**UNIVERSITÀ DEGLI STUDI DI VERONA**
**DIPARTIMENTO DI INFORMATICA**

# STUDIO DI ALGORITMI DI APPRENDIMENTO PER RINFORZO SULLA PIATTAFORMA OpenAI

**22 Novembre 2018**

PROF.
ALESSANDRO FARINELLI

STUDENTE
GIACOMO FERRO

A.A. 2017/2018

**INDICE**

# INTRODUZIONE

Nel contesto dell'intelligenza artificiale, l'apprendimento per rinforzo ha come scopo quello di modificare il comportamento di un agente per imparare a compiere azioni opportune in un determinato ambiente. Questo apprendimento prevede l'interazione agente-ambiente e presuppone che l'ambiente invii dei segnali all'agente per indicare quando le azioni di quest'ultimo generano dei risultati positivi o negativi.

L'obiettivo del lavoro svolto durante questo tirocinio consiste nell'applicazioni di algoritmi di apprendimento basati su tecniche che tentano di costruire una associazione tra stati ed azioni senza stimare un modello dell'ambiente (Q-Learning) e tecniche che invece richiedono un modello dell'ambiente (Policy e Value iteration). I due approcci hanno delle differenze sostanziali ma possono essere confrontati sul piano delle performance dell'agente, considerando in particolare il valore accumulato dall'agente per valutare in quale caso si ottengono dei risultati migliori.

Per raggiungere tale scopo ho utilizzato la piattaforma OpenAI gym. Tale piattaforma sarà spiegata meglio in seguito e per ora ci limitiamo a dire che implementa vari ambienti di gioco categorizzati ed è molto utilizzata perché rappresenta una risorsa libera e supportata da vari sviluppatori per creare agenti di apprendimento per rinforzo. Gli ambienti da me scelti appartengono alla categoria Toy Text: è stata una delle primissime categorie sviluppate e contiene i giochi più semplici da analizzare.

Ho prodotto tre versioni sequenziali del software che esegue questi algoritmi su giochi di nome Taxi-v2 e Roulette-v0 presenti sulla piattaforma OpenAiGym. Tali releases riguardano:

1. Q-learning e Variante Sarsa
2. Policy e Value Iteration
3. Relativi grafici di confronto

Dai grafici mostrati in seguito, si evince che nel caso di ambiente non completamente stocastico (come nel caso di Toy Text con Taxi-v2) i risultati teorici migliori sono prodotti dalla iterazione delle politiche.

Nelle pagine seguenti verranno presentate le strutture degli algoritmi usati e successivamente verrà analizzata la struttura dell'ambiente su cui questi algoritmi verranno applicati.

In sostanza, questo lavoro vuole rappresentare una unione dei vari tentativi fatti da altri utenti (che hanno costruito varie tipologie di agenti RL su OpenAIgym) sotto una comune struttura di implementazione. Le tre versioni sono state integrate attraverso l'implementazione di altre tipologie di algoritmi e rappresentano quindi delle versioni via via migliorate.

# CAPITOLO PRIMO

Apprendimento per rinforzo

L'apprendimento per rinforzo permette ad un agente di acquisire una strategia di comportamento, in un certo ambiente, tramite dei tentativi che all'inizio potrebbero essere, ad esempio, scelti casualmente. Il feedback dell'ambiente che permetterà all'agente di capire quando è stata fatta una mossa favorevole o meno è chiamata **ricompensa o rinforzo.** Negli scacchi il rinforzo è la vittoria della partita, nel ping pong ogni punto è un rinforzo, in PacMan il rinforzo sono le ricompense distribuite sul campo di gioco. In genere sappiamo che le ricompense sono considerate un **input** percettivo dell'agente.

Quindi l'apprendimento per rinforzo ha lo scopo di trovare una politica ottima che permetta di massimizzare il numero di ricompense. Con politica intendiamo una serie di comportamenti che l'agente esegue per ottenere le massime ricompense possibili nell'ambiente.

Per eseguire questo tipo di apprendimento possiamo utilizzare tre tipi di agenti:

1. **agente basato sull'utilità** → apprende una funzione di utilità sugli stati per ottenere il punteggio massimo.
2. **agente Q-learning** → apprende una funzione azione-valore. Quindi esegue una serie di azioni per ritornare la massima ricompensa attesa.
3. **agente reattivo** → apprende una politica che mette in relazione stati e azioni.

L'agente Q-learning non necessità di un modello dell'ambiente, ovvero non ha bisogno di conoscere gli effetti delle azioni; però questo agente, non conoscendo gli effetti delle azioni (soprattutto a lungo termine) apprende le azioni ottime più lentamente.

Ci sono due tipi di apprendimento:

1. **apprendimento passivo** = politica dell'agente è fissa e l'agente cerca di capire l'utilità degli stati. Questo potrebbe richiedere la comprensione del modello dell'ambiente.
2. **apprendimento attivo** = l'agente deve imparare cosa fare. Questo è messo in atto tramite l'**esplorazione**. Ci sono vari metodi di apprendimento che saranno analizzati.

Per poter spiegare i metodi di apprendimento utilizzati è utile introdurre i **processi decisionali di Markov**. Definiamo in generale come modello di transizione (o modello) la specifica delle probabilità di tutti gli esiti delle azioni in ogni possibile stato.

Quindi: T(s,a,s') indica la probabilità di raggiungere lo stato s' partendo da s ed eseguendo una azione a. In genere si suppone sempre che la transizione sia Markoviana: ovvero che la probabilità di successo sia determinata dallo stato inziale s e dalla azione 'a' e non dalla storia degli stati e azioni passati. Poi, in ogni stato s, l'agente riceve una **ricompensa R(s)** che può essere positiva o negativa ma necessariamente limitata. Nella maggioranza dei casi parliamo un problema sequenziale (quindi l'utilità è influenzata dall'ordine degli stati) e, di conseguenza, **l'utilità della storia dell'ambiente è pari alla somma delle ricompense ricevute**.

Processo decisionale di Markov

La specifica di un problema di decisione sequenziale per un ambiente completamente osservabile con modello di transizione markoviano e ricompense additive prende il nome di **processo decisionale di Markov o MDP**.

Un MDP è definito da:

1. Stato iniziale: $S_0$
2. modello di transizione: T(s,a,s')
3. funzione di ricompensa: R(s)
4. Insieme degli stati S
5. Insieme delle azioni A

Non abbiamo mai una soluzione fissa e deterministica perché l'agente potrebbe non essere in grado di controllare totalmente l'ambiente. In altre parole, l'agente non conosce al 100% l'effetto delle sue azioni. **Quindi la soluzione deve specificare il comportamento dell'agente in ogni stato potenzialmente raggiungibile**. Una soluzione di questo tipo si chiama **politica**. Si usa il pi-greco $\pi$ o $\pi(s)$ per indicare l'azione raccomandata dalla politica $\pi$ nello stato s. Se la politica è completa l'agente saprà sempre quale azione eseguire indipendentemente dalle azioni precedenti.

Ogni volta che una politica viene eseguita, la natura stocastica dell'ambiente porta a risultati diversi e quindi ad una storia diversa. Una politica si dice **ottima** ($\pi^*$) se porta la più alta utilità attesa.

Definiamo l'utilità come $\mathbf{U_h([s_0, s_1,..., s_n])}$.

L'utilità cambia se abbiamo un orizzonte finito o infinito: ovvero se abbiamo un tempo limite o meno. Con un orizzonte finito l'azione ottima potrà cambiare col tempo quindi **la politica ottima non è stazionaria**. In questo caso la politica non dipende solo dallo stato corrente ma anche da quanto è il tempo rimanente. **Viceversa, la politica è stazionaria se l'azione ottima dipende solo dallo stato corrente**.

Assegnazione delle ricompense in caso di stazionarietà:

1. **Ricompense additive = $U_h([s_0, s_1,..., s_n]) = R(s_0)+R(s_1)+\ldots+R(S_n)$**
2. **Ricompense scontate = $U_h([s_0, s_1,..., s_n]) = R(s_0)+ \gamma R(s_1)+\ldots+ \gamma^n R(S_n)$** in cui gamma è numero compreso tra 0 e 1 ed è detto **fattore di sconto**. Quando il numero è vicino a zero le ricompense per quella serie di stati sono considerate insignificanti, mentre quando gamma vale 1 le ricompense sono identiche a quelle additive.

Ne consegue che le ricompense additive sono un caso particolare delle scontate quando gamma è sempre uguale ad uno. A questo punto dobbiamo pensare a come risolvere la questione delle ricompense e utilità infinite quando c'è la possibilità di non raggiungere mai uno stato terminale. **Abbiamo tre soluzioni a questo problema**:

1. Se abbiamo sequenze scontate l'utilità di una sequenza infinita risulta finita. In particolare se abbiamo che le ricompense sono limitate a $R_{max}$ e gamma < 1 allora: $U_h([s_0, s_1,..., s_n]) \sum_{t=0 \text{ a inf}} \gamma^t R(s_t) <= \sum_{t=0 \text{ a inf}} \gamma^t R_{max} = R_{max}/1-\gamma$.
2. Se l'ambiente contiene stati terminali ed è garantito che l'agente prima o poi ne raggiunga uno allora non avremo mai sequenze infinite. La politica che porta al raggiungimento di uno stato terminale si dice essere **politica propria.**
3. Un'ultima possibilità è quella di confrontare sequenze infinite in base alla **ricompensa media.** Infatti, se in un percorso abbiamo una casella A che ha come ricompensa 0,1 mentre tutte le altre danno 0,01 la politica che fa in modo di passare più volte possibili sulla casella A avrà una ricompensa media maggiore. Questo fatto delle volte può tornare utile.

Ora analizziamo come si fa a scegliere la politica migliore per un certo problema.

Ricordando che una politica genera un insieme di sequenze di stati ognuna con una particolare probabilità determinata dal modello di transizione dell'ambiente, **il Valore di una politica è la somma attesa delle ricompense scontate su tutti gli stati che potrebbero andare in esecuzione.**

Ora analizzeremo con quali algoritmi si possono determinare le politiche ottime quando il MDP è completamente specificato e quindi conosciamo il modello di transizione e la funzione di ricompensa.

<u>Iterazione dei valori</u>

L'idea base è calcolare l'utilità di tutti gli stati e quindi sfruttare tale informazione per selezionare l'azione ottima in ogni stato. Quindi ora vogliamo calcolare **l'utilità degli stati.** In generale l'utilità di uno stato è uguale all'utilità attesa delle sequenze di stati che potrebbero seguirlo. Ovviamente le sequenze sono stabilite dalla politica e quindi definiamo $U^{\pi}(s)$ come l'utilità dello stato 's' nella politica $\pi$. Inoltre, se indichiamo

con $s_t$ lo stato in cui si trova l'agente dopo aver lanciato $\pi$ per t passi abbiamo che $\mathbf{U^{\pi}(s) = \sum_{t=0\ a\ inf} \gamma^t\ R(s_t)|\pi,\ s_0=s}$ come il valore massimo ottenuto sommando le ricompense scontate secondo la politica scelta. L'utilità migliore si avrà quindi eseguendo $U^{\pi^*}(s)$ ovvero è la somma delle ricompense scontate ottenute dall'esecuzione della politica ottima.

N.B. R(s) è ricompensa a breve termine per il fatto di trovarsi in 's'. U(s) è la ricompensa finale a lungo termine da s in avanti.

La funzione di utilità U(s) permette di massimizzare l'utilità attesa degli stati successivi: $\pi^*(s) = \text{argmax}_a \sum_{s'} T(s,a,s')\ U(s')$.

Quindi se l'utilità di uno stato è la somma delle ricompense scontate da quello stato in avanti; ci deve essere una relazione diretta con quella dei suoi vicini. L'utilità di uno stato è la somma tra la ricompensa immediata e l'utilità scontata attesa dello stato successivo, presumendo che l'agente esegua l'azione ottima.

Quindi abbiamo **che $\mathbf{U(s) = R(s) + \gamma\ max_{a\ (a\ è\ l'azione)} \sum_{s'} T(s,a,s')\ U(s')}$.** Tale equazione è l'equazione di Bellman.

Ora guardiamo all'algoritmo di iterazione dei valori.

Tale algoritmo usa n equazioni di Bellman poiché ci sono n stati. Ogni funzione di utilità è una incognita quindi abbiamo da calcolare la parte destra e sostituirla nella parte sinistra. Sia $U_i(s)$ il valore dell'utilità dello stato s alla i-esima iterazione: questo è detto **aggiornamento di Bellman:** $U_{i+1}(s) = R(s) + \gamma\ \text{max}_a \sum_{da\ s'} T(s,a,s')\ U_i(s')$.

L'idea base è che prima o poi tutto finisca per propagazione dell'informazione finchè non si trovano l'unica soluzione ottime. In particolare, avremo una convergenza verso il valore di utilità ottimo per come è costruita l'equazione di Bellman.

***function*** *ITERAZIONE_VALORI (mdp , ε)* ***returns*** *funzione di utilità*

> ***inputs****: mdp, un MDP con stati S, modello di transizione T, funzione di ricompensa R e sconto γ mentre ε è l'errore massimo consentito nell'utilità di ogni stato*
>
> ***variabili locali****: U, U' sono vettori di utilità per gli stati in S, inizialmente a zero, mentre δ è la massima variazione di utilità per qualsiasi stato ad ogni iterazione.*
>
> ***repeat***
>
> *U=U'; δ=0*

***for each*** *stato s* ***in*** *S* ***do***

*U'[s] = R[s] + γ max $_a$ $\sum_{da\ s'}$ T(s,a,s')U[s']*

***if*** *|U'[s]-U[s]| > δ* ***then*** *δ = |U'[s]-U[s]|*

***until*** *δ < ε(1-γ)/γ*

***return*** *U*

Iterazione delle politiche

Partendo da una politica $\pi_0$ questo algoritmo alterna due passi fondamentali:

a. **valutazione della politica** = data una politica πi si calcola Ui = U(πi) come l'utilità di ogni stato qualora si dovesse eseguire la politica πi
b. **miglioramento della politica =** si calcola una nuova politica U ($\pi_{i+1}$) scegliendo gli stati successivi in base a U(i).

L'algoritmo termina quando il passo di miglioramento della politica non porta a nessun miglioramento dell'utilità. A questo punto sappiamo che la funzione di utilità Ui è un punto fisso dell'aggiornamento di Bellman e quindi πi sarà una politica ottima.

In generale ricordiamo che all' i-esima iterazione, la politica πi nello stato s produce l'azione πi(s). Quindi l'azione a = πi(s). **In definitiva Ui(s) = R(s) + γ Σ da s' T(s, πi(s),s') Ui(s).**

Prima di presentare il codice specifichiamo che le equazioni generate sono lineari poiché l'operatore max è stato rimosso.

***function*** *ITERAZIONE_POLITICHE(mdp)* ***returns*** *una politica*

***inputs****: mdp, un MDP con stati S e modello di transizione T*

***variabili locali****: U, U' sono vettori di utilità per gli stati in S, inizialmente a zero e π un vettore di politiche indicizzate per stato, inizialmente casuali.*

***repeat***

*U=VALUTAZIONE_POLITICA(π,U,mdp)*

*(tale funzione calcola la politica migliore per l'iterazione corrente)*

*immutata?=true*

***for each*** *state s* ***in*** *S* ***do***

***if*** *max $_a$* ***Σ*** *da s' T(s,a,s')U[s'] >* ***Σ*** *da s' T(s,π[s],s')U[s']* ***then***

*π[s] = argmax $_a$* ***Σ*** *da s' T(s,a,s')U[s']*

*immutata?=false*

***until*** *immutata?*
***return*** *π*

Apprendimento di una funzione azione-valore senza modello: Q-learning

Questo è un modello alternativo rispetto ai due precedenti ed è chiamato **Q-learning.** Permette di acquisire i Q-valori per la coppia azione-valore. **Esso non apprende le utilità ma le rappresentazioni azione-valore.** Quindi Q(a,s) indicherà il valore dell'esecuzione dell'azione a nello stato s. Da ciò: U(s) = max a Q(a,s) dove semplicemente l'utilità di uno stato s corrisponde al valore massimo di una certa azione svolta in quello stato.

La cosa fondamentale di tutto ciò è che questo agente **non necessità di un modello né per l'apprendimento né per la scelta delle azioni.** Il Q-LEARNING è definito infatti privo di modello. I Q valori fanno riferimento alla seguente equazione:

$Q(a,s) = R(s) + \gamma \sum$ da s' $T(s,a,s') \max a' Q(a',s')$

La tabella T viene ricalcolata ogni volta che l'azione 'a' è eseguita nello stato s con il risultato il passaggio nello stato s'. Il codice seguente implementa il Q-LEARNING tenendo una tabella che traccia le statistiche per le frequenze. Se usiamo un approccio random (ovvero l'agente esegue azione casuali in una certa frazione di passi che decresce col tempo) allora la tabella non sarebbe necessaria e solo in questo caso sarebbe veramente un apprendimento model-free.

***function*** *AGENTE_Q_LEARNING(percezione)* ***returns*** *azione*

***inputs****: percezione, indica lo stato corrente s' e ricompensa r'*
***static****:*
*Q è una tabella di valori di azioni indicizzata per stato e azione*
*$N_{as}$ è una tabella di frequenze di coppie di stato-azione*
*s,a,r sono lo stato, azione,ricompensa precedenti. All'inizio sono null*
***if*** *s non è null* ***then do***
    *scegli azione a da stato s usando politica derivata da Q[a,s]*
    *incrementa $N_{sa}$[s,a]*
    *$Q[a,s] = Q[a,s] + \alpha(N_{sa}[s,a])(r + \gamma \max a' Q[a',s'] - Q[a,s])$*
***if*** *TERMINALE?[s']* ***then*** *s,a,r = null*
***else*** *s,a,r = s', argmax a' Q[a',s']$N_{sa}$[a',s'], r*
***return*** *a*

Apprendimento di una funzione azione-valore senza modello: Variante SARSA

La principale differenza rispetto al Q-learning è che la ricompensa per lo stato successivo non è usata per aggiornare la tabella dei Q-valori. Quello che avviene è che una nuova azione e la ricompensa sono selezionate dalla stessa politica che ha determinato l'azione precedente.

*PSEUDOCODICE:*

***inizializza*** *Q(s,a) random*
***repeat*** *(per ogni trial):*
*inizializza s*
*scegli azione a da s usando politica derivata da Q*
***repeat****:*
***esegui*** *a osservando r,s'*
*scegli a' da s' usando politica derivata da Q*
***aggiorna*** *tabella Q(s,a)*
*s=s' e a=a'*
***until*** *done*

## CAPITOLO SECONDO

Piattaforma OpenAIGym

Sappiamo che l'apprendimento del rinforzo presuppone che ci sia un agente che si trova in un ambiente. L'agente intraprende un'azione, riceve un'osservazione e una ricompensa dall'ambiente. Un algoritmo RL cerca di massimizzare una certa misura della ricompensa totale dell'agente, in quanto l'agente interagisce con l'ambiente.

OpenAIGym è una piattaforma open-source che fornisce una serie di strumenti utili per sviluppare e testare gli algoritmi di apprendimento. Essa implementa molti ambienti che mostrerò in seguito. Si concentra sull'impostazione episodica dell'apprendimento per rinforzo, in cui l'esperienza dell'agente è costruita in una serie di episodi. In ogni episodio, lo stato iniziale dell'agente viene generato a caso da una distribuzione di probabilità e l'interazione con l'ambiente procede finché l'ambiente non raggiunge uno stato terminale. L'obiettivo è comunque sempre lo stesso: quello di massimizzare l'aspettativa di ricompensa totale per episodio nel minor numero possibile di episodi.

Come già anticipato, i due concetti fondamentali sono l'agente e l'ambiente. L'ambiente viene implementato dalla piattaforma e non è di facile modifica. Quello che gli utenti possono scrivere facilmente è l'agente che opera in un certo ambiente. Questa scelta è stata fatta per massimizzare la comodità per gli utenti e consentire loro di implementare diversi stili di interfaccia agente.

Precisiamo che la performance di un algoritmo può essere misurata su due fronti: il primo è la performance finale, il secondo è il tempo impiegato per apprendere la politica. La performance è misurata in genere con il calcolo della media delle ricompense ottenute in un episodio mentre il tempo di apprendimento può essere misurato in molti altri modi in OpenAIGym: il conteggio del numero di episodi sotto una soglia di punteggio oppure il tempo impiegato per completare una serie di episodi. Tale soglia è scelta ad hoc in base al tipo di ambiente in questione. Ovviamente gli algoritmi possono scegliere la via di apprendimento migliore concentrandosi sul tempo di calcolo o sulle ricompense.

Ora mostriamo quali sono le cose da sapere per poter utilizzare correttamente questa piattaforma.

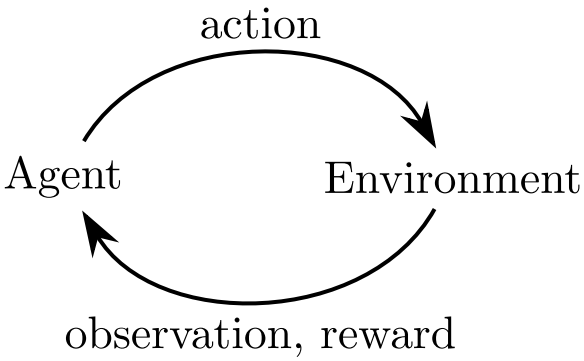
Observation space

Esistono tre funzioni principali. Una di queste, la funzione **step()**, è la funzione che permette di eseguire un'azione ritornando all'agente un feedback su cosa l'azione ha fatto all'ambiente. Tale feedback è detto "osservazione" o meglio **"observation".** Tale osservazione avrà il formato codificato a seconda del tipo di spazio di osservazione del gioco in questione.

Entrando più nello specifico la funzione ritorna quattro valori:

- Osservazione: un oggetto specifico dell'ambiente che rappresenta la codifica del nuovo scenario dopo l'esecuzione dell'azione.
- Ricompensa: un oggetto di tipo float che corrisponde alla ricompensa ottenuta dalla precedente azione. L'obiettivo di ogni agente è sempre quello di massimizzare la somma di tale valore nel tempo.
- Stato "done": un oggetto boolean che indica se l'episodio è finito o meno. In altre parole, dice se l'agente è capitato in uno stato terminale. Se tale flag è attivo è necessario lanciare la funzione **reset()** per cominciare un nuovo episodio.
- Info: è un dizionario con alcune informazioni che potrebbero essere utili in fase di debug del codice. L'agente non può utilizzare tali informazioni per l'apprendimento.

Nella seguente immagine è rappresentato quello che avviene durante l'interazione agente-ambiente:




L'ultima funzione da analizzare è la funzione **render()** che permette di stampare a video lo stato dell'ambiente in un certo momento.

Ora procediamo ad analizzare i tipi di spazi di azioni che OpenAIGym mette a disposizione.

Action Space

Ogni ambiente ha un suo spazio di azione proprio. L'action space descrive il formato corretto delle azioni eseguibili su un certo ambiente. Per controllare il tipo di spazio in un ambiente occorre andare a cercare la tipologia di gioco nella cartella "envs" della Gym.

L'esempio di spazio più comune è quello **discreto** che permette come valori dei numeri non negativi. In questo caso le azioni possono essere codificate con 0 o 1 ad esempio.

Classi di ambienti

Esistono varie classi di ambiente che raggruppano ognuna vari tipi di giochi.

- *Classic control e toy text*: questa è la prima categoria che raggruppa i giochi classici e più semplici della Gym. In genere gli spazi di osservazioni sono tutti discreti e quindi gli algoritmi di RL possono essere testati con facilità.
- *Algorithmic*: comprendono giochi che eseguono conversioni di stringhe o calcoli fra numeri. In genere l'approccio da usare per risolvere tali giochi è quello di imparare dagli esempi.
- *Atari*: comprende i giochi classici Atari.
- *2D and 3D robots*: si tratta di una sorta di simulazione con un robot. Questa è decisamente la categoria di giochi più difficile da trattare: contiene spazi di osservazione e azioni di solito complessi e continui tali per cui non è possibile applicare con facilità gli algoritmi RL.

Ambienti usati: Taxi v2 e Roulette v0

Sono stati scelti questi due ambienti perché appartengono ad una categoria di ambienti discreti e per loro natura più facili da analizzare.

Gli ambienti usati fanno parte della sezione “toy text”. Nel primo caso l’ambiente taxi è implementato come ambiente discreto in cui sono definite 6 azioni deterministiche (spazio delle azioni di taglia 6). Lo spazio delle osservazioni è composto invece da 500 stati possibili definiti dalla classe “Discrete.py”.

L’ambiente Taxi di base è fatto nel seguente modo:

```
MAP = [
    "+---------+",
    "|R: | : :G|",
    "| : : : : |",
    "| : : : : |",
    "| | : | : |",
    "|Y| : |B: |",
    "+---------+",
]
```

Dove le lettere valgono:

- B: passeggero
- R: destinazione (colore magenta)
- Y: taxi vuoto (colore giallo)
- G: taxi con passeggero (colore verde)

Le ricompense sono collegate all’esecuzione dell’azione nella configurazione corrente dell’ambiente.

Nel caso della Roulette abbiamo sempre spazi tutti discreti con uno spazio di osservazione unico (taglia 1) che rappresenta la roulette del casinò. Lo spazio delle azioni è di taglia 38 dal momento che esse rappresentano i numeri della roulette americana che possono essere selezionati dal giocatore.

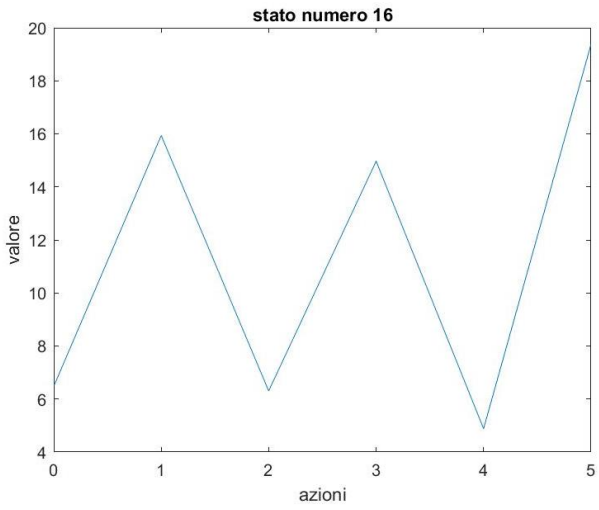
Nel caso della roulette non esiste un rendering vero e proprio a video ma il risultato di ogni iterazione è rappresentata solo da un intero.

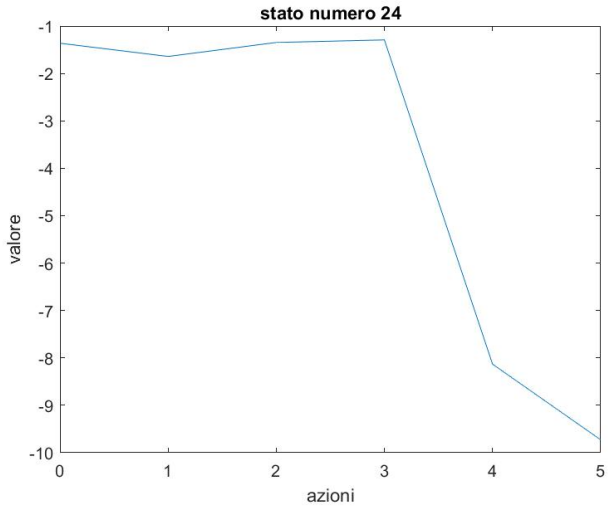
## RISULTATI E CONCLUSIONI

*Analisi di flusso di esecuzione di Taxi v2*

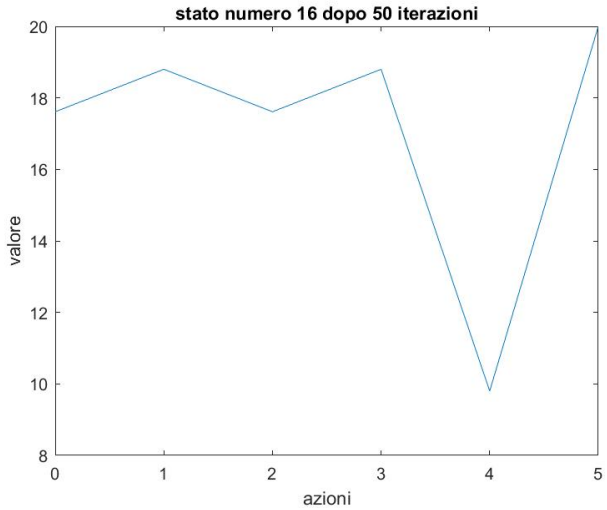
Prima di passare alla parte di discussione dei risultati mostriamo un rendering che mostra l'andamento di apprendimento del Q-learning su Taxi-v2:

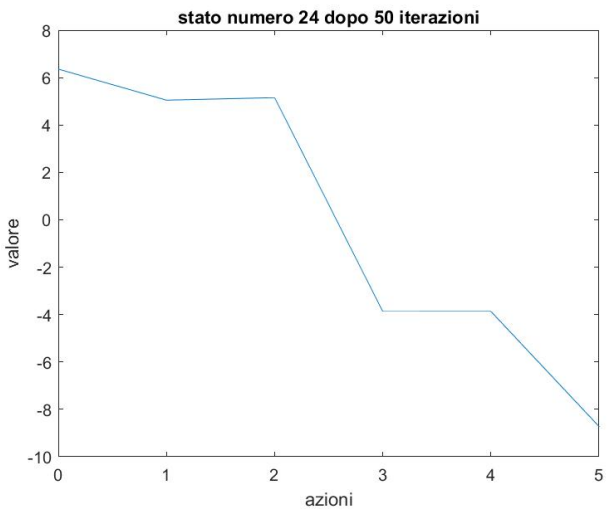
Stato iniziale dei valori delle azioni per gli stati di osservazione numero 16, 24:



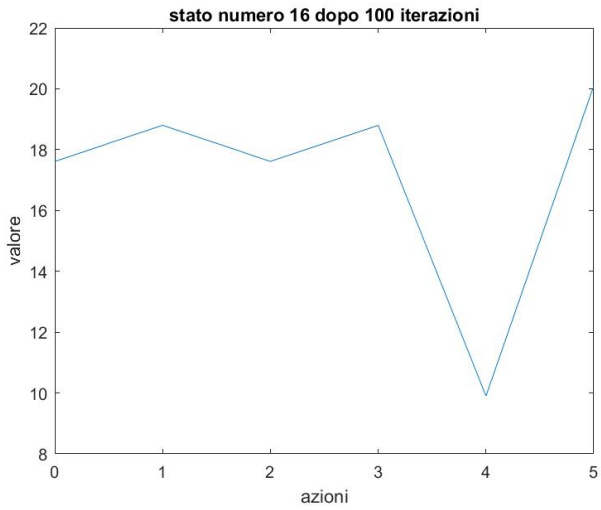



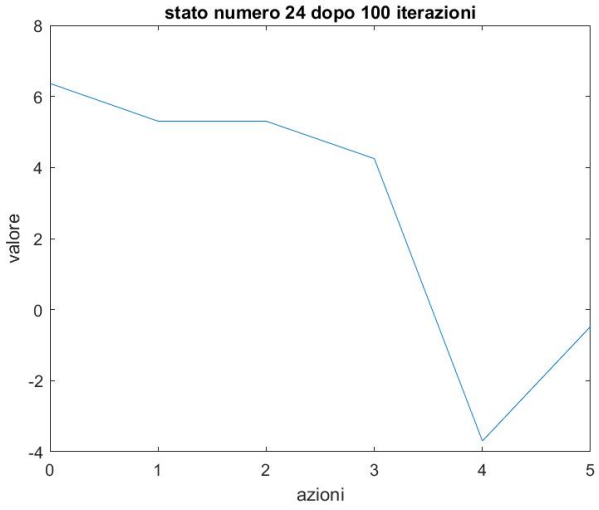

Stato dei valori delle azioni (array) dopo 50 iterazioni:



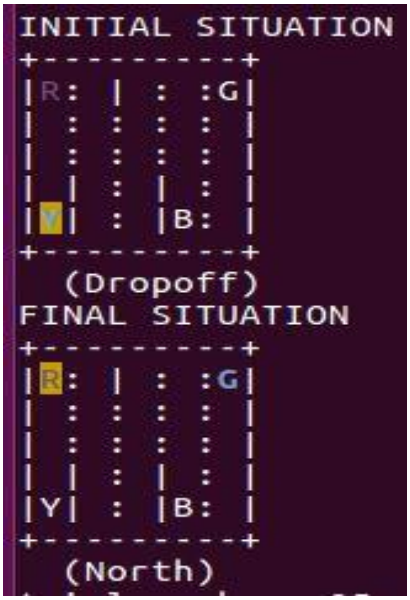
Stato dei valori delle azioni alla fine delle iterazioni (100 iterazioni totali):




Quello che si nota è che le azioni ottimali da eseguirsi in questi due stati particolari saranno le prime tre (azione 0, azione 1, azione 2).

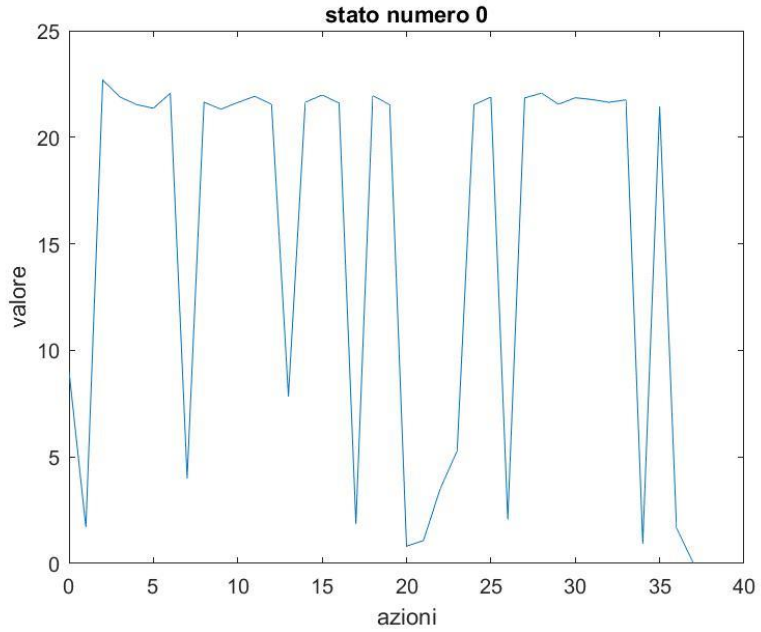
Quello che mostriamo ora è l'immagine che mostra il successo dell'agente nel portare a destinazione il passeggero sulla postazione R a partire da una situazione iniziale casuale.



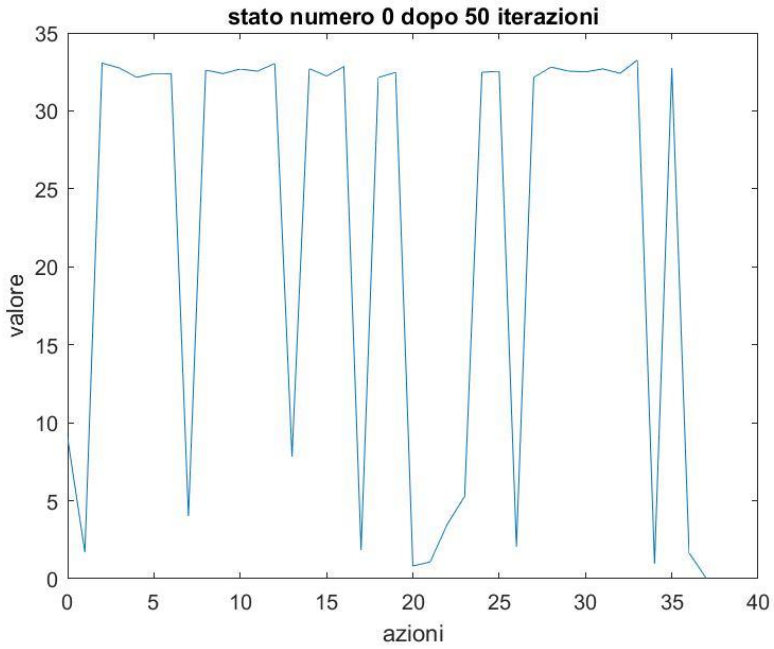
*Analisi del flusso di esecuzione di Roulette v0*

In questo caso abbiamo uno solo stato osservabile (stato 0) come già anticipato precedentemente e la sua evoluzione è mostrata come segue:

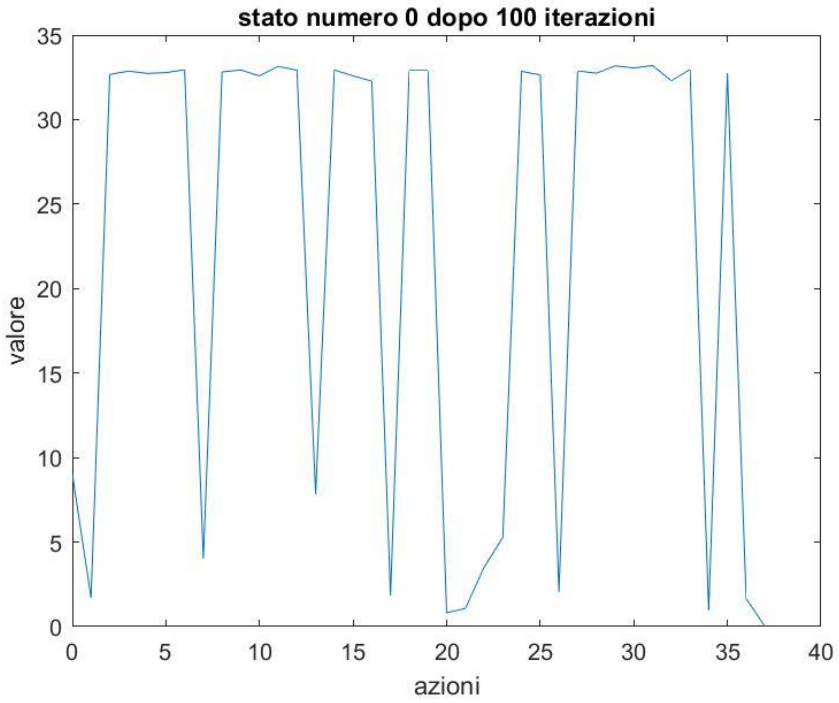
Stato iniziale dei valori per le 38 azioni del gioco:




Stato dei valori delle azioni dopo 50 iterazioni:



Stato finale dei valori delle azioni (dopo 100 iterazioni):



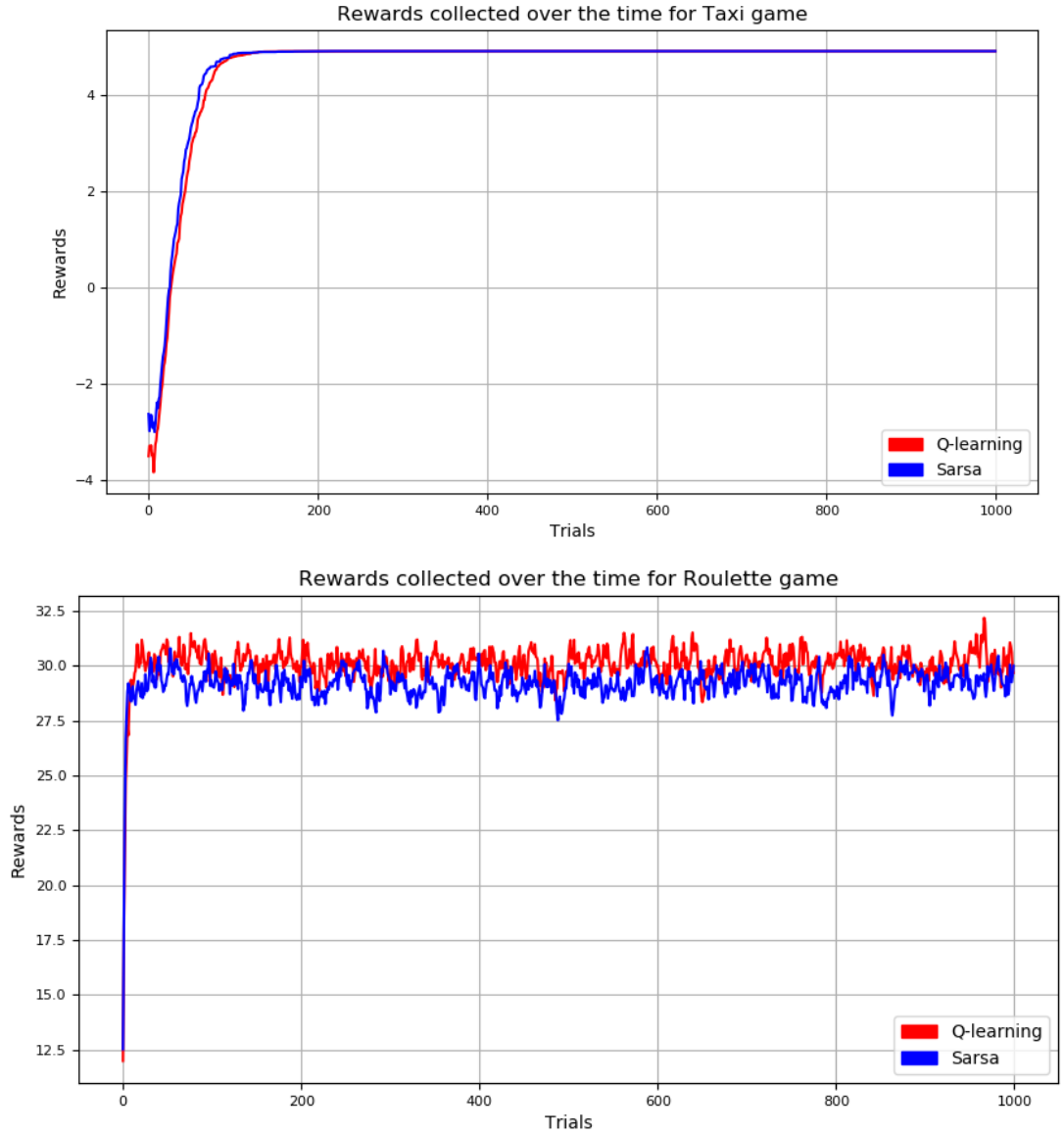

Quello che si nota in questi grafici è ancora una volta la natura stocastica di tale ambiente e quindi i valori delle azioni corrisponderanno al valore medio per il teorema di Bernoulli.

Nel caso della Roulette la funzione render non è implementata perché di fatto non c'è un ambiente osservabile vero e proprio ma solo un valore che rappresenta il punteggio.

Discussione dei risultati

I risultati migliori sono stati ottenuti con l'iterazione dei valori e delle politiche dal momento che seguono una politica precedentemente stimata in maniera iterativa per compiere le azioni. Il Q-Learning ha a disposizione soltanto una tabella con le associazioni stato-azione.

Nei grafici seguenti vengono mostrate le medie su ogni trial delle ricompense ottenute nel caso di Q-Learning e SARSA. Con "trial" si intende un tentativo completo cioè da quando si inizia ad agire sull'ambiente nuovo a quando si entra in uno stato terminale.




Nel caso della roulette abbiamo che il confronto tra Q-learning e Sarsa produce dei risultati con una varianza maggiore dal momento che l'ambiente in questione, seppur discreto, è di natura stocastica e quindi non prevedibile (non a caso la roulette è conosciuta per essere un classico gioco d'azzardo!). Nel caso del Taxi-v2, la

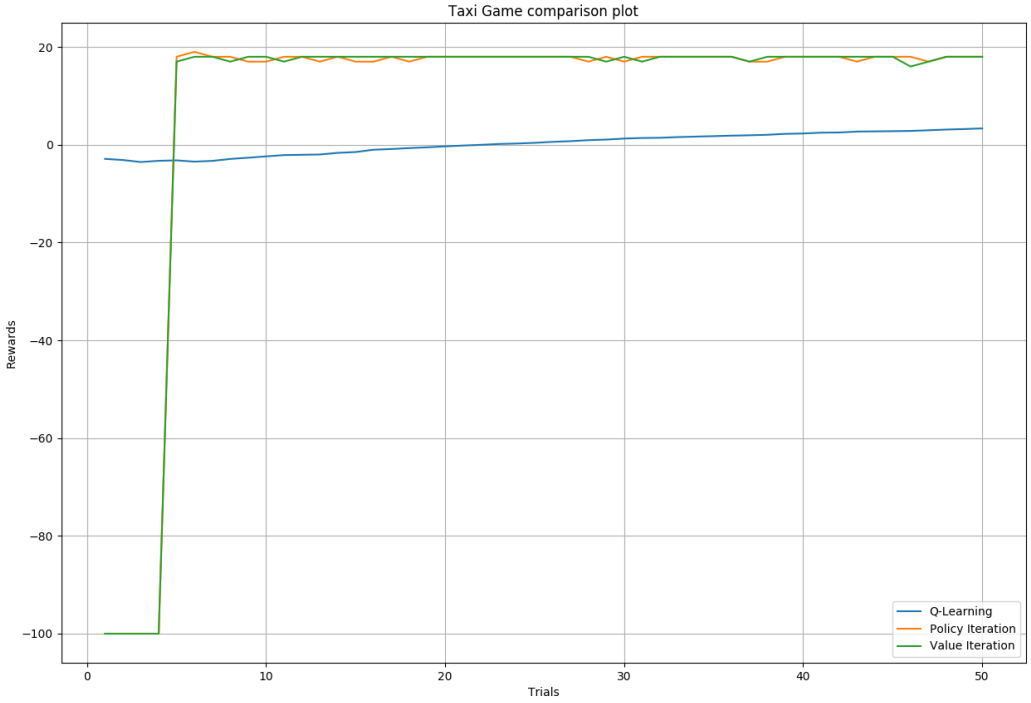
randomizzazione è presente nella creazione dell'ambiente iniziale e, nel 5% dei casi, nella scelta dell'azione sull'ambiente.

Nel secondo caso è stata adottata una strategia "softmax-eps" per la scelta di azioni. Questo è utile nel caso di giochi completamente o parzialmente stocastici come quello della Roulette (secondo grafico). Questa metodologia permette di evitare totalmente le azioni randomiche in un ambiente in cui prendere questo tipo di azioni non permette di ottenere buoni risultati. Si può quindi scegliere un'azione della politica oppure un'azione sempre della politica con una componente maggiore di randomizzazione.

Per concludere, dal triplice confronto tra le tre metodologie, emerge un'informazione interessante: notiamo che il Q-Learning porta a risultati migliori, all'inizio, attestando le ricompense a valori pressoché costanti poco sopra lo zero. Gli altri due metodi invece generano ricompense all'inizio bassissime e poi via via sempre maggiori fino ad un plateau pari al valore massimo.

Questi risultati sono coerenti rispetto a quelli attesi per varie ragioni. In primo luogo, i due metodi con modello dell'ambiente hanno molte più informazioni utilizzabili e quindi è logico aspettarsi che da un certo punto in poi riescano a ottenere maggiori ricompense. Poi è stata rimossa completamente la casualità (come eps greedy o softmax eps del Q-learning) nella scelta delle azioni dal momento che si eseguono esclusivamente le azioni scelte dalla politica considerando il modello dell'ambiente.

Nella seguente immagine viene mostrato il risultato grafico di tale confronto:



# BIBLIOGRAFIA


1. Intelligenza Artificiale, “Un approccio moderno” Vol. 2 di Stuart Russell e Peter Norvig
2. Reinforcement Learning, an introduction di Richard S. Sutton and Andrew G. Barto
3. “OpenAI Gym” di Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, Wojciech Zaremba
4. Informazioni sugli ambienti: https://github.com/openai/gym
5. Informazioni sul funzionamento della piattaforma: https://gym.openai.com/docs/
6. Il codice per intero è reperibile alla pagina: https://github.com/GiacomoFerro