SAPIENZA
UNIVERSITÀ DI ROMA

Analisi e confronto di algoritmi di Reinforcement Learning in un ambiente di controllo discreto

INGEGNERIA DELL'INFORMAZIONE, INFORMATICA E STATISTICA
INGEGNERIA INFORMATICA E AUTOMATICA [L-270 - Ordin. 2019]

Alessandro Carotenuto
Matricola 1847282

Relatore
Francesco Delli Priscoli

Correlatore
Andrea Wrona

Anno Accademico 2022/2023

Analisi e confronto di algoritmi di Reinforcement Learning in un ambiente di controllo discreto

Tesi di Laurea Triennale. Sapienza Università di Roma

Questa tesi è stata composta con LaTeX e la classe Sapthesis.

Email dell'autore: carotenuto.1847282@studenti.uniroma1.it

Sommario

Questo studio analizza e confronta tre algoritmi di Reinforcement Learning: SARSA, Q-Learning ed Expected SARSA nel contesto di un ambiente di controllo discreto, l'obiettivo dello studio è valutare, nello stesso contesto, in che modo variazioni nei parametri con cui possono essere settati gli algoritmi possano influenzare le prestazioni degli stessi, e il perché di queste variazioni.

Nel capitolo 1, vengono esposti i concetti fondamentali del Reinforcement Learning e le principali componenti dei sistemi che sfruttano questa modalità di apprendimento, successivamente una breve introduzione storica mediante alcuni dei principali paper pubblicati dai primi del novecento in poi a cui segue una descrizione dei Markov Decision Process e un'introduzione sui concetti di *funzioni di valore di stato e di azione*, per concludere con una breve panoramica dei principali approcci all' apprendimento automatico facenti parti del Reinforcement Learning e non.

Nel capitolo 2, viene approfondito uno degli approcci citati nel capitolo precedente, il Temporal Difference Learning, vi è una spiegazione sulla classe di algoritmi $TD(\lambda)$, le sue compoenti e sul suo sottoinsieme $TD(0)$, di tale sottoinsieme sono poi stati scelti tre algoritmi da esporre nel dettaglio: SARSA, Q-Learning ed Expected Sarsa.

Nel capitolo 3, viene effettuata l'analisi dei tre algoritmi, vengono applicati allo stesso ambiente discreto mediante la stessa politica di esplorazione, inizialmente una **ϵ-Greedy** e vengono confrontate le prestazioni in termini di *reward per episodio, lunghezza episodio e errore di Temporal Difference medio per episodio.* Tale confronto avviene con lo stesso set di parametri, successivamente vengono uno ad uno variati il **learning rate**, il **discount factor** e l'**esplorazione iniziale**, per infine cambiare anche la politica di esplorazione in favore di una **Upper Confidence Bound**. Vengono giustapposte visualizzazioni delle prestazioni degli algoritmi e commentati i risultati.

Indice

Capitolo 1

Markov Decision Process e il Reinforcement Learning

1.1 Concetti fondamentali del Reinforcement Learning

Il Reinforcement Learning è una branca dell' apprendimento automatico ispirato dai paradigmi di apprendimento del regno umano e animale ed è interamente basato sull' intuizione che un **agente** impari dall'interazione con l'ambiente circostante e dai feedback che tali interazioni generano. Lo scopo dei metodi di Reinforcement Learning è massimizzare un segnale numerico di **ricompensa** (detto **reward**) in questo quindi, differiscono dall'apprendimento *superivisionato* che mira a generalizzare pattern di soluzione e da quello *non-supervisionato* il cui scopo è spesso trovare strutture nascoste e sconosciute in determinati in set di dati [1].

Agente e componenti

Definiamo **agente** l'entità che deve apprendere, le caratteristiche fondamentali che deve possedere sono:

- Una percezione dell' ambiente circostante
- La possibilità di interagire con esso

Oltre l'agente, altre componenti che formano un sistema di apprendimento di Reinforcement Learning sono:

- **Policy** π: Una policy (politica) è una funzione che mappa stati dell'ambiente ad azioni da eseguire in quel determinato stato (risulta simile all'associazione stimolo-risposta negli umani)
- **Reward** R_t: Ricompensa, definisce l'obiettivo, ciò che è giusto/sbagliato nel breve periodo, ad ogni unità di tempo (o step) dell'ambiente
- **Value Function**: Funzioni *valore di stato* o *valore di azione*. Una funzione dei reward, definisce ciò che è giusto/sbagliato nel lungo periodo, può essere associata ai singoli stati o alle coppie stato-azione

Opzionalmente, può essere presente un **modello** che mima l'ambiente e che viene utilizzato per la pianificazione. L'apprendimento per rinforzo si basa sul concetto di **stato**, come input per la policy e la funzione di valore, e considerando il modello nel suo complesso, lo stato può essere visto sia come un input allo stesso, che un output dello stesso e non è altro che una *descrizione dell' ambiente* in un determinato istante di tempo.

Applicazioni

Non sorprende che il Reinforcement Learning sia utilizzato come strumento computazionale pratico per la costruzione di sistemi autonomi che migliorano con l'esperienza. Queste applicazioni spaziano dalla robotica alla produzione industriale, ai problemi di ricerca combinatoria come il gioco al computer. Il reinforcement learning ha avuto risultati impressionanti nei giochi come la dama di Samuel e l'algoritmo TD-Gammon per il backgammon, dimostrando l'efficacia di questa tecnica. È utilizzato anche nella robotica e nel controllo, ad esempio per l'insegnamento di abilità ai robot e per l'ottimizzazione dei sistemi di automatici. È un campo di ricerca promettente per sistemi intelligenti, offrendo soluzioni per problemi complessi [2].

1.2 Cenni Storici

Nella storia dello sviluppo dei metodi di apprendimento automatici, troviamo generalmente due percorsi di ricerca che sono stati perseguiti indipendentemente l'uno dall'altro per poi intersecarsi nel formare quello che conosciamo oggi come Reinforcement Learning. Il primo filone di ricerche è originario della psicologia e dell' apprendimento animale ed è basato sull' apprendimento *trial-and-error* mentre l'altro percorso è incentrato principalmente sullo sviluppo di un controllo ottimo usando anche valori di stato o valori di azioni. Entrambi questi filoni di ricerca hanno iniziato ad unirsi alla fine degli anni Ottanta in quello che è chiamato oggi Reinforcement Learning.

- *Il controllo ottimo,* come termine, nasce intorno agli anni Cinquanta in riferimento al problema di progettazione di un controllore per minimizzare/massimizzare una misura del comportamento di un sistema dinamico. Uno degli approcci a questo problema è la rielaborazione di Richard Bellman di una teoria originariamente nata da Hamilton e Jacobi, che utilizza lo stato di un sistema dinamico e il concetto di *value function* o *optimal return function*, per definire un insieme di equazioni ora chiamate **Equazioni di Bellman dell' ottimalità** [3].

- *L'apprendimento Trial-Error,* è un'idea che risale al Diciannovesimo secolo, all' idea di apprendimento mediante sperimentazione di Alexander Bain e più recentemente a fine secolo per quanto riguarda le osservazioni sull' apprendimento animale di Conway Lloyd Morgan. La prima formalizzazione è dovuta a Edward Thorndike nel 1911 ed è stata chiamata *Law of Effect* [4], ha generato diverse rielaborazioni e controversie nel corso dei secoli. La terminologia "Apprendimento per rinforzo" appare per le prime volte probabilmente nel 1927

nei lavori di Pavlov sui riflessi condizionati, in cui il rinforzo è formalizzato come *rafforzamento di un pattern comportamentale di un animale in seguito della ricezione di uno stimolo, in un arco temporale appropriato ad un altro stimolo o risposta.* L'idea di implementare questi concetti in un computer risale al 1948 ed è dovuta proprio ad Alan Turing, con il design di un sistema *piacere-dolore* per un computer [5].

Particolarmente influente è stato il saggio di Minsky intitolato "Steps Towards Artificial Intelligence" del 1961 [6], che ha discusso diversi argomenti rilevanti per l'apprendimento trial-and-error, inclusa la previsione, l'aspettativa e quello che lui ha chiamato il *problema fondamentale dell'assegnazione del merito per i sistemi complessi di apprendimento per rinforzo*: Come distribuire il merito per il successo tra le molte decisioni che potrebbero essere state coinvolte nella sua produzione? Diversi algoritmi rispondono a questa domanda in modo differente, parte dell' analisi presente di seguito è volta a comprendere meglio le differenze che alcuni algoritmi hanno nel rispondere a questa domanda.

1.3 Markov Decision Processes

I Processi Decisionali di Markov (Markov Decision Processes, MDP) sono un'astrazione del problema di apprendimento guidato da obiettivi attraverso l'interazione. Nel caso generale del problema di apprendimento per rinforzo, le azioni dell'agente determinano non solo la sua ricompensa immediata, ma anche (almeno probabilisticamente) lo stato successivo dell'ambiente. Agente e ambiente in un Markov Decision Process interagiscono in una sequenza di passi temporali discreti. Ad ogni passo (*step*), l'agente riceve una rappresentazione dello stato dell'ambiente, seleziona un'azione, ottiene una ricompensa ed una nuova rappresentazione dello stato. La sequenza classica che otteniamo è *stato-azione-reward-nuovo stato*. Obiettivo del sistema è sempre massimizzare una funzione del **reward**, un ritorno atteso. Spesso questa funzione è relativa a dei sottoinsiemi di interazioni agente-ambiente chiamati ***episodi***.

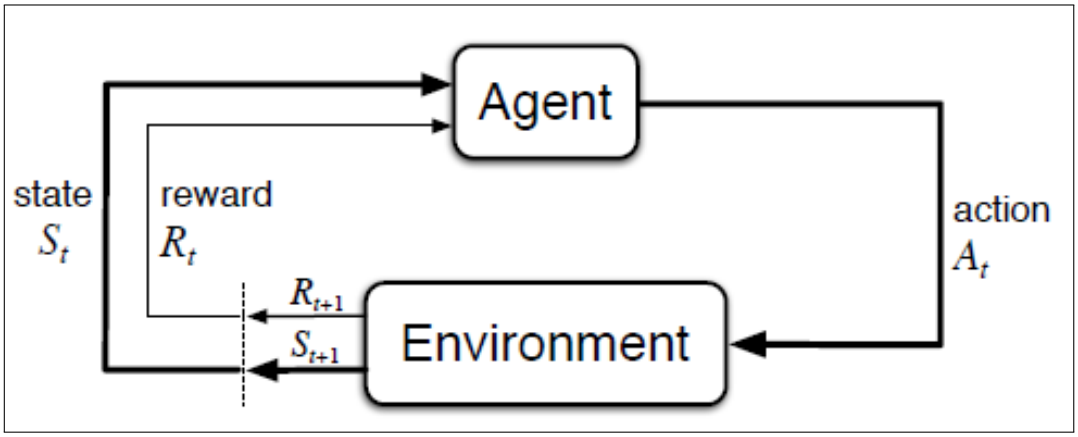

Figura 1.1. Schema del Reinforcement Learning [1]

La regola generale da considerare è: *tutto ciò che non può essere cambiato arbitrariamente dall'agente fa parte del suo ambiente*. Il confine tra agente e ambiente rappresenta il limite del **controllo assoluto** dell'agente, non della sua conoscenza.

1.3.1 Proprietà di Markov

La proprietà di Markov è un concetto fondamentale all'interno degli MDP. La proprietà di Markov afferma che lo stato futuro di un sistema dipende unicamente dallo stato corrente e dall'azione intrapresa, indipendentemente dalla storia passata degli stati e delle azioni. In altre parole, uno stato futuro è condizionato solo dallo stato corrente e dalla scelta dell'azione, senza considerare come si è arrivati a quel punto. Se un MDP soddisfa la proprietà Markov, allora il problema può essere drasticamente semplificato ed affrontato utilizzando il concetto di *valore di stato* o *valore di azione*, senza dover considerare l'intera storia delle transizioni precedenti.

1.3.2 Ambienti Parzialmente Osservabili

L'osservabilità completa è necessaria per i metodi di apprendimento basati sui processi decisionali di Markov ma purtroppo in molti ambienti reali, non sarà possibile per l'agente avere una percezione perfetta e completa dello stato dell'ambiente. Il modello formale a cui si fa riferimento è chiamato processo decisionale parzialmente osservabile di Markov o POMDP (Partially Observable Markov Decision Process) e vi sono diverse categorie di approcci pe questo tipo di problemi [2].

- **Politiche deterministiche senza stato**, ignorano la parziale osservabilità trattando le osservazioni come stati dell'ambiente. Tuttavia, questo approccio non garantisce un comportamento ottimale in tutte le occasioni.
- **Politiche stocastiche senza stato**, introducono casualità nelle azioni dell'agente per evitare di rimanere bloccati in situazioni ambigue. Tuttavia, trovare una politica stocastica globalmente ottimale è un problema *NP-Hard*.
- **Politiche con stato interno**, si basano sulla memoria delle azioni e delle osservazioni precedenti per distinguere lo stato corrente. Ci sono diversi approcci per apprendere tali politiche, come l'utilizzo di reti neurali ricorrenti o sistemi di classificazione. Inoltre, si possono utilizzare modelli nascosti di Markov per apprendere un modello dell'ambiente e delle sue transizioni.

1.4 Stima di Valore di Stato e Valore di Azione

Valori di funzioni associati a **stati** o a coppie **stato-azione** rappresentano quanto è vantaggioso trovarsi in un certo stato o intraprendere una data azione se si è in un certo stato, possono essere stimati dall' esperienza con le interazioni con l'ambiente, nel caso in cui il numero di queste interazioni si avvicini all' infinito, la media dei nostri **return** converge al valore del nostro stato. Possiamo quindi definire:

$$v_\pi(s) = \mathbb{E}_\pi[G_t|S_t = s] = \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)[r + \gamma v_\pi(s')]$$
$$q_\pi(s, a) = \sum_{s',r} p(s', r|s, a)[r + \gamma q_\pi(s', a')]$$

Dove:

- G_t è la nostra funzione di valore atteso all'istante di tempo t
- γ è detto fattore di sconto.
- $v_\pi(s)$ è il valore dello stato s sotto la policy π
- $q_\pi(s)$ è il valore della coppia stato-azione s, a sotto la policy π
- p è la probabilità di transizione verso s' con reward r dallo stato s eseguendo a

1.4.1 Cenni sulle Equazioni di ottimalità di Bellman

Le equazioni seguenti

- $v_*(s) = \max_a \sum_{s',r} p(s', r|s, a)[r + \gamma v_*(s')]$
- $q_*(s) = \sum_{s',r} p(s', r|s, a)[r + \gamma \max_a q_*(s', a')]$

Sono dette **equazioni di ottimalità di Bellman** e sono condizione necessaria per l'ottimalità, la risoluzione di tali equazioni se l'ambiente è perfettamente noto, si riduce alla risoluzione di un sistema di equazioni lineari, in tal caso parliamo di **Programmazione Dinamica** mentre la maggior parte degli algoritmi di intelligenza artificiale non opera su un assunzione di conoscenza perfetta dell'ambiente e quindi tenta di stimare tali valori ottimali gradualmente.

1.4.2 Bootstrapping & Discount Factor

Il **bootstrapping** nel contesto del reinforcement learning è un termine che viene usato per riferirsi all'utilizzo di stime o valori approssimati per aggiornare i valori di stato o azione durante il processo di apprendimento, in modo da renderlo molto più rapido. Questa tecnica consente all'agente di fare previsioni sulle ricompense future senza dover attendere il feedback completo. L'agente utilizza le informazioni disponibili per stimare i valori mancanti e migliorare l'efficienza dell'apprendimento. Tuttavia, l'uso del bootstrapping introduce il rischio di errori di stima, poiché le approssimazioni possono non essere sempre accurate. Pertanto, è necessario un equilibrio tra l'accelerazione dell'apprendimento e l'accuratezza delle stime per ottenere risultati ottimali, il cosiddetto **fattore di sconto** gioca un ruolo imporante in quanto si tratta di un coefficiente compreso tra **0** e **1** e geneeralmente una sua funzione pesa l'importanza delle ricompense future negli aggiornamenti delle stime.

1.4.3 Sfruttamento-Esplorazione

Bilanciare *sfruttamento* e *esplorazione* è una delle sfide principali degli algoritmi di Reinforcement Learning. Il concetto è in sostanza scegliere la frequenza relativa con cui selezionare, in ogni istante di tempo, **la migliore azione possibile** secondo le nostre stime (un'azione detta **greedy**) o un'azione detta **esplorativa.**

Scegliere un'azione greedy ovviamente è molto efficace sulla base di buone stime ma è una pessima strategia se le stime sull'ambiente non sono particolarmente buone. Scegliere un'azione esplorativa di contro, significa scegliere volontariamente un'azione subottimale. Il motivo di questa scelta riguarda l'ottenimento di nuove informazioni sull' ambiente dal momento che la "migliore azione", viene sempre selezionata sulla base delle nostre stime attuali, riguardanti gli stati che sono stati visitati e le azioni che sono state effettuate, ma questo rischia di precludere la visita di stati che anche se temporaneamente sembrano subottimali, sul lungo periodo portare l'agente a performare meglio.

In pratica, ottenendo più informazioni sull' ambiente le nostre stime tendono a migliorare e le prossime azioni greedy sono *meglio informate* e acquisiscono più valore. Questo effetto va ad ogni modo bilanciato perché scegliendo sempre un'azione esplorativa, si finisce per prendere "sempre" un'azione subottimale e di non convergere mai ad una politica ottimale o quasi-ottimale.

Vi sono differenti approcci a questa problematica, che non ha una soluzione oggettiva ed univoca per tutti i casi, i due approcci che verranno implementati nell'analisi seguente sono:

- **Politica** $\epsilon - greedy$, dove ϵ è la probabilità ad ogni passo (*step*, ovvero una singola interazione con l'ambiente) di scegliere un'azione **esplorativa** (e quindi $1-\epsilon$ è la probabilità di scegliere un'azione **greedy**). Questo tipo di politica sarà il tipo principale che sarà utilizzato nelle implementazioni degli algoritmi esposti nel prossimo capitolo, risulta spesso particolarmente semplice da implementare e con il giusto *tuning* del parametro ϵ, anche particolarmente efficace. E' bene notare che in **ambienti stazionari**, il risultato teorico più efficiente prevede di iniziare l'apprendimento con una percentuale di azioni esplorative e diminuirla gradualmente con l'aumento degli step, questo permette di ottenere il meglio dell' avere un'alta esplorazione iniziale (si analizzano più stati e si ottengono più informazioni sull' ambiente) e bassa finale (si tende a convergere ad un risultato senza rimanere bloccati in politiche subottimali). L'azione esplorativa viene, nel caso più semplice, scelta casualmente tra le disponibili, ciò è molto semplice da realizzare ma fa in modo che questa politica di *action selection* abbia ricevuto molte critiche relative alla sua caratteristica di avere le stesse probabilità, nella selezione di un'azione esplorativa, sia di selezionare un'azione che risulta in un percorso di azioni migliore sul lungo termine sia di selezionare un'azione pessima o subottimale anche per il lungo periodo.

- **Politica *Upper Confidence Bound***, che seleziona ad ogni istante di tempo un'azione in funzione dell'incertezza su quella stessa azione, l'algoritmo sceglierà l'azione con il valore massimo calcolato utilizzando la formula $[Q_t(a)+c\sqrt{\frac{\ln(t)}{N_t(a)}}]$, che tiene conto sia della stima corrente della qualità dell'azione che dell'incertezza associata ad essa. Ciò permette di bilanciare l'esplorazione di nuove azioni e lo sfruttamento delle azioni già conosciute come buone.

Ovviamente questi sono solamente due degli innumerevoli approcci possibili per la gestione del bilancio Esplorazione-Sfruttamento, altri approcci possibili possono essere ad esempio basati sulla distribuzione probabilistica delle azioni nel loro spazio, la *Boltzmann Exploration* ad esempio, funziona assegnando una priorità alle azioni in funzione dei loro valori attesi (e quindi della loro distribuzione).

Generalizzazione
Mappare ogni singolo stato ad ogni singola azione in tabelle di valori è un procedimento realistico ed efficace solamente con spazi di stato e azioni piuttosto ridotti in quanto, per spazi più grossi risulta particolarmente pesante dal punto di vista dei requisiti di memoria e del tempo di computazione. Altra caratteristica imporante è che un metodo tabulare fa un utilizzo inefficiente dell' esperienza, in quanto, se operiamo con ambienti con grandi spazi di stato e di azioni, spesso ci aspettiamo che stati simili possano avere azioni ottimali simili, mentre utilizzare metodi tabulari implica il trattamento di questi stati come completamente differenti. Apprendere con spazi di stato estesi richiedere tecniche di generalizzazione, che spesso però vanno adattate in maniera specifica al problema, vi sono diversi approcci che puntano alla cosiddetta *approssimazione di funzione* come reti neurali, logica fuzzy e metodi basati sulla memoria locale.

1.5 Cenni Generali sui metodi

Possiamo dare numerose suddivisioni dei metodi di Reinforcement Learning. Possiamo avere persino metodi che non stimano mai le cosiddette *Value Functions*, come ad esemptio **metodi evolutivi**, che applicano policy fisse e in risposta a variazioni casuali di tale policy seleziona gli agenti più performanti, in maniera concettualmente simile a come avviene l'evoluzione naturale.

1.5.1 Metodi Evolutivi

L'*evolutionary computation* è una branca di algoritmi di machine learning che si basa sulla simulazione dell'evoluzione di agenti [7], le più celebri varianti di questi algoritmi sono

- *Algoritmi Genetici*, intuitivi e di facile implementazione inizializzano una popolazione di policy e soluzioni che viene progressivamente alterata, successivamente le individuali soluzioni più performanti vengono selezionate per la successiva "generazione" di individui.
- *Strategie Evolutive*, in spazi di soluzione continui e utilizza "mutazioni" per generare nuove soluzioni individuali e selezionare deterministicamente i migliori agenti, questo sfruttando parametri esterni settati dallo sviluppatore, manualmente o mediante altre tecniche.
- *Programmmazione Genetica*, algoritmi il cui scopo principalmente è l'ottimizzazione dei programmi informatici. Ogni programma è rappresentato da un albero in cui i nodi interni sono gli operatori e le foglie sono le variabili. Gli operatori genetici includono l'eliminazione e l'aggiunta di nodi e sotto-alberi.

Nonostante i metodi evolutivi siano una branca di studio generalmente separata, non è rara la creazione di algoritmi considerati ibridi, nel quale paradigmi evolutivi vengono integrati in algoritmi di Reinforcement Learning o viceversa.
Ad esempio, il Q-Learning, algoritmo di Reinforcement Learning che vedremo nel dettaglio più avanti, ha riscontrato successo nell' essere combinato con altri algoritmi, ad esempio il *NEAT* (NeuroEvolution of Augmenting Topologies).

1.5.2 Metodi basati su Funzioni di Valore

Nel reame dei metodi basati sulle *Value Functions* troviamo

- **Programmazione Dinamica**, è un insieme di metodologie e algoritmi ideate per convergere a soluzioni ottimali se operanti su prefette rappresentazioni dell' ambiente sotto forma di un MDP, le tecniche di programmazione dinamica suddividono un problema complesso in sotto-problemi più piccoli, risolvendo e memorizzando i risultati dei sotto-problemi, si evita il calcolo ripetuto, migliorando l'efficienza. I due più celebri metodi di programmazione dinamica sono *Policy Iteration* e *Value Iteration*.

- **Metodi Monte-Carlo**, sono metodi che non hanno bisogno di un modello e di una perfetta rappresentazione del sistema, possono convergere alla soluzione basandosi solamente sulle interazioni con l'ambiente. I metodi Monte-Carlo si basano sui ritorni complessivi generati in determinati episodi da una policy, e soprattutto sulla media di essi, campionando interazioni con l'ambiente per un intero episodio e apprendendo da esso alla fine. I metodi Monte-Carlo risultano precisi e altamente adattabili a problemi di dimensione differente, tuttavia sono computazionalmente molto impegnativi e possono essere lenti a convergere.

- **Metodi Temporal Difference**, sono metodi che combinano intuizioni di metodi Monte-Carlo e di Programmazione Dinamica, guidano l'apprendimento mediante la sola esperienza e aggiornano le stime delle funzioni sulla base di altre stime senza attendere per il risultato finale dell' episodio. Per via della loro stessa natura, tendono a essere più dipendenti dalle condizioni iniziali del sistema, avendo un bias. In alcune circostanze, questo può portare a performance non propriamente ottimali e tale bias va considerato nell'applicazione di tali algoritmi [8].

Misura delle performance di apprendimento
Come si fa a valutare un algoritmo di Reinforcement Learning e quali sono le caratteristiche importanti da considerare nella valutazione della qualità dell' apprendimento? Dal punto di vista teorico può essere utile pensare ad un **eventuale convergenza all' ottimalità**, per alcuni algoritmi è stata infatti dimostrata la convergenza ad una soluzione ottimale, tuttavia per quanto questo possa essere un parametro da prendere in considerazione, in effetti, nella pratica risulta di poca utilità. Per fare un esempio pratico: è molto più utile un agente che converge velocemente al 99% dell' ottimalità che un'agente molto lento con una garanzia di convergenza dato un tempo infinito.

Risulta quindi più interessante giudicare la *velocità di convergenza all' ottimalità*, o l'ancora più utile in pratica velocità di convergenza alla *quasi-ottimalità*, che richiede ovviamente che venga definita la quasi-ottimalità sufficiente, parametro che varierà di caso in caso. Volendo paragonarlo con l' *apprendimento supervisionato* citato precedentemente, in quest'ultimo le caratteristiche più importanti da valutare tendono ad essere l'efficacia statistica dell' agente o l'accuratezza attesa.

Capitolo 2

Metodi Temporal Difference

Un *contro* importante dei metodi Monte-Carlo è che, basandosi solamente sul risultato del singolo episodio rischiano di validare, in seguito ad un episodio con esito positivo, tutto il comportamento dell'agente durante lo stesso, e di scartare tutto il comportamento di un agente in seguito ad un esito negativo. Questo può portare a convergenze particolarmente lente. Come accennato prima invece i metodi Temporal Difference (TD) aggiornano le stime delle funzioni sulla base di altre stime senza attendere per il risultato finale dell' episodio, sostanzialmente imparando ad ogni singola interazione con l'ambiente e sfruttando il bias intrinseco della correlazione tra gli stati dovuto alla Proprietà di Markov di cui prima. Possono essere implementati facilmente in maniera online e incrementale.

2.1 TD(0) e TD(λ)

Introduciamo prima i metodi più semplici di Temporal Difference, TD(0). Tale classe di metodi (sottoclasse della più ampia TD(λ)) contiene quei metodi che apprendono da un singolo step, pesando solo tale step nella fase di aggiornamento delle stime. Per un metodo di Temporal Difference, definiamo il cosiddetto **Errore di Temporal Difference**

$$\delta_t = R_{t+1} + \gamma V(S_{t+1}) - V(S_t)$$

Dove

- $V(S_t)$ è la stima del valore dello stato al tempo t.
- R_{t+1} è il **reward** al tempo $t+1$ (stato successivo)
- γ è il **discount factor**, un coefficiente che pesa quanto la stima dello stato *successivo* va considerata nell' aggiornamento della stima attuale

Questo errore di temporal difference è ciò che aggiorna le stime in un algoritmo TD(0), guardando un passo nel futuro e cambiando la valutazione dello stato attuale in base a quella dello stato successivo che viene raggiunto, il *come* questo stato successivo viene raggiunto sarà la differenza principale tra gli algoritmi TD(0) che vedremo in seguito. Un metodo TD(0) che stima il valore di uno stato, esegue

l'update della stima ad ogni step secondo la regola:

$$V(S_t) = V(S_t) + \alpha[\delta_t]$$

Ovvero

$$V(S_t) = V(S_t) + \alpha[R_{t+1} + \gamma V(S_{t+1}) - V(S_t)]$$

Dove

- α è il **learning rate** o **step-size parameter**, che determina quanto pesare la modifica dell'attuale stima

I Metodi TD(0) sono un caso particolare dei metodi TD(λ), dipendenti da un parametro λ, compreso tra 0 e 1, nel caso in cui ovviamente, tale parametro sia nullo. I metodi TD(λ) introducono un ulteriore peso da applicare all' errore-TD (δ_t) per ogni singolo stato visitato, tale parametro aggiuntivo è detto **eleggibilità** di uno stato, ed è definito come:

$$e(s_t) = \sum_{k=1}^{t} (\lambda\gamma)^{t-k} \delta_{s_t, s_k}$$

Dove

- $\delta_{s,s_k} = \begin{cases} 1 & \text{if } s = s_k \\ 0 & \text{altrimenti} \end{cases}$

E rappresenta il **grado di visita** di un determinato stato nel passato recente. Di conseguenza, per quanto riguarda il parametro λ :

- Se il parametro è 0, abbiamo un metodo TD(0)
- Se il parametro è 1, è approssimativamente equivalente ad aggiornare tutti gli stati in base al numero di volte in cui sono stati visitati entro la fine dell'episodio (più simile ad un metodo Monte-Carlo).

Learning rate e convergenza dei metodi TD(0)
A volte è conveniente variare il learning rate di volta in volta. Indichiamo con $\alpha_n(a)$ il learning rate utilizzato per elaborare la ricompensa ricevuta dopo l'n-esima selezione dell'azione a, la scelta $\alpha_n(a) = 1/\text{n}$ porta al cosiddetto *metodo della media campionaria*, che è garantito convergere ai veri valori delle azioni secondo la legge dei grandi numeri. Tuttavia, naturalmente, la convergenza non è garantita per tutte le scelte della sequenza $\alpha_n(a)$. Un risultato ben noto nella teoria dell'approssimazione stocastica ci fornisce le condizioni necessarie per garantire la convergenza con probabilità 1:

$$\sum_{n=1}^{\infty} \alpha_n(a) = \infty \qquad \sum_{n=1}^{\infty} \alpha_n^2(a) < \infty$$

La prima condizione è necessaria per garantire che i passi siano sufficientemente grandi da superare eventuali condizioni iniziali o fluttuazioni casuali. La seconda condizione garantisce che i passi diventino alla fine sufficientemente piccoli da garantire la convergenza.

E' inoltre dimostrabile che per qualsiasi politica fissata π, un metodo TD(0) converge a v_π. La convergenza avviene *in media*, per un learning rate costante e sufficientemente piccolo, e con probabilità 1 se il learning rate è variabile ma diminuisce secondo le condizioni di approssimazione stocastica enunciate prima.

Batch Updating e convergenza
Nel caso in cui avessimo a disposizione solo una quantità finita di esperienza, come un limite di episodi o passi temporali possiamo presentare ripetutamente l'esperienza fino a quando il metodo converge verso una risposta. Data una funzione valore approssimata, V , gli update delle stime vengono calcolati per ogni passo temporale t in cui viene visitato uno stato non terminale, ma nella tecnica di *batch updating* la funzione valore viene modificata solo una volta, sommando tutti gli incrementi. Quindi tutta l'esperienza disponibile viene elaborata nuovamente con la nuova funzione valore per produrre un nuovo incremento complessivo, e così via, fino a quando la funzione valore converge. Questo approccio viene chiamata in questo modo perché gli aggiornamenti vengono effettuati solo dopo l'elaborazione di ogni batch completo di dati di addestramento.

Con l'aggiornamento batch, un metodo TD(0) converge in modo deterministico verso una singola risposta indipendentemente dal learning rate α , a condizione che α sia scelto sufficientemente piccolo.

On Policy vs Off Policy
Definiamo un metodo come **On Policy** se tale algoritmo aggiorna le stime della stessa policy che utilizza per generare nuovi stati in quel momento (come ad esempio il SARSA), mentre un metodo **Off-Policy** utilizza due policy differenti, l'agente mantiene un modello di valore o di azione che stima il valore ottimo indipendentemente dalle azioni che sceglie effettivamente (ad esempio il Q-Learning o l'Expected Sarsa). Questa differenza tra poliche di apprendimento e di esplorazione può influenzare il trade-off tra l'esplorazione e lo sfruttamento dell'agente e può avere implicazioni sulla velocità e sulla stabilità dell'apprendimento

2.2 On Policy: SARSA

Introduciamo il primo algoritmo TD(0). Si tratta di un algoritmo On-Policy detto SARSA, il nome deriva dalla tupla S, A, R, S', A' ovvero *Stato-Azione-Reward-Stato (Prossimo)-Azione (Prossima)* e il suo funzionamento si basa sulla correzione progressiva della stima di una funzione $Q(S, a)$, che associa valori alle coppie *stato-azione*, lo scopo è ottenere una mappatura completa in tabella di tutte le combinazioni [9]. La correzione progressiva della stima avviene mediante la seguente regola di update, istanza di metodo TD(0)

$$Q(S, A) = Q(S, A) + \alpha[R + \gamma Q(S', A') - Q(S, A)]$$

In particolar modo, questo significa aggiornare la stima attuale in funzione di $Q(S', A')$. Fare ciò implica che la coppia stato-azione attuale acquisisca valore (o disvalore) proporzionalmente al valore della prossima coppia stato-azione, con "prossima" si intende la funzione Q dovuta a *stato successivo* e *l'azione che prenderemo quando saremo nello stato successivo* dove "azione che prenderemo" (A') in questo caso implica che stiamo utilizzando la stessa policy che abbiamo utilizzato per scegliere l'azione A, da cui la denominazione del SARSA come On-Policy.

Algorithm 1 Pseudocodice SARSA

Require: $\alpha \in (0, 1], \epsilon \in (0, 1], \gamma \in (0, 1], n \geq 0$ ▷ Iperparametri
  Inizializzare $Q(S, A)$ per ogni S e ogni A eccetto $Q(S_{\text{terminal}}, -) = 0$
  for $i \leq n$ **do** ▷ Per n episodi
    Inizializza S
    Scegli A da $\mathcal{A}(S)$ attraverso una policy π
    while S$\neq S_{\text{terminal}}$ **do** ▷ Per ogni step
      Esegui A
      Osserva R e S'
      Scegli A' da $\mathcal{A}(S')$ attraverso la stessa policy π
      $Q(S, A) \leftarrow Q(S, A) + \alpha[R + \gamma Q(S', A') - Q(S, A)]$
      $S \leftarrow S'$
      $A \leftarrow A'$
    end while
  end for

Si può notare come tale metodo sia un metodo On-Policy, di fatto la Q-Map che aggiorniamo ad ogni step, è la stessa che ad ogni step sfruttiamo, mediante magari una $\epsilon - Greedy$ per scegliere le nostre azioni. La scelta delle azioni può però essere fatta con una qualsiasi politica.

2.3 Off Policy: Q-Learning

Un algoritmo Off-Policy fra i metodi TD(0) è ad esempio il Q-Learning, sviluppato da Christopher Watkins nel 1989 [10] [11] è stato per anni uno degli algoritmi di Reinforcement Learning più utilizzati e celebri, La regola di update del Q-Learning è infatti, leggermente diversa da quella del SARSA

$$Q(S,A) = Q(S,A) + \alpha[R + \gamma \max_a Q(S',a) - Q(S,A)]$$

La differenza è ciò che rende questo metodo Off-Policy, difatti nonostante venga utilizzata una certa policy per selezionare l'azione A, la stima Stato-Azione non viene aggiornata in funzione della prossima azione che prenderemmo se seguissimo la stessa policy, ma viene aggiornata in funzione della "migliore aziore azione disponibile" dal prossimo stato. In pratica, se il prossimo stato possiede un'azione particolarmente di valore tra le selezionabili, andare in quello stato è un'azione di valore.

Algorithm 2 Pseudocodice Q-Learning

Require: $\alpha \in (0,1], \epsilon \in (0,1], \gamma \in (0,1], n \geq 0$ ▷ Iperparametri
  Inizializzare $Q(S,A)$ per ogni S e ogni A eccetto $Q(S_{\text{terminal}}, -) = 0$
  for $i \leq n$ **do** ▷ Per n episodi
    Inizializza S
    while S$\neq S_{\text{terminal}}$ **do** ▷ Per ogni step
      Scegli A da $\mathcal{A}(S)$ attraverso una policy π
      Esegui A
      Osserva R e S'
      $Q(S,A) \leftarrow Q(S,A) + \alpha[R + \gamma \max_a Q(S',a) - Q(S,A)]$ ▷ Greedy update
      $S \leftarrow S'$
    end while
  end for

La discrepanza principale è che il Q-Learning presenta una differente update rule, di conseguenza non c'è bisogno nemmeno di proiettare preventivamente l'azione che prenderemmo, in questo caso. Anche in questo caso possiamo implementare qualsiasi politica per la selezione delle azioni.

2.4 Off Policy: Expected SARSA

L'Expected SARSA è un'altro algoritmo TD(0) e l'unica differenza dal Q-Learning è effettivamente l'update rule che risulta essere:

$$Q(S,A) = Q(S,A) + \alpha[R + \gamma\mathbb{E}[Q(S',A'|S')] - Q(S,A)]$$

Ciò significa semplicemente che va calcolato il valore atteso della prossima coppia stato-azione, sapendo lo stato successivo. Il valore atteso viene calcolato sommando tutti i valori $Q(S,a)$ pesati con le probabilità che le rispettive azioni vengano scelte. Nelle nostre implementazioni di $\epsilon-Greedy$, nel caso di una scelta esplorativa abbiamo un'azione casuale, in tal caso le probabilità di selezione sono

$$\pi(S,a) = \begin{cases} 1 - \epsilon + \dfrac{\epsilon}{|\mathcal{A}|} & \text{Se } a = a_* \\ \epsilon\frac{|\mathcal{A}|-1}{|\mathcal{A}|} & \text{Altrimenti} \end{cases}$$

Dove

- $\mathcal{A}$ = Spazio di tutte le azioni disponibili
- $|\mathcal{A}|$ = Cardinalità dello spazio delle azioni

La sua implementazione è generalmente considerata Off-Policy tuttavia vi sono versioni di questo algoritmo implementate On-Policy. Peculiarità di tale algoritmo è la capacità di performare bene anche con learning rate più alti (cosa generalmente non vera per SARSA e Q-Learning).

Capitolo 3

Analisi e Confronto

Il capitolo seguente espone la risoluzione mediante SARSA, Q-Learning ed Expected SARSA di un ambiente di controllo discreto noto come "problema del Taxi" con successivo confronto di prestazioni tra i vari algoritmi.

3.1 Descrizione del problema del Taxi

Il problema del taxi riguarda la navigazione all'interno di un ambiente a griglia al fine di raggiungere i passeggeri, prenderli e successivamente lasciarli in una delle quattro posizioni disponibili [12] [13].
Nel problema del taxi, siamo in un ambiente di griglia 5x5 con quattro posizioni designate per il prelievo e la consegna (Rosso, Verde, Giallo e Blu). Il taxi parte da una casella casuale e il passeggero si trova in una delle posizioni designate. L'obiettivo è spostare il taxi nella posizione del passeggero, prenderlo a bordo, spostarsi verso la destinazione desiderata del passeggero e poi lasciarlo. Una volta che il passeggero viene lasciato, l'episodio termina. Il giocatore riceve ricompense positive per il corretto deposito del passeggero nella posizione corretta. Riceve invece ricompense negative per tentativi errati di prelievo o consegna del passeggero, nonché per ogni passo compiuto senza ricevere altre ricompense.

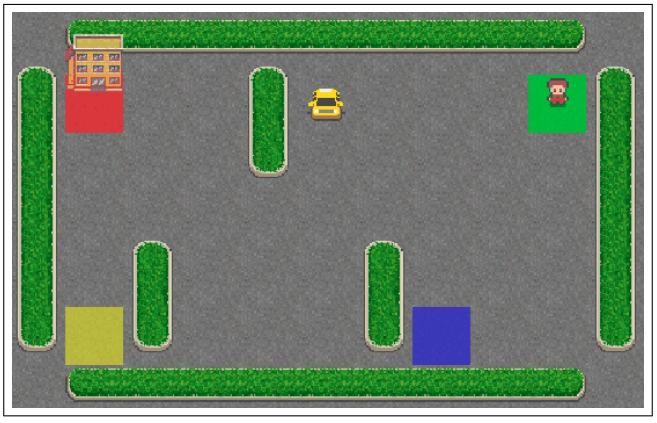

Figura 3.1. Mappa di Esempio

Lo spazio delle **azioni** contiene:

- 0: Spostati verso sud (giù)
- 1: Spostati verso nord (su)
- 2: Spostati verso est (destra)
- 3: Spostati verso ovest (sinistra)
- 4: Prendi il passeggero
- 5: Lascia il passeggero

Lo spazio delle **osservazioni** contiene:

- Posizione del Passeggero (Rosso/Verde/Giallo/Blu/A Bordo)
- Posizione della Destinazione (Rosso/Verde/Giallo/Blu)
- Posizione del Taxi

Abbiamo 500 stati discreti, ma solo 400 sono raggiungibili se eliminiamo gli stati in cui passeggero e destinazione si trovano immediatamente nello stesso luogo. Aggiungendo i 4 stati "terminali" in cui taxi e passeggero sono entrambi nella posizione di destinazione, abbiamo uno spazio di 404 stati.

I reward sono

- -1 Ogni passo se non si ricevono altri reward
- -10 per azioni di Pick-Up o Drop-Off fallita
- +20 per episodio portato a termine correttamente

Analisi e Confronti
La maggior parte delle analisi seguenti saranno attuate variando alcuni parametri su più algoritmi, in maniera standardizzata in modo da avere risultati comparabili. Occasionalmente vi saranno simulazioni fuori dallo schema per mettere in luce alcuni comportamenti peculiari. Per valutare un agente, dopo l'addestramento, è stato utilizzato anche un **Greedy Tester**, un programma che simula l'environment e prende la migliore azione conosciuta dall'agente ad ogni step, testando la sua abilità per 100.000 Episodi e fornendo la media dei reward per episodio.

Implementazione Politiche di esplorazione
Segue l'implementazione di una politica $\epsilon - Greedy$

Algorithm 3 Politica $\epsilon - Greedy$ (con selezione casuale)

Require: $\epsilon \in (0, 1], S \neq S_{\text{terminal}}$
  $N_{\text{rand}} \leftarrow$ Numero casuale tra 0 e 1
  if $N_{\text{rand}} < \epsilon$ **then**
    Seleziona casualmente $A \in \mathcal{A}(S)$
  else
    Seleziona $A = a$ tale per cui $Q(S, a) = \max Q(S, A)$ con $A \in \mathcal{A}(S)$
  end if

Nel caso invece di un algoritmo di *Upper Confidence Bound.* l'algoritmo sceglierà l'azione con il valore massimo calcolato utilizzando la formula $[Q_t(a) + c\sqrt{\frac{\ln t}{N_t(a)}}]$, che tiene conto sia della stima corrente della qualità dell'azione che dell'incertezza associata ad essa. Ciò permette di bilanciare l'esplorazione di nuove azioni e lo sfruttamento delle azioni già conosciute come buone.

In tale formula

- $Q_t(a)$, il valore stimato della nostra azione per questo istante di tempo, nel nostro caso sarà, per i nostri algoritmi $Q(S, a)$
- c il **grado di esplorazione** del nostro algoritmo, maggiore sarà e più sarà *importante* valutare l'incertezza di un'azione per la scelta
- t è lo step attuale di tempo (globale, non per il singolo episodio)
- $N_t(a)$ è il numero di volte in cui l'azione a è stata scelta (globalmente, anche questo)

Se un'azione è stata scelta 0 volte, deve essere scelta, è considerata un azione massimizzante. Un possibile modo di implementare l'Upper Confidence Bound è il seguente:

Algorithm 4 Politica Upper Confidence Bound

Require: $S \neq S_{\text{terminal}}, N_t(A) \geq 0$
  Inizializza UCBval(S,A) = 0 per ogni $A \in \mathcal{A}(S)$
  for $A \in \mathcal{A}(S)$ **do**
    if $N_t(A) = 0$ **then**
      $N_t(A) \leftarrow N_t(A) + 1$
      Seleziona A ▷ Algoritmo viene interrotto alla selezione di un'azione
    else
      UCBval(S,A)$\leftarrow Q(S, A) + c\sqrt{(\ln(t)/N_t(A))}$
    end if
  end for
  $N_t(A) \leftarrow N_t(A) + 1$
  Seleziona $A = a$ tale per cui UCBval$(S, a) = \max$ UCBval(S, A) con $A \in \mathcal{A}(S)$

3.2 Soluzione mediante SARSA e Q-LEARNING

Prendiamo innanzitutto in considerazione gli algoritmi SARSA e Q-Learning, rispettivamente On e Off Policy. Nel nostro caso, utilizzeremo valore iniziale di ϵ che faremo decrescere fino al valore finale entro la metà del numero di episodi scelti per l'addestramento. Questa sarà una costante per tutti i test che verranno eseguiti d'ora in poi. In parole povere, in ogni simulazione avremo un decay della nostra ϵ ad ogni episodio, pari a $\epsilon_{decay} = 2\epsilon/N_{ep}$

Consideriamo una prima simulazione con i seguenti parametri

- Numero di Episodi = 100
- $\epsilon_0 = 1$
- $\epsilon_{\text{final}} = 0.05$
- α (Learning Rate) = 0.1
- γ (Discount Factor) = 0.95
- Politica di Esplorazione = Sampling Casuale

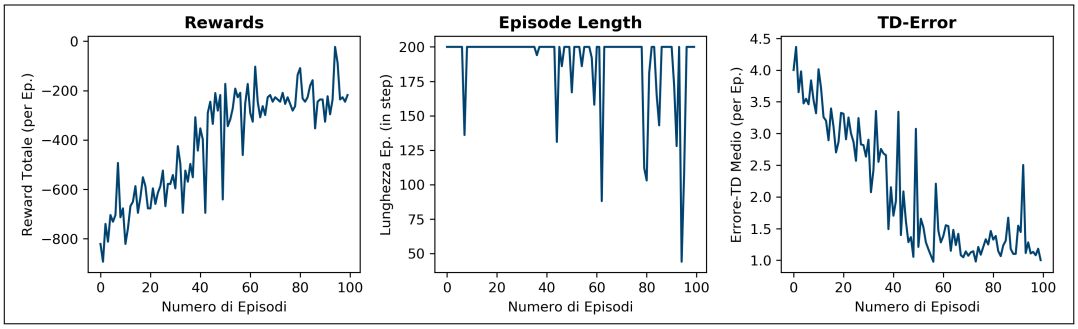

Figura 3.2. Algoritmo SARSA, 100 Ep.

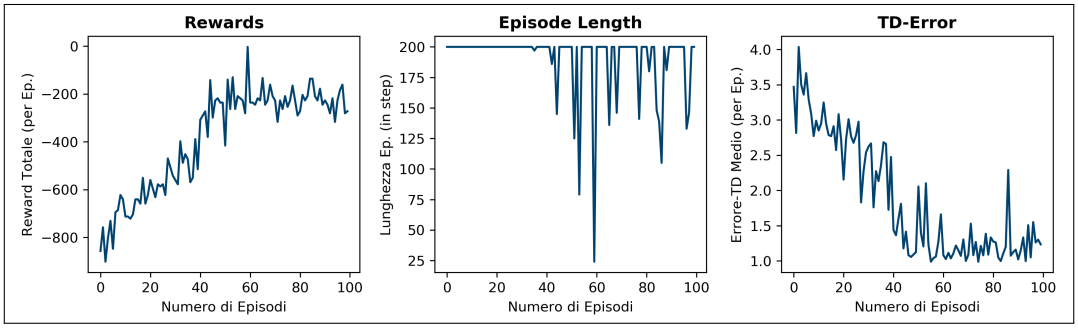

Figura 3.3. Algoritmo Q-Learning, 100 Ep

Questi grafici mostrano i **reward** per ogni episodio, la **lunghezza** di ogni episodio e la media dell'errore di temporal difference, per ogni episodio. Come possiamo intuire dai grafici, nonostante le ricompense aumentino, sintomo che l'agente stia effettivamente imparando, 100 episodi sono chiaramente insufficienti ad entrambi i nostri algoritmi per sperare di convergere ad una soluzione.

Di seguito simulazioni di addestramento di 1000 episodi con lo stesso set di parametri e politiche, per entrambi gli algoritmi.

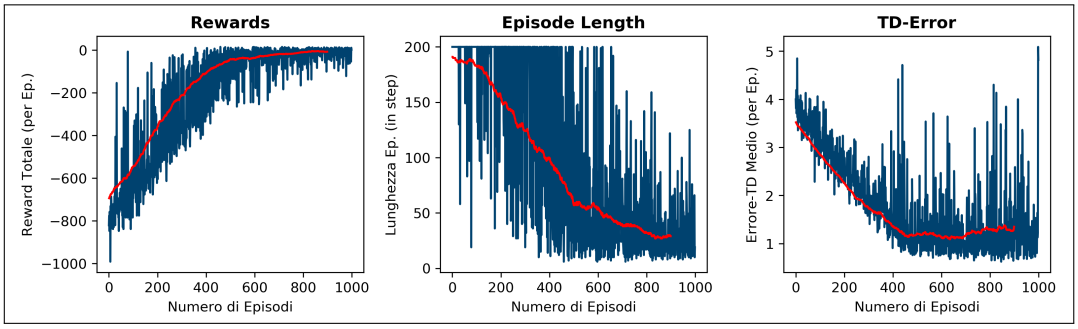

Figura 3.4. Algoritmo SARSA, 1000 Ep

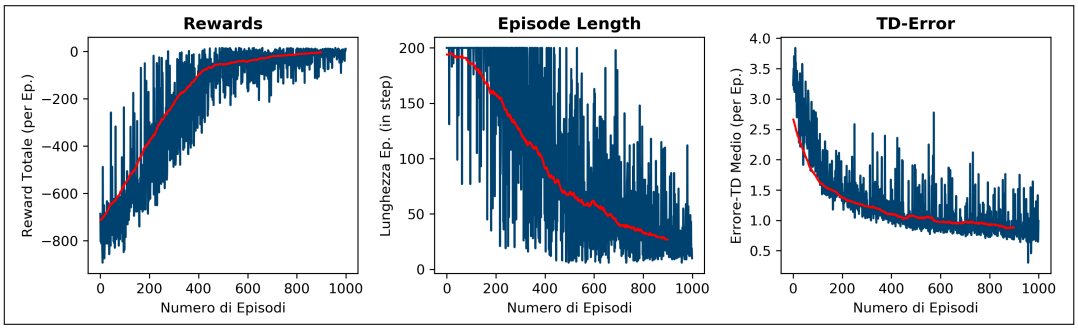

Figura 3.5. Algoritmo Q-Learning, 1000 Ep

In rosso è evidenziata la media mobile per 100 episodi di tutti i valori resi sul grafico. Guardando anche la prossima immagine, relativa al Q-Learning è evidente come le situazioni siano praticamente analoghe e le differenze siano minime.
Possiamo notare come il TD-Error sia minore per il Q-Learning che per il SARSA sin dall' inizio, tuttavia appaiono convergere nella stessa zona.

Un'altra differenza è che la soluzione a cui giungono entrambi è, però, ancora subottimale. Nel dettaglio il **Greedy Tester** citato prima, fornisce per questa simulazione un reward medio di -134.918 per il SARSA e di -118.825 per il Q-Learning, entrambi molto negativi, ma questo ci mostra come il Q-Learning già risulti un algoritmo più rapido nella convergenza, rispetto al SARSA.
Per ottenere una politica che approssima una politica ottimale, tuttavia, dobbiamo almeno iterare per più episodi, almeno 10.000.

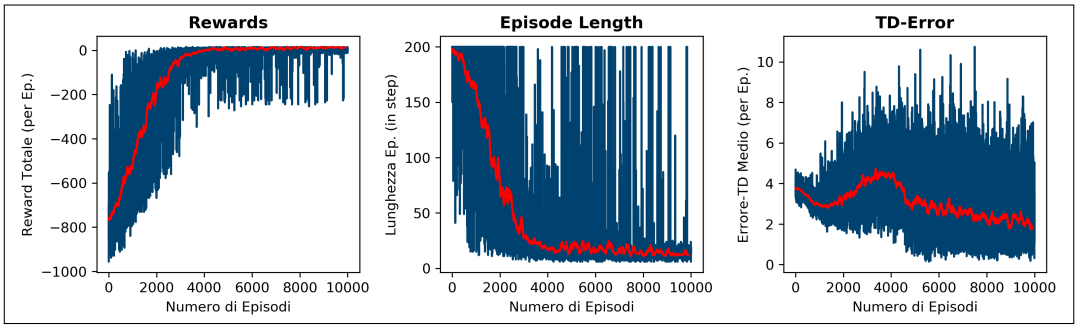

Figura 3.6. Algoritmo SARSA, 10.000 Ep

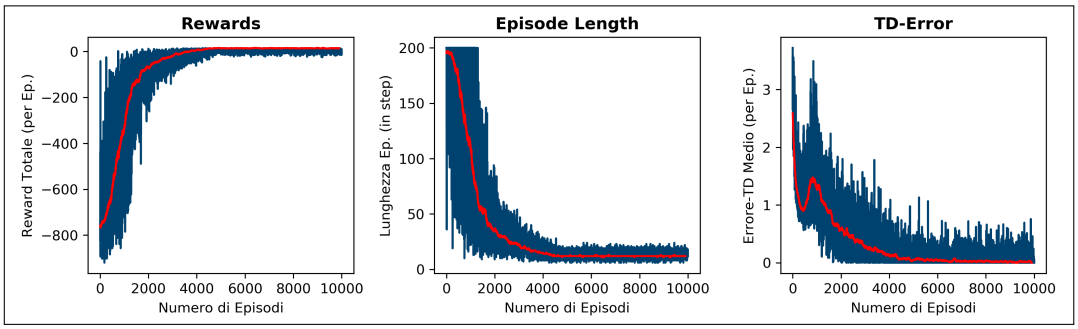

Figura 3.7. Algoritmo Q-Learning, 10.000 Ep

Iniziamo a notare delle differenze più importanti, specialmente nella velocità di convergenza. E' chiaro sia osservando la lunghezza degli episodi che il training error che non solo il Q-Learning risulta particolarmente più veloce ma che intorno a 5.000 episodi ha iniziato a stabilizzarsi e a convergere ad una soluzione mentre il SARSA sembra ancora non aver raggiunto completamente una soluzione.
Osserviamo i reward degli algoritmi più da vicino:

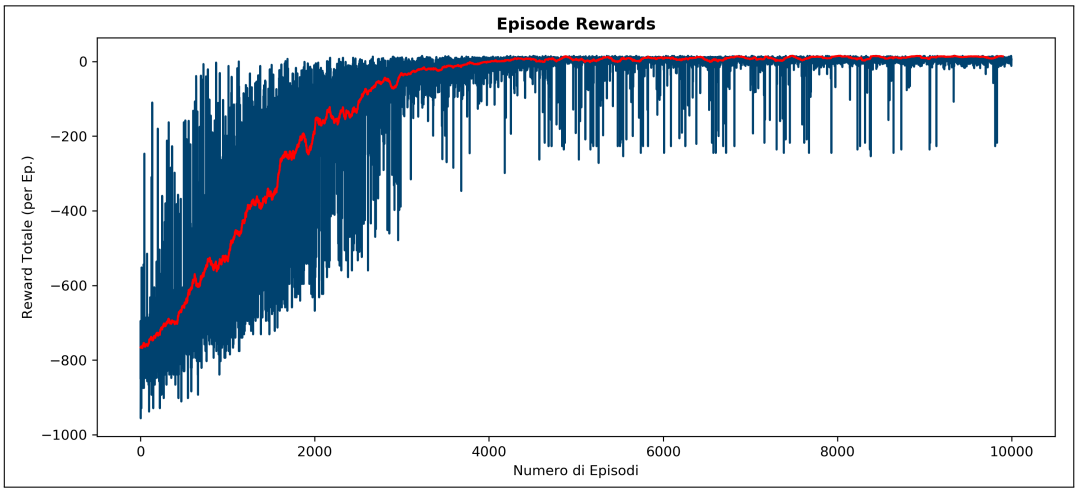

Figura 3.8. SARSA

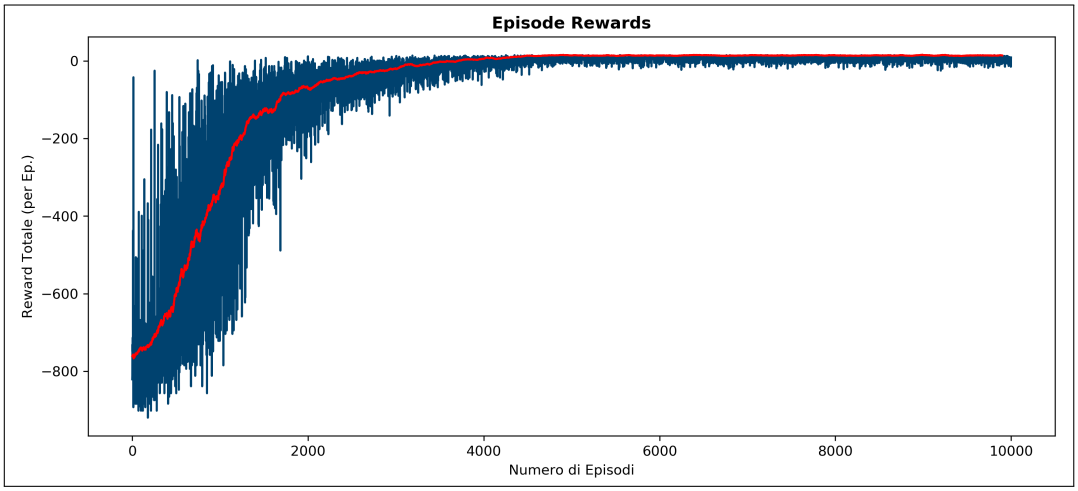

Figura 3.9. Q-Learning

Il **Greedy Tester** per l'agente SARSA fornisce un reward medio di 7.6314.
E' difficile, in molti ambienti di Reinforcement Learning sapere se la convergenza è avvenuta a una policy ottimale, soprattutto considerando il fatto che stiamo, in queste simulazioni, utilizzando un learning rate **fisso** e che non diminuisce gradualmente. Per questo ambiente, empiricamente considereremo l'ottimalità della policy di un agente quella che, testata dal **Greedy Tester** avrà un reward medio tra 7.8 e 7.9, in quanto nessun esperimento ha portato risultati migliori e nella visualizzazione dell' ambiente è chiaramente verificabile una policy ottimale.
Il **Greedy Tester** per il Q-Learning fornisce infatti, in questo caso, un reward medio di 7.9415. Ovviamente, dato che l'ambiente ha un certo grado di randomizzazione, il reward medio fluttuerà sempre un po', motivo per il quale viene fatta una media su 100.000 Episodi di test, pratica che riduce tale effetto.

3.2.1 Esempio di Politica Ottimale

Illustriamo un piccolo esempio di una politica ottimale a cui converge un agente

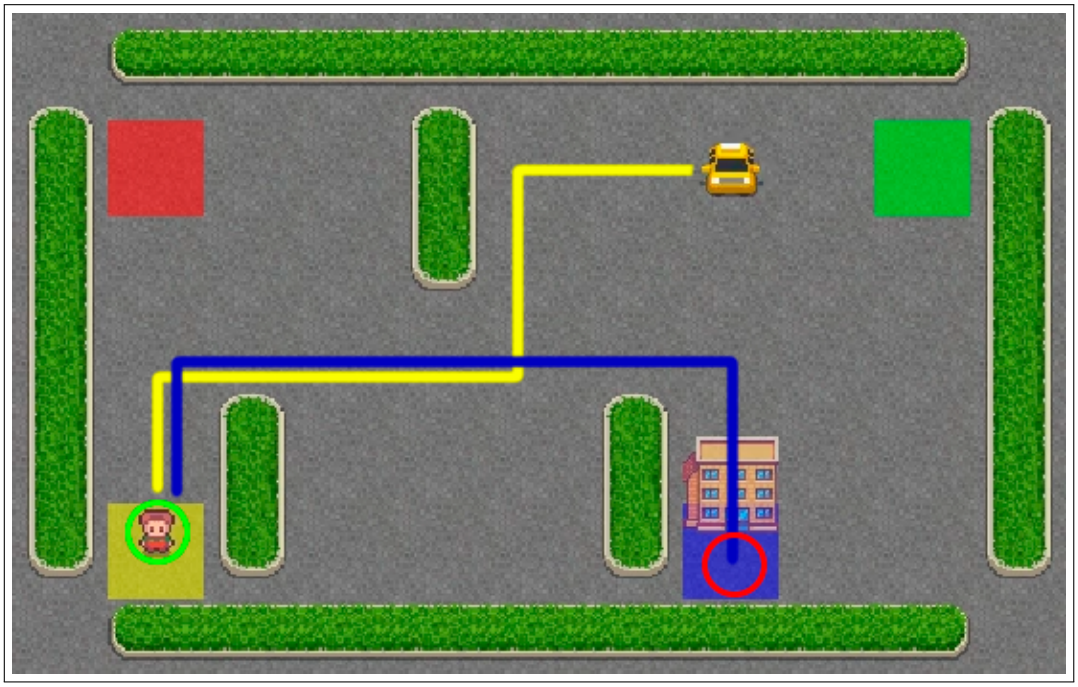

Figura 3.10.
Linea Gialla:Percorso Posizione Iniziale-Passeggero
Linea Blu:Percorso Passeggero-Destinazione
Cerchi Verde/Rosso: Operazioni di Pick-Up/Drop-Off

3.2.2 Variazione Parametri

Ora, partendo da questa configurazione di parametri come benchmark, variamone uno alla volta per studiare in che modo influenzano l'apprendimento degli algoritmi e per vedere se la variazione di uno stesso parametro può essere gestita diversamente da algoritmi differenti.

Riduzione Learning Rate α=0.001

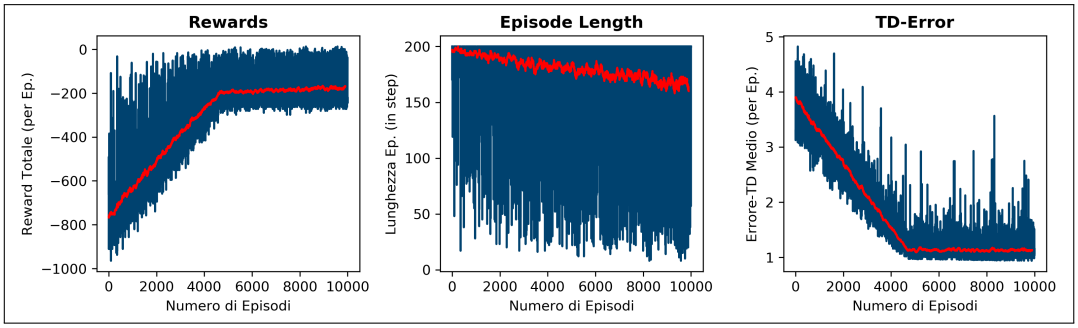

Figura 3.11. SARSA, α=0.001

Ridurre il Learning Rate da α =0.1 a α =0.001, risulta in un apprendimento molto più lento, tuttavia, con un errore medio minore per il SARSA rispetto al caso precedente.

E' interessante notare l'interazione tra il parametro di esplorazione e il Learning Rate, in quanto è visibile, soprattutto nel grafico dei reward, il cambio di andamento nell'apprendimento dell'agente quando, a metà addestramento, l'esplorazione smette di variare e si stabilizza.

Per il Q-Learning, abbiamo un comportamento perfettamente analogo.

Riduzione Esplorazione Iniziale $\epsilon_0 = 0.30$ (30%)

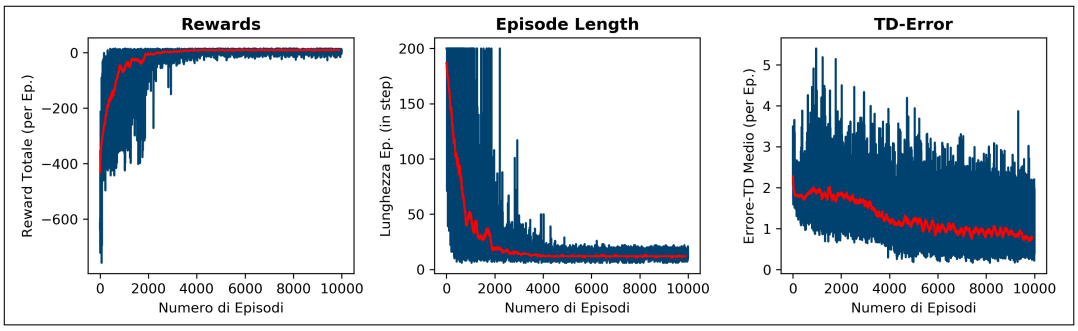

Figura 3.12. SARSA, $\epsilon_0 = 0.30$ (30%)

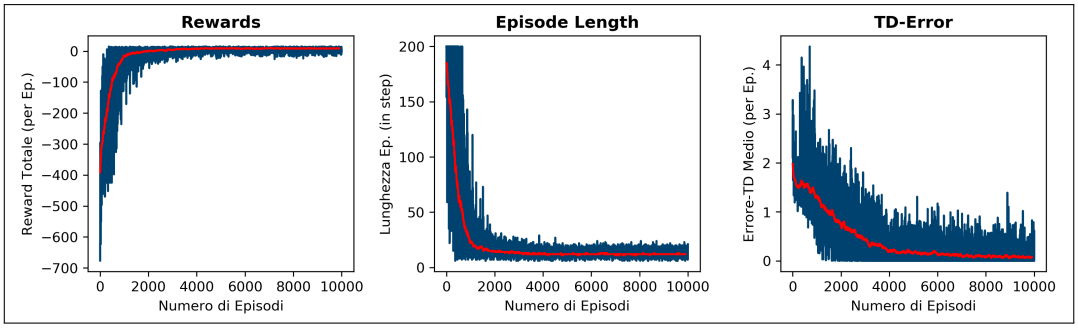

Figura 3.13. Q-Learning, $\epsilon_0 = 0.30$ (30%)

I due algoritmi sembrano gestire allo stesso modo la riduzione dell'esplorazione iniziale. Settata a 30%, un valore più ragionevole per questo ambiente del 100% impostato inizialmente, entrambi gli algoritmi riescono a convergere alla soluzione in maniera più rapida. Questo è anche un esempio di come il Q-Learning sia meno intaccato negativamente da un alto valore esplorativo, in quanto i miglioramenti per l'algoritmo non sono particolarmente ingenti, segno di una buona performance anche con i valori precedenti.

Riduzione Discount Factor $\gamma = 0.8$

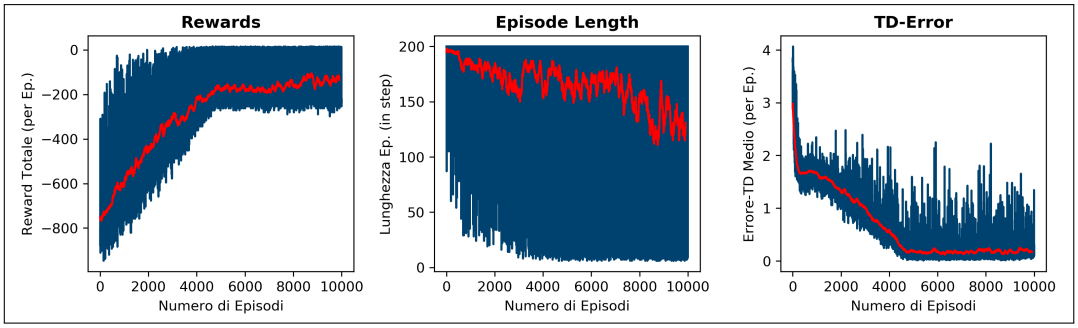

Figura 3.14. SARSA, $\gamma = 0.8$

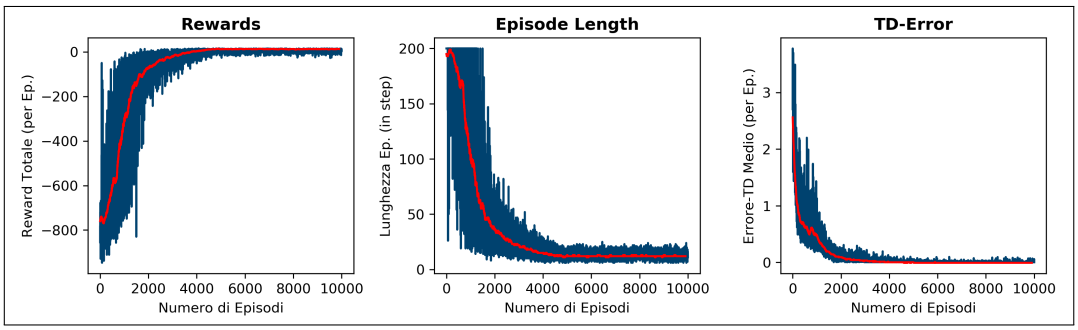

Figura 3.15. Q-Learning, $\gamma = 0.8$

Per quanto riguarda la diminuzione del discount factor, da γ=0.95 a γ=0.8 abbiamo una differenza drastica. Il Discount Factor, per ripetere colloquialmente, è un parametro che pesa l'importanza del guardare "tot" passi nel futuro (nel nostro caso, uno) nella fase di apprendimento dell' agente. Tale diminuzione sembra intaccare la prestazione del Q-Learning in maniera molto meno importante che nel caso del SARSA, dove troviamo la convergenza e i miglioramenti rallentare drasticamente (si riduce, anche in questo caso, l'errore medio)

3.3 Variazioni dell' Action Selection

3.3.1 Seconda Migliore Azione

E' interessante confrontare anche gli algoritmi secondo una politica di esplorazione leggermente diversa. Fino ad adesso, abbiamo scelto casualmente le nostre **azioni esplorative**, indipendentemente dal loro valore. In questo caso, introduciamo una modifica che sceglie come azione esplorativa sempre la **seconda migliore azione**, quindi la migliore tra le **non-greedy**.

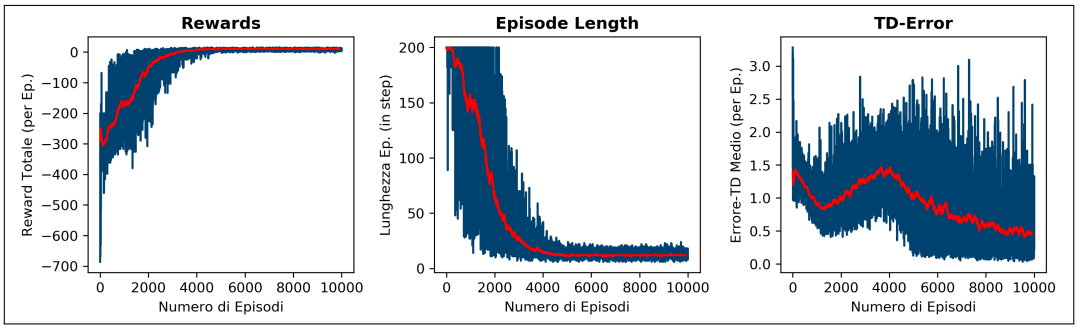

Figura 3.16. SARSA, Action Selection Modificata

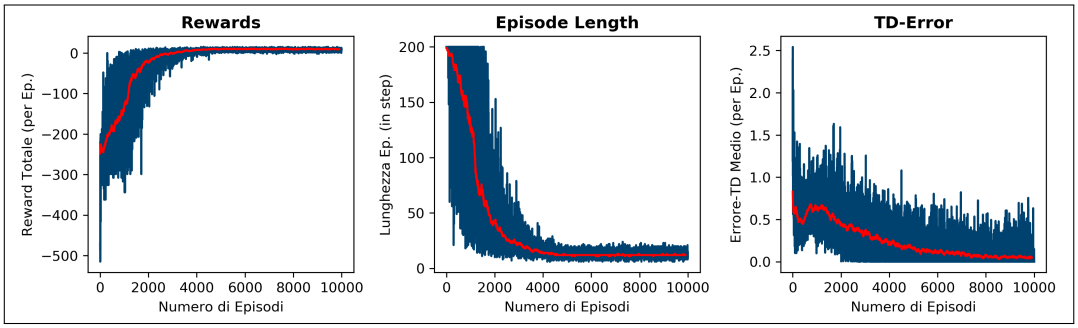

Figura 3.17. Q-Learning, Action Selection Modificata

Anche se non presentano grandi differenze nel risultato finale, sono chiari i miglioramenti del SARSA per quanto riguarda la velocità di convergenza di questa seconda politica di esplorazione, ricordando che la versione precedente era:

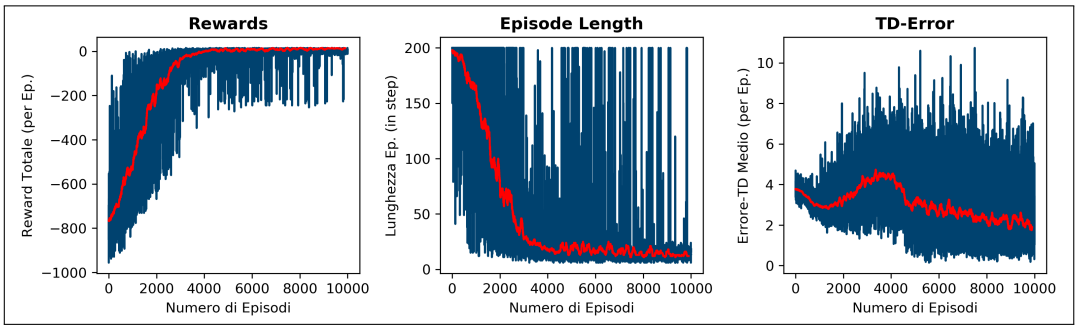

Figura 3.18. SARSA, Random Sampling

Interessante notare come, sia per il SARSA, sia per il Q-Learning, **ridurre il learning rate** contemporaneamente alla modifica della politica di Action Selection non produce un apprendimento visivamente diviso in due parti come prima, in quanto lo stacco tra politica di esplorazione ed exploitation è molto meno sentito, ci ritroviamo ad avere un apprendimento molto più uniforme.

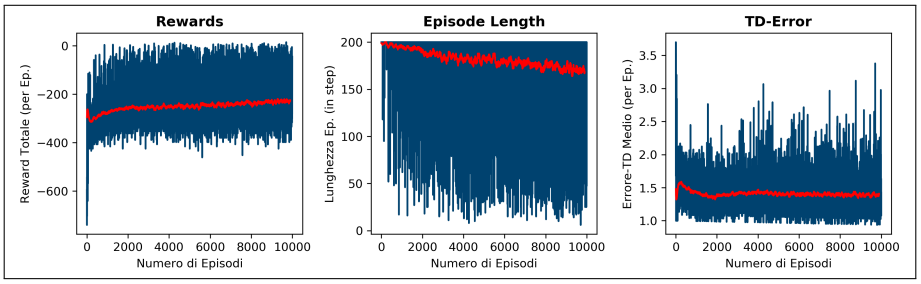

Figura 3.19. SARSA, Learning Rate ridotto e Action Selection Modificata

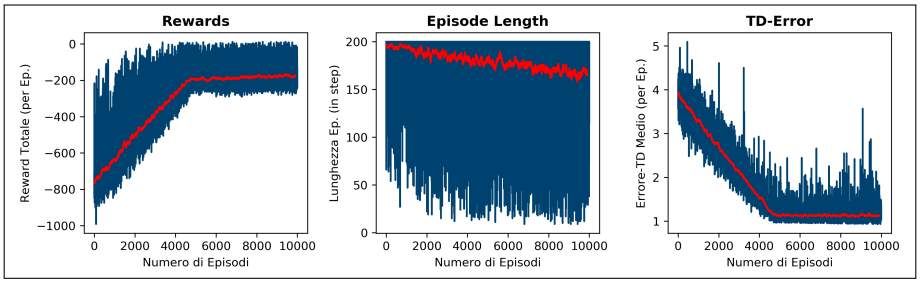

Figura 3.20. SARSA, Learning Rate ridotto e Random Sampling

Lo stesso identico fenomeno si manifesta analogamente nel Q-Learning.

3.3.2 Upper Confidence Bound

E' possibile inoltre, cambiare totalmente la nostra politica di action selection, passando da una $\epsilon - Greedy$ ad una **Upper Confidence Bound.** L'algoritmo assegna un valore a ciascuna azione, che indica la fiducia dell'algoritmo nella bontà di quell'azione. Questo valore viene calcolato utilizzando sia le ricompense osservate delle azioni che una stima dell'incertezza associata a ciascuna azione. Maggiore è l'incertezza su una azione, più tale azione viene selezionata, nel dettaglio viene associato ad ogni azione a il valore $[Q_t(a) + c\sqrt{\frac{\ln t}{N_t(a)}}]$ dove, per ripetere:

- $Q_t(a)$, il valore stimato della nostra azione per questo istante di tempo, nel nostro caso sarà, per i nostri algoritmi $Q(S, a)$
- c il **grado di esplorazione** del nostro algoritmo, maggiore sarà e più sarà *importante* valutare l'incertezza di un'azione per la scelta
- t è lo step attuale di tempo (globale, non per il singolo episodio)
- $N_t(a)$ è il numero di volte in cui l'azione a è stata scelta (globalmente,anche questo)

Questo algoritmo, generalmente è più complesso da adattare ad ambienti di Reinforcement Learning disparati tra loro, tende a perfomare meglio delle politiche $\epsilon - Greedy$ come si può notare da questi confronti per soli 5.000 Episodi.

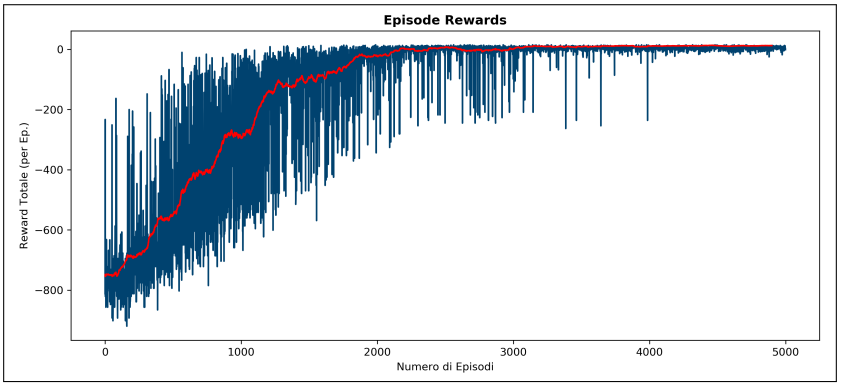

Figura 3.21. SARSA, $\epsilon - Greedy$ w/Random Sampling,
$\epsilon_0 = 1, \epsilon_{final} = 0.05, \alpha = 0.1, \gamma = 0.95,\ Greedy - Tester(100k) : 6.7774$

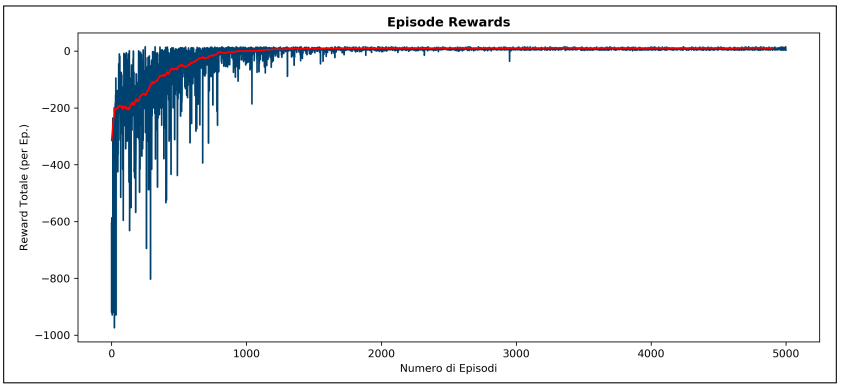

Figura 3.22. SARSA, Upper Confidence Bound,
$c = 2, \alpha = 0.1, \gamma = 0.95,\ Greedy - Tester(100k) : 7.847$

E' verificabile lo stesso effetto sulla riduzione del learning rate visibile con la $\epsilon - Greedy$ e la *seconda azione migliore*, ma inoltre, con una politica Upper Confidence Bound il SARSA sembra non risentire più in maniera eccessiva della riduzione del discount factor, al contrario di prima.

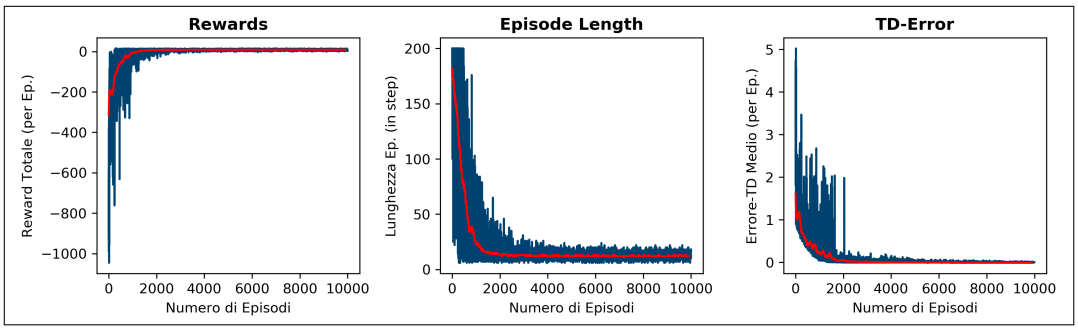

Figura 3.23. SARSA, Upper Confidence Bound,$c = 2, \alpha = 0.1, \gamma = 0.8$ (*ridotto*),

3.4 Expected Sarsa

E' opportuno citare un altro celebre algoritmo (generalmente Off-Policy) oltre il Q-Learning, detto Expected Sarsa

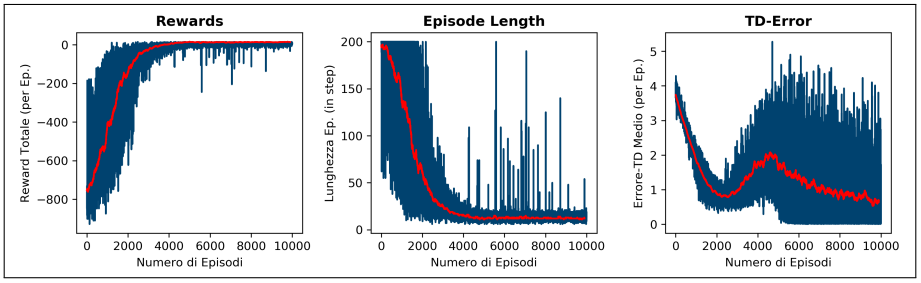

Figura 3.24. Expected Sarsa, $\epsilon - Greedy$
Greedy-Tester (100k): 7.1308

Tuttavia tale algoritmo, in questo ambiente e con questi parametri, non converge con 10.000 Episodi, come vediamo dal tester. Ciò è realisticamente dovuto al valore finale di ϵ mantenuto per tutti questi test, pari a 0.05 (5%). Dato che l'Expected Sarsa esegue gli update basandosi sul valore atteso, pesando le varie azioni con la loro probabilità di essere prese, l'algoritmo cerca di adattarsi troppo ad un ambiente che in questo caso, risulta piuttosto semplice, mettendoci più tempo ad imparare. Volendo tenere questi parametri per ottenere una convergenza abbiamo bisogno di un maggiore numero di episodi. Una caratteristica interessante tuttavia, dell' Expected SARSA, è che al contrario del SARSA, performa bene su lunghi periodi anche con learning rate molto maggiori, possiamo aumentare il learning rate a 1 e confrontarli.

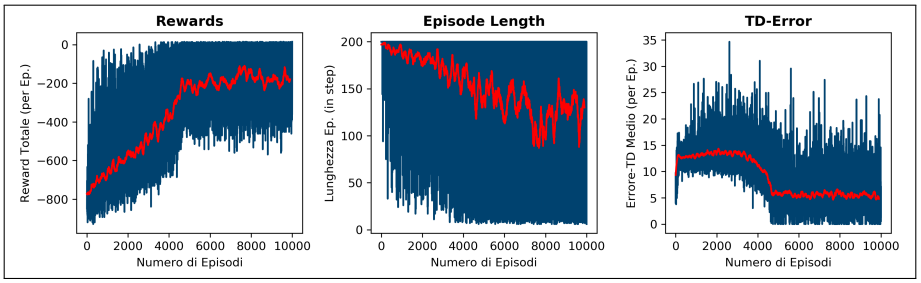

Figura 3.25. SARSA, $\epsilon - Greedy$ w/Random Sampling,
$\epsilon_0 = 1, \epsilon_{final} = 0.05, \alpha = 1, \gamma = 0.95$

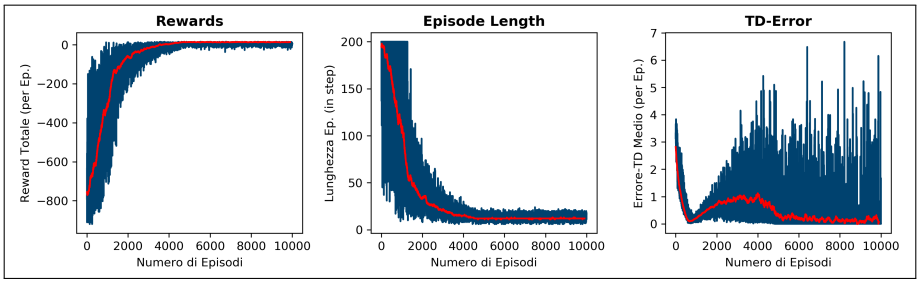

Figura 3.26. Expected Sarsa, $\epsilon - Greedy$ w/Random Sampling,
$\epsilon_0 = 1, \epsilon_{final} = 0.05, \alpha = 1, \gamma = 0.95$

Conclusione

Il Reinforcement Learning è un ramo dell'apprendimento automatico ispirato all' addestramento animale ed esistono molteplici algoritmi che, partendo da questa idea, gestiscono l'apprendimento di un agente. Gli algoritmi confrontati in questo testo fanno parte della categoria TD(0) (Temporal Difference), ed aggiornano le stime di una coppia Stato-Azione guardando un passo nel futuro, se opportunamente configurati attraverso i cosiddetti Iperparametri.

Nello specifico sono stati confrontati, in un ambiente discreto, il SARSA, Q-Learning e l'Expected SARSA rispetto alle variazioni dei seguenti Iperparametri: Learning Rate, Esplorazione iniziale, Discount Factor. Sono state considerate inoltre politiche esplorative differenti: due varianti di $\epsilon-$Greedy e la politica Upper Confidence Bound.

Tutti gli algoritmi convergono ad una soluzione, tuttavia il Q-Learning si è dimostrato più efficace e rapido del SARSA nella convergenze ad una politica attimale, risentendo meno negativamente delle variazioni legate al parametro di esplorazione iniziale e al discount factor, oltre che risentire molto meno della variazione nelle politiche di action selection. Introdurre politche più efficienti di Action Selection ha infatti beneficiato principalmente il SARSA, riuscendo a portare l'algoritmo ad un livello di prestazioni buono e ad una convergenza rapida, inoltre, l'introduzione di politche come l'Upper Confidence Bound, hanno reso l'algoritmo SARSA meno sensibile alle variazioni del discount factor rispetto a prima. Nell' Expected SARSA, computazionalmente leggermente più pesante, abbiamo riscontrato una velocità di convergenza leggermente minore rispetto agli altri due, ma caratteristiche intermedie di stabilità per il processo di apprendimento, soprattutto per quanto riguarda learning rate particolarmente elevati, che non hanno rallentato il training o portato a problemi di convergenza, anzi, hanno avuto l'effetto opposto.

Appendice

La seguente appendice contiene un'implementazione del Q-Learning in Python 3.7.0, testata su una macchina con:

- OS: Windows 7 Ultimate 64-bit
- CPU: Intel Core i7 2600K @ 3.40GH
- RAM: 16,0GB Dual-Channel DDR3 @ 668MHz

```
import gymnasium as gym
import time
import pygame
import numpy as np
from collections import defaultdict
from tqdm import tqdm
import matplotlib.pyplot as plt
from enum import Enum
from math import log as ln
from math import sqrt


def default_actions():
    return np.zeros(env.action_space.n)

#DIVISORE TESTUALE
def divisor(length=20):
    ret=""
    for i in range(length):
        ret=ret+"-"
    print("\n"+ret)

#CLASSE AGENTE ------------------------------------------------
class Taxi_Driver:
    def __init__(
            self,
            learning_rate: float,
            initial_epsilon: float,
            epsilon_decay: float,
            final_epsilon: float,
            actionspace: int,
            discount_factor: float
            ):

        #INIZIALIZZA Q-MAP
        self.q_values= defaultdict(default_actions)
```

```
37
38          #INIZIALIZZA IPERPARAMETRI
39          self.lr = learning_rate
40          self.disc_factor = discount_factor
41          self.epsilon = initial_epsilon
42          self.epsilon_decay = epsilon_decay
43          self.final_epsilon = final_epsilon
44
45          self.actionspace=actionspace
46          self.global_action_counts=np.zeros(actionspace)
47
48
49      #SELEZIONI AZIONI --------------------------------
50
51      #GREEDY ---
52      def get_greedy_action(self,state):
53          actions=self.q_values[state]
54          return int(np.argmax(actions))
55
56      #EPSILON-GREEDY ---
57      def get_eps_greedy_random(self,state):
58          if np.random.random()<self.epsilon:
59              return int(env.action_space.sample())
60          else:
61              return self.get_greedy_action(state)
62
63      #EPSILON-GREEDY con SECONDA MIGLIORE AZIONE ---
64      def get_eps_greedy_SBA(self,state):
65          if np.random.random()<self.epsilon:
66              actions=self.q_values[state]
67              x=np.argsort(actions)
68              return int(x[-2])
69          else:
70              return self.get_greedy_action(state)
71
72      #UPPER CONFIDENCE BOUND ---
73      def get_UCB(self,state,t):
74          self.localUCBValues=np.zeros(self.actionspace)
75          for i in range(self.actionspace):
76              if self.global_action_counts[i]==0:
77                  self.global_action_counts[i]+=1
78                  return i
79              else:
80                  uncertainty=sqrt((ln(t)/self.global_action_counts[i])
    )
81                  self.localUCBValues[i]=self.q_values[state][i]+self.
    epsilon*uncertainty
82          index=np.argmax(self.localUCBValues)
83          self.global_action_counts[index]+=1
84          return index
85
86      #FUNZIONE DI UPDATE ---
87      #   LA FUNZIONE DI UPDATE, PER OGNI UPDATE RITORNA IL TD-ERROR
88      def update(
89              self,
90              state,
91              action: int,
```

```
92              reward: float,
93              terminated: bool,
94              next_state
95              ):
96                  future_Q_val=(not terminated) * np.max(self.q_values[
    next_state])
97                  temporal_difference = (reward + self.disc_factor*
    future_Q_val-self.q_values[state][action])
98                  self.q_values[state][action]=(self.q_values[state][
    action]+self.lr*temporal_difference)
99
100                 return abs(temporal_difference)
101
102     def decay_epsilon(self):
103         self.epsilon = max(self.final_epsilon, self.epsilon -
    epsilon_decay)
104
105
106 #SETTING ------------------------------------------------
107
108 #BOOLEANI PER SELEZIONE
109 correctchoice=False
110 randomexp=False
111 sbaexp=False
112 ucbexp=False
113
114 #POLITICA DI ESPLORAZIONE ----------------
115
116 while not correctchoice:
117     choose=input("Exploration Method? [Random/SBA/UCB] ")
118     if choose.upper()=="RANDOM":
119         randomexp=True
120         correctchoice=True
121     elif choose.upper()=="SBA":
122         sbaexp=True
123         correctchoice=True
124     elif choose.upper()=="UCB":
125         ucbexp=True
126         correctchoice=True
127     else:
128         print("Inserito input non previsto, ritentare")
129
130
131 #IPERAPARAMETRI ----------------
132
133 n_episodes=int(input("Number of Episodes: "))
134 learning_rate=float(input("Learning Rate: "))
135 discount_factor=float(input("Discount Factor: "))
136 if not ucbexp:
137     start_epsilon=float(input("Starting Epsilon (Default=1): "))
138     final_epsilon=float(input("Final Epsilon (Default=0.05): "))
139     # DECAY: RIDUCE l'ESPLORAZIONE GRADUALMENTE
140     epsilon_decay = start_epsilon / (n_episodes / 2)
141 else:
142     start_epsilon=float(input("Exploration Degree: "))
143     epsilon_decay=False
144
```

```
145
146 #CREAZIONE AMBIENTE ---------------------------------------------
147
148 env = gym.make('Taxi-v3')
149 env = gym.wrappers.RecordEpisodeStatistics(env, deque_size=n_episodes
    )
150 trainerrorsums=[] #Coda degli errori
151
152 #NASCITA AGENTE ---------------------------------------------
153 Agent=Taxi_Driver(learning_rate,start_epsilon,epsilon_decay,
    final_epsilon,env.action_space.n,discount_factor)
154
155 divisor()
156 globalsteps=0 #step complessivi
157 for episode in tqdm(range(n_episodes), desc="Q-Learning"):
158     state,info = env.reset()
159     done= False
160     episodetotalerror=0
161     steps=0 #step per episodio corrente
162     while not done:
163
164         if randomexp:
165             action=Agent.get_eps_greedy_random(state)
166         elif sbaexp:
167             action=Agent.get_eps_greedy_SBA(state)
168         elif ucbexp:
169             action=Agent.get_UCB(state,globalsteps)
170
171         new_state, reward, terminated, truncated, info =env.step(
    action)
172
173         #UPDATE Q-MAP e Aggiornamento dell'errore dell' episodio
174         episodetotalerror=episodetotalerror+Agent.update(state,action
    ,reward,terminated, new_state)
175
176         #UPDATE STATE AND AGENT
177         done=terminated or truncated
178
179         state=new_state
180         steps+=1
181         globalsteps+=1
182
183     #REDUCE EXPLORATION
184     Agent.decay_epsilon() #Riduzione esplorazione
185     episodetotalerror=episodetotalerror/steps #Errore totale diventa
    la media
186     trainerrorsums.append(episodetotalerror) #Coda degli errori medi
187
188 #GREEDY TESTING ---------------------------------------------
189
190 allow=int(input("Simulare? 1/0: "))
191
192 if allow:
193     tests=int(input("Per quanti episodi testare?: "))
194     env = gym.make('Taxi-v3')
195     env = gym.wrappers.RecordEpisodeStatistics(env, deque_size=
    n_episodes)
```

```
196
197     for episode in tqdm(range(tests)):
198         state,info = env.reset()
199         done= False
200
201         while not done:
202             action=Agent.get_greedy_action(state) #SELEZIONA GREEDY
203             new_state, reward, terminated, truncated, info =env.step(
    action)
204
205             #UPDATE STATE AND AGENT
206             done=terminated or truncated
207             state=new_state
208
209
210     returns=np.array(env.return_queue) #Coda dei Return
211     avg=np.mean(returns) #Media della coda dei Return
212     print("L'Agente ha prodotto una media reward di ",avg)
213
214 env.close()
```

Bibliografia

[1] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. MIT press, 2018.

[2] L. P. Kaelbling, M. L. Littman, and A. W. Moore, "Reinforcement learning: A survey," *Journal of artificial intelligence research*, vol. 4, pp. 237–285, 1996.

[3] R. Bellman, "Dynamic programming," *Science*, vol. 153, no. 3731, pp. 34–37, 1966.

[4] E. L. Thorndike, *Animal intelligence: Experimental studies*. Transaction Publishers, 1911.

[5] A. Turing, "Intelligent machinery (1948)," 2004.

[6] M. Minsky, "Steps toward artificial intelligence," *Proceedings of the IRE*, vol. 49, no. 1, pp. 8–30, 1961.

[7] M. M. Drugan, "Reinforcement learning versus evolutionary computation: A survey on hybrid algorithms," *Swarm and Evolutionary Computation*, vol. 44, pp. 228–246, 2019.

[8] R. S. Sutton, "Learning to predict by the methods of temporal differences," *Machine learning*, vol. 3, pp. 9–44, 1988.

[9] G. A. Rummery and M. Niranjan, *On-line Q-learning using connectionist systems*, vol. 37. University of Cambridge, Department of Engineering Cambridge, UK, 1994.

[10] C. J. C. H. Watkins, "Learning from delayed rewards," 1989.

[11] C. J. Watkins and P. Dayan, "Q-learning," *Machine learning*, vol. 8, pp. 279–292, 1992.

[12] T. G. Dietterich, "Hierarchical reinforcement learning with the maxq value function decomposition," *Journal of artificial intelligence research*, vol. 13, pp. 227–303, 2000.

[13] F. Foundation, "Taxi problem, gymnasium environment implementation." `https://gymnasium.farama.org/environments/toy_text/taxi/`. Accessed: 09/07/2023.

Ringraziamenti

Grazie ai miei genitori, Giuseppe e Carla, per avermi dato la possibilità di studiare e avermi sempre sostenuto durante il percorso, avendomi sempre messo nella miglior condizione possibile semplificandomi la vita in ogni modo.

Grazie a tutti i miei amici, a tutte le persone frequentate spesso in questi anni, molte delle quali ormai frammentate in tanti piccoli gruppi e disperse in molti luoghi, difficili da coltivare tutte contemporaneamente e difficili da elencare in maniera completa senza, quindi, fare scontento qualcuno, ma per tutte le persone che si sono laureate (o si dovranno laureare) che mi hanno sopportato nei miei sproloqui nel frattempo, principalmente in università, grazie mille.

Soprattutto, grazie alle cialde del Caffè Borbone.