UNIVERSITÀ DEGLI STUDI DI CAGLIARI
INFORMATICA APPLICATA E DATA ANALYTICS
CORSO DI DATA VISUALIZATION

PING PONG

Corrias Samuele 60/79/00090
Pedranghelu Daniele 60/79/00061

Abstract

In questo progetto, creiamo un agente che impara, tramite Reinforcement Learning, a giocare a ping pong in un ambiente virtuale. In seguito all'addestramento, l'agente (una racchetta virtuale) è in grado di colpire la pallina e di giocare contro un bot.

Introduzione

Pong è uno dei videogiochi arcade più conosciuti che ha fatto la sua comparsa nel 1972, diventando uno dei primi successi nell'industria dei videogiochi. Creato dai fondatori di Atari, Pong rappresenta una simulazione virtuale a due dimensioni del classico ping-pong, coinvolgendo uno o talvolta due giocatori che controllano ognuno una barra verticale sullo schermo. L'obiettivo è analogo a ping-pong: respingere una pallina da un lato all'altro dello schermo senza farla cadere nel proprio bordo. La semplicità del gioco ha reso Pong un gioco iconico nel mondo dei videogiochi.

In questo progetto l'obiettivo è di creare un Pong, più avanzato graficamente e funzionalmente, dove l'agente (che nel caso di Pong, corrisponde ad una racchetta) imparerà a giocare contro un avversario (che in questo caso sarà un bot).
Le racchette si potranno muovere nell'asse *y* e *z* in uno spazio predefinito all'interno del campo di gioco di Unity e verrà addestrata con l'obiettivo di giocare per più tempo possibile contro un'altra racchetta.

L'idea iniziale si basava su rendere un gioco tridimensionale, facendo muovere la racchetta anche nell'asse *x* (che corrisponde al movimento in orizzontale verso l'altra racchetta). Questo faceva si che oltre a rendere l'apprendimento più lungo, venivano riscontrati dei problemi nel modo in cui la racchetta imparava: l'agente rimaneva costantemente nella metà campo per poter restituire immediatamente la palla. Nonostante i risultati, si è arrivati a una decisione: rimuovere il movimento nell'asse *x* delle racchette poiché possedevano uno stile di gioco poco realistico, quindi sono state impostate le racchette ad un movimento bidimensionale lungo asse *y* e *z*.

L'ambiente è una sala da ping-pong composta da un tavolo posizionato al centro e dei muri invisibili posizionati sulle estremità più lunghe di esso, per far sì che la pallina non esca dal campo e interrompa la partita tra le racchette.
L'agente è dotato di sensori di movimento posti davanti a essa, che permettono alla racchetta di poter vedere il tavolo e la palla.
L'agente può percorrere l'asse *y* e l'asse *z* mentre avrà una forza prestabilita nell'asse *x* per poter rispedire la palla dall'altra parte del campo.
L'avversario del nostro agente è la racchetta destra. Essa è un bot e per essere creato è venuto in aiuto un

paper online[1]. Nonostante il paper mostri un bot per il tennis, è stato possibile adattarlo al nostro progetto.

Essendo un progetto di Reinforcement Learning ed avendo utilizzato ML-Agents, l'agente subisce penalità e ricompense (*rewards*) in base ai suoi comportamenti, fondamentale per poter imparare e migliorare nel tempo.

Allo stato dell'arte non si è riusciti a trovare paper affini alla ricerca per due principali motivazioni, ML-Agents è a tutti gli effetti una libreria molto recente e quando si attua un Reinforcement Learning su Pong ci si basa principalmente sul gioco Atari con lo scopo di creare un bot che superi le prestazioni umane, rendendo il gioco di fatto ingiocabile.
È stato possibile trovare dei paper analoghi come [2]: questo progetto si focalizza sull'implementazione di un ambiente di gioco basato su Pong per la piattaforma Gym di OpenAI.
Gym di OpenAI è un altro tool-kit con scopi similari a ML-Agents che supporta lo sviluppo e il confronto di algoritmi di apprendimento per rinforzo.
In questo contesto, sono stati creati quattro agenti di intelligenza artificiale. Due di essi vengono addestrati con approcci diversi: uno tramite una rete neurale convoluzionale (CNN) basata su Deep Q-Learning (una variabile del Deep Learning) e l'altro con una tecnica più semplice di Q-Learning con un range di valori d'apprendimento α compreso tra lo 0 e 1, dove 0 indica un training dove il modello si basa principalmente sui vecchi episodi, mentre 1 se il modello tende alle informazioni correnti che sta apprendendo. Gli altri due agenti invece utilizzano caratteristiche predefinite prima dell'addestramento per generare azioni ottimali.
Lo scopo principale di questo lavoro è molto sperimentale, risiede nella capacità di addestrare una rete neurale su un gioco specifico e confrontare i modelli.

Tuttavia cercando altri paper similari al progetto che è stato creato è possibile trovare [3] che è un progetto molto analogo a quello che si vuole creare. È possibile notare come abbiano riscontrato le stesse limitazioni, ovvero la restrizione del gioco a due assi *xy*, limitando l'apprendimento dell'agente. E inoltre, la mancanza di risorse computazionali ha rallentato notevolmente il loro processo di addestramento.

Strumenti e metodi

Unity
È stato optato l'utilizzo di Unity come motore di gioco per il nostro progetto, poiché le sue caratteristiche offrono la possibilità di creare ambienti ideali per l'addestramento degli agenti. Le funzionalità di Unity consentono la generazione di ambienti con condizioni ideali per il training, mentre la sua presenza di forme primitive come sfere, cubi e piani rende agevole la modellazione degli scenari di

addestramento.
In particolare, la semplicità nella gestione delle collisioni tramite un sistema di tag assegnato a ciascun oggetto è stato di grande aiuto. Inoltre, tramite lo store degli asset di Unity, è possibile scaricare il modello 3D della scena di gioco come racchette, tavolo, pareti, semplificando ulteriormente lo sviluppo del progetto[4].

C#
All'interno dell'ambiente Unity viene utilizzato il linguaggio di programmazione C#. Essendo un linguaggio ad oggetti, garantisce un approccio conforme alla struttura gerarchica dell'engine.
Tutta la logica dell'ambiente e il movimento della racchetta e della pallina, tra cui la logica della racchetta avversaria (bot) sono state gestiti tramite degli script in C#.

ML-Agents
ML-Agents è un toolkit che fornisce tutte le architetture e gli algoritmi per l'implementazione degli agenti. Il supporto per l'addestramento parallelo di più agenti ha permesso di velocizzare i tempi di training.
Per l'installazione e l'utilizzo ci siamo aiutati grazie a diversi tools online, tra cui diversi video su YouTube[5].

Enviroment
Il training dell'agente è stato svolto in un ambiente virtuale che permette di gestire le dipendenze dei pacchetti Python.
In questo ambiente sono stati installati diversi pacchetti necessari per il training come *Pytorch*, *Numpy* ed *ML-Agents*.

Tensorboard
È stato utilizzato Tensorboard per visualizzare graficamente i diversi training. Questo framework, collegato all'ambiente di simulazione ci ha permesso di tenere traccia di diversi parametri utili per capire se l'agente stesse imparando o meno.

Reinforcement Learning
Il Reinforcement Learning è un'area del machine learning che si occupa di come un'agente può imparare a compiere delle azioni in un ambiente tramite delle ricompense.
Il processo di Reinforcement Learning coinvolge tipicamente tre figure principali: l'*agente*, l'*ambiente* e le *azioni*.

Raycast
Il Raycast è una tecnica che viene utilizzata nei motori di gioco o nelle simulazioni che permette, tramite dei raggi virtuali, di determinare delle intersezioni con altri oggetti presenti sulla scena. L'agente, emette un raggio da un punto di origine in una direzione specifica e determina se e dove questo raggio interseca un oggetto nel mondo virtuale. Unity e altri motori forniscono API per eseguire facilmente delle operazioni di raycasting.

In questo progetto, la racchetta poteva vedere la posizione del tavolo e della pallina, aiutandola a capire dove

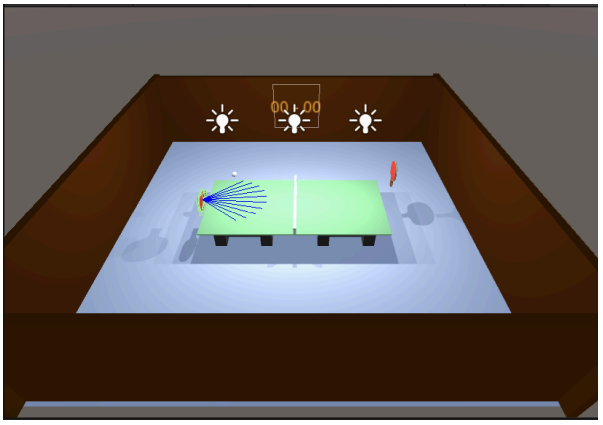
dovesse andare per colpire la pallina [6].

Modello PPO
L'algoritmo di Reinforcement Learning che è stato utilizzato è il modello PPO. Questo algoritmo ha aiutato ad ottimizzare le politiche di decisione dell'agente (racchetta) in modo da ottimizzare la ricompensa cumulativa nel tempo.

Implementazione

Creazione della scena
L'ambiente di addestramento consiste in uno spazio tridimensionale *x***y***z*.
Le racchette sono impostate con un movimento bidimensionale *y***z* mentre la pallina può muoversi in tutte e tre le dimensioni.

All'inizio dell'episodio la pallina parte da una posizione casuale lungo l'asse *z* del tavolo da gioco e la racchetta ha il compito di colpire la pallina per mandarla dall'altra parte del tavolo.
Una volta che la pallina viene colpita dall'agente e mandata nel campo avversario, il bot rimanda la pallina verso l'agente che a sua volta dovrà colpire la pallina con lo scopo di giocare contro l'altra racchetta.



Ricompense e penalizzazioni
Nel caso in cui il nostro agente colpisce la pallina viene dato un reward positivo di +0.5, ugualmente se la pallina riesce a sorpassare il bot.
Mentre, nel caso in cui la racchetta non riuscisse a prendere la pallina viene punita con un reward negativo di -1, con anche la fine dell'episodio.

Bot
Il bot ha un funzionamento molto semplice e intuitivo: abbiamo progettato il bot dandogli sempre la posizione della palla in modo che esso possa raggiungerla più volte possibile.
Per rendere possibile la sconfitta del bot abbiamo scelto di dargli un *movement speed* minore rispetto all'agente, così che una volta che il match sta iniziando a diventare competitivo (dalle 5 alle 6 battute) il bot inizierà ad avere difficoltà a raggiungere la palla in tempo, facendo sì che non riesca a prendere la pallina, causando la sua sconfitta.

Palla

La palla utilizzata in questo progetto è un oggetto sferico tridimensionale con una texture appropriata per renderla più verosimile ad una pallina da ping-pong reale.

È dotata di un RigidBody che ne permette di gestire la fisica, presenta una massa di 0.3 ed è sensibile alla gravità. È associato un componente Collider che permette di gestire le collisioni con gli altri oggetti dell'ambiente, come le racchette e il tavolo. Per simulare il movimento della pallina reale, nonché quello di rimbalzare quando viene colpita è stato aggiunto anche un "*bounce*" con un *bounciness* uguale a 1, un *attrito* medio e un *rimbalzo* impostato al massimo empiricamente.

Il movimento della pallina è stato controllato tramite script (*Ball.cs*). È importante evidenziare come si comporti la pallina all'inizio dell'episodio: essendo un oggetto tridimensionale si trova con gli assi *x* e *y* prestabiliti mentre l'asse *z* è una posizione random tra i limiti del tavolo da gioco. Una volta fatta partire la simulazione del gioco, la pallina cade dall'altezza prestabilita, rimbalzando sul tavolo che risulta essere leggermente inclinato per far arrivare la pallina verso il nostro agente. Questa situazione si ripete ogni volta che l'episodio inizia, con la differenza che dal secondo episodio la pallina parte da una posizione più alta.

Agente Racchetta

A differenza della pallina, il nostro agente, come già anticipato si muove bidimensionalmente sugli assi *y* e *z*.

I suoi movimenti sono limitati in altezza (asse *y*) per non far andare la racchetta troppo in alto o troppo in basso, mentre sul lato corto del tavolo (asse *z*) per non far uscire le racchette dall'area di gioco delimitata dai muri.

In caso di mancanza di input da parte dell'utente l'agente rimarrà fermo.

L'implementazione del movimento è stata realizzata con azioni continue perché il movimento della racchetta deve variare in modo fluido e continuo, consentendo una maggiore precisione e adattabilità per l'utente durante il gioco.

L'agente si muove con una velocità fissa (*moveSpeed* = 5f) che viene impostata dallo script che controlla l'agente (*MoveAgent.cs)*.

Per quanto riguarda la vecchia implementazione, nella quale, la racchetta agente aveva un movimento tridimensionale le differenze a livello di impostazioni della racchetta erano veramente sottili: la più importante riguardava quando la racchetta colpiva la pallina, infatti non era applicata una forza fissa, ma variava in base alla velocità della racchetta (considerata dal *Ridigbody velocity*) nell'asse *x*.

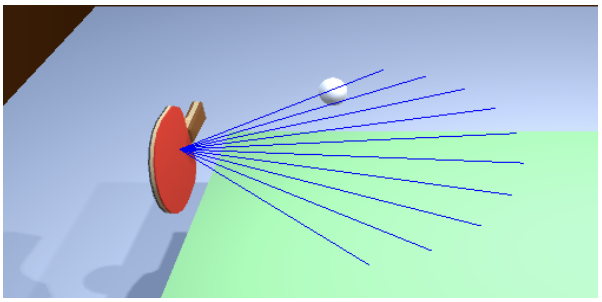
Osservazioni

La racchetta prende decisioni e si migliora osservando l'ambiente che la circonda.

Le metodologie di osservazione che abbiamo adottato sono:

- Posizione e Rigidbody *y*z* dell'agente (float);
- Posizione e Rigidbody *x*y*z* della palla (float);
- Velocità normalizzata *x*y*z* dell'agente e della palla (float);
- I raycast, tramite questi raggi, l'agente individua la posizione del tavolo da gioco e della pallina.

Per quanto riguarda l'impostazione del Raycast sono stati utilizzati 10 raggi dalla lunghezza di +1f con una rosa di 180°. Il Raycast è gestito tramite script *RayCast.cs* e identifica gli oggetti con tag *Ball* e *Table*.



Risultati

Per valutare i risultati e l'apprendimento effettivo dell'agente, si è utilizzato i grafici offerti da *Tensorboard*.

I grafici che riguardano due contesti specifici, il primo che mostrerà i risultati definitivi dell'addestramento, creati in una situazione di gioco bidimensionale con l'utilizzo di Raycast e l'aggiunta di cinque *Stacked Vectors* e il secondo grafico che mostra un addestramento precedente in una situazione di gioco con il movimento della racchetta tridimensionale e un Raycast con raggi più corti e che permetteva la visione solo della palla.

All'interno di un ambiente dove la racchetta si può muovere in bidimensionalmente sono stati ottenuti i risultati migliori, grazie anche ad un implementazione migliorata del Raycast, con aggiunta delle osservazioni di tavolo e pallina.

In particolare si può osservare come, dopo un piccolo punto di stallo verso i 400 mila steps, che la curva inizi ad aumentare in modo più costante e ripido, questo ci ha fatto pensare che il modello potesse essere addestrato per molti altri steps, ma una volta provato ad addestrare per ulteriori episodi, il modello tendeva a bloccarsi e non apprendere più, l'addestramento ci ha richiesto diverse ore e crediamo sia un problema legato a risorse computazionali.

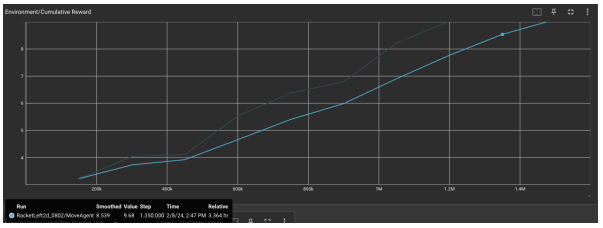
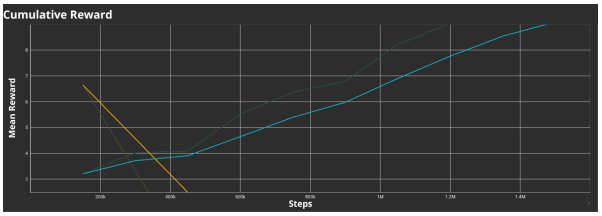


Grafico dell'ultimo training effettuato - 2D

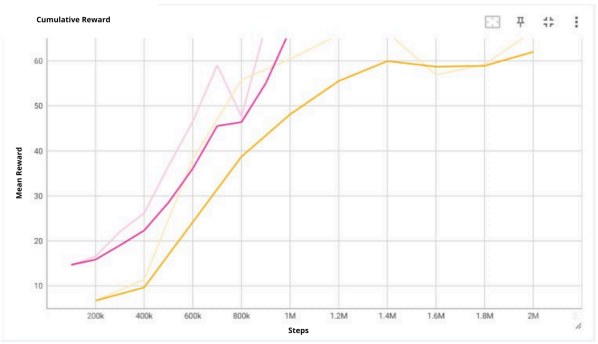


Linea Blu: training effettuato in 2D
Linea Gialla: training inizializzato dalla linea Blu

Come si può vedere dal seguente istogramma, la distribuzione della partita è molto eterogenea ma nel complesso migliora con il passare del tempo, si crede di avere questi risultati perché essendo che la racchetta e la palla ad ogni episodio sono generati in una posizione casuale la racchetta può essersi specializzata in diversi casi (per esempio, nel caso la palla sia davanti la racchetta ad inizio episodio o se parte da una specifica altezza).

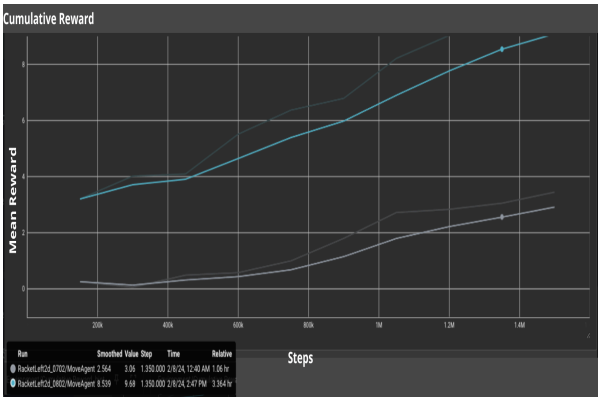
Istogramma dell'ultimo training effettuato - 2D

Mentre, è possibile notare come, quando si aveva un ambiente tridimensionale con reward non standardizzati, la situazione non fosse delle migliori: guardando il grafico al momento dei training sembrava che la racchetta avesse imparato, ma guardando effettivamente la racchetta giocare si poteva notare che l'agente non apprendeva, giocando in maniera scorretta.



Esempio di due training 3D con reward non standardizzati

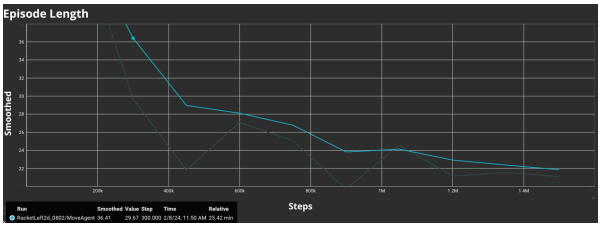
Per avere un confronto equo sono stati standardizzati i reward del Pong in tre dimensioni.
Come possiamo notare abbiamo riscontrato risultati peggiori nel CR a tre dimensioni, empiricamente invece si riscontra che dopo 1.5 milioni di step circa il gioco a tre dimensioni fa ancora fatica a giocare bene mentre quello bidimensionale riesce a tenere testa al bot per un paio di battute. È possibile che sia dovuto al fatto che il gioco a tre dimensioni debba disporre di più tempo e risorse computazionali per poter apprendere dato che, a differenza dell'altro, la forza dipenda dalla spinta dell'agente e non da una forza prestabilita.



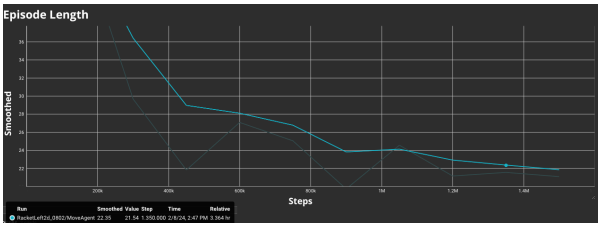
Linea Blu: CR con training 2D, reward standardizzati - Linea Grigia: CR con training 3D, reward standardizzati

Viene mostrato ora il grafico che raffigura la lunghezza degli episodi.

Possiamo notare come, all'inizio l'agente impieghi circa 36 azioni per finire l'episodio, mentre dopo 1.2 milioni di step impiega circa 22 azioni.
Da qui si può intuire che la racchetta sarà più precisa nei movimenti.



Lunghezza episodio primo confronto 36 azioni dopo 25 minuti circa - 2D



Lunghezza episodio secondo confronto 22 azioni dopo 3.30 ore circa - 2D

È importante notare che il processo di training richiede una maggiore potenza computazionale e che non è stato possibile addestrare in parallelo più agenti.

Conclusione e Sviluppi futuri

In conclusione, il nostro progetto ha dimostrato che tramite il Reinforcement Learning è possibile insegnare ad una racchetta a giocare a Pong. I grafici mostrano come non siamo riusciti a raggiungere risultati perfetti a livello di prestazioni. Ci riteniamo soddisfatti dei risultati ottenuti.

Riguardo sviluppi futuri ci potremmo concentrare in un addestramento di ambo le racchette, un tuning degli iperparametri molto più consistente ed in una conversione del progetto in tre dimensioni riguardante il movimento della racchetta sull'asse *x*, che potrebbe imparare a colpire con una determinata forza in base al suo movimento.
Per questi aggiornamenti ci dovremmo servire sicuramente di una potenza computazionale maggiore.

Bibliografia

[1] - Making a tennis game in Unity: *github.com/sinoriani/Unity-Projects/tree/master/Tennis%20Game*

[2] - Playing Atari Pong in RL: *github.com/netanelhugi/Pong-RL/blob/master/final%20report.pdf*

[3] - Learning to play Table Tennis using Multi-agent Reinforcement Learning: *sowmyavoona96.github.io/csci527/TP%20(2).pdf*

[4] - Asset utilizzati in Unity: *assetstore.unity.com/packages/3d/props/low-poly-table-tennis-set-181749*

[5] - ML-Agents basics Code Monkey: *youtube.com/playlist?list=PLzDRvYVwl53vehwiN_odYJkPBzcqFw110&si=cyV5RJJsP3qpqytt*

[6] - Raycast in Unity: *youtu.be/B34iq4O5ZYI?si=BQ4YO7l-7yR8RgfR*