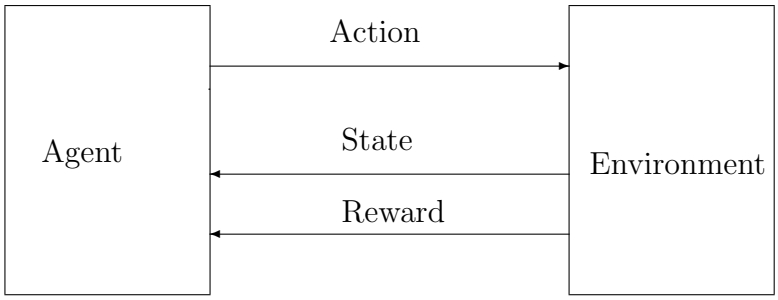
4 Reinforcement Learning – Basic Algorithms

4.1 Introduction

RL methods essentially deal with the solution of (optimal) control problems using on-line measurements. We consider an agent who interacts with a dynamic environment, according to the following diagram:



Our agent usually has only partial knowledge of its environment, and therefore will use some form of *learning* scheme, based on the observed signals. To start with, the agent needs to use some parametric *model* of the environment. We shall use the model of a stationary MDP, with given state space and actions space. However, the state transition matrix $P = (p(s'|s,a))$ and the immediate reward function $r = (r(s,a,s'))$ may not be given. We shall further assume the the observed signal is indeed the state of the dynamic proceed (fully observed MDP), and that the reward signal is the immediate reward r_t, with mean $r(s_t, a_t)$.

It should be realized that this is an *idealized* model of the environment, which is used by the agent for decision making. In reality, the environment may be non-stationary, the actual state may not be fully observed (or not even be well defined), the state and action spaces may be discretized, and the environment may contain other (possibly learning) decision

makers who are not stationary. Good learning schemes should be designed with an eye towards robustness to these modelling approximations.

Learning Approaches: The main approaches for learning in this context can be classified as follows:

- Indirect Learning: Estimate an explicit model of the environment ($\hat{P}$ and $\hat{r}$ in our case), and compute an optimal policy for the estimated model ("Certainty Equivalence").
- Direct Learning: The optimal control policy is learned without first learning an explicit model. Such schemes include:

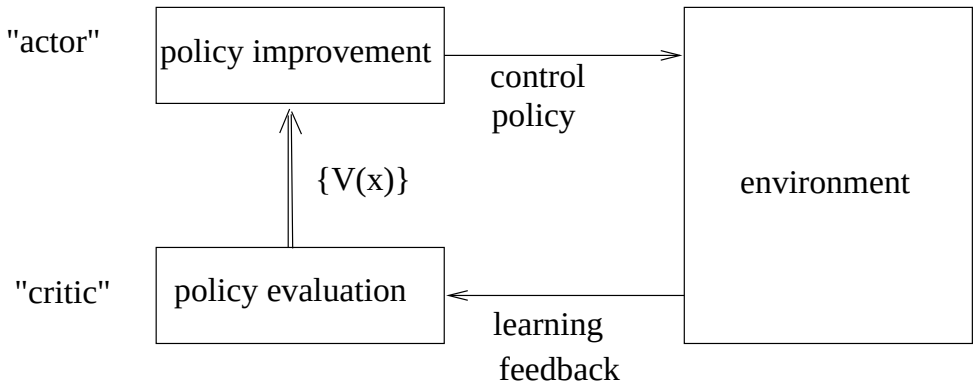
  a. Search in policy space: Genetic Algorithms, Policy Gradient....

  b. Value-function based learning, related to Dynamic Programming principles: Temporal Difference (TD) learning, Q-learning, etc.

RL initially referred to the latter (value-based) methods, although today the name applies more broadly. Our focus in the chapter will be on this class of algorithms.

Within the class of value-function based schemes, we can distinguish two major classes of RL methods.

1. Policy-Iteration based schemes ("actor-critic" learning):



The "policy evaluation" block essentially computes the value function under the current policy (assuming a fixed, stationary policy). Methods for policy evaluation include:

a. "Monte Carlo" policy evaluation.

b. Temporal Difference methods - TD(λ), SARSA, etc.

The "actor" block performs some form of policy improvement, based on the policy iteration idea: $\bar{\pi} \in \text{argmax}\{r + pV\}$. In addition, it is responsible for implementing some "exploration" process.

2. Value-Iteration based Schemes:

These schemes are based on some on-line version of the value-iteration recursions: $V^*_{t+1} = \max_\pi[r^\pi + P^\pi V^*_t]$. The basic learning algorithm in this class is *Q-learning*.

4.2 Example: Deterministic Q-Learning

To demonstrate some key ideas, we start with a simplified learning algorithm that is suitable for a *deterministic* MDP model, namely:

$$s_{t+1} = f(s_t, a_t)$$
$$r_t = r(s_t, a_t)$$

We consider the discounted return criterion:

$$V^\pi(s) = \sum_{t=0}^{\infty} \gamma^t r(s_t, a_t)\,, \quad \text{given } s_0 = s, a_t = \pi(s_t)$$
$$V^*(s) = \max_\pi V^\pi(s)$$

Recall our definition of the Q-function (or *state-action value function*), specialized to the present deterministic setting:

$$Q(s,a) = r(s,a) + \gamma V^*(f(s,a))$$

The optimality equation is then

$$V^*(s) = \max_a Q(s,a)$$

or, in terms of Q only:

$$Q(s,a) = r(s,a) + \gamma \max_{a'} Q(f(s,a), a')$$

Our learning algorithm runs as follows:

- *Iniialize:* Set $\hat{Q}(s,a) = Q_0(s,a)$, for all s, a.
- At each stage $n = 0, 1, \ldots$:
  - Observe s_n, a_n, r_n, s_{n+1}.
  - Update $\hat{Q}(s_n, a_n)$: $\hat{Q}(s_n, a_n) := r_n + \gamma \max_{a'} \hat{Q}(s_{n+1}, a')$

We note that this algorithm does not tell us how to choose the actions a_n. The following result is from [Mitchell, Theorem 3.1].

Theorem 1 (Convergence of Q-learning for deterministic MDPS)
Assume a deterministic MDP model. Let $\hat{Q}_n(s,a)$ denote the estimated Q-function before the n-th update. If each state-action pair is visited infinitely-often, then $\lim_{n\to\infty}\hat{Q}_n(s,a) = Q(s,a)$, for all (s,a).

Proof: Let

$$\Delta_n \triangleq \|\hat{Q}_n - Q\|_\infty = \max_{s,a} |\hat{Q}_n(s,a) - Q(s,a)| \,.$$

Then at every stage n:

$$\begin{aligned}
|\hat{Q}_{n+1}(s_n,a_n) - Q(s_n,a_n)| &= |r_n + \gamma \max_{a'} \hat{Q}_n(s_{n+1},a') - (r_n + \gamma \max_{a''} Q(s_{n+1},a''))| \\
&= \gamma |\max_{a'} \hat{Q}_n(s_{n+1},a') - \max_{a''} \hat{Q}_n(s_{n+1},a'')| \\
&\le \gamma \max_{a'} |\hat{Q}_n(s_{n+1},a') - Q_n(s_{n+1},a')| \;\le\; \gamma\Delta_n \,.
\end{aligned}$$

Consider now some interval $[n_1,n_2]$ over which all state-action pairs (s,a) appear at least once. Using the above relation and simple induction, it follows that $\Delta_{n_2} \le \gamma\Delta_{n_1}$. Since $\gamma < 1$ and since there is an infinite number of such intervals by assumption, it follows that $\Delta_n \to 0$. □

Remarks:

1. The algorithm allows the use of an arbitrary policy to be used during learning. Such as algorithm is called *Off Policy*. In contrast, *On-Policy* algorithms learn the properties of the policy that is actually being applied.

2. We further note that the "next-state" $s' = s_{n+1}$ of stage n need not coincide with the current state s_{n+1} of stage $n+1$. Thus, we may skip some sample, or even choose s_n at will at each stage. This is a common feature of off-policy schemes.

3. A basic requirement in this algorithm is that all state-action pairs will be samples "often enough". To ensure that we often use a specific *exploration* algorithm or method. In fact, the speed of convergence may depend critically on the efficiency of exploration. We shall discuss this topic in detail further on.

4.3 Policy Evaluation: Monte-Carlo Methods

Policy evaluation algorithms are intended to estimate the value functions V^π or Q^π for a given policy π. Typically these are on-policy algorithms, and the considered policy is assumed to be stationary (or "almost" stationary). Policy evaluation is typically used as the "critic" block of an actor-critic architecture.

Direct Monte-Carlo methods are the most straight-forward, and are considered here mainly for comparison with the more elaborate ones. Monte-Carlo methods are based on the simple idea of averaging a number of random samples of a random quantity in order to estimate its average.

Let π be a fixed stationary policy. Assume we wish to evaluate the value function V^π, which is either the discounted return:

$$V^\pi(s) = E^\pi(\sum_{t=0}^{\infty} \gamma^t r(s_t, a_t)|s_0 = s)$$

or the total return for an SSP (or *episodial*) problem:

$$V^\pi(s) = E^\pi(\sum_{t=0}^{T} r(s_t, a_t)|s_0 = s)$$

where T is the (stochastic) termination time, or time of arrival to the terminal state.

Consider first the episodial problem. Assume that we operate (or simulate) the system with the policy π, for which we want to evaluate V^π. Multiple trials may be performed, starting from arbitrary initial conditions, and terminating at T (or truncated before).

After visiting state s, say at time t_s, we add-up the total cost until the target is reached:

$$\hat{v}(s) = \sum_{t=t_s}^{T} R_t \,.$$

After k visits to s, we have a sequence of total-cost estimates:

$$\hat{v}_1(s), \cdots, \hat{v}_k(s) \,.$$

We can now compute our estimate:

$$\hat{V}_k(s) = \frac{1}{k} \sum_{i=1}^{k} \hat{v}_i(s) \,.$$

By repeating these procedure for all states, we estimate $V^\pi(\cdot)$.

State counting options: Since we perform multiple trials and each state can be visited several times per trial, there are several options regarding the visits that will be counted:

a. Compute $\hat{V}(s)$ only for initial states ($s_0 = s$).

b. Compute $\hat{V}(s)$ each time s is visited.

c. Compute $\hat{V}(s)$ only on first visit of s at each trial.

Method (b) gives the largest number of samples, but these may be correlated (hence, lead to non-zero bias for finite times). But in any case, $\hat{V}_k(s) \to V^\pi(s)$ is guaranteed as $k \to \infty$. Obviously, we still need to guarantee that each state is visited enough – this depends on the policy π and our choice of initial conditions for the different trials.

Remarks:

1. The explicit averaging of the $\hat{v}_k$'s may be replaced by the iterative computation:

$$\hat{V}_k(s) = \hat{V}_{k-1}(s) + \alpha_k \Big[\hat{v}_k(s) - \hat{V}_{k-1}(s) \Big] ,$$

with $\alpha_k = \frac{1}{k}$. Other choices for α_k are also common, e.g. $\alpha_k = \frac{\gamma}{k}$, and $\alpha_k = \epsilon$ (non-decaying gain, suitable for non-stationary conditions).

2. For discounted returns, the computation needs to be truncated at some finite time T_s, which can be chosen large enough to guarantee a small error:

$$\hat{v}(s) = \sum_{t=t_s}^{T_s} (\gamma)^{t-t_s} R_t \, .$$

4.4 Policy Evaluation: Temporal Difference Methods

a. The TD(0) Algorithm

Consider the total-return (SSP) problem with $\gamma = 1$. Recall the fixed-policy Value Iteration procedure of Dynamic Programming:

$$V_{n+1}(s) = E^{\pi}(r(s,a) + V_n(s')) = r(s,\pi(s)) + \sum_{s'} p(s'|s,\pi(s))V_n(s')\,, \quad s \in S$$

or $V_{n+1} = r^{\pi} + P^{\pi}V_n$, which converges to V^{π}.

Assume now that r^{π} and P^{π} are not given. We wish to devise a "learning version" of the above policy iteration.

Let us run or simulate the system with policy π. Suppose we start with some estimate $\hat{V}$ of V^{π}. At time n, we observe s_n, r_n and s_{n+1}. We note that $[r_n + \hat{V}(s_{n+1})]$ is an unbiased estimate for the right-hand side of the value iteration equation, in the sense that

$$E^{\pi}(r_n + \hat{V}(s_{n+1})|s_n) = r(s_n,\pi(s_n)) + \sum_{s'} p(s'|s_n,\pi(s_n))V_n(s')$$

However, this is a *noisy* estimate, due to randomness in r and s'. We therefore use it to modify $\hat{V}$ only slightly, according to:

$$\begin{aligned}\hat{V}(s_n) &:= (1-\alpha_n)\hat{V}(s_n) + \alpha_n[r_n + \hat{V}(s_{n+1})] \\ &= \hat{V}(s_n) + \alpha_n[r_n + \hat{V}(s_{n+1}) - \hat{V}(s_n)]\end{aligned}$$

Here α_n is the *gain* of the algorithm. If we define now

$$d_n \triangleq r_n + \hat{V}(s_{n+1}) - \hat{V}(s_n)$$

we obtain the update rule:

$$\hat{V}(s_n) := \hat{V}(s_n) + \alpha_n d_n$$

d_n is called the *Temporal Difference*. The last equation defines the TD(0) algorithm.

Note that $\hat{V}(s_n)$ is updated on basis of $\hat{V}(s_{n+1})$, which is itself an estimate. Thus, TD is a "bootstrap" method: convergence of $\hat{V}$ at each states s is inter-dependent with other states.

Convergence results for TD(0) (preview):

1. If $\alpha_n \searrow 0$ at suitable rate ($\alpha_n \approx 1/\text{no. of visits to } s_n$), and each state is visited i.o., then $\hat{V}_n \to V^\pi$ w.p. 1.

2. If $\alpha_n = \alpha_0$ (a small positive constant) and each state is visited i.o., then $\hat{V}_n$ will "eventually" be close to V^π with high probability. That is, for every $\epsilon > 0$ and $\delta > 0$ there exists α_0 small enough so that

$$\lim_{n\to\infty} \text{Prob}(|\hat{V}_n - V^\pi| > \epsilon) \leq \delta\,.$$

b. TD with ℓ-step look-ahead

TD(0) looks only one step in the "future" to update $\hat{V}(s_n)$, based on r_n and $\hat{V}(s_{n+1})$. Subsequent changes will not affect $\hat{V}(s_n)$ until s_n is visited again.

Instead, we may look ℓ steps in the future, and replace d_n by

$$\begin{aligned} d_n^{(\ell)} &\triangleq [\sum_{m=0}^{\ell-1} r_{n+m} + \hat{V}(s_{n+\ell})] - \hat{V}(s_n) \\ &= \sum_{m=0}^{\ell-1} d_{n+m} \end{aligned}$$

where d_n is the one-step temporal difference as before. The iteration now becomes

$$\hat{V}(s_n) := \hat{V}(s_n) + \alpha_n d_n^{(\ell)}.$$

This is a "middle-ground" between TD(0) and Monte-Carlo evaluation!

c. The TD(λ) Algorithm

Another way to look further ahead is to consider all future Temporal Differences with a "fading memory" weighting:

$$\hat{V}(s_n) := \hat{V}(s_n) + \alpha(\sum_{m=0}^{\infty} \lambda^m d_{n+m}) \tag{1}$$

where $0 \leq \lambda \leq 1$. For $\lambda = 0$ we get TD(0); for $\lambda = 1$ we obtain the Monte-Carlo sample!

Note that each run is terminated when the terminal state is reached, say at step T. We thus set $d_n \equiv 0$ for $n \geq T$.

The convergence properties of TD(λ) are similar to TD(0). However, TD(λ) often converges faster than TD(0) or direct Monte-Carlo methods, provided that λ is properly chosen. This has been experimentally observed, especially when function approximation is used for the value function.

Implementations of TD(λ):

There are several ways to implement the relation in (1).

1. Off-line implementation: $\hat{V}$ is updated using (1) at the end of each simulation run, based on the stored (s_t, d_t) sequence from that run.

2. Each d_n is used as soon as becomes available, via the following backward update (also called "on-line implementation"):

$$\hat{V}(s_{n-m}) := \hat{V}(s_{n-m}) + \alpha \cdot \lambda^m d_n\,, \qquad m = 0, \ldots, n\,. \tag{2}$$

This requires only keeping track of the state sequence $(s_t, t \geq 0)$. Note that is some state s appears twice in that sequence, it is updated twice.

3. Eligibility-trace implementation:

$$\hat{V}(s) := \hat{V}(s) + \alpha d_n e_n(s)\,, \qquad s \in S \tag{3}$$

where

$$e_n(s) = \sum_{k=0}^{n} \lambda^{n-k} 1\{s_k = s\}$$

is called the *eligibility trace* for state s.

The eligibility trace variables $e_n(s)$ can also be computed recursively. Thus, set $e_0(s) = 0$, and

$$e_n(s) := \lambda e_{n-1}(s) + 1\{s_n = s\} \;=\; \begin{cases} \lambda \cdot e_{n-1}(s) + 1 & \text{if} \quad s = s_n \\ \lambda \cdot e_{n-1}(s) & \text{if} \quad s \neq s_n \end{cases} \tag{4}$$

Equations (3) and (4) provide a fully recursive implementation of TD(λ).

d. TD Algorithms for the Discounted Return Problem

For γ-discounted returns, we obtain the following equations for the different TD algorithms:

1. TD(0):

$$\begin{aligned} \hat{V}(s_n) &:= (1-\alpha)\hat{V}(s_n) + \alpha[r_n + \gamma \hat{V}(s_{n+1}] \\ &= \hat{V}(s_n) + \alpha \cdot d_n, \end{aligned}$$

with $d_n \triangleq r_n + \gamma V(s_{n+1}) - V(s_n)$.

2. ℓ-step look-ahead:

$$\begin{aligned} \hat{V}(s_n) &:= (1-\alpha)\hat{V}(s_n) + \alpha[r_n + \gamma r_{n+1} + \cdots + \gamma^{\ell} V_{n+\ell}] \\ &= \hat{V}(s_n) + \alpha[d_n + \gamma d_{n+1} + \cdots + \gamma^{\ell-1} d_{n+\ell-1}] \end{aligned}$$

3. TD(λ):

$$\hat{V}(s_n) := \hat{V}(s_n) + \alpha \sum_{k=0}^{\infty} (\gamma\lambda)^k d_{n+k}\,.$$

The eligibility-trace implementation is:

$$\begin{aligned} \hat{V}(s) &:= \hat{V}(s) + \alpha d_n e_n(s)\,, \\ e_n(s) &:= \gamma\lambda e_{n-1}(s) + 1\{s_n = s\}\,. \end{aligned}$$

e. Q-functions and their Evaluation

For policy improvement, what we require is actually the Q-function $Q^\pi(s,a)$, rather than $V^\pi(s)$. Indeed, recall the policy-improvement step of policy iteration, which defines the improved policy $\hat{\pi}$ via:

$$\hat{\pi}(s) \in \text{argmax}\{r(s,a) + \gamma \sum_{s'} p(s'|s,a)V^\pi(s)\} \equiv \text{argmax}\, Q^\pi(s,a)\,.$$

How can we estimate Q^π?

1. Using $\hat{V}^\pi$: If we know the one-step model parameters r and p, we may estimate $\hat{V}^\pi$ as above and compute

$$\hat{Q}^\pi(s,a) \stackrel{\triangle}{=} r(s,a) + \gamma \sum p(s'|s,a)\hat{V}^\pi(s')\,.$$

When the model is not known, this requires to estimate r and p on-line.

2. Direct estimation of Q^π: This can be done the same methods as outlined for $\hat{V}^\pi$, namely Monte-Carlo or TD methods. We mention the following:

The SARSA algorithm: This is the equivalent of of TD(0). At each stage we observe $(s_n, a_n, r_n, s_{n+1}, a_{n+1})$, and update

$$\begin{aligned} Q(s_n,a_n) &:= Q(s_n,a_n) + \alpha_n \cdot d_n \\ d_n &= r_n + \gamma Q(s_{n+1},a_{n+1}) - Q(s_n,a_n) \end{aligned}$$

Similarly, the SARSA(λ) algorithm uses

$$\begin{aligned} Q(s,a) &:= Q(s,a) + \alpha_n(s,a) \cdot d_n e_n(s,a) \\ e_n(s,a) &:= \gamma\lambda e_{n-1}(s,a) + 1\{s_n = 1, a_n = a\}\,. \end{aligned}$$

Note that:
– The estimated policy π must be the one used ("on-policy" scheme).
– More variables are estimated in Q than in V.

4.5 Policy Improvement

Having studied the "policy evaluation" block of the actor/critic scheme, we turn to the policy improvement part.

Ideally, we wish to implement policy iteration through learning:

(i) Using policy π, evaluate $\hat{Q} \approx Q^\pi$. Wait for convergence.

(ii) Compute $\hat{\pi} = \operatorname{argmax} \hat{Q}$ (the "greedy policy" w.r.t. $\hat{Q}$).

Problems:

a. Convergence in (i) takes infinite time.

b. Evaluation of $\hat{Q}$ requires trying all actions – typically requires an exploration scheme which is richer than the current policy π.

To solve (a), we may simply settle for a finite-time estimate of Q^π, and modify π every (sufficiently long) finite time interval. A more "smooth" option is to modify π slowly in the "direction" of the maximizing action. Common options include:

(i) Gradual maximization: If a^* maximizes $\hat{Q}(s,a)$, where s is the state currently examined, then set

$$\begin{cases} \pi(a^*|s) := \pi(a^*|s) + \alpha \cdot [1 - \pi(a^*|s)] \\ \pi(a|s) := \pi(a|s) - \alpha \cdot \pi(a|s), \quad a \neq a^* \,. \end{cases}$$

Note that π is a *randomized* stationary policy, and indeed the above rule keeps $\pi(\cdot|s)$ as a probability vector.

(ii) Increase probability of actions with high Q: Set

$$\pi(a|s) = \frac{e^{\beta(s,a)}}{\sum_{a'} e^{\beta(s,a)}}$$

(a Boltzmann-type distribution), where β is updated as follows:

$$\beta(s,a) := \beta(s,a) + \alpha[\hat{Q}(s,a) - \hat{Q}(s,a_0)].$$

Here a_0 is some arbitrary (but fixed) action.

(iii) "Pure" actor-critic: Same Boltzmann-type distribution is used, but now with

$$\beta(s,a) := \beta(s,a) + \alpha[r(s,a) + \gamma\hat{V}(s') - \hat{V}(s)]$$

for $(s,a,s') = (s_n, a_n, s_{n+1})$. Note that this scheme uses directly $\hat{V}$ rather than $\hat{Q}$. However it is more noisy and harder to analyze than other options.

To address problem (b) (exploration), the simplest approach is to superimpose some randomness over the policy in use. Simple local methods include:

(i) ϵ-exploration: Use the nominal action a_n (e.g., $a_n = \text{argmax}_a\, Q(s_n, a)$) with probability $(1-\epsilon)$, and otherwise (with probability ϵ) choose another action at random. The value of ϵ can be reduced over time, thus shifting the emphasis from exploration to exploitation.

(ii) Softmax: Actions at state s are chosen according to the probabilities

$$\pi(a|s) = \frac{e^{Q(s,a)/\theta}}{\sum_a e^{Q(s,a)/\theta}}\,.$$

θ is the "temperature" parameter, which may be reduced gradually.

(iii) The above "gradual maximization" methods for policy improvement.

These methods however may give slow convergence results, due to their local (state-by-state) nature.

Another simple (and often effective) method for exploration relies on the principle of *optimism in the face of uncertainty*. For example, by initializing $\hat{Q}$ to high (optimistic) values, we encourage greedy action selection to visit unexplored states. We will revisit those ideas later on in the course.

Convergence analysis for actor-critic schemes is relatively hard. Existing results rely on a *two time scale* approach, where the rate of policy update is assumed much slower than the rate of value-function update.

4.6 Q-learning

Q-learning is the most notable representative of *value iteration* based methods. Here the goal is to compute directly the *optimal* value function. These schemes are typically *off-policy* methdos – learning the optimal value function can take place under any policy (subject to exploration requirements).

Recall the definition of the (optimal) Q-function:

$$Q(s,a) \triangleq r(s,a) + \gamma \sum_{s'} p(s'|s,a) V^*(s') \,.$$

The optimality equation is then $V^*(s) = \max_a Q(s,a)\,,\ s \in S$, or in terms of Q only:

$$Q(s,a) = r(s,a) + \gamma \sum_{s'} p(s'|s,a) \max_{a'} Q(s',a') \,, \quad s \in S, a \in A \,.$$

The value iteration algorithm is given by:

$$V_{n+1}(s) = \max_a \{ r(s,a) + \gamma \sum_{s'} p(s'|s,a) V_n(s') \} \,, \quad s \in S$$

with $V_n \to V^*$. This can be reformulated as

$$Q_{n+1}(s,a) = r(s,a) + \gamma \sum_{s'} p(s'|s,a) \max_{a'} Q_n(s',a') \,, \tag{5}$$

with $Q_n \to Q$.

We can now define the on-line (learning) version of the Q-value iteration equation.

The Q-learning algorithm:

– initialize $\hat{Q}$.

– At stage n: Observe (s_n, a_n, r_n, s_{n+1}), and let

$$\begin{aligned} \hat{Q}(s_n,a_n) &:= (1-\alpha_n)\hat{Q}(s_n,a_n) + \alpha_n [r_n + \gamma \max_{a'} \hat{Q}(s_{n+1},a')] \\ &= \hat{Q}(s_n,a_n) + \alpha_n [r_n + \gamma \max_{a'} \hat{Q}(s_{n+1},a') - \hat{Q}(s_n,a_n)] \,. \end{aligned}$$

The algorithm is obviously very similar to the basic TD schemes for policy evaluation, except for the maximization operation.

Convergence: If all (s, a) pairs are visited i.o., and $\alpha_n \searrow 0$ at appropriate rate, then $\hat{Q}_n \to Q^*$.

Policy Selection:

- Since learning of Q^* does not depend on optimality of the policy used, we can focus on exploration during learning. However, if learning takes place while the system is in actual operation, we may still need to use a close-to-optimal policy, while using the standard exploration techniques (ϵ-greedy, softmax, etc.).
- When learning stops, we may choose a greedy policy:

$$\hat{\pi}(s) = \max_a \hat{Q}(s, a)\,.$$

Performance: Q-learning is very convenient to understand and implement; however, convergence may be slower than actor-critic (TD(λ)) methods, especially if in the latter we only need to evaluate V and not Q.