Alma Mater Studiorum - Università di Bologna

Corso di Laurea in Ingegneria e Scienze informatiche

Integrazione di Agenti con Machine Learning per Dynamic Difficulty Adjustment (DDA) nei videogiochi multiplayer

Tesi di laurea magistrale

Presentata da:
Leo Marzoli

Relatore:
Chiar.mo Prof.
Andrea Omicini

Appello III
Anno Accademico 2024-2025

A chi ha creduto in me.

Abstract

Questa tesi esplora l'integrazione tra agenti intelligenti e tecniche di machine learning per implementare un sistema di *Dynamic Difficulty Adjustment* (DDA) in videogiochi multiplayer, con particolare attenzione alla sua applicazione in Unity. L'obiettivo principale è stato sviluppare un ambiente di gioco dinamico e bilanciato, in cui l'adattamento automatico della difficoltà contribuisca a migliorare l'esperienza dei giocatori.

Nella prima parte della tesi, viene introdotto il contesto dei motori di gioco, con un'analisi comparativa tra i principali strumenti disponibili, e viene giustificata la scelta di Unity come piattaforma di sviluppo. Successivamente, vengono approfonditi gli aspetti tecnici relativi alla programmazione in C# e alle funzionalità offerte da Unity, come le richieste web (*Unity Web Request*) e il pacchetto *Mirror* per lo sviluppo di videogiochi multiplayer.

La seconda parte si concentra sull'architettura BDI (*Belief-Desire-Intention*), utilizzata per gestire agenti intelligenti all'interno del sistema. In particolare, vengono analizzati i linguaggi di programmazione per agenti, come JASON, e la loro integrazione in un ambiente Java, con un focus sull'interazione tra gli agenti e l'ambiente di gioco tramite API REST.

La terza parte introduce le tecniche di machine learning, con una descrizione dei modelli utilizzati, delle metriche di valutazione e del processo di training. Il modello è stato progettato per prevedere situazioni di svantaggio nel gameplay, consentendo agli agenti di adattare dinamicamente le regole del gioco per bilanciare la competizione.

I risultati ottenuti dimostrano l'efficacia dell'approccio proposto nel migliorare l'equità e l'engagement dei giocatori. Inoltre, il sistema si è dimostrato robusto e flessibile, con buone prestazioni in termini di accuratezza delle predizioni e tempi di risposta. Le conclusioni evidenziano il potenziale di questa integrazione per lo sviluppo di nuove esperienze di gioco, mentre gli sviluppi futuri includono potenziali espansioni del sistema in modo da ricoprire futuri scenari più complessi.

Questa ricerca rappresenta un passo significativo verso l'adozione di tecnologie intelligenti nel settore dei videogiochi e della programmazione ad agenti, fornendo un framework versatile e innovativo per la creazione di giochi adattivi, coinvolgenti e una base di sviluppo per quello che è il collegamento tra agenti e machine learning.

Indice

Capitolo 1

Introduzione ai videogiochi

1.1 Motori di gioco, quale scegliere?

> «*Un Game Engine è un ambiente di sviluppo integrato che, fornendo un livello di astrazione sufficiente attraverso interfacce grafiche e tutta una serie di librerie software, rende più semplice il processo di creazione di un videogioco. I motori di gioco possono essere responsabili del sistema di input, della logica di un gioco, del rendering della grafica, delle leggi fisiche e del rilevamento delle collisioni, della gestione della memoria, dell'intelligenza artificiale che risponde alle azioni del giocatore e di molte altre funzionalità.*» (DB Game Academy, 2024)

I motori di gioco forniscono componenti riutilizzabili che gli sviluppatori utilizzano per la creazione di videogiochi e sono frequentemente impiegati nello sviluppo di più titoli, rappresentando così un investimento valido nel lungo periodo. La realizzazione di un videogioco, o anche solo di una parte del suo mondo, coinvolge numerosi aspetti, e i Game Engine risultano fondamentali per ottimizzare i tempi e facilitare il conseguimento degli obiettivi di sviluppo.

Attualmente, i Game Engine disponibili sul mercato possono essere sufficienti per la creazione di un videogioco; tuttavia, per progetti di grande portata e ambizione, potrebbe rendersi necessaria la costruzione di un motore di gioco personalizzato da zero. È fondamentale distinguere tra Game Engine proprietari e quelli di terze parti.

Un Game Engine proprietario, o sviluppato in-house, è un motore di gioco appartenente a uno studio di sviluppo o a un publisher. Sono software che vengono creati da zero internamente per rispondere in modo più puntuale alle esigenze del team e del progetto, ma soprattutto non prevedono il pagamento di un canone di licenza, che in caso di giochi che venderanno molte copie è un fattore economico da non sottovalutare. Poiché i motori di gioco richiedono tempo per essere sviluppati, personale specializzato e possono avere costi molto elevati, molti studi di sviluppo si rivolgono ad aziende di terze parti che creano motori di gioco appositamente per concederli in licenza ad altri studi. Per questi motivi, si è optato per un motore di gioco di terze parti e nelle prossima sezione si analizzerà in dettaglio questo argomento esponendo quelli che sono i principali motori di gioco sul mercato.

1.1.1 Unreal Engine

Unreal Engine è un motore di gioco multipiattaforma sviluppato da Epic Games ed è tra i più usati nel settore a livello professionale perché, come Unity, permette di creare la stessa esperienza di gioco su console, PC e piattaforme mobile includendo anche software per la Realtà Estesa (AR, VR e MR). La prima versione dell'engine è stata distribuita nel 1998 per il famoso sparatutto in prima persona Unreal da cui prende il nome.

Unreal Engine è dotato di un potente script visivo integrato chiamato Blueprint che permette di utilizzare un'interfaccia basata su nodi per creare elementi di gioco direttamente dentro l'editor. Il motore può essere usato anche con C++, infatti le classi C++ possono essere usate come base per le classi Blueprint. Il gioco di punta di Epic Games è il fenomeno mondiale Fortnite, ma con Unreal sono stati creati anche giochi tripla A come le serie di Gears of War e Borderlands e giochi indipendenti come Kena: Bridge of Spirits e Omno.

1.1.2 Gotdot

Godot è un motore di gioco open source multipiattaforma che può essere utilizzato per creare giochi 2D e 3D da pubblicare su piattaforme desktop, mobili, console e web. Poiché il motore di gioco è open source, gli aggiornamenti sono spesso mirati all'aggiunta di funzionalità e correzione dei problemi, inoltre può contare su un ampio supporto da parte della community. Nello sviluppo indipendente è un engine molto apprezzato e in crescita, si trova infatti nella top 5 dei motori più utilizzati su Itch.io.

Godot permette di combinare linguaggi di scripting secondo le proprie esigenze. Oltre allo scripting visivo, utilizza principalmente GDScript, un linguaggio di programmazione di alto livello con una sintassi simile a Python. Con la tecnologia GDNative, il motore interagisce con librerie condivise ed esegue codice C o C++. Tra i giochi creati con Godot ci sono Dungeondraft, Resolutiion e il gioco in sviluppo The Garden Path.

1.1.3 CRYENGINE

CryEngine è un motore sviluppato da Crytek, una società che ha sempre creato prodotti che spingono le capacità tecniche nel settore dei videogiochi. La prima versione del CryEngine è stata utilizzata per lo sparatutto Far Cry. Da una versione molto modificata del CryEngine è nato anche il Dunia Engine, che Ubisoft attualmente usa per la serie Far Cry. Il CryEngine è un motore gratuito basato su un sistema di royalties, e può essere usato per creare giochi complessi su diverse piattaforme come Xbox One, PlayStation 4, PC Windows e Oculus Rift.

Tra le sue principali caratteristiche ci sono strumenti potenti per gestire l'illuminazione e la fisica in modo realistico. Offre sistemi avanzati di animazione e

rendering per creare personaggi realistici. Inoltre, ha una tecnologia AI che utilizza sistemi sensoriali modulari come udito e vista per gestire i comportamenti dei personaggi non giocanti. Per estendere le funzionalità tramite scripting bisogna conoscere LUA. Alcuni titoli creati con il CryEngine sono Sniper Ghost Warrior 3, Kingdom Come: Deliverance e Crysis 3.

1.2 Unity

Nonostante le caratteristiche degli altri motori di gioco si è optato per utilizzare Unity, ma prima di evidenziare i motivi principali di questa scelta parliamo in generale di come nasce questo motore di gioco:

> «*Unity è un Game Engine multipiattaforma sviluppato da Unity Technologies, annunciato e pubblicato per la prima volta nel giugno 2005 alla Apple Worldwide Developers Conference come motore di gioco esclusivo per Mac OS X. Dal 2018 è stato ampliato per supportare più di 25 piattaforme, tra cui Nintendo Switch. Il motore è molto flessibile e può essere utilizzato per creare giochi 3D, 2D, VR e AR, nonché simulazioni e altre esperienze come film e cinematiche.*» (HTML.it, 2024)

Gli aggiornamenti frequenti aggiungono sempre nuove funzionalità, che nella maggior parte dei casi sono sufficienti per creare videogiochi ricchi di dettagli. È una valida opzione per chiunque voglia iniziare a creare videogiochi grazie anche a una serie di tutorial online e progetti già pronti per essere scomposti e modificati, così da imparare le basi di un platform o uno sparatutto in prima persona.

Unity ti permette di creare componenti di gioco con gli script. Con questi file, puoi modificare le proprietà dei componenti, aggiungere interazioni e rispondere agli input degli utenti. Unity usa C# come linguaggio di programmazione nativo. È un linguaggio molto usato in diversi campi, simile a C++ e Java, poiché anch'esso fa parte del paradigma ad oggetti e Unity cerca proprio di replicare l'oggetto nel paradigma come un gameObject modificabile nel motore di gioco seguendo la precedentemente citata architettura a componenti. Alcuni giochi di successo creati con Unity sono Hearthstone, Hollow Knight, Beat Saber e Ori and the Blind Forest.

1.2.1 Perché proprio Unity?

Per scegliere Unity tra i vari game engine disponibili, ho valutato alcuni elementi essenziali. Unity permette l'integrazione con JNI per utilizzare Java, consentendo di scrivere codice Java e trasformarlo in DLL, facilitando il richiamo delle funzioni dell'agente nel game loop in C#.

Inoltre, Unity offre vari framework per il networking, con cui si è voluto effettuare degli esperimenti, e ha una gestione della compilazione più leggera, sia in termini di risorse sia di tempi il ché lo rende eseguibile facilmente su qualsiasi hardware

facilitando la distribuzione del videogioco in quanto multiplayer . Questo è vantaggioso rispetto a Unreal Engine, dato che consente di creare build standalone e di ridurre i tempi di testing. Se confrontato con Godot, Unity è simile per leggerezza, ma conoscendo già Unity, l'apprendimento di Godot avrebbe richiesto più tempo, anche se l'esperienza nella programmazione videoludica fa sempre comodo.

1.3 Linguaggio di programmazione C# in Unity

Unity permette l'uso di tre linguaggi di programmazione: C#, Javascript e Boo. Si possono ottenere risultati simili con ognuno di questi linguaggi per quanto riguarda gli elementi di Unity. In altre parole, se una funzionalità può essere creata in uno dei tre linguaggi, è accessibile anche negli altri due.

La differenza principale è nello stile dei linguaggi. Chi preferisce un linguaggio con un forte typing, anche se un po' più lungo, troverà C# soddisfacente. Chi già conosce JavaScript può scegliere questo linguaggio. Infine, per quanto riguarda Boo essendo un linguaggio di nicchia è quello che viene utilizzato di meno.

All'interno di questo contesto la scelta nell'utilizzo di questo linguaggio di programmazione è stata ulteriormente favorita in vista dello sviluppo dell'agente le cui tecnologie seguono sempre un paradigma ad oggetti più orientato sul linguaggio Java. Per non creare troppa eterogeneità nel sistema di per sè già complesso, si è mantenuto questo principio.

Detto ciò, bisogna anche specificare che il linguaggio C# sviluppato in ambiente Unity assume una stesura differente dal comune e dovendola descrivere in sintesi, gli script all'interno di Unity sono classi che derivano da Monobehavior, diventando così dei gameObject. Una classe rappresenta una categoria di oggetti, fisici o virtuali, con proprietà comuni.

Ogni classe è in un file di testo (nel nostro caso, un file con estensione .cs), che include anche tutte le proprietà degli oggetti derivanti. La definizione della classe comprende tutte le funzioni (metodi) associate agli oggetti della specifica classe.

Poiché tutti gli script derivano da Monobehavior, esistono funzioni invocate automaticamente da Unity in risposta a eventi. Tali funzioni permettono di strutturare il flusso del gioco e di reagire a azioni come l'interazione tra due oggetti o la disattivazione di un oggetto. Consideriamo un esempio pratico.

Quando parliamo di classe, intendiamo la definizione che stabilisce le caratteristiche condivise dai vari oggetti, delineando funzioni e proprietà. È utile pensare a queste funzioni e proprietà come statiche; esse concernono la classe stessa e non i singoli oggetti.

Quando usiamo i termini "oggetto" o "istanza", ci riferiamo a un elemento figlio della classe che ha proprie proprietà con valori distintivi rispetto agli altri elementi della stessa classe. Di seguito si riporta lo sviluppo di uno script minimale che opera in Unity.

```
using UnityEngine;
using System.Collections;
public class MyScript : MonoBehaviour {
        // Use this for initialization
        void Start () { }
        // Update is called once per frame
        void Update () { }
}
```

Codice 1.1: Contenuto di uno script che assume il ruolo di gameObject in Unity

1.4 Unity Web Request

> «*Unity Web Request rappresenta una classe altamente versatile all'interno dell'ambiente Unity, consentendo l'invio e la ricezione di dati tra client e server web. Essa sostituisce l'ormai obsoleta classe WWW, offrendo un livello superiore di flessibilità e controllo sulle richieste HTTP. La classe supporta diverse tipologie di richieste, quali GET, POST, PUT e DELETE, risultando particolarmente adatta per gestire interazioni complesse come autenticazioni, download di asset, trasmissione di statistiche di gioco e comunicazione in tempo reale con i server.*» (Unity Technologies, 2024a)

Tra gli aspetti principali di questa tecnologia, si evidenziano:

- Semplicità d'uso: Unity Web Request facilita le operazioni asincrone, permettendo la gestione della comunicazione in parallelo al ciclo di gioco senza compromettere le operazioni di rendering e aggiornamento.
- Download/Upload di dati: È possibile inviare e ricevere vari tipi di contenuti, inclusi stringhe, JSON, immagini e audio.
- Callback e coroutine: Questa classe è frequentemente utilizzata in combinazione con coroutine per un aggiornamento fluido dell'interfaccia utente o per il caricamento dei contenuti in background, migliorando così l'esperienza complessiva dell'utente.
- Gestione degli errori: Fornisce strumenti utili per verificare la validità delle richieste e delle risposte ricevute, accompagnati da messaggi dettagliati in caso di errori.

Questi elementi rendono Unity Web Request uno strumento potente ed efficace per la comunicazione client-server, fornendo il controllo necessario per integrare in modo ottimale componenti di rete all'interno di un progetto sviluppato con Unity.

La sua integrazione nello sviluppo del progetto è stata cruciale in quanto inizialmente si era pensato di integrare l'agente direttamente nel motore di gioco Unity, ma

questo chiaramente avrebbe apportato degli svantaggi essendo il videogioco distribuito questo doveva essere obbligatoriamente scalabile, infatti sviluppare un agente il cui scopo è quello di utilizzare tecniche di machine learning per applicare un bilanciamento sul gioco in tempo reale e distribuirlo su ciascuna istanza del videogioco avrebbe appesantito notevolemente sia le performance del gioco stesso, ma anche dell'hardware locale. Per questo motivo si è dovuto esternalizzare e rendere modulare ciascun aspetto del sistema suddividendolo in tre servizi: Unity, agente e machine learning. E tutte le comunicazioni tra di essi devono avvenire tramite Web request/response. In seguito verranno esposti i modi in cui questa cosa si è poi sviluppata.

1.5 Multiplayer Videogames in Unity

I videogiochi multiplayer, da un punto di vista tecnico, presentano una serie di sfide e considerazioni fondamentali per garantire una comunicazione efficace tra i giocatori e un'esperienza di gioco priva di interruzioni. I principali aspetti da considerare includono il networking, la sincronizzazione dello stato di gioco nonché la sua replicazione sui vari dispositivi, la gestione delle latenze e delle interpolazioni per compensare le variazioni della rete, nonché la sicurezza e la scalabilità dell'infrastruttura. (Unity Technologies, 2024b)

I videogiochi multiplayer seguono tre principali architetture:

- **Client-server**: Questo rappresenta uno dei modelli più diffusi, in cui un server centrale è responsabile della gestione del gioco e tutti i client si connettono ad esso per sincronizzare le azioni e ricevere gli aggiornamenti. Tale approccio facilita l'amministrazione dei dati e delle decisioni, ma richiede un server robusto in grado di gestire il volume di traffico.
- **Peer-to-peer (P2P)**: In questo modello, i client scambiano direttamente i dati tra loro. È particolarmente adatto per giochi meno competitivi, poiché presenta potenziali problematiche relative alla sincronizzazione e alla sicurezza.
- **Server dedicato vs. Hosting**: Alcuni giochi adottano server dedicati al fine di ridurre la latenza e migliorare la stabilità del servizio, mentre altri consentono ai giocatori di ospitare le partite sui propri dispositivi, comportando così un risparmio sui costi del server ma con una maggiore variabilità nelle prestazioni.

Per questo progetto si è deciso di ricorrere ad un package esterno sviluppato da terzi che prende il nome di Mirror, basato su una architettura client-server.

1.5.1 Package "Mirror"

> «*Mirror rappresenta una libreria open-source specificatamente progettata per il networking e la creazione di giochi multiplayer su Unity.*» (Mirror Networking, 2024)

Essa ha guadagnato ampia diffusione come alternativa e successore non ufficiale del precedente sistema UNet, offrendo molte delle sue funzionalità con significativi miglioramenti. Le sue principali caratteristiche sono:

- Facilità di implementazione: Mirror presenta un'interfaccia intuitiva per gli sviluppatori di Unity, consentendo una rapida configurazione delle funzionalità multiplayer mediante componenti di rete predefiniti. Gli sviluppatori possono così realizzare giochi multiplayer senza dover affrontare la complessità intrinseca dei protocolli di rete.
- Server Authority e sincronizzazione: Mirror adotta un modello basato sull'autorità del server, implicando che tutte le decisioni e le dinamiche di gioco siano gestite da un server centrale. Questo approccio assicura una maggiore sicurezza e coerenza, poiché il server è in grado di autenticare e validare ogni azione intrapresa.

  La sincronizzazione degli oggetti di gioco tra i client è gestita in modo automatico attraverso NetworkIdentity e NetworkTransform, i quali si occupano della sincronizzazione delle proprietà degli oggetti, inclusi posizione, rotazione e stato, in tempo reale.
- Supporto alla Scalabilità: Mirror consente l'implementazione di server dedicati e fornisce strumenti per ottimizzare il carico di rete mediante la compressione dei dati e l'ottimizzazione della larghezza di banda. Inoltre, offre la possibilità di configurare server per giochi di maggiori dimensioni o per esperienze ludiche con un numero elevato di giocatori simultanei, grazie alla sua compatibilità con infrastrutture server scalabili come quelle basate su Docker o Amazon Web Services (AWS).
- Supporto al Cross-Platform: Mirror è compatibile con la maggior parte delle piattaforme supportate da Unity, consentendo lo sviluppo di giochi cross-platform senza necessità di modifiche al codice per ogni sistema operativo.

Mentre le principali componenti per poterlo utilizzare sono:

- **NetworkManager**: rappresenta il componente centrale che coordina l'intera rete. Esso gestisce il ciclo di vita della connessione, dall'avvio del server alla gestione dei client connessi. Il NetworkManager consente inoltre la configurazione delle impostazioni di rete, quali la porta di ascolto del server, il numero massimo di giocatori e l'host di connessione. Oltre a occuparsi della gestione delle connessioni, facilita anche il matchmaking e la sincronizzazione delle variabili globali, rendendo Mirror facilmente scalabile su diverse piattaforme.

- **NetworkBehaviour**: estende le capacità del MonoBehaviour di Unity, fornendo metodi preconfigurati per la gestione della rete che si occupano di eventi e sincronizzazione delle variabili. Gli sviluppatori hanno la possibilità di integrare il NetworkBehaviour negli oggetti che necessitano di essere sincronizzati, ad esempio per gestire posizioni e rotazioni. Alcune delle funzioni associate al NetworkBehaviour includono:
  - Command: consente ai client di inviare comandi al server.
  - ClientRpc: permette al server di trasmettere istruzioni a tutti i client.
  - TargetRpc: consente al server di inviare un comando specifico a un singolo client, risultando particolarmente utile per la personalizzazione degli eventi privati.
- **NetworkMessage**: Mirror consente una comunicazione altamente personalizzata attraverso l'utilizzo di essi, i quali rappresentano pacchetti di dati specifici scambiati tra client e server. Tale modalità comunicativa offre una flessibilità superiore nella gestione degli eventi di gioco, permettendo l'invio di informazioni aggiuntive quali statistiche di gioco o eventi particolari tra i giocatori.
- **NetworkTransform**: E' un componente fondamentale per la gestione della sincronizzazione della posizione, rotazione e scala degli oggetti all'interno di un ambiente di gioco, sia dal lato client che server. La sua utilità si manifesta in particolare nella necessità di mantenere la coerenza degli oggetti in tempo reale, prevenendo così discrepanze nelle posizioni. Questo sistema impiega tecniche di compressione dei dati e interpolazione al fine di minimizzare la latenza percepita e garantire una coerenza visiva tra i vari client.

1.6 Sviluppo del gioco in ottica agenti

Questa sezione tratta nello specifico quali sono state le idee che hanno portato alla scelta di un videogioco specifico che poi avrebbe dovuto condividere la sua logica con una sviluppata ad agenti.

Per la riuscita del progetto era necessario disporre di un videogioco Unity di piccole dimensioni che fosse distribuito e le cui meccaniche di gioco producessero sufficienti dati da poter essere raggrupati ed inviati all'agente per futura elaborazione. Inoltre, il gioco non poteva essere banale, ma necessariamente doveva disporre di un contesto modificabile per cui applicare il bilanciamento del gioco nel momento in cui l'agente avesse fornito tale indicazione.

Tanti erano i videogiochi che ricadevano entro questi limiti, tra i più favorevoli c'erano le seguenti categorie: MOBA, FPS, giochi co-operativi e kartgame. Non a caso si è scelto quest'ultimo.

Si è sviluppato un mini kartgame prendendo ispirazione da un learning game che Unity stesso nella versione 2021.3 LTS mette a disposizione e lo si è esteso nel seguente modo:

- Implementazione del package Mirror per renderlo distribuito con conseguente adattamento al nuovo gioco impostando le varie componenti precedentemente descritte nella sezione dedicata a Mirror.
- Creazione di una nuova mappa su cui gareggiare.
- Aggiunta di ostacoli con effetti, come: aumento della velocità dei giocatori, incremento del tempo di gara o potenziali rallentamenti.
- Aggiunta di GameObject che recuperassero i dati del giocatore e degli altri giocatori in competizione e li ragruppasse in modo da fornire le informazioni all'agente.

Questi sono stati passaggi essenziali nell'ottenere una base solida da cui partire per la realizzazione del progetto e che hanno permesso di creare un videogioco in grado di rispettare i requisiti minimi precedentemente descritti per poter integrare una logica ad agenti di supporto.

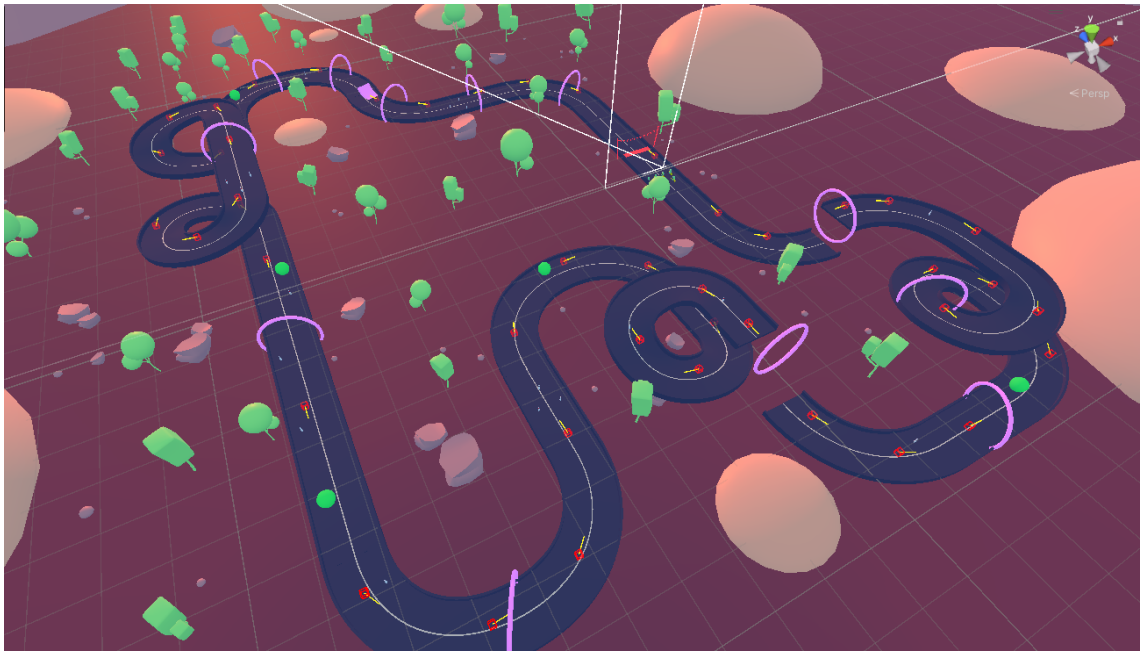

Figura 1.1: Vista dall'alto della nuova mappa di gioco realizzata esclusivamente per il lavoro di tesi.

1.6.1 Problematiche con l'introduzione del distribuito.

Uno dei problemi maggiori affrontati nel videogioco era proprio la gestione dell'aspetto distribuito. Due erano i principali motivi

1. Come testare la scalabilità del sistema?
2. Come estendere il framework Mirror per considerare una logica ad agenti?

Per risolvere il primo problema, inizialmente si è deciso di creare N versioni del gioco da testare con utenti reali, al fine di verificare le meccaniche di base e il loro funzionamento. Tuttavia, questa soluzione si è rivelata insostenibile nel lungo termine. Per questo motivo, si è implementato un numero arbitrario di kart "AI" nel motore di gioco locale, in grado di simulare il comportamento di giocatori esterni.

Questi kart non sono stati progettati come semplici entità programmate attraverso una classica State Machine, approccio comunemente utilizzato per creare intelligenze artificiali nei videogiochi. Al contrario, si è sfruttata una serie di script (componenti) forniti da Unity all'interno dell'ambiente di gioco messo a disposizione nel progetto di learning. Questo ha permesso di creare una rete neurale relativamente semplice, adatta alle limitazioni imposte dall'importazione di modelli in Unity come asset, con uno spazio disponibile limitato.

La rete neurale presenta un learning rate elevato e un numero ridotto di epoche e livelli, caratteristiche che riflettono la necessità di bilanciare le prestazioni e le limitazioni tecniche. Durante l'addestramento, sono state utilizzate diverse configurazioni di veicoli (ArcadeDriver, 4x4Driver, MuscleDriver e RoadsterDriver), ciascuna con parametri specifici per rappresentare tipologie differenti. Tuttavia, nel gioco sviluppato, l'unico veicolo effettivamente impiegato è ArcadeDriver, al quale si fa riferimento per il funzionamento principale.

L'addestramento della rete neurale è avvenuto tramite l'uso di sensori virtuali, che si attivano quando il kart entra in collisione con i box collider. Questo approccio ha consentito di simulare efficacemente il comportamento dei kart, migliorando l'accuratezza delle simulazioni e la coerenza con le dinamiche del gioco.

```
default:
    trainer: ppo
    batch_size: 1024
    beta: 5.0e−3
    buffer_size: 10240
    epsilon: 0.2
    hidden_units: 128
    lambd: 0.95
    learning_rate: 3.0e−4
    learning_rate_schedule: linear
    max_steps: 5.0e10
    memory_size: 256
    normalize: false
    num_epoch: 3
    num_layers: 2
    time_horizon: 64
    sequence_length: 64
    summary_freq: 1000
    use_recurrent: false
    vis_encode_type: simple
    reward_signals:
        extrinsic:
            strength: 1.0
            gamma: 0.99

ArcadeDriver:
    batch_size: 512
    learning_rate: 2.0e−4

4x4Driver:
    beta: 5.0e−4
    batch_size: 512

MuscleDriver:
    batch_size: 512

RoadsterDriver:
    batch_size: 512
    learning_rate: 2.0e−4
```

Codice 1.2: Lista dei parametri utilizzati per addestrare l'AI dei kart.

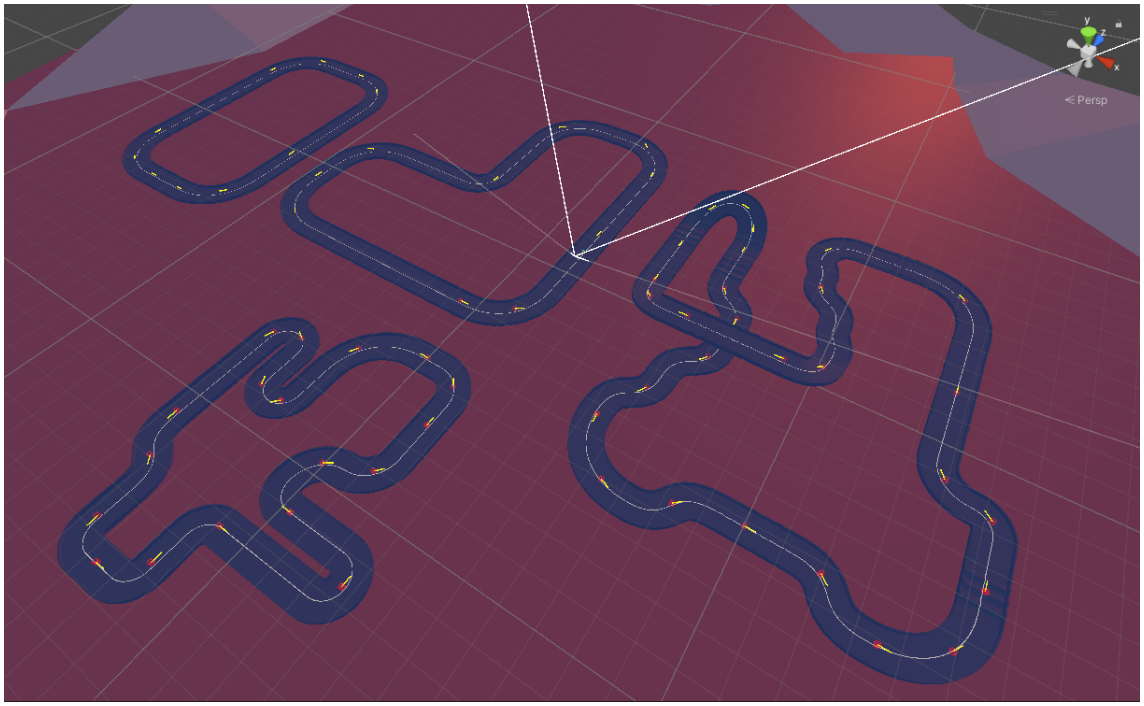

Figura 1.2: Scena di gioco contentente le mappe utilizzate per l'addestramento dei kart pilotati da AI.

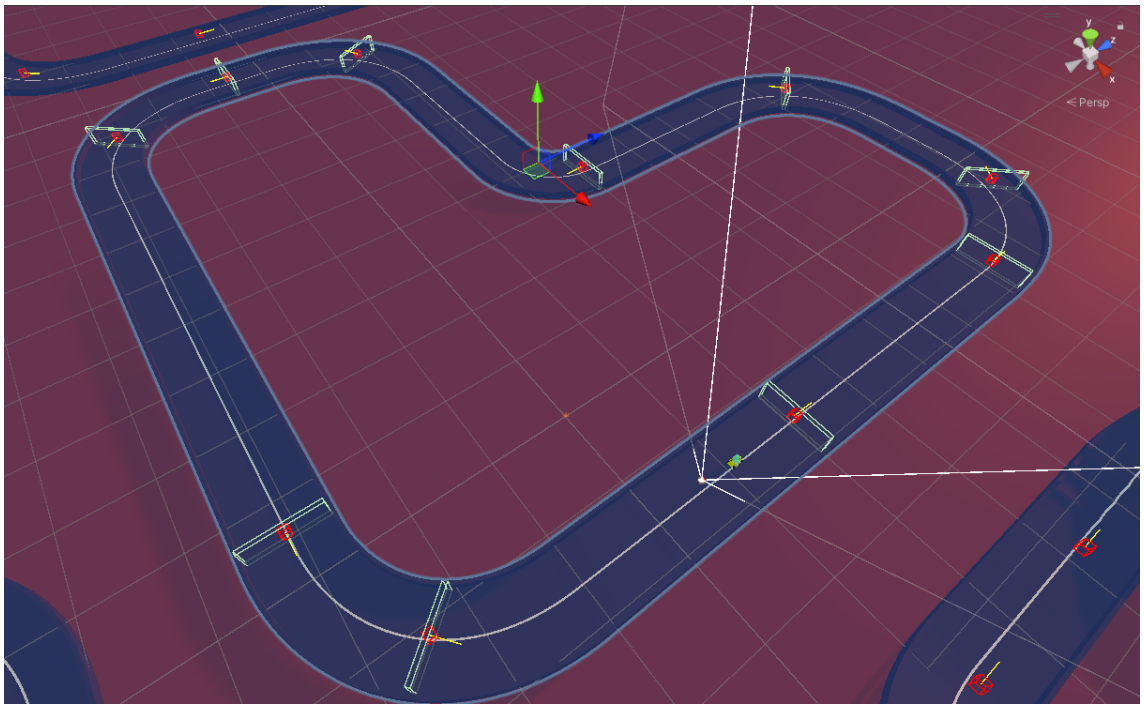

Figura 1.3: Vista dall'alto di una delle mappe di addestramento con il posizionamento dei box collider per l'attivazione sei sensori.

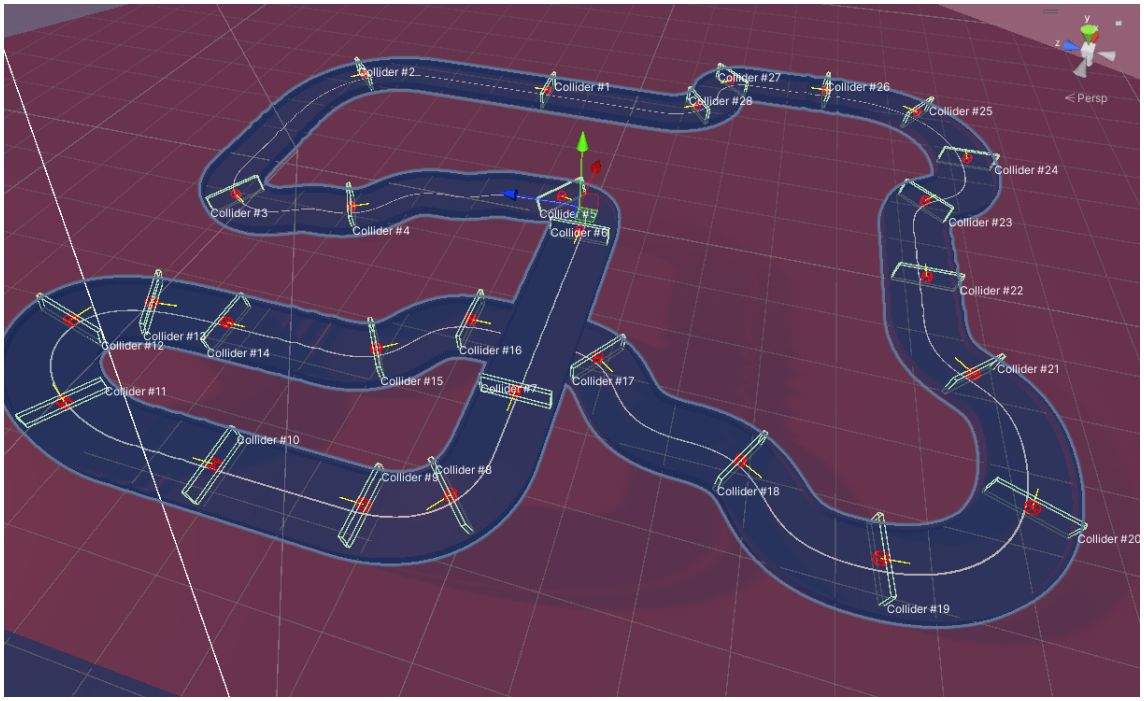

Figura 1.4: Si evidenziano la presenza dei box collider in una delle mappe di gioco utilizzate per l'addestramento.

Dunque, detto ciò si è attaccato una componente AI a diversi ArcadeKart nel gioco finale in modo da simulare quanto più possibile il comportamento realistico di altri giocatori e si è integrato i box collider all'interno della mappa per attivare i sensori dei vari kart.

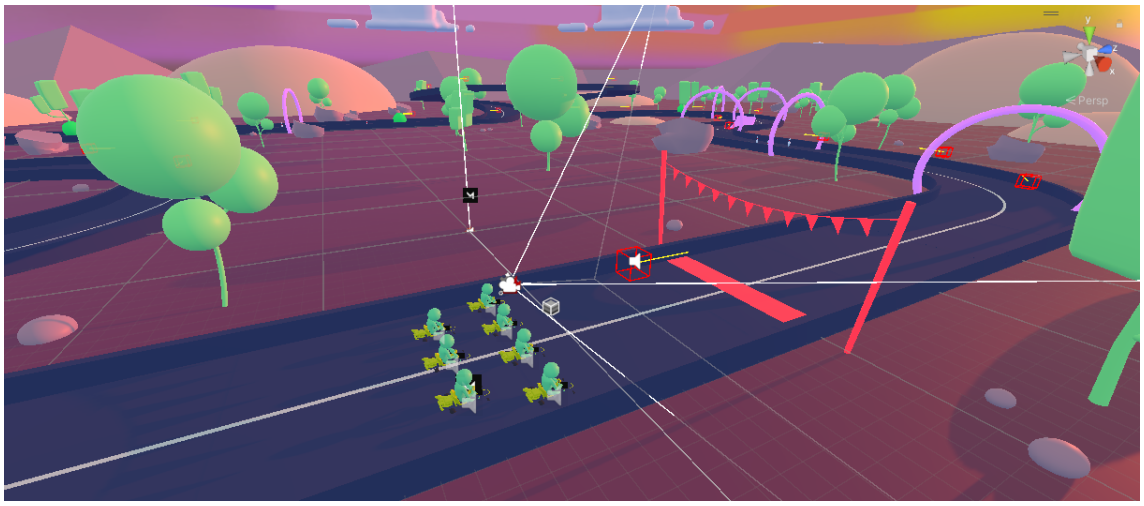

Figura 1.5: Preparazione della gara con i Kart AI messi in fila.

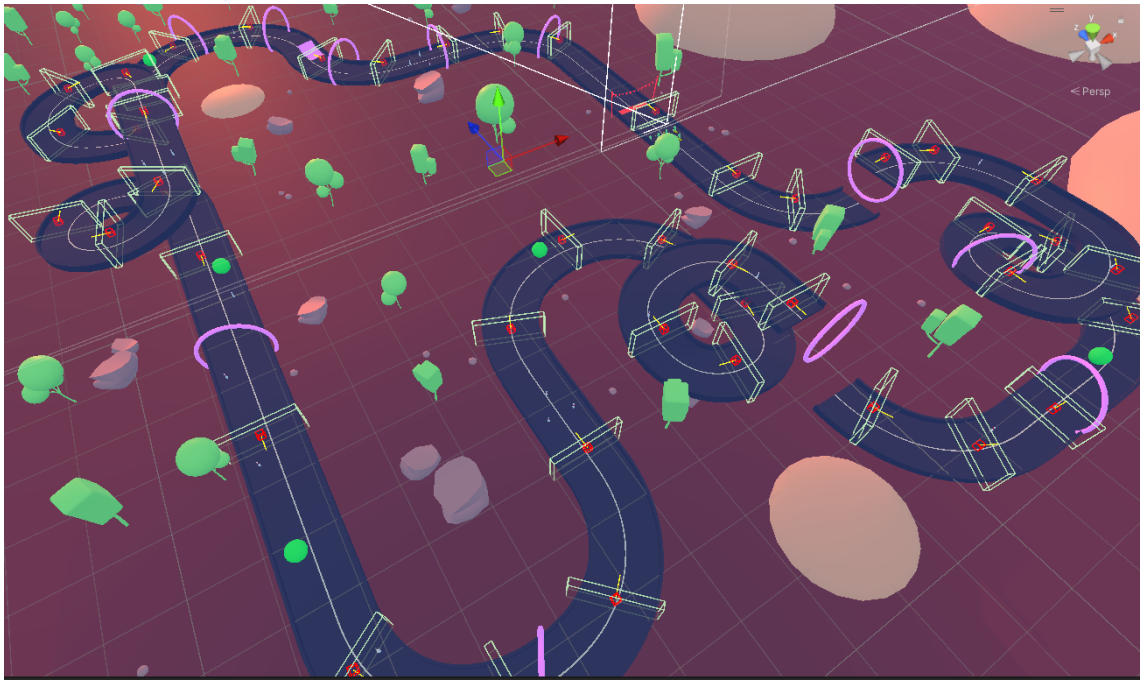

Figura 1.6: Vista della mappa finale con i box collider.

Questo ha permesso di testare in maniera totalmente illimitata la scalabilità del gioco permettendo anche di addestrare maggiormente a seconda delle necessità l'agente su quelle che sono le dinamiche di gioco e di conseguenza il bilanciamento. In quanto, una maggiore presenza di ArcadeKart porta ad un più veloce sbilanciamento di gioco, mentre una presenza minima porterebbe all'esatto opposto. Dunque, anche sotto questo aspetto si è scelto di testare l'agente con un intervallo di giocatori pari a 8, di cui 7 sono AI e 1 è il giocatore locale.

Adesso bisogna risolvere il secondo problema, ovvero come estendere il framework Mirror per considerare una logica ad agenti. Per poter parlare di questo problema prima di tutto bisogna chiedersi il come mai di questa necessità: lo scopo del gioco è quello di generare dei dati su cui addestrare il modello di machine learning con cui poi l'agente eseguirà la sua logica BDI per decidere se applicare o meno un bilanciamento al videogioco. Dunque due sono gli aspetti che il videogioco deve sicuramente ricoprire:

1. Fornire dati all'agente.
2. Fornire un riscontro dell'applicazione del DDA da parte dell'agente.

Proprio per questo si è valutato attentamente quali dati utilizzare, ma siccome il gioco in questione è stato appositamente creato come un kartgame per questo motivo, ovvero facilitare i due punti sopra elencati. Alla fine i dati scelti da comunicare tramite richiesta HTTP all'agente risultano essere:

- Numero di checkpoints: all'interno della scena di gioco vi sono degli anelli, ciascuno di essi rappresenta un checkpoint.
- Velocità corrente: la velocità con cui un kart sta proseguendo nella gara.
- Velocità massima: la velocità massima raggiunta da un kart.
- Distanza dal traguardo: essendo un circuito per ogni kart viene calcolata la distanza dal traguardo, che in questo caso coincide col punto di partenza.
- Distanza dal kart di fronte: per ciascun kart si calcola la distanza dal kart di fronte ad esso, in caso sia il primo la distanza è 0.
- Distanza dal kart dietro: stessa cosa ma con il kart dietro, in caso sia l'ultimo la distanza è 0
- Rank: è una classifica calcolata di volta in volta per ogni giocatore in base alla distanza dal prossimo checkpoint, non si è potuto basarla sul punto di partenza poiché è un circuito e la classifica superata la metà del percorso sarebbe stata invertita.
- Posizione di ogni kart: per ciascun kart si salvano le coordinate x,y,z nella scena di gioco.

Una piccola considerazione su ciò che accumuna maggiormente questi dati che essendo un kartgame, praticamente tutti i dati inviati riguardano i gli arcade kart. In giochi di dimensioni più grandi e con più meccaniche la quantità di dati poteva notevolmente aumentare. Ma per lo scopo di questo progetto sono risultati sufficienti.

Si è dunque, creato un gameobject all'interno della scena di gioco che il cui scopo è quello di raggruppare tutti questi dati e poi inviarli tramite una richiesta POST al server dell'agente. Si riporta di seguito una breve lista di codici C# che caratterizzano questa fase.

```
1   void Start()
    {
3       objective = FindObjectOfType<Objective>();
        checkpointManager = FindObjectOfType<CheckpointManager>();
5       finishLineTransform = GameObject.FindGameObjectWithTag("StartFinishLine
            ")?.transform;
        if (objective == null)
7       {
            Debug.LogError("Something's wrong with references not found on the
                GameObject.");
9       }
        else
11      {
            Debug.Log("Objective component found: " + objective.GetType().Name);
13          checkpointCount = objective.NumberOfActivePickupsRemaining();
        }
15  }
```

Codice 1.3: Codice avviato allo start del game per evidenziare i soggetti coinvolti, ed inizializzare i principali script coinvolti.

```
    void Update()
    {
        checkpointManager.UpdateDistancesForEveryKartFromCheckpoint(
            kartCheckpointPickedMap.Keys.ToList());
        sortedKarts = checkpointManager.GetRankedKarts(kartIdMap.Keys.ToList());

        if (Time.time - lastUpdateTime > 2f)
        {
            foreach (var entry in kartIdMap)
            {
                ArcadeKart kart = entry.Key;
                string playerId = entry.Value;

                int kartCheckpointCount = kartCheckpointPickedMap[kart];
                SendPlayerData(playerId, kart, kartCheckpointCount);
            }

            lastUpdateTime = Time.time;
        }
    }
```

Codice 1.4: Questo codice è eseguito assieme al game loop, e non fa altro che richiamare periodicamente con un tempo prefissato la funzione che manda i dati necessari all'agente.

```
void SendPlayerData(string playerId, ArcadeKart kart, int checkpointCount)
    {
        ArcadeKart.Stats stats = kart.baseStats;
        float currentSpeed = kart.GetMaxSpeed();
        Vector3 kartPosition = kart.transform.position;
        float distanceToFinish = Vector3.Distance(kartPosition, finishLineTransform.
            position);

        int rank = sortedKarts.IndexOf(kart) + 1;

        float? distanceToFront = null;
        float? distanceToBack = null;

        if (rank > 1)
        {
            ArcadeKart kartInFront = sortedKarts[rank \- 2];
            distanceToFront = Vector3.Distance(kartPosition, kartInFront.transform.
                position);
        }
        else
        {
```

```
            distanceToFront = 0;
        }

        if (rank < sortedKarts.Count)
        {
            ArcadeKart kartInBack = sortedKarts[rank];
            distanceToBack = Vector3.Distance(kartPosition, kartInBack.transform.
                position);
        }
        else
        {
            distanceToBack = 0; // Oppure un altro valore speciale come float.
                MaxValue
        }


        string jsonPayload = $"{{\"player_id\":\"{playerId}\",\"checkpoints\":{
            checkpointCount}," +
                                $"\"current_speed\":{currentSpeed},\"top_speed\":{
                                    stats.TopSpeed}," +
                                $"\"acceleration\":{stats.Acceleration},\"position\":
                                    {{\"x\":{kartPosition.x},\"y\":{kartPosition.y},\"z
                                    \":{kartPosition.z}}}," +
                                $"\"distance_to_front\":{distanceToFront},\"
                                    distance_to_back\":{distanceToBack}," +
                                $"\"rank\":{rank},\"distance_to_finish\":{
                                    distanceToFinish}}}";

        Debug.Log("Sending this payload for Kart:" + kart.name + ":" + jsonPayload)
            ;
        StartCoroutine(SendDataToServer(jsonPayload));
    }
```

Codice 1.5: Questo è il codice che manda tutti i dati al server dell'agente tramite una richiesta HTTP POST utilizzando Unity Web Request.

```
    IEnumerator SendDataToServer(string jsonPayload)
    {
        using (UnityWebRequest www = new UnityWebRequest(agentUrl, "POST"))
        {
            byte[] jsonToSend = new System.Text.UTF8Encoding().GetBytes(
                jsonPayload);
            www.uploadHandler = new UploadHandlerRaw(jsonToSend);
            www.downloadHandler = new DownloadHandlerBuffer();
            www.SetRequestHeader("Content-Type", "application/json");

            yield return www.SendWebRequest();

            if (www.result == UnityWebRequest.Result.Success)
            {
                Debug.Log("Player data sent successfully.");
```

```
            }
            else
            {
                Debug.LogError("Error sending player data: " + www.error);
            }
        }
    }
```

Codice 1.6: Infine, un codice a parte che manda tramite Unity Web Request il payload della richiesta in formato JSON al server dell'agente.

Nei codici precedenti si vuole evidenziare come, si sia scelto di raggruppare i dati in formato JSON, il formato è stato scelto appositamente per facilitare la comunicazione HTTP. Ma anche per schematizzare meglio i dati in quanto nei prossimi capiti questo tornerà utile per effettuare la loro estrapolazione e manipolazione.

Dunque, in questa sezione si termina la prima fase del sistema che riguardava la creazione del videogioco e tutto ciò che è stato considerato durante il suo sviluppo per poter integrare al meglio la tecnologia ad agenti che andremo a descrivere meglio nel prossimo capitolo.

Capitolo 2

BDI Agents

2.1 Cenni di Programmazione Logica

In questa sezione verranno descritte le modalità per scrivere codice AgentSpeak in Jason al fine di implementare sistemi multi-agente (MAS) composti da agenti BDI. Prima di procedere, è necessario chiarire alcuni termini fondamentali che saranno utilizzati:

- Ogni simbolo (sequenza di caratteri) che inizia con una lettera minuscola è detto **atom**. Un atom è equivalente a una costante e viene utilizzato per rappresentare individui o oggetti.
- Un simbolo che inizia con una lettera maiuscola viene interpretato come una **variabile**. Inizialmente, tutte le variabili sono non istanziate, e il processo attraverso cui vengono istanziate è detto *unificazione*. Una formula è detta **ground** (fondata) se tutte le sue variabili sono state istanziate.
- Il termine **term** si riferisce a una costante, una variabile o una struttura.
- Le informazioni sono rappresentate in forma simbolica attraverso i **predicati**, che esprimono particolari proprietà.
- Un **letterale** (literal) è un predicato (o la sua negazione) che rappresenta una relazione fra oggetti.
- Un'**annotazione** (annotation) è un termine complesso (una struttura) che fornisce dettagli strettamente associati a un particolare *belief*.

Questa terminologia costituisce la base per comprendere e utilizzare i concetti fondamentali della programmazione in AgentSpeak con Jason che verranno successivamente descritti maggiormente nel dettaglio, ma prima bisogna descrivere cosa si intende per agente e quali sono i suoi utilizzi.

2.2 Cosa è un agente intelligente?

Per parlare di un agente si cita la seguente definizione:

> « *Gli agenti possono essere definiti come entità computazionali autonome che encapsulano il controllo e possiedono un criterio per governarlo. Tale autonomia consente loro di operare in modo indipendente, dando origine a una serie di caratteristiche chiave. Gli agenti sono infatti interattivi, sociali, proattivi e situati; possono avere obiettivi o compiti da perseguire, oppure essere reattivi, intelligenti o mobili.* » (Omicini, 2013)

Gli agenti intelligenti rappresentano entità di diversa natura in grado di percepire l'ambiente circostante attraverso sensori e di eseguire azioni specifiche mediante attuatori. Nel caso dell'essere umano, i sensori possono corrispondere agli occhi e alle orecchie, mentre gli attuatori sono identificabili con mani e piedi. Nell'ambito dell'intelligenza artificiale, si fa riferimento a dispositivi specializzati; tuttavia, il fattore cruciale non risiede nell'oggetto stesso, bensì nella sua progettazione. L'obiettivo fondamentale è quello di sviluppare un agente intelligente che, per definizione, sia capace di compiere la scelta appropriata nel momento opportuno. Pertanto, analogamente a quanto avviene per le diverse tecnologie dell'intelligenza artificiale, è imperativo programmare gli agenti intelligenti stabilendo le azioni da intraprendere e le circostanze in cui queste devono essere attuate. È necessario definire il modello a priori, permettendo così un confronto tra i risultati ottenuti.

La caratterizzazione dell'agente intelligente avviene attraverso il contesto di applicazione, le percezioni utilizzate, gli obiettivi perseguiti e le azioni intraprese. L'acronimo inglese **PAGE**(Percepts, Actions, Goals, Environment) serve precisamente a definire l'agente e a delinearne le caratteristiche distintive.

È fondamentale considerare che gli agenti tendono a ottimizzare le proprie prestazioni, cercando di massimizzarle in base alle loro percezioni. Non si tratta esclusivamente dell'esecuzione di un compito, ma anche della raccolta di informazioni. In ogni caso, indipendentemente dallo scopo per cui viene sviluppato un agente intelligente, la procedura da seguire rimane invariata. In linea generale, è necessario coniugare un programma informatico con l'architettura. Quest'ultima rappresenta l'hardware incaricato di eseguire i calcoli fisici, sia esso un computer o un robot. A dirigere il funzionamento della macchina intervengono algoritmi appositamente progettati per consentire alla macchina di svolgere compiti specifici in situazioni determinate.

A titolo esemplificativo, per facilitare la comprensione della procedura di confronto, si consideri l'ipotesi di due agenti intelligenti coinvolti nella risoluzione dello stesso puzzle.

- le percezioni: consistono nei pixel delle immagini dei singoli pezzi del puzzle;
- le azioni: riguardano la raccolta delle tessere e il relativo incastro;
- l'obiettivo: è quello di completare il puzzle;

- l'ambiente: in cui opera è dato dal tavolo su cui poggiano le varie tessere del puzzle.

Un agente software si caratterizza principalmente per le seguenti proprietà:

- **Situatedness**: indica che un agente è collocato in un determinato ambiente e che è sensibile ai suoi cambiamenti.
- **Autonomia**: caratterizza il grado di libertà che possiede l'agente nel prendere decisioni riguardo alle azioni che deve compiere. È anche un sinonimo di intelligenza.
- **Proattività**: è la capacità di un agente di prendere decisioni autonome a prescindere dagli stimoli che provengono dall'ambiente che lo circonda. Un agente può quindi avere un comportamento goal-oriented.
- **Reattività**: è la capacità di un agente di reagire ai cambiamenti dell'ambiente in cui esegue.
- **Abilità sociali**: gli agenti sono in grado di interagire tra loro ed eventualmente anche con gli esseri umani. La comunicazione deve avvenire attraverso linguaggi con semantiche concordate in precedenza.
- **Mobilità**: gli agenti sono in grado di migrare da un nodo computazionale ad un altro.
- **Persistenza**: nel caso di sospensione dell'esecuzione, l'agente è in grado di salvare il proprio stato all'interno dell'ambiente in cui è situato. In questo modo è in grado di continuare la propria esecuzione quando viene riattivato.
- **Adattività**: indica la capacità di un agente di apprendere sia dalle proprie azioni che dall'ambiente che lo circonda. Un agente è quindi in grado di migliorare nel tempo la propria base di conoscenza.
- **Benevolenza**: un agente prova sempre ad eseguire il compito che gli è stato affidato controllando che i goal assegnati non siano in contrasto tra loro.
- **Fidatezza**: un agente si definisce fidato se fornisce la garanzia di non comunicare a terzi informazioni riservate.
- **Cooperazione**: un agente interagisce con altri agenti per raggiungere uno scopo comune. In questo modo un agente può ottimizzare le proprie operazioni ed offrire servizi ad altri agenti presenti nell'ambiente in cui esegue. È possibile anche che vi sia competitività tra agenti nel caso in cui essi tentino di accedere contemporaneamente alle stesse risorse.

È possibile classificare gli agenti in due categorie:

- **Deboli**: se possiedono solo le prime sette proprietà;
- **Forti**: se possiedono tutte le proprietà elencate.

In seguito, con il termine agente si intenderà sempre un agente forte.

2.2.1 Suddivisione degli agenti

> *«Gli agenti intelligenti possono manifestarsi in diverse forme e sono classificati in agenti e sotto-agenti, a seconda delle loro caratteristiche distintive, sempre considerando la loro composizione PAGE. Ad esempio, è possibile distinguere tra agenti fisici, dotati di sensori e attuatori, e agenti temporali, che si avvalgono di informazioni temporali per fornire istruzioni e raccogliere input al fine di modificare i comportamenti in seguito alla situazione analizzata.»* (Intelligenza Artificiale, 2024)

A determinare la classificazione degli agenti è il grado di intelligenza percepita e alle abilità. Seguendo questi elementi si ottengono 5 classi:

1. Agenti con riflessi semplici: con il compito solamente di reagire a una determinata azione esterna.

2. Agenti con riflessi basati su un preciso modello: L'agente memorizza il proprio stato e consente di avere dati che descrivono ambiti che non possono essere osservati in condizioni normali. Vanno abbinati i dati raccolti con altre informazioni conosciute dagli operatori per poter avere una fotografia completa e precisa dell'ambiente in cui si intende lavorare;

3. Agenti operanti per obiettivi: agiscono memorizzando l'informazione su situazioni desiderabili. In questo caso l'agente ha il compito di selezionare la migliore tra diverse possibilità per riuscire a raggiungere l'obiettivo;

4. Agenti improntati all'utilità: hanno la sola possibilità di fare una distinzione tra stati goal e non-goal. Definiscono le misure circa la desiderabilità di ogni stato, da stabilire attraverso una funzione d'utilità che mappa i valori delle stesse utilità;

5. Agenti che lavorano sull'apprendimento: una tipologia di intelligenza artificiale autonoma: le azioni vengono compiute in maniera indipendente e il software è in grado di adattarsi alle più diverse condizioni in evoluzione. Questo tipo deve disporre di caratteristiche specifiche: capacità di imparare attraverso l'interazione con l'ambiente; adattamento online e in tempo reale; imparare velocemente e acquisire una quantità elevata di informazioni; accogliere nuove regole per migliorare la risoluzione dei problemi; detenere una memoria ampia con la capacità di recupero; avere parametri per selezionare le informazioni dalla memoria tenendo conto di vari criteri; avere la capacità di esaminare i comportamenti inclusivi di sbagli e successi.

Al fine di ottenere la massima efficienza degli agenti intelligenti è necessario organizzare una struttura gerarchica composta da svariati sub-agenti concepiti per svolgere mansioni di basso livello o comunque limitate.

Ecco alcune tipologie di sotto-agenti:

- agenti temporali: che basano le proprie decisioni sul tempo;
- agenti spaziali: che applicano la fisica del mondo reale;
- agenti di input: che si occupano di elaborare le informazioni ricevute attraverso i sensori, ne sono un esempio le reti neurali;
- agenti di elaborazione: capaci di attuare la risoluzione dei problemi, ne sono esempi i sistemi dotati di riconoscimento vocale;
- agenti decisionali: che sono in grado di prendere decisioni in maniera autonoma;
- agenti dell'apprendimento: che acquisiscono dati e li ordinano creando un database.Tutti questi elementi possono essere combinati tra loro ed essere parte di un agente per apportare le funzioni utili a chi li detiene.

2.2.2 Sistemi multi agente (MAS)

Per parlare di un MAS si cita la seguente frase:

> « *Secondo la letteratura, un Sistema Multi-Agente (MAS) può essere concepito come un'aggregazione di molteplici loci di controllo distinti, che interagiscono tra loro attraverso lo scambio di informazioni* » (Omicini & Viroli, 2011).

In particolare, si fa notare come il modo in cui gli agenti interagiscono in un sistema: tramite l'ambiente condiviso, tramite messaggi strutturati (ontologie, protocolli di interazione) ha a che fare con un MAS che può essere definito in termini di entità interagenti, e in particolare di agenti. La comunicazione può variare da forme semplici a forme sofisticate. Una forma semplice di comunicazione è quella limitata a segnali semplici, con interpretazioni fisse. Tale approccio è stato utilizzato da Georgeff nella pianificazione multi-agente per evitare conflitti quando un piano veniva sintetizzato da diversi agenti. Una forma di comunicazione più elaborata avviene tramite una struttura a lavagna. Una lavagna è una risorsa condivisa, solitamente divisa in più aree, secondo diversi tipi di conoscenza o diversi livelli di astrazione nella risoluzione dei problemi, in cui gli agenti possono leggere o scrivere le corrispondenti informazioni rilevanti per le loro azioni.

L'autonomia è un'altra caratteristica importante degli agenti, quando definiscono un MAS, detti anche "sistemi auto-organizzati", consentendo loro di trovare la migliore soluzione ai loro problemi "senza intervento". La caratteristica principale che si ottiene quando si sviluppano sistemi multi-agente è la flessibilità, poiché un sistema multi-agente può essere aggiunto, modificato e ricostruito, senza la necessità di riscrivere dettagliatamente l'applicazione. Il MAS tende inoltre a prevenire la propagazione dei guasti, ad autoripristinarsi e ad essere tollerante ai guasti, principalmente a causa della ridondanza dei componenti.

```
mas         → "MAS" <ID> "{"
                 [ "infrastructure" ":" <ID> ]
                 [ environment ]
                 agents
              "}"
environment → "environment" ":" <ID> [ "at" <ID> ]
agents      → "agents" ":" ( agent ";" )+
agent       → <ASID>
                 [ filename ]
                 [ options ]
                 [ "agentArchClass" <ID> ]
                 [ "agentClass" <ID> ]
                 [ "#" <NUMBER> ]
                 [ "at" <ID> ]
filename    → [ <PATH> ] <ID>
options     → "[" option ( "," option )* "]"
option      → <ID> "=" ( <ID> | <NUMBER> | <STRING> )
```

Figura 2.1: Formato di un file MAS in grammatica EBNF.

La configurazione di un sistema multi-agente completo è data da un semplice file di testo. In questa grammatica, **NUMBER** è utilizzato per i numeri interi, **ASID** sono identificatori AgentSpeak, che devono iniziare con una lettera minuscola, **¡ID¿** è un identificatore qualsiasi (come al solito) e **PATH** è richiesto per definire i percorsi dei file..

L'**ID** utilizzato dopo la parola chiave **MAS** è il nome del sistema. La parola chiave "Infrastructure" viene utilizzata per specificare in quale delle due infrastrutture messe a disposizione da Jason si opera. Le opzioni attualmente disponibili sono entrambe "Centralizzato" o "Saci"; quest'ultima opzione consente agli agenti di essere eseguiti su diversi macchine in rete. È importante notare che l'ambiente dell'utente e le classi di personalizzazione rimangono le stesse con entrambe le infrastrutture.

Successivamente è necessario fare riferimento a un ambiente. Questo è semplicemente il nome della classe Java utilizzata per programmare l'ambiente e vedremo successivamente in cosa consiste questo ambiente. La parola chiave "agent" viene utilizzata per definire l'insieme di agenti che prenderanno parte nel **MAS**. Un agente viene specificato innanzitutto tramite il suo nome simbolico come termine AgentSpeak(L) (ovvero un identificatore che inizia con una lettera minuscola); Questo è il nome che gli agenti utilizzeranno per riferirsi ad altri agenti nel sistema.

Quindi, è possibile fornire un nome file opzionale in cui l'estensione viene fornito

il codice sorgente AgentSpeak per quell'agente; per impostazione predefinita Jason presume che il codice sorgente di AgentSpeak è nel file **nome.asl**, dove **nome** è quello dell'agente.

L'utente può modificare le impostazioni iniziali dell'interprete AgentSpeak disponibile in Jason, o trasmettere le impostazioni alle classi agente racchiudendo in quadrati tra parentesi alcune istruzioni di configurazione. Questi hanno la forma di una parola chiave, seguito da '=' e poi dal valore (eventualmente parole chiave predefinite) attribuito a loro.

2.3 Descrizione dell'architettura BDI

Un famoso modello concettuale noto come BDI, acronimo delle parole Beliefs (credenze, conoscenze), Desires (desideri), Intentions (intenzioni). Esso è stato progettato con intenti molteplici, ma, primariamente, per agevolare la progettazione di sistemi multi-agente, offrendo una rappresentazione degli agenti qualificata mediante uno «stato mentale» analogo a quello umano e consentendo di programmare gli agenti incidendo sul loro stato mentale anziché rimanendo al vecchio e severo metro delle singole istruzioni indotte.

Un aspetto di grandissima rilevanza nello schema concettuale BDI è quello per cui il progettista, in fase di sviluppo, non è tenuto a conoscere necessariamente i particolari operativi con cui gli agenti realizzeranno i fini concertati, avendo la diretta cura di definire i profili organizzativi e intenzionali di essi. Per tale ragione il modello permette di rappresentare gli agenti come entità autonome ed intelligenti, con un comportamento quasi a imitazione di quello umano.

In dettaglio, i tre tronconi principali dell'architettura BDI sono:

- **Beliefs (credenze)**: rappresentano la conoscenza che un agente possiede riguardo all'ambiente in cui opera. Analogamente alla conoscenza umana, le credenze possono essere incomplete o errate, portando a una visione imperfetta dell'ambiente circostante. Esse costituiscono la base su cui si fondano i fini e le azioni dell'agente.
- **Desires (desideri)**: rappresentano gli obiettivi o gli stati che l'agente vorrebbe raggiungere. I desideri definiscono le situazioni ideali che l'agente aspira a realizzare e influenzano le sue decisioni future, delineando possibili linee di condotta. Tuttavia, i desideri possono entrare in conflitto tra loro.
- **Intentions (propositi)**: sono quei desideri che l'agente decide di perseguire, trasformandoli in un piano d'azione concreto. Le intenzioni guidano l'agente nel preparare e intraprendere azioni per raggiungere uno specifico obiettivo.

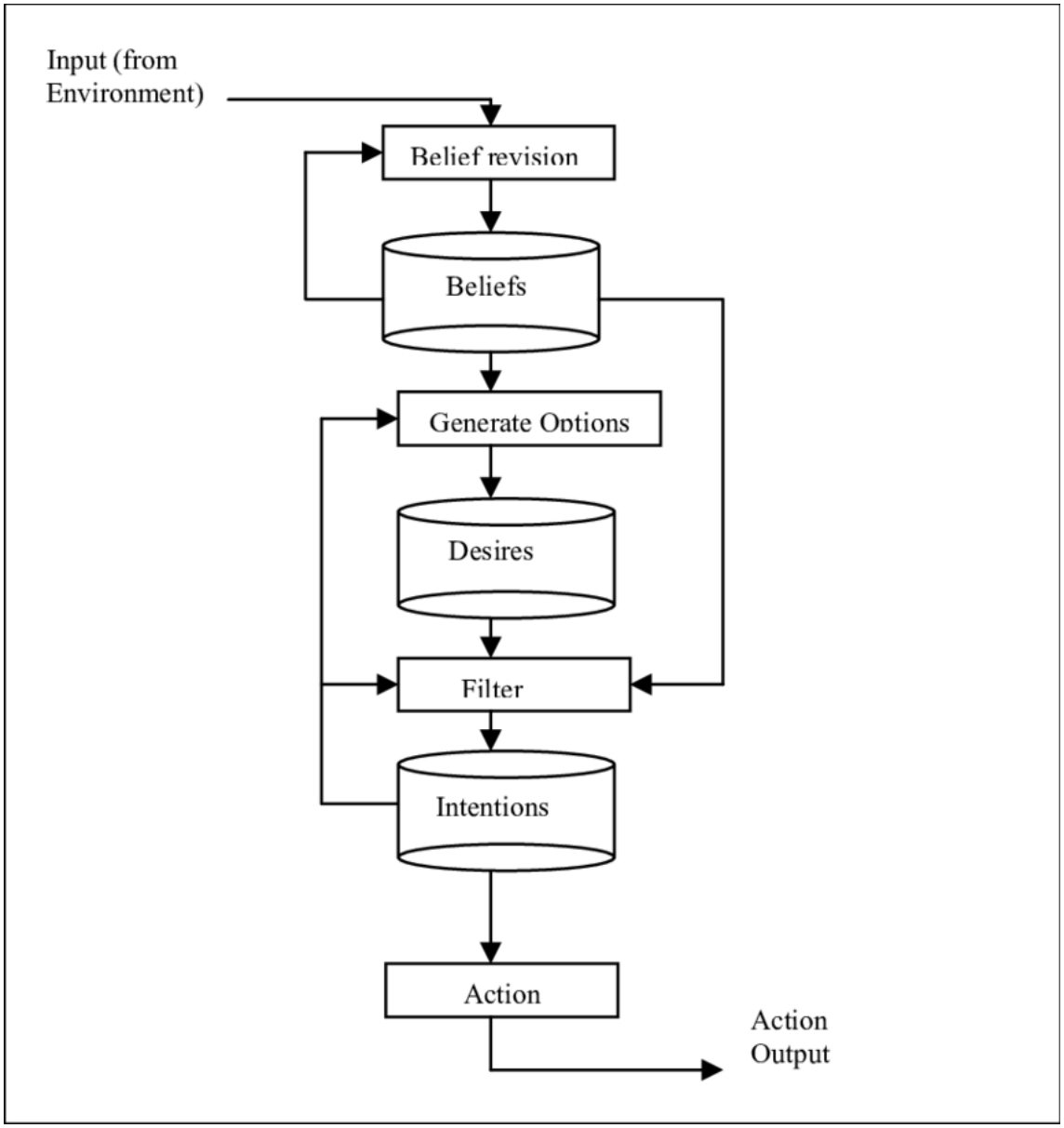

Figura 2.2: Rappresentazione dell'architettura BDI.

2.4 Linguaggi di programmazione ad Agenti

Partiamo dal dire che:

> « *L'unità di base del software sono gli agenti, che, in linea di principio, racchiudono tutto, seguendo semplicemente il modello dell'evoluzione. Qualunque cosa sia un agente, è importante comprenderne le caratteristiche desiderate. Gli agenti potrebbero, in linea di principio, essere riutilizzati in una varietà di situazioni. Gli agenti hanno il controllo*

sul proprio stato. Gli agenti sono attivi; non possono essere richiamati. Il controllo di un agente è incapsulato, e gli agenti sono entità autonome.» (Franklin & Graesser, 1996).

I primi sviluppi di applicazioni orientate agli agenti attraverso l'uso di linguaggi di programmazione risalgono ai primi anni Novanta. I linguaggi di programmazione per agenti facilitano ai programmatori la definizione e lo sviluppo degli agenti di cui hanno bisogno, in base a uno specifico modello di agente fornito dal linguaggio stesso. Uno dei modelli più utilizzati dai linguaggi di programmazione per agenti è l'architettura Belief-Desire-Intention (BDI). In generale, i linguaggi di programmazione per agenti sono scelti per simulare gli agenti come parte di un sistema multi-agente (MAS) o come un mix di asserzioni e clausole guidate da un interprete.

Sono principalmente due i modelli in cui sono stati sviluppati i linguaggi di programmazione per agenti: il modello orientato agli oggetti e la programmazione logica. Soprattutto nella programmazione di quei linguaggi che hanno scelto di seguire il paradigma orientato agli oggetti, la tecnologia Java è stata spesso l'argomento principale.

Di seguito seguirà una esposizione a quello che è uno dei fra i maggiori linguaggi di programmazione ad agenti utilizzato anche per la realizzazione di questo progetto.

2.4.1 JASON

«Jason è un framework per programmare agenti basato sul linguaggio AgentSpeak(L). Sviluppato per supportare l'architettura BDI (Belief-Desire-Intention), Jason permette agli sviluppatori di definire il comportamento degli agenti attraverso un linguaggio dichiarativo orientato ai piani. Jason offre funzionalità avanzate per simulare e gestire ambienti multi-agente ed è progettato per essere estensibile, consentendo l'integrazione di nuovi moduli e librerie.» (University of Bologna, 2024)

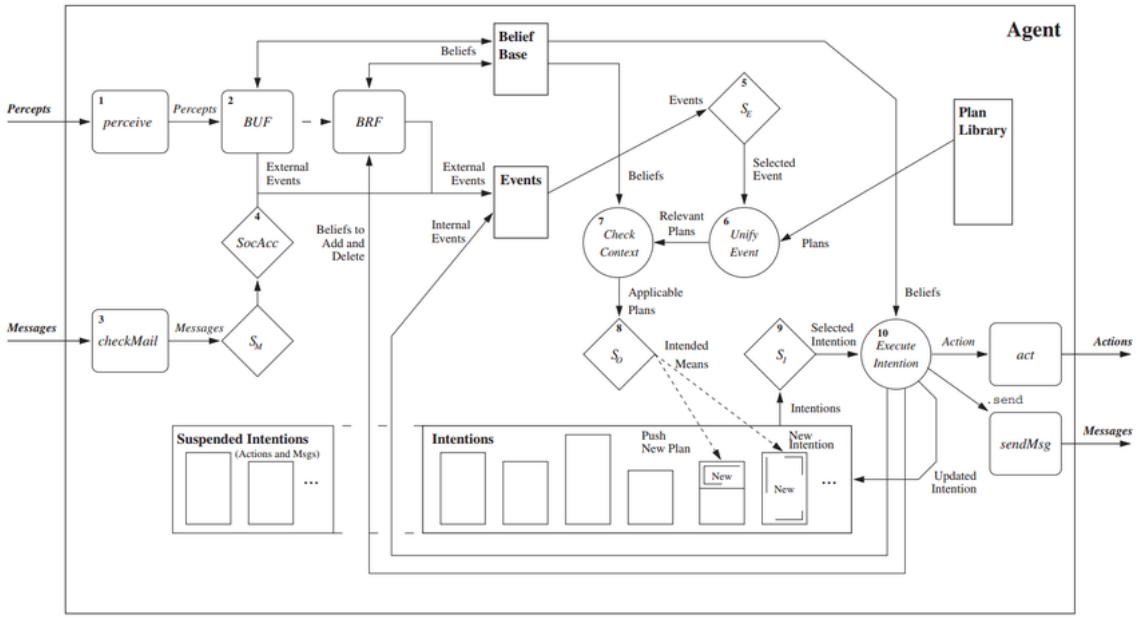

Figura 2.3: Ciclo di ragionamento utilizzato da un agente JASON.

Jason, creato da R.H.Bordini, è dunque un interprete per il linguaggio di programmazione (o meglio, una sua versione modificata) AgentSpeak(L). E' scritto in Java rendendolo multi-piattaforma ed è offerto sotto licenza GNU LGPL. Jason è utilizzato per creare agenti che possono reagire ai cambiamenti nell'ambiente, perseguire obiettivi, comunicare e collaborare con altri agenti. L'approccio basato sui piani consente di definire comportamenti condizionali in risposta agli eventi, facendo di Jason una scelta flessibile e potente per sviluppare sistemi multi-agente complessi.

Di seguito un esempio di codice Jason che implementa un semplice agente che reagisce all'evento di trovare un oggetto e ne esegue l'azione associata:

```
1  +found(object) : not holding(object) <- pick(object).
   +found(object) : holding(object) <- say("Already holding object.").
```

Codice 2.1: Esempio di codice base per mostrare la sintassi di JASON.

In questo esempio, l'agente reagisce all'evento di "trovare un oggetto" (`+found(object)`) con due condizioni: - Se l'agente non ha già l'oggetto (`not holding(object)`), lo raccoglie con l'azione `pick(object)`. - Se l'agente ha già l'oggetto, comunica un messaggio di stato.

Jason consente inoltre di eseguire la simulazione del comportamento degli agenti e di monitorare le interazioni tra agenti e ambiente.

2.4.2 Concetti fondamentali per programmare un agente JASON

La prima operazione da compiere nella programmazione di un agente in Jason consiste nello specificare i *belief* di partenza. In AgentSpeak, il *belief base*, ovvero l'insieme dei *belief* iniziali di ciascun agente, è rappresentato attraverso un insieme di predicati e letterali che costituiscono le credenze dell'agente. È importante notare che i *belief* non sono concetti veri in senso assoluto, bensì fatti che l'agente considera attualmente veritieri.

Goal: Achievement e Test

Un elemento fondamentale nella programmazione ad agenti è il concetto di *goal* (obiettivo). In AgentSpeak, esistono due tipi principali di *goal*:

- **Achievement goal** (`!g(t)`): esprimono l'intenzione dell'agente di raggiungere uno stato dell'ambiente in cui il *goal* specificato è considerato un *belief* veritiero.
- **Test goal** (`?g(t)`): rappresentano l'intenzione dell'agente di verificare se un determinato *goal* è un *belief* vero o se può essere derivato da quelli già presenti nello stato corrente dell'agente.

Quando un agente rappresenta un *achievement goal* `!g` nel suo programma, si impegna ad agire per modificare l'ambiente fino a quando, attraverso la percezione, non considererà `g` come un *belief* veritiero. Ad esempio, aggiungendo il *goal* iniziale `!own(house)`, l'agente agirà per fare in modo che `own(house)` diventi parte del suo *belief set*.

Un *test goal*, invece, viene generalmente utilizzato per recuperare informazioni dal *belief base* dell'agente. Queste informazioni possono essere già presenti o possono richiedere l'esecuzione di determinate azioni, come il *sensing* dell'ambiente. Ad esempio, il *goal* `?saldoCC(xEuro)` potrebbe corrispondere a una ricerca nel *belief set* di un letterale del tipo `saldoCC(300.00)`. Qualora tale letterale non fosse presente, potrebbe attivare l'esecuzione di un *plan* per ottenere direttamente il saldo dal sistema bancario.

Plan in AgentSpeak

Un *plan* rappresenta una sequenza di azioni che un agente esegue in risposta a un evento. In AgentSpeak, un *plan* è composto da tre parti principali:

- **Triggering event**: specifica l'evento che attiva il *plan*.
- **Context**: indica le condizioni che devono essere soddisfatte affinché il *plan* venga eseguito. Insieme al *triggering event*, costituisce la *head* del *plan*.

- **Body**: definisce le azioni che l'agente deve eseguire.

Queste parti sono separate dai simboli `:` e `<-`, secondo la seguente sintassi:

```
triggeringEvent : context <- body.
```

Belief e Goal Iniziali

Analogamente ai *belief*, è possibile specificare dei *goal* iniziali nel codice dell'agente. Questi rappresentano gli obiettivi che l'agente tenterà di raggiungere sin dall'inizio della sua esecuzione. Le variazioni nei *belief* e nei *goal* costituiscono eventi che attivano l'esecuzione dei *plan*.

2.4.3 AgentSpeak(L)

AgentSpeak(L) è un linguaggio di programmazione per agenti autonomi basato sull'architettura BDI. E' in un linguaggio logico ed equivale alla programmazione logica in quanto vi siano regole di reazione a eventi e piani di azione nell'introduzione. Bisogna necessariamente notare come le tre componenti dell'achitettura BDI vengono realizzate in AgentSpeak tramite i Belief, i Goal e i Plan: c'è quindi un leggero scostamento da quelli che sono i pilastri originali della teoria (Belief, Desire e Intention), infatti il significato è lievemente diverso e molto più pratico.

2.5 Integrazione di agenti JASON e AgentSpeak(L) in Java

L'integrazione di Jason e AgentSpeak(L) in un ambiente Java può essere realizzata utilizzando l'IDE IntelliJ IDEA e il sistema di build Gradle. Questo approccio consente di sfruttare la potenza di un ambiente di sviluppo moderno per la progettazione, l'esecuzione e il debugging di sistemi multi-agente basati sull'architettura BDI. Di seguito, vengono descritti i principali passaggi per configurare e utilizzare un progetto Jason all'interno di IntelliJ IDEA, utilizzando un file `MAS` per definire il sistema multi-agente e un file Gradle per automatizzarne l'esecuzione.

2.5.1 Configurazione del Progetto

1. **Creazione del Progetto Gradle in IntelliJ IDEA**: Per iniziare, è necessario creare un nuovo progetto Gradle in IntelliJ IDEA. Durante la configurazione, assicurarsi di selezionare Java come linguaggio principale. Gradle sarà utilizzato per gestire le dipendenze e per automatizzare l'esecuzione del progetto.

2. **Aggiunta della Dipendenza Jason**: Modificare il file `build.gradle` per includere Jason come dipendenza. Ad esempio:

```
plugins {
    id 'java'
    id 'application'
}

repositories {
    mavenCentral()
}

dependencies {
    implementation 'org.jason--lang:jason:3.1.1'
}

application {
    mainClass = 'jason.runtime.RunMas' // Classe principale per eseguire il
        MAS
}
```

Codice 2.2: Esempio di configurazione Gradle per Jason

Questa configurazione include Jason come dipendenza e specifica `RunMas` come classe principale per l'esecuzione del sistema multi-agente.

3. **Creazione del File MAS**: Il file `MAS` (Multi-Agent System) definisce la configurazione del sistema multi-agente, inclusi gli agenti, l'ambiente e i parametri di esecuzione. Un esempio base di file `MAS` potrebbe essere il seguente:

```
MAS mas_example {
    agents {
        agent1: my_agent.asl [beliefs = "init_beliefs.bb"]
    }
    environment: my_env.env
}
```

Codice 2.3: Esempio di file MAS.

4. **Organizzazione del Codice**: Il codice sorgente del progetto deve essere strutturato seguendo una convenzione logica. Ad esempio:

   - `src/main/java`: per il codice Java personalizzato, come l'ambiente (`Environment`) o le classi di supporto.
   - `src/main/agents`: per i file `.asl` contenenti i piani degli agenti.
   - `src/main/mas`: per i file `MAS`.

2.5.2 Esecuzione del Sistema Multi-Agente

L'esecuzione del sistema multi-agente può essere automatizzata utilizzando un task Gradle. È sufficiente aggiungere un comando personalizzato nel file `build.gradle` per eseguire il file MAS desiderato:

```
tasks.register('runMasExample', JavaExec) {
    group = 'application'
    description = 'Esegue il sistema multi-agente mas_example.mas'
    mainClass = 'jason.runtime.RunMas'
    classpath = sourceSets.main.runtimeClasspath
    args = ['src/main/mas/mas_example.mas']
}
```

Codice 2.4: Esecuzione automatizzata tramite Gradle.

L'esecuzione può quindi essere avviata dal terminale con il comando:

```
./gradlew runMasExample
```

Codice 2.5: Comando inserito nel prompt per avviare l'agente.

Vantaggi dell'Integrazione

L'uso di IntelliJ IDEA e Gradle per lo sviluppo di sistemi multi-agente Jason offre numerosi vantaggi, tra cui:

- Un ambiente di sviluppo integrato per il codice Java e i file agent-oriented.
- Automazione del processo di compilazione ed esecuzione grazie a Gradle.
- Debugging avanzato per analizzare i comportamenti degli agenti.

Questo approccio rende la programmazione multi-agente più gestibile e consente di sviluppare sistemi complessi in modo modulare ed efficiente.

2.6 Collegamento tra agente e video gioco sviluppato

In questa sezione si vuole evidenziare quali sono state le considerazioni messe in atto per creare un sistema efficiente che comunicasse tempestivamente con il videogiochi Unity. Si vuole volutamente saltare la descrizione di tutte quelle parti relative al compimento della valutazione da parte dell'agente, poiché verranno descritte in seguito nel successivo capitolo, ma in questo si vuole evidenziare come è stata pensata la logica dietro l'agente. Ma prima bisogna esporre una serie di tecnologie coinvolte.

2.6.1 API REST con Flask Server

> «*Flask è un web framework open source scritto con Python caratterizzato da flessibilità, leggerezza e semplicità d'uso. Fa parte della categoria dei micro-framework per via del suo approccio allo sviluppo minimalista non opinionato, che lascia agli sviluppatori la possibilità di scegliere quando e soprattutto come implementare ogni aspetto delle loro applicazioni.*» (Pallets Projects, 2024)

Nel caso di Flask micro-framework significa quindi possedere un core semplice e performante che mette a disposizione funzionalità fondamentali quali server di sviluppo e debugger, routing, supporto unit testing integrato, protezione contro cross-site scripting (XSS) e l'impiego di Jinja 2 come template engine. A differenza dei web framework full stack come Django, Flask non include ad esempio una database API, un sistema di autenticazione, upload o di validazione dei form: questo genere di funzionalità vengono incluse in una web app sito scritto con Flask tramite l'utilizzo di estensioni dedicate, che una volta integrate nel progetto potranno essere usate e facilmente come se facessero parte del framework stesso.

Nel sito ufficiale di Flask è presente un vero e proprio registro delle estensioni moderato dagli sviluppatori del framework e aggiornato regolarmente in modo da garantire un alto livello di qualità per tutte le estensioni più importanti. Alcune estensioni famose sono ad esempio Flask-Login per l'aggiunta di un sistema di autenticazione, Flask-SQLAlchemy e Flask-Migrate per lavorare comodamente col database, Flask-WTF per la gestione dei form ed altre ancora. Flask può quindi essere utilizzato per qualsiasi tipologia di progetto: si trova in cima alla lista dei web framework più utilizzati assieme a Django ed entrambi possono vantare ben oltre 60 mila stelle su GitHub a prova del loro grande valore nella Community di Python.

Tra i suoi utilizzatori celebri in vanta Linkedin e Pinterest, ed è impiegato per la creazione di siti, web app complete e microservizi in progetti e aziende di ogni livello. Grazie al suo approccio allo sviluppo flessibile, che incoraggia la personalizzazione del codice partendo da una base minimalista, facilita la creazione di componenti e servizi altamente personalizzati.

Detto ciò, l'agente è stato pensanto come un servizio esterno (ma comunque in localhost sulla porta 5000) collegato direttamente con il videogioco Unity. La comunicazione tra i due avviene tramite un API RESTful con uno scambio di payload in formato JSON.

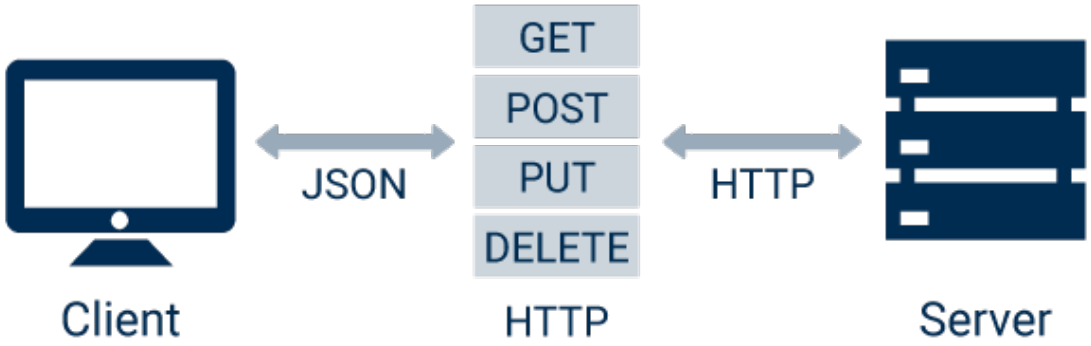

Figura 2.4: Rappresentazione dell'architettura RESTful nel sistema.

I dati raccolti dal server Flask se non presentano errori, vengono salvati all'interno di un dizionario locale che poi vengono inviati tramite una route GET effettuata dall'agente verso il server al momento della sua esecuzione per iniziare ad operare. Nel caso in cui questi dati non vi siano presenti l'agente rimane in attesa.

```
@app.route('/api/agent/data', methods=['POST'])
def update_player_data():
    data = request.json
    player_id = data.get('player_id')

    if player_id:
        # Aggiorna o aggiungi i dati del giocatore
        player_data[player_id] = {
            "checkpoints": data.get('checkpoints'),
            "current_speed": data.get('current_speed'),
            "top_speed": data.get('top_speed'),
            "acceleration": data.get('acceleration'),
            "position": data.get('position'),
            "distance_to_front": data.get('distance_to_front'),
            "distance_to_back": data.get('distance_to_back'),
            "rank": data.get('rank'),
            "distance_to_finish": data.get('distance_to_finish')
        }

        return jsonify({
            "message": f"Data updated for player {player_id}",
            "player_data": player_data[player_id]
        }), 200
    else:
        return jsonify({"error": "Player ID not provided"}), 400
```

Codice 2.6: Route POST inserita nel server Flask per catturare i dati inviati dal videogioco in Unity.

```
@app.route('/api/agent/data', methods=['GET'])
```

```
def get_all_player_data():
    return jsonify({
        "all_player_data": player_data
    }), 200
```

Codice 2.7: Route GET utilizzata dall'agente per raccogliere i dati inviati da Unity al server Flask.

2.6.2 Ambiente JASON in Java

Dunque, per poter continuare bisogna prima rispondere ad una semplice domanda: come implementare un agente in Java? Per fortuna l'utilizzo delle tecnologie scelte è risultato furbo e comodo specialmente in questo caso, poiché JASON stesso è stato sviluppato dai suoi ideatori per creare un ambiente estendibile in Java tramite l'implementazione nell'editor di apposite dipendenze.

```
import jason.asSyntax.ListTerm;
import jason.asSyntax.Literal;
import jason.asSyntax.Structure;
import jason.asSyntax.Term;
import jason.environment.Environment;
```

Codice 2.8: Lista delle dipendenze necessarie per poter creare un ambiente per agenti in Java, l'editor usato è IntelliJ IDEA.

Una volta importate nel progetto queste dipendenze è possibile creare un ambiente per sviluppare agenti in Java semplicemente creando una classe che "estende" dalla superclasse **Environment**. Questa classe appare nel seguente modo nella sua forma minimalista:

```
public class AgentEnvironment extends Environment {

    @Override
    public void init(final String[] args) {...}

    @Override
    public void stop(){...}

    @Override
    public boolean executeAction(String agName, Structure action) {...}

    @Override
    public void addPercept(java.lang.String agName, Literal... per){...}

    @Override
    public boolean removePercept(java.lang.String agName, Literal per){...}

    ...
}
```

Codice 2.9: Esempio di una classe minimale e basilare per creare un ambiente per agenti in Java.

Dovendo descrivere brevemente i metodi del codice riportato, è necessario evidenziare che:

- Init: Chiamato prima dell'esecuzione del MAS con gli argomenti informati nel file .mas2j, l'ambiente utente potrebbe sovrascriverlo.
- Stop: Chiamato subito prima della fine dell'esecuzione del MAS, l'ambiente dell'utente potrebbe sovrascriverlo.
- executeAction: Esegue un'azione sull'ambiente. Questo metodo è probabilmente sovrascritto nella classe dell'ambiente utente.
- addPercept: Aggiunge una percezione per tutti gli agenti.
- removePercept: Rimuove una percezione per un agente.

Ovviamente la dipendenza riporta più metodi di quelli elencati, ma quelli evidenziati sono a mio parere i metodi minimi necessari per poter avere un agente che operi in maniera intelligente, nonché gli stessi metodi riportati sono anche quelli utilizzati nella riuscita di questo progetto di tesi.

Ciò su cui bisogna particolarmente fare leva parlando dell'agente nel sistema è la sua capacità di compiere le famose "actions" e le "perceptions. Nelle sotto-sezioni successive parleremo meglio di queste.

2.6.3 Azioni dell'agente

In JASON, le azioni rappresentano comandi eseguibili che consentono agli agenti di interagire con l'ambiente o modificare il proprio stato interno. Queste azioni vengono definite nella classe `Environment` ed ogni azione è rappresentata da un **funtore** (il nome dell'azione) e da eventuali **argomenti**. La logica di esecuzione per ciascuna azione viene implementata nel metodo `executeAction`.

Di seguito è riportato il sistema di azioni utilizzato per il videogioco kartgame sviluppato. Questo ambiente supporta azioni come `update_checkpoint_belief`, `send_data_to_ml_model` e `train_the_model`:

```
@Override
public boolean executeAction(String agName, Structure action) {
    try {
        if (action.getFunctor().equals("update_checkpoint_belief")) { //Azione
            updatePlayerDataBelief();
            return true;
        }
        if (action.getFunctor().equals("send_data_to_ml_model")) { //Azione
```

```
            ListTerm beliefsList = (ListTerm) action.getTerm(0);
            List<String[]> structuredData = new ArrayList<>();

            for (Term term : beliefsList) {
                if (term instanceof ListTerm) {
                    ListTerm tuple = (ListTerm) term;
                    String playerId = tuple.get(0).toString();
                    String parameter = tuple.get(1).toString();
                    String value = tuple.get(2).toString();
                    structuredData.add(new String[]{playerId, parameter, value});
                }
            }

            sendDataToMLModel(structuredData);
            return true;
        }

        if (action.getFunctor().equals("train_the_model")) { //Azione
            trainTheModel();
            return true;
        }
    } catch (Exception e) {
        logger.severe("Errore nell'esecuzione dell'azione: " + e.getMessage());
    }
    return false; // Ritorna false se l'azione fallisce.
}
```

Codice 2.10: Esempio di ambiente JASON con azioni.

Gli agenti utilizzano le azioni per comunicare con l'ambiente. Ad esempio, il seguente piano AgentSpeak consente all'agente di aggiornare continuamente i propri belief, inviare i dati dei giocatori a un modello di machine learning e richiedere previsioni, anche dette percezioni:

```
+!update_checkpoint_belief_cycle
  <- .abolish(player_data(_, _, _)); // Cancella i vecchi belief.
     .wait(1000);
     update_checkpoint_belief; // Aggiorna i belief.
     .wait(1000);
     !send_player_data_to_ml_model; // Invia i dati al modello ML.
     .wait(2000);
     train_the_model; // Allena il modello ML.
     .wait(1000);
     take_prediction_from_model; // Richiedi previsioni dal modello ML.
     .wait(1000);
     !update_checkpoint_belief_cycle. // Riavvia il ciclo.

+!send_player_data_to_ml_model
  <- .findall([PlayerId, Param, Val], player_data(PlayerId, Param, Val), B);
     send_data_to_ml_model(B).
```

Codice 2.11: Piano AgentSpeak per aggiornamenti continui dei dati.

- `update_checkpoint_belief`: Recupera e aggiorna i dati dei giocatori come belief utilizzando informazioni provenienti da un server Flask esterno.
- `send_data_to_ml_model`: Invia i dati strutturati dei giocatori a un modello di machine learning per ulteriori elaborazioni.
- `train_the_model`: Avvia il processo di addestramento del modello ML inviando una richiesta POST a un endpoint definito.

Queste azioni descrivono la logica comportamentale dell'agente BDI sviluppato, il suo compito dunque riassumendo è quello di aggiungere ai propri beliefs quelle che sono le statistiche inviate dal kart game sviluppato in Unity. Poi, esegue l'azione di inviare i dati al modello ed infine l'ultima azione compiuta è quella di addestrare il modello di machine learning per effettuare una predizione da restituire per poi applicare il bilanciamento del gioco.

2.6.4 Percezioni: Raccolta e Aggiornamento dei Belief

In JASON, abbiamo detto che le **azioni** permettono agli agenti di interagire attivamente con l'ambiente, mentre ora analizzeremo le **percezioni** che, invece, rappresentano le informazioni che gli agenti ricevono dall'ambiente. Questi due meccanismi lavorano in sinergia per consentire un comportamento reattivo e proattivo degli agenti.

Le percezioni vengono utilizzate per aggiornare i belief degli agenti, riflettendo lo stato corrente dell'ambiente. Nel seguente esempio (che per motivi di spazio è stato ridotto rispetto l'originale), le percezioni dei dati dei giocatori (come posizione, velocità, checkpoint, ecc.) vengono estratte dal server Flask tramite una richiesta GET e trasformate in belief utilizzabili dagli agenti.

```
public void updatePlayerDataBelief() {
    try {
        JSONObject playerData = getPlayersData(); // Ottieni i dati dal server Flask.
        if (playerData != null) {
            JSONObject allPlayerData = playerData.getJSONObject("all_player_data")
                ;

            Iterator<String> keys = allPlayerData.keys();
            while (keys.hasNext()) {
                String playerId = keys.next();
                JSONObject playerInfo = allPlayerData.getJSONObject(playerId);
```

```
                // Estrazione dei parametri del giocatore
                int checkpoints = playerInfo.getInt("checkpoints");
                float currentSpeed = (float) playerInfo.getDouble("current_speed");
                float topSpeed = (float) playerInfo.getDouble("top_speed");
                JSONObject position = playerInfo.getJSONObject("position");
                float posX = (float) position.getDouble("x");
                float posY = (float) position.getDouble("y");
                float posZ = (float) position.getDouble("z");
                ...

                // Aggiunta delle percezioni come belief
                addPercept(Literal.parseLiteral("player_data('" + playerId + "',"
                    checkpoints'," + checkpoints + ")"));
                addPercept(Literal.parseLiteral("player_data('" + playerId + "',"
                    current_speed'," + currentSpeed + ")"));
                addPercept(Literal.parseLiteral("player_data('" + playerId + "',"
                    position',[" + posX + "," + posY + "," + posZ + "])"));
                ...
            }
        }
    } catch (Exception e) {
        logger.severe("Errore durante l'aggiornamento dei belief:" + e.getMessage());
    }
}
```

Codice 2.12: Esempio di aggiornamento delle percezioni per l'agente.

Le percezioni raccolte dall'ambiente vengono utilizzate dagli agenti per eseguire azioni che modificano ulteriormente l'ambiente o lo stato interno degli agenti. Ad esempio, dopo l'aggiornamento dei belief tramite `updatePlayerDataBelief`, un agente può utilizzare le informazioni per inviare dati a un modello di machine learning o per prendere decisioni strategiche.

Esempio di Piano AgentSpeak

Il seguente piano dimostra come un agente utilizzi le percezioni per aggiornare i belief, inviare dati a un modello ML, e richiedere previsioni:

```
+!send_player_data_to_ml_model
  <- .findall([PlayerId, Param, Val], player_data(PlayerId, Param, Val), B);
     .print(B);
     send_data_to_ml_model(B). // Action.
```

Codice 2.13: Esempio di piano con percezioni e azioni.

In JASON, il costrutto `.findall` consente di raccogliere tutti i belief che soddisfano una determinata condizione e di restituirli come una lista. Questo è particolarmente utile quando si devono elaborare più belief contemporaneamente o raccogliere dati da utilizzare in un'azione.

- La sintassi generale è:

$$.findall(Variabile, Condizione, Lista)$$

  Dove:

  - *Variabile* è il termine da raccogliere.
  - *Condizione* è il predicato che i belief devono soddisfare.
  - *Lista* è la variabile che conterrà tutti i risultati.

- Ad esempio:

```
.findall([PlayerId, Param, Val], player_data(PlayerId, Param, Val), B);
```

  In questo caso, tutti i belief che corrispondono al predicato `player_data(PlayerId, Param, Val)` verranno raccolti nella lista `B`.

`.findall` è un costrutto potente per iterare e aggregare dati, facilitando l'elaborazione di informazioni complesse all'interno di un piano.

E con questo si è riassunto il comportamento dell'agente e come già accennato nel prossimo capitolo si analizzerà nel dettaglio quella che è il ragionamento dietro le decisioni dell'agente e come questo ha potuto evolvere il proprio "pensiero" tramite l'integrazione del machine learning.

Capitolo 3

Machine Learning

3.1 Cosa è il machine learning?

> *«Il Machine Learning (ML) è un sottoinsieme dell'intelligenza artificiale (AI) che si occupa di creare sistemi che apprendono o migliorano le performance in base ai dati che utilizzano. Intelligenza artificiale è un termine generico e si riferisce a sistemi o macchine che imitano l'intelligenza umana. I termini machine learning e AI vengono spesso utilizzati insieme e in modo interscambiabile, ma non hanno lo stesso significato. Un'importante distinzione è che sebbene tutto ciò che riguarda il machine learning rientra nell'intelligenza artificiale, l'intelligenza artificiale non include solo il machine learning.»* (Corporation, 2024)

Quindi, l'apprendimento automatico, o machine learning, è il campo dell'informatica che fornisce ai computer la capacità di imparare ad eseguire un task senza essere stati esplicitamente programmati per la sua esecuzione. Evolutosi dagli studi sul riconoscimento di pattern e sull'apprendimento computazionale teorico nel campo dell'intelligenza artificiale, il machine learning esplora lo studio e la costruzione di algoritmi che permettono l'apprendimento di informazioni a partire da dati disponibili e forniscono la capacità di predire nuove informazioni alla luce di quelle apprese.

Attraverso la costruzione di un modello che impara automaticamente a predire nuovi dati a partire da osservazioni, questi algoritmi superano il classico paradigma delle istruzioni strettamente statiche. Il machine learning trova il suo impiego principale in quell'insieme di problemi di computazione in cui la progettazione e l'implementazione di algoritmi ad-hoc non è praticabile o è poco conveniente. Esempi di applicazioni oggigiorno spaziano in vari ambiti e all'interno di questa tesi vi si è utilizzato per l'apprendimento del comportamento del videogioco kart game sviluppato.

Il machine learning presenta profondi legami col campo dell'ottimizzazione matematica, il quale fornisce metodi, teorie e domini di applicazione. Molti problemi di apprendimento automatico, infatti, sono formulati come problemi di minimizzazione

di una certa funzione di perdita, detta `loss function` nei confronti di un determinato set di esempi che prende il nome di `training set`. Questa funzione esprime la discrepanza tra i valori predetti dal modello in fase di allenamento e i valori attesi per ciascuna istanza di esempio.

L'obiettivo finale è dunque quello di insegnare al modello la capacità di predire correttamente i valori attesi su un set di istanze non presenti nel training set, detto `test set` mediante la minimizzazione della loss function in questo insieme di istanze. Questo porta ad una maggiore generalizzazione delle capacità di predizione.

Una più formale ed ampiamente citata definizione per il machine learning è stata formulata:

> « *Si dice che un programma per computer impara dall'esperienza E nei confronti di una certa classe di compiti C e insieme di misure di performance P se le sue performance sui compiti di C, secondo le misure definite in P, migliorano grazie all'esperienza E.*» (da Turing, 1950)

Questa definizione è rilevante poiché formula il machine learning in termini fondamentalmente operazionali piuttosto che cognitivi, seguendo la proposta formulata:

> « *La frase "Le macchine possono pensare?" risulta poco appropriata e dovrebbe essere sostituita da "Possono le macchine fare quello che noi (come esseri pensanti) possiamo fare?".*» (da Turing, 1950)

3.2 Modelli diversi, per problemi diversi

I diversi compiti del machine learning sono tipicamente classificati in tre ampie categorie, caratterizzate dal tipo di feedback su cui si basa il sistema di apprendimento:

- Apprendimento supervisionato: vengono presentati al computer degli input di esempio ed i relativi output desiderati, con lo scopo di apprendere una regola generale in grado di mappare gli input negli output. Questo scenario è quello considerato nello sviluppo di questa tesi;
- Apprendimento non supervisionato: al computer vengono forniti solo dei dati in input, senza alcun output atteso, con lo scopo di apprendere una qualche struttura nei dati d'ingresso. L'apprendimento non supervisionato può rappresentare un obiettivo a se stante (ad esempio per la scoperta di pattern nascosti nei dati) o essere rivolto all'estrapolazione di caratteristiche salienti dei dati dette `feature`, utili per l'esecuzione di un altro task di machine learning;
- Apprendimento con rinforzo: il computer interagisce con un ambiente dinamico nel quale deve raggiungere un certo obiettivo (ad esempio, guidare un'automobile o affrontare un avversario in un gioco). Man mano che il computer esplora il dominio del problema, gli vengono forniti dei feedback in termini di ricompense o penalità, in modo da apprendere al meglio la soluzione migliore.

Mentre la suddivisione appena vista è incentrata sulla tipologia dei dati. Un altro metro di giudizio secondo il quale è possibile distinguere diverse categorie di task è il tipo di output atteso da un certo sistema di machine learning. Tra le principali categorie troviamo:

- **La classificazione**, nella quale gli input sono divisi in due o più classi e il sistema di apprendimento deve produrre un modello in grado di assegnare ad un input una o più classi tra quelle disponibili. Questi tipi di task sono tipicamente affrontati mediante tecniche di apprendimento supervisionato. Un esempio di classificazione è l'assegnamento di una o più etichette ad una immagine in base agli oggetti o soggetti contenuti in essa;
- **La regressione**, concettualmente simile alla classificazione con la differenza che l'output ha un dominio continuo e non discreto. Anch'essa è tipicamente affrontata con l'apprendimento supervisionato. Un esempio di regressione è la predizione di un valore continuo da un insieme di valori in un piano cartesiano.
- **Il clustering**, nel quale, come nella classificazione, un insieme di dati viene diviso in gruppi che però, a differenza di questa, non sono noti a priori. La natura stessa dei problemi appartenenti a questa categoria li rende tipicamente dei task di apprendimento non supervisionato.

3.2.1 Approcci di machine learning ai diversi problemi.

Allo stato attuale, esistono diversi tipi di approcci e tecniche per la progettazione e l'implementazione di sistemi computerizzati per l'apprendimento automatico. Tra i più importanti troviamo:

- **Gli alberi di decisione**: l'apprendimento fa uso, appunto, di un albero di decisione come modello di predizione. Il modello risultante permette di mappare le osservazioni riguardanti un oggetto a determinate conclusioni riguardanti il valore obiettivo relativo a quell'oggetto;
- **Il clustering**: come illustrato precedentemente, è una tecnica che, mediante l'analisi di un insieme di dati, permette la suddivisione di questo in sottoinsiemi (detti cluster) accomunati da uno o più criteri di similitudine;
- **La programmazione logica induttiva**: dall'inglese *inductive logic programming* (ILP), costituisce un approccio all'apprendimento di regole che utilizza la programmazione logica per la rappresentazione degli esempi di input, della conoscenza di base e delle ipotesi. A partire da una certa rappresentazione della conoscenza di base e del set di esempi sotto forma di fatti logici, un sistema di ILP può produrre un programma logico in grado di implicare tutti gli esempi positivi e non quelli negativi;

- **Gli algoritmi genetici**: sono algoritmi di ricerca euristica che cercano di imitare il processo della selezione naturale mediante l'uso di tecniche quali la mutazione e l'incrocio, con l'obiettivo di generare un nuovo genotipo (anche detto cromosoma, ovvero un insieme di parametri che definiscono una determinata soluzione) che rappresenti la soluzione migliore ad un determinato problema;

- **Le reti neurali artificiali**: un algoritmo di apprendimento mediante rete neurale artificiale, comunemente chiamato rete neurale, è un algoritmo che si ispira, sia dal punto di vista strutturale che del funzionamento, alle reti neurali biologiche. La computazione è strutturata in termini di gruppi interconnessi di neuroni artificiali. segue un approccio di tipo connettivista in cui il risultato si manifesta come comportamento emergente di un insieme interconnesso di unità semplici. Le reti neurali vengono spesso impiegate per la modellazione di relazioni complesse tra dati di input e output corrispondenti e per la ricerca di strutture nascoste nei dati;

- **Il deep learning**: rappresenta un'evoluzione delle reti neurali incoraggiata anche dal calo dei prezzi e dal grande sviluppo tecnico nel campo delle GPU general purpose avvenuto negli ultimi anni. Il deep learning si basa sull'utilizzo di particolari reti neurali costituite da una moltitudine di livelli nascosti di neuroni. Questo approccio trae ispirazione dal funzionamento della parte del cervello umano che si occupa della visione e dell'udito, e cerca di modellarne la struttura.

3.2.2 Machine learning contro Deep learning

> *«Più che una disciplina a se stante, il termine deep learning indica una serie di strumenti specifici per risolvere un particolare gruppo di problemi. Mentre il termine machine learning si riferisce alla disciplina, deep learning si riferisce al modo in cui la macchina impara. Il termine di traduce come apprendimento profondo perché ha a che vedere con le reti neurali.»* (Diario di un Analista, 2024)

Seppur i modelli di machine learning differenziano notevolmente in termini di performance rispetto a quei modelli che rientrano nella categoria "deep learning" per una questione meramente implementativa e algoritmica. Per lo sviluppo di questa tesi si è optato per modelli di machine learning e di seguito si riportano i principali motivi:

3.2.3 Specializzazione

Il deep learning è un insieme di tecniche e algoritmi sicuramente più adatto per problemi specifici, solitamente molto complessi.

Poiché le reti neurali possono modellare qualsiasi funzione se hanno risorse e tempi illimitati, queste sono usate per fronteggiare problemi che coinvolgono dati non strutturati, come testo, video e audio.

Per i dati tabellari come nel caso dei dati forniti dal kart game preso in esame, quindi strutturati, tecniche di machine learning meno complesse possono anche superare le performance di un algoritmo di deep learning. Anche se, le reti neurali sono più efficaci degli algoritmi tradizionali ad imparare da dataset molto più grandi (big data).

3.2.4 Complessità

Sebbene esistano algoritmi di machine learning tradizionale molto complessi, come XGBoost, gli algoritmi di deep learning sono per definizione più complessi.

Progettare modelli di deep learning è una delle sfide più importanti nella data science. Ogni giorno, i *machine learning engineers* si prodigano per innovare nel campo. Dunque, essendo lo scopo di questa tesi quello di collegare il comportamento di un agente intelligente con un modello di apprendimento si è voluto rimanere con una complessità gestibile sotto questo aspetto.

3.2.5 Richiesta computazionale

Tipicamente, gli algoritmi di machine learning tradizionale richiedono meno potenza computazionale rispetto a quelli di deep learning.

Questo perché le reti neurali possono sfruttare le GPU (*graphic processing units* o schede video) per aumentare la velocità di addestramento.

Oltre alle GPU, possono sfruttare anche le TPU (*tensor processing units*), che sono dei chip ottimizzati proprio per il deep learning.

Anche alcuni algoritmi di machine learning tradizionale possono sfruttare le GPU: tra questi ci sono XGBoost, LightGBM e Catboost. Ma in questo caso non avendo a disposizione delle GPU potenti, ma tutto il sistema doveva essere sviluppato in locale si è dovuto tenere in considerazione anche la potenza delle componenti hardware.

3.2.6 Interpretabilità

Un aspetto spesso non considerato quando si parla di differenze tra machine learning e deep learning è quanto i modelli e algoritmi appartenenti a questi siano interpretabili.

Le reti neurali sono tipicamente considerate delle *black box* - vale a dire che sappiamo come queste funzionino, ma non sappiamo come queste raggiungano il risultato atteso, né possiamo prevederlo.

Solo sperimentando con diverse architetture possiamo gradualmente avvicinarci alla configurazione migliore.

Algoritmi tradizionali invece, come gli alberi decisionali, sono facilmente interpretabili e comunicare come questi funzionino e come raggiungano i risultati è relativamente facile. Non a caso il modello di machine learning utilizzato in questa tesi fa riferimento proprio a questo.

3.2.7 Riassunto

Machine learning e deep learning sono essenzialmente la stessa cosa: metodi e tecniche che permettono ad una macchina di fare inferenze abbastanza precise da poter essere utilizzate in un contesto lavorativo e non. Queste metodiche dipendono dal contesto che abbiamo di fronte. È assolutamente superfluo usare tecniche di deep learning su dataset tabellari di dimensioni ridotte, perché un "banale" *random forest* potrebbe performare molto meglio e convergere più velocemente alle soluzioni.

E detto ciò analizziamo la prossima sezione dove viene riportato il comportamento del modello integrato assieme all'agente per la riuscita del bilanciamento di gioco.

3.3 Metriche del modello

Si scrive questa sezione perché si vuole evidenziare che l'approccio scelto quando si applica del machine learning in un dominio specifico comporta anche la scelta di alcune metriche che servono per misurare quanto è performante il modello utilizzato.

Siccome, il progetto rientra in quello che è un dominio di classificazione: perché si vuole valutare se il gioco risulta bilanciato o sbilanciato per poterci eventualmente applicare dell'aggiustamento dinamico della difficoltà. Allora, evidenzieremo maggiormente quelle che sono le metriche relative ai problemi di classificazione. Di seguito verrà riportata una lista delle metriche più utilizzate in questo ambito.

3.3.1 Accuracy

L'**accuracy** misura la proporzione di predizioni corrette rispetto al numero totale di predizioni effettuate. Formalmente, è definita come:

$$\text{Accuracy} = \frac{\text{Numero di predizioni corrette}}{\text{Totale delle predizioni}} \tag{3.1}$$

Pro:

- Facile da calcolare e interpretare.
- Adatta per dataset bilanciati dove le classi hanno dimensioni simili.

Contro:

- Può essere fuorviante in presenza di dataset sbilanciati, dove una classe dominante potrebbe portare a un'alta accuracy anche se il modello non è efficace nel distinguere le classi.
- Non considera i falsi positivi e i falsi negativi separatamente.

3.3.2 Precision

La **precision** misura la qualità delle predizioni positive ed è definita come:

$$\text{Precision} = \frac{\text{Vero Positivi (TP)}}{\text{Vero Positivi (TP)} + \text{Falsi Positivi (FP)}} \tag{3.2}$$

Pro:

- Importante in contesti dove i falsi positivi devono essere minimizzati (es. diagnosi mediche).

Contro:

- Non considera i falsi negativi, il che potrebbe risultare problematico in alcuni casi.

3.3.3 Recall

Il **recall**, noto anche come sensibilità o true positive rate, misura la capacità del modello di identificare correttamente i positivi ed è definito come:

$$\text{Recall} = \frac{\text{Vero Positivi (TP)}}{\text{Vero Positivi (TP)} + \text{Falsi Negativi (FN)}} \tag{3.3}$$

Pro:

- Utile in scenari in cui è fondamentale ridurre i falsi negativi, come il rilevamento di frodi o malattie rare.

Contro:

- Non considera i falsi positivi, il che potrebbe portare a un'elevata quantità di falsi allarmi.

3.3.4 F1-Score

L'**F1-Score** combina precision e recall in una singola metrica armonica:

$$\text{F1-Score} = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}} \tag{3.4}$$

Pro:

- Bilancia precision e recall, fornendo una visione più equilibrata della performance del modello.

Contro:

- Può essere meno intuitivo rispetto ad altre metriche.
- Non distingue i pesi assegnati a precision e recall.

3.3.5 Scelta dell'Accuracy

In questo lavoro, si è scelto di utilizzare l'**accuracy** come metrica di valutazione principale. Questa scelta è giustificata dal fatto che il dataset in analisi è bilanciato e che l'obiettivo principale è valutare la capacità del modello di classificare correttamente i dati nel loro complesso. In scenari più complessi o con dati sbilanciati, si potrebbe considerare l'adozione di metriche complementari come il *F1-Score* o una *confusion matrix* per un'analisi più dettagliata.

3.4 Collegamento tra machine learning ed agente

E' giunto il momento di descrivere il punto critico di questa tesi, cioè: la progettazione e la conseguente implementazione di un collegamento "più" intelligente tramite machine learning a quello che è il comportamento BDI dell'agente sviluppato.

3.4.1 Problematiche affrontate

Innanzitutto, si è dovuto pensare a come integrare nel sistema un modello di machine learning. Poiché diverse erano le limitazioni da tenere in considerazione:

1. Lo sviluppo dell'agente era avvenuto in JASON e in un ambiente Java su IntelliJ IDEA, mentre i modelli di machine learning vengono sviluppati principalmente in Python.
2. Quale modello scegliere tra i tanti?
3. Come fornire i dati al modello? Dunque, come mettere in relazione l'agente con la fase di traning del modello scelto?

Dunque, evidenziando queste prime problematiche sicuramente la prima cosa da fare è stata quella di evitare una dipendenza diretta tra videogioco kart game in Unity e il modello di machine learning. Questo perché il videogioco raccoglie delle statistiche (dunque, dei dati) per poi inviarli all'agente, come già descritto nel dettaglio nel capitolo precedente. Questi stessi dati, potevano essere anche inviati al modello di machine learning evitando l'agente, ma così facendo la sua utilità nel sistema sarebbe venuta meno. Mentre, lo scopo di questa tesi è proprio l'opposto. Cioè, studiare e sperimentare il comportamento dell'agente con il support di un modello di machine learning.

Allora, si è deciso di replicare la struttura utilizzata per l'agente in precedenza. Si è strutturato il sistema come un servizio esterno che espone una API RESTful le cui route sono utilizzate solamente dall'agente. Anche in questo caso torna utile l'utilizzo di un server Flask per lo sviluppo dell'API in "localhost" sulla porta 5001, tecnologia già vista nel capitolo precedente.

3.4.2 Logica comportamentale del modello

Ciò che si vuole descrivere a questo punto è l'idea di come questo modello dovrà comportarsi in conseguenza delle decisioni prese dall'agente.

Quando l'agente invia i propri `beliefs` al modello tramite l'azione specifica 2.6.3 viene invocato il seguente metodo dell'agente:

```
    public void sendDataToMLModel(List<String[]> structuredData) {

        JSONArray jsonArray = new JSONArray();
        for (String[] data : structuredData) {
            JSONObject jsonObj = new JSONObject();
            jsonObj.put("player_id", data[0]);
            jsonObj.put("parameter", data[1]);
            jsonObj.put("value", data[2]);
            jsonArray.put(jsonObj);
        }
```

```
        try {
            URL url = new URL("http://localhost:5001/send_data");
            HttpURLConnection conn = (HttpURLConnection) url.openConnection();
            conn.setRequestMethod("POST");
            conn.setRequestProperty("Content-Type", "application/json; utf-8");
            conn.setDoOutput(true);

            try(OutputStream os = conn.getOutputStream()) {
                byte[] input = jsonArray.toString().getBytes("utf-8");
                os.write(input, 0, input.length);
            }

            try(BufferedReader br = new BufferedReader(
                    new InputStreamReader(conn.getInputStream(), "utf-8"))) {
                StringBuilder response = new StringBuilder();
                String responseLine;
                while ((responseLine = br.readLine()) != null) {
                    response.append(responseLine.trim());
                }
                System.out.println("Response from ML model: " + response.toString()
                    );
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
```

Codice 3.1: Metodo dell'agente per inviare i propri beliefs come dataset del modello.

In breve, questo codice converte i beliefs dell'agente che erano stati precedentemente salvati (2.6.3) in una lista di stringhe, in un formato JSON così da inserirlo nel payload qundo successivamente viene eseguita una POST sulla route di acquisizione dei dati da parte del server del modello di machine learning.

```
@app.route('/send_data', methods=['POST'])
def receive_data():
    global data_storage
    incoming_data = request.get_json()

    for record in incoming_data:
        record['player_id'] = record['player_id'].replace("", "")
        record['parameter'] = record['parameter'].replace("", "")
        if record['parameter'] == 'position':
            record['value'] = json.loads(record['value'])

    with lock:
        data_storage.extend(incoming_data)

    return jsonify({"status": "data received"}), 200
```

Codice 3.2: Route POST per l'acquisizione del dataset.

I dati raccolti vengono salvati in un dizionario che prende il nome di *data_storage* e i dati inizialmente quando vengono inviati assumeranno la seguente forma forma, dovuto al loro modo di essere storicizzati come beliefs nel linguaggio AgentSpeak(L) dell'agente:

```
"{"'player_id'": "'1'", "'parameter'": "'speed'", "'value'": '45.0'}"
```

Dunque, è necessario effettuare una prima manipolazione dei dati in quanto non sono direttamente utilizzati in questo formato, e come si può vedere nel codice avviene una prima pulizia dei dati rimuovendo virgolette non necessarie ed effettuando l'unrolling dell'unico dato composto *position* che è composto a sua volta dalle coordinate x,y,z. Ottenendo:

```
"{"player_id": "1", "parameter": "speed", "value": 45.0}"
```

3.5 Training del Modello di Machine Learning

Il modello scelto per la fase di training è un Random Forest Classifier, un algoritmo di apprendimento supervisionato che combina più alberi decisionali per ottenere una maggiore accuratezza e robustezza. Questa scelta è stata motivata da:

- **Robustezza**: Il Random Forest è meno incline al *overfitting* rispetto a singoli alberi decisionali.
- **Flessibilità**: Può gestire dati non lineari e funziona bene sia per problemi di classificazione che di regressione, questo aspetto torna molto utile in vista del principio di "anticipation of change" in caso in cui si voglia potenziare il comportamento dell'agente passando da un problema di classificazione ad uno di regressione.
- **Interpretabilità**: Offre misure di importanza delle *features*, facilitando l'interpretazione delle decisioni.

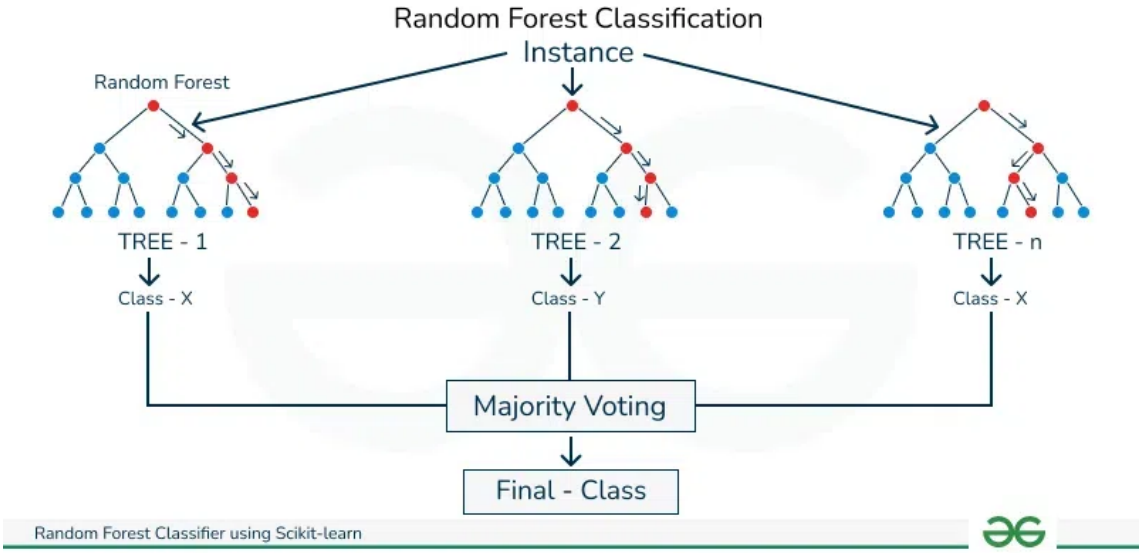

Figura 3.1: Rappresentazione della logica esecutiva di un Random Forest.

Inoltre, il modello è stato progettato per essere riaddestrato periodicamente, poiché i dati inviati dall' agente, che corrispondono ai suoi belefies (che si vuole ricordare equivalere ai dati inviati dal kart game in Unity) vengono aggiornati a intervalli regolari. Per affrontare questa sfida, il sistema raccoglie e preprocessa i dati ricevuti, li suddivide in set di training e test, e utilizza il nuovo batch di dati per aggiornare il modello.

3.5.1 Descrizione della logica effettuata per il traning del modello

A questo punto bisogna evidenziare ciascuna delle sottofasi svolte durante la fase di training:

1. Il pre-processing dei dati.
2. L'etichettatura dei dati con dei labels in quanto random forest opera in un'ambiente supervisionato.
3. l'addestramento con il calcolo dell'accuratezza.

Di seguito viene riportata una breve descrizione di quelle che sono le fasi descritte, evidenziando prima il codice e poi una breve descrizione del suo comportamento.

```
1 def preprocess_data():
      global data_storage
3     processed_data = {}
```

```
    for record in data_storage:
        player_id = record['player_id']
        parameter = record['parameter']
        value = record['value']

        if player_id not in processed_data:
            processed_data[player_id] = [None] * (len(all_parameters_sent_via_Json) +
                3) # Spazio per X, Y, Z

        if parameter == 'position':
            try:
                value_x = float(value[0])
                value_y = float(value[1])
                value_z = float(value[2])

                processed_data[player_id][-3:] = [value_x, value_y, value_z]
            except (ValueError, IndexError):
                continue
        else:
            try:
                value = float(value)
            except ValueError:
                continue

            if parameter in all_parameters_sent_via_Json:
                index = all_parameters_sent_via_Json.index(parameter)
                processed_data[player_id][index] = value


    for player, params in processed_data.items():
        processed_data[player] = [value for value in params if value is not None]

    return np.array(list(processed_data.values()))
```

Codice 3.3: Metodo per processare i dati durante la fase di training.

`preprocess_data()`

Questo metodo gestisce la pulizia e la trasformazione dei dati raccolti. I dati grezzi vengono estratti dalla struttura `data_storage` e trasformati in un formato adatto per il training:

- **Aggregazione per `player_id`**: I dati vengono raggruppati per giocatore, con ogni giocatore rappresentato da un array di parametri.
- **Gestione della posizione**: Se il parametro è `position`, i valori delle coordinate `x`, `y`, e `z` vengono separati e memorizzati come elementi distinti.

- **Rimozione dei valori** `None`: I parametri mancanti vengono eliminati per garantire la consistenza del dataset.

Infine, il metodo restituisce i dati preprocessati come un array bidimensionale di tipo `numpy`, poiché è un requisito per poter operare con un classificatore di tipo Random Forest.

```
def classify_balance(data):
    top_speed_diff_threshold = 6
    distance_diff_threshold = 10
    checkpoint_diff_threshold = 2
    acceleration_diff_threshold = 5
    position_diff_threshold = 200
    rank_diff_threshold = 15


    speeds = data[:, all_parameters_sent_via_Json.index('top_speed')]
    distances_to_finish = data[:, all_parameters_sent_via_Json.index('distance_to_finish')]
    checkpoints = data[:, all_parameters_sent_via_Json.index('checkpoints')]
    positions_x = data[:, -3]
    positions_y = data[:, -2]
    positions_z = data[:, -1]

    speed_diff = max(speeds) - min(speeds)
    distance_diff = max(distances_to_finish) - min(distances_to_finish)
    checkpoint_diff = max(checkpoints) - min(checkpoints)

    position_diffs = []
    for i in range(len(positions_x)):
        for j in range(i+1, len(positions_x)):
            pos_diff = np.sqrt((positions_x[i] - positions_x[j])**2 +
                               (positions_y[i] - positions_y[j])**2 +
                               (positions_z[i] - positions_z[j])**2)
            position_diffs.append(pos_diff)

    max_position_diff = max(position_diffs) if position_diffs else 0

    votes = 0

    if speed_diff > top_speed_diff_threshold:
        votes += 1
    if distance_diff > distance_diff_threshold:
        votes += 1
    if checkpoint_diff > checkpoint_diff_threshold:
        votes += 1
    if max_position_diff > position_diff_threshold:
        votes += 1

    if votes > NUMBER_OF_PARAMETERS_TO_EVALUATE / 2:
        return 1
    else:
```

```
        return 0
```

Codice 3.4: Metodo per etichettare il dataset prima di addestrare il modello.

classify_balance()

Questa funzione determina se una partita è bilanciata o meno, basandosi su differenze tra i parametri dei giocatori, ovvero quelli che sono i beliefs inviati dall'agente:

- **Soglie di squilibrio**: Differenze significative in velocità, distanza dal traguardo, checkpoint attraversati e posizione spaziale contribuiscono al calcolo.
- **Distanze Euclidee**: Le differenze di posizione tra i giocatori vengono calcolate utilizzando la distanza euclidea.
- **Sistema a voti**: Ogni parametro che supera una soglia contribuisce con un voto; il bilanciamento finale si basa sulla maggioranza dei voti.

La funzione restituisce `1` se la partita è squilibrata, altrimenti `0`.

```
@app.route('/train_model', methods=['POST'])
def train_model():
    global model

    training_data = preprocess_data()

    if len(training_data) == 0 or len(data_storage) == 0:
        return jsonify({"error": "Insufficient or malformed training data."}), 400

    labels = np.array([classify_balance(training_data) for _ in range(len(training_data))
        ])

    X_train, X_test, y_train, y_test = train_test_split(training_data, labels, test_size=0.2)

    model = RandomForestClassifier()
    model.fit(X_train, y_train)

    accuracy = model.score(X_test, y_test)

    return jsonify({"status": "model trained", "accuracy" : accuracy }), 200
```

Codice 3.5: Route invocata periodicamente dall'agente per effettuare il training.

train_model()

Questo metodo corrisponde all fase finale di training del modello dove ultimate quelle precedentemente descritte addestra il modello sul dataset etichettato e ne calcola

l'accuratezza, successivamente poi questo modello verrà usato per effettuare delle prediction:

1. **Preprocessing**: Viene chiamato `preprocess_data()` per trasformare i dati grezzi in un formato utile per il training.
2. **Generazione dei label**: Per ogni record, `classify_balance()` viene utilizzato per generare etichette di classificazione.
3. **Suddivisione del dataset**: I dati vengono divisi in set di training (80%) e di test (20%) utilizzando `train_test_split`.
4. **Training**: Il modello Random Forest viene addestrato sui dati di training.
5. **Valutazione**: L'accuratezza del modello viene calcolata sui dati di test, fornendo un'indicazione della sua performance.

L'accuratezza calcolata viene restituita come parte della risposta HTTP. E viene ritornata all'agente insieme ad un messaggio di *status* che ne conferma l'addestramento corretto.

3.6 Predizioni del modello

La fase di prediction consente di utilizzare il modello di machine learning addestrato per generare previsioni utilizzando il modello addestrato descritto precedentemente. Questo processo avviene tramite una specifica route (`/prediction`) e viene successivamente catturato dall'agente tramite una chiamata HTTP GET. Il codice di come questa è stata implementata è mostrato di seguito:

```
@app.route('/prediction', methods=['GET'])
def predict():
    global model

    if model is None:
        return jsonify({"error": "Model is not trained yet."}), 400

    with lock:
        if len(data_storage) == 0:
            return jsonify({"error": "No data available for prediction."}), 400

        player_data = preprocess_data()[-1, :]

    prediction = model.predict([player_data])

    return jsonify({"prediction": int(prediction[0])}), 200
```

Codice 3.6: Route utilizzata per effettuare predizioni

Descrivendo brevemente il comportamento di questo codice si ottiene che:

- **Controllo del modello**: La funzione verifica che il modello sia stato addestrato; in caso contrario, restituisce un messaggio di errore.
- **Dati per la prediction**: Viene utilizzato l'ultimo record disponibile in `data_storage`, preprocessato tramite la funzione `preprocess_data()`, per generare una previsione, che corrisponderà all'ultimo player in gara nel kart game.
- **Output della prediction**: Il risultato viene calcolato usando il metodo `model.predict()` del modello Random Forest. La predizione è restituita in formato JSON, con un valore numerico che rappresenta lo stato del bilanciamento della partita (`1` per *sbilanciato*, `0` per *bilanciato*).

3.6.1 Chiamata della Route da parte dell'Agente

L'agente utilizza il metodo `takePredictionFromModel()` per richiedere una predizione dal server Flask. Di seguito il codice Java per la chiamata HTTP:

```
public void takePredictionFromModel() {
    try {
        URL url = new URL("http://localhost:5001/prediction");
        HttpURLConnection conn = (HttpURLConnection) url.openConnection();
        conn.setRequestMethod("GET");

        int responseCode = conn.getResponseCode();
        if (responseCode == HttpURLConnection.HTTP_OK) {
            BufferedReader in = new BufferedReader(new InputStreamReader(conn.
                getInputStream()));
            String inputLine;
            StringBuilder content = new StringBuilder();
            while ((inputLine = in.readLine()) != null) {
                content.append(inputLine);
            }
            in.close();

            JSONObject jsonResponse = new JSONObject(content.toString());
            logger.info("Prediction received: " + jsonResponse.toString());

            int prediction = jsonResponse.getInt("prediction");
            sendPredictionToServer(prediction);

        } else {
            logger.warning("GET prediction request failed.");
        }
    } catch (Exception e) {
        logger.severe("Error retrieving prediction from ML model: " + e.getMessage());
    }
}
```

Codice 3.7: Metodo dell'agente per richiedere una prediction

Tralasciando quello che è il comportamento del metodo "*sendPredictionToServer(prediction);*" visibile in questo codice poiché verrà discusso nel capitolo conclusivo, il resto del funzionamento è il seguente:

- **Chiamata HTTP GET**: L'agente invia una richiesta GET al server Flask, alla route `/prediction`.
- **Gestione della risposta**: Se la risposta è valida (`HTTP_OK`), il JSON ricevuto viene parsificato per estrarre la predizione.
- **Gestione degli errori**: In caso di errore nella connessione o di risposta non valida, il metodo logga un messaggio di errore.

In questo modo si ottiene un sistema modulare che facilita l'aggiornamento periodico dei dati e la generazione dinamica di predizioni, garantendo un modello adattabile alle condizioni di gioco in tempo reale.

Dunque, si è giunti al termine della descrizione del servizio relativo al modello di machine learning integrato con il comportamento dell'agente intelligente. Si è mostrato il ciclo esecutivo di questi ultimi fino ad arrivare al modo in cui viene effettuata la classificazione del kart game da parte dell'agente. Rimane soltanto una domanda a cui rispondere: come si fa a rendere una partita bilanciata se l'agente predice uno sbilanciamento? Vedremo la risposta a questa domanda nel prossimo e ultimo capitolo.

Capitolo 4

Conclusioni

4.1 Trasferimento della Prediction dal dall'agente e al videogioco Unity

Dopo che l'agente ha ottenuto la predizione dal modello di machine learning tramite il metodo `takePredictionFromModel()`, questa viene inviata al suo server Flask per essere resa accessibile al videogioco Unity tramite una `route apposita`. Il codice Java utilizzato dall'agente per inviare la predizione al server Flask è mostrato di seguito:

```
private void sendPredictionToServer(int prediction) {
    try {
        URL url = new URL("http://localhost:5000/api/agent/prediction");
        HttpURLConnection conn = (HttpURLConnection) url.openConnection();
        conn.setRequestMethod("POST");
        conn.setDoOutput(true);

        JSONObject jsonPrediction = new JSONObject();
        jsonPrediction.put("prediction", prediction);

        conn.setRequestProperty("Content-Type", "application/json");
        try (OutputStream os = conn.getOutputStream()) {
            byte[] input = jsonPrediction.toString().getBytes("utf-8");
            os.write(input, 0, input.length);
        }

        int responseCode = conn.getResponseCode();
        if (responseCode == HttpURLConnection.HTTP_OK) {
            logger.info("Prediction sent successfully.");
        } else {
            logger.warning("POST prediction request failed with code: " +
                responseCode);
        }
    } catch (Exception e) {
        logger.severe("Error sending prediction to server: " + e.getMessage());
```

```
    }
}
```

Codice 4.1: Metodo per inviare la predizione tramite POST

La descrizione di questo metodo è la seguente:

- **Creazione della richiesta**: L'agente costruisce una richiesta HTTP POST verso l'endpoint `/api/agent/prediction` esposto dal server Flask.
- **Formato dei dati**: La predizione è incapsulata in un oggetto JSON con la chiave `prediction`.
- **Invio e gestione della risposta**: Il JSON è inviato nel corpo della richiesta, e il metodo gestisce la risposta HTTP, loggando il risultato.

Come detto in precedenza il server Flask espone due endpoint per la gestione della predizione:

- **POST** `/api/agent/prediction`: Consente di ricevere la predizione dall'agente per poi memorizzarla in una variabile globale.
- **GET** `/api/agent/prediction`: Rende disponibile la predizione al videogioco Unity.

Il codice Python per la gestione degli endpoint è il seguente:

```
@app.route('/api/agent/prediction', methods=['POST'])
def set_agent_prediction():
    global agent_prediction
    data = request.json
    agent_prediction = data.get('prediction')

    if agent_prediction is not None:
        return jsonify({"message": "Prediction set successfully", "prediction":
            agent_prediction}), 200
    else:
        return jsonify({"error": "Prediction value not provided"}), 400

@app.route('/api/agent/prediction', methods=['GET'])
def get_agent_prediction():
    if agent_prediction is not None:
        return jsonify({"prediction": agent_prediction}), 200
    else:
        return jsonify({"error": "No prediction available"}), 404
```

Codice 4.2: Gestione della predizione nel server Flask dell'agente.

4.1.1 Integrazione con Unity

Così facendo, Unity può accedere alla predizione tramite l'endpoint "/api/agent/-prediction". E questo avviene tramite l'utilizzo di un gameObject appositamente sviluppato che ha il compito periodicamente di invocare la route per catturare quello che è il valore della predizione. Il seguente codice Unity, sviluppato nella classe che prende il nome di `AgentResponseHandler`, implementa la logica per ricevere la predizione dell'agente dal server Flask, applicare il Dynamic Difficulty Adjustment (DDA) ai kart con un rango specifico, e aggiornare le loro statistiche in base alla predizione.

```
1  public class AgentResponseHandler : MonoBehaviour
   {
3      private const int PLAYERS_TO_APPLY_DDA_BELOW_WHAT_RANK = 4;
       private const float INCREMENT_VALUE = 10f;
5      private string agentPredictionUrl = "http://localhost:5000/api/agent/prediction";
       private CheckpointTracker checkpointTracker;
7
       void OnEnable()
9      {
           CheckpointTracker.OnCheckpointTrackerEnabled +=
               OnCheckpointTrackerEnabled;
11     }

13     void OnDisable()
       {
15         CheckpointTracker.OnCheckpointTrackerEnabled -=
               OnCheckpointTrackerEnabled;
       }
17
       void OnCheckpointTrackerEnabled()
19     {
           checkpointTracker = FindObjectOfType<CheckpointTracker>();
21
           if (checkpointTracker != null)
23         {
               Debug.Log("CheckpointTracker found!");
25             StartCoroutine(GetAgentPrediction());
           }
27         else
           {
29             Debug.LogError("CheckpointTracker still not found.");
           }
31     }

33     IEnumerator GetAgentPrediction()
       {
35         while (true)
           {
37             using (UnityWebRequest www = UnityWebRequest.Get(
```

```
            agentPredictionUrl))
        {
            yield return www.SendWebRequest();

            if (www.result == UnityWebRequest.Result.Success)
            {
                Debug.Log("Agent prediction response received: " + www.
                    downloadHandler.text);

                AgentPredictionResponse response = JsonUtility.FromJson<
                    AgentPredictionResponse>(www.downloadHandler.text);

                if (response != null && response.prediction > 0)
                {
                    ApplyDDAToKarts();
                }
            }
            else
            {
                Debug.LogError("Error receiving agent prediction: " + www.error);
            }
        }

        yield return new WaitForSeconds(5);
    }
}

void ApplyDDAToKarts()
{
    Dictionary<ArcadeKart, string> karts = checkpointTracker.kartIdMap;

    var kartsBelowRank = karts.Keys.Where(kart =>
    {
        float distanceToFinish = Vector3.Distance(kart.transform.position,
            checkpointTracker.finishLineTransform.position);
        int rank = 1 + checkpointTracker.kartIdMap.Keys.Count(k => Vector3.
            Distance(k.transform.position, checkpointTracker.finishLineTransform.
            position) < distanceToFinish);
        return rank > 1 && rank <=
            PLAYERS_TO_APPLY_DDA_BELOW_WHAT_RANK;
    });

    foreach (var kart in kartsBelowRank)
    {
        ArcadeKart.Stats stats = kart.baseStats;
        stats.TopSpeed += stats.TopSpeed * INCREMENT_VALUE;
        kart.baseStats = stats;
    }
    Debug.Log($"Updated stats for Kart {kart.name}. New top speed: {kart.
        baseStats.TopSpeed}, New acceleration: {kart.baseStats.Acceleration}");
```

```
    }

    [System.Serializable]
    public class AgentPredictionResponse
    {
        public float prediction;
    }
}
```

Codice 4.3: Codice della classe utilizzata per catturare la predizione in Unity.

Anche se la logica del funzionamento della classe è stata descritta, si vuole evidenziare ancora più nel dettaglio il suo comportamento:

1. **Inizializzazione del `CheckpointTracker`:** Quando il `CheckpointTracker` è abilitato, il metodo `OnCheckpointTrackerEnabled` associa la classe al tracker dei checkpoint, che monitora la posizione dei kart durante la gara.
2. **Richiesta della predizione al server Flask:** Il metodo GetAgentPrediction utilizza una coroutine per inviare richieste GET al server Flask sull'endpoint `/api/agent/prediction` ogni 5 secondi. Se la risposta è valida e contiene una predizione maggiore di 0, viene chiamato il metodo `ApplyDDAToKarts`.
3. **Applicazione del Dynamic Difficulty Adjustment (DDA):**
   - **Filtraggio dei kart:** Vengono considerati solo i kart con rango maggiore di 1 e minore o uguale a `PLAYERS_TO_APPLY_DDA_BELOW_WHAT_RANK` (valore predefinito: 4).
   - **Aggiornamento delle statistiche:** Per ogni kart selezionato, la velocità massima (`TopSpeed`) viene aumentata di una percentuale definita da `INCREMENT_VALUE` (valore predefinito: 10%).
4. **Log delle modifiche:** Il sistema registra nel debug console il nome del kart e le sue nuove statistiche.

4.2 Risultati finali

Quindi, ultimato questo lavoro di tesi si è ottenuto un sistema di adattamento dinamico della difficoltà (DDA) di videogiochi multiplayer tramite l'utilizzo di un agente intelligente supportato da un modello di machine learning. Il sistema finale risulta funzionante, replicabile e capace di adattarsi a futuri cambiamenti per via delle scelte implementative fatte durante lo sviluppo. Grazie alla comunicazione bidirezionale tra l'agente e il gioco, è stato possibile adattare dinamicamente la difficoltà, garantendo un'esperienza più bilanciata e competitiva.

Il sistema ha dimostrato una notevole capacità di individuare i giocatori in difficoltà, intervenendo in tempo reale per migliorare le loro performance senza compromettere l'integrità della gara. In particolare, i kart con posizioni svantaggiate

hanno beneficiato di modifiche alle loro statistiche, come un aumento della velocità massima, che ha permesso loro di ridurre la distanza dai leader. Questo approccio ha portato a una maggiore competitività, evidenziata da un numero più elevato di sorpassi e una percezione generale di equità nel corso delle gare.

Dal punto di vista tecnico, il sistema si è dimostrato affidabile, con tempi di risposta rapidi e una gestione efficace della comunicazione tra l'agente e Unity. Anche il sistema di distribuzione delle predizioni tramite API REST ha funzionato in modo fluido, integrandosi perfettamente nel gameplay senza introdurre ritardi percepibili, dovuto anche all'utilizzo di intervalli di tempo molto piccoli per effettuare il comportamento periodico, come l'invio delle statistiche e la cattura delle predizioni.

Un gruppo di tester ha valutato l'esperienza di gioco offerta dal sistema. La maggior parte ha riscontrato un significativo miglioramento nel coinvolgimento e nella competizione durante le gare. Inoltre, il bilanciamento automatico della difficoltà è stato percepito come naturale e non intrusivo, a conferma dell'efficacia delle predizioni e del loro utilizzo nel contesto di gioco.

4.3 Conclusione

In conclusione, il progetto ha dimostrato come l'integrazione di tecnologie di machine learning nei videogiochi multiplayer possa migliorare sensibilmente l'esperienza dei giocatori, promuovendo una competizione equilibrata e stimolante. Questi risultati aprono nuove prospettive per lo sviluppo di sistemi intelligenti capaci di adattare il gameplay in base alle esigenze dinamiche dei giocatori, gettando le basi per innovazioni future nel settore dei videogiochi e della programmazione ad agenti.

4.4 Sviluppi futuri

Nonostante i risultati positivi, sono emerse alcune limitazioni che rappresentano opportunità per sviluppi futuri. Ad esempio, l'accuratezza del modello potrebbe essere ulteriormente migliorata ampliando il dataset di training e introducendo caratteristiche più complesse per rappresentare i comportamenti dei giocatori. Inoltre, l'ottimizzazione del sistema per scenari con un numero maggiore di giocatori o con architetture server distribuite potrebbe rappresentare una sfida interessante da affrontare.

Bibliografia

Corporation O., 2024, *What is Machine Learning?*, https://www.oracle.com/it/artificial-intelligence/machine-learning/what-is-machine-learning/

DB Game Academy 2024, *Game Engine: sviluppare videogiochi*, https://dbgameacademy.it/game-engine-sviluppare-videogiochi/

Diario di un Analista 2024, *Machine Learning vs Deep Learning*, https://www.diariodiunanalista.it/posts/machine-learning-vs-deep-learning/

Franklin S., Graesser A., 1996, in Müller J. P., Wooldridge M. J., Jennings N. R., eds, LNCS, Vol. 1193, Intelligent Agents III. Agent Theories, Architectures, and Languages. Springer, pp 21–35, doi:10.1007/BFb0013570, http://www.springerlink.com/content/w5m511674402vr07/

HTML.it 2024, *Introduzione allo scripting*, https://www.html.it/pag/45528/introduzione-allo-scripting/

Intelligenza Artificiale 2024, *Agenti intelligenti: cosa sono?*, https://www.intelligenzaartificiale.it/agenti-intelligenti/

Mirror Networking 2024, *Mirror Networking*, https://mirror-networking.com/

Mitchell T. M., 1997, *Machine Learning*. McGraw Hill, New York

Omicini A., 2013, *Nature-inspired Coordination Models: Current Status, Future Trends*, ISRN Software Engineering, 2013

Omicini A., Viroli M., 2011, *Coordination Models and Languages: From Parallel Computing To Self-Organisation*, The Knowledge Engineering Review, 26, 53

Pallets Projects 2024, *Flask Documentation*. https://flask.palletsprojects.com/en/stable/

Turing A. M., 1950, *Computing Machinery and Intelligence*, Mind, 59, 433

Unity Technologies 2024b, *Unity Networking Overview*. https://docs.unity3d.com/540/Documentation/Manual/UNetOverview.html

Unity Technologies 2024a, *UnityWebRequest*. https://docs.unity3d.com/ScriptReference/Networking.UnityWebRequest.html

University of Bologna 2024, *Slides Jason*, http://lia.deis.unibo.it/corsi/2007-2008/SMA-LS/slides/Xa-SlidesJason.pdf