DTU Compute
Department of Applied Mathematics and Computer Science

Guiding 3D Human Pose Estimation using Feet Pressure Sensors

Master Thesis

Guiding 3D Human Pose Estimation using Feet Pressure Sensors

Master Thesis
February, 2023

By
Jorge Sintes Fernández

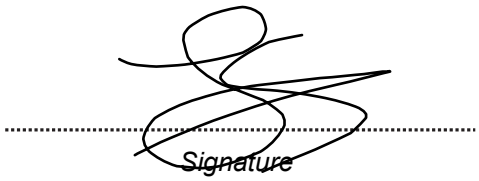
Cover photo: Vibeke Hempler, 2012
Published by: DTU, Department of Applied Mathematics and Computer Science, Richard Petersens Plads, Building 324, 2800 Kgs. Lyngby Denmark
www.compute.dtu.dk

Approval

This thesis has been prepared over the course of five months at Grazper Technologies ApS, with supervision of the Section for Visual Computing, Department of Applied Mathematics and Computer Science, at the Technical University of Denmark, DTU, in partial fulfilment for the degree Master of Science in Mathematical Modelling and Computation.

It is assumed that the reader has a basic knowledge in the areas of linear algebra, multivariable calculus, deep learning, computer vision and software engineering.

Jorge Sintes Fernández - s202581



Signature

01/02/23

Date

Abstract

Human Pose Estimation is a task in the field of computer vision that involves identifying and capturing the positions and orientations of the human body. This is typically done by predicting the locations of specific keypoints, such as hands, head, and elbows, in an image. Human Pose Estimation has various applications in different industries, including robotics, augmented reality, gaming, accessibility, sports, and security.

Grazper Technologies ApS, the partner for this thesis, is working on developing a real-time 3D human pose estimation system using a multicamera setup. The primary application of this system is in the field of security. However, one of the main challenges in implementing this system is the requirement of multiple cameras to view the same scene from different angles. This restriction limits the usability of the system, especially in security applications where it is unlikely to have more than one or two cameras pointing at the same location at the same time.

The aim of the present thesis is to study whether we can improve the 3D pose estimation in these cases by incorporating knowledge about foot contact. To do so, we will acquire an IoT-connected sole pair that can make pressure measurements, and incorporate it into Grazper's current video acquisition setup.

During the course of the thesis, we designed a reliable, stable, and automated data acquisition setup, enabling Grazper to easily record high-quality datasets with the potential to obtain synchronized ground truth sole pressure signals. We prove the feasibility of predicting sole pressure based on the pose using deep learning techniques. Finally, we show how sole contact can enhance the performance of a pose detector in scenarios with fewer cameras.

These results offer a strong proof of concept for future AI solutions and demonstrate the potential of this technique for further development and advancement.

Acknowledgements

I thank my supervisor, Siavash, for his guidance, support, and encouragement throughout this project.

I would also like to thank the Grazper team for their expertise, patience, openness, and sense of humor. They have been invaluable in helping me stay focused and motivated throughout this thesis.

I would like to thank my parents for always being there, supporting me in any decision I take, and being responsible for pretty much every success in my life.

Thanks to José Pablo, Sergio, Juan, Yasmina, Klara, Elvira, Luis, and Vicente. Because, for some reason, they keep being my friends.

And thanks to Ana for making me a better person.

Contents

List of Figures

1 Introduction

Human Pose Estimation is a task in the field of computer vision that involves identifying and capturing the positions and orientations of the human body. This is typically done by predicting the locations of specific keypoints, such as hands, head, and elbows, in an image. Human Pose Estimation has various applications in different industries, including robotics, augmented reality, gaming, accessibility, sports, and security.

Grazper Technologies ApS, the partner for this thesis, is working on developing a real-time 3D human pose estimation system using a multicamera setup. The primary application of this system is in the field of security. However, one of the main challenges in implementing this system is the requirement of multiple cameras to view the same scene from different angles. This restriction limits the usability of the system, especially in security applications where it is unlikely to have more than one or two cameras pointing at the same location at the same time.

The aim of the present thesis is to study whether we can improve the 3D pose estimation in these cases by incorporating knowledge about foot contact. To do so, we will acquire an IoT-connected sole pair that can make pressure measurements, and incorporate it into Grazper's current video acquisition setup.

We will investigate in which way can pressure readings be incorporated into Grazper's pose detection, and design a test environment to characterize the improvements. Furthermore, we will design and train several machine learning models in a supervised fashion, to try to predict these pressure values from the pose itself. This way, we study the possibility of removing the dependency on the sole sensors, allowing our proposals to be incorporated into a real-life application.

2 Background

Human 3D Pose Estimation has seen significant progress in the recent years, driven by the advancements in computer vision and deep learning techniques. In this chapter, we provide an overview of the state-of-the-art techniques for human 3D pose estimation from 2D images.

There are different approaches on how to represent the human body structure in the literature: the skeleton-based model, commonly used in 2D human pose estimation [1] and naturally extended to 3D. It contains many keypoints in specific joints of the human body and connects the adjacent ones using edges. This structure is the one Grazper uses, since their focus is on understanding the person's position, and therefore it will be the ones we will work with in this thesis. More recent works use a triangulated mesh to represent the human skin [2] or the more recently proposed surface-based model [3].

As per datasets, Human3.6M [4] and HumanEva [5] are the standard for 3D human pose estimation, both acquired with a motion capture system and representing different common scenarios, walking, jogging, gesturing, smoking, talking on the phone... [6] provides an extensive list of all the different datasets.

The advent of deep learning has revolutionized the field of human 3D pose estimation. The approaches can be divided in two categories: direct 3D pose estimation, tries to predict 3D coordinates of joints directly from input images via 3D heatmaps [7], or lifting from 2D to 3D pose, inspired by the rapid development of 2D human pose estimation algorithms [8]. This latter approach is the one Grazper is taking, since it is much less computationally expensive and their solution is supposed to run real-time.

On the topic of incorporating foot contact information into 3D pose estimation, we found some articles. [9] proposed a network architecture to detect ground contact events from 2D keypoint estimations to reduce foot artifacts. However, they rely solely on estimates from the images themselves, and not on real ground truth data. Moreover, [10] tackles the 3D pose estimation problem doing a physics based model of the human body and therefore inferring knowledge about foot contact indirectly. However, this setup is more complex than the one proposed by Grazper, which requires to be run real-time..

For sole-pressure sensors, Grazper reviewed several commercial options, considering factors such as intrusiveness, product quality, ease of integration into the data acquisition pipeline, and cost. After a demo with Moticon OpenGo soles [11] and Sensoria smart socks [12], Grazper chose the former as the SDK was more suitable for its integration in the data acquisition pipeline, and since the sensors have to be shared among different actors, soles seemed better than socks for this purpose.

3 3D pose estimation

Pose estimation is a fundamental problem in computer vision, with numerous applications in fields such as augmented reality, robotics, and human-computer interaction. This chapter provides a condensed overview of the approach we have taken on the topic, exploring how to estimate the 3D poses of humans from 2D images.

3.1 Pinhole camera model

Before focusing on pose estimation, we think explaining the pinhole camera model is appropriate. The notation we will use is extracted from [13]. This model describes a camera that projects scene 3D points into the image plane through a perspective transformation. We have two sets of coordinate axes, the world coordinates X_W, Y_W, and Z_W, and the camera coordinates X_C, Y_C, and Z_C. The latter sit at the camera, having X_C and Y_C parallel to the camera plane and Z_C perpendicular. Figure 3.1 depicts the two sets of coordinates and the relative camera plane in blue, where the image gets projected. Two additional sets of 2D coordinates are sitting in the relative camera plane; x and y in world units (in our case meters), and u and v in pixel values. Notice that their origin is not in the same position; the first pair's origin sits where the optical axis intersects the plane, while the latter sits at the top left corner of the image frame.

The homogeneous transformation between world coordinates and camera coordinates is encoded by the extrinsic parameters, a translation and rotation matrix $[R|t]$. If we have a point in world coordinates (P_W) we can express it from camera coordinates (P_C) by:

$$P_C = \begin{bmatrix} R & t \\ 0 & 1 \end{bmatrix} P_W = \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_x \\ r_{21} & r_{22} & r_{23} & t_y \\ r_{31} & r_{32} & r_{33} & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} P_{X_W} \\ P_{Y_W} \\ P_{Z_W} \\ 1 \end{bmatrix}.$$

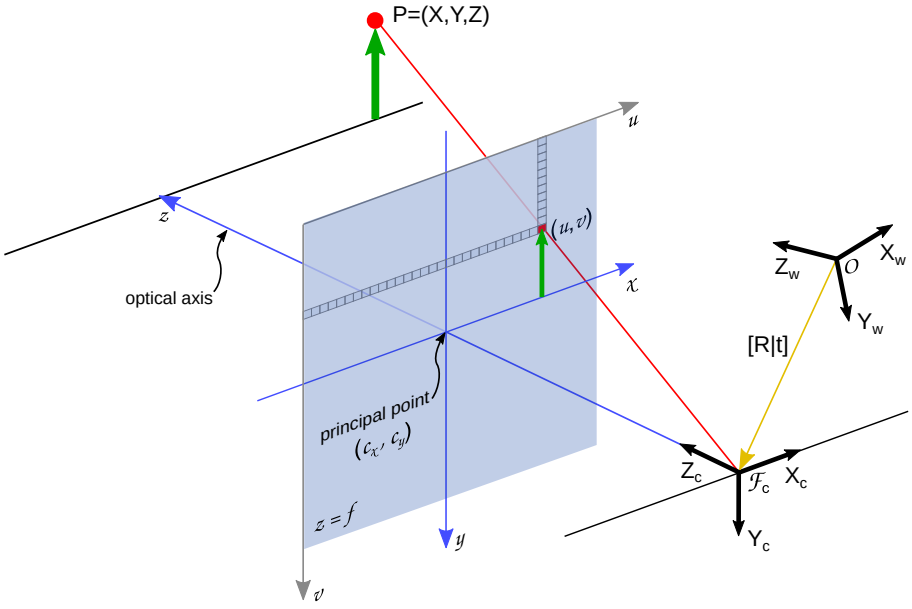


Figure 3.1: Schema of the pinhole camera model [14].

To project 3D points expressed in the camera coordinate system to 2D pixel coordinates, we use the camera intrinsic matrix, composed of the focal lengths f_x and f_y expressed in pixel units, and the principal point (c_x, c_y), usually close to the image center:

$$s \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X_C \\ Y_C \\ Z_C \end{bmatrix},$$

where s is a scaling constant, in this case equal to Z_C. We loose one dimension by doing this transformation. Figure 3.1 depicts how any point on the red line will be projected to the same (u, v) position. We only know this scaling constant when we pass from camera to pixel coordinates. Trying to do the opposite, the scaling constant is unknown and we encounter a perspective problem, we don't know how far away that point is in reality.

Additionally, lenses usually have some radial and tangential distortion, and we compensate for it with the help of some more camera parameters. We follow the same procedure as the recommended one in OpenCV's documentation [15].

In practice, the intrinsic and extrinsic camera parameters are estimated with the help of checkerboard patterns. The intrinsic camera parameters are specific for each camera, so we need to calibrate the cameras individually. Figure 3.2a shows a calibration routine developed by Grazper and used internally for estimating the intrinsic camera parameters. The fisheye distortion of the lens is visible in the image. For the extrinsic parameters, even though each camera has its own, we need to calibrate all cameras simultaneously. We do so by placing a big checkerboard pattern on the floor, in a position where all cameras can see it, and running an OpenCV routine to set the world coordinates at a specific location in the pattern and calculate each camera's extrinsic parameters (see figure 3.2b).

(a) Intrinsic calibration routine

(b) Extrinsic calibration

Figure 3.2: Intrinsic and extrinsic camera calibration from the camera point of view

3.2 2D pose estimation

We decided to approach 3D human pose estimation by estimating the 2D pose using multiple calibrated cameras and then triangulating it. This is the method Grazper uses when annotating recording data since, provided enough camera angles, is very accurate. Grazper has developed its own pose estimation networks, but we will use one of the state-of-the-art networks for this thesis. We decided to use Google's MediaPipe BlazePose [16] for several reasons.

First the skeleton topology. Unlike other models, BlazePose outputs two keypoints per foot: heel and toe, which is very convenient for our use case. Plus, using off-the-shelf software makes our work more reproducible. Furthermore, Mediapipe is licensed under the Apache License 2.0, a permissive license that allows for commercial use, modification, and patent use. For this reason, Grazper had already integrated this skeleton typology into its software stack, allowing us to use its data annotation software without any modifications. Finally, BlazePose has been designed to achieve real-time performance on mobile phones using CPU inference, so it is fast when run on powerful computing platforms.

MediaPipe BlazePose expands the standard COCO topology of 17 keypoints [17] to 33, including keypoints in hands, face, and feet. The BlazePose skeleton topology is depicted in figure 3.3. The model works in two steps, first it locates the pose region-of-interest (ROI) within a frame and then uses a pose detector to predict the location of all 33 pose keypoints on the ROI. On subsequent frames, the ROI is estimated with a tracker based on the previous frame's pose keypoints, reducing the computational cost. The first part of the network is only re-run on frames where the keypoints cannot be found on the estimated ROI.

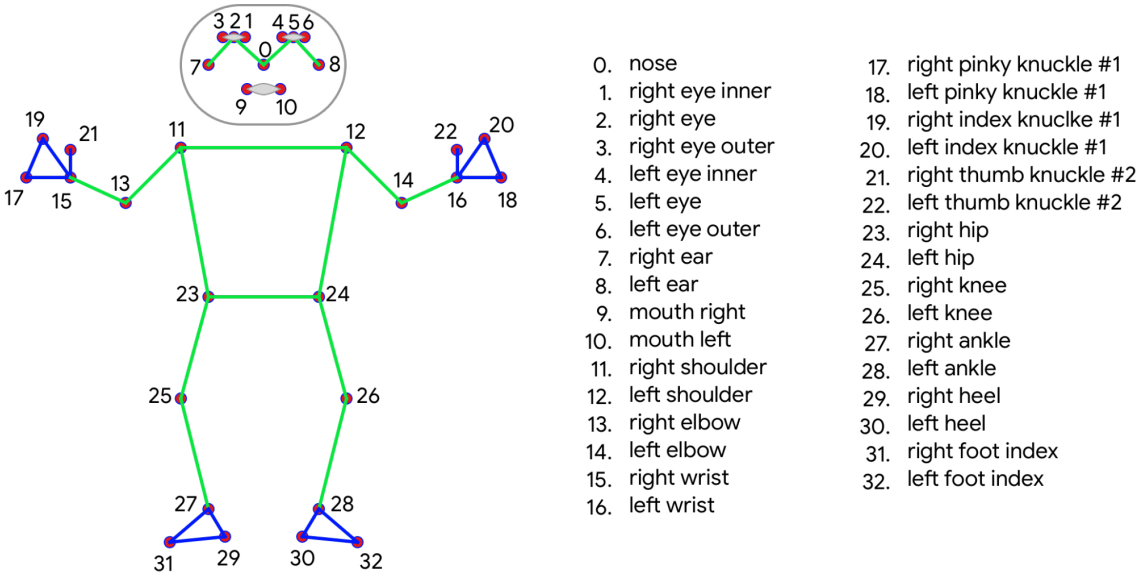


Figure 3.3: BlazePose 33 keypoint topology as COCO (colored with green) superset

We used the BlazePose GHUM Full network for all 2D pose estimations in this thesis, with all the default parameters. The output from the network is a list of pose landmarks, each consisting of:

- x and y coordinates normalized to $[0, 1]$ by the image width and height, respectively, from which we can calculate the u and v values.
- z represents a depth estimate of each keypoint, taking the hips' midpoint as the origin. The Mediapipe docs state that the magnitude of z uses roughly the same scale as x, which is not entirely clear.
- `visibility`: A value in $[0, 1]$ indicating the likelihood of the landmark being visible (i.e. not occluded) in the image.

3.3 3D triangulation

Using multiple (N) cameras, a 3D pose can be estimated by triangulating keypoints. We can assume that the cameras are synchronized. So, both frames in each camera's video feed correspond to the same moment in time. Later, in chapter 4, we will explain how we achieve the synchronization in practice. We then triangulate frame by frame by doing the following:

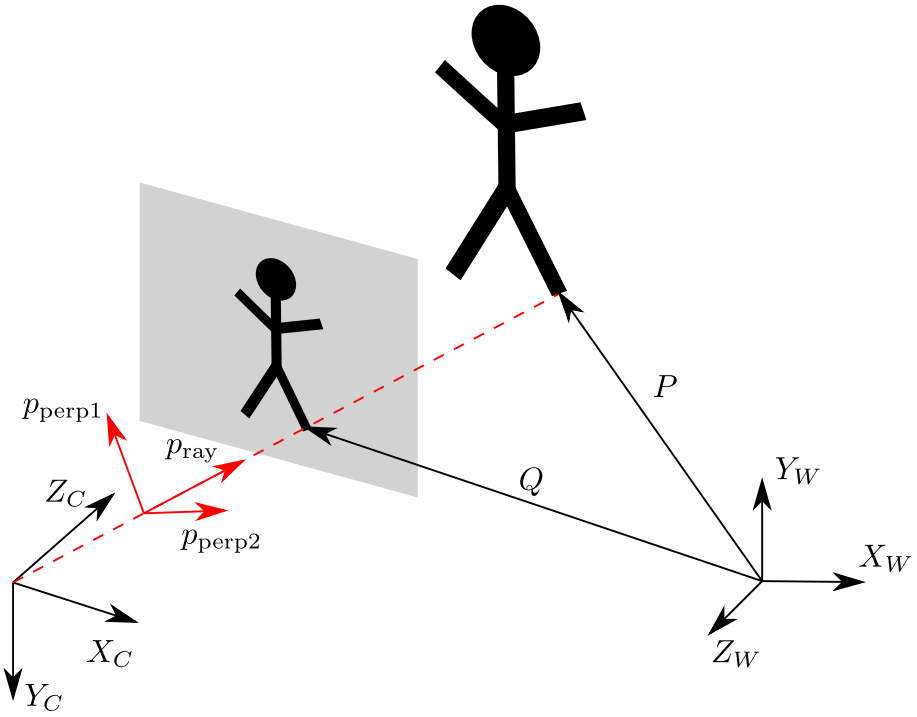


Figure 3.4: Diagram showing the intuition behind our triangulation algorithm.

Every keypoint in the skeleton is tracked separately. Figure 3.4 illustrates the situation when triangulating one keypoint, and it will help the reader understand the intuition behind our 3D triangulation method. Only one camera is depicted for the sake of simplicity. Since we know the camera's intrinsic parameters, we can obtain the position of the keypoint in the relative camera plane in camera coordinates (from $[X_C, Y_C, Z_C]$). p_ray depicts the direction of this vector. If the setup were perfect, each camera's line would intersect in the real position of the keypoint.

However, since our setup has multiple sources of error: the camera matrices and the lens calibration are estimated, points can be occluded and the pose detection model is imperfect the lines will therefore not intersect. We can solve this problem geometrically by finding the closest point to all lines.

We want to obtain p_ray in world coordinates. If we translate the keypoint vector from camera coordinates to world coordinates we will obtain vector Q. We can finally subtract the camera position from world coordinates Q and, after normalising, we obtain p_ray. The camera position is the 4th column of the inverse extrinsic matrix.

We can now obtain an orthonormal set of coordinates through Gram-Schimdt orthonormalization, p_{perp1} and p_{perp2}. With this orthonormal basis, one can see that the projection of P onto the plane formed by p_{perp1} and p_{perp2} is the same as the projection of Q. In other words, if we had vectors P and Q expressed in this new basis without translating, their vector components in the direction of p_{perp1} and p_{perp2} would be the same. Since we know the components of these vectors in world coordinates, we know the rotation matrix that corresponds to this transformation.

Let us call q the rotated Q vector, and write the following linear system of equations:

$$\begin{bmatrix} p^x_{ray} & p^y_{ray} & p^z_{ray} \\ p^x_{perp1} & p^y_{perp1} & p^z_{perp1} \\ p^x_{perp2} & p^y_{perp2} & p^z_{perp2} \end{bmatrix} \begin{bmatrix} P^x \\ P^y \\ P^z \end{bmatrix} = \begin{bmatrix} q^{p_{ray}} \\ q^{p_{perp1}} \\ q^{p_{perp2}} \end{bmatrix}.$$

We know this system doesn't have a solution, since the first line is simply wrong. We will address this issue shortly. For now, we can write a similar system for all cameras in the scene. Each camera will have its own p_{ray}, p_{perp1}, p_{perp2}, and q, following the same procedure mentioned above. The world set of coordinates is unique, since the cameras are calibrated. Then, we can write them all together in an overdetermined linear system of the form $Ax = b$:

$$\begin{bmatrix} \begin{bmatrix} p^x_{ray} & p^y_{ray} & p^z_{ray} \\ p^x_{perp1} & p^y_{perp1} & p^z_{perp1} \\ p^x_{perp2} & p^y_{perp2} & p^z_{perp2} \end{bmatrix}^1 \\ \vdots \\ \begin{bmatrix} p^x_{ray} & p^y_{ray} & p^z_{ray} \\ p^x_{perp1} & p^y_{perp1} & p^z_{perp1} \\ p^x_{perp2} & p^y_{perp2} & p^z_{perp2} \end{bmatrix}^N \end{bmatrix} \begin{bmatrix} P^x \\ P^y \\ P^z \end{bmatrix} = \begin{bmatrix} \begin{bmatrix} q^{p_{ray}} \\ q^{p_{perp1}} \\ q^{p_{perp2}} \end{bmatrix}^1 \\ \vdots \\ \begin{bmatrix} q^{p_{ray}} \\ q^{p_{perp1}} \\ q^{p_{perp2}} \end{bmatrix}^N \end{bmatrix}.$$

If the setup was perfect, we could remove the first row of each camera's triplet, and the system would have one unique solution. Since that is not the case, the system will have no solution. However, we can find an estimate using Weighted Least Squares. We set this vector of weights with a 0 in the first line of each camera's system to avoid these wrong lines from contributing to the solution. For the other weights, we propose to use each camera's `visibility` for the keypoint that is being triangulated. That way the solution will be closer to the cameras that have a better estimate of the keypoint.

However, this method breaks when the keypoint can only be seen from one camera. Then, the WLS solution will just be the vector Q. For this reason, we added the condition that each keypoint should be visible from at least two cameras with a `visibility` of 0.5 in order to triangulate.

3.4 One-camera 3D pose estimation

We also approached the one-camera 3D pose estimation problem in this thesis, which requires a different approach than triangulation. Our proposal is combining the pseudo3D output from BlazePose, together with foot contact knowledge, to obtain a 3D pose estimation. We will describe our approach and mention the several problems we ran into in this section.

For every frame in the video, we give each foot's heel and toe a score based on their `visibility` and the pressure each channel received, and then select the one with the highest score. Since we know that this joint is in contact with the ground, we know it is contained in the ground plane. So, the real position of the keypoint would sit at the intersection of the keypoint line with the ground floor. This point is known, and we can then find the scaling contact s.

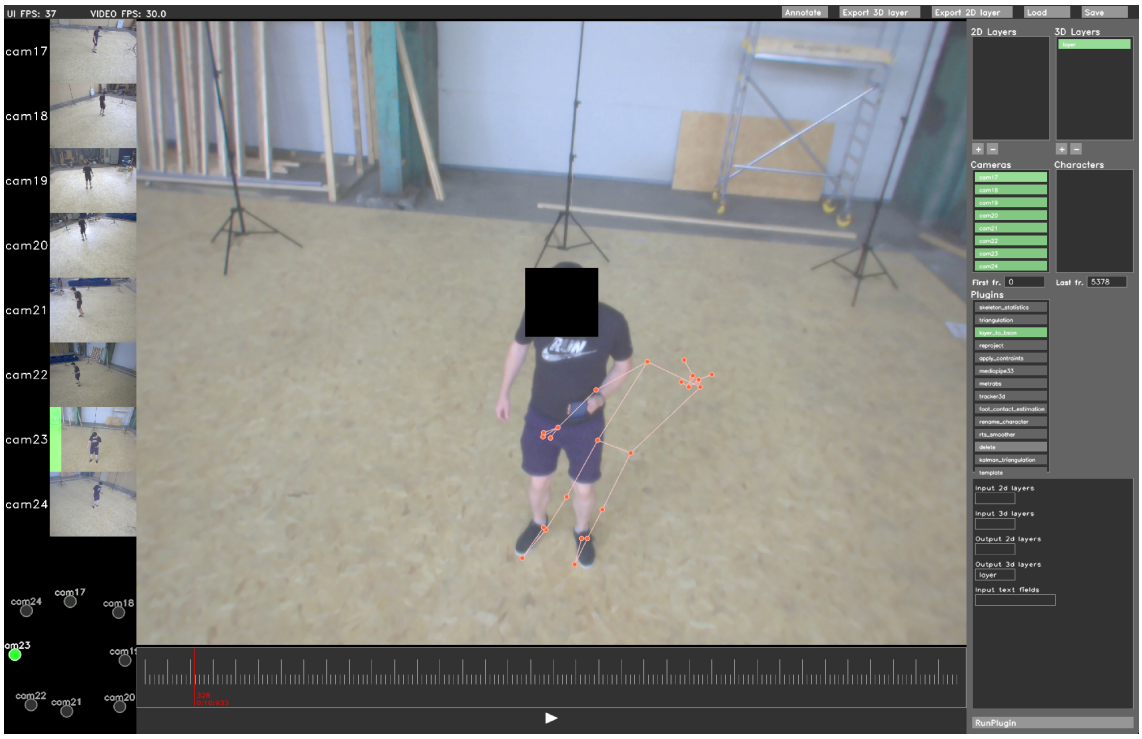
Of course, this only solves the scaling problem for that specific keypoint, and we rely on how good the pseudo-3D prediction actually is. Here is where we encountered the issue. We could not find a good way to use the model's z estimation. The documentation is not clear on how this output should be used. We tried several interpretations, and none yielded a reasonable-looking skeleton. In order to have so, it was needed to divide by an arbitrary constant we could not argue for.

On top of that, the output was always tilted at roughly the same angle as the camera. It almost seems like the model was expecting the camera to be looking in a horizontal direction, parallel to the ground. Figure 3.5 depicts this behaviour. Here, we ran BlazePose on view 3.5a and used the method described above to calculate the 3D pose. When viewing from 3.5b this tilting behaviour becomes evident.

We could not find an easy way around this issue. One possible alternative was, first, tilting the output ourselves, which posed the question, can we be sure the tilting is consistent and it depends solely on the camera position? How do we know if we are the ones causing the issue by misinterpreting how this depth estimate should be used? Another way was picking a state-of-the-art model like MeTRAbs [18], which also uses the camera matrices as an input to the model and claims to provide a better 3D pose estimation from only one angle. However, it lacks all the benefits that made us choose BlazePose in the beggining. Alternatively, we could have calculated the 3D pose by triangulating from all eight cameras, project this skeleton into each 2D camera and use this annotations to train MediaPipe's model further in order to correct the tilting behaviour.

Even though we think some of these options would have worked, some were far from the scope of the thesis, and some others were deemed too time-consuming. Thus, we decided to leave the one-view case and focus on the several-camera scenario for the rest of the thesis.

(a) View from the camera running the model.



(b) Side view of the 3D output, showcasing the tilting issue.

Figure 3.5: One-camera 3D pose estimation attempt, from the view in (a).

4 Data Acquisition

In this chapter, we will present the data acquisition setup used for this thesis and the considerations made to design it. The pipeline has been designed to be modular, flexible, and fast to work with, allowing for easy integration of future additional sensing devices. It consists of two stages: raw data acquisition and preprocessing.

In order to conduct our experimental work, we need a dataset of videos of a single person performing a set of activities (standing, walking, running, jumping) recorded from several angles with different cameras. The video timestamps need to be synchronized accurately since the video frames will be used to triangulate the 3D skeleton of the person in the scene. Luckily for us, Grazper was starting to build a video rig in Refshaløen to record their own pose detection datasets when we initiated the collaboration, so we were involved with the recording process from the beginning. Apart from the video feeds, we need some data on the foot contact. The person in the scene will wear a pair of soles with integrated pressure sensors while performing the activities.

Since the nature of the two capturing methods (video and pressure) is very different, so is their acquisitions and there is little interplay between them. We will describe them separately in the following sections of this chapter.

4.1 Raw data

4.1.1 Video

The video rig consists of a flat wooden floor of 10x10 meters, with scaffolding around it where the lighting and cameras are placed. Grazper has developed its own cameras using an Nvidia Jetson Xavier NX [19] with a USB camera. Each camera is an embedded device running Linux, powered through PoE. The camera and its components can be seen in image 4.1.

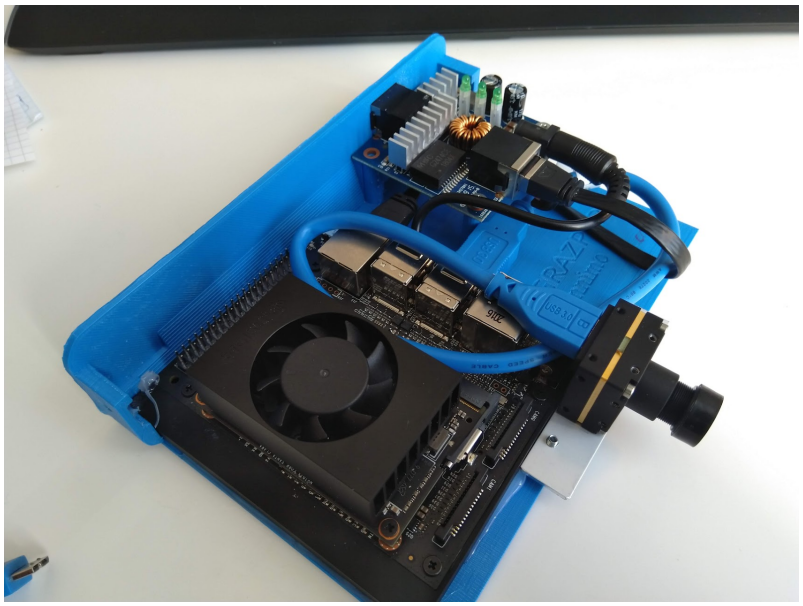


Figure 4.1: Picture of the interior of a Grazper camera. The Nvidia Jetson, USB camera and power supply are visible in the image.

The cameras are calibrated individually with the help of a checkerboard pattern to determine their intrinsic parameters and correct for lens distortion. These calibration parameters are specific to each camera and are not expected to change unless something unexpected happens; only if they receive a hit or the lens moves. Before recording, the cameras are calibrated between them by setting a big checkerboard pattern in the center of the scene, ensuring it can be seen from all angles. The origin of world coordinates is set with the checkerboard pattern, and each camera estimates its extrinsic parameters.

The setup consists of 8 cameras connected through a switch to a central computer, the rig server. Each camera runs a Precision Time Protocol (PTP) [20] daemon that synchronizes its clock with the central computer's clock, which runs the PTP server. A UI application lets the user visualize the cameras' feed, set some camera parameters (such as exposure time and gamma correction), and start the recording process.

Each camera records at 30 fps and saves the raw image frames in RGB format to its local hard disk. Once it has done recording, the central server downloads and stores each sequence of frames.

4.1.2 Soles

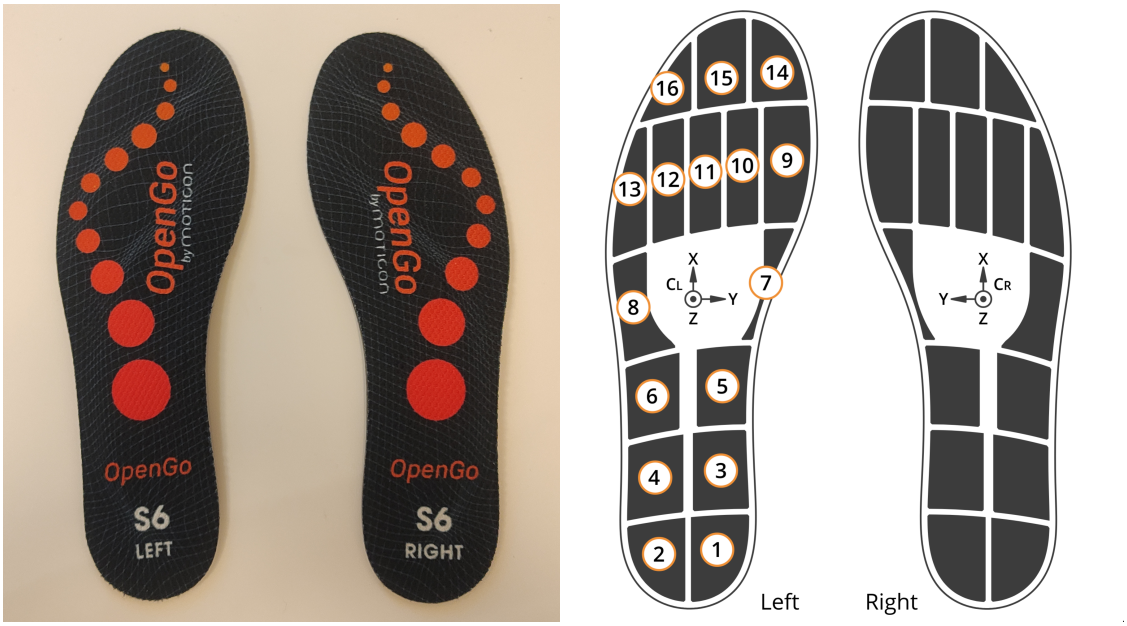


Figure 4.2: Sensor insoles picture and outline with sensor positions.

Grazper acquired a pair of sensor insoles [11] to gather data on foot pressure, a commercial solution that includes 16 pressure sensors and an accelerometer per foot. Figure 4.2 shows the outline of the soles and the sensor positions. Each sole is powered by a rechargeable coin cell and is connected through Bluetooth Low Energy to an Android phone. Moticon provides a phone app, see image 4.3, that can be used to control the sole pair. They record at 100 Hz and can be set to record directly in their local memory. The raw measurements can then be transferred to a computer using the phone app, getting a csv file with all the different channels timestamped.

The recording process is entirely separate from the video process, making it easy to integrate with Grazper's system. However, this separation also introduced some challenges in terms of synchronization, which ended up being tricky to solve. After contacting Moticon, they informed us that the phone app uses the phone's clock for timestamping when

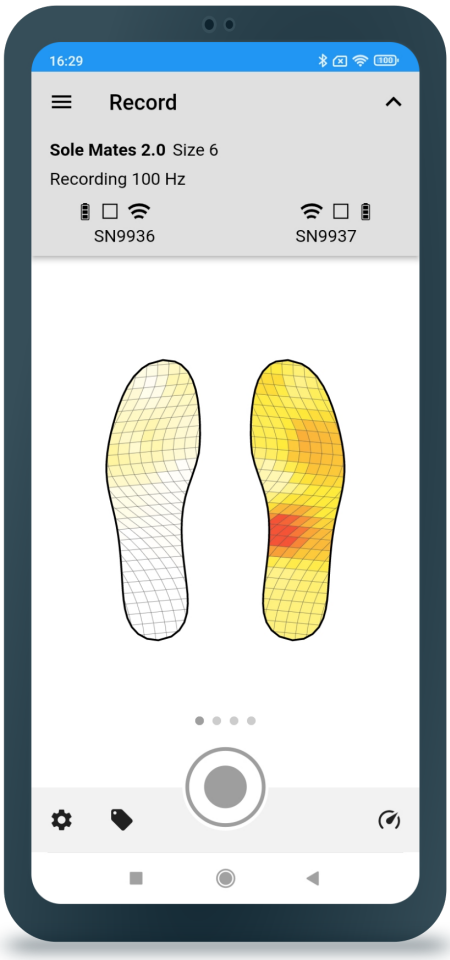


Figure 4.3: Screenshot of the Moticon OpenGo Android app.

the acquisition starts. This was a problem since the phone clock runs independently from the server's clock, and we somehow needed to synchronize them.

Since both the rig server and the phone use the Network Time Protocol (NTP, much less precise than PTP) [21] to set their clocks, our first try was setting the server to sync to the same NTP server as the phone. Recording this way resulted in an offset of 3-4 seconds, with much inconsistency.

In order to use this solution, one would need to synchronize the two feeds manually. We developed a small python app that allowed manually synchronizing the two feeds. The app allows the user to play/pause the video and select any frame in the video using the bottom slider. When paused, the user can move frame by frame. At the same time, the raw soles signal is shown at the bottom. The app lets the user add time offsets to the soles signal until they become synchronized. Then, the user can save the offset to the soles file. A screenshot of the app is shown in image 4.4. Jumping at the beginning of the video and falling on both feet, one had a step-function-like signal on the pressure that could be used to adjust it down to the same frame of error.

However, this solution poses several obvious inconveniences. First, the person needs to jump at the beginning of every taken recording, or the adjustment will be less precise. One must go manually through each video, which poses an inconvenience and an additional source of error. This solution was unacceptable since Grazper intended to incorporate

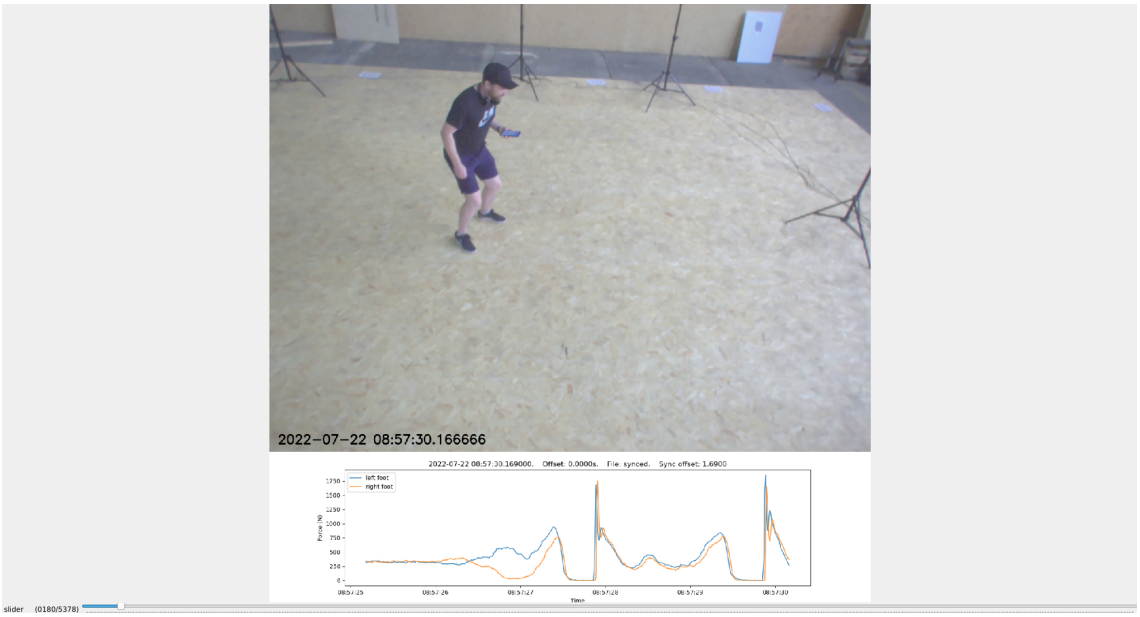


Figure 4.4: Screenshot of the manual synchronization app.

the sole pressure in their recording setup. The synchronization had to be done programmatically.

It turns out that setting the clock manually in an Android phone is a restricted operation because of security reasons, and it cannot be done unless the device is rooted. To this end, Grazper acquired a cheap Xiaomi Redmi 9A, and we proceeded to root it. Once it was rooted, we first tried running a PTP daemon on the phone, just like the cameras, for which we installed Termux, a terminal emulator. However, the network interface on the phone does not support hardware or software timestamping, so PTP was out of the picture. The next attempt was using NTP. We set an NTP server running on the rig server and the phone to synchronize to it through NTP. This way, we brought the offset down to 0.8-1 seconds, but the offset was still pretty inconsistent, which made the solution a bit better than the previous one but still unusable.

We hypothesized that the source of inconsistency was the synchronization being conducted over a WiFi network. In order to address this issue, we decided to attempt synchronization through USB. The Android Debug Bridge (ADB) offers a way to access the Android phone's shell from a computer. Within the Android shell, there is a binary called `date` which, according to its documentation, should allow for time setting with microsecond precision. However, in practice, it could only function with seconds and would produce an error when attempting to use a more precise setting.

However, an environment variable is available in Linux and Android shells, `EPOCHREALTIME`, that provides the system clock with a resolution of microseconds. We discovered that this variable obtains the time from a kernel function, `gettimeofday`, and a corresponding `settimeofday` also exists, which supports microseconds. We developed two straightforward C programs to access and set the phone's time using these functions. The programs return and accept a Unix timestamp with a decimal value. The code of both programs, `get_time.c` and `set_time.c`, are included in the appendix A for reference. It should be noted that the setter function does not include any checks, and the consequences of setting a negative time are unknown. The two programs were copied to the Android phone and compiled using Termux and the GNU C Compiler (gcc) version available through its

package manager.

Once compiled, the executable sits in the phone and can be run from Termux and our computer when connected via ADB. However, we observed a delay when writing the time on the phone directly from the computer. Thus we implemented a bash script that computes an estimate of the round trip time of this operation by setting and reading several times and compensates for the delay, assuming that the operation of reading time is negligible vs. setting it. The bash script is also available to the reader in the appendix A.

We tested this synchronization method experimentally, and all the trials showed that the sole signal consistently lagged by approximately 50 milliseconds. Upon further analysis, it was determined that the phone app introduced the delay during the timestamping of the recording start. However, as the delay was constant in time, the compensation was easy to implement, ultimately allowing for successful synchronization of the signals down to the same frame.

In conclusion, assuming the cameras have already been calibrated one by one for estimating their intrinsic parameters, the final acquisition method consists of 4 steps:

1. Calibrate all cameras to find the set of world coordinates in the checkerboard pattern and estimate each camera's extrinsic parameters.
2. Synchronize the phone's clock to the computer where the PTP server is running. In our case, the same machine is used as a camera recording server.
3. Record several videos using the Moticon app to signal the soles when to start and stop the acquisition.
4. Once done, download the videos to the recording server and dump the soles csv files in the server. This data then gets automatically copied to a central file-storing server so it can be accessed from any computer in the Grazper network.

4.2 Preprocessing

4.2.1 Video

The raw RGB frames are at this step corrected for lens distortion and exported to mp4 using h264 encoding, yielding eight synchronized videos in mp4 format. There is an additional json file that stores metadata of the recording: no. of participants, labels on what tasks they are performing, extrinsic and intrinsic camera parameters, and timestamps of each frame in the recording.

4.2.2 Soles

As for the soles, the raw data has a capture frequency of 100 Hz, far superior to the 30 fps of the video. This number of measurements per second would let us observe changes in pressure with a frequency of up to 50 Hz (Nyquist theorem), which is an order of magnitude above human body movement. Tapping a foot to a metronome set at 300 bpm is as quick as it gets, and this would just be 5 Hz. Also, the measurements do not correspond to any specific video frame since they do not happen simultaneously.

For this reason, we decided to undersample the soles signal. For each video frame, we interpolated the value of each channel linearly from the two closest real soles samples. We initially tried just picking the nearest value. However, we observed that, for very rapid events like kicking something or stomping and raising a foot very fast, one could be unlucky with the interpolation and end up with these events disappearing from the final signal. In these edge cases, linear interpolation would at least detect some pressure, and

thus we decided to use this method in the final pipeline. After undersampling, the data is saved in a json file.

The Mediapipe skeleton we use for pose detection (see chapter 3) has two keypoints per foot, one in the heel and one in the toe. So, to get an estimate of the pressure on each keypoint we take a mean of readings: the mean of sensors 1 to 8 will account for the heel, and the mean of sensors 7 to 16 will account for the toe, see figure 4.2.

Finally, since this pressure estimate depends on the weight of the wearer, we needed to normalize the signal. We observed that through our captured recordings, the pressure distribution on the toes differed from the one on the heels. The toe channels consistently received more pressure than the heel ones. Thus, we decided to standardize them separately.

We take both left and right toe channels and sum them. Then, we find the mean pressure along the entire recording and half it. We finally normalize by dividing each channel by this value. We normalize the heel channels in the same way. By normalizing this way, a value of 1 roughly corresponds to a quarter of the pressure when the person is resting.

5 Experiment pipeline

We will now discuss the pipeline we designed to conduct the experiments in this thesis. Our purpose is to show that incorporating knowledge about foot contact can improve the 3D pose estimation when fewer cameras are involved. As mentioned in the introduction, knowledge about foot contact unlocks the possibility of performing the task from only one camera angle and improving the several-camera case.

In the following sections, we will show the different components of the pipeline and how they work together. They are designed in a modular way that allows changing them quickly. Figure 5.1 shows a diagram of the experiment pipeline.

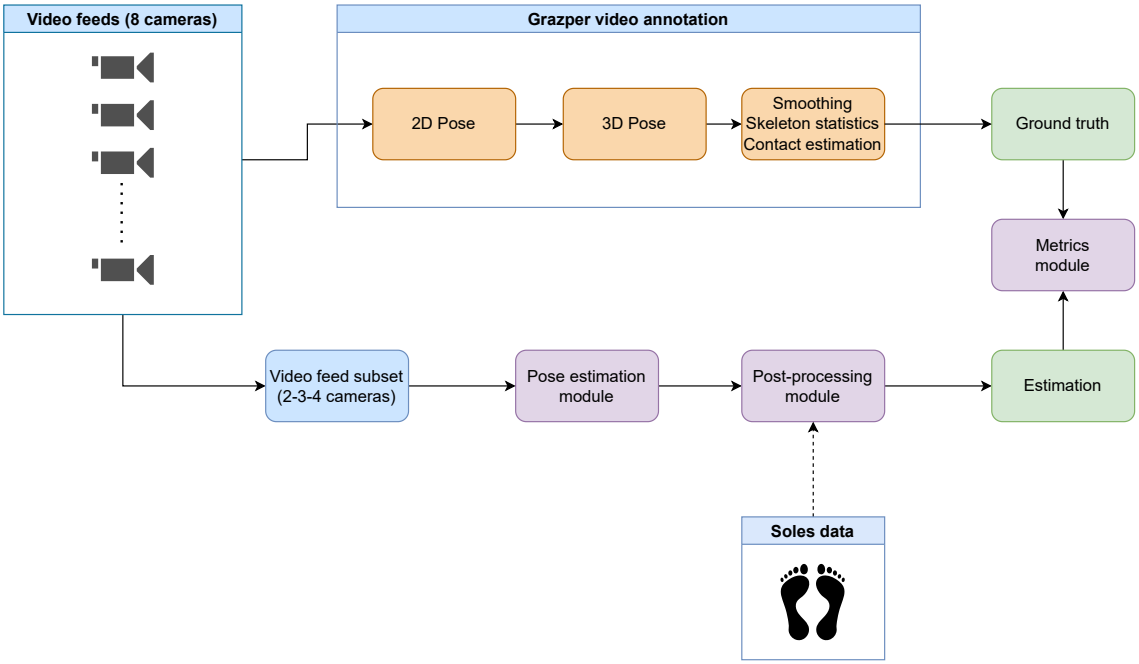


Figure 5.1: Diagram of the experiment pipeline building blocks

5.1 Ground truth annotation

First, we need some annotations on the videos. We need a 3D skeleton that acts as ground truth to have reference points to compare our proposed methods' performance. We will use Grazper's annotation tool (figure 5.2) to achieve it. They use this tool internally to annotate all their recordings and train their own 3D pose estimation models. Grazper developed this solution, so we will describe it shortly without providing much detail since it is out of the scope of this thesis.

The tool extracts a 3D pose as described in chapter 3, computing a 2D pose estimation in each camera view and then triangulating a 3D skeleton using the extrinsic camera parameters. Since the triangulation is done frame by frame, the resulting skeleton is often tremulous. For this reason, they smooth the skeleton in the temporal domain using a Kalman filter. Later, they perform some skeleton statistics, calculate the bones' length and estimate contact points based on the velocity and position of specific keypoints, and recalculate the skeleton enforcing these constraints.

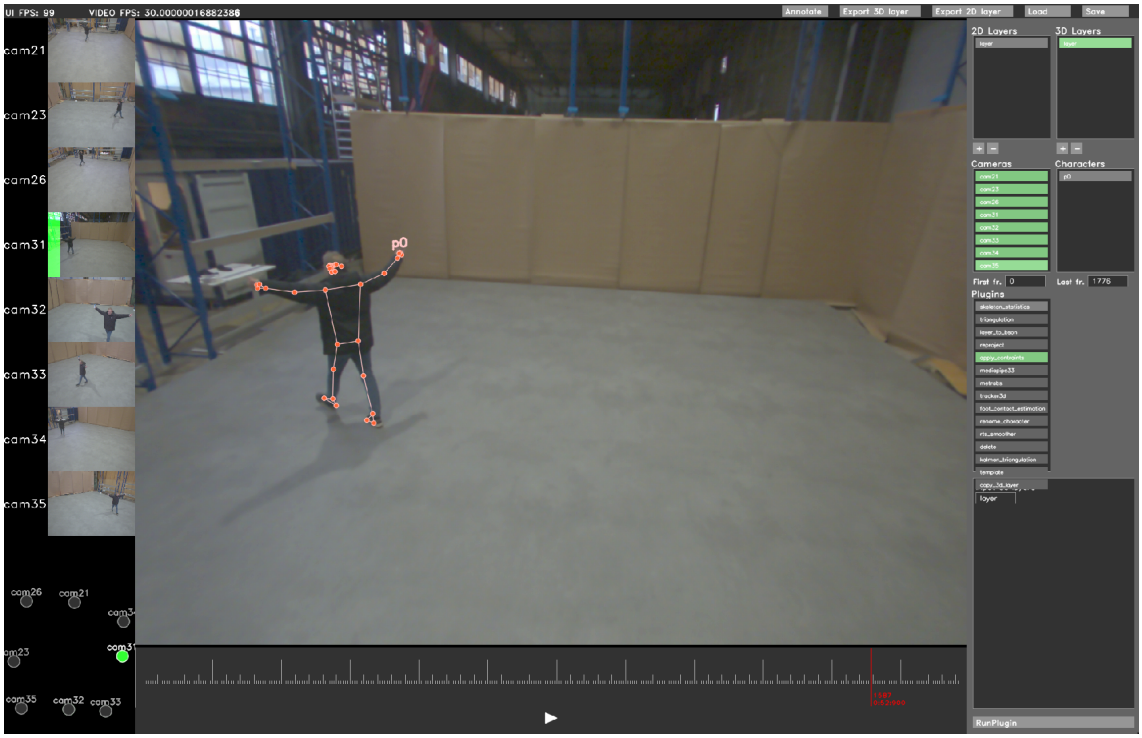


Figure 5.2: Screenshot of Grazper's annotation tool.

The result is a skeleton that looks much more natural and realistic, with smooth movements. Also, as the 3D pose is estimated from 8 different camera angles, the location of the keypoints is very accurate, and there are hardly any occlusions, mainly when dealing with only one person in the scene. Grazper's annotation tool also allows manually annotating particular frames when the automatic annotation fails.

Next, we will describe the different building blocks of the experiment pipeline in detail. It is worth mentioning that every output of every building block has been designed to be compatible with Grazper's video annotation tool. This allows us to visualize every output easily and has been of great help when building it and interpreting the results.

5.2 Pose estimation module

This block selects a subset of cameras and runs them through the process described in 3. However, due to the reduced number of cameras, occlusions are to be expected, and in some frames, a particular keypoint may only be visible from one camera angle. This may result in incomplete 3D skeletons since our triangulation script requires each keypoint to be visible with a certain level of confidence from at least two cameras.

The output of this block will be a sequence of 3D skeletons, one per frame, with the 3D position + a confidence mask (1 if could triangulate, 0 if not) for each keypoint, represented in a 2D matrix of 33 by 4.

5.3 Postprocessing module

The postprocessing module takes the base 3D skeleton from the pose estimation module and modifies it, incorporating knowledge from foot contact, either by taking the captured sole signal or estimating it. We will discuss how we estimate the pressure signal later in chapter 6. Let us focus on how we use the signal to modify the 3D skeleton.

We built two different modules; PostProcessor Offline (PP Offline), meant to process the skeletons having access to the whole recording, and PP Online, to only look back in time so that it could potentially be used in online applications.

5.3.1 PP Offline

This module takes the entire soles sequence, the value of pressure at each foot's heel and toe, and thresholds it by a parameter to binary classify the signal, whether there is contact or not. After, we find the different sequences of continuous contact of each of the keypoints in the video.

Based on the assumption that feet do not move when there is contact, we take each sequence of contact of every keypoint and compute the average 3D position of the keypoint during the entire sequence. We then replace the position of said keypoint with the average position in every frame in the sequence. Keypoints that could not be triangulated are not included in this calculation.

It is easy to imagine edge cases where this postprocessing method would break: videos where a person is sliding or on a skateboard. Those cases have foot contact, but the feet do move. However, the module could be expanded to account for them: one could monitor the velocity of the keypoint and only act at points where the velocity is close to 0. We tried this approach but encountered another issue. As the triangulation in the pose estimation is done frame by frame, the 3D skeleton may flicker, leading to inaccurate velocity estimates and compromising the entire postprocessing. A solution to this problem would be to smooth the skeleton, similar to the way Grazper does for annotating data. However, as we deemed the implementation of Kalman filters as out of the scope of this thesis, we decided to leave the postprocessing method as it is.

5.3.2 PP Online

This module works the same way as PP Offline. We first threshold the signal and find the contact sequences for each keypoint. Then go through all the sequences and compute the mean position of the keypoint in the sequence. However, in this case, we only take its mean by looking backward in time. Hence, the position of the keypoint will not be constant throughout the sequence.

This post-processor could be set up to work online. Of course, in this experiment, we are just recreating it. The online setup would only make sense when estimating the pressure value if we also use an online model for doing so. Pressure prediction will be discussed in more detail in chapter 6.

Notice that both post-processors only affect the position of the 4 keypoints in the feet. We have restricted ourselves to this simple case because of time constraints. However, the potential of this approach can be much more significant. Combining these post-processors with some knowledge about the length of bones, one could also recalculate the position of the whole lower part of the body and make it less flickering, yielding a more realistic output in the whole skeleton.

5.4 Metrics module

This last module is where the 3D skeleton result of the postprocessing is compared against the ground truth 3D skeleton. We have used several measures throughout the development of this thesis. Some might not be that interesting from a results point of view, but they have helped in the debugging process. These measures include:

- Bias - Variance: Avg. absolute difference of the distance from the predicted skeleton to the ground truth annotation over time and per keypoint. Also, the variance of this metric.
- Root Mean Squared Error (RMSE): the squared root of the mean squared difference of the predicted and the true skeletons.
- Percentage of Correct Keypoints (PCK): detected keypoints are marked as correct if the distance between the predicted and true joint is within a particular selected threshold. We take the average over the number of frames for each keypoint.
- Cloud points: A facet grid of 3x3 plots, where we visualize the distance vector between the ground truth and the predicted skeleton on the different combinations of the x, y, and z-axis.

6 Sole pressure estimation

Sole pressure estimation constitutes the critical element of this Master's thesis. If we cannot predict it, the gains brought with foot contact knowledge will be relegated to the recording case, where the person in the scene is wearing the sensor insoles. In this chapter, we describe our approach to preparing the data, designing the models, and training them.

6.1 Input data

Our model should be able to take information from the scene and predict whether there is foot contact or not. There are several options for what the input data should look like.

One approach could be using the video frames as input, designing the model first to detect the person in the image and determine whether the person is touching the ground. If a person were to tackle this problem, having this kind of input would be the easiest. Even in cases where the perspective might produce some confusion, one can use additional information like the hair's position or the clothes' movement to determine the contact. However, one would need an extensive annotated dataset, with variability on the person performing the task, the clothes they are wearing, the perspective of the cameras, and the scenario for the trained model to be generalizable to any other scene. This approach was not feasible, as we didn't have the time or means to create such a dataset in this thesis' span.

Using the skeleton output from the pose estimation is a better option since it is more independent of these details. The skeleton features will still vary from person to person, but these differences frame a much easier problem for artificial intelligence to learn than the previous one. One could use the 2D pose output from Mediapipe and the camera parameters as input to the model. This solution would also be useful in the one-camera case.

Since we are limiting ourselves to the several-camera case in this thesis, we can use the triangulated 3D pose in world coordinates directly. This has the added benefit that the same frame can be triangulated from different subsets of cameras, which would, in essence, be the same pose. However, this yields a slightly different skeleton, with different joints missing due to occlusion. This sort of jitter in the dataset benefits the model to learn [22], making our model more resilient to missing joints. Since these triangulations are highly correlated, it is crucial to ensure they do not end in both the training and validation set. This will be addressed again in section 6.4, where we discuss the training setup.

We decided to work with a subset of four cameras out of the eight we have available. This gives us a total of $\binom{8}{4} = 70$ different 3D skeletons from a single video. We decided to split the video in sequences of `seq_length` number of frames, a hyperparameter that will need tuning in the training phase. Later, we normalize the input by taking the mean position over frames of the skeleton's "center of mass", the mean position of all 33 keypoints. This way, we normalize the input around 0 but still keep the relative movements during that sequence, which are needed to predict sole contact. Figure 6.1 shows 4 frames out of a normalized sequence of 30, with a skeleton walking.

6.2 Models

We propose two model architectures to tackle this task. A simple model meant to act as a baseline, and a more modern architecture, suited to work with time-series data. It is important to note that details such as the no. of hidden layers and their sizes have changed during the experimentation and tables 6.1 and 6.2 only showcase our final choices.

6.2.1 Baseline

A feed-forward neural network as described in [23]. It simultaneously takes the entire flattened sequence of frames, a vector of $\texttt{seq\_length} \times 33 \times 4$. We use six fully connected hidden layers, diminishing in size. We use a Rectified Linear Unit (ReLU) as an activation function [24], except in the output where we use a hyperbolic tangent. We use this to match the other model's output, described in the following subsection. Table 6.1 shows the model's architecture in more detail.

Layer	Type	Size
Input		`seq_length` x 33 x 4
1	Fully-connected ReLU	2048
2	Fully-connected ReLU	1024
3	Fully-connected ReLU	512
4	Fully-connected ReLU	256
5	Fully-connected ReLU	128
6	Fully-connected tanh	`seq_length` x 4
Output		`seq_length` x 4

Table 6.1: Baseline model architecture

6.2.2 SoleNet

Given the temporal nature of our dataset, we decided to use a Recurrent Neural Network (RNN) as the model for sole pressure estimation, more specifically, an architecture based on the unidirectional Gated Recurrent Unit (GRU) [25]. The GRU has a gating mechanism that helps the network remember temporal dependencies by reusing its previous outputs in the computation of a new one. In each forward pass, the GRU computes the following operation:

$$
\begin{aligned}
z_t &= \sigma\left(W^{(z)}x_t + U^{(z)}h_{t-1}\right), \\
r_t &= \sigma\left(W^{(r)}x_t + U^{(r)}h_{t-1}\right), \\
h'_t &= tanh\left(W^{(h)}x_t + r_t \odot U^{(h)}h_{t-1}\right), \\
h_t &= z_t \odot h_{t-1} + (1 - z_t) \odot h'_t.
\end{aligned}
$$

x_t denotes the layer input at a given time t, and h_t the output, also referred to as hidden state. They can have different sizes, and thus `hidden_size` is a design choice. z_t and r_t denote the update and the reset gates, respectively, both vectors of length `hidden_size`. W denotes three different matrices of weights for the input, with size $\texttt{hidden\_size} \times \texttt{input\_size}$,

while U denotes three square matrices of size `hidden_size`×`hidden_size`. σ denotes the sigmoid function and $\odot$ the Hadamard product, commonly referred to as the element-wise product. Each new output h_t is calculated with the input x_t and the previous output h_{t-1}, which allows the network to learn temporal patterns. It is worth remarking that the output of the GRU is a hyperbolic tangent, which produces an output between -1 and 1.

Our proposed architecture consists of several layers of GRUs, also diminishing in size. Each frame is fed one by one as an input, and each layer will use its previous output to calculate the new output. The architecture is described in table 6.2.

Layer	Type	Size
Input		33 x 4
1	GRU	1024
2	GRU	256
3	GRU	64
4	GRU	16
4	GRU	4
Output		4

Table 6.2: Solenet model architecture

6.3 Labels

After the process described in section 4.2.2, we obtain a reading of average pressure in the left heel, left toe, right heel, and right toe, matching the video frames and normalizing so they are independent of the weight of the person. A reading of 1 roughly corresponds to each channel's pressure in a resting standing pose. Already by walking, there are moments where most of the weight is applied to one of the channels, yielding a value of 3 or 4 on this scale. When jumping and falling on one foot, this value can even reach 10. In the previous section, we have seen that both proposed models have an output restricted to the $[-1, 1]$ domain, so some modification must be taken before we can use them.

One possible approach is modifying the network, by changing the last activation function to be something different than a hyperbolic tangent, for example, a ReLU. This step is easy to do in the baseline model, but on the GRU, that change compromises the whole structure of the last layer and would hurt training. Another possibility is adding a simple linear layer to let the model learn the scaling.

However, since we will use this model to predict whether there is contact, we are not so interested in if the value of the pressure is 5 or 10 since we are going to classify both in the same group after thresholding. Nevertheless, we are interested in the first moments of contact, where the signal is smaller than 1, and the transition between contact and no contact takes place. For this reason, applying the hyperbolic tangent to the pressure signal made sense, saturating the moments with no contact and contact to -1 and 1, respectively, and only focusing on the area of interest between these two. That is the approach we decided to take, scaling the signal by two constants a and b that were found experimentally trying different values and inspecting the resulting signals:

$$y'_t = tanh\left(a \cdot y_t + b\right), \quad \forall t \in [0, \texttt{seq\_length}],$$

where $a = 3.5$ and $b = -2.5$. A sample of the resulting signals can be seen in figure 6.1, where the values saturate at -1 and 1, but keep the transients when the balance changes from the heel to the toe.

6.4 Training

We split the data into three sets for training and validating the models: training, validation and test. We set aside one specific video for the test set, use the rest of the recordings for training and validation, and split them, leaving 70% of the chunks in the training set. At this step, we ensure that all the different camera triangulations of a sequence fall into the same set. Otherwise, we would have sequences in the validation set practically equal to others the model was trained with, perverting any validation metric. We decided to use Mean Squared Error as the loss function and train the models using Adam [26], a gradient-based optimization algorithm, and train both models for several epochs, feeding the input sequences in batches. The hyperparameters that can be set are thus `seq_length`, `no_epochs`, `batch_size` and the initial `learning_rate`.

Halfway through the experiment phase, we decided to add data augmentation in our training setup to prevent overfitting. Following the advice in [22] and [27], we created an augmentor module that, if enabled, takes the sequence before it is fed to the model in the training loop and rotates it by a random amount around the Y axes and adds some random jitter to every joint in the skeleton in every frame. These random translation variances are also hyperparameters.

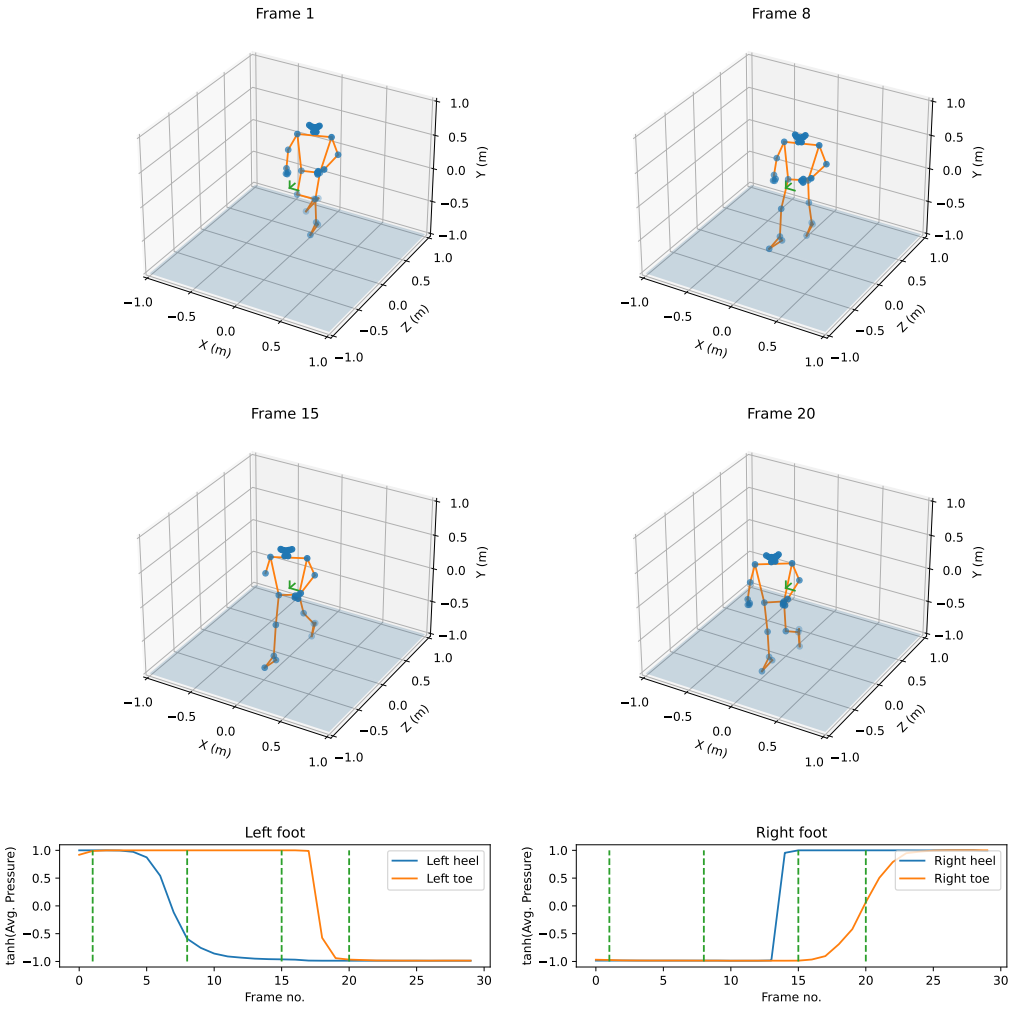


Figure 6.1: Frames of a sequence of 30 frames extracted from the dataset, depicting a walking skeleton, together with the corresponding pressure labels.

7 Experiments and results

In this chapter, we will describe the experiments carried over the development of this thesis. The methodology has been described in chapters 4, 5 and 6 and presented with a clear separation of the different tasks and the different modules that tackle each task.

It is worth mentioning that, in reality, the progress has been more convoluted, and all three parts have been growing simultaneously, making it hard to describe the experiments in the order they were conducted. Data acquisition was the starting point, but we started working on the experiment pipeline from the first moment we got some synchronized data. It was only when the synchronization problems in the data acquisition were solved that we started working on pressure estimation, at which point the experiment pipeline was pretty advanced. However, we could not conduct the final experiments without the trained models, so we finished that part before running the final experiments.

Instead, we will present each part, focusing on making them clear to the reader, explaining the purpose of each step and the obtained results. First, we will focus on data acquisition, then sole pressure estimation, and finally on the experiment pipeline results.

Figure 7.1: Picture of the unfinished wooden floor in Refshaløen

7.1 Data acquisition

The different challenges presented regarding synchronization and the final working state of the data acquisition pipeline have been discussed thoroughly in chapter 4. However, we have not addressed all the problems of different natures we encountered in this phase. It is often overlooked how complicated it can be to acquire a quality dataset, especially at a time when we can find high-quality, curated datasets ready to be used in a csv file from websites like Kaggle.

As mentioned in the chapter, Grazper's idea was to build a recording rig in Refshaløen, Copenhagen, from scratch, starting in an empty industrial warehouse to record their

datasets. Many problems arose, such as building a flat 10x10 meter wooden floor, scaffolding around the stage to hold the cameras and lights, setting up the whole network, recording server, sole integration, delays in material deliveries, problems with leaking pipes, figuring out with lawyers the simplest legal way to be able to have someone recorded in datasets for commercial purposes, and a long list of etceteras. All these inconveniences eliminated the possibility of bringing people in to record at the time of the thesis. So, we had to resort to recording with people from Grazper whenever possible. Late November and December 2022 were two months when the temperatures in Copenhagen dropped a lot, reaching -10ºC, which also made recording difficult.

To top this up, the first sensor insoles Moticon provided were faulty and broke before we could take the recordings needed for the thesis. Almost one-third of the pressure sensors stopped working. Figure 7.2 shows a screenshot where the faulty sensors are highlighted. This compromised the method for obtaining one channel of average pressure per heel and toe described in chapter 4, especially in the right foot. We changed the computation to use only the working sensors.

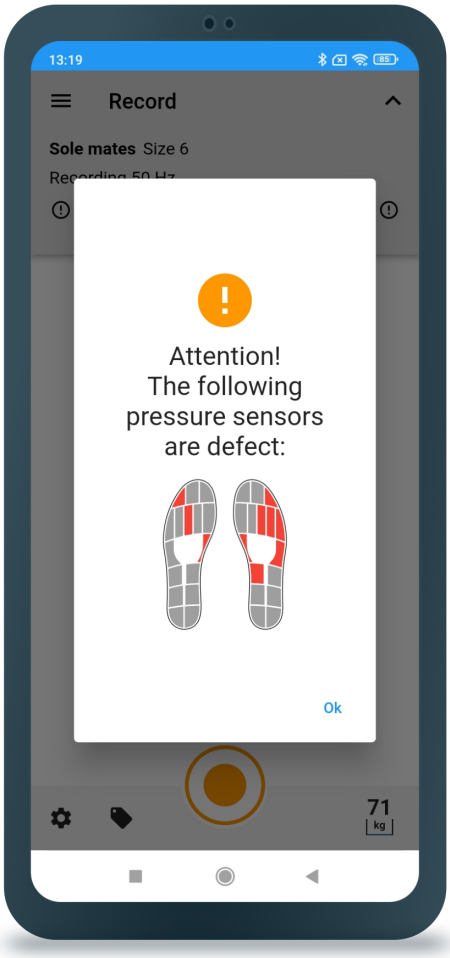


Figure 7.2: Screenshot of the Moticon OpenGo App, highlighting the faulty sensors in red.

The final dataset consists of seven videos, one and a half to two minutes long each, recorded from 8 synchronized cameras and with a person wearing the sensor insoles. Six of these recordings were made with faulty soles. In total, we recorded three people performing acts like walking, running, sitting in a chair, or even doing some acrobatics.

7.2 Sole pressure prediction

Now we will describe the process of training the models for sole prediction. The details of how we treat the recordings in order to obtain the dataset we will use for training have already been discussed in chapter 6. The final dataset consists of sequences of `seq_length` frames from a total of 6 different recordings, with a normalised 3D skeleton on them, and the pressure signal for each keypoint in the feet.

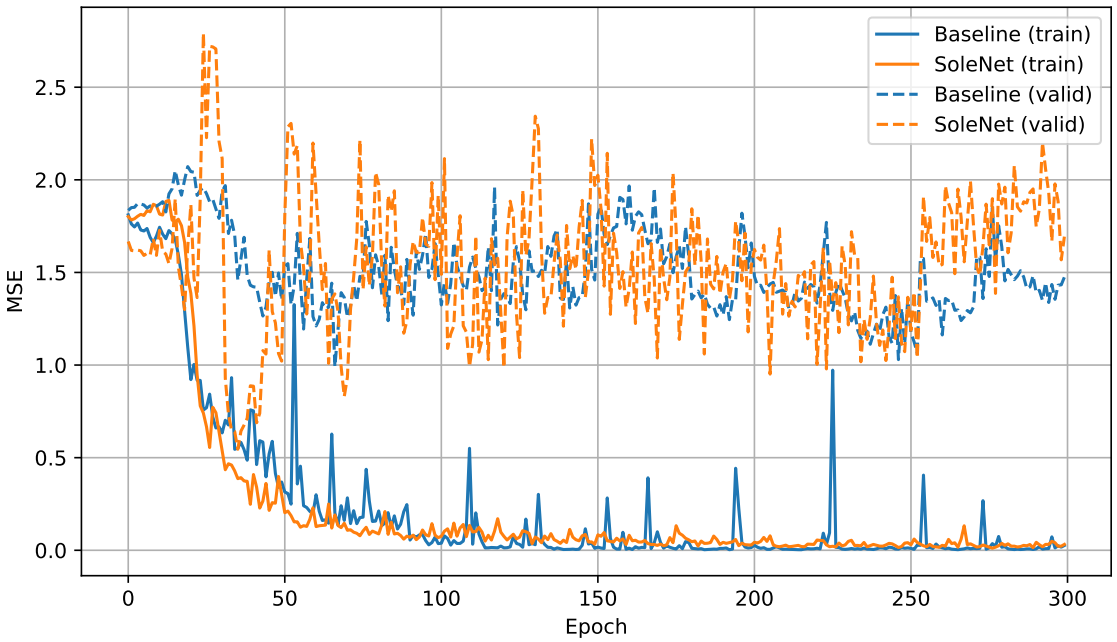
7.2.1 Simple dataset

We split the dataset in two, training and validation, keeping 70% of the sequences in the training set. We keep a separate recording as a test set, in order to test the final model with sequences that are as uncorrelated as possible, and to help us visualize the behaviour of our network in a full recording.

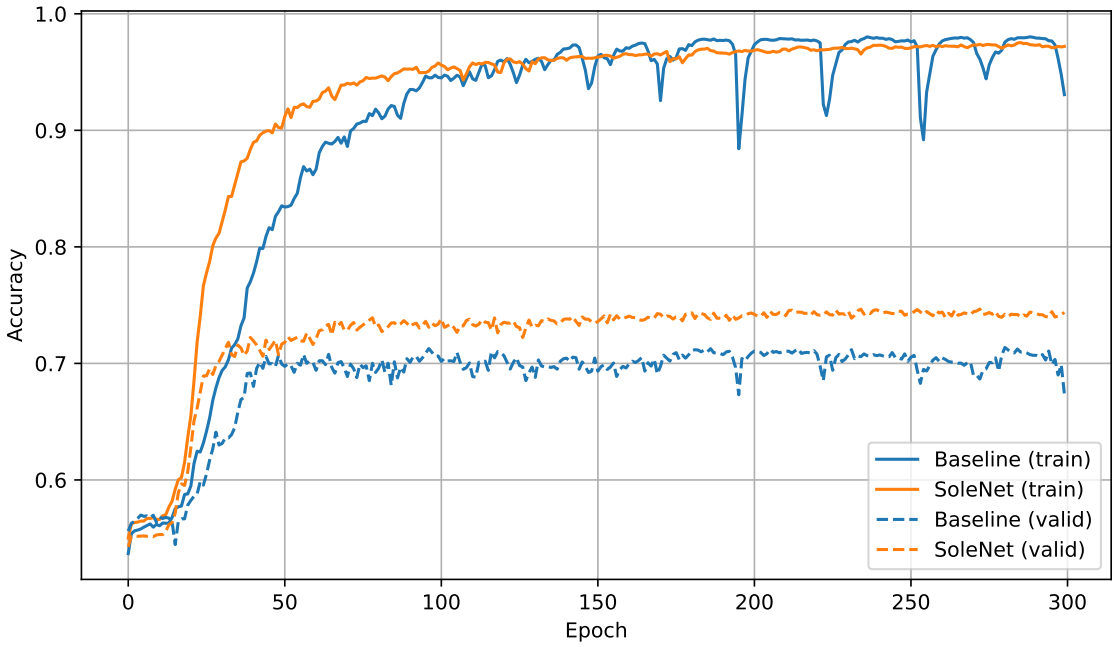
It is important to remark that we tried out different model complexities, some that would not learn for being too simple, some that would overfit very easily, until we found a good compromise with the model architectures described in tables 6.1 and 6.2.

We pick MSE as our loss function, and we train the networks for 300 epochs, passing through the entire training set in each epoch. After each training epoch, we check on the validation set, to monitor overfitting. After trying out different hyperparameters, we picked a `seq_length` of 30 frames, a `batch_size` of 256 and an initial `learning_rate` of 10^{-3}. We also monitor the accuracy, by using a threshold on both predicted and real labels we chose to be 0. At every step of training, we changed this hyperparameters arounds in search of a better option, but we consistently found they were a pretty good choice and kept coming back to them.

Figure 7.3 shows the loss and accuracy evolution for both training and validation for both models. Looking at the training loss curve, it is clear that both models are learning. The validation loss is however not going down at all, indicating that the model is overfitting. Weirdly enough, this doesn't have a negative effect in validation accuracy, just seems to stagnate. We suspect this behaviour is caused by the low amount of data we are working with, and how correlated it is. So, we decided to implement some augmentations in the training data.



(a) MSE Loss



(b) Accuracy

Figure 7.3: Loss and accuracy during training on the simple dataset.

7.2.2 Augmented data

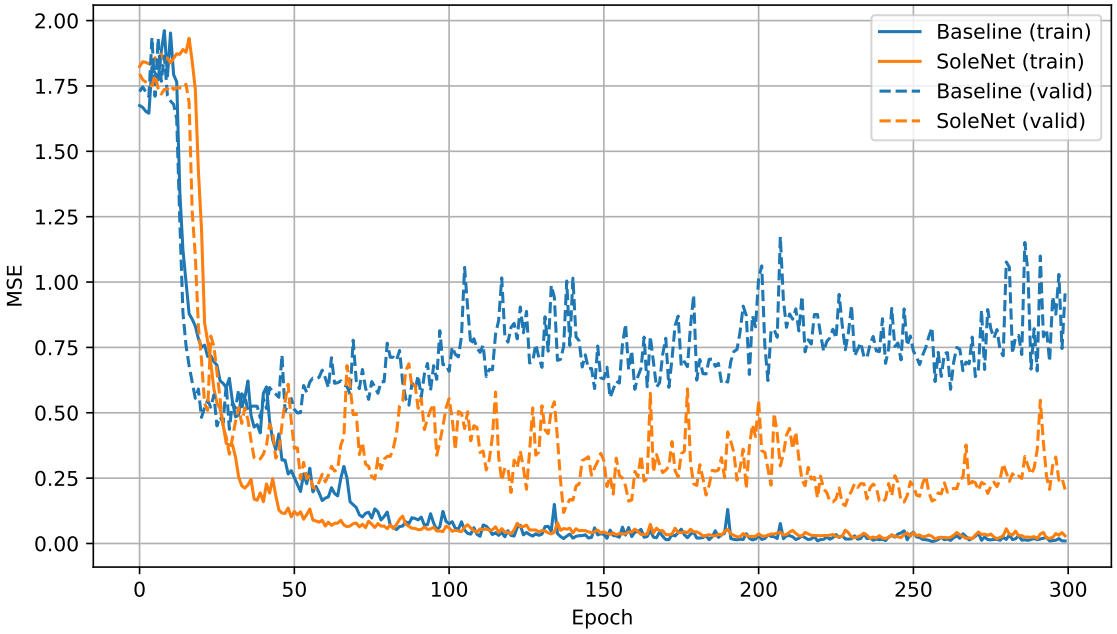
The method for performing data augmentation is explained in section 6.4. We perform rotations by a random angle of the sequences and add jitter to each keypoint in every frame before feeding them in to the model in the training phase. The loss and accuracy curves during training on the augmented data are shown in figure 7.4.

The impact of data augmentation is clear when looking at the validation loss, reinforcing our previous hypothesis: the current model architectures could learn much more if we had more data. Training for more than 300 epochs would again cause overfitting. So, we tried a last attempt to prevent this overfitting, by adding L2 regularization.

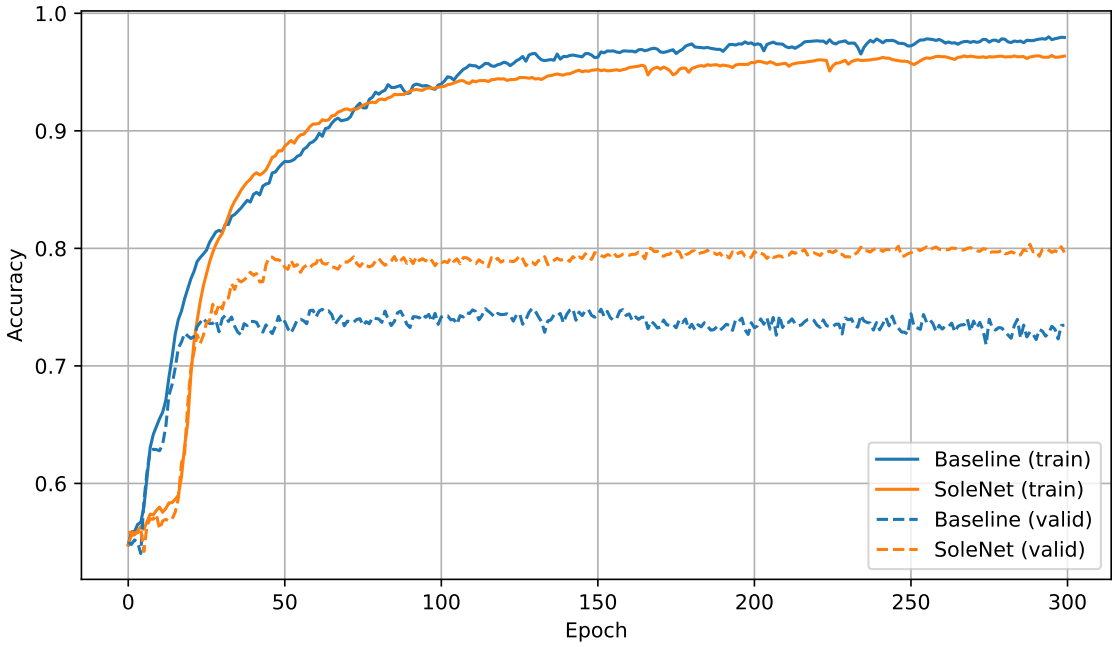
7.2.3 Augmented data + L2 weight regularization

We added L2 regularization on the network weights. We found it would work best with a parameter of 10^{-5} for the baseline, and $5 \cdot 10^{-6}$ for SoleNet. The results are shown in figure 7.5.

Even though it may seem, by the validation loss curve, that the models are not learning correctly, the accuracy tells a different story. The regularized solenet performs better than the previous one, while the baseline remains being practically the same.

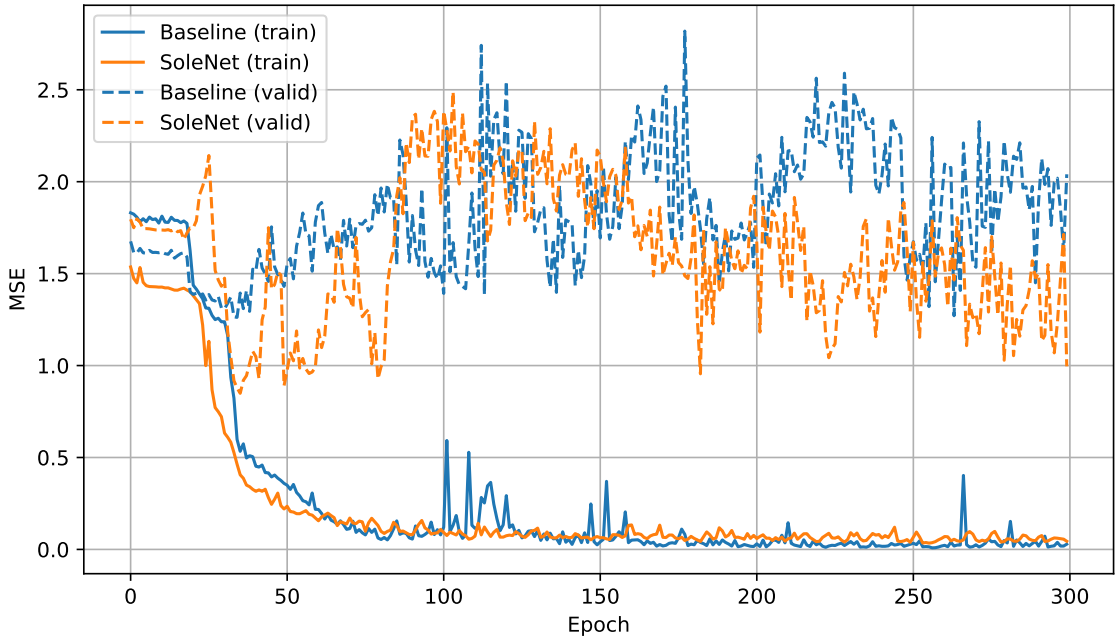


(a) MSE Loss

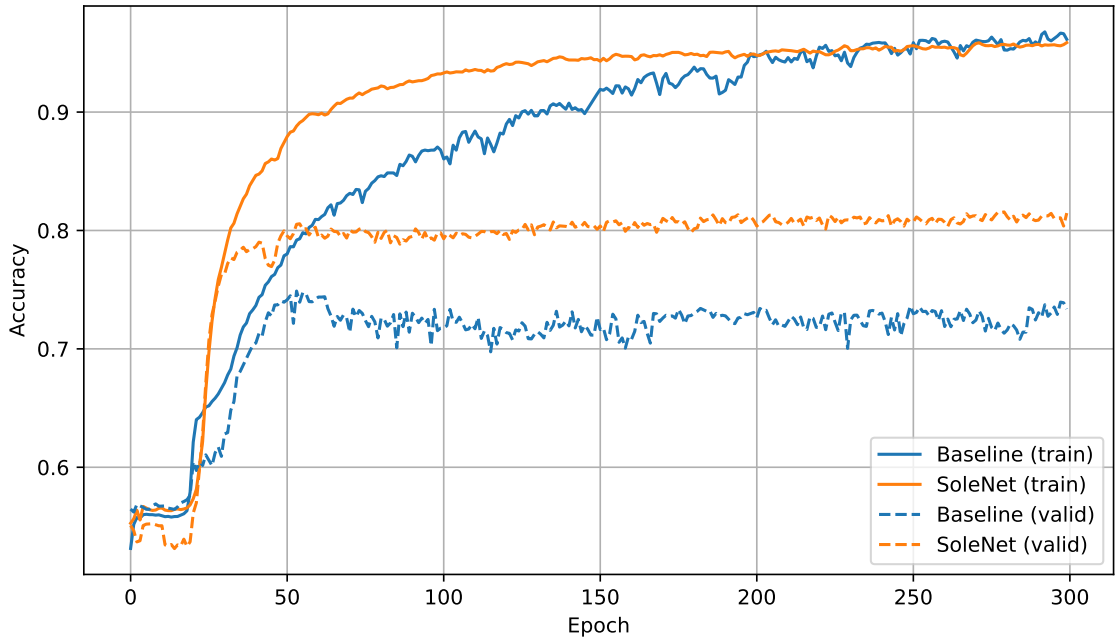


(b) Accuracy

Figure 7.4: Loss and accuracy over epochs during training with data augmentation.



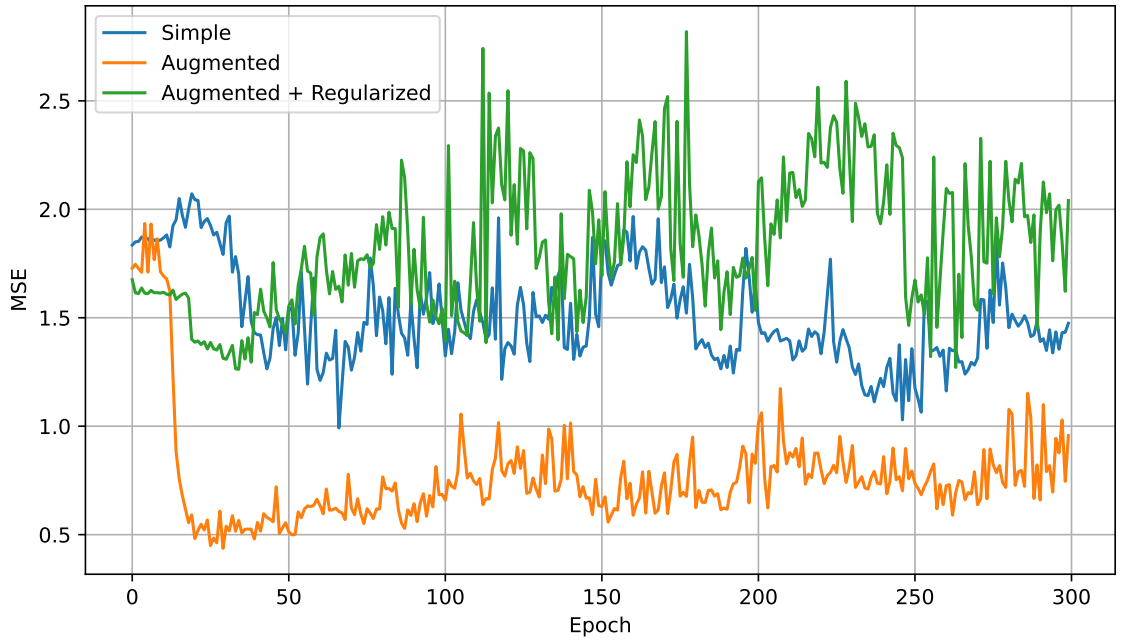
(a) MSE Loss



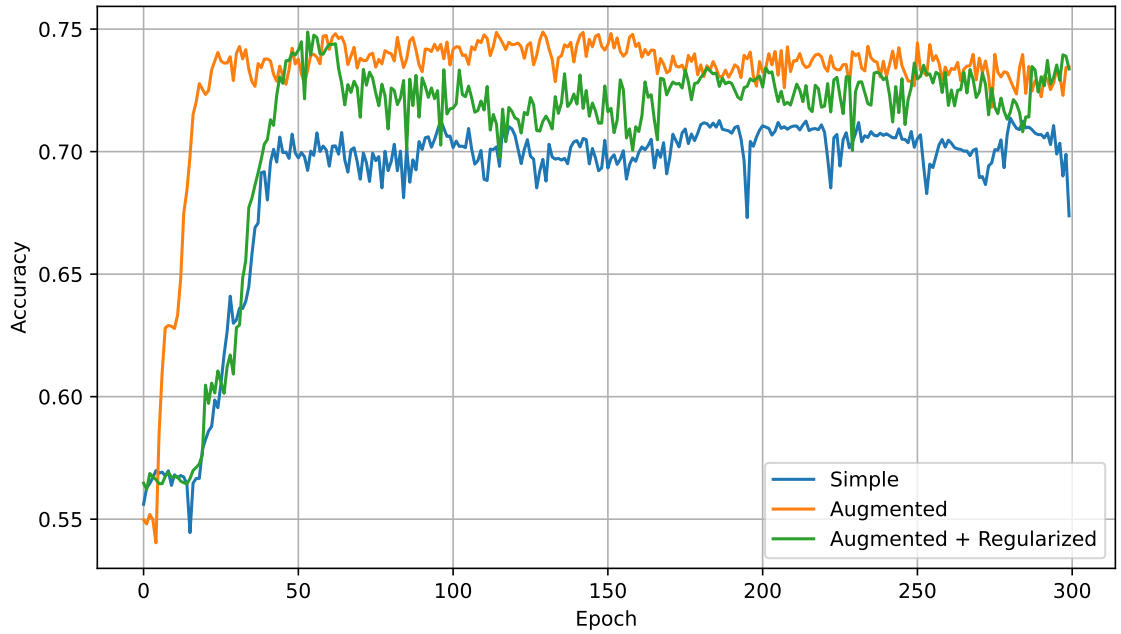
(b) Accuracy

Figure 7.5: Loss and accuracy over epochs adding L2 regularization.

For easiness of comparison, we'll add two more figures, comparing validation loss and accuracy for the different methods of training. Figure 7.6 shows the comparison for the baseline model, while figure 7.7 shows the comparison for the SoleNet.

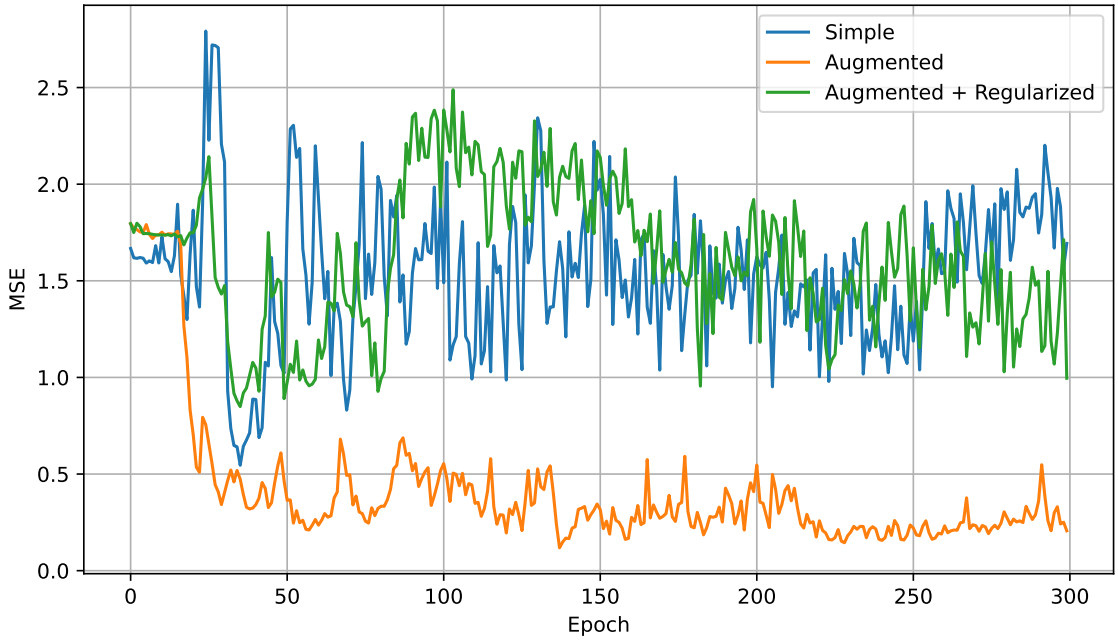


(a) MSE Loss

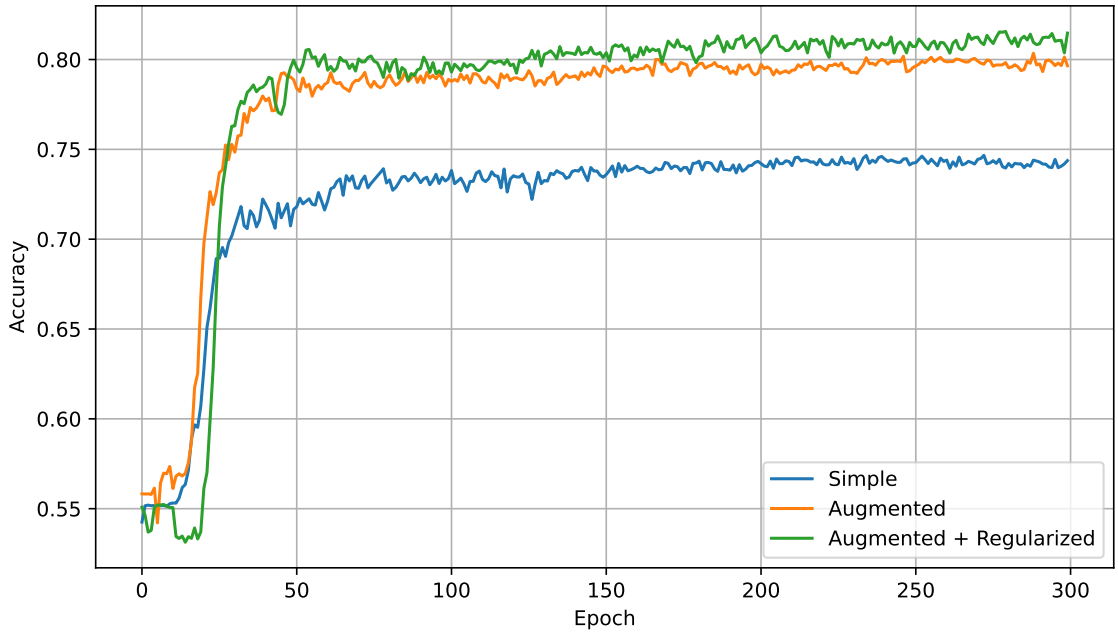


(b) Accuracy

Figure 7.6: Validation loss and accuracy over epochs for the baseline model on the different runs.



(a) MSE Loss



(b) Accuracy

Figure 7.7: Validation loss and accuracy over epochs for the SoleNet model on the different runs.

In both figure 7.6 and figure 7.7, we can see that we get the most positive effect from augmenting the data. Weight regularization helps a bit in solenet, but not that much on the baseline. We strongly think this a sign we would need more data.

To finalize, we run the trained models from the three different methods on the test set, yielding the final results shown in table 7.1. Overall, we can affirm that both models are, in fact, learning to predict the sole pressure values from the pose. However, the values of the loss are an order of magnitude higher than the ones we were obtaining on the validation set. This indicates that the training and validation sets, even though they contain different sequences, are not that uncorrelated within them. This makes sense, they come from the same video and have the same person on them.

Network	Training method	Loss (MSE)	Accuracy (%)
Baseline	Simple	32.1	66.6
	Augmented	26.6	70.83
	Augmented + regularized	**26.15**	**73.6**
SoleNet	Simple	29.06	71.77
	Augmented	23.15	75.15
	Augmented + regularized	**21.76**	**76.52**

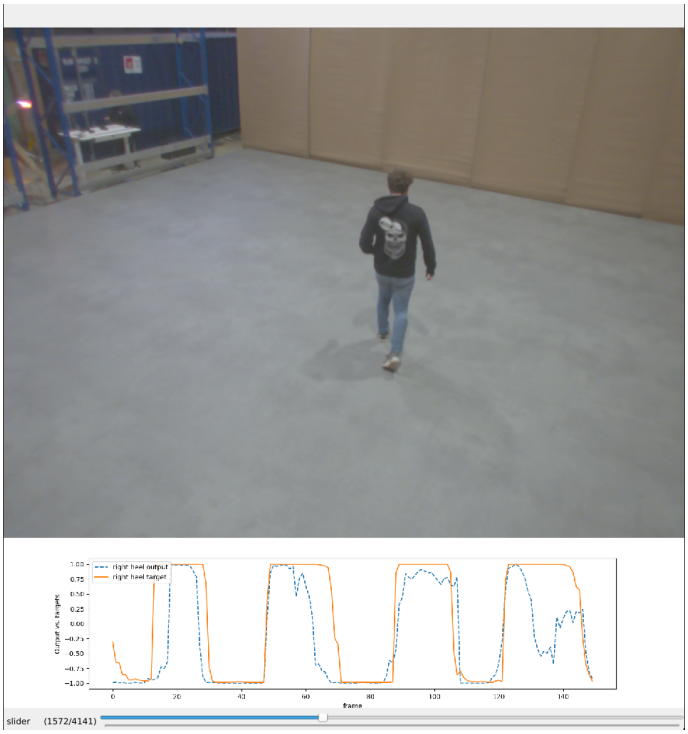
Table 7.1: Test results of the trained models.

Let us now see how the output of both models actually looks like on the test set. Figures 7.8 and 7.9 show the predicted output vs the real pressure of the right heel pressure channel of both models. The first figure showcases a person walking, while the other one a person getting up from sitting in the ground.

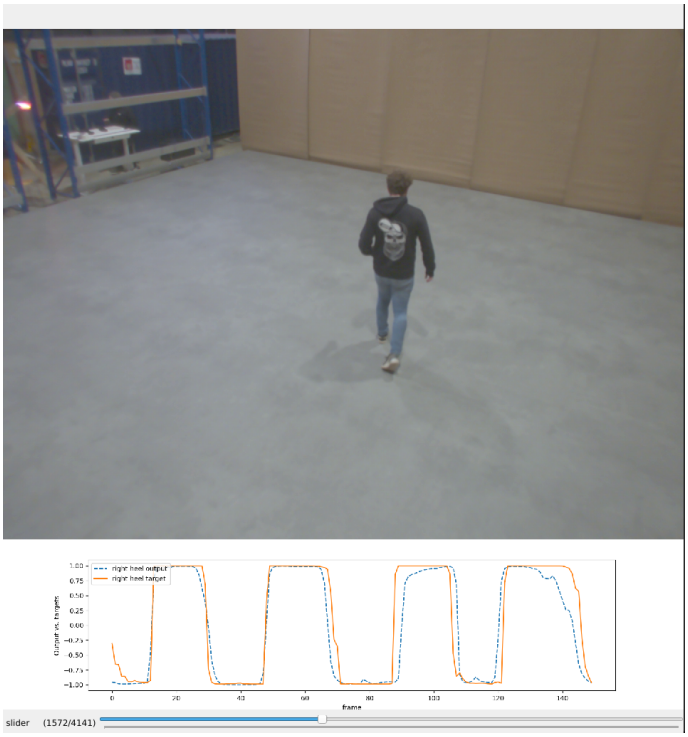
Looking at figure 7.8, it's clear that the model has learnt successfully the task of walking, besides some weird artifacts in the baseline model that could be caused by the way we are feeding the data into the model. The output from SoleNet is remarkably close to the real signal with a smooth output, which we attribute to the GRU architecture.

However, figure 7.9 tells us a different story. When the model is fed a task it hasn't seen before, like someone standing, the output is just as bad as random noise, evidence of our previous suspicion: our dataset is too narrow, and we would need a bigger pool of different tasks recorded in order to obtain a model that could be used in a real-life application. One can even notice a wave-like behaviour in the solenet output that matches the walking frequency, showing that our model is overfitting on that specific task.

However, considering the small amount of data we had, the fact that it was acquired in-house using our pipeline and the simplicity of the proposed models, we consider this result a sound success. We have successfully proved that estimating the sole pressure from the 3D pose is not only feasible, but also within reach for Grazper with their current setup.

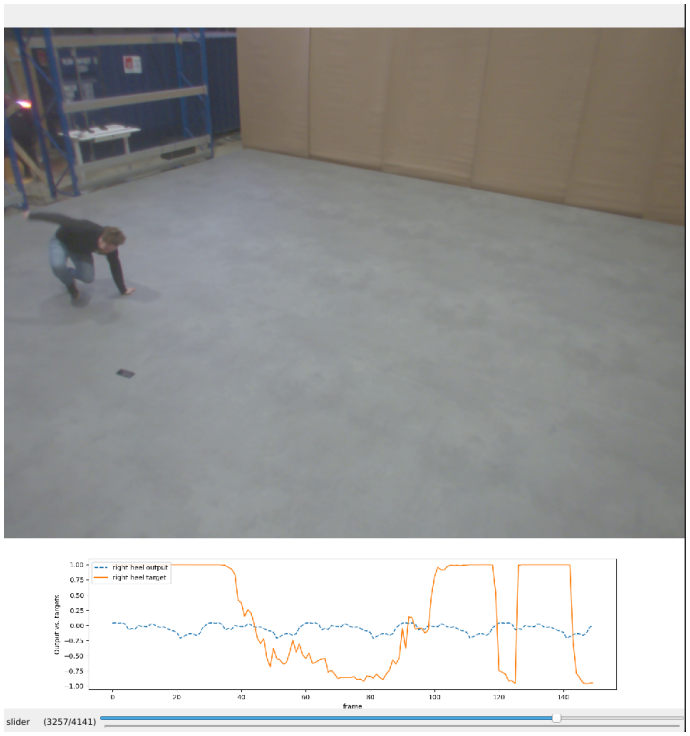


(a) Baseline model

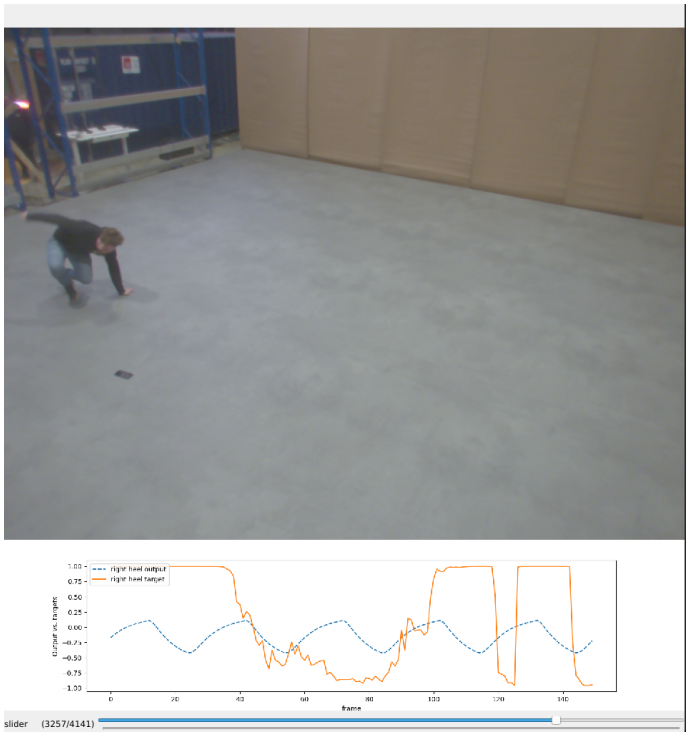


(b) SoleNet model

Figure 7.8: Models output vs. true label on the test recording, showing a person walking.



(a) Baseline model



(b) SoleNet model

Figure 7.9: Models output vs. true label on the test recording, showing a person getting up from sitting on the floor.

7.3 Experiment pipeline

After showing in last section that sole pressure estimation from pose is possible, it is time to finally incorporate the SoleNet model into the experiment pipeline described in chapter 5 and obtain this thesis' final results.

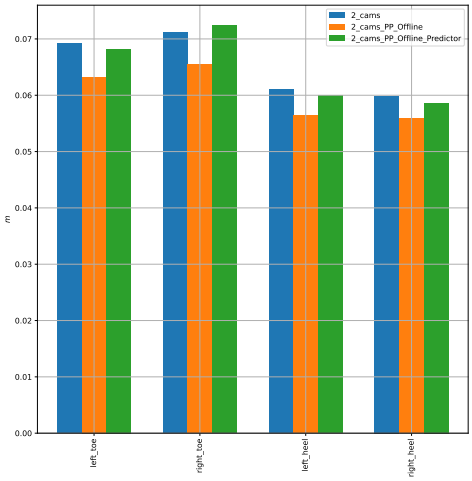
The purpose of these experiments is to show that by incorporating knowledge about foot contact, either a real signal from the insole sensors or a prediction from SoleNet, we can improve the pose detection algorithm currently used by Grazper in the fewer-camera case. For this purpose, we will take the test recording and select a number of cameras. We will then use each possible combination of cameras to obtain a 3D pose out of each of them. Then, we will modify the output pose with by using PP Offline and PP Online, both using the real sole signal and the estimated one to modify the pose. We will then compare the output to the real ground truth 3D pose, obtained with Grazper's annotation tool.

Since we are using all possible combinations of cameras, and we were running the experiment pipeline on all videos, the computational cost of this operation was rather high, and some experiments could take up to 2-3 hours. Since we were also setting some hyperparameters in the triangulation routine (minimal `visibility` from 2 camera angles, see section 3.3) and on the postprocessing modules (threshold on the soles signal), we needed to tune them and run the pipeline several times, which ended up taking a long time.

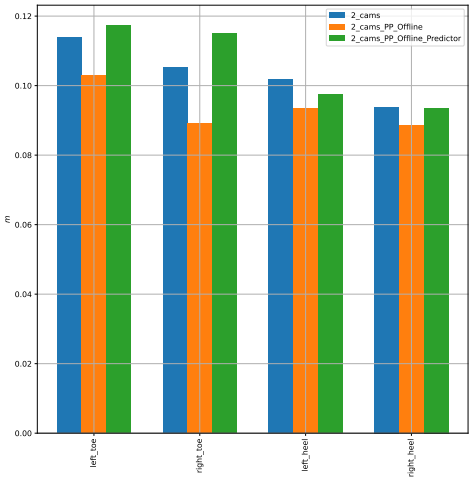
This encouraged us to rewrite the pipeline in a more efficient manner, incorporating a caching system that eliminated unnecessary recalculations and wrote the triangulation part making use of Python's multiprocessing module, since the triangulation task can be parallelized and we had a 24-thread CPU that was not being used. This helped reduce the running time of the pipeline, and allowed us to obtain the following results.

We can observe in both figure 7.10 and 7.11, the effect of treating a triangulated skeleton from 2 cameras with PP Offline and PP Online respectively on the test recording. Each plot shows the effect of using no postprocessor, the postprocessor with the real pressure signal, and the postprocessor with the estimated pressure signal by SoleNet. We show the average bias per keypoint throughout the entire recording and triangulated from all possible combinations of 2 cameras. We are excluding the PCK and RMSE plots out of the results, since in the end they do not add any relevant information that cannot be seen in the bias-standard deviation plot.

Overall, we observed very little difference from using PP Offline or PP Online as a postprocessor. We can notice some improvement both in the error, and in the standard deviation of said error when going from the triangulated signal to using the postprocessor with the real sole signal. However, the postprocessing seems to break when using the predictor. This behaviour was expected though, given how poorly the predicting model performed in tasks like the one in 7.9b.

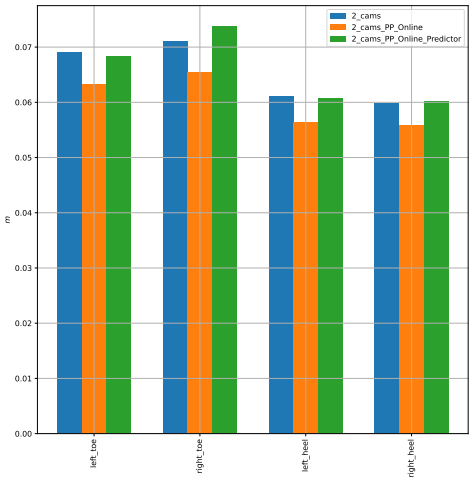


(a) Bias

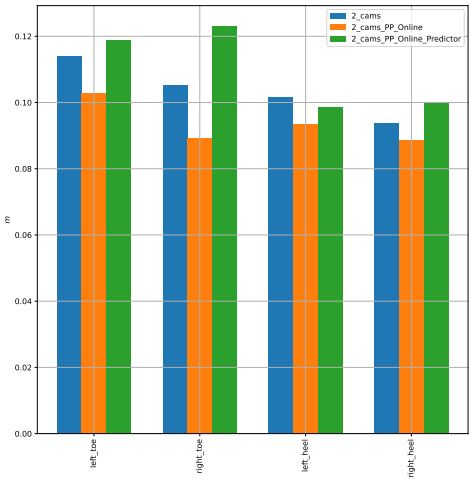


(b) Standard deviations

Figure 7.10: Average error and standard deviation of the error per keypoint in the test recording from 2 cameras with PP Offline.



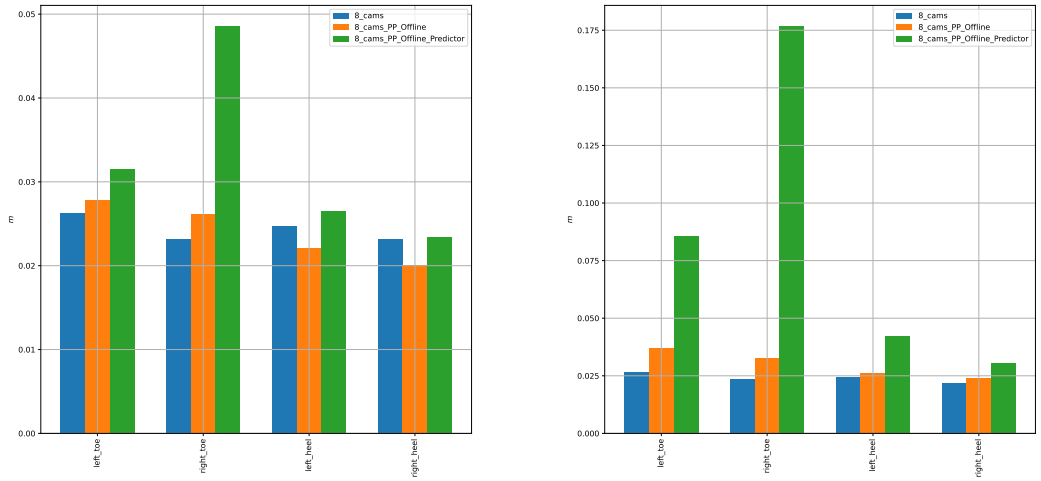
(a) Bias



(b) Standard deviations

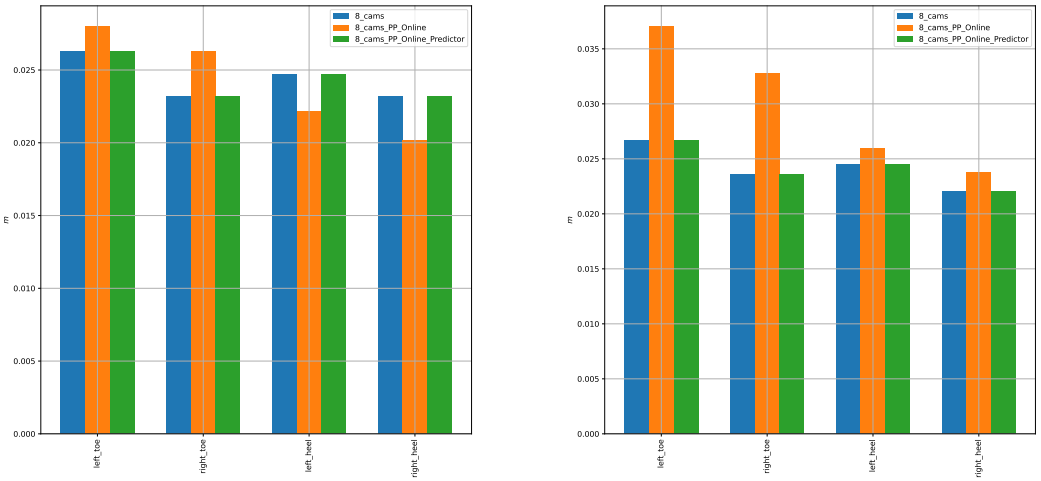
Figure 7.11: Average error and standard deviation of the error per keypoint in the test recording from 2 cameras with PP Online.

Both our postprocessors seem to have a positive impact when modifying the triangulated pose from 2 cameras. We will now inspect the opposite case, when triangulating from 8. The results are shown in figure 7.12 and 7.13. In this case, the triangulation is itself much closer to the ground truth skeleton, and the postprocessing do not provide a consistent improvement. This also aligns with our expectations, sole data is useful in the case where less cameras are involved.



(a) Bias (b) Standard deviations

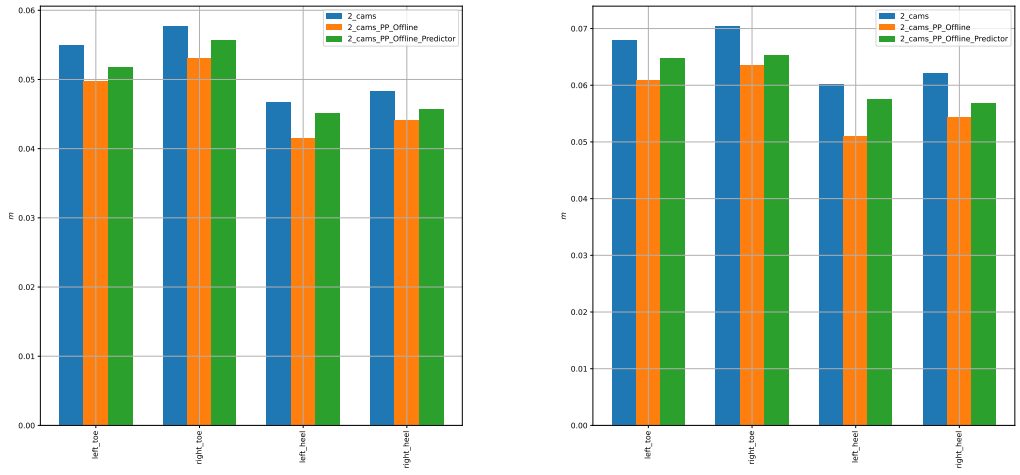
Figure 7.12: Average error and standard deviation of the error per keypoint in the test recording from 8 cameras with PP Offline.



(a) Bias (b) Standard deviations

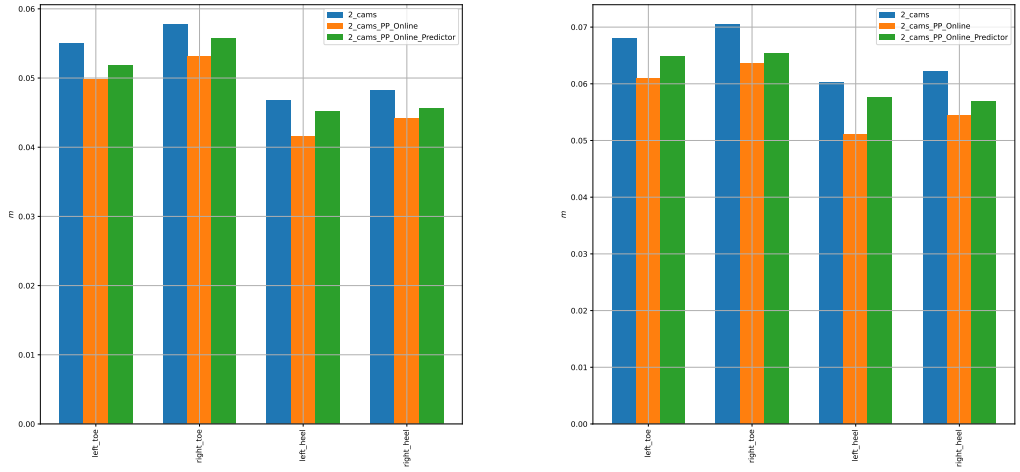
Figure 7.13: Average error and standard deviation of the error per keypoint in the test recording from 8 cameras with PP Online.

Since the behaviour of both postprocessing modules was very chaotic when using SoleNet, we decided to run an experiment with a recording that the model had seen during training with 2 cameras, to perform a sanity check. This way we can make sure the reason behind this behaviour is in fact, not having enough data in the training pipeline. The results are shown in figures 7.14 and 7.15. Here, we can observe not only that the postprocessor helps, but that the predictor approaches this solution. Again, since we are using 2 cameras the postprocessor was expected to help. And, since the model has been trained on some of the sequences and validated on others, we expect it to approach the result for the real one, but not quite be there.



(a) Bias (b) Standard deviations

Figure 7.14: Average error and standard deviation of the error per keypoint in one of the training recordings from 2 cameras with PP Offline.



(a) Bias (b) Standard deviations

Figure 7.15: Average error and standard deviation of the error per keypoint in one of the training recordings from 2 cameras with PP Online.

8 Conclusion

We believe the work in this thesis to be an overall success. Through the course of the study we have been able to set up a data acquisition capable of producing synchronized videos with sole pressure signal and videos and automatically process them. We have implemented our own 2D pose detection and 3D triangulation module that predicts a 3D pose comparable to that produced by Grazper's annotation tool using fewer cameras. We have used our generated poses, together with the sole signals to train two model architectures for predicting sole pressure based solely on the pose. Finally, we have designed two ways of incorporating sole data that make our predicted skeleton be closer to the one obtained from 8 cameras.

These results serve as a strong proof of concept that has the potential for further development and integration into AI solutions like Grazper's.

The final data acquisition setup proves to be reliable, stable, and automated, enabling Grazper to easily record high-quality datasets with the potential to obtain synchronized ground truth sole pressure signals.

Our research has demonstrated the feasibility of predicting sole pressure using deep learning techniques. However, room for improvement still exists. Our model proposals were basic, leaving room for exploration of more complex models, potentially even using the image as input instead of the 3D skeleton.

Furthermore, the amount of data used to train our networks was found to be insufficient. To resolve this, more recordings that encompass a wider range of tasks could be taken. Alternatively, a different approach could be taken, considering the binary nature of the pressure signals. By annotating existing datasets with information on contact or non-contact based on the position and velocity of keypoints, we open the possibility of using readily available datasets in our pipeline, which we believe could significantly improve performance.

Finally, we have shown how the sole signal can enhance the performance of a pose detector in scenarios with fewer cameras. This area also offers opportunities for improvement. In the case of single camera, current state-of-the-art struggles to accurately place the pseudo-3D pose in the scene, and our approach has the potential to significantly improve this.

In the multi-camera scenario, we have already developed a simple method to enhance the triangulation by incorporating sole data. However, there is still room for improvement in this area through the use of more complex post-processing techniques.

In conclusion, our thesis has achieved its goals and delivered significant results. We have demonstrated the ability to enhance the accuracy of pose detection in scenarios with fewer cameras through the incorporation of sole signals, and the possibility to estimate them. These results offer a strong proof of concept for future AI solutions and demonstrate the potential of this technique for further development and advancement.

Bibliography

[1] Zhe Cao et al. “OpenPose: Realtime Multi-Person 2D Pose Estimation using Part Affinity Fields”. In: *CoRR* abs/1812.08008 (2018). arXiv: 1812.08008. URL: http://arxiv.org/abs/1812.08008.

[2] Matthew Loper et al. “SMPL: A Skinned Multi-Person Linear Model”. In: *ACM Trans. Graph.* 34.6 (Nov. 2015). ISSN: 0730-0301. DOI: 10.1145/2816795.2818013. URL: https://doi.org/10.1145/2816795.2818013.

[3] Riza Alp Güler, Natalia Neverova, and Iasonas Kokkinos. “DensePose: Dense Human Pose Estimation In The Wild”. In: *CoRR* abs/1802.00434 (2018). arXiv: 1802.00434. URL: http://arxiv.org/abs/1802.00434.

[4] Catalin Ionescu et al. “Human3. 6m: Large scale datasets and predictive methods for 3d human sensing in natural environments”. In: *IEEE transactions on pattern analysis and machine intelligence* 36.7 (2013), pp. 1325–1339.

[5] Leonid Sigal, Alexandru O Balan, and Michael J Black. “Humaneva: Synchronized video and motion capture dataset and baseline algorithm for evaluation of articulated human motion”. In: *International journal of computer vision* 87.1-2 (2010), p. 4.

[6] Jinbao Wang et al. “Deep 3D human pose estimation: A review”. In: *Computer Vision and Image Understanding* 210 (2021), p. 103225. ISSN: 1077-3142. DOI: https://doi.org/10.1016/j.cviu.2021.103225. URL: https://www.sciencedirect.com/science/article/pii/S1077314221000692.

[7] Georgios Pavlakos et al. “Coarse-to-Fine Volumetric Prediction for Single-Image 3D Human Pose”. In: *CoRR* abs/1611.07828 (2016). arXiv: 1611.07828. URL: http://arxiv.org/abs/1611.07828.

[8] Julieta Martinez et al. “A simple yet effective baseline for 3d human pose estimation”. In: *CoRR* abs/1705.03098 (2017). arXiv: 1705.03098. URL: http://arxiv.org/abs/1705.03098.

[9] Yuliang Zou et al. “Reducing Footskate in Human Motion Reconstruction with Ground Contact Constraints”. In: (2020), pp. 448–457. DOI: 10.1109/WACV45572.2020.9093329.

[10] Kevin Xie et al. “Physics-based Human Motion Estimation and Synthesis from Videos”. In: *CoRR* abs/2109.09913 (2021). arXiv: 2109.09913. URL: https://arxiv.org/abs/2109.09913.

[11] Moticon ReGo AG. *Moticon OpenGo sensor insoles product information*. URL: https://moticon.com/opengo/sensor-insoles.

[12] Sensoria Fitness Inc. *Sensoria smart socks product information*. URL: https://www.sensoriafitness.com/smartsocks/.

[13] “2D and 3D Vision Formation”. In: *An Introduction to 3D Computer Vision Techniques and Algorithms*. John Wiley & Sons, Ltd, 2009. Chap. 3, pp. 15–94. ISBN: 9780470699720. DOI: https://doi.org/10.1002/9780470699720.ch3. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/9780470699720.ch3.

[14] Nvidia Corporation. *Vision Programming Interface Documentation*. URL: https://docs.nvidia.com/vpi/appendix_pinhole_camera.html.

[15] Open Source Computer Vision Library. *Camera Calibration and 3D Reconstruction*. URL: https://docs.opencv.org/4.x/d9/d0c/group__calib3d.html.

[16] Valentin Bazarevsky et al. “BlazePose: On-device Real-time Body Pose tracking”. In: *CoRR* abs/2006.10204 (2020). arXiv: 2006.10204. URL: https://arxiv.org/abs/2006.10204.

[17] Common Objects in Context. *COCO 2020 Keypoint Detection Task*. URL: https://cocodataset.org/#keypoints-2020.

[18] István Sárándi et al. "MeTRAbs: Metric-Scale Truncation-Robust Heatmaps for Absolute 3D Human Pose Estimation". In: *CoRR* abs/2007.07227 (2020). arXiv: 2007.07227. URL: https://arxiv.org/abs/2007.07227.

[19] Nvidia Corporation. *Nvidia Jetson Xavier NX Series product information*. URL: https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-xavier-nx/.

[20] *IEEE Standard for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems*. Standard. New Jersey, US: IEEE Instrumentation and Measurement Society, June 2020. URL: https://standards.ieee.org/ieee/1588/6825/.

[21] D.L. Mills. "Internet time synchronization: the network time protocol". In: *IEEE Transactions on Communications* 39.10 (1991), pp. 1482–1493. DOI: 10.1109/26.103043.

[22] R.M Zur, Y Jiang, and C.E Metz. "Comparison of two methods of adding jitter to artificial neural network training". In: *International Congress Series* 1268 (2004). CARS 2004 - Computer Assisted Radiology and Surgery. Proceedings of the 18th International Congress and Exhibition, pp. 886–889. ISSN: 0531-5131. DOI: https://doi.org/10.1016/j.ics.2004.03.238. URL: https://www.sciencedirect.com/science/article/pii/S0531513104006697.

[23] Murat H. Sazlı. "A brief review of feed-forward neural networks". In: *Communications Faculty of Sciences University of Ankara Series A2-A3 Physical Sciences and Engineering* (2006), pp. 0 - 0. URL: https://dergipark.org.tr/en/pub/aupse/article/890416.

[24] Tomasz Szandala. "Review and Comparison of Commonly Used Activation Functions for Deep Neural Networks". In: *CoRR* abs/2010.09458 (2020). arXiv: 2010.09458. URL: https://arxiv.org/abs/2010.09458.

[25] KyungHyun Cho et al. "On the Properties of Neural Machine Translation: Encoder-Decoder Approaches". In: *CoRR* abs/1409.1259 (2014). arXiv: 1409.1259. URL: http://arxiv.org/abs/1409.1259.

[26] Diederik P. Kingma and Jimmy Ba. *Adam: A Method for Stochastic Optimization*. cite arxiv:1412.6980Comment: Published as a conference paper at the 3rd International Conference for Learning Representations, San Diego, 2015. 2014. URL: http://arxiv.org/abs/1412.6980.

[27] Connor Shorten and Taghi M. Khoshgoftaar. "A survey on Image Data Augmentation for Deep Learning". In: *J. Big Data* 6 (2019), p. 60. DOI: 10.1186/s40537-019-0197-0. URL: https://doi.org/10.1186/s40537-019-0197-0.

A Appendix: Code listings

get_time.c

```c
#include <stdio.h>
#include <sys/time.h>

int main(void)
{
    struct timeval tv;
    struct timezone tz;
    gettimeofday(&tv, &tz);
    printf("%d.%d\n", tv.tv_sec, tv.tv_usec);
    return 0;
}
```

set_time.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>

void set_time(time_t seconds, suseconds_t microseconds);

void set_time(time_t seconds, suseconds_t microseconds)
{
    struct timeval tv;
    tv.tv_sec = seconds;
    tv.tv_usec = microseconds;
    settimeofday(&tv, NULL);
}

int main(int argc, char *argv[])
{
    int seconds, microseconds;
    sscanf(argv[1], "%d.%d",&seconds, &microseconds);
    set_time(seconds, microseconds);
    return 0;
}
```

check_time_reset.sh

```
#!/bin/bash
# Checks the offset between phone's and computer's clock every second.

while true
do
    phone_time=$(adb shell "su -c '/data/data/com.termux/files/home/time/get_time'")
    current_time=$(date +"%s.%6N")
    diff=$(echo $phone_time-$current_time | bc)

    echo "[$(date -d @$current_time)] Offset: $diff" | tee -a log
    sleep 1
done
```

sync.sh

```
#!/bin/bash
# Program for synchronizing a phone's clock to a computer.
# Needs to have set_time.c and get_time.c compiled on the phone.
# It calculates the offset of writing and getting the time on the
# phone doing several trials (5 by default) and then writes the time
# compensating it (assuming that get_time is negligible against set_time).

# Seconds to send the start acquisition signal from the phone to the soles.
phone_to_soles_delay=0.05

n=${1:-5}
echo "Synchronizing! Estimating offset with $n iterations" | tee -a log.txt

path="/data/data/com.termux/files/home/time" # path to set_time and get_time
offset=0 # Variable that will store the running avg of the offset
num_accepted=0

while [ $num_accepted -lt $n ]
do
    # Writing and reading the current time
    current_time=$(date +"%s.%6N")
    adb shell "su -c '$path/set_time $current_time'"
    phone_time=$(adb shell "su -c '$path/get_time'")
    current_time=$(date +"%s.%6N")
    # Calculate difference
    diff=$(echo $phone_time-$current_time | bc)

    # Only accept negative diffs. Sometimes adb takes longer to read the time from the
    # phone and returns a great positive diff, which makes no sense
    if (( $(echo "$diff < 0" | bc -l) ))
    then
        offset=$(echo "($num_accepted * $offset + $diff)/($num_accepted+1)"| bc -l)
        num_accepted=$(($num_accepted+1))
        # Print out
        echo "[$(date -d @$current_time)] It. $num_accepted. \
        Offset: $diff. Running avg: $offset" | tee -a log.txt
```

```
    fi
    sleep 1
done

# Write compensating the offset
current_time=$(date +"%s.%6N")
offseted_time=$(echo "$current_time - $offset + $phone_to_soles_delay" | bc)
adb shell "su -c '$path/set_time $offseted_time'"
echo "[$(date -d @$current_time)] Synced!" | tee -a log.txt

./check_time_reset.sh
```

Technical
University of
Denmark

Richard Petersens Plads, Building 324
2800 Kgs. Lyngby
Tlf. 4525 1700

www.compute.dtu.dk