BedBasedEcho

Joey Haas
Rembrandt Klazinga
Nick van Stijn
Jasper Teunissen
Ying Zhang

Client: Eelko Ronner, ZoekBeter B.V
Coach: Marco Loog, Delft Univeristy of Technology
Coordinator: Otto Visser, Delft University of Technology

July 15, 2020

Foreword

This is the report of the BedBasedEcho Bachelor End Project. The BedBasedEcho is an ambitious project, which aims to reduce the workload in hospitals by robotising echocardiography. This report is not only meant to highlight the final product, but also to highlight the process of development. We will describe highs and lows, since there has been a considerable amount of both.

We would like to explicitly thank our project coach, professor Marco Loog, who provided us with valuable advice and experience during the project. We would also like to thank Dr. Eelko Ronner, who has an inexhaustible well of energy and imagination, which he uses to lead us all into the next stage of echocardiography.

Lastly, we would like to thank the reader for showing interest in our project.

Sincerely,

Joey Haas
Rembrandt Klazinga
Nick van Stijn
Jasper Teunissen
Ying Zhang

Contents

1 Summary 1

2 Introduction 2
- 2.1 Medical Context 2
  - 2.1.1 Physiology 2
  - 2.1.2 Quick-look echos 3
- 2.2 Motivation 3
- 2.3 Problem Definition 4
  - 2.3.1 Automation 4
- 2.4 Problem Analysis 4
  - 2.4.1 Current state 4
  - 2.4.2 Requirements 4
- 2.5 Outline of the report 6

3 Design 7
- 3.1 Data Collection 7
  - 3.1.1 Design considerations 8
  - 3.1.2 Data Collection Controller 9
  - 3.1.3 Random Walker 11
  - 3.1.4 Data Writer 13
  - 3.1.5 Graphical User Interface 13
  - 3.1.6 Image Capturing 14
  - 3.1.7 Usage in practice 15
- 3.2 Machine Learning 16
  - 3.2.1 Data format/storage 17
  - 3.2.2 Transforming to the internal format 18
  - 3.2.3 Training a convolutional model 19
- 3.3 Deployment 21
  - 3.3.1 Frame Saver 21
  - 3.3.2 TensorFlow component 22
  - 3.3.3 Instruction sending 23
  - 3.3.4 Autonomous Controllers 23
- 3.4 Testing 24
  - 3.4.1 Automated Testing 24
  - 3.4.2 Manual Testing 25
  - 3.4.3 Software Quality Assurance 25

4 Challenges 26
- 4.1 Software Architecture 26
  - 4.1.1 C++ (Testing) and CI 26

4.2 Data Gathering. 26
4.2.1 Gyroscope and Accelerometer 26
4.2.2 Visual Motion Tracking 27
4.2.3 Data gathering in practice 28
4.2.4 Outcome of challenges. 29
4.3 Learning. 29
4.3.1 Mismatch between perceived and absolute translational vector . 29
4.3.2 Noisy images. 29
4.3.3 Hardware limitations and memory issues 29
4.3.4 Overfitting 29
4.3.5 Normalization of translational and rotational data 30

5 Process 31
5.1 Tools and techniques 31
5.1.1 SCRUM 31
5.1.2 Gitlab 32
5.1.3 Continuous Integration 32
5.2 Communication 33
5.2.1 In-group communication and cooperation 33
5.2.2 Communication with coach and client 34
5.3 Non-disclosure agreement. 34
5.3.1 Intellectual Property 35
5.3.2 Patient Data 35

6 Conclusion 36
6.1 Results Based on Requirements. 36
6.2 Results Based on Design Goals 37

7 Discussion 39
7.1 Deployment Results. 39
7.2 Recommendations 40
7.2.1 Improving inference speed 40
7.2.2 Data storage 41
7.2.3 Improving model accuracy 41
7.2.4 Human Computer Interaction 42
7.3 Ethical considerations 42
7.3.1 Positive influences on health care 43
7.3.2 Possible concerns 43

Glossary 45

A SIG Feedback 46
A.1 Introduction 46
A.2 Duplication. 46
A.3 Unit Size. 46
A.4 Unit Complexity 47
A.5 Unit Interfacing 47
A.6 Module Coupling 47
A.7 Second Feedback 48

B Project Description 49

C Model details 51

D Info Sheet 52

E Research Report 53

E.1 Introduction 53

E.2 Hardware Components. 56

E.3 Software Components 57

E.4 Frameworks 66

Bibliography 69

1 Summary

Many cardiovascular problems can be solved, or at least alleviated, with very cheap and low-risk medicine (Braunschweig et al., 2011; Weinstein and Stason, 1985). If we can detect such problems early, we can increase the average life-expectance significantly. The most comprehensive detection method is ultrasound. However, ultrasound imaging is a complex job, and sonographers are scarce (Craig, 2003; Bowman et al., 2019).

The BedBasedEcho could be the solution to this problem. It is a robotic system, If the BedBasedEcho would be provided with software to autonomously gather images of the heart, we can alleviate the pressure currently exerted on sonographers.

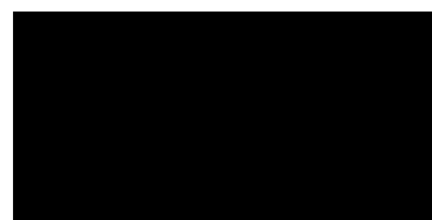


Figure 1.1: Render of the BedBasedEcho prototype

The aim of this project is to develop a start-to-finish process that entails gathering data for, training, and deploying an AI model. This pipeline will be used to train the robot to find one specific view. The deployment of the AI model would point to where the robot should move the echo probe, according to the output images of the probe. This would eventually lead to the probe moving to the optimal position for that specific view of the heart.

To realise these three components in one project, the team has followed a weekly workflow, with each week containing two smaller sprints. The team met on Mondays and Thursdays, and kept a continuous backlog of features. These features were extracted from the project requirements, and were often assigned to sub-groups of the team.

As a result of this workflow, the team was able to realise all previously stated goals. Applying the developed process on the Parasternal Long Axis view (PLAX), yielded promising results. When the probe was position high above the PLAX, the trained model was able to autonomously move towards the correct view[1].

[1]https://youtu.be/WZ7-pRN4qHY

2

Introduction

To give the reader the necessary background to understand this report, we will first give an introduction on multiple topics. This includes a brief medical context, the motivation and ultimate aim of the BedBasedEcho, the scope of the project, and finally an overview of the report.

2.1. Medical Context

As this project ultimately revolves around a medical topic, it is prudent to start by explaining the medical terminology and background.

2.1.1. Physiology

The heart contains 4 chambers, as can be seen in figure 2.1.

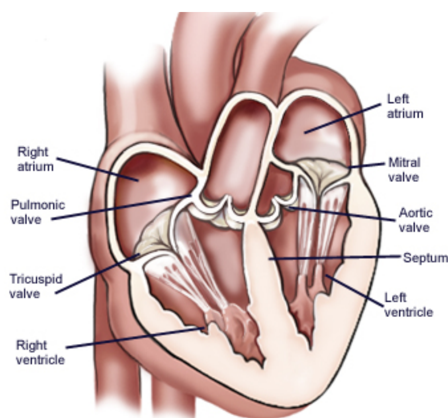


Figure 2.1: Cross-section of the heart (Cleveland Clinic)

The topic at hand is "cardiac echos": videos of the heart, taken by a ultrasound probe, in a process also known as sonography. The probe produces sound waves, which enter the body and reflect back when hitting different materials. The type of the material determines the strength of the reflection, which is seen in the echo view as a brighter or darker spot. Notably, blood reflects very little, while muscle reflects more. This means that the structure of the heart (being comprised largely of muscle) can be clearly shown using sonography. An echo probe can capture these "frames" quickly, at a rate of more than 30 per second. The result is a near-live video of the heart in motion, which lets echocardiographers diagnose complex issues of the heart.

An echo probe takes a 2D slice of whatever it is looking at, in a fan-like pattern. This fan can be seen as protruding from the head of the probe, slicing into the body. To look at a different

part of the heart, the probe can be rotated on any axis, or moved laterally. In echocardiography there are a number of "views": these are fixed positions relative to the heart, that sonographers can consistently find and use to make a diagnosis (Landzaat, 2020). For instance, the **apical 4-chamber view** (A4C) looks at the heart from below, through the apex, showing all 4 chambers of the heart. Another view discussed in this project is the **parasternal long axis view** (PLAX). To reach this view, one places the probe perpendicular to the center of the chest (or sternum), and rotates it such that 2 chambers are seen: the left ventricle and the left atrium (see figure 2.1).

2.1.2. Quick-look echos

Sonographers often make so called "quick-look echos". These involve making images of the heart according to the aforementioned standard views. While there are fourteen of such views in total, a quick-look echo usually only needs five of them. Multiple views are required to be able to identify most common issues.

During a quick-look echo, a sonographer will position the probe such that the display will show one of the required views. When the display shows the correct image, the sonographer will record a short video to show the motion of the heart. The sonographer will repeat this process for each of the required views. The result that will be analyzed by the cardiologist will be a set of short videos, each showing one view.

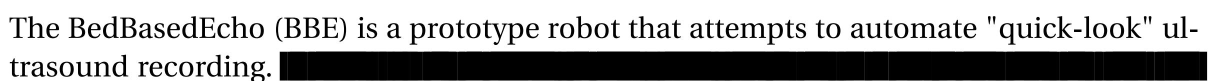
2.2. Motivation

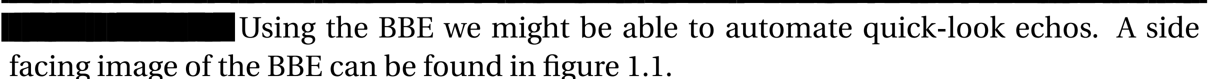
In the Netherlands, roughly 29 percent of 140.000 deaths in 2019 are caused by cardiovascular disease. Many cardiovascular problems can be solved, or at least alleviated, with very cheap and low-risk medicine. If we can detect such problems early, we can increase the average life-expectancy significantly.

There is a plethora of cardiographic tools available, some of the more well known being: ECG, MRI and most importantly, **ultrasound**. Ultrasound is one of the most useful tools for cardiologists. It provides them with an actual image of the inside workings of the heart, while at a fraction of the cost of other visual methods like MRI.

The one downside of ultrasound, is that it takes skilled experts to extract the important details from the heart. For example, in the Reinier de Graaf Hospital in Delft, Netherlands, in total, five sonographers are employed. These five are responsible for making 10,000 high-end ultrasound recordings each year. If we could find some way to automate this process, cardiologists can elect to use echocardiography more freely, and therefore detect cardiovascular problems earlier.

The BedBasedEcho (BBE) is a prototype robot that attempts to automate "quick-look" ultrasound recording.





Using the BBE we might be able to automate quick-look echos. A side facing image of the BBE can be found in figure 1.1.

2.3. Problem Definition

When a patient with suspected heart issues enters the hospital, a cardiographer ideally makes a quick-look echo to find out if there is something wrong with the heart. Unfortunately, due to the limited number of sonographers, this is not always possible. Instead, cardiographers have to resort to the easier, but less comprehensive ECG scans instead (Ahmed, 2020). The goal of the BedBasedEcho is therefore to automate quick-look echos.

2.3.1. Automation

Automating quick-look echos would allow more patients to receive such a treatment, while freeing up time for the sonographers to do more complex echos.

When the patient is laid down on the BBE, the software should make a full quick-look echo, without any human intervention. This means that the probe should move towards the correct position for each of the required views, and save a short video. When the system has imaged all the required views, it should move the probe to a resting position and allow the patient to get up, all without human interaction.

2.4. Problem Analysis

Creating the BedBasedEcho is a large project, possibly spanning multiple years of development and testing. The time allotted to this project is merely ten weeks. Therefore, the scope of this project is limited compared to that of the entire BedBasedEcho. Luckily, a substantial part of the project is already in the prototyping stage. In this section, we will first explain what has already been done, followed by what we will do, and what is left as future work.

2.4.1. Current state

The hardware side of the BedBasedEcho is in an advanced stage. These mechanical components can be controlled in software. There are still improvements to be made, but the functionality is sufficient for what we need to do with it.

The available scanning software for the BBE is a lot more limited. It is already capable of finding an apical 4-chamber view with very limited success. The search algorithm implemented is based on a heuristic. It estimates the quality of the current image using a neural network, and switches between searching modes like: "lawnmower", "angle-vary", and "gray-scale" based on this estimation.

2.4.2. Requirements

In this project, we will write software that finds the parasternal long axis view autonomously. Our software should be written such that it is usable in the final product. This requires that the software is easy to extend and maintain. It should also be resilient to errors, such that no manual intervention is required under regular conditions. Explanations for these design goals can be found in appendix section E.1.4.

To find the parasternal long axis view, we will train a network that is able to determine the desired direction given a short video fragment recorded by the echo-probe.

The algorithm will do this by outputting a vector pointing in the direction of the PLAX view.

Our requirements have slightly changed since the research report, reasons for this can be found in chapter 4. Also, these requirements are not prioritised since they all depend on the success of their predecessors (Non-essential requirements can be found in section E.1.5). The extracted requirements (with changes highlighted in **bold**) are as follows:

- The software must be able to read echo images from the probe on command, using the Software Development Kit provided by ██████, manufacturer of the echo probe.
- The algorithm must generate instructions that would move the probe towards the **Parasternal Long-axis view**.
- There must be a way to record the position and angle of the probe **as it is suspended in the BBE**.
- There must be a way to record the aforementioned data in conjunction with echo images, such that the results are stored together in a synchronised fashion. This allows them to be used as training data.
- The algorithm must have low enough latency to run in semi-real-time: it is allowable that movement instructions sometimes stop for no more than 5 seconds to allow processing of video data, such that the next movement may be determined.
- There must be an method of executing the trained algorithm, either simulated **or implemented in the real world**, thereby visualising its behaviour.

These requirements for the algorithm can be divided into three categories corresponding with the final three components of the system: data collection, network training and deployment. Each category is dependant on the success of the previous category. In the following sections the specific requirements for each of the categories will be explained.

Data collection

To train our algorithm, we need echo data combined with the position of the echo probe. During the research phase, we found no data sets supplying both properties. This requires us to collect our own data. To do that, we must have the ability to record and store echo images, and we must have the ability to record the position of the echo probe. Finally, we must be able to link both recordings.

Usually, a neural network requires a lot of data to train. Because of this, one of the goals is to be able to gather training data in bulk, and with high accuracy. It should be convenient for an experienced sonographer to gather new data for training the network.

Network training

The final product is a neural network trained in TensorFlow (TensorFlow, 2020) that takes a series of echo images, and produces a vector indicating the direction of the parasternal long axis view. The final product should run on a Ryzen 3200G CPU with eight gigabyte of memory, so it should not be too resource intensive. To label the model performance as successful, it must find the correct view within three minutes in 80% of the cases on the supplied hardware. It must do so without pausing for more than five seconds. The algorithm

should also provide some way to identify when the optimal parasternal long axis view is reached. The algorithm could be optimised such that the movement instructions run real-time. That is to say, the instructions are generated with such low latency that movement never has to stop.

During this project, we will only focus on probe movement towards the parasternal long axis view. Other views will not be considered. The algorithm will use plain echo images only, and will not be able to change to colour doppler mode when the correct view is reached. It will not set color boxes around key features of the image either. Lastly, the algorithm is only required to support Windows 10. No explicit support for other operating systems will be given.

Deployment

In the end, the algorithm defined above should be able to control the probe To achieve this, the deployment part of the software should support sending images captured by the probe to the neural network. These images should be combined with position data received from the hardware in the BBE. The network will then process that data and give a vector as output. That vector should be converted to movement commands for . That way, could move towards the correct parasternal long axis view.

2.5. Outline of the report

We will start by discussing the design of the final solution, after this we will elaborate on the implementation of this design. Because the project was volatile in nature (It required many design revisions to get right), we would also like to highlight the process that has taken the team from the start to the final design. After this, we will show the conclusions that can be taken from our research and design, and finally we will discuss our future recommendations and ethical considerations.

3

Design

In this chapter, we will discuss the final design of each of the three components of the project, in the logical order of data flow. However, to grasp the physical layout of these components, we will first quickly discuss the physical architecture.

Physical Architecture

The architecture is comprised of two separate components. One component is built around the Robot Operating System (ROS) framework, which on the BedBasedEcho (BBE). The other component is built around a Software Development Kit (SDK) by [redacted] which is needed to interact with the probe.

Since ROS only runs on Linux systems, and the [redacted] drivers are written only for 32-bit Windows, the only option for now is to use two machines: The aforementioned BBE and a second machine running Windows 10. Via an Ethernet cable, these two systems are connected and are able to share information.

The Windows machine is responsible for communicating with the probe. For this, it needs to have the [redacted] drivers installed. The probe can be connected using a simple USB cable. The Windows machine will also produce predictions using a pre-trained TensorFlow model. It will try to do this at the highest rate possible using an Intel i7 7700HQ CPU.

The more complicated physical component is embedded into the BBE. It is responsible for recording positional, angular and temporal data. The positional and angular data is needed to calculate a vector to the desired position for each timestamp. This conversion is discussed further in section 3.1.7.2. Temporal data is needed to sync the related positional and angular data with the captured images.

Later on in the product pipeline, when the network has been trained to provide predictions, the BBE also functions as an interpreter for the generated predictions. This element is elaborated on in section 3.3.4.

3.1. Data Collection

Before we could train any neural network, we needed to collect a large amount of accurate training data. Notably, this data collection is not only about volume and accuracy, but

also ease of use: one of the core design goals of the project is to make collecting new data in the future simple for any sonographer, even if they are otherwise unfamiliar with the project.

3.1.1. Design considerations

Data labelling

The trained network expects to receive a still frame, or multiple frames, from the echo probe. In return, it outputs a vector to where it thinks the heart is, in both translation and rotation. For instance, if the probe is placed too high on the chest, we want the network to output a vector pointing down. In the final product, the robot would simply have to follow this vector to reach the heart.

To train this behaviour, the training data should consist of frame-by-frame echo images, where each image has a label: this label is the exact vector pointing towards the target view, at the instant the image was taken.

Core problems

Any proposed solution to gather these images and labels would have to address a number of points:

- How can the probe be moved to the desired view?
- How can the images be read from the probe? As mentioned, this can not be done by the BBE robot, as its operating system is incompatible with the ██████ SDK.
- How can we read out the position of the probe on demand?
- How can the position and the image be synchronised in time?
- How do we ensure the collected data is diverse, i.e. that it covers the entire heart and its surroundings?

Data gathering concept

To answer these questions, multiple options were explored. Those that did not emerge successful have been documented in section 4.2. The method we settled on uses the BBE robot itself: the echo probe is mounted into the BBE, which moves it autonomously *and* tracks the position of the probe at every moment. This solves the problem of finding the probe's position accurately.

A ██████████ controller is attached to the BBE, which can be used to adjust all 6 movement axes independently. This addresses how the correct view can be reached in the first place.

As mentioned, a second computer, running Windows, is connected to the echo probe. When a recording is started, it stores images continuously. To aid navigation, it always displays the latest echo frame in a GUI.

Lastly, both these computers use a combination of internal clocks and ethernet communication to synchronise when images and positions are read out. This solves the second-to-last requirement.

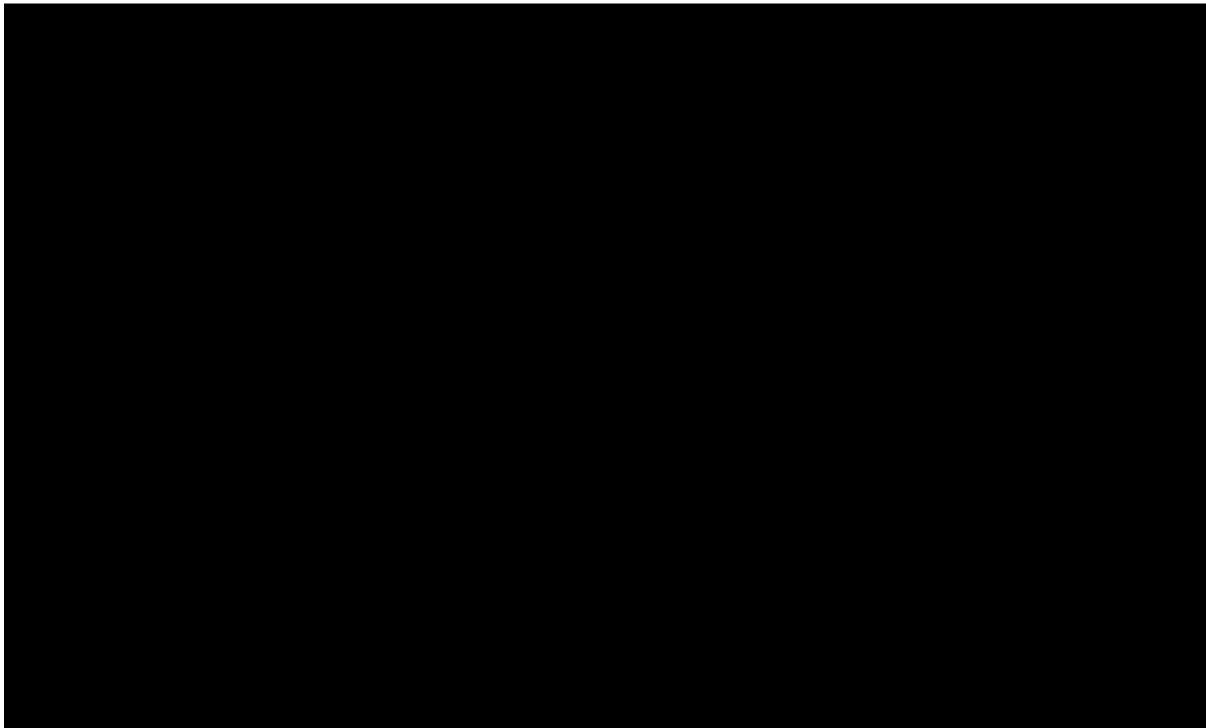


Figure 3.1: Component Diagram of the Data Collection Component

The basic idea is that we lie a member of the team on the BBE. A sonographer will then use the controller to move the probe towards the target view, using the live echo image to navigate. Then, we will make the probe move randomly around the heart, and collect images and positions during this random movement. Over a longer period of time, the random movement should cover the area around the target view, giving us a diverse dataset.

To facilitate this data collection, we have designed a software architecture which automatically exports our needed data. An overview of the components it contains and how those components interact with each other can be found in figure 3.1. Furthermore, we will describe each of the relevant components in their own separate section

3.1.2. Data Collection Controller

The Data Collection Controller (DCC) is the component responsible for controlling the BBE robot during data collection. The core behaviour when collecting data revolves around knowing our goal position (a good view of the heart), and then moving to points away from this position, while recording where we are and what this position looks like on the echo probe.

Design Considerations

The DCC should operate in "cycles", where each cycle produces a series of labelled images. Before every cycle, the probe must already be positioned at the optimal position for the desired view. When the cycle starts, it starts recording positional data, and tells the Windows machine to start recording image data. It then finds a next random position to move to using a "random walker" (a component described in section 3.1.3). While recording, the robot

should continue moving randomly around the heart, until an interrupt is issued.

When interrupted, the robot should move back to the optimal position, and stop recording image and positional data when it arrives. This recording is then written to one separate file, and the robot is immediately ready to start recording again. This design makes it possible to create 'checkpoints' in the data, so that when something goes wrong, we do not lose all previous data.

With the "checkpointing" system, it is also possible to adjust what the robot sees as the "optimal" position. For instance, if the person on the bed moved a bit, or if we decide we can improve the view somewhat with manual input, this respite between the recordings can be used for adjustments with the dual-axis controller (see figure 3.1). When the movement cycle is started anew, the adjusted position will be used as the goal.

A visualisation of the complete cycle can be found in figure 3.2.

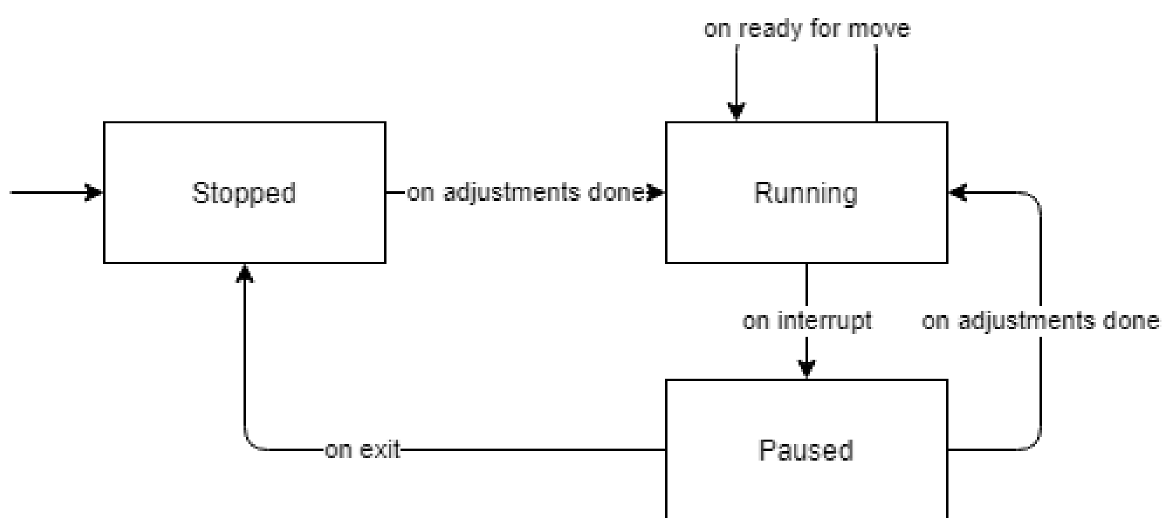


Figure 3.2: State machine diagram of the data collection cycle

Implementation

The implementation of the component above was straightforward in terms of behaviour. The DCC would have to be encapsulated as a ROS node, which subscribes to topics and can publish commands.

To illustrate an issue in implementation that arose due to ROS, we will elaborate on how the BBE uses it. ROS, in combination with Python, is used to parse input, make decisions, and physically move the robot arm. For our purposes, this poses a problem: ROS is near impossible to install on Windows, and requires Python 2 (which is deprecated in favour of Python 3) to communicate with directly. Moreover, it is unlikely that ROS-dependent code could run in a CI setting. The existing bed code uses Python 2 and basic ROS, thereby sacrificing CI and testability. However, if at all possible our additional code should be written in Python 3, avoiding the issues mentioned above. What we needed for this was a way to separate our components from the BedBasedEcho, while retaining access to ROS.

The solution to this problem came in the form of roslibpy. Roslibpy is a bridging library built for Python 3, which allows abstracting a ROS installation behind socket calls. Effec-

tively, this meant our code could run in Python 3, run in any CI scenario, and be testable on systems that did not even have ROS installed.

Very briefly, roslibpy accomplishes this by establishing socket-based links to the services and topics provided by a ROS installation. This allows it to communicate with other ROS nodes as intended, while running on a completely different version of the Python language. It is testable by mocking these linking calls away. You can read the call list of these mocks to determine how the roslibpy node behaves.

The DCC can be seen as a "mock" ROS node, which in fact uses roslibpy to communicate with the rest of the BBE. The same was done for all other major components on the bed, such as the Data Writer (section 3.1.4) and the Autonomous Controllers (section 3.3.4).

3.1.3. Random Walker

The Random Walker is a sub-component used by the DCC to generate movement over time.

Design Considerations

The most important feature of the random walker is simply that it should "walk". That is, the movement should be coherent (not too jerky), but also not too predictable.

Coherence is important for training recurrent networks: if consecutive frames do not move in about the same direction, it becomes very hard for a network that takes time into account to give a meaningful prediction

On the other hand, movement should not be in perfectly straight lines. A recurrent neural network could infer the direction of travel from a series of images, and simply output that as the direction to move in.

Lastly, movement should be limited in some sense, such that the walker does not arbitrarily go anywhere on the body: the heart is still the centre of attention.

The desired behaviour is one where the walker circles the heart randomly, but is 'pulled' harder towards the optimal position, the further away from the optimal position it gets.

As an added bonus, we value *exactly perfect* views, not just views that are quite close but just barely not there. Hence, the walker should pass straight through the optimal position whenever it comes close, instead of barely missing it.

The walker should move in this way on both the translational and rotational axes.

Implementation

To adhere to these design choices, we opted for a random heading change system. To this end, the walker was given a heading, in the form of an angle, and a continuous change in this heading. The change in heading is adjusted for every movement step, leading to coherent left or right turns. Furthermore, the change in heading is biased, such that it prefers to adjust left if that is the direction of the optimal position, or vice versa. This effect becomes stronger the further the walker is from the optimal position.

Furthermore, this change in angle also decays with 20% each cycle, which facilitates a natural bias back to 0°change (moving straight). Also, for each cycle, we check if the walker is

within a certain radius and angle threshold to the optimal position. If it is, we set its heading to point exactly to the optimal position, causing it to 'snap' to it, and to exactly cross it.

A visualisation of this walker can be found in figure 3.3. Note that over a longer random walk, most areas are explored at least once, and that a lot more data is gathered near the center. Also note that very close to the center, the paths are straight, and pass through a single point: this is the snapping behaviour mentioned above: it ensures there are many data points of a movement towards the perfect view.

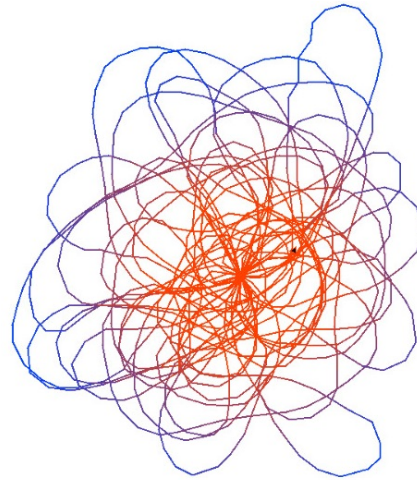


Figure 3.3: Visualisation of the random walker around an optimal position

Besides altering the position each cycle, all three rotational axes should be varied in a similar fashion. For this purpose, a similar velocity based system was implemented. This time, each cycle, all angles are updated according to their own velocities. These velocities also get randomly updated and decayed each cycle. An example of these randomly generated angle changes can be found in figure 3.4.

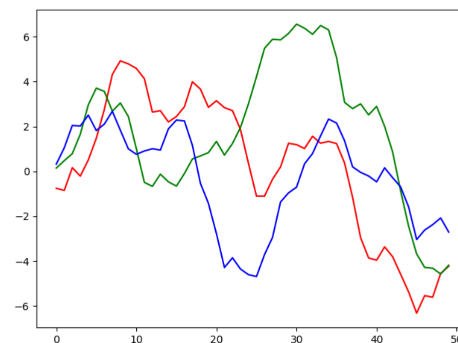


Figure 3.4: Visualisation of the randomly varied rotational angles

3.1.4. Data Writer

The data writer is a separate roslibpy node that is responsible for writing the positional and angular data to disk. It receives this information via a separate ROS topic that is published to whenever a move command is made on the hardware. It also stores the timestamp at that instant. This is needed so we can synchronise the position data with the echo images after data collection is completed (the images are stored locally on the Windows computer, with their own timestamps). The chosen data format is a file, with the timestamp stored n nanoseconds, and all six axes stored separately as floats

3.1.5. Graphical User Interface

In order to also provide meaningful interaction with the end user, a graphical user interface is a must. The most important functions of this interface would be viewing the ultrasound image coming from the probe, and controlling ultrasound settings such as depth, frequency and gain. Lastly, the user should be able to change the save location of the gathered data.

The example code provided by contained a rudimentary GUI written in the Due to the complexity of extending a form like this, we decided to rewrite this form in C# using the library, allowing us to make use of the managed aspect of this language. This gave us access to a more robust what-you-see-is-what-you-get form designer and event functionality, speeding up development significantly.

Outside of the previously mentioned fundamentals, we have also added a control to enable a built-in despeckling algorithm, named

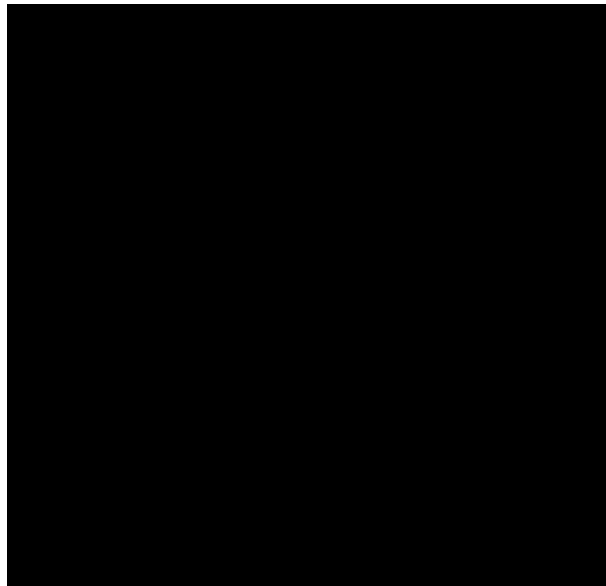


Figure 3.5: The Graphical User Interface

3.1.6. Image Capturing

During a data gathering session, we want to store the images recorded by the probe to a hard-drive. To make that possible, three components are required. Since recording is started and stopped by user input on the BedBasedEcho, the first component is one that receives messages in some way. Next, a separate component should gather images from the echo probe. It passes these to a third component, which stores them on disk. In the following sections, each component is described in detail.

Socket communication and recordings

To enable communication between the Windows machine and the BBE, the choice was made for a socket server over ethernet. On the Windows side, this required a socket server that would listen for incoming connections. The socket server uses blocking IO, so we run it in a separate thread to prevent interference with the other components.

When the bed component starts a cycle as described in section 3.1.7, the bed will send a start message to the socket server. That message contains a unique recording ID and a start time. These will be sent to a property manager, responsible for maintaining recording state and generating file-names. Upon receiving a start message, the property manager will start a stopwatch with the initial time set to the start time in the message.

The property manager generates file paths for the recorded images, which are stored in a folder corresponding with the cycle ID. The filename is simply the current time at the moment of recording. When a cycle ends, the BBE sends a stop message with the same ID as the start message. This allows the protocol to support multiple parallel recordings, although this does not occur in the current implementation. The property manager will stop a running recording if it receives a stop message with the ID of the current recording. All other stop messages are ignored.

Frame Saver

Our goal was to record the view of the probe at the highest frame-rate possible. The probe doesn't produce images with a consistent frame-rate, so setting a fixed frame-rate beforehand could result in duplicated or missing frames. Instead, we opted for an event-based method where the SDK would notify us when a new frame is ready. This allows for the highest frame-rate without duplicates.

The [illegible] SDK documentation provided a code sample for processing ultrasound images in real-time. [illegible] We extended the sample by additionally passing the image to a frame saver.

The frame saver will check the property manager to see whether a recording is in progress. When we are recording, it will enqueue the image asynchronously, together with the filename generated by the property manager, for later processing. The reason for this is that the SDK will only start recording the next frame after all callbacks have returned. Since the primary design goal is the highest possible frame rate, the majority of the processing work was moved outside of the callback function.

Image Processor

The image processor is the component responsible for actually saving the images to the disk. It listens on its own thread for tasks added to a queue. When an image-name pair is received, the image will be saved to a file with the specified name. We write to disk on a separate thread, since it is a blocking operation.

The frame saver will still be able to write images to the queue when the image processor is blocked. The queue will act as a buffer when the image processor temporarily can't keep up with the probe recording speed. The buffering should not take too long, because it is still running in a 32 bit process, which has a maximum memory limit of about 2 GB. The external disk we are using is able to keep up with the frame-rate of the probe, so for us this is not a problem. We do have to ensure that the disk is spinning when we start recording, since the disk's startup time exceeds the time we can buffer.

3.1.7. Usage in practice

This section will describe the workflow of a data gathering session in detail. A data gathering session usually consists of three parts: setup, random walk cycles, and post-processing.

Setup

In terms of setup, a number of steps have to be taken.

- An ethernet connection between the Windows machine and the BBE.
- A test subject on the bed with their optimal position roughly in the middle of the hole in the BBE.
- A started socket server on the Windows machine.
- The probe positioned on the chest of the subject, such that the image is perfect

When all these components are ready, we are ready to start our data collection session. This was described in more detail in section 3.1.2.

Post-cycle

After a sufficient number of cycles are recorded, the data has to be post-processed into a form that is usable for the network to train on. For this, a process called vector interpolation is applied to each frame of video.

The goal of vector interpolation centres on finding a "label" for a given frame of echo video. When this frame was taken, the probe was at a specific position relative to the goal. The vector from this position to the goal is the label for the image, as this is also what we want the network to produce as output: given an image, where do we have to go to find the heart? The problem is that there are comparatively few positional data-points saved on the bed: this only happens if the probe gets a new target to move to, which it does in a straight line.

Hence, the vector interpolation component first takes the instant at which the frame was captured, and looks for the nearest 2 positions that were saved: the one before, and the one after. Because we know the time of these 2 positions, and that the robot moves at a constant speed in a straight line, we can linearly interpolate the exact position.

After this, we calculate the vector from this position to the final stored position, which is defined to be the optimal goal that we want to move towards (every clip ends on the heart). Once we have this data, we are ready to proceed to the next component, the learning.

3.2. Machine Learning

In this section, the software components associated with our use of machine learning are described. Even though this project only considers one echo view, the machine learning setup should support as many as needed. Hence, modularity is important. This caused us to split the software architecture into two main logic components: one to transform any training data into an internally consistent format, and one to train the network based on this data. Note that this section does not extensively cover the machine learning models that were tried. Instead, these are described individually in appendix C.

To allow for modularity, an interface between the data gathering component and the machine learning architecture is necessary. This interface will be responsible for transforming the data format. Its output should be flexible, as different models could receive different kinds of labels. As the same data might be used to train multiple models, some kind of storage is necessary in between these components as well. For an overview of the component structure, see figure 3.6.

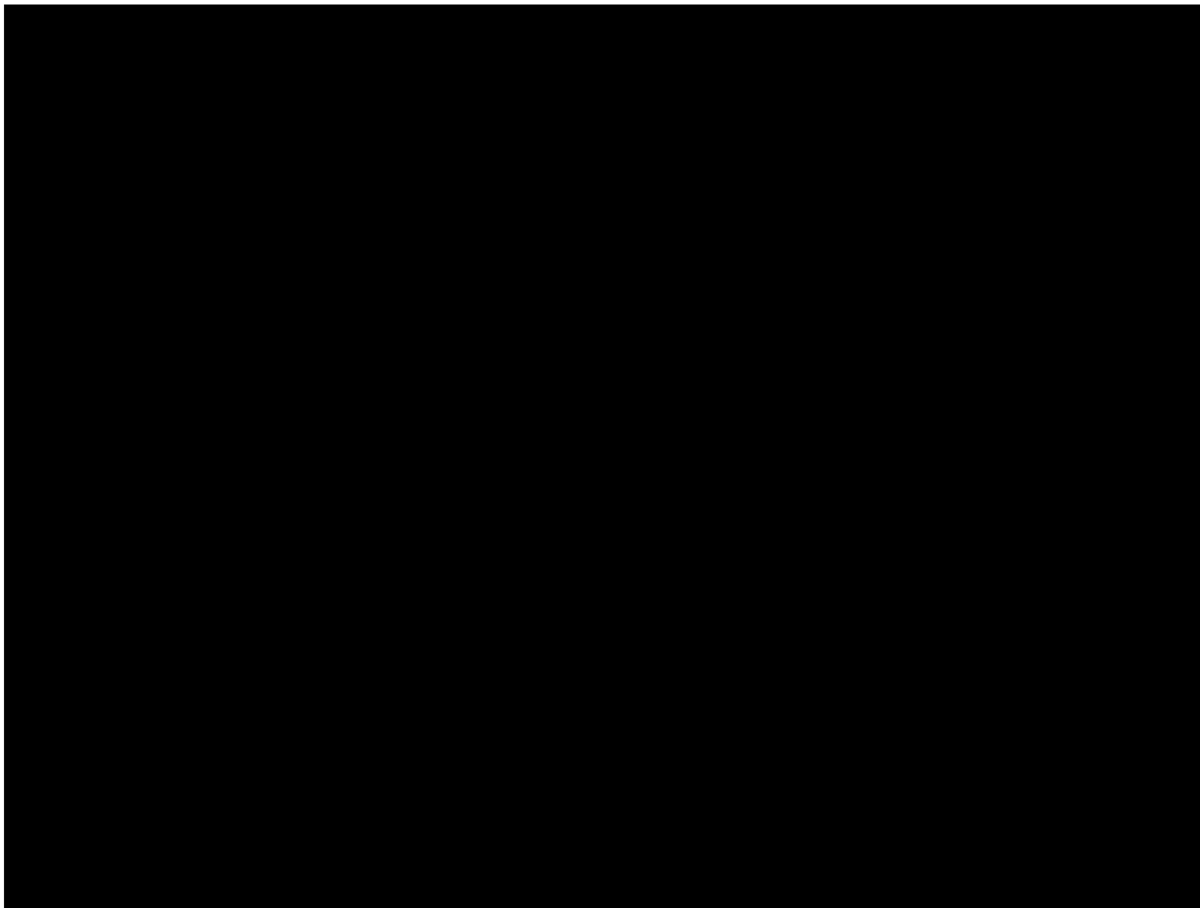


Figure 3.6: Component Diagram of the Machine Learning architecture

3.2.1. Data format/storage

The machine learning component should be compatible with multiple data sources. To allow this, we put restrictions on the format the data should be in. We also need to make sure that it supports adding other information (e.g. labels) and that it properly translates the data between the components.

Design considerations

The data storage and format have a few functional requirements:

- It should allow for storing large binary files, as the majority of the data stored will be image files.
- It should allow for relations between the image files and some label data, which can take multiple forms.
- Data should be separable based on specific attributes, since more views might be added in the future.
- It should allow for multiple learning strategies (e.g. regression, multi-class classification, multi-label classification).

A few options have been considered for the exact storage solution and format. A standard solution used within the machine learning community is allowing the directory structure to dictate the labels, i.e. every image with label X should be stored in a folder called X. While this is a viable strategy for single-label classification problems, it is flawed for regression problems, since labels are continuous, which will result in the data being overly segmented.

Another option would be to encode label information within the file name. This allows for the labels to support multiple types and amounts. Some issues, however, are that there will a small complexity cost, as the format of the labels is not immediately clear. Furthermore, issues arise regarding extensibility and consistency, due to it being difficult to constrain the exact format image names should have. Storing label data in the image name also comes with a small computational cost for retrieving, validating and parsing.

Two more options have been considered, which were based on popular database storage schemas. The first was document-based storage. This allows for easy separation (using collections) and extensible and flexible labels. It also allows for data analysis using multiple documents. However, the flexibility that document-based storage has, allows for separate documents to have different structures. While the flexibility of the storage solution is necessary to allow for new strategies and different kinds of data, samples do need to have the same labels in order to train a model with them. In a sense, this option is *too* flexible.

Finally we considered a relation-based solution. This allows for enough flexibility to expand the data for newer strategies, by adding new relations, as well as good separation between different relations. The largest downside is the difficulty to store binary data within a relation. However, due to how closely a relational based schema reflects the relation between an image and corresponding label data, it was the most appropriate solution.

Instead of storing images directly within the relational schema, we opted to store a reference to the image, saving the image itself to a file system.

Besides the image reference itself and the labels, there was also need to store extra information about the image, as they might belong to specific sessions, or are captured in a specific order. This needs to be stored alongside the image.

Implementation

As the current data only has a single relation, $image \rightarrow labels$, we opted to use a to store these relations instead of setting up The functionality that offers would be largely unused, as there is no need to serve multiple clients concurrently, nor are we cross-referencing information. The separation is provided by simply creating different for each group, as it is not possible for a single sample to belong to multiple groups (in case of views and similar criteria). The exact relation used for each image is We opted to use a natural key instead of a surrogate key, as the uniqueness of the data is guaranteed based on the We also do not need the performance boost of using smaller keys, as cross-referencing and selection are seldom performed. The validity of the file should be checked during the creation or insertion of the data, this allows us to prevent explicit checks during the reading of the labels.

3.2.2. Transforming to the internal format

The data transformation component should serve as a black box that interfaces between the data collection component and the machine learning model, by transforming the output from the data collection component into the internally consistent format.

Design considerations

This conversion process exists out of three major steps:

1. Parse the data from the collection component.
2. Transform the parsed data.
3. Write the data in the correct format to the store.

However, as the exact format to store the data in depends on the strategy for the learning architecture, the transformations should also be modular. We therefore provide an interchangeable and composable transformation component. The parsing and writing should be compatible with the majority of possible transformations. Also, they should not need to change based on the exact transformation used. Parsing the data depends only on the format provided by the data collection architecture and provides a consistent interface for the transformations. Writing the data to the store is slightly more complex, as this requires the output to be flexible enough to allow for multiple possible results from the transformations.

Implementation

A generic transformation interface from which all classes can inherit was implemented, the interface is set up in such a way that it both defines the output and the labels. The label writer then combines those appropriately. Besides these, simple transforms and a compose have been implemented. New transformations can be easily added, however, no restrictions to the output and input has been explicitly made, so it is the responsibility of the programmer to ensure the transformation can be composed.

3.2.3. Training a convolutional model

The learning problem boils down to predicting the probe movement based on ultrasound images. As it entails processing images, a convolutional neural network (CNN) seems most appropriate. A few variants of CNN are used to further improve the accuracy of the prediction. For the exact details on how these models have been created, see appendix C. Besides the different model architectures, we also used two different approaches: multi-label classification and regression. For both approaches the model structure roughly remains similar except the final layers of the network. The input, however, should stay consistent for each network, as the data should follow the interface described in section 3.2.1.

Input layer

There are two variants of input layers that are needed, one which takes a sequence of images, and one that takes an individual image. This is to allow for both a recurrent variant as well as a regular convolutional network. As the network itself should not be bound to the exact resolution and color type of the image, these details can be changed without much work.

Regression

Our problem fit naturally to a regression problem, as the movement of the probe is essentially a 6 dimensional vector, with all values representing movement on a specific translational or rotational axis. However, it is difficult to compare different dimensional vectors, as they might seemingly differ a lot. For example, there are different sets of rotations that have, when combined, the same effect: 90 degrees pitch, 90 degrees roll and 90 degrees yaw combined are equivalent to just 90 degrees roll. Hence, before using the regression data as labels, we normalized the translational vector (consisting out of a x, y, z component), added a distance value, and finally calculated a directional vector based on the Euler angles, using (α as the yaw, β as the pitch, and γ as the roll:

$$\begin{bmatrix} \cos(\alpha) & -\sin(\alpha) & 0 \\ \sin(\alpha) & \cos(\alpha) & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos(\beta) & 0 & \sin(\beta) \\ 0 & 1 & 0 \\ -\sin(\beta) & 0 & \cos(\beta) \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(\gamma) & -\sin(\gamma) \\ 0 & \sin(\gamma) & \cos(\gamma) \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} \tag{3.1}$$

Classification

The natural representation of the problem needed some conversion in order to create a classification problem from it. We decided to subdivide the translational and rotational axis into bins. To do this, the network classifies the situation of each axis [redacted] and [redacted] This classification picks one of multiple bins: how many of these bins are used drastically changes both the precision of the model, and the training difficulty. For example, 2 bins implies a simple decision for one axis: "go left" or "go right". 3 bins allows for a third choice, "no movement". Upping the bins to 5 provides more granularity, allowing for "far left", "a little left", "no movement", "a little right", and "far right". The number of bins is a trade-off between ease of training (few bins), and precision of output (many bins).

As translational axes need more granularity we decided to use 5 bins for the [redacted] axis and only 3 bins for the [redacted] axis. Furthermore, 3 more bins were created to classify the distance to the desired position. For every axis, a single sample should only fit in exactly

one bin, so for each sample and bin we use a one-hot encoding, to represent the label. We use as our loss function, and also track the total accuracy of the problem. For backpropagation we use a For the final layer we use a

CNN with

We used a simple network consisting out of the output is then fed through For the exact details, see appendix C. A small variation, while still using the same structure, Image feature extractors like ImageNet (Krizhevsky et al., 2017) and SqueezeNet (Iandola et al., 2016) have been considered, as they either perform well accuracy-wise or speed-wise.

CNN with

Temporal data is relevant for detecting the position of the probe, as attributes like heartbeats can contribute significantly to the prediction. might boost accuracy. This would entail adding The model would still need to infer the correct labels.

The Pipeline

A single model is composed of at least an input layer, and However, the exact model should be flexible: the architecture itself should allow for multiple variants, as that would ease the burden of trying different approaches. We divided the training pipeline into multiple stages, each having their own responsibility. A model has 4 components, a builder, a trainer, an oracle and a evaluator. The builder is responsible for creating the exact structure of the model: All these attributes are variable to a certain extent, while still being easily applicable to classifying ultrasound images. The trainer is responsible for the flow of the training process: The oracle is responsible for predicting specific images after the model has been trained. This component should provide some flexibility, as the exact details of the output and input are defined within the model itself. The oracle provides abstraction, such that the requested input mirrors the data format specified in 3.2.1, but the model can still process the data. Simultaneously, it provides the ability for the output to have varying numbers of labels. Finally, the evaluator allows for a rough indication of the performance of the model. These metrics are also tracked during training. For example, the evaluation allows for on data the model has never seen before, to estimate how well the model generalises to new data.

Implementation

For the exact implementation of each model, refer to appendix C.

3.3. Deployment

Deployment involves setting up a software solution that allows the trained network to make predictions in practice. Originally, our plan was to do this virtually, using a 3D visualisation of the AI's movement. However, as the project progressed and working with the BedBasedEcho hardware turned out easier than expected, this plan was adjusted: instead, we would attempt to physically deploy the trained network on the BedBasedEcho, such that it can move the probe based on what it sees. For this, a few components are required. From start to finish, we must pass images from the probe to the network, pass the network's answer to the BBE, and move the BBE robot arm based on this message.

Core Challenges

- The SDK only runs on Windows, in a 32-bit executable
- TensorFlow only runs in a 64-bit executable
- The bed only runs on Linux, since it uses ROS.

Architecture

To account for these constraints, we decided to divide our setup into four connected components.

First, the FrameSaver component performs almost the same function as in data gathering: it is written in C#, runs on the Windows machine, and collects images from the probe. These are passed to the second, TensorFlow, component, which is implemented in Python and runs on Windows. The resulting raw network output is sent using a third component, the Instruction Sender. Lastly, these instructions are received by one of two Autonomous Controllers, programmed in Python and running on the BBE. The autonomous controllers produce the actual commands that move the robot. An overview of these components can be found in figure 3.7.

In the following sections we will explain how each of those components will work and how they communicate, in the order they were introduced.

3.3.1. Frame Saver

This framesaver component is largely the same as the framesaver component used in data gathering (section 4.2).

Design

The TensorFlow component requires a rolling window of images. Because it runs in a different process than the probe, we need a fast way to transfer these image files between processes. Using Inter Process Communication techniques we can share information and files between these two processes.

Implementation

The frame saver is responsible for writing images to shared memory accessible by the TensorFlow component. Most of the functionality required here is already implemented in the frame saver, described in section 3.1.6. We only need the save location to be in memory rather than on disk. Note that we no longer need separate recording IDs. We completely

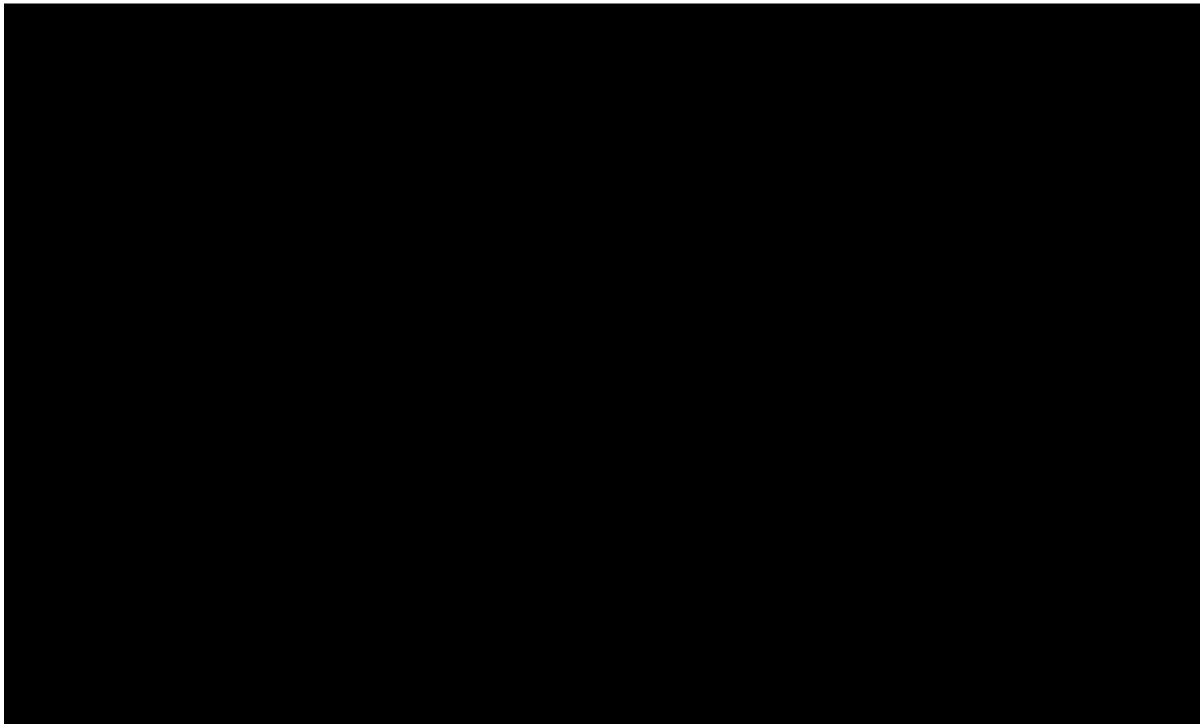


Figure 3.7: Component diagram of the Deployment Component

reused the property manager and the frame saver, but immediately tell the property manager to start recording, instead of waiting for a start command.

To this end, a new image processor has been written, responsible for maintaining the rolling window and writing it to shared memory. We use [redacted] to communicate with the TensorFlow component. [redacted]

3.3.2. TensorFlow component

The TensorFlow component should run image predictions based on the rolling window defined in the previous section. The results should be decoded and passed to the instruction sending component. Ideally, we run a new prediction every time that the rolling window moves. However, we anticipate that TensorFlow will not keep up. Luckily, not all images are required for correct movement, so the actual achieved speed is allowed to be much slower.

We initially intended to program the TensorFlow component using the official C++ TensorFlow library, or unofficial C# bindings. However, the C++ support for TensorFlow is very low level, and doesn't have most of the tools provided by Python. The alternative is also not an option, since the unofficial C# bindings were not updated for TensorFlow 2.0 yet. Lastly, we are forced to use kernel-level inter process communication regardless of the chosen solution, because you can't have a mixed 32-bit ([redacted] SDK) and 64-bit (TensorFlow) binary. If the images will have to be communicated across processes, we decided that we might as

well provide ourselves the convenience of writing this component in Python.

To load all images without problems, a mutex needs to be used in both the TensorFlow component and the frame saver. The frame saver already creates a kernel level Mutex, but reading that in Python proved to be difficult. Therefore, we couldn't load more than one or two images at a time before running into synchronization issues.

Luckily we've only tested models that take a single image as input during our deployment sessions. Therefore, This circumvents the synchronization issue. The TensorFlow predictions take on average twice as long as capturing a frame from the probe, which is sufficient for our purposes.

3.3.3. Instruction sending

The Instruction Sender is a simple roslibpy component with the sole task of passing instructions along, from the Windows machine to the BBE. The ROS Topic that this instruction is published to, has subscribers on the BBE in the form of 2 autonomous controllers.

3.3.4. Autonomous Controllers

The autonomous controllers are roslibpy nodes running in separate threads on the BedBasedEcho. Fundamentally, they transform raw network predictions to coherent move commands for the robot itself.

The majority of internal logic in these components deals with inaccuracies and noise in the raw output of the neural network. There are two variations of these autonomous controllers, which are distinct in the type of network output they expect. One is for regression networks, which try to output an exact vector to the goal. Such output only requires smoothing out and averaging to be useful. The other variation is for classification networks, which give simple hard answers. An example would be, "are we too far left, too far right, or in the middle"? This produces a very rough answer that gives us little information. Therefore, more post-processing is required to get coherent movement.

Regression Controller

The regression controller (RC) receives raw commands from regression networks one by one, in normalised form.

The first step is to un-normalise the commands and smooth them out somewhat. This is done by taking a simple average of multiple commands, called a 'batch'. From this point onward the batch is viewed as a single coherent command.

Next, we want to determine what move we would like to perform. There are a number of factors to keep in mind for this decision:

- The expected maximum FPS is but
-

- The network can be expected to perform worse in terms of absolute accuracy, the further it is from the heart.

To account for [redacted] we only send a movement command once a batch is filled [redacted] Every time we have a new batch, we look at the last few batches in another averaging operation. This is because even [redacted] commands averaged together could still be too noisy, especially far away from the heart. Depending on the estimated distance from the heart, we use more or fewer batches in this second average. Notably, we use a weighted average, that biases more strongly for the recent batches. This has a logical reason: more recent batches work on more recent data, and are therefore more representative. The produced effect is that we take a very broad, rough average if we are far away, which hopefully produces smooth, inaccurate movement that tends in the right direction. As we get closer, the average uses fewer batches, making it "sharper" and more precise, at the cost of smoothness. Within [redacted] we take no average at all, and just use the last batch as our direct answer.

Every separate move command is scaled such that, at a physical movement rate of around [redacted] the ordered move should be done by the time the next batch is filled.

Finally, we stop moving if the estimated distance goes below a fixed threshold, such as [redacted]

Classification Controller

The Classification Controller (CC) receives raw commands as well, but only from classification networks. It expects an output in terms of the chosen bins, one for each axis. These bins were described in more detail in section 3.2.3.3.

The CC accepts as input a list of choices, one for each axis, picking a bin the network finds most representative. The mean value of each bin has been set in the CC, so that when these bins are selected, we form an "expected" value for it. As an example, if the samples sorted into the "far left" bin on the [redacted] had a mean [redacted] value of [redacted] then this is the value we would use here as well, should this bin be selected. With this method we form a vector, stating the approximate direction to the heart. This vector is then used to create a batch, using exactly the same method as described in the section on regression. All further steps, eventually leading to movement, work the same as in the RC as well.

3.4. Testing

A crucial part of ensuring software quality is testing. It can prevent unexpected behaviour from occurring, and helps to verify intended behaviour. Hence, we ensured that all relevant code was tested in some form.

3.4.1. Automated Testing

As our project was set up using two main languages, C# and Python, multiple testing frameworks were used. Furthermore, to assure test quality, each test suite uses coverage analysis, and undergoes manual reviewing from peers during merge requests to prevent blindly trusting the metrics (Bouwers et al., 2012).

C# Testing

Tests were written using an xUnit style testing library called NUnit. However, as C# has harsh limitations on inheritance, i.e. the need to declare virtual methods if behaviour should be overridden, code had to be closely designed to be compatible with testing. This resulted in all code being programmed with interfaces abstracting behaviour, and dependency injection being used for multiple classes. This structure allowed mocking to be possible using the Moq testing framework. Other commercial alternatives for mocking were available, which did not require such strict programming to interfaces. However, as this also made our code more maintainable and modular, we decided to use the approach described earlier. To ensure that behaviour was well-tested, branch-coverage analysis was performed on the tests, reaching over 90% branch coverage. Code that was not covered either included external code, or SDK reliant code. However, due to abstracting most of the dependency on the SDK, the coverage missed was minimal.

Python Testing

For the Python components tests, were written using PyTest. Due to the flexibility of Python's type system and powerful reflection and introspection capabilities, mocking code was relatively simple. Most behaviour has been unit-tested, except launch scripts and external code. As we created abstractions for interaction with back-ends like Keras, most code was easily testable due to mocking external dependencies. Similar to C#, branch-coverage analysis has been performed to ascertain the quality of the tests, reaching 91% branch coverage.

3.4.2. Manual Testing

Not all our components are equally testable with unit and integration tests. Most importantly, the performance and prediction delivered by the neural network can vary greatly. To make sure that performance of this network was up to par, we have performed manual tests. These tests also ascertain that the whole system runs as expected, and no unexpected crashes happen during normal execution, mainly with respect to issues like memory leaks.

3.4.3. Software Quality Assurance

In order to keep the code maintainable and well-structured, several static analysis, codestyle and linting tools have been used. Furthermore, code also needed to be well documented and commented. While most of these requirements were already checked using automated tools, like Resharper, DupFinder, PyLint, Radon, etc, all code afterwards has been peer-reviewed extensively, with code reviews going up to 66 comments per merge request. Furthermore, an independent group (the Software Improvement Group) has performed some code evaluation, after which we received a review of the entire code base, identifying our great and not so great points. After the feedback, the code was adjusted to comply with most requirements. For the complete review on how we processed the feedback, refer to appendix A.

4
Challenges

The largest body of work in this project could be described as experimental research and development. As such, many of the currently functioning components had predecessors that, using different methods, were less functional. This chapter highlights key points in the development process where we discovered flaws in our plans, and how we readjusted accordingly.

4.1. Software Architecture

4.1.1. C++ (Testing) and CI

Problems arose when trying to test code that is dependent on the [redacted] This SDK does not support testing and is very difficult to mock, which is why we've decided to forego testing on purely framework-dependent code, which is written in Native C++. The higher logic, written in C#, can be tested.

Another problem is the testing that can be done on the data gatherer cannot be automated through Continuous Integration (CI). The TUDelft Gitlab supplies us with test runners for Linux only, which works fine for our Python implementations, but as Windows-based runners are not available, all tests of this C++/C# code have to be run manually.

4.2. Data Gathering

Although the ultimate goal of the BEP was to design and train a neural network, most of the actual work done was in creating a data gathering setup. The resulting data would then be used to train the aforementioned network. Setting up data gathering posed a number of unique challenges, as described below.

4.2.1. Gyroscope and Accelerometer

Initially, a plan was made to gather translational and angular positional data using [redacted] However, shortly after implementing a basic way to read out of these sensors using an [redacted], a flaw in this plan appeared: it seemed that the [redacted] was nowhere near as accurate as we needed it to be.

This flaw came from trying to extract positional data from acceleration data. The code initialised the position of the ███ as 0, 0, 0. Then, using acceleration read from the probe, the change in velocity could be calculated, which in turn could be used to update the position. The problem with this, is that small errors in the acceleration measurement and in the timing will cause the error of the position to increase exponentially, rendering the positional data useless within seconds.

This lead us to discover that this method was completely unfeasible. Our machine learning would need very precise data, so we experimented with two alternatives: Visual Motion Tracking (section 4.2.2), and moving the probe with the robot instead of by hand (section 3.1.3). Of these options, we chose the latter, while the former is described below.

4.2.2. Visual Motion Tracking

To get positional and angular data, we would put a large number of tracking dots on the echo probe, and externally record it while it is being used. We would then have to post-process the footage in existing motion tracking software, which would solve the motion of the dots into a coherent set of coordinates for every frame of video.

This option presented a few unique advantages:

- The probe can be operated by hand, which is much more natural for the sonographer. They can perform exactly the movements they would in real life.
- Motion tracking has the possibility of giving very precise results: the internal consistency of the model, measured in pixels of error, is typically around 0.5 pixels. This means that on a 1080p video captured at such a distance that the screen width is 100cm, a half-pixel accuracy can be viewed equivalently as a 0.26mm accuracy.
- No hardware or software setup would be required beforehand.

Video tracking was explored in a number of practical tests: small household items were 3D tracked using the VFX package in Blender 2.82a, using video from a phone camera. The pixel error was calculated, and the result assessed by eye. In this testing, a number of disadvantages were revealed:

- Video tracking is quite labour-intensive if tracking dots happen to get obscured from view, only to reappear later. This requires manually remarking them. It can also happen that fast motion or other interference causes Blender to lose track of a dot. This makes motion tracking quite doable for <5 min clips, ████████████
- The coordinates produced by the solve are not necessarily axis-aligned or scaled to match those of the robot. Each solve could end up using a different coordinate system. Ultimately, we want all of the coordinate systems to align with the real world as much as possible, because they dictate the image labels, which in turn impact the output of the neural network. If the labels do not correspond well with the real world because they are scaled poorly or off-axis, the same will happen to the output of our network.
- Getting data for each frame is not guaranteed. A blip in the camera footage, or the sonographer moving their body between the camera and the probe, or any other technical mishap could all lead to a complete loss of data for a few frames. We would

prefer a more systematic setup that can (to some extent) guarantee we find accurate data for every frame.

Ultimately, these negatives together outweighed the possible benefits, and we opted for automated movement by the robot.

4.2.3. Data gathering in practice

After agreeing to use robotic movement to steer the probe during data gathering, a first Data Collection Controller was written. Unfortunately, during our first data gathering sessions, multiple flaws were revealed in this controller and in other parts of the design.

Movement cycle

The first method of randomly moving the probe was a "sunflower pattern" (Vogel, 1979): a set of points arrayed in a circular shape that grows outwards infinitely. The points are distributed to cover the area as evenly as possible. The random movement came in cycles: starting in a "home" position, the probe would go to one of the points in the sunflower pattern (typically towards increasingly far points). Upon arrival, recording of the probe was started as it moved back towards the home position. When it came back, the operator had the chance to adjust the probe's position and angle by small increments, if needed. Finally, the recording would be stopped again. By pressing a button, this cycle can be repeated.

The largest problem with this strategy of movement is a lack of diversity: ideally, we want many points close the heart to be traversed in many directions, not just straight at the heart. This data would give the neural network some "bad examples" together with the good ones, which improves network performance. Also, if the movement is always in a perfectly straight line, there is the possibility that the "straightness" is interpreted to be the desired quality. A trained recurrent network could infer the direction of movement and achieve perfect accuracy by giving this movement direction as the label (since we are always moving directly at the heart). Finally, the movement strategy does not vary the angle at all, even though this is another key component the network has to get right in deployment.

All of these problems were addressed in a second version of the random walk cycle, as described in section 3.1.3.

Ease-of-use

Ultrasound imaging is normally done by holding the probe in your hand. As with any tool, an experienced user would feel the tool to be an extension of the hand. Mounting the probe [redacted] and providing a controller distances the user from the feeling of the probe.

An experienced cardiologist, who assisted us in finding the apical 4-chamber view, could not work with this controller, instead relaying commands to a third person who moved the probe as instructed. Despite many attempts with many different subjects, we were unable to consistently pinpoint the correct location and rotation for a proper view.

As previously listed in 2.4.2, these difficulties with finding the apical 4-chamber view moved us to change our efforts into finding the parasternal long-axis view instead.

4.2.4. Outcome of challenges

A week after the first data gathering session, we had implemented the new random walker and switched to the parasternal long-axis view. With these changes, and the practical experience gained in the previous week, we were able to find a good view, and gather data around it quickly and without much user input. In total, we gathered 355.000 images on that day, which we used

4.3. Learning

Finding an appropriate network architecture for the problem often is a process of trial and error. There are many hyper-parameters that can be tweaked, the format of the data can be adjusted, etc. We also encountered numerous challenges before finding an appropriate architecture.

4.3.1. Mismatch between perceived and absolute translational vector

Label data gathered from the probe stored the position based on the position however as the images were rotated, this did not reflect the actual translation vector on the image itself. The network was therefore rightly confused about learning the direction to move in, as it could not be extracted from the features of the image itself. To fix these, we performed an inverse rotation transformation on the absolute translation vector to find the vector corresponding to the image. The performance of the model improved significantly, having an increase in accuracy of 20% on all axes.

4.3.2. Noisy images

4.3.3. Hardware limitations and memory issues

The data gathered existed out of over 350 000 images. All these images do not fit within memory and should therefore processed in batches. Data generators can be used for this purpose, which only load the images corresponding to a specific batch into memory. While this solves the memory issues regarding loading samples. The amount of VRAM required to load the model was lacking for complex models, Hence, we opted to move towards simpler models instead.

4.3.4. Overfitting

An issue that often occurs during training of a model is overfitting. Seemingly this happened a lot in the earlier, bad performing models, as the mean absolute error quickly increased for the validation set. To prevent this we added dropout layers (Srivastava et al., 2014), without much success. The later models did not have a large issue with overfitting, however to at least be able to rewind the model back to before it overfitted, we implemented a checkpointing system, that saves only the best model during training and the final model after training has been completed.

4.3.5. Normalization of translational and rotational data

As the labels consist out of data that have different scales, i.e. translational and rotational data, a good normalization technique needed to be used to put all the data within the same range. While it was trivial to extract a directional vector from the translation vector, by normalizing the components, finding similar operations for the distance and rotation was more difficult. We opted to scale the total distance by the maximum distance the probe could travel, i.e. For rotation we transformed the euler angles to a directional vector by applying rotational transformations to the unit vector $(1,0,0)$.

5
Process

In this chapter, we will touch upon the methods used during development. Firstly, we will discuss the tools and techniques we used to drive the development process. Secondly we will cover the method of communication, both within the development group and with our client and coach. This also includes a section on every member's code contributions. Finally, we briefly discuss a non-disclosure agreement that had to be signed to access the SDK.

5.1. Tools and techniques

To support our development cycles, we have opted to use techniques to keep track of development, and tools to help us keep track of version control and testing. These can be considered must-haves for software development.

5.1.1. SCRUM

To keep our development in order, we made use of an agile software developing method, loosely resembling SCRUM. Due to the short time period of our project, we had decided that sprints should last half a week. As short iteration times typically introduce a large overhead in terms of time spent on meetings, we have decided to keep these meetings short and to the point.

Our fixed meeting times were as follows:

- Monday 10.00 - 11.00
- Thursday 10.00 - 11.00

During these meetings, we would both do a short retrospective and also plan the next sprint. Unfinished backlog tasks were forwarded to the next sprint and, if needed, tasks would be formulated more concretely or tasks would be added, usually quite abstractly in the beginning.

Product review did not regularly take place at set times. Instead, our client would sometimes join us during sessions, whenever we gathered data or tested deployment.

5.1.2. Gitlab

As the TU Delft hosts its own GitLab instance, we were expected to use this for version control.

We made use of the so-called 'Git Feature Branch Workflow'. As described by Atlassian (nd), this means the `master` branch should always contain working code, while development on new features takes place in so-called feature branches. When a feature is finished, a merge request is to be created, allowing other developers to review this code. This way, we always ensure we have a working copy of our code, whilst keeping new features separate from the main codebase until these are deemed finished and of high quality.

The agreement was made that code review should be done independently by two different developers, only giving their approval when they felt the code was of high enough quality to be incorporated into the `master` branch.

5.1.3. Continuous Integration

Continuous Integration(CI) provides us with automated code analysis and testing. As explained in subsection 4.1.1, automated C++/C# testing was unavailable. Only code written in Python was automatically analyzed and tested.

Our CI-pipeline contained four steps: Setup, static analysis, testing and a test coverage check. Setup sets up the cache, the latter three together produce one of these three outputs:

- Pass: All tests pass, no further work required.
- Warning: All tests pass, but the code quality is low or testing is insufficient.
- Fail: One or more tests fail.

Static Analysis

In order to guarantee readable and maintainable code, static analysis checks the code for two things: whether the programmer has followed the agreed-upon code conventions and whether the programmer has written code that has a low cyclomatic complexity (i.e., the amount of independent paths through a piece of code). Additionally, this helps us adhere better to the SIG analysis as described in Appendix A.

To force adherence to code conventions, we analyze our code using `pylint`, which will return a rating on a scale of 1 to 10. Code conventions are important to us, which is why we've decided that code rated lower than a 9 will produce a warning.

Low cyclomatic complexity is important for code maintainability, and in this case the lower this metric is, the better. We analyze our code using `xenon` on the strictest settings: It produces a warning when the program receives a grade worse than A. According to Radon (2020), a grade of A means a complexity no higher than 5.

Testing

Tests written for Python will be executed to test the program, this is further described in section 3.4. For this, `pytest` is used. This job fails if any of these tests fail.

Test Coverage Check

Code coverage is a measure of how many statements and branching paths were actually tested. High code coverage gives a greater guarantee that all program functionality is tested. For this, `coverage.py` is used.

Using the data from the previous job, this job checks whether testing can be considered sufficient. A warning will be produced when the average branch coverage of the code is below 85%.

5.2. Communication

At the time of writing, the university is still closed due to COVID-19. Physical meetings were discouraged, with the government and companies advising people to work from home whenever possible. For this reason, meetings were primarily done digitally.

5.2.1. In-group communication and cooperation

Initially, group meetings were held using Jitsi, an open-source software package, hosted by the TU Delft. Eventually, it was decided to switch to Discord, as most group members were more familiar with this software and it produced fewer technical issues. Meeting notes and any other documentation that was not protected by a non-disclosure agreement were all hosted on Google Drive. Additionally, a WhatsApp group was created for any communication not directly related to a meeting.

COVID-19 made things significantly more difficult. There were no physical group meetings until a few weeks in, with one group member not being able to attend any these meetings in person due to symptoms. The main drive towards meeting physically was for data gathering and deployment, which made working from home impossible in this instance. Additionally, the close quarters near the bed combined with the government-mandated 1.5m distance made development challenging.

As is quite often the case, group workload was not uniformly distributed, with some members eventually having done more work than others. For example, machine learning was probably the biggest time sink during the project, but knowledge of machine learning is not equal within the group, meaning the more knowledgeable members spent a large amount of time on this.

Despite this, the division of labour was as follows:

All Members

- Contributed to the report

Joey Haas

- Designed the GUI of the data gatherer
- Contributed to the probe interface used for data gathering and deployment

Rembrandt Klazinga

- Wrote the Random Walker (translation)
- Investigated Visual Motion Tracking

- Set up BedBasedEcho for deployment
- Wrote the Data Collection Controller and Data Writer
- Wrote the Autonomous Controllers

Nick van Stijn

- Wrote the Random Walker (rotation)
- Wrote the Data Collection Controller and Data Writer
- Set up BedBasedEcho for deployment
- Test subject for data gathering

Jasper Teunissen

- Contributed to the Frame Saver
- Set up CI with caching
- Contributed to final deployment
- Implemented memory mapped files

Ying Zhang

- Performed all machine learning research and implementation
- Contributed to the Frame Saver
- Implemented memory mapped files
- Ensured software quality in Merge Requests

5.2.2. Communication with coach and client

The communication with our client, Eelko Ronner, was mostly informal. This started out with a few meetings on Jitsi, getting to know each other. Later on, when starting the data gathering, he joined our sessions to help us out with cardiac ultrasounds and to give us an introductory lecture on cardiology. When COVID-19 lockdown rules were relaxed and restaurants opened back up, he even took us to have a more informal meeting over dinner. Just like with our core development group, we created a WhatsApp group with our client and other important parties to keep them informed.

Our communication with our coach, Marco Loog, was more formal and less frequent. During these meetings, hosted by him on Zoom, we discussed eventual problems we were facing. He was of great help to us, especially in readjusting our targets for the project and in fixing issues with machine learning performance. Any communication outside of these meetings was done over email.

5.3. Non-disclosure agreement

There were two reasons why a non-disclosure agreement(NDA) was necessary: intellectual property and patient data.

5.3.1. Intellectual Property

The document called for a broad non-disclosure clause on everything that can reasonably be understood to be confidential. In practice, this referred to the SDK, which was, by itself, protected by an NDA signed by our client.

When initially sharing the SDK files, care was taken to encrypt these files before we uploaded this a 3rd party source. While Gitlab contains some header files and interop libraries, the actual SDK libraries are not put into the repositories.

5.3.2. Patient Data

The second non-disclosure clause on the document was for GDPR-complaince (referred to as the AVG in Dutch). The initial plan was to use patient data for training purposes, which meant that any footage had to be re-encoded with `ffmpeg` to redact any personal data.

Eventually we decided to forego using patient data and use exclusively use data gathered ourselves using the data gatherer component.

6 Conclusion

In this chapter, we will assess if the requirements and design goals that were set at the start of the project, have been reached by the end of it. Based on our final deployment day, during which we briefly managed to see the network in action on the BedBasedEcho, we can determine how well the requirements are met. This is described factually in section 6.1, and analysed in more detail in section 7.1.

6.1. Results Based on Requirements

In order for us to consider the project a success, we need the requirements defined in section 2.4.2 to be met. To judge this, we will list these requirements below and estimate separately if each one was fulfilled.

- **The software must be able to read echo images from the probe on command, using the provided Software Development Kit.** The developed Frame Saver is capable of doing this at a refresh rate of █ FPS or higher.
- **The algorithm must generate instructions that would move the probe towards the Parasternal Long-axis view.** During the final deployment day, the algorithm was able to correctly identify where the probe should move next, as tested on four different cardinal directions. When allowed to issue movement commands, there was limited success in reached the target view. Further interpretation of these results can be found in section 7.1.
- **There must be a way to record the position and angle of the probe as it is suspended in the BBE.** This was successfully done using the Data Writer component, executed on the BBE.
- **There must be a way to record the aforementioned data in conjunction with echo images, such that the results are stored together in a synchronised fashion. This allows them to be used as training data.** This is done by the aforementioned Frame Saver, and the separate Data Collection Controller, which runs on the BedBasedEcho. By using tools developed for model training, this data can be synchronised reliably.
- **The algorithm must have low enough latency to run in semi-real-time: it is allow-**

able that movement instructions sometimes stop for no more than 5 seconds to allow processing of video data, such that the next movement may be determined. This requirement was exceeded by some margin: the complete deployment pipeline is able to run at ███ FPS, in real-time. At no moment is there a technical limitation that requires movement to stop.

- **There must be an application for the trained algorithm, either simulated or implemented in the real world, making the progress visual.** This was done successfully in the deployment phase, where live images of the chest were used to move the robot.

6.2. Results Based on Design Goals

We will list our design goals as defined at the start of the project. For each one, we discuss if we think these goals have been achieved.

Performance

The algorithm should run on a Ryzen 3 2200G. Because the prediction algorithm was run on a separate machine, it is currently not clear if this is the case. However, the algorithm did run on an Intel i7 7700HQ, which is very similar in performance. Furthermore, both machines have the same amount of RAM (8 GB). Because of these reasons, we consider this design goal to be met.

Accuracy

The algorithm should find the apical four chamber-view within 3 minutes for 80% of cases.
This design goal might have been somewhat ambitious. First of all, the algorithm has not been tested on anyone other than one team-member, the same team-member that the data was trained on. Moreover, the algorithm cannot find the trained view completely autonomously (see section 7.1).

Presentability

We need the product to appeal to individuals unfamiliar with echos / software engineering.
A Graphical User Interface (GUI) was designed to create this appeal. Also, by implementing the algorithm into the BBE, demo's can be given to indicate the potential of the product. We consider this design goal to be met.

Consistency

We cannot have the system stop in the middle of a scanning session, causing discomfort to the patient.
We have been careful to ensure the system has failsafes in place for this criterion. On top of this, we consider the different software components to be well-tested. At the time of writing, the system has never stopped in the middle of a scanning session. Thus, we consider this design goal to be met.

Maintainability/Extensibility

The software architecture should be both maintainable and extensible. By using libraries like roslibpy (Section 3.1.2.2), we have decoupled the separate modules (the existing code,

recording, and executing) as much as possible. Doing this resulted in a maintainable and extendable project. Therefore, we consider this design goal to be met.

Recommendations for further development can be found in section 7.2.

7
Discussion

In this chapter, we will first discuss our interpretation of the results, after which we will state our recommended further course of action for the product as a whole. Finally, we will discuss some important ethical aspects of the project.

7.1. Deployment Results

Overall, we consider the final deployment of our product a success, since it was able to prove the potential of the BBE, and able to fulfil all our success criteria. The deployment took place in stages. Firstly, the probe was placed in the wrong location, relatively close to the heart, and not allowed to move. When placed too far up, down, left, or right, the model correctly classified its location and indicated the right direction to move in.

Secondly, we allowed movement in steps: by periodically toggling whether the robot was allowed to move, we produced slower movement, in bursts. This gave us the advantage of stopping the movement when we judged the probe to be at the right view. This produced the most promising results of the deployment. The model gave directions that moved it in a nearly straight line to a decent PLAX view[1].

Finally, the robot was allowed to move without restriction. This did not produce the desired results. It seemed that the sternum, a dimple in the centre of the chest, acted as a trap for the model. The sternum often became partially visible, after which the model got confused and went *towards* it rather than away from it (which would have been the correct action).

There was a second issue with autonomous movement, namely that [redacted] was known to be poor, even beforehand. This meant that the model had great difficulty determining whether it had actually reached its goal. Even if it steered to the right spot, it could not "step on the brakes".

Hence, when moving fully autonomously, even if the probe came quite close to a good view, a lack of an autonomous "stop" command, combined with the sternum trap mentioned above, caused it to veer far from the heart. At this distance, there was no training data, so

[1] `https://youtu.be/WZ7-pRN4qHY`

the network had little chance of returning to familiar territory.

This list of findings leads to some foreseen future developments. Recommendations for these can be found in the following section.

7.2. Recommendations

While this project could be considered successful within its own scope, many improvements should be made in different areas in the future. We will provide recommendations which either improve the scientific value or the commercial value of the product.

7.2.1. Improving inference speed

First, we will discuss how to improve inference time. Currently the predictions can be performed in or near real-time. In case data becomes more complex (higher resolution images, more color channels, etc), the inference speed could drop below acceptable levels. In order to prevent this scenario, we give some suggestions on how to improve model inference.

Built-in pre-processing

Currently, images are pre-processed right before they are An advantage of this is that data gathered from different sources (e.g. from another probe) can undergo the exact same pre-processing procedure as the images currently used. However, as probes differ a lot from each other, even if the images experience similar pre-processing, they cannot be mixed in. An option would be to move the pre-processing to the probe setup. Many probe frameworks already provide some functionality for pre-processing images, such as despeckling. This might reduce the pre-processing overhead when inferring images. Besides the built-in functionality of many probes, the image capturing happens in parallel with prediction. This means that this burden is moved to another process than the inference. This might give a performance increase, as inference is currently the bottleneck within the architecture.

Real-time inference with

Freezing graphs

Freezing graphs might improve inference time, as there is less overhead when processing data through the network. Weights are frozen and therefore do not allow for possible adjustments. The overall size of the model is also reduced, as many hyperparameters are not required anymore.

Pruning models to reduce the number of neurons

Model pruning might be an option as well. Often, the model used has inactive neurons, which only encumbers the forward propagation through the network. Pruning would reduce the amount of neurons and therefore decrease the amount of calculations needed for a single prediction.

7.2.2. Data storage

The current data solution is scalable in case new views are added, as it would only require the creation of However, issues could arise if there was a need to centralise the database of images: it would then be possible for multiple devices to access the same data concurrently. Some guarantee for atomicity would be required. Furthermore, while data might be easily extensible for different views, there could be a situation where more variables are at play. In that case, a multi-relational schema would provide better support.

SQL

A suggestion would be to move the current data-store to a SQL-backed one, for example PostgreSQL. During this move, it would be advisable to introduce surrogate keys instead of natural keys as well. We believe that moving towards an existing database solution would improve the maintainability of the system.

7.2.3. Improving model accuracy

While we have trained a model which can make predictions about the movement of the probe, it is not sufficient for actual use within hospitals. This is due to the lack of precision in certain areas, as well as missing information for large subject groups. Furthermore, the consistency of the system is lacking. Since this product considers the health of humans, the performance of the current model is not acceptable.

Expanding the sample set

The main improvement that should be made is extending our sample set. Currently all data has been gathered on a single day, with a single subject. Different subject groups however might give drastically different results. We recommend that multiple ethnicities, genders, and health conditions should be taken into account when gathering a complete data set. However, it is hardly feasible to expect all edge cases to be taken into account. We, therefore, suggest an always evolving dataset, where the user of the product can provide their own samples, to improve accuracy in specific areas. Besides that, data should also be gathered in a more distributed time period, as things like environment and gel applied might influence the quality of the data.

Mixing

Include realistic temporal data

The pattern in which the data currently is gathered (random walking) does not completely reflect how the probe should move. Image sequences that are currently within the data set are therefore rarely seen, when actually deploying the software. A recommendation would be to also gather more natural movements of the probe, by actually simulating movements that happen during normal sonography. This might improve the results of recurrent neural networks. Another improvement for recurrent data, is to allow for more variable amounts

of time between images. Short time intervals allow the software to see how a single *location* evolves over time, while larger intervals show how the *probe* behaves over time.

Increasing the complexity of the networks

Our current setup did not allow for training extremely complex models, as the hardware we used for training could not cope with the amount of data that needed to be processed. However, using existing parts of well-performing machine learning networks can prove to be beneficial, should these hardware limitations be lifted in the future.

Deep reinforcement learning

The current scope of the project is mainly focused on the neural network itself. However, interesting research has been done on the application of Reinforcement Learning on similar problems (Milletari et al., 2019). Since the movement of an agent can be relatively easily mapped to a reinforcement problem, applying similar concepts to our problem might give better results.

Optimizing hyperparameters

Furthermore, using different hyperparameters or different optimization algorithms can give drastically different results. Some more trial and error might be needed to achieve satisfactory performance.

7.2.4. Human Computer Interaction

The product is meant to be used by medical professionals. Therefore, the software should feel intuitive to them. While we have not performed any physical tests with medical experts using the interface, we believe it to be severely lacking. To improve the accessibility and usability of the software, we provide some recommendations.

Improving the user interface

Currently the user interface is very bare-bones and suffices only for initial testing and research. In the case it should become a commercial product to be used by doctors and other medical experts, the interface would have to be made more user-friendly. Currently it would be difficult for someone without a computer science background to completely set up the project and run it on a local machine. Furthermore, components need to be started up separately. When integrating all components within the same user interface, creating a start up sequence would be advisable.

Improving manual controls

Currently, manual input is provided using with a custom control scheme. This can seem unintuitive to the user, as there is no natural mapping from the controller to the probe. Other control schemes could be tested, such as touch controls, a single joystick, or simply a mimic of the probe itself.

7.3. Ethical considerations

Developing a product for the medical field requires careful consideration, as software often influences the health and lives of human beings. Besides the implementation and design of software, there are many ethical considerations to be taken into account. We try to address the major concerns, which range from safety to privacy, within this section.

This section is split into two parts. First, we explain the necessity of our product from a moral standpoint. Then, we will try to evaluate possible ethical issues. Each part is subdivided in a few arguments examining different aspects of the product.

7.3.1. Positive influences on health care

Currently there many heart patients that cannot be appropriately treated due to a lack of diagnostic tools.

Availability

The main cause for the lack of tools is due to the limited availability of sonographers. There is a large gap between the number of sonographers and the number of patients requiring diagnosis. Our product reduces the gap by providing replacements for professional sonographers, thereby allowing health care to be more effective and equally available for all.

Affordability

In comparison to human labour, and expensive probe equipment, the product is affordable. It allows for high quality imaging by taking advantage of the fact that the patient lies on their chest. Furthermore, the client has chosen a mass-producible probe in consult with the probe manufacturer and a few medical experts. The initial research and development costs are also relatively low. The result is that diagnosis can be provided with a lower price tag than before. This would allow people in third world countries to take a test as well.

7.3.2. Possible concerns

Safety

As our product will be primarily used as a method of testing for specific health issues, which concerns human lives, it is crucial that tests are taken accurately and safely. Therefore, the product should not be used unless actually proven to perform similarly or better than a medical expert. Currently the BBE does not provide direct interpretation and requires a medical professional to interpret the data correctly. As the system allows for manual adjustments, and requires some human interpretation, similar levels of safety to current situations can easily be reached. As the apparatus does not directly endanger a patient condition, no steps have been taken to improve safety there. However, more autonomous version of the system that might be developed in the future need to take extra care to keep this concern in mind.

Responsibility

The product goal is to replace the need for a sonographer when taking quick-look ultrasounds. However, in case of a misdiagnosis, it will be hard for the machine itself to take responsibility. Often when software fails, responsibility lies with the development team. An issue with this, however, is that while we have knowledge about programming, our knowledge about ultrasounds does not rival those of the medical experts. We have consulted with medical experts about the process and implemented their advice. However, this does not remove all risk. As such we suggest that the tool should not be used as a complete replacement, but more as a tool to reduce the workload on sonographers. Also, it should be possible to make manual adjustments to the probe's location. This keeps the sonographer in the loop and avoids putting all responsibility on the machine.

Privacy

As the product will handle personal medical data, we need to carefully consider how to handle this data. While the software has the need for limited networking capabilities (in order to communicate between the components), we limit this to only a local network. Data should therefore never leak outside. Data will further only be stored temporarily within on the local system and should thereafter be moved to the intended storage location, already used by hospitals alike.

Glossary

A4C Apical 4-Chamber View. 3

BBE Bed Based Echo. 3–6, 36, 37, 39, 43

CI Continuous Integration, automated testing. ii, 10, 11, 26, 32, 52

FPS Measure of amount of Frames Per Second. 23, 36, 37

inference When a trained neural network provides a prediction, based on input. iii, 40

PLAX Parasternal Long Axis View. 1, 3, 5, 39

ROS Robot Operating System. 7, 10, 11, 13, 21, 23

roslibpy A bridging library for Python 3, that allows communicating with a ROS system without having ROS installed yourself. 10, 11, 13, 23, 37

SDK Software Development Kit, a set of tools to ease software development. In the case of the _______ SDK, it contains the libraries required to interface with the ultrasound probe.. 7, 8, 14, 21, 22, 25, 26, 35

SIG Software Improvement Group. 32, 46–48

sternum Long, flat bone in the chest, connecting the ribs.. 3, 39

A

SIG Feedback

A.1. Introduction

At the 75% mark of the project, the code was submitted for review at the Software Improvement Group (SIG). The results from this submission and our response are described from here up to section A.6. Section A.7 lists the final scores from the 100% submission.

Property	Rating
Duplication	4.9
Unit Size	3.9
Unit Complexity	4.5
Unit Interfacing	3.6
Module Coupling	5.5

Table A.1: SIG refactoring candidates

A.2. Duplication

Code Duplication is simply a measure of how often duplicate code is found in the project. When code is copied, the maintenance effort for fixing bugs or making changes increases. In our submission, we received 4,9 stars for Code Duplication, as we had two instances of code duplication. Both of these were removed in a refactor, which moved our FrameSaver code base from C++ to C#. This allowed us to use existing mocking tools, which replaced our own, hence removing one of the duplications. The second code duplicate came from running the same test setup in two tests. This was refactored to the better convention of using a setup method when testing.

A.3. Unit Size

Unit Size is gauged by measuring the lines of code (LOC) of all methods and functions in the project. Behind Unit Interfacing this metric is the second worst aspect of our project. There are nine methods with an unacceptable Unit size in the project, causing us to receive a rating of 3.9 stars. The problematic methods will have their functionality split, separated and extracted into additional methods. It is worth noting that the team has opted to ignore

some minor unit size violations, in favour of code readability. This avoids the "treating the metric" pitfall of software design (Fowler, 2013).

A.4. Unit Complexity

The Unit Complexity analysis makes use of the McCabe complexity measure to grade the projects Unit Complexity. Complex code has more execution paths, making it harder to understand and requires more test cases. In our submission, we received 4.5 stars for Unit Complexity, with two problematic methods, present in the FrameSaver component. To fix these, we extracted some functionality present in the problematic methods to separate methods.

A.5. Unit Interfacing

The Unit Interfacing metric is determined by the number of parameters a method or class constructor has. SIG deems 0-2 parameters to be low risk, and 3-4 parameters to be moderate risk. Unit Interfacing was the lowest scoring metric in our submission, resulting in 3.6 stars for the project. This score came from multiple sources: a few methods were obviously sub-optimal, with one taking 10 parameters as input.

A larger number of methods were still very much suited for improvements, having around 5 parameters and decently high complexity.

Finally, around 15 methods had a parameter count of 3 or 4, and a low cyclomatic complexity and line count. These were lowest in SIG's list of refactoring candidates, and we applied our best judgement to these cases, choosing to leave most of them as is. To explain our reasoning for this we can look at an example. The utility method "limit" takes in the number we want to limit, as well as two other parameters: the maximum and minimum value used when limiting. This allows you to ensure a number is capped within the given range. To make this method risk-free, we would likely have to encapsulate the min and max into a tuple or other data structure. This would improve our Unit Interfacing score, but we think it would ultimately harm the readability of our code: well-documented "max" and "min" parameters are clearer and quicker to use than a tuple.

On top of this, many of the smaller violations are in Python, which supports named default parameters. Where possible, we have opted to make all parameters named, such that a programmer using the method, can clearly see which parameters are which. This avoids confusion with parameters ordering.

A.6. Module Coupling

Module Coupling is determined by the number of incoming dependencies in a module. SIG has drawn the boundary for acceptable MC to be 10 dependencies per module. In our submission, we received 5.5 stars for Module Coupling, as there were no modules with a coupling higher than 10, both in the Python and C++ components. We are obviously quite happy with this score, and can see no way to improve it. We will try to keep all dependencies below 10 for new files as well.

A.7. Second Feedback

At the time of writing, this feedback has not been made available.

Property	Rating
Duplication	TBD
Unit Size	TBD
Unit Complexity	TBD
Unit Interfacing	TBD
Module Coupling	TBD

Table A.2: SIG scores after refactor

B

Project Description

The following text was taken verbatim from this project's description on Project Forum[1]

Project proposals for a BEP project - information technologie/it and should keep 5 busy for ten weeks.

As a cardiologist in Delft, I hereby submit a proposal for a Bachelor's Final Bachelor Project (BEP) for the first half of 2020 (until summer).

As known, and from a personal experience it is a serious problem that there are too few people and means worldwide to diagnose heart disease well, quickly and cheaply. We have therefore robotized an echo our workhorse, the echo of the heart, with a number of enthusiastic TU Delft Bachelor students (Nick v Stijn, Nick Yu). Our prototype will be presented on 24 January. This is a robot product for short echoes in between, at night in an emergency, pre-operatively, or echoes used as the first screening tool, for example.

In terms of software, we face challenges that are tackled to a small extent by the participating IT bachelor students in the robotics project.

The challenges ahead in terms of software is of a scope and depth that they are suitable for a BEP project I believe (in cooperation with the students involved.. I am happy to explain that to you. Furthermore, it would of course be nice if it could be offered as a BEP for the group that will graduate at the end of the summer (and perhaps especially interesting for the current students to continue with).

Specific problem: Current robotics product with the two students from your department, makes 1 kind of images from 1 direction of looking at the heart.

Ultrasounds are made via standard routines in multiple directions, via 3 spots on the chest. Various settings are varied (measurements in 2D, doppler, "color flow" for example). In certain situations (a narrowed valve, fluid around the heart) extra determinations are made, whereby measurements are needed to take part in calculations.

BEP proposal; Extending current software with pattern recognition and complex adjustment of echo probe settings, with the aim of generating an optimal data set for images. (So

[1] `https://projectforum.tudelft.nl/course_editions/19/projects/721`

focused on image acquisition optimization). I would like to discuss details of wishes / specs if the possibility of a Bachelor's final project is possible and with any input.

In addition; pathology is usually from top 5 diseases. (mitral valve leaks, aorta valve calcifies, walls movement decrease, heart or compartments thickened/widened and right sided heart abnormalities, on it's own, or lung-related). For each case, different heuristic steps and measurements are of importance and one finding leads to cascades of events, and thus echo steps to be taken, leading to a check on findings (one finding should be logical related to other findings in other views. Else, a warning for inconsistency). It is with either LUMC or Sorbonne data that we can work. (we hear this week) Else, with Reinier de Graaf data.

Other information As a startup we are in the process of first preliminary funding and creating a team; no funding yet, minor expenses can of course be covered by me.

C

Model details

REDACTED

Info Sheet

Issued by: Zoekbeter BV
Final presentation date: 02-07-2020

Description

The core challenge of the BedBasedEcho BEP project is to create an algorithm to find the heart, and apply it on a robotic echocardiography solution. The team has found multiple complex solutions that are related to this problem, and has extracted useful information from these solutions to apply to this problem. However, some of these complex solutions were too complex, causing the team to run out of physical resources, or to have the solution fail entirely. By taking a step back, and simplifying the solution, the team has managed to create a system that performs marginally better than the complex solutions. The designed product consists of three major components: the data gathering, the learning, and the deployment. When used in this order, the result is an algorithm that can predict which way it should move to gain the optimal view of the heart. The algorithm will be used as a component in a larger automated echocardiography system. Ultimately, the algorithm showed promise by autonomously finding a good view of the heart.

Members

Name: Joey Haas
Interests and experience: Embedded Systems, IT Security, FinTech
Contributions: GUI, Ultrasound Interfacing

Name: Rembrandt Klazinga
Interests and experience: AI, Automation
Contributions: Data collection, random walk algorithm, robot controller for regression and classification networks, deployment

Name: Nick van Stijn
Interests and experience: Robotics, Founding member of BedBasedEcho
Contributions: Robotic control, Data Collection, Prediction Handling, deployment

Name: Jasper Teunissen
Interests and experience: CI/CD, Concurrent programming, caching
Contributions: probe-bed communication, main loop deployment, ultrasound image gathering, CI, static analysis, caching

Name: Ying Zhang
Interests and experience: AI, Embedded Systems, Legacy system interaction
Contributions: Machine learning (models and architecture), ultrasound probe interaction, data storage and transfer, software quality assurance.

Client: Eelko Ronner
Affiliation: Founder of Zoekbeter BV and BedBasedEcho

Coach: Marco Loog
Affiliation: EEMCS, Intelligent Systems

Contact: Eelko Ronner
Email: eronner@icloud.com

The final report for this project can be found at: http://repository.tudelft.nl

E

Research Report

E.1. Introduction

E.1.1. Problem definition

Nowadays, medicine for cardiovascular disease can be as cheap as five euros per month. For just these five euros per month, the life expectancy and quality of life of people with cardiovascular problems can be extended and increased greatly. There is a catch, however. Who do we administer this medicine to? To find out if an individual needs this treatment, we need to diagnose them first.

Currently, the most accurate option is echo-cardiograms, however, creating one is a time-consuming process, and there is a large shortage of sonographers, not only in the Netherlands, but worldwide. There are other options but these cannot replace the accuracy that an echo-cardiogram provides. Due to the shortage, doctors are forced to use these less effective alternatives, while they actually would like an echo-cardiogram to be made.

This is where the BedBasedEcho (BBE) comes into play. The BBE is a robotic solution f it can find a certain view of the heart automatically, it can reduce the currently large queue for echo-cardiograms in size. Currently, the search algorithm implemented in the BBE is based on a heuristic. It estimates the quality of the current image using a neural network, and switches between searching modes like: "lawnmower", "angle-vary", and "gray-scale" based on this estimation. Its ability to find the targeted view (the apical 4-chamber view) is moderate at best.

Now, the client needs a more accurate and consistent approach to find the heart, based on the images that we gather. Furthermore, since integrating it into the robot might lie outside of the scope, the client would like some way of visualising the result of the algorithm.

This leads us to our problem definition. How do we consistently find this apical 4-chamber view of the heart? More specifically, given an input consisting of image data, how to we extract instructions to move the probe?

Very generally, the current plan includes gathering expert image and motion data. By using a we plan to gather motion data from a trained professional. This data will then be used by pre-processing and machine learning technologies combined to decide the optimal next move.

E.1.2. Target Audience

This report is intended for interested individuals with prior machine learning experience and some rudimentary medical knowledge.

E.1.3. Current State

There is a small existing body of research around the topic we are studying. Notably, we could not find any previous project that matches with our goals of robotising an echo probe *and* looking at the heart. Some existing projects have overlap with our work:

- A software system was developed by Mustafa et al. that finds the liver and scans it completely, and autonomously. This system was successfully run on a 6-axis robot arm. Notable software features include finding the liver using echo images, and applying sufficient but not excessive force as the patient breathes.
- Van Woudenberg et al. developed software that is capable to identify one of 14 cardiac views, as well as determine the quality of the current view.
- More specific research exists that can help us with subsections of the project. This includes cardiac edge detection (Ketout et al., 2011), ultrasound denoising (Tay et al., 2010), and user guidance in echocardiography (Toporek et al., 2019).

E.1.4. Design Goals

In section E.1.5, we will define the requirements for the project. To be able to infer these requirements we will first define our design goals. This section lists these goals and explains them very briefly.

Performance

We want the software to be able to provide a robot with instructions in a reasonable amount of time, such that the patient is not waiting on the robot to finish calculations. This means that it should run without issues on a Ryzen 3 2200G, and should not consume more than 8 GB of RAM at any point in time.

Accuracy

To ensure that the product is worth using over manual work, we need it to finish operation within a reasonable amount of time. The goal is to find the apical four chamber-view within 3 minutes for 80% of cases, at minimum.

Presentability

We need the product to appeal to individuals unfamiliar with echos / software engineering. A separate visualisation tool that shows simulated movement can make the product more attractive, and simultaneously allow us to check the behaviour of the algorithm more easily.

Consistency

We cannot have the system stop in the middle of a scanning session, causing discomfort to the patient. We allow some crashes, should they appear not fixable, but the system should be able to restart itself. Barring issues outside our control, such as power or component failure, the software should never stop the procedure unintentionally.

Maintainability/Extensibility

Optimally, the software will be used in the final version of the BBE. This means that the software architecture should be both maintainable and extensible, in case a future development team continuous work on the project, perhaps to add more features.

E.1.5. Requirements

We formulate the requirements using the MoSCoW method, meaning all requirements are categorized as either a "must have", "should have", "could have" or as a "won't have".

Must have

- The software must be able to read echo images from the probe on command, using the provided SDK.
- The algorithm must generate instructions that would move the probe towards the apical four-chamber view.
- There must be a way to track the position and angle of the probe as it is being used by a person: this is to gather training data only.
- There must be a way to record the aforementioned data in conjunction with echo images, such that the results are stored together in a synchronised fashion. This allows them to be used as training data.
- The algorithm must have low enough latency to run in semi-real-time: it is allowable that movement instructions sometimes stop for no more than 5 seconds to allow processing of video data, such that the next movement may be determined.

Should have

- There should be a way to identify when the optimal apical four chamber view is reached
- There should be a separate software component that can visualise the movement instructions

Could have

- The algorithm could be further extended such that the movement instructions actually work on the physical machine: an interface would have to be made that supports this conversion of commands.
- The algorithm could be optimised such that the movement instructions run real-time. That is to say, the instructions are generated with such low latency that movement never has to stop.

Won't have

- The algorithm will not be able to change to colour doppler mode when the apical four chamber view is reached.
- The algorithm will not set color boxes around key features of the image.
- The algorithm will not consider views other than the apical four-chamber view.
- The product will not have explicit support for other any operating system other than Windows 10

E.1.6. Success Criteria

The design goals and requirements will be guidelines for the success of the project. These criteria have been decided in consultation with the client. For the project to be considered successful, we require that all "must haves" should be met, as well as most design goals. We put emphasis mostly on the software quality, however, due to the demands of the client, we also take presentability into consideration. So, we would like the visualization software to be included in the success criteria as well. However, due to limitations in resources, some concessions have to be made. Therefore, while we do strife to create a general applicable algorithm, we take into account that it will be difficult to apply the algorithm to females and specific edge cases in the initial development. This project can be seen as mostly a proof of concept and exploration of the possibilities.

In summary, the project is considered a success when we can lay the foundations for a maintainable algorithm can consistently and accurately find the apical four-chamber view, and this search process can be captured visually to appeal to the client.

E.2. Hardware Components

E.2.1. Gyro and Accelerometer

We will train our algorithm with video data from a human using the echo probe. However, in order to collect additional training data, we also want to track the angle and position of the probe while it is in use. This allows us to directly correlate visual data from the echo with it's exact orientation in space, which is hopefully very helpful for any algorithm we attempt to train later on.

Gyroscopic and acceleration data can be gathered in a number of ways. The simplest would be to use our own phones: apps such as “Physics Toolbox Sensor Suite" can directly record the data from these sensors to a readable file format. However, we quickly discounted this as an option, because the angle seemed to fluctuate with an error of +-4 degrees, which we deemed to be too noisy. Another issue is that a phone is quite large and heavy, making use of the probe uncomfortable for extended periods.

As an alternative, we can also purchase these sensor chips directly. This would allow us to pick a possibly higher-end component (giving us more accuracy), while remaining low weight and low cost overall. Chips that stands out for us are the MPU-6500, and the MPU-6050 produced by InvenSense. They contain gyroscopic and acceleration sensors and are purpose-built for motion tracking.

Because of the lightweight solution these chip offers, we elect to use one of them in the project. Because of their low cost, we opt to try them both and see which trade-off we want to make, implementability or update rate.

	Smartphone	MPU-6500	MPU-6050
Cost	−−	++	++
Accuracy	+−	+	+
Implementability	−	+	++
Update rate	+−	++	+

Table E.1: Comparisons between the investigated gyro/accelerometer combo's

E.2.2. Microcontroller

To read data from the MPU-6500/6050, we need a microcontroller that can power it and interface with the data pins. The microcontroller must be lightweight, similar to the sensor chip. The first choice would be one of the multiple Raspberry Pi's we already have from other projects. Closer inspection of the MPU-6500 reveals that there are 10 different pinouts we would have to interface with, which eliminates the use of a smaller Raspberry Pi board. A model like the Pi 2B could be used, but would certainly be overkill for the simple use case we have. Instead, we have opted for a smaller board that still fulfills the functionality requirements: an Arduino Nano. As the name might imply, this board has a small form factor, making it easier to mount on the probe without causing ergonomic issues. Additionally, Arduino's are often used for lower-level applications like this one, meaning there is plenty of existing documentation on reading out sensor chips, for example.

	Rasberry Pi 2B	Arduino Uno	Arduino Nano
Cost	–	+	++
Ease of implementation	–	+	+

Table E.2: Comparisons between the investigated microcontrollers

E.2.3. Echo Probe

The probe is a The applications of this probe include abdominal imaging and cardiology. The driver hardware(beamformer) is a which has support for most common types of ultrasound imaging, including B-mode and color doppler imaging. We will use this probe because it was the choice of our client.

Nevertheless, these properties make it ideal for cardiac ultrasound imaging.

E.3. Software Components

The following section will describe the team's considerations regarding the algorithms to be used.

E.3.1. Pre-processing

Some amount of pre-processing may be required before we feed the data into our main algorithm. The original echocardiographic images could contain more information than is needed. To remove the unnecessary noise, the useful information would then have to be extracted. This can be done in a few ways, and the ways described can also possibly be combined. However, these methods might not be necessary: nowadays, image pre-processing is becoming obsolete, as deep CNN's will just "learn" the pre-processing if it is needed. We will still consider pre-processing as there might still be performance increase to be gained from it.

It is worth noting that echo data has a very low Signal-to-noise (STN) ratio, meaning that there is a lot of noise and so-called snow. Often, traditional edge detection algorithms perform worse on low-STN data. Hence, if we do use feature detection, it would be more optimal to use purpose-built algorithms, as described below.

Denoising and Despeckling

Because of the low-STN nature of echocardiographic data, we cannot apply an edge-detection algorithm directly to it. We need a way to bring out the contrast between walls and emptiness. This operation is called denoising (or despeckling). For all denoising algorithms described below, it holds that no open-source implementation was found.

The first despeckling algorithm we will inspect is the Squeeze Box Filter (SBF). Proposed by Tay et al. in 2010, SBF was specifically developed with ultrasound in mind. Performance of SBF has been empirically tested on ultrasound data and has shown to outperform various standard algorithms in this use case.

The second algorithm to consider is the Multi-Valued Neurons Cellular Neural Network filter (MVN-CNN) (Ketout et al., 2011). This filter has the effect of very accurately turning white walls of the heart into white pixels, and components like the atria as black pixels. this makes it so that when we apply a fuzzy edge detection algorithm, we end up with relatively accurate representations of the edges of the heart (also described by Ketout et al. and mentioned further in section E.3.1.2).

The third method uses a genetic algorithm-based mixture model (Uddin et al., 2016). However, by visual inspection, it can be determined that this method preserves more detail than other methods, making it less suited for our purposes: Ultrasound despeckling can be done with the aim to improve diagnosability, but in our case we want to despeckle to determine our general location, which will likely benefit little from very detailed images. The large-scale structure is more important.

Finally, the manufacturer of the used echo probe described in E.2.3 has also implemented some speckle reduction methods into their SDK. These methods are described in section 10.1.13 of their software user manual. The methods described include their proprietary
technologies. These are not well documented in the manual, and will have to be tested to give an indication of their efficiency.

	SBF	MVN-CNN	
Estimated Effectivity	+	++	−
Implementability	−	−−	++
Speed	+	−	+

Table E.3: Comparisons between the investigated denoising methods

In practice, the best approach will probably amount to a combination of the integrated
methods, combined with the squeeze box filter, or another fairly implementable despeckling algorithm.

Edge Detection

For the edge detection, we have found a number of options: Canny (using the Canny operator or otherwise), Differential and FCNN.

Firstly, the Canny edge detection algorithm is discussed. The Canny edge detection algorithm can be run with a multitude of operators such as:

- Canny
- Deriche

- Sobel or dimension extended Sobel (Abdul and Lateef, 2019)
- Prewitt or dimension extended Prewitt (Abdul and Lateef, 2019)
- Roberts Cross

The most interesting of these are extended Sobel and Prewitt. By extending their 3x3 neighbourhood to a larger neighbourhood, we trade resolution for more continuous and thicker edges. This is preferable for our use case, since we only want to highlight the most apparent edges in our low-STN images.

Whether Canny edge detection with any of these operators is a suitable solution will have to be tested on actual data. That way their individual performance can be seen, compared to the other options.

The second option is differential edge detection. This edge detection method is able to detect edges with sub-pixel accuracy by computing a second-order differential equation. Differential edge detection does not have any other benefits over Canny-like methods other than this sub-pixel accuracy. For our use case, we don't need this accuracy at all, so we discard this method.

Lastly, we will discuss Fuzzy Cellular Neural Networks (FCNN). FCNNs have proven to be more effective than traditional edge detection methods when the data is low-STN (Yang and Yang, 1997). Especially because our data is low-STN, this method could work very well. An example of FCNN used on echocardiograms specifically shows that it has high potential to accurately highlight important edges, with little to no false positives (Ketout et al., 2011).

One downside of FCNNs is that up until this moment in time, an implementation in a major computer vision library has yet to be found. This would mean that the team would have to implement this complex algorithm themselves, which might lie outside the scope of the project.

	ext Sobel	ext Prewitt	FCNN
Estimated Effectivity	+	+	++
Implementability	+	+	−−
Speed	+	+	−

Table E.4: Comparisons between the investigated edge detection methods

E.3.2. Learning Architecture

The software should be able to route the probe such that high-quality ultrasound images are acquired. More traditional image processing techniques for finding the instructions for the probe based on the general position of the heart have proven unsuccessful in earlier endeavors. Therefore, we look towards machine learning architectures to provide us with a solution. After researching the application of machine learning on the guidance of ultrasound equipment, we identified three major architectures. Due to the usefulness of a convolutional neural network when dealing with image data, this model is used in all three major architectures. The first one uses just a CNN. The second architecture combines a CNN with a RNN. The final architecture uses deep reinforcement learning. Note that the latter solution differs a lot from the first two, which will be discussed in more depth in E.3.2.3.

Convolutional Neural Network

Convolutional neural networks are often used when dealing with image data. As the majority of the input of the algorithm will be an ultrasound image, they could provide a solution. However, as our problem considers more than just the image, it would ideally also to take the position of the probe into account. In the approach used by Toporek et al. (2019), optical recognition is used to track the 6-DOF of the probe. The 6-DOF are all based on a relative location, which has been marked by the sonographer. As we do not have the hardware needed to do optical recognition, this should be replaced by another technique, for which we propose using the gyroscope and accelerometer. Besides the required meta information, the ultrasound images should be labeled based on their quality. As this requires medical knowledge, an expert would perform this labeling.

The convolutional model itself exists of multiple components. Toporek et al. (2019) describe using SqueezeNet as a primary feature extractor with eight fire modules, followed by one convolutional layer and a global average pooling. To predict the instructions, two separate regression layers with a $\pi \tanh$ activation function were added after the primary feature extractor. Furthermore, for the quality assessment and view classification, a softmax classification layer was added.

This approach is close to the desired solution. However, there are some notable differences. First, due to the lack of optical recognition hardware, another technique should be used to track the 6-DOF, as suggested before, this could be done using the gyroscope and accelerometer. Second, the approach used is generalized for multiple views used; they consider apical two-chamber, apical four-chamber, parasternal long-axis, parasternal short-axis, subcostal long-axis, and subcostal four-chamber. For this project, we only consider apical four-chamber views, and therefore the model should be adjusted to not classify the view. Since the final layer that is used for view classification is also relevant for quality assessment, no drastic changes to the model should be necessary.

Summary of CNN

Data	Usage
x,y-coordinate	This is used to find the relative location of the probe, this is used as the expected result during learning only. (for example as data for backpropagation)
Rotation	Similar to the x,y-coordinates, this is used as the expected result during learning only. (for example for backpropagation through the model)
Ultrasound image	This is used as input for the convolutional network.
Quality score	This represents the quality of the image, this is used as the expected result during learning only, (for example for backpropagation)

Table E.5: Necessary components of the data

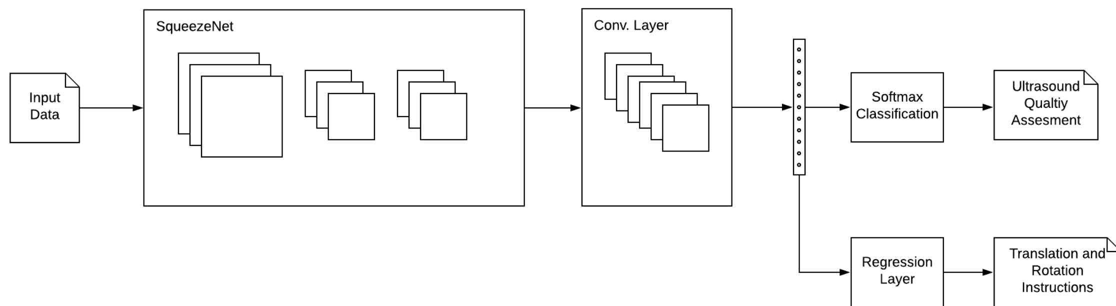


Figure E.1: Architecture described by Toporek et al. (2019)

Recurrent Neural Network

Creating an echocardiogram might benefit from using former predictions (Toporek et al., 2018). Therefore, the usage of a Recurrent Neural Network would be a possibility. We examine the approach used by Van Woudenberg et al. (2018). However, as this approach is mainly focused on assessing the quality of an ultrasound image, methodologies for extracting probe guidance instructions are not mentioned.

Van Woudenberg et al. (2018) use feature extraction by a neural network, which is fed forwards to an RNN. The RNN architecture in question is Long Short Term Memory (LSTM), a very popular architecture that lends itself well to videos. To make a single complete classification, Van Woudenberg et al. work with 10 frame clips. Each frame is classified separately with the LSTM network, which has a memory component that allows it to remember previous classifications in detail. This gives 10 separate verdicts on what view is seen in the 10 frames, and what quality this view has. Both of these factors are averaged together to produce the final classification of the whole clip.

Since our probe will also provide sequential video data, we believe using this temporal component in classification is certainly attractive for our purposes. It is notable that Van Woudenberg et al. had to perform feature extraction with a CNN, which in turn was fed to the RNN. Likely, the same architecture would be required in our case, as using only an RNN does not give the network enough depth to properly analyze the highly detailed and possibly noisy images. A key difference between the work of Van Woudenberg et al. and our own is the situation in which the RNN is used: they provide it with an echo view, and the only task for the network is to choose one of 14 preset options, and determine how good the quality is. In our case, the probe could be in any orientation around the heart, possibly so far that the heart is barely on screen at all. Our RNN would have a more abstract task to perform, namely reasoning about its position, and then reasoning the direction in which the heart is likely to be. Once it is close to a good view, this prediction would have to similarly increase in accuracy so as to end up with high picture quality. We conclude that if we were to use RNN's, they would need to be used in a more complex architecture than in the work by Van Woudenberg et al., to allow for this behavior.

Summary of RNN

Data	Usage
Ultrasound cine	This is used as input for the convolutional network.
Quality score	This represents the quality of the image, this is used as the expected result during learning only, (for example for backpropagation)

Table E.6: Necessary components of the data

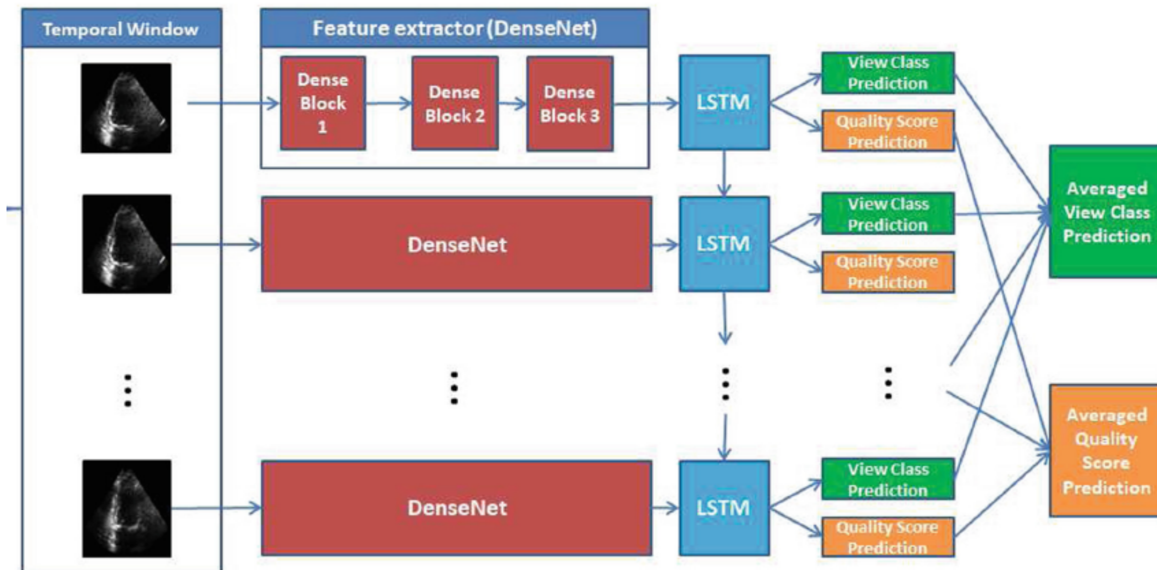


Figure E.2: Architecture described by Van Woudenberg et al. (2018)

Reinforcement Learning

The problem of giving instructions based on specific observations can be modeled using reinforcement learning, where the policy predicts the instructions needed. As traditional deep learning approaches would be hard to perform using image data, we instead examine deep reinforcement learning.

An important issue that needs to be covered, however, is the need for a learning environment. It is impossible to realize a learning environment using physical equipment, as training would be too slow, not to mention that it would require a human subject. (Milletari et al., 2019) suggest a simulated training environment as a replacement. They construct a simulated training environment by subdividing the chest area into several bins. Each bin is filled with at least 25 frames of data. All the bins that are within the wanted area are marked as correct. These parameters allow the policy to learn about how to position the probe. However, additional information is needed to find the correct rotation and tilt. To incorporate these instructions into the policy, it is required to fill "correct" bins with 5 additional clips. 50 'correct' frames, 50 frames for both excessive clockwise and counterclockwise rotation, and 50 frames for both excessive inferomedial and superolateral tilt should be captured.

Besides an environment providing the state of agent, the actions an agent has access to should be modelled as well. It was suggested to support the following set of actions:

Action	Effect
NOP	Stops the virtual probe. Should be issued at correct view
Move Lateral	Translates the probe towards the patients left
Move Medial	Translates the probe towards the patients right
Move Superior	Translates the probe towards the patients head
Move Inferior	Translates the probe towards the patients feet
Tilt Supero-laterally	Tilts the probe towards the head of the patient
Tilt Inferro-medially	Tilts the probe towards the feet of the patient
Rotate Clockwise	Rotates the probe clockwise
Rotate Counter-clockwise	Rotates the probe counter-clockwise

Table E.7: Set of actions that should be supported by the agent as suggested by Milletari et al. (2019)

Milletari et al. employ a deep Q-network to learn the policy. The Q-values are found using a convolutional neural network. The agent learns by simulating movement of the probe within the constructed virtual environment. The virtual probe is reset to a random location, and observes one of the images assigned to that location (i.e. bin). Based on the observation it will decide the best action, and is rewarded only if it moves to the "correct" bin. The exploration uses an ϵ-greedy strategy. As we'd like the network to learn based on the images and not the exact location of the bin, which might differ from person to person, the state is decided based solely on the image. A possible improvement would be taking into account relative position, which should increase accuracy with less danger of overfitting than using absolute position.

Summary of RL

Data	Usage
x,y-coordinate	To decide which images fall into which bin
Rotation	To decide which image belong to which rotation in a bin
Ultrasound image	The images should be taken in a grid, such that a virtual environment can be created
Labeling of images	To be able to label the correct bins

Table E.8: Necessary components of the input data

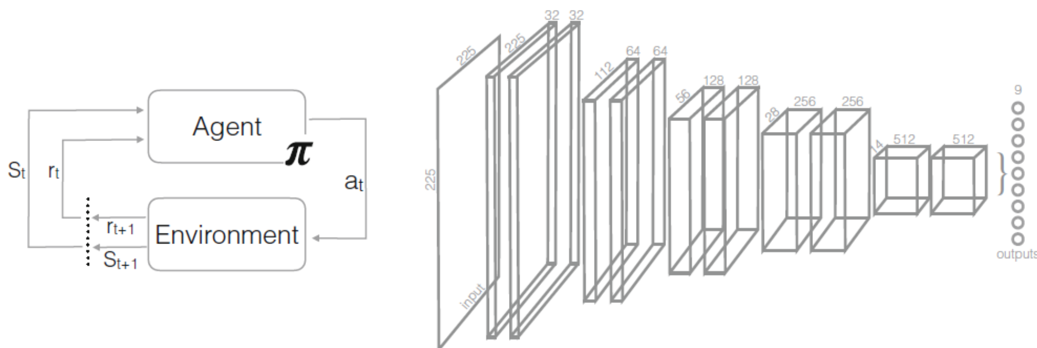


Figure E.3: Architecture described by Milletari et al. (2019)

Comparisons

	CNN	RNN	RL
Data collection	Ultrasound images, which have been labeled with position, rotation and quality	Ultrasound images, which have been labeled with position, rotation, quality and time, and have been grouped based on sessions	Ultrasound images of the chest area in a grid pattern, which have been labeled with position. Images within the right areas have been marked as correct. For correct areas, images with different rotation and tilt have been taken and labeled by quality.
Model adjustment	Data collection needs to be altered to use the gyroscope and accelerometer	Model should be adjusted to predict instructions, instead of only classifying views and assessing quality	Almost no adjustments needed
Data set size used	30 subjects, 611,000 frames	1,600,000 frames (assuming 100 frame cines)	27 subjects, 200,000 frames
Accuracy obtained	classification: 98% quality: 89% transl.: 2.0 ± 1.6mm rotation: $3.5 \pm 2.7°$	guidance: 86.1%	classification: 86.21%
Predicted performance	fastest (25 Hz on premium mobile device)	slowest (3 Hz on Samsung S8+)	slow

Table E.9: Comparisons between the methods proposed by Toporek et al. (2019), Van Woudenberg et al. (2018), Milletari et al. (2019)

	CNN	RNN	RL
Ease of data collection	+	-	–
Model adjustment	-	–	+
Accuracy:	+	++	+
Speed	+	–	-

Table E.10: Quick estimation of performance by Toporek et al. (2019), Van Woudenberg et al. (2018), Milletari et al. (2019)

Conclusion

Initially, we have thought about combining CNN, RNN, and RL to create an accurate model. However, this may lead to a complex model and require specific data to be gathered. Furthermore, training will be slow as multiple components need to be trained correctly. To reduce complexity and to keep training an accurate model feasible, we have chosen to use
This should result in accurate instruction tracking as well as should be an improvement over earlier approaches. To implement this model, we will use components suggested by both Toporek et al. (2019) and Van Woudenberg et al. (2018). However, there are some limitations due to the technicalities of this project. We have a limited amount of data sources available, as well as data being inherently noisy. As the only test subjects we currently have available consist of young males, the trained model might be lacking in identifying females and older subjects.

E.3.3. Metrics

To be able to judge the model's performance, some metric needs to be chosen. To find the most applicable metric, we explain several different metrics and compare them (Minaee, 2019).

Accuracy

$$\text{Accuracy} = \frac{\text{Number of correct predictions}}{\text{Total number of predictions}}$$

Accuracy is a good metric when using nominal classes and having equivalently distributed data across these classes. It fails to provide good performance, however, when the distribution is unbalanced, as it favors the classes with the most data the most. Another issue is that it does not consider relations between possible results, as there is no such concept between nominal data. However, when dealing with ordinal or interval data, this can be relevant. Despite this, accuracy often provides us with a simple way to quantify performance quickly.

Logarithmic Loss

$$\text{Logarithmic Loss} = \frac{-1}{N} \sum_{i=1}^{N} \sum_{j=1}^{N} y_{ij} * \log(p_{ij})$$

N : The number of samples
M : The number of classes
y_{ij} : Whether the sample i belongs to class j (1) or not (0)
p_{ij} : The probability of sample i belonging to class j

Logarithmic loss works when the classifier assigns some probability to each class for a specific sample. The loss function penalizes false classifications and for optimization purposes should be as

close to 0 as possible. A notable issue is that logarithmic loss goes up based on the number of samples as well as the number of classes. Therefore, it is difficult to use when comparing data sets.

F1 Score

$$F1 = \frac{2}{\frac{1}{\text{precision}} + \frac{1}{\text{recall}}}$$

F1 score tries to find the balance between precision and recall. This fixes one of the problems with accuracy, that specific cases are inaccurate but are not reflected in the accuracy score. This is due to precision measuring the instances that are classified as correct are correct and recall measuring the number of samples classified as correct. However, the issue with not taking into account interval data is still existent.

Mean Square Error

$$\text{MSE} = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y})^2$$

The mean square error is often used for regression problems. It calculates the average squared error between the predicted and the actual values. An issue still remains, as large errors are exaggerated due to the squaring behavior of the metric. This might result in images taken far away from the target position influences the algorithm too much, and thereby it may result in a decrease in accuracy.

Mean Absolute Error

$$\text{MSE} = \frac{1}{N}\sum_{i=1}^{N}|y_i - \hat{y}|$$

The mean absolute error is similar to the mean square error but suffers less from large errors impacting the overall prediction. There are still some issues with the mean absolute error, as errors far away from the target are expected to have a larger deviation due to either a decrease in data density or simply due to the larger distance. These values have the same impact on the metric as values closer to the target, which has a smaller expected deviation.

Conclusion

As the problem of finding the apical four-chamber view consists of both deciding the quality of the image, which can be seen as a binary classification problem and finding the instructions to improve the image, which is essentially a regression problem, we need to choose at least two different metrics. We decided that for regression MAE has the least problems and is the most applicable to the problem. We could also suggest altering MAE by using the square root of each term, as this would resolve the issue with differing deviations. For the binary classification problem, we will initially start with using accuracy. However, if it proves insufficient, we will consider F1 Score.

E.4. Frameworks

There are two things that we need to decide on in this section: programming language and framework. We argue that the chosen framework will be the main contributing factor in how the software will be built. Therefore, we will first decide which framework will be used, and only secondly decide the language.

E.4.1. Libraries

PyTorch and TensorFlow seem to be the most used libraries. TensorFlow is two years older than PyTorch. PyTorch is considerably more popular under researchers, while TensorFlow remains more popular in a corporate environment. Support for both applications should be a non-issue.

TensorFlow is a low level library. Keras and Sonnet are both higher-level libraries built on top of it. PyTorch is considered to be higher level than TensorFlow, but still low level compared to Keras.

The general consensus is to use a high level library, unless you need features that are not offered by it. Therefore, the ideal library would be a high level library that provides a fallback to a lower level one.

Some members in our team had good experiences with Keras, so we focus our attention there. Keras used to be a high level API that could run on several backends. TensorFlow was by far the most popular of the several choices available. Since the release of TensorFlow 2.0 in the end of 2019, the Keras api is fully integrated into TensorFlow and the support for other backends is discontinued.

Because of the tight integration between Keras and TensorFlow, we expect that starting with the Keras API in TensorFlow will be the best choice. We will be able to use the high level APIs for most of our features, while the tight integration with TensorFlow will allow us to fall back to the lower level APIs should pure Keras not suffice.

E.4.2. Language

Python is by a large margin the most popular tool for machine learning. While TensorFlow is implemented in C++, TensorFlow, too, is mostly used in a Python environment. Most of the available documentation is also about the Python interfaces, making it the language of choice for TensorFlow.

It is not the only choice, however. Several other programming languages are officially supported by the TensorFlow library. However, only the Python API provides the Keras API. Because we specifically want to use the Keras API, we have to use Python.

However, there is a second factor that determines our language choice: we need to use the ██████ SDK in order to interact with the probe. This SDK only supports C++/.Net, C# and native C++. We identified two possible solutions to solve this problem.

The first option is to split our software in two. The part of our software that has to train the model needs the Keras API, but doesn't need to interact with the probe. This part can be written in Python. The other part will be in charge of controlling the robot using the trained model. This part needs to make use of the ██████ SDK. Since the C++ API provided by TensorFlow does allow us to load a pre-trained Keras model, we can program this part in C++.

The second option is to write both parts in Python. This would require us to write Python bindings for the ██████ SDK. Considering the size of the SDK, doing this manually would require significant work, even if we only write bindings for the methods that we use. There are automated tools available, but we've not evaluated their results on large projects like the ██████ SDK.

In our team, none of us have experience with writing bindings, while there is experience with writing C++. Considering this, we choose to use the first option where we train the model in Python, and control the robot in C++. Since the SDK only supports Windows, there is no reason not to use the .Net libraries, so we'll use those as well.

E.4.3. Conclusion

We will write a Python program that trains a TensorFlow model. We will use the Keras API wherever possible, only falling back to raw TensorFlow when the Keras API doesn't fulfill our needs. The part of our software that interacts with the [redacted] SDK and is responsible for the real-time predictions will be written in C++ using the .Net libraries. The model trained using the Python program will be imported in the C++ version of TensorFlow.

Bibliography

Abdul, M. and Lateef, R. (2019). Expansion and implementation of a 3x3 sobel and prewitt edge detection filter to a 5x5 dimension filter. *Baghdad College of Economic sciences University*.

Ahmed, M. (2017 (accessed 23rd of June, 2020)). *Echocardiogram vs. ECG*. `https://myheart.net/articles/echocardiogram-vs-ekg-explained-by-a-cardiologist/`.

Atlassian (n.d.). Git Feature Branch Workflow.

Bouwers, E., Visser, J., and Van Deursen, A. (2012). Getting what you measure. *Queue*, 10(5):5056.

Bowman, A., Harreveld, R. B., and Lawson, C. (2019). A discussion paper on key issues impacting the sonographer workforce in australia. *Sonography*, 6(3):110–118.

Braunschweig, F., Cowie, M. R., and Auricchio, A. (2011). What are the costs of heart failure? *EP Europace*, 13(suppl_2):ii13–ii17.

Cleveland Clinic (2018 (accessed 23rd of June, 2020)). *4 Valves of the Heart: What Are They & How They Work*. `https://my.clevelandclinic.org/health/articles/17067-heart-valves`.

Craig, M. (2003). Sonographer shortages: A day late and a dollar short? *Journal of Diagnostic Medical Sonography*, 19(3):199–199.

Fowler, M. (2013). An appropriate use of metrics.

Iandola, F., Han, S., Moskewicz, M., Ashraf, K., Dally, W., and Keutzer, K. (2016). Squeezenet: Alexnet-level accuracy with 50x fewer parameters and <0.5mb model size. *arXiV*.

Ketout, H., Gu, J., and Horne, G. (2011). Mvn_cnn and fcnn for endocardial edge detection. In *2011 1st Middle East Conference on Biomedical Engineering*, pages 208–212. IEEE.

Krizhevsky, A., Sutskever, I., and Hinton, G. E. (2017). Imagenet classification with deep convolutional neural networks. *Commun. ACM*, 60(6):8490.

Landzaat, J. (2020 (accessed 23rd of June, 2020)). *Scan Views*. `http://echocardiografie.nl/scan-views/`.

Milletari, F., Birodkar, V., and Sofka, M. (2019). Straight to the point: Reinforcement learning for user guidance in ultrasound. In Wang, Q., Gomez, A., Hutter, J., McLeod, K., Zimmer, V., Zettinig, O., Licandro, R., Robinson, E., Christiaens, D., Turk, E. A., and Melbourne, A., editors, *Smart Ultrasound Imaging and Perinatal, Preterm and Paediatric Image Analysis*, pages 3–10, Cham. Springer International Publishing.

Minaee, S. (2019).

Mustafa, A. S. B., Ishii, T., Matsunaga, Y., Nakadate, R., Ishii, H., Ogawa, K., Saito, A., Sugawara, M., Niki, K., and Takanishi, A. (2013). Development of robotic system for autonomous liver screening using ultrasound scanning device. *2013 IEEE International Conference on Robotics and Biomimetics, ROBIO 2013*, pages 804–809.

Radon (2020). The cc command.

Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. (2014). Dropout: A simple way to prevent neural networks from overfitting. *J. Mach. Learn. Res.*, 15(1):19291958.

Tay, P. C., Garson, C. D., Acton, S. T., and Hossack, J. A. (2010). Ultrasound despeckling for contrast enhancement. *IEEE Transactions on Image Processing*, 19(7):1847–1860.

TensorFlow (2020). Tensorflow.

Toporek, G., Naidu, R. S., Xie, H., Simicich, A., Gades, T., and Raju, B. (2019). User guidance for point-of-care echocardiography using a multi-task deep neural network. In Shen, D., Liu, T., Peters, T. M., Staib, L. H., Essert, C., Zhou, S., Yap, P.-T., and Khan, A., editors, *Medical Image Computing and Computer Assisted Intervention – MICCAI 2019*, pages 309–317, Cham. Springer International Publishing.

Toporek, G., Wang, H., Balicki, M., and Xie, H. (EasyChair, 2018). Autonomous image-based ultrasound probe positioning via deep learning. EasyChair Preprint no. 119.

Uddin, M. S., Tahtali, M., Lambert, A. J., Pickering, M. R., Marchese, M., and Stuart, I. (2016). Speckle-reduction algorithm for ultrasound images in complex wavelet domain using genetic algorithm-based mixture model. *applied optics*, 55(15):4024–4035.

Van Woudenberg, N., Liao, Z., Abdi, A. H., Girgis, H., Luong, C., Vaseli, H., Behnami, D., Zhang, H., Gin, K., Rohling, R., Tsang, T., and Abolmaesumi, P. (2018). Quantitative echocardiography: Real-time quality estimation and view classification implemented on a mobile android device. In Stoyanov, D., Taylor, Z., Aylward, S., Tavares, J. M. R., Xiao, Y., Simpson, A., Martel, A., Maier-Hein, L., Li, S., Rivaz, H., Reinertsen, I., Chabanas, M., and Farahani, K., editors, *Simulation, Image Processing, and Ultrasound Systems for Assisted Diagnosis and Navigation*, pages 74–81, Cham. Springer International Publishing.

Vogel, H. (1979). A better way to construct the sunflower head. *Mathematical Biosciences*, 44(3):179 – 189.

Weinstein, M. C. and Stason, W. B. (1985). Cost-effectiveness of interventions to prevent or treat coronary heart disease. *Annual Review of Public Health*, 6(1):41–63. PMID: 2859868.

Yang, T. and Yang, L.-B. (1997). Fuzzy cellular neural network: A new paradigm for image processing. *International journal of circuit theory and applications*, 25(6):469–481.