# Sviluppo di un Agente Intelligente per il Gioco Connect 4

## Abstract

Il gioco Connect 4 rappresenta una sfida strategica in cui due giocatori competono per allineare quattro pedine consecutive 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 rule-based e casuali, ricevendo ricompense in base alle azioni effettuate e agli esiti delle partite.

Il progetto affronta diverse problematiche, tra cui la gestione delle mosse illegali, la difficoltà di apprendimento, la scelta delle rewards e iperparametri ottimali, il gosso numero di stati (? Visto che possono esserci veramentr ctoppe configurazioni tabellone in griglia 6\*7?).

Infine, sono state analizzate le prestazioni degli agenti RL in scenari deterministici e stocastici.

## Introduzione al Problema

Lo scopo di un agente intelligente nel contesto del gioco Connect 4 è quello di apprendere una strategia efficace per vincere contro altri avversari. Qui l’agente deve operare in uno spazio discreto e altamente strategico, dove ogni mossa può determinare l’esito della partita.

In questo progetto, l’agente è chiamato a prendere decisioni in tempo reale, valutando lo stato del tabellone e anticipando le mosse dell’avversario.

Per affrontare questa sfida, è stato progettato un ambiente di gioco che consente di variare il livello dell’avversario, simulando partite contro agenti casuali, rule-based di livello 1 e 2, e altri agenti RL, quindi una sorta di curriculm learning. L’agente apprende attraverso l’interazione con l’ambiente, ricevendo ricompense per mosse valide, non valide, vittorie, sconfitte, difese riuscite e attacchi strategici.

Il progetto si articola in diverse fasi: progettazione dell’ambiente, definizione delle ricompense, addestramento degli agenti, valutazione delle prestazioni e confronto tra algoritmi. In particolare, sono stati analizzati i comportamenti emergenti degli agenti DQN e PPO, evidenziando le differenze tra approcci on-policy e off-policy.

La repository con il codice prodotto nell’ambito del progetto è reperibile al seguente link: .

## Metodologia

### Creazione dell’Ambiente di Gioco

L’ambiente virtuale per il seguente progetto è stato sviluppato utilizzando la libreria **Gymnasium (dimmi cos’è e a che serve e perché la usiamo anche per compatibilità con stabelbaseline3 e altre libreirie etc..)**, e che viene implementato in file **Connect4Env.py**. Questo ambiente simula fedelmente le regole del gioco Connect 4, con una griglia 6x7. Dici che ci sono queste cose def \_\_init\_\_(*self*, *opponent*=None, *render\_mode*=None, *first\_player*=1):

        super().\_\_init\_\_()

*self*.opponent = *opponent*

*self*.first\_player = *first\_player*

*self*.next\_player\_to\_play = *first\_player*

*self*.render\_mode = *render\_mode*

        # Azioni = colonne disponibili

*self*.action\_space = spaces.Discrete(COLUMNS\_COUNT)

        # Osservazioni = griglia 6x7

*self*.observation\_space = spaces.Box(

*low*=-1, *high*=1, *shape*=(ROWS\_COUNT, COLUMNS\_COUNT), *dtype*=np.float32

        )

*self*.reset()

    # ----------------------

    # Utility funzioni base

    # ----------------------

    def reset(*self*, *seed*=None, *options*=None):

        super().reset(*seed*=*seed*)

*self*.board = np.zeros((ROWS\_COUNT, COLUMNS\_COUNT), *dtype*=np.float32)

*self*.next\_player\_to\_play = *self*.first\_player

*self*.last\_move\_row = None

*self*.last\_move\_col = None

*self*.winner = None

        return *self*.board.copy(), {"action\_mask": *self*.get\_action\_mask()}

e parla anche delle altre funzioni del file connect4env che hai di cosa fanno ( c0me step, is\_valid\_ etcc…)

L’ambiente è compatibile con agenti esterni ed è stato progettato per supportare sia il rendering testuale (console) che grafico (GUI), tramite il modulo gui\_rend.

Nel contesto grafico, l’interfaccia visualizza lo stato del tabellone in tempo reale, con pedine rosse per il giocatore X e gialle per il giocatore O. Le colonne sono numerate e la griglia viene aggiornata ad ogni mossa. Il rendering è utile per il debugging e per la valutazione visiva delle strategie apprese dagli agenti.

L’ambiente gestisce anche le mosse illegali, penalizzando l’agente che tenta di inserire una pedina in una colonna piena. In fase di training, il turno non viene passato e l’agente ha la possibilità di correggere il proprio errore, ricevendo una penalità.

\*\*Metti pragrafo gui che usa pygame per far vedere tabellone (ci sono i files gui\_rend.py con funzioni di visualizzazione e gui\_config che contienen dimensioni, colori font)

### Agenti

Il progetto include diversi tipi di agenti, ciascuno con caratteristiche e comportamenti distinti:

* **RandomAgent**: seleziona mosse casuali tra quelle valide.
* **RuleBasedL1Agent**: segue regole semplici, come sfruttare mosse vincenti proprie.
* **RuleBasedL2Agent**: implementa strategie difensive avanzate, come sfruttare mosse vincenti il blocco di mosse vincenti dell’avversario.
* **DQNAgent**: agente RL basato su Deep Q-Network.
* **PPOAgent**: agente RL basato su Proximal Policy Optimization.
* **HumanAgent**: consente l’interazione manuale con l’ambiente.

Ogni agente è collegato all’ambiente tramite il parametro env, e può accedere alla griglia, alle azioni valide e alle informazioni sullo stato corrente.

Gli agenti RL utilizzano una rete neurale per stimare la qualità delle azioni, mentre gli agenti rule-based seguono logiche predefinite.

### 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 in base alle azioni effettuate e aggiorna la propria policy per massimizzare la ricompensa cumulativa.

La funzione di reward è progettata per incentivare comportamenti strategici:

* **Mossa valida**: +0.1
* **Vittoria**: +1.0
* **Sconfitta**: -1.0
* **Pareggio**: 0.0
* **Mossa illegale**: -1.0
* **Attacco riuscito (creazione di 3 pedine)**: +0.3
* **Difesa riuscita (blocco di 3 avversarie)**: +0.3

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): +0.3 e Difesa riuscita (blocco di 3 avversarie): +0.3, poi cosa succendo inserendo queste rewards, se l’apprendimenti è migliore ]

\*\* Diciamo anche che usiamo stablebaseline3 e perché.

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

* **DQN (Off-Policy)**: apprende da esperienze passate, anche se non generate dalla policy corrente.
* **PPO (On-Policy)**: aggiorna la policy direttamente dalle esperienze recenti.

Il confronto tra DQN e PPO è stato effettuato per valutare la stabilità, la capacità di generalizzazione e la reattività degli agenti in scenari diversi.

\*\*Aggiungi paragrafo training e i vari test con iperparametri diversi e il fatto sempre di stabelbasleine3 come lo fa. E come viene effettuato etcc.. anche il fatto della casualità introdotto tra chi inizia etc…

## Metodologia

**2.1. Creazione dell’Ambiente di Gioco**

L’ambiente virtuale utilizzato per questo progetto è stato sviluppato in Python tramite la libreria **Gymnasium**, uno standard per la creazione di ambienti compatibili con algoritmi di reinforcement learning. Gymnasium consente di definire ambienti con interfacce uniformi, facilitando l’integrazione con librerie come **Stable-Baselines3**, che forniscono implementazioni pronte di algoritmi RL come DQN e PPO. L’uso di Gymnasium garantisce modularità, compatibilità e scalabilità del progetto.

L’ambiente è implementato nel file Connect4Env.py, e simula fedelmente le regole del gioco Connect 4 su una griglia 6x7. La classe Connect4Env eredita da gym.Env e definisce:

def \_\_init\_\_(self, opponent=None, render\_mode=None, first\_player=1):

super().\_\_init\_\_()

self.opponent = opponent

self.first\_player = first\_player

self.next\_player\_to\_play = first\_player

self.render\_mode = render\_mode

self.action\_space = spaces.Discrete(COLUMNS\_COUNT)

self.observation\_space = spaces.Box(

low=-1, high=1, shape=(ROWS\_COUNT, COLUMNS\_COUNT), dtype=np.float32

)

self.reset()

* action\_space: definisce le azioni disponibili (colonne da 0 a 6).
* observation\_space: rappresenta lo stato del tabellone come matrice 6x7 con valori {-1, 0, 1}.

La funzione reset() inizializza la board e imposta il giocatore iniziale. Altre funzioni chiave includono:

* step(action): esegue una mossa, aggiorna lo stato, calcola il reward e gestisce il turno.
* is\_action\_valid(action): verifica se una mossa è legale.
* get\_valid\_actions(): restituisce le colonne disponibili.
* check\_win\_around\_last\_move(): controlla se l’ultima mossa ha causato una vittoria.
* get\_action\_mask(): restituisce una maschera binaria per le azioni valide.

L’ambiente è compatibile con agenti esterni e supporta il rendering testuale (console) e grafico (GUI), tramite il modulo gui\_rend.

**2.1.1. Interfaccia Grafica**

La visualizzazione grafica è realizzata con la libreria **Pygame**, attraverso i file gui\_rend.py e gui\_config.py. Questi moduli definiscono:

* Dimensioni della finestra.
* Colori delle pedine (rosso per X, giallo per O).
* Font e layout del tabellone.

Il tabellone viene aggiornato in tempo reale ad ogni mossa, mostrando lo stato corrente e facilitando il debugging e l’analisi visiva delle strategie apprese.

**2.2. Agenti**

Il progetto include diversi tipi di agenti, ciascuno con comportamenti distinti:

* **RandomAgent**: seleziona mosse casuali tra quelle valide.
* **RuleBasedL1Agent**: sfrutta mosse vincenti proprie.
* **RuleBasedL2Agent**: blocca mosse vincenti dell’avversario e attua difese avanzate.
* **DQNAgent**: agente RL basato su Deep Q-Network.
* **PPOAgent**: agente RL basato su Proximal Policy Optimization.
* **HumanAgent**: consente l’interazione manuale con l’ambiente.

Ogni agente è collegato all’ambiente tramite il parametro env, e può accedere alla griglia, alle azioni valide e alle informazioni sullo stato corrente. Gli agenti RL utilizzano una rete neurale per stimare la qualità delle azioni, mentre gli agenti rule-based seguono logiche predefinite.

**2.3. Reinforcement Learning**

Il **reinforcement learning** è una tecnica di apprendimento automatico in cui un agente interagisce con un ambiente, riceve ricompense in base alle azioni effettuate e aggiorna la propria strategia per massimizzare la ricompensa cumulativa. A differenza dell’apprendimento supervisionato, non riceve etichette corrette, ma apprende tramite esplorazione e feedback.

Nel progetto, gli agenti DQN e PPO sono addestrati tramite la libreria **Stable-Baselines3**, che offre implementazioni robuste e ottimizzate degli algoritmi RL. L’ambiente è compatibile con questa libreria grazie all’interfaccia Gymnasium.

Di base la rete neurale usata da stable baseline3 è:

2 hidden layer fully-connected

64 neuroni per layer

attivazione 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).

La funzione di reward è progettata per incentivare comportamenti strategici:

| **Evento** | **Reward** |
| --- | --- |
| Mossa valida | +0.1 |
| Vittoria | +1.0 |
| Sconfitta | -1.0 |
| Pareggio | 0.0 |
| Mossa illegale | -1.0 |
| Attacco riuscito (3 pedine) | +0.3 |
| Difesa riuscita (blocca 3) | +0.3 |

In fase iniziale, l’agente riceve solo reward per vittorie e mosse valide. Tuttavia, si è osservato che contro avversari forti come RuleBasedL2, l’agente perde frequentemente e riceve quasi sempre reward negativi. Per migliorare l’apprendimento, sono state introdotte ricompense intermedie per attacchi e difese riuscite. Questo ha portato a un miglioramento tangibile nella qualità delle strategie apprese.

**2.4. Tipologie di Policy**

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

* **DQN (Off-Policy)**: apprende da esperienze passate, anche se non generate dalla policy corrente.
* **PPO (On-Policy)**: aggiorna la policy direttamente dalle esperienze recenti.

Il confronto tra DQN e PPO è stato effettuato per valutare:

* Stabilità dell’apprendimento.
* Capacità di generalizzazione.
* Reattività in scenari dinamici.

**2.5. Addestramento e Test**

L’addestramento è stato effettuato tramite model.learn() di Stable-Baselines3, con logging su TensorBoard per monitorare l’evoluzione delle azioni e delle ricompense. Sono stati testati diversi iperparametri:

* Learning rate.
* Batch size.
* Gamma (discount factor).
* Strategia di esplorazione.

È stata introdotta **casualità nell’apertura** delle partite, alternando chi inizia o forzando l’avversario a fare la prima mossa. Questo ha permesso di raccogliere esperienze più varie e migliorare la robustezza dell’agente.

## Risultati

**3.1. Metriche di Valutazione**

Per valutare le prestazioni degli agenti addestrati con gli algoritmi di reinforcement learning **DQN** e **PPO**, sono state adottate diverse metriche:

* **Tasso di vittoria** contro avversari di diversa difficoltà (Random, RuleBasedL1, RuleBasedL2).
* **Distribuzione delle azioni** selezionate durante il training, monitorata tramite TensorBoard.
* **Media dei reward** ottenuti per episodio.
* **Capacità di attacco e difesa**, valutata empiricamente osservando la simulazione.

L’osservazione diretta ha permesso di identificare comportamenti strategici emergenti. Fattori positivi includono:

* La tendenza a giocare al centro del tabellone (mossa strategica).
* La capacità di bloccare mosse vincenti dell’avversario.
* L’esecuzione di attacchi in sequenza (creazione di file da 2 a 3 pedine).

Fattori negativi osservati:

* Ripetizione di mosse in colonne già piene (mosse illegali).
* Mancanza di reazione contro avversari rule-based avanzati.
* Comportamenti passivi contro agenti che bloccano sistematicamente le mosse vincenti.

In particolare, contro **RuleBasedL2**, l’agente RL perde frequentemente, ricevendo reward negativi costanti. Questo ha evidenziato la necessità di introdurre ricompense intermedie per incentivare l’apprendimento anche in partite non vincenti.

**3.2. Ottimizzazione dei Parametri**

L’addestramento ha richiesto una fase esplorativa per identificare la combinazione ottimale di iperparametri. I parametri principali considerati sono:

* **Ricompense e penalità**: definite in env\_config.py, calibrate per premiare vittorie e mosse strategiche, penalizzare mosse illegali e incentivare attacchi/difese.
* **Numero di episodi**: variato tra 500 e 1500 per testare la convergenza.
* **Numero massimo di iterazioni per episodio**: fissato a 42 (numero massimo di mosse in Connect 4).
* **Probabilità di esplorazione**: inizializzata a 1.0, decrescente fino a 0.05.
* **Tasso di decrescenza dell’esplorazione**: regolato per garantire una fase di esplorazione sufficiente.
* **Learning rate**: testato tra 0.01 e 0.0001.
* **Fattore di sconto γ**: mantenuto costante a 0.99.

L’ottimizzazione è stata guidata dalle metriche di valutazione e dall’osservazione diretta del comportamento dell’agente. In particolare, si è notato che:

* Un **learning rate troppo alto** (es. 0.01) porta a instabilità e comportamenti incoerenti.
* Un **learning rate ottimale** è risultato essere **0.001**, garantendo stabilità e miglioramento progressivo.
* Un **epsilon finale troppo basso** (es. 0.001) riduce l’esplorazione e può causare overfitting su strategie subottimali.

**3.3. Configurazione Ottimale**

La configurazione che ha prodotto i risultati migliori è la seguente:

* **Learning rate**: 0.001
* **Gamma (γ)**: 0.99
* **Episodi**: 1100
* **Exploration fraction**: 0.3
* **Exploration final epsilon**: 0.05
* **Batch size**: 64
* **Buffer size (DQN)**: 50.000
* **Policy**: MlpPolicy

Durante il training, è stato introdotto un meccanismo di **alternanza del primo giocatore**, per evitare che l’agente imparasse solo da una prospettiva. Inoltre, è stato testato un setup in cui l’avversario (es. RandomAgent) effettua la prima mossa, simulando scenari più realistici e variabili.

L’introduzione delle ricompense intermedie per **attacco riuscito** e **difesa riuscita** ha portato a un netto miglioramento nella qualità delle strategie apprese. L’agente ha iniziato a riconoscere pattern di gioco e a reagire in modo più efficace alle minacce avversarie.

**3.4. Discussione dei Risultati del DQN**

Per quanto riguarda i risultati ottenuti con l’algoritmo **DQN**, durante le sessioni di addestramento l’agente ha mostrato una progressiva capacità di apprendere strategie efficaci. In particolare, con la configurazione ottimale dei parametri descritta in precedenza, l’agente ha raggiunto picchi di performance significativi, come la selezione strategica del centro del tabellone e la creazione di file di tre pedine.

Tuttavia, l’agente non risulta sempre performante nel riconoscere e bloccare minacce avversarie. In diversi episodi, si è osservato che l’agente tende a ripetere mosse in colonne già piene o a ignorare opportunità di difesa, soprattutto contro avversari come **RuleBasedL2**, che bloccano sistematicamente le mosse vincenti. In assenza di ricompense intermedie, il DQN riceve spesso reward negativi, compromettendo l’apprendimento.

L’introduzione di ricompense per **attacco riuscito** e **difesa riuscita** ha migliorato sensibilmente il comportamento dell’agente, incentivando la costruzione di file di tre pedine e il blocco di minacce avversarie. Tuttavia, la strategia appresa resta fragile in scenari complessi, e l’agente mostra difficoltà nel generalizzare contro avversari non visti.

**3.5. Confronto con i Risultati del PPO**

Dopo l’analisi dei risultati ottenuti con DQN, è stato implementato e testato l’agente basato su **PPO**, utilizzando gli stessi parametri per garantire un confronto equo. PPO, essendo un algoritmo on-policy, ha mostrato una maggiore regolarità nell’apprendimento, con una curva di reward più stabile e meno oscillazioni tra gli episodi.

Nonostante le prestazioni medie siano leggermente inferiori rispetto a DQN, PPO ha dimostrato una maggiore capacità di adattamento, soprattutto nelle fasi iniziali del training. L’agente PPO tende a esplorare più a lungo e a evitare comportamenti ripetitivi, risultando più robusto contro avversari casuali e rule-based di livello 1.

Anche in questo caso, l’agente presenta difficoltà contro RuleBasedL2, e le problematiche legate alla mancanza di esperienze positive si ripresentano. L’introduzione di ricompense intermedie ha portato benefici anche per PPO, ma non ha completamente risolto le limitazioni strutturali dell’algoritmo in ambienti altamente strategici come Connect 4.

## Conclusioni

In conclusione, il progetto ha dimostrato che è possibile addestrare agenti intelligenti in grado di apprendere strategie di gioco efficaci per Connect 4 utilizzando algoritmi di reinforcement learning come DQN e PPO. Tuttavia, entrambi gli approcci presentano limiti nella gestione di scenari complessi, in particolare contro avversari che bloccano sistematicamente le mosse vincenti.

Una delle principali difficoltà riscontrate è la mancanza di esplorazione completa dello spazio degli stati, soprattutto in ambienti dinamici dove le configurazioni cambiano rapidamente. Questo porta l’agente a non riconoscere situazioni critiche e a non reagire in modo ottimale.

Una possibile soluzione consiste nell’ampliare il campo di osservazione dell’agente, permettendogli di prevedere minacce e opportunità con maggiore anticipo. Tuttavia, questo comporta un aumento significativo della dimensione dello stato, con conseguente difficoltà di esplorazione e rischio di overfitting.

Per il futuro, si propone:

* L’adozione di **architetture più complesse**, come reti neurali convoluzionali (CNN), per catturare pattern spaziali.
* L’implementazione di tecniche di **Curriculum Learning**, per far crescere gradualmente la difficoltà dell’avversario.
* La sperimentazione di algoritmi avanzati come **AlphaZero-style MCTS** o **Deep Q-Learning con attenzione**, per migliorare la capacità strategica dell’agente.

Il progetto ha posto solide basi per lo sviluppo di agenti intelligenti in ambienti strategici, e apre a numerose possibilità di estensione e miglioramento.