

Substitution and Suppression reduce the consumption of location sensing by using lower-powered sensors in place of GPS. Substitution decides when to use lower-powered sensors such as mobile networks instead of GPS, taking into account the available accuracy. Suppression blocks GPS polling by using alternative low-power sensors to gather information on if the device is moving or not (such as the accelerometer). *Piggybacking* forces different applications to synchronise their location-sensing requests, reducing the overall time that the GPS sensor is in use without compromising on user experience. *Adaption* adjusts sensing parameters such as polling frequency depending on the available battery power. A lower battery power could mean a lower polling frequency, so as to reduce battery consumption. *Probabilistic Models* can be used to infer future user locations so as to reduce the number of times the

location sensors are polled. Finally, they presented 2 different mobile operating systems (*CondOS*, *ErdOS*) that implement some of the previously listed solutions, one that tries to leverage the interaction between applications and operating system without necessarily being context aware, while the other uses context-detection to manage resources seamlessly for all applications.

Radu et al have analysed sensor fusion approaches for shallow and deep learning classifiers, and compared how the techniques fare with existing practices, under what circumstances to use deep learning, and which type of techniques should be deployed [1]. Two different types of deep learning architectures were studied:

- Feature concatenation (FC): The inputs of the various sensors are concatenated at the input of the neural network. This implies easier training, but is much harder to specialize [1]. Moreover, if one particular input generates a significantly larger amount of input data compared to others, it may cause bias in the classification.

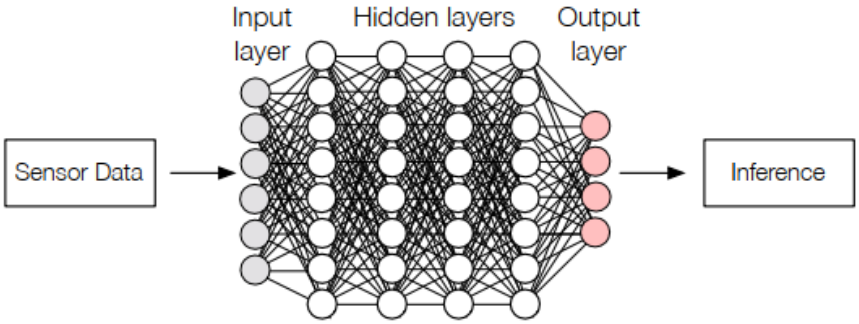

Figure 2.3. Feature concatenation [1]

- Modality-Specific architecture (MA): This architecture uses two types of hidden layers, multiple sets of layers that are specific to a sensor, then a second set of hidden layers that unify the outputs of the previous hidden layers. This allows to finetune each network to increase the performance for their specific sensor.

These two architectures have been used with two different types of neural networks:

- Deep Neural Networks (DNN)
- Convolutional Neural Networks (CNN)

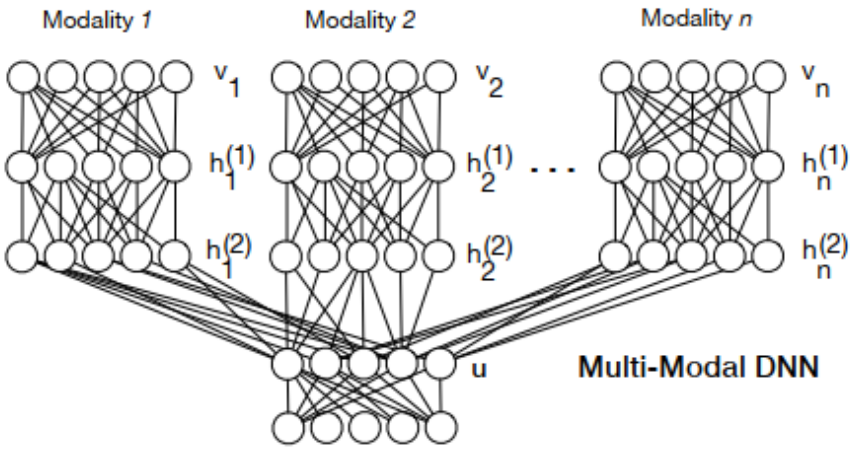

Figure 2.4. Modality-Specific architecture [1]

The two shallow learning techniques that were compared to the deep learning techniques were:

- Random Forest (RF)
- Decision Tree (DT)
- 5 other purpose built shallow classifiers were used for the GAIT dataset exclusively.

The comparisons between shallow and deep learning have been run on two mobile systems-on-chip; **Qualcomm Snapdragon 400** and **Qualcomm Snapdragon 800**. The datasets used for comparison were:

- STISEN dataset [18], containing readings of the Accelerometer and Gyroscope of users performing 6 different activities.
- GAIT dataset, containing data from Accelerometer and Gyroscope sensors, separated into 5 different types of walking.
- Sleep-EDF database [19], containing physiological data annotated with 6 sleep stages.
- Indoor-Outdoor dataset [20], containing data from light, proximity, magnetic, microphone, cell, battery thermometer sensors, divided into indoor or outdoor classes.

The performance between RF, DT, FC-DNN, FC-CNN, MA-DNN, FC-DNN were compared using the F1-score. Having tested the performance of all previously listed algorithms on the datasets, deep learning classifiers were found to outperform shallow classifiers across all 4 datasets, with the deep classifiers having an average accuracy difference over shallow classifiers of 27%. CNN-based architectures outperformed all other architectures in all except GAIT dataset [1].

Modality-Specific architectures outperformed Feature Concatenation architectures in all datasets, however Feature Concatenation deep learning would still outperform the shallow classifiers by 24% in many cases [1].

Deep classifiers on average would execute in less than 10ms, with at most an energy consumption of 18.7mJ on the **Qualcomm Snapdragon 400** using Linux, while the execution time on the **Qualcomm Snapdragon 800** was at worst 134.4ms, with an energy consumption of 220.5mJ on Android. The difference between the two hardware devices was mostly due to the Android Scheduling System, which resulted in much lower performance [1].

They also explain some assumptions and decisions they made while developing the code [1]:

- *Learning Rate*: They found that values between *1e-1* and *1e-5* gave the best results, depending on the dataset.
- *Number of Epochs*: They restricted the number of epochs to 400 with an adaptive learning rate
- *Network initialization*: Auto-encoders con extract features from unlabelled data by reproducing the input to the output on sections of the network. However, given that the datasets were already labelled and the network had a small number of layers, the impact of auto-encoders was minimal.
- *Dropout Ratio*: A dropout layer was implemented to avoid reliance on a few dominant neurons
- *Batch Normalization*: This layer allowed for a higher learning rate to be used, speeding up the training process.

2.2 Artificial Neural Networks

Due to their ability to manage complex inputs, the ever increasing power of GPUs and distributed computing, and the wide availability of large datasets,

Artificial neural networks have gained a lot of popularity in particular from the data analysis community [9]. Due to their ability to recognise patterns in different images or for speech recognition, they have become an invaluable tool that is widely in use today. Before Artificial Neural Networks, classification algorithms had to be manually engineered to recognise the different classes, a task that was made even more difficult by having to identify specific identifying features of each class manually [16]. Neural Networks use feature learning or representation learning to automatically discover (learn) the representations needed for classification or feature detection. Two main methods of feature learning exist:

- Supervised learning
- Unsupervised learning

In *supervised learning*, the input data is labelled with it's respective class. The network will attempt to learn what features identify that class, so as to be able to classify unlabelled data once it has been fully trained. The training data is run through the network multiple times, and the output is compared to the expected output (the class label). A loss function calculates how "wrong" the output is, and that difference is back-propagated through the network. In *unsupervised learning* or Hebbian learning, the input data is unlabelled. The network will attempt to group together data with similar characteristics, through clustering or similar methods.

Artificial Neural Networks are software systems that use a set of interconnected elements, called Neurons [8]. They can be seen as a framework that allows for multiple machine learning algorithms to work together to analyse complex input data. Inspired by the workings of a human brain, they are typically used for machine learning applications such as classification or regression. The network itself is not an algorithm, but a collection of multiple machine learning algorithms that work together and interact to process complex an unexpected input data. There are three types of popular neural networks, each specialized in one field of application:

- Fully connected neural network
- Convolutional neural network
- Recurrent neural network

Fully connected neural networks are the simplest of the three architectures. They use three different types of layers; Input, Output and multiple Hidden layers. They are most often used for classification problems.

Convolutional neural networks use Fully connected neural networks, but specializes in image recognition. They place convolution and pooling layers before the fully connected network so as to extract features from adjacent pixels in the analysed image.

Recurrent neural networks, unlike the previously listed networks, have a feedback loop, allowing for the network to maintain a "memory state" of what it has previously analysed. This makes them particularly good at analysing sequential data, as they can make connections between subsequent inputs.

2.2.1 Neurons

Individual neurons in a neural network contain an activation function [8]. These activation functions "activate" or "deactivate" a neuron depending on the input values [7]. Neurons tend to do 3 main jobs:

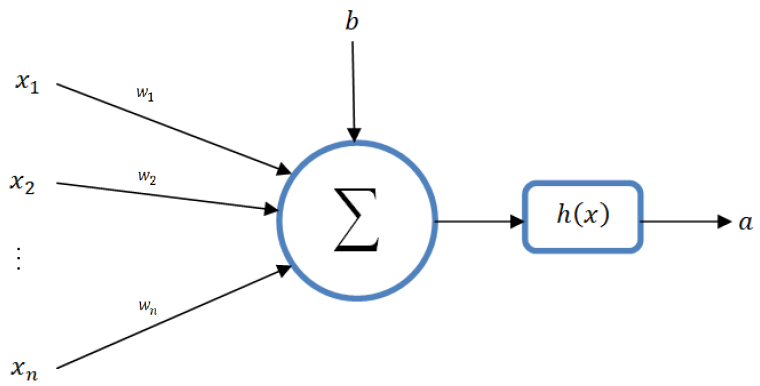

Figure 2.5. Single neuron

- Sum of all weighted inputs.
- Application of a bias.
- Calculation of the activation function.

The weights are updated during the learning process, and are one of the most important elements that define the neuron's output, modelling different strengths between the inputs and the final output. The bias shifts the activation function along the x plane, allowing the network to simulate conditional relationships. There are multiple possible activation functions available to use [7], some are listed below:

- Step function

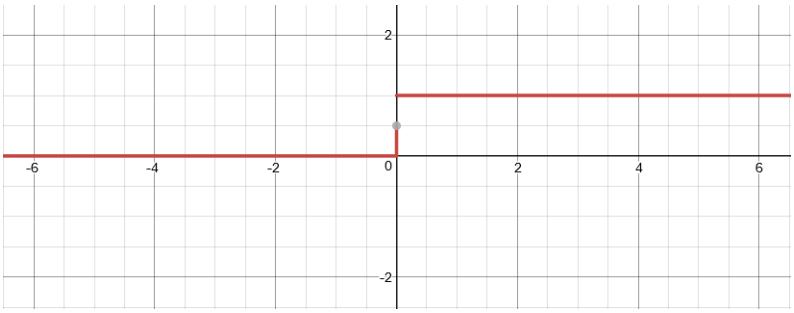

Figure 2.6. Step function

$$h(x) = step(x) \tag{2.1}$$

This function is rarely used due to it's binary nature. Because a neuron can only assume the value 0 or 1, the network becomes harder to train. This is because training depends on the calculation of the derivative of the activation function, and this function can't be derived for x = 0.

- Sigmoid function

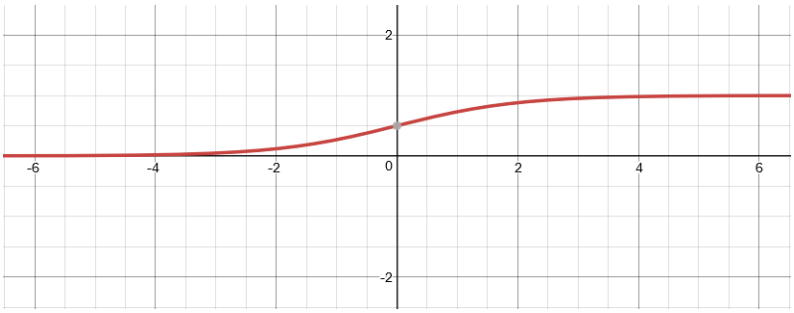

Figure 2.7. Sigmoid function

$$h(x) = \frac{1}{1 + e^{-x}} \tag{2.2}$$

This function allows neurons to assume any value between 0 and 1, assisting in the training of the network. However, it can cause the problem of "vanishing gradients" when the x value is too high or too low, causing the derivative to become nearly 0. Because of this, it becomes difficult to back-propagate the gradient towards the preceding layers.

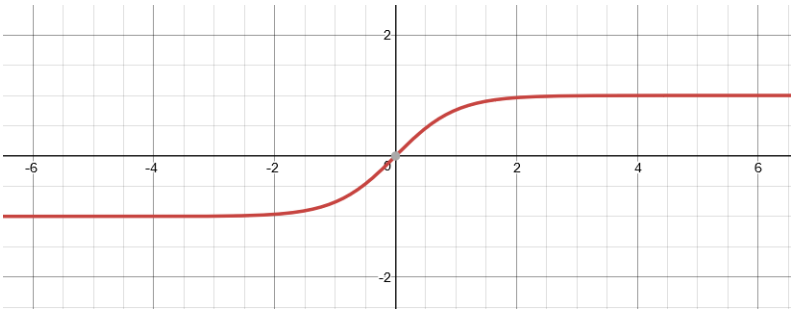

Figure 2.8. tanh function

- Tanh function

$$h(x) = tanh(x) = \frac{2}{1 + e^{-2x}} - 1 \tag{2.3}$$

This function has very similar characteristics to the sigmoid function. The gradient is much stronger than that of the sigmoid [13], so the choice between this function and sigmoid depends on the requirement of the task at hand.

- ReLU function

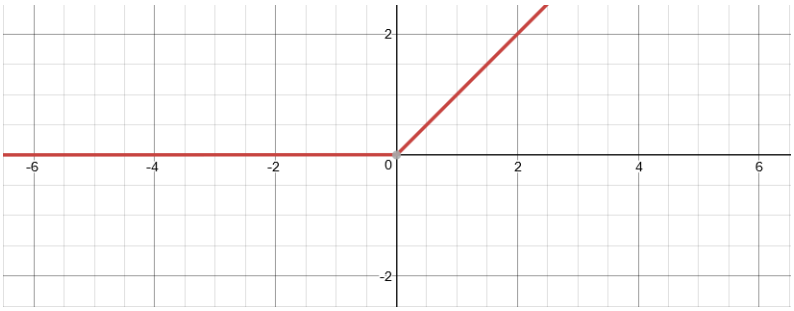

Figure 2.9. ReLu function

$$h(x) = max(0, x) \tag{2.4}$$

This function doesn't saturate for any value in input. Because all values < 0 give as output 0, large parts of the network will not be activated making the network lighter. However, this function also can cause a "dying ReLu problem", where parts of the network will never fire, thus never updating the weights of the neuron. By changing the function to a Leaky ReLu, this problem is avoided [14].

$$h(x) = max(0.01x, x) \tag{2.5}$$

This function is often used in Convolutional Neural Networks.

- SELU function

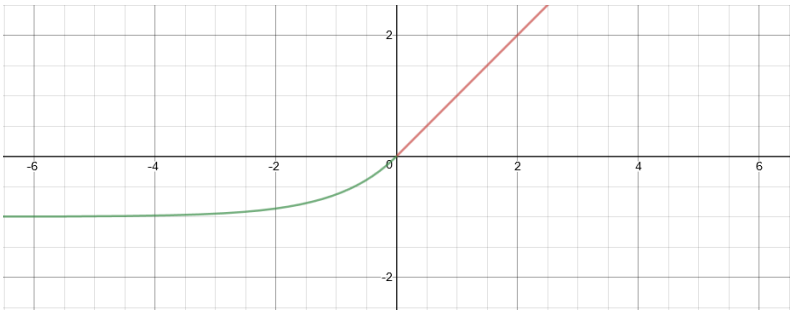

Figure 2.10. ReLu function

$$h(x) = \lambda \begin{cases} x & \text{if } x \geq 0 \\ \alpha e^x - \alpha & \text{if } x < 0 \end{cases} \tag{2.6}$$

α and λ are fixed parameters, meaning we don't modify them during backpropagation. Much like the leaky ReLU function, when $x > 0$, the output increases as x increases. However, the behaviour when $x < 0$ changes slightly compared to ReLU and leaky ReLU, with the output going below 0.

2.2.2 Layers

Fully connected neural networks are composed of 3 different types of layers; *Input*, *Hidden*, *Output*.

The Input layer receives an *Nx1* tensor of data, where N is the number of input neurons. Often the data must be flattened to be used as an input (for example, if an image is $20px * 20px$, the input layer will be of size *400x1*). All data coming from outside the network will enter this layer. Each input node represents a variable of the input data.

There can be multiple hidden layers contained in a single neural network. In the case of a Fully Connected Neural Network, all nodes from one layer are connected directly to all the nodes in the following hidden or output layer. For other network architectures (such as convolutional neural networks), there can be different network topologies. These layers try to generate useful representations of the input data. They contain activation functions that, once properly trained, will activate when recognising specific

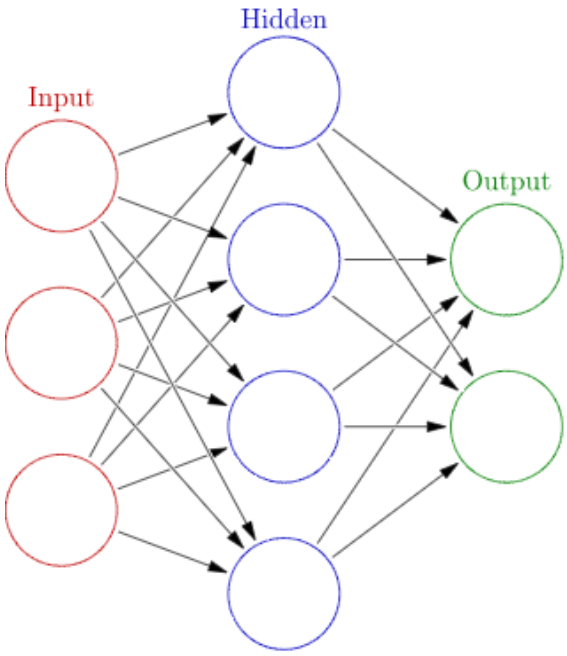

Figure 2.11. Example of a simple fully connected neural network [10]

patterns in input. Each layer will produce different representations of the input data, so as to extract features that initially are not obvious. By combining and ranking the outputs of all the nodes in the hidden layers, the network is able to figure out the overall pattern in input, allowing the output layer to give an appropriate response to the input data.

The output layer provides the result of the neural network's classification. Depending on what input pattern it identified, the output will change depending on how the network was originally trained. The number of output nodes depends on the type of classification required; in the case of a binary classification, one node is all that is necessary. In the case of multiple classes (for example, recognising a written number between 0 and 9), there will be one output node per class, that provides both the classification result and it's respective confidence level. This allows for the loss to be computed while training the network.

The number of neurons in the hidden layers has large influence on how the network will perform. Using too few neurons could cause underfitting, where the network cannot properly model or transform the data, causing it to not be able to recognise patterns. This will cause bad performance on the training data. Too many neurons could cause overfitting, where the network

starts to identify noise as important information in the patterns. This will cause the network to perform well on the training data, but will not be able to generalise information, causing poor performance on data that wasn't used for training.

2.2.3 Training

Models using unsupervised learning have to figure out connections or common patterns in the training data. There are multiple different types of unsupervised learning, such as clustering (k-means, DBSCAN, hierarchical clustering, etc.), Anomaly detection, or Hebbian learning. These methods have the great advantage of not needing the data to be classified beforehand, allowing for unexpected classes or associations to come to light. Supervised Learning is the most often used method for training a neural network [16]. With supervised learning, the network is trained on a large set of already classified data, by running the data through the network in batches (feed-forward phase), calculating the error compared to the expected result (loss calculation), then backwards propagating the error, updating the weights on the individual neurons so as to lower the error (backpropagation phase) [8].

During the **feed-forward phase**, each output neuron will contain an associated score. The neuron with the highest associated score will define the class of the input data. This allows for the **loss calculation** to calculate an error using a cost function. The cost function can be made ad-hoc, or standard functions can be used (such as cross entropy, binary cross entropy, quadratic, exponential, etc.) [7].

Once the loss has been calculated, **backpropagation** is used to update the various weights contained inside the neural network [7]. It uses the chain rule to allow for the gradient descent optimization algorithm to be used to update the weights of the inputs of each neuron by calculating the gradient of the loss function. This algorithm doesn't guarantee that the global minimum will be found, as the algorithm could end up stuck in a local minimum. However, given a large enough network and enough input data, the chance of getting stuck in a local minima becomes extremely small [17].

2.2.4 Example of a Neural Network

Taking Figure 2.12 as an example of a simple neural network, the first layer containing the values x_1 and x_2 is the input layer. The second layer is a hidden layer, while the last neuron is the output layer, given that this

network has to do a simple binary classification. The value $h_1^{(3)}$ is the output of a feed-forward phase.

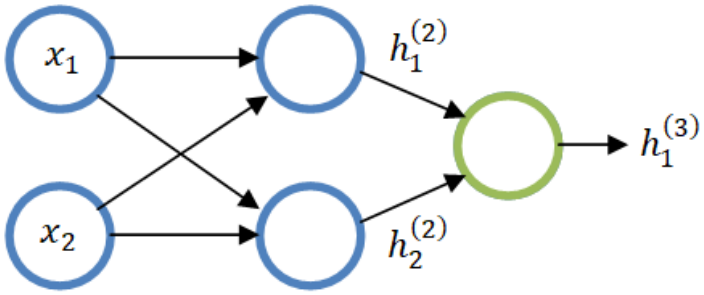

Figure 2.12. Simple neural network

We state that:

- $h_i^{(l)}$ is the output of node x_i in layer l
- $w_{ij}^{(l)}$ is the weight of the connection between nodes x_i in layer l and x_j in layer $l+1$
- L is the loss function
- $f(z)$ is the activation function
- $b_i^{(l)}$ is the bias of node x_i in layer l
- B is the set of biases $b_i^{(l)}$
- $z_i^{(l)}$ is the sum of all inputs into a node x_i in layer l
- W is the neural network's weights
- S^r is the input of a single training sample
- E^r is the desired output for the corresponding input sample S^r

Thus, we can state that:

$$h_i^{(l+1)} = f(\sum_{i=1}^{n}(w_{ij}^{(l)} h_i^{(l)}) + b_i^{(l)}) = f(z_i^{(l)}) \tag{2.7}$$

We consider, for example, a Cross-Entropy loss function:

$$L(W, B, S^r, E^r) = -\sum_{i=1}^{n}[E_i^r \ln h_i^{(l)} + (1 - E_i^r) \ln(1 - h_i^{(l)})] \tag{2.8}$$

We can calculate how much a change in a weight $w_{ij}^{(l)}$ affects the loss function L by using the chain function 2.9.

$$\frac{\partial L}{\partial w_{ij}^{(l)}} = \frac{\partial L}{\partial h_j^{(l+1)}} \frac{\partial h_j^{(l+1)}}{\partial z_j^{(l+1)}} \frac{\partial z_j^{(l+1)}}{\partial w_{ij}^{(l)}} \tag{2.9}$$

This function is then iterated over going backwards along the neural network layers, allowing the gradient to be calculated with respect to the network parameters of preceding layers.

2.3 Convolutional Neural Networks

Convolutional Neural Networks allow for state-of-the-art image classification [15], by exploiting correlations between adjacent inputs in a given image or time series. Three main layers are used in this type of neural network; *convolutional*, *pooling*, *normalization* [13].

The convolutional layer is a moving window or filter across the data being studied. Given a size of the window, each point in it is assigned a weight. The filter will run across the set of data, multiplying the data points according to the weights, and then accumulating the results. This process will be repeated for all possible positions in the set of data. Because the convolutional layer

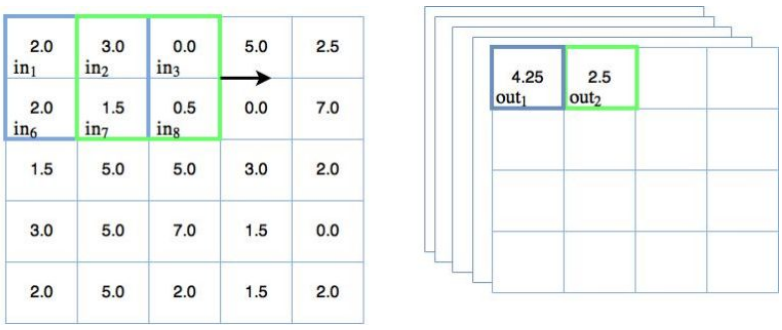

Figure 2.13. Multiple convolutional 2x2 filters [12]

is used to discover certain features of adjacent data points, it will be run

multiple times with different assigned weights, creating different channels that will be trained to recognise certain key features in the dataset.

In the pooling layer, a stride and a window size is defined. The window will analyse all data points contained in it and apply a statistical function on them. The most common type of pooling (Max pooling) is performed by taking the largest value in the window. This procedure helps to reduce the amount of data being analysed, and generalise lower level data [13].

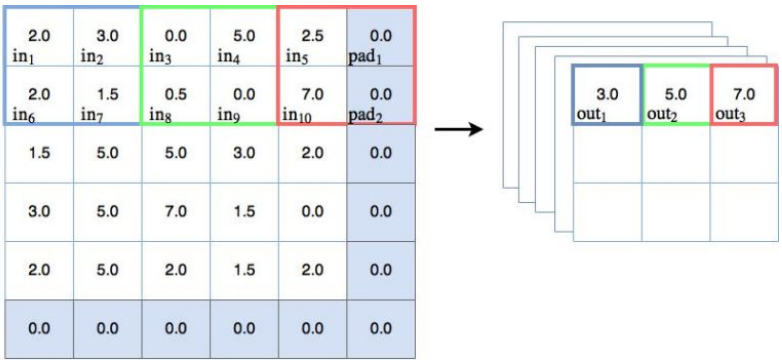

Figure 2.14. Max pooling example (with padding) [12]

Convolution and pooling layers generally use ReLU activation functions, that provide a non-linear behaviour that is characteristic of neural networks. It helps to limit the complexity of the computations by setting negative values to 0 (or making them very small, in the case of leaky ReLU). Differentiation for training and back-propagation becomes extremely easy to calculate.

The batch normalization layer applies a transformation that usually forces the mean activation to be 0 and the activation standard deviation to be 1. This speeds up training of the neural network by avoiding the need of extra training to make each layer of the network adapt to a distribution.

The dropout layer is a solution to avoid overfitting in neural networks by "dropping" or ignoring certain neurons during training. This avoids overreliance on one specific neuron, and allows the network to develop in a more uniform way. This layer is normally used only during training, and disabled during inference.

The section of the network with the previously listed layers will normally be followed by one or more fully connected layers, as shown in Figure 2.15. The data in input to the fully connected layer will have to be flattened to a single *Nx1* tensor, allowing the fully connected layer to interpret the results and produce a classification result.

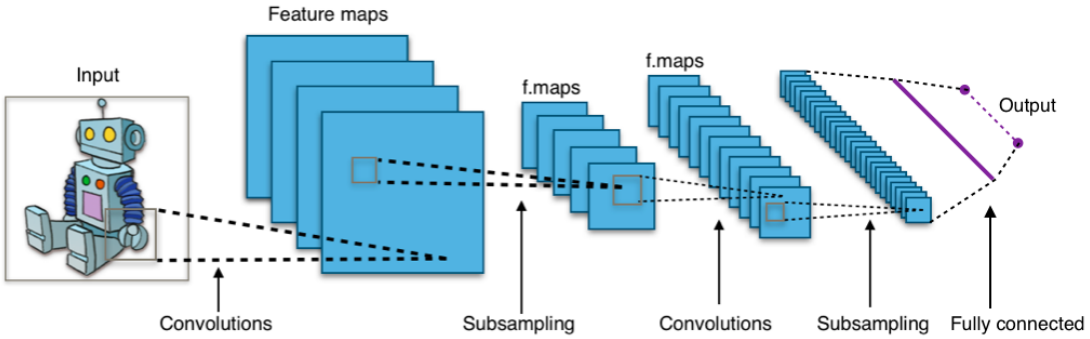

Figure 2.15. Convolutional Neural Network [11]

The figure shows how the input data is passed through a series of convolutions, creating multiple channels, and pooling layers (subsampling), reducing the size of the data. Finally the data is flattened before being passed through the fully connected layer, producing the classification result.

Chapter 3

CNN-based Camera-less user attention detector

3.1 Motivation

Smartphones are now an integral part of out daily lives. As time has progressed, they have become more powerful and offer us an ever-increasing array of sensors to use. Their displays have increased to the point where 6" displays are the norm, causing the battery life to become shorter and shorter. We propose a solution to reduce the energy consumption of smartphones when they transition from "in use" to "not in use". At the moment the standard is a simple timeout, which leaves the display on for a fixed amount of time before switching it off and starting to disable other non-vital functions. Our solution aims to understand in as little time as possible when the device stops being used, without using sensors that could be cause for privacy concerns (such as the front-facing camera) or sensors that would cause battery drain while active. This would allow for the device to turn the display and non-vital functions off much faster. Secondarily, it can also be used to avoid scenarios where the device is in use, but switches off because the user hasn't touched the screen before the timeout has ran out.

3.2 Sensors used for collecting data

We considered and collected data from multiple sensors that are available on regular smartphones, including;

- *Gyroscope*

- *Light sensor*
- *Linear Accelerometer*
- *Uncalibrated Accelerometer*
- *Microphone*
- *Proximity sensor*
- *Touches on the screen*

The gyroscope and the two accelerometer sensors were selected because they give information about the orientation and relative movement of the device. The proximity sensor and the light sensor when taken together can give information about if the device is in a pocket, face-down on a surface, or if the user is in a dark environment or not, giving calculates as to the time of day or if the user is indoors or outdoors. The microphone gives important clues as to the social context of the user; we can try to listen for other people or loud sounds. The touch events give a strong indication that the user is interacting with the device, which we assumed would be useful for our type of classification. However, only a subset of the sensors were used for training the neural network, as explained in 3.4.

To understand if the device was in use or not while the data was being collected, we decided to use the *Front-facing Camera* to classify the data by periodically taking a photo and analysing it to verify if a face was visible or not. This is not a perfect classification, but is good enough for our needs: a user normally won't point their phone's screen towards their face when they aren't using the device.

3.2.1 Camera

The data collected from this sensor was used to classify the data. The front-facing camera would take 1 photo every 10 seconds. The frequency of photos was chosen as a trade-off between collecting large amounts of data and the performance of the device collecting the data. To make classification easier, the java class FaceDetector [23] provided by Google was used to check if a face was visible to the front-facing camera. Due to limitations of FaceDetector, the photo would be rotated and analysed 3 times: once for if the phone was being held in portrait mode, twice for if the phone was being held in either of the two possible landscape modes. If a face was detected in any of the

3 analyses, a "1" would be recorded on the camera's csv file, otherwise a "0" would begin recorded, along with the timestamp at which the photo was taken. The photo analysis would be run locally on the device and the photo would be discarded after being analysed, so as to reduce the amount of data that would have to be uploaded, and to respond to privacy concerns of the volunteers collecting the data.

This method does have some limitations, in particular it would misclassify data if the location in which the volunteer was dark, or if the device was being used in such a way that the face of the user wasn't visible to the front-facing camera. The volunteers were asked to avoid using the data collection app in dark places to reduce this problem. The following is the data format used to store the camera data, where [0/1] represents "face not present" or "face present":

timestamp	[0/1]

3.2.2 Gyroscope

For devices that have a Gyroscope, this data was also collected. A sensor listener was created to collect the sensor's readings, with a period of 50ms (20Hz). The values saved on the gyroscope's csv file were the rate of rotation around the x, y, z axis in rad/s, and the timestamp at which the data was collected. If the device didn't have a gyroscope, the csv file would not be generated. The following is the data format used to store the gyroscope data:

timestamp	x	y	z

3.2.3 Light Sensor

The light sensor was collected in a similar way to the gyroscope, with a period of 50ms (20Hz). The frequency at which the sensor is asked to poll at is only considered a hint to the system, so the data is not guaranteed to be exactly collected at 20Hz, and will have to be managed later. The light sensor's csv file contained the measured light in lux, and the timestamp at which the data was collected. The following is the format used to store the light sensor data:

timestamp	*light*

3.2.4 Linear Accelerometer

This sensor is only available if the device also contains a gyroscope. As with the other sensors, it would poll at 20Hz. It gives information about the acceleration force along a x, y, z axis, excluding the force of gravity in m/s^2. If the device had a gyroscope, it would store in the linear accelerometer's csv the x, y, z data along with the timestamp. The following is the format used to store the linear accelerometer data:

timestamp	x	y	z

3.2.5 Uncalibrated Accelerometer

This sensor was available on all devices used to collect data. It measures the acceleration force along the x, y, z axis, but unlike the Linear Accelerometer, it also includes the force of gravity. This helps to understand the position of the device in space (face-down, portrait, landscape, face-up, ...). Measurements were taken at 20Hz, the data was stored in the same way as the previous sensors, x, y, z axis measurements along with a timestamp. The uncalibrated accelerometer data was stored in the following format:

timestamp	x	y	z

3.2.6 Microphone

The microphone would collect raw audio data. The service taking care of the camera would also start and stop the microphone recording. Before taking a photo, 2 seconds of audio would be recorded in Pulse-code modulation (PCM), then the photo would be taken and analysed. The recording was mono, with a sampling rate of 8000Hz, and a 16 bit depth. Due to being a PCM file, the audio could be saved as a list of integer numbers (1 per sample), along with the corresponding timestamp on the microphone's csv. The csv containing the microphone data used the following format:

timestamp	*micsample*

3.2.7 Proximity sensor

Unlike previous sensors, this sensor doesn't have a fixed sample rate. Every time that the state of this sensor changes, the app was informed and collected the data coming from the sensor. Most devices return the distance measured from the sensor in *cm*, but others might just return 2 different values representing "near" and "far". Each time that the sensor gave an updated reading, that value was saved in the proximity sensor's csv, along with the timestamp, with the format shown below:

timestamp	*proximity*

3.2.8 Touches on the screen

To check if the user was touching the screen or not, the app would draw a transparent overlay of $1px * 1px$ in the top corner of the device. A listener was added to overlay that would start a method every time that the user touched a part of the screen that was outside of the $1px * 1px$ overlay. Each time that the method was called, the timestamp would be saved in the corresponding csv file using the following format:

timestamp

3.3 Data collection app

The data collection app was developed in Java using Android Studio from Google [24]. The app has 1 textbox with an explanation in English or Italian about what data is being collected, so that the users don't participate in data collection if they feel uncomfortable with providing the information necessary for this thesis. Two buttons are available: the first to start and stop the data collection, the other to upload the data to an online server.

The first time that the "Start" button is pressed, the user is prompted to give the app access to the previously listed sensors, to allow the app to draw over other apps (used for the camera and for collecting touch information), and to remove battery optimization for the app if the device uses Android 6.0 Marshmallow to avoid the app being shut down while the device is in standby. Once all the permissions are granted, the app starts two Services: one for the camera and microphone, the other for touch and all other sensors. Services are used because even by giving the app a permanent wakelock, some sensors would shut down after the device was in standby for more than a couple of minutes. This resulted in some rows of the initial data being incomplete, as the camera, microphone and touch sensor would still collect data due to having their own services, but all other sensors shutting down after a while. No data while the phone was in use was lost, as all sensors would run while the screen was on. As listed before, the camera takes a photo every 10s, the microphone records 2 seconds of audio every 10 seconds, the touch and proximity sensors record every event that happens, while all the rest of the

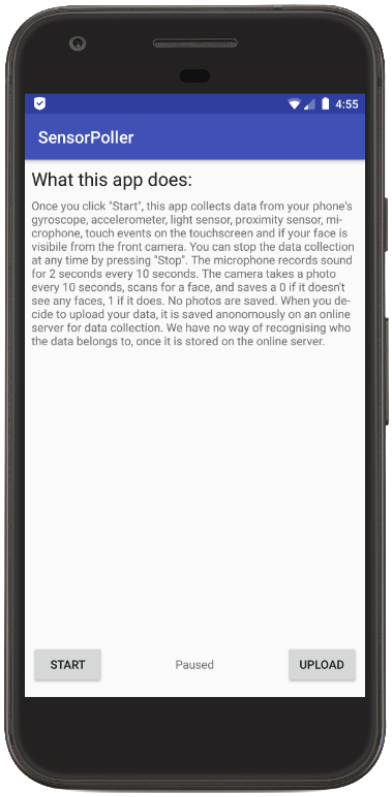

Figure 3.1. Data collection app

sensors run at 20Hz. Each sensor records their data on separate csv files, and allow for the recording to be paused and restarted at any moment.

Once the user has decided that they want to upload the data, they can press the "Upload" button. A notification will appear informing the user of the upload status, and the csv files will be uploaded to a Firebase Storage [25] server online. Once a file is fully uploaded, it is deleted from the user's smartphone to avoid filling up the device's memory. The set of files are assigned a random name composed of a UUID and the sensor's name. This gives the user some privacy, as the only identifiable information is the microphone data.

3.4 Data cleaning and preprocessing

For the training of the convolutional neural network, only a few of the available data sources were selected:

- Camera (to allocate the data from the other sensors into the different classes)
- Uncalibrated accelerometer
- Light sensor
- Proximity sensor
- Touch sensor

The gyroscope and the linear accelerometer data was ignored due to both those measurements not being available on all devices. The microphone was discarded after reading about the experiments by U. Christoph et al [6], discovering that the microphone reduced the accuracy of their context detection algorithm. Furthermore this solves privacy related problems by removing data that can be personally identifiable, and simplifies the neural network by not introducing a sensor that produces magnitudes of data more than the other sensors. Due to the uncertainty of the sensors' frequency and difference of sensor outputs amongst different devices, all the data needed to be cleaned and integrated together to be used with the designed convolutional neural network. A program would run through all the camera csv files one by one. For each UUID, all the corresponding csv files for the studied sensors were opened. 2 seconds was selected as the space of time in which to use data for the classification, taking into account that the longer the window for data collection was, the more likely that the user switches between the two considered states (using the device and not using the device) during the window. The window was divided into 20 slots so the final frequency of the data for classification was 10Hz, making it more likely that each slot in the window contained at least 1 value. For the uncalibrated accelerometer, 1 value of x, y, z was taken for each time slot, and then the values were appended to the camera value in a final csv. If a window didn't contain at least 20 values x, y, z, the whole row of data was discarded. The light sensor was treated in the same way as the accelerometer, but if data was missing from a slot, it was filled in with a copy of the previous light measurement. The data for the proximity sensor was considered as "near" ("0"), or "far" (1). Any values that were different from 0 were considered as "far" ("1"). The last known value was used to fill in all following time slots until a new value was found. This data was also then appended to the final csv. The touch sensor data would save a "1" every time that a touch was registered in a slot, otherwise a "0" would be saved. As before, the data was appended to the final csv.

3 different variations of the final csv were created;

- Raw data with x, y, z values of the accelerometer grouped together according to each sample;
  $C, L_0, ..., L_{19}, P_0, ..., P_{19}, T_0, ..., T_{19}, x_0, y_0, z_0, ..., x_{19}, y_{19}, z_{19}$
- Raw data with x, y, z values of the accelerometer separated;
  $C, L_0, ..., L_{19}, P_0, ..., P_{19}, T_0, ..., T_{19}, x_0, ..., x_{19}, y_0, ..., y_{19}, z_0, ..., z_{19}$
- Normalized data with x, y, z values of the accelerometer separated;
  $C, L_0, ..., L_{19}, P_0, ..., P_{19}, T_0, ..., T_{19}, x_0, ..., x_{19}, y_0, ..., y_{19}, z_0, ..., z_{19}$

Where C is the camera value, L_x are the light values, P_x are the proximity values, T_x are the touch values, x_x, y_x, z_x are the uncalibrated accelerometer values.

Normalization of the data was done for only a few of the sensors, as Touch and Proximity only contain the values 0 or 1, and the Camera is the class label. For the other sensors, Equation 3.1 was used separately for the light sensor, x, y, z values of the uncalibrated accelerometer.

$$x_{norm} = \frac{x - \overline{x}}{\sigma(x)} \tag{3.1}$$

Where x_{norm} is the normalized value of x, $\overline{x}$ is the average of x, $\sigma(x)$ is the variance of x.

To make testing easier, a second version of the data cleaning program would save each row of data in one of ten different csv files randomly. This allowed for the neural network to be evaluated using a 10-fold cross-validation.

After having cleaned and formatted the data, the quantity of samples was as follows:

- **Total:** 12524
- **In use (1):** 2051
- **Not in use (0):** 10473

The classes are unbalanced, due to how the smartphones collecting data were used. Most of the time, the data collection application was running while the users were at work, and didn't have the possibility of using the device. This does however represent real-life usage, where for the majority of the time, the device will be idle (while the user is at work, sleeping, or doing any other activity that doesn't require a smartphone).

3.5 Model Selection and Training

3.5.1 Tools

Two main tools were used to create the neural networks evaluated in this thesis:

- Python
- PyTorch

Python is a high-level programming language first released in 1991. It offers automatic memory management, easy code readability, supports multiple programming paradigms and has a large selection of additional libraries [26]. This makes it an ideal programming language for research, in particular related to machine learning.

PyTorch is an open source Python library based on Torch, mainly developed by Facebook. It provides 2 high-level features [27]:

- Tensor computation with strong GPU acceleration
- Deep neural networks built on a tape-based autograd system, also known as reverse-mode automatic differentiation. This allows PyTorch to easily and quickly compute the derivatives necessary for back-propagation.

It uses a *dynamic graph* (unlike TensorFlow, which uses a *static graph*), allowing for the network to be defined and then modified at runtime, avoiding the need to start from scratch each time the network changes. PyTorch is built to be deeply integrated into Python, allowing for all Python debuggers to inspect the execution of the network step-by-step, or for new network layers to be written in Python directly using libraries such as Cython or Numba.

3.5.2 Evaluated Neural Networks

To start off the neural networks, a custom dataset class was created to transform the csv data into data that a neural network created with PyTorch can interpret. Given the good ability of neural networks to classify images, and according to the results of previous research [1] the dataset was transformed into a virtual 20x6 image before being inserted into the network, where each "pixel" value was the value of the sensor at a given instance. In the case where all sensors were used, each row represented a sensor (or the x, y, z values read

by the uncalibrated accelerometer), and each column represented one of 20 timeslots (in the case that the uncalibrated acceleration values were organized with separated accelerometer values). This gave the network a $6 * 20$ pixel image to analyse, as shown in Table 3.1.

L_0	L_1	...	L_{19}
P_0	P_1	...	P_{19}
T_0	T_1	...	T_{19}
$accX_0$	$accX_1$	...	$accX_{19}$
$accY_0$	$accY_1$	...	$accY_{19}$
$accZ_0$	$accZ_1$	...	$accZ_{19}$

Table 3.1. Data input format

In the case of the removal of a sensor as in section 4.1.13, the row or rows representing the sensor would be removed from the "image", giving a $5 * 20$ or $3 * 20$ pixel image for the network.

Two main types of networks were evaluated;

- Fully Connected Neural Network, as shown in Figure 3.2
- Convolutional Neural Network, as shown in Figure 3.3

Both of their main structures mostly remained unchanged during the evaluation of their various performances, while parameters and data formatting changed.

Parameters that were modified were:

Number of Epochs

It represents the number of full training cycles done. All data will run through the network and backpropagate the loss exactly once per epoch. This number has to be high enough to have a convergence, but not so high that we have overfitting. Values between 5 epochs and 400 epochs were tested, as the paper by Radu et al [1] suggests that going beyond 400 epochs reduced the classifier's performance.

Batch size

To avoid overloading the memory of the training device, we split the training data into batches. This means that less memory is necessary to train the

network, and typically the network will train faster with smaller batch sizes. However, the smaller the batch size, the less accurate the estimate of the gradient will be, causing it to fluctuate much more in comparison to a higher batch size value.

Learning rate

It defines how much the weights in the network are adjusted with respect to the loss gradient. To avoid the weights changing too quickly with new information, and failing to converge or even diverge, we use the learning rate to slow down the changes, allowing the network to converge. A too high value for the learning rate could cause the network to not converge or diverge, as stated before, but a too low leaning rate could cause the network to converge much too slowly, wasting training time.

Loss function

The loss function calculates the "error" between the network's output and the expected output. This allows for the network to easily understand how "wrong" one solution is compared to another, and decide which of the two is better. It is used during the training phase of the network, and is vital for the back-propagation gradient descent. Choosing the loss function that best represents the classes that need to be modelled is vital for a fast and efficient training of the network.

Activation function

The activation function is used in between each convolution and each pooling at each layer for convolutional neural networks. In the case of fully connected neural networks, it is present at each layer. Different activation functions can be used for the CNN and the FCNN. This function defines how each neuron acts for each activation. This function has to be chosen depending on the type of classes you want so as to avoid the problem of "vanishing gradients", which leads to poor learning.

Class weight

Due to the unbalanced nature of the classes, a weight needs to be assigned to avoid the network trying to get the highest accuracy possible at the expense

of one of the two classes (e.g. If one of the classes is 99% of the total set of samples, the network could predict that class no matter what the input is, achieving a 99% overall accuracy). This has to be chosen analysing the number of samples available, and choosing the best weight for the training set.

Fully Connected Layers shape (For CNN only)

One or more fully connected layers are attached at the output of the convolutional neural network, collecting all of the output data, flattening it, and then feeding it through the network to get the desired outputs. The shape of the network will modify the speed at which the network is trained and how efficiently it can reach a solution.

Weight decay

The weight decay is a L2 penalty used to regularise the weights used in the neural network to try and avoid overfitting. It avoids that the weights become too large, by penalising the largest weights during the gradient computation. The weight decay value should be chosen so that it is weaker the more training examples we have, but larger the more parameters we have.

Data formatting

Two different formats of input data were evaluated, one as shown in table 3.1, and one where instead of placing $accX, accY, accZ$ on different rows, they were placed in order $accX_0, accY_0, accZ_0, ...,$
$accX_{19}, accY_{19}, accZ_{19}$, wrapping onto new rows once the current row was filled. This affects the performance of the convolutional neural network, as convolutional and pooling layers will have different sets of data to work with depending on how the input data is formatted.

Data normalization

Data normalization is often suggested for neural network training. By normalizing all inputs to the same scale, we can avoid that one type of sensor have a greater weight than others (e.g. The proximity sensor contains values 0 or 1, while the light sensor can contain values that go from 0 to >100). However, this is not always guaranteed to increase performance, and we need

to consider how the system is going to be implemented in the future. Not normalizing data would simplify the calculations that the device would have to do, reducing the power consumption of this solution. Only data from the accelerometer and the light sensor were normalized using the Formula 3.2, as the values from the proximity sensor and touches are only 0 or 1, thus normalization of those sensors was useless. Each set of values x, y, z from the accelerometer and light values from the light sensor were normalized separately.

$$x_{norm} = \frac{x - \overline{x}}{\sigma(x)} \tag{3.2}$$

Used sensors

Multiple different sensors were used as inputs, but as demonstrated in the experiments of Christoph et at [6], some of the sensors might not be useful for the classification that we are trying to do, as they might not have discernible patterns in each of the different classes. By removing one sensor at a time, we can check how useful each sensor is, if any are introducing noise, and decide on which could be excluded in future works.

Batch Normalization

This normalization is applied to the batches of training data that run through the network. Much like the data normalization, they increase the stability of the neural network by normalizing each batch amongst itself. This does change the performance of the network, as it changes the "strength" of sensors with high values in favour of sensors with lower output values. Differently from input normalization, however, batch normalization is also implemented between internal layers of the network, and not only on the input.

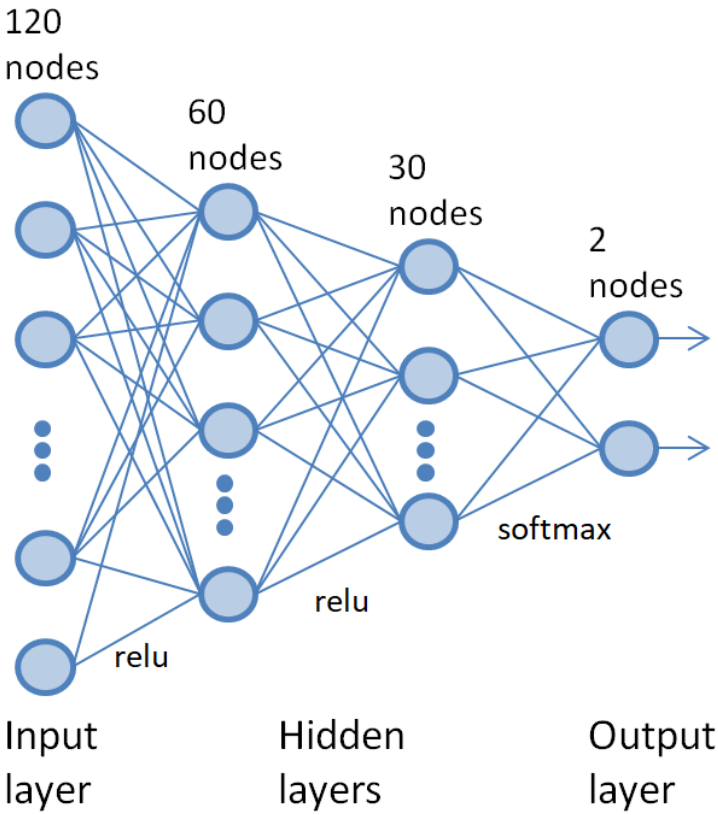

Figure 3.2. Shape of the studied Fully Connected Neural Network

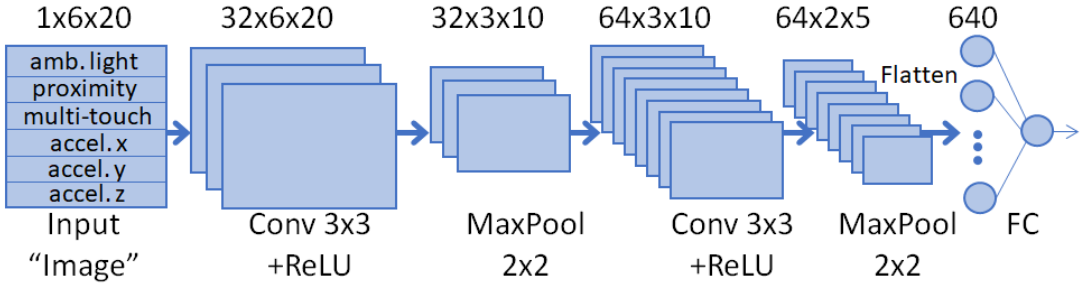

Figure 3.3. Shape of the studied Convolutional Neural Network

3.5.3 Model Evaluation

Due to the unbalanced nature of the two studied classes ("in use" and "not in use"), we cannot evaluate the network performance simply by its accuracy, as explained above. Instead, there are multiple different measurements to take into account to measure the performance of the network. To do this, we collect the results of running the validation set through the network, and divide it into four separate parts;

- T_p, or True positive. It represents the amount of "in use" that were correctly recognised.
- T_n, or True negative. It represents the amount of "not in use" that were correctly recognised.
- F_p, or False positive. It represents the amount of "not in use" that were incorrectly recognised as "in use"
- F_n, or False negative. It represents the amount of "in use" that were incorrectly recognised as "not in use"

Accuracy (Equation 3.3) defines the ratio of overall correctly classified inputs. This of course doesn't give all the necessary information, because if the classes are imbalanced, the accuracy will be taking the most frequent class much more into account than the least frequent one.

$$accuracy = \frac{T_p + T_n}{T_p + T_n + F_p + F_n} \tag{3.3}$$

Precision (Equation 3.4) or Sensitivity, defines the ratio of correctly classified "in use" out of all outputted "in use". This defines how likely it is that, if the network guesses that the device is in use, that the device is actually in use. The smaller the class we are analysing is, the harder it is to have a high value of precision.

$$precision = \frac{T_p}{T_p + F_p} \tag{3.4}$$

Specificity (Equation 3.5) is the same as precision or sensitivity, but instead of considering "positive" (in our case "in use"), the "negative" ("not in use") is considered. This is useful for our application, as we need to understand how good the network is at recognising that the device is not in use anymore.

$$specificity = \frac{T_n}{T_n + F_n} \tag{3.5}$$

Recall (Equation 3.6) defines the ratio of correctly classified "in use" out of all inputted "in use". This defines how often the network correctly classifies the class.

$$recall = \frac{T_p}{T_p + F_n} \tag{3.6}$$

F-score (Equation 3.7) is the harmonic average of the precision and recall, where the perfect score is 1, and the worst score is 0. This allows for us to verify the performance of the network despite there being two unbalanced classes. We can also use β, which is a weight that controls the importance of recall and precision. For our experiments, β was set to 1 (same importance to precision and recall).

$$F - score = \frac{(1 + \beta^2) \cdot recall \cdot precision}{(\beta^2 \cdot recall) + precision} \tag{3.7}$$

Due to the small amount of samples available, to verify each architecture's performance, a 10-fold Cross Validation was used. The data was randomly split into 10 different folds, 9 of which were used to train the network, while the last one was used for the network evaluation. This was repeated ten times, changing the testing and training set at each iteration, so as to guarantee that all ten folds were used for evaluation. The average of the different performances was then used to compare the different architectures, and the standard deviation was calculated to make sure that the performance didn't change drastically from one iteration to the next.

Chapter 4

Experimental Results

4.1 Results

We want a neural network that, given an input from the accelerometer, light sensor, proximity sensor and touch events, can quickly and accurately infer if the user is using their device or not. To choose the best overall network, we evaluate the average F-score over ten folds, and pick the network that has the highest F-score while maintaining the lowest possible complexity.

In this chapter, we list the results of the various tests we ran to find the best possible parameters for our neural network. Each subsection begins with a list of all the network parameters used for the corresponding experiment.

4.1.1 Fully Connected Neural Network

In this section we evaluate the performance of a Fully Connected Neural Network as a classifier for our problem. The shape of the network is the one detailed in Figure 3.2. The parameters were as follows:

- Layer 1: 120 neurons
  Layer 2: 60 neurons
  Layer 3: 30 neurons
  Layer 4: 2 neurons
- Epochs: 50
- Learning Rate: 0.01
- Activation Function: ReLU

- Final activation function: Sigmoid
- Loss function: Negative log likelihood loss

The final result of this network was:

	Predicted 0	**Predicted 1**	**Total**
Actual 0	1020	0	1020
Actual 1	218	0	218
Total	1238	0	

Table 4.1. Confusion Matrix for the FCNN.

Accuracy [%]	**Sensitivity** [%]	**Specificity** [%]	**F-Score** [%]
82.39	0.00	100.00	-

Table 4.2. FCNN scores. F-score could not be calculated.

Where Sensitivity is the percentage of "in use" (1) samples correctly identified as such, Specificity is the percentage of "not in use" (0) samples correctly identified as such.

The network evidently doesn't manage to distinguish between the two classes, and ends up training itself to recognise all inputs as "0", or "not in use". This allows for the network to have a good accuracy and specificity, but terrible sensitivity. This solution was discarded due to it's poor performance, and a CNN solution was evaluated from here on.

4.1.2 Learning Rate

We evaluated different learning rates for the CNN to try and find one that gave the best overall performance. Two tests were conducted, the first to try and find the best rough learning rate value, the second to compare that value with the learning rate value used in [1].

- Epochs: 400
- Batch size: 100
- CNN activation function: ReLU
- Loss function: BCELoss

- Data format: Separate rows
- Raw data
- Class weight: 0.198
- Weight decay: 5e-7
- FC layer activation function: Sigmoid
- All sensors used

Learning Rate	Average F-score	Standard Deviation
0.1	-	-
0.01	0.778675864	0.041582924
1e-3	0.779184181	0.031184446
1e-4	0.747069887	0.03328041
1e-5	0.69895098	0.023688185
1e-6	-	-
1e-7	-	-

Table 4.3. Average F-score and standard deviation for different learning rates over 10 folds.

Learning rate value of 0.1 is too large, the network doesn't manage to converge to any good result as shown in figure 4.1. Instead the learning rate values of 1e-6 and 1e-7 are too small, so the network doesn't manage to converge in good time or converge at all, as shown in figure 4.2. A follow up test was run to compare the learning rate of 1e-3 with 3e-3, used by Radu et at [1]. This test used the same settings as before, but only 200 epochs instead of 400, so the values of 1e-3 are slightly different. The final chosen learning rate was 3e-3, due to it's superior performance compared to all other learning rates.

Learning Rate	Average F-score	Standard Deviation
1e-3	0.777695793	0.031062781
3e-3	0.785595937	0.028426898

Table 4.4. Average F-score and standard deviation for different learning rates over 10 folds.

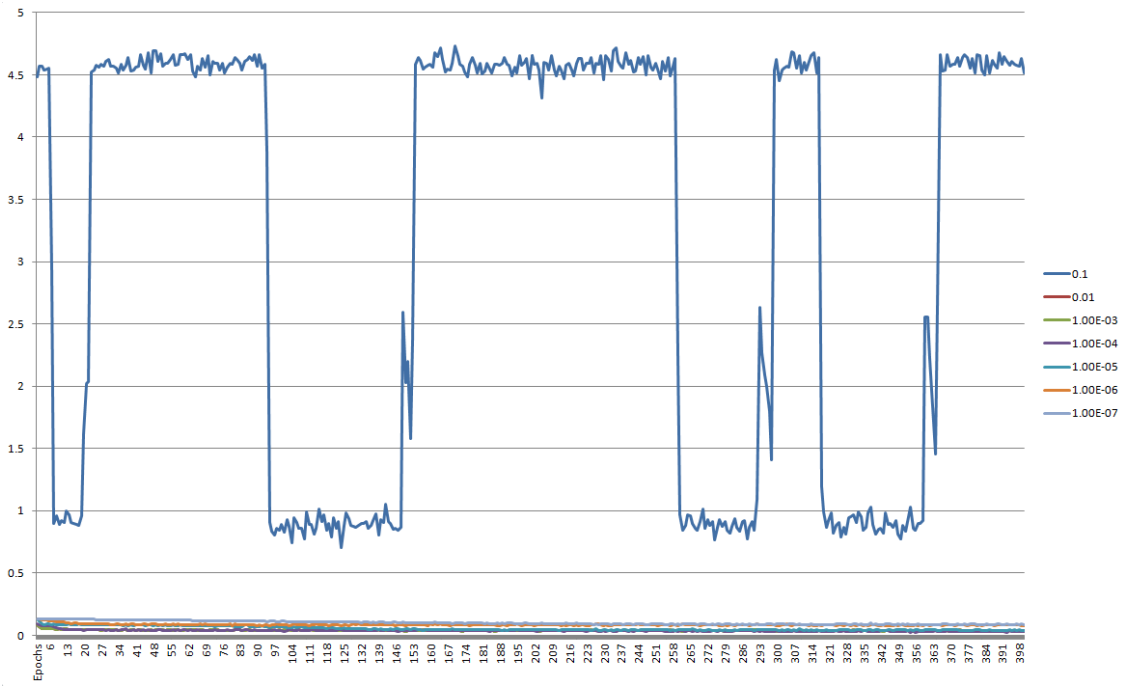

Figure 4.1. Loss over 400 epochs, including learning rate = 0.1

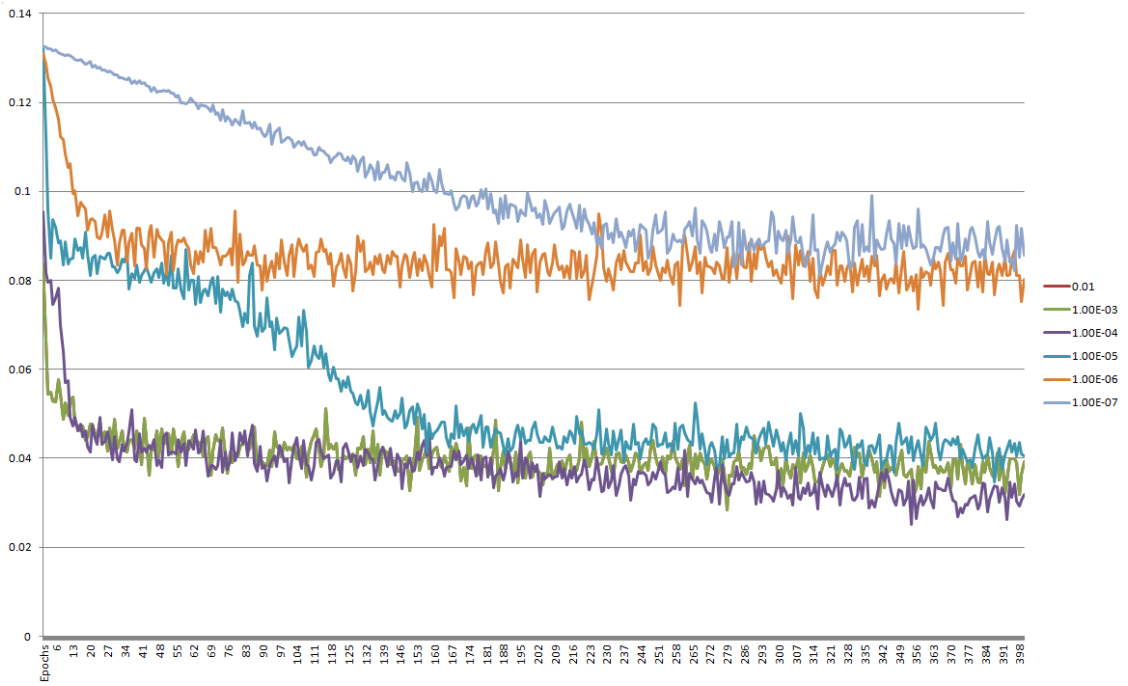

Figure 4.2. Loss over 400 epochs, excluding learning rate = 0.1

4.1.3 Epochs

After finding the optimal learning rate, we evaluated different numbers of training epoch, to find the setting that trained the network well while avoiding overfitting. We avoided going beyond 400 epochs, as Radu et al. discovered that there was no performance increase beyond that value [1]. For this set of tests, the fixed values and settings were:

- Learning rate: 3e-3
- Batch size: 100
- CNN activation function: ReLU
- Loss function: BCELoss
- Data format: Separate rows
- Raw data
- Class weight: 0.198
- Weight decay: 5e-7
- FC layer activation function: Sigmoid
- All sensors used

As seen in the Figure 4.2, the loss converges quickly within the first 50 epochs, then slows down. During the epochs between 50 and 200, the network learns the finer peculiarities of the datasets, improving it's performance. As shown in Table 4.1.3, 5 and 10 epochs are clearly too few for the network to be trained properly, so the network suffers from underfitting. The performance steadily improves as the number of epochs increase until 200, then slightly worsens from 200 to 400. This is probably due to the first signs of overfitting, where the network extracts some variations that are present in the training set but not in the testing set. In fact the loss continues decreasing as the epochs increase, but that is due to the loss being calculated over the training set. The network becomes overfitted for the training set, and starts misclassifying some of the inputs in the testing set.

Epochs	Average F-score	Standard Deviation
5	0.662011021	0.077618367
10	0.682206494	0.032808018
25	0.711913286	0.026850068
50	0.752412219	0.033408197
100	0.769300854	0.022718051
200	0.785595937	0.028426898
400	0.779184181	0.031184446

Table 4.5. Average F-score and standard deviation for different number of epochs over 10 folds.

4.1.4 Batch size

Different batch sizes were tested, to see if changing the batch size gave any significant change in performance. For this set of tests, the fixed values and settings were:

- Learning rate: 3e-3
- Epochs: 200
- CNN activation function: ReLU
- Loss function: BCELoss
- Data format: Separate rows
- Raw data
- Class weight: 0.198
- Weight decay: 5e-7
- FC layer activation function: Sigmoid
- All sensors used

Most batch sizes don't seem to have produced big differences between one and the other, apart from a slight increase with batch size = 100. This value allows for the network to accurately estimate the gradient while training while simultaneously using less memory to train the network. Given that in

average we have 11271 samples, we will update the weights in the network a total of 113 times for each epoch. This gives a good balance between the amount of samples in the batch (given that the gradient is estimated over all the samples), and amount of batches. This gives us a well performing network that is ideal for our objective in this thesis.

Batch size	Average F-score	Standard Deviation
10	0.774100787	0.037742
25	0.777847537	0.02741
50	0.776468874	0.032258
100	0.785595937	0.028427
200	0.77497445	0.020402
400	0.773387817	0.032811
800	0.760939027	0.029705

Table 4.6. Average F-score and standard deviation for different batch sizes over 10 folds.

4.1.5 Batch Normalization layer

A network using the batch normalization layer for training was compared to one without any batch normalization layer, to improve the performance of the CNN.

- Learning rate: 3e-3
- Epochs: 200
- Batch size: 100
- CNN activation function: ReLU
- Loss function: BCELoss
- Data format: Separate rows
- Raw data
- Class weight: 0.198
- Weight decay: 5e-7

- FC layer activation function: Sigmoid
- All sensors used

The performance degrades significantly while using the BatchNorm layers after the ReLU activation function in the two convolutional layers. It normalizes the outputs of the ReLU functions, and changes how the outputs are considered by the rest of the network. Evidently, it is changing the "strength" of the sensors, by putting more emphasis on sensors with less important information for the network, while lowering the emphasis on the more important sensors. Viewing the results of section 4.1.13, we can assume that the accelerometer is losing "strength" in favour of the other three sensors. Because of this, the BatchNorm layer was discarded. This also allows us to have a simpler network, speeding up execution time and saving some more energy compared to a more complex style of network.

BatchNorm layer	Average F-score	Standard Deviation
No	0.785595937	0.028427
Yes	0.732471198	0.04820304

Table 4.7. Average F-score and standard deviation for a CNN with and a CNN without a BatchNorm layer over 10 folds.

4.1.6 Data formatting

Two different data formats were compared as inputs, allowing us to identify the optimal input format. Two different formats were tested, one that followed how the data had been originally collected, and one that split the data into separate rows.

- Learning rate: 3e-3
- Epochs: 200
- Batch size: 100
- CNN activation function: ReLU
- Loss function: BCELoss
- Data format: Separate rows

- Raw data
- Class weight: 0.198
- Weight decay: 5e-7
- FC layer activation function: Sigmoid
- All sensors used

Two different shapes of the input data image were tested; One with the accelerometer values on different rows as in Table 4.8, the other with the three x, y, z accelerometer values kept together as in Table 4.9, with the row overflowing into the lower rows.

L_0	L_1	...	L_{19}
P_0	P_1	...	P_{19}
T_0	T_1	...	T_{19}
$accX_0$	$accX_1$	...	$accX_{19}$
$accY_0$	$accY_1$	...	$accY_{19}$
$accZ_0$	$accZ_1$	...	$accZ_{19}$

Table 4.8. Data input format with x, y, z on different rows

L_0	L_1	L_2	L_3	...	L_{19}
P_0	P_1	P_2	P_3	...	P_{19}
T_0	T_1	T_2	T_3	...	T_{19}
$accX_0$	$accY_0$	$accZ_0$	$accX_1$	...	$accY_6$
$accZ_6$	$accX_7$	$accY_7$	$accZ_7$	...	$accX_{13}$
$accY_{13}$	$accZ_{13}$	$accX_{14}$	$accY_{14}$	...	$accZ_{19}$

Table 4.9. Data input format with x, y, z on the same row

As shown on table 4.10, the CNN performs much better when the x, y, z values of the accelerometer are separated into different rows. This allows for each column to represent one sample instance, and for the convolutions and pooling to create connections between the data from each sample. Having all x, y, z accelerometer values next to each other for each time slot is particularly useful, and isn't guaranteed with the format that has x, y, z on the same row due to the row being 20 spaces long, and not a multiple of 3. This problem can be clearly seen in the table 4.9.

Data Format	Average F-score	Standard Deviation
Alternate	0.760673	0.024805
Separate Rows	0.785595937	0.028427

Table 4.10. Average F-score and corresponding standard deviation for each test

4.1.7 Normalized data

We tested serving inputs as raw data or as normalized data, to see if normalizing the data before training and testing was necessary or not to have good performance.

- Learning rate: 3e-3
- Epochs: 200
- Batch size: 100
- CNN activation function: ReLU
- Loss function: BCELoss
- Data format: Separate rows
- Class weight: 0.198
- Weight decay: 5e-7
- FC layer activation function: Sigmoid
- All sensors used

Much like with the BatchNorm layer, using data normalization over the whole dataset made the performance much worse compared to using raw data, this could be due to using a wrong type of normalization for our data. However, the good performance of raw data suits our study well, as it would allow the trained network to immediately use the data coming from the device's sensors without any pre-processing needed, saving computation time.

Data Type	Average F-score	Standard Deviation
Normalized	0.633345585	0.030012
Raw	0.785595937	0.028427

Table 4.11. Average F-score and corresponding standard deviation for normalized and raw input data.

4.1.8 Loss Function

We tested two different final layer shapes, each with their own specific loss function to find the one with the greatest overall performance. The chosen loss function depended on the shape of the final Fully Connected Neural Network. If it had one single neuron in the output layer, then BCELoss (Binary Cross Entropy Loss) was used. If instead it had two output neurons (one representing "not in use" and the other "in use"), then CrossEntropyLoss was used.

- Learning rate: 3e-3
- Epochs: 200
- Batch size: 100
- CNN activation function: ReLU
- Data format: Separate rows
- Raw data
- Class weight: 0.198
- Weight decay: 5e-7
- FC layer activation function: Sigmoid
- All sensors used

As the results show, BCELoss gives the best performance. This is mainly due to the dataset having just two classes, which the BCELoss was designed for. CrossEntropyLoss instead would perform better if there were more than 2 classes, such as in the MNIST dataset (10 classes), where we are obliged to use multiple output neurons.

Loss function	Average F-score	Standard Deviation
BCELoss	0.785595937	0.028427
CrossEntropyLoss	0.736163352	0.028107956

Table 4.12. Average F-score and corresponding standard deviation for the different loss functions.

4.1.9 Class weight

Different class weights were tested to find the optimal values for training. Given that our classes are imbalanced, we considered two weights: 0.5 to avoid assigning one class more importance than the other, and 0.198, which was calculated so as to correctly balance the "not in use" class with the "in use" class.

- Learning rate: 3e-3
- Epochs: 200
- Batch size: 100
- CNN activation function: ReLU
- Loss function: BCELoss
- Data format: Separate rows
- Raw data
- Weight decay: 5e-7
- FC layer activation function: Sigmoid
- All sensors used

A class weight was assigned to give more importance to the "in use" class over the "not in use" class, as the classes were imbalanced. As expected, assigning a class weight appropriate to the class distribution improved the network performance, as the Sensitivity increased due to a larger weight assigned to the "in use" class. However, giving the two classes equal weights (0.5) didn't make the f-score much lower than in the other case, showing us that the network is still able to recognise the difference between the two classes despite there being no difference in weight.

Class weight	Average F-score	Standard Deviation
0.5	0.77956205	0.02449572
0.198	0.785595937	0.028427

Table 4.13. Average F-score and corresponding standard deviation for each class weight

4.1.10 Weight decay

We tested a CNN with and without a weight decay (L2 penalty), to see if any difference in performance could be detected.

- Learning rate: 3e-3
- Epochs: 200
- Batch size: 100
- CNN activation function: ReLU
- Loss function: BCELoss
- Data format: Separate rows
- Raw data
- Class weight: 0.198
- FC layer activation function: Sigmoid
- All sensors used

The weight decay prevents the weights in the network from growing too large unless it is strictly necessary. This allows for the network to decrease it's complexity while maintaining good performance. The weights in the network are made to decay in proportion to their size, and allows to limit the number of free parameters in the model so as to avoid overfitting. As the results show, there is a slight increase in performance using the weight decay, which brought us to decide to use the weight decay (L2 penalty) as standard for the rest of the tests.

Weight decay	Average F-score	Standard Deviation
Static	0.775759333	0.009511629
Decaying	0.785595937	0.028426898

Table 4.14. Average F-score and corresponding standard deviation for weight decay and no weight decay.

4.1.11 CNN activation function

Two different types of activation functions were tested for the CNN layers, the one with the highest F-score was selected.

- Learning rate: 1e-3
- Epochs: 200
- Batch size: 100
- Loss function: BCELoss
- Data format: Separate rows
- Raw data
- Class weight: 0.198
- Weight decay: 5e-7
- FC layer activation function: Sigmoid
- All sensors used

Both ReLU and SELU were used as activation functions in the two CNN layers. As the results show, ReLU had a significant performance increase over SELU. This could be due to SELU not being ideal with the type of raw data we are feeding the network, and needing a custom weight initialization that has not been implemented. Furthermore, SELU is slightly more complex to compute compared to ReLU or Leaky ReLU, which leads us again to select ReLU as default to reduce the complexity of the computations necessary for the classification.

Activation function	Average F-score	Standard Deviation
ReLU	0.770716966	0.038065456
SELU	0.735663848	0.052935547

Table 4.15. Average F-score and corresponding standard deviation for each activation function.

4.1.12 FC Layer activation function

Much like for the CNN layers, we tested different activation functions for the FC layer of the CNN.

- Learning rate: 3e-3
- Epochs: 200
- Batch size:
- CNN activation function: ReLU
- Loss function: BCELoss
- Data format: Separate rows
- Raw data
- Class weight: 0.198
- Weight decay: 5e-7
- All sensors used

Activation function	Average F-score	Standard Deviation
Sigmoid	0.777695793	0.031062781
ReLU then Sigmoid	0.770716966	0.038065456
Sigmoid then Sigmoid	0.766370727	0.041025276

Table 4.16. Average F-score and corresponding standard deviation for each activation function.

As the results show, there is a negligible difference in performance between a final fully connected network with 640 nodes that directly feed into

1 node using the sigmoid activation function and a final fully connected neural network with 640 nodes that feed into 100 nodes using ReLU or sigmoid, followed by a second sigmoid activation function on the final single node. Because of these small differences, the simplest version of the three was selected (640 to 1, using sigmoid), as it requires less calculations to complete, speeding up execution.

4.1.13 Removing sensors

We tested the network while removing sensors from the input, to verify that the sensors were all giving useful data for our classification and to discover which of the sensors impacted the most on the network performance.

- Learning rate: 3e-3
- Epochs: 200
- Batch size:
- CNN activation function: ReLU
- Loss function: BCELoss
- Data format: Separate rows
- Raw data
- Class weight: 0.198
- Weight decay: 5e-7
- FC layer activation function: Sigmoid

Removed sensor	Average F-score	Standard Deviation
None	0.785595937	0.028426898
Light	0.771717464	0.031471
Proximity	0.772076349	0.031352
Touch	0.779843888	0.026049
Accelerometer	0.42236505	0.059225

Table 4.17. Average F-score and corresponding standard deviation for different input sensor sets.

This test was done taking inspiration from [6], where the microphone input data made the classifier performance worse. As the results clearly show, the accelerometer is the only vital sensor for our classification, meaning that the position and relative movement of the device is necessary to understand if the device is in use or not. The performance does decrease when other sensors are removed, but only by a small amount. This means that we could explore the possibility of removing some of those sensors in the future, possibly reducing the energy required for collecting data and simplifying the CNN compared to our best result. This also confirms that the touch events aren't the only reason for the good performance of the network, so the network is able to identify if the device is in use even if the user is not directly interacting with elements on the screen (for example, when reading an article or watching a video clip).

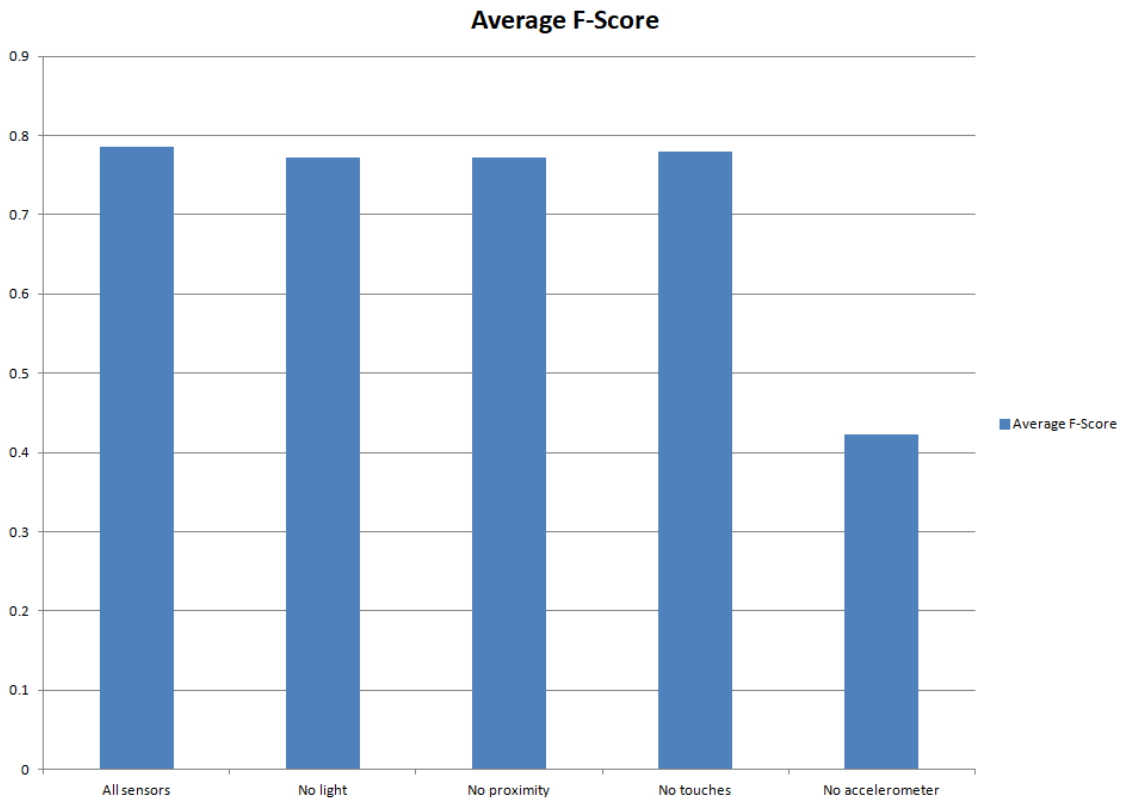

Figure 4.3. Average F-score with removed sensors

4.1.14 Best result

The best result was given by a CNN with these parameters:

- Learning rate: 3e-3
- Epochs: 200
- Batch size: 100
- CNN activation function: ReLU
- Loss function: BCELoss
- Data format: Separate rows
- Raw data
- Class weight: 0.198

- Weight decay: 5e-7
- FC layer activation function: Sigmoid
- All sensors used

Table 4.18. CNN Confusion Matrix. (0 represents "device not in use", while 1 represents "device in use")

	Predicted 0	**Predicted 1**	**Total**
Actual 0	1001.8	45.4	1047.2
Actual 1	42.4	161.8	204.2
Total	1044.2	207.2	

Table 4.19. CNN Scores

Accuracy [%]	**Sensitivity** [%]	**Specificity** [%]	**F-Score** [%]
92.98	79.23	95.66	78.66

As stated before, these tests were run using a 10-fold cross-validation. Because of this, the numbers in the confusion matrix are fractional, as they contain the average value from each of those 10 folds. The samples for "device not in use" have a higher accuracy than "device in use", as shown with the high specificity, while accuracy for the "device in use" samples is lower, but still has a decent accuracy (as shown with the sensitivity). This is probably due to the larger amount of training samples for "device not in use" than for "device in use". This however is not a large problem for this study, as specificity is more important than sensitivity for our energy management goals. We want to detect when the device changes state from "in use" to "not in use", as that is when a large amount of power is consumed unnecessarily. The state change from "not in use" to "in use" is not of particular interest, as it is immediately dictated by the used (by pressing a button or tapping on the screen to wake up the device). However, we want a high sensitivity value to improve the user experience, and avoid switching off device functions while the device is still in use. To help our system to avoid worsening the user experience, the OS could create wakelocks to block the device from switching to a suspended state in certain scenarios where we know without a doubt that the device is being used (such as when a game is opened, or a

video is playing). Once that wakelock is released, our system could restart trying to infer if the device is in use or not.

4.2 Comparison of different standby methods

In this section, we compare three different types of standby methods, highlighting the pros and the cons for each and finally comparing the difference in saved energy amongst the three.

4.2.1 Simple timeout

According to Carroll et al [30], phones in a suspended state (standby) consume 25mW. Phones in idle state (screen on, no apps running) consume 3338mW. These values were evaluated on a Google Nexus One mobile phone. Other devices tested were the HTC Dream and the Openmoko Neo Freerunner (revisionA6). Both of those devices were found to have a similar idle state consumption to the Google Nexus One, but a much higher suspended state consumption (circa 70mW) due to the poor low-power state of the SoC compared to the Nexus device.

Thus, we can state that $P_{off} \approx 25mW$, and $P_{idle} \approx 333mW$. An additional $P_{display} \in [38 \dots 257]mW$ is consumed when the display is on at various brightness levels. For simplicity, we assume that the display is always kept at medium brightness, so we state that $P_{display} \approx 150mW$.

An often used approach is based on two separate timeouts [33];

- $T_{to,1}$ for turning off the display, we assume it is $T_{to,1} = 10s$
- $T_{to,2}$ to transition to a suspended state, we assume it is $T_{to,2} = 30s$

With this information, we can calculate the amount of wasted energy for every time that the device transitions from "in use" to "not in use":

$$E_{waste,to} = T_{to,2} \cdot (P_{idle} - P_{off}) + T_{to,1} \cdot P_{display} = 30 \cdot 0.458 + 10 \cdot 0.150 = 10.74J \tag{4.1}$$

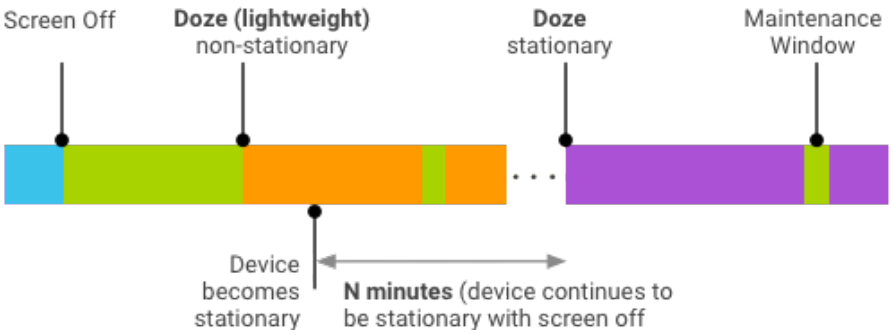

Figure 4.4. Doze lifecycle [33]. We consider the time between the start of the timeline (screen still on) and screen off as $T_{to,1}$, and the time between screen off and the start of Doze stationary as $T_{to,2}$.

4.2.2 Camera detection

Xiang Chen et al [28] have analysed how power-hungry the camera on a smartphone is. They examined 5 different Samsung products:

- Nexus S
- Galaxy S1
- Nexus
- Galaxy S2
- Galaxy S3

The screens of all of these devices are AMOLED, allowing for the display to be turned off by overlaying a pure black image over the running application. For a 4 minute camera recording, the average power consumption is 1400mW with low-quality recording (video resolution was 640x360, 30fps), and 1650mW with high-quality recording (video resolution was 1280x720, 30fps).

Sai Suren Kumar Kasireddy and Vishnuvardhan Reddy Bojja [32] analysed the power consumption of various smartphone applications including the camera on 4 different devices;

- Sony Ericsson C902
- Sony Ericsson Elm
- Sony Ericsson Xperia

- HTC Dream

The camera app was be accessed for a period of 30s, a fixed number of photos were taken and the average energy consumption was measured. The average energy consumption of the devices during the 30s of testing was between circa 35J (for the HTC Dream) and circa 44J (for the Sony Ericsson Elm). Transforming the value into Watts, we can see that the value is similar to the ones found by Xiang Chen et al.

$$\frac{35J}{30s} = 1.467W = 1467mW$$

Knowing this, we can analyse a solution that uses the front-facing camera to understand if the phone is in use or not. Assuming that a photo is taken every 2s to check if the device is being used or not, we can say that in the worst case scenario, the device takes 2s before transitioning from idle to suspended state. We assume that the face detection algorithm is the golden standard, as we used that for classification of our data for the CNN-based solution. Assuming that a face detection algorithm uses negligible amounts of power, and that the camera needs to be on for 1s to take a photo for auto-focus and capture, we can state that $T_{sample} = 2s$, $T_{cam} = 1s$, $P_{cam} \approx 1400mW$.

Thus, we can calculate the energy overhead for the transition from "in use" to "not in use":

$$E_{waste,cam} = T_{sample} \cdot (P_{idle} + P_{oled} - P_{off}) + T_{cam} \cdot P_{cam} = 2 \cdot 0.458 + 1 \cdot 1.400 = 2.316J$$

This corresponds to a 78% reduction in wasted energy when transitioning from "in use" to "not in use" compared to a timeout-based solution, but will have a large impact on battery life while the device is in use, given that photos must be taken an analysed every 2s while the device is in use. Due to this, after using the device for more than 14 seconds, all energy savings are lost.

$$N_{samples} = \frac{14}{T_{sample}} = 7$$

$$E_{waste,facedetection} = P_{cam} \cdot T_{cam} \cdot N_{samples} = 1.400 \cdot 7 = 9.800J$$

$$E_{waste,facedetection} + E_{waste,cam} = 9.800 + 2.316 = 12.116J > E_{waste,to} = 10.74J$$

This solution is also impractical in daily use, as it won't be able to work in environments with low-light, or can easily misclassify a photo if it is blurry or the camera is covered.

Despite wasting less energy during the transition, we need to take into account all the extra photos taken during normal use, which will have a bigger impact on the battery life the longer the device is in use.

4.2.3 CNN-based solution

The proposed CNN-based solution uses 4 sensors as inputs:

- Accelerometer
- Proximity
- Light
- Touch

Inayat et al compared the difference in energy consumption between the Accelerometer, Magnetic Field sensor, Proximity sensor, Light sensor, Orientation sensor and GPS on a QMobile A12 smartphone [29]. They considered 4 different conditions for the device to see if the sensors changed behaviour in different conditions;

- Stationary indoor
- Moving indoor
- Stationary outdoor
- Moving outdoor

As shown in Figure 4.5, the sensors used in the CNN-based solution are the ones that overall consume the least amount of power.

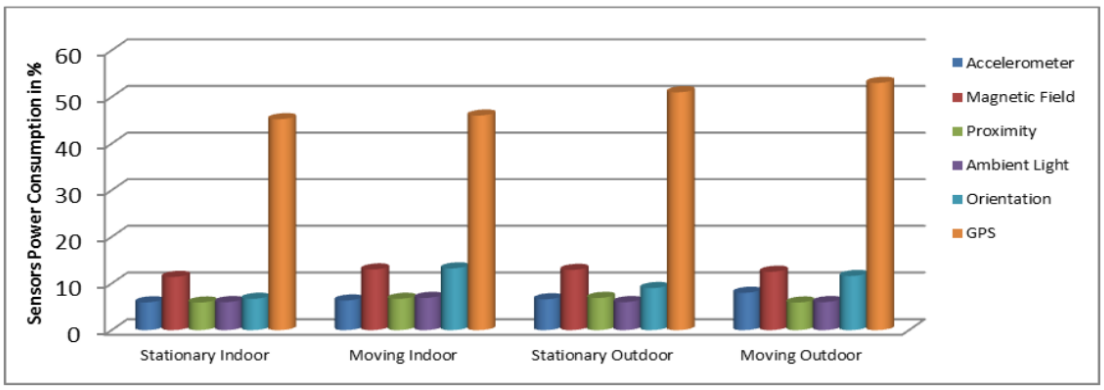

Figure 4.5. Overall energy consumption in different scenarios [29]

König et al measured the energy consumption of multiple smartphone sensors on the Samsung Galaxy S2 and Samsung Galaxy S3 [31]. All measurements were taken 3 times and then averaged out using a voltmeter and an ammeter. An internal software API was also used to verify the measurements. Considering just the proximity and accelerometer sensors active, the average power varied from phone to phone, giving a best-case scenario as 129.76mW on the Samsung Galaxy 2 GT I9100G, and the worst-case scenario as 514.55mW on the Samsung Galaxy 2 GT I9100. These results are similar to the ones measured by Carroll et al [30], in fact all of the sensors used for the CNN are active while the device is in an idle state:

- The accelerometer is active while the device is idle state, given that it is used for the rotation of the screen.
- The proximity sensor is assumed to be active, given that it is so low power, and it is used to avoid waking up devices that have double-tap to wake functionality while kept in a trouser pocket.
- The light sensor is active while the device is idle, as it is used to change the screen brightness.
- The touch sensor has to be active to allow us to interact with the device while the screen is on.

With these assumptions, we can state that no extra power is needed to make the CNN-based solution work, outside of the negligible amount of power used for the CNN analysis itself.

As stated before, the CNN-based solution has an accuracy of $p = 0.96$ when the device is not in use. Assuming that this solution is run with the same frequency as the camera-based solution (T_{sample}), we can also state that every analysis with the CNN will have it's own independent set of data, given that the window used for the classification is 2 seconds. We can calculate the probability of correctly guessing that the device is not in use in k samples using a geometric distribution:

$$P(X = k) = (1 - p)^{(k-1)} \cdot p$$

Knowing this, the average time before a correct classification is calculated as:

$$T_{avg} = T_{sample} \cdot E(X) = \frac{T_{sample}}{p} = 2.08s$$

With this information, and knowing that the energy consumption of the sensors is negligible, we can calculate the overhead of the CNN-based solution:

$$E_{waste,cnn} = T_{avg} \cdot (P_{idle} + P_{oled} - P_{off}) = 2.08 \cdot 0.458 = 0.952J$$

This approach consumes 59% less than the camera-based solution for the transition, and 91.1% less than the timeout-based solution. The sensors do not consume extra amount of energy compared to the regular behaviour of a device, with the only extra overhead being the CNN, which is considered negligible. Figures 4.6 and 4.7 show the variation of the energy gains with

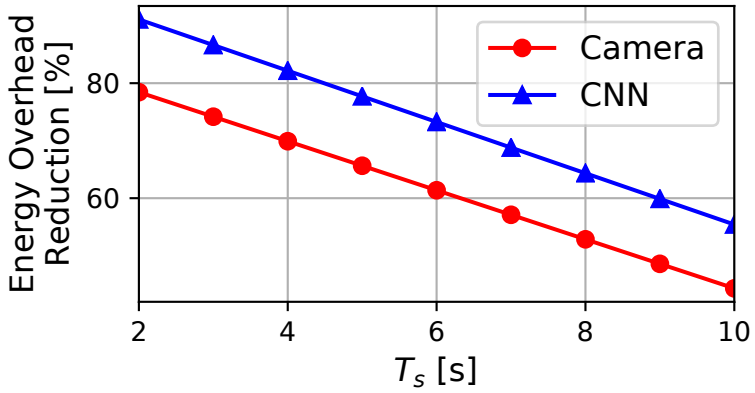

Figure 4.6. Overhead reduction with respect to a timeout-based approach for different values of T_{sample} (shortened to T_s)

respect to the timeout value as a function of two design parameters: the sampling period (T_s) and the specificity of the model (p). The first graph shows the energy overhead reduction our CNN-based model and a camera-based model have compared to the standard timeout-based model. As the graph clearly shows, even if the sampling period is increased to 10s there is a significant energy overhead reduction for both solutions compared to the timeout-based solution. The CNN-based solution has a consistently lower overhead compared to the camera-based solution, providing an even greater benefit.

The second graph still compares the camera-based solution and our CNN-based one to the timeout-based solution, but highlights the effect the specificity (p) has on the benefit the CNN-based solution has over the camera-based one. As the graph shows, even if the specificity decreases by a large

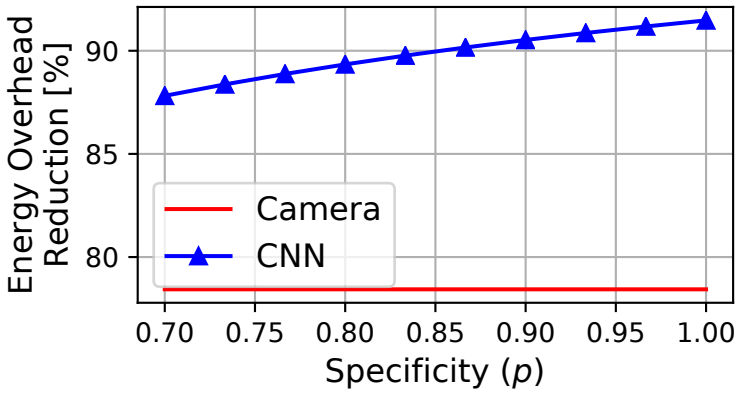

Figure 4.7. Overhead reduction with respect to a timeout-based approach for different values of Specificity (p), with $T_{sample} = 2s$

amount, the CNN-based solution still has a lower overhead than the camera-based one. The camera-based solution is constant, as we use it as our "golden reference", given that it classified the data we then used to train the neural network.

Chapter 5

Conclusions and Future Work

As smartphones have become more powerful, they have also increased the amount of energy consumed while not in standby. This work proposed an alternative method to recognise when a smartphone transitions from "in use" to "not in use" with two seconds of collected data from low-powered sensors, such as the uncalibrated accelerometer, proximity sensor, light sensor and touch events. Knowing when this event happens could allow us to switch off unnecessary functions on the device, and quickly switch to an idle state. By training a convolutional neural network using data collected with a developed android app from sensors that are available on all recent smartphones, we managed to obtain a network with an overall accuracy of 92.98% and F-score of 78.66, which could consume up to 59% less energy on a transition from "in use" to "not in use" compared to a camera-based solution, and up to 91.1% compared to a timeout-based solution, which is in use today. Furthermore, we know that the accuracy of the network is not strictly linked to touch events on the screen, as in the case of a timeout-based system, but is connected to the uncalibrated accelerometer readings. This means that the network is able to recognise if the device is in use or not even when the user is not directly interacting with elements on the screen (such as reading an article or watching a video clip).

In the future we envision implementing this network on a smartphone device through a developed application, which could automatically implement power-saving policies (such as switching off the screen) once a transition from "in use" to "not in use" has been detected and measuring the real-world performance and energy savings on the device itself.

Bibliography

[1] V. Radu, C. Tong, S. Bhattacharya, N.D. Lane, C. Mascolo, M.K. Marina, F. Kawsar, *Multimodal Deep Learning for Activity and Context Recognition*, Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies, Vol. 1, No. 4, Article 157, November 2017

[2] Github repository 2017. *Multimodal Deep Learning Framework*, https://github.com/vradu10/deepfusion.git. (2017).

[3] H. Wu, M. Siegel, S. Ablay, *Sensor Fusion for Context Understanding*, IEEE Instrumentation and Measurement Technology Conference, Anchorage, AK, USA, 21-23 May 2002

[4] Vallina-Rodriguez, Narseo, and Jon Crowcroft. *The case for context-aware resources management in mobile operating systems.* Mobile Context Awareness. Springer, London, 2012. 97-113.

[5] Attal, F., Mohammed, S., Dedabrishvili, M., Chamroukhi, F., Oukhellou, L., & Amirat, Y. *Physical Human Activity Recognition Using Wearable Sensors*, Sensors 15.12 (2015): 31314-31338.

[6] Uta Christoph, Janno von Stülpnagel, Karl-Heinz Krempels, Christoph Terwelp, *Context Detection on Mobile Devices*, Second Workshop on Context-Systems Design, Evaluation and Optimisation (CoSDEO 2011). 2010.

[7] G. Ian, B. Yoshua, C. Aaron, *Deep Learning* MIT Press, 2016, http://www.deeplearningbook.org.

[8] B. Watson, *The Machine Learning Dictionary*, http://www.cse.unsw.edu.au/ billw/mldict.html

[9] H. Tann, S. Hashemi, R. Bahar, and S. Reda, *Runtime configurable deep neural networks for energy-accuracy trade-off*, in Proceedings of the Eleventh IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis. ACM, 2016, p. 34

[10] By Glosser.ca - Own work, Derivative of File:Artificial neural network.svg, CC BY-SA 3.0,

https://commons.wikimedia.org/w/index.php?curid=24913461

[11] By Aphex34 - Own work, CC BY-SA 4.0, https://commons.wikimedia.org/w/index.php?curid=45679374

[12] By Andy - https://adventuresinmachinelearning.com/convolutional-neural-networks-tutorial-in-pytorch

[13] M. Milad, M. Rohit, and S. Richard, *Cs 224d: Deep learning for nlp*, Spring 2015.

[14] A. L. Maas, A. Y. Hannun, A. Y. Ng, *Rectifier Nonlinearities Improve Neural Network Acoustic Models*, Proc. icml. Vol. 30. No. 1. 2013.

[15] H. Tann, S. Hashemi, R. Bahar, S. Reda, *Runtime configurable deep neural networks for energy-accuracy trade-off*, in Proceedings of the Eleventh IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis. ACM, 2016, p. 34

[16] Y. LeCun, Y. Bengio, G. Hinton, *Deep learning*, nature, vol. 521, no.7553, p. 436, 2015

[17] Goodfellow, Ian, Yoshua Bengio, and Aaron Courville. Deep learning. MIT press, 2016.

[18] A. Stisen, H. Blunck, S. Bhattacharya, T. S. Prentow, M. B. Kjærgaard, A. Dey, T. Sonne, M. M. Jensen, *Smart Devices are Different: Assessing and Mitigating Mobile Sensing Heterogeneities for Activity Recognition* In The 13th ACM Conference on Embedded Networked Sensor Systems, 2015

[19] A. L. Goldberger, L. A. N. Amaral, L. Glass, J. M. Hausdorff, P. Ch. Ivanov, R. G. Mark, J. E. Mietus, G. B. Moody, C. K. Peng, H. E. Stanley, *PhysioBank, PhysioToolkit, and PhysioNet: Components of a New Research Resource for Complex Physiologic Signals.*, Circulation 101, 23 (2000), e215–e220. Circulation Electronic Pages: http://circ.ahajournals.org/cgi/content/full/101/23/e215 PMID:1085218; doi: 10.1161/01.CIR.101.23.e215, 2000.

[20] V. Radu, P. Katsikouli, R. Sarkar, M. K. Marina. 2014, *A Semi-supervised Learning Approach for Robust Indoor-outdoor Detection with Smartphones.*, In Proceedings of the 12th ACM Conference on Embedded Network Sensor Systems (SenSys '14). ACM, New York, NY, USA, 280–294. https://doi.org/10.1145/2668332.2668347

[21] N. Vallina-Rodriguez, P. Hui, J. Crowcroft, A. Rice, *Exhausting battery statistics: understanding the energy demands on mobile handsets*, in Proceedings of the second ACM SIGCOMM workshop on Networking, systems, and applications on mobile handhelds, MobiHeld '10, (New York, NY, USA), pp. 9–14, ACM, 2010.

[22] Z. Zhuang, K.-H. Kim, J. P. Singh, *Improving energy efficiency of location sensing on smartphones*, in Proceedings of the 8th international conference on Mobile systems, applications, and services , MobiSys '10, (New York, NY, USA), pp. 315–330, ACM, 2010.

[23] Google LLC, *FaceDetector*, https://developers.google.com-/android/reference/com/google/android/gms/vision/face/FaceDetector

[24] Google LLC, *Android Studio*, https://developer.android.com/studio/

[25] Google LLC, *Firebase*, https://firebase.google.com/

[26] Python Software Foundation, *About Python*, https://www.python.org/about/

[27] PyTorch, *About PyTorch*, https://github.com/pytorch/pytorch#more-about-pytorch

[28] Xiang Chen, Yiran Chen, Zhan Ma, Felix C. A. Fernandes, *How is Energy Consumed in Smartphone Display Applications?*, Proceedings of the 14th Workshop on Mobile Computing Systems and Applications. ACM, 2013.

[29] Inayat Khan, Shah Khusro, Shaukat Ali, Jamil Ahmad, *Sensors are Power Hungry: An Investigation of Smartphone Sensors Impact on Battery Power from Lifelogging Perspective*, Bahria University Journal of Information & Communication Technology 9.2 (2016): 8-19.

[30] Aaron Carroll, Gernot Heiser, *An Analysis of Power Consumption in a Smartphone*, USENIX annual technical conference. Vol. 14. 2010

[31] Immanuel König, Abdul Qudoos Memon, Klaus David, *Energy consumption of the sensors of Smartphones*, The Tenth International Symposium on Wireless Communication Systems 2013

[32] Sai Suren Kumar Kasireddy, Vishnuvardhan Reddy Bojja, *Measurements of Energy Consumption in Mobile Applications with respect to Quality of Experience* (2012).

[33] Google, Platform Power Management, https://source.android.com-/devices/tech/power/platform_mgmt