Sviluppo di un Agente per la Navigazione Indoor

Primo Progetto - Intelligenza Artificiale

Ludovica Genovese matr. 0522501743 - Manuel Sica matr. 0522501870

Contents

1 Introduzione **3**

2 Metodologia di Implementazione **3**
2.1 Primo Approccio . . . 3
2.1.1 Creazione dell'Ambiente 2D . . . 3
2.1.2 Struttura Generale e Parametri . . . 4
2.1.3 Metodo `__init__` . . . 4
2.1.4 Definizione degli Stati e delle Azioni . . . 4
2.1.5 Metodo `reset` . . . 5
2.1.6 Metodo `step` . . . 5
2.1.7 Metodo `render` . . . 6
2.1.8 Metodi Ausiliari per il Posizionamento . . . 6
2.1.9 Metodo `get_state` e `close` . . . 6
2.2 Algoritmi Implementati . . . 7
2.3 Agente Q-Learning . . . 7
2.3.1 Struttura della Classe . . . 7
2.3.2 Scelta dei Parametri . . . 8
2.3.3 Metodo `choose_action` . . . 8
2.3.4 Metodo `learn` . . . 8
2.4 Agente SARSA . . . 9
2.4.1 Struttura della Classe . . . 9
2.4.2 Metodo `choose_action` . . . 10
2.4.3 Metodo `learn` . . . 10
2.5 Agente DQN . . . 10
2.5.1 Architettura della Rete Neurale (`DQN`) . . . 10
2.5.2 Struttura della Classe `DQNAgent` . . . 10
2.5.3 Metodo `choose_action` . . . 11
2.5.4 Metodo `learn` . . . 11
2.5.5 Replay Buffer . . . 11
2.6 Valutazione delle Prestazioni . . . 12
2.6.1 Q-Learning . . . 12
2.6.2 SARSA . . . 13
2.6.3 DQN . . . 14
2.7 Secondo Approccio . . . 16
2.7.1 Creazione dell'Ambiente 2D . . . 16
2.7.2 Definizione degli Stati e delle Azioni . . . 16
2.7.3 Algoritmi Implementati e Confronto . . . 17

2.7.4 Agente Double - DQN 17
2.7.5 Implementazione della Rete Neurale (`DQNetwork`) 17
2.7.6 Struttura della Classe DQNAgent 18
2.7.7 Differenze Chiave Rispetto al Primo Approccio 18
2.8 Valutazione delle Prestazioni e Confronti 19

3 Conclusioni **21**
3.1 Sintesi dei Risultati Ottenuti 21
3.2 Limitazioni 22
3.3 Prospettive Future 22

1 Introduzione

La navigazione autonoma rappresenta un problema centrale nell'ambito dell'intelligenza artificiale e della robotica, con applicazioni che spaziano dalla logistica alla mobilità assistita. Il presente progetto si propone lo sviluppo e l'addestramento di un agente intelligente, basato su algoritmi di apprendimento per rinforzo, per la navigazione autonoma in un ambiente bidimensionale caratterizzato dalla presenza di ostacoli e target da raggiungere.

L'obiettivo principale consiste nella progettazione di un agente in grado di prendere decisioni ottimali durante la navigazione, massimizzando l'efficienza dei movimenti e minimizzando il rischio di collisioni. A tale scopo il progetto prevede:

- La creazione di un ambiente simulato 2D con ostacoli posizionati in modo casuale.
- L'implementazione di un algoritmo di apprendimento per rinforzo per addestrare l'agente a navigare nell'ambiente, evitando gli ostacoli e raggiungendo il target nel minor numero di passi possibile.
- La definizione di stati rappresentativi dell'ambiente, che includano informazioni riguardanti la posizione degli ostacoli e la direzione del target.
- L'ottimizzazione delle azioni disponibili, affinché l'agente possa apprendere una strategia di navigazione efficiente.

L'approccio adottato prevede una fase di addestramento iterativo, nella quale l'agente apprende progressivamente a migliorare le proprie prestazioni attraverso un sistema di ricompense e penalità. I risultati saranno valutati misurando varie metriche, confrontando le performance ottenute da almeno due differenti algoritmi di apprendimento per rinforzo. Inoltre, verranno analizzate le principali sfide affrontate durante l'implementazione e discusse le strategie adottate per superarle, fornendo un'analisi critica delle prestazioni ottenute e prospettive per evoluzioni future del sistema.

2 Metodologia di Implementazione

In questa documentazione vengono presentati due approcci distinti al problema della navigazione autonoma.

Nel **primo approccio** l'ambiente risulta parzialmente osservabile: le osservazioni fornite includono esclusivamente le posizioni dell'agente e del target, mentre informazioni rilevanti quali la posizione degli ostacoli non sono visibili. Di conseguenza, l'agente non dispone di una rappresentazione completa dello stato dell'ambiente, caratterizzandosi così un contesto di parziale osservabilità.

Nel **secondo approccio** l'ambiente è completamente osservabile: lo stato è rappresentato integralmente dalla griglia 10x10, in cui ogni cella indica esplicitamente se è vuota (0), contiene l'agente (1), il target (2) o un ostacolo (3). In tal modo, l'agente dispone di tutte le informazioni necessarie per pianificare un percorso ottimale.

2.1 Primo Approccio

2.1.1 Creazione dell'Ambiente 2D

La presente sezione descrive in dettaglio l'implementazione dell'ambiente di navigazione `NavigationEnvironment`, sviluppato mediante il framework `gym` e la libreria grafica `pygame`. L'ambiente è stato progettato per simulare un'area bidimensionale con ostacoli, in cui un agente

deve muoversi per raggiungere un target, evitando collisioni e minimizzando i percorsi ridondanti.

2.1.2 Struttura Generale e Parametri

L'ambiente accetta i seguenti parametri:

- **grid_size**: dimensione della griglia (default $(10, 10)$).
- **num_obstacles**: numero di ostacoli da posizionare (default 15).
- **cell_size**: dimensione di ogni cella in pixel (default 50).

La dimensione della finestra grafica (`screen_size`) viene calcolata in base a tali impostazioni, aggiungendo uno spazio extra per il rendering di informazioni e pulsanti di controllo. La classe `NavigationEnvironment` estende la classe base di `gym.Env` e integra le funzionalità di `pygame` per la visualizzazione.

2.1.3 Metodo `__init__`

Nel costruttore vengono inizializzate le seguenti componenti:

- Inizializzazione di `pygame` e creazione della finestra grafica.
- Definizione degli spazi delle azioni e delle osservazioni.
- Creazione di una griglia numerica (`grid`) per rappresentare l'ambiente; le celle con valore `0` indicano spazi liberi, mentre le celle con valore `-1` rappresentano ostacoli.
- Definizione delle variabili di stato, quali la posizione dell'agente (`agent_position`), quella del target (`target_position`) e il set delle posizioni già visitate (`visited_positions`), utile per evitare loop.
- Configurazione dei pulsanti per la regolazione della velocità (FPS) e impostazione della velocità iniziale (`speed`).
- Invocazione del metodo `reset()` per inizializzare lo stato dell'ambiente.

2.1.4 Definizione degli Stati e delle Azioni

- **Action Space:** L'ambiente prevede quattro possibili azioni (su, giù, sinistra, destra) implementate mediante uno spazio discreto:

$$\texttt{action\_space} = \texttt{spaces.Discrete(4)}$$

- **Observation Space:** Lo stato osservabile è costituito dalla posizione dell'agente e dal target, organizzati in una struttura di tipo dizionario:

$$\texttt{observation\_space} = \texttt{spaces.Dict}\Big(\{\text{"}agent\text{"} : \texttt{Tuple(Discrete, Discrete)}, \text{"}target\text{"} : \texttt{Tuple(D}$$

2.1.5 Metodo `reset`

Il metodo `reset` opera secondo le seguenti modalità:

- Se l'ultimo episodio si è concluso con successo (quando `last_episode_success` risulta vero), viene ricostruita l'intera griglia:
  - La griglia viene inizializzata con zeri.
  - Gli ostacoli vengono posizionati mediante il metodo `place_obstacles()`.
  - Vengono posizionati, rispettivamente, il target e l'agente tramite `place_target()` e `place_agent()`.
- In caso di fallimento, l'agente viene semplicemente riposizionato, mantenendo invariata la griglia corrente.
- Il set delle posizioni visitate viene reimpostato, includendo la nuova posizione iniziale dell'agente.

Il metodo restituisce lo stato corrente ottenuto tramite `get_state()`.

2.1.6 Metodo `step`

Il metodo `step` gestisce l'evoluzione dello stato in seguito all'applicazione di una determinata azione. I passaggi principali sono:

1. **Calcolo della Nuova Posizione:** In base all'azione ricevuta (0: su, 1: giù, 2: sinistra, 3: destra), viene determinata la nuova posizione dell'agente, assicurandosi che questa non ecceda i limiti della griglia.
2. **Verifica di Collisione:** Qualora la cella di destinazione contenga un ostacolo (valore `-1` nella griglia), l'episodio termina con una penalità severa (-800) e viene restituito un messaggio esplicativo.
3. **Calcolo della Distanza:** Vengono calcolate le distanze euclidee, sia quella precedente che quella successiva al movimento, rispetto al target. Se l'agente raggiunge il target, l'episodio termina con una ricompensa elevata (12000).
4. **Reward Shaping:**
   - L'incentivo per avvicinarsi al target è proporzionale alla variazione di distanza, moltiplicata per un fattore pari a 40.
   - Viene applicata una penalità di 150 se l'agente non si sposta.
   - Se la nuova posizione è già stata visitata, viene applicata un'ulteriore penalità di 1000, al fine di scoraggiare loop.
   - A ogni movimento valido viene aggiunta una ricompensa fissa di 20.
   - Infine, viene sottratto un decremento marginale di -0.1 dal reward.

Il metodo restituisce una tupla contenente:

- Lo stato aggiornato (dizionario con le posizioni dell'agente e del target).
- Il reward ottenuto.
- Un flag booleano che indica se l'episodio è terminato.
- Un dizionario contenente informazioni ausiliarie (ad es., il motivo della terminazione in caso di collisione o raggiungimento del target).

2.1.7 Metodo `render`

Il metodo `render` si occupa della visualizzazione grafica dell'ambiente e opera nel seguente modo:

- Gestisce gli eventi di `pygame`, inclusa la chiusura della finestra e l'interazione con i pulsanti per la regolazione della velocità (`speed`).
- La finestra viene suddivisa in una griglia 2x2, consentendo la visualizzazione simultanea degli stati di più agenti o di differenti episodi, se necessario.
- Per ciascun sotto-schermo:
  - Le celle della griglia vengono disegnate, utilizzando i seguenti colori:
    - **Verde:** Cella contenente l'agente.
    - **Rosso:** Cella contenente il target.
    - **Nero:** Celle occupate da ostacoli.
  - Viene disegnata una griglia con bordi grigi, per una migliore visualizzazione.
  - Vengono mostrate informazioni relative all'episodio (numero dell'episodio e risultato ottenuto).
- Vengono disegnati i pulsanti per aumentare o diminuire la velocità, insieme all'indicazione del valore corrente degli FPS.
- Al termine dell'esecuzione, il display viene aggiornato e il clock di `pygame` viene regolato in base alla velocità impostata.

2.1.8 Metodi Ausiliari per il Posizionamento

- `place_agent`: Seleziona casualmente una cella libera (valore 0 nella griglia) che non coincida con il target.
- `place_target`: Seleziona casualmente una cella libera per posizionare il target.
- `place_obstacles`: Posiziona un numero prefissato di ostacoli nella griglia, verificando con il metodo `can_place_obstacle` l'assenza di ostacoli adiacenti (inclusi i vicini diagonali), al fine di garantire la validità del layout.
- `can_place_obstacle`: Controlla, per una data cella, l'assenza di ostacoli nelle celle adiacenti; tale vincolo evita raggruppamenti eccessivamente ravvicinati.

2.1.9 Metodo `get_state` e `close`

- `get_state`: Restituisce lo stato corrente dell'ambiente sotto forma di dizionario contenente le posizioni (riga, colonna) dell'agente e del target.
- `close`: Chiude l'ambiente grafico invocando il metodo `pygame.quit()`.

2.2 Algoritmi Implementati

In questa sezione vengono descritte in dettaglio tre differenti implementazioni di agenti per il reinforcement learning:

- **Q-Learning:** L'agente utilizza una Q-Table inizializzata dinamicamente, una politica ϵ-greedy potenziata da una softmax parametrizzata tramite una temperatura variabile, aggiornamenti con bonus per successi consecutivi e un decadimento adattativo sia di ϵ sia della temperatura. L'accesso concorrente alla Q-Table è gestito mediante un lock.
- **SARSA:** L'agente on-policy aggiorna la Q-Table basandosi sulla sequenza (s, a, r, s', a'). L'algoritmo segue la politica ϵ-greedy e utilizza un lock per garantire la sicurezza in ambienti concorrenti.
- **DQN (Deep Q-Network):** L'agente sfrutta una rete neurale per approssimare la funzione Q. Il modello viene addestrato mediante experience replay e l'ottimizzazione avviene tramite backpropagation della loss MSE, utilizzando l'ottimizzatore Adam.

Si descrive in dettaglio la scelta dei parametri relativi al Q-Learning, in quanto il ragionamento seguito per SARSA e DQN è identico. Gli algoritmi sono stati testati su diverse fasce di episodi: 2000, 10000, 20000 e 300000. I risultati presentati si riferiscono esclusivamente a 300000 episodi (con un allenamento di circa 12 ore), in quanto rappresentano la configurazione più significativa; gli altri test sono stati effettuati esclusivamente per individuare una soluzione ottimale e verranno presi come esempio e discussi.

2.3 Agente Q-Learning

2.3.1 Struttura della Classe

La classe `QLearningAgent` è caratterizzata dai seguenti attributi:

- **Azioni:** `actions` — l'insieme delle azioni possibili.
- **Q-Table:** `q_table` — dizionario che mappa ogni stato (identificato tramite una chiave) ai relativi Q-values per ciascuna azione.
- **Lock:** `lock` — utilizzato per sincronizzare l'accesso alla Q-Table.
- **Parametri di apprendimento:**
  - α (learning rate) = 0.1,
  - γ (fattore di sconto) = 0.99.
- **Politica di esplorazione:** Il parametro ϵ (inizialmente 1.0) decresce fino a un minimo di 0.05, con un decadimento pari a 0.995.
- **Softmax e Temperature:** La temperatura, inizialmente impostata a 1.0, modula la distribuzione softmax sui Q-values e decresce fino a 0.1 con un decadimento pari a 0.999.
- **Success Streak:** `success_streak` — contatore utilizzato per incentivare episodi di successo consecutivi, applicando bonus al Q-value.

2.3.2 Scelta dei Parametri

Per ottimizzare i parametri è stato fondamentale trovare un buon equilibrio tra esplorazione e sfruttamento. A tale scopo è stato adottato un approccio ϵ-greedy, che prevede un periodo iniziale in cui si privilegia l'esplorazione (determinato dal valore iniziale di ϵ). Con il progredire degli episodi, il valore di ϵ viene progressivamente diminuito, sfruttando le informazioni acquisite durante la fase esplorativa. L'obiettivo era individuare il giusto tasso di decadimento di ϵ che garantisse un compromesso ottimale tra esplorazione e sfruttamento.

Inizialmente, il tasso di decadimento di ϵ è stato fissato a 0.995. Per verificare l'efficacia di tale scelta, sono state sperimentate diverse impostazioni con valori di decadimento pari a 0.99, 0.999, 0.9995, 0.997 e 0.998. Alcuni di questi valori hanno portato a una persistenza eccessiva dell'esplorazione anche dopo numerosi episodi.

Parallelamente, per affrontare il problema dell'esplorazione è stata implementata la tecnica softmax. In questo caso, le azioni vengono selezionate sulla base di una distribuzione di probabilità derivante dai relativi Q-values, in modo da favorire le azioni con valori maggiori, pur mantenendo una quota di esplorazione. A supporto della softmax è stato introdotto un parametro di temperatura che modula la distribuzione:

- Un valore elevato di temperatura aumenta la probabilità di esplorare.
- Un valore basso favorisce la selezione delle azioni con Q-values maggiori.

Anche la temperatura viene decisa tramite un decadimento, inizialmente impostata a 0.999. Per verificare l'equilibrio ottenuto, sono state testate impostazioni alternative (ad es., 0.998 e 0.997). Si è osservato che, con un decadimento pari a 0.997, l'agente tendeva a finire in loop dopo circa 2000 episodi; per tale motivo non sono stati esplorati ulteriori valori.

2.3.3 Metodo `choose_action`

I passaggi fondamentali sono i seguenti:

1. **Conversione dello Stato in Chiave:** Il metodo `state_to_key` normalizza le posizioni (assumendo una griglia 10x10) e le formatta in una stringa con precisione di due decimali.
2. **Inizializzazione:** Se lo stato non è presente nella Q-Table, esso viene inizializzato con piccoli valori casuali (compresi in $[0.0, 0.01]$) per ciascuna azione.
3. **Politica ϵ-greedy:** Con probabilità ϵ viene scelta un'azione casuale.
4. **Scelta Tramite Softmax:** In assenza di esplorazione, i Q-values vengono normalizzati mediante una softmax controllata dalla temperatura attuale, e l'azione viene scelta in base alla distribuzione risultante.

2.3.4 Metodo `learn`

L'aggiornamento della Q-Table avviene secondo i seguenti passaggi:

1. Conversione degli stati corrente e successivo in chiavi.
2. Inizializzazione degli stati mancanti nella Q-Table.
3. Aggiornamento secondo la seguente formula:

$$Q(s,a) \leftarrow Q(s,a) + \alpha\Big(r + \gamma \max_{a'} Q(s',a') - Q(s,a)\Big)$$

4. Se l'episodio termina (`done`) e il reward è positivo, il contatore `success_streak` viene incrementato; in caso contrario, esso viene resettato. Un bonus, proporzionale a $100 \times$ `success_streak`, viene aggiunto al Q-value.

5. Decadimento di ϵ e della temperatura tramite i metodi privati $\_decay\_epsilon()$ e $\_decay\_temperature()$.

Si evidenzia come, a differenza degli algoritmi successivi, nel Q-Learning sia stata sperimentata una configurazione multi-agente impiegando quattro agenti, sia in modalità cooperativa che competitiva. Tale approccio è stato tuttavia rapidamente scartato per le seguenti ragioni:

1. **Incoerenza dei Valori Q:** L'utilizzo di più agenti ha portato ad ottenere valori Q completamente fuori contesto, rendendo difficile per ciascun agente comprendere l'obiettivo finale.

2. **Uniformità dell'Ambiente:** Era necessario garantire che tutti gli agenti operassero nello stesso ambiente, con una disposizione identica di ostacoli e target. Poiché i target erano dinamici (la loro posizione cambiava ad ogni raggiungimento) e gli ostacoli aumentavano con il crescere del numero di target acquisiti in un episodio, assicurare un ambiente uniforme comportava tempi di addestramento molto lunghi. In effetti, non è stata trovata una soluzione nemmeno dopo 20.000 episodi. In scenari cooperativi, qualora un agente fallisse o si riproponeva la stessa configurazione, si rendeva necessario bloccare gli altri tre agenti.

3. **Gestione delle Modalità Cooperativa e Competitiva:**

   - *Cooperativo:* Per questo approccio, le Q-Table dei quattro agenti venivano unite in un'unica tabella.
   - *Competitivo:* In questa modalità veniva salvata la Q-Table dell'agente migliore. Durante l'inferenza, tuttavia, l'agente non si dirigeva verso il target, ma agiva in maniera casuale, dimostrando di non aver compreso l'obiettivo finale.

Per tali ragioni, l'approccio multi-agente nel Q-Learning è stato completamente abbandonato.

2.4 Agente SARSA

2.4.1 Struttura della Classe

La classe `SARSAAgent` implementa un algoritmo on-policy che utilizza:

- **Azioni:** `actions` — l'insieme delle azioni possibili.
- **Q-Table:** `q_table` — un dizionario che mappa gli stati (convertiti in stringa) ai Q-values.
- **Parametri di Aggiornamento:**
  - $\alpha = 0.1$ (learning rate),
  - $\gamma = 0.99$ (fattore di sconto).
- **Politica di Esplorazione:** Basata su ϵ-greedy, con ϵ inizialmente pari a 1.0, che decresce fino a 0.01 con un decadimento di 0.995.
- **Lock:** Utilizzato per gestire l'accesso concorrente alla Q-Table.

2.4.2 Metodo `choose_action`

- Lo stato viene convertito in chiave tramite `str(state)`.
- Se un numero casuale risulta inferiore a ϵ oppure lo stato non è presente nella Q-Table, viene scelta un'azione casuale.
- Altrimenti, viene restituita l'azione con il Q-value massimo.

2.4.3 Metodo `learn`

L'aggiornamento segue la regola SARSA:

$$Q(s,a) \leftarrow Q(s,a) + \alpha\Big(r + \gamma\, Q(s',a') - Q(s,a)\Big)$$

- **Selezione della Prossima Azione:** La variabile `next_action` viene scelta applicando la stessa politica ϵ-greedy sullo stato successivo.
- **Inizializzazione:** Se lo stato corrente o quello successivo non sono presenti nella Q-Table, essi vengono inizializzati con zero per ciascuna azione.
- **Decadimento di ϵ:** Al termine dell'aggiornamento, ϵ viene ridotto, moltiplicandolo per il fattore di decadimento.

2.5 Agente DQN

2.5.1 Architettura della Rete Neurale (`DQN`)

La classe `DQN` definisce un modello neurale semplice:

- **Input Layer:** Dimensione pari a `input_dim` (ad esempio 100).
- **Hidden Layer:** Un layer completamente connesso con 128 neuroni e funzione di attivazione ReLU.
- **Output Layer:** Numero di neuroni pari al numero delle azioni, ciascuno rappresentante il Q-value stimato per l'azione corrispondente.

2.5.2 Struttura della Classe `DQNAgent`

Il `DQNAgent` sfrutta il modello neurale per approssimare la funzione Q:

- **Azioni:** L'insieme delle azioni possibili.
- **Replay Buffer:** Un buffer per memorizzare esperienze sotto forma di tuple (s, a, r, s', d).
- **Modello:** Istanza della rete `DQN`.
- **Ottimizzatore:** Adam, con learning rate pari a 0.001.
- **Fattore di Sconto:** $\gamma = 0.99$.
- **Lock:** Per la gestione sicura delle risorse condivise.

2.5.3 Metodo `choose_action`

- Con una probabilità fissa (ad esempio, il 10%) viene scelta un'azione casuale per favorire l'esplorazione.
- Altrimenti, lo stato (flattened in un vettore) viene convertito in un tensore e passato attraverso la rete per ottenere i Q-values; l'azione scelta è quella che massimizza il Q-value.

2.5.4 Metodo `learn`

Il processo di aggiornamento avviene mediante experience replay e segue i seguenti passaggi:

1. Se il replay buffer contiene meno di 32 esperienze, l'aggiornamento viene saltato.
2. Viene campionato un batch casuale di 32 esperienze.
3. Per ciascuna esperienza, il **target** viene calcolato come:

$$\text{target} = r + \gamma \cdot \max_{a'} Q(s', a') \cdot (1 - d)$$

   dove d indica se lo stato successivo è terminale.
4. La loss viene calcolata utilizzando l'errore quadratico medio (MSE) tra il Q-value stimato per l'azione scelta e il target.
5. L'ottimizzatore aggiorna i pesi della rete mediante backpropagation, minimizzando così la loss.

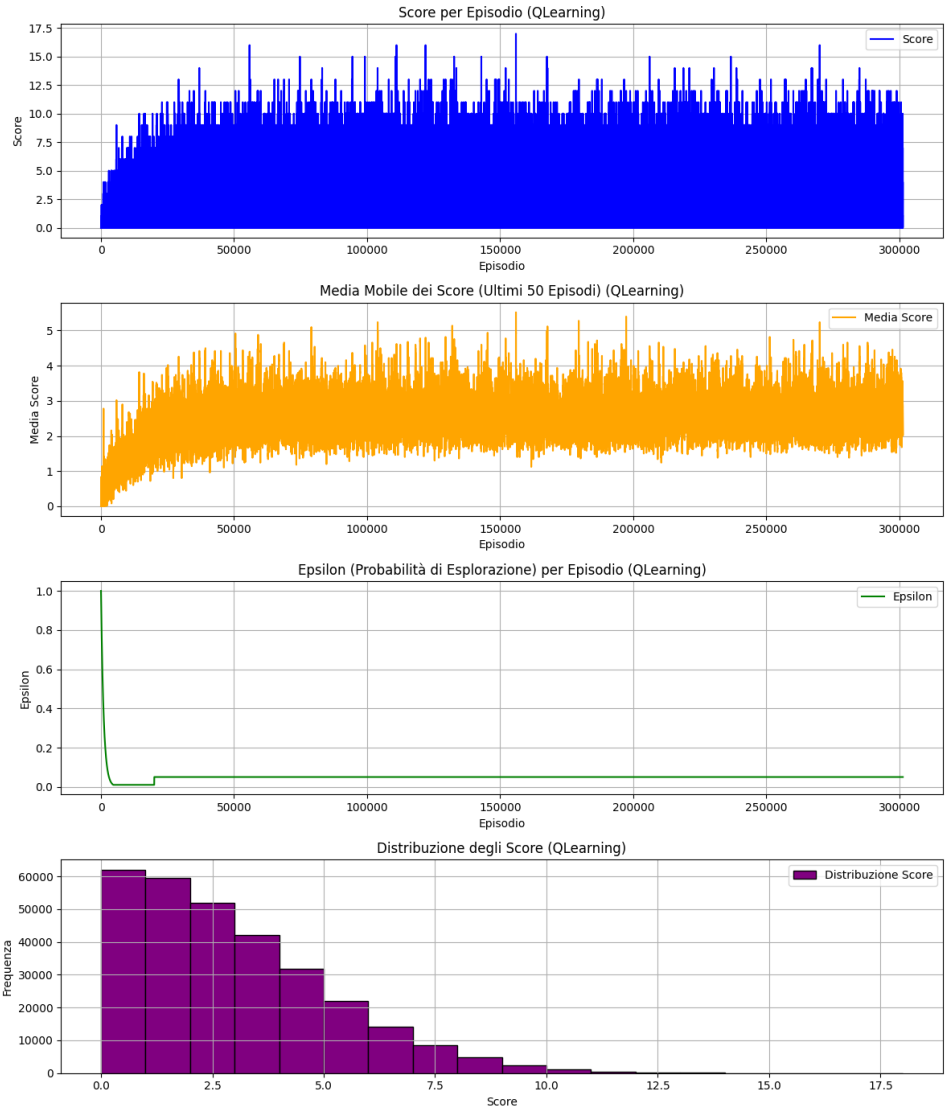
2.5.5 Replay Buffer

La classe `ReplayBuffer` gestisce:

- **Memorizzazione:** Le esperienze vengono aggiunte in un buffer di capacità fissa; in caso di capacità esaurita, viene utilizzata una struttura circolare (FIFO).
- **Campionamento:** Il metodo `sample(batch_size)` restituisce un batch casuale di esperienze per l'aggiornamento del modello.

2.6 Valutazione delle Prestazioni

2.6.1 Q-Learning



Score per Episodio

- La crescita iniziale risulta simile a quella osservata nel DQN, sebbene con un plateau leggermente inferiore e una maggiore variabilità degli score.

Media Mobile degli Score (Ultimi 50 Episodi)

- Il trend è positivo, ma si stabilizza sotto i 4 punti. Ciò indica che, sebbene l'apprendimento sia efficace, la performance complessiva risulta meno ottimale, suggerendo la necessità di un miglioramento nella politica di aggiornamento o nella funzione di ricompensa.

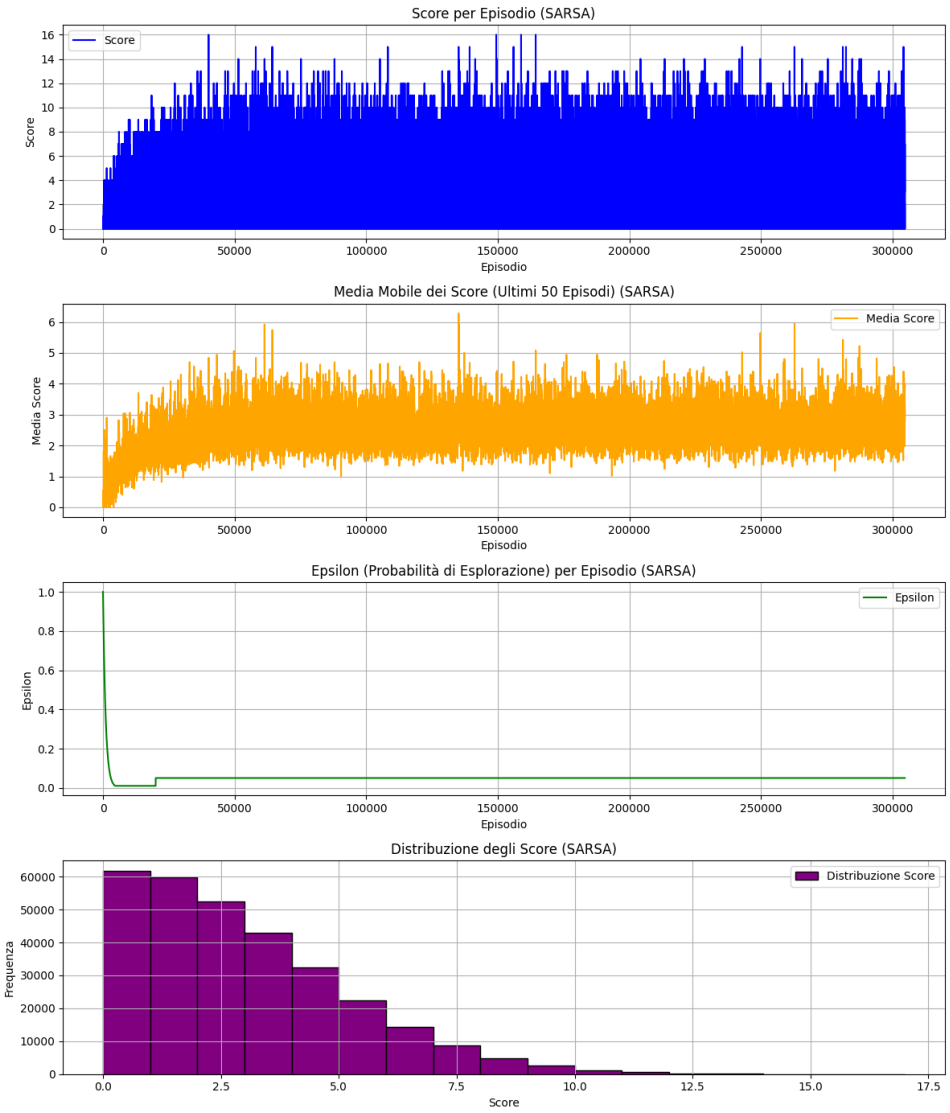
Epsilon per Episodio

- La decrescita dell'epsilon è rapida e simile a quella osservata nel DQN, con un breve plateau prima della discesa finale, suggerendo un periodo di esplorazione costante prima della piena exploitation.

Distribuzione degli Score

- La distribuzione degli score presenta una maggiore concentrazione di valori bassi rispetto al DQN, evidenziando come il Q-Learning tenda a generare episodi di performance inferiori rispetto a quelli ottenuti con il Double DQN.

2.6.2 SARSA



Score per Episodio

- L'andamento degli score è simile a quello del Q-Learning, con una leggera tendenza a stabilizzarsi su valori inferiori rispetto al DQN, dovuta al carattere on-policy dell'algoritmo, che rende il comportamento dell'agente più conservativo.

Media Mobile degli Score (Ultimi 50 Episodi)

- La media mobile mostra una crescita costante, con un plateau leggermente superiore a quello del Q-Learning ma inferiore a quello del DQN. L'agente apprende strategie affidabili, pur adottando un approccio meno aggressivo.

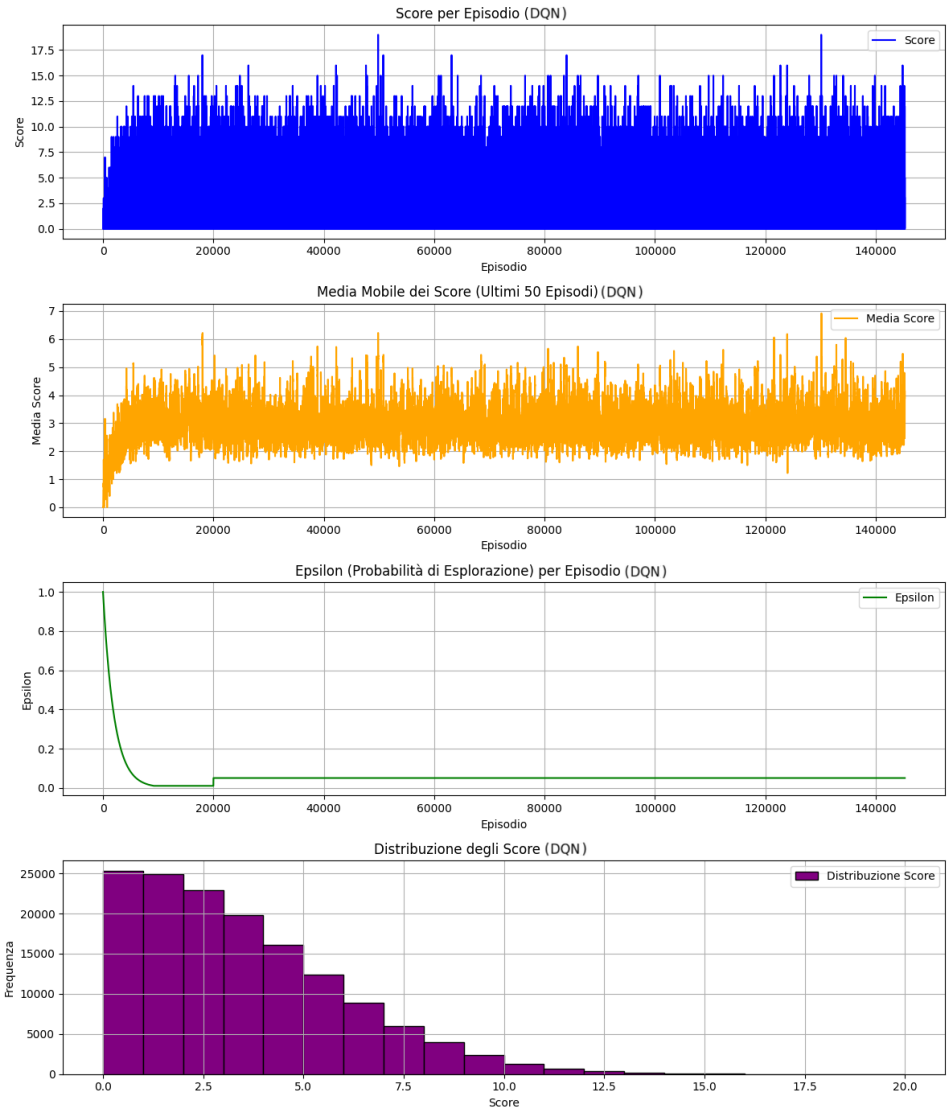
Epsilon per Episodio

- La decrescita di ϵ risulta simile agli altri algoritmi, sebbene la fase iniziale di esplorazione sia più prolungata, consentendo un apprendimento più vario ma meno focalizzato sull'ottimizzazione massimale delle performance.

Distribuzione degli Score

- La distribuzione mostra una concentrazione ancora maggiore di score bassi rispetto agli altri algoritmi, evidenziando come SARSA, per il suo approccio conservativo, tenda a produrre performance stabili ma meno eccezionali.

2.6.3 DQN



Score per Episodio

- Lo score cresce rapidamente nelle fasi iniziali, raggiungendo un plateau intorno a 10-13 punti, con picchi sporadici che arrivano fino a 17-18 punti. L'agente apprende in modo efficace nella fase iniziale, anche se la variabilità residua suggerisce che, in alcuni episodi, l'agente commette errori, probabilmente a causa di dinamiche ambientali complesse.

Media Mobile degli Score (Ultimi 50 Episodi)

- La curva mostra un aumento costante e una stabilizzazione intorno a 4-5 punti, indicando che il modello ha raggiunto una performance stabile, pur non essendo massimizzata.

Epsilon per Episodio (Probabilità di Esplorazione)

- L'epsilon decresce rapidamente, stabilizzandosi vicino allo zero dopo circa 20.000 episodi. L'agente, pertanto, smette presto di esplorare nuove strategie, concentrandosi sull'exploitation, il che spiega la stabilizzazione degli score, sebbene possa limitare ulteriori miglioramenti.

Distribuzione degli Score

- La distribuzione presenta una modalità intorno a 2-3 punti, con una coda lunga fino a 17-18 punti. La maggior parte degli episodi mostra performance medie, intervallate da alcuni episodi eccellenti. La presenza di score bassi suggerisce che, in alcune situazioni, l'agente fatica a gestire scenari particolarmente complessi.

Riassumendo, si può osservare il seguente confronto:

Aspetto	**DQN**	**Q-Learning**	**SARSA**
Score Medio	Alto e stabile	Moderato con alta variabilità	Moderato-basso e conservativo
Epsilon Decay	Rapido, con focus su exploitation	Rapido con breve plateau intermedio	Più graduale, con esplorazione prolungata
Distribuzione Score	Maggior frequenza di score elevati	Predominanza di score medi e bassi	Frequenza maggiore di score bassi
Stabilità	Alta	Media	Alta coerenza, con meno picchi positivi

Table 1: Confronto tra gli algoritmi di reinforcement learning.

- **DQN** risulta il più efficace nel raggiungere performance elevate e stabili, grazie al meccanismo di double estimation.
- **Q-Learning** offre una buona performance complessiva, ma risente di una maggiore variabilità.
- **SARSA** adotta un comportamento più conservativo, garantendo una maggiore coerenza, particolarmente utile in scenari in cui la sicurezza e la stabilità delle decisioni sono prioritarie.

2.7 Secondo Approccio

Nel **primo approccio**, il Q-Learning tabellare viene utilizzato in ambienti con spazi di stato discreti, dove la funzione Q è memorizzata in una Q-Table e aggiornata tramite:

$$Q(s,a) \leftarrow Q(s,a) + \alpha\Big(r + \gamma \max_{a'} Q(s',a') - Q(s,a)\Big)$$

Sebbene questa metodologia risulti efficace per problemi a bassa dimensionalità, essa si rivela limitata in termini di scalabilità quando l'ambiente diventa più complesso.

Il **secondo approccio** impiega invece un Deep Q-Network (DQN) con architettura *Dueling* e aggiornamenti Double. In questo caso la funzione Q è approssimata da una rete neurale che sfrutta due flussi distinti:

- **Value Stream:** stima il valore dello stato $V(s)$,
- **Advantage Stream:** stima i vantaggi relativi alle azioni $A(s,a)$.

Il Q-value viene quindi ricostruito come:

$$Q(s,a) = V(s) + \Big(A(s,a) - \frac{1}{|\mathcal{A}|}\sum_{a'} A(s,a')\Big)$$

L'approccio *Double DQN* utilizza due reti neurali (policy network e target network) per ridurre l'overestimation, aggiornando periodicamente la target network con i pesi della policy network. Nel secondo approccio, poiché i risultati sono simili tra Q-Learning e SARSA, non è stato implementato l'algoritmo SARSA.

2.7.1 Creazione dell'Ambiente 2D

L'ambiente 2D è implementato come descritto nel primo approccio e prevede:

- Una griglia fissa (ad es. 10x10) con celle di dimensione predefinita.
- La presenza dinamica di un agente, di un target e di ostacoli. Quando il target viene raggiunto, l'agente genera un nuovo ostacolo e il target viene riposizionato; in caso di collisione, gli ostacoli vengono resettati.
- Un'interfaccia grafica basata su `pygame` che include controlli interattivi (pulsanti per interrompere l'allenamento e regolare gli FPS).

Le funzionalità di `GridEnvironment` (reset, gestione degli eventi, rendering) rimangono invariati rispetto al primo approccio.

2.7.2 Definizione degli Stati e delle Azioni

Lo stato dell'ambiente è rappresentato da una tupla:

(`agent_x`, `agent_y`, `target_x`, `target_y`, `obstacle_hash`)

dove:

- `agent_x` e `agent_y` indicano la posizione dell'agente.
- `target_x` e `target_y` rappresentano la posizione del target.
- `obstacle_hash` codifica la disposizione degli ostacoli nella griglia.

Le azioni sono codificate come numeri interi:

- 0: Su,
- 1: Giù,
- 2: Sinistra,
- 3: Destra.

2.7.3 Algoritmi Implementati e Confronto

Sono stati implementati due approcci principali:

1. **Q-Learning Tabellare:** L'agente apprende una Q-Table aggiornata mediante una politica ϵ-greedy. Questo metodo, sebbene semplice ed efficace in ambienti a bassa dimensionalità, non scala bene in scenari più complessi.
2. **Dueling Double DQN:** L'agente utilizza una rete neurale per approssimare la funzione Q, sfruttando una struttura dueling e il meccanismo Double DQN per stabilizzare l'apprendimento. Inoltre, viene impiegato un replay buffer per campionare esperienze e migliorare la convergenza. Questo approccio è particolarmente adatto a spazi di stato elevati o continui e consente un addestramento più robusto rispetto al Q-Learning tabellare.

Di seguito verrà descritto in dettaglio il secondo approccio, evidenziandone le differenze rispetto al primo.

2.7.4 Agente Double - DQN

Il Dueling Double DQN supera alcune limitazioni del Q-Learning tabellare:

- **Scalabilità:** In ambienti complessi con spazi di stato elevati, l'approccio tabellare risulta inefficiente. Utilizzando una rete neurale, il DQN è in grado di generalizzare su stati simili.
- **Stabilità dell'Apprendimento:** L'architettura dueling separa il valore dello stato dai vantaggi delle azioni, permettendo una migliore stima dei Q-values. Inoltre, l'uso di due reti (Double DQN) riduce l'overestimation.
- **Experience Replay:** La memorizzazione delle esperienze e il successivo campionamento casuale rompono la correlazione tra dati consecutivi, migliorando la stabilità dell'apprendimento.

A differenza del Q-Learning tabellare, nel quale ogni stato viene trattato in maniera isolata, il DQN apprende una funzione continua, consentendo un migliore adattamento in ambienti dinamici e complessi.

2.7.5 Implementazione della Rete Neurale (`DQNetwork`)

La classe `DQNetwork` implementa la seguente architettura:

- **Input Layer:** Riceve lo stato, di dimensione 5 (comprensivo delle coordinate e dell'hash degli ostacoli).

- **Layer Intermedi:** Due layer fully-connected di 128 neuroni ciascuno, con attivazione ReLU.
- **Value Stream:** Un layer lineare che produce $V(s)$, il valore dello stato.
- **Advantage Stream:** Un layer lineare che produce $A(s, a)$ per ciascuna delle 4 azioni.

Il Q-value finale è calcolato combinando i due stream secondo:

$$Q(s, a) = V(s) + \left(A(s, a) - \frac{1}{4}\sum_{a'} A(s, a')\right)$$

2.7.6 Struttura della Classe DQNAgent

Il `DQNAgent` gestisce il processo di apprendimento come segue:

- **Politica di Esplorazione:** Viene adottata una politica ϵ-greedy, simile a quella del Q-Learning tabellare, applicata ai Q-values stimati dalla rete neurale.
- **Replay Buffer:** Le esperienze (stato, azione, reward, stato successivo, done) vengono memorizzate in un buffer di capacità fissa. In fase di aggiornamento, viene effettuato un campionamento casuale in batch (ad esempio 64 esperienze).
- **Aggiornamento della Rete:** Per ogni batch, si calcola il target:

  $$\text{target} = r + \gamma \cdot \max_{a'} Q_{\text{target}}(s', a') \cdot (1 - d)$$

  dove Q_{target} rappresenta la stima della target network e d indica se lo stato successivo è terminale. La loss viene calcolata come l'errore quadratico medio (MSE) tra il Q-value stimato dalla policy network e il target.
- **Double DQN:** I pesi della policy network vengono periodicamente copiati nella target network per stabilizzare l'apprendimento.

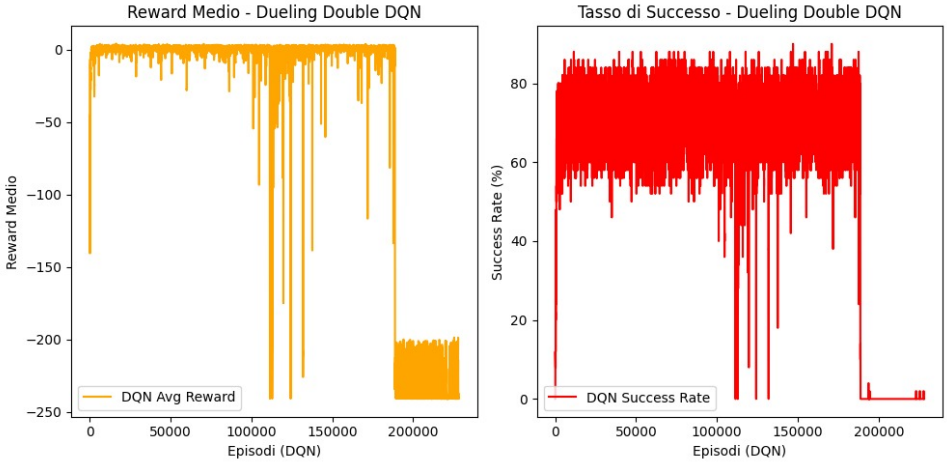
2.7.7 Differenze Chiave Rispetto al Primo Approccio

- **Funzione di Approssimazione:**
  - *Primo Approccio:* Utilizza una Q-Table per mappare stati e azioni, trattando ciascuno stato in maniera discreta.
  - *Secondo Approccio:* Impiega una rete neurale per approssimare la funzione Q, permettendo una generalizzazione su dati simili.
- **Gestione dello Stato:**
  - *Primo Approccio:* Ogni stato viene indicizzato come chiave nel dizionario, senza tenere conto della correlazione tra stati simili.
  - *Secondo Approccio:* La rete neurale apprende rappresentazioni continue dello stato, catturando le relazioni tra stati adiacenti.
- **Stabilità dell'Apprendimento:**
  - *Primo Approccio:* L'aggiornamento della Q-Table è diretto, ma può risultare instabile in ambienti caratterizzati da numerose variabili.

- *Secondo Approccio:* L'uso di due reti (policy e target) e del replay buffer garantisce un apprendimento più robusto e stabile.

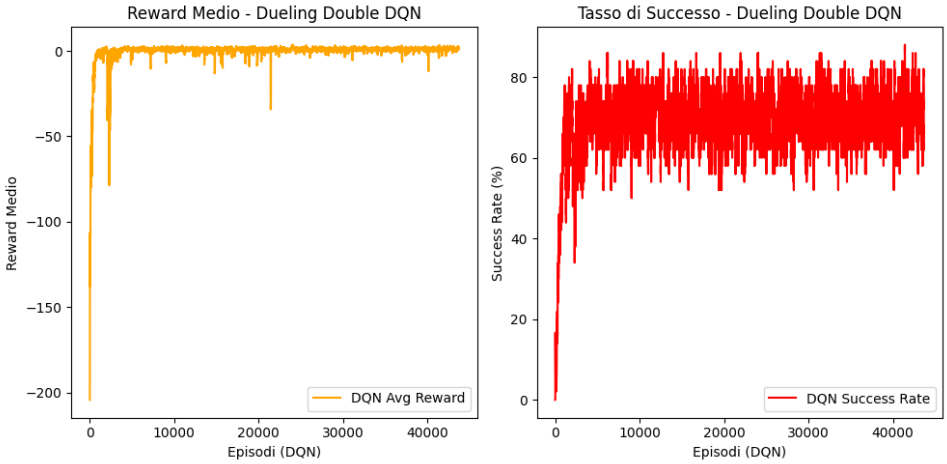
- **Scalabilità:**
  - *Primo Approccio:* Adeguato per ambienti a bassa dimensionalità, ma inefficiente in scenari complessi.
  - *Secondo Approccio:* Consente la gestione di spazi di stato elevati o continui, grazie alla capacità di generalizzazione della rete neurale.

Si osserva inoltre che in entrambi gli approcci si è verificato il problema del loop, ovvero l'agente che procede ripetutamente avanti e indietro (ad esempio: avanti → indietro → sinistra → destra → avanti → indietro, ecc.). Tale problematica persiste nel primo approccio e, pur essendo parzialmente mitigata con il modello Double-DQN, l'agente tende a bloccarsi dopo circa 200000 episodi, interrompendo l'addestramento. Nonostante siano state adottate soluzioni, come l'impossibilità di rimanere fermi su una cella, il problema non è stato completamente risolto.



2.8 Valutazione delle Prestazioni e Confronti

In questa sezione viene presentato il confronto delle prestazioni, concentrandosi principalmente sul DQN, non si mostra i risultati del Q-learning, poiché durante l'inferenza i problemi del loop continuano a persistere.



Reward Medio - Dueling Double DQN

- Il reward medio parte da valori molto negativi (oltre -200), evidenziando una performance iniziale scarsa.
- Nei primi 2000 episodi si osserva un rapido miglioramento, seguito da una stabilizzazione intorno a valori prossimi allo zero o leggermente positivi.
- Occasionalmente si registrano picchi negativi, anche in fase di stabilizzazione.

Da questi risultati si deduce che:

- Il rapido aumento iniziale del reward medio riflette l'efficacia dell'algoritmo nell'apprendere strategie di base.
- La stabilizzazione intorno allo zero indica che il modello è in grado di evitare penalità gravi, pur non massimizzando completamente le ricompense.
- I picchi negativi sporadici possono derivare da scenari particolarmente complessi o situazioni difficili da generalizzare.

Tasso di Successo - Dueling Double DQN

- Il tasso di successo parte da valori vicini allo 0% nei primi episodi, crescendo rapidamente fino a raggiungere un plateau compreso tra il 70% e l'85% dopo circa 5000 episodi.
- Nonostante il plateau, si osservano fluttuazioni, con valori che oscillano tra il 60% e l'85%.

Da ciò si evince che:

- L'incremento rapido del tasso di successo indica una veloce acquisizione di strategie efficaci da parte dell'agente.
- Le fluttuazioni residue possono essere attribuite sia all'esplorazione residua, dovuta a un epsilon non completamente decaduto, sia alla complessità intrinseca dell'ambiente.

Facendo un confronto con gli algoritmi del primo approccio, si riassume quanto segue:

Velocità di Apprendimento

- Il Dueling Double DQN mostra una velocità di apprendimento superiore rispetto agli altri algoritmi, raggiungendo rapidamente un elevato tasso di successo e un reward medio stabile.
- Anche il DQN tradizionale apprende rapidamente, ma l'architettura Dueling ne potenzia ulteriormente le performance.

Stabilità delle Performance

- Il DQN presenta una maggiore stabilità degli score rispetto al Dueling Double DQN, che tuttavia registra ancora alcune oscillazioni nel tasso di successo.
- SARSA e Q-Learning mostrano performance più conservative, con SARSA che privilegia la stabilità a scapito della performance massima.

Aspetto	Dueling Double DQN	DQN	Q-Learning	SARSA
Reward Medio	Rapido miglioramento, stabilizzazione a ∼0	Crescita moderata con picchi fino a 10–13	Crescita meno stabile, maggiore variabilità	Performance più conservativa, stabile ma bassa
Tasso di Successo	Raggiunge 70–85% rapidamente	Non presente nei grafici	Non presente nei grafici	Non presente nei grafici
Stabilità	Buona, sebbene con fluttuazioni residue	Alta stabilità sugli score	Variabilità significativa	Stabilità conservativa
Apprendimento Iniziale	Molto rapido	Rapido ma più graduale rispetto al Dueling	Moderato	Graduale e conservativo

Table 2: Confronto tra gli algoritmi di reinforcement learning tra primo e secondo approccio

Gestione delle Situazioni Complesse

- Le oscillazioni riscontrate nel Dueling Double DQN, sia nel tasso di successo sia nei reward occasionalmente negativi, suggeriscono che, pur essendo più efficiente, l'algoritmo non è ancora perfettamente robusto in tutti gli scenari complessi.
- In confronto, il DQN tradizionale riesce a mantenere performance più costanti una volta raggiunto il plateau.

In sintesi, si può concludere che:

- Il **Dueling Double DQN** rappresenta un'evoluzione efficace rispetto al DQN tradizionale, mostrando un apprendimento più rapido e un tasso di successo più elevato.
- Tuttavia, esso presenta ancora oscillazioni che potrebbero essere ridotte mediante un fine-tuning dei parametri o modifiche nella politica di esplorazione.
- Il **DQN** tradizionale rimane più stabile per quanto riguarda gli score complessivi, mentre **Q-Learning** e **SARSA** mostrano performance inferiori, sebbene caratterizzate da un comportamento più prevedibile e conservativo.

3 Conclusioni

3.1 Sintesi dei Risultati Ottenuti

I risultati ottenuti evidenziano come l'utilizzo di algoritmi di apprendimento per rinforzo, in particolare il DQN e il Dueling Double DQN, consenta di raggiungere performance notevoli in termini di velocità di apprendimento e tasso di successo, sebbene persistano alcune oscillazioni e problematiche legate alla gestione di scenari complessi.

3.2 Limitazioni

Nonostante i risultati promettenti ottenuti, il sistema presenta diverse criticità che ne limitano la robustezza e la scalabilità in scenari più complessi. In particolare:

- **Dinamica e variabilità dell'ambiente:** La disposizione casuale degli ostacoli impedisce all'agente di apprendere pattern fissi, rendendo difficile la generalizzazione delle strategie e richiedendo un continuo adattamento a configurazioni imprevedibili.
- **Cicli ripetitivi (loop):** L'agente tende a incorrere in comportamenti ciclici, soprattutto in situazioni ambigue, il che rallenta il processo di apprendimento e può compromettere l'efficienza della navigazione.
- **Scalabilità dell'approccio tabellare:** Sebbene la Q-Table funzioni bene in ambienti a bassa dimensionalità, essa non è adatta a gestire spazi di stato elevati. L'adozione di reti neurali (come nei modelli DQN) migliora la capacità di generalizzazione, ma introduce ulteriori sfide in termini di stabilità e ottimizzazione.
- **Sensibilità ai parametri:** La performance degli algoritmi è fortemente influenzata dalla scelta dei parametri (learning rate, fattore di sconto, decadimento di ϵ, temperatura, ecc.). Un bilanciamento inadeguato può comportare una riduzione dell'esplorazione o un prolungamento eccessivo dell'addestramento.
- **Oscillazioni nei reward e nel tasso di successo:** Alcuni modelli, in particolare le architetture avanzate come il Dueling Double DQN, evidenziano fluttuazioni sia nei reward che nel tasso di successo, segnale di una certa instabilità in presenza di scenari complessi.
- **Gestione delle configurazioni multi-agente:** Le sperimentazioni con approcci multi-agente hanno messo in luce problemi di coerenza nella Q-Table e difficoltà nel mantenere un ambiente uniforme, fattori che hanno notevolmente allungato i tempi di addestramento.
- **Overfitting e gestione del replay buffer:** Nei modelli DQN, la gestione del replay buffer è cruciale. Un campionamento non ottimale può portare a un eccessivo adattamento su esperienze passate, riducendo la capacità dell'agente di affrontare nuove situazioni.

Questi aspetti indicano che, pur rappresentando una base valida, l'approccio adottato richiede ulteriori ottimizzazioni – sia nella configurazione degli algoritmi che nella progettazione dell'ambiente – per migliorare la robustezza e garantire una migliore scalabilità in contesti reali e dinamici.

3.3 Prospettive Future

Il progetto apre a numerose direzioni di sviluppo volte a superare le limitazioni attuali e a potenziare l'efficacia del sistema in scenari complessi. Tra le possibili evoluzioni, si evidenziano le seguenti aree:

- **Ottimizzazione degli iperparametri e strategie di esplorazione:** Un tuning più fine di parametri quali il learning rate, il decadimento di ϵ e la temperatura, unitamente all'adozione di tecniche di esplorazione avanzate (ad esempio, l'Upper Confidence Bound o meccanismi basati su entropia), potrebbe migliorare la convergenza e ridurre la frequenza di comportamenti ciclici.

- **Architetture neurali avanzate:** L'impiego di modelli più sofisticati, come reti convoluzionali o architetture ricorrenti (RNN, LSTM), potrebbe garantire una migliore cattura delle informazioni spaziali e temporali. Inoltre, l'integrazione di tecniche di transfer learning favorirebbe l'adattamento a nuove configurazioni ambientali.

- **Sviluppo di approcci multi-agente:** La progettazione di sistemi in cui più agenti cooperano o competono in maniera coordinata potrebbe ampliare le applicazioni del framework. In questo contesto, meccanismi di comunicazione e sincronizzazione tra agenti potrebbero migliorare l'efficienza complessiva e ridurre le incoerenze nei valori Q.

- **Gestione avanzata del replay buffer:** L'adozione di tecniche come il prioritized experience replay permetterebbe di dare maggiore rilevanza alle esperienze più significative, migliorando la robustezza dell'addestramento e mitigando il rischio di overfitting.

- **Validazione in ambienti reali e simulati:** Estendere il framework a scenari reali, integrando dati provenienti da sensori e affrontando il rumore e le imprecisioni intrinseche nella percezione, rappresenta un passo fondamentale per testare l'efficacia del sistema in situazioni concrete. Ciò includerebbe anche il passaggio dalla simulazione al mondo reale (sim-to-real transfer).

- **Integrazione di apprendimento ibrido:** Combinare l'apprendimento per rinforzo con approcci supervisionati o di imitation learning, sfruttando feedback esterni o dati etichettati, potrebbe accelerare il processo di apprendimento e rendere le strategie adottate dall'agente più robuste e affidabili.

Queste prospettive future mirano a creare un sistema più scalabile, stabile e in grado di operare efficacemente in ambienti dinamici e complessi, aprendo la strada a applicazioni pratiche nell'ambito della navigazione autonoma.