Reinforcement Learning

Author: Nicklas Hansen (s153077@student.dtu.dk)

Chapter: 7, 8

Exercise 6.13

In Double Q-learning, the update rule is

$$Q_1\left(S_t, A_t\right) \leftarrow Q_1\left(S_t, A_t\right) + \alpha\left[R_{t+1} + \gamma Q_2\left(S_{t+1}, \arg\max_a Q_1\left(S_{t+1}, a\right)\right) - Q_1\left(S_t, A_t\right)\right]$$

To produce a similar update equation for the Double Expected Sarsa algorithm with an ϵ-greedy target policy, recall that the update equation for regular Expected Sarsa is

$$\begin{aligned} Q\left(S_t, A_t\right) &\leftarrow Q\left(S_t, A_t\right) + \alpha\left[R_{t+1} + \gamma \mathbb{E}_\pi\left[Q\left(S_{t+1}, A_{t+1}\right) | S_{t+1}\right] - Q\left(S_t, A_t\right)\right] \\ &\leftarrow Q\left(S_t, A_t\right) + \alpha\left[R_{t+1} + \gamma \sum_a \pi(a|S_{t+1}) Q\left(S_{t+1}, a\right) - Q\left(S_t, A_t\right)\right] \end{aligned}$$

so to construct a Double Expected Sarsa we need to switch out the maximization of Q_1 over a in Double Q-Learning with the expectation of Q_2 given policy π:

$$Q_1\left(S_t, A_t\right) \leftarrow Q_1\left(S_t, A_t\right) + \alpha\left[R_{t+1} + \gamma \sum_a \pi(a|S_{t+1})\ Q_2(S_{t+1}, a) - Q_1\left(S_t, A_t\right)\right]$$

Exercise 7.1

The MC error can be written as a sum of TD errors if the value estimates don't change from step to step:

$$\begin{aligned} G_t - V\left(S_t\right) &= R_{t+1} + \gamma G_{t+1} - V\left(S_t\right) + \gamma V\left(S_{t+1}\right) - \gamma V\left(S_{t+1}\right) \\ &= \delta_t + \gamma\left(G_{t+1} - V\left(S_{t+1}\right)\right) \\ &= \delta_t + \gamma\delta_{t+1} + \gamma^2\left(G_{t+2} - V\left(S_{t+2}\right)\right) \\ &= \delta_t + \gamma\delta_{t+1} + \gamma^2\delta_{t+2} + \cdots + \gamma^{T-t-1}\delta_{T-1} + \gamma^{T-t}\left(G_T - V\left(S_T\right)\right) \\ &= \delta_t + \gamma\delta_{t+1} + \gamma^2\delta_{t+2} + \cdots + \gamma^{T-t-1}\delta_{T-1} + \gamma^{T-t}(0-0) \\ &= \sum_{k=t}^{T-1} \gamma^{k-t}\delta_k \end{aligned}$$

Keeping this assumption, we can also write the n-step error used in equation (7.2) of the book as a sum of TD errors in a similar fashion, where

$$
\begin{aligned}
G_{t:t+n} &\doteq R_{t+1} + \gamma R_{t+2} + \cdots + \gamma^{n-1} R_{t+n} + \gamma^n V_{t+n-1}\left(S_{t+n}\right) \\
&= \gamma^n V_{t+n-1}(S_{t+n}) + \sum_{k=1}^{n} \gamma^{k-1} R_{t+k}
\end{aligned}
$$

Writing out the n-step error then yields

$$
\begin{aligned}
G_{t:t+n} - V(S_t) &= R_{t+1} + \gamma R_{t+2} + \ldots + \gamma^{n-1} R_{t+n} + \gamma^n V(S_{t+n}) - V(S_t) \\
&= \delta_t + \sum_{k=t+1}^{t+n-1} \gamma^{k-1} R_{t+k} \\
&\text{which expands to...} \\
&= \sum_{k=t}^{t+n-1} \gamma^{k-t} \delta_k
\end{aligned}
$$

if we define $\delta_t = R_{t+1} + V(S_{t+1}) - V(S_t)$ analogous to the definition in the book.

Exercise 7.3

Using a larger state space for the random walk task is necessary in order to produce an accurate measure of performance for larger values of n. Considering a given episode of the random walk, we know that the n-step TD methods degrade to a MC method for any trajectory of length n or more. Therefore, we should not expect to see any significant difference in performance of, say, 256-step TD and 512-step TD for a small state space of just 5 like in the original formulation of the task.
As indicated by Figure (7.2) of the book, the average RMS error tends to increase as n increases (relative to state space), so we should expect 1-step TD methods and 2-step TD methods to perform the best on the original, smaller random walk.
Changing the value of the left-side outcome from 0 to -1 results in a faster convergence, as left-side state-values are updated to negative values whereas the right-side of the walk is updated towards 1. As n-step TD methods update multiple states at once, states in the left side of the walk should quickly get abandoned for smaller values of n.

Exercise 7.5

We modify the n-step TD method pseudo-code on page 144 to initialize two policies; a target policy π and the behaviour policy b like in the off-policy case shown on page 149.
Further, for $\tau \geq 0$ we want to compute the importance sampling ratio $\rho_t = \prod_{i=\tau+1}^{\min(\tau+n,T-1)} \frac{\pi(A_i \mid S_i)}{b(A_i \mid S_i)}$ and then use the new update rule for G as proposed by equation (7.13) in conjunction with equation the n-step TD update rule from equation (7.2):

$G \leftarrow 0$

$h \leftarrow \min(\tau + n, T)$

Loop for $i = \tau + 1 \dots h$

$\quad G \leftarrow \rho_i(R_i + \gamma G)$

If $\tau + n < T$, then $G \leftarrow G + (1 - \rho_{\tau+1})V_{h-1}(S_{\tau+1})$

$V(S_\tau) \leftarrow V(S_\tau) + \alpha \left[G - V(S_\tau)\right]$

Exercise 7.11

We know that the general recursive definition of the tree-backup n-step return is

$$G_{t:t+n} \doteq R_{t+1} + \gamma \sum_{a \neq A_{t+1}} \pi(a \mid S_{t+1}) Q_{t+n-1}\left(S_{t+1}, a\right) + \gamma\pi\left(A_{t+1}|S_{t+1}\right) G_{t+1:t+n}$$

and we also know that $\delta_t \doteq R_{t+1} + \gamma \overline{V}_t(S_{t+1}) - Q(S_t, A_t)$ where $\overline{V}_t(s) \doteq \sum_a \pi(a \mid s) Q(s, a)$, which allows us to rewrite similarly to what was done in exercise (7.1):

$$\begin{aligned}
G_{t:t+n} - Q(S_t, A_t) &= R_{t+1} + \gamma \overline{V}_t(S_{t+1}) - \gamma\pi(A_{t+1} \mid S_{t+1}) Q(S_{t+1}, A_{t+1}) \\
&\quad + \gamma\pi\left(A_{t+1}|S_{t+1}\right) G_{t+1:t+n} - Q(S_{t+1}, A_{t+1}) \\
&= R_{t+1} + \gamma \overline{V}_t(S_{t+1}) - Q(S_{t+1}, A_{t+1}) + \gamma\pi(A_{t+1} \mid S_{t+1}) \left[G_{t+1:t+n} - Q(S_{t+1}, A_{t+1})\right] \\
&= \delta_t + \gamma\pi(A_{t+1} \mid S_{t+1}) \left[G_{t+1:t+n} - Q(S_{t+1}, A_{t+1})\right]
\end{aligned}$$

If we add back the $Q(S_t, A_t)$ and continue expanding the recursion we get

$$\begin{aligned}
G_{t:t+n} &= Q(S_t, A_t) + \delta_t + \gamma\pi(A_{t+1} \mid S_{t+1}) \cdot \\
&\qquad \delta_{t+1} + \gamma\pi(A_{t+2} \mid S_{t+2}) \left[G_{t+2:t+n} - Q(S_{t+2}, A_{t+2})\right] \\
&\qquad \text{which eventually gives us...} \\
&= Q\left(S_t, A_t\right) + \sum_{k=t}^{h} \delta_k \prod_{i=t+1}^{k} \gamma\pi\left(A_i|S_i\right)
\end{aligned}$$

where $h = \min(t + n, T) - 1$. As such, we have shown that the tree-backup return can be written as the sum of expectation-based TD errors.

Exercise 8.1

Recall that an n-step bootstrapping method, like the ones developed in Chapter 7, do consider multiple steps when updating the action-value function (for $n > 1$, that is), but only state-action pairs on the specific trajectory taken in that specific episode are updated. In contrast, planning allows the agent to update many more action values, independent of its current trajectory. As such, a planning agent can learn a lot more about an environment than a non-planning agent given a limited number of episodes to learn from.

Exercise 8.2

As the Dyna-Q+ algorithm is rewarded for taking exploratory steps by increasing the produced reward for long-untried actions during simulated experiences, it is encouraged to explore paths from the very beginning. This results in a pretty good model of the grid world with a smaller number of time-steps elapsed, which in turn also increases the cumulative reward as the agent more quickly finds the optimal path, as evident by Figure (8.4) and Figure (8.5) of the book.

Exercise 8.3

Dyna-Q+ learns the optimal path from S to G relatively faster than Dyna-Q due to its exploratory property (same reasoning as previous exercise), but then continues to explore long-untried actions although the world is entirely static until time-step 3000. Therefore, the Dyna-Q agent starts to catch up to Dyna-Q+ agent once it also learns the optimal path to G. After the change, however, Dyna-Q+ improves significantly due to its exploratory property whereas Dyna-Q sticks to the "optimal" path it found earlier.

Exercise 8.5

In step (e) of the tabular Dyna-Q algorithm shown on page 164 of the book, we set $Model(S, A)$ to R, S' and use those values during planning. This assumes that the environment is deterministic and thus always produces the same reward and new state given a state-action pair. If we change the contents of $Model(S, A)$ to contain a list of all rewards received and the corresponding new state when taking action A in S, we can randomly select a (R, S') pair from $Model(S, A)$ during planning, which would simulate a stochastic environment.
If the environment is changing, however, randomly sampling reward and new state from all previous experiences does not account well for this and as a result, an agent may perform poorly. As such, we could potentially weight our (R, S') pairs such that the probability of selecting a given pair decreases as the number of more recently experienced pairs increases (much like a moving average). We may also want to enforce a decay in the value of rewards stored in our model, such that recently experienced pairs are also more valuable (like in the Dyna-Q+ algorithm).

Implementation (*SARSA and Q-learning*)

Code and results are appended below. Although the two TD-methods are applied to two distinct problems and a direct comparison thus cannot be made, it should be noted that there appears to be a tendency for the SARSA to have a smoother learning curve than the Q-learning algorithm, which is in line with our expectations. Q-learning, however, might produce slightly more optimal results (refer to the cliff walking problem discussed in the book) at the expensive of higher inter-episode reward variance due to falling off the cliff in some occasions.

```
# Keeps track of useful statistics
stats = plotting.EpisodeStats(
    episode_lengths=np.zeros(num_episodes),
    episode_rewards=np.zeros(num_episodes))

# The policy we're following
policy = make_epsilon_greedy_policy(Q, epsilon, env.action_space.n)

def action(state):
    p = policy(state)
    return np.random.choice(np.arange(len(p)), p=p)


for i_episode in range(num_episodes):
    # Print out which episode we're on, useful for debugging.
    if (i_episode + 1) % 100 == 0:
        print("\rEpisode {}/{}.".format(i_episode + 1, num_episodes), end="")
        sys.stdout.flush()

    # Reset environment and set initial state
    St = env.reset()
    t, done = 0, False

    # Take steps until termination
    while not done:

        # Select action, take step, select action
        # S(tate), A(ction), R(eward), S(tate), A(ction)
        At = action(St)
        St_new, R, done, _ = env.step(At)
        At_new = action(St_new)

        # Update running stats
        stats.episode_lengths[i_episode] = t
        stats.episode_rewards[i_episode] += R

        # TD incremental update
        TD_target = R + gamma * Q[St_new][At_new]
        Q[St][At] = Q[St][At] + alpha * (TD_target - Q[St][At])

        # Update current state-action pair
        St = St_new
        At = At_new
        t += 1
```

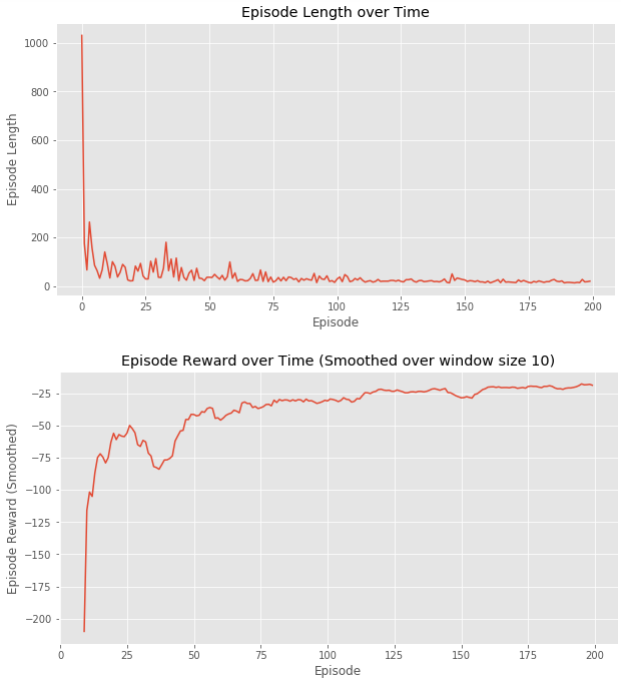

Figure 1: Windy grid-world, SARSA.

```python
# Keeps track of useful statistics
stats = plotting.EpisodeStats(
    episode_lengths=np.zeros(num_episodes),
    episode_rewards=np.zeros(num_episodes))

# The policy we're following
policy = make_epsilon_greedy_policy(Q, epsilon, env.action_space.n)

def action(state):
    p = policy(state)
    return np.random.choice(np.arange(len(p)), p=p)


for i_episode in range(num_episodes):
    # Print out which episode we're on, useful for debugging.
    if (i_episode + 1) % 100 == 0:
        print("\rEpisode {}/{}.".format(i_episode + 1, num_episodes), end="")
        sys.stdout.flush()

    # Reset environment and set initial state
    St = env.reset()
    t, done = 0, False

    # Take steps until termination
    while not done:

        # Select action, take step, select action
        # S(tate), A(ction), R(eward), S(tate), A(ction)
        At = action(St)
        St_new, R, done, _ = env.step(At)
        At_new = np.argmax(Q[St_new])

        # Update running stats
        stats.episode_lengths[i_episode] = t
        stats.episode_rewards[i_episode] += R

        # TD incremental update
        At_new = np.argmax(Q[St_new])
        TD_target = R + gamma * Q[St_new][At_new]
        Q[St][At] = Q[St][At] + alpha * (TD_target - Q[St][At])

        # Update current state
        St = St_new
        t += 1
```

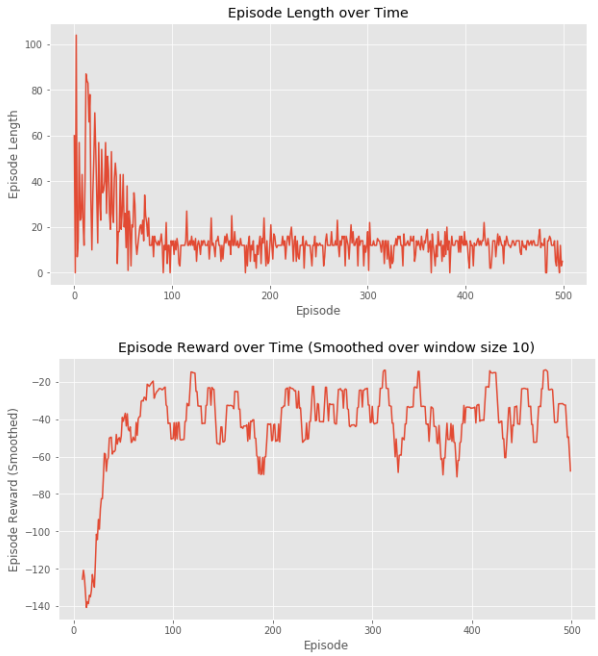

Figure 2: Cliff walking, Q-learning.