Reinforcement Learning - Exercise Sheet 3

Tom Lieberum - ID 13253042 Erik Jenner - ID 13237896

16. September 2020

3.1 Coding Assignment - Monte Carlo

1.

First we expand V_n:

$$V_n = \frac{\sum_{k=1}^{n} W_k G_k}{n} = \frac{W_n G_n + \sum_{k=1}^{n-1} W_k G_k}{n}$$

Multiplying with n yields

$$nV_n = W_n G_n + \sum_{k=1}^{n-1} W_k G_k = W_n G_n + (n-1)V_{n-1}$$

$$\begin{aligned}\iff V_n &= \frac{n-1}{n} V_{n-1} + \frac{1}{n} W_n G_n \\ &= V_{n-1} + \frac{1}{n}(W_n G_n - V_{n-1})\end{aligned}$$

2.

See the autograd file.

3.

The key difference between Dynamic Programming and Monte Carlo methods is that the former does a wide, one-step backup, whereas the latter compute a maximally deep but very narrow backup. So, if it is easy to simulate trajectories, but expensive to compute the wide backups for all states and actions or the transition probabilities are not known, then we should use Monte Carlo methods. However, if simulating trajectories is expensive, maybe because the simulation runs in a detailed physics engine or we can only sample trajectories on real robots, then we want to use Dynamic Programming instead. Another important advantage of Dynamic Programming is that it does not depend on the MDP terminating. This means it can be used in continuing environments or as an online updating scheme, whereas Monte Carlo methods need each episode to terminate before they can perform an update.

4.3 Coding assignment - Temporal Difference Learning

1.

See the autograd file.

2.

In contrast to example 6.6 from RL:AI, we observe that Q-learning performs better than SARSA in learning the given task. Presumably, this is because in this particular environment there is no cliff which would yield a large negative reward. The agent only gets pushed back a bit when acting sub-optimally. Therefore a greedy strategy will be successful.

4.4 Maximization Bias

1.

If we assume that Sarsa acts ϵ-greedily w.r.t. its estimate of the Q-function we obtain

Sarsa:

$$\begin{aligned} Q(B, a_1) &= 0 \\ Q(B, a_2) &= 1 \\ Q(B, a_3) &= 0.5 \\ Q(B, a_4) &= 0.5 \\ Q(A, L) &= 0.7 \\ Q(A, R) &= (1-\epsilon) + 0.5\epsilon = 1 - 0.5\epsilon \end{aligned}$$

Q-learning

$$\begin{aligned} Q(B, a_1) &= 0 \\ Q(B, a_2) &= 1 \\ Q(B, a_3) &= 0.5 \\ Q(B, a_4) &= 0.5 \\ Q(A, L) &= 0.7 \\ Q(A, R) &= 1 \end{aligned}$$

2.

$$\begin{aligned} Q(B, a_i) &= 0.5, \text{ where } i \in \{1,2,3,4\} \\ Q(A, L) &= 0.7 \\ Q(A, R) &= \sum_{i=1}^{4} Q(B, a_i) = Q(B, a_1) = 0.5 \end{aligned}$$

3.

Moth methods suffer from maximization bias which can be observed in the estimate for $Q(A, R)$, which those methods calculate by taking an (ϵ-)greedy approach in the successor state B. Since both methods compute the maximum over actions[1], they overestimate the value of the subsequent state B.

4.

The root of the problem of maximization bias is that the learned Q-function is used both to sample/select the next action as well as for judging the selected action. Thus those two estimates are correlated, leading to a bias.

In Double Q-learning, we learn two Q-functions on different datasets, i.e. different sets of episodes. For this example, that would mean that they observe different values for the transition from $B \to T$, resulting in different estimates of $B's$ value. The estimates Q_1 and Q_2 would be independent and so we could use $Q_2(B, a)$ as an unbiased estimate of the value of taking action a in state B when updating the value of Q_1 of taking action R in state A (or vice versa)[2].

[1]Sarsa does this with probability $1 - \epsilon$.

[2]See also Hado van Hasselt: 'Double Q-Learning'