SAPIENZA
UNIVERSITÀ DI ROMA

Apprendimento per Rinforzo basato su Q-Learning per Agenti Competitivi: Strategie di Attacco e Difesa in un Ambiente Grid-World

Facoltà di Ingegneria dell'Informazione, Informatica e Statistica
Laurea in Ingegneria Informatica e Automatica

Leonardo Angelini
Matricola 1983722

Relatore
Prof. Antonio Pietrabissa

Correlatore
Danilo Menegatti, PhD

Anno Accademico 2024/2025

Apprendimento per Rinforzo basato su Q-Learning per Agenti Competitivi: Strategie di Attacco e Difesa in un Ambiente Grid-World

Sapienza Università di Roma

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

Email dell'autore: angelini.1983722@studenti.uniroma1.it

"Fatti non foste a viver come bruti,
ma per seguir virtute e canoscenza"

Sommario

L'automatica ha come scopo principale la progettazione di sistemi che mantengano il comportamento desiderato di un processo, sia in presenza di perturbazioni esterne che di incertezze interne. Questo campo di studio si focalizza sullo sviluppo di dispositivi e di algoritmi che possano regolare autonomamente l'azione di un sistema per raggiungere e mantenere uno stato operativo ottimale detto riferimento. Per raggiungere tale obiettivo si sfruttano naturalmente supporti informatici, difatti per questo progetto di tesi si è adoperato uno dei tre pilastri fondanti del Machine Learning (ML): il Reinforcement Learning (RL).

Lo scopo del documento sarà innanzitutto quello di fornire le basi teoriche rigorose per comprendere al meglio il contesto del ML e in particolar modo del RL con un livello di dettaglio via maggiore in funzione degli strumenti che saranno utilizzati per la parte simulativa. Il RL nella letteratura scientifica è sfruttato come metodo di apprendimento funzionale alla risoluzione di una vasta gamma di task (compiti) senza necessitare di una conoscenza a priori delle dinamiche d'ambiente del problema.

In particolar modo, si vorrà mettere in evidenza l'apprendimento di due agenti che interagiranno e formuleranno strategie in funzione dei loro rispettivi ruoli di Attaccante e Difensore. Questo scenario pone le basi dei sistemi multi agente, dall'inglese Multi-Agent Systems (MAS), nei quali un certo numero di agenti interagiscono contemporaneamente con il medesimo ambiente al passare del tempo al fine di perseguire il proprio obiettivo, che può essere individuale oppure condiviso.

Il presente elaborato si propone di unire il paradigma del RL con quello dei MAS, fornendo una panoramica ad alto livello dell'apprendimento per rinforzo multi agente, dall'inglese Multi-Agent Reinforcement Learning (MARL), il quale trova applicazione in molteplici ambiti quali:

- Sorveglianza e sicurezza
- Ricerca e soccorso
- Monitoraggio ambientale

I campi applicativi sopra menzionati rappresentano solo alcuni esempi della totalità degli ambiti nei quali la presente tecnologia viene sfruttata, basandosi su un approccio interdisciplinare fra diverse aree dell'ingegneria.

A tal proposito, la tesi propone dapprima una attenta analisi della letteratura scientifica relativa ai più recenti avanzamenti metodologici in ambito MARL e relative applicazioni all'avanguardia, per poi focalizzarsi sull'implementazione di uno degli algoritmi di RL (Q-Learning) ad un problema di navigazione in ambiente grid-world concernente due agenti, aventi rispettivamente il compito di distruggere e di difendere una determinata cella; ne consegue che il primo dovrà mettere a punto una strategia di attacco, mentre il secondo una di difesa. Infine, i risultati delle simulazioni vengono presentati e commentati puntualmente. La struttura risultante consiste in:

- **Capitolo 1:** Introduzione al RL e descrizione dei principali algoritmi;
- **Capitolo 2:** Implementazione dell'algoritmo Q-Learning a scopi progettuali;
- **Capitolo 3:** Descrizione dei risultati delle simulazioni;
- **Capitolo 4:** Conclusioni.

Infine, per completezza, l'**Appendice A** reca il codice sorgente dell'algoritmo implementato.

Indice

Elenco delle figure

Capitolo 1

Introduzione

1.1 Machine Learning

Il RL è una sottocategoria dell'apprendimento automatico, dall'inglese Machine Learning (ML), branca dell' Intelligenza Artificiale (AI) il cui obiettivo consiste nel "consentire ai calcolatori e alle macchine di imitare il modo in cui gli esseri umani imparano, realizzano task in maniera autonoma, e migliorano la loro performance e accuratezza attraverso l'esperienza e l'esposizione a più dati" [1].

Sfruttare la potenza computazionale dei moderni calcolatori permette di accelerare in modo significativo la risoluzione di problemi complicati in molteplici ambiti, dalla sicurezza informatica alla finanza, dalla medicina ai sistemi di controllo.

Nonostante siano numerosi gli algoritmi di apprendimento automatico sviluppati nel corso degli anni, tutti quanti condividono l'obiettivo di inferire conoscenza a partire da dati, non a caso si parla di *data-driven programming*; in breve, il particolare algoritmo considerato risulta in grado di apprendere, attraverso un set di regole a lui specifiche, le proprietà statistiche dei dati di cui dispone mediate un processo di *trial-and-error*, a fini di generalizzazione. A tal proposito, è possibile distinguere tre paradigmi di apprendimento:

- Apprendimento Supervisionato, dall'inglese *Supervised Learning*
- Apprendimento Non Supervisioanto, dall'inglese *Unsupervised Learning*
- Apprendimento per Rinforzo, dall'inglese *Reinforcement Learning*

In generale, tali approcci differiscono in base alla tipologia di dati utilizzati durante l'apprendimento.

L'**Apprendimento Supervisionato** si basa su set di dati etichettati, ovvero dati annotati con informazioni aggiuntive in grado di descrivere o classificare il contenuto, si pensi a immagini recanti etichette che descrivono la classe di appartenenza ad

esempio. Il compito di algoritmi basati su dati di questo tipo è quello di identificare e modellare schemi ricorrenti, come accade in compiti di classificazione e regressione.

L'**Apprendimento Non Supervisionato** si basa su dati non etichettati, pertanto compito dell'algoritmo è quello di identificare autonomamente schemi ricorrenti, raggruppandoli in categorie basate su somiglianze o correlazioni. L'apprendimento non supervisionato viene utilizzato quando i dati non presentano risposte strutturate o oggettive, ad esempio per identificare gruppi di clienti con comportamenti o preferenze simili.

L'**Apprendimento per Rinforzo** consente l'apprendimento di una strategia di controllo attraverso l'interazione tra un controllore, detto agente, e un sistema dinamico, detto ambiente, ricevendo un feedback, sotto forma di ricompense e penalità, in base al tipo di interazione avuta. Obiettivo dell'agente è quello di individuare la migliore strategia di controllo possibile al fine di massimizzare le ricompense ricevute. L'RL trova applicazioni in numerosi campi, quali la guida autonoma e la robotica.

La Figura 1.1 descrive graficamente la suddivisione dell'apprendimento automatico nelle categorie sopra descritte. Si noti che, in ogni caso, l'obiettivo che gli algoritmi perseguono è la generalizzazione delle proprie previsioni su dati non impiegati ai fini dell'apprendimento [2].

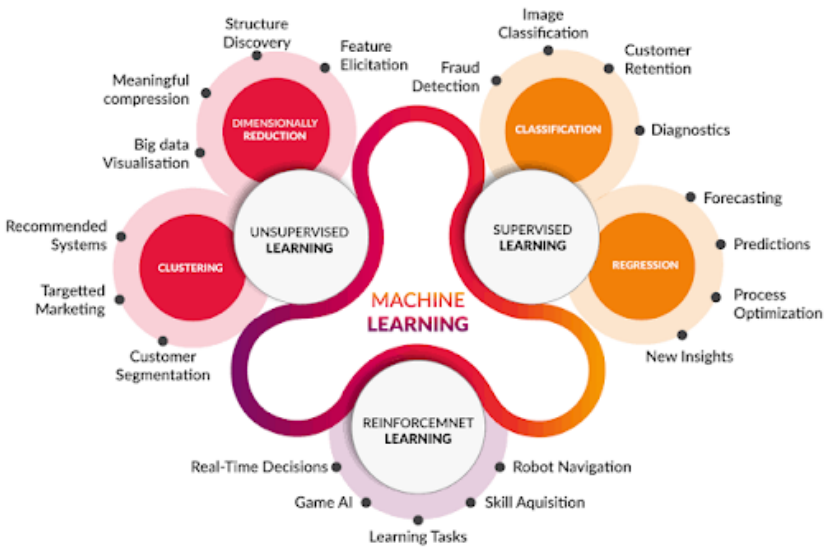

Figura 1.1. I tre pilastri dell'apprendimento automatico.

1.2 Reinforcement Learning

L'apprendimento per rinforzo, come accennato nella sezione precedente, rappresenta la tipologia di apprendimento automatico più adeguata per l'apprendimento di strategie di controllo; del resto, viene definito come "una tecnica di ML per la risoluzione di problemi di controllo, anche detti decisionali, tramite la costruzione di un agente che impara dall'ambiente interagendo con esso attraverso *tentativi ed errori*, ricevendo ricompense, positive o negative, come feedback" [3].

Il processo iterativo alla base dell'apprendimento di una strategia di controllo viene illustrato visivamente nella Figura 1.2. L'interazione può essere così descritta; l'agente, a partire dall'osservazione dello stato ("States") dell'ambiente, elabora una azione ("Actions") secondo una certa strategia, a partire dalla quale l'ambiente muta il suo stato; oltre a osservare il nuovo stato, l'agente riceve un feedback ("Reward") che descrive quantitativamente l'efficacia del suo operato.

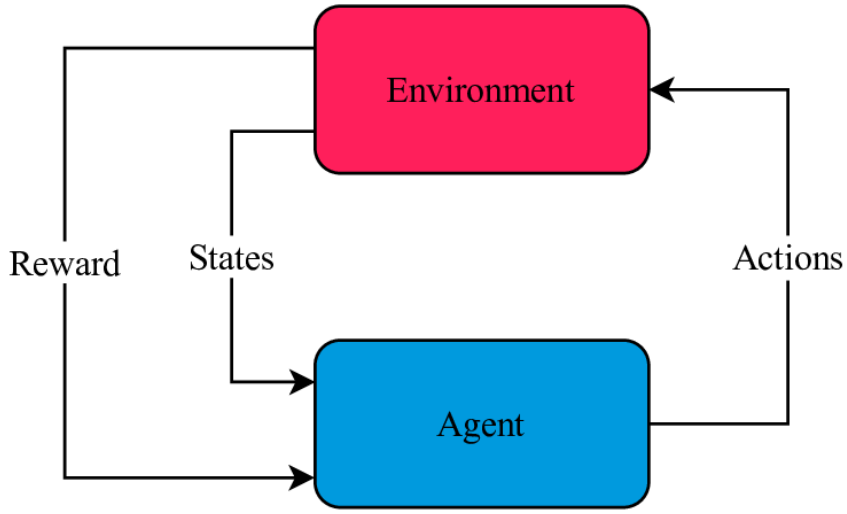

Figura 1.2. Interazione agente-ambiente nell'ambito dell'apprendimento per rinforzo.

La descrizione dell'interazione tra agente e ambiente ci permette di comprendere due cose; da un lato tale processo iterativo è a tempo discreto, dall'altro è necessario in mancanza di conoscenza a priori dell'ambiente, ovvero in mancanza di un modello. In altre parole, l'apprendimento per rinforzo può essere considerato un paradigma di controllo model-free a tempo discreto.

A tal proposito, il feedback rappresenta l'essenza del processo iterativo, in quanto garantisce all'agente consapevolezza del suo operato. In generale, esso consente di valutare se quel determinato obiettivo sia stato o meno raggiunto; ad esempio, per noi esseri umani "camminare" non è azione particolarmente complessa, anzi performiamo molto bene perché in età piuttosto giovane impariamo senza neanche rendercene conto...essendo questo un comportamento naturale per il nostro organismo. Se però

dovessimo spiegare a qualcuno che non sa camminare "come" si cammina, saremmo in difficoltà perché si tratta di un automatismo complesso da descrivere. Da questo punto di vista, il feedback, detto *reward*, guida l'agente verso la scelta di azioni considerate utili ai fini del raggiungimento del suo obiettivo.

Tali considerazioni affondano le proprie radici nel campo della psicologia e delle neuroscienze; gli organismi biologici sfruttano un meccanismo simile per imparare nei quali il premio, invece che composto da bit, è rappresentato da un segnale di gratificazione dovuto ad un rilascio di neurotrasmettitori da parte del cervello. Autori della psicologia comportamentale sostengono infatti che le risposte che in una particolare situazione producono un effetto soddisfacente, acquistano maggiori probabilità di essere prodotte di nuovo in futuro nella stessa circostanza [4]. L'apprendimento di un comportamento può dunque essere condizionato dalla presenza di uno stimolo esterno o di una ricompensa. Questo venne dimostrato tramite gli esperimenti sul condizionamento operante prima di [5] e poi di [6]; fu proprio quest'ultimo a proporre di utilizzare il termine "rinforzo". Un rinforzo positivo può dunque essere utilizzato per insegnare una specifica azione: se associato ad un comportamento determinerà una maggiore e più frequente produzione dello stesso. I nostri agenti informatici infatti a loro volta sfruttano un processo simile di apprendimento, nel quale il premio, invece che da un segnale biologico è determinato da un feedback sotto forma di una sequenza di zero e uno.

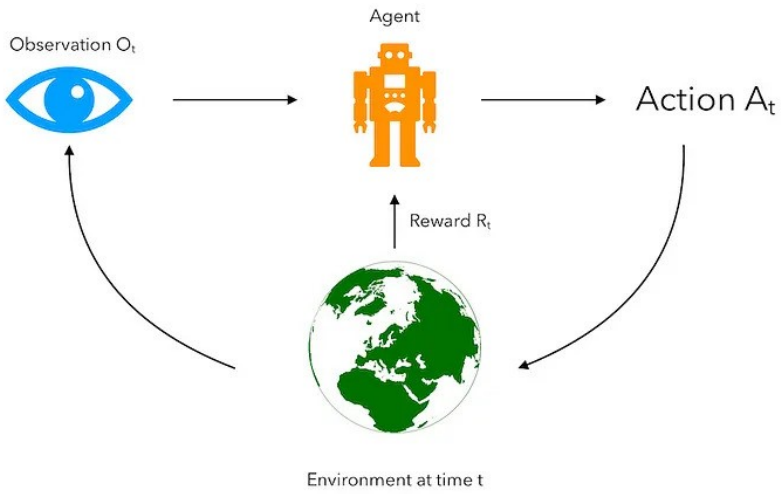

Figura 1.3. Feedback intrinseco nel RL.

L'apprendimento per rinforzo si distingue da quello supervisionato in quanto non si basa su dati etichettati a priori, bensì su una comprensione autonoma da parte dell'agente basata sulla massimizzazione di un segnale di ricompensa attraverso

l'interazione con l'ambiente. L'agente impara a mappare situazioni ad azioni per aumentare il premio guadagnato, questo processo è intrinsecamente legato a cicli di feedback, quindi ad anello chiuso, nel quale le azioni eseguite dall'agente influenzano le sue decisioni future, come mostrato in Figura 1.3.

L'agente è definibile come un'entità che interagisce con l'ambiente in cui si trova per raggiungere un obiettivo, il quale prende decisioni sulla base di una strategia di controllo che apprende interagendo con l'ambiente; tale interazione avviene sulla base di sensori e di attuatori - così come un essere umano possiede occhi, orecchie e altri organi di senso che fungono da sensori, e mani, gambe e così via come attuatori, così un agente robotico potrebbe avere telecamere e misuratori a infrarossi come sensori, e vari motori come attuatori.

L'ambiente può essere definito come tutto ciò con cui l'agente interagisce; in pratica, corrisponde alla porzione di universo di cui ci interessano gli stati. In altre parole, corrisponde alla parte di spazio che influisce su ciò che l'agente percepisce e sul quale a loro volta incidono le azioni di quest'ultimo. Mentre per percezione si intende il contenuto che i sensori dell'agente stanno rilevando e immagazzinando, per sequenza di percezione la storia completa di tutto ciò che ha mai percepito.

In generale, ciò che l'agente esperisce prende il nome di *stato* dell'ambiente s_t, inteso come una descrizione completa e pregnante dell'ambiente. Talvolta potrebbe accadere che l'agente non sia in grado di esperire direttamente lo stato, bensì una porzione di esso, che prende il nome di *osservazione* o_t. Tipicamente, stato e osservazione coincidono.

L'obiettivo di un agente consiste nell'apprendimento di una strategia di controllo $\pi(s_t, a_t)$, intesa come una legge che sia in grado di associare un'azione a_t a un dato stato, che può essere di due tipi:

- Probabilistica $a_t \sim \pi(s_t, a_t)$, andando ad associare una distribuzione di probabilità alle azioni possibili in un certo stato;
- Deterministica $a_t = \pi(s_t)$, andando ad associare direttamente una azione a uno stato.

La modalità di apprendimento della politica si basa sulla massimizzazione della reward cumulativa, corrispondente alla somma del segnale fornito dall'ambiente all'agente al termine di ciascuna interazione, detta *ritorno* R_t; indicando con r_t il reward ottenuto al tempo t, si ottiene:

$$R_t = r_t + r_{t+1} + r_{t+2} + \dots. \tag{1.1}$$

Dato che tale formula consiste in una somma infinita di termini, i quali si riferiscono a interazioni successive, al fine di differenziare il contributo dei termini

relativi al presente e al futuro, si introduce il *fattore di sconto* $\gamma \in [0, 1]$ nel ritorno, ottenendo il ritorno scontato:

$$R_t = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \ldots = \sum_{k=0}^{\infty} \gamma^k r_{t+k}. \tag{1.2}$$

Qualora $\gamma < 1$, i reward futuri avranno un peso minore rispetto a quello presente, pertanto saranno scoraggiati comportamenti volti a massimizzare guadagni futuri. Se $\gamma = 0$, l'agente sarà interessato a massimizzare solo le ricompense immediate, mentre per $\gamma = 1$ l'agente tenderà a fornire la stessa importanza a tutte le ricompense.

Generalmente l'allenamento di un agente di apprendimento per rinforzo risulta scandito in *episodi*, ciascuno avente un massimo di interazioni con l'ambiente corrispondente a $T \in \mathbb{R}^+$, secondo un approccio di tipo esplorazione, dall'inglese *exploration*, e sfruttamento, *exploitation*.

L'esplorazione consiste nel comportamento da parte dell'agente volto sostanzialmente a scrutare l'ambiente visitando più stati possibili e acquisendo conoscenza del mondo circostante. Durante questa fase, le azioni prese sono volte alla raccolta di informazioni; qualora scegliesse fin dall'inizio l'azione che crede ottimale non convergerebbe sicuramente verso una politica ottimale, in quanto avrebbe troppe poche informazioni per decidere saggiamente.

Lo sfruttamento si riferisce alla strategia con cui un agente sceglie l'azione che massimizza la ricompensa attesa in base alle informazioni già acquisite. In altre parole, l'agente sfrutta la conoscenza derivante dalle visite già effettuate per ottenere la migliore ricompensa immediata possibile, piuttosto che provare nuove coppie stato-azione.

Dato che ogni azione, oltre a fornire una ricompensa, fornisce anche informazioni sul comportamento dell'ambiente, l'agente deve mettere in campo una strategia di compromesso tra lo sfruttamento e l'esplorazione, sacrificando le ricompense immediate per massimizzare quelle da ottenere in iterazioni future; tale approccio prende il nome di ϵ-*greedy*, e verrà analizzato in seguito. In altre parole, il comportamento dell'agente si può riassumere nel modo seguente: "Nel mondo reale dobbiamo costantemente decidere se continuare nella nostra esistenza confortevole o lanciarci nell'ignoto nella speranza di una vita migliore" [7].

La formalizzazione dei problemi di apprendimento per rinforzo avviene tipicamente per mezzo dei processi decisionali dei Markov, dall'inglese Markov Decision Process (MDP), basati sull'assunzione fondamentale che il processo descritto non abbia memoria, nel senso che la transizione da uno stato futuro dipende unicamente da quello attuale e non dagli stati precedenti; in altre parole, il futuro del processo è compiutamente descritto dal presente e non dal passato.

Un MDP è descrivibile per mezzo di una tupla di elementi $[S, A, T, R, \gamma, T]$ corrispondenti a:

- S corrisponde allo spazio di stato, quindi rappresenta l'insieme di tutti i possibili stati;
- A corrisponde allo spazio delle azioni, quindi rappresenta l'insieme di tutte le azioni ammissibili;
- $T(s_t, a_t, s_{t+1}) : S \times A \times S \to [0, 1]$ la probabilità di transizione, intensa come la probabilità che l'ambiente evolva dallo stato s_t a s_{t+1} sotto l'azione a_t;
- $R(s_t, a_t, s_{t+1}) : S \times A \times S \to \mathbb{R}$ la funzione di ricompensa, che restituisce il valore della ricompensa dopo l'applicazione di a_t;
- γ, il fattore di sconto [8];
- T la lunghezza di ciascun episodio.

In generale, lo spazio di stato e quello delle azioni possono essere infiniti o finiti. Di nostro interesse il secondo caso, rispetto al quale si parla di MDP finito [9]. Data la stocasticità intrinseca formalizzata per mezzo della probabilità di transizione, è possibile ricavare delle espressioni per la probabilità di transizione e per la ricompensa attesa. Detti s un generico stato, a una generica azione, s' lo stato cui si giunge a partire dal primo per mezzo della seconda, e r una generica ricompensa, la probabilità di transizione:

$$p(s', r|s, a) = \Pr\{s_{t+1} = s', r_{t+1} = r \mid s_t = s, a_t = a\}. \tag{1.3}$$

Ne consegue che la ricompensa attesa per una coppia stato-azione assume la seguente espressione:

$$r(s, a) = E[r_{t+1} \mid s_t = s, a_t = a] = \sum_r r \sum_{s' \in S} p(s', r|s, a). \tag{1.4}$$

Ne consegue che la funzione probabilità di transizione assume la seguente forma:

$$T(s'|s, a) = \Pr\{S_{t+1} = s' \mid S_t = s, A_t = a\} = \sum_r p(s', r|s, a). \tag{1.5}$$

Infine, la ricompensa attesa per una tripla stato-azione-stato successivo è:

$$r(s, a, s') = E[R_{t+1} \mid S_t = s, A_t = a, S_{t+1} = s'] = \frac{\sum_r rp(s', r|s, a)}{p(s'|s, a)}. \tag{1.6}$$

Molti algoritmi di RL assumono che la proprietà di Markov sia soddisfatta nonostante questa richieda la completa osservabilità degli stati, quindi l'agente può accedere a qualsiasi informazione su di essi. Sappiamo bene ormai che non è spesso possibile che ciò accada, infatti si tratta di una approssimazione che si fa. Nella realtà molte applicazioni degli algoritmi sono calate in contesti detti MDP parzialmente osservabili (POMDP). Anche il progetto presentato nei capitoli successivi farà parte di questa categoria di processi.

1.3 Tassonomia del Reinforcement Learning

Prima di addentrarci nelle tipologie degli algoritmi di RL bisogna distinguerne alcune famiglie: gli algoritmi *model-based*, che si basano su un modello fornito a priori all'agente da quelli *model-free*, nei quali l'agente non ha alcuna precedente conoscenza delle dinamiche dell'ambiente né ha bisogno di costruirsi un modello per apprendere una politica ottimale.

Quando si possiede un modello ci si riferisce in particolar modo a due funzioni note, la funzione di probabilità di transizione T e la funzione di ricompensa R che sono sufficienti per descrivere le dinamiche dell'ambiente e quindi bastanti all'agente per prevedere cosa potrebbe accadere in seguito ad una sua azioni. La sintesi dei risultati ottenuti, immagazzinati dall'agente, rappresenta la politica appresa.

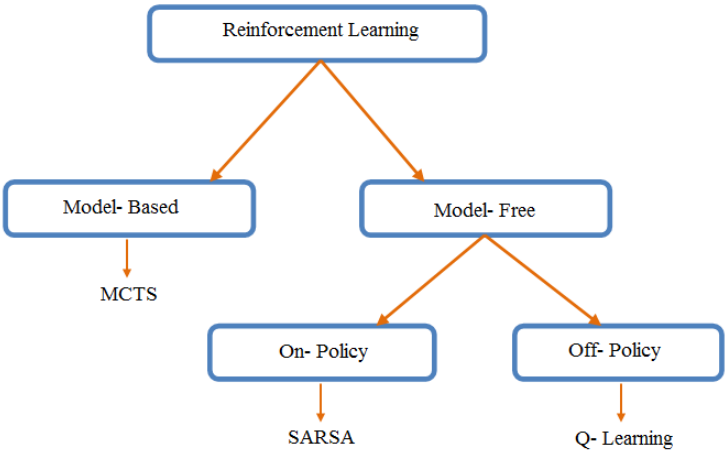

Figura 1.4. Tassonomia algoritmi di RL.

Nella realtà un modello dell'ambiente non è quasi mai disponibile, pertanto l'agente dovrà ricorrere alla sua esperienza diretta per costruirne uno. Tutto questo richiede una notevole potenza di calcolo e di risorse, specialmente all'aumentare delle dimensioni dello spazio di stato.

Per questo motivo gli algoritmi model-free, rappresentano una valida alternativa. Questi sono meno onerosi da implementare, sono molto popolari nella letteratura scientifica in quanto ampiamente sviluppati e testati.

Nel *model-based* RL, l'agente utilizza un modello della dinamica dell'ambiente per interpretare il segnale di ricompensa e prendere decisioni informate su come agire. Questo modello può essere inizialmente sconosciuto e appreso progressivamente osservando gli effetti delle proprie azioni, oppure può essere noto a priori. Ad esempio, nel gioco degli scacchi, l'agente può conoscere le regole del gioco fin dall'inizio, pur non sapendo ancora quali mosse siano ottimali, pertanto imparerà come agire facendo esperienza in base alle condizioni della scacchiera.

Un esempio di applicazione del Reinforcement Learning basato sul modello lo troviamo nella robotica industriale, in particolare in certi utilizzi dei robot manipolatori si vuole trovare un modo per aumentare l'efficienza temporale dell'attività di pick and place (prendi e posiziona). É possibile far lanciare gli oggetti al robot, invece di riporli. In questo caso si può parlare di pick and throw (prendi e lancia). Ciò consente di allargare il confine dello spazio raggiungibile del robot, sfruttando la sua cosiddetta manualità estrinseca, il progetto di tesi [10] ne descrive tutti i dettagli.

Nel *model-free* RL invece, l'agente non dispone né apprende un modello della dinamica dell'ambiente. Invece di prevedere le transizioni di stato, impara direttamente una strategia ottimale basandosi sull'esperienza accumulata. Questo avviene attraverso il miglioramento delle stime delle funzioni di valore o politiche di azione, aggiornate iterativamente in base alle ricompense ricevute. Vi sono due *modi operandi* alla base di questo paradigma:

- **Metodi Value-based**: basati sulle funzioni di valore.
- **Metodo Policy-based**: basati sulla ricerca della politica.
- **Actor-Critic**: basati sulla combinazione dei precedenti.

I metodi *value-based* comportano la valutazione di una funzione di valore o funzione degli stati (o ancora coppia stato-azione) che stima quanto è buono per l'agente essere in un dato stato (o performare un' azione in un dato stato). La nozione di "buono" è definita in termini di ricompensa futura che può essere ottenuta. Naturalmente quest'ultima dipende dalle azioni performate dall'agente, pertanto ogni funzione di valore è definita rispetto ad una particolare politica.

La politica, come sopra introdotto (π), è una mappatura da ogni stato $s \in S$, e azione $a \in A(s)$ verso la probabilità $\pi(a|s)$ di prendere un'azione a quando ci si trova in uno stato s. Il valore dello stato s sotto una politica π è denotata da $v_\pi(s)$, ed è

il valore atteso quando si parte in s e si segue π da quello stato in poi. Formalmente definiamo $v_\pi(s)$ nel seguente modo:

$$V^\pi(s) = E_\pi \left[G_t \mid S_t = s \right] = E_\pi \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k} \mid S_t = s \right]. \tag{1.7}$$

Dove $E_\pi(\cdot)$ denota il valore atteso di una variabile casuale dato un agente che segue la politica π, e t indica un piccolo lasso di tempo (time step). Notiamo che il valore dello stato terminale è sempre zero perchè non c'è più valore da accumulare $v(s_{terminale}) = 0$. La funzione v_π è detta funzione stato-valore per la policy π.

In maniera analoga definiamo il valore ottenuto in seguito all'aver preso una azione a in uno stato s seguendo π, denotato stavolta con $q_\pi(s, a)$, il rendimento atteso a partire da s per aver performato a seguendo questa politica è:

$$Q_\pi(s, a) = E_\pi \left[G_t \mid S_t = s, A_t = a \right] = E_\pi \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k} \mid S_t = s, A_t = a \right]. \tag{1.8}$$

Chiamiamo q_π la funzione azione-valore per la politica π. Le funzioni valore v_π e q_π possono essere stimate dall'esperienza. Ad esempio se un agente segue una politica π e mantiene una media, per ogni stato visitato, dei rendimenti effettivi che quello stato ha portato, allora questa media convergerà al valore dello stato $v_\pi(s)$. Se vengono mantenute medie separate per ogni azione intrapresa in uno stato, allora queste convergeranno in maniera simile ai valori dell'azione $q_\pi(s, a)$.

Quindi una volta che tutte le possibile coppie stato-azione con i valori ottimali sono state mappate, la soluzione del problema di RL è rappresentata dalla politica ottima (π^*) che viene trovata scegliendo per ciascuno stato l'azione a avidamente (greedily), formalmente abbiamo:

$$\pi^*(s) = \arg\max_a q^*(s, a). \tag{1.9}$$

La relazione che lega le due funzioni di valore nella politica ottimale è:

$$v^*(s) = \max_{a \in A(s)} q^*(s, a). \tag{1.10}$$

Una proprietà fondamentale delle funzioni di valore utilizzate nel RL e nella programmazione dinamica (dynamic programming) è che soddisfano una particolare relazione ricorsiva. Per ogni politica π e ogni stato s, vale la seguente condizione di

coerenza tra il valore di s e quello dei suoi possibili stati successori:

$$\begin{aligned} V^{\pi}(s) &= E^{\pi}[G_t \mid S_t = s] \\ &= E^{\pi}\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k} \mid S_t = s\right] \\ &= E^{\pi}\left[R_t + \gamma \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \mid S_t = s\right] \\ &= \sum_a \pi(a|s) \sum_{s'} \sum_r p(s', r|s, a) \left[r + \gamma E^{\pi}\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \mid S_{t+1} = s'\right]\right] \\ &= \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a) \left[r + \gamma V^{\pi}(s')\right]. \end{aligned} \tag{1.11}$$

Nella quale implicitamente intendiamo che l'azione a è stata presa dall'insieme $A(s)$, lo stato successivo s' è preso dall insieme S e la ricompensa r dall'insieme R. L'espressione finale può essere letta come un valore atteso, si tratta di una sommatoria rispetto tutte e tre le variabili a, s' e r. Per ogni tripla calcoliamo la sua probabilità $\pi(a|s)p(s', r|s, a)$, pesiamo la quantità tra parentesi quadre per quella probabilità e sommiamo tutte le combinazioni per ottenere un valore atteso.

L'espressione sopra mostrata è l'*equazione di Bellman per* v_π, ed esprime una relazione tra il valore di uno stato e il gli stati successori. Per visualizzare meglio come nella programmazione dinamica vengano apprese le funzioni di valore sfruttiamo una versione delle equazioni di Bellman in forma ricorsiva:

$$V^{\pi}(s_t) = E_{s_{t+1}}\left[r_{t+1} + \gamma V^{\pi}(s_{t+1})\right]. \tag{1.12}$$

$$Q^{\pi}(s_t, a_t) = E_{s_{t+1}}\left[r_{t+1} + \gamma Q^{\pi}(s_{t+1}, \pi(s_{t+1}))\right]. \tag{1.13}$$

Nelle quali si considera la somma del premio immediato che si ottiene grazie ad una certa transizione e il valore scontato (moltiplicato per γ) dello stato successivo a quello che stiamo considerando.

L'idea alla base delle equazioni di Bellman è quella di formulare il problema della massimizzazione del *ritorno scontato atteso* in termini di relazioni ricorsive, nelle cosiddette *Equazioni di Ottimalità di Bellman* per $v^*(s)$ e $q^*(s, a)$. Tali equazioni però rimangono possibili sono in linea teorica poiché si richiede da parte dell'agente l'accesso alla conoscenza delle dinamiche di transizione e premio che nel contesto model-free che stiamo analizzando non sono contemplate.

I metodi *policy-based* sono una classe di algoritmi che hanno il compito di individuare una politica ottimale tramite una ricerca diretta senza passare per le funzioni di valore. L'idea è quella di parametrizzare la politica ottenendo π_θ, questa fornirà come output una distribuzione di probabilità sulle azioni (politica stocastica):

$$\pi_\theta(s) = \mathbb{P}[A|s, \theta]. \tag{1.14}$$

In pratica questa politica, dato uno stato, ritorna una distruzione di probabilità sulle

azioni di quello stato. Il nostro obiettivo è quello di massimizzare la performance della policy parametrizzata utilizzando i metodi della salita del gradiente (gradient ascent). Analizzeremo proprio questa tipologia di metodi.

I metodi policy-gradient sono una sottoclasse dei metodi basati sulla politica che permettono di stimare i parametri della politica ottimale utilizzando la sopracitata salita del gradiente. Rispetto ai metodi basati sulle funzioni di valore sono caratterizzati da proprietà che garantiscono una migliore convergenza. Risultano anche più efficienti quando lo spazio delle azioni ha una dimensione elevata, o nel caso in cui le azioni non siano discretizzabili e quindi possano solo essere considerate come continue; quest'ultima circostanza rende inapplicabili le value-function. Il nostro obiettivo con la π_θ è quello di trovare i parametri migliori che massimizzino una funzione obiettivo $J(\theta)$ della politica, la quale fornisce una valutazione delle performance.

$$J(\theta) = E\left[\sum \gamma r \mid \pi_\theta\right]. \tag{1.15}$$

Se riusciamo ad individuare i migliori parametri stiamo sostanzialmente trovando la politica ottimale, nonché soluzione del problema di RL. Possiamo considerare questo processo come un problema di ottimizzazione, che cerchiamo di risolvere tramite due passi ripetuti iterativamente. Il primo consiste nel definire una funzione obiettivo appropriata e valutare la qualità della politica con i parametri attuali, il secondo nell'aggiornare i parametri in base alla salita del gradiente della politica, quindi nella direzione di maggior incremento.

Per la scelta della funzione obiettivo nel primo passo, abbiamo diverse formulazioni a seconda della tipologia di ambiente. Nel caso discreto, quindi in cui abbiamo un ambiente episodico, consideriamo di avere delle traiettorie finite che partono dallo stesso stato iniziale, il che ci permette di misurare la qualità della politica in termini di valore atteso delle ricompense scontate cumulative raccolte durante l'episodio. Se siamo in un ambiente continuo, non potendo definire uno stato iniziale dobbiamo considerare una media pesata dei ritorni scontati ottenuti a partire da diversi stati, dove i pesi sono rappresentati proprio dalle probabilità che questi occorrano. Potemmo quindi voler ottimizzare , in media, la ricompensa ottenuta a seguito di ciascuna interazione con l'ambiente. In tal caso individuiamo la funzione obiettivo con la media pesata delle ricompense immediate che si ricevono dall'ambiente. In un ambiente episodico, ad esempio, considerando traiettorie finite che partono sempre dallo stesso stato iniziale, la qualità della politica può essere misurata in termini di valore atteso delle ricompense scontate raccolte durante l'episodio, a partire da tale stato. In un ambiente continuo, invece, senza uno stato iniziale fisso, si valuta generalmente una media ponderata dei ritorni scontati ottenuti da stati diversi, dove i pesi corrispondono alle probabilità di occorrenza di ciascuno stato. Se l'obiettivo

è ottimizzare la ricompensa media ottenuta da ogni interazione con l'ambiente, la funzione obiettivo si basa sulla media (anch'essa ponderata) delle ricompense immediate che è possibile ricevere.

Passando al passo successivo vogliamo incrementare le performance della politica, apportando cambiamenti ai parametri. Mentre queste modifiche influenzano in modo immediato le azioni, determinare come influenzano la distribuzione degli stati richiede una comprensione più profonda delle dinamiche dell'ambiente e delle probabilità di transizione. Per semplificare il problema sfruttiamo il *Teorema del Gradiente della Politica* che fornisce un metodo per calcolare il gradiente della funzione obiettivo in un algoritmo di apprendimento basato sulle politiche. Il gradiente di $J(\theta)$ rispetto ai parametri θ della politica è dato dalla seguente espressione:

$$\nabla_\theta J(\theta) = E_{\pi_\theta} \left[\nabla_\theta \log \pi_\theta(a|s) \cdot G_t \right] \tag{1.16}$$

dove

- $\pi_\theta(a|s)$ è la politica parametrizzata che sancisce la probabilità di scegliere l'azione a dato lo stato s;
- G_t è il ritorno cumulativo scontato, che dipende dalla ricompensa futura ottenuta a partire dallo stato s_t;
- $\nabla_\theta \log \pi_\theta(a|s)$ è il gradiente del logaritmo della probabilità di selezionare l'azione a in uno stato s rispetto ai parametri θ.

Tramite il gradiente possiamo dedurre la direzione di maggiore incremento della probabilità logaritmica di ottenere una certa traiettoria con π_θ e la funzione obiettivo sotto forma di ritorno calcolato sull'intera durata dell'episodio, G_t. Essendo un valore atteso, quello che otteniamo è una stima di tale gradiente ottenuta dai dati forniti da campioni empirici di episodi differenti, pertanto possono sorgere problemi legati ad una varianza piuttosto elevata. Una volta calcolata la stima del gradiente i parametri vengono aggiornati nel seguente modo:

$$\theta \leftarrow \theta + \alpha \nabla_\theta \hat{J}(\theta) \tag{1.17}$$

In cui oltre ai parametri individuati da θ, abbiamo α che indica il tasso di apprendimento, che controlla la velocità con cui vengono aggiornati i parametri e $\nabla_\theta \hat{J}(\theta)$ che è la stima del gradiente. In questo modo ad ogni iterazione del metodo vengono aggiornati i parametri nella direzione che aumenta la funzione obiettivo $J(\theta)$ cercando di migliorare la politica e ottenere una ricompensa maggiore.

I metodi *actor-critic* combinano le funzioni di valore con una rappresentazione esplicita della politica, dando luogo ai metodi attore-critico. L'attore, è rappresentato

da una politica $\pi_\theta(s, a)$ caratterizzata dai parametri θ simile a quella vista poc'anzi nei metodi policy-based. Esso apprende attraverso il feedback fornito dal critico, il quale è costituito dalla stima di una funzione valore (a volte la state-value altre la action-value). I due hanno i seguenti ruoli: l'attore prende le decisioni selezionandole in base alla politica corrente, la sua responsabilità riguarda l'esplorazione dello spazio delle azioni per massimizzare la ricompensa cumulativa ridefinendo iterativamente la politica, adattandola alla natura dinamica dell'ambiente; il critico giudica le azioni prese dall'attore fornendo il feedback sulle performance, il suo ruolo è fondamentale nel guidare l'attore attraverso le azioni che portano ad alti ritorni, contribuendo al miglioramento del processo di apprendimento.

L'algoritmo funziona nel seguente modo: si definisce una funzione obiettivo per l'algoritmo attore-critico risultante da una combinazione dei due metodi, policy gradient (Actor) e value function (Critic). La funzione obiettivo complessiva è espressa solitamente dalla somma di due componenti. Il primo è il gradiente della politica dell'attore:

$$\nabla_\theta J(\theta) \approx \frac{1}{N} \sum_{i=0}^{N} \nabla_\theta \log \pi_\theta(a_i \mid s_i) \cdot A(s_i, a_i). \tag{1.18}$$

dove $J(\theta)$ rappresenta il ritorno atteso sotto la politica parametrizzata da θ; $\pi_\theta(a \mid s)$ è la politica; N il numero di episodi; $A(s, a)$ è la funzione vantaggio che rappresenta il beneficio di prendere l'azione a nello stato s; i è l'indice dell'episodio. La seconda componente è data dall'aggiornamento della funzione valore:

$$\nabla_w J(w) \approx \frac{1}{N} \sum_{i=1}^{N} \nabla_w \left(V_w(s_i) - Q_w(s_i, a_i) \right)^2 . \tag{1.19}$$

in cui $\nabla_w J(w)$ è il gradiente della funzione perdita rispetto ai parametri w del critico; N gli episodi; $V_w(s)$ è il valore stimato dello stato s dal critico rispetto w; e $Q_w(s, a)$ è la stima del valore ottenibile prendendo l'azione a nello stato s; i è sempre l'indice. Il fulcro della questione avviene ora con le regole di aggiornamento per l'attore e il critico, le quali involvono la correzione dei rispettivi parametri usano la salita del gradiente (per l'attore) e la discesa del gradiente (per il critico). Si opera nel seguente modo:

$$\theta_{t+1} = \theta_t + \alpha \nabla_\theta J(\theta_t). \tag{1.20}$$

per aggiornare i parametri dell'attore, compaiono sia il tasso di apprendimento α che il time step t; poi si passa ai parametri del critico:

$$w_t = w_t - \beta \nabla_w J(w_t). \tag{1.21}$$

dove β è il tasso di apprendimento del critico. Ci calcoliamo la funzione vantaggio $A(s, a)$ che misura il beneficio di prendere un azione a nello stato s rispetto al valore atteso dello stesso stato sotto la politica corrente.

$$A(s, a) = Q(s, a) - V(s). \tag{1.22}$$

In questo modo ci viene fornita una misura di quanto in meglio o in peggio un azione vale rispetto al valore medio delle azioni. Con questa espressione matematica mettiamo in evidenza il calcolo essenziale nel metodo Actor-Critic. L'attore è incoraggiato a prendere azioni con alti benefici, tramite gradiente della politica, il critico deve minimizzare la differenza tra il valore stimato e l'action-value.

Abbiamo constatato che il nostro progetto rientra nel secondo caso trattato, quello dell'apprendimento con rinforzo senza modello (model-free). In questa sezione

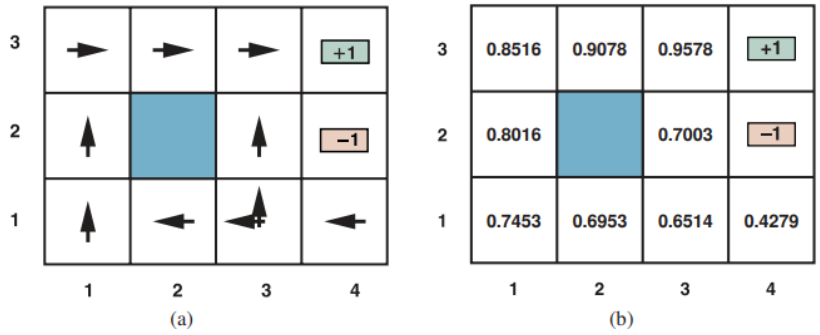

Figura 1.5. (a) Esempio di policy ottima per l'ambiente stocastico. Nello stato (3,1) ci sono due policy perché sia sopra (up) che sinistra (left) sono considerate azioni ottime. (b) Le utility degli stati in mondo 4x3, data la policy π.

approfondiamo il discorso RL con alcuni concetti importanti. Come mostrato in Figura 1.5, a sinistra sono rappresentate due politiche sullo stesso ambiente grid-world, le frecce indicano quali azioni di movimento hanno permesso l'ottenimento del punteggio maggiore. Sulla destra, in alcune celle, sono presenti dei valori (utilità), che mostrano quanto è desiderabile un certo stato, in questo esempio più il valore si avvicina ad $+1$ e meglio è per l'agente trovarsi in quella posizione. La cella blu rappresenta un ostacolo che il robot deve evitare. Infine in corrispondenza delle celle con $+1$ e -1 abbiamo due stati terminali. Dai concetti appena esposti distinguiamo due modalità di apprendimento:

- RL passivo
- RL attivo

Nel *RL passivo* un agente possiede già una politica fissa $\pi(s)$ che determina le sue azioni. Quest'ultimo prova ad apprendere la Utility function $U^{\pi}(s)$, ossia la ricompensa totale attesa quando la policy π è eseguita a partire dallo stato s. Durante l'apprendimento passivo l'agente non conosce né il modello di transizione: $T(s'|s,a)$, il quale specifica la probabilità di raggiungere lo stato s' trovandosi in s dopo aver preso un azione a; né la Reward function $R(s,a,s')$, che specifica la ricompensa associata ad ogni transizione.

Riprendendo l'esempio mostrato in Figura 1.5 che mostra le due politiche ottime in quell'ambiente e le corrispondenti utilità fornite in ciascuno stato, notiamo come esse derivino da una serie di prove che l'agente ha eseguito nell'ambiente sfruttando la sua policy $\pi(s)$. In ogni episodio l'agente inizia nello stato $(1,1)$ e passa attraverso una serie di stati fino raggiungerne uno terminale, $(4,2)$ o $(4,3)$. Esempi di percorsi scelti in funzione di $\pi(s)$) sono:

$$(1,1) \xrightarrow[-0.04]{\text{Sopra}} (1,2) \xrightarrow[-0.04]{\text{Sopra}} (1,3) \xrightarrow[-0.04]{\text{Destra}} (1,2) \xrightarrow[-0.04]{\text{Sopra}} (1,3) \xrightarrow[-0.04]{\text{Destra}} (2,3) \xrightarrow[-0.04]{\text{Destra}} (3,3) \xrightarrow[+1]{\text{Destra}} (4,3)$$

$$(1,1) \xrightarrow[-0.04]{\text{Sopra}} (1,2) \xrightarrow[-0.04]{\text{Sopra}} (1,3) \xrightarrow[-0.04]{\text{Destra}} (2,3) \xrightarrow[-0.04]{\text{Destra}} (3,3) \xrightarrow[-0.04]{\text{Destra}} (3,2) \xrightarrow[-0.04]{\text{Sopra}} (3,3) \xrightarrow[+1]{\text{Destra}} (4,3)$$

$$(1,1) \xrightarrow[-0.04]{\text{Sopra}} (1,2) \xrightarrow[-0.04]{\text{Sopra}} (1,3) \xrightarrow[-0.04]{\text{Destra}} (2,3) \xrightarrow[-0.04]{\text{Destra}} (3,3) \xrightarrow[-0.04]{\text{Destra}} (3,2) \xrightarrow[-1]{\text{Sopra}} (4,2)$$

Ogni transizione è individuata sia con l'azione performata (sopra e destra), sia con il punteggio (positivo e negativo) ricevuto finendo nello stato successivo. L'obiettivo è quello di utilizzare l'informazione riguardante la ricompensa per stimare l'utilità attesa (expected utility) $U^{\pi}(s)$ associata ad ogni stato non terminale s. L'utilità è definita come la somma attesa del premio scontato ottenuto se la policy π è seguita. L'equazione che descrive questa funzione è :

$$U^{\pi}(s) = E\left[\sum_{t=0}^{\infty} \gamma^t R(S_t, \pi(S_t), S_{t+1})\right] \tag{1.23}$$

In cui $R(S_t, \pi(S_t), S_{t+1})$ è la ricompensa ottenuta quando l'azione $\pi(S_t)$ è presa nello stato S_t e fa raggiungere all'agente lo stato S_{t+1}. Notiamo che S_t è una variabile che denota uno stato qualsiasi del mondo raggiunto al tempo t eseguendo la policy π, a cominciare da $S_0 = s$. Il significato del fattore di sconto (discount factor) γ verrà rimarcato successivamente, in questa sede ha valore $\gamma = 1$ pertanto non vi è alcuno sconto [7].

Nel *RL attivo*, un agente che sfrutta l'apprendimento con rinforzo attivo, invece, deve scegliere quale azione prendere. Stavolta sfrutteremo tramite il nostro agente la *programmazione dinamica adattativa*, in inglese adaptive dynamic programming (ADP), un metodo per risolvere problemi di decisione sequenziale, aggiornando iterativamente una funzione di valore o una politica sulla base dell'esperienza rac-

colta. Facciamo leva sulla "libertà" decisionale di cui dotiamo l'agente. Nel corso dell'allenamento apprenderà un modello di transizione completo con probabilità di esito per tutte le azioni, anziché il modello della politica fissa utilizzato nel passive RL. Dobbiamo tenere conto che l'agente può scegliere una delle azioni disponibili. Le utilità che deve calcolare e associare alle varie coppie stato-azione, sono quelle definite dalla policy ottima; queste devono rispettare l' *equazione di Bellman per l'ottimo*:

$$U(s) = \max_{a \in A(s)} \sum_{s'} P(s'|s,a) \left[R(s,a,s') + \gamma U(s') \right] \tag{1.24}$$

Questa equazione può essere risolta per ottenere la utility function U utilizzando gli algoritmi di iterazione del valore o iterazione della politica; tuttavia la sua risoluzione non è di nostro interesse ma può essere approfondita in [11].

Ci resta da capire cosa fa l'agente effettivamente durante ogni passo. Dopo aver ottenuto la funzione di utilità U, che è ottima per il modello appreso, l'agente può estrarre un'azione ottimale facendo una previsione di un passo (look-ahead) per massimizzare l'utilità attesa. In pratica sta scegliendo autonomamente l'azione che porta alla ricompensa più alta. Il processo decisionale, però, non è così semplice, si compone di due meccanismi citati in precedenza: l'*esplorazione* e lo *sfruttamento*.

Abbiamo visto come la categoria degli algoritmi model-free si classifica in base al tipo di approccio utilizzato per ricavare la soluzione ad un problema di RL. Esistono metodi basati sulle funzioni di valore (Value-based method) e metodi basati sulla ricerca della politica (Policy-based method). Abbiamo poi visto come nella letteratura sono presenti dei metodi ibridi che combinano i vantaggi dei due metodi e che prendono il nome di metodi attore-critico (Actor-Critic method). Ci concentreremo in maniera più approfondita sul metodo utilizzato per il progetto ossia quello basato sul valore.

Capitolo 2

Reinforcement Learning Multi Agente

2.1 Scenario Applicativo

Nel resto dell'elaborato viene proposta un applicazione di un algoritmo di RL: il *Q-Learning*. Questo sarà sfruttato da due agenti all'interno di una mappa bidimensionale denominata grid-world, ad ognuno sarà assegnato il proprio obiettivo. Ci troviamo in un contesto detto non-cooperativo, ossia nel quale i due agenti non cooperano tra di loro per conseguire un fine comune, bensì sono in competizione tra di loro.

Lo scopo sarà proprio quello di far apprendere a ciascuno dei due una strategia ottima per completare il proprio task, i ruoli saranno quelli di Attaccante e Difensore. Oltre all'agente avversario contro il quale si andrà a competere nel grid-world saranno presenti anche degli ostacoli, quest'ultimi come è naturale pensare debbono essere schivati onde evitare il danneggiamento del robot. Questa specifica richiede quindi di implementare anche l'evitamento degli ostacoli, in inglese obstacle avoidance. Abbiamo definito, per il momento in maniera astratta e concettuale, lo scopo principale che si vuole conseguire; dobbiamo sfruttare uno strumento per valutare in termini quantitativi l'efficacia e sopratutto la corretta applicazione dell'algoritmo di Q-Learning.

Ci aspettiamo che sia l'Attaccante che il Difensore, nel tempo, apprendano una strategia ottima e quindi che al termine dell'allenamento entrambi convergano verso un percorso desiderato. Questo dovrà minimizzare il tempo necessario per raggiungere l'obiettivo e permettere all'agente di collezionare il maggior numero di punti. Lo strumento utilizzato per valutare questa performance è una funzione denominata: Reward Function. Durante ogni episodio l'agente accumulerà dei punti, la somma totale potrà essere positiva o negativa, se la ricompensa dell'episodio i-esimo, $R_i > 0$ allora il comportamento è considerabile accettabile, al contrario se $R_i < 0$ potrebbe

esser stato commesso un errore. Andando a mostrare l'andamento complessivo della Reward Function vogliamo che tenda all'ottimo globale che un certo agente può ottenere. Per far si che ciò accada c'è bisogno di un certo numero di iterazioni che l'algoritmo dovrà effettuare, ogni simulazione potrà differire leggermente circa il numero di episodi necessari per avere una convergenza, così come non tutte le funzioni convergeranno nello stesso identico modo, elementi di aleatorietà nel comportamento del robot avversario porteranno a risultati simili ma non identici. Ogni allenamento però dovrà portare ad un percorso e una politica ottimali, indice del fatto che il task sarà stato portato a termine ogni volta e quindi il Q-Learning ha permesso un apprendimento efficace.

2.2 Formalizzazione Matematica

Un MDP multi agente (MA-MDP) è una generalizzazione del MDP che abbiamo descritto nel capitolo precedente. Come suggerisce il nome si tratta di un'estensione nella quale non consideriamo solamente un agente bensì un numero strettamente maggiore di agenti che popolano la mappa. Questo ambiente è chiamato sistema multi agente poiché prevede molteplici attori che interagiscono tra di loro e con esso. Gli agenti in contesti di questo tipo devono affrontare quelli che sono chiamati problemi di pianificazione multi agente; vedremo in maniera rigorosa la natura di questi problemi e una tecnica appropriata di risoluzione. La scelta del metodo risolutivo si effettua in funzione della relazione che lega i due agenti. Approfondiamo questo discorso. Un MA MDP è descrivibile per mezzo di una tupla di elementi $(S; A_1, ..., A_n; P; r_1, ..., r_N; \gamma_1, ..., \gamma_N)$ corrispondenti a:

- S l'insieme dei possibili stati del sistema che descrivono la configurazione degli agenti nell'ambiente, ogni stato s è a sua volta un vettore di stati individuali $s = (s_1, s_2, ..., s_N)$, dove con s_i rappresentiamo lo stato dell'i-esimo agente;

- A_i è l'insieme delle azioni che l'agente i-esimo può compiere in ogni stato. L'insieme complessivo delle azioni è rappresentato dal prodotto cartesiano $A = A_1 \times \cdots \times A_N$;

- $P(s'|s, a)$ è la funzione di transizione che descrive la probabilità di passare dallo stato s allo stato s' in seguito all'azione collettiva $a = (a_1, a_2, ..., a_N)$ presa da tutti gli agenti;

- $r_i(s, a)$ è la ricompensa fornita dall'ambiente all'i-esimo agente.

- γ_i è il fattore di sconto che valorizza le ricompense di ciascun agente.

Segue dalla risoluzione del problema multi agente che ognuno svilupperà una politica $\pi_i(s)$. Prima di passare all'algoritmo che risolve un MA MDP forniamo altre importanti nozioni. Ci sono due distinzioni da fare, quella in cui vi è un solo agente responsabile delle decisioni (one decision maker) e quella in cui questo numero è pari a quello degli agenti (multiple decision makers). Introduciamo brevemente e per completezza il primo caso riportato, nonostante è bene osservare che il progetto rientri nel secondo.

Nel caso in cui vi è un un agente all'interno di un ambiente che contiene molteplici attori, questo ha il compito di sviluppare dei piani d'azione anche per gli altri e comunicarglieli. Ci si basa su un assunzione detta dell'agente benevolo, per cui gli altri agenti eseguiranno semplicemente ciò che gli viene imposto da esso. Gli attori dovranno sincronizzare le loro azioni, sia che avvengano contemporaneamente, in maniera mutualmente esclusiva, o in maniera sequenziale. Esistono tre tipi di panificazione: multi effettore, quando un agente deve gestire più effettori e l'interazione tra di essi; multi corpo quando gli effettori sono fisicamente separati; decentralizzata quando abbiamo dei vincoli sulle informazioni fornite dai sensori, che impediscono una conoscenza globale dello stato di essi e dobbiamo fare riferimento alle informazioni locali.

Passiamo al caso dei molteplici agenti, nel nostro progetto due, in cui ognuno sceglie ed esegue il proprio piano. Li chiameremo controparti. Possiamo distinguere due ulteriori possibilità: quella in cui perseguono un obiettivo comune, che si traduce in un problema di coordinazione nel quale entrambi gli agenti devono assicurarsi di progredire verso la medesima direzione senza interferire con i piani dell'altro; e la seconda in cui ogni agente segue le sue preferenze personali sfruttando al meglio la propria conoscenza. Quando non sono possibili accordi tra gli agenti abbiamo un gioco non cooperativo. É bene notare che in generale in questo tipo di situazioni gli agenti potrebbero comunque decidere di cooperare autonomamente, nel caso in cui questa decisione rientri nel loro interesse.

Noi abbiamo vincolato la scelta dei punteggi in modo tale da impedire sempre la cooperazione e valutare la risoluzione del problema di ciascun agente in maniera individuale. Un piano è corretto quando porta l'agente a raggiungere il goal, sembra piuttosto semplice, tuttavia bisogna tenere in considerazione la concorrenza rappresentata dall'antagonista. Il protagonista dovrà tenere in considerazione che le azioni altrui potrebbero compromettere le risorse di cui si avvale per prendere la sua di azione, ad esempio una cella occupata preventivamente dall'avversario e resa inutilizzabile. La sfida principale da affrontare nel caso multi agente è la gestione delle azioni, proprio a causa delle influenze che queste hanno sull'ambiente e che si riversano nelle condizioni iniziali in cui al passo successivo si sceglieranno le azioni da eseguire. I

modelli per la gestione delle azioni concorrenti sono state un epicentro della ricerca nelle comunità di computer science per decenni, tuttavia nessun modello definitivo ha prevalso sugli altri. Nonostante ciò i seguenti tre approcci sono divenuti largamente utilizzati. Il primo consiste nel considerare l'*esecuzione alternata* delle azioni dei rispettivi piani. Supponiamo di avere gli agenti A e D, con i seguenti piani d'azione, $A : [a_1, a_2]$ e $D : [d_1, d_2]$. L'unico vincolo che rispetta il modello è quello di prendere le azioni nei due piani nell'ordine in cui appaiono. Assumendo che le azioni siano discrete esistono sei differenti modi in cui possono essere eseguite concorrentemente: $[a_1, a_2, d_1, d_2]; [d_1, d_2, a_1, a_2]; [a_1, d_1, a_2, d_2]; [d_1, a_1, d_2, a_2]; [a_1, d_1, d_2, a_2]; [d_1, a_1, a_2, d_2]$. Per preservare la correttezza del piano è evidente che debba essere altrettanto valida ognuna delle possibili permutazioni parziali elencate. Questo modello è ampiamente utilizzato nell'esecuzione multi thread nei processori, tuttavia non può modellare azioni che avvengono nello stesso momento. Inoltre all'aumentare del numero delle azioni nei piani le permutazioni crescono esponenzialmente rendendo computazionalmente molto oneroso da sfruttare.

Il secondo approccio è detto di *vera concorrenza*, in esso si assume solo un ordine parziale evidenziato dai pedici della azioni, ad esempio sappiamo che a_1 debba essere eseguita prima di a_2 cosi come d_1 prima di d_2, tuttavia non importa quale tra a_1 o d_1 venga eseguita prima. Possono avvenire in qualsiasi ordine, anche contemporaneamente. Risulta un modello più realistico, ma decisamente più complesso da implementare.

Infine il terzo approccio assume la *perfetta sincronizzazione*: c'è un orologio globale a cui tutti gli agenti hanno accesso e che scandisce il tempo in intervalli discreti. Tutte le azioni durano la stessa quantità di tempo e vengono gestite simultaneamente.

La **risoluzione** di un MA MDP richiede un approccio che prende in considerazione le interazioni tra gli agenti. Nelle condizioni da noi poste il modo migliore è sfruttare l'algoritmo di Q-Learning.
Analizziamo per prima cosa lo pseudocodice appena mostrato (1) seguendo i passi:

- Inizializziamo tutte le coppie stato-azione $Q(s, a)$ con un valore arbitrario (zero per esempio);
- Dopodiché inizia un ciclo sul numero degli episodi fissati a priori;
- L'episodio viene inizializzato al tempo $t = 0$ e si sceglie uno stato iniziale s_0 dalla distribuzione X;
- Si entra in altro ciclo che si ripete finché l'agente non giunge ad uno stato finale;

Algorithm 1 Q-learning (Watkins e Dayan, 1992)

Inizializza $Q(s, a)$ arbitrariamente
for episodio = 1 **to** max_episodi **do**
    Inizializza $s_t \in S$ seguendo $\chi \in (X)$, $t = 0$
    while s_t non è uno stato finale **do**
        Scegli un'azione a_t con strategia ε-greedy:
            con probabilità ε, scegli un'azione casuale
            altrimenti, scegli $a_t = \arg\max_{a \in A} Q(s_t, a)$
        Esegui a_t, osserva s_{t+1} e r_t
        Aggiorna $Q(s_t, a_t)$ con la formula:
            $Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha_t \left(r_t + \gamma \max_{a'} Q(s_{t+1}, a') - Q(s_t, a_t)\right)$
        Aggiorna stato e tempo: $s_t \leftarrow s_{t+1}$, $t \leftarrow t + 1$
    end while
end for

- Viene selezionata un'azione a_t scegliendo con probabilità ϵ un'azione casuale e con $1 - \epsilon$ quella con il massimo valore $Q(s, a)$;
- Si esegue l'azione scelta a_t e l'ambiente restituisce il nuovo stato s_{t+1} e la ricompensa r_t;
- Si aggiorna la funzione Q tramite l'equazione di aggiornamento, si passa allo stato s_{t+1} e al time-step successivo;
- Il ciclo si interrompe se si è giunti ad uno stato finale, altrimenti si continua con il ciclo interno.

Procediamo verso la traducibilità in codice della teoria fin'ora proposta. Per spiegare in dettaglio lo pseudocodice partiamo dalla Funzione Action-Value descritta in 1.13 fondamentale nel contesto del RL. Abbiamo visto che essa stima il valore totale delle ricompense che un agente può aspettarsi di accumulare partendo da uno stato s e prendendo un'azione a, il singolo elemento è denotato da $Q_i(s, a)$. La funzione corrisponde alla seguente equazione di aggiornamento:

$$Q_{k+1}(s, a) = r(s, a) + \gamma \sum_{s' \in S} T(s'|s, a) \max_{a' \in A} Q_k(s', a') \tag{2.1}$$

Tuttavia in questa forma, l'algoritmo richiede la conoscenza del Markov Decision Process a causa della presenza delle componenti dinamiche di transizione e ricompensa. Il modello MDP lo abbiamo approfondito precedentemente e abbiamo compreso che non è facilmente utilizzabile nella pratica. Ricapitolando, gli elementi che compongono l'equazione sono:

- Stati (S): L'insieme di tutte le possibili posizioni nella mappa che l'agente può occupare;

- Azioni (A): L'insieme delle azioni che l'agente può prendere in ogni stato;
- Ricompensa immediata $r(s, a)$: Il feedback che l'agente riceve per aver eseguito un'azione in uno stato;
- Funzione di transizione $T(s'|s, a)$: La probabilità di spostarsi dallo stato s allo stato s' una volta eseguita a;
- Fattore di sconto (γ): Indica quanto le ricompense future influenzano le decisioni attuali.

L'idea dietro questo algoritmo è di aggiornare il valore $Q_{i+1}(s, a)$ ad ogni iterazione in modo tale che il valore finale associato ad ogni coppia stato-azione sia pari al punteggio effettivo che l'agente ottiene nel lungo termine per aver preso determinate decisioni. In pratica per ogni cella della mappa che non è uno stato terminale convergerà un determinato valore, come precedentemente mostrato in 1.5. Maggiore è questo numero e più il robot è attratto dal percorrere quella strada.

Sappiamo che sia la funzione di transizione $T(s'|s, a)$, che le ricompense $r(s, a)$ sono sconosciute, pertanto l'agente non sa in anticipo quale sarà il risultato esatto delle sue azioni. Pertanto dobbiamo passare da una versione del Q-learning model-based ad una model-free implementabile al calcolatore. L'idea chiave del Q-learning riformulato è di sostituire la sommatoria su tutti gli stati $s' \in S$ in (2.1) con una solo sugli stati effettivamente visitati dall'agente. Quindi il robot non avrà bisogno di conoscenze a priori, ma nel corso degli episodi imparerà sbagliando, quelle che sono le scelte corrette da prendere per portare a termine con successo il compito.

Questa sostituzione rende il Q-learning un algoritmo model-free, poiché l'agente apprende i valori $Q(s, a)$ basandosi esclusivamente sulle esperienze raccolte durante l'interazione con l'ambiente, senza dover conoscere esplicitamente la distribuzione di probabilità che regola le transizioni $T(s'|s, a)$ né le ricompense $r(s, a)$.

Progettazione Q-Learning La politica utilizzata dal robot per raccogliere dati, che chiameremo π_e, è fondamentale per garantire il buon funzionamento del Q-Learning. Dopotutto, invece di utilizzare la funzione di transizione per determinare s', abbiamo sostituito queste informazioni con i dati effettivamente raccolti dal robot. Se la politica π_e non esplora le diverse parti dello spazio stato-azione, è facile immaginare che la nostra stima $\hat{Q}$ sarà una scarsa approssimazione della funzione ottimale Q^*. È anche importante notare che, in questa situazione, la stima di Q^* in tutti gli stati $s \in S$ sarà imprecisa, non solo in quelli visitati da π_e. Questo perché il fulcro del Q-Learning è quello di sfruttare un vincolo che lega insieme i valori di tutte le coppie stato-azione, il valore di una coppia $Q(s, a)$ non è indipendente dagli altri, ma è condizionato dagli aggiornamenti iterativi dell'algoritmo.

É quindi fondamentale scegliere correttamente π_e per raccogliere dati. Siamo noi a decidere in che modo far interagire l'agente con l'ambiente. Possiamo ad esempio scegliere una politica che porti il robot a selezionare in maniera completamente casuale un'azione dall'insieme A, in maniera uniforme. Tale politica visiterebbe con buona probabilità tutti gli stati della mappa, perlomeno se le dimensioni sono ancora contenute come nel nostro caso. Ci aspettiamo che in una distribuzione uniforme questo succeda, tuttavia sarebbero necessarie molte traiettorie. Non è sufficiente quindi utilizzare l'esplorazione casuale come unico metodo di apprendimento, giungiamo così alla seconda idea chiave di questo Q-Learning, ovvero affiancare lo *sfruttamento*. Le implementazioni tipiche del Q-Learning collegano la stima attuale di Q alla politica π_e stabilendo:

$$\pi_e(a \mid s) = \begin{cases} \arg\max_{a'} \hat{Q}(s, a') & \text{con probabilità } 1 - \epsilon \\ \text{scelta uniforme in } A & \text{con probabilità } \epsilon \end{cases}$$

Dove ϵ è detto "parametro di esplorazione", è scelto arbitrariamente dallo sviluppatore e ovviamente può variare nell'intervallo $[0, 1]$. Fino ad ora la π_e è stata chiamata politica di esplorazione; in seguito a questa integrazione assume il nome di politica ϵ-greedy. Sceglie l'azione ottimale sfruttando la conoscenza acquisita e immagazzinata in una struttura dati con probabilità $1 - \epsilon$, mentre esplora casualmente scegliendo dall'insieme A con probabilità ϵ. Ora attraverso l'unione di queste due tecniche abbiamo formulato una politica più equilibrata.

É importante notare che quando si utilizza un'esplorazione basata sulla stima attuale della funzione valore-azione $\hat{Q}$, è necessario risolvere periodicamente il problema di massimizzazione: $\arg\max_{a'} \hat{Q}(s, a')$. Le implementazioni tipiche del Q-Learning effettuano un aggiornamento su un piccolo gruppo di dati utilizzando alcune coppie stato-azione dal data-set raccolto, tipicamente quelle ottenute al passo precedente del robot dopo ogni azione eseguita con π_e. Aggiornare $Q(s, a)$ dopo ogni singola esperienza è inefficiente, per questo raccoglie un piccolo insieme di transizioni (s, a, r, s') che usa dopo ogni azione per aggiornare i valori di Q. Questo escamotage permette di ridurre la varianza migliorando l'apprendimento.

La proprietà di auto-correzione del Q-Learning Il data-set raccolto dal robot durante l'applicazione del Q-Learning cresce nel tempo. Sia la politica di esplorazione π_e che la stima $\hat{Q}$ si evolvono man mano che il robot raccoglie più dati. Questo ci fornisce un'idea chiave sul perché questo algoritmo funzioni bene. Consideriamo uno stato s: se ad un'azione a corrisponde un valore elevato secondo la stima corrente $\hat{Q}(s, a)$, allora la politica ϵ-greedy ha una maggiore probabilità di selezionare questa azione. Se tale azione non è effettivamente la migliore, gli

stati futuri che si raggiungono avendo scelto essa, avranno ricompense basse. Il successivo aggiornamento dell'obiettivo del Q-Learning ridurrà quindi il valore $\hat{Q}(s,a)$, diminuendo la probabilità di selezionare a la prossima volta che il robot visiterà lo stato s. Le azioni errate, ad esempio quelle il cui valore è sovrastimato in $\hat{Q}(s,a)$, vengono esplorate dall'agente, ma il loro punteggio viene corretto nel successivo aggiornamento dell'obiettivo del Q-Learning. Le azioni corrette, invece, il cui valore $\hat{Q}(s,a)$ è alto, vengono scelte più frequentemente dal robot e quindi rafforzate. Questa proprietà permette di dimostrare che l'algoritmo può convergere alla politica ottimale anche se inizia con una politica casuale π_e [12].

Questa capacità, non solo di raccogliere nuovi dati, ma soprattutto di aggiornarli in modo tale che siano corretti e utili al conseguimento dell'obiettivo è la caratteristica centrale degli algoritmi di apprendimento per rinforzo. Il Q-Learning, affiancato all'impiego di reti neurali profonde, è uno strumento molto potente anche per affrontare problemi moderni tramite apprendimento per rinforzo; a partire da [13], articolo che ha segnato la "rinascita" del RL, aprendo la strada agli sviluppi successivi come AlphaGo [14] e gli algoritmi di RL avanzati.

Implementazione Q-Learning Finalmente possiamo descrivere l'algoritmo vero e proprio che sarà tradotto in codice e usato per l'apprendimento dei nostri agenti. Ciascuno di essi durante l'allenamento fa esperienza tramite le seguenti tappe:

- Osserva lo stato s in cui si trova;
- Sceglie e performa un'azione a (tramite politica ϵ-*greedy*);
- Osserva lo stato successivo s';
- Riceve una ricompensa immediata($r(s,a)$);
- Regola il valore $Q(s,a)$ fornendo un'approssimazione più corretta.

La versione dell'algoritmo di Q-Learning che descrive questo aggiornamento è la seguente:

$$Q(s,a) \leftarrow Q(s,a) + \alpha \left[r + \gamma \max_{a'} Q(s',a') - Q(s,a) \right] \tag{2.2}$$

I parametri rappresentano:

- α: **tasso di apprendimento** (learning rate), controlla quanto velocemente l'agente aggiorna il valore $Q(s,a)$ in risposta a nuove informazioni; se $\alpha \to 1$ dà enfasi ai valori recenti, se $\alpha \to 0$ dà più peso alla conoscenza pregressa;
- γ: **fattore di sconto** (discount factor), determina quanto peso dare alla ricompensa calcolata nel nuovo stato rispetto a quello corrente; se $\gamma \to 1$

considera più importanti le ricompense future, mentre se $\gamma \to 0$ considera solo la ricompensa immediata;

- r: **ricompensa immediata** (immediate reward), ossia il premio fornito al passo t.

Vale la pena notare che nella teoria $Q_n(s,a) \to Q^*(s,a)$ per $n \to \infty$ se e solo se sono soddisfatte le seguenti condizioni. La ricompensa deve essere finita, quindi contenuta entro un certo numero: $|r_n| \leq R$. Il tasso di apprendimento deve essere assegnato in questo modo: $0 \leq \alpha_n < 1$ e

$$\sum_{i=1}^{\infty} \alpha_{n^i}(s,a) = \infty, \quad \sum_{i=1}^{\infty} [\alpha_{n^i}(s,a)]^2 < \infty, \quad \forall s,a$$

allora la convergenza di cui sopra avrà probabilità 1. Poiché è impossibile per ogni stato provare infinite volte un azione la nostra implementazione produrrà un'approssimazione, com'è ovvio che sia, naturalmente questa differenza è irrilevante ai fini dell'elaborato. I valori $Q(s,a)$, in continuo aggiornamento, vengono registrati all'interno di una struttura dati di supporto chiamata *Q-Table*.

Q-Table La Q-Table è una struttura dati di dimensioni SxA nella quale vengono memorizzati i valori attesi corrispondenti ad una certa azione a tra quelle disponibili, in un certo stato s. I valori iniziali di Q, $Q_0(s_0,a)$ per ogni coppia stato-azione sono assunti dati, compete a chi implementa l'algoritmo la scelta del valore iniziale. Procedendo con le iterazioni abbiamo constatato che il punteggio converge correttamente, per questo non è necessario inizializzare le celle con una quantità in particolare. Noi assumeremo che la Q-Table avrà tutte le celle inizialmente poste a zero. Vediamo adesso un primo esempio di come si presenta visivamente la tabella, ipotizziamo di avere in dato istante questa situazione:

Tabella 2.1. Esempio generico di Q-Table.

	a1	**a2**	**a3**
s1	0.1	0.2	0.3
s2	0.4	0.5	0.6
s3	0.7	0.8	0.9
s4	0.11	0.12	0.13

Conosciamo già l'algoritmo di apprendimento 2.2 e supponiamo anche di conoscere i parametri $\alpha = 0.1$ e $\gamma = 0.8$. Ci troviamo nello stato $s1$ ed eseguiamo l'azione $a1$ che ci porta in $s2$ ottenendo come ricompensa immediata $r = 5$. Ci chiediamo a questo punto quale sia il valore aggiornato della cella $Q(s1,a1)$. Lo stato in cui finiamo è $s2$, pertanto per il calcolo del nuovo valore prendiamo come descritto dal Q-Learning

il massimo punteggio ottenibile nello stato in cui giungiamo dopo la transizione, in $s2$ il valore massimo è "0.6", che corrisponde all'azione $a3$, quindi avremo:

$$Q'(s1, a1) = 0.1 + 0.1 * (5 + 0.8 * 0.6 - 0.1) = 0.638$$

Questo valore rappresenta il *max expected value* che possiamo ottenere in queste condizioni di partenza e che andrà a sostituire il dato precedente nella cella $s1xa1$.

Tabella 2.2. Prima cella della Q-Table aggiornata.

	a1	**a2**	**a3**
s1	0.1 → 0.638	0.2	0.3

Per comprendere meglio la struttura di una Q-Table ne prendiamo una relativa al progetto di tesi. É riportata nel seguito una parte di tabella relativa al Difensore.

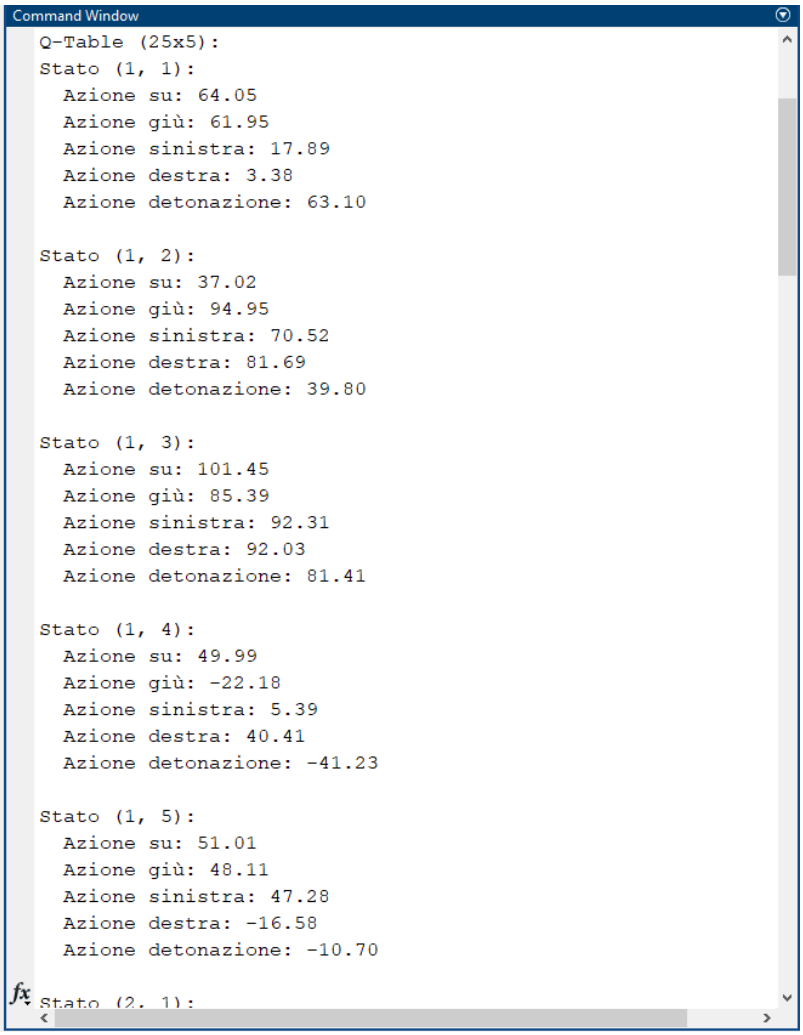

Figura 2.1. Output Q-Table MatLab.

Infine consideriamo ed analizziamo una Q-Table completa risultante al termine dell'allenamento (1000 iterazioni), di cui andiamo ad analizzare gli elementi.

	'Su'	'Giù'	'sinistra'	'Destra'
1	27.74	23.50	23.59	36.99
2	24.76	-72.15	-172.59	12.58
3	-1.54	9.31	-110.09	9.25
4	9.63	10.40	8.51	3.85
5	-32.13	57.91	-116.52	-20.38
6	22.55	23.34	23.60	21.67
7	0.00	0.00	0.00	0.00
8	8.41	-13.35	10.33	-15.70
9	-3.28	10.74	11.26	73.17
10	-35.54	80.35	-42.74	-11.94
11	23.60	24.76	23.93	21.50
12	11.21	-0.49	23.34	17.05
13	0.00	0.00	0.00	0.00
14	11.99	7.43	10.74	2.26
15	-7.66	4.00	-17.85	-15.83
16	24.12	25.79	25.96	22.68
17	52.74	-21.44	23.73	6.86
18	-2.91	18.39	-68.73	-5.28
19	-50.15	85.09	-42.07	1.02
20	-6.90	100.00	71.03	70.55
21	25.54	25.84	26.15	26.28
22	26.61	26.49	26.19	25.19
23	-0.17	27.33	25.23	85.09
24	88.28	83.55	88.17	100.00
25	0.00	0.00	0.00	0.00

Figura 2.2. Q-Table completa agente allenato.

Questa tabella riporta in maniera esaustiva tutti i valori calcolati dal Q-Learning al termine dell'allenamento, questo significa che in ogni cella è riportato il valore a cui converge l'algoritmo per ogni combinazione stato-azione. La mappa si compone di 25 stati come descritto nella seguente sezione dell'appendice: A.1 ; il numero di colonne è evidentemente pari a 4, queste rappresentano le generiche azioni di movimento riproducibili in un ambiente bidimensionale.

Ogni valore $Q(s, a)$ rappresenta una stima del valore atteso che un agente può ottenere eseguendo l'azione a trovandosi in s. Com'è naturale pensare saranno preferite dal robot le azioni in corrispondenza delle quali si ottengono ricompense positive e maggiori, mentre saranno scoraggiate quelle con i punteggi più bassi. Ci si presentano di fronte tre possibili casi:

- **Valore positivo:** indica che l'azione a nello stato s conduce a ricompense future elevate;
- **Valore negativo:** indica che l'azione a nello stato s porta a penalità e ostacola il raggiungimento dell'obiettivo;
- **Valore nullo (pari a 0):** indica che una certa azione non ha impatto immediato sul punteggio e può essere neutrale ai fini dell'obiettivo.

Le reward positive e negative risultano intuitive, rappresentano rispettivamente scelte giuste e sbagliate. I valori nulli, invece, sono meno immediati da comprendere. Bisogna notare innanzitutto che all'inizio dell'allenamento tutti gli elementi $Q(s, a)$ sono nulli in quanto il robot non ha esplorato la mappa. Al termine però, se in un certo stato s ogni azione presa dall'agente non porta ad un reward futuro (ho una riga di zeri), mi trovo in corrispondenza di un caso particolare. Questi sono detti *stati terminali*, ossia corrispondono a situazioni in cui l'episodio (la nostra simulazione) termina. Non a caso sono in corrispondenza delle celle dove: sono presenti gli ostacoli o si tratta della casella *Goal* (obiettivo dell'Attaccante). Questo ci fa dedurre come difatti la Q-Table mostrata appartenga all'allenamento dell'agente attaccante. Infine, potremmo imbatterci nel caso in cui solo in corrispondenza di alcune colonne il valore $Q(s, a)$ risulta nullo, in sintesi possiamo dedurre che ci troviamo in una delle seguenti condizioni:

- Non è stata ancora presa una certa azione nello stato in questione, quindi banalmente non abbiamo dati a riguardo con cui aggiornare il valore della cella;
- L'azione presa non influisce rispetto al raggiungimento dell'obiettivo da parte dell'agente, quindi la ricompensa è pari a zero.

2.2.1 Progettazione dell'Ambiente

Per la parte progettuale è stato molto interessante approfondire e prendere spunto da un articolo, [15] che analizziamo. Alcuni ricercatori hanno affrontato nell'ambito dei robot mobili il problema dell'evitamento degli ostacoli tramite sensori e algoritmi di apprendimento per rinforzo. Una grande differenza si riscontra tra il caso in cui l'ambiente sia completamente noto e quello in cui la conoscenza sia parziale o addirittura assente. Nel primo caso strategie classiche di movimento globale e di pianificazione del percorso possono essere applicate con ottimi risultati come descritto in [16]. In ambienti più complessi e non strutturati queste tecniche peggiorano rapidamente in quanto è necessaria una modellazione considerevole che non sempre si ha la possibilità di attuare. Un buon metodo per risolvere il problema in maniera

model-free, come abbiamo dimostrato, è tramite RL dal momento che non è richiesta questa conoscenza completa dell'enviroment. Il robot utilizzato in questo esperimento è un Bot n' Roll One A (sviluppato da botnroll.com) e mostrato qui di seguito. Gli

Figura 2.3. Bot n' Roll One A.

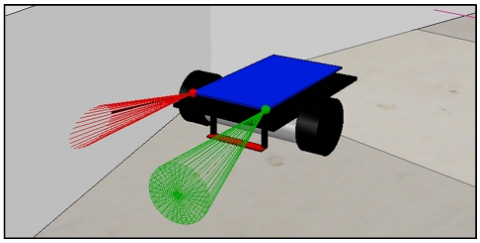

Figura 2.4. Screenshot dell'ambiente simulato.

ambienti dove è stato testato sono tre: si tratta di labirinti simulati al calcolatore. L'obiettivo dell'agente è quello di trovare l'uscita evitando i muri perimetrali del percorso. Per far apprendere la politica ottimale è stato utilizzato il Q-Learning. Oltre all'hardware di cui è composto il robot (Arduino, motori DC, ruote e circuiteria varia), esso è stato dotato un sensore ad infra-rossi per rilevare la distanza con gli ostacoli. Lo script di controllo riceve i dati dal sensore, questi vengono processati attraverso l'algoritmo di Q-Learning ottenendo in uscita i risultati desiderati per il campione fornito. Dopodiché gli output, i parametri impostati per la simulazione e il controllo vengono inviati al simulatore. Per avere successo l'agente deve essere in grado di raggiungere il punto finale risolvendo i tre labirinti di complessità via crescente, mostrati in Figura 2.5.

Il primo è composto da una linea retta e stretta di 1.2 metri circa. Il secondo lungo circa 5 metri ha due curve in entrambe le direzioni, che obbligano il robot ad imparare come sterzare quando il perimetro non è dritto e quindi evitare i muri da entrambi i lati. Anche il terzo, di 8.3 metri circa, è dotato di curve su entrambi i lati, la larghezza della strada è poca, ci sono bordi affilati e rientranze profonde. La distanza trai muri in media è di 0.5 metri.

Spazio di stato I sensori offrono dati discreti, fornendo informazioni binarie: 0 se non viene rilevato nulla, 1 se c'è un ostacolo. Avendo un sensore a destra e uno a sinistra in totale si hanno 4 possibili combinazioni, ossia stati del sensore. Il volume di spazio rilevato ha una forma conica, la distanza massima di rilevamento è di 20 cm e l'angolo di visione è pari a 20°.

Spazio delle azioni L'azione a è generata da uno script di controllo inviato all'ambiente di simulazione e viene eseguito all'agente in tempo reale. Il comando

[0]I sensori appaiono rossi quando rilevano un ostacolo e verdi quando non ne rilevano.

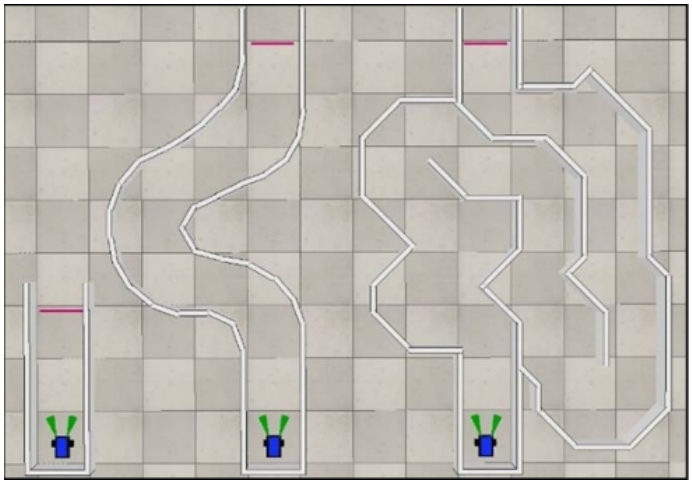

Figura 2.5. I tre labirinti di complessità crescente (da sinistra a destra). Il punto di partenza è rappresentato dai robot, mentre il punto di arrivo è indicato dalle linee magenta.

non è altro che una scelta da un insieme pre-determinato di possibili azioni (spazio delle azioni discreto). L'insieme è composto dalle seguenti azioni di movimento: *avanti*, *dietro*, *gira a destra* e *gira a sinistra*; ciascuna può essere compiuta con 3 differenti modalità di velocità. Inoltre sono attuabili 2 movimenti di rotazione sul posto, per un totale di 14 possibili azioni.

Politica Anche qui come nel nostro progetto viene sfruttata la funzione stato-azione aggiornata ad ogni iterazione, quindi qualvolta l'azione a_t venga scelta si sostituisce il punteggio in $Q(s_t, a_t)$ fino a che non terminano gli episodi di allenamento. A questo punto in funzione dell'efficacia dell'apprendimento si avrà la convergenza verso le utilità ottime.

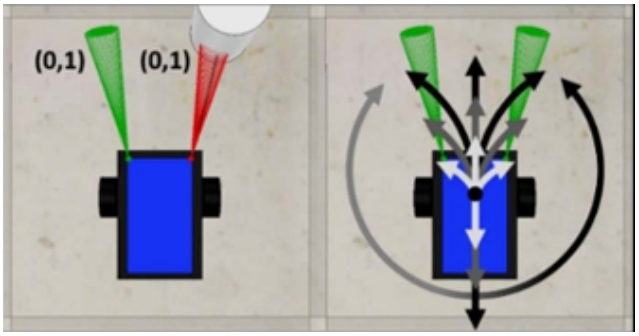

Figura 2.6. Dimostrazione visiva dello spazio di stato (a sinistra) e dello spazio delle azioni (a destra) con l'agente simulato.

Ricompensa La funzione ricompensa è denotata da R, ed è caratterizzata dalle seguenti componenti dove l'indice $i \in \{1, 2, 3\}$ è una variabile discreta in cui le tre cifre rappresentano il livello di difficoltà del labirinto. La funzione è così definita:

$$R(i) = \begin{cases} -1, & \text{per ogni passo} \\ -10^3 \delta_u(i-1), & \text{per timeout} \\ -10^3(40^{i-1}), & \text{per collisione} \\ 10^3(50^{i-1}), & \text{per successo} \end{cases} \tag{2.3}$$

Il basso punteggio negativo fornito ad ogni passo temporale, obbliga l'agente a cercare di raggiungere il successo il più velocemente possibile (come nel nostro caso) per massimizzare la ricompensa. Per quanto riguarda il caso di timeout, δ_u rappresenta una funzione di impulso unitario descritta da:

$$\delta_u(t-T) = \begin{cases} 1, & t - T = 0 \\ 0, & t - T \neq 0 \end{cases} \tag{2.4}$$

Il robot riceve una penalità quando rimane bloccato eseguendo la stessa azione senza fare progressi, questo impedisce che resti fermo indefinitamente. Solamente mentre affronta il primo livello, questo tipo di penalità è importante nelle prime fasi dell'apprendimento, per evitare che il robot sviluppi strategie inefficaci. Nei labirinti successivi, l'algoritmo è già migliorato e la penalità non è più necessaria. Man mano che il robot avanza nei livelli, le ricompense aumentano esponenzialmente, incentivandolo ad imparare strategie più avanzate e a risolvere i labirinti in modo più efficiente. Una volta impostati i parametri si è definita la strategia di bilanciamento tra esplorazione e sfruttamento utilizzando, come abbiamo sfruttato noi, l'$\epsilon - greedy$.

$$\pi(s) = \begin{cases} \text{casuale: } a \in A_s, & \text{se } E < e \\ \arg\max a, & \text{se } E \geq e \end{cases} \tag{2.5}$$

Dove $E \in [0, 1]$ è un numero casuale uniforme estratto ad ogni iterazione. Sono state testate diverse funzioni $\epsilon - greedy$ decrescenti nel tempo per il confronto dei risultati. In secondo luogo, sono stati utilizzati i Valori Iniziali Ottimistici (OIV), una tecnica usata nel RL per incoraggiare l'esplorazione degli stati da parte dell'agente. Gli OIV consistono nell'inizializzare tutti i valori di $Q(s, a)$ con un numero notevolmente alto, in questo modo poiché il Q-learning seleziona le azioni in base ai valori più alti di Q, il robot inizialmente tenderà a provare ogni azione almeno una volta per verificare se la ricompensa reale corrisponde al valore iniziale. Con il tempo, il valore di $Q(s, a)$ verrà aggiornato con i dati reali. Se un'azione non porta ad una

buona ricompensa, il valore $Q(s, a)$ si abbasserà e l'agente smetterà gradualmente di sceglierla. Questo processo lo costringe ad esplorare tutte le opzioni disponibili, senza rimanere bloccato su azioni sub-ottimali troppo presto. Pertanto, tutte le variabili si stabilizzeranno eventualmente sui loro veri valori.

Sono state definite due varianti del Q-Learning, la variante A prevede che il rispettivo valore $Q(s, a)$, dopo esser stato aggiornato, invii un'azione al simulatore, questo avviene ad un intervallo fisso Δt, il che significa che il robot eseguirà un numero costante di iterazioni al secondo. Per ogni Δt, un'azione verrà calcolata e selezionata tramite la strategia di esplorazione/sfruttamento $\epsilon - greedy$, e inviata all'agente. Questo metodo permette all'agente di sperimentare diverse azioni anche se non c'è un cambiamento nello stato; l'implementazione B ha le stesse caratteristiche per quanto riguarda Δt, ossia leggerà i valori del sensore ogni intervallo di tempo e calcolerà il $Q(s_t, a_t)$ aggiornato, tuttavia, questo metodo differisce dal precedente poiché sceglie una nuova azione solo quando si verifica un cambiamento di stato. L'agente continuerà ad eseguire l'azione ricevuta e a calcolare Q fino a quando non avviene una transizione di stato. Solo quando i dati del sensore cambiano, allora un'azione passerà attraverso la strategia $\epsilon - greedy$. La principale differenza tra i due algoritmi si verifica quando l'agente rimane nello stesso stato per un periodo di tempo significativo. Il Metodo A può alterare l'azione che il robot sta eseguendo ad ogni iterazione, ricalcolando $Q(s_t, a_t)$ e sfruttando la politica greedy. Il Metodo B invece, deve mantenere l'azione scelta fino a che non avviene una transizione di stato. Entrambi i metodi possiedono vantaggi e svantaggi, i risultati li metteranno in evidenza.

Risultati L'articolo riporta attraverso i numerosi grafici in maniera esaustiva i risultati dell'apprendimento e di altre metriche. Noi ci focalizziamo sull'analisi della Reward Function. Cominciamo visualizzando le performance del primo caso (A):

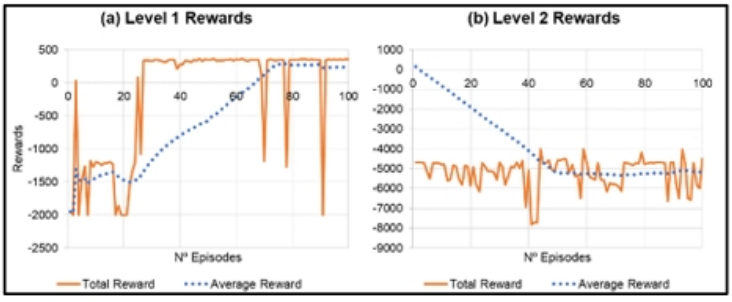

Figura 2.7. Metodo A grafica della ricompensa.

I grafici riportano sull'asse delle ascisse il numero degli episodi (100) e sull'asse delle ordinate il punteggio. La linea continua in arancione evidenzia il punteggio

ottenuto al termine di ogni episodio. L'andamento tratteggiato, in blu, ne evidenzia la media. Come si evince dall'immagine, il primo livello (sinistra) è stato superato con successo, il punteggio in media aumenta verso la metà degli episodi e soprattutto converge verso un valore positivo. Il secondo labirinto (destra) non viene superato con successo, la funzione scende vertiginosamente verso valori negativi. Essendo il terzo caso ancora più complesso non è riportato i quanto questo metodo A si è dimostrato strategicamente più debole. Proseguiamo con l'algoritmo B, di cui seguono i seguenti risultati:

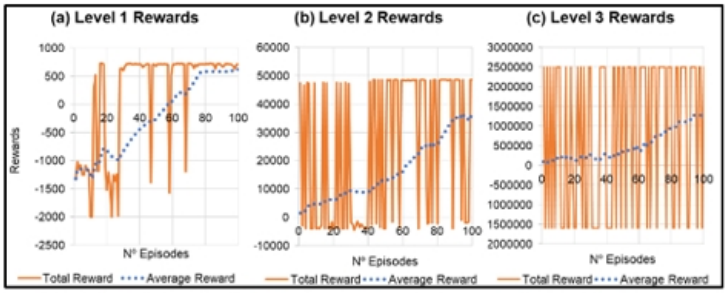

Figura 2.8. Metodo B grafica della ricompensa.

Con questa versione il robot è stato in grado di superare tutti e tre i livelli, gestendo in maniera ottima la scelta delle azioni, rendendosi conto che i buoni risultati a cui alcune di esse portavano, erano solamente causati da eventi fortuiti. Nonostante l'incertezza nell'apprendimento, il punteggio medio risultante è crescente e molto positivo. Quindi alla fine il metodo A si è rivelato sicuro (meno oscillazioni) ma lento nell'apprendimento, inefficace per il caso studio nonostante i numerosi iperparametri testati dai ricercatori. L'agente tendeva a ristagnare il suo apprendimento, in pratica tendeva a girare su se stesso in cerca di un muro e poi prendeva una direzione evitandolo. Si illudeva di star agendo correttamente perché in questo modo sceglieva due azioni che sommate corrispondevano ad un valore positivo.

Questo comportamento nel primo livello completamente dritto, gli ha permesso di superarlo anche se in maniera estremamente lenta, già dal secondo nel quale erano presenti numerose curve questo approccio è risultato improduttivo.

Il metodo B si è rivelato più rapido ed ha permesso la risoluzione dei tre labirinti. Nel livello uno l'apprendimento è stato fluido, nel secondo è risultato abbastanza instabile e nel terzo completamente instabile. Nonostante ciò le funzioni di ricompensa sono cresciute rapidamente dimostrando il successo dell'apprendimento dell'agente. Si riporta una rappresentazione visiva dell'evoluzione della navigazione del robot per il metodo B. I colori: rosso, arancione, giallo e verde (dal peggiore al migliore), corrispondono al numero dell'episodio nel quale viene tracciato il percorso.

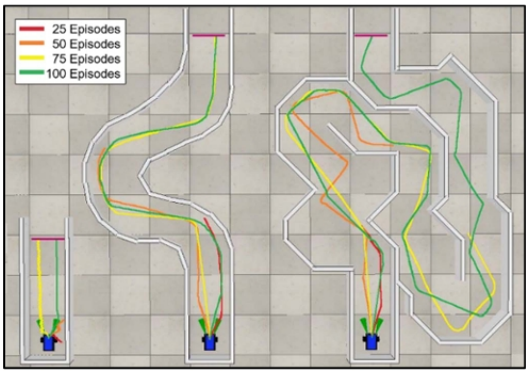

Figura 2.9. Evoluzione navigazione nei tre labirinti.

Ogni 25 episodi viene riportato il percorso seguito dall'agente, mostrando il perfezionamento delle sue scelte e il raggiungimento del traguardo in ognuno dei labirinti.

Capitolo 3

Simulazioni

3.1 Interpretazione Strategica

Nei paragrafi successivi ci immergeremo in un ambiente bidimensionale, nel quale, come fosse un gioco, osserveremo due agenti sfidarsi ognuno per raggiungere il proprio obiettivo, da noi formulato. Il motivo principale di questo gioco, nonché scopo dell'elaborato, è quello di testare l'efficacia del Q-Learning per risolvere problemi di questo tipo. Prima di passare alla simulazione, tuttavia analizziamo come nella letteratura ci siano una molteplicità di altri contesti nei quali vale la pena studiare lo scontro fra due agenti in un contesto avversariale. I giochi di sicurezza (security games) ne sono un esempio, essi forniscono un quadro per modellare l'interazione tra attaccanti e difensori. Questi giochi e le loro soluzioni di equilibrio sono utilizzate come basi per prendere decisioni formali e sviluppare algoritmi, così come può essere di interesse prevedere il comportamento di un attaccante. I campi di applicazione sono svariati, dalle reti sociali a quelle wireless o ancora per quanto riguarda i veicoli che circolano in una rete stradale.

Modello dei giochi di sicurezza La sicurezza di rete può essere immaginata come una partita strategica giocata tra attaccanti malevoli, il cui obiettivo è quello di compromettere la rete, e i difensori, rappresentati dagli amministratori o possessori dell'infrastruttura che la difendono. Durante il gioco interagiscono numerosi sistemi, gli attaccanti sfruttano le vulnerabilità e i difensori rispondono con delle contromisure. La teoria dei giochi (game theory) fornisce le basi matematiche rigorose per formalizzare questi "giochi metaforici" riguardanti le reti e l'interazione delle due fazioni. La teoria dei giochi si occupa delle interazioni strategiche tra diversi agenti, chiamati giocatori, che prendono decisioni in contesti di interdipendenza. Ogni giocatore ha una sua funzione obiettivo che prova a massimizzare, come il caso dei nostri due agenti con la loro utility function, o minimizzare se questa funzione

rappresenta un costo o una perdita. Abbiamo detto che i giocatori sono interconnessi nell'ambiente infatti il singolo non può semplicemente ottimizzare la propria funzione indipendentemente dalle scelte degli altri giocatori. Quindi anche in un contesto non cooperativo le azioni degli agenti sono comunque connesse in qualche modo. Nel nostro progetto non è possibile alcun tipo di cooperazione pertanto rientriamo nella teoria dei giochi non-cooperativa, nella quale solo uno dei due agenti ha la meglio sull'altro e trova soluzione al suo problema. Generalmente questo tipo di giochi sono detti a somma zero (zero sum), perché sommando le funzioni obiettivo di entrambi gli agenti otterremmo zero, insomma la funzione obiettivo di uno è l'opposto dell'altro $f_A = -f_D$.

Nel nostro caso è bene notare che non è valido questo principio, in quanto oltre alla risoluzione del task vogliamo che ciò avvenga nel minor numero di passi possibili, quindi abbiamo penalizzato ogni spostamento assegnando -1. Inoltre anche per chi colpisce un ostacolo è previsto un malus. Il gioco è detto "finito" se il numero di alternative che può scegliere un giocatore è limitato, i nostri insiemi S ed A lo sono, pertanto come preannunciato nella sezione 1.2 rientriamo in questa categoria. Un altro nome che identica questa tipologia è proprio giochi matrice o matrix games in inglese. Infine un'altra caratteristica del nostro caso studio, è la presenza di aleatorietà nell'evoluzione del gioco, questo rende il gioco stocastico invece di deterministico.

Torniamo ai giochi di sicurezza, com'è naturale pensare possono essere utili sia per addestrare la parte offensiva che quella difensiva, per entrambi si punta al giusto compresso tra lo sfruttamento delle risorse e la massimizzazione del risultato. Dipende da quale punto di vista vogliamo analizzare la battaglia e qual è l'applicazione nel mondo reale che stiamo considerando; nel nostro progetto non siamo schierati, abbiamo interesse a valutare le capacità di entrambe le facce della medaglia. Abbiamo nel corso dell'elaborato introdotto le componenti necessarie per definire un security game, che ricapitolando sono:i giocatori (attaccanti e difensori); lo spazio delle azioni (insieme di azioni per entrambe le fazioni); il risultato (ricompensa o costo per ogni azione-reazione) e la struttura delle informazioni (completa o parziale osservabilità da parte dell'agente).

Oltre alle azioni che permettono ai robot di interagire con l'ambiente definiamo cos'è una *strategia*. Essa è una regola di decisione, il cui risultato è un azione. Nei giochi statici a singolo turno (static one-shot games) la distinzione tra azione e strategia si appiattisce fino a scomparire, perché ogni giocatore prende una decisione una sola volta, senza la possibilità di modificare la propria scelta in base alle azioni future degli altri. Un esempio classico e interessante è il Dilemma del Prigioniero, in cui due giocatori devono decidere contemporaneamente se cooperare o tradire. In

questo caso, ciascun giocatore sceglie la propria risposta senza poter modificare la decisione in base all'apprendimento o alla conoscenza della scelta dell'altro, come avverrebbe in giochi con più turni o possibilità di interazione.

Nel progetto verrà proposta una dinamica in cui saranno presenti delle strategie che i due agenti adotteranno, si veda ad esempio la sezione A.7 in cui ad ogni agente sarà fornito un sensore che permetterà decisioni mirate. Il risultato (outcome) è quantificato dal guadagno o dalla perdita che ogni giocatore ha subito al termine del gioco. Formalmente il nostro progetto rientra nei giochi a somma non-zero, ed avendo due giocatori viene detto gioco bi-matrice (bi-matrix).

Concludiamo questa sezione introduttiva del capitolo con un particolare tipo di giochi di sicurezza detti stocastici o di Markov. Rientrano particolarmente nel nostro interessare visto il contesto della parte progettuale. Questa sezione estende i giochi deterministici, infatti sebbene gli stocastici siano matematicamente più complessi, ci consentono di studiare l'interazione attaccante-difensore in modo più realistico. Attraverso la teoria della probabilità è possibile modellare anche parametri sconosciuti e incontrollabili che insorgono nei problemi di sicurezza, oltre alla natura dinamica di questi parametri di gioco nascosti, le interdipendenze e i fattori esterni che vengono catturati all'interno della caratteristica stocastica.

Markov Security Games Gli approcci probabilistici sono stati ampiamente utilizzati per modellare guasti, errori e falle nei sistemi di rete per quanto riguarda affidabilità e dipendenza. Tuttavia lo studio degli incidenti di sicurezza differiscono dall'analisi di affidabilità tradizionale. A differenza dei guasti e degli errori non pianificati (casuali) nel sistema, le manomissioni della sicurezza sono causati da attaccanti con precise intenzioni malevoli. I modelli probabilistici e approcci derivanti dalla teoria dei giochi sono stati utilizzati per valutare questo tipo di rischi riguardanti la sicurezza. I modelli stocastici di Markov catturano la complessità e le proprietà ignote della rete sottostante il sistema molto meglio dei modelli deterministici. Mentre nei giochi di sicurezza statici i giocatori ottimizzano le strategie tenendo conto dei costi scontati futuri, il che consente di perfezionare in un orizzonte temporale; i giochi di Markov si basano sugli MDP come fondamento teorico per lo sviluppo e l'analisi delle strategie dei giocatori. In questo caso giocano un attaccante P^A e un difensore P^D nello spazio di stato rappresentante l'ambiente. Uno stato può essere una modalità operativa del sistema in rete, come quali unità sono operative, contromisure attive, o se parti del sistema sono compromesse. In questi modelli gli stati evolvono in modo probabilistico secondo un processo stocastico definito con la proprietà di Markov. Questa permette di modellare l'effetto delle azioni dei giocatori sulle proprietà del sistema. Allo stesso tempo il modello dello spazio di stato stocastico della rete fornisce una base per analizzare le sue proprietà di sicurezza.

Sulla base delle strategie dell'attaccante ad esempio le probabilità di transizione tra gli stati possono essere calcolate per vari casi, quindi si ottiene un diagramma dello stato di sicurezza della rete che incorpora esplicitamente il comportamento dell'attaccante. I giocatori nei giochi di sicurezza sono spesso avversari diretti, quindi vengono modellati come giochi a somma zero nella maggior parte dei casi. Metodi di programmazione dinamica sono utilizzati per risolvere questi problemi.

Risolvere i giochi di Markov Esiste una stretta relazione tra i modi di risoluzione dei giochi di Markov e i processi decisionali di Markov. Se uno dei giocatori in un gioco di sicurezza stocastico adotta una strategia fissa, allora il gioco degenera in un problema di ottimizzazione per l'altro giocatore e il gioco di Markov si trasforma in un MDP. In pratica se in un gioco di Markov ci sono più giocatori che prendono decisioni contemporaneamente se uno di loro usa una politica fissa ecco che solo un giocatore sta effettivamente apprendendo. Di conseguenza, i metodi per risolvere gli MDP, come l'iterazione valore (value iteration), sono direttamente applicabili [17]. Nel nostro progetto si avrà una situazione intermedia esposta nel paragrafo successivo.

3.2 Ambiente Grid-World

L'ambiente nel quale evolveranno i comportamenti e quindi gli episodi riguardanti i nostri due agenti è di tipo bidimensionale, è equivalente ad una matrice, per questo può essere chiamato grid-world. Corrisponde infatti ad una griglia in cui ogni casella può essere una posizione occupata dai robot. Le dimensioni dell'enviroment sono di 5 righe e 5 colonne quindi con un totale di 25 possibili stati. Introduciamo poi quelle che sono le posizioni iniziali dell'Attaccante e del Difensore, in questa fase di progetto, rispettivamente le caselle $[1, 1]$ e $[1, 5]$ e gli altri elementi fissi, quali gli ostacoli, due, nelle celle $[2, 2]$ e $[3, 3]$. Infine la casella goal che rappresenta la Base del nostro Difensore e contemporaneamente l'obiettivo da distruggere per il nostro Attaccante situata in $[5, 5]$. Per approfondimenti sulla parte di codice si può consultare A.1. L'output risultante, nella configurazione iniziale è mostrato in Figura 3.1:

L'attaccante è raffigurato dal quadrato verde, il difensore da quello blu e la base in rosso. In grigio sono evidenziati gli ostacoli che entrambi gli agenti dovranno imparare a riconoscere come pericolosi ed evitare.

3.3 Azioni

Entrambi gli agenti sceglieranno autonomamente le azioni da performare tramite Q-Learning e la politica $\epsilon - greedy$. La parte algoritmica è già stata analizzata

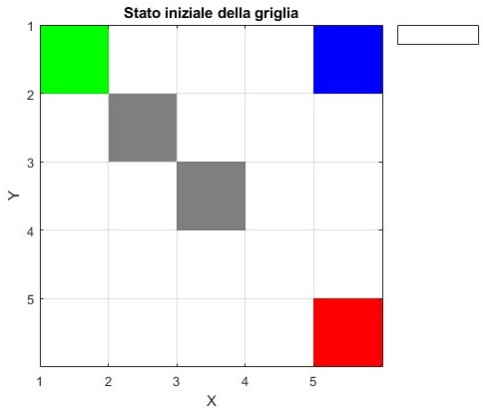

Figura 3.1. Mappa nello Stato Iniziale.

approfonditamente, per l'implementazione informatica consultare le sezioni A.2 e A.3 dell'appendice. In ogni stato il robot può scegliere se muoversi nella mappa tramite le mosse: *su*, *giù*, *sinistra*, *destra*; o se attaccare mediante la *detonazione*. Durante le simulazioni l'Attaccante dovrà giungere in prossimità della base del Difensore, quest'ultima rappresenta il target da distruggere. Nel frattempo il Difensore sarà libero di muoversi per ostacolare l'avversario nel suo intento e difendere la casella rossa.

3.4 Punteggio delle Rewards

Le ricompense sono il fulcro intorno al quale ruota il nostro algoritmo, sappiamo che quantificano il feedback che un agente riceve dall'ambiente dopo aver compiuto un'azione. Come per altre quantità citate nell'elaborato, non c'è un valore preciso da assegnare, siamo liberi di premiare o penalizzare gli agenti a nostro piacimento. Per una scelta di progetto si è voluto assegnare un punteggio negativo piccolo ogni volta che viene presa un'azione, con il fine di completare l'obiettivo nel minor numero di passi possibili e grande quando il robot mette in grave pericolo la propria incolumità, cioè in caso di collisione con gli ostacoli o di detonazione inefficace (inutile). Per quanto riguarda i punteggi positivi, saranno alti e verranno assegnati in caso di raggiungimento dell'obiettivo prefissato. Possiamo quindi riassumere le ricompense assegnate:

- -1 punto $\forall$[1] spostamento.
- -10 $\forall$ collisione con gli ostacoli.

E in funzione del ruolo avremo per l'Attaccante:

[1]Simbolo che indica "per ogni"

- +100 se raggiunge la casella goal.
- −100 se fatto detonare dal Difensore.

e per il Difensore in maniera complementare:

- +100 se annienta l'Attaccante.
- −100 se non protegge dall'Attaccante la Base.
- −10 se detona a vuoto (detonazione inefficace).

3.5 Allenamento

Dopo aver sviluppato le altre numerose funzioni, riportate per completezza nei capitoli in appendice (A.4, A.5, A.6), che gestiscono le nostre simulazioni, possiamo eseguire il codice nella sua totalità e valutarne nelle condizioni attuali l'efficacia. Quello che stiamo esaminando, quindi, è l'apprendimento dei nostri agenti. Vogliamo allenare sia l'Attaccante che il Difensore su un totale di 1000 episodi, lungo i quali diversi scenari si paleseranno, il nostro obiettivo è che entrambi apprendano delle strategie efficienti per completare i loro *task*. Valuteremo la bontà dell'algoritmo di Q-Learning con il quale l'Attaccante dovrà: esplorare la mappa, trovare la base e una volta arrivato ad una casella adiacente a questa farla detonare. Il Difensore avrà come compito quello di difendere la propria base e detonare in prossimità dell'Attaccante trovandosi sempre in una cella adiacente ad esso.

Vogliamo addestrare entrambi gli agenti, ma solamente uno per volta. Quindi mentre uno sta apprendendo, l'altro fungerà da elemento dinamico di disturbo. Quello che si allena sfrutta il Q-Learning, la propria Q-Table (aggiornata ogni ciclo) e la politica $\epsilon - greedy$; l'altro agente "fantoccio" (dummy), che rappresenta l'avversario sarà comunque un operatore in movimento, tuttavia prenderà le proprie decisioni in maniera casuale, quindi senza sfruttare l'algoritmo di apprendimento per rinforzo. Rappresenterà se vogliamo un elemento stocastico, di incertezza per l'agente che invece tenterà di risolvere il suo problema di massimizzazione del punteggio. Allenare solo uno degli agenti alla volta ci permette di far convergere l'algoritmo. Se li allenassimo simultaneamente (come testato) interferirebbero con il processo di apprendimento dell'altro, imparerebbero a compensare le strategie dell'avversario ostacolando la convergenza verso la *policy* ottimale. Come esposto precedentemente questo è un fenomeno tipico nei giochi non-cooperativi, dove la vittoria di uno incide sull'altro in maniera negativa. La ricerca di un equilibrio competitivo risulterebbe fallimentare in quanto non sarebbe possibile raggiungere il cosiddetto equilibrio di Nash.

3.5.1 Attaccante

Visualizziamo ora l'output al termine di una simulazione, completati gli episodi il punteggio dell'Attaccante converge, tutte le simulazioni forniscono lo stesso valore di convergenza che si assesta a 94 punti. Le funzioni ricompensa, che mostrano l'andamento della performance variano naturalmente in funzione della simulazione, dato che vi sono elementi aleatori, tuttavia non è rilevante ai fini della convergenza ultima delle utilità. In questo ambiente ristretto, possiamo verificare manualmente che torna, considerando i punteggi che abbiamo assegnato. Se ogni passo toglie 1 al nostro agente e il raggiungimento dell'obiettivo invece ne fa guadagnare 100, il massimo ottenibile si otterrà percorrendo il minor numero di passi, 6, poi detonando (-6+100 = 94). Per valutare l'apprendimento nel tempo esaminiamo la seguente *Reward Function*.

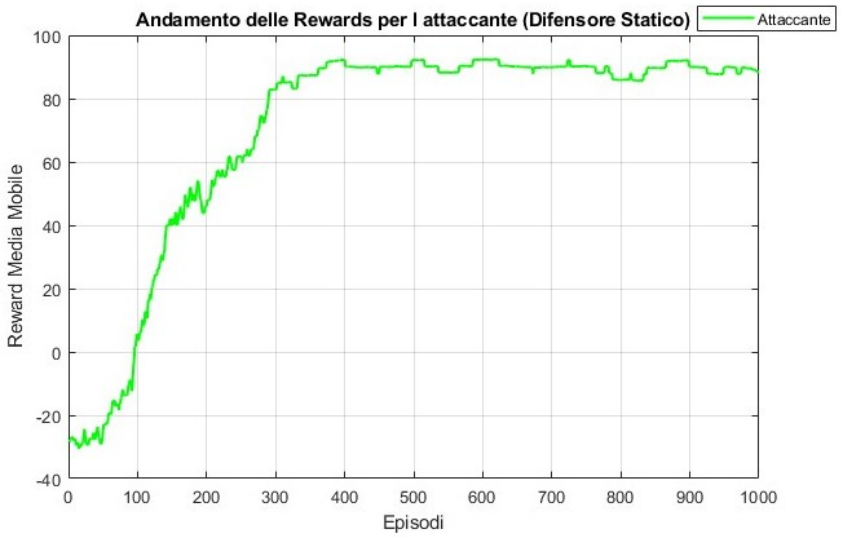

Figura 3.2. Training Attaccante.

3.5.2 Difensore

In maniera speculare alleniamo il Difensore mentre l'Attaccante prende azioni esclusivamente randomiche, notiamo che il *target* del Difensore non è fisso come lo è la casella goal per l'Attaccante, quest'ultimo seppur in maniera casuale si muove, pertanto è più complesso trovare una policy ottima per la difesa. A discapito di del Difensore, in alcuni episodi l'Attaccante tende a scontrarsi con gli ostacoli vicini, facendo terminare la simulazione. Il Difensore non ha il tempo necessario ad esplorare le zone adiacenti che deve ricominciare dalla cella di partenza. Per una scelta di realismo, anche l'agente dummy può interagire con l'ambiente, tra cui

questi ostacoli che simulano un danneggiamento dei robot. Nonostante ciò, il numero di episodi è sufficiente a far convergere le utilità della Q-Table. Questo si traduce in una maggiore oscillazione della *Reward Function*, la quale comunque permette al Difensore di convergere verso una *policy* ottima. L'andamento sarà il seguente:

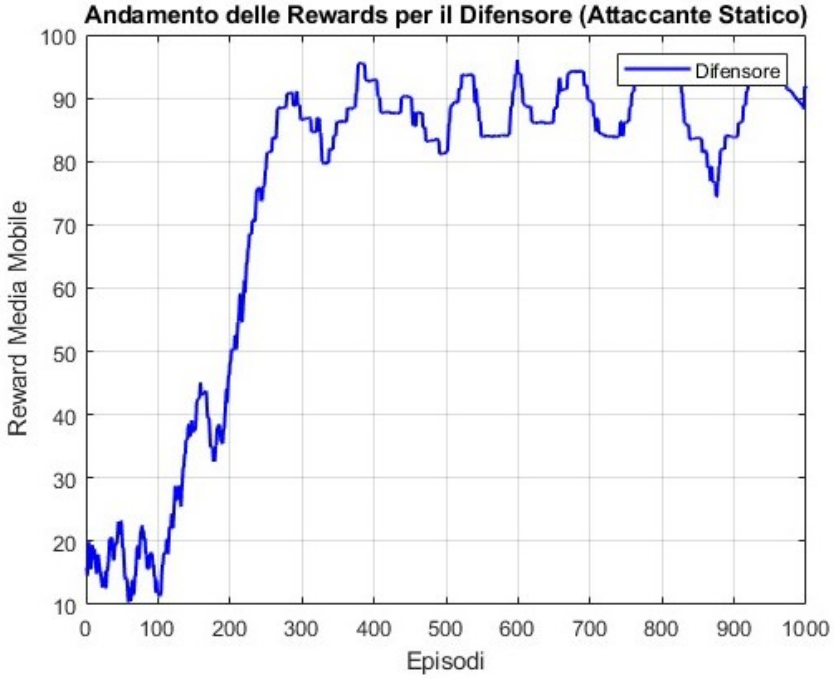

Figura 3.3. Training Difensore.

Si noti come risulti di fondamentale importanza processare e leggere i dati correttamente, l'output visualizzato tramite grafico della Reward Function rappresenta i risultati tramite media mobile[2] [18].

Questo approccio è adottato per rendere più chiara la tendenza generale delle ricompense ricevute nel tempo, è stata utilizzata una finestra temporale di 50 elementi per avere un intorno sufficientemente grande per valutare il trend complessivo del comportamento degli agenti, ma non troppo piccolo da rendere i dati indecifrabili. Una finestra di mille elementi (pari al numero di episodi) avrebbe senza dubbio fornito delle funzioni con meno oscillazioni, più *smooth*, tuttavia avrebbe mascherato il vero andamento delle ricompense, rendendo più complesso intervenire per risolvere problemi di mancato apprendimento.

[2]Un esempio di processo non markoviano con una rappresentazione markoviana è una serie temporale a media mobile.

3.6 Sensoristica

Nella letteratura diversi articoli analizzano come coniugare l'algoritmo di Q-Learning con lo sfruttamento di sensori, con il solito fine di far apprendere strategie ottime agli agenti tramite il paradigma del *trial and error*, ma evitando direttamente situazioni potenzialmente pericolose, sfruttando le capacità percettive dei sensori senza la necessità di sperimentarle fisicamente.

Torniamo al nostro progetto, nel quale implementiamo anche noi (virtualmente) dei sensori,che verranno sfruttati nella stessa mappa e per gli stessi task di prima. Per rendere la simulazione più realistica ipotizziamo quindi di dotare entrambi di un sensore di prossimità ad esempio ad infrarossi. Questo strumento permette a chi lo sta utilizzando di rilevare se vi è un altro oggetto in una posizione adiacente alla sua, quindi funziona entro un certo *raggio* (range). Permetterà ai robot di avere un'informazione in più con cui poter scegliere attivamente l'azione più corretta. Nel nostro progetto questo si traduce nelle seguenti possibilità:

- Il Difensore può ingaggiare ed inseguire l'Attaccante;
- L'Attaccante può fuggire dal Difensore;
- L'Attaccate può rilevare la base nemica e farla detonare in maniera mirata.

La funzione ausiliaria che implementa il sensore è descritta in dettaglio nella sezione A.7. Alleniamo gli agenti in seguito alle modifiche di comportamento adottate e valutiamone le *Reward Functions*. I risultati sono mostrati nelle Figure 3.4 e 3.5.

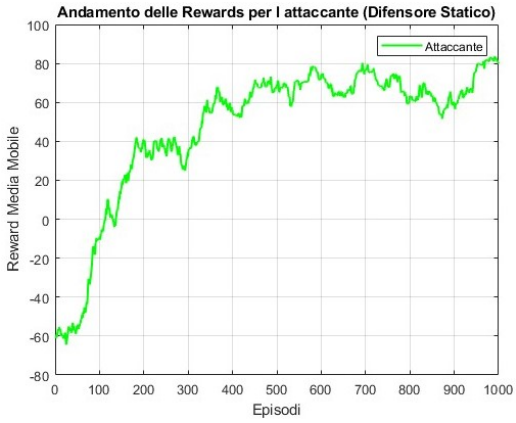

Figura 3.4. Apprendimento Attaccante con sensore.

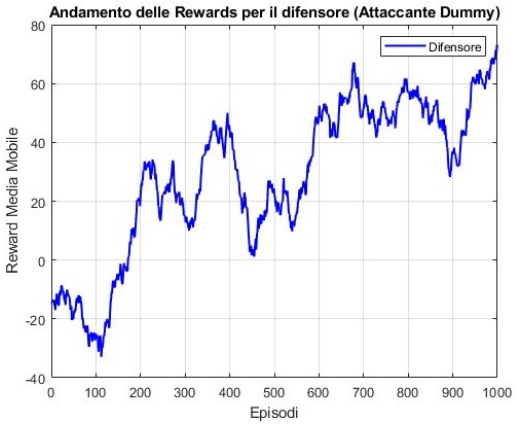

Figura 3.5. Apprendimento Difensore con sensore.

Dai grafici riscontriamo immediatamente come entrambi gli agenti abbiano raggiunto gli obiettivi con performance positive. Notiamo che l'Attaccante apprende sempre

leggermente meglio rispetto al Difensore. É complice il fatto che durante l'allenamento del Difensore l'Attaccante fantoccio colpisce in media più volte un ostacolo essendone più in prossimità nella mappa, questo fa terminare l'episodio e quindi l'apprendimento del Difensore viene interrotto e il punteggio inizializzato a zero. Il sensore permette: di detonare esclusivamente in maniera efficace all"Attaccante, consentendo a quest'ultimo di distruggere la casella goal, e al Difensore con una maggiore precisione di annientare l'Attaccante. L'unico caso in cui la difesa non distrugge con successo il nemico è quando quest'ultimo (il robot in attacco) sceglie un azione di movimento allontanandosi, mentre l'azione di deflagrazione è stata presa dal Difensore, rendendola inefficace. **N.B.** L'agente di difesa non può deflagrare in prossimità della propria base altrimenti danneggerebbe anche questa, quindi la sua politica lo porterà a pressare il nemico per evitare che questo si avvicini al goal.

La **politica e il percorso** sono altri strumenti decisamente utili per visualizzare cosa tramite Q-Learning gli agenti sono stati in grado di apprendere sono la visualizzazione della politica e il percorso (path) mostrati sull'ambiente di gioco. La policy ricordiamo che rappresenta il "*codice di comportamento*" che un agente ha consolidato in funzione di ogni stato che ha visitato. Non è altro che la traduzione della Q-Table rispetto alla mappa dove avvengono le simulazioni. Il path rappresenta il *percorso ottimo* che al termine dell'allenamento l'agente sfrutta per raggiungere con una probabilità maggiore il proprio obiettivo. Per capirne meglio l'utilità vediamo la politica e il percorso relativo alle ultime due *Reward Functions* mostrate poco fa (3.4, 3.5) rappresentate graficamente in 3.6, 3.7, 3.8 e 3.9.

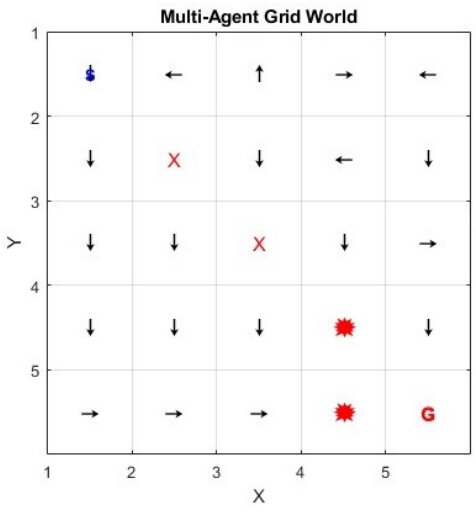

Figura 3.6. Policy Attaccante.

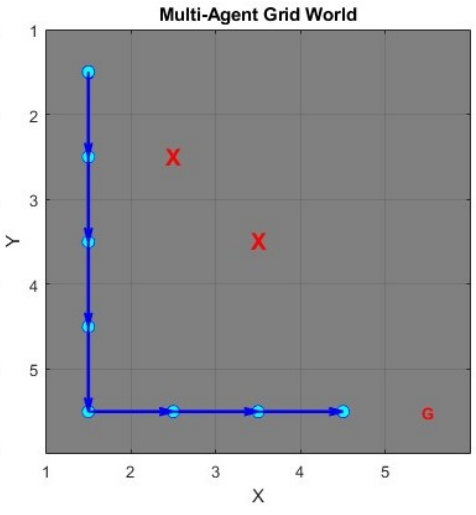

Figura 3.7. Path Attaccante.

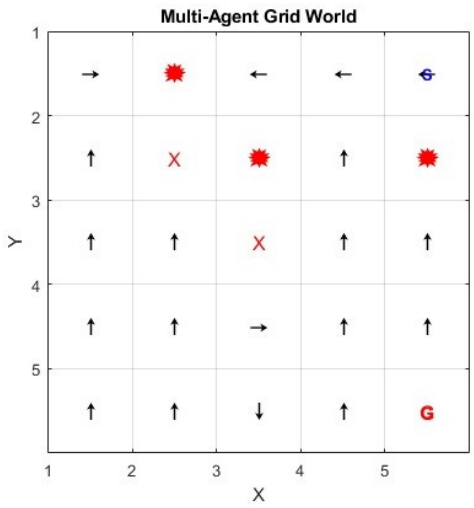

Figura 3.8. Policy Difensore.

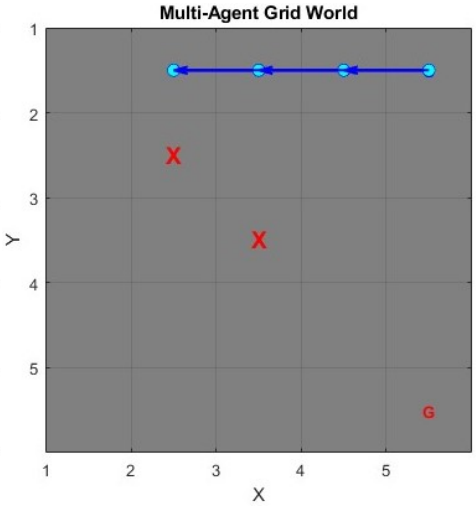

Figura 3.9. Path Difensore.

3.7 Robustezza

Nelle simulazioni analizzate fino ad ora le posizioni iniziali dei due agenti nell'*enviroment* erano fisse. Risulta però utile estendere lo studio anche ad altri casi iniziali.

Aggiungiamo una componente di aleatorietà rendendole casuali le celle di partenza dei due robot, in modo tale da valutare l'adattano delle strategie di ognuno. In particolar modo il Difensore patirà da una delle caselle adiacenti alla base $((4,4),(5,4),(4,5))$, mentre l'Attaccante da in una di quelle adiacenti al vertice in alto a sinistra$([1,1] \vee [1,2] \vee [2,1])$. Seguono i risultati in varie configurazioni iniziali. Cominciamo con gli allenamenti dell'Attaccante a partire dalla Figura 3.10 in poi:

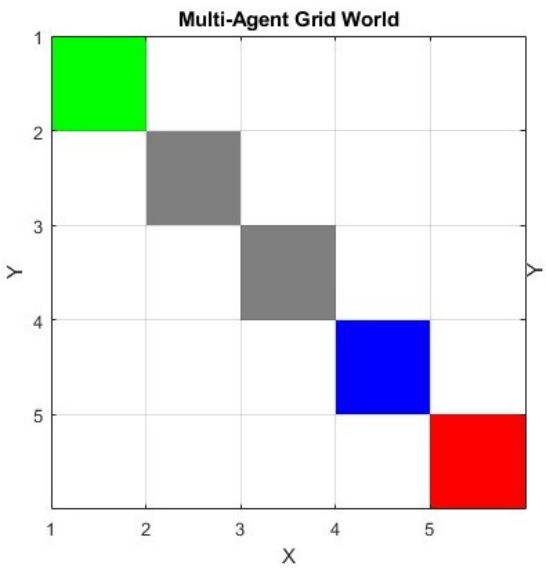

Figura 3.10. Stato i. Att.1.

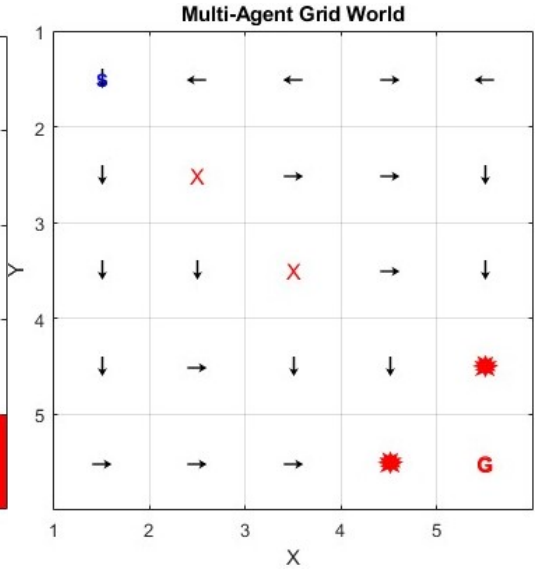

Figura 3.11. Policy Att.1.

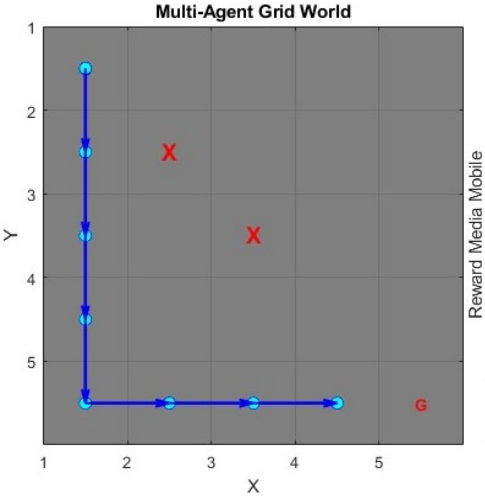

Figura 3.12. Path Att.1.

Figura 3.13. Reward Att.1.

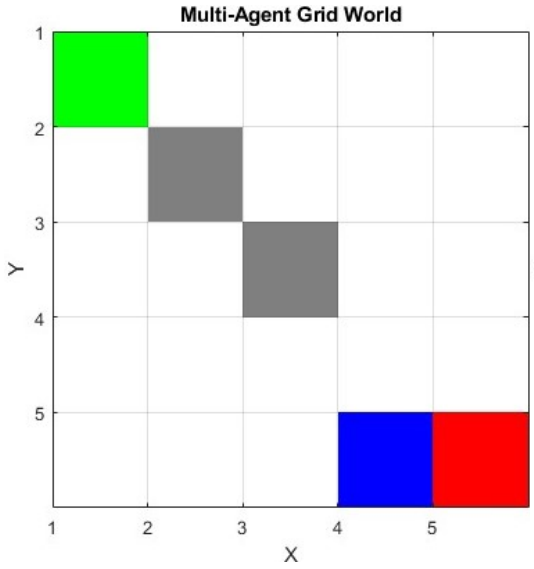

Figura 3.14. Stato i. Att.2..

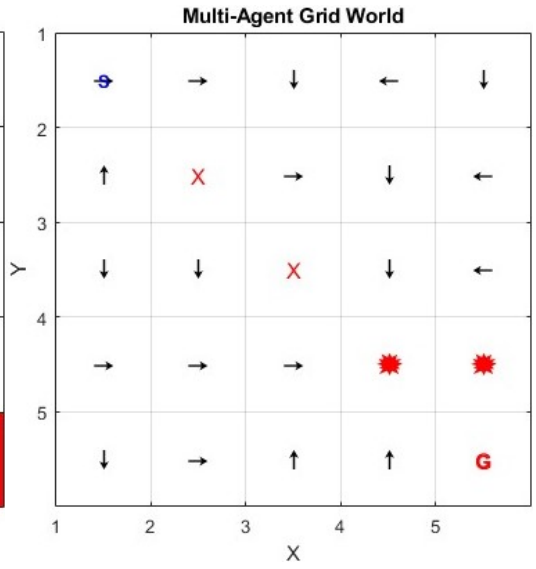

Figura 3.15. Policy Att.2.

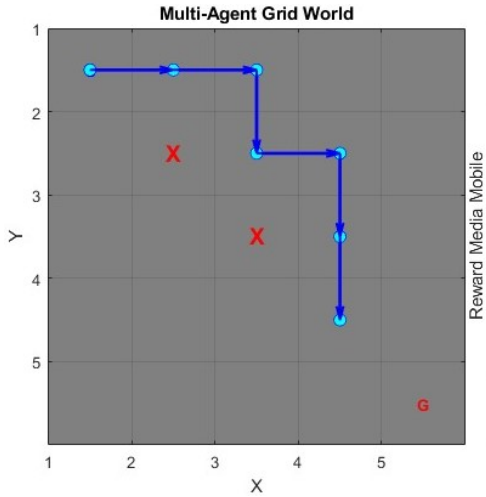

Figura 3.16. Path Att.2.

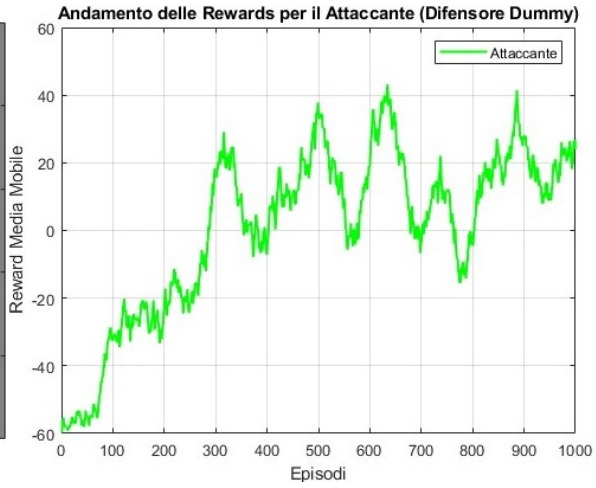

Figura 3.17. Reward Att.2.

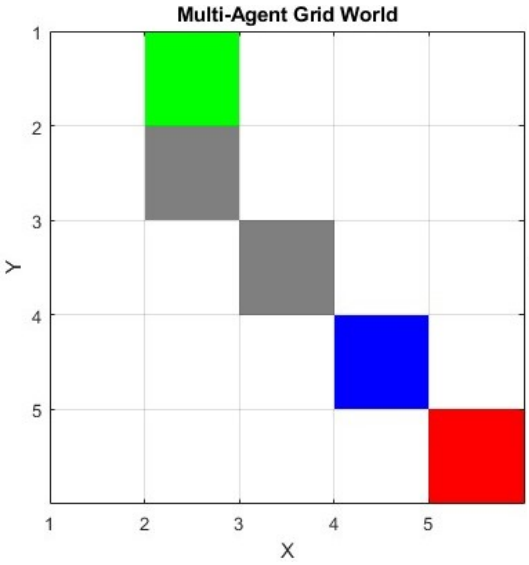

Figura 3.18. Stato i. Att.3..

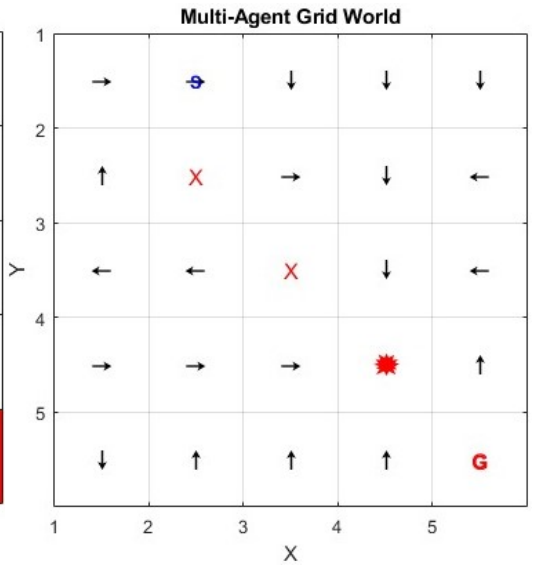

Figura 3.19. Policy Att.3.

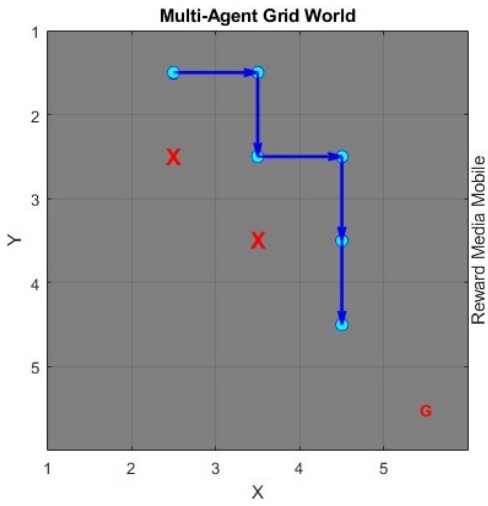

Figura 3.20. Path Att.3.

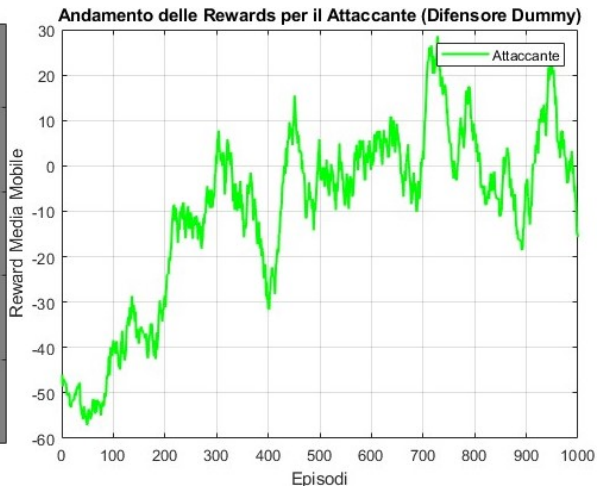

Figura 3.21. Reward Att.3.

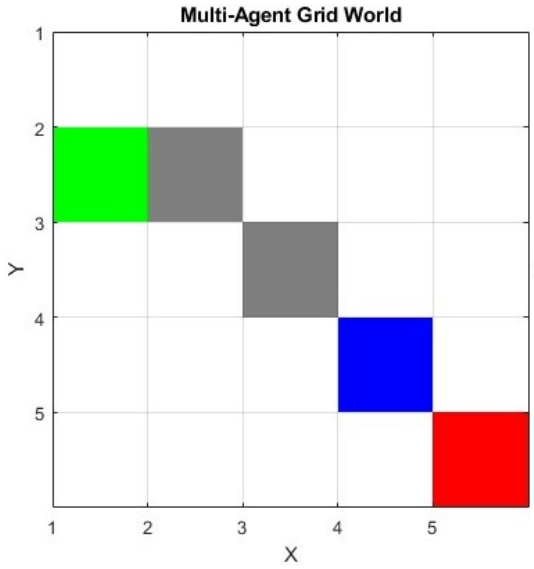

Figura 3.22. Stato i. Att.4..

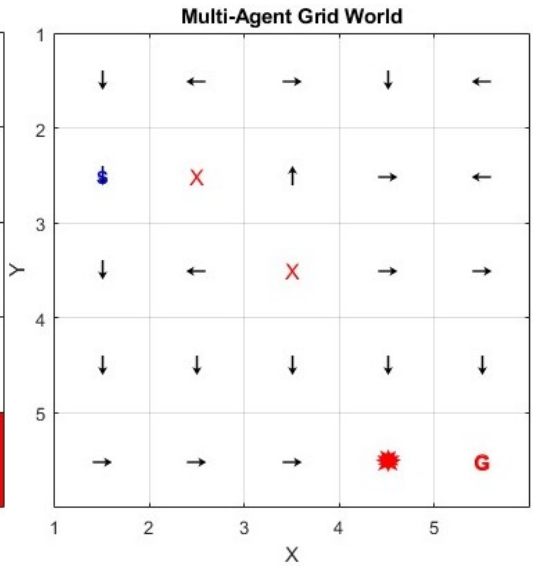

Figura 3.23. Policy Att.4.

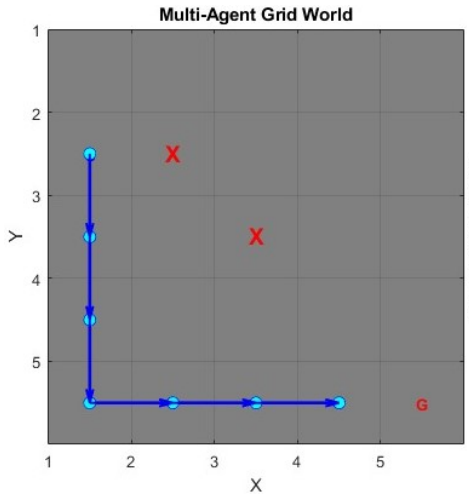

Figura 3.24. Path Att.4.

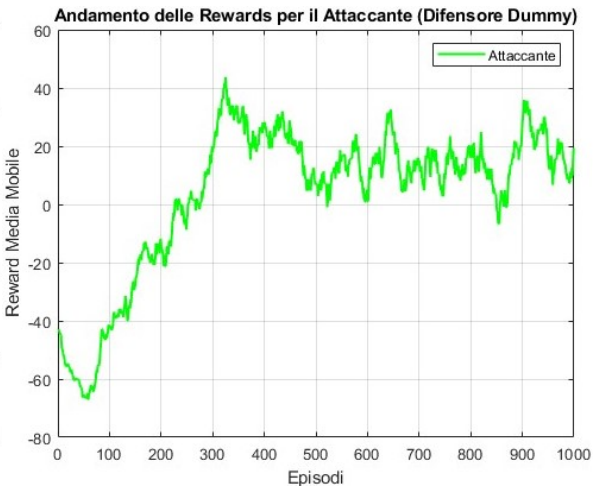

Figura 3.25. Reward Att.4.

Passiamo agli allenamenti del Difensore:

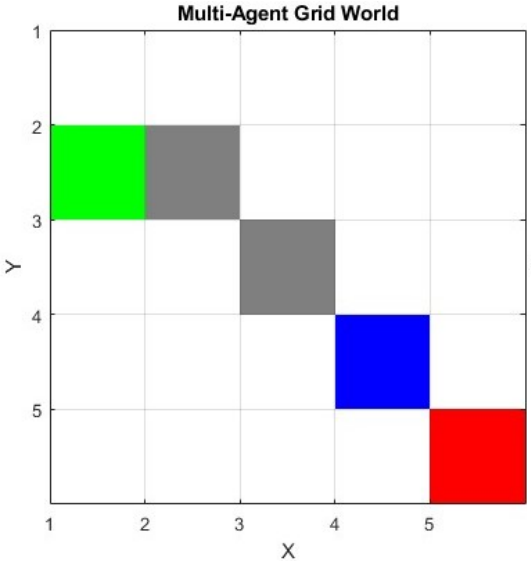

Figura 3.26. Stato i. Dif.1.

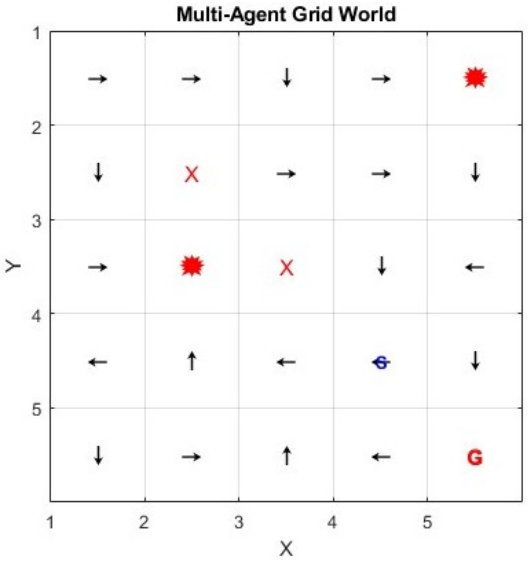

Figura 3.27. Policy Dif.1.

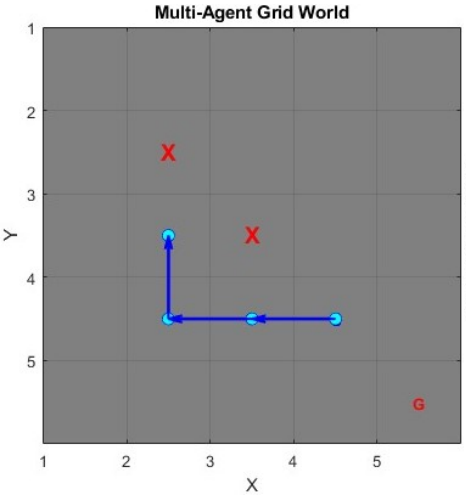

Figura 3.28. Path Dif.1.

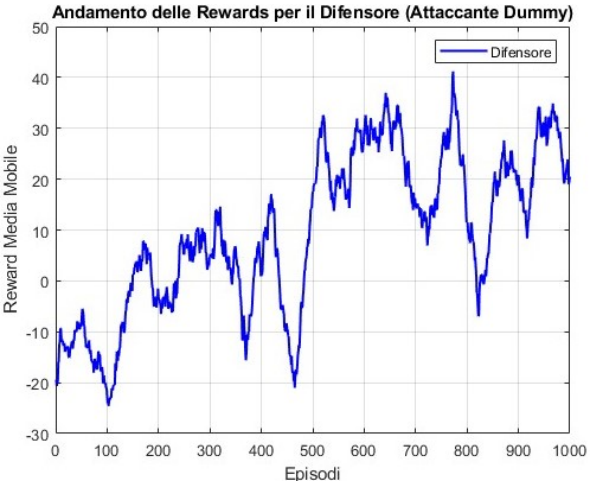

Figura 3.29. Reward Dif.1.

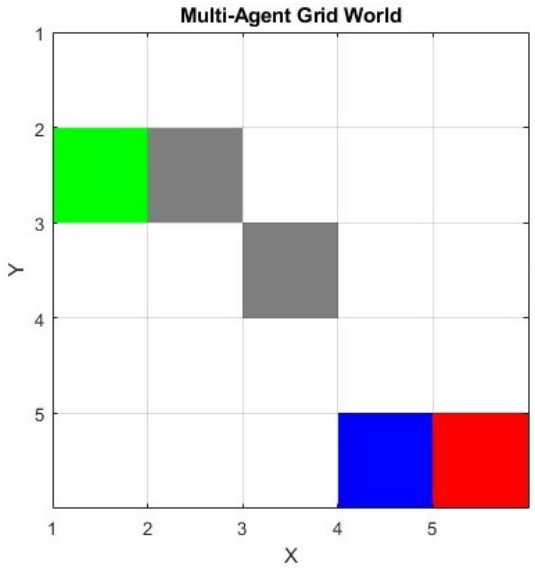

Figura 3.30. Stato i. Dif.2.

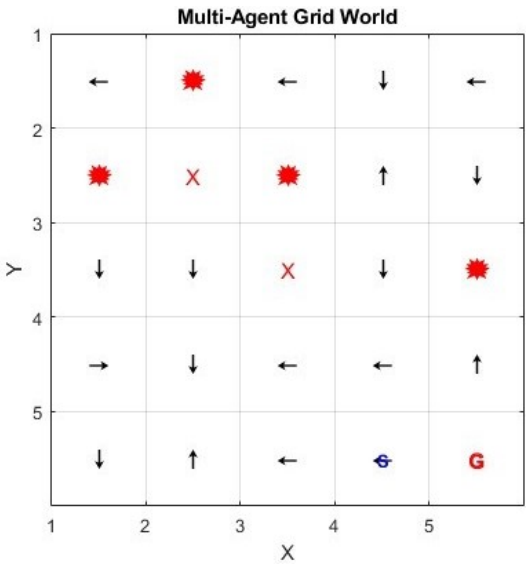

Figura 3.31. Policy Dif.2.

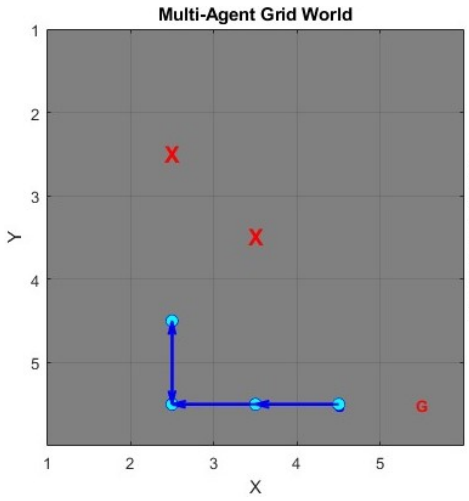

Figura 3.32. Path Dif.2.

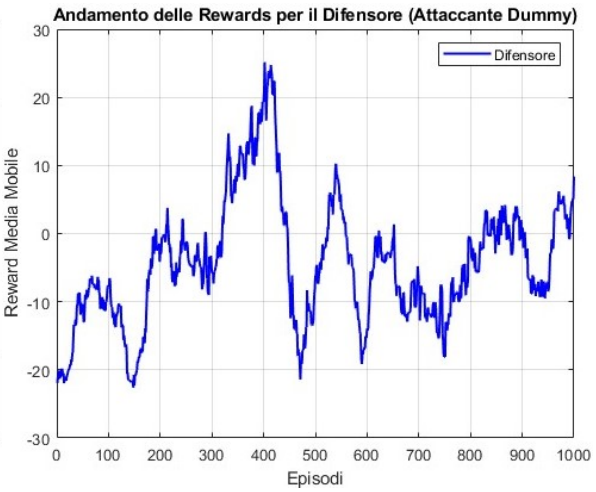

Figura 3.33. Reward Dif.2.

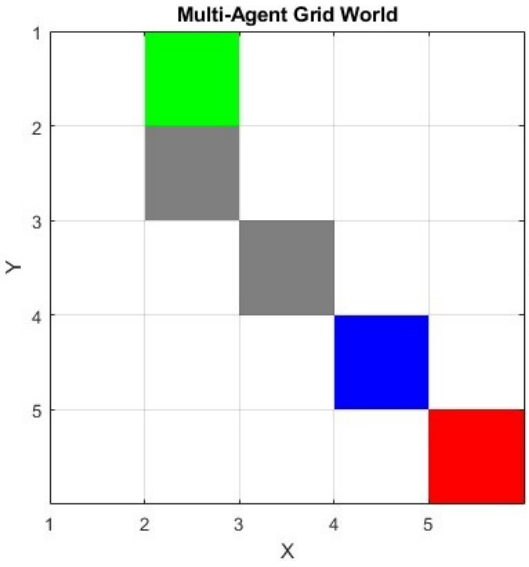

Figura 3.34. Stato i. Dif.3.

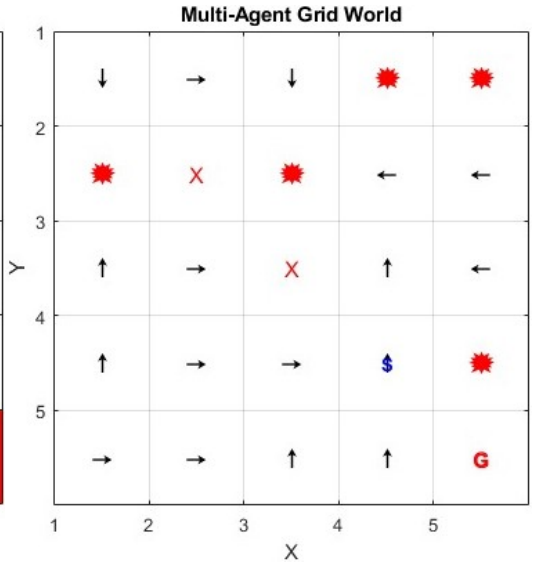

Figura 3.35. Policy Dif.3.

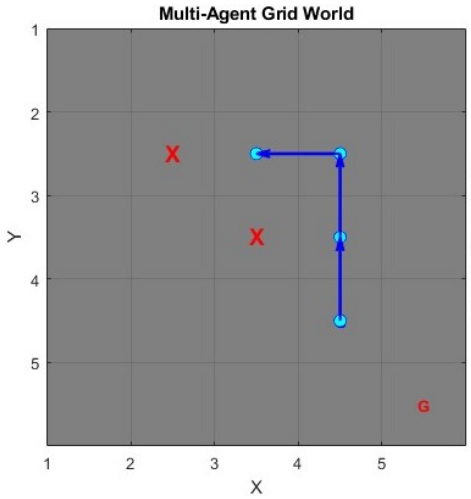

Figura 3.36. Path Dif.3.

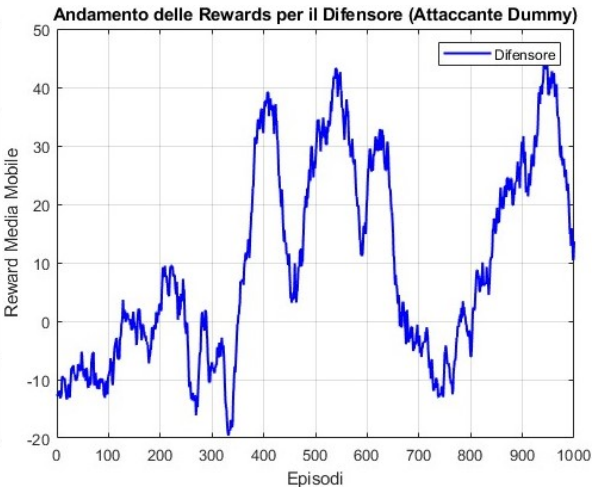

Figura 3.37. Reward Dif.3.

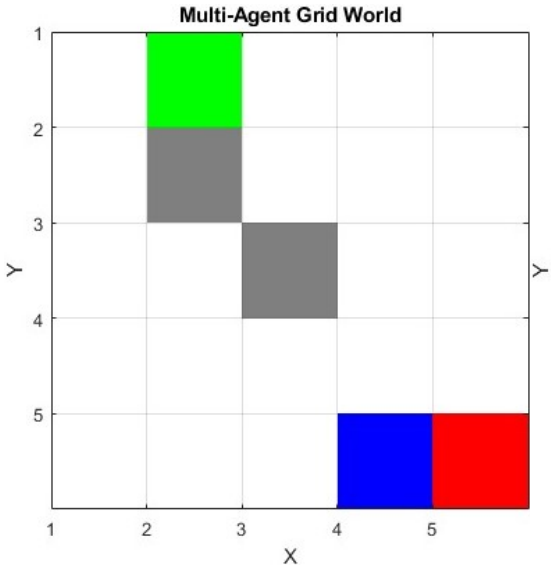

Figura 3.38. Stato i. Dif.4.

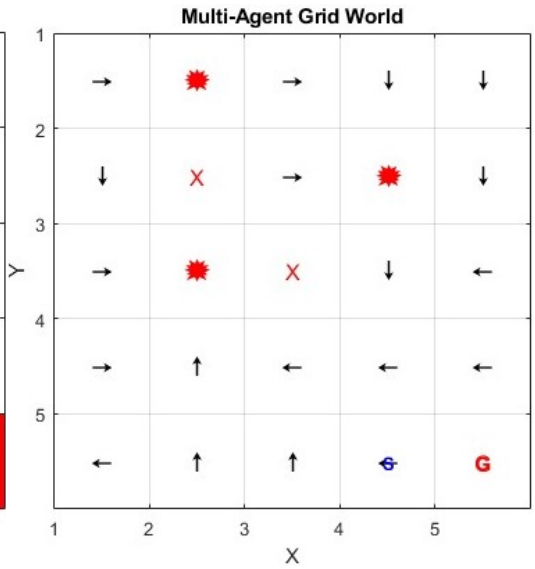

Figura 3.39. Policy Dif.4.

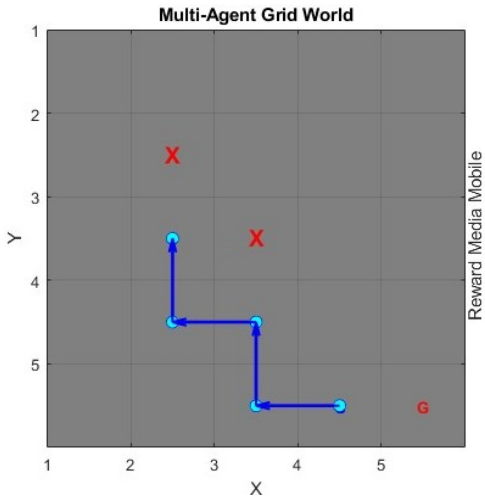

Figura 3.40. Path Dif.4.

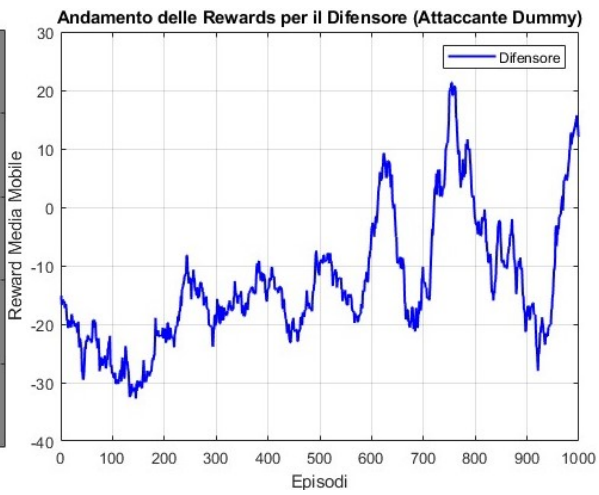

Figura 3.41. Reward Dif.4.

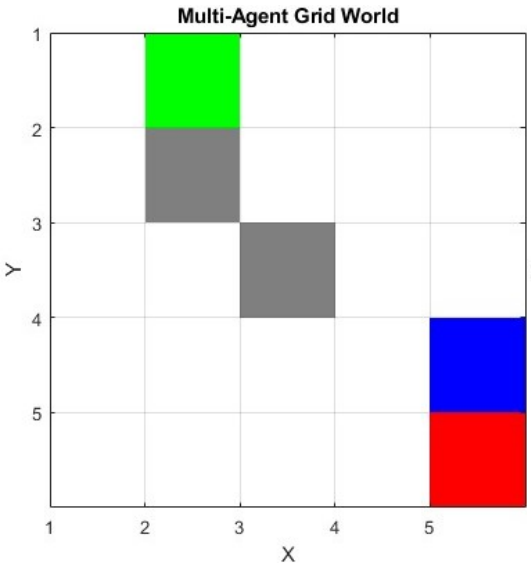

Figura 3.42. Stato i. Dif.5.

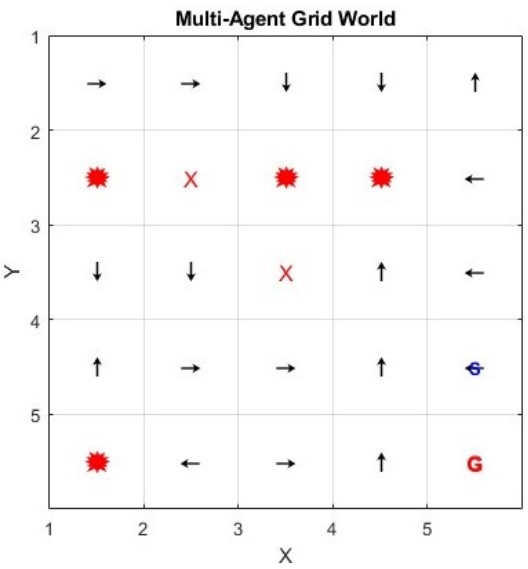

Figura 3.43. Policy Dif.5.

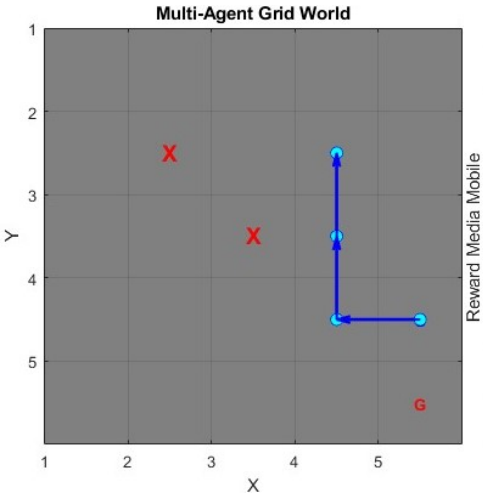

Figura 3.44. Path Dif.5.

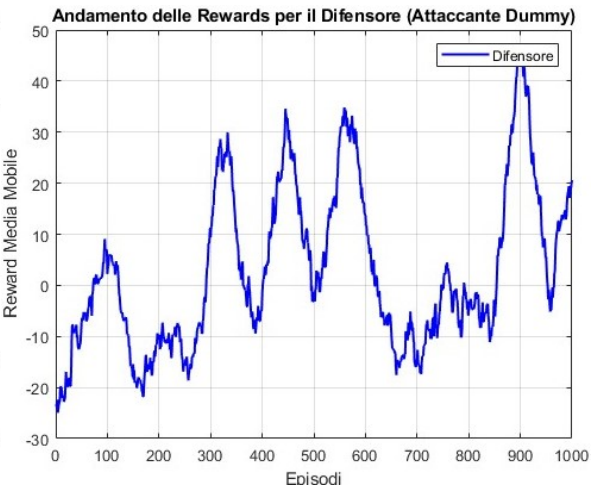

Figura 3.45. Reward Dif.5.

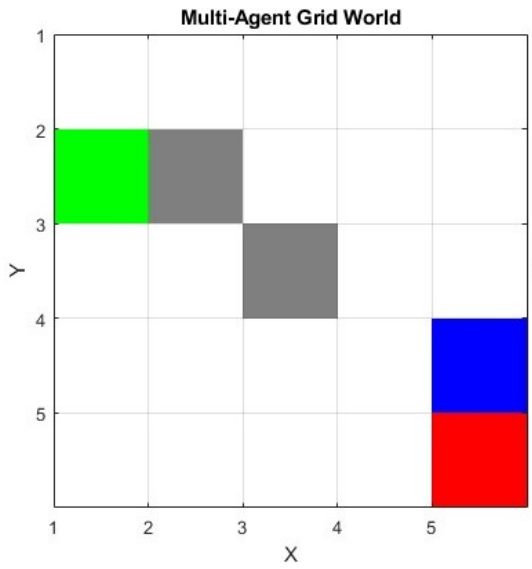

Figura 3.46. Stato i. Dif.6.

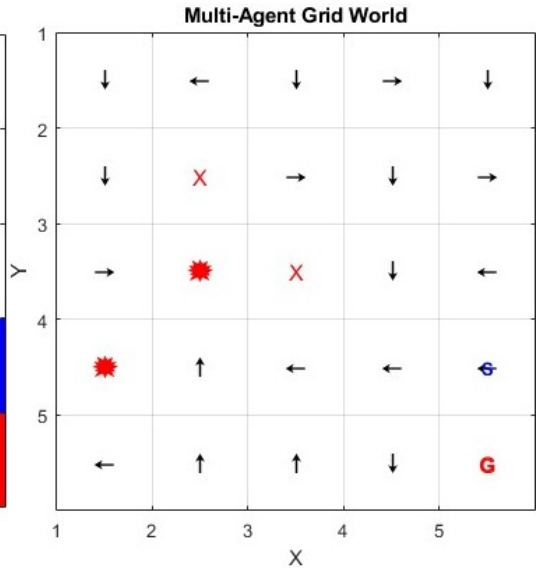

Figura 3.47. Policy Dif.6.

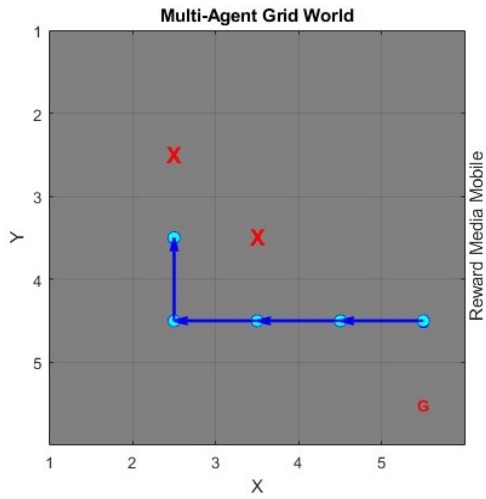

Figura 3.48. Path Dif.6.

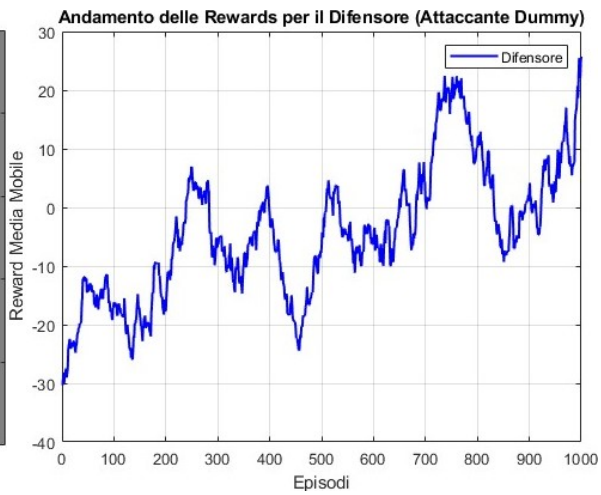

Figura 3.49. Reward Dif.6.

Capitolo 4

Conclusioni

Al termine delle simulazioni possiamo considerare di aver centrato l'obiettivo principale del progetto: testare l'efficacia del Q-Learning come algoritmo di apprendimento per strategie risultate ottimali per entrambe le fazioni Attacco e Difesa. Programmare ogni volta i passi da far eseguire ad un agente, cercando di prevedere le numerose situazioni dell'ambiente diventa naturalmente molto inefficiente; se abbiamo un caso come questo in cui l'enviroment possiede una componente dinamica (l'agente nemico) dal comportamento imprevedibile, diventa oneroso e sostanzialmente impossibile programmare a priori il comportamento del nostro robot in funzione dell'ambiente circostante e di come questo evolve. Nella teoria questo algoritmo di RL è stato ampiamente studiato e sviluppato, cattura il nostro interesse visto la grande varietà di contesti in cui è applicabile.

Nell'elaborato dopo aver fornito una panoramica solida, seppur per nulla esaustiva, della teoria e della matematica che sono dietro alla parte di programmazione e controllo, abbiamo verificato in prima persona l'efficacia del Q-Learning che ha permesso ad entrambi gli agenti di raggiungere i loro obiettivi nonostante la mappa ignota, gli ostacoli e l'avversario. Essi sono stati in grado di adattarsi alle diverse configurazioni di partenza convergendo nella quasi totalità delle simulazioni, verso una politica ottimale con più o meno rapidità ed incertezza. Quindi è stato tracciato un percorso che ha massimizzato il punteggio ottenuto. Naturalmente è possibile variare la velocità di convergenza verso la soluzione ottima modificando il valore dei parametri dell'algoritmo, così come facilitare la convergenza stessa in un ambiente privo di ostacoli o assegnando malus negativi maggiori in caso di errore da parte del robot. Sono pertanto molteplici le alternative progettuali valide che si possono testare. Il codice è estendibile ad ambienti più vasti con ostacoli di vario genere e a tutti i dettagli del caso. Il Q-Learning è sfruttabile in infinite possibili mappe, risulta quindi robusto ed estremamente adattabile a qualsiasi ambiente. Infine lo sfruttamento attivo dei sensori ha permesso ai robot di unire l'apprendimento per rinforzo con una

scelta mirata delle azioni, prese in funzione degli input immagazzinati dal sensore.

Questo elaborato non pretende di essere esaustivo, ma intende contribuire alla comprensione di una famiglia di problemi di navigazione, suggerendo possibili spunti per futuri approfondimenti personali della materia. Si può fare ad esempio riferimento ad un lavoro recente [19] redatto dal Dipartimento di Automatica dell' Università di Roma La Sapienza. In questo articolo viene proposta una valida e originale soluzione a problemi di navigazione in un contesto multi-agente, nel quale l'utilizzo dei sensori di rilevamento è limitato e gli agenti si affidano alle informazioni locali. Viene dimostrato l'apprendimento tramite RL del percorso ottimo che fa giungere a destinazione il robot, inoltre quest'ultimo impara in quali circostanze i sensori possono essere spenti per preservare energia e diminuire il consumo di batteria. Si sfrutta un protocollo di comunicazione limitata di interesse in scenari che riguardano aree ad alto rischio esplosivo o di sorveglianza. Il metodo è stato testato tramite simulazioni e confrontato con algoritmi esistenti nella letteratura, come quello in [20], che presuppone una condivisione costante delle informazioni di posizione. L'articolo dimostra un'efficacia superiore dell'algoritmo proposto, con il vantaggio di un minor consumo di risorse. Tale proposta di un modello MARL decentralizzato, in cui i robot apprendono a navigare con interazioni minime, è un approccio utile in scenari critici come il monitoraggio ambientale o il soccorso in aree pericolose. Sono molteplici le sfide moderne che per essere risolte dovranno coniugare le conoscenze della robotica, dell'automatica e dell'informatica, affrontarle ci permetterà di elevare la conoscenza a fondamento di un progresso finalizzato al benessere dell'Umanità e alla ricerca della virtù.

Appendice A

Codice del progetto

A.1 Mappa di gioco

Definiamo per prima cosa l'ambiente di gioco (sfondo bianco), una matrice 5x5 e la posizione degli ostacoli fissi (grigio). Inizializziamo le caselle start per l'Attaccante (verde), per il Difensore (blu) e la base/goal (rosso). Assegniamo poi il valore ai parametri della formula di Q-Learning prendendo spunto da questa risorsa [21].

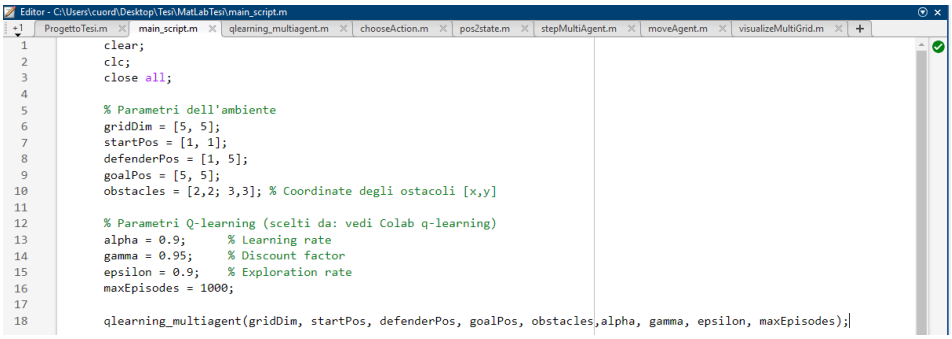

Figura A.1. Main.

A.2 Q-Learning

A partire dalla formulazione matematica e dalla seguente implementazione [21] in linguaggio Python siamo passati al codice MatLab dell'algoritmo di Q-Learning. All'interno della funzione principale vengono richiamate diverse altre funzioni ausiliarie che supportano la realizzazione della simulazione.

```
%Implemento la funzione Q-learning
function qlearning_multiagent(gridDim, startPos, defenderPos, goalPos, obstacles,alpha, gamma, epsilon, maxEpisodes)
 % Inizializzazione Q-table
    numStates = gridDim(1) * gridDim(2);
    numActionsAttacker = 4; %su, giù, sinistra, destra
    numActionsDefender = 5;  % su, giù, sinistra, destra, detonazione
    Q_attacker = zeros(numStates, numActionsAttacker);
    Q_defender = zeros(numStates, numActionsDefender);
    for episode = 1:maxEpisodes
        currentPosAttacker = startPos;
        currentPosDefender = defenderPos;
        done = false;
        totalRewardAttacker = 0;
        totalRewardDefender = 0;

        while ~done
            % Stato corrente per entrambi gli agenti
            currentStateAttacker = pos2state(currentPosAttacker, gridDim);
            currentStateDefender = pos2state(currentPosDefender, gridDim);

            % Scelgo le azioni per entrambi gli agenti (epsilon-greedy)
            actionAttacker = chooseAction(Q_attacker, currentStateAttacker, epsilon);
            actionDefender = chooseAction(Q_defender, currentStateDefender, epsilon);

            % Eseguo le azioni e ottengo nuovi stati e ricompense
            [nextPosAttacker, nextPosDefender, rewardAttacker, rewardDefender, done] = ...
                stepMultiAgent(currentPosAttacker, currentPosDefender, actionAttacker, actionDefender, gridDim, goalPos, obstacles);
```

Figura A.2. Q-Learning MatLab prima parte.

```
            % Calcolo nuovi stati
            nextStateAttacker = pos2state(nextPosAttacker, gridDim);
            nextStateDefender = pos2state(nextPosDefender, gridDim);

            % Aggiorno Q-tables
            Q_attacker(currentStateAttacker, actionAttacker) = Q_attacker(currentStateAttacker, actionAttacker) + ...
                alpha * (rewardAttacker + gamma * max(Q_attacker(nextStateAttacker, :)) - Q_attacker(currentStateAttacker, actionAttacker));

            Q_defender(currentStateDefender, actionDefender) = Q_defender(currentStateDefender, actionDefender) + ...
                alpha * (rewardDefender + gamma * max(Q_defender(nextStateDefender, :)) - Q_defender(currentStateDefender, actionDefender));

            % Aggiorno posizioni
            currentPosAttacker = nextPosAttacker;
            currentPosDefender = nextPosDefender;

            totalRewardAttacker = totalRewardAttacker + rewardAttacker;
            totalRewardDefender = totalRewardDefender + rewardDefender;
        end
        %Ogni 100 episodi stampo le info per attaccante e difensore
        if mod(episode, 100) == 0
            fprintf('Episodio %d, Reward Attaccante: %.2f, Reward Difensore: %.2f\n', ...
                episode, totalRewardAttacker, totalRewardDefender);
        end
    end
end
```

Figura A.3. Q-Learning MatLab continuo.

A.3 ChooseAction

La funzione *chooseAction* implementa esattamente $\epsilon - greedy$. La politica bilancia la scelta tra esplorazione ed sfruttamento. Un valore equilibrato di ϵ permette di:

- Esplorare nuove zone della mappa, prendendo azioni casuali e scoprendo se ve ne sono di migliori che può prendere in determinati stati;
- Sfruttare la Q-Table e quindi scegliere nuovamente azioni che ha già eseguito in determinati stati e che si sono dimostrate efficaci nell'ottenere ricompense positive.

Generiamo un numero casuale r, compreso fra 0 e 1, avremo che con una certa probabilità l'agente esplori (se r$<\epsilon$) e una probabilità $1-\epsilon$ che invece scelga l'azione sfruttando il maggior valore $Q(s,a)$ appartenente alla Q-Table. Una buona

Policy fa sì che il valore di ϵ sia alto nei primi episodi in modo tale da premiare l'esplorazione nelle fasi iniziali e decresca nelle fasi più avanzate della simulazione così da ricompensare lo sfruttamento della conoscenza acquisita. Per questo motivo, nel progetto, ad ogni episodio ϵ viene decrementato linearmente seguendo questa espressione:

$$\epsilon = max(0.1, \epsilon \times 0.995) \tag{A.1}$$

In questo modo decresce ad ogni ciclo del programma, lasciando come *lower bound* (limite inferiore) il valore "0.1"; tale scelta diminuisce le situazioni di stallo, cioè quando l'agente tende a restare nella stessa casella per molti turni rischiando di non muoversi più da quella posizione.

Il codice della sola politica è il seguente. Una versione leggermente più com-

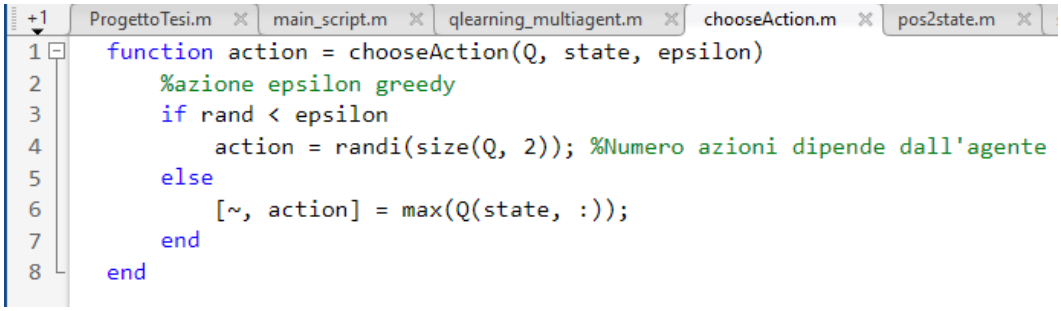

Figura A.4. Implementazione $\epsilon - greedy$.

plessa della funzione *chooseAction* gestisce la scelta delle azioni, in particolar modo implementa in base a quale agente si sta allenando l'utilizzo o meno della politica $\epsilon - greedy$. Ricordiamo infatti che gli allenamenti nelle simulazioni avvengono solo per un agente alla volta, l'altro denominato "fantoccio" sarà obbligato a prendere azioni casuali durante ogni episodio dell'allenamento.

In questa versione la prima parte del codice gestisce le strategie di inseguimento del Difensore dovute al rilevamento dell'avversario da parte dei sensori. Viene scelta razionalmente l'azione lungo la direzione con la quale c'è meno distanza dall'avversario. In alternativa se non vi è alcuna preferenza si opta per una mossa casuale tra quelle di movimento.

```
function action = chooseAction(Q, state, epsilon, isDefender, attackerDetected, defenderPos, attackerPos)
    if isDefender && attackerDetected
        % Calcola la differenza tra le coordinate
        deltaX = attackerPos(1) - defenderPos(1); % Differenza asse X
        deltaY = attackerPos(2) - defenderPos(2); % Differenza asse Y
        % Strategia di avvicinamento basata sulla distanza
        if abs(deltaX) > abs(deltaY)
            if deltaX > 0
                action = 2; % Giù
            else
                action = 1; % Su
            end
        elseif abs(deltaY) > abs(deltaX)
            if deltaY > 0
                action = 4; % Destra
            else
                action = 3; % Sinistra
            end
        else
            % Se le distanze sono uguali, scegli casualmente tra X e Y
            if rand > 0.5
                if deltaX > 0
                    action = 2; % Giù
                else
                    action = 1; % Su
                end
            else
                if deltaY > 0
                    action = 4; % Destra
                else
                    action = 3; % Sinistra
                end
            end
        end
    else
        %azione epsilon greedy
        if rand < epsilon
            action = randi(size(Q, 2)); %Numero azioni dipende dall'agente, il 2
            %indica che sta prendendo le colonne(secondo parametri della matrice)
        else
            [~, action] = max(Q(state, :));
        end
    end
end
```

Figura A.5. chooseAction sensori MatLab.

A.4 Pos2state

Per tradurre in tempo reale (ogni *time-step*) la posizione dell'agente nel grid-world in uno stato utile per calcolare la Q-Table, dobbiamo mappare la coordinata spaziale $[x, y]$ in un numero da 1 a 25 ($5x5$). Per scelta di progetto procediamo per righe, quindi alla posizione $[1, 1]$ corrisponderà lo stato $s1$, mentre alla posizione $[1, 2]$ lo stato $s6$. L'equazione che seguiamo per la mappatura è la seguente:

$$s_i = (y - 1) \times N + x \tag{A.2}$$

Con N numero di righe/colonne.

```
+2  ProgettoTesi.m  main_script.m  qlearning_multiagent.m  choo
1   function state = pos2state(pos, gridDim)
2       state = (pos(1) - 1) * gridDim(2) + pos(2);
3   end
```

Figura A.6. Pos2State MatLab.

A.5 StepMultiAgent

Una volta che l'agente sceglie quale azione prendere, dovremo aggiornare la posizione sulla mappa, incrementare o decrementare la *reward* dell'agente e qualora si dovesse terminare l'episodio impostare la variabile *booleana done* = *true*. I casi che portano all'interruzione dell'episodio sono:

- Il Difensore detona distruggendo l'Attaccante;
- L'Attaccante detona distruggendo la base;
- Uno dei due agenti si scontra contro un ostacolo.

Il tutto viene gestito tramite *stepMultiAgent()*, in questa versione si fa riferimento al caso in cui sono implementati i sensori.

```
% Funzione per l'esecuzione di un'azione
function [nextPosAttacker, nextPosDefender, rewardAttacker, rewardDefender, done] = ...
    stepMultiAgent(currentPosAttacker, currentPosDefender, actionAttacker, actionDefender, gridDim, goalPos, obstacles,detectionRange)
    % Movimento normale dell'agente in attacco
    nextPosAttacker = moveAgent(currentPosAttacker, actionAttacker, gridDim);

    % Controllo sensore: il difensore rileva l'attaccante?
    attackerDetected = detectAttacker(currentPosDefender, currentPosAttacker, detectionRange);
    % Verifico se attaccante è nel raggio di azione per la detonazione
    if attackerDetected
        if actionDefender == 5
            rewardDefender = 100;  % Ricompensa elevata per il difensore
            rewardAttacker = -100;  % Penalità severa per l'attaccante
            nextPosDefender = currentPosDefender; %difensore fermo perchè è detonato
            done = true;
            return
        else
            % Movimento normale del difensore se non esegue detonazione, ma ha comunque sentito l'attaccante
            nextPosDefender = moveAgent(currentPosDefender, actionDefender, gridDim);
        end
    elseif actionDefender == 5
            %scoraggio la detonazione inefficace
            rewardDefender = -10;
            nextPosDefender = currentPosDefender;
    else
        %nel caso non abbia sentito l'attaccante prende una decisione di movimento
        nextPosDefender = moveAgent(currentPosDefender, actionDefender, gridDim);
    end
```

Figura A.7. StepMultiAgent prima parte.

```
        % Verifico collisioni con ostacoli
        if any(all(nextPosAttacker == obstacles, 2))
            nextPosAttacker = currentPosAttacker;
            rewardAttacker = -10;
        elseif any(all(nextPosDefender == obstacles, 2))
            nextPosDefender = currentPosDefender;
            rewardDefender = -10;
        end
        % Verifico raggiungimento obiettivo
        if all(nextPosAttacker == goalPos)
            rewardAttacker = 100;
            rewardDefender = -100;
            done = true;
            return;
        end
        % Ricompense di default per movimento
        rewardAttacker = -1;
        rewardDefender = -1;
        done = false;
    end
```

Figura A.8. StepMultiAgent continuo.

A.6 MoveAgent

Questa funzione gestisce nella pratica l'aggiornamento della posizione sulla mappa dei due agenti. Partendo dalla *currentPos*, presa un'azione, l'agente dovrà muoversi all'interno dell'ambiente, il compito di questa funzione è esattamente questo: tradurre l'azione in uno spostamento. L'implementazione è la seguente:

```
function nextPos = moveAgent(currentPos, action, gridDim)
    nextPos = currentPos;
    switch action
        case 1  % su
            nextPos(1) = max(1, currentPos(1) - 1);
        case 2  % giù
            nextPos(1) = min(gridDim(1), currentPos(1) + 1);
        case 3  % sinistra
            nextPos(2) = max(1, currentPos(2) - 1);
        case 4  % destra
            nextPos(2) = min(gridDim(2), currentPos(2) + 1);
    end
end
```

Figura A.9. MoveAgent MatLab.

A.7 Sensoristica

Il sensore di prossimità è simulato da una funzione che controlla in un dato raggio in input se vi sono oggetti che ricadono all'interno. La distanza per il rilevamento è calcolata in maniera euclidea essendo l'enviroment bidimensionale e piano, questo permette di tenere all'interno della soglia del *detectionRange*, raggio di rilevamento (pari a 1.5), l'eventuale posizione dell'oggetto rilevato in una cella diagonale. Con

un banale calcolo tramite il teorema di Pitagora, notiamo che il valore massimo nel caso diagonale è pari a $sqrt(2) = 1.4142$ (su MatLab). Quindi strettamente minore del *detectionRange*. Il risultato è il seguente.

```
function detected = detectAttacker(defenderPos, attackerPos, detectionRange)
    % Calcolo la distanza tra il difensore e l'attaccante
    distance = abs(defenderPos(1) - attackerPos(1)) + abs(defenderPos(2) - attackerPos(2));

    % Se la distanza è minore o uguale alla soglia rilevo l'attaccante
    detected = distance <= detectionRange;
end
```

Figura A.10. Funzione di Rilevamento in MatLab.

Bibliografia

[1] IBM, “What is machine learning,”

[2] Adobe, “Machine learning — definition, application, and methods,”

[3] R. S. Sutton and A. G. Barto, *Reinforcement Learning An Introduction*. MIT Press, 2014-2015.

[4] P. Gray, *Psicologia*. Zanichelli, 2012.

[5] E. L. Thorndike, “Animal intelligence: An experimental study of the associative processes in animals. psychological monographs: General and applied, 2(4), i-109.,” 1898.

[6] B. Skinner, “The behavior of organisms: An experimental analysis. new york: Appleton-century,” 1938.

[7] S. Russel and P. Norvig, *Artificial Intelligence, A Modern Approach (4th Edition)*. Pearson Global Edition, 2021.

[8] O. AI, “Part 1: Key concept in rl,” 2018.

[9] D. S. R. course and S. CS234, “Finite markov decision processes,cmps 4660/6660: Reinforcement learning,”

[10] N. Turcato, *Model-Based Reinforcement Learning for industrial robotics applications*. Università degli studi di Padova, 2022.

[11] S. Russel and P. Norvig, *CAP. 17 Artificial Intelligence, A Modern Approach (4th Edition)*. Pearson Global Edition, 2021.

[12] C. Watkins and P. Dayan, “Q-learning,” 1992.

[13] D. Technologies, “Q-playing atari with deep reinforcement learning,” 2013.

[14] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, and A. Bolton, “Mastering the game of go without human knowledge,” 2017.

[15] T. Ribeiro, F. Gonçalves, I. Garcia, G. Lopes, and A. F. Ribeiro, "Q-learning for autonomous mobile robot obstacle avoidance," 2019.

[16] M. G. Mohanan and A. Salgoankar, "A survey of robotic motion planning in dynamic environments," robotics and autonomous systems.," 2018.

[17] T. Alpcan and T. Başar, *Network Security:A Decision and Game Theoretic Approach*. Cambridge University Press, 2010.

[18] A. Cristofaro and V. Suraci, *Fondamenti di Data Processing*.

[19] D. Menegatti, A. Giuseppi, and A. Pietrabissa, "Distributed marl with limited sensing for robot navigation problems," 2023.

[20] C. Yu, M. Zhang, Ren, and G. Tan, "Multi-agent learning of coordination in loosely coupled multi-agent systems," 2015.

[21] "Q-learning python implementation." `https://colab.research.google.com/github/d2l-ai/d2l-pytorch-colab/blob/master/chapter_reinforcement-learning/qlearning.ipynb#scrollTo=674999d4&uniqifier=1`.