Lez23_MDP2

December 15, 2023

1 MDP parte 2

Abbiamo considerato *rappresentazioni tabulari*, ovvero cose del tipo:

State/Action	a_1	a_2	...
s_1	$Q(s_1, a_1)$	$Q(s_1, a_2)$	...
s_2	$Q(s_2, a_1)$	$Q(s_2, a_2)$	...
...	...		
s_n	$Q(s_n, a_1)$	$Q(s_n, a_2)$	...

Questa va bene per piccole dimensioni, però la domanda cresce come $O(|S| \times |A|)$, inoltre è poco generalizzabile, ovvero ciò che accade per uno stato, vale solo per quello stato, non è "utilizzabile" da altri, cioè altri stati, aventi caratteristiche simili, non possono apprendere. In ultima istanza, è difficile lavorare in casi di spazi *continui*, in quanto la tabella stessa, per definizione, è discreta. Nell'autoscaling, avevamo affrontato la quantità λ, risolvendo il tutto *discretizzando*, quindi se ottenevamo un valore 3.1, lo associavamo a 3 piuttosto che a 4.

2 Approssimazione con Value Function

L'idea è usare un'*approssimazione parametrica* della value function, ovvero:

$$V_\pi(s) \approx \hat{V}(s, \boldsymbol{w}), \text{ or}$$
$$Q_\pi(s, a) \approx \hat{Q}(s, a, \boldsymbol{w})$$

dove:

- $w \in \mathbb{R}^d$ è il vettore di parametri.
- Memorizziamo w e non la table Q, riducendo la domanda di memoria richiesta.
- E' più generalizzabile, in quanto l'esperienza è usata per aggiornare w, quindi un singolo update può impattare il valore di altri stati, oltre che usabile in contesti di spazi continui.

La *Value function* indica ritorno atteso, non dipende dalla politica deterministica, in quanto vale per qualsiasi tipo di politica.

La Action value function ci dice che essendo un certo stato, e compiendo una certa azione, *mediamente* abbiamo un certo ritorno.

Possiamo vedere, ad esempio, $\bar{V}(s, w) = W_0 + x \cdot w_1 + y \cdot w_2$

Noi però vogliamo approssimare una funzione che non conosciamo, il che è simile al Q-learning, in cui scoprivo grazie all'esperienza. Qui non è molto diverso. Quindi il tutto assomiglia a regressione, ma è sbagliato parlare di regressione, bensì di *Function Approximation*. Perchè tale differenza?

- Nella *Regressione*, abbiamo set limitato di dati, e vogliamo approssimare anche per punti dove non abbiamo dati.
- Con *Function Approximation*, possiamo avere anche già pronta la funzione di approssimare, generalmente i punti li conosciamo, dobbiamo solo trovare una funzione che li approssima.

Ciò che dobbiamo fare, in ordine, è:

- Scegliere $\bar{Q}$.
- Determinare il valore di w
- Cercare una funzione e il vettore w per approssimare V (o Q) nel miglior modo possibile.

La prima funzione che possiamo considerare è l'*errore quadratico medio*, ovvero:

$$J(\boldsymbol{w}) = \sum_{s \in \mathcal{S}} \mu(s) \left[V_\pi(s) - \hat{V}(s, \boldsymbol{w}) \right]^2$$

dove $\mu(s) \geq 0$ è una distribuzione tra gli stati, riflettendo l'importanza (o la frequenza) degli stati. Cioè quanto vanno pesati.

Presa questa funzione, come aggiorniamo i parametri w ? Usiamo la **discesa del gradiente**, associato a funzione costo, muovendomi in direzione opposta.

$$\begin{aligned} \boldsymbol{w}_{t+1} &= \boldsymbol{w}_t - \frac{1}{2} \alpha \nabla_{\boldsymbol{w}} J(\boldsymbol{w}_t) = \\ &= \boldsymbol{w}_t + \alpha \sum_{s \in \mathcal{S}} \mu(s) \left[V_\pi(s) - \hat{V}(s, \boldsymbol{w}) \right] \nabla_{\boldsymbol{w}} \hat{V}(s, \boldsymbol{w}) \end{aligned}$$

Però, sorgono due problemi:

1. La somma è tra tutti gli stati, può essere costosa.
2. Non abbiamo i valori veri di $V_\pi(s)$, a meno che non abbiamo usato un altro algoritmo per calcolare la tabella. Se così fosse, tutte queste operazioni sarebbero superflue.

2.0.1 Come risolviamo questi problemi?

Per il primo problema, usiamo **stochastic gradient descent**:

$$\boldsymbol{w}_{t+1} = \boldsymbol{w}_t + \alpha \sum_{s \in \mathcal{S}} \mu(s) \left[V_\pi(s) - \hat{V}(s, \boldsymbol{w}) \right] \nabla_{\boldsymbol{w}} \hat{V}(s, \boldsymbol{w})$$

Gradient computed over all states...

Stochastic gradient descent

one (or few) samples $(s_t, V_\pi(s_t))$ at each step

$$\boldsymbol{w}_{t+1} = \boldsymbol{w}_t + \alpha \left[V_\pi(s_t) - \hat{V}(s_t, \boldsymbol{w}) \right] \nabla_{\boldsymbol{w}} \hat{V}(s_t, \boldsymbol{w})$$

Per il secondo problema, come facciamo? Non abbiamo quel valore, perchè lo stiamo imparando. Possiamo sostituirlo con una stima U_t.

Quindi il calcolo diventa il seguente:

$$\boldsymbol{w}_{t+1} = \boldsymbol{w}_t + \alpha \left[U_t - \hat{V}(s_t, \boldsymbol{w}) \right] \nabla_{\boldsymbol{w}} \hat{V}(s_t, \boldsymbol{w})$$

A possible approach (inspired by Q-learning):

$$U_t = r_t + \gamma \hat{V}(s_{t+1}, \boldsymbol{w}_t)$$

E si parla di **Stochastic SEMI-gradient descent**. Quando osserviamo reward, e li mettiamo nella formula, i pesi si aggiornano per predizioni future.

Miglioriamo stima corrente sfruttando reward e storia passata pesata per α.

La stima futura è basata sui parametri che ho attualmente, non posso fare di meglio, e per questo uso $\bar{V}$.

2.1 Linear Function Approximation

E' l'approssimazione più semplice.

$$\hat{V}(s, \boldsymbol{w}) = \boldsymbol{w}^T \boldsymbol{\phi}(s) = \sum_{i=1}^{d} w_i \phi_i(s)$$

Weights

$\boldsymbol{w} \in \mathbb{R}^d$

Features

$\boldsymbol{\phi} : \mathcal{S} \to \mathbb{R}^d$

in quanto semplifica la regola di aggiornamento:

$$\nabla_{\boldsymbol{w}} \hat{V}(s, \boldsymbol{w}) = \boldsymbol{\phi}(s)$$

$$\boldsymbol{w}_{t+1} = \boldsymbol{w}_t + \alpha \left[U_t - \hat{V}(s_t, \boldsymbol{w}_t) \right] \boldsymbol{\phi}(s_t)$$

2.1.1 Esempio

Per l'approssimazione, potremmo assumere che due informazioni importanti per $S = (x, y)$ possano essere: $\phi_1(s) = x$ e $\phi_2(s) = x - y$

Allora $\bar{V}(s) = w_0 + w_1\phi_1(s) + ...$

2.1.2 Equivalenza per Q

Per Q, valgono gli stessi ragionamenti:

$$\hat{Q}(s, a, \boldsymbol{w}) = \boldsymbol{w}^T \boldsymbol{\phi}(s, a) = \sum_{i=1}^{d} w_i \phi_i(s, a)$$

Weights
$w \in \mathbb{R}^d$

Features
$\boldsymbol{\phi} : \mathcal{S} \times \mathcal{A} \to \mathbb{R}^d$

$$\nabla_{\boldsymbol{w}} \hat{Q}(s, a, \boldsymbol{w}) = \boldsymbol{\phi}(s, a)$$

$$\boldsymbol{w}_{t+1} = \boldsymbol{w}_t + \alpha \left[U_t - \hat{Q}(s_t, a_t, \boldsymbol{w}_t) \right] \boldsymbol{\phi}(s_t, a_t)$$

??

Non sappiamo però U_t, per questo entra in gioco la funzione di approssimazione lineare:

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

Noi compiamo un'azione, otteniamo esperienza (cioè reward + stato successivo) ed otteniamo un nuovo Q(s,a). Per stimare il valore dello stato successivo, ora utilizzeremo l'aapprossimazione di Q, non tabella come prima.

$t \rightarrow 0$
Initialize w
Loop
  choose action a_t
  gather experience $\langle s_t, a_t, r_t, s_{t+1} \rangle$
  $U_t \leftarrow r_t + \gamma \max_{a' \in \mathcal{A}} \hat{Q}(s_{t+1}, a', w_t)$
  $w_{t+1} = w_t + \alpha \left[U_t - \hat{Q}(s_t, a_t, w_t) \right] \phi(s_t, a_t)$
  $t \leftarrow t + 1$
EndLoop

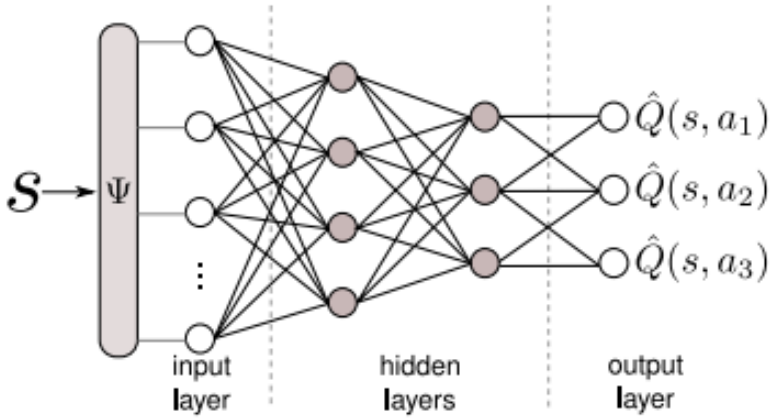
Tuttavia, non abbiamo certezza sulla convergenza. Inoltre, tale approccio è fortemente dipendente da come vengono rappresentati stati ed azioni, il che richiede la valutazione di un "esperto" nel campo in esame.

Per questo motivo, rientra in gioco il **Deep Learning**, avente il vantaggio di non richiedere in input le feature ottimali.

3 Deep Q Network

L'idea alla base è proporre una rete che approssimi Q function mediante deep learning. In input prende uno stato s, possibilmente pre-processato, quindi una sua trasformazione, ma non mirata a trovare feature migliori, solo per semplificare il tutto. In ouput, per *ogni azione* a, si ha $\bar{Q}(s, a)$.

Può funzionare con *spazi di stato continui*, purchè approssimati. Ad esempio, la velocità di un'auto ha range continuo, e tra 59 k/h o 60 k/h non cambia nulla. Non possiamo lavorare su *spazi di azioni continui*, in quanto per ciascuno si associa una funzione.



Non era più semplice prendere <*stato,azione*> e in output fornire un unico valore?

Non va bene per ragioni di efficienza: Quando scegliamo azione, vediamo per un certo stato tutte le azioni possibili, e prendiamo quella con *reward maggiore*. Quindi ha più senso predirre tutte le azioni e poi scegliere il massimo, invece che ottenerne una in output alla volta. Più complesso, ma fornisce benefici.

Ovviamente ci serve sempre un **training**: Sarebbe esoso esplorare tutti gli stati, l'unica cosa che abbiamo è l'agente che ottiene reward "at-the-moment" (o anche detto *on-line*), cioè mentre compie azione, quindi con l'esperienza.

Experience	**Training Sample**
$\langle s_t, a_t, s_{t+1}, r_t \rangle$	$(s_t, a_t) \rightarrow r_t + \gamma \max_{a'} \hat{Q}(s_{t+1}, a', \boldsymbol{w})$
$\langle s_{t-1}, a_{t-1}, s_t, r_{t-1} \rangle$	$(s_{t-1}, a_{t-1}) \rightarrow r_{t-1} + \gamma \max_{a'} \hat{Q}(s_t, a', \boldsymbol{w})$
$\langle s_{t-2}, a_{t-2}, s_{t-1}, r_{t-2} \rangle$	$(s_{t-2}, a_{t-2}) \rightarrow r_{t-2} + \gamma \max_{a'} \hat{Q}(s_{t-1}, a', \boldsymbol{w})$
...	

Tutte le azioni che l'agente compie, formano l'esperienza. (nell'immagine, è "experience"). Quindi possiamo fare la stima, includendo la stima dei costi futuri, per s_{t+1} sfruttando la rete neurale stessa, sommando per r_t. ("training sample"). Man mano che l'agente compie azioni ed ottiene reward, facciamo un mini batch e facciamo iterazione per aggiornare i pesi della rete. Tuttavia queste azioni incluse nel *mini-batch* sono dipendenti tra loro. Se nel labirinto ho un vicolo cieco, azioni future sono fortemente influenzate da dove mi trovo ora. Inoltre, se mi allontano nel labirinto, "dimentico" ciò che avevo all'inizio.

La soluzione è **Experience replay**, ovvero usare un *buffer circolare finito FIFO* , cioè memorizza le ultime B osservazioni. Come lo alleno?

Quando voglio addestrare rete su minibatch, prendo *casualmente* delle tuple da questo buffer, ovvero sia esperienze recenti, sia passate, togliendo la correlazione tra loro, oltre alla possibilità di prendere un dato più volte nel training, migliorando la *data efficiency*. L'algoritmo è una variante del Q-learning:

```
Initialize w
Initialize empty buffer B
i ← 0
Loop
    choose action a_i
    gather experience ⟨s_i, a_i, r_i, s_{i+1}⟩ and add to B
    sample minibatch of b ⟨s_j, a_j, r_j, s_{j+1}⟩ tuples from B
    y^(j) ← r_j + γ max_{a'} Q̂(s_{i+1}, a', w), j = 1, ..., b
    L^(j) = (y^(j) − Q̂(s_j, a_j, w))^2          /* Loss */
    update w using, e.g., SGD on the minibatch
    i ← i + 1
EndLoop
```

1. Inizializziamo i pesi randomicamente
2. Inizializzo buffer dell'esperienza
3. $i \leftarrow 0$
4. **In loop**
5. Scegliamo l'azione.
6. Otteniamo **esperienza** = $<s_i, a_i, r_i, s_{i+1}>$ da mettere in B.
7. Prende, a caso, delle tuple dal buffer.
8. Per calcolare stima usiamo questa espressione.
9. Calcoliamo la differenza tra il valore vero e la stima fatta mediante loss function.
10. Aggiorniamo w con back-propagation, usando SGD o Adam, etc..
11. $i \leftarrow i + 1$
12. **End Loop**

Nb: s_i = stato attuale, a_i = azione attuale, r_i = ricompensa attuale, s_{i+1} = stato successivo. Queste compongo l'**esperienza**.

Dopo molti step, verrà appresa la politica vicina alla politica ottima.

4 Target Network

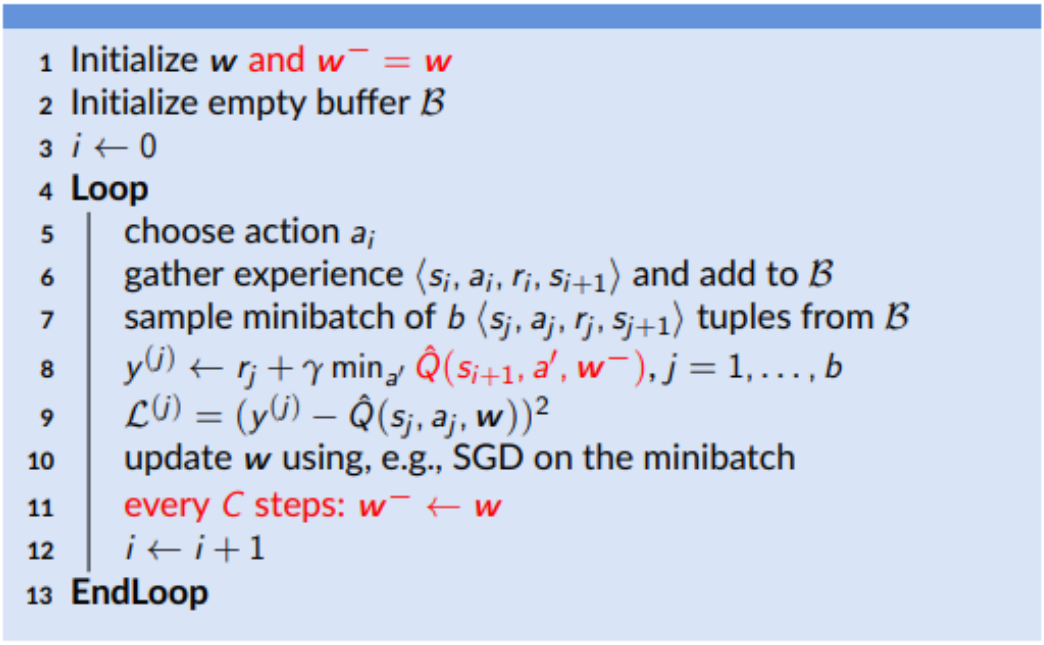
Può capitare che il training sia divergente. Nella rete neurale, abbiamo dei target di riferimento, e il training non viene cambiato. Un certo dato avrà un certo risultato. Nel nostro caso di **Deep Q-learning** ciò non avviene, ad ogni istante cambiano i target. La rete segue valori target che continuano a cambiare.

C'è sovrastima, perchè prendiamo il *max* in :

$y(j) \leftarrow r_j + \gamma \min_{a'} \bar{Q}(s_{i+1}, a', w)$,

bisogna trovare una soluzione.

L'idea è aggiornare *meno frequentemente* la rete target. La prima rete aggiorna w, la seconda Q, e periodicamente la seconda prende i valori della prima dopo ogni tot "passi". Ciò è visibile a riga 11, dove ogni C steps, prendiamo i valori w e li mettiamo in w^-.



Un riferimento interessante, fornito da *OpenAI* è il seguente:

https://www.gymlibrary.dev/environments/atari/

Qui vengono proposti alcuni esempi di giochi classici, con azioni ed agenti predefiniti.

Per il **Reinforcement Learning** non c'è una documentazione esaustiva, sia perchè recente, sia perchè è difficile applicarla a contesti totalmente diversi, e quindi fare una documentazione generale. Esiste però la libreria di TensorFlow **TF-Agents**, o implementare ex-novo l'algoritmo

https://www.tensorflow.org/agents/tutorials/1_dqn_tutorial?hl=en.