Business Analytics – 2020/21

Approximate dynamic programming and reinforcement learning for discrete states

↳ Apprendo sul campo

↳ Il modello ce l'ho, ma è complicato e non posso applicarlo

Prof. Paolo Brandimarte
Dip. di Scienze Matematiche – Politecnico di Torino
e-mail: paolo.brandimarte@polito.it
URL: https://staff.polito.it/paolo.brandimarte

This version: May 15, 2023

References

These slides are taken from my book:

P. Brandimarte. *From Shortest Paths to Reinforcement Learning: A MATLAB-Based Introduction to Dynamic Programming*. Springer, 2021.

Other references:

- D.P. Bertsekas. *Dynamic Programming and Optimal Control Vol. 2*, (4th ed.). Athena Scientific, 2012.

- D.P. Bertsekas *Reinforcement Learning and Optimal Control*. Athena Scientific, 2019.

- D.P. Bertsekas, J.N. Tsitsiklis. *Neuro-Dynamic Programming*. Athena Scientific, 1996.

- W.B. Powell. *Approximate Dynamic Programming: Solving the Curses of Dimensionality* (2nd ed.). Wiley, 2011.

- R.S. Sutton, A.G. Barto. *Reinforcement Learning: An Introduction* (2nd ed.). MIT Press, 2018.

ADP and model-free DP

Approximate dynamic programming (**ADP**) is the collective name for a wide array of numerical DP methods that, unlike the standard methods do not ensure that an optimal policy will be found. → Non abbiamo la garanzia di trovare la politica ottima, ma ne vogliamo una migliore rispetto a quella che stiamo usando

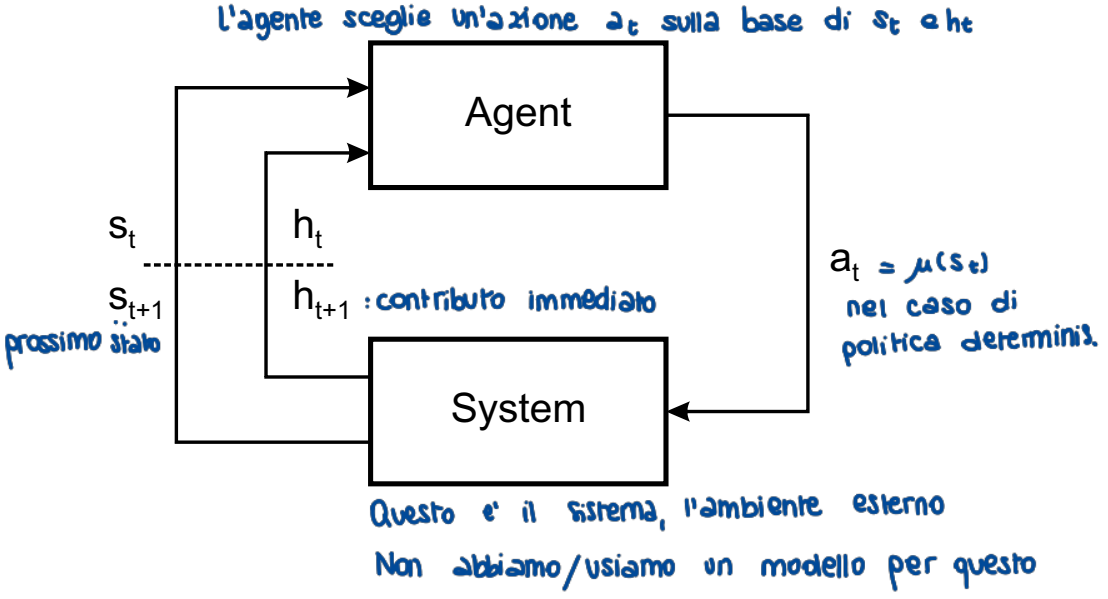
In exchange for their approximate nature, ADP methods aim at easing the overwhelming curses of DP.

Here, we consider approximate methods for finite MDPs in the infinite-horizon setting. Model-free versions of both value iteration and policy iteration are collectively labeled under the umbrella of reinforcement learning (**RL**).

The basic scheme of RL is shown in the figure.

An agent interacts with a system in order to learn a suitable control policy. The agent has no model of the system, but can observe at time t the current state s_t and try an action a_t.

The agent will then observe the immediate contribution h_{t+1} and the next state s_{t+1}.



Bisogna sempre bilanciare tra ora e poi per non usare una politica miope => mi servirebbe una Value Function e quindi delle info su s_{t+1}
↳ opt $f(s,a) + \gamma \mathbb{E}(V|s)$ Ma in questo modo non posso usare il reinforcement learning perche' ho i problemi sul valore atteso — dimensionalita'
\ conoscenza di π
Soluzione_ Utilizziamo i Q-factor, calcolabili tramite sampling

A good policy must balance short- and long-term objectives, which are measured by the immediate contribution and by the value of the next state.

↳ Non ho le probabilita' di transizione

A key point in model-free RL is that even if we have perfect knowledge of the value function $V(s)$ for each state, we cannot really assess the suitability of an action, if we lack any information about the possible transitions to a next state.

A possible remedy is to introduce state–action values, in the form of Q-factors $Q(s, a)$.

If we replace state values with state–action values, we may adopt sampling strategies to learn a decision policy.

Here we assume that the size of the state–action space is not too large and that a tabular representation of Q-factors is feasible.

Quite often, this is not the case, and a compact representation is needed.

Cosa significa fare sampling e stimare i Q-factor?

Apprendo facendo esperimenti

Sampling and estimation in non-stationary settings

Consideriamo di essere in un contesto statico

In probability theory, we are often interested in the expected value of a function $h(\cdot)$ of a random variable X or a vector $\mathbf{X}$ of random variables:

$$\theta = \mathrm{E}\big[h(\mathbf{X})\big] = \int_{\mathcal{X}} h(\mathrm{x}) f_{\mathrm{X}}(\mathrm{x})\, d\mathrm{x},$$

→ difficile se il supporto è su tante dimensioni: utilizziamo Monte Carlo campionando e compiendo una media campionaria

where $\mathcal{X}$ is the support of the distribution.

Since evaluating high-dimensional integrals is no piece of cake, we may resort to Monte Carlo sampling:

La stima è non deviata, questo ci piace molto

$$\widehat{\theta} = \frac{1}{m} \sum_{k=1}^{m} h\big(\mathbf{X}^{(k)}\big)$$

provides an unbiased estimate of the true value of θ.

Ma! Noi abbiamo un problema dinamico, non statico ...

In MDPs, however, we would like to estimate state values $V(i)$ or Q-factors $Q(i,a)$. In this setting, statistical learning must face two sources of complication:

- the need to balance exploration and exploitation;
- the difficulty in estimating a non-stationary target.

↗ Sto stimando qualcosa che si muove

→ Quando campiono (profitto o costo) sto imparando e apprendendo la politica, ma quindi cambia la mia legge di controllo
=> non va bene un Monte Carlo statico

The exploration vs. exploitation tradeoff

Suppose that we are engaged in learning the set of Q-factors $Q(i,a)$ by sampling, and that we are currently at state i. Which action should we choose?

One possibility would be to rely on the current estimates of Q-factors and pick the most promising action. However, this makes sense when the values of each action at that state have been assessed with sufficient accuracy.

However, this will probably not be the case when we have just started our learning process. There could be an action that does not look good because its immediate contribution has been poorly assessed; furthermore, the action may lead to a state which is a good one, but we just do not know it yet.

These considerations raise the fundamental issue of balancing *exploitation* and *exploration*.

Let us simplify the matter and consider a static learning problem. We have to choose one among a finite set $\mathcal{A}$ of alternative courses of action. Each action $a \in \mathcal{A}$ results in a random reward $R(a)$.

If we knew the expected value $\nu(a)$ of the random reward for each action, under a risk neutrality assumption, we would just rank actions according to their expected rewards and choose the most promising alternative.

However, imagine that we are only able to obtain noisy estimates $\widehat{\nu}(a)$ by sampling.

According to a pure exploitation strategy, we would select the action maximizing $\widehat{\nu}(\cdot)$, but this may be an unwise strategy.

: Bandito a molte braccia

Multiarmed bandits

As an example, let us consider a simple multi-armed bandit problem.

Tirando la leva, provo l'azione, ottengo un pay-off

The bandit is a slot machine with different arms, associated with different probability distributions of payoff. Let us consider three normal distributions and sample 10 payoff observations for each one, repeating the experiment 10 times:

```
rng default  % for repeatability
for k=1:10 → Per 10 volte campiono 10 volte
    fprintf(1, 'X1=%.2f   X2=%.2f   X3=%.2f\n', ...
        mean(normrnd(20,5,10,1)), mean(normrnd(25,10,10,1)),... — Consideriamo 3 distribuzioni normali
        mean(normrnd(30,15,10,1)))
end
```

attesa ↓ varianza — Provo un'azione per 10 volte, dopo concludo su cosa devo fare per sempre

The expected value is largest for the third arm, $\mu_3 = 30$; however this also features the largest standard deviation, $\sigma_3 = 15$.

```
X1=23.12   X2=32.05   X3=34.90 → e' andata bene perche' l'azione migliore e' effettivamente quella che dovrebbe
X1=17.79   X2=27.06   X3=28.36
X1=20.64   X2=22.35   X3=34.54
X1=18.77   X2=26.96   X3=24.69
X1=21.39   X2=22.91   X3=19.78
X1=20.19   X2=25.72   X3=24.76
X1=20.12   X2=27.59   X3=30.11
X1=19.36   X2=24.90   X3=21.59
X1=20.23   X2=26.62   X3=25.80
X1=20.31   X2=27.51   X3=20.13
```

=> Volte in cui ho appreso male perche' ho un campione troppo piccolo

It is quite likely that we will miss the optimal choice, with such a small sample size.

Quando si apprende dinamicamente bisogna bilanciare tra exploration e exploitation

On the opposite end of the spectrum, the alternative of pure exploration consists of a random selection of alternatives, based on uniform probabilities. Clearly, we need to find some sensible compromise between these extremes.

Scelgo una probabilità ϵ e con $1-\epsilon$ provo l'azione più promettente = sei avido, cioè scegli l'azione che ti sembra migliore ma con prob ϵ cambi e ne scegli un'altra

- **The static ϵ-greedy approach.** In the static version, we fix a probability ϵ. Then, with probability $1-\epsilon$ we select the most promising action, and with probability ϵ we select a random action. → exploitation / ↳ exploration

In questo caso cambi ϵ dinamicamente → all'inizio deve essere grande, poi può diminuire

- **The dynamic ϵ-greedy approach.** A possible refinement is to change ϵ along the way. One possibility would be to set, at iteration k,

cambia se il sistema è sempre uguale oppure se cambia nell'arco del mio apprendimento

$$\epsilon^{(k)} = \frac{c}{d+k}, \quad \text{or} \quad \epsilon^{(k)} = d + \frac{c}{k}.$$

ϵ va a zero — tende a zero, ma non ci arriva

Questo ha senso quando il sistema è dinamico nel senso che cambia e quindi non smetti realmente mai di apprendere

- **Boltzmann exploration.** This approach relies on a soft-max function to define the probability $\epsilon(a)$ of choosing action $a \in \mathcal{A}$:

Fai una via di mezzo tra scegliere sempre in maniera deterministica o stocastica

$\rho = 0$ pura esplor. — $\rho = \infty$ pura exploi.

parametro di controllo che media tra esplor. e exploi.

$\widehat{\nu}(a)$ è il reward stimato a livello medio

$$\epsilon(a) = \frac{\exp\left(\rho\,\widehat{\nu}(a)\right)}{\sum_{a' \in \mathcal{A}} \exp\left(\rho\,\widehat{\nu}(a')\right)}.$$

When $\rho = 0$, we have pure exploration, whereas the limit for $\rho \to +\infty$ is pure exploitation.

We observe that we are effectively resorting to random policies.

Stai apprendendo online

Non-stationarity and exponential smoothing

Even though the system itself might be stationary, the policy that we follow to make decisions is not, as we are learning along the way.

Consider the familiar sample mean used to estimate the expected value $\theta = \mathsf{E}[X]$ of a scalar random variable X. In the sample mean, all of the collected observations have the same weight.

Consideriamo $\hat{\theta}^{(m)}$ la media campionaria → medie smorzate

Let $\widehat{\theta}^{(m)}$ be the estimate of θ after collecting m observations:

$$\widehat{\theta}^{(m)} = \frac{1}{m}\sum_{k=1}^{m} X^{(k)} = \frac{1}{m}\left(X^{(m)} + \sum_{k=1}^{m-1} X^{(k)}\right) = \frac{1}{m}\left(X^{(m)} + (m-1)\widehat{\theta}^{(m-1)}\right)$$

(nuova info; vecchia stima)

$$= \frac{1}{m}X^{(m)} + \frac{m-1}{m}\cdot\widehat{\theta}^{(m-1)} = \widehat{\theta}^{(m-1)} + \frac{1}{m}\left(X^{(m)} - \widehat{\theta}^{(m-1)}\right).$$

(Il peso di questa correzione tende a zero → A seconda l'errore correggiamo se siamo andati lunghi oppure corti)

↳ Facciamo combinazioni tra nuove e vecchie informazioni

When we obtain a new observation $X^{(m)}$, we apply a correction that is proportional to the forecasting error. to the old estimate $\theta^{(m-1)}$ The amount of correction is smoothed by $1/m$, and tends to vanish as m grows.

In a non-stationary setting, we may wish to keep the amount of correction constant:

$$\widehat{\theta}^{(m)} = \widehat{\theta}^{(m-1)} + \alpha\left(X^{(m)} - \widehat{\theta}^{(m-1)}\right) = \alpha X^{(m)} + (1-\alpha)\widehat{\theta}^{(m-1)}, \quad (1)$$

(coefficiente di smorzamento (smorza le oscillazioni); → combinazione convessa)

(non si cambia mai idea; considero solo l'ultima idea; E' il learning rate)

where the coefficient $\alpha \in (0,1)$ specifies the weight of the new information with respect to the old one. ↳ coefficiente che regola l'impatto della mia correzione

This approach is known as **exponential smoothing**:

Si può notare un'equazione di tipo ricorsiva

$$\begin{aligned}\widehat{\theta}^{(m)} &= \alpha X^{(m)} + (1-\alpha)\widehat{\theta}^{(m-1)} \\ &= \alpha X^{(m)} + \alpha(1-\alpha)X^{(m-1)} + (1-\alpha)^2\,\widehat{\theta}^{(m-2)} \\ &= \sum_{k=0}^{m-1} \alpha(1-\alpha)^k X^{(m-k)} + (1-\alpha)^m\,\widehat{\theta}^{(0)},\end{aligned}$$

con $m \to \infty$ i pesi $\alpha(1-\alpha)^k$ somma a 1 con un profilo esponenziale
- con $m \to \infty$ perdi l'impatto della condizione iniziale

where $\widehat{\theta}^{(0)}$ is an initial estimate. Unlike the standard sample mean, the weights of older observations $X^{(k)}$ are exponentially decreasing.

The coefficient α is known as **learning rate**, or smoothing coefficient, or even forgetting factor.

The larger the learning rate α, the larger the amount of correction in Eq. (1), resulting in more responsive and nervous updates; if, on the contrary, we keep the learning rate small, we introduce a larger amount of inertia and noise filtering.

If the non-stationarity is only due to learning while applying a decision policy, and not to changes in the system dynamics, we may also consider a decreasing sequence of learning rates (step sizes) $\alpha^{(k)}$, to be applied at step k. Sensible and commonly used conditions are

Usiamo degli α che vanno a zero se il target è stazionario
↓
non troppo lentamente

$$\sum_{k=1}^{\infty} \alpha^{(k)} = \infty, \qquad \sum_{k=1}^{\infty} \left[\alpha^{(k)}\right]^2 < \infty.$$

Se volessimo calcolare $\hat{V}(s_0)$? Otteniamo una sequenza di transizioni di stato $h(s_t, \mu(s_t), s_{t+1})$ calcolando la stima tramite la seguente formula, cioè $\hat{V}(s_0) = \sum_{t=0}^{T} \gamma^t h(s_t, \mu(s_t), s_{t+1})$

La «fregatura» di questo approccio è la lunghezza perché dovrei fare questo lavoro per tutti gli stati

Learning by temporal differences and SARSA → è equivalente a policy iteration. Il primo task è apprendere il valore della politica

↓ Policy iteration quando non si hanno le probabilità

We have introduced the operator $\mathcal{T}_\mu$, whose definition is repeated here:

$$[\mathcal{T}_\mu \widetilde{\mathbf{V}}](i) = \sum_{j \in \mathcal{S}} \pi(i, \mu(i), j) \left\{ h(i, \mu(i), j) + \gamma \widetilde{V}(j) \right\}, \qquad i \in \mathcal{S}, \tag{2}$$

↓ Supponiamo μ stazionaria e deterministica

↳ Equazione del punto fisso

where μ is a stationary policy and $\widetilde{\mathbf{V}}$ is a value function.

Fare learning = scelta una politica calcolarne il valore

The value $\mathbf{V}_\mu$ of μ may be obtained by finding the fixed point of $\mathcal{T}_\mu$, i.e., by solving the fixed-point equation $\mathcal{T}_\mu \mathbf{V}_\mu = \mathbf{V}_\mu$.

Let us rewrite the fixed-point equation in terms of Q-factors $Q_\mu(s, a)$ associated with a stationary policy μ:

↳ Per renderlo efficiente conviene apprendere questa tipologia di value function

$$Q_\mu(i, \mu(i)) = \mathsf{E}\left[h(i, \mu(i), j) + \gamma Q_\mu(j, \mu(j)) \right]. \tag{3}$$

↳ Consideriamo il valore atteso = somma sulle probabilità da calcolare stimando

Here the expectation is taken with respect to the next state j, and the action $a = \mu(i)$ is defined by the policy that we want to evaluate.

Cosa fare per trovare il punto fisso di un operatore.

Let us abstract a bit and consider a fixed-point equation

$$\mathbf{y} = H\mathbf{y},$$

where $\mathbf{y}$ is a vector in some linear space and H is an operator.

One possibility to solve the equation is plain fixed-point iteration,

$$\mathbf{y}^{(k)} = H\mathbf{y}^{(k-1)},$$

whose convergence is not guaranteed in general.

Even if the operator H is a contraction, we cannot evaluate it exactly, since we do not know the probability distribution needed to evaluate the expectation in Eq. (3).

Let us rewrite the fixed-point equation as follows:

$$\mathbf{y} = (1-\alpha)\mathbf{y} + \alpha\mathbf{y} = (1-\alpha)\mathbf{y} + \alpha H\mathbf{y} = \mathbf{y} + \alpha(H\mathbf{y} - \mathbf{y}).$$

Then, we may consider using this equation to define an iterative scheme:

$$\mathbf{y}^{(k)} = \mathbf{y}^{(k-1)} + \alpha(H\mathbf{y}^{(k-1)} - \mathbf{y}^{(k-1)}). \tag{4}$$

We may apply exponential smoothing, replacing the expectation with random observations, which is exactly what we do in statistical learning.

Say that we are at state $s^{(k)} = i$, after observing k state transitions starting from an initial state $s^{(0)}$. Then, we apply action $a^{(k)} = \mu(i)$ and, as a result, we will observe:

- the next state $s^{(k+1)} = j$;
- the immediate contribution $h\big(i, \mu(i), j\big)$, or just $f\big(i, \mu(i)\big)$ if it is deterministic.

NB In tutti questi conti non ci sono le "probabilità", è un approccio interamente statistico
Ma tutto questo non risolve il problema della dimensionalità

The adaptation of the scheme of Eq. (4) to this setting leads to the following iterative scheme to learn the value of a given stationary policy:

$\Delta^{(k)}$=0 se avessi i valori corretti perché vale eq pt fisso

$$\Delta^{(k)} = \left[h\left(i,\mu(i),j\right) + \gamma \widehat{Q}_{\mu}^{(k-1)}\left(j,\mu(j)\right)\right] - \widehat{Q}_{\mu}^{(k-1)}\left(i,\mu(i)\right) \quad (5)$$

$$\widehat{Q}_{\mu}^{(k)}\left(i,\mu(i)\right) = \widehat{Q}_{\mu}^{(k-1)}\left(i,\mu(i)\right) + \alpha\Delta^{(k)}, \quad (6)$$

Questa la intendiamo come una «correzione»

Aggiorniamo la stima del Q-factor

$h(i,\mu(i),j)$: è random; $\gamma \widehat{Q}_{\mu}^{(k-1)}(j,\mu(j))$: lo è anche il prossimo stato

Il valore atteso giusto dovrebbe essere 0, ma non lo è perché lavoriamo con le stime

Dovrebbe essere il $\mathbb{E}[\cdot]$, che non sapendolo calcolare considero una realizzazione di una v.a.

α: coefficiente di smorzamento, perché $\Delta^{(k)}$ è rumoroso

$\widehat{Q}_{\mu}^{(k-1)}$: stima precedente

$\Delta^{(k)}$: correzione che è un valore atteso random

where $i = s^{(k)}$ and $j = s^{(k+1)}$. The quantity $\Delta^{(k)}$ in Eq. (5) is an example of a **temporal difference**.

Opt. 1

↳ alternativa a un Monte Carlo costoso

concetto importante

$\Delta^{(k)}$ è una variabile casuale che dipende dal reward e dallo stato precedente

The temporal difference plays the role of the term $H\mathbf{y}^{(k-1)} - \mathbf{y}^{(k-1)}$ in Eq. (4), where the role of operator H is played by the operator $\mathcal{T}_{\mu}$.

The expected value of the temporal difference, if we could use the true Q-factors, would be zero because of Eq. (3). Rather than the exact expected value of a random variable, we have to settle for an observation of the random variable itself.

A non-zero value suggests that we should correct the current estimate. Since corrections are noisy, they are smoothed by the learning rate α.

Opt. 2

Since use estimates to obtain another estimate, the term **bootstrapping** is used to refer to this kind of scheme.

uso delle stime per trovare altre stime

The above algorithm is known as **SARSA**, which is the acronym for the sequence (State, Action, Reward, State, Action). In fact, we observe the current state i, select the action $\mu(i)$, observe a new state j and an immediate reward $h\left(i,\mu(i),j\right)$, and the action $\mu(j)$ at the new state, chosen according to the stationary policy μ that we are evaluating.

Associata sempre a policy iteration

s_i, $\mu(s_i)$, $h(s_i,\mu(s_i),s_j)$, s_j, $\mu(s_j)$

→ Siamo nella fase di learning, abbiamo una politica fissata e cerchiamo di apprenderne il valore

SARSA is an **on-policy** learning approach, as we behave according to a given policy to learn the value of that policy. This is what we are supposed to do in policy iteration.

We need two further ingredients:

1. A suitable amount of exploration, which may be obtained by introducing ϵ-greedy exploration. With a probability $\epsilon^{(k)}$ we deviate from the incumbent policy and select a random action.

2. With SARSA, we will learn the value of a policy only in the limit, after a possibly large amount of steps. Clearly, we have to stop prematurely and perform an improvement step of the incumbent policy before the exact assessment of its value.
   Non ho la garanzia che questa politica sia migliore di quella precedente!
   This approach is known as **generalized** or **optimistic policy iteration**. The term "optimistic" is due to the fact that we optimistically assume that we were able to assess the true value of the current stationary policy.

   This may have a positive effect on the speed of learning, but a detrimental one on convergence.

When we stop learning and have an approximation $\widehat{Q}_\mu(i,a)$ of the Q-factors, we try improving the incumbent policy $\mu(\cdot)$ by a greedy approach:

$$\tilde{\mu}(i) \in \arg\operatorname*{opt}_{a\in\mathcal{A}(i)} \widehat{Q}_\mu(i,a), \qquad i \in \mathcal{S}.$$

Consideriamo i Q-factor in forma tabulare, non ci sono problemi di stime per via delle interpolazioni

↳ Avrei bisogno dei Q-factor di tutte le azioni per migliorare quindi bilanciamo tra exploration e exploitation

Unlike exact iteration, there is no guarantee that the new policy $\tilde{\mu}(\cdot)$ is really improved, as the evaluation of the incumbent policy $\mu(\cdot)$ need not be exact.

Se facessi iterazioni esatte avrei la certezza di ottenere una politica migliore, ma con le varie approssimazioni che ho non ne sono affatto sicura.

Algoritmo per fare reinforcement learning senza il modello. Andiamo a ragionare su Q-factor ottimali, non strettamente legati a una politica

Da qui accennare

Q-learning for finite MDPs

Q-learning is an alternative model-free reinforcement learning approach to cope with finite MDPs.

Let us recall the essential equations for DP in terms of optimal Q-factors:

Notiamo che i Q(i,a) dovrebbero essere ottimi, non sono legati a una politica nello specifico ⇒ differenza con SARSA

$$Q(i,a) = \sum_{j \in \mathcal{S}} \pi(i,a,j) \left[h(i,a,j) + \gamma \underset{a' \in \mathcal{A}(j)}{\text{opt}} Q(j,a') \right], \qquad i \in \mathcal{S},\ a \in \mathcal{A}(i)$$

↳ Ad ogni iterata avremo un'ottimizzazione

$$V(i) = \underset{a \in \mathcal{A}(i)}{\text{opt}} Q(i,a).$$ → Avendo tutti i Q-factor, scelgo il più bello

Note that, when dealing with SARSA, we have used equations for the Q-factors associated with a stationary policy μ, whereas here we deal with the *optimal* factors.

We have to learn about $h(i,a,j)$ and $\pi(i,a,j)$ by experimentation and sampling. At iteration k, when we are about to choose action $a^{(k)}$, we have a current set of estimates of Q-factors

$$\widehat{Q}^{(k-1)}(i,a).$$

The superscript $(k-1)$ points out that these estimates were updated at the previous step, after observing the effect of action $a^{(k-1)}$.

Now we are at state $s^{(k)} = i$ and, based on current estimates, we select the next action by considering what looks best:

$$a^{(k)} \in \arg\underset{a \in \mathcal{A}(i)}{\text{opt}}\, \widehat{Q}^{(k-1)}(i,a). \quad (7)$$

→ Facendo solo questo non riuscirai a convergere

It is worth observing that in Q-learning we are applying the same off-policy logic the is used in value iteration.

Unlike policy iteration, we are not evaluating a given policy μ. The policy is implicit in the Q-factors and, since we continuously update them, we keep changing the policy; in some sense, we use a policy to learn about another one.

After applying the selected action we will observe:

- the next state $s^{(k+1)} = j$;
- the immediate contribution $h\big(i, a^{(k)}, j\big)$, or just $f\big(i, a^{(k)}\big)$ if it is deterministic.

Then, we can use this information to update the estimate of $Q\big(s^{(k)}, a^{(k)}\big)$. We have obtained a new observation of a random variable that expresses the tradeoff between the short-term objective (immediate contribution) and the long-term objective (value of the next state), when we are at state $s^{(k)} = i$ and select action $a^{(k)}$:

contributo immediato

$$\widetilde{q} = h\big(i, a^{(k)}, s^{(k+1)}\big) + \gamma \operatorname*{opt}_{a' \in \mathcal{A}(s^{(k+1)})} \widehat{Q}^{(k-1)}\big(s^{(k+1)}, a'\big). \tag{8}$$

differenza con SARSA perche' non seguo una politica specifica

Note that this includes a random observation of the next state $s^{(k+1)} = j$, as well as its value when following the policy implicit in the Q-factor estimates at step $k-1$:

$$\widehat{V}^{(k-1)}\big(s^{(k+1)}\big) = \operatorname*{opt}_{a' \in \mathcal{A}(s^{(k+1)})} \widehat{Q}^{(k-1)}\big(s^{(k+1)}, a'\big).$$

Then, we update the Q-factor for the state–action pair $\left(s^{(k)}, a^{(k)}\right)$, by using the same logic of exponential smoothing:

Informazione nuova — Informazione vecchia

$$\widehat{Q}^{(k)}\left(s^{(k)}, a^{(k)}\right) = \alpha\widetilde{q} + (1-\alpha)\widehat{Q}^{(k-1)}\left(s^{(k)}, a^{(k)}\right). \tag{9}$$

Here we assume a constant smoothing coefficient α, but we could use a diminishing one. We may also express the same idea in a temporal difference style:

È la differenza temporale che ci serve per aggiornare il Q-factor

$\widetilde{q}$

$$\Delta^{(k)} = \left[h\left(s^{(k)}, a^{(k)}, j\right) + \gamma \operatorname*{opt}_{a' \in \mathcal{A}(j)} \widehat{Q}^{(k-1)}\left(j, a'\right)\right] - \widehat{Q}^{(k-1)}\left(s^{(k)}, a^{(k)}\right) \tag{10}$$

$$\widehat{Q}^{(k)}(s,a) = \begin{cases} \widehat{Q}^{(k-1)}(s,a) + \alpha\Delta^{(k)} & \text{if } s = s^{(k)} \text{ and } a = a^{(k)}, \\ \widehat{Q}^{(k-1)}(s,a) & \text{otherwise,} \end{cases} \tag{11}$$

where $j = s^{(k+1)}$ is the next observed state.

The difference between SARSA and Q-learning may be appreciated by comparing Eqs. (5) and (10). These expressions are both corrections to be applied to current state–action value estimates, but the former case is on-policy and refers to an incumbent stationary policy μ; on the contrary, the latter case is off-policy, as we keep changing the policy.

MATLAB implementation of Q-learning

```
function [QFactors, policy, visits] = QLearning(systemObj,objSense, ...
    discount,numSteps,startQFactors,alpha,epsFlag,epsilon)
if nargin < 7, epsFlag = false; end
QFactors = startQFactors;
visits = 0*startQFactors; % initialize to zero, using shape of Q table
currentState = systemObj.currentState;
for k = 1:numSteps
    if epsFlag
        if rand > epsilon
            [~, action] = findBest(currentState);
        else
            action = datasample(systemObj.feasibleActions{currentState},1);
        end
    else
        [~, action] = findBest(currentState);
    end
    visits(currentState, action) = visits(currentState, action) + 1;
    [contribution, newState] = systemObj.getNext(action);
    qtilde = contribution + discount*findBest(newState);
    QFactors(currentState,action) = alpha*qtilde + ...
                                    (1-alpha)*QFactors(currentState,action);
    currentState = newState;
end % for
```

```
% now find optimal policy
[numStates,~] = size(QFactors);
policy = zeros(numStates,1);
for i = 1:numStates
    [~, policy(i)] = findBest(i);
end

% nested function
    function [value, action] = findBest(state)
        actionList = systemObj.feasibleActions{state};
        qf = QFactors(state,actionList);
        if strcmpi(objSense, 'max')
            [value, idx] = max(qf);
        else
            [value, idx] = min(qf);
        end
        action = actionList(idx);
    end

end
```

A possible MATLAB implementation of Q-learning is shown in Fig. **??**. The function `QLearning` produces a matrix `QFactors` of Q-factors, a vector `policy` describing the stationary policy, and a matrix `visits` counting the number of times that a state–action pair has occured. The function needs the following input arguments:

- `systemObj`, an object of class `systemClass` (to be described below) implementing the simulation of the dynamic system we want to control;
- `objSense`, a flag setting the direction of optimization, `min` or `max`;
- `discount`, the discount factor;
- `numSteps`, the number of simulation steps to learn the policy;
- `startQFactors`, a matrix containing the initial Q-factors;
- `alpha`, the learning rate.

This code is also able to implement a simple (static) ϵ-greedy action selection. If the `epsFlag` flag is set to true, a random action is selected with probability `epsilon`. We use the `datasample` MATLAB function for the random selection of an action; here, the function selects at random an element of a vector containing the feasible actions in the current state. The nested function `findBest` picks the optimal action, taking care of the selected optimization sense.

MATLAB definition of a generic system class

Since Q-learning is a model-free approach, we want to keep the Q-learning algorithm well separated from the simulation model of the system. The interface with the system is provided here by the MATLAB class `systemClass`.

Programmazione ad oggetti

```
classdef systemClass < handle
    properties
        numStates
        numActions
        feasibleActions
        transArray
        payoffArray
        currentState
    end
    methods
        % Constructor
        function obj = systemClass(transArray, payoffArray, ...
                        feasibleActions, initialState)
            obj.transArray = transArray;
            obj.payoffArray = payoffArray;
            obj.feasibleActions = feasibleActions;
            obj.currentState = initialState;
            [obj.numStates, obj.numActions] = size(payoffArray);
        end
```

```
        function obj = setState(obj, state)
            obj.currentState = state;
        end

        function [contribution, nextState] = getNext(obj,action)
            contribution = obj.payoffArray(obj.currentState, action);
            nextState = find(1 == ...
                 mnrnd(1,obj.transArray(obj.currentState,:,action)));
            obj.currentState = nextState;
        end
    end % methods
end
```

Il Q-learning vede il sistema solo attraverso questa funzione → Nascondo tutto in una "scatola nera"

NOTE: In a serious object-oriented implementation, this should be an *abstract* class defining only the interface between the Q-learner and the system.

The class properties include: `feasibleActions`, a cell array containing, for each state, the list of feasible actions; `transArray`, a three-dimensional array of transition probability matrix (actions correspond to the third index); `payoffArray`, a matrix giving, for each state–action pair, the immediate contribution.

The essential method of this class is `getNext`, which provides the immediate contribution and the next state for a selected action, based on the `currentState`.

Note that we associate the contribution with the pair, which implicitly assumes that it is either deterministic or that we know its expected value. In order to implement alternative systems, we could just instantiate a lower level class overriding `getNext`.

A numerical example

Let us apply Q-learning to the recurrent optimal stoping problem, where we find, assuming a discount factor of $\gamma = 0.8$:

$$V(1) = 9.67742, \quad V(2) = 17.7419, \quad V(3) = 27.7419, \quad V(4) = 37.7419.$$

The corresponding optimal policy is to always reset the state and collect the immediate reward.

In order to apply Q-learning, we need to instantiate a specific `systemObj` object of class `systemClass` and run the `QLearning` function, choosing the initial factors as well.

The setting of parameters is the same that we have used for value and policy iteration, and the initial values of the Q-factors are set to 50 for the wait action, and to the immediate reward for the reset action.

```
probs = [0.7; 0.8; 0.9; 0];
payoff = [0; 10; 20; 30];
discount = 0.8;
payoffArray = [zeros(4,1), payoff];
% build transition matrices
transReset = [ones(4,1), zeros(4, 3)];
auxProbs = probs(1:(end-1)); % the last one is not relevant
transWait = diag([auxProbs;1]) + diag(1-auxProbs,1);
transArray(:,:,1) = transWait;
transArray(:,:,2) = transReset;
feasibleActions = {1, [1;2], [1:2], 2};
initialState = 1;
systemObj = systemClass(transArray,payoffArray,feasibleActions,initialState);

rng('default')
numSteps = 1000; → Stiamo apprendendo a comare, ho bisogno di piu' step
alpha = 0.1;
startQFactors = [ repmat(50,4,1), payoff(:)];
[QFactors, policy] = QLearning(systemObj,'max',discount,numSteps, ...
    startQFactors,alpha);
display(startQFactors);
display(QFactors);
disp(['policy  ', num2str(policy')]);
```

We set the smoothing coefficient to $\alpha = 0.1$ and run 1000 iterations, obtaining the following output:

```
startQFactors =
    50     0
    50    10
    50    20
    50    30
QFactors =
   11.0476         0
    9.8825   17.6721
   19.8226   26.9803
   50.0000   47.7861
policy  1   2   2   2
visits =
   608     0
   168   132
    61    22
     0     9
```

La politica implicita che gli si passa e' "aspetta sempre", lui ci mette un po' a capire che aspettare non e' la soluzione migliore

Alcuni Q-factor saranno beccati in maniera ottima e altri molto meno

→ Ha beccato correttamente la politica
1: Aspetta
2: Schiaccia il bottone

→ Anche qui non avviene l'aggiornamento perche' non e' uno stato visitato

→ Non viene mai visitata, quindi 0 non e' aggiornato

If we compare the Q-factors corresponding to the optimal decisions against the state values obtained by value iteration, we find them in fairly good agreement, with the exception of the last state, which is visited quite rarely.

What really matters, however, is that the state–action value estimates are good enough to induce the optimal policy.

This may look reassuring, but what if we change the initialization a bit?

In the second run, we initialize the factors for the wait action ($a = 1$) to 900:

```
startQFactors =
   900     0
   900    10
   900    20
   900    30

QFactors =
   10.9569         0
   14.0696   10.0000
   19.9080   25.5919
  900.0000   58.8755

policy  1  1  2  2
```

Supponiamo ora di dare Q-factor differenti => in questo caso non vede la politica ottima

→ Qua non capisce che la soluzione migliore non è aspettare in questo caso 1000 step sono troppi pochi per capire

↓

Avrebbe senso prendere un learning rate più grande così che le nuove informazioni abbiano più peso rispetto a quelle vecchie

With this bad initialization, we assign a very large value to wait, and after 1000 iterations we fail to find the optimal policy.

Note that the factor $Q(2, 2)$ for reset is not changed, as this action is never applied.

Arguably, we may solve the issue by adopting a larger value of α, at least in the initial steps, in order to accelerate learning.

If we increase α to 0.5, we obtain

```
startQFactors =
   900     0
   900    10
   900    20
   900    30

QFactors =
   16.4106         0
    9.5378   22.2903
   19.6721   32.5977
  900.0000  114.9065

policy  1  2  2  2
```

→ Arrivo alla politica, ma mi tiro dentro tanto rumore! Converge, ma in un sistema complicato probabilmente non succederebbe

Note that the poor estimates for state 4 are inconsequential.

In this case, it seems that a faster learning rate does the job. However, what if we keep $\alpha = 0.5$ and use an alternative initialization?

```
startQFactors =
     1     0
     1     0
     1     0
     1     0
QFactors =
    7.2515         0
   14.3098         0
   21.2052         0
    1.0000   38.8769
policy  1  1  1  2
visits =
   195     0
   294     0
   452     0
     0    59
```

In questo caso non arriva a convergenza, lo ε rimane e l'azione non la proviamo mai

Quindi non "schiaccerà" mai il bottone e la politica ottima non verrà mai trovata a meno che non facciamo exploitation

Con Q-factor alti diamo l'incentivo a provare delle azioni, cioè siamo molto ottimisti

If the Q-factors for the reset action are initialized to zero, then this action will never be tried, and we will never discover that it is, in fact, the optimal one.

Indeed, we observe from `visits` that we never apply action reset at states 2 and 3.

Changing the learning rate is of no use, as the Q-factors for wait cannot get negative.

This illustrates the need of balancing exploitation and exploration.

As a remedy, we could resort to ϵ-greedy exploration. Let us try it with $\epsilon = 0.1$:

```
numSteps = 10000;
[QFactors, policy] = QLearning(systemObj,'max',discount,numSteps, ...
    startQFactors,alpha,true,0.1);
```

This yields the following result:

```
startQFactors =
     1     0
     1     0
     1     0
     1     0
QFactors =
   10.5408         0
   16.3370   18.0537
    1.7721   26.2163
    1.0000    7.1070
policy  1  2  2  2
visits =
        7611           0
         129        2214
          19          25
           0           2
```

Notiamo che un'idea intelligente sarebbe aggiornare dinamicamente il learning rate e e dell'ϵ greedy

We observe again that the state values are not exact, but now they are good enough to find the optimal policy. However, there is a price to pay in terms of computational effort.