universität
innsbruck

Reinforcement Learning - Applied to a Maze

Jakob Floß*

July 3, 2024

*jakob.floss@student.uibk.ac.at

Contents

1 Introduction to Reinforcement Learning

Machine Learning is a powerful tool. It can be used to automate tasks. Consider processing a hand-filled form for a bank transfer. With image recognition the letters of the boxes can be analyzed and the form can be digitized. The training could be done via supervised learning, as shown in the lecture. But what about more complex systems where providing such labeled learning sets are difficult or even impossible to produce?

Reinforcement learning is one option to tackle such kinds of problems. How? It lets an agent interact with an environment via action. Then the agent will get rewards based on his actions. From this he will learn in a way, maximizing the cumulative reward.

In the following I will follow a tutorial [1] found on Medium.

For the example of solving a maze the environment will be the maze and its logic implemented in a python class. The agent will be trying to solve this maze in an iterative way, learning from his previous attempts.

In order to explain the two entities involved in this case, let's take a closer look at their implementations.

2 Code Implementation

Generally the code is split into three (productive) parts:

- The Environment `maze.py`,
- The Agent `agent.py` and
- A learning or main routine `train.py`.

The core functionality of the maze is provided in the environment class, whereas the `train.py` file manages the training, i.e. the interaction between agent and environment.

2.1 Environment: Maze

Upon initialization the maze will either load the walls from a text file or create a default seven by seven maze. It will then administer the game. Via its public method `get_moves` it provides the agent with the possible moves he can take. The agent can in turn then take a provided move by passing it back as argument to the `move` method. Next the state and the current reward are given to the agent via the public `get_state_and_reward` method. In our simple case the reward will always have a value of $reward = -1$, regardless of the position. The most important attributes and methods are illustrated in figure (1).

2.2 Agent

The implementation of the agent is more interesting in the context of reinforcement learning. The agent can be initialized with 3 different parameters all constrained on the interval [0,1]:

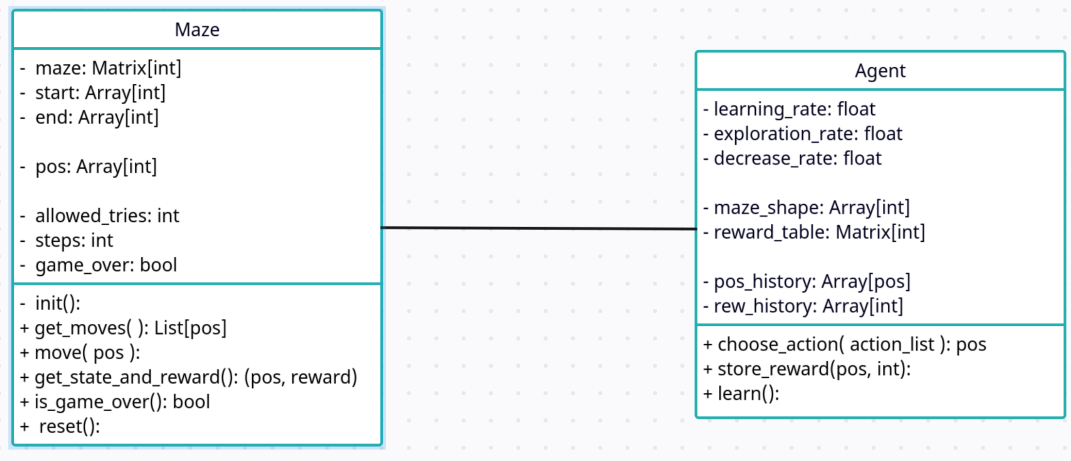

Figure 1: Class diagrams of the `Maze` and the `Agent`

1. The learning rate α,
2. The exploration rate ϵ and
3. The decrease rate γ.

At the beginning the agent creates a reward matrix (G-table). This matrix is used to map each position within the maze to an expected cumulative reward G_{pos}. This matrix is initialized with zero. During one game (episode) the agent will store the sequence of positions (i.e. his path) and the respective rewards associated with each position. It is only after one game that he will learn. Let's take a closer look at this.

2.2.1 Agent's `learn` method

When the agent finishes a game, either by exceeding the step limit or by finding the exit of the maze, he will try to learn from that game. Figure (2) illustrates the behavior. The agent starts at the end of his path. He then computes the cumulative reward along the path G_{path}. In this simple case, as the reward is always -1, the cumulative reward will measure the distance along the path towards the end (with a negative sign). As such the maximization of cumulative rewards means finding the shortest path. After computing the value G_{path} for a given position, the stored value in the G-table of that position will be adjusted:

$$G_{\text{pos}} = G_{\text{pos}} + \alpha(G_{\text{path}} - G_{\text{pos}}) \tag{1}$$

The agent will add a percentage (depending on the learning rate α) of the deviation between the observed cumulative reward (G_{path}) and the expected cumulative reward (G_{pos}) to the expected cumulative reward. In simpler words: *the agent adjusts its expected reward towards the observed reward.* Lastly the agent will decrease its learning rate by multiplying it with the decrease rate:

$$\alpha = \alpha \cdot \gamma \tag{2}$$

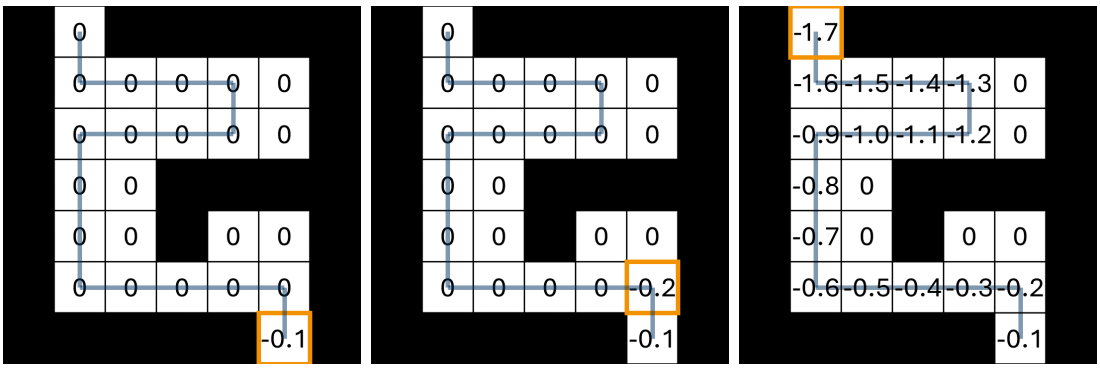

Figure 2: Illustration of the learning process after the first game. Before the first game the agent's reward matrix is initialized with zeros. The agent follows the path backwards, computes the cumulative reward along the path (G_{path}) and then learns according to formula (1) with a value of $\alpha = 0.1$.
Left: Last step: $reward = -1 \Rightarrow G_{\text{path}} = -1 \Rightarrow G_{\text{pos}} = -0.1$
Middle: Second last step: $reward = -1 \Rightarrow G_{\text{path}} = -2 \Rightarrow G_{\text{pos}} = -0.2$
Right: 17^{th} last step: $reward = -1 \Rightarrow G_{\text{path}} = -17 \Rightarrow G_{\text{pos}} = -1.7$

This ensures a decrease of α with time and thus more exploitation of the gained knowledge about the maze. We thus *hope* the agent converges on a (good) solution. After having understood how the agent learns his entries of the reward table, let's look at how he uses his knowledge in order to choose his next move.

2.2.2 Agent's `choose_action` method

The `choose_action` method can be explained rather quickly: Usually the agent will follow a "greedy" algorithm, meaning he chooses the action that maximises his (immediate) cumulative reward. Standing at some position he will ask the environment for the possible moves (actions). He will then evaluate the expected cumulative reward for each move. The move resulting in the highest cumulative reward will be chosen. Every once in a while however (tuned by the exploration rate ϵ) he will instead choose his move randomly. This modified approach is the called an *Epsilon-Greedy Algorithm* [2] also known from the *Multi-Armed Bandit Problem*.

2.3 The training flow

In order to understand the whole process of training better, lets shortly cover the training flow. A (minimal) version is shown in listing (1). First important parameters like α, ϵ, γ as well as `N_iters`, the number of episodes for learning, are set. The maze and the agent get initialized. In a loop the agent plays the game `N_iters` times. Each game is played as follows: The agent asks for the moves and chooses one of them. After the execution of the action, the agent stores the reward provided by the maze. This continues until the game is over either because the agent reached the end of the maze, or because he exceeded the maximum amount of steps he may take. After each game the agent invokes

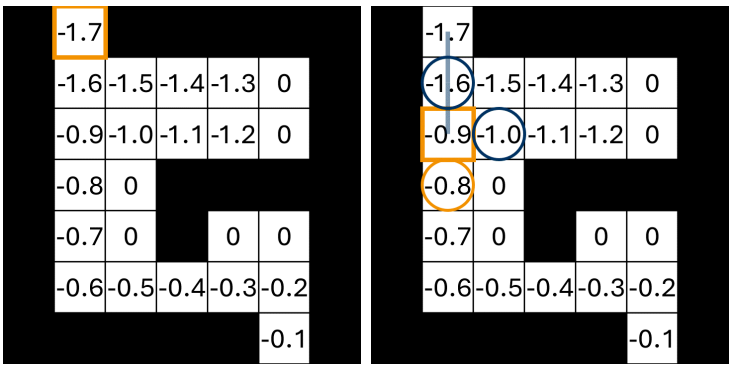

Figure 3: Illustration of how the agent chooses his action. Orange box: position of the agent, circles: possible moves to take, orange circle: selected move. The exploration part of the algorithm is not shown.
Left: Animation showing the agent and how he chooses his actions
Right: Chosen action for the third step (if animation doesn't play)

his `learn` method described in section (2.2.1). Before the next game the maze gets reset to its initiallization state.

```
def train():
    maze_name = 'big_maze'
    learning_rate = 0.15
    exploration_rate = 0.3
    decrease_rate = 0.99

    N_iters = 100
    m = maze.Maze(maze=maze_name)
    a = agent.Agent(learning_rate, exploration_rate, decrease_rate, maze_shape=m.shape)

    for i in range(N_iters):
        a.store_reward(*m.get_state_and_reward())

        while m.is_game_over() == False:
            action_list = m.get_moves()
            action = a.choose_action(action_list)
            m.move(action())
            a.store_reward(*m.get_state_and_reward())

        a.learn()
        m.reset()
```

Listing 1: The training routine for the agent. Describtion in section (2.3)

3 Learning Parameters

Now follows an investigation of how the agent performs. Figure (4) shows the agents solution after 500 iterations of learning. We can clearly see the impact of the decrease rate ϵ. If ϵ is a relatively low value (0.95) the exploration rate reduces so quickly that the agent is basically unable to find the exit of the maze. Raising the value to 0.99 we can see that the agent is able to solve the maze. The transition between solved and unsolved however is very sudden, indicating, that finding the solution can be somewhat a matter

of luck. If we raise ϵ even further, we can see that he starts to succeed however to reliably find the solution it took a little longer that in the previous case. With the discussion about the decrease rate γ we can also conclude the discussion about the exploitation rate ϵ. Both parameters control the how extensively the agent explores the environment. The exploration rate however is simply a scalar multiplicator whereas the decrease rate determines the speed with which the exploration rate decays exponentially and is thus a more important parameter. Maybe the names are misleading and could be chosen better.

Going further to the learning rate α. It also showed less effect on the overall performance than the decrease rate. Figure (5) shows the last plot of figure (4) but with a decreased learning rate of $\alpha = 0.05$. I suspect the learning rate to be more relevant in more complex systems. For such small systems as shown here the G-table entries are not overall good approximations of the distance towards the exit of the maze. As such the agent is not concerned with learning the correct values but once he has found a way he will continously choose this same path. Along this path the the entries of the G-table *will* converge and thus *are* good approximations, but this only happens *after* the agent has already found the way. In this sense I suspect the importance of α to grow when the options of different paths increas and the relative difference along those paths decreases. With this I want to conclude the discussion about the parameters and turn to a different topic with wich I delt more. This is the design of the two core algorithms: the learning and the selection of the action.

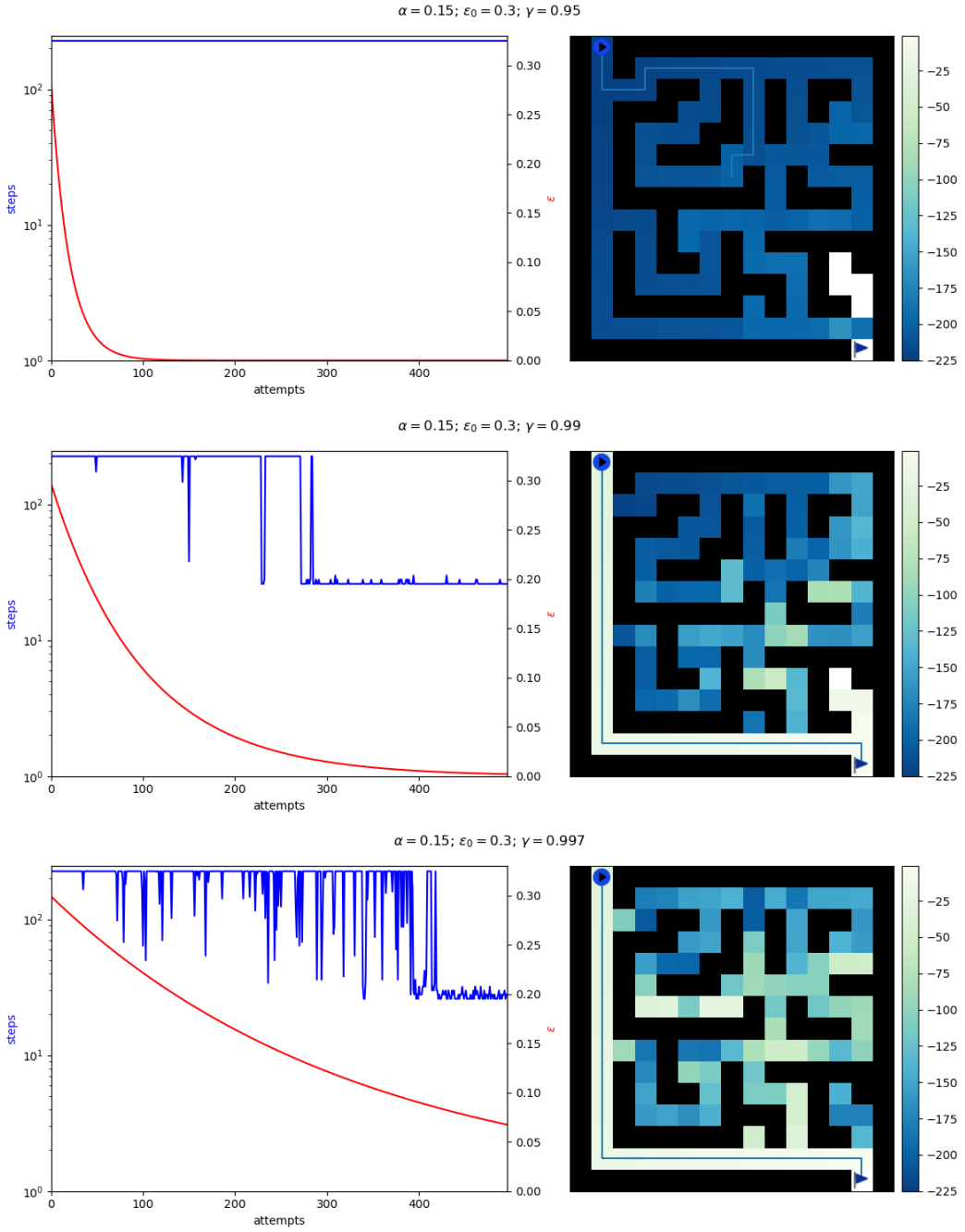

Figure 4: The training of an agent on a maze with 500 iterations. Each row corresponds to a differen decrease rate (ϵ) (0.95, 0.99 and 0.997 from top to bottom).
Left column: The number of steps it takes the agent to finish a game. It is limited to 255 attempts (blue). The exploration rate (ϵ) in red.
Right column: The maze (black walls) and the G-table (colored fields) of the agent after the last attempt. The path of the last attempt as blue line.

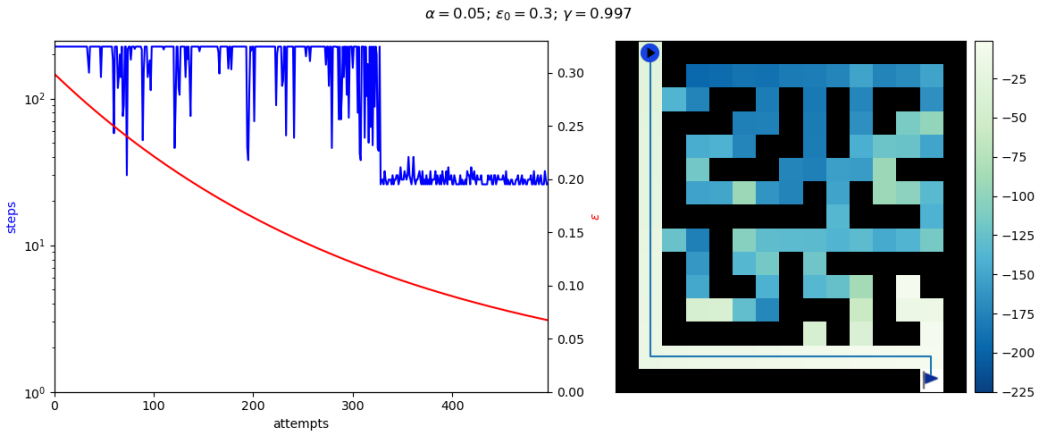

Figure 5: The training of an agent on a maze with 500 iterations. Training parameters are set to $\alpha = 0.05$, $\epsilon_0 = 0.3$ and $\gamma = 0.997$. In comparison with figure (4) bottom only the learning rate is reduced. This shows litte effect on the overall outcome.

4 Algorithmic Improvements

During different training runs I realized one thing. Often the agent does not walk very far at all. Sometimes he steps into a *hallway* but immediately steps back out other times simply runs back and forth. Figure (6) shows two examples of the agent's behavior.

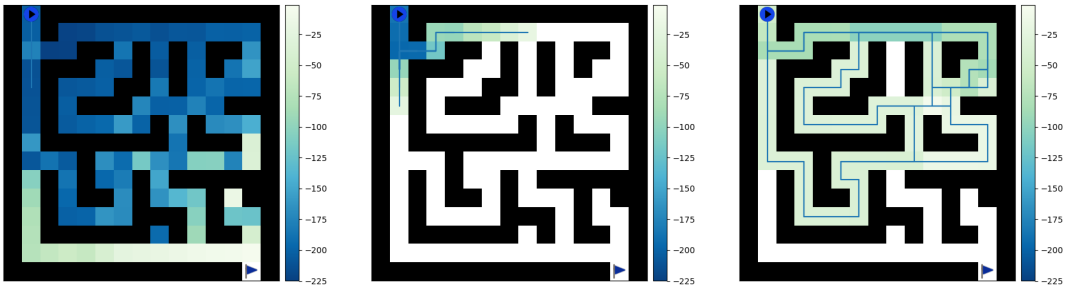

Figure 6: *Left:* A particular extreme example of the agent beeing stuck running only back and forth. This even happend after training.
Middle: A typical first attempt of the agent on a new maze. The exploration territory is quite small.
Right: After the new `choose_action` algorithm the agent exploration territory increased significantly. place.

4.1 Improved `choose_action` Algorithm

My approach to solving this problem was quite simple: after selecting an action the algorithm checks if this action results in stepping on the field he came from. If so, he

also checks if can choose from more than on action. If it is the only action available the agent is in a dead end. Thus, it is okay to go back and he will stick with the action. If however he has more than one option, he disregars the selected option and chooses again. This simple adaptation of the algorithm proved to be a drastic improvement in how quickly the agent explores the territory.

4.2 Improved Learning Algorithm

Another problem appears when the agent tries to learn from an unsuccessful attempt. Figur (7) shows such an occurence. To understand why this might be a problem, the same figure also shows an example of how the cumulative rewards might look like. We obsere that for some fields multiple cumulative rewards can exist. Instead of learning the more "conservative" option (i.e. the lower reward) the agent first learns the higher value and the later during the learning algorithm he revisits the same Fields and learns the lower option. Another option that showed slight improvements (especially without the improved `choose_action` algorithm) is to only select the lower value of the two and only learn from that. The last figure (8) shows the performance on a bigger maze. For both plots the improved `choose_action` version is use, only the lower one uses the ned learning variant. As can be seen the new learning varant improves the speed with which a solution is found. With only the new learning algorithm the agent is significantly slower, without any improvements I could not make the agent find any solution.

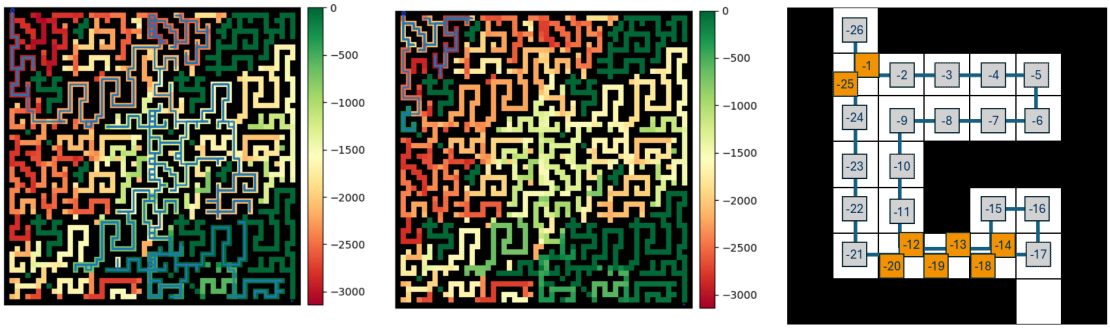

Figure 7: The agent tries to learn from an unsuccessful attempt.
Left: Reward matrix prior to the attempt.
Middle: Attempt and the reward matrix after the attemt.
When focusing on the upper left corner we can see, that the entries within the reward matrix increased, even though the agent did not find any connection between them and the exit of the maze.
Right: A example of how the cumulative rewards along an unsuccessfull path could look like. Most likely there are fields for wich multiple cumulative rewards exist. Originally the agent would learn from both these values, effectively learning the mean value.

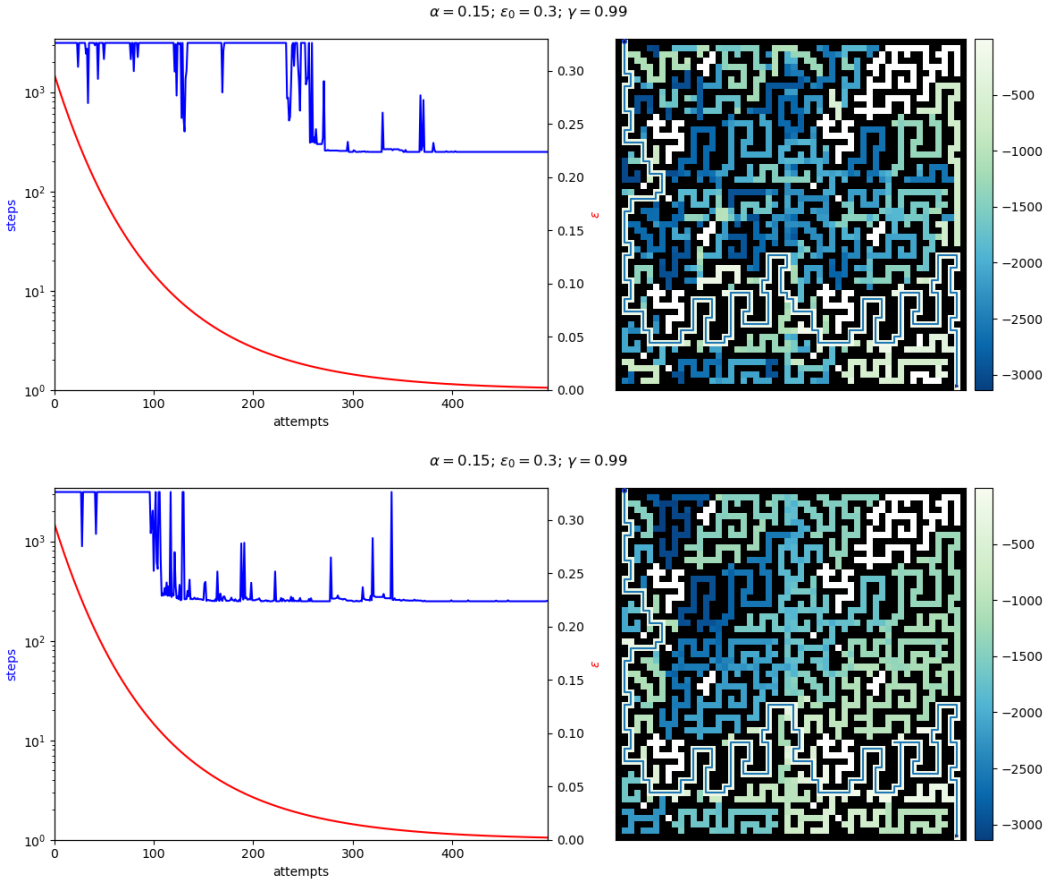

Figure 8: The agent solving a larger maze.
Top: The agent only uses the improved `choose_action` algorithm.
Bottom: The agent additionly employs the new learning algorithm. This results in better performance.

5 Conclusions and Outlook

Finally let's shortly discuss the learnings. I introduced a machine learning paradigm called reinforcement learning. With the example of a maze i constructed an algorithm which could solve small mazes. After investigation of the effect of the different parameters I concluded that the most important parameter is the decrease rate γ. Additionly I was able to find to adaptations of the algorithms presented in the tutorial which enabeled me to tackle much bigger mazes than previously.

What I haven't covered yet are any adaptations to the general problem. As witten in our case the maze always return a reward of -1. This seems limiting. An easy adaptation would be to introduce different Fields within the maze. One could imagine the along some paths the ground is more sandy and thus movement take more effort or a puddle has to be crossed. This should in principle be only a slight modification, this time on the side of the maze. There are many more problems that could and will be solved by reinforcement learning so let's see what is to come.

6 References

References

[1] Neha Desaraju. *Hands-On Introduction to Reinforcement Learning in Python*. July 2022. URL: https://towardsdatascience.com/hands-on-introduction-to-reinforcement-learning-in-python-da07f7aaca88.

[2] Avery Parkinson. *The Epsilon-Greedy Algorithm for Reinforcement Learning*. Dec. 2019. URL: https://medium.com/analytics-vidhya/the-epsilon-greedy-algorithm-for-reinforcement-learning-5fe6f96dc870.

Declaration

Hereby I declare, that this report is fully written by myselve and any necessary sources are referenced.

.....	.....
Jakob Floß	Date