# 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 fornisce un’interfaccia standardizzata per agenti RL, facilitando 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 tre 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 (cos’è in generale???) è la tecnica utilizzata per addestrare gli agenti DQN e PPO. L’agente interagisce con l’ambiente, riceve ricompense e penalità in base alle azioni effettuate .

La funzione di reward è progettata per incentivare comportamenti strategici: quindi abbiamo rewards per **Mossa valida, Vittoria**, **Sconfitta**, **Pareggio**, **Mossa illegale**, **Attacco riuscito (creazione di 3 pedine)**, **Difesa riuscita (blocco di 3 avversarie)**

Questa struttura consente all’agente di apprendere non solo dalle vittorie, ma anche da mosse promettenti e difensive, migliorando la qualità delle strategie apprese. [Però qua c’è da dire che abbiamo visto prima cosa succede senza Attacco riuscito (creazione di 3 pedine e Difesa riuscita (blocco di 3 avversarie): , poi cosa succendo inserendo queste rewards, se l’apprendimenti è migliore – questa cosa non so se va scritta in questo paragrafo o in successivi vedi tu dove va meglio]

Diciamo anche che usiamo stablebaseline3 come libreria per implementare dqn e ppo e perché (vedi tu bene i perché però diciamo che usato libreria per avere algoritmi già implementati dove cambiare solo configurazioni come ipermparamtri e struttura rete neurale] Inoltre forse da dire che si è scelto di usare rete neurlae perché avendo molte azioni in una griglia 6\*7 ad esempio usare qlesrning senza rete neurale portava ad avere tabelle troppo grandi di dimensioni e poco gestibili (o sbaglio?)

Gli algoritmi RL utilizzati si distinguono per il tipo di policy:

* **DQN (Off-Policy)**
* **PPO (On-Policy)**

Dimmi DQN e PPO cosa sono, cosa significa Off-Policy e On-Policy, se ci sono formule legate a questi algoritmi e cosa significano etc…

Dire anche che per quando rigurada stabelbaseline3 per rete neurale Di base la rete neurale usata da stable baseline3 è:2 hidden layer fully-connected64 neuroni per layerattivazione ReLU

output layer con dimensione = numero di azioni possibili (cioè i Q-values).

Schema completo:

Input (osservazioni dallo spazio Box)

↓

Linear (dim\_obs → 64) + ReLU

↓

Linear (64 → 64) + ReLU

↓

Linear (64 → n\_actions)

Quindi: MLP a 3 strati totali (2 nascosti da 64 + output). [è così davvero????] Poi magari accenna che è stata usata anche una configurazione differente che verrà nominata in paragrafi succesivi.

POI AGGIUNGI PARAGRAFO TRAINING

Qui mi vaia a dire come il training è avvenuto per DQN e PPO.

Per entrambi è stato unsato un curriculm learning con avversari di difficoltà crescente. In particolare prima con Random poi RuleBsedL1 e Poi rulebasedL2.

Inoltre per ogni avversario (per incentivare l’esplorazione ed evitare overfit su alcune colonne???) sono stati condotti training dove la prima mossa era sempre casuale e training dove era agente a decidere prima colonna dove mettersi.

In particolare sono stati fatti 100\_000 time\_steps (dici cosa sono) dqn vs Random con prima mossa casuale, 100\_000 time\_steps dqn vs Random non con mossa iniziale cadsuale, 120\_000 timesteps dqn vs RuleBasedL1 con con prima mossa casuale, 120\_000 time\_steps dqn vs RuleBasedL1 non con mossa iniziale cadsuale, 150\_000 timesteps dqn vs RuleBasedL2 con con prima mossa casuale, 150\_000 time\_steps dqn vs RuleBasedL2 non con mossa iniziale cadsuale. Stessa cosa per ppo.

Inoltre sono stati fatti piu training cambiando iperparatrri e ricompense e struttura rete neurale per vedere come cambiava apprendimento.

POI AGGIUNGI PARAGRAFO CON I RISULTATI

Qui dici che dopo training modello sono state fatte sia per dqn che ppo, 200 partite vs Random, 200 partite vs RulebasedL1 e 200 partite RulebasedL2. Alla fine è stato fatto giocare anche dqn vs PPO.

***Difese e Attacchi***

* Difesa
  + Se un agente ha una fila di 3 pedine, ad esempio O ha 3 pedine

Se X blocca le 3 pedine allora -> Difesa attuata, altrimenti mancata

* Attacco
  + Se agente ha una fila di 2 pedine, ad esempio X ha 2 pedine

Se X aggiunge la 3 allora -> Attacco riuscito, altrimenti no

* + Se agente ha una fila di 3 pedine, ad esempio X ha 3 pedine

Se X aggiunge la 4 allora -> Attacco riuscito (vince anche), altrimenti no [questa cosa non so in che paragrafo metterla] però credo qui perché il file get\_statiscics ha controlalto queste cose per i risultati oltre vittorie scofitte pareggi.

## Conclusioni

Scrivimi conclusioni e anche sviluppi futuri. Sviluppi futuri potrebbe essere testare altre architetture di rete? O usane CNN ? o fare tuning iperparamtri e rewards? Avere agenti ancora piu avanzati ? etc ?? dimmi tu anche aggiungere altre rewards intemedie etcc..

# Risultati – Agente DQN

L’agente DQN è stato addestrato utilizzando diverse configurazioni di iperparametri e funzioni di ricompensa, al fine di individuare la combinazione più efficace.  
Dopo il training descritto nella sezione precedente (100k+100k vs Random, 120k+120k vs RBL1, 150k+150k vs RBL2), l’agente è stato valutato su 200 partite contro avversari di difficoltà crescente.

**Configurazione 1 – Base**

**Iperparametri**:

learning\_rate=1e-3,

buffer\_size=50000,

batch\_size=64,

exploration\_initial\_eps=1.0,

exploration\_final\_eps=0.05,

exploration\_fraction=0.3,

gamma=0.99,

target\_update\_interval=500

**Rewards**:

"win": 1.0,

"lose": -1.0,

"draw": 0.5,

"valid\_move": 0.0,

"invalid": -2.0

**Risultati (200 partite)**:

* vs Random → **162-38-0**
* vs RuleBasedL1 → **151-49-0**
* vs RuleBasedL2 → **5-194-1**

📌 L’agente impara strategie solide contro avversari deboli (Random, RBL1), ma non riesce a generalizzare contro RBL2. Penalità forte per mosse illegali (-2) rende l’apprendimento prudente ma poco esplorativo.

**Configurazione 2 – Iperparametri modificati**

**Cambiamenti**:

* learning\_rate ridotto a **1e-4** → aggiornamenti più lenti ma più stabili.
* batch\_size aumentato a **128** → aggiornamento più consistente.
* exploration\_final\_eps ridotto a **0.01** e exploration\_fraction=0.5 → esplorazione più lunga.
* target\_update\_interval=1000 → aggiornamenti meno frequenti della rete target.

**Risultati (200 partite)**:

* vs Random → **166-34-0**
* vs RBL1 → **138-62-0**
* vs RBL2 → **56-144-0**

📌 Rispetto alla Configurazione 1:

* **Miglioramento contro RBL2** (da 5 vittorie a 56).
* **Peggioramento contro RBL1** (da 151 a 138 vittorie).
* **Stabilità maggiore** ma l’apprendimento è più lento.

**Configurazione 3 – Rewards intermedie (attacco/difesa)**

**Rewards**:

"win": 1.0,

"lose": -1.0,

"draw": 0.5,

"valid\_move": 0.0,

"invalid": -2.0,

"create\_three": 0.3,

"block\_three": 0.3

**Risultati (200 partite)**:

* vs Random → **186-14-0**
* vs RBL1 → **162-38-0**
* vs RBL2 → **54-146-0**

📌 L’aggiunta di ricompense intermedie incentiva schemi offensivi e difensivi → miglioramento netto contro Random e RBL1, ma non contro RBL2 (prestazioni simili alla Config.2).

**Configurazione 4 – Rewards intermedie + penalità ridotta**

**Rewards**:

"invalid": -1.0, # meno severa

"create\_three": 0.4,

"block\_three": 0.4

**Risultati (200 partite)**:

* vs Random → **170-30-0**
* vs RBL1 → **131-69-0**
* vs RBL2 → **91-109-0**

📌 Penalità meno pesante per mosse illegali → maggiore esplorazione.

* Miglioramento contro RBL2 (91 vittorie rispetto a 54).
* Leggero peggioramento contro Random e RBL1.
* Trade-off tra robustezza contro avversari forti e performance contro i più deboli.

**Configurazione 5 – Rete neurale più grande (128 neuroni)**

**Modifiche**:

* Architettura: due hidden layer da **128 neuroni** (anziché 64).
* Iperparametri adattati:

learning\_rate=5e-4,

buffer\_size=100000,

batch\_size=256,

exploration\_fraction=0.6,

exploration\_final\_eps=0.01,

target\_update\_interval=2000

**Risultati (200 partite)**:

* vs Random → **148-52-0**
* vs RBL1 → **120-80-0**
* vs RBL2 → **96-104-0**

📌 Con una rete più grande:

* peggioramento contro Random e RBL1 (probabile overfitting e apprendimento più complesso),
* **miglioramento contro RBL2** (96 vittorie, meglio di tutte le configurazioni precedenti).

**Confronto complessivo e migliore configurazione**

* **Config.1** → ottima contro avversari deboli, pessima contro RBL2.
* **Config.2** → più stabile, buone prestazioni generali, inizia a reggere contro RBL2.
* **Config.3** → forte contro Random/RBL1 grazie a ricompense intermedie, ma non migliora su RBL2.
* **Config.4** → buon compromesso, migliora su RBL2 a scapito delle prestazioni contro avversari facili.
* **Config.5** → migliore contro RBL2, ma meno efficace contro Random e RBL1.

📌 **Conclusione**:  
La configurazione **più bilanciata** è la **Config.4**, che con rewards intermedie e penalità meno severa ottiene buone performance contro tutti i livelli di avversario, riducendo il gap con RBL2.  
La **Config.5** è promettente contro avversari forti, ma meno stabile contro quelli semplici → utile se l’obiettivo è massimizzare la capacità strategica generale.

Vuoi che ti prepari delle **tabelle comparative + un grafico a barre** per visualizzare subito le differenze tra configurazioni (Random, RBL1, RBL2)?