Navigating the Unknown: Reinforcement Learning and Thymio Robots

Mobile Robots

Anna Ricker(61287), Ariana Dias (53687)

1 Introduction

Reinforcement learning (RL) is a powerful machine learning technique that allows an agent to learn how to make decisions in an environment by maximizing a reward signal.

For this project, we trained a Thymio Robot to explore a maze and learn his way out using RL.

To train a robot using reinforcement learning, we needed to define the environment, actions, states, and rewards of the task, choose an appropriate learning algorithm and exploration strategy, and then run the learning process by letting the robot interact with the environment and update its policy based on the feedback it receives.

Section 2 will describe the environment, while Section 3 will shortly describe Reinforcement learning and the algorithm we chose. Following Section 4 will explain the implementation process. Finally, the results and the outlook is presented in section 5.

2 The Maze

The training will take place in a virtual environment. Afterwards, we will transfer the training result to the real-life robot that will then explore a maze built out of cardboard with the same layout as the virtual maze we used for the training. Our environment is a 5x5 maze with one exit, no entry, and 25 possible positions for the robot to be. The agent is a Thymio robot. Each position of the virtual maze will get a number between 0 and 25 as shown in Figure 1 so that we can easily identify the position of the robot and simplify the problem to be Discrete. The Thymio has the choice of 4 actions in each position: move North, move East, move South, and move West. As the start position, we will always place the robot at field 14 in the maze, as shown in Figure 1. The starting point is always going to be the same while training the robot.

We do not visualize the goal, as we can just measure the distance to the goal by comparing the robot's position and the goal's position. The goal for the robot is to reach position 10. The distance to the goal returned by our method is a number between 0 and 6 and defines the distance measured in fields between the goal and the robot's position.

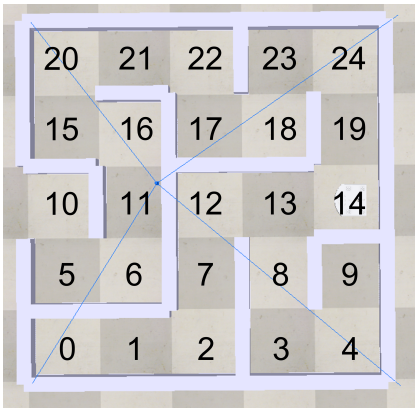

Figure 1: Setting up the Maze in Coppelia Sim

3 Approach

Reinforcement Learning (RL) is a type of machine learning where an agent learns to make decisions through interaction with an environment. The agent receives feedback in the form of rewards or penalties based on its actions, and its goal is to maximize the cumulative reward over time. RL is inspired by the way humans learn from trial and error.

Through the iterative process of receiving rewards based on its action, the agent updates its decision-making policy to make better choices in the future. RL is particularly useful in scenarios where explicit training data is limited, and the agent needs to learn from its own experiences.

For this project, we decided to work with Deep Q-Network (DQN) RL algorithm. Deep Q-Network (DQN) is a reinforcement learning algorithm that combines Q-learning with deep neural networks. It works by approximating the action-value function (Q-function) using a deep neural network called the Q-network. The Q-network takes the current state of the environment as input and predicts the expected future reward for each possible action.

During training, the agent interacts with the environment, stores its experiences in a replay buffer, and samples mini-batches of experiences to update the Q-network. The Q-network is trained to minimize the difference between the predicted and target Q-values, which are computed using the Bellman equation.

DQN is suitable for training a Thymio robot to solve a maze using RL due to its ability to handle complex maze environments, learn from experience, balance exploration and exploitation, and drive autonomous improvement.

4 Implementation

For the Implementation, we set everything up in an Anaconda environment. As programming language we used Python and we transferred the training results with the help of Jupyter Notebook and tdmclient to communicate with the robot. We also used Coppelia sim edu to set up our virtual environment. To communicate with the virtual environment we worked with coppeliasim_zmqremoteapi_client.

4.1 Set up the Environment

For the Implementation, we first started using the template for Reinforcement learning given by our teacher for setting up our base for the QLearning environment. This template consists of three classes: RL_test, ThymioControl, ThymioEnv.

4.1.1 RL_test class

RL_test is the main class of our code. We use this class to train our virtual robot or to execute the trained model in the simulation or the real world. As a library to train our model, we use stable_baselines3 as suggested by the teacher.

The RL_test class also includes the classes: ThymioControlM and ThymioControlC. In the first version, both of these classes include the following methods: forward, turn, check_valid_state, and out_of_bounds. The last two methods we didn't use for the rest of the project so we will not further look into those. More interesting however are the methods turn and forward. These methods simulate the actions the Robot can do in the first version of the project: turn and move forward. The ThymioControlC simulates those two possible movements used in the training while ThymioControlM just positions the Robot at the new position after the respective movement. The ThymioControlM class exists to lighten the training process. So, for the training process, we will always use ThymioControlM but for the actual virtual execution, we can use ThymioControlC. For our first try we used the same action as implemented in the template: turn and move forward.

4.1.2 ThymioControl

The ThymioControl is the superclass to the classes ThymioControlC and ThymioControlM which can be found in the RL_test class. This class defines the methods that can be used to control the virtual Thymio. In our project work, we barely changed anything in this class because it rather functions as an interface to simulate the Thymio movement.

4.1.3 ThymioEnv

The ThymioEnv of the template consists of five methods: __init__, step, _getObs, distance, and reset. The __init__ initializes the environment and the reset function is to reset this environment after each episode. _getObs returns the current position of the Tymio, while distance returns the distance to a given point. In our case, we use the distance function to calculate the distance to the goal.

The most important method is the step method. It processes every step of the training. The step method performs the actual step simulation and afterwards can return observations e.g. proximity sensor values. This method also returns the reward or punishment value for the training step and can even terminate the current episode. In the template, the reward was set like the following:

- For moving forward +1 point was added to the robot's reward for the current step.
- For moving left or right +0.5 point was added to the robot's reward for the current step.
- For the distance the robot has to the goal, Thymio got punished by reward-=dist**2.
- For reaching the goal the robot was rewarded with +100.0.
- If the proximity sensors detect any obstacle, the robot will be punished by reward-=prox.

4.1.4 Minor Adaptions to the template

At first, we tried to adapt the code only minimally to our conditions. First, we set up the maze to our needs. Therefore we added boxes to add the maze walls and changed each boxes (walls) special properties to collidable, measurable, and detectable. In the example the observation space was just a 4x4 field, so we changed the self.observation_space to a 5x5 grid as shown in the code snipped below.

```
self.observation\_space = gym.spaces.Box(low=-5, high=5, shape=(3,))
```

The first training result was not satisfying. The Thymio just took two steps forward as a result of the training. That we tried to fix this by changing the following:

- We left out the punishment for moving further away from the goal.
- We changed the reward for turning left to +0.5 so that the Thymio prefers one of the two turns.

Later we figured out that we defined the goal coordinates wrong, so the training technically was successful. Also, we found out way later that the previously mentioned check_valid_state method was not working properly, so it made one of the necessary steps the robot has to take to the goal always invalid. That made the goal technically unreachable. That is why we didn't use this method in the end.

4.2 Code Refactor

After this first unsuccessful minor adaption, we decided to do a great refactor of the code. Instead of using the methods forward and turn, we implemented the methods north, south, east, and west. This reduces the complexity of our problem a lot because we do not consider the angle of the robot in each step and simplify the problem to be just a discrete problem. The north method of the four methods is presented in Figure 2.

```
def north(self,distance):
    self.sim.setObjectOrientation(self.handles[self.names['robot'][0]],self.sim.handle_world,[0,0,1.5708])
    prox=self.getProximity()
    if (prox > 0):
        #print("wall in front at north")
        return False
    else:
        p1=self.sim.getObjectPose(self.handles[self.names['robot'][0]],self.sim.handle_world)
        p1[1] = (round(p1[1],2)) + distance
        self.sim.setObjectPose(self.handles[self.names['robot'][0]],self.sim.handle_world,p1)
        return True
```

Figure 2: New Implemented North Method from ThymioControlM

Each step the robot executes is exactly one field in the respective direction. Before each time, the robot executes the step, the robot just turns in the direction it wants to go and we check if a wall is in front of the robot. If there is a wall in front, the robot will not execute the step and gets a high punishment for choosing this step, in this case, e.g. the north method returns false to transfer the information about the wall to the step method that executes the step. Also, in this case, this training episode will be terminated, because the robot did an invalid step. The maze walls are detected by the proximity sensors of the robot. For the proximity, we decided that as soon as the value of the proximity sensor in the front center of the Thymio is higher than 0, there is a wall in front. To ensure that the walls are always recognized, we checked that the walls in our Coppelia Sim environment all have the same distance from the robot. As the robot is set exactly in the center of a field and each of Thymio's steps is exactly as big as the size of one field (0.5), we ensured that the maze walls were all set exactly in the middle of the field corners.

We also added a reward for each step the robot does, to encourage the robot to explore the maze and we added the punishment that increases the further Thymio gets away from the goal.

Still, also this training did not give us any mention-worthy results. Especially the termination of the robot each time it wants to go through a wall prevented the robot from exploring the maze enough. He didn't get far through the maze because, at each position in the maze, the probability of running into a wall is 50 percent.

4.3 Bugfixing and Reward Adaption

After a long period of playing around with the rewards and also the training parameters, we ended up with the following rewards that are inspired by the webpage we found solving a similar problem (see https://www.samyzaf.com/ML/rl/qmaze.html):

- Each step to another field costs -0.04.
- Reaching the goal gives a reward of 1
- An attempt to go through a wall will cost -0.75, but will not terminate the episode
- visiting a cell that was already visited before will be penalized by -0.25 points.
- to prevent endless loop, the episode will be terminated when a total reward of more than -12,5 points is reached.
- in each step a reward for the distance is added depending on the distance to the goal: reward = reward + (1 - (dist/6)**0.4)

Especially not ending the episode when the robot wants to go through a wall was a great success. This way the robot had the chance to actually discover the whole maze while still in its exploring phase. On the other hand, the threshold of -12.5 (25x0.5) reward helped to ensure that hopeless episodes are not unnecessarily drawn out. If this threshold is reached, we assumed the robot went in the wrong direction and already made too many errors from which it already learned enough, and should proceed to a new fresh episode.

The result finally led to a successful training and let us move to the next step of our task: transferring the training results to our real-life problem.

4.4 Transferring the Results to the Real World

To transfer the result to commands we can give the real robot, we first translated the resulting training .zip file to a simple text file containing the step sequence to the goal.

We then added a new file to our Repository: RealLifeImplementation.ipynb. This file we used to first connect the Thymio via tdmclient. Then we implemented the different steps (north, east, south, west) for the real-life environment, as can be seen for example for method north in figure 3. The problem here is, that in the real environment, we do not have a camera that can tell us in which direction the robot is turned to turn the Thymio in the respective direction for its next move. That is why we defined that as in the training at the beginning, we will position the robot pointing with his front to the west. For the following steps, we will remember the last step the robot took to know its orientation. Based on that, in each of the methods, we first communicate the robot to turn in the respective direction and afterwards move forward for a defined time. After the timer is expired, we communicate the robot to set the motor speed of both wheels to 0 again.

```
def turn90(direction, speed):
    set_var(motor_right_target=(direction * (-1)) * speed ,motor_left_target=(direction * 1) * speed)
    time.sleep(1.15)
    set_var(motor_right_target=0,motor_left_target=0)

def forward (speed):
    set_var(motor_right_target=speed,motor_left_target=speed)
    time.sleep(2.9)
    set_var(motor_right_target=0,motor_left_target=0)

def moveNorth(previousStep):
    dist = 200
    turnspeed = 200
    if previousStep == 1:
        turn90(-1, turnspeed)
    elif previousStep == 2:
        turn90(1, turnspeed)
        turn90(1, turnspeed)
    elif previousStep == 3:
        turn90(1, turnspeed)
    forward(dist)
```

Figure 3: New Implemented North Method from ThymioControlM

Technically the execution of this code lets the robot solve the real-life maze. But due to the low quality of the cardboard maze and the uneven fields, which do not all measure exactly

the same size, it is hard for the robot to reach the goal in the real maze. Adaptions for the future could be to use proximity sensors so that the robot does not bump into walls due to this inequality. But besides that, the robot perfectly solves the maze by running all the necessary steps to reach the goal.

5 Results

After tuning the parameters of the model, and the rewards/punishments, we were finally able to get a solution to the maze through it. The solution for the given maze is the following: $North \longrightarrow North \longrightarrow West \longrightarrow South \longrightarrow West \longrightarrow North \longrightarrow West \longrightarrow West \longrightarrow South \longrightarrow East \longrightarrow South \longrightarrow South \longrightarrow West \longrightarrow North.$

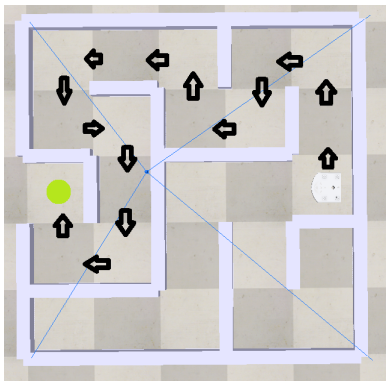

Figure 4: Maze in Coppelia Sim with arrows indicating the training result

The solution obtained represents the most efficient path for the Thymio robot to navigate through the maze. This allows us to determine that the training result delivers the optimal result for this maze configuration. Also, without any further proof, we assume that the successful solution obtained is not limited to the specific maze used in this project. The insights gained from tuning the parameters and rewards/punishments can be applied to a wide range of maze configurations and starting positions of the robot.

Overall, the results of obtaining a solution to the maze through parameter tuning and rewards/punishments highlight the success of our approach. They demonstrate the applicability of Reinforcement Learning in the area of implementing intelligent robots.

However, it must also be said that in this specific example, there are simpler and faster solutions to solve these kinds of problems, for example, using path-finding algorithms to determine the best path for the robot. It nevertheless provides a good example of the interplay between machine learning and the subsequent transmission of commands to the robot.

References

[1] Coppelia Robotics API Functions. https://www.coppeliarobotics.com/helpFiles/en/apiFunctions.htm.

[2] João Bimbo. RM_thymio. https://github.com/joaobimbo/RM_thymio.

[3] Neptune.ai. The Best Tools for Reinforcement Learning in Python. *Neptune.ai Blog*.

[4] Yves Piguet. tdmclient Project description. https://pypi.org/project/tdmclient/0.1.1/.

[5] Stable Baselines3. Deep Q-Network (DQN) - Stable Baselines3 Documentation. https://stable-baselines3.readthedocs.io/en/master/modules/dqn.html#example.

[6] Samy Zafrany. Qmaze - A Reinforcement Learning Example. https://www.samyzaf.com/ML/rl/qmaze.html.