# Reinforcement Learning for Mobile Missile Launching

**Bestärkendes Lernen für den mobilen Raketenabschuss**
Bachelor-Thesis von Daniel Palenicek aus Frankfurt am Main
Tag der Einreichung:

1. Gutachten: Prof. Dr. Jan Peters
2. Gutachten: -
3. Gutachten: -

TECHNISCHE
UNIVERSITÄT
DARMSTADT

Reinforcement Learning for Mobile Missile Launching
Bestärkendes Lernen für den mobilen Raketenabschuss

Vorgelegte Bachelor-Thesis von Daniel Palenicek aus Frankfurt am Main

1. Gutachten: Prof. Dr. Jan Peters
2. Gutachten: -
3. Gutachten: -

Tag der Einreichung:



For my parents, who always support me.

# Erklärung zur Bachelor-Thesis

Hiermit versichere ich, die vorliegende Bachelor-Thesis ohne Hilfe Dritter nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus Quellen entnommen wurden, sind als solche kenntlich gemacht. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Darmstadt, den 27. Oktober 2016

(Daniel Palenicek)

# Abstract

In the scope of this thesis a mobile robot was built. It's main components are a USB Missile Launcher, a 4WD Arduino Mobile Platform and a webcam.
The robot was taught to find and shoot a specified target, using image processing and reinforcement learning methods. For image processing the SURF and FLANN algorithms were used. For reinforcement learning the Q-Learning algorithm with an $\epsilon$-greedy policy was applied.
The effects of two different $\epsilon$-values on the robot's ability to learn the task were evaluated in a series of experiments. It turned out, that the smaller $\epsilon$-value performed better in regards to the robot's learning performance.

# Zusammenfassung

Im Rahmen dieser Thesis wurde ein mobiler Roboter gebaut. Seine Hautptkomponenten sind ein USB-Raketenwerfer, eine 4WD Arduino-Mobile-Plattform und eine Webkamera.
Dem Roboter wurde beigebracht, ein spezifiziertes Ziel zu finden und abzuschießen. Hierbei wurden Bildverarbeitungsmethoden und ein „Bestärkendes Lernen"-Algorithmus (reinforcement learning) verwendet. Für die Bildverarbeitung kommen der SURF- and FLANN-Algorithmus zum Einsatz. Das „Bestärkende Lernen" nutzt den Q-Learning-Algorithmus mit einer $\epsilon$-greedy-Strategie.
In einer Reihe von Experimenten wurde der Einfluss zweier unterschiedlicher $\epsilon$-Werte auf die Lernfähigkeit des Roboters untersucht. Es stellte sich heraus, dass der kleinere $\epsilon$-Wert zu einem besseren Lernerfolg führt.

# Contents

# Figures and Tables

## List of Figures



## List of Tables

# Abbreviations, Symbols and Operators

## List of Abbreviations

| Notation | Description |
|---|---|
| AI | Artificial Intelligence |
| FLANN | Fast Library for Approximate Nearest Neighbors |
| SURF | Speeded Up Robust Features |

## List of Symbols

| Notation | Description |
|---|---|
| $A_m$ | Set of actions for the state machine approach. |
| $A_{rl}$ | Set of actions the robot can perform in the reinforcement learning approach. |
| $A_{scene}$ | The area size of the scene. |
| $A_{target}$ | The area size of the target object within the scene. |
| $area_{rel}$ | The relation between the target's area and the scene's area. |
| $\epsilon$-greedy | Policy that determines which action to choose. |
| $p_{direction}$ | Distance between the target object and the camera's center, measured in pixels with $p_{direction} \in \mathbb{N}$ and $direction \in \{left, right\}$. |
| $Q(s,a)$ | Value of taking action $a \in A_{rl}$ in state $s \in S_{rl}$. |
| $r_{s,a}$ | Reward for action $a \in A_{rl}$ in state $s \in S_{rl}$. |
| $S_m$ | Set of states for the state machine approach. |
| $S_{rl}$ | States set for the reinforcement learning approach. |
| $v_{cp}$ | A vector of four corner points. |

# 1 Introduction

Artificial Intelligence (AI) is becoming a key feature in numerous products and is increasing in importance for diverse industries. One example are self driving cars, like Google[1] and Tesla[2] build. Another one are consumer drones that can sense obstacles, avoid crashes, and autonomously follow a target while filming, like the DJI Phantom 4[3]. Even Google Photos and the Google Search Prompt are now using Neural Networks and are therefore powered by AI.
At the end of April 2016, Google announced their believe that the world "will move from mobile first to an AI first world" [1]. That statement alone shows the importance and the potential of AI. A big field of research of AI is machine learning, which is a subfield of computer science, that "gives computers the ability to learn without being explicitly programmed" [2]. For some tasks this is a big advantage, because the programmer "only" needs to specify the problem to be solved and let the computer figure out how to. There are many challenges, where this approach is far more promising, than programming the problem solving algorithms by hand. For instance object recognition in photos or teaching robots to solve specified tasks. The latter is the main focus of this thesis.

## 1.1 Description of Task

The goal of this thesis is to teach a small, custom built robot to find and shoot a specified target using nerf gun like projectiles. To achieve this, it is necessary to familiarize with controlling USB devices programmatically, image processing (to identify a target in an image taken by the robot's on board camera), and to implement the robot's ability to fulfill the desired task.
There are two approaches that are both implemented to develop the robots AI. The first approach is to manually develop and hardcode all the rules for the behavior of the robot by hand. Doing this expands the knowledge about the robot and how to program it. With this experience gained, the next step is to implement the second approach. Namely to implement a reinforcement learning algorithm and to enable the robot to teach itself how to find and shoot the target.

## 1.2 Robot

The robot consists of multiple components. These different parts have individual strengths and weaknesses and therefore impose restrictions on the robot. These constraints have to be considered when defining the environment the robot operates in and when designing the software. The different hardware components are:

- **USB Missile Launcher**[4] by Dream Cheeky. It features three nerf gun like projectiles, can rotate about the horizontal axis and adjust the firing height on the vertical axis.
- **Logitech C160**[5] webcam for the robot to be able to *see* its environment. This webcam is a rather old and low performing webcam with a resolution of 640x480 pixels at a maximum frame rate of 15 fps. The webcam turned out to be a great limitation to the image processing which is explained later on.
- **4WD Arduino Mobile Platform**[6] with four wheels which are powered by four independent motors, hence it is capable of all wheel drive. The whole robot setup is mounted on top of it, which gives it the freedom of moving around.
- **Arduino Uno R3**[7] in combination with an **Arduino Motor Shield R3**[8] is used to control the mobile platform. Unfortunately, the motor shield at hand only features two channels for controlling motors. This fact poses a limitation to the mobile platform's all wheel drive capabilities. Furthermore, the two channels deliver less power than four channels would and, thus, the mobile platform is less powerful than it could be.

[1] https://www.google.com/selfdrivingcar/
[2] https://www.tesla.com/presskit/autopilot
[3] http://www.dji.com/phantom-4
[4] https://web.archive.org/web/20101119095325/http://www.dreamcheeky.com/usb-missile-launcher
[5] http://support.logitech.com/en_us/product/webcam-c160
[6] http://www.robotshop.com/en/dfrobot-4wd-arduino-mobile-platform.html
[7] http://www.arduino.org/products/boards/arduino-uno
[8] https://www.arduino.cc/en/Main/ArduinoMotorShieldR3

- A **Raspberry Pi Model B**[9] was considered to take the job of the main processing unit. As it turned out, the image processing is very demanding and the Raspberry Pi could not even partly deliver the processing power that is necessary to drive the mobile missile launcher at a reasonable speed. The Raspberry Pi is still mounted to the mobile missile launcher and it is possible to run the software on the Raspberry Pi, if desired, but during development and testing a more powerful laptop was used.

**Figure 1.1.:** Front and side view of the robot.

## 1.3 Related Work

This thesis has two predecessor bachelor theses. Both used the USB Missile Launcher in combination with different learning algorithms in order to teach it different abilities. They are explained in the following two sections 1.3.1 and 1.3.2.

### 1.3.1 Comparison of Different Learning Algorithms on a USB Missile Launcher

The first preceding thesis by A. Zimpfer is entitled "Comparison of Different Learning Algorithms on a USB Missile Launcher"(2012) [3]. Zimpfer's thesis focused on the USB Missile Launcher (section 1.2) and the webcam. He first built a control software, that allowed the user to control the robot using keyboard inputs.
The next task was to precisely control the launcher so that it could aim at a specified target autonomously. For that he let the launcher turn in a random direction for a random time and saved the time and the number of pixels turned in the form $\Delta time = \Delta pixel$, resulting in a set of numerous measurements. These measurements where then fead to six different supervised learning algorithms. After that, the outcomes of these learning algorithms were compared, and Nearest-Neighbor was identified as being able to best predict the needed turning times for a desired number of pixels.
Finally, he implemented a face recognition algorithm which, in combination with the precise control, allows the launcher to aim autonomously and shoot at faces that appeared in front of it.

### 1.3.2 Learning Mobile Missile Launching

The second preceding thesis is named "Learning Mobile Missile Launching" [4] by S. Alexeev and dates from 2015. He essentially had a very similar robot setup and a similar task to this thesis. He also attempted to teach the robot to find and shoot a specified target using reinforcement learning. However, the robot could only move forwards and backwards, while the launcher mounted on top of the Arduino Base could turn in all directions. He used a Temporal Different learning algorithm called "State-Action-Reward-State-Action" (SARSA) with a softmax policy.

[9] https://www.raspberrypi.org/products/model-b/

# 2 Reinforcement Learning for Mobile Missile Launching

The basic idea was to implement a software solution that enables the robot to operate autonomously. There are two different approaches that were implemented. The first is a manually developed and the second one is a reinforcement learning approach. The software architecture was designed in a very modular way, and therefore, the general structure (figure 2.1) of the two approaches is identical.
Within the program's lifecycle the software repeatedly passes through different stages which implement different functionality that is explained in further detail in the next sections. The general structure of these stages is as follows:
The first step is to grab an image from the camera and process it (section 2.1). This stage uses state of the art image processing methods.
After processing the image the obtained results are interpreted. This enables the robot to understand its environment in such a way that it can determine its *relative position* towards the target (section 2.2). The relative position correlates to a state $s$, with $s \in S_m$ in case of the state machine approach and $s \in S_{rl}$ in case of the reinforcement learning approach. Therefore, being aware of the relative position is crucial for determining how to continue.
Proceeding, a decision has to be made on how to react to the relative position and the state $s$ the robot has recognized to be in (section 2.3). The robot decides which action to perform next. The decision making step is the one that is different for the state machine and the reinforcement learning approach. In case of the state machine approach, the decision is based on predefined rules. In contrast to that, when reinforcement learning is used, the robot learns said rules on its own. In order to get better at solving the task, it has to be allowed to train. It continuously learns from its mistakes and always tries to improve its *policy*.
Lastly, the chosen action has to be executed. Additionally, in the case of the reinforcement learning approach the result of that action also has to be evaluated and remembered for later, in order to learn from it. This is important, so that decisions that turned out to be good can be repeated and others can be avoided in the further.

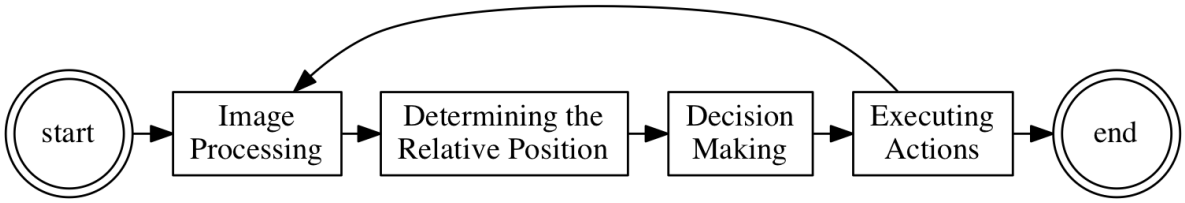


**Figure 2.1.:** The general structure of the task solving process, showing different stages the robot passes through.

As a result of the general structure being the same for the two approaches, they can share a lot of the same logic and programming code. This means, that in further development the software can be extended more easily. Furthermore, it is easy to replace different algorithms used within the software. The different stages of figure 2.1 are explained in detail in the following sections.

## 2.1 Image Processing

We as humans make sense of our environment by using our senses and interpreting the sensations that those deliver to our brain. For us this is a natural and therefore easy task. When it comes to computers this task quickly becomes non trivial. This thesis only focusses on *vision*, which is one of our senses. In order to gain an understanding of its environment, the robot was given the ability to have a basic vision, which was realized by mounting a camera to the robot. The basic idea is to capture a frame of the environment right in front of it. The captured frame is then processed using an open source image processing library called OpenCV. The goal of the image processing is to detect a specified target (figure 2.2) in the scene. When doing image processing there are a lot of different factors and limitations that have to be considered and dealt with, e.g. the lighting in the scene might change between frames or might be different in the scene and in the reference picture of the target object. Other problems are, especially when using a low budget camera, that the picture might be too dark, blurred, distorted or not have enough resolution and detail to recognize the object from a certain distance or angle. Despite these circumstances the robot has to have a sense of its environment to be able to fulfill its task. Otherwise, it could not make any *intelligent* decisions and would just be able to blindly guess what to do next. Therefore, aforementioned obstacles have to be overcome, by for example performing validation (section 2.1.2) and sampling (section 2.2.1).


**Figure 2.2.:** Reference picture of the robot's target object.

To implement the image processing stage the OpenCV[1] image processing library is used. It is state of the art and the de facto standard for image processing [16]. It provides a variety of different algorithms. For finding the target object in the scene and determining its position within the frame, its implementations of the SURF- and FLANN-Algorithms (section 2.1.1) are used.
Beyond that other techniques to enhance the information gotten from the processing algorithms were developed. These are explained in sections 2.1.2 and 2.2.1. The combination of these approaches and techniques enables the robot to *see* and understand important parts of its environment.

### 2.1.1 SURF and FLANN

The robot's task of finding the target object can be reduced to the task of recognizing the target object (figure 2.2) within the camera frame. This is essentially what the robot needs to be able to do to detect where the target object is located. There are a lot of fundamentally different approaches like color detection, edge detection, feature matching and more to achieve this. Feature matching is used due to its suitability for this task.
It works by finding and matching key points between two images. This process can be divided into three main steps:

1. detect interest points
2. describe interest points
3. match the interest points between two pictures

The following explains how "Speeded Up Robust Features (SURF)" is used to detect and describe the key points in both the camera scene and the target object and how "Fast Library for Approximate Nearest Neighbors (FLANN)" is utilized to match the key points between the two pictures.
The SURF detector accomplishes the tasks of finding interest points within a picture. It is based on the Hessian matrix due to of its good performance and accuracy [5]. This makes it much faster than other comparable feature detectors, for instance "Scale-invariant feature transform (SIFT)" or PCA-SIFT [6].
For each interest point found by the detector the SURF descriptor describes the distribution of the intensity content within the interest point's neighborhood [5]. The combination of the detector and the descriptor makes it possible to reduce a picture to its key features. This process is still repeatable even if the picture is rotated, zoomed, or transformed up to a certain degree. Repeatable in this context means that the algorithm will detect and describe the same interest points in the same way so that they can be clearly identified.
The next step is to match the corresponding interest points from both pictures to each other. "For many computer vision problems, the most time consuming component consists of nearest neighbor matching in high-dimensional spaces. There are no known exact algorithms for solving these high-dimensional problems that are faster than linear search. Approximate algorithms are known to provide large speedups with only minor loss in accuracy" [7]. FLANN chooses and runs the fastest algorithm that is best suited for the provided data set [7].
The end result of these three steps is a vector $v_{cp} = (\begin{smallmatrix} a & b & c & d \end{smallmatrix})$ with $a, b, c, d \in \mathbb{R}_0^2$ which are essentially coordinates within the scene of the form $\binom{x}{y}$. $v_{cp}$ represents the target's four corner points found in the scene. This means they represent the location at which the target object was found.

[1] `http://opencv.org`

### 2.1.2 Validation

By definition the image processing step always returns a vector $v_{cp} = (\,a\ \ b\ \ c\ \ d\,)$ containing four points. This is the case because FLANN always matches the interest points with the smallest distance. If the target is not in the scene or if the interest points were chosen badly, then the points that will be matched do not represent the object. Since the algorithms of the following decision making step (section 2.3) rely on $v_{cp}$ representing the target object's corners, $v_{cp}$ has to pass a validation step. The decision made in the validation step determines whether the proposed vector $v_{cp}$ will be used in further calculations or will be rejected.
The validation step uses geometric analysis of the shape formed by the four coordinates in $v_{cp}$. Only if every single filtering step is passed, the proposed target object's corners are assumed to represent the real target object's corners and $v_{cp}$ will be used in later calculations. Otherwise, it will be rejected and it is assumed that the target object is in fact not present in the scene.
The validation consists of the following filtering steps:

1. **Object area filter:** This filter calculates the object's area in relation to the scene's area. Only if the area of the target object is larger than 1% of the camera frame's area, $v_{cp}$ passes this filter (equation 2.1). This approach filters incoming false positives in regard to the object's area reliably, since SURF in combination with the given camera would only detect objects considerably lager than 1%. The area filter is hence the simplest and probably one of the most effective.

$$area_{rel} = \frac{A_{target}}{A_{scene}} > 0.01 \tag{2.1}$$

2. **Corner position filter:** It tests the position of the different points in $v_{cp}$ in relation to each other. Only if $a$ is the top left, $b$ is the top right, $c$ is the bottom right and $d$ is the bottom left point $v_{cp}$ will pass this filter (equation 2.2). This indirectly allows the target to be rotated 90° at most in either direction.

$$a_x < b_x \ \wedge \ d_x < c_x \ \wedge \ a_y < d_y \ \wedge \ b_y < c_y \tag{2.2}$$

3. **Edge relation filter:** It checks the ratio between the length of the opposing edges (equation 2.3). Objects that are extremely distorted, caused by errors of the SURF algorithm, will be filtered out. Other trapezoid-like distorted objects will pass this filter, as these distortions are likely to be caused by placing the robot not right in front of the target object but off to the side which is perfectly valid.

$$\frac{1}{2}(b_x - a_x) < (c_x - d_x) < 2(b_x - a_x) \ \wedge \ \frac{1}{2}(d_y - a_y) < (c_y - b_y) < 2(d_y - a_y) \tag{2.3}$$

## 2.2 Determining the Relative Position

After the scene has been processed and the corner points have been found valid, the target is considered to be recognized in the scene. Knowing, that the target object is present, the robot's relative position towards the target needs to be calculated. This is necessary so the robot can then analyze its relative position and decide how to react in consideration of the state it is in. Determining the relative position is again done by interpreting the geometric shape that is formed by the four corner points found in $v_{cp}$.

### 2.2.1 Sampling

The camera's suboptimal picture quality often will result in the object not being detected although it is present in the scene, if the distance towards the target is to long. Often the results of the image processing step will differ for subsequent pictures taken from the exact same position and without any changes to the environment. The more extreme the scene is with regards to the lighting, distance or angle towards the object the more often this effect will occur. This can result in the object being detected in one picture and then "disappear" in the next picture that was taken from the exact same spot.
It might also occur that the detected object's corners will be shifted by a number of pixels from their actual location. If the offset is too high, the corner points are not usable either, because they are not precise enough to rely on when controlling the robot. This is a fundamental problem, that has to be handled.

An approach, that is proposed in this thesis, to solve this problem is called *sampling*. The basic idea is to take multiple pictures from one position without moving in between, analyzing them independently and then combining the result into a single result (algorithm 1). Whereby all samples without a target detected, are rejected. This means that all of the samples have to be empty in order for the robot not to detect the target in the scene. If all the samples are empty we can be rather sure that this was not a measurement error and conclude, that the target really is not in the scene. The bigger the number of samples used the more accurate the detection will be. We reasonably assume, that the averaged result will be more accurate than most of the single results on their own, because extremes will cancel each other out. So the end result is good enough to rely on.
By default the robot uses a number of three samples which can be configured in the properties file (appendix A).
A single sampling result is represented as a vector $v_{cp}$ as explained in section 2.1.2.

$i \leftarrow 0$
$a, b, c, d \leftarrow \binom{0}{0}$
**for** *each sample* **do**
  **if** *object detected in sample* **then**
    $a \leftarrow a + sample_a$
    $b \leftarrow b + sample_b$
    $c \leftarrow c + sample_c$
    $d \leftarrow d + sample_d$
    $i \leftarrow i + 1$
  **end**
**end**
**return** $\left\{\frac{a}{i}, \frac{b}{i}, \frac{c}{i}, \frac{d}{i}\right\}$

**Algorithm 1:** The pseudocode for the sampling algorithm.

The sampling process turned out to be very effective. Figure 2.3 depicts how multiple samples are combined into one solution. While some of the samples are good (blue) others are rather bad (red). If the launcher had relied on them it would have been sub optimal. The end result (green) is a good approximation of the target.

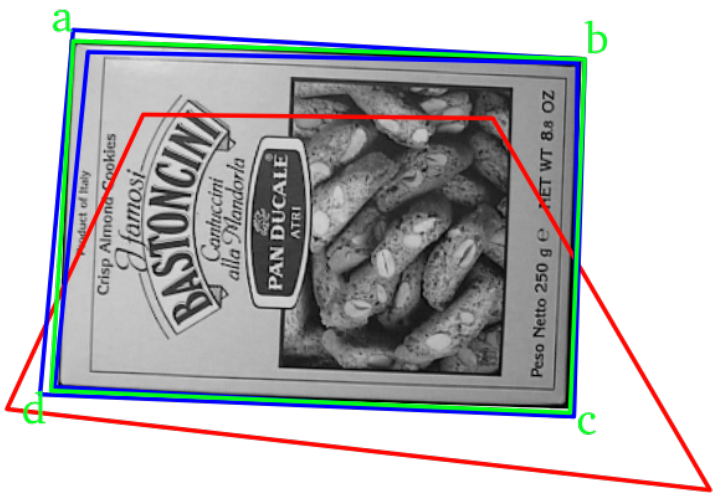


**Figure 2.3.:** Example of the sampling process. red=rejected, blue=sample, green=end result. This picture was redrawn by hand because the image quality of the camera is insufficient for printing.

## 2.3 Decision Making

As mentioned in the introduction of this chapter, there are two different approaches. The first one is to manually develop the robot's logic and refine all the parameters by hand. The second one is to implement reinforcement learning to accomplish the same task. Both approaches were implemented and successfully fulfill the task of finding and shooting the target. In the following, both are explained in separate sections.

### 2.3.1 State Machine Approach

The core idea when implementing the logic for the robot manually was to develop a state machine. A state machine consists of states and transitions. The machine can only be in one state at a time and can only change states by transitioning into another. This is an approach to greatly simplify complex tasks by dividing them into smaller pieces, since these can be controlled and debugged easier.
The state machine defined for the robot is shown in figure 2.4. The different states represent the different situations the robot can be in. In each state either a decision has to be made or an action has to be executed.
The state machine was also the technical basis for the later reinforcement learning approach. By implementing the state machine a flexible framework for controlling the USB Missile Launcher, the mobile platform, and performing image processing was developed. Working on the first approach, a lot about the hardware and all the necessary libraries has been learned. So when it came to implementing the reinforcement learning approach one could focus on that task, since all the groundwork was already done.

#### States

The state set $S_m$ for the state machine approach was chosen carefully. Each state represents a situation the robot is faced with while executing its task.

$$S_m = \{start,\ frameProcessed,\ noTargetDetected,\ targetDetected,\ badPosition,\ goodPosition,\ movedToNewPosition,\ end\} \tag{2.4}$$

#### Actions

In addition to the states, transitions between them are needed. This is done by making the robot execute actions. All actions the state machine uses are listed in the following equation.

$$A_m = \{turnLeft,\ turnRight,\ turnTowardTarget,\ driveForward,\ fire\} \tag{2.5}$$

The different actions in $A_m$ are explained in detail in section 2.4. Notice, that the state machine does not make use of all the possible actions the hardware could execute.

#### State Machine

The state machine as it was implemented is depicted below (figure 2.4). It is a combination of states $S_m$ and actions $A_m$. To transition into a new state, the robot either performs an action or makes a decision, depending on the state.

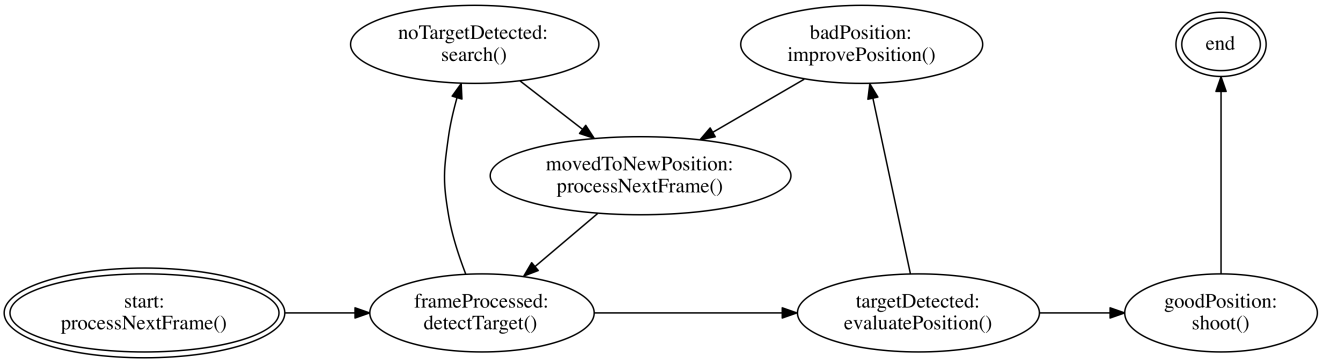


**Figure 2.4.:** State machine the manual approach is based on.

The state machine starts by processing a frame. If no target is detected in the scene, the robot will *move systematically*, following a predefined route to get to a new position, thereby a *zick-zack*-route was used as default (figure 2.7). After the

new position is reached, it processes the next frame. This procedure is repeated until the target is detected. When the target is detected, the state machine switches into the *targetDetected* state. Here it has to rate the quality of the robot's relative position towards the target, which can be either *good* or *bad*. A good position is characterized by the target being close enough and the robot being turned towards the target, so that its projectiles would hit the target. If the position is good, the state machine will transition into the *goodPosition* state and then shoot the target. On the other hand a bad relative position is characterized by being too far away or not being turned right towards the target. So before the target can be shot the relative position has to be improved. The state machine will therefore transition into the *badPosition* state which will lead to the position being improved by the following action.
Note that in the actual implementation of the state machine some of the states and actions have been merged to optimize the code. However, the general logic remains the same and it is easier to understand, as shown in figure 2.4.

### 2.3.2 Reinforcement Learning Approach

Machine learning is a subfield of computer science. Which can be divided into different subtypes: *supervised learning*, *unsupervised learning* and *reinforcement learning*. The idea behind reinforcement learning is to learn by interacting with the environment [8] (figure 2.5). It is therefore closely related to how humans learn. Especially young children explore the world around them and learn for the future by receiving feedback (*rewards*). For them the feedback they receive can come in different forms, e.g. pleasure or pain. Depending on the feedback they received, they will change their future behavior since they can expect the same or a similar feedback for repeating the same actions. An example is that the pain they feel when touching a hot surface will make them remember not to touch hot surfaces in the future. Another example is that when a child is learning to walk every time it falls to the ground it will learn from its mistakes until it manages to properly walk. Nobody tells the child how to walk. It is faced with the problem set and then tries to solve the problem by trial and error.
The same is true for reinforcement learning where the learner is not explicitly told what to do or how to do it. As Sutton puts it: "Reinforcement learning is defined not by characterizing learning methods, but by characterizing a learning *problem*. Any method that is well suited to solving that problem, we consider a reinforcement learning method" [8, p.4]. The learner learns by interacting with and exploring its environment. For every action he takes he receives a reward in the form of a "numerical reward value" [8, p.3]. The learner's long term goal is to maximize the cumulative reward he earns over time. By continuously repeating actions with a high reward and from time to time taking random actions to explore the environment, the learner tries to find the optimal solution over time.

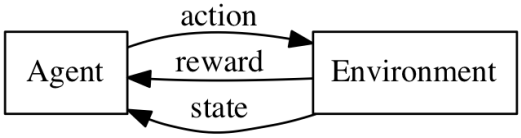


**Figure 2.5.:** The agent-environment interaction in reinforcement learning. Based on [8, p.52]

The main elements that a reinforcement learning system consists of are the agent, the environment, a *policy*, a *reward function*, a *value function*, and, optionally, a *model* of the environment [8]. Figure 2.5 shows the agent-environment interaction.
The *policy* is the only thing that ultimately determines which action to choose in which state. Therefore, one could say that it determines the robots *behavior*.
In the *reward function* the goal the learner is chasing is encoded. This means that it is a way to control the learner into choosing some actions over others by setting higher rewards for them. It determines what is good in the sense of which action will bring the highest immediate reward.
The *value function* determines which actions will generate the highest long term reward. Although only returning a small immediate reward, a specific action can result in a specific state, from which higher rewards can be collected in the future. Therefore, the learner will orientate on the value function when learning to maximize its cumulative rewards. The function $Q(s, a)$, which will be used later, represents the value of taking action $a \in A_{rl}$ in state $s \in S_{rl}$.
The *model* is optional as already mentioned and is not used in Q-Learning. Therefore, it will not be discussed.
Reinforcement learning faces a challenge that is unique and not present in other machine learning disciplines: finding the balance between exploration and exploitation. To maximize the cumulative reward the agent executes those actions he knows to have the highest value. But how can the agent be sure that another action, that he has not tried yet, is not going to turn out to have a higher value than expected? This is where exploration comes into play. Exploration simply means that the agent instead of taking the best action will take a *random* action from time to time. Doing this he gives said action a chance to turn having a higher value than assumed. There are many different strategies on how to handle

the balance between exploration and exploitation. In this thesis a $\epsilon$-greedy approach was used, which is explained in the next section. As in the case of exploration, there are a lot of different strategies for the exploitation as well. Here a Q-Learning algorithm was chosen.

Reinforcement learning algorithms can be divided into three groups:

- "**Dynamic programming** (DP) refers to a collection of algorithms that can be used to compute optimal policies given a perfect model of the environment as a Markov decision process (MDP)" [8, p.89].
- "**Monte Carlo methods** are ways of solving the reinforcement learning problem based on averaging sample returns" [8, p.111]
- **Temporal-Difference Learning** (TD) "is a combination of Monte Carlo ideas and dynamic programming (DP) ideas. Like Monte Carlo methods, TD methods can learn directly from raw experience without a model of the environment's dynamics. Like DP, TD methods update estimates base in part on other learned estimates, without waiting for a final outcome (they bootstrap)" [8, p.133]

### Q-Learning Algorithm

According to Watkins the development of *Q-Learning* was one of the most important breakthroughs in reinforcement learning [9]. Q-Learning is a *model-free off-policy temporal-difference* reinforcement learning algorithm. The simplest form, one step Q-Learning, is shown in equation 2.6.

$$Q(s,a) \leftarrow Q(s,a) + \alpha\Big[r + \gamma \max_{a'} Q(s',a') - Q(s,a)\Big] \tag{2.6}$$

Q-Learning updates its values $Q(s,a)$ of the previous state directly, using the best $Q(s,a)$ of the new state, as equation 2.6 shows. So "the learned action-value function, $Q$, directly approximates $Q^*$, the optimal action-value function, independent of the policy being followed" [8, p.148], therefore, it is *off-policy*. The policy used still has the usual functionality of determining which states are being visited. As long as all the states are regularly updated $Q(s,a)$ will converge [8, p.148].
Algorithm 2 shows the Q-Learning algorithm as it was implemented in this thesis. It loops as long as the last action taken was not the firing action and the total reward is bigger than -30. In each iteration it will choose the next action $a \in A_{rl}$ using an $\epsilon$-greedy policy. The chosen action will then be executed. The outcome of executing that action are a new state $s \in S_{rl}$ and a received reward $r$ calculated using the reward function. The next step is to update $Q(s,a)$ by using equation 2.6.

**for** *each episode* **do**
  $totalReward \leftarrow 0$
  **do**
    Choose $a$ from $s$ using $\epsilon$-greedy
    Take action $a$
    Observe $r$, and $s'$
    $Q(s,a) \leftarrow Q(s,a) + \alpha\Big[r + \gamma \max_{a'} Q(s',a') - Q(s,a)\Big]$
    $s \leftarrow s'$
    $totalReward \leftarrow totalReward + r$
  **while** $a \neq fire$ **and** $totalReward > -30$;
**end**

**Algorithm 2:** The pseudocode of how Q-Learning was implemented. Based on [8, p.149].

### $\epsilon$-greedy Policy

As mentioned earlier one of the most important tasks in reinforcement learning is finding the right balance between exploration and exploitation. There are different ways on how to approach this task from very simple to extremely complicated. $\epsilon$-greedy is one of the simpler approaches. The idea is to choose greedily most of the time, i.e. performing exploitation. Sometimes, however, a random action will be chosen in order to perform exploration. How often a random action will be chosen is determined by a threshold $\epsilon$. This essentially means that $\epsilon * 100\%$ of the time the policy will choose a random action and $(1-\epsilon) * 100\%$ of the time the policy will choose the greedy option.

How the $\epsilon$-greedy algorithm works can be seen in the following algorithm 3:

$r \leftarrow$ random number $\in (0, 1)$
**if** $r < \epsilon$ **then**
  $a \leftarrow$ random action $\in A_{rl}$
**else**
  $a \leftarrow a_{greedy} \in A_{rl}$ with $\max_{a_{greedy}} Q(s, a_{greedy})$
**end**
**return** $a$

**Algorithm 3:** Pseudocode of the $\epsilon$-greedy implementation.

### States

One big part of the reinforcement approach learning is the selection of states $s \in S_{rl}$ the robot can be in. The states of the reinforcement learning approach are defined in the following equation. Important to note is that $S_{rl} \neq S_m$.

$$S_{rl} = \{noTargetDetected,\ badPosition,\ tooFarAway,\ goodPosition\} \tag{2.7}$$

Which state the robot is in at a certain point in time depends on the environment. If the target was not detected in the scene, the robot would be in the state *noTargetDetected*. The state *tooFarAway* is used when the robot is too far away from the target. When the robot is close enough but the center point of the scene does not intersect the target, the robot is in state *badPosition*, because it would miss the target if it tried to shoot. If non of the other states are true then the robot is in state *goodPosition*, which means that he is close enough to the target and positioned in the right way to shoot it. The criteria for the different states are listed in the equation 2.8.

$$s = \begin{cases} noTargetDetected & if\ \neg target\ detected \\ tooFarAway & if\ target\ detected \wedge area_{rel} < 0.15 \\ badPosition & if\ target\ detected \wedge \neg(s = tooFarAway) \wedge \neg intersects \\ goodPosition & otherwise \end{cases} \tag{2.8}$$

### Actions

To transition between different states, the robot has to execute actions. It does not know which state the execution of an action will result in. Technically each action can be performed in any state. Through reinforcement learning the robot will learn which one to execute at a specific time. The actions the robot can perform are listed in $A_{rl}$. Again important to note that $A_{rl} \neq A_m$.

$$\begin{aligned} A_{rl} = \{&turnLeft,\ turnRight,\ turnTowardTarget,\ driveForward,\\ &driveBackward,\ searchSystematically,\ fire\} \end{aligned} \tag{2.9}$$

The different actions in $A_{rl}$ are explained in detail in section 2.4.

### Reward Function

The reward function's purpose is to determine the reward the robot receives for executing an action in a specific state. It is implemented as a static look up table. In the following, it is split up into different functions to make it more readable. A reward value is determined depending on a state and an action.

$$r_{noTargetDetected,a} = \begin{cases} -1 & if\ a = searchSystematically \\ -2 & if\ a \in \{driveForward,\ driveBackward\} \\ -3 & if\ a \in \{turnLeft,\ turnRight\} \\ -5 & if\ a = turnTowardTarget \\ -20 & if\ a = fire \end{cases} \tag{2.10}$$

$$r_{badPosition,a} = \begin{cases} 3 & if\ a = turnTowardTarget \\ 0 & if\ a \in \{driveForward,\ driveBackward\} \\ -1 & if\ a \in \{turnLeft,\ turnRight,\ searchSystematically\} \\ -20 & if\ a = fire \end{cases} \tag{2.11}$$

$$r_{tooFarAway,a} = \begin{cases} 3 & if\ a = driveForward \\ -1 & if\ a = turnTowardTarget \\ -2 & if\ a = driveBackward \\ -3 & if\ a \in \{turnLeft,\ turnRight,\ searchSystematically\} \\ -1 & if\ a = fire \end{cases} \tag{2.12}$$

$$r_{goodPosition,a} = \begin{cases} 10 & if\ a = fire \\ -1 & if\ a \in \{turnTowardTarget,\ driveForward\} \\ -2 & if\ a = driveBackward \\ -3 & if\ a \in \{turnLeft,\ turnRight,\ searchSystematically\} \end{cases} \tag{2.13}$$

## 2.4 Executing Actions

In order to interact with its environment the robot has to be able to perform different maneuvers. So the fourth step for the robot is to execute the action $a \in A$ that was chosen during the first three steps and which represents a maneuver that the robot con execute. Some of the actions are performed by the missile launcher while others are performed by the Arduino Base. In total, the robot does support nine actions of which five were used. These actions were used to enable the robot to perform a number of maneuvers.

### 2.4.1 Missile Launcher

The missile launcher can perform five actions in total. These are:

- **Rotating around the horizontal axis** in either left or right direction
- **Rotating around the vertical axis** either up or down
- **Firing** a projectile

Technically there is also a "stop" action which stops the command that is currently being executed but here a more abstract level is considered, where this action can be ignored. In this thesis only the missile launcher's firing action was used and the first four actions were left out. The reason for this is that the rotation actions around the horizontal axis are redundant to the Arduino Base's ability to rotate around the horizontal axis. This would just have added unnecessary complexity. The rotation around the vertical axis was left out in the beginning for sake of simplicity. This way the robot only has to look for the target at floor level instead of also looking up and down. They were not implemented later either. The knowledge for the technical implementation is based on Zimpfert's [3] implementation. It was rewritten in C++ and ported from libusb-0.1 to libusb-1.0 which is a rewrite of the library.

### 2.4.2 4WD Arduino Base

The Arduino 4WD Base houses four electric motors which can be controlled independently. This makes the base capable of four wheel drive. However the Arduino Motorshield, which drives the base, has only two channels to control electric motors. This makes it impossible to control each wheel individually. Therefore, the motors were wired up in such a way,

that the left (front and back) motors are controlled via the first channel, and both right motors are controlled via the second channel.
When the Arduino Motorshield applies the same voltage to both channels, the base will move either forwards or backwards. When, however, the Motorshield applies inverse voltage to the different channels, the base will execute a turn maneuver. The angle of rotation depends on the time $t$ the voltage is applied. The *angle of rotation* in the context of this thesis correlates with a number of pixels. This is convenient, because on the camera image the target's vertical distance towards the camera center can be easily measured in pixels $p_{direction} \in \mathbb{N}$ with $direction \in \{left, right\}$.
In order to be able to control the robot's movement precisely and to predict how many pixels it will turn, when the motors spin for $t$ milliseconds, a series of measurements was performed. Approximately 600 measurements of the form $\Delta t = \Delta p_{direction}$ were collected. These are used to approximate the factors $a$ and $b$ for two functions of the form $t(p_{direction}) = a + b * p_{direction}$. The approximation is based on a linear regression (figure 2.6). Looking at the measurements, it is obvious, that the rotation angles not only depend on the given time $t$, but also on the turning *direction*.

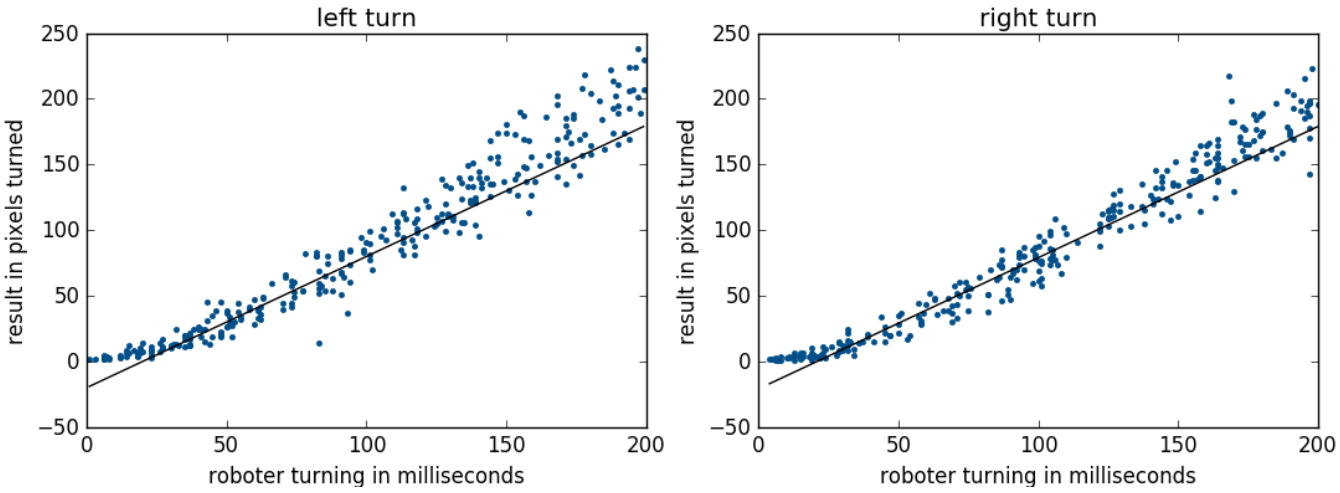


**Figure 2.6.:** Linear regressions on $\Delta t = \Delta p_{direction}$ measurements, resulting in equations 2.14 and 2.15.

$$t(p_{left}) = -20.0850806037 + 1.09547933756 * p_{left} \tag{2.14}$$

$$t(p_{right}) = -21.0363099537 + 1.05450212881 * p_{right} \tag{2.15}$$

So by applying either the same or inverse voltage to the two control channels and choosing the time according to the determined equations (2.14, 2.15), the base can perform the following maneuvers:

- **Drive** forwards and backwards
- **Rotate** around the horizontal axis in either left or right direction
- **Turn towards** the target on the horizontal axis. This maneuver turns the robot towards the target, namely exactly so far, that the target will be perfectly centered after this maneuver.
- **Search systematically**: This maneuver is the most flexible, because it allows to perform different actions at different points in time. For that purpose, the robot is provided a predefined search pattern (sequence of actions). The idea is to let the robot perform one action of the predefined sequence at a time. Each time *search systematically* is invoked the next action from the sequence is performed. The default search pattern (configured via robot_search_strategy property. See appendix A) is a *zick-zack* search patter shown in figure 2.7. The robot will drive forward (1), then turn left twice (2 and 3), drive forward (4) again and then turn right twice (5 and 6). This pattern can be repeated infinitely.

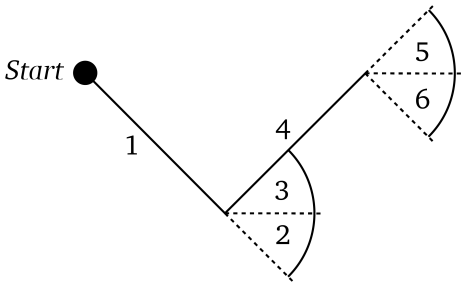


**Figure 2.7.:** Predefined *zick-zack* search pattern.

## 2.5 Technical Views

Technical views visualize the technical realization of the software on a very abstract level [10]. In the following a building block view (section 2.5.1) and a deployment view (section 2.5.2) of the mobile missile launcher application are presented.

### 2.5.1 Building Block View

The building block view (figure 2.8) visualizes all the software- and implementation-components. Different components are shown at different scales of detail [10]. The components are explained in table 2.1.

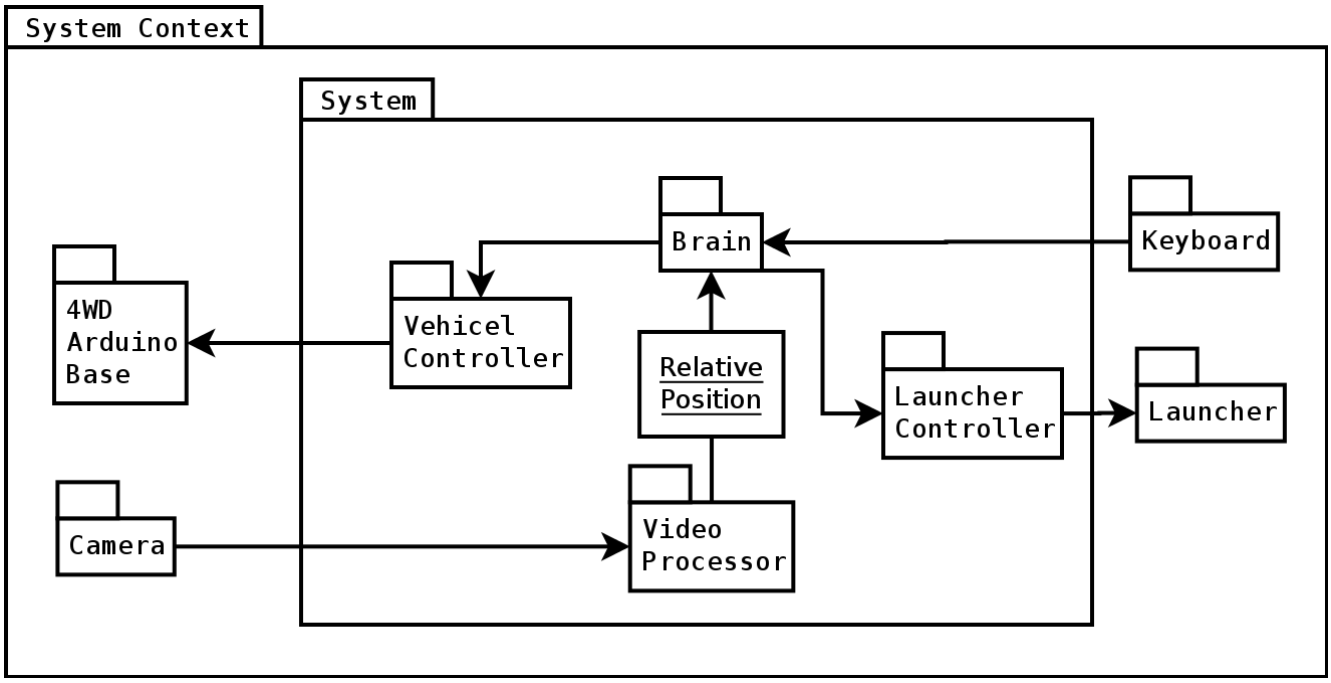


**Figure 2.8.:** Building block view of the mobile missile launcher application.

**Table 2.1.:** Description of the different components in the building block view.

| Component | Description |
|---|---|
| Brain | Performs the main processing for both the state machine and the reinforcement learning approach, e.g. it calculates the states and rewards, initiates actions, and manages the learning cycle. It connects the different components and handles the communication between them. |
| Launcher Controller | Controls the USB Missile Launcher. It translates the program's actions and writes them onto the serial port. |
| Relative Position | The result of the *Video Processor's* analyzation process is sent to the *Brain* via the *Relative Position* object. It is the central data object which also performs the validation and the sampling process (section 2.1.2) |
| Vehicle Controller | Controls the 4WD Arduino Base. It translates the programs actions and writes them onto the serial port. |
| Video Processor | Reads video input from the camera and analyzes it, looking for the target in every video frame. |

### 2.5.2 Deployment View

The deployment view (figure 2.9) visualizes two tings: First it shows the technical environment of the software in the form of the different hardware components. Second it shows the shows the deployment of the software across the hardware components.

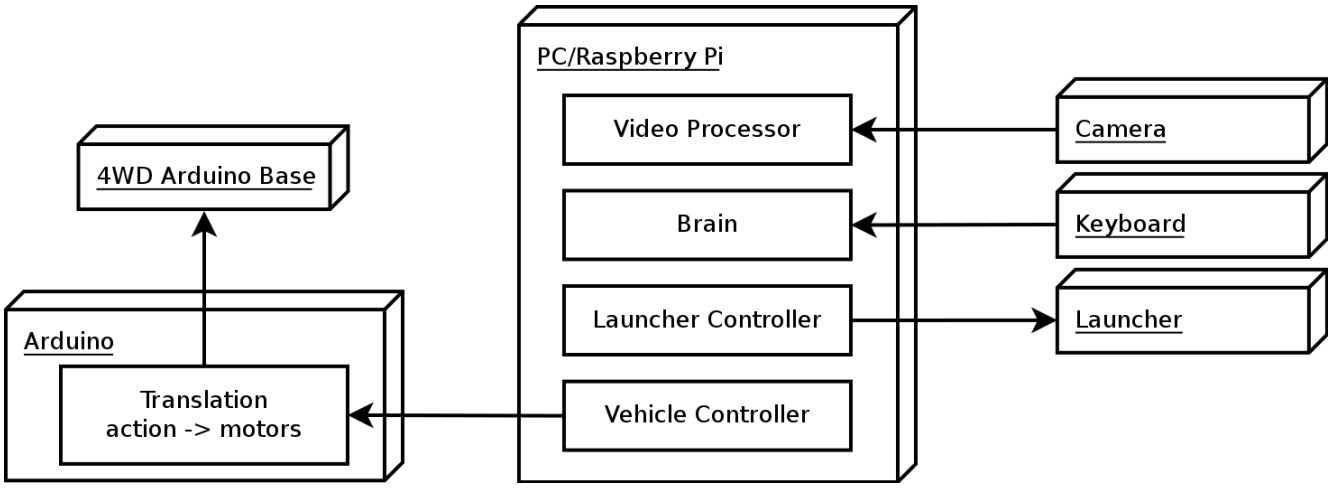


**Figure 2.9.:** Deployment view of the mobile missile launcher application.

Each of the components functionality is explained in table 2.1

## 2.6 Deploying and Running

The application was tested running under Ubuntu 16.04 and Rasbian Jessie, which is a Debian based Linux distribution for the Raspberry Pi. There are two different ways to deploy the software. The first one is the traditional way of building from source. The second one is using containerization. What it is and how it is done is explained in seciton 2.6.2.

### 2.6.1 Building from Source

The following assumes Linux as the host system.

The full source code for the mobile missile launcher application is provided with this thesis. To build the project from source all the dependencies have to be installed which are:

- **libusb-1.0** provides capabilities to control USB devices. This is done by writing commands to the serial port that the USB device is connected to.
- **OpenCV 2.4.9** an open-source image processing library.
- **SDL2** provides keyboard input event handling functionality. It is only used in manual mode which is to control the launcher manually via keyboard input.

To compile the source code a cmake file is provided. If the cmake file has problems finding the libraries the paths have to be configured manually. Cmake generates a makefile that is then used to build the project.

### 2.6.2 Using Docker

A more straight forward way for deployment is using containers. The benefit of containerization is that the application is essentially shipped as a self contained image that will run the same on virtually any system. A popular containerization platform is Docker[2].
Docker is lightweight, because unlike a virtual machine the host's kernel is shared directly with the container and so the amount of overhead is minimal. It is essentially a piece of software that wraps the application up in a standardized image. This image is then used to start a container which is a complete filesystem that contains everything the application needs to run. So everything that is required to run any container, is a local installation of docker on the host system.
For the launcher a Dockerfile is provided which can be used to automatically build a docker image, that contains the missile launcher application and all the necessary dependencies. A docker image is made up of different layers. The first layer is the base layer. For the launcher the base layer is an Ubuntu 16.04 image. This is essentially a stripped down version of Ubuntu 16.04. The other layers install the necessary dependencies (see section 2.6.1) within the image. When all the dependencies are installed the source code for the mobile missile launcher application is copied into the image.
For the commands to build the Docker image and run the container refer to the technical documentation provided with the source code. All the commands are explained on the main page. Looking at "Learn Docker by Example"[11] might also be helpful.
Note that Docker for Raspberry Pi is a new release at the time and is not fully featured yet. Therefore, it has some limitations due to the Raspberry Pi's ARM architecture. Currently, only images with Rasbian as their base image can run on the Raspberry Pi. This means, that it is mandatory to build the image on the Raspberry Pi and use Rasbian as the base image. Hence, with this thesis a second Dockerfile (specifically for the Raspberry Pi) is provided. The Rasbian Dockerfile can be found on the git branch "raspbian-docker" of the project. It is located in the same spot as the Ubuntu Dockerfile on the main branch. To build a Docker image for the Raspberry Pi one needs to switch said branch and then proceed as described in the documentation.
Once the Docker image is built it can be deployed to any machine by just copying it over (for the Raspberry Pi it can be copied between different Raspberry Pis). This means it would be possible to easily deploy the mobile missile launcher application to a swarm of robots by distributing the docker image.
Pre-built Docker images, with Ubuntu and Raspbian as the base images are provided with this thesis. Due to storage restriction on the CD they were saved on the Raspberry Pis SD card. They can be found on the Raspberry Pis desktop.

### 2.6.3 Running the Mobile Missile Launcher Application

To run the mobile missile launcher application in Docker container, the container needs to be started first. After that running the application is the same whether it is run natively or in a container. It accepts different flags to run in different modes:

- **manual mode** using the flag *-m*. In manual mode the robot can be controlled using the keyboard.
- **autonomous mode** using *-a*. Here the robot tries to shoot the target autonomously by using the state machine approach.
- **reinforcement learning mode** using *-r*. Here the robot tries to shoot the target using the reinforcement learning approach.

The Launcher application also accepts a *-h* flag which prints out a help text.

[2] `https://www.docker.com`

# 3 Experimental Setup and Results

After having built the robot and developed the software, an experiment was performed. The aim of the experiment was to find out and to visualize how fast and well the robot learns to perform the task of finding and shooting the target. In the following section 3.1, the setup of the experiment is explained. Subsequently, the results of the experiment are presented and discussed in section 3.2.

## 3.1 Setup

The goal of the experiment was to investigate the robot's ability to learn the task of finding and shooting the target in a specific environment. The impact of two different $\epsilon$-values on the learning was evaluated. The environment was defined to be an open space with a flat floor and the following constraints: The robot's distance towards the target should be approximately 130$cm$ and the robot should be placed at about a 45° angle towards the target (the setup is drafted in figure 3.1). The target image's dimensions should be 10cm x 15cm, and it should be located right at the camera's height with the target's center being about 16$cm$ above the floor.

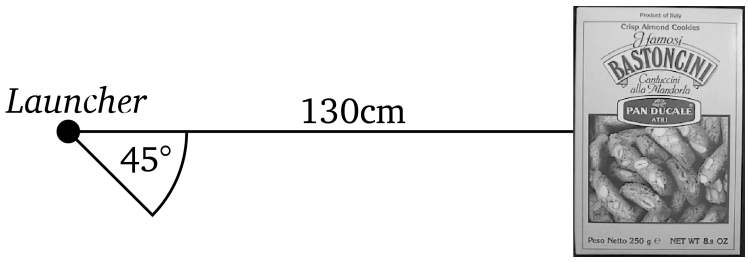


**Figure 3.1.:** Draft of the experimental setup.

The total reward earned over the course of one episode correlates to how well the robot performed at the task. The higher the total reward, the better the task is solved. In general, the more the robot learns the higher the total reward he earns should become. How fast and reliable the robot learns can be determined by monitoring the total rewards. The in the beginning inexperienced robot will gain experience over time and improve its total reward.
To set up the experiment, the first step is to reset the robot's $Q(s,a)$ function values to zero for every action in every state. This is essentially taking away his experience and making it *obtuse*. Then multiple training episodes have to be run and for each episode the total reward earned has to be saved. During these episodes the robot gains experience and should improve its ability to solve the task over time. In the experiment, two different $\epsilon$-values were used (0,02 and 0,1) in order to compare the outcome. For both $\epsilon$-values multiple sequences of 20 consecutive episodes each were run. In the following, one sequence of 20 consecutive episodes will be referred to as *one cycle*. After each cycle the $Q(s,a)$ function was reset and the next cycle was run. By plotting the total rewards over the course of one cycle, the robot's ability to learn can be visualized. The total rewards obtained during the experiment were saved in the "totalRewards_epsilon02.txt" and "totalRewards_epsilon10.txt" files respectively. Each line within these files represents a cycle while the columns represent the different episodes.
To obtain usable data it is necessary to run multiple cycles, since during each episode random steps are taken (because of exploration) and can therefore tremendously impact the reward. Even if the robot is already trained very well, a mistimed exploration action can result in a big negative reward. To deal with the variation that can occur between the different cycles, multiple cycles have to be recorded. Then an averaged cycle is calculated from the individual cycles. The average cycle represents the robot's ability and speed of learning and solving the task on average.
For the experiment 28 cycles were recorded while setting the Q-Learning parameters $\alpha$,$\gamma$ to 0.1. On half of the cycles $\epsilon$ was set to 0.02 and on the other half to 0.1.

## 3.2 Results

The result of the experiment with $\epsilon = 0.1$ is shown in figure 3.2. It shows that, just as expected, the total rewards increase over the course of several episodes. The first seven episodes can be identified as the main learning phase, where the total rewards increase the most. During the following episodes the total rewards remain relatively stable at a high level.

The error bars represent the standard deviation. Figure 3.2 shows that although the robot learns fast and reliable, the standard deviation is relatively high in some episodes, especially in the first five. Looking at the individual cycles, the reason for this becomes clear: The robot performs quite well on average, however, in those cases where it does not, the total reward will turn out significantly worse. The reason for its poor performance from time to time is because of exploration. While training the robot and collecting the data, it became clear that random steps at the beginning of an episode did not pose any difficulties to the robot. Random steps at later points, especially when the robot was very close to the target turned out to be fatal, however. For instance, a random left turn in the state *goodPosition* would cause the robot to drive by, thus, unable to find the target anymore. If the target had been shot, the reward would have been high, but when loosing the target, the reward would drop drastically, since the robot would drive in the wrong direction, loosing total reward until it shoots randomly. A bigger $\epsilon$ usually yields faster learning but it also comes with a bigger error rate when the random action taken turns out to be a poor choice.


**Figure 3.2.:** Average total rewards over the course of multiple episodes, for $\epsilon = 0.1$. The error bars show the standard deviation.

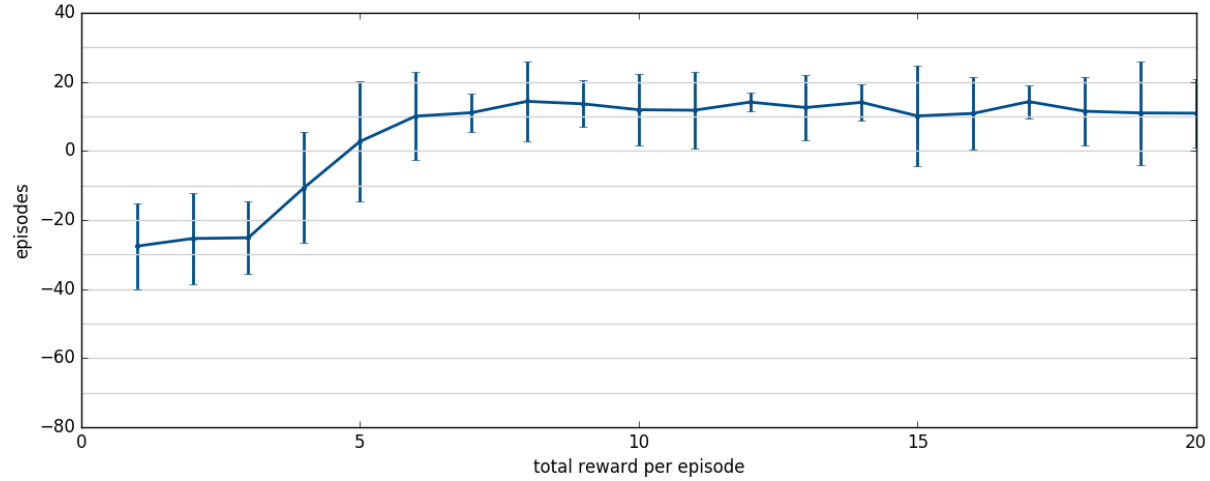


**Figure 3.3.:** Average total rewards over the course of multiple episodes, for $\epsilon = 0.02$. The error bars show the standard deviation.

The data generated in the second experiment, where $\epsilon = 0.02$ was used, can be seen in figure 3.3. The graph is similar to figure 3.2: The robot learns at a high level for the first seven episodes and then settles in at its maximum level of total rewards, which are higher than the one of $\epsilon = 0.1$. Additionally, it can be observed, that the standard deviation is significantly smaller for this $\epsilon$. This means, that the robot learns similarly well, but performs considerably more reliable. This is due to the fact that the task can be learned just as well and that exploration is performed less often, and therefore, does not lead to big negative rewards that often.

Figure 3.4 compares the two graphs. It can be seen, that for the first three steps $\epsilon = 0.1$ provides a much higher learning rate than $\epsilon = 0.02$. However, after episode three $\epsilon = 0.02$ is superior. This means, that less contingency makes the robot perform better at learning the task. Nonetheless, setting $\epsilon = 0$ would not be preferable, since without any exploration the robot would most likely not be able to solve the task at all. Instead of using $\epsilon$-greedy, a different strategy, where $\epsilon$ would decrease over time, might lead to even better results.

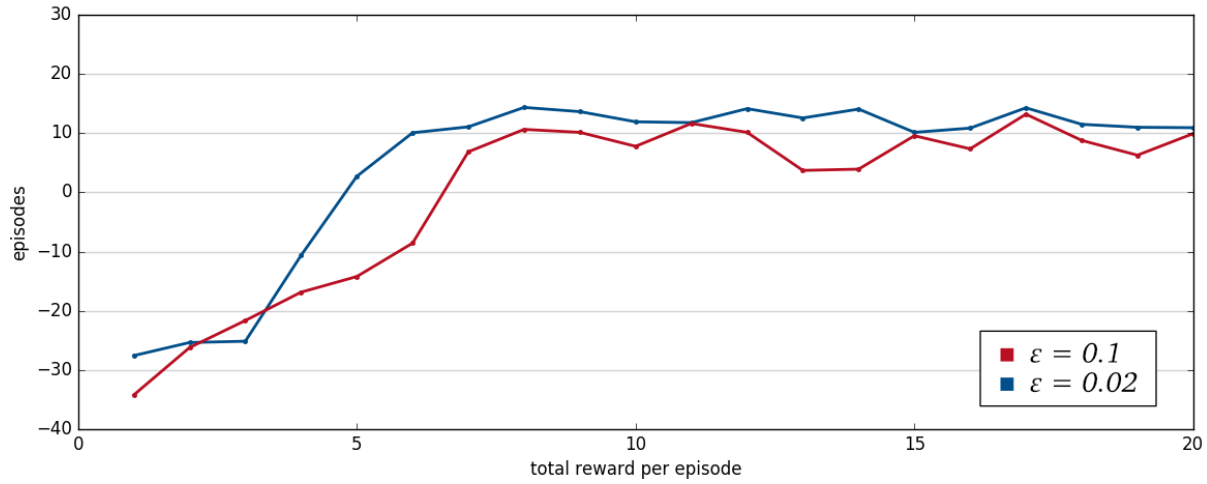


**Figure 3.4.:** Comparing total rewards earned for $\epsilon = 0.02$ and $\epsilon = 0.1$.

Towards the end of the robot's learning cycle, some typical trajectories start showing. These are the trajectories that turn out to be best suited for solving the task at hand. There are more than one, because the robot will sometimes detected the target at a sooner, and sometimes later point in time. This depends on a number of different factors, for example the lighting. If it detects the target, it will drive straight towards it, otherwise it will keep on driving the search pattern. All of the typical trajectories have in common that one can identify the predefined *zick-zack* search pattern within them.
Two of the most typical trajectories are shown in figures 3.5 and 3.6.
The trajectory in figure 3.5 shows the following: From step 1 to 8, the robot has not detect the target yet. It follows the predefined search pattern exploring its environment. After the 45° left turn in step 8, the robot detects the target. In step 9, executing the action *turnTowardTarget*, it turns the exact angle needed to face the target directly. Afterwards, it drives forward multiple times towards the target. Depending on how accurate the turning action in step 9 was it might also need to correct its angle between the different forward movements. The multiple forward and correction movements are combined in step 10. Finally, in step 11, the robot shoots the target.

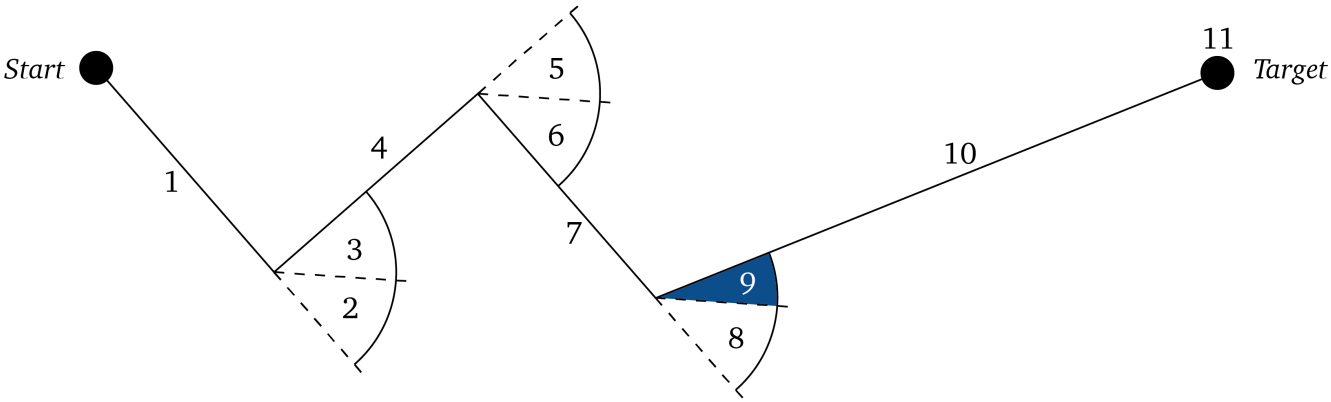


**Figure 3.5.:** Example of a typical trajectory the robot will follow while executing the task.

The trajectory in figure 3.6 differs therein, that the robot detects the target sooner in the process of exploring the environment. The first five steps are caused by following the search pattern. After the 45° right turn in step 5, the robot detects the target. It then corrects its angle by executing the *turnTowardTarget* action. In step 7, it drives toward the target (and possibly slightly correct's the angle along the way). Finally, in step 8, it shoots the target.

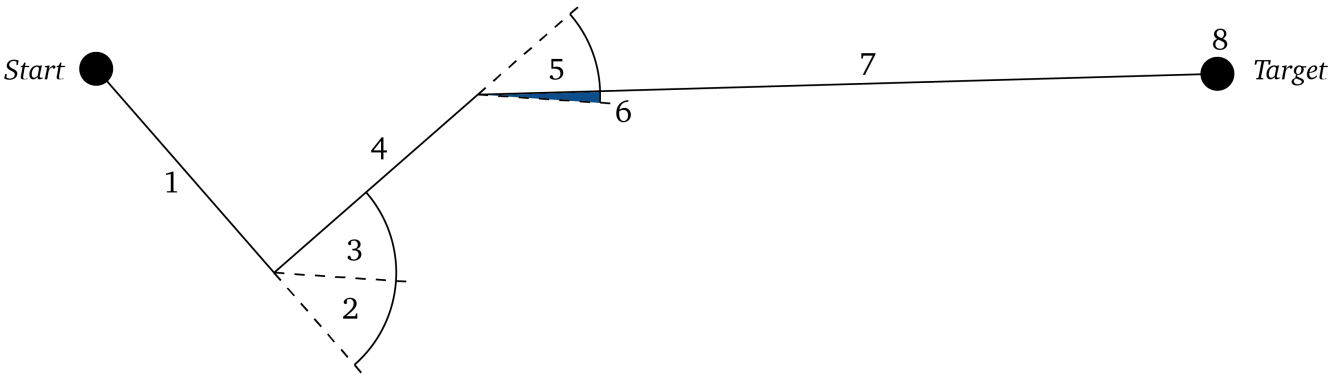


**Figure 3.6.:** Example of a typical trajectory the robot will follow while executing the task.

At the end of each episode the robot has been fully trained to perform the task. If $\epsilon$ and $\alpha$ were set to zero, the robot would just exploit what it has learned so far and stop exploring. That way, the target would be shot every time. The following table 3.1 shows an example of the $Q(s, a)$ function after one cycle of learning.

**Table 3.1.:** Example Q-Values learned after a cycle of 20 episodes.

| state/action | left | right | toward | fire | forward | backward | search |
|---|---|---|---|---|---|---|---|
| noTargetDetected: | -1.71142 | -1.7456 | -2.37835 | -9.41061 | -1.35053 | -1.17696 | -1.03652 |
| badPosition: | -0.1 | -0.1 | 3.16247 | 0 | 0.0296299 | 0 | -0.110245 |
| tooFarAway: | -0.3 | -0.784077 | -0.200634 | -0.1 | 3.20238 | 0 | 0 |
| goodPosition: | -0.28972 | -0.287748 | -0.1 | 7.52805 | 0 | 0 | -0.309892 |

Table 3.1 shows the $Q(s, a)$ function values. The robot's behavior can be derived from these values. The preferred actions are underlined. These are the options that were learned to be best in the given states.

Summarized, the robot learned to

- *search*, if no target was detected,
- *turn toward* the target, when the position is bad,
- drive *forward*, if it is too far away from the target,
- and *shoot*, when the position is good.

# 4 Conclusion

## 4.1 Summary & Discussion

Within the scope of this thesis a framework was developed that allows to control the launcher. The Arduino Base was calibrated using numerous test measurements and a linear regressing approach. The robot's exact turning behavior was calibrated for the fully charged batteries of the 4WD Arduino Base, therefore, the batteries have to be changed after only about 10 to 20 minutes for optimal results.
Two different approaches for controlling the launcher where implemented: state machine and reinforcement learning.
The state machine approach is characterized by non-flexible, predefined behavior. Nevertheless, it enables the robot to solve the task, provided that an appropriate search pattern (figure 2.7) has been configured in the properties.txt file (property robot_search_strategy, see appendix A).
In the next step the reinforcement learning approach was implemented. Characteristic of this approach is, that it enables the launcher to *learn* the task. As long as the robot has not detected the target, it uses the predefined search pattern to learn exploring its environment. Therefore, just as in the state machine approach, the robot needs this kind of *support*. Currently, it cannot learn a search pattern on its own, which prevents it from learning more complex tasks, where it has to drive bigger distances and plan ahead.
Lastly, containerization was used, in order to make deployment easy. Since Docker is new on the Raspberry Pi it turned out to have certain limitations, however.

## 4.2 Outlook

This thesis gives multiple opportunities for future work. The different starting points can be grouped into three sections: Hardware, Software, and the Task. The three groups are shortly discussed in the following sections.

### 4.2.1 Hardware

In future work, the robot could be improved hardware wise by switching components that turned out to be big constraints:

- A better **camera** with a higher resolutions, an autofocus and a better sensor. This could greatly improve the image processing and the overall robot. Then it might even be possible to explore different strategies, since detecting the target from greater distances was not a possibility so far. The probability of loosing the target object, when processing the next frame, would also decrease. Therefore, the robot could rely more on the information it receives from the camera.

- Different **sensors** could be added, which the robot could use to understand its environment. These could be proximity sensors, so it could have a sense of distance. A gyroscope would enable the robot to know exactly how far it turned. Pressure sensitive sensors would help the robot to detect when it drives against an obstacle.

- More **computing power** could be obtained by using a newer model of the Raspberry Pi. A more complex approach would be to use cloud computing. The hardware intensive image processing could be done in the cloud and the *slow* Raspberry Pi would have the task of sending the image data to the cloud and executing the actions calculated in the cloud.

- An **external battery** would open up the possibility of not having to connect the robot with any wires.

### 4.2.2 Software

Software wise, there are a number of things which could be addressed in future work in order to enable the robot to fulfill more complex tasks:

- The implemented reinforcement learning algorithm could be extended to incorporate *planning*. Currently the robot only learns one best action for each state. If it could plan ahead a number of steps, it might learn its own search pattern technique for the *noTargetDetected* state.
- The sampling process could be improved by combining different samples from different image processing algorithms. SIFT could for example be implemented and the results from SURF and SIFT could be combined.
- More states could be added to $S_m$ and $S_{rl}$. The robot could, for example, recognize when the angle towards the target is bigger than a certain threshold. This would allow it to position itself right in front of the target, before shooting.
- The other maneuvers of the USB Missile Launcher could be implemented.

### 4.2.3 Task

The task the robot is faced with, could be modified in order to make it more challenging:

- Obstacles, that the robot would need to navigate around, while searching for the target, could be added. It could either recognizing these obstacles using image processing or by adding proximity sensors.
- Since the USB Missile Launcher has multiple projectiles, the robot could be instructed to find and shoot multiple targets within one episode. That way it would not only be faced with the problem of finding the targets, but also with figuring out the right order in which to shoot them to earn the highest reward.

# Bibliography


[1] Sundar Pichai, CEO, Google, "This year's founders' letter." https://googleblog.blogspot.de/2016/04/this-years-founders-letter.html?m=1, 2016. Accessed: 2016-10-12.

[2] A. L. Samuel, "Some studies in machine learning using the game of checkers," *IBM J. Res. Dev.*, vol. 3, pp. 210–229, July 1959.

[3] A. Zimpfer, *Vergleich verschiedener Lernalgorithmen auf einem USB-Missilelauncher*. PhD thesis, 2012.

[4] S. Alexev, *Reinforcement Learning für eine mobile Raketenabschußplattform*. PhD thesis, 2015.

[5] H. Bay, A. Ess, T. Tuytelaars, and L. Van Gool, "Speeded-up robust features (surf)," *Comput. Vis. Image Underst.*, vol. 110, pp. 346–359, June 2008.

[6] O. G. Luo Juan, "A comparison of sift, pca-sift and surf," *International Journal of Image Processing (IJIP)*, pp. 143 – 152, 8 2009.

[7] M. Muja and D. G. Lowe, "Fast approximate nearest neighbors with automatic algorithm configuration," in *In VISAPP International Conference on Computer Vision Theory and Applications*, pp. 331–340, 2009.

[8] R. Sutton and A. Barto, *Reinforcement Learning: An Introduction*. A Bradford book, Bradford Book, 1998.

[9] C. J. C. H. Watkins, *Learning from Delayed Rewards*. PhD thesis, King's College, Cambridge, UK, May 1989.

[10] G. Starke, *Effektive Softwarearchitekturen: Ein praktischer Leitfaden*. Carl Hanser Verlag GmbH & Company KG, 2015.

[11] "Learn docker by example." https://docs.docker.com/engine/tutorials/. Accessed: 2016-10-01.

[12] K. Pohl, C. Rupp, and I. R. E. Board, *Basiswissen Requirements Engineering: Aus- und Weiterbildung zum "Certified Professional for Requirements Engineering"; Foundation Level nach IREB-Standard*. dpunkt-Verlag, 2015.

[13] B. Stroustrup, *Die C++-Programmiersprache*. Addison-Wesley, 1992.

[14] M. Rochkind, *UNIX Programmierung für Fortgeschrittene*. Carl Hanser Verlag GmbH & Company KG, 1988.

[15] H. Herold, *Linux-Unix-Kurzreferenz*. Linux Unix und seine Werkzeuge, Addison-Wesley, 1999.

[16] G. García, O. Suarez, J. Aranda, J. Tercero, I. Gracia, and N. Enano, *Learning Image Processing with OpenCV*. Community experience distilled, Packt Publishing, 2015.

[17] "libusb-1.0 api reference." http://libusb.sourceforge.net/api-1.0/api.html.

[18] "Opencv api reference." http://docs.opencv.org/2.4.9/.

[19] A. Rosebrock, "Install opencv and python on your raspberry pi 2 and b+." http://www.pyimagesearch.com/2015/02/23/install-opencv-and-python-on-your-raspberry-pi-2-and-b/, 2015. Accessed: 2016-10-01.

[20] M. Labbé, "Find-Object." http://introlab.github.io/find-object, 2011. Accessed: 2016-10-01.

[21] "Features2d + homography to find a known object." http://docs.opencv.org/2.4/doc/tutorials/features2d/feature_homography/feature_homography.html, 2016. Accessed: 2016-10-01.

# A Properties Configuration File

The code base of the mobile missile launcher application includes a properties.txt file. The file is a global spot to set a number of important properties. The most important properties are described in the table below.

**Table A.1.:** Description of the different properties in the properties.txt file

| Property | Description |
|---|---|
| webcam_device_name | The camera that should be used for the image processing. The default is zero 0, but if more than one camera are connected to the PC, or if there is a built in webcam this might need to be changed. |
| frame_skipping | This parameter is used to turn frame skipping on and off. It is explained in the documentation of the VideoProcessor::getNextFrameFromCamera() function. It is recommended to leave frame skipping on. |
| threashold_multiplicator | This parameter is a factor that is multiplied with the frame skipping threshold. It is explained in the documentation of the VideoProcessor::getNextFrameFromCamera() function. |
| max_buffer_size | This parameter represents the maximum buffer size of the OpenCV Capture buffer. It is explained in the documentation of the VideoProcessor::getNextFrameFromCamera() function. |
| frame_debugging_output | This parameter can turn the debugging output for frame skipping on and off. It is used in the VideoProcessor::getNextFrameFromCamera() function. The parameter can be used to debug the frame skipping process. |
| vehicle_port | The port that the Arduino is connected to. Default is "/dev/ttyACM0". |
| vehicle_turn_left_intersect | Intersection value of the Arduino Base's turn left function. |
| vehicle_turn_left_slope | Slope value of the Arduino Base's turn left function. |
| vehicle_turn_right_intersect | Intersection value of the Arduino Base's turn right function. |
| vehicle_turn_right_slope | Slope value of the Arduino Base's turn right function. |
| vp_target_image_path | The path to the target image. By default it is set to "targets/box.png". |
| vp_min_Hessian | Property for the SURF and FLANN image processing. Default is 500. |
| vp_sample_size | The sample size of the sampling process (section 2.2.1) |
| robot_search_strategy | This property holds the predefined robot's search strategy that is needed for both approaches. The search strategy is represented in the form of a string. Each character represents one action, that will be executed once, when calling the *searchSystematically()* function. Characters f (forward), b (backward), l (45° left turn) and right (45° right turn) can be used within this string. The default setting is "fllfrr", which corresponds to the *zick-zack* patter discussed in the thesis. |
| rl_alpha | $\alpha$ value of the Q-learning reinforcement learning algorithm. |
| rl_gamma | $\gamma$ value of the Q-learning reinforcement learning algorithm. |
| rl_epsilon | $\epsilon$ value of the Q-learning reinforcement learning algorithm. |
| rl_qValues_path | Path to the saved Q-values. |

# B Project File Structure

The following show the projects file structure. The most important files and folders are briefly explained here.

- **Arduino/**
  This folder contains the software that runs on the Arduino.
- **Docker/**
  This folder contains the pre built Docker images. One for the Rasberry Pi and one for other Linux distributions, using Ubuntu as the base image.
- **Documentation/**
  This folder contains a complete documentation of the source code in form of an html page. The documentation can be viewed by opening the *html/index.html* file.
- **Raspberry/**
  This folder contains the entire application's code and files.
  - **resources/**
    - **calibration/**
      This folder contains the measurements that were taken for calibrating the Arduino Base and a python script to evaluate the measurements.
    - **Targets/**
      This folder contains the target image.
    - **training/**
      This folder contains several file for the reinforcement learning.
      - **qValues/**
        This folder contains the qValues that were learned during each training run.
      - **trainingRuns/**
        This folder contains the output logs of the individual training runs. Each training run can be completely reconstructed using these files.
      - **averageLearning.py**
        This python script build the graphs that show the experiments results.
      - **learn.sh**
        This bash script was used to run the experiment. It executes 20 episodes in a row and saves the results of each episode.
      - **qValues.txt**
        This file contains the Q-values that the application will use, when in reinforcement learning mode.
      - **totalReward_epsilon02.txt**
        This file contains the total reward results of the experiment for $\epsilon = 0.02$.
      - **totalReward_epsilon10.txt**
        This file contains the total reward results of the experiment for $\epsilon = 0.1$.
    - **properties.txt**
      This file contains global properties for the application. Its content is explained in appendix A
  - **src/**
    This folder contains the application's source files.
  - **CMakeLists.txt**
    This file is used by cmake and contains the building instructions.
  - **Dockerfile**
    This file is used by Docker to build the Docker image, see section 2.6.2.