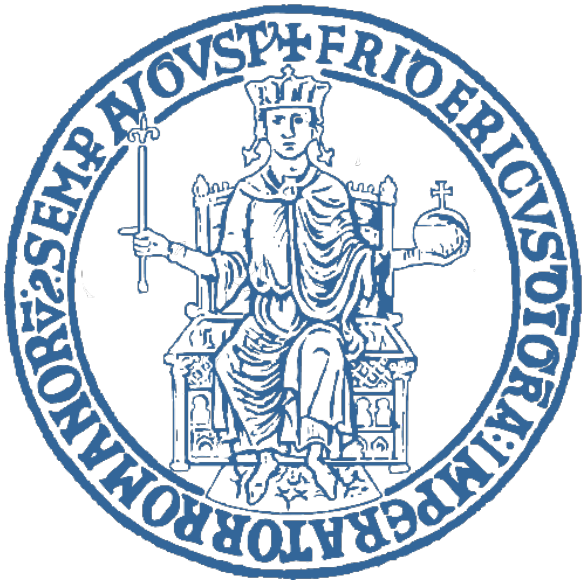
UNIVERSITA' DEGLI STUDI DI NAPOLI FEDERICO II



Scuola Politecnica e delle Scienze di Base

Dipartimento di Ingegneria Elettrica e Tecnologie dell'Informazione

PROGETTO DI SISTEMI PER IL GOVERNO DEI ROBOT

UN APPROCCIO BASATO SU Q-LEARNING PER LA RISOLUZIONE DI UN PROBLEMA DI GOALS E SUB-GOALS IN UN AMBIENTE DISCRETO E MULTIAGENTE

Studenti:

Paolo Domenico Lambiase N97000267

Davide Trepiccione N97000285

Contents

Chapter 1

Traccia

Si realizzi un algoritmo che, in ambiente simulato, risolva, tramite Reinforcement Learning, un problema di navigazione di due agenti, i quali devono, concorrentemente, all'interno di una mappa discreta, soddisfare due goal :

- Eseguire un task di navigazione intermedio, differente per ogni agente.
- Eseguire un task di navigazione finale, comune ad entrambi gli agenti.

In uno scenario reale si può considerare questo problema come quello vari robot che operano in una fabbrica/azienda, i quali hanno come obiettivo quello di prelevare dei materiali/kit di un qualsiasi tipo, (ogni robot è incaricato di prelevare un determinato oggetto), e consegnarli ad un banco di lavoro/macchinario.

Chapter 2

Introduzione e stato dell'arte

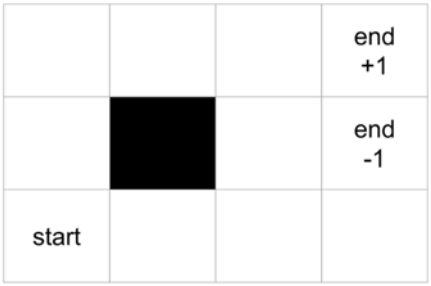
2.1 Introduzione su Reinforcement Learning

Il Machine learning è un sottoinsieme dell'intelligenza artificiale che fornisce alle macchine la capacità di apprendere automaticamente senza essere esplicitamente programmate. L'apprendimento per rinforzo (Reinforcement Learning [1]) è una tecnica di Machine Learning che consiste nell'intraprendere azioni adeguate per massimizzare la ricompensa (reward) in una particolare situazione. Viene impiegato tipicamente per trovare il miglior comportamento possibile o il miglior percorso che dovrebbe essere intrapreso in una situazione specifica. In altre parole si può dire che con il Reinforcement Learning si tenta di stabilire o incoraggiare un modello di comportamento.

2.2 Domini applicativi

Uno dei domini più studiati nella storia dell'Intelligenza Artificiale è quello dei giochi, con l'obiettivo di sviluppare algoritmi in grado di raggiungere capacità sovrumane nella risoluzione di tali giochi. Tra i più mirati, troviamo i giochi di Scacchi, Go e l'Atari 2600. In questo contesto, il Reinforcement Learning ha acquisito un ruolo molto significativo per l'implementazione di agenti in grado di apprendere questi compiti complessi. Nel 2013, i ricercatori di DeepMind hanno presentato un algoritmo di Deep Learning capace di giocare sette giochi Atari 2600 dell' Arcade Learning Environment, superando tutti gli approcci precedenti su sei dei giochi e facendo meglio di un esperto umano su tre di essi [2] L'algoritmo di RL è implementato con una Rete Neurale Deep, con due strati Convoluzionali e due strati densi completamente connessi, con un unico output per ogni accesso valido, che può dipendere dai giochi considerati. L'algoritmo ottenuto con questa architettura è denominato Deep Q-Learning (Poichè si stima la Q-Funciton con reti deep) Questo approccio è stato ulteriormente studiato e sviluppato dai ricercatori di DeepMind, che hanno dimostrato che il loro algoritmo DQL [3] può imparare a giocare e raggiungere prestazioni paragonabili a quelle di un tester professionale di giochi umani su un set di 49 giochi Atari 2600, utilizzando lo stesso modello e gli stessi parametri. Questa versione migliorata dell'algoritmo riceve in input solo l'immagine grezza dello stato del gioco e il punteggio del gioco. Un'altra delle applicazioni più rilevanti e di successo del Reinforcement Learning è l'algoritmo AlphaGo, che è stato sviluppato anche dal team DeepMind, ed è stato in grado di sconfiggere il campione umano di Go europeo per 5 a 0, al gioco del Go [4]. Questo gioco è uno dei giochi più difficili e impegnativi per le AI. L'approccio proposto utilizza una rete per valutare le posizioni ed altre reti per selezionare le mosse. Una combinazione di apprendimento supervisionato da giochi di esperti umani, e l'apprendimento per rinforzo da giochi competitivi è impiegato per allenare queste reti. Inoltre, viene presentato un nuovo algoritmo di ricerca, basato sulla ricerca ad albero di Montecarlo, reti di valori e reti di scelte, che consente di raggiungere

un tasso di vincita del 99,8I successi dell'apprendimento per rinforzo negli ultimi anni hanno portato molti ricercatori a sviluppare metodi di governo dei robot utilizzando il RL [5]. La motivazione è ovvia: possiamo automatizzare il processo di progettazione degli algoritmi di rilevamento, pianificazione e controllo lasciando che il robot li apprenda autonomamente? Se potessimo, risolveremmo due dei nostri problemi in una volta sola; risparmieremmo il tempo e l'energia che spendiamo nella progettazione di algoritmi specifici per i problemi che sappiamo risolvere oggi (robot industriali) e otterremmo soluzioni a quei problemi più difficili per i quali non abbiamo una soluzione attuale. L'apprendimento per rinforzo ha riscosso un considerevole successo grazie agli incredibili risultati ottenuti in vari videogiochi e giochi da tavolo, o problemi di controllo simulato relativamente semplici. In questi compiti, l'ambiente su cui l'agente impara è lo stesso su cui deve operare, e possiamo usare la simulazione in modo efficiente per consentire all'agente molte prove prima che impari a risolvere il compito. Gli algoritmi di apprendimento di rinforzo profondo sono notoriamente inefficienti in termini di dati e spesso richiedono milioni di tentativi prima di imparare a risolvere un compito come giocare a un gioco dell'Atari. Se provassimo ad applicare gli stessi metodi per addestrare il nostro robot nel mondo reale, ci vorrebbe una quantità di tempo non realistica, e probabilmente distruggeremmo il robot nel processo. In alternativa potremmo usare una simulazione per addestrare il nostro agente e poi implementare la politica addestrata sul robot reale. L'esempio di mondo simulato più basilare è quello di una griglia. L'ambiente è dunque discreto ed un agente occupa una casella e deve muoversi tra le caselle ad esso adiacenti al fine di raggiungere un obiettivo. Tipicamente alcune caselle sono inaccessibili per l'agente. Tali caselle stanno a rappresentare muri o pareti.



Chapter 3

Descrizione Algoritmi RL ed introduzione al Q-Learning

3.1 Concetti base del reinforcement learning

Il Reinforcement Learning [1] è una classe di problemi e soluzioni di Machine Learning dove l'attenzione principale è posta sulla mappatura stati-azioni, con l'obiettivo di massimizzare una reward numerica. Il contesto generale del problema prevede che un'entità, chiamata *Agente*, che non sa a priori quali azioni intraprendere, deve imparare ad interagire con l'ambiente circostante, chiamato *Environment*, cercando di scoprire quali azioni producono la più alta ricompensa cumulata sul lungo periodo, piuttosto che nell'immediato futuro. L'agente seleziona le azioni in base allo stato attuale del sistema. L'Environment, reagisce alle azioni dell'agente, e restituisce una ricompensa numerica e il nuovo stato del sistema (Figura 3.1).

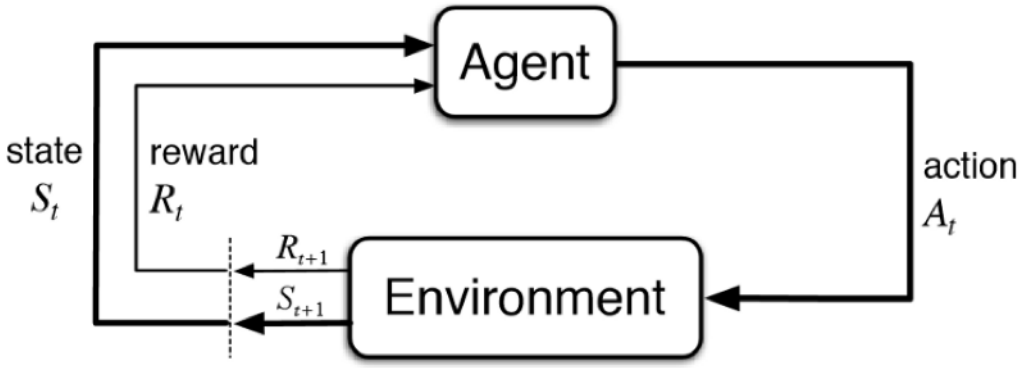


Figure 3.1: Schema che rappresenta le interazioni tra le componenti chiave di un problema di Reinforcement Learning.

La principale differenza tra Reinforcement Learning e Supervised Learning è che esso non basa l'apprendimento su una collezione di esempi etichettati, che descrivono un target di classificazione.. Si differenzia anche dall'Unsupervised Learning in quanto il suo scopo primario non è quello di trovare strutture implicite in raccolte di dati non etichettati. Il RL è molto simile ad un problema di ottimizzazione, in quanto cerca di massimizzare un valore numerico, che può anche portare a scoprire pattern impliciti nella sequenza di selezione delle azioni che formano il comportamento dell'agente, ma non è uno degli obiettivi principali.

Oltre ad Agenti e Ambiente, RL è caratterizzata anche da una serie di altri elementi fondamentali, essenziali per la sua formalizzazione:

- **Azione** (A) - Tutte le possibili mosse che l'agente può compiere.
- **Policy** (π) - Descrive il comportamento dell'agente in un dato momento. È l'elemento principale di un agente RL in quanto è sufficiente a determinare completamente il suo comportamento. Può essere formalizzato come una semplice tabella di ricerca o richiedere formulazioni più complesse, ed in alcuni casi può essere stocastico.
- **Reward** (R) - Definisce l'obiettivo in un problema di apprendimento per rinforzo. Rappresenta la risposta dell'ambiente all'azione scelta dall'agente e stima quanto siano buoni gli eventi per l'agente nell'immediato futuro. Se un'azione selezionata dalla policy produce una ricompensa bassa o negativa, allora la policy può essere modificata per selezionare qualche altra azione in quella situazione in futuro.
- **Modello dell'Ambiente** - Fornisce una rappresentazione dell'ambiente e permette di dedurne il comportamento. Può essere perfettamente noto, di solito nel caso di problemi di pianificazione,(problemi tipicamente indicati come *Model-Based*, o parazialmente/completamente sconosciuti in problemi privi di modelli in cui l'agente deve esplicitamente imparare con un meccanismo "*Trial and Error*" (*Model-Free*).
- **Value Function** - La Value Function stima la ricompensa futura attesa, a partire dallo stato attuale. Fondamentalmente, essa preme l'auspicabilità a lungo termine di quello stato, prendendo in considerazione gli stati che con una certa probabilità gli stati che potrebbero essere scelti e le relative reward.

Il RL può essere formalizzato come un processo decisionale sequenziale di Markov (MDP), in cui la scelta di un'azione ha un impatto non solo sulla ricompensa immediata, ma anche su situazioni, stati e ricompense future. Questo tipo di problemi comporta la necessità di un feedback valutativo rispetto alla scelta di azioni diverse in situazioni diverse. I MDP introducono la necessità di valutare il trade-off tra ricompensa immediata e ricompensa futura. Nella formalizzazione di un MDP, l'agente e l'ambiente interagiscono in una sequenza di passi temporali discreti, producendo una sequenza della forma: $S_0, A_0, R_1, S_1, A_1, R_2, S_2, A_2, R_3, \ldots$. dove A_t è un'azione scelta dall'insieme delle azioni disponibili per un dato stato (se l'insieme delle azioni non è lo stesso in tutti gli stati), mentre le variabili R_t e S_t hanno distribuzioni di probabilità discrete ben definite che dipendono solo dallo stato e dall'azione precedente.
In RL, l'obiettivo principale è quello di massimizzare la ricompensa, quindi un concetto importante è la *Expected Return*, che consiste nella ricompensa cumulata che deve essere massimizzata nel lungo periodo (Eq. 3.1). Le interazioni tra agenti e ambiente sono suddivise in sequenze chiamate episodi, che terminano al raggiungimento di uno stato terminale, dopo di che l'ambiente ritorna ad una configurazione di partenza.

$$G_t = R_{t+1} + R_{t+2} + R_{t+3} + ... + R_T \tag{3.1}$$

La reward attesa può essere scalata con "*Discount Factor*" - che viene poi chiamato *Discounted Return* (Eq. 3.2) - facendo pesare più o meno la ricompensa immediata rispetto alle ricompense raccolte in futuro.

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma R_{t+3} + ... = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \tag{3.2}$$

Un passo importante nel Reinforcement Learning, è quello di stimare la Funzione Valore rispetto ad una policy π, con la State-Value Function che è l'Expected Return quando S è lo stato iniziale e π è la policy dell'agentte (Eq. 3.3). La Value-Action Function, invece, stima il rendimento futuro atteso scegliendo l'azione a quando lo stato è s (Eq. 3.4).

$$v_\pi(s) = \mathbb{E}[G_t|S_t = s] \tag{3.3}$$

$$q_\pi(s, a) = \mathbb{E}[G_t|S_t = s, A_t = a] \tag{3.4}$$

Una delle sfide che ha una grande importanza nel RL, e che in altri tipi di apprendimento (supervisionato e non supervisionato) è meno significativa, è il compromesso tra Exploration (scegliere azioni al di fuori della policy π) ed Exploitation (sfruttare l'apprendimento dell'agente, e seguire dunque in considerazione la policy π).

Un apprendimento per rinforzo tipicamente consta di questi passaggi:

1. Osservazione dell'ambiente
2. Decidere come agire utilizzando una qualche strategia
3. Agire di conseguenza
4. Ricevere una ricompensa o una penalità
5. Imparare dalle esperienze e affinare la nostra strategia
6. Iterare fino a quando non si trova una strategia ottimale

Ci sono 3 approcci principali quando si tratta di implementare un algoritmo RL.

- **Basato sul valore** - in un metodo di apprendimento per rinforzo basato sul valore, si cerca di massimizzare una funzione di valore $v(i)$. L'obiettivo principale è quello di trovare un valore ottimale.
- **Basato sulla policy** - in un metodo di apprendimento di rinforzo basato sulla politica, si cerca di trovare una policy tale che l'azione eseguita in ogni stato sia ottimale per ottenere la massima ricompensa in futuro. L'obiettivo principale è quello di trovare la politica ottimale.
- **Basato sul modello** - in questo tipo di apprendimento di rinforzo, si crea un modello virtuale per ogni ambiente, e l'agente impara ad eseguire in quell'ambiente specifico.

3.2 Esempio di RL basato sul valore : Q-Learning.

Il Q-Learning è una formalizzazione di una famiglia di metodi chiamati Temporal Difference. I metodi TD imparano direttamente dall'esperienza senza richiedere un modello dell'ambiente, aggiornando le stime basate in parte su altre stime apprese, senza aspettare un risultato finale (si dice che fanno "*Bootstrap*"), ma aspettando solo fino al passo successivo. L'aggiornamento viene propagato direttamente sulla transizione dallo stato attuale $S_t$al nuovo stato S_{t+1}, che ha fruttato la reward R_{t+1}. Considerato il nuovo stato e la reward ottenuta, la nuova stima aggiornata per lo stato S al momento t si ottiene attraverso la seguente equazione:

$$V(s_t) = V(S_t) + \alpha[R_{t+1} + \gamma V(S_{t+1}) - V(S_t)] \tag{3.5}$$

dove $R_{t+1} + \gamma V(S_{t+1})$ è chiamato target, ed è una stima, in quanto campiona il valore atteso e allo stesso tempo utilizza la stima corrente della funzione di valore V . Inoltre, la differenza tra $V(S_t)$ e il target è chiamata $TD\ error$ nella stima effettuata nell'istante t, che in realtà è disponibile solo un passo dopo e quindi può essere indicato come l'errore in $V(S_t)$ nell'istante $t + 1$.

L'obiettivo di un algoritmo di Q-learning è quello di apprendere una policy, che suggerisce ad un agente quali azioni intraprendere in quali circostanze. Non richiede un modello dell'ambiente (è "model-free"), ed è in grado di gestire i problemi di transizione stocastica e di reward, senza richiedere adattamenti. L'obiettivo dell'agente è quello di massimizzare la sua ricompensa totale futura. Tale obiettivo si ottiene aggiungendo la massima ricompensa ottenibile dagli stati futuri alla ricompensa dello stato attuale, influenzando efficacemente l'azione attuale con la potenziale ricompensa futura. Questa ricompensa potenziale è una somma ponderata dei valori attesi delle ricompense di tutti i passi futuri a partire dallo stato attuale. Quando viene eseguito un algoritmo di Q-Learning si crea quella che viene chiamata Q-Table che è una struttura dati di forma [stato, azione].I valori della tabella sono inizializzati ad un dato valore e poi aggiornati episodio dopo episodio. Questa tabella verrà utilizzata dall'agente per selezionare l'azione migliore in base al valore Q. Essa contiene i valori attesi Q della *action-value function* q^*. Il passo successivo è semplicemente quello di far interagire l'agente con l'ambiente e di aggiornare i valori per le coppie stato-azione nella nostra Q-Table $Q[state, action]$.
Un'altra proprietà chiave del Q-Learning è che esso generalizza un mondo non-deterministico. In altre parole, se la transizione da uno stato all'altro è non-deterministica, la regola di aggiornamento che abbiamo discusso in precedenza, continua a far convergere q a q^* [6]

3.2.1 Deterministic Case Learning Rule

Formalmente, l'obbiettivo di massimizzare la ricompensa totale futura si ottiene massimizzando il valore atteso Q della *action-value function* q^* , anzichè della *state-value function*, indipendentemente dalla policy seguita. La *Q-Function* (Eq. 3.7) restituisce un valore che indica la qualità della scelta di eseguire un'azione A in un certo stato S.

$$Q(S, A) = R + \gamma \max_{a'} Q(S, a') \tag{3.6}$$

I riferimenti alla precedente equazione sono riportartati nel paragrafo seguente, poichè condivisi con il caso non deterministico. Si può dimostrare che con questa regola di aggiornamento Q converge all' ottimo $Q*$

3.2.2 Non Deterministic Case Learning Rule

E' utile introdurre il concetto di Non-Determinismo nell'ambito della robotica poichè può formalizzare eventi inattesi comuni del mondo reale come il rumore osservato nelle

letture dei sensori o nell'attuazione dei meccanismi hardware dei robot. Introducendo il Non-Determinismo nell' Environment si necessita di modificare la regola di training del paragrafo precedente poichè l'Environment ora produce rewards diverse ogni volta che la transizione dallo stato S_t allo stato S_{t+1} viene effettuata, dato che la transizione ora è probabilistica. Più specificamente, quando un'agente sceglie un'azione da eseguire, l'esecuzione di quest'ultima avviene con una certa probabilità, e potrebbe dunque compiere un'azione diversa da quella scelta. La regola d'aggiornamento 3.6 con questo setting non riesce a convergere a $Q*$. Utilizzando $Q(S_t, A_t)$ per indicare la stima dell'agente all' iterazione t, la seguente regola di aggiornamento garantisce la convergenza :

$$Q(S_t, A_t) = Q(S_t, A_t) + \alpha(R_t + \gamma \max_{a'} Q(S_{t+1}, a') - Q_n(S_t, A_t)) \tag{3.7}$$

In pratica questa regola d'aggiornamento esprime il fatto che si aggiornano i valori Q in base alla differenza tra i nuovi valori pesati e i vecchi valori. Pesiamo i nuovi valori utilizzando γ e utilizzando il tasso di apprendimento (α)) esprimiamo quanto consideriamo l'esperienza pregressa per l'addestramento. Di seguito sono riportati alcuni riferimenti :

- **Learning Rate** α : Determina quanto le nuove informazioni acquisite sovrascriveranno le informazioni pregresse. E' compreso fra 0 ed 1. Un valore 0 impedisce all'agente di apprendere, mentre un valore 1 fa si che l'agente si interessi solo alle informazioni contingenti.

- **Gamma** γ : E' un fattore di peso. Viene utilizzato per bilanciare la ricompensa immediata e quella futura. Nella regola di aggiornamento di cui sopra si applica gamma alla ricompensa futura. Un fattore pari a 0 porta l'agente a considerare solo le ricompense attuali, (nella regola di aggiornamento di cui sopra), mentre un fattore che si avvicina a 1 lo porterà a cercare una ricompensa elevata a lungo termine. Tipicamente questo valore può variare da 0,8 a 0,99.

- **Reward** R : la reward è il valore restituito all'agente dopo aver scelto una certa azione in un determinato stato. Una reward può essere a valori discreti (Sparse Reward) o assumere valori in uno spazio continuo (Shaped Reward).

- $\boldsymbol{\max_{a'} Q(S_{t+1}, a')}$: Si prende il massimo valore di ricompensa futura e lo si applica alla ricompensa per lo stato attuale. Rappresenta l'impatto dell'azione attuale sulla possibile ricompensa futura. Così facendo si aiuta l'agente a selezionare le migliori azioni che massimizzano la ricompensa totale.

3.2.3 Aggiornamento della tabella q

Gli aggiornamenti avvengono dopo ogni passo o azione e terminano al termine di un episodio. La fine di un episodio si presenta quando l'agente raggiunge uno stato terminale. Uno stato terminale, per esempio, può essere il raggiungimento della fine di un gioco, il completamento di qualche obiettivo desiderato, ecc. L'agente non imparerà molto dopo un singolo episodio, ma episodio dopo episodio l'algoritmo approssimerà con accuratezza crescente i valori ottimali di Q : ($Q*$) . Ecco i 3 passi fondamentali :

1. L'agente inizia in uno stato ($s1$) compie un'azione ($a1$) e riceve una ricompensa ($r1$)

2. L'agente seleziona l'azione facendo riferimento alla tabella Q con il valore più alto (max) o a caso (ε)

3. 3. Aggiornamento dei valori q

3.2.4 Exploration e Exploitation:

Un agente può scegliere un'azione da compiere in due modi differenti. Il primo è quello di usare la Q-Table come riferimento e tra tutte le azioni possibili per un dato stato, selezionarla in base al valore massimo Q di tali azioni. Questa modalità di interazione è nota come exploitation, poiché usiamo le informazioni che abbiamo a disposizione per prendere una decisione. Il secondo è quello di agire di in modo casuale. Questo si chiama esplorare. Invece di selezionare azioni basate sulla massima ricompensa futura, si seleziona un'azione randomica. Agire in modo casuale è importante perché permette all'agente di esplorare e scoprire nuovi stati che altrimenti non potrebbero essere selezionati durante il processo di exploitation. È possibile bilanciare exploration ed exploitation usando la costante ε. Tale costante esprime il rapporto tra le volte in cui l'agente sfrutta i Q-Values presenti nella Q-Table per eseguire un'azione rispetto alla scelta random di quest'ultima .

Chapter 4

Descrizione Algoritmo

L'algoritmo simula la risoluzione di un problema di navigazione di due robot in maniera concorrente all'interno di una mappa nella quale degli agenti sono liberi di muoversi per raggiungere determinati goals. Ogni agente deve soddisfare due: un task intermedio, distinto per ognuno dei robot ed uno comune. Questo simula una situazione reale nella quale due robot devono muoversi in un ambiente di lavoro per prendere degli oggetti in diverse zone e portarli in un unico punto che potrebbe essere ad esempio un tavolo da lavoro. Gli agenti devono imparare autonomamente a raggiungere goal e sottogoals. E' stato ritenuto opportuno utilizzare come approccio di apprendimento il Q-learning poichè permette di introdurre il nondeterminismo nella scelte delle mosse degli agenti. Il non determinismo consente di dotare gli agenti di un meccanismo di obstacle avoidance in quanto, la presenza di un ostacolo improvviso non necessita di arresto da parte dell'agente ma di una semplice deviazione, rendendo dunque il comportamento dell'agente più flessibile

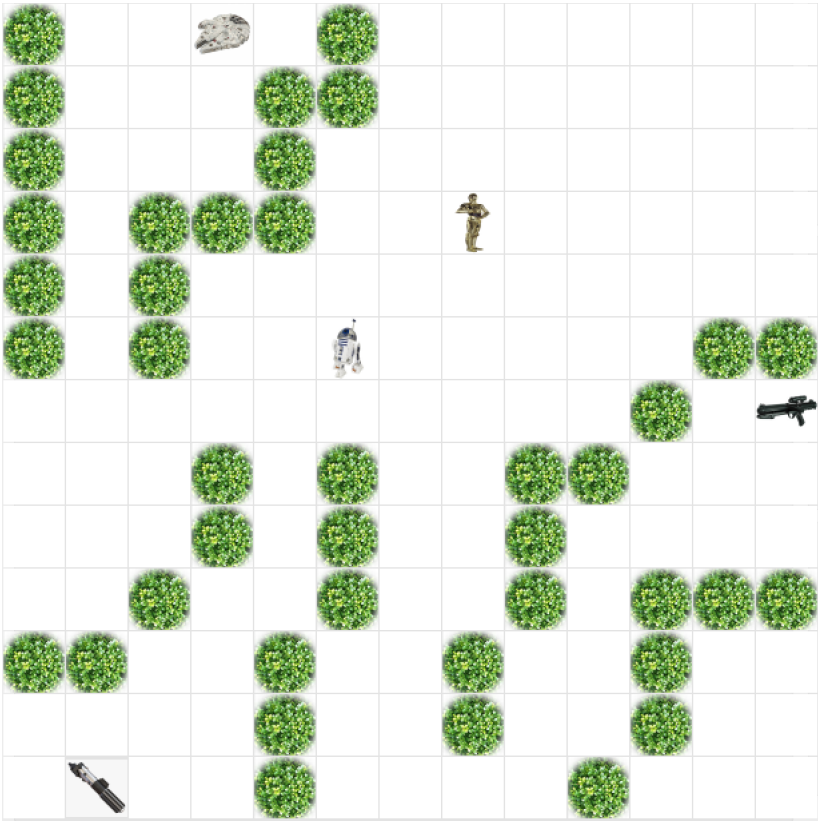


Figure 4.1: Mappa di gioco

4.1 Modello

Il modello è di tipo discreto, ossia è rappresentato da una griglia nella quale gli agenti possono muoversi nelle quattro direzioni. Gli elementi del modello, ovvero le diverse tipologie di celle, sono :

- Celle libere : Rappresentano le caselle che un agente può raggiungere .
- Kit : (Blaster e Spada Laser in 4.1) Rappresenta un obiettivo intermedio di un agente.
- Goal :(Nave spaziale in 4.1) Rappresenta l'obiettivo finale di un agente.
- Muri : Rappresentano caselle inaccessibili all'utente

4.2 Funzione di Reward

La funzione di reward utilizzata è di tipo sparso, essa associa un valore negativo agli stati in step intermedi ed un valore positivo agli stati terminali. Nel caso in questione gli stati relativi al goal intermedio e finale sono considerati terminali. Immaginando lo scenario realistico, dove un utente umano attende i due kit da lavoro che verranno consegnati dai due agenti, il principale requisito funzionale è quello che i robot impieghino il minor tempo possibile nella consegna del proprio kit. E' per ottenere questo risultato che si è scelto di costruire una funzione di reward che assegna un punteggio negativo ad ogni mossa; in questo modo l'agente per massimizzare la reward totale sarà portato a compiere meno mosse possibili. E' importante notare che si è scelto di non penalizzare l'agente semmai esso decidesse di oltrepassare un muro, poiché questa possibilità è stata inibita. Questa scelta è motivata dal fatto che in un ambiente reale, tipicamente la lettura dei sensori evita all'agente di scontrarsi con degli ostcoli, e dunque l'agente in fase di learning, quando incontra un ostacolo non sceglie mai di scontrarvisi per poi tornare nel punto di partenza e ricominciare. Inoltre tale scelta, come vedremo più avanti ha reso l'apprendimento più rapido e stabile.

Listing 4.1: Reward function

```
def giveReward(self):
    if self.state == self.win:
        return 0
    else:
        return -5
```

4.3 Spazio degli stati e azioni

Per ogni cella della mappa vi sono 4 stati : 2 per ogni agente. Uno stato relativo al task del raggiungimento del goal intermedio ed un altro relativo a quello del raggiungimento del goal finale.

Sia W il numero di celle orizzontali ed H il numero di celle verticali. Il numero degli stati sarà dunque:

$$|S| = W \times H \times 4 \tag{4.1}$$

L'insieme delle azioni per ogni stato ogni agente è composto da alcune tra le quattro azioni di navigazione della griglia : (up, down, left, right) . Il fatto che per alcuni stati non siano disponibili tutte e quattro le azioni dipende dal fatto che alcune azioni potrebbero portare fuori dalla mappa o allo scontro con degli ostacoli o con altri agenti.

4.4 Learning Rule E Addestramento

La regola di aggiornamento utilizzata è quella classica del Q-Learning mostrata nel capitolo precedente. (Eq. 3.7)
La tabella Q ,come abbiamo già visto, contiene per ogni coppia (stato, azione) i Q-Values che esprimo quanto effettuare una determinata azione in un determinato stato è conveniente per l'agente in termini di reward futura.

Listing 4.2: Learning Rule Code

```
def QLearn(self, rounds,j,train):
        count=0
        global GRID,PROGRESSVALUE
        passi = 0
        passiDelta = 0
        i = 0
        direct = "right"
        self.exp_rate = 0.3
        prev=self.setEnvironment(j)
        while i < rounds:
            passiDelta = 0
            while True :
                passiDelta+=1
                # we reached the end
                if self.State[j].isEnd :
                    Q_Delta[self.id][j].append(passiDelta)
                    if self.State[j].giveReward() == 0 :
                        #Decaying Learning Rate
                        self.exp_rate = self.exp_rate * 0.9997
                         i += 1
                        # "For agent (self.id)  Episode number: i

                    self.Q_prev = copy.deepcopy(self.Q_values)
                    prev = self.setEnvironment(j)
                    self.reset(j)
                    break
                # modify Value function
                action = self.chooseAction(j,train,direct)
                finalState = self.takeAction(action,j)
                if finalState != self.State :
                    passi = passi + 1
                  # Direzione d'esplorazione
                    if passi == sogliaDirezione :
                        if direct == "right" :
                            direct = "left"
                        elif direct == "left" :
                            direct = "up"
                        elif direct == "up" :
                            direct = "down"
                        else :
                            direct = "right"
                        passi = 0
                    reward = finalState.giveReward()
                    finalState.isEndFunc()
                    if train :
                    #AGGIORNAMENTO Q-TABLE (START LEARNING RULE SECTION)
                        maximum = float("-inf")
                        for a in self.actions :
                         #INVALID ACTIONS
                            if (finalState.state[0]==0 and a=="up") or (finalState.state[0]==R...
                                    continue
                            cur = self.Q_values[j][finalState.state][a]
                            if cur > maximum :
                                maximum = cur
                        current_q_value =  self.Q_values[j][self.State[j].state][action]

                        actual = round(self.Q_values[j][self.State[j].state][action] +
                        self.lr*(reward + self.gamma*maximum -
                        self.Q_values[j][self.State[j].state][action]),3)

                        self.Q_values[j][self.State[j].state][action] = actual
                    #FINE AGGIORNAMENTO Q-TABLE (END LEARNING RULE SECTION)
                    self.State[j] = finalState
```

Durate la fase d'apprendimento entrambi gli agenti sono tra di loro invisibili, cioè la presenza di un altro agente non è contemplata tra i possibili elementi della mappa. Questo per garantire che un agente non interpreti come ostacolo la presenza momentanea di un elemento sulla mappa, alterando l'apprendimento. Finito l'addestramento i Q-values degli agenti vengono salvati ed usati successivamente per muoversi sulla mappa.

4.5 Policy

Durante l'addestramento la policy dell'agente è la seguente : L'agente inizialmente compirà solo scelte random (exploration), ed episodio dopo episodio aumenteranno le volte in cui la scelta dipenderà dalla tabella Q (exploitation). Per migliorare l'esplorazione e velocizzare il learning, le scelte random sono state condizionate ad una direzione :

l'agente compie scelte random preferendo tuttavia una delle quattro direzioni. Tali direzioni cambiano ciclicamente dopo un certo numero di passi. Questo accorgimento porta l'agente ad esplorare più porzioni di mappa rispetto ad un'esplorazione totalmente random, poiché in questo modo l'agente tende sempre a non discostarsi troppo dal punto di partenza. [7]

Listing 4.3: Action to perform choice

```
def chooseAction(self,j,train,direct):
    # choose action with most expected value
    mx_nxt_reward = float("-inf")
    action = ""
    if np.random.uniform(0, 1) <= self.exp_rate and train :
       while(True):
           if not train :
             if direct=="up" :
               action = np.random.choice(["up", "left", "right","down"], p=[0.4,0.2,0.2,0.2]
             if direct=="left" :
               action = np.random.choice(["up", "left", "right","down"], p=[0.2,0.4,0.2,0.2]
             if direct=="right" :
               action = np.random.choice(["up", "left", "right","down"], p=[0.2,0.2,0.4,0.2]
             if direct=="down" :
               action = np.random.choice(["up", "left", "right","down"], p=[0.2,0.2,0.2,0.4]
                #INVALID ACTIONS
             if (self.State[j].state[0]==0 and action=="up") or (self.State[j].state ...
                       continue
             else:
                return action
     else:
        # greedy action
        for a in self.actions:
           current_position = self.State[j].state
             #INVALID ACTIONS
             if (current_position[0]==0 and a=="up") or (current_position[0]==R....
                   continue
             nxt_reward = self.Q_values[j][current_position][a]
             if nxt_reward > mx_nxt_reward:
                action = a
                mx_nxt_reward = nxt_reward

    return action
```

Come si evince dal codice la funzione *choseAction* restituisce un'azione da compiere, tuttavia questa non sarà l'azione finale, poichè, essendo tale scelta non deterministica un'ulteriore funzione,*chooseActionProb*, che prende in input tale azione, da in output l'azione finale, che con una certa probabilità non è quella restituita dalla funzione *choseAction*.

Listing 4.4: Probabilistic chioce

```
    def chooseActionProb(self, action):
        if action == "up":
            action = np.random.choice(["up", "left", "right"], p=[0.8, 0.1, 0.1])
            return action
        if action == "down":
            action = np.random.choice(["down", "left", "right"], p=[0.8, 0.1, 0.1])
            return action
        if action == "left":
            action = np.random.choice(["left", "up", "down"], p=[0.8, 0.1, 0.1])
            return action
        if action == "right":
            action = np.random.choice(["right", "up", "down"], p=[0.8, 0.1, 0.1])
            return action
```

4.6 Testing

Gli agenti useranno come conoscenza dell'ambiente per muoversi in esso i Q-values salvati durante la fase di training. Durante il testing, differentemente da quanto accade durante il training gli agenti non sono invisibili l'uno all'altro, ed è necessario che essi comunichino tra di loro per non scontrarsi. Il setting scelto è quello di Master-Slave, dove uno dei due agenti si muove indipendentemente dalle azioni dell'altro agente e quest'ultimo ha l'onere di evitare di intralaciare il cammino del Master. E' possibile inoltre scegliere di effettuare un test con scelte di azioni deterministiche o non deterministiche. Utilizzando un testing deterministico l'agente impiegherà meno passi a compiere i suoi compiti, tuttavia sarà più sensibile a cambiamenti dinamici della mappa di gioco (Aggiunta di ostacoli permanenti). Con un testing deterministico, viceversa, l'agente compierà mediamente più passi per completare i task, ma sarà meno sensibile all'introduzione di ostacoli.

Chapter 5

Graphical User Interface

L' interfaccia è stata sviluppata al fine dimostrativo del funzionamento dell'algoritmo implementato. Essa è stata implementata utilizzando il framework **PyQt5**. Il tutto è stato implementato utilizzando un'unica window, le varie funzionalità sono illustrate di seguito.

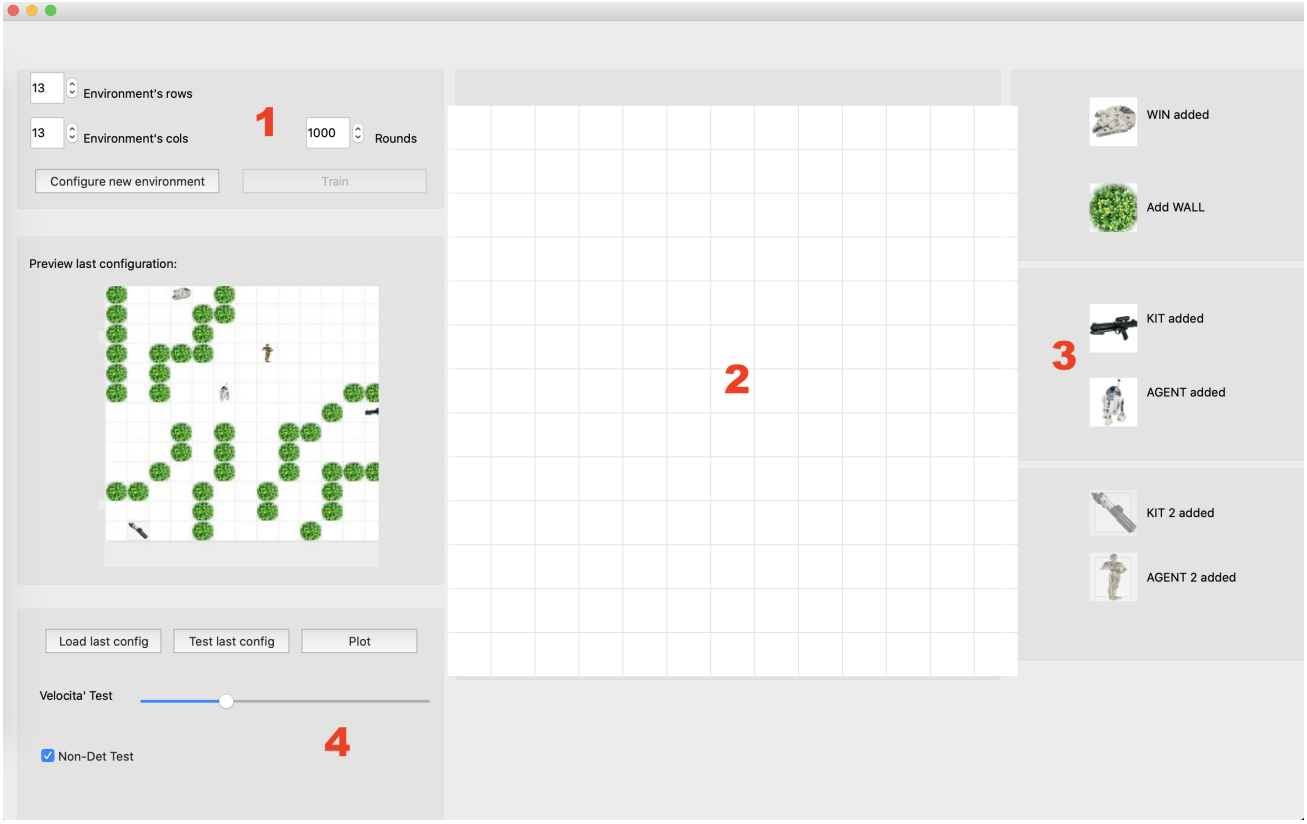


Figure 5.1: Interfaccia Grafica Completa

Osserviamo le diverse zone dell'interfaccia dell'immagine 5.1 :

1. settaggi per la costruzione dell' ambiente ove gli agenti opereranno.
2. mappa di gioco
3. component button, per selezionare l' entità da inserire nella mappa.
4. Preview dell'ultima configurazione salvata della mappa.

L'utente può costruire la mappa scegliendone la dimensione in termini di celle utilizzando gli slider che si trovano nella zona 1(Fig. 5.2) e successivamente premendo il pulsante "**Configure Environment**". Questa interazione creerà la mappa di gioco, composta da celle vuote. A questo punto sarà possibile interagire con la mappa, per riempirla degli elementi che caratterizzano il problema in oggetto.

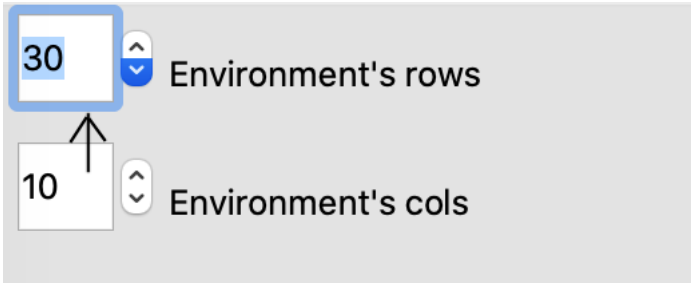


Figure 5.2: Slider Dimensione

Per inserire gli elementi quali agenti, goals e muri, è possibile selezionarli dalla zona 3, cliccando sulle relative icone, e inserirli nella mappa di gioco, cliccando sulla cella in cui si desidera inserire l'elemento. Una volta creata la mappa di gioco con tutti gli elementi è possibile lanciare il training tramite il pulsante "**Train**". Il numero di epoche è selezionabile tramite lo slider nella ziona 1 (Fig. 5.3).

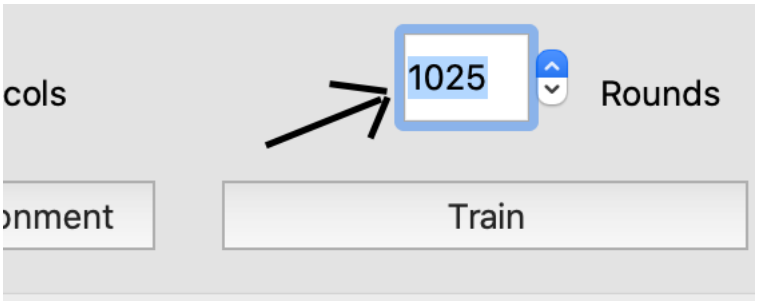


Figure 5.3: Numero di Epoche

Nell'immagine sottostante si può notare la finestra di dialogo che mostra l'avanzamento del training (Fig. 5.4).

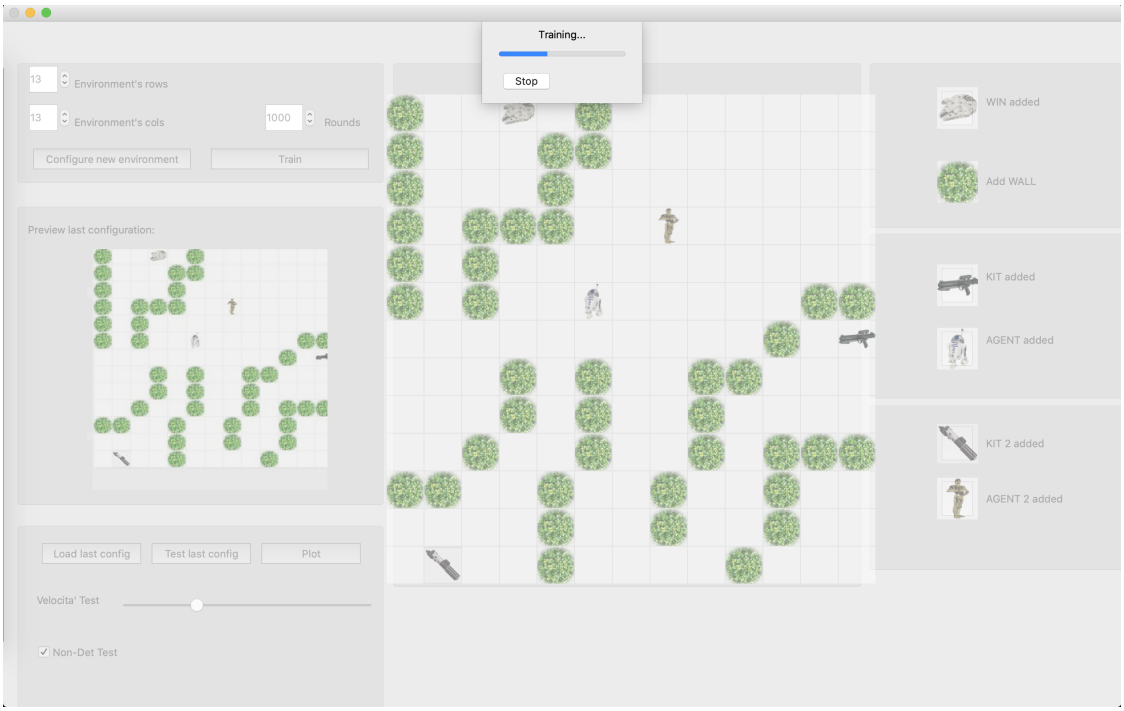


Figure 5.4: Schermata Avanzamento Training

Una volta terminato il training, si può testare il modello interagendo con la zona 4 dell'interfaccia o consultare i grafici che descrivono l'andamento del training tramite il pulsante "**Show Plot**" In questa porzione di UI viene mostrata la configurazione della mappa di gioco e utilizzando il pulsante "**Test last configuration**" si può testare il modello (Fig. 5.5). La configurazione in preview verrà caricata nella mappa centrale della zona 2, e gli agenti si muoveranno nella mappa utilizzando la Q-Map appresa durante l'addestramento. Si può scegliere, tramite la checkbox "**Non-Det Test**" se consentire il non determinismo durante il test. Alternativamente per visualizzare solamente, ed eventualmente modificare la mappa dell'ultima configurazione, per poi riaddestrare il modello si può utilizzare il pulsante "**Load Last Config**" Durante questa fase, l'utente può comunque interagire con la finestra, aumentando o diminuendo la velocità degli agenti tramite lo slider "**Velocità Test**" o introducendo durante l'esecuzione degli ostacoli sulla mappa di gioco premendo con il puntatore sulle caselle.

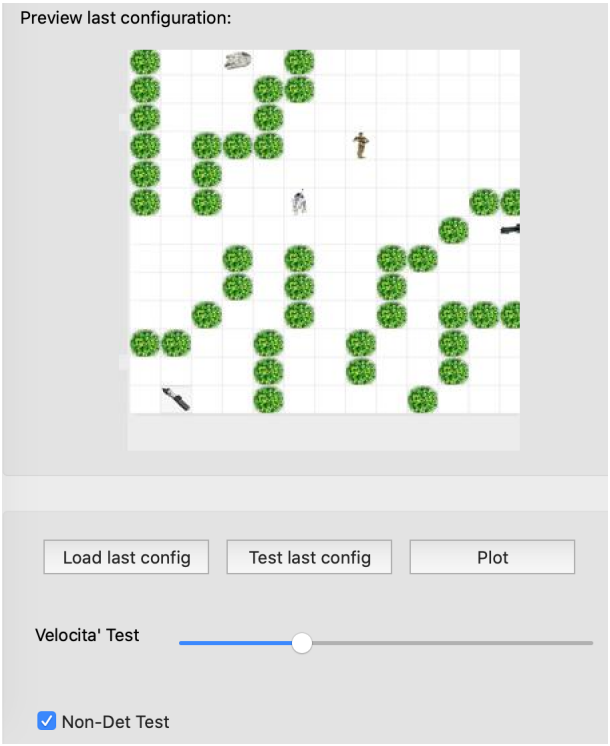


Figure 5.5: Zona Ultima Configurazione

Chapter 6

Valutazione Risultati

Lo scopo del progetto è stato quello di utilizzare un approccio di reinforcement learning in un problema di navigazione con goal e sub-goals. Essendo, come già riportato nel capitolo precedente, il principale requisito funzionale quello di raggiungere i goal nel minor tempo possibile, si è scelto di valutare il comportamento degli agenti in base al numero di passi necessari per raggiungere gli stati terminali. Questa scelta è dettata anche dal fatto che essendo il problema in questione un problema di navigazione, e che non sono essendo ipotizzati stati terminali di sconfitta, l'utilizzo del classico meccanismo di valutazione del reinforcement basato sul rapporto vittore/sconfitte è del tutto superfluo. Con l'aumentare della dimensione della mappa, ovviamente i tempi di training crescono proporzionalmente Per testare il modello si è scelto di testare tempi e numero di passi medi su 50 computazioni per 4 dimensioni della mappa differenti : 10x10, 15x15,20x20 e 25x25. Per le prime 3 dimensioni è stata preso per riferimento un numero di 3000 epoche, mentre per l'ultima di 4000. Tale scelta è dovuta al fatto che per la grandezza della mappa, utilizzare solo 3000 epoche come nei primi 3 casi il porta il modello, talvolta, a non convergere ad una policy ottimale.

La configurazione della macchina su cui sono stati eseguiti i test :

- CPU : intel i5-7600k.
- RAM: 8gb ddr4.

Di seguito i risultati dei test :

6.1 Tempi medi training

Tabella tempi :

10 x 10 (3000 epoche)	25,61 sec ± 1,67
15x15 (3000 epoche)	45,84 sec ± 2,56
20x20 (3000 epoche)	52,76 sec ± 3,46
25x25 (4000 epoche)	2 min 57.08 sec,± 8.34

6.2 Numeri dei passi

Per quanto riguarda i test sul numero di passi, per ciascuna delle 4 dimensioni scelte si è calcolata, per ogni epoca la media del numero di passi necessari al raggiungimento del goal. Di seguito i grafici :

10 x 10

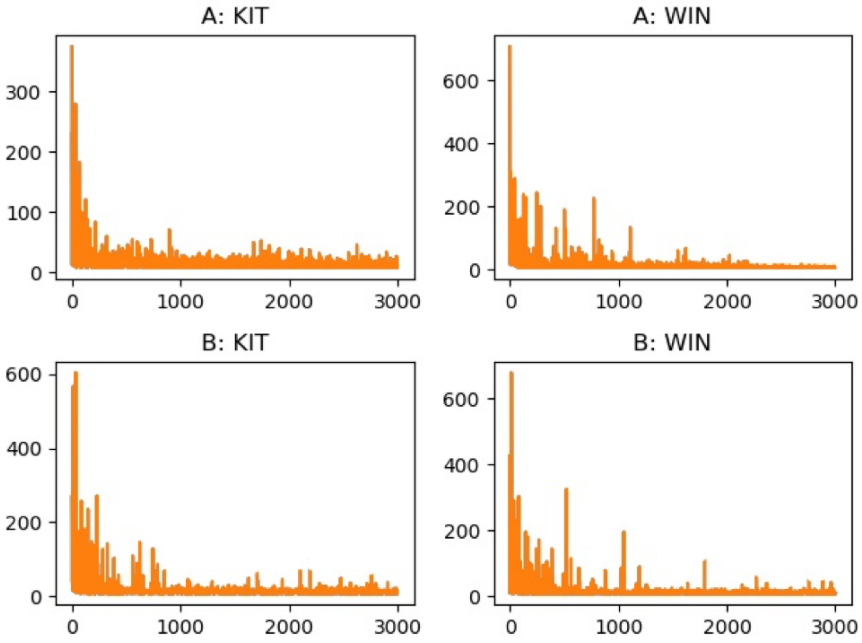


Figure 6.1: Numero di passi medi su mappa 10x10

15 x 15

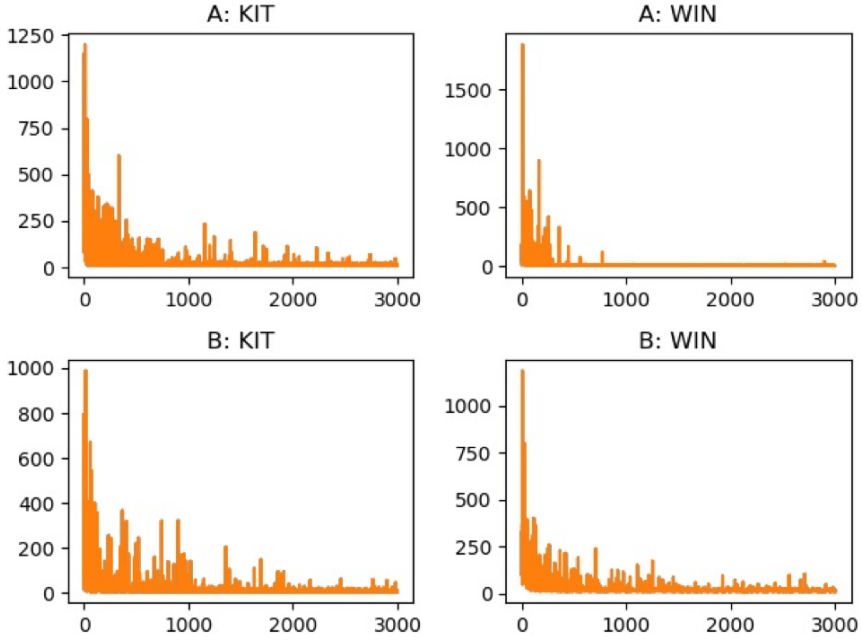


Figure 6.2: Numero di passi medi su mappa 15x15

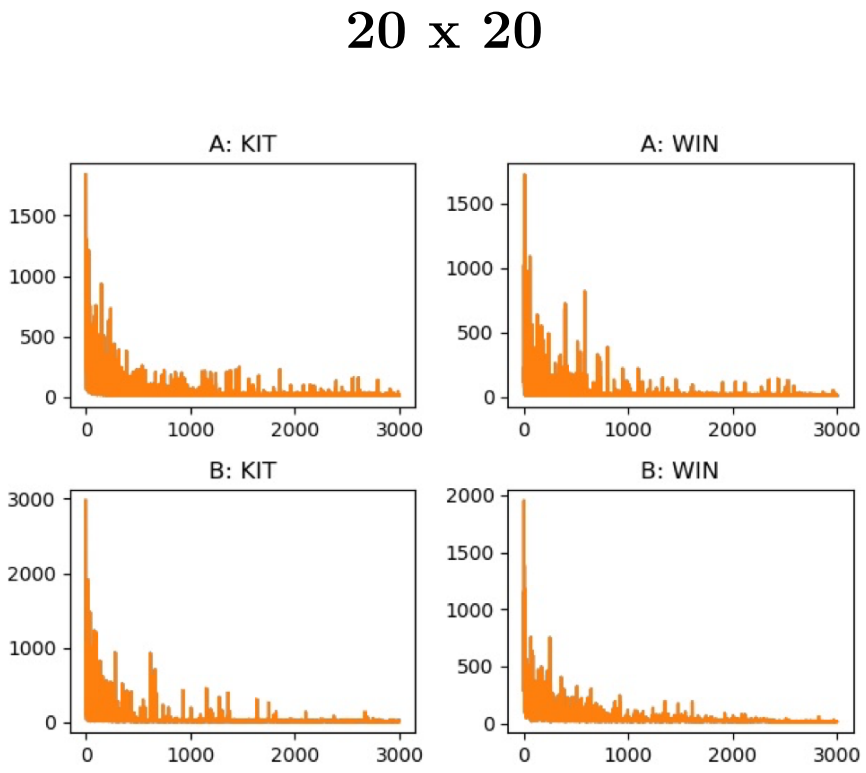


Figure 6.3: Numero di passi medi su mappa 20x20

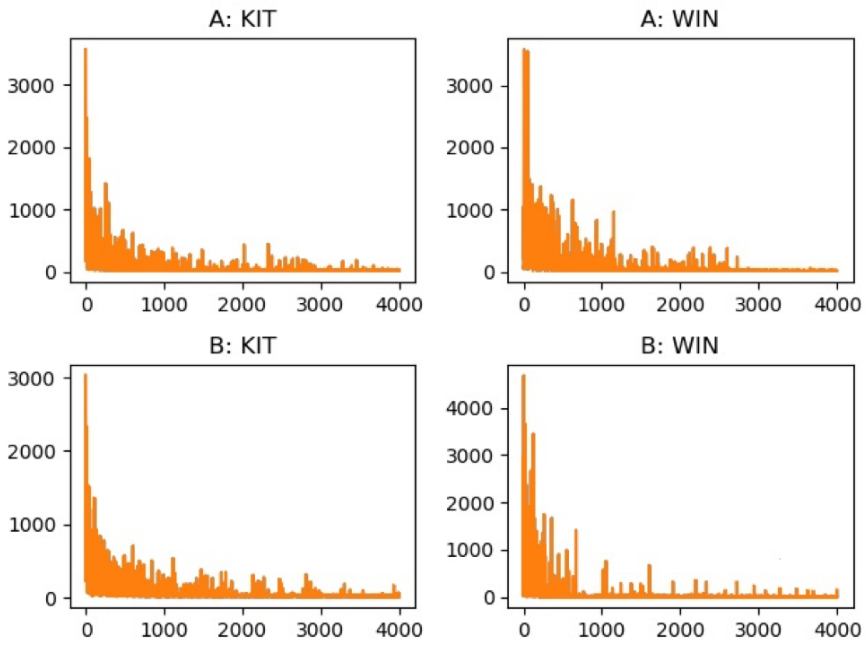


Figure 6.4: Numero di passi medi su mappa 25x25

Come si può apprendere dai grafici gli agenti imparano effettivamente a raggiungere goals e sub-goals via via in un minor numero di passi. I picchi saltuari che si possono notare sono dovuti al fatto che il procedere degli agenti è non deterministico e dunque essi subiscono delle deviazioni nel loro tragitto verso i gols. Nonostante questi picchi, si nota che l'andamento generale della curva è decrescente e che già dopo poche centinaia di epoche, anche sulla mappa 25x25, gli agenti calano drasticamente il numero di passi necessari a raggiungere i goal.

Chapter 7

Conclusioni e sviluppi futuri

Il progetto implementato soddisfa i requisiti concordati, come già illustrato nel capitolo " Valutazione dei risultati" abbiamo ottenuto un apprendimento che corrispondeva ad uno scenario reale. I punti che potrebbero essere sviluppati in futuro sono:

1. Trasformare gli agenti da operatori individuali a operatori concorrenti: cosi facendo, utilizzando un meccanisco di comunicazioni tra di essi, si potrebbe puntare a svincolare le coppie kit-agenti e consentendo un apprendimento legato al kit che consente al robot di raggiungere in meno passi il goal finale.

2. Consentire una modifica real time dell environment cosi da poter valutare e apprezzare in modo massiccio il funzionamento del Q-learning.

Bibliography

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

[2] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[3] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

[4] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587):484, 2016.

[5] Dídac Busquets, Ramon López de Màntaras, and Carles Sierra. Reinforcement learning for landmark-based robot navigation. In *Proceedings of the first international joint conference on Autonomous agents and multiagent systems: part 2*, pages 841–842, 2002.

[6] Christopher JCH Watkins and Peter Dayan. Q-learning. *Machine learning*, 8(3-4):279–292, 1992.

[7] Karl Pearson. The problem of the random walk. *Nature*, 72(1867):342–342, 1905.