Unity Machine Learning Agents

Marco Faleri

1 Abstract

Il lavoro si prefissa lo scopo di sperimentare l'utilizzo del (deep) reinforcement learning nello sviluppo di intelligenze artificiali per videogiochi, tramite la libreria ml-agents di Unity, nota piattaforma di sviluppo di videogiochi. In particolare tale lavoro si delinea nei seguenti punti:

- Introduzione al deep reinforcement learning ed ai Policy gradient methods. Algoritmo PPO.
- Libreria ml-agents di Unity
- Esperimenti

2 Deep reinforcement learning

Per reinforcement learning, o apprendimento per rinforzo, si intende la branca dell'apprendimento automatico che attua sistemi (*agenti*) in grado di adattarsi alle mutazioni degli *ambienti* in cui sono immersi, attraverso un sistema di *ricompense* (reward), che valuti le loro prestazioni e indirettamente guidi tali agenti verso il comportamento ottimale.

Sostanzialmente, quando si parla di apprendimento per rinforzo, si parla di un'agente che opera all'interno di un ambiente, sul quale agisce mediante azioni, che vengono decise in base alle osservazioni e alle ricompense ricevute. Iterando tali passi, al generico tempo t:

1) L'agente esegue nell'ambiente un'*Azione* A_t
2) L'agente riceve dall'ambiente un'*Osservazione* O_{t+1}
3) L'agente riceve dall'ambiente una *Ricompensa* R_{t+1}

È da notare come a ciascun tempo t avvenga un processo di *decision making*, attraverso il quale l'agente sceglie l'azione opportuna. Onde evitare ambiguità, occorre ben definire ciascun termine che occorre quando si tratta un problema di apprendimento per rinforzo.

Definizione 1 (Ricompensa). Una *Ricompensa* (reward) R_t è un segnale scalare di feedback, il quale indica all'agente l'ottimalità (spesso proporzionale alla vicinanza a uno o più *goal*) della situazione in cui quest'ultimo si trova all'istante t.

L'obiettivo ultimo dell'agente è di fatti massimizzare le ricompense che riceve durante l'interazione con l'ambiente, sia essa *episodica* (ovvero con uno stato terminale) o meno. Non mancano complicazioni: in primis perchè delle azioni potrebbero avere degli effetti a lungo termine, in seconda istanza perchè i reward potrebbero essere dati soltanto in alcuni momenti, ad esempio al compimento finale dell'obiettivo.

L'apprendimento per rinforzo si basa sull' *Ipotesi di ricompensa*: ovvero sul fatto che ciascun *obiettivo* dell'interazione (ovvero sostanzialmente quello che vogliamo che il nostro agente faccia), sia ottenibile (o sia ottenibile più efficacemente) tramite la massimizzazione della funzione cumulativa delle ricompense attese.

Definizione 2 (Osservazione ed Azione). A questo punto di trattazione, per Osservazione si intende la rappresentazione dell'ambiente che l'agente riceve, mentre per Azione possiamo intendere un segnale emesso dall'agente verso l'ambiente.

Definizione 3 (Storia e Stati). La sequenza di tutte le Azioni, Osservazioni e Ricompense prende il nome di *Storia*. Funzione della storia sono gli *Stati*, dell'ambiente o dell'agente, che raccolgono le loro informazioni private.

Definizione 4 (Stato dell'ambiente). Si definisce *Stato dell'ambiente* la rappresentazione interna all'ambiente.

Gli stati dell'ambiente raccolgono l'informazione che l'ambiente utilizza per generare rewards e osservazioni.

Definizione 5 (Stato dell'agente). Si definisce *Stato dell'agente* la rappresentazione interna all'agente.

Gli stati dell'agente raccolgono l'informazione che l'agente utilizza per determinare l'azione successiva.

Chiaramente stato dell'agente e dell'ambiente possono differire, sia l'ambiente che l'agente possono avere informazioni *private*, ad esempio perchè tramite le Osservazioni non tutte le informazioni dell'ambiente vengono veicolate verso l'agente.

Definizione 6 (Ambienti completamente o parzialmente osservabili). L'ambiente è *completamente osservabile* se l'agente ha accesso a tutte le informazioni (rilevanti) riguardo lo stato dell'ambiente; in caso contrario si dice che l'ambiente è *parzialmente osservabile*.

Sostanzialmente, si parla di ambiente completamente osservabile quando un'agente riceve tramite le osservazioni tutta la conoscenza possibile riguardo l'ambiente, ovvero quando la rappresentazione ricevuta dall'ambiente corrisponde all'ambiente stesso.

Per semplicità di trattazione, supporremmo ambienti completamente osservabili.

Un altra distinzione che è importante fare fin da subito è tra approcci *model-based* e *model-free*: nell'approccio model-based l'agente cerca di apprendere un modello dell'ambiente, che utilizza per prevedere cosa succederà

M. Faleri, Metodi Statistici per l'Apprendimento, A/A 2017-2018, Università degli studi di Milano, via Celoria 14, Milano, Italia
E-mail: marco.faleri@studenti.unimi.it

(importantissimo nel planning), mentre nell'approccio model-free l'agente non si preoccupa di modellare il comportamento dell'ambiente, quanto più semplicemente di valutare quale sia la migliore azione in ogni stato.

Nel processo di sequential decision making che caratterizza tutto il problema di apprendimento per rinforzo, si possono distinguere due fasi:

- Learning: l'agente interagisce con l'ambiente, con il duplice scopo di conoscere l'ambiente (*Exploration*), inizialmente sconosciuto, e di apprendere il comportamento ottimale (correlato all'*Exploitation*, il concentrarsi su "punti" caratterizzati da un buon reward, per ottimizzare).
- Planning: una volta che l'agente ha acquisito un *modello* dell'ambiente, questi interagisce con questo modello, senza azioni esterne, per migliorare ulteriormente la policy ed il proprio comportamento (attraverso una ricerca, *Tree-search* ad esempio.)

Tali fasi valgono in generale per gli algoritmi model-based, mentre per gli algoritmi model-free il tutto risulta semplificato: rimane ovviamente il learning, che tuttavia viene dedicato soltanto all'apprendimento di un comportamento ottimale (più o meno direttamente) e non di un modello, e non avendo un modello è impossibile fare del vero e proprio planning.

2.1 Markov Decision Process

Spesso i problemi di apprendimento (Model-Based) sono riconducibili a *Processi Decisionali Markoviani*, ovvero a situazioni in cui tutto quello che succederà dipende soltanto dallo stato corrente, e possiamo pertanto ignorare la Storia. (Il futuro è indipendente dal passato, dato il presente).

Definizione 7 (Proprietà di Markov). Uno stato S_t è *Markoviano* se e soltanto se (con un lieve abuso di notazione)

$$\mathbb{P}(S_{t+1}|S_t, S_{t-1}, \ldots, S_0) = \mathbb{P}(S_{t+1}|S_t) \tag{1}$$

Quando si estende la proprietà di Markov ad una sequenza di stati, si ottiene un *Processo Markoviano*

Definizione 8 (Markov Process). Un *Processo Markoviano* è una coppia (S, P) in cui:

- S è un insieme (finito) di stati
- P è una matrice probabilistica di transizione tra stati, definita nel seguente modo:

$$P_{\text{ij}} = \mathbb{P}[S_{\text{t+1}} = s_j|S_t = s_i] \tag{2}$$

A questo punto, aggiungendo i Reward e le Azioni, otteniamo i Markov Decision Process

Definizione 9 (Markov Decision Process). Un *Processo Decisionale Markoviano* è definito da una tupla (S,A,P,R,γ)

- S è un insieme finito di stati
- A è un insieme finito di azioni
- P è una matrice probabilistica di transizione tra stati, definita nel seguente modo:

$$P^a_{ij} = \mathbb{P}[S^a_{t+1} = s_j|S_t = s_i, A_t = a] \tag{3}$$

- R è una funzione di ricompensa, definita nel seguente modo:

$$R^a_s = \mathbb{E}[R_{t+1}|S_t = s, A_t = a] \tag{4}$$

- γ è un fattore di sconto (discount factor) $\gamma \in [0, 1]$

Per meglio compendere il ruolo delle ricompense e, soprattutto, del fattore γ è opportuno introdurre la nozione di *Ritorno*.

Definizione 10 (Ritorno). Il *Ritorno* (Return) G_t è la somma, scontata mediante γ, delle ricompense, a partire dall'istante t

$$G_t = R_{t+1} + R_{t+2} + \ldots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \tag{5}$$

Come possiamo vedere pertanto, il fattore γ soppesa il valore delle ricompense future nel calcolo del Ritorno attuale.

Strettamente connessi al concetto di ricompense e di Ritorno ci sono le funzioni di valore, che si utilizzano per valutare stati o azioni. Tali funzioni stimano le ricompense future, a partire da una certa azione o stato.

Definizione 11 (State value function e Action value function). Le *State Value Function* $v_\pi(s)$o *Action Value Function*$q_\pi(a)$ rappresentano il Ritorno atteso a partire da uno stato o da un'azione, seguendo la policy π

$$v_\pi(s) = \mathbb{E}_\pi[G_t|S_t = s] \tag{6}$$

$$q_\pi(a) = \mathbb{E}_\pi[G_t|S_t = s, A_t = a] \tag{7}$$

Ma cosa sarebbe la policy? La policy è ciò che definisce e determina il comportamento dell'agente, a seconda dello stato in cui quest'ultimo si trova.

Definizione 12 (Policy). La policy π è una distribuzione sulle azioni, condizionate dagli stati:

$$\pi(a|s) = \mathbb{P}[A_t = a|S_t = s] \tag{8}$$

2.2 Il problema

A questo punto, avendo definito gli elementi chiave, possiamo giungere al cuore del problema dell'apprendimento per rinforzo: la ricerca di una policy ottimale.

Definizione 13 (Optimal State Value Function). La *Optimal State Value Function* $v_*(s)$ è la massima state value function tra tutte le policy.

$$v_*(s) = \max_\pi v_\pi(s) \tag{9}$$

Definizione 14 (Optimal Action Value Function). La *Optimal Action Value Function* $q_*(s)$ è la massima action value function tra tutte le policy.

$$q_*(s) = \max_\pi q_\pi(s) \tag{10}$$

Così come per le value function, è possibile definire anche delle policy ottimali.

Definizione 15 (Optimal Policy). Una policy π_* è ottimale se

$$v_{\pi_*}(s) \geq v_\pi(s), \forall \pi, v, s \in S \tag{11}$$

Chiaramente, una policy ottimale ottimizza anche le action value function. Allo stesso modo, è facile dedurre che:

- $v_{\pi_*}(s) = v_*(s), \forall v, s \in S$
- $q_{\pi_*}(s,a) = q_*(s,a), \forall s \in S, a \in A$

Per ottenere tale policy, la quale è facile dedurre che massimizzi il Ritorno atteso, ci sono tre metodi principali:

- Ricercare direttamente q_* o v_*, sapendo i quali si può facilmente ottenere una policy ottimale (ad esempio utilizzando un approccio $\epsilon - greedy$).
- Parametrizzare direttamente la Policy e ricercare la policy ottimale (ovvero che massimizza la funzione che misura le performance), tramite metodi di ascesa del gradiente (*Policy Gradient Methods*).
- Approcci misti, come quello Actor-Critic

2.3 Policy Gradient Methods

A questo punto, tracciamo un breve percorso che ci porti all'algoritmo di *Proximal Policy Optimization* (PPO), che è l'algoritmo offerto dalla libreria Unity Ml-agents. Tale algoritmo fa parte dei policy gradient methods, pertanto partiremo subito parametrizzando la policy, come accennato poco sopra.

$$\pi_\theta(s,a) = \mathbb{P}[a|s,\theta] \tag{12}$$

A questo punto ci interessa comprendere quali siano i"migliori" valori di θ. Per tale scopo ci serve un metodo per misurare la qualità di una policy: le Policy Objective Functions $J(\theta)$. In ambienti episodici possiamo considerare lo *start value*

$$J_1(\theta) = v_{\pi_\theta}(s_1) \tag{13}$$

In ambienti continui, si pu utilizzare il valore medio:

$$J_2(\theta) = \sum_s d^{\pi_\theta}(s) v_{\pi_\theta}(s) \tag{14}$$

Dove $d^{\pi_\theta}(s)'$ rappresenta una distribuzione probabilistica degli stati.

Una volta scelta la $J(\theta)$, si fa ottimizzazione tramite l' ascesa del gradiente: seguendo ovvero la direzione indicata da $\alpha \nabla J(\theta)$, in cui α è parametro di dimensione del singolo passo di ascesa.

Risulta un grande problema in questo: come possiamo stimare il gradiente di π rispetto a θ quando il gradiente dipende fortemente dagli effetti che i cambiamenti della policy hanno sulla distribuzione degli stati? Per quanto riguarda le azioni, conoscendo lo stato in cui ci si trova, è semplice dedurre gli effetti dei cambiamenti di θ sulla scelta di queste, questo non è vero per gli stati: come abbiamo accennato sopra, gli stati sono spesso determinati dalle osservazioni che si fanno dall'ambiente, e di conseguenza da come si comporta l'ambiente. A meno di avere un modello (cosa che complica non banalmente l'approccio), la funzione di "evoluzione" dell'ambiente è sconosciuta.

Tale problema viene risolto tramite il Policy Gradient Theorem, il quale ci permette di lavorare comodamente con il gradiente di ciascuna objective function, dandoci un espressione analitica che non coinvolge la distribuzione degli stati.

Teorema 16. Data una policy differenziabile $\pi_\theta(s,a)$. Data una qualsiasi delle policy objective funcions: J_1, J_2.

Il Policy Gradient risulta essere:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}[\nabla_\theta log \pi_\theta(s,a) Q_{\pi_\theta}(s,a)] \tag{15}$$

Tale teorema sostanzialmente definisce il metodo di aggiornamento stocastico utilizzato dall'algoritmo baseline del settore: REINFORCE.

Algorithm 1 REINFORCE

Inizializza arbitrariamente il vettore θ dei parametri.
for all episodes $\{s_1, a_1, r_2, \ldots, s_{T-1}, a_{T-1}, r_T\} \sim \pi_\theta$ **do**
  for $t = 1 \ldots T-1$ **do**
    $\theta \leftarrow \theta + \alpha \gamma^t \nabla_\theta log \pi_\theta(s_t, a_t) G_t$
return θ

Da notare come nell'algoritmo si utilizzi G_t in luogo di $Q_{\pi_\theta}(s,a)$. Questo deriva dal semplice fatto che $Q_{\pi_\theta}(s,a) = \mathbb{E}_{\pi_\theta}[G_t|S_t, A_t]$ (Pertanto, possiamo "togliere" il valore atteso, essendo dentro ad un altro valore atteso, nel policy gradient.)

La direzione dell'aggiornamento, in tale algoritmo, viene indicata da G_t. Utilizzare quindi un qualcosa che dipende dal "futuro" richiede ambienti episodici, ed assomiglia molto da vicino alle tecniche MonteCarlo: in quanto ci occorre "simulare" tutto l'episodio per assegnare opportunamente un valore a G_t. Tale algoritmo soffre di problemi di alta varianza, e viene pertanto generalizzato introducendo un baseline. Tale baseline, che viene sottratto a G_t, può essere qualsivoglia funzione, purché non sia funzione delle azioni (per motivi che esulano da questa relazione, serve per non intaccare il valore atteso ma soltanto la varianza). Solitamente ciò consiste nel sostituire a G_t una funzione $\hat{A}_t$ (*Advantage function*) della forma: $G_t - \hat{v}(S_t)$.

Intuitivamente la funzione $\hat{v}(s)$ è un buon baseline in quanto aiuta a discriminare il valore delle azioni in ogni stato: se pensiamo ad esempio ad uno stato *buono*, questo sarà caratterizzato da molte azioni similmente *buone*, e lo stesso vale in uno stato *cattivo*; $\hat{v}(s)$ risulta quindi ottimo perché si adatta alla situazione in cui si trova, risultando enormemente più performante di baseline costanti.

Tuttavia tale approccio fa nascere la necessità di apprendere $\hat{v}(s)$, il quale può essere appreso utilizzando lo stesso approccio usato per apprendere la policy. Pertanto introduciamo la funzione $\hat{v_\omega}(s)$, in cui ω è il vettore dei parametri.

Algorithm 2 REINFORCE con Baseline

Inizializza arbitrariamente il vettore θ dei parametri.
Inizializza arbitrariamente il vettore ω dei parametri.
for all episodes $\{s_1, a_1, r_2, \ldots, s_{T-1}, a_{T-1}, r_T\} \sim \pi_\theta$ **do**
  for $t = 1 \ldots T-1$ **do**
    $G_t \leftarrow$ Ritorno dallo step t
    $\delta \leftarrow G_t - \hat{v}_\omega(s_t)$
    $\omega \leftarrow \omega + \alpha_\omega \delta \gamma^t \nabla_\omega \hat{v}_\omega(s_t)$
    $\theta \leftarrow \theta + \alpha_\theta \delta \gamma^t \nabla_\theta log \pi_\theta(s_t, a_t) G_t$
return θ

Da notare i diversi $\alpha_\theta, \alpha_\omega$ i quali sono rispettivamente lo step-size per θ e ω.

È importante sottolineare che in seguito l'advantage function è stata migliorata per evitare di collassare al caso

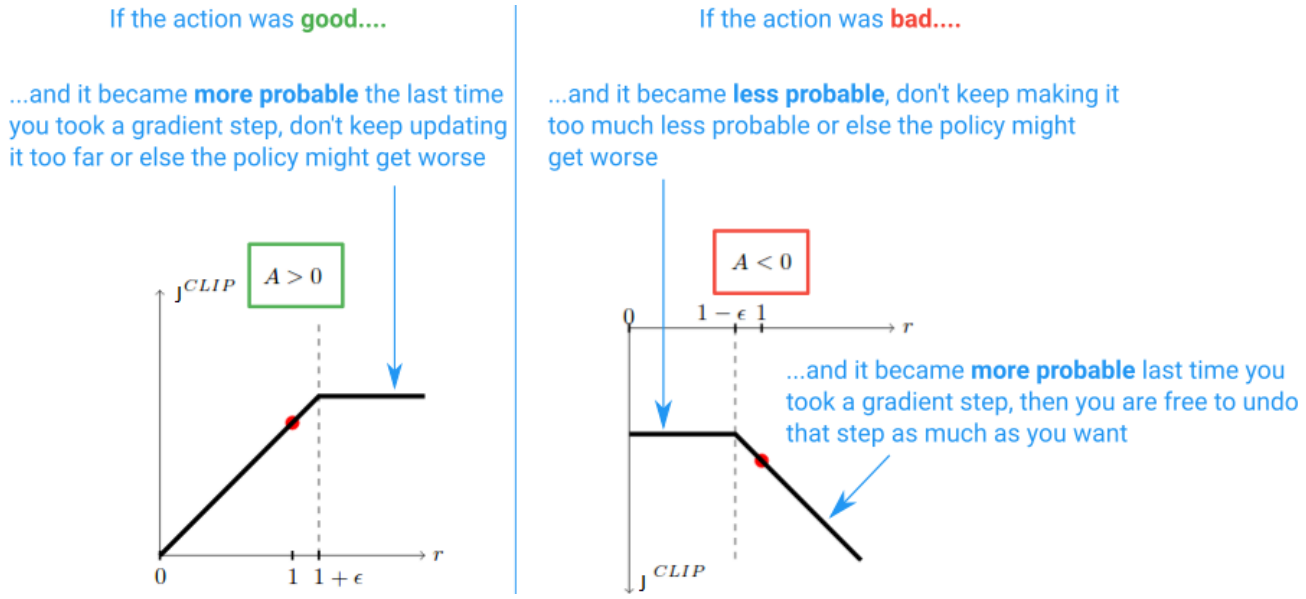

Fig. 1: Funzionamento del clipping in un singolo timestep. Sulla sinistra si ha un Advantage positivo, sulla destra un Advantage negativo.

Monte-Carlo, introducendo il parametro λ, e troncando ad un certo punto il calcolo di G_t (andando pertanto a produrne una stima). In particolare, la Generalized Advantage Estimation (GAE) è sostanzialmente una media pesata esponenzialmente, e si definisce nel seguente modo:

$$G_t(\lambda) = (1-\lambda)\sum_{n=1}^{\infty}\lambda^{n-1}G_t^{(n)};$$

$$G_t^{(n)} = \sum_{l=0}^{n-1}\gamma^l r_{t+l} + \gamma^n V(s_{t+n})$$

$G_t^{(n)}$ tronca il calcolo di G_t al punto $t+n$, da lì in poi usa la V dell'opportuno stato come stima. La GAE fa una media tra varie $G_t^{(n)}$ pesate esponenzialmente secondo il parametro λ, la cui scelta quindi determina quanto ci appoggiamo alla stima corrente (alle funzioni di valore di stato V) piuttosto che ai reward immediati che ci dà l'ambiente: una grande λ indica che consideriamo anche $G_t^{(n)}$ con n grandi, mentre λ tendente a 0 indica che consideriamo soltanto $G_t^{(n)}$ per n piccoli, che collassano pertanto in $V(s_t)$ o quasi.

2.4 PPO

Proximal Policy Optimization è un metodo di ascesa stocastica del gradiente che utilizza più epoche (per aggiornare i parametri della policy). Questo metodo segue i metodi chiamati *Trust Region Policy Optimization*, ereditandone la stabilità e l'affidabilità, ma non la complessità implementativa, e ne migliora le performance. Difatti per passare da REINFORCE a PPO basta cambiare poche linee di codice, mentre l'implementazione di TRPO è ben più complessa.

Il primo passo è sostituire il termine logaritmico del policy gradient theorem con il rapporto tra probabilità di compiere l'azione con la policy attuale e probabilità di compiere un'azione con la policy predecente:

$$r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta old}(a_t|s_t)} \tag{16}$$

Passando quindi a ottimizzare la seguente funzione:

$$J(\theta) = \mathbb{E}_{\pi_\theta}[\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta old}(a_t|s_t)}\hat{A}_t] \tag{17}$$

L'utilizzo di tale $r_t(\theta)$ si presta ad un problema: se un'azione diventa molto più probabile con la nuova policy, abbiamo dei passi spropositati di ascesa del gradiente. Per evitare questo problema ci sono due possibili soluzioni: TRPO (*Trust-Region Policy Optimization*) fa numerosi controlli per limitare il passo di ascesa del gradiente, rendendo il tutto estremamente *complicato*; mentre PPO, introducendo il *Clipped Surrogate Objective*, ottiene lo stesso effetto facendo controllare l'ampiezza del passo alla funzione stessa, che diventa difatti:

$$J^{CLIP}(\theta) = \mathbb{E}_{\pi_\theta}[min(r_t(\theta)\hat{A}_t, clip(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t)] \tag{18}$$

Il motivo di tale funzione è chiaramente indicato e commentato nella figura 1. La quale mostra come si cerchi di non sbilanciare troppo gli update verso policy che sembrano localmente ottimali, viceversa sono ben accetti forti update atti a riparare "scelte sbagliate" prese in precedenza.

Gli update fatti per epoche sono presi direttamente dai classici algoritmi di discesa stocastica a mini-batch. Sostanzialmente si fa interagire l'agente (gli agenti) per raccogliere un numero T di esperienze. Questo processo viene ripetuto per tutti gli N attori, ottenendo NT differenti advantage function e surrogate objectives. Questi valori vengono dunque suddivisi in mini-batch, per eseguirvi gli algoritmi di discesa stocastica per K epoche (il gradiente viene sostanzialmente ottenuto sommando i singoli gradienti delle singole clipped objective function calcolate per ogni valore degli $\hat{A}_t$ del mini-batch).

Quello che succede è che per ogni iterazione, dopo aver campionato l'ambiente con π_{old} (e quando iniziamo a eseguire l'ottimizzazione), la nostra policy π sarà esattamente uguale a π_{old}. Quindi, all'inizio, nessuno dei nostri aggiornamenti verrà "clippato" e sicuramente avremo

degli aggiornamenti consistenti. Tuttavia, man mano che aggiorniamo π usando più epoche, l'obiettivo inizierà a colpire i limiti di clipping, il gradiente andrà a 0 per quei campioni e l'allenamento si fermerà gradualmente, finché non passeremo alla successiva iterazione e a raccogliere nuovi campioni.

Algorithm 3 PPO

Inizializza arbitrariamente il vettore θ dei parametri.
for iterazione = 1, 2... **do**
  for attore = 1, 2...N **do**
    Agisci seguendo la policy π_{θ_i} per T passi
    Calcola le $\hat{A}_1, ..., \hat{A}_T$
  Ottimizza J^{CLIP} rispetto a θ, mediante discesa stocastica del gradiente a mini-batch (e a multi-epoche)
  $\theta_{old} \leftarrow \theta$

2.5 Policy Network

Questa sezione copre brevemente la parte *deep* del deep reinforcement learning (purtroppo non sono riuscito a trovare fonti che approfondissero a sufficienza la questione). Sostanzialmente difatti per adesso abbiamo parlato di reinforcement learning, introducendo anche alcune policy objective function tuttavia sempre rimanendo nello "shallow" learning. Adesso ci occupiamo di mostrare come si può semplicemente inserire una parte *deep* nei policy gradient methods.

Sostanzialmente difatti si parla di Deep Reinforcement Learning quando semplicemente si utilizza una Deep Neural Network (di qualsiasi tipo, sia essa Fast-Forward, Convoluzionale, Ricorsiva...) per approssimare una qualsiasi funzione usata per il reinforcement learning. Nel nostro caso per approssimare la policy. La rete difatti prende in input osservazioni e reward e fa uscire in output azioni: solitamente si distinguono diverse metodologie di input: input discreti, che indicano opportuni valori dell'ambiente, o continui, che possono anche essere immagini. Tale distinzione si ha anche nell'output: che può essere una codifica di azioni, quando queste sono finite, oppure una serie di valori continui, come ad esempio accade quando si ha un output un vettore di forze da applicare per il movimento.

L'algoritmo è anche in questo caso il classico BackPropagation, guidato dall'aggiornamento della policy indicato da PPO.

3 Unity ML-Agents

Tale sezione illustrerà la libreria ML-Agents di Unity. Sostanzialmente tale libreria è composta da due parti interconnesse: l'ambiente di apprendimento, il quale consiste in una scena Unity (o un eseguibile) con gli opportuni agenti e le opportune entità che approfondiremo a breve, e la python API, nella quale si lavora con gli algoritmi di reinforcement learning, settando opportunamente i parametri per l'apprendimento. Le due parti comunicano tramite Socket.

In una scena adibita all'utilizzo degli ML-Agents si distinguono 3 principali Entità:

- *Agent*.
- *Brain*.
- *Academy*.

3.1 Agent

Il componente Agent viene utilizzato per indicare direttamente che un GameObject all'interno di una scena è un agente. Questo componente è responsabile della raccolta di osservazioni (e di ricompense) dall'ambiente e dell'esecuzione di azioni all'interno di quell'ambiente. Ciascun Agente può essere collegato soltanto ad un Brain, con il quale comunica dando le osservazioni raccolte e ricevendo le azioni da compiere. Chiaramente all'interno di una scena ci possono essere un qualsivoglia numero di Agenti.

A basso livello, un Agente è caratterizzato da due funzioni che è necessario implementare:

- CollectObservations(). In cui si collezionano le osservazioni dell'Agente. (Chiamando la funzione AddVectorObs() con ogni dato che si vuole usare come osservazione)
- AgentAction(). Che esegue l'azione indicata dal Brain e assegna un reward allo stato corrente.

3.2 Brain

I Brain si occupano di fare Decision Making, ricevendo in input le osservazioni degli Agenti che sono con loro collegati. Dal punto di vista dell'Agente, è il Brain che fornisce la politica da seguire. È possibile collegare più agenti a un singolo Brain, e questi prenderà le decisioni (indipendenti) per tutti loro. Ciascun Brain contiene una precisa definizione degli spazi delle osservazioni e delle azioni all'interno dei quali vale la propria politica, e pertanto soltanto Agent con corrispondenti configurazioni possono essere connessi al Brain in questione. É interessante sottolineare che gli Agenti possono richiedere l'intervento dei Brain sia a intervalli fissi sia ad intervalli dinamici. I Brain possono essere di diversi tipi (ovvero il processo decisionale può essere condotto in vari modi):

- Player. Attraverso l'input del giocatore
- Heuristic. Attraverso scripting
- Internal. Attraverso reti neurali incorporate (precedentemente ricavate tramite addestramento)
- External. Attraverso l'interazione con l'API Python (utilizzato per addestrare nuove reti neurali)

È chiaramente possibile includere più Brain nella stessa scena, nel caso si desiderino diverse politiche e diversi comportamenti.

3.3 Academy

La componente Academy viene utilizzata per tenere traccia dei passaggi della simulazione e per fornire funzionalità come la possibilità di ripristinare l'ambiente, impostare la velocità di simulazione e la frequenza dei fotogrammi. L'Academy contiene anche la possibilità di definire i parametri di reset, che possono essere utilizzati per cambiare la configurazione dell'ambiente a runtime. Come si deduce dalla sua funzionalità, ci può essere soltanto un Academy per scena.

A livello implementativo, le funzioni principali che caratterizzano l'Acadamy sono le seguenti (tutte implementabili facoltativamente):

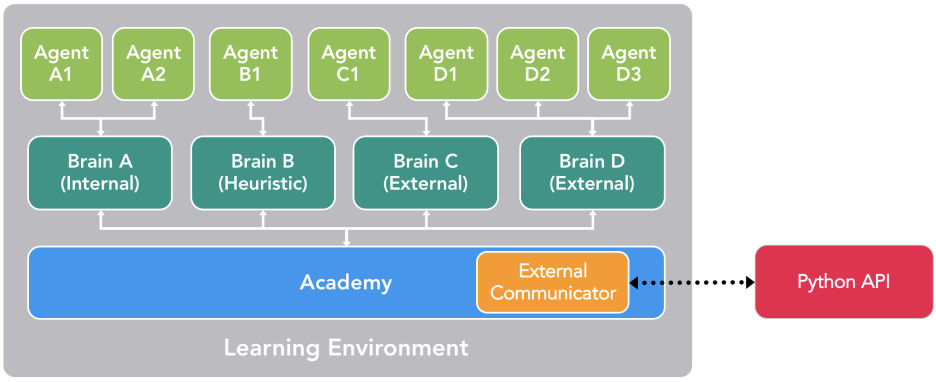

Fig. 2: Struttura della libreria ML-Agents

- InitializeAcademy(). Per preparare l'ambiente per la prima volta.
- AcademyReset(). Per preparare l'ambiente e l'agente per ogni episodio successivo al primo.
- AcademyStep(). Per preparare l'ambiente per il prossimo step della simulazione.

3.4 Il loop di training ed esecuzione

Ad ogni episodio, viene invocato AcademyReset() e AgentReset() per ogni agente nella scena. Academy orchestra l'esecuzione all'interno di un episodio in questo caso:

1) CollectObservations() viene invocata per ogni agente nella scena
2) Vengono invocati i Brain di ogni agente, passandogli le osservazioni
3) AcademyStep()
4) AgentAction() per ogni agente nella scena. Di seguito passa l'azione scelta al Brain corrispondente.

Quando il numero di passi nell'episodio raggiunge il valore Max Step indicato nell'Academy, si inizia un nuovo episodio, ripartendo da AcademyReset().

Chiaramente può anche succedere che sia l'agente stesso a terminare il proprio compito (oppure fallisca irrimediabilmente), in tal caso è l'agente a invocare la funzione Done() dentro alla AgentAction(). Facoltativamente anche l'agente ha un proprio valore di Max Step oltre il quale si considera terminato. Un'altra proprietà importante dell'agente è ResetOnDone(), che indica che se un agente termina, questi viene resettato (senza attendere di resettare l'intero episodio).

3.5 Progettare un ambiente di apprendimento

Dopo aver definito tutti i componenti, si arriva quindi alla definizione dell'ambiente di apprendimento. E lo si fa appunto andando a definire le osservazioni, le azioni e la funzione di reward; tutte e tre le componenti del reinforcement learning possono essere definite nell'Agente:

- Le Osservazioni possono essere sia vettori di numeri reali, sia una serie di immagini output di varie camere nella scena. In generale, è possibile definire osservazioni che corrispondono a qualsiasi informazione disponibile per l'agente, ad esempio anche risultati di ray-cast, segnali sonori, posizioni di agenti e quant'altro.
- Le Azioni possono essere discrete o continue (solitamente vettori di interi o vettori di numeri reali).
- Per quanto riguarda la funzione di Reward c'è poco da dire, questa può essere semplicemente definita e modificata in qualsiasi momento tramite scripting.

Ogni simulazione può essere considerata "Terminata" sia dall'agente stesso sia dall'ambiente. Solitamente comunque si tratta o di una qualche chiamata, che può avvenire sia internamente a Unity sia esternamente tramite la Python API, o del raggiungimento di un massimo numero di passi precedentemente definito.

3.6 Python API

Come detto precedentemente, la libreria ml-agents include un pacchetto Python, utilizzato come sovrastruttura alle implementazioni in tensorflow degli algoritmi. Il pacchetto Python fornito contiene una classe chiamata UnityEnvironment che pu essere utilizzata per avviare e interfacciare gli eseguibili Unity (nonché l'Editor) che contengono i componenti richiesti descritti sopra. La comunicazione tra Python e Unity avviene tramite un protocollo di comunicazione gRPC e utilizza i messaggi protobuf. L'interazione con l'API Python funziona come segue:

- env = UnityEnvironment (filename) - Avvia un ambiente di apprendimento dato un nome di file o percorso e stabilisce la comunicazione con l'API python.
- env.reset (): reimposta l'ambiente e restituisce un dizionario di oggetti BrainInfo contenente le osservazioni iniziali di tutti gli agenti.
- env.step (actions) - Passa l'ambiente di apprendimento fornendo un insieme di azioni per tutti gli agenti presenti nel dizionario BrainInfo precedentemente ricevuto e restituisce un dizionario di oggetti BrainInfo contenente nuove osservazioni dagli agenti che richiedono decisioni.
- env.close () - Invia il segnale di terminazione all'ambiente di apprendimento.

BrainInfo è una classe che contiene elenchi di osservazioni, azioni precedenti, ricompense e variabili di stato varie. In ogni fase di simulazione, BrainInfo contiene informazioni per tutti gli agenti attivi all'interno della scena

che richiedono decisioni affinché la simulazione possa procedere.

3.7 Invocare l'apprendimento da shell

È possibile far partire l'addestramento anche da shell. Per fare questo basta invocare Per invocare l'apprendimento basta invocare *mlagents-learn* con gli opportuni parametri:

- Il path in cui trovare l'ambiente/l'eseguibile. Se omesso l'apprendimento avverrà direttamente in Unity.
- Il path in cui trovare il file .yaml contenente tutti gli iperparametri.
- –train per dire che si vuole apprendere.
- Varie ed eventuali: –no-graphics perchè l'apprendimento avvenga in background, –load se si vuole proseguire con l'addestramento di un modello già addestrato.

4 Simulazioni

4.1 Iperparametri

Per addestrare tramite PPO è necessario avere a che fare con un enorme numero di parametri, visibili in figura 3:

1) β Che pesa il fattore di entropy-regularization. Ovvero si aggiunge alla funzione obiettivo anche l'entropia, pesata da β, che si cerca di massimizzare. Tale Entropia si usa per evitare di rendere troppo probabile un'azione rispetto alle altre, cosa che all'inizio potrebbe portare a fermarsi in ottimi locali.
2) γ, il discount factor che pesa l'influenza dei reward futuri distanti.
3) λ, che viene utilizzato durante il calcolo della Generalized Advantage Function.
4) ϵ, per il threshold nel Clipped Surrogate Objective di PPO
5) Time Horizon, che serve a decidere quanto avanti si guarda quando si calcola $G_t^{(n)}$.
6) Max Step, il numero massimo di step che vengono eseguiti durante l'apprendimento.
7) Summary Freq, che indica ogni quanto si desidera avere un resoconto durante l'appredimento.

Ed i parametri classici del Deep Learning:

1) Buffer Size, il numero di iterazioni (osservazione, azione, reward) dell'esperienza che si devono raccogliere per attivare l'algoritmo di apprendimento
2) Batch Size, il numero di iterazioni di esperienza usato da un passo di Ascesa del gradiente (che difatti avviene a mini-batch)
3) Numero di hidden layers e di unità, che determinano l'architettura della rete neurale (il policy network)
4) Learning rate, parametro che influenza la grandezza dei singoli step di Ascesa del gradiente.
5) Normalization, che può essere attivo o meno, dipendetemente se si vuole normalizzare o meno il vettore delle osservazioni in input.

Tutti gli altri parametri, come memory size, sequence length, sono correlati all'uso di reti ricorrenti, che non è stato fatto, lo stesso vale per curiosity e tutti i parametri ad essa relativi.

```yaml
default:
    trainer: ppo
    batch_size: 1024
    beta: 5.0e-3
    buffer_size: 10240
    epsilon: 0.2
    gamma: 0.99
    hidden_units: 128
    lambd: 0.95
    learning_rate: 3.0e-4
    max_steps: 5.0e4
    memory_size: 256
    normalize: false
    num_epoch: 3
    num_layers: 2
    time_horizon: 64
    sequence_length: 64
    summary_freq: 10000
    use_recurrent: false
    use_curiosity: false
    curiosity_strength: 0.0
    curiosity_enc_size: 128

RollerAgent:
    learning_rate: 5.0e-3
    max_steps: 400000
```

Fig. 3: File Yaml contenente gli iperparametri usati per il raccoglitore di cubi nel paragrafo a seguire. I parametri non specificati nel corrente documento sono utilizzati per tipologie di learning che non ho utilizzato per il progetto.

4.2 Raccoglitore di cubi

Il primo ambiente, utilizzato principalmente per sperimentare e per apprendere l'utilizzo della libreria è estremamente semplice: Un agente (una sfera) rotola su un piccolo piano mentre cerca di raccogliere cubi. In sostanza: nella scena vi è sempre un cubo, e appena questo viene raccolto, si genera un altro in posizione casuale. La strategia che si cerca di far imparare all'agente è di raccogliere più cubi possibili nel minor tempo possibile e senza cadere dalla piattaforma. Ogni volta che il cubo viene raccolto, un altro viene generato in una posizione casuale.

Reward Type	Value
Cubo raccolto	+1
Agente cade dalla piattaforma	-1
Ogni istante di tempo	-0.001

TABLE 1: Rewards Raccoglitore cubi

Le osservazioni che sono state date all'agente riguardano:

- La sua posizione rispetto ai bordi del piano
- La sua velocità
- Il vettore distanza tra agente e obiettivo

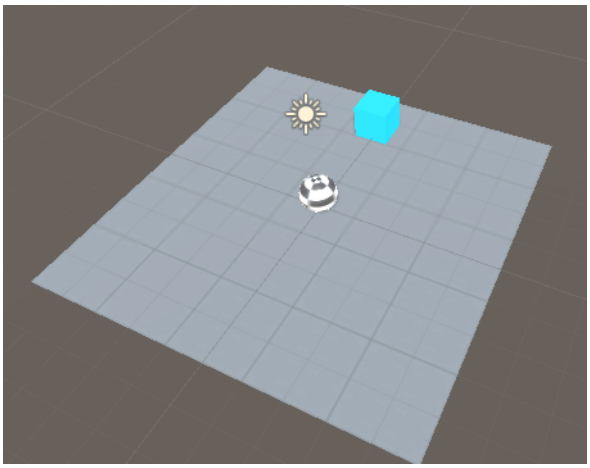

Fig. 4: Ambiente del raccoglitore di cubi.

5 Hockey da Tavolo

L'ambiente principale del progetto è invece questo. In tale ambiente si vuole ottenere un agente che riesca a giocare in maniera sufficientemente competitiva a hockey da tavolo.

5.1 Svolgimento e dettagli implementativi

Tale ambiente è caratterizzato da due agenti che competono l'uno contro l'altro, guidati comunque dallo stesso Brain.

Ogni campo è inoltre caratterizzato da due porte, uno script arbitro e un disco. Sostanzialmente, quando il disco viene in contatto con una porta, tale porta comunica all'arbitro che è avvenuto un goal, e l'arbitro si occupa di aggiornare i punteggi e di assegnare i reward di goalDone e goalTaken agli agenti.

Per poter addestrare due agenti con lo stesso Brain è stato necessario fare sì che i due agenti fornissero al Brain gli stessi numeri, questo è stato realizzato rendendo i due agenti figli di due gameObject vuoti nella scena, posizionati opportunamente nelle loro rispettive porte e rivolti l'uno contro l'altro. In tale modo tutti gli agenti si muovono credendo di avere lo stesso sistema di riferimento. Questo mi ha anche permesso di costruire diversi campi da gioco, così da raccogliere molte più esperienze simultaneamente.

Ad ogni goal gli agenti vengono resettati nelle loro posizioni di partenza, cos come il disco, al quale viene data una piccola velocità in direzione randomica per cercare di rendere più varia l'esperienza dei giocatori. Non c'è una causa di fine partita: gli agenti giocano finchè l'esecuzione non viene interrotta. Infine, per accelerare il processo di apprendimento, ho imposto che l'ambiente venga resettato dopo 3000 passi di decision making da parte degli agenti (il qualche viene invocato ogni 5 frame): questo perchè gli agenti inizialmente rimangono per lungo tempo fermi ai bordi, e l'episodio rischia di non terminare, cosa che appunto allungherebbe di molto gli step di addestramento necessari per ottenere una buona IA.

La valutazione del comportamento degli agenti, essendo abbastanza controintuitiva da fare analiticamente (in quanto, addestrandoli uno contro l'altro, i valori effettivi di reward rimangono sostanzialmente bilanciati, a meno di un dettaglio che spiego nel sottocapitolo successivo), è stata ben più facilmente fatta giocando contro gli agenti o facendo giocare gli agenti con diversi brain l'uno contro l'altro.

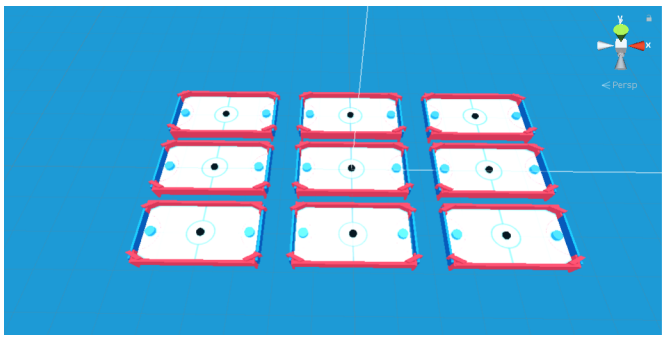

Fig. 5: È possibile instanziare un qualsivoglia numero di campi di hockey per addestrare i Brain in parallelo

Le osservazioni che sono state date a ogni agente riguardano semplicemente:

- La posizione e la velocità di quest'ultimo
- La posizione e la velocità del disco

5.2 Rewards

La progettazione dei reward è stato un processo iterativo. In quanto ho cercato in primis di vedere come il gioco si comportava soltanto con reward di gol, e poi ho cercato di aggiungere dei micro-reward per aggiustare alcuni viziosi comportamenti emergenti e favorirne degli altri. In particolare è importante sottolineare due reward sicuramente più interessanti: la punizione per la distanza dalla porta e il reward per collisione con il disco. Il primo è stato realizzato per creare un agente leggermente più difensivo, che non invadesse troppo il campo dell'altro giocatore (cosa che infastidirebbe un ipotetico giocatore umano utente dell'applicazione). Il secondo invece mi ha aiutato a discriminare meglio quando gli agenti si comportavano in maniera attiva: se c'è davvero del gioco, allora il disco viene toccato spesso; tale reward mi ha anche aiutato nelle prime fasi del training nel far capire agli agenti cosa fare.

Reward Type	Value
Gol Fatto	+10
Gol Subito	-10
Ogni istante di tempo	-0.001
Se troppo lontano dalla propria porta	-0.001
Collisione con il disco	+1*

TABLE 2: Rewards Air Hockey. *Tale reward viene analizzato nel penultimo capitolo

5.3 Scelta dei parametri

Una volta trovato dei reward che sembravano portare sulla strada giusta (anche se ho ritoccato anche questi in seguito), ho cominciato a perfezionare tutti gli iperparametri. Partendo dai valori utilizzati in Raccoglitore di Cubi, sono giunto a un set completamente diverso di iperparametri, principalmente perchè avevo bisogno di fare apprendimento molto più spesso, essendo l'ambiente certamente più complesso. Per lo stesso motivo ho allungato

```
HockeyBrain:
    max_steps: 5.0e5
    learning_rate: 1e-3
    batch_size: 128
    buffer_size: 2000
    beta: 1.0e-2
    hidden_units: 256
    time_horizon: 128
    summary_freq: 2000
```

Fig. 6: Gli iperParametri cambiati per hockey da tavolo

l'orizzonte temporale (per la relazione colpo e gol segnato conseguente). Grazie al reward di tocco palla, posso mostrare i seguenti grafici del reward accumulato nelle due sessioni di apprendimento, per palesare, anche se con il dovuto bias del fatto che un agente combatteva contro un agente con lo stesso brain, la differenza di performance tra i due modelli.

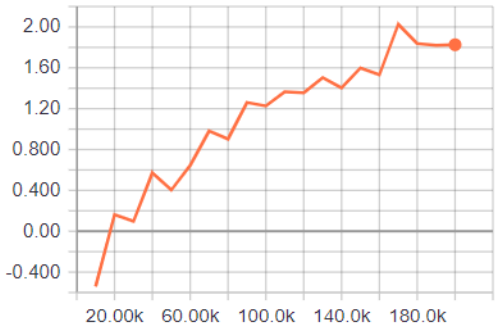

Fig. 7: Cumulative Reward con i vecchi parametri

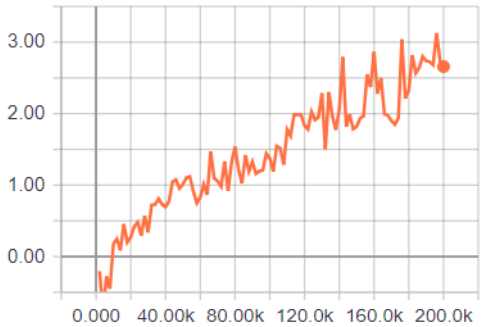

Fig. 8: Cumulative Reward con i nuovi parametri

Un punto in più dopo li stessi step è di per sé un guadagno considerevole: ma questo punto non rende giustizia alla schiacciante differenza tra i due modelli. Per tale motivo ho aggiunto nella documentazione un video in cui mostro una partita tra i due modelli differenti: partita a senso unico.

5.4 Altre statistiche

Per completezza di trattazione sui nuovi parametri, mostriamo e commentiamo alcune delle statistiche ottenute tramite apprendimento.

Come prima statistica, mi preme mostrare che aver imposto un limite di tempo ai singoli episodi non abbia introdotto grossi bias. Come si vede dal grafico infatti, la

Fig. 9: Fotogramma della partita tra nuovi e vecchi parametri

durata media degli episodi non rimane "clippata" dal limite da me posto, ma anzi si riduce nel tempo. Fatto di per sé indicativo che le partite vengono risolte nelle prime mosse quasi sempre.

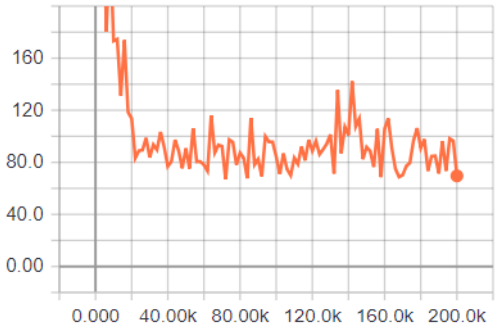

Fig. 10: Durata degli episodi

Un'altra statistica per me particolarmente interessante è l'Entropia della policy. Come possiamo vedere dal grafico 11, l'Entropia all'inizio aumenta, segno che l'agente non riesce a catturare propriamente un comportamento ottimo per ogni stato, e piuttosto punta a massimizzare l'entropia: questo è anche dato dal grande β che ho utilizzato. Poi tuttavia decrementa, mostrando che l'agente comincia a catturare un certo comportamento ottimale.

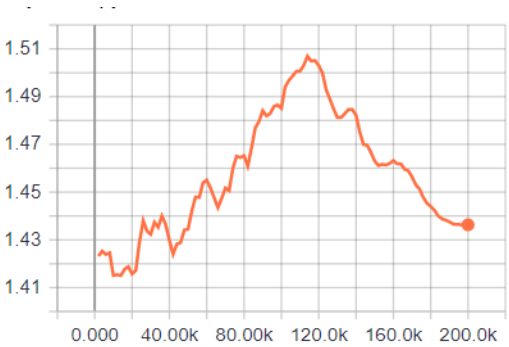

Fig. 11: Entropia della policy

Due grafici che ci mostrano l'andamento efficace dell'apprendimento riguadando il Learning Rate (figure 12 e 13), che decresce nel tempo, e la Value Estimate (il valore medio degli stati). Il primo decresce giustamente, il secondo invece aumenta.

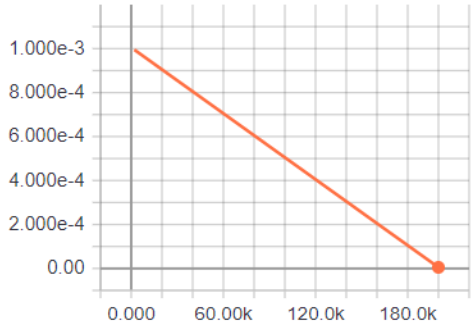

Fig. 12: Learning Rate

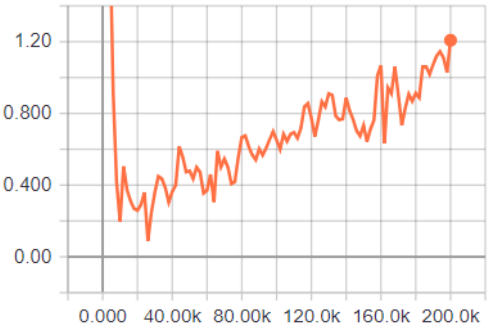

Fig. 13: Value Estimate

Il grafico in 14 invece riguarda la policy loss. Questa dovrebbe decrementare nel tempo, a indicare che ci si avvicina ad una policy ottimale e non ci sono più cambiamenti da fare alla policy attuale. Il grafico ci mostra che questo valore è fortemente instabile, e questo rappresenta come nell'ambiente si presentano continuamente nuove situazioni che vanno a modificare la policy.

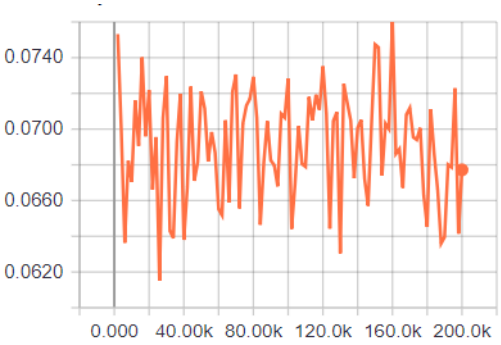

Fig. 14: Policy Loss

5.5 Il punto di equilibrio

L'aver utilizzato un reward per il tocco del disco ha introdotto un bias non indifferente nell'esperienza degli agenti, avendo difatti creato un "punto di equilibrio" del gioco, in quanto, se gli agenti avessero cooperato per passarsi la palla molto rapidamente, questi avrebbero ottenuto dei grandissimi rewards. Curioso di tale possibilità ho provato ad allungare di molto l'apprendimento (1 milione di step), e, come si vede da uno dei video nella documentazione, ho ottenuto tale atteggiamento cooperativo.

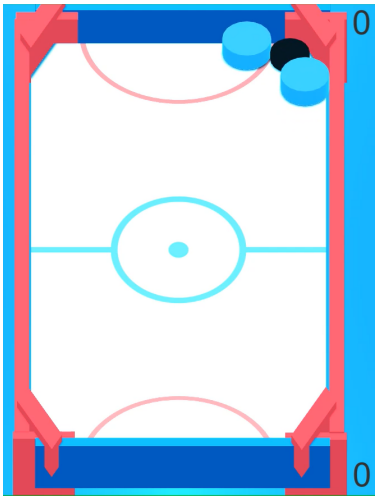

Fig. 15: I due agenti cooperano: si incontrano all'angolo e cominciano a passarsi la palla

Tuttavia, senza reward per collisione con il disco, gli agenti non hanno uno stimolo a giocare inizialmente, e i tempi di apprendimento si allungano notevolmente (con 200k passi, che produce un ottimo giocatore nell'altro caso, qui non si ottiene un comportamento intelligente).

La soluzione che ho alla fine pensato è piuttosto semplice: suddividere il training in due fasi. Nella prima fase si apprende a collidere con il disco, pertanto la collisione con il disco ha un reward pari a 1; nella seconda fase invece si dà per consolidato il fatto che gli agenti abbiano una certa "intuizione" che colpendo la palla in un certo modo possono guadagnare reward ben più grandi, e quindi li si insegna a giocare in generale e a segnare punti, con un reward per collusione con il disco pari a 0 (ho provato anche con 0.1, ma con risultati inferiori).

La foto dimostra l'efficacia dell'addestramento compiuto in tale modo.

Nel cercare il giusto bilanciamento tra le due fasi, su un totale di un milione di step di apprendimento, ho fatto diversi tentativi; provando diversi approcci nella suddivisione di tali fasi. In generale l'apprendimento di ogni Brain è stato interrotto a 200k e a 500k, così da darmi la possibilità di verificare se quel tipo di parametri e di reward portasse a un comportamento interessante. Alla fine il miglior agente è stato ottenuto dividendo equamente l'apprendimento nelle due fasi sopra citate: 500k con reward per collisione con disco e 500k senza tale reward. In foto 17 la "finale" contro l'agente della immagine precedente 16 che invece ha le fasi suddivise in 200k e 800k.

Chiaramente aver cambiato ambiente ad un certo punto dell'addestramento ha per un momento destabilizzato i nostri agenti, che da un certo momento si sono visti privare di una ricompensa che avevano ben imparato a ottenere.

Fig. 16: Il modello superiore è stato ottenuto da quello inferiore semplicemente continuando il training in un nuovo ambiente senza il reward per il tocco

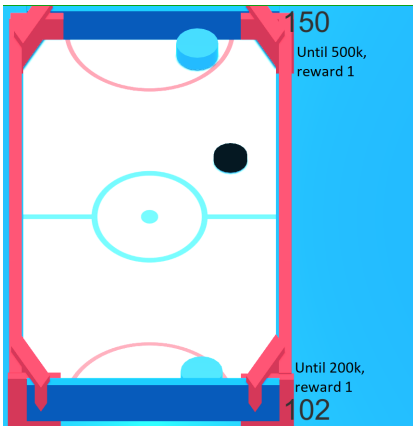

Fig. 17: Agenti addestrati per 1 Milione di passi. L'equilibrio tra le due fasi ci offre un agente più performante

Questo fatto è illustrato bene dal grafico del cumulative reward:

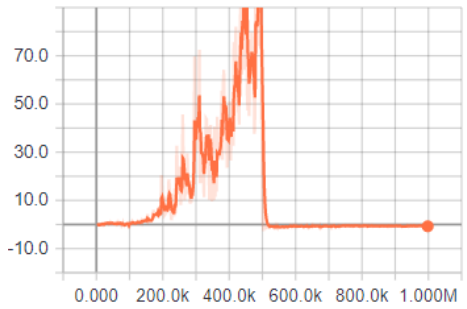

Fig. 18: Cumulative Reward viene azzerato quando si toglie il reward per la collisione del disco

Inoltre, aver fermato l'apprendimento a 200k e a 500k ha fatto riaumentare il learning rate a ogni ripartenza, come si vede in figura 19.

Similmente per l'Entropia (in figura 20), che si comporta nelle singole sezioni dell'apprendimento come faceva nella figura 11.

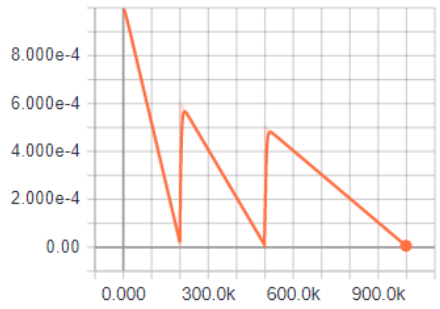

Fig. 19: A ogni ripartenza dell'apprendimento il learning rate riassume un valore alto

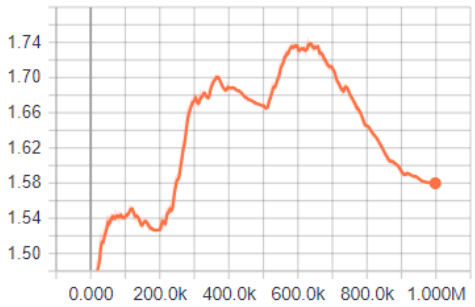

Fig. 20: Entropia nell'addestramento del miglior agente

5.6 Commento e Sviluppi futuri

L'aver adottato dei parametri che permettessero aggiornamenti più frequenti e l'aver introdotto le due fasi di apprendimento relativamente al reward per collisione col disco mi ha permesso di produrre un agente sufficientemente competente nel gioco dell'hockey da tavolo. In particolare è stato interessante vedere come senza un aggiornamento frequente della policy l'agente non riusciva ad apprendere alcun comportamento interessante; allo stesso modo il suddividere l'apprendimento in due fasi mi ha permesso di creare un agente che non è troppo aggressivo nelle prime fasi e poi immobile (come succedeva con la versione a 200k con reward per collisione con disco), ma piuttosto gioca più safe e si blocca molto meno spesso.

Infine, non avendo il problema del punto di equilibrio, posso far migliorare gli agenti a piacere, continuando ad addestrarli quanto voglio. Come dimostro in figura 21, in cui una versione con 500k steps in più sconfigge facilmente la versione con 1 milione di steps.

Tale versione è quella che si può sfidare nell'exe.

Per quanto riguarda possibili sviluppi futuri:

- Sicuramente è possibile continuare a ritoccare i parametri per un tempo indefinibile.
- Per evitare di "rovinare"(o influenzare troppo) l'esperienza di apprendimento non sono state usate diverse informazioni facilmente accessibili: non sono state usate informazioni riguardante l'avversario nelle osservazioni, né sono state utilizzate euristiche "complesse" per valutare l'esperienza dei giocatori, come ad esempio punirli se sono più distanti dalla propria porta del disco. Tuttavia sarebbe interessante

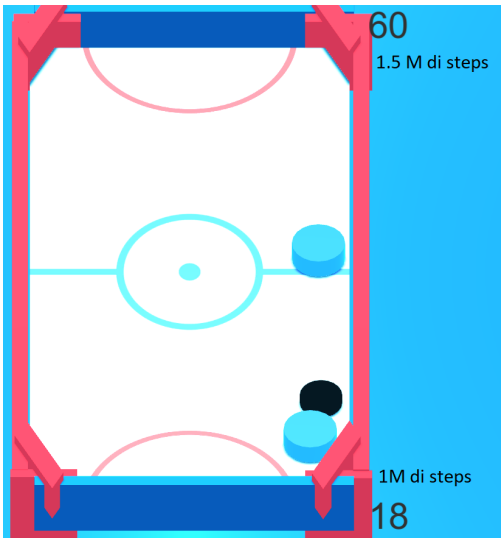

Fig. 21: La versione sopra è ottenuta semplicemente allungando l'apprendimento a partire da quella sotto.

vedere a quali comportamenti possano portare tali approcci.

- Un altro punto che sarebbe interessante approfondire è vedere come una serie di partite contro operatori umani possa influenzare l'apprendimento del nostro agente.
- L'ultimo punto di spunto che mi sento di aggiungere è un ipotetico scenario in cui si gioca 2v2: in tal caso sarebbe interessante vedere se differenziare i brain degli agenti (per avere un difensore e un attaccante) offra delle performance migliori a una squadra di agenti uguali.

References

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

[2] A. Juliani, V.-P. Berges, E. Vckay, Y. Gao, H. Henry, M. Mattar, and D. Lange, "Unity: A general platform for intelligent agents," *arXiv preprint arXiv:1809.02627*, 2018.

[3] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," *arXiv preprint arXiv:1707.06347*, 2017.

[4] D. Silver, "Reinforcement learning course," http://www0.cs.ucl.ac.uk/staff/d.silver/web/Teaching.html, 2015.