Sviluppo di un Agente Intelligente per il Gioco Ping-pong

Salvatore Sirica

Università degli Studi di Salerno

December 23, 2024

Contents

1 Introduzione

Il Ping-pong è un gioco classico che combina riflessi rapidi, controllo preciso e strategie dinamiche. Nel contesto dell'intelligenza artificiale, il gioco rappresenta un interessante problema di apprendimento per rinforzo (*Reinforcement Learning, RL*) in cui un agente deve apprendere a controllare la racchetta per colpire la palla in modo ottimale e mantenere il gioco il più a lungo possibile.

Questo documento presenta un approccio innovativo per sviluppare un agente intelligente in grado di giocare a Ping-pong utilizzando gli algoritmi di RL **SARSA** e **Q-Learning**. L'apprendimento per rinforzo consente all'agente di apprendere strategie ottimali attraverso l'interazione con l'ambiente, ricevendo ricompense per azioni efficaci e penalità per errori.

Nel dettaglio:

- **Q-Learning**: Un algoritmo *off-policy*, in cui l'agente apprende il valore massimo futuro indipendentemente dalle azioni attuali. Questo approccio fornisce flessibilità all'agente nell'esplorazione di nuove strategie.

- **SARSA**: Un algoritmo *on-policy*, che aggiorna la strategia dell'agente in modo coerente con le azioni effettivamente eseguite, risultando spesso più cauto rispetto a Q-Learning.

L'ambiente di gioco è stato creato utilizzando **OpenAI Gym**, che garantisce modularità e scalabilità, mentre la visualizzazione è gestita tramite **Pygame**. Questi strumenti offrono una solida base per implementare un agente RL in un contesto personalizzato come il gioco Ping-pong.

I risultati ottenuti vengono valutati attraverso il confronto delle performance di SARSA e Q-Learning, con particolare attenzione alla velocità di convergenza e all'ottimizzazione delle ricompense cumulative. Dalle analisi emerge che Q-Learning, grazie alla sua natura *off-policy*, mostra una convergenza più rapida e prestazioni superiori rispetto a SARSA.

Questo progetto dimostra come l'apprendimento per rinforzo possa essere applicato con successo a un ambiente di gioco come il Ping-pong, offrendo spunti interessanti per ulteriori sviluppi e ottimizzazioni future.

2 Obiettivi del Progetto

Durante lo sviluppo del progetto sono stati delineati i seguenti obiettivi principali:

1. **Controllo Ottimale delle Racchette**: Creare un ambiente in grado di simulare il gioco Ping-pong, dove due agenti controllano le rispettive racchette per colpire la palla e mantenere il gioco attivo il più a lungo possibile.

2. **Ottimizzazione dei Movimenti**: Addestrare gli agenti a eseguire movimenti ottimali, tra cui spostamenti verso l'alto, verso il basso, o rimanere fermi, per anticipare la traiettoria della palla e massimizzare il punteggio.

3. **Definizione degli Stati e delle Azioni**: Mappare in modo accurato lo stato dell'ambiente (posizione della palla, velocità e posizioni delle racchette) e le possibili azioni per ciascun agente.

4. **Gestione delle Ricompense e Penalità**: Implementare un sistema di ricompense e penalità per incentivare comportamenti desiderati, come colpire la palla, e penalizzare errori, come far uscire la palla dal campo.

2.1 Controllo Ottimale delle Racchette

L'ambiente, denominato `MultiplayerPongEnv`, è stato sviluppato utilizzando **OpenAI Gym** per garantire conformità agli standard di RL e scalabilità. Le racchette sono controllate attraverso un sistema discreto di azioni che consente agli agenti di muoversi verticalmente nel campo:

```
# Azioni disponibili per le racchette
self.action_list = [0, 0.04, -0.04]  # Nessun movimento, Su, Giù
```

L'aggiornamento delle posizioni avviene nel metodo `step`, in cui vengono processate le azioni degli agenti e aggiornate le posizioni delle racchette e della palla.

```
# Aggiornamento delle racchette
self.left_paddle_y += self.action_list[left_action]
self.right_paddle_y += self.action_list[right_action]
# Aggiornamento della pallina
self._update_ball_position()
```

2.2 Ottimizzazione dei Movimenti

Gli algoritmi **SARSA** e **Q-Learning** vengono utilizzati per ottimizzare i movimenti delle racchette. La Q-table viene aggiornata dinamicamente per ciascun agente in base alle transizioni di stato, azione e ricompensa. Dunque la definizione delle ricompense risulta un punto critico che caratterizza il successo dell'apprendimento dei diversi agenti

2.3 Definizione degli Stati e delle Azioni

Lo stato dell'ambiente è rappresentato da:

- Posizione della palla (x, y).
- Velocità della palla (v_x, v_y).
- Posizione delle racchette (sinistra e destra).

Questo stato viene discretizzato utilizzando una funzione specifica:

```
# Discretizzazione dello stato
continuous_state = self._get_continuous_state()
return self.discretizer.discretize(continuous_state)
```

Le azioni possibili sono tre:

- Movimento verso l'alto.
- Movimento verso il basso.
- Nessun movimento.

2.4 Gestione delle Ricompense e Penalità

Il sistema di ricompense guida l'apprendimento degli agenti:

- **Ricompensa Positiva**: +1 per un colpo riuscito.
- **Penalità**: -1 quando la palla supera il confine del campo, non colpita dalla racchetta.

Questo sistema è integrato nel metodo `step` e varia in base al comportamento dell'agente:

```
if self.ball_x <= 0:  # Collisione lato sinistro
    if self.left_paddle_y <= self.ball_y <= self.left_paddle_y + self.paddle_height:
        left_reward = 1  # Ricompensa positiva
    else:
        left_reward = -1  # Penalità
```

3 Metodologie ed Implementazione

3.1 Creazione dell'Ambiente di Ping-pong

L'ambiente di gioco, denominato `MultiplayerPongEnv`, è stato sviluppato utilizzando la libreria **OpenAI Gym**, garantendo modularità e compatibilità con diversi algoritmi di apprendimento per rinforzo.

L'ambiente simula un campo da Ping-pong dove due agenti controllano rispettivamente una racchetta, cercando di mantenere la palla in gioco. La dinamica del gioco è rappresentata attraverso stati e azioni definiti in modo accurato, supportati dalla discretizzazione.

Caratteristiche principali dell'ambiente:

- **Dimensioni del campo**: Altezza e larghezza normalizzate a 1.0.
- **Movimento delle racchette**: Tre azioni disponibili (*su, giù, fermo*).
- **Discretizzazione dello stato**: Lo stato continuo, composto da posizione e velocità della palla e posizione delle racchette, è discretizzato in 12 bin per dimensione, come mostrato di seguito:

```
bins_per_dimension = [12, 12, 2, 2, 12, 12]
self.discretizer = Discretizer(bins_per_dimension=bins_per_dimension)
```

Codice del metodo `step`: Questo metodo aggiorna lo stato dell'ambiente in base alle azioni dei due agenti e calcola le ricompense:

```
def step(self, actions):
    left_action, right_action = actions
    self.left_paddle_y += self.action_list[left_action]
```

```
self.right_paddle_y += self.action_list[right_action]
# Aggiorna posizione della palla e gestisce collisioni
```

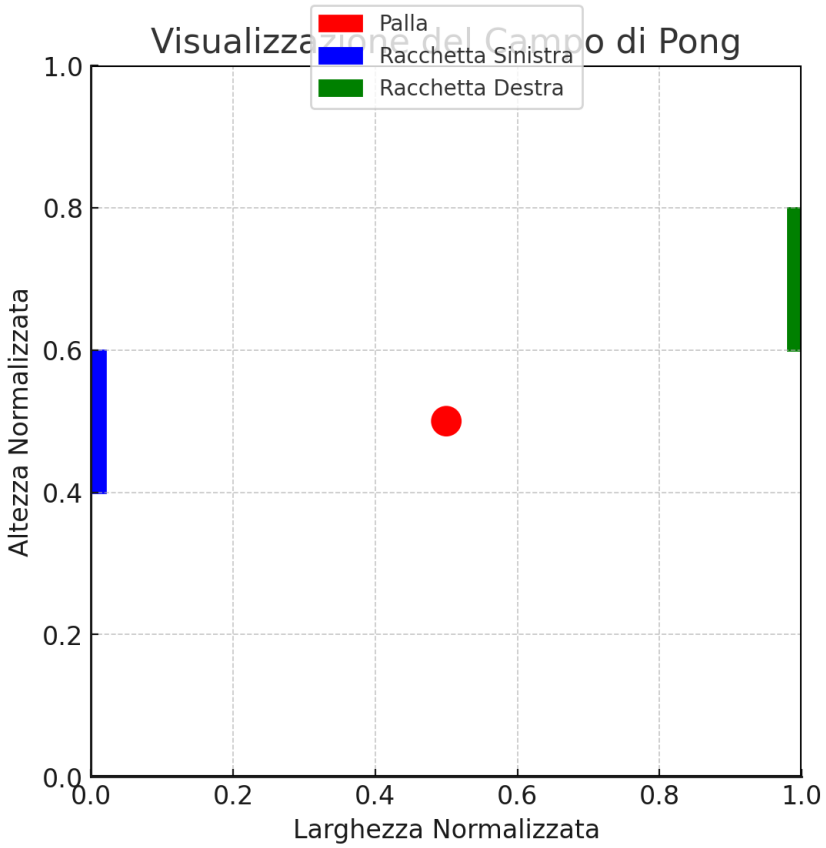

Figure 1: Visualizzazione del Campo di Pong con posizione della palla e delle racchette.

3.2 Implementazione degli Algoritmi di Apprendimento per Rinforzo

Sono stati implementati due algoritmi principali:

1. **Q-Learning**: Algoritmo *off-policy* che aggiorna la Q-table in base alla ricompensa futura massima.
2. **SARSA**: Algoritmo *on-policy* che aggiorna la Q-table in base alla sequenza di azioni effettivamente eseguite.

Entrambi i modelli derivano dalla classe `BaseAgent`, che fornisce funzionalità comuni come la gestione della Q-table, la selezione delle azioni (*epsilon-greedy policy*) e l'aggiornamento dei parametri.

Q-Learning Agent: Il metodo `observe` utilizza l'equazione di Bellman per aggiornare i valori Q:

```
old_q = self.q_table[(state, action)]
next_max = max([self.q_table[(next_state, a)] for a in self.actions])
new_q = old_q + adjusted_alpha * (reward + self.gamma * next_max - old_q)
self.q_table[(state, action)] = new_q
```

SARSA Agent: Il metodo `observe` segue un approccio simile, ma il valore futuro è calcolato sulla base dell'azione successiva selezionata:

```
next_action = self.get_action(next_state)
next_q = self.q_table[(next_state, next_action)]
new_q = old_q + adjusted_alpha * (reward + self.gamma * next_q - old_q)
```

3.3 Definizione degli Stati e delle Azioni

Gli stati includono:

- Posizione della palla (x, y).
- Velocità della palla (v_x, v_y).
- Posizione delle racchette (sinistra e destra).

Le azioni disponibili per le racchette sono:

- Movimento verso l'alto.
- Movimento verso il basso.
- Nessun movimento.

Gli stati continui sono discretizzati per adattarsi alla Q-table:

```
continuous_state = self._get_continuous_state()
return self.discretizer.discretize(continuous_state)
```

3.4 Addestramento Iterativo

Durante l'addestramento, l'agente esplora lo spazio di stato-azione seguendo una politica *epsilon-greedy* che bilancia esplorazione e sfruttamento. La Q-table viene aggiornata iterativamente per migliorare la politica dell'agente.

Strategia *epsilon-greedy*:

```
eps_threshold = self.epsilon_end + (self.epsilon_start - self.epsilon_end)
                * math.exp(-1.0 * self.steps_done / self.epsilon_decay)
if random.uniform(0, 1) < eps_threshold:
    return random.choice(self.actions)  # Esplora
else:
    return self.get_best_action(state)  # Sfrutta
```

3.5 Valutazione delle Prestazioni

Le prestazioni sono state valutate considerando la capacità degli agenti di mantenere la palla in gioco e massimizzare il punteggio cumulativo. I risultati sono stati visualizzati tramite grafici prodotti con la libreria `matplotlib`, evidenziando la velocità di convergenza e la stabilità della politica appresa.

4 Modelli

L'agente è stato addestrato utilizzando algoritmi di *apprendimento per rinforzo* (RL), sfruttando un meccanismo di ricompense per apprendere strategie ottimali in un contesto competitivo di gioco Ping-pong. In particolare, sono stati implementati e valutati due algoritmi principali:

- **Q-Learning** (off-policy)
- **SARSA** (on-policy)

Le prestazioni dei due modelli sono state analizzate accuratamente per determinare quale sia più efficace nel contesto applicativo.

4.1 Q-Learning

Il **Q-Learning** è un algoritmo di apprendimento per rinforzo *off-policy*, particolarmente adatto per ambienti discreti come il nostro. Esso sfrutta la funzione $Q(s, a)$ per stimare il valore di ogni coppia stato-azione, con l'obiettivo di massimizzare la ricompensa cumulativa nel lungo termine.

La funzione Q viene aggiornata iterativamente utilizzando l'equazione di Bellman:

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

dove:

- s e s' sono rispettivamente lo stato corrente e quello successivo.
- a è l'azione corrente, mentre a' è la migliore azione successiva.
- r è la ricompensa ottenuta.
- α è il tasso di apprendimento.
- γ è il fattore di sconto.

Durante l'addestramento, l'agente esplora l'ambiente bilanciando esplorazione e sfruttamento grazie alla politica *epsilon-greedy*. Questo approccio permette all'agente di provare nuove strategie pur privilegiando le azioni più promettenti.

Implementazione dell'osservazione:

```
old_q = self.q_table[(state, action)]
next_max = max([self.q_table[(next_state, a)] for a in self.actions])
new_q = old_q + adjusted_alpha * (reward + self.gamma * next_max - old_q)
self.q_table[(state, action)] = new_q
```

4.2 SARSA

SARSA (State-Action-Reward-State-Action) è un algoritmo di RL *on-policy* che aggiorna la funzione $Q(s, a)$ in base all'azione realmente eseguita nel nuovo stato. L'equazione di aggiornamento è:

$$Q(s, a) \leftarrow Q(s, a) + \alpha \left[r + \gamma Q(s', a') - Q(s, a) \right]$$

La differenza principale rispetto a Q-Learning è che SARSA tiene conto della politica corrente per aggiornare il valore Q.

Implementazione dell'osservazione:

```
next_action = self.get_action(next_state)
next_q = self.q_table[(next_state, next_action)]
new_q = old_q + adjusted_alpha * (reward + self.gamma * next_q - old_q)
self.q_table[(state, action)] = new_q
```

4.3 Confronto tra Q-Learning e SARSA

Q-Learning si è dimostrato più efficace rispetto a SARSA grazie alla sua natura *off-policy*, che consente all'agente di esplorare strategie alternative indipendentemente dalla politica corrente. In un ambiente competitivo come Ping-pong, caratterizzato da dinamiche rapide e una forte interazione tra gli agenti, la capacità di Q-Learning di valutare l'azione ottimale ha portato a prestazioni superiori.

In contrasto, SARSA, a causa della sua natura *on-policy*, è risultato più cauto nelle decisioni, aggiornando la Q-table sulla base delle azioni effettivamente eseguite. Questa caratteristica ha limitato la capacità dell'agente di esplorare strategie più rischiose ma potenzialmente più efficaci.

5 Risultati e Motivazioni

5.1 Miglior Punteggio nel Gioco

L'agente addestrato con l'algoritmo **Q-Learning** ha dimostrato eccellenti capacità nel mantenere la palla in gioco, ottimizzando il controllo della racchetta. Attraverso un processo di addestramento iterativo, l'agente ha imparato a reagire efficacemente alle dinamiche dell'ambiente, migliorando gradualmente le proprie prestazioni.

Ricompense cumulative: Durante l'addestramento, si osserva un costante miglioramento delle ricompense ottenute, evidenziando l'efficacia dell'ottimizzazione.

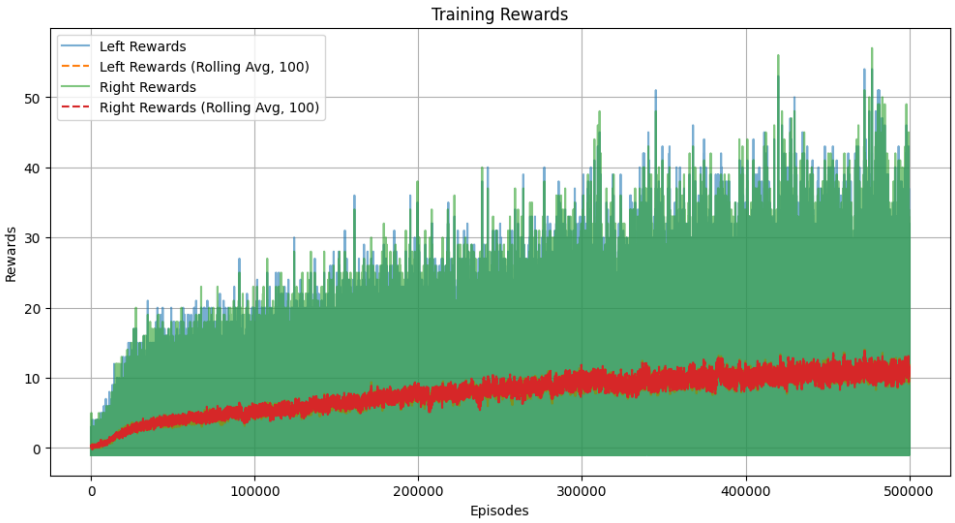

Figure 2: Ricompense cumulative durante l'addestramento con Q-Learning (500,000 episodi).

5.2 Confronto tra SARSA e Q-Learning

Per confrontare le prestazioni dei due algoritmi, sono stati generati grafici che mostrano le **ricompense cumulative** ottenute durante l'addestramento. In particolare, i risultati evidenziano come il Q-Learning superi costantemente SARSA in termini di velocità di convergenza e punteggio massimo raggiunto.

Grafico di confronto diretto tra Q-Learning e SARSA:

Analisi dei risultati:

- **Velocità di Convergenza**: I grafici dimostrano che Q-Learning converge più rapidamente, raggiungendo valori di ricompensa più elevati rispetto a SARSA.
- **Stabilità**: SARSA presenta una crescita più graduale e stabile, ma raggiunge ricompense inferiori a quelle di Q-Learning.
- **Esplorazione vs Sfruttamento**: La natura *off-policy* di Q-Learning permette all'agente di esplorare strategie più rischiose ma efficaci, mentre SARSA, a causa della sua natura *on-policy*, tende a essere più conservativo.

Questi risultati dimostrano come Q-Learning sia più adatto a un ambiente competitivo come il Ping-pong, dove la capacità di esplorare strategie ottimali è cruciale per massimizzare le prestazioni.

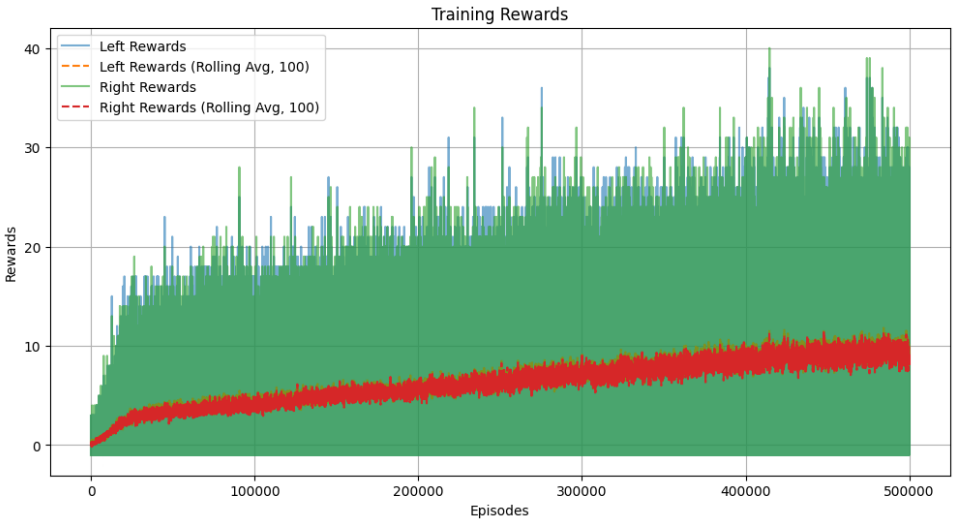

Figure 3: Andamento delle ricompense con SARSA (500,000 episodi).

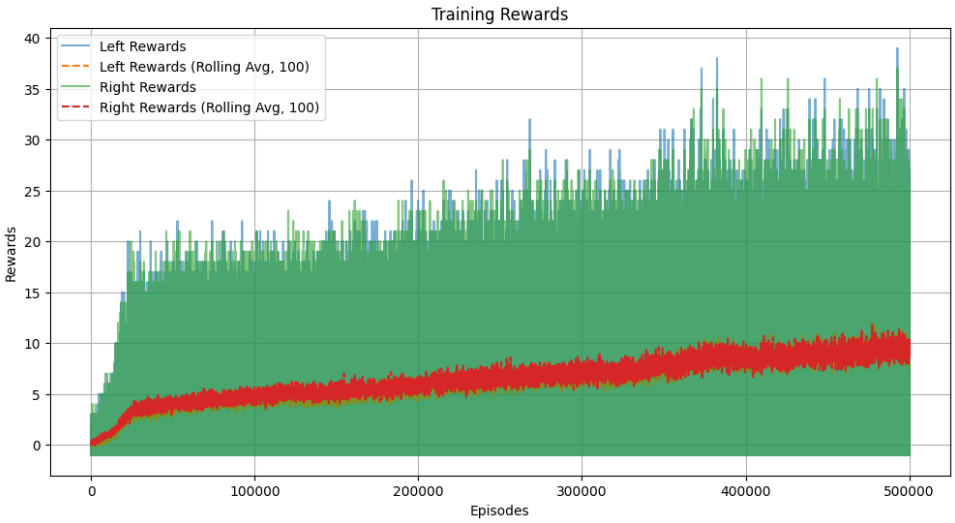

Figure 4: Confronto tra Q-Learning e SARSA durante l'addestramento (500,000 episodi).

5.3 Sfide Affrontate Durante l'Implementazione

Durante l'implementazione, sono emerse alcune sfide principali:

- **Gestione della Q-Table:** Inizialmente, l'aggiornamento della Q-table non era sincronizzato con l'azione successiva degli agenti, causando instabilità. Questo problema è stato risolto garantendo un aggiornamento accurato della Q-table a ogni passo.

- **Bilanciamento tra esplorazione e sfruttamento:** La politica *epsilon-greedy* è stata fondamentale per permettere all'agente di esplorare l'ambiente in modo

efficace, riducendo gradualmente l'epsilon per concentrarsi sulle azioni ottimali.

- **Discretizzazione dello stato:** La discretizzazione degli stati ha richiesto un'attenta definizione dei bin per dimensione al fine di bilanciare la granularità con l'efficienza computazionale.

5.4 Conclusioni

In conclusione, l'algoritmo **Q-Learning** ha dimostrato prestazioni superiori nel contesto dell'ambiente `MultiplayerPongEnv`. La capacità di esplorare strategie più efficaci ha permesso all'agente di ottimizzare il controllo delle racchette e massimizzare il punteggio cumulativo. D'altra parte, SARSA, pur mostrando una convergenza stabile, è risultato meno performante a causa della sua natura più cauta.

Sviluppi futuri: L'ambiente e i modelli implementati possono essere ulteriormente estesi:

- Testare altri algoritmi di RL, come il Deep Q-Learning (DQN), per valutare l'efficacia di modelli più complessi.
- Introdurre varianti dell'ambiente, come cambiamenti nella velocità della palla o nella dimensione delle racchette.
- Aumentare la granularità della discretizzazione per migliorare ulteriormente la precisione dei modelli.