

Reinforcement Learning per la risoluzione di labirinti 3D: un framework per la simulazione in Unity

Relatore: *Prof. Gianluigi Ciocca*

Correlatore: *Prof. Davide Marelli*

Relazione della prova finale di:

Asia Zakiah Piazza

Matricola 899552

Anno Accademico 2024-2025

Abstract

Questa tesi si propone di simulare, mediante tecniche di Reinforcement Learning, il comportamento di un agente intelligente capace di risolvere labirinti caratterizzati dalla presenza di ostacoli. Utilizzando la libreria Unity ML-Agents e l'algoritmo Proximal Policy Optimization (PPO), l'agente apprende una strategia esplorativa efficace, guidata da una funzione di ricompensa progettata ad hoc. A supporto dell'allenamento, è stato sviluppato un framework che non solo facilita e ottimizza l'addestramento, ma consente anche la creazione e personalizzazione dell'agente stesso. I diversi allenamenti necessari per ottenere il modello finale sono stati condotti sfruttando tale framework, che ha permesso una gestione efficiente degli ambienti e dei parametri. I risultati ottenuti dal modello finale evidenziano la sua capacità di generalizzare il comportamento appreso anche su labirinti non visti in fase di addestramento, con strutture variabili e crescente complessità.

Indice

| | | |
|----------|---|-----------|
| 1 | Introduzione al progetto | 1 |
| 1.1 | Descrizione generale | 1 |
| 1.1.1 | Agente | 1 |
| 1.1.2 | Labirinti | 2 |
| 1.1.3 | Ambienti di allenamento | 3 |
| 1.1.4 | Menù di simulazione | 4 |
| 1.1.5 | Benchmark del modello | 4 |
| 1.2 | Obiettivi del progetto | 5 |
| 1.2.1 | Framework | 5 |
| 1.2.2 | Agente | 5 |
| 1.3 | Reinforcement Learning per la risoluzione di labirinti 3D | 5 |
| 2 | Fondamenti teorici | 7 |
| 2.1 | Introduzione al Reinforcement Learning | 7 |
| 2.2 | Componenti principali | 7 |
| 2.2.1 | Esempi | 8 |
| 2.3 | Principi del Reinforcement Learning | 8 |
| 2.3.1 | Ottimizzazione della policy | 9 |
| 2.3.2 | Curriculum Learning | 9 |
| 2.4 | Machine Learning in Unity | 10 |
| 2.4.1 | Panoramica sul toolkit | 10 |
| 2.4.2 | Training Configuration File | 11 |
| 3 | Set-up del progetto | 12 |
| 3.1 | Preparazione dell'ambiente | 12 |
| 3.1.1 | Versioni, dipendenze, librerie utilizzate | 12 |
| 3.1.2 | Struttura generale del progetto | 13 |
| 3.2 | Generazione dei labirinti 3D | 13 |
| 3.2.1 | Architettura del labirinto | 15 |
| 3.2.2 | Algoritmo Recursive Backtracking | 16 |
| 3.3 | Creazione dell'agente | 16 |
| 3.3.1 | CubeTraining e CubeTesting | 17 |
| 3.3.2 | Behaviour Parameters | 18 |
| 3.3.3 | Decision Requester | 20 |
| 3.3.4 | Spazio delle azioni | 20 |
| 3.3.5 | Spazio delle osservazioni | 21 |
| 3.3.5.1 | RayPerceptionSensorComponent3D | 21 |
| 3.3.5.2 | Stato delle celle | 22 |
| 3.4 | Ambienti di allenamento | 23 |

| | | |
|--------------------|--|-----------|
| 3.4.1 | Randomizzazione dell'ambiente | 24 |
| 3.4.2 | Randomizzazione dell'agente | 24 |
| 3.4.3 | Corridoio | 24 |
| 3.4.4 | Corridoio con ostacolo | 25 |
| 3.4.5 | Labirinti NxN | 26 |
| 3.5 | Implementazione del reward system | 26 |
| 3.5.1 | Reward per il movimento | 27 |
| 3.5.2 | Reward per l'esplorazione | 27 |
| 3.5.3 | Reward per il superamento degli ostacoli | 27 |
| 3.6 | Menù di simulazione | 28 |
| 3.7 | Integrazione e usabilità | 28 |
| 3.7.1 | Modifica dei valori del reward system | 29 |
| 3.7.2 | Tool di generazione di ambienti per l'allenamento | 29 |
| 3.7.3 | Raccolta dei dati per l'analisi del modello | 30 |
| 4 | Assets 3D | 31 |
| 4.1 | Creazione di assets per Unity | 31 |
| 4.2 | Creazione dei modelli per i labirinti | 32 |
| 4.3 | Creazione del modello 3D dell'Agente | 34 |
| 5 | Allenamento | 35 |
| 5.1 | Parametri di training | 35 |
| 5.1.1 | Configurazione del Training File | 35 |
| 5.1.1.1 | Iperparametri comuni | 35 |
| 5.1.1.2 | Setting della rete neurale | 36 |
| 5.1.1.3 | Reward estrinseco e intrinseco | 37 |
| 5.1.2 | Reward System | 37 |
| 5.1.3 | Imparare a muoversi | 37 |
| 5.1.4 | Imparare ad esplorare | 38 |
| 5.1.5 | Imparare a superare gli ostacoli | 38 |
| 5.2 | Fasi del Curriculum Learning | 39 |
| 5.2.1 | Primo allenamento: corridoio | 39 |
| 5.2.2 | Secondo allenamento: corridoio con ostacolo | 41 |
| 5.2.3 | Terzo allenamento: labirinti 5x5 con 5 ostacoli | 43 |
| 6 | Risultati | 45 |
| 6.1 | Analisi della bontà del modello | 45 |
| 6.1.1 | Esplorazione del modello | 46 |
| 6.1.2 | Superamento degli ostacoli e utilizzo del salto | 49 |
| 6.1.3 | Generalizzazione del modello | 51 |
| 7 | Conclusioni e sviluppi futuri | 53 |
| 7.1 | Conclusioni | 53 |
| 7.2 | Sviluppi futuri | 53 |
| 7.2.1 | Bilanciamento tra esplorazione e raggiungimento dell'obiettivo | 54 |
| 7.2.2 | Miglioramento della memoria dell'agente | 54 |
| Riferimenti | | 56 |
| Siti | | 56 |

Elenco delle figure

| | | |
|------|--|----|
| 1.1 | Una volta allenato l'agente, è possibile testarlo in labirinti diversi | 1 |
| 1.2 | Agente all'interno di un labirinto | 2 |
| 1.3 | Esempio di labirinti 3x3, 5x5, 7x7, 10x10 | 2 |
| 1.4 | Labirinto 10x10 con 10 ostacoli | 3 |
| 1.5 | Alcuni labirinti utilizzati negli ambienti di allenamento | 3 |
| 1.6 | L'agente all'interno del labirinto esplora i diversi percorsi | 4 |
| 2.1 | Schema del ciclo di interazione agente-ambiente | 8 |
| 2.2 | Esempi di ambienti di allenamento su Unity | 11 |
| 3.1 | Interfaccia per la generazione di labirinti | 14 |
| 3.2 | Interfaccia MazeGeneratorTesting | 15 |
| 3.3 | Gerarchia degli oggetti nel labirinto generato in Unity. | 16 |
| 3.4 | Editor per la modifica dei parametri dell'agente | 18 |
| 3.5 | Esempio di Behaviour Parameters utilizzato | 19 |
| 3.6 | Decision Requester | 20 |
| 3.7 | FloorSensor | 22 |
| 3.8 | WallTargetSensor | 22 |
| 3.9 | In blu <i>FloorSensor</i> , in rosso/bianco <i>WallTargetSensor</i> | 22 |
| 3.10 | Configurazione corrente | 23 |
| 3.11 | Agente durante la prima fase di allenamento | 25 |
| 3.12 | L'agente supera l'ostacolo e va verso il target | 25 |
| 3.13 | Serie di labirinti 5x5, con 5 ostacoli, sempre randomizzati | 26 |
| 3.14 | L'agente ha raggiunto il target in un labirinto 10x10 con 8 ostacoli | 28 |
| 3.15 | Editor per la modifica dei parametri dell'agente | 29 |
| 3.16 | Esempio di istanze di labirinti diversi generati con il tool | 30 |
| 3.17 | Esempio di dati raccolti con la class BenchmarkLogger | 30 |
| 4.1 | Assets utilizzati per la generazione dei labirinti | 33 |
| 4.2 | Esempio di labirinto 7x7 generato con i nuovi assets | 33 |
| 4.3 | Asset 3D dell'agente | 34 |
| 4.4 | Target raggiunto dall'agente | 34 |
| 4.5 | Agente mentre esplora un labirinto | 34 |
| 5.1 | Reward cumulativo del primo allenamento | 40 |
| 5.2 | Value Loss del primo allenamento | 40 |
| 5.3 | Entropia del primo allenamento | 41 |
| 5.4 | Reward cumulativo della seconda fase di allenamento | 42 |
| 5.5 | Value loss della seconda fase di allenamento | 42 |
| 5.6 | Entropia della seconda fase di allenamento | 42 |

| | | |
|-----|---|----|
| 5.7 | Reward cumulativo nella terza fase di allenamento | 43 |
| 5.8 | Value loss nella terza fase di allenamento | 43 |
| 5.9 | Entropia nella terza fase di allenamento | 44 |
| 6.1 | Legenda boxplot | 47 |
| 6.2 | Boxplot nei casi di successo/insuccesso labirinti 5x5 | 48 |
| 6.3 | Boxplot nei casi di successo/insuccesso labirinti 7x7 | 48 |
| 6.4 | Boxplot nei casi di successo/insuccesso labirinti 10x10 | 48 |
| 6.5 | Diagrammi a torta tra la relazione dei successi e il numero di celle rivisitate | 49 |
| 6.6 | Relazione tra reward e numero di salti nei labirinti 5x5 | 50 |
| 6.7 | Relazione tra reward e numero di salti nei labirinti 7x7 | 50 |
| 6.8 | Relazione tra reward e numero di salti nei labirinti 10x10 | 51 |
| 6.9 | Diagrammi a torta tra il numero di casi di insuccesso e successo | 52 |
| 7.1 | Confronto tra diverse configurazioni di percezione spaziale | 54 |

Elenco delle tabelle

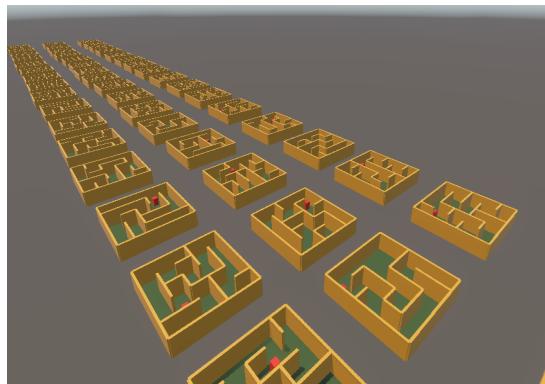
| | | |
|-----|---|----|
| 3.1 | Spazio delle osservazioni della configurazione | 23 |
| 5.1 | Reward utilizzati per l'agente durante l'episodio | 37 |
| 5.2 | Reward utilizzati per l'esplorazione | 38 |
| 5.3 | Reward utilizzati per il superamento degli ostacoli | 39 |
| 6.1 | Metriche medie in caso di successo | 45 |
| 6.2 | Metriche medie in caso di insuccesso | 46 |
| 6.3 | Metriche delle performance complessive | 46 |
| 6.4 | Correlazioni tra numero di steps e celle visitate (univoche / rivisitate), nei casi di successo e insuccesso. | 46 |

1

Introduzione al progetto

1.1 Descrizione generale

Il progetto consiste nello sviluppo di un framework in Unity finalizzato alla creazione e all'addestramento di agenti intelligenti tramite Reinforcement Learning, con l'obiettivo di risolvere labirinti tridimensionali. Il framework fornisce ambienti personalizzabili per l'allenamento dell'agente e un menù di simulazione utile per valutare le capacità del modello una volta completata la fase di apprendimento.



(a) Ambiente di allenamento 5x5



(b) Menù di simulazione

Figura 1.1: Una volta allenato l'agente, è possibile testarlo in labirinti diversi

1.1.1 Agente

L'agente è modellato come un parallelepipedo in grado di muoversi nelle **quattro direzioni, eseguire salti e combinare tali azioni** per navigare all'interno del labirinto. L'interazione con l'ambiente avviene attraverso una serie di **sensori**, progettati per raccogliere informazioni rilevanti sullo stato circostante, successivamente utilizzate nel processo di apprendimento. Il comportamento dell'agente è parametrizzabile: l'utente può modificare caratteristiche come la velocità di movimento, l'intensità dei reward e

delle penalità, nonché altre variabili legate alla dinamica decisionale, tramite un'apposita interfaccia grafica integrata nel framework.

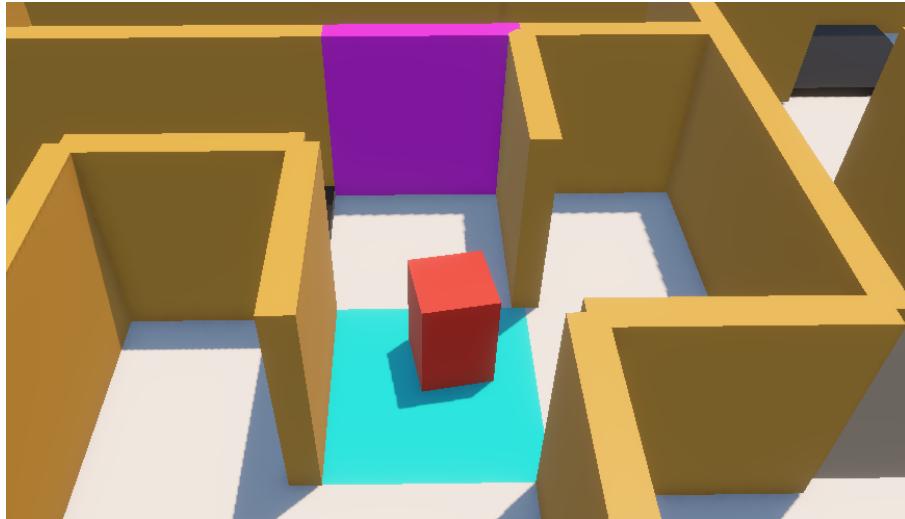


Figura 1.2: Agente all'interno di un labirinto

1.1.2 Labirinti

I labirinti costituiscono l'ambiente principale per l'allenamento e il testing dell'agente. Essi vengono *generati proceduralmente* tramite uno script dedicato (vedi 3.2), che consente di variarne le dimensioni. Una volta generato, ogni labirinto può essere salvato per essere riutilizzato nelle successive fasi di addestramento. Gli **ostacoli** e il **target** (uscita) vengono inseriti successivamente, in modo da rendere possibile il riutilizzo dello stesso layout con configurazioni differenti. Il target può essere in qualsiasi posizione all'interno del labirinto, e di conseguenza, è possibile raggiungerlo da al massimo 2 direzioni diverse. All'interno del labirinto sono previsti due tipi principali di ostacoli: **blocchi in rilievo** e **aperture nel pavimento**. Questi rispettano due condizioni:

- Due ostacoli non possono essere adiacenti.
- L'ostacolo non può essere posizionato alla fine di un vicolo cieco.

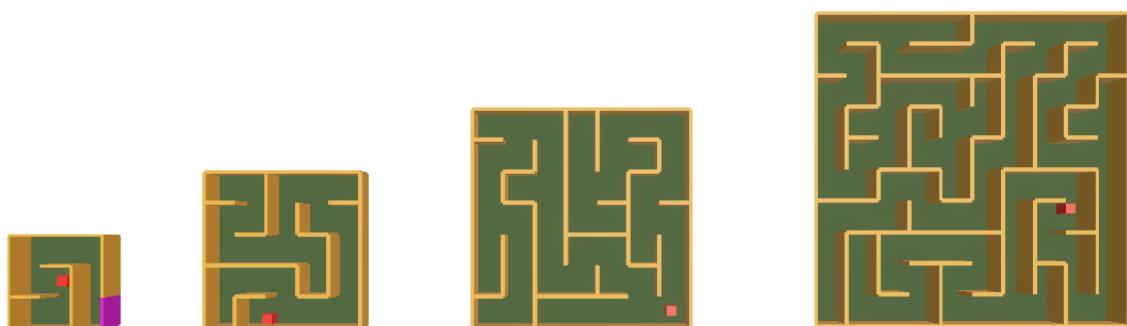


Figura 1.3: Esempio di labirinti 3x3, 5x5, 7x7, 10x10

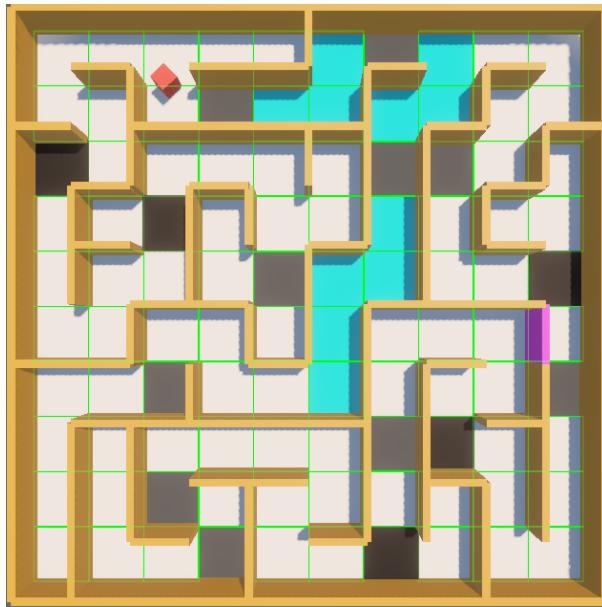


Figura 1.4: Labirinto 10x10 con 10 ostacoli

1.1.3 Ambienti di allenamento

Al fine dell’allenamento del modello, sono necessari diversi ambienti dove migliorare progressivamente la capacità dell’agente di risolvere labirinti. Sono presenti 4 scene, con labirinti gradualmente più complessi:

1. Corridoio senza ostacoli.
2. Corridoio con un ostacolo.
3. Serie di labirinti 5x5 con 5 ostacoli.
4. Serie di labirinti 7x7 con 7 ostacoli.

L’utente, oltre a poter utilizzare le scene già presenti, potrà crearne di nuove con l’apposito editor, per allenare l’agente in ambienti a suo piacimento (vedi 3.7.2).

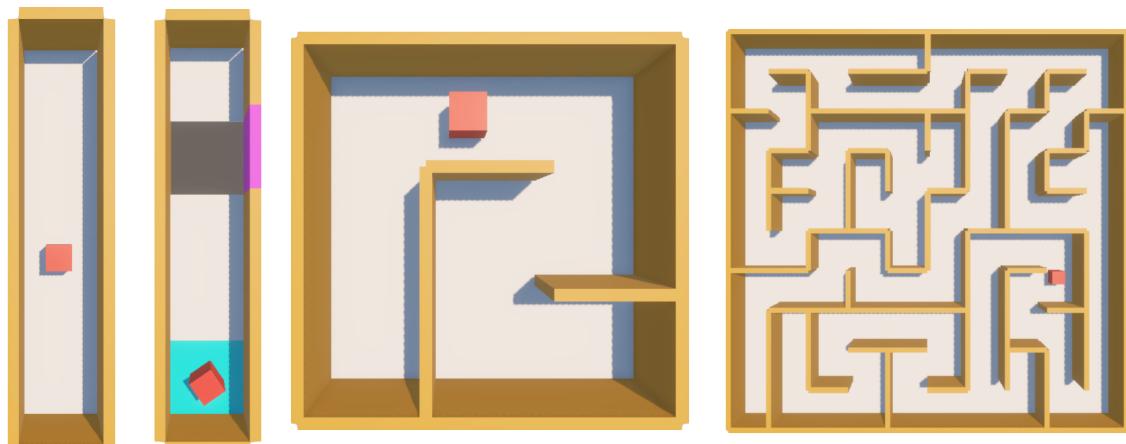


Figura 1.5: Alcuni labirinti utilizzati negli ambienti di allenamento

1.1.4 Menù di simulazione

Il menù di simulazione permette di visualizzare l'esecuzione del modello nella risoluzione di labirinti sempre diversi. L'utente può generare labirinti di diverse dimensioni (5x5, 7x7, 10x10, 15x15) popolati da ostacoli in quantità variabile. La posizione di questi, dell'agente e del target all'interno dei labirinti è sempre casuale. La simulazione non verrà svolta utilizzando le semplici primitive 3D native di Unity impiegate durante l'allenamento, ma si effettuerà con nuovi modelli 3D creati appositamente su *Blender*.



Figura 1.6: L'agente all'interno del labirinto esplora i diversi percorsi

1.1.5 Benchmark del modello

Per valutare le prestazioni di un modello già allenato, i dati relativi al comportamento dell'agente all'interno del labirinto vengono raccolti automaticamente tramite una classe di benchmark dedicata. Durante l'esecuzione, tutti i dati rilevanti vengono salvati in un file .csv nella cartella **Benchmarks**, da cui è possibile estrarre statistiche e trarre conclusioni sull'efficacia del modello. Vengono raccolti i seguenti dati, per ogni episodio:

- successo nella risoluzione del labirinto;
- totale di steps;
- totale di celle visitate;
- totale di celle **univoche visitate**;
- totale di celle **visitate più di 2 volte**;
- reward cumulativo;
- numero di salti.

1.2 Obiettivi del progetto

Gli obiettivi si dividono in due aree principali: la **flessibilità del framework** e il **comportamento dell'agente**, con attenzione non solo al raggiungimento dell'uscita, ma anche alla serie di sequenze che portano a questo risultato (es. l'agente è in grado di muoversi in avanti nel modo corretto, evitando di muoversi saltando).

1.2.1 Framework

L'ambiente deve permettere all'utente di:

1. Generare facilmente ambienti di allenamento diversi, con labirinti variabili per complessità e struttura, in modo tale da creare un modello in grado di generalizzare la risoluzione di qualsiasi labirinto.
2. Modificare a piacimento il sistema di ricompense dell'agente, così da poter esplorare differenti strategie di apprendimento e comportamenti diversi.
3. Sostituire facilmente il modello 3D dell'agente e dei labirinti.
4. Salvare e caricare configurazioni di labirinti e agente, per facilitare test ripetibili.
5. Raccogliere i dati relativi al comportamento di un modello nei diversi ambienti, per poterne valutare la bontà.

1.2.2 Agente

L'agente deve simulare il comportamento di una persona che risolve un labirinto inesplorato, di cui non sa la posizione dell'uscita. Deve essere in grado di:

1. Navigare efficacemente all'interno del labirinto.
2. Memorizzare le strade già percorse.
3. Evitare percorsi con vicoli ciechi già visitati in precedenza.
4. Raggiungere il target se nella sua area visibile.
5. Superare eventuali ostacoli lungo il percorso.

1.3 Reinforcement Learning per la risoluzione di labirinti 3D

Esistono numerosi algoritmi in grado di risolvere labirinti complessi, la cui efficacia varia a seconda del grado di osservabilità del labirinto. Il miglior algoritmo in termini di performance è **A***.

A* è una tecnica di ricerca **euristica** utilizzata per trovare il percorso ottimale in un grafo o uno spazio di ricerca. È particolarmente efficace per problemi come la ricerca del *percorso più breve* (ad esempio nei giochi, nella robotica o nei sistemi di navigazione) e combina le caratteristiche della ricerca in ampiezza (Breadth-First Search) e della

ricerca costo-uniforme (Uniform-Cost Search). Questo metodo, pur essendo efficiente e deterministico, richiede una **conoscenza esplicita e completa dell'ambiente** e non si adatta bene a scenari dinamici o parzialmente osservabili [1]. In questo caso il reinforcement learning offre un approccio più flessibile e vantaggioso quando si desidera costruire agenti in ambienti sconosciuti o complessi, a causa della presenza di ostacoli nel percorso. Inoltre, può generalizzare a nuovi ambienti dopo essere stato allenato su una varietà di situazioni.

2

Fondamenti teorici

Allenare un modello per simulare un comportamento richiede conoscenze di base in diversi campi. A partire dal funzionamento di una semplice rete neurale, è necessario capire come funziona il Reinforcement Learning e perché funziona nel suo intento, scegliendo nel modo più corretto parametri e informazioni da raccogliere per l'allenamento.

2.1 Introduzione al Reinforcement Learning

Il Reinforcement Learning è una branca del machine learning che studia come gli agenti intelligenti siano in grado di prendere decisioni tramite un approccio "trial-and-error" per massimizzare un reward cumulativo. L'agente, interagendo in un ambiente, riceve un feedback positivo o negativo in base all'azione presa, imparando così a prendere azioni che portano verso un reward cumulativo più alto, e quindi, verso l'obiettivo [17].

2.2 Componenti principali

Nel Reinforcement Learning, esistono alcuni elementi fondamentali che ne definiscono il funzionamento [4] [17]:

- **Agente:** è l'entità che apprende tramite l'interazione con l'ambiente. Può trattarsi, ad esempio, di un robot, un programma software o un algoritmo.
- **Ambiente:** rappresenta il contesto fisico o simulato in cui l'agente si muove e prende decisioni. È il mondo con cui l'agente interagisce.
- **Stato:** descrive la situazione attuale dell'ambiente, così come percepita dall'agente. È la base su cui l'agente prende decisioni.
- **Azione:** è una delle possibili scelte che l'agente può compiere in un determinato stato dell'ambiente. Le azioni influenzano l'evoluzione dello stato.

- **Ricompensa:** è un valore numerico che l'agente riceve come feedback dopo aver compiuto un'azione. Indica quanto l'azione è stata utile per il raggiungimento dell'obiettivo.

2.2.1 Esempi

- Un robot aspirapolvere (**agente**) si muove all'interno di una casa (**ambiente**) per pulire il pavimento. Usa informazioni sulla posizione, ostacoli e aree pulite (**stato**) per decidere se andare avanti, girare o tornare alla base (**azione**). Riceve punti quando pulisce nuove zone e penalità se urta mobili (**reward**).
- Un drone (**agente**) vola sopra una città (**ambiente**) per consegnare pacchi. Valuta la propria posizione, il pacco e la meta' (**stato**), poi sceglie se salire, scendere o cambiare direzione (**azione**). Se si avvicina alla destinazione o completa la consegna viene premiato, mentre se si schianta o sbaglia rotta viene punito (**reward**).

2.3 Principi del Reinforcement Learning

Il *Reinforcement Learning* è un paradigma basato sul modello matematico del *Markov Decision Process* (MDP) [6]. Questo modello descrive problemi decisionali in cui un agente interagisce con un *ambiente variabile* nel tempo, suddiviso in intervalli discreti chiamati *step*. Ad ogni step, l'agente osserva lo stato attuale dell'ambiente, seleziona un'azione da compiere e riceve in risposta una nuova osservazione dello stato e un segnale di ricompensa (reward), che riflette la correttezza dell'azione eseguita. L'ambiente quindi evolve in base all'azione intrapresa, e il processo continua iterativamente.

Attraverso il meccanismo di *trial-and-error*, l'agente apprende una strategia di comportamento ottimale, detta **policy**, che associa ad ogni stato **l'azione migliore da compiere per massimizzare il reward cumulativo** nel tempo.

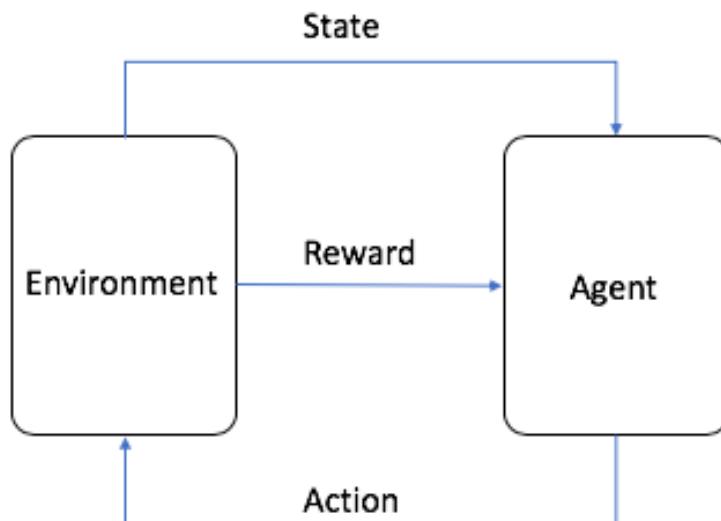


Figura 2.1: Schema del ciclo di interazione agente-ambiente

2.3.1 Ottimizzazione della policy

Nel reinforcement learning, gli approcci per apprendere una **policy ottimale**, ovvero la strategia che massimizza la *ricompensa cumulativa*, possono essere suddivisi in due categorie principali [2]:

- **Policy-Based Methods:** si basano sull'apprendimento diretto della policy, cioè della strategia che l'agente utilizza per scegliere le azioni. Questi metodi non stimano esplicitamente il valore degli stati o delle azioni, ma ottimizzano direttamente i parametri della policy tramite tecniche come la *discesa del gradiente*.
- **Value-Based Methods:** si fondano sull'apprendimento di una funzione di valore, che stima quanto sia vantaggioso trovarsi in uno stato (o eseguire una certa azione). L'agente utilizza questa funzione per scegliere azioni che massimizzano il valore atteso della ricompensa futura.

Nel caso specifico della risoluzione dei labirinti, il modello è stato allenato utilizzando l'algoritmo *Proximal Policy Optimization* (PPO), un metodo *policy-based* sviluppato da *OpenAI* [7]. PPO ottimizza direttamente la policy dell'agente attraverso il gradiente, ma introduce un meccanismo di *clipping* per limitare gli aggiornamenti troppo bruschi rispetto alla policy precedente.

2.3.2 Curriculum Learning

L'agente può incontrare difficoltà nell'apprendere una policy efficace se viene esposto fin da subito ad un ambiente eccessivamente complesso: più elevato è il livello di difficoltà del compito, maggiori saranno le sfide che l'agente dovrà affrontare per apprendere una strategia efficace. Le principali difficoltà che si possono incontrare durante l'addestramento sono:

- **Ricompense sparse:** il feedback fornito dall'ambiente è minimo, quindi l'agente può compiere numerose azioni senza ricevere alcuna indicazione sull'efficacia del proprio comportamento. Ad esempio, in un labirinto, l'agente potrebbe dover esplorare a lungo senza ricevere alcuna ricompensa, ottenendola solo al momento dell'uscita.
- **Tempi di allenamento lunghi:** nei compiti complessi, l'agente deve esplorare un vasto insieme di combinazioni di azioni, procedendo per tentativi casuali. Questo porta a tempi di convergenza elevati durante l'allenamento.

Per risolvere queste problematiche, viene introdotto il paradigma del **Curriculum Learning**. L'idea è quella di adottare un approccio progressivo nell'allenamento [11]: l'agente viene inizialmente introdotto in una versione semplificata dell'ambiente, per poi affrontare, in modo **graduale**, scenari sempre più difficili. Il concetto di "difficoltà" è definito in funzione dell'obiettivo specifico da raggiungere e varia a seconda del contesto di apprendimento. Questa idea è simile al modo in cui l'uomo impara qualcosa di nuovo, passo dopo passo. Si possono identificare diversi vantaggi, tra cui:

- **Convergenza più rapida:** iniziando con compiti più semplici, l'agente apprende più velocemente i concetti di base, costruendo gradualmente una strategia utilizzabile per affrontare, successivamente, ambienti più complessi.

- **Migliore generalizzazione:** affrontando una varietà di situazioni più semplici, l'agente diventa più adattabile anche a scenari mai visti prima.
- **Gestione semplice di compiti complessi:** compiti molto articolati possono risultare troppo difficili se affrontati direttamente. Suddividendoli in *sotto-task* semplificati, l'agente riesce a imparare in modo incrementale, affrontando la complessità passo dopo passo.

2.4 Machine Learning in Unity

Il machine learning è un campo molto utilizzato in Unity per quanto riguarda l'allenamento di comportamenti di personaggi nei videogiochi. Un caso d'uso ricorrente è quello dell'allenamento di **non-playable characters** (NPCs) per l'adattamento dinamico nei vari ambienti [14].

Il toolkit **ML-Agents** consente di addestrare agenti intelligenti tramite tecniche di *Machine Learning*. Gli agenti operano all'interno di ambienti di simulazione 2D o 3D di Unity, imparando a risolvere specifici *task* mediante approcci di *Reinforcement Learning*. Il toolkit rappresenta quindi un **ponte tra lo sviluppo di videogiochi e il machine learning**, permettendo agli agenti di interagire con l'ambiente virtuale, raccogliere dati e migliorare progressivamente il proprio comportamento. ML-Agents supporta una varietà di metodologie di apprendimento, tra cui il *Proximal Policy Optimization* (PPO), il *Soft Actor-Critic* (SAC) e l'*Imitation Learning*, rendendolo uno strumento flessibile per tutti i tipi di allenamento.

2.4.1 Panoramica sul toolkit

Una volta costruita una *scena*, è possibile convertirla in un ambiente di allenamento per l'agente, grazie alle seguenti componenti e classi del *toolkit*:

- **Behavior Parameters:** insieme di parametri che definiscono l'interazione tra l'agente e l'ambiente di allenamento.
- **Decision Requester:** componente che regola ogni quanto tempo l'agente deve prendere una decisione e richiedere un'azione.
- **Sensors:** raccolgono informazioni dall'ambiente e le trasformano in osservazioni che l'agente può utilizzare. `RayPerceptionSensorComponent3D`, `VectorSensor` e `CameraSensorComponent` sono alcuni dei sensori presenti, ognuno con uno scopo diverso.
- **Classe Agent:** rappresenta la classe base da estendere per definire il comportamento personalizzato dell'agente all'interno dell'ambiente. È fondamentale per implementare la logica dell'agente e il sistema di ricompense.
- **Academy:** componente globale che controlla la simulazione e coordina l'inizio/fine degli episodi, la scala temporale e l'interazione con Python per l'addestramento.

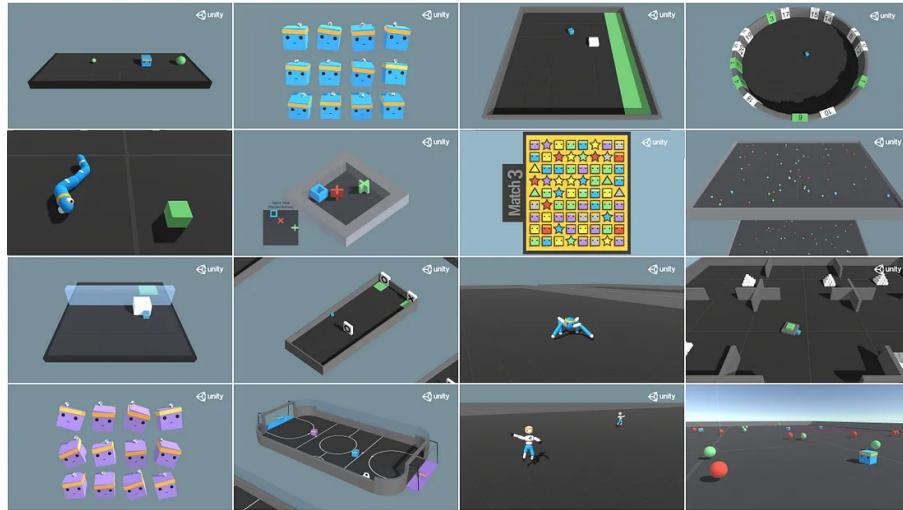


Figura 2.2: Esempi di ambienti di allenamento su Unity

2.4.2 Training Configuration File

Il *Training Configuration File* è il file per la definizione dei parametri di addestramento di un agente. Specifica i metodi di allenamento, gli iperparametri e valori aggiunti da usare durante le fasi di allenamento. È diviso in diverse sezioni [15]:

- **trainer_type**: definisce l'algoritmo di reinforcement learning utilizzato per l'addestramento, ad esempio `ppo` o `sac`.
- **hyperparameters**: comprende i parametri di ottimizzazione dell'apprendimento, come:
 - `batch_size`: dimensione dei batch per ogni aggiornamento.
 - `buffer_size`: dimensione del buffer di esperienze.
 - `learning_rate`: tasso di apprendimento del modello.
- **network_settings**: specifica l'architettura della rete neurale, tra cui:
 - `hidden_units`: numero di neuroni per livello nascosto.
 - `num_layers`: numero di layer della rete.
 - `normalize`: normalizzazione delle osservazioni.
- **max_steps**: determina il numero massimo di passi di addestramento.
- **summary_freq**: indica ogni quanti passi vengono registrate le metriche di training.

I parametri configurabili sono numerosi e permettono una personalizzazione dettagliata dell'addestramento. È possibile specificare impostazioni dedicate per uno specifico *trainer* (come PPO o SAC), per i segnali di ricompensa *intrinseca* ed *estrinseca*, nonché per le caratteristiche della memoria dell'agente.

3

Set-up del progetto

In questo capitolo viene descritta la creazione dell’ambiente di sviluppo, includendo la struttura generale e le componenti tecniche utilizzate. Viene mostrato il sistema di generazione dei labirinti 3D, la progettazione dell’agente intelligente e gli ambienti impiegati per l’allenamento. Si analizza nel dettaglio il sistema di ricompense adottato per guidare l’apprendimento dell’agente e si illustrano le modalità di simulazione del modello una volta addestrato.

3.1 Preparazione dell’ambiente

Il progetto è strutturato in modo da permettere all’utente di utilizzare il framework comodamente, in base ai suoi bisogni. È prima necessario definire le dipendenze e le librerie utilizzate, che andranno installate prima di procedere con l’utilizzo.

3.1.1 Versioni, dipendenze, librerie utilizzate

La versione di Unity utilizzata è la versione 6.0 (6000.0.16f1). È necessario installare le seguenti versioni dei package, tramite la finestra integrata Package Manager:

- `com.unity.ml-agents` (3.0.0)
- `com.unity.probuilder` (6.0.5)
- `com.unity.ugui` (2.0.0)
- `com.unity.render-pipelines.universal` (17.0.3)
- `com.unity.inputsystem` (1.9.0)

Per mantenere l’ambiente pulito e indipendente da altre installazioni, è stato creato un ambiente virtuale tramite il modulo `venv`. Una volta attivato l’ambiente virtuale, si è proceduto con l’installazione di PyTorch [16], per la costruzione di reti neurali a partire dalle informazioni raccolte grazie alle classi apposite del toolkit.

3.1.2 Struttura generale del progetto

Il progetto è suddiviso in 7 directory principali, ognuna per un aspetto specifico del progetto:

- **Scripts:** contiene gli script in C# basati sulla classe `MonoBehaviour`, responsabili della logica del progetto. In particolare:
 - `CubeTesting` e `CubeTraining` gestiscono il comportamento dell’agente e l’interazione con ML-Agents.
 - `MazeController`, `MazeGeneratorTraining` e `MazeGeneratorTesting` si occupano della generazione procedurale dei labirinti e del loro controllo logico per la raccolta delle osservazioni per la rete neurale.
 - `AgentState`, `OptionController`, `ZoomToScrool`, `MainMenuController` gestiscono l’interfaccia grafica e l’interazione con il menù di simulazione.
 - `BenchmarkLogger` si occupa di raccogliere e salvare i dati relativi ad un allenamento e simulazione, per poter valutare la bontà del modello.
- **Scenes:** contiene le scene di Unity utilizzate per le diverse fasi del progetto, in particolare per l’allenamento e il testing dell’agente. I nomi delle scene sono autoesplicativi e riflettono la tipologia di ambiente simulato (es. `ReachTarget`, `SolveMaze5x5`, `JumpObstacles`).
- **Editor:** contiene gli script in C# basati sulle classi `EditorWindow` e `Editor`, per la modifica del reward system dell’agente (`RewardSettingsEditor`), la creazione di ambienti d’allenamento personalizzati (`MulitpleMazeSpawner`) e la generazione di labirinti (`MazeGeneratorEditor`).
- **Material:** contiene i materiali utilizzati per i modelli 3D utilizzati in fase di training.
- **Models Blender:** contiene i modelli 3D e materiali per il testing del modello una volta allenato, divisi in sottocartelle.
- **Models:** cartella contenente il modello allenato, utilizzato di default per il testing.
- **Prefab:** contiene i *prefab*, i modelli riutilizzabili di un `GameObject` [5], per la creazione delle scene, sia di allenamento che simulazione.

3.2 Generazione dei labirinti 3D

Generare un labirinto 3D è un’operazione necessaria al fine del progetto, poiché creare manualmente un labirinto, anche non complesso, è un’operazione che richiederebbe tempo. Dal punto di vista implementativo bisogna definire da quali componenti 3D è composto e come è possibile generarne sempre diversi. A tal scopo, sono stati sviluppati due script dedicati, uno per la fase di allenamento e una per la simulazione a modello allenato, che implementano l’algoritmo **Recursive Backtracker** [18] per la generazione.

`MazeGeneratorTraining` è lo script dedicato alla generazione di labirinti per la fase di allenamento. In una scena, si crea un `GameObject` vuoto e aggiungere lo script come

componente. Selezionando l'oggetto, apparirà un *pannello* da cui è possibile impostare altezza, larghezza, dimensione delle celle, il prefab della cella per il pavimento e il prefab delle mura. Premendo il pulsante "Generate Maze", il labirinto verrà generato e automaticamente salvato nella cartella **Assets/GeneratedMazes**. In questa fase non vengono generati né gli ostacoli, né l'uscita, né il target: questi elementi verranno inseriti solo successivamente durante la fase di allenamento vero e proprio, in modo tale da poter usare uno stesso labirinto ma con configurazione diverse.

Una volta generato il labirinto, è necessario posizionare al suo interno il prefab dell'agente, chiamato *Agent*, situato nella cartella **Training Assets**. A questo punto, bisogna compilare i campi pubblici dell'agente assegnando i riferimenti corretti al labirinto e agli altri oggetti della scena (vedi figura 3.15).

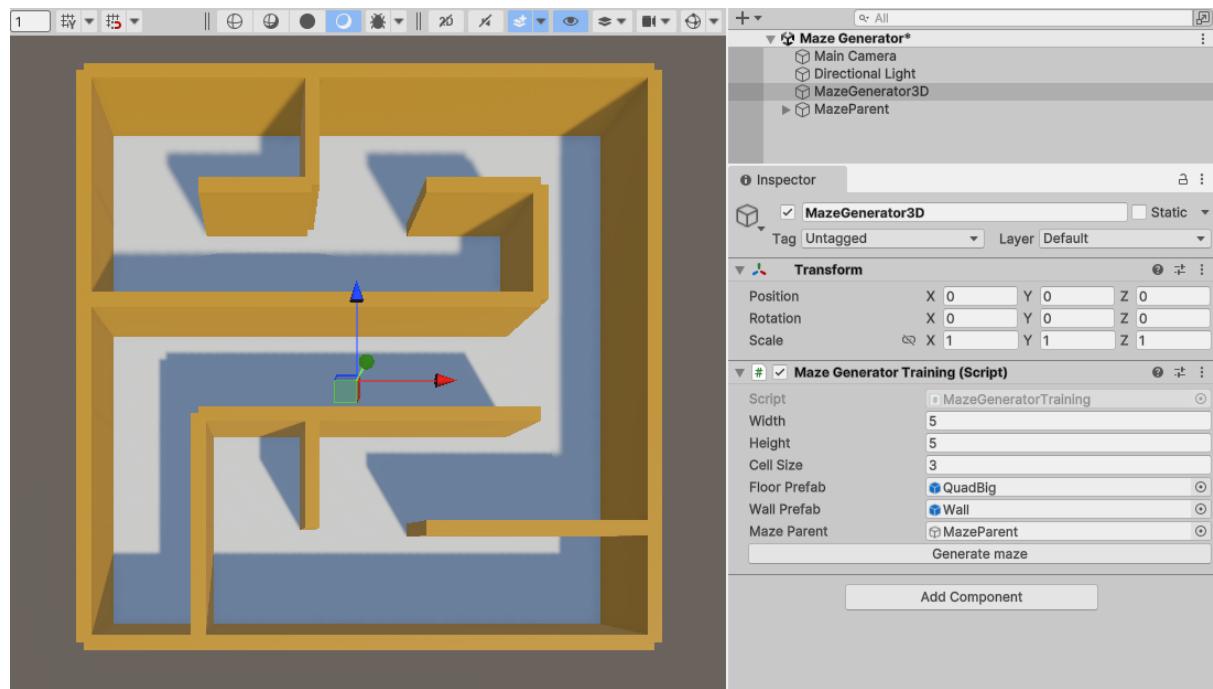


Figura 3.1: Interfaccia per la generazione di labirinti

MazeGeneratorTesting, invece, viene utilizzato esclusivamente nella fase di simulazione. La generazione del labirinto è controllata tramite un menù apposito presente nella scena **Main**. Rispetto alla versione per l'allenamento, questo script usa *modelli 3D* per la generazione differenti (vedi 4.2) e **posiziona automaticamente** all'interno del labirinto l'agente, il target e gli ostacoli (se previsti), così da permettere di testare direttamente l'agente in ambienti completi.

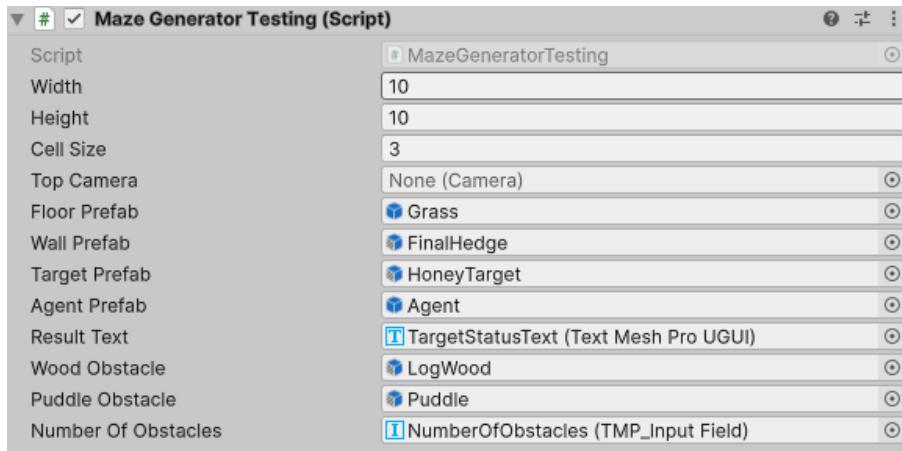


Figura 3.2: Interfaccia MazeGeneratorTesting

3.2.1 Architettura del labirinto

Un labirinto generico è composto da 4 elementi principali: mura perimetrali e interne, pavimento, un'uscita (alias *target*) ed optionalmente ostacoli di due tipi (buco nel pavimento e blocco rialzato da superare). Per facilitarne l'uso nei diversi labirinti, per ogni componente è stato creato un prefab riutilizzabile, nella cartella **Prefab**. I 3 prefab (ostacoli esclusi) sono stati forniti di un **tag** distinto: "Wall", "Floor", "Target".

Il pavimento è suddiviso in una griglia $n \times n$ di blocchi (alias *celle*) per facilitare la raccolta di informazioni per l'allenamento e la logica d'implementazione. Per visualizzare meglio la posizione dell'agente, ogni volta che l'agente visita una cella, questa assume una densità di colore sempre più scura in base al numero di volte in cui è stata visitata (in fase di training), mentre in fase di simulazione se una cella viene visitata, verrà inserito il modello 3D di **un fiore** sopra ad essa (vedi 4.1c).

Dal punto di vista strutturale, ogni labirinto generato è organizzato gerarchicamente sotto un oggetto principale chiamato **MazeParent**. Questo oggetto agisce da contenitore per tre GameObject secondari:

- **WallParent**, che contiene tutte le mura del labirinto.
- **FloorParent**, che contiene le celle che compongono il pavimento.
- **ObstaclesParent**, che ospita eventuali ostacoli.

Questa suddivisione è utile non solo per motivi organizzativi e di leggibilità all'interno della gerarchia della scena Unity, ma anche per semplificare operazioni successive di distruzione e rigenerazione del labirinto durante le diverse fasi di addestramento e simulazione.

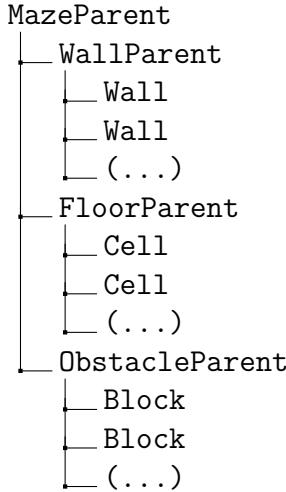


Figura 3.3: Gerarchia degli oggetti nel labirinto generato in Unity.

3.2.2 Algoritmo Recursive Backtracking

Questo algoritmo per la generazione produce labirinti con una struttura ramificata ma connessa, senza loop, che garantisce un percorso unico tra ogni coppia di punti. L'elemento di casualità nella scelta delle celle adiacenti genera strutture variabili e complesse, mentre l'uso del backtracking garantisce che tutte le celle siano raggiunte [18]. L'implementazione ricorsiva è concettualmente semplice ma può risultare meno scalabile rispetto alla versione iterativa, specialmente in ambienti a bassa profondità di stack. E' basato sull'algoritmo di ricerca in profondità DFS e viene implementato nel seguente modo:

Algorithm 1 Recursive Backtracking

Require: Cella corrente C

- 1: Contrassegna C come *visitata*
 - 2: **while** C ha celle adiacenti non visitate **do**
 - 3: Scegli casualmente una cella vicina non visitata N
 - 4: Rimuovi il muro tra C e N nella struttura dati del labirinto
 - 5: GeneraLabirinto N
 - 6: **end while**
-

3.3 Creazione dell'agente

Per permettere all'agente di muoversi e apprendere all'interno dei vari labirinti, è necessario definire tutte le azioni che può compiere e il modo in cui percepisce l'ambiente circostante (osservazioni). Questo processo avviene attraverso la configurazione dell'agente tramite la classe **Agent** fornita da Unity ML-Agents, che deve essere estesa in base alle esigenze: **CubeTraining** per la fase di allenamento e **CubeTesting** per la fase di simulazione. In queste classi vengono implementati i metodi fondamentali per il funzionamento dell'agente, tra cui **OnEpisodeBegin**, **CollectObservations**, **OnActionReceived** ed **Heuristic**. Gli script vengono aggiunti come componenti, rispettivamente, ai prefab **Agent** e **BearFinal**, i GameObject che rappresentano l'agente all'interno dell'ambiente. Questi prefab includono anche altri componenti necessari come il **Rigidbody**, il **Collider**

e i **Behaviour Parameters**, essenziali per l'interazione fisica e l'apprendimento tramite ML-Agents. Attraverso questi script, l'agente è in grado di:

- osservare l'ambiente (celle adiacenti, ostacoli, obiettivo, ecc.);
- eseguire azioni (muoversi nelle quattro direzioni e saltare);
- apprendere le strategie per risolvere i labirinti durante l'addestramento.

Questa sezione descrive la struttura dell'agente e le scelte progettuali adottate per garantirne un comportamento efficiente e generalizzabile.

3.3.1 CubeTraining e CubeTesting

CubeTraining è la classe estesa da **Agent**, implementata per la raccolta delle osservazioni, la gestione del sistema di reward (vedi 3.5), i riferimenti ai GameObjects genitori e impostazioni dell'agente in ambiente Unity (vedo 3.2.1). È assegnata come componente al prefab **Agent**, il GameObject utilizzato durante la fase di allenamento. I parametri modificabili, implementati appositamente nella classe, sono:

- **Max Steps for episode**: il numero massimo di passi per episodio durante l'allenamento.
- **Move speed**: velocità di movimento dell'agente.
- **Allow agent to jump**: se attivo, l'agente sarà in grado di saltare nell'ambiente.
- **Time Scale**: parametro che controlla la velocità con cui il tempo scorre all'interno del gioco. Durante l'allenamento di si aumenta (di default pari a 20 [10]) per far procedere gli episodi più velocemente.
- **Number of obstacles**: numero di ostacoli che verranno istanziati in un labirinto.
- **Environment references**: campi *da assegnare manualmente* quando si aggiunge l'agente in un labirinto generato, con i relativi GameObject genitori del pavimento e delle mura.

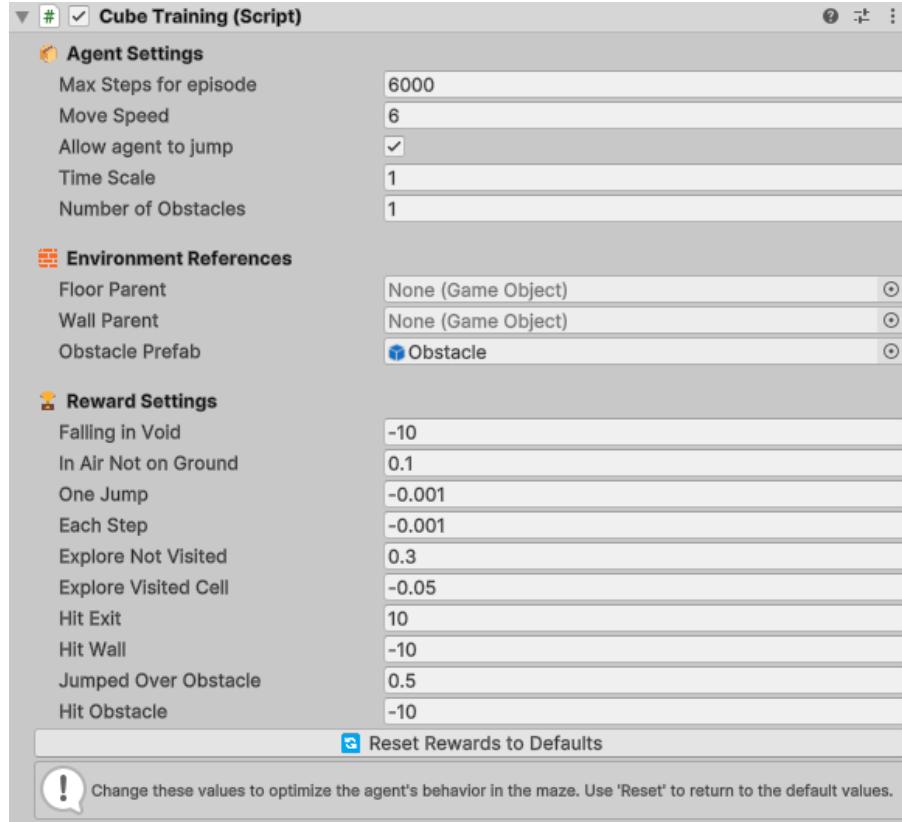


Figura 3.4: Editor per la modifica dei parametri dell’agente

La classe **CubeTesting**, a sua volta estesa da **Agent**, viene utilizzata solo nella fase di simulazione, successiva all’addestramento del modello. A differenza della fase di training, essa non gestisce il reward system, in quanto durante l’inferenza l’agente non apprende più, ma si limita ad eseguire le azioni sulla base delle decisioni apprese dal modello precedentemente allenato.

Anche in questa fase vengono mantenuti i riferimenti alle componenti dell’ambiente (*environment references*), ma non è necessario assegnarli poiché viene fatto automaticamente. Viene aggiunto il campo **Flowers**, che assegna il GameObject contenitore dei fiori che vengono istanziati sulle celle visitate, in sostituzione del cambiamento di colore utilizzato durante l’allenamento, con lo scopo di visualizzare meglio il percorso fatto dall’agente.

3.3.2 Behaviour Parameters

Behaviour Parameters è il componente essenziale per impostare il comportamento e le **proprietà del cervello (brain)** di un’istanza di agente. Questo componente, quando collegato ad un GameObject, permette di modificare i parametri legati allo spazio delle azioni e osservazioni per l’allenamento, aggiungere il file del modello una volta allenato e modificare il suo tipo di comportamento [8].

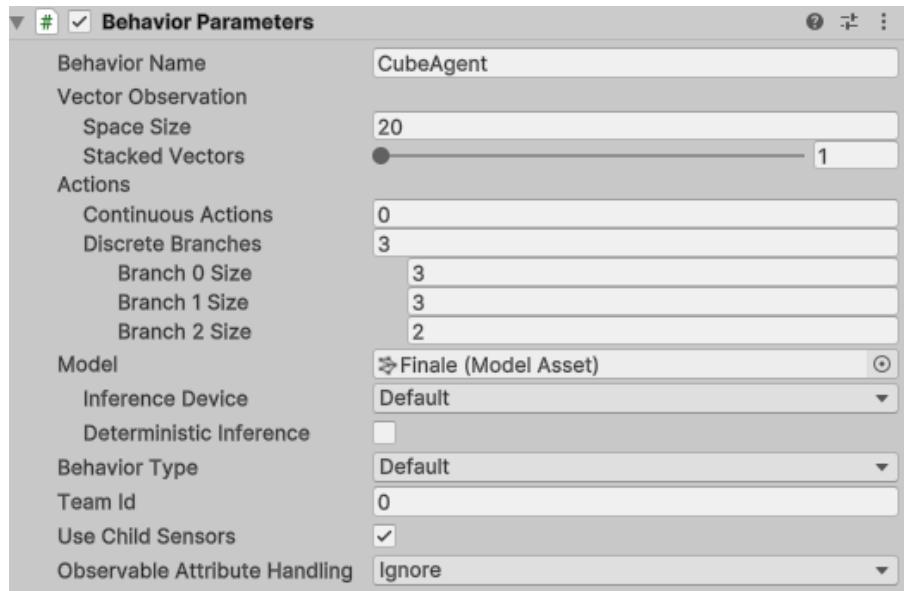


Figura 3.5: Esempio di Behaviour Parameters utilizzato

- **Behavior Name:** nome univoco associato al comportamento dell'agente, utilizzato per collegarlo al file di configurazione durante l'addestramento e all'esportazione del modello.
- **Vector Observation:** insieme di valori numerici che rappresentano le informazioni percepite dall'agente sull'ambiente circostante. È possibile impostare anche le dimensioni dei **Discrete Branches**, ovvero il numero di stati associati ad un'azione.
- **Actions:** definizione dello spazio delle azioni che l'agente può compiere. Può includere azioni discrete o continue.
- **Model:** da questa impostazione è possibile **caricare il modello**, una volta che l'allenamento è terminato.
- **Inference Device:** imposta l'hardware per l'inferenza nel momento in cui il modello è caricato in Unity per il test o l'esecuzione. È possibile scegliere tra GPU e CPU.
- **Deterministic Inference:** da attivare in fase di inferenza, una volta allenato il modello per rendere la scelta delle azioni non stocastica.
- **Behavior Type:** definisce il tipo di comportamento che l'agente utilizzerà. Nella modalità **Default** l'agente userà il processo remoto di allenamento per prendere decisioni. Se non è disponibile, utilizzerà inferenza grazie al modello caricato. Se non viene fornito, l'agente utilizzerà la funzione euristica definita.
- **Team Id:** utilizzato nella modalità di allenamento *self-play*.
- **Use Child Sensors:** se attivo, permette di utilizzare tutti i componenti di tipo sensore collegati ai GameObject figli dell'agente.
- **Observable Attribute Handling:** attributo che si può usare nel codice C# per indicare automaticamente quali proprietà o metodi di una classe devono essere osservati.

3.3.3 Decision Requester

Decision Requester è il componente che **automaticamente richiede all'agente di prendere una decisione ad intervalli regolari**: senza di esso, sarebbe necessario chiamare manualmente la funzione `RequestDecision()` [9]. È necessario definire un **Decision Period**, ovvero la frequenza con cui l'agente richiede una nuova decisione, insieme a due parametri chiave: **Decision Step**, che rappresenta il passo temporale in cui viene presa la decisione, e **Take Actions Between Decisions**, che indica se l'agente deve eseguire l'ultima azione decisa anche nei passi intermedi. Questo componente, insieme a **Behavior Parameters** è collegato al `GameObject` dell'agente.



Figura 3.6: Decision Requester

3.3.4 Spazio delle azioni

Le azioni dell'agente per l'allenamento sono definite nel cosiddetto *spazio delle azioni*. Per azioni si intende la serie di movimenti che l'agente può eseguire nell'ambiente. La libreria ML-Agents ne distingue due tipi:

- **Continue**: l'agente restituisce un valore reale (float) per ogni dimensione dell'azione, consentendo un controllo fluido e continuo (es. velocità di movimento di un braccio robotico).
- **Discrete**: l'agente sceglie tra un numero finito di opzioni per ciascun ramo di decisione. Combinazioni di azioni discrete multiple sono gestite tramite i *Discrete Branches*, dove ogni branch rappresenta una scelta indipendente (es. direzione di movimento, salto sì/no).

L'agente si dovrà muovere nelle **4 direzioni** e sarà in grado di **saltare**. Queste azioni sono di tipo **discrete** perché si tratta di scelte ben distinte che non richiedono valori numerici continui o variazioni fluide, ma solo decisioni separate e finite.

Le azioni vengono definite nella funzione `Heuristic`. Questa funzione viene utilizzata per specificare manualmente le azioni dell'agente, per controllarlo senza usare una rete neurale, per esempio nel caso in cui volessimo muovere l'agente da tastiera.

```

1 public override void Heuristic(in ActionBuffers actionsOut)
2 {
3     var actions = actionsOut.DiscreteActions;
4     actions[0] = Input.GetAxisRaw("Vertical") >= 0
5         ? Mathf.RoundToInt(Input.GetAxisRaw("Vertical"))
6         : 2;
7     actions[1] = Input.GetAxisRaw("Horizontal") >= 0
8         ? Mathf.RoundToInt(Input.GetAxisRaw("Horizontal"))
9         : 2;
10    actions[2] = Input.GetKey(KeyCode.Space) ? 1 : 0;

```

Listing 3.1: Implementazione della funzione Heuristic

Per l’agente definiamo 3 azioni discrete. Possiamo impostare la dimensione dei branch in base alla logica dei movimenti:

- **Branch movimento verticale:** ha dimensione pari a 3, poiché l’agente può scegliere se muoversi in avanti, indietro oppure rimanere fermo.
- **Branch movimento orizzontale:** ha dimensione pari a 3, poiché l’agente può muoversi a destra, a sinistra oppure rimanere fermo.
- **Branch salto:** ha dimensione pari a 2, poiché l’agente può decidere se saltare oppure no.

3.3.5 Spazio delle osservazioni

Lo **spazio delle osservazioni** è l’insieme di tutte le informazioni, raccolte tramite script oppure sensori, che vengono fornite alla rete neurale per descrivere lo stato corrente dell’ambiente percepito dall’agente posizionato nell’ambiente. Queste osservazioni possono essere di due tipi principali:

- **Osservazioni vettoriali:** rappresentano informazioni numeriche strutturate, come posizioni relative, stati logici (es. presenza di un ostacolo), flag binari, distanza dall’obiettivo, ecc. Sono spesso raccolte direttamente tramite script.
- **Osservazioni visive:** consistono in immagini catturate da una o più telecamere virtuali (Camera Sensor) posizionate sull’agente. Queste osservazioni sono utili per compiti in cui la visione spaziale o la percezione visiva è fondamentale, e vengono elaborate da reti neurali convoluzionali (CNN).

In questo progetto sono state raccolte osservazioni solo di tipo *vettoriale*, poiché i dati raccolti con i sensori e via codice sono più che sufficienti e gli ambienti di allenamento non sono troppo diversificati da richiedere osservazioni visive.

3.3.5.1 RayPerceptionSensorComponent3D

Il **RayPerceptionSensorComponent3D** [3] è un componente che permette a un agente di percepire l’ambiente circostante tramite raggi 3D, simulando un senso di “vista” direzionale simile a un radar. I raggi possono individuare gli oggetti nella scena che hanno un determinato *tag*, da noi assegnato, in modo tale da raccogliere informazioni solo sugli oggetti a noi interessati. Si raccolgono in particolare:

- distanza dall’oggetto rilevato;
- tag dell’oggetto rilevato;
- direzione del raggio.

Per ogni serie di raggi della singola componente è possibile definire diverse impostazioni, tra cui: i tag rilevabili, la lunghezza del raggio e posizione, il numero di raggi per direzione e il loro angolo massimo. Ad ogni sensore è possibile assegnare un colore per facilitarne il riconoscimento in fase di allenamento, nel caso in cui un oggetto venga colpito dal raggio o meno. All’agente sono stati assegnati due sensori: È possibile impostare più di un componente **RayPerceptionSensorComponent3D**, per differenziare ciò che l’agente osserva da diverse angolazioni o con scopi diversi. All’agente sono stati aggiunti due sensori:

- *FloorSensor* è utile per rilevare se la cella corrente e quella successiva all'agente fanno parte del pavimento, quindi sono sicure e visitabili, oppure se rappresentano ostacoli da evitare (ovvero tutto ciò che non è taggato "floor").
- *WallTargetSensor* si occupa di raccogliere informazioni sugli oggetti che circondano l'agente, come mura e obiettivi (target). In questo modo fornisce una sorta di campo visivo che permette all'agente di navigare correttamente, senza urtare le mura, e di raggiungere l'uscita quando è alla portata dei raggi.

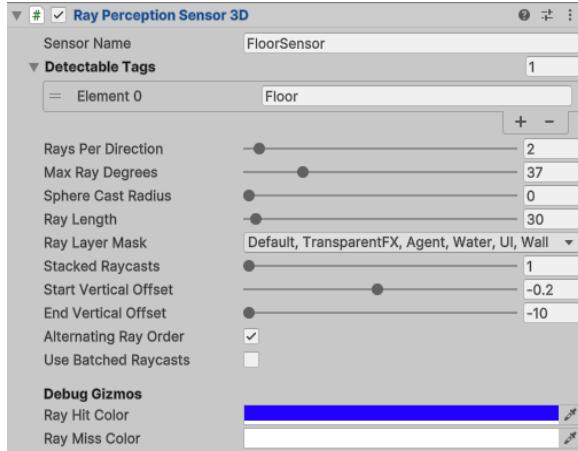


Figura 3.7: FloorSensor

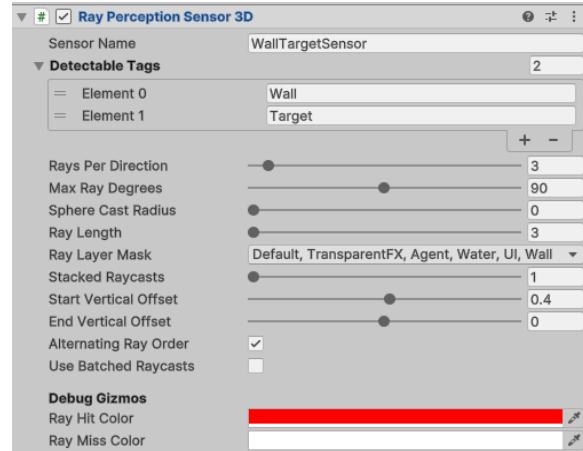
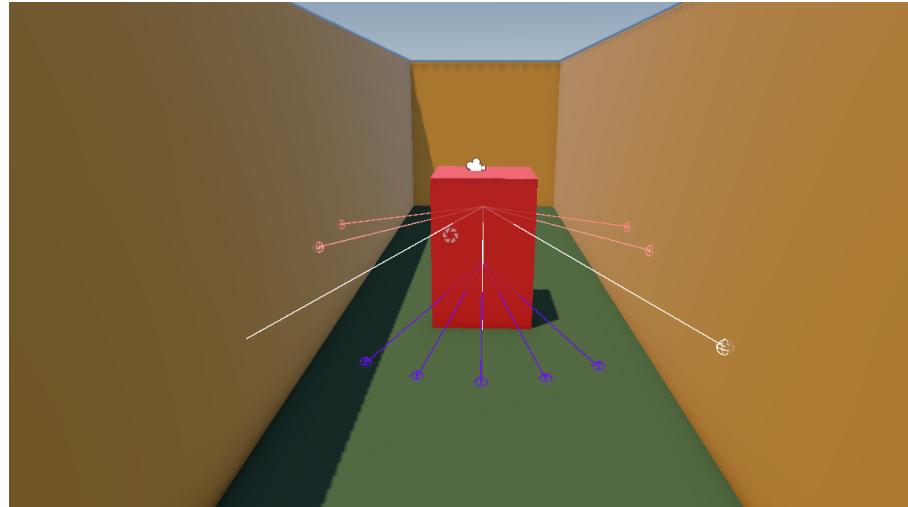


Figura 3.8: WallTargetSensor

Figura 3.9: In blu *FloorSensor*, in rosso/bianco *WallTargetSensor*

3.3.5.2 Stato delle celle

Un obiettivo fondamentale del modello è quello di evitare di percorrere strade già visitate. Bisogna creare una sorta di memoria locale, che permette all'agente di "ricordarsi" quale singola cella vicino a ha già visitato e quante volte. Queste informazioni si possono raccogliere facilmente via codice.

Sia x la cella corrente, ovvero la cella in cui **l'agente è posizionato**. Per ciascuna delle 4 celle adiacenti ad x vengono salvate le informazioni riguardo al loro stato: *visitata*, *non visitata* e *invalida* (ovvero un ostacolo oppure cella adiacente non esistente). Viene

inoltre registrato il numero di volte in cui ciascuna di queste celle viene visitata. Le informazioni sono utili per guidare il comportamento dell'agente durante l'esplorazione, permettendogli di:

- Evitare celle già visitate frequentemente, incentivando l'esplorazione.
- Identificare le celle invalide (come ostacoli o bordi della mappa) ed evitarle.
- Dirigersi verso celle non ancora esplorate, favorendo la scoperta di nuove aree.

Lo stato della cella viene registrato come osservazione utilizzando una codifica **one-hot**: si utilizza un vettore binario di 3 bit, ad esempio [0,0,0], in cui ciascun bit corrisponde a uno dei tre stati possibili della cella: **non visitata**, **visitata**, **invalida**. Il bit corrispondente allo stato effettivo della cella assume valore 1, mentre gli altri restano a 0.

Il numero di volte in cui una cella viene visitata è invece un singolo valore *float*, normalizzato da 0 a 1. Il totale delle osservazioni raccolte per ogni step è quindi **20**: 3×5 per lo stato, 5 per il numero di visite. Considerando l'attuale configurazione nell'immagine, l'observation space sarà il seguente:

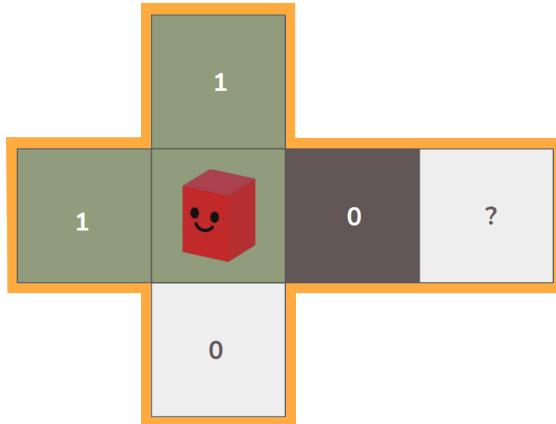


Figura 3.10: Configurazione corrente

| | One-hot | Visite |
|-----------------------|---------|--------|
| Cella corrente | [0,1,0] | 0.3 |
| Cella sinistra | [0,1,0] | 0.1 |
| Cella destra | [0,0,1] | 0 |
| Cella avanti | [0,1,0] | 0.1 |
| Cella dietro | [1,0,0] | 0 |

Tabella 3.1: Spazio delle osservazioni della configurazione

3.4 Ambienti di allenamento

Per creare un modello generalizzato in grado di risolvere labirinti, è necessario allenarlo in ambienti sempre diversi e casuali. Questo per evitare di allenare un modello over-fitted, in grado di risolvere solo **labirinti simili**. Gli aspetti casuali da tenere in considerazione per la generalizzazione di un modello in grado di risolvere labirinti sono divisibili per **randomizzazione dell'ambiente** e **randomizzazione dell'agente**.

Applicando il paradigma del **curriculum learning** (spiegato in 2.3.2), è fondamentale allenare l'agente partendo da ambienti semplici per favorire un apprendimento graduale. Posizionare l'agente sin dalla prima fase di addestramento all'interno di un labirinto di media o elevata complessità potrebbe comprometterne l'efficacia. Per questo motivo, il training inizia da scenari molto semplici (come corridoi lineari) per poi passare, man mano che l'agente acquisisce competenze, a labirinti più complessi.

3.4.1 Randomizzazione dell’ambiente

Per migliorare la capacità di generalizzazione dell’agente e prevenire l’overfitting su un layout fisso, viene introdotta la **randomizzazione dell’ambiente** per ogni allenamento. In particolare, ad ogni episodio vengono variati i seguenti elementi:

- **Posizione degli ostacoli:** gli ostacoli vengono disposti casualmente all’interno del labirinto, rispettando i vincoli imposti precedentemente.
- **Tipologia degli ostacoli:** si possono alternare diversi tipi di ostacolo, ovvero i buchi nel terreno e quelli rialzati.
- **Posizione dell’uscita:** l’uscita viene collocata in un punto casuale accessibile.
- **Layout del labirinto:** viene generata una nuova configurazione del labirinto utilizzando algoritmi di generazione procedurale.

3.4.2 Randomizzazione dell’agente

Allo stesso modo, vogliamo randomizzare la posizione e la rotazione di partenza ad ogni episodio dell’agente, per ogni fase di allenamento:

- **Posizione:** ad inizio di un nuovo episodio, l’agente viene casualmente posizionato su una delle celle che compongono il labirinto. Non viene posizionato sopra agli ostacoli, poiché sarebbe uno scenario poco utile per l’allenamento.
- **Rotazione:** la rotazione iniziale dell’agente è casuale per permettere di apprendere strategie che siano indipendenti dall’orientamento iniziale, dato che i *raggi* del RaySensorPerception3D vengono posizionati sempre nella stessa direzione e angolo.

3.4.3 Corridoio

In una prima fase di addestramento, l’agente viene posto in un ambiente **molto semplice**, privo di ostacoli e con la sola presenza dell’uscita. Lo scopo di questa fase iniziale è quello di permettere all’agente di apprendere i concetti fondamentali necessari per affrontare ambienti più complessi. In particolare, l’agente deve imparare a:

- associare il raggiungimento dell’uscita ad una ricompensa positiva;
- evitare di urtare contro i muri del labirinto;
- muoversi verso l’uscita quando questa è visibile, altrimenti esplorare l’ambiente per trovarla.

Questa fase costituisce la base del processo di *curriculum learning*, facilitando l’apprendimento graduale e riducendo la complessità iniziale del compito.

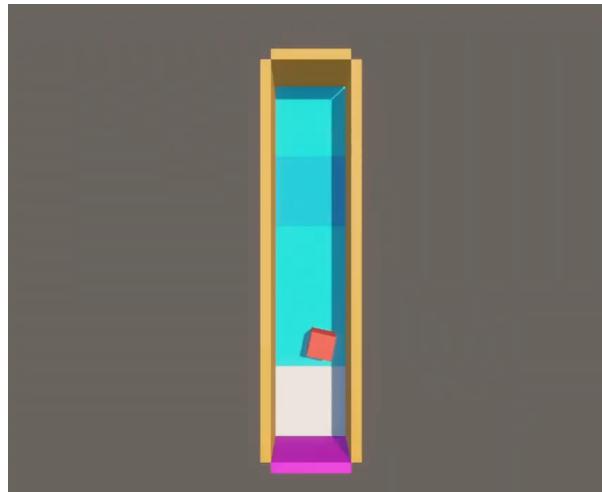


Figura 3.11: Agente durante la prima fase di allenamento

3.4.4 Corridoio con ostacolo

Nella seconda fase di addestramento, l’agente viene sempre posto all’interno di un corridoio lineare, ma con all’interno **un ostacolo**, *randomizzato* in termini di posizione e tipo. Questa fase permette all’agente di imparare a:

- superare gli ostacoli attraverso il salto;
- evitare di eseguire salti inutili durante l’esplorazione del labirinto.
- evitare cadute in buchi presenti sul pavimento;
- consolidare le conoscenze acquisite nella fase precedente.

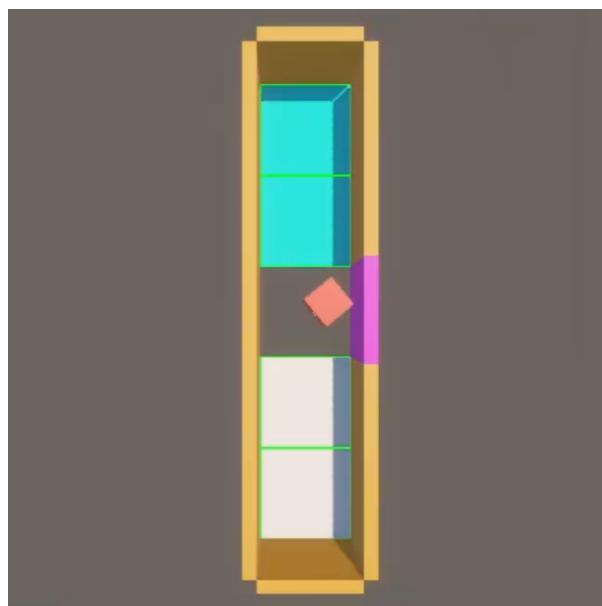


Figura 3.12: L’agente supera l’ostacolo e va verso il target

3.4.5 Labirinti NxN

Nella terza e ultima fase di addestramento, la più complessa e articolata, l’agente viene posto all’interno di labirinti generati proceduralmente, caratterizzati da dimensioni variabili e **configurazioni sempre diverse**. Tutti gli elementi dell’ambiente, come la posizione degli ostacoli, il layout del labirinto e la posizione dell’uscita, sono randomizzati secondo le modalità descritte in precedenza. Questa fase consente di valutare la capacità dell’agente di generalizzare le competenze acquisite nelle fasi precedenti e di adattarsi a nuove situazioni, come per esempio svoltare un angolo o saltare più ostacoli di fila, sviluppando le strategie corrette per risolvere labirinti di complessità crescente.

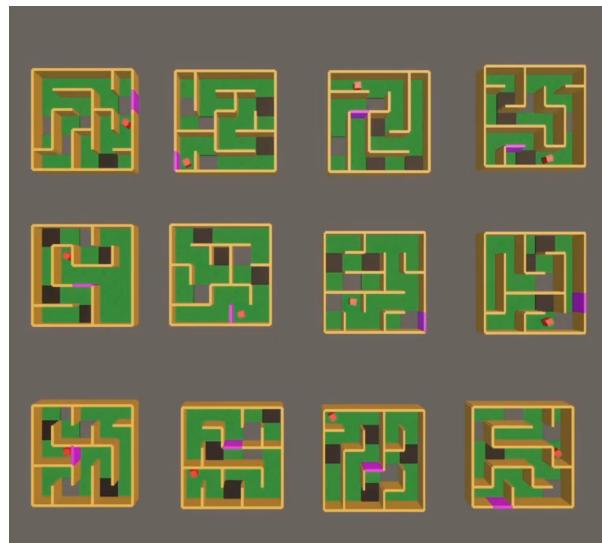


Figura 3.13: Serie di labirinti 5x5, con 5 ostacoli, sempre randomizzati

3.5 Implementazione del reward system

Nel progetto, il *sistema di ricompense* è implementato nello script `CubeTraining`, utilizzando i metodi `SetReward` e `AddReward`.

Il metodo `SetReward` viene usato per impostare il valore della ricompensa istantanea dell’agente nello step corrente. Aggiorna il reward cumulativo dell’episodio in base alla differenza tra il nuovo reward e quello precedente, senza sovrascriverlo. Può essere usato in qualsiasi momento dell’episodio, anche alla fine.

Il metodo `AddReward`, invece, viene impiegato per aggiungere progressivamente ricompense o penalità durante l’episodio, sulla base dei comportamenti dell’agente.

Il metodo `ApplyReward` serve a semplificare e rendere più leggibile l’assegnazione delle ricompense all’interno dello script dell’agente.

```

1  private void ApplyReward(float value, string reason)
2  {
3      AddReward(value);
4      Debug.Log($"[Reward] {value} for {reason}");
5  }

```

Listing 3.2: Implementazione della funzione `ApplyReward`

3.5.1 Reward per il movimento

Il reward system è gestito a partire dal metodo `OnActionReceived`, utilizzato per l'esecuzione delle azioni possibili dell'agente durante l'allenamento:

- **Penalità per movimenti inefficienti:** in ogni `OnActionReceived`, viene applicata una piccola penalità costante per ogni step al fine di incentivare l'agente a trovare la soluzione nel minor numero di passi possibile. Inoltre, sono applicate ulteriori penalità per rotazioni ripetute o avanzamenti contro muri.
- **Penalità per scontro contro un muro:** grazie al sistema di `Colliders` dei `GameObject`, è possibile notificare grazie ai metodi `OnTriggerEnter` e `OnCollisionEnter` quando l'agente `collide` contro un muro ed assegnare la punizione.
- **Ricompensa per raggiungimento del target:** come per la collisione contro un muro, ricompensiamo l'agente quando collide con il target.

3.5.2 Reward per l'esplorazione

Il reward system è gestito nel metodo `EvaluateExplorationReward` utilizzando un `Dictionary(<Transform, int>)` che tiene traccia di quante volte una cella viene visitata. E' così gestito:

- **Ricompensa se visita una nuova cella:** l'agente riceve una piccola ricompensa positiva ogni volta che si sposta su una cella non ancora visitata. Per verificare se una cella è nuova, viene tenuta una mappa delle celle visitate a partire dalla posizione dell'agente.
- **Punizione se visita una cella già esplorata più volte:** se l'agente ritorna su una cella che ha già visitato **più di 1 volta**, viene penalizzato. Più volte visita la stessa cella, maggiore sarà la penalità cumulativa. Questo scoraggia il comportamento ciclico.

3.5.3 Reward per il superamento degli ostacoli

Il reward system per il salto è gestito nel metodo `EvaluateJumpRewards`, sfruttando il booleano `jumpedOverObstacle`. Questo valore viene determinato all'interno del metodo `OnCollisionEnter`, dove viene verificato se la cella adiacente, nella direzione del movimento, è una cella valida (quindi non un muro o uno spazio vuoto) e risulta diversa dalla cella da cui l'agente ha effettuato il salto. Se la condizione è soddisfatta, significa che l'agente ha effettivamente superato un ostacolo e il flag `jumpedOverObstacle` viene impostato su `true`.

Prima di ricompensare l'agente per aver effettivamente superato un ostacolo, è necessario che l'agente impari che **saltare è una strategia utile**. Per questo motivo, viene assegnata una piccola ricompensa quando l'agente si trova **in aria sopra un ostacolo**, suggerendo così che questa azione è desiderabile e può portare a una ricompensa maggiore. In questo modo, si guida l'agente a esplorare e consolidare il comportamento corretto senza fornire subito la ricompensa finale.

3.6 Menù di simulazione

Una volta completato l’addestramento, il modello può essere testato attraverso il **menù di simulazione** disponibile a runtime. Per procedere, è sufficiente assegnare il file del modello al campo **Model** della componente **Behaviour Parameters** associata al prefab **BearFinal**, situato nella cartella “Final Assets”. Successivamente, avviando la scena **Main** in modalità di esecuzione, sarà possibile valutare *visivamente* le prestazioni del modello. La scena prevede la generazione procedurale di labirinti di diverse dimensioni e configurazioni, permettendo così di testare la capacità di generalizzazione e adattamento dell’agente a vari ambienti.

All’avvio della scena, l’interfaccia consente di configurare le seguenti opzioni:

- **Dimensioni del labirinto** (es. 5×5 , 7×7 , 10×10 , 15×15).
- **Numero di ostacoli** presenti nel percorso.

E’ possibile generare un nuovo labirinto premendo il pulsante assegnato e successivamente **Start** e osservare il comportamento dell’agente. Durante la simulazione, è possibile visualizzare informazioni come il numero di celle visitate e l’eventuale successo nel raggiungere il target, zoomare sulla mappa e mini-mappa.



Figura 3.14: L’agente ha raggiunto il target in un labirinto 10×10 con 8 ostacoli

3.7 Integrazione e usabilità

Il framework è progettato per offrire flessibilità consentendo all’utente di allenare facilmente più modelli. Grazie alla possibilità di modificare sia il reward system, sia i parametri della rete neurale e dell’agente, è possibile esplorare diverse configurazioni e identificare i valori ottimali per il comportamento desiderato. Inoltre, l’integrazione di strumenti per la generazione automatizzata degli ambienti di allenamento semplifica e velocizza il processo di setup.

3.7.1 Modifica dei valori del reward system

Il sistema consente la modifica dinamica dei valori del reward system direttamente dall’interfaccia dell’editor di Unity, senza la necessità di intervenire sul codice. Questa funzionalità è resa possibile grazie alla classe `RewardSettingsEditor`, che implementa un’interfaccia grafica personalizzata. Da qui l’utente può regolare i pesi associati alle diverse componenti della funzione di ricompensa, facilitando il tuning del comportamento dell’agente. E’ possibile modificare anche altri parametri rilevanti relativi all’agente, come mostrato nella sezione 3.3.1.

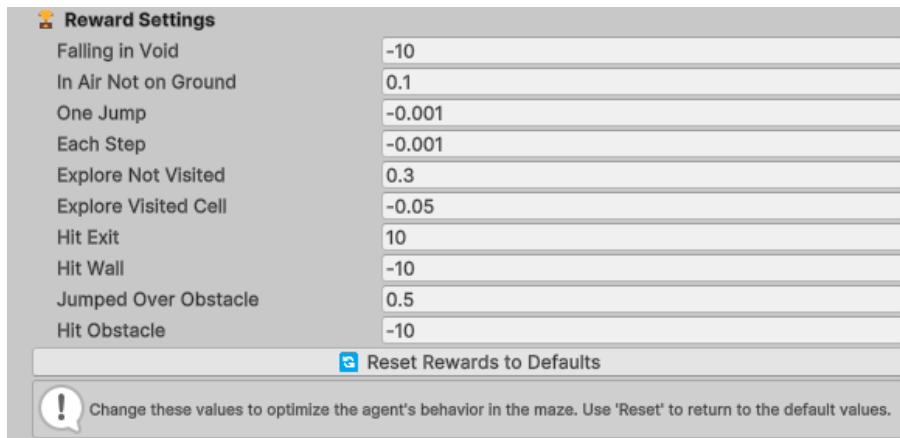


Figura 3.15: Editor per la modifica dei parametri dell’agente

3.7.2 Tool di generazione di ambienti per l’allenamento

È stata realizzata una classe di tipo `EditorWindow` che funge da interfaccia grafica personalizzata all’interno dell’editor di Unity, con l’obiettivo di facilitare la generazione delle scene di allenamento. Durante il processo di training, è spesso necessario istanziare più copie dello stesso ambiente per massimizzare la raccolta di dati e velocizzare l’apprendimento dell’agente. Questo strumento consente all’utente di selezionare quali ambienti (labirinti) istanziare e in quale quantità, evitando operazioni manuali e ripetitive di copia e incolla. Inoltre, è presente una funzionalità per il reset completo dell’editor, che permette di eliminare rapidamente tutte le istanze create, facilitando la configurazione di nuove sessioni di allenamento.

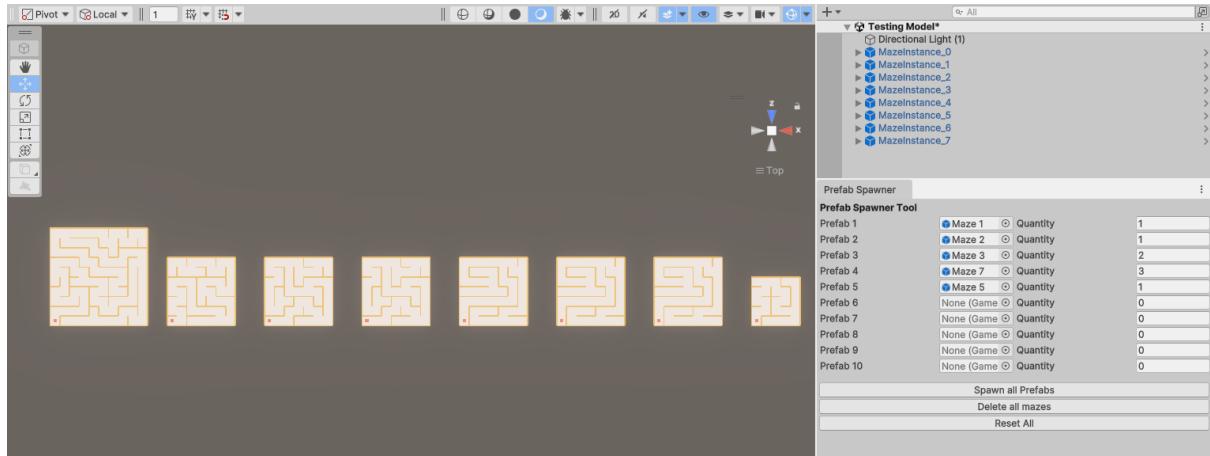


Figura 3.16: Esempio di istanze di labirinti diversi generati con il tool

3.7.3 Raccolta dei dati per l'analisi del modello

Dopo aver ipotizzato i parametri migliori per l'allenamento, è utile verificare che il modello sia effettivamente ottimo e se è necessario effettuare un ulteriore fase di allenamento o addirittura ricominciare da zero. A questo scopo è stata implementata la classe `BenchmarkLogger`. Per ogni *episodio*, che sia di successo o meno, vengono salvati i dati relativi al numero di celle esplorate e il reward cumulativo in un file .csv nella cartella `Benchmarks`.

| Success | Steps | CellsVisitedUnique | RivisitedCells | TotalCells | CumulativeReward | numberOfJumps |
|---------|-------|--------------------|----------------|------------|------------------|---------------|
| 0 | 76 | 43 | 116 | 159 | 9,927 | 117 |
| 0 | 5930 | 61 | 105 | 166 | -15,92875 | 105 |
| 0 | 188 | 46 | 94 | 140 | 9,813 | 95 |
| 0 | 1025 | 58 | 97 | 155 | -11,02399 | 94 |
| 0 | 3801 | 63 | 113 | 176 | -13,79691 | 94 |
| 0 | 305 | 42 | 129 | 171 | -10,3 | 93 |
| 0 | 1725 | 54 | 138 | 192 | 8,275975 | 93 |
| 1 | 5903 | 79 | 111 | 190 | 4,098245 | 93 |
| 0 | 587 | 47 | 117 | 164 | 9,418004 | 91 |
| 0 | 695 | 49 | 136 | 185 | 9,306005 | 91 |
| 1 | 5735 | 62 | 118 | 180 | 4,269233 | 89 |
| 1 | 5918 | 68 | 103 | 171 | 4,084246 | 89 |

Figura 3.17: Esempio di dati raccolti con la class `BenchmarkLogger`

4

Assets 3D

In questo capitolo vengono descritti gli asset realizzati per la costruzione dell’ambiente di simulazione in Unity. L’obiettivo principale è stato quello di sviluppare elementi grafici che potessero essere integrati in un sistema di generazione procedurale dei labirinti. Oltre all’ambiente, è stato creato un modello 3D per rappresentare visivamente l’agente, migliorando l’esperienza durante la fase di simulazione. Gli asset sono stati progettati in Blender ottimizzati per l’utilizzo in tempo reale, garantendo performance adeguate in Unity.

4.1 Creazione di assets per Unity

Nella creazione di asset 3D in *Blender* da utilizzare in Unity ci sono diversi aspetti tecnici importanti da tenere in considerazione per il corretto funzionamento della scena, prima di procedere con modellazione:

- La **scala dell’oggetto deve essere 1:1** rispetto a Unity, applicando le trasformazioni prima dell’esportazione.
- In Unity l’origine dell’oggetto è usata come punto di riferimento per il posizionamento. In Blender, è necessario **impostare correttamente il punto di origine** (pivot), centrato rispetto alla geometria [13].
- Più **vertici** sono presenti in un modello, maggiore sarà il **carico computazionale** necessario per il rendering, l’elaborazione fisica e le collisioni. Questo può influire negativamente sulle performance complessive dell’applicazione, soprattutto nei casi in cui molti modelli sono presenti simultaneamente nella scena, come nel caso dei muri di un labirinto molto grande.

In un contesto di *reinforcement learning*, la realizzazione di nuovi asset 3D da impiegare nella generazione dei labirinti durante la simulazione richiede particolare attenzione. È fondamentale sottolineare che il modello è stato addestrato **in un ambiente in cui gli oggetti 3D presentano dimensioni specifiche**, così come l’agente stesso ha una

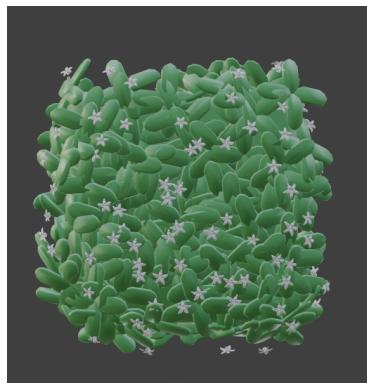
scala ben definita. Anche una variazione minima nelle dimensioni degli asset potrebbe compromettere il corretto funzionamento del modello. Questo perché le osservazioni vengono acquisite tramite il componente `RayPerceptionSensorComponent3D`, che rileva anche la **distanza dall'oggetto colpito** (vedi 3.3.5.1).

4.2 Creazione dei modelli per i labirinti

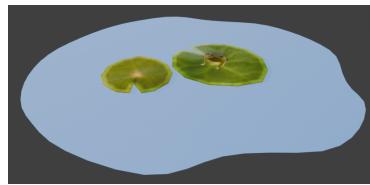
I labirinti per la fase di training sono stati costruiti con le **primitive base di Unity**, per consentire in fase di allenamento l'uso meno dispendioso di risorse per il rendering. In fase di simulazione possiamo creare assets più complessi, tenendo in considerazione quello detto precedentemente:

- **Siepe** (fig. 4.1a): realizzata a partire da un parallelepipedo, al quale sono stati aggiunti due *particle emitter*, uno per le foglie e uno per i fiori. Le componenti sono state poi unite in un unico oggetto. Le dimensioni complessive corrispondono a quelle del prefab `Wall`.
- **Ostacoli** : sono stati creati due tipi di ostacoli. Il primo è un insieme di **tronchi di legno** (fig. 4.1d) che formano un blocco sopraelevato da superare con un salto; il secondo è una **pozzanghera** (fig. 4.1b) che simula un buco nel terreno. Entrambi hanno dimensioni pari a una cella del labirinto.
- **Target** (fig. 4.1e): modellato come una siepe a forma di arco con un vaso di miele posizionato al centro, utilizzando lo stesso modello di siepe impiegato per le mura del labirinto.
- **Fiore** (fig. 4.1c): il modello del fiore viene utilizzato in fase di simulazione per visualizzare il percorso seguito dall'agente. Ogni volta che l'agente attraversa una cella, un fiore viene istanziato al centro della cella stessa.

Durante la fase di simulazione viene caricata a *runtime* la scena `Main`, che rappresenta l'ambiente principale in cui avvengono le simulazioni dell'agente una volta allenato il modello. All'interno di questa scena è presente un oggetto **GameObject** denominato `MazeGenerator3D`, al quale è associato lo script `MazeGeneratorTesting`. Questo consente di specificare in modo semplice gli *assets* da utilizzare per la costruzione del labirinto (pavimenti, muri, ostacoli, agente).



(a) Siepe



(b) Pozzanghera



(c) Fiore



(d) Tronchi di legno



(e) Target

Figura 4.1: Assets utilizzati per la generazione dei labirinti



Figura 4.2: Esempio di labirinto 7x7 generato con i nuovi assets

4.3 Creazione del modello 3D dell’Agente

Il *parallelepipedo* utilizzato inizialmente per rappresentare l’agente è stato sostituito da un **modello 3D di un orso animato**, al fine di rendere la simulazione visivamente dinamica. Il modello è stato importato da Blender in formato .fbx ed è stato associato a un **Animator Controller in Unity**. Quest’ultimo gestisce due animazioni principali: una di idle (fermo) e una di camminata (walking animation). Il passaggio tra le animazioni avviene automaticamente in base allo stato dell’agente, grazie a dei parametri definiti all’interno dell’Animator.

Una volta creato il modello dell’orso e importato correttamente in Unity, è necessario riassegnare le stesse componenti presenti sull’agente originale allo stesso modo anche al nuovo modello. In particolare, vanno trasferiti componenti come il **Rigidbody**, il **Collider**, lo script **CubeTesting** che gestisce il comportamento dell’agente e i componenti della libreria ML-Agents. Non è necessario modificare in alcun modo quest’ultimi poiché **il funzionamento dell’agente non dipende dal modello grafico utilizzato se assume le stesse dimensioni dell’asset originale**. Finché vengono mantenute le stesse componenti e configurazioni funzionali, l’agente continuerà a comportarsi correttamente, indipendentemente dalla sua rappresentazione visiva.



Figura 4.3: Asset 3D dell’agente



Figura 4.4: Target raggiunto dall’agente

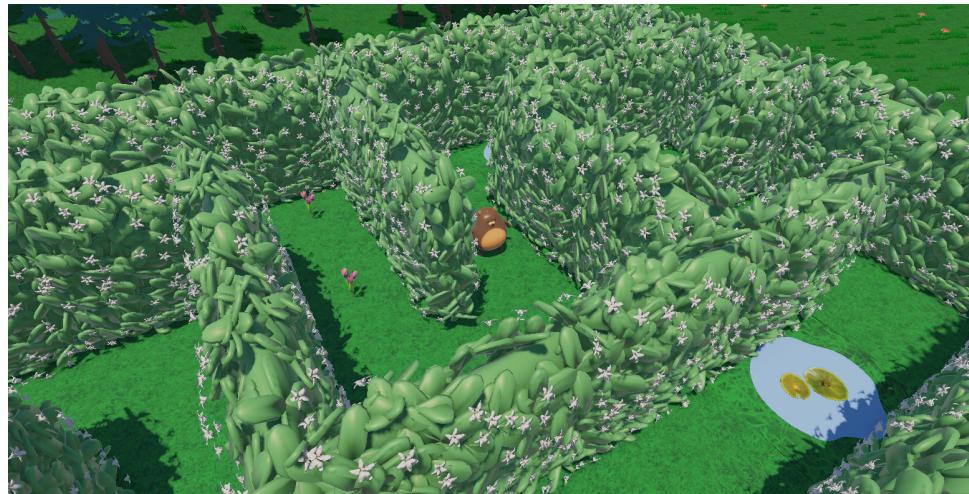


Figura 4.5: Agente mentre esplora un labirinto

5

Allenamento

Una volta creati gli ambienti di addestramento e configurato l’agente, si procede con la fase di training vera e propria. In questa fase, l’agente interagisce con l’ambiente per apprendere, attraverso un processo basato sulle ricompense ricevute. La configurazione dei parametri, la definizione del reward system e la strategia di curriculum learning sono elementi fondamentali per guidare l’agente verso un comportamento ottimale e generalizzabile.

5.1 Parametri di training

Per allenare un modello in modo efficace, è fondamentale definire accuratamente i valori del training configuration file e del reward system. Una configurazione ben progettata influisce direttamente sulla capacità del modello di convergere verso una politica ottimale. I parametri utilizzati in questo progetto sono stati scelti attraverso un processo iterativo di trial-and-error, osservando il comportamento dell’agente durante i vari cicli di allenamento, a partire dal *range di valori* consigliato dalla documentazione ufficiale [15]. Questo approccio ha permesso di effettuare un fine tuning progressivo, migliorando le performance dell’agente passo dopo passo. Verranno elencati gli allenamenti che hanno portato al risultato del modello finale.

5.1.1 Configurazione del Training File

Il file di configurazione utilizzato per l’allenamento dell’agente **CubeAgent** è riportato di seguito, una sezione alla volta, con le motivazioni alla base della scelta dei parametri:

5.1.1.1 Iperparametri comuni

```
CubeAgent:  
    trainer_type: ppo  
    hyperparameters:  
        batch_size: 128
```

```

buffer_size: 2048
learning_rate: 0.0003
beta: 0.005
epsilon: 0.2
lambd: 0.95
num_epoch: 3
shared_critic: false
learning_rate_schedule: linear
beta_schedule: linear
epsilon_schedule: linear
checkpoint_interval: 20000
max_steps: 1000000

```

Questi parametri sono quelli comuni ad entrambi i *trainers* (ppo e sac). In particolare è stato utilizzato ppo, come spiegato nella sezione 2.3.1.

- I valori assegnati a `batch_size` e `buffer_size` sono strettamente collegati tra loro. Il valore di `batch_size` deve essere **molto più piccolo** di quello di `buffer_size`. Poiché utilizziamo PPO con azioni discrete (vedi 3.3.4), il range ottimale è un valore tra 32-512 per `batch_size` e 2048-409600 per `buffer_size`
- Il `learning_rate` pari a 0.0003 è un valore comune nei contesti PPO, abbastanza basso da assicurare stabilità ma sufficiente per un apprendimento corretto.
- I valori di `beta`, `epsilon` e `lambd` (rispettivamente 0.005, 0.2 e 0.95) sono i valori di default tipicamente utilizzati per il controllo di varianza nella stima dei vantaggi ed entropia.

5.1.1.2 Setting della rete neurale

```

network_settings:
  normalize: false
  hidden_units: 256
  num_layers: 2
  vis_encode_type: simple
  memory: null
  goal_conditioning_type: hyper
  deterministic: false

```

La normalizzazione è disabilitata (`normalize: false`), dato i dati vengono già normalizzati da codice. Poiché le azioni e il goal da raggiungere sono relativamente semplici, **2 reti** da 256 unità ciascuno (`num_layers: 2`, `hidden_units: 256`), sono più che sufficienti per l'allenamento. L'encoder visivo, per la raccolta di dati visuali, è semplice (`vis_encode_type: simple`) è adatto a input visivi poco complessi, riducendo il carico computazionale. L'assenza di memoria ricorrente (`memory: null`) indica che l'agente prende decisioni basate solo sulle osservazioni correnti, semplificando la rete e velocizzando l'allenamento.

5.1.1.3 Reward estrinseco e intrinseco

```
reward_signals:
  extrinsic:
    gamma: 0.99
    strength: 1.0
```

Il parametro `gamma` è impostato a 0.99 per garantire che l'agente consideri soprattutto le ricompense future: è importante perché in un labirinto spesso la soluzione richiede una sequenza di **azioni coerenti e lunghe**. Il `strength` è fissato a 1.0 per utilizzare la ricompensa estrinseca così com'è per mantenere stabile il processo di training.

Infine, quando viene avviato un nuovo processo di addestramento, il file di configurazione può includere ulteriori parametri, ad esempio per indicare un modello pre-esistente da cui partire nel caso si voglia eseguire un *fine tuning*.

5.1.2 Reward System

Il *reward system* è lo strumento attraverso cui si definisce il comportamento che vogliamo che l'agente apprenda per risolvere i labirinti. L'obiettivo finale, esplorare il labirinto fino a trovare il *target*, deve essere scomposto in una serie di *sottotask*, ognuno dei quali rappresenta un passo intermedio nel processo di apprendimento.

Questi sottotask vengono formalizzati mediante un sistema di ricompense e penalità, associate ai comportamenti che si desidera incentivare o scoraggiare. Il valore numerico attribuito a ciascuna ricompensa o punizione ha un impatto cruciale sull'efficacia dell'apprendimento: se un certo comportamento deve essere evitato, allora sarà necessaria una penalità sufficientemente negativa. Allo stesso modo, per favorire l'apprendimento di un comportamento corretto, il relativo reward dovrà avere un valore abbastanza alto da renderlo conveniente per l'agente [12].

Il reward viene assegnato **per ogni step**, dove questo rappresenta un singolo passaggio del ciclo di interazione tra agente e ambiente, che corrisponde al numero di frame impostati nel **Decision Requester** (vedi 3.3.3). Alla fine di ogni episodio viene assegnato un reward cumulativo, che corrisponde alla somma di ogni reward per ogni step.

5.1.3 Imparare a muoversi

L'agente deve imparare a muoversi correttamente all'interno del labirinto. In particolare, l'agente:

1. Non deve urtare i muri.
2. Deve evitare movimenti non necessari.
3. Deve raggiungere il target se in area (ovvero visibile dai `RayPerceptionSensor`).

| Condizione | Tipo | Reward |
|--------------------|-----------|--------|
| Colpisce un muro | SetReward | -10.0 |
| Per ogni step | AddReward | -0.001 |
| Salta senza motivo | AddReward | -0.001 |
| Target raggiunto | SetReward | +10.0 |

Tabella 5.1: Reward utilizzati per l'agente durante l'episodio

- **Colpisce un muro (-10.0)**: un reward fortemente negativo viene assegnato per penalizzare severamente i comportamenti che rallentano l'esplorazione e portano l'agente ad una situazione scorretta, come andare in avanti contro un muro.
- **Per ogni step e salto senza motivo (-0.001)**: un piccolo reward negativo per ogni passo e salto serve a incentivare la ricerca di soluzioni brevi ed evitare di rimanere fermo oppure saltare sul posto.
- **Target raggiunto (+10.0)**: un reward positivo significativo viene assegnato quando l'agente raggiunge l'obiettivo. Questo rafforza fortemente i comportamenti che portano al completamento del labirinto e guida l'apprendimento nella direzione corretta.

5.1.4 Imparare ad esplorare

L'agente deve essere in grado di esplorare correttamente il labirinto, evitando di visitare gli stessi percorsi già visitati:

1. Non deve esplorare celle già visitate più volte.
2. Non deve rimanere bloccato nei vicoli ciechi.
3. Deve preferire le celle non ancora visitate.
4. Deve essere in grado di riconoscere quando è necessario tornare indietro.

| Condizione | Tipo | Reward |
|----------------------------|-----------|--|
| Visita nuova cella | AddReward | +0.3 |
| Visita cella già esplorata | AddReward | $-0.005 \times$ numero di visite (se > 2) |

Tabella 5.2: Reward utilizzati per l'esplorazione

- **Visita a una nuova cella (+0.3)** Questo valore positivo, relativamente elevato, è stato scelto per incentivare fortemente l'agente a esplorare aree nuove del labirinto.
- **Visita a una cella già esplorata ($-0.005 \times$ numero di visite)**: la penalità è proporzionale al numero di visite già effettuate nella stessa cella. Bisogna distinguere i due casi in cui l'agente continua a visitare *in loop* un percorso e il caso in cui l'agente *torna indietro* da un percorso. Scalando per il numero di volte in cui una cella viene visitata e punendo solo se una cella viene visitata **più di 2 volte**, è possibile distinguere i due casi.

5.1.5 Imparare a superare gli ostacoli

L'agente deve saper superare correttamente gli ostacoli di entrambi i tipi, utilizzando l'azione del salto in modo appropriato. In particolare:

- Deve riconoscere visivamente la presenza di un ostacolo davanti a sé.
- Deve apprendere che il salto è necessario solo in presenza di un ostacolo superabile.

- Non deve saltare quando l'ostacolo non è presente.
- Deve evitare di cadere nel vuoto durante il salto.
- Deve imparare a sincronizzare il salto con il movimento in avanti per superare l'ostacolo.

| Condizione | Tipo | Reward |
|---------------------------|-----------|--------|
| Colpisce un ostacolo | SetReward | -10.0 |
| Cade nel vuoto | SetReward | -10.0 |
| Salta un ostacolo | AddReward | +0.5 |
| In aria sopra un ostacolo | AddReward | +0.1 |

Tabella 5.3: Reward utilizzati per il superamento degli ostacoli

- **Colpisce un ostacolo (-10.0)** Un valore fortemente negativo, utilizzato per far comprendere all'agente che colpire un ostacolo è un comportamento da evitare a sempre.
- **Cade nel vuoto (-10.0)** Anche in questo caso si tratta di una penalità severa, applicata al fallimento dell'agente nel saltare un ostacolo, cadendo quindi nel vuoto.
- **Salta un ostacolo (+0.5)** Un reward positivo abbastanza alto, volto a rafforzare l'apprendimento di una strategia attiva ed efficace per superare gli ostacoli.
- **In aria sopra un ostacolo (+0.1)** Ricompensa più leggera, utilizzata per sostenere la fase di esecuzione del salto, fornendo feedback positivo durante il salto *sopra un ostacolo*.

La progettazione di un reward system ben bilanciato è essenziale per un addestramento efficace. Tuttavia, non esiste una formula universale: ogni ambiente presenta specificità proprie e richiede una definizione personalizzata dei valori di *reward* e *punishment*.

5.2 Fasi del Curriculum Learning

L'allenamento dell'agente è stato fatto in 3 fasi, seguendo il paradigma del *curriculum learning* e il *fine tuning* dal modello precedente. In ogni fase, l'agente impara un comportamento nuovo per poter progressivamente imparare come risolvere labirinti sempre più complessi. In questo capitolo vengono analizzati i risultati dei singoli allenamenti ed i grafici creati grazie alla libreria **Tensorflow**.

5.2.1 Primo allenamento: corridoio

Il primo allenamento viene fatto nell'ambiente più semplice, trattato nella sezione 3.4.3. Poiché si tratta del primo allenamento, l'agente agisce in modo del tutto **casuale**, non disponendo ancora di una politica appresa. Inizialmente, i reward ottenuti si aggirano intorno a -10, principalmente a causa degli urti contro i muri. Con il progredire dell'allenamento, però, il valore del reward cumulativo (fig. 5.1) **cresce gradualmente**, fino

a stabilizzarsi attorno a +10, che corrisponde alla ricompensa per il *raggiungimento del target*.

Questo comportamento è ben visibile nella curva del reward cumulativo, dove si osserva una chiara e rapida tendenza alla **convergenza verso il valore massimo +10**, segno che l'agente ha imparato ad associare il target come obiettivo ottimale.

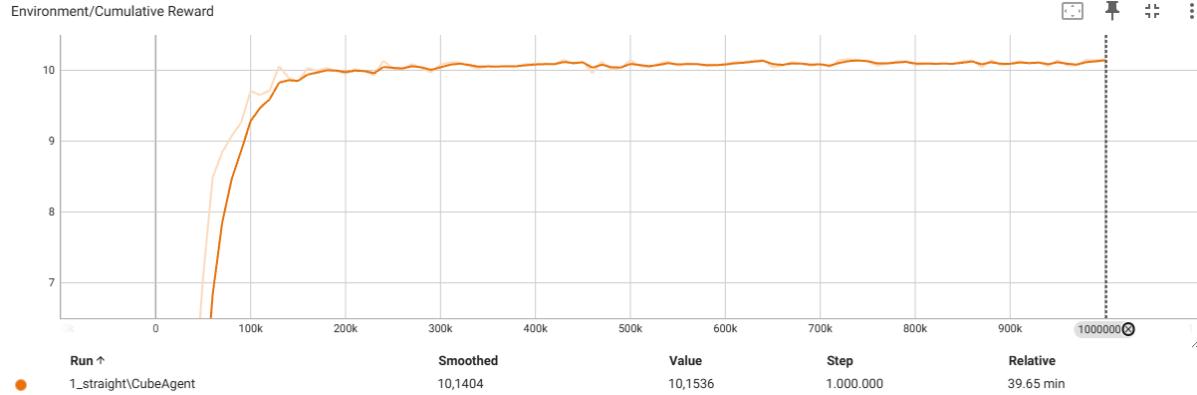


Figura 5.1: Reward cumulativo del primo allenamento

Tra i grafici generati automaticamente da Tensorflow, tra i più rilevanti ci sono:

- *Value Loss*: rappresenta l'andamento della funzione di perdita associata al valore stimato dall'agente durante l'allenamento. In un allenamento corretto, prima è alta e poi si stabilizza verso il basso.
- *Entropy*: mostra quanto è incerta o esplorativa la policy dell'agente nel tempo durante l'allenamento. Se l'entropia scende troppo in fretta, l'agente potrebbe smettere di esplorare e cadere in un ottimo locale.

Dai due grafici si nota come analogamente, anche i valori della *value loss* e dell'*entropia* mostrano un andamento decrescente nel corso dell'allenamento.

Il grafico della value loss (fig. 5.2) mostra una rapida discesa intorno ai 100.000 passi, segno che il modello sta imparando a stimare correttamente il valore degli stati. Allo stesso tempo, nel grafico della ricompensa cumulativa, osserviamo una stabilizzazione nel range massimo proprio intorno a quella soglia, indicando che l'agente sta imparando un comportamento efficace.

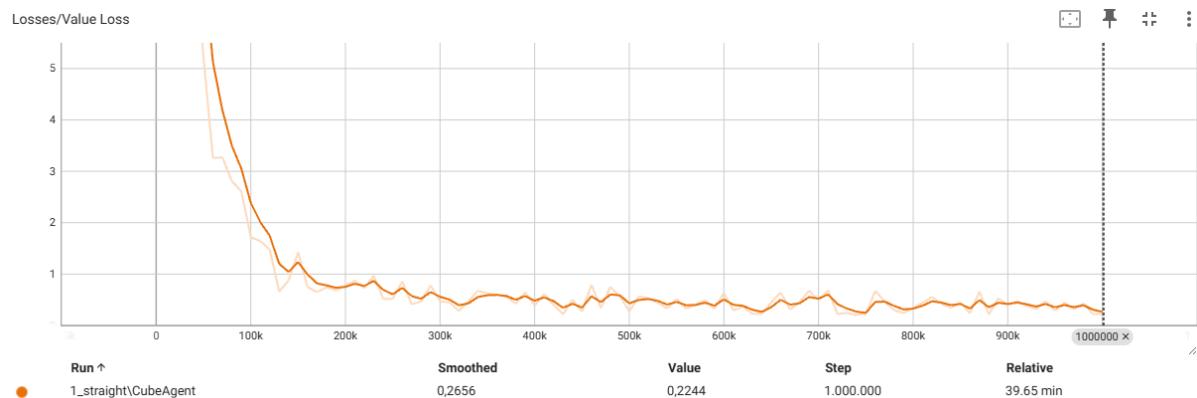


Figura 5.2: Value Loss del primo allenamento

La curva dell'entropia (fig. 5.3), invece, presenta una discesa graduale, suggerendo una progressiva riduzione dell'esplorazione man mano che l'agente converge verso comportamenti più deterministici e ottimizzati.

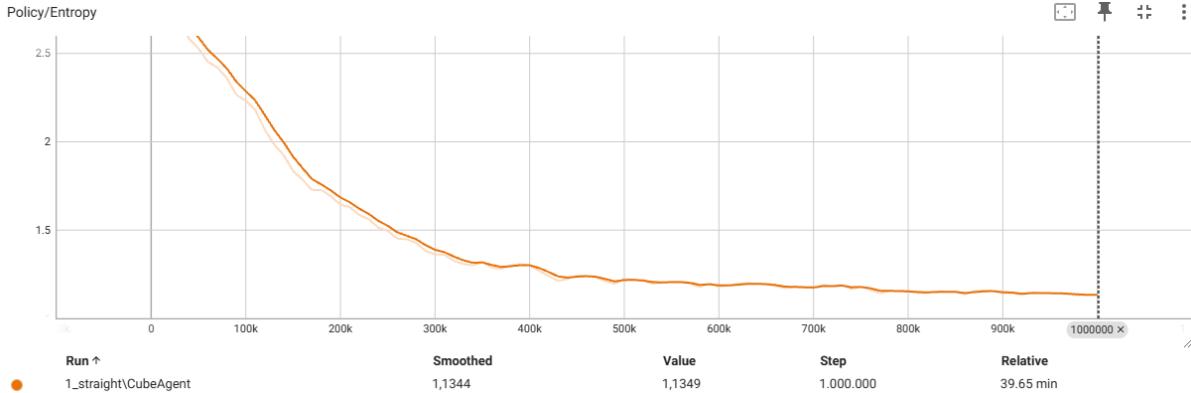


Figura 5.3: Entropia del primo allenamento

Analizzando i dati raccolti durante l'allenamento, relativi a circa **5000 episodi**, è possibile trarre diverse considerazioni per valutare la correttezza dell'andamento dell'agente. Poiché la posizione iniziale dell'agente e quella del target vengono assegnate casualmente all'inizio di ogni episodio, il *numero di celle visitate* varia in modo significativo da un episodio all'altro. Il coefficiente di correlazione calcolato, pari a -0.34 , indica che **non vi è una correlazione lineare tra il successo dell'episodio e il numero totale di celle visitate**. Questo risultato suggerisce correttamente che il raggiungimento dell'obiettivo (l'uscita) non dipende, in questo caso, direttamente dall'ampiezza dell'esplorazione, bensì dalla capacità dell'agente di individuare l'uscita all'interno dei percorsi.

5.2.2 Secondo allenamento: corridoio con ostacolo

Nella seconda fase di allenamento, l'obiettivo è quello di far imparare all'agente **a superare gli ostacoli nel percorso** (vedi 3.4.4), senza però dimenticare il comportamento acquisito nell'allenamento precedente, ovvero di raggiungere il target ed esplorare correttamente l'ambiente.

Il reward cumulativo converge più lentamente intorno a un valore di circa $+10$. Sebbene l'agente sia già in grado di muoversi all'interno del labirinto, ci si potrebbe aspettare che il reward medio iniziale sia pari a 10 . Tuttavia, ciò non avviene perché l'agente, nelle prime fasi dell'addestramento, non ha ancora appreso come gestire correttamente la presenza di ostacoli: tende quindi a scontrarsi con essi o a salirci sopra, causando la terminazione anticipata dell'episodio e ricevendo una penalità. Dal grafico (fig. 5.4) vediamo che a 270000 passi la curva comincia stabilizzarsi, significando che l'agente ha imparato una *policy* che gli permette di raggiungere il target superando gli ostacoli.

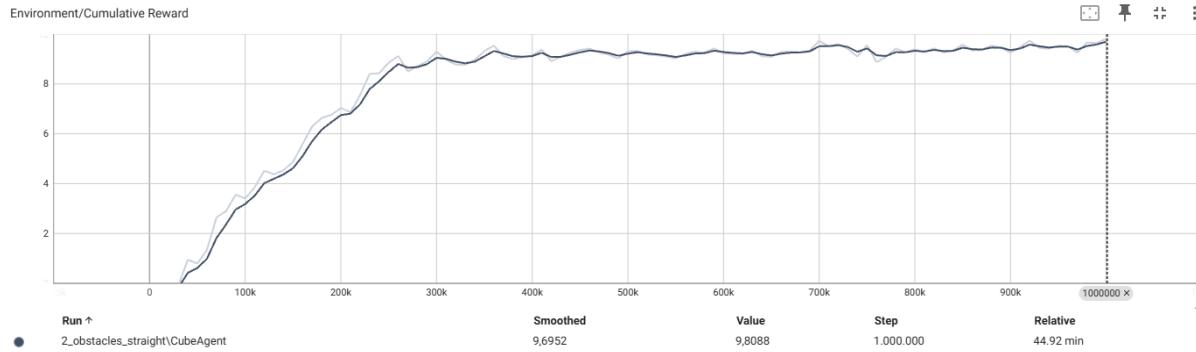


Figura 5.4: Reward cumulativo della seconda fase di allenamento

L’ambiente con ostacoli presenta una difficoltà maggiore e l’agente impiega più tempo a imparare buone stime del valore. Nonostante la *value loss* (fig. 5.5) cali nel tempo, rimane leggermente oscillante, indicando che potrebbe ancora migliorare la sua comprensione delle dinamiche dell’ambiente.

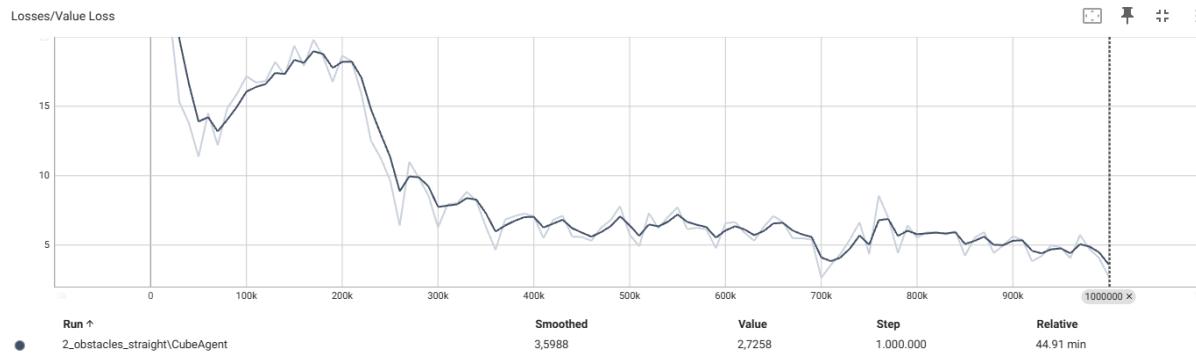


Figura 5.5: Value loss della seconda fase di allenamento

Per verificare se la policy appresa tende a un’ottima soluzione globale piuttosto che a un ottimo locale, analizziamo l’andamento dell’entropia (fig. 5.6) intorno ai 270000 step, punto in cui la curva del reward cumulativo converge. Da questo punto in poi, la curva dell’entropia inizia ad appiattirsi, indicando che la policy sta convergendo verso una strategia stabile, riducendo progressivamente la componente esplorativa.

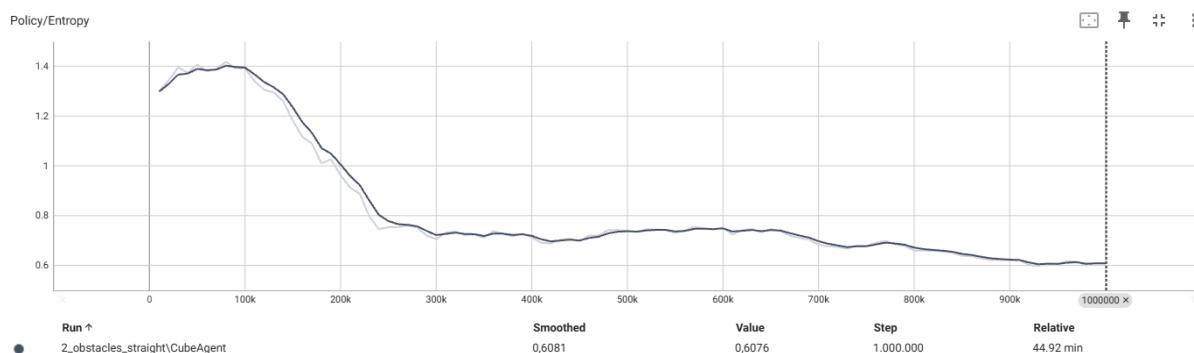


Figura 5.6: Entropia della seconda fase di allenamento

5.2.3 Terzo allenamento: labirinti 5x5 con 5 ostacoli

Nell'ultima fase dell'allenamento, l'agente deve dimostrare di saper **generalizzare la policy appresa**, applicando correttamente i salti non solo in ambienti semplici come un corridoio, ma anche all'interno di labirinti più complessi. In questi scenari, la difficoltà non risiede soltanto nella presenza di ostacoli, ma anche nella necessità di comprendere quando svoltare, come affrontare ostacoli disposti in modo alternato o posizionati agli angoli, e nel mantenere un comportamento esplorativo efficace.

In questo caso, il reward cumulativo (fig. 5.7) raggiunge un valore più basso rispetto agli altri, fermandosi intorno a 9,4. La curva cresce in modo molto più lento rispetto alle situazioni precedenti e non riesce a stabilizzarsi su un valore preciso. Si notano anche molte piccole oscillazioni, che, pur essendo di bassa intensità, indicano che l'agente ha maggiori difficoltà nel trovare una policy efficace. Questo comportamento si riflette anche nei grafici dell'entropia e della value loss.

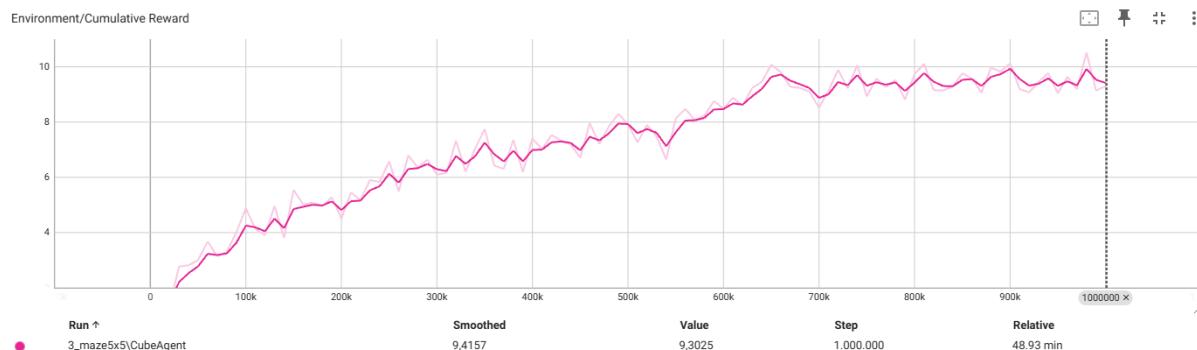


Figura 5.7: Reward cumulativo nella terza fase di allenamento

La value loss (fig. 5.8) mostra delle oscillazioni, a testimonianza del fatto che la stima della funzione di valore non è ancora sufficientemente accurata. L'agente, infatti, fatica a prevedere correttamente il valore atteso delle sue azioni, il che compromette la qualità del processo decisionale e, di conseguenza, l'efficacia della policy appresa.

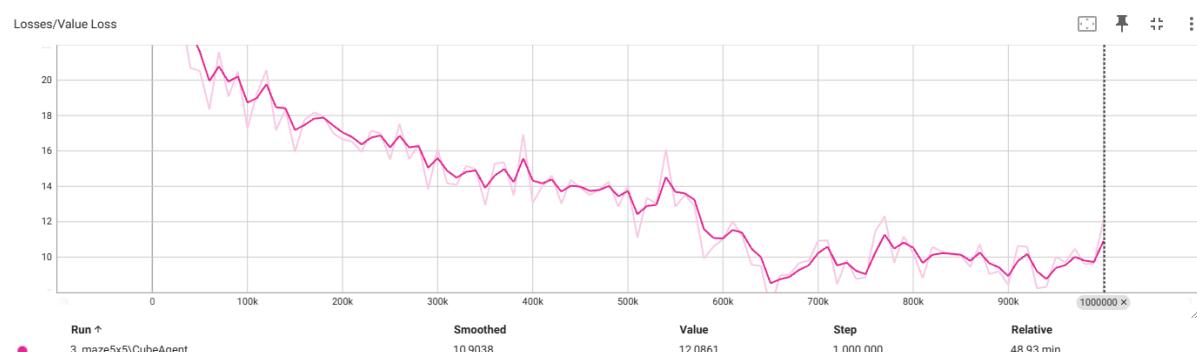


Figura 5.8: Value loss nella terza fase di allenamento

Anche l'entropia (fig. 5.9), pur mantenendosi su valori relativamente bassi, presenta comunque una certa variabilità: ciò indica che l'agente conserva ancora un certo grado di esplorazione e non è riuscito a stabilizzarsi completamente su una politica. Questo è coerente con la difficoltà nel convergere verso una soluzione efficace in un ambiente più

complesso.

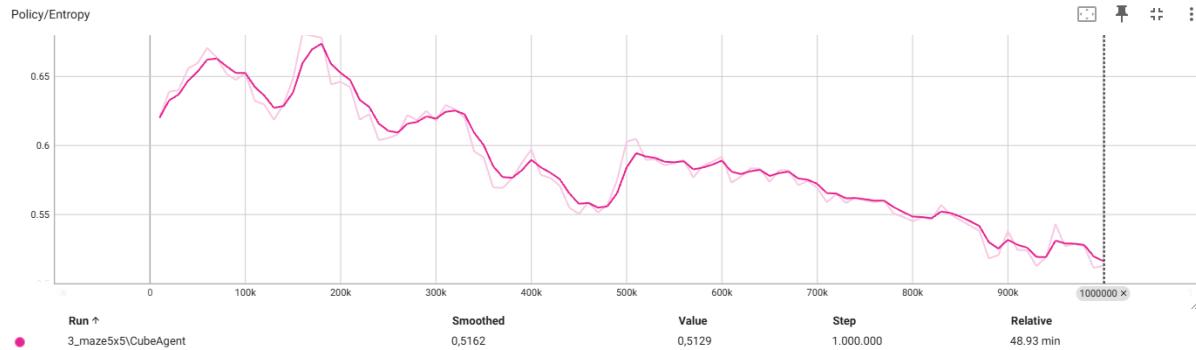


Figura 5.9: Entropia nella terza fase di allenamento

6

Risultati

Per ottenere il modello migliore sono stati necessari numerosi tentativi, durante i quali sono stati testati diversi sistemi di ricompensa e parametri della rete neurale. I risultati riportati nel seguente capitolo si riferiscono al miglior modello attualmente disponibile, addestrato utilizzando i parametri e le osservazioni descritti in precedenza.

6.1 Analisi della bontà del modello

Una volta completati i tre allenamenti, otteniamo un modello in grado di risolvere labirinti di diverse dimensioni. In particolare, verifichiamo se il modello è **in grado di generalizzare a labirinti di dimensioni maggiori** e se **ha sviluppato una strategia efficace di esplorazione**. La tabella seguente riporta le medie delle metriche raccolte su circa 5000 episodi, distinguendo tra episodi conclusi con successo, falliti e il totale, per ciascuna dimensione del labirinto:

- Labirinti 5x5 con 5 ostacoli (20 celle)
- Labirinti 7x7 con 7 ostacoli (42 celle)
- Labirinti 10x10 con 10 ostacoli (90 celle)

Tutte le colonne, escluse **"Reward"** e **"Successo"**, sono state normalizzate rispetto alla dimensione del labirinto, dividendo i valori per il numero totale di celle esplorabili (escludendo gli ostacoli). Il numero di **salti**, invece, è stato normalizzato rispetto al numero di ostacoli presenti nell'ambiente. Vengono qui riportate le corrispettive tabelle per i casi di successo, insuccesso e totale .

| Ambiente | Steps | Celle univoche | Celle rivisitate | Celle totali | Reward | Salti |
|-----------------|-------|----------------|------------------|--------------|--------|-------|
| Labirinti 5x5 | 22,42 | 0,44 | 0,17 | 0,63 | 9,55 | 2,00 |
| Labirinti 7x7 | 18,50 | 0,37 | 0,20 | 0,56 | 9,22 | 1,82 |
| Labirinti 10x10 | 15,32 | 0,31 | 0,18 | 0,49 | 8,61 | 1,64 |

Tabella 6.1: Metriche medie in caso di successo

| Ambiente | Steps | Celle univoche | Celle rivisitate | Celle totali | Reward | Salti |
|-----------------|-------|----------------|------------------|--------------|--------|-------|
| Labirinti 5x5 | 29,61 | 0,40 | 0,33 | 0,73 | -10,59 | 2,45 |
| Labirinti 7x7 | 25,24 | 0,36 | 0,46 | 0,82 | -10,41 | 2,83 |
| Labirinti 10x10 | 19,28 | 0,35 | 0,44 | 0,80 | -9,70 | 2,64 |

Tabella 6.2: Metriche medie in caso di insuccesso

| Ambiente | Success Rate | Steps | Celle univoche | Celle rivisitate | Celle totali | Reward | Salti |
|-----------------|--------------|-------|----------------|------------------|--------------|--------|-------|
| Labirinti 5x5 | 0,85 | 21,61 | 0,40 | 0,20 | 0,60 | 6,59 | 1,918 |
| Labirinti 7x7 | 0,81 | 19,57 | 0,36 | 0,25 | 0,61 | 5,53 | 1,99 |
| Labirinti 10x10 | 0,67 | 16,64 | 0,32 | 0,27 | 0,60 | 2,50 | 1,97 |

Tabella 6.3: Metriche delle performance complessive

6.1.1 Esplorazione del modello

Nei casi in cui l'agente non riesce a raggiungere il target, il numero medio di passi compiuti risulta **significativamente più alto** rispetto ai casi di successo. Verifichiamo se l'aumento di numeri di passi è correlato al numero di celle univoche visitate, per vedere se quando l'agente non trova l'uscita, continua ad esplorare celle nuove oppure quelle che ha già visitato. In questo modo, distinguiamo i casi in cui l'agente ha insuccesso perché **non fa in tempo a trovare l'uscita, entro gli steps limite** oppure il caso in cui **rimane bloccato in un loop esplorativo**, continuando a visitare celle già esplorate. Per comprendere meglio il comportamento, è stata analizzata la correlazione tra il numero di *steps* effettuati e due metriche di esplorazione:

- **Celle univoche visitate:** quantifica l'esplorazione in aree nuove.
- **Celle rivisitate:** quantifica la ridondanza del percorso (loop).

I dati sono stati suddivisi tra *episodi di successo* (l'agente raggiunge il target) e *insuccesso*, e riportati sui tre ambienti:

| Correlazione | | 5×5 | 7×7 | 10×10 |
|--------------------------------|--|------|------|-------|
| Celle univoche visitate | | | | |
| Insuccesso | | 0,66 | 0,66 | 0,54 |
| Successo | | 0,83 | 0,83 | 0,88 |
| Celle rivisitate | | | | |
| Insuccesso | | 0,96 | 0,67 | 0,47 |
| Successo | | 0,93 | 0,94 | 0,95 |

Tabella 6.4: Correlazioni tra numero di steps e celle visitate (univoche / rivisitate), nei casi di successo e insuccesso.

Analizzando le due correlazioni deduciamo che:

- Nel caso delle celle univoche, la correlazione positiva in entrambi i casi indica che un maggior **numero di steps porta naturalmente a visitare più zone**. Tuttavia, nei casi di **successo**, la correlazione è significativamente più elevata (fino a 0,88

su labirinti 10×10), suggerendo un'esplorazione più ampia. Nei fallimenti, la correlazione scende progressivamente con l'aumentare della dimensione dell'ambiente, indicando che l'agente esplora meno efficacemente **nuove celle**, non arrivando così all'uscita.

- Nel caso delle celle rivisitate, la correlazione rimane molto alta (0,93–0,95) nei successi, suggerendo che l'agente tende comunque a ripassare spesso su aree già esplorate per poi eventualmente raggiungere l'uscita. Nei fallimenti, invece, si osserva una correlazione molto alta nei labirinti più piccoli (0,93 su 5×5), che però diminuisce significativamente nei labirinti più grandi (0,47 su 10×10). Ciò indica che **nei contesti più ampi l'agente tende meno ad esplorazioni ridondanti** a causa dei corridoi più lunghi, ma comunque non riesce a raggiungere l'obiettivo.

I seguenti boxplot illustrano la distribuzione delle celle univoche visitate nei labirinti di dimensioni 5×5 , 7×7 e 10×10 . Osservando i grafici, si nota una forte somiglianza tra i diversi ambienti, in particolare tra i labirinti 5×5 e 7×7 . Questo indica che l'agente, nei casi di insuccesso, non fallisce per una scarsa esplorazione: il numero di **celle univocamente visitate risulta infatti molto simile tra gli episodi di successo e quelli di fallimento**. Nei labirinti 10×10 , invece, il numero di celle univoche visitate nei casi di insuccesso, al 50° percentile è leggermente superiore rispetto a quelli di successo. Questo avviene perché, in ambienti più grandi, **una singola direzione sbagliata può portare l'agente a esplorare lunghi percorsi inutili**, prolungando l'esplorazione e aumentando la probabilità di terminare l'episodio senza successo, pur avendo visitato molte celle differenti.

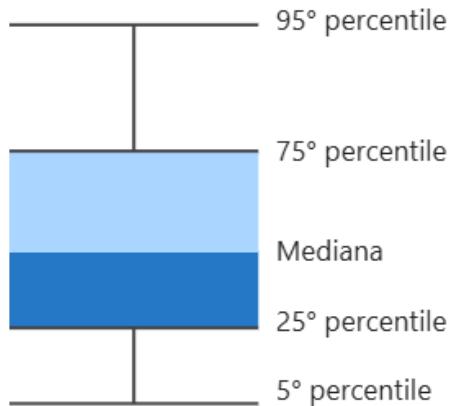


Figura 6.1: Legenda boxplot

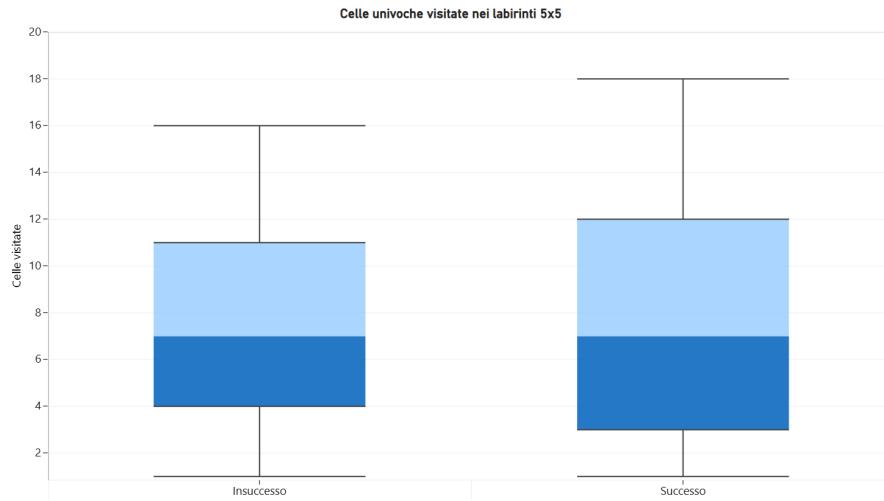


Figura 6.2: Boxplot nei casi di successo/insuccesso labirinti 5x5

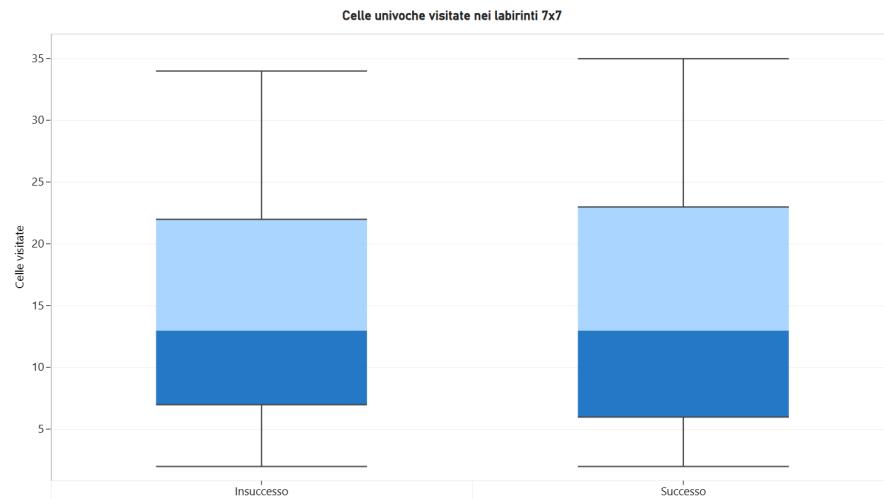


Figura 6.3: Boxplot nei casi di successo/insuccesso labirinti 7x7

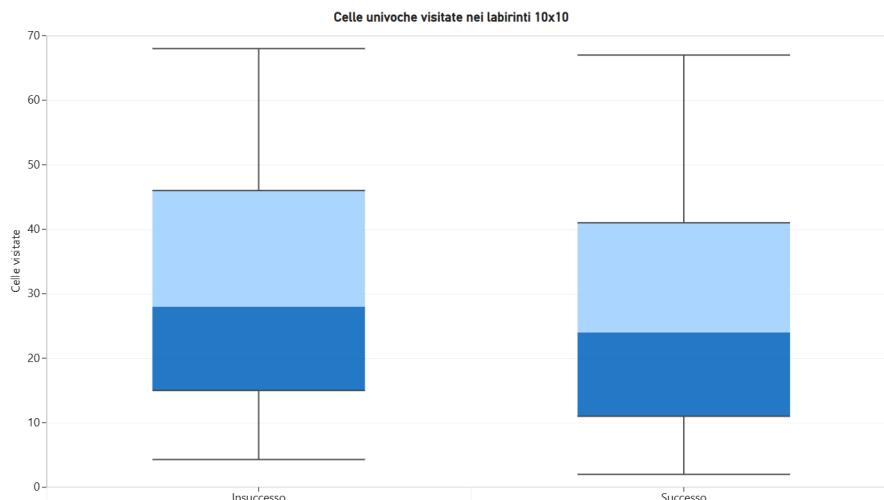


Figura 6.4: Boxplot nei casi di successo/insuccesso labirinti 10x10

Dal diagramma a torta 6.5, si osserva chiaramente come la quantità di celle rivisitate sia maggiormente concentrata nei casi di **insuccesso** rispetto a quelli di **successo**.

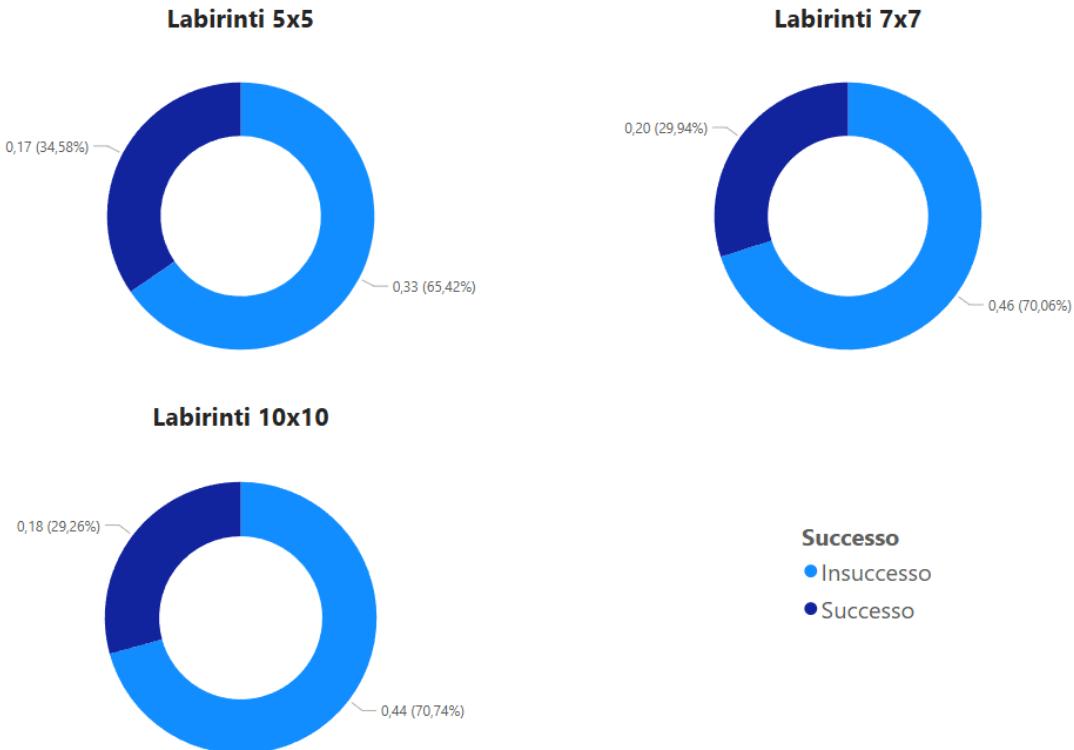


Figura 6.5: Diagrammi a torta tra la relazione dei successi e il numero di celle rivisitate

6.1.2 Superamento degli ostacoli e utilizzo del salto

Vogliamo verificare che il modello utilizzi il salto principalmente per superare ostacoli, e non in modo improprio. Nei casi di successo (fig. 6.1), si osserva che il **tasso di salti è inversamente proporzionale al numero di ostacoli presenti**. Questo dato non dipende unicamente dalla quantità di ostacoli, ma anche dalla capacità dell'agente di esplorare efficacemente l'ambiente, come analizzato precedentemente (sec. 6.1.3).

Dai seguenti scatterplot si può notare che, con l'aumentare della dimensione del labirinto, si manifestano più frequentemente episodi in cui **il numero di salti è elevato e la ricompensa cumulativa è alta**. Tali episodi non compaiono nei labirinti 5x5 (fig. 6.6), ma iniziano a manifestarsi nei 7x7 (fig. 6.7) e diventano più frequenti nei 10x10 (fig. 6.8). Questo comportamento suggerisce che l'agente, in alcuni casi, è riuscito a individuare **strategie alternative** per massimizzare la ricompensa piuttosto che raggiungere il target, saltando ripetutamente sopra ostacoli per ottenere ricompense continue, come previsto dal reward system "in aria sopra un ostacolo" (vedi tabella 5.3).

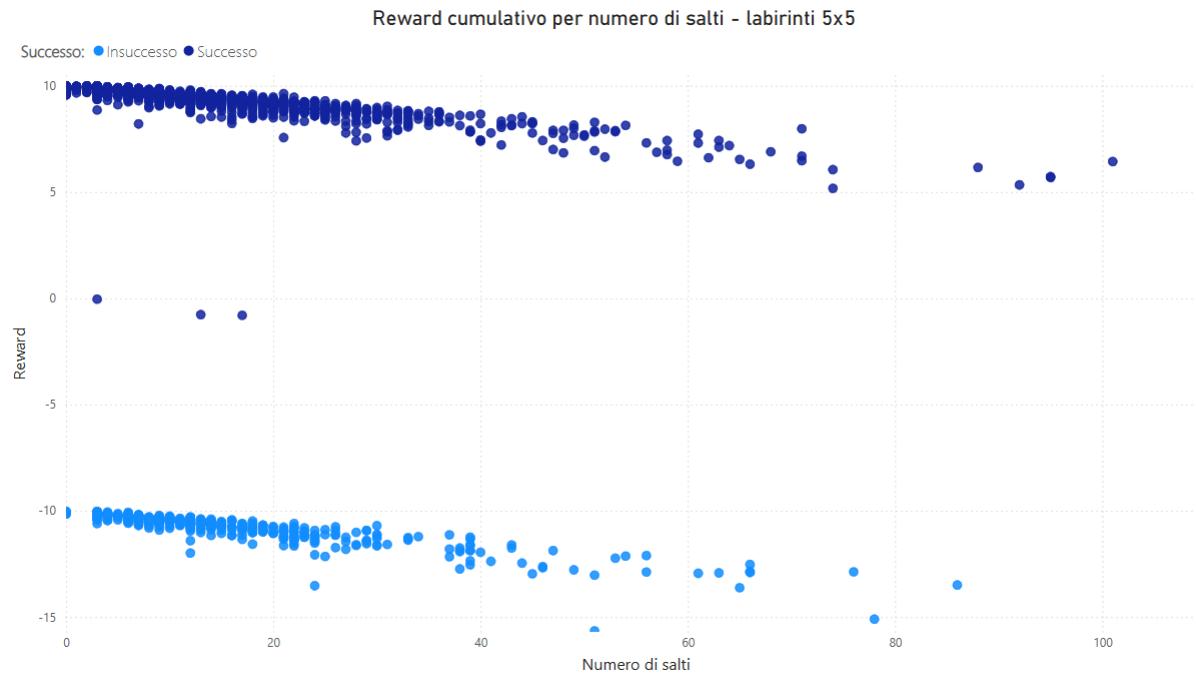


Figura 6.6: Relazione tra reward e numero di salti nei labirinti 5x5

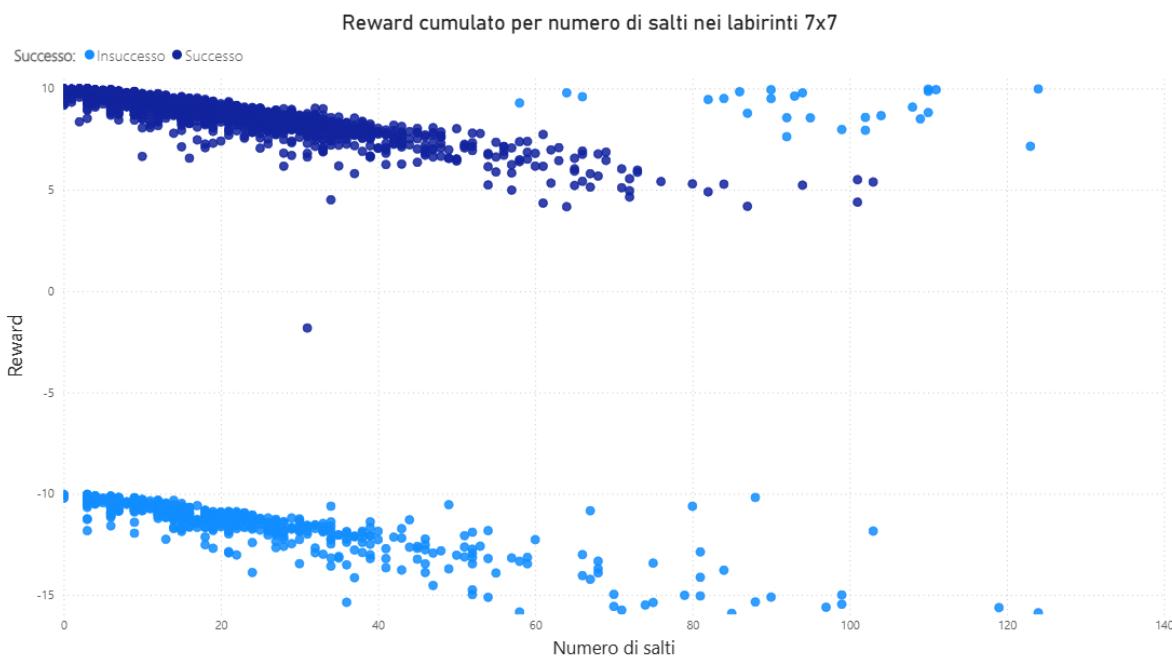


Figura 6.7: Relazione tra reward e numero di salti nei labirinti 7x7

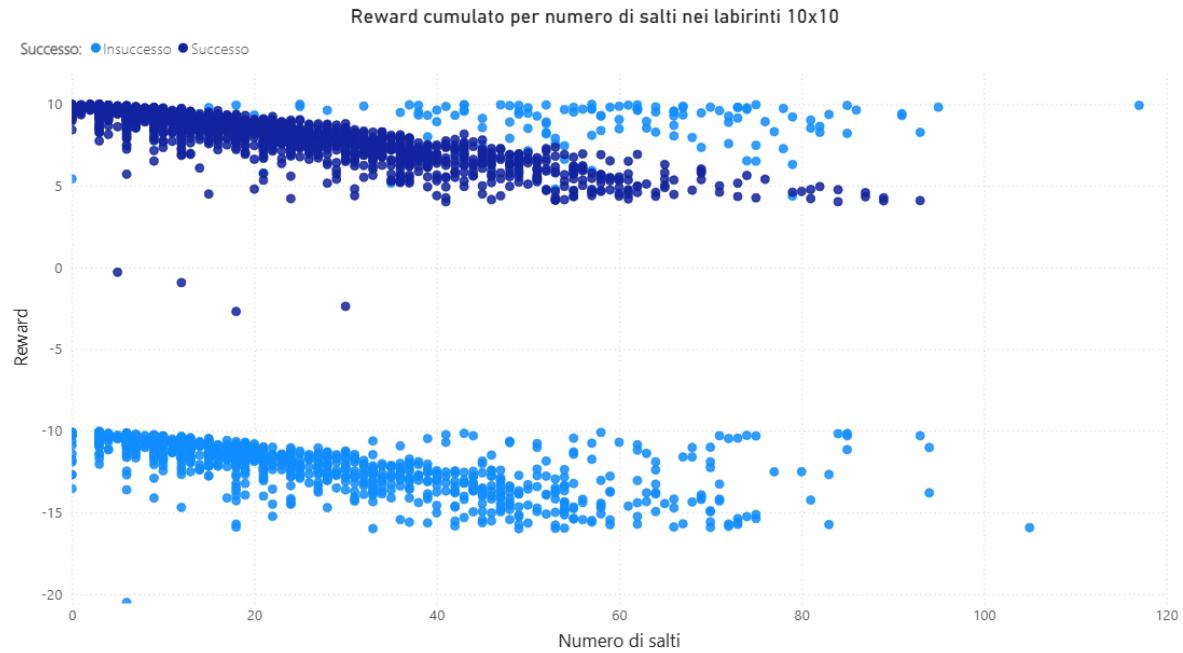


Figura 6.8: Relazione tra reward e numero di salti nei labirinti 10x10

6.1.3 Generalizzazione del modello

Per valutare la capacità di generalizzazione del modello nell'affrontare labirinti di diversa struttura e dimensione crescente, è stato effettuato un confronto tra le metriche normalizzate riportate nella tabella 6.3. I risultati mostrano che, nonostante l'incremento della dimensione dell'ambiente, il comportamento dell'agente rimane relativamente stabile in termini di numero di salti, celle totali visitate e distribuzione tra celle univoche e rivisitate. Tuttavia, dalla tabella si osserva che, **all'aumentare della dimensione del labirinto, il success rate tende a diminuire**. Questo indica che il modello ha maggiori difficoltà nell'affrontare ambienti più ampi e complessi, perché c'è più probabilità di incorrere in più percorsi dove rimanere in stato di loop esplorativo, e anche in diversi casi, continuare a saltare sopra un ostacolo per accumulare reward. Questo comportamento evidenzia i limiti della capacità di generalizzazione del modello, che riesce ad adattarsi bene a scenari più semplici, ma perde efficacia in contesti più articolati.

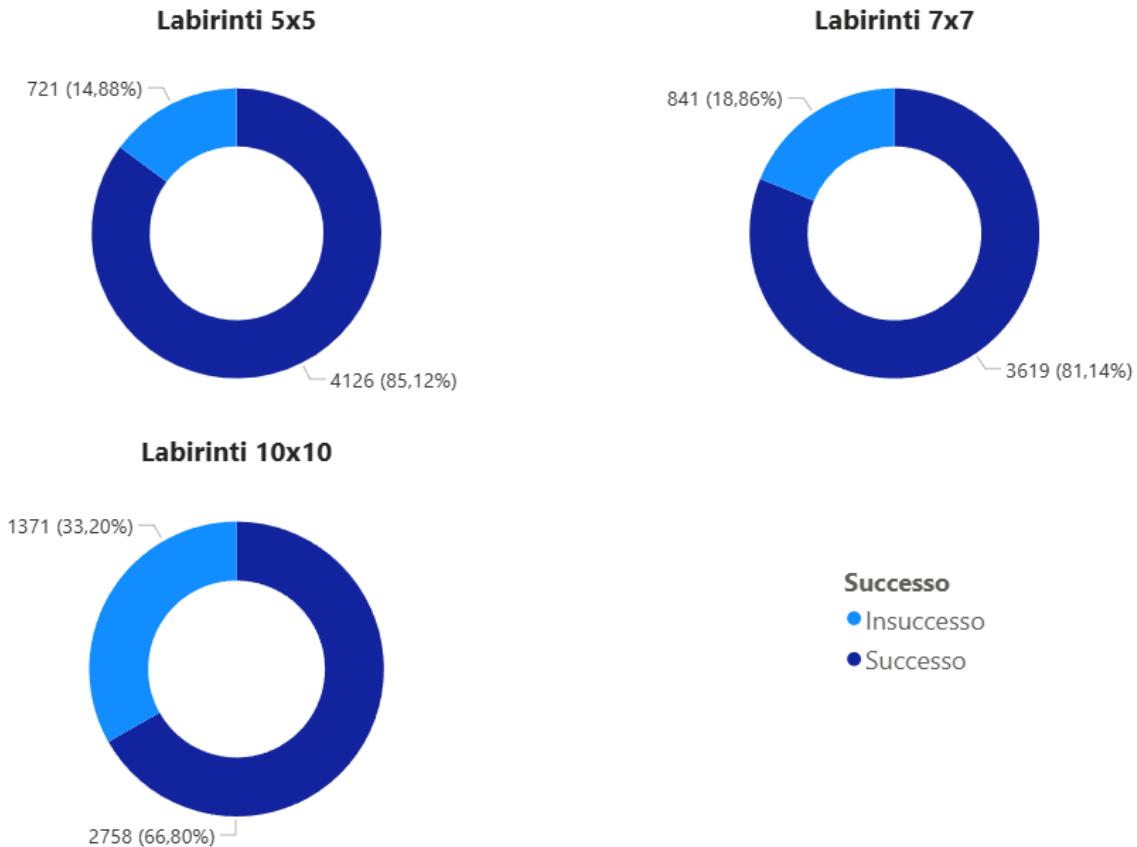


Figura 6.9: Diagrammi a torta tra il numero di casi di insuccesso e successo

7

Conclusioni e sviluppi futuri

In questo capitolo conclusivo vengono discusse le considerazioni finali riguardanti il modello allenato, valutandone i punti di forza e le criticità emerse. Si propongono infine possibili direzioni di miglioramento e sviluppi futuri per migliorarne le prestazioni e la capacità di generalizzazione.

7.1 Conclusioni

Il modello allenato dimostra una buona capacità nel risolvere labirinti di **piccole e medie dimensioni**, evitando gli ostacoli e raggiungendo il target al termine dell'esplorazione. Tuttavia, emergono alcune criticità quando il modello viene impiegato in **ambienti di dimensioni maggiori**. Con l'aumentare della grandezza del labirinto, infatti, l'agente tende spesso a percorrere ripetutamente gli stessi tratti, soprattutto a causa della presenza di numerosi vicoli ciechi. Questo comportamento deriva da una policy non ancora sufficientemente efficace nel riconoscere e distinguere le celle già visitate, con il risultato che l'agente può entrare in loop esplorativi. Inoltre, si osservano alcuni casi limite in cui l'agente esegue continui salti sopra agli ostacoli per massimizzare la ricompensa, comportamento incentivato dal sistema di reward attualmente adottato. Nonostante queste problematiche, complessivamente l'agente riesce a completare labirinti con diversi livelli di difficoltà e configurazioni di ostacoli, confermando la validità dell'approccio seguito in fase di allenamento.

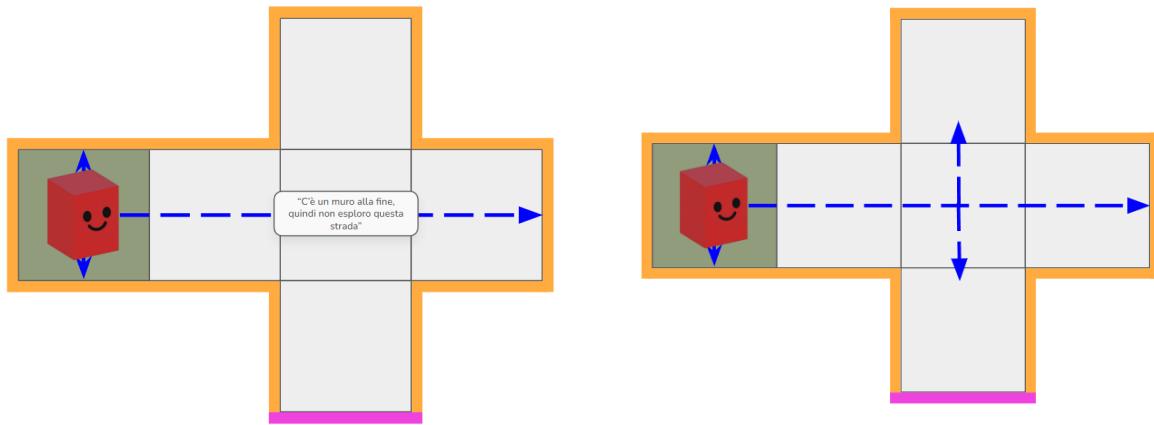
7.2 Sviluppi futuri

Tra le principali problematiche riscontrate, emerge una certa inefficienza nell'esplorazione, causata dal frequente movimento avanti e indietro tra celle già visitate. Oltre alla possibilità di intervenire sui parametri del *reward system* e sull'architettura della rete neurale, è possibile apportare miglioramenti significativi modificando lo spazio delle osservazioni e l'utilizzo dei `RayPerceptionSensor3d`

7.2.1 Bilanciamento tra esplorazione e raggiungimento dell'obiettivo

Un aspetto importante nella progettazione dell'ambiente percettivo dell'agente riguarda la configurazione dei sensori, in particolare la lunghezza dei raggi `RayPerceptionSensor3D`. Questi sensori permettono all'agente di raccogliere informazioni sull'ambiente circostante in base alla distanza percorsa dai raggi (vedi 3.3.5.1). Si può dedurre che una lunghezza eccessiva dei raggi può portare a comportamenti non corretti. Infatti, l'agente potrebbe ignorare obiettivi vicini non direttamente visibili, come ad esempio un target nascosto dietro l'angolo in un vicolo cieco. L'informazione rilevata da raggi più lunghi tende a privilegiare il raggiungimento del target se posizionato in una strada lunga, penalizzando però la percezione dettagliata di ciò che si trova nelle strade vicine in tutti gli altri casi.

Per migliorare la percezione spaziale dell'agente e guidare l'esplorazione in modo più efficiente, è possibile introdurre configurazioni alternative. Un buon compromesso consiste, ad esempio, nell'**aggiungere raggi disposti a forma di "L"**, in cui la parte finale è relativamente corta. Questo permette di rilevare la presenza di corridoi laterali che si trovano oltre l'ostacolo visibile frontalmente, suggerendo all'agente che l'ambiente si estende anche ai lati. In alternativa, è possibile includere tra le osservazioni dell'agente anche **informazioni logiche**, come ad esempio il numero di vicoli ciechi presenti lungo il cammino corrente.



(a) Esempio di bilanciamento non ottimale tra esplorazione e `RayPerceptionCastSensor3D` (b) Esempio di esplorazione con percezione bilanciata

Figura 7.1: Confronto tra diverse configurazioni di percezione spaziale

7.2.2 Miglioramento della memoria dell'agente

Un aspetto fondamentale per migliorare le prestazioni dell'agente nella risoluzione di labirinti complessi riguarda **l'implementazione di una memoria efficace**, che possa prevenire percorsi ridondanti e loop esplorativi. A questo proposito, si possono adottare due strategie principali. La prima consiste nell'integrare nel file di configurazione del training il modulo `memory`, che abilita l'uso di una rete neurale ricorrente. Questo tipo di rete mantiene uno stato interno, detto hidden state, tra i vari step temporali, permettendo così all'agente di "ricordare" informazioni importanti derivanti dalle osservazioni e dalle azioni compiute in precedenza. In questo modo, l'agente non si basa esclusivamente sull'osservazione attuale, ma anche sulle decisioni prese precedentemente.

La seconda strategia riguarda invece l'estensione dello spazio di osservazione fornito all'agente. Attualmente, l'agente raccoglie informazioni sulle quattro celle adiacenti a quella in cui si trova, tenendo conto sia del loro stato sia del numero di visite effettuate. Per aumentare la consapevolezza spaziale e fornire un quadro più completo dell'ambiente circostante, è possibile ampliare queste osservazioni **includendo anche le celle adiacenti a quelle già analizzate**. Questa espansione consente all'agente di ottenere una visione più ampia e dettagliata dell'area circostante, facilitando decisioni più efficaci e una migliore esplorazione del labirinto.

Riferimenti

Siti

- [1] *Algoritmi per la risoluzione di labirinti, A**. 2024. URL: https://it.wikipedia.org/wiki/Algoritmi_per_la_risoluzione_di_labirinti#Algoritmo_del_percorso_pi%C3%B9_breve (cit. a p. 6).
- [2] Abhishek Bhowmick. *Value-based vs Policy-based Reinforcement Learning*. 2021. URL: <https://papers-100-lines.medium.com/value-based-vs-policy-based-reinforcement-learning-92da766696fd> (cit. a p. 9).
- [3] *Class RayPerceptionSensor / ML Agents / 1.0.8*. 2024. URL: <https://docs.unity3d.com/Packages/com.unity.ml-agents@1.0/api/Unity.MLAgents.Sensors.RayPerceptionSensor.html> (cit. a p. 21).
- [4] GeeksforGeeks. *What is Reinforcement Learning?* 2025. URL: <https://www.geeksforgeeks.org/machine-learning/what-is-reinforcement-learning/> (cit. a p. 7).
- [5] HTML.it. *Prefab e istanze in Unity*. 2025. URL: <https://www.html.it/pag/44092/prefab-e-istanze-in-unity/> (cit. a p. 13).
- [6] IBM. *What is Reinforcement Learning?* n.d. URL: <https://www.ibm.com/think/topics/reinforcement-learning> (cit. a p. 8).
- [7] *Proximal Policy Optimization / OpenAI*. 2024. URL: <https://openai.com/index/openai-baselines-ppo/> (cit. a p. 9).
- [8] Unity Technologies. *BehaviorParameters*. 2025. URL: <https://docs.unity3d.com/Packages/com.unity.ml-agents@2.0/api/Unity.MLAgents.Policies.BehaviorParameters.html> (cit. a p. 18).
- [9] Unity Technologies. *DecisionRequester*. 2025. URL: <https://docs.unity3d.com/Packages/com.unity.ml-agents@1.0/api/Unity.MLAgents.DecisionRequester.html> (cit. a p. 20).
- [10] Unity Technologies. *ML-Agents Python API – EngineConfigurationChannel*. 2024. URL: <https://github.com/Unity-Technologies/ml-agents/docs/Python-API.md> (cit. a p. 17).
- [11] Amit Thakkar. *Reinforcement Learning with Curriculum Learning for Complex Tasks*. 2021. URL: <https://medium.com/@heyamit10/reinforcement-learning-with-curriculum-learning-for-complex-tasks-4f306470ed42> (cit. a p. 9).
- [12] Unity Discussions. *Package ML-Agents: How reward system works, observed parameters, statistics, manipulation, network co...* 2021. URL: <https://shorturl.at/UnX9D> (cit. a p. 37).

- [13] Unity Discussions. *Preparing Blender Model for Unity*. Unity Discussions. 2013. URL: <https://discussions.unity.com/t/preparing-blender-model-for-unity/616842/2> (cit. a p. 31).
- [14] Unity Technologies. *ML-Agents Toolkit Documentation*. 2023. URL: <https://docs.unity3d.com/Packages/com.unity.ml-agents@3.0/manual/index.html> (cit. a p. 10).
- [15] Unity Technologies. *ML-Agents Training Configuration File*. 2025. URL: <https://unity-technologies.github.io/ml-agents/Training-Configuration-File/> (cit. alle pp. 11, 35).
- [16] Unity Technologies. *ML-Agents Installation Guide*. 2025. URL: <https://unity-technologies.github.io/ml-agents/Installation/> (cit. a p. 12).
- [17] *What is Reinforcement Learning? - Reinforcement Learning Explained*. 2024. URL: <https://aws.amazon.com/what-is/reinforcement-learning/> (cit. a p. 7).
- [18] Wikipedia. *Maze generation algorithm – Randomized depth-first search*. 2025. URL: https://en.wikipedia.org/wiki/Maze_generation_algorithm#Randomized_depth-first_search (cit. alle pp. 13, 16).