Multiagent Systems: progetto con reti neurali e Q-learning multiagente

Marioemanuele Ghianni
E-mail address
marioemanuele.ghianni@stud.unifi.it

Abstract

L'elaborato che verrà presentato è parte dell'esame del corso magistrale in ingegneria informatica di Multiagent Systems. L'implementazione è stata fatta in linguaggio di programmazione Python e mostra un "toy example" di reinforcement learning multiagente.
Verrà presentato inoltre l'algoritmo di reinforcement learning utilizzato e alcune varianti che sono state poste a confronto in alcuni tests esposti a fine paper.
L'elaborato si pone dunque come approfondimento pratico di argomenti trattati nel corso e mostra una metodologia e un approccio di base che può essere applicato ad un ampia gamma di problemi di learning multiagente.

1. Introduzione e concetti fondamentali

Il progetto qui esposto si pone di risolvere un particolare task in cui dei robot, o comunque degli agenti mobili, devono coordinarsi per un compito particolare cercando di limitare allo stesso tempo le collisioni ambientali e tra di loro.
Il tutto è stato implementato in Python ed è usata la libreria Pygame per mostrare, sottoforma di "toyexample", il problema su una griglia 8x8 che viene animata ad ogni step.
Il progetto si colloca nel contesto del **Reinforcement Learning** e ha richiesto l'implementazione di un algoritmo di **Q-learning**.
in particolare, è stata implementata una sua variante denominata **Deep Q-Learning** che porta 3 principali miglioramenti e che verrà esposta nel dettaglio in seguito.
A tal proposito, sono state impiegate librerie per reti neurali quali **Keras** e **Tensorflow** e l'utilizzo di una GPU Nvidia con relativo toolkit CUDA è indicato per velocizzare le fasi di training e portarle a tempi ragionevoli

1.1. Reinforcement Learning

Il Reinforcement Learning è un campo del Machine Learning basato su un **apprendimento automatico** che punta a realizzare agenti autonomi in grado di scegliere azioni da compiere per il conseguimento di determinati obiettivi tramite interazione con l'ambiente in cui sono immersi.
E' utilizzato da vari software e macchine per determinare il miglior percorso o comportamento possibile in specifiche situazioni e differisce dall'apprendimento supervisionato in quanto quest'ultimo necessita di dati di training e quindi di una sorta di "guida" per apprendere.
Nel caso del reinforcement learning, l'agente ha la piena responsabilità delle proprie azioni e attraversa un processo di apprendimento basato sulla propria esperienza sul campo e guidato unicamente da dei rewards che possono essere anche sparsi o parziali.
Prendendo in considerazione un agente definito da un modello di decisione di Markov, si ha che questo è in primo luogo caratterizzato da uno spazio degli stati X, da uno spazio delle azioni U e da una funzione di transizione di probabilità $f : X * U * X \rightarrow [0, \inf]$ atta a fornirgli informazione, generalmente non deterministica, nei confronti dell'ambiente col quale interagisce.
Obiettivo di un agente autonomo e quindi quello di scegliere ad ogni istante di tempo t, a seconda dello stato $x(t) \in X$ in cui questi si trova, l'azione $u(t)$ migliore tra quelle appartenenti allo spazio delle azioni.
La qualità dell'azione intrapresa è determinata da una funzione detta **reward** che può essere parziale o finale.
L'agente dunque definirà una **policy**; ovvero una funzione che definisce l'azione che l'agente deve applicare a seconda dello stato in cui si trova.
Nel caso deterministico, si ha che, applicando l'azione u_k nello stato x_k, l'agente otterrà un reward pari a
$r_{k+1} = r(x_k, u_k)$. L'obiettivo di un algoritmo di RL è dunque, nello specifico, quello di definire una policy che garantisca all'agente, ad ogni istante di tempo, non il reward istantaneo ottimo (algoritmo greedy) ma di massimizzare il reward futuro e quindi di poter scegliere a priori, a seguito di sufficiente esperienza, la sequenza di azioni ottima.

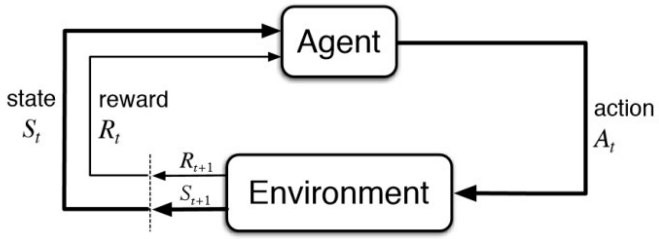

Figure 1: Schema concettuale e semplificato dell'interazione agente/environment.

Entrando nello specifico, per questo elaborato, si andranno a considerare **problemi deterministici ad orizzonte infinito** su cui andremo a definire una *value function* e una *policy ottima*.
La value function è una funzione che si associa ad una policy h e restituisce, per ogni stato, ad ogni istante di tempo il reward futuro ottenuto prendendo ad ogni istante t la decisione $u(t)$ per mezzo dell'applicazione della policy h in questione.
Tale funzione si definisce quindi come:

$$V^h(x(t),t) = r(x(t),u(t)) + \gamma V^h(x(t+1),t+1)$$
$$\text{con } \gamma \in [0,1)$$

Dove γ rappresenta il fattore di sconto necessario affinché la serie non diverga e per attribuire valore via via più inferiore ai reward più lontani temporalmente.
L'obiettivo del RL è dunque andare ad individuare, tra tutte le policy possibili, quella che rende massima la value function e che quindi permette all'agente di ottenere un reward complessivo maggiore.
Si giunge dunque a definire la *value function ottima* risolvendo una particolare equazione detta *equazione di Bellman della programmazione dinamica* :

$$V^o(x(t),t) = max_{\text{u(t)}} r(x(t),u(t)) + \gamma V^o(x(t+1),t+1)$$

Una volta ricavato V^o per ogni stato, si può ricavare la policy ottima come:

$$h^o(x,t) = argmax_{\text{u(t)}} r(x(t),u(t)) + \gamma V^o(x(t+1),t+1)$$

Fondamentale è, dunque, il calcolo della value function ottima che può essere ricavata tramite due principali metodologie: la *value iteration* e la *policy iteration*.

1.2. Q-Learning

Il Q-learning rientra nell'approccio di value iteration ed è uno degli algoritmi più usati nel campo del RL.
Si introduce dunque la funzione:

$$Q^o(x(t),u(t)) = r(x(t),u(t)) + \gamma V^o(x(t+1),t+1)$$

detta *action value function* che fornisce il reward ottenuto applicando le migliori decisioni da $t+1$ in poi e l'azione $u(t)$ al tempo t in cui l'agente si trova nello stato $x(t)$.
Possiamo poi esprimere sia V^o che h^o in funzione dell'action value function e, calcolando il valore di $Q^o(x(t),u(t))$ per ogni possibile coppia stato/azione, è possibile determinare la policy ottima senza conoscere ne la funzione di reward ne quella di transizione (particolarità del Q-learning è appunto il fatto di essere model-free).
La policy ottima sarà dunque quella che selezionerà ad ogni istante di tempo l'azione $u(t)$ che massimizza $Q^o(x(t),u(t))$ con $x(t)$ stato corrente.
La funzione Q^o è calcolabile mediante *value iteration asincrona* dove ad ogni iterazione k si aggiorna il valore di una singola coppia x_k, u_k.
Partendo dunque da una $Q^o(x,u)$ iniziale, per $k = 0, 1, .. :$

- Si genera una coppia (x_k, u_k).
- Si effettua un passo di PD

$$Q_{\text{k+1}}(x(k),u(k)) =$$
$$r(x(k),u(k)) + \alpha max_{\text{u(t+1)}} Q_\text{k}(x(t+1),u(t+1)).$$

- Si lasciano invariati i valori:

$$Q_{\text{k+1}}(x,u) = Q_\text{k}(x,u)\ \forall (x,u) \neq (x_\text{k}, u_\text{k})$$

E si avrà che, sotto opportune ipotesi (tutti gli stati devono essere stati visitati un numero sufficiente di volte), l'algoritmo converge:

$$\lim_{k\to\inf} Q_\text{k}(x,u) = Q^o(x,u)$$

Nel caso in cui lo spazio stati/azioni risulti discreto e di dimensionalità trattabile, la funzione Q può essere espressa tramite una tabella deterministica tuttavia, in molti problemi reali, ciò non è possibile e si vanno ad usare tecniche di *Q-learning approssimato*.
Viene dunque introdotta una funzione approssimante che cercherà di essere il più vicina possibile alla Q-function ma che tuttavia non sempre garantisce la convergenza.

L' **idea** di base originaria è introdurre dei parametri $\omega(t)$ che vengono via via aggiornati tramite un passo di discesa del gradiente per migliorare di iterazione in iterazione la stima.

2. Deep Q-Learning (DQN)

Quando si ha a che fare con reti neurali per il calcolo approssimato della funzione Q, emergono 2 problematiche principali:

- il target della rete dipende dall'attuale vettore dei parametri $w(t)$; aggiornando la Q ad ogni istante temporale si va a modificare anche il target della rete, che può dare fastidio: introducendo la ricorsione nell'aggiornamento dei w si può andare incontro a fenomeni di **oscillazione/instabilità**.
- l'ordine con cui arrivano i dati non è completamente casuale, ovvero i dati successivi sono fortemente correlati tra loro, andando a perdere le buone proprietà di convergenza del gradiente stocastico.

Il **Deep Q-Learning** (Mnhin et al. 2015) si presenta come una variante del classico algoritmo di Q-learning con 3 contributi fondamentali che vanno a escludere le problematiche sopra esposte.

- come accennato, invece di una funzione Q tabulare, DQN utilizza le *multi-layer neural networks* per approssimare la funzione Q.
- invece di utilizzare una sola funzione Q per calcolare sia il target della rete sia la nuova stima, DQN usa due reti separate.
  Abbiamo dunque una rete neurale *target* e una rete neurale *online* e, nello specifico, per ogni esperienza $< x(t), u(t), r(t), x(t+1) >$ DQN usa la target network per calcolare il target $r(t) + \gamma max_{\text{u(t+1)}} \tilde{Q}(x(t+1), u(t+1); \tilde{w}(t))$ e usa la rete online per calcolare la stima $Q(x(t), u(t); w(t))$.
  Infine, il target e la stima sono usati per calcolare la loss:
  $[r(t) + \gamma max_{\text{u(t+1)}} \tilde{Q}(x(t+1), u(t+1) - Q(x(t), u(t); w(t))]^2$
- invece di usare l'ultima esperienza ottenuta per aggiornare la funzione Q, DQN salva le esperienze in una **replay memory** e usa dei mini-batch per processare le esperienze prese in maniera uniforme random per aggiornare i parametri della rete online usando la discesa del gradiente stocastico per minimizzare la loss.
  In tutto ciò la target network non è ottimizzata direttamente ma aggiornata periodicamente con i valori della rete online permettendo al training di procedere in maniera più stabile.

Algorithm 1: deep Q-learning with experience replay.
Initialize replay memory D to capacity N
Initialize action-value function Q with random weights θ
Initialize target action-value function $\hat{Q}$ with weights $\theta^- = \theta$
For episode = 1, M **do**
  Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$
  For t = 1,T **do**
    With probability ε select a random action a_t
    otherwise select $a_t = \text{argmax}_a Q(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in D
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from D
    Set $y_j = \begin{cases} r_j & \text{if episode terminates at step j+1} \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^-) & \text{otherwise} \end{cases}$
    Perform a gradient descent step on $\left(y_j - Q(\phi_j, a_j; \theta)\right)^2$ with respect to the network parameters θ
    Every C steps reset $\hat{Q} = Q$
  End For
End For

Figure 2: pseudocodice dell'algoritmo di DQN tratto da [1]. In questo caso θ rappresenta i pesi, ϕ lo stato e a le azioni.

Analizzando nel dettaglio l'algoritmo in figura 4 vediamo come è impostato su una procedura iterativa di appredimento fatta di *episodi* e *steps* per ogni episodio.
Ogni episodio prevede la generazione di una serie di azioni che, tramite apprendimento, tendono, auspicabilmente, a convergere verso la successione ottima di azioni per il raggiungimento di un certo obiettivo.
Ad ogni step viene selezionata un'azione che, in base a una certa probabilità, viene scelta casualmente oppure come l'azione che massimizza la funzione approssimante dell'action value function per lo stato corrente (ϵ-greedy policy).
L'azione viene poi eseguita e viene osservato il reward e lo stato futuro; questi dati di transizione vengono poi salvati nella *memoria replay*.
Successivamente vengono fatti degli aggiornamenti della funzione Q tramite minibatch sfruttando esperienze campioni estratte in maniera random uniforme dalla memoria.
Possiamo notare infine come, appunto, vengano usate due reti differenti; nello specifico, ogni C steps viene clonata la rete Q per ottenere una *target network* $\hat{Q}$ e viene usata quest'ultima per generare i successivi C aggiornamenti di Q.
Come esposto in precedenza, questa modifica porta notevoli vantaggi di stabilità rispetto a un algoritmo standard di Q-learning evitando fenomeni di oscillazione e divergenza della policy.
Infatti, generare i targets y_i con un insieme più vecchio di parametri aggiunge un ritardo tra il tempo che viene fatto

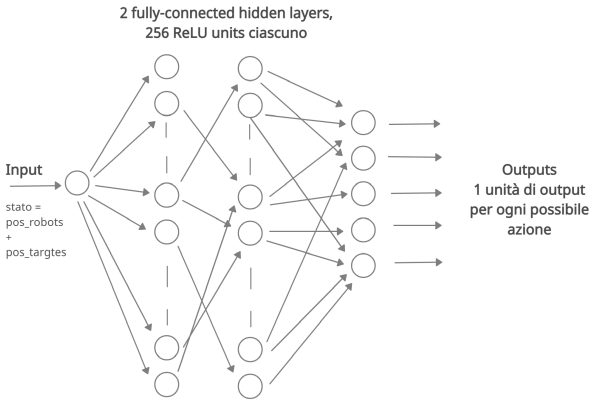

Figure 3: Struttura concettuale delle reti neurali utilizzate.

un aggiornamento della Q e il tempo che gli aggiornamenti influiscono sui targets.

3. RL multiagente

Si considerano N agenti indipendenti
$x(t+1) = f(x(t), u_1(t), ..., u_{\mathrm{N}}(t), \epsilon(t))$ in cui ogni agente osserva lo stato $x(t)$ o una sua parte $x_{\mathrm{i}}(t)$ e decide quale controllo applicare per massimizzare un obiettivo che dipende da una sommatoria dei rewards di tutti gli agenti.
Si distinguono tre casi: il caso **cooperativo** dove tutti gli agenti hanno lo stesso reward, il caso **competitivo** dove, nel caso $N = 2$, vale $r_1 = -r_2$ e infine il caso misto che si configura più complicato.
Ci sono due approcci principali per arrivare ad ottenere il controllo ad ogni istante : un approccio **globale** e un approccio **locale**.
L'approccio globale prevede, per ogni agente, l'apprendimento di una funzione Q che dipende anche dalle decisioni degli altri agenti ottenendo quindi una sorta di apprendimento centralizzato.
Questo approccio ha il problema della complessità crescente in modo esponenziale con il numero N di agenti.
Si ha inoltre necessità di un modello di decisione per gli altri agenti, dal momento che si conosce solo $x_{\mathrm{i}}, u_{\mathrm{i}}$ e gli altri u no.
Si può fare in casi particolari come nel caso cooperativo a informazione completa.
Questo tipo di approccio sarebbe stato più che valido per il problema affrontato in questo elaborato ma si è optato per il secondo approccio per avere maggiore scalabilità nei test con numero crescente di agenti.
L'approccio locale, a differenza di quello globale, prevede un indipendent learning con loss quadratica dove si considerano gli altri agenti come parte dell'ambiente.
In questo caso si deve conoscere semplicemente, per ogni agente i, $x_{\mathrm{i}}(t), u_{\mathrm{i}}(t), r_{\mathrm{i}}(t), x_{\mathrm{i}}(t+1)$.
La difficoltà in questo caso sta nel fatto che : se anche gli altri agenti stanno imparando l'ambiente generalizzato formato da ambiente+altri agenti, questo non è più **stazionario** ma dipende da t.
La non-stazionarietà dell'ambiente fà si che la convergenza del Q-learning non sia più garantita e in particolare può dare problemi con l'*experience replay* in quanto si può avere che la memoria non riflette più la corrente dinamica dell'ambiente che gli agenti dovrebbero imparare.
Infatti, riutilizzando esperienze obsolete si può confondere in continuazione la rete neurale e rendere il training instabile.
Vedremo in seguito alcune modifiche e se questa problematica diverrà rilevante nei test.

4. Prioritized Experience Replay

Recentemente è stata associata alle reti DQN una variante dell'experience replay denominata *"prioritized experience replay"* (PER).
In questo progetto è stata implementata e provata anche questa tecnica su cui sono stati fatti test e confronti con l'experience replay standard (UER).
[2]Experience replay, come abbiamo visto, permette agli agenti nel campo del reinforcement learning di ricordare e riusare le esperienze passate.
Le esperienze di transizione vengono campionate in maniera uniforme da una memoria di replay; tuttavia, questo approccio semplicemente replica le transizioni con la stessa frequenza con cui sono state originariamente osservate, senza tener conto della loro importanza e del loro impatto sul learning.
L'idea di base è dunque dare un grado di priorità maggiore per il campionamento a esperienze che risultano più significative ai fini del learning.
Questa tecnica risulta particolarmente utile per ottenere uno speed-up nelle prime fasi dell'apprendimento; infatti, nella fase di exploration, gli agenti si muoveranno in maniera random e sarà rarissimo il caso in cui si arrivi, nei limiti temporali, a un completamento del compito.
In questi rari casi si avrà che la differenza tra i risultati attesi (reward prettamente negativo e fallimento) e output effettivo (reward positivo e compito eseguito) sarà molto significativa portando a una probabilità molto alta che questa esperienza venga effettivamente campionata più volte.
In sostanza sfruttiamo più volte le esperienze significative cercando, in un certo senso, di far capire alla rete qual'è la direzione giusta.
Questa "differenza" di cui si parla è l'*errore TD* che è dato, appunto, dalla differenza tra il target e i valori stimati per questa esperienza.
In formule, abbiamo che p_{t}, la probabilità di campionamento risulta proporzionale a questa differenza:

$$p_t \propto (|r(t) + \gamma max_{u(t+1)}\tilde{Q}(x(t+1), u(t+1); \tilde{w}(t)) - Q(x(t), u(t); w(t))| + \epsilon)^{\omega}$$

dove ϵ è una costante piccola positiva che ci assicura che nessuna esperienza abbia una probabilità nulla di venir campionata.
L'esponente ω, invece, determina quanta priorità viene usata; con $\omega = 0$ siamo nel caso uniforme.
Per l'Atari benchmark suite di Bellemare et al.(2013), l'uso della prioritized experience replay ha portato sia a un learning più rapido sia a una miglior policy finale in vari giochi rispetto all'uniform experience replay.
A livello di programmazione, questa tecnica è stata implementata con una struttura dati differente da un semplice buffer di memoria, come nel caso dell'experience replay standard.
Infatti, ordinare tutte le esperienze in un buffer in base alla priorità non risulta efficiente in quanto abbiamo una complessità di $O(nlogn)$ per l'inserimento nella struttura dati e $O(n)$ per il campionamento.
Viene dunque usata una struttura ad albero chiamata **SumTree**; esso è fondamentalmente un albero binario, ovvero un albero dove ogni nodo parente ha al massimo due nodi figli con la particolarità che il nodo parente ha valore pari alla somma dei valori dei nodi figli.
Questa struttura è particolarmente comoda e efficiente per effettuare il campionamento di dati ad alta priorità portando la complessità a $O(logn)$.

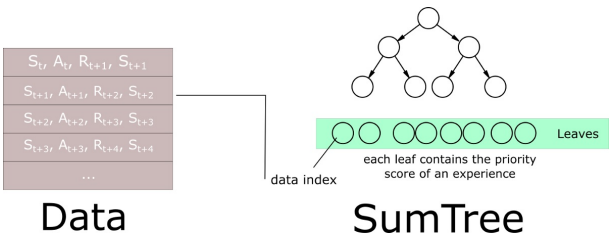

Figure 4: struttura dei dati nel caso del prioritized experience replay.

5. Funzioni e sezioni salienti del codice

Come accennato, il codice è scritto in Python per preferenze personali e per il grande supporto di librerie di machine learning quali keras e tensorflow.
Vi sono vari file python all'interno del progetto che si occupano di aspetti implementativi distinti e in seguito verrà data una breve documentazione.

main.py: è il file principale per l'esecuzione e si occupa principalmente della fase di configurazione, inizializzazione e inserimento dei parametri.
Ha al suo interno la classe *Setup* che viene inizializzata con alcuni parametri fondamentali che definiscono, ad esempio, il numero di episodi, il numero di steps per episodio, il numero di steps tra gli update della target network (il parametro sopra indicato con C), ecc..
Questa classe ha inoltre il metodo *run()* che, chiamando in causa classi e funzioni presenti negli altri file, dà il via alla simulazione e all'apprendimento.

pygame_settings.py: questo script si occupa della gestione dell'interfaccia grafica tramite la libreria *pygame*.
La classe presente, *Game*, si occupa di inizializzare la griglia e, attraverso la funzione *render()*, di eseguire il display e l'aggiornamento grafico per ogni step.
Le funzioni chiave poi sono *update_positions()* e *step()*; la prima, dato in ingresso le azioni da compiere, verifica la validità di quest'ultime, le eventuali collisioni e aggiorna gli indici posizionali degli agenti.
Abbiamo infine la funzione *reset()* che reinizializza il "gioco" al termine di un episodio.

experience_replay.py: contiene semplicemente la classe *Memory* che fà da buffer per le esperienze.
Ha una capacità fissata tramite parametro e le nuove esperienze vengono via via aggiunte in coda tramite la funzione *remember()*.
Vi è inoltre una funzione *sample()* che estrae in maniera random uniforme n esempi di esperienza nella memoria con n parametro in ingresso.

sum_tree.py: contiene l'omonima classe *SumTree* che implementa la struttura dati ad albero precedentemente descritta.
Sono presenti funzioni classiche delle strutture dati quali *add()* e *get()* ma anche la funzione *update()* che permette, in caso di cambiamenti nella struttura, di aggiornare i dati e propagare dalla radice ai nodi.

prioritized_experience.py: in questo file è presente, analogamente all'experience replay standard, una classe *Memory* che stavolta utilizza il SumTree.
Particolare è, appunto, la funzione *get_priority()* che, dato l'errore TD in ingresso, l'epsilon fissato dalla classe e il fattore di prioritizzazione, rende la priorità corrispondente a un' esperienza che poi viene usata nel metodo *update()* per invocare l'aggiornamento sulla struttura dati.
In modo analogo, la funzione *remember()* invoca l'add sulla memoria e *sample()* ne estrae i campioni.

logic.py: è il file che contiene la classe *Brain* responsabile del modulo di intelligenza artificiale.
La classe contiene funzioni classiche quali *build_model()* che si occupa, appunto, di impostare la rete e definire un

modello.
A tal proposito, è stata utilizzata una rete a 3 strati dove i primi due sono *fully-connected*; ognuno dei quali comprensivo di 256 unità rettificatrici.
Lo strato di output è anch'esso fully connected ma lineare e rende un singolo output per ogni possibile azione.
Abbiamo poi ovviamente la funzione *train()* che esegue il fit del modello con x input della rete e y output e la funzione *predict()* che, preso in ingresso lo stato corrente rende in uscita le predizioni per la target network e la local network.
Queste funzioni verranno poi utilizzate dalla classe del successivo file che definisce gli agenti.

agent.py: questo file contiene, appunto, la classe *Agent*; ogni "robot" avrà dunque un'istanza separata di questa classe.
Questa classe viene inizializzata sia con un'istanza di *Memory* che con un *Brain* per sfruttare le loro funzioni.
Infatti abbiamo metodi come *observe()* che si occupa di fare lo store delle esperienze nella memoria di replay e il metodo *greedy_actor()* che, utilizza un parametro in decadimento, *epsilon*, per decidere, ad ogni step se deve essere effettuata un'azione random o quella che massimizza l'action value function approssimata predetta dal brain.
Il metodo *decay_epsilon()* decrementa, appunto, il parametro epsilon via via che proseguiamo nel training in base a un parametro *max_exploration_step*.
Questo ci permette di avere una prima fase di esplorazione dove gli agenti si muovono in maniera prevalentemente randomica in modo da coprire gran parte dei possibili stati per poi affinarsi sempre di più fino ad avere un comportamento tutt'altro che random ma basato sull'esperienza e sul learning eseguito.
Infine, la funzione *replay()* si occupa di estrarre dalla memoria un campione di esperienze da processare in un batch, trovare i targets e eseguire con quest'ultimi il train del brain.

6. Descrizione dettagliata del problema

Il problema base analizzato è strutturato come in figura 5; abbiamo, inizialmente due robot e vogliamo che questi raggiungano in contemporanea due celle target.
Il setting è chiaramente ispirato alla situazione descritta nel paper [3] "Multi-agent Reinforcement Learning: An Overview".
Il compito è ritenuto completo quando entrambi i robot occupano una cella target raggiungendo quindi una determinata formazione o supponendo che abbiano premuto in simultanea un pulsante.

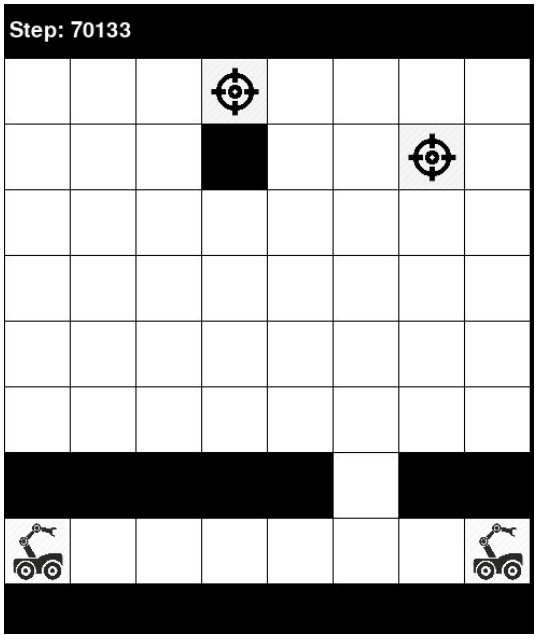

Figure 5: struttura di partenza del problema analizzato.

Le azioni possibili per ogni robot sono: "UP", "DOWN", "LEFT", "RIGHT" e "STAY" e a ogni step si esegue un azione per ogni robot in successione.
le celle nere rappresentano celle non navigabili e quindi ostacoli; vi è dunque una sorta di "imbuto" iniziale che spesso relega i robot in una zona lontana dai target e che può portare a una contesa per il passaggio.
E' inoltre presente un altro ostacolo davanti a un target in alto per evitare un accesso diretto.
Se viene selezionata un'azione che porta a una collisione con una cella non navigabile o con l'altro robot, si ottiene essenzialmente uno "STAY" e dunque non si modifica lo stato.
Gli episodi inoltre, sono stati limitati a 200 steps; se il compito non è completo entro questi steps, l'episodio termina con un fallimento, la situazione si resetta e parte l'episodio successivo.

6.1. gestione dei rewards

Per quanto riguarda i reward degli agenti, inizialmente sono stati gestiti nel seguente modo: ad ogni step, si assegna un reward di -1 se l'agente non è sul target, 0 altrimenti.
Se invece entrambi gli agenti sono sul target il compito è completo e viene assegnato un reward di +200.
Da questa configurazione si ha che gli episodi con reward negativo sono sostanzialmente dei fallimenti mentre il

reward positivo indica il successo; in particolare il ”path ottimo” ha un reward di 179.
Inizialmente non è stata implementata una penalità in caso di collisione in quanto, selezionando il path ottimo a fine addestramento, si supponeva di ottenere una sequenza di azioni a 0 collisioni; ciò tuttavia, non è risultato del tutto vero in quanto si può ottenere il reward ottimo anche se l'agente che arriva sul target più in alto esegue una collisione con l'ostacolo adiacente mentre aspetta che l'altro agente giunga sull'altro target.
Non è raro infatti che si converga proprio a questa sequenza di azioni.
Per ovviare a questo comportamento indesiderato, è stata resa la funzione di reward più flessibile e introdotta una penalità di -1 a collisione nelle fasi avanzate dell'apprendimento ottenendo così il comportamento desiderato e la convergenza alla sequenza senza collisioni di reward 179.

7. Specifiche rete neurale e iperparametri

La struttura della rete neurale è piuttosto semplice; non vengono utilizzati strati convoluzionali in quanto l'input alla rete non sono immagini ma semplici informazioni di stato e quindi un vettore (posizione agenti e posizione dei targets).
Si è considerato dunque una rete neurale a 3 strati dove i primi due sono fully-connected e dotati di 256 unità rectifier a testa mentre l'output è fully-connected lineare con un singolo output per ogni possibile azione.
Nello specifico, ogni agente è dotato di due reti neurali, online e target, per approssimare la propria funzione Q e di un buffer di memoria (o sumtree nel caso della PER) per lo store delle esperienze.
La capacità di memoria è stata impostata in alcuni test a 50.000 steps mentre in altri a 10.000; è stata impostata con queste limitazioni in modo che, una volta superata la capacità, si vadano a scartare le esperienze più vecchie e obsolete e si includano quelle nuove (questo, come accennato in precedenza, aiuta ad ovviare a una problematica dell'approccio locale nel RL multiagente).
Sono impostati 1000 steps come l'intervallo tra un aggiornamento dei pesi e l'altro della target network mentre i target e il modello vengono aggiornati ogni 4 step.
I batch con cui vengono estratte le esperienza dalla memoria e addestrato il modello hanno dimensione 64.
Per quanto riguarda la loss del modello, è stata usata la **huber loss** con delta pari a 1 mentre come ottimizzatore è stato provato sia *RMSprop* che *Adam* con un learning rate di 0.00005 ma quest'ultimo è risultato più efficace ed è quindi stato adottato nei test successivi.
Infine, per quanto riguarda l'esecuzione con PER, è stato usato un fattore di prioritizzazione di 0.5.

8. Test effettuati

Inizialmente sono stati eseguiti dei test molto limitati a 800 episodi in cui è stato provato sia l'approccio con l'experience replay uniforme che quello con priorità.
E' stato notato che, mediamente, si ha maggior probabilità e una convergenza più rapida con la prioritizzazione; tuttavia, nei casi in cui l'experience replay standard arriva a convergere, ha una stabilità molto maggiore.
Questa situazione si inizia a comprendere osservando la figura 6 dove abbiamo uno dei casi di convergenza rapida dell'UER a confronto con la PER.

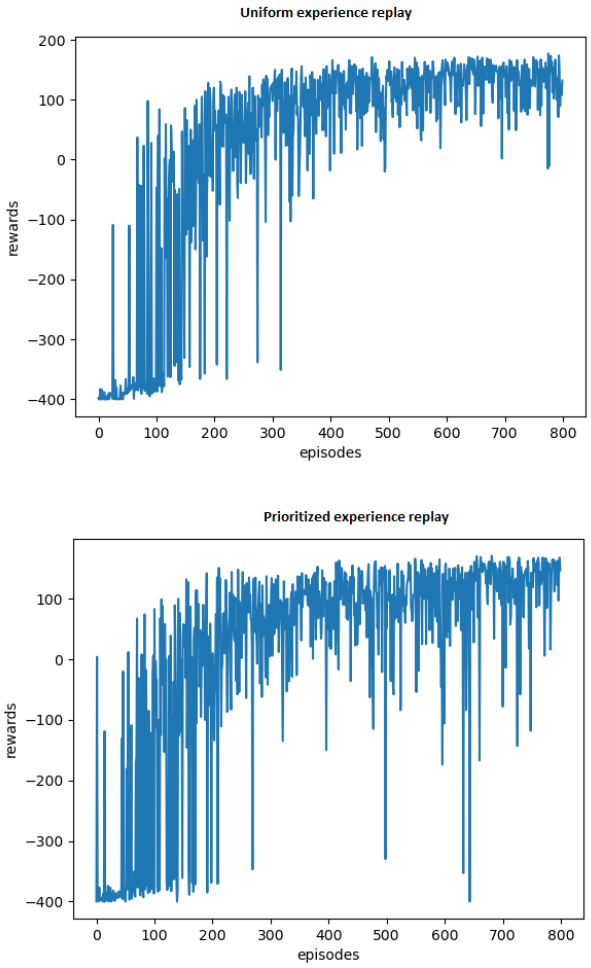

Figure 6: UER vs PER in un caso di addestramento limitato a 800 episodi.

Si nota come, andando oltre i 300 episodi circa, l'experience replay uniforme non fallisce il compito in nessun episodio pur con qualche oscillazione dove il reward è prossimo a 0.
Per quanto riguarda l'experience replay con priorità, invece, vi sono alcuni casi dove l'episodio si conclude con un fallimento e in generale, più casi di successo con reward basso.
La problematica non è dovuta a una scarsità di esempi in memoria o al learning limitato in quanto dai test più estesi a 5000 episodi, l'instabilità rimane presente e anzi, si mostra in maniera molto più marcata.

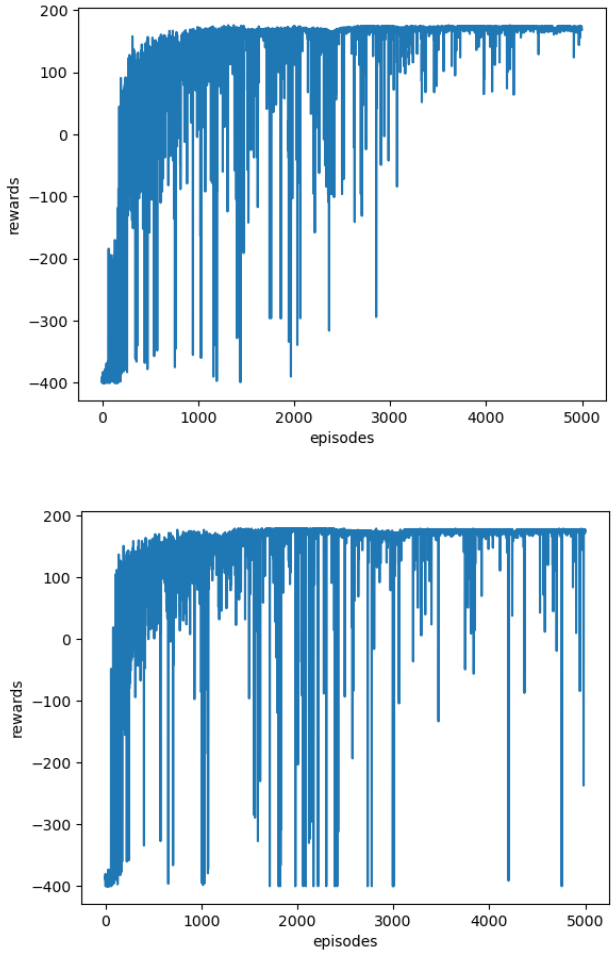

Figure 7: UER vs PER in un caso di addestramento a 5000 episodi.

Possiamo osservare la situazione in figura 7 dove, nonostante un gran numero di iterazioni e valori del reward oramai ottimi, persistono casi di instabilità dove l'episodio fallisce.
L'experience replay uniforme invece, persiste nella sua condizione di maggiore stabilità osservata nel test a 800 episodi.
La motivazione risiede probabilmente nel fatto che, tramite PER, alla lunga, si può andare incontro ad un *overfitting* prioritizzando sempre le stesse esperienze che non vengono scartate anche se obsolete portando a galla una delle problematica dell'approccio RL locale.
L'idea di base che è emersa per i prossimi test è stata dunque quella di combinare la capacità del prioritized experience replay di avere un learning più rapido e una curva più ripida nelle prime fasi con la stabilità data dall'experience replay standard.
Per ottenere un effetto simile, è stato usato un approccio *ibrido* dove, per i primi 1500 episodi viene utilizzata la tecnica dell'experience replay con priorità, mentre in seguito si passerà a un experience replay uniforme per ottenere stabilità.
l'approccio implementato ha dato riscontri più che positivi e, come è possibile vedere dal grafico in figura 8 , è, mediamente, il miglior approccio tra quelli considerati per questo tipo di problemi garantendo la policy ottima nel minor tempo possibile e una miglior stabilità generale a lungo termine.

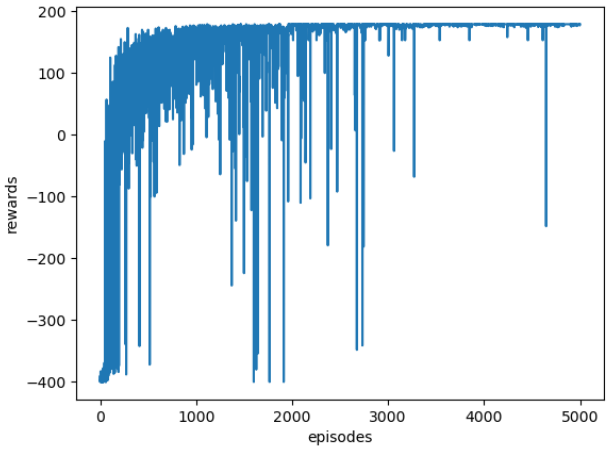

Figure 8: risultato nel test a 5000 episodi dell'approccio ibrido PER + UER.

Un test ulteriore che è stato fatto è stato di aggiungere un altro robot agente e un'altra posizione target in modo da complicare il compito e valutare le capacità di *scaling* dell'algoritmo.
La situazione di partenza è descritta in figura 9 con l'aggiunta di un robot a sinistra oltre il "collo di bottiglia" ma che deve spostarsi verso destra andando spesso a intersecare la traettoria degli altri robot.
I risultati, visibili in figura 10, ci mostrano come l'approccio funzioni abbastanza bene seppur con una prima fase di addestramento più travagliata e una curva che generalmente impiega più tempo a salire.
La lentezza di partenza deriva dal fatto che, almeno inizialmente, gli agenti si muovono in modo prevalentemente randomico e il completamento del compito a 3 agenti diventa particolarmente improbabile finchè non si riduce via via la randomicità.
Aumentando il numero degli agenti e dei target, la problematica diviene inevitabilmente sempre più evidente; una soluzione potrebbe essere modificare di conseguenza la funzione di reward impostando dei "reward intermedi" che ci permettono di avvicinare passo passo al completamento del compito per poi essere eventualmente tolti in seguito per evitare una convergenza verso dei massimi locali.

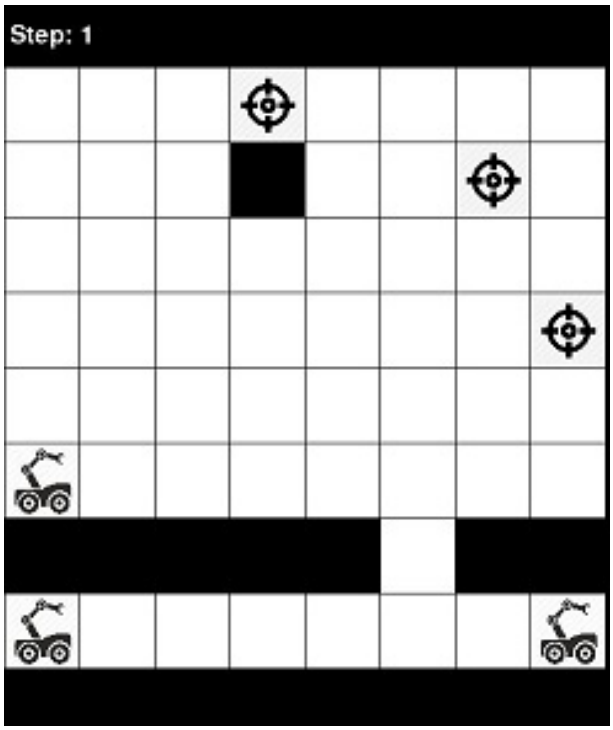

Figure 9: struttura di partenza del problema a 3 agenti.

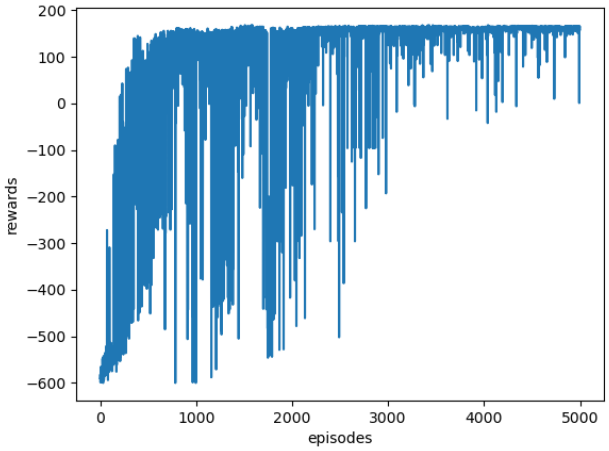

Figure 10: risultato nel test a 5000 episodi dell'approccio ibrido con 3 agenti e 3 target.

9. Conclusioni e sviluppi futuri

Per concludere, in questa relazione è stata presentata un'architettura di base per lo studio e la risoluzione di compiti cooperativi multiagente tramite reinforcement learning che può essere facilmente estesa e modificata per andare a coprire un'ampia gamma di problemi.
Dai test effettuati emerge chiaramente l'efficacia del Q-learning nel contesto analizzato in quanto già dopo 1000 episodi e poco tempo di training, si riesce a convergere, approsimativamente, alla policy ottima e dunque alla sequenza di azioni ottima.
Si è visto inoltre come, in alcuni casi particolari, si possa utilizzare dei rewards dinamici invece che statici per raffinare ulteriormente la policy ad un certo punto dell'apprendimento (vedesi il caso delle collisioni esposto in precedenza).
E' emerso inoltre come l'approccio di base con l'experience replay sia in generale efficace e un must per questo tipo di problemi ma tuttavia migliorabile con la prioritizzazione nelle prime fasi e alcuni accorgimenti.
Ulteriore miglioramento ma non trattato all'interno della relazione in quanto non strettamente necessario, può essere la modifica del codice e l'implementazione del DDQN (Double Q learning) [4] che dovrebbe raffinare ancora di più l'algoritmo.

References

[1] Volodymyr Mnih et al. *Human-level control through deep reinforcement learning*. 2015.

[2] Tom Schaul et al. *PRIORITIZED EXPERIENCE REPLAY*. 2016.

[3] Lucian Busoniu, Robert Babuska, and Bart De Schutter. "Multi-agent Reinforcement Learning: An Overview". In: vol. 310. July 2010, pp. 183–221. ISBN: 978-3-642-14434-9. DOI: 10.1007/978-3-642-14435-6_7.

[4] Hado van Hasselt, Arthur Guez, and David Silver. *Deep Reinforcement Learning with Double Q-learning*. 2015. arXiv: 1509.06461 [cs.LG].