Autonomous Agents 1
Assignment 2

By Group 4: Gieske, Gornishka, Koster, Loor

November 24, 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, 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 and e(s, a) = 0 for all s,a
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 [1]

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.

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

Initialize Q(s,a) arbitrarily and e(s, a) = 0 for all s,a
Repeat (for each episode):
    Initialize s,a
    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)
        $\delta \leftarrow$ r + γ Q(s',a') - Q(s,a)
        e(s,a) $\leftarrow$ e(s,a) + 1
        For all s,a:
            Q(s,a) $\leftarrow$ Q(s,a) + $\alpha\delta$ e(s,a)
            e(s,a) $\leftarrow$ $\gamma\lambda$ e(s,a)
        s $\leftarrow$ s'; a $\leftarrow$ a'
    until s is terminal

Figure 2: The algorithm for Sarsa [1]

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.

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

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 $e^{Q_t(a)/\tau}$ to grow very larger, thus increasing their differences. 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 analyzation.

Analysis

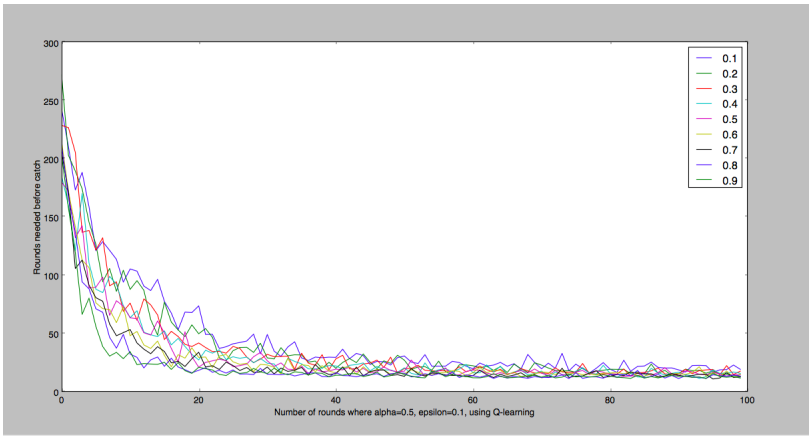
Q-Learning

Sarsa

ϵ-greedy vs. softmax

Effect of discount rates on learning

Q-learning



Conclusion

Files attached

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

Sources

References

[1] Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 1998.

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