Project 2
Bomb Away! An implementation of Q-learning and ϵ-greedy on-line Monte Carlo

Christoph Metzner (cmetzner, 000473394)
ECE 517 - Reinforcement Learning
University of Tennessee, Knoxville

November 2021

1 Introduction

This work focuses on the implementation and experimentation of two common reinforcement learning methods, Q-learning and Monte Carlo learning, in a simulation. Specifically, Q-learning and online Monte Carlo following a stochastic ϵ-greedy policy. The simulation is about a robot (i.e., agent) moving inside a 2-dimensional grid (i.e., environment) with the goal of removing a bomb outside grid. The bomb is placed somewhere inside the grid. The grid is surrounded by a *river*. The code was developed using `python3`.

- The robot (agent) is moving on a grid of size $d \times d$.
- The robot knows its own location and the location of the bomb on the grid.
- The grid is completely surrounded by a river.
- The robot moves only once cell per move (action).
- The robot can only push the bomb forward. That is, when the robot moves onto the cell the bomb is on, it pushes it one cell forward. The direction the bomb moves depends on the direction the robot comes from.
- The robot's starting position is random.
- The bomb's starting position is random.
- If the robot falls into the river, moves outside the grid, the robot gets placed on its previous position.
- The terminal state is reached if the robot was able to push the bomb outside the grid, i.e., into the river surrounding the grid.

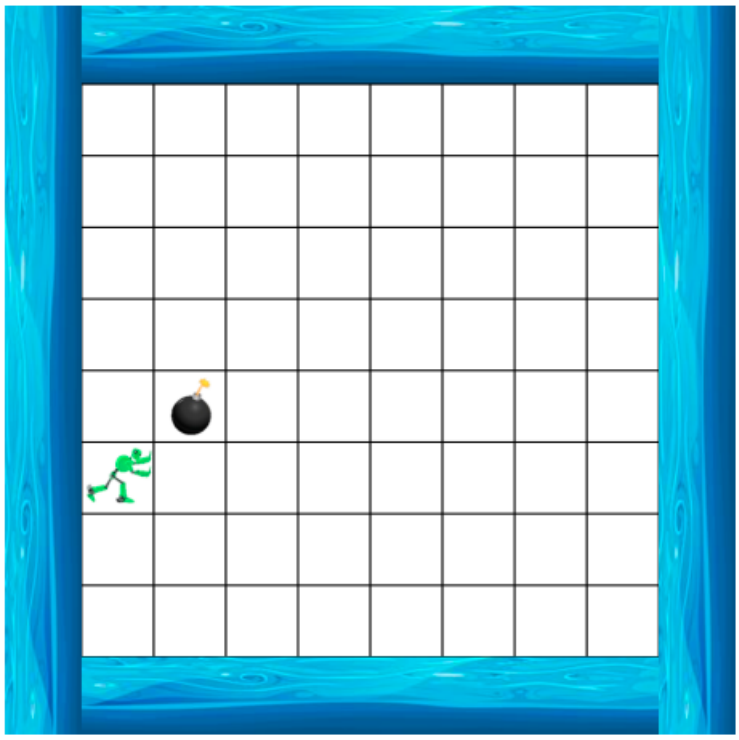


Figure 1: An example setup of the simulation.

1.1 Example Setup

An example setup can be seen in Figure 1. The starting state is set to the positions of the robot and bomb from that figure. Furthermore, the grid will have a dimension of 8×8. The value for the step-size α is 0.1 and for the exploration term ϵ is 0.1 for both learning methods. There will be no discount factor during the update of the action-values for both learning methods.

2 Description of the Markov Decision Process

The problem can be framed as a Markov Decision Process (MDP). The MDP = {S, A, R, P} can be described as following:

2.1 States - S:

The state space of the MDP depends on the dimension d of the squared 2-dimensional grid resulting in a state-space of size = $(d \times d \times d \times d)$. Specifically, on the positions of the robot and bomb, that are:

- y-position of the robot
- x-position of the robot
- y-position of the bomb
- x-position of the bomb

The indices of any state are as follows:

S(y-position of robot | x-position of robot | y-position of bomb | x-position of bomb)

2.2 Actions - A:

The robot can move in four directions on the grid, that are *north*, *south*, *east*, and *west*. The indices of each individual Q-values and policy for a specific state-action-pair follow the same pattern, i.e., north: 0 (index), south: 1, east: 2, and west: 3.

2.3 Rewards - R:

In this simulation, both learning methods are evaluated under two different reward structure regimes to see how important the selection of rewards are for the learning success.

2.3.1 Reward Structure - 1

In this reward structure the only reward for the robot is a negative for moving one cell on the grid.

- $R_{move} = -1$

2.3.2 Reward Structure - 2

This reward structure provides rewards for moving, pushing the bomb away from the center of the grid, and pushing the bomb into the river.

- $R_{move} = -1$
- $R_{bomb\_away\_center} = +1$
- $R_{bomb\_push\_river} = +10$

2.4 Probabilities - P:

The dynamics of the environment are rather unknown. The only thing that is known is that the probability of moving from one cell taking a specific action is one.

- $P_{move} = 1$

3 Data Structures and Code Design

In this section, the main design choices for the data structures of the matrices containing the action-values and policies, as well as, for the developed functions of the code are shown.

3.1 Data Structures

The first four indices of the data structures for the state-space, Q-values, and policies follow the same patterns as introduced in Section 2.1.

- State-space
  - numpy array
  - Dimension: (d, d, d, d)
- Action-values (i.e., Q-values)
  - numpy array
  - Dimension: (d, d, d, d, 4)
  - The number 4 represents the four actions the agent can take at any given state of the environment, i.e., each action has its own action-value.
- Policies
  - numpy array
  - Dimension: (d, d, d, d, 4)
  - The number 4 represents the four actions the agent can take at any given state of the environment, i.e., each action has its own probability of being taken.

3.2 Code Design

In the following section all major functions of the code are introduced and explained. For a more detailed representation of the code, the reader is referred to the appended code.

3.2.1 Function - `get_reward`

This function calculates the reward for a given move (action) and state and selected reward structure. As indicated in section 2.3, this implementation of this particular simulation includes two different reward structures. While the first reward structure is straight forward in its calculation for the reward per move, as it only ever gives -1, the second reward structure needed more thought. For the second reward structure, if-conditions were introduced to check once the bomb has been moved, that the robot indeed moved the bomb away from the center of the grid. First, the invisible center-line represented as an index of the grid was computed for the case of an even and uneven dimension of the grid. Second, the absolute distance of the old position is compared with the absolute distance of the new position to the center-line. If the new distance is larger, then the bomb was moved away from the center. The calculations are sensitive to the selected action.

3.2.2 Function - `do_Q_learning`

This function performs one episode of epsilon-greedy Q-learning. Every step of the episode, the functions `get_action`, `get_next_state`, `get_reward`, and `get_pi` are called in sequence. A while-loop performs the sampling until the agent transitioned to a terminal state or a maximum number of moves is reached. Q-learning is a off-policy learning method, in that, the Q-update is

greedy, whereas, the next state follows a policy. The action-values and the policy of the visited states are updated on the fly. The function returns, the total return G for that episode, the updated policy π, and the updated Q-values.

$$Q(S_t, A_t) = Q(S_t, A_t) + \alpha(R_t + \max_a Q(S_{t+1}, A) - Q(S_t, A_t)) \tag{1}$$

3.2.3 Function - `do_MC_learning`

This function performs one episode of epsilon-greedy on-line Monte Carlo learning. This algorithm samples a trajectory consisting of tuples=(S_t, A_t, R_{t+1}) by calling the functions `get_action`, `get_next_st` and `get_reward` with the help of a while-loop. This while-loop ends when the agent transitioned to a terminal state or a maximum number of moves is reached. After generating one trajectory, the action-values and policy of the visited states are updated starting at time-step $T - 1$. The following implementation of the on-line Monte Carlo learning method was implemented. The function returns, the total return G for that episode and the updated policy π.

$$Q(S_t, A_t) = Q(S_t, A_t) + \alpha(G - Q(S_t, A_t))$$

where,
G: expected return from terminal state to the visited state at T-t. (2)

3.2.4 Function - `execute_learning`

This function performs the learning for n episodes. Here the function, specifically calls either the Monte Carlo learning method or the Q-learning method by interpreting a user provided input. This functions stores the calculated returns per episode in an array and returns the array of all the calculated returns for plotting the learning progress.

3.2.5 Function - `plot_episode`

This function plots the episode given a starting state and a trained policy. First, a trajectory is sampled until the terminal state is reached. Second, each visited state since the starting state is plotted for the sampled trajectory from the first step.

3.2.6 Function - `get_pi`

This function updates the stochastic policy for both learning methods for a given state and the calculated updates of the action-values. This epsilon greedy approach sometimes causes to have two actions to have maximum value. Therefore, the probabilities have to be divided by the number of actions having such maximum value n_{max}. This code is used for both learning methods.

$$\pi(a_t, s_t) = \begin{cases} \frac{1-\epsilon}{n_{max}} & \text{if a = argmax Q(s, a),} \\ \frac{\epsilon}{|A|-n_{max}} & \text{otherwise.} \end{cases}$$

3.2.7 Function - `get_action`

This function selects an action given a state and the current policy (, i.e., probabilities of selecting any specific action) for that state. The action given a state is selected using a stochastic ϵ-greedy policy.

3.2.8 Function - `get_next_state`

This function performs all the movements of the robot and the bomb. For a given action, the new position of the robot is checked if the robot moves outside the grid. If the robot moves outside the grid, the robot is placed back onto the previous position of the grid, i.e., the previous state of the environment is returned as the successor state. Next, the function checks if the position of the robot is equal to the position of the bomb. If this is the case, the function checks if the bomb is moved outside the grid. If this is the case, then the episode is terminated and the current state is returned, otherwise the bomb is moved and a new state (s') is returned.

3.2.9 Function - `init_starting_state`

This function initializes the grid and the positions of the robot and the bomb. A while-loop was deliberately included to re-initialize the position of the bomb in case the first initialization causes the robot and bomb to start on the same position in the grid.

3.2.10 Function - `plot_total_returns`

This function plots the total return per episode for all episodes of training.

4 Results using Example Setup

The results are based on the example setup introduced in section 1.1. The results are presented in the following way: a) Q-values for the initial state are shown using the Q-learning method b) the trajectory the robot takes. c) tasks a) and b) are repeated for both reward structures. Furthermore, the two learning methods, Q-learning and Monte Carlo, are compared by examining the evolution of the total return per episode during training for both reward structures and the time in seconds it takes.

4.1 Q-Learning - Reward Structure 1

The agent was not able to converge to any optimal policy. Different total number of epochs were tried to see if more training was necessary. Figure 2. shows the Q-values of the initial starting state shown in the example setup for different epochs of training. This update strategy was not able to identify the best action for the selected starting state. Since the agent was unable to learn an optimal policy, the trajectory of the initial setup exceeds the number of plots that can be presented within the context of this report. Figure 5a) emphasizes the inability of convergence even more.

```
Epoch: 2000
Action-Values for initial Setup:
[-5.12861517 -5.11813327 -5.08374781 -5.083926  ]
Epoch: 4000
Action-Values for initial Setup:
[-10.53859708 -10.52883871 -10.56195756 -10.46124338]
Epoch: 6000
Action-Values for initial Setup:
[-15.88683439 -15.79488837 -15.75441978 -15.72688569]
Epoch: 8000
Action-Values for initial Setup:
[-21.2673485  -21.38669867 -21.36006601 -21.31570997]
Epoch: 10000
Action-Values for initial Setup:
[-26.69016108 -26.83063589 -26.66794969 -26.75848247]
Epoch: 12000
Action-Values for initial Setup:
[-32.40025827 -32.34605922 -32.32515078 -32.32255524]
Epoch: 14000
Action-Values for initial Setup:
[-37.88138577 -37.91694268 -37.95509299 -37.89289079]
Epoch: 16000
Action-Values for initial Setup:
[-43.26490181 -43.25924069 -43.20364312 -43.27496822]
Epoch: 18000
Action-Values for initial Setup:
[-48.72945653 -48.78294184 -48.71480181 -48.71612733]
Epoch: 20000
Action-Values for initial Setup:
[-54.26184963 -54.24654143 -54.23486606 -54.2131522 ]
```

Figure 2: Action-values for the initial setup state S(5, 0, 4, 1) for different epochs.

4.2 Q-Learning - Reward Structure 2

The agent was able to converge to an optimal ϵ-greedy policy within 20,000 epochs. Figure 3 presents the action-values for the initial state S(5, 0, 4, 1) as shown in Figure 1. As expected, the maximum action-value of 106.87 can be found for the action *east* with index 2. Figure 4. shows the trajectory of the agent for this learning. As we can see the agent does indeed follow an optimal policy.

```
Action-Values for initial Setup:
[ -0.4981     -0.4086524 106.8737973  -0.49     ]
```

Figure 3: Action-values for the initial setup state S(5, 0, 4, 1) after training.

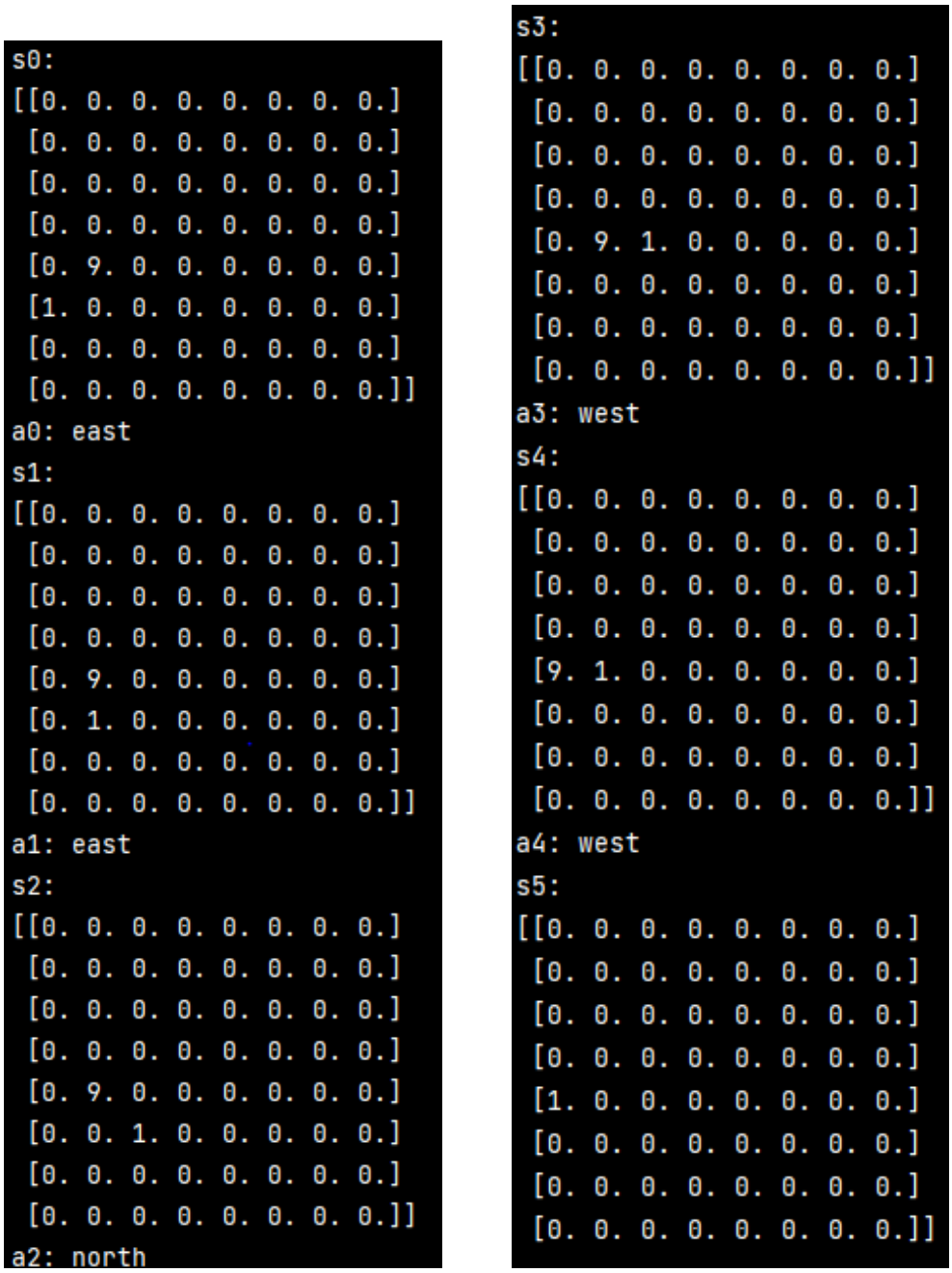
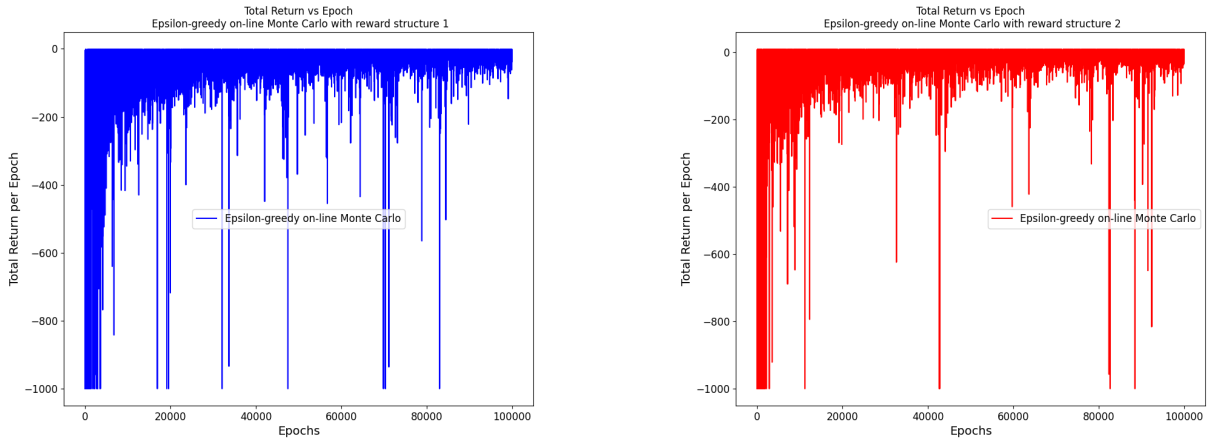


Figure 4: Trajectory of agent following optimal ϵ-greedy policy learned via Q-learning and reward structure 2 for setup example.

4.3 Total Returns and Training Time

In general, Q-learning performs better than the ϵ-greedy on-line Monte Carlo approach with the correctly selected reward structure, i.e., a reward structure that provides a positive signal to the agent when the bomb is pushed into the river. Therefore, both learning methods work better on reward structure 2, a structure that provides positive stimulus to the robot if the robot either pushed the bomb away from the center or into the river. Only using a negative reward for moving does not signalize the agent that moving the bomb away from the center or into the river is a good thing.

The evolution of the total returns per epoch for the Monte Carlo learning are presented in Figure 5 and for Q-learning in Figure 6. Monte Carlo approaches need longer training in epochs to converge to a optimal policy than Q-learning approaches. For the Monte-Carlo approach reward structure 1 exhibit higher variability in the total returns throughout the training. The agent using Q-learning to learn the environment with access to reward structure 1 is unable to learn appropriately. This specific approach shows by far the highest variability in total returns. Apparently, Q-learning cannot learn with only negative rewards. This drawback of Q-learning could stem from the way the action-values are updated and the actions are selected based on an ϵ-greedy policy. A rewards structure with only negative rewards may cause the agent to keep exploring by selecting the action with the maximum value, i.e., only negative rewards cause the actual optimal action to become smaller than the other actions. It is interesting to see that if the agent is provided positive stimulus as well, the learning process becomes significantly better. All approaches show some form of variability due to the ϵ-greedy nature of the selected policy.

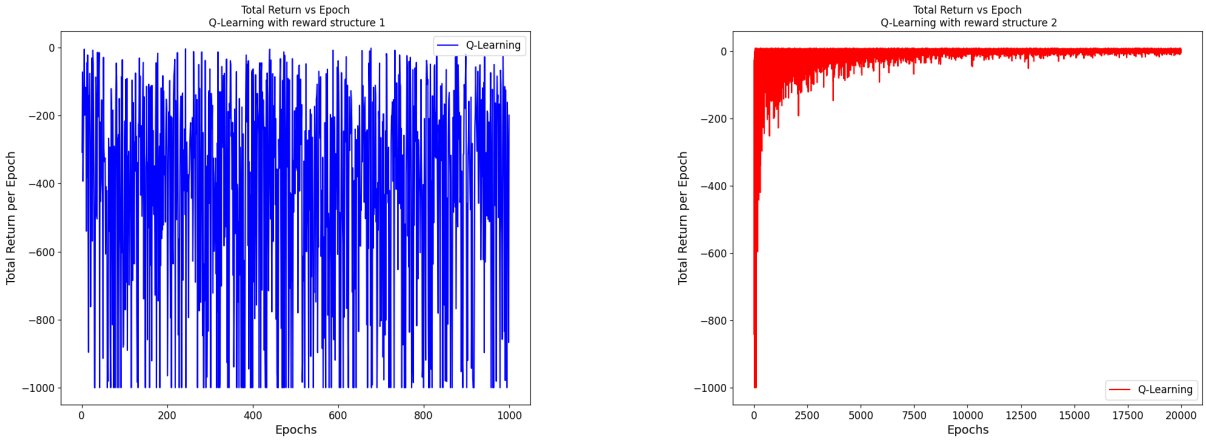


(a) Reward Structure: 1

(b) Reward Structure: 2

Figure 5: Evolution of the total returns for the agent learning via ϵ-greedy on-line Monte Carlo for both reward structures.

In general, Monte Carlo approaches takes longer than Q-learning since at first the complete trajectory of the episode is generated and then in a second step each visited state is updated. Also, general it takes more epochs for the Monte Carlo approaches to get converge to an optimal policy than Q-learning. Q-learning updates each visited state once actually visited while moving through the episode. The training times for all four learning method and reward structure combination are presented in Table 4.3. The time presented is the average training time per epoch for the first $n = 20,000$ epochs, for better comparability. The fastest learner is the agent using Q-learning guided by reward structure 2. This could stem from the design of the update rule. The q-values and policies are immediately updated once the state is visited. The Monte Carlo approaches take longer since in a first step the trajectory of the episode is generated, and in a second one, the Q-values are actually updated by looping through the trajectory again. Also further analysis of the training process shows, that Monte Carlo is slow at the beginning of the training due to the updates being couple with the going through the whole trajectory. Q-learning with reward structure 1, only negative reward for moving, took by far the longest time. This happened because the agent often exhausted the maximum number of moves allowed per episode.



(a) Reward Structure: 1

(b) Reward Structure: 2

Figure 6: Evolution of the total returns for the agent learning via Q-learning for both reward structures. The plot for reward structure 1 only shows the total returns for the first 1000 epochs to highlight the inability of convergence and variability in total returns.

Learning Method	Reward Structure	Time per Epoch in [s]
Monte Carlo	1	0.0025675
Monte Carlo	2	0.001287
Q-Learning	1	0.03135
Q-Learning	2	0.0009185

5 Conclusions

The key takeaway of these experiments is that the implemented reward structure has a significant impact on the convergence performance of the selected learning method. Furthermore, Q-learning, if designed correctly (reward structure), can significantly outperform Monte Carlo learning. Q-learning requires positive rewards as well to make more meaning of the environment.
From a coding perspective, better design of the functions and which task should be a function not only made the coding experience easier, but also provide much better readability of the code.