UNIVERSITÀ DEGLI STUDI
DI NAPOLI FEDERICO II

DIPARTIMENTO DI INGEGNERIA ELETTRICA E TECNOLOGIE DELL'INFORMAZIONE

CORSO DI LAUREA MAGISTRALE IN INFORMATICA

Documentazione del

Progetto d'esame

DEL CORSO DI INTELLIGENT ROBOTICS

Creazione di un ambiente 3D in Unity per l'addestramento di agenti utilizzando Imitation Learning e Curriculum Learning

Docenti
Prof. Finzi Alberto
Dott. Caccavale Riccardo

Studenti
Rauso Giuseppe N97000357
Longobardi Francesco N97000344

Anno Accademico

2021/22

1. Introduzione

1.1 Obiettivo del progetto

Lo scopo principale del progetto è quello di investigare le metodologie esistenti nell'ambito dell'apprendimento per rinforzo. Nello specifico il goal è quello di addestrare un agente utilizzando i paradigmi di *Imitation Learning e Curriculum Learning*. Per questo task è stato scelto di utilizzare il motore messo a disposizione da Unity con le relative librerie di supporto al Machine Learning.

1.2 Unity engine

Unity è un game engine *cross-platform* sviluppato da **Unity Technologies**. Grazie alla sua versatilità e semplicità d'uso, viene utilizzato in diversi settori tra cui: sviluppo di videogiochi 2D e 3D o in realtà virtuale, simulazioni, robotica, architettura. L'engine mette a disposizione sia un sistema di scripting in C# che di visual scripting. [1] L'ambiente da noi sviluppato per l'addestramento degli agenti è tridimensionale e costruito con **GameObject** primitivi messi a disposizione da Unity, quali cubi, sfere, piani, etc.

1.2.1 MLAgents

MLAgents è la libreria messa a disposizione da Unity per lo sviluppo e l'addestramento di agenti all'interno di ambienti sviluppati con l'editor.

Tra i suoi punti principali vediamo che un agente, nel proprio ambiente, deve specificare [2]:

- Osservazioni: possono essere di varia natura, fra cui vettoriali, basate su raycast 2D o 3D oppure visive
- Azioni: è possibile specificare azioni di tipo discreto o continuo, eventualmente anche in maniera ibrida
- Decision Requester: un modulo che scandisca la tempistica per la richiesta di decisioni e/o azioni da eseguire.
- Reward: sotto forma di funzioni per reward cumulativi oppure reward fissi

Per quanto riguarda il training, è stato utilizzato il modulo definito *mlagents-learn*. Esso è uno strumento utilizzabile da riga di comando per avviare un processo di training, fra i parametri figurano:

- Config: un file .yaml di configurazione, esso contiene l'algoritmo da utilizzare oltre che la lista di iperparametri da utilizzare per l'addestramento.
- Env: un campo a cui assegnare il file .exe in cui è stato impacchettato l'ambiente di esecuzione dell'agente
- Run-id: specifica un id univoco con il quale individuare un training run
- Initialize-from: specifica un training id da cui inizializzare i pesi

2. Ambiente

2.1 Scena

L'ambiente sviluppato vuole simulare un'invasione di un'isola posta al centro della mappa. L'agente ha a disposizione un cannone per fermare l'avanzata delle navi; questo può ruotare di 360 gradi su una base e di 45 gradi in alto e in basso. Nella versione definitiva della simulazione le navi si muovono con velocità distribuite uniformemente su un intervallo da 0 a 5. Tuttavia, nella prima fase dell'addestramento le navi sono ferme, come spiegato nel **capitolo 3**. L'agente può sparare un solo proiettile: un GameObject con componente **Rigidbody** per abilitare la gravità. In questo modo, l'angolazione del cannone dovrà essere regolata in base alla velocità del proiettile e all'accelerazione di gravità.

Ogni episodio inizia con il posizionamento della nave nemica in maniera casuale in un'area circolare con al centro l'agente e di raggio pari alla massima gittata del cannone, e termina con un colpo a segno o mancato, oppure con il raggiungimento dell'isola da parte della nave nemica. La funzione di reward pensata per questi casi è specificata nel paragrafo **2.4**.

2.2 Osservazioni dell'agente

L'agente, rappresentato da un'isola con un cannone, riceve in ingresso le seguenti osservazioni:

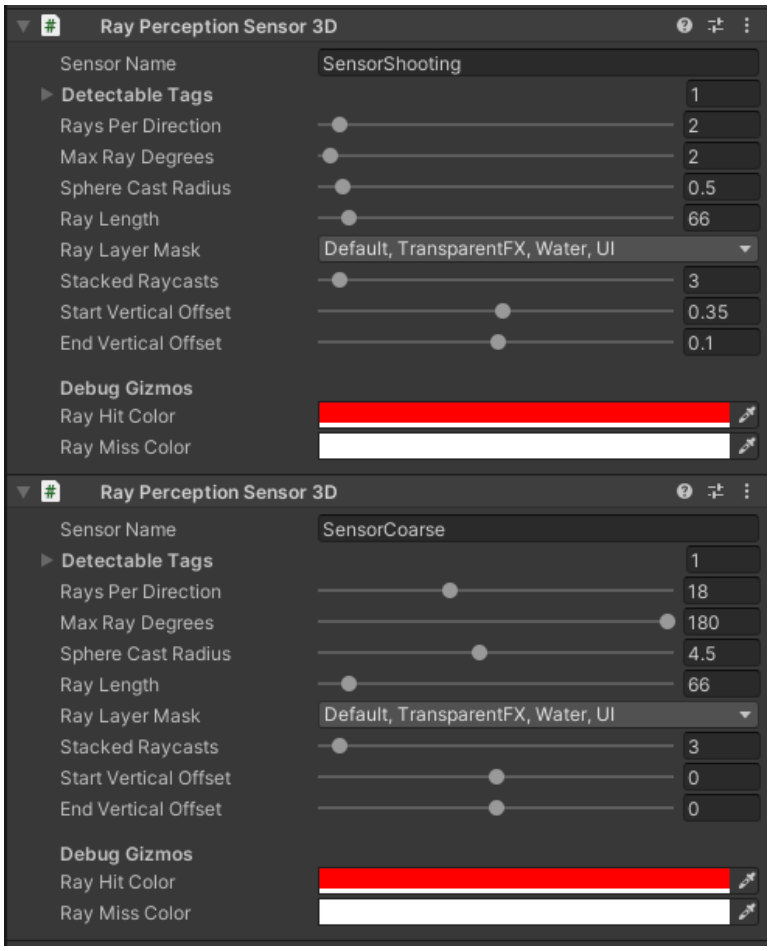

Figura 1: Pannello dell'inspector per i raycast

- Informazioni provenienti da un RayPerceptionSensor3D, un tipo di sensore pensato per simulare un LIDAR. In particolare, sono presenti due fasci di raggi di densità e funzione diversa. Il primo fascio (SensorCoarse) copre l'agente a 360° ed è stato introdotto per osservare l'intero campo di gioco e rilevare la presenza e la distanza di navi nemiche. Il secondo fascio (SensorShooting), molto più fitto, è composto da cinque raggi in uscita dalla bocca del cannone, essi sono pensati per fornire informazioni precise sulla distanza e posizione del bersaglio. Entrambi i fasci sono *stacked:* ad un dato step t, le osservazioni inviate sono la tripla $(obs_{t-2}, obs_{t-1}, obs_t)$

- Informazioni vettoriali manualmente raccolte dal metodo *CollectObservations(VectorSensor sensor)* rappresentate da due float: rispettivamente l'angolo di rotazione della CannonBase e l'angolo di elevazione del Cannone

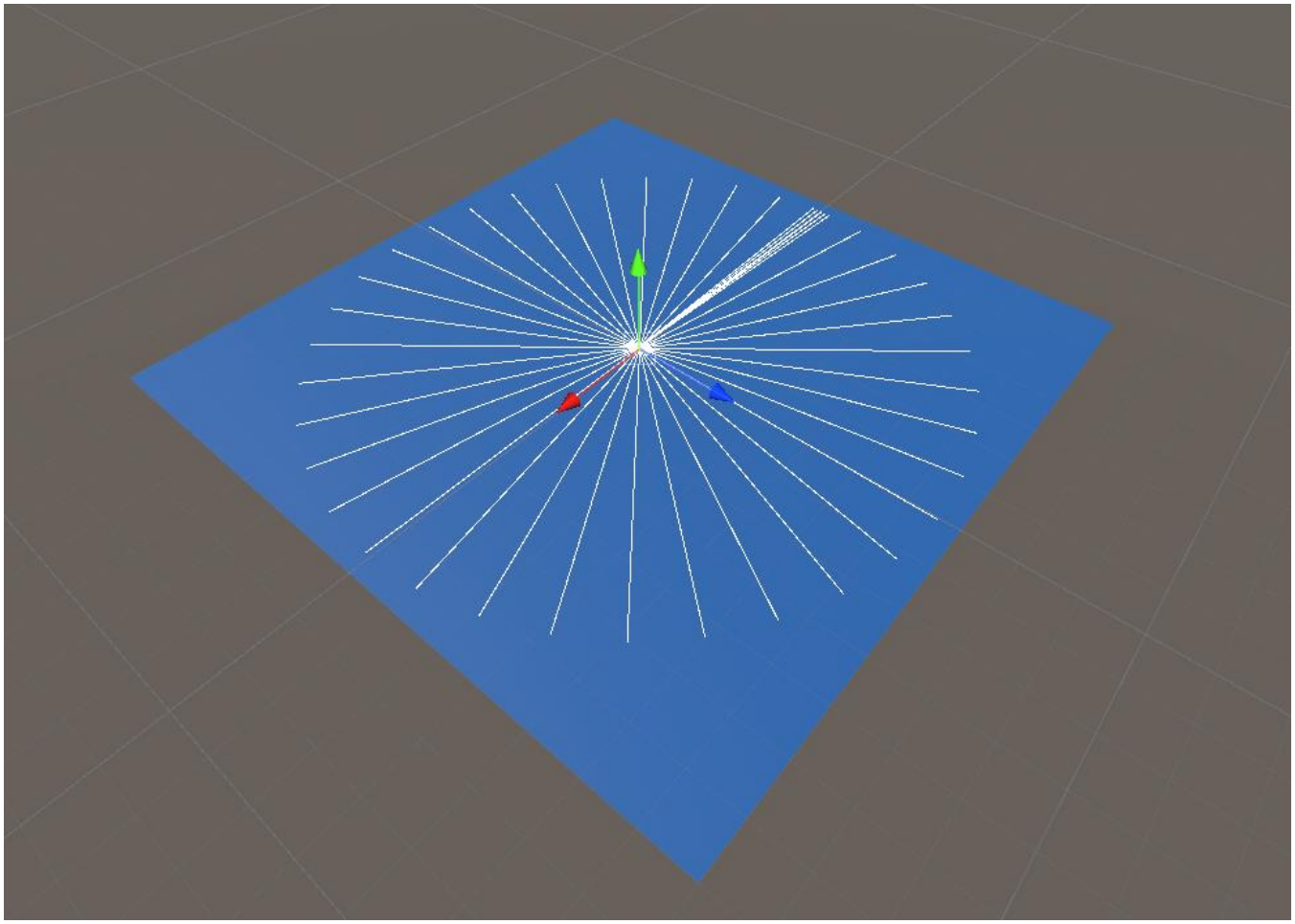

Figura 2: Visualizzazione delle osservazioni dell'agente

In totale il numero di osservazioni è dato da:

$$(ObservationStacks_{Sensor1}) \cdot (1 + 2 \cdot RaysPerDirection_{Sensor1}) \cdot (\#DetectableTags + 2) +$$
$$+ (ObservationStacks_{Sensor2}) \cdot (1 + 2 \cdot RaysPerDirection_{Sensor2}) \cdot (\#DetectableTags + 2) + 2$$

Dove i primi due termini della somma derivano dalle osservazioni generate dai raycast, mentre il +2 finale è dovuto alle due osservazioni vettoriali aggiunte.

La lista *DetectableTags* è in comune ad entrambi i sensori ed individua l'insieme di GameObject marcati con i tag specificati, da tenere in considerazione quando si osserva l'ambiente [3].

2.3 Azioni dell'agente

Le azioni scelte per questo task sono solo discrete e suddivise in tre *branch*. Nel dettaglio:

1. Movimento
    - Base del cannone
        - Ruota a destra (incremento dell'angolo sull'asse y)
        - Ruota a sinistra (decremento dell'angolo sull'asse y)
        - Non ruotare
    - Cannone
        - Ruota in alto (decremento dell'angolo sull'asse z)
        - Ruota in basso (incremento dell'angolo sull'asse z)
        - Non ruotare
    - Sparo
        - Spara
        - Non sparare

La velocità di rotazione è fissata, sia per la base che per il cannone. In questo modo l'agente non deve specificare di quanto ruotare, solo se ruotare oppure no e in quale direzione. Questo semplifica l'apprendimento ma può causare problemi di precisione se la velocità di rotazione non è impostata correttamente. Nel nostro caso, una velocità di $10\ {}^{gradi}/_{sec}$ è risultata adatta.

2.3.1 Decision Requester

Il package MLAgents di Unity mette a disposizione un **DecisionRequester** come componente, che permette di evitare di gestire via codice la gestione delle decisioni e tramite interfaccia è possibile selezionare il periodo di decisione in step Academy[1] e se l'agente può eseguire azioni mentre attende la decisione. Nel nostro caso, però, è stato necessario chiamare manualmente la funzione *RequestDecision()* del modulo sopracitato ad ogni step, avendo cura di interrompere queste chiamate nel momento in cui venga deciso di sparare. In questo modo è stato possibile evitare che vengano richieste decisioni durante il tempo di volo del proiettile, comportamento che avrebbe portato all'assegnazione dei reward ad azioni sbagliate, avvenute dopo l'ultima azione possibile, cioè lo sparo.

2.4 Funzione di Reward

La funzione di reward utilizzata per l'apprendimento di questo task è basata su quattro "eventi" principali:

- Lo scorrere del tempo: per ogni step l'agente riceve una penalità cumulativa pari a $-\frac{1}{MaxStep}$ dove MaxStep è stato fissato a 5000 e corrisponde al massimo numero di step disponibili all'agente in un singolo episodio.
- Bersaglio mancato: quando l'agente spara mancando il bersaglio, l'episodio termina ed esso riceve una penalità pari a

$$penalty(distance) = \begin{cases} -0.02 * distance, & -0.02 * distance > -1 \\ -1, & altrimenti \end{cases}$$

[1] La classe Academy (Class Academy | ML Agents | 1.0.8 (unity3d.com)) gestisce il training e il processo di decisione dell'agente. Gli step Academy sono conteggiati nel loop FixedUpdate (Unity - Scripting API: MonoBehaviour.FixedUpdate() (unity3d.com)).

Quindi, chiamando questa penalità semplicemente *penalty*, al termine dell'episodio il reward cumulativo sarà

$$Reward(penalty) = \begin{cases} penalty + CumulativeReward, & penalty + CumulativeReward > -1 \\ -1, & altrimenti \end{cases}$$

- Bersaglio colpito: è lo stato goal, quindi l'episodio termina ed esso riceve un reward pari a 3 che andrà a sommarsi con il *CumulativeReward* attuale
- Sbarco sull'isola: se l'agente viene raggiunto da una nave nemica, l'episodio termina ed esso riceve una penalità fissa (che sovrascrive il reward cumulativo) pari a -1

3. Metodo

Il task comprende far apprendere all'agente come colpire bersagli in movimento a velocità casuale in un intervallo che abbiamo definito essere $[0,5]$. Per raggiungere questo goal abbiamo diviso il training in due fasi: nella prima fase abbiamo fatto uso di tecniche di Imitation Learning per apprendere il task con bersagli fissi. In seguito, dopo aver ottenuto buoni risultati abbiamo utilizzato la tecnica del Curriculum Learning per affinare il comportamento dell'agente sul task generale.

3.1 Policy Gradient

In apprendimento per rinforzo, gli algoritmi basati su Policy Gradient sono quelli che mirano a modellare la policy e ad ottimizzarla direttamente. Tipicamente la modellazione adottata si basa su parametri come i pesi θ e la strategia che dipende da essi $\pi_\theta(a|s)$.

La funzione obiettivo (reward) è definita come:

$$J(\theta) = \sum_{s \in S} d^\pi(s)\, V^\pi(s) = \sum_{s \in S} d^\pi(s) \sum_{a \in A} \pi_\theta(a|s)\, Q^\pi(s,a)$$

Dove $d^\pi(s)$ è la distribuzione di probabilità stazionaria $d^\pi(s) = \lim_{t \to \infty} P(s_t = s | s_0, \pi_\theta)$

Si può notare come il valore di questa funzione dipenda dalla policy e di conseguenza da θ e diversi algoritmi possono essere impiegati per ottimizzarla. È possibile, infatti, utilizzare il metodo del gradient ascent per "spostare" i pesi lungo la direzione del gradiente $\nabla_\theta J(\theta)$.

Calcolare direttamente il seguente gradiente risulta essere complicato:

$$\nabla_\theta J(\theta) = \nabla_\theta \sum_{s \in S} d^\pi(s) \sum_{a \in A} Q^\pi(s,a) \pi_\theta(a|s)$$

Utilizzando il **Policy Gradient Theorem** [4] (Sec. 13.1) è possibile riscriverlo in una forma proporzionale:

$$\nabla_\theta J(\theta) = \nabla_\theta \sum_{s \in S} d^\pi(s) \sum_{a \in A} Q^\pi(s,a) \pi_\theta(a|s) \;\propto\; \sum_{s \in S} d^\pi(s) \sum_{a \in A} Q^\pi(s,a) \nabla_\theta \pi_\theta(a|s) \;=$$

$$= \sum_{s \in S} d^\pi(s) \sum_{a \in A} \pi_\theta(a|s) Q^\pi(s,a) \frac{\nabla_\theta \pi_\theta(a|s)}{\pi_\theta(a|s)} \;=\; E_\pi\left[Q^\pi(s,a) \nabla_\theta ln\left(\pi_\theta(a|s)\right)\right]$$

3.1.1 Actor Critic

I metodi basati su Actor-Critic sfruttano l'idea che apprendere la value function $V^\pi(s)$ insieme alla policy possa guidare gli aggiornamenti della policy stessa come, ad esempio, riducendo la varianza del gradiente.

In generale, si hanno due modelli:

- **Actor** aggiorna i parametri θ della policy $\pi_\theta(a|s)$ nella direzione suggerita da critic
- **Critic** aggiorna i parametri w che, a seconda del modello, si riferiscono a $Q^w(s,a)$ o a $V^w(s)$

3.2 Proximal Policy Optimization

PPO nasce sulla base dell'idea introdotta con **Trust Region Policy Optimization (TRPO)** [5], e cioè quella di evitare aggiornamenti dei parametri che cambino troppo la policy dopo un singolo step. Il problema, quindi, diventa quello di massimizzare la funzione obiettivo "surrogata" mantenendo un vincolo di distanza tra la vecchia policy e la nuova, calcolata con la **divergenza KL**[2]:

$$\max_{\theta} \quad \mathbb{E}\left[\frac{\pi_\theta(a|s)}{\pi_{\theta_{old}}(a|s)}\widehat{A}_{\theta_{old}}(s,a)\right]$$

$$\text{subject to} \quad \mathbb{E}\left[\mathrm{KL}(\pi_{\theta_{old}}(\cdot\,|s)\,||\,\pi_\theta(\cdot\,|s))\right] \le \delta$$

PPO implementa un vincolo simile ma semplificato, usando una funzione obiettivo surrogata troncata, mantenendo però le stesse performance.

L'idea è quella di forzare il rapporto $r(\theta) = \frac{\pi_\theta(a|s)}{\pi_{\theta_{old}}(a|s)}$ ad avere valori nell'intervallo $[1-\epsilon, 1+\epsilon]$, dove ϵ è un iperparametro. La funzione obiettivo quindi sarà

$$J^{CLIP}(\theta) = \mathbb{E}\left[\min\left(r(\theta)\widehat{A}_{\theta_{old}}(s,a), \mathrm{clip}(r(\theta), 1-\epsilon, 1+\epsilon)\widehat{A}_{\theta_{old}}(s,a)\right)\right]$$

La funzione $\mathrm{clip}(r(\theta), 1-\epsilon, 1+\epsilon)$ tiene il rapporto $r(\theta)$ tra $1-\epsilon$ e $1+\epsilon$. In questo modo la funzione obiettivo sceglie il minimo tra il valore originale e quello "clippato", facendo perdere quindi l'incentivo a cambiare drasticamente i pesi della policy per dei reward migliori. [6]

Un altro approccio, usato come alternativa al clipping o in aggiunta ad esso (come nel nostro caso), consiste nell'utilizzo di una penalità sulla divergenza KL utilizzando un coefficiente adattivo in modo tale da raggiungere un valore target d_{target} della KL ad ogni aggiornamento della policy.

Di seguito riportati i passi della versione base di questo algoritmo (solo penalità su divergenza KL):

Per ogni aggiornamento della policy:
– Si ottimizza la funzione obiettivo con penalità KL usando diverse epoche di minibatch SGD:

$$J^{KLPEN}(\theta) = \mathbb{E}\left[\frac{\pi_\theta(a|s)}{\pi_{\theta_{old}}(a|s)}\widehat{A}_{\theta_{old}}(s,a) - \beta\mathrm{KL}(\pi_{\theta_{old}}(\cdot\,|s)\,||\,\pi_\theta(\cdot\,|s))\right]$$

- Si calcola $d = \mathbb{E}\left[\mathrm{KL}(\pi_{\theta_{old}}(\cdot\,|s)\,||\,\pi_\theta(\cdot\,|s))\right]$
  – *Se* $d < d_{targ}/1.5$, $\quad \beta \leftarrow \beta/2$
  – *Se* $d > d_{targ} \times 1.5$, $\quad \beta \leftarrow \beta \times 2$

Il valore aggiornato di β è usato per il prossimo aggiornamento della policy, e il suo valore iniziale è un altro iperparametro che però ha poca importanza in pratica perché viene sistemato velocemente. La scelta di 1.5 e 2 è euristica; in realtà l'algoritmo non è molto sensibile ai cambiamenti di questi valori. [6]

[2] Divergenza di Kullback-Leibler: $\mathrm{KL}(Q||P) = \sum_i Q(i)\log_2\left(\frac{Q(i)}{P(i)}\right)$

In una ricerca successiva (**Hsu et al., 2020** [7]) si evidenziano tre problematiche legate agli algoritmi PPO:

1. Su spazi di azioni continui, l'algoritmo standard PPO non è stabile quando un segnale di reward porta la policy in una regione con reward molto bassi
2. Sugli spazi di azioni discreti con reward alti sparsi, l'algoritmo standard PPO si blocca spesso su azioni subottimali
3. L'inizializzazione della policy può incidere significativamente quando ci sono azioni ottime localmente in partenza

Nel paper si propone di discretizzare lo spazio degli stati o usare una distribuzione β[3] per evitare 1 e 3 nel caso di una policy Gaussiana ($\pi_\theta(a|s) = \mathcal{N}\left(\mu_\theta(s), \sigma_\theta^2(s)\right)$). Usando invece la regolarizzazione KL come alternativa al modello surrogato risolve i problemi 1 e 2.

3.3 Behavioral Cloning

Una delle tecniche più semplici di apprendimento per imitazione è il behavioral cloning: avendo delle dimostrazioni di un esperto seguendo una policy π^*, si vuole trovare una policy $\hat{\pi}$ che imita al meglio l'esperto minimizzando una loss surrogata $\ell(s, \pi)$.

Introduciamo la notazione usata in [8]. Denotiamo con Π la classe di policy che il learner considera e T l'orizzonte del task. Per ogni policy π, denotiamo con d_π^t la distribuzione di stati al tempo t se il learner ha seguito la policy π dallo step 1 allo step $t-1$ e con $d_\pi = \frac{1}{T}\sum_{t=1}^{T} d_\pi^t$ la distribuzione media degli stati seguendo la stessa policy per t step. Dato uno stato s, indichiamo con $C(s,a)$ il costo immediato atteso di fare un'azione a in uno stato s ed indichiamo con $C_\pi(s) = \mathbb{E}_{a\sim\pi(s)}[C(s,a)]$ il costo immediato atteso di π in s. Assumiamo C in $[0,1]$. Il costo totale dell'esecuzione di π per T passi è indicato con $J(\pi) = \sum_{t=1}^{T} \mathbb{E}_{s\sim d_\pi^t}[C_\pi(s)] = T\mathbb{E}_{s\sim d_\pi^t}[C_\pi(s)]$.

Si vuole quindi limitare $J(\pi)$ per ogni funzione di costo C basato su quanto bene π imita la policy dell'esperto π^*. Minimizziamo quindi una loss surrogata ℓ di π rispetto a π^* invece di C. È interessante notare che in molti casi ℓ e C possono essere la stessa funzione: ad esempio quando si vuole ottimizzare l'abilità del learner di predire l'azione scelta dall'esperto.

Formalizzando, il goal è

$$\hat{\pi} = \arg\min_{\pi\in\Pi} \mathbb{E}_{s\sim d_\pi}[\ell(s,\pi)]$$

Siccome le dinamiche del sistema sono per assunto sconosciute e complesse, non è possibile calcolare direttamente d_π e si può solo campionare seguendo π. Il problema principale è che utilizzando un approccio basato su supervised learning i dati si assumono i.i.d. . Questo però è impossibile nel caso di un processo markoviano che si assume alla base delle dinamiche dell'ambiente e quindi di una dipendenza dei dati in input dalla policy π stessa. Questo porta a dei problemi e ad una ottimizzazione difficile (funzione obiettivo non convessa) anche se la loss è convessa in π per ogni stato s. [8]

[3] $\pi_\theta(a|s) = f\left(\frac{a-l}{r-l}, \alpha_\theta(s), \beta_\theta(s)\right)$ con $f(x,\alpha,\beta) = \frac{\Gamma(\alpha+\beta)}{\Gamma(\alpha)\Gamma(\beta)} x^{\alpha-1}\left(1-x^{\beta-1}\right)$ e $a \in [l,r]$ le azioni dello spazio di azioni limitato da l e r. [11]

Nel nostro caso l'approccio tradizionale messo a disposizione da Unity, cioè un algoritmo di supervised learning, ha dato buoni risultati. Tuttavia, in letteratura sono state proposte metodologie che aiutano a mitigare il problema sopra descritto, come in [9] con l'algoritmo di **forward training** e **SMILe**, o in [8] con la tecnica di **Dataset Aggregation (DAGGER)**.

In particolare, nella libreria messa a disposizione da Unity, la loss nel caso discreto è calcolata semplicemente come somma cambiata di segno delle probabilità in logaritmo delle azioni eseguite dall'agente che ha eseguito anche l'esperto negli stessi stati visitati. In questo modo, ad esempio, la loss sarà più bassa se l'agente ha una probabilità alta di eseguire la stessa azione che ha eseguito l'esperto in un dato stato.

Nei nostri esperimenti abbiamo usato il BC in combinazione con l'algoritmo PPO per il reinforcement learning: abbiamo impostato un valore di strength per il BC pari a 0.5 e per il RL a 1. Questo valore viene moltiplicato al learning rate utilizzato poi per l'aggiornamento dei pesi. In particolare, la policy viene aggiornata prima da PPO, con strength 1 e quindi con il learning rate specificato, e poi da BC con strength 0.5, quindi con learning rate dimezzato.

3.4 Training

Al fine di raggiungere il goal specificato, la fase di training dell'agente è stata divisa in due passi indipendenti. Durante la prima fase l'agente ha appreso come colpire bersagli stazionari, a distanza variabile. Nella seconda fase è stato affinato il comportamento dell'agente addestrandolo su bersagli in movimento.

3.4.1 Prima fase di training

Durante questa fase, tutti i bersagli vengono generati ad una distanza compresa nell'insieme $[5, CannonMaxRange]$ e con velocità pari a 0. L'agente, in questa prima parte, è stato addestrato utilizzando **ppo** con **behavioral cloning** fornendo un file di dimostrazione *.demo* contenente 500 episodi svolti da un *aimbot* sviluppato ad-hoc per questo task.

3.4.1.1 Aimbot

L'aimbot sviluppato influenza i due angoli di mira dell'agente: l'angolo di rotazione orizzontale della base del cannone (che può ruotare a 360°) e l'angolo di elevazione verticale del cannone (che è ristretto ad un intervallo compreso fra $[117°, 45°]$: in questo contesto elevare il cannone verso l'alto equivale a diminuire l'angolo di partenza e viceversa). La computazione degli angoli necessari viene riservata ai rispettivi moduli che controllano il comportamento dei pezzi coinvolti.

- Nel caso della **base del cannone**, viene calcolato il vettore direzione $||enemyPosition - cannonBasePosition||$ con il quale viene calcolata la rotazione necessaria per il puntamento, rispetto all'asse di interesse, facendo uso del metodo *LookRotation* implementato nella classe **Quaternion.**

- Per quanto riguarda l'elevazione del **cannone**, definiamo:

$x = \sqrt{(target.x - cannon.x)^2 + (target.z - cannon.z)^2}$

$y = cannon.y - target.y$

L'angolo di fuoco sarà dato da:

$$\alpha = arctan\left(\frac{v^2 + \sqrt{(v^4 - g(x^2 g + 2yv^2)}}{g\mathrm{x}}\right)$$

In seguito, verrà convertito da radianti a gradi e processato per le caratteristiche di Unity:

$$\alpha = -(\frac{360\alpha}{2\pi} + 90)$$

Gli angoli così ricavati vengono sommati agli angoli di base del cannone e della base, così ottenendo una traiettoria che colpisca il bersaglio.

Alla fine di questa fase si sono ottenuti dei pesi (**brain**) che svolgessero il sottotask in maniera affidabile.

3.4.2 Seconda fase di training

Durante questa seconda fase, si è deciso di affinare la policy appresa durante la fase precedente sottoponendo l'agente ad un task di difficoltà incrementale.

Questa tecnica, conosciuta col nome di **curriculum learning**, si basa sul suddividere un goal in lezioni di difficoltà incrementale, metodo che permette sia di raggiungere più velocemente la convergenza che di ottenere dei minimi locali migliori [10]. Nel nostro caso ogni lezione è caratterizzata da un elemento di variabilità per qualche parametro dell'ambiente, come nel nostro caso la velocità dei bersagli, e da una metrica da "osservare" fra cui progress e **reward**. In particolare, consideriamo una lezione come completata quando accade che negli ultimi 100 episodi, il reward cumulativo medio superi un certo threshold che noi abbiamo fissato a 2.5.

La nostra suddivisione comprende 5 lezioni, utilizzando la seguente regola per incrementare la difficoltà:

Data la lezione $L_i \mid i \in [1,5]$

La velocità del bersaglio per un certo episodio di L_i è data da:

$$v \sim U(0, i)$$

È importante notare anche che la distanza minima di generazione dei bersagli è stata aumentata da 5 a 30, in modo da evitare che bersagli con velocità elevate venissero generati troppo vicino all'agente e vi si scontrassero prima che il puntamento avesse tempo sufficiente per avvenire.

Il training è stato svolto a partire dai pesi ottenuti al passo precedente, avendo cura di ridurre il learning rate e il parametro epsilon di ppo.

Alla fine di questa seconda fase, l'agente ha appreso il task di colpire bersagli in movimento in un certo intervallo di variabilità. Il prossimo capitolo sarà dedicato all'analisi dei risultati ottenuti.

4. Risultati Sperimentali

Di seguito viene presentata una rassegna di tutti i grafici riguardanti entrambe le fasi di training.

4.1 Risultati prima fase di training

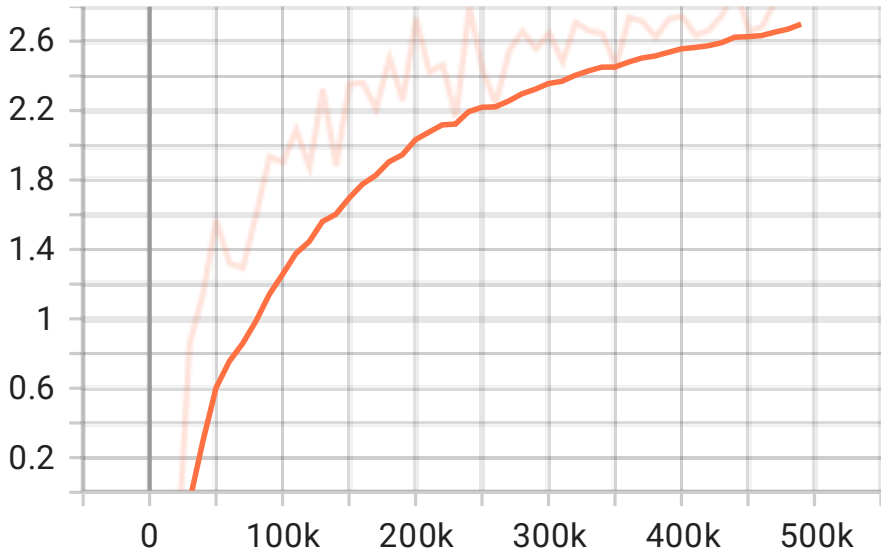

Figura 3: Cumulative reward

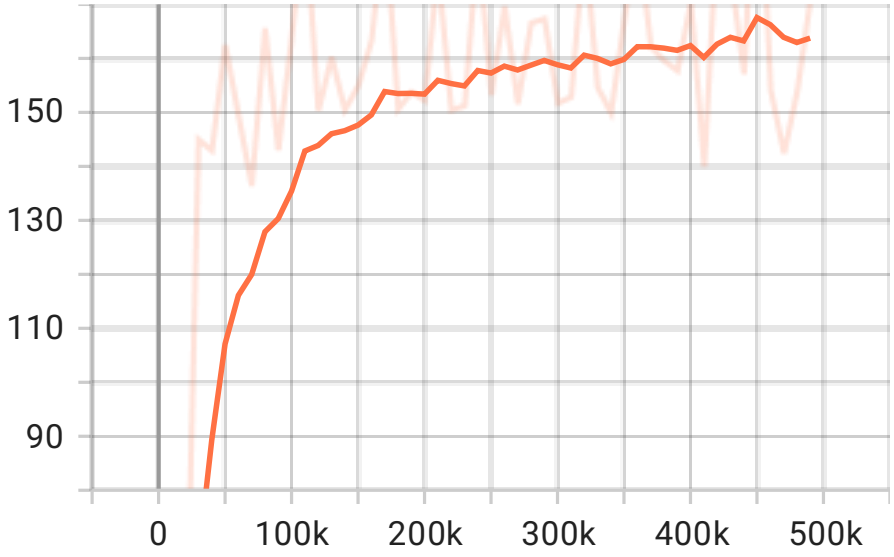

Figura 4: Episode length

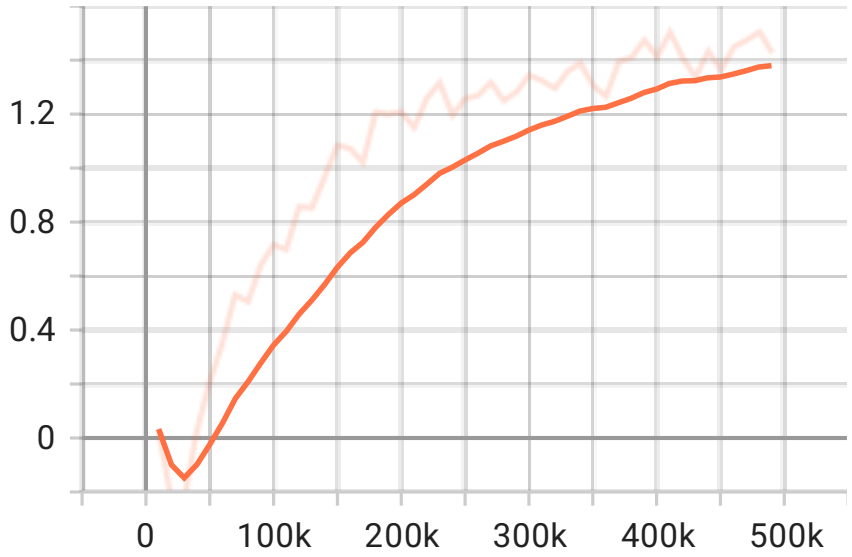

Figura 5: Extrinsic Value estimate

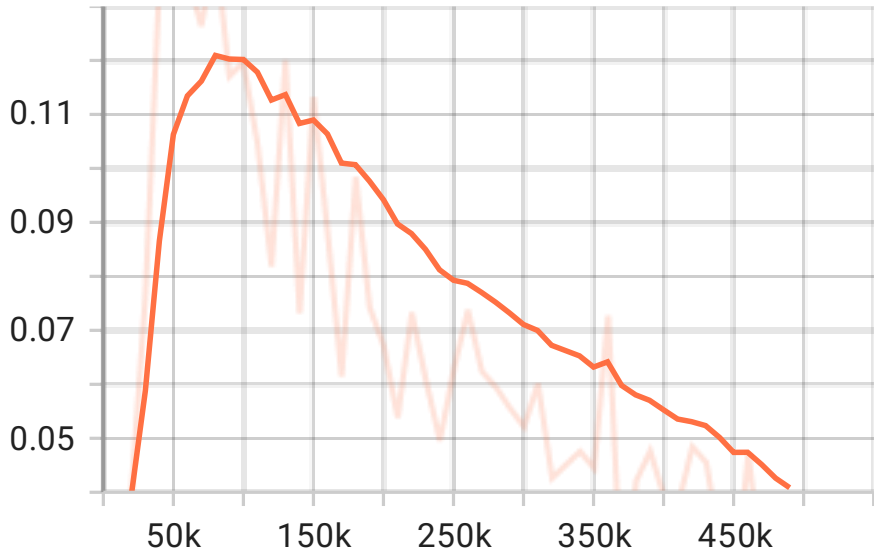

Figura 6: Value loss

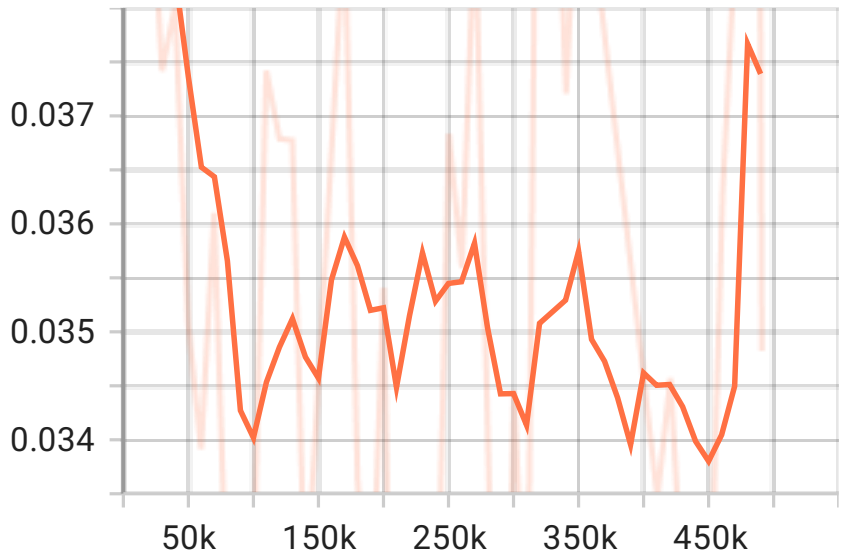

Figura 7: Policy loss

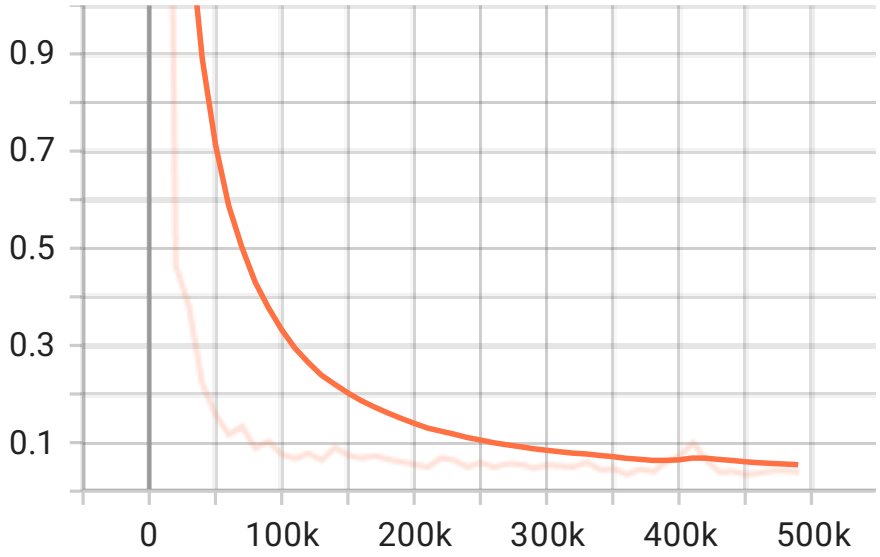

Figura 8: Pretraining loss: riguarda le performance dei pesi del behavioral cloning

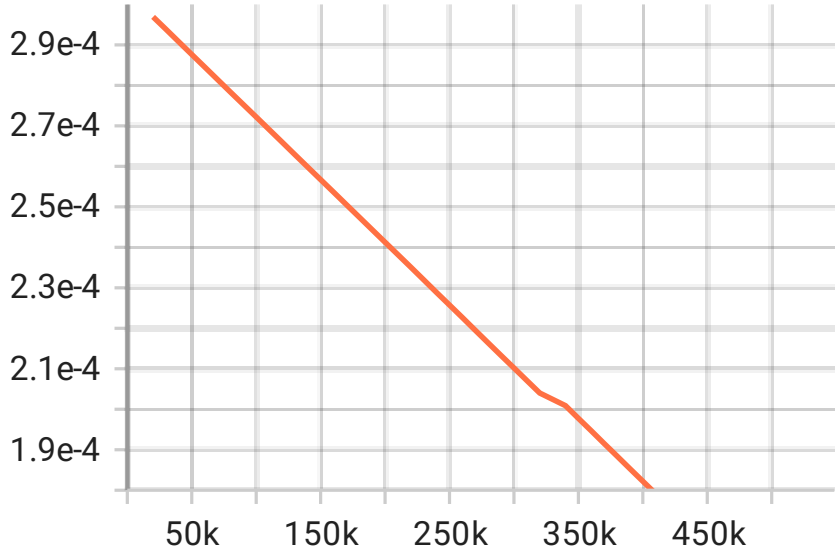

Figura 9: Learning rate

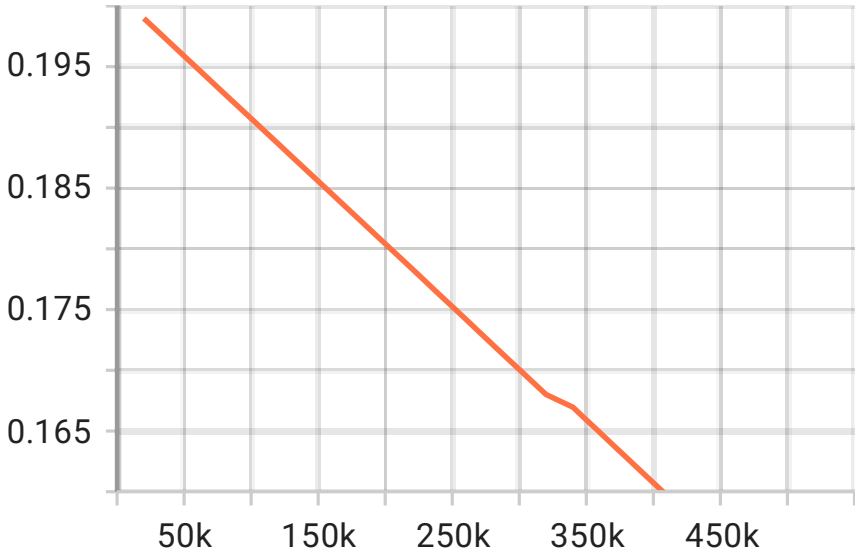

Figura 10: Epsilon

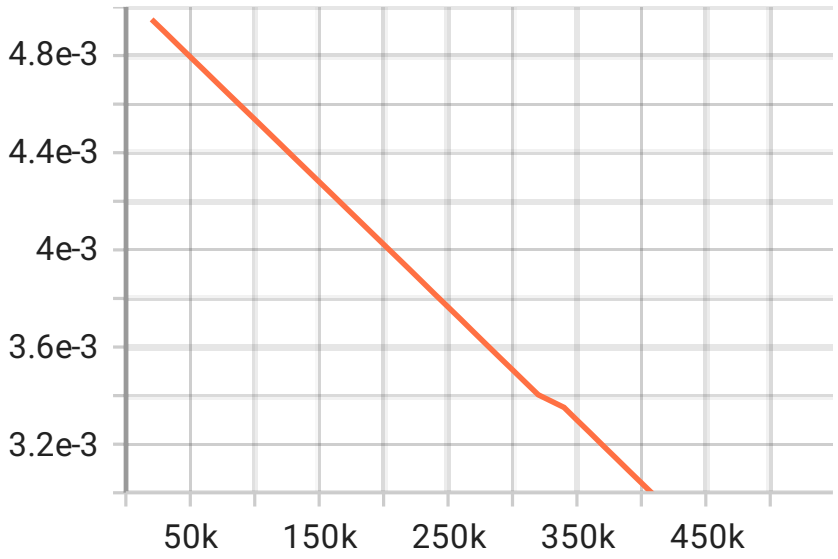

Figura 11: Beta

4.2 Risultati seconda fase di training

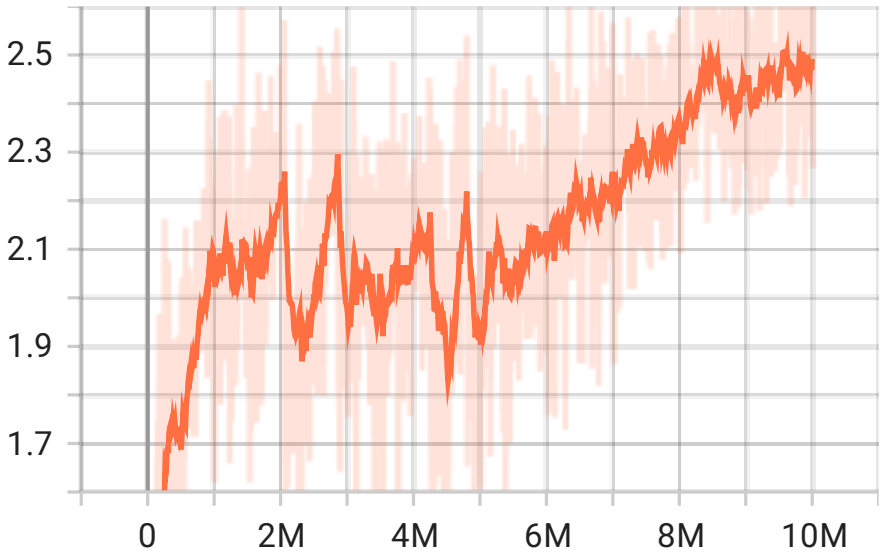

Figura 12: Cumulative Reward

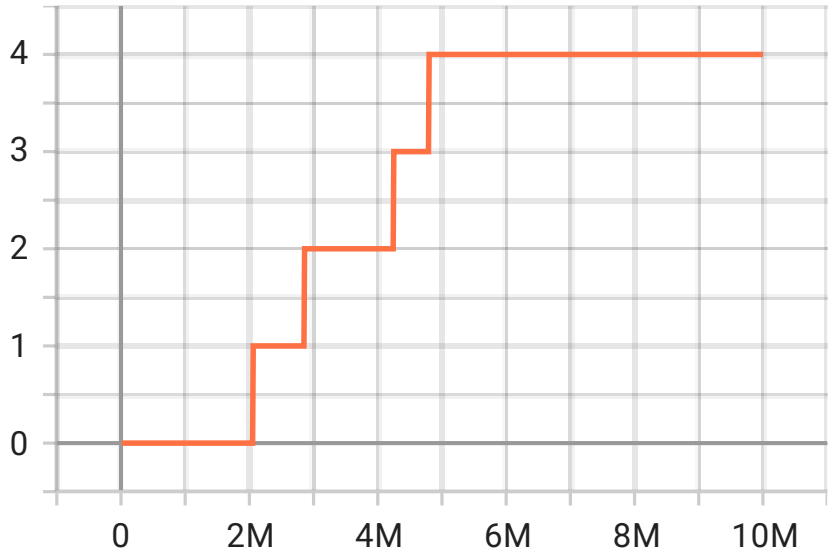

Figura 13: Indice della lezione

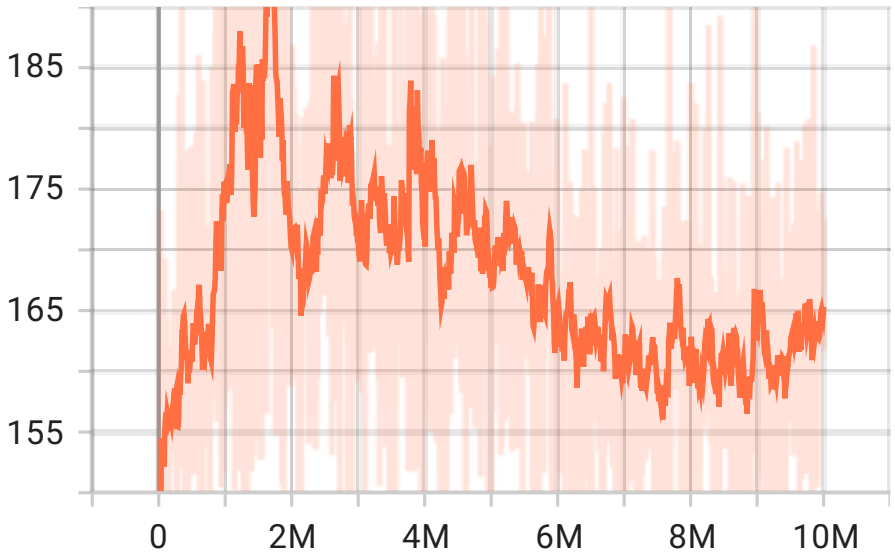

Figura 14: Episode Length

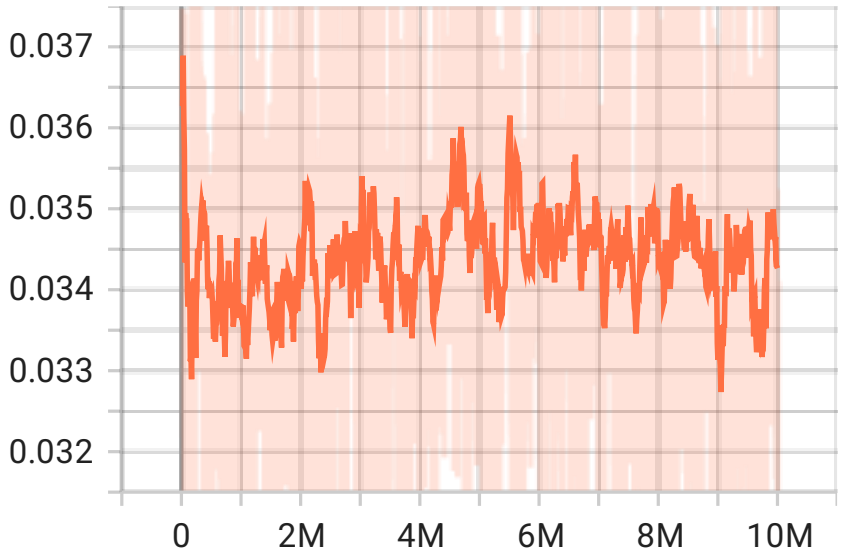

Figura 3: Policy Loss

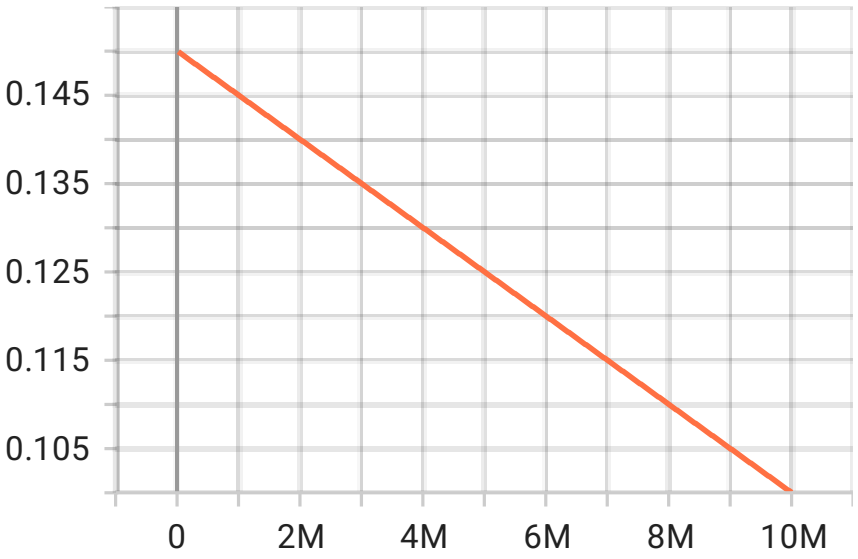

Figura 16: Epsilon

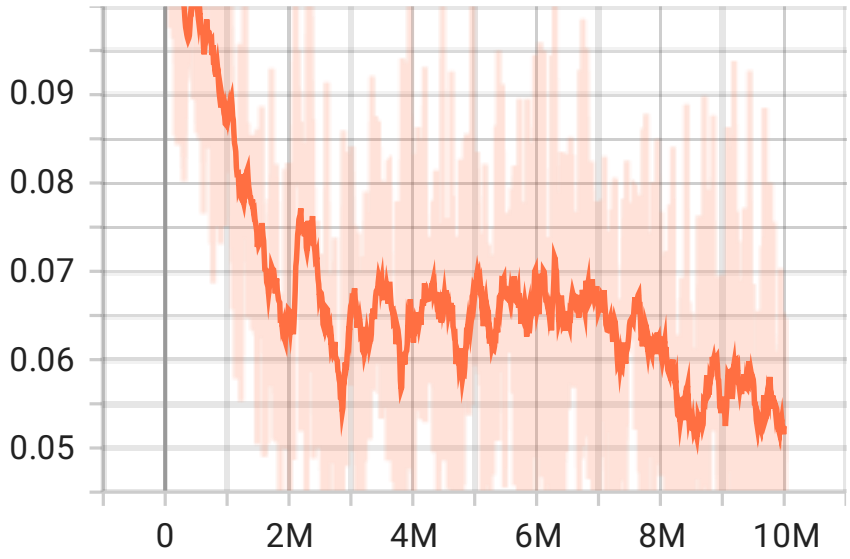

Figura 17: Value Loss

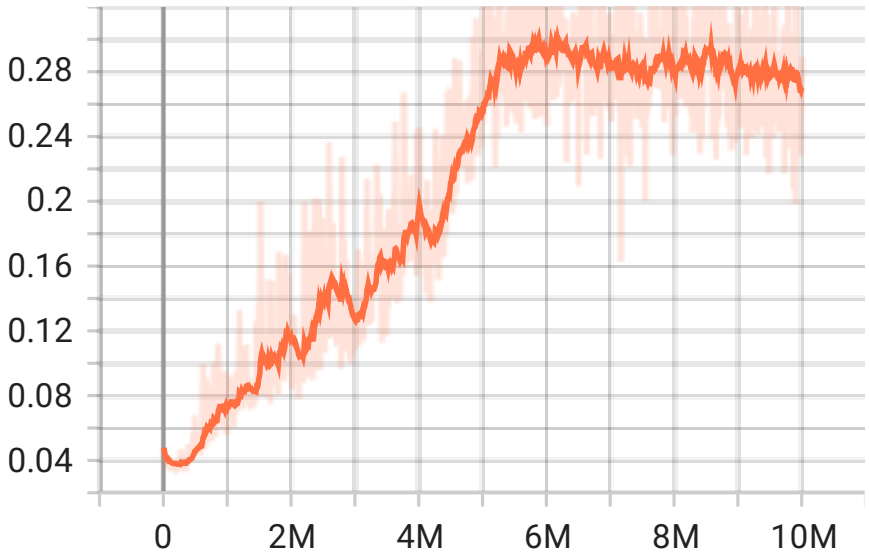

Figura 18: Entropia

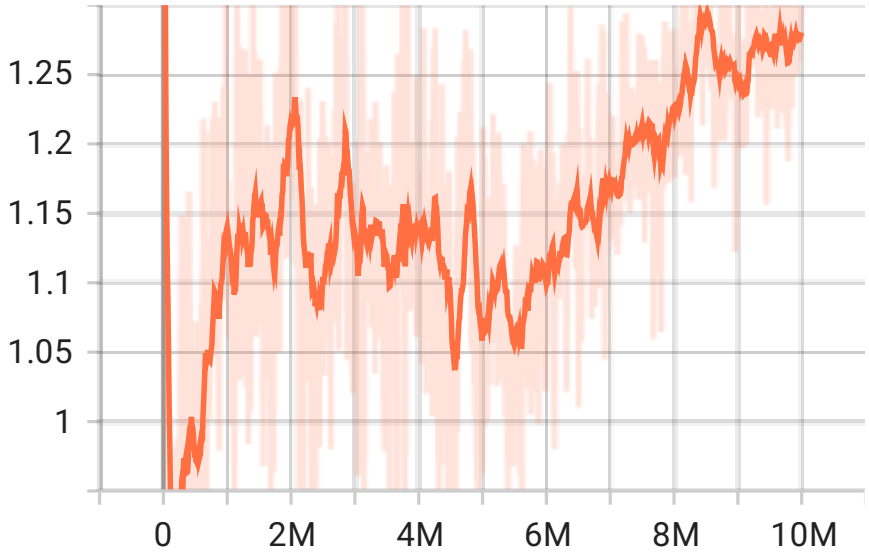

Figura 19: Extrinsic Value Estimate

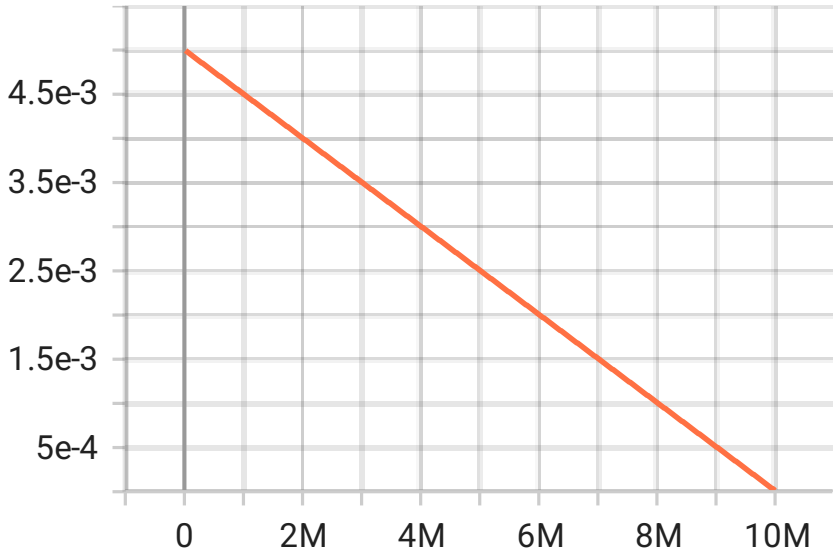

Figura 20: Beta

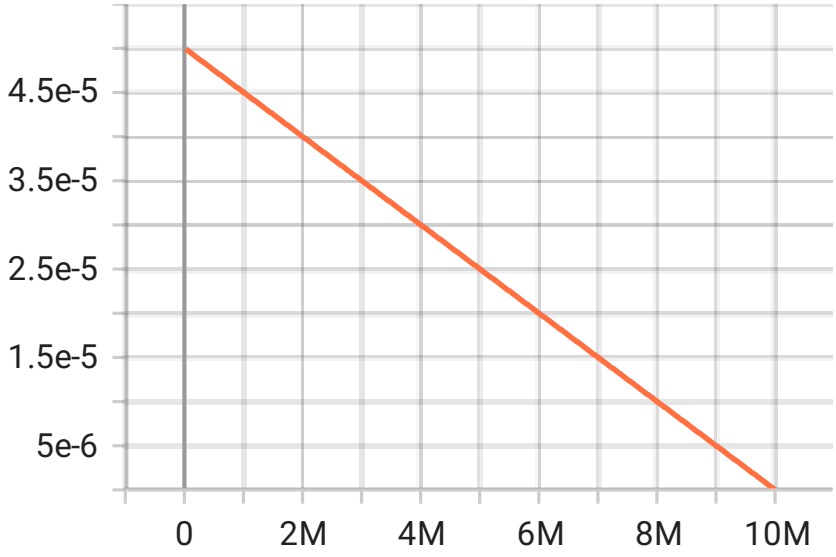

Figura 21: Learning rate

4.3 File di configurazione per il training

4.3.1 File config primo training

```
behaviors:
  CannonTraining:
    trainer_type: ppo
    hyperparameters:
      batch_size: 512
      buffer_size: 10240
      learning_rate: 0.0003
      beta: 0.005
      epsilon: 0.2
      lambd: 0.95
      num_epoch: 3
      learning_rate_schedule: linear
      beta_schedule: linear
      epsilon_schedule: linear
    network_settings:
      normalize: true
      hidden_units: 256
      num_layers: 3
      vis_encode_type: simple
      memory: null
      goal_conditioning_type: hyper
      deterministic: false
    reward_signals:
      extrinsic:
        gamma: 0.99
        strength: 1.0
        network_settings:
          normalize: false
          hidden_units: 128
          num_layers: 2
          vis_encode_type: simple
          memory: null
          goal_conditioning_type: hyper
          deterministic: false
    init_path: null
    keep_checkpoints: 5
    checkpoint_interval: 500000
    max_steps: 1000000
    time_horizon: 512
    summary_freq: 10000
    threaded: true
    self_play: null
    behavioral_cloning:
      demo_path: Demos\AimbotStacked_1.demo
```

```
      steps: 0
      strength: 0.5
      samples_per_update: 0
      num_epoch: null
      batch_size: null
```

4.3.2 File config secondo training

```
behaviors:
  CannonTraining:
    trainer_type: ppo
    hyperparameters:
      batch_size: 512
      buffer_size: 10240
      learning_rate: 5.0e-05
      beta: 0.005
      epsilon: 0.15
      lambd: 0.95
      num_epoch: 3
      learning_rate_schedule: linear
      beta_schedule: linear
      epsilon_schedule: linear
    network_settings:
      normalize: true
      hidden_units: 256
      num_layers: 3
      vis_encode_type: simple
      memory: null
      goal_conditioning_type: hyper
      deterministic: false
    reward_signals:
      extrinsic:
        gamma: 0.99
        strength: 1.0
        network_settings:
          normalize: false
          hidden_units: 128
          num_layers: 2
          vis_encode_type: simple
          goal_conditioning_type: hyper
          deterministic: false
    init_path: results\ImitationLearning\CannonTraining\checkpoint.pt
    keep_checkpoints: 5
    checkpoint_interval: 500000
    max_steps: 10000000
    time_horizon: 512
    summary_freq: 10000
    threaded: true
environment_parameters:
```

```
  enemy_speed:
    curriculum:
    - value:
        sampler_type: uniform
        sampler_parameters:
          seed: 5651
          min_value: 0.0
          max_value: 1.0
      name: Lesson0
      completion_criteria:
        behavior: CannonTraining
        measure: reward
        min_lesson_length: 100
        signal_smoothing: true
        threshold: 2.5
        require_reset: false
    - value:
        sampler_type: uniform
        sampler_parameters:
          seed: 5652
          min_value: 0.0
          max_value: 2.0
      name: Lesson1
      completion_criteria:
        behavior: CannonTraining
        measure: reward
        min_lesson_length: 100
        signal_smoothing: true
        threshold: 2.5
        require_reset: false
    - value:
        sampler_type: uniform
        sampler_parameters:
          seed: 5653
          min_value: 0.0
          max_value: 3.0
      name: Lesson2
      completion_criteria:
        behavior: CannonTraining
        measure: reward
        min_lesson_length: 100
        signal_smoothing: true
        threshold: 2.5
        require_reset: false
    - value:
        sampler_type: uniform
        sampler_parameters:
          seed: 5654
          min_value: 0.0
```

```
91.           max_value: 4.0
92.       name: Lesson3
93.       completion_criteria:
94.         behavior: CannonTraining
95.         measure: reward
96.         min_lesson_length: 100
97.         signal_smoothing: true
98.         threshold: 2.5
99.         require_reset: false
100.    - value:
101.        sampler_type: uniform
102.        sampler_parameters:
103.          seed: 5655
104.          min_value: 0.0
105.          max_value: 5.0
106.      name: Lesson4
107.      completion_criteria: null
```

La documentazione ufficiale per i file di configurazione si trova al link ml-agents/Training-Configuration-File.md at main · Unity-Technologies/ml-agents · GitHub.

4.4 Test baseline

I test finali sono stati effettuati su 100 episodi mettendo in competizione l'agente con i pesi finali e due giocatori umani (50 episodi ciascuno). I risultati sono i seguenti:

Tabella 1: Risultati di test, il grassetto indica il valore migliore

Giocatore	***% hit***	***Max dist miss***	***Min dist miss***	***Media dist miss***	***Reward medio***	***Tempo medio episodio (step)***
Agente	**83%**	**10.7**	1.3	**3.85**	**2.44**	**181.38**
Umano (baseline)	51%	12	**0.75**	6.22	1.41	340.19

È possibile notare come l'agente sia molto più consistente nel numero di bersagli colpiti e anche nella distanza media quando i bersagli vengono mancati. È interessante vedere come i giocatori umani abbiano ottenuto una distanza minima dal bersaglio in un caso di **miss** più bassa della controparte dell'agente. Infine, un punto importante riguarda la velocità di esecuzione del task che, nel caso dell'agente è risultata essere quasi del doppio più veloce.

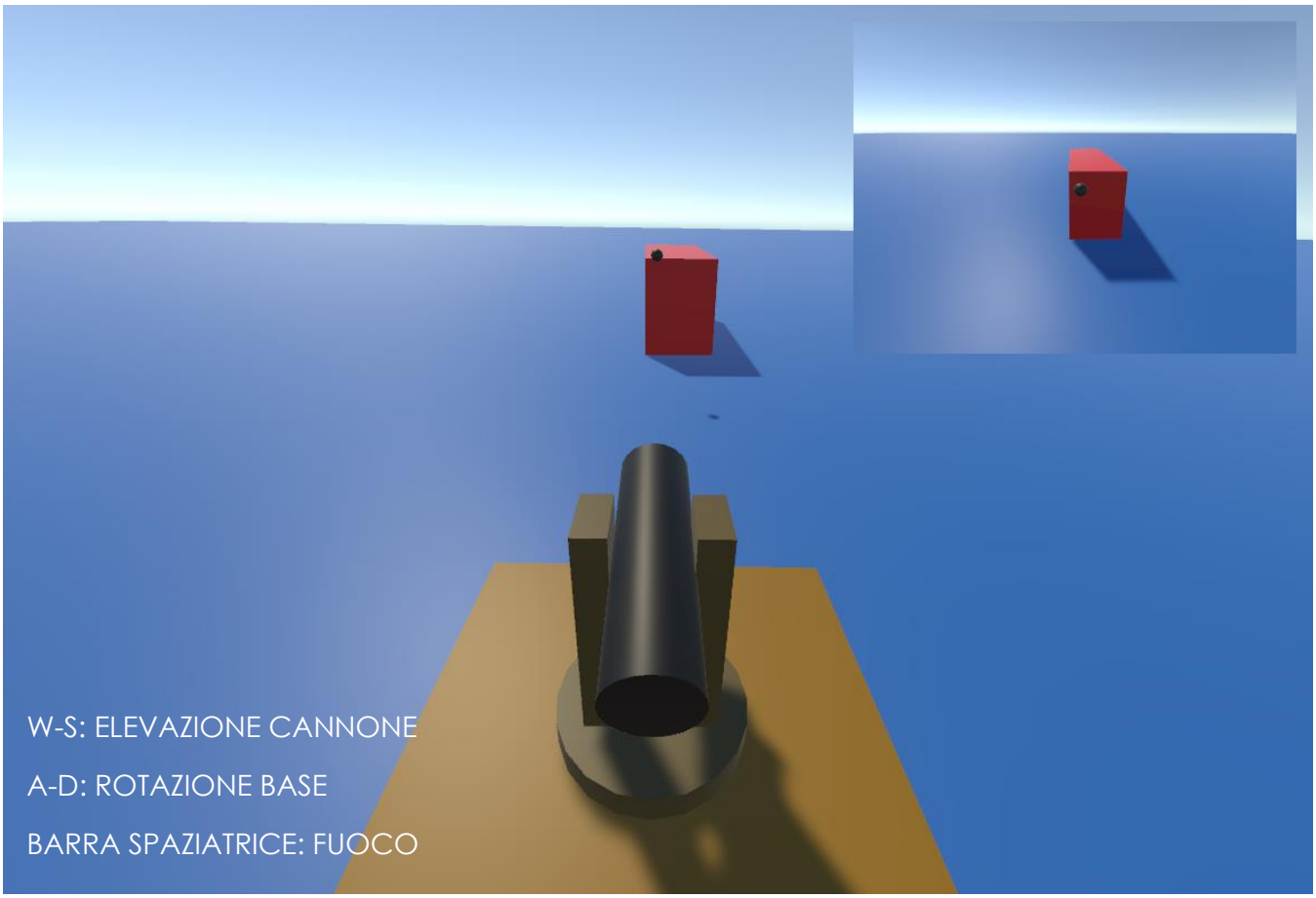

Figura 22: Screenshot d'esempio di un episodio giocato da un umano

5. Conclusione

In conclusione, con questo elaborato vogliamo confermare l'efficacia della combinazione Imitation Learning e Reinforcement Learning; il primo training, molto breve (circa 30 minuti) è stato sufficiente per garantire dei buoni pesi di partenza per poter affrontare la seconda fase, durata circa 5 ore. Possiamo aspettarci quindi una durata molto maggiore senza la prima fase di training. L'approccio basato su Curriculum si è dimostrato utile per questa applicazione, in quanto dall'inizio di ogni lezione l'agente affronta delle situazioni che non sono molto diverse da quelle della lezione precedente, ma sufficientemente distanziate in modo da poter fare dei progressi.

In seguito, sono stati effettuati ulteriori training a partire dai pesi finali utilizzati per i test in questo documento, senza però raggiungere dei risultati soddisfacenti: in tutti gli esperimenti le performance peggioravano gradualmente con il passare del tempo. Questo è dovuto probabilmente alla scelta degli iperparametri. Ad ogni modo, non abbiamo esplorato a fondo ulteriori direzioni per migliorare le performance in quanto riteniamo abbastanza soddisfacente il risultato raggiunto ai fini del progetto.

Infine, ci sono varie strade percorribili per estendere il seguente elaborato:

- Complicare le traiettorie: può essere effettuato considerando traiettorie rettilinee che non seguano la direzione precisa verso l'agente, ma che si muovano verso un intorno di punti attorno all'agente.
- Ampliare il range di velocità possibili
- Includere nella formulazione del problema il movimento dell'agente
- Aumentare il numero di nemici
- Addestrare più agenti in competizione considerando il movimento

Ai fini dell'esame ci siamo riservati di non complicare ulteriormente il problema per non estendere troppo i processi di training; tuttavia, in futuro gli spunti possibili sono molteplici, come mostrato precedentemente.

Riferimenti

[1] «Unity,» Unity Technologies, [Online]. Available: https://unity.com/.

[2] A. Juliani , V.-P. Berges, E. Teng, A. Cohen, J. Harper, C. Elion, C. Goy, Y. Gao, H. Henry, M. Mattar e D. Lange , «Unity: A general platform for intelligent agents,» *arXiv preprint arXiv:1809.02627,* 2020.

[3] «Learning-Environment-Design-Agents,» [Online]. Available: https://github.com/Unity-Technologies/ml-agents/blob/main/docs/Learning-Environment-Design-Agents.md#raycast-observations.

[4] Sutton e Barto, Reinforcement Learning: an introduction, 2017.

[5] J. Schulman, S. Levine, P. Moritz, M. Jordan e P. Abbeel, «Trust Region Policy Optimization,» 2015.

[6] J. Schulman, F. Wolski, P. Dhariwal, A. Radford e O. Klimov, «Proximal Policy Optimization Algorithms,» *OpenAI,* 2017.

[7] C. C.-Y. Hsu, C. Mendler-Dünner e M. Hardt, «Revisiting Design Choices in Proximal Policy Optimization,» 2020.

[8] S. Ross, G. J. Gordon e J. A. Bagnell, «A Reduction of Imitation Learning and Structured Prediction to No-Regret Online Learning,» 2011.

[9] S. Ross e J. A. Bagnell, «Efficient reductions for imitation learning,» 2010.

[10] Y. Bengio, J. Louradour, R. Collobert e J. Weston, «Curriculum Learning,» 2009.

[11] P.-W. Chou, D. Maturana e S. Scherer, «Improving stochastic policy gradients in continuous control with deep reinforcement learning using the Beta distribution,» 2017.