

Reinforcement Learning with LEGO Mindstorms

Project Report

Per Steffen Czolbe (wjq874)

Handed in: January 4, 2019



1 Abstract

Contents

1	Abstract	1
2	Introduction	2
2.1	EV3 robot	2
3	Development Setup	2
3.1	Ev3dev Platform	3
3.2	Control by external Computer	3
4	Sensors and Motors	3
4.1	Time Delays	3
4.2	MOVE: The Colour Sensor	4
4.3	Motors	5
4.4	Gyroscope accuracy	7
4.5	Ultrasonic Sensor	7
4.6	Calibration	8
5	Colour Detection	9
5.1	Problem	9
5.2	The Robot	9
5.3	The Learning	9
5.4	Results	11
5.5	Discussion	11
6	Crawl-Robot	13
6.1	Robot Design	13
6.2	Formulation of the Learning Problem	14
6.3	Results	15
6.4	Discussion	16
7	Swing-Robot	18
7.1	Problem	18
7.2	The Robot	18
7.3	The Learning	18
7.4	Results	18
7.5	Discussion	18
8	Future Work	18

2 Introduction

scientific goal, educational goals, why machine learning for school, why LEGO, agenda by lego, why reinforcement learning, brief intro to machine learning, outline of the report, reference to github

2.1 EV3 robot

The EV3 system is the third version of the LEGO Mindstorm family of robot toys. It contains a central processing component, called "Brick", to which motors, sensors and other peripheral devices can be attached. All these components are embedded into plastic casings, which make them compatible with other LEGO parts. The LEGO Group markets the EV3 system to educational facilities and private persons in a package with build instructions and parts for multiple robots. The current product offering consists of 2 different packages to build a total of 9 robots, including the self-balancing two wheeled "Gyroboy" and a sorting machine that orders LEGO bricks based on their colour. The playful development of further robots by end users is encouraged. A large community of engaged hobby robot builders has formed around the EV3 platform.

Hardware

The Brick is the central processing component of each EV3 robot. It can be controlled via multiple buttons, features a small LCD screen and can be connected to a computer via Wifi, Bluetooth or USB. For peripherals, up to 4 motors and 4 Sensors can be connected. The specifications of the embedded system are [1]:

CPU: 32bit, 300MHz ARM9 Processor

Memory: 64Mb DDR RAM

Storage: 16Mb Flash, up to 32Gb SD-Card

Communication: USB, Bluetooth, Wifi

Peripherals: 4 motors, 4 Sensors

Software

The LEGO company ships the EV3 platform with a software system centred around its own proprietary graphical programming language based on LabVIEW. The graphical interface enables people with limited programming experience to quickly develop simple programs that can be executed on the Brick. The graphical language is supported by an editor available as a program for Windows and Mac, and as an app for Android and iOS.

While the default software is well supported by the editor and operating system of the brick, more experienced programmers might prefer a more capable language. Multiple community driven software packages exist for this purpose. These support languages such as matlab, python, java, javascript, C++, C, GO, Ruby, Perl, R, Lua, even direct control of the robot via command line is possible [4]. We discuss a specific development set up in the next section.

3 Development Setup

In this section we present the development set up chosen for this project. It consists of the open source operating system ev3dev to power the brick, the ev3dev python library to control motors and sensors, and remote procedure calls to communicate with the robot from a host computer the via jupyter notebook.

3.1 Ev3dev Platform

Ev3dev is a Debian Linux-based operating system that runs on several LEGO Mindstorms compatible platforms including the LEGO Mindstorms EV3 and Raspberry Pi-powered BrickPi. It is maintained by a community of enthusiasts and offers libraries to interact with LEGO Mindstorms motors and sensors in many programming languages, such as Java, Ruby and Python [4]. The Linux disk image can be downloaded at www.ev3dev.org and flashed on a SD card. Once the SD card is inserted into the LEGO Mindstorms brick, the brick will boot into the ev3dev operating system. A set up guide with detailed explanations on installation and guides on how to get started with development are available at the ev3dev website.

3.2 Control by external Computer

While the brick, after flashed with the ev3dev image, runs a Debian Linux system that can execute general purpose programs, the processing and memory constraints imposed by the hardware limit the complexity of programs executed on the brick severely. Installing a python package takes 2 minutes. Executing more computationally expensive tasks like deep learning or real time video processing on the brick is not feasible.

To circumvent these problems, a setup of the main program running on a computer and interacting with the brick to move motors or read sensor data has been employed by multiple community projects, such as a rubics cube solver [3]. We follow the design approach pioneered by these projects. The brick runs a remote procedure call server, implemented by the python3 library `rpyc`. This allows a host computer to connect to the brick and remotely interact with motors and sensors. All program logic is executed on the computer, the brick is listening to commands. Next to the performance improvements, another benefit is that the program can be executed in a jupyter notebook on the computer, which allows to better integrate documentation and visualization into the development process.

4 Sensors and Motors

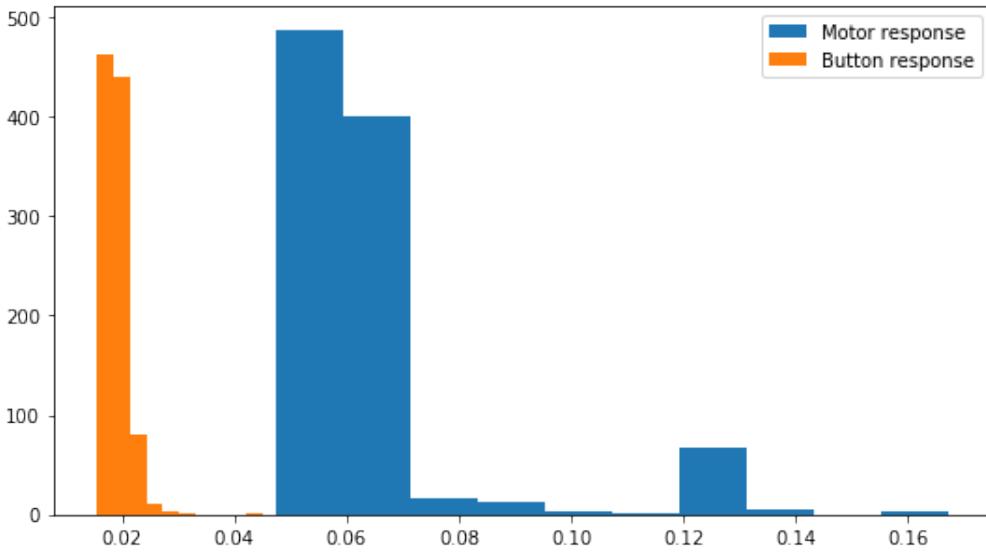
In this section we examine the peripheral devices of the EV3 platform. We examine the behaviour of motors and sensors, design experiments to measure accuracy and time delays. We conclude this section by discussing the need for calibration of some sensors.

4.1 Time Delays

We tested the response speed of the motors, buttons and the colour sensor, the code for which can be found commented in experiments/Response speed experiment.ipynb.

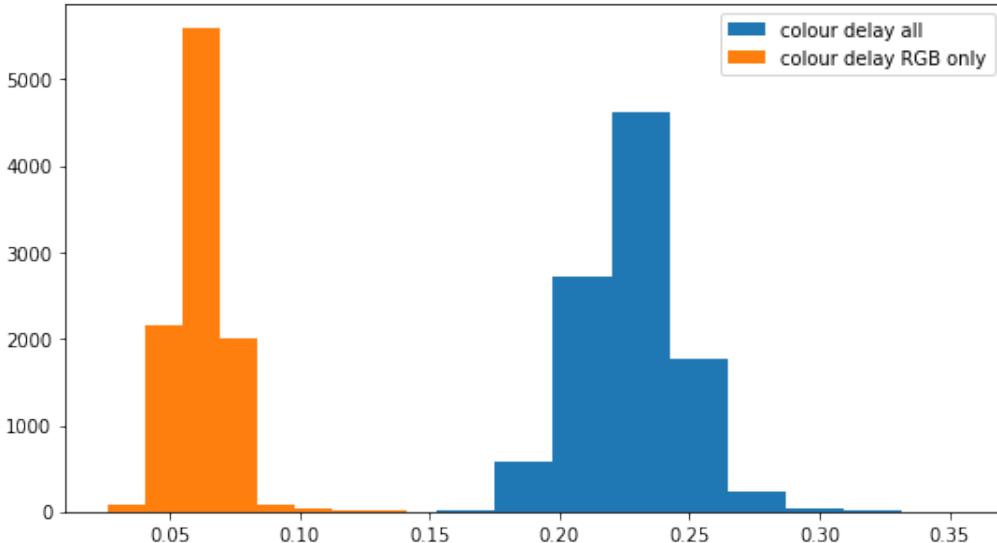
For the button, a single button was connected, and we measured the time until the robot noticed that the button wasn't pressed, and repeated, this setup was very simple and was repeated 1000 times taking 19 seconds. Mean button response was $0,019$ seconds with a very low variance at $4,8 \cdot 10^{-6}$.

Motor reaction speed was tested with a slightly more complicated setup, in which a large motor had an arm attached, that had to move to hit a button. The arm was positioned right in front of the button, such that any amount of movement would set off the button, and it would reset position after each round. A gif of this setup can be found in the jupyter notebook. Motor response testing was repeated 1000 times, and took 85 seconds. Since the time involved a button delay, a value without this was calculated by subtracting the button delay mean. The mean of the new motor-only delay was $0,066$ and the motor had much higher variance than the button at $3,6 \cdot 10^{-4}$ and a 95% percentile of $0,124$. This is also clearly visible in the below plot.



From this it appears that motor response is not only slower, but also a bit more unstable than the button response. Note though, that more factors, such as speed and real-world interference could have affected the motor response speed experiment. Still, even if assuming the 95% percentile, we should be able to get at least 8 actions per second.

The colour sensor turned out to react slower than the button, probably due to the fact that it has to cycle between different LEDs. An experiment was made in which a colour sensor made an RGB observation and this time was logged. It was compared to an experiment in which it observed both RGB, reflected light and ambient light. Both were run 10.000 times, taking 10 and 38 minutes respectively.



With only RGB, we observed a mean response time of 0, 062 seconds, which is very comparable to the motor response time, but with a lower variance at $1, 1 \cdot 10^{-4}$. With everything, the mean response was 0, 228 seconds and a variance of $3, 8 \cdot 10^{-4}$. It might be possible to reduce the delay by storing information on the brick such that the transfer between brick and computer only happens once, but since the sensor uses different lights for each observation, time would still be significantly longer than RGB only.

4.2 MOVE: The Colour Sensor

Calling values from the sensor will cause it to shine one of or all three different colours of light and measure values of the reflections. The sensor can measure colours, ambient light and reflected light, but since these are



Figure 1: Overview of current (2018) LEGO motors. Upper row: Power Functions large and X-large motor. Bottom row: EV3 medium and large motor. Images from [7].

three different modes, calling all of them incurs triple sensor delay, which may be too slow for real-time use. The colour sensor has a built-in list of possible colours, and can measure and predict white, blue, black, green, red, yellow, brown and no colour. Since the sensor measures reflection, pointing it into the air will measure (close to) (0,0,0) which it categorizes as no colour.

4.3 Motors

The LEGO product line includes a variety of 9V DC electrical motors, which can be used to power a wide range of LEGO models, such as cars, trains and robots. The current product offering consists of two main families: “Power Functions” motors and “EV3” servo motors. The most common sizes of these product lines are shown in figure 1.

Power Functions Motors

The “Power Functions” line of motors is designed to work with a battery pack and is intended to be controlled either with a physical switch on the battery pack or a remote control. Adapters for the power cables are required to use them in combination with the EV3 platform. [11]

By changing the current and polarity of the DC power supply, these motors can spin in both directions and different speeds. Because ‘Power Functions’ motors do not measure rotation or angle, robots utilizing them often require other means of measuring mechanical movements. Compared to the EV3 servo motors, they offer similar torque and mechanical power, have faster reaction speeds but offer no feedback. [7]

EV3 Servo Motors

The EV3 robot kit contains a medium and two large servo motors. The EV3 motors use tacho feedback to measure their alignment and rotation, which allows for more precise control compared to “Power Functions” motors. [6]. In strength and speed they are comparable to their “Power Functions” counterparts. The large EV3 motor is roughly twice as powerful as the medium one, and comes in a bigger case with different mounting points. Detailed comparisons of the mechanical and electrical properties of all LEGO motors is available at [7].

The ev3dev programming environment offers functions for low level interactions with the EV3 motors. Available are methods to start and stop the motor, reading the angle, setting the speed and three different stopping patterns (coast, brake and hold). The medium and large EV3 Motor offer identical programming interfaces. For angle measurements, the motor saves the rotation at system start as 0 degrees. During operation, the current rotation angle offset to the starting point can be read by the motor. [2]

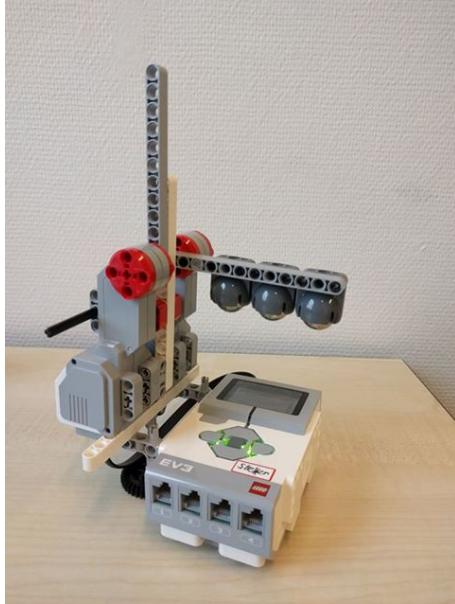


Figure 2: Experiment to measure motor accuracy. The two arms (gray) swing back and forth in a 90 degree angle. One arm is weighted down by three heavy metal balls.

Motor Accuracy

Reinforcement learning algorithms rely on repetition of the same motions many times. To assume a non-adversarial environment and ensure reproducible results, actions taken by the motors need to be accurate and consistent. We design an experiment to measure the accuracy of the motors after a series of repeated actions.

Experimental Setup

The experiment consists of an arm attached to a motor. We rotate the arm in a 90 degree angle back and forth. After every 25 iterations we measure the offset from the starting position externally with a set square. To measure the influence of weight on the results we use two independent arms, one without a weight and the other one weighted by three heavy metals balls. The experimental set-up is shown in figure 2.

The ev3dev python bindings offer two different movement patterns: '`on_for_degrees()`', which rotates the motor relative to the current position, and '`move_to_pos()`', which rotates the motors to an absolute value. Further, three stopping patterns exists: '`brake`', '`hold`' and '`coast`'. In the experiment we will test different combinations of movement and holding patterns, to determine the configuration with the highest amount of accuracy and reproducibility. Since we want high accuracy, we do not test the '`coast`' setting. The coasting option allows the motors to '`coast`' until standstill, instead of enforcing them to stop at a specific position.

Move Pattern	Stop Pattern	Weight	inaccuracy after 25 iterations	after 50it	after 75it	after 100it
relative	hold	no	1°	1°	2°	2°
relative	hold	yes	9°	11°	35°	45°
relative	brake	no	1°	1°	1°	2°
relative	brake	yes	35°	50°	>50°	>50°
absolute	hold	no	0°	1°	2°	1°
absolute	hold	yes	2°	1°	4°	3°
absolute	brake	no	1°	2°	2°	3°
absolute	brake	yes	4°	6°	5°	4°

Table 1: Experimental result. Angle offset in degrees after 25, 50, 75 and 100 iterations for different configurations.

Results

The angle offsets measured after 25, 50, 75 and 100 iterations for different motor control settings and configurations are shown in table 2. As we can see, the non-weighted arm is very accurate in all scenarios. The results for the weighted arm differ drastically between configurations. We can observe two trends:

1. The '`brake`' stopping pattern creates more offset than the '`hold`' pattern. Further observations of the experiment let us conclude that '`brake`' stops the motion of the motor, but does not block the motor for further movement after the initial momentum has been eliminated. This leads to the weighted arm pulling slightly further down between the breaking phase and the next swing of the arm. The '`hold`' stop motion holds the arm in place even after the initial momentum is eliminated. It leads to more accurate results under load.
2. Movements of absolute positions are more precise than relative movements. While inaccuracies of relative movements accumulate over time, movements to absolute positions only carry the inaccuracy of the last movement.

During the experiment we also measured the range of motion that is possible without receiving any push-back from the motors. This amount of slack is consistently measured at about 12 degrees. No configuration lowered this inaccuracy. If it causes a problem later on, mechanical solutions such as a worm gear could be used to reduce the slack.

In conclusion, to attain the highest level of accuracy and reproducibility, motors should be moved to absolute positions and the stop pattern '`hold`' should be used. We note however that motors still allow for free movement of about 12 degrees without providing any breaking or pushback.

4.4 Gyroscope accuracy

4.5 Ultrasonic Sensor

The EV3 Ultrasonic Sensor generates sound waves and reads their echoes to detect and measure distance from objects. It can also send single sound waves to work as sonar or listen for a sound wave that triggers the start of a program. Distances measured range from 0-250cm. While the python bindings of this sensor return a distance measurement with an accuracy of up to 1mm [2], LEGO advertises an accuracy of "up to +/-1cm". [10]

While working with the sensor on the crawling robot, we noticed the importance of mounting the sensor on a stable frame. It should not tilt vertically or horizontally, as any change in angles also changes the distance to a target. To attain accurate and reliable distance measurements, the target should be a rather large, solid plane, oriented orthogonal to the ultrasonic waves emitted by the sensor. The accuracy is sufficient to measure sub-centimetre movements. Roughly 0.1% of measurements were faulty and reported the maximum distance value of 255cm. No further experiments to evaluate the accuracy of the ultrasonic sensor have been performed.



Figure 3: The EV3 ultrasonic sensor. Image from [10].

4.6 Calibration

Servo motors and gyroscope sensors are initialized when the brick is restarted. All other sensors require no calibration.

On system start, the position of the motor is initialized as 0 degrees. If the robot is dependent on absolute positional values, it is important to calibrate the 0-point. This can either be done by moving the motor forcefully into the correct position before starting the brick, or using more elaborate constructions of moving the motor against a button, until the button is pressed. This gives certainty over the 0-position of the motor and can be used to calibrate the rotation.

The gyroscope initializes its orientation at system start as 0 degrees. Further, it assumes to be not in motion during system start. All acceleration measurements are in relation to this value taken during system start. Since the rotational angle returned by the gyroscope is calculated as the integral over acceleration measurements, even a slight inaccuracy in acceleration will accumulate to a significant drift in angle over time. This drift has to be monitored and adjusted for. This can be done either programmatic, or by restarting the system to re-calibrate the sensor.

5 Colour Detection

5.1 Problem

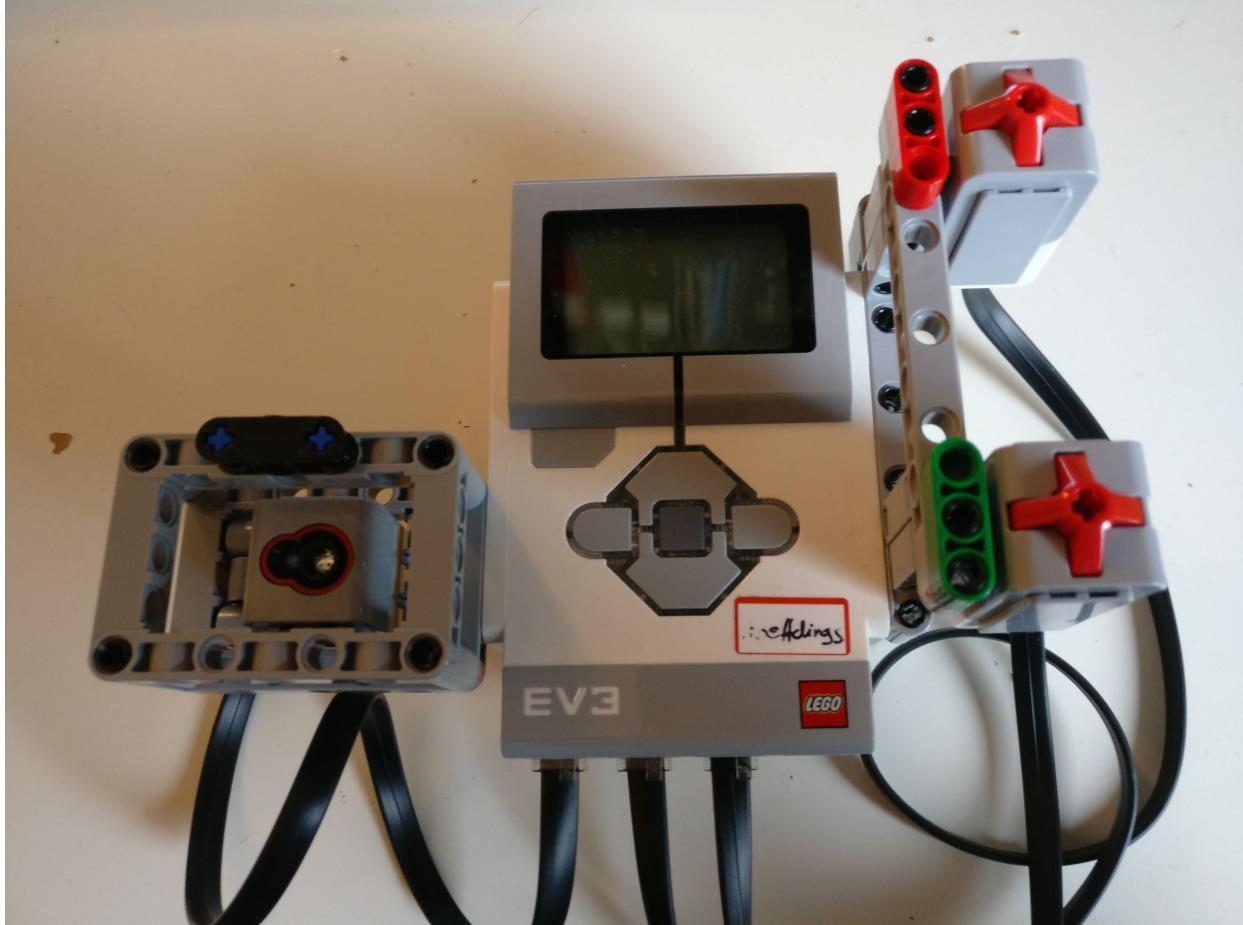
The colour sensor was chosen as an option for easy-entry into simple machine learning. We wanted to be able to visualize how machine learning updated after data was added and what the basis for decisions were, using a simple and easy to understand setup.

5.2 The Robot

The robot was made to be easy to use and consistent to allow similar observations from the colour sensor at different times.

A cage was made for the sensor, that easily plugged into the robot, and which could provide readings from everything placed on top of it, allowing for regular distances to the observed object at one regular lego technic block. Two buttons where added to the side of the robot for controls.

The final design is simple and modular, allowing it to be easily disassembled into smaller parts and reassembled later.



something about figure number and description?

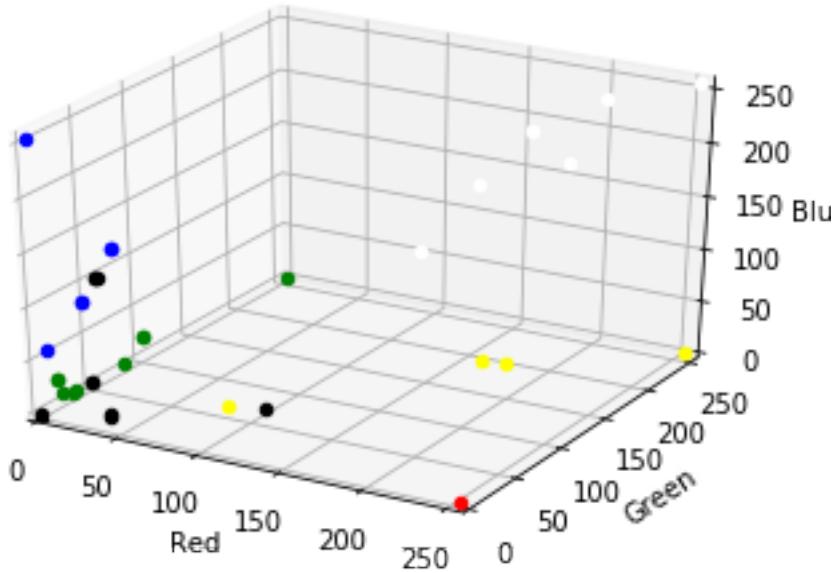
Since the robot has no moving parts, the results should be replicateable as along as the robot is built with 2 buttons and a color sensor, and the matter is kept still at a distance of one block.

5.3 The Learning

Since we wanted to visualize how learning worked, we implemented KNN, as one of the simplest and most intuitive machine learning algorithms. For the data input we used only the RGB output from the color sensor, as we wanted to plot in real time. For visualization, we used a 3d-plot of the observed colors, with the axes being

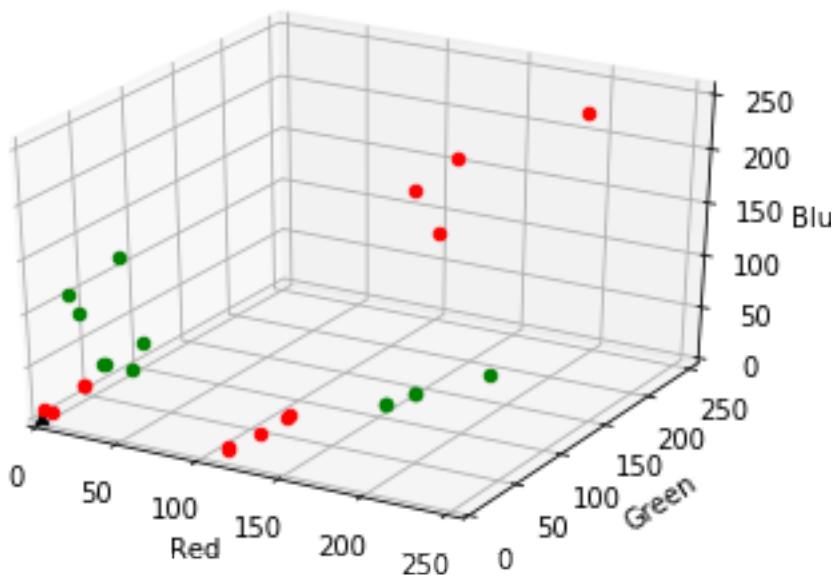
red, green and blue values.

We made two versions of KNN with the color sensor, found commented in simple_learning/KNN colors.ipynb. The first version is slightly silly, but colorful. Our dataset is started with either a base set (simple RGB values) or a few observations, in which we use the color sensor's built in prediction. Real-time KNN is then run, and we output our prediction as well as a 3d plot of our current observation (black triangle), together with all previous observations (coloured dots). Additional dots can be added by pressing a button, which adds the current observation and prediction as a true point in the data. This produces colorful plots, like the below plot.



something about figure number and description?

Since the above version uses starting data and the built-in predictions from lego, we also tried a different setup. Instead of prediction colours directly, we predict colour groups, which we will call group "red" and group "green". In this setup, the robot again predicts and plots in real time, but we now start with zero data, and rely only on the buttons. One button adds the prediction to the data set with label "red" and the other "green". This allows us to not rely on anything from lego, and allows us to make different splits, such as light versus dark colours, which could learn to differentiate light and dark blue. The plots are similar to before, but with fewer colours.



something about figure number and description?

We also tried implementing a more complicated system to work with other visualizations. We implemented a neural network, and wanted to visualize how learning and training works in this setup. The code for this is commented in reinforcement_colors/reinforced_colors.ipynb.

In this case, the robot has a list of allowed colors, starts with no data, and predicts from an untrained network. It reads RGB colours, but also reflected light and ambient light, since we are not running real-time. It displays scores for each color, and reads the highest scoring color aloud. A button press indicates whether this was correct or wrong, and it trains based on this feedback. This allows the user to view updates in predictions from step to step. The predictions are output on a list like this

```
'Black : 0.13146167'  
'Blue : 0.20897976'  
'Green : 0.15094219'  
'Red : 0.22930987'  
'White : 0.14744666'  
'Yellow : 0.13185982'
```

which updates with every new prediction.

5.4 Results

Two versions of a robot that uses KNN were implemented and built with visualization in mind. Both versions are functional, and can be used to explain and link how data affects predictions in machine learning.

We implemented a simple Neural Network model, that may be useful in explaining more complicated machine learning models.

We also learned a great deal about visualization and limitations to this, which will be presented and discussed in the following section.

5.5 Discussion

We encountered a number of difficulties related to these experiments, but have also found many avenues of possible future work.

For visualization, we originally wanted to show the decision boundary for our learning system, to show how it changes with time. Unfortunately, we found no good way to visualize this in 3d, so instead we looked into dimensionality reduction. We tried reducing dimensionality using PCA, and plot a decision boundary in the resulting 2d space, but this proved problematic, both in our setup and generally. Since we wanted to run real-time we had issues running the PCA on the decision boundary. To get enough points for a proper boundary in 3d space we had to use at least 1 million points, which took way too long to process to be able to run real-time. If instead we didn't run real-time, we could plot the PCA values and a decision boundary in this space, but the values would be mostly meaningless. In the 3d space it at least somewhat makes sense, you have an intuition about what changes if we increase on the "red" axis and so on. In the PCA space we're left with principal components instead, and the points from previous observations would not even be stationary.

Some ways to get around this issue would be to only read two colour channels (removing the need to reduce dimensionality), or find another experiment which only takes two inputs.

The neural network setup was made to demonstrate that the colour sensor setup can be extended to more complicated machine learning systems, to show some width of the space of possible education. It was implemented, but as a very simple form. It takes a number of colours initially, and predicts between these, but for more than 2 colours (or colour groups) this makes training complicated. The model was implemented such that if it predicted the first colour on our list, and we respond correct, it trains with (1,0,0,0) (or similar for more colours) while

instead if we respond wrong, it trains (0,1,1,1) (so it improves every other colour). This setup is not optimal, and is simply a POC. If instead we want to focus on this task to improve it, it may be able to use methods from reinforcement or online learning, since we're dealing with a reduced feedback (right/wrong) setup (this was an idea from the start, as is obvious from the file name). The reason this wasn't implemented was a time constraint built into the setup. At the moment, prediction is done by pressing both buttons, which causes the prediction list to update, and the robot to say the highest scoring colour. At this point you can press a button for yes or no. Unfortunately, you need a lot of observations to properly train the network, which makes it very time consuming and not very education-friendly. We ran a test with 150 entries, observing only 3 different colours. After this, it could still not differentiate the 3 colours. We believe that implementing a proper reinforcement or bandit setup would greatly improve the training, but 150 observations or just 50 that all have to be done manually is still a significant barrier for the neural network setup. Other options could be to pre-load it with some amount of data, but this would remove the chance to see early learning. The output for the neural network model could of course be combined with a plot like the 3d we made for the KNN model, or a PCA/decision boundary or similar, with most of the same yields and issues as discussed before. We believe the neural network setup has potential, but has some hurdles to overcome before it can be useful in an educational setting.

6 Crawl-Robot

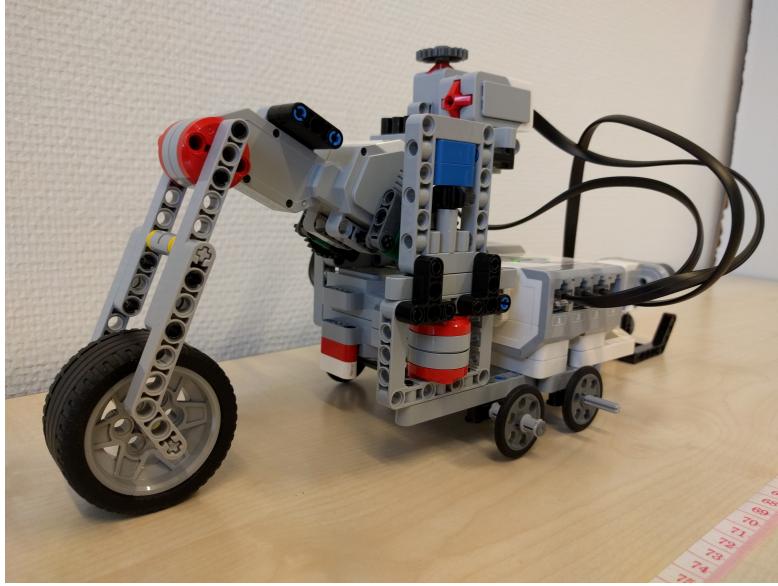


Figure 4: Crawl Robot, frontal view.

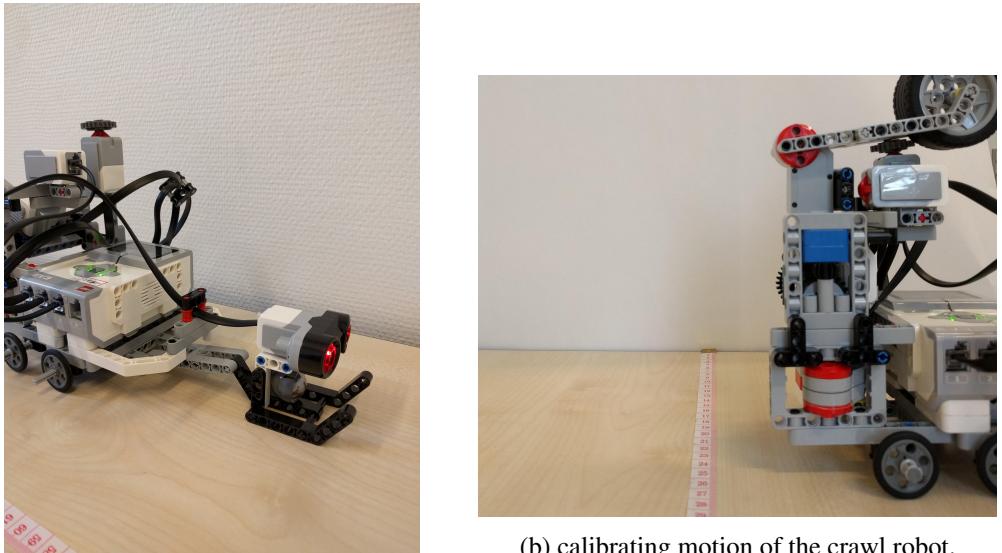
To demonstrate reinforcement learning in educational settings, we seek to have a physical model for demonstration. The training process should be robust, as to deliver a reliable demonstration in teaching scenarios outside of laboratories, and fast, to keep the attention of an audience. While many robot designs and learning goals, like pole balancing or swinging, are possible, we selected a crawling robot as a suitable object for demonstration. The concept of a robot using one arm to crawl, or drag itself along a linear track has been explored by multiple other studies [8] [12]. We selected this task, as it is a comparatively simple design, with few moving parts. The system is more robust towards external force from curious bystanders than for example a loosely swinging pole. By Defining a limited state space we can simplify the problem and get a training time suitable for demonstration. Realizing this robot will provide us with a model for demonstration and equip us with the experience and knowledge to solve more complex problems in the future.

6.1 Robot Design

To facilitate the crawling motion, the robot needs a movable arm to pull itself forward. The arm consist of two elements: the base arm, which is attached to the robot, and a pivot arm attached to the base arm. Both arm joints rotate around the same axis, allowing the hand at the far end of the pivot to move on a plane. To allow the arm to pull the robot, the ground friction of the arm has to be maximized, and the ground friction of the robot minimized. This is achieved by a big rubber tyre firmly attached (not spinning) to the robots hand. The robot body sits on wheels which can spin freely. See Figure 4 for a photo of the robot arm, hand and wheel basis.

To measure distance travelled, the ultrasonic sensor is used. This sensor is attached to the back of the robot and records the distance to a target from the rear end of the robot. As the wheel basis of the robot might tilt when the arm is pressing into the ground, while the ultrasonic sensor requires a stable angle towards the target for consistent measurements, the sensor is mounted on a sled which is pulled behind the robot. The sled is visible in figure 5a.

To calibrate the motors of the arm, two buttons are attached to the top of the robot. During the start up process, the arm is moved all the way back, until base arm and pivot arm hit their respective buttons. This motion is depicted in Figure 5b. The certainty about the arm position attained by this action is used to initialize the 0-angle of both motors.



(a) Ultrasonic sensor on a sled at the back of the robot.

(b) calibrating motion of the crawl robot.

Figure 5: Designs of the Crawl Robot

6.2 Formulation of the Learning Problem

To train the robot how to crawl, we use Q-learning. Q-learning is a form of reinforcement learning. It provides agents with the capability of learning to act optimally in Markovian domains by experiencing the consequences of actions. To learn, an agent tries an action at a particular state, and evaluates its consequences in terms of the immediate reward or penalty it receives and its estimate of the value of the state to which it is taken. By trying all actions A in all states S repeatedly, it models the value $Q(s, a)$ of each action state pair. The algorithm learns which state-actions pairs are the best overall, judged by long-term discounted reward [13].

The Q-learning algorithm fits the domain of the crawl robot. The implementation chosen is shown in Algorithm 1 and follows [9]. Training is grouped into epochs, and each epoch consists of multiple steps (loops in line 2 and 4 of the algorithm). In each step, we take one action. This action is determined with the ϵ -greedy strategy (lines 5-10), where with probability ϵ we perform a random action, and with probability $1 - \epsilon$ we greedily select the action a with the highest current value. In line 11 we perform the sampled action a , observe the reward r and transition into the next state s' . In line 12 the state value function $Q(s, a)$ is updated. We end the step by setting $s \leftarrow s'$ and then repeating (line 13, 14). Each epoch starts with start state s_0 . To execute the learned policy, we set ϵ to 0 and no longer update the state value function.

To apply Q-learning to the crawl robot domain, we model an action as moving the arm. The immediate reward or penalty received is proportional to the distance traveled by performing the action, as measured by the ultrasonic sensor. As learning speed is of importance, we define a discrete state space with 9 states. These states are displayed in figure 6. By selecting the state space to be small, the learning algorithm has to explore only few states, which speeds up learning. We could have subdivided the arm movement plane into more states, with the drawback of longer learning time. More formally, the choices for the action-space A and state space S are:

$$A = \{ \text{move base arm up, move base arm down, move pivot arm far, move pivot arm closer} \}$$

$$\begin{aligned} S = & \{ \text{base up pivot far, base mid pivot far, base down pivot far,} \\ & \text{base up pivot mid, base mid pivot mid, base down pivot mid,} \\ & \text{base up pivot down, base mid pivot down, base down pivot down,} \} \end{aligned}$$

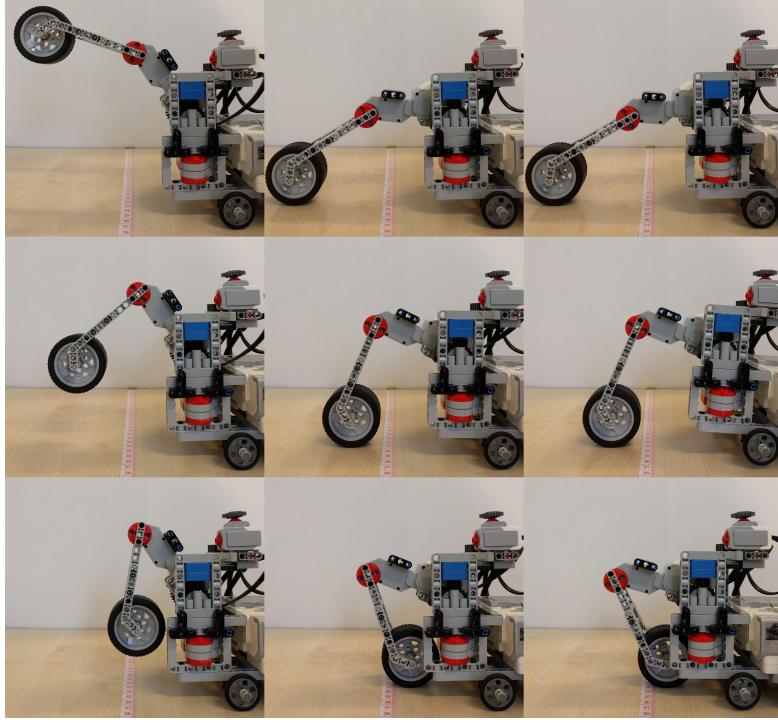


Figure 6: Arm positions (states) of the crawl robot. From top to bottom: Pivot arm in positions far, mid, close. From left to right: Base arm in positions up, mid, down. The tilt of the robot base in the right column is caused by the arm pressing into the floor.

The initial state for each epoch is given by $s_0 = \text{base up pivot far}$, which is the furthest extension of the arm. For fast learning, we got good results with a learning rate $\alpha = 0.5$, discount rate $\gamma = 0.8$. The exploration factor ϵ is initially set to 0.5 and declines exponentially throughout training, allowing lots of exploration at the start of training, while gradually reducing exploration for later epochs. We chose 35 steps per epoch, as this amount allows each epoch to explore all states multiple times. Training is stopped once the motion appears to be executed with little error.

6.3 Results

The crawl robot was trained using tabular Q-learning with the parameter choices stated in the previous section. In this section we discuss the training process and the learned result. For training, the robot is placed on a linear track. At the back of the robot, a solid target is placed to act as a point for range measurement by the ultrasonic sensor. During training, the robot explores actions and moves back and forth on the linear track.

After 20 minutes of training, the learned motion appeared near perfect and training was stopped. This corresponds to 65 epochs of training, for a total of 2275 actions. The reward gathered in each epoch is plotted in figure 7a. We can see that the reward gathered in each epoch has a clear upward trend, indicating that the robot got progressively better at the task. The exploration factor is exponentially declining, as shown in figure 7b. This leads to less exploration in later epochs. The result of this is visible in the reward plot, as in later epochs the results get more consistent.

The robot is not fully autonomous during training. It requires human intervention in three cases:

- The ultrasonic sensor has a maximum range of 250cm. If the robot moves further away from its range measuring target, it can no longer track rewards. Similarly, if it pushes into the range finding target, it can not move further into the direction of the target. Both cases disrupt assessing the rewards of actions. To handle these cases, the robot will interrupt training when it gets too close to either end of the spectrum. Training is resumed once the robot is manually placed on a valid part of the training track.

Algorithm 1: Q-learning with ϵ -greedy exploration

Input : learning rate $\alpha \in]0, 1]$, discount rate $\gamma \in [0, 1[$

1 initialize $Q(s, a)$ with 0 for all s, a

2 **foreach** epoch **do**

3 | return to initial state s_0

4 | **repeat**

5 | sample x uniform at random for $[0, 1]$

6 | **if** $x < \epsilon$ **then**

7 | | sample a at random

8 | **else**

9 | | select $a \leftarrow \arg \max_a(Q(s, a))$

10 | **end**

11 | take action a , observe r, s'

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

13 | $s \leftarrow s'$

14 **until** epoch end;

15 **end**

Output: state value function Q

- To properly assess the range to its range measuring target, the ultrasonic sensor needs to be orthogonal to the target. If the robot rotates during training, the angle to the range finding target can change. An angle significantly different from 90 degrees leads to skewed reward tracking, and in special cases the sensor can no longer find the target, which prohibits range measuring entirely. If the orientation of the robot changes significantly, it has to be manually rotated to its initial position.
- In rare cases, the motors of the arm have slightly too little torque to push against resistance caused by the hand hitting the floor. The motors become stuck. In these cases the robot needs to be slightly pushed to overcome the resistance and release the motors.

A recording of the full training and a presentation of the learned motion is accessible as a video online at [5]. The video shows the training process at 7 times normal speed. Manual interaction is shown as well.

6.4 Discussion

With the presented framework, the crawling robot is able to learn a forward crawling motion in 20 minutes. As the training process requires constant human interaction for resetting the robot, we think this amount training time is appropriate. Especially in educational settings it should be considered as an upper limit. Since the training is interactive, observing how the robot develops an increasingly stronger forwards drive during training is quite rewarding to the observer.

The learned motion is not smooth and appears abrupt, which results from the use of a discrete state space. Learning a crawling motion with this robot on a continuous state space has the potential of leading to a smoother, and possibly faster movement. A possible drawback is increased training time. Moving the arm on a continuous state space is a topic for future research.

The robot construction can be improved in the future. The current design has some mechanical issues, such as the strength of the motors in the arm and a tendency to slightly drift off the linear direction of movement. These problems could be addressed by re-constructing the robot and track of movement. A guide rail could constrict sideways the movement and gear conversions in the arm can be used to increase torque. The need for human supervision could be reduced by introducing a third motor to automatically move the robot back to its starting position after every epoch.

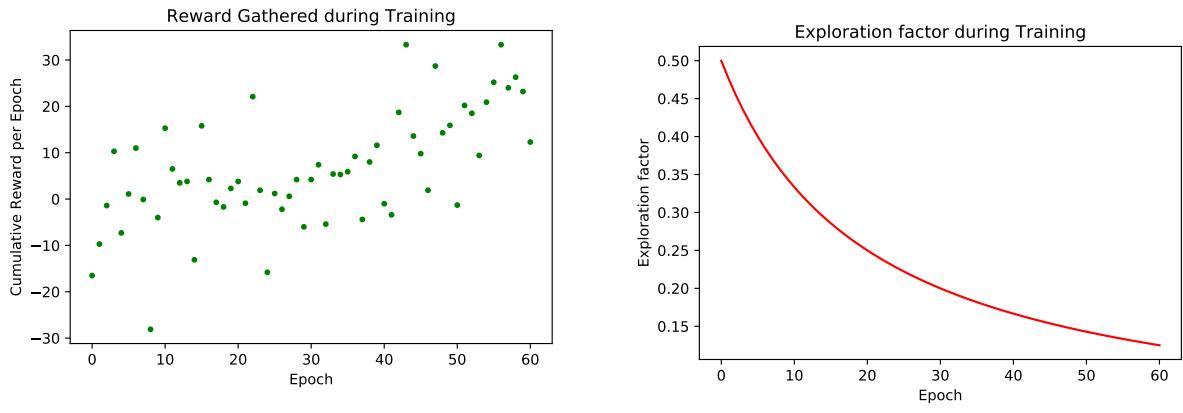


Figure 7: Q-learning of the Crawl Robot

In conclusion, we build a one-armed robot using LEGO parts, motors and sensors. We formulated a tabular q-learning problem with the goal of teaching the robot how to crawl. With some human interaction, the robot learned a forward crawling motion in 20 minutes.

7 Swing-Robot

7.1 Problem

what task solved

7.2 The Robot

description of the robot build

7.3 The Learning

implementation of learning, choices made in picking and implementing algorithm

7.4 Results

final results

7.5 Discussion

includes issues / future work

8 Future Work

Shared/independant future robots/ multi actor learning

9 Individual contributions

Table 2 shows the individual contributions to the report by each of the graded students.

Steffen	Asger
Development Setup	Introductions
Motor Accuracy	Time delays
Ultrasound Sensor	Colour Sensor
Calibration	Colour Detection
Crawl Robot	

Table 2: Sections of the report written by each of the students

References

- [1] Lego mindstorms ev3 hardware developer kit. *ev3dev.org*, 2013.
- [2] EV3 python bindings. *Github Repository*, 2018.
- [3] LEGO Mindstorms Rubics Cube Solver. *Github Repository*, 2018.
- [4] Ev3dev plattform. *ev3dev.org*, 2019.

- [5] S. Czolbe. Reinforcement learning with lego mindstorms, training montage. Available at <https://youtu.be/NUTv-oNWEYO>.
- [6] T. L. Group. EV3 Large Servo Motor, shop descipton.
- [7] Philippe Hurbain. LEGO® 9V Technic Motors compared characteristics. *philohome.com*, 2018.
- [8] V. Rousseau. Lego mindstorms: One-legged robot learning how to crawl with q-learning. Available at <https://youtu.be/F-kNXAFeffs>.
- [9] R. S. Sutton and A. G. Barto. *Introduction to reinforcement learning*, volume 135. MIT press Cambridge, 1998.
- [10] The LEGO Group. EV3 Ultrasonic Sensor - Product details. *LEGO.com*, 2018.
- [11] The LEGO Group. Products and Sets - LEGO® Power Functions. *LEGO.com*, 2018.
- [12] T. Timm. Reinforcement learning with lego mindstorms - crawling robot. Available at <https://youtu.be/bVbT9zkPIvs>.
- [13] C. J. C. H. Watkins and P. Dayan. Q-learning. *Machine Learning*, 8(3):279–292, May 1992.