Lez22_MDP

December 15, 2023

1 Formulazione MDP

1.1 Esempio sistemi cloud

1.1.1 Autoscaling

Ci poniamo in un ambiente **cloud**, in cui a seconda delle richieste alteriamo il numero di macchine virtuali disponibili in parallelo. Il nostro agente si sveglia una volta ogni tot di tempo (ad esempio 1 minuto) e compie un'azione. Questo vale sia in caso discreto che nel caso discreto.

Definiamo lo *stato* al time slot i come $si = (k_i, \lambda_i)$, dove k_i è la componente del parallelismo (ad esempio il numero di macchine in parallelo di cui dispongo) e λ_i rappresenta il tasso di arrivo medio (come richieste, job, data,…).

Si ha che l'azione al time slot i è rappresentata da $a_i \in \{0, +1, -1\}$, ovvero altero o meno il numero di macchine virtuali.

Poichè λ è continuo, essendo un numero in virgola mobile, semplifichiamo la trattazione, suddividendolo in *gradini*, in modo da discretizzare.

L'azione compiuta modifica k, cioè incrementa, diminuisce o non altera il numero di *macchine virtuali*.

In questo **MDP** le transizioni hanno componente determinista (k) e una stocastica (λ), perchè non controllabile, ma solo stimabile.

Si ha che:

State of the system $s = (k, \lambda)$

- $1 \leq k \leq K^{max}$ Component parallelism
- λ avg. input rate
  - λ is discretized , i.e., $\lambda_i \in \{0, \Delta\lambda, 2\Delta\lambda, (L-1)\Delta\lambda\}$
  - $\Delta\lambda$ quantization step size, L number of discrete values

Available actions $\mathcal{A} = \{-1, 0, +1\}$

Transition probabilities $p(s'|s, a) = p\left((k', \lambda')|(k, \lambda), a\right)$

$$
\begin{aligned}
p(s'|s,a) &= P[s_{t+1} = (k', \lambda')|s_t = (k, \lambda), a_t = a] = \\
&= \begin{cases} P[\lambda_{t+1} = \lambda'|\lambda_t = \lambda] & k' = k + a \\ 0 & \text{otherwise} \end{cases} = \\
&= \mathbb{1}_{\{k'=k+a\}} P[\lambda_{t+1} = \lambda'|\lambda_t = \lambda]
\end{aligned}
$$

Come facciamo a conoscere questa *probabilità*? Possiamo stimarla usando una traccia storica, vedendo com'è variata λ in un certo periodo.

1.1.2 Funzione Costo

Definiamo tre componenti:

- $k + a$ è ciò che avrò al prossimo slot, normalizzo dividendo per $k_{\max}$.
- Nella seconda parte, premiamo o meno la *performance*, se abbiamo rispettato o meno il tempo di risposta. Dovremmo sapere le richieste, le repliche messe a disposizione, e relazionarle. A volte è difficile avere questi dati, e ciò porta a non poter sempre usare **MDP**.
- Paghiamo qualcosa se modifichiamo la *configurazione*.

Per tutti e tre abbiamo tre pesi w, che attribuiscono importanza relativa ai tre termini. Ad esempio, nel caso di riconfigurazione incide di meno rispetto a quello delle performance. Se non ci fosse una penalità sulla riconfigurazione, l'agente potrebbe cambiare configurazione sempre, ma sappiamo che tirar su altre macchine virtuali, passare risorse, etc… non è un qualcosa che si fa velocemente, quindi bisogna scalare quando sappiamo di poter mantenere la nuova configurazione per un certo lasso di tempo.

$$c(s,a,s') = w_{res}\frac{k+a}{K^{max}} + w_{perf}\mathbb{1}_{\{R(s,a,s')>R^{max}\}} + w_{rcf}\mathbb{1}_{\{a\neq 0\}}$$

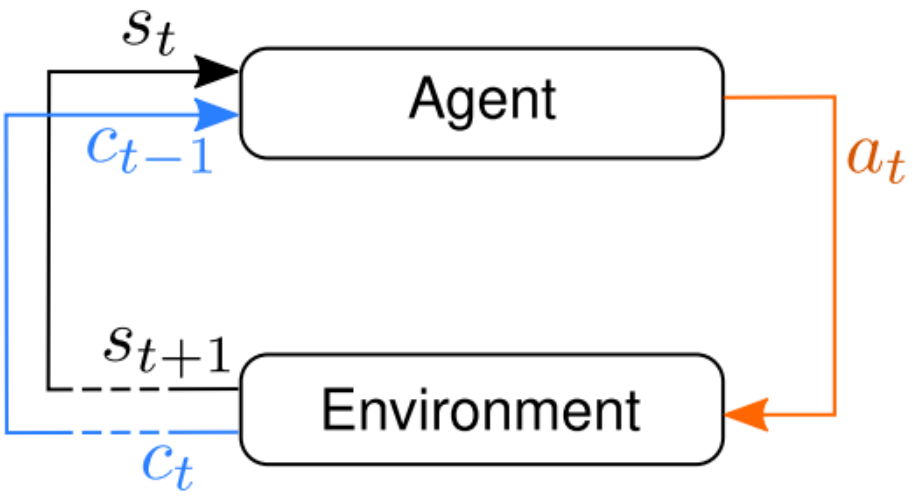
Resource Cost Performance Reconfiguratio

Si può vedere che, a seconda del valore dei pesi, si ha una configurazione più o meno *conservativa*.

1.2 Problema di MDP

Risolvere MDP richiede la totale conoscenza del modello sottostante, ma ciò non è sempre possibile. Spesso, la politica va *appresa tramite l'interazione con l'ambiente*, senza partire dal conoscere già una politica ottima. Questo funge da introduzione per il **Reinforcement Learning**.

1.3 Reinforcement Learning



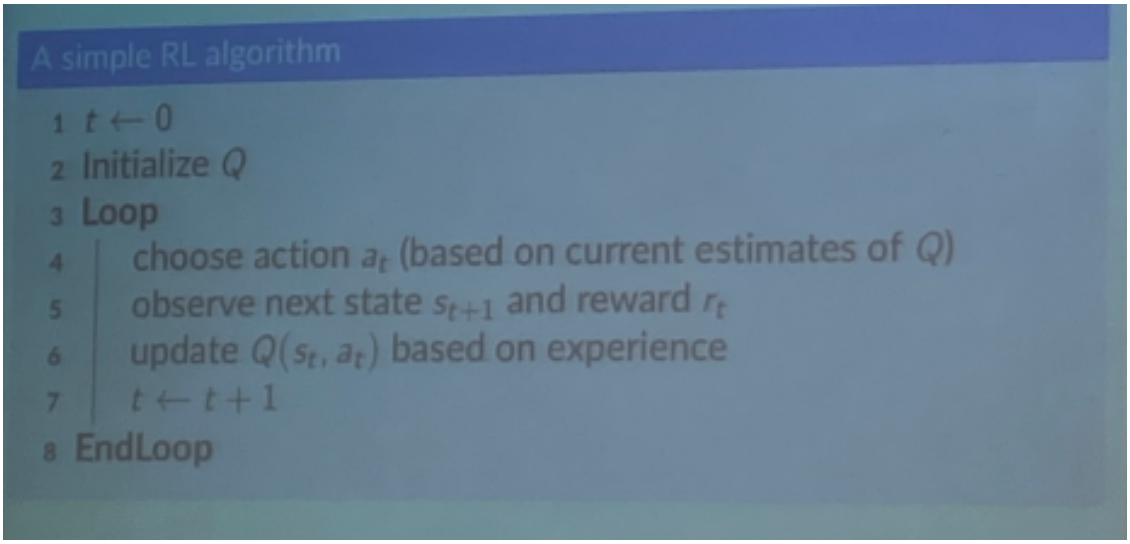
Abbiamo due modelli:

- **Model Free RL**: Nessun modello dell'ambiente, non sappiamo nulla, dobbiamo scoprire tutto, inclusa la politica ottima.
- **Model Based RL**: Abbiamo conoscenze parziali, magari si ha una funzione costo, anche se stimata. E' comunque meglio di nulla, per questo sfruttiamo reinforcement learning con algoritmi leggermente diversi. In questa famiglia rientrano anche i casi in cui si conoscono perfettamente tutti i parametri. Allora perchè applicarlo qui? Basti pensare ad un *labirinto senza uscita*, perchè il reinforcement learning apprenderà sulla porzione di stati rilevante (i.e: dove effettivamente posso muovermi), mentre MDP sarebbe più complesso, e con informazioni su stati che non attraveso mai. Lo stesso ragionamento è con un *labirinto infinito*, dove con MDP dovremmo apprendere infiniti stati.

Abbiamo un'altra classificazione:

- **Value-based RL**: Il loro scopo è calcolare politica ottima tramite *value function*. La policy viene di conseguenza. E' un algoritmo RL semplice, che vedremo.
- **Policy based RL**: Non sono interessati alla value function, ma la distribuzione di probabilità delle azioni dato uno stato, ovvero la politica ottima.
- Esiste un terzo approccio **hybrid**.

1.4 Simple value-based RL algorithm



1. t $\rightarrow 0$
2. Inizializza Q.
3. **Loop**
4. Scegli un'azione a_i basata sulle stime attuali di Q per lo stato corrente
5. Osserva lo stato successivo s_{t+1} e la ricompensa r_t derivante dall'azione presa
6. Aggiorna il valore $Q(s_t, a_t)$ basandoti sull'esperienza
7. $t \leftarrow t + 1$
8. **EndLoop**

Dato lo stato Q, scegliamo a_i, osserviamo lo stato s_{t+1} e il reward r_t, portando quindi ad un aggiornamento di Q.

1.4.1 Q-learning Action Selection

Converge alla politica ottima, ma dopo n steps, potenzialmente non finiti.

1.4.2 Come scegliamo un'azione ad ogni step?

Supponiamo che, dopo aver inizializzato Q, allo stato s_1, abbiamo tre azioni, $a_1 = 7$, $a_2 = 0$ e $a_3 = 0$. Se volessimo massimizzare il *reward*, andremmo su a_1, ma le altre due non le abbiamo ancora esplorate, non ci siamo mai andati sopra. Se fossero più redditizie? Ciò porta al dilemma tra *exploration* e *exploitation*, dove nel primo caso valorizziamo la ricerca per conoscere meglio l'ambiente; nel secondo caso preferiamo l'azione conosciuta.

Non possiamo fare *sempre* l'esplorazione, o sempre l'exploitation. Bisogna bilanciare. Inoltre questo algoritmo converse se tutte le coppie *stato-azioni* sono visitate infinite volte al tempo $t \to$ infinito.

1.4.3 ϵ-Greedy Exploration

Con probabilità $(1-\epsilon)$ scegliamo l'azione greedy, con probabilità ϵ, scegliamo un'azione a caso **tra tutte**. Per n all'infinito, faremo quindi alcune scelte casuali. All'inizio esploro di più, andando avanti descresce.

1.4.4 Softmax Action Selection

Evita il problema dell'iperparametro ϵ. Si usa un ragionamento simile alla *softmax*. Abbiamo una stima di Q (quello visto prima, con stati ed azioni). Applicando softmax, verrebbero associate probabilità a quei valori legati alle azioni. Proprio questo è ciò che viene fatto:

$$\pi(a|s) = \frac{exp(Q(s,a)/\tau)}{\sum_{a' \in \mathcal{A}} exp(Q(s,a')/\tau)}$$

Quindi, l'azione che valeva 7, avrà una certa probabilità, mentre quelle che valevano 0, non avrano probabilità 0, bensì > 0, quindi sono sempre considerabile. Ovviamente, in questo caso, la probabilità associata a 7 preverrà, e per evitare ciò si introduce la **temperatura** τ, tale che:

- all'inizio, un τ è grande, e porta ad una scelta randomica. All'inizio infatti vorremmo esplorare.
- andando avanti, τ è piccola (arriva ad 1), preferendo un comportamento greedy.

Quindi τ parte grande e decresce andando avanti.

2 Q-learning: Updating Q

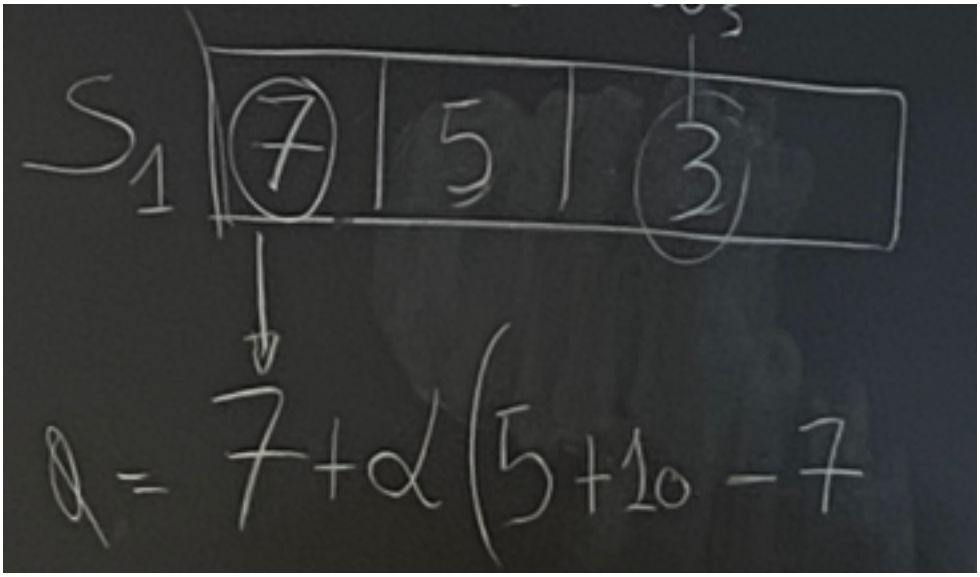
Sappiamo che, con modelli noti, Q è computabile iterativamente come:

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

Il **Q-learning** usa **punti stimati dall'esperienza**, cioè una stima puntale basata su $\{s_t, a_t, c_t, s_{t+1}\}$

$$Q(s_t,a_t) \leftarrow Q(s_t,a_t) + \underbrace{\alpha_t}_{\text{Learning Rate}} \left[\underbrace{r_t + \gamma \max_{a' \in \mathcal{A}} Q(s_{t+1},a')}_{\text{Target}} - Q(s_t,a_t) \right]$$

Siamo in s_1, compio azione a_1, e riceviamo *reward*, che è variabile aletatoria perchè l'agente del RL non sa quanto sarà, ma comunque sappiamo che è un valore atteso. Diciamo $r_t = 5$, non sappiamo se ci è andata bene. Potremmo finire in svariati stati $s_1, ..., s_n$,con una probabilità e stima sul costo futuro. Nel Q-learning non sa nulla degli stati futuri, non sa le probabilità. Ha visto solo che è entrato nello stato s_1, si basa solo su quello, per calcolare la stima in arancione, considera il reward *campionato* r_t (non atteso), il massimo tra le Q partendo dallo stato s_{t+1} e per Q prende la transizione che ha fatto adesso, non tiene conto di ciò che poteva succedere ma non è successo. Poi sottrae la Q attuale. Tutte le componenti nelle parentesi quadre, dopo α_t, mi indicano l'**errore**, ed incide in piccola parte sul risultato finale. Facendolo tante volte, r_t verrà aggiornato ogni volta, e tenderà al valore medio. Campionando la transizione, si avrà che per infinite volte, convergiamo alle probabilità reali.



Possiamo anche vederla come:

$Q(s, a) \leftarrow (a - \alpha) \cdot Q(s, a) + \alpha[r_t + \gamma \max_{a'} Q(s', a')]$

dove $Q(s, a)$ è attuale, max$Q(s', a')$ è una stima, influenzata da ciò che campiono, quindi peso per un qualcosa che incide poco. Ignoro probabilità e reward non deterministici, quando ho stima puntale la uso, convergendo ai valori attesi.

2.1 Algoritmo Q-learning

Otteniamo:

Q-learning

1 $t \rightarrow 0$
2 Initialize Q (e.g., zero-initialized)
3 **Loop**
4 choose a_t (e.g., ϵ-greedy or softmax selection)
5 observe next state s_{t+1} and reward r_t
6 $Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha_t \left[r_t + \gamma \max_{a' \in \mathcal{A}} Q(s_{t+1}, a') - Q(s_t, a_t) \right]$
7 $t \leftarrow t + 1$
8 **EndLoop**

L'occupazione di memoria dell'algoritmo, è $O(\#S \times \#A)$, ovvero prodotto tra spazio stati e spazio azioni, questo è dovuto alla memorizzazione di Q, mantenente sia lo stato sia le azioni, oltretutto modificando una sola componente alla volta.

2.2 Algoritmi ON-policy e OFF-policy

Q-learning è di tipo *off-policy*, in quanto durante l'aggiornamento di Q assume una certa politica di selezione (*greedy* nello specifico), mentre quando deve scegliere un'azione usa un'altra politica (come la *softmax*, ad esempio). Ciò si traduce nel dire che nella stima dei costi futuri, a partire dallo stato S', c'è l'assunzione che le azioni siano scelte privilegiando la migliore (ovvero, arrivato in S', preferisco l'azione a' che massimizza la Q, quindi esplora poco). La scelta dell'azione è invece fatta in maniera più esplorativa. Quindi ciò che fa è diverso da ciò che stima, esplora poco perchè prediligie l'azione più remunerativa ora.

Gli algoritmi *on-police* usano la stessa politica sia per le azioni, sia per aggiornare la Q. Se la scelta dell'azione è random, lo sarà anche la stima dei costi futuri.

L'algoritmo **SARSA** è *on-policy*, in cui la stima del costo futuro è dipendente dalla scelta che farò nel prossimo stato. Ovvero, osserva lo stato successivo, sceglie l'azione da fare nel prossimo stato (con le stesse tecniche *softmax* o ϵ-*greedy* viste prima). Quando stima la Q, la stima dei costi futuri non è data dal massimo delle azioni che posso prendere nel nuovo stato, ma la stima di a_{t+1} in s_{t+1}, ovvero la stima del costo futuro sapendo che io ho scelto una certa azione nel prossimo stato. Non il migliore dei casi futuri dato uno stato, ma unicamente in funzione dell'azione scelta, anche esplorativa.

Converge meglio, ma vincolato dalla scelta di una *specifica politica*. In Q-learning, potremmo scegliere 100 azioni casuali, salvarle, e poi aggiornare Q in base a queste informazioni. Con *SARSA* siamo più rigidi.

NB: Il nome **SARSA** deriva dalle 5 componenti che compongono la regola di aggiornamento, ovvero: stato s, azione a, reward r, stato successivo s_{t+1} e azione successiva a_{t+1}.

2.2.1 Osservazione esempio Labirinto

Esistono vari metodi di reward, ad esempio:

- Reward unicamente se si arriva la fine, non importa come ti sposti o quanto ti sposti.
- Premiare, con un reward positivo, l'uscita dal labirinto, e penalizzare (reward negativo) un crash.

Anche se rispetto alla scelta iniziale, funzionano leggermente peggio. Ciò che non specifico con reward (sia positivi che negativi) è più difficile da apprendere.

[ ]: