

Sviluppo di un Agente Intelligente per il Gioco Connect 4

Abstract

Connect 4 è un gioco in cui due giocatori competono per allineare quattro pedine consecutive in direzione orizzontale, verticale, obliqua destra e obliqua sinistra, su una griglia 6x7.

L'obiettivo di questo progetto è lo sviluppo di un agente intelligente capace di apprendere strategie vincenti attraverso tecniche di **Reinforcement Learning**. In particolare, sono stati utilizzati e confrontati due algoritmi: **DQN** (Deep Q-Network) e **PPO** (Proximal Policy Optimization), con l'intento di valutare la loro efficacia nel prendere decisioni ottimali contro avversari di diversa difficoltà.

La simulazione si svolge in un ambiente virtuale che replica fedelmente le regole di Connect 4. L'agente apprende a giocare attraverso l'interazione con avversari, ricevendo ricompense in base alle azioni effettuate e agli esiti delle partite.

Durante lo sviluppo sono state affrontate diverse sfide, tra cui la gestione delle mosse illegali, la definizione di un sistema di ricompense, la scelta degli iperparametri e la complessità dello spazio degli stati che, a causa delle numerose configurazioni possibili della griglia, risulta estremamente vasto.

Infine, sono state analizzate le prestazioni degli agenti in diversi scenari per valutare la robustezza delle strategie apprese.

Introduzione al Problema

Nel gioco Connect 4, due giocatori si alternano nel posizionare pedine su una griglia 6x7, cercando di allinearne quattro in fila per vincere. L'agente opera in uno spazio discreto, ovvero un ambiente in cui le azioni possibili (le colonne in cui inserire una pedina) sono finite e ben definite. Ogni mossa può influenzare l'esito della partita.

Per affrontare questa sfida, è stato progettato un ambiente virtuale che replica fedelmente le regole del gioco e consente di variare il livello dell'avversario. L'agente può confrontarsi con avversari casuali, oppure con agenti rule-based di livello crescente. Questo approccio progressivo rientra nel cosiddetto Curriculum Learning, una tecnica che prevede l'addestramento dell'agente partendo da compiti semplici e aumentando gradualmente la complessità, favorendo un apprendimento più stabile ed efficace.

L'agente apprende attraverso l'interazione con l'ambiente, ricevendo ricompense in base alla qualità delle mosse effettuate: azioni valide, vittorie, sconfitte, difese riuscite e attacchi strategici vengono premiati o penalizzati secondo un sistema di reward progettato ad hoc.

Il progetto si articola in diverse fasi:

- progettazione dell'ambiente di gioco,
- definizione del sistema di ricompense,
- addestramento degli agenti,
- valutazione delle prestazioni,
- confronto tra algoritmi.

In particolare, sono stati analizzati i comportamenti di due algoritmi di reinforcement learning: **DQN (Deep Q-Network)** e **PPO (Proximal Policy Optimization)**.

La repository contenente il codice sviluppato è disponibile al seguente link:

https://github.com/DaMa29A/Connect4_AI

Progettazione dell'Ambiente di Gioco

L'ambiente virtuale sviluppato per questo progetto è stato realizzato utilizzando la libreria **Gymnasium**, un framework ampiamente utilizzato per la creazione di ambienti di simulazione compatibili con algoritmi di *reinforcement learning*. Gymnasium facilita l'integrazione con librerie come **Stable-Baselines3** e altri tool di training avanzato.

L'ambiente è implementato nel file **Connect4Env.py** e riproduce fedelmente le regole del gioco Connect 4 su una griglia 6x7.

La classe **Connect4Env** estende **gym.Env** e definisce gli spazi di azione e osservazione:

- **action_space**: rappresenta le 7 colonne disponibili per inserire una pedina.
- **observation_space**: rappresenta lo stato della board, come matrice 6x7 con valori -1, 0, 1 per O, vuoto e X rispettivamente.

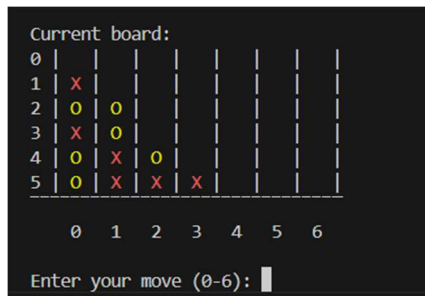
Durante l'inizializzazione, vengono definite alcune variabili fondamentali per gestire lo stato del gioco:

- **opponent_symbol**: indica se l'avversario gioca come X (1) o O (-1).
- **opponent**: è la classe dell'agente avversario (es. RandomAgent), che viene istanziata e integrata nell'ambiente.
- **render_mode**: può essere "**console**", "**gui**" o **None**, e determina il tipo di visualizzazione del tabellone.
- **first_move_random**: opzionale, permette di randomizzare il primo turno.
- **self.first_player = 1**: stabilisce che il giocatore X (rappresentato da 1) è sempre il primo a giocare.
- **self.next_player_to_play = self.first_player**: tiene traccia di chi deve effettuare la prossima mossa.
- **self.opponent_symbol**: memorizza il simbolo assegnato all'avversario.
- **self.board**: inizializza il tabellone come matrice 6x7 piena di zeri, dove ogni cella rappresenta una posizione vuota.
- **self.last_move_row** e **self.last_move_col**: registrano la posizione dell'ultima mossa effettuata.
- **self.winner**: inizializza lo stato del vincitore, che verrà aggiornato solo alla fine della partita.

Il metodo **step** è il cuore dell'interazione tra agente e ambiente. Verifica se la mossa è valida, aggiorna il tabellone inserendo la pedina, calcola le ricompense in base all'esito della mossa (vittoria, pareggio, sconfitta, difesa o attacco), e gestisce il cambio turno. Qui viene avviata anche la **gestione delle mosse illegali**. Se l'agente tenta di inserire una pedina in una colonna piena, riceve una penalità. Tuttavia, la partita non viene terminata: l'agente è costretto a riprovare finché non seleziona una mossa valida. Questa scelta progettuale è stata adottata per favorire l'apprendimento corretto del vincolo di validità, evitando che l'agente associ penalità a mosse casuali (come accadrebbe se si sostituisse la mossa con una random). Il numero di tentativi non è limitato, ma l'agente riceve penalità ad ogni errore, incentivando la convergenza verso comportamenti validi.

L'ambiente supporta due modalità di rendering:

- **Console:** stampa testuale del tabellone, utile per il debugging.

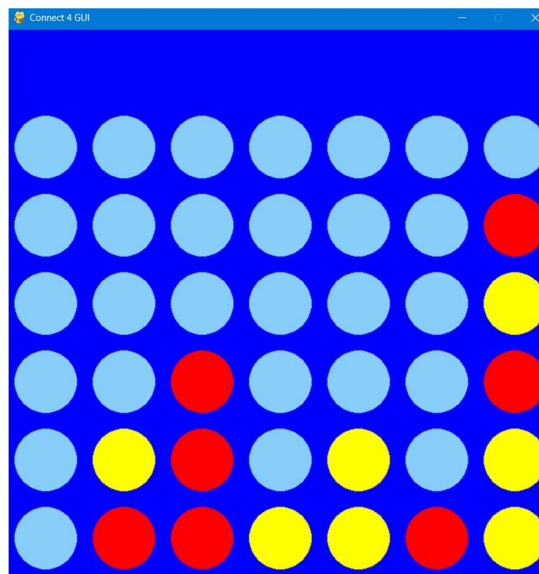


Current board:

0							
1		X					
2		O		O			
3		X		O			
4		O		X		O	
5		O		X		X	
	0	1	2	3	4	5	6

Enter your move (0-6):

- **GUI:** visualizzazione grafica tramite **pygame**, con pedine rosse per X e gialle per O. I file **gui_rend.py** e **gui_config.py** gestiscono rispettivamente il rendering e le configurazioni visive (dimensioni, colori, font). La griglia viene aggiornata in tempo reale ad ogni mossa, rendendo il rendering utile per la valutazione visiva delle strategie apprese.



Agenti

Il progetto prevede diversi tipi di agenti, ciascuno con comportamenti e strategie distinti:

- **HumanAgent:** permette l'interazione diretta dell'utente con l'ambiente, giocando manualmente.
- **RandomAgent:** sceglie casualmente una mossa tra quelle valide disponibili.
- **RuleBasedL1Agent:** applica regole semplici, come completare una sequenza di quattro pedine per ottenere la vittoria.
- **RuleBasedL2Agent:** estende le logiche del livello 1 aggiungendo capacità difensive. Oltre a cercare mosse vincenti, rileva anche potenziali minacce dell'avversario (come una tripla) e le blocca tempestivamente.
- **DQNAgent:** agente basato su RL che utilizza una rete neurale con architettura Deep Q-Network.
- **PPOAgent:** altro agente RL, fondato sull'algoritmo Proximal Policy Optimization, anch'esso supportato da una rete neurale.

Reinforcement Learning

Il **reinforcement learning** (RL), o *apprendimento per rinforzo*, è una tecnica di *machine learning* in cui un agente impara a compiere azioni ottimali interagendo con un ambiente. L'apprendimento avviene per tentativi ed errori, attraverso l'assegnazione di **ricompense** per azioni corrette e **penalità** per quelle errate. L'obiettivo è che l'agente sviluppi una **strategia** capace di massimizzare le ricompense totali nel tempo, raggiungendo così il comportamento desiderato.

Nel progetto Connect 4, il reinforcement learning è stato utilizzato per addestrare due tipi di agenti: **DQN (Deep Q-Network)** e **PPO (Proximal Policy Optimization)**. Entrambi permettono all'agente di apprendere comportamenti strategici, ma si distinguono per il tipo di policy adottata:

- DQN

Il DQN (Deep Q-Network) è un algoritmo **off-policy** (impara da esperienze passate, indipendentemente dalla politica attuale) e **value-based** (il suo obiettivo è imparare il valore di ogni azione in ogni stato).

Rappresenta un'evoluzione del Q-Learning classico. Mentre il Q-Learning tradizionale utilizza una tabella (chiamata Q-table) per memorizzare il valore $Q(s, a)$ di ogni coppia stato-azione, il DQN sostituisce la Q-table con una rete neurale profonda (DNN) con pesi θ . Questa rete non memorizza i valori, ma impara ad approssimarli: $Q(s, a) \approx Q(s, a; \theta)$.

L'obiettivo non è più aggiornare un valore della tabella, ma **ottimizzare i pesi θ della rete** affinché l'errore tra la predizione $Q(s, a; \theta)$ e il target sia minimo. Questo si ottiene minimizzando una **Loss Function** (funzione di perdita), tipicamente l'Errore Quadratico Medio (MSE), tramite *backpropagation*.

Formula Q-Learning:

$$Q(s, a) \leftarrow Q(s, a) + \alpha \left[\underbrace{\left(r + \gamma \max_{a'} Q(s', a') \right)}_{\text{Target}} - Q(s, a) \right]$$

- $Q(s, a)$ è il valore attuale
- $r + \gamma \max_{a'} Q(s', a')$ è il **Target**, ovvero la stima aggiornata basata sulla ricompensa r ricevuta e sul miglior valore futuro $\max_{a'} Q(s', a')$
- α è il learning rate (tasso di apprendimento)
- L'intero termine $[(r + \gamma \max_{a'} Q(s', a')) - Q(s, a)]$ è l'**Errore Temporale Differenziale (TD Error)**

Formula Loss Function DQN:

$$L(\theta) = \mathbb{E}_{(s, a, r, s') \sim D} \left[(Y_{\text{target}} - Q(s, a; \theta))^2 \right]$$

Dove il **Target** Y_{target} (che rappresenta l'equazione di Bellman) è:

$$Y_{\text{target}} = r + \gamma \max_{a'} Q(s', a'; \theta^-)$$

- $Q(s, a; \theta)$ è il Q-value che la rete neurale principale (con pesi θ) attualmente stima per la coppia stato-azione (s, a) .
- Y_{target} è il valore che la rete *dovrebbe* predire, calcolato usando l'equazione di Bellman. Rappresenta la ricompensa immediata r più il miglior Q-value futuro stimato (scontato da γ).
- D è il *Replay Memory*, un buffer di memoria che archivia le transizioni passate (s, a, r, s') . Il simbolo $E_{(s,a,r,s') \sim D}$ significa che stiamo calcolando la media della loss su un *mini-batch* di campioni estratti casualmente da questa memoria.
- θ : I pesi della rete neurale principale (quella che viene addestrata attivamente).
- θ^- : I pesi della *Target Network*, una copia della rete principale che viene aggiornata più lentamente per dare stabilità al Y_{target} .
- r : La ricompensa immediata ricevuta dopo aver eseguito l'azione a nello stato s .
- γ : Il *discount factor* (fattore di sconto), che bilancia l'importanza delle ricompense immediate rispetto a quelle future.

- PPO

PPO (Proximal Policy Optimization) è un algoritmo *on-policy* (impara dall'esperienza raccolta dalla politica *attuale*) e *policy-based* (il suo obiettivo è ottimizzare direttamente la politica dell'agente). Spesso viene implementato in un'architettura **Actor-Critic**. In questa architettura, l'agente è diviso in due parti: l'**Actor**, che decide l'azione, e il **Critic**, che valuta l'efficacia di quell'azione.

A differenza del DQN, che impara il *valore* di ogni azione (approccio *value-based*), PPO addestra direttamente una **politica** (policy) stocastica, $\pi_{\theta}(a|s)$. Questa è una rete neurale (con parametri θ) che mappa uno stato s a una distribuzione di probabilità sulle possibili azioni a ed è conosciuta come l'**Actor** (l'attore), poiché è la componente che *decide quale azione compiere*.

PPO ottimizza la politica (l'Actor) cercando di massimizzare una funzione obiettivo che stima il miglioramento della performance. Per far sì che l'apprendimento avvenga in modo stabile, PPO utilizza un meccanismo di **clipping** (limitazione) che impedisce agli aggiornamenti della politica di essere troppo grandi e destabilizzanti.

L'obiettivo (noto come L^{CLIP}) che l'Actor cerca di massimizzare è:

$$L^{\text{CLIP}}(\theta) = \mathbb{E}_t \left[\min \left(r_t(\theta) \hat{A}_t, \text{clip}(r_t(\theta), 1 - \epsilon, 1 + \epsilon) \hat{A}_t \right) \right]$$

Per capire questa formula, i due termini chiave sono:

- \hat{A}^t (Advantage): Fornito dalla seconda rete, il Critic. \hat{A}^t ci dice quanto l'azione a_t (scelta dall'Actor) è stata migliore (vantaggio positivo) o peggiore (vantaggio negativo) rispetto alla media delle azioni possibili in quello stato s_t .
- $r_t(\theta)$ (Probability Ratio): Misura quanto la nuova politica (che stiamo ottimizzando) sia diversa dalla vecchia politica (quella che ha raccolto i dati):

$$r_t(\theta) = \frac{\pi_{\theta}(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)}$$

Funzione di reward

La funzione di reward è stata progettata per incentivare comportamenti strategici e non solo la vittoria finale. Le ricompense includono:

- Mossa valida
- Mossa illegale
- Vittoria
- Sconfitta
- Pareggio
- Creazione di una sequenza di 3
- Creazione di una sequenza di 2
- Blocco di una sequenza di 3 avversaria
- Blocco di una sequenza di 2 avversaria

Questa struttura consente all'agente di apprendere anche da **mosse intermedie**, migliorando la qualità delle strategie apprese. Durante lo sviluppo, è stato osservato che l'inserimento delle ricompense per **attacco e difesa** ha significativamente migliorato l'apprendimento rispetto a una funzione di reward più semplice.

Libreria utilizzata: Stable-Baselines3

Per implementare gli algoritmi DQN e PPO è stata utilizzata la libreria **Stable-Baselines3**, che offre versioni robuste e ottimizzate degli algoritmi RL più diffusi. Questa scelta ha permesso di concentrarsi sulla progettazione dell'ambiente e sulla configurazione degli iperparametri.

Architettura della rete neurale

La rete neurale di default in Stable-Baselines3 è una **MLP (Multi-Layer Perceptron)** composta da:

- **Due hidden layer** completamente connessi da **64 neuroni** ciascuno, con attivazione **ReLU**
- **Un output layer** con dimensione pari al numero di azioni possibili (7 colonne), che rappresentano:
 - o i **Q-values** nel caso di DQN
 - o le **probabilità di azione** nel caso di PPO

Nel corso del progetto è stata testata anche una configurazione alternativa più profonda da [128, 128, 64] neuroni, che verrà discussa nei paragrafi successivi.

Training

Il processo di addestramento degli agenti DQN e PPO è stato strutturato secondo una logica di **curriculum learning**, ovvero un approccio progressivo in cui l'agente affronta avversari di difficoltà crescente. Questo metodo consente di facilitare l'apprendimento iniziale e di consolidare strategie più complesse man mano che l'agente acquisisce competenze.

Il curriculum è stato articolato in tre fasi:

1. **Avversario casuale** (RandomAgent)
2. **Avversario rule-based di livello 1** (RuleBasedL1Agent)
3. **Avversario rule-based di livello 2** (RuleBasedL2Agent)

Per ciascun avversario, sono stati condotti due tipi di training:

- **Con prima mossa casuale:** la prima pedina di ogni partita viene posizionata in una colonna scelta casualmente.
- **Con prima mossa controllata:** l'agente decide autonomamente dove posizionare la prima pedina.

Questa distinzione è stata introdotta per incentivare l'esplorazione ed evitare l'overfitting su pattern iniziali ricorrenti. La casualità iniziale costringe l'agente ad adattarsi a configurazioni diverse, migliorando la generalizzazione.

Il training è stato misurato in **time steps**, dove un time step rappresenta una singola interazione tra agente e ambiente.

Dettaglio delle sessioni di training

Algoritmo	Avversario	Prima mossa	Time steps
DQN	RandomAgent	Casuale	50_000
DQN	RandomAgent	Controllata	50_000
DQN	RuleBasedL1Agent	Casuale	150_000
DQN	RuleBasedL1Agent	Controllata	150_000
DQN	RuleBasedL2Agent	Casuale	225_000
DQN	RuleBasedL2Agent	Controllata	225_000
PPO	RandomAgent	Casuale	50_000
PPO	RandomAgent	Controllata	50_000
PPO	RuleBasedL1Agent	Casuale	150_000
PPO	RuleBasedL1Agent	Controllata	150_000
PPO	RuleBasedL2Agent	Casuale	225_000
PPO	RuleBasedL2Agent	Controllata	225_000

Le sessioni di training documentate nella tabella precedente sono state progettate per stabilire una baseline di performance. Sebbene questo numero di iterazioni non sia sufficiente per raggiungere una convergenza completa dell'agente (per cui sarebbero necessari molti più time steps), questa fase è stata fondamentale per osservare le tendenze di apprendimento iniziali e l'impatto della difficoltà crescente degli avversari (da RandomAgent a RuleBasedL2Agent).

In aggiunta, sono stati condotti **esperimenti** variando:

- gli **iperparametri**
- la **funzione di reward**
- la **struttura della rete neurale**.

Queste varianti hanno permesso di analizzare l'impatto di ciascun componente sull'efficacia dell'apprendimento, evidenziando come la combinazione di reward shaping e architettura più profonda migliori la capacità dell'agente di affrontare avversari più sofisticati.

Risultati

Al termine del training, sono stati condotti test di valutazione per entrambi gli agenti DQN e PPO. Ogni modello è stato testato su **500 partite** contro ciascun avversario di riferimento:

- 500 partite contro **RandomAgent**
- 500 partite contro **RuleBasedL1Agent**
- 500 partite contro **RuleBasedL2Agent**

Infine, è stato effettuato anche uno scontro diretto tra i due agenti: **DQN vs PPO**, per analizzare le differenze strategiche apprese.

Oltre al conteggio di **vittorie, sconfitte e pareggi**, sono stati raccolti dati più dettagliati relativi a **difese e attacchi strategici**, calcolati tramite il modulo **get_statistics.py**. Questi indicatori permettono di valutare non solo il risultato finale, ma anche la qualità tattica delle mosse effettuate.

Difesa riuscita

Una difesa viene considerata riuscita quando l'agente blocca una minaccia avversaria. In particolare: Se l'avversario ha una fila di 3 pedine espandibili a 4 (es. O ha 3 pedine) e l'agente (es. X) inserisce una pedina che interrompe la sequenza, viene registrata una difesa attuata.

Attacco riuscito

Un attacco viene considerato riuscito quando l'agente estende una propria sequenza. Ad esempio, se l'agente ha una fila di 3 pedine consecutive e aggiunge la quarta, o ha una fila di 2 ed aggiunge la terza, viene registrato un attacco.

Configurazioni DQN

Config	Rete	LR	Buffer	Batch	Gamma	Target Upd	Exp Final	Exp Fract	Create 3	Block 3	Create 2	Block 2	Valid
1	[64,64]	1E-03	50 K	64	0.99	500	0.05	0.3	0	0	0	0	0
2	[128,128,64]	1E-03	50 K	64	0.99	500	0.05	0.3	0	0	0	0	0
3	[128,128,64]	1E-04	50 K	64	0.99	1000	0.01	0.8	0	0	0	0	0
4	[128,128,64]	1E-04	50 K	64	0.99	1000	0.01	0.8	0.4	0.4	0.1	0.1	-0.01
5	[128,128,64]	1E-04	100 K	64	0.995	1000	0.01	0.8	0.4	0.4	0.1	0.1	-0.01
6	[128,128,64]	2E-04	200 K	128	0.995	1000	0.01	0.8	0.4	0.4	0.1	0.1	-0.01

Tutte le configurazioni DQN testate utilizzano un'architettura di rete neurale [128, 128, 64], ad eccezione della Config 1 ([64, 64]). Le principali differenze risiedono nel **learning rate**, nella dimensione del **buffer** di replay, nel fattore **gamma**, nei parametri di **esplorazione** e nell'introduzione del **reward shaping** (ricompense intermedie).

Dall'analisi dei risultati emerge un chiaro percorso di miglioramento. Le **Configurazioni 1-3**, prive di reward shaping, mostrano **scarse performance** contro l'agente RBL2 (da 0.4% a 7.8%). L'introduzione del **reward shaping nella Configurazione 4** ha portato a un **salto significativo** (21% vs RBL2). La **Configurazione 5**, ottimizzando ulteriormente il **buffer size** e il **gamma**, ha raggiunto le **prestazioni migliori in assoluto**, con percentuali di vittoria elevate contro tutti gli

Config	vs Rnd	vs RBL1	vs RBL2
1	84.2%	69.4%	0.4%
2	71.2%	43.0%	2.6%
3	72.6%	45.6%	7.8%
4	85.6%	61.4%	21.0%
5	93.6%	78.6%	27.6%
6	69.6%	46.8%	16.2%

avversari (93.6% vs Random, 78.6% vs RBL1, 27.6% vs RBL2). Al contrario, la **Configurazione 6**, che tentava di migliorare ulteriormente modificando buffer, batch size e learning rate, ha mostrato un **netto peggioramento**.

Si può quindi affermare che l'introduzione del **reward shaping** è stata fondamentale e la **Configurazione 5 rappresenta la scelta ottimale** tra quelle testate per DQN, ottenendo il miglior bilanciamento tra iperparametri e ricompense.

Risultati PPO

Config	Rete	LR	n_steps	Batch	n_epochs	Gamma	gae_lambda	clip_range	ent_coef
1	[128,128,64]	3E-04	4096	512	10	0.995	0.95	0.2	0.01
2	[128,128,64]	3E-04	2048	256	10	0.995	0.95	0.2	0.01
3	[128,128,64]	2.5E-04	1024	128	10	0.995	0.95	0.2	0.01
4	[128,128,64]	3E-04	1024	128	10	0.995	0.95	0.2	0.01

Tutte le configurazioni condividono la stessa architettura di rete neurale [128,128,64], ma differiscono per il learning rate, il numero di passi per aggiornamento (*n_steps*) e la dimensione del batch.

Dall’analisi dei risultati emerge che la **Configurazione 2** mostra le performance più equilibrate e complessivamente migliori nei confronti degli avversari più semplici, con l’**88.8% di vittorie contro Random** e il **64.2% contro RuleBasedL1**. Tuttavia, la **Configurazione 3** si distingue nettamente per le prestazioni contro l’avversario più difficile, **RBL2 (35.4%)**, evidenziando una maggiore capacità di adattamento a scenari più complessi. Le configurazioni 1 e 4, invece, risultano inferiori su tutte le metriche.

Config	vs Rnd	vs RBL1	vs RBL2
1	85.8%	61.2%	27.2%
2	88.8%	64.2%	27.4%
3	85.6%	61.4%	35.4%
4	79.0%	48.4%	27.0%

Si può affermare che la **Configurazione 2** rappresenta la scelta ideale per un comportamento complessivamente stabile, mentre la **Configurazione 3** appare più promettente quando l’obiettivo è massimizzare le prestazioni contro avversari forti.

Risultati finali

Match	Vittorie Agente 1	Vittorie Agente 2
DQN vs PPO	29.6%	70.4%
PPO vs DQN	68.2%	31.8%

Nei confronti diretti tra le configurazioni migliori per ciascun algoritmo (DQN Config 5 e PPO Config 3), PPO è emerso come l'agente più forte. Nelle partite giocate l'uno contro l'altro, PPO ha ottenuto la vittoria in circa il 70% dei casi, indipendentemente da quale agente iniziasse la partita. Questa superiorità potrebbe essere attribuita alla natura *on-policy* di PPO, che forse si adatta meglio alle dinamiche del *curriculum learning* e del *reward shaping* impiegati in questo specifico ambiente.

Conclusioni

Il progetto ha dimostrato l'efficacia del **Reinforcement Learning** nell'addestramento di agenti intelligenti per il gioco Connect 4. Attraverso l'impiego degli algoritmi **DQN** e **PPO**, integrati in un ambiente Gymnasium personalizzato, gli agenti hanno iniziato ad apprendere strategie complesse, superando avversari basati su regole di crescente difficoltà. L'adozione di un approccio basato sul **Curriculum Learning** con una **funzione di reward strutturata** (*reward shaping*), si è rivelata fondamentale per guidare l'agente verso comportamenti strategicamente validi.

Sviluppi futuri

Il progetto apre a numerose possibilità di estensione e miglioramento:

1. **Tuning avanzato degli iperparametri:** Sebbene la sperimentazione manuale abbia identificato configurazioni promettenti, l'impiego di tecniche di ottimizzazione automatica potrebbe rivelare combinazioni di iperparametri ancora più performanti per DQN e PPO, massimizzando l'efficacia dell'apprendimento.
- **Test di architetture neurali alternative:** oltre alla MLP, si potrebbero esplorare reti più profonde o architetture CNN (Convolutional Neural Networks), che potrebbero catturare meglio le strutture spaziali della board.
- **Introduzione di nuove ricompense intermedie:** ad esempio, penalità per mosse inutili, premi per controllo del centro, o per prevenzione di doppie minacce.
- **Estensione del Curriculum Learning e Self-Play:** Il curriculum attuale si ferma a RBL2. L'introduzione di avversari progressivamente più sofisticati e dedicando sessioni di training molto più lunghe (milioni di *time steps*) a queste fasi avanzate, è essenziale per raggiungere un livello di gioco superiore. In aggiunta, si potrebbe esplorare l'implementazione di tecniche di self-play, dove l'agente impara giocando contro versioni precedenti o attuali di sé stesso, potenzialmente superando le limitazioni degli agenti rule-based.