Autonomous Agents

Assignment 2: Single Agent Learning 5 October, 2012

By:
Paris Mavromoustakos
Georgios Methenitis
Patrick de Kok
Marios Tzakris

Introduction

In this assignment, we implement the learning scenario: the transition function is unknown to the agents, and so is the reward function. We have implemented several model-based algorithms, which give agents the ability to learn high-reward policies while ignoring the model.

Our code is based on the code handed in for the previous assignment, where we have solved the same problem in a planning setting. Moreover, we have explicitly used the 21 statespace environment representation which reduced our algorithms' runtime on the previous assignment.

Exercise 1

For this first exercise, we have implemented the Q-Learning algorithm with ϵ-greedy action selection, as described in chapter 6, section 5 of the Sutton and Barto book.

On each iteration of this algorithm, we chose an action a of state s, and observed the next state and reward we got from it. Then, we updated the Q-learning table according to the following update rule:

$$Q(s,a) \leftarrow Q(s,a) + \alpha \left[r + \gamma \max_{a'} Q(s',a') - Q(s,a) \right]$$

In this update rule, $Q(s,a)$ represents the value of the state-action pairs based on the previous iteration, while $Q(s',a')$ represents the observed state and possible actions, after action a is chosen. $Q(s,a)$ could be written as $Q(s_t,a_t)$ and $Q(s',a')$ as $Q(s_{t+1},a_{t+1})$.

α represents the algorithm's learning rate, whereas $\alpha = 0$ means that the Q-learning table will not be updated (the agent is not learning anything at all), while $\alpha = 1$ means that that the agent will base the updated value solely and completely on its obtained reward, discount factor and maximum value for the next state's action.

γ represents the discount factor, in the same way as described for the Dynamic Programming algorithms. $\max_{a'} Q(s',a')$ represents the action which is most likely to return the maximum reward among all the possible actions in state s'.

From Q, we derive a policy which is ϵ-greedy. We have tested this for different values of ϵ. Q has been initialized on different values as well. If not stated otherwise, $\epsilon = 0.1$ and $Q(s,a) = 15$, for any state s and action a.

The figures below indicate the performance of the predator over time, given different values for α , and different values for γ.

In this update rule, $Q(s,a)$ represents the value of the state-action pair on the previous iteration, while $Q(s',a')$ represents the observed state and possible actions, after action a is chosen. $Q(s,a)$

could be written as $Q(s_t, a_t)$ and $Q(s', a')$ as $Q(s_{t+1}, a_{t+1})$. α represents the algorithm's learning rate, whereas $\alpha = 0$ means that the value of the state-action pair will not be updated (the agent is not learning anything at all), while $\alpha = 1$ means that that the agent learns based only on its immediate past. γ represents the discount factor, in the same way as described in DP algorithms, and $max_{a'}$ represents the action which is most likely to return the maximum reward among all the possible actions in state s'. ϵ was given the value of 0.1 and the value of each state-action pair was initialized with 15.0 being assigned to all values. The figures below indicate the performance of the predator over time, given different values for α , and different values for γ.

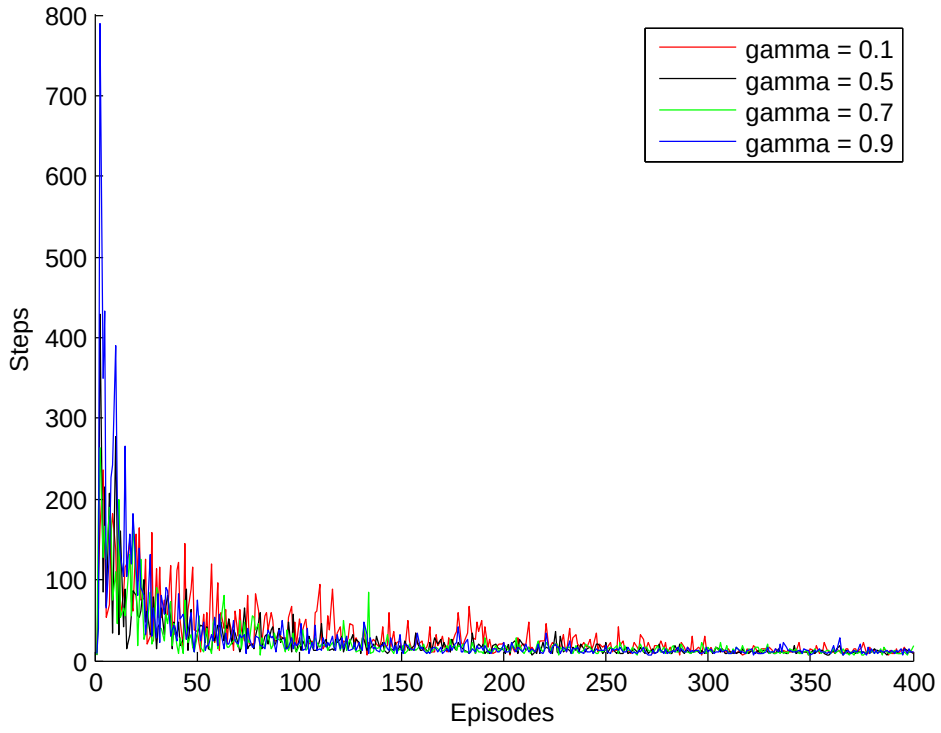

Figure 1: Q-learning results for different values of discount factor γ, having learning rate $\alpha = 0.1$.

Figures 1, 2, 3, and 4, depict clearly indicate how the predator learns faster while we increment the value of α. Figure 1 shows us that for $\alpha = 0.1$ the predator needs about 150 episodes to converge to the optimal (ϵ-greedy) policy. However, while we increase alpha, the predator will need less episodes for the state-action pair values to converge. Increasing the value of α means that we consider the most recent information we observe, more important. A value of $\alpha = 0$ would mean that the predator is not learning at all, while $\alpha = 1$ means that the predator will overwrite the current value of the state-action pair with the new value, as can be easily seen:

$$Q(s,a) \leftarrow Q(s,a) + 1\left[r + \gamma \max_{a'} Q(s',a') - Q(s,a)\right] \iff$$
$$Q(s,a) \leftarrow Q(s,a) - Q(s,a) + r + \gamma \max_{a'} Q(s',a') \iff$$
$$Q(s,a) \leftarrow r + \gamma \max_{a'} Q(s',a')$$

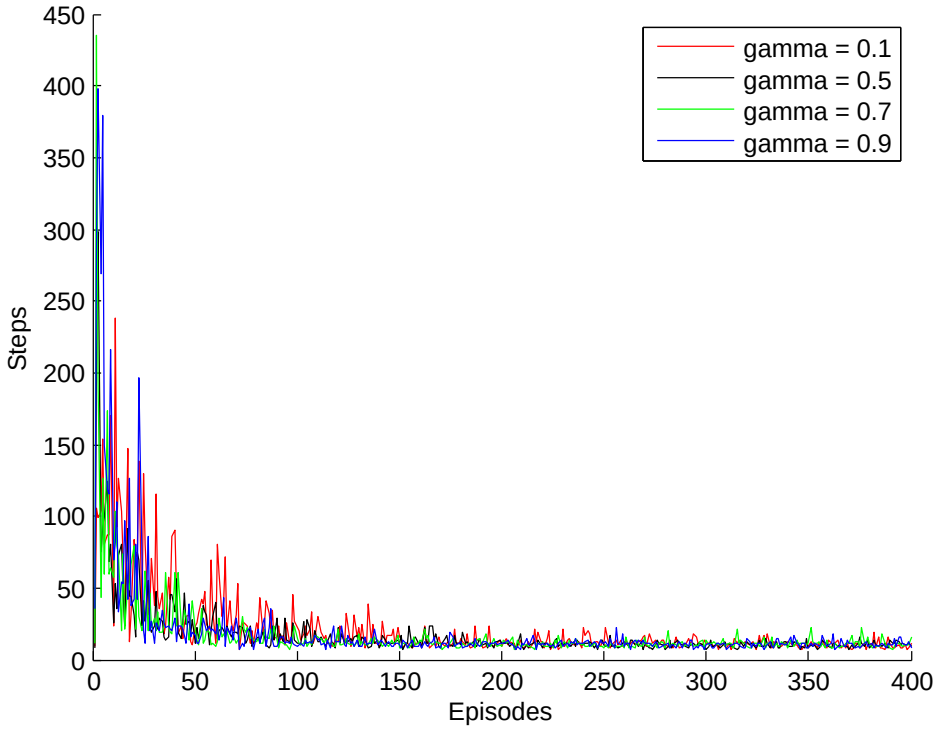

Figure 2: Q-learning results for different values of discount factor γ, having learning rate $\alpha = 0.2$.

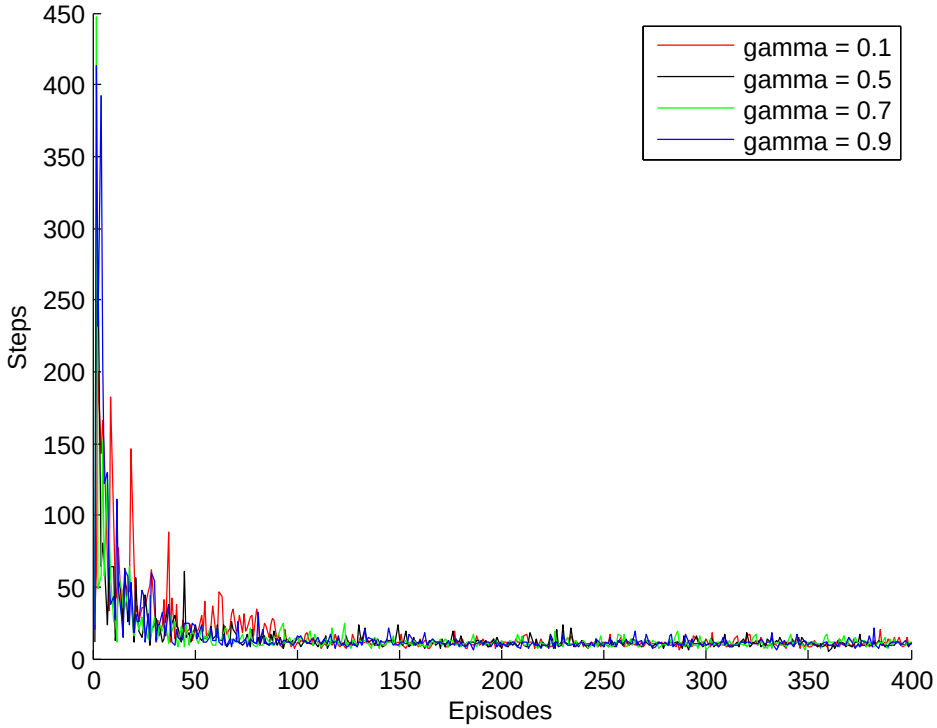

Figure 3: Q-learning results for different values of discount factor γ, having learning rate $\alpha = 0.3$.

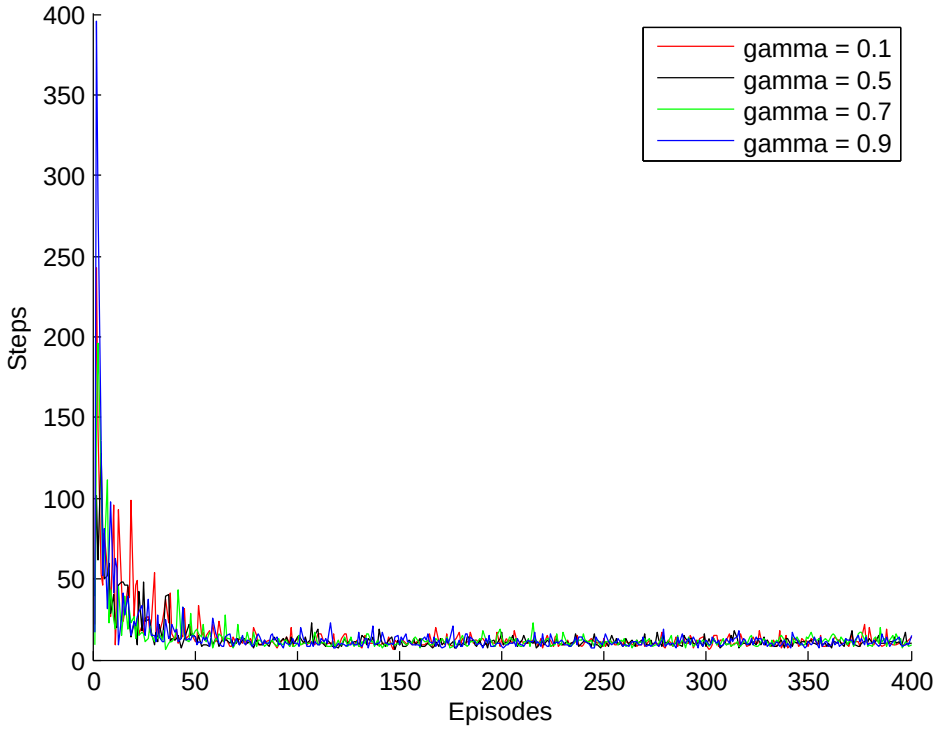

Figure 4: Q-learning results for different values of discount factor γ, having learning rate $\alpha = 0.5$.

Exercise 2

In this exercise we have implemented Q-learning with ϵ-greedy action decision for different values for ϵ , using different initial values for the state-action pairs each time. We have used 3 different values for the initialization of the values of the state-action pair: a pessimistic one (0), a realistic one (10) and an optimistic one (15), to see how the agent behaves for different values of ϵ in each case. On the pessimistic implementation, we expect the agent to converge to the optimal path really fast (given low values for ϵ). On the other hand, the optimistic implementation will cause more of an exlporatory behavior.

Figure 5, shows our tests with ϵ-greedy action selection for the optimistic, realistic and pessimistic initialization of the values of the state-action pairs. As shown, the pessimistic value helps the agent converge to the optimal path faster than the other initializations of the state-action values, while the optimistic value lets the agent explore a lot more before it converges. Using a pessimistic value to initialize the state-action values could lead the agent's policy into a local optimal, but not necessarily into a global optimal. In cases that there are three or more different rewards (in our example, we only have 0 and 10), an agent can only explore until it finds the local optimal policy to the state with the sub-maximal award. A higher value for the initialization would have lead this agent to explore the world better before its policy converge to a locally optimal policy.

We have chosen particular values for ϵ to test how, and what the predator will learn to do. For $\epsilon = 0$, the predator will always choose the action with the maximum value given the current state. Thus, the agent will converge to the optimal path really fast if the initial values of the state-action pairs are lower than any immediate reward, but it will become explorative if the state-action pair values are optimistically initialized. This corresponds with a (deterministic) greedy policy. If $\epsilon = 0.1$, the agent will try to exploit the action for which the Q-table has the maximum value with a probability of $1 - \epsilon = 0.9$. That means that our predator is trying to exploit more than it is trying to explore, and

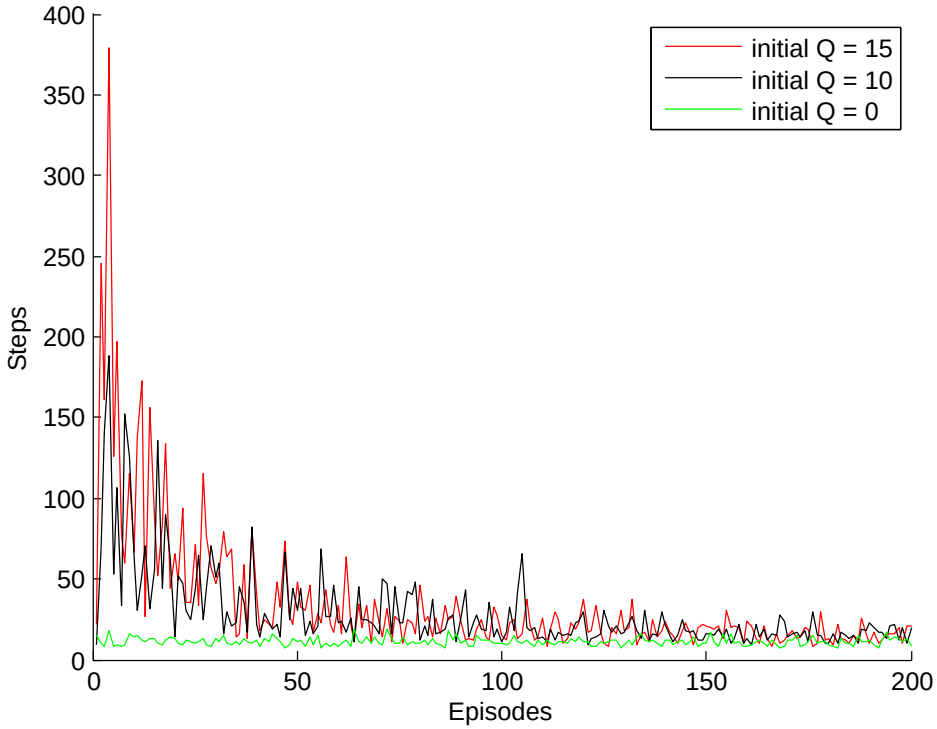

Figure 5: Comparing implementations for different values for $Q_{initial}$ with $\alpha = 0.1$, $\gamma = 0.7$.

there is a small probability that it will not choose the action it considers optimal.

We have used the value of 0.8 for ϵ as well, because in this case, each possible action for the predator has the same probability to be chosen. Because the predator has 5 possible actions, there will be $1 - \epsilon = 0.2$ probability for the optimal action to be chosen, and each of the other possible states will also have $\epsilon/4 = 0.2$ probability to be chosen. As a consequence, in the average case scenario our agent will be moving randomly. If there are less than 5 possible moves, he will be most likely be choosing one of the non-optimal moves.

Finally, we have also included $\epsilon = 0.99$ in our experimentation, for the reason that this value leaves 0.01 probability for the agent to choose the optimal action. So, what happens in this case is that the predator learns not to use the optimal action each time he has to take a decision. As we see in the graphs, the number of steps it takes him to find the prey is increased dramatically as time goes by.

It should be pointed out that for initial values of the state-action pairs 10 and 15, the graphs are quite similar, with the plot of $\epsilon = 0.99$ dominating the other values. We only included this graph to show how this particular value of ϵ affects the behavior of the agent.

Exercise 3

In this exercise we implemented SoftMax action selection policy, besides ϵ-greedy.

SoftMax differs from ϵ-greedy in one important detail: ϵ-greedy gives each non-optimal action a probability of ϵ divided by the number of non-optimal actions. SoftMax distributes weighted probability corresponding to the actions' values. This means, that in ϵ-greedy the worst possible action has the same probability to be chosen as the second-to-best action, while in SoftMax, the worst possible action will typically have the lowest probability to be chosen.

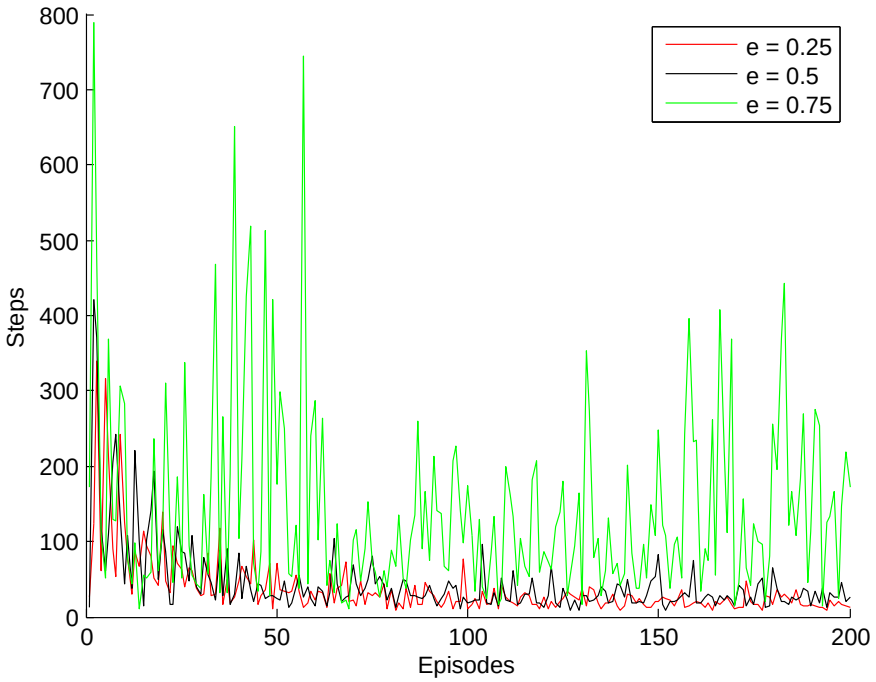

Figure 6: Comparing different values of ϵ with ϵ-greedy action selection. For $\epsilon = 0.75$, the agent will be deciding its actions almost randomly.

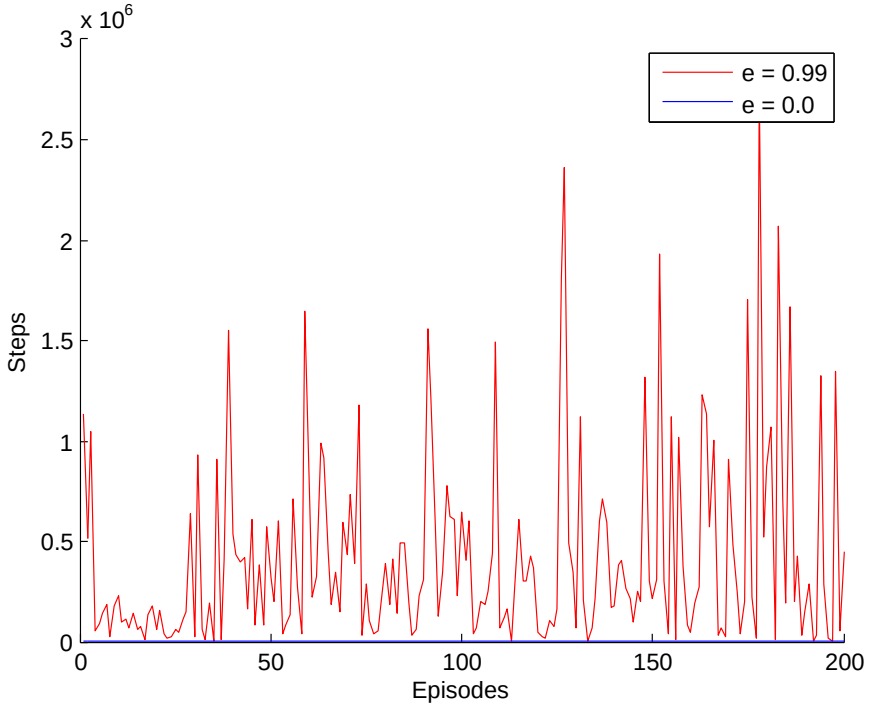

Figure 7: When the value of ϵ is close to 1, the agent tends to avoid the maximum expected reward states.

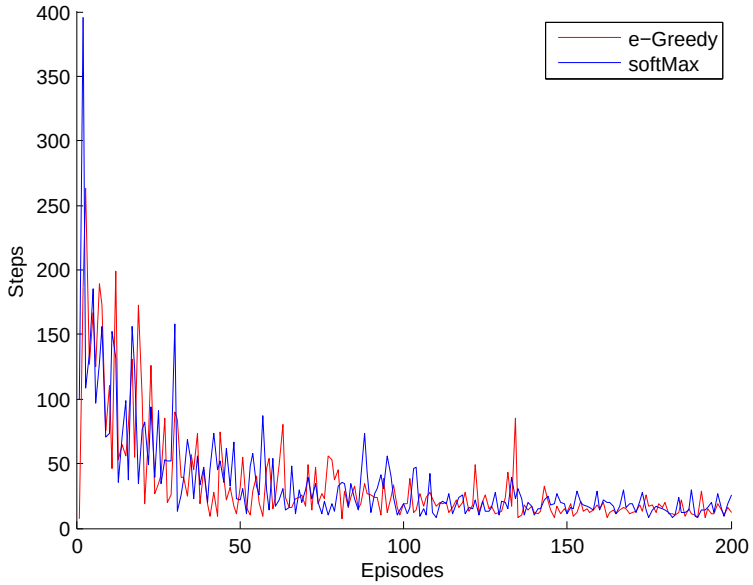

Figure 8: Comparing ϵ-greedy and SoftMax implementations in Q-learning for $\alpha = 0.1$, $\gamma = 0.7$, $\tau = 0.1$, $\epsilon = 0.1$ and $Q_{initial} = 15$.

SoftMax chooses each action with probability equal to $\frac{e^{Q_t(a)/\tau}}{\sum_{b=1}^{n} e^{Q_t(b)/\tau}}$ where τ is a positive variable called "temperature". If $\tau \to \infty$ the possible actions tend to be equally probable and thus corresponds to a random policy. However, if $\tau \to 0$, SoftMax tends to choose the next action greedily.

In our simulation, we found that ϵ-greedy and SoftMax action selection are very much alike, and do not show any important difference. In Figure 8, we have plotted the behavior of the agent with ϵ-greedy and SoftMax implementation.

Exercise 4

In the last exercise we went past the Q-learning algorithm and implemented the Sarsa, on-policy Monte Carlo and off-policy Monte Carlo algorithms.

SARSA

SARSA is an on-policy temporal difference control method. On-policy TD methods use a certain policy (for example, ϵ-greedy or softmax), which defines their future actions, and based on the reward they receive, they evaluate this policy.

Differences from Q-Learning

SARSA and Q-learning have a lot in common, but there is an important difference that discriminates the two algorithms: When agent is at state s and takes action a, Q-learning will update the Q table based on this action's reward. However, SARSA will further observe the next state-action pair (s', a') that derives from the next state s' produced by action a. So, obviously, SARSA needs two action selection steps, while Q-learning only needs one. For a greedy agent these two algorithms are identical. When exploration is happening, however, they differ significantly. Because Q-learning uses the best Q-value, it pays no attention to the actual policy being followed. It is better to learn a Q-function for

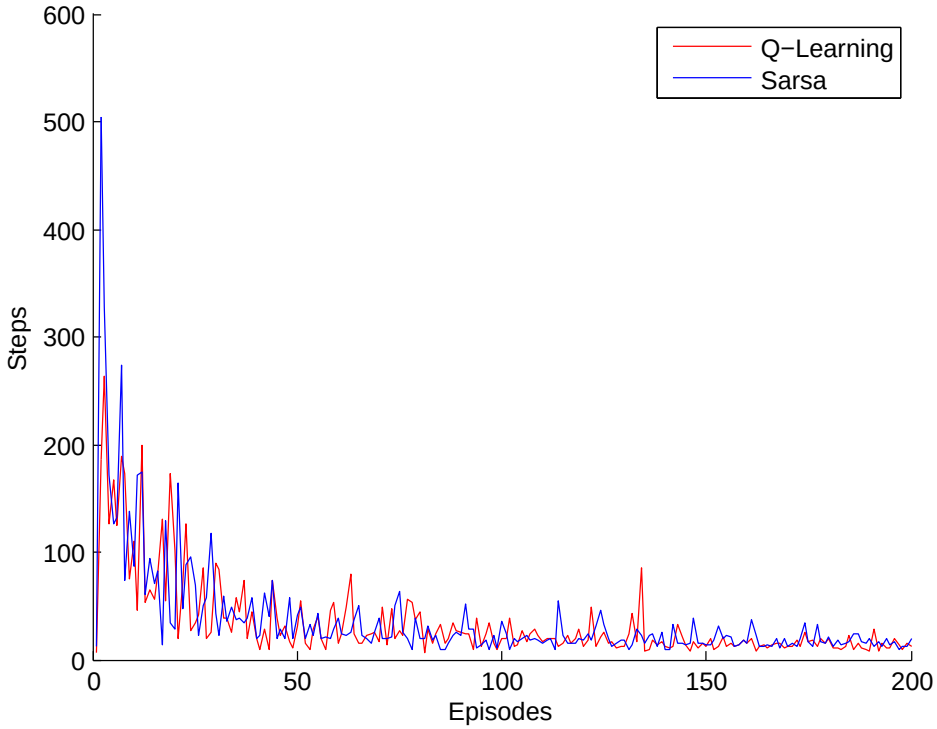

Figure 9: Comparing SARSA and Q-learning algorithms for $\alpha = 0.1$, $\gamma = 0.7$ and $\epsilon = 0.1$.

what will actually happen rather than what the agent would like to happen. Thus, Sarsa is considered an on-policy TD control method while Q-learning is considered to be off-policy.

Implementation

Figures 9, and 10, present the the performance of the two learning algorithms in our grid world problem, with different values of learning rate and based on the same ϵ-greedy policy selection. There are not obvious differences and this may happen because of the nature of our problem. We saw in class, problems which had "bad" state-action pairs with negative reward, and in these problems SARSA performed better than Q-Learning. We can conclude that SARSA algorithm has the same performance in this type of problem but can be better or worst than Q-Learning in others.

On-policy Monte Carlo

Monte Carlo control methods come in on-policy and off-policy variants as well. Where Q-learning needs only the current state-action pair and the observed next state and reward, Monte Carlo methods needs to know the *return* of a state-action pair. The return R_t of the tth state-action pair in the episode of T state-action pairs is computed by equation 3.3 of the Sutton and Barto book:

$$R_t = \sum_{k=0}^{T} \gamma^k r_{t+k+1}$$

This means we need to finish the whole episode before we can update Q. Therefore, Monte Carlo control methods are unsuitable for continuing tasks.

The idea behind Monte Carlo control methods is that they use the average return value for each state-action pair as the updated value for that state-action pair. Off-policy Monte Carlo weights the

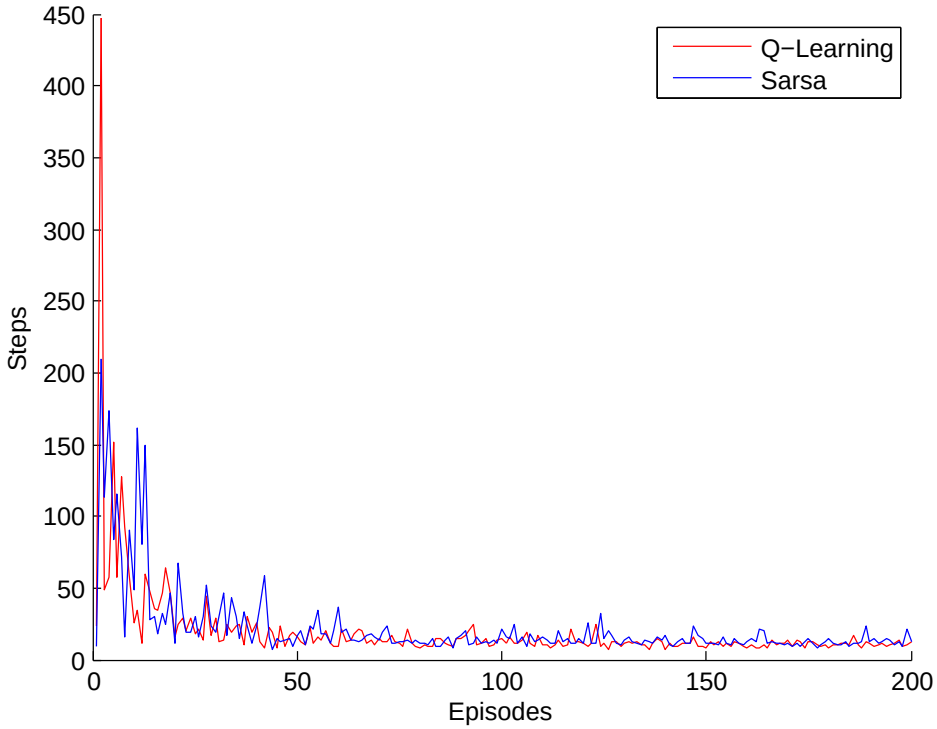

Figure 10: Comparing Sarsa and Q-learning algorithms for $\alpha = 0.3$, $\gamma = 0.7$ and $\epsilon = 0.1$.

average by the probability that the current action and each following action is chosen by its behaviour policy π', and computes this average only once per episode for each state-action pair. On-policy Monte Carlo does not computed a weighted average. Instead, it averages over every occurrence of each state-action pair, even when a pair repeatedly occurs in one episode.

Figure 11 shows the number of steps per episode while learning through on-policy Monte Carlo control, for different values of the discount factor γ. For both $\gamma = 0.75$ as well as $\gamma = 0.25$ is a very high peak to be found early on in the training process. This might be explained by it taking some exploratory actions, and then walking around almost randomly through the unexplored area, with higher values than the other state-action pairs have. We see that for all three discount factors, the policy converges quite early.

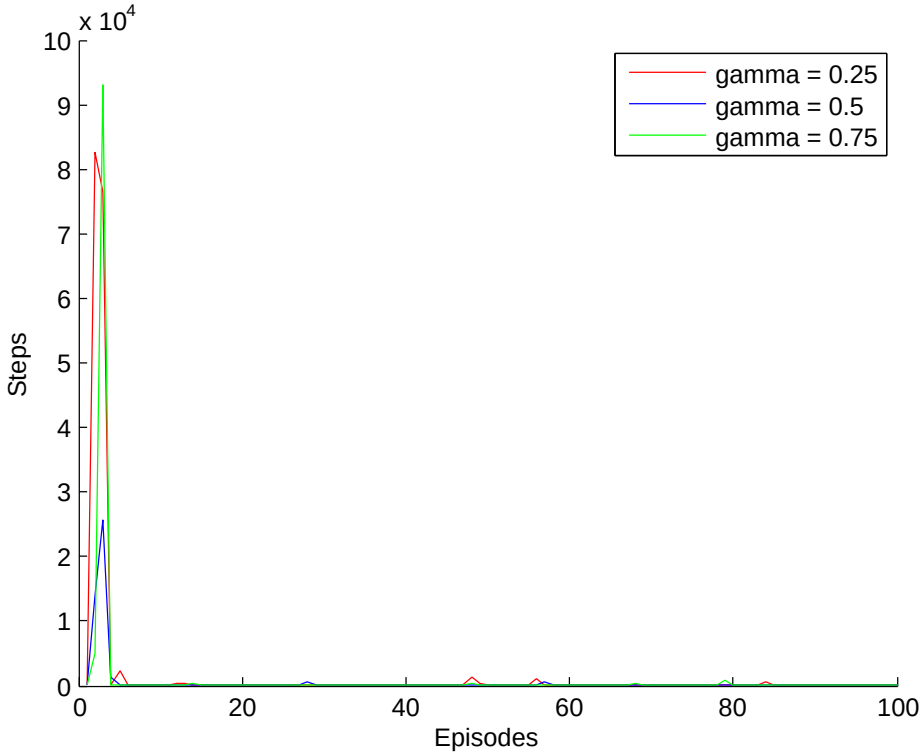

Figure 11: On-Policy Monte Carlo. Each state-action pair has an associated value of 15.

Off-policy Monte Carlo

The Off-policy Monte Carlo control method is divided in 2 parts; the behavior function and the estimation function. The behavior function (or behavior policy) is the one used for the agent's learning, generating learning episodes which provide "knowledge" for the estimation policy to use later. The behavior policy can be arbitrary but must be deterministic, and the table that it creates is updated at the end of each episode, updating pairs of states and actions that the agent used to find the prey. Consequentially, the predator's knowledge and its performance depend on the number of training episodes that the agent generates. First, we make use of ϵ-greedy action selection policy to generate episodes. These episodes start always with the predator and prey at maximal distance, and they end when the predator catches the prey. The behavior policy's Q table is computed as follows:

1. the behavior policy generates an episode making use of the ϵ-greedy action selection policy.
2. starting from the end of each episode, we are finding the state τ in which was the last time where $a_\tau \neq \pi(s_\tau)$ where π represents the behavior policy.
3. For each state-action pair appearing in the episode at time $\geq \tau$ from the first occurrence of (s, a), $w \leftarrow \prod_{k=t+1}^{T-1} \frac{1}{\pi'(s_k, a_k)}$, where w is called "weight" and is used for importance sampling, computing how important the return of an action is.
4. $N(s, a) \rightarrow N(s, a) + wR_t$, where R_t represents the return of action a at time t, weighted by w.
5. $D(s, a) \rightarrow D(s, a) + w$
6. $Q(s, a) \rightarrow \frac{N(s,a)}{D(s,a)}$

Using ϵ-greedy does not mean that the action selection policy selects more optimal actions than a random approach. This happens because behavior policy depends on the Q-table which was initialized with equal values for all state-action pairs. This was the learning part of this algorithm. It is obvious that it depends a lot on the action selection policy to update the values of each state-action pair that exists in the state. Throughout the learning process and for small number of episodes, there might be non-visited state action pairs. For this reason, a good selection policy, as well as a large number of episodes have to exist in order for this learning algorithm to converge into an optimal policy.

The estimation policy π will use $argmax_a Q(s, a)$ to compute the next action from state s, which brings us to the conclusion that the estimation policy is always choosing the next action greedily. As shown clearly in figure 11, the more training episodes the behavior policy generates, the better results we get from the estimation policy. Figures 12, 13, and 14, present how the estimation policy behaves for different discount factor values γ while choosing the next action ϵ-greedily for 500, 1000, and 10 000 episodes respectively. Looking at these figures, we realize that for a small discount factor and a large number of episodes this algorithm converges to the optimal policy. In contrary, even with a large number of episodes the predator tends to need more steps in order to catch the prey. This happens due to the fact that using a large discount factor, we gave the chance to the predator wait further for its reward. Figure 14 presents the computed policy that leads predator to catch prey in a very small number of steps, implied the functionality of the Monte Carlo off-line learning algorithm.

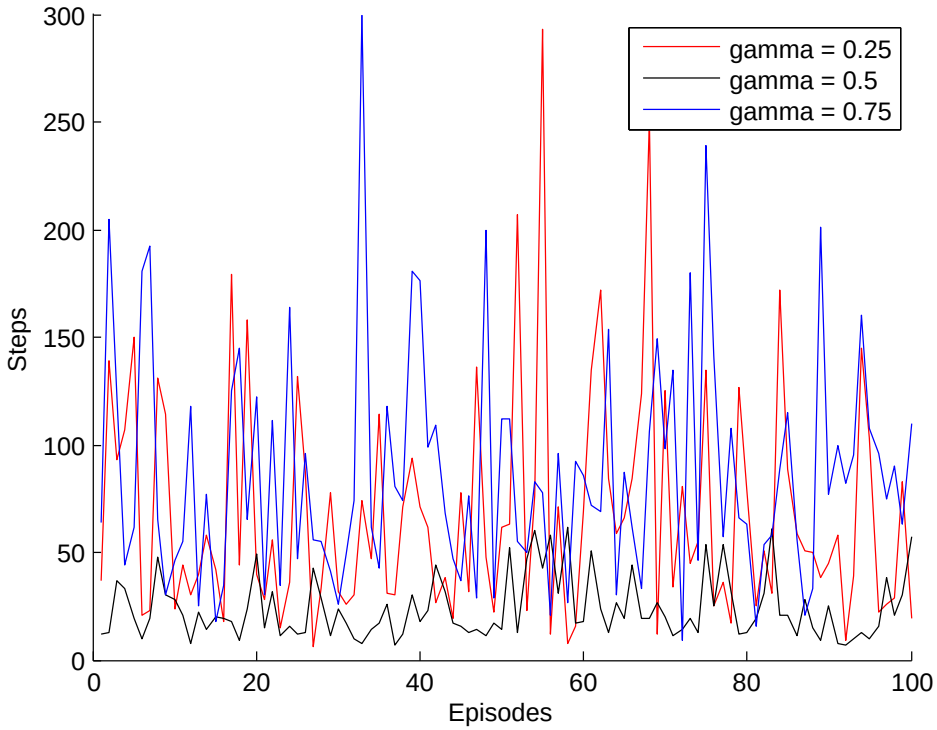

Figure 12: Comparing Off-Policy Monte Carlo for 500 training episodes, with ϵ-greedy action selection ($\alpha = 0.2$, $\gamma = 0.7$).

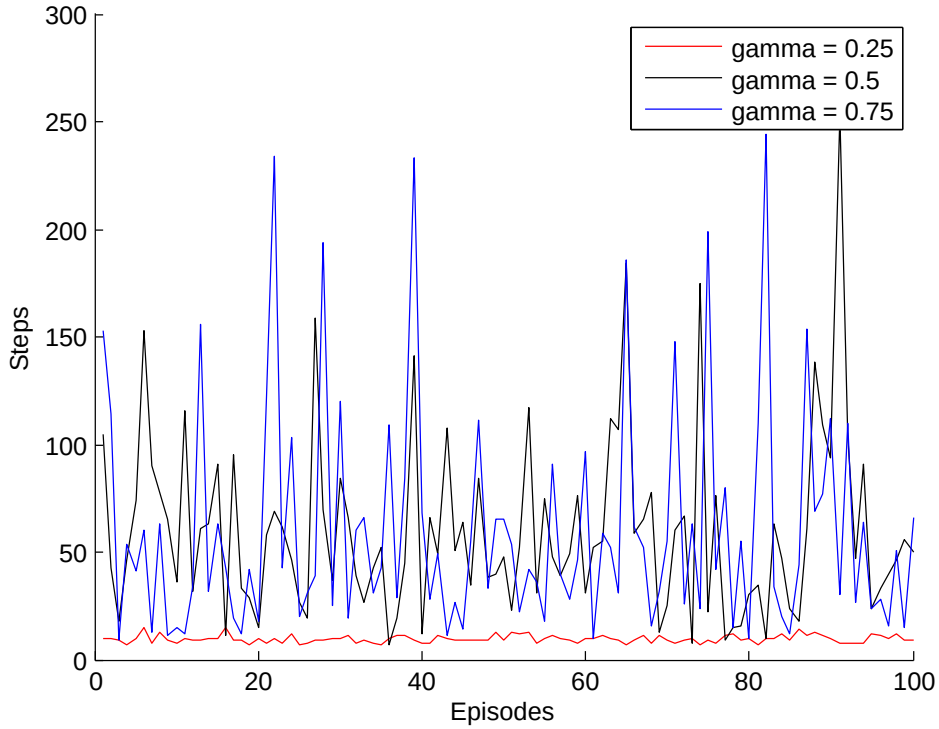

Figure 13: Comparing Off-Policy Monte Carlo for different values of γ, for 1000 training episodes.

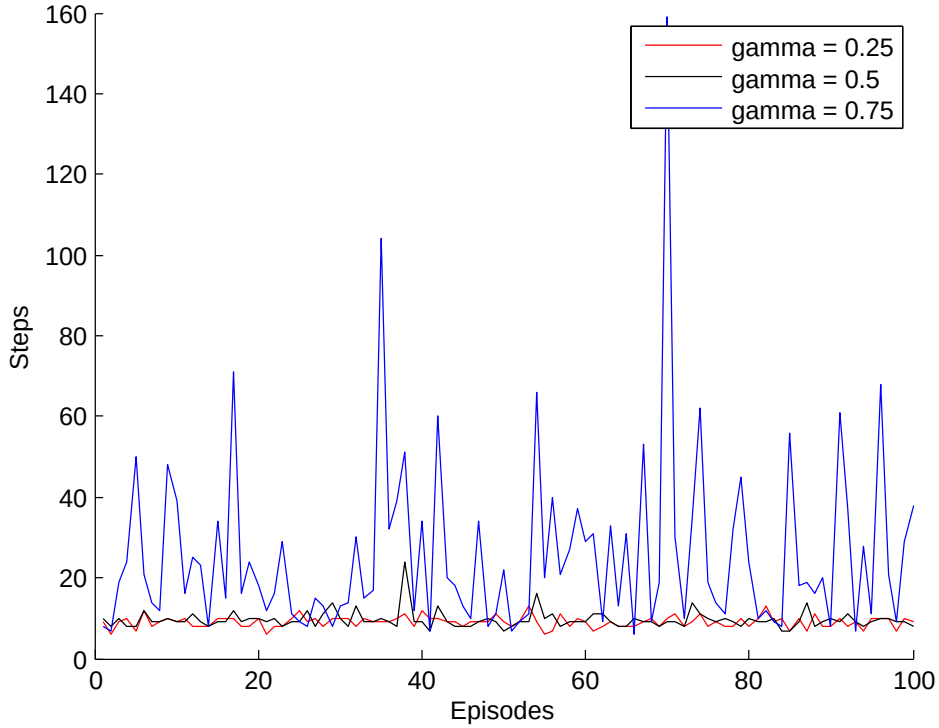

Figure 14: Comparing Off-Policy Monte Carlo for different values of γ, for 10000 training episodes.