Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Distributed Computing

SwitP: Mobile Application for Real-Time Swimming Analysis

Semester Thesis

Daniel Wirzberger Raimundo

wirdanie@student.ethz.ch

Distributed Computing Group
Computer Engineering and Networks Laboratory
ETH Zürich

Supervisors:
Darya Melnyk, Simon Tanner
Prof. Dr. Roger Wattenhofer

June 22, 2020

Acknowledgements

First of all, I want to thank Darya Melnyk and Simon Tanner for the insights they gave me during our weekly meetings and their feedback concerning this thesis.

I also want to thank the Aquatic Masters Team (AMT) Zürich for their flexibility, patience, and interest in this project. Special thanks go to their "coach" Beat Schilt, who was always ready to adapt the training to have more relevant data on the watch.

Abstract

Smartwatches in the Pool (SwitP) is an Android Wear application that implements the convolutional neural network (CNN) developed by Brunner et al. [1] to recognize swimming styles. Basing on the measurements from acceleration, rotation, and magnetic field sensors, this neural network can reliably predict what swimming style was swum.

The neural network itself is adapted to this mobile environment by using the Tensorflow Lite framework [2], which integrates both the conversion and prediction functions. Preprocessing functions, optimized for both embedded performance and adaptability to other functions, filter and resample the raw sensor data to make it usable by the CNN.

Using the resampled sensor data as well as the classification results, the application determines the number of lengths and the number of strokes that the swimmer has swum. A user interface gives live feedback and control to the swimmer and is designed to be used in the pool.

Contents

Chapter 1

Introduction

Nowadays, smartwatches allow for the recognition of complex human activity, one of which being swimming in the pool. Traditionally (in commercially available sports watches), this is done using classical processing methods on the signals generated by the different sensors in the watches. This approach provides acceptable results but lacks some robustness and adaptability to the swimming abilities of the watch wearer. For example, if the swimmer has to slow down in a crowded lane, such methods might recognize one lap too much. Or if the swimmer has a weak wall push-off (often encountered in beginners), the watch might count one lap less. Furthermore, swimming style recognition is often limited, and the user often has to specify manually what style he is swimming.

The known flexibility of neural networks, and the possibility to improve their performance for a specific user, neural networks can provide a good solution to the swimming recognition problem. Recent progress in training methods and neural network integrability for low-power embedded devices, make them also a viable solution for real-time mobile applications.

Multiple works (some of them are introduced in Section 1.1) have investigated the recognition performance of such networks, but generally in an offline, high performance, artificial setup.

In this specific project, the main goal is to be able to tell what the actual swimming style is in realtime on a smartwatch available on the public market, as well as count how many laps have been swum, while having a user-friendly interface. The neural network used for this project was designed and trained by Brunner et al. [1] (detailed in Section 2), and will be adapted to this embedded application.

The app designed during this thesis, and the project in general (which mainly consists of the app) will be referenced by SwitP, an abbreviation of *Smartwatches in the Pool*.

1.1 Related Work

SwitP is mainly a continuation of the work presented in [1]. This paper presents a method to acquire swimming data from a wrist-worn smartwatch, and structure it to train a specific neural network, which in turn can predict the swimming style with high precision. Furthermore, this work introduces a method to count laps based on the class predictions, which will also be used in SwitP. While this thesis aims to extend this project to provide real-time (on the smartwatch) classification and post-processing of the data, some modifications were brought to the original approach, they will be detailed in the upcoming chapters.

Wang [3] also based on [1] and aimed to extend the recognition capabilities beyond swimming styles and the pool, in the more general context of Human Activity Recognition (HAR). It also provided some insights as to what channels and sampling rates are relevant, as well as some alternative solutions for data-augmentation and stroke-counting methods. Furthermore, it has shown some limitations of the sensors available on the smartwatch.

Mooney et al. [4] provide an overview of different papers (87 in total) on swimming data acquisition and interpretation. This work shows some of the extensive research carried out in sensor selection and placement and compiles insights for the post-processing methods (lap detection and stroke counting will be discussed in Section 3.4).

Siirtola et al. [5] use discriminant analysis of the first and second-order to do an offline classification of swimming styles. They also show that depending on the sensor placement (on the upper back instead of the wrist), the estimation accuracy of specific swimming parameters can be increased. [5] also deals with lap and stroke counting, which will be discussed in 3.4.

Pansiot et al. [6] also use another sensor placement, on the swimmer's goggles. The analysis is done offline, after uploading the data stored in the accelerometer to another device. The swimming styles are recognized using k-Means clustering basing on the body angle values, computed from the acceleration values.

Further works and their solutions to specific problems will be introduced in the Sections which are most closely related to them.

Chapter 2

Background

The majority of the background is contained in [1], which is a paper detailing the data acquisition methods as well as the general Neural Network structure, training methods, and evaluation processes.

This chapter aims to summarize the main elements on which this project is based.

2.1 General Network Structure

The neural network structure used for this work is detailed in [1, ch.4]. Briefly, the input tensor is a `n_ch`*annels* × `n_sa`*mples* data frame, which is then passed through four convolutional layers accompanied by ELU activations and max-pooling, increasing the feature map depth to 64, progressively reducing the height, but keeping a width of `n_ch`*annels*. Between these layers, 3×1 convolution filters ensure data is not mixed between sensor channels. The last convolutional layer is connected to a 128 units ELU-Activated fully connected layer, which is then connected to a softmax-activated `n_cl`*asses* units fully connected layer, used as an output for class probabilities.

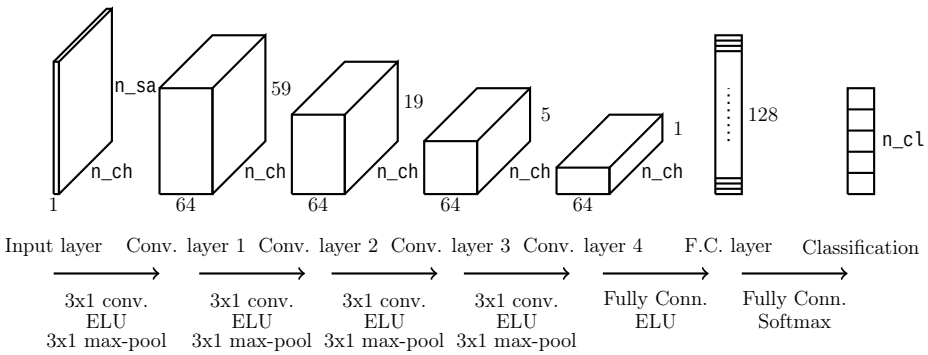

Figure 2.1: Structure of the CNN used for recognition [1]

The specific layer dimensions are primarily defined by the watch capabilities, but are also highly adaptable to the context:

- `n_ch`, the number of sensor channels used as an input, is dependent on the sensors available on the watch, and on the number of channels each of them has (some of them returning a 3-axis triplet). As shown in [3], networks based on fewer channels do not necessarily show worse precision.
- `n_sa`, the number of samples used to build the input frame, can be determined by the sampling rate of the incoming signals (possibly resampled between sensors and neural network input), and the considered length of the input. In this project, the timing parameters were kept as in [1], namely `n_sa` = 180 samples, and a sampling rate of 30 Hz for each channel, effectively making 6 s windows.
- `n_cl`, effectively the number of classes to be recognized at the output, is defined by the user, as the number of classes one wants to recognize. Here, we used `n_cl` = 5, with the following classes (the enumeration numbers correspond to the output tensor ordering):
  0. `Null`: Default class, corresponding to resting periods and turns;
  1. `Freestyle`: for classifying crawl;
  2. `Breaststroke`;
  3. `Backstroke`;
  4. `Butterfly`.

2.2 Training Method and Data

Acquisition The training data is acquired by equipping swimmers with the watch. During their training, the data provided by the different available sensors is stored on the watch, and generally, a person overseeing the training logs what specific swimming styles were swum (this can also be done by the swimmer himself, especially if he/she follows a specific program).

Labeling In a second step, the data is first resampled and then manually labeled; each timestamp of the signal is assigned one of the previously mentioned classes. This is done by using a graphical tool (which has also been expanded during this project) to increase user-friendliness.

Training To train the network, windows of sensor data of size `n_ch` × `n_sa` are prepared. At this step, the following methods are applied to increase the robustness of the algorithm, in this order:

- *time-scaling*: two copies $y_\alpha(t)$ of the original data $x(t)$ are generated, using $y_\alpha(t) = x(\alpha t)$, and with $\alpha \in \{0.9, 1.1\}$;
- *window-cutting*: at this step, the 3 (original and time-scaled) versions are separated into $\texttt{n\_ch} \times \texttt{n\_sa}$ windows. To always have a window having its center close to the classified timestamp, the windows overlap by $\frac{5}{6} \cdot \texttt{n\_sa}$;
- *normalization*: to avoid magnitude variations between swimmers, sensors and watches, each channel of each window is normalized;
- *noise addition*: zero-mean Gaussian noise with a standard deviation of 1% is added to each normalized window;
- *measurement reversal*: to simulate the user wearing the watch on the other wrist, the additive inverse of channels $\texttt{acc}_\texttt{x}, \texttt{mag}_\texttt{x}, \texttt{gyro}_\texttt{y,z}$ is used with a probability of 50%;
- *measurement rotation*: for the tridimensional sensors, the windows are rotated by an angle θ uniformly sampled from the interval $[-\frac{\pi}{6}, \frac{\pi}{6}]$, approximating the signals obtained by using a loosely worn watch.

Other techniques could be applied for further improvements, like time-reversal, some of them have been tested in [3].

Finally, the neural network is trained by minimizing the negative log-likelihood on mini-batches consisting of 64 windows stratified across users and classes. All the available and valid (with one dominant class, and no unlabeled parts) windows are then used to train the network.

Chapter 3

SwitP Application

The main result of this project is the application that will be presented in this chapter. It implements the original classification method, a graphical user interface as well as pre- and post-processing functions.

During development, priority was given to develop a user-friendly interface, keep the code simple and relatively efficient, as well as to make the implemented solutions compatible with other Android smartwatches. To avoid freezing the UI by overloading the main thread, and allow the scheduler to load balance across the multiple cores constituting the CPU (4 cores for the Snapdragon 2100 used in the Nixon the Mission), the computationally intensive tasks are run asynchronously in different threads.

The application was developed from scratch on a Nixon The Mission smartwatch [7], which at the time runs Wear OS `2.17` and stock (non-unlocked nor rooted) Android `8.0.0`. The code is developed on Android Studio and stored/versioned/built (using continuous deployment) on:

`gitlab.ethz.ch/disco-students/fs20/smartwatches_swimming_daniel`

This repository also gives information on how to install the application on the watch. In the following, an overview of SwitP is given.

3.1 General Structure

The UI of the system, which also controls the whole backend, is mainly based on two screens (named activities in Android).

Figure 3.1 gives an overview of how these two activities are linked, as well as what parts are related to which activity. The first activity appears on the launch, just shows start/exit buttons, and a sliding drawer menu. This options menu allows to choose if data has to be "simulated" (use data from a specific file on the watch), what parameters have to be exported, as well as set up the length of the pool. The second one is related to the recording, and the only user input is whether the recording has to be stopped. This activity manages the backend

lifecycle as and refreshes swimming parameters (number of strokes, of laps, total time) shown on the watch.

The GUI is also discussed in Section 4.

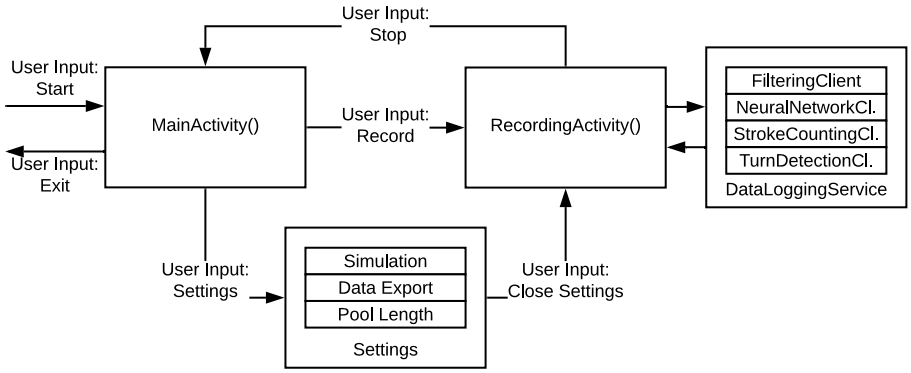

Figure 3.1: General SwitP flowchart

The backend has been divided in multiple parts, which do the pre- and post-processing of the data, and also handle the neural network. The chart shown in Figure 3.1 synthesizes the data flow going through the different steps, which are explained in the upcoming sections.

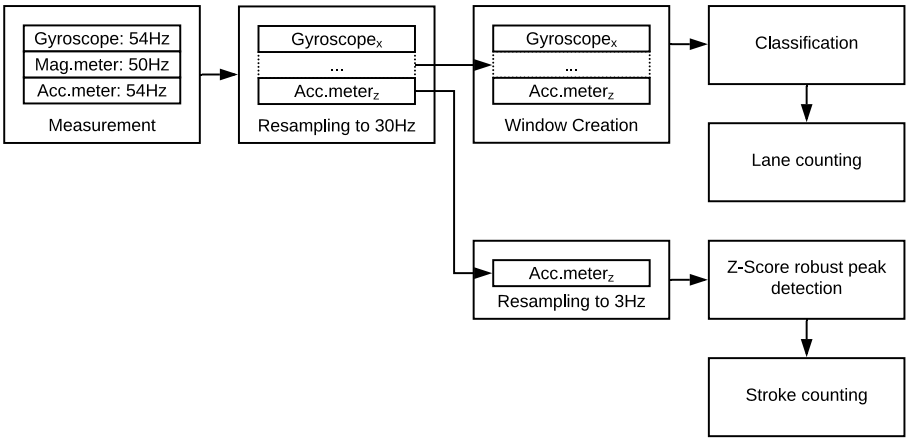

Figure 3.2: Backend structure

3.2 Preprocessing of the Sensor Data

The preprocessing step mainly aims to format the input data to make it usable by the neural network. One main concern is that the sample rates can vary across devices and even sensors. Furthermore, the Android sensor stack [8] does not

allow precise setting of the sampling frequency (just a delay is given) and limits the performance of the embedded sensors [9]. To keep the program adaptable to other devices, this hardware abstraction layer was kept, and another abstraction layer, to preprocess the sensor data, is added.

For this project, the depth of the windows was defined (see Section 2.1) as 180 normalized samples at 30 Hz, which is what constrains the resampler output. In this specific case, the system was limited to three 3D sensors / 9 channels, namely accelerometer, gyroscope, and magnetometer, which are available on most commercial smartwatches (unlike light or pressure sensors), and have shown good precision in [3] and [1]. Furthermore, keeping a small number of channels reduces power consumption, not only by actively polling fewer sensors but also by performing fewer tensor operations when using the neural network.

Resampling - original approach Originally (in [1]), the sampling rate problem is addressed by using cubic interpolation on the raw sensor signals, sampled at 100 Hz and 50 Hz. While this solution is simple to implement for the training stage and still allows a good classification performance, it violates the Shannon-Nyquist sampling theorem (it can be related to downsampling, which introduces aliasing if not preceded by adapted filters).

Resampling - this thesis First, the sampling rate is set as low as possible (while still ≥ 30 Hz). The actual sampling rates on the Nixon the Mission are measured (by using the available timestamps) at 50 Hz for the magnetometer, and 52 Hz for the accelerometer and gyroscope, which use the same measurements. These frequencies are then used to design a resampler, which first upsamples the signal, filters it (considering only the samples kept after decimation, for efficiency) to the output Nyquist frequency (15 Hz in this case), and finally downsamples it. To be able to use the old (saved) as well as the new recordings, a few filters are designed for frequencies between 50 Hz and 104 Hz, corresponding to the available sampling rates. These FIR equiripple filters are designed using the FilterDesigner tool in MATLAB, with a passband ripple of 0.4 dB, a stopband attenuation of 20 dB, and transition frequencies depending on the specific filter. For filters respecting these constraints with a small number of taps, the specificity of the constraints is artificially increased beyond the aforementioned characteristics to obtain a group delay of 195 $ms \pm 1$ ms for all resamplers, ensuring the sensor signals are all in phase after filtering.

Window creation A timer is used to create a new window every second, with the latest available data using the latest available timestamp across all sensor channels. This data is then normalized by channel, by computing the average and standard deviation of the 180 selected samples. This step could be optimized,

if the intermediate values needed for normalization were stored and updated, instead of computing them from scratch every time. However, as the measurement frequency can show small jitter, which in turn makes window length vary a bit, this would need a more complex approach to compensate for border effects.

Resampling for stroke counting For stroke counting, the required sampling frequency is much lower, as the stroke rate of swimmers is limited (between $50 - 100$ strokes per minute depending on the style and distance), and therefore lower sampling rates retain most if not all of the necessary information. As the measurement is done on one arm only, a maximal frequency of about 0.8 Hz is expected, meaning the sampling frequency should be at least 1.6 Hz. Nevertheless, Davey et al. [10] use a frequency of 0.5 Hz to do peak detection in realtime, while Chakravorti et al. [11] detect zero-crossings on a $1Hz$ sampling rate signal. The work of Bächlin and Tröster [12] is based on the gradient of a signal with a sampling rate of 2.56 Hz. Finally, Wang [3], using the same watch as the one used for SwitP, shows that for cutoff frequencies between 1.5 Hz and 15 Hz, also using different combinations of all the available sensors, the best results are obtained with the lowest frequencies. For this project, the sampling frequency of the signal used for stroke counting is set to 3 Hz, and a filter with the same constraints as for the other resamplers is designed. To save on filter taps and computational load, the signal is resampled from the 30 Hz signal instead of the raw sensor output.

3.3 Adaptation to a Mobile Application

In [1], the neural network was designed to run on a fully-fledged Tensorflow install, which is not available on embedded devices. The TFLite framework [2] allows converting a Tensorflow model into a TFLite model, which uses different data structures to store models, and is provided with interpreters to run it on mobile devices. This framework also allows different levels of optimization, ranging from the mentioned data structure optimization to detecting the range of activations and quantizing them to `uint8`, using appropriate scale factor and intercept.

For this project, the classification speed of the watch was first evaluated using the TFLite Model Benchmark Tool [13] which generates an executable running the converted Neural Network with random inputs. While this is not a completely representative test, it allows estimating inference performance without having the preprocessing parts. For windows with 11 channels and 180 samples, the inference time was under 7 ms per evaluation (while theoretically, the maximum available time would be $1s$ to have every window ready exactly on time, without accounting for the rest of the application).

This successful test showed that the default settings of TFLite, with no quan-

tization, on the specific Nixon the Mission smartwatch, run the inference fast enough to not require further optimizations.

3.4 Postprocessing

Lap counting A method for lap counting basing on the probability p_0 of the `Null` class is proposed in [1, Ch. 4], and can be synthesized as follows:

First the clear (with marked transitions) laps are detected:

1. Generate binary signal $r(t)$ by thresholding p_0 with threshold $\tau_p = 0.5$ (active high);
2. Filter out transitions for which the contiguous $r(t) < t_{min}^{(transition)} = 6\ s$;
3. Filter out laps (for which $r(t) = 0$) which are shorter than $t_{min}^{(lap)} = 22\ s$;

This removes most of the false positives which can be detected when the swimmer is resting, or if the neural network predicts the `Null` class for a singular time point in the middle of a lap. However, it also removes turns which are too short, and also turns in which the network is less confident (p_0 small). This can lead to predictions of very long laps.

To avoid this, an upper bound of $t_{max}^{(lap)} = 75\ s$ is introduced, and if the time between two detected laps is higher than this bound, the lap is recovered by progressively reducing $t_{min}^{(transition)}$ and τ_p, until the thresholding gives a new lap.

The original implementation of this method is designed to be run offline, meaning the initial three steps were applied to all timestamp-probability pairs first, and then missing transitions are detected for the whole signal. For this project, however, the goal is to have it running in real-time, so this setup was converted to a finite state machine, illustrated in Figure 3.3

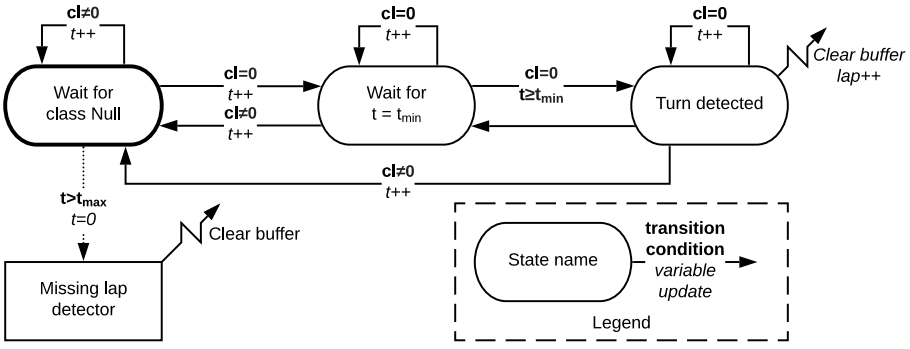

Figure 3.3: Finite State Machine for Lap Detection

If no laps have been detected for too long, a "missing lap detector" is started with the data that is missing turns (according to $t_{max}^{(lap)} = 75\ s$) as input, running asynchronously in the background.

Stroke counting Basing on the 3 Hz-resampled z-channel of the accelerometer, different approaches can be used to extract the number of strokes a swimmer does.

The main goal is to have a computationally inexpensive approach which still shows robustness to changing swimming styles, intensities, and even watches.

Multiple authors use classical approaches for this task. Siirtola, et al. [5] use measured data to determine a threshold level for detecting strokes, and remove spurious results by limiting the stroke rate to realistic values. Bächlin and Tröster [12] compute the gradient in a 0.2 s sliding window, and then use its maximums to determine both wall pushes and arm strokes. Chakravorti et al. [11] stream the data to a computer, which then filters the data and uses zero-crossing detection to find strokes. N.Davey, M.Anderson et al. [10] approach this task by first calculating the mean, finding minimas / maximas below / above the mean, and discarding false positives by removing peaks of same polarity not separated by a peak of opposite polarity.

Methods based on learning the data have also been developed, and use different approaches. Wang [3] first filters the data, performs Principal Component Analysis to find the most useful channels, and then tunes a peak detector by using grid search against the manually counted strokes. Schmidt et al. [14] use a convolutional neural network to detect peaks and the baseline in a more general signal processing context. They do so by using the available data to tune a "filter" constituted of a CNN, followed by a non-linear readout layer, and show its performance in synthetic contexts is better than optimized continuous wavelet filtering methods.

All the mentioned approaches make a compromise between hard-coding some thresholds, or needing specific data to optimize them during training. The method proposed in this project adds an adaptive component, also eliminating the need for a supplementary normalization layer. It is based on computing the z-Score, which is a statistical value indicating how far (in term of standard deviations σ) a specific sample x is from a population with mean μ. It is defined as:

$$z = \frac{x - \mu}{\sigma}$$

Intuitively, having a large absolute value of the z-Score means the sample x is an outlier, or part of a peak. To use this on real-time data, Van Brakel [15] proposed an adaptive method, which takes 3 parameters:

- t_d: dynamic threshold, number of standard deviations necessary to consider

a sample as an outlier;

- l: lag, size of the window used for the computation of μ_i and σ_i;
- q: influence, between 0 and 1, a parameter for filtering the effect of new measurements.

A parameter t_s was added for SwitP, which adds some noise tolerance by ensuring a minimum, non-adaptive, acceleration threshold has to be passed before a peak is signaled. x_i is used to denote the i^{th} input sample , while y_i indicates if it is an outlier, and p_i if it is considered as a peak (because groups of outliers can be contained in a single peak). Mathematically, the algorithm can be written as follows:

$$\tilde{x_i} = \begin{cases} x_i, i \in \{0, \dots, l-1\}; \\ q \cdot x_i + (1-q) \cdot x_i, otherwise. \end{cases}$$

$$\mu_i = \frac{\Sigma_{i-l+1}^{i} \tilde{x_i}}{l} \qquad \sigma_i = \sqrt{\frac{\Sigma_{i-l+1}^{i} \left(\tilde{x_i}^2 - \mu_i\right)^2}{l-1}}$$

$$z_i = \frac{\tilde{x_i} - \mu_i}{\sigma_i}$$

$$y_i = \begin{cases} 1, z_i \geq max(t_d \cdot \sigma_i, t_s); \\ 0, otherwise. \end{cases} \qquad p_i = \begin{cases} 1, y_i - y_{i-1} = 1; \\ 0, otherwise. \end{cases}$$

The small number of tests (discussed in Section 4) did not allow a precise setting of the 4 parameters. Just as a baseline, values $t_d = 2, t_s = 0.25, l = 5, q = 0.15$ were chosen.

Figure 3.4 visually shows how these parameters affect the computation of the peaks, and the adaptive characteristics of the algorithm. It also shows how for slower rising peaks, the first of the outliers is selected, which is not necessarily the largest value.

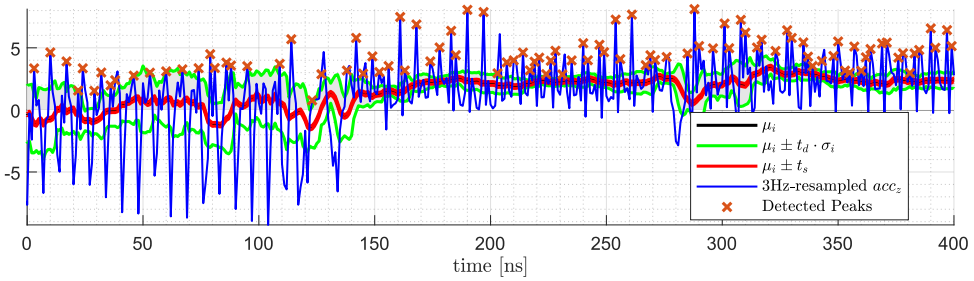

Figure 3.4: Example of z-Score based stroke counting, with intermediate variables used for peak detection

Chapter 4

Testing

The testing of the different parts of SwitP could not be done as foreseen, as all pools closed before the app had been developed, and did not reopen for the testing to be performed as planned. Nevertheless, the basic functionality could be tested in multiple ways, which will be explained in this chapter.

A first testing method is to read existing recordings, unlabeled, and feed them through the data pipeline shown in Figure 3.1 (replacing the "Measurement" block). This is done by using the "simulate" option in the menu shown in Figure 4.2 (b). Specific data fields (selected using the same options menu) can then be exported to a `.CSV` file, making the data available for further analysis. As an example, Figure 4.1 is based on a recording done in 2018, during which the 4 recognized swimming styles were swum over 50 m each, two times in a row.

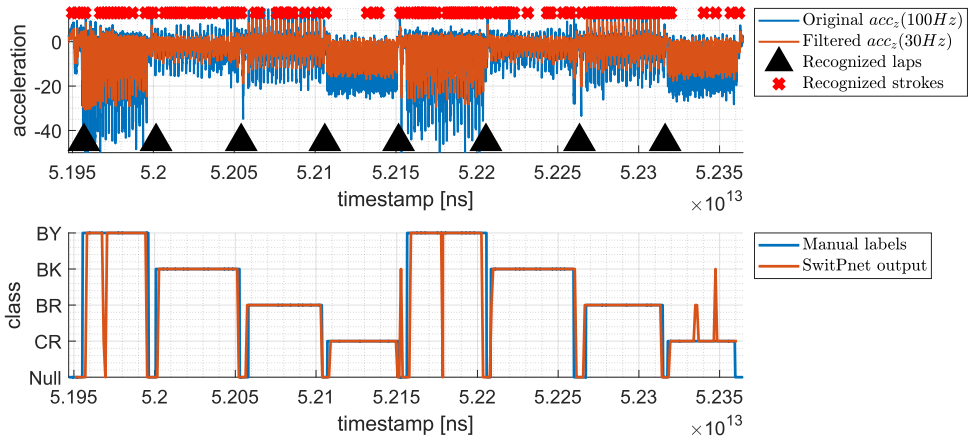

Figure 4.1: Returned values on a $2 \cdot 200$ m medley

Figure 4.1 shows that the raw predicted classes are generally very close to the manual labels for this user (which was excluded from the training set). The predictions could be improved by implementing some smoothing, looking at estimations from window covering the same timespan, or by classifying more than

once a second, and also using some kind of averaging. The lower part of Figure 4.1 shows that for this training set, the laps were recognized correctly. The stroke recognition part shows inconsistent results, but as discussed before, it needs some fine-tuning to provide good results.

While the user interface could not get thoroughly tested and reviewed by persons external to the project, it proved to be well usable in the pool. Figure 4.2 shows some of the most important menus appearing on the watch. The start and stop buttons have to be pressed and held during a certain time, during which a loading circle is displayed (as seen in (c)), after which a confirmation screen like in (d) is shown, to ensure no false deactivations are made in the pool.

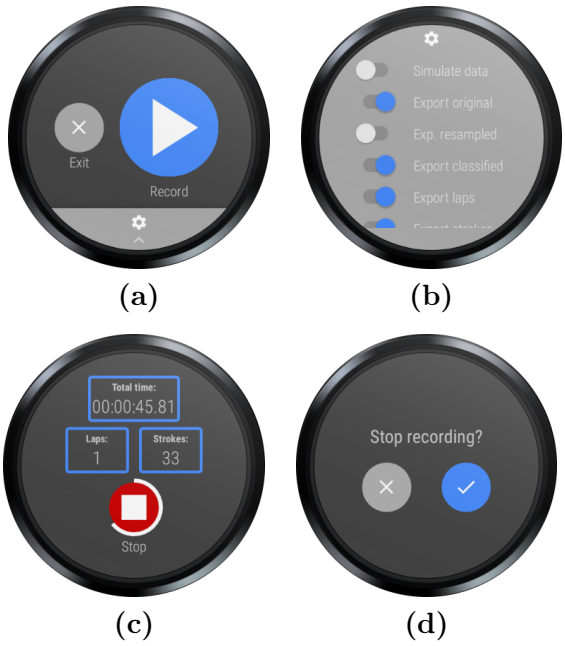

Figure 4.2: Screenshots of the SwitP application: **(a)** Start menu **(b)** Options drawer **(c)** Recording screen **(d)** Confirmation screen

Per default, Android applications all support backswipe to change activities, but also to exit an application. As this function can also be wrongly activated by pool water passing on the screen, for SwitP backswipe is disabled, and the exit button (as shown in (a)) is used to exit the application. The only physical button available on the Nixon the Mission used for this project still has its default function, namely returning to the main menu.

Finally, the last tests show that while putting the watch in airplane and battery saver modes increases its autonomy by several days, it also temporarily hangs the app and measurements, which should be addressed in the next iteration of SwitP development.

Chapter 5

Conclusions

SwitP implements the convolutional neural network developed by Brunner et al. [1] to recognize swimming styles on an Android smartwatch. Using only the rotation, acceleration, and magnetic field sensors (available on most modern devices), this neural network can reliably predict what swimming style was swum.

The conversion of the neural network for mobile applications is done by using the Tensorflow Lite framework [2], which integrates both the conversion and prediction functions. Beforehand, the raw data produced by the sensors is filtered and resampled to be usable by the neural network, which is done by using an implementation optimized both for embedded performance and for adaptability to other watches.

The classification results as well as the resampled data are then used to determine the number of lengths and the number of strokes that the swimmer has swum. Finally, a user interface is added to the application, designed to be used in the pool and giving live feedback to the swimmer.

5.1 Future Work

A structured user-test of SwitP would help improving the application. On one hand, letting the interface be used by persons external to the project will test its ergonomy. On the other hand, these tests are required to fine-tune the adaptable parameters (e.g for stroke-counting) and get reliable results.

The adaptability of the preprocessing system can be increased, by automatically choosing the optimal sampling rates and resampler settings. This will remove the necessity to change these parameters between watches. As mentioned in Section 3.2, the window creation method can be made more efficient by implementing a "rolling" system, instead of computing everything from scratch every time.

The convolutional neural network itself can be improved in multiple ways. First, using more training data, the kicking exercises done by swimmers might

be recognizable as a new class. Secondly, more data augmentation techniques, such as those tested by Wang [3], could increase its precision.

Finally, the training experience could be improved by allowing programming of stuctured workouts, with a live training progress indicator in the watch UI.

Bibliography

[1] G. Brunner, D. Melnyk, B. Sigfússon, and R. Wattenhofer, "Swimming style recognition and lap counting using a smartwatch and deep learning," in *ISWC '19, London, United Kingdom*, Sep. 2019.

[2] tensorflow.org. Tensorflow Lite. [Online]. Available: https://www.tensorflow.org/lite

[3] Y. Wang, "Swimming activity recognition with smartwatches and deep learning," Master's thesis, ETH Zürich, Oct. 2019.

[4] R. Mooney, G. Corley, A. Godfrey, L. R. Quinlan, and G. ÓLaighin, "Inertial sensor technology for elite swimming performance analysis: A systematic review," *Sensors*, vol. 16, no. 1, Dec. 2015.

[5] P. Siirtola, P. Laurinen, J. Röning, and H. Kinnunen, "Efficient accelerometer-based swimming exercise tracking," in *2011 IEEE Symposium on Computational Intelligence and Data Mining (CIDM)*, 2011, pp. 156–161.

[6] J. Pansiot, B. Lo, and G. Yang, "Swimming stroke kinematic analysis with bsn," in *2010 International Conference on Body Sensor Networks*, 2010, pp. 153–158.

[7] Nixon. The Mission. [Online]. Available: https://www.nixon.com/ch/en/mission-support

[8] Android. Sensor stack. [Online]. Available: https://source.android.com/devices/sensors/sensor-stack

[9] L. Sigcha, I. Pavón, P. Arezes, N. Costa, G. De Arcas, and J. López, "Occupational risk prevention through smartwatches: Precision and uncertainty effects of the built-in accelerometer," in *Sensors*, vol. 18, no. 3805, 2018.

[10] N. Davey, M. Anderson, and D. A. James, "Validation trial of an accelerometer-based sensor platform for swimming," *Sports Technology*, vol. 1, no. 4-5, pp. 202–207, 2008. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/jst.59

[11] N. Chakravorti, T. Le Sage, S. E. Slawson, P. P. Conway, and A. A. West, "Design and implementation of an integrated performance monitoring tool for swimming to extract stroke information at real time," *IEEE Transactions on Human-Machine Systems*, vol. 43, no. 2, pp. 199–213, 2013.

[12] M. Bächlin and G. Tröster, "Swimming performance and technique evaluation with wearable acceleration sensors," *Pervasive and Mobile Computing*, vol. 8, pp. 68–81, 02 2012.

[13] tensorflow.org. Tensorflow Lite Benchmark Tool. [Online]. Available: https://github.com/tensorflow/tensorflow/tree/master/tensorflow/lite/tools/benchmark

[14] M. N. Schmidt, T. S. Alstrøm, M. Svendstorp, and J. Larsen, "Peak detection and baseline correction using a convolutional neural network," in *ICASSP 2019 - 2019 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2019, pp. 2757–2761.

[15] J.-P. van Brakel. Smoothed z-score algorithm. [Online]. Available: http://stackoverflow.com/questions/22583391/peak-signal-detection-in-realtime-timeseries-data