Università degli Studi di Bari Aldo Moro

Dipartimento di Informatica e Tecnologie per la Produzione del Software

Strategie Evolutive per il controllo coordinato di Sistemi Multi-Agente

Laureando:
Giuseppe Trivisano

Relatore:
Dott.ssa Gabriella Casalino
Co-relatori:
Dott.ssa Alessandra Vitanza
Dott. Paolo Pagliuca

a.a 2023/2024

Indice

Elenco delle figure

Elenco delle tabelle

Abstract

Lo studio dei comportamenti collettivi osservati in natura trova naturale applicazione pratica nei sistemi multi-agente. Comprendere le condizioni fondamentali alla base di comportamenti complessi presenta un interessante problema di ricerca per lo studio di fenomeni sociali, con possibili applicazioni rilevanti anche nel mondo della robotica. L'obiettivo di questa tesi è indagare l'applicazione delle strategie evolutive per il controllo coordinato di sistemi multi-agente per lo specifico problema di locomozione e aggregazione tra più agenti in un ambiente simulato, e studiarne le condizioni minime affinché ciò avvenga, insieme all'analisi delle performance. A questo scopo, sono state formulate e descritte delle funzioni di fitness da applicare durante le simulazioni e tre diverse metodologie per il calcolo complessivo delle componenti: sommativo, sequenziale e incrementale. Le simulazioni sono state svolte attraverso il simulatore Evorobotpy3, utilizzando l'algoritmo evolutivo OpenAI-ES. I risultati mostrano come la definizione della funzione di fitness e la sequenzialità degli obiettivi incidano sulle capacità degli agenti di sviluppare comportamenti coordinati. Le conclusioni della tesi sottolineano la necessità di utilizzare strategie evolutive più raffinate per bilanciare i due comportamenti analizzati (locomozione e aggregazione) e vengono suggerite alcune possibili direzioni future per il proseguo della ricerca.

Capitolo 1

Introduzione

I sistemi multi-agente [5, 60] rappresentano un settore di studio fondamentale nell'ambito dell'intelligenza artificiale e della robotica, dove più agenti autonomi interagiscono all'interno di un ambiente condiviso per risolvere problemi complessi, spesso in modo decentralizzato e auto-organizzato. La robotica di sciame [13, 50], che ne è una branca particolare, si ispira ai comportamenti collettivi osservabili in natura, come quelli delle colonie di insetti o degli stormi di uccelli, e si pone l'obiettivo di ottenere comportamenti emergenti complessi a partire da regole semplici applicate a singoli agenti, i quali spesso sono dotati di una rete neurale per eseguire i processi decisionali individuali. Tra le varie metodologie utilizzabili per affrontare problemi di questo tipo hanno particolare successo gli algoritmi evolutivi [26] e, nello specifico, le strategie evolutive [46, 53], che consentono di risolvere problemi di ottimizzazione complessi con un approccio basato sui principi darwiniani dell'evoluzione naturale. Quando si cerca di fare emergere in un sistema multi-agente più di un comportamento si è di fronte ad un problema di ottimizzazione multi-obiettivo [35], la cui risoluzione rappresenta una sfida non banale.

In questo lavoro di tesi ci siamo proposti di esplorare l'utilizzo delle strategie evolutive e di definire le condizioni minime per far emergere i comportamenti di locomozione e aggregazione in un sistema multi-agente simulato, prendendo spunto dal lavoro presente nell'articolo [42]. Nello specifico ci siamo concentrati sulla formulazione e sulle modalità di applicazione della funzione di fitness che guida il processo evolutivo e sullo studio di alcune proprietà fondamentali di questo specifico problema.

L'elaborato si articola in una prima parte teorica, in cui vengono approfonditi i concetti di sistemi multi-agente, reti neurali, algoritmi evolutivi e vengono definite brevemente le caratteristiche e alcuni metodi di ottimizzazione multi-obiettivo. Successivamente, viene descritta la metodologia sperimentale illustrando alcune componenti specifiche degli esperimenti quali il simulatore Evorobotpy3 [45] e l'algoritmo OpenAI-ES [51], utilizzato per l'ottimizzazione dei parametri delle reti neurali che controllano

gli agenti. Infine vengono descritte le scelte progettuali prese e i risultati ottenuti nel corso dei vari esperimenti e le relative conclusioni.

L'obiettivo generale di questo lavoro è contribuire alla comprensione dei meccanismi evolutivi che possono favorire l'emergenza di comportamenti coordinati nei sistemi multi-agente, aprendo la strada a possibili applicazioni pratiche in scenari come la robotica di sciame, la sorveglianza ambientale e le missioni di esplorazione autonoma.

Capitolo 2

Background Teorico

In questo capitolo verranno approfonditi tutti i concetti teorici necessari alla comprensione ed utilizzati nel lavoro di tesi esposto nei capitoli successivi.

2.1 Sistemi multi-agente: definizioni

Un sistema multi-agente (o MAS, *Multi-Agent System*) [5, 60] è un sistema composto da più agenti intelligenti che interagiscono in un ambiente comune per risolvere problemi complessi, raggiungere obiettivi comuni o competere sulle risorse disponibili.

Un agente in un MAS ha la caratteristica di essere autonomo, ovvero deve decidere da sé quali azioni intraprendere in base all'obiettivo che è stato addestrato a raggiungere. Poiché un agente ha solo una visione locale del sistema, ciò implica che ognuno ha una conoscenza limitata alle sole componenti rilevanti per il proprio compito specifico. Questo porta all'ultima caratteristica fondamentale dei MAS, ovvero la decentralizzazione, secondo cui ogni agente opera in modo indipendente basandosi sui dati locali a sua disposizione e sui propri processi decisionali, senza affidarsi ad un'unità di controllo centrale [60]. Quest'ultima è la caratteristica fondamentale per distinguere un MAS da un classico sistema monolitico.

L'ambiente è lo spazio in cui gli agenti devono operare. Può essere *fisico*, come nel caso di robot o droni, oppure *virtuale*, come nel caso di sistemi simulati e/o videogiochi. Può, inoltre, essere *statico* o *dinamico*: nel primo caso l'ambiente muta solo in seguito alle azioni degli agenti che vi sono immersi, mentre nel secondo caso esso è soggetto anche ad altri processi, esterni agli agenti. Un ambiente fisico, ad esempio, è altamente dinamico. Quanto più è dinamico un ambiente tanto più un agente deve avere alte capacità di adattamento e una forte reattività. Oltre a queste classificazioni, un ambiente può essere anche *deterministico* o *non deterministico*: in un ambiente deterministico ogni azione ha un effetto garantito e certo, mentre, al contrario, in un ambiente non deterministico ogni azione produce effetti incerti, come accade nella maggior parte delle

applicazioni reali. La progettazione di un agente dipende dall'ambiente nel quale dovrà operare.

Nella definizione introduttiva si è parlato di agenti "intelligenti". Con questo aggettivo si vuole esprimere la capacità di un agente di percepire l'ambiente nel quale è immerso attraverso degli input e, basandosi su questi, di elaborare autonomamente una reazione, per poi metterla in atto attraverso un sistema di output. Questi processi decisionali possono essere implementati da architetture basate su regole, in cui ad ogni stimolo è associata una reazione predefinita, oppure da architetture di *machine learning*[1], come le reti neurali. Le prime sono molto limitate e si adattano bene solo in contesti semplici e per lo più deterministici, mentre le seconde consentono di far fronte a problemi complessi in ambienti non deterministici.

Gli agenti possono essere programmi informatici, robot o altri tipi di entità autonome. Nel caso dei robot, gli input sono forniti dai sensori e gli output sono eseguiti dagli attuatori. Un sottoinsieme specifico dei MAS che si occupa di agenti robot è la robotica di sciame (*swarm robotics*) [13, 50]. In tal caso si può parlare di sistema multi-robot (o MRS, *Multi-Robot System*) [19].
Nello specifico, la robotica di sciame è lo studio di come progettare sistemi indipendenti di robot senza controllo centralizzato che fanno affidamento solo sulle interazioni locali tra robot e sulle interazioni tra i singoli robot e l'ambiente [13]. La progettazione di tali sistemi è guidata dalle idee emerse nel campo dell'intelligenza di sciame (*swarm intelligence*) [30], ovvero la disciplina che si occupa di sistemi naturali e artificiali composti da molti individui che si coordinano utilizzando il controllo decentralizzato e l'auto-organizzazione. Esempi di sistemi biologici di questo tipo sono le colonie di formiche e termiti, i banchi di pesci, gli stormi di uccelli, le mandrie di animali terrestri [12] e, in generale, tutte le situazioni in natura in cui si verifica il comportamento a sciame. Regole individuali relativamente semplici possono produrre un ampio ventaglio di comportamenti complessi dello sciame, a patto che ci sia una qualche forma di comunicazione nel gruppo che possa costituire un sistema di feedback costante. Rispetto ai robot individuali, uno sciame può generalmente scomporre le missioni assegnategli in sottocompiti. Inoltre, uno sciame è più robusto al fallimento parziale ed è più flessibile e scalabile. Esempi classici di attività che potrebbero essere affrontate con successo grazie alla robotica di sciame sono la ricerca e il salvataggio, l'esplorazione planetaria o subacquea e la sorveglianza. La robotica di sciame è stata anche utilizzata per interessanti fini di ricerca scientifica, ad esempio, lo studio dell'evoluzione, della comunicazione e del processo decisionale collettivo [6, 52].

La ricerca sui MAS, in generale, affronta una serie di problemi tecnici, tra cui: la comunicazione e la propagazione delle informazioni tra gli agenti; l'emergenza di norme, convenzioni e ruoli all'interno del sistema stesso; la progettazione di MAS per incentivare determinati comportamenti negli agenti; e la progettazione di algoritmi che

[1] Il *machine learning* è un sottoinsieme dell'intelligenza artificiale che utilizza metodi statistici per migliorare le performance di un algoritmo nell'identificare dei pattern nei dati.

consentano a uno o più agenti di raggiungere obiettivi specifici.
Gli ultimi due problemi sono quelli in parte indagati in questo lavoro di tesi.

Una vasta gamma di applicazioni può essere implementata utilizzando metodologie MAS, tra cui la gestione del traffico urbano, la gestione di magazzini e fabbriche a carico dei robot, il trading automatizzato, la simulazione di scenari complessi nei videogiochi, le operazioni di salvataggio tramite robot, la simulazione di dinamiche sociali e economiche per fini analitici, ecc...

2.1.1 Tecniche di apprendimento: approcci filogenetico e ontogenetico

Specialmente nei contesti di intelligenza artificiale e modellazione, i concetti di filogenesi e ontogenesi [15] si intrecciano e offrono due differenti modalità di sviluppo dei MAS e due strumenti concettuali per comprenderne la natura intrinseca.

La filogenesi si occupa dello studio dell'evoluzione delle specie o di gruppi di specie nel corso di milioni di anni. Si basa prevalentemente su concetti evoluzionistici come la selezione naturale, la mutazione, la deriva genetica e la migrazione. Questi fattori determinano il cambiamento morfologico, genetico e comportamentale della specie.
L'ontogenesi, invece, studia i cambiamenti e le trasformazioni del singolo individuo durante tutto il suo ciclo di vita. Include processi come l'apprendimento comportamentale, la maturazione fisica e le influenze ambientali. I risultati della filogenesi si ripercuotono sull'ontogenesi, nel senso che le capacità di sviluppo individuali sono frutto di un lungo processo evolutivo.

Nel contesto dell'intelligenza artificiale e dello sviluppo di agenti in un MAS, questi due concetti possono essere usati per approcci diversi ma con obiettivi simili: l'approccio filogenetico consiste nel simulare l'evoluzione di intere popolazioni di agenti per molte generazioni attraverso l'uso degli algoritmi evolutivi [26], mentre l'approccio ontogenetico consiste nel simulare il processo di apprendimento di un singolo agente che migliora le proprie capacità in risposta all'ambiente attraverso l'uso dell'apprendimento per rinforzo (*reinforcement learning*) [54].

Ad esempio, se si volesse verificare l'emergere di uno specifico comportamento in un MAS si potrebbe usare l'approccio filogenetico facendo evolvere insieme tutti gli agenti per generazioni (secondo certe regole espresse dagli iperparametri dell'algoritmo evolutivo usato), oppure si potrebbe usare l'approccio ontogenetico e considerare ogni agente nel sistema come indipendente e in grado di apprendere individualmente attraverso tecniche di rinforzo e punizione, portando il suo comportamento individuale ad una evoluzione.

2.2 Reti neurali: definizioni

Come accennato precedentemente, le reti neurali possono essere usate come sistema decisionale che permette agli agenti di valutare gli input e produrre degli output adeguati al compito e al contesto.
Nell'ambito dell'apprendimento automatico, le reti neurali artificiali (o ANN, *Artificial Neural Networks*) [11] sono un modello computazionale che si ispira alla struttura di una rete neurale biologica e si basa sul concetto di "neurone artificiale".

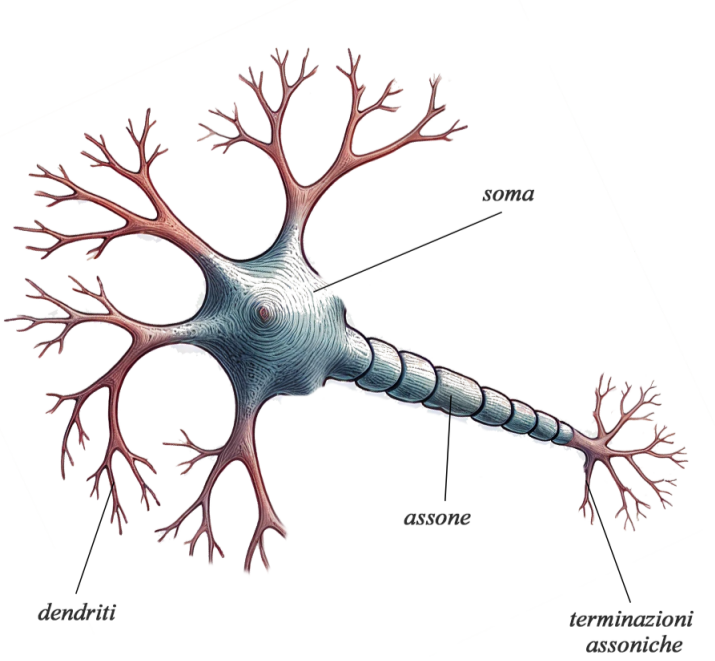


Figura 2.1: *Rappresentazione semplificata di un neurone naturale**

Un neurone naturale (Figura 2.1) è composto da diverse parti, riassumibili in: dendriti, soma e assone. I dendriti sono le ramificazioni che ricevono il segnale elettrico da altri neuroni e lo conducono verso il soma. Il soma risulta essere la parte centrale del neurone che si occupa di elaborare le informazioni ricevute dai dendriti. Infine, l'assone è un lungo prolungamento che trasporta il segnale elettrico fino alle terminazioni assoniche, permettendo così la comunicazione con altri neuroni attraverso le sinapsi.

Un neurone artificiale (Figura 2.2) è una semplice struttura matematica composta da n input, dei pesi w_{ij} associati agli archi che collegano ciascun input i con il neurone j, una funzione di attivazione e un output. I valori degli input, dei pesi e dell'output sono valori numerici reali.
In un neurone artificiale viene calcolata la somma pesata degli input, alla quale viene

*Immagine generata con AI

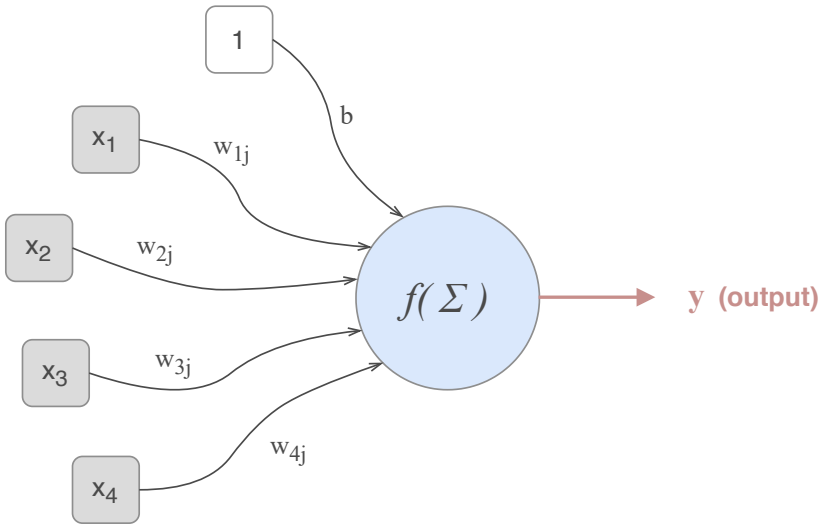


Figura 2.2: *Schema di un neurone artificiale*

aggiunto un valore di *bias* che consente di spostare la funzione di attivazione lungo l'asse delle ascisse e quindi rende flessibile la soglia di attivazione del neurone. Il risultato ottenuto rappresenta l'argomento della funzione di attivazione:

$$a = (x_1 w_{1j} + x_2 w_{2j} + \ldots + x_n w_{nj}) + b = \left(\sum_{i=1}^{n} x_i w_{ij} \right) + b \tag{2.1}$$

La funzione di attivazione può essere una funzione lineare, come la funzione identità, oppure una funzione non lineare, come la funzione *threshold*, la tangente iperbolica, la sigmoide, la ReLU (*Rectified Linear Unit*) [2] e la *softmax* [18]. Ognuna di queste funzioni ha le sue peculiarità ed ha risultati migliori in determinate applicazioni rispetto alle altre. In generale le funzioni lineari non consentono di trovare pattern complessi nei dati per via della loro "rigidità" e della loro sostanziale proporzionalità rispetto all'argomento della funzione, al contrario delle funzioni non lineari che, per questo motivo, sono quelle principalmente utilizzate nella maggior parte delle applicazioni che usano le reti neurali. Il risultato della funzione di attivazione rappresenta l'output del neurone artificiale, quindi $y = f(a)$.

Una rete neurale artificiale è composta da uno o più neuroni artificiali connessi tra loro. La semplice rete mostrata nella Figura 2.2 rappresenta il *Perceptron* di Rosenblatt [48], un modello ispirato ai lavori di McCulloch e Pitts [37], che avevano teorizzato una rete neurale più semplice ma priva di un meccanismo di apprendimento.

Una rete neurale è generalmente organizzata in strati di neuroni: lo strato di input (*input layer)* e lo strato di output (*output layer*) sono necessari e onnipresenti, ma tra

loro spesso si trovano uno o più strati interni o nascosti (*hidden layer*). Quando gli strati nascosti sono due o più si parla di *deep neural network*, se invece è solo uno si parla di *shallow neural network*. In una rete neurale i neuroni possono essere collegati tra loro in diversi modi, e in base a ciò le reti possono essere ricorrenti (RNN, *Recurrent Neural Network*) [29], quando gli output di uno o più neuroni vengono mandati in input alla rete stessa, o *feed-forward* [55], quando ogni neurone di uno strato è connesso ad ogni neurone dello strato immediatamente successivo (Figura 2.3). La caratteristica principale delle RNN è che, grazie alle connessioni interne a ciclo chiuso tra i neuroni, sono in grado di avere una "memoria" che gli consente di valutare dati in cui l'ordine di input è importante rispetto all'output atteso. Le RNN trovano con successo applicazione in compiti sequenziali, come il NLP (*Natural Language Processing*), il riconoscimento vocale e il riconoscimento delle serie temporali.

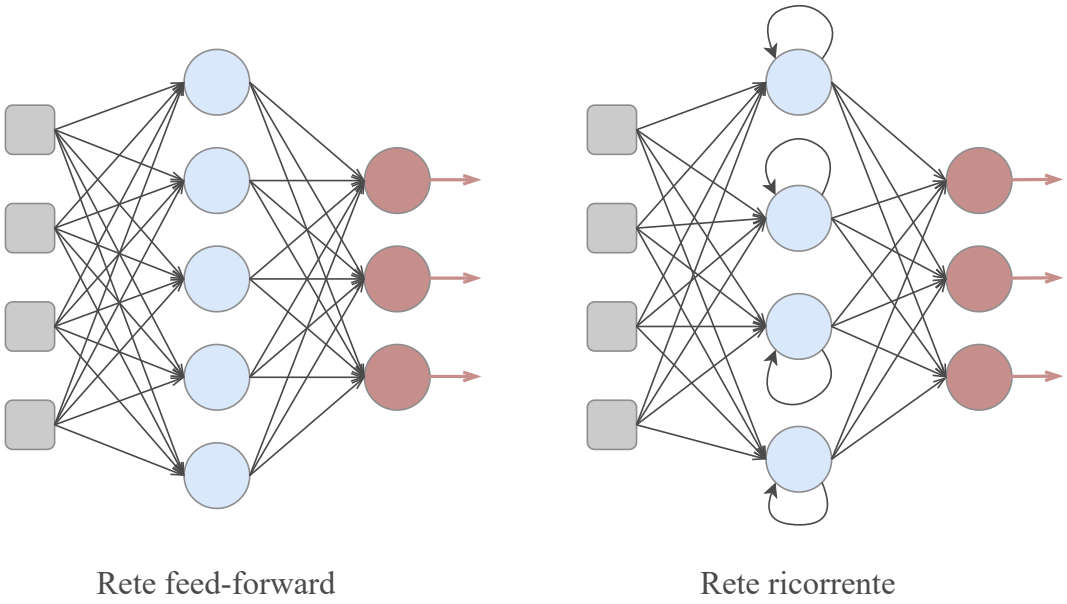


Figura 2.3: *Schema di reti neurali feed-forward e ricorrente*

Un altro tipo di ANN sono le reti neurali a impulso (SNN, *Spiking Neural Networks*) [36], considerate come le reti neurali di terza generazione e oggetto di interessanti ricerche scientifiche. Sono state progettate per imitare in modo più fedele il comportamento delle reti neurali biologiche e colmare ulteriormente il divario tra neuroscienze e *machine learning*. Nelle reti neurali biologiche l'informazione è trasmessa attraverso impulsi elettrici discreti nel tempo e non attraverso attivazioni continue basate su funzioni matematiche, come viene fatto nelle reti neurali artificiali descritte precedentemente. I neuroni biologici, infatti, comunicano fra loro attraverso *spike*, ovvero impulsi elettrici che vengono propagati quando il potenziale di membrana di un neurone supera una certa soglia, e il timing di emissione di questi impulsi può fungere da discriminante

per la codifica delle informazioni [1], permettendo alle SNN di sfruttare la dimensione temporale nel processamento dei dati. I modelli di *neuroni spike* più diffusi sono il LIF (*Leaky Integrate and Fire*) [34], che è il modello più semplice e con un costo computazionale più basso, l'HH (*Hodgkin–Huxley*) [39], che è biologicamente più accurato ma computazionalmente costoso, e infine il modello *Izhikevich* [28], che combina la semplicità computazionale del LIF con la capacità di riprodurre un'ampia gamma di comportamenti neuronali tipici del modello HH.
Le SNN offrono vantaggi significativi rispetto alle ANN tradizionali in termini di elaborazione delle informazioni temporali, efficienza energetica e plausibilità biologica, poiché imitano meglio il funzionamento del cervello umano [61]. Tuttavia sono ancora presenti alcune problematiche relative all'implementazione ottimale di queste reti, soprattutto per quanto riguarda l'architettura e l'addestramento, vista l'assenza di un metodo di retropropagazione efficace per l'ottimizzazione dei parametri.

2.2.1 Addestramento di una rete neurale

Per far sì che una rete neurale artificiale diventi capace di rispondere correttamente agli input in base al problema posto bisogna addestrarla, ovvero bisogna fare in modo che, attraverso un algoritmo, vengano modificati opportunamente i valori dei pesi della rete (compresi i bias) affinché i calcoli eseguiti al suo interno possano produrre le risposte corrette per più classi di input possibili.

I tre principali paradigmi di apprendimento sono l'apprendimento supervisionato (*supervised learning*), l'apprendimento non supervisionato (*unsupervised learning*) e l'apprendimento per rinforzo (*reinforcement learning*) [54].
Nell'apprendimento supervisionato la rete neurale viene addestrata con dati etichettati, ovvero dati di input associati ad output noti. In fase di addestramento la rete riceve degli input da un sottoinsieme dei dati disponibili (*training set*), produce un output e, successivamente, attraverso una funzione di perdita (o *loss function*) viene misurata la differenza tra l'output prodotto dalla rete e l'output noto relativo all'input. Calcolata questa quantità, viene utilizzato un algoritmo di retropropagazione degli errori (*error backpropagation*) [49] per modificare a ritroso i pesi della rete nella misura nella quale hanno contribuito a generare l'errore. In tal modo, addestrando una rete su una quantità sufficiente di dati etichettati, accuratamente scelti e organizzati, si potrà minimizzare la funzione di costo ed avere i giusti pesi per ottenere risultati corretti con un margine di errore accettabile per la maggior parte degli input.
Nel caso dell'apprendimento non supervisionato, invece, la rete neurale viene addestrata solo con dati di input, senza alcuna informazione sugli output relativi. Questo genere di addestramento ha l'obiettivo di identificare attraverso la rete delle strutture o dei pattern nascosti nei dati ed è utile per raggruppare entità simili o identificare anomalie senza ricevere una guida esterna. Esempi di algoritmi basati su questo paradigma includono il clustering, che raggruppa entità simili in base a metriche di distanza o similitudine (es. algoritmo K-Means [3]), gli algoritmi associativi, che individuano

regole descrivendo le relazioni tra elementi nei dati (es. algoritmo Apriori [24]), e le tecniche di riduzione della dimensionalità, che riducono il numero di variabili in un dataset preservando le informazioni essenziali (es. algoritmo PCA, *Principal Component Analysis* [47]).
Infine, l'apprendimento per rinforzo si occupa di problemi di decisioni sequenziali, in cui l'azione da compiere dipende dallo stato attuale del sistema e ne determina lo stato futuro. La qualità di un'azione è determinata dal valore numerico di ricompensa (*reward*) o punizione (*punishment*), che è proporzionata alla qualità dell'azione intrapresa dall'agente nel contesto in cui si trova. Questo metodo ha lo scopo di incoraggiare i comportamenti corretti dell'agente ed è utilizzato spesso nei giochi e nella robotica.

2.3 Algoritmi evolutivi: principi di funzionamento

Gli algoritmi evolutivi [26] sono una famiglia di tecniche stocastiche di ottimizzazione per la risoluzione di problemi ispirate ai meccanismi dell'evoluzione naturale [56]. L'idea di base è che con il prezioso ausilio degli elaboratori elettronici e delle loro capacità di calcolo si può simulare in (relativamente) poco tempo un processo che in natura risulta molto efficace ma impiega migliaia di anni per dare i suoi frutti. Il loro obiettivo principale è trovare una soluzione il più possibile vicina all'ottimo di un problema complesso.

L'evoluzione naturale, secondo la teoria darwiniana, procede senza uno scopo ed è guidata in parte da una componente casuale e in parte dalla legge di sopravvivenza del più adatto. Quindi, così come una popolazione di organismi deve essere adatta all'ambiente che lo circonda per sopravvivere e riprodursi, allora una possibile soluzione deve essere adatta a risolvere il suo problema [56]. Da questo scenario si possono ricavare le analogie utili per modellare un problema risolvibile tramite un algoritmo evolutivo: l'ambiente corrisponde al problema da risolvere, un individuo corrisponde ad una soluzione e una popolazione è formata da individui (ovvero soluzioni) che possono essere più o meno diversi tra loro, e infine l'adattamento indica la qualità di una soluzione nel risolvere il problema. Questa valutazione viene eseguita da una funzione di *fitness*, o funzione obiettivo, appositamente formulata per il problema in questione.

In generale gli algoritmi evolutivi partono da un insieme di soluzioni casuali o generate secondo qualche particolare euristica e procedono in modo iterativo. Ogni iterazione può essere vista come una generazione. Per emulare il passo evolutivo di ogni generazione vengono utilizzati tre operatori fondamentali: la ricombinazione (o *crossover*), la mutazione e la selezione. La ricombinazione consiste nel mescolare il contenuto di due soluzioni per generare una nuova soluzione, la mutazione consiste nell'introdurre una variazione casuale all'interno di una piccola parte di soluzioni e, infine, la selezione consiste nel creare delle repliche delle migliori soluzioni e mantenerle per la successiva generazione. Da questa descrizione si evince la natura "stocastica" degli algoritmi evolutivi, che appunto introducono elementi aleatori e, di conseguenza, un fattore di

incertezza in tutto il processo. In ogni generazione vengono classificate, solitamente in ordine decrescente, le migliori soluzioni sulla base del loro valore di *fitness*.
Nella Figura 2.4 sono mostrati due esempi di ricombinazione e mutazione. Nel caso della ricombinazione, sulla sinistra ci sono le due soluzioni di partenza (codificate in 0 e 1) e sulla destra c'è il risultato della loro ricombinazione, unendo porzioni differenti delle soluzioni di partenza. Mentre nel caso della mutazione, a sinistra si trova la soluzione di partenza e a destra si può osservare la soluzione modificata casualmente in due punti.

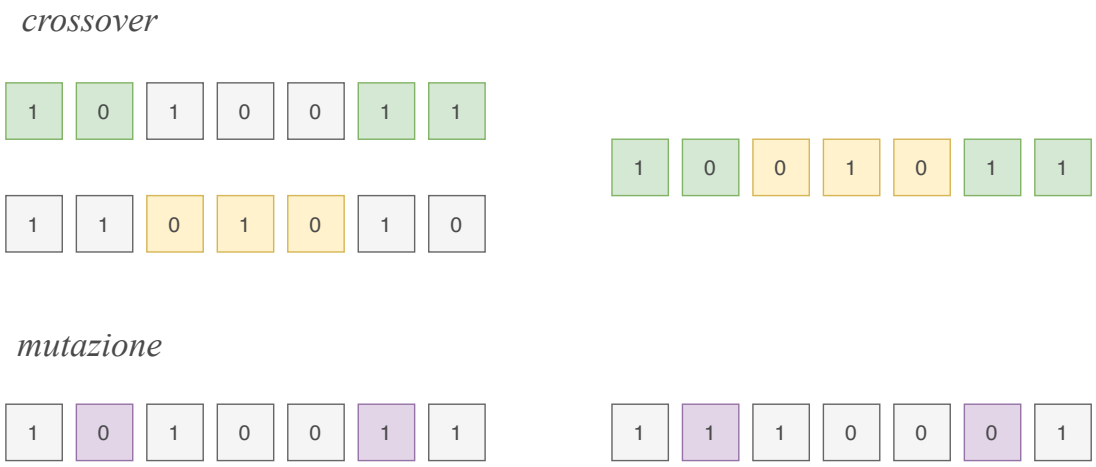


Figura 2.4: *Esempi visuali di crossover e mutazione*

Quando si parla di algoritmi evolutivi si è soliti distinguere tra algoritmi genetici, programmazione genetica, strategie evolutive e programmazione evolutiva.
Un algoritmo genetico [21] è costituito da una popolazione di individui di eguale dimensione, solitamente codificati da stringhe binarie. La programmazione genetica [33], invece, è una tecnica di risoluzione di problemi in cui gli individui che evolvono sono programmi informatici, rappresentati come alberi sintattici in cui i nodi interni sono funzioni e le foglie costituiscono i simboli terminali del programma.
Le strategie evolutive [46, 53], diversamente dagli algoritmi genetici, sono caratterizzati da individui rappresentati da vettori di valori reali con lunghezza fissa e la popolazione iniziale è generata in modo casuale, entro un certo intervallo di valori e utilizzando una distribuzione uniforme. Vengono utilizzati degli operatori particolari di ricombinazione, nello specifico: la ricombinazione discreta, che genera i figli con distribuzione uniforme rispetto ai genitori, la ricombinazione intermedia, che fa la media dei valori dei due genitori, e la ricombinazione casuale, che determina in modo casuale i pesi da attribuire ai genitori per la creazione dell'individuo figlio [17].
Infine, la programmazione evolutiva [16] nasce come approccio all'intelligenza artificiale e prevede che ogni individuo della popolazione costituisca una macchina a stati finiti (o FSM, *Finite State Machine*), la quale ha la caratteristica di ricevere in input

una serie di simboli e restituisce in output una serie di stati, basandosi esclusivamente sugli stati correnti e l'input. L'obiettivo della programmazione evolutiva è predire la prossima configurazione del sistema.

L'applicazione degli algoritmi evolutivi spazia dall'ambito industriale all'intelligenza artificiale. Vengono usati per ottimizzare attività di pianificazione, scegliendo tra diversi modi alternativi di impiegare le risorse quello con il minor costo e le più alte prestazioni, di progettazione, ovvero determinare la disposizione ottimale di elementi entro certi requisiti, di simulazione e di controllo.

2.4 Ottimizzazione multi-obiettivo

Come già detto, gli algoritmi evolutivi hanno lo scopo di ottimizzare una funzione obiettivo, rappresentata dalla funzione di *fitness*. Quando questa funzione è formulata in modo da esprimere un solo obiettivo si parla di ottimizzazione mono-obiettivo. Le cose diventano più complesse quando si vogliono ottimizzare più obiettivi contemporaneamente e, come spesso accade, questi obiettivi sono parzialmente in conflitto tra loro. In questo caso si entra nel campo dell'ottimizzazione multi-obiettivo [10], che ha appunto lo scopo di cercare soluzioni che bilanciano più obiettivi conflittuali [14].

L'ottimizzazione in generale consiste nel minimizzare o massimizzare una funzione obiettivo. Per i problemi di ottimizzazione mono-obiettivo vale il concetto di soluzione ottima, ovvero l'unica soluzione che ottimizza, meglio delle altre, l'obiettivo. Invece nei problemi di ottimizzazione multi-obiettivo nessuna soluzione è ottima in quanto gli obiettivi sono in conflitto. In tal caso si parla di soluzione non-dominata (o di soluzione Pareto-ottima [7]), che si realizza quando non è possibile alcuna nuova soluzione del problema che migliori i risultati di almeno una funzione obiettivo senza diminuire i risultati delle altre. Quindi, nei problemi di ottimizzazione multi-obiettivo bisogna innanzitutto identificare le soluzioni ammissibili non-dominate, ossia la regione Pareto-ottima o il "fronte di Pareto", e successivamente bisogna scegliere una soluzione tra esse. È bene notare che il numero di soluzioni sul fronte di Pareto può essere potenzialmente infinito e vario, pertanto esistono diverse filosofie e approcci che si concentrano sul trovare una soluzione che soddisfi le preferenze soggettive di un decisore umano (o DM, *Decision Maker*).
Nell'esempio mostrato nella Figura 2.5 viene illustrato graficamente un fronte di Pareto (linea rossa) di un problema di ottimizzazione con due obiettivi. Gli assi f_1 e f_2 rappresentano le due funzioni obiettivo, da massimizzare o minimizzare, i quadrati blu rappresentano le soluzioni e, nello specifico, i quadrati blu scuro sulla linea rossa sono le soluzioni Pareto-ottime.

I metodi di ottimizzazione multi-obiettivo possono essere suddivisi in quattro classi. I primi sono i metodi senza preferenza, nei quali viene trovata una soluzione di compromesso neutrale, senza ricevere alcuna informazione da parte del DM sulle preferenze rispetto alla soluzione. Poi ci sono i metodi a priori, in cui le preferenze vengono

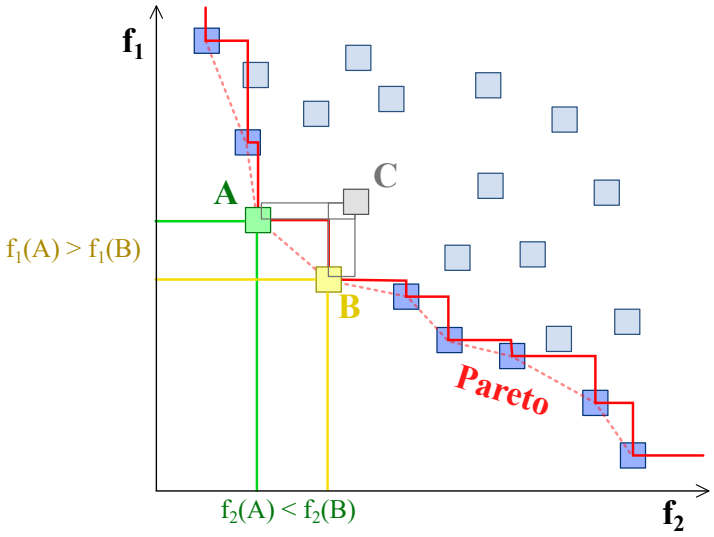


Figura 2.5: *Esempio di fronte di Pareto**

descritte preventivamente dal DM e, solo in seguito, viene implementato il metodo in modo da trovare la soluzione che soddisfa meglio tali preferenze. Con un approccio opposto, ci sono i metodi a posteriori, nei quali viene prima identificato un insieme di soluzioni Pareto-ottimali e dopodiché il DM è portato a scegliere una di esse. Infine ci sono i metodi interattivi, nei quali ad ogni iterazione vengono mostrate al DM le soluzioni Pareto-ottimali ed egli descrive come potrebbero essere migliorate, quindi il decisore "guida" la ricerca della miglior soluzione.

Gli algoritmi evolutivi fanno parte principalmente dei metodi a posteriori, insieme alla programmazione matematica e ai metodi di apprendimento profondo.
I metodi di apprendimento profondo sono un nuovo approccio in questo specifico campo e si basano sull'idea di sfruttare le capacità di generalizzazione delle reti neurali profonde per ottenere le soluzioni sul fronte di Pareto a partire da pochi esempi di un determinato problema, addestrando un modello.
Nella programmazione matematica, invece, un algoritmo viene ripetuto più volte ed ogni sua esecuzione produce una soluzione Pareto-ottimale. Un esempio di metodo di programmazione matematica è la combinazione lineare dei pesi, in cui viene associato ad ogni funzione obiettivo un coefficiente che funge da peso e il problema si riduce ad un problema mono-obiettivo in cui va ottimizzata la somma pesata degli obiettivi, quindi: $\sum_{i=1}^{k} w_i f_i(x)$. L'insieme delle soluzioni non-dominate può essere generato variando i pesi w_i nella funzione obiettivo, per tutti i k obiettivi [35].
Quando si riduce un problema di ottimizzazione multi-obiettivo ad un problema di ottimizzazione mono-obiettivo si parla di scalarizzazione, e il metodo della combinazione

*`https://commons.wikimedia.org/wiki/File:Front_pareto.svg`, autore Johann Dréo (Nojhan), CC BY-SA 3.0 `http://creativecommons.org/licenses/by-sa/3.0/`, via Wikimedia Commons

lineare dei pesi è un esempio di scalarizzazione lineare. Un altro metodo che usa la scalarizzazione è quello dell'ϵ-vincolo, in cui uno degli obiettivi è ottimizzato mentre gli altri sono usati come estremi da rispettare (appunto vincolanti) e sono rappresentati da termini parametrici noti (Formula 2.2). Anche in tal caso la soluzione ottima dipende dai parametri ϵ_i associati ai $k-1$ obiettivi.

$$\begin{aligned} &\text{ottimizzare } f_j(x) \\ &f_i(x) \geq \epsilon_i \text{ (oppure } f_i(x) \leq \epsilon_i) \qquad \forall i \in \{1,\ldots,k\}\backslash\{j\} \end{aligned} \tag{2.2}$$

Alcuni algoritmi di programmazione matematica richiedono una conoscenza del dominio del problema e possono essere sensibili alla forma o alla continuità del fronte di Pareto.
Negli algoritmi evolutivi le soluzioni generate e rappresentate dalla popolazione sono approssimazioni del fronte di Pareto e nessuna delle soluzioni è dominata dalle altre. Possono essere usati quando si ha una scarsa conoscenza del dominio e, in più, sono in grado di fornire una maggiore conoscenza di esso e permettono di capire la struttura delle possibili soluzioni. Di contro gli algoritmi evolutivi impiegano più tempo di esecuzione rispetto agli algoritmi di programmazione matematica e la Pareto-ottimalità delle soluzioni generate non è garantita [35]. Gli algoritmi evolutivi più utilizzati per problemi di ottimizzazione multi-obiettivo sono l'NSGA-II [9] e lo SPEA-2 [31].

Capitolo 3

Metodologia

In questo capitolo verranno illustrate le diverse metriche di valutazione definite per la valutazione dei due obiettivi (locomozione e aggregazione) e verranno spiegate nel dettaglio le proprietà del simulatore e dell'algoritmo evolutivo scelto per questo lavoro di tesi.

3.1 Obiettivi e metriche delle strategie evolutive

Come accennato nel paragrafo 2.3, la funzione di *fitness* guida tutto il processo evolutivo e la sua formulazione è un aspetto cruciale dell'esperimento [38]. Il punto di partenza per la definizione delle metriche è stato fornito dall'articolo *"Enhancing Aggregation in Locomotor Multi-Agent Systems: a Theoretical Framework"* [42], in cui vengono illustrate le funzioni di fitness per la locomozione e l'aggregazione come segue:

$$\text{Fitness} = \frac{1}{N}\sum_{i=1}^{N} P_i + D_i + S_i + J_i \tag{3.1}$$

dove N è il numero di agenti, P_i è la funzione usata per incentivare la locomozione (Formula 3.2), D_i è la funzione usata per incentivare l'aggregazione (Formula 3.3), S_i è una funzione che favorisce il risparmio energetico (Formula 3.4) e, infine, J_i penalizza gli agenti che spingono i loro giunti al limite (Formula 3.5).

$$P_i = \| pos_{curr} - pos_{prev} \| \tag{3.2}$$

$$D_i = e^{-100 \cdot \frac{1}{N-1} \sum_{j=1}^{N-1} | dist_{target} - distij |} \tag{3.3}$$

$$S_i = -0.01 \cdot \frac{1}{N_m} \sum_{j=1}^{N_m} m_j^2 \tag{3.4}$$

$$J_i = -0.1 \cdot N_j \tag{3.5}$$

Le variabili pos_{curr} e pos_{prev} indicano rispettivamente la posizione corrente e la posizione dell'agente allo step precedente; il valore $dist_{target}$ indica la distanza ideale che gli agenti dovrebbero avere tra di loro durante lo stato di aggregazione e $dist_{ij}$ indica la distanza effettiva tra l'agente i e ogni agente $j \neq i$ al momento della valutazione.
La funzione S_i penalizza l'uso eccessivo degli N_m motori, ognuno identificato da m_j, per favorire delle strategie energeticamente efficienti, mentre la funzione J_i penalizza l'uso degli N_j giunti al limite, per favorire strategie più naturali.

Dai risultati dell'articolo [42] si nota come tutti gli agenti abbiano correttamente sviluppato la locomozione, mentre l'aggregazione del gruppo non risulta particolarmente performante. Con molta probabilità, dunque, il problema della Formula 3.1 è la componente D_i, la quale viene dominata dai valori, più alti, della funzione P_i.
Così come la funzione relativa alla locomozione produce dei valori tanto più alti quanto più è alta la distanza percorsa dagli agenti in un episodio, allo stesso modo la funzione relativa all'aggregazione dovrebbe produrre dei valori tanto più alti quanto più la distanza tra le coppie di agenti è vicina alla distanza target, ovvero la distanza ideale. La D_i (Formula 3.3), invece, nel caso ideale restituisce semplicemente il valore 1 (Figura 3.1). Quindi, fatta questa considerazione, abbiamo formulato due nuove possibili funzioni da testare per sostituire la D_i:

$$D\_Fratta_i = \left(\epsilon + \frac{1}{N-1} \sum_{j=1}^{N-1} | \, dist_{target} - dist_{ij} \, | \right)^{-1} \tag{3.6}$$

$$D\_Gaussiana_i = max \cdot e^{-\frac{\left(\frac{1}{N-1} \Sigma_{j=1}^{N-1} | \, dist_{target} - dist_{ij} \, | \right)^2}{\sigma^2}} \tag{3.7}$$

Per quanto riguarda la funzione "fratta", si può vedere chiaramente dal grafico di confronto in Figura 3.1 che nei pressi della distanza ideale $dist_{target}$ il valore prodotto da questa funzione si avvicina a $\frac{1}{\epsilon}$, dove $\epsilon > 0$ è molto piccolo, in modo da evitare che il denominatore possa mai essere pari a 0, ed è scelto arbitrariamente. Nella funzione D_Fratta usata per produrre il grafico in figura $\epsilon = 0.1$. La funzione "gaussiana", invece, ha una crescita più graduale e meno improvvisa della precedente ed è determinata dai valori max, che indica il punto massimo raggiunto dalla curva nel momento in cui si raggiunge la distanza target, e σ che indica la deviazione standard, ovvero l'ampiezza della curva. Nell'esempio in Figura 3.1 $max = 7$ e $\sigma = 8$.

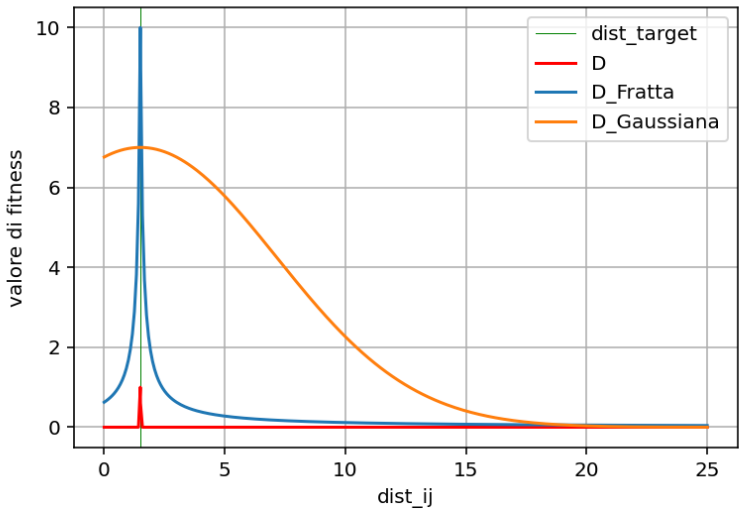


Figura 3.1: *Confronto tra le fitness D, D_Fratta e D_Gaussiana per valori arbitrari di* $dist_{ij}$

Calcolo sequenziale e incrementale della fitness

La funzione obiettivo (Formula 3.1) è *sommativa* in quanto le singole parti di cui è composta sono sommate tra loro. Altri due approcci che abbiamo provato, in alternativa a questo, sono il calcolo *sequenziale* e il calcolo *incrementale* delle componenti. Ciò che ci ha spinti a tentare nuovi approcci è il fatto che, eseguendo delle simulazioni usando la sola funzione di fitness legata all'aggregazione, gli agenti sviluppavano la strategia di rimanere fermi sul posto, in piedi in equilibrio (paragrafo 4.3). Probabilmente il comportamento di aggregazione ha bisogno della locomozione come abilità pre-esistente, quindi abbiamo pensato di far sviluppare prima la locomozione e poi l'aggregazione. Questa è l'idea dell'approccio sequenziale, in cui nella prima metà di simulazione viene usata la Formula 3.1 senza la componente D, mentre nella seconda parte viene usata la stessa funzione, ma stavolta senza la componente P. Con il calcolo incrementale, invece, la prima metà rimane esclusiva della locomozione, ugualmente al calcolo sequenziale, mentre nella seconda metà viene aggiunta alla locomozione anche l'aggregazione, ritornando praticamente alla formula della funzione sommativa originale.
L'implementazione e i risultati di entrambi questi approcci sono descritti con maggiore dettaglio nei paragrafi 4.5 e 4.8.

3.2 Modelli di simulazione: Evorobotpy3

Per condurre gli esperimenti di questo lavoro è stato usato un software open-source chiamato Evorobotpy3, sviluppato da Paolo Pagliuca, Stefano Nolfi e Alessandra Vi-

tanza [45]. Come è intuibile dal nome, è scritto in Python, esclusa la sezione che si occupa della rete neurale che è scritta in C++ perché, essendo computazionalmente critica, era necessario adottare un linguaggio in grado di generare un codice oggetto più veloce. Questo software CLI (*Command Line Interface*) permette di simulare l'evoluzione di robot in ambienti configurabili ad-hoc e consente di utilizzare sia algoritmi evolutivi che algoritmi di apprendimento per rinforzo.

3.2.1 Architettura del software

Il software Evorobotpy3 è costituito da tre moduli principali interconnessi tra loro: il modulo *environment*, il modulo *algorithm* e il modulo *policy*, come mostrato nella Figura 3.2. Il modulo *environment* si occupa della definizione dell'ambiente, ovvero il problema da risolvere e le sue proprietà; il modulo *algorithm* si occupa di fornire un'interfaccia per l'implementazione dei diversi algoritmi di tipo evolutivo e/o di apprendimento per rinforzo; il modulo *policy* si occupa della gestione relativa alla creazione delle reti neurali e ai calcoli da eseguire al loro interno, grazie all'ausilio del modulo *net*. Nella Figura 3.2 sono sintetizzate le interfacce dei moduli con i nomi di alcune delle loro classi implementative.

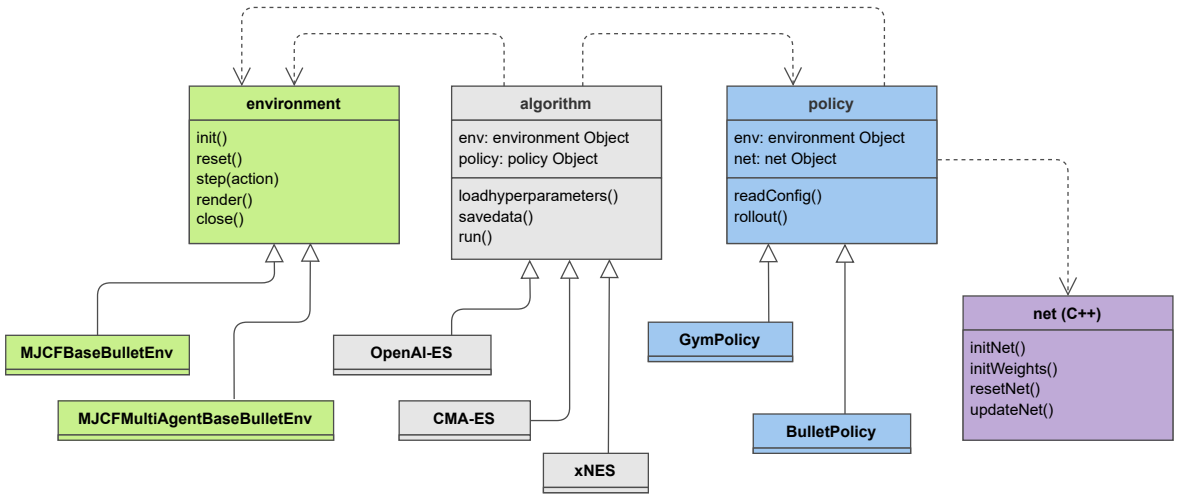


Figura 3.2: *Diagramma dell'architettura generale di Evorobotpy3**

Gymnasium

Il modulo *environment* fa uso della libreria Gymnasium [58], la quale offre un'architettura per lo sviluppo e il confronto di algoritmi di apprendimento per rinforzo ed è un fork della libreria OpenAI-Gym.
Gymnasium contiene una serie di ambienti di default classici (Cart-Pole, Pendulum,

*Fonte immagine: [45]

Acrobot, ecc...) e fornisce un'interfaccia per la creazione di ambienti personalizzati. La possibilità di creare ambienti personalizzati ha permesso una vasta diffusione di ambienti open-source, più e meno complessi, compatibili con essa. Ad esempio, la libreria PyBullet [8] integrata in questo software, oltre a implementare un sistema di simulazione fisica in 3D, include degli ambienti compatibili con Gymnasium, in particolare i locomotori, ovvero ambienti contenenti robot di varie forme (umanoide, quadrupede, insetto), mostrati nella Figura 3.3, che possono essere allenati per task come il nuoto, il salto e la camminata [40] e di cui è stato fatto uso in questo lavoro di tesi. La struttura e le caratteristiche fisiche dei robot (ad es., dimensioni degli elementi che compongono il corpo, connessioni tra i vari elementi, variazioni angolari dei giunti) sono definite in un file XML (*eXtensible Markup Language*) che viene caricato al momento in cui l'ambiente viene creato (Algoritmo 1, riga 2).

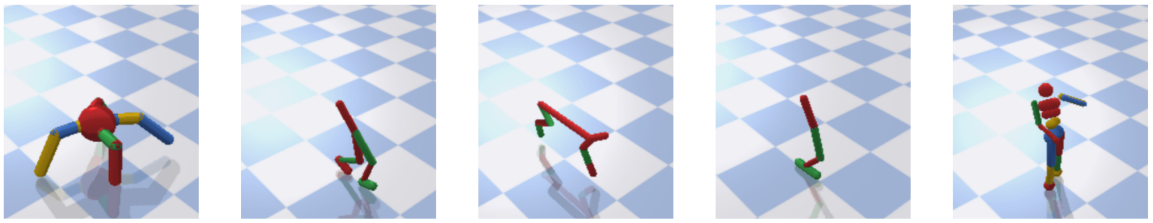


Figura 3.3: *Esempi di locomotori standard in PyBullet**

Algoritmo 1 Esempio di utilizzo delle funzioni della libreria Gymnasium

```
1: import gymnasium as gym
2: env = gym.make("EnvironmentName")
3: env.reset()
4: for _ in range(n) do
5:     env.render()
6:     observation_space, reward, terminated, truncated, info = env.step(action_space)
7: end for
8: env.close()
```

Le funzioni fondamentali di questa libreria sono mostrate nell'Algoritmo 1. Dopo l'*import* della libreria, alla riga 2 viene invocata la funzione *make* per generare un'istanza di un ambiente, passando come argomento il nome della classe che lo implementa nel codice. La funzione *reset* alla riga 3 serve a portare l'ambiente e gli agenti al suo interno allo stato di partenza, per poter successivamente avviare un nuovo episodio da zero. Il ciclo *for* alle righe 4-7 rappresenta gli n "momenti discreti" di un singolo episodio di valutazione. La funzione *step* alla riga 6 è la più importante: riceve come argomento un vettore di azione (*action_space*), che contiene gli input da fornire alla rete dell'agente sotto esame, il quale, basandosi su di essi e sui parametri della sua rete

*Fonte immagine: [4]

(che sono noti al di fuori di questa procedura), restituisce un vettore di osservazione (*observation_space*), che contiene i valori di output della rete in seguito alle computazioni appena eseguite, e un *reward*, che rappresenta la ricompensa calcolata dalla funzione di fitness utilizzata nell'ambiente specifico. Oltre a questi valori, la funzione *step* restituisce anche altri valori di carattere informativo come *terminated*, che indica se l'episodio è finito oppure no, *truncated*, che indica se l'episodio è terminato in modo prematuro causa condizioni o errori particolari, e *info*, che può essere utilizzato per memorizzare informazioni di cui si vuole tenere traccia. Le funzioni *render* e *close*, rispettivamente alle righe 5 e 8, servono a mostrare graficamente l'ambiente e a chiudere il rendering grafico alla fine degli n steps dell'episodio.

OpenAI-ES

Il software Evorobotpy3 include l'implementazione di diversi algoritmi di apprendimento. In questo lavoro abbiamo utilizzato l'algoritmo OpenAI-ES[1] (dove ES sta per *Evolutionary Strategy*) [51], ideato da OpenAI per fornire una valida alternativa all'apprendimento per rinforzo in problemi di ottimizzazione. Questo algoritmo offre diversi vantaggi computazionali, come la relativa semplicità di implementazione, la possibilità di esecuzione sfruttando il calcolo parallelo e la riduzione di circa $\frac{2}{3}$ della computazione per episodio a causa dell'assenza della *backpropagation* [51]. In alcuni problemi tipicamente affrontati con l'apprendimento per rinforzo, come Atari e MuJoCo [57], si è dimostrato essere una valida alternativa [51] e anche confrontandolo con altri algoritmi evolutivi, come CMA-ES [23] e xNES [59], ha ottenuto delle prestazioni più che soddisfacenti e dei risultati qualitativamente migliori [43, 44].

Questo algoritmo ha lo scopo di trovare i migliori parametri di una determinata *policy* per avvicinarsi il più possibile all'obiettivo prefissato. La *policy* rappresenta le regole e i parametri che permettono agli agenti di prendere decisioni per il raggiungimento di un obiettivo definito. Nel nostro lavoro la *policy* è una rete neurale e i suoi parametri sono i pesi e i bias della rete. Come accennato nel paragrafo 2.3, negli algoritmi evolutivi si parla metaforicamente di popolazioni di individui. In questo contesto un individuo, anche chiamato genotipo, è rappresentato da un vettore contenente tutti i parametri della rete neurale che lo caratterizzano, ovvero valori numerici reali. Essendo OpenAI-ES un algoritmo di ottimizzazione, funziona seguendo il paradigma di "ipotesi e controllo": partendo da una inizializzazione casuale di parametri, questi vengono leggermente perturbati e successivamente valutati per capire quale perturbazione ha spostato il vettore di partenza verso un risultato migliore.
Nello specifico, è mostrato a titolo di esempio lo pseudocodice (Algoritmo 2). Alle righe 1-6 vengono dichiarati gli iperparametri e alle righe 8-19 viene effettivamente espresso l'algoritmo che ne fa uso.
Gli iperparametri di un modello sono particolari proprietà da dichiarare prima di avviare l'algoritmo e ne influenzano il risultato. Gli algoritmi di apprendimento per rinforzo

[1] https://openai.com/index/evolution-strategies/

hanno bisogno generalmente di più iperparametri rispetto agli algoritmi evolutivi e leggère variazioni in essi possono cambiare in maniera significativa l'esito degli esperimenti, mentre in ES è stata testata una buona robustezza rispetto al loro settaggio, e ciò significa che, per alcuni problemi, configurazioni differenti degli iperparametri non cambiano radicalmente l'esito degli esperimenti. Nello pseudocodice (Algoritmo 2) i valori degli iperparametri coincidono con i valori utilizzati nell'implementazione effettiva dell'algoritmo. Nello specifico i valori di σ e di α sono valori che euristicamente funzionano bene nella maggior parte dei problemi, mentre il valore λ ideale è solitamente 250 ma per problemi non particolarmente difficili può essere ridotto a 20 [40].

Algoritmo 2 Pseudocodice di OpenAI-ES con ottimizzatore Adam

1: n_{gen} = millions (numero di generazioni)
2: $\lambda = 20$ (numero di perturbazioni)
3: $\sigma = 0.02$ (deviazione standard del rumore gaussiano)
4: $\alpha = 0.01$ (tasso di apprendimento)
5: $F(\cdot)$ (funzione obiettivo, o *fitness*)
6: θ (vettore di parametri della rete)
7:
8: inizializzazione di θ_1
9: **for** $g = 1, 2, \ldots, n_{gen}$ **do**
10: **for** $i = 1, 2, \ldots, \lambda$ **do**
11: $\epsilon_i \leftarrow \mathcal{N}(0, 1)$
12: $f_i^+ \leftarrow F(\theta_i + \sigma\, \epsilon_i)$
13: $f_i^- \leftarrow F(\theta_i - \sigma\, \epsilon_i)$
14: $u_i = ranks(f_i^+ - f_i^-)$
15: normalizzazione: $u_i \in [-0.5, +0.5]$
16: **end for**
17: stima del gradiente: $\tilde{\nabla} = \frac{1}{\lambda\sigma} \sum_{i=1}^{\lambda} (u_i\, \epsilon_i)$
18: $\theta_{g+1} \leftarrow \theta_g + \alpha \cdot \text{Adam}(\tilde{\nabla})$
19: **end for**

Partendo da un vettore θ_1 contenente un'inizializzazione casuale (o secondo determinate euristiche) dei parametri della rete neurale (righe 6 e 8), viene generato un vettore ϵ_i con dei valori estrapolati da una distribuzione gaussiana (riga 11). Successivamente vengono moltiplicati i valori di ϵ_i per σ e poi vengono aggiunti e sottratti a θ_i. Il valore σ indica la varianza da applicare alla distribuzione gaussiana di perturbazioni casuali. Un valore troppo basso di σ darà luogo a vettori molto vicini a θ, rallentando l'apprendimento, mentre un valore troppo alto di σ genererà vettori lontani da θ e il processo potrebbe non convergere mai ad una buona soluzione. In seguito a questo calcolo si otterranno due nuovi vettori (righe 12-13), con perturbazioni opposte tra loro, da usare come argomento della funzione obiettivo per calcolarne le prestazioni f_i^+ e f_i^-. Il calcolo di due vettori speculari tra loro migliora la precisione della stima della

fitness [40], mantenendo equilibrato il ventaglio di soluzioni testate intorno a θ.
Fatte queste operazioni, viene eseguito il ranking della differenza delle due fitness simmetriche e successivamente questo valore viene normalizzato (righe 14-15), nel caso dello pseudocodice di esempio nell'intervallo $[-0.5, +0.5]$. Questa operazione è anche chiamata *fitness shaping* [51] e ha diversi vantaggi: innanzitutto la normalizzazione tramite ranking riduce l'impatto degli individui con fitness estremamente alta o bassa (*outlier*), che potrebbero distorcere la stima del gradiente e rendere l'addestramento meno stabile; poi, questa normalizzazione rende l'algoritmo invariante rispetto alla scala dei valori di fitness provenienti da funzioni obiettivo differenti, appiattendo in ogni caso i valori ottenuti sulla stessa scala; infine, la normalizzazione tramite ranking aiuta a prevenire che l'algoritmo rimanga intrappolato in *local optima*, ottimi locali, ovvero soluzioni sotto-ottimali raggiunte prematuramente durante l'addestramento per la prevaricazione di un piccolo numero di fitness molto alte che dominano il processo di selezione. Le due tecniche di normalizzazione più utilizzate sono la normalizzazione Min-Max (Formula 3.8) e la normalizzazione Z-Score (Formula 3.9): nella prima vengono usati il valore massimo e il valore minimo dell'insieme dei valori da normalizzare, mentre nella seconda vengono usati la loro media μ e varianza σ.

$$\text{Min-Max} = \frac{x - \text{min\_score}}{\text{max\_score} - \text{min\_score}} \tag{3.8}$$

$$z = \frac{x - \mu}{\sigma} \tag{3.9}$$

Alla riga 17 viene effettuata una stima del gradiente. Intuitivamente, il vettore gradiente di una funzione indica la direzione di massima crescita e il suo modulo ne indica l'intensità. Per ottenere una stima del gradiente, e conoscere dunque la direzione verso cui muovere l'intero processo evolutivo, viene calcolata la somma dei vettori perturbati ϵ_i pesata con i valori normalizzati rappresentati da u_i, questa somma viene poi normalizzata usando un fattore pari a $\frac{1}{\sigma}$, per evitare che la scelta dell'iperparametro σ influenzi eccessivamente la stima, e viene fatta la media complessiva di tutti i valori appena calcolati moltiplicandoli per $\frac{1}{\lambda}$. Il gradiente stimato indica la direzione media che migliora il risultato della funzione obiettivo.
Per terminare, viene effettuato l'aggiornamento del vettore dei parametri della rete (riga 18), aggiungendo a questo il gradiente sottoposto all'ottimizzatore Adam (*Adaptive Moment Estimation*) [32], ponderato dal tasso di apprendimento α. Il tasso di apprendimento indica di quanto i parametri del modello vengono aggiornati ad ogni iterazione, quindi, di fatto, la velocità del processo di apprendimento. Con l'uso di un valore troppo alto di α si rischia di saltare il valore ottimale della funzione obiettivo, mentre con un valore troppo basso si rischia di avere un apprendimento troppo lungo e, di conseguenza, una convergenza molto lenta. Per trovare il valore giusto del tasso di apprendimento vengono impiegate strategie più articolate, come ad esempio l'uso dell'ottimizzatore stocastico Adam, utilizzato nell'implementazione di OpenAI-ES in

questo lavoro di tesi. Questo ottimizzatore ha la caratteristica di evitare che i parametri della rete subiscano aggiornamenti troppo forti o troppo deboli e adatta il *learning rate* per ogni parametro in base agli aggiornamenti precedenti.
Terminato ciò, viene ripetuto il procedimento per un numero prefissato (n_{gen}) di iterazioni, ripartendo dal vettore di parametri θ_{g+1} appena aggiornato.

Questo è lo scheletro dell'algoritmo, ma la sua implementazione nel software Evorobotpy3 presenta delle peculiarità. Ossia, in seguito alla valutazione dei singoli individui durante le generazioni, il vettore contenente i valori calcolati dalla funzione di fitness è ordinato in ordine crescente e gli individui migliori sono soggetti ad una o più post-valutazioni che hanno lo scopo di confermarne la qualità e la capacità di generalizzazione. Questi saranno gli individui soggetti ai test qualitativi una volta terminata l'esecuzione dell'algoritmo. Inoltre, tra il calcolo del gradiente stimato e l'ottimizzazione attraverso Adam, può essere impostato il calcolo del *weight decay* [22], una tecnica ampiamente diffusa nel machine learning utile per penalizzare i valori alti nei pesi di una rete, portando ad una serie di benefici quali la riduzione dell'*overfitting* e il miglioramento della stabilità del modello.

Policy

Il modulo *policy* consente l'utilizzo di diverse architetture di reti neurali, le quali gestiscono il sistema decisionale degli agenti durante l'evoluzione e sono fondamentali per la valutazione delle strategie evolutive eseguita tramite la funzione *rollout*. Tra le architetture disponibili ci sono le reti *feed-forward*, le RNN e le LSTM (*Long-Short Term Memory*) [25]. Per ognuna di queste possono essere configurati diversi iperparametri, tra cui quelli relativi al numero di strati della rete, al numero di neuroni interni e al tipo di funzioni di attivazione dei neuroni interni e dei neuroni di output.
Questo modulo fa uso del modulo *net* che implementa gli algoritmi e le strutture dati per manipolare le reti neurali. I file, scritti in C++, vengono compilati e convertiti in codice Python attraverso l'uso del superset Cython[2], in modo da garantire le prestazioni ottimizzate fornite dal codice in C++.

3.2.2 Iperparametri

Gli iperparametri necessari ad avviare le simulazioni e a testarne le diverse configurazioni vanno specificati in un file con estensione *.ini* appositamente creato e strutturato in tre sezioni, come mostrato nella Figura 3.4: sotto *EXP* vanno specificate le informazioni relative all'ambiente e all'algoritmo, sotto *ALGO* vanno specificati gli iperparametri necessari per eseguire l'algoritmo scelto e sotto *POLICY* vanno specificate le proprietà della rete neurale, della funzione obiettivo e degli episodi.
Di seguito verranno spiegati brevemente tutti gli iperparametri ed elencati sinteti-

[2]https://cython.org/

camente nella Tabella 3.1. Una spiegazione dettagliata della maggior parte degli iperparametri si può trovare in [40].

```
[EXP]                                   pepisodes = 3
environment = AntSwarmBulletEnv-v0      maxsteps = 1000
algo = OpenAI-ES                        nhiddens = 20
                                        nlayers = 1
[ALGO]                                  bias = 1
maxmsteps = 50                          out_type = 3
stepsize = 0.01                         architecture = 0
noiseStdDev = 0.02                      afunction = 2
sampleSize = 20                         winit = 1
saveeach = 1                            action_noise = 1
wdecay = 1                              normalize = 1
                                        clip = 1
[POLICY]                                fit_id = 2
nrobots = 5                             seq = 0
heterogeneous = 0                       sum_comp = 1
episodes = 1
```

Figura 3.4: *Esempio di file di configurazione (da colonna sinistra a colonna destra)*

Per quanto concerne la sezione *ALGO*, gli iperparametri dipendono fortemente dal tipo di algoritmo utilizzato, che in questo caso è OpenAI-ES. Con riferimento all'Algoritmo 2, i valori di n_{gen}, λ, α e σ sono rappresentati rispettivamente dagli iperparametri *maxmsteps*, che indica il numero di milioni di step da raggiungere prima di terminare il processo evolutivo, *sampleSize*, *stepsize* e *noiseStdDev*. Il valore *wdecay* è un booleano che indica se applicare o meno il *weight decay*, accennato alla fine del paragrafo su OpenAI-ES. Infine, il valore *saveeach* è un valore che indica la periodicità del salvataggio delle informazioni relative all'andamento del processo evolutivo in uno specifico file.

La sezione *POLICY* è la più ricca delle tre. Il valore *nrobots* indica il numero di agenti da inserire nell'ambiente. Il valore *heterogeneous* è un booleano che indica se evolvere i parametri della rete in modi differenti per agenti diversi oppure se evolvere gli stessi parametri per tutti gli agenti. I valori *episodes* e *pepisodes* indicano rispettivamente il numero di episodi di valutazione e post-valutazione, mentre *maxsteps* indica il numero massimo di step di un episodio, quindi la sua durata massima (il valore n nell'Algoritmo 1). I valori *nhiddens* e *nlayers* indicano rispettivamente la quantità di neuroni nascosti e di strati nascosti della rete, mentre i valori *afunction* e *out_type* indicano un valore che identifica, nel codice, una specifica funzione di attivazione, rispettivamente per i neuroni interni e per i neuroni di output. Nell'esempio in Figura 3.4 è settata come *afunction* la funzione *tanh* e come *out_type* è settata la funzione lineare. Il valore *architecture* indica un numero identificativo del tipo di rete neurale da utilizzare, nell'esempio la rete *feed-forward*. Il valore *winit* indica un valore che identifica la metodologia per inizializzare i pesi della rete. Nel caso dell'esempio è usato il metodo *Xavier* [20], particolarmente efficace in combinazione con la funzione di attivazione *tanh*, che consiste nell'utilizzare una distribuzione normale nella forma (3.10) per assegnare i valori iniziali ai pesi della rete. Il valore *action_noise* è un boo-

leano che indica se aggiungere del rumore ai neuroni motori e può essere utilizzato per incrementare la robustezza delle soluzioni evolutive [40]. Infine, valore *normalize* è un booleano che indica se usare o meno la *virtual batch normalization* [27] che permette, in alcuni casi, di migliorare l'esplorazione degli ambienti, incoraggiando una maggiore varietà di azioni [51].

$$w \sim \mathcal{N}\left(0, \sqrt{\frac{2}{n_{input} + n_{output}}}\right) \tag{3.10}$$

Gli ultimi tre valori della sezione *POLICY*, mostrati nell'esempio in Figura 3.4, riguardano la fitness. Il valore *fit_id* è un numero che identifica il tipo di funzione di fitness da usare per l'aggregazione, il valore *seq* indica se usare un calcolo sommativo, sequenziale o incrementale (accennati nel paragrafo 3.1) e infine il valore *sumComp* è un booleano che indica se sommare o meno le componenti ed è stato usato per testare la funzione di fitness (Formula 3.1) senza la componente P della locomozione (esperimento nel paragrafo 4.3).

3.2.3 Output generati

Il simulatore, durante l'esecuzione di ogni esperimento, genera e aggiorna diversi file grazie ai quali è possibile, poi, analizzare quantitativamente e qualitativamente i risultati ottenuti dalla simulazione. Di seguito sono descritti i file più significativi, utilizzati nelle analisi degli esperimenti.
Il file di testo *bestFitS*.txt* contiene due valori numerici che rappresentano i migliori risultati delle fitness ottenuti rispettivamente negli episodi di valutazione e di post-valutazione durante tutto il processo evolutivo. Il file *bestgS*.npy* contiene l'array dei parametri del miglior individuo in assoluto negli episodi di post-valutazione. L'estensione *.npy* indica che il file è stato salvato con la libreria Python *numpy*[3], usata per gestire strutture numeriche complesse come vettori e matrici, la quale consente poi di caricarlo nuovamente nel codice attraverso un'apposita funzione. Infine, il file *statsS*.npy* è una matrice di 6 colonne che salva periodicamente e progressivamente delle righe con informazioni relative al numero di step eseguiti fino a quel momento, le migliori fitness negli episodi di valutazione e post-valutazione ottenute fino a quel momento, la migliore fitness ottenuta nella singola generazione (ovvero il vettore $(\theta_i \pm \sigma\, \epsilon_i)$ che ha ottenuto la migliore valutazione), la fitness media dei vettori $(\theta_i \pm \sigma\, \epsilon_i)$ della generazione e il valore medio dei pesi del vettore θ_i. Il nome di ognuno di questi file è contrassegnato dal suffisso *S**, in cui *** è un intero che rappresenta il *seed* della simulazione specificato da riga di comando con il parametro *-s* prima di avviarla. Grazie al seed è possibile mandare in esecuzione più simulazioni contemporaneamente su una stessa configurazione di un problema, differenziando i file generati e differenziando anche la serie di numeri pseudocasuali generati durante il processo evolutivo.

[3] https://numpy.org/

Nome	**Descrizione**
environment	nome dell'ambiente
algo	nome dell'algoritmo
maxmsteps	numero di milioni di step da raggiungere prima di terminare l'algoritmo (n_{gen})
sampleSize	dimensione del numero di individui da generare (λ)
stepsize	tasso di apprendimento (α)
noiseStdDev	deviazione standard del rumore gaussiano da applicare agli individui (σ)
wdecay	weight decay
saveeach	frequenza di salvataggio dei dati
nrobots	numero di agenti nell'ambiente
heterogeneous	parametri eterogenei o omogenei per tutti gli agenti
episodes	numero di episodi di valutazione
pepisodes	numero di episodi di post-valutazione
maxsteps	numero massimo di steps di un episodio
nhiddens	numero di neuroni interni della rete
nlayers	numero di strati interni della rete
afunction	funzione di attivazione dei neuroni interni
out_type	funzione di attivazione dei neuroni di output
architecture	tipo di rete neurale
winit	metodologia per inizializzare i pesi della rete
action_noise	aggiunta di rumore in input
normalize	aggiunta della virtual batch normalization
fit_id	tipo di fitness di aggregazione da usare
seq	modalità di calcolo della fitness
sumComp	attivare la somma delle componenti della fitness

Tabella 3.1: *Elenco degli iperparametri usati da Evorobotpy3, divisi nelle sezioni EXP, ALGO e POLICY*

Capitolo 4

Esperimenti

Per il raggiungimento dell'obiettivo di questo lavoro di tesi si è esteso il problema mono-agente AntBullet della libreria PyBullet (Figura 4.1) e utilizzato un approccio filogenetico con l'algoritmo OpenAI-ES. In questo capitolo verranno mostrati tutti i diversi esperimenti svolti, ciascuno dei quali ha fornito indicazioni sulle strade che valeva la pena percorrere e le strade che era ragionevole escludere. Verranno illustrati gli strumenti utilizzati per condurre gli esperimenti, le condizioni comuni agli episodi e verranno descritti nel dettaglio, in ordine cronologico, i risultati più significativi ottenuti durante questa ricerca.

4.1 Setup sperimentale

Prima di iniziare la trattazione degli esperimenti è necessario introdurre i mezzi utilizzati per eseguire le simulazioni e descrivere le proprietà fondamentali del problema.

4.1.1 Dispositivi e tempi di esecuzione

La durata di una singola simulazione può variare in base al problema, all'algoritmo utilizzato e ad alcuni iperparametri relativi, oltre che alle specifiche tecniche della macchina su cui viene eseguita. Ad esempio, l'iperparametro *maxmsteps* regola il numero di iterazioni da dedicare al processo evolutivo e più è alto più la durata di una simulazione aumenta. Un altro fattore significativo è dato dall'architettura della rete neurale utilizzata come policy: una RNN esegue al suo interno più operazioni di una normale rete feed-forward (per via delle sue connessioni ricorsive) e una LSTM è ancora più complessa sia a livello architetturale che a livello computazionale. Per ognuna di queste reti, poi, è determinante anche il numero di neuroni di cui si compone in quanto un numero alto di neuroni in una rete neurale aumenta il numero di computazioni per ogni step e quindi aumenta la durata complessiva del processo evolutivo. Inoltre, data la natura stocastica degli algoritmi evolutivi, per ogni problema ed esperimento

specifico vanno eseguite più simulazioni per cercare di cogliere l'andamento medio delle strategie evolutive sviluppate e per trarre delle conclusioni che siano statisticamente rilevanti e interessanti.
Per tutte queste ragioni le simulazioni relative agli esperimenti mostrati nei prossimi capitoli sono state eseguite su un *cluster* del CNR (*Consiglio Nazionale delle Ricerche*). Un cluster è un insieme di computer interconnessi tramite una rete, configurati per operare come un'unica unità di calcolo e la sua potenza di elaborazione complessiva è data dalla somma delle risorse dei nodi coinvolti, consentendo un'elevata capacità di parallelizzazione dei processi. Per l'accesso al cluster si è usato un certificato per il collegamento da remoto tramite VPN alla rete ISTC (*Istituto di Scienze e Tecnologie della Cognizione*) usando delle credenziali personali per accedere ad un account creato appositamente per questo lavoro di tesi. Il cluster possiede, lato hardware, 4 Intel Xeon-Gold 6252N (2.3GHz/24-core/150W), per un totale di 96 core e dunque 96 processi eseguibili in parallelo, e 128 GB di RAM e, lato software, un sistema operativo Ubuntu 20.04. Grazie a queste specifiche tecniche è stato possibile avviare da remoto, tramite collegamento SSH, più simulazioni in contemporanea ottimizzando di molto i tempi, che altrimenti sarebbero stati impraticabili.
Una volta terminate le varie simulazioni sul cluster, si è utilizzata localmente una macchina virtuale con Ubuntu 24.04 per scaricare e visualizzare graficamente i risultati, tramite il software Evorobotpy3, e analizzare i dati attraverso valutazioni sia qualitative che quantitative.

4.1.2 Condizioni di partenza degli esperimenti

Di seguito vengono riportate le caratteristiche iniziali degli agenti in ogni episodio, che rimarranno immutate per tutti i vari esperimenti descritti nei capitoli successivi.

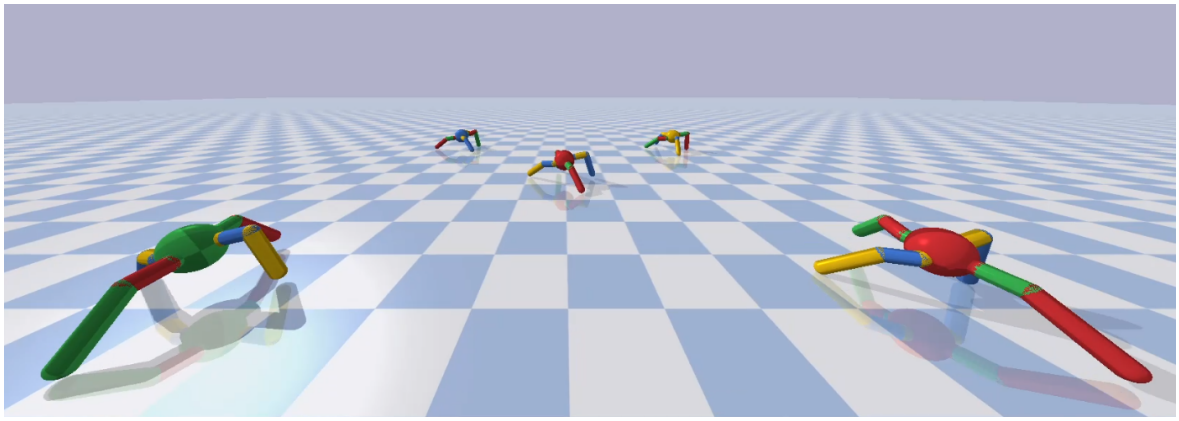


Figura 4.1: *Posizione degli agenti all'inizio di ogni episodio*

Il setup iniziale degli agenti è stato ereditato direttamente dall'articolo [42]. Gli agenti in ogni episodio sono 5 e sono disposti a croce, come mostrato in Figura 4.1. La distanza tra l'agente centrale e tutti gli altri è la stessa ed è circa 5.5 metri, mentre gli

agenti sul "lato" del quadrato sono distanti circa 8 metri e gli agenti agli angoli opposti sono distanti circa 11 metri. La distanza target per il raggiungimento dell'aggregazione, citata nelle Formule (3.3), (3.6) e (3.7), è stata fissata a 1.5 metri.
La durata di ogni episodio è di massimo 1000 steps e gli episodi di post-valutazione a cui sono sottoposti gli individui migliori di ogni generazione sono 3, ognuno da massimo 1000 steps. Questi episodi terminano prematuramente nel momento in cui il "torso" (parte sferica del robot) di ogni agente tocca terra, quindi l'agente viene considerato caduto. Nel paragrafo 4.6 è illustrato il tentativo di rimuovere questo vincolo per verificare se costituisse un limite per lo sviluppo di strategie ottimali.
Infine, vengono usate reti neurali omogenee, quindi durante il processo vengono evoluti i parametri di una sola rete neurale, utilizzata da tutti gli agenti in ogni episodio.
Nella Tabella 4.1 sono riassunti i valori degli iperparametri principali costanti per tutti gli esperimenti descritti in questo capitolo, divisi nei gruppi indicati.

Gruppo	**Parametro**	**Valore**
algoritmo	deviazione standard del rumore gaussiano (σ)	0.02
	tasso di apprendimento (α)	0.01
	# di perturbazioni (λ)	20
policy	reti neurali eterogenee	no
	# di strati interni della rete	1
	funzione di attivazione dei neuroni interni	*tanh*
	funzione di attivazione dei neuroni di output	lineare
ambiente	# di steps per episodio	1000
	# di episodi di post-valutazione	3
	# di agenti in ogni episodio	5
	vincolo della caduta	si

Tabella 4.1: *Valori degli iperparametri costanti per tutti gli esperimenti*

Per quanto riguarda la struttura della rete neurale, invece, nei primi esperimenti è stata utilizzata la stessa descritta in [42], relativa all'ambiente *AntSwarmBulletEnv*, e sintetizzata nella Tabella 4.2.

Eccetto per l'Esperimento 1, descritto nel paragrafo 4.2, per ogni esperimento specifico sono state eseguite 10 simulazioni con seed differente (da 1 a 10), per vedere i risultati medi delle strategie evolutive in ogni specifico scenario.

Id input	Descrizione
0	differenza tra l'altezza del "torso" centrale dell'agente e l'altezza di partenza
1-2	angolo tra il "torso" dell'agente e il punto target (impostato idealmente lontano)
3-5	velocità degli agenti sui tre assi
6-7	*roll* e *pitch*, ovvero le inclinazioni avanti-dietro e destra-sinistra dell'agente
8-23	posizione e velocità dei giunti, che sono 8 per agente
24-27	flag 1 se la "zampa" dell'agente tocca terra; le "zampe" sono 4 per agente
Id output	**Descrizione**
0-7	posizione degli 8 giunti delle "zampe" dell'agente

Tabella 4.2: *Valori di input e output della rete neurale per il problema AntSwarmBulletEnv*

4.2 Esperimento 1: ottimizzazione della rete

Il primo esperimento è stato rivolto all'ottimizzazione dell'architettura della rete con l'intento di rendere computazionalmente più veloci le future simulazioni cercando di mantenere il più possibile invariata la sua efficacia.
Per questo scopo abbiamo eseguito una simulazione abbassando il numero di neuroni interni della rete da 50 a 20, lasciando immutate tutte le altre proprietà dell'esperimento descritto in [42], riassunte nella Tabella 4.3 e nella Tabella 4.1. Questa operazione è di natura sperimentale poiché non c'è una regola precisa per determinare il numero esatto di neuroni interni che una rete neurale multistrato dovrebbe avere per il raggiungimento ottimale di un determinato obiettivo.

Parametro	Valore
# di milioni di step (msteps) massimo da raggiungere (n_{gen})	50
architettura della rete neurale	feed-forward
# di neuroni nello strato interno	**20**
calcolo della fitness	sommativa
funzione di fitness per l'aggregazione	originale (Formula 3.3)

Tabella 4.3: *Setup della simulazione dell'Esperimento 1*

Il valore della miglior fitness nell'episodio di post-valutazione ottenuto da questa simulazione è di circa 2198.020 (Figura 4.2), superiore alla fitness media di 1769.389 ottenuta in [42]. Dunque abbiamo deciso che valeva la pena abbassare a 20 il numero di neuroni interni della rete per dimezzare il tempo di esecuzione delle future simulazioni, dato che le prestazioni non sembrano degradate, anzi, al contrario, sembrano addirittura migliorate sulla base di questa singola simulazione di prova.

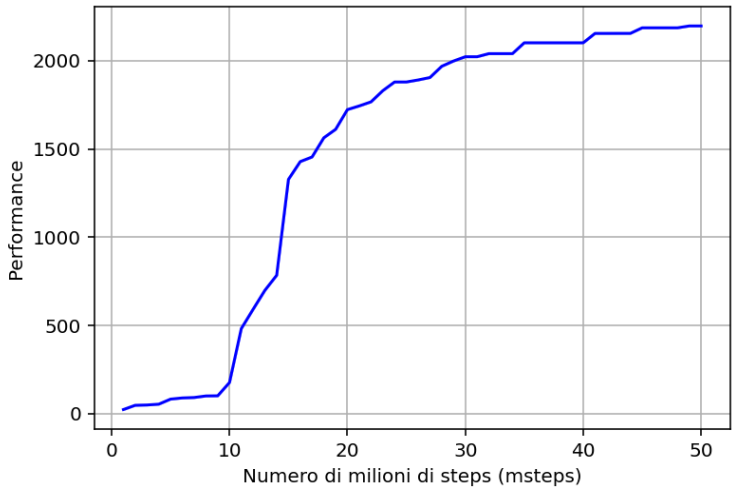


Figura 4.2: *Grafico dei valori della miglior fitness assoluta durante gli msteps della simulazione dell'Esperimento 1*

4.3 Esperimento 2: aggregazione senza locomozione

Per il secondo esperimento ci siamo chiesti se potesse bastare la sola funzione di fitness relativa all'aggregazione per portare il processo evolutivo ad indurre gli agenti al comportamento di locomozione e aggregazione. Inoltre, appurata l'inefficacia della fitness di aggregazione originale (Formula 3.3), spiegata precedentemente nel paragrafo 3.1, l'abbiamo sostituita con la funzione di aggregazione fratta (Formula 3.6).
Nella Tabella 4.4 sono riassunte le proprietà specifiche di questa simulazione.

I valori delle migliori fitness delle 10 simulazioni sono tutti negativi (Figura 4.3) e osservando gli agenti negli episodi di post-valutazione, questi cadono immediatamente oppure, in rari casi, restano fermi in piedi in equilibrio.
Dunque la naturale conclusione di questo esperimento è che la componente relativa alla locomozione (Formula 3.2) è essenziale nel calcolo della fitness sommativa e gli obiettivi di locomozione e aggregazione non si possono ridurre ad una sola delle due componenti per raggiungere entrambi gli obiettivi.

Parametro	**Valore**
# di milioni di step (msteps) massimo da raggiungere (n_{gen})	50
architettura della rete neurale	feed-forward
# di neuroni nello strato interno	20
calcolo della fitness	**sommativa (senza locomozione)**
funzione di fitness per l'aggregazione	**fratta (Formula 3.6)**

Tabella 4.4: *Setup delle simulazioni dell'Esperimento 2*

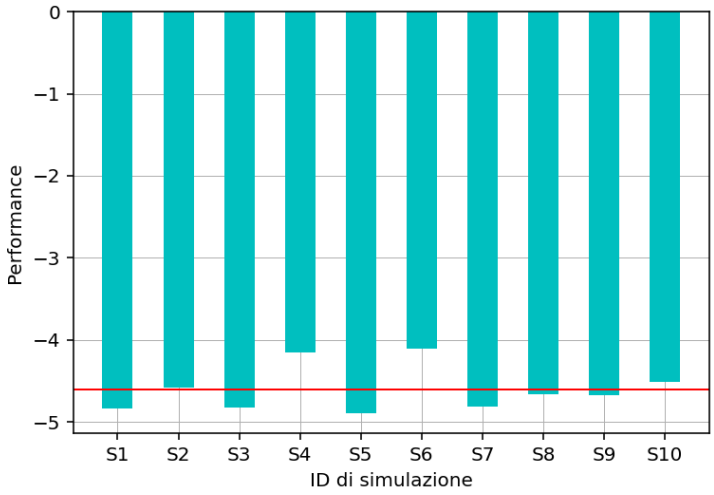


Figura 4.3: *Grafico a barre per confrontare i migliori valori di fitness assoluti negli episodi di post-valutazione dell'Esperimento 2*

4.4 Esperimento 3: aumento del numero di msteps

Dopo l'Esperimento 2, abbiamo deciso di provare ad eseguire delle simulazioni utilizzando la fitness sommativa completa, con la nuova componente D_Fratta (Formula 3.6). Quindi la nuova funzione di fitness risulta come mostrato nella Formula 4.1:

$$\text{Fitness} = \frac{1}{N}\sum_{i=1}^{N} P_i + D\_Fratta_i + S_i + J_i \tag{4.1}$$

Nella Tabella 4.5 sono riassunte le proprietà principali delle simulazioni di questo esperimento. Le simulazioni hanno ottenuto dei buoni risultati quantitativi. Come si può notare dal grafico a barre nella Figura 4.4a il valore medio della fitness (1812.062

Parametro	**Valore**
# di milioni di step (msteps) massimo da raggiungere (n_{gen})	50
architettura della rete neurale	feed-forward
# di neuroni nello strato interno	20
calcolo della fitness	**sommativa**
funzione di fitness per l'aggregazione	fratta (Formula 3.6)

Tabella 4.5: *Setup delle simulazioni dell'Esperimento 3 (con 50msteps)*

con deviazione standard di 321.778) supera, seppur di poco, quello ottenuto nell'articolo [42] (1769.389 con deviazione standard di 419.417), segno che la funzione di aggregazione fratta mediamente ha un'influenza maggiore rispetto alla funzione di aggregazione originale, come avevamo previsto. Tuttavia a livello qualitativo non ci sono stati grossi cambiamenti in quanto nella maggior parte degli episodi di post-valutazione gli agenti camminano senza aggregarsi oppure, raramente, l'agente centrale si avvicina a uno o due agenti laterali. Inoltre, osservando il grafico in Figura 4.5a che descrive l'andamento medio dei migliori valori di fitness negli episodi di post-valutazione durante tutto l'arco delle simulazioni, sembra che la simulazione si blocchi quando è ancora in una fase ascendente, quindi si blocca potenzialmente prima di raggiungere dei migliori risultati e un miglior valore di soglia massimo.

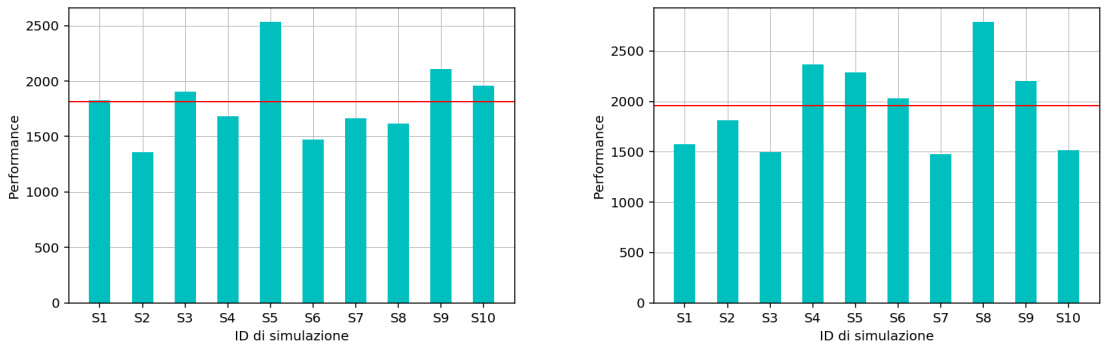
Fatta questa considerazione, abbiamo deciso di riavviare queste stesse simulazioni, raddoppiando il numero di steps da raggiungere prima della fine del processo evolutivo, passando da 50 msteps a 100 msteps (Tabella 4.6).

Parametro	**Valore**
# di milioni di step (msteps) massimo da raggiungere (n_{gen})	**100**
architettura della rete neurale	feed-forward
# di neuroni nello strato interno	20
calcolo della fitness	sommativa
funzione di fitness per l'aggregazione	fratta (Formula 3.6)

Tabella 4.6: *Setup delle simulazioni dell'Esperimento 3 (con 100msteps)*

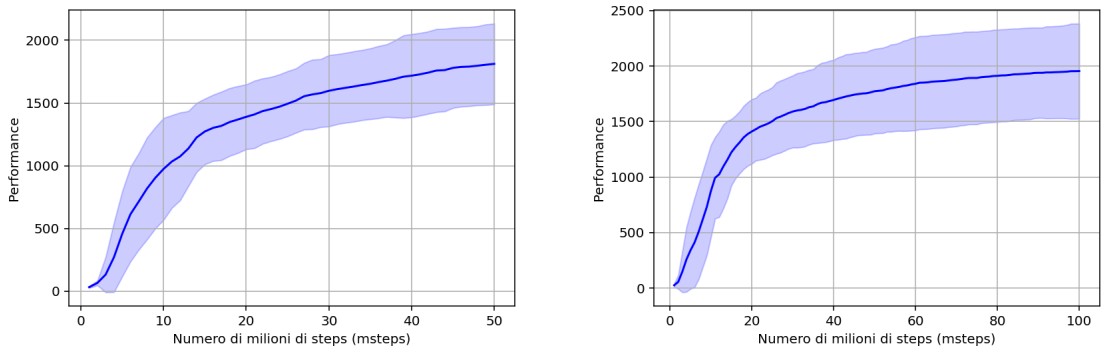
Il risultato è mostrato nel grafico in Figura 4.5b: a differenza dell'andamento mostrato in Figura 4.5a, qui sembra che si sia raggiunto un valore di soglia oltre il quale non c'è molta possibilità di miglioramento. Inoltre le simulazioni hanno raggiunto dei valori di fitness mediamente più alti, con una media di 1955.255 (Figura 4.4b) e una deviazione standard di 428.378. Questa modifica ha portato mediamente allo svilup-

po di una migliore locomozione, ma per quanto riguarda le strategie di aggregazione, queste appaiono ancora poco performanti, come nelle simulazioni iniziali.



(a) **(b)**

Figura 4.4: *Grafici a barre per confrontare i migliori valori di fitness assoluti negli episodi di post-valutazione dell'Esperimento 3, nei casi di 50msteps* ***(a)*** *e 100msteps* ***(b)***



(a) **(b)**

Figura 4.5: *Grafico dei valori medi della miglior fitness assoluta durante gli msteps delle 10 simulazioni in blu con banda di confidenza opaca dell'Esperimento 3, nei casi di 50msteps* ***(a)*** *e 100msteps* ***(b)***

4.5 Esperimento 4: calcolo sequenziale della fitness

Dopo aver raddoppiato il numero di msteps da raggiungere durante il processo evolutivo, abbiamo introdotto il calcolo sequenziale della fitness, già accennato nel paragrafo 3.1, nato dall'intuizione che probabilmente il task di aggregazione necessita della locomozione come sua pre-condizione. Dunque ci è sembrato ragionevole dividere l'intera simulazione in due parti, di cui la prima dedicata esclusivamente allo sviluppo della locomozione e la seconda dedicata esclusivamente allo sviluppo dell'aggregazione.

Questo nuovo metodo ha richiesto l'implementazione di una nuova logica di controllo per salvare i migliori individui delle due parti distinte di simulazione, allo scopo di svolgere delle analisi mirate e visualizzare graficamente il comportamento degli agenti nella prima metà e nella seconda metà di simulazione. Il flusso di operazioni spiegato di seguito è descritto sinteticamente nel diagramma di sequenza (Figura 4.6). Aggiungendo un apposito iperparametro (*seq=1* per indicare il calcolo sequenziale) nel file di configurazione della simulazione (il file *.ini*), abbiamo fatto in modo che fosse reperibile nel codice aggiungendolo alla lista di parametri letti dalla funzione *readConfig* della classe *Policy* (accennata nello schema in Figura 3.2). In questo modo la classe *Policy*, la cui istanza durante l'esecuzione viene passata alla classe *Algo* specifica di OpenAI-ES, ha tra i suoi attributi il valore di *seq*. Una volta avviato l'algoritmo evolutivo attraverso la funzione *run*, il valore di seq viene verificato e se è pari ad 1 verranno salvati (e aggiornati eventualmente) i file "Pre" e "Post" (ad esempio *bestgPreS*.npy*, *bestgPostS*.npy*), confrontando il numero di step attuale e la metà della durata massima della simulazione ($\frac{self.maxmsteps}{2}$).

Parametro	**Valore**
# di milioni di step (msteps) massimo da raggiungere (n_{gen})	100
architettura della rete neurale	feed-forward
# di neuroni nello strato interno	20
calcolo della fitness	**sequenziale**
funzione di fitness per l'aggregazione	fratta (Formula 3.6)

Tabella 4.7: *Setup delle simulazioni dell'Esperimento 4*

Le proprietà di questo esperimento sono riassunte nella Tabella 4.7. Il grafico in Figura 4.7 mostra l'andamento medio dei migliori individui di ogni generazione, evidenziando lo stacco tra le due metà di simulazione con la linea rossa al centro. Il valore medio massimo della fitness nella prima metà di simulazione è di circa 1750, mentre nella seconda metà viene raggiunta subito una soglia massima di circa 170. Il raggiungimento prematuro della soglia nella seconda parte di simulazione è stato un segnale negativo, confermato dalla valutazione qualitativa degli agenti nella loro visualizzazione grafica: infatti, se nella prima metà si è sviluppata mediamente una buona locomozione, nella seconda metà gli agenti hanno sviluppato la strategia di restare fermi immobili sul posto (Figura 4.8), perdendo la capacità di locomozione acquisita nella prima metà e non aggregandosi.

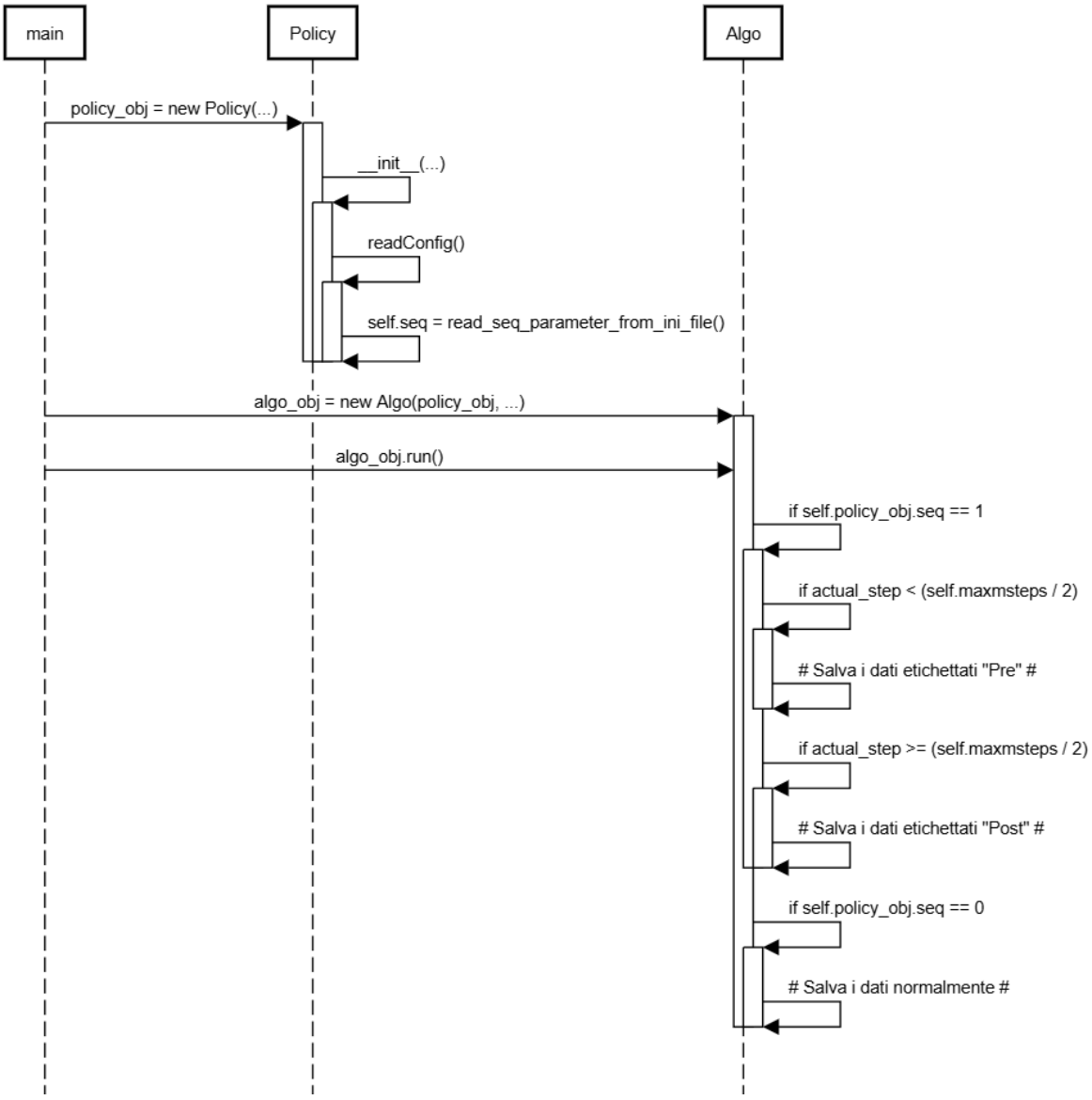


Figura 4.6: *Diagramma di sequenza per descrivere il salvataggio dei file con il calcolo sequenziale della fitness*

4.6 Esperimento 5: rimozione del vincolo della caduta

Successivamente ci siamo chiesti quali impostazioni dei parametri potessero influire, costituendo potenzialmente un limite al raggiungimento dell'obiettivo. Il vincolo della caduta, secondo cui se, durante un episodio, un agente tocca terra con il suo "torso" (la sfera centrale) l'intero episodio termina prematuramente, è stato così messo in discussione e verificato empiricamente per questo specifico task. Abbiamo, quindi,

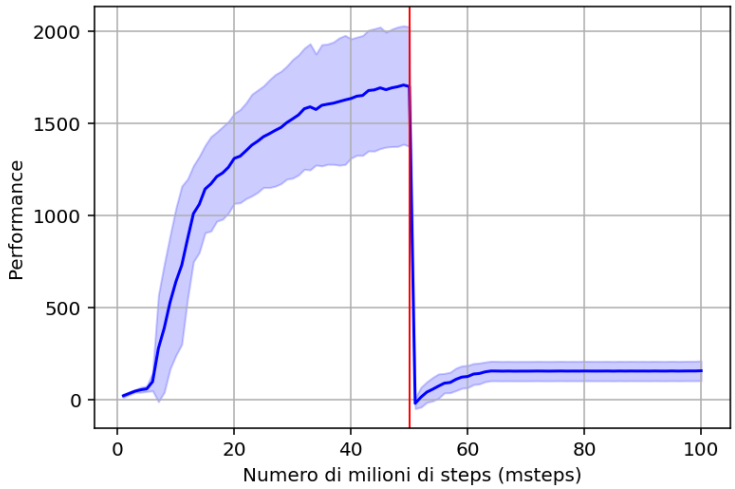


Figura 4.7: *Grafico dei valori medi della miglior fitness in ogni generazione con banda di confidenza opaca dell'Esperimento 4*

modificato temporaneamente il codice del simulatore per avviare delle simulazioni senza il vincolo della caduta e con le proprietà riassunte nella Tabella 4.8.

Parametro	**Valore**
# di milioni di step (msteps) massimo da raggiungere (n_{gen})	100
architettura della rete neurale	feed-forward
# di neuroni nello strato interno	20
calcolo della fitness	sequenziale
funzione di fitness per l'aggregazione	fratta (Formula 3.6)
vincolo della caduta	**no**

Tabella 4.8: *Setup delle simulazioni dell'Esperimento 5*

I risultati sono visibili nel grafico in Figura 4.9, mostrando dei valori massimi di fitness molto scarsi per la maggior parte delle simulazioni, segno del fatto che gli agenti non si sono sviluppati bene, a parte i due casi specifici delle simulazioni S4 e S7. Infatti la media ottenuta è di circa 890.790 con una deviazione standard di 832.808. Quindi abbiamo concluso che il vincolo della caduta è non solo benefico ma praticamente essenziale, in questo task, per lo sviluppo di buone strategie e per i successivi esperimenti lo abbiamo nuovamente introdotto e mantenuto.

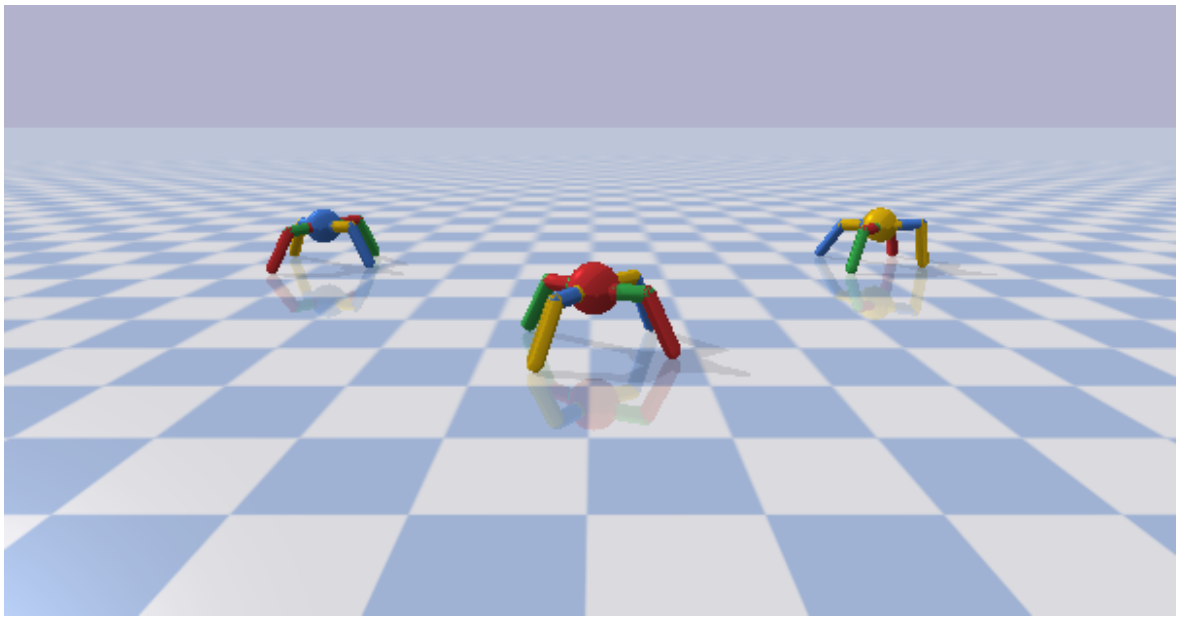


Figura 4.8: *Agenti fermi in equilibrio nei test degli migliori individui della seconda metà di simulazioni dell'Esperimento 4*

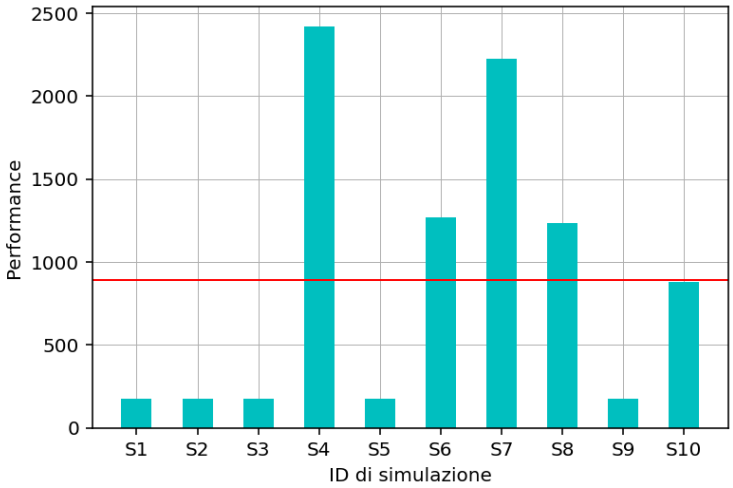


Figura 4.9: *Grafico a barre per confrontare i migliori valori di fitness assoluti negli episodi di post-valutazione dell'Esperimento 5*

4.7 Implementazione di dispositivi di feedback

Dato l'obiettivo di trovare le "condizioni minime" per lo sviluppo di locomozione e aggregazione e per poter analizzare tutte le possibilità, abbiamo deciso di introdurre un meccanismo di feedback esplicito negli agenti, in conformità con i principi della

robotica di sciame accennati nel paragrafo 2.1, e di analizzare, di conseguenza, la sua influenza nel processo di apprendimento. Nello specifico abbiamo introdotto due nuovi tipi di agenti, uno dotato di una camera e uno dotato di camera e dispositivo di prossimità. Nella Figura 4.10 è mostrato uno schema dettagliato delle classi che implementano queste modifiche.

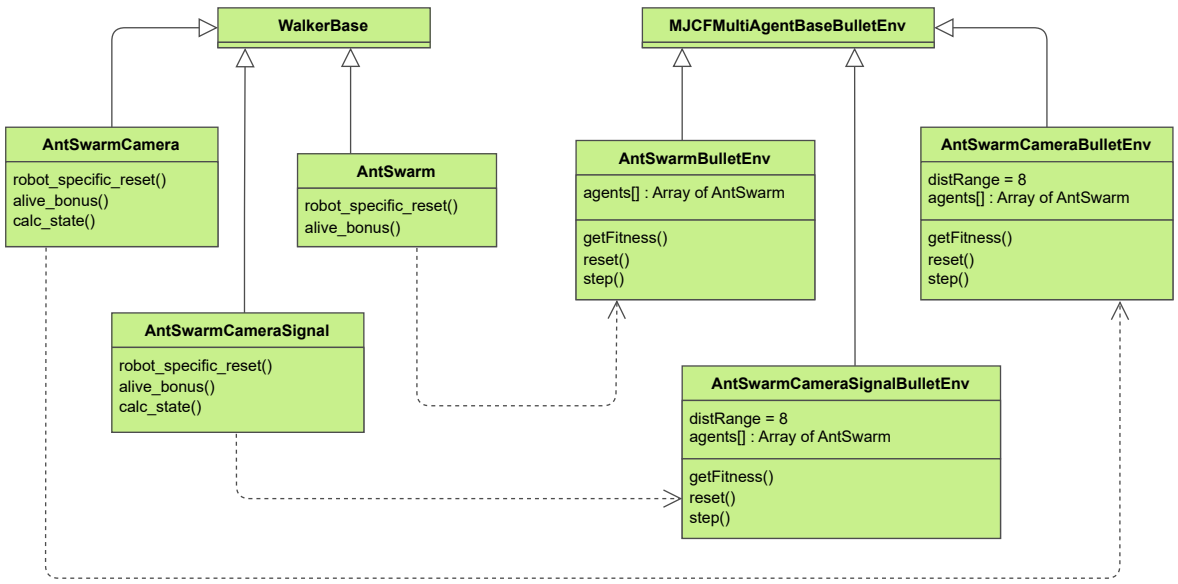


Figura 4.10: *Diagramma delle classi generale dell'architettura degli ambienti e degli agenti*

Innanzitutto la classe *MJCFMultiAgentBaseBulletEnv* definisce le proprietà di base per la creazione di ambienti multi-agente, mentre la classe *WalkerBase* definisce le proprietà di base per l'implementazione di agenti locomotori. Ogni ambiente utilizza degli agenti specifici. Ad esempio, nell'articolo [42], dal quale sono partiti i nostri esperimenti, e fino all'Esperimento 5 illustrato nel paragrafo 4.6 l'ambiente utilizzato è *AntSwarmBulletEnv*, che definisce le proprietà di un ambiente composto dagli agenti specificati nella classe *AntSwarm*. Il numero e il tipo di input/output di questi agenti è stato descritto precedentemente nella Tabella 4.2.
Per introdurre le modifiche desiderate, quindi, abbiamo creato delle nuove classi di agenti e delle nuove classi specifiche per gli ambienti che devono gestirli, ovvero l'ambiente *AntSwarmCameraBulletEnv* che gestisce gli agenti del tipo *AntSwarmCamera* e l'ambiente *AntSwarmCameraSignalBulletEnv* che gestisce gli agenti del tipo *AntSwarmCameraSignal*. Le classi degli agenti, come si può notare dallo schema in Figura 4.10, non estendono la classe *AntSwarm* perché questi tre tipi di agenti hanno dei vettori di azione e osservazione differenti. Nella Tabella 4.9 sono indicati gli input e gli output, aggiunti a quelli già indicati nella Tabella 4.2.

Id input	Descrizione
28-33	il valore della distanza da un agente catturata dai 6 settori in cui è divisa la camera 360° di cui è dotato l'agente
34	contatore del numero di fonti percepite entro la distanza prefissata
Id output	**Descrizione**
8	valore binario per l'emissione del segnale

Tabella 4.9: *Valori di input e output della rete neurale aggiunti dai problemi AntSwarmCameraBulletEnv e AntSwarmCameraSignalBulletEnv*

In entrambi i nuovi ambienti sono stati effettuati degli adattamenti per controllare e aggiornare lo stato degli input/output relativi alla camera e al segnale. Nello specifico, nella funzione *step* viene controllata la distanza tra gli agenti e aggiornato lo stato dei valori dedicati alla camera e al segnale, sulla base del valore di *distRange* (fissato a 8 metri in questo caso), mentre nella funzione *reset* vengono portati questi valori a 0, per l'inizio degli episodi. Scendendo più nel dettaglio, nella Figura 4.11 è mostrato un esempio di implementazione della gestione della camera nella funzione *step*. Per ogni coppia di agenti viene verificato se sono ad una distanza tale da "percepire" la presenza altrui (riga 1227) e, nel caso, viene calcolato l'angolo della posizione di un agente relativamente all'altro (righe 1230-1232). Fatto ciò viene divisa la camera di 360° in 6 settori e, in base all'angolo calcolato in precedenza, viene aggiornato il valore corrispettivo del settore interessato con la distanza dell'agente appena calcolata (righe 1239-1249).

Queste nuove implementazioni ci hanno concesso di effettuare nuovi esperimenti e di affinare ulteriormente l'esplorazione delle soluzioni per la risoluzione di questo task; inoltre, hanno arricchito la libreria di ambienti utilizzabili nel simulatore Evorobotpy3.

4.8 Esperimento 6: calcolo incrementale della fitness

Un'idea interessante che abbiamo voluto testare è stata l'implementazione del calcolo incrementale della fitness, accennato anch'esso nel paragrafo 3.1. Molto similmente al calcolo sequenziale, implementato e illustrato nel paragrafo 4.5, il calcolo incrementale consiste nel dividere la simulazione in due parti, la prima delle quali dedicata esclusivamente alla locomozione e la seconda dedicata a locomozione e aggregazione sommate. Anche per il salvataggio dei file le due metodologie si comportano allo stesso modo, salvando le informazioni in maniera distinta per la prima e per la seconda metà di simulazione (Figura 4.6). La differenza di questo approccio è che nella seconda parte di simulazione la locomozione, essendo già sviluppata, può essere regolata arbitrariamen-

gym_locomotion_envs2.py > AntSwarmCameraBulletEnv > step

```
1032  class AntSwarmCameraBulletEnv(MJCFMultiAgentBaseBulletEnv):
1196    def step(self, a):
1220      # Now examine the distances between agents so as to update the camera activations
1221      for i in range(self.nrobots):
1222          for j in range(self.nrobots):
1223              if i != j:
1224                  dist = np.linalg.norm([
1225                    self.robots[j].body_xyz[1] - self.robots[i].body_xyz[1],
1226                    self.robots[j].body_xyz[0] - self.robots[i].body_xyz[0]])
1227                  if dist <= self.distRange:
1228                      # The robot can detect is
1229                      # Compute relative angle
1230                      relAng = math.atan2(
1231                        self.robots[j].body_xyz[1] - self.robots[i].body_xyz[1],
1232                        self.robots[j].body_xyz[0] - self.robots[i].body_xyz[0]) - angle[i]
1233                      # Set relative angle in the range [0°,360°]
1234                      if relAng < 0.0:
1235                          relAng += 2.0 * math.pi
1236                      if relAng > 2.0 * math.pi:
1237                          relAng -= 2.0 * math.pi
1238                      # Now compute the corresponding sector
1239                      idx = None
1240                      sector = 0
1241                      cameraSize = 6
1242                      found = False
1243                      cang = 0.0
1244                      while cang <= 360.0 and not found:
1245                          if cang > relAng:
1246                              # Found sector
1247                              idx = ((i + 1) * rob_state_len) - cameraSize + sector
1248                              found = True
1249                          cang += 360.0 / cameraSize
1250                      if not found:
1251                          print("FATAL ERROR!!! Distance below threshold but sector not identified!!!")
1252                          sys.exit()
1253                      # Set corresponding input as the inverse of the ratio between
1254                      # distance and max distance (dist = maxDist -> inp = 0; dist = 0 -> inp = 1)
1255                      state[idx] = (1.0 - np.float32(dist / self.distRange))
1256    #state = np.clip(state, -5, +5) # Maybe useless
```

Figura 4.11: *Gestione degli input relativi alla fotocamera nella funzione step della classe AntSwarmCameraBulletEnv*

te attraverso l'aggiunta di un peso per evitare che gli agenti rimangano fermi, come è accaduto nell'esperimento del calcolo sequenziale della fitness. Un esempio di formula di calcolo incrementale è mostrata nella Formula 4.2.

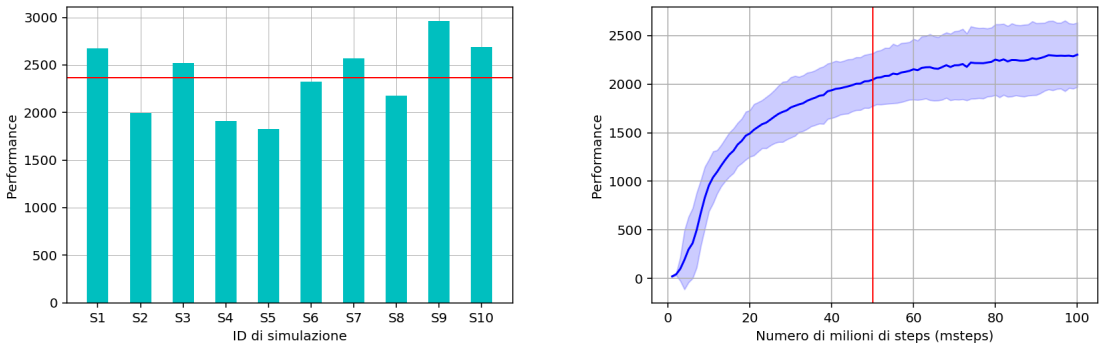
$$\begin{aligned} &\text{I metà:} \quad \text{Fitness} = \frac{1}{N}\sum_{i=1}^{N} P_i + S_i + J_i \\ &\text{II metà:} \quad \text{Fitness} = \frac{1}{N}\sum_{i=1}^{N} w \cdot P_i + D\_Fratta_i + S_i + J_i \end{aligned} \tag{4.2}$$

Nella Tabella 4.10 sono riassunte le proprietà di questo esperimento. Oltre ad essere il primo esperimento con il calcolo incrementale della fitness, è stato anche il primo test dell'ambiente con gli agenti dotati di camera. Il grafico a barre nella Figura 4.12a evidenzia il raggiungimento di ottimi risultati qualitativi, con un valore medio di

Parametro	**Valore**
# di milioni di step (msteps) massimo da raggiungere (n_{gen})	100
architettura della rete neurale	feed-forward
# di neuroni nello strato interno	20
calcolo della fitness	**incrementale** ($w = 1$)
funzione di fitness per l'aggregazione	fratta (Formula 3.6)
environment	**AntSwarmCamera**

Tabella 4.10: *Setup delle simulazioni dell'Esperimento 6*

fitness di 2364.407 e una deviazione standard di 359.023. Il grafico nella Figura 4.12b mostra come in questo caso, a differenza del calcolo sequenziale (Figura 4.7) ci sia una certa continuità nei valori della fitness. Infine, osservando graficamente i risultati, la locomozione è ottima ma l'aggregazione risulta parziale e incompleta. Tuttavia, a differenza degli esperimenti precedenti, gli agenti si muovono in maniera più dinamica, evidenziando un minimo criterio di posizionamento basandosi sui valori giunti dagli input della camera. Sintomo, questo, che la camera può fornire informazioni spaziali utili agli agenti per prendere "consapevolezza" della posizione relativa degli altri.



(a) **(b)**

Figura 4.12: *Grafico a barre per confrontare i migliori valori di fitness assoluti negli episodi di post-valutazione* ***(a)*** *e grafico dei valori medi della miglior fitness in ogni generazione con banda di confidenza opaca* ***(b)*** *dell'Esperimento 6*

4.9 Esperimento 7: architettura di rete RNN

Fino a questo momento abbiamo visto il task di locomozione e aggregazione come un tipo di task in cui in ogni istante ogni agente ha bisogno di percepire gli agenti intorno a se e reagire di conseguenza, senza la necessità di una memoria sulle osservazioni passate. In questo esperimento abbiamo indagato la possibilità di utilizzare una RNN, per introdurre una sorta di capacità di memoria grazie ai collegamenti ricorsivi della rete, cercando di capire se e quanto fosse positivo o conveniente intraprendere questa strada in un task del genere.

Scenario	**Risultati**
1. AntSwarmCamera, calcolo sequenziale della fitness	1778.028 [632.807]
2. AntSwarmCamera, calcolo incrementale della fitness	2472.904 [282.629]
3. AntSwarmCameraSignal, calcolo sequenziale della fitness	1409.268 [818.128]
4. AntSwarmCameraSignal, calcolo incrementale della fitness	2211.910 [391.244]
5. AntSwarmCamera, calcolo incrementale della fitness, con locomozione pesata ($w = 0.1$)	1583.512 [472.473]
6. AntSwarmCameraSignal, calcolo incrementale della fitness, con locomozione pesata ($w = 0.1$)	1590.637 [669.738]

Tabella 4.11: *Sintesi degli scenari e dei risultati dell'Esperimento 7*

Per avere un quadro ampio di questa nuova possibilità, l'abbiamo testata in diversi scenari, ognuno dei quali indicati nella Tabella 4.11 con i relativi risultati calcolati su 10 simulazioni per ogni scenario e illustrati in tabella secondo lo schema "miglior fitness media [deviazione standard]".
Nelle simulazioni con il calcolo sequenziale gli agenti, nella seconda metà, sviluppano la strategia di rimanere fermi sul posto annullando la locomozione, così come mostrato nell'Esperimento 4. Nelle simulazioni con il calcolo incrementale i risultati, quantitativi e qualitativi, sono generalmente migliori, anche se nella 2 e nella 4 i risultati sono simili alle simulazioni dell'Esperimento 6 mentre nella 5 e nella 6 si è verificato qualcosa di interessante: il peso $w = 0.1$ applicato alla locomozione nella seconda metà di simulazione ha fatto sì che gli agenti mantenessero la capacità di locomozione (superando il problema delle simulazioni con il calcolo sequenziale della fitness) e la regolassero. Infatti la strategia osservata graficamente mostra che gli agenti, nella seconda metà, hanno sviluppato una locomozione più moderata ed "elegante"[1] rispetto alla locomozione estrema e "galoppante"[2] favorita dalla componente P pura (Formula 3.2) nelle precedenti simulazioni. Questo ci ha dato degli spunti interessanti per affrontare il prossimo esperimento. Per quanto concerne il comportamento di aggregazione, questo

[1]Esempio grafico della camminata "elegante": `https://youtu.be/gW_LdjBOIbs`
[2]Esempio grafico della camminata "galoppante": `https://youtu.be/MRquB4HhEFo`

è emerso tanto quanto le simulazioni precedenti con l'unica differenza che durante gli episodi è capitato diverse volte che uno o più agenti si bloccassero, rimanendo fermi in equilibrio.
In generale, il passaggio da rete feed-forward a RNN ha peggiorato di poco i risultati, mostrandosi come un'alternativa interessante all'utilizzo di una rete feed-forward. Tuttavia la maggiore complessità computazionale di una RNN, con il conseguente aumento della durata delle simulazioni, e l'obiettivo iniziale di ricerca sulle "condizioni minime" per lo sviluppo di locomozione e aggregazione hanno fatto sì che scartassimo per i prossimi esperimenti questo tipo di configurazione, lasciandola come una possibile alternativa da indagare eventualmente in futuro per scopi di ricerca simili.

4.10 Esperimento 8: fitness di aggregazione gaussiana

Il nostro ultimo esperimento si è concentrato sulla funzione di fitness relativa all'aggregazione. Visti i risultati, abbiamo ipotizzato che probabilmente la fitness di aggregazione fratta ha una crescita troppo lenta e improvvisa (Figura 3.1) e lo sviluppo dell'aggregazione si blocca in un massimo locale, non riuscendo a trovare delle strategie per ottenere ricompense più alte e venendo dominata dalla componente di locomozione. Per questo motivo abbiamo pensato di introdurre la fitness di aggregazione gaussiana (Formula 3.7), che presenta una crescita più graduale e quindi potrebbe permettere alle perturbazioni generate intorno al vettore θ di individuare un gradiente più intenso e di definire meglio la direzione di sviluppo di una strategia ottimale. Inoltre, abbiamo introdotto questa modifica in concomitanza con l'aggiunta di un peso $w = 0.1$ alla componente di locomozione nel calcolo incrementale della fitness. Le proprietà di questo ultimo esperimento sono mostrate nella Tabella 4.12.

Parametro	**Valore**
# di milioni di step (msteps) massimo da raggiungere (n_{gen})	100
architettura della rete neurale	feed-forward
# di neuroni nello strato interno	20
calcolo della fitness	**incrementale** ($w = 0.1$)
funzione di fitness per l'aggregazione	**gaussiana (Formula 3.7)**
environment	**AntSwarmCamera**

Tabella 4.12: *Setup delle simulazioni dell'Esperimento 8*

I grafici nelle Figure 4.14 mostrano i risultati di questo esperimento. Dato il passaggio dalla fitness di aggregazione fratta a quella gaussiana, questi dati non sono confrontabili con i dati delle precedenti simulazioni. La miglior fitness media delle simulazioni di questo esperimento è di 4444.679 con una deviazione standard di 230.541. A livello qualitativo, gli agenti tendono a mantenere una velocità di locomozione moderata e tendono a sparpagliarsi, arrivando ad aggregarsi massimo a 3 (Figura 4.13), quindi non completamente.

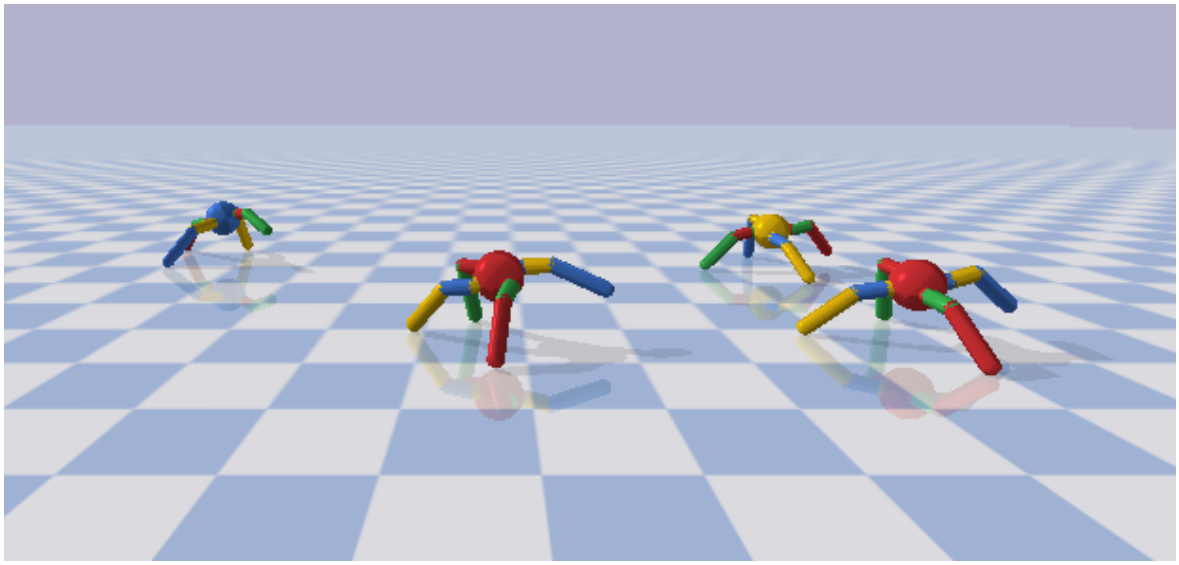
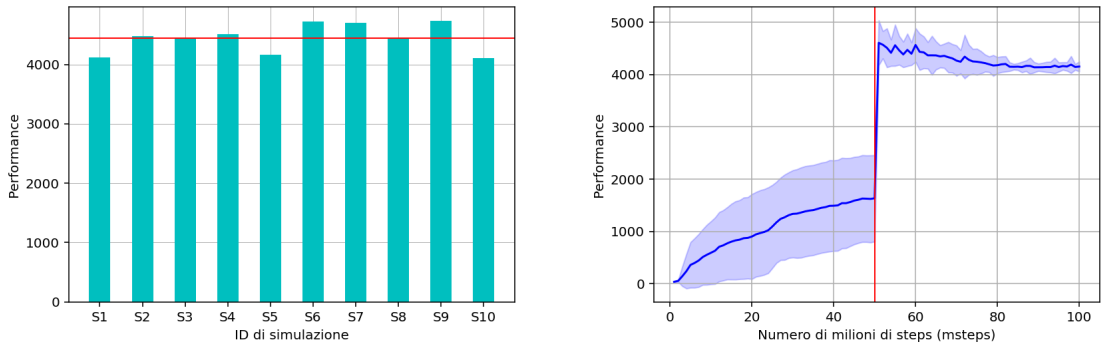


Figura 4.13: *Agenti aggregati nei test dei migliori individui della seconda metà di simulazioni dell'Esperimento 8*



(a) **(b)**

Figura 4.14: *Grafico a barre per confrontare i migliori valori di fitness assoluti negli episodi di post-valutazione* ***(a)*** *e grafico dei valori medi della miglior fitness in ogni generazione con banda di confidenza opaca* ***(b)*** *dell'Esperimento 8*

Capitolo 5

Conclusioni

In questo lavoro di tesi siamo partiti dall'indagine del problema di ottimizzazione multi-obiettivo riguardante lo sviluppo della locomozione e dell'aggregazione in un sistema multi-agente simulato attraverso l'uso dell'algoritmo OpenAI-ES. Nel corso degli esperimenti abbiamo affinato le proprietà del problema, ottenendo diversi buoni risultati. Abbiamo ottimizzato la struttura interna della rete neurale, passando da 50 a 20 neuroni interni, velocizzando, così, le simulazioni senza perdita di efficacia della rete. Abbiamo elaborato nuove funzioni di fitness per l'aggregazione, risolvendo le criticità presenti nella formula di fitness iniziale, e abbiamo introdotto le metodologie del calcolo sequenziale e incrementale della fitness per tentare degli approcci più dinamici allo sviluppo delle strategie evolutive. Infine, abbiamo implementato dei nuovi ambienti con agenti dotati di sistemi di feedback, nello specifico una camera direzionale e un dispositivo di prossimità, per introdurre un senso di spazialità nel calcolo interno della rete di ogni agente. In generale abbiamo ottenuto dei miglioramenti rispetto ai risultati dai quali siamo partiti, registrando dei risultati quantitativi migliori e dei comportamenti emergenti che mostrano un'aggregazione parziale tra gli agenti negli episodi di valutazione. La funzione di fitness gaussiana si è dimostrata promettente, così come l'aggiunta di pesi alle componenti della fitness complessiva. Per sviluppi futuri di questo problema si potrebbero integrare gli approcci filogenetico e ontogenetico, unendo quindi le potenzialità di OpenAI-ES per lo sviluppo della locomozione e l'evoluzione mirata dell'apprendimento per rinforzo per lo sviluppo dell'aggregazione. Inoltre si potrebbero utilizzare reti neurali eterogenee per determinare il comportamento dei diversi agenti, così da esplorare la loro capacità di adattamento alle abilità degli altri [41].

Ringraziamenti

Ringrazio la mia famiglia, i miei fratelli e i miei amici più cari per l'amore e l'affetto incondizionato e per il supporto costante. Grazie.

Bibliografia

[1] Moshe Abeles, Hagai Bergman, Eyal Margalit, and Eilon Vaadia. Spatiotemporal firing patterns in the frontal cortex of behaving monkeys. *Journal of neurophysiology*, 70:1629–38, 11 1993. doi: 10.1152/jn.1993.70.4.1629.

[2] Abien Fred Agarap. Deep learning using rectified linear units (relu), 2019. URL `https://arxiv.org/abs/1803.08375`.

[3] Mohiuddin Ahmed, Raihan Seraj, and Syed Mohammed Shamsul Islam. The k-means algorithm: A comprehensive survey and performance evaluation. *Electronics*, 9(8), 2020. ISSN 2079-9292. doi: 10.3390/electronics9081295. URL `https://www.mdpi.com/2079-9292/9/8/1295`.

[4] Oladayo S. Ajani, Sung-ho Hur, and Rammohan Mallipeddi. Evaluating domain randomization in deep reinforcement learning locomotion tasks. *Mathematics*, 11 (23), 2023. ISSN 2227-7390. doi: 10.3390/math11234744. URL `https://www.mdpi.com/2227-7390/11/23/4744`.

[5] Parasumanna Gokulan Balaji and Dipti Srinivasan. An introduction to multi-agent systems. *Innovations in multi-agent systems and applications-1*, pages 1–27, 2010.

[6] Manuele Brambilla, Eliseo Ferrante, Mauro Birattari, and Marco Dorigo. Swarm robotics: a review from the swarm engineering perspective. *Swarm Intelligence*, 7:1–41, 2013.

[7] Yair Censor. Pareto optimality in multiobjective problems. *Applied Mathematics and Optimization*, 4(1):41–59, 1977.

[8] Erwin Coumans and Yunfei Bai. Pybullet, a python module for physics simulation for games, robotics and machine learning, 2016.

[9] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan. A fast and elitist multiobjective genetic algorithm: Nsga-ii. *Trans. Evol. Comp*, 6(2):182–197, April 2002. ISSN 1089-778X. doi: 10.1109/4235.996017. URL `https://doi.org/10.1109/4235.996017`.

[10] Kalyanmoy Deb, Karthik Sindhya, and Jussi Hakanen. Multi-objective optimization. In *Decision sciences*, pages 161–200. CRC Press, 2016.

[11] AD Dongare, RR Kharde, Amit D Kachare, et al. Introduction to artificial neural network. *International Journal of Engineering and Innovative Technology (IJEIT)*, 2(1):189–194, 2012.

[12] M. Dorigo and M. Birattari. Swarm intelligence. *Scholarpedia*, 2(9):1462, 2007. doi: 10.4249/scholarpedia.1462. revision #138640.

[13] M. Dorigo, M. Birattari, and M. Brambilla. Swarm robotics. *Scholarpedia*, 9(1): 1463, 2014. doi: 10.4249/scholarpedia.1463. revision #138643.

[14] Michael T. M. Emmerich and André H. Deutz. A tutorial on multiobjective optimization: fundamentals and evolutionary methods. *Natural Computing*, 17 (3):585–609, Sep 2018. ISSN 1572-9796. doi: 10.1007/s11047-018-9685-y. URL `https://doi.org/10.1007/s11047-018-9685-y`.

[15] Dario Floreano, Phil Husbands, and Stefano Nolfi. Evolutionary robotics. *Handbook of robotics*, 2008.

[16] L.J. Fogel, A.J. Owens, and M.J. Walsh. *Artificial Intelligence Through Simulated Evolution*. Wiley, 1966.

[17] Gianluigi Folino. Algoritmi evolutivi e programmazione genetica: strategie di progettazione e parallelizzazione. *Consiglio Nazionale delle Ricerche, Istituto di Calcolo e Reti ad Alte Prestazioni*, 55, 2003.

[18] Bolin Gao and Lacra Pavel. On the properties of the softmax function with application in game theory and reinforcement learning, 2018. URL `https://arxiv.org/abs/1704.00805`.

[19] Avinash Gautam and Sudeept Mohan. A review of research in multi-robot systems. In *2012 IEEE 7th International Conference on Industrial and Information Systems (ICIIS)*, pages 1–5, 2012. doi: 10.1109/ICIInfS.2012.6304778.

[20] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pages 249–256. JMLR Workshop and Conference Proceedings, 2010.

[21] David E Goldberg and Jon Richardson. Genetic algorithms with sharing for multimodal function optimization. In *Genetic algorithms and their applications: Proceedings of the Second International Conference on Genetic Algorithms*, volume 4149. Hillsdale, NJ: Lawrence Erlbaum, 1987.

[22] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning*. MIT press, 2016.

[23] Nikolaus Hansen and Andreas Ostermeier. Completely derandomized self-adaptation in evolution strategies. *Evolutionary computation*, 9(2):159–195, 2001.

[24] Markus Hegland. *THE APRIORI ALGORITHM – A TUTORIAL*, pages 209–262. doi: 10.1142/9789812709066_0006. URL `https://www.worldscientific.com/doi/abs/10.1142/9789812709066_0006`.

[25] S Hochreiter and J Schmidhuber. Long short-term memory. *Neural Computation MIT-Press*, 1997.

[26] John H Holland. Adaptation in natural and artificial systems, univ. of mich. press. *Ann Arbor*, 1975.

[27] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015.

[28] Eugene M Izhikevich. Simple model of spiking neurons. *IEEE Transactions on neural networks*, 14(6):1569–1572, 2003.

[29] L. C. Jain and L. R. Medsker. *Recurrent Neural Networks: Design and Applications*. CRC Press, Inc., USA, 1st edition, 1999. ISBN 0849371813.

[30] James Kennedy. *Swarm Intelligence*, pages 187–219. Springer US, Boston, MA, 2006. ISBN 978-0-387-27705-9. doi: 10.1007/0-387-27705-6_6. URL `https://doi.org/10.1007/0-387-27705-6_6`.

[31] Mifa Kim, Tomoyuki Hiroyasu, Mitsunori Miki, and Shinya Watanabe. Spea2+: Improving the performance of the strength pareto evolutionary algorithm 2. In Xin Yao, Edmund K. Burke, José A. Lozano, Jim Smith, Juan Julián Merelo-Guervós, John A. Bullinaria, Jonathan E. Rowe, Peter Tiňo, Ata Kabán, and Hans-Paul Schwefel, editors, *Parallel Problem Solving from Nature - PPSN VIII*, pages 742–751, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg. ISBN 978-3-540-30217-9.

[32] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv:1412.6980*, 2014.

[33] JRGP Koza. On the programming of computers by means of natural selection. *Genetic programming*, 1992.

[34] Petr Lansky and Susanne Ditlevsen. A review of the methods for signal estimation in stochastic diffusion leaky integrate-and-fire neuronal models. *Biological cybernetics*, 99(4):253–262, 2008.

[35] Antonio López Jaimes, Saúl Zapotecas-Martínez, and Carlos Coello. *An Introduction to Multiobjective Optimization Techniques*, pages 29–57. 01 2011. ISBN 978-1-61122-818-2.

[36] Wolfgang Maass. Networks of spiking neurons: the third generation of neural network models. *Neural networks*, 10(9):1659–1671, 1997.

[37] Warren Mcculloch and Walter Pitts. A logical calculus of ideas immanent in nervous activity. *Bulletin of Mathematical Biophysics*, 5:127–147, 1943.

[38] Andrew L Nelson, Gregory J Barlow, and Lefteris Doitsidis. Fitness functions in evolutionary robotics: A survey and analysis. *Robotics and Autonomous Systems*, 57(4):345–370, 2009.

[39] Mark Nelson and John Rinzel. The hodgkin-huxley model. *The book of genesis*, 2, 1995.

[40] Stefano Nolfi. *Behavioral and Cognitive Robotics: An Adaptive Perspective*. 01 2021.

[41] Paolo Pagliuca and Alessandra Vitanza. N-mates evaluation: a new method to improve the performance of genetic algorithms in heterogeneous multi-agent systems. In *Proceedings of the 24th Edition of the Workshop From Object to Agents (WOA23)*, volume 3579, pages 123–137, 2023.

[42] Paolo Pagliuca and Alessandra Vitanza. Enhancing aggregation in locomotor multi-agent systems: a theoretical framework. 07 2024.

[43] Paolo Pagliuca and Alessandra Vitanza. A comparative study of evolutionary strategies for aggregation tasks in robot swarms: Macro- and micro-level behavioral analysis. *IEEE Access*, 2025.

[44] Paolo Pagliuca, Nicola Milano, and Stefano Nolfi. Efficacy of modern neuro-evolutionary strategies for continuous control optimization. *Frontiers in Robotics and AI*, 7:98, 2020.

[45] Paolo Pagliuca, Stefano Nolfi, and Alessandra Vitanza. Evorobotpy3: a flexible and easy-to-use simulation tool for evolutionary robotics. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion (GECCO 2025 Companion)*, 2025.

[46] Ingo Rechenberg. Evolutionsstrategie: Optimierung technischer systeme nach prinzipien der biologischen evolution, frommann–holzboog. *Stuttgart-Bad Cannstatt*, page 47, 1973.

[47] Markus Ringnér. What is principal component analysis? *Nature Biotechnology*, 26(3):303–304, Mar 2008. ISSN 1546-1696. doi: 10.1038/nbt0308-303. URL `https://doi.org/10.1038/nbt0308-303`.

[48] Frank Rosenblatt. Principles of neurodynamics: perceptrons and the theory of brain mechanisms. cornell aeronautical laboratory. *Inc., Cornell University: Buffalo, NY, USA*, 1962.

[49] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *nature*, 323(6088):533–536, 1986.

[50] Erol Şahin. Swarm robotics: From sources of inspiration to domains of application. In *International workshop on swarm robotics*, pages 10–20. Springer, 2004.

[51] Tim Salimans, Jonathan Ho, Xi Chen, and Ilya Sutskever. Evolution strategies as a scalable alternative to reinforcement learning. 03 2017. doi: 10.48550/arXiv.1703.03864.

[52] Melanie Schranz, Martina Umlauft, Micha Sende, and Wilfried Elmenreich. Swarm robotic behaviors and current applications. *Frontiers in Robotics and AI*, 7:36, 2020.

[53] Hans-Paul Schwefel. *Numerische Optimierung von Computer-Modellen mittels der Evolutionsstrategie: mit einer vergleichenden Einführung in die Hill-Climbing-und Zufallsstrategie*, volume 1. Springer, 1977.

[54] Richard S Sutton and Andrew Barto. Reinforcement learning: An introduction. *Cambridge MIT Press*, 1998.

[55] Daniel Svozil, Vladimír Kvasnicka, and Jiří Pospichal. Introduction to multi-layer feed-forward neural networks. *Chemometrics and Intelligent Laboratory Systems*, 39(1):43–62, 1997. ISSN 0169-7439. doi: https://doi.org/10.1016/S0169-7439(97)00061-0. URL `https://www.sciencedirect.com/science/article/pii/S0169743997000610`.

[56] Andrea Tettamanzi. Algoritmi evolutivi: Concetti e applicazioni. 2005, 03 2005.

[57] Emanuel Todorov, Tom Erez, and Yuval Tassa. Mujoco: A physics engine for model-based control. In *2012 IEEE/RSJ international conference on intelligent robots and systems*, pages 5026–5033. IEEE, 2012.

[58] Mark Towers, Ariel Kwiatkowski, Jordan Terry, John U Balis, Gianluca De Cola, Tristan Deleu, Manuel Goulao, Andreas Kallinteris, Markus Krimmel, Arjun KG, et al. Gymnasium: A standard interface for reinforcement learning environments. *arXiv preprint arXiv:2407.17032*, 2024.

[59] Daan Wierstra, Tom Schaul, Tobias Glasmachers, Yi Sun, Jan Peters, and Jürgen Schmidhuber. Natural evolution strategies. *The Journal of Machine Learning Research*, 15(1):949–980, 2014.

[60] Michael Wooldridge. *An Introduction to MultiAgent Systems*. Wiley Publishing, 2nd edition, 2009. ISBN 0470519460.

[61] Kashu Yamazaki, Viet-Khoa Vo-Ho, Darshan Bulsara, and Ngan Le. Spiking neural networks and their applications: A review. *Brain Sciences*, 12(7), 2022. ISSN 2076-3425. doi: 10.3390/brainsci12070863. URL `https://www.mdpi.com/2076-3425/12/7/863`.