Autonomous Agents Assignment 2

Tobias Stahl 10528199
Spyros Michaelides 10523316
Ioannis Giounous Aivalis 10524851
Francesco Stablum 6200982

October 4, 2013

1 Introduction

In this report we are going to use an already implemented Predator versus Prey grid to illustrate some of the differences and features of learning algorithms. In this scenario the Predator agent does not have any prior knowledge towards the environment and will make use of model-free learning, in order to reach its goal which is to catch the Prey. This will be attempted using Temporal Difference learning methods (TD), a combination of Monte Carlo and dynamic programming. TD learning methods can learn directly from experience, rather than having to rely on a model of the environment. TD methods can update action state values estimates based on other learned estimates, without having to wait for the final outcome.

2 Algorithms

In this part of the report the used algorithms are introduced on the basis of their Pseudocode description.

2.1 Q-Learning

The initial exercise of this lab assignment is to implement Q-learning, a temporal-difference (TD) learning algorithm. This algorithm is to be used by the predator agent to catch the prey.
Q-learning is an off-policy TD control algorithm. An off-policy TD algorithm is one in which the estimated value functions can be updated using

hypothetical actions, without having actually executed the actions themselves. Using this approach the algorithm can separate exploration from control, since the deterministic learned policy is separate to the exploration, meaning the agent could learn through the environment without necessarily having had the explicit experience. The steps involved can be summarised into the following Algorithm 1

Algorithm 1 Q-learning

```
Initialise Q(s, a) arbitrarily
for all Episodes do
    Initialise s
    for all Steps of episodes do
        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';
    end for
    Until s is terminal
end for
```

Where:

- α is the learning rate. Setting it to a high value will force learning to occur faster, whereas for a low value it will occur slower.
- $max_{a'}$ is the maximum reward reachable in the state s'
- γ is the value which gives future rewards less worth than immediate ones

2.2 Q-learning using ϵ-greedy

When the Q-learning algorithm to selects an action, there needs to be some form of trade-off between selecting the action with the highest estimated reward so far, and the the rest of the actions available. Limiting the action selection policy to only the best action learned so far, would mean potentially losing out on a better action in the future, being in a given state. To satisfy

this trade-off, ϵ-greedy policy uses a (in this example a small) probability of ϵ to select randomly between all between all the actions available in a given state, excluding the most optimal one so far. In turn, (as in this scenario ϵ is small), the most optimal action is chosen with a much larger probability, $1-\epsilon$, most of the time, giving the policy a tendency to exploit the best action so far most of the time, but not lose out on potentially better actions, which could be found through exploration of the other actions.

Based on the task at hand, the predator should directly learn a high reward policy without learning a model, since the agent is not supposed to know not know the transition probabilities, nor the reward structure of the environment. It is assumed that convergence should occur as long as all state action pair values continue to be updated using a certain policy (in this case ϵ-greedy.

2.3 Softmax Action-Selection Policy

In this section, a different action selection policy will be used in Q-learning instead of ϵ-greedy. ϵ-greedy policy satisfies the exploration/exploitation variance which is desired to be used in the Q-learning algorithm, although the way in which it achieves this could be a disadvantage in scenarios where the least favourable action has a much worse pay-off than the e.g., the second-best one. When the algorithm explores with a probability ϵ, it does not do this by taking into consideration the performance of the individual non-best actions themselves. The probability distribution between which non-best action is selected, is uniformly distributed and therefore, each one is as likely to be chosen as the rest. To optimise the performance of the Q-learning algorithm, it could be more efficient to make the selection among the non-best actions by weighing them according to their action-value estimates, thus increasing probability of selection for the higher action-value estimates, and hence making a more guided (by value) exploration. Algorithm 2 differs from ϵ-greedy in Operation 5.
Where:

- τ is a positive parameter called *temperature*. A high temperature leads the actions to be almost equiproble, low tempreature values cause a greater difference with $\tau \rightarrow 0$ beeing the same as greedy action selection.

Algorithm 2 Softmax

Initialise $Q(s, a)$ arbitrarily
for all Episodes **do**
  Initialise s
  for all Steps of episodes **do**
    Choose Choose a with probability $\frac{\exp^{Q_t(a)/\tau}}{\sum \exp^{Q_t(b)/\tau}}$
    Take action a, observe r, s'
    $Q(s, a) \leftarrow Q(s, a) + \alpha[r + \gamma max_{a'} Q(s', a') - Q(s, a)]$
    $s \leftarrow s'$;
  end for
  Until s is terminal
end for

2.4 Sarsa

Sarsa, like Q-learning is a temporal difference algorithm, meaning that it compares temporally successive predictions. Unlike Q-learning though, Sarsa is an on-policy TD method. In on-policy TD learning the algorithm learns the value of the policy that is used to make the decisions, meaning directly through experience. This is in contrast to off-policy where value functions are not updated solely on experienced actions. The action selection policies previously discussed in Q-learning are also applicable for use in Sarsa. Again, there is the choice of specifying the trade-off between exploitation/exploration by setting the ϵ parameter when using ϵ-greedy, or using the softmax policy.

Algorithm 3 Sarsa

Initialise $Q(s, a)$ arbitrarily
for all Episodes **do**
  Initialise s
  Choose a from s using derived from Q (e.g., ϵ-greedy)
  for all Steps of episodes **do**
    Take action a, observe r, s'
    Choose a' from s' using policy derived from Q (e.g., ϵ-greedy)
    $Q(s, a) \leftarrow Q(s, a) + \alpha[r + \gamma Q(s', a') - Q(s, a)]$
    $s \leftarrow s'$; $a \leftarrow a'$;
  end for
  Until s is terminal
end for

2.5 Monte-Carlo Methods

Monte-Carlo methods are also usually used in situations where there is no model of the environment's dynamics, and so can learn from the the experience they gain through each complete episode. The main difference between Monte-Carlo and Temporal Difference (TD) methods is that in TD methods the final outcome does not need to be reached until the state-action values can be updated, whereas in Monte-Carlo methods, it does. Monte-Carlo methods also use a variation of generalized policy iteration for the control problem, so the main difference lies in the way way the prediction problem is solved. Through the number of episodes (i.e. experience), the action value function for each state should average towards the true value function, and the optimal action for each state could then be computed. For this to be true, the hypothesis must be taken in that:

- The start of each episode must start at a state-action pair, and all state-action pairs have a chance of being used as a starting state-action. With a large enough number of episodes, each state-action pair should be visited enough times in order to make sufficient exploration for the best action to take, given a certain state.

This hypothesis can be unrealistic in some real-problem scenarios where for example the number of episodes generated cannot be large enough to obtain optimal state-action values. This be overcome in two ways, which will be explained in the following subsections.

2.5.1 On-policy Monte-Carlo Control

On-Policy Monte-Carlo Control methods evaluate or improve the making of decisions policy. In order to achieve this, the problem of 'too many episodes needed for exploring all the start state-action value pairs' needs to be dealt with. Using an action selection policy such as ϵ-greedy, the policy would gradually shift to a deterministic optimal policy. The algorithm uses first-visit Monte-Carlo methods to estimate the action value function for the current policy, a method in which the average of just the first visit to state s, for all the given episodes, is determined, and generalised policy iteration will be moved toward an ϵ-greedy policy. Thus the greedification is, as in Q-learning and Sarsa, replaced by soft greedification. ϵ-greedy policy will guarantee there is an improvement over any policy using ϵ-soft policies.

Algorithm 4 On-policy Monte-Carlo Control

for all $s \in S, a \in A(s)$ **do**
  Initialise:
  $Q(s,a) \leftarrow$ empty list
  $Returns(s,a) \leftarrow$ empty list
  $\pi \leftarrow$ an arbitrary ϵ-soft policy
  loop
    Generate and episode using π
    for each pair s,a appearing in the episode **do**
      $R \leftarrow$ return following the first occurrence of s,a
      Append R to $Returns(s,a)$
      $Q(s,a) \leftarrow$ average($Returns(s,a)$)
    end for
    for each s in the episode **do**
      $a* \leftarrow argmax_a Q(s,a)$
      for all $a \in A(s)$ **do**

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

      end for
    end for
  end loop
end for

2.5.2 Off-policy Monte-Carlo Control

As in Q-learning, in Off-Policy Monte Carlo Control the estimate of the value of a policy is separated from the policy used for control. This algorithm is off-policy in the same sense as Q-learning, meaning that it can separate exploration from control, and and similar to On-policy Monte-Carlo Control in that it can learn from the the experience it gains only through each complete episode.

Algorithm 5 Off-policy Monte-Carlo Control

for all $s \in S, a \in A(s)$: **do**
  Initialise:
  $(s, a) \leftarrow$ arbitrary
  $N(s, a) \leftarrow 0$; Numerator and
  $D(s, a) \leftarrow 0$; Denominator of $Q(s, a)$
  $\pi \leftarrow$ an arbitrary deterministic policy
  loop
    Select a policy π' and us it to generate an episode:
      $s_0, a_0, r_1, s_1, a_1, r_2, ..., s_{T-1}, a_{T-1}, r_T, s_T$
    $\tau \leftarrow$ latest time at which $a_r \neq \pi(s_\tau$
    for each pair s, a appearing in the episode at time τ or later **do**
      $t \leftarrow$ the time of the first occurrence of s, a such that $t \geq \tau$
      $w \leftarrow \prod_{k=t+1}^{T-1} \frac{1}{\pi'(s_k, a_k)}$
      $N(s, a) \leftarrow N(s, a) + wR_t$
      $D(s, a) \leftarrow D(s, a) + w$
      $Q(s, a) \leftarrow \frac{N(s,a)}{D(s,a)}$
    end for
    for each $s \in S$ **do**
      $\pi \leftarrow argmax_a Q(s, a)$
    end for
  end loop
end for

3 Experiments

This section describes the properties of the system the experiments were tested on and give an overview of the achieved results including plots to visualize them.

3.1 System Properties

The experiments were performend on a

3.2 Experiment 1

The source code for the implementation of this experiment can be found under the package $aa2013.Assignment2$ in the file $Experiment2\_1.java$.

3.2.1 Hypotheses

The first experiment aims to measure the performance of the predator catching the predator with different learning rates α and different discount factors γ over time. Therefore the average performance of 100 simulations with different α and γ for learning with an episode count from 50 to 950 is taken into account. The reduced state space with 11x11 states is chosen, in order to save computational time.
The expected outcome is that high learning rates tend to always replace the current value with the new estimates and converge quickly, while a small learning rate value leads to a slow convergence and seems to trust the current estimate. [2]
A small discount factor γ lets the agent appear short-sighted, while a value close to one increases the agents ambition for a long-term high reward.

3.2.2 Results

The results of this experiment are plottet in Table 1. The five chosen learning rates α are shown in each of the four graphs with different discount factors γ.
The interpretation of these results will follow in the next section.

3.2.3 Interpretation

As shown in the plots of the results, the hypothesis that a high learning rate α converges faster than a low one was correct. All of the four plots show that agents with a learning rate α of 0.1 or 0.2 need the longest time to perform well and agents with a learning rate of 0.5 achieve a good performance the fastest. Best to see is that in the plots with $\gamma = 0.7$ and $\gamma = 0.9$.
The influence of γ can be visible, if the different plots are compared to each other. No matter which learning rate α, all agents perform better earlier with a value close to one.

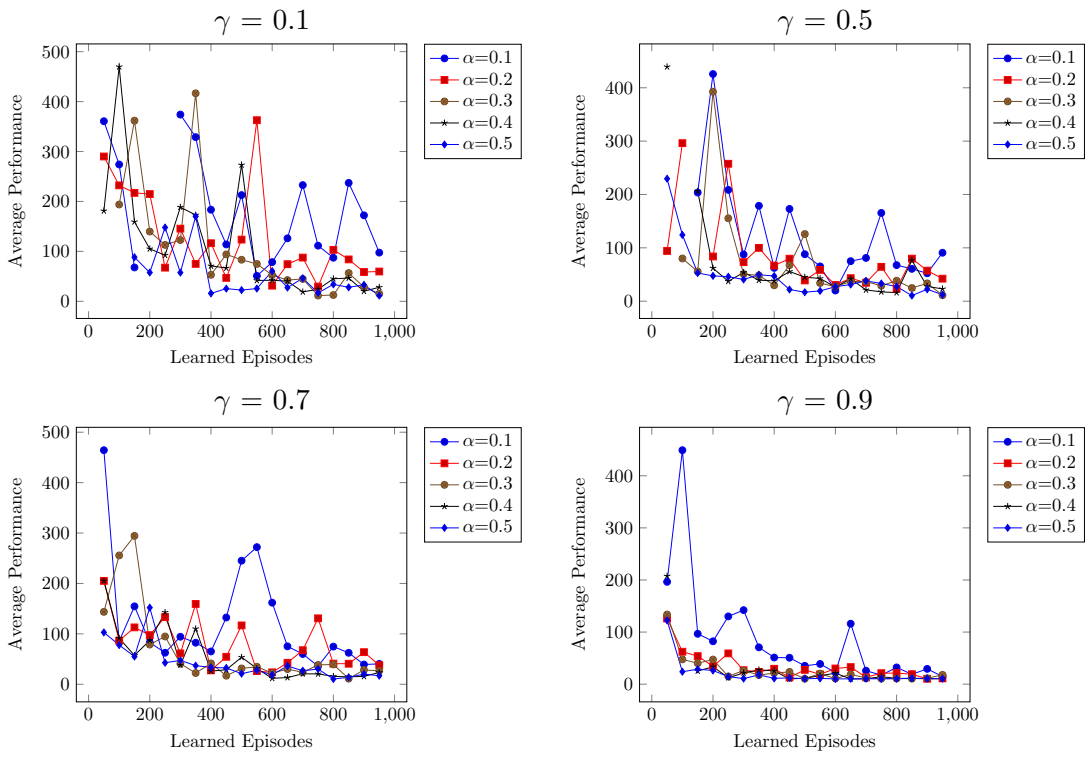

Table 1: Plots show the average performance over time spendt learning the policy with learning rates α from 0.1 to 0.5. The first plot shows the curve for discount factor $\gamma = 0.1$, second $\gamma = 0.5$, third $\gamma = 0.7$ and fourth $\gamma = 0.9$

3.2.4 Findings

Higher learning rates and discount factors close to one seem to achieve faster convergence in multi-armed bandit problems.

3.3 Experiment 2

The source code for the implementation of this experiment can be found under the package $aa2013.Assignment2$ in the file $Experiment2\_2.java$.

3.3.1 Hypotheses

The second experiment observes the impact of ϵ in the ϵ-greedy action selection policy and the initialization of the Q-table. ϵ determines the exploration rate, with values close to zero favouring exploitation and values close to 1 preferring exploration.
In order to test this, the constant learning rate $\alpha = 0.5$ and the constant discount factor $\gamma = 0.9$ are chosen, since these values had the best performance in the previous experiment.
In the previous experiment the Q-Table was initialized optimistically, with values higher than the actual reward to receive (15), which encourages an early exploration, since any actual reward is less than the actual reward. Changing this value to a pessimistic initialization (0) on the other hand inspires exploitation.
A high ϵ is assumed to almost always exploit and thus might not find a good policy, while low values are considered to converge to the best policy. The performance over time is measured again, in order to show how these variables influence the learning speed.

3.3.2 Results

The results of this experiment are plottet in 2.

3.3.3 Interpretation

The results of this experiment supports the hypothesis that high ϵ struggle to find a well-performing policy, as ϵ=0.8 and ϵ=0.9 proof. On the other hand ϵ=0.1 and ϵ=0.2 outperform all other agents with different ϵ.
A significant difference between the Q-Value initialization is not evident, but the agents with pessimistically initialized Q-Values need slightly longer to achieve the same performance as the agents with an optimitic intialization.

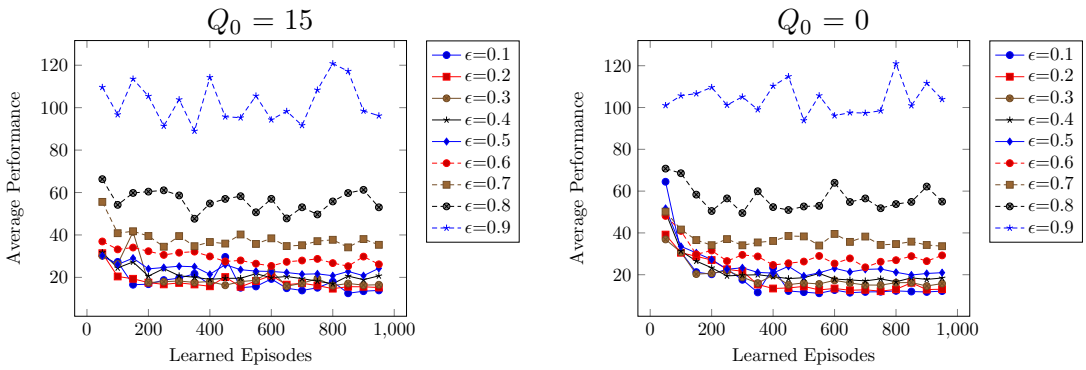

Table 2: Plots show the average performance over time spendt learning the policy with ϵ from 0.1 to 0.9. The first plot shows the curve for pessimistically initialized Q-Values = 0 and the second optimistic Q-Values = 15

3.3.4 Findings

It is preferable to initialize the Q-Values optimistically. Low ϵ values in an ϵ-greedy action selection policy outperform high values.

3.4 Experiment 3

The source code for the implementation of this experiment can be found under the package *aa*2013.*Assignment*2 in the file *Experiment*2_3.*java*.

3.4.1 Hypotheses

In this experiment the difference between the ϵ-greedy and the softmax action selection policy is researched. The ϵ-greedy action selection chooses all actions, except the action with the highest estimated reward so far, with the same probability, while the softmax approach weights the other actions and chooses actions with more pay-off with a higher probability.
The setup of this experiment is that softmax action selection policy is compared to an 0.1-ϵ-greedy action selection policy, since this showed the best results in the previous experiment, with the same values for γ, α and initial Q-Values
Given the fact that softmax weights all action, the hypothesis is that softmax outperforms ϵ-greedy as long as its *temperature* τ is properly tuned.

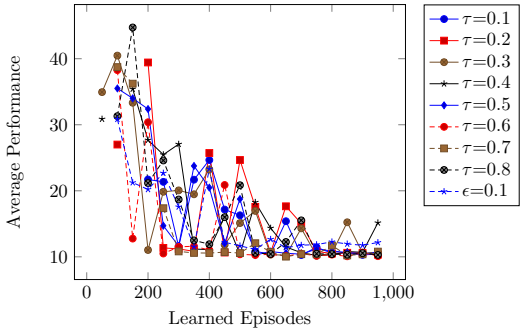

Table 3: Plots show the average performance over time spendt learning the policy with τ from 0.1 to 0.8 compared to $\epsilon = 0.1$.

3.4.2 Results

The results of this experiment are plotted in 3.

3.4.3 Interpretation

The results of this experiment supports the hypothesis that softmax over the course of the 1,000 learned episodes, outperforms ϵ-greedy. It can also be observed that for low values of τ the performance is better at more episodes since the actions will have been applied more often resulting in an optimally weighted rank between choices of action, but consequently performs poorly in the initial number of episodes compared to high values of τ, since the actions have been applied too few times for it to have a good enough weighted rank. The opposite can be said about the high values of τ, by observing that in fewer episodes it performs better, and in more episodes worse than it's counter part low valued τ

3.4.4 Findings

Softmax, does outperform ϵ-greedy when used by Q-learning, but the most optimal performance would be achieved if the τ temperature of Softmax could be tuned according to the number of learned episodes.

3.4.5

3.5 Experiment 4

3.5.1 Experiment 4.1

The source code for the implementation of this experiment can be found under the package $aa2013.Assignment2$ in the file $Experiment2\_4OnMC.java$.

3.5.2 Hypotheses

3.5.3 Results

3.5.4 Interpretation

3.5.5 Findings

3.5.6 Experiment 4.2

The source code for the implementation of this experiment can be found under the package $aa2013.Assignment2$ in the file $Experiment2\_4OffMC.java$.

3.5.7 Hypotheses

3.5.8 Results

3.5.9 Interpretation

3.5.10 Findings

3.5.11 Experiment 4.1

The source code for the implementation of this experiment can be found under the package $aa2013.Assignment2$ in the file $Experiment2\_4Sarsa.java$.

3.5.12 Hypotheses

3.5.13 Results

3.5.14 Interpretation

3.5.15 Findings

4 Conclusion

References

[1] Richard S. Sutton and Andrew G. Barto , *Reinforcement Learning: An Introduction*. The MIT Press, Cambridge, Massachusetts

[2] Eyal Even-Dar and Yishay Mansour , *Learning Rates for Q-learning*. Journal of Machine Learning Research 5 (2003)