Course: Reinforcement Learning (SS2020)
Prof. Dr. Elmar Rueckert & Honghu Xue, M.Sc.
Assignment Report Nr. 2
Date: 22. Mai 2021

Team Nr. 1
Mevlude Tigre
Fazli Faruk Okumus
Contact: fazli.okumus@student.uni-luebeck.de
mevluede.tigre@student.uni-luebeck.de

Task I: Theoretical understanding on Monte-Carlo Methods and Temporal Difference Learning

1) The main difference between Dynamic Programming (DP) approaches and Monte-Carlo (MC)/ Temporal Difference Learning (TD) approaches is whether or not being model-based learning. Policy Iterations and Value Iterations are algorithms based on dynamic programming which requires knowledge of complete MDP where all environment variables are known. Here the goal of the agent was to learn a policy. This problem is called as Model-Based Learning. However, in real scenarios, we do not have knowledge of the environment and we have to learn about the environment by experimenting or interacting with the environment. To handle not having knowledge about the environment, we use model-free based learning techniques model like Monte-Carlo or Temporal Differences Learning.

Additionally, DP is effective for medium-sized problems (millions of states) and it suffers Bellman's curse of dimensionality in large problems. The number of states grows exponentially with the number of state variables. On the other hand, Monte-Carlo (MC) / Temporal Difference Learning (TD) approaches use sample rewards and sample transitions instead of reward function $\boldsymbol{R}$ and transition dynamics $\boldsymbol{P}$. So, it doesn't need to have advanced knowledge of MDP. It breaks the curse of dimensionality through sampling and the cost of backup is constant and independent of the number of states.

2) The reinforcement learning agent is doing two things generally,

- Taking action (i.e., using a policy) and;
- Learning which actions are good and which actions are bad in a given state. Using this learning the agent updates his Q values. The agent has to use a policy to update Q values also.

Behaviour Policy: Behaviour policy is the policy that the agent uses to determine its action (behaviour) in a given state.

Target Policy: Target policy is the policy that the agent uses for learning, i.e., to determine updated Q value.

If the target policy is same as behaviour policy, then it is called **on-policy** learning.

In on-policy reinforcement learning, the policy π_k is updated with data collected by π_k itself. We optimise the current policy π_k and use it to determine what spaces and actions to explore and sample next. That means we will try to improve the same policy that the agent is already using for action selection.

If the target policy is different from behaviour policy, then it is called is called **off-policy** learning.

In the classic off-policy,new policy π_k collects samples from $\pi 0, \pi 1, \ldots, \pi k$,and all of this data is used to train an updated new policy π_{k+1}. Off-policy learning allows the use of older samples (collected using the older policies) in the calculation. To update the policy, experiences are sampled from experiences/interactions that are collected from its own predecessor policies. This improves sample efficiency since we don't need to recollect samples whenever a policy is changed.

3) A stationary policy, π_t, is a policy that does not change over time, that is, $\pi_t = \pi \; \forall t \geq 0$. A **non-stationary policy** is a policy that is **not** stationary. More precisely, π_i may not be equal to π_j, for $i \neq j \geq 0$, where $\boldsymbol{i}$ and $\boldsymbol{j}$ are thus two different time steps.

In non-stationary situations, step size takes a constant value. On the other hand, in stationary situations, step size is used for weighting different experiences.

$$\alpha(\text{s}) = \frac{1}{\text{number of visits to state s}}$$

In this case, the Q and V values are the exact arithmetic average of the experiences.

4) Another condition is that balancing exploration/exploitation using epsilon (ϵ) and setting the value of how often you want to explore vs exploit. As an example if we select pure greedy method (epsilon = 0) then we are always selecting the highest q value among the all the q values for a specific state. This causes issue in exploration as we can get stuck easily at a local optima.

5) Although -greedy action selection is an effective and popular means of balancing exploration and exploitation in reinforcement learning, one drawback is that when it explores it chooses equally among all actions. This means that it is as likely to choose the worst-appearing action as it is to choose the next-to-best action. In tasks where the worst actions are very bad, we will need a solution. The solution is to vary the action probabilities as a graded function of estimated value. The greedy action is still given the highest selection probability, but all the others are ranked and weighted according to their value estimates. These are called *softmax* action selection rules. It chooses action a on the tth play with probability

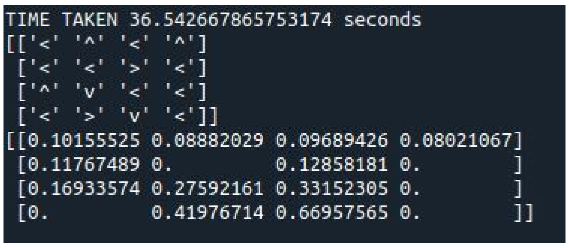
$$P_t(a) = \frac{\exp\left(q_t(a)/\tau\right)}{\sum_{i=1}^{n} \exp\left(q_t(i)/\tau\right)}$$

where τ is a positive parameter called the *temperature*. High temperatures cause the actions to be all (nearly) equiprobable. Low temperatures cause a greater difference in selection probability for actions that differ in their value estimates. In the limit as $\tau \to 0$, softmax action selection becomes the same as greedy action selection.

The biggest advantage over ϵ-greedy is that information about likely value of the other actions can also be taken into consideration. If there are 4 actions available to an agent, in e-greedy the 3 actions estimated to be non-optimal are all considered equally, but in Boltzmann exploration they are weighed by their relative value. This way the agent can ignore actions which it estimates to be largely sub-optimal and give more attention to potentially promising, but not necessarily ideal actions.

Task II: Programming part on MC and on-policy and off-policy one-step TD methods

1) Results of the On-policy first-visit MC control algorithm in MC_frozen_lake.py is shown below which are provided (Actions and Heatmap images respectively) by script itself



```
TIME TAKEN 36.542667865753174 seconds
[['<' '^' '<' '^']
 ['<' '<' '>' '<']
 ['^' 'v' '<' '<']
 ['<' '>' 'v' '<']]
[[0.10155525 0.08882029 0.09689426 0.08021067]
 [0.11767489 0.         0.12858181 0.        ]
 [0.16933574 0.27592161 0.33152305 0.        ]
 [0.         0.41976714 0.66957565 0.        ]]
```

Figure 1 Policy of the Monte-Carlo with $\epsilon = 0.4$

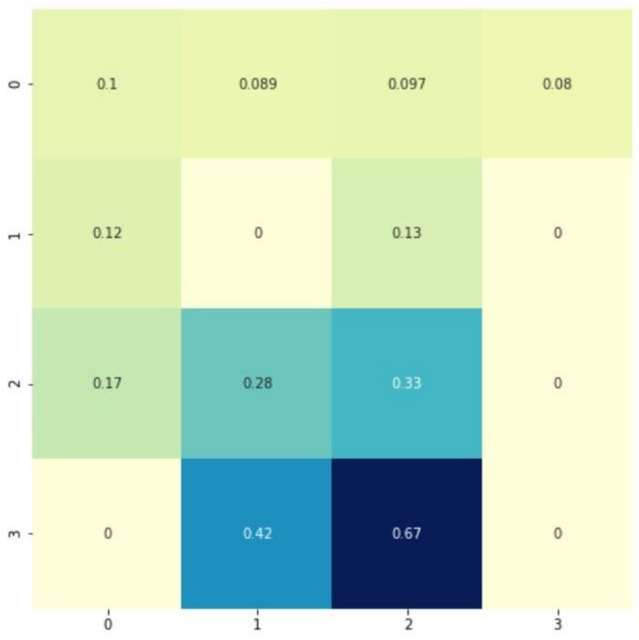


Figure 2 Heatmap of the Monte-Carlo with $\epsilon = 0.4$

2) Results of the SARSA algorithm in SARSA.py is shown below which is provided by script itself

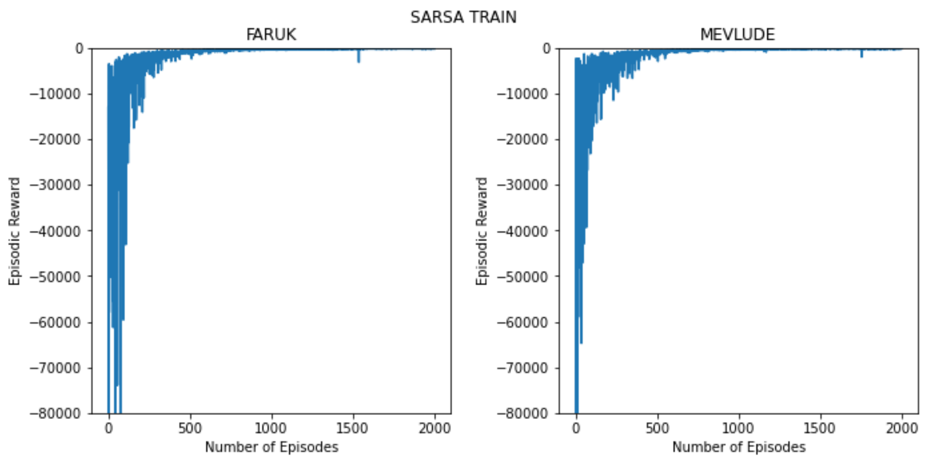


Figure 3 Results of the SARSA algorithm

3) Results of the Q-learning algorithm in Q_learning.py is shown below which is provided by script itself

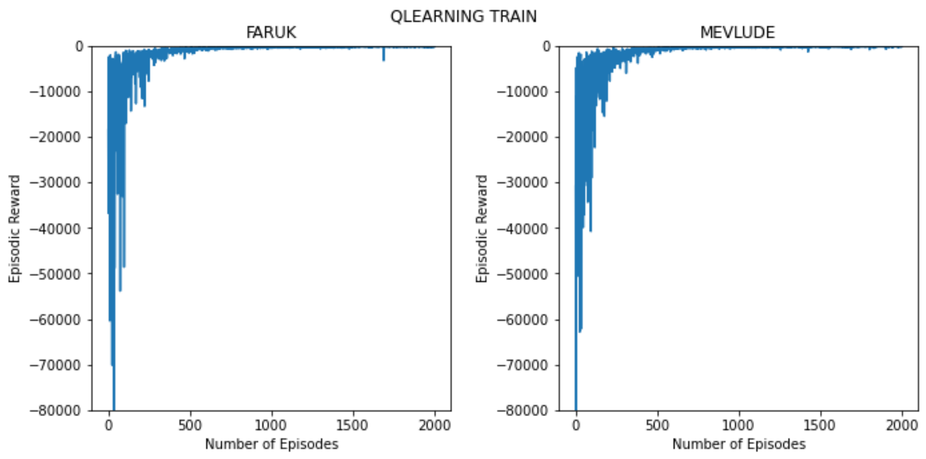
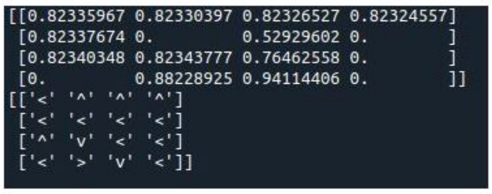


Figure 4 Results of the Q-learning algorithm

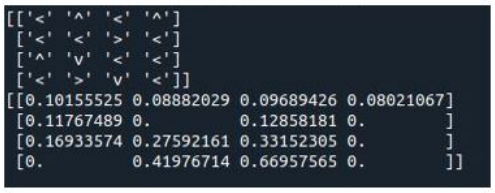
Task III: Going Deeper

1) Unlike PI/VI, we do not assume complete knowledge of the environment. In the Monte Carlo method, the agent finds the goal by trying whole possible options that converge to the goal. One of our primary goals for Monte Carlo methods is to estimate q_*. We want to estimate $q_\pi(s, a)$, the expected return when starting in state s, taking action a, and thereafter, following policy π for each episode. Here, after each episode, the observed returns are used for policy evaluation, and then the policy is improved at all the states visited in the episode. In the first states, because of the not enough visited state, there are some differences between the policy matrix of Monte-Carlo and PI/VI. However, as more returns are observed, the average converges to the optimal policy and optimal value function.



```
[[0.82335967 0.82330397 0.82326527 0.82324557]
 [0.82337674 0.         0.52929602 0.        ]
 [0.82340348 0.82343777 0.76462558 0.        ]
 [0.         0.88228925 0.94114406 0.        ]]
[['<' '^' '^' '^']
 ['<' '<' '<' '<']
 ['^' 'v' '<' '<']
 ['<' '>' 'v' '<']]
```

PI/VI POLICY



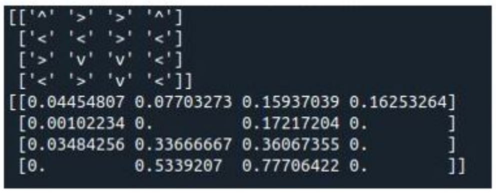
```
[['<' '^' '<' '^']
 ['<' '<' '>' '<']
 ['^' 'v' '<' '<']
 ['<' '>' 'v' '<']]
[[0.10155525 0.08882029 0.09689426 0.08021067]
 [0.11767489 0.         0.12858181 0.        ]
 [0.16933574 0.27592161 0.33152305 0.        ]
 [0.         0.41976714 0.66957565 0.        ]]
```

MC 0.4 EPSILON POLICY

Figure 5 Comparison of PI/VI and MC with $\epsilon = 0.4$

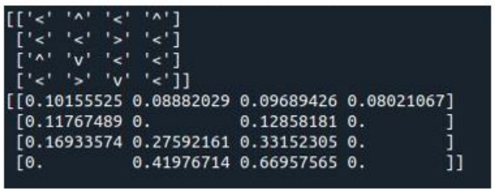
2) When epsilon is fixed as 0.05, then agent start doing, probably most of the times, exploitation because of the epsilon greedy algorithm and "1-epsilontterm approaches to 1. On the other hand, the MC algorithm with epsilon fixed to 0.4, do more exploration. This causes the agent to learn more about the environment it is in.

When we look at the differences between two MC algorithms with different epsilon values in terms of learned policy, MC algorithm with higher epsilon value have two different states (2,6) when it is compared to Policy or Value Iteration approaches based on dynamic programming which requires knowledge of complete MDP where all environment variables are known. On the other hand, MC algorithm with lower epsilon have more than two different states (0,1,2,6,8,10) because it is more greedy than MC with epsilon fixed as 0.4 and do less exploration, have less information about environment



```
[['^' '>' '>' '^']
 ['<' '<' '>' '<']
 ['>' 'v' 'v' '<']
 ['<' '>' 'v' '<']]
[[0.04454807 0.07703273 0.15937039 0.16253264]
 [0.00102234 0.         0.17217204 0.        ]
 [0.03484256 0.33666667 0.36067355 0.        ]
 [0.         0.5339207  0.77706422 0.        ]]
```

MC 0.05 EPSILON POLICY



```
[['<' '^' '<' '^']
 ['<' '<' '>' '<']
 ['^' 'v' '<' '<']
 ['<' '>' 'v' '<']]
[[0.10155525 0.08882029 0.09689426 0.08021067]
 [0.11767489 0.         0.12858181 0.        ]
 [0.16933574 0.27592161 0.33152305 0.        ]
 [0.         0.41976714 0.66957565 0.        ]]
```

MC 0.4 EPSILON POLICY

Figure 6 Comparison of MC with $\epsilon = 0.05$ and MC with $\epsilon = 0.4$

3) When alpha set to $\alpha = 1/N_{visit}$, it encouraged for weighing different experiences. Agent explore more and learn more about environment which is in and each episode takes more time than in which situation has fixed alpha.That concludes agent reaches near optimal policy because it had different experiences,returns than agent has constant alpha value. As more different returns are observed, the average converges to the optimal policy.

Secondly, time to converge to a near-optimal action value depends on the initial Q-value. Optimistic initial values help agents learn more, explore more and that takes more time than exploitation. Keeping the number of episodes constant and just changing initial Q values we can observe that the elapsed time for converging to optimal actions may change.

4) We can force the system to explore by fixing high initial value (for example $Q_0(a) = 5$) and if the reward is less than the initial value, then it will continue to explore (since $\max_a Q(a)$ is still within the actions not chosen). It is called *optimistic initial values.* The purpose of optimistic initial values is that encourages the agent to explore in the beginning. By doing this, agent will choose action never chosen before because it has bigger Q value.

For our case, when we look at the Q-table, since values of each state-action pair are vary between 0 and approximately -100,we have to choose bigger values than this interval. So, we can choose option B and E with values of 50 and 5 respectively and those options can be considered as valid, optimistic.