Lecture 05: Temporal-Difference Learning

Oliver Wallscheid
Temporal-difference (TD) learning combines the previous ideas introduced in DP and MC:

- From Monte Carlo (MC) methods: Learns directly from experience.
- From dynamic programming (DP): Updates estimates based on other learned estimates (bootstrap).

Hence, TD characteristics are:

- Allows model-free prediction and control in unknown MDPs.
- Updates policy evaluation and improvement in an online fashion (i.e., not per episode) by bootstrapping.
- Still assumes finite MDP problems (or problems close to that).
Table of Contents

1. Temporal-Difference Prediction
2. Temporal-Difference On-Policy Control: Sarsa
3. Temporal-Difference Off-Policy Control: $Q$-Learning
4. Expected Sarsa
5. Maximization Bias and Double Learning
General TD Prediction Updates

Recap the every-visit MC update rule (4.3) for non-stationary problems:

\[
\hat{v}(x_k) \leftarrow \hat{v}(x_k) + \alpha [g_k - \hat{v}(x_k)]. 
\]  

(5.1)

- \( \alpha \in \{\mathbb{R} | 0 < \alpha < 1\} \) is the forgetting factor / step size.
- \( g_k \) is the target of the incremental update rule.
- To execute the update (5.1) one has to wait until the episode’s termination since only then \( g_k \) is available (MC requirement).

One-step TD / TD(0) update

\[
\hat{v}(x_k) \leftarrow \hat{v}(x_k) + \alpha [r_{k+1} + \gamma \hat{v}(x_{k+1}) - \hat{v}(x_k)]. 
\]  

(5.2)

- Here, the TD target is \( r_{k+1} + \gamma \hat{v}(x_{k+1}) \).
- TD is bootstrapping: estimate \( \hat{v}(x_k) \) based on \( \hat{v}(x_{k+1}) \).
- Delay time of one step and no need to wait until the episode’s end.
Algorithmic Implementation: TD-Based Prediction

**input:** a policy $\pi$ to be evaluated

**output:** estimate of $v_\pi$ (i.e., value estimates for all states $x \in \mathcal{X}$)

**init:** $\hat{v}(x) \forall x \in \mathcal{X}$ arbitrary except $v_0(x) = 0$ if $x$ is terminal

**for** $j = 1, \ldots, J$ **episodes** **do**

  Initialize $x_0$;

  **for** $k = 0, 1, 2 \ldots$ **time steps** **do**

    $u_k \leftarrow$ apply action from $\pi(x_k)$;

    Observe $x_{k+1}$ and $r_{k+1}$;

    $\hat{v}(x_k) \leftarrow \hat{v}(x_k) + \alpha [r_{k+1} + \gamma \hat{v}(x_{k+1}) - \hat{v}(x_k)]$;

    Exit loop if $x_{k+1}$ is terminal;

**Algo. 5.1:** Tabular TD(0) prediction

- Note that the algorithm can be directly adapted to action-value prediction as it will be used for the later TD-based control approaches.
TD Error

Fig. 5.1: Back up diagram for TD(0)

TD as well as MC use sample updates.

Looking ahead to a sample successor state including its value and the reward along the way to compute a backed up value estimate.

The **TD error** is:

\[ \delta_k = r_{k+1} + \gamma \hat{v}(x_{k+1}) - \hat{v}(x_k). \]  

\[ (5.3) \]

\[ \delta_k \] is available at time step \( k + 1 \).

Iteratively \( \delta_k \) converges towards zero.
TD Error and its Relation to the MC Error

Let’s assume that the TD(0) estimate $\hat{v}(x)$ is not changing over one episode as it would be for MC prediction:

$$g_k - \hat{v}(x_k) = r_{k+1} + \gamma g_{k+1} - \hat{v}(x_k) + \gamma \hat{v}(x_{k+1}) - \gamma \hat{v}(x_{k+1}),$$

MC-error

$$= \delta_k + \gamma (g_{k+1} - \hat{v}(x_{k+1})),$$

$$= \delta_k + \gamma \delta_{k+1} + \gamma^2 (g_{k+2} - \hat{v}(x_{k+2})),$$

$$= \delta_k + \gamma \delta_{k+1} + \gamma^2 \delta_{k+2} + \gamma^3 (g_{k+3} - \hat{v}(x_{k+3})),$$

$$= \ldots,$$

$$= \sum_{i=k}^{T-1} \gamma^{i-k} \delta_i. \tag{5.4}$$

- MC error is the discounted sum of TD errors in this simplified case.
- If $\hat{v}(x)$ is updated during one episode (as expected in TD(0)), the above identity only holds approximately.
Overview of the RL Methods Considered so far

Fig. 8.11: A slice through the space of reinforcement learning methods, highlighting the two of the most important dimensions explored in Part I of this book: the depth and width of the updates.

ranging from one-step TD updates to full-return Monte Carlo updates. Between these is a spectrum including methods based on $n$-step updates (and in Chapter 12 we will extend this to mixtures of $n$-step updates such as the $\lambda$-updates implemented by eligibility traces).

Dynamic programming methods are shown in the extreme upper-right corner of the space because they involve one-step expected updates. The lower-right corner is the extreme case of expected updates so deep that they run all the way to terminal states (or, in a continuing task, until discounting has reduced the contribution of any further rewards to a negligible level). This is the case of exhaustive search. Intermediate methods along this dimension include heuristic search and related methods that search and update up to a limited depth, perhaps selectively. There are also methods that are intermediate along the horizontal dimension. These include methods that mix expected and sample updates, as well as the possibility of methods that mix samples and distributions within a single update. The interior of the square is filled in to represent the space of all such intermediate methods.

A third dimension that we have emphasized in this book is the binary distinction between on-policy and off-policy methods. In the former case, the agent learns the value function for the policy it is currently following, whereas in the latter case it learns the

Fig. 5.2: Comparison of the RL methods considered so far with regard to the update rules (source: R. Sutton and G. Barto, Reinforcement learning: an introduction, 2018, CC BY-NC-ND 2.0)
Tab. 5.1: Exemplary driving home journey assuming a given policy and all numbers denote minutes (source: R. Sutton and G. Barto, Reinforcement learning: an introduction, 2018, CC BY-NC-ND 2.0)
Driving Home Example (2)

Figure 6.1: Changes recommended in the driving home example by Monte Carlo methods (left) and TD methods (right).

Exercise 6.2
This is an exercise to help develop your intuition about why TD methods are often more efficient than Monte Carlo methods. Consider the driving home example and how it is addressed by TD and Monte Carlo methods. Can you imagine a scenario in which a TD update would be better on average than a Monte Carlo update? Give an example scenario—a description of past experience and a current state—in which you would expect the TD update to be better. Here’s a hint: Suppose you have lots of experience driving home from work. Then you move to a new building and a new parking lot (but you still enter the highway at the same place). Now you are starting to learn predictions for the new building. Can you see why TD updates are likely to be much better, at least initially, in this case? Might the same sort of thing happen in the original scenario?

Fig. 5.3: Updates by MC (left) and TD (right) for $\alpha = 1$ (source: R. Sutton and G. Barto, Reinforcement learning: an introduction, 2018, CC BY-NC-ND 2.0)

- TD can learn before knowing the final outcome.
  - TD learns after every step.
  - MC must wait until the episode’s end.
- TD could learn without a final outcome.
  - MC is only applicable to episodic tasks.
  - TD can learn from incomplete sequences, i.e., in continuing tasks.
Let’s reuse the forest tree MDP example with *fifty-fifty policy* and discount factor $\gamma = 0.8$ plus disaster probability $\alpha = 0.2$:

*Fig. 5.4: Forest MDP with fifty-fifty-policy including state values*
Fig. 5.5: State-value estimate of forest tree MDP using TD(0) prediction over the number of episodes being evaluated (mean and standard deviation are calculated based on 2000 independent runs)
Fig. 5.6: Averaged mean of state-value estimates of forest tree MDP using TD(0) and MC over 1000 independent runs with $\hat{v}_0(x) = 0 \forall x \in \mathcal{X}$
TD(0) vs. MC Prediction Example: Forest Tree MDP (2)

\[ \sqrt{\frac{1}{N} \sum_{i=1}^{N} (v(x_i) - \hat{v}(x_i))^2} \]

\(0 \leq T \leq 200\)

\begin{align*}
&TD(\alpha_{TD} = 0.2) \\
&MC(\alpha_{MC} = 0.2) \\
&TD(\alpha_{TD} = 0.1) \\
&MC(\alpha_{MC} = 0.1) \\
&TD(\alpha_{TD} = 0.05) \\
&MC(\alpha_{MC} = 0.05)
\end{align*}

Fig. 5.7: Averaged mean of state-value estimates of forest tree MDP using TD(0) and MC over 1000 independent runs with \( \hat{v}_0(x) \approx v(x) \ \forall x \in X \)
Convergence of TD(0)

Theorem 5.1: Convergence of TD(0)

Given a finite MDP and a fixed policy \( \pi \) the state-value estimate of TD(0) converges to the true \( v_\pi \)

- in the mean for a constant but sufficiently small step-size \( \alpha \) and
- with probability 1 if the step-size holds the condition

\[
\sum_{k=1}^{\infty} \alpha_k = \infty \quad \text{and} \quad \sum_{k=1}^{\infty} \alpha_k^2 < \infty.
\] (5.5)

Above \( k \) is the sample index (i.e., how often the TD update was applied).

- In particular, \( \alpha_k = \frac{1}{k} \) meets the condition (5.5).
- Often TD(0) converges faster than MC, but there is no guarantee (see examples above and in Barto/Sutton book).
- TD(0) can be more sensitive to bad initializations \( \hat{v}_0(x) \) compared to MC.
Batch Training

- If experience $\to \infty$ both MC and TD converge $\hat{v}(x) \to v(x)$.
- But how to handle limited experience, i.e., a finite set of episodes

$$x_{1,1}, u_{1,1}, r_{2,1}, \ldots, x_{T_1,1},$$
$$x_{1,2}, u_{1,2}, r_{2,2}, \ldots, x_{T_2,2},$$
$$\vdots$$
$$x_{1,j}, u_{1,j}, r_{2,j}, \ldots, x_{T_j,j},$$
$$\vdots$$
$$x_{1,J}, u_{1,J}, r_{2,J}, \ldots, x_{T_J,J}.$$

Batch training

- Process all available episodes $j \in [1, J]$ repeatedly to MC and TD.
- If the step size $\alpha$ is sufficiently small both will converge to certain steady-state values.
6.3. Optimality of TD(0)

The learning curves shown in Figure 6.2. Note that the batch TD method was consistently better than the batch Monte Carlo method.

Fig. 6.2: Performance of TD(0) and constant-MC under batch training on the random walk task.

Under batch training, constant-MC converges to values, \( V(s) \), that are sample averages of the actual returns experienced after visiting each state \( s \). These are optimal estimates in the sense that they minimize the mean-squared error from the actual returns in the training set. In this sense it is surprising that the batch TD method was able to perform better according to the root mean-squared error measure shown in the figure to the right. How is it that batch TD was able to perform better than this optimal method? The answer is that the Monte Carlo method is optimal only in a limited way, and that TD is optimal in a way that is more relevant to predicting returns.

Example 6.4: You are the Predictor

Place yourself now in the role of the predictor of returns for an unknown Markov reward process. Suppose you observe the following eight episodes:

- \( A, 0, B, 1 \)
- \( B, 1 \)
- \( B, 1 \)
- \( B, 1 \)
- \( B, 1 \)
- \( B, 1 \)
- \( B, 1 \)
- \( B, 0 \)

This means that the first episode started in state \( A \), transitioned to \( B \) with a reward of \( 0 \), and then terminated from \( B \) with a reward of \( 0 \). The other seven episodes were even shorter, starting from \( B \) and terminating immediately. Given this batch of data, what would you say are the optimal predictions, the best values for the estimates \( \hat{V}(A) \) and \( \hat{V}(B) \)?

Everyone would probably agree that the optimal value for \( \hat{V}(B) \) is \( \frac{3}{4} \), because six out of the eight times in state \( B \) the process terminated immediately with a return of \( 1 \), and the other two times in \( B \) the process terminated immediately with a return of \( 0 \).

But what is the optimal value for the estimate \( \hat{V}(A) \) given this data? Here there are two reasonable answers. One is to observe that 100% of the times the process was in state \( A \) it traversed immediately to \( B \) (with a reward of \( 0 \)); and because we have already decided that \( \hat{V}(B) = \frac{3}{4} \), therefore \( \hat{V}(A) = \frac{3}{4} \) as well.

One way of viewing this answer is that it is based on first modeling the Markov process, in this case as shown to the right, and then computing the correct estimates given the model, which indeed in this case gives \( \hat{V}(A) = \frac{3}{4} \).

Fig. 5.8: Example environment (source: R. Sutton and G. Barto, Reinforcement learning: an introduction, 2018, CC BY-NC-ND 2.0)

Tab. 5.2: Example state-reward sequences for Fig. 5.8
First, recap MC and TD(0) update rules:

\[
\text{MC: } \hat{v}(x_k) \leftarrow \hat{v}(x_k) + \alpha [g_k - \hat{v}(x_k)], \\
\text{TD: } \hat{v}(x_k) \leftarrow \hat{v}(x_k) + \alpha [r_{k+1} + \gamma \hat{v}(x_{k+1}) - \hat{v}(x_k)].
\]

Then, in steady state one receives:

\[
\text{MC: } 0 = \alpha [g_k - \hat{v}(x_k)] = g_k - \hat{v}(x_k), \\
\text{TD: } 0 = \alpha [r_{k+1} + \gamma \hat{v}(x_{k+1}) - \hat{v}(x_k)] = r_{k+1} + \gamma \hat{v}(x_{k+1}) - \hat{v}(x_k).
\]

Considering a batch learning sweep over \( j = 1, \ldots, J \) episodes:

\[
\text{MC: } 0 = \sum_{j=1}^{J} g_{k,j} - \hat{v}(x_{k,j}), \\
\text{TD: } 0 = \sum_{j=1}^{J} r_{k+1,j} + \gamma \hat{v}(x_{k+1,j}) - \hat{v}(x_{k,j}).
\]
Apply the previous equations first to state B. Since B is a terminal state, 
\[ \hat{v}(x_{k+1}) = 0 \] and \[ g_{k,j} = r_{k+1,j} \] apply, i.e., the MC and TD updates are identical for B:

\[
\begin{align*}
\text{MC}\big|_{x=B} : \quad 0 &= \sum_{j=1}^{J} g_{k,j} - \hat{v}(x_{k,j}) \quad \iff \quad \hat{v}(B) = \frac{1}{J} \sum_{j=1}^{J} g_{k,j}, \\
\text{TD}\big|_{x=B} : \quad 0 &= \sum_{j=1}^{J} r_{k+1,j} - \hat{v}(x_{k,j}) \quad \iff \quad \hat{v}(B) = \frac{1}{J} \sum_{j=1}^{J} g_{k,j}.
\end{align*}
\]

This is the average return of the available episodes from Tab. 5.2, i.e., 
\[ 6 \times 1 \text{ and } 2 \times 0 : \]

\[ \hat{v}(B)|_{\text{MC}} = \hat{v}(B)|_{\text{TD}} = \frac{6}{8} = 0.75. \quad (5.6) \]
Batch Training: AB-Example (4)

Now consider state A assuming the steady state of batch learning process:

- The instantaneous reward is always $r = 0$.
- The TD bootstrap estimate of B is $\hat{v}(x_{k+1,j}) = \hat{v}(B) = \frac{3}{4}$.

\[ \text{MC: } 0 = \sum_{j=1}^{J} g_{k,j} - \hat{v}(x_{k,j}) = \sum_{j=1}^{J} g_{k,j} - \hat{v}(A), \]
\[ \text{TD: } 0 = \sum_{j=1}^{J} r_{k+1,j} + \gamma \hat{v}(x_{k+1,j}) - \hat{v}(x_{k,j}) = \sum_{j=1}^{J} \gamma \hat{v}(B) - \hat{v}(A). \]

Looking at Tab. 5.2 there is only one episode visiting state A, where the sample return is $g_{k,j} = 0$. Hence, it follows:

\[ \hat{v}(A)_{\text{MC}} = 0, \quad \hat{v}(A)_{\text{TD}} = \gamma \hat{v}(B) = \frac{3}{4}. \]

Where does this mismatch between the MC and TD estimates come from?
Certainty Equivalence

- MC batch learning converges to the least squares fit of the sampled returns:
  \[
  \sum_{j=1}^{J} \sum_{k=1}^{T_j} (g_{k,j} - \hat{v}(x_{k,j}))^2.
  \]  
  \[\text{(5.7)}\]

- TD batch learning converges to the maximum likelihood estimate such that \(\langle \mathcal{X}, \mathcal{U}, \hat{P}, \hat{R}, \gamma \rangle\) explains the data with highest probability:
  \[
  \hat{p}_{xx'}^u = \frac{1}{n(x, u)} \sum_{j=1}^{J} \sum_{k=1}^{T_j} 1(X_{k+1} = x' | X_k = x, U_k = u),
  \]
  \[\text{(5.8)}\]
  \[
  \hat{R}^u_x = \frac{1}{n(x, u)} \sum_{j=1}^{J} \sum_{k=1}^{T_j} 1(X_k = x | U_k = u)r_{k+1,j}.
  \]

- Here, TD assumes a MDP problem structure and is absolutely certain that its model estimate describes the real world perfectly (so-called certainty equivalence).
Table of Contents

1. Temporal-Difference Prediction

2. Temporal-Difference On-Policy Control: Sarsa

3. Temporal-Difference Off-Policy Control: $Q$-Learning

4. Expected Sarsa

5. Maximization Bias and Double Learning
Applying Generalized Policy Iteration (GPI) to TD Control

GPI concept is directly applied to the TD framework using action values:

$$\pi_0 \rightarrow \hat{q}_{\pi_0} \rightarrow \pi_1 \rightarrow \hat{q}_{\pi_1} \rightarrow \cdots \pi^* \rightarrow \hat{q}_{\pi^*}. \quad (5.9)$$

One-step TD / TD(0) action-value update (Sarsa)

The TD(0) action-value update is:

$$\hat{q}(x_k, u_k) \leftarrow \hat{q}(x_k, u_k) + \alpha [r_{k+1} + \gamma \hat{q}(x_{k+1}, u_{k+1}) - \hat{q}(x_k, u_k)]. \quad (5.10)$$

Sarsa: state, action, reward, (next) state, (next) action evaluation

- In contrast to MC: continuous online updates of policy evaluation and improvement.
- On-policy approach requires exploration, e.g., by an $\varepsilon$-greedy policy:

$$\pi_i(u|x) \leftarrow \begin{cases} 1 - \varepsilon + \varepsilon/|U|, & u = \tilde{u}, \\ \varepsilon/|U|, & u \neq \tilde{u}. \end{cases} \quad (5.11)$$
TD-Based On-Policy Control (Sarsa)

**parameter:** \( \varepsilon \in \mathbb{R} \mid 0 < \varepsilon << 1 \), \( \alpha \in \mathbb{R} \mid 0 < \alpha < 1 \)

**init:** \( \hat{q}(x, u) \) arbitrarily (except terminal states) \( \forall \{x \in X, u \in U\} \)

**for** \( j = 1, 2, \ldots \) **episodes** do

- Initialize \( x_0 \);
- Choose \( u_0 \) from \( x_0 \) using a soft policy (e.g., \( \varepsilon \)-greedy) derived from \( \hat{q}(x, u) \);
- \( k \leftarrow 0 \);
- repeat
  - Take action \( u_k \), observe \( r_{k+1} \) and \( x_{k+1} \);
  - Choose \( u_{k+1} \) from \( x_{k+1} \) using a soft policy derived from \( \hat{q}(x, u) \);
  - \( \hat{q}(x_k, u_k) \leftarrow \hat{q}(x_k, u_k) + \alpha [r_{k+1} + \gamma \hat{q}(x_{k+1}, u_{k+1}) - \hat{q}(x_k, u_k)] \);
  - \( k \leftarrow k + 1 \);
- until \( x_k \) is terminal;

**Algo. 5.2:** TD-based on-policy control (Sarsa)

Convergence properties are comparable to MC-based on-policy control:

- Policy improvement theorem Theo. 4.1 holds.
- Greedy in the limit with infinite exploration (GLIE) from Def. 4.1 and step-size requirements in Theo. 5.1 apply.
Example: Sarsa with Windy Gridworld

- \( r = -1 \) per time step
- No discounting
- South-north wind according to value below each column
- \( \varepsilon = 0.1 \)
- \( \alpha = 0.5 \)
- Initial \( \hat{q}(x,u) = 0 \)

What could be a possible issue when applying MC control?

![Windy Gridworld Environment](source: R. Sutton and G. Barto, Reinforcement learning: an introduction, 2018, CC BY-NC-ND 2.0)
Fig. 5.10: Sarsa-based control with $\alpha_{Sarsa} = 0.2$ and $\varepsilon$-greedy policy with $\varepsilon = 0.2$ of forest tree MDP over the number of episodes being evaluated (mean and standard deviation are calculated based on 2000 independent runs)
Sarsa Example: Forest Tree MDP (2)

Fig. 5.11: Sarsa-based control with $\alpha_{Sarsa} = 0.1$ and $\varepsilon$-greedy policy with $\varepsilon = 0.2$ of forest tree MDP over the number of episodes being evaluated (mean and standard deviation are calculated based on 2000 independent runs)
Fig. 5.12: Sarsa-based control with $\alpha_{Sarsa} = 0.05$ and $\varepsilon$-greedy policy with $\varepsilon = 0.2$ of forest tree MDP over the number of episodes being evaluated (mean and standard deviation are calculated based on 2000 independent runs)
Fig. 5.13: Sarsa-based control with adaptive $\alpha_{Sarsa} = \frac{1}{\sqrt{j}}$ ($j =$episode) and $\varepsilon$-greedy policy with $\varepsilon = 0.2$ of forest tree MDP over the number of episodes being evaluated (mean and standard deviation are calculated based on 2000 independent runs)
<table>
<thead>
<tr>
<th>1</th>
<th>Temporal-Difference Prediction</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>Temporal-Difference On-Policy Control: Sarsa</td>
</tr>
<tr>
<td>3</td>
<td>Temporal-Difference Off-Policy Control: $Q$-Learning</td>
</tr>
<tr>
<td>4</td>
<td>Expected Sarsa</td>
</tr>
<tr>
<td>5</td>
<td>Maximization Bias and Double Learning</td>
</tr>
</tbody>
</table>
$Q$-Learning Approach

Similar to Sarsa updates, but $Q$-learning directly estimates $q^*$:

$Q$-learning action-value update

The $Q$-learning action-value update is:

$$\hat{q}(x_k, u_k) \leftarrow \hat{q}(x_k, u_k) + \alpha \left[ r_{k+1} + \gamma \max_u \hat{q}(x_{k+1}, u) - \hat{q}(x_k, u_k) \right]. \quad (5.12)$$

This is an off-policy update, since the optimal action-value function is updated independent of a given behavior policy.

Requirement for $Q$-learning control:

- Coverage: behavior policy $b$ has nonzero probability of selecting actions that might be taken by the target policy $\pi$.
- Consequence: behavior policy $b$ is soft (e.g., $\varepsilon$-soft).
- Step-size requirements (5.5) regarding $\alpha$ apply.
TD-Based Off-Policy Control (\(Q\)-Learning)

**Parameter:** \(\varepsilon \in \mathbb{R} \mid 0 < \varepsilon << 1\), \(\alpha \in \mathbb{R} \mid 0 < \alpha < 1\)

**Init:** \(\hat{q}(x, u)\) arbitrarily (except terminal states) \(\forall \{x \in X, u \in U\}\)

**for** \(j = 1, 2, \ldots \) episodes **do**

- Initialize \(x_0\);
- \(k \leftarrow 0\);

**repeat**

- Choose \(u_k\) from \(x_k\) using a soft policy derived from \(\hat{q}(x, u)\);
- Take action \(u_k\), observe \(r_{k+1}\) and \(x_{k+1}\);
- \(\hat{q}(x_k, u_k) \leftarrow \hat{q}(x_k, u_k) + \alpha [r_{k+1} + \gamma \max_u \hat{q}(x_{k+1}, u) - \hat{q}(x_k, u_k)]\);
- \(k \leftarrow k + 1\);

**until** \(x_k\) is terminal;

**Algo. 5.3:** TD-based off-policy control (\(Q\)-learning)

- Even simpler compared to Sarsa implementation in Algo. 5.2.
- As discussed with MC-based off-policy control: avoidance of the exploration-optimality trade-off for on-policy methods.
- No importance sampling required as for off-policy MC-based control.
Q-Learning Control Example: Cliff Walking

What is the backup diagram for Q-learning? The rule (6.8) updates a state–action pair, so the top node, the root of the update, must be a small, filled action node. The update is also from action nodes, maximizing over all those actions possible in the next state. Thus the bottom nodes of the backup diagram should be all these action nodes. Finally, remember that we indicate taking the maximum of these "next action" nodes with an arc across them (Figure 3.4-right). Can you guess now what the diagram is? If so, please do make a guess before turning to the answer in Figure 6.4 on page 134.

Example 6.6: Cliff Walking

This gridworld example compares Sarsa and Q-learning, highlighting the difference between on-policy (Sarsa) and off-policy (Q-learning) methods.

- $r = -1$ per time step
- Large penalty if you fall off the cliff
- No discounting
- $\epsilon = 0.1$

Why is Sarsa better in this example?

And what policy’s performance is shown here in particular?

Fig. 5.14: Cliff walking environment (source: R. Sutton and G. Barto, Reinforcement learning: an introduction, 2018, CC BY-NC-ND 2.0)
Table of Contents

1 Temporal-Difference Prediction
2 Temporal-Difference On-Policy Control: Sarsa
3 Temporal-Difference Off-Policy Control: $Q$-Learning
4 Expected Sarsa
5 Maximization Bias and Double Learning
Expected Sarsa: A Hybrid Approach

Expected Sarsa action-value update

The expected Sarsa action-value update is:

\[
\hat{q}(x_k, u_k) \leftarrow \hat{q}(x_k, u_k) + \alpha [r_{k+1} + \gamma \mathbb{E}_\pi [\hat{q}(x_{k+1}, u_{k+1})|x_{k+1}] - \hat{q}(x_k, u_k)],
\]

\[
\leftarrow \hat{q}(x_k, u_k) + \alpha \left[ r_{k+1} + \gamma \sum_u \pi(u|x_{k+1})\hat{q}(x_{k+1}, u) - \hat{q}(x_k, u_k) \right].
\]

(5.13)

This is an off or on-policy update, depending on whether the action leading to \(x_{k+1}\) was taken from the target or a varying behavior policy.

- Moves deterministically in the same direction as Sarsa moves in expectation (accordingly, it is called expected Sarsa).
- Computationally more complex than Sarsa but reduces variance due to random selection of \(u_{k+1}\).
- If \(\pi\) is greedy and \(b\) is an exploratory behavior policy, then expected Sarsa is exactly \(Q\)-learning (generalization).
Commonalities:

- Use sample updates based on a one step look ahead evaluation.
- Improve estimates based on other estimates (bootstrap).

Distinctions:

- Sarsa updates based on specific state-action transitions.
- $Q$-learning updates the optimal policy estimate $q^*$
- Expected Sarsa updates considering the expected transitions.
Fig. 5.18: All algorithms used an $\varepsilon$-greedy policy with $\varepsilon = 0.1$. Asymptotic performance is an average over 100,000 episodes whereas interim performance is an average over the first 100 episodes. These data are averages of over 50,000 and 10 runs for the interim and asymptotic cases respectively (source: R. Sutton and G. Barto, Reinforcement learning: an introduction, 2018, CC BY-NC-ND 2.0)
Table of Contents

1 Temporal-Difference Prediction

2 Temporal-Difference On-Policy Control: Sarsa

3 Temporal-Difference Off-Policy Control: $Q$-Learning

4 Expected Sarsa

5 Maximization Bias and Double Learning
Maximization Bias

All control algorithms discussed so far involve maximization operations:

- Q-learning: target policy is greedy and directly uses $\max$ operator for action-value updates.
- Sarsa: typically uses an $\epsilon$-greedy framework, which also involves $\max$ updates during policy improvement.

This can lead to a significant positive bias:

- Maximization over sampled values is used implicitly as an estimate of the maximum value.
- This issue is called maximization bias.

Small example:

- Consider a single state $x$ with multiple possible actions $u$.
- The true action values are all $q(x, u) = 0$.
- The sampled estimates $\hat{q}(x, u)$ are uncertain, i.e., randomly distributed. Some samples are above and below zero.
- Consequence: The maximum of the estimate is positive.
Double Learning Approach

Split the learning process:

▶ Divide sampled experience into two sets.
▶ Use sets to estimate independent estimates \( \hat{q}_1(x, u) \) and \( \hat{q}_2(x, u) \).

Assign specific tasks to each estimate:

▶ Estimate the maximizing action:

\[
    u^* = \arg \max_u \hat{q}_1(x, u). \tag{5.14}
\]

▶ Estimate corresponding action value:

\[
    q(x, u^*) \approx \hat{q}_2(x, u^*) = \hat{q}_2(x, \arg \max_u \hat{q}_1(x, u)). \tag{5.15}
\]

▶ Doubles memory requirements, but amount of computation per step remains the same.
Double $Q$-Learning Algorithm

**parameter:** $\varepsilon \in \{\mathbb{R}|0 < \varepsilon << 1\}$, $\alpha \in \{\mathbb{R}|0 < \alpha < 1\}$

**init:** $\hat{q}_1(x, u), \hat{q}_2(x, u)$ arbitrarily (except terminal states) $\forall \{x \in \mathcal{X}, u \in \mathcal{U}\}$

**for** $j = 1, 2, \ldots$ **episodes do**

  Initialize $x_0$;
  $k \leftarrow 0$;
  repeat
  Choose $u_k$ from $x_k$ using the policy $\varepsilon$-greedy based on $\hat{q}_1(x, u) + \hat{q}_2(x, u)$;
  Take action $u_k$, observe $r_{k+1}$ and $x_{k+1}$;
  if $n \sim \mathcal{N}(\mu = 0, \sigma) > 0$ then
    $\hat{q}_1(x_k, u_k) \leftarrow \hat{q}_1(x_k, u_k) +$
    $\alpha [r_{k+1} + \gamma \hat{q}_2(x_{k+1}, \text{arg max}_u \hat{q}_1(x_{k+1}, u)) - \hat{q}_1(x_k, u_k)];$
  else
    $\hat{q}_2(x_k, u_k) \leftarrow \hat{q}_2(x_k, u_k) +$
    $\alpha [r_{k+1} + \gamma \hat{q}_1(x_{k+1}, \text{arg max}_u \hat{q}_2(x_{k+1}, u)) - \hat{q}_2(x_k, u_k)];$
  $k \leftarrow k + 1;$
  until $x_k$ is terminal;

**Algo. 5.4:** TD-based off-policy control with double learning
Maximization Bias Example

Fig. 5.19: Comparison of $Q$-learning and double $Q$-learning on a simple episodic MDP. $Q$-learning initially learns to take the left action much more often than the right action, and always takes it significantly more often than the 5% minimum probability enforced by $\varepsilon$-greedy action selection with $\varepsilon = 0.1$. In contrast, double $Q$-learning is essentially unaffected by maximization bias. These data are averaged over 10,000 runs. The initial action-value estimates were zero. (source: R. Sutton and G. Barto, Reinforcement learning: an introduction, 2018, CC BY-NC-ND 2.0)
Summary: What You’ve Learned Today

- TD unites two key characteristics from DP and MC:
  - From MC: Sample-based updates (i.e., operating in unknown MDPs).
  - From DP: Update estimates based on other estimates (bootstrapping).
- TD allows certain simplifications and improvements compared to MC:
  - Updates are available after each step and not after each episode.
  - Off-policy learning comes without importance sampling.
  - Exploits MDP formalism by maximum likelihood estimates.
  - Hence, TD prediction and control exhibit a high applicability for many problems.
- Batch training can be used when only limited experience is available, i.e., the available samples are re-processed again and again.
- Greedy policy improvements can lead to maximization biases and, therefore, slow down the learning process.
- TD requires careful tuning of learning parameters:
  - Step size $\alpha$: how to tune convergence rate vs. uncertainty / accuracy?
  - Exploration vs. exploitation: how to visit all state-action pairs?
Thanks for your attention and have a nice week!