Breakout Q-Learning

Valutazione di semplici modelli di Reinforcement Learning nel videogioco arcade Breakout

Emanuele Falanga

Corso di Fondamenti di Intelligenza Artificiale
https://github.com/Emanuele19/BreakoutQLearning

Università degli Studi di Salerno
Data: 2 luglio 2025

Indice

1 Introduzione

Il *Reinforcement Learning* (RL), o apprendimento per rinforzo, è una delle principali aree di ricerca e applicazione nell'ambito dell'intelligenza artificiale. Si tratta di un paradigma di apprendimento automatico in cui un agente apprende a interagire con un ambiente al fine di massimizzare una ricompensa cumulativa. Per fare questo, l'agente deve scoprire una strategia ottimale, detta policy, bilanciando azioni atte all'esporazione dell'ambiente e allo sfruttamento delle conoscenze acquisite. La complessità dell'ambiente è il fattore che più spesso porta alla scelta di algoritmi costosi in termini di potenza computazionale. Questo lavoro vuole valutare le capacità di semplici algoritmi di RL di trovare strategie ottimali nel videogioco arcade Breakout. Verranno anche discusse le principali problematiche incontrate nell'uso di tali algoritmi e come potrebbero essere mitigate.

2 Obiettivi

L'obiettivo principale di questo lavoro è studiare i trade-offs tra alcuni algoritmi di RL. Per mancanza di hardware non verranno utilizzati algoritmi basati su reti neurali. Gli algoritmi studiati per questo progetto sono stati *Q-Learning*, *Double Q-Learning* e *S.A.R.S.A.*. Nel contesto del gioco l'obiettivo è quello di rompere tutti i mattoncini posti in cima al riquadro del livello. La ricerca della policy ottimale consiste quindi nel trovare un modo per indirizzare la palla verso i mattoncini rimanenti senza farla cadere.

3 Specifica P.E.A.S.

- **Performance**: Le metriche utilizzate per valutare le performance dell'agente sono la media dei rewards ottenuti, la media dei mattoncini rotti e il numero di vittorie ottenute.
- **Environment**: L'ambiente è il livello di gioco. Consiste nello *slider* posto in basso alla schermata ed una fila di *mattoncini* posti in cima. Inoltre è presente una *palla* che l'agente deve intercettare per farla rimbalzare verso i mattoncini, rompendoli.
- **Actuators**: L'agente può intervenire sull'ambiente muovendosi sull'asse orizzontale oppure rimanendo fermo.
- **Sensors** : L'agente viene informato dello stato dell'ambiente tramite un vettore di stato. Ogni elemento del vettore rappresenta una caratteristica dell'ambiente. È definito come segue:

  $$S = \{ball\_x, slider\_x, ball\_direction\_horizontal, brick\_map\}$$

  Ovvero rispettivamente, l'ascissa della palla, l'ascissa dello slider, la direzione orizzontale della palla (verso destra o sinistra) e la mappa binaria dei mattoncini ancora in gioco (dove 1 = in gioco e 0 = distrutto).

4 Caratteristiche dell'ambiente

- **Completamente osservabile**: L'ambiente è completamente osservabile. Viene rappresentato tramite un vettore di stato (Definito nella Sezione 3 al punto *Sensors*);
- **Deterministico**: Dato lo stato attuale S_t si può determinare lo stato successivo S_{t+1} esclusivamente conoscendo S_t e l'azione da attuare A_t;
- **Episodico**: Un episodio è definito come il tempo che intercorre tra il suo inizio e la rottura di tutti i mattoncini, la caduta della palla al di sotto della schermata oppure lo scadere del tempo massimo di gioco, impostato all'equivalente di 2 minuti in frame di gioco;
- **Dinamico**: La collisione tra la palla ed un mattoncino causa la rottura di quest'ultimo, modificando l'ambiente;
- **Discreto**: Le coordinate della palla e dello slider sono rappresentate in dominio continuo nei limiti della rappresentazione decimale, tuttavia l'ambiente è stato discretizzato (Sezione 7);
- **Single-agent**: Lo slider è l'unico agente.

5 Algoritmi utilizzati

In questa sezione verranno discusse le caratteristiche degli algoritmi scelti, gli iperparametri che li caratterizzano e come questi sono stati impostati.

5.1 Iperparametri

Gli iperparametri comuni a tutti gli algoritmi trattati sono:

- **Learning rate**($0 < \alpha \leq 1$): Questo parametro rappresenta la grandezza del passo in direzione di un minimo della *loss function*. Un learning rate basso rallenta la convergenza e l'agente rischia di imparare una politica sub-ottimale. Di contro, un learning rate alto può portare ad *overshooting*, ovvero a mancare la soluzione ottimale facendo un passo troppo grande;

- **Discount factor**($0 \leq \gamma \leq 1$): Questo parametro rappresenta il peso dei premi futuri. Un *discount factor* basso porta l'agente a preferire azioni che lo portano a massimizzare i premi nel breve termine, uno alto invece lo porta a massimizzare il reward cumulativo nel lungo termine;

- **Exploration rate**($0 \leq \varepsilon \leq 1$): Questo parametro determina con quanta probabilità l'agente esplorerà l'ambiente con azioni casuali oppure sfrutterà quanto appreso fino a quel momento (*exploration-exploitation trade-off*). Questa strategia viene detta *Epsilon greedy* ed è definita come segue:

$$a_t = \begin{cases} \text{azione ottimale} & \text{con probabilità } 1 - \varepsilon, \\ \text{azione casuale} & \text{con probabilità } \varepsilon. \end{cases}$$

  Dove a_t è l'azione scelta all'istante t.

- **decay rate**: Indica il tasso di decadimento di una funzione esponenziale. Viene utilizzato nelle funzioni di decadimento dei parametri;

- **episodes**: Numero di episodi di addestramento;

6 Rewards

- **min/max penalty**: quando la palla cade fuori dallo schermo viene assegnata una penalty all'agente proporzionale alla distanza tra palla e agente. Questa coppia di penalty definisce gli estremi dell'intervallo da cui viene campionata la penalty finale;
- **tick penalty**: assegnata ad ogni frame di gioco;
- **ceiling penalty**: assegnata al tocco con il soffitto;
- **time exceeded penalty**: assegnata se l'agente supera il tempo massimo di gioco;
- **bounce reward**: assegnato quando la palla rimbalza sullo slider;
- **brick reward**: assegnato quando la palla distrugge un mattoncino;
- **win reward**: assegnato quando tutti i mattoncini sono distrutti;

6.1 Q-Learning

Il Q-Learning [1] cerca di trovare una funzione $Q(s, a)$ che massimizzi il valore atteso della ricompensa cumulativa. Per farlo viene definita una tabella che associa ad ogni coppia stato-azione il valore atteso del reward che si otterrebbe eseguendo quella azione e seguendo poi una politica ottimale. L'aggiornamento della tabella viene fatto tramite la seguente formula:

$$Q(s, a) \leftarrow Q(s, a) + \alpha \left[r + \gamma \max_{a'} Q(s', a') - Q(s, a) \right] \tag{1}$$

Come spiegato nella sezione 5.1 viene utilizzata una politica *ε-greedy*. L'exploration rate è stato impostato a 1 per aumentare l'esplorazione nelle fasi iniziali e poi fatto decadere per aumentare lo sfruttamento nelle fasi dove l'agente ha acquisito esperienza. Sono state confrontate due politiche di decadimento:

- **Lineare** con $\varepsilon_0 = 1$ e $\varepsilon_{\min} = 0.01$:

$$\varepsilon \leftarrow \max \left(\varepsilon_{\min}, \varepsilon_0 - \frac{\text{current\_episode} \cdot (\varepsilon_0 - \varepsilon_{\min})}{\text{total\_episodes}} \right)$$

- **Esponenziale** con rateo di decadimento(λ) impostato a 0.00003:

$$\varepsilon_n = \varepsilon_0 \cdot e^{-\lambda n}$$

6.1.1 Risultati

La politica di decadimento esponenziale (Figura 1) si è rivelata nettamente migliore. La causa potrebbe essere data da un decadimento troppo lento se lineare, che lascia l'agente prendere decisioni casuali per una parte considerevole dell'allenamento, introducendo rumore che renderà difficile l'aggiornamento corretto della tabella. Si notino in particolare i valori nella sezione rossa nei grafici di comparazione delle prestazioni (Figura 2): questi indicano il numero di volte che l'agente è rimasto bloccato in una sequenza ripetitiva di azioni che lo portava a far rimbalzare la palla sempre nello stesso punto. Evidentemente l'agente si sofferma su una politica sub-ottimale. Tuttavia, fin ora è stato utilizzato un valore di learning-rate costante. È stata quindi effettuata una serie di tre test per valutare la capacità di convergenza del q-learning quando il learning-rate decade esponenzialmente (Figura 3). Durante questi test sono state utilizzate le migliori configurazioni per il decadimento del tasso di esplorazione. In questo caso la politica di decadimento ha portato l'aumento del punteggio medio dell'agente, tuttavia non ha risolto il problema dei loop di azioni. La motivazione potrebbe essere che il learning rate decada troppo velocemente quando l'agente comincia ad esplorare lo spazio degli stati in fasi avanzate del gioco.

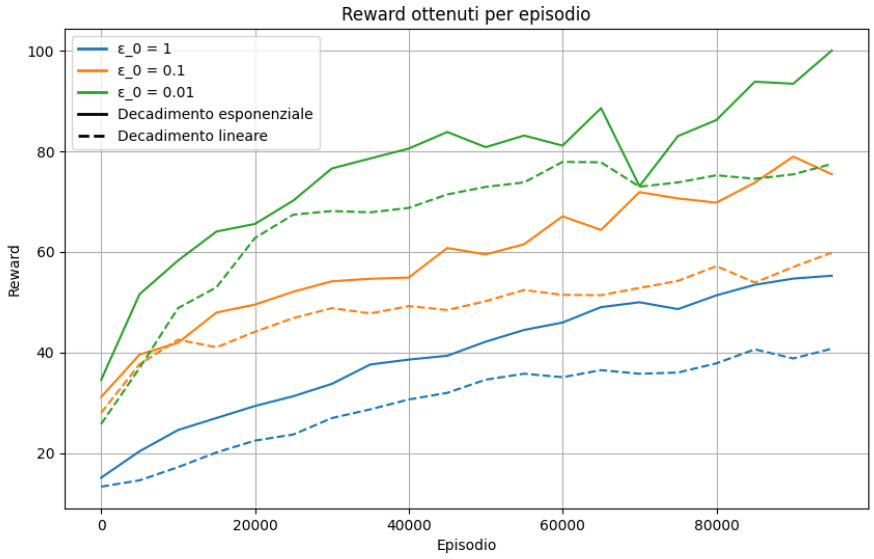
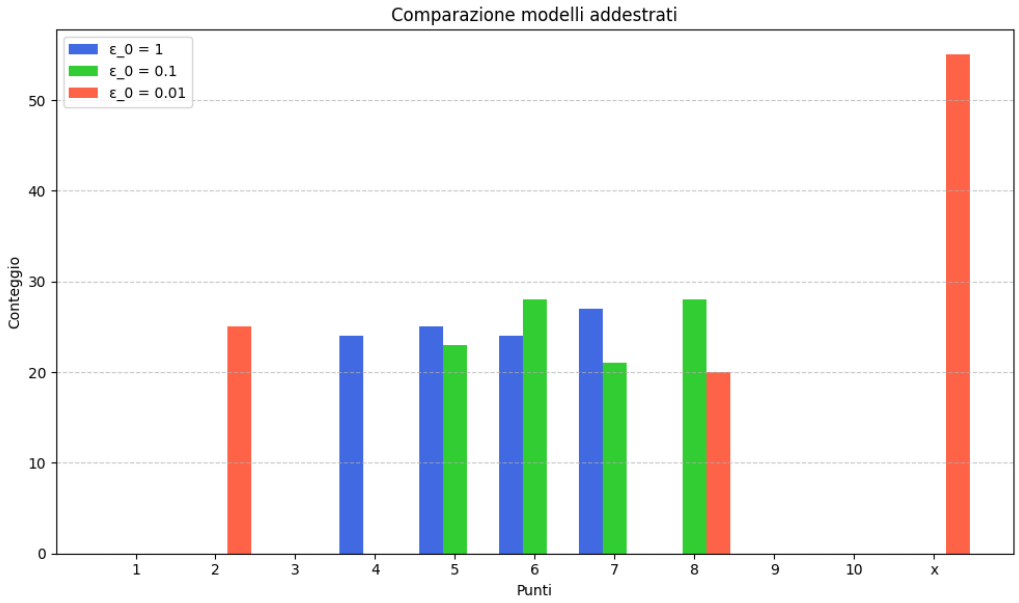
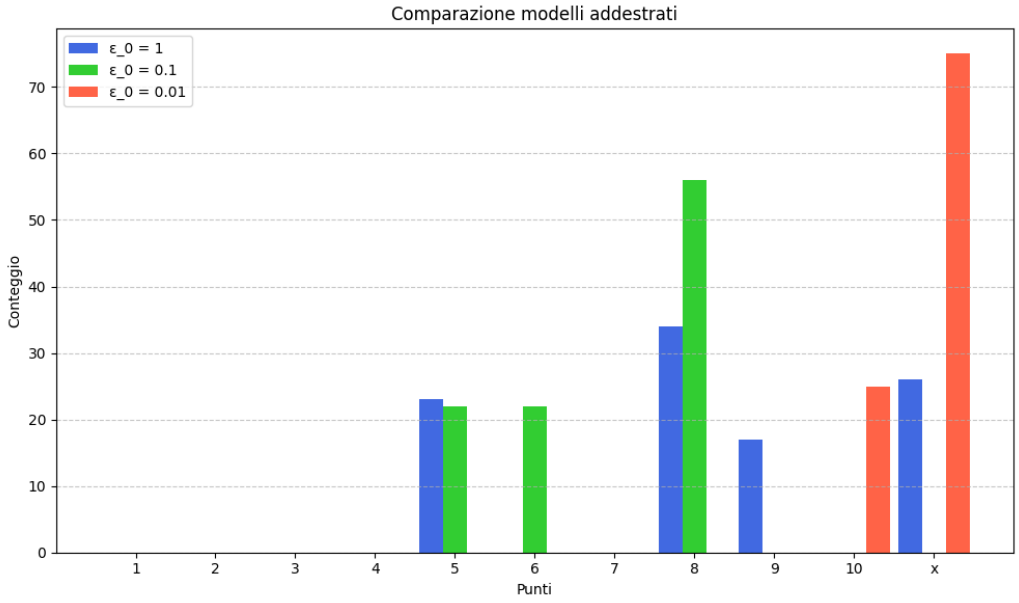


Figura 1: Confronto tra decadimento lineare ($\varepsilon_{min} = \varepsilon_0/100$) ed esponenziale (decay rate= 0.00003) con tre valori di epsilon iniziali. Per leggibilità ogni punto è la media dei 5000 punti realmente registrati.



(a) Grafico dei punti ottenuti da ogni modello (ε decay lineare). L'ultima colonna indica il numero di volte che l'agente si è bloccato in un loop.



(b) Grafico dei punti ottenuti da ogni modello (ε decay esponenziale). L'ultima colonna indica il numero di volte che l'agente si è bloccato in un loop.

Figura 2: Confronto dei modelli con ε decay lineare ed esponenziale.

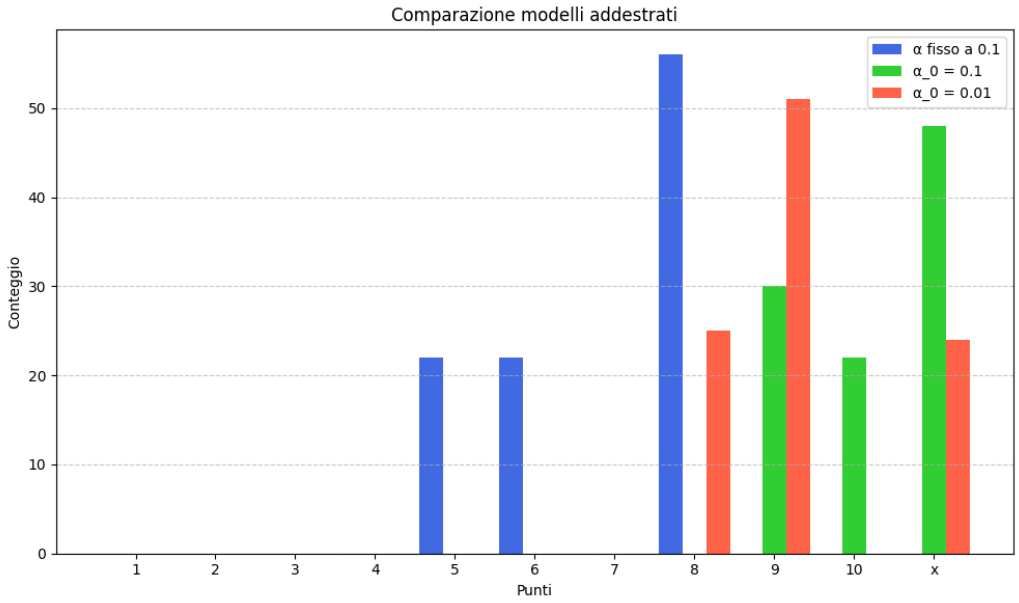


Figura 3: Confronto tra learning rate fisso e due learning rate base fatti decadere esponenzialmente con decay rate= 0.00003

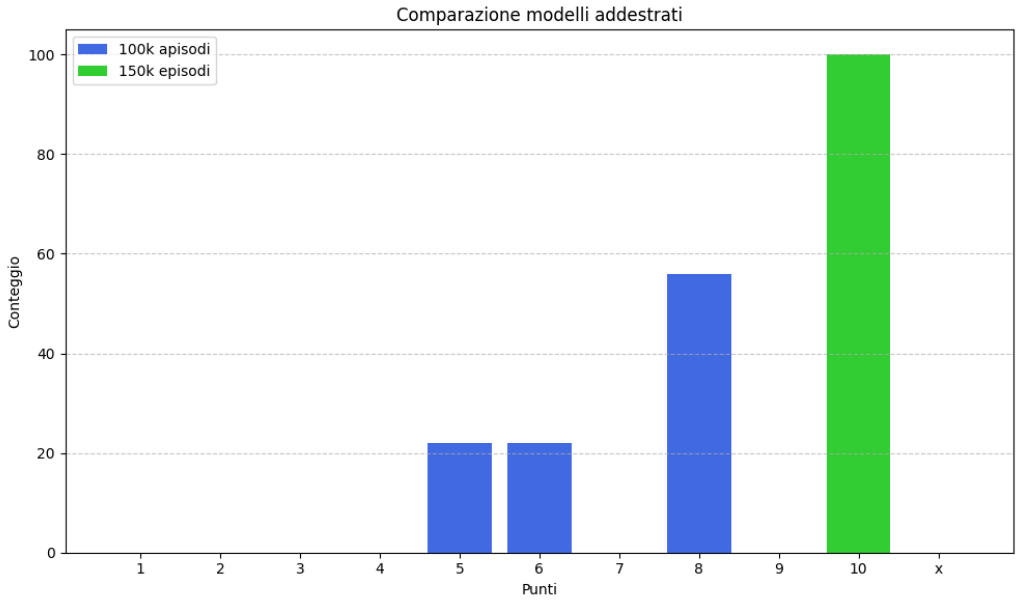


Figura 4: Confronto tra modello addestrato su 100.000 episodi e 150.000 episodi

L'ultimo test è stato fatto per confrontare il modello allenato su 100.000 e 150.000 episodi (Figura 4). Complessivamente l'agente riesce a risolvere il livello se addestrato su 150.000 episodi. I modelli discussi in seguito sono stati riportati come tentativo di ridurre il tempo di training garantendo che l'agente riesca a risolvere il livello.

6.1.2 Implementazione

Algorithm 1 Q-Learning Tabellare

Require: Numero di episodi N, tasso di apprendimento α, fattore di sconto γ, parametro di esplorazione ε

Inizializza $Q(s, a) \leftarrow 0$ per tutti gli stati s e le azioni a
for episodio $= 1$ to N **do**
    Inizializza lo stato s
    while lo stato s non è terminale **do**
        Seleziona un'azione a con la politica ε-greedy su $Q(s, a)$
        Esegui l'azione a, osserva la ricompensa r e il nuovo stato s'
        Aggiorna $Q(s, a)$ secondo:

$$Q(s,a) \leftarrow Q(s,a) + \alpha \left[r + \gamma \max_{a'} Q(s', a') - Q(s,a) \right]$$

        $s \leftarrow s'$
    end while
end for
return Q

6.2 Double Q-Learning

Il double Q-Learning [2] nasce dalla necessità di risolvere un problema del q-learning conosciuto come *overestimation bias*. Analizzando la formula di aggiornamento del q-learning (1) notiamo che l'aggiornamento viene sempre fatto usando il termine $\max_{a'} Q(s', a')$. Questo termine è responsabile della stima del valore atteso del reward cumulativo, ovvero una parte fondamentale nel q-learning. Tuttavia nel processo di addestramento del modello, specialmente nelle fasi iniziali, questo termine è la causa della propagazione di un errore di incertezza. L'aggiornamento viene quindi fatto tramite su tabelle per ridurre la propagazione di un errore di stima in un dato istante:

$$
\begin{aligned}
Q_1(s,a) &\leftarrow Q_1(s,a) + \alpha\big[r + \gamma Q_2(s', \arg\max_a Q_1(s',a)) - Q_1(s,a)\big], \\
Q_2(s,a) &\leftarrow Q_2(s,a) + \alpha\big[r + \gamma Q_1(s', \arg\max_a Q_2(s',a)) - Q_2(s,a)\big]. \quad (2)
\end{aligned}
$$

Questo problema è particolarmente noto quando si lavora in ambienti stocastici. L'ambiente utilizzato in questo esperimento è invece deterministico, tuttavia ho ritenuto che questa variante del q-learning potesse comunque conferire ulteriore stabilità al modello, aiutandolo a convergere. Per confermare questa teoria, l'agente è stato allenato con i migliori parametri trovati con gli esperimenti della sezione 6.1.

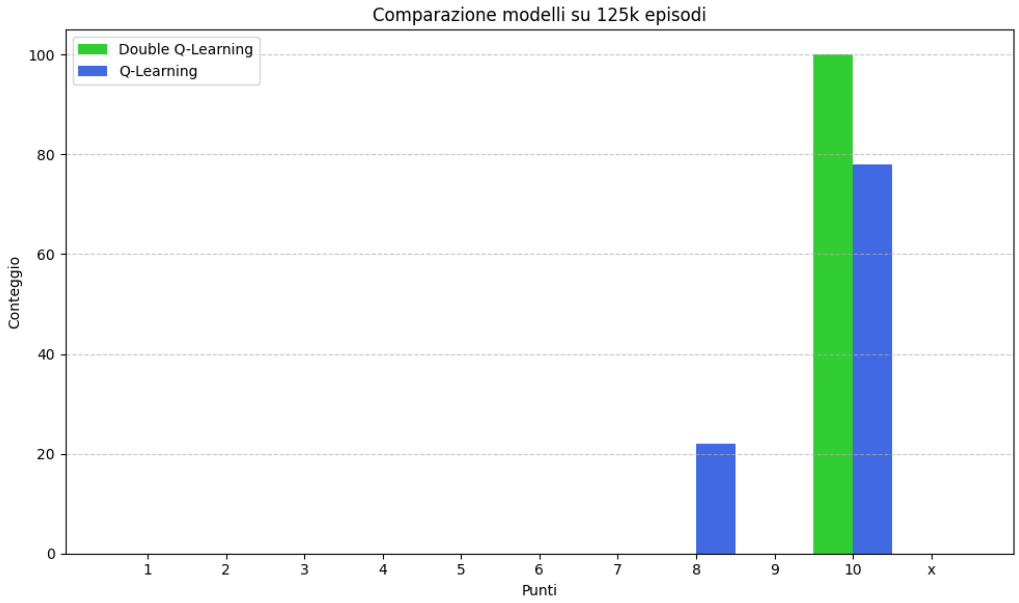


Figura 5: Comparazione agente allenato con Q-Learning e Double Q-Learning su 125.000 episodi

6.2.1 Risultati

L'esperimento ha avuto successo: già a 125.000 episodi di addestramento l'agente trova una politica che risolve il livello (Figura 5). L'agente basato su Q-Learning semplice, invece, vince il livello solo il 78% delle volte.

6.2.2 Implementazione

Algorithm 2 Double Q-Learning

Require: Numero di episodi N, tasso di apprendimento α, fattore di sconto γ, parametro di esplorazione ε

Inizializza $Q_1(s,a) \leftarrow 0$ e $Q_2(s,a) \leftarrow 0$ per tutti gli stati s e le azioni a
for episodio $= 1$ to N **do**
    Inizializza lo stato s
    while lo stato s non è terminale **do**
        Seleziona un'azione a con la politica ε-greedy utilizzando $Q_1 + Q_2$
        Esegui l'azione a, osserva la ricompensa r e il nuovo stato s'
        if con probabilità 0.5 **then**
            Aggiorna Q_1:

$$Q_1(s,a) \leftarrow Q_1(s,a) + \alpha\big[r + \gamma Q_2(s', \arg\max_a Q_1(s',a)) - Q_1(s,a)\big]$$

        else
            Aggiorna Q_2:

$$Q_2(s,a) \leftarrow Q_2(s,a) + \alpha\big[r + \gamma Q_1(s', \arg\max_a Q_2(s',a)) - Q_2(s,a)\big]$$

        end if
        $s \leftarrow s'$
    end while
end for
return Q_1 e Q_2

6.3 S.A.R.S.A.

S.A.R.S.A (State Action Reward State Action), è un algoritmo di RL **on-policy**: l'apprendimento avviene unicamente tramite le azioni intraprese direttamente dall'agente a differenza del q-learning che è **off-policy** perché aggiorna la tabella tramite una stima della politica ottimale. Questa differenza è visibile nella formula di aggiornamento:

$$Q(s,a) \leftarrow Q(s,a) + \alpha \left[r + \gamma Q(s',a') - Q(s,a) \right] \tag{3}$$

6.3.1 Risultati

L'approccio applicato è analogo a quello avuto con il Double Q-Learning: sono stati utilizzati i migliori parametri trovati nella Sezione 6.1. Questa volta l'agente non è riuscito a trovare una politica che gli permettesse di vincere il livello. Allenato su 150.000 epsisodi riesce a vincere solo il 58% delle volte (Figura 6.

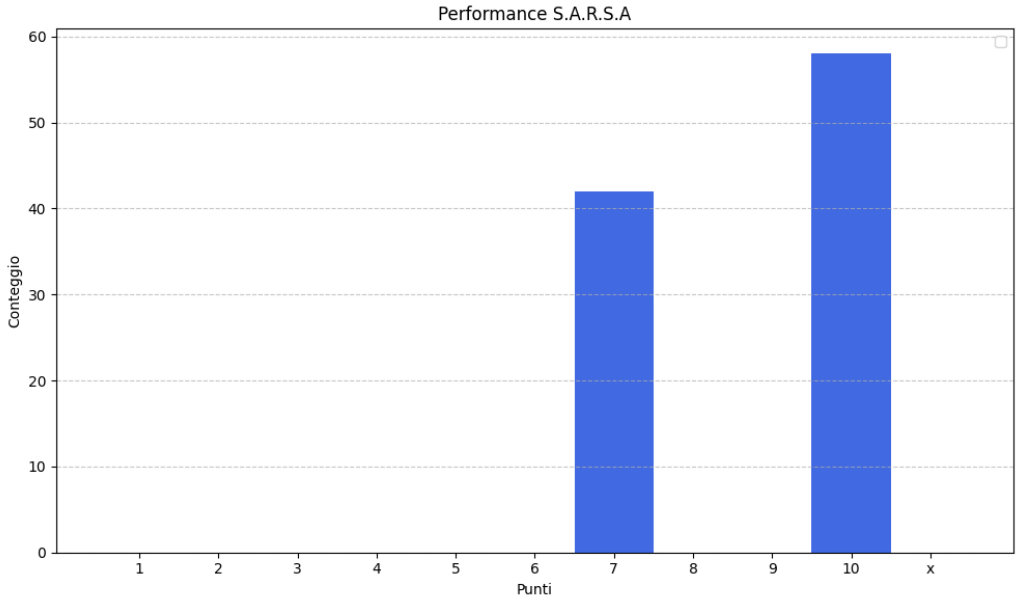


Figura 6: S.A.R.S.A addestrato su 150.000 episodi

6.3.2 Implementazione

Algorithm 3 SARSA (State-Action-Reward-State-Action)

Require: Numero di episodi N, tasso di apprendimento α, fattore di sconto γ, parametro di esplorazione ε

Inizializza $Q(s, a)$ arbitrariamente per tutti gli stati s e le azioni a
for episodio $= 1$ to N **do**
    Inizializza lo stato s
    Seleziona un'azione a usando la politica ε-greedy su $Q(s, a)$
    while lo stato s non è terminale **do**
        Esegui l'azione a, osserva la ricompensa r e il nuovo stato s'
        Seleziona un'azione a' dal nuovo stato s' usando la politica ε-greedy
        Aggiorna $Q(s, a)$:

$$Q(s, a) \leftarrow Q(s, a) + \alpha\big[r + \gamma Q(s', a') - Q(s, a)\big]$$

        $s \leftarrow s'$, $a \leftarrow a'$
    end while
end for
return Q

7 Discretizzazione dello spazio degli stati

La natura tabulare degli algoritmi utilizzati non li rende adatti ad uso in ambienti continui. Sono stati individuati due metodi che permettono di aggirare il problema:

- **Funzione di approssimazione**: Si parametrizza la funzione Q tramite un *feature vector* e si predispone un meccanismo di apprendimento dedicato a trovare il vettore di parametri che meglio la approssima:

$$Q(s, a; \theta) = f(s, a; \theta) \tag{2}$$

  La funzione utilizzata per l'approssimazione può essere lineare o meno. Il primo caso difficilmente catturerebbe la complessità del task, il secondo richiede l'utilizzo di reti neurali, collocandosi al di fuori dello scope di questo lavoro.

- **Discretizzazione**: Le caratteristiche continue vengono campionati. Per le coordinate si applica una griglia al livello di gioco, quindi la posizione di ogni oggetto non è più precisa ma diventa la posizione della cella nella quale si trova. La direzione della palla diventa invece binaria (0 = sinistra, 1 = destra), la direzione verticale non viene tenuta in considerazione. Infine la presenza di mattoncini viene codificata come un vettore binario dove 1 = in gioco e 0 = distrutto. La grandezza delle celle permette di regolare la granularità della discretizzazione.

È stata scelta la seconda opzione in quanto semplice da realizzare ed efficace con granularità adeguata. Tuttavia questa tecnica risulta essere pesante in termini di occupazione di memoria. Fissata una grandezza di campionamento c, che indica il numero di suddivisioni dello spazio fatte sull'asse verticale e su quella orizzontale e il numero di possibili posizioni per lo slider, un numero di mattoncini n, tenendo in considerazione la direzione trattata come un parametro binario e assumendo che la grandezza di un intero sia di $4B$ e quella di un booleano di $1B$ si può calcolare l'occupazione di una q-table per Q-Learning e S.A.R.S.A come $8c^2 \times 2^n$ e per il Double Q-Learning il doppio. Per i test è stata impostata $c = 20$ e $n = 10$, quindi l'occupazione ammonta a ≈ 3.1MB.

Per questo particolare problema, la rappresentazione degli stati risulta essere particolarmente restrittiva: basti pensare che la sola rappresentazione dei mattoncini è necessariamente in forma esponenziale (2^n), rendendo impossibile simulare un livello completo del gioco arcade originale,

8 Conclusioni

Complessivamente sia il Q-Learning che la variante Double Q-Learning riescono a trovare una politica che risolve il livello. In particolare, il secondo modello sembra essere più stabile in addestramento, portando ad una convergenza più rapida. Tuttavia, il Double Q-Learning utilizza comunque il doppio dello spazio di memoria rispetto al Q-Learning classico, condizione potenzialmente svantaggiosa in presenza di spazi di stati più vasti. La rappresentazione dello spazio degli stati risulta essere il fattore più limitante: basti pensare che la sola rappresentazione dei mattoncini è necessariamente in forma esponenziale (2^n stati possibili tra mattoncini rotti e in gioco), rendendo impraticabile l'addestramento del modello su un ambiente anche solo con 20 mattoncini. Per sviluppi futuri sarebbe interessante valutare le prestazioni di questi algoritmi in un ambiente più complesso, come con più mattoncini, utilizzando una funzione di approssimazione degli stati, e algoritmi di deep learning come il Deep Q-Learning.

Riferimenti bibliografici

[1] Christopher J. C. H. Watkins e Peter Dayan. "Q-Learning". In: *Machine Learning* 8.3-4 (1992), pp. 279–292.

[2] Hado Hasselt. "Double Q-learning". In: *Advances in Neural Information Processing Systems*. A cura di J. Lafferty et al. Vol. 23. Curran Associates, Inc., 2010. URL: https://proceedings.neurips.cc/paper_files/paper/2010/file/091d584fced301b442654dd8c23b3fc9-Paper.pdf.