Lez21_ReinforcementLearning2

December 12, 2023

1 Markov Decision Process

1.1 Recap

Riprendiamo le caratteristiche di un contesto **MDP**:

- $\mathcal{S}$: a (finite) set of states
- $\mathcal{A}$: a (finite) set of actions
- p: state transition probabilities

$$p(s'|s,a) = P[S_{t+1} = s'|S_t = s, A_t = a]$$

- r: reward function (or, c: cost function)
  1. $r(s,a) = E[R_t|S_t = s, A_t = a]$
  2. $r(s,a,s') = E[R_t|S_t = s, A_t = a, S_{t+1} = s'] \longrightarrow$ $r(s,a) = \sum_{s'} p(s'|s,a) r(s,a,s')$

Anche il *reward* non è deterministico, come un gratta e vinci. La funzione di reward $r(s,a)$ è quindi un *valore atteso*, in funzione dello *stato* e dell'*azione*.

Rilevante è l'*obiettivo* che ci poniamo. Abbiamo parlato di *task episodici*, aventi una fine, come una partita a scacchi. La somma dei reward da t alla fine dell'episodio al tempo T è la somma dei reward. Noi vogliamo *minimizzare* tale aspetto.

Se i task non hanno fine naturale? Ne è un esempio la gestione dei semafori. Ciò porterebbe ad una somma divergente di *reward*. La soluzione è l'uso di **expected cumulative discounted reward**:

$$G_t = R_t + \gamma R_{t+1} + \ldots = \sum_{k=0}^{\infty} \gamma^k R_{t+k}$$

where $\gamma \in [0,1)$ is the discount factor

Quindi andando avanti nel tempo, il reward sarà minore, premiando i *reward* nel breve periodo.

Tornando ai semafori, sono molto interessato al comportamento attuale rispetto a quello futuro, portando ad una scelta di γ vicino ad 1, come 0.99, perchè non vogliamo penalizzare troppo.

1.2 Massimizzazione reward vs minimizzazione costo

Sono equivalenti, in particolare vale che:

$r(s, a) = -c(s, a)$

1.3 Calcolo della policy

Trattasi di politica di azione o comportamento. E' distribuzione definita sullo spazio delle azioni dato uno stato s. Ovvero la probabilità di scegliere l'azione a al tempo t, dato lo stato s.

$\pi(a|s) = p(A_t = a|S_t = s)$

Ciò descrive pienamente il comportamento dell'agente, in funzione unicamente dello stato attuale. (Nulla vieta di creare dipendenza da più stati, ma sarebbe inutile, perchè si ha la proprietà di *memoryless*).

Un caso speciale è la *policy deterministica*, dove si ha $\pi : \mathcal{S} \to \mathcal{A}$, dove ogni stato mi da una *singola* scelta con probabilità 1, e altre scelte con probabilità 0.

Sotto alcune assunzioni, per la stragrande maggioranza degli MDP considerabili, la politica ottima è proprio la politica deterministica. Può non essere unica.

Un esempio di policy deterministica è:

State	Action
s_1	a_1
s_2	a_1
s_3	a_2
s_4	a_1

1.3.1 Value function

Funzione che predice i costi/reward futuri, in base allo stato attuale ed azione da compiere. Ci fa capire la bontà di una scelta.

State	a_1	a_2	a_3
s_1	10	5	3
s_2	8	6	4
s_3	6	5	6
s_4	5	4	6
s_5	4	3	7
s_6	1	5	9
s_7	0	9	15

s	$\pi(s)$
s_1	a_3
s_2	a_3
s_3	a_2
s_4	a_2
s_5	a_2
s_6	a_1
s_7	a_1

Come si legge? Non è $r(s, a)$, ma predizione dei **costi** (valori attesi) che ottengo da adesso fino alla fine dell'episodio. In s_1, l'azione che mi dà più reward è a_1. Se minimizzassi, allora punterei i costi in rosso, ovvero quelli minimi.

Esistono **due value functions**:

- **Q(uality) Function/Action value**: Per ogni stato ed azione, mi dice valore atteso di ritorno, partendo dallo stato s, prendendo azione a e poi seguendo la policy π. $Q_\pi(s, a) = E_\pi[G_t|S_t = s, A_t = a]$
- **State value function**: Costo atteso, partendo dallo stato s e seguendo la policy π, senza consideare lo stato a. $V_\pi(s) = E\pi[G_t|S_t = s]$

Action Value Functions Composta da *costo/reward immediato* e poi *i costi dallo stato succesivo fino all'infinito.*

$$
\begin{aligned}
Q_\pi(s, a) &= E_\pi[G_t|S_t = s, A_t = a] \\
&= E_\pi[C_t + \gamma C_{t+1} + \gamma^2 C_{t+2} \ldots |S_t = s, A_t = a] \\
&= E_\pi[C_t + \gamma (C_{t+1} + \gamma C_{t+2} \ldots) |S_t = s, A_t = a] \\
&= E_\pi[C_t + \gamma G_{t+1}|S_t = s, A_t = a] \\
&= c(s, a) + \gamma E_\pi[G_{t+1}|S_t = s, A_t = a]
\end{aligned}
$$

Come vediamo, viene messo in evidenza γ, portandoci ad avere la somma di tutti i costi, cioè G_{t+1}. Successivamente, essendo valore atteso, possiamo spezzare in $c(s, a)$ e nell'altra componente, pari al valore atteso di G_{t+1}.

Il tutto può essere riscritto nel seguente modo:

$$
Q_\pi(s, a) = c(s, a) + \gamma \sum_{s' \in S} p(s'|s, a) Q_\pi(s', \pi(s'))
$$

ovvero nell'**equazione di Bellman**. L'idea è che, in Q_π abbiamo spezzato in due termini, il primo il costo atteso, il secondo il costo atteso per tutti gli istanti successivi. Questa stima dei costi da $t + 1$ all'infinito ricorda la **Value Function**, stima dei costi futuri. L'unica differenza è che questa non parte dallo stato successivo, ma da quello attuale, e quindi ci riportiamo a questa scrittura. Così possiamo dire che la Q attuale dipende dal costo attuale e la Q degli stati futuri.

Questo ragionamento può essere applicato anche per la **State Value Functions**:

$$
\begin{aligned}
V_\pi(s) &= E_\pi[G_t|S_t = s] \\
&= E_\pi[C_t + \gamma C_{t+1} + \gamma^2 C_{t+2} \ldots |S_t = s] \\
&= E_\pi[C_t + \gamma (C_{t+1} + \gamma C_{t+2} \ldots) |S_t = s] \\
&= E_\pi[C_t + \gamma G_{t+1}|S_t = s]
\end{aligned}
$$

e conseguente **Equazione di Bellman**:

$$V_\pi(s) = c(s, \pi(s)) + \gamma \sum_{s'} p(s'|s, \pi(s)) V_\pi(s')$$

Optimal Value Function $Q*(s;a)$ è il massimo *action value* tra tutte le policy se: $Q^*(s,a) = \max_\pi Q_\pi(s,a)$

Optimal state value Function $V^*(s)$ è il massimo *value-function* tra tutte le policy se: $V^*(s) = \max_\pi V_\pi(s)$

1.3.2 Bellman Optimality Equazions

$$Q_\pi(s,a) = c(s,a) + \gamma \sum_{s' \in S} p(s'|s,a) Q_\pi(s', \pi(a'))$$

$$\Downarrow$$

$$\boxed{\begin{gathered} Q^*(s,a) = c(s,a) + \gamma \sum_{s' \in S} p(s'|s,a) \min_{a'} Q^*(s',a') \\ V^*(s) = \min_a Q^*(s,a) \\ Q^*(s,a) = c(s,a) + \gamma \sum_{s' \in S} p(s'|s,a) V^*(s') \end{gathered}}$$

(dovrebbe esserci qualche errore nella formula)

La prima formula nel riquadro si rivela utile se sapessimo il valore ottimo dello stato in cui *andremo a finire*, e quindi potremmo trovare anche lo stato corrente.

1.3.3 Optimal Policy

La politica ottima prende l'azione che massimizza o minimizza $Q*(s,a)$, facile se è nota la *Optimal value function*.

$$\pi^*(s) = a^*(s) = \arg\min_{a \in \mathcal{A}} Q^*(s, a)$$

State	a_1	a_2	a_3
s_1	10	5	3
s_2	8	6	4
s_3	6	5	6
s_4	5	4	6
s_5	4	3	7
s_6	1	5	9
s_7	0	9	15

$\Longrightarrow$

Optimal Action
a_3
a_3
a_2
a_2
a_2
a_1
a_1

Il problema è:

Chi ci fornisce la *Optimal Value Function* V^*?

1.4 Value Iteration

Da Belman, si ha:

$$Q^*(s, a) = c(s, a) + \gamma \sum_{s' \in S} p(s'|s, a) \min_{a'} Q^*(s', a')$$

Supponiamo di conoscere $Q^*(s', a')$, allora computiamo ciclicamente, per tutti gli s e tutte le a, finchè non convergono al valore ottimo, ovvero:

$$Q^*(s, a) \leftarrow c(s, a) + \gamma \sum_{s' \in S} p(s'|s, a) \min_{a'} Q^*(s', a')$$

Quindi inizializziamo tutte le Q^*, eseguiamo gli aggiornamenti, e i valori si avvicineranno al valore corretto, portando ad una miglior stima della Optimal Value Function. Supponiamo di avere stati ed azioni finite. In pseudo-codice:

Value Iteration

1 $i \leftarrow 0$
2 $Q_i(s,a) \leftarrow 0, \forall s \in \mathcal{S}, \forall a \in \mathcal{A}(s)$
3 **repeat**
4   **forall** $s \in \mathcal{S}$ **do**
5   **forall** $a \in \mathcal{A}(s)$ **do**
6             $Q_{i+1}(s,a) \leftarrow$
              $c(s,a) + \gamma \sum_{s' \in \mathcal{S}} p(s'|s,a) \min_{a' \in \mathcal{A}(s')} Q_i(s',a')$
7   **end**
8   **end**
9     $i \leftarrow i+1$
10 **until** $\max_{s,a} |Q_i(s,a) - Q_{i-1}(s,a)| < \epsilon$
11 $\pi^*(s) = \arg\min_a Q_i(s,a), \forall s \in \mathcal{S}$

Ci si ferma quando tra il nuovo e vecchio valore di $Q_i(s,a)$ la differenza è irrilevante. Ciò va fatto per ogni stato. Capiamo che l'esecuzione è pesante, ma garantisce la convergenza. Nella riga 6, usiamo $Q(s,a)$ per determinare la policy, ma nello stesso momento $Q(s,a)$ è calcolata rispetto una policy. Usiamo l'una per calcolare l'altra, e la cosa funziona!

Esiste una piccola variante:

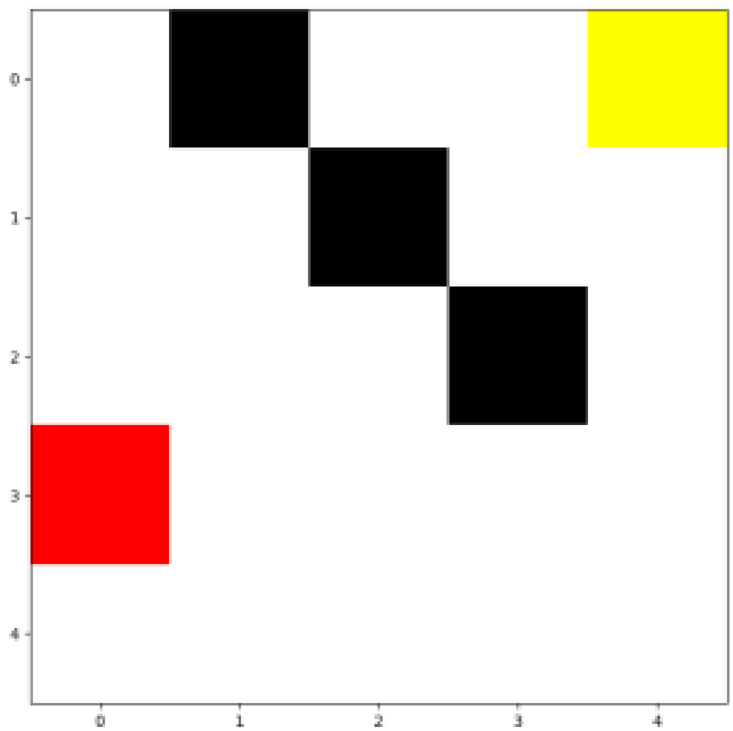
Value Iteration - Alternative

1 $i \leftarrow 0$
2 $Q_i(s,a) \leftarrow 0, \forall s \in \mathcal{S}, \forall a \in \mathcal{A}(s)$
3 $V_i(s) \leftarrow 0, \forall s \in \mathcal{S}$
4 **repeat**
5   **forall** $s \in \mathcal{S}$ **do**
6   **forall** $a \in \mathcal{A}(s)$ **do**
7             $Q_{i+1}(s,a) \leftarrow c(s,a) + \gamma \sum_{s' \in \mathcal{S}} p(s'|s,a) V_i(s)$
8   **end**
9         $V_{i+1}(s) = \min_{a' \in \mathcal{A}(s)} Q_{i+1}(s,a')$
10   **end**
11     $i \leftarrow i+1$
12 **until** $\max_{s,a} |Q_i(s,a) - Q_{i-1}(s,a)| < \epsilon$
13 $\pi^*(s) = \arg\min_a Q_i(s,a), \forall s \in \mathcal{S}$

in cui risparmio qualche calcolo, sfruttando più memoria.

1.5 Labirinto Maze

Prendiamo una griglia $R \times C$. L'agente è posto randomicamente, e deve arrivare nella casella *gialla* nel minor tempo possibile. Supponiamo di non sapere nulla di ingegneria degli algoritmi (No shortest path, infatti l'agente non sa nulla dell'ambiente circostante), e che le celle nere siano bloccati, e che la cella rossa sia *scivolosa*, quindi mi porta a compiere un passo doppio (anche fuori dalla griglia).



Definiamo:

- Obiettivo, identificato dalle coordinate (x', y').
- Lo stato, identificato dalla posizione corrente (x, y)
- Le azioni, sono gli spostamenti, $a \in \{(0, 1), (0, -1), (1, 0), (-1, 0)\}$
- Il reward è:
  - 0 se raggiunge l'obiettivo,
  - $-M$ se esce dalla griglia o sbatte in una cella bloccata, con M molto grande, per indicare quanto io voglia evitare questo esito (sono dei *vincoli*).
  - -1 altrimenti.

Non stiamo "apprendendo", perchè non c'è nulla da imparare, stiamo usando probabilità e costi noti, con reinforcement learning, l'agente non sa nulla.

1.6 Esempio codice Maze

la *Class Maze* è di supporto, con celle animazioni.

Il *RandomAgent* ha metodo `next_action`, per la prossima azione, ed `update`. Non apprende, quindi non usa questo metodo.

La logica dell'episodio è nella funzione `episode`, che prende agente, modello di reward (come definito prima), e posizione finale. L'episodio continua fino a quando non si arriva nella posizione finale, o se si supera un certo numero di tentativi. Ci si chiede se la cella è valida, o se sia scivolosa (raddoppiando lo spostamento appena fatto) e ricontrollare se la cella è quella finale. Nel mentre, si aggiornano i reward.

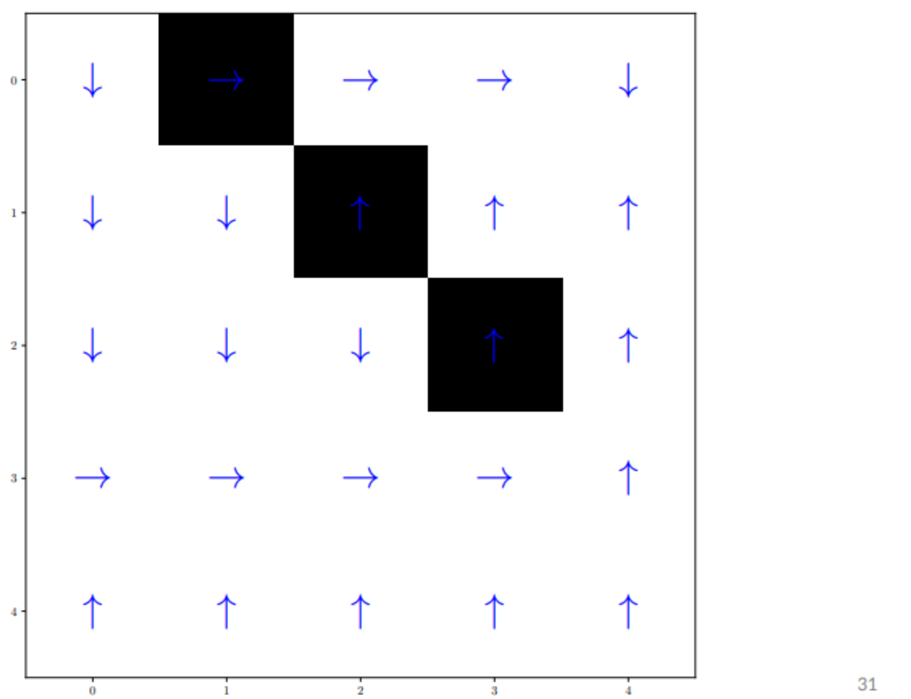
```
pos = new_pos
        positions.append(pos)

        if not np.array_equal(pos, final_pos):
            reward = MOVE_REWARD
            total_reward += reward
            agent.update(reward, pos-a, a, pos)
        else:
            reward = GOAL_REWARD
            total_reward += reward
            agent.update(reward, pos-a, a, pos)
```

Lo si avvia con `python main.py --agent mdp`, e ritorna il *reward* e le iterazioni.

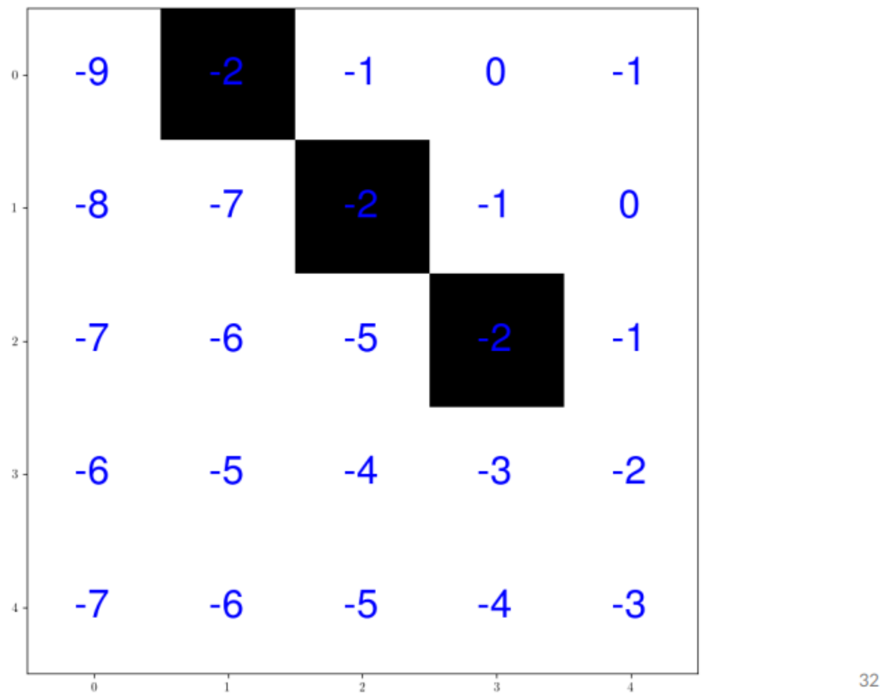
Si pu cambiare la dimensione della griglia, con: `python main.py --agent mdp --size 15`

e le celle scivolose con `python main.py --agent mdp --size 10 --slippery_cells 20`

La **politica ottima** è rappresentata da:



mentre la **Optimal value Function** è:



Partendo dalla cella $(0, 0)$, il valore migliore è -9, meno di quello non posso fare. Partendo dalla cella $(1, 4)$, si ha valore migliore pari a 0, infatti ci basta un solo passo per arrivare a $(4, 4)$.

[ ]:

```
```