N-step Methods, Function Approximation

Milan Straka

November 05, 2018

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

Monte Carlo Assignment

```
Q = np.zeros([env.states, env.actions])
C = np.zeros([env.states, env.actions])
epsilon = args.epsilon
evaluating = False

while True:
    # Perform episode
    state = env.reset(evaluating)
    states, actions, rewards = [], [], []
    while True:
        if evaluating or np.random.uniform() > epsilong: action = np.argmax(Q[state])
        else:                                            action = np.random.randint(env.actions)

        next_state, reward, done, _ = env.step(action)

        states.append(state)
        actions.append(action)
        rewards.append(reward)

        state = next_state
        if done:
            break

    if env.episode >= args.episodes:
        evaluating = True
```

Monte Carlo Assignment

```
if not evaluating:
    # Sum discounted rewards
    for i in reversed(range(len(rewards) - 1)):
        rewards[i] += args.gamma * rewards[i + 1]

    # update Q and C
    for i in range(len(rewards)):
        C[states[i]][actions[i]] += 1
        Q[states[i]][actions[i]] += 1 / C[states[i]][actions[i]] * (rewards[i] - Q[states[i]][actions[i]])

    if args.epsilon_final:
        epsilon = np.exp(np.interp(env.episode + 1,
                                   [0, args.episodes],
                                   [np.log(args.epsilon), np.log(args.epsilon_final)]))
```

Sarsa and Q-learning

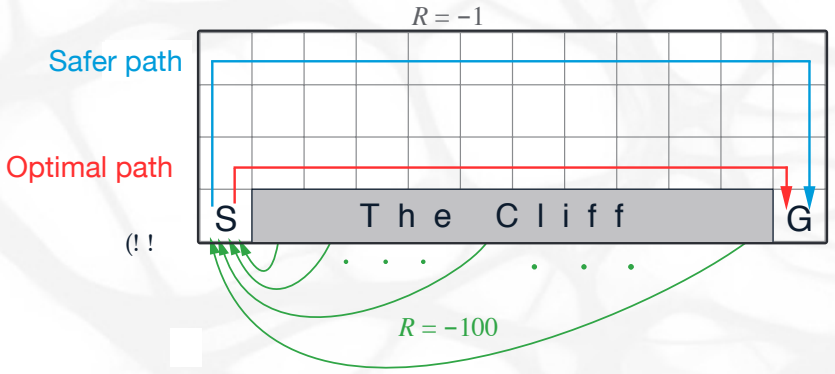
A straightforward application to the temporal-difference policy evaluation is Sarsa algorithm, which after generating $S_t, A_t, R_{t+1}, S_{t+1}, A_{t+1}$ computes

$$q(S_t, A_t) \leftarrow q(S_t, A_t) + \alpha \left[R_{t+1} + \gamma q(S_{t+1}, A_{t+1}) - q(S_t, A_t)\right].$$

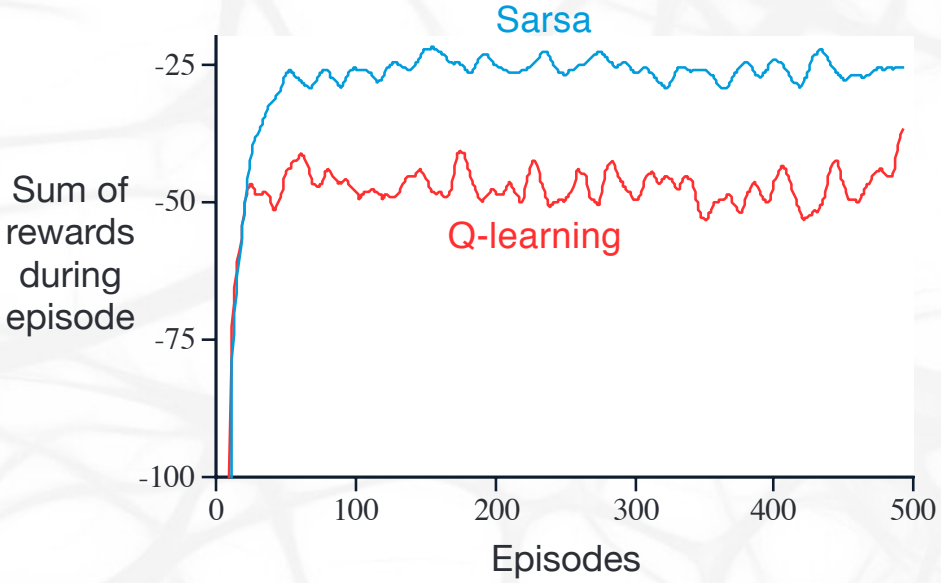
Q-learning was an important early breakthrough in reinforcement learning (Watkins, 1989).

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

Refresh – Q-learning versus Sarsa



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



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

Refresh – 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

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

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

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

Refresh – 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), but usually has smaller variance.

Refresh – Off-policy Monte Carlo Prediction

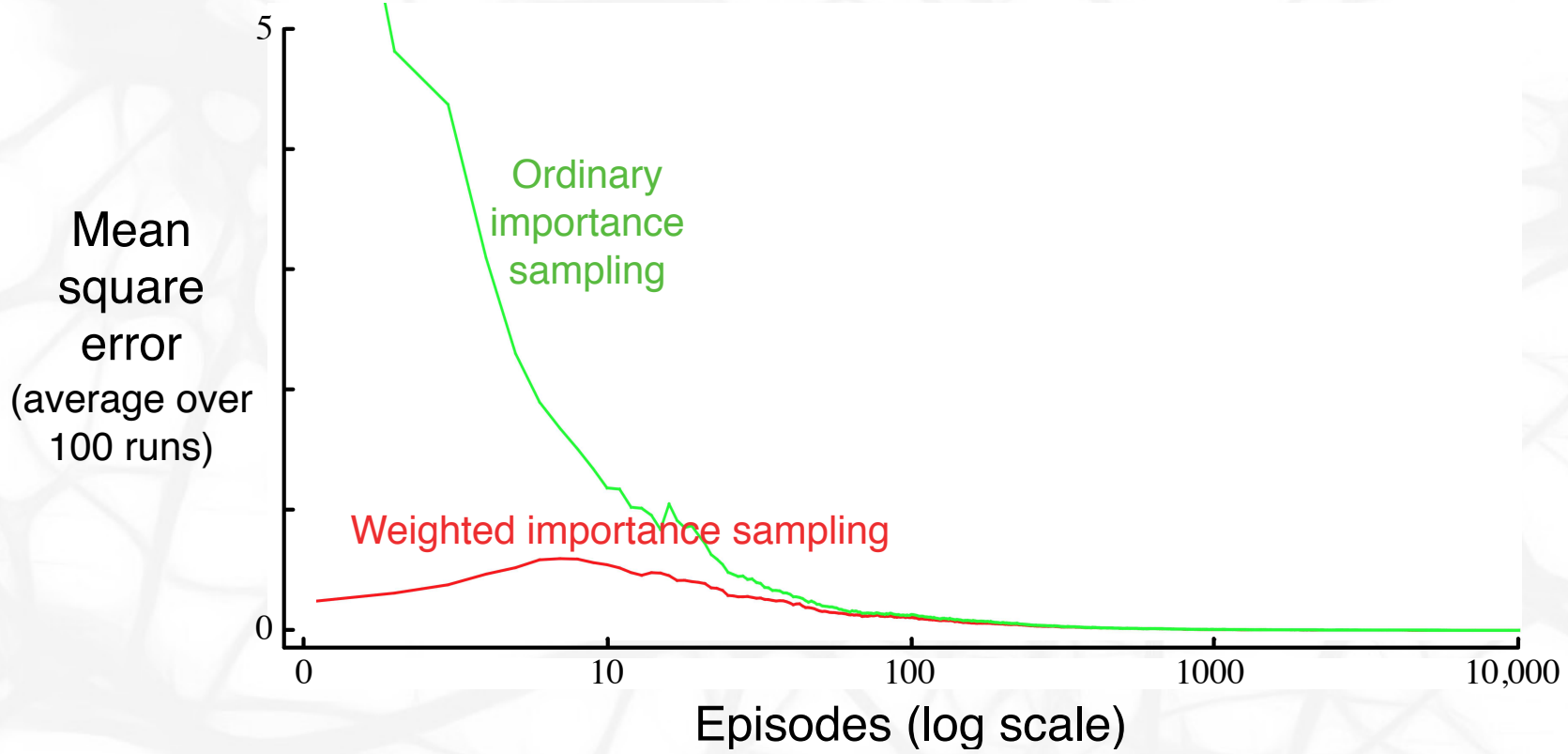


Figure 5.3 of "Reinforcement Learning: An Introduction, Second Edition".

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

Refresh – Expected Sarsa

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

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

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

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 $|\mathcal{A}|$.

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.

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 estimate its value.

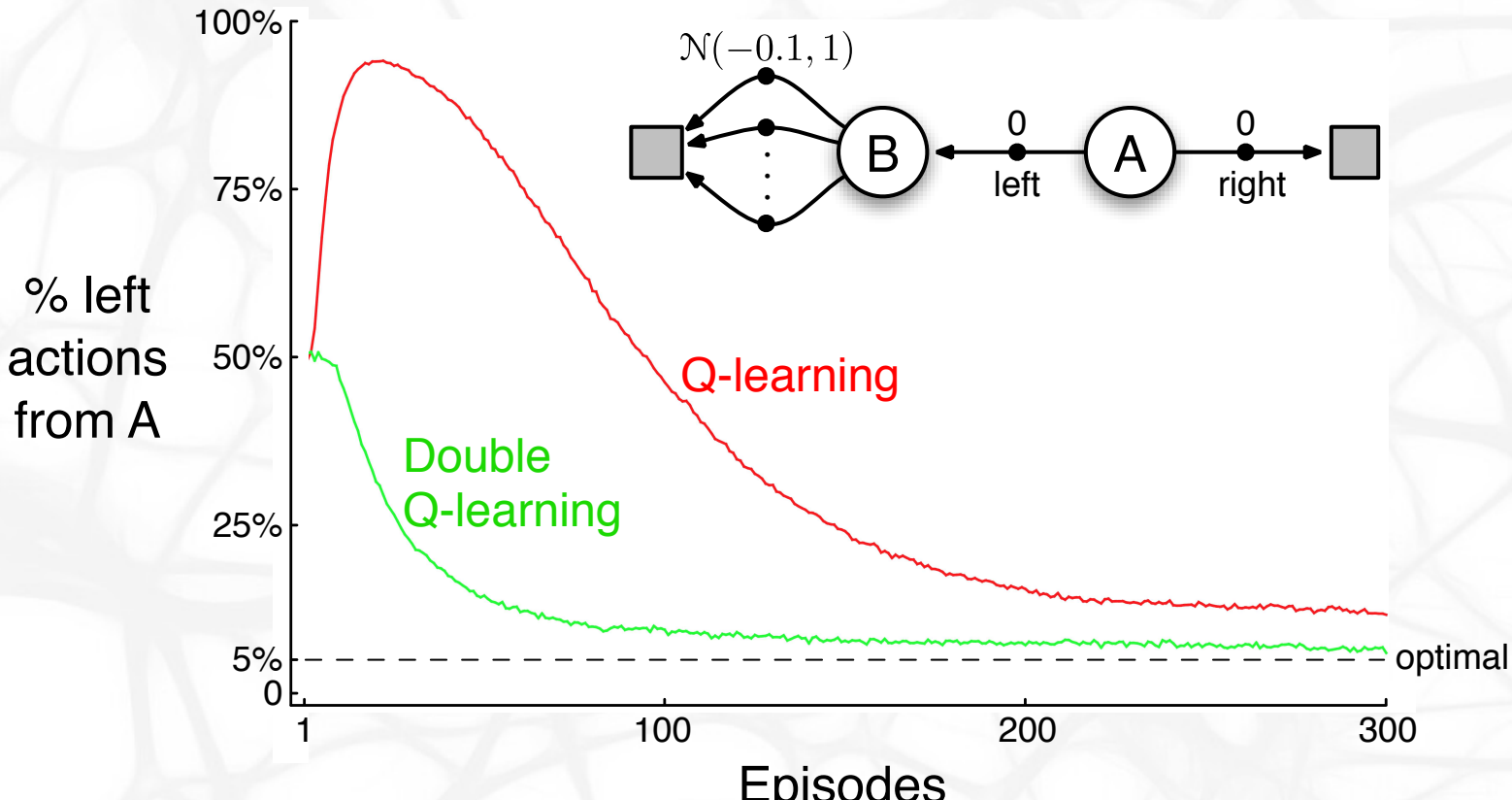


Figure 6.5 of "Reinforcement Learning: An Introduction, Second Edition".

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 $Q(\mathit{terminal}, \cdot) = 0$

Loop for each episode:
  Initialize S
  Loop for each step of episode:
    Choose A from S using the policy ε-greedy in $Q_1 + Q_2$
    Take action A, observe R, S'
    With 0.5 probabilility:

$$Q_1(S, A) \leftarrow Q_1(S, A) + \alpha\Big(R + \gamma Q_2\big(S', \operatorname{argmax}_a Q_1(S', a)\big) - Q_1(S, A)\Big)$$

    else:

$$Q_2(S, A) \leftarrow Q_2(S, A) + \alpha\Big(R + \gamma Q_1\big(S', \operatorname{argmax}_a Q_2(S', a)\big) - Q_2(S, A)\Big)$$

    $S \leftarrow S'$
  until S is terminal

Modification of Algorithm 6.7 of "Reinforcement Learning: An Introduction, Second Edition".

n-step Methods

Full return is

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

one-step return is

$$G_{t:t+1} = R_{t+1} + \gamma V_t(S_{t+1}).$$

We can generalize both into n-step returns:

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

with $G_{t:t+n} \stackrel{\text{def}}{=} G_t$ if $t + n \geq T$.

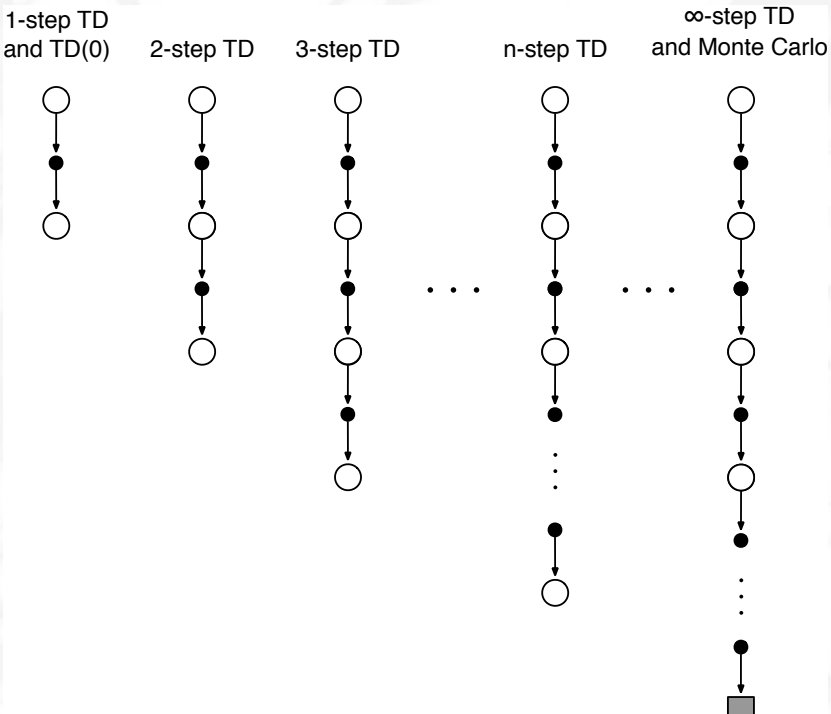


Figure 7.1 of "Reinforcement Learning: An Introduction, Second Edition".

A natural update rule is

$$V_{t+n}(S_t) \overset{\text{def}}{=} V_{t+n-1}(S_t) + \alpha \left[G_{t:t+n} - V_{t+n-1}(S_t) \right].$$

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

Input: a policy π
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 $T \leftarrow t + 1$
  | $\tau \leftarrow t - n + 1$ (τ 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: $G \leftarrow G + \gamma^n V(S_{\tau+n})$ $(G_{\tau:\tau+n})$
  | $V(S_\tau) \leftarrow V(S_\tau) + \alpha \left[G - V(S_\tau) \right]$
  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:

Average RMS error over 19 states and first 10 episodes

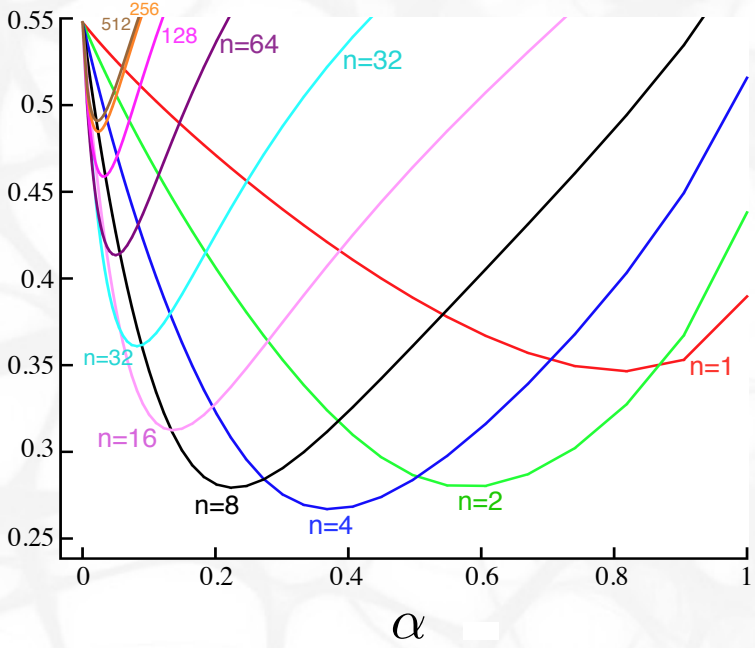


Figure 7.2 of "Reinforcement Learning: An Introduction, Second Edition".

n-step Sarsa

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

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

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

$$Q_{t+n}(S_t, A_t) \overset{\text{def}}{=} Q_{t+n-1}(S_t, A_t) + \alpha \left[G_{t:t+n} - Q_{t+n-1}(S_t, A_t) \right].$$

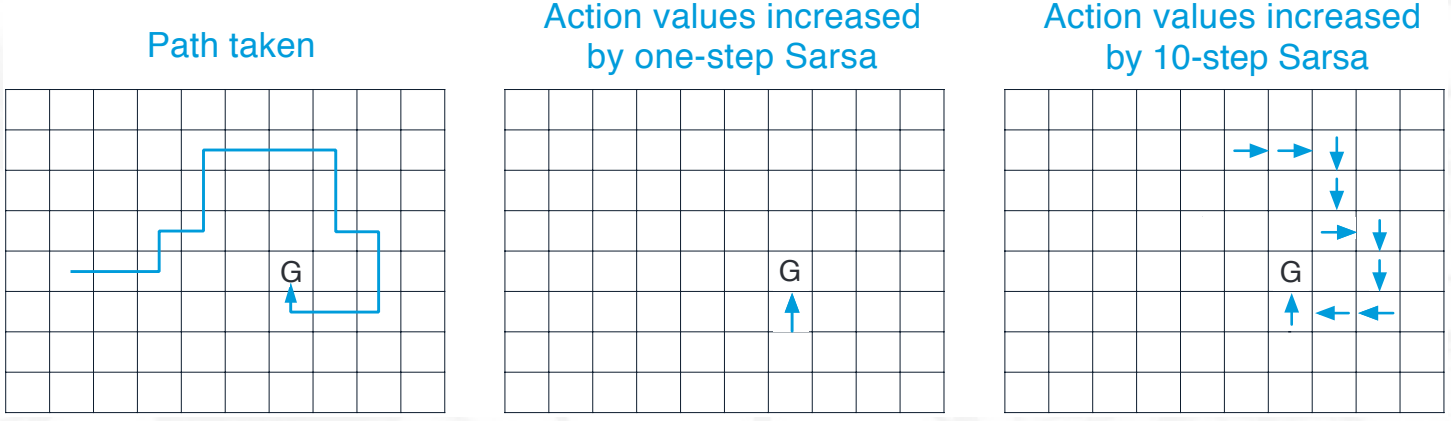


Figure 7.4 of "Reinforcement Learning: An Introduction, Second Edition".

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 π to be ε-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 $t = 0, 1, 2, \ldots$:
  | 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 $A_{t+1} \sim \pi(\cdot|S_{t+1})$
  | $\tau \leftarrow t - n + 1$ (τ 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 π is being learned, then ensure that $\pi(\cdot|S_\tau)$ is ε-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

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

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

$$V_{t+n}(S_t) \stackrel{\text{def}}{=} V_{t+n-1}(S_t) + \alpha\rho_{t:t+n-1}\left[G_{t:t+n} - V_{t+n-1}(S_t)\right].$$

Similarly, n-step Sarsa becomes

$$Q_{t+n}(S_t, A_t) \stackrel{\text{def}}{=} Q_{t+n-1}(S_t, A_t) + \alpha\boldsymbol{\rho_{t+1:t+n}}\left[G_{t:t+n} - Q_{t+n-1}(S_t, A_t)\right].$$

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 π 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 $A_{t+1} \sim b(\cdot|S_{t+1})$
  | $\tau \leftarrow t-n+1$ (τ is the time whose estimate is being updated)
  | If $\tau \geq 0$:
  | $\rho \leftarrow \prod_{i=\tau+1}^{\min(\tau+n-1,T-1)} \frac{\pi(A_i|S_i)}{b(A_i|S_i)}$ $(\rho_{\tau+1:t+n-1})$
  | $G \leftarrow \sum_{i=\tau+1}^{\min(\tau+n,T)} \gamma^{i-\tau-1} R_i$
  | If $\tau+n < T$, then: $G \leftarrow G + \gamma^n Q(S_{\tau+n}, A_{\tau+n})$ $(G_{\tau:\tau+n})$
  | $Q(S_\tau, A_\tau) \leftarrow Q(S_\tau, A_\tau) + \alpha\rho\left[G - Q(S_\tau, A_\tau)\right]$
  | If π is being learned, then ensure that $\pi(\cdot|S_\tau)$ is greedy wrt Q
  Until $\tau = T-1$

Algorithm 7.3 of "Reinforcement Learning: An Introduction, Second Edition".

Off-policy n-step Without Importance Sampling

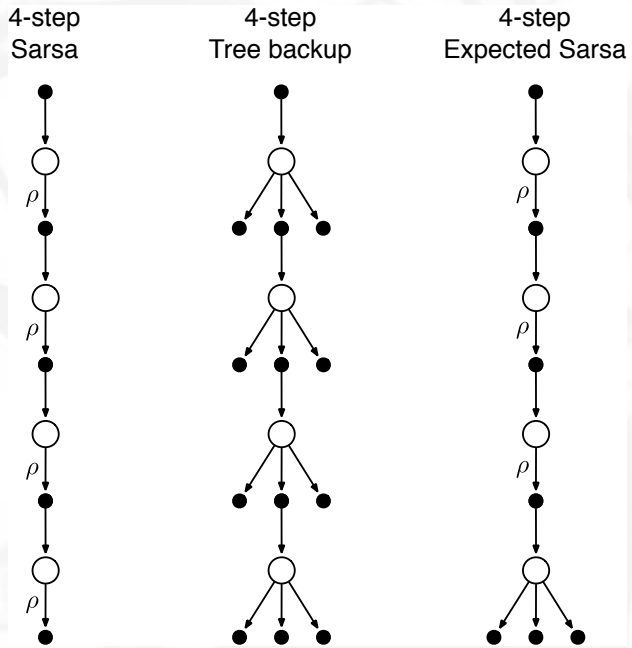


Figure 7.5 of "Reinforcement Learning: An Introduction, Second Edition".

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:

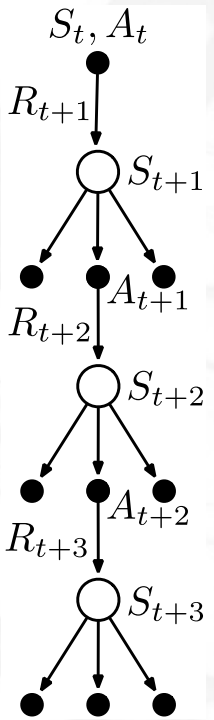
$$G_{t:t+1} \stackrel{\text{def}}{=} R_{t+1} + \sum_a \pi(a|S_{t+1}) Q_t(S_{t+1}, a).$$

For two-step, we get:

$$G_{t:t+2} \stackrel{\text{def}}{=} R_{t+1} + \gamma \sum_{a \neq A_{t+1}} \pi(a|S_{t+1}) Q_t(S_{t+1}, a) + \gamma \pi(A_{t+1}|S_{t+1}) G_{t+1:t+2}.$$

Therefore, we can generalize to:

$$G_{t:t+n} \stackrel{\text{def}}{=} R_{t+1} + \gamma \sum_{a \neq A_{t+1}} \pi(a|S_{t+1}) Q_t(S_{t+1}, a) + \gamma \pi(A_{t+1}|S_{t+1}) G_{t+1:t+n}.$$



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 π 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}
    $\tau \leftarrow t+1-n$ (τ 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 $\tau + 1$:
        $G \leftarrow R_k + \gamma \sum_{a \neq A_k} \pi(a|S_k) Q(S_k, a) + \gamma \pi(A_k|S_k) G$
      $Q(S_\tau, A_\tau) \leftarrow Q(S_\tau, A_\tau) + \alpha \left[G - Q(S_\tau, A_\tau) \right]$
      If π 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".

Function Approximation

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

We will 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 allow an objective for finding the best function approximation.

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

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

Function Approximation

For on-policy algorithms, μ is usually on-policy distribution. That is the stationary distribution under π for continuous tasks, and for the episodic case it is defined as

$$\eta(s) = h(s) + \sum_{s'} \eta(s') \sum_{a} \pi(a|s') p(s|s', a),$$

$$\mu(s) = \frac{\eta(s)}{\sum_{s'} \eta(s')},$$

where $h(s)$ is a probability that an episodes starts in state s.

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

$$\begin{aligned}\boldsymbol{w}_{t+1} &\leftarrow \boldsymbol{w}_t - \frac{1}{2}\alpha\nabla\left[v_\pi(S_t) - \hat{v}(S_t, \boldsymbol{w}_t)\right]^2 \\ &\leftarrow \boldsymbol{w}_t + \alpha\left[v_\pi(S_t) - \hat{v}(S_t, \boldsymbol{w}_t)\right]\nabla\hat{v}(S_t, \boldsymbol{w}_t).\end{aligned}$$

As usual, the $v_\pi(S_t)$ is estimated by a suitable sample. For example in Monte Carlo methods, we use episodic return G_t, and in temporal difference methods, we employ bootstrapping and use $R_{t+1} + \gamma\hat{v}(S_{t+1}, \boldsymbol{w})$.

Monte Carlo Gradient Policy Evaluation

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

Input: the policy π to be evaluated
Input: a differentiable function $\hat{v} : \mathcal{S} \times \mathbb{R}^d \to \mathbb{R}$
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 π
  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

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

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

$$\boldsymbol{w}_{t+1} \leftarrow \boldsymbol{w}_t + \alpha \left[v_\pi(S_t) - \hat{v}(\boldsymbol{x}(S_t), \boldsymbol{w}_t) \right] \boldsymbol{x}(S_t).$$

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 go 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:

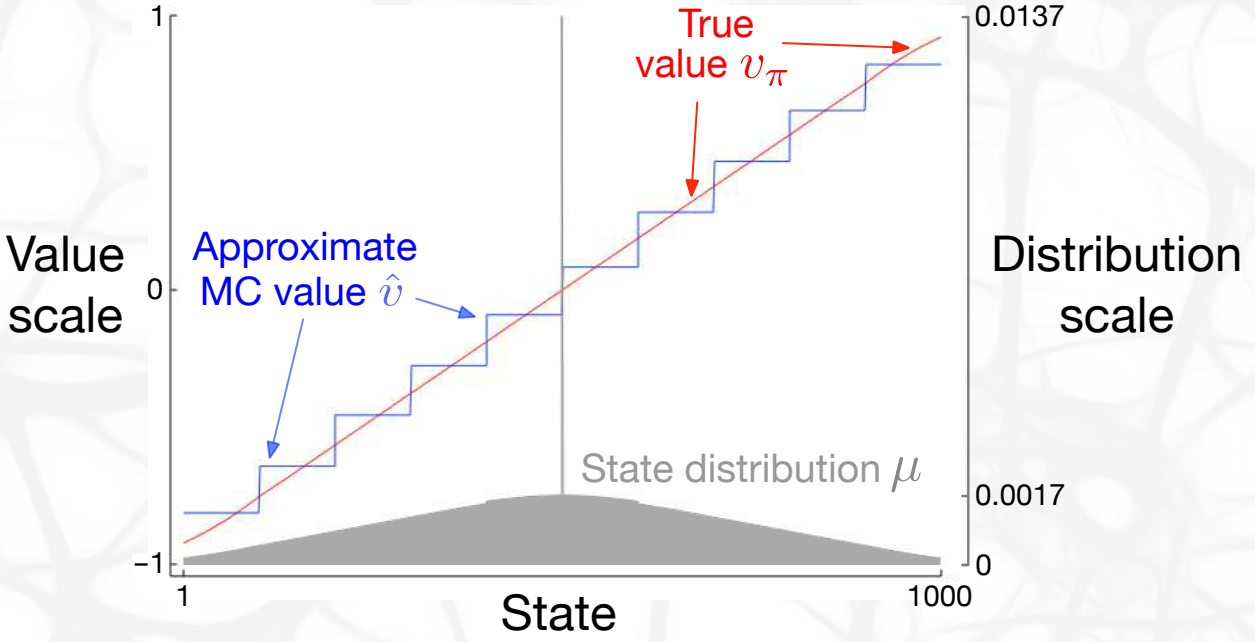


Figure 9.1 of "Reinforcement Learning: An Introduction, Second Edition".

Feature Construction for Linear Methods

Many methods developed in the past:

- polynomials
- Fourier basis
- tile coding
- radial basis functions

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

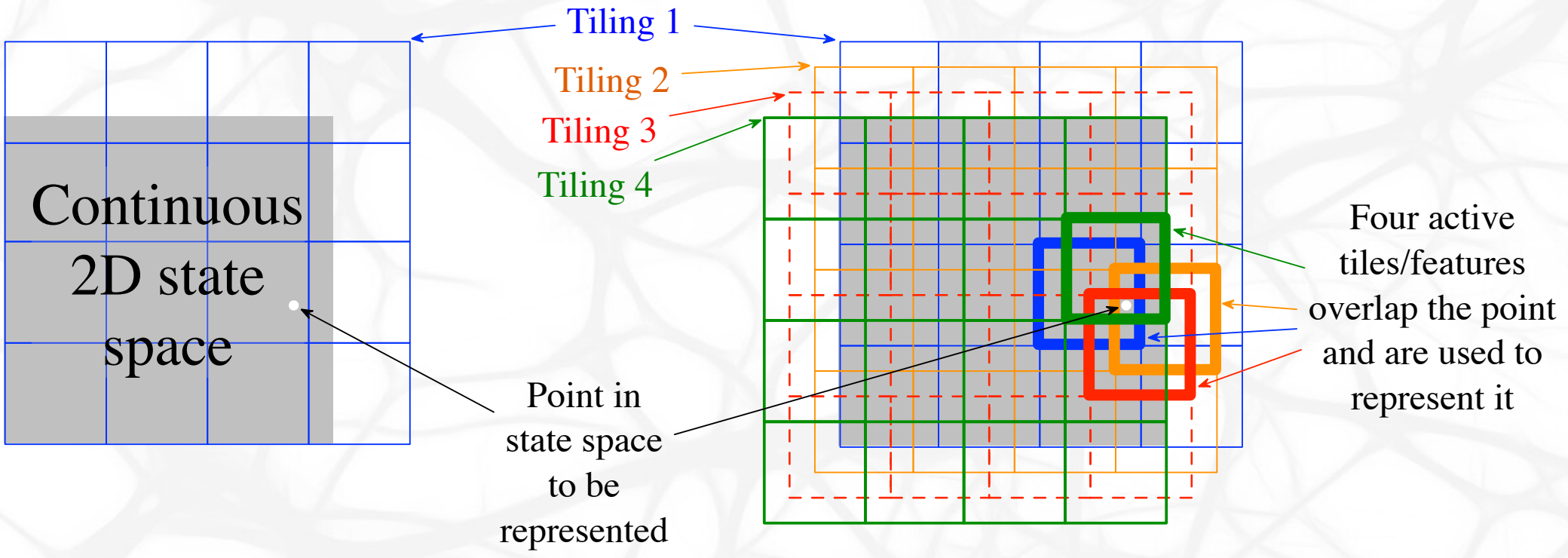


Figure 9.9 of "Reinforcement Learning: An Introduction, Second Edition".

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 tile coding surpasses state aggregation:

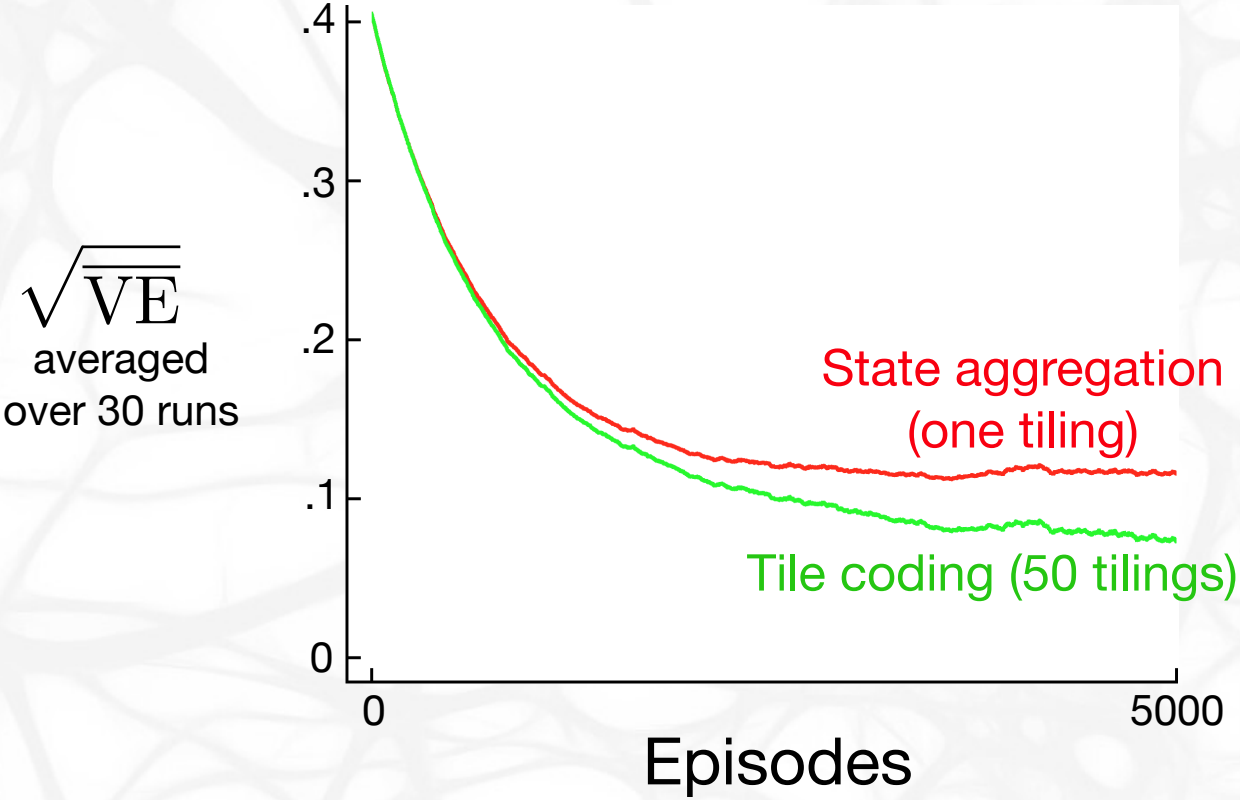


Figure 9.10 of "Reinforcement Learning: An Introduction, Second Edition".

Asymmetrical Tile Coding

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

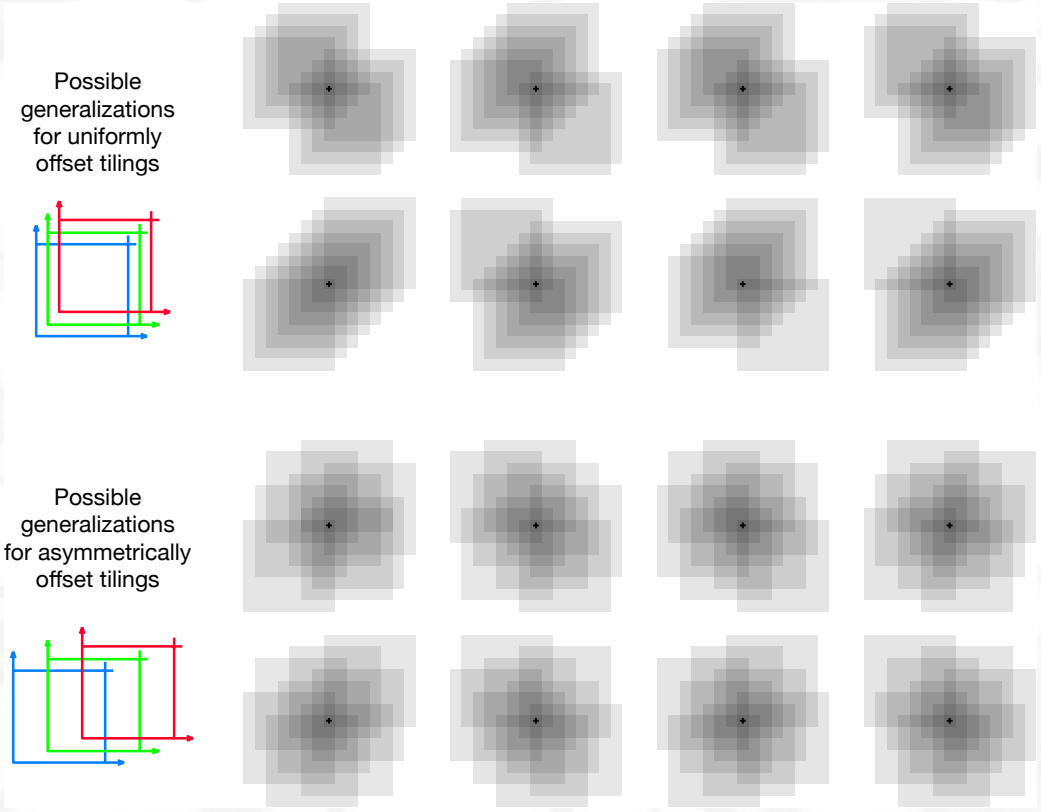


Figure 9.11 of "Reinforcement Learning: An Introduction, Second Edition".