Autonomous Agents 1
Assignment 2

By Group 4: Gieske, Gornishka, Koster, Loor

November 28, 2014

Introduction

This report contains the analysis of an implementation of single agent reinforcement learning. In these algorithms, the agent has no prior knowledge of the transition or reward function. Two different solutions to this problem are presented: in model-based learning, the agent learns the model from experience and plans on the learned model. In model-free learning, the agent learns state-values or Q-values directly. In this paper, model-free learning methods are considered.
Some prominent model-free reinforcement learning methods are on- and off-policy Monte Carlo Control, Q-learning (off-policy) and Sarsa (on-policy), where on-policy methods evaluate the values of a policy π *as it's being followed*, while off-policy methods allow the agent to follow a different policy π'. This paper reports on Q-learning and its on-policy equivalent, Sarsa. Moreover, the difference in action-selection between ϵ-greedy and softmax are compared.

Theory

Temporal difference learning methods

Temporal difference learning methods estimate state-values or Q-values by updating them on every time step (as opposed to the episodic updates of Monte Carlo-methods). Temporal difference learning methods are similar to Monte Carlo methods in that they both learn from experience directly, without requiring a model (thus, they do not exploit the Bellman-equations in equations 1 and 2) and they are similar to dynamic programming methods in that they both *bootstrap*, i.e. they estimate values based on earlier estimates.

$$V^*(s) = \sum_{s'} \mathcal{P}^a_{ss'} \left[\mathcal{R}^a_{ss'} + \gamma \max_a V^*(s') \right] \tag{1}$$

$$Q^*(s,a) = \sum_{s'} \mathcal{P}^a_{ss'} \left[\mathcal{R}^a_{ss'} + \gamma \max_a Q^*(s',a') \right] \tag{2}$$

Thus, in general, an update-rule for a TD-method would be similar to that of TD(0) in equation 3.

$$V(s) \leftarrow V(s) + \alpha[r + \gamma V(s') - V(s)] \tag{3}$$

Both Q-learning and Sarsa are TD-methods, and are described in more detail in sections Q-learning and Sarsa.

Q-Learning

Q-learning is a temporal difference method that uses[1] the update rule in equation 4. Because it retrieves the Q-value of the state-action pair where Q(s', a) is maximized, it is an *off-policy* method. The algorithm for Q-learning can be found in pseudocode in figure 1.

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left[r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t) \right] \tag{4}$$

```
Initialize Q(s,a) arbitrarily
Repeat (for each episode):
    Initialize s
    Repeat (for each step of episode):
        Choose a from s' using policy derived from Q (e.g., ε-greedy)
        Take action a, observe r, s'
        Q(s,a) ← Q(s,a) + α[r + γ max'_a Q(s', a') − Q(s, a)]
        s ← s';
    until s is terminal
```

Figure 1: The algorithm for one-step Q-learning [**?**]

Sarsa

Sarsa is a temporal difference method that uses the update rule in equation 5. Because it retrieves the Q-value of the state-action pair (s', a') where a' is selected using the policy π that is being evaluated, it is an *on-policy* method. The algorithm for Sarsa can be found in pseudocode in figure 2.

[1] More specifically, this is *one-step Q-learning*, which is the algorithm evaluated in this paper.

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left[r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t) \right] \tag{5}$$

Initialize Q(s,a) arbitrarily
Repeat (for each episode):
    Initialize s
    Choose a from s' using policy derived from Q (e.g., ϵ-greedy)
    Repeat (for each step of episode):
        Take action a, observe r, s'
        Choose a' from s' using policy derived from Q (e.g., ϵ-greedy)
        Q(s,a) $\leftarrow$ Q(s,a) + $\alpha[r + \gamma Q(s', a') - Q(s, a)]$
        s $\leftarrow$ s';
    until s is terminal

Figure 2: The algorithm for Sarsa [**?**]

Monte Carlo Methods

As stated before, there are several ways of learning a estimating value functions and discovering optimal policies. Again, Monte Carlo methods do not know (nor need) complete knowledge of the environment. Monte Carlo methods require only experience, such as sample sequences of states, actions, and rewards, from on-line interaction with an environment. Unlike TD-methods, Monte Carlo methods require a model. The model only needs to generate sample transitions, unlike Dynamic Programming methods which need complete probability distributions of all possible transitions. Therefore, Monte Carlo methods are an important tool for an agent to use when only the model is known.

On-policy Monte Carlo Control

On-policy Monte Carlo control (ONMC) does not use exploring starts. In order to ensure that each state is visited and each action is selected infinitely often is to keep selecting these actions. There are two ways of implementing OPMCC; first-visit Monte Carlo and every-visit Monte Carlo. In this case, every-visit Monte Carlo is implemented which is reflected in the algorithm below.

Initialize, for all $s \in (S), a \in (A)(s)$
    Q(s,a) $\leftarrow$ arbitrary
    Returns(s,a) $\leftarrow$ empty list
    $\pi \leftarrow$ an arbitrary ϵ-soft policy
Repeat forever:
    Generate an episode using exploring starts and π
    For each pair s,a appearing in the episode:
        $\mathcal{R} \leftarrow$ return following the first occurrence of s,a
            Append $\mathcal{R}$ to $Returns(s, a)$
            $Q(s, a) \leftarrow$ average($Returns(s, a)$)
        For each s in the episode:
            $a^* \leftarrow \arg\max_a Q(s, a)$
            For all $a \in \mathcal{A}(s)$:

$$\pi(s, a) \leftarrow \begin{cases} 1 - \epsilon + \dfrac{\epsilon}{|\mathcal{A}(s)|} & \text{if } a = a^* \\ \dfrac{\epsilon}{|\mathcal{A}(s)|} & \text{if } a \neq a^* \end{cases}$$

Figure 3: The algorithm for on-policy Monte Carlo [**?**]

Off-policy Monte Carlo Control

Off-policy Monte Carlo control (OFFMC) does not use exploring starts. In order to ensure that each state is visited and each action is selected infinitely often is to keep selecting these actions. There are two ways of implementing OFMCC; first-visit Monte Carlo and every-visit Monte Carlo. In this case, every-visit Monte Carlo is implemented which is reflected in the algorithm below.

Initialize, for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$:
- Q(s,a) $\leftarrow$ arbitrary
- N(s,a) $\leftarrow$ 0; Numerator
- D(s,a) $\leftarrow$ 0; Denominator of Q(s,a)
- $\pi \leftarrow$ an arbitrary deterministic policy

Repeat forever:
- Select a policy π' and use it to generate an episode:
  - $s_0, a_0, r_1, s_1, r_2, ..., s_{T-1}, a_{T-1}, r_T, s_T$
- $\tau \leftarrow$ latest time at which $a_t \neq \pi(s_\tau, a_\tau)$
- For each pair s, a appearing in the episode at time τ or later:
  - $\tau \leftarrow$ the time of first occurrence of s, a such that $t \leq \tau$
  - $w \leftarrow \prod_{k=t+1}^{T-1} \frac{1}{\pi'(s_k, a_k)}$
  - N(s,a) $\leftarrow$ N(s,a) + wR_t
  - D(s,a) $\leftarrow$ D(s,a) + w
  - Q(s,a) $\leftarrow \frac{N(s,a)}{D(s,a)}$
- For each $s \in \mathcal{S}$:
  - $\pi(s) \leftarrow \arg\max_a$ Q(s,a)

Figure 4: The algorithm for off-policy Monte Carlo [**?**]

Action selection methods

In order to select which action to choose according to a given policy, a tradeoff between exploration and exploitation must take place. This tradeoff is important when performing reinforcement learning as the rewards must be maximized, but exploration may lead to finding higher rewards. There are several action selection methods which can be used to select actions. The two techniques analyzed in this report are ϵ-greedy and softmax action selection.

In the case of ϵ-greedy, the best action a^* is given a probability of $1-\epsilon$. All actions a (including a^*) then receive an equal portion of ϵ as a probability. This is formalized in equation 6.

$$\forall a \in \mathcal{A} : p(a) = \begin{cases} 1 - \epsilon + \frac{\epsilon}{|\mathcal{A}|}, & \text{if } a = a^* \\ \frac{\epsilon}{|\mathcal{A}|} & \text{otherwise} \end{cases} \tag{6}$$

So that in $1-\epsilon$ of the cases, a^* is selected, and in ϵ of the cases a random action is chosen uniformly. Thus, an ϵ-greedy policy mainly exploits the known states to maximize the immediate reward. At probabilistically determined times however, this policy will explore new states. This may lead to undesired behavior as it is possible for the agent to stand beside the goal state when the ϵ-greedy policy turns to explore a new state, which can lead to high negative rewards in particular cases, such as a robot falling off a cliff. Also, ϵ-greedy does not consider the quality of non-optimal actions: an action with a value just below that of a^* receives

the same probability as an action with a much lower value. Another note using ϵ-greedy is the intuition that ϵ should decay as the agent has explored more states, as exploration is adding less information over time. There is, however, no clear-cut way to do decide how and when to decay ϵ.

Softmax action-selection offers a solution to one problem presented by ϵ-greedy policies. The greedy action a^* is still assigned the highest probability, but all other probabilities are ranked and weighted according to their values [1]. There are several ways of implementing the softmax action selection method, but for the purposes of this paper softmax uses a Boltzmann distribution, formalized in 7.

$$\forall a : p_t(a) = \frac{\mathrm{e}^{Q_t(a)/\tau}}{\sum_{b=1}^{n} \mathrm{e}^{Q_t(b)/\tau}} \tag{7}$$

The parameter τ in equation (2) is the *temperature* of the distribution. Low temperatures cause the values of $Q_t(a)$, and consequently those of $\mathrm{e}^{Q_t(a)/\tau}$, to grow much larger, thus increasing their differences. This leads to assigning a high probability to the optimal action and assigning lower probabilities to suboptimal actions. Of course, terrible actions get very low probabilities. For high temperatures, the opposite is true, where in the limit $\tau \to 0$, all actions become equiprobable [1].

It is unclear whether ϵ-greedy action selection is better than softmax action selection. The performance of either method may depend on the task at hand and human factors. Experimenting with both methods will lead to more insight in both algorithms.

Implementation

The implementation consists of the following files:

Agents_new
: This file contains implementions of the Agent class, the Prey class and the Predator class. Both the predator and the prey inherit functions of the Agent class. The Agent class contains functions any agent needs, such as a set of actions, a policy and other functions. As the predator is the agent is the agent this implementation focuses on, the predator class contains more functions than the predator class.

Helpers
: This file contains many helper functions. These functions aid in computation and decision making, but cannot (and need not) be part of a specific class.

Other_objects
: This file contains the Policy and Environment classes. The environment of the game as well as the rules are implemented in the Environment class. The Policy class contains the implementation of Q-Learning, Sarsa, ϵ-greedy, softmax action selection and more functions that help in determining and optimizing a policy as well as choosing an action of this policy.

Newstate
: This file contains the Game class as well as a demonstration function. The Game class instantiates the game, initialized the predator and the prey and assigns policies to these. The game is run N times and the result is printed. The demonstration function also performs Q-Learning, Sarsa and Monte Carlo. It also uses ϵ-greedy and softmax action selection. The results are printed in the command line and graphs are used for analysis.

Analysis

This section discusses the results of the implementations. In order to display and compare results, graphs are used. The title describes which parameters are analysed and the legend shows which color represents which setting of said parameters.

Q-Learning

Different discount factors

First, the effects of different discount factors on Q-learning is analysed. The results are displayed in the figure below.

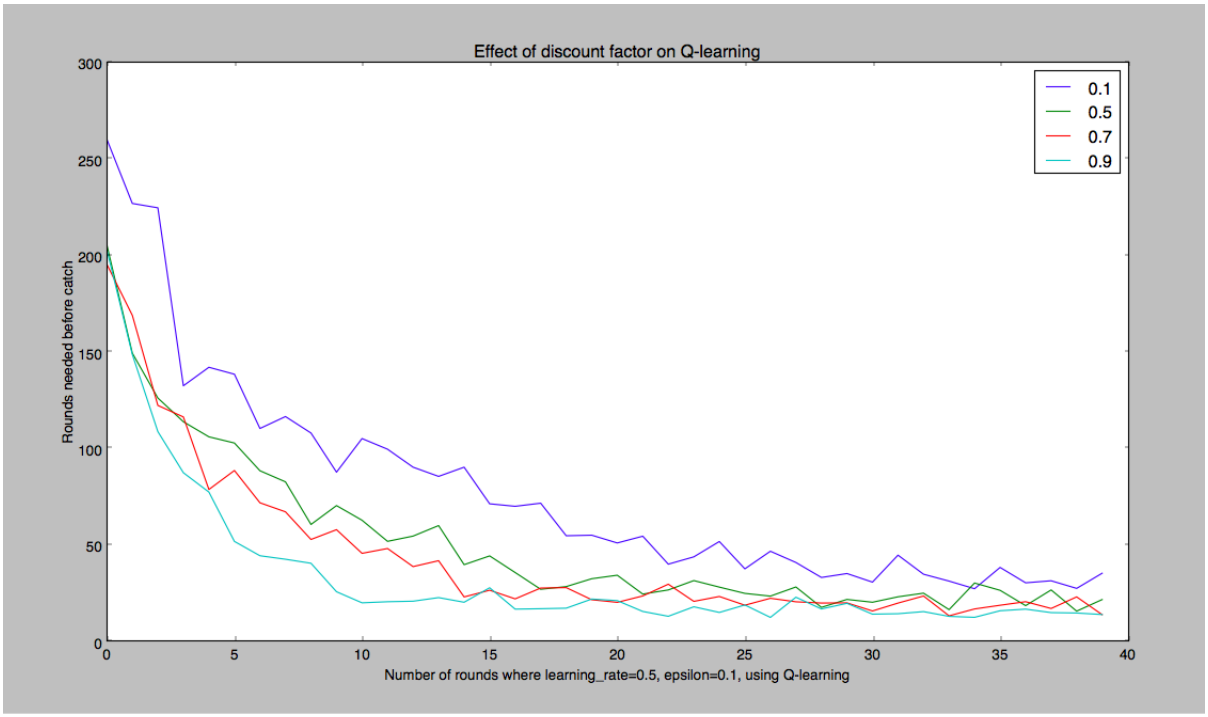

Figure 5: Q-learning using different discount factors

As shown, the higher the discount factor, the better the result. The height of the discount factor determines the importance of future rewards. A low discount rate tries to achieve a high immediate reward and a high discount rate tries to achieve a high overall reward. Since reaching each state yields a reward of 0, except for the goal state which yields a reward of 10, the future reward must be maximized. Therefore, this behaviour can be expected.

The effect of ϵ

As Q-learning is a learning algorithm, exploration and exploitation must be balanced well to find the optimal behaviour. The following graph shows different settings for ϵ.

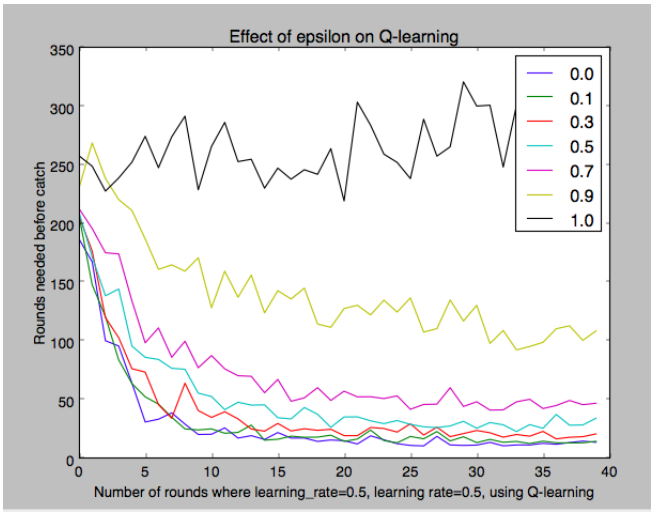

Figure 6: Q-learning using different epsilon values

As shown in the figure above, the lower the ϵ value is, the quicker the algorithm finds the optimal path. This can be expected since, as stated in the theory, an ϵ value of 0 leads to a greedy algorithm. What is interesting is that an ϵ value of 0.9 performs quite well compared to a completely exploratory policy, which seems to lead to random results. This shows how important explorations is and how strong an ϵ-greedy policy is. Even with the slightest bit of greedy behaviour, good results are achieved.

Different learning rates

It is possible to set different learning rates in Q-learning. The learning rate determines to which extent the new information will overwrite the old information. The following graph shows how important the relationship of old and new information is to achieve an optimal policy.

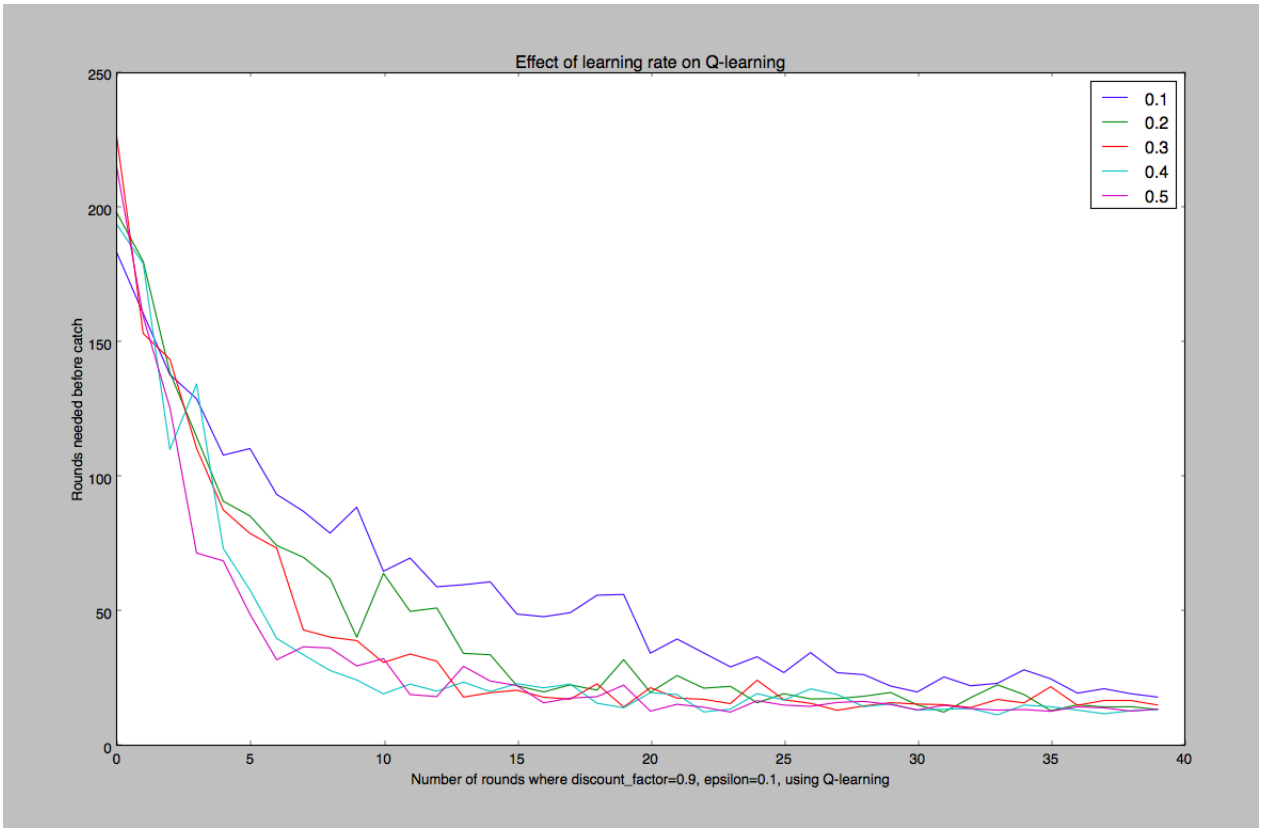

Figure 7: Q-learning using different learning rates

The figure above shows that for each learning rate between, and including, 0.1 and 0.5 leads to conversion. However, a low conversion rate leads to slower conversion than a high conversion rate. It is interesting to note that the learning rates of both 0.4 and 0.5 appear to converge at about the same speed. Remember that the learning rate determines to which extent the new information will overwrite the old information. It appears that, even when storing little new data, this leads to convergence. The more new data overwrites the old data the quicker the algorithm converges. Intuitively, this seems to be correct. However, overwriting all old data might lead to less than optimal behaviour. As stated before, learning rates of 0.4 and 0.5 seem to yield the same result. It is well possible that this is the limit until which the learning rate factor is beneficial to the algorithm. As this lies beyond the scope of this report, this can be researched in the future.

Different initialization for Q-learning

It was advised to initialize the policy of the predator optimistically, when performing Q-learning. When initializing a policy optimistically, it is interesting to see what happens when a policy is not optimistically initialized. The values chosen are 15, 10, 5, 0 and -5. Also, to see the effect of learning, the ϵ-value was set to either 0 or 0.1. The figure below shows the results of ϵ value 0.1.

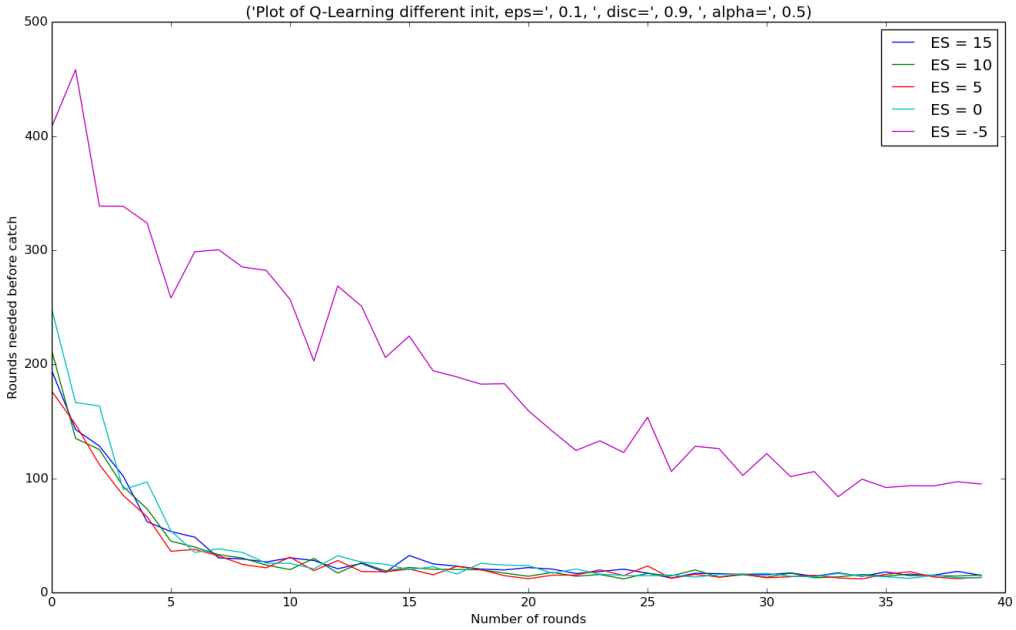

Figure 8: Q-learning with different policy initialization values and ϵ is 0.1

Each non-negative initialization appears to yield good results. Though it takes an initialization of 0 longer to converge, the policy does eventually converge to the same level as the optimistic policy. The negatively initialized policy behaves very poor. It does improve, but it takes significantly longer to converge than the non-negative initialized policies. It is expected that both the negative and zero initialization lead to slow learning. However, it is possible that the zero initialization is a border case where this is (one of) the last case(s) where the algorithm still learns enough to perform well.

Now let's analyse the same initialization values with an ϵ-value of 0.

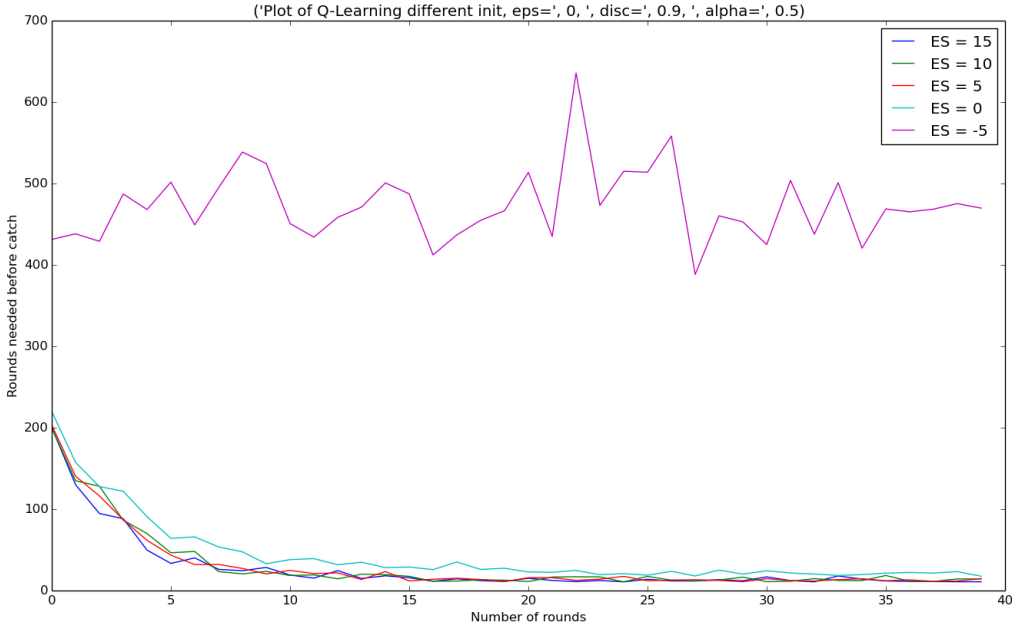

Figure 9: Q-learning with different initialization values and ϵ is 0

This leads to similar results as before. One of the differences is that the policy initialized with 0 performs worse than before and does not converge. This behaviour can be expected. At the non-negative initializations, there are values larger than zero. Therefore, executing a greedy policy still leads to convergence. As stated, it is expected that the initialization with zero values should behave similar to the negative initialization. Again, this is not the case. This is still a border case, as stated before.

In order to determine whether the zero initialization is indeed a border case, more runs can be performed. However, this falls beyond the scope of this report. It is important to test this is the future to determine the limits of this implementation.

ϵ-greedy vs. softmax

As the effect of ϵ-greedy action selection on Q-learning has already been discussed, let's analyse the effect of softmax action value selection.

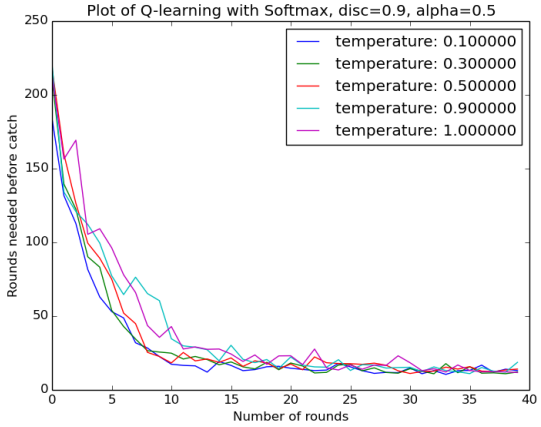

Figure 10: Q-learning using different temperatures in Soft-max

The graph shows that all results converge. It shows that the lower the temperature, the better the results. This is to be expected, as the lower the temperature leads to a greedy action selection. However, the temperature of the softmax action selection algorithm can not be zero, as the temperature is divided by. This ensures both optimal action selection and exploration. When analysing the higher temperatures, it these converge at about the same speed as the low temperatures. However, the higher the temperature, the less smooth the graph. As a high temperature (e.g. τ is 1) leads to an equiprobable action selection, the algoritm both explores and exploits to a certain level. This causes the algorithm to converge, but also taking longer (compared to lower temperatures such as τ is 0.1) to catch the prey in an episode.

Now let's look at the results between softmax and ϵ-greedy. As stated in the theory, the difference in effect between softmax action selection and ϵ-greedy action selection is unknown. The performance of either relies on the goal of the implementation, as well as human factors. Therefore, it is imperative to research the effects on this implementation. The ϵ value chosen for the ϵ-greedy algorithm is the same as the temperature, τ. This means that both algorithms behave greedy, but still explore from time to time.

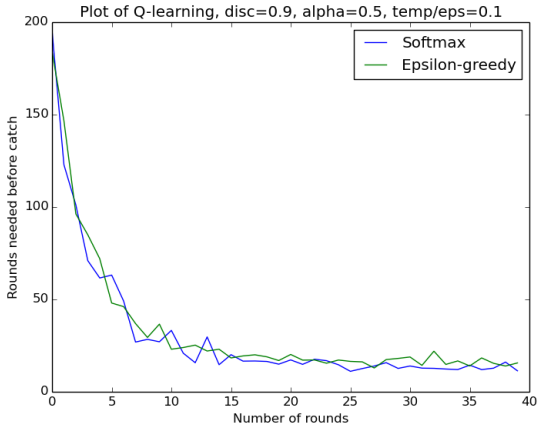

Figure 11: Q-learning ϵ-greedy and softmax action selection

The figure shows that both softmax and ϵ-greedy action selection converge at the same speed. However, softmax action selection eventually leads to capturing the prey in less rounds. Therefore, softmax action selection is preferred over epsilon-greedy action selection when the the prey needs to be caught in a minimum amount of timesteps.

Learning types

Aside from Q-learning, Sarsa and on-policy Monte Carlo were implemented. It is interesting to see the difference in performance between the algorithms. The results are shown in the figure below.

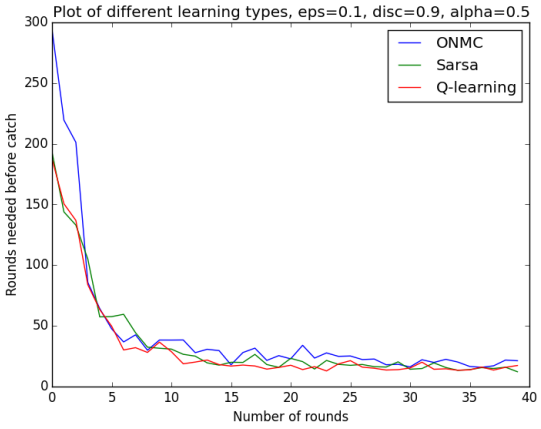

Figure 12: Different learning types set out against one another

The figure shows that all algorithms converge. Q-learning converges quickest, with Sarsa not much slower. On-policy Monte Carlo also converges quickly, but it starts with the worst Q-values. It also shows that of all algorithms, on-policy Monte Carlo converges to the least optimal number of rounds that lead to convergence.

The following table shows how many rounds were needed, on average, for the prey to catch the predator. This was calculated for the first 1000 and the last 1000 runs. This means that the learning in the very beginning and the learning in the very end are set out against each other.

	Avg nr/rounds (first 1000)	**Avg nr/rounds (last 1000)**	**STD (first 1000)**	**STD (last 1000)**
Q-learning	88.68	14.52	124.39	18.84
Sarsa	80.61	14.70	109.74	22.45
ONMC	79.68	15.78	112.11	21.64

Conclusion

Q-learning is an effective method to learn the Q-values of the states of environments.

After seeing the results, it can be concluded that the use of both Monte Carlo methods as well a temporal difference methods converge quickly. However, temporal difference methods outperform on-policy Monte Carlo. Of the temporal difference methods, Q-learning outperforms Sarsa as well. The differences are minimal, but they are there. Nevertheless, both converge to almost the same result. This leads to the conclusion that either Sarsa or Q-learning can be implemented, whatever the requirements, and the same results may be expected.

Files attached

- newstate.py
- agents_new.py
- other_objects.py
- helpers.py ...

Sources

1 Barto and Sutton (http://webdocs.cs.ualberta.ca/ sutton/book/the-book.html) ...