# 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 efficace, 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>

## Metodologia

### 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**, **Ray RLlib** 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 avviene 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.
* **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** (premi) per azioni corrette e **penalità** (rinforzi negativi) per quelle errate. L’obiettivo è che l’agente sviluppi una **politica** (o 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:

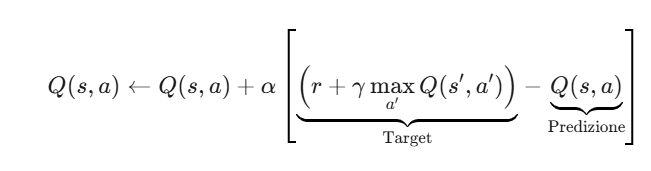
* **DQN**

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 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: .

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

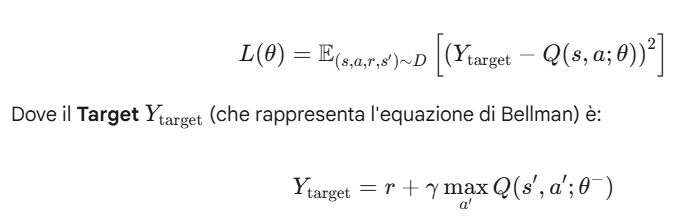
*Formula Q-Learning:*





* + è il valore attuale
  + è il **Target**, ovvero la stima aggiornata basata sulla ricompensa r ricevuta e sul miglior valore futuro
  + è il learning rate (tasso di apprendimento)
  + L'intero termine è l'**Errore Temporale Differenziale (TD Error)**

*Formula Loss Function DQN:*



* + è il Q-value che la rete neurale principale (con pesi ) 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 ).
  + **D** è il *Replay Memory*, un buffer di memoria che archivia le transizioni passate (. Il simbolo significa che stiamo calcolando la media della loss su un *mini-batch* di campioni estratti casualmente da questa memoria.
  + : I pesi della rete neurale principale (quella che viene addestrata attivamente).
  + : I pesi della *Target Network*, una copia della rete principale che viene aggiornata più lentamente per dare stabilità al .
  + **r**: La ricompensa immediata ricevuta dopo aver eseguito l'azione nello stato .
  + : Il *discount factor* (fattore di sconto), che bilancia l'importanza delle ricompense immediate rispetto a quelle future.
* **PPO**

Il **PPO** è un algoritmo *on-policy* e *policy-based*. A differenza del DQN, PPO addestra direttamente la **politica** (policy) dell'agente, $\pi\_\theta(a|s)$, ovvero una funzione che mappa gli stati alle probabilità di scegliere ciascuna azione.

PPO ottimizza la politica cercando di massimizzare una funzione obiettivo "surrogata", ma utilizza un meccanismo di *clipping* (limitazione) per evitare che gli aggiornamenti della politica siano troppo grandi e destabilizzanti.

L'**obiettivo surrogato (clipped)** di PPO è:

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

Ecco una versione rivista e completa del paragrafo sul PPO, formattata per la tua documentazione.

**PPO (Proximal Policy Optimization)**

**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).

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 $\theta$) che mappa uno stato $s$ a una distribuzione di probabilità sulle possibili azioni $a$.

L'innovazione chiave di PPO è ottimizzare la politica in modo stabile. Invece di calcolare un gradiente che potrebbe portare a un cambiamento drastico e destabilizzante della politica, PPO ottimizza una **funzione obiettivo "surrogata"** che limita la dimensione dell'aggiornamento.

**1. L'Obiettivo Surrogato (Clipped) - $L^{CLIP}$**

Questo è il cuore di PPO e riguarda l'aggiornamento della politica (l'**Actor**). L'idea è di incoraggiare le azioni che si sono rivelate "buone" e scoraggiare quelle "cattive", ma solo entro un intervallo di sicurezza.

La formula dell'obiettivo surrogato (clipped) è:

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

Spiegazione dei termini:

* $\hat{\mathbb{E}}\_t$: Indica la media calcolata sui campioni $(s\_t, a\_t)$ raccolti nell'ambiente.
* $r\_t(\theta)$ (Rapporto di Probabilità): Questo è il rapporto tra la probabilità dell'azione $a\_t$ data dalla nuova politica $\pi\_\theta$ (che stiamo ottimizzando) e quella data dalla vecchia politica $\pi\_{\theta\_{\text{old}}}$ (quella usata per raccogliere i dati):

$$r\_t(\theta) = \frac{\pi\_\theta(a\_t|s\_t)}{\pi\_{\theta\_{\text{old}}}(a\_t|s\_t)}$$

* + Se $r\_t(\theta) > 1$, la nuova politica rende quell'azione *più* probabile.
  + Se $r\_t(\theta) < 1$, la nuova politica la rende *meno* probabile.
* **$\hat{A}\_t$ (Vantaggio - Advantage):** Questo valore cruciale, calcolato da un'altra rete chiamata **Critic**, stima *quanto* l'azione $a\_t$ è stata migliore (vantaggio positivo) o peggiore (vantaggio negativo) rispetto all'azione media attesa nello stato $s\_t$.
* **$\epsilon$ (Epsilon):** Un piccolo iperparametro (es. 0.2) che definisce l'intervallo di *clipping* $[1 - \epsilon, 1 + \epsilon]$.
* **$\text{clip}(...)$:** Questa funzione "forza" $r\_t(\theta)$ a rimanere all'interno dell'intervallo $[1 - \epsilon, 1 + \epsilon]$.
* **$\min(...)$:** L'obiettivo prende il minimo tra il rapporto non limitato ($r\_t(\theta) \hat{A}\_t$) e quello limitato ($\text{clip}(...) \hat{A}\_t$).
  + **Perché?** Se il Vantaggio $\hat{A}\_t$ è **positivo** (azione buona), l'algoritmo non permetterà a $r\_t(\theta)$ di crescere oltre $1 + \epsilon$. Questo impedisce all'agente di diventare "troppo avido" e cambiare la politica troppo velocemente.
  + Se il Vantaggio $\hat{A}\_t$ è **negativo** (azione cattiva), l'algoritmo non permetterà a $r\_t(\theta)$ di scendere sotto $1 - \epsilon$. Questo limita la "penalità" per evitare aggiornamenti eccessivamente punitivi.

**2. La Loss Function Completa di PPO**

Nella pratica, PPO ottimizza una loss function combinata che include altri due termini per addestrare un algoritmo **Actor-Critic** completo:

$$L^{PPO}(\theta, \phi) = \hat{\mathbb{E}}\_t \left[ -L^{CLIP}(\theta) + c\_1 L^{VF}(\phi) - c\_2 S[\pi\_\theta](s\_t) \right]$$

Spiegazione dei nuovi termini:

* **$-L^{CLIP}(\theta)$:** La loss dell'Actor (politica). Si usa il segno meno perché gli ottimizzatori *minimizzano* la loss, mentre noi vogliamo *massimizzare* l'obiettivo $L^{CLIP}$.
* $L^{VF}(\phi)$ (Value Function Loss): Questa è la loss del Critic (con parametri $\phi$). È un errore quadratico medio (MSE) che addestra il Critic a stimare correttamente il valore (ricompensa futura attesa) di uno stato. È essenziale per calcolare il vantaggio $\hat{A}\_t$.

$$L^{VF}(\phi) = \left( V\_{\text{target}} - V\_{\phi}(s\_t) \right)^2$$

* **$S[\pi\_\theta](s\_t)$ (Bonus di Entropia):** Questo termine misura quanto è "casuale" o "indecisa" la politica. Si sottrae un bonus di entropia (o si aggiunge l'entropia negativa) per incoraggiare l'agente a **esplorare** di più e non convergere prematuramente a una politica sub-ottimale.
* $c\_1, c\_2$: Coefficienti (iperparametri) che bilanciano l'importanza dei tre termini della loss.

#### 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
* Vittoria
* Sconfitta
* Pareggio
* Mossa illegale
* Attacco riuscito (creazione di una tripla)
* Difesa riuscita (blocco di una tripla 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. L’agente ha iniziato a riconoscere configurazioni vantaggiose e a bloccare minacce in modo più efficace.

#### 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, senza dover implementare manualmente la logica degli algoritmi.

#### Perché usare una rete neurale

In Connect 4, lo spazio degli stati è estremamente vasto: ogni cella della griglia 6x7 può assumere tre valori (vuoto, X, O), generando milioni di configurazioni possibili. Utilizzare metodi tabellari come Q-learning classico richiederebbe una **tabella di dimensioni elevate**, difficile da gestire e generalizzare. Le reti neurali permettono invece di **approssimare la funzione di valore** in modo compatto e scalabile, apprendendo pattern strategici anche in stati mai visti.

#### 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:
  + i **Q-values** nel caso di DQN
  + le **probabilità di azione** nel caso di PPO

Nel corso del progetto è stata testata anche una configurazione alternativa con **128 neuroni per layer**, che verrà discussa nei paragrafi successivi dedicati all’ottimizzazione degli iperparametri.

### 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** e **evitare l’overfitting su pattern iniziali ricorrenti**, come l’abuso di colonne centrali. 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 | 100,000 |
| DQN | RandomAgent | Controllata | 100,000 |
| DQN | RuleBasedL1Agent | Casuale | 120,000 |
| DQN | RuleBasedL1Agent | Controllata | 120,000 |
| DQN | RuleBasedL2Agent | Casuale | 150,000 |
| DQN | RuleBasedL2Agent | Controllata | 150,000 |
| PPO | RandomAgent | Casuale | 100,000 |
| PPO | RandomAgent | Controllata | 100,000 |
| PPO | RuleBasedL1Agent | Casuale | 120,000 |
| PPO | RuleBasedL1Agent | Controllata | 120,000 |
| PPO | RuleBasedL2Agent | Casuale | 150,000 |
| PPO | RuleBasedL2Agent | Controllata | 150,000 |

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 **200 partite** contro ciascun avversario di riferimento:

* 200 partite contro **RandomAgent**
* 200 partite contro **RuleBasedL1Agent**
* 200 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. In particolare:

* Se l’agente ha una fila di **2 pedine consecutive** e aggiunge la **terza**, viene registrato un **attacco intermedio**.
* Se l’agente ha una fila di **3 pedine consecutive** e aggiunge la **quarta**, viene registrato un **attacco vincente** (che coincide con la vittoria).

#### Grafici

Queste metriche sono state utilizzate per generare **grafici comparativi** tra le diverse configurazioni testate, evidenziando:

* La percentuale di vittorie per ciascun agente contro ogni avversario
* Il numero medio di attacchi riusciti per partita
* Il numero medio di difese attuate

I grafici risultanti sono riportati di seguito e offrono una panoramica visiva dell’efficacia delle strategie apprese.

**Analisi delle configurazioni DQN con rete neurale 64×64 e 128×128**

Sono state testate cinque configurazioni dell’agente DQN, tutte basate su architetture multilayer perceptron. Le prime quattro utilizzano una rete con due layer da 64 neuroni, mentre la quinta adotta una rete più profonda da 128 neuroni per layer. Le varianti hanno riguardato l’ottimizzazione degli iperparametri e la definizione delle ricompense, con l’obiettivo di migliorare sia l’efficacia che la qualità strategica del comportamento.

**Configurazione 1**  
Con un learning rate elevato (1e-3) e batch size ridotto (64), l’agente ha mostrato buone performance contro avversari semplici (86.5% di vittorie contro Random), ma ha faticato contro strategie più strutturate (solo 13.5% contro Rule-Based L2). L’esplorazione limitata ha probabilmente favorito un comportamento rigido e poco adattivo.

**Configurazione 2**  
Riducendo il learning rate (1e-4), aumentando il batch size (128) e prolungando la fase di esplorazione (50% del training), l’agente ha migliorato la stabilità e la capacità di generalizzazione. Ha ottenuto l’87.0% di vittorie contro Random e ha raddoppiato il win-rate contro Rule-Based L2 (24.0%), segno di una strategia più matura e meno impulsiva.

**Configurazione 3**  
Introducendo una penalità più severa per le mosse non valide (-2.0), l’agente è diventato più cauto ma meno efficace. Il win-rate è sceso al 79.5% contro Random e al 5.5% contro Rule-Based L2. L’eccessiva punizione ha inibito l’esplorazione e ridotto la capacità di rischio, compromettendo l’apprendimento strategico.

**Configurazione 4**  
Reintroducendo la penalità standard (-1.0) e aggiungendo ricompense intermedie per mosse tattiche (+0.4 per triple create o bloccate), l’agente ha mostrato il miglior comportamento complessivo. Ha raggiunto il 93.0% di vittorie contro Random, l’84.5% contro Rule-Based L1 e il 31.5% contro Rule-Based L2. I reward intermedi hanno incentivato l’agente a costruire e difendere attivamente, migliorando la qualità delle decisioni e la profondità strategica.

**Configurazione 5 – Architettura 128×128**  
In questa variante, l’unico cambiamento riguarda l’architettura della rete neurale, che passa da [64, 64] a [128, 128]. L’obiettivo era verificare se una rete più profonda potesse migliorare la capacità dell’agente di riconoscere pattern strategici complessi.  
Contro RandomAgent, l’agente ha ottenuto 171 vittorie su 200 (85.5%), leggermente inferiore rispetto alla configurazione precedente. Contro Rule-Based L1 ha vinto 121 partite (60.5%), mostrando una flessione rispetto alla configurazione 4. Tuttavia, contro Rule-Based L2 ha ottenuto 74 vittorie (37.0%), il miglior risultato finora contro questo avversario.  
Questi dati indicano che la rete più profonda ha effettivamente migliorato la capacità dell’agente di affrontare situazioni complesse.

## Conclusioni

Il progetto ha dimostrato l’efficacia del reinforcement learning nell’addestramento di agenti intelligenti per il gioco Connect 4. Attraverso l’uso di algoritmi DQN e PPO, l’agente ha appreso strategie vincenti in un ambiente discreto e altamente strategico, riuscendo a competere con avversari di difficoltà crescente.  
L’approccio basato su **curriculum learning**, unito a una funzione di reward ben strutturata, ha permesso all’agente di sviluppare comportamenti non solo vincenti, ma anche tatticamente raffinati, come la capacità di bloccare minacce avversarie e costruire sequenze vantaggiose.

I risultati ottenuti evidenziano una buona generalizzazione contro avversari rule-based e una netta superiorità contro agenti casuali. L’inserimento di ricompense intermedie per attacco e difesa ha avuto un impatto positivo sull’apprendimento, migliorando la profondità strategica delle decisioni.

## Sviluppi futuri

Il progetto apre a numerose possibilità di estensione e miglioramento:

* **Tuning avanzato degli iperparametri**: l’uso di strumenti come Optuna o WandB potrebbe automatizzare la ricerca delle configurazioni ottimali, migliorando ulteriormente le prestazioni.
* **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 del tabellone.
* **Introduzione di nuove ricompense intermedie**: ad esempio, penalità per mosse inutili, premi per controllo del centro, o per prevenzione di doppie minacce.
* **Addestramento contro agenti più avanzati**
* **Analisi strategica post-partita**: sviluppare moduli che analizzano le partite giocate per identificare pattern ricorrenti, errori critici o mosse chiave.