Off-Policy Methods, N-step, Function Approximation

Milan Straka

March 4, 2024

EUROPEAN UNION
European Structural and Investment Fund
Operational Programme Research,
Development and Education

Charles University in Prague
Faculty of Mathematics and Physics
Institute of Formal and Applied Linguistics

Sarsa

A straightforward application to the temporal-difference policy evaluation is Sarsa algorithm, which after generating $S_t, A_t, R_{t+1}, S_{t+1}, A_{t+1}$ computes

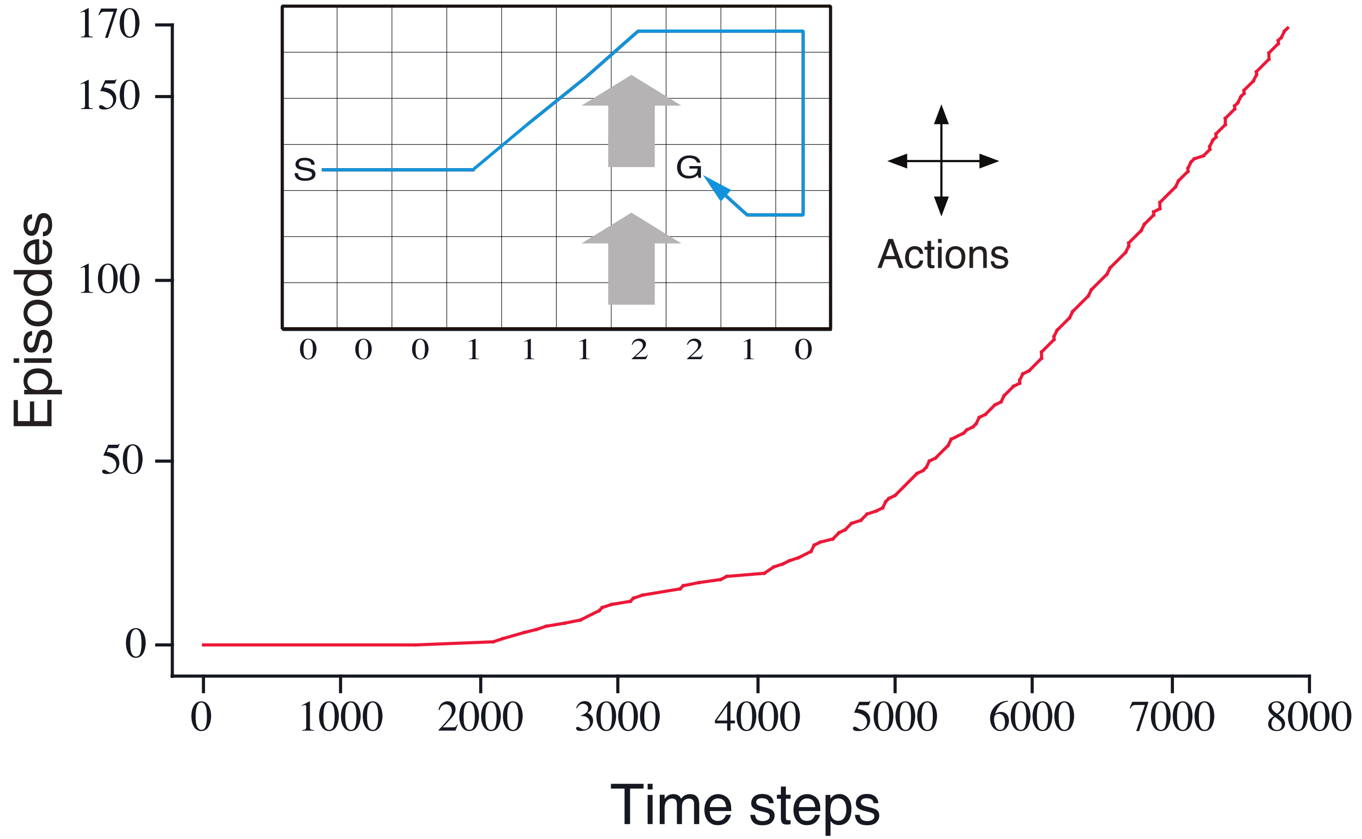
$$q(S_t, A_t) \leftarrow q(S_t, A_t) + \alpha\Big(R_{t+1} + [\neg\text{done}] \cdot \gamma q(S_{t+1}, A_{t+1}) - q(S_t, A_t)\Big).$$

Sarsa (on-policy TD control) for estimating $Q \approx q_*$

```
  Algorithm parameters: step size $\alpha \in (0,1]$, small $\varepsilon > 0$
  Initialize $Q(s,a)$, for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily except that $Q(terminal, \cdot) = 0$

Loop for each episode:
    Initialize $S$
    Choose $A$ from $S$ using policy derived from $Q$ (e.g., $\varepsilon$-greedy)
    Loop for each step of episode:
        Take action $A$, observe $R$, $S'$
        Choose $A'$ from $S'$ using policy derived from $Q$ (e.g., $\varepsilon$-greedy)
        $Q(S,A) \leftarrow Q(S,A) + \alpha\big[R + \gamma Q(S',A') - Q(S,A)\big]$
        $S \leftarrow S'$; $A \leftarrow A'$;
    until $S$ is terminal
```

Modification of Algorithm 6.4 of "Reinforcement Learning: An Introduction, Second Edition" (replacing S+ by S).

Sarsa



Example 6.5 of "Reinforcement Learning: An Introduction, Second Edition".

MC methods cannot be easily used, because an episode might not terminate if the current policy causes the agent to stay in the same state.

Q-learning

Q-learning was an important early breakthrough in reinforcement learning (Watkins, 1989).

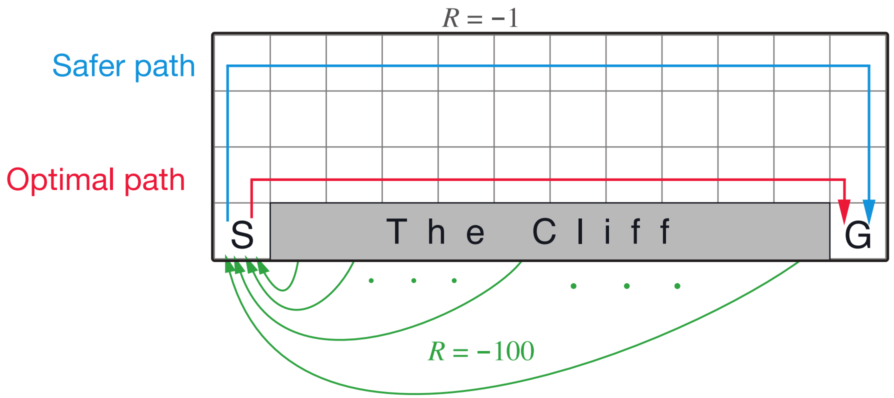
$$q(S_t, A_t) \leftarrow q(S_t, A_t) + \alpha\Big(R_{t+1} + [\neg\text{done}] \cdot \gamma \max_a q(S_{t+1}, a) - q(S_t, A_t)\Big).$$

Q-learning (off-policy TD control) for estimating $\pi \approx \pi_*$

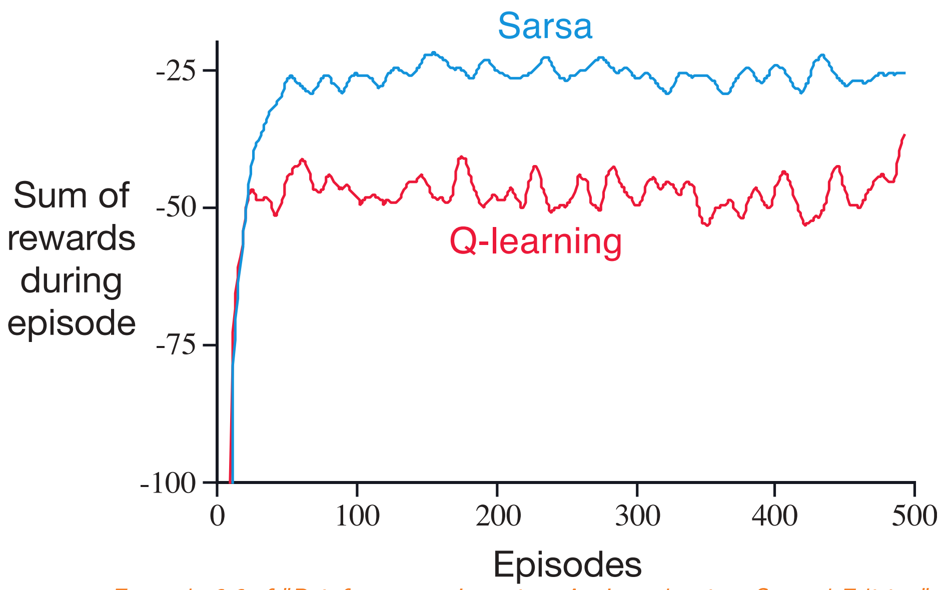
```
  Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$
  Initialize $Q(s, a)$, for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily except that $Q(\mathit{terminal}, \cdot) = 0$

Loop for each episode:
    Initialize $S$
    Loop for each step of episode:
        Choose $A$ from $S$ using policy derived from $Q$ (e.g., $\varepsilon$-greedy)
        Take action $A$, observe $R$, $S'$
        $Q(S, A) \leftarrow Q(S, A) + \alpha\big[R + \gamma \max_a Q(S', a) - Q(S, A)\big]$
        $S \leftarrow S'$
    until $S$ is terminal
```

Modification of Algorithm 6.5 of "Reinforcement Learning: An Introduction, Second Edition" (replacing S+ by S).

Q-learning versus Sarsa



Example 6.6 of "Reinforcement Learning: An Introduction, Second Edition".



Example 6.6 of "Reinforcement Learning: An Introduction, Second Edition".

On-policy and Off-policy Methods

So far, most methods were **on-policy**. The same policy was used both for generating episodes and as a target of value function.

However, while the policy for generating episodes needs to be more exploratory, the target policy should capture optimal behaviour.

Generally, we can consider two policies:

- **behaviour** policy, usually b, is used to generate behaviour and can be more exploratory;
- **target** policy, usually π, is the policy being learned (ideally the optimal one).

When the behaviour and target policies differ, we talk about **off-policy** learning.

On-policy and Off-policy Methods

The off-policy methods are usually more complicated and slower to converge, but are able to process data generated by different policy than the target one.

The advantages are:

- can compute optimal non-stochastic (non-exploratory) policies;
- more exploratory behaviour;
- ability to process *expert trajectories*.

Off-policy Prediction

Consider prediction problem for off-policy case.

In order to use episodes from b to estimate values for π, we require that every action taken by π is also taken by b, i.e.,

$$\pi(a|s) > 0 \Rightarrow b(a|s) > 0.$$

Many off-policy methods utilize **importance sampling**, a general technique for estimating expected values of one distribution given samples from another distribution.

Importance Sampling

Assume that p and q are two distributions and let x_i be N samples of p. We can then estimate $\mathbb{E}_{\mathrm{x}\sim p}\big[f(x)\big]$ as $\frac{1}{N}\sum_i f(x_i)$.

In order to estimate $\mathbb{E}_{\mathrm{x}\sim q}\big[f(x)\big]$ using the samples x_i, we need to account for different probabilities of x_i under the two distributions. It is straightforward to verify that

$$\mathbb{E}_{\mathrm{x}\sim q}\big[f(x)\big] = \mathbb{E}_{\mathrm{x}\sim p}\left[\frac{q(x)}{p(x)} f(x)\right].$$

Therefore, we can estimate

$$\mathbb{E}_{\mathrm{x}\sim q}\big[f(x)\big] \approx \frac{1}{N}\sum_i \frac{q(x_i)}{p(x_i)} f(x_i),$$

with $q(x)/p(x)$ being the **relative probability** of x under the two distributions.

Both estimates mentioned on this slide are *unbiased*.

Off-policy Prediction

Given an initial state S_t and an episode $A_t, S_{t+1}, A_{t+1}, \ldots, S_T$, the probability of this episode under a policy π is

$$\prod_{k=t}^{T-1} \pi(A_k|S_k) p(S_{k+1}|S_k, A_k).$$

Therefore, the relative probability of a trajectory under the target and behaviour policies is

$$\rho_t \stackrel{\text{def}}{=} \frac{\prod_{k=t}^{T-1} \pi(A_k|S_k) p(S_{k+1}|S_k, A_k)}{\prod_{k=t}^{T-1} b(A_k|S_k) p(S_{k+1}|S_k, A_k)} = \prod_{k=t}^{T-1} \frac{\pi(A_k|S_k)}{b(A_k|S_k)}.$$

The ρ_t is usually called the **importance sampling ratio** or *relative probability*.

Therefore, if G_t is a return of episode generated according to b, we can estimate

$$v_\pi(S_t) = \mathbb{E}_b[\rho_t G_t].$$

Off-policy Monte Carlo Prediction

Let $\mathcal{T}(s)$ be a set of times when we visited state s. Given episodes sampled according to b, we can estimate

$$v_\pi(s) = \frac{\sum_{t\in\mathcal{T}(s)} \rho_t G_t}{|\mathcal{T}(s)|}.$$

Such simple average is called **ordinary importance sampling**. It is unbiased, but can have very high variance.

An alternative is **weighted importance sampling**, where we compute weighted average as

$$v_\pi(s) = \frac{\sum_{t\in\mathcal{T}(s)} \rho_t G_t}{\sum_{t\in\mathcal{T}(s)} \rho_t}.$$

Weighted importance sampling is biased (with bias asymptotically converging to zero, i.e., a *consistent* estimate), but has smaller variance.

Off-policy Multi-armed Bandits

As a simple example, consider the 10-armed bandits from the first lecture, with single-step episodes.

Let the *behaviour policy* be a uniform policy, so the return is a reward of a randomly selected arm.

Let the *target policy* be a greedy policy always using action 3.

Assume that the first sample from the behaviour policy produced action 3 with reward R. Then

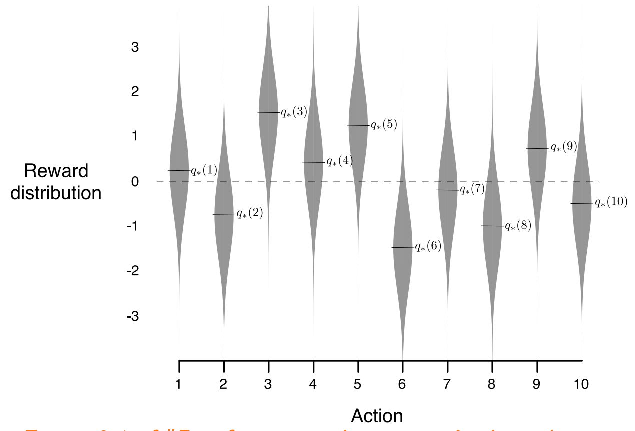


Figure 2.1 of "Reinforcement Learning: An Introduction, Second Edition".

- Ordinary importance sampling estimates the return for the target policy as

$$\frac{\pi(a)}{b(a)}R = \frac{1}{1/10}R = 10 \cdot R.$$

  The factor 10 is present because the behaviour policy returns action 3 in 10% cases.

- Weighted importance sampling estimates the return for target policy as average of rewards for action 3.

Off-policy Monte Carlo Policy Evaluation

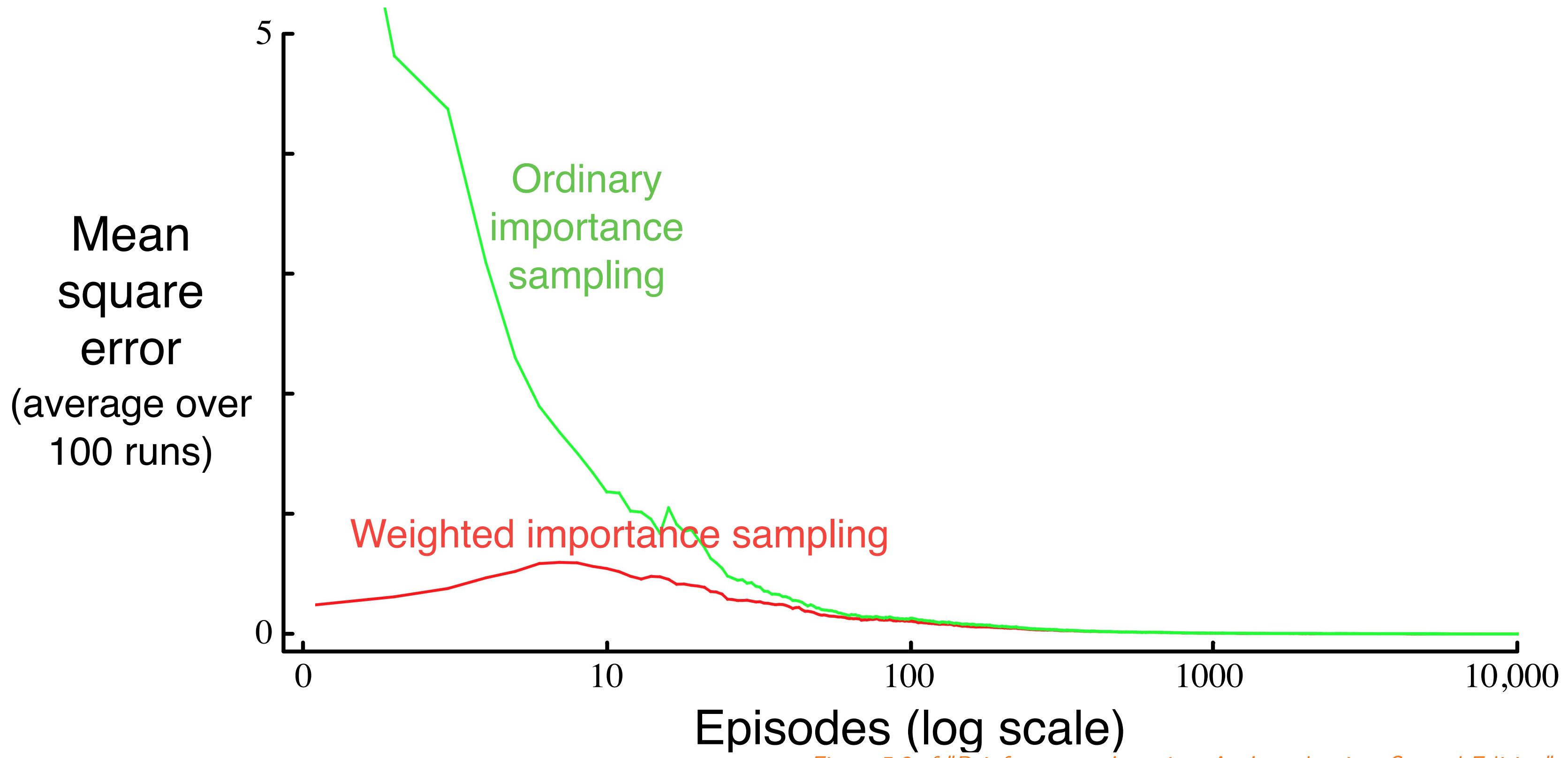


Figure 5.3 of "Reinforcement Learning: An Introduction, Second Edition".

Comparison of ordinary and weighted importance sampling on Blackjack. Given a state with dealer showing a deuce and sum of player's cards 13 with a usable ace, we estimate target policy of sticking only with a sum of 20 and 21, using uniform behaviour policy.

Off-policy Monte Carlo Policy Evaluation

We can compute weighted importance sampling similarly to the incremental implementation of Monte Carlo averaging.

Off-policy MC prediction (policy evaluation) for estimating $Q \approx q_\pi$

```
Input: an arbitrary target policy $\pi$
Initialize, for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$:
    $Q(s, a) \in \mathbb{R}$ (arbitrarily)
    $C(s, a) \leftarrow 0$

Loop forever (for each episode):
    $b \leftarrow$ any policy with coverage of $\pi$
    Generate an episode following $b$: $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$
    $G \leftarrow 0$
    $W \leftarrow 1$
    Loop for each step of episode, $t = T-1, T-2, \ldots, 0$, while $W \neq 0$:
        $G \leftarrow \gamma G + R_{t+1}$
        $C(S_t, A_t) \leftarrow C(S_t, A_t) + W$
        $Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \frac{W}{C(S_t, A_t)} [G - Q(S_t, A_t)]$
        $W \leftarrow W \frac{\pi(A_t|S_t)}{b(A_t|S_t)}$
```

Algorithm 5.6 of "Reinforcement Learning: An Introduction, Second Edition".

Off-policy Monte Carlo

Off-policy MC control, for estimating $\pi \approx \pi_*$

```
Initialize, for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$:
    $Q(s, a) \in \mathbb{R}$ (arbitrarily)
    $C(s, a) \leftarrow 0$
    $\pi(s) \leftarrow \operatorname{arg\,max}_a Q(s, a)$    (with ties broken consistently)

Loop forever (for each episode):
    $b \leftarrow$ any soft policy
    Generate an episode using $b$: $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$
    $G \leftarrow 0$
    $W \leftarrow 1$
    Loop for each step of episode, $t = T-1, T-2, \ldots, 0$:
        $G \leftarrow \gamma G + R_{t+1}$
        $C(S_t, A_t) \leftarrow C(S_t, A_t) + W$
        $Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \frac{W}{C(S_t, A_t)} [G - Q(S_t, A_t)]$
        $\pi(S_t) \leftarrow \operatorname{arg\,max}_a Q(S_t, a)$    (with ties broken consistently)
        If $A_t \neq \pi(S_t)$ then exit inner Loop (proceed to next episode)
        $W \leftarrow W \frac{1}{b(A_t|S_t)}$
```

Algorithm 5.7 of "Reinforcement Learning: An Introduction, Second Edition".

Expected Sarsa

The action A_{t+1} is a source of variance, providing correct estimate only *in expectation*.

We could improve the algorithm by considering all actions proportionally to their policy probability, obtaining Expected Sarsa algorithm:

$$\begin{aligned} q(S_t, A_t) &\leftarrow q(S_t, A_t) + \alpha\Big(R_{t+1} + [\neg\mathrm{done}] \cdot \gamma\mathbb{E}_\pi q(S_{t+1}, a) - q(S_t, A_t)\Big) \\ &\leftarrow q(S_t, A_t) + \alpha\Big(R_{t+1} + [\neg\mathrm{done}] \cdot \gamma \sum\nolimits_a \pi(a|S_{t+1}) q(S_{t+1}, a) - q(S_t, A_t)\Big). \end{aligned}$$

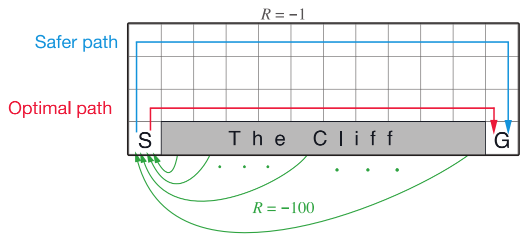
Compared to Sarsa, the expectation removes a source of variance and therefore usually performs better. However, the complexity of the algorithm increases and becomes dependent on number of actions $|\mathcal{A}|$.

Expected Sarsa as an Off-policy Algorithm

Note that Expected Sarsa is also an off-policy algorithm, allowing the behaviour policy b and target policy π to differ.

Especially, if π is a greedy policy with respect to current value function, Expected Sarsa simplifies to Q-learning.

Expected Sarsa Example



Example 6.6 of "Reinforcement Learning: An Introduction, Second Edition".

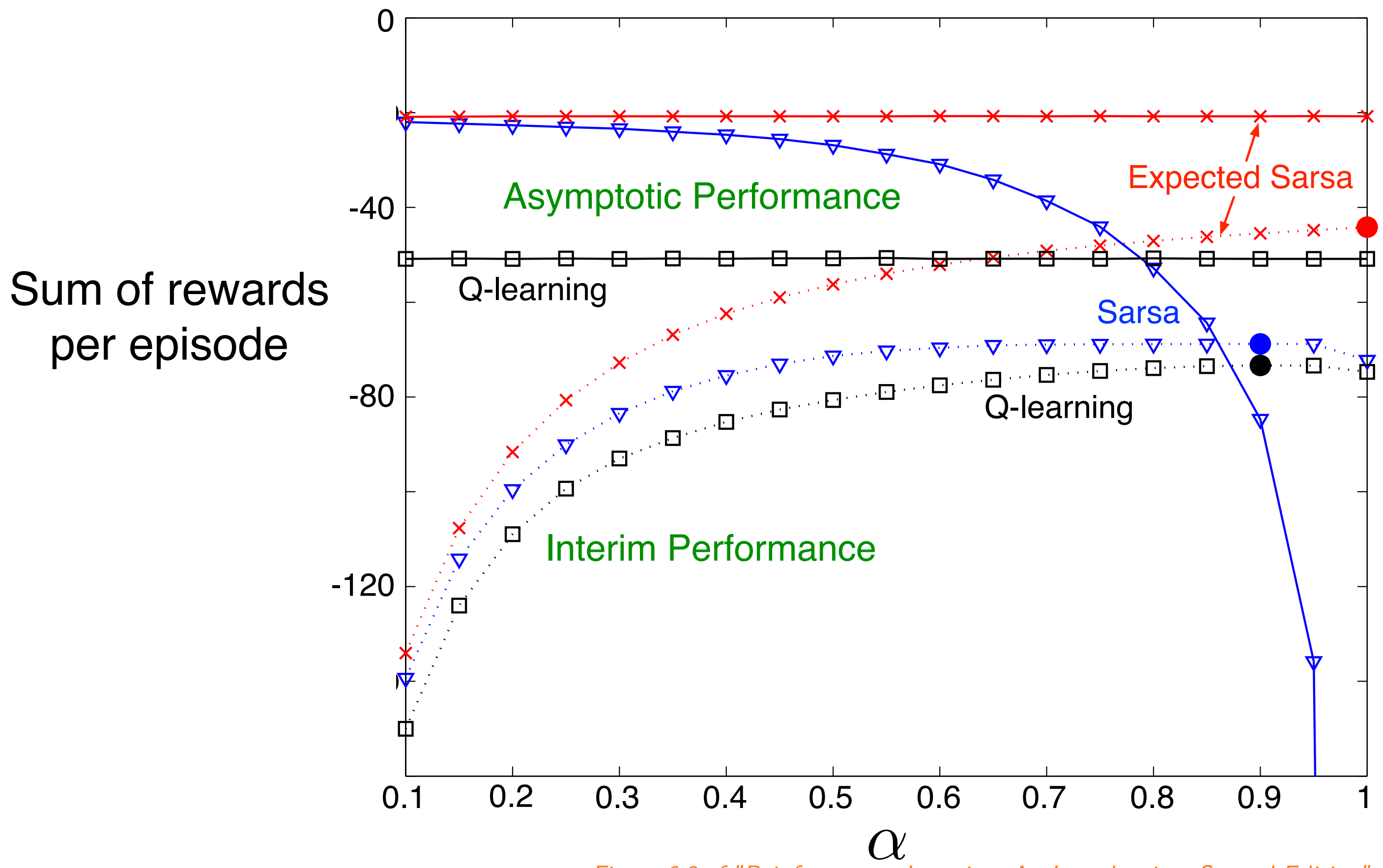


Figure 6.3 of "Reinforcement Learning: An Introduction, Second Edition".

Asymptotic performance is an average over 100k episodes (10 runs), interim performance over the first 100 episodes (50k runs); ε-greedy policy with $\varepsilon = 0.1$ is used.

Q-learning and Maximization Bias

Because behaviour policy in Q-learning is ε-greedy variant of the target policy, the same samples (up to ε-greedy) determine both the maximizing action and its value estimate.

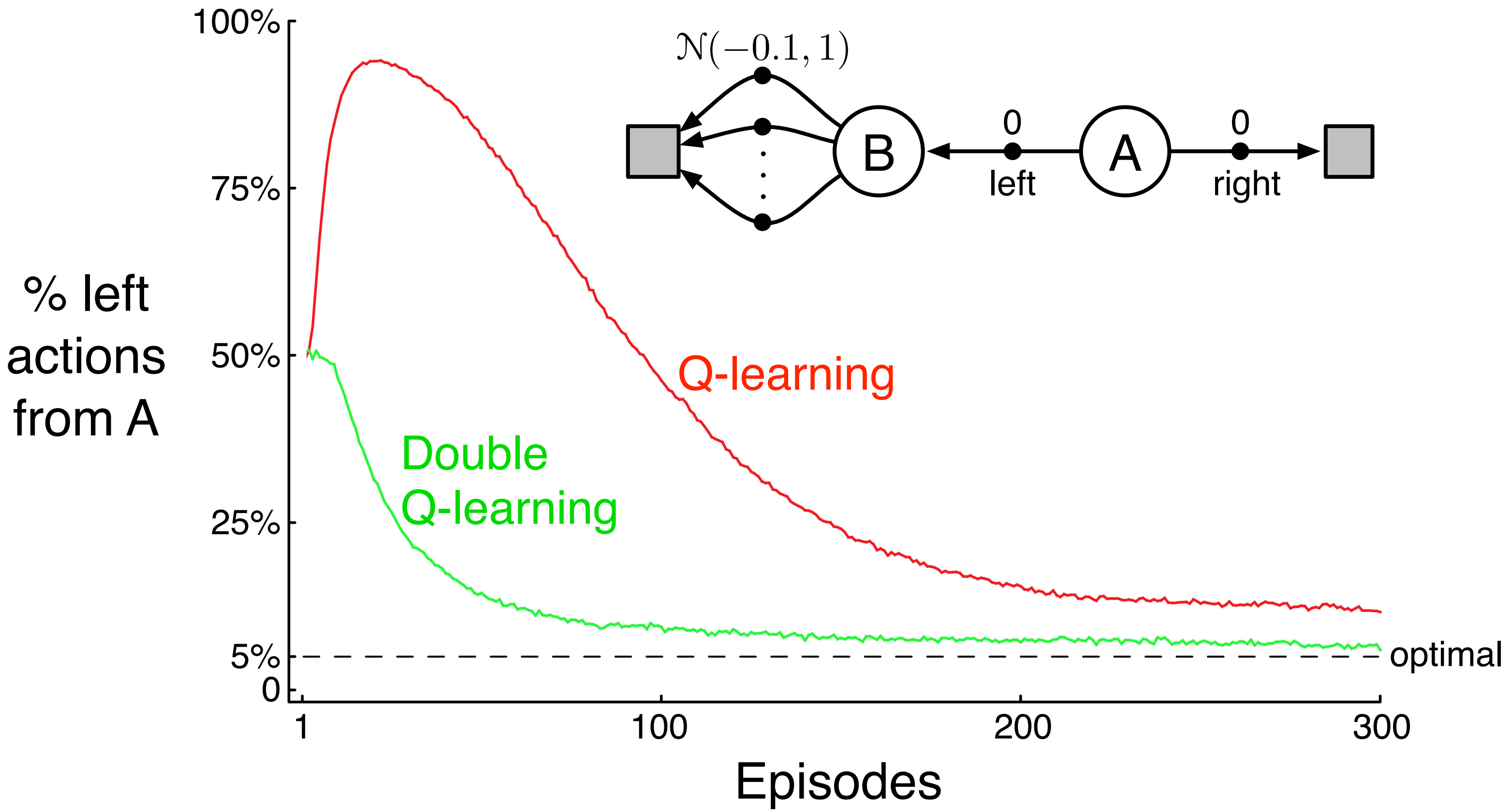


Figure 6.5 of "Reinforcement Learning: An Introduction, Second Edition".

Double Q-learning

Double Q-learning, for estimating $Q_1 \approx Q_2 \approx q_*$

```
Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$
Initialize $Q_1(s, a)$ and $Q_2(s, a)$, for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$, such that $Q(terminal, \cdot) = 0$

Loop for each episode:
    Initialize $S$
    Loop for each step of episode:
        Choose $A$ from $S$ using the policy $\varepsilon$-greedy in $Q_1 + Q_2$
        Take action $A$, observe $R$, $S'$
        With 0.5 probabilility:
            $Q_1(S, A) \leftarrow Q_1(S, A) + \alpha\Big(R + \gamma Q_2\big(S', \arg\max_a Q_1(S', a)\big) - Q_1(S, A)\Big)$
        else:
            $Q_2(S, A) \leftarrow Q_2(S, A) + \alpha\Big(R + \gamma Q_1\big(S', \arg\max_a Q_2(S', a)\big) - Q_2(S, A)\Big)$
        $S \leftarrow S'$
    until $S$ is terminal
```

Modification of Algorithm 6.7 of "Reinforcement Learning: An Introduction, Second Edition" (replacing S+ by S).

n-step Methods

Full return is

$$G_t = \sum_{k=t}^{\infty} \gamma^{k-t} R_{k+1},$$

one-step return is

$$G_{t:t+1} = R_{t+1} + [\neg \text{done}] \cdot \gamma V(S_{t+1}).$$

We can generalize both into n-step returns:

$$G_{t:t+n} \overset{\text{def}}{=} \left(\sum_{k=t}^{t+n-1} \gamma^{k-t} R_{k+1} \right) + \gamma^n V(S_{t+n}).$$

with $G_{t:t+n} \overset{\text{def}}{=} G_t$ if $t + n \geq T$ (episode length).

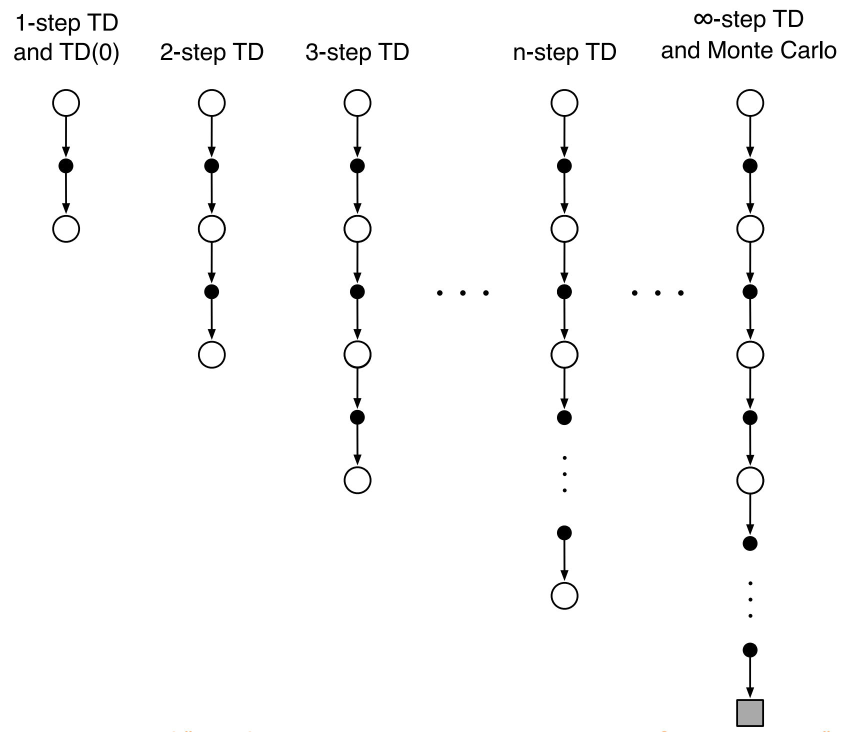


Figure 7.1 of "Reinforcement Learning: An Introduction, Second Edition".

n-step Methods

A natural update rule is

$$V(S_t) \leftarrow V(S_t) + \alpha\big(G_{t:t+n} - V(S_t)\big).$$

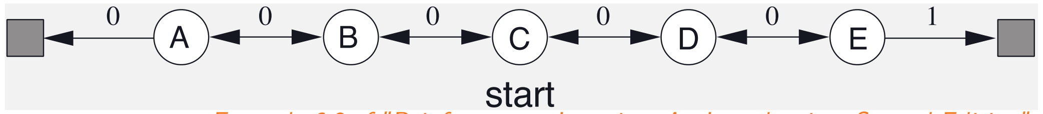
n-step TD for estimating $V \approx v_\pi$

```
Input: a policy $\pi$
Algorithm parameters: step size $\alpha \in (0,1]$, a positive integer $n$
Initialize $V(s)$ arbitrarily, for all $s \in \mathcal{S}$
All store and access operations (for $S_t$ and $R_t$) can take their index mod $n+1$

Loop for each episode:
   Initialize and store $S_0 \neq$ terminal
   $T \leftarrow \infty$
   Loop for $t = 0, 1, 2, \ldots$ :
   |  If $t < T$, then:
   |     Take an action according to $\pi(\cdot|S_t)$
   |     Observe and store the next reward as $R_{t+1}$ and the next state as $S_{t+1}$
   |     If $S_{t+1}$ is terminal, then $T \leftarrow t+1$
   |  $\tau \leftarrow t - n + 1$   ($\tau$ is the time whose state's estimate is being updated)
   |  If $\tau \geq 0$:
   |     $G \leftarrow \sum_{i=\tau+1}^{\min(\tau+n,T)} \gamma^{i-\tau-1} R_i$
   |     If $\tau + n < T$, then: $G \leftarrow G + \gamma^n V(S_{\tau+n})$          $(G_{\tau:\tau+n})$
   |     $V(S_\tau) \leftarrow V(S_\tau) + \alpha\,[G - V(S_\tau)]$
   Until $\tau = T - 1$
```

Algorithm 7.1 of "Reinforcement Learning: An Introduction, Second Edition".

n-step Methods Example

Using the random walk example, but with 19 states instead of 5,



Example 6.2 of "Reinforcement Learning: An Introduction, Second Edition".

we obtain the following comparison of different values of n:

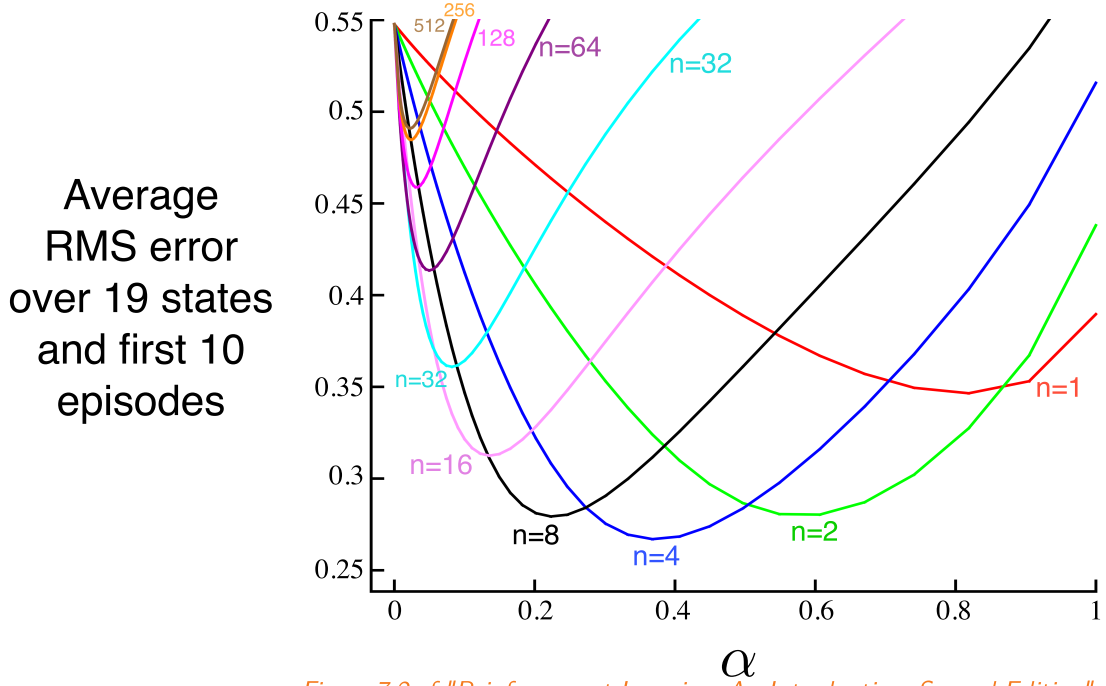


Figure 7.2 of "Reinforcement Learning: An Introduction, Second Edition".

n-step Sarsa

Defining the n-step return to utilize action-value function as

$$G_{t:t+n} \overset{\text{def}}{=} \left(\sum_{k=t}^{t+n-1} \gamma^{k-t} R_{k+1}\right) + \gamma^n Q(S_{t+n}, A_{t+n})$$

with $G_{t:t+n} \overset{\text{def}}{=} G_t$ if $t+n \geq T$, we get the following straightforward algorithm:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha\big(G_{t:t+n} - Q(S_t, A_t)\big).$$

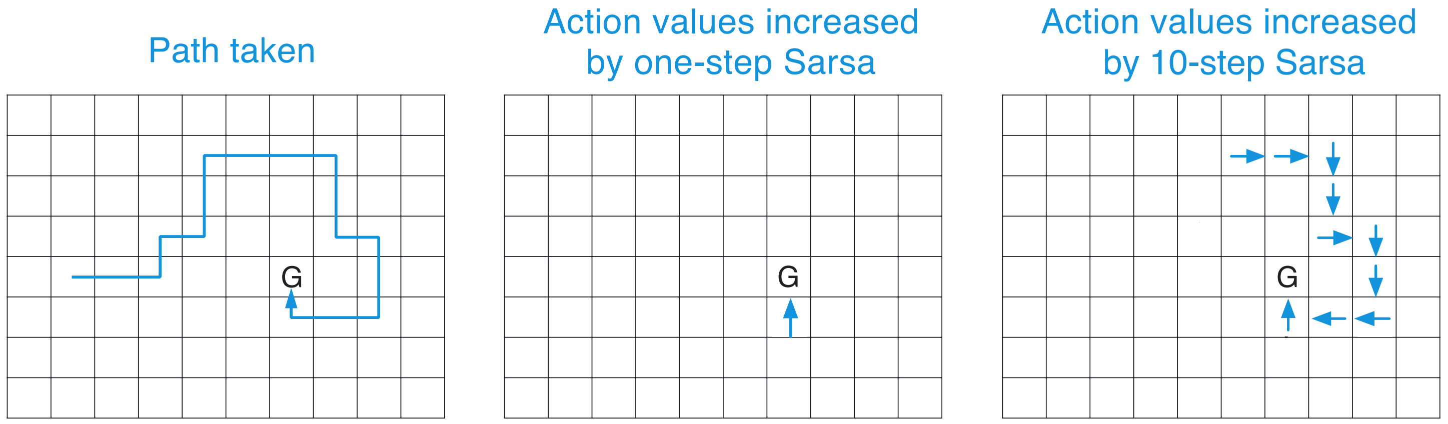

Path taken
Action values increased by one-step Sarsa
Action values increased by 10-step Sarsa
G
G
G

Figure 7.4 of "Reinforcement Learning: An Introduction, Second Edition".

n-step Sarsa Algorithm

n-step Sarsa for estimating $Q \approx q_*$ or q_π

```
Initialize $Q(s,a)$ arbitrarily, for all $s \in \mathcal{S}, a \in \mathcal{A}$
Initialize $\pi$ to be $\varepsilon$-greedy with respect to $Q$, or to a fixed given policy
Algorithm parameters: step size $\alpha \in (0,1]$, small $\varepsilon > 0$, a positive integer $n$
All store and access operations (for $S_t$, $A_t$, and $R_t$) can take their index mod $n+1$

Loop for each episode:
   Initialize and store $S_0 \neq$ terminal
   Select and store an action $A_0 \sim \pi(\cdot|S_0)$
   $T \leftarrow \infty$
   Loop for $t = 0, 1, 2, \dots$ :
   |  If $t < T$, then:
   |     Take action $A_t$
   |     Observe and store the next reward as $R_{t+1}$ and the next state as $S_{t+1}$
   |     If $S_{t+1}$ is terminal, then:
   |        $T \leftarrow t+1$
   |     else:
   |        Select and store an action $A_{t+1} \sim \pi(\cdot|S_{t+1})$
   |  $\tau \leftarrow t - n + 1$   ($\tau$ is the time whose estimate is being updated)
   |  If $\tau \geq 0$:
   |     $G \leftarrow \sum_{i=\tau+1}^{\min(\tau+n,T)} \gamma^{i-\tau-1} R_i$
   |     If $\tau + n < T$, then $G \leftarrow G + \gamma^n Q(S_{\tau+n}, A_{\tau+n})$          $(G_{\tau:\tau+n})$
   |     $Q(S_\tau, A_\tau) \leftarrow Q(S_\tau, A_\tau) + \alpha\,[G - Q(S_\tau, A_\tau)]$
   |     If $\pi$ is being learned, then ensure that $\pi(\cdot|S_\tau)$ is $\varepsilon$-greedy wrt $Q$
   Until $\tau = T - 1$
```

Algorithm 7.2 of "Reinforcement Learning: An Introduction, Second Edition".

Off-policy n-step Sarsa

Recall the relative probability of a trajectory under the target and behaviour policies, which we now generalize as

$$\rho_{t:t+n} \overset{\text{def}}{=} \prod_{k=t}^{\min(t+n,T-1)} \frac{\pi(A_k|S_k)}{b(A_k|S_k)}.$$

Then a simple off-policy n-step TD policy evaluation can be computed as

$$V(S_t) \leftarrow V(S_t) + \alpha\rho_{t:t+n-1}\big(G_{t:t+n} - V(S_t)\big).$$

Similarly, n-step Sarsa becomes

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha\rho_{\boldsymbol{t+1:t+n}}\big(G_{t:t+n} - Q(S_t, A_t)\big).$$

Off-policy n-step Sarsa

Off-policy n-step Sarsa for estimating $Q \approx q_*$ or q_π

```
Input: an arbitrary behavior policy $b$ such that $b(a|s) > 0$, for all $s \in \mathcal{S}, a \in \mathcal{A}$
Initialize $Q(s,a)$ arbitrarily, for all $s \in \mathcal{S}, a \in \mathcal{A}$
Initialize $\pi$ to be greedy with respect to $Q$, or as a fixed given policy
Algorithm parameters: step size $\alpha \in (0,1]$, a positive integer $n$
All store and access operations (for $S_t$, $A_t$, and $R_t$) can take their index mod $n+1$

Loop for each episode:
    Initialize and store $S_0 \neq$ terminal
    Select and store an action $A_0 \sim b(\cdot|S_0)$
    $T \leftarrow \infty$
    Loop for $t = 0, 1, 2, \ldots$ :
    |   If $t < T$, then:
    |       Take action $A_t$
    |       Observe and store the next reward as $R_{t+1}$ and the next state as $S_{t+1}$
    |       If $S_{t+1}$ is terminal, then:
    |           $T \leftarrow t+1$
    |       else:
    |           Select and store an action $A_{t+1} \sim b(\cdot|S_{t+1})$
    |   $\tau \leftarrow t - n + 1$   ($\tau$ is the time whose estimate is being updated)
    |   If $\tau \geq 0$:
    |       $\rho \leftarrow \prod_{i=\tau+1}^{\min(\tau+n, T-1)} \frac{\pi(A_i|S_i)}{b(A_i|S_i)}$                  $(\rho_{\tau+1:t+n})$
    |       $G \leftarrow \sum_{i=\tau+1}^{\min(\tau+n,T)} \gamma^{i-\tau-1} R_i$
    |       If $\tau + n < T$, then: $G \leftarrow G + \gamma^n Q(S_{\tau+n}, A_{\tau+n})$        $(G_{\tau:\tau+n})$
    |       $Q(S_\tau, A_\tau) \leftarrow Q(S_\tau, A_\tau) + \alpha\rho\,[G - Q(S_\tau, A_\tau)]$
    |       If $\pi$ is being learned, then ensure that $\pi(\cdot|S_\tau)$ is greedy wrt $Q$
    Until $\tau = T - 1$
```

Modified from Algorithm 7.3 of "Reinforcement Learning: An Introduction, Second Edition" by changing ρ_{τ+1:τ+n-1} to ρ_{τ+1:τ+n}.

Off-policy n-step Without Importance Sampling

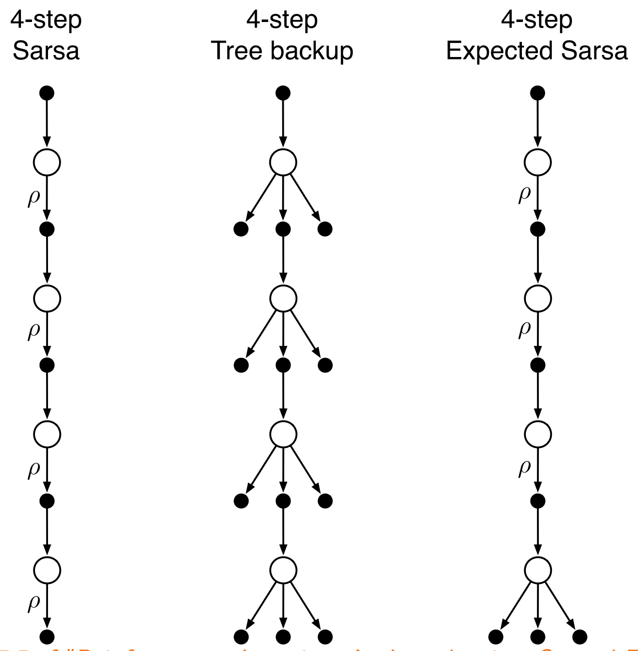


Figure 7.5 of "Reinforcement Learning: An Introduction, Second Edition".

Q-learning and Expected Sarsa can learn off-policy without importance sampling.

To generalize to n-step off-policy method, we must compute expectations over actions in each step of n-step update. However, we have not obtained a return for the non-sampled actions.

Luckily, we can estimate their values by using the current action-value function.

Off-policy n-step Without Importance Sampling

We now derive the n-step reward, starting from one-step:

$$G_{t:t+1} \overset{\text{def}}{=} R_{t+1} + [\neg\text{done}] \cdot \gamma \sum\nolimits_a \pi(a|S_{t+1})Q(S_{t+1}, a).$$

For two-step, we get:

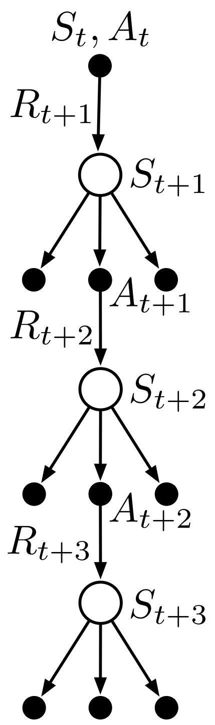
$$G_{t:t+2} \overset{\text{def}}{=} R_{t+1} + \gamma \sum\nolimits_{a \neq A_{t+1}} \pi(a|S_{t+1})Q(S_{t+1}, a) + \gamma\pi(A_{t+1}|S_{t+1})G_{t+1:t+2}.$$

Therefore, we can generalize to:

$$G_{t:t+n} \overset{\text{def}}{=} R_{t+1} + \gamma \sum\nolimits_{a \neq A_{t+1}} \pi(a|S_{t+1})Q(S_{t+1}, a) + \gamma\pi(A_{t+1}|S_{t+1})G_{t+1:t+n},$$

with $G_{t:t+n} \overset{\text{def}}{=} G_{t:T}$ if $t + n \geq T$ (episode length).

The resulting algorithm is n-step **Tree backup** and it is an off-policy n-step temporal difference method not requiring importance sampling.



the 3-step tree-backup update

Example in Section 7.5 of "Reinforcement Learning: An Introduction, Second Edition".

Off-policy n-step Without Importance Sampling

n-step Tree Backup for estimating $Q \approx q_*$ or q_π

```
Initialize $Q(s, a)$ arbitrarily, for all $s \in \mathcal{S}, a \in \mathcal{A}$
Initialize $\pi$ to be greedy with respect to $Q$, or as a fixed given policy
Algorithm parameters: step size $\alpha \in (0, 1]$, a positive integer $n$
All store and access operations can take their index mod $n + 1$

Loop for each episode:
    Initialize and store $S_0 \neq$ terminal
    Choose an action $A_0$ arbitrarily as a function of $S_0$; Store $A_0$
    $T \leftarrow \infty$
    Loop for $t = 0, 1, 2, \ldots$ :
    |   If $t < T$:
    |       Take action $A_t$; observe and store the next reward and state as $R_{t+1}, S_{t+1}$
    |       If $S_{t+1}$ is terminal:
    |           $T \leftarrow t + 1$
    |       else:
    |           Choose an action $A_{t+1}$ arbitrarily as a function of $S_{t+1}$; Store $A_{t+1}$
    |   $\tau \leftarrow t + 1 - n$    ($\tau$ is the time whose estimate is being updated)
    |   If $\tau \geq 0$:
    |       If $t + 1 \geq T$:
    |           $G \leftarrow R_T$
    |       else
    |           $G \leftarrow R_{t+1} + \gamma \sum_a \pi(a|S_{t+1}) Q(S_{t+1}, a)$
    |       Loop for $k = \min(t, T - 1)$ down through $\tau + 1$:
    |           $G \leftarrow R_k + \gamma \sum_{a \neq A_k} \pi(a|S_k) Q(S_k, a) + \gamma \pi(A_k|S_k) G$
    |       $Q(S_\tau, A_\tau) \leftarrow Q(S_\tau, A_\tau) + \alpha [G - Q(S_\tau, A_\tau)]$
    |       If $\pi$ is being learned, then ensure that $\pi(\cdot|S_\tau)$ is greedy wrt $Q$
    Until $\tau = T - 1$
```

Algorithm 7.5 of "Reinforcement Learning: An Introduction, Second Edition".

Where Are We

- Until now, we have solved the tasks by explicitly calculating expected return, either as $v(s)$ or as $q(s,a)$.
  - Finite number of states and actions.
  - We do not share information between different states or actions.
  - We use $q(s,a)$ if we do not have the environment model (a *model-free* method); if we do, it is usually better to estimate $v(s)$ and choose actions as $\arg\max_a \mathbb{E}[R + v(s')]$.
- The methods we know differ in several aspects:
  - Whether they compute return by simulating a whole episode (Monte Carlo methods), or by bootstrapping (temporal difference, i.e., $G_t \approx R_t + v(S_t)$, possibly n-step).
    - TD methods more noisy and unstable, but can learn immediately and explicitly assume Markovian property of value function.
  - Whether they estimate the value function of the same policy they use to generate the episodes (on-policy) or not (off-policy).
    - The off-policy methods are more noisy and unstable, but more flexible.

Function Approximation

We now approximate the value function v and/or the action-value function q, selecting it from a family of functions parametrized by a weight vector $\boldsymbol{w} \in \mathbb{R}^d$.

We denote the approximations as

$$\hat{v}(s; \boldsymbol{w}),$$
$$\hat{q}(s, a; \boldsymbol{w}).$$

Weights are usually shared among states. Therefore, we need to define state distribution $\mu(s)$ to obtain an objective for finding the best function approximation (if we give preference to some states, improving their estimates might worsen estimates in other states).

The state distribution $\mu(s)$ gives rise to a natural objective function called **Mean Squared Value Error**, denoted $\overline{VE}$:

$$\overline{VE}(\boldsymbol{w}) \stackrel{\text{def}}{=} \sum_{s \in \mathcal{S}} \mu(s) \big(v_\pi(s) - \hat{v}(s; \boldsymbol{w})\big)^2.$$

Function Approximation

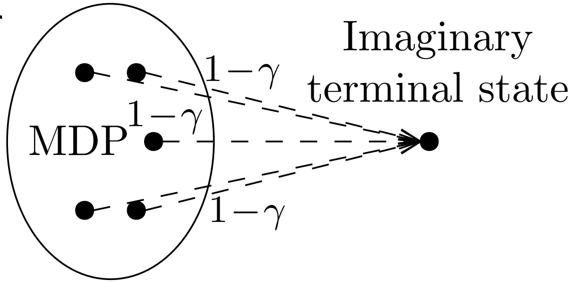
For on-policy algorithms, $\mu(s)$ is often the on-policy distribution (fraction of time spent in s).

- For **episodic tasks**, let $h(s)$ be the probability that an episodes starts in state s, and let $\eta(s)$ denote the number of time steps spent, on average, in state s in a single episode:

$$\eta(s) = h(s) + \sum\nolimits_{s'} \eta(s') \sum\nolimits_a \pi(a|s')p(s|s',a).$$

  The on-policy distribution is then obtained by normalizing: $\mu(s) \overset{\text{def}}{=} \frac{\eta(s)}{\sum_{s'} \eta(s')}$.

  If there is discounting ($\gamma < 1$), it should be treated as a form of termination, by including a factor γ to the second term of the $\eta(s)$ equation.



- For **continuing tasks**, we require $\gamma < 1$, and employ the same definition as in the episodic case.

Gradient and Semi-Gradient Methods

The functional approximation (i.e., the weight vector $\boldsymbol{w}$) is usually optimized using gradient methods, for example as

$$\begin{aligned}\boldsymbol{w}_{t+1} &\leftarrow \boldsymbol{w}_t - \tfrac{1}{2}\alpha\nabla_{\boldsymbol{w}_t}\big(v_\pi(S_t) - \hat{v}(S_t;\boldsymbol{w}_t)\big)^2\\ &\leftarrow \boldsymbol{w}_t + \alpha\big(v_\pi(S_t) - \hat{v}(S_t;\boldsymbol{w}_t)\big)\nabla_{\boldsymbol{w}_t}\hat{v}(S_t;\boldsymbol{w}_t).\end{aligned}$$

As usual, the $v_\pi(S_t)$ is estimated by a suitable sample of a return:

- in Monte Carlo methods, we use episodic return G_t,
- in temporal difference methods, we employ bootstrapping and use one-step return

$$R_{t+1} + [\neg\mathrm{done}]\cdot\gamma\hat{v}(S_{t+1};\boldsymbol{w})$$

  or an n-step return.

Monte Carlo Gradient Policy Evaluation

Gradient Monte Carlo Algorithm for Estimating $\hat{v} \approx v_\pi$

```
Input: the policy $\pi$ to be evaluated
Input: a differentiable function $\hat{v} : \mathcal{S} \times \mathbb{R}^d \rightarrow \mathbb{R}$
Algorithm parameter: step size $\alpha > 0$
Initialize value-function weights $\mathbf{w} \in \mathbb{R}^d$ arbitrarily (e.g., $\mathbf{w} = \mathbf{0}$)

Loop forever (for each episode):
    Generate an episode $S_0, A_0, R_1, S_1, A_1, \ldots, R_T, S_T$ using $\pi$
    Loop for each step of episode, $t = 0, 1, \ldots, T-1$:
        $\mathbf{w} \leftarrow \mathbf{w} + \alpha\big[G_t - \hat{v}(S_t,\mathbf{w})\big]\nabla\hat{v}(S_t,\mathbf{w})$
```

Algorithm 9.3 of "Reinforcement Learning: An Introduction, Second Edition".

Linear Methods

A simple special case of function approximation are linear methods, where

$$\hat{v}\big(\boldsymbol{x}(s); \boldsymbol{w}\big) \stackrel{\text{def}}{=} \boldsymbol{x}(s)^T \boldsymbol{w} = \sum x(s)_i w_i.$$

The $\boldsymbol{x}(s)$ is a representation of state s, which is a vector of the same size as $\boldsymbol{w}$. It is sometimes called a *feature vector*.

The SGD update rule then becomes

$$\boldsymbol{w}_{t+1} \leftarrow \boldsymbol{w}_t + \alpha\big(v_\pi(S_t) - \hat{v}(\boldsymbol{x}(S_t); \boldsymbol{w}_t)\big)\boldsymbol{x}(S_t).$$

This rule is the same as in the tabular methods if $\boldsymbol{x}(s)$ is the one-hot representation of the state s.

State Aggregation

Simple way of generating a feature vector is **state aggregation**, where several neighboring states are grouped together.

For example, consider a 1000-state random walk, where transitions lead uniformly randomly to any of 100 neighboring states on the left or on the right. Using state aggregation, we can partition the 1000 states into 10 groups of 100 states. Monte Carlo policy evaluation then computes the following:

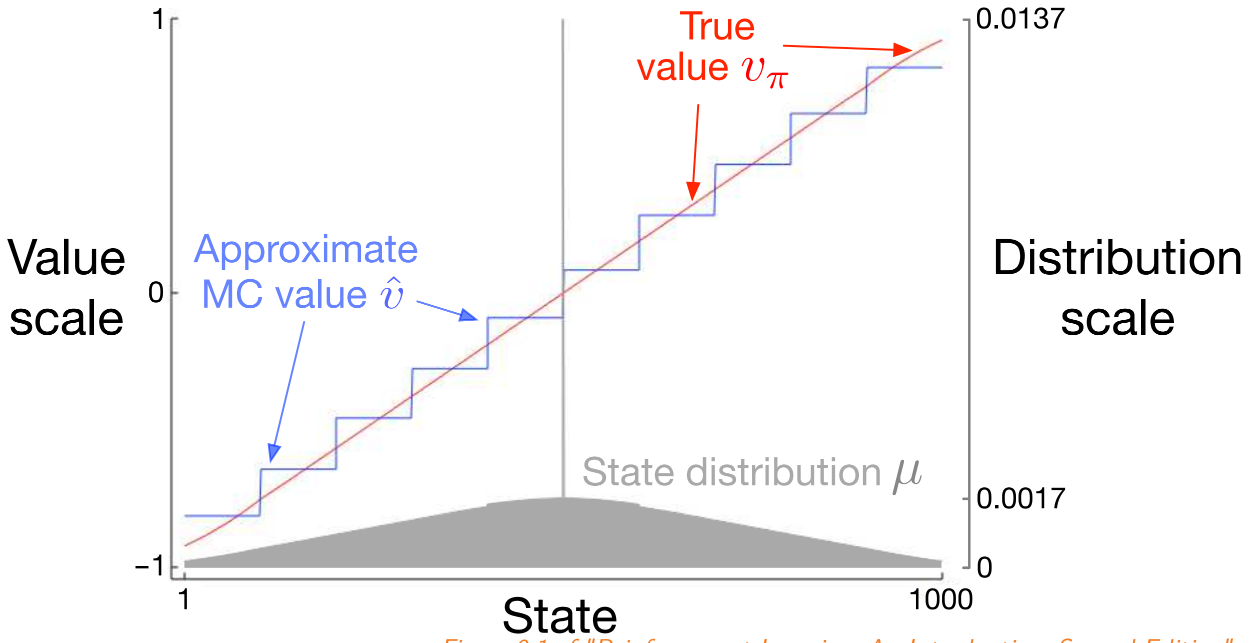


Figure 9.1 of "Reinforcement Learning: An Introduction, Second Edition".

Feature Construction for Linear Methods

Many methods for construction features for linear methods have been developed in the past:

- polynomials,
- Fourier bases,
- radial basis functions,
- tile coding,
- ...

But of course, nowadays we use deep neural networks, which construct a suitable feature vector automatically as a latent variable (the last hidden layer).

Tile Coding

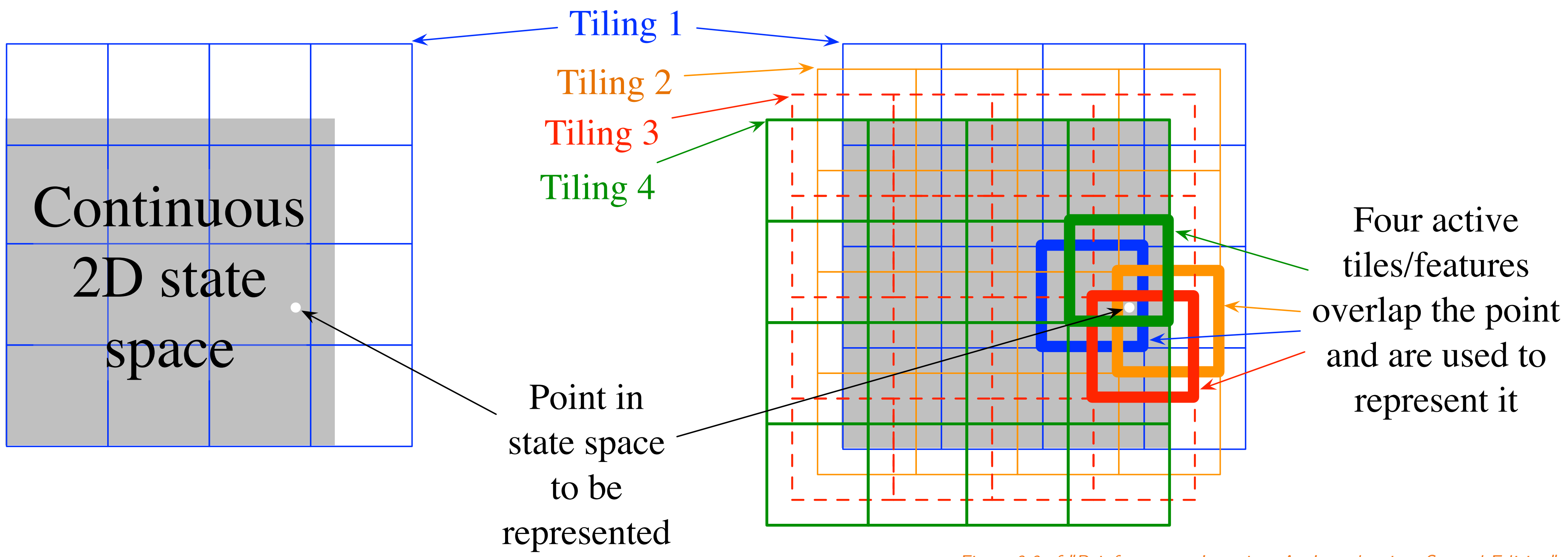

Tiling 1
Tiling 2
Tiling 3
Tiling 4
Continuous 2D state space
Point in state space to be represented
Four active tiles/features overlap the point and are used to represent it

Figure 9.9 of "Reinforcement Learning: An Introduction, Second Edition".

If t overlapping tiles are used, the learning rate is usually normalized as α/t.

Tile Coding

For example, on the 1000-state random walk example, the performance of the tile coding surpasses state aggregation:

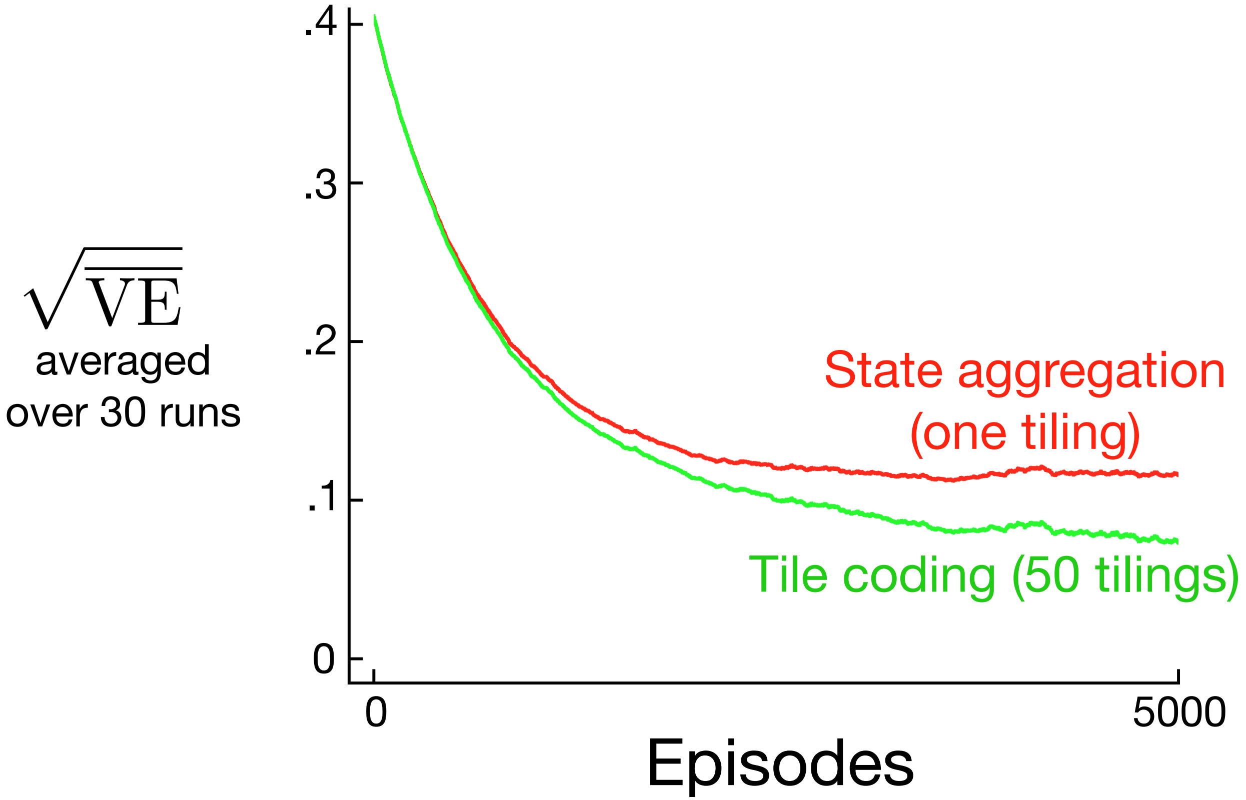


Figure 9.10 of "Reinforcement Learning: An Introduction, Second Edition".

Each tile covers 200 states, and when multiple tiles are used, they are offset by 4 states.

Asymmetrical Tile Coding

In higher dimensions, the tiles should have asymmetrical offsets, with a sequence of $(1, 3, 5, \ldots, 2d-1)$ proposed as a good choice.

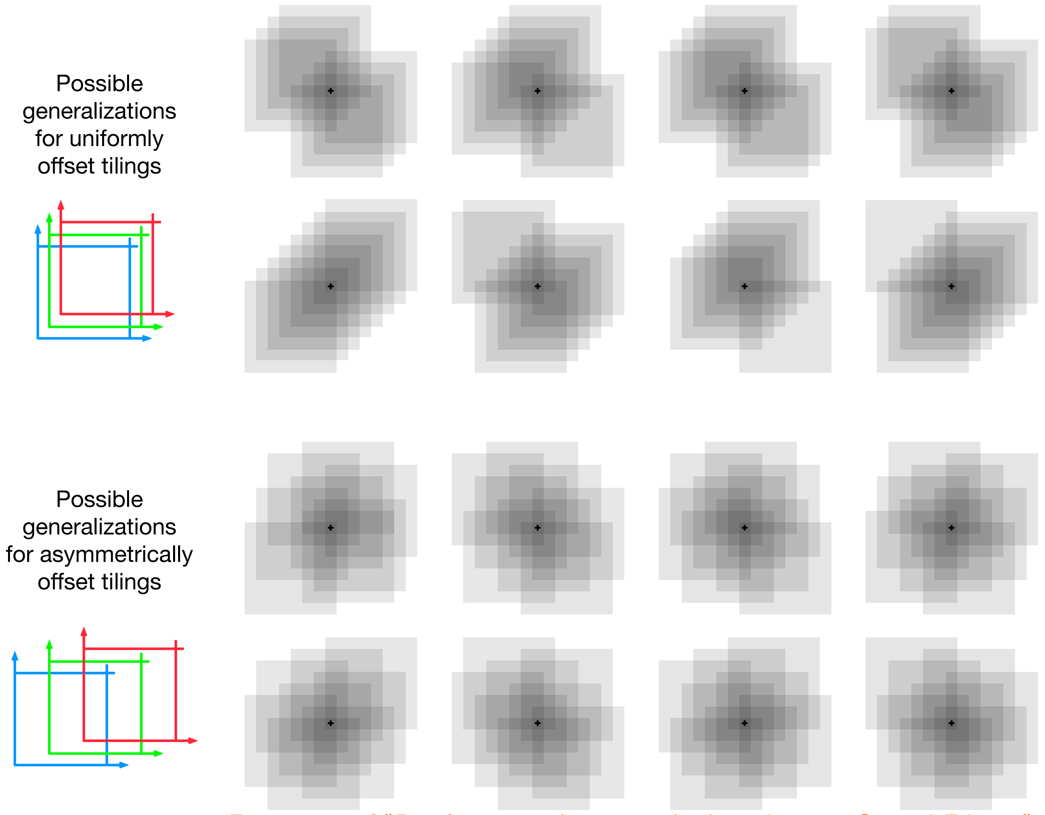

Possible generalizations for uniformly offset tilings
Possible generalizations for asymmetrically offset tilings

Figure 9.11 of "Reinforcement Learning: An Introduction, Second Edition".