Final Report

Using Reinforcement Learning to Push a Box in a Line

José Luís Gaytán
Claus Meschede
Jan Rudolph

July 10, 2016

1 Algorithm

In this project we implemented a Q-learning algorithm solving the task of pushing a box in a desired direction with the ePuck robot. This algorithm can be used for solving model-free control tasks by randomly sampling over all action values and is very easy to implement in its simple form. By taking the control policy into account during learning, we could also use the SARSA algorithm. Due to the simple formulation of our problem, the outcome of both algorithm should be the same. The Q-learning algorithm uses the following update formula:

$$Q(S,A) \leftarrow Q(S,A) + \alpha \left(R + \gamma \max_{A'} Q(S',A') - Q(S,A) \right)$$

In our problem formulation, the only reward R that is given during the learning process is when the robot reaches its desired state. Its value can be chosen arbitrarily and was set to $R = 100$ in this case. To deal with uncertain state transitions, we chose a learning rate of $\alpha = 0.5$ reducing the amount previously learned action values are overwritten. Furthermore, we set the discount factor γ to 0.8 to penalize the number of movements it takes to get to the goal state.
To ensure convergence to the optimal policy, we chose an epsilon-greedy control policy according to

$$p_{randomAction} = \left(\frac{1}{e} \right)^g$$

with e being the number of training episodes and g a factor for regulating the control policy. Each episode e is initialized by setting a random orientation for the robot and is terminated after reaching the goal state. When the robot reaches the goal state, we perform one additional action to ensure learning in the goal state as well. Otherwise, the robot would learn how to push the box in this direction, but it would not learn how to stay there. The factor g is used for regulating the amount of exploration and was finally set to 0.4 in our problem.

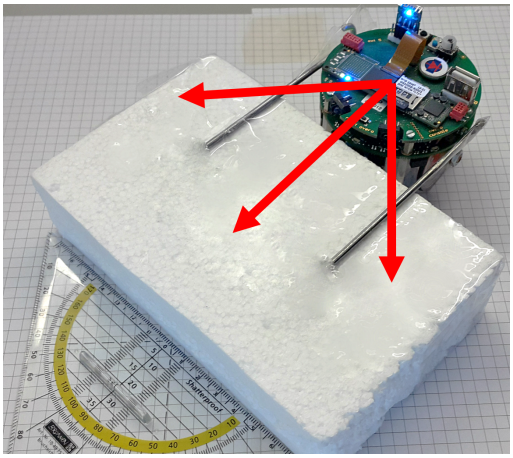
2 Challenges & Solutions

2.1 Movements

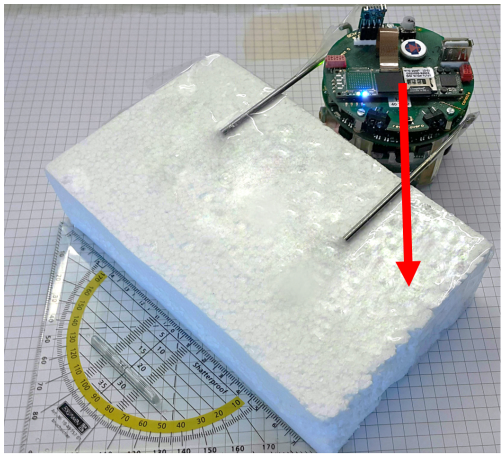
At the beginning, we had serious problems with making the robot move in our framework. We had to implement 2 movements: turning in-place for a specified angle and driving forward for a specified distance. However, due to the design of the serial console we used to control the robot, we could only set the speed of the motors, instead of being able to set a number of steps the motors should move. The speeds would also not get set at the same time, and there were sporadic glitches that were a result of bugs in the firmware where the speed for one of the wheels was reversed when setting it. We tried inserting waiting periods during the communication, which improved the situation, but the glitches still occurred so we had to manually remove those training steps, which was very tedious.

It was also difficult to prevent the robot from slipping, which could be mitigated by ramping the motor speed at the cost of more frequent glitches.

The situation finally was resolved when Martin Knopp introduced us to a new API and firmware (the RobotAPI) for the ePuck, that implemented turning and driving forward actions in the firmware itself.



(a) Calculate N_a equiangular actions relative to the box



(b) Select one and drive for distance d straight

Figure 1: Movements and action selection process for the robot.

2.2 Transitions

In order to optimize the learning procedure, we tried to maximize the transition probabilities from one state to another, i.e. driving forward should make the robot stay in the same state, driving to the right should have a high probability of reaching the adjacent state. This is important for achieving faster learning rates and avoiding reinforcement of non-optimal policies. The transition probabilities in our problem can be influenced by the following parameters:

- distance d: a longer distance increases the total rotation of the box per movement step, but also increases the probability for unwanted rotation when pushing the box straight
- number of actions N_a: a smaller number of actions makes learning faster, but a having more actions to choose from should enable a trained robot to reach its target in fewer steps.
- action angle α: The bigger the angle, the more the box rotates. An angle too small will cause issues with the box not turning but being pushed. If the angle gets too high, the robot will drive away from the box.
- number of states N_s: bigger state-space increases the precision in pushing the box to the correct direction, but also slows down learning.

Figure 2 shows the average transition probabilities that have been computed by combining all movements made in 150 episodes using the parameters in table 1.

Parameter	d in cm	N_a	α	N_s
Value	9.0	3	45°	10

Table 1: Movement parameters used in the box pushing task.

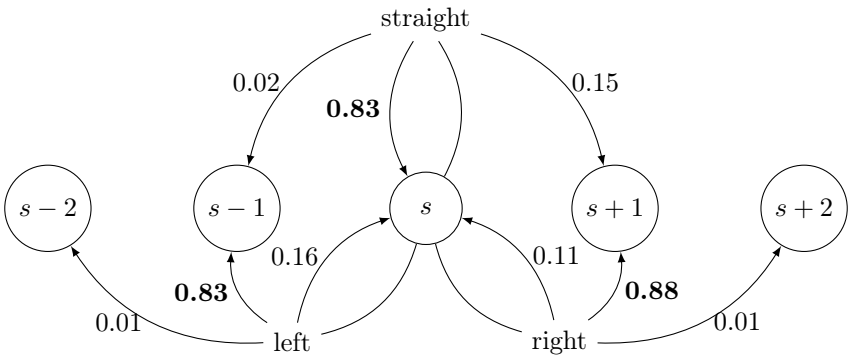


Figure 2: Transition Probabilities approximated by averaging over all state transitions for the training run shown in figure 5 (150 episodes).

2.3 Boxtracking

To make the ePuck push a box we need to know where the box is located in reference to the ePuck. As mentioned in the stage 2 report, the IR proximity sensors alone do not provide sufficient information about the location of the box center, they only give us information about the position of the box side. Our solution is to physically make sure that the robot is always positioned in the center of the box. We simply added two metal bars above the IR sensors that keep the robot in place (visible in figure 1). Computationally estimating the mass center of the box using visual information, for example, is a much harder task and not applicable in the time frame of this project.
Having solved that problem, we needed to estimate the angle that the ePuck has to turn to have the box right in front of it. In this project we used the infrared sensors (Figure 4a) giving us distance values from 0 to 4000 (higher values correspond to smaller distances). Mapping these 8 proximity values to a box position angle can be understood as a regression task.
A simple algorithm that can be used for such regression tasks is the k-Nearest-Neighbor algorithm. Based on labeled data, it computes similarity measures to compare the input vector with

the data and weights the known target values estimating the output. This algorithm is easy to implement and it doesn't have to be trained, but in order to get good results sufficient data is needed.
Our first idea was to get the labeled data from within the simulation, because we could easily compute the target values, then normalize it and compare the actual proximity measurements with the kNN algorithm. Unfortunately, this approach turned out to be too inaccurate, because of measurement differences between the simulation and the real robot. Therefore, we used the real robot to get our labeled data "by hand" (figure 1 and figure 3). Figure 4b shows the resulting proximity values when placing the box around the robot in 10°-steps.
For estimating the relative angle of the box, we then used inverse distance weights (Euclidean distance to input vector) to combine the target values of 3 nearest neighbors.
In order to further improve the precision of the turning movements, we also estimate the position of the box twice: After the first estimation, we turn the robot towards the box, after which we execute the second estimation, which then benefits from the higher number of sensors in the front.

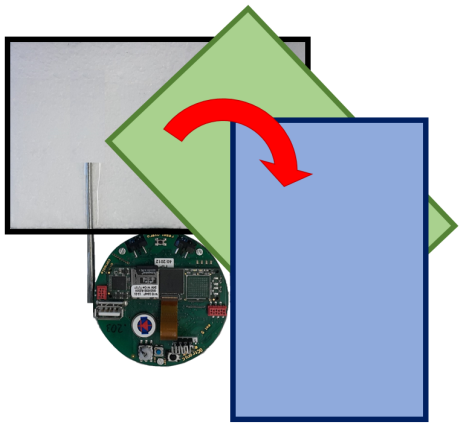
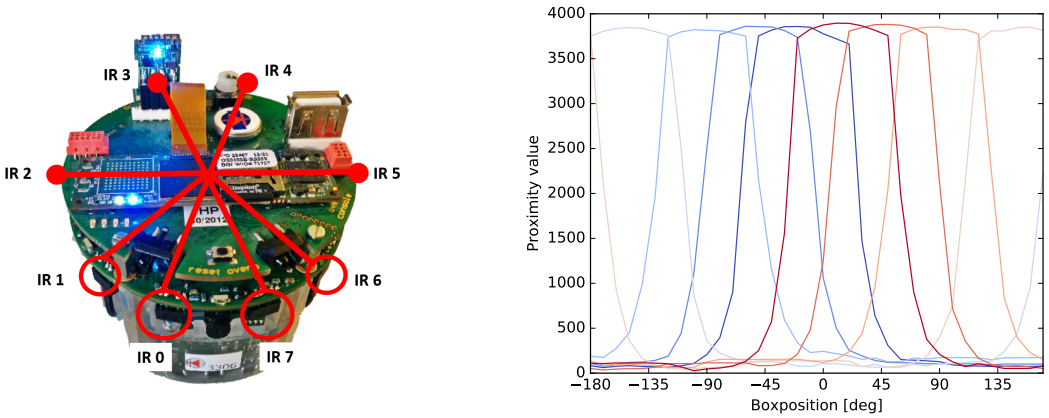


Figure 3: Calibration process

3 Results

Figure 5a and 5b show the learning progress and the resulting action values after 150 episodes, respectively. In less than 20 episodes the algorithm has learned to minimize the movements to the goal state fairly well. The horizontal line in figure 5a marks the maximum number of steps that it would take to get to the goal state in a perfect transition model. In our case, the control policy and the imperfect transition model can lead to more than 5 movements from time to time. Although the algorithm seems to learn the correct greedy policy very fast, the underlying action values need a lot longer to converge. Even after 150 episodes, the average action value is still increasing, albeit with a decreasing rate.
In figure 5b the action values after performing 150 episodes are shown. Reaching the goal state 0 is rewarded in this example. The algorithm has learned to push the box straight, if it's in the goal state. In the states 1 to 4 the highest action values are for the "right" actions, in states 6 to 9 the "left" actions, which is the expected and desired outcome. Furthermore, the effect of the discount factor can be seen by the degrading action values. State 5 faces exactly the opposite direction of our goal state, therefore the highest action value can be either going "left" or "right"



(a) The location of the proximity sensors on the ePuck. The movement direction of the robot is to the bottom. The sensors are more tightly spaced in the front.

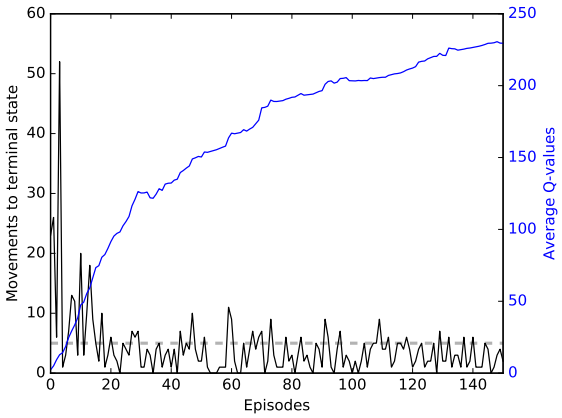
(b) Raw sensor values from the calibration process. In the middle of the plot is the front of the robot, where more sensors are available. Each color is one specific sensor

Figure 4: The ePucks proximity sensors and the calibration process for the box tracking algorithm.

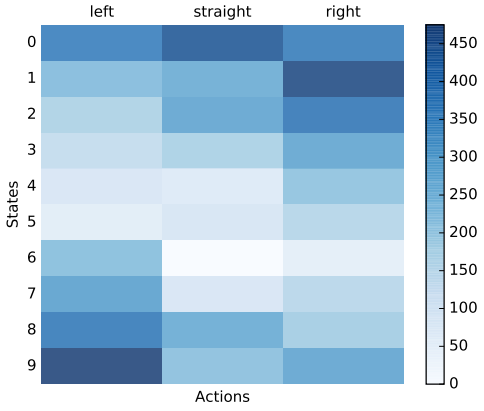
depending on the random initialization.
A common problem in reinforcement learning algorithms that can be seen here is the exploration of the state space. In the beginning of the learning process, the agent randomly transitions between the states until it reaches the terminal state and gets a reward. Even though the state space in this project is very small, it took over 50 movements to get there (figure 5a) in one episode. If the states that are visited during these movements haven't been updated yet, the algorithm won't learn anything until it reaches the goal state.

4 Conclusion

In retrospect, we learned a lot about the practical hurdles when applying RL algorithms to real-world problems. A difficult step is formalizing the problem such that it can be expressed as a MDP. It is very important to keep an eye on the size of the action and state space, to keep the problem solvable in a realistic timeframe, especially when working with real robots, as those have to be trained in realtime. A clever problem formulation can significantly reduce the size of the MDP while maintaining the important parts of the problem to be solved by the algorithm.
We also learned a lot about real world engineering and project planning. Even tasks that seem easy and straight-forward to solve can take a lot of time and effort until they work properly. High level tasks like turning or estimating object positions usually rely on many different components and conditions, that have to be considered to make them work together.
Simulations can be used to try out ideas and different settings, but they are often limited in their capability to correctly imitate the underlying physics. Therefore, the transferability of simulations to real world problems is often limited and should be considered when working on such problems.
The sparse design of the reward in our state space works well due to the limited size of our problem formulation. It can easily be seen that this gets a lot harder for bigger sized state



(a) Learning Curve. The gray line marks the maximum number of steps the ePuck would need with a perfect transition model.



(b) Q-values in episode 150

Figure 5: Resulting action values after 150 episodes with $d = 9cm$, $N_a = 3$, $\alpha = 45°$, $N_s = 10$, $\gamma = 0.8$ and $\alpha = 0.5$.

spaces, because the chances of getting a reward will be very low. Extending the state space in our problem as described in the project proposal, to push the box not only in a certain direction, but on a line would probably require a more sophisticated formulation of the reward function.